CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS (Année 2003 / 2004)

MÉMOIRE D'EXAMEN PROBATOIRE

Spécialité : INFORMATIQUE, RÉSEAUX, SYSTEMES, MULTIMEDIA

SUJET N°24 :

ADA 95 et la répartition

Recommandation **: traiter Ada 95 et les normes CORBA**

Auditeur : Olivier ALZEARI (I039763)

Mémoire déposé le 5 janvier 2004

Président du Jury : Professeur KAISER

TABLES DES MATIÈRES

INTRODUCTION

Le développement des réseaux de télécommunications ont permis le déploiement des connexions entre ordinateurs, depuis le réseau local d'entreprise jusqu'au réseau mondial qu'est l'Internet.

Parallèlement, l'évolution des matériels et des logiciels a entraîné le passage de l'informatique centralisée à l'informatique distribuée : les applications « monolithiques » ont fait place à des applications « réparties ».

La répartition a contribué à améliorer les performances des applications (mise en commun de puissance de calcul et de ressources) mais impose de nouvelles contraintes liées à l'utilisation des réseaux et aux modes de communication entre composants applicatifs distants.

Dans un système réparti, les nombreuses architectures matérielles et logicielles existantes doivent pouvoir échanger des informations pour effectuer les traitements attendus. Le bon déroulement d'une application répartie dépend de la qualité des communications et des interactions entre composants évoluant dans des environnements hétérogènes.

Il existe différentes solutions pour prendre en charge les problèmes liés à la répartition. Le but de cette étude est de présenter deux approches différentes de cette problématique :

- ✓ l'utilisation d'un langage distribué
- ✓ la mise en œuvre d'un intergiciel (ou Middleware)

La première phase de l'étude consiste à exposer les principes de fonctionnement des systèmes répartis et des différents modèles de répartition. Ceux-ci peuvent être mis en œuvre par un langage distribué ou par un intergiciel de répartition.

Ensuite, nous étudierons en détail les deux approches envisagées à travers un exemple concret : Ada 95 et son annexe des systèmes répartis en tant que langage distribué, et CORBA en tant qu'intergiciel de répartition.

Nous serons alors en mesure d'effectuer une approche comparée entre Ada 95 (DSA) et CORBA, et de comprendre la portée de l'adaptation de certains services CORBA à l'environnement Ada 95. Ces évolutions successives nous amènerons à clôturer notre étude par la présentation d'un « intergiciel schizophrène », PolyORB.

I. SYSTÈMES RÉPARTIS ET MODÈLES DE RÉPARTITION

A. Principes de la répartition

1. LES SYSTÈMES RÉPARTIS

Un système réparti rassemble des systèmes physiquement séparés, éventuellement hétérogènes, en un seul système cohérent qui fournit à l'utilisateur un accès aux diverses ressources maintenues par le système.

1.1. Définition d'un système réparti

Un système réparti (ou distribué) est constitué :

- ✓ d'un ensemble d'ordinateurs autonomes capables d'effectuer des traitements,
- ✓ qui ne partagent ni mémoire (physique locale), ni horloge,
- ✓ et qui échangent des informations entre eux par l'intermédiaire d'un réseau de communication.

N.B. par la suite, il sera employé indifféremment le terme de « réparti » ou de « distribué ».

1.2. Utilité d'un système réparti

Une amélioration des performances…

Un système réparti représente une unité fonctionnelle, affranchie des contraintes géographiques. En effet, les ordinateurs participants au système collaborent à l'amélioration des performances d'une application :

- ✓ Partage des ressources : partage de fichiers, exploitation de bases de données centralisées, partage de services communs à plusieurs entités.
- ✓ Efficacité : l'accélération des calculs par la mise en commun de plusieurs unités de calcul autorise la parallélisme des calculs et la répartition de charge.
- ✓ Disponibilité : la redondance du matériel et des données permet de pallier les défaillances et le transfert de fonctions.
- ✓ Communication : chaque ordinateur effectue des traitements (autonomie locale) et échange des messages avec les autres membres du système réparti via un réseau de communication.
- ✓ Evolutivité : adaptation de l'architecture technique et logicielle aux besoins.

Dans l'idéal, un système réparti doit apparaître comme un système centralisé conventionnel. L'interface utilisateur d'un tel système ne doit pas faire la différence entre ressources locales et distantes, et il appartient à ce système de localiser les ressources et de mettre en place l'interaction appropriée. En outre, un système réparti doit fournir les mécanismes permettant la synchronisation des processus, la gestion des communications interprocessus, une solution au problème d'interblocage, etc… ; soit, la gestion de tous les problèmes inconnus dans un système centralisé.

… mais des contraintes supplémentaires :

Cependant, la répartition des traitements au sein d'un système réparti impose certaines contraintes :

- ✓ Fiabilité des communications : le bon fonctionnement de l'application dépend de traitements s'effectuant sur des ordinateurs distants qui communiquent leurs résultats via un réseau de communication qui doit donc s'avérer fiable.
- ✓ Hétérogénéité des environnements d'exécution : la diversité des architectures matérielles et logicielles rencontrées nécessite des traitements particuliers.
- ✓ Conception de l'application : la coordination des programmes (absence de hiérarchie), et la maintenance d'un état global cohérent du système (contrôle décentralisé) pose des problèmes algorithmiques spécifiques.

La répartition est une source de complexité qui s'ajoute aux problèmes habituels de conception et de réalisation d'une application.

Historiquement, il appartenait aux développeurs d'applications de résoudre eux-mêmes les problèmes liés à la répartition. En particulier, ils devaient programmer directement la couche communication de chaque machine, s'assurer de la coordination entre composants, et choisir une représentation de l'information commune acceptée par tous les participants.

C'est pour faciliter la conception et le développement d'applications réparties que des modèles de répartition ont été proposés.

2. LES MODÈLES DE RÉPARTITION

2.1. Présentation des différents modèles de répartition

Un modèle de répartition représente un ensemble d'abstractions permettant de spécifier les interactions entre composants selon une logique fonctionnelle plutôt qu'en termes de moyens techniques utilisés pour transporter l'information de l'un à l'autre. Par exemple, le modèle « objets répartis » décrit les interactions en termes d'appels de méthodes entre des objets déployés sur des nœuds différents.

Modèles de répartition : stratégies permettant aux applications réparties de communiquer.

2.1.1 Communication par passage de messages

Un protocole de bas niveau…

Il s'agit d'un protocole de communication de bas niveau (échange de flux non structurés d'octets) entre processus ou threads répartis, utilisé par les systèmes répartis primitifs. La sémantique qui anime cet échange est du type envoi/réception (conversationnel) : des messages indépendants unidirectionnels sont envoyés sur un réseau fiable (TCP) ou non fiable (UDP). On se situe « au ras du câble ».

La connexion par sockets symbolise le type de connexion « poste à poste » (« peer to peer » ou « d'égal à égal ») : l'envoi de message se fait au moyen de primitives spécialisées du système d'exploitation.

Le schéma du dialogue est celui de l'architecture Client/Serveur : un Serveur de socket attend des requêtes entrantes de la part d'un appelant sur un port spécifique, dès qu'il reçoit une requête, il effectue les calculs adéquats et retourne les résultats à l'appelant qui les attendait. La plupart des premières applications Client/Serveur ont été implémentée avec ce protocole.

… qui suppose l'existence des fonctionnalités suivantes :

- ✓ Localisation des services : l'émetteur doit disposer d'une référence qui désigne le destinataire sans ambiguïté au sein de l'application répartie.
- ✓ Synchronisation : l'échange de messages implique une synchronisation entre émetteur et destinataire qui doit être prévue par le programmeur (à quel moment les traitements locaux peuvent-ils se poursuivre ?)
- ✓ Hétérogénéité : pour faire coopérer des calculateurs hétérogènes, le programmeur doit définir lui-même une représentation commune des données, ainsi que les fonctions de conversion des représentations natives propres à chaque calculateur vers la représentation commune qui a été choisie (emballage des données), et réciproquement (déballage des données). A cette fin, on peut utiliser un dictionnaire du type XDR (eXternal Data Representation) qui normalise le format des entiers et des flottants.

2.1.2 Appels de sous-programmes à distance (RPC : Remote Procedure Call)

Un développeur d'application peut désirer un service de communication plus structurée que l'échange de messages, afin de résoudre les problèmes récurrents de représentation des données et de synchronisation.

Un protocole de haut niveau synchrone…

Le schéma Client/Serveur mis en œuvre par l'envoi de message est très similaire à un appel de sous-programme local. La sémantique qui anime cet échange est du type requête/réponse. La communication est synchrone : l'appelant attend les résultats de sa requête auprès d'un autre composant avant de poursuivre son exécution. Le but est d'étendre l'appel de sous-programme standard au cas où les composants « appelant » et « appelé » résident sur des nœuds différents d'un système réparti. On introduit alors un mécanisme d'appel de sous-programmes à distance.

… qui suppose l'existence des fonctionnalités suivantes :

- ✓ Localisation des services : l'association d'un Client et d'un Serveur est appelée liaison (« binding »). L'information de liaison peut être figée chez le Client ou découverte au moment de l'exécution : liaison dynamique via un annuaire de services.
- ✓ Hétérogénéité : comme dans le cas du passage de messages, les données manipulées doivent pouvoir être mises sous une représentation commune à tous les nœuds.
- ✓ Passage de paramètres : du fait du passage de paramètres, un sous-programme à distance ne peut être appelé directement : il faut connaître sa signature (type des paramètres et des valeurs de retour). La solution est de générer automatiquement deux composants logiciels : une souche (« stub ») et un squelette (« skeleton ») de sous-programmes.

  La souche est un module qui réalise l'emballage des paramètres (« marshalling ») de l'appel, et l'envoi de la requête au nœud sur lequel ce sous-programme réside. La souche se met ensuite en attente d'un message. Sur ce second nœud, le squelette , qui est en attente de tels messages demandant l'exécution de sous-programmes, déballe les paramètres « unmarshalling »), et appelle le corps du sous-programme réel. Lorsque cet appel se termine, il emballe la valeur de retour dans un nouveau message, qui est envoyé au nœud demandeur. La souche récupère et déballe alors la valeur de retour et la retourne à l'appelant, dont l'exécution se poursuit.

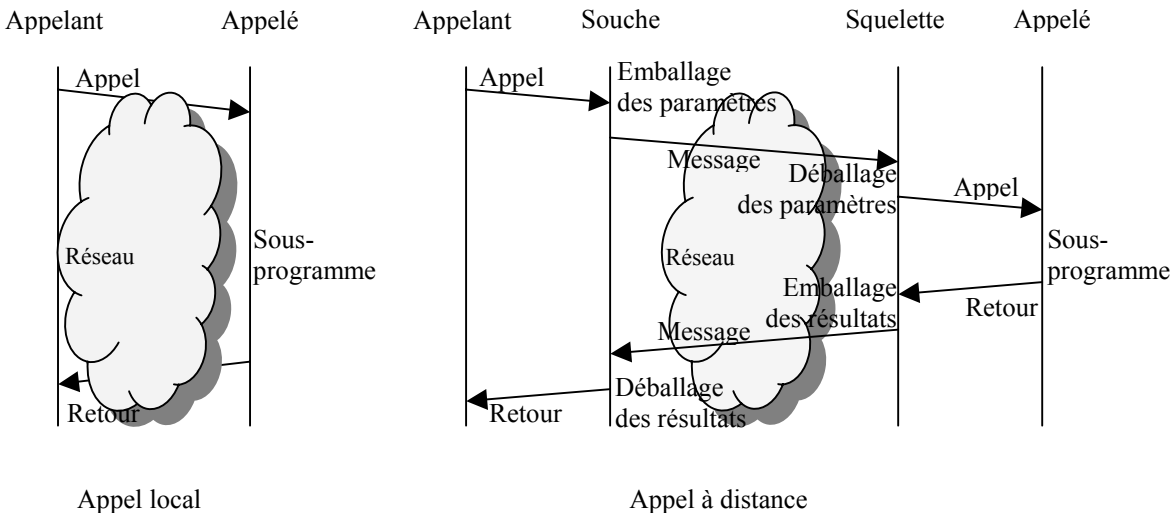


Fig.1 : Mécanisme d'appel de sous-programme

Du point de vue de l'appelant, tout se passe donc exactement comme si le sous-programme appelé était local. De même, du point de vue de la mise en œuvre du sous-programme, rien ne distingue un appel émanant d'un nœud distant d'un appel purement local.

- ✓ Communication : Le passage de messages sous-jacent est entièrement masqué par le code généré et les bibliothèques de communication qu'il utilise : RPC gère le canal de communication.

… et qui dispose de services évolués :

- ✓ Service de nommage : enregistre les coordonnées des services distants.
- ✓ Service de sécurité : fournit des fonctions d'authentification et de chiffrement.
- ✓ Gestion des pannes : contrôles par timeout et RPC orientée connexion.

2.1.3 Les objets répartis et l'invocation de méthodes à distance (RMI : Remote Method Invocation)

Programmation Orientée Objets

Par rapport à la programmation structurée, la Programmation Orientée Objets, en introduisant les concepts d'encapsulation, de polymorphisme et d'héritage, est beaucoup mieux adaptée à la création de systèmes Client/Serveur flexibles parce que les données et la logique de fonctionnement (méthodes) sont « encapsulées » dans les objets, ce qui leur permet de s'implanter n'importe où dans un système réparti. La granularité de la distribution est améliorée.

L'ensemble des méthodes d'un objet constitue son interface. Un objet étant unique, on peut lui associer une référence permettant d'appeler ses méthodes. Le polymorphisme et l'héritage facilitent la réutilisation du code, et permettent la maintenance et l'ajout de fonctionnalités à une classe d'objets.

Une application se présente désormais comme un ensemble d'objets qui s'envoient des messages pour appeler les méthodes désirées.

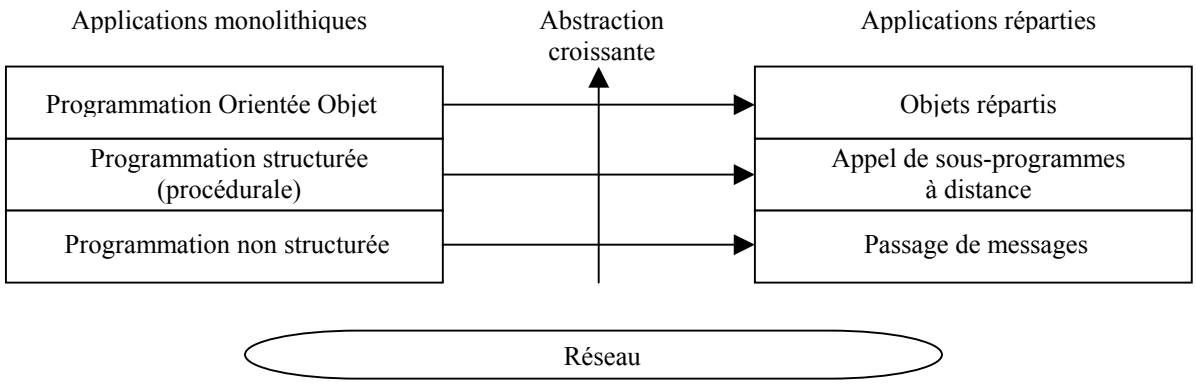


Fig.2 : corrélation entre méthodes de développement et modèles de répartition

Objets répartis

Un objet « classique » n'a d'existence que dans un seul programme (seul le compilateur qui les crée connaît leur existence).

Un objet « réparti » est un objet qui peut vivre n'importe où sur le réseau et sur lequel il est possible d'invoquer à distance une méthode de l'objet.

Au niveau fonctionnel, les objets répartis peuvent être vus comme des composants logiciels autonomes (« boîtes noires ») : fragments de code qui ne sont pas liés à un programme, un langage, un système d'exploitation ou un réseau particuliers. Les composants sont des objets libérés des chaînes qui les liaient à un langage ou une plate-forme particulière. Une application répartie se présente alors comme une composition de composants. Dans les systèmes à objets répartis, le composant est l'unité de travail et de répartition.

Invocation de méthodes distantes

Pour effectuer un appel de méthode à distance, l'appelant doit disposer d'une référence identifiant l'objet cible. Une référence distante, désignant un objet cible dans une application répartie, est une information communicable entre nœuds de l'application qui désigne sans ambiguïté cet objet, et permet d'effectuer des appels de méthodes à distance. Un objet réparti est en fait un pointeur sur un objet distant.

L'exécution d'un appel de méthode sur un objet distant se déroule en deux temps :

1. l'appelant utilise la référence distante désignant l'objet pour déterminer sur quel nœud celui-ci réside,
2. il peut alors composer et émettre le message représentant l'appel de la méthode, comme le ferait une souche cliente dans le cas d'un appel de sous-programme à distance. En plus de l'identité du sous-programme appelé et des paramètres, le message doit contenir l'identité de l'objet destinataire.

Le modèle « objets répartis » suppose l'existence des fonctionnalités suivantes :

- ✓ Localisation des services : une mise en œuvre du modèle « objets répartis » comporte donc en principe :
  - Un moyen d'affecter à un objet une référence distante. Une telle référence doit identifier l'objet au sein de l'application répartie : nœud sur lequel il réside, ainsi que l'identité de l'objet.
  - Un mécanisme permettant d'exécuter un appel de méthode sur un objet connu par une telle référence, au moyen d'un double aiguillage (d'abord en fonction du nœud destinataire, puis en fonction de la classe de l'objet).
  - Un moyen de diffuser ces références sur le réseau.

- ✓ Hétérogénéité : convenir d'un modèle de type global, permettant d'associer à chaque référence d'objet, la détermination de l'interface offerte par l'objet qu'elle désigne : des mécanismes de conversion de types doivent être offerts, permettant la conversion d'une référence d'une classe en une référence d'une autre classe (on ne parle plus de conversion entre types de données).

2.1.4 Mémoire partagée répartie (DSM : Distributed Shared Memory)

Les interactions entre composants se font par échanges de données et non plus par échanges de messages

Une application répartie, basée sur la mémoire partagée répartie, procède selon un modèle de partage d'informations, et non plus d'échange d'informations comme dans les cas précédents. Les zones de mémoire partagées ne sont pas localisées sur un nœud particulier mais dupliquées sur chacun des nœuds qui y accèdent. Des copies des objets manipulés sont conservées sur différents nœuds (à l'opposé des modèles précédents où les abstractions du modèle sont localisées sur un nœud particulier). Les mécanismes d'échange d'informations sont dissimulés par l'exécutif de répartition qui assure la synchronisation des copies locales des informations partagées.

Existence des fonctionnalités suivantes :

- ✓ Localisation des services : pour le développeur, ce modèle de répartition est transparent car il n'a pas à se préoccuper de la localité des différents composants. Tout se passe comme si ceux-ci s'exécutaient sur une même machine, et accédaient aux mêmes données.
- ✓ Communication : Les échanges de messages nécessaires sont déterminés par le choix d'un modèle de cohérence : ensemble de propriétés que doivent vérifier les accès en lecture et en écriture à un objet partagé donné, et aux copies locales de cet objet dont dispose chaque nœud.
- ✓ Hétérogénéité : tout se passe comme si les différents composants accédaient aux mêmes données (modèle de partage d'informations).

2.1.5 Files d'attente de messages (MOM : Message Oriented Middelware)

Un protocole de haut niveau asynchrone…

Mécanisme qui permet d'échanger des messages dans un système Client/Serveur au moyen de files d'attente de messages. Les applications communiquent sur le réseau en déposant et retirant simplement des messages dans des files d'attente (paradigme producteurs/consommateurs). Ce système permet aux Clients et au Serveur de communiquer de manière asynchrone sans être liés par une connexion logique. Le Serveur et le Client peuvent opérer à des vitesses et à des moments différents. Un consortium MOM a été créé en 1993 pour normaliser le Middleware orienté messages.

… qui suppose l'existence des fonctionnalités suivantes :

- ✓ Localisation des services : la plupart des implémentations permettent à l'émetteur de désigner le nom de la file d'attente des réponses.
- ✓ Hétérogénéité : des descripteurs de format indiquent au récepteur comment interpréter les données du message.
- ✓ Communication : MOM masque aux applications toute la partie communication.

… et qui dispose de services évolués :

- ✓ Service de nommage : nommage hiérarchique.
- ✓ Service de sécurité : fournit des fonctions d'authentification et de chiffrement.
- ✓ Gestion des défaillances : les files d'attente persistantes (sur disque, opposées aux non persistantes en mémoire) offrent un premier niveau de tolérance aux pannes. Une forme de protection transactionnelle permet à une file d'attente de participer à un protocole de validation à deux phases. Les messages peuvent être re-routés vers des files d'attente de remplacement en cas de défaillance du réseau.

2.2. Fonctionnalités complémentaires des modèles de répartition

Dans un souci de cohérence et d'homogénéité, on pourrait s'attendre à ce qu'un certain nombre de fonctionnalités soient intégrées par les modèles de répartition Ces fonctionnalités sont généralement remplies par le Middleware mais de façon spécifique pour chacun. Il s'agit des fonctionnalités suivantes :

2.2.1 Gestion des exceptions

Un mécanisme d'exceptions peut être utilisé pour signaler à un composant appelant qu'un appel de sous-programme ou de méthode s'est terminé d'une façon anormale. Si le langage hôte supporte lui-même un mécanisme d'exception, ce message peut être traduit en une exception native qui sera propagée au composant applicatif appelant.

2.2.2 Avortement d'appels distants

L'utilisateur peut souhaiter annuler avant l'issue normale de son exécution un appel de méthode ou de sous-programme (surtout dans le cas de systèmes temps réel). Comme la levée d'exceptions, l'avortement est lié au mode d'interaction par requête/réponse, et ne s'applique pas au cas du passage de messages.

2.2.3 Appels unidirectionnels

L'utilisateur peut désirer poursuivre son traitement local sans attendre la confirmation de fin d'exécution de l'appel à distance. Un appel unidirectionnel est donc la construction qui, dans une application répartie basée sur l'appel de sous-programmes distants ou les objets répartis, se rapproche le plus d'un simple passage de message.

2.2.4 Transactions

Des ensembles d'échanges (messages, appels de sous-programmes ou de méthodes, actions sur un espace de stockage partagé) peuvent être groupés au sein de transactions, afin de préserver certaines propriétés de cohérence globale du système. Les propriétés généralement offertes par les mécanismes de transactions sont connues sous le nom d'ACID (Atomicité, Cohérence, Isolation, Durabilité).

B. Mise en œuvre des modèles de répartition

Dans les premiers temps des applications distribuées, le développeur dépensait beaucoup d'énergie à gérer les interfaces de programmation réseau (sockets TCP ou UDP par exemple).

Avec les nouvelles exigences liées à la répartition et à l'approche objet, il devenait nécessaire d'adopter des techniques de développement plus structurées. Il est préférable que le développeur se concentre sur les aspects fonctionnels de l'application plutôt que sur les problèmes de communication de bas niveau.

L'utilisation de modèles de répartition libère le développeur de contraintes liées à la gestion des communications entre nœuds du réseau ou à l'hétérogénéité des architectures matérielles et logicielles.

Nous allons voir ici comment certains modèles de répartition sont mis en œuvre selon les deux approches étudiées au chapitre précédent, à savoir dans un langage distribué comme Ada 95 et dans un intergiciel comme CORBA.

1. LES LANGAGES DISTRIBUÉS

1.1. Apport de la conception Orientée Objet

La Programmation Orientée Objet, avec ses mécanismes d'abstraction, d'héritage, d'encapsulation, et d'instanciation, a modifié la façon de concevoir les applications : un ensemble de composants logiciels indépendants qui s'échangent des messages. La tentation était grande de vouloir étendre les capacités de tels langages objets de façon à ce qu'ils prennent en charge les spécificités engendrées par une utilisation délocalisée de ces composants sur un réseau. Le langage doit donc soit intégrer en son sein, soit trouver dans des bibliothèques externes, toutes les fonctionnalités nécessaires à la bonne marche d'une application répartie (fournir des abstractions de haut niveau qui rendent transparentes au développeur toutes les primitives implémentées pour fournir des services de bas niveau).

1.2. Avantages

Puisque le langage s'occupe des différentes corvées liées à la répartition, tous les modules de l'application seront développés à l'aide de ce langage unique. Cette absence d'interopérabilité avec d'autres langages a des aspects positifs :

- ✓ Développer dans un environnement homogène permet d'obtenir des applications fiables et robustes, avec l'assurance de la conservation des types (pas de conversion de types) et de pourvoir réutiliser des modules écrits dans un langage ou la compatibilité ascendante est assurée. L'environnement de développement et le compilateur peuvent effectuer un ensemble de contrôles, sémantiques notamment, qui s'appliquent à tous les éléments entrant dans la composition de l'application.
- ✓ La définition de l'interface des services et leur implémentation se fait dans le même langage au sein des programmes. L'implémentation est identique pour une utilisation locale ou distante.
- ✓ La mise au point de programme est aisée car elle s'effectue dans l'environnement de programmation avec une gestion des erreurs bien définie.
- ✓ Chaque service peut être spécialisé et optimisé pour gérer au mieux un problème particulier.

1.3. Exemples

- ✓ Ada 95, langage orienté objet reconnu (norme ISO), s'est donc vu adjoindre, selon ce principe, une annexe traitant des problèmes soulevés par les systèmes distribués et les applications réparties (annexe E).

✓ Un autre exemple de langage distribué, qui rend transparent l'exécution de méthodes distantes, est celui de Java associé à RMI (Remote Method Invocation).

2. LES INTERGICIELS DE RÉPARTITION

N.B. dans tout ce qui suit, il sera employé indifféremment les termes de « Middleware » ou d' « intergiciel».

2.1. Rôle de l'intergiciel

La mise en œuvre d'un système réparti suppose l'existence de logiciels qui fournissent aux composants applicatifs les abstractions d'un ou plusieurs modèles de répartition.

Un intergiciel fournit au développeur d'application répartie, les moyens de faire interagir des composants distants sans se préoccuper des tâches liées à la répartition.

Il s'agit donc d'une solution logicielle chargée de mettre en place une couche d'abstraction de haut niveau dans le but de simplifier le développement et le fonctionnement d'applications distribuées.

Un intergiciel simplifie le développement d'applications réparties en prenant en charge certains aspects de la répartition : gestion des communications entre les noeuds, hétérogénéité des architectures matérielles et logicielles, apport de nombreux services de haut niveau.

Tout doit se passer comme si l'application se déroulait en local...

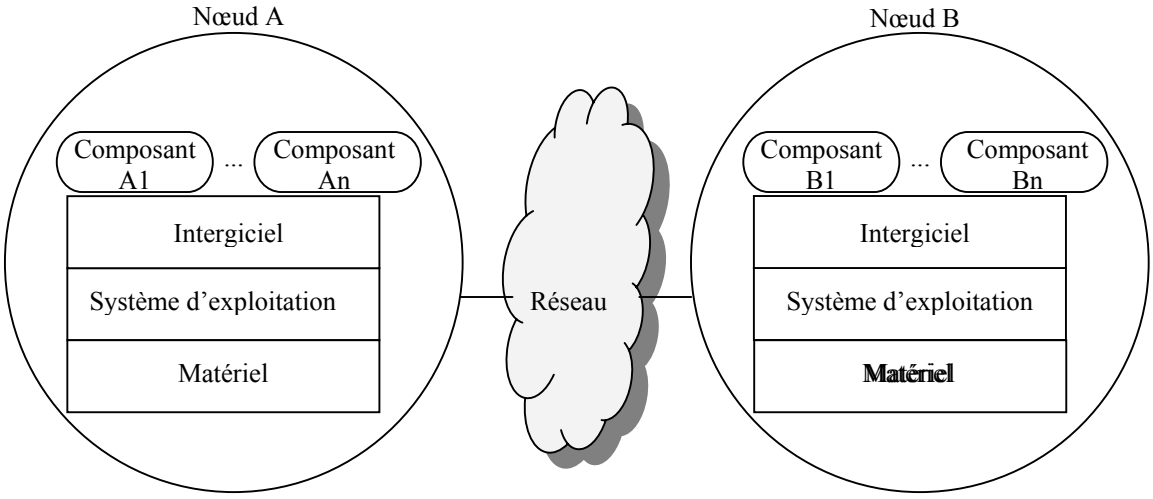


Fig.3 : structure d'une application répartie utilisant un intergiciel

2.1.1 Transparence vis à vis de la localisation des composants :

Pour le développeur d'une application utilisant un modèle de répartition de haut niveau (appel de sous-programmes à distance, objets répartis ou mémoire partagée, par exemple), la répartition est introduite de façon « transparente », dans la mesure où tout se passe comme si l'application se déroulait localement.

La répartition consiste alors à projeter les entités ainsi délimitées sur les différents noeuds participant à l'application répartie.

Les échanges de messages sous-tendus par le modèle de répartition sont effectués par l'intergiciel.

2.1.2 Independance vis à vis de la plate-forme d'exécution :

Les composants de l'application utilisent les fonctions de l'intergiciel pour communiquer entre eux de façon transparente ; l'intergiciel réduit la dépendance des composants vis-à-vis de l'environnement d'exécution, en masquant les aspects d'interfaçage de bas niveau avec des primitives de communication de haut niveau.

Chaque nœud possède donc une instance de l'intergiciel. Ces instances communiquent entre elles au moyen de primitives de bas niveau, et jouent le rôle d'intermédiaires entre les composants de l'application qui résident sur le même nœud et ceux qui résident sur d'autres nœuds. Les interactions avec ces derniers sont réalisées par échange de messages entre les instances d'intergiciel locale et distante.

2.2. **Fonctions de l'intergiciel**

... ce qui suppose que le Middleware fournit des fonctionnalités qui « masquent » la répartition…

2.2.1 Adressage/nommage

L'intergiciel doit avant tout définir un moyen de nommer les diverses entités qui participent à une application répartie : il leur attribue des identifiants susceptibles d'être échangés et compris par chacun des nœuds de l'application. L'intergiciel remplit une fonction d'adressage des composants de l'application.

2.2.2 Transport

L'intergiciel assure le transport de messages entre les nœuds, en utilisant le réseau de communication sous-jacent.

2.2.3 Protocole et représentation

L'intergiciel met en œuvre un ou plusieurs protocoles de communication et se charge également de mettre les données à échanger sous une forme transmissible, et compréhensible par les nœuds distants, c'est-à-dire qu'il applique une syntaxe et une sémantique aux messages.

Protocole et représentation sont indépendants d'un langage de programmation ou d'une architecture matérielle. L'utilisation d'un intergiciel permet donc de faire interagir, au sein d'une même application répartie, des composants s'exécutant sur des architectures matérielles et dans des environnements logiciels différents.

En contrepartie, le choix d'un protocole et d'une représentation limite les interactions possibles aux composants dont l'intergiciel supporte ce protocole.

2.2.4 Liaison

Un objet qui dispose d'une référence désignant un autre objet, à distance, ne peut directement en appeler les méthodes. Il est nécessaire, au préalable, que l'intergiciel fasse la liaison entre la référence d'objet et les ressources nécessaires à l'appel de méthodes sur l'objet distant.

2.2.5 Activation/ cycle de vie

L'intergiciel doit s'assurer que la structure concrète associée à un identifiant objet existe bien et que l'association entre les deux composants soit maintenue. On désigne cette fonction de contrôle du cycle de vie des associations entre identifiants et composants applications par le terme d'activation.

2.2.6 Exécution

L'intergiciel permet la synchronisation des échanges et des traitements. Là encore, il reflète le modèle de répartition mis en œuvre :

- Exécution synchrone : un composant qui demande à un autre d'exécuter un service, attend la fin de l'exécution de ce service (matérialisée par la réception d'un message) avant de poursuivre son déroulement.
- Exécution asynchrone : un composant qui demande à un autre d'exécuter un service, poursuit son propre traitement pendant que le second composant traite sa demande.

2.3. **Limites introduites par l'intergiciel**

... mais qui engendrent certaines contraintes.

L'utilisation d'un intergiciel dans une application répartie, par opposition à l'utilisation directe des primitives de communication fournies par le système d'exploitation, permet l'utilisation d'un modèle de répartition sophistiqué. Ce gain en abstraction facilite également la mise en place d'applications hétérogènes.

Toutefois, les apports des intergiciels s'accompagnent de plusieurs contreparties.

2.3.1 Coûts relatifs des différents modèles de répartition

À chaque modèle de répartition correspond un coût de mise en œuvre.

Plus les fonctionnalités offertes sont importantes, plus l'intergiciel est volumineux, et plus les ressources nécessaires sont considérables.

Un intergiciel offrant le modèle « passage de messages » peut n'introduire qu'un surcoût minime par rapport à l'utilisation directe des primitives de communication du système d'exploitation sous-jacent. En effet, il rend un service d'un faible niveau d'abstraction, dont la réalisation transparente ne nécessite pas d'opérations complexes.

Lorsqu'un modèle plus évolué (appel de sous-programmes à distance ou objets répartis) est mis en œuvre, en revanche, l'intergiciel effectue des traitements complexes. Il occupe donc plus d'espace en mémoire. Les fonctions avancées sont également susceptibles d'introduire des délais de traitement supplémentaires pour tous les messages échangés, et d'augmenter le volume de données à transmettre entre les nœuds.

2.3.2 choix d'un modèle de répartition

Les intergiciels sont liés au choix d'un modèle de répartition en général, et au choix de leur mise en oeuvre en particulier. Deux mises en œuvre du même modèle ne sont pas nécessairement compatibles.

Cette contrainte tendant à imposer le choix d'un seul intergiciel, pour l'ensemble d'une application, s'oppose à l'interopérabilité entre composants divers, et à fait naître le besoin d'interconnecter les intergiciels par des passerelles réalisant l'adaptation entre modèles de répartition. Celles-ci fondent la notion de communication entre intergiciels, ou « M2M » (Middleware to Middleware).

2.4. **Exemples**

- ✓ CORBA et son sous-système de communication ORB.
- ✓ COM+ (Common Object Model) et son sous-système de communication DCOM (Distributed Common Object Model) dans le monde Microsoft.

II. EXEMPLES D'IMPLÉMENTATIONS DE MODÈLES DE RÉPARTITION

Dans la première partie, nous avons parcouru les aspects théoriques de la répartition : quels modèles existent et comment ils peuvent être mis en oeuvre théoriquement dans le cas d'un langage distribué et dans celui d'un intergiciel.

Forts de cet acquis technique, nous allons étudier en détail deux façons d'implémenter des modèles de répartition : Ada 95 et son annexe des systèmes répartis en tant que langage distribué, et CORBA en tant qu'intergiciel de répartition.

A. ADA 95 et DSA : du langage de programmation au langage distribué

Le langage Ada est bien introduit dans les domaines dits « sensibles » (aéronautique, aviation, domaine militaire, nucléaire, contrôle de processus) grâce à un certain nombre de qualités intrinsèques.

Le développement d'applications réparties fait l'objet de l'annexe E du manuel de référence du langage Ada 95 : l'annexe des systèmes répartis (DSA : Distributed System Annex).

N.B. par la suite, il sera employé indifféremment le terme de « annexe des systèmes répartis» ou de « DSA ».

1. LE LANGAGE ADA

1.1. Historique

En 1975, le ministère de la Défense américain lance un appel d'offre pour définir un nouveau langage de programmation capable de remplacer ceux (plus de 450 !) utilisés par ses services.

Ada fut conçut avec des objectifs bien définis : respect des principes de génie logiciel, langage généraliste (applications de gestion, scientifique, temps-réél et embarqués), développement de projets importants, en équipe, et de maintenance aisée.

Ada 95 : premier langage objet normalisé (1995)

La publication du standard ANSI (American National Standard Institute) en 1983 (Ada 83), a été certifiée par l'ISO (International Standards Organization) en 1987. Un processus de révision débuté en 1988 (Ada 9X), aboutit en 1995 (Ada 95) à un nouveau standard ISO, consacrant Ada comme premier langage objet a avoir été normalisé (trois ans avant le C++).

Outre l'adoption de nouvelles fonctionnalités orientées objets par rapport à Ada 83, un modèle pour le développement d'applications réparties et de nouveaux mécanismes pour le temps-réel ont été intégrés. La compatibilité ascendante a aussi été assurée.

En marge des applications embarquées de l'aéronautique et des transports, qui sont ses fers de lance, Ada est aussi utilisé pour l'apprentissage de la programmation avec des bases rigoureuses.

1.2. Caractéristiques

Ada 95 favorise la conception orientée objet

De nombreux atouts permettent à Ada de se prévaloir des qualités de fiabilité, robustesse, réutilisabilité, extensibilité qui lui sont habituellement reconnues :

- ✓ Typage fort : détection d'erreurs à la compilation, exécution plus efficace.
- ✓ Modularité : le but est de réduire la complexité globale d'un problème en le scindant en petites unités indépendantes, en « composants logiciels » réutilisables. L'emploi de sous-programmes, paquetages (collections d'entités et de « ressources » logiques), et de la composition incrémentale (paquetages et bibliothèques hiérarchiques) permet d'améliorer le découpage en modules. Cette structuration en unités facilite la mise au point et l'évolution des programmes, la compilation séparée, la réutilisation, la portabilité. De plus, il existe un certain nombre de bibliothèques normalisées (paquetages prédéfinis) qui fournissent des fonctionnalités valides et sûres.
- ✓ Généricité : les unités génériques (sous-programmes et paquetages) permettent de factoriser des parties de code pour des types différents.
- ✓ 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 (« spécification » ou « déclaration »), 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.
- ✓ Programmation Orientée Objet : Ada support l'héritage simple (multiple avec des constructions de base du langage via la composition des génériques), le polymorphisme, la liaison dynamique (aiguillage dynamique d'appel de méthodes). Une classe se définit par un paquetage déclarant un type étiqueté « tagged » (toute instanciation de ce type crée un objet). Ada 95 est le premier langage objet normalisé par l'ISO.
- ✓ Sécurité : synchronisation des interactions et des échanges d'informations entre tâches par le mécanisme des rendez-vous (intégrité), méthode de récupération systématique des exceptions (robustesse), contrôle de l'élaboration des paquetages, validation des compilateurs Ada par un institut indépendant qui garantit le respect de la norme.
- ✓ Efficacité : la concurrence permet d'écrire des programmes qui exécutent des tâches parallèles.
- ✓ Annexes spécialisées : des annexes optionnelles (pas obligatoirement fournies avec le compilateur validé) et normatives (celles fournies respectent les spécifications) portent sur des domaines spécialisés comme par exemple :
  - Interfaçages avec les langages normalisés (C, C++, Fortran, Cobol) (annexe B)
  - Programmation système (annexe C)
  - Systèmes temps réel (annexe D)
  - Systèmes distribués et les applications réparties (annexe E)
  - Sûreté de fonctionnement (annexe H)

Ada 95 possède une annexe spécialisée traitant des applications réparties.

Dans le cadre de cette étude, nous allons nous intéresser en particulier à l'annexe E du langage, à savoir celle qui traite de la mise en œuvre des systèmes distribués et des applications réparties.

2. L'ANNEXE DES SYSTÈMES RÉPARTIS : DSA (DISTRIBUTED SYSTEM ANNEX)

L'annexe des systèmes répartis indique comment développer en Ada dans le cadre des systèmes distribués et des applications réparties, tout en respectant les spécifications du langage (notamment le typage fort). Elle étend les mécanismes d'abstraction du langage, matérialisés par les paquetages, au cas d'applications réparties.

2.1. Principes de l'annexe

Ada 95 permet de construire des systèmes répartis sans sortir du cadre du langage

2.1.1 Directives de compilation

Les mêmes sources des programmes doivent servir à la fois pour une application monolithique et pour sa version répartie.

Ainsi, la norme Ada ne prévoit aucune fonction, ni aucun type spécifique à utiliser à cette fin. Ce sont des directives de compilation (ou « pragma » : indication à destination du compilateur sans effet sémantique) qui permettent d'attribuer des propriétés supplémentaires à certaines unités de bibliothèque (unités de programmes non imbriquées dans une partie déclarative donc unités d'abstraction de plus haut niveau) vis-à-vis de la répartition. Ces directives garantissent que les règles de typage, visibilité et sécurité seront préservées.

N.B. par la suite, il sera employé indifféremment le terme de « directive de compilation» ou de « pragma ».

2.1.2 Notion de partition

En Ada 95, **l'unité de partition est le paquetage**, eux-mêmes regroupés en partitions qui représentent des nœuds logiques. On distingue deux types de partitions :

1. Les partitions actives : émettent et reçoivent des requêtes.
2. Les partitions passives : servent d'espace de stockage utilisé par une ou plusieurs tâches situées dans les partitions actives.

La communication entre partitions actives s'effectue par le biais d'appels de sous-programmes ou d'invocation de méthodes sur des objets répartis. Pour des raisons de compatibilités ascendantes, il n'est pas possibilité d'utiliser, pour la synchronisation, le mécanisme des rendez-vous entre tâches situées sur des partitions différentes.

2.2. Mise en oeuvre de l'annexe

2.2.1 Utilisation des directives de compilation

L'annexe introduit trois nouvelles directives de compilation dont le but est d'attribuer des propriétés de répartition à certains paquetages :

1. **Remote_Types** (RT) : désigne un paquetage dont les types peuvent être transportés entre les différentes partitions. Les types transportables peuvent être transmis entre partitions en conservant la même signification sémantique. Des références sur des sous-programmes ou sur des objets distants pourront être définies. Un paquetage de ce type sera dupliqué sur toutes les partitions qui le référencent.
2. **Remote_Call_Interface** (RCI) : désigne un paquetage dont les sous-programmes peuvent être appelés à distance (RPC : Remote Procedure Call). Un tel paquetage définit l'interface d'un service localisé sur un nœud particulier d'une application répartie : il ne peut être dupliqué.
3. **Shared_Passive** (SP) : désigne un paquetage contenant des variables ou objets communs entre plusieurs partitions qui seront accessibles via un support partagé (mémoire, fichier, base de données). Un tel paquetage définit une partition passive partagée par l'ensemble des partitions qui prennent part à une application répartie : il ne peut être dupliqué.

L'annexe E prévoit aussi le pragma « Pure » : il contient 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 » a pour effet de rendre tout appel à un sous-programme unidirectionnel. Toute levée d'exception est alors ignorée.

La directive de compilation « All_Calls_Remote », appliquée à une unité de bibliothèque catégorisée « Remote_Call_Interface », a pour effet d'obliger tout appel à un sous-programme distant à transiter par le sous-système de communication même si l'appel peut être résolu en local (utile lors de la mise au point de l'application).

Exemple : le service Echo

Déclaration du service Echo	Utilisation du service Echo

package EchoSvc is
  pragma Remote_Call_Interface;
  -- les sous programmes présentés ici
  -- seront appelables à distance
  function EchoString(S :String)
  return S;
end EchoSvc
	with Ada.Text_IO; use Ada.Text_IO;
	with EchoSvc; use EchoSvc;
	procedure CallEcho is
	  begin
	  Put_Line(EchoString("abcde"));
	  -- génère appel à distance
	  exception
	  when others => Put_Line("Error EchoString");
	end CallEcho;

2.2.2 Localisation des services

En Ada, il est possible d'obtenir une référence ou pointeur distant, sur deux types d'entités qui appartiennent à un paquetage catégorisé Remote_Types ou Remote_Call_Interface :

1. **Références sur objets distants** : il faut se rappeler qu'un objet réparti :
   - n'est utilisé qu'à travers un pointeur sur objet distant (type étiqueté limité privé). Une telle référence peut pointer aussi bien sur un objet local que sur un objet distant,
   - n'est accédé qu'à travers ses méthodes. L'appel dynamique de méthode nécessite un double aiguillage (détermination de la partition où a été élaboré l'objet, puis de la méthode de l'objet),
   - ne se déplace pas physiquement, seules les références sont échangées.

2. **Références sur sous-programmes distants** : des pointeurs sur sous-programmes, déclarés dans les paquetages Remote_Types ou Remote_Call_Interface, deviennent automatiquement des pointeurs sur sous-programmes distants.

2.2.3 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. Les

exceptions levées remontent à l'appelant (types d'exceptions non perdus entre partitions). On a toujours la possibilité de rattraper une exception inconnue via une clause « when others ».

2.2.4 Sous-système de communication (PCS : Partition Communication Subsystem)

Un paquetage unique, « System.RPC », définit les types et les sous-programmes à utiliser comme interface avec le sous-système de communication (indépendance compilateur/communication). Le sous-système de communication prend en charge :

- ✓ l'envoi des requêtes d'appels de sous-programmes ou de méthodes à distance vers un Serveur,
- ✓ l'analyse des requêtes sur le Serveur pour lancer les traitements adéquats,
- ✓ l'envoi des résultats des traitements au Client,
- ✓ l'avortement d'appels distants.

2.2.5 Partitionnement

L'annexe ne décrit pas comment une application répartie doit être configurée, c'est-à-dire comment constituer des noeuds logiques à partir des unités de partition que sont les paquetages. Le partitionnement (assemblage de paquetages en partitions) est une opération post-compilatoire non normalisée à la charge du développeur. Le partitionnement permet :

- ✓ de passer d'une version monolithique à une version répartie et vice versa,
- ✓ de repartitionner une application sans recompilation,
- ✓ de debugger une application répartie en la rendant centralisée.

B. CORBA : le standard actuel de l'intergiciel de répartition

1. ORIGINES

1.1. L'OMG :

L'OMG (Object Management Group) a été fondé en 1989. Il s'agit d'un consortium international regroupant plus de 900 acteurs du monde informatique : constructeurs (ex : IBM, Sun), éditeurs de logiciels (ex : Netscape, Inprise, Iona Tech.), utilisateurs (ex : Boeing), universités (ex : INRIA).

L'OMG poursuit les objectifs suivants :

- ✓ Promouvoir la conception d'applications informatiques distribuées, interopérables et ouvertes.
- ✓ Développer des spécifications (spécifier les interfaces) mais pas des produits (implémentations ouvertes à la concurrence).
- ✓ Définir l'Object Management Architecture (OMA, architecture distribuée fondée sur les technologies orientées objet permettant la réutilisabilité et la portabilité des composants logiciels, l'hétérogénéité et l'interopérabilité des environnements informatiques.

1.2. Spécification de l'OMG : CORBA

L'OMG définit la spécification de l'architecture d'un système à objets répartis appelée CORBA (Common Object Request Broker Architecture ou architecture d'arbitrage de demande d'objet commun).

CORBA représente une couche logicielle intermédiaire (intergiciel ou Middleware) qui permet de faire communiquer des applications dans un environnement hétérogène au niveau des plate-formes matérielles et logicielles.

CORBA associe les concepts d'objet et de répartition dans une approche intégrée : **l'objet est l'unité de répartition**.

La première spécification de CORBA date de 1992. La norme 2.0 a été adoptée en 1995 et modifiée en 1998 (norme 2.2). La norme 3.0 a été publiée en 2000.

2. PRÉSENTATION

2.1. Caractéristiques

Des objets sont répartis sur un réseau et des applications peuvent communiquer avec eux. Ce postulat de base est satisfait par CORBA au moyen de concepts issus de la technologie objet.

2.1.1 Orientation Objet :

- ✓ Conception des applications en terme d'objets coopérants : la relation Client/Serveur s'applique entre des objets pour qu'ils coopèrent (coordination des interactions entre objets).
- ✓ Transparence vis à vis de la localisation des objets : les Clients n'ont à connaître ni l'endroit où réside l'objet distribué (même processus en local ou sur une machine distante connectée par Internet), ni le système d'exploitation sur lequel il s'exécute.
- ✓ Séparation stricte entre l'interface et l'implémentation des objets : les Clients n'ont pas plus à connaître l'implémentation de l'objet Serveur (langages et outils utilisés). **La seule chose que le Client a besoin de connaître, c'est l'interface que l'objet Serveur rend publique**. Cette propriété permet d'utiliser les interfaces des objets pour « encapsuler » des applications existantes (réutilisation de code existant).

2.1.2 Spécification de l'interface des objets distribués : l'IDL

L'IDL, un langage purement déclaratif

Les spécifications d'interface sont rédigées dans un langage générique – purement déclaratif (syntaxe proche de C++) – nommé (Interface Definition language). Le langage de définition d'interface permet de décrire les méthodes et leurs paramètres, les attributs, les classes parentes, les exceptions sur un objet, mais ne donne aucun détail sur l'implémentation de l'objet.

Ainsi, les fournisseurs de composants spécifient, via l'IDL, l'interface et la structure des objets qu'ils fournissent. Un contrat, défini par l'IDL, lie les fournisseurs de services d'objets distribués à leurs clients. En effet, un Client qui invoque une méthode répartie va communiquer avec l'interface précisée par l'IDL. De son côté, l'objet Serveur implémente l'interface indiquée par l'IDL.

Exemple d'interface IDL :

```
// fichier CompteBancaire.idl
Interface CompteBancaire        {
        void credit(in long unsigned montantC);
        void debit(in long unsigned montantD);
        long solde();
} ;
```

Les méthodes spécifiées dans l'IDL peuvent être écrites et invoquées dans tout langage capable de fournir des liens CORBA (C, C++, SmallTalk, Ada, COBOL et Java). Une **projection** est la traduction d'une spécification IDL dans un langage d'implémentation. Ce sont ces règles de projection qui ont été normalisées par l'OMG pour les langages cités.

Chaque produit CORBA fournit un pré-compilateur IDL pour les langages supportés (donc portabilité non garantie).

Nous verrons dans les paragraphes suivants que, au sein même de l'architecture CORBA, les éléments CORBAservices ou CORBAfacilities sont assemblés sous forme de composants dont les interfaces sont définies en IDL (généralisation du concept aux services).

L'ambition de CORBA est de « IDL-iser » (selon [L1]) tous les composants qui se trouvent liés à l'ORB.

De même, le Référentiel d'interfaces (IR : Interface Repository) contient la définition de toutes les interfaces cibles mais aussi des informations pour que les composants puissent se découvrir dynamiquement (à l'exécution) : c'est ce qui permet de dire que CORBA est auto-descriptif.

2.1.3 Concept de bus à objets:

- ✓ Autorise la communication entre éléments hétérogènes (langages, outils, systèmes d'exploitation, réseaux). Le Client n'a pas à connaître les mécanismes utilisés pour communiquer avec les objets Serveur quelles que soient les plate-formes mises en oeuvre.
- ✓ Masque la mise en œuvre des différentes couches logicielles nécessaires pour réaliser un système réparti.

2.2. **OMA (Object Management Architecture)**

CORBA s'inscrit dans le cadre de la définition du modèle d'objets distribués et interopérables : l'OMA.

Cette architecture de gestion des objets comprend un ensemble de « services objet » qui étendent les possibilités du bus à objet.

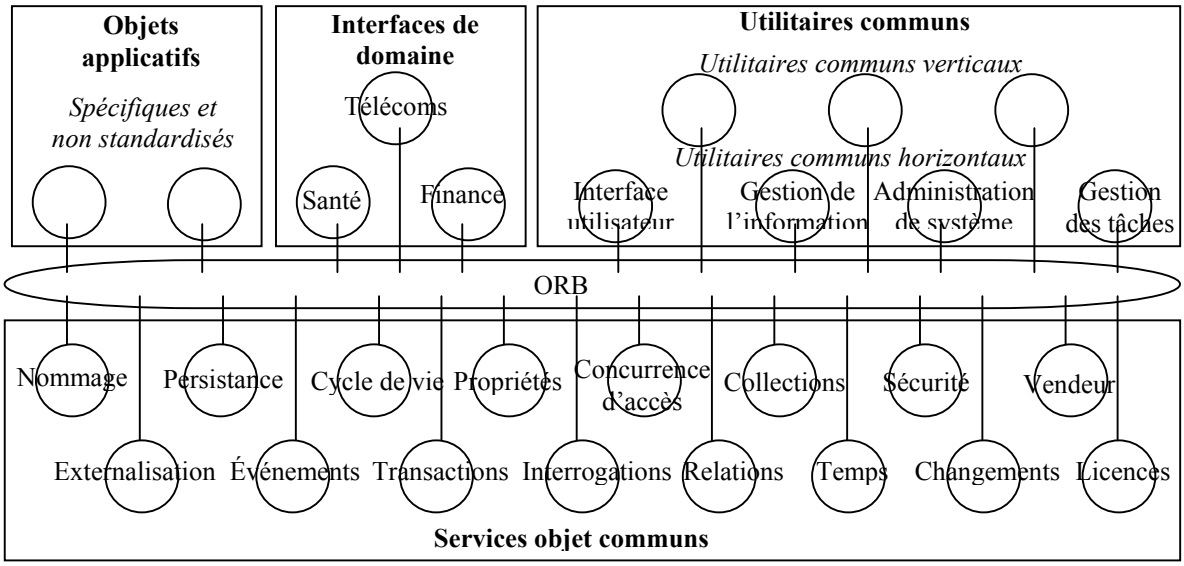


Fig.4 : OMA

2.2.1 ORB (Object Request Broker)

L'ORB ou « courtier d'objets », définit le bus à objets CORBA.

L'ORB constitue la colonne vertébrale de CORBA

L'ORB assure la véritable communication entre Client et Serveur. Il établit des relations de nature Client/Serveur entre des objets (mais uniquement pour coordonner les interactions entre deux objets).

L'ORB permet aux objets d'émettre de façon transparente des requêtes destinées à d'autres objets, locaux ou distants, et d'en recevoir des réponses.

Il permet également l'interopérabilité entre des composants d'applications réparties dans des environnements distribués hétérogènes.

Caractéristiques :

1. Invocation de méthodes statiques et dynamiques : ces deux mécanismes complémentaires permettent de soumettre les requêtes aux objets.
   - ✓ En statique, les invocations sont contrôlées à la compilation (définition des appels de méthodes à la compilation).
   - ✓ En dynamique, les invocations doivent être contrôlées à l'exécution (découverte dynamique des propriétés d'un objet à l'exécution).

2. Liaison avec des langages de programmation de haut niveau : les invocations de méthodes peuvent se faire via le langage de haut niveau de son choix : l'OMG a défini officiellement cette liaison pour les langages C, C++, SmallTalk, Ada, COBOL et Java. Etant donné que CORBA sépare l'interface de l'implémentation, et fournit des types de données indépendants du langage, les appels aux objets sont indépendants du langage et du système d'exploitation.

3. Transparence des invocations : les requêtes aux objets semblent toujours être locales, le bus CORBA se charge de les acheminer en s'occupant des piles de transport, de la localisation des Serveurs, de l'activation des objets, des différences d'agencement des octets entre plate-formes hétérogènes ou encore du système d'exploitation cible.

4. Système auto-descriptif : un référentiel d'interfaces contient, en temps réel, les informations décrivant les fonctions (et leurs paramètres) fournies par un Serveur. Les Clients utilisent ces métadonnées pour découvrir à l'exécution comment appeler certains services (permet génération de code « à la volée »).

5. Interopérabilité entre bus : un ORB peut fonctionner en mode « standalone » ou être interconnecté à un autre ORB (provenant d'un autre fournisseur d'ORB) grâce aux services des protocoles :
   - ✓ GIOP (General Inter-ORB Protocol) : ensemble de messages et représentations communes de données (CDR : Common Data Representation) pour assurer la communication entre ORB.
   - ✓ IIOP (Internet Inter-ORB Protocol) : définit la façon dont les messages GIOP s'échangent sur un réseau TCP/IP.
   - ✓ ESIOP (Environment-Specific Inter-ORB Protocols) pour les interactions particulières entre réseaux spécifiques (exemple : ESIOP DCE).

Pour être conforme à CORBA 2.0, un ORB doit supporter GIOP sur TCP/IP (ou s'y connecter via un pont). De même, un ORB doit créer un IOR (Interoperable Object References) lorsqu'une référence objet est transmise à travers plusieurs ORB. L'interopérabilité inter-ORB n'est donc pas garantie.

Exemple pratique :

Un Client, issu d'un programme écrit en C++, utilisant CORBA, ignore si un objet Serveur a été implémenté sous la forme d'un ensemble de classes C++ ou par un million de lignes de COBOL déjà existantes.

Composants :

Le bus CORBA fournit les composantes suivantes :

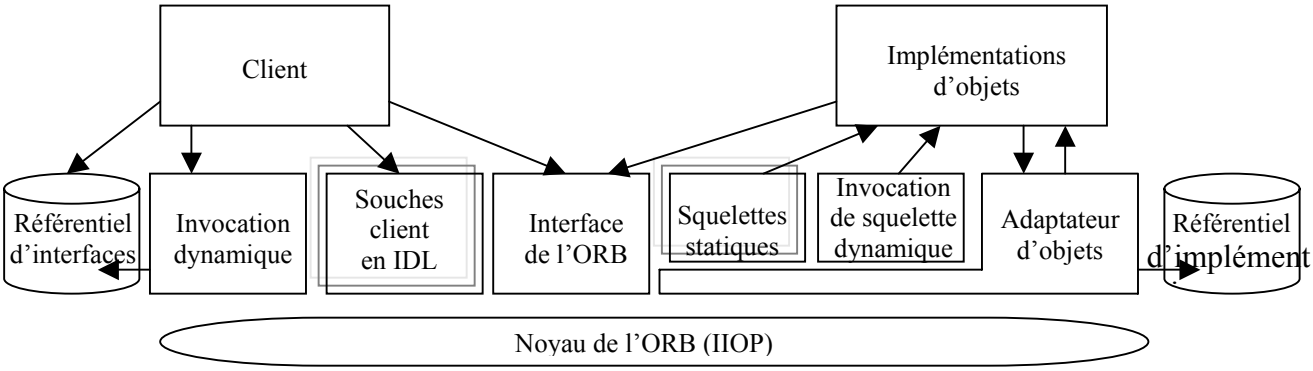
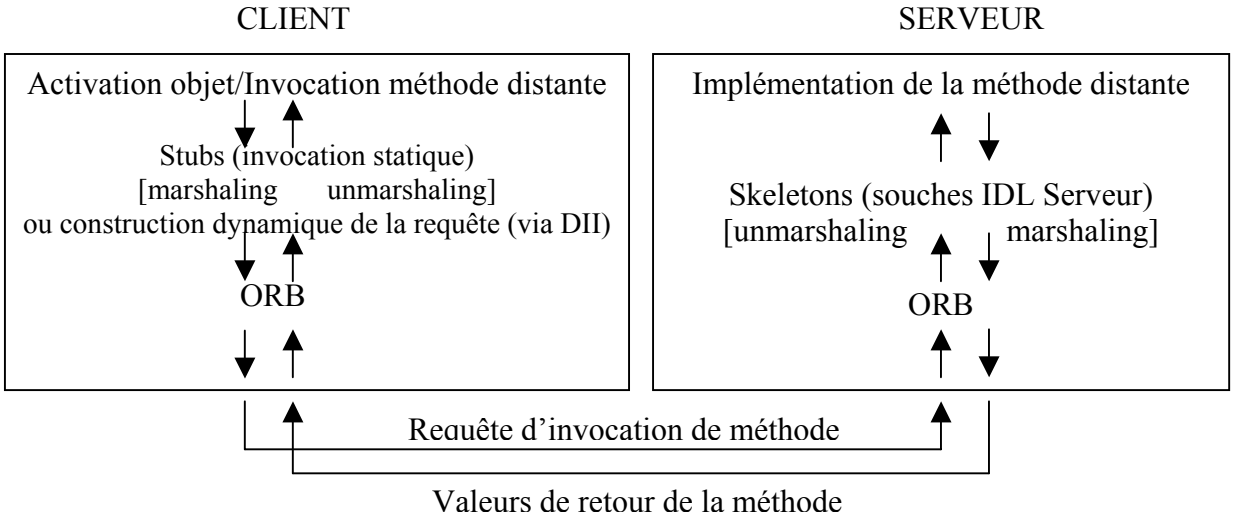


Fig. 5 : structure de l'ORB CORBA

L'ORB (Object Request Broker) permet la localisation des objets, le transport des requêtes vers un objet, le transport des résultats vers l'appelant, et intègre au minimum GIOP/IIOP. L'ORB est défini par ses interfaces.

On trouve un ORB de chaque côté Client et Serveur.



Constituants ORB côté Client :

Il appartient à l'ORB du Client de :

- ✓ trouver l'objet distant dans le système réparti,
- ✓ router les appels de la méthode distante vers le Serveur approprié,
- ✓ accepter les résultats en provenance du Serveur.

Il utilise pour cela les constituants suivants :

- **Static Invocation Interface** (SII) : souches (« stubs ») IDL Client : fournissent les interfaces d'invocations statiques (contrôlées à la compilation) aux services

objet. Cette interface est générée à partir de définitions OMG-IDL. Le stub joue le rôle de proxy (mandataire) local pour un objet Serveur à distance. Le stub inclut du code pour exécuter l'assemblage (« marshalling ») et le désassemblage (« unmarshalling ») de l'opération et de ses paramètres.

- **Dynamic Invocation Interface** (DII) : interfaces à appel dynamique : fournit les interfaces d'invocations dynamiques permettant de construire à l'exécution des requêtes vers un objet (découverte d'objets et de leur interface).
- **Interface Repository** (IR) : référentiel des interfaces contenant une représentation des interfaces OMG-IDL accessible par les applications durant l'exécution. C'est une base de données distribuée contenant la description des interfaces des composants, de leurs méthodes et paramètres (référentiel dynamique de métadonnées).
- ORB Core/ORB interface : fournit quelques API (Application Program Interface) aux services locaux (exemple : échange de références d'objet)

Constituants ORB côté Serveur :

Il appartient à l'ORB du Serveur de :

- ✓ permettre au Serveur d'enregistrer de nouveaux objets CORBA (service d'enregistrement d'objet conserve références sur objets répartis),
- ✓ accepter les requêtes de l'ORB Client pour invoquer la méthode distante sur le Serveur,
- ✓ transmettre les valeurs de retour au Client.

Il utilise pour cela les constituants suivants :

- **Skeleton Static Interface** (SSI) : squelettes (« skeleton ») IDL Serveur : fournissent les interfaces statiques à chaque service exporté par le Serveur. Cette interface est générée à partir de définitions OMG-IDL.
- **Dynamic Skeleton Interface** (DSI) : interface de Squelette Dynamique : permet d'intercepter dynamiquement toute requête pour des composants ne possédant pas d'interface SSI. C'est le pendant de DII pour un Serveur. Il peut recevoir des appels statiques ou dynamiques.
- **Object Adapter** (OA) : l'adaptateur d'objets instancie les objets Serveur et leur assigne des identificateurs d'objets (ID ou références d'objet). Il enregistre les classes et les objets dans le Référentiel d'implémentations (en particulier lors de l'activation de l'objet car, dans les requêtes suivantes, le BOA appelle la méthode appropriée en utilisant les skeletons). Chaque ORB doit supporter un adaptateur standard appelé Basic Object Adapter (BOA). CORBA 3.0 introduit une version portable du BOA appelé POA (Portable Object Adapter) qui permet de réaliser l'activation automatique si nécessaire.
- **Implementation Repository** : référentiel d'implémentations fournit des informations sur les classes que supporte un Serveur, les objets instanciés et leur ID (utilisé à l'exécution).
- ORB Core/ORB interface : fournit quelques API aux services locaux (identiques à celles fournies côté Client).

Ces différentes composantes sont toutes décrites dans le langage OMG-IDL, ce qui les rend accessibles au travers du bus.

Si l'ORB constitue la colonne vertébrale de cette architecture, alors le langage OMG-IDL fournit les articulations, c'est-à-dire tous les points de liaison entre le bus à objets et tous les éléments qui s'y rattachent.

2.2.2 COS (Common Object Services)

Les **services objet communs** (Common Object Services ou encore CORBAservices) fournissent les frameworks («canevas d'objets») de niveau système, nécessaires à la gestion d'objets CORBA, qui étendent les possibilités du bus.

L'OMG a défini des standards pour 16 services objet :

- ✓ Les annuaires : le service **Nommage (**« COS Naming »**)**, équivalent des « pages blanches », fournit un système de désignation pour retrouver des objets (« binding »), et le service **Vendeur** (« COS Trading »), équivalent des « pages jaunes » pour objets, leur permet de faire connaître et d'offrir leurs services : on accède aux objets par leurs propriétés et non plus par leur nom.
- ✓ Le service **Cycle de vie** : définit les opérations de création, copie, déplacement, suppression des objets sur le bus. Il recouvre la notion de fabrique d'objets ou « Object Factory ».
- ✓ Le service **Evénements** : il définit un objet appelé « canal d'événements » qui collecte des événements et les distribue à des composants de manière asynchrone via des objets anonymes.
- ✓ Le service **Transactions** : il fournit une coordination du type validation à deux phases (ensemble de traitements accomplis comme une seule opération).
- ✓ Le service **Relations** entre objets permet de créer des associations dynamiques entre des composants qui ignorent tout les uns des autres.
- ✓ Le service **Persistance** fournit une interface unique afin de stocker les composants de manière persistante sur des Serveurs de stockage (SGBDO, SGBDR ou simples fichiers).
- ✓ Le service **Propriétés** : permet d'associer dynamiquement des propriétés à des objets. Ne modifie pas l'interface IDL des objets.
- ✓ Le service **Sécurité** : fournit les fonctions systèmes permettant d'identifier et d'authentifier les Clients des objets, de certifier et de chiffrer les communications sur le bus.
- ✓ Le service **Contrôle de la concurrence d'accès** : système de gestion des accès concurrents à une ressource (mécanismes de verrou).
- ✓ Le service **Contrôle des licences** : fournit des mécanismes de base pour contrôler, mesurer et limiter l'utilisation des objets.
- ✓ Le service **Externalisation** : fournit un mécanisme standard pour fixer ou extraire l'état des objets du bus. Le déplacement et la sauvegarde reposent sur ce service
- ✓ Le service **Interrogations** : permet d'interroger les attributs des objets.
- ✓ Le service **Temps** : permet d'obtenir une horloge globale sur le bus.
- ✓ Le service **Collections** : utilisées conjointement avec le service d'interrogations.
- ✓ Le service **Changements** : permet de gérer et suivre l'évolution des différentes versions des objets.

Au fur et à mesure des besoins, l'OMG ajoute de nouveaux services communs.

2.2.3 Les utilitaires communs

Les utilitaires communs (CORBAfacilities) définissent les frameworks applicatifs horizontaux et verticaux directement utilisés par des objets métier. Ils fixent des règles pour que les composants métier coopèrent de façon efficace.

Exemple : Workflow (gestion des tâches), IHM (interface utilisateur), Administration du système, etc.

2.2.4 Les interfaces de domaine

Les interfaces de domaine (Domain interfaces ou encore CORBAdomains) définissent des objets métier spécifiques à des secteurs d'activités (marchés verticaux comme la finance, le commerce électronique, la santé, les autoroutes de l'information, etc).

Exemple : Telecom DTF (liaison CORBA/télécommunications) ou CORBAmed (interopérabilité entre acteurs de santé).

2.2.5 Les objets applicatifs

Les objets applicatifs (Application Objects) sont ceux qui sont spécifiques à une application répartie et ne sont donc pas standardisés (mais qui peuvent l'être par l'OMG en fonction de leur utilisation).

III. ÉVOLUTIVITÉ : ADA 95 ET LES NORMES CORBA

Après l'exposé des principes de la répartition au premier chapitre, nous avons étudié successivement deux cas pratiques de mise en œuvre de modèles de répartition : dans le langage Ada 95 (grâce à son annexe E) et dans l'intergiciel CORBA.

Nous avons vu suffisamment en détail le fonctionnement de chacune des plate-formes pour tenter une approche comparée entre les deux univers. L'étude d'outils réalisés dans le but de rapprocher Ada 95 et DSA, de CORBA, nous amènera à découvrir l'évolution ultime de travaux de recherche mené dans le sens de l'ouverture d'Ada 95 à l'interopérabilité : PolyORB, un « intergiciel schizophrène ».

A. Approche comparée entre ADA 95 (DSA) et CORBA

Le terme d' « approche comparée » se justifie ici dans la mesure où il ne s'agit pas d'une comparaison rigoureuse entre deux outils qui prétendent remplir les mêmes fonctionnalités. En effet, les objectifs qui ont conduit à l'élaboration de DSA et de CORBA divergent fortement :

- ✓ Le but de l'OMG est de produire un standard pour le développement d'applications réparties en environnements hétérogènes. CORBA spécifie le modèle d'architecture (OMA) basé sur des composants qui coopèrent via un réseau.
- ✓ L'annexe des systèmes répartis d'Ada 95 permet de développer des applications distribuées en respectant la rigueur sémantique imposée par le langage. Elle constitue une extension de la norme qui gomme les limites qui existent entre programmes centralisés et programmes répartis.

Si CORBA et Ada 95 ont en commun de respecter le modèle objet, notamment par la séparation stricte entre l'interface et l'implémentation des services, on peut constater qu'il existe de nombreuses divergences entre les deux approches.

1. PHILOSOPHIE

CORBA est entièrement organisé autour de la notion de répartition. Le modèle repose sur la spécification des interfaces : les objets répartis sont définis par leur interface (via IDL) et communiquent aisément par un bus à objets (ORB). Même les services CORBA sont des composants dont les interfaces ont été spécifiées mais leur implémentation ne l'est pas. Si le principe semble simple, la mise en pratique s'avère souvent lourde et complexe, prix à payer pour assurer le dialogue entre différents langages.

Inversement, l'annexe des systèmes répartis d'Ada a pour but de supprimer la barrière existant entre les programmes centralisés et les programmes répartis. DSA est d'un abord abscons : le manuel de référence lui-même insiste sur les restrictions et les contraintes imposées pour le développement d'applications réparties (rigueur sémantique). Par contre, à l'usage, le langage permet de passer très facilement (pragmas) de la version monolithique d'une application à sa version distribuée. C'est le partitionnement, processus post-compilatoire, qui assure la répartition entre les nœuds du réseau.

2. DÉCLARATIONS DES SERVICES

Avec CORBA, la déclaration des objets et de leurs méthodes se fait au moyen de l'OMG-IDL, langage purement déclaratif. Pour obtenir une référence sur un objet distant, il faut faire appel explicitement à un service de nommage que le fournisseur de l'outil CORBA utilisé aura pris soin d'implémenter.

Ada est son propre IDL : la structure du paquetage prévoit la déclaration des services. Tout est intégré dans le langage. Les fonctions d'annuaire sont prises en charge automatiquement par le système.

3. MODÈLES DE RÉPARTITION

CORBA est uniquement orienté objet. Il permet de réaliser des systèmes à objets répartis où le composant-objet est l'unité de répartition. Les files d'attente de messages sont aussi possibles.

Les modèles supportés par Ada dérivent des abstractions fournies par le langage. En effet, les directives de compilation autorisent l'appel de sous-programmes, l'invocation de méthodes à distance, et un modèle original de partage d'informations, de mémoire et d'objets à distance (DSM).

4. DÉVELOPPEMENT DES APPLICATIONS

Le premier travail avec CORBA, après avoir conçu l'application pour une utilisation distribuée, est d'écrire le contrat IDL. Le code généré pour les langages supportés (stubs et skeletons) s'impose au développeur qui doit l'intégrer à son propre code (et éventuellement l'adapter) du côté Client et du côté Serveur.

Avec DSA, l'interface et l'implémentation sont réalisées en Ada, ce qui simplifie le développement et ne nécessite pas de conversion entre types, ni l'intégration d'un code externe généré pour le Client ou le Serveur. Dans un premier temps, on utilise une version centralisée de l'application (qui contient les pragmas adéquats) pour des facilités de mise au point. Ensuite, on l'étendra au mode distribué par l'opération de partitionnement. La répartition est indépendante du code source.

5. PORTABILITÉ DES APPLICATIONS

CORBA spécifie les interfaces et laisse les implémentations des spécifications fonctionnelles à l'initiative des fabricants qui peuvent librement interpréter des directives pas toujours explicites. Il en résulte de nombreux produits ou services « conformes CORBA » mais néanmoins spécifiques. Cela tend à prouver un échec de CORBA en tant que Middleware standard car la portabilité n'est pas garantie. Le développeur se trouve lié à une solution particulière : on développe une application Orbix (fabriqué par Iona) plutôt qu'une application CORBA.

Avec DSA, la portabilité est garantie étant donné que la configuration d'une application répartie se fait en dehors du code lui-même. Le partitionnement est un processus post-compilatoire et paramétrable. L'annexe précise aussi les interfaces avec le sous-système de communication (PCS) pour pouvoir passer d'une implémentation de la couche communication, à une autre (pas toujours suffisant en pratique).

6. HÉTÉROGÉNÉITÉ ET INTEROPÉRABILITÉ

Dans CORBA, c'est l'ORB qui permet l'interopérabilité entre des composants d'applications réparties dans des environnements distribués hétérogènes. Il utilise pour cela une représentation commune des données (CDR : Common Data Representation), et un ensemble de messages (GIOP : General Inter-ORB Protocol).

Le manuel de référence d'Ada n'impose même pas l'utilisation de DSA pour que des services fonctionnent correctement sur différentes plate-formes (si ils ont été écrits en Ada 95). A l'inverse, l'interopérabilité avec d'autres langages distribués ou des Middleware de répartition n'est pas prévue par Ada 95, pourvu ou non de son annexe des systèmes répartis.

7. SOUS-SYSTÈME DE COMMUNICATION

Elément clé d'un système réparti, le sous-système de communication a la capacité de faire communiquer les nœuds du réseau. Que ce soit dans CORBA (ORB) ou avec Ada (PCS), cet élément assure au minimum le transport des requêtes vers un objet et celui des résultats vers l'appelant.

Là encore, une fonctionnalité remplie par un service dans CORBA, sera souvent considérée comme interne à DSA (paquetage « System.RPC ») :

- ✓ la localisation des objets : dans CORBA, le service de nommage est lui-même un objet réparti avec une interface IDL standard. Avec Ada 95 et DSA, la localisation d'un service est à la charge du PCS. En règle générale, tout ce qui n'est pas fait par le compilateur Ada vis à vis de la répartition, doit être assuré par le PCS.
- ✓ la remontée d'exceptions : un ORB est incapable de propager une exception qui n'a pas été clairement définie dans l'IDL. Avec le PCS, une exception, même inconnue (clause « when others »), sera remontée jusqu'à l'appelant en « traversant » plusieurs partitions si besoin.
- ✓ autres différences : un ORB doit intégrer au minimum GIOP/IIOP. PCS doit détecter l'avortement d'appels distants.

8. COS (COMMON OBJECT SERVICES)

L'OMG a spécifié un ensemble de services objet communs (COS) qui standardisent des composants utiles dans le cadre de systèmes répartis (services d'annuaire, événements, contrôle des licences, cycle de vie, etc). L'interface de ces services est standardisée au moyen de l'IDL, mais leur implémentation est laissée à l'initiative du fournisseur d'ORB.

DSA ne met pas à la disposition du développeur un tel éventail de services (même si certains, comme le service de nommage, sont inclus en natif dans Ada). Pourtant, Ada inclus en son sein des fonctionnalités absentes de CORBA comme l'avortement d'appels à distance, la détection de terminaison distante, la vérification de la consistance des versions (« versionning »), la propagation des exceptions. Mais ces services internes n'ont pas évolués depuis la normalisation du langage.

L'absence de bibliothèques de composants supplémentaires a toujours fait défaut à Ada et à son annexe, qui n'autorise pas l'interopérabilité avec d'autres systèmes.

B. Adaptations de Ada 95 (DSA) à la norme CORBA

Conscients des déficiences du duo Ada 95/DSA, des élèves de l'ENST (Ecole Nationale Supérieure de Télécommunications), et notamment messieurs Pautet, Quinot et Tardieu, ont mené des recherches et créé des outils afin de faire converger l'annexe des systèmes répartis vers la norme CORBA en adaptant certains services CORBA à DSA. Ensuite, nous verrons que l'interopérabilité entre les deux systèmes a été réalisée par un mécanisme de passerelle (CIAO).

1. GLADE (1995 À 1999)

GLADE constitue une mise en oeuvre de l'annexe des systèmes répartis d'Ada 95 pour l'environnement de compilation libre GNAT (compilateur Ada 95 distribué sous licence GNU par Ada Core Technologies). De plus, GLADE fait de GNAT le seul compilateur conforme à la norme sur l'ensemble des annexes.

Son objectif est double : enrichir DSA avec des services fournis en standard par CORBA, et offrir un pré-compilateur et une implémentation de l'ORB sous forme de logiciel gratuit et open-source.

1.1. Adaptation de services CORBA à DSA

Un des points forts de CORBA réside dans l'apport d'un ensemble de services objet communs (COS) qui fournissent des fonctionnalités très utiles pour le développement d'applications réparties (services d'annuaire, événements, contrôle des licences, cycle de vie, etc).

Implémentation de certains services communs CORBA dans DSA

Le premier objectif de GLADE a été d'adapter certains de ces services pour compléter DSA :

- ✓ Services d'annuaire et d'événements : ces deux services ont été développés conformément aux spécifications de l'OMG (reproduction stricte en Ada du service existant dans CORBA). Les auteurs ont pris conscience que si l'interface était clairement définie dans un fichier IDL, les spécifications fonctionnelles restaient floues et le comportement était à définir par celui qui implémente le service (d'où les problèmes de portabilité déjà évoqués).
  En conséquence, les autres services implémentés l'ont été directement sous la forme d'unités Ada avec toute la rigueur sémantique associée.

- ✓ Service contrôle de la concurrence d'accès : le service est basé sur une version distribuée de l'algorithme d'exclusion mutuelle (Mutex).

- ✓ Service d'invocation dynamique : dans CORBA, les éléments Interface Repository (IR) et Dynamic Invocation Interface (DII) permettent de construire à l'exécution des requêtes vers un objet (découverte dynamique d'objets et de leur interface).

  DSA n'offre pas en standard cette possibilité. Pour disposer du service d'invocation dynamique, il faut d'abord s'adjoindre les services de l'Interface Repository. En Ada, un paquetage de type Remote_Call_Interface joue le rôle de Serveur DSA : les fournisseurs de services (autres paquetages de types RCI ou RT) enregistrent leur interface et les Clients découvrent les méthodes d'un objet.

1.2. Vers un ORB Ada : le projet AdaBroker

La norme CORBA a prévu que des méthodes Ada spécifiées dans l'IDL puissent être invoquées par un Client dans un langage capable de fournir des liens CORBA (C, C++, SmallTalk, Ada, COBOL et Java). Pour cela, chaque spécification IDL est traduite dans un langage d'implémentation (projection) au moyen d'un pré-compilateur IDL

Des pré-compilateurs et des implémentations de l'ORB se trouvent aisément en version libre pour des langages comme C ou C++. Les mêmes outils n'existaient pas en version open-source pour Ada : le projet AdaBroker était né.

Implémentation d'un ORB CORBA, gratuit et open-source, pour Ada

Basé sur un ORB existant, omniORB (conçu pour le C++), AdaBroker, conforme aux règles de projection normalisées par l'OMG dans le langage Ada, fournit un ensemble d'outils et de bibliothèques pour développer des applications CORBA en Ada. AdaBroker permet de générer, via un pré-compilateur, des souches et des squelettes en Ada, et fournit un ensemble de paquetages pour s'interfacer avec un ORB.

AdaBroker a permis de dépasser des contraintes liées à l'usage de omniORB et est pourvu des fonctionnalités avancées comme POA (Portable Object Adapter).

1.3. Partitionnement

L'annexe ne décrit pas comment une application répartie doit être configurée.

L'opération de partitionnement est à la charge du développeur (gestion des partitions et de leur répartition sur le réseau).

Réalisation d'outils d'aide à la répartition

GLADE comprend un outil de configuration et un sous-système de communication entre partitions :

- ✓ GNATDIST: outil de déploiement d'une application (pas de spécifications de l'annexe sur le partitionnement) qui permet à l'utilisateur de créer des partitions et indiquer sur quel nœud ces partitions s'exécuteront.
- ✓ GARLIC (Generic Ada Reusable Library for Interpartition Communication) est une bibliothèque de communication de haut niveau qui implémente l'interface entre le PCS et la couche de communication réseau.

GLADE : des avancées mais pas d'ouverture vers d'autres systèmes

Ces améliorations, même si elles constituent une avancée indéniable vers le standard CORBA, ne permettent pas de faire interagir des composants autres que ceux écrits en Ada.

L'objectif suivant est d'améliorer l'interopérabilité entre des utilisateurs de CORBA et des composants écrits en Ada. Bref, de permettre à des Clients CORBA d'accéder à des services DSA.

2. CIAO : CORBA Interface for Ada 95 distributed Objects (1999/2000)

Pour développer un service en profitant de la sûreté du langage Ada 95 et de la facilité d'utilisation de DSA tout en bénéficiant de l'ouverture à des Clients hétérogènes offerte par CORBA, il faut faire interagir des composants utilisant les deux environnements (DSA et CORBA), afin de cumuler leurs avantages.

CIAO : Services DSA pour Clients CORBA

Dans le projet CIAO (CORBA Interface for Ada 95 distributed Objects), un mécanisme de génération automatique d'interfaces IDL, puis de passerelles permet à des Clients CORBA d'accéder à des services définis en utilisant l'annexe des systèmes répartis d'Ada 95 (DSA).

CIAO s'appuie sur les projets antérieurs GLADE et AdaBroker.

2.1. Génération automatique d'interface IDL

Le développeur d'un Client CORBA, qui veut faire appel à un service écrit en Ada, a besoin de la description écrite en OMG-IDL de ce service afin de générer le stub correspondant.

La spécification de l'interface IDL peut être générée automatiquement à partir des déclarations contenues dans les paquetages DSA. A cette fin, un schéma formel de traduction d'une spécification de paquetage DSA vers une spécification OMG-IDL a du être établi.

Traduction de spécifications de services DSA au format OMG-IDL

Un outil, nommé « Traducteur CIAO », se base sur ASIS (Ada Semantic Interface Specification) - une interface de programmation entre l'environnement de compilation Ada et toute demande d'informations sur celui-ci – pour extraire l'arbre sémantique d'une déclaration de paquetage Ada du compilateur GNAT.

L'analyse de cet arbre permet d'effectuer les traductions utiles tout en respectant la structure des services :

- ✓ Un paquetage de bibliothèque = un fichier source IDL.
- ✓ Un paquetage Ada = un module IDL.
- ✓ Un paquetage « Remote_Types » (type objet réparti) = une interface et une primitive du type objet = une opération de cette interface.
- ✓ Un paquetage « Remote_Call_Interface » (sous-programmes appelables à distance) = une interface.
- ✓ Etc… chaque construction étant traduite par son équivalent IDL.

2.2. Transformation des requêtes CORBA en invocation de méthodes Ada

Pour que des Clients CORBA puissent faire appel à des services DSA, il est nécessaire de traduire les requêtes CORBA en invocation de méthodes DSA. Cette traduction est assurée, à l'exécution, par des composants applicatifs, appelés passerelles, qui se comportent à la fois comme un Serveur de requêtes CORBA, et à la fois comme un Client vis à vis de l'annexe des systèmes répartis d'Ada 95.

Interopérabilité par génération de passerelles

Un outil, nommé « Générateur CIAO », utilise le contrat IDL généré par le « Traducteur CIAO » et l'arbre sémantique généré par ASIS, pour générer automatiquement d'une part un squelette Serveur CORBA en Ada 95 (via AdaBroker), et d'autre part, l'implémentation de ce même Serveur CORBA.

Au final, par le jeu des générations automatiques, nous obtiendrons pour chaque paquetage DSA défini, un paquetage squelette CORBA et un paquetage correspondant à l'implémentation des interfaces contenues dans le squelette : c'est ce « paquetage-relais » (il relaye les références d'objets) qui permet l'invocation de la primitive de l'objet DSA (Client DSA) et la conversion des types de données CORBA en types Ada et inversement.

La passerelle est une mise en œuvre particulière du contrat IDL, dans laquelle le comportement associé à chaque méthode consiste à émettre la requête correspondante sur l'objet Ada 95.

2.3. Inconvénients du modèle CIAO

2.3.1 Les passerelles sont unidirectionnelles

Les passerelles générées ne permettent l'interaction que dans un seul sens, entre des Clients CORBA et des Serveurs DSA uniquement. Elles ne permettent pas à des Clients DSA d'accéder aux services d'un objet CORBA. En particulier, tout mécanisme de rappel automatique (callback) entre objets est impossible.

2.3.2 Difficulté de conversion de types

Le schéma formel de traduction d'une spécification de paquetage DSA vers une spécification OMG-IDL impose des simplifications qui sont incapables de conserver fidèlement la sémantique des services DSA.

2.3.3 Coût de l'utilisation de passerelles

Le composant passerelle constitue un point de passage obligé pour toute requête émise par un objet CORBA à destination d'un objet DSA.

Nous avons vu que chaque paquetage DSA se traduit par la génération d'un paquetage-relais supplémentaire, qu'il ne faut d'ailleurs pas oublier d'intégrer au moment du partitionnement.

Une passerelle introduit un coût supplémentaire (réception de la requête, conversion des types CORBA vers les types natifs Ada 95, envoi vers l'objet destinataire réel) lors de chaque appel de méthode dû à l'indirection qui en découle qui pénalise les performances.

De plus, la passerelle est un point de faiblesse de l'architecture : la défaillance de ce seul composant entraîne une indisponibilité de l'application.

C. Perspectives d'évolutions : PolyORB, un intergiciel schizophrène

CIAO, générateur de passerelles statiques, permet l'interopérabilité entre l'annexe des systèmes répartis d'Ada 95 et les objets répartis CORBA (service DSA pour Client CORBA). Cependant, cette approche de l'interopérabilité entre intergiciels, par génération de passerelles, n'est pas satisfaisante ni généralisable pour les raisons évoquées ci-dessus.

Aux incompatibilités entre modèles de répartition, (ou pour un même modèle, entre les différentes mises en œuvre), s'ajoutent celles entre intergiciels (abritent la mise en œuvre d'un ou plusieurs modèles). Pourtant, avec le développement du modèle d'application répartie, il paraît primordial de pouvoir faire communiquer des composants quel que soit le modèle de répartition utilisé par chacun.

La question se pose donc de la mise en relation d'objets utilisant des modèles de répartition et des intergiciels différents.

Une possibilité est d'utiliser des passerelles entre modèles de répartition et entre intergiciels. Dans ce cadre, le projet CIAO a montré les limites de la génération de passerelles statiques.

Une autre solution est de réaliser un intergiciel capable d'offrir simultanément plusieurs modèles de répartition.

C'est cette dernière solution qui a conduit à la réalisation de PolyORB.

PolyORB : un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables

Une trentaine de développeurs ont participé à l'élaboration de PolyORB, intergiciel qui a fournit le sujet de la thèse de doctorat de Thomas Quinot soutenue en mars 2003 : « Conception et réalisation d'un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables ».

Nous allons étudier ce qui se cache sous ce terme « d'intergiciel schizophrène », puis nous verrons les services qu'il fournit.

1. NOTION D'INTERGICIEL SCHIZOPHRÈNE

1.1. Trois objectifs à satisfaire

Après l'expérience CIAO, le même groupe de travail (une trentaine d'étudiants, chercheurs, universitaires, et Thomas Quinot de ACT) a poursuivi ses recherches dans le but de tirer la « substantifique moëlle » de chacune des deux plate-formes, Ada 95-DSA et CORBA.

Une de leurs études a montré que les intergiciels traditionnels sont liés au choix d'un modèle de répartition et que deux mises en œuvre du même modèle ne sont pas

nécessairement compatibles. Outre les problèmes d'incompatibilités nés de ce constat, il est apparu évident qu'un unique intergiciel traditionnel, au comportement et aux fonctionnalités fixés une fois pour toutes, ne peut répondre à la diversité des besoins des applications (et à leur évolution), et à la diversité des ressources disponibles sur les différents nœuds.

Cette étude a aussi permis d'établir que pour être en mesure de répondre aux besoins d'applications réparties avancées, faisant intervenir plusieurs modèles de répartition, un intergiciel doit présenter trois aspects essentiels : l'interopérabilité, la généricité et la configurabilité. Le but final étant de définir un intergiciel capable d'adapter son comportement et ses fonctionnalités à des besoins spécifiques.

1.1.1 L'interopérabilité

L'utilisation d'intergiciels permet de s'affranchir de l'hétérogénéité des architectures matérielles et logicielles, mais les incompatibilités entre modèles de répartition et entre intergiciels font apparaître de nouvelles contraintes sur les possibilités d'interactions. Il apparaît donc nécessaire de favoriser l'interopérabilité, au sein d'une même application, entre des nœuds utilisant des modèles de répartition et des intergiciels différents. Cette propriété d'interopérabilité ne peut être réalisée au moyen d'intergiciels traditionnels (passerelles) sans souffrir d'une explosion combinatoire rédhibitoire (Cf. CIAO).

PolyORB intégre l'interopérabilité directement en son sein (au lieu de l'appliquer comme une surcouche d'un système existant) pour permettre à des composants applicatifs d'interagir avec des entités réparties issues de différents modèles de répartition.

1.1.2 La généricité

Les intergiciels génériques sont fondés sur l'identification de fonctionnalités récurrentes. Ces fonctionnalités sont alors factorisées sous forme de modules génériques (c'est-à-dire indépendant d'un modèle de répartition ou d'un protocole particulier) et qui sont réutilisés.

Les architectures existantes d'intergiciels génériques (Jonathan et Quaterware font l'objet d'une étude dans [P4]) introduisent la **notion de personnalité** et considèrent que chaque variante d'intergiciel est une vue différente des mêmes fonctions fondamentales.

Une personnalité d'intergiciel est un ensemble de vues sur les fonctions assurées par l'intergiciel. Elle associe :

- ✓ un modèle de répartition,
- ✓ une interface exposée aux composants applicatifs (interface de programmation permettant de manipuler les abstractions du modèle de répartition),
- ✓ une interface exposée aux autres intergiciels (protocoles permettant de mettre en œuvre ces abstractions).

PolyORB prolonge cette démarche en permettant à différentes personnalités de cohabiter dans un même intergiciel, et d'utiliser simultanément une mise en œuvre générique mutualisée des fonctions de répartition.

PolyORB est générique car il permet la réutilisabilité des modules entre les différentes personnalités.

1.1.3 Configurabilité

Pour pouvoir répondre à la diversité des besoins de chaque application et des contraintes imposées par l'environnement de chaque nœud, l'intergiciel doit être configurable. La configuration permet la prise en compte des contraintes spécifiques de l'application et de l'environnement d'exécution, et de n'intégrer que les modules essentiels. Par exemple, dans le cas d'un système embarqué temps réel (ressources disponibles réduites), les fonctionnalités de l'intergiciel doivent être limitées à l'essentiel, et ne pas

introduire de fonctionnalités telles que le parallélisme et les synchronisations entre tâches. On parle alors de profil de fonctionnalités.

Un **profil de fonctionnalités** d'un intergiciel est constitué d'un sous-ensemble des fonctionnalités qui peuvent être instanciées sur un nœud, en excluant celles qui n'y sont pas nécessaires.

PolyORB est configurable car il permet d'apdapter les paramètres de l'intergiciel aux besoins de l'application répartie.

1.2. **Personnalités multiples**

1.2.1 Découplage des personnalités

Pour satisfaire les besoins d'interopérabilité entre modèles de répartition, il est nécessaire de briser le couplage entre les deux interfaces de l'intergiciel :

- ✓ l'interface applicative : les composants réalisant l'interface entre les objets applicatifs locaux et l'intergiciel n'effectuent aucun traitement, ne manipulent aucune représentation, qui soit spécifique à un protocole particulier.
- ✓ l'interface protocolaire (exposée aux intergiciels tiers) : les composants réalisant un protocole ne font aucune hypothèse particulière sur le choix d'une interface entre objets de l'application et intergiciel.

Ce point constitue une extension des architectures génériques existantes en introduisant la distinction entre les personnalités applicatives et les personnalités protocolaires.

1.2.2 Personnalité applicative

« Une personnalité applicative est la spécification d'une interface entre des objets applicatifs et un intergiciel en vue d'en utiliser les services pour interagir avec d'autres objets applicatifs. » (extrait de [P4])

1.2.3 Personnalité protocolaire

« Une personnalité protocolaire est la spécification d'une interface entre deux intergiciels destinée à l'échange de messages représentant les interactions entre les objets hébergés par ces intergiciels. » (extrait de [P4])

1.3. **Définition d'un intergiciel schizophrène**

« Nous qualifions de schizophrène, un intergiciel capable de disposer, simultanément, de plusieurs personnalités applicatives et protocolaires et de les faire interagir efficacement. » (extrait de [P4])

2. ARCHITECTURE SCHIZOPHRÈNE

2.1. Schéma général

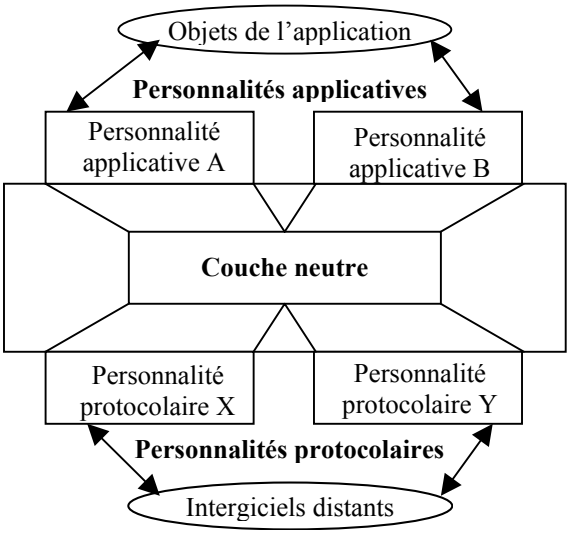


Fig.6 : schéma général de l'architecture schizophrène (extrait de [P4])

2.2. Découplage des personnalités

Les intergiciels traditionnels ne supportent en général qu'une seule combinaison entre personnalité applicative et protocolaire : les deux personnalités sont fortement couplées.

La spécificité de l'architecture d'un intergiciel schizophrène tient dans le découplage entre les deux des personnalités (applicatives et protocolaires).

2.3. Introduction d'une couche neutre

Les personnalités applicatives et protocolaires sont organisées autour d'une couche neutre mettant en œuvre les fonctions récurrentes des intergiciels. Elle offre les représentations et les services nécessaires pour effectuer la mise en correspondance transparente entre personnalités applicatives et protocolaires : elle est la clé de l'interopérabilité.

Un objet basé sur une personnalité applicative donnée peut utiliser un protocole quelconque pour interagir avec d'autres objets. Par exemple, un objet créé en utilisant la personnalité applicative de l'annexe des systèmes répartis d'Ada 95 peut communiquer aussi bien avec des objets CORBA utilisant le protocole GIOP qu'avec un Serveur utilisant le protocole SOAP (Simple Object Access Protocol).

Réciproquement, un même protocole peut être associé à des objets utilisant des personnalités applicatives différentes. Par exemple, un objet dont on souhaite qu'il soit accessible à travers GIOP pourra être développé selon le modèle de CORBA, en utilisant un contrat IDL et un générateur de souches et de squelettes, ou bien au moyen de l'annexe des systèmes répartis du langage Ada 95.

3. COMPOSANTS ACTUELS ET FUTURS

3.1. Services génériques

Dans le premier chapitre, nous avions étudié les fonctions fondamentales qu'un intergiciel digne de ce nom doit remplir : Adressage, Transport, Protocole, Représentation, Liaison, Activation, Exécution (Cf. I-B-2.3).

C'est la couche neutre qui va mettre en œuvre les fonctions génériques récurrentes (comportements communs aux différentes personnalités applicatives et protocolaires) des intergiciels. Ainsi, elle autorise la mise en place de personnalités multiples, et elle assure leur cohabitation dans une même instance d'intergiciel pour en assurer l'interopérabilité. C'est aussi ce qui permet à PolyORB de communiquer avec des intergiciels qui implémentent différents modèles de répartition : PolyORB autorise le Middleware to Middleware (« M2M »).

Il ne faut pas oublier que cet intergiciel a été développé principalement en Ada 95 et que grâce au mécanisme des paquetages Ada, la modularité et à la conception orientée objet, il a été possible de réutiliser de nombreuses bibliothèques issues des projets précédents : AdaBroker et GLADE. Cependant, certains éléments ont du être modifiés, comme par exemple le compilateur GNAT pour pouvoir produire des souches et des squelettes dont la structure soit adaptée à PolyORB. Ou encore modifier la bibliothèque de communication pour adapter le code généré à l'interface de la couche neutre : l'un des objectifs de PolyORB est de pouvoir être utilisé comme moteur de communication dans le cadre de la mise en œuvre de l'annexe des systèmes répartis d'Ada 95. Enfin, un nouveau compilateur IDL (idlac) a vu le jour.

En conséquence, PolyORB, intergiciel conforme à CORBA, inclus les constituants ORB suivants :

- ✓ Services d'annuaire et d'événements (COS Naming et COS Events)
- ✓ Service d'invocation dynamique : Interface Repository (IR), Dynamic Invocation Interface (DII), et Dynamic Skeleton Interface (DSI)
- ✓ Object Adapter : Portable Object Adapter (POA)
- ✓ GIOP (General Inter-ORB Protocol) 1.0 à 1.2
- ✓ Support des échanges asynchrones de type MOM (Message-Oriented Middleware)

3.2. Personnalités

3.2.1 Personnalités applicatives

Des personnalités applicatives issues de divers modèles de répartition ont été implémentées :

- ✓ une personnalité CORBA, basée sur le modèle « objets répartis »,
- ✓ une personnalité mettant en œuvre l'annexe des systèmes répartis du langage Ada 95, offrant les modèles « objets répartis » et « appel de sous-programmes à distance »,
- ✓ une personnalité de passage de messages (MOM, Message-Oriented Middleware), nommée MOMA (MOM for Ada).

3.2.2 Personnalités protocolaires

Deux personnalités protocolaires sont disponibles :

- ✓ GIOP, protocole associé à la norme CORBA;
- ✓ SOAP (Simple Object Access Protocol), protocole proposé pour les services Web (fonctionne sur HTTP, SMTP et FTP).

3.3. **Travaux en cours**

PolyORB était un prototype au moment de la thèse de T. Quinot. C'est maintenant un produit industriel commercialisé et supporté par ACT Europe.

Des améliorations sont prévues :

- ✓ Optimisation des performances, notamment de la couche neutre
- ✓ Nouvelles personnalités applicatives (personnalité services Web basée sur l'interface de programmation AWS (Ada Web Server)) et protocolaires (personnalité HTTP).
- ✓ Configuration temps réel : la configurabilité permet l'utilisation de PolyORB dans les systèmes embarqués. De nouveaux services augmenteront encore son intérêt pour les applications de ce type.
- ✓ Aide au déploiement d'applications réparties : adaptation de GNATDIST, outil de partitionnement inclus dans GLADE
- ✓ Autres services génériques : service d'espace de stockage partagé qui permettrait d'intégrer le modèle de répartition « mémoire partagée répartie » dans PolyORB.

4. RÉSUMÉ : POLYORB VIS À VIS DE CORBA

PolyORB est un intergiciel. CORBA est un modèle de répartition avant tout. Donc, on ne peut pas vraiment les comparer.

PolyORB est poly…ORB

PolyORB implémente CORBA et aussi d'autres modèles de répartition (DSA, MOM, et bientôt AWS). Donc, PolyORB ne remplit pas le rôle d'un seul ORB mais bien de plusieurs.

PolyORB est poly…morphe

PolyORB constitue une nouvelle architecture d'intergiciel, qui est simultanément :

- ✓ interopérable : pour permettre la coexistence de multiples modèles de répartition au sein de la même instance d'intergiciel,
- ✓ générique : pour factoriser le code indépendant des personnalités et prévenir l'explosion combinatoire en découplant les différents aspects de personnalité,
- ✓ configurable : pour adapter et optimiser les composants de l'intergiciel par rapport aux contraintes des applications, et limiter la lourdeur de mise en œuvre liée aux aspects d'interopérabilité et de généricité.

PolyORB est poly…céphale…

Les modèles de répartition sont nombreux et en raison de composants hérités de projets exitants, le développeur d'applications réparties peut être amené à assembler une application à partir de composants de modèles hétérogènes. Le découplage entre les personnalités applicatives et les personnalités protocolaires prend ici toute sa valeur.

…donc schizophrène

PolyORB élimine l'hétérogénéité et autorise l'interopérabilité grâce à son caractère schizophrène (qui est la caractéristique majeure de PolyORB).

PolyORB est poly…valent

La conception modulaire de PolyORB en fait un intergiciel flexible avec des qualités intrinsèques d'extensibilité et de réutilisabilité.

Son caractère configurable (gestion de profil de fonctionnalités, gestion du « Ravenscar profile ») lié à la fiabilité du code qui le compose (Ada 95), font de lui un candidat privilégié pour les applications temps réel ou embarquées (certains de ces composants ont été modélisés par réseaux de Pétri et certaines propriétés vérifiées).

Mais son domaine d'application n'a pas de limites.

CONCLUSION

Au cours de cette étude, nous avons pu observer deux approches différentes destinées à pallier les problèmes nés de la répartition. La répartition constitue une source de complexité supplémentaire pour les développeurs d'applications de ce type.

Des modèles de répartition permettent de faciliter la conception et le développement d'applications réparties. Après l'étude théorique de ces modèles, nous avons montré comment ils étaient implémentés dans le cadre d'un langage distribué (Ada 95 et DSA) d'une part, et d'autre part, dans le cadre d'un intergiciel qui fait référence en la matière : CORBA.

Ada 95, premier langage objet à avoir été normalisé, offre la possibilité de développer aisément et sûrement des applications réparties grâce aux qualités intrinsèques du langage et à son annexe des systèmes répartis. Sur la base d'une mise en œuvre de l'annexe des systèmes répartis d'Ada 95, plusieurs outils ont été réalisés (GLADE) dans le but de simplifier le développement d'applications réparties et intégrer dans DSA, certains services standards de CORBA. Un générateur de passerelles (CIAO) a même permis à un Client CORBA d'utiliser des services écrits en Ada. Ces évolutions successives de l'environnement Ada 95/DSA n'ont pas permis d'obtenir une solution satisfaisante pour résoudre le problème majeur de DSA : l'absence d'interopérabilité avec d'autres systèmes.

De son côté, CORBA constitue une solution ouverte et évolutive (quoique complexe à mettre en œuvre), et autorise l'interopérabilité entre des composants d'applications réparties dans des environnements distribués hétérogènes. Cependant, si CORBA spécifie les interfaces, l'implémentation des services est laissée à l'initiative du fournisseur d'ORB : la portabilité n'est pas garantie.

De plus, si l'utilisation d'intergiciels rend les applications réparties insensibles à l'hétérogénéité des environnements matériel et logiciel, elle introduit de nouvelles incompatibilités entre les différents modèles de répartition, voire même pour un même modèle (selon l'implémentation qui en faite). Cette situation constitue « le paradoxe de l'intergiciel » (Laurent Pautet).

Ce constat, joint aux différents travaux menés précédemment dans le sens de l'ouverture d'Ada 95 à l'interopérabilité, a donné naissance à un nouveau type d'intergiciel : « l'intergiciel schizophrène » PolyORB.

PolyORB répond aux trois objectifs d'interopérabilité, de généricité et de configurabilité. Son architecture originale permet la distinction entre les « personnalités applicatives » et les « personnalités protocolaires », d'où son aspect schizophrène ». Une « couche neutre » assure le fonctionnement des services récurrents des intergiciels de façon générique.

Dans le cadre de cette infrastructure, réalisée principalement en Ada 95, des modèles de répartition comme DSA ou CORBA peuvent être considérés comme des instances particulières de cette architecture.

PolyORB atteint un niveau d'abstraction supérieur à ceux de ces concurrents (même s'il est difficile de comparer un modèle de répartition - CORBA – à un intergiciel plus généraliste – PolyORB).

Sur le papier, il semble bien que l'élève ait dépassé le maître. Cependant, PolyORB vient à peine de dépasser le stade de prototype et il lui reste à « faire ses gammes ».

Pour poursuivre sur l'analogie musicale, je dirai que ce n'est pas parce qu'un élève dépasse techniquement son maître que le succès lui est garanti pour autant. En effet, la promotion de l'artiste

est un passage obligé pour se faire connaître du plus grand nombre et pouvoir jouer dans les plus grandes salles de concert.

Dans ce cadre, le support fourni à CORBA par l'OMG, qui est à la fois juge et partie pour l'établissement de la norme, et qui bénéficie de l'aide de tous les principaux acteurs du marché, est énorme. De plus, la dernière version de CORBA (3.0) permet une compatibilité encore plus grande avec les principales solutions de développement d'applications réparties : Java/RMI sur IIOP, modèles de composants Entreprise JavaBeans (EJB), interopérabilité avec DCOM de Microsoft, etc.

Concernant Ada 95, on peut regretter le peu d'efforts faits pour la diffusion de ce langage standardisé comme langage objet trois ans avant le C++, notamment au niveau de son enseignement (j'exclus bien sûr le CNAM de ce constat !). On s'aperçoit que son utilisation est encore prisée dans les domaines « sensibles », pour des systèmes temps réel ou embarqué, où sa fiabilité et sa pérennité font merveille : d'Ariane au TGV en passant par l'Airbus.

La communauté Ada est réduite mais de qualité : la réalisation de PolyORB en est un exemple manifeste. Même si PolyORB doit maintenant faire ses preuves au niveau industriel, on peut espérer que ses concepts novateurs soient reconnus comme ils le méritent.

Si la disproportion entre les forces en présence - OMG contre communauté Ada - laisse peu de place au rêve, du moins peut-on prédire à PolyORB un succès certain parmi la communauté internationale d'Ada 95. PolyORB représente aussi un argument supplémentaire pour convertir de nouveaux adeptes à la programmation Ada.

BIBLIOGRAPHIE

Publications

[P1] « CORBA vs ADA 95 DSA. A programmer's view » - Yvon Kermarrec (ENST Bretagne)

[P2] « CORBA and CORBA Services for DSA » - Laurent Pautet, Thomas Quinot, Samuel Tardieu (ENST Paris) and the AdaBroker team

[P3] « CORBA & DSA: Divorce or Marriage? » - Laurent Pautet, Thomas Quinot, Samuel Tardieu (ENST Paris)

[P4] « Conception et réalisation d'un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables » - thèse de doctorat de Thomas Quinot (mars 2003) (1ère partie)

Livres

[L1] « Client/Serveur, Guide de survie » - Orfali, Harkey, Edwards

[L2] « Vers Ada 95 par l'exemple » - Fayard, Rousseau

Cours CNAM

[C1] « Présentation de l'architecture CORBA » – cours de Yves LALOUM (UV 16847)

[C2] « Introduction à CORBA et à DCOM » – cours de René CHEVANCE (UV 16847)

Sites Internet

[S1] Site de l'OMG : www.omg.org

[S2] Site de ACT Europe : http://libre.act-europe.fr/polyorb/

[S3] Site ACM Portal (ACM SIGAda Ada Letters) : http://portal.acm.org