SIMES - 961620
Système d'Information Multimédia
Pour l'Environnement Subsaharien

Software development

Deliverable number : D3
Nature:P
Contractual Date of Delivery: 14 August 1998
Task WP6 : Project management

Nom du rédacteur : Jean-Claude Derniame
Institut : INRIA
Adress
Batiment LORIA, Campus Scientifique. B.P. 239
54506 Vandoeuvre-lès-Nancy Cedex
France
derniame@loria.fr

Abstract:

Ce document décrit une proposition d'implantation d'un outil commun pour réaliser le support d'accès à des objets distribués sur un réseau selon la technologie CORBA. C'est un guide d'installation de l'outil MICO sur Windows 95/NT ainsi qu'un guide d'utilisation. MICO peut également être implanté sur de nombreuses autres machines. MICO est un produit libre. Ce guide est à l'usage des partenaires de SIMES et particulièrement des projets pilotes. Il a été expérimenté avec succès. Ce travail a été réalisé pendant le séjour que Georges Edouard KOUAMOU a effectué au LORIA à Nancy de juin à septembre 1998.

This document proposes an implementation for a common tool to realize the access support to distributed objects on a net following the Corba technology. It is an installation guide of MICO tool on Windows 95/NT and also an user manual. MICO can also be implemented on numerous systems. MICO is a free ware. This guide is for the use of SIMES partners and particularly for pilot operations.It has been successfully experimented. This work has been realized during the stay of Georges Edouard KOUAMOU at LORIA in Nancy, June to September 1998.

Les Standards de SIMES

Installation et Utilisation de MICO, une réalisation de CORBA

1. PRÉSENTATION GÉNÉRALE DE CORBA ... 5

1.1. LE MODÈLE OBJET ... 5
1.2. L'INTERFACE DEFINITION LANGUAGE (IDL) ... 6
1.3. STRUCTURE ET FONCTIONNEMENT DE L'ORB ... 7
1.3.1. Les Clients ... 8
1.3.2. L'implantation d'objets ... 8
1.3.3. Les proxy ... 8
1.3.4. L'Interface d'Invocation Dynamique (DII) ... 8
1.3.5. Le Dictionnaire d'Interface (IR: Interface repository) ... 9
1.3.6. Le Squelette d'Interface Statique (SSI: Static Skeleton Interface) ... 9
1.3.7. Le Squelette d'Interface Dynamique (DSI: Dynamic Skeleton Interface) ... 9
1.3.8. L'Adaptateur d'Objet (OA: Object Adapter) ... 9
1.3.9. Le Dictionnaire d'implémentation (Implementation Repository) ... 10
1.3.10. L'Interface de l'ORB ... 10
1.4. CORBA VS. JAVA ... 10
1.5. CONCLUSION ... 11

2. LES PRÉREQUIS À L'INSTALLATION DE MICO ... 12

2.1. CYGNUS CDK ... 12
2.2. MAKE ... 12
2.3. COMPILATEUR C++ ... 12
2.4. JAVA ... 13
2.5. JAVACUP ... 13
2.6. LES VARIABLES D'ENVIRONNEMENT ... 13

3. INSTALLATION DE MICO ... 14

4. UTILISATION DE MICO ... 14

4.1. CRÉATION D'UNE APPLICATION MICO ... 14
4.2. CAS 1: CLIENT ET OBJET DANS UN MÊME FICHIER ... 15
4.3. CAS 2: CLIENT ET OBJET SÉPARÉS ... 16
4.3.1. Exécution sur la même machine, éventuellement sur des shells différents ... 16
4.3.2. Exécution sur des machines distantes ... 17

5. CONCLUSION: NOTRE EXPÉRIENCE ... 18

6. BIBLIOGRAPHIE ... 19

Introduction

Les langages modernes utilisent le *paradigme objet* pour structurer les programmes s'exécutant sur un processus unique d'un système d'exploitation donné. La tendance actuelle est d'étendre cette logique à des applications distribuées s'exécutant sur plusieurs processus d'une ou de plusieurs machines. Cependant, dans un environnement réseau, les machines diffèrent par leur architecture; de même, leur système d'exploitation et les langages utilisés pour coder les composants peuvent être différents. Faire communiquer des objets dans un tel environnement nécessite la définition d'un référentiel commun mis à disposition par une une plate-forme intermédiaire complexe. CORBA[1], [2], [3],[5] est la spécification d'une telle plateforme et MICO [4] en est une implantation.
CORBA, pour Common Object Request Brocker Architecture, est un standard de l'OMG de plus en plus souvent utilisé dans les applications distribuées. Permettant la déclaration d'objets accessibles depuis les stations d'un réseau, il se présente comme un ensemble de services de gestion de ces objets et de réalisation d'applications sur ces objets.
MICO est le fruit d'un projet conjoint entre l'Université de Francfort et l'Institut d'Informatique de Berkeley. L'intention est d'offrir une implémentation gratuite et complète de CORBA 2.0. Il est implémenté avec C++ [7] et il s'exécute sur plusieurs plates- formes parmi lesquelles:

- Solaris 2.5.1 sur Sun SPARC,
- Linux 2.0.x sur Intel x86,
- AIX 4.2 sur IBM RS/6000,
- Digital Unix 4.x sur DEC Alpha,
- HP-UX 10.20 sur PA-RISC;
- Ultrix 4.2 sur DEC Mips
- Windows 95/NT

Il est disponible en code source uniquement pour toute éventuelle installation sur Windows NT/95. Son installation exige des prérequits. Sur Windows 95/NT, il faut disposer de *Cygnus CDK* un support des outils GNU pour Win32.
Avant d'essayer de compiler les sources, s'assurer de la disponibilité des packages suivants: Cygnus CDK, Make, un compilateur C++ (voir en †3.3, “Compilateur C++,”) la liste précise des compilateurs acceptés, Java et JavaCup [6].
Dans la suite, nous commencerons par une présentation générale de CORBA. La section 3 détaille les prérequis en terme de connaissances, de savoir faire et des outils indispensables à l'installation de MICO. Ensuite nous enchaînerons avec les manipulations d'installation et enfin la dernière section s'intéressera à quelques exemples d'utilisation.

1. Présentation générale de CORBA

CORBA est un environnement hétérogène distribué qui permet d'intégrer des applications diverses et hétérogènes, en d'autres termes écrites avec différents langages de programmation et s'exécutant sur des plates-formes variées. Son intérêt réside dans la définition et la promotion d'une architecture et d'un ensemble de spécifications, basés sur la technologie objet, réalisant l'interopérabilité entre applications distribuées sur des réseaux de systèmes hétérogènes .

1.1. Le modèle objet

Le modèle objet sous-jacent offre une présentation organisée de la terminologie et des concepts d'objet. Il est abstrait en ce sens qu'il ne peut être directement réalisé par une technologie particulière.

Client

Le client d'un objet est une entité qui souhaite effectuer des opérations (requêtes) sur cet objet.

Objet

C'est une entité identifiable et encapsulée qui offre des services qui peuvent être demandées par un client.

Requête

C'est un évenement. A une requete sont associés l'opération, l'objet interrogé, les paramètres et optionnellement un contexte.

Interface

L'interface decrit un ensemble d'opérations possibles qu'un client pourrait demander à un objet. IDL (Interface definition Language) permet de les spécifier.

Opération

C'est une entité identifiable qui contribue à un service offert aux clients.

Attribut

Un interface contient éventuellement des attributs. Un attribut est logiquement équivalent à la déclaration d'une paire de fonctions: une pour l'extraction de la valeur et l'autre pour la mise à jour.

Implémentation de l'objet

C'est un concept relatif à la réalisation des comportements des objets dans un système automatique. Le modèle d'implémentation comprend le modèle d'exécution qui décrit le traitement des services et le modèle de construction qui décrit la définition des services.

1.2. L'Interface Definition Language (IDL)

L'IDL est un langage utilisé pour décrire l'interface auxquels ont accès les clients de l'objet. C'est un langage de spécification qui offre très peu de détails sur l'implantation mais des correspondances vers différents langages de programmation peuvent être définies. Par conséquent, le client ne peut être écrit en IDL mais dans l'un des langages pour lequel la correspondance des concepts de l'IDL sont définies. L'IDL permet l'interopérabilité entre clients et serveur d'objets (figure 1)écrit dans différents langages de programmation.

L'IDL est constitué d'un ensemble de conventions lexicales comprenant des mots clés, des commentaires, des identificateurs et des littéraux. Sa grammaire est un sous- ensemble de celle de C++ augmenté des instructions nécessaires aux mécanismes d'invocation des opérations. Elle supporte la syntaxe des constantes, des types et de la déclaration des opérations, mais elle n'inclut pas la définition des variables et des structures algorithmiques. Attention, l'IDL permet une interopérabilité presqu'uniquement syntaxique, en ce sens que seuls les profils des opérations sont définis. Le modèle d'objet permet de capturer un peu de sémantique (le comportement) mais il reste encore beaucoup a faire .

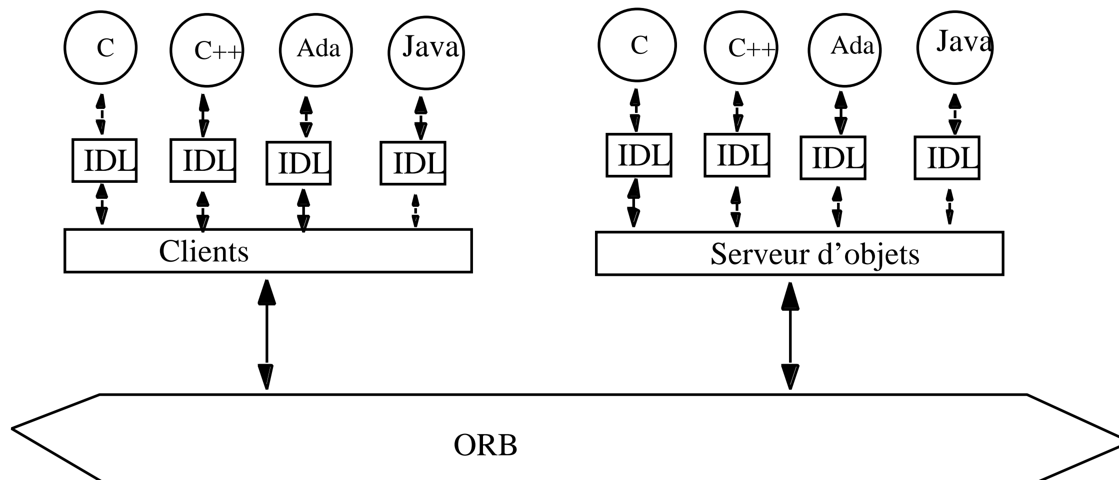

Figure 1: L'interopérabilité Client/Serveur offerte par l'IDL

1.3. Structure et fonctionnement de l'ORB

L'Object Request Broker (ORB) est responsable de toutes les interactions entre Clients et Objets. Il doit être vu comme un ensemble logique de services plutôt qu'une bibliothèque ou un processus particulier.

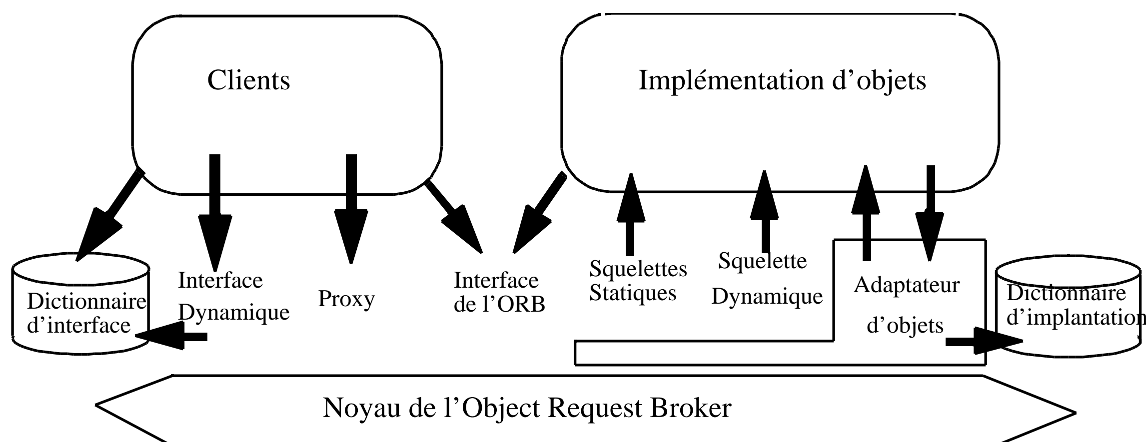

Figure 2 : anatomie de l'ORB

Pour formuler une requête, le client peut utiliser l'interface d'Invocation Dynamique ou les Proxy IDL. Il peut directement intéragir avec l'ORB pour des fonctions particulières. Du côté de l'objet, il peut recevoir la requête à travers le Squelette Statique ou le Squelette Dynamique. Pendant le traitement de la requête l'implantation de l'objet fait appel à l'Adaptateur d'Objets de l'ORB.

1. Le client accède à la référence de l'objet pour connaître le type d'objet et les opérations offertes. Cet accès fait appel à l'un des Interfaces d'Invocation (Statique ou Dynamique). Les requêtes statiques et dynamiques satisfont la même sémantique, donc le récepteur ne peut faire aucune différence entre ces deux types de requête.

2. L'ORB localise l'implémentation appropriée, transmet les paramètres et transfert le contrôle à l'implémentation d'objet par le squelette dynamique ou statique. Pour traiter la requête, l'implantation d'objet sollicite des services de l'ORB par l'Adaptateur d'objet. Les résultats sont transmis au client dès que la requête est terminée.

1.3.1. Les Clients

On ne peut parler de client que par rapport à un objet dont il a accès à la référence. Il connaît uniquement la structure logique de l'objet relativement à son interface et teste le comportement de l'objet à travers les invocations. Par conséquent, l'implémentation d'un objet peut être le client d'autres objets.

1.3.2. L'implantation d'objets

L'implantation d'un objet offre la sémantique de l'objet, en définissant les données pour les instances et le code des méthodes de l'objet. Toutefois il est possible d'utiliser des logiciels additionnels ou d'autres objets pour implémenter le comportement de l'objet, par exemple un programme par méthode, des bibliothèques, une BDOO, etc.

1.3.3. Les proxy

Encore appelés Clients Stubs, ils présentent aux clients l'accès aux opérations sur l'objet définies en IDL. Globalement, ils sont définis statiquement pour permettre aux clients d'invoquer des opérations sur les objets. Pour la correspondance d'un langage non orienté objet, la programmation d'un interface aux proxys est indispensable pour chaque type d'interface. Si plusieurs ORB sont disponibles, il y aura un proxy correspondant à chacun d'entre eux. Dans ce cas l'association d'un proxy correct avec la référence d'un objet particulier revient à l'ORB et au langage de correspondance.

1.3.4. L'Interface d'Invocation Dynamique (DII)

Tout comme les proxys, l'Interface d'Invocation Dynamique (DII) est utilisé pour invoquer des requêtes sur l'objet à la seule différence que ces requêtes sont créées dynamiquement. Un client devrait préciser l'objet, l'opération à exécuter et un ensemble de paramètres pour cette opération.
Les requêtes sont définies en terme de pseudo-objet **Request**. Elles utilisent les types NVList pour décrire la liste des arguments. Les opérations sur les requêtes sont définies en terme de méthodes sur des pseudo-objets. Elles utilisent la structure **NVlist** qui ne peut être créée qu'en utilisant l'opération create_list de l'ORB.

1. Create_request: C'est une méthode de l'interface **Object**. Cette opération crée un pseudo-objet Request. Elle est exécutée sur l'objet destination i.e qui reçoit la requête. Plusieurs syntaxes cohabitent dont la plus orthodoxe est la suivante:

```
Status create_request(
in Context          ctx,   // contexte
in Identifier       operation    // nom de l'opération sur l'objet
in NVList          arg_list,     // les arguments de l'opération
inout NamedValued        result,  // le résultat de l'opération
out Request              request,      // nouvelle requête créée
in Flags                 req_flags     // contrôle de l'allocation mémoire
);
```

Les autres opérations, définies ci-dessous, agissent sur l'objet "Request" et permettent de le modifier (ajout d'arguments et suppression de la requête), de l'envoyer ou de recevoir la réponse.

2. Add_arg: ajoute incrémentallement des arguments à la requête. Les arguments qui ne sont pas spécifiés pendant la création de la requête sont ajoutés en utilisant add_arg. Toutefois l'utilisation des deux méthodes à la fois pour spécifier les arguments de la même requête n'est pas encore acceptable.
3. Delete: détruit la requête et libère la mémoire allouée à ses arguments.
4. invoque: Cette opération appelle l'ORB qui exécute la méthode appropriée. Si le résultat est correct, il est placé dans la variable result spécifiée dans la requête. Le comportement est imprévisible si la requête a été utilisée précédemment avec invoque, send ou send_multiple_request.
5. Send: initie une opération en se basant sur les informations contenues dans la requête. Send retourne le contrôle au client sans attendre la fin de l'exécution de l'opération. *get_response* et *get_next_response* permettent au client de savoir si l'exécution de la requête est achevée.

1.3.5. Le Dictionnaire d'Interface (IR: Interface repository)

L'IR est la composante de l'ORB qui offre le stockage des objets persistants que représentent les définitions d'interfaces écrits en IDL. Il contient des objets CORBA dont l'élément minimal est **IRObject** c'est à dire que tous les autres objets héritent de ses propriétés. On peut regrouper ces objets sous forme d'ensembles et d'éléments dont la relation d'appartenance et/ou d'inclusion symbolisent l'héritage.

1.3.6. Le Squelette d'Interface Statique (SSI: Static Skeleton Interface)

Il contient la signature de l'implantation d'objet générée à partir des interfaces définis en IDL. Son existence n'implique pas nécessairement celle du proxy correspondant parce que le client peut éventuellement utiliser le DII pour interroger l'objet. Toutefois il est possible de développer un adaptateur d'objet qui n'utilise pas le SSI pour accéder aux méthodes de l'implémentation de l'objet.

1.3.7. Le Squelette d'Interface Dynamique (DSI: Dynamic Skeleton Interface)

Le DSI permet de délivrer dynamiquement des requêtes de l'ORB à un objet c'est à dire sans que le squelette de l'objet soit statiquement compilé dans le programme. L'implémentation de l'objet est atteinte à partir d'un interface qui offre l'accès au nom de l'opération et à ses paramètres.
L'implémentation de l'objet donne à l'ORB la description des paramètres de l'opération et l'ORB lui transmet les valeurs de chacun des paramètres d'entrée. Ensuite l'implémentation de l'objet donne à l'ORB la valeur de chacun des paramètres de sortie ou une exception que l'ORB transmet au client.

1.3.8. L'Adaptateur d'Objet (OA: Object Adapter)

L'OA est le moyen primaire utilisé par l'implémentation de l'objet pour accéder aux services offerts par l'ORB. Il existe plusieurs ORB disponibles Les services offerts par l'ORB à travers l'OA sont: la génération et l'interprétation des références d'objet, l'invocation des méthodes la sécurité des interactions, l'activation et la désactivation d'un objet ou de son implémentation, le passage d'une référence d'objet à son implantation et l'enregistrement des

implémentations.

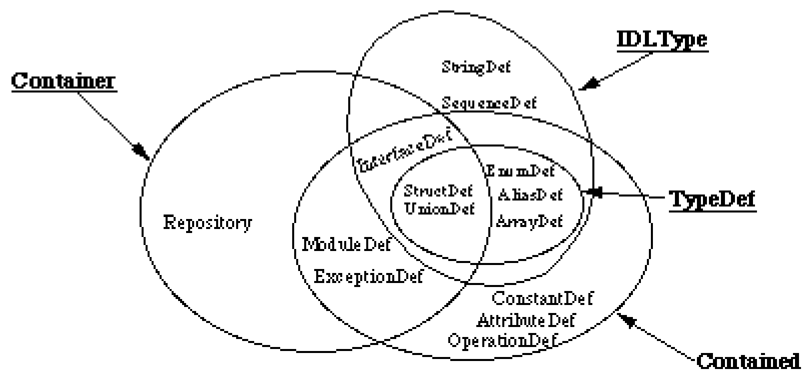

Figure 3 : Schéma d'héritage entre objets CORBA

Au vue de la granularité (le degré de détail de la connaissance de l'objet), du cycle de vie, des styles d'implémentation et autres propriétés des objets, il est difficile au noyau de l'ORB d'offrir un interface unique convenable à tous les objets. Ainsi l'OA permet d'atteindre un groupe particulier d'objets ayant des besoins similaires.

1.3.9. Le Dictionnaire d'implémentation (Implementation Repository)

Le Dictionnaire d'implémentation contient les informations nécessaires qui permettent à l'ORB de localiser et d'activer les implantations des objets. La plupart de ces informations sont spécifique à un ORB ou à un environnement d'exécution.

1.3.10. L'Interface de l'ORB

La plupart des fonctionnalités de l'ORB sont accessibles par l'OA, les proxys, le DII, le SSI et le DSI. L'interface de l'ORB est accessible aux clients et aux objets sans distinction aucune. Il communique directement avec l'ORB et contient quelques opérations utiles. Il ne dépend pas de l'OA sur lequel il s'exécute mais il est le même pour tous les ORBs et tous les objets.

1.4. CORBA vs. Java

Au delà de la portabilité des composantes, CORBA prend en compte, à partir de la version 2, l'interopérabilité entre les ORBs . Le protocole **GIOP** (***General Inter ORB Protocol***) assure la communication inter-ORB. Il spécifie un ensemble de formats de messages et représentations des

données communes pour la communication entre ORBs. Il est conçu pour travailler sur tout protocole de connection orienté transport. Le protocole **IIOP** (***Internet Inter ORB Protocol***) spécifie comment sont échangés les messages GIOP sur des réseaux TCP/IP. Il rend possible la liaison entre ORBs en utilisant Internet comme pivot.

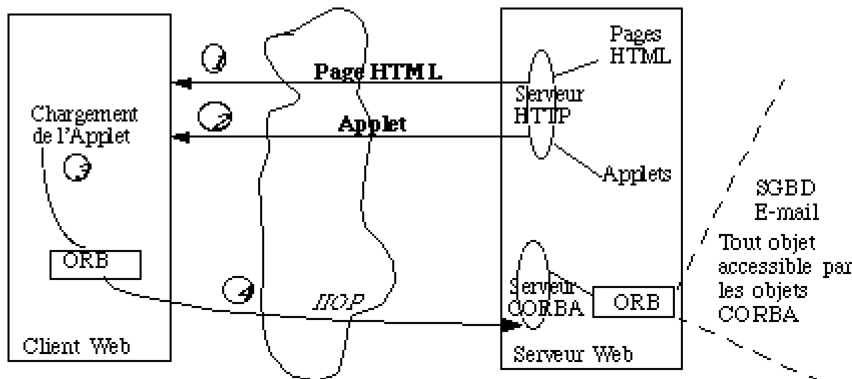

Figure 4. Interaction Clients/objets sur l'object Web

La tendance future consisterait à exploiter les performances d'expression de Java. En effet, le Web interactif tel que connu actuellement utilise le protocole HTTP avec CGI pour charger les pages Web, les applets et les images. Le protocole CGI présente des lacunes telles que la lenteur, l'encombrement. CORBA restreint ces lacunes en offrant la possibilité de distribution des objets sur les intranets et Internet. CORBA peut donc remplacer le protocole CGI/HTTP comme couche intermédiaire de communication entre objets. En d'autres termes, les deux protocoles (IIOP et HTTP) s'exécuteront sur les même réseaux.
Toutefois, Java offre une infrastructure d'objets portables qui s'exécutent sur différentes plate-formes et une grande flexibilité pour le développement des applications distribuées, mais ne supporte pas actuellement le paradigme client/serveur; c'est à dire qu'il n'est pas facile aux applets Java d'invoquer des méthodes sur des objets distants. Java pourrait être le langage idéal pour l'écriture des clients et des objets. CORBA quant à lui offre une infrastructure d'objets distribués hétérogènes qui permet aux applications d'accroître leur richesse à travers les réseaux, les langages de programmation et les systèmes d'exploitation. Alors la plate-forme CORBA/Java serait meilleure pour la création des applications Client/Serveur basées sur le Web, car CORBA assurera la transparence réseau et Java assurera la transparence d'implémentation. Ce mariage entre objets distribués et le W3 est connu sous le nom de **«Object Web»**. Le schéma de la figure†4 illustre l'interaction entre clients et objets sur l'object Web.

1. Chargement de la page HTML. Dans ce cas, la page inclut des références pour contenir les applets Java.
2. Extraction de l'applet Java du serveur HTTP. Le serveur HTTP extrait l'applet et le charge sous forme de codes binaires.
3. Chargement de l'applet. L'applet est chargé en mémoire
4. Invocation du serveur d'objets CORBA. L'applet Java peut contenir des talons du client générés par l'IDL qui lui permet d'invoquer l'objet sur l'ORB. Alternativement, l'applet peut utiliser l'Interface d'Invocation Dynamique pour générer des requêtes. Cette session entre Applet Java et serveur d'objets CORBA persistera jusqu'à ce que l'un se déconnecte.

1.5. Conclusion

En conclusion, CORBA[1] est la composante clé de l'OMA (object Management Architecture) de l'OMG spécifiée pour faire face aux défis du développement des systèmes distribués:

¥ Rendre le développement des applications distribuées aussi facile que le développement des programmes centralisés.

¥ Offrir une infrastructure pour intégrer les composantes d'une application dans un système distribué.

Cette section a décrit de façon sommaire les principales composantes de l'architecture

[1] Cette présentation sera complétée par un cours CORBa et MICO,lors d'une prochaineréunion

CORBA. Il est à noter qu'il devient le standard actuel pour plusieurs applications distribuées. Parmi les implémentations on peut citer: ORBIX de Iona, Visibroker de Visigenic et Netscape, Joe de Sun, ObjectBroker de Digital, SOM de IBM, ORB Plus de HP, et MICO dont nous allons présenter les détails d'installation et d'utilisation.

2. Les prérequis à l'installation de MICO

Pour s'initier à l'épreuve d'installation, il faut sortir des carcans des interfaces graphiques, ergonomiques et faciles d'utilisation pour s'intéresser aux lignes de commande, à l'édition des script sous MSDOS et particulièrement sous Unix.

2.1. Cygnus CDK

Cygnus CDK bêta 19 est un support des outils gnu pour Win32. Il est disponible en binaire sur ftp://ftp.cygnus.com/pub/gnu-win32/. Son installation est relativement facile. Il suffit d'exécuter cdk.exe ou setup.exe et de suivre les directives. Il comprend à la base toutes les commandes Unix, entre autre bison1.2x, flex2.5. le compilateur C/C++ (gcc 2.7) et les bibliothèques correspondantes.
Mais ce compilateur ne supporte pas convenablement les exceptions par exemple il parait incapable d'utiliser la mémoire virtuelle. Ainsi des erreurs du type ***"virtual memory alloc fail"*** surviendront pendant la compilation de vos codes. Cette situation nous a contraint à installer egcs 1.0.3. qui est un outil gnu comprenant une version du compilateur C/ C++ de gnu.
Il est conseillé de maintenir le répertoire par défaut que propose cdk pour éviter toutes les manipulations nécessaires qui permettront à bison de retrouver l'arborescence de ses fichiers. Après l'installation, cdk s'exécute dans une fenêtre de commande MsDos.

2.2. Make

Le configurateur *make* est un outil (Unix) permettant de définir dans un fichier texte dont le nom sera *Makefile* une liste de règles de construction de fichier. Il permet à l'utilisateur de spécifier les règles de construction de ses exécutables avec les options de son choix. Il permet également de déterminer les règles à exécuter et celles qui n'ont pas besoin d'être lancées. Quelques instructions remarquables:

#fichier makefile
Le caractère # indique un commentaire

Définition des variables
Elle est faite tout simplement par VAR = chaîne de caractères
ex: OPTION= -g -O2 # Ceci est la définition d'une variable.

Règles de construction générique d'un fichier objet (.o) à partir d'un fichier source (.c)
.c.o:
cc $(OPTION) -c $<
Il n'en existe bien d'autres encore que nous ne pouvons détailler dans le cadre du présent document.

2.3. Compilateur C++

Il est important de disposer d'un compilateur de la liste suivante avec les bibliothèques correspondantes:

- g++ 2.7.2.x et libg++ 2.7.2 ou
- g++2.8.x et libg++ 2.8.x ou
- egcs 1.x

l'un d'entre eux dépend du degré de facilité de l'installation; et du point de vue technique, cette liste est proposée en tenant compte de la gestion des exceptions. Le compilateur est utile pendant l'installation et également pendant l'exécution. En fait les programmes de compilation et d'édition des liens de MICO sont des scripts qui intègrent les options de l'utilisateur et font appel au compilateur C++ natif.
Nous avons opté pour egcs pour les raisons suivante: il est légèrement plus facile à installer que g++ et une bibliothèque C++ y est incorporée d'avance.

2.4. Java

Télécharger Java pour Windows NT/95 depuis le site http://www.sun.com/cgi-bin/ . Exécuter le programme binaire importé en suivant les directives d'installation.

2.5. Javacup

Comme YACC sous Unix, CUP est un système pour générer des analyseurs LALR pour Java à partir de spécifications simples. Il est écrit en Java contrairement a YACC développé en C. Il utilise des spécifications comprenant des codes Java et produit des analyseurs implémentés en Java.
Cup est disponible sur le site http://www.cs.princeton.edu/~appel/modern/java/ CUP/cup0.10 pour ce qui est de la version 0.10. Pour l'installer procéder comme il suit.

1. Télécharger **java_cup.V010g.tar.gz** et le sauvegarder dans un répertoire de votre choix de préférence dans un répertoire des «classes» accessible par l'interpréteur Java.
2. Décompresser le fichier: taper **tar -xfz java_cup.V010g.tar.gz** sur la ligne de commande bash de cdk et s'assurer que le répertoire cible est présent dans votre variable ***classpath***
3. Compiler les sources à partir du répertoire des codes sources: taper javac java_cup/*.java java_cup/runtime/*.java ou bien faite le en deux fois. d'abord : javac java_cup/*.java
   ensuite: javac java_cup/runtime/*.java
4. Tester

Pour le faire, se placer dans le sous-répertoire simple_calc de java_cup.
taper: **rm parser.java sym.java** pour détruire ces deux fichier.
taper ensuite: **java java_cup.Main < parser.cup**. Elle génère les fichier précédemment détruits.
Ce package ainsi que JDK ne sont pas requis par MICO, mais ils sont utiles pour l'affichage du Dictionnaire d'Interface Graphique. Donc MICO peut être installé et exécuté sans le package Java.

2.6. Les variables d'environnement

Cygnus importe automatiquement les variables définies pour Windows NT/95. Dans le fichier cygnus.bat se trouve d'autres variables utiles. Nous présentons ici l'extrait d'un script qui positionne les variables d'environnement de cdk.
@ECHO OFF
SET MAKE_MODE=unix
SET CYGROOT=z:\home\cg32
SET MICOROOT=z:\home\mico # Répertoire où réside les sources de Mico
SET CYGFS=z:/home/cg32 # Répertoire où est installé Cygnus
SET CYGREL=B19

SET GCC_EXEC_PREFIX=%CYGROOT%\H-i386-cygwin32\lib\gcc-lib\
SET TCL_LIBRARY=%CYGROOT%\share\tcl8.0\
SET GDBTK_LIBRARY=%CYGFS%/share/gdbtcl

SET JAVA_HOME=z:\home\jdk1.1.6\bin
SET JCUP_HOME=z:\Home\jdk1.1.6\classes\java_cup
Spécification des chemins de recherche des classes java. Il est conseillé d'installer Java_cup dans le répertoire classes afin qu'il soit accessible par l'interpréteur java
SET CLASSPATH=./;../;%JAVA_HOME%\..\classes;%JAVA_HOME%\..\lib\classes.zip
LD_LIBRARY_PATH indique le chemin des bibliothèques C/C++ d'abord celle de MICO, ensuite de gcc 2.7 inclu dans cygnus et enfin celles de egcs1.0.3 que nous avons installé.

SETLD_LIBRARY_PATH=%MICOROOT%\orb;%CYGROOT%\H-i386-cygwin32\lib;%CYGROOT%\H- i386-cygwin32\i386-cygwin32\lib;%LD_LIBRARY_PATH%
SETLD_LIBRARY_PATH=%CYGROOT%\H-i386-cygwin32\i386-pc-cygwin32\lib;%LD_LIBRARY_ PATH%
Il faut mettre à jour le chemin de recherche des exécutables: commandes de Cygnus, l'interpréteur java et éventuellement les exécutables de MICO s'il sont installés ailleurs que dans le répertoire des commandes de Cygnus.
SETPATH=%CYGROOT%\H-i386-cygwin32\bin;%JAVA_HOME%;%JAVA_HOME%\..\classes;%JCUP_HOME%;%PATH%
echo Cygnus Cygwin32 %CYGREL%
Ce script peut être enregistré sous le nom cygnus.bat. Ouvrir une fenêtre de commande MSDOS et exécuter ce script. Taper ensuite bash et votre interpréteur de commande est actif. Il faut passer à la compilation de votre programme MICO.

3. Installation de MICO

L'installation de MICO passe par les étapes suivantes:

1. Les sources sont disponibles sur le site: http://www.vsb.cs.uni-francfurt.de/~mico/ Il faut le télécharger et le sauvegarder dans le répertoire de votre choix.
2. Décompresser le fichier en utilisant la commande: **gzip -dc mico-2.1.1.tar.gz | tar xf -** en supposant que c'est la version 2.1.1 que vous installez. Ceci crée un répertoire mico contenant les sources.
3. Se placer dans ce nouveau répertoire et taper: **./configure --help** pour prendre connaissances de toutes les options de configuration. configure est un script qui permet d'éditer le fichier makefile, de vérifier le système et bien d'autres configurations.
4. taper : **make** pour commencer la compilation des bibliothèques et des programmes.
5. **make install** n'est pas nécessaire si vous avez spécifié les sous-répertoires de mico comme chemin de recherche des commandes.

Pendant ces opérations, il est possible d'éditer makefile pour spécifier les chemins de recherche des fichiers d'entête, des bibliothèques pour obtenir un résultat satisfaisant, car par défaut, les scripts sont conçus pour fonctionner sous le modèle d'organisation de Unix et donc un apport extérieur est utile pour les adapter à l'organisation de la machine hôte.

4. Utilisation de MICO

Cette section présente à travers un exemple, comment développer et exécuter une application MICO. L'exemple est basé sur la gestion d'un compte bancaire. Nous illustrerons davantage le style de programmation par rapport aux performances du logiciel.
Un compte offre trois types d'opération: déposer une certaine somme d'argent, retirer une certaine somme et vérifier l'état du compte. L'état d'un objet de ce type concerne le montant qui s'y trouve. Le fragment de code suivant montre la déclaration de la classe d'un tel objet. Il décrit l'interface et l'état d'un objet compte.

```
class Account {
long _current_balance;
public:
Account(); /* Constructeur */
void deposit (unsigned long amont); /*dépôt d'argent */
void withdraw (unsigned long amountx); /* retrait */
long balance (); /* vérification de l'état du compte*/
};
```

4.1. Création d'une application MICO.

L'implémentation reflète le comportement de l'objet. Pour les applications MICO, on doit

séparer la spécification de l'*interface* de l'objet de l'*implémentation*, ceci parce les objets peuvent être implémentés dans différents langages de programmation. Le processus de création d'une telle application est schématisé par la figure 5.

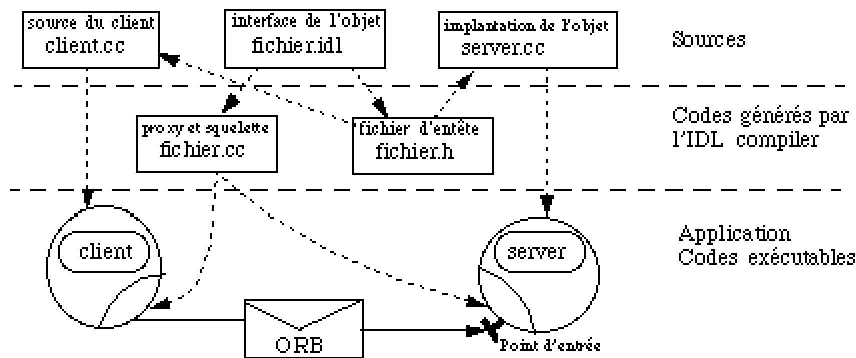

Figure 5. Processus de création d'une application.

Le développeur écrit les codes sources représenté par fichier.idl, client.cc et server.cc. A partir de cette figure, on découvre les liens de dépendance qui existent entre les différents composants nécessaires à l'édification de l'application.
Fichier.idl décrit l'interface de l'objet qui est l'ensemble des services offerts aux clients ie les opérations visibles de l'extérieur de l'objet. Server.cc décrit l'état de l'objet et le comportement des services fournis aux clients. Client.cc décrit l'état et le comportement du client.
Le compilateur d'IDL génère à partir de l'interface de l'objet:

- un fichier d'entête utilisé par le client et l'implantation de l'objet pendant la compilation (#include "fichier.hx),
- et un autre fichier contenant les proxys et le squelette d'invocation statique utilisé par le client et l'implantation de l'objet pendant l'édition des liens.

A l'exécution l'ORB joue le rôle de la poste. Il assure les interactions entre client et objet. L'interface de communication est assuré par les proxys coté client et le squelette coté objet.

4.2. Cas 1: Client et objet dans un même fichier

```
interface Account {
   void deposit( in unsigned long amount );
   void withdraw( in unsigned long amount );
   long balance();
};
```

Ce fragment de code est la déclaration de l'interface de l'objet "Account". On suppose qu'il est enregistré sous le nom account.idl. L'utilisation de l'IDL compiler de MICO se fait de la manière suivante: **idl account.idl**
Ensuite, il faut décrire le comportement des opérations de l'interface et écrire le programme qui utilise ces objets. Voici un extrait (main.cc):

```
#include "account.h" /* Déclaration virtuelle de la classe générée de l'IDL */
class Account_impl : virtual public Account_skel {
private:
  CORBA::Long _current_balance; /*Ajouter le comportement de l'objet */
public:
  Account_impl() { _current_balance = 0; }; /* constructeur */
  void deposit( CORBA::ULong amount ){ _current_balance += amount; };
  void withdraw( CORBA::ULong amount ) { _current_balance -= amount; };
  CORBA::Long balance(){ return _current_balance;};
};
/* La suite concerne l'utilisation de l'objet spécifié. Il est composé de deux parties: le serveur
qui offre des services et le client qui invoque ces services offerts par le serveur*/
```

```
int main( int argc, char *argv[] )
{
  // initialisation de l'ORB: Ces fonctions sont utilisées pour obtenir un pointeur sur l'ORB
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa");

// Coté serveur de l'objet
/* Instantiation d'un objet account appélé "server"
  Account_impl* server = new Account_impl;
  CORBA::String_var ref = orb->object_to_string( server );
  cout << "Server reference: " << ref << endl;
  //----------------------------------

// Coté du client
  CORBA::Object_var obj = orb->string_to_object( ref );
  Account_var client = Account::_narrow( obj );
  client->deposit( 700 );
  client->withdraw( 250 );
  cout << "Balance is " << client->balance() << endl;

/* On a plus besoin de l'objet, il est libéré par l'appel de la méthode release. Ce code appartient à l'implantation de l'objet */
  CORBA::release( server );
  return 0;
}
```

Pour générer un exécutable (account.exe), compiler les codes de la façon suivante:

mico-c++ -I. -c main.cc -o main.o *# Création des codes objets*
mico-c++ -I. -c account.cc -o account.o
mico-ld -o account main.o account.o -lmico2.1.1 *# Edition des liens*

Exécuter account.exe et apprécier le résultat.

4.3. Cas 2: Client et objet séparés

Les applications MICO ne sont intéressantes que lorsque l'objet et le client constituent deux processus exécutables sur une seule machine ou sur différentes machines.

4.3.1. Exécution sur la même machine, éventuellement sur des shells différents

Le problème majeur dans ce cas est de transmettre la référence de l'objet au client. Les deux processus se partagent un système de fichier unique. L'objet convertit sa référence en une chaîne qu'il écrit dans un fichier accessible par le client en lecture.

```
// fichier server.cc
#include "account.h" /* Déclaration virtuelle de la classe générée de l'IDL */
#include <iostream.h>
#include <fstream.h>

class Account_impl : virtual public Account_skel {
/*......................;Inchangé: voir la section précédente */
};
int main( int argc, char *argv[] )
{
// initialisation de l'ORB: Ces fonctions sont utilisées pour obtenir un pointeur sur l'ORB
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa");

/* Instantiation d'un objet account appélé "server"
  Account_impl* server = new Account_impl;
  CORBA::String_var ref = orb->object_to_string( server );/*convertir la reférence une
```

```
chaine*/
  ofstream out ("/tmp/account.objid"); /* Le sauvegarder dans le fichier account.objid */
out << "Server reference: " << ref << endl;
out.close();

/* activé l'objet implémenté par server en utilisant la méthode impl_is_ready() du BOA*/
boa->impl_is_ready( CORBA::ImplementationDef::_nil());
orb->run();

/* On a plus besoin de l'objet, il est détruit par l'appel de la méthode release.*/

  CORBA::release( server );
  return 0;
}
//     fichier client.cc
#include "account.h" /* Déclaration virtuelle de la classe générée de l'IDL */
#include <iostream.h>
#include <fstream.h>

int main( int argc, char *argv[] )
{
// initialisation de l'ORB: Ces fonctions sont utilisées pour obtenir un pointeur sur l'ORB
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa");

/* Lecture de la reference de l'objet */
ifstream in ("/tmp/account.objid");
char ref[1000];
in > >ref ;
in.close();

/*convertir la chaine en une reférence d'objet*/
CORBA::Object_var obj= orb->string_to_object(ref);
/* Convertir l'objet CORBA en un proxy de l'objet account appelé client */
Account_var client= Account::_narrow (obj);

/* Invocation de quelques opérations*/
client->deposit( 700 );
  client->withdraw( 250 );
  cout << "Balance is " << client->balance() << endl;
/* le client est détruit automatiquement dès la fin de son exécution */
return 0;
}
```

Compiler les sources pour générer deux exécutables server.exe et client.exe.

```
mico-c++ -I. -c server.cc -o server.o # Création des codes objets
mico-c++ -I. -c client.cc -o client.o # Création des codes objets
mico-c++ -I. -c account.cc -o account.o
mico-ld -o server server.o account.o -lmico2.1.1 # Edition des liens
mico-ld -o client client.o account.o -lmico2.1.1 # Edition des liens
```

Exécuter d'abord server.exe et ensuite client.exe sur un autre shell.

4.3.2. Exécution sur des machines distantes

Trois types d'adresse peuvent être utilisés pour identifier les processus sur une machine: adresse internet, adresse Unix et adresse locale. Voici un exemple d'utilisation:

```
/ / --------------------------------------------fichier server.cc
#include "account.h" /* Déclaration virtuelle de la classe générée de l'IDL */
```

```
class Account_impl : virtual public Account_skel {
/*......................;Inchangé: voir la section précédente */
};

int main( int argc, char *argv[] )
{
// initialisation de l'ORB: Ces fonctions sont utilisées pour obtenir un pointeur sur l'ORB
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa");

/* Instantiation d'un objet account appélé "server". Les opérations de conversion et d'écriture
de la référence de l'objet sont supprimées */
  Account_impl* server = new Account_impl;

/* activé l'objet implémenté par server en utilisant la méthode impl_is_ready() du BOA*/
boa->impl_is_ready( CORBA::ImplementationDef::_nil());
orb->run();

/* On a plus besoin de l'objet, il estdétruit par l'appel de la méthode release.*/
  CORBA::release( server );
  return 0;
}
//--------------------------------------------------------------- Client.cc
#include "account.h"
int main( int argc, char *argv[] )
{
   // initialisation de l'ORB
   CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
   CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");
/* Après l'initialisation, utiliser bind() pour rechercher un objet dont l'IR est IDL:Account:1.0
et qui s'exécute sur le port 8888 de la même machine*/
CORBA::Object_var obj = orb->bind ("IDL:Account:1.0", "inet:localhost:12456");

assert (!CORBA::is_nil (obj)); /* Continuer si une reférence a été trouvée: test d'échec*/
   Account_var client = Account::_narrow( obj );

   client->deposit( 700 );
   client->withdraw( 250 );
   cout << "Balance is " << client->balance() << endl;
   return 0;
}
mico-c++ -I. -c server.cc -o server.o # Création des codes objets
mico-c++ -I. -c client.cc -o client.o
mico-c++ -I. -c account.cc -o account.o
mico-ld -o server server.o account.o -lmico2.1.1 # Edition des liens
mico-ld -o client client.o account.o -lmico2.1.1
```

Charger le démon: micod -ORBIIOPAddr inet:localhost:12456 &
Le mettre en veilleuse: **trap "kill <micod_pid>"** où micod_pid est le numéro de processus du démon micod.
Faire enregistrer l'objet par le démon: imr create Account shared server 'IDL:Account:1.0' -ORBImplRepoAddr inet:localhost:12456 &
Exécuter le client: **./client**

5. Conclusion: notre expérience

Nous avons expérimenté l'installation de MICO sur NT. Toutes les étapes ont été réalisées en commençant par les prérequis. Tous les packages ont été chargés à partir des sites ftp ou des miroirs.

Les principales difficultés que nous avons rencontrées sont dues à la modification du script de configuration afin de l'adapter à l'environnement NT. On peut noter: le remplacement des liens entre fichier par des copies physiques sur disques, l'indication explicite des chemins d'accès aux bibliothèques et fichiers d'entête, la modification du contenu de certaines variables.
Un aspect non négligeable de cette épreuve est la confrontation à des cas pratiques de compilation modulaire. Cette expérimentation sera poursuivie sur Windows 95 et Linux qui sont des systèmes d'exploitation pour les PC Intel x86 et compatibles assez répandus en dehors des grosses Unités de recherche.

6. Bibliographie

[1] R. Orfali, D. Harkey, J. Edwards. Instant CORBA. Wiley Computer Publishing, New- York, 1997.

[2] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeous Environments. IEEE Communication Magazine, Vol. 35, No. 2, February 1997.

[3] OMG98. The Common Object Request Broquer: Architecture and Spécification. OMG, February 1998. Availlable on http://www.omg.org/corba/corbaiiop.htm

[4] K. Römer and A. Puder. MICO is CORBA: CORBA 2.0 implementation. Availlable on http://www.vsb.cs.uni-frankfurt.de/~mico or http://www.icsi.berkeley.edu/~mico/.

[5] T. J. Mowbray and W.A Ruh: Inside CORBA: Distributed Objects Standards and Applications, Adison-Wesley, 1997.

[6] S. E. Hudson. CUP user's manual. Graphics Visualisation and Usability Center-Georgia Institute of Technologie. Availlable on http://www.cs.priceton.edu/~appel/modern/java/CUP/cup0.10.

[7] D. Badouel & A. Khaled. La programmation C et C++. Edition Hermes, 1993.