Introduction à CORBA

Georges Linares

IUP-GMI - Master

Université d'Avignon et des Pays de Vaucluse

1. Introduction à CORBA

1.1 Généralités

Contexte

Ces dernières décénies ont vues une évolution rapide du matériel et du logiciel. Cette évolution a notament porté sur les réseaux qui ont évolués à la fois quantitativement (performances) et qualitativement (conectivité).Ceci à conduit progressivement à une organisation moins centralisée des ressources informatiques.

Problèmes

Ces évolutions ont amené un certain nombre de problèmes :
- dispersion des ressources
- coexistence d'architectures hétérogènes
- applications figées, fermées, isolées
- accés, maintenance, développement difficiles
- faible réutilisabilité des composants logiciels

Comment exploiter les ressources réparties ?
- communication : les applications et les composants doivent pouvoir échanger de l'information
- interopérabilité : les composants ou les applications doivent pouvoir interagir, c'est à dire se piloter ou s'utiliser mutuellement.
- intégration: les entreprises disposent d'un capital logiciel et d'un ensemble de ressources hétérogène. Comment les intégrer à une structure globale, unifiée qui permettent leur exploitation ?

Solution CORBA : plate-forme pour l'intégration d'objets distribuées.

Addresses : omg.org, linas.org/linux./corba.html

1.2 Historique

CORBA : Comon Object Request Architecture (Architecture à objets distribués).
C'est un standard developpé par l'OMG (Object Management Group). L'OMG est un consortium composé des acteurs majeurs de l'informatique (IBM, SUN, DEC, etc..).

1989 : Naissance de l'OMG (11 membres)
1990 : première publication (guide OMA : Object Management Architecture)
1992 : CORBA 1.1 (200 membres)
1994 : CORBA 2.0 (400 membres)
1996 : Définitions des services communs

1.3 Objectifs

Le but de l'OMG était de définitir d'un standard largement reconnu qui permette l'interopérabilité et l'intégration de composants hétérogènes. La diversité des composants

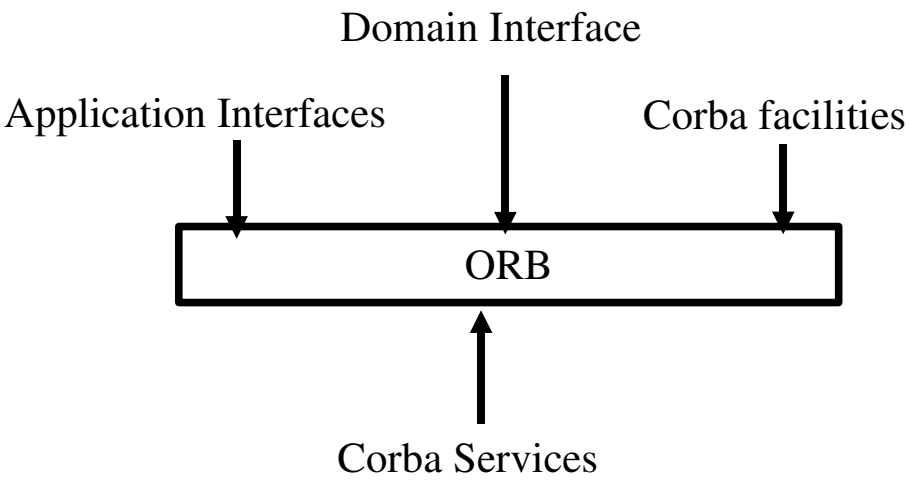
porte à la fois sur leur technologie de développement (langage de programmation, compilateur, système hôte...) et sur le contexte d'exécution (architecture matérielle de la machine hôte, technologie réseau, système d'exploitation dasn lequel un composant est implanté, ...).
Plus qu'une technologie permettant l'interopérabilité, l'ambition de l'OMG était de proposer un environnement complet, incluant infrastrucure et services.

1.4 CORBA : principes

L'OMG ne fournit que des **spécifications** : aucune implémentation de CORBA n'est proposée. Ces spécifications sont très précises et complètes. Elles doivent garantir la standardisation des produits proposés par les différents éditeurs.
CORBA est basé sur la **technologie objet**. L'ensemble des développements réalisés dans un environnement CORBA doit se faire dans cet esprit, bien que d'autres technologies puissent être intégrées.
CORBA repose sur une **architecture client/serveur** et un modèle de communication de type RPC. Cependant, CORBA se veut très universel. D'autres modèles de communications peuvent être instanciés dans une architecture CORBA, notament la **communication par messages**.
Un point essentiel est l'**indépendance** de la plateforme aux contextes de developpement et d'execution des applications.

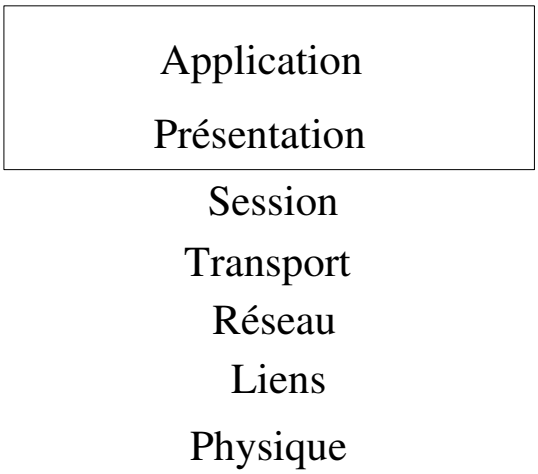
1.4 Le modèle Objet OMA

C'est la description abstraite d'une architecture objet.



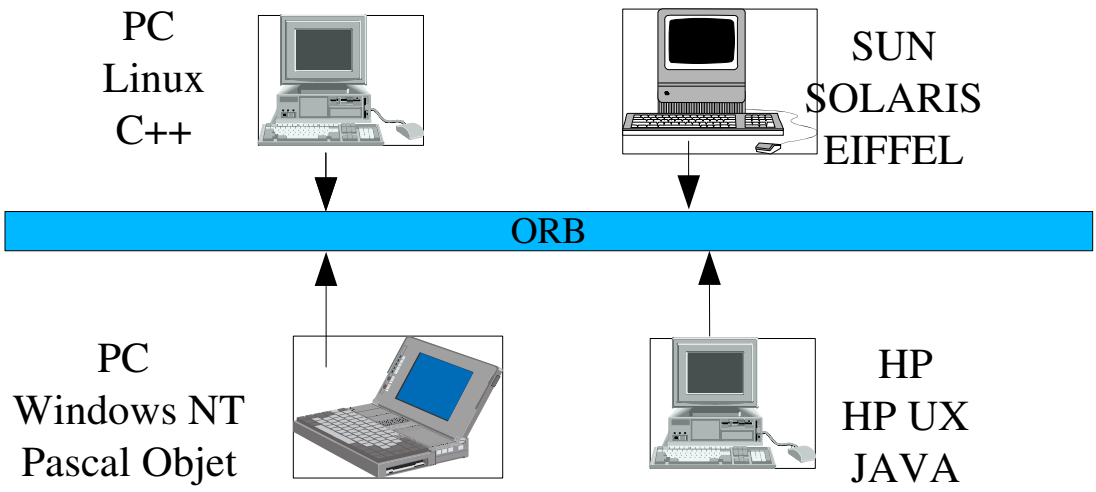
ORB (Object Request Broker) : bus objet. C'est le noeud central de l'architecture. Toutes les informations échangées entre les différents composants transitent par l'ORB.
Services : cycle de vie, persistance, nomage, événements, sécurité, licences, etc.
Interfaces de domaines : objets dédiés à un domaine applicatif (santé, finance, etc..)
CORBA Facilities : gestion des processus, des interfaces utilisateurs, des systèmes)
Application Interfaces : applications.

1.5 CORBA dans le modèle OSI



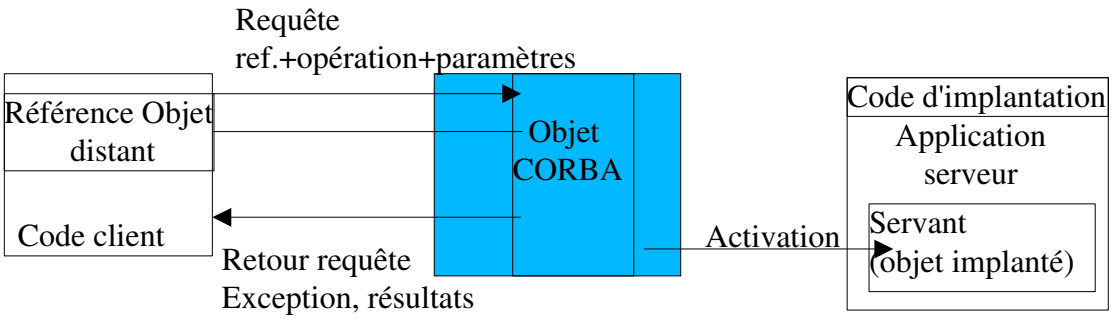
1.6 L'ORB

L'ORB permet la communication entre objets clients et serveurs, quelquesoit le langage d'implémentation, des objets, leurs environnements respectifs, leur position et leur état dans le réseau. L'ORB masque l'ensemble de l'architecture sous jacente (réseau, système,etc..).



1.7 Les requêtes

Les objets communiquent par l'intermédiaire de l'ORB en mode client-serveur, en envoyant ou en répondant à des requêtes. La requête est le mécanisme d'invocation d'une opération sur un objet. Elle contient une référence à l'objet, l'opération à effectuer et les parmaètres de cette opération. La reponse à la requête contient le résultat de l'opération (éventuellement l'information relative à une éventuelle erreur).



Le code client peut être écrite dans un langage de programmation quelconque et exploiter des objets distants. Un composant n'est pas intrinsecement client ou serveur. Dans un scénario de communication entre deux objets, à un moment donné, l'un joue le rôle de client (il émet une requête), l'autre joue le rôle de serveur (il répond à une requête).
La **Référence** est une structure désignant l'objet CORBA. Elle est contenue dans l'application cliente, et permet au bus d'identifier et de localiser l'objet. La référence est construite à partir d'une interface et d'un objet CORBA.
L'interface définit un type abstrait d'objet CORBA (atributs, méthodes). Elles spécifie le format de chaque méthode. Les interfaces sont écrites dans un langage de spécification définit par l'OMG (IDL :Interface Definition Language)
L'**objet CORBA** est l'objet qui reçoit les requêtes émises par l'application cliente, dans laquelle il est identifié par une référence. L'invocation de l'objet provoque l'activation de l'objet d'implantation associé, qui traitera effectivement la requête elle même.
Le **processus d'activation** associe un objet d'implantation à un objet CORBA. Plusieurs objets CORBA peuvent être associés au même objet d'implantation.
L'**objet d'implantation** code, à un moment donné, un objet CORBA. C'est lui qui contient la définition de l'objet.
L'application serveur est le processus hôte hébergeant les objets d'implantation.

1.8 Invocations statiques et dynamiques des méthodes

La communication inter-objet se fait par invocation de méthodes d'objets éventuellement distants. Les invocations sont transparentes : il n'y a pas de différentes visible entre invocations locales et distantes. Le langage dans lequel les objets ont été implementés n'est pas apparent. Ces invocations peuvent être **statiques** ou **dynamiques** :

- **statiques** : l'invocation est spécifiée à la compilation; les informations de type, la nature de opérations sont spécifiées et vérifiées lors de la compilation. Les spécifications sont écrites en IDL, puis traduites par un compilateur IDL en interfaces dans un langage de programmation (par exemple c++, Java, eiffel..).
- **dynamiques** : les interfaces de communication sont spécifiées pendant l'execution, par l'intermédiaire d'un référentiel des interfaces. Ce RDI contient des meta-données sur les interfaces d'objets, i.e. des descriptions de descripteur d'objets que sont les interfaces. De cette façon, toute application peut interroger le bus pour connaître la définition des objets.

Ce mécanisme très souple permet l'intégration et l'exploitation d'objet qui sont spécifiés seulement en cours d'execution.

1.9 Protocoles inter ORBs

CORBA propose un ensemble de spécifications, mais aucune technologie n'est imposée pour l'implémentation de l'ORB elle-même. Pour que différentes implémentations puissent interagir, un ensemble de règles de communication entre implémentations du bus sont définies, notament :

-**GIOP**: Général Inter Orb Protocol: fonctionne comme une surcouche de n'importe quel prototcole de transport. GIOP contient plusieurs formats de messages pour supporter l'ensemble des sémantiques possibles des appels et des réponses des méthodes distantes.

GIOP utilise un format de représentation des données universel (*CDR : common Data Representation*) et transportable à travers les réseaux.

-**IIOP** : Internet Intr Orb Protocol (istanciation de GIOP sur TCP/IP)

-**ESIOP** : protocoles pour les environnments spécifiques (DCE).

2. Le Langage IDL

2.1. Principe

IDL est un langage purement déclaratif dont la vocation est de décrire l'ensemble des opérations disponibles pour un objet, et comment elles peuvent être invoquées. Il permet de définir des contrats entre fournisseurs de sevices et clients. Ces contrats sont écrits sous forme de spécifications en IDL, puis traduits dans un langage de programmation classique (mapping). Ces spécifications sont universelles; elles sont indépendantes du langage cible, de l'environnement de développement, et de l'implémentation de l'ORB; la dépendance au contexte du développement apparait après compilation des sources IDL.

2.2. Généralités

- langage modulaire, fortement typé
-syntaxe proche du C++
-langage purement déclaratif :
pas de définitions, de fonctions, etc..

Une source IDL contient de déclarations :

- de types
- de constantes
- d'exceptions
- d'interfaces
- de modules

2.3 Structure d'une spécification IDL

Exemple :

```
// fichier exemple.idl
module finance {
 interface compte {
 readonly attributes string titulaire;
readonly attributes float balance ;
void  depot(in float mountant, out nouveau_credit) ;
}
;
interface action {
readonly attributes string nom ;
attributes float valeur ;
}
} ;
```

Une spécification est un ensemble de modules contenant des définitions.

2.4. Les modules

Le module est un espace de définitions. Il permet de regrouper des groupes cohérents de définitions, et limite les problèmes de conflits de noms et de localisation d'une définition. Les

déclarations faites dans un module sont accessibles par un opérateur de résolution de portée .
Exemple:
finance : :compte // fait référence à l'interface compte du module finance
Une source IDL peut contenir un module définit dans différents segments de la spécification.

2.5. Interface

Une interface décrit les données et les traitements associés à un objet distribué. L'interface contient toute les informations nécessaires à l'exploitation de l'bojet par une application cliente quelconque.

2.5.1 Les attributs

Les attributs sont les variables implémentées par l'objet. L'écriture des variables peut être limité en préfixant la déclaration par *readonly* (par defaut*, readwrite).*
Attention : les attributs ne sont pas des variables d'états !

2.5.2 Les méthodes

Syntaxe

La syntaxe des spécifications des méthodes fournies est semblable à celle de C++.
<type retour> <id>(<liste des paramétres>);

Modes de passage des paramètres

La définitions des paramètres d'une méthode doit impérativement s'accompagner d'une spécification du mode de passage des paramètre :

in : entrée
out : sortie
in/out : entrée/sortie

Les exceptions

L'interface IDL doit spécifier les exceptions qui peuvent être générée par l'objet. CORBA supporte deux types d'exceptions : les exceptions systèmes et les exceptions utilisateurs. Les exceptions systèmes sont définies implicitement ; les exceptions utilisateurs doivent être définies dans les modules IDL avant d'être émises.

Définition d'une exception :
exception <id > { attributs } ;
<déclaration méthode > raises (<id>)

Exemple :
exception retrait_interdit { string motif; } ;
void retrait (in float montant, out float solde) raises (retrait_interdit) ;

Modes d'invocations

Par défaut, toutes les opérations sont synchrones : le client attend la réponse de l'objet pour poursuivre son exécution. L'IDL permet de rendre une opération asynchrone, i. e. non blocante pour le client :
oneway <déclaration méthode>
Exemple :
oneway void as_retrait(in float montant);

Une fonction asynchrone ne peut avoir avoir de paramètres out ou inout, et ne peut pas

retourner de valeur. La transmission des exceptions n'est pas garantie.

Contexte
IDL permet d'associer des informations contextuelles à une opération ;
Exemple :
void retrait(in float montant, out float solde) context (»sys_time ») ;

2.5.3 L'Héritage
L'IDL supporte l'héritage simple et multiple.
Syntaxe C++ ;
Exemple :
module finance {
interface compte{} ;
interface compte_courant : compte{} ;
interface compte_epargne : compte {} ;
interface compte_rem : compte_courant, compte_epargne {} ;
}
Les conflits de noms sont interdits à l'intérieur d'un espace de définitions ; il n'y a pas de surcharge de méthodes. Les problèmes d'héritages répétés sont gérés directement par le compilateur ; ils ne provoquent pas de conflit de nom ni la duplication d'attributs.
Toutes les interfaces IDL héritent implicitement d'une interface *object* (objet générique).

2.5.4 Définition différée
Il est possible d'annoncer l'existence d''une interface avant sa déclaration :
interface compte ;
interface client{} ;
interface compte {} ;

2.6. Les types

Les types simples

Short	16 bits signé
Unsigned short	16 bits non-signé
Long	32 bits
Unsigned long	32 bits non signé
long long	64 bits signé
float	IEEE simple précision
Double	IEEE double précision
Long double	Float 128 bits
Char	charactères 8bits iso
Boolean	TRUE ou FALSE
octet	Valeur 8 bits non-convertie

2.6.2 Les type complexes

Enumérations

module finance {
enum monnaie {livre, franc, dollard} ;
interface compte{
readonly attribute monnaie monnaie_cpt ;
}

Structures

module finance {
struct individu{
string nom ;
short age ;
}
interface compte{
readonly attribute individu titulaire ;
}

Séquence

IDL permet de déclarer des séquences d'objet quelconques. Comme les strings, les séquences peuvent être bronées ou non-bornées :

- bornées : *sequence<compte, 500> comptes* ;
- non-bornées : *sequence<compte> comptes* ;

String

Deux formats suivant le type de déclaration

- bornée : la taille maximale de la chaîne est spécifiée (Ex : *attribute string passe<10>*),
- non bornée : *attribute string adresse;*

Tableaux

- Les tableaux peuvent être multi-dimensionels, mais ont toujours une taille fixée.
- *compte comptes[500]* ;

Les paramètres des méthodes peuvent être de ce type, à condition d'avoir étés redéfinis par un *typedef.*

2.6.3 Définitions de types et constantes

Identique au C++ :
typedef compte tab_compte[500] ;
const long max_comptes 1000 ;
Pas de constantes de type octet

2.6.4 Les types dynamiques

Les types dynamiques permettent la définition de fonctions dont les types seront fixés dynamiquement.
any : type de donnée générique ; une donnée de type *any* code une information sur la valeur de la donnée, mais aussi une information sur le type de cette donnée.
Ex :
interface test{
Void op(in any data) ;
}
L'extraction et le codage d'une variable se fait par redéfinition des opérateurs d'affectation à droite (<<=) et à gauche (>>=).
Le type d'une donnée est considéré comme un pseudo-objet de type *typecode*. Ce mécanisme permet de définir dynamiquement un type à partir des types de base et d'un certains nombre de paramètres.

2.7 Compilation IDL

Directives de compilation (type C++):

#ifdef, #else, #endif, #include

Compilation :

omniidl -bcxx <nom.idl>

omniidl -bxx -Wbexample <nom.idl> // génère un executable d'exemple

- traduit les définitions idl dans le langage cible (c++)
- génère les souches de communication (fihciers <id>SK.cc, <id>.hh)
  (partie de l'infrastructure de communication dépendante de l'implémentation)

omniorb:

v3.2- produit developpé à ATT; mapping C++, python. Licence GPL

www.uk.research.att.com

3. Mapping C++

3.1 Principe

L'IDL permet de définir les interfaces des composants distribués du l'application, mais il est impossible d'invoquer directement une opération à partir de sa seule description IDL Le code IDL est projeté dans les langages natifs de l'application cliente (coté client) et de l'application serveur (coté serveur). La projection des spécification est ensuite inséré dans le code de l'application elle-même. Les règles de projections sont définies par l'OMG.

3.2Règles de projections

Modules

- namespace
- convention de nomage : les définitions sont préfixées par le nom du module
- encapsulation dabs des classes sans attributs ni opérations

Interfaces

Les interfaces sont traduites en plusieurs classes C++ :

- la classe d'interface elle-même (<id>),

- la classe _var (<id_var>): c'est une classe de pointeurs sur <id>, dont la gestion mémoire est auomatisée,

-la classe _ptr (<id>_ptr) : classe de pointeurs sur <id>.

Les classes projetées dérivent toutes d'un classe abstraite CORBA::Object.

Héritage

Les liens d'héritage des classes de base vers leurs dérivées sont projetés dans une hiérarchie de classes C++.

Exemple:

```
class compte{...
void fixe_decouvert_maxi(in float val);
};
```

Dans le client :

```
main(){
compte_courant_var cpt;
    cpt->depot(1000.00);
}
```

Cette **projection des liens d'héritage** est faite pour toutes les classes générées par le compilateur IDL. Exemple:

compte_courant dérive de *compte*

compte_courant_ptr dérive de *compte_ptr*

compte_courant_var dérive de *compte_var*

Généralisation

Les convertions des classes dérivées vers les classes de base utilisent un mécanisme de généralisation (*widening*) mis en oeuvre implicitement :

compte_courant_ptr -> compte_ptr

compte_courant_var -> compte_ptr

Il n'y a pas de conversion implicite entre les types *_var*. Ces conversions doivent être faites par l'utilisation de la fonction membre *_duplicate()* :

```
compte_courant_var ccVar;
ccvar = ....                    //récupération des références
compte_var  cvar;
cvar = ccvar;                   //interdit
cvar = compte_courant_var::_duplicate(ccvar);
```

Le mécanisme de généralisation nécéssite une gestion, au niveau de l'objet, des références à l'objet.

Spécialisation

Les conversions des références de classes de base vers les classes dérivées ne peuvent pas être faites implicitement; elles utilisent un mécanisme de spécialisation (narrowing).

Ex :

```
compte_ptr  aptr;
compte_courant_ptr  captr;
aptr = ...              //récupération des références
captr  = compte_courant::_narrow(aptr);
```

La spécialisation (narrowing) est le mécanisme inverse de la dérivation (widening) définie implicitement. La conversion d'un _ptr vers un _var transfère la propriété de l'objet vers la variable automatique.

```
compte_courant_ptr cptr = ...;
compte_courant_var cvar = cptr; // transfert de propriété
```

Les attributs

Les attributs sont projetés en variables membres. Lorsque l'attribut est readwrite, deux fonction d'accés sont insérés; ces fonctions portent le même nom que l'attribut, et servent à lire ou à écrire la variable. Si l'attribut est en readonly, seule la méthode de lecture est insérée. Des fonctions d'accés aux attributs sont automatiquement générées par le compilateur IDL; ces fonctions portent le nom de l'attribut; la fonction d'accés en lecture n'a pas d'argument; l'a fonction d'accés prend comme argument une valeur à écrire dans l'attribut.

Les opérations

Les opérations sont projetées en méthodes virtuelles de la classe d'interface. La liste des paramètres contient, en plus de ceux définis dans la source IDL, des paramètres pour le support de la directive context et éventuellement des exceptions. Les types de paramètres sont projetés dans les types standards CORBA. Un paramètre est ajouté aux fonctions pour le support des exceptions.

Exemple :

interface compte{...

void depot(in float val, out float solde);

};

classe compte {

void depot(CORBA::float val, CORBA::float &solde);

}

Mode de passage des paramètres

L'utilisation de pointeurs ou de références en paramètres de méthodes distantes n'a pas de sens : les adresses des données coté client ou coté serveur sont relatives aux espaces d'adressages locaux.

Le passage d'une référence doit donc s'accompagner du passage du bloc de données correspondant. Ceci est problèmatique pour les données de taille variable (séquence, strings, tableaux,etc..).

Paramètres in :

L'ORB effectue une copie de l'objet passé en argument (passage par valeur)

Paramètres inout/out :

Les données sont dupliquées du client vers le serveur, puis du serveur vers le client. La taille des données de retour n'est pas forcement identique à celle d'entrée. Ce sont les mécanisme spécifiques de chaque type qui gèrent ces situations.

La projection des types

Types de base

IDL	***C++***	***C++***
short	CORBA::Short	CORBA::Short_out
long	CORBA::Long	CORBA::Long_out
unsigned short	CORBA::UShort	CORBA::UShort_out
u	CORBA::ULong	CORBA::ULong_out
float	CORBA::Float	CORBA::Float_out
double	CORBA::Double	CORBA::Double_out
char	CORBA::Char	CORBA::Char_out
boolean	CORBA::Boolean	CORBA::Boolean_out

IDL	***C++***	***C++***
octet	CORBA::Octet	CORBA::Octet_out
any	CORBA::Any	CORBA::Any_out

Des types références sont généralement définis :

CORBA::Short_out

CORBA:Long_out

.....

Types Complexes

***enum** : c++ plus entier 32bits*

Ex:

```
//idl
enum couleur{bleu, vert}
//c++
enum couleur{bleu, vert, IT_ENUM_Colour=CORBA_ULONG_MAX}
```

struct : structures C++
string : tableau de charactères terminés par un \0. Les string bornées et non-bornées sont une sur-définition du type char *. La vérification des tailles des espaces allouées est faite automatiquement par l'Orb. Cela ne concerne pas les utilisations locales des variables (pas de vérification en C++) :

```
char *string_alloc(ULong len);
char *string_dup(const char *);
void string_free(char *);
```

-surcharge des opérateur de recopie, allocation, libération, etc. dans les classes *String_var, String_out.*

Séquences

Les séquences sont projetées dans des strucures composées d'un tableau de données, d'une taille maxi du tableau et d'une taille courante. Les séquences non-bornées sont des tableaux bufferisés : une taille est initialement fixée; tout dépassement dynamique de la taille initiale provoque une réallocation du tableau. Il est possible de contrôler la taille du buffer.

Exemple :

```
//IDL
typedef sequence<long> tablong;

//C++
class tablong {
public :
tablong();                      // construction max 0, taille 0
tablong( const tablong&);       // construction par recopie
tablong(                        // construction sur un buffer externe
CORBA::ULong max,
CORBA:ULong lenght,
CORBA::Long *data,
CORBA::Boolean release =0);// possession du buffer
tablong(CORBA::ULong max);
```

static CORBA::Long allocbuf(CORBA:ULong nelems);*
*static void freebuf(CORBA::Long *data);*
CORBA::ULong maximum() const; // max
CORBA::ULong lenght() const // lecture
void length(CORBA::ULong) // écriture taille
.....// copie, [], etc..

Les séquences bornées fonctionnent de la même façon, mais l'attribut maximum ne peut pas être écrit (il fait partie de la définition du type).

Tableaux

Les tableaux sont projetés en C++ vers un tableau C++ classique.

Exemple :

// IDL

typedef long tabl[1];

// C++

typedef CORBA::Long tabl[10][3];

L'allocation et la libération des tableaux doit être faite par des fonctions membres spécifiques au type défini.

Ex :

tabl_alloc(CORBA:ULong)

tabl_free()

Mapping des exceptions

Les exceptions sont projetées vers des classes dérivées de *CORBA*::*Exception*:

- exceptions systèmes CORBA::SystemeException
  - définies dans le module CORBA
  - émises par les fonctions de la librairie standard ou dépendante de l'ORB
  - deux méthodes de classe :
    - CompletionStatus completed() (retour : COMPLETED_NO, COMPLETED_YES, COMPLETED_MAYBE)
    - void completed(CompletionStatus);
    - ULong minor(): détails suplémentaires dépendant de l'exception (TIMEOUT, DTRING_TOO_BIG,etc..)
    - void minor (Ulong)
- exceptions utilisateurs CORBA::UserException

4. Les adaptateurs d'objets

4.1 Introduction

- Intermédiaire entre objets CORBA et implémentations (servant)
- gestion des activations par un adaptateur d'objets
  - transparente & automatique
  - activations dynamiques

Les objets exportés par les serveurs ne sont pas tous constamment utilisés par les applications clientes. De façon à optimiser l'utilisation des serveurs, CORBA dispose d'un mécanisme d'activation dynamique liant objet d'implantation de objet CORBA. Ce mécanisme est transparent pour les applications clientes. Les activations sont réalisées par un adaptateur d'objets. L'A.O est une couche logicielle entre ORB et objets d'implantation (servants).

4.2 Fonctionnalités d'un adaptateur d'objets.

- gestion d'un référentiel des implantations
- génération et interprétation des références d'objets
- activation des objets d'implantation
- traitment des requêtes
- sécurité
- désactivation des implantations

4.3 Les différents adaptateurs d'objets

BOA (Basic Object Adaptator):standard inclus dans toutes les implantations du BUS CORBA. BOA fournit plusieurs stratégies d'activation.(sous-spécifié)
LOA (Library Object Adaptator) : prise en charge d'objets contenus dans une bibliothèque d'objets.
OODA (Object Oriented Database Adaptator) : adaptateurs de bases de données objets. Particularité : c'est la base qui fournit le mécanisme d'activation. Tous les objets peuvent êtres potentiellement actif sur le bus).
POA (Portable Object Adaptator) : nouveau standard; mieux spécifié, plus portable.

4.4 BOA

BOA gère l'activation d'objets contenus dans des exécutables serveurs. C'est une version ancienne et obsolète des adaptateurs.

Modes d'activation

Il y a quatre stratégies d'activation :

- serveur partagé (*sharedActivationMode*):
  - plusieurs activations simultanées dans un même processus. Le serveur est globalement activé à la première requête destiné à un de ses objets. Les requêtes sont ensuite directement adressées au processus serveur. Il n'y a qu'un processus serveur actif en même temps.

- serveur persistant (*persistentActivationMode*) :

  - identique au précédent mais pas d'activation automatique. Ce type de serveur doit être executé manuellement.

- Serveur non-partagé (*nonsharedActivationMode*): chaque objet réside dans un processus qui ne contient que cet objet. La première invocation de l'objet active le processus, qui traitera toutes les requêtes.

- Serveur par méthode (*perMethodeActivationMode*) : à chaque invocation, un processus est crée et détruit lorsque la requête est traitée.

L'Interface BOA

Le BOA est un pseudo-objet décrit par une interface

Initialisation :

CORBA : :ORB_ptr orb = CORBA : :ORB_init(argc,argv, »Orbix ») ;
CORBA ::BOA_ptr boa = orb->BOA_init(argc,argv, »Orbix_BOA)
Argc, Argv : options d'initialisation

Méthodes de gestion des références (idl) :
Create/dispose : création & destruction des références

create (in ReferenceData id, // identificateur opaque de l'objet
In InterfaceDef_ptr intf,//objet du référenciel des interfaces
In ImplementationDef impl) ; // reference de l'implementation (nom de serveur) ;
void dispose(in Object obj) ;

*void Impl_is_ready(in ImplementationDef impl) ; //*Enregistrement du serveur
void desactivate_impl(in Object obj);// Désactivation de l'implantation.
obj_is_ready et desactivate_obj : idem mais pour les serveurs non-partagés.

4.5 POA (Portable Object Adaptator)

Objectifs :
- liens entre objets CORBA et Servants
- gestion des activations/desactivations
- implémentations portables
- support des objets persistants
- gestion extensible du comportement des objets (policy)

Composants :
client : application cliente
server : application serveur
servant : objet serveur
id :
- identificateur sur un objet corba. (ça n'est pas un objet d'implantation)
- géré par l'adpatateur ou l'implémentation

– cachés aux clients

objet POA : objets contenant des id, définissant un comportement (objet policy) pour les objets implémentés
objet policy : comporemtent de l'objet
POA manager : gestion de l'ensemble des objets POA

Association ID/Servants
Principe
Le client référence un objet serveur par un Interoperable Objets Reference (IOR)
IOR : information sur le serveur, l'hote, l'adpatateur d'objets.
Le POA utilise l'ID pour trouver le servant.
Il envoie les requêtes aux servant (d'abord traitée par le code du squelette).

Un servant peut incarner un ou plusieurs objets en fonction de ses policies.
Il peut etre activé plusieurs fois sur differents servants

Mapping Objet/Servants
Principe :
Coté client,l'objet est manipulé par une référence qui contient une IOR (Interoperable Objets Reference). L'IOR cotient l'information nécéssaire à l'identifiaction de l'objet :
– addresse du server
– l'objet POA gérant le servant
– l'object Id
– Coté serveur : l'object Id est utilisé par le POA pour trouver le servant

Options d'activation :
– active object map : le POA préserve le lien objet/servant tant qu'il n'y a pas de desactivation explicite, ou de destruction du POA.
– Servant manager : permet d'activer à la demande :activation par methode, ou des la première invocation
– default servant : un objet pour traiter toutes les requêtes reçues par le POA

Le comportement de l'adaptateur peut être redefini de façon a combiner les stratégies d'activation

Création d'un POA

POA racine : *resolve_initial_references(„POA“);*
Il s'agit d'un adapatateur dont le comportement est fixé, systématiquement crée dans l'application serveur.
Coexsitence de plusieurs POA :
– regrouper les servants dans differents groupes de comportements identiques
– contrôle des requêtes pour un ensemble d'objets (ouverture /fermeture des services)
Création en 4 étapes :
– définir les policies
– création du POA lui-même (create_POA() sur un POA existant)
– Si policy USE_SERVANT_MANAGER, initialisation du servant manager
– démarrage du POA par activate() sur son gestionnaire (poa_manager)

Définition des POA Policies

policy :

- objets du module CORBA
- attachés à un POA
- définisant le comportment commun à tous les servant attachés au POA

Exemple :

CORBA::PolicyList policies; // Séquence d'objets policy

Rétention

create_servant_retention_policy() : liens par l'intermédiare d'un active objets map qui maintient eventuellement la liasion

- RETAIN : le POA préserve les servants actif dans l' *Object Active Map*
- NON_RETAIN : le POA n'a pas d'OAM; les liens sont recréer à chaque requête, et détruits ensuite. Il doit y avoir un mécanisme de liaison alternatif (USE_DEFAUL_SERVANT,USE_SERVANT_MANAGER)

Requête

create_request_processing_policy():

USE_ACTIVE_OBJECT_MAP_ONLY : utilisation d'une structure décrivant les liens servants/Ids

USE_SERVANT_MANAGER : utilisation d'un gestionnaire de serveurs

USE_DEFAULT_SERVANT : utilisation du serveur par défaut

Cycle de Vie

create_lifespan_policy() : détermine la durée de vie des objets

TRANSIENT (defaut) : les références ne survivent pas au POA; le servant est attaché au processus serveur.

PERSISTENT : les références restent valides après la destruction de l'adaptateur d'objets et du processus serveur; le referentiel des implémentation est utilisé pour maintenir une image virtuelle valide de l'impélementation.

Affectation des Identificateurs

create_activating_polivy()

- SYSTEM_ID : assignation automatique
- USER_ID : utilisateur (souvent utilisé pour lesobjets persistants)

Unicité des liens

create_id_uniqueness_policy()

UNIQUE_ID: un servant, un unique objet

MULTIPLE_ID : un servant, plusieurs Id.

Activation

create_implicit_activation() :

IMPLICIT_ACTIVATION : activation implicite; l'id doit être affectée automatiquement (SYSTEM_ID), et la retention du lien à RETAIN.

L'objet est activé par l'invocation de la méthode *_this*(). *_this* réalise deux tâches :

- interrroge le POA si le servant existe et est connecté à un Id. Si ca n'est pas le cas, , il active l'objet et l'enregistre dans l'active object map.

– Génère une référence

NO_IMPLICIT_ACTIVATION :
utilisation de *activate_object()* ou de *activate_objetc_with_id()*
Deux moyens d'obtenir une référence :
– *id_to_references()*
– *_this*()

5- Développement d'une application distribuée sous omniorb

5.1 Principe

Au moins 3 cas de figure :
- utilisation de serveurs compatibles CORBA
- encapsulation de code non-corba des des objets CORBA
- dévéloppement complet

5.2 Intégration d'objets CORBA

- récupération des interfaces IDL
- recompilation éventuelle (projection des interfaces vers le client)
- développement éventuel d'autres serveurs
- développement du client :
  - récupération des classes d'interfaces
  - programmation du code client
  - compilation/édition de liens

5.3 Encapsulation

- principe :
  - utiliser des composants logiciels pré-existant sans les modifier
  - construire une classe CORBA qui invoque localement l'objet

Exemple :

Classe existante

```
// -------- pile.hpp --------
const unsigned int MAX= 10;
class Pile{
 private :
        unsigned int taille;
        unsigned int pile[MAX];
 public :
        Pile() {taille=0;};
        void empile(unsigned int);
        unsigned int depile();
};

// -------- pile.cpp --------
#include "pile.hpp"
void Pile::empile(unsigned int i){
 pile[taille++]=i;
}
unsigned int Pile::depile(){
 return pile[--taille];
}
```

Interface IDL :

interface DPile {
void empiler(in long e);
long depiler();
}

Projection :

class DPile;
class _objref_DPile {
...
void empiler(CORBA::Long e);
CORBA::Long depiler();
...

class _impl_DPile{
...
virtual void empiler(CORBA::Long e) = 0;
virtual CORBA::Long depiler() = 0;
...
}

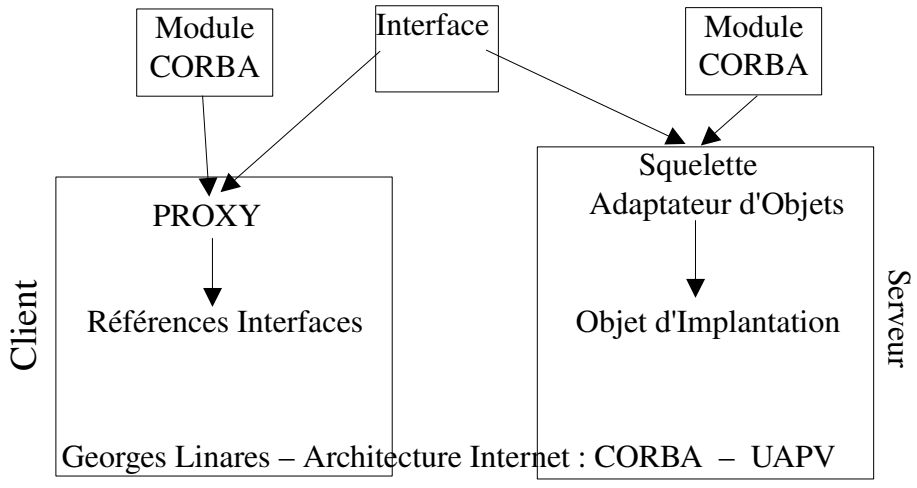
Implémentation :

class DPile_i {
*Pile *p;*
...
void empiler(*CORBA::Long e) { p->empiler((CORBA::Long) e);}*
CORBA::Long depiler() {return (CORBA::Long) p->depiler();}
...
}

5.4 Développement sur le middleware

- modélisation
- définition des interfaces
- compilation IDL (génération des souches de communication)
- implémentation des objets serveurs
- écriture du client
- compilation (liens aux libraires CORBA, réseaux, etc.)

Classes et communication en mode statique



Proxy :

image locale de l'objet distant
intermédiaire de toute communication à travers l'orb
manipulé implicitement par des références

Squelette :

super-classe de la classe d'implémentation
connection directe à l'ORB ou utilisation d'un adaptateur d'objet

5.5 Développement coté Serveur

1- initialisation de l'ORB

CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, « omniorb3 »);

arg 3 : identifant de l'orb
arrg 1 et 2 : passage de paramètres à l'orb
Exemple -ORBid 'omniorb3'
Echec : ptr nil en retour
Lecture du fichier de onfiguration oniorb.cfg

2 - initialisation de l'adaptateur d'objets (ici POA)

CORBA::Object_var obj = orb->resolve_initial_references(« RootPOA »);
PortableServer::ObjectId_var poa = PortableServer::POA::_narrow(obj);

3 - Création de l'objet serveur (instanciation de la classe d'implémentation)

DPile_i mapile = new Dpile();*

4 - Acivation de l'objet

PortableServer::ObjectId_var mapile_id = poa->activate_object(mapile);

5 - Récupération d'un identifiant de l'objet

- information sur l'objet, l'hôte, le port, le process, etc.
- utilisable coté client pour identifier l'objet distant

CORBA::Object_var = mapile->_this();
CORBA::string_var id_obj(orb->object_to_string(obj));
*cerr << (char *) id_obj;*

6 - Destruction de la référence

mapile->_remove_ref();

7- Récupération et activation du gestionnaire POA

PortableServer::POAManager_var pman=poa->the_POAManager();
pman->activate();

8- Déconnection :

orb->run();
orb->destroy();

7- Les services

7.1- Principe

Fonctionnalités fréquement utilisées dans les applications distribuées

Normalisés par l'OMG (1996)

Regroupés en packages COSS (Common Object Specification COSS)

7.2- Les packages de services

COSS-1 : nommage, événnements, persistance, cycle de vie

COSS-2 : transactions, concurrence d'accés, externalisation, relations

COSS-3 : sécurité, temps

COSS4 : licences, propriétés, requêtes

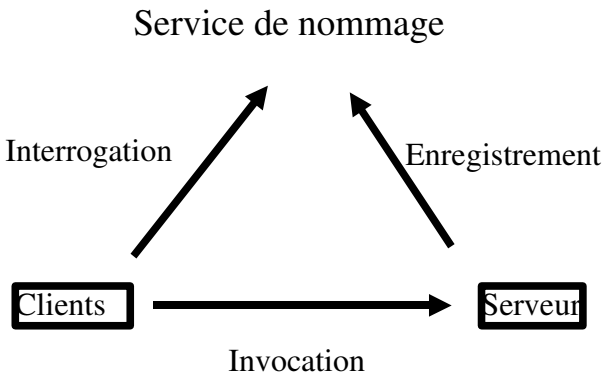
COSS5 : vendeur, collections

Tous les services ne sont pas nécéssaires présents dans toutes les imlémentations

7.3- Les fonctionalités

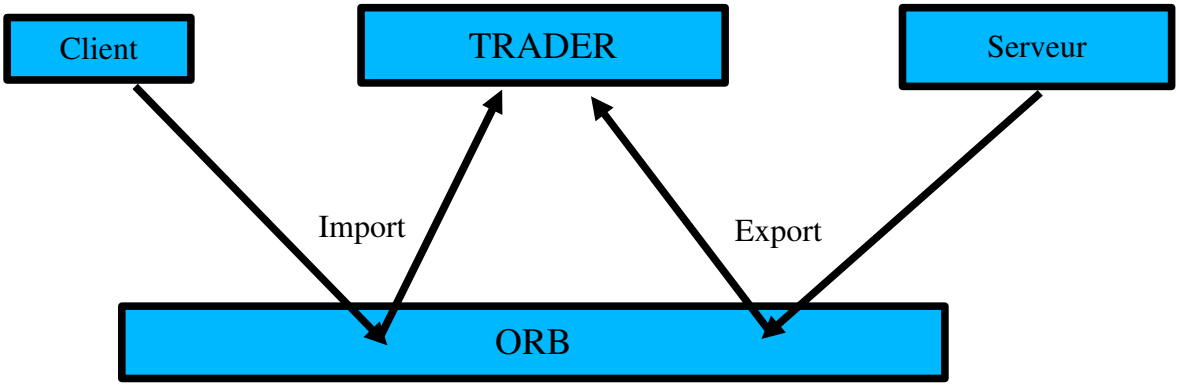
Services de nommage

But : identifier un objet par une étiquette intelligible



Service vendeur (trader)

pages jaunes : les objets sont identifiés par certaines de leur fonctionnalités



Service événnements(Event service)

dépot/retrait d'événnements dans un canal d'évenennements partagé entre consommateurs et producteurs d'événnements

Service Propriété (Property service)

Modification dynamique des propriétés d'un objet (ajout de propriétés)

Adaptation du serveur aux besoins spécifiques d'un client

Pas de modification des interfaces IDL

Service cyvcle de vie (Life cycle service)

Création, duplication, déplacement, destruction d'objets

Utilisation d'une usine d'objets (*object factory*)

Manipulation de graphes d'objets

Externalisation (Externalization Service)

Echanges objets-flux

Codage de l'état d'un objet

Récuperation de l'état d'un objet

Mécanisme du type entrée/sortie c++ (<< , >>)

Persistance (Persistent Object Service)

Stockage des objets : base de données objets, relationnelles, fichiers

Utilisation d'une convention de codage des données :

- ODL : Object Definition Langage
- Dynamic Data Object : description universelle des données

Concurrence d'accès (Concurrency service)

Contrôle des accés simultannés :

- mécanisme de verrouillage des méthodes

- ordonnancement des requêtes
- sérialisation des requêtes

Service sécurité (security service)

Contrôle de l'accés aux serveurs

- identification/authentification des clients
- cryptage des requêtes
- contrôle des autorisations d'accés aux objets

Licence service

Gestion des licence d'utilisation d'objets serveurs

- distribution d'autorisation d'utilisation d'objets
- contrôle de l'usage, facturation

Le service de temps (time service)

Horloge partagée par les objets connectés au bus

- synchronisation
- alarmes
- calcul de durées

8. Le service de nommage

8.1- Objectif

Par défaut, l'identification et la localisation des objets serveurs se fait par l'intermédiaire d'IOR (références stringifiées). Le service de nommage est un annuaire “pages blanches”, i.e. qui permet :

- de trouver un objet à partir d'une clef connue
- un étiquetage intelligible des objets

8.2- Principe

- association nom-référence (*name-binding*)
- utilisation de contextes de nomages
- groupement d'objets par contexte de nomages
- standardisation des convention d'écriture des noms et des chemins d'accés (des contextes)
- pas d'interprétation des noms

8.3- Les contextes de nommage

-codage arborescent des contextes

- feuilles de l'arbre : les noms des objets
- 1 objet peut avoir plusieurs noms
- un nom ne peut être associé qu'a un seul objet

8.4- Les composants

Principe

Structure codant les noms : suite d'étiquettes de contextes, suivie d'une étiquette d'objet.

Exemple :

Calcul numérique	Traitement du signal	Acoustique	Objet Analyse temps/frequence

Codage : séquence de composants

Structure

- *type NameComponen*t
- Deux attributs :
  - *string id* : nom du composant
  - *string kind* : type de composant (libre)

Exemple :

Name mon_obj;

mon_obg.length(3);

*mon_obj[0].id= (const char *) « CalculNumérique »*

*mon_obj[0].kind= (const char *) « calcul et fichiers »*

*mon_obj[1].id= (const char *) «AnalyseDeDonnées »*

*mon_obj[1].kind= (const char *) «calcul »*

*mon_obj[2].id= (const char *) « AnalyseFactorielle»*

*mon_obj[2].kind= (const char *) « calcul»*

Les séquences peuvent être écrites sous forme de chaine (*stringified names*)

8.5- L'interface du service de nommage

Définitions dans le module *CosNaming:*

- méthode d'association nom-références
- structure de codage des contextes : *NamimgContext*
- structure de codage des noms : *Name*
- Méthodes du *NamingContext*:
  - enregistrer un nom (bind, rebind)
  - résoudre une association resolve
  - supprimer une association : unbind
  - lister le contenu du service de nomage : list
  - suppression d'un contexte : destroy

8.6- Le pseudo-objet *service de nommage*

- objet systématiquement connu
- récuperer un référence sur le service de nommage :
  - nom *: NameService*

```
// Obtain a reference to the root context of the Name service:
   CORBA::Object_var  obj = orb->resolve_initial_references("NameService");
// Narrow the reference returned.
   -     rootContext = CosNaming::NamingContext::_narrow(obj);
```

8.7- Enregistrement d'un objet

Méthode coté serveur : *bind*

void bind(in Name, in Object obj) raises (NotFound, Cannot Process, InvalidName, AlreadyBound)

8.8-Récuperation d'une référence

object_var resolve(in Name nom) (NotFound, Cannot Process, InvalidName);

8.9 Destruction d'une association

void unbind(in Name nom);

ref->unbind(name);

8.10- Création d'un nouveau context :

NamingContext bind_new_context(in Name)