

Systemes repartis : CORBA

Fabrice Rossi

<http://apiacoa.org/contact.html>.

Universite Paris-IX Dauphine

Systemes repartis : CORBA - p.1/45

CORBA

- LE standard pour les systemes repartis objets
- *Common Object Request Broker Architecture*
- norme produite par l'Object Management Group (<http://www.omg.org>) et X/Open
- version actuelle 3.0 (decembre 2002), mais en pratique 2.6
- construit a partir de l'Object Management Architecture (qui definit entre autre le modele objet utilise par CORBA)
- composants :
 - un modele objet
 - un bus logiciel pour faire dialoguer les objets (l'Object Request Broker, 1190 pages de specifications a lui tout seul !)
 - un ensemble de services (publication, activation, etc.)
 - un ensemble de *facilities* (groupes de services)

Remarque : terminologie un peu lourde...

Systemes repartis : CORBA - p.3/45

Plan

1. Architecture et modele objet
2. *Interface Definition Language*
3. Accs a l'ORB
4. *Naming Service*
5. Processus de developpement

Systemes repartis : CORBA - p.2/45

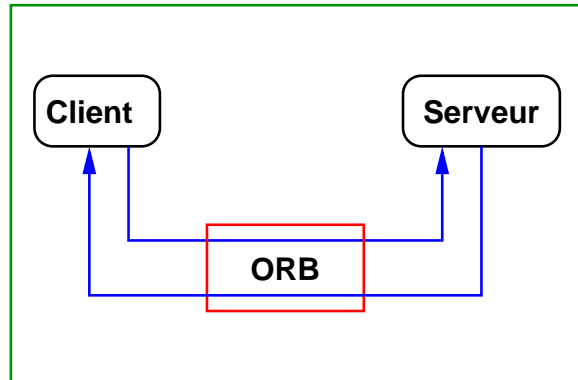
Principe du bus logiciel

L'ORB est un bus logiciel :

- tout appel depuis un client vers un serveur passe obligatoirement par au moins un ORB qui se charge de tout :
 - localisation et activation du serveur
 - independance par rapport au langage, au systeme et au hardware
 - transparence reseau
- les ORBs communiquent entre eux par le *General Inter-Orb Protocol*, tres souvent la version TCP/IP : *Internet Inter-Orb Protocol* (IIOP)
- communications client et ORB : *stubs* et API directe (appel a certains services)
- communications serveur et ORB : *skeleton*, *Object Adapter* et API

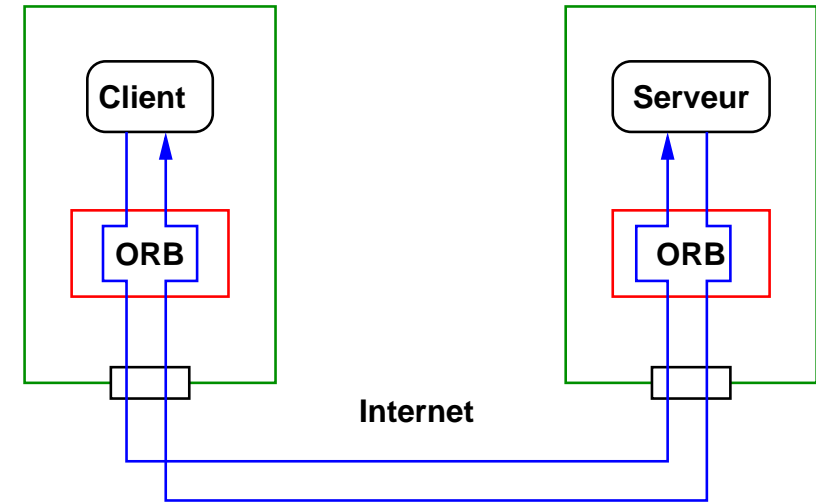
Systemes repartis : CORBA - p.4/45

Une seule machine



Systèmes répartis : CORBA – p.5/45

Deux machines



Systèmes répartis : CORBA – p.6/45

Organisation des spécifications CORBA

- un modèle objet
- un langage pour décrire les objets (IDL)
- une traduction de l'IDL vers les langages classiques (C, Java, etc.)
- des APIs :
 - décrites en IDL et associées à une sémantique précise
 - pour l'ORB (fonctions du noyau CORBA)
 - pour les *Object Adapters* (pour l'implémentation concrète des objets CORBA)
 - pour les services et les *facilities*
- des spécifications de plus bas niveau pour assurer l'interopérabilité entre les ORBs (GIOP et IOP par exemple)
- un modèle de composants (*CORBA Component Model*) de type entreprise Java Bean, mais indépendants du langage (depuis la version 3.0)

Systèmes répartis : CORBA – p.7/45

Le modèle objet de CORBA

Version "concrète" du modèle "abstrait" de OMA :

- un objet est une entité identifiable (par une **référence** (au sens CORBA)) qui propose une ou plusieurs **opérations**
- un client peut envoyer à un objet une **requête** destinée à une opération (ou service) de l'objet
- une requête peut avoir des **paramètres** :
 - des valeurs ou des références vers des objets
 - trois modes de passage : en entrée, en sortie ou mixte
- les objets et les paramètres des requêtes possèdent des **types** :
 - types de base classiques : entiers, réels, booléen et caractères ;
 - types de base plus évolués : chaînes de caractères et any (joker) ;
 - types évolués : *structs*, unions et tableaux ;
 - types objets : interfaces (abstraites ou non) et "valeurs".

Systèmes répartis : CORBA – p.8/45

Objets “valeurs”

- CORBA de base : objets manipulés par référence
- depuis 2.3, objets avec sémantique valeur (*value type*)
- attention, la sémantique de passage est liée au type
- un *value type* peut proposer :
 - des opérations
 - des données privées et/ou publiques
- problèmes subtils :
 - CORBA est multilingage : sémantique différente selon le langage ?
 - interface d'un *value type* : une forme de spécification
 - requête sur une opération : appel **local**

Aspect dynamique

- CORBA est très dynamique, comme C++ (RTTI) et Java (réflexion) :
- *Dynamic Invocation Interface* :
 - côté client
 - permet de faire une requête sans passer par les *stubs*
 - principe : création d'un “objet” *Request* et d'une liste de paramètres
 - *Dynamic Skeleton Interface* :
 - côté serveur
 - permet d'implémenter une interface sans passer par les *skeletons*
 - même principe que pour le client : *ServerRequest* et liste de paramètres

Aspect dynamique (2)

- *Interface Repository*
 - stockage persistant de définition d'interfaces par l'ORB
 - même principe que la réflexion Java
 - divers objets décrivent les interfaces et les types disponibles
- *Dynamic Any*
 - DII et DSI concernent les objets, *DynAny* correspond aux types classiques traités par le joker *any*
 - permet l'obtention du type réel de la valeur, l'extraction de valeur, la construction de valeur, etc.

Implémentation d'un objet

- Partie très subtile de CORBA :
- les langages cibles ne sont pas tous orientés objets : un objet CORBA n'est pas obligatoirement représenté par un objet
 - un domestique (*servant*) est un “truc” (un objet dans un langage OO) qui implémente un ou plusieurs objets CORBA :
 - l'idée de base est qu'une requête à un objet CORBA peut être traitée par un domestique alors que la requête suivante sera traitée par un autre domestique
 - de plus, un même domestique peut répondre à des requêtes correspondant à des objets différents
 - l'association objet/domestique est gérée grâce à un *Object Adapter*, en particulier le *Portable Object Adapter*
 - le POA propose en plus l'activation et la persistance des objets

Interface Definition Language

IDL :

- langage permettant de décrire les interfaces des objets CORBA
- purement déclaratif
- associé à une traduction officielle (*mapping*) vers la plupart des langages classiques (C, C++, Java, Smalltalk, etc.)
- syntaxe très inspirée du C++
- exemple :

```
1 module HelloApp {
2     interface Hello
3     {
4         string sayHelloTo(in string firstname,
5                             in string lastname);
6     };
7 };
```

Systemes répartis : CORBA - p.13/45

Définitions IDL

Un fichier IDL définit (ou déclare) :

- des types (principe des `typedef` du C++)
- des constantes (principe des `const` du C++)
- des exceptions
- des interfaces (principe des interfaces de Java)
- des modules (principe des *namespaces* du C++)
- des "valeurs" (types objets passés par valeur)

Systemes répartis : CORBA - p.15/45

Syntaxe IDL

Eléments de syntaxe :

- commentaires du C++ (et de Java)
- *preprocessing* strictement identique à celui du C++ (`#include`, `#define`, etc.)
- un seul espace de noms par portée
- *case sensitive*, mais un peu étrange :
 - si on utilise le nom `tot0`, on doit toujours l'écrire de cette façon
 - deux noms ne peuvent pas avoir pour seule différence la case des lettres qui les composent (i.e., si `toTo` est utilisé dans une portée, `TOTO` n'est plus permis)
- un fichier IDL est constitué de **définitions**, chaque définition se terminant par un point virgule

Systemes répartis : CORBA - p.14/45

Types de base

Les définitions s'appuient sur des types de base classiques :

- entiers signés ou non (`short` (2 octets), `long` (4 octets) et `long long` (8 octets))
- réels (IEEE `float`, `double`, `long double` ainsi qu'un type `fixed` correspondant à des réels en virgule fixe)
- caractères (`char` ISO-Latin 1, un octet, et `wchar` pour les autres formats type UNICODE)
- booléen (`boolean`)
- deux curiosités : `octet` (pas de transformation par l'ORB) et `any` (joker)

Systemes répartis : CORBA - p.16/45

Définition de types

- exactement le même principe qu'en C/C++
- enregistrements (`struct`), dont les définitions récursives (de type liste)
- `union`
- énumérations (`enum`), sans correspondance numérique
- séquences (`sequence`) : tableaux à une dimension (éventuellement à taille bornée)
- chaînes de caractères (`string` et `wstring`), éventuellement à taille bornée
- tableaux à plusieurs dimensions, toutes de taille fixe

Exemple

```
1 struct UserId {
2     wstring id;
3     wstring group;
4     wstring pass;
5 };
6 enum mode {Read, ReadWrite};
7 enum category {User, Group, Other};
```

Module

- `module nom {...};`
- les modules CORBA ont pour rôle d'organiser les autres définitions
- version CORBA des
 - `package` Java
 - `namespace` C++
- structure hiérarchique
- noms complets : principe du C++, séparateur `::`. Par exemple : `HelloApp::Hello`
- un module crée un espace de noms

Interface

- l'élément le plus important en IDL
- décrit entre autre un type objet CORBA
- contient :
 - des définitions de types, exceptions et constantes : rôle d'organisation (comme un module)
 - des définitions d'**opérations** et d'**attributs** : type objet
- la surcharge d'opérations est **interdite**
- une interface peut hériter d'une ou **plusieurs** interfaces (syntaxe du C++)
- les types, exceptions et constantes hérités peuvent être redéfinis

Interface (2)

- définitions d'opérations :
 - similaires aux méthodes de Java et C++
 - les paramètres peuvent être in (en lecture), out (en écriture) ou inout (les deux)
 - une opération peut lever une ou plusieurs exceptions, ce qui s'indique par une clause raises (similaire à throws)
- définitions d'attributs
 - équivalent à un couple d'opérations (lecture et écriture de l'attribut)
 - en lecture seule grâce à readonly

Exemple

```
Dictionary.idl
1 module Dictionary {
2     interface ReadOnlyDictionary {
3         wstring definition(in wstring word);
4     };
5     interface ReadWriteDictionary : ReadOnlyDictionary {
6         void removeWord(in wstring word);
7         void insertWord(in wstring word,in wstring definition);
8     };
9 };
```

Exemple (2)

```
DictionaryServer.idl
1 #include "Dictionary.idl"
2 module DictionaryServer {
3     struct UserId {
4         string id;
5         string group;
6         string pass;
7     };
8     interface Server {
9         Dictionary::ReadWriteDictionary create(in UserId user);
10        Dictionary::ReadWriteDictionary getReadWrite(in UserId user);
11        Dictionary::ReadOnlyDictionary getReadOnly(in UserId user);
12    };
13 };
```

Exceptions

- exception nom {... };
- traitement des erreurs
- principes similaires à ceux des exceptions en C++ et en Java
- une exception peut avoir un contenu, comme pour une *struct*
- exemple :

```
Erreur.idl
1 enum Level {warning, error, fatal};
2 exception AnError {
3     Level level;
4     string message;
5 };
```

Accès à l'ORB

Pour pouvoir écrire un programme CORBA (client ou serveur), il faut accéder au "noyau" CORBA, c'est-à-dire à l'ORB :

- l'ORB est une interface CORBA (décrite en PIDL) qui correspond à un pseudo objet CORBA
- pseudo objet : "truc" spécifié en pseudo IDL qui ne correspond pas toujours à un vrai objet CORBA (dépend du *mapping*) mais qui s'utilise en général comme un objet CORBA
- propose une opération de *bootstrapping*, *init*, qui permet d'obtenir une référence vers l'ORB (après l'avoir initialisé)

Object

- Object est la représentation d'une **référence** vers un objet CORBA
- l'interface définit diverses opérations utiles :
 - gestion de l'aspect dynamique :
 - obtention d'une description de l'interface de l'objet référencé
 - création d'un objet Request
 - manipulation de la référence (copie, association à un objet, implémentation d'une interface, etc.)
- Attention, ne pas trop comparer à Remote en Java : en général, on ne peut pas faire un *cast* Java d'un Object vers l'interface Java associé à l'objet référencé

Services de l'ORB

L'ORB propose :

- une résolution simple (*resolve_initial_references*) : associe des noms à des références importantes. Par exemple, les deux objets les plus importants :
 - NameService : le service de publication/découverte de CORBA
 - RootPOA : l'*Object Adapter* principal
- une conversion (bidirectionnelle) entre références et chaînes de caractères portables en ORB (*string_to_object* et *object_to_string*)
- une opération de connexion (*connect*) d'un domestique (*servant*) à l'ORB, ce qui a pour effet de rendre disponible une implémentation d'un objet
- beaucoup d'autres opérations, essentiellement destinées à l'aspect dynamique de CORBA (découverte d'interfaces, etc.)

Naming Service

Le service le plus important de CORBA :

- annuaire d'objets : association d'un nom à un objet (*name binding*)
- chaque nom est un couple de strings : un identificateur et une catégorie
- chaque association est gérée par un contexte (un objet CORBA, le *naming context*)
- les contextes peuvent être nommés (dans un autre contexte), ce qui donne une structure hiérarchique et des noms composés (*compound name*)
- on obtient le contexte racine grâce à l'ORB (*bootstrapping*)

Naming Service (2)

- type `NameComponent` : composant, i.e., couple identificateur et catégorie
- type `Name` : séquence de `NameComponents`
- interface `NamingContext` :
 - association (`bind` et `rebind`) et résolution (`resolve`) pour les objets
 - manipulation de sous contextes
 - navigation dans la liste des associations
- interface `NamingContextExt` : hérite de `NamingContext` et ajoute des opérations de conversion entre un `Name` et une chaîne de caractères
- représentation d'un `Name` : composants séparés par le caractère `/` (`.` est utilisé pour séparer l'identificateur de la catégorie)
- représentation possible par URL (assez complexe...)

Systemes répartis : CORBA - p.29/45

Cas de Java

Nombreuses implémentations de CORBA pour Java :

- le jdk 1.4 (spécifications 2.3)
- `openorb`
- `JacORB` (spécifications 2.4)
- pleins de choses commerciales...

Exemple traité ici : le jdk 1.4. Outil de base : `idlj` pour la traduction d'une interface `idl` en Java. Quelques éléments du *mapping* :

- module CORBA → *package* Java
- interface `<interface>` → interface Java `<interface>Operations` mais aussi `<interface>`

Systemes répartis : CORBA - p.31/45

Processus de développement

1. Ecriture de l'interface du serveur en IDL
2. Traduction de l'interface en :
 - l'équivalent dans le langage visé (par exemple une interface en Java)
 - des *stubs* pour les clients
 - des *skeletons* pour le serveur
3. Développement indépendant du serveur et des clients

Il faut connaître la traduction pour pouvoir programmer !

Systemes répartis : CORBA - p.30/45

Exemple

```
----- Hello.idl -----
1 module HelloApp {
2     interface Hello
3     {
4         string sayHelloTo(in string firstname,
5                             in string lastname);
6     };
7 };
```

```
----- HelloApp/HelloOperations -----
1 package HelloApp;
2 public interface HelloOperations
3 {
4     String sayHelloTo (String firstname, String lastname);
5 }
```

Systemes répartis : CORBA - p.32/45

Programmation du domestique

- en utilisant le POA
- idlj -fall Hello.idl engendre (entre autre) :
 - HelloOperations.java
 - HelloPOA.java : objet de base dont on hérite pour développer le serveur
- exemple :

```
1 import HelloApp.*;
2 public class HelloImpl extends HelloPOA {
3     public String sayHelloTo (String firstname,
4                               String lastname) {
5         return "Bonjour "+firstname+" "+lastname;
6     }
7 }
```

Démarrage du serveur

- partie un peu lourde (comparée à RMI par exemple)
- synopsis :
 1. ORB.init : initialisation de l'ORB
 2. obtention du RootPOA (avec resolve_initial_references) et activation de celui-ci grâce à son POAManager
 3. mise en place du domestique :
 - (a) création de l'objet
 - (b) enregistrement auprès du POA
 - (c) obtention du NamingContextExt
 - (d) enregistrement de l'objet dans *Naming Service*

Exemple

```
1 import HelloApp.*;
2 import org.omg.CosNaming.*;
3 import org.omg.CORBA.*;
4 import org.omg.PortableServer.*;
5 public class HelloStart {
6     public static void main(String args[]) {
7         try{
8             // initialisation de l'ORB
9             ORB orb = ORB.init(args, null);
10            // obtention du POA racine
11            POA rootpoa =
12                POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
13            // activation de celui-ci
14            rootpoa.the_POAManager().activate();
15            HelloImpl helloImpl = new HelloImpl();
16            // enregistrement auprès du POA
17            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
```

Exemple (2)

```
18 // obtention du Naming Service
19 NamingContextExt ncRef =
20     NamingContextExtHelper.narrow(
21         orb.resolve_initial_references("NameService"));
22 // enregistrement du serveur
23 NameComponent path[] = ncRef.to_name("Hello");
24 ncRef.rebind(path, ref);
25 System.out.println("Démarrage réussi");
26 orb.run();
27 } catch (Exception e) {
28     System.err.println("Erreur : " + e);
29     e.printStackTrace(System.out);
30 }
31 System.out.println("Terminé !");
32 }
33 }
```

Démarrage effectif

- il faut démarrer un serveur de nom !
- avec le jdk 1.4, on utilise orbd
- option importante : `-ORBInitialPort`. Précise le port du serveur de nom
- orbd est un serveur persistant : si on le redémarre, les associations ne sont pas perdues
- démarrage du serveur : `java HelloStart -ORBInitialPort 1050` (même option !)

Le Client

- mêmes grands principes que le démarrage du serveur
- synopsis :
 1. initialisation de l'ORB
 2. obtention du NamingContextExt
 3. récupération de la référence vers l'objet cherché
 4. `cast (narrow)` et appel des méthodes voulues
- démarrage effectif : `java HelloClient -ORBInitialPort 1050`

Exemple

HelloClient

```
1 import HelloApp.*;
2 import org.omg.CosNaming.*;
3 import org.omg.CosNaming.NamingContextPackage.*;
4 import org.omg.CORBA.*;
5 public class HelloClient {
6     public static void main(String args[]) {
7         try{
8             ORB orb = ORB.init(args, null);
9             NamingContextExt ncRef =
10                 NamingContextExtHelper.narrow(
11                     orb.resolve_initial_references("NameService"));
12             Hello helloImpl = HelloHelper.narrow(ncRef.resolve_str("Hello"));
13             System.out.println(helloImpl.sayHelloTo("Joe","Bob"));
14         } catch (Exception e) {
15             System.out.println("Erreur : " + e);
16             e.printStackTrace(System.out);
17         }
18     }
19 }
```

Exemple en C (burk !)

HelloClient.c

```
1 #include "stdio.h"
2 #include "orb/orbit.h"
3 #include "Hello.h"
4
5 int
6 main (int argc, char *argv[])
7 {
8     CORBA_Environment ev;
9     CORBA_ORB orb;
10
11     FILE * ifp;
12     char * ior;
13     char filebuffer[1024];
14     CORBA_char *answer;
15     HelloApp_Hello hello_client;
16
17     CORBA_exception_init(&ev);
18     orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &ev);
```

Exemple en C (burk 2 !)

HelloClient.c

```
19  /*
20  * en utilisant l'IOR (à la place du NS)
21  */
22  ifp = fopen("hello.ior","r");
23  if( ifp == NULL ) {
24      g_error("No hello.ior file!");
25      exit(-1);
26  }
27  fgets(filebuffer,1023,ifp);
28  ior = g_strdup(filebuffer);
29  fclose(ifp);
30  hello_client = CORBA_ORB_string_to_object(orb, ior, &ev);
31  if (!hello_client) {
32      printf("Cannot bind to %s\n", ior);
33      return 1;
34  }
```

Systemes répartis : CORBA - p.41/45

Exemple en C (burk 3 !)

HelloClient.c

```
35  printf("Calling the server\n");
36  answer=HelloApp_Hello_sayHelloTo(hello_client,"Toto","Doe",&ev);
37  if(ev._major != CORBA_NO_EXCEPTION) {
38      printf("we got exception %d from Hello!\n", ev._major);
39      return 1;
40  } else {
41      printf("Answer received\n");
42      printf("%s\n",(char *)answer);
43  }
44  CORBA_Object_release(hello_client, &ev);
45  CORBA_Object_release((CORBA_Object)orb, &ev);
46  return 0;
47  }
```

Systemes répartis : CORBA - p.42/45

Serveur persistant

- une des fonctionnalités proposées par le POA
- attention au sens de persistant :
 - persistance d'un objet CORBA
 - en tant que **service**
 - pas de mécanisme de persistance de l'état du domestique qui implémente le service
 - se contente de démarrer le serveur quand le besoin s'en fait sentir
- modification du démarrage du serveur
- avec le jdk 1.4, on utilise l'outil `servertool` :
 - `servertool -ORBInitialPort 1050`
 - `register -server <classe> -applicationName <nom> -classpath <path>`

Systemes répartis : CORBA - p.43/45

Exemple

HelloPersistant

```
1  import HelloApp.*;
2  import org.omg.CosNaming.*;
3  import org.omg.CORBA.*;
4  import org.omg.PortableServer.*;
5  public class HelloPersistant {
6      public static void main(String args[]) {
7          try{
8              ORB orb = ORB.init(args, null);
9              POA rootPOA =
10                 POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
11                 HelloImpl helloImpl = new HelloImpl();
12                 // création d'un POA dont les domestiques sont persistants
13                 Policy[] persistentPolicy = new Policy[1];
14                 persistentPolicy[0] =
15                     rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
16                 POA persistentPOA = rootPOA.create_POA("MonPOA", null,
17                                                         persistentPolicy );
18                 persistentPOA.the_POAManager().activate();
19             }
20         }
```

Systemes répartis : CORBA - p.44/45

Exemple (2)

HelloPersistant

```
19 // activation
20 persistentPOA.activate_object(helloImpl);
21 NamingContextExt ncRef =
22     NamingContextExtHelper.narrow(
23         orb.resolve_initial_references("NameService"));
24 NameComponent path[] = ncRef.to_name("Hello");
25 ncRef.rebind(path, persistentPOA.servant_to_reference(helloImpl));
26 System.out.println("Démarrage réussi");
27 try {
28     Thread.sleep(5000);
29 } catch (InterruptedException e) {
30 }
31 orb.shutdown(true);
32 } catch (Exception e) {
33     System.err.println("Erreur : " + e);
34     e.printStackTrace(System.out);
35 }
36 System.out.println("Terminé !");
37 }
38 }
```