UNIVERSITÉ TOULOUSE III PAUL SABATIER — UPSSITECH ÉCOLE D'INGÉNIEURS

CORBA

Ph. Truillet

Octobre 2020 – v. 3.4

CORBA®

0. déroulement du TP

- Comprendre l'architecture de CORBA
- Programmer un serveur d'objets et un client avec l'adaptateur BOA

1. introduction

CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture - *http://www.corba.org*) est un standard décrivant une architecture pour la mise en place d'objets distribués. Elle a été spécifiée par l'OMG (Object Management Group) en 1992 pour la première fois. La dernière spécification de CORBA (version **3.4beta** CORBA® - *http://www.omg.org/spec/CORBA*) date de mai 2020. CORBA est basé sur un bus, l'ORB (**O**bject **R**equest **B**roker), qui assure les collaborations entre applications (cf. figure 1). La version 3.1.1 (Août 2011) a été publiée par l'ISO sous les standards ISO/IEC 19500-1, 19500-2 et 19500-3.

Les communications sont basées sur le mécanisme **d'invocation de procédures distantes** (comme pour RMI) et requièrent la création d'amorces qui se branchent au bus et permettent l'émission et la réception de messages entre les clients et les serveurs.

L'ORB prend généralement la forme d'une *bibliothèque de fonctions* assurant la communication entre les clients et les serveurs. **Dans le cadre des Travaux Pratiques, nous utiliserons l'ORB fourni par Java depuis sa version 1.3** (OMG CORBA API).

CORBA est une spécification et non un langage. C'est pourquoi plusieurs éditeurs sont présents sur le marché : c'est le cas par exemple d'Inprise avec Visibroker, d'Orbix Web de Iona ou encore JavaOrb (*http://www.jacorb.org*) qui est un ORB libre.

Les communications sont basées sur le mécanisme d'invocation de procédures distantes (comme pour RMI) et requièrent la création d'amorces qui se branchent au bus et permettent l'émission et la réception de messages entre les clients et les serveurs (cf. figure1).

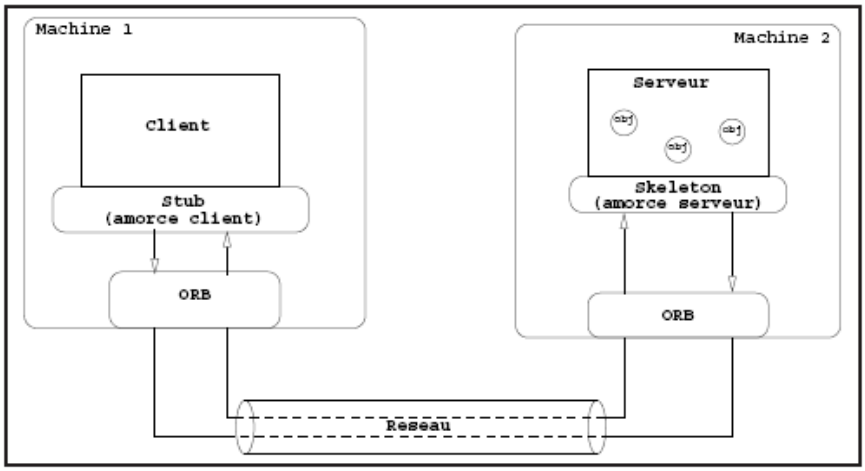

Figure 1 : architecture CORBA

2. Interface Definition Language (IDL)

Le rôle d'un serveur est de mettre un ensemble d'objets à la disposition des clients. Pour pouvoir accéder à ces objets, les clients doivent pouvoir connaître l'ensemble des méthodes qu'ils peuvent invoquer sur ces objets.
Ceci est fait par l'intermédiaire de « **contrats** » définis à l'aide de l'**I**nterface **D**efinition **L**anguage (IDL).

Le langage IDL permet d'exprimer la coopération entre les fournisseurs et les utilisateurs de services en séparant l'interface de l'implémentation.

2.1 interface IDL

Le contrat entre les fournisseurs et les clients s'exprime sous la forme d'un ensemble d'interfaces spécifiées à l'aide du langage IDL. Une interface décrit l'ensemble des opérations fournies par un type d'objet CORBA.
La notion d'interface est similaire à celle de classe utilisée en programmation orientée objet. Une interface met en oeuvre des méthodes et des attributs dont il est nécessaire de définir le type. CORBA étant destiné à créer des applications inter-opérables, **les types de base utilisés sont spécifiques au langage IDL** (*il ne s'agit ni de types Java, ni de types* C++ - cf. Figure 2). Ces types sont ensuite projetés sur les langages dans lesquels sont réalisées les implémentations.

IDL	Java
`module`	`package`
`interface`	`interface`
`operation`	`method`
`exception`	`exception`

Type IDL	Type Java
`boolean`	`boolean`
`char/ wchar`	`char`
`octet`	`byte`
`short / unsigned short`	`short`
`long / unsigned long`	`int`
`long long / unsigned long long`	`long`
`float`	`float`
`double`	`double`
`string / wstring`	`String`

Figure 2 : Correspondances entre le langage IDL et java

2.2 passage des arguments

Pour la description de fonctions prenant un ou plusieurs arguments, IDL propose 3 types de passages de paramètres :

- **`in`** : indique que le paramètre est passé au serveur,
- **`out`** : indique que le paramètre est retourné au client,
- **`inout`** : indique que le paramètre est passé au serveur où il peut être modifié et ensuite retourné au client.

Nota : En IDL, si on définit un passage de paramètres en sortie pour le type `xxx` (par exemple : `Short`), le type `xxxHolder` (`ShortHolder`) sera utilisé en Java (défini dans le package `org.omg.corba`).

Jusqu'à la norme CORBA 2.1, il n'existait qu'un seul adaptateur défini par l'OMG : le **BOA** (**B**asic **O**bject **A**dapter) qui fournit des services permettant la création d'objets CORBA. Néanmoins, BOA peut être ambigu dans certains cas et des fonctions essentielles ne sont pas proposées. Pour des raisons de compatibilité, cet adaptateur reste disponible.

Le nouveau standard **POA** (**P**ortable **O**bject **A**dapter) fournit quant à lui de nouvelles fonctionnalités et permet aux développeurs de concevoir des implémentations d'objets portables sur différentes implémentations d'ORB.

3. génération des amorces avec BOA

Récupérez tout d'abord le projet **horloge** à l'adresse suivante :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/Horloge.idl`

Le **contrat IDL** va être utilisé pour **générer les amorces**. Vérifiez tout d'abord que le chemin vers la JDK ou la JRE est bien défini dans votre environnement.
Avec Java, la génération des amorces se fait par l'intermédiaire de la commande :

```
idlj –fall –oldImplBase Horloge.idl
```

L'option `–fall` indique que l'on génère les amorces pour le serveur et le client (`all`). L'option `–oldImplBase` indique qu'on va utiliser l'adaptateur à objets **BOA** (**B**asic **O**bject **A**daptor) rétro-compatible avec les anciennes versions de Java.

Au cours de la pré-compilation, un ensemble de fichiers est généré. La souche correspond au fichier `_HorlogeStub.java` (partie client) et le squelette correspond au fichier `_HorlogeImplBase.java` (partie serveur).
Dans la souche et le squelette générés, vous trouverez les étapes (cachées pour le programmeur) d'encodage (*marshalling*), invocation des méthodes et décodage (*unmarshalling*).

4. implémentation des classes

Une fois le contrat IDL et les amorces définis, il est nécessaire d'implémenter les fonctions proposées par le serveur. L'implémentation consiste généralement en la définition d'une classe `XXXServant` héritant de la classe `_XXXImplBase` (i.e. le squelette).

5. mise en place du serveur d'objet avec BOA

Le but du serveur est de mettre des objets (correspondant à l'implémentation) à la disposition de ses clients, de recevoir les requêtes.
Son fonctionnement se déroule en 6 étapes :

1. initialisation de l'ORB
2. création de l'objet
3. activation de l'objet
4. enregistrement de l'objet auprès du BOA
5. mise en attente des requêtes des clients

Téléchargez le fichier `Horloge_Server` à l'adresse suivante :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/Horloge_Server.zip`

Ouvrez le fichier `HorlogeServant.java` (préalablement défini) pour vérifier que l'implémentation est valide
Avec Java générez les amorces côté serveur : `idlj –fserver –oldImplBase Horloge.idl`
Créer un projet sous Eclipse en y ajoutant les fichiers générés et les fichiers du projet. Lancez le projet `HorlogeServer`

6. mise en place d'un client avec BOA

Le but du client est d'accéder à l'objet distant et d'invoquer les méthodes proposées par cet objet. Son fonctionnement se déroule en 5 étapes :

1. initialisation de l'ORB
2. obtention d'une référence à l'objet distant
3. extraction de l'objet (lien au stub) correspondant
4. invocation de la méthode distante

Téléchargez le fichier Horloge_Client à l'adresse suivante :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/HorlogeClient.zip`

Avec Java générez les amorces côté client : `idlj –fclient –oldImplBase Horloge.idl`

Créer un projet sous Eclipse en y ajoutant les fichiers générés et les fichiers du projet. Lancez le projet `HorlogeClient`

Nota : Pensez à changer l'adresse du bus généré par la partie serveur.

7. le service de nommage (Naming Service)

CORBA propose la possibilité d'accéder à un objet distant à partir d'un nom plus facile à mémoriser que la référence retournée par la méthode `object_to_string`. La mise en correspondance d'un nom et d'un objet se fait grâce au service de nommage.
Avec Java, le service de nommage est lancé à l'aide de la commande :

`tnameserv –ORBInitialPort port où port` est relatif au port utilisé (port 900 par défaut)

Au niveau du serveur, l'utilisation du service de nommage se déroule en 3 étapes :

1. Accès à un objet `NamingContext` correspondant à un « annuaire » des objets mis à la disposition des clients,
2. Définition du nom associé à l'objet (`NameComponent`),
3. Association de l'objet à son nom à l'aide de la méthode `rebind`.

Au niveau du client, l'utilisation du service de nommage se déroule en 3 étapes :

1. « Récupération » d'un objet `NamingContext` correspondant au service de nommage,
2. Obtention d'une référence à l'objet distant à partir de sa dénomination (`resolve`),
3. Conversion de la référence (`org.omg.CORBA.Object`) en un objet utilisable (`Horloge`) à l'aide de `XXXHelper.narrow()`.

Exemple :

- Téléchargez `Horloge2.zip` à l'adresse suivante :
  `https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/Horloge2.zip`

Remplacez le fichier `HorlogeServer.java` et `HorlogeClient.java` dans les projets respectifs. Lancer `tnameserv`, le serveur et le client.

Concernant l'utilisation du POA, vous pouvez consulter le tutorial disponible à l'adresse suivante :
`http://download.oracle.com/javase/1.4.2/docs/guide/idl/jidlTieServer.html`

8. exercices

1. Développez l'objet d'implantation, le serveur et le client de cette description IDL

```
interface Addition {
   long addition(in long a , in long b);
   long soustraction(in long a, in long b);
   void memorise(in long a);
   long get_memory();
};
```

***Nota** : le type long IDL se projette en « int » en Java*

2. Développez un **annuaire** en CORBA
   a. Définissez son interface IDL
   b. Codez l'objet d'implantation et le serveur
   c. Codez un exemple de client qui utilise ce serveur

***Nota** : java.lang.String a pour type IDL string ou wstring*

9. CORBA et RMI

RMI-IIOP permet d'utiliser RMI avec le protocole internet InterORB (IIOP) comme sous-couche de transport des données. Cela permet de combiner à la fois la simplicité de RMI avec la puissance de CORBA.

Les tâches de base reprennent celles utilisées par RMI ou CORBA, c'est-à-dire :

1. Définir les méthodes de la classe « remote » (interface)
2. Ecrire l'implémentation de la classe
3. Ecrire, lancer le serveur et s'inscrire à l'annuaire de noms
4. Ecrire et lancer le client.

Récupérez tout d'abord le projet CORBA-RMI (exemple « *Hello Wold* ») à l'adresse suivante :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/CORBA-RMI.zip`

Nota : la JDK doit être installée dans votre environnement et son chemin doit être positionné dans les variables d'environnement.

Dézippez le fichier et allez dans le répertoire `server`. Lancer successivement `compile_server.bat` puis `go_server.bat` (**Attention** : une première fenêtre s'affiche permettant de lancer le serveur de noms, revenir dans la deuxième fenêtre pour lancer ensuite le serveur)
Aller dans le répertoire `client`. Lancer successivement `compile_client.bat` puis `go_client.bat`. La communication s'est effectuée !
Reprenez maintenant votre exercice sur l'annuaire (TP « RMI ») pour faire communiquer le client et le serveur à travers le protocole IIOP.

10. aller plus loin :

- *Tutoriaux CORBA* :
  `http://download.oracle.com/javase/1.4.2/docs/guide/idl/index.html`