

TP 3 – CORBA¹

1 Rappels

CORBA (Common Object Request Broker Architecture) est une norme qui permet de définir une interface de services disponibles à distance et indépendante d'un langage particulier. Pour cela, le langage descriptif IDL (Interface Definition Language) sert de support et permet une projection (traduction) dans les principaux langages objets. Cette norme permet l'interopérabilité entre programmes écrits dans différents langages.

1.1 Fonctionnement

Une interface définie en IDL est projetée dans un langage, c'est-à-dire que son équivalent est défini dans ce dernier. Le déroulement du développement et de l'exécution est alors similaire à celui de RMI, notamment le principe de transparence de l'appel pour le client et l'utilisation d'un stub et d'un squelette qui s'occupent de la sérialisation et de la transmission/réception des communications.

En CORBA, on appelle *servant* l'objet implantant une interface IDL. Un servant est *logé* sur une application *serveur*.

1.2 Développement & compilation

Le développement et la compilation d'un service CORBA se fait en plusieurs étapes. Tout le code qui suit forme un ensemble minimal. Toute classe importée, implémentée ou étendue l'est obligatoirement, toute exception gérée doit l'être.

1. **Développement** de l'interface de l'objet en langage IDL.

```
// fichier I<classname>.idl
interface I<classname>
{
    <return_type_idl> myfunction1 ([<in/out/inout> <type_idl> arg]*) ;
    ...
} ;
```

2. **Projection** de l'interface IDL en un langage donné (Java dans ce cours, avec l'outil *idlj*) :

```
idlj -fall -oldImplBase I<classname>.idl
```

Ceci crée les fichiers *_I<classname>ImplBase.java* (skeleton), *_I<classname>Stub.java* (proxy), *I<classname>Helper.java* (assistant de transtypage entre un objet CORBA et un objet Java), *I<classname>Operations.java*, *I<classname>.java* (interfaces projetées en Java) et *I<classname>Holder.java* (pour le passage de paramètre de type *out* ou *inout*, voir plus bas).

3. **Programmation** d'une implémentation des méthodes correspondant à cette interface. Pour cela, il faut regarder le fichier *I<classname>Operations.java*, qui spécifie l'interface à implémenter.

```
// fichier I<classname>Impl.java
import org.omg.CORBA.* ;

public class I<classname>Impl
    extends _I<classname>ImplBase // étends le skeleton
```

1. Sujet créé à partir de documents de Guillaume LATU et de Kenneth VANHOEY

```

{
    // membres
    ...
    public I<classname>Impl() // constructeur
    {
        super() ;
    } ;

    public <return type java> myfunction1 ([<type_java> arg]*)
    {
        // corps
    }
    ...
}

```

4. **Développement** d'un programme serveur qui mettra à disposition l'objet :

```

import org.omg.CORBA.* ;

public class Serveur
{
    public static void main(String[] args)
    {
        try
        {
            //initialiser le bus ORB
            ORB orb = ORB.init(args, null ) ;

            // Déclarer l'objet
            I<classname>Impl myobj = new I<classname>Impl() ;
            // Dédure l'identifiant unique IOR
            String ior = orb.object_to_string( myobj ) ;
            // Afficher l'IOR
            System.out.println( ior ) ;

            orb.run() ;
        }
        catch( org.omg.CORBA.SystemException ex ) { ex.printStackTrace() ; }
    }
}

```

Si l'on n'utilise pas d'annuaire, comme dans cet exemple, la chaîne identifiant l'objet (IOR) devra être transmise aux clients.

5. **Développement** d'un client qui utilise l'objet en question via des appels à distance.

```

import java.io.* ;
import org.omg.CORBA.* ;

public class Client
{
    public static void main(String args[])
    {
        if( args.length < 2 )
        {
            System.out.println( "Usage: java Client <ior>" ) ;
            System.exit( 1 ) ;
        }
    }
}

```

```

try
{
    // initialiser l'ORB.
    ORB orb = ORB.init( args, null ) ;

    // Lire l'IOR
    String ior = args[0] ;

    // Traduire IOR en objet CORBA
    org.omg.CORBA.Object obj = orb.string_to_object(args[0]) ;
    // Traduire l'objet CORBA en Objet I<classname> grace au Helper
    I<classname> service = I<classname>Helper.narrow(obj) ;

    // Utiliser l'objet service récupéré
    ...
}
catch( org.omg.CORBA.SystemException ex )
{
    System.err.println( "Erreur !" ) ;
    ex.printStackTrace() ;
}
}
}

```

6. **Compilation** des programmes serveur et client dès que leur code est terminé. Le client doit avoir accès à l'interface (projetée en java) lors de sa compilation. Le serveur doit avoir accès à l'implémentation de l'interface lors de la compilation.
7. Pour l'**exécution**, il faut d'abord lancer le serveur, ce qui aura pour effet de générer un IOR (identifiant unique) :

```
java Serveur
```

Le serveur doit avoir accès à l'interface et au squelette lors de l'exécution.

Le client peut alors être exécuté, à condition d'avoir le stub à disposition. Il devra spécifier l'IOR du service mis en place par le serveur.

```
java Client <IOR>
```

1.3 Variante : ajout du POA

L'option `-oldImplBase` d'`idlj` permet de générer un certain type de squelette qui ne fonctionne qu'avec les ORB de Sun, mais est facile d'utilisation. Afin d'assurer l'interopérabilité avec tous les langages, le POA (Portable Object Adapter) se place entre le squelette et l'ORB, donc sur le serveur. L'implémentation de l'interface étends alors `<classname>POA` au lieu de `_I<classname>ImplBase` et le serveur doit initialiser et activer le POA afin de générer l'IOR :

```

// initialiser le POA
POA poa = POAHelper.narrow( orb.resolve_initial_references( "RootPOA" ) ) ;

// activer le POA
poa.the_POAManager().activate() ;

// créer la référence vers le servant
org.omg.CORBA.Object poaobj = poa.servant_to_reference( myobj ) ;

// Générer l'IOR à partir de la référence
String ior = orb.object_to_string( poaobj ) ;

```

Remarque : le serveur doit importer `org.omg.PortableServer.*` et traiter l'exception `org.omg.CORBA.UserException`.

2 Exercice

Définissez pas à pas un code mettant à disposition une classe *Couleur* composé d'un triplet RGB, puis l'utilisant via un mécanisme CORBA client/serveur. Cette classe aura une méthode *setColorRGB* permettant de définir ses trois composantes, et une méthode *getLuminance* qui renvoie la luminance de la couleur en question. Pour information, la luminance d'une couleur définie en RGB vaut $0.299 * R + 0.587 * G + 0.114 * B$. Vous utiliserez l'implémentation la plus simple, sans POA.

1. Écrivez l'interface *ICouleur.idl*, sa projection en Java et son implémentation ;
2. Écrivez le code d'une application serveur et d'une application cliente ;
3. Écrivez les instructions à exécuter dans le bon ordre afin de faire fonctionner l'interaction Corba.

3 Définition de l'interface IDL

Le langage IDL a ses propres structures de données et sa syntaxe.

Types de base : *void*, *short* (16 bits), *unsigned short*, *long* (32 bits), *unsigned long*, *long long* (64 bits), *unsigned long long*, *float* (32 bits, IEEE), *double* (64 bits, IEEE), *long double* (128 bits), *boolean*, *octet* (8 bits), *char* (8 bits, ISO Latin-1), *string*.

Déclarations En IDL, une valeur constante se définit en préfixant la déclaration de variable par le mot-clé *const*, un alias, une énumération et une structure s'écrivent comme en C grâce aux mots-clé respectifs *typedef*, *enum* et *struct*.

Tableaux et séquences se déclarent grâce au mot-clé *sequence*. Une séquence peut être bornée ou non :

séquence bornée *sequence* <type, borne>

séquence non bornée *sequence* <type>

Exceptions :

```
exception nom_exception
{
    <type> nom_membre1 ;
    <type> nom_membre2 ;
    ...
    <type> nom_membreN ;
} ;
```

Passage de paramètres de type *in*, *out* ou *inout*. Les cas *out* et *inout* doivent pouvoir fonctionner comme un passage par référence en C++.

3.1 Questions

1. Comment fonctionne le passage de paramètres en Java ?
2. Qu'est-ce qu'une classe *Holder*, générée par l'outil de projection *idlj* ? Comment opèrent-elles ?
3. Ajoutez à votre interface *ICouleur.idl* une fonction permettant de récupérer ses valeurs *CMY*.
Donnez l'interface Java générée par *idlj*. Ré-écrivez le client afin de pouvoir récupérer ces valeurs.
4. Par quel mot-clé peut-on rendre une fonction IDL non-bloquante ?
Quelles sont les conditions pour qu'une fonction puisse être définie comme étant non bloquante ?
Quelles méthodes parmi celles de votre interface IDL pourraient être définies comme non bloquantes ?

4 Carré

1. Rendez-vous dans le répertoire *1-Carre/*. Celui-ci contient l'interface IDL d'un service de calcul du carré d'un nombre.
2. Réaliser la projection de l'interface IDL :

```
idlj -fall -oldImplBase Icarre.idl
```

Notez tous les fichiers **.java* qui sont générés automatiquement ; puis lancez la compilation complète par **./compile**.

Important : ouvrez les fichiers *Icarre.java* et *IcarreOperations.java*. C'est ce dernier fichier **Operations.java* qui vous donne l'interface Java que vous devrez utiliser pour votre implémentation. À chaque fois que vous écrirez ou utiliserez une interface IDL, il faudra regarder ce fichier.

3. Lancez l'application en commençant par le serveur :

```
java Serveur
```

qui normalement affiche son *IOR* (Interoperable Object Reference).

Attention : la chaîne « *IOR* : » fait partie de l'IOR du serveur.

Puis lancer le client :

```
java Client <IOR_du_serveur> <nombre>
```

Question (à répondre dans *reponse.txt*) : Comment le Client arrive-t-il à savoir sur quelle machine se trouve le Serveur ?

4. N'oubliez pas de tuer le serveur !

5 Paramètres et Holders

1. Recopier *1-Carre/* dans un nouveau répertoire que vous appellerez *2-Carre_Holder* et référez-vous au cours/TD pour cet exercice (paramètres de type *in*, *out* ou *inout* et les classes *Holder*).
2. Questions (à répondre dans *reponse.txt*) :
 - Comment génère-t-on une classe *Holder* ?
 - Pour quels types de paramètres a-t-on besoin d'une classe *Holder* ?
3. Modifier la fonction « *long carre(in long source)* » afin de faire passer le résultat de la fonction en second paramètre. La nouvelle signature de la fonction devient donc « *void carre(in long source, out long resultat)* ».

Pour cela vous interviendrez sur les codes de : *Icarre.idl*, puis *IcarreImpl.java*, puis *Client.java*.

6 POA

Précédemment, nous avons utilisé le paramètre **-oldImplBase** à la compilation avec **idlj** et avons ainsi généré une interface CORBA compatible avec des ORB d'avant la JDK 1.4. Depuis cette version, cela se passe différemment, grâce aux POA (Portable Object Adaptor).

1. Recopiez *1-Carre/* dans un nouveau répertoire que vous appellerez *3-Carre_POA* et référez-vous au cours/TD pour cet exercice.
2. On va utiliser le POA dans cette nouvelle version. Par rapport à l'exercice 1, un intermédiaire (le POA) se place entre le squelette de l'objet distant et l'ORB au niveau du serveur. Ce POA normalisé permet de faire interopérer le serveur avec différents ORBs (pas uniquement celui de java SUN).

Vous modifierez le fichier *compile* en enlevant l'option **-oldImplbase** lorsque vous appelez **idlj**.

Vous ferez attention que la projection IDL vers java change alors (le squelette généré n'est plus *_IcarreImplBase* mais *IcarrePOA*). Les fichiers à modifier sont alors : *IcarreImpl.java* et *Serveur.java* (veillez à importer *PortableServer*).

7 MatricesNN, le retour

Rendez-vous dans le répertoire *4-MatricesNN/*. Le fichier *OpMatrice.idl* définit l'interface IDL d'un service proposant la multiplication de deux matrices. Projetez l'interface IDL en Java. Regarder en quel type Java est projeté le type IDL « *long* ».

Proposer une implémentation CORBA de ce service équivalente à ce que l'on a fait précédemment avec RMI dans le TP2 (vous avez à créer les fichiers *OpMatriceImpl.java*, *Client.java* et *Serveur.java*). Cette fois-ci on pourra multiplier des matrices rectangulaires quelconques : « A » de dimension $n \times k$ et « B » de dimension $k \times m$. Vous utiliserez obligatoirement le POA.

Votre Client demandera au service distant de multiplier les deux matrices A et B telles que :

```
int[] [] A = { {1, 0, 0, 0}, {0, 3, 0, 0}, {0, 1, 3, 0}, {0, 4, 0, 7} } ;  
int[] [] B = { {1, 4}, {1, 8}, {2, 3}, {1, 0} } ;
```

$$\text{Vous devez bien sur trouver : } res = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 4 & 2 & 7 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 8 \\ 2 & 3 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 24 \\ 7 & 17 \\ 11 & 32 \end{bmatrix}$$

Remarque pour éviter de faire un copier/coller de l'IOR à partir du shell, on peut écrire l'IOR dans un fichier avec le Serveur puis le lire depuis le Client. À cet effet, une constante « *iorfile* » contenant le nom du fichier où l'on va stocker l'IOR est définie dans le fichier IDL.

Dans le serveur pour écrire l'IOR dans un fichier, utilisez le code :

```
import java.io.* ;  
...  
FileOutputStream file = new FileOutputStream(iorfile.value) ;  
PrintWriter out = new PrintWriter(file) ;  
out.println(ior) ; out.flush() ;  
file.close() ;
```

Dans le client pour lire l'IOR dans un fichier, utilisez le code :

```
import java.io.* ;  
...  
FileReader file = new FileReader(iorfile.value) ;  
BufferedReader in = new BufferedReader(file) ;  
ior = in.readLine() ;  
file.close() ;
```

8 Annuaire

Rendez-vous dans le répertoire *5-Annuaire/*. Le fichier *Annuaire.idl* définit l'interface IDL d'un service d'annuaire. Proposer une implémentation CORBA de ce service équivalente à celle vue avec RMI. Pour cela, vous utiliserez le POA. Pour démarrer, vous pouvez utiliser la correction de l'exercice RMI dans le répertoire *TelephoneExerciceRMI/*.