Polytech Marseille

TD
CORBA

Informatique - Formation par alternance HUGo

Rappel : tous les exemples du cours sont téléchargeables sur la page web du cours :
`http://nicolas.durand.perso.luminy.univ-amu.fr/pub/`

En annexe à la fin du sujet de ce TD), vous trouverez l'exemple complet "hello world".

Exercice 1. Gestion d'un parking

On considère un parking dont on souhaite automatiser la gestion en l'administrant à distance avec un middleware CORBA. Le parking peut accueillir au maximum 50 véhicules. Une barrière gère les entrées et les sorties de véhicules : elle délivre un ticket à l'entrée d'un véhicule et elle contrôle le ticket fourni par l'automobiliste à la sortie. Un automate de paiement permet sur présentation du ticket, de régler le montant correspondant au temps de stationnement. L'automobiliste règle le montant du stationnement à l'automate avant de franchir la barrière pour sortir. La barrière, l'automate et le parking sont représentés chacun par un objet CORBA.

L'interface `BarriereItf` de la barrière fournit deux méthodes :
- `entrer` : correspond à l'entrée d'un véhicule dans le parking. Retourne `vrai` si le véhicule peut rentrer (si place libre), `faux` sinon (et dans ce cas le véhicule ne peut pas entrer). Cette méthode fournit en paramètre de sortie une structure `Ticket` comprenant un numéro de ticket de type entier et une heure d'entrée sous la forme d'une chaîne de caractères.
- `sortir` : correspond à la sortie d'un véhicule. Retourne `vrai` si l'automobiliste a réglé le montant du stationnement, `faux` sinon (et dans ce cas le véhicule ne peut pas sortir). Cette méthode prend en paramètre d'entrée une structure `Ticket`.

L'interface `AutomateItf` de l'automate fournit deux méthodes :
- `payer` : correspond au paiement en fonction de la durée de stationnement. Retourne `vrai` si le montant fourni par l'automobiliste est suffisant, `faux` sinon. Cette méthode prend en paramètre d'entrée une structure `Ticket` et un montant de type réel double. Cette méthode fournit également en sortie une valeur de type réel double représentant la monnaie à rendre à l'utilisateur.
- `cestregle` : à partir d'une structure `Ticket` fournie en entrée, retourne `vrai` si le montant du ticket a été réglé, `faux` sinon.

L'interface `ParkingItf` du parking fournit trois méthodes :
- `nbPlacesLibres` : retourne un entier indiquant le nombre de places libres dans le parking.
- `entreeVehicule` : signale l'entrée d'un véhicule dans le parking. Ne prend aucun paramètre, ne retourne rien.
- `sortieVehicule` : signale la sortie d'un véhicule du parking. Ne prend aucun paramètre, ne retourne rien.

1. Définissez la structure `Ticket` et les interfaces IDL correspondant à ces trois objets CORBA dans un module `ParkingPkg`.
2. Quel mot-clé faut-il indiquer dans la déclaration des méthodes `entreeVehicule` et `sortieVehicule` pour qu'elles ne soient pas bloquantes ?

3. Par quelle classe le ticket en paramètre de la méthode entrer sera t'il remplacé en Java ? Quelle est son utilité ?

Exercice 2. Espace de données partagées

Le but de cet exercice est de définir un serveur CORBA de données partagées (`ServDP`) dans lequel des clients viennent déposer et retirer des données. Un tel service peut servir par exemple dans un environnement distribué, à ce que des stations de travail déposent des travaux à effectuer, et qu'une ou plusieurs autres stations, en fonction de leurs disponibilités, récupèrent ces travaux et les exécutent éventuellement de façon concurrente.

`ServDP` stocke des couples clé-valeur, où clé est une chaîne de caractères et valeur une séquence d'octets. Le stockage de chaque valeur est ainsi identifié par une clé unique.

Son interface `ServDPItf` fournit les méthodes suivantes :
- `deposer` : prend 2 paramètres qui permettent au client de spécifier la clé et la valeur à stocker,
- `retirer` : le premier paramètre est la clé à retirer de l'espace de données. La valeur associée à la clé est fournie au client par un deuxième paramètre,
- `lire` : idem retirer sauf que le couple clé-valeur reste stocké dans l'espace de données. L'interface comprend aussi un attribut de type entier contenant le nombre de couples clé-valeur stockés.

1. Définissez le contrat en IDL CORBA correspondant à cette spécification. Remarque : utilisez le type `octet` pour représenter un octet.
2. Définissez des exceptions pouvant survenir pour chacune des méthodes de l'interface. Modifiez le contrat en conséquence.
3. En ne considérant pas les exceptions, quelle(s) méthode(s) de l'interface définie en question 1 pourrait(aient) être déclarée(s) `oneway` ? Rappelez les caractéristiques des appels de méthodes `oneway`.
4. Comment sera représenté la séquence d'octets en Java ?
   Quel sera le type Java du paramètre correspondant à la valeur dans les méthodes retirer et lire ? Quelle est son utilité ?
5. Jusqu'à présent, les clients qui ont besoin de lire ou de retirer des données n'ont d'autres alternatives, lorsque la donnée est absente, que d'interroger périodiquement le serveur. Cette interrogation réalisée à distance, est potentiellement coûteuse. On souhaite mettre en place un mode plus interactif dans lequel, en cas d'absence de données, les clients sont prévenus dès que cette donnée devient disponible. Donnez un mécanisme répondant à ce problème. Expliquez la solution à mettre en place.

L'objectif de cet exercice est de développer une application de forum interrogeable à distance, permettant de poster ou de récupérer des messages. Pour cela, vous allez développer le contrat IDL (interface), la classe d'un futur objet serveur (servant), le serveur et le client.

Le contrat IDL

Le contrat IDL sera défini dans le fichier "Forum.idl".

Les messages échangés ne sont pas des objets CORBA. Ils sont représentés par la structure suivante :

```
struct Message {
 string title;
 string author;
 string date;
 string body;
};
```

Le forum est un objet CORBA dédié à la gestion des messages d'un thème particulier (attribut theme) et sous la responsabilité d'un administrateur (attribut moderator). Il est représenté par l'interface suivante :

```
interface Forum {
 readonly attribute string theme;
 readonly attribute string moderator;
 boolean postMessage(in Message m);
 Message getMessage(in string title);
 boolean removeMessage(in string title);
};
```

Placez vos définitions IDL (structure Message et interface Forum) dans le module `tp`.

Générez les fichiers d'amorce pour le serveur et le client avec `idlj`.
Exemple : `idlj -fall Forum.idl`

La classe d'un servant

Vérifiez que le fichier `ForumPOA.java` a bien été généré et implantez la classe d'un futur objet servant (`ForumImpl`) en héritant de cette classe.

`ForumImpl` contient, bien entendu, les attributs `theme` et `moderator`, et les méthodes `postMessage`, `getMessage` et `removeMessage`.

Pour stocker les messages (i.e. les objets issus de la classe `Message` générée à l'exercice précédent), on utilisera un attribut `messages` qui sera une `ConcurrentHashMap` associant chaque message à son titre. Les clés sont donc les titres et les valeurs sont les messages.

Le serveur

Ecrivez une classe `ForumServeur` qui effectue des étapes suivantes (similaires à la classe `HelloServer` de l'exemple "Hello World").

Lancez le service de nommage avec le port de votre choix.
`orbd -ORBInitialPort` *`port`*

Compilez et lancez le serveur.
`java ForumServer -ORBInitialPort` *`port`* `-ORBInitialHost` *`localhost`*

Attention : vous devrez peut-être écrire le nom de votre package devant "ForumServer".

Le client

Ecrivez une classe `ForumClient` (inspirez vous de la classe `HelloClient` de l'exemple "Hello World"). Dans cette classe, invoquez de méthodes distantes afin de tester l'application.

Compilez et testez votre client.

`java ForumClient -ORBInitialPort` *`port`* `-ORBInitialHost` *`localhost`*

Attention : vous devrez peut-être écrire le nom de votre package devant "ForumClient".

Les exceptions

On souhaite maintenant améliorer notre forum en lui faisant lever des exceptions en cas d'erreur. Pour cela, il faut d'abord définir en IDL les exceptions avec la syntaxe suivante :

```
exception ExceptionName { string message; };
```

Ensuite, il faut indiquer dans l'interface du forum les méthodes qui lèvent une exception. Rappelons qu'une méthode suivie du mot-clé `raises` et du nom d'une exception entre parenthèses, indique que la méthode est susceptible de lever ce type d'exception au niveau du client au retour de l'appel.
Mettre en œuvre l'exception `Reject` qui sera levée par la méthode `postMessage()` lorsque un message de même titre existe déjà, et par les méthodes `getMessage()` et `removeMessage()` s'il n'existe pas de message avec ce nom.

Les séquences

Modifiez les fichiers de votre application de forum afin d'intégrer un type séquence de messages (`MessageSet`), puis ajouter dans l'interface `Forum` l'opération `getMessages()` qui renvoie la liste des messages du forum.
Rappel : en IDL, une séquence correspond à un tableau de taille variable. Un type séquence de nom `TSet` contenant des objets de type `T` est déclaré par :

```
typedef sequence <T> TSet;
```

Les paramètres out

Dans cet exercice nous souhaitons ajouter à l'interface `Forum` la méthode `getInfo()` qui retourne les différentes informations sur le forum :

```
interface Forum {
 void getInfo (out string theme, out string moderator, out long size);
};
```

Les classes `IntHolder` et `StringHolder`, entre autres, permettent de transmettre des paramètres par référence. Cela permet de récupérer la valeur si elle a été modifiée par le serveur.

ANNEXE : Exemple ”Hello World” en CORBA avec Java

Objectif : lire, comprendre, réaliser et tester l'exemple "Hello World".

Attention : ici, il n'y a pas de package spécifié contrairement à l'exemple téléchargeable sur la page web du cours. À vous d'adapter si besoin.

1. Module et interface : fichier "Hello.idl"

```
module HelloApp
{
  interface Hello
  {
    string sayHello();
  };
};
```

2. Projection vers JAVA

```
idlj -fall Hello.idl
```

idlj est le compilateur IDL vers java (inclus dans le JDK).

Les fichier suivants sont alors générées : _HelloStub.java, Hello.java, HelloOperations.java, HelloPOA.java, HelloPOATie.java, HelloHelper.java, HelloHolder.java

La classe ”...Operations” : reprend l'interface IDL en une interface Java.

Les classes Helper : tous les types utilisateurs définis dans l'IDL donneront naissance à des classes suffixées par le mot clé Helper. Elles permettent entre autres de gérer les types any et typecode de l'IDL. Elles contiennent une méthode narrow qui permet de faire le lien entre la référence d'un objet à la classe à laquelle il appartient.

Les classes Holder : partie de code générée automatiquement par un compilateur IDL vers un langage de programmation cible. Tous les types (prédéfinis ou utilisateurs) possèdent une classe Holder qui est utilisée lors des passages en out ou inout lors de l'invocation de méthode.

3. La classe des futurs servants

La classe ”HelloImpl” qui implémente l'interface ”Hello”, doit dériver de "HelloPOA".

```
import org.omg.CORBA.ORB;
import HelloApp.HelloPOA;

class HelloImpl extends HelloPOA {
        public String sayHello() {
                return "\nHello World!\n";
        }
}
```

Remarquez que rien n'empêche de définir d'autres attributs et d'autres méthodes dans l'implémentation (qui n'étaient pas dans l'interface). Ces méthodes ne pourront pas être appelées à distance, c'est tout.

4. Le programme serveur

Les tâches du programme serveur sont :

- Création et initialisation de l'ORB ;
- Récupération de la référence de l'adaptateur d'objets racine et activation du manager ;
- Création de l'objet servant ;
- Récupération de la référence du servant ;
- Récupération de la référence de l'objet du service de nommage en utilisant sa référence initiale "NameService" ; Conversion vers un type utilisable au moyen de la méthode narrow() de la classe NamingContextExtHelper ;
- Association d'un nom à la référence du servant, dans le service de nommage ;
- Mise en attente des requêtes.

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

public class HelloServer {
 public static void main(String args[]) {
  try{
 // create and initialize the ORB
 ORB orb = ORB.init(args, null);

 // get reference to rootpoa & activate the POAManager
 POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
 rootpoa.the_POAManager().activate();

 // create servant and register it with the ORB
 HelloImpl helloImpl = new HelloImpl();

 // get object reference from the servant
 org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
 //Hello href = HelloHelper.narrow(ref);

 // get the root naming context
 // NameService invokes the name service
 org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
 // Use NamingContextExt which is part of the Interoperable
 // Naming Service (INS) specification.
 NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

 // bind the Object Reference in Naming
 String name = "Hello";
 NameComponent path[] = ncRef.to_name( name );
 //ncRef.rebind(path, href);
 ncRef.rebind(path, ref);

 System.out.println("HelloServer␣ready␣and␣waiting␣...");

 // wait for invocations from clients
 orb.run();
 }
 catch (Exception e) {
        System.err.println("ERROR:␣" + e);
        e.printStackTrace(System.out);
  }
  System.out.println("HelloServer␣Exiting␣...");
 }
}
```

5. Le programme client

Les tâches du programme client sont :

- Création et initialisation de l'ORB ;
- Récupération de la référence de l'objet du service de nommage ;
- Récupération de la référence du servant grâce au service de nommage ;
- Conversion de la référence vers le type Hello en utilisant la méthode narrow() de la classe HelloHelper ;
- Invocation de méthodes distantes.

```
public class HelloClient {
        static Hello helloImpl;

        public static void main(String args[])
        {
         try{
                // create and initialize the ORB
                ORB orb = ORB.init(args, null);

                // get the root naming context
                org.omg.CORBA.Object objRef =
                   orb.resolve_initial_references("NameService");
                NamingContextExt ncRef =
                   NamingContextExtHelper.narrow(objRef);
                // resolve the Object Reference in Naming
                String name = "Hello";
                helloImpl =
                        HelloHelper.narrow(ncRef.resolve_str(name));

                System.out.println(helloImpl.sayHello());

        } catch (Exception e) {
                System.out.println("ERROR : " + e) ;
                e.printStackTrace(System.out);
        }
 }

}
```

6. Exécution

Service de nommage à lancer avant le serveur :

```
orbd -ORBInitialPort <port>
```

Lancer le serveur :

```
java HelloServer -ORBInitialPort <port>
                -ORBInitialHost <servermachinename>
```

Lancer le client :

```
java HelloClient -ORBInitialPort <port>
                -ORBInitialHost <servermachinename>
```