CORBA avec OpenORB

Samir Torki et Patrice Torguet

1 Présentation de CORBA

CORBA (Common Object Request Broker Architecture) 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 version de CORBA (version 2.5) date de septembre 2001. CORBA est basée sur un bus, l'ORB (Object Request Broker), qui assure les collaborations entre applications (fig.1).
Les communications sont basées sur le mécanisme d'invocation de procédure distantes (comme pour les RMIs) 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.

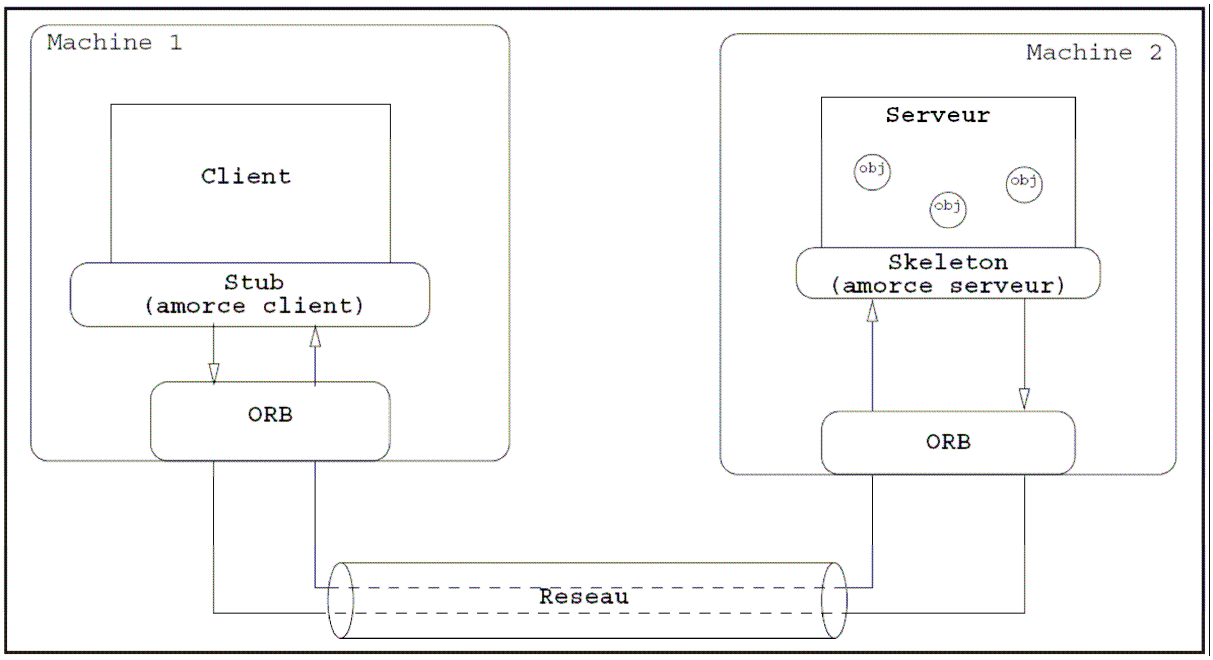

Figure 1 : Architecture CORBA

L'ORB prend généralement la forme d'une bibliothèque de fonctions assurant la communication entre les clients et les serveurs. Dans le cadre des TPs, nous utiliserons l'ORB OpenSource OpenORB (http://openorb.sourceforge.net/).

2 L'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'Interface Definition Language (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 interopérables, les types de base utilisés sont spécifiques au langage IDL (il ne s'agit ni de types Java ni de types C++). Ces types sont ensuite projetés sur les langages dans lesquels sont réalisées les implémentations.
Note : IDL permet de définir l'interface représentant le comportement **public** d'un objet CORBA et de ce fait, ce contrat ne présage en rien du comportement interne et/ou privé de l'objet CORBA (c'est à dire de son implémentation).
Remarque : on appelle souvent servant, l'implémentation d'un objet CORBA.

```
interface Horloge {
       // Corps de l'interface : attributs et méthodes
       /* Rq. Les commentaires utilisent
       la même syntaxe qu'en C, C++,
       Java ...
       */
};
```

Figure 2 : Interface IDL

2.2 Types IDL de base

Les types IDL de base ainsi que les types Java correspondants sont présentés dans la figure 3.

Type IDL	Type Java
boolean	boolean
char	char
wchar	char
octet	byte
string	java.lang.String
wstring	java.lang.String
short	short
unsigned short	short
long	int
long long	long
unsigned long	int
float	float
double	double

Figure 3 : Types IDL de base

La figure 4 présente la description IDL d'un objet Horloge sur lequel on peut appliquer la méthode get_time().
Cette méthode ne prend pas d'arguments et retourne l'heure sous la forme d'une chaîne de caractères (string).

```
interface Horloge {
       string get_time();
};
```

Figure 4 : Interface Horloge

2.3 Passage de paramètres

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.
Ainsi, il est possible de définir une fonction get_time_int()[1] prenant deux arguments entiers heure et minute dans lesquels le serveur placera l'heure courante (fig.5).

```
interface Horloge {
      string get_time_string();
      void get_time_int(out short heure, out short min);
};
```

Figure 5 : Interface Horloge

Note : en plus des paramètres en **out** ou **inout** on peut aussi, bien évidemment, utiliser les retours d'appels pour renvoyer une information du serveur vers le client (cf. la méthode get_time de la figure 4 qui renvoie une chaîne de caractère au client).

3 Génération des amorces

Le contrat IDL va être utilisé pour générer les amorces. Dans un premier temps, vérifiez que OpenORB/lib/openorb-1.3.1.jar et OpenORB/lib/openorb_tools-1.3.1.jar sont dans votre CLASSPATH.
Avec OpenORB, la génération des amorces se fait par l'intermédiaire de la commande :

```
java org.openorb.compiler.IdlCompiler -verbose Horloge.idl -d .
```

La classe IdlCompiler contenue dans openorb_tools-1.3.1.jar correspond au précompilateur IDL. L'option –verbose permet d'obtenir plus de détails lors de la génération.

Au cours de la précompilation, un ensemble de fichiers sont générés et placés dans le répertoire spécifié avec l'option -d. La souche correspond au fichier _HorlogeStub.java et le squelette correspond au fichier HorlogePOA.java.

Exercice : Définissez l'interface IDL correspondant à un objet calculatrice proposant les méthodes suivantes (fig.6) et précompilez le. La méthode extraire_memoire prend en paramètre un argument registre dans lequel sera placé la valeur enregistrée en mémoire.

```
float ajouter(float a1,float a2);
float soustraire(float a1,float a2);
float multiplier(float a1,float a2);
void memoriser_dernier_resultat();
float extraire_memoire(float registre);
```

Figure 6 : Fonctions proposées par l'interface Calculatrice

Ouvrez la souche et le squelette générés et analysez les 3 étapes constituant l'invocation à distance :
1. *marshalling*,
2. invocation de la méthode,
3. *dé-marshalling*.

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 NomInterfaceImpl héritant de la classe NomInterfacePOA (i.e. le squelette).

[1] IDL n'autorise pas la surcharge : deux fonctions de même nom ne peuvent pas coexister au sein d'une même interface.

```
import java.util.Date;
import org.omg.CORBA.ShortHolder;
// La classe d'implementation herite de la classe
// HorlogePOA
public class HorlogeImpl extends HorlogePOA {
      public String get_time_string() {
            return ((new Date()).toLocaleString());
      }
      /* Les classes ...Holder sont utilisés dans le cas de OUT / INOUT
      * pour permettre de modifier les paramètres
      * .... qui sont passés uniquement par valeur en Java
      * (pas de passage de paramètre par référence)
      */
      public void get_time_int(ShortHolder heure, ShortHolder min) {
            Date date = new Date();
            heure.value = (short) date.getHours();
            min.value = (short) date.getMinutes();
      }
}
```

Figure 7: Classe HorlogeImpl

Exercice : Réalisez la classe d'implémentation correspondant à l'interface IDL Calculatrice.

5 Mise en place d'un serveur

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. récupération du POA (le POA – Portable Object Adaptor – est un adaptateur d'objet qui permet de gérer de façon portable plusieurs objets CORBA)
3. création de l'objet d'implémentation (appelé aussi servant)
5. activation du servant dans le POA
4. activation du POA
6. affichage de l'IOR
7. boucle infinie d'attente de la fin de l'exécution de l'ORB

```
public class Server {
      public static void main(String[] args) throws Exception {
            // Initialisation de l'ORB
            org.omg.CORBA.ORB orb = org.openorb.CORBA.ORB.init(args,null);
            // Récupération du POA racine
            org.omg.PortableServer.POA rootPOA =
            org.omg.PortableServer.POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            //      Creation d'une instance de l'objet distant
            HorlogeImpl objet = new HorlogeImpl();
            //      Activation de l'objet dans le POA
            byte[] monId = rootPOA.activate_object(objet);
            //      Activation du POA
            rootPOA.the_POAManager().activate();
            //      On affiche l'IOR et on attends la fin de l'exécution de l'ORB
            //      (attente infinie ici)
            String IOR = orb.object_to_string(rootPOA.servant_to_reference(objet));
            System.out.println("Server running ...\n"+IOR);
            orb.run();
      }
}
```

Figure 8: Serveur

Le lancement du serveur se fait alors par l'intermédiaire de la commande :

```
java -Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB Server
```

Exercice : Mettez en place un serveur fournissant aux clients un objet Calculatrice.

6 Mise en place d'un client

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 (fig.9):

1. initialisation de l'ORB
3. obtention d'une référence générique vers l'objet distant (à partir de l'IOR)[2]
4. conversion de la référence générique (de type org.omg.CORBA.Object) en un type utilisable (Horloge) à l'aide de HorlogeHelper.narrow().
5. invocation de la méthode distante

```
public class Client {
      public static void main(String[] args) {
            // Initialisation de l'ORB
            org.omg.CORBA.ORB orb = org.openorb.CORBA.ORB.init(args,null);
            // On recupere une reference a l'objet distant
            // Il s'agit d'un org.omg.CORBA.Object
            org.omg.CORBA.Object obj = orb.string_to_object("IOR:00000000...");
            // La classe Helper fournit une fonction "narrow" pour obtenir un
            // objet sur lequel on peut invoquer les methodes
            Horloge horloge = HorlogeHelper.narrow(obj);
            // On invoque les m\'ethodes sur l'objet distant comme s'il s'agissait
            // d'un objet local
            System.out.println(horloge.get_time_string());
            org.omg.CORBA.ShortHolder heure = new org.omg.CORBA.ShortHolder();
            org.omg.CORBA.ShortHolder min = new org.omg.CORBA.ShortHolder();
            horloge.get_time_int(heure,min);
            System.out.println(""+heure.value+":"+min.value);
      }
}
```

Figure 9: Client

Le lancement du client se fait alors par l'intermédiaire de la commande :

```
java -Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB Client
```

Exercice : Mettez en place un client invoquant les méthodes proposées par l'objet Calculatrice fourni par le 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 OpenORB, le service de nommage est lancé à l'aide de la commande :

```
java -Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB
  org.openorb.util.MapNamingContext -ORBPort=2001
```

Au niveau du serveur, l'utilisation du service de nommage se déroule en 3 étapes (fig.10):

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 bind.

[2] Il faut remplacer dans le programme IOR:00000… par l'IOR affichée par le programme serveur.

```
import org.omg.CosNaming.*;
public class ServerNamingService {
       public static void main(String[] args) {
          try {
              // Initialisation de l'ORB
              org.omg.CORBA.ORB orb = org.openorb.CORBA.ORB.init(args,null);
              // Récupération du POA racine
              org.omg.PortableServer.POA rootPOA =
              org.omg.PortableServer.POAHelper.narrow(orb.resolve_initial_references("RootPOA"))
              ;
              //      Creation d'une instance de l'objet distant
              HorlogeImpl objet = new HorlogeImpl();
              //      Activation de l'objet dans le POA
              byte[] monId = rootPOA.activate_object(objet);
              //      Activation du POA
              rootPOA.the_POAManager().activate();

              // Recuperation de la reference du service de nommage
              org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");
              // Extraction d'un objet NamingContext correspondant
              NamingContext naming = NamingContextHelper.narrow(ns);
              // On associe l'objet au nom MyWatch
              NameComponent[] name = new NameComponent[1];
              name[0] = new NameComponent();
              name[0].id="MyWatch";
              name[0].kind="";
              naming.bind(name,rootPOA.servant_to_reference(objet));

              // on attends la fin de l'exécution de l'ORB (attente infinie ici)
              orb.run();
          } catch (Exception e) {
                    e.printStackTrace();
          }
       }
}
```

Figure 10 : Serveur utilisant le service de nommage

Au niveau du client, l'utilisation du service de nommage se déroule en 3 étapes (fig.11):
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 générique (org.omg.CORBA.Object) en un type utilisable (Horloge) à l'aide de HorlogeHelper.narrow().

Exercice : Reprenez l'exemple de la calculatrice avec l'utilisation du service de Nommage pour accéder à la référence de l'objet distant.

```
public class ClientNamingService {
      public static void main(String[] args) {
            try {
                  ... Initialisation ORB
                  // Recuperation de la reference du service de nommage
                  org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");
                  // Extraction d'un objet NamingContext correspondant
                  org.omg.CosNaming.NamingContext naming = org.omg.CosNaming.
                  NamingContextHelper.narrow(ns);
                  // On va rechercher la reference a l'objet distant a partir de son nom
                  org.omg.CosNaming.NameComponent[] name = new org.omg.
                  CosNaming.NameComponent[1];
                  name[0] = new org.omg.CosNaming.NameComponent();
                  name[0].id="MyWatch";
                  name[0].kind="";
                  org.omg.CORBA.Object obj = naming.resolve(name);
                  // On recupere l'objet Horloge a partir de la reference
                  Horloge horloge = HorlogeHelper.narrow(obj);
                  // On invoque les méthodes sur l'objet distant comme s'il s'agissait
                  // d'un objet local
                  System.out.println(horloge.get_time_string());
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

Figure 11 : Client utilisant le service de nommage

8 Création de types utilisateurs

Le langage C++ a fortement influencé les premières phases de la conception du langage IDL. Ainsi, ce dernier offre la possibilité de définir de nouveaux types à partir de types existants.

8.1 Les alias

Un alias est un identifiant représentant un type de données. Pour définir un alias, il faut respecter le format présenté dans la figure 12.

```
// typedef type_de_reference nom_de_l_alias
typedef string horaire;
interface Horloge() {
      horaire get_time();
};
```

Figure 12 : Les alias

8.2 Les structures

Lors du passage de paramètres à une opération, il est souvent utile de pouvoir regrouper dans une entité unique un ensemble de valeurs de types différents. Pour cela, IDL propose un notion de structure identique à celle rencontrée en C (fig.13):

```
struct nom_de_la_structure {
      type_du_membre_1 nom_du_membre1;
      type_du_membre_2 nom_du_membre2;
      ...
};
```

Figure 13 : Les structures

Il devient ensuite possible d'utiliser cette structure en tant que type de retour ou type de paramètre d'une fonction. (fig.14).

```
struct time { short hour; short minute; };
interface Horloge {
        void get_time(out time hour_and_minute);
}
```

Figure 14 : Passage d'une structure en paramètre

8.3 Les énumérations

Les énumérations permettent de définir un ensemble de constantes symboliques regroupées sous un même type. Comme pour les structures, la syntaxe utilisée est identique à celle du C++ :

```
enum Jour {
       Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi,
Dimanche
};
```

Figure 15 : Les énumérations

Chaque membre de l'énumération correspond à un entier : Lundi correspond à 0, Dimanche correspond à 6.

8.4 Les tableaux

IDL autorise également la définition de tableaux de taille fixe pour regrouper un ensemble de valeurs de même types. La taille et la dimension du tableau sont fixés dès la définition du type. Un tableau ne peut pas être passé tel quel en paramètre d'une fonction. Pour "passer un tableau en paramètre", il faut passer soit par l'intermédiaire d'une structure (8.2) ou d'un alias (8.1).

```
typedef short Tableau[3];
...
void get_time_array(out Tableau tab);
```

Figure 16 : Les tableaux

8.5 Les séquences

Pour ce qui est des tableaux dynamiques (i.e. dont la taille n'est pas fixée dès la déclaration), IDL propose la notion de séquence. Tout comme pour les tableaux, il n'est pas possible de passer une séquence en paramètre d'une fonction. Il est nécessaire de passer par l'intermédiaire d'un alias ou d'une structure.

```
// sequence < type de la sequence > nom_de_la_sequence;
sequence < float > vecteur;
```

Figure 17 : Les séquences

8.6 Les exceptions

CORBA propose également un mécanisme d'exceptions permettant de signaler un problème d'exécution à l'appelant d'une méthode distante. La déclaration d'une exception est très similaire à celle d'une structure :

```
exception HeureErronnee {
       string raison;
}

interface Horloge {
       void set heure minute(in short heure, in short minute)
              raises(HeureErronnee);
};
```

Figure 18 : Les exceptions

Note : les exceptions déclarées dans le contrat IDL deviendront après, compilation IDL, des exceptions Java dérivant de la classe CorbaUserException.

8.6 Les modules

IDL permet aussi de définir des modules similaires aux packages java :

```
module Temps {
	exception HeureErronnee {
		string raison;
	};

	interface Horloge {
		void set heure minute(in short heure, in short minute)
			raises(HeureErronnee);
	};
};
```

Figure 19 : Les modules

9 Exemples de types utilisateurs

```
module Temps {
	struct Time {
		short hour;
		short minute;
	};
	enum TimeZone {
		Paris,NewYork,Tokyo
	};
	typedef short DateArray[3];
	typedef sequence <short> DateSequence;

	interface Horloge {
		void get_time_struct(out Time h_s);
		string get_time_string();
		void get_time_array(out DateArray a);
		void get_time_sequence(out DateSequence s);
		void get_GMT_time(inout Time time, in TimeZone zone);
	};
};
```

Note : il faut toujours qu'un type (structure, alias, interface, exception…) soit défini avant d'être utilisé.

Exercice : Reprenez le sujet du TP 4 (Banque RMI), mettez le en oeuvre avec CORBA et analysez les classes générées (notamment les fichiers dont le nom se termine par Holder.java).