Introduction à Corba

Eric Goubault et Matthieu Martel

Corba est une architecture distribuée dans laquelle des clients émettent des requêtes à destination d'objets, qui s'exécutent dans des processus serveurs. Ces objets répondent aux clients par la transmission d'informations bien que ces deux éléments (client et serveur) soient localisés dans des espaces mémoire différents, généralement sur des machines distantes.

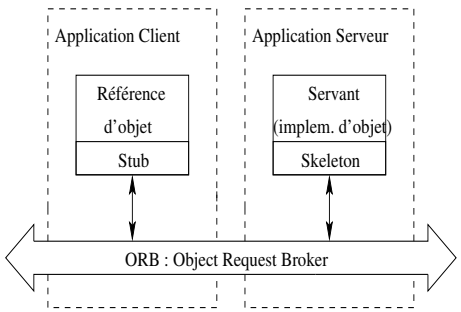

Figure 1: **Le dialogue Corba : Client, Serveur et ORB.**

Lors du développement d'une application Corba, le dialogue entre les différents constituants est rendu totalement transparent pour le développeur. Le serveur déclare mettre à disposition ses objets et le client se contente de demander une connexion à un ou à certains de ces objets, sans pour autant en connaître obligatoirement la localisation ou le format. Dans ces deux cas, l'ORB (Object Request Broker) se charge de localiser les objets, de les charger en mémoire et de transmettre au serveur les demandes du client (voir la Figure 1). Il assure ensuite des opérations de gestion ou de maintenance, comme la gestion des erreurs ou leur destruction. Dans cette architecture, l'application cliente ne se préoccupe pas des détails d'implémentation des objets serveurs, elle se contente de s'y connecter et de les utiliser. L'ORB prend en charge la communication entre les différents composants du système distribué.

1 Le cycle de développement Corba

Le cycle de développement d'une application Corba, illustré par la Figure 2, se compose d'un ensemble d'opérations qui seront autant de points de passage obligatoires. Nous en donnons un aperçu ci-dessous. Ces différentes étapes sont détaillées dans les sections suivantes.

1. *Ecriture de l'interface de l'objet en IDL* : définition dans le langage de description d'interfaces Corba (ou Interface Definition Language pour les anglophones) des opérations disponibles sur l'objet.

2. *Compilation de l'IDL* : génération des modules stub (pour le client) et skeleton (pour le serveur). Ces modules gèrent l'invocation des méthodes distantes.

3. *Implémentation de l'objet* : dérivation d'une classe depuis le skeleton généré à l'étape précédente. Les données et méthodes de cette classe doivent correspondre à celles qui sont décrites dans l'IDL. Lors de l'appel des méthodes sur l'objet, le code défini ici sera exécuté par l'application serveur.

4. *Rédaction de l'application serveur* : ce code sert à initialiser l'objet et à le mettre à disposition des différentes applications clientes.

5. *Compilation du serveur* : génération de l'exécutable de l'application serveur.

6. *Réalisation de l'application cliente* : cette application se connecte à l'objet distant pour lui demander d'exécuter des méthodes. Elle traite ensuite les données transmises par le serveur à l'issue de l'exécution des traitements distants.

7. *Compilation du client* : génération de l'exécutable de l'application client.

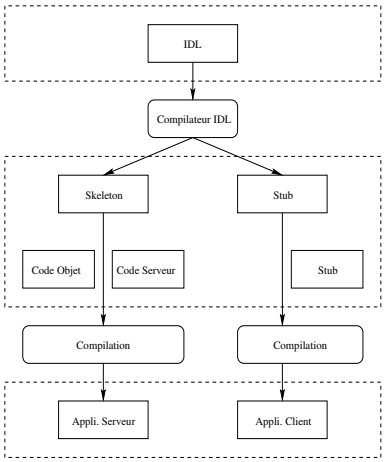

Figure 2: **Cycle de développement Corba.**

2 Ecriture de l'interface de l'objet en IDL

Dans cette première étape, nous allons créer l'interface de l'objet en définissant les opérations fournies par l'objet `Vecteur` (données et méthodes) qui seront accessibles à partir d'applications distantes. Pour cela nous utilisons le langage IDL.

```
interface Vecteur {
  typedef double vect[100]; // Définition du type vect
  attribute long size;      // size est un attribut de type long (entier)
  attribute vect T ;        // T est un attribut de type vect
  double maxElement();      // maxElement est une méthode retournant un double
  void mult(in double a);   // mult est une méthode prenant pour paramètre
                            // un double transmis par le client (mot clé in)
};
```

Les principaux types définis par le langage IDL sont `short` et `long` pour les entiers, `float` et `double` pour les flottants, `char` et `string` pour les caractères. Il est possible de définir des types

composés : des tableaux comme dans l'exemple ci-dessus ainsi que d'autres structures à l'aide des mots clés `enum`, `struct` ou `union`, etc. que l'on ne détaillera pas ici.

`attribute` est un mot clé du langage IDL, tout comme `in` qui indique le sens de passage du paramètre (du client vers le serveur). Symétriquement le type de passage des paramètres `out` permet de réaliser un passage par adresse. Par exemple, nous aurions aussi pu définir la fonction `maxElement` par `void maxElement(out double m);`. Le type `inout` peut être utilisé pour les paramètres devant être lus et écrits.

☞ **Question 1** *Créer un fichier `Vecteur.idl` contenant l'interface ci-dessus.*

3 Compilation de l'IDL

Nous allons maintenant utiliser le compilateur IDL pour générer les fichiers Java correspondant aux modules stub et skeleton de notre interface.

☞ **Question 2** *Compiler l'interface par la commande `idlj -fall Vecteur.idl`. Examiner le contenu de votre répertoire : plusieurs fichiers ont été créés que nous allons décrire ci-dessous.*

Les principaux fichiers qui ont été générés sont les suivants :

- `VecteurOperations.java` : définition de l'interface qui contient la signature des méthodes de l'objet `Vecteur`.
- `Vecteur.java` : définition de l'interface `Vecteur` qui hérite de `VecteurOperations`.
- `VecteurHelper.java` : regroupe des méthodes destinées à vous aider lors de l'utilisation des objets `Vecteur`.
- `VecteurHolder.java` : outils pour prendre en charge le passage de paramètres avec les méthodes de l'objet `Vecteur`.
- `_VecteurStub.java` : stub de l'objet `Vecteur`. Cette classe sera utilisée dans le code de notre client de façon implicite, elle représentera l'objet distant en mettant à notre disposition les méthodes qui s'y trouvent.
- `_VecteurImplBase.java` : squelette de l'objet `Vecteur`. C'est la classe de base de notre futur objet `vecteur`. On en dérivera une classe pour coder le fonctionnement réel de notre objet.

☞ **Question 3** *Avant d'aller plus loin, examiner le contenu de ces différents fichiers.*

4 Implémentation de l'objet

Il nous faut maintenant créer la classe de l'objet que nous souhaitons mettre à disposition des différents clients. Pour cela nous utilisons le squelette généré lors de l'étape précédente. La classe `VecteurImpl` que nous allons créer étendra la classe `_VecteurImplBase` produite par le compilateur IDL et implémentera les méthodes définies par l'interface `VecteurOperations.java`. Voici à quoi ressemble cette classe :

```
public class VecteurImplem extends _VecteurImplBase {

    private int size=100;
    private double T[];

    VecteurImplem(double[] Tab) {
        ...
    }

    VecteurImplem() {} ;

    public int size() {
        return size;
    }

    public void size(int newSize) {
        size = newSize;
    }
    ...
    }
}
```

Pour être utilisable, cette classe doit implémenter toutes les méthodes définies par l'interface `VecteurOperations`. C'est aussi ici que l'on déclare les attributs définis dans l'interface IDL. Il faut naturellement définir les méthodes de création des objets de la classe. La méthode `maxElement` retourne l'élément maximal du vecteur et la méthode `mult` multiplie tous les éléments du vecteur par la valeur du paramètre.

☞ **Question 4** *Ecrire entièrement la classe `VecteurImplem`.*

5 Réalisation du serveur Corba

Nous disposons maintenant d'un objet complet correspondant à l'interface IDL `Vecteur` que nous allons mettre à disposition des applications clientes. Pour cela, nous allons créer une application hôte, un serveur qui contiendra l'objet `Vecteur`. Le code ressemblera à ceci :

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class VecteurServeur {
    public static void main(String args[]) {
        try{
            // Crée et initialise l'ORB
            ORB orb = ORB.init(args, null);
            // Crée le servant et l'enregistre sur l'ORB
            VecteurImplem vecteurRef = new VecteurImplem();
            orb.connect(vecteurRef);
            // Obtention de la reference du RootPOA
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

```
            // Enregistre la reference de l'objet
            NameComponent nc = new NameComponent("Vecteur", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, vecteurRef);
            // attend les invocations de clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }
        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Les principales actions réalisées par le serveur sont :

- Créer une instance de l'ORB.
- Créer une instance du servant (l'implémentation d'un objet `Vecteur`) et l'enregistrer sur l'ORB.
- Obtenir une référence à un objet Corba pour le nommage et y enregistrer le nouvel objet sous le nom "Vecteur".
- Attendre l'invocation de méthodes de l'objet.

De plus amples détails concernant ce point sont disponibles sur la page `http://java.sun.com/products/jdk/1.2/docs/guide/idl/tutorial/GSserver.html`.

☞ **Question 5** *Implémenter le serveur Corba.*

6 Compilation du serveur

☞ **Question 6** *Compiler les fichiers Java relatifs au serveur.*

7 Réalisation du client

Nous avons maintenant une application autonome qui, lors de son lancement, crée un objet `Vecteur` et le met à disposition des autres applications. Tout comme le serveur, le client devra initialiser à l'ORB. Il pourra ensuite se connecter à l'objet et l'utiliser.

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class VecteurClient {
    public static void main(String args[]) {
        try{
            // Cree et initialise l'ORB.
            ORB orb = ORB.init(args, null);
```

```
            // Obtention de la reference du RootPOA
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            // trouve la référence à l'objet
            NameComponent nc = new NameComponent("Vecteur", "");
            NameComponent path[] = {nc};
            Vecteur vecteurRef = VecteurHelper.narrow(ncRef.resolve(path));
            // Utilisation de l'objet
            ...
        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

Le client initialise l'ORB et obtient une référence au contexte de nommage. Il y recherche ensuite "Vecteur" et reçoit une référence de cet objet Corba. Il peut ensuite invoquer les méthodes de l'objet. Des détails complémentaires sont disponibles sur la page `http://java.sun.com/products/jdk/1.2/docs/guide/idl/tutorial/GSapp.html`.

☞ **Question 7** *Implémenter le client Corba.*

8 Compilation du client

☞ **Question 8** *Compiler les fichiers Java relatifs au client.*

9 Exécution de l'application

Il faut procéder dans l'ordre suivant :

- Initialiser le serveur de noms : `tnameserv -ORBInitialPort 1050`. 1050 est le port utilisé par le serveur. Sous Solaris, il faut être root pour utiliser un port inférieur à 1024.
- Initialiser le serveur :

  `java VecteurServer -ORBInitialHost namerserverhost -ORBInitialPort 1050`.

  `nameserverhost` est le nom de la machine sur laquelle s'exécute le serveur de noms.
- Exécuter le client :

  `java VecteurClient -ORBInitialHost namerserverhost -ORBInitialPort 1050`

☞ **Question 9** *Créer plusieurs serveurs d'objets `Vecteur` sur des machines différentes. Créer un client manipulant un grand vecteur `grandV`. Pour réaliser des opérations sur `grandV`, le client le décomposera en sous-vecteurs qui seront traités par des serveurs différents.*