# *Objets distribués - Corba - Code Mobile*

Damien Olivier

Damien.Olivier@univ-lehavre.fr

Laboratoire d'informatique du Havre

# *Introduction*

On évoque les technologies actuelles permettant la distribution d'objets dans des environnements de réseaux informatiques : les objets distribués avec Java/RMI, les brokers d'objets à la norme CORBA, les agents mobiles, les applications aux au e-commerce.

# Organisation du cours

1 Concepts sur les systèmes distribués à objets et l'IDL/CORBA *(Lundi 20/01/2003 - D. Olivier)*

2 Les bases de CORBA : ORB, projections, invocation statique *(Mardi 21/01/2003 - D. Olivier)*

3 Service de nommage, adaptateur d'objet (POA) et référentiel d'interfaces en CORBA *(Vendredi 24/01/2003 - D. Olivier)*

4 Les mécanismes dynamiques en CORBA *(Lundi 27/01/2003 - D. Olivier)*

5 Codes mobiles et Aglets *(Mardi 28/01/2003 - D. Olivier)*

6 Applications professionnelles : le e-commerce (1) *(Lundi 18/02/2003 - B. Adouobo)*

7 Applications professionnelles : le e-commerce (2) *(Lundi 19/02/2003 - B. Adouobo)*

8 Applications professionnelles : le e-commerce (3) *(Mardi 20/02/2003 - B. Adouobo)*

# Bilio / Webo graphie de base

- OMG : `http://www.omg.org`
- Livres :

“Objets répartis, guide de survie”

“Client server programming with Java & Corba”

*Orfali, Edwards, Harley - ITP France*

“CORBA : des concepts à la pratique”

*Geib, Gransart, Merle - Masson 97*

“The CORBA reference guide”

*A. Pope - Addison Wesley 97*

“Au cœur de Corba avec Java”

*J. Daniel - Vuibert 2000*

- Web :

ORBacus `http://www.ooc.com`

CorbaScript `http://corbaweb.lifl.fr`

D. Schmidt Page perso `http://www.cs.wustl.edu/~schmidt/`

U. Vienne `http://www.infosys.tuwien.ac.at/Research/Corba/`

Portail `http://www.cetus-links.org/oo_corba.html`

# 6. Les mécanismes dynamiques en CORBA

# 6.1 Les types dynamiques

- Les types dynamiques sont à la base des mécanismes dynamiques du Bus Corba : ils permettent de découvrir dynamiquement les types de données qui y sont véhiculées.
- Ils sont auto-descriptifs : Ils contiennent donc à la fois une valeur et une information décrivant le type de celle-ci.
- Les principaux sont any et `TypeCode`.

# 6.1.1 Le type Any

- Le type any est un type dynamique qui contient, de manière interne :
  - la valeur effective de la donnée ;
  - la description du type de la valeur (de type CORBA : :TypeCode).
- Le receveur d'une structure de type any est ainsi toujours capable d'interpréter correctement cette information auto-suffisante.

# 6.1.1 Le type Any

## Pourquoi le type any ?

- Gestion par une invovation dynamique, du résultat d'une requête sans connaître à priori sa nature.
- Permet de donner un caractère générique à des attributs IDL.
- Exemple : Représentation IDL d'une liste de longueur arbitraire contenant des paramètres arbitraires.

```
typedef sequence<any> ListGenerique;
```

- Par ailleurs, spécification récente d'un type DynAny pour construire dynamiquement des valeurs any ...

# 6.1.1 Le type Any

## Projection/Utilisation du type Any en Java

- Créer : On utilise l'opération `create_Any` de la classe `ORB`. On notera la différence de syntaxe (any en IDL, Any en Java ou C++).
  Exemple :
  ```
  org.omg.CORBA.Any a = orb.create_any();
  ```
- Insérer une valeur : On utilise une méthode `insert_xxx` (où `xxx` désigne un nom de type) de la classe `org.omg.CORBA.Any`.
  Exemple :
  ```
  a.insert_string("Bonjour");
  a.insert_long(6);
  a.insert_float(3.14);
  ```

# 6.1.1 Le type Any

## Projection/Utilisation du type Any en Java

- Insérer une valeur d'un type utilisateur préalablement décrit en IDL : On utilise la classe `Helper` associé au type qui contient elle-même une opération `insert`.

  Exemple : soit la description IDL suivante

```
struct personne
{ string nom; string adresse;};
```

  Après compilation, une classe `Helper` nommée `personneHelper` est générée pour la structure `personne`.

```
personne p = new personne();
p.nom="moi"; p.adresse="ici";
org.omg.CORBA.Any a = orb.create_any();
personneHelper.insert(a, p);
```

# 6.1.1 Le type Any

## Projection/Utilisation du type Any en Java

- Extraire une valeur d'un Any : On utilise une méthode `extract_xxx` (où `xxx` désigne un nom de type) de la classe `org.omg.CORBA.Any`.

```
float x = a.extract_float();
```

Pour les types utilisateurs, on utilise la méthode `extract` de la classe `Helper` correspondante.

```
personne p = personneHelper.extract(a);
```

Le problème qui se pose est bien évidemment de savoir

Comment connaître le type de données contenue dans un Any ?

C'est la notion de `TypeCode`, que l'on présentera plus loin, qui permet d'y répondre.

# 6.1.1 Le type Any

Projection/Utilisation du type Any en C++

- Créer : On effectue une déclaration à l'aide du type `CORBA : :Any`.
  Exemple :
  ```
  CORBA::Any a ;
  ```
- Insérer une valeur : On utilise l'opérateur <<= (écriture dans un flux CORBA).
  Exemple :
  ```
  a <<= "Bonjour";
  a <<= (CORBA::Long) 65;
  ```
  L'insertion d'une valeur d'un type utilisateur préalablement décrit se fait de la même manière. La projection en C++ de l'interface génèrera automatiquement les opérateurs de surcharges.
- Extraire une valeur : On utilise l'opérateur >>= (lecture à partir d'un flux CORBA).
  Exemple :
  ```
  CORBA::Long val;
  a >>= val;
  ```
  La question est donc toujours :

  Comment connaître le type de données contenue dans un Any ?

# Commentaire

```
a <<= "Bonjour";
a <<= (CORBA::Long) 65;
```

- La première insertion réalise un “copie profonde” de Bonjour ;
- La seconde insertion désalloue Bonjour et insére 65. On remarquera le cast sur la constante entière C++.

# 6.1.1 Le type Any

## Projection/Utilisation du type Any en C++

Particularités à prendre en compte dans le cas de la projection du type any en C++ :

- La projection IDL en C++ permet que différents types IDL soient projetés sur le même type C++. `IDL char`, `boolean`, `octet` peuvent tous les trois être projetés au final dans le type `C++ char`. Il faut alors utiliser des méthodes spécifiques de la classe `CORBA : :Any`, à savoir `from_xxx` pour insérer ou extraire ces valeurs.

  Exemple :

```
CORBA::Any a;
CORBA::Char c = 'x';
a <<= c     // genere une erreur de compilation !
a <<= CORBA::Any_from_char(c); // Correct !
```

# 6.1.1 Le type Any

## Projection/Utilisation du type Any en C++

Particularités à prendre en compte dans le cas de la projection du type any en C++ :

- L'insertion d'un pointeur (une chaîne de caractère, par exemple) dans un any revient à recopier son adresse. Il faudra veiller à ne pas désallouer le pointeur inséré par la suite : ce qui désallouerait aussi l'adresse stockée dans la variable any.

```
CORBA::Any a;
const char *p;
a <<= ''Y a un bug'';
a >>= (char *)p;
cout << ''Le contenu de a extrait dans p est : '' << p << endl;
a <<= (CORBA::Long) 65;
cout << ''p = '' << p << endl; // OUPS .....
```

# 6.1.1 Le type Any

## Projection/Utilisation du type Any en C++

Particularités à prendre en compte dans le cas de la projection du type any en C++ :

- Pour utiliser un tableau dans un type Any, on ne pourra pas utiliser la surcharge de opérateurs précédents et on devra utiliser une méthode `aliasDuTypeTableau_forany(nom du tableau)`, générée automatiquement. Le tableau extrait n'est pas modifiable, il faut le copier avec `aliasDuTypeTableau_dup`.

# 6.1.2 Les TypeCodes et TCKinds

- Tout élément IDL (standard ou défini par l'utilisateur) est associé à un `TypeCode` lui-même de type `CORBA : :TypeCode` (ou encore `org.omg.CORBA.TypeCode` en Java).
- La classe `TypeCode` fournit des opérations retournant des informations de description.
- Les types sont rangés en "genre de type", appelés `TCKind`

# 6.1.2 Les TypeCodes et TCKinds

## Le type TCKind

- Le type IDL `TCKind` (`CORBA : :TCKind` en C++ ou `org.omg.CORBA.TCKInd` en Java) est une énumération qui attribue à chaque type prédéfini `xxx`, un identifiant `tk_xxx` ou encore l'identifiant `tk_objref` à toute interface définie par l'utilisateur.

# 6.1.2 Les TypeCodes et TCKinds

## Utilisation d'un TypeCode

- **Connaître le genre du type décrit**
  On appelle la méthode `kind` de la classe `TypeCode` qui renvoie un `TCKind` ;
- **Connaître le nom et l'ID du type décrit**
  On utilise respectivement les méthodes `id` et `name` (Attention : cela ne s'applique pas aux types primitifs) ;
- **Les opérations et les membres**
  Pour les types IDL comportant des membres (structures, énumérations, ...), on dispose des opérations : `member_count` (nombre de membres présents dans le type), `member_name(i)` (nom du membre d'indice i), `member_type(i)` (TypeCode du membre d'indice i)

# 6.1.2 Les TypeCodes et TCKinds

Connaître le type d'un Any Revenons à notre problème précédemment posé : comment connaître le type de la valeur contenue dans un type Any ?

La classe Any propose l'opération `type` qui renvoie le TypeCode de la valeur qu'il contient. A partir de celui-ci, on peut donc connaître précisément la nature de la valeur contenue dans l'Any.

# 6.1.2 Les TypeCodes et TCKinds

## Exemple d'utilisation d'un TypeCode

- Interface IDL

```
struct sExemple {
   long membreA ;
   string membreB;
};
```

- Opérations de manipulation en Java

On dispose d'un objet de type `sExemple` sans en connaître la nature. On la découvre alors de manière dynamique.

```
org.omg.CORBA.TypeCode tc =
  sExempleHelper.type();

  // on doit alors comparer la valeur du
  // type avec tous les types connus ...
  if (tc.kind().value ==
      org.omg.CORBA.TCKind.tk_struct)
    {
      System.out.println(
       "Nom de la structure : " +
        tc.name());
      System.out.println(
       "ID de la structure : " + tc.id());

      System.out.println(
       "Nombre de membres : " +
       tc.member_count());
      for (int i=0; i<tc.member_count(); i++)
      {
       System.out.println(
        "Nom du membre d'ordre " + i +
        " : " + tc.member_name(i));
      }
    }
```

# 6.1.2 Les TypeCodes et TCKinds

- Opérations de manipulation en C++

```
#include <OB/CORBA.h>
#include "sExemple.h"

using namespace std;

int main(int argc, char**argv)
{
  sExemple ex;
  CORBA::Any a;
  CORBA::TypeCode_var tc;
  int i;

  a <<= ex;
  tc = a.type();
  switch(tc->kind())
    {
      //....
    case CORBA::tk_struct :
      cout << "Nom de la structure : "
        << tc->name() << endl;
      cout << "ID de la structure : "
        << tc->id() << endl;
      cout << "Nombre de membres : "
        << tc->member_count() << endl;
      for (i=0; i<tc->member_count(); i++)
        cout << "Nom du membre d'ordre "
         << i << " : "
         << tc->member_name(i) << endl;
    }
  //...
}
```

# 6.1.2 Les TypeCodes et TCKinds

## Exemple d'utilisation d'un TypeCode

- Résultat de l'exécution

```
Nom de la structure : sExemple
ID de la structure : IDL:sExemple:1.0
Nombre de membres : 2
Nom du membre d'ordre 0 : membreA
Nom du membre d'ordre 1 : membreB
```

# 6.2 Scénario d'une invocation dynamique

1 Trouver la référence de l'objet : On peut utiliser, par exemple, le servive de nommage ;

# 6.2 Scénario d'une invocation dynamique

1. Trouver la référence de l'objet : On peut utiliser, par exemple, le servive de nommage ;
2. Obtenir l'interface de l'objet puis la description de l'opération à invoquer : on utilise l'IFR pour consulter l'interface de l'objet (cf. chap. 5, s. 179-184). Il reste à alors à sélectionner (par comparaison des chaînes des noms) l'opération voulue ;

# 6.2 Scénario d'une invocation dynamique

1. Trouver la référence de l'objet : On peut utiliser, par exemple, le servive de nommage ;
2. Obtenir l'interface de l'objet puis la description de l'opération à invoquer : on utilise l'IFR pour consulter l'interface de l'objet (cf. chap. 5, s. 179-184). Il reste à alors à sélectionner (par comparaison des chaînes des noms) l'opération voulue ;
3. Construire la requête ...

# 6.2 Scénario d'une invocation dynamique

1. Trouver la référence de l'objet : On peut utiliser, par exemple, le servive de nommage ;
2. Obtenir l'interface de l'objet puis la description de l'opération à invoquer : on utilise l'IFR pour consulter l'interface de l'objet (cf. chap. 5, s. 179-184). Il reste à alors à sélectionner (par comparaison des chaînes des noms) l'opération voulue ;
3. Construire la requête ...
4. Invoquer la requête ...

# 6.2 Scénario d'une invocation dynamique

1. Trouver la référence de l'objet : On peut utiliser, par exemple, le servive de nommage ;
2. Obtenir l'interface de l'objet puis la description de l'opération à invoquer : on utilise l'IFR pour consulter l'interface de l'objet (cf. chap. 5, s. 179-184). Il reste à alors à sélectionner (par comparaison des chaînes des noms) l'opération voulue ;
3. Construire la requête ...
4. Invoquer la requête ...
5. Traiter les résultats ...

# 6.2 Scénario d'une invocation dynamique

1. Trouver la référence de l'objet : On peut utiliser, par exemple, le servive de nommage ;
2. Obtenir l'interface de l'objet puis la description de l'opération à invoquer : on utilise l'IFR pour consulter l'interface de l'objet (cf. chap. 5, s. 179-184). Il reste à alors à sélectionner (par comparaison des chaînes des noms) l'opération voulue ;
3. Construire la requête ...
4. Invoquer la requête ...
5. Traiter les résultats ...

Les trois derniers points vont maintenant être développés.

# 6.2.1 Construire la requête

On doit d'abord créer un objet CORBA, `request` qui représente la requête elle-même. Ensuite plusieurs solutions sont possibles :

- On construit une liste spécifique pour ranger les arguments de la requête ;
- On insère directement les arguments avec les méthodes appropriées.

On présente juste après une version C++ mettant en œuvre la première solution qui est la plus complète des deux. On présentera la deuxième solution dans la version Java.

# 6.2.1 Construire la requête

Les étapes de construction, pour la solution choisie sont donc :

1 On crée une requête vide en appelant la méthode ad-hoc prédéfinie sur l'objet à invoquer qui est ici `objcompteur` :

```
CORBA::Request_var request_set =
    objcompteur->_request("set_somme");
```

2 On crée une liste vide d'arguments, de type `NVList`, chargée de contenir les arguments correspondant à la requête :

```
CORBA::NVList_ptr list_arg =
    request_set->arguments();
```

# 6.2.1 Construire la requête

3 On insère les arguments de la requête

- Chaque argument doit être de type `Any` ;
- Chaque argument est associé à un identificateur ;
- Il faut préciser le mode de passage (`in`, `out` ou `inout`) pour chaque argument.

Il faut alors effectuer les opérations suivantes :

- Insérer l'identifiant de l'argument dans la requête elle-même ;
- Insérer le triplet (identifiant, valeur de l'argument, mode de passage) dans la liste.

```
CORBA::Any initialisation;
initialisation <<= (CORBA::Long) 0;
request_set->add_in_arg("initialisation");
list_arg->add_value("initialisation",
                    initialisation,
                    CORBA::ARG_IN);
```

# 6.2.1 Construire la requête

4 On précise le type de la valeur de retour attendue de l'invocation.

```
request_set->set_return_type(CORBA::_tc_void);
```

# 6.2.2 Invoquer la requête

- On appelle la méthode :

```
request_set->invoke();
```

C'est une invocation synchrone qui bloque l'application jusqu'au retour du résultat.

Corba propose 2 autres modes d'invocation :

- L'invocation différée (`send_deferred`) : permet à l'application de continuer son exécution pendant l'exécution de la requête. On obtient ultérieurement le résultat :
  - soit par une attente bloquante (`get_response`),
  - soit par scrutation non bloquante (`poll_response`).

  Ce type d'invocation n'est possible qu'avec le DII (en non dynamique, il faut threader ! cf. TP).
- L'invocation asynchrone (`send_oneway`) : la méthode invoquée doit être elle-même déclarée `oneway`. Elle ne renvoie pas de valeur de retour et donc on ne récupère pas de résultat de la requête.

# 6.2.3 Traiter les résultats

L'instruction suivante permet de récupérer la valeur de retour de la requête :

```
request->return_value() >>= valeur;
```

Il est également possible de récupérer des paramètres de sortie (en `inout` ou `out`) : ceux-ci sont entreposés dans la liste des arguments passée à la création de la requête et à laquelle on accède par la méthode `arguments` de l'objet requête.

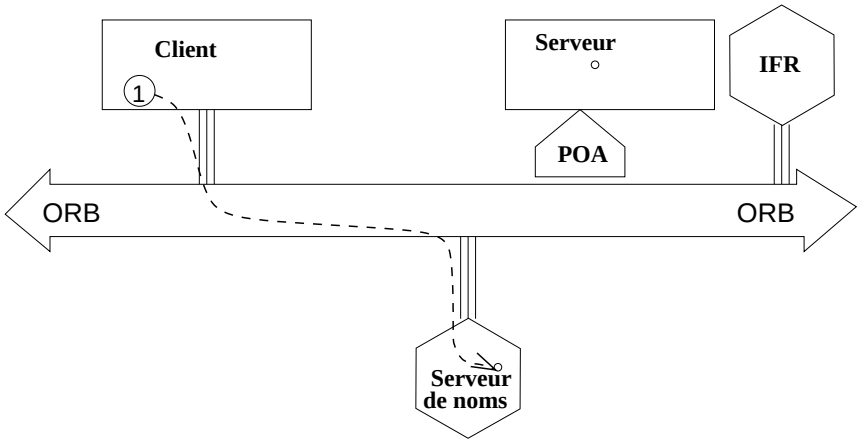
# 6.2.4 Requête ... conclusion

Pour conclure sur la gestion des requêtes, il faut savoir que d'autres scénari sont possibles utilisant éventuellement d'autres objets et d'autres méthodes, comme, par exemple la méthode `create_request` de l'objet ORB.

Réf. : le livre d'Orfali et Harkey donné en bibliographie.

# 6.3 Un client dynamique en C++

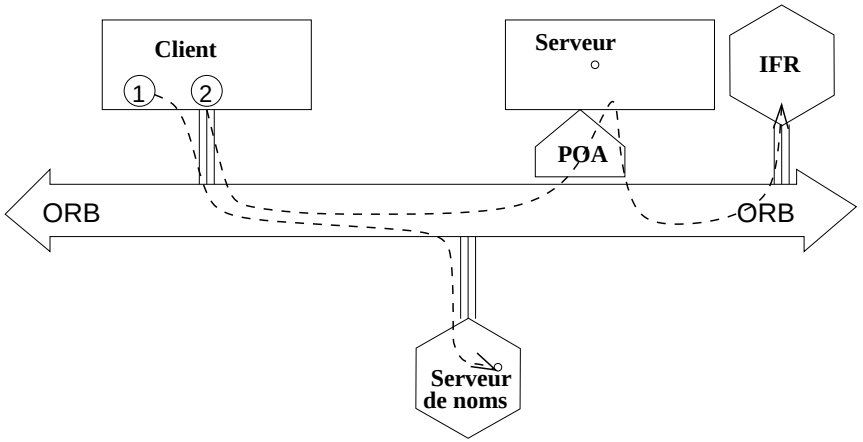
L'invocation proposée ici se fait en 3 étapes :




- Le client recherche l'objet à invoquer sur le serveur de noms ;

# 6.3 Un client dynamique en C++

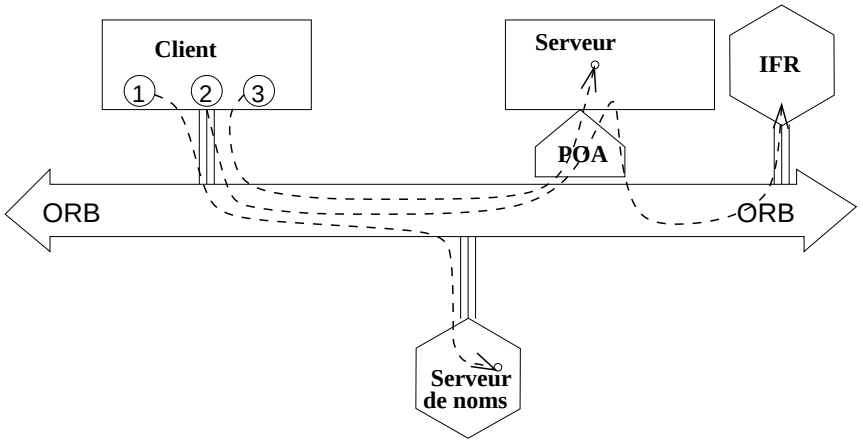
L'invocation proposée ici se fait en 3 étapes :




- Le client recherche l'objet à invoquer sur le serveur de noms ;
- Le client va rechercher la description de l'interface de l'objet dans l'IFR.
  ⇒ La méthode que l'on utilisera consiste à propager cette recherche au serveur qui est lié à l'IFR.
  ⇒ Le client n'a alors pas besoin de référencer l'IFR.

# 6.3 Un client dynamique en C++

L'invocation proposée ici se fait en 3 étapes :




- Le client recherche l'objet à invoquer sur le serveur de noms ;
- Le client va rechercher la description de l'interface de l'objet dans l'IFR.
  ⇒ La méthode que l'on utilisera consiste à propager cette recherche au serveur qui est lié à l'IFR.
  ⇒ Le client n'a alors pas besoin de référencer l'IFR.
- Le client construit sa requête dynamique puis fait sa requête à l'objet cible du serveur.

# 6.3 Un client dynamique en C++

On reprend l'exemple du compteur auquel on va accéder par invocation dynamique.
Il faut faire une invocation dynamique à l'attribut somme, notamment pour l'initialiser. Pour faire cela, on doit utiliser des méthodes d'accès prédéfinies qui s'appellent alors (attention à l'_).

- `_set_somme` pour affecter une valeur à l'attribut ;
- `_get_somme` pour renvoyer la valeur de l'attribut.

# 6.3 Un client dynamique en C++

```
#include <iostream>
#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include <OB/DII.h>
#include <OB/IFR.h>
using namespace std;

int main(int argc, char**argv)
{
  CORBA::ORB_var orb;
  int i;
  try {
    cout <<"initialisation de l'ORB\n'';
    orb = CORBA::ORB_init(argc, argv);
  } catch(...) {
    cerr << "Impossible d'init. l'ORB\n";
    return 1;
  }
 CORBA::Object_var objet =
    orb->resolve_initial_references(
                        "NameService");
 CosNaming::NamingContext_var ns =
    CosNaming::NamingContext::_narrow(objet);
  CosNaming::Name nom;
  nom.length(2);
  nom[0].id = CORBA::string_dup("CTX");
  nom[0].kind = CORBA::string_dup("");
  nom[1].id = CORBA::string_dup("Compteur");
  nom[1].kind = CORBA::string_dup("");
  CORBA::Object_var objcompteur =
     ns->resolve(nom);
```

```
   objcompteur->_get_interface();
// On demande la description de l'interface
CORBA::InterfaceDef::FullInterfaceDescription
    idesc = idef->describe_interface();
cout << "name= " << idesc->name << endl;
cout << "id= " << idesc->id << endl;
cout << "Operations : " << endl;
for(i = 0; i < idesc->operations.length(); i+
 cout << idesc->operations[i].name << endl;
cout << "attributs :" << endl;
for(i = 0 ; i < idesc -> attributes.length()
 cout << idesc -> attributes[i].name << endl;
CORBA::Request_var request1;
CORBA::Request_var request2;
CORBA::NVList_ptr list_arg;
CORBA::Any initialisation;
// Creation d'une requete vide
request1 = objcompteur->_request(
                      "_set_somme");
// Cration de la liste d'arguments
// que l'on passe  la mthode
orb->create_list(0, list_arg);
list_arg = request1->arguments();
// On met dans la liste les arguments
initialisation <<= (CORBA::Long) 5;
request1->add_in_arg("initialisation");
list_arg->add_value("initialisation",
                    initialisation,
                    CORBA::ARG_IN);
request1->set_return_type(CORBA::_tc_void);
request1->invoke();
cout << "initialisation du compteur  5"
```

# 6.3 Un client dynamique en C++

```
  for(i=0; i < 10; i++)
    {
      request2 = objcompteur->_request("increment");
      cout << "increment"; cout.flush();
      request2->set_return_type(CORBA::_tc_long);
      CORBA::Long valeur;
      cout << i << ' '; cout.flush();
      request2->invoke();
      request2->return_value() >>= valeur;
      cout << "resultat = " << valeur << endl;
      request2->invoke();
      request2->return_value() >>= valeur;
      cout << "resultat = " << valeur << endl;
    }
    cout << "resultat = " << valeur << endl;

  exit(0);
}
```

- On aurait pu utiliser `idesc->operations[i].name` plutôt qu' `increment`.

# 6.4 Un client dynamique en Java

```
// client_compteur.java
package M_compteur;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.io.*;
import org.omg.CosNaming.*;
import
 org.omg.CosNaming.NamingContextPackage.*;

public class client_compteur {
 public static void main(String args[]) {
  int i,j;
  String reponse;

  // desactivation de l'orb du jdk
  // pour le remplacer par celui d'orbacus
  java.util.Properties prop =
   System.getProperties();
  prop.put("org.omg.CORBA.ORBClass",
           "com.ooc.CORBA.ORB");
  prop.put("org.omg.CORBA.ORBSingletonClass",
           "com.ooc.CORBA.ORBSingleton");
  ORB orb = null;

  try {
    orb = ORB.init(args, prop);
    org.omg.CORBA.Object objet =
    orb.resolve_initial_references(
                      "NameService");
    NamingContext ns =
      NamingContextHelper.narrow(objet);
```

```
    nom[0].id = "CTX";
    nom[0].kind = "";
    nom[1] = new NameComponent();
    nom[1].id = "Compteur";
    nom[1].kind = "";
    org.omg.CORBA.Object objcompteur =
      ns.resolve(nom);

    org.omg.CORBA.Object defObj =
      objcompteur._get_interface_def();
    if (defObj == null)
      System.out.println("pb IFR !!");
    InterfaceDef idef =
      InterfaceDefHelper.narrow(defObj);

    org.omg.CORBA.InterfaceDefPackage.
              FullInterfaceDescription
         idesc = idef.describe_interface();

    System.out.println("name = " + idesc.name);
    System.out.println("id = " + idesc.id);
    System.out.println("operations : ");
    for (i=0; i<idesc.operations.length; i++)
      {
       System.out.println(i + " : " +
             idesc.operations[i].name);
      }

    Request request1 =
     objcompteur._request("_set_somme");
    request1.add_in_arg().insert_long(5);
    request1.invoke();
```

# 6.4 Un client dynamique en Java

```
 Request request2;
 long resultat=0;

 for (i=0; i<10; i++) {
  request2 = objcompteur._request("increment");
  request2.set_return_type(
    orb.get_primitive_tc(TCKind.tk_long));
  request2.invoke();
  resultat =
    request2.return_value().extract_long();
 }
   System.out.println("resultat : " + resultat);
   System.out.println("Arret du serveur (o/n) ? : ");
   BufferedReader in =
      new BufferedReader(
           new InputStreamReader(System.in));
   reponse  = in.readLine();
   if (reponse.equals("o") || reponse.equals("O")) {
      System.out.println( "arret du serveur demande");
      Request request3 =
         objcompteur._request("desactivation");
      request3.invoke();
    }
}
catch(Exception erreur) {
    System.out.println("Exception declanchee : "
                       +erreur);
    System.exit(1);
}
```

# 6.5 Gestion et lancement de l'application dynamique répartie

Construction d'un fichier de configuration `orb.config` :

```
ooc.orb.service.NameService=corbaloc::
   unemachine.univ-lehavre.fr:5000/NameService
ooc.orb.service.DefaultRepository=corbaloc::
   unemachine.univ-lehavre.fr:5003/DefaultRepository
```

- Lancement du serveur de nom :
  `/usr/local/bin/nameserv -OAport 5000`
- Lancement du référentiel d'interfaces :
  `irserv --ior compteur.idl > Repository.ref`
- Lancement du serveur :
  `serveur -ORBservice InterfaceRepository 'cat Repository.ref' -ORBconfig orb.config`
- Lancement du client :
  `client -ORBconfig serveur.conf}`

## 6.6.1 Interface de squelette dynamique (DSI)

- C'est l'analogue du mécanisme dynamique mais ici appliqué aux serveurs. Un serveur n'a plus besoin d'implanter un squelette pour chaque objet mais il utilise une interface générique s'appliquant à n'importe quel service.
- On fait donc référence à des implémentations de services non interfacés ... plus vraiment d'objet Corba !
- Fonctionnement transparent pour le client.
- A l'origine, le processus a été mis en place pour gérer les passerelles entre différents bus Corba. On peut ainsi faire transiter des requêtes d'un ORB à un autre, à travers un bus non-IIOP (Protocole d'interopérabilité ... cf. paragraphe suivant). Pour cela il faut écrire une passerelle entre un protocole IIOP vers un protocole propriétaire : c'est ce que permet le DSI.
- Ce mécanisme permet donc aussi l'intégration de serveurs spécifi ques et propriétaires existants dans un ORB.

# 6.6.1 Interface de squelette dynamique (DSI)

## Interface ServerRequest

Cette interface représente une requête à destination d'un objet implanté dynamiquement par un serveur.

Elle fournit des opérations pour obtenir

- le nom de l'opération invoquée `op_name` ;
- l'objet de l'IFR décrivant la signature de l'opération invoquée `op_def` ;
- les variables de contexte `ctx` ;
- la liste des paramètres `params` ;
- l'opération `result` permet d'indiquer la valeur de retour de la requête ;
- l'opération `exception` permet d'indiquer l'exception éventuellement déclanchée.

Voici la partie de l'interface de l'ORB qui la décrit :

```
module CORBA {
  ...
  typedef string Identifier;
  ...
  interface ServerRequest {
    Identifier op_name();
    OperationDef op_def();
    Context ctx();
    void params (inout NVList params);
    void result (in any r);
    void exception (in any r);
  };};
```

# 6.6.2 Interopérabilité

Spécifie la gestion des communications entre ORB.
Précise les protocoles d'échanges utilisés :

- GIOP (General Inter-ORB Protocol) protocole générique avec définition commune des données, des références d'objets et des messages de communication.
- IIOP (Internet IOP) implémentation de GIOP au-dessus de TCP-IP
- ESIOP (Environment Specific IOP) pour répondre aux préoccupations de certains membres de l'OMG ... implémenté au-dessus de DCE/RPC devient DCE-ESIOP

# 6.6.3 Le référentiel des implémentations (IMR)

Dans les exemples précédents, on a procédé au lancement des serveurs puis des clients.
Dans la réalité, si un client a besoin d'un service, celui-ci risque de ne pas être actif au moment utile ! ... On ne peut pas mettre en attente tous les services disponibles.
La requête va passer par l'IMR qui propose un service de liaison indirect sur des objets persistents, en s'occupant de faire démarrer les serveurs nécessaire à une requête.

Les spécifi cations CORBA de l'OMG ne standardisent pas comment les serveurs et l'IMR interagissent et cette interface est alors strictement propriétaire de l'éditeur de l'ORB. Par contre, les interactions entre le client et l'IMR sont défi nies dans les spécifi actions du GIOP.