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`

4. Les bases de CORBA

4.1 Des implémentations d'ORB

Il en existe un grand nombre en croissance constante ... parmi lesquel :

- Visibroker (Borland-Inprise/Visigenic)
- ILU (Xerox)
- ORBit (GNU/ Red Hat)
- Mico et JacORB (Open Source)
- Orbix, ORBacus (Iona)
- ... etc ...

4.2 Caractéristiques d'ORBacus

- Utilisation "académique" libre
- Multi-plateformes/OS
- Projections C++ et Java
- Différents services : noms, évènements, propriétés, ...

4.3 Scénario de développement

1 Ecriture des contrats IDL

4.3 Scénario de développement

1 Ecriture des contrats IDL
2 Projections des interfaces

4.3 Scénario de développement

1 Ecriture des contrats IDL
2 Projections des interfaces
3 Implémentation du service

4.3 Scénario de développement

1. Ecriture des contrats IDL
2. Projections des interfaces
3. Implémentation du service
4. Implémentation du gestionnaire de service et du client

4.3 Scénario de développement

1. Ecriture des contrats IDL
2. Projections des interfaces
3. Implémentation du service
4. Implémentation du gestionnaire de service et du client
5. Raccordement au Bus

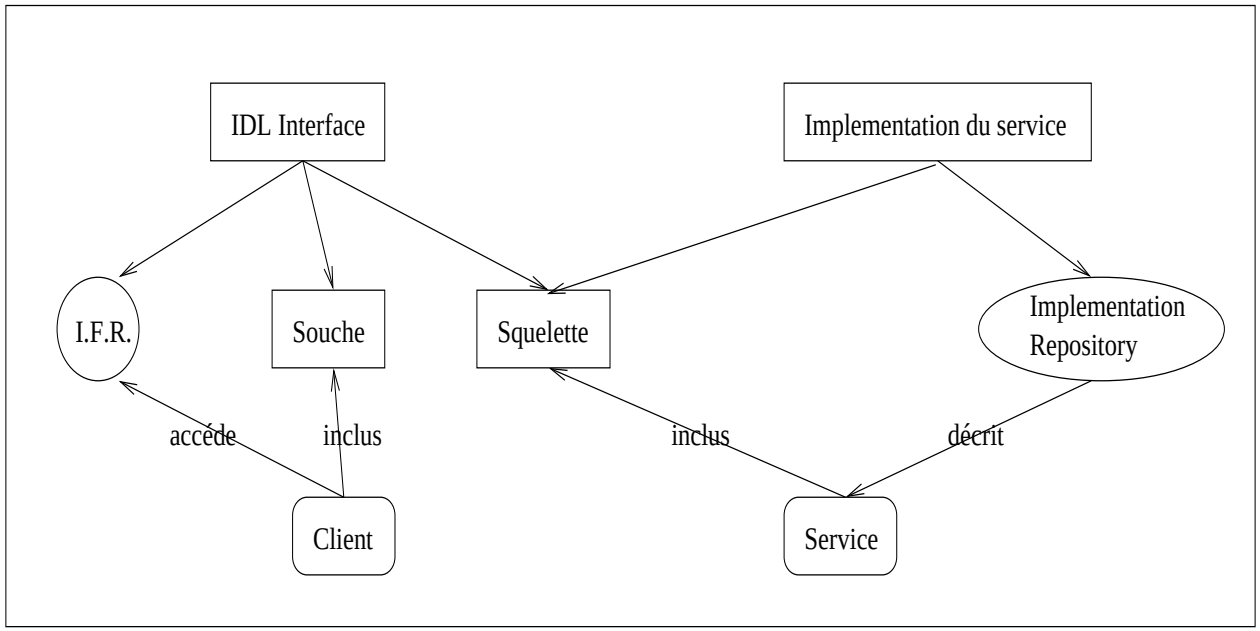
4.3 Scénario de développement

1. Ecriture des contrats IDL
2. Projections des interfaces
3. Implémentation du service
4. Implémentation du gestionnaire de service et du client
5. Raccordement au Bus
6. Lancement du service

4.3 Scénario de développement

1 Ecriture des contrats IDL
2 Projections des interfaces
3 Implémentation du service
4 Implémentation du gestionnaire de service et du client
5 Raccordement au Bus
6 Lancement du service
7 Lancement du client

4.3 Scénario de développement



Commentaire

- Les interfaces des objets sont définies en IDL
- La définition des interfaces correspond à :
  - à des objets dans l'Interface Repository ;
  - du code souche (stubs) pour le client ;
  - du code squelette (skeleton) pour le service ;
  - les réalisations sont décrites par des objets dans l'implementation Repository.

4.4 Construction d'une interface IDL

Exemple de mise en place d'un service élémentaire permettant de se focaliser sur les technologies Corba.

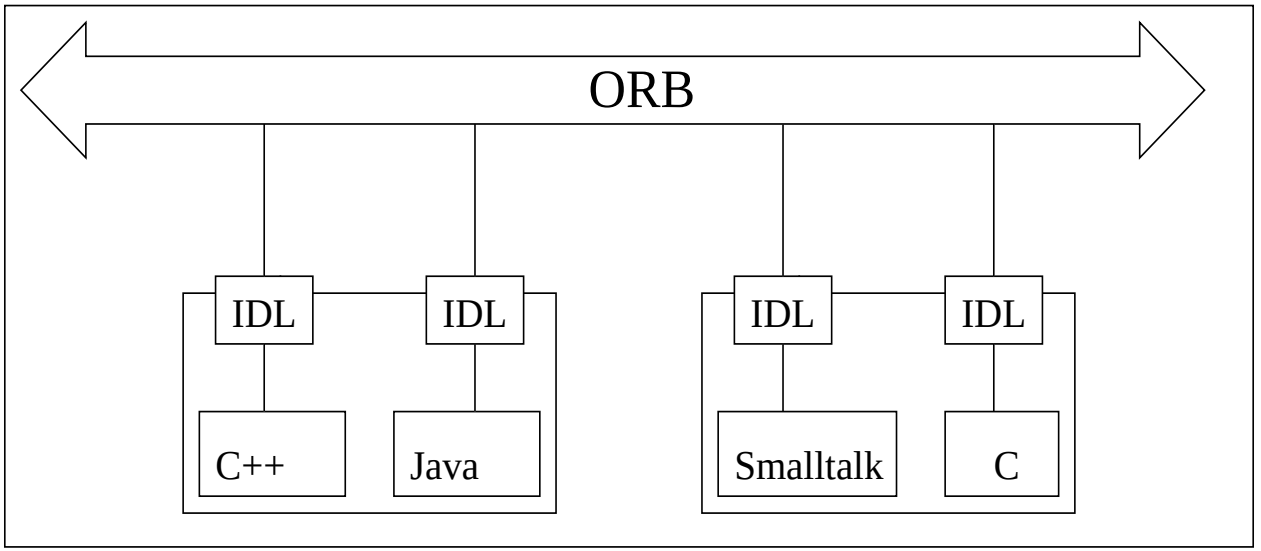
```
// fichier compteur1.idl
module M_compteur {
   interface I_compteur {
      attribute long somme;
      long increment();
   }
};
```

Remarque : Un module peut être réouvert, c'est à dire que le fi chier IDL peut s'écrire sous la forme :

```
module A { ... };
module B { ... };
module A {
  // reouverture du module pour complements
          ...
        };
```

4.4 Construction d'une interface IDL

IDL : Langage pivot



4.5 La projection IDL en C++

- Une projection est défi nie pour un langage donné

4.5 La projection IDL en C++

- Une projection est défi nie pour un langage donné
  - IDL $\rightarrow$ java ;

4.5 La projection IDL en C++

- Une projection est défi nie pour un langage donné
  - IDL $\rightarrow$ java ;
  - IDL $\rightarrow$ C++ ;

4.5 La projection IDL en C++

- Une projection est défi nie pour un langage donné
  - IDL $\rightarrow$ java ;
  - IDL $\rightarrow$ C++ ;
  - IDL $\rightarrow$ ...

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;
  - La projection des références vers les objets ;

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;
  - La projection des références vers les objets ;
  - La projection des appels de méthodes : passage des paramètres, réception des résultats

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;
  - La projection des références vers les objets ;
  - La projection des appels de méthodes : passage des paramètres, réception des résultats
  - La manipulation des exceptions ;

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;
  - La projection des références vers les objets ;
  - La projection des appels de méthodes : passage des paramètres, réception des résultats
  - La manipulation des exceptions ;
  - La projection des accès aux attributs.

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;
  - La projection des références vers les objets ;
  - La projection des appels de méthodes : passage des paramètres, réception des résultats
  - La manipulation des exceptions ;
  - La projection des accès aux attributs.
  - L'API de l'ORB ;

4.5 La projection IDL en C++

- Elle spécifi e, pour le langage cible :
  - La projection des types de base IDL ;
  - La projection des constructeurs de types : const, enum, struct, union, sequence, array, typedef ;
  - La projection des références vers les objets ;
  - La projection des appels de méthodes : passage des paramètres, réception des résultats
  - La manipulation des exceptions ;
  - La projection des accès aux attributs.
  - L'API de l'ORB ;
- Elle décrit l'utilisation des différents Services Corba pour le langage cible.

4.5 La projection IDL en C++

Projection des identifi cateurs en C++

- Les identifi cateurs sont préservés dans le code C++ généré ;
- Il faut éviter d'utiliser des mots clefs C++ en tant qu'identifi cateur ;
- Ne pas utiliser de double _.

Projection des modules en C++

- Les modules IDL sont projetés en espace de nommage (namespace) ;

Module IDL

```
module Conteneur {
  // Definitions ...
  module Contenu {
    interface Interieur {
      short f();
    };
  };
};}
```

Projection C++

```
namespace Conteneur
{
 //...
 namespace Contenu
 {
  class Interieur
  {
   public:
     virtual CORBA::Short f();
  };
 }
}
```

Projection des modules en C++

Utilisation de l'espace de nommage

```
using namespace Conteneur::Contenu;
//...
Interieur objet;


objet.f();
//...
```

Commentaire

Les espaces de nommage sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur. Leur but est essentiellement de regrouper les identificateurs logiquement et d'éviter les conflits de noms entre plusieurs parties d'un même projet. Par exemple, si deux programmeurs définissent différemment une même structure dans deux fichiers différents, un conflit entre ces deux structures aura lieu au mieux à l'édition de liens, et au pire lors de l'utilisation commune des sources de ces deux programmeurs. Ce type de conflit provient du fait que le C++ ne fournit qu'un seul espace de nommage de portée globale, dans lequel il ne doit y avoir aucun conflit de nom. Grâce aux espaces de nommage non globaux, ce type de problème peut être plus facilement évité, parce que l'on peut éviter de définir les objets globaux dans la portée globale.
Lorsque le programmeur donne un nom à un espace de nommage, celui-ci est appelé un espace de nommage nommé. La syntaxe de ce type d'espace de nommage est la suivante :

```
namespace nom
{
    declarations | definitions
}
```

`nom` est le nom de l'espace de nommage, et déclarations et définitions sont les déclarations et les définitions des identificateurs qui lui appartiennent.

Commentaire

Contrairement aux régions déclaratives classiques du langage (comme par exemple les classes), un namespace peut être découpé en plusieurs morceaux. Le premier morceaux sert de déclaration, et les suivants d'extensions. La syntaxe pour une extension d'espace de nommage est exactement la même que celle de la partie de déclaration.

```
namespace A   // Dclaration de l'espace de nommage A.
{
   int i;
}

namespace B   // Dclaration de l'espace de nommage B.
{
    int i;
}

namespace A   // Extension de l'espace de nommage A.
{
    int j;
}
```

Commentaire

Les identificateurs déclarés ou définis à l'intérieur d'un même espace de nommage ne doivent pas entrer en conflit. Ils peuvent avoir les mêmes noms, mais seulement dans le cadre de la surcharge. Un espace de nommage se comporte donc exactement comme les zones de déclaration des classes et de la portée globale.
L'accès aux identificateurs des espaces de nommage se fait par défaut grâce à l'opérateur de résolution de portée (: :), et en qualifiant le nom de l'identificateur à utiliser du nom de son espace de nommage. Cependant, cette qualification est inutile à l'intérieur de l'espace de nommage lui-même, exactement comme pour les membres des classes à l'intérieur de leur classe.

```
int i=1;    // i est global.

namespace A
{
    int i=2; // i de l'espace de nommage A.
    int j=i; // Utilise A::i.
}

int main(void)
{
    i=1;     // Utilise ::i.
    A::i=3;  // Utilise A::i.
    return 0;
}
```

Commentaire

Les fonctions membres d'un espace de nommage peuvent être définies à l'intérieur de cet espace, exactement comme les fonctions membres de classes. Elles peuvent également être définies en dehors de cet espace, si l'on utilise l'opérateur de résolution de portée. Les fonctions ainsi définies doivent apparaître après leur déclaration dans l'espace de nommage.

```
namespace A
{
    int f(void);   // Dclaration de A::f.
}

int A::f(void)     // Dfinition de A::f.
{
    return 0;
}
```

Commentaire

Il est possible de définir un espace de nommage à l'intérieur d'un autre espace de nommage. Cependant, cette déclaration doit obligatoirement avoir lieu au niveau déclaratif le plus externe de l'espace de nommage qui contient le sous-espace de nommage. On ne peut donc pas déclarer d'espaces de nommage à l'intérieur d'une fonction ou à l'intérieur d'une classe.

```
namespace Conteneur
{
    int i;              // Conteneur::i.
    namespace Contenu
    {
        int j;          // Conteneur::Contenu::j.
    }
}
```

Projection des types en C++

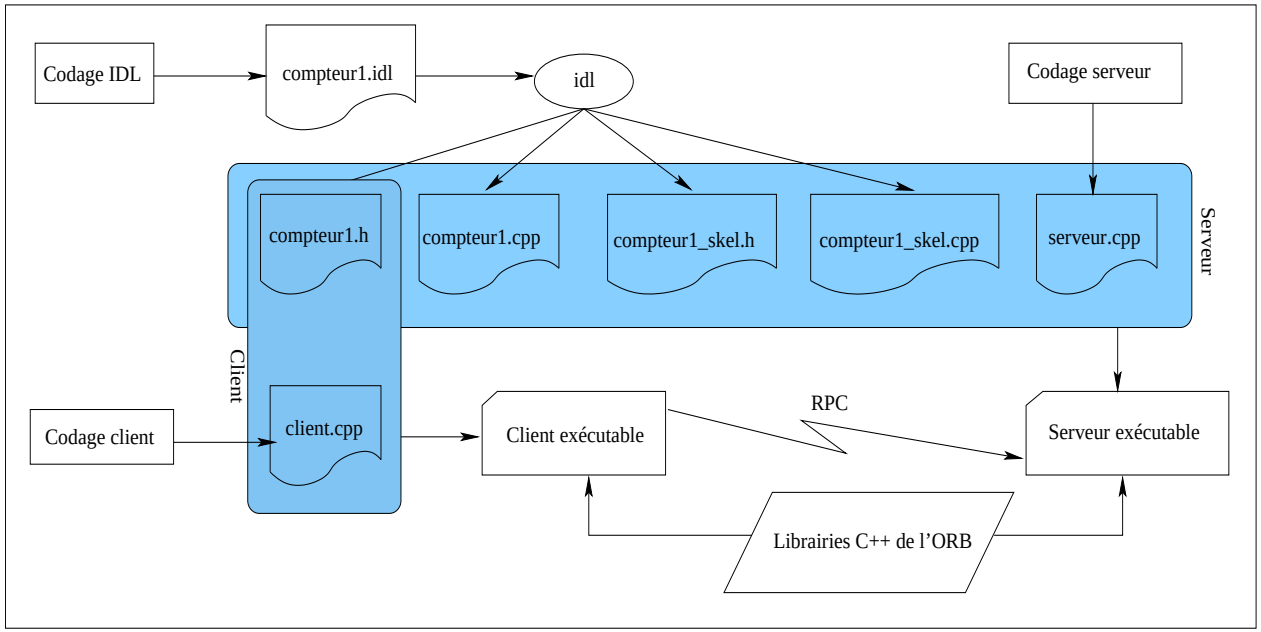
En raison des tailles des types non standards en C++, on utilisera des types prédéfi nis CORBA : :xxx.

Type IDL	Type C++	Type standard analogue
boolean	CORBA : :Boolean	bool ou unsigned char
octet	CORBA : :Octet	unsigned char
short unsigned short	CORBA : :Short CORBA : :UShort	short ou int unsigned ...
long unsigned long	CORBA : :Long CORBA : :ULong	int ou long unsigned ...

Projection des types en C++

fbat double	CORBA : :Float CORBA : :Double	fbat long
char string	CORBA : :Char char *	char char*

4.5.1 Compilation IDL pour C++



Commentaire

En fait, le serveur se décompose en général en deux parties, respectant le schéma d'utilisation précédent :

- Le servant qui correspond à l'implémentation du service lui-même ;
- Le gestionnaire du service qui s'ocupe de l'interfaçage du servant avec le bus CORBA.

4.5.2 Appel du préprocesseur idl

On utilise le préprocesseur `idl` d'ORBacus qui génère automatiquement les souches/squelettes par projection en C++.

```
module M_compteur
{ interface I_compteur
  {
    attribute long somme;
    long increment();
  };
};

idl compteur1.idl
```

Cette commande va générer les 4 fi chiers suivants :

- `compteur1.h` et `compteur1.cpp`
- `compteur1_skel.h` et `compteur1_skel.cppp`

4.5.3 Fichiers générés par le préprocesseur

- `compteur1.h` Contient les défi nitions de type C++ défi nies dans `compteur1.idl`, ainsi que les souches de classes qui correspondent aux interfaces défi nies en IDL. En-tête à inclure dans le client et le servant.
- `compteur1.cpp` Contient le code source pour les types et les classes défi nis dans `compteur1.h`.
- `compteur1_skel.h` Contient les défi nitions spécifi ques au servant. Inclus `compteur1.h`.
- `compteur1_skel.cpp` Contient les classes squelettes qui fournissent l'interface au servant.

4.5.4 Projection de l'interface

- La projection d'un module va générer un espace de nommage ou `namespace`
- La projection de l'interface générée dans `compteur1.h` correspond alors à une classe virtuelle dans cet espace de nommage.

```
namespace M_compteur
{
  class I_compteur : virtual public CORBA::Object
  {
    virtual CORBA::Long somme() = 0;
    virtual void somme(CORBA::Long) = 0;
    virtual CORBA::Long increment() = 0;
    // ....
  };
} // End of namespace M_compteur
```

Commentaire

Un grand nombre d'autres méthodes sont générées dans le fichier `compteur1.h`, notamment des constructeurs et destructeur, mais ne sont pas décrites ici.
On remarquera qu'un attribut IDL, comme `somme`, génère "implicitement" deux méthodes : une pour consulter la valeur (`virtual CORBA : :Long somme()`) et l'autre pour modifier la valeur (`virtual void somme(CORBA : :Long)`).
Toutes les classes générées par `idl` héritent de `CORBA_Object`. L'héritage est virtuel car il peut y avoir héritage multiple à partir d'une interface idl.

C'est une classe "proxy" . Un proxy agit comme un ambassadeur local pour l'objet serveur.

4.5.5 Références d'objets - Interface

A partir de cette classe virtuelle d'interface, plusieurs classes de références d'objets sont générés par le préprocesseur `idl` :

- `M_compteur::I_compteur_var` et `M_compteur::I_compteur_ptr` sont des classes de pointeurs vers des objets `M_compteur::I_compteur`.
- `xxx_ptr` est un simple pointeur vers `xxx`. Il est redéfi ni par

```
typedef xxx* xxx_ptr;
```

  On défi nit deux méthodes le manipulant : `void OBDuplicate(xxx_ptr)` et `void OBRelease(xxx_ptr)` qui sont utilisées pour la duplication et la désallocation et `void OBMarshal(xxx_ptr, OB::OutputStreamImpl*)` ainsi que `void OBUnmarshal(xxx_ptr&, OB::InputStreamImpl*)` pour le "marshalling".
- `xxx_var` est un type spécifi que de l'ORB obtenu par spécialisation au type `xxx` de la classe `OBObjVar`

```
typedef OBObjVar<xxx> xxx_var;
```

  Les duplications et désallocations sont gérées respectivement par l'opérateur d'affectation et un destructeur.

Commentaire

Notion de marshalling

Le marshalling est l'opération qui consiste à traduire tous les paramètres qu'une fonction reçoit en une représentation standard, que l'on peut communiquer sur un réseau à un autre programme, sur une autre machine et éventuellement dans un autre système d'exploitation. Cette représentation standard constitue donc une convention d'appel générique et absolument portable, qui permet aux différentes implémentations d'un ORB de parler la même langue. Inversement, les paramètres de retour de la fonction appelée, ainsi que ses codes d'erreurs et ses exceptions éventuelles, doivent être renvoyés à l'appelant. Ils sont donc également transmis via les mécanismes standards de marshalling de l'ORB.

Commentaire

Le type`xxx_var`

Pour chaque nouveau type xxx, la projection C++ crée un type `xxx_var`. Un objet de type `xxx_var` est initialisé avec un pointeur et lorsque la fin de vie est atteinte, son destructeur désalloue la mémoire. Il est parfois difficile de comprendre quand utiliser le type `xxx_var` et quand ne pas l'utiliser. Prenons un exemple : Ecrire une fonction C qui lit une chaîne de caractères à partir d'un périphérique d'entrée et retourne cette chaîne. Il y a plusieurs approches pour résoudre ce problème :

- Utilisation de la mémoire statique

```
const char *lecture_chaine()
{ static char buffer[10000];
  /* On lit la chaine que l'on stocke dans buffer ...*/
  return buffer;
}
```

Cette méthode n'est pas satisfaisante car la chaîne peut être plus grande que prévue, ou très petite. De plus chaque appel de la fonction `lecture_chaine()` écrase le résultat du précédent appel. Il faut donc en faire une copie avant un nouvel appel. Enfin cette fonction n'est pas réentrante, si plusieurs threads appellent de façon concurente `lecture_chaine()`, il risque d'y avoir une sur-écriture d'un autre résultat.

Commentaire

Pointeur statique sur un emplacement de la mémoire dynamique

```
const char *lecture_chaine()
{
 static char *resultat = NULL;
 static size_t taille_stockage = 0;
 static const size_t TAILLE_BLOC = 512;
 size_t taille_lu = 0;

 while(il_reste_des_donnees_a_lire())
   {
    /* On lit un bloc de donnes */
    if (taille_stockage - taille_lu < TAILLE_BLOC)
      {
       taille_stockage += TAILLE_BLOC;
       resultat = realloc(resultat, taille_stockage);
      }
    /* On ajoute les donnes au rsultat */
    taille_lu += TAILLE_BLOC;
   }
 return resultat;
}
```

Par rapport à l'approche précédente nous n'avons pas de problème de taille. Par contre cette fonction pose toujours des problèmes si le code est réentrant et on écrase toujours le résultat précédent.

Commentaire

Mémoire allouée par la fonction appelante

```
size_t lecture_chaine(char *resultat, size_t taille)
{
 /* Lecture d'au plus taille octets
    que l'on stocke dans resultat. */
 return nb_octets_lu;
}
```

C'est la technique utilisée par `read()` d'Unix. Le problème est la taille de la mémoire allouée par la fonction appelante.

Pointeur sur la mémoire dynamique

```
char *lecture_chaine()
{
 char *resultat = NULL;
 size_t taille_stockage = 0;
 static const size_t TAILLE_BLOC;

 while(il_reste_des_donnees_a_lire())
   {
    /* On lit un bloc de donnes */
    taille_stockage += TAILLE_BLOC;
    resultat = realloc(resultat, taille_stockage);
    /* On ajoute les donnes au rsultat */
   }
 return resultat;
}
```

Cette version est correcte et n'a aucun des inconvénients des précédentes, cependant elle laisse la charge à la fonction appelante de désallouer alors que l'allocation n'a pas eu lieu dans cette fonction.

Commentaire

```
/* ...  */
{
 char *chaine;
 chaine = lecture_chaine();
 /* ... Utilisation de chaine ... */
 free(chaine);
 /* ... */
 chaine = lecture_chaine();
 /* ... */
} /* !!! La dernire dsallocation est absente */
```

`_var` désalloue lorsque le nombre de référence est nul. Un type `_ptr` est un type qui nécessite une allocation et une désallocation explicite.

4.6 La projection IDL en Java

Projection des identifi cateurs en Java

- En général les identifi cateurs sont préservés ;
- Il faut éviter d'utiliser les mots clefs Java ;
- En cas de problème l'identifi cateur est préfi xé par _.

4.6 La projection IDL en Java

Projection des modules en Java

- Les modules IDL sont projetés en package ;

Module IDL

```
module Conteneur {
  // Definitions ...
  module Contenu {
    interface Interieur {
      short f();
    };
  };
};}
```

Projection Java

```
//Interieur.java
package Conteneur.Contenu;
public interface Interieur
  extends InterieurOperations
{ }

//InterieurOperations
package Conteneur.Contenu;
public interface
  InterieurOperations
{
 short f();
}
```

4.6 La projection IDL en Java

Projection des types en Java

Ici les tailles des types sont imposés par le langage et utilisera des types prédéfi nis Java.

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

Compilation IDL pour java

4.6.2 Appel du préprocesseur idl

On génère les souches/squelettes de la projection en Java avec la commande :

```
jidl compteur1.idl
```

Un sous-répertoire pour le package `M_compteur` est généré : il correspond au module de même nom.

4.6.1 Fichiers générés par le préprocesseur

Les fi chiers générés sont :

- `I_compteur.java` : interface Java de l'interface IDL dérivant elle-même de l'interface `I_compteurOperations` et de l'interface `CORBA.Object`
- `I_compteurHelper.java` : Classe qui regroupe des méthodes d'utilisation(liaison à un objet ...) des objets distribués ;
- `I_compteurHolder.java` : Classe "enveloppe" pour le passage de paramètres ;
- `_I_compteurStub.java` : souches de l'objet distant `I_compteur` qui sont utilisées par le client ;
- `I_compteurOperations.java` : squelette de l'objet `I_compteur` utilisé pour implémenter le service, par dérivation.
- `I_compteurPOA.java` : méthodes permettant au POA de rediriger les invocations de l'ORB vers le servant.

4.6.2 Projection de l'interface

- Comme en C++, les attributs génèrent 2 méthodes d'accès, en lecture et en modifi cation ;
- Héritage multiple traduit par des implémentations d'interfaces java

La projection des méthodes de `compteur` se trouve dans l'interface `I_compteurOperations` :

```
public interface I_compteurOperations  {
    int somme();
    void somme(int val);
    int increment();
}
```

4.6.3 Projection du passage de paramètres

- Java n'autorise que le passage de paramètre par valeur ! Donc problème pour le passage de paramètres en OUT ou INOUT...
- La solution : des classes "enveloppes" (Holder) contenant le stockage d'une valeur et un ou des constructeurs avec passage d'une valeur initiale.

La valeur pourra être modifiée dans la méthode et la modification conservée

4.6.3 Projection du passage de paramètres

class I_ compteurHolder

```
final public class I_compteurHolder
          implements org.omg.CORBA.portable.Streamable
{
    public I_compteur value;
    public I_compteurHolder() {}
    public I_compteurHolder(I_compteur initial)
    {   value = initial; }
    public void _read(
          org.omg.CORBA.portable.InputStream in)
    {   value = I_compteurHelper.read(in); }
    public void _write(
          org.omg.CORBA.portable.OutputStream out)
    {   I_compteurHelper.write(out, value); }
    public org.omg.CORBA.TypeCode _type()
    {   return I_compteurHelper.type(); }
}
```

Commentaire

```
public class Demo
{
 public void modif(short i)
 {
   i = i + 10;
 }
 public static void main(String[] args)
 {
   Demo obj = new Demo();
   short nb = 10;
   obj.modif(x);
   System.out.println(x);
 }
}
```

x n'est pas modifiée.

Commentaire

On définit une classe conteneur `ShortHolder` pour résoudre le problème.

```
public class ShortHolder
{
  public short value;
  public ShortHolder(short initial)
    {
        value = initial;
    }


}

public class Demo
{
  public void modif(ShortHolder i)
  {
    i.value = i.value + 10;
  }
  public static void main(String[] args)
  {
    Demo obj = new Demo();
    ShortHolder nb = new ShortHolder( (short) 10);
    obj.modif(nb);
    System.out.println(nb.value);
  }
}
```

Commentaire

Un exemple d'utilisation

Soit l'objet `obj` sur lequel on doit invoquer la méthode dont la description IDL est la suivante :

```
void modif(OUT short p)
```

Pour faire cette invocation en passant un paramètre pour lequel on est succeptible de récupérer une valeur de sortie, il faut construire un objet `ShortHolder` et appeler la méthode avec ce paramètre :

```
ShortHolder nb = new ShortHolder( (short) 0);
obj.modif(nb);
```

Pour récupérer la valeur de sortie du paramètre, on utilisera son champ `value`, défini dans toutes les classes de type `Holder` :

```
System.out.println(
  "Nouvelle valeur de nb : " + nb.value);
```

4.6.4 Classe `helper`

- Tout type IDL `Xx` possède une classe Java `XxHelper` ;
- Cette classe permet la conversion ;
- La méthode `narrow()` permet de convertir une référence d'objet d'un type donné en une référence d'objet d'un autre type.

```
org.omg.CORBA.Object objet_corba = orb.string_to_object(ref);
Xx objet_Xx = XxHelper.narrow(objet_corba);
```

4.7 Les connexions au Bus CORBA : l'ORB et le POA

4.7.1 CORBA : :ORB

- L'interface ORB est le point de départ pour accéder à toutes les composantes techniques du bus.
- Elle est utilisée par les applications serveurs et clientes.
- On en décrit dans la suite quelques fonctionnalités.

4.7.1 CORBA : :ORB

Initialisation

- Une application devra initialiser localement le bus CORBA de façon à pouvoir s'y connecter.
- ```
  module CORBA {
    //--> initialisation de l'ORB
    ORB ORB_init(...);
    ...
  };
  ```
- Cette opération est défi nie en dehors de toute interface : L'OMG a dérogé à ses spécifi cations afi n de pouvoir obtenir une première référence sur un objet CORBA !
  ```

4.7.1 CORBA : :ORB

Gestion des services/objets notoires

- Une fois connectée au bus CORBA, l'application doit rechercher les références des *objets notoires* qu'elle va utilisée.
- Un objet notoire est un objet CORBA primordial permettant d'accéder à d'autres objets/services disponibles sur le bus. On en décrira quelques uns : l'IFR (Référentiel d'interfaces) et les services, notamment le service de noms.
- La méthode `resolve_initial_references(...)` permet d'obtenir les références à partir d'un identifi cateur symbolique qui est souvent une chaîne de caractères.

```
module CORBA {
  interface ORB {
    //--> initialisation de l'ORB
    typedef string ObjectId;
    exception InvalidName{};
    Object resolve_initial_references(in ObjectId ident)
       raises (InvalidName);
    ...
  };
};
```

Conversion des références d'objets

- Afi n de pouvoir transmettre les références des objets instanciés sur le bus CORBA, un mécanisme les convertissant en une chaîne de caractères est mis en œuvre.
- Il permet notamment de permettre des échanges via des fi chiers, ou encore pour l'interoparabilité des bus CORBA, on parle alors d'IOR (Interoperable Object Reference).
- Cet IOR doit alors contenir, en plus de l'identifi ant de l'object sur le bus, l'adresse IP de la machine et le port TCP/IP utilisé.

```
module CORBA {
  interface ORB {
    //--> recuperation d'un IOR par reference interne
    string object_to_string(in object o);
    //--> recuperation d'une reference via un IOR
    object string_to_object(in string s);
    ...
  };
  ORB ORB_init(...);
  ...
```

4.7.2 Adaptateur d'objets (OA)

Son rôle

- Module de "connexion" du serveur sur l'ORB ;

4.7.2 Adaptateur d'objets (OA)

Son rôle

- Module de "connexion" du serveur sur l'ORB ;
- Créé ou active un object suite à une invocation ;

4.7.2 Adaptateur d'objets (OA)

Son rôle

- Module de "connexion" du serveur sur l'ORB ;
- Créé ou active un object suite à une invocation ;
- Désactive un objet ;

4.7.2 Adaptateur d'objets (OA)

Son rôle

- Module de "connexion" du serveur sur l'ORB ;
- Créé ou active un object suite à une invocation ;
- Désactive un objet ;
- Assure la réception des requêtes auprès des objets et informe l'ORB du bon acheminement + sécurisation des échanges ;

4.7.2 Adaptateur d'objets (OA)

Les différents types

- BOA (Basic OA) OA de base initialement spécifi é de manière incomplète

4.7.2 Adaptateur d'objets (OA)

Les différents types

- BOA (Basic OA) OA de base initialement spécifi é de manière incomplète
- POA (Portable OA) nouvel OA de base avec compléments de spécifi cation

4.7.2 Adaptateur d'objets (OA)

Les différents types

- BOA (Basic OA) OA de base initialement spécifi é de manière incomplète
- POA (Portable OA) nouvel OA de base avec compléments de spécifi cation
- OODA (Object Oriented Database Adapter) accès à des Bases de Données OO

4.7.2 Adaptateur d'objets (OA)

Les différents types

- BOA (Basic OA) OA de base initialement spécifi é de manière incomplète
- POA (Portable OA) nouvel OA de base avec compléments de spécifi cation
- OODA (Object Oriented Database Adapter) accès à des Bases de Données OO
- LOA (Library OA) accès à des objets stockés dans des bibliothèques logicielles.

4.7.2 Adaptateur d'objets (OA)

POA : un rapide aperçu

Le POA est constitué de :

- Un manager ...
- ... qui lance et active les programmes servants
- Intervient du coté serveur.

Au niveau du POA, on parle de :

- Servant implante une interface IDL ;
- Serveur accueille un ou plusieurs servants ;
- Référence d'objet vision de la part du client d'une interface IDL ;

Nous reprendrons plus en détail son fonctionnement plus loin.

4.8 Mise en place du gestionnaire de service

Cette construction se fait en deux étapes :

- On créé le programme servant qui implémente le service CORBA.
- On créé le programme de gestion de ce service sur l'ORB pour le rendre accessible.

4.8 Mise en place du gestionnaire de service

Implémentation du servant en C++

Pour implémenter le servant, on défi nit une classe pour l'interface `I_compteur` , appelée `C_compteur_impl` :

- elle appartient à l'espace de nommage `M_compteur` ;
- elle dérive de la classe squelette `I_compteur`, défi nie dans l'espace de nommage `POA_M_compteur`, classe défi nie dans `compteur_skel.h` qu'il faut donc inclure.

4.8.1 Programme service - header

```
//************************************
//* fichier "compteur_impl.h"
//************************************
#include "compteur_skel.h"
class C_compteur_impl :
        public POA_M_compteur::I_compteur,
        public PortableServer::RefCountServantBase
{
  private :
    CORBA::Long _somme;
  public :
    // Gestion de l'attribut par le couple
    // de methodes suivantes

    // renvoie la valeur du compteur
    CORBA::Long somme() throw(CORBA::SystemException);

    // affectation d'une valeur a somme
    void somme(CORBA::Long val) throw(CORBA::SystemException);

    // incrementation de somme
    CORBA::Long increment() throw(CORBA::SystemException);
};
```

4.8.2 Programme service - implémentation

```
//************************************
//* fichier "compteur_impl.cpp"
//************************************
#include <OB/CORBA.h>
#include "compteur_impl.h"

// valeur du compteur
CORBA::Long C_compteur_impl::somme()
        throw(CORBA::SystemException);
{ return _somme; }


// affectation d'une valeur a somme
void C_compteur_impl::somme(CORBA_Long val)
        throw(CORBA::SystemException);
{  _somme = val; }

// incrementation de somme
CORBA::Long C_compteur_impl::increment()
        throw(CORBA::SystemException);
{ return ++_somme; }
```

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).
- On contruit un pointeur sur l'objet-service de type `C_compteur_impl`.

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).
- On contruit un pointeur sur l'objet-service de type `C_compteur_impl`.
- On instancie et active un ou plusieurs servants "compteur".

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).
- On contruit un pointeur sur l'objet-service de type `C_compteur_impl`.
- On instancie et active un ou plusieurs servants "compteur".
- On convertit leurs références en chaîne de caractères afi n pour le stocker dans le fi chier-interface avec le client (`object_to_string`).

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).
- On contruit un pointeur sur l'objet-service de type `C_compteur_impl`.
- On instancie et active un ou plusieurs servants "compteur".
- On convertit leurs références en chaîne de caractères afi n pour le stocker dans le fi chier-interface avec le client (`object_to_string`).
- On construit le fi chier contenant cette chaîne récupérée par le client.

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).
- On contruit un pointeur sur l'objet-service de type `C_compteur_impl`.
- On instancie et active un ou plusieurs servants "compteur".
- On convertit leurs références en chaîne de caractères afi n pour le stocker dans le fi chier-interface avec le client (`object_to_string`).
- On construit le fi chier contenant cette chaîne récupérée par le client.
- On active le manager du POA.

4.8.3 Gestionnaire du service sur l'ORB

- On initialise l'ORB (`CORBA : :ORB_init`) et le POA, adaptateur d'objets sur cet ORB en recherchant l'objet notoire qui correspond à sa racine (`RootPOA`).
- On contruit un pointeur sur l'objet-service de type `C_compteur_impl`.
- On instancie et active un ou plusieurs servants "compteur".
- On convertit leurs références en chaîne de caractères afi n pour le stocker dans le fi chier-interface avec le client (`object_to_string`).
- On construit le fi chier contenant cette chaîne récupérée par le client.
- On active le manager du POA.
- On lance la boucle du serveur (`orb->run()`).

4.8.3 Gestionnaire du service sur l'ORB

```
//**********************************
//*  fichier "gestServeur.cpp"
//**********************************
#include <fstream>
#include <OB/CORBA.h>
#include "compteur_impl.h"

using namespace std;

int main(int argc, char** argv) {

 CORBA::ORB_var orb =
  CORBA::ORB_init(argc, argv);

 CORBA::Object_var obj =
  orb->resolve_initial_references("RootPOA");

 PortableServer::POA_var poa =
  PortableServer::POA::_narrow(obj);

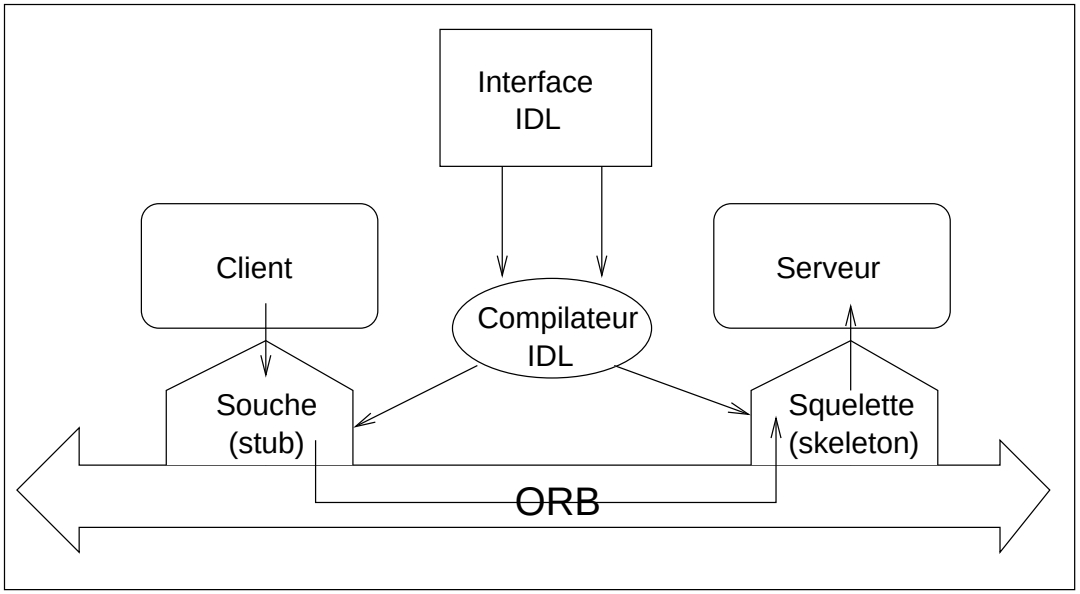
 C_compteur_impl servant_compteur;
 M_compteur::I_compteur_var compteur =
  servant_compteur._this();

 CORBA::String_var str =
  orb->object_to_string(compteur);
 const char* refFile = "compteur.ref";
 ofstream out(refFile);
 out << s << endl;
 out.close();

 PortableServer::POAManager_var manager =
  poa->the_POAManager();
 manager->activate();

 cout  << "lancement du serveur" << endl;
 orb->run();
 return 0;
}
```

4.9 Mise en place d'un client à invocation statique



- Le programme client doit faire une invocation statique du service : il faut donc que les souches aient été générées par la commande "`idl compteur.idl`" comme décrit précédemment. Le programme client doit alors n'utiliser que les objets ou types construits à partir du fi chier d'interface `idl`

4.9 Mise en place d'un client à invocation statique

Client en C++

```
//************************************
//* fichier "client.cpp"
//************************************
#include <iostream>
#include <fstream>
#include <OB/CORBA.h>
#include "compteur.h"

using namespace std;

void main(int argc, char**argv)
{
 cout <<"initialisation de l'ORB" << endl;
 try {
   CORBA::ORB_var orb =
    CORBA::ORB_init(argc, argv);
 } catch(...) {
   cerr << "Impossible d'initialiser l'ORB\n";
   return 1;
  }

 const char* refFile = ''compteur.ref'';
 ifstream in(refFile);
 char s[10000];
 in >> s;
 in.close();

 CORBA::Object_var objcompteur =
  orb->string_to_object(s);

 M_compteur::I_compteur_var compteur =
  M_compteur::I_compteur::_narrow(objcompteur);

 cout << "somme a 0" << endl;
 compteur->somme((CORBA::Long) 0);

 for(int i=0; i<100; i++)
  compteur->increment();
 cout << "resultat du traitement : "
      << compteur->somme() << endl;

 return 0;
}
```

4.10 Commandes de compilation et de lancement des applications

- Création des souches et squelettes en C++ :

```
idl compteur.idl
```

- Création du servant

```
g++ -c compteur.cpp  ...
   -I. -I/usr/local/include/OB
g++ -c compteur_skel.cpp ...
   -I. -I/usr/local/include/OB
g++ -c compteur_impl.cpp ...
   -I. -I/usr/local/include/OB
```

- Création du gestionnaire de service

```
g++ -o serveur serveur_compteur.cpp ...
   compteur.o compteur_skel.o compteur_impl.o ...
   -lOB -lJTC -lpthread -ldl ...
   -I. -I/usr/local/include/OB
```

4.10 Commandes de compilation et de lancement des applications

- Création du client

```
g++ -o client client.cpp compteur1.o ...
   -lOB -lJTC -lpthread -ldl ...
   -I. -I/usr/local/include/OB
```

- Lancement du serveur

```
serveur &
```

- Lancement du client

```
client
```

4.11 Gestion de la désactivation du service

- Dans la version précédente, le serveur reste en fonctionnement, sans moyen de l'interrompre proprement (sans "CTRL-C" ou "kill -9").
- Pour effectuer cela, on va ajouter aux services proposés par le servant, une fonction gérant sa propre désactivation.
- La nouvelle interface IDL de `compteur` est donc la suivante :

```
// fichier compteur.idl
module M_compteur {
   interface I_compteur {
      attribute long somme;
      long increment();
      void desactivation();
   }
};
```

4.11 Gestion de la désactivation du service

- Il est alors nécessaire d'ajouter l'implémentation de cette fonctionnalité dans le fi chier d'implémentation des méthodes du servant, à savoir `compteur_impl.cpp` et dans son fi chier d'en-têtes `compteur_impl.h`.
- Il est nécessaire d'écrire un constructeur qui permet de mémoriser l'adresse de l'ORB dans un attribut local. C'est sur celui-ci que se fera la désactivation lorqu'elle sera invoquée.
- Compléments de `compteur_impl.h`

```
class C_compteur_impl :
   public POA_M_compteur::I_compteur,
   public PortableServer::RefCountServantBase
{
  // attribut prive
  private :
     CORBA::ORB_var orb_;
     ...

  public :
    // en-tete du constructeur
    C_compteur_impl(CORBA::ORB_ptr);

    // en-tete de la desactivation
    void desactivation()
       throw(CORBA::SystemException);
    ...
};
```

4.11 Gestion de la désactivation du service

- Compléments de `compteur_impl.cpp`

```
C_compteur_impl::C_compteur_impl(CORBA::ORB_ptr)
   : orb_(CORBA::ORB::_duplicate(orb))
{ }

void C_compteur_impl::desactivation()
       throw(CORBA::SystemException)
{ orb_ -> shutdown(false); }
```

- Exemple d'invocation de la désactivation par le client On pourra, par exemple, ajouter les lignes de codes suivantes, après la boucle d'invocation de la méthode d'incrémentation et l'affi chage du résultat :

```
char reponse;
cout << "arret du serveur (o/n) ? : ";
cin >> reponse;
try {
   if (reponse=='o' || reponse=='O') {
      cout << "arret serveur demande" << endl;
      compteur -> desactivation();
   }
}
catch (...) {}
```

4.12.1 Désactivation de l'ORB intégré dans les versions Java 1.2 ou sup.

- Attention, en cas d'utilisation de Java 1.2 ou supérieur, il faut désactiver l'ORB intégré pour le remplacer par celui de ORBacus.
- Il s'agit en fait d'indiquer à la machine virtuelle que l'on souhaite utiliser un autre ORB, en lui redéfi nissant les 2 classes d'implémentation suivantes :
  - `org.omg.CORBA.ORBClass`
  - `org.omg.CORBA.ORBSingletonClass`

4.12.1 Désactivation de l'ORB intégré à Java

Cette manipulation peut se faire de plusieurs façons :

- On indique ce changement directement dans le code du programme. Il s'agit ici de remplacer la ligne d'initialisation de l'ORB du code suivant :

```
ORB orb = ORB.init(args, null);
```

par les lignes qui indiquent cette substitution de classe à l'ORB initialisé :

```
java.util.Properties proprietes =
   System.getProperties();
proprietes.put("org.omg.CORBA.ORBClass",
               "com.ooc.CORBA.ORB ");
proprietes.put("org.omg.CORBA.ORBSingletonClass ",
               "com.ooc.CORBA.ORBSingleton ");
System.setProperties(proprietes);
ORB orb = ORB.init(args, proprietes);
```

- On indique ce changement au lancement de la machine virtuelle. On lance `java` avec l'option `-D` :

```
java -Dorg.omg.CORBA.ORBClass=
         com.ooc.CORBA.ORB
     -Dorg.omg.CORBA.ORBSingletonClass=
         com.ooc.CORBA.ORBSingleton
```

4.12.1 Désactivation de l'ORB intégré à Java

- On construit un fi chier de confi guration. Il faut insérer ce fi chier qui devra forcément s'appeler "ORB.properties" dans votre répertoire Java Home. (cf.J.Daniel livre donné en référence)

```
public class instal_orbacus
{
  public static void main(String [] args)
  {
    System.out.println("ORB.properties doit etre dans :" +
       System.getProperty("java.home") + "/lib");
  }
}
```

Le fi chier "ORB.properties" doit contenir les deux lignes suivantes :

```
org.omg.CORBA.ORBClass=com.ooc.CORBA.ORB
org.omg.CORBA.ORBSingletonClass=
             com.ooc.CORBA.ORBSingleton
```

4.12.2 Implémentation du service en Java

```
// fichier ‘‘C_compteur_impl.java’’
//*****************************
package M_compteur;

public class C_compteur_impl extends I_compteurPOA {
  private int _somme;
  public int somme () { return _somme; }
  public void somme (int val) { _somme = val; }
  public int increment () { return ++_somme; }
  public void desactivation()
        { _orb().shutdown(false); }
}
```

4.12.3 Gestionnaire du service en Java

```
// fichier ‘‘serveur_compteur.java’’
//*****************************
package M_compteur;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.io.*;

public class serveur_compteur {
 public static void main(String args[]) {
  // desactivation de l’ORB du JDK et
  // son remplacement par celui d’Orbacus
  java.util.Properties proprietes =
    System.getProperties();
  proprietes.put("org.omg.CORBA.ORBClass",
    "com.ooc.CORBA.ORB ");
  proprietes.put(
    "org.omg.CORBA.ORBSingletonClass ",
    "com.ooc.CORBA.ORBSingleton ");
  System.setProperties(proprietes);
  ORB orb = null;

  try {
   orb=ORB.init(args, proprietes);
   POA rootPOA = POAHelper.narrow(
    orb.resolve_initial_references("RootPOA"));
   POAManager manager =
    rootPOA.the_POAManager();

   C_compteur_impl servant_compteur =
    new C_compteur_impl();
   I_compteur compteur =
    servant_compteur._this(orb);

   String str = orb.object_to_string(compteur);
   String refFile = "compteur.ref";
   PrintWriter out = new PrintWriter(new
    FileOutputStream(refFile));
   out.println(str); out.flush();
   out.close();

   manager.activate();
   orb.run();
  } catch(Exception erreur) { System.exit(1);}
 }
}
```

4.12.4 Client statique en Java

```
//************************************
//* fichier "client_compteur.java"
//************************************
package M_compteur;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.io.*;

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

  // desactivation de l'ORB du JDK et
  // son remplacement par celui d'Orbacus
  java.util.Properties proprietes =
   System.getProperties();
  proprietes.put("org.omg.CORBA.ORBClass",
   "com.ooc.CORBA.ORB ");
  proprietes.put(
   "org.omg.CORBA.ORBSingletonClass ",
   "com.ooc.CORBA.ORBSingleton ");
  System.setProperties(proprietes);
  ORB orb = null;

  try {
   orb=ORB.init(args, proprietes);
   String refFile = "compteur.ref";
   BufferedReader in =
    new BufferedReader(
            new FileReader(refFile));
   String ref = in.readLine();
   in.close();

   org.omg.CORBA.Object objcompteur =
    orb.string_to_object(ref);
   I_compteur compteur =
    I_compteurHelper.narrow(objcompteur);

   System.out.println("somme a 0");
   compteur.somme(0);

   for (i=0; i<100; i++) compteur.increment();
   System.out.println(
                "resultat du traitement :"+
                compteur.somme());
   System.out.println("arret serveur (o/n)?:");
   in = new BufferedReader (
             new InputStreamReader(System.in));
   rep = in.readLine();
   if (rep.equals("o") || rep.equals("O")) {
     System.out.println("arret serveur");
     compteur.desactivation();
   }
  } catch (Exception e) { System.exit(1);}
  System.exit(0);
 }
}
```

4.12.5 Compilation et lancement des programmes

1. Création des souches et squelettes en C++ :

```
jidl compteur.idl
```

2. Création du servant, du gestionnaire de service et du client

```
javac M_compteur/*.java
```

3. Lancement du serveur

```
java M_compteur.serveur_compteur &
```

4. Lancement du client

```
java M_compteur.client_compteur
```

Commentaire

- ORBacus ne fonctionne pas correctement avec le JDK 1.4 tant que le boot classpass n'est pas positionné correctement. Il y a un bug au niveau du POA de l'ORB du JDK. La compilation pourra se faire par exemple par :

```
javac -bootclasspath  /usr/local/OB-4.1.2/JOB-4.1.2/lib:\
/usr/local/OB-4.1.2/lib:$JDK_HOME/jre/lib/rt.jar code.java
```

- Le boot classpath doit être défini quand on exécute une application ORBacus avec le JDK 1.4

```
java -Xbootclasspath/p:$CLASSPATH code
```

- On se reportera au TP d'installation d'ORBacus.