Implémentation CORBA : Orbix

Janvier 1998

Virginie Amar
Centre Scientifique et Technique du Bâtiment (CSTB)
amar@cstb.fr

Orbix

Sommaire

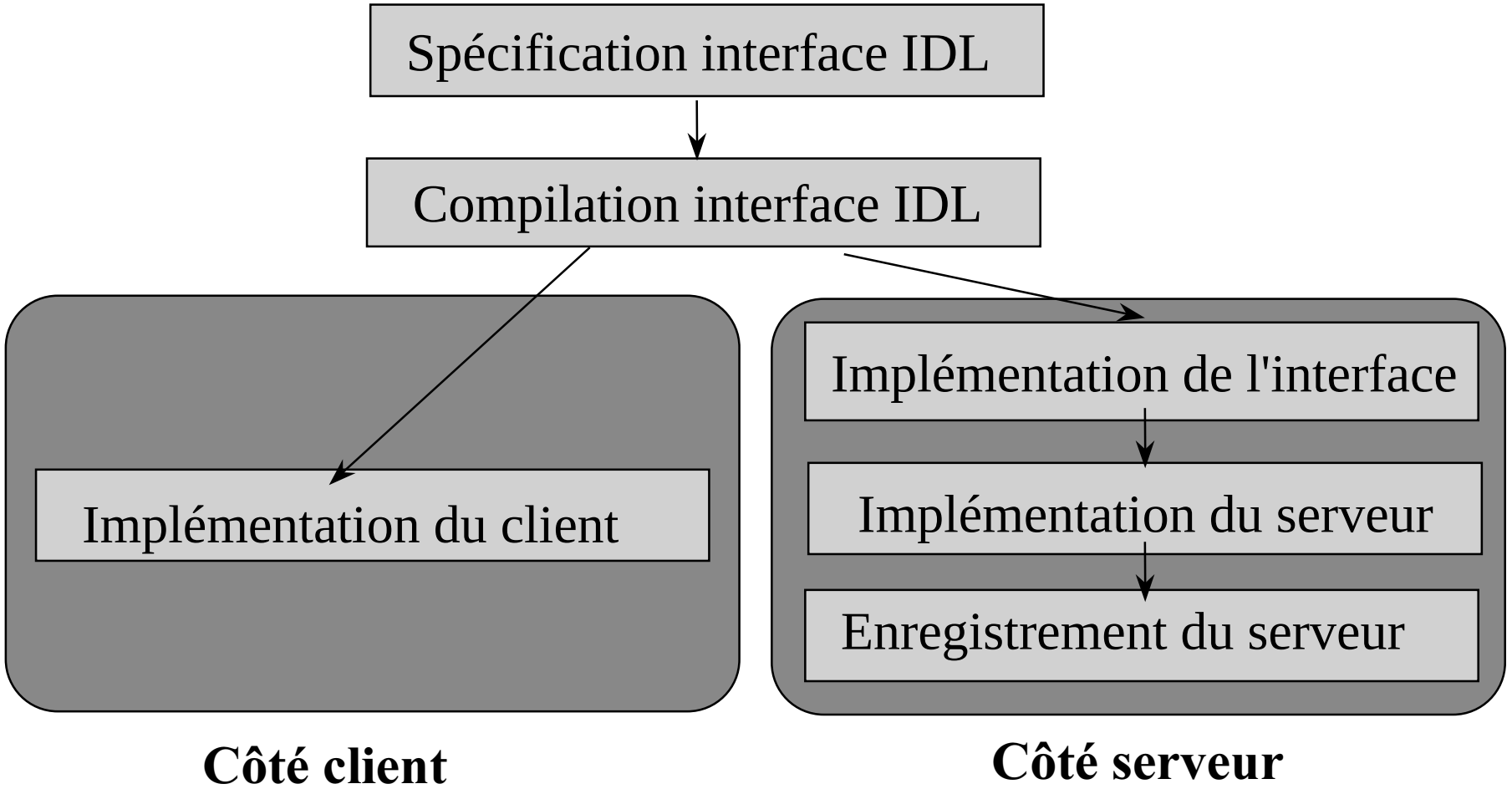
Introduction

Orbix de IONA Technologies Ltd (Irlande)

- ☞ conforme à la spécification CORBA 2.0;
- ☞ génération de code C++ pour les spécifications IDL; (binding C++)
- ☞ leader du marché;
- ☞ fonctionnalités supplémentaires par rapport à CORBA 2.0.

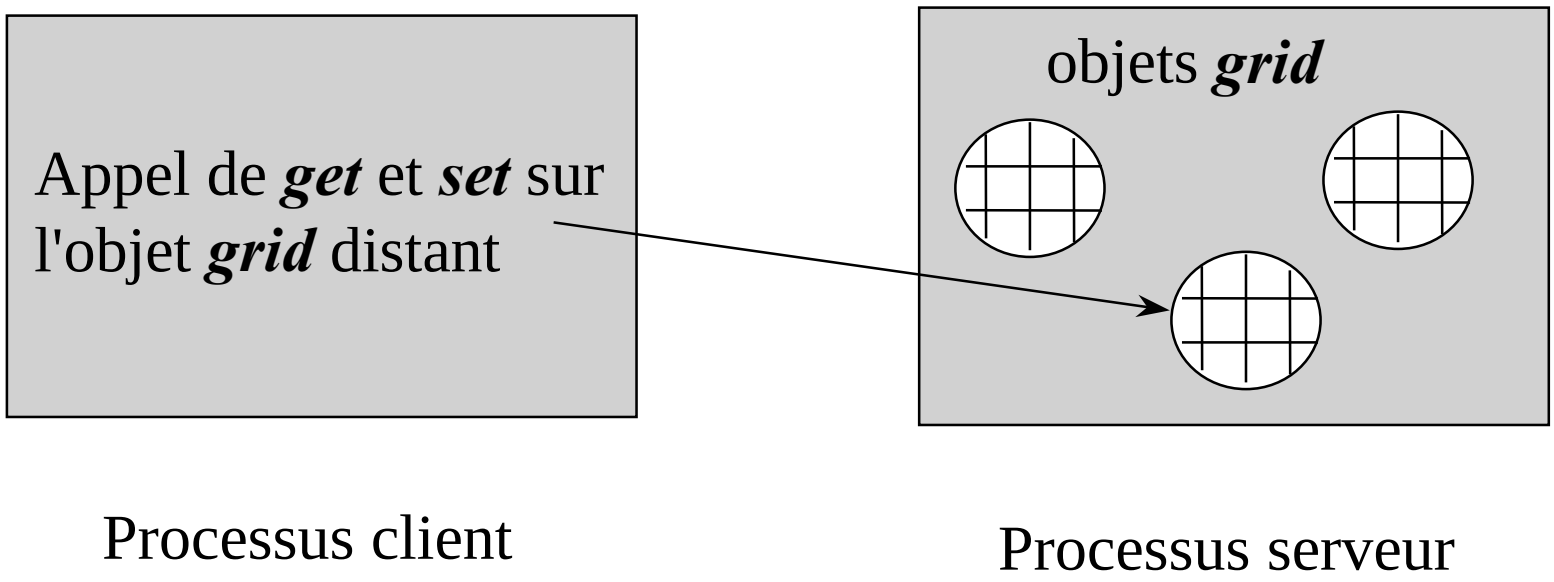
1- Exemple introductif

Les différentes phases de développement d'une application distribuée



Spécification interface IDL (1/2)

Un objet *grid* est un tableau contenant des valeurs.
Les valeurs sont accéssibles grâce aux opérateurs ***get*** et ***set*** qui peuvent être invoqués à distance.

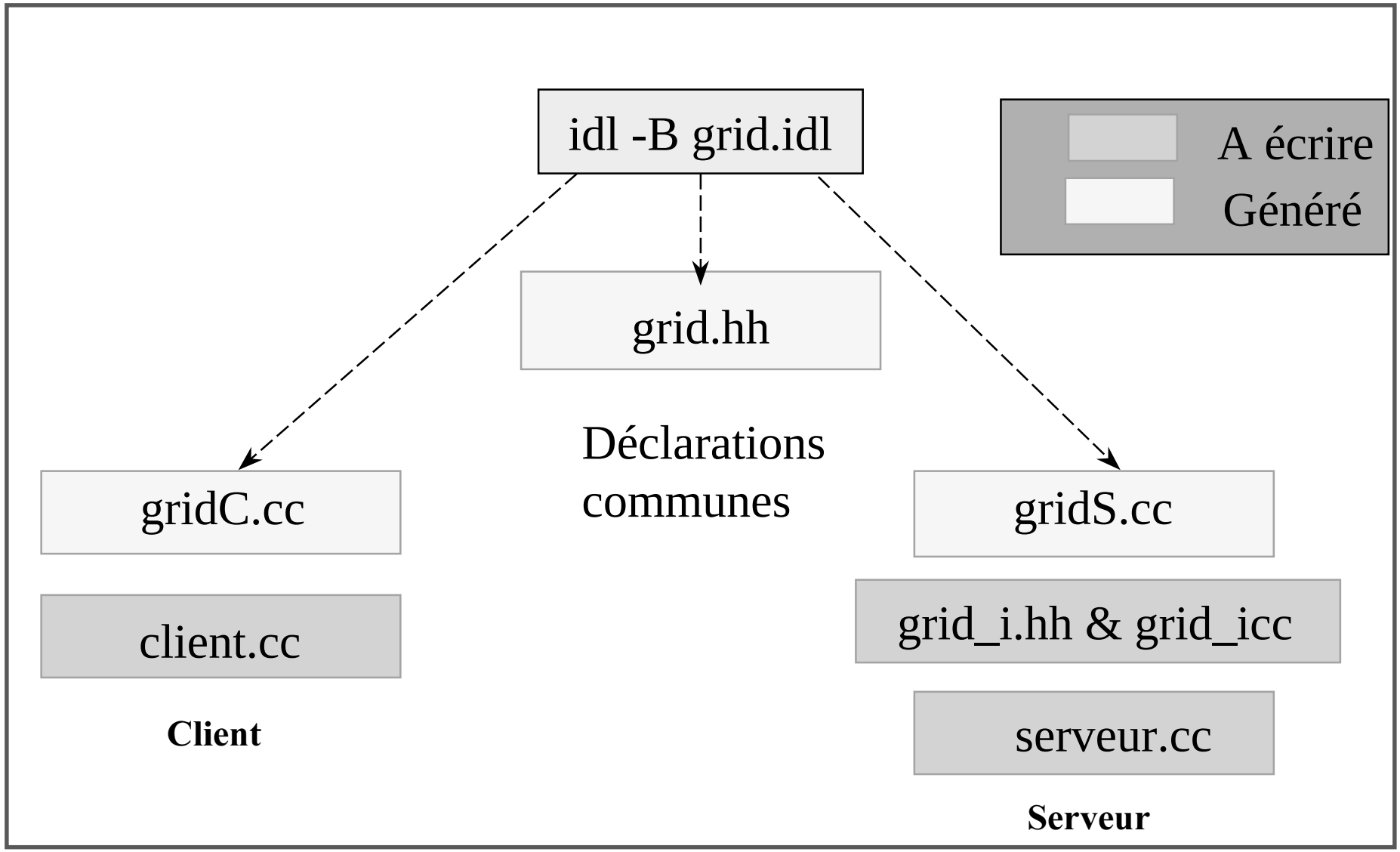


CSTB le futur en construction

Spécification interface IDL (2/2)

```
interface grid {
  readonly attribute short height;
  readonly attribute short width;
  void set (in short n, in short m, in long value);
  long get(in short n, in short m);
};
```

Compilation interface IDL



CSTB le futur en construction

Fichier des déclarations communes

```
// Extrait fichier grid.hh
class grid : public virtual CORBA::Object {
  public :
    static grid_ptr _bind(...);
    virtual CORBA::Short height() throw (CORBA::SystemException);
    virtual CORBA::Short width() throw (CORBA::SystemException);
    virtual void set (CORBA::Short n, CORBA::Short m,
                      CORBA::Long value
                      throw (CORBA::SystemException);
    virtual CORBA::Long get (CORBA::Short n, CORBA::Short m)
                      throw (CORBA::SystemException);
};
```

Implémentation de l'interface (1/7)

Approche BOA (Basic Object Adapter) :
Ecrire une classe C++ qui hérite de la classe ***gridBOAImpl*** afin d'hériter des comportements permettant de distribuer l'objet ***grid***.

Fichier grid_i.hh

```
#include "grid.hh"
class grid_i : public virtual gridBOAImpl {
   CORBA::Short m_height;
   CORBA::Short m_width;
   CORBA::Long **m_a;
 public :
   //constructor
   grid_i(CORBA::Short h, CORBA::Short w);
   //destructor
   virtual ~grid_i();
```

Implémentation de l'interface (2/7)

Fichier grid_i.hh (suite)

```
virtual CORBA::Short width (
        CORBA::Environment &env=CORBA::default_environment);
virtual CORBA::Short height (
        CORBA::Environment &env=CORBA::default_environment);
virtual void set (CORBA::Short n, CORBA::Short m,
        CORBA::Long value,
        CORBA::Environment &env=CORBA::default_environment);
virtual CORBA::Long get (CORBA::Short n, CORBA::Short m,
        CORBA::Environment &env=CORBA::default_environment);
};
```

Implémentation de l'interface (3/7)

Fichier grid_i.cc

```
#include "grid_i.hh"
grid_i::grid_i(CORBA::Short h, CORBA::Short w) {
  m_height =h;
  m_width = w;
  m_a = new CORBA::Long* [h];
  for (in i = 0; i<h; i++)
    m_a[i] = new CORBA::Long[w];
}
grid_i::~grid_i() {
  for(int i = 0; i< m_height; i++)
          delete[] m_a[i];
  delete[] m_a;
}
```

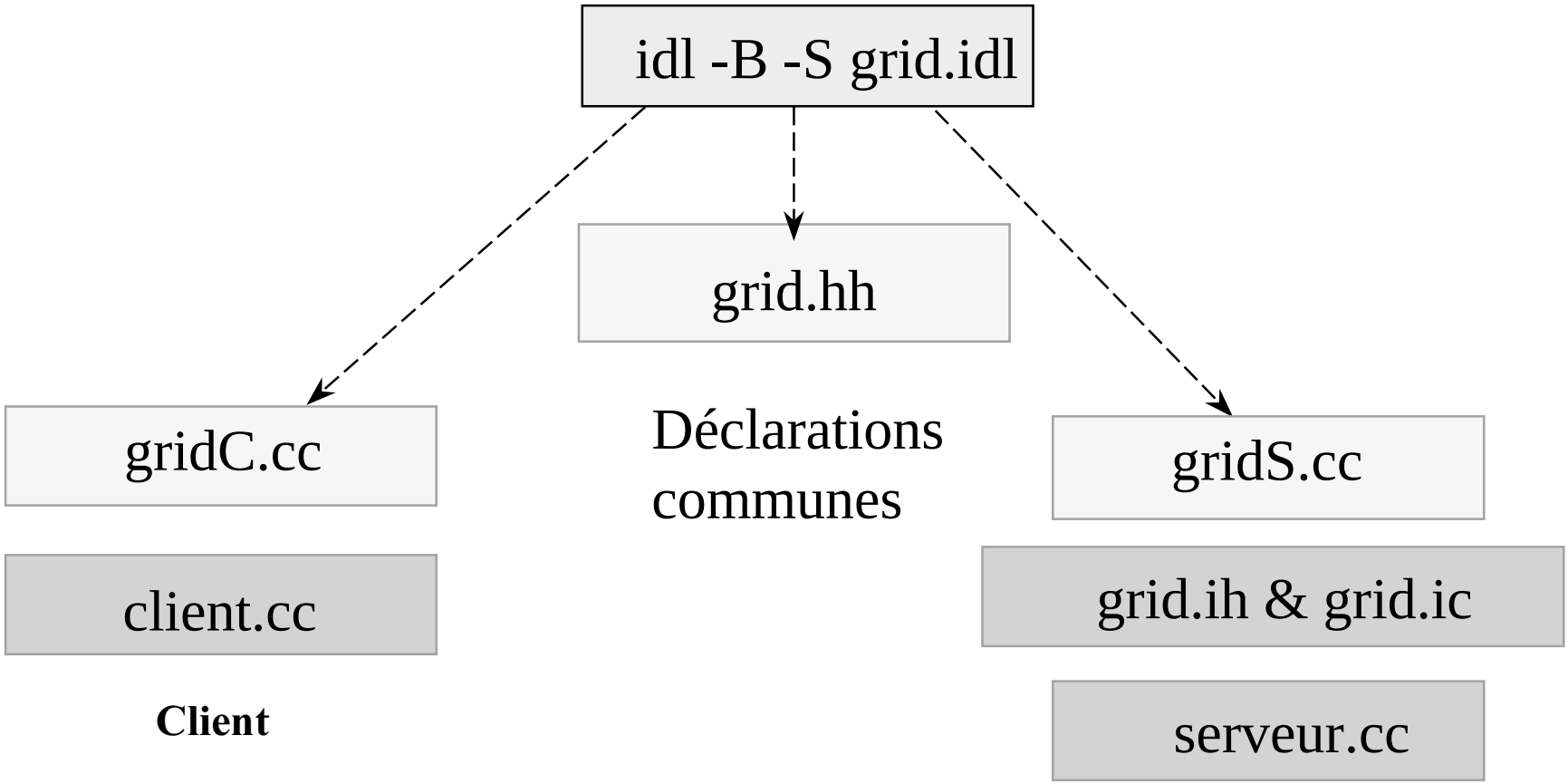
Implémentation de l'interface (4/7)

Fichier grid_i.cc (suite)

```
CORBA::Short grid_i::width(CORBA::Environment&) {
  return m_width;
}
CORBA::Short grid_i::height(CORBA::Environment&) {
  return m_height;
}
void grid_i::set(CORBA::Short n, CORBA::short m,
          CORBA::Long value, CORBA::Environment&){
  m_a[n][m] = value;
}
CORBA::Long grid_i::get(CORBA::Short n,
          CORBA::short m,CORBA::Environment&){
  return m_a[n][m];
}
```

Implémentation de l'interface (5/7)

Possibilité d'utiliser l'option -S pour implémenter plus rapidement les interfaces :



Implémentation de l'interface (6/7)

Fichier grid.ih (suite)

```
#include "grid.hh"

class grid== {
public:
    virtual CORBA::Short height (CORBA::Environment &);
    virtual CORBA::Short width (CORBA::Environment &);
    virtual void set (CORBA::Short n, CORBA::Short m,
                      CORBA::Long value,
                      CORBA::Environment &) ;
       virtual CORBA::Long get (CORBA::Short n,
                      CORBA::Short m,
                      CORBA::Environment &) ;
};
```

Implémentation de l'interface (7/7)

Fichier grid.ic (suite)

```
#include "grid.ih"
CORBA::Short grid==::  height (
                        CORBA::Environment &IT_env)  {}

CORBA::Short grid==::  width (
                        CORBA::Environment &IT_env)  {}

void grid==:: set (CORBA::Short n, CORBA::Short m,
                        CORBA::Long value,
                        CORBA::Environment &IT_env) {}

CORBA::Long grid==:: get (CORBA::Short n,
                        CORBA::Short m,
                        CORBA::Environment &IT_env) {}
```

Implémentation du serveur (1/2)

Fichier serveur.cc

```
#include "grid_i.hh"
#include <stream.h>

main() {
  grid_i myGrid(100,100);

  try {
    CORBA::Orbix.impl_is_ready("GridSrv");
  } catch (...) {...}
  cout << "server terminating" << endl;
}
```

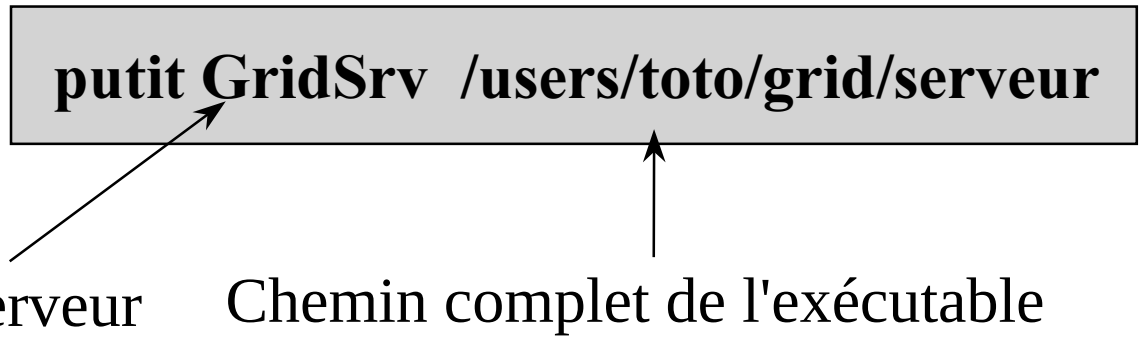
Implémentation du serveur (2/2)

Possibilité d'utiliser les *markers* afin de nommer explicitement les objets instanciés sur le serveur :

Fichier serveur.cc

```
#include "grid_i.hh"
main() {
  grid_i myGrid1(100,100);
  myGrid._marker(« grid1»);
  grid_i myGrid2(200,200);
  myGrid2._marker(« grid2»);
  try {
    CORBA::Orbix.impl_is_ready("GridSrv");
  } catch (...) {...}
  cout << "server terminating" << endl;
}
```

Enregistrement du serveur



Si le serveur n'est pas actif quand un client émet une requête,
le démon Orbix l'activera.
Le serveur doit être enregistré dans le référentiel des implémentations (*Implementation Repository*).

Manipulation du référentiel des implémentations

Commandes inspirées d'Unix pour la manipulation du référentiel des implémentations :

putit : ajoute un serveur dans le Réf des Impl
rmit : retire un serveur du Réf des Impl
lsit : liste l'ensemble des serveurs disponibles
catit : affiche les caractéristiques d'un serveur
chmodit : permet de définir des droits d'accès sur un serveur
chownit : permet de modifier des droits d 'accès sur un serveur
killit : détruit un processus serveur
mkdirit : crée un répertoire dans le Réf des Impl
rmdirit : efface un répertoire dans un Réf des Impl
pingit : permet de vérifier si le démon Orbix tourne
psit : liste l'ensemble des processus actifs connus par le démon Orbix

Implémentation du client (1/4)

Fichier client.cc

```
main(int  argc, char** argv) {
  grid_var p;

  try {
    p = grid::_bind(":gridSrv", argv[0]);
  }
  catch (const CORBA::SystemException& se) {
    cerr << &se << endl;
  }
  catch(...) {
    cerr << "bind to object failed" << endl;
    cerr << "Unexpected exception"
    exit(-1)
};
```

CSTB le futur en construction

Implémentation du client (2/4)

Fichier client.cc (suite)

```
try {
    cout << "height is " << p->height();
    cout <<"width is " << p->width();
}
catch (const CORBA::SystemException& se) {
    cerr << &se << endl;
}
catch(...) {
    cerr << "call to height "
            << "or width failed : " << endl;
    cerr << "Unexpected exception " << end;
    exit(-1);
}
```

Implémentation du client (3/4)

Fichier client.cc (fin)

```
try {
    p->set(2,4,123);
    cout << "grid[2,4] is " << p->get(2,4);
}
catch (const CORBA::SystemException& se) {
     cerr << &se << endl;
}
catch(...) {
  cerr << "call to set or get failed: " << endl;
  cerr << "Unexpected exception " << endl;
  exit(-1);
 }
} //end of the main
```

Implémentation du client (4/4)

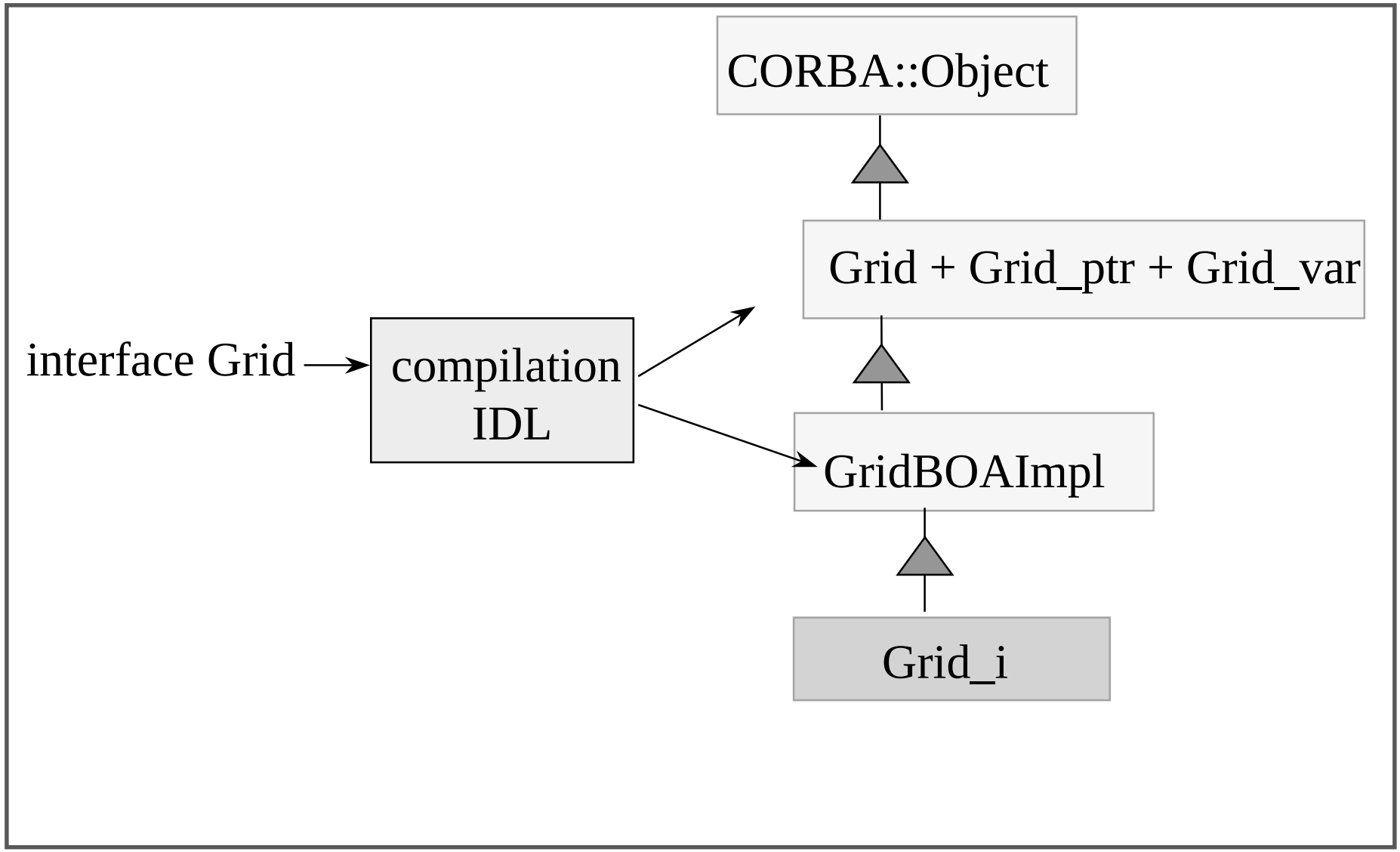
La méthode statique *bind* permet de récupérer un *proxy* sur un objet distant :

Syntaxe : bind(nom du marker:nom du serveur, host)
Exemple : grid::bind(«obj1:gridSrv»,ss20-2)

Le *marker* peut être omis ainsi que le nom du serveur.
Dans ce cas l'objet implémentation recherché porte le nom de l'interface.

Exemple : grid::bind(« »,ss20-2)

Hiérarchie des objets



CSTB le futur en construction

Hiérarchie des objets

```
typedef grid* grid_ptr;

class grid : public virtual CORBA::Object {
  ...
 public :
  virtual void set (CORBA::Short n ,
                    CORBA::Short m,
                    CORBA::Long value, ...) {
       ...
       CORBA::Request  IT_r (this, «set», ...);
        IT_r  << n;   IT_r  << m;   IT_r << value;
        IT_r.invoke(...);
};

class grid_var : public CORBA::_var {
 public:
     ...
   grid_var &operator= (grid* IT_p) {...}
   grid* operator->() { return _ptr; }
 protected :
   grid* _ptr;
};
```

```
class gridBOAImpl : public virtual grid {
  ...
 public :
  virtual void set (CORBA::Short n ,
                    CORBA::Short m,
                    CORBA::Long value, ...) = 0;
};

class grid_i : public virtual gridBOAImpl {
 ...
 public :
  virtual void set (CORBA::Short n ,
                    CORBA::Short m,
                    CORBA::Long value, ...);
};
```

```
grid_var gv,
grid_ptr gp;

gp = grid::_bind(...);
gv = grid::_bind(...);

gp = gv.operator->();
gv->set(2,4,123);
```

CSTB
le futur en construction

2- Gestion mémoire

Gestion spécifique des références sur les objets pour éviter les fuites mémoire (memory leak) du côté client et du côté serveur.

Règles d'usage dépendantes du types de passage des paramètres (in, inout, out), des types des paramètres (type de base, type à longueur fixe ou à longueur varibale) et de la valeur de retour (return).

Gestion mémoire : types de base

Gestion simple

```
interface T {
  float op1(in long l, inout short s, out char c);
};
```

```
T_var p = ....;
CORBA::Char c; CORBA::Float f;
CORBA::Short s = 10 ; CORBA::Long l = 1000;
f= p->op1(l, s, c);
cout << l;     // 1000
cout << s;    //  3
cout << c;    // 'x'
cout << f;    //  2.2
```

Client

```
CORBA::Float T_i::op1(CORBA::Long l,
                      CORBA::Short s,
                      CORBA::Char &c, ...) {
  cout << l;     // 1000
  cout << s;    // 10
  s = 3;
  c = 'x'
  return 2.2;
};
```

Objet implémentation

Gestion mémoire : pointeurs

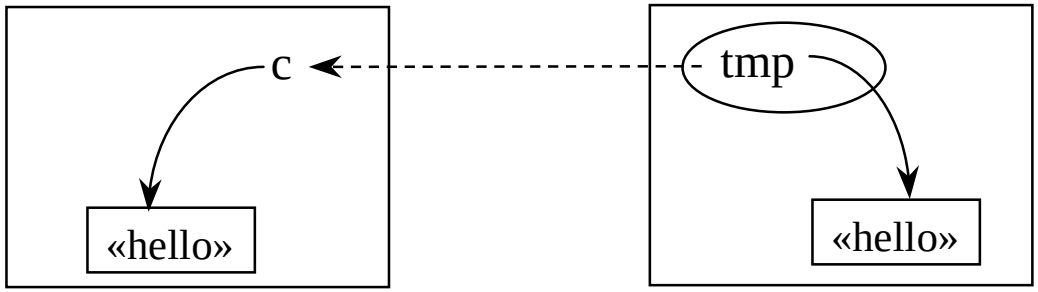
Gestion difficile car les clients et les serveurs sont dans des processus différents.

Problèmes : Qui fait l'allocation de la mémoire ? (malloc, new)

Qui fait la désallocation de la mémoire ? (delete)

```
interface T {
  string op1()
};
```

```
char* T_i::op1() {
  char* tmp = CORBA::string_alloc(6);
  strcpy(tmp, «hello»);
  return tmp;
};
```



Pointeurs mode IN

Règle pour le type de passage de paramètre **IN**

Côté client

- allocation mémoire
- initialisation des valeurs
- désallocation mémoire

Côté serveur

- accés aux valeurs
- pas d'allocation à faire
- pas de désallocation à faire
- Orbix désalloue automatiquement les pointeurs => Attention !!!!!
  Pour garder copie des valeurs des paramètres il faut en faire une copie explicitement.

Pointeurs mode IN : exemple

Trouver les erreurs

```
//IDL
typedef long vector[2];
interface foo {
   void op1(in string s, in vector v);
};
```

```
// Code Client
{
  foo_var fVar = ...;
  char *s1;
  vector_slice *vs = vector_alloc();
  vs[0] = 12; vs[1] = 13;
  fVar->op1(s1, vs);
  CORBA::string_free(s1);
}
```

```
// Code objet implémentation
class foo_i : public fooBOAImpl {
  char *m_string;
 ...
};
void foo_i::op1(const char *s, const vector v, ...)
{
  m_string = s;
  cout << vector[0] << vector[1] << endl;
  vector_free(v);
}
```

Pointeurs mode INOUT

Règle pour le type de passage de paramètre **INOUT**

Côté client

- allocation mémoire
- initialisation des valeurs (pointeur != null)
- désallocation mémoire
- si le client veut garder les valeurs initiales des objets alors il faut faire une copie explicitement.

Côté serveur

- accés aux valeurs et modification
- si espace mémoire > à celui du paramètre alors il faut désallouer le pointeur et allouer un nouvel espace mémoire ou faire une affectation en utilisant un type _var
- Orbix désalloue automatiquement les pointeurs => Attention !!!!!
  Pour garder copie des valeurs des paramètres il faut en faire une copie explicitement.

Pointeurs mode INOUT : exemple 1

Exemple 1 : Trouver les erreurs

```
//IDL
interface foo {
   void op1(inout string s);
};
```

```
// Code Client
{
  foo_var fVar = ...;
  char *s1 = CORBA::string_alloc(6);
  strcpy(s1, «salut»);
  char *s2 = s1;
  fVar->op1(s1);
  cout << s2 << endl;
}
```

```
// Code objet implémentation
class foo_i :: public fooBOAImpl {
  char *m_string;
 ...};
void foo_i::op1(char*& s, ...)
{
  m_string = CORBA::string_alloc(strlen(s)+1);
  strcpy(m_string, s);
  CORBA::string_free(s);
  s = CORBA::string_alloc(12);
  strcpy(s, «nouvelle st»);
}
```

Pointeurs mode INOUT : exemple 2

Exemple 2 : Trouver les erreurs

```
//IDL
interface account;
interface foo {
   void op1(inout account b);
};
```

```
// Code Client
{
  foo_var fVar = ...;
  account_var bVar = ...;
  fVar->op1(bVar);
}
```

```
// Code objet implémentation
void foo_i::op1(account_ptr &b, ...)
{
  b = new account_i();
}
```

Pointeurs mode OUT et return

Règle pour le type de passage de paramètre **OUT** et **return**

Côté client

- déclaration du pointeur = NULL
- pas d'allocation mémoire
- désallocation mémoire
- si le client veut modifier les valeurs il faut qu'il fasse une copie

Côté serveur

- allocation
- initialisation
- Orbix désalloue automatiquement les pointeurs => Attention !!!!!
  Pour garder copie des valeurs des paramètres il faut en faire une copie explicitement.

OUT et Return

Pointeurs mode OUT : exemple

Exemple 1 : Trouver les erreurs

```
//IDL
interface foo {
   void op1(out string s);
};
```

```
// Code Client
{
  foo_var fVar = ...;
  char *s = CORBA::string_alloc(10);
  fVar->op1(s);
  cout << «Nouvelle valeur de s = » << s;
}
```

```
// Code objet implémentation
class foo_i :: public fooBOAImpl {
  char *m_string;
 ...};

void foo_i::op1(char* &s, ...)
{
  char* s = CORBA::string_alloc(6);
  strcpy(s, «Salut»);
  m_string = s;
}
```

Pointeurs mode return : exemple

Exemple 2 : Trouver les erreurs

```
//IDL
interface account;
interface bank{
    account op1();
};
```

```
// Code Client
{
  bank_var bVar = ...;
  account_var aVar;
  aVar = bVar->op1();
}
```

```
// Code objet implémentation
class bank_i :: public bankBOAImpl {
  account_var  tab_account[10];
 ...};

account_ptr bank_i::op1(...)
{
  ...
  CORBA::ULong i = 0;
  acount_ptr  p = new account_i();
  tab_account[i++] = p;
  return p;
}
```

Pointeurs mode IN : solution

Exemple : Solutions

```
//IDL
typedef long vector[2];
interface foo {
   void op1(in string s, in vector v);
};
```

```
// Code Client
{
 foo_var fVar = ...;
 char *s1 = CORBA::string_alloc(6);
 strcpy(s1, «salut»);
 vector_slice *vs = vector_alloc();
 vs[0] = 12; vs[1] = 13;
 fVar->op1(s1, vs);
 CORBA::string_free(s1);
 vector_free(vs);
}
```

```
// Code objet implémentation
class foo_i :: public fooBOAImpl {
  char *m_string;
 ...};
void foo_i::op1(const char *s, const vector v, ...)
{
  m_string = CORBA::string_alloc(strlen(s)+1);
  strcpy(m_string, s);
  cout << vector[0] << vector[1] << endl;
  vector_free(v);  // KO Orbix désalloue v
}
```

Pointeurs mode INOUT : solution

Exemple1 : Solutions

```
//IDL
interface foo {
   void op1(inout string s);
};
```

```
// Code Client
{
 foo_var fVar = ...;
 char *s1 = CORBA::string_alloc(6);
 strcpy(s1, «salut»);
 char *s2 =
     CORBA::string_alloc(strlen(s1));
 strcpy(s2, s1);
 fVar->op1(s1);
 cout << s2 << endl;
 CORBA::string_free(s1);
 CORBA::string_free(s2);
}
```

```
// Code objet implémentation
class foo_i :: public fooBOAImpl {
  char *m_string;
 ...};
void foo_i::op1(char*& s, ...)
{
  m_string = CORBA::string_dup(s);
  CORBA::string_free(s);
  s = CORBA::string_alloc(12);
  strcpy(s, «nouvelle st»);
}
```

Pointeurs mode INOUT : solution (1/2)

Exemple 2 : Solutions

```
//IDL
interface account;
interface foo {
   void op1(inout account b);
};
```

```
// Code Client
{
  foo_var fVar = ...;
  account_var bVar = ...;
  fVar->op1(bVar);
}
```

```
// Code objet implémentation
void foo_i::op1(account_ptr &b, ...)
{
  CORBA::release(b);
  b = new account_i();
}
```

Pointeurs mode INOUT : solution (2/2)

```
typedef account* account_ptr;
class account : public virtual CORBA::Object {
   ...
};

class account_var : public CORBA::_var {
  public:
     ...
    account_var() { _ptr = account_nil(); }
    account_var (account* IT_p) { _ptr = IT_p; }
    account_var &operator= (account* IT_p) {
        account_release(_ptr);
        _ptr = IT_p;
       return (*this);
    }
    account* operator->() { return _ptr; }
  protected :
    account* _ptr;
};

class accountBOAImpl : public virtual account {
  ...
};

class account_i : public virtual accountBOAImpl {
  ...
};
```

```
// Code objet implémentation
void foo_i::op1(account_ptr &b, ...)
{
  CORBA::release(b);
  b = new account_i();}
```

Autre possibilité :

```
// Code objet implémentation
void foo_i::op1(account_ptr &b, ...)
{
  account_var btmp = b;
  b = new account_i();
}
```

Pointeurs mode OUT : solution

Exemple 1 : Solutions

```
//IDL
interface foo {
   void op1(out string s);
};
```

```
// Code Client
{
 foo_var fVar = ...;
 char* s;
 //  char *s = CORBA::string_alloc(10);
 // allocation perdue (memory leak)
 fVar->op1(s);
 cout << «Nouvelle valeur de s = » << s;
 CORBA::string_free(s);
}
```

```
// Code objet implémentation
class foo_i :: public fooBOAImpl {
 char *m_string;
 ...};

void foo_i::op1(char* &s, ...)
{
 char* s = CORBA::string_alloc(6);
 strcpy(s, «Salut»);
 m_string = CORBA::string_dup(s);
}
```

Pointeurs mode return : solution (1/1)

Exemple 2 : Solutions

```
//IDL
interface account;
interface bank{
    account op1();
};
```

```
// Code Client
{
  bank_var bVar = ...;
  account_var aVar;
  aVar = bVar->op1();
}
```

```
// Code objet implémentation
class bank_i :: public bankBOAImpl {
  account_var  tab_account[10];
 ...};

account_ptr bank_i::op1(...)
{
  CORBA::ULong i = 0;
  ...
  acount_ptr  p = new account_i();
  account::_duplicate (p);
  tab_account[i++] = p;
  return p;
}
```

Pointeurs mode return : solution (2/2)

```
typedef account* account_ptr;
class account : public virtual CORBA::Object {
   ...
};

class account_var : public CORBA::_var {
  public:
     ...
    account_var() { _ptr = account_nil(); }
    account_var (account* IT_p) { _ptr = IT_p; }
    account_var &operator= (account* IT_p) {
        account_release(_ptr);
        _ptr = IT_p;
       return (*this);
    }
    account* operator->() { return _ptr; }
  protected :
    account* _ptr;
};

class accountBOAImpl : public virtual account {
  ...
};

class account_i : public virtual accountBOAImpl {
  ...
};
```

```
// Code objet implémentation
class bank_i :: public bankBOAImpl {
  account_var  tab_account[10];
 ...};

account_ptr bank_i::op1(...)
{
  CORBA::ULong i = 0;
   ...
  acount_ptr  p = new account_i();
  account::_duplicate (p);
  tab_account[i++] = p;
  return p;
}
```

CSTB
le futur en construction

Interface bank

```
interface account {
  exception amount_too_low ();
  void AddMoney (in float amout);
  void GetMoney (in float amount)
              raises (amout_too_low);
  readonly attribute float balance;
  readonly attribute long id;
};
```

```
interface bank {
  exception number_account_exceeded {};
  exception wrong_account_id {};
  exception account_already_exists {};
  account CreateAccount (in long id)
              raises (number_account_exceeded,
                      wrong_account_id );
  void CloseAccount (in long id)
              raises (wrong_account_id);
  account OpenAccount (in long id)
              raises (wrong_account_id);
};
```

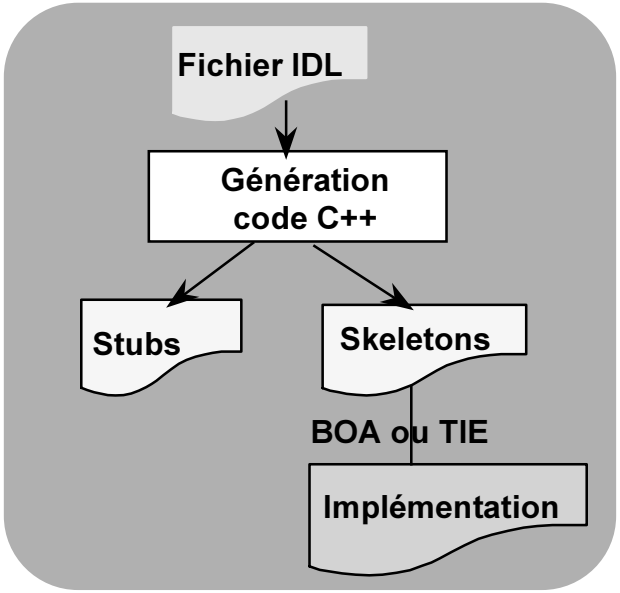
3- Fonctions supplémentaires

- Objets TIE : Facilite l'intégration des applications existantes.
- Filtres : Possibilité d'ajout de code pendant le transfert de requête.
- Smart Proxies : Spécialisation des proxies (cache).
- Loader :Gestion de la persistance des objets C++.
- Locator : Possibilité de définir son algorithme de recherche des serveurs.

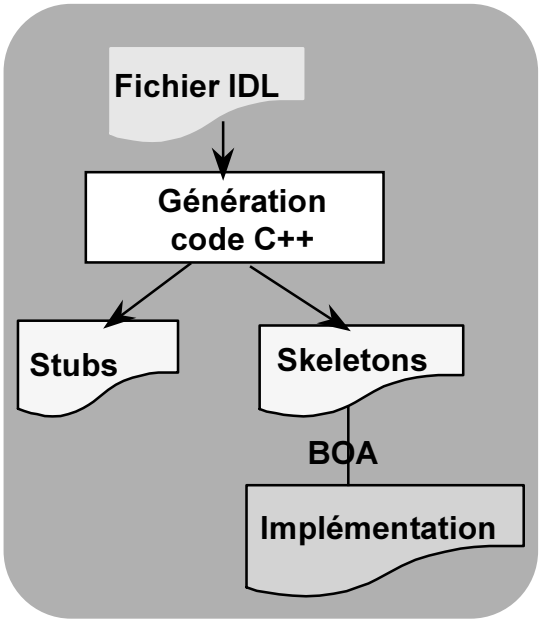
Objets TIE

Orbix propose deux approches différentes pour connecter les objets implémentation avec les skeletons :

- ☞ approche BOA (Basic Object Adapter) basée sur l'héritage
- ☞ approche TIE basée sur la délégation



Approche BOA




```
interface grid {
  readonly attribute short height;
  readonly attribute short width;
  ...
}
```

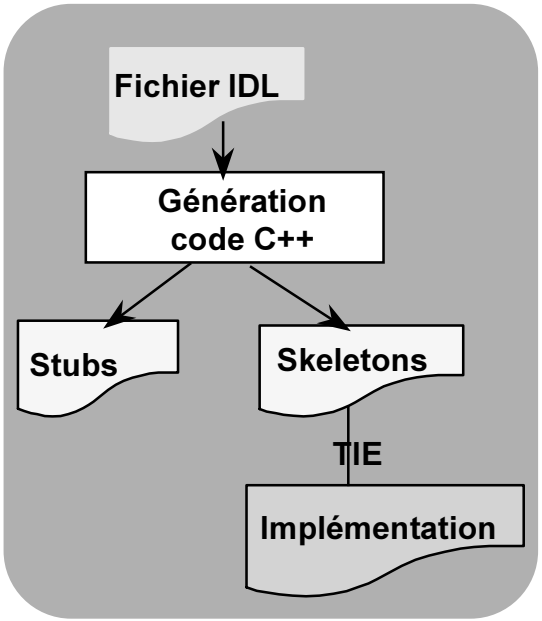
```
class grid
: public virtual CORBA::Object {
   ...
}
```

```
class gridBOAImpl
: public virtual grid {
   ...
   virtual CORBA::Short height() = 0;
   virtual CORBA::Short width() = 0;

...
}
```

Héritage

```
class grid _i : public gridBOAImpl {
  ...
  CORBA::Short height();
  CORBA::Short width();
...
}
```

Approche TIE (1/3)




```
class grid
: public virtual CORBA::Object {
   ...
}
```

```
interface grid {
  readonly attribute short height;
  readonly attribute short width;
  ... }
```

Macro

```
#define TIE_grid(X) grid##X
#define DEF_TIE_grid(X) \
    class grid##X : public virtual grid { \
        X* m_obj; \
      public :    \
        grid##X (X* objp, ...)
        : grid(), CORBA::Object(), m_obj(objp) {
          ... };
        virtual CORBA::Short height (...) { \
          return m_obj->height(...);  }; \
        virtual void width(...) {\
          return m_obj->height(...); }; \
        .... };
```

Srv_main

```
int main() {
  grid_ptr  myGrid = new TIE_grid(grid_i)(new grid_i(100,100));
   ...
   impl_is_ready(«grid»);
   ...
}
```

```
class grid _i {
   ...
  CORBA::Short height();
  void height(const CORBA::Short);

   CORBA::Short width();
   void width(const CORBA::Short);
   ...
};
DEF_TIE_grid(grid_i);
```

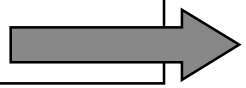
Approche TIE (2/3)

Fichier grid.hh

```
#define DEF_TIE_grid(X) \
   class grid##X : public virtual grid { \
      X* m_obj; \
     public :    \
       grid##X (X* objp, ...)
       : grid(), CORBA::Object(), m_obj(objp) {
         ... };
       virtual CORBA::Short height (...) { \
         return m_obj->height(...);  }; \
       virtual void width(...) {\
         return m_obj->height(...); }; \
    .... };
```

Fichier grid_i.hh

```
class grid _i {
  ...
 CORBA::Short height();
 void height(const CORBA::Short);

  CORBA::Short width();
  void width(const CORBA::Short);
  ...
};
DEF_TIE_grid(grid_i);
```



Définition de classe :

```
class gridgrid_i : public virtual grid { \
      grid_i* m_obj; \
    public :    \
      gridgrid_i (grid_i* objp, ...)
      : grid(), CORBA::Object(), m_obj(objp) {
        ... };
      virtual CORBA::Short height (...) { \
        return m_obj->height(...);  }; \
      virtual void width(...) {\
        return m_obj->height(...); }; \
.... };
```

Approche TIE (3/3)

Fichier grid.hh

```
#define TIE_grid(X) grid##X
```

Srv_main

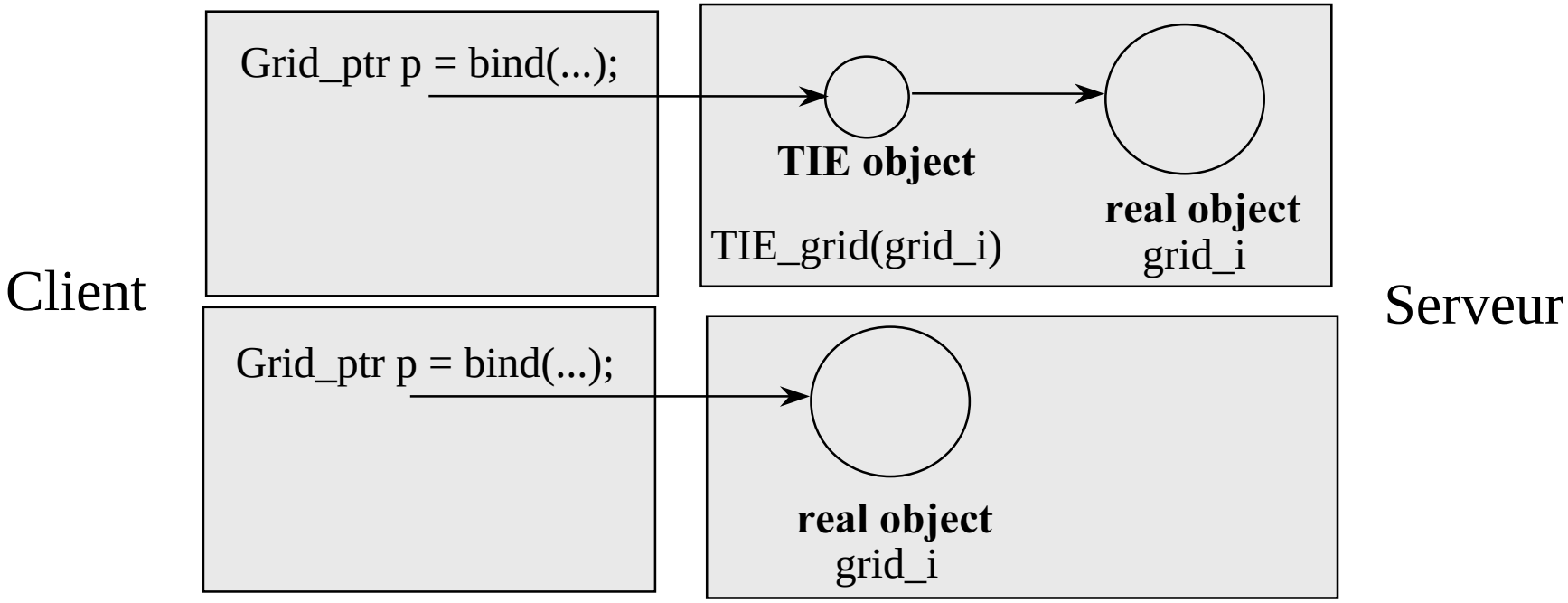
```
int main() {
  grid_ptr  myGrid = new TIE_grid(grid_i)(new grid_i(100,100));
   ...
   impl_is_ready(«grid»);
   ...
}
```

```
grid_i* g = new grid_i(100,100);
grid_ptr myGrid = new TIE_grid(grid_i)(g);
```

```
grid_ptr myGrid = new gridgrid_i(g);
```

Inconvénients approche TIE

- Approche spécifique à Orbix (pas CORBA).
- Création d'un objet supplémentaire qui relie l'objet implémentation avec le squelette.



CSTB le futur en construction

Filtres (1/2)

☞ Objet C++ qui permet d'ajouter du code pendant le transfert d'une requête :

- quand une invocation quitte le client;
- quand une invocation arrive au serveur;
- quand la réponse quitte le serveur;
- quand la réponse arrive au client;

☞ Utilisation pour le *debugging*, pour l'authentification grâce au *piggy-backing*.

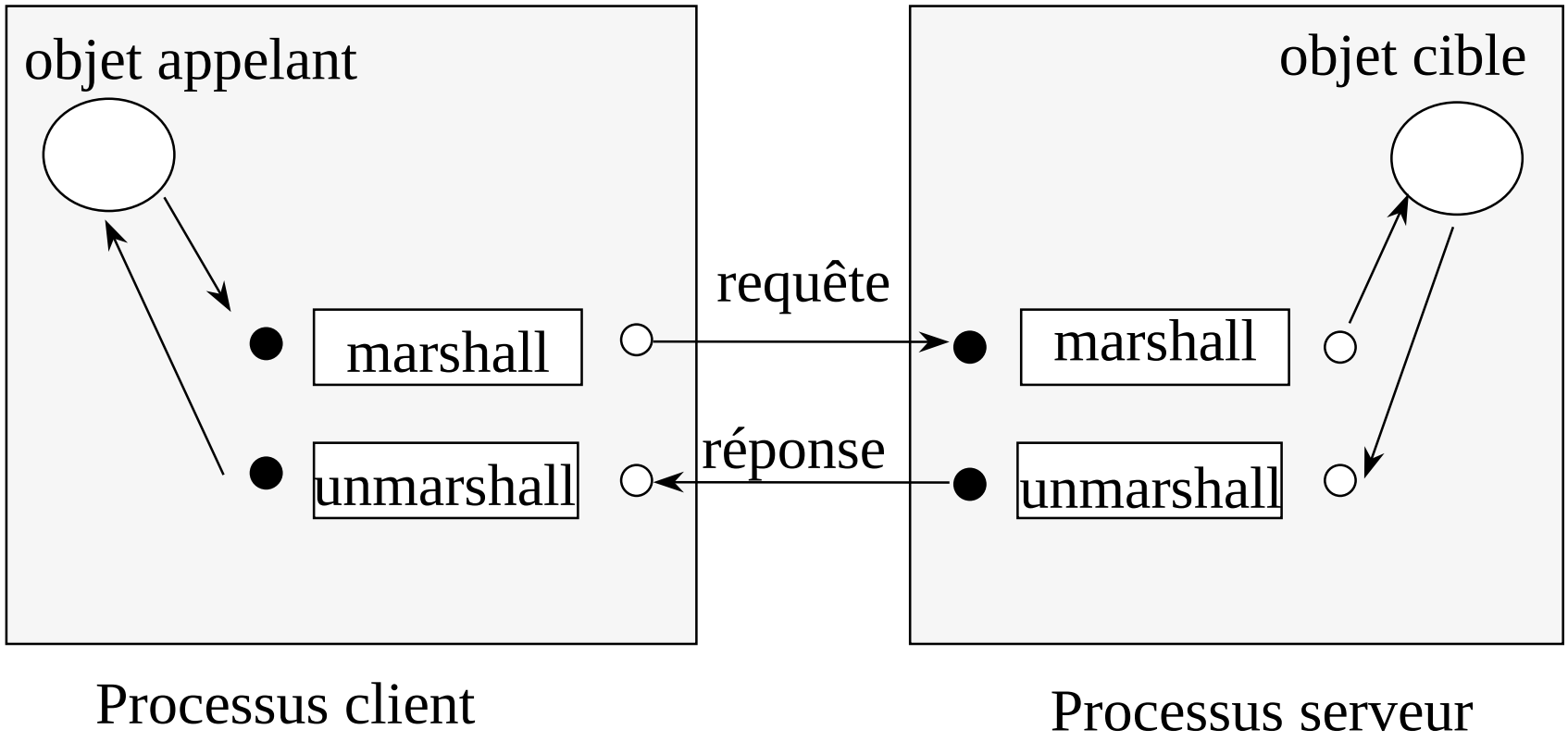
Filtres (2/2)

Deux catégories de filtre :

- Filtre par processus :
  - applicable au client et au serveur;
  - spécialisation de la classe *CORBA::Filter*
- Filtre par objet :
  - applicable au serveur uniquement;
  - approche TIE obligatoire;
  - possibilité de chaînage des filtres.

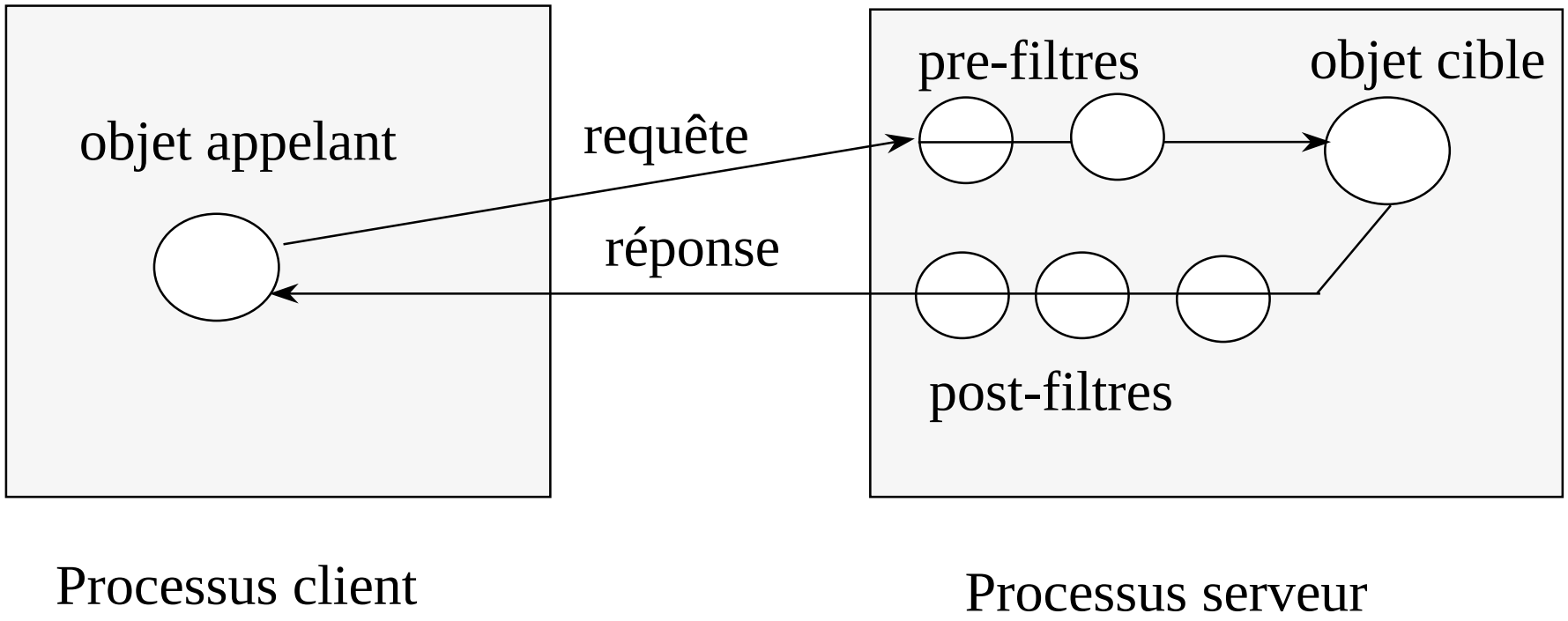
Filtre par processus

☞ 8 points d'ancrage



Filtre par objet

☞ 2 points d'ancrage



CSTB
le futur en construction

Proxies : génération

☞Le compilateur IDL génère pour chaque interface :
- une classe appelée «*proxy*»
- une classe appelée «*factory*»
- une instance de la classe «*factory*» (objet global)

☞ Exemple pour l'interface IDL Grid :
- *Grid* : classe *proxy*
- *GridProxyFactoryClass* : classe *factory*
- *GridProxyFactory* : instance de la classe *GridProxyFactoryClass*

CSTB
le futur en construction

Proxies : fonctions

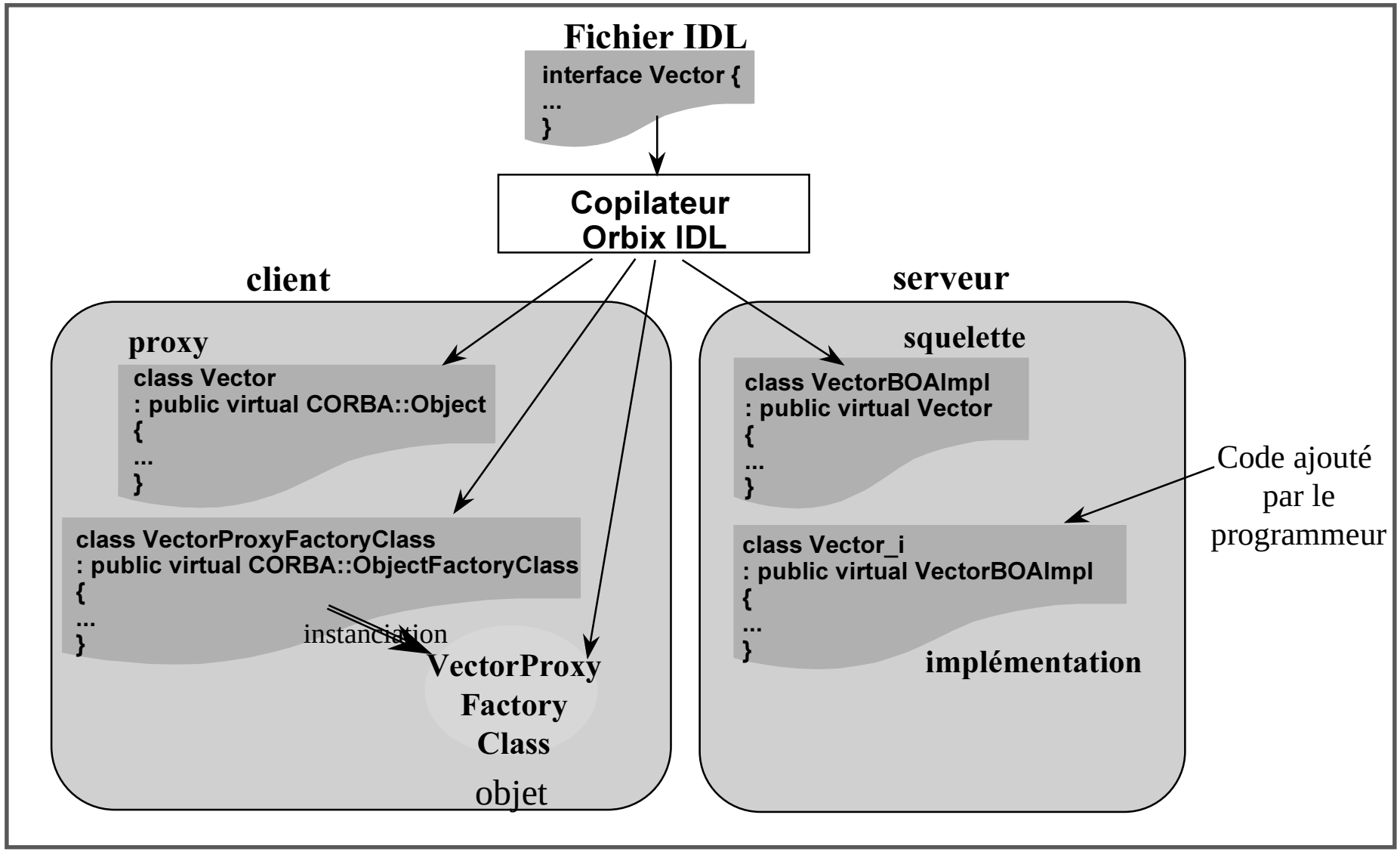
☞ Un *proxy* est un objet C++ (instance des *stubs*) qui a pour fonction :

- le *marshalling* des paramètres *in* et *inout*;
- la transmission de la requête;
- la réception de la réponse;
- le *unmarshalling* des paramètres *out* et *inout* et de la valeur de retour.

☞ Le *proxy* est créé par un objet C++ nommé *factory*.

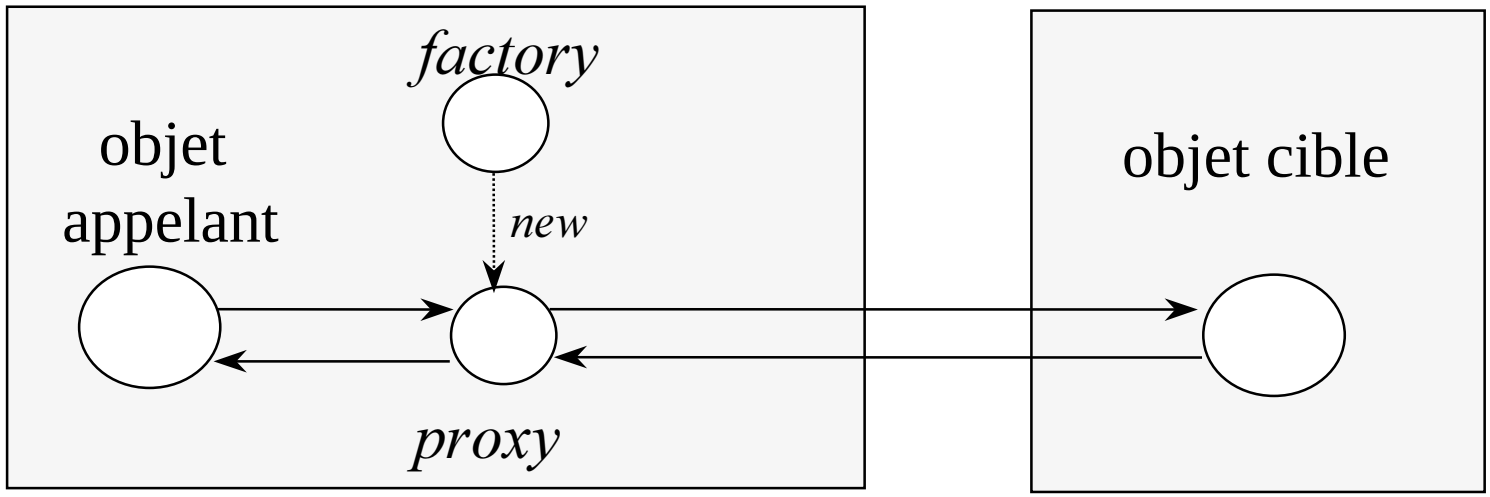
☞ L'objet *factory* définit un opérateur *New* qui à pour fonction de créer un *proxy*.

Proxies : exemple



Proxies : illustration

Création d'un *proxy*



CSTB le futur en construction

Smart Proxies : fonctions

- Un *smart proxy* est un objet C++ qui hérite des fonctionnaltiés de l'objet *proxy* et qui ajoute des traitements spécifiques.
- L'objet *smart proxy* est créé par l'objet C++ *smart factory* .
- L'objet *smart factory* hérite des fonctionnalités de l'objet *factory* et redéfinit l'opérateur *New* afin de créer non pas un objet *proxy* mais un objet *smart proxy*.

CSTB le futur en construction

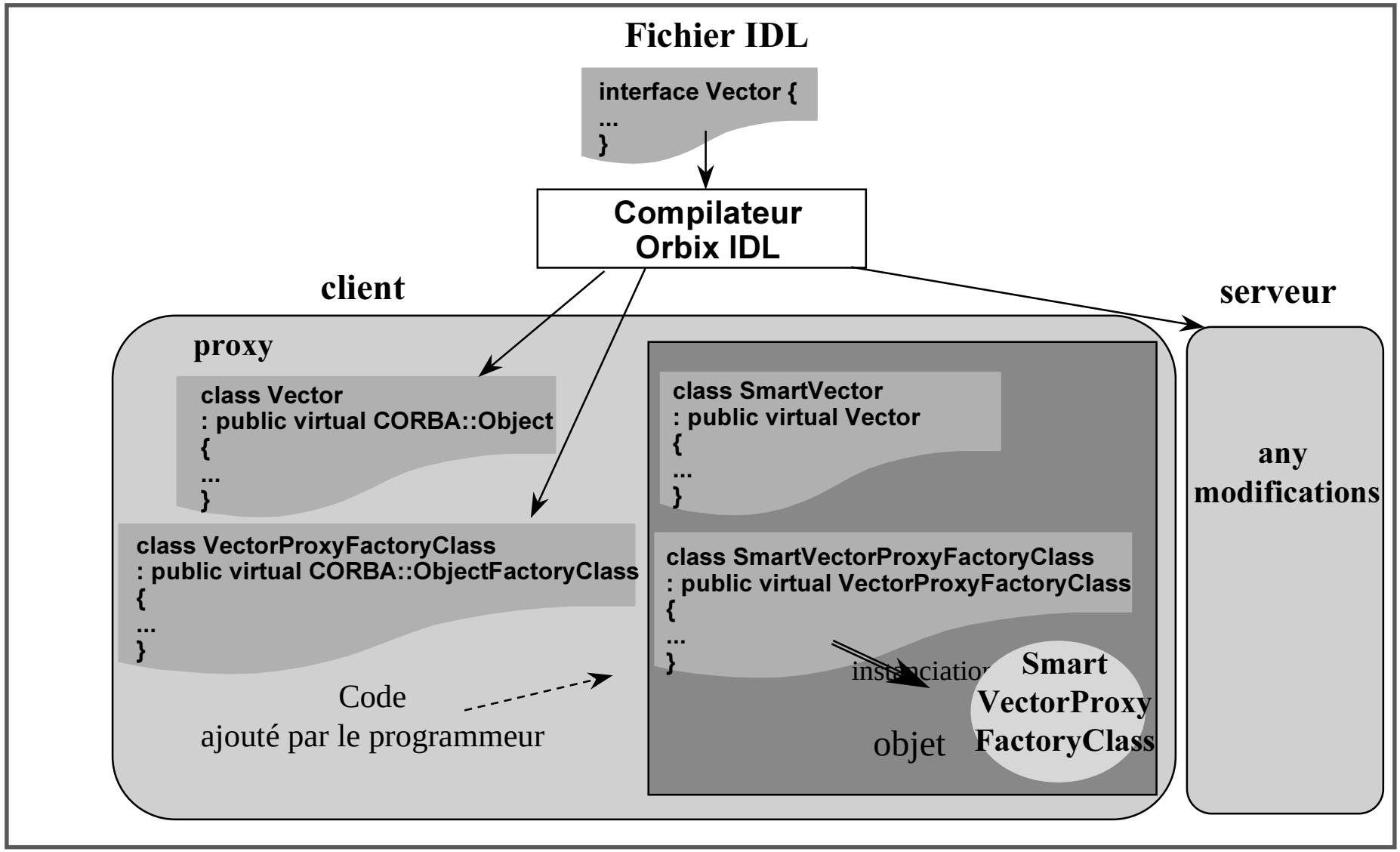
Smart Proxies : utilisation

☞ Utilisation lors des invocations statiques uniquement;

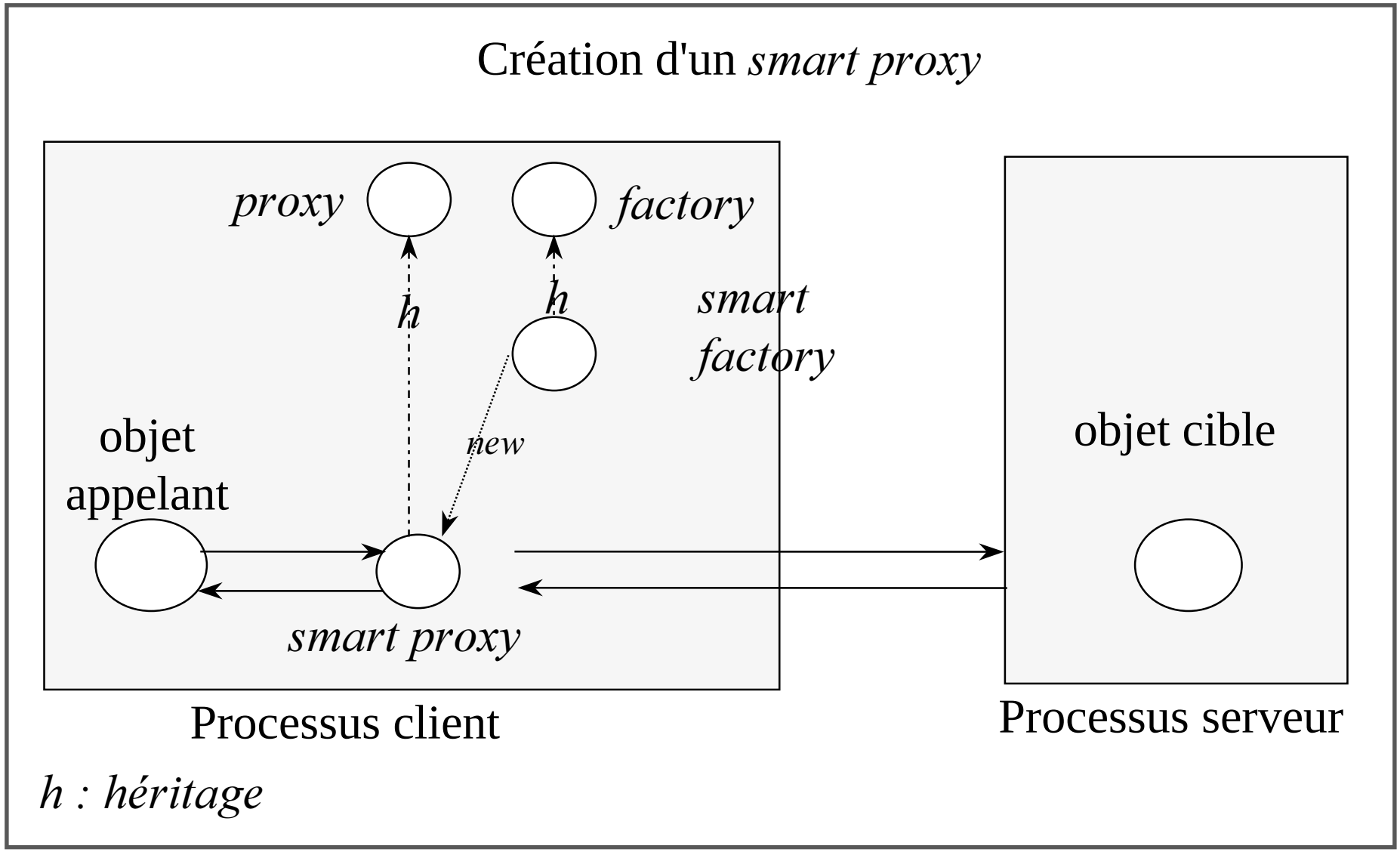
☞ Rôle des smart Proxies :

- cacher des valeurs chez les clients;
- utiliser des ressources locales au client (imprimante, ...);
- faire du "load balancing" (sélection du serveur le moins chargé);

Smartproxies : exemple



Smart Proxies : illustration

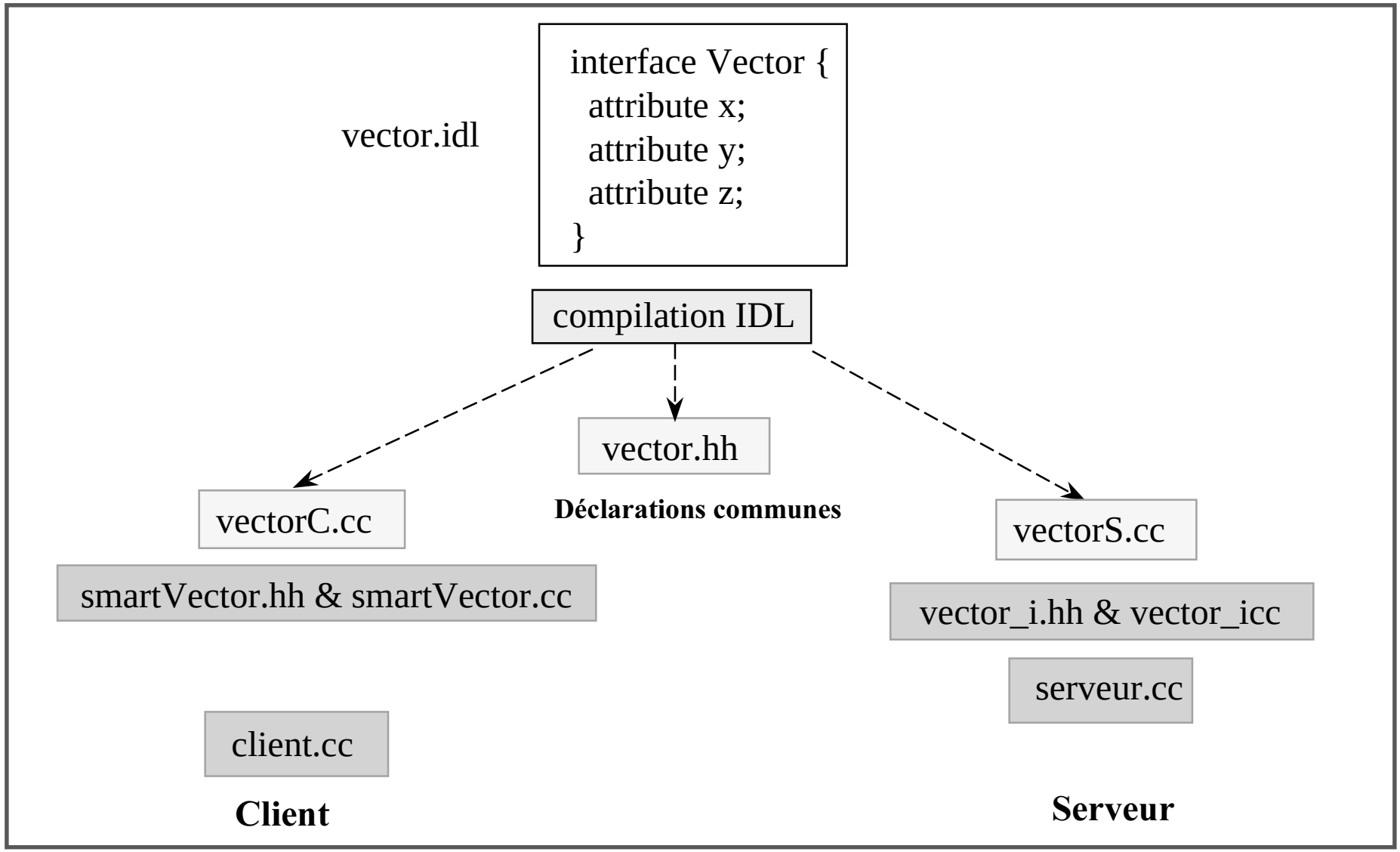


CSTB le futur en construction

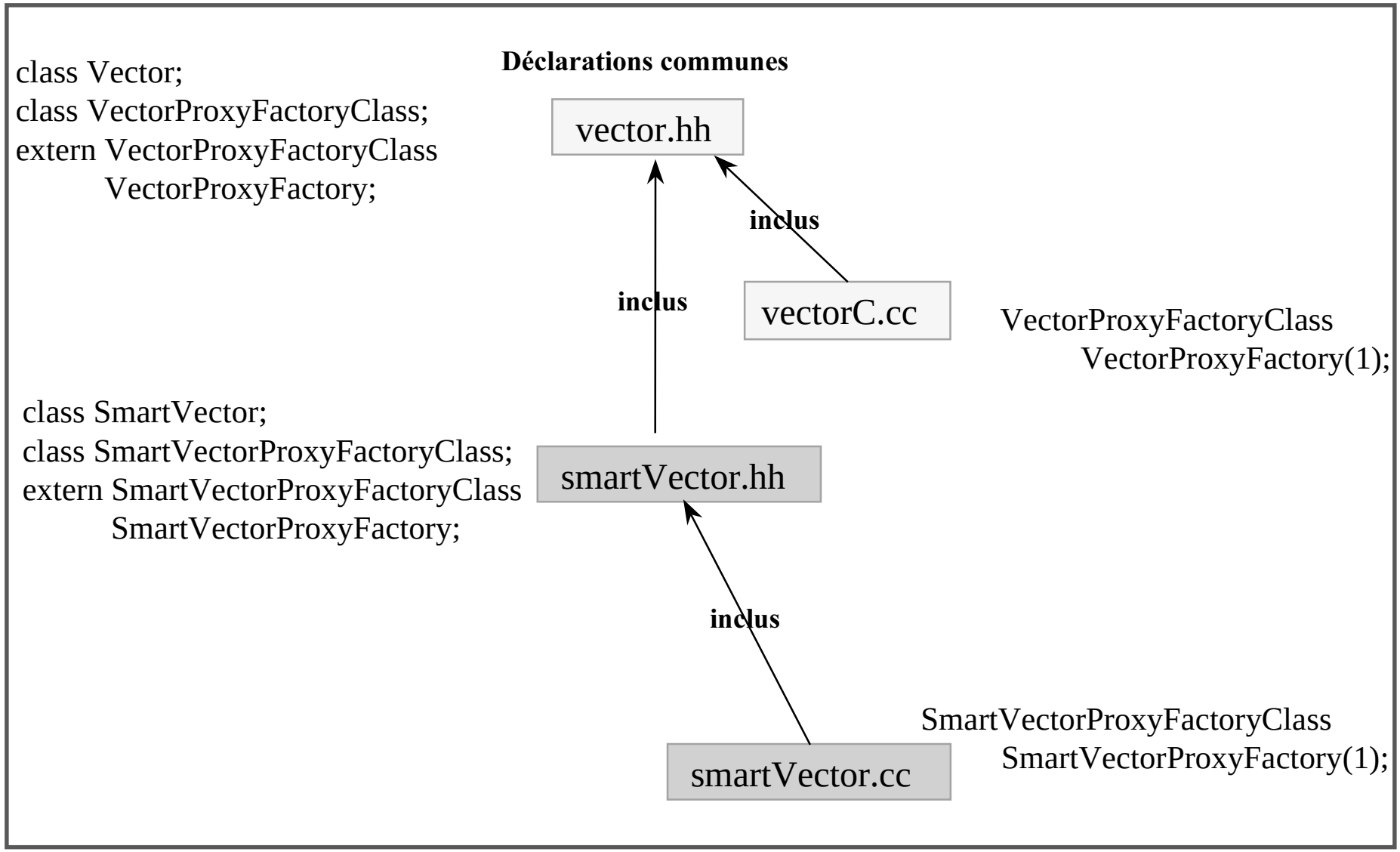
Smart proxy factories

- Les objets *factory* (instances des *factory* classes) sont gérés par un manager.
- Une chaîne de *factories* peut être associée à un *proxy*.
- Le manager enregistre la chaîne de *factories* associée à un *proxy*, la dernière *factory* est la factory créée par défaut.
- Lors de la demande de création d'un *proxy*, le manager parcourt la liste des *factories* jusqu'à ce qu'une *factory* puisse créer le bon proxy.

Exemple : mécanisme de cache



Exemple : mécanisme de cache



Exemple : mécanisme de cache

smartVector.hh

Héritage proxy par défaut généré par le compilateur IDL

```
class SmartVector :
public virtual Vector {
  protected:
    CORBA::Double x_cache, y_cache, z_cache;
    CORBA::Boolean x_valid, y_valid, z_valid;
  public:
    SmartVector(char* objref);

    virtual void x (CORBA::Double x, ...);
    virtual CORBA::Double x(...);

    virtual void y (CORBA::Double y, ...);
    virtual CORBA::Double y(...);

    virtual void z (CORBA::Double z, ...);
    virtual CORBA::Double z(...);
};
```

Héritage factory par défaut généré par le compilateur IDL

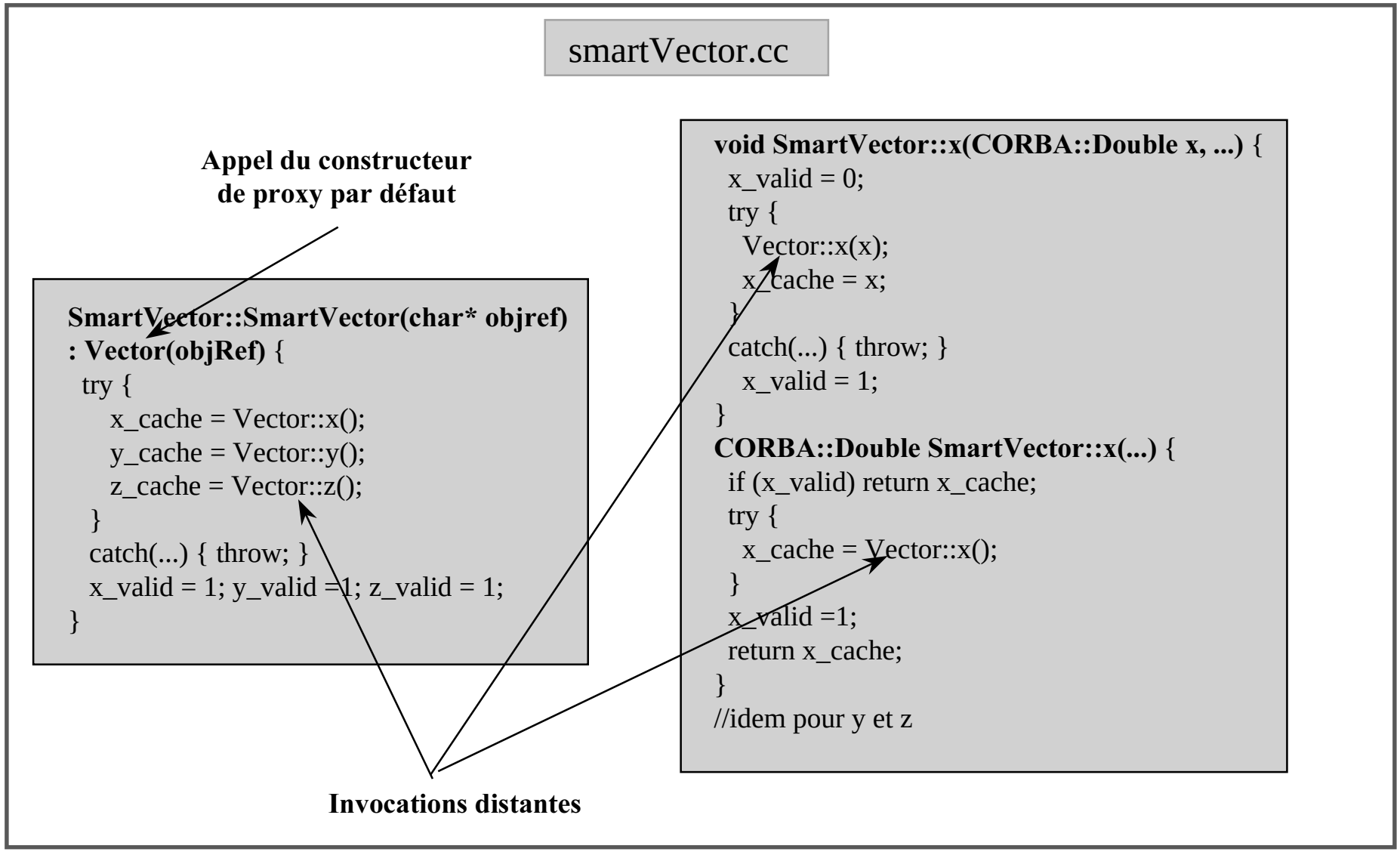
```
class SmartVectorProxyFactoryClass :
public virtual VectorProxyFactoryClass {
  public :
    SmartVectorProxyFactoryClass
    : CORBA::ProxyFactory(Vector_IR) {}
   virtual void* New(...) {
      return (Vector_ptr)new SmartVector(...);
   }
};
```

Création d'un SmartVector et non plus d'un Vector

```
SmartVectorProxyFactoryClass SmartVectorProxyFactory;
```

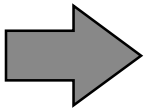
Création de l'objet global factory permet de créer à la demande des SmartVector

Exemple : mécanisme de cache



Mécanisme de notification

- L'implémentation d'un mécanisme de cache chez les clients requiert un mécanisme de notification.
- Les clients doivent être notifiés par le serveur en cas de changement des valeurs des variables gérées par le cache.
- Mécanisme qui permet de gérer la cohérence des données résidant sur le serveur et dupliquées chez les clients.



Implémentation d'un mécanisme de notification

CSTB le futur en construction

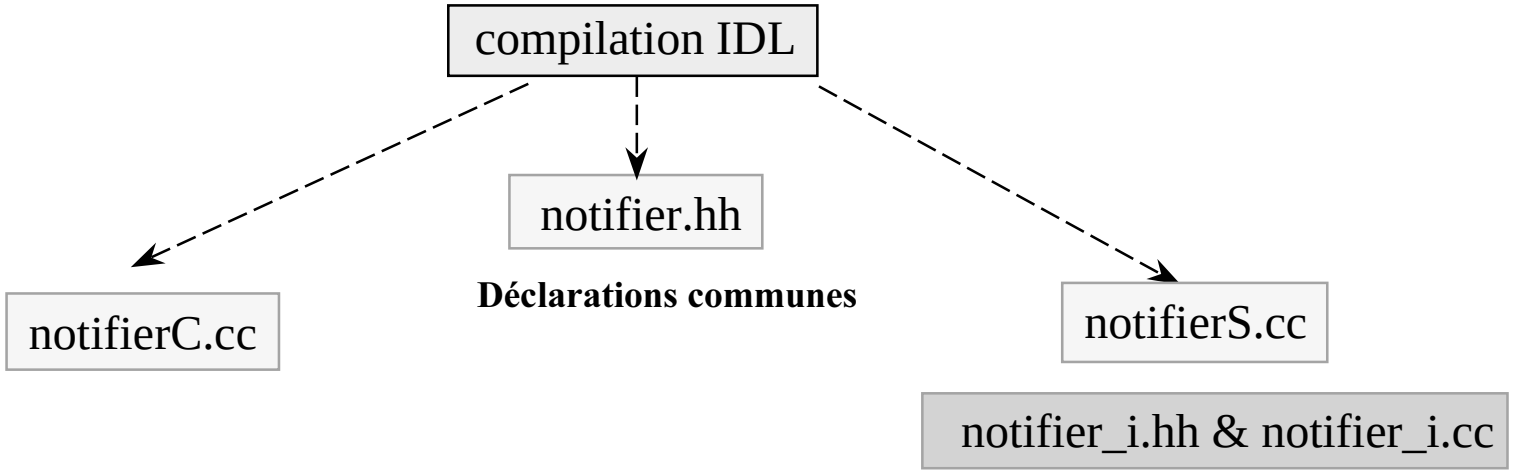
Les différentes étapes

- Création de 2 nouvelles interfaces IDL :
  cacheInvalidate et registerProxy
- Codage des objets implémentations
- Intégration de la notification avec le mécanisme de cache :
  création d'une nouvelle interface : smartRegisterVector
  création d'une nouvelle factory : smartRegisterVectorFactorClass

Spécification IDL

notifier.idl

```
interface CacheInvalidate {
  oneway void callback(in char c);
};
interface RegisterProxy {
  void signalOn (in CacheInvalidate proxy);
  void signalOff (in CacheInvalidate proxy);
};
```



Objet implémentation

notifier_i.hh

```
class CacheInvalidate_i {
  virtual void callback (CORBA::Char c, ...);
};

class RegisterProxy_i
: public virtual RegisterProxyBOAImpl {
    CacheInvalidate_ptr m_proxies[20];
    CORBA::Ulong m_proxy;
  protected :
    virtual void notify(CORBA::Char);
  public :
    RegisterProxy_i() : m_proxy(0) {};
    virtual void signalOn (CacheInvalidate_ptr proxy, ...);
    virtual void signalOff (CacheInvalidate_ptr proxy, ...);
};
```

Pas d'héritage
Création d'objet par la méthode TIE

Tableau permettant d'enregistrer les adresses des clients devant être notifiés

Nombre de client à notifier

Méthode qui effectue la notification

Enregistre (signalOn) l'adresse du client qui devra être notifié
Supprime (signalOff) l'adresse du client

Objet implémentation

notifier_i.cc

```
#include «notifier_i.hh»
void RegisterProxy_i::notify(CORBA::Char c) {
  CORBA::ULong i;
  for (i=0; i<m_proxy; i++)
    try {
      m_proxies[i]->callback(c);
    }
    catch (...) { ...}
}

void RegisterProxy_i::signalOn (CacheInvalidate_ptr proxy, ...) {
  if(m_proxy >= 20) return;
  CacheInvalidate::_duplicate(proxy);
  m_proxies[m_proxy++] = proxy;
}

void RegisterProxy_i::signalOff (CacheInvalidate_ptr proxy, ...) {
  CORBA::Ulong i;
  for(i =0; i> m_proxy; i++)
    if(m_proxies[i] == proxy) break;
  CORBA::release(m_proxies[i]);
  m_proxy--;
  while(i<m_proxy) m_proxies[i] = m_proxies[++i];
}
```

Invocation distante

Intégration de la notification

```
interface CacheInvalidate {
  oneway void callback(in char c);
};
interface RegisterProxy {
  void signalOn (in CacheInvalidate proxy);
  void signalOff (in CacheInvalidate proxy);
};
```

notifier.idl

```
#include «notifier.idl»
interface Vector {
  attribute x;
  attribute y;
  attribute z;
};
interface RegisterVector : RegisterProxy, Vector {};
```

vector.idl

vectorC.o | notifierS.o | smartVector.o | notifier_i.o | client.cc

Client Vector
Serveur Notifier

vectorS.o | notifierS.o | vector_i.o | notifier_i.o | serveur.cc

Serveur Vector
Client Notifier

Adaptation du cache

smartVector.hh

```
class SmartVector :
public virtual Vector {
   ...
  // idem
};

class TIE_CacheInvalidate(SmartRegisterVector);

class SmartRegisterVector
: public virtual RegisterVector,
  public virtual SmartVector {
    TIE_CacheInvalidate(SmartRegisterVector)* m_self;
  public :
    SmartRegisterVector(char*);
    virtual ~SmartRegisterVector();
    virtual void callback(CORBA::Cahr c, ...);

    virtual void x(CORBA::Double x, ...);
    virtual CORBA::Double x (...);
   ...
};
```

Référence avant de la classe «CacheInvalidateSmartRegisterVector» défini plus loin

Définition d'un pointeur sur un objet de type «CacheInvalidateSmartRegisterVector»

Adaptation du cache

smartVector.hh

```
DEF_TIE_CacheInvalidate(SmartRegisterVector);

class SmartRegisterVectorFactoryClass
: public virtual RegisterVectorProxyFactoryClass {
  public :
    SmartRegisterVectorProxyFactoryClass()
    : CORBA::ProxyFactory(RegisterVector_IR) {}
    virtual void* New(char *OR, ...) {
      return (RegisterVector_ptr) new SmartRegisterVector(OR);
    }
};

SmartRegisterVectorFactoryClass srvfc;
```

Définition de la classe «CacheInvalidateSmartRegisterVector»

Héritage factory par défaut généré par le compilateur IDL

Création d'un SmartRegisterVector et non plus d'un SmartVector

Création de l'objet global factory permet de créer à la demande des SmartVector

Adaptation du cache

smartVector.cc

```
SmartRegisterVector::SmartRegisterVector(char* OR)
: RegisterVector(OR), SmartVector(OR) {
  try {
    my_self = new TIE_CacheInvalidate(SmartRegisterVector)(this);
    RegisterVector::signalOn(m_self);
  }
  catch(const CORBA::SystemException& se) {...}
}

SmartRegisterVector::~SmartRegisterVector() {
  signalOff(m_self);
}

void SmartRegisterVector::callback(CORBA::Char c, ...) {
  switch (c) {
    case 'x' : x_valid = 0; break;
    case 'y' : y_valid = 0; break;
    case 'z' : z_valid = 0; break;
    default : ...
  }
}
```

Instanciation de la class «CacheInvalidateSmartRegisterVector»

Invocation distante à l'objet implémentation RegisterVector_i

Invocation distante à l'objet implémentation RegisterVector_i

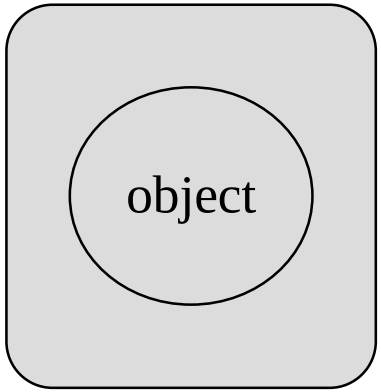
Adaptation du cache

```
void SmartRegisterVector::x(CORBA::Double x, ...) {
  CORBA::Orbix.processEvents(0);
  SmartVector::x(x, ...);
}

CORBA::Double SmartRegisterVector::x( ...) {
  CORBA::Orbix.processEvents(0);
  return SmartVector::x(...);
}

//idem pour y et z
```

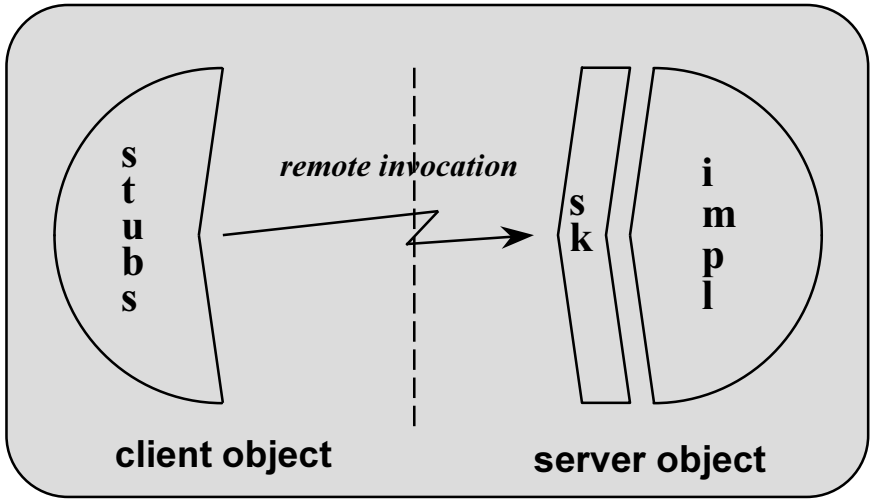
Orbix traite tous les évènements avant de sortir de la fonction (équivalent impl_is_ready)

Objet distribué

View of an object in a monolithic application

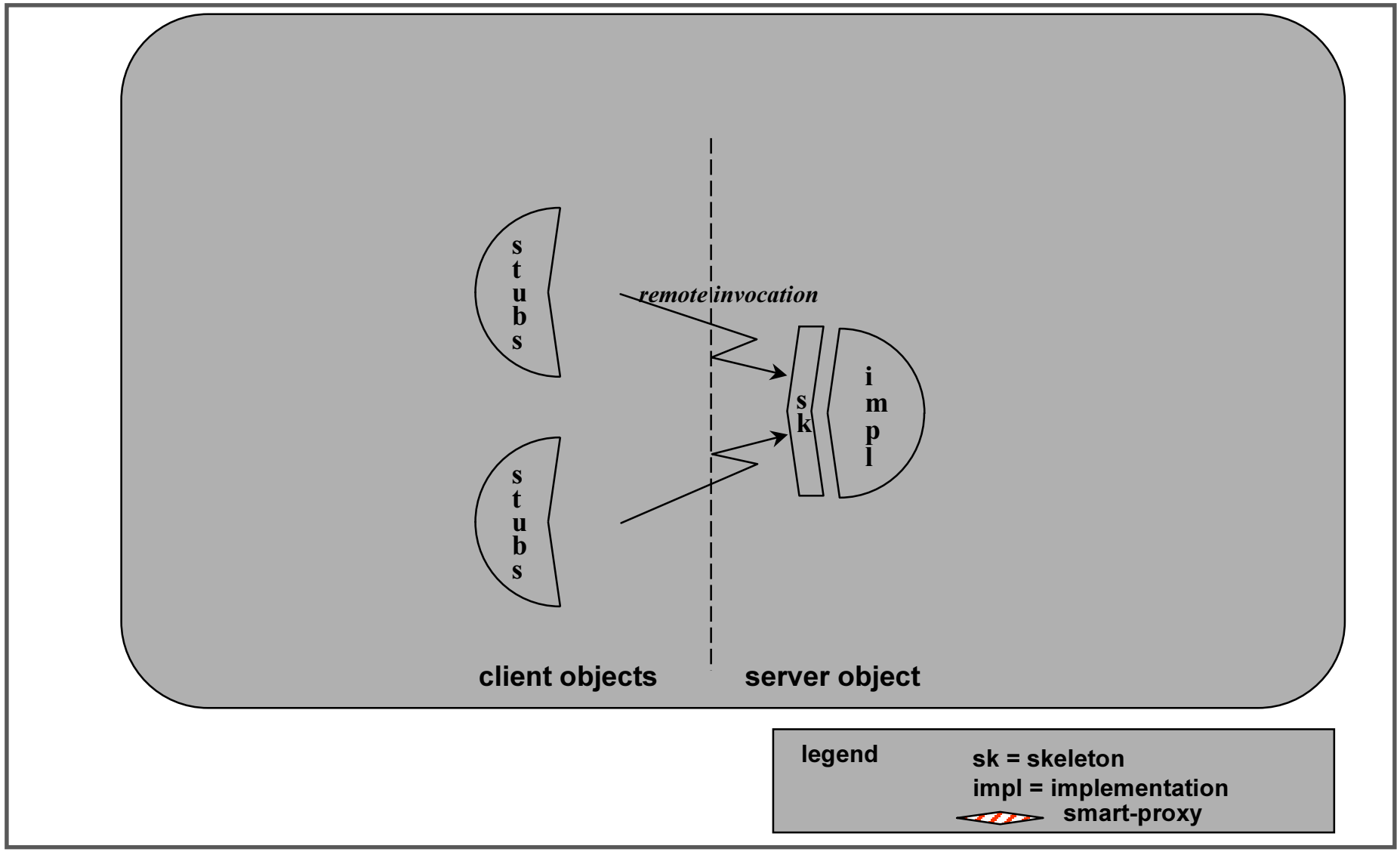


View of an object in a distributed application

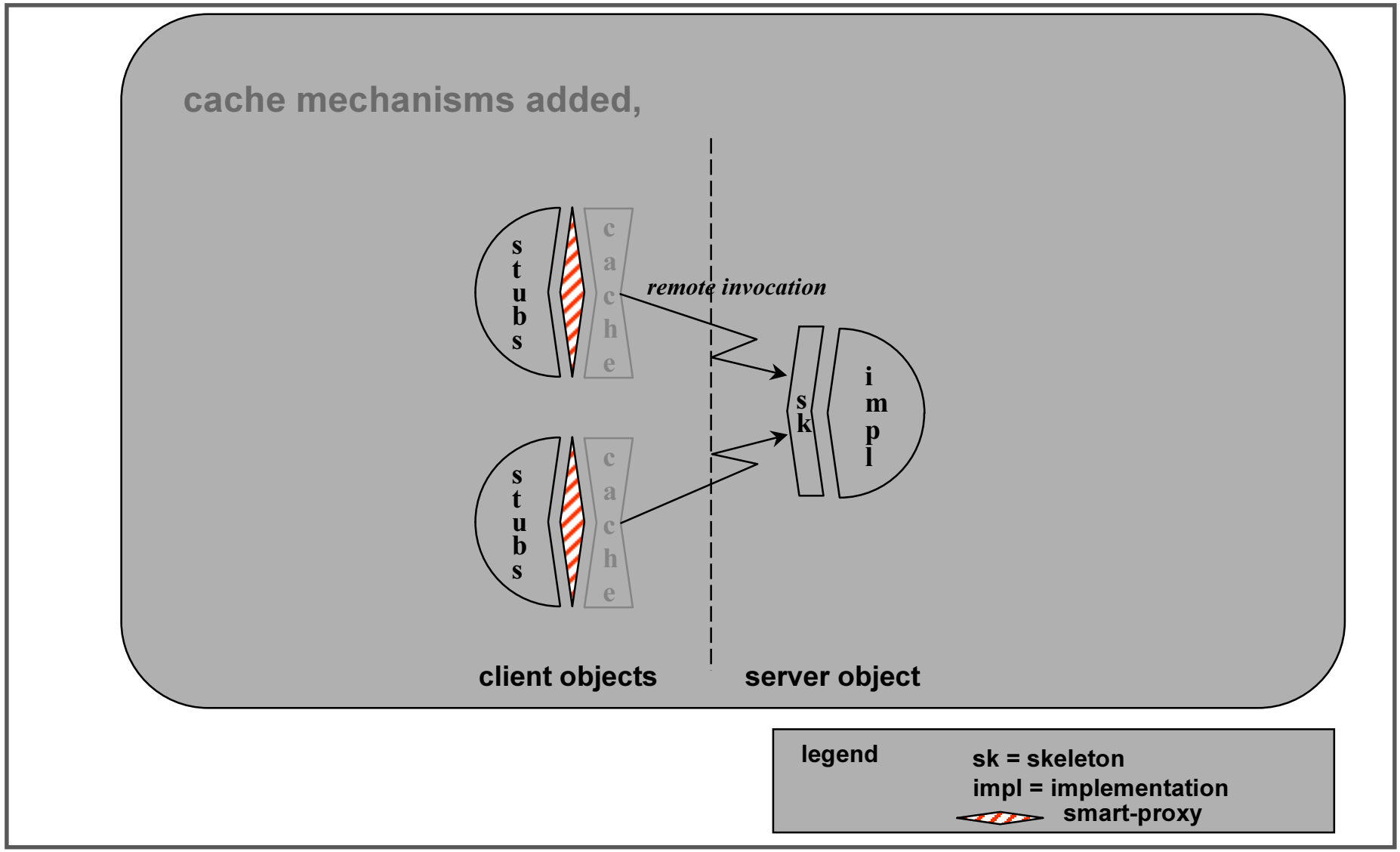


legend	sk = skeleton impl = implementation

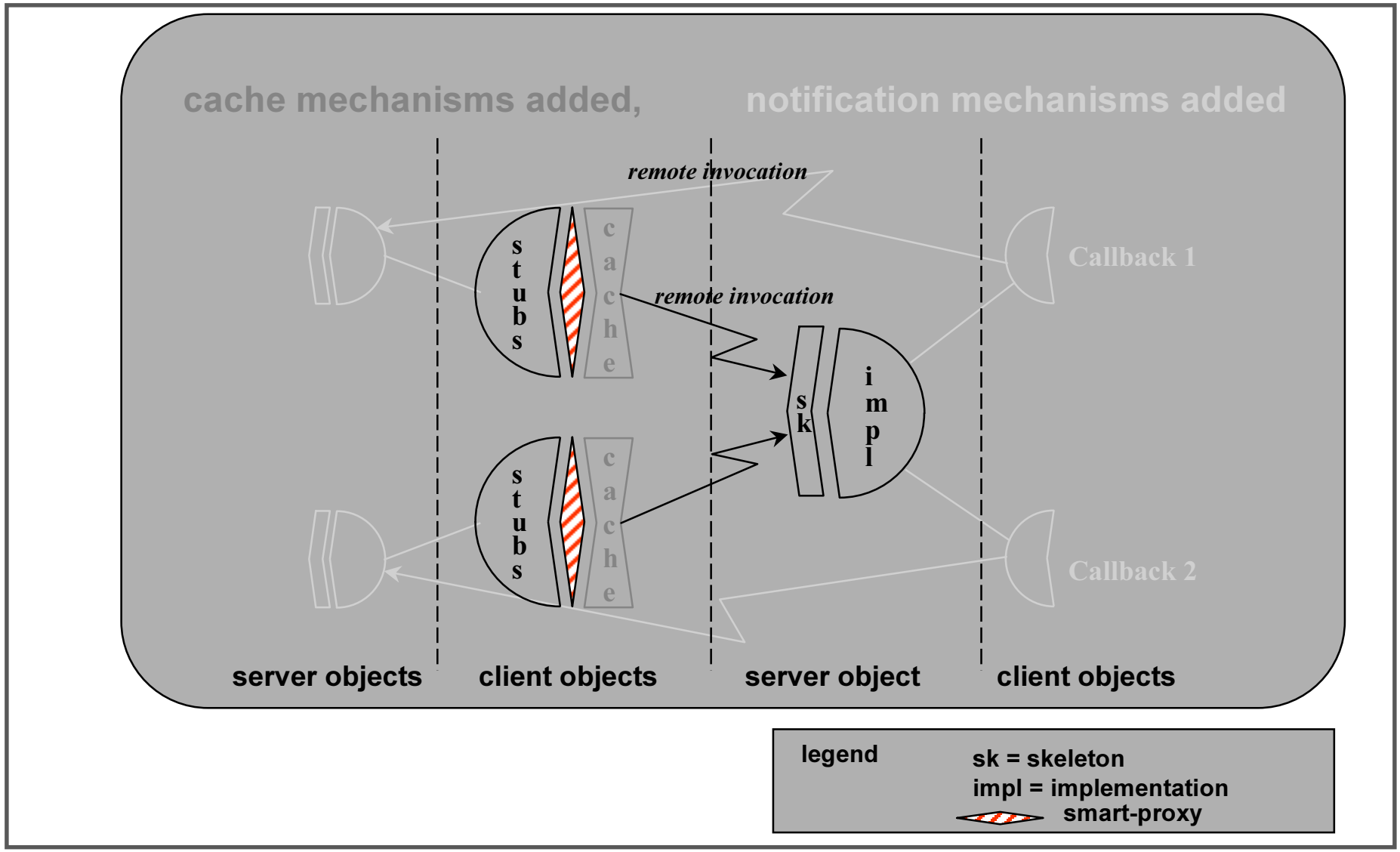
Objet distribué



Objet distribué + cache



Objet distribué + cache + notification



CSTB
le futur en construction

Loader

☞ Objet C++ permettant de supporter la persistance des objets;

☞ Spécialisation de la classe *CORBA::LoaderClass;*

☞Différentes architectures :

- un loader par classe;
- un loader par processus;
- un loader par base de données.

CSTB
le futur en construction

Locator

☞ Objet C++ permettant aux clients de rechercher un serveur sur le réseau.

☞ Un locator par défaut utilise deux fichiers spécifiques :
- Orbix.hosts
- Orbix.hostgroups

☞ Possibilité d'écrire son propre locator en spécialisant la class *CORBA::locatorClass*.

4- Intégration d'Orbix

Orbix peut-être intégré à d'autres produits :

- Microsoft OLE (Object Linking and Embedding);
- Isis pour la tolérance aux pannes;
- VXWorks pour le temps rééel;
- ObjectStore pour la persistance;
- OrbixWeb avec un binding Java.

Références

[Orbix 96] "Orbix 2 Programming Guide" Iona Technologies Ltd

[Orbix 96] "Orbix 2 Reference Guide" Iona Technologies Ltd

World Wide Web : http://www.iona.com