PyExp

Python pour les Expériences

Laurent Pointal - LURE (CNRS-CEA-MEN)

laurent.pointal@lure.u-psud.fr

Présentation

- Situation du contexte.
- Problématique.
- Objectifs & structure de PyExp.
- Liaisons avec des composants externes.
- [démonstration]

.

Le LURE*

Sources nationales de Rayonnement Synchrotron

400 personnes

1800 utilisateurs par an

40 postes expérimentaux (50 montages)

3200 heures de fonctionnement/an

*Laboratoire pour l'Utilisation du Rayonnement Electromagnétique

http://www.lure.u-psud.fr/

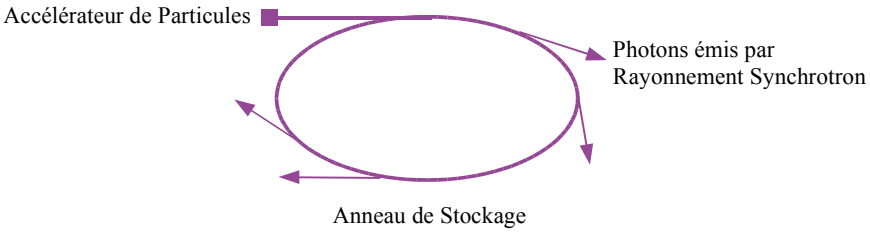
Laboratoire mixte CNRS + CEA + Ministère de l'Education Nationale.

Environ 230 Ingénieurs, Techniciens et Administratifs, 110 chercheurs, 70 thésards et postdoctoraux. Collaborations avec ~30 laboratoires extétieurs et ~20 partenaires industriels.

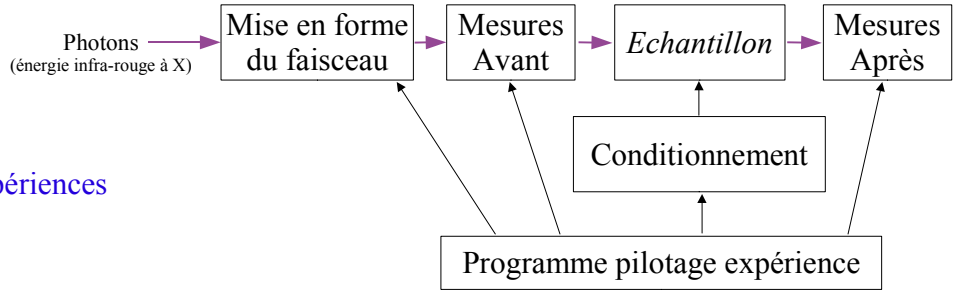
1/3 d'utilisateurs étrangers.

Installations

Sources



Expériences



Problématique

- Applications multi-expériences.
- Profils d'utilisateurs variés.
- Besoins applicatifs divers.
- Matériels disparates.
- Fonctionnement par scripts.
- Stabilité (applications tournant 24h/24).
- Prévention des pannes (logs...).

Multi-expériences:
ciblé au LURE d'ici la fermeture: 2~3 postes.
sur SOLEIL: plus de 20 postes à terme
(et on espère bien voir le système repris ailleurs)

Profils d'utilisateurs:
connaissances ou non en informatique
préférence GUI / ligne de commande
utilisateur occasionnel / périodique / permanent

Besoins applicatifs:
expériences standard, répétitives
expériences très changeantes (protocole expérimental, matériels utilisés)
à venir sur SOLEIL, expériences à réglage automatique (asservissements)

Matériels disparates:
on peut définir un standard pour les périphériques courants (moteurs, compteurs)...
mais recherche oblige, des matériels spécifiques avec des caractériqtiques précises
permettant de répondre à des besoins sont là.

Choix de Python

- Accès facile.
- Moderne (OOP, exceptions multithreading...).
- Extensible.
- Dynamique.
- Portable.
- Libre (gratuit et open-source).
- Communauté dynamique et en croissance.

Accès Facile:
Les chercheurs doivent pouvoir apprende le langage et écrire des scripts, pour exprimer des logiques expérimentales (et les réaliser), pour piloter de nouveaux matériels.

Moderne:
Facilite la conception de l'application autour des objets représentant les concepts liés aux expériences, standardise (et force) le traitement des erreurs, autorise les traitements parallèles (gestion de plusieurs instruments).

Extensible:
Nous permet d'écrire nos extensions pour les fonctionnalités où Python n'a pas de module standard (ex. accès bus instrumentation GPIB).

Dynamique:
Nous permet de charger les modules nécessaires en fonction des besoins de l'expérience, permet aux utilisateurs de définir leurs propres modules.

Portable:
Permet l'utilisation sous Windows et Linux principalement.

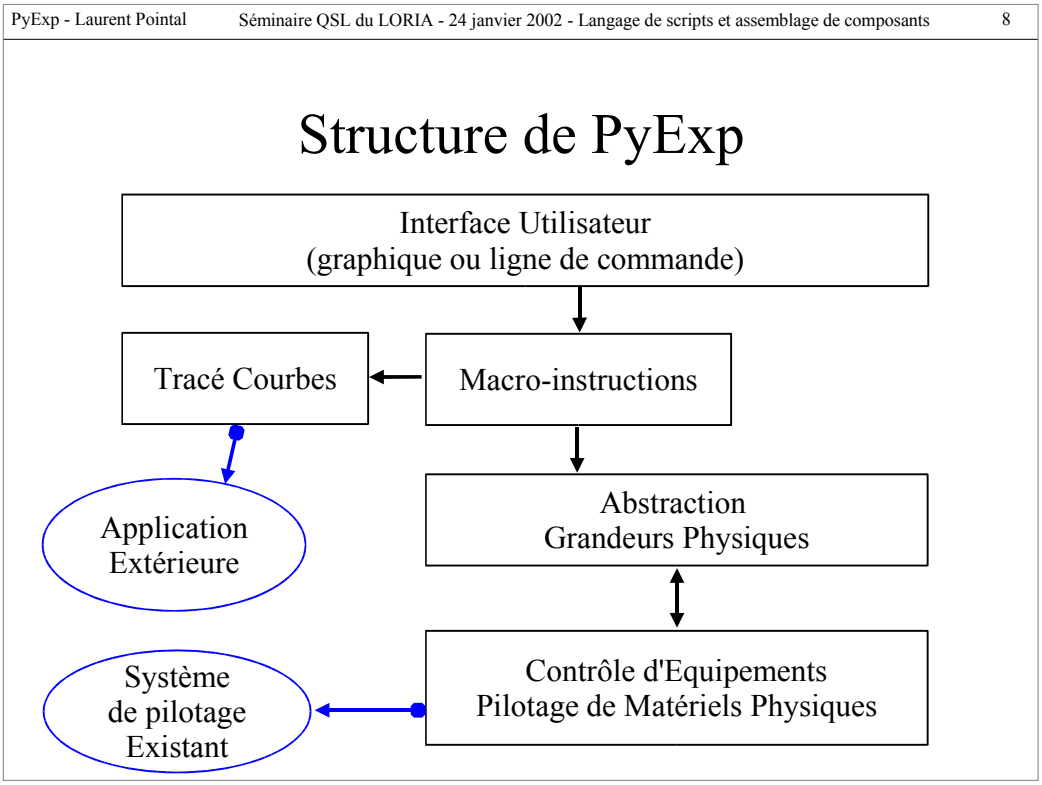
URLs:
Site principal de Python:
http://www.python.org/
Collection de liens (dont pas mal en français):
http://www.rimbault.net/python/
Où Eric S. Raymond explique pourquoi (et comment) il est passé à Python:
http://www.linuxjournal.com/article.php?sid=3882
Présentation du langage:
http://www.linux-center.org/articles/9812/python.html

Objectifs de PyExp

- Scripts pour la logique expérimentale,
- Abstraction grandeurs physiques de l'expérience,
- Macros-instructions,
- Vision séquentielle du processus logique,
  - masquage du multithread, synchronisations...
- Adaptation à différents matériels / protocoles,
- Possibilité d'interface graphique,
- Extensible, multi-plateforme, libre...

Les scripts expérimentaux doivent pouvoir être écrits par deschercheurs aussi bien que par des informaticiens.

Les macros-instructions doivent permettre d'exprimer des séquences de logique expérimentale récurrentes, et de les paramétrer.



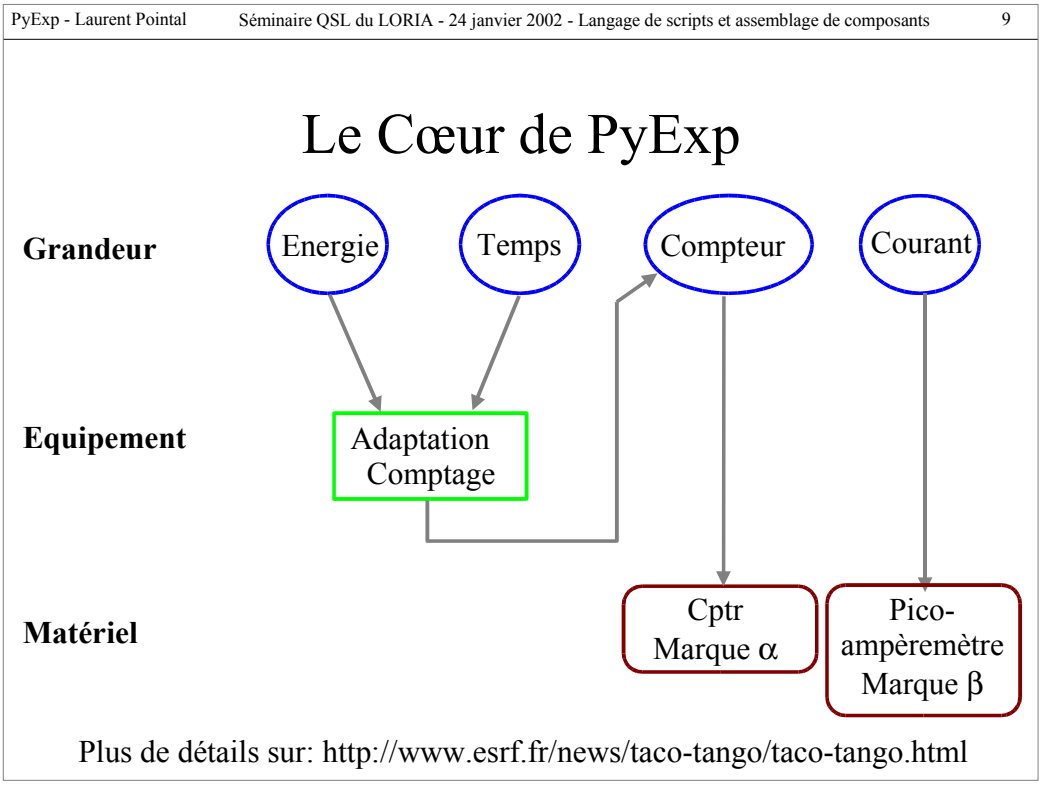
Différents profils d'utilisateurs, certainspréfèrent la GUI, d'autres ne jurent que par la ligne de commande.

Déport du tracé des courbes (et éventuellement des pré-traitements) vers outils externes: pas notre métier, logiciels existants très bien et connus par les chercheurs.

Système de graphe d'acteurs (grandeurs, équipements, matériels) pour:

- implémenter l'abstraction des grandeurs physiques et les relations entre ces grandeurs, les équipements complexes contrôlés, et les matériels physiques pilotés.
- masquer la réalisation des opérations dans des threads concurrents.

Nécessité de pouvoir se connecter à des systèmes existants (pas uniquement notre InfoExp), pour éviter d'avoir à coder, pour permettre l'utilisation de pyexp hors du LURE (...SOLEIL).
Intérêt de Python et de ses capacités d'intégration/agrégation de technologies.



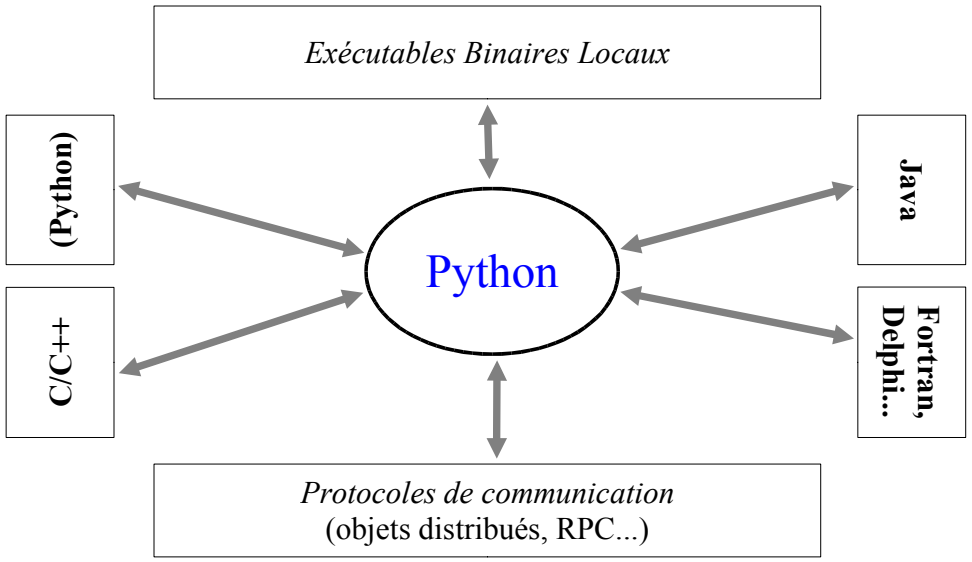
Exemple très simplifié.

Dans la réalité, PyExp devrait ammener à des graphes nettement plus important.

C'est ce système d'arborescence, accompagné d'envoi de messages, qui est utilisé pour masquer le multithreading. *Partant des noeuds désignés pour une requête d'opération, celle-ci est préparée en étant transmise entre les noeuds du graphe, les noeuds finaux générant des objets ActionRequest qui sont remontés à la source de l'appel. Les ActionRequest sont alors regroupés en liste par noeud cible, et chaque liste est envoyée au noeud correspondant dans le cadre d'un thread séparé.*

Au niveau des matériels, nécessité de pouvoir se lier avec différents systèmes de contrôles existants (si l'on veut pouvoir diffuser le logiciel) : TACO (RPC), TANGO (CORBA), ACE (CORBA), EPICS (Socket)... et avec des interfaces en tous genres (série, GPIB - General Purpose Instrument Bus, ...)

Liaisons



Python

-Import direct de modules: `import xxx`

-**PYRO** (Python Remote Object): http://pyro.sourceforge.net/

-**DOPY** (Distributed Object for Python): http://www.users.cloud9.net/~proteus/dopy/welcome.html

Binaires

-**Simple exécution**: Module **`os`**. Fonctions **`exec`**..., **`fork`**, **`spawn`**..., **`startfile`**, **`system`**.

-**Avec redirection / capture des I/O**: Module **`popen2`**.

-Création d'un **module** d'extension Python **spécifique**... si le binaire offre un moyen de communiquer.

C/C++

-Module d'extension "**à la main**".

-**SWIG** (Simplified Wrapper and Interface Generator): à partir du header d'une librairie, génération du code source d'un module d'extension Python. http://www.swig.org/

-Module **`calldll`** sous Windows.

-Extension avec Python: **Embeded Python**

Java

-**Jython**: Interpréteur Python écrit en Java. http://www.jython.org/

-**JPE** (Java Python Extension): accès à C-Python pour Java, accès à Java-Swing pour C-Python. http://sourceforge.net/projects/jpe

Fortran, Delphi...

-**F2PY** (Fortran to Python Interface Generator). http://cens.ioc.ee/projects/f2py2e/

-**Python for Delphy**.http://www.multimania.com/marat/delphi/python.htm

-Extension avec Python (appel de l'API C d'embeded Python).

-Prototoles standards (ex. Python <=> VB avec COM/DCOM).

Protocoles de communication

Ecriture de serveurs et/ou de clients à partir de ces protocoles, indépendance vis à vis du langage de l'autre partie, indépendance vis à vis de la plateforme de l'autre partie (sauf COM & Co).

-**CORBA**: **OmniORBpy** (http://www.uk.research.att.com/omniORB/omniORBpy/index.html), **ILU** (ftp://ftp.parc.xerox.com/pub/ilu/ilu.html), **PyORBit** (http://orbit-python.sault.org/), **Fnorb** (http://www.fnorb.org/).

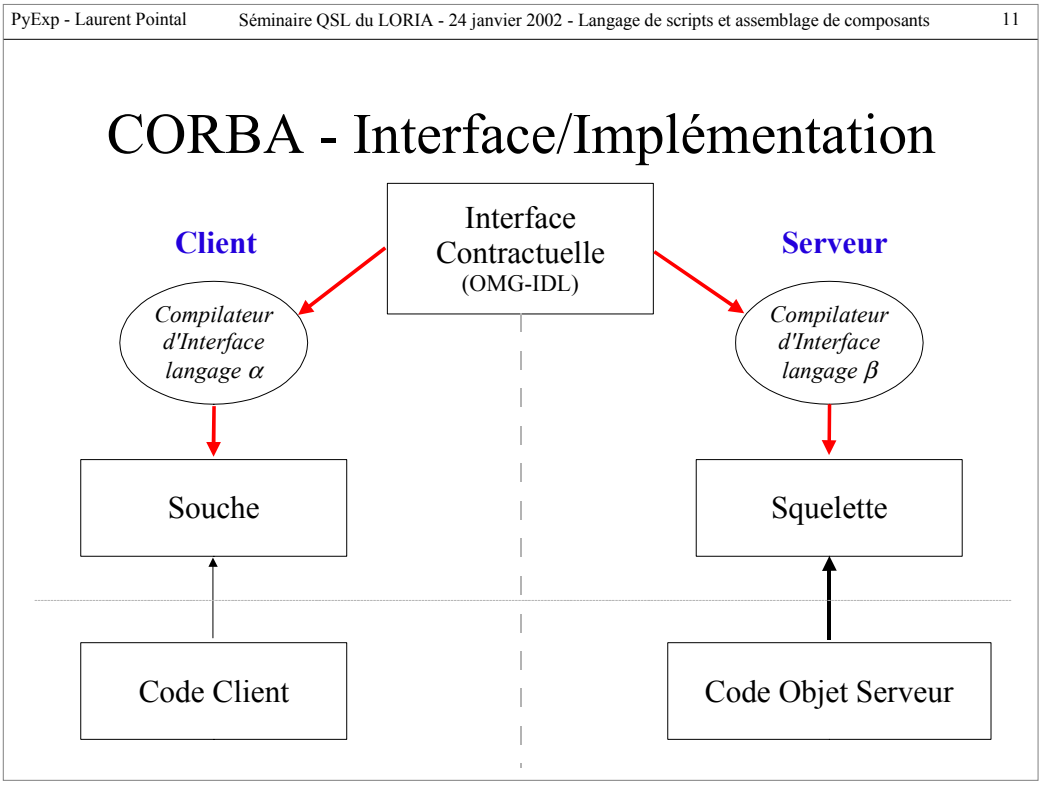
-Windows: **COM / DCOM / Active X**: Extension Win32. http://aspn.activestate.com/ASPN/Python

-**SOAP**: SOAP.py (http://sourceforge.net/projects/pywebsvcs/), SOAPy (http://soapy.sourceforge.net/).

-**XML-RPC**: py-xmlrpc (http://sourceforge.net/projects/py-xmlrpc/), xmlrpclib (http://www.pythonware.com/products/xmlrpc/).

-**email**, **ftp**, **http**...: Librairies standard.

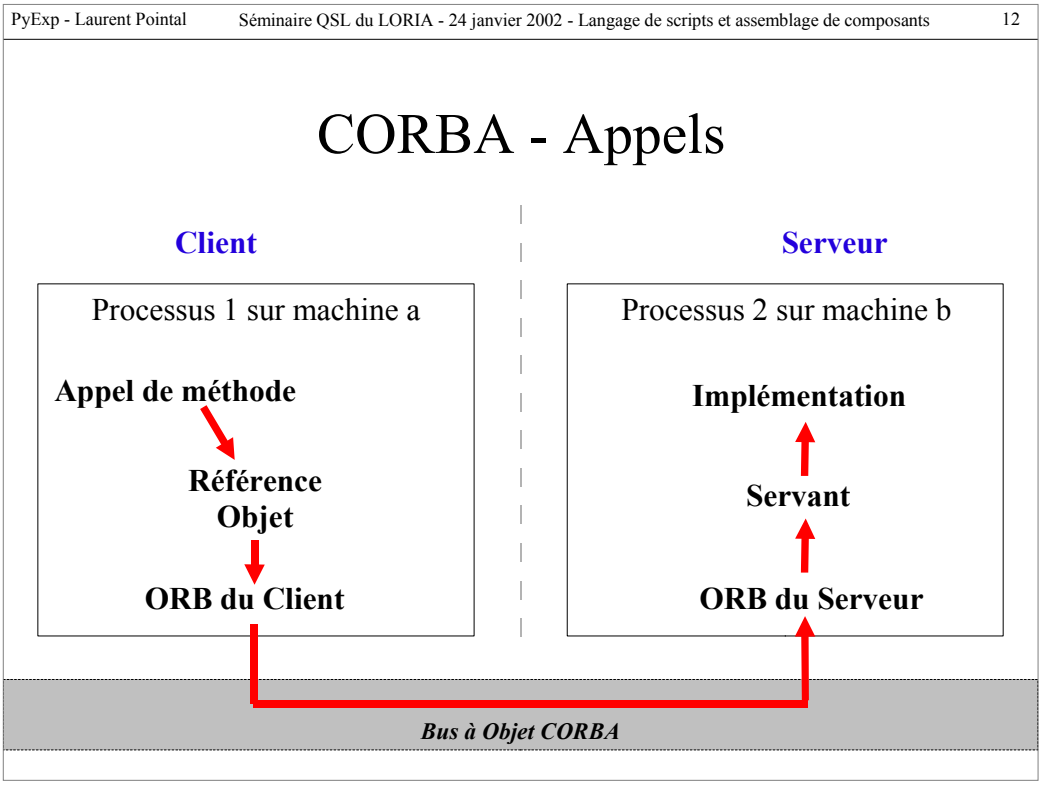
-Autres: Ecriture module d'extension.



CORBA des concepts à la pratique
http://corbaweb.lifl.fr/CORBA_des_concepts_a_la_pratique/index.html

Avec CORBA, l'interface contractuelle, qui permet aux clients et objets serveurs de communiquer, est définie en OMG-IDL (Object Management Group - Interface Definition Langage).

L'utilisation de CORBA avec Python est assez simple. On évite tous les problèmes mémoire (propriété des données) du C++.



Fonctionnement simplifié (manque l'adaptateur d'objet - POA, l'activation...).

Exemple Simple - IDL

```
/* DemoLoria.idl */
#pragma prefix "pointal.org"
module DemoLoria
  {
  interface Demo
    {
    void Afficher (in string phrase) ;
    string Questionner (in string question) ;
    double Sommer (in double v1, in double v2) ;
    } ;
  } ;
```

Compilation de l'IDL: **`omniidl -bpython DemoLoria.idl`**
Génération de: **`DemoLoria/`**, **`DemoLoria__POA/`**, `DemoLoria_idl.py`

On définit un module (espace de noms), ainsi qu'une interface (classe) avec des méthodes. L'interface peut aussi posséder des attributs en lecture et/ou écriture. Il est aussi possible de définir des types et des constantes en IDL.

Code Serveur 1/2

```
import sys
from omniORB import CORBA
import DemoLoria__POA
import CosNaming

class MaDemo_i (DemoLoria__POA.Demo) :
    def Afficher(self, phrase):
        print "Affichage:",phrase
    def Questionner(self, question):
        return raw_input ("Question: %s"%question)
    def Sommer(self, v1,v2):
        print "Somme de",v1,"et",v2
        return v1+v2

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")
```

Liste des modules importés, remarquer CORBA, DemoLoria__POA qui a été généré à partir de l'IDL. CosNaming est le service de noms CORBA.

On définit une classe chargée de la réalisation du service, qui reprend les méthodes définies dans l'interface IDL (il y a un document sur le mapping OMG-IDL ==> Python sur le site CORBA de l'OMG).

Ensuite on initialise l'ORB, et on accès au POA (Portable Object Adapter, chargé de faire le lien entre le côté objet distribué CORBA et l'implémentation).

Code Serveur 2/2

```
demoi = MaDemo_i ()      # Implementation du service
demoo = demoi._this ()   # Objet corba
rootContext = orb.resolve_initial_references
      ("NameService")._narrow(CosNaming.NamingContext)
name = [CosNaming.NameComponent("DemoLoria", "Object")]
try :
    rootContext.bind(name, demoo)
except CosNaming.NamingContext.AlreadyBound:
    rootContext.rebind(name, demoo)
poaManager = poa._get_the_POAManager()
poaManager.activate()
orb.run()
```

On crée une instance de la classe chargée de la réalisation du service, et on accède à son correspondant CORBA (vie _this()).

Ensuite, on accède au service de noms et on y enregistre l'objet CORBA, afin que le client puisse le trouver.

Finalement, on active l'adaptateur d'objet, puis on se bloque en laissant l'ORB traiter les requêtes.

Code Client 1/2

```
import sys
from omniORB import CORBA
import DemoLoria
import CosNaming

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")
rootContext = orb.resolve_initial_references
      ("NameService")._narrow(CosNaming.NamingContext)
name = [CosNaming.NameComponent("DemoLoria", "Object")]
demoo = rootContext.resolve(name)._narrow
                                        (DemoLoria.Demo)
```

Remarquer au niveau de l'import, que pour le client on importe le module DemoLoria qui a été généré à partir de l'IDL.

La phase d'initialisation de l'ORB est la même que pour le serveur.

On se connecte au service de noms CORBA, et cette fois on récupère la référence à l'objet serveur Demo, on transtype cette référence dans le bon type CORBA.

Code Client 2/2

```
demoo.Afficher("Bonjour")
print "Ca marche->",demoo.Questionner
                              ("Est-ce que ca marche?")
v1=float (input ("Valeur 1:"))
v2=float (input ("Valeur 2:"))
print "Somme:",demoo.Sommer(v1,v2)
print "Termine."
```

On utilise l'objet simplement, comme s'il était local.

Démos

- Client/serveur de l'exemple.
- Même client, serveur en C++ avec GUI (MFC).
- Liaison via CORBA d'un script Python avec le logiciel Igor Pro, réalisé pour l'interface de tracé de courbes de PyExp.