Parallélisme

Eric Goubault (et Matthieu Martel)
Commissariat à l'Energie Atomique
Saclay

JavaParty

- Package JAVA développé à l'université de Karlsruhe, voir, `http://www.ipd.uka.de/JavaParty/`,
- Permet de créer directement des classes distantes (`Remote`) sans avoir à gérer la complexité de RMI,
- Est construit au dessus de RMI (ou au dessus de KaRMI, uka.transport, ParaStation - pour une plus grande efficacité),
- Les appels aux méthodes etc. des objets distants sont transparents pour l'utilisateur (comme du JAVA normal),
- Pour améliorer les performances, on peut déclarer des objets "résidents" sur une machine, ou au contraire, faire migrer un objet d'une machine vers une autre.

Il est intéressant de regarder l'implémentation du package.

JavaParty - exemple

```
package examples;

public remote class HelloJP {
  public void hello() {
    // Print on the console of the virtual machine
    // where the object lives
    System.out.println("Hello JavaParty!");
  }
```


Exemple (fin)

```
  public static void main(String[] args) {
    for (int n = 0; n < 10; n++) {
      // Create a remote object on some node
      HelloJP world = new HelloJP();
      // Remotely invoke a method
      world.hello();
    }
  }
}
```

Compilation et exécution

```
> setenv PATH ${PATH}:/usr/local/jp-karmi/bin
> mkdir classes
> jpc -d classes HelloJP.java
generating JavaParty code for class 'HelloJP'
Generating stubs & skeletons for
 examples.HelloJP_instance_impl (skeleton only)
 examples.HelloJP_class_impl
```


Fichiers JAVA engendrés

```
> cd classes
> ls
examples                        HelloJP_class_impl_KStub.java
HelloJP_class_impl_KSkel.java   HelloJP_instance_impl_KSkel.java
> ls examples
HelloJP.class                   HelloJP_instance_impl.class
HelloJP_class_impl.class        HelloJP_instance_impl_KSkel.class
HelloJP_class_impl_KSkel.class  HelloJP_instance_impl_KStub.class
HelloJP_class_impl_KStub.class  HelloJP_instance_intf.class
HelloJP_class_intf.class        HelloJP.signature
```


Description de la machine distribuée

```
> more ~/.jp-nodefile
loire
cher
>
```


Exécution

```
> jp invite HelloJP
Hello...
```

Remarque: `jpc -g` permet d'avoir des messages d'erreur à l'exécution améliorés et `-s` de générer le code JAVA correspondant.

Utilisation de synchronized

Pas plus compliqué qu'en JAVA, même entre objets distants:

```
public remote class R {
  public synchronized void foo() {
    // A synchronized remote method
  }
}
```


Utilisation de synchronized

```
public void foo() {
  R remoteObject = ...;

  synchronized (remoteObject) {
    // A block synchronized
    // on a remote object
  }
}
```


Localité des données

- Les classes non déclarées en `Remote` sont locales à la JVM d'activation
- On peut contrôler l'emplacement réel des classes `Remote` (pour une plus grande efficacité), en utilisant le système de priorité affecté à chaque machine, de la classe (JavaParty) `Distributor`,
- On peut placer manuellement des classes `Remote` sur la machine numéro n de l'environnement distribué comme suit (attention, c'est le commentaire qui est important!):

Localité des données

```
import jp.lang.DistributedRuntime;

public void foo() {
  int cnt = DistributedRuntime.getMachineCnt();
  for (int n = 0; n < cnt; n++) {
    /** @at n */
    R r = new R();
  }
}
```

Migration

- Les objets `Remote` dont aucune méthode n'est en exécution et qui ne sont pas déclarées résidentes (voir transparent suivant),
- Les références à l'objet migré restent valides après migration.

```
boolean success = jp.lang.DistributedRuntime.migrate(obj, n);
```

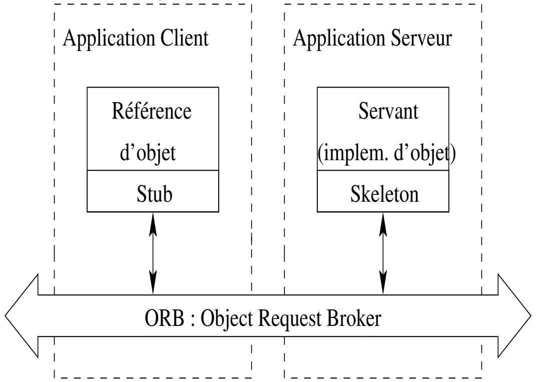

Classes résidentes

```
remote class R implements jp.lang.Resident {
  ...
}
```

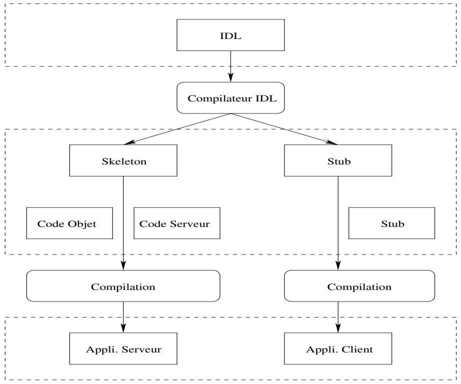
Intérêt: meilleures performances.

CORBA

Le dialogue Corba : Client, Serveur et ORB:



Cycle de développement CORBA



Cycle de développement CORBA

1. *Ecriture de l'interface de l'objet en IDL* : définition dans le langage de description d'interfaces Corba (ou "Interface Definition Language") des opérations disponibles sur l'objet.
2. *Compilation de l'IDL* : génération des modules stub (pour le client) et skeleton (pour le serveur). Ces modules gèrent l'invocation des méthodes distantes.
3. *Implémentation de l'objet* : dérivation d'une classe depuis le skeleton généré à l'étape précédente. Les données et méthodes de cette classe doivent correspondre à celles qui sont décrites dans l'IDL. Lors de l'appel des méthodes sur l'objet, le code défini ici sera exécuté par l'application serveur.

Cycle de développement CORBA

1. *Rédaction de l'application serveur* : ce code sert à initialiser l'objet et à le mettre à disposition des différentes applications clientes.
2. *Compilation du serveur* : génération de l'exécutable de l'application serveur.
3. *Réalisation de l'application cliente* : cette application se connecte à l'objet distant pour lui demander d'exécuter des méthodes. Elle traite ensuite les données transmises par le serveur à l'issue de l'exécution des traitements distants.
4. *Compilation du client* : génération de l'exécutable de l'application client.

Ecriture de l'interface de l'objet en IDL

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


Explication

- principaux types définis par le langage IDL: `short` et `long` pour les entiers, `float` et `double` pour les flottants, `char` et `string` pour les caractères.
- types composés : tableaux ainsi que d'autres structures à l'aide des mots clés `enum`, `struct` ou `union`, etc.
- `attribute` est un mot clé du langage IDL, tout comme `in` qui indique le sens de passage du paramètre (du client vers le serveur).
- Symétriquement le type de passage des paramètres `out` permet de réaliser un passage par adresse.
- `void maxElement(out double m);`. Le type `inout` peut être utilisé pour les paramètres devant être lus et écrits.

Compilation de l'IDL

Compiler l'interface par la commande `idlj -fall Vecteur.idl`. Les principaux fichiers qui ont été générés sont les suivants :

- `VecteurOperations.java` : définition de l'interface qui contient la signature des méthodes de l'objet `Vecteur`.
- `Vecteur.java` : définition de l'interface `Vecteur` qui hérite de `VecteurOperations`.
- `VecteurHelper.java` : regroupe des méthodes destinées à vous aider lors de l'utilisation des objets `Vecteur`.

Compilation de l'IDL

- `VecteurHolder.java` : outils pour prendre en charge le passage de paramètres avec les méthodes de l'objet `Vecteur`.
- `_VecteurStub.java` : stub de l'objet `Vecteur`. Cette classe sera utilisée dans le code de notre client de façon implicite, elle représentera l'objet distant en mettant à notre disposition les méthodes qui s'y trouvent.
- `_VecteurImplBase.java` : squelette de l'objet `Vecteur`. C'est la classe de base de notre futur objet `vecteur`. On en dérivera une classe pour coder le fonctionnement réel de notre objet.

Implémentation de l'objet

Créer la classe de l'objet que nous souhaitons mettre à disposition: pour cela, on utilise le squelette généré précédemment:

- la classe `VecteurImplem` que nous allons créer étendra la classe `_VecteurImplBase` produite par le compilateur IDL
- et implémentera les méthodes définies par l'interface `VecteurOperations.java`.

Implémentation de l'objet

```
public class VecteurImplem extends _VecteurImplBase {
    private int size=100;
    private double T[];
    VecteurImplem(double[] Tab) {
        for(int i=0;i<size;i++) {
            T[i]=Tab[i]; } }

    VecteurImplem() {} ;

    public int size() {
        return size; }
```

Implémentation de l'objet

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

public double[] T() {
    return T; }

public void T(double[] newT) {
    for(int i=0;i<size;i++) {
        T[i]=newT[i]; } }
```


Implémentation de l'objet

```
public double maxElement () {
    double m = T[0];
    for (int i=1;i<size;i++) {
        if (T[i]>m) { m = T[i]; } };
    return m; }

public void mult(double a) {
    for (int i=0;i<size;i++) {
        T[i]=T[i]*a; } } }
```


Implémentation de l'objet

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

Serveur CORBA

Nous allons créer une application hôte, un serveur qui contiendra l'objet `Vecteur`:

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

public class VecteurServeur {
    public static void main(String args[]) {
        try{
```

Serveur CORBA

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


Serveur CORBA

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


Serveur CORBA

Les principales actions réalisées par le serveur sont :

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

voir `http://java.sun.com/products/jdk/1.2/docs/guide/idl/tutorial/GSserver.html`.

Réalisation du client

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

Réalisation du client

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

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


Réalisation du client

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


Explication

- Le client initialise l'ORB et obtient une référence au contexte de nommage.
- Il y recherche ensuite “Vecteur” et reçoit une référence de cet objet Corba.
- Il peut ensuite invoquer les méthodes de l'objet.

Voir `http://java.sun.com/pro ducts/jdk/1.2/docs/guide/idl/tutorial/GSapp.html`.

Compilation

```
javac *.java
```

Exécution

Il faut procéder dans l'ordre suivant :

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

  `java VecteurServer -ORBInitialHost namerserverhost -ORBInitialPort 1050`.

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

  `java VecteurClient -ORBInitialHost namerserverhost -ORBInitialPort 1050`