Parallélisme

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

Remote Method Invocation

- Permet d'invoquer des méthodes d'un objet distant, c'est à dire appartenant à une autre JVM, sur une autre machine
- Architecture de type client/serveur; similaire aux "Remote Procedure Calls" POSIX
- Se rapproche de plus en plus de CORBA (langage indépendant etc., voir cours suivant)

<u>Références</u>: JAVA, Network Programming and Distributed Computing, D. Reilly et M. Reilly, Addison-Wesley.
et `http://java.sun.com/products/jdk/rmi/`

Architecture

- Classe qui implémente la méthode distante (serveur):
  - dont les méthodes renvoient un objet `serializable`
  - ou plus généralement une classe `remote`
- Client qui utilise les méthodes distantes
- Registre d'objets distants qui associe aux noms d'objects l'adresse des machines qui les contiennent

Les classes implémentant `Serializable`

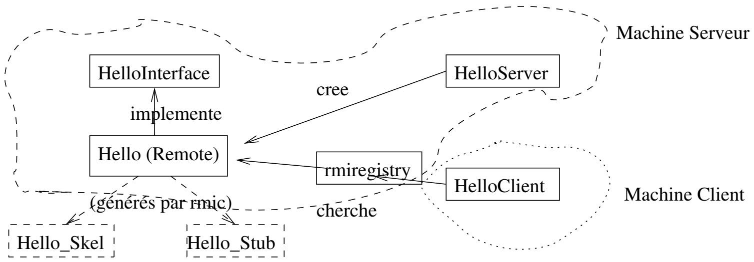
- leurs instances peuvent être envoyées d'une machine à une autre
- la plupart des classes (et leurs sous-classes) de base `String` etc. sont `Serializable`
- dans le cas où on passe une classe `Serializable`, il faut que la définition de cette classe soit connue ($\Rightarrow$ copiée sur les différentes machines) des clients et du serveur
- il peut y avoir à gérer la politique de sécurité (sauf pour les objets "simples", comme `String` etc. voir plus loin).

Les classes remote

- leurs instances sont des objets ordinaires dans l'espace d'adressage de leur JVM
- des "pointeurs" sur ces objets peuvent être envoyés aux autres espaces d'adressage

Exemple - on retourne un Serializable

"Hello World" distribué: on va construire les classes:



Interface de l'objet distant

```
import java.rmi.*;

public interface HelloInterface extends Remote {
    public String say() throws RemoteException;
}
```


Hello World: implémentation de l'objet distant

```
import java.rmi.*;
import java.rmi.server.*;

public class Hello extends UnicastRemoteObject
  implements HelloInterface {
  private String message;

  public Hello(String msg) throws RemoteException {
    message = msg; }
```

Hello World: implémentation de l'objet distant

```
  public String say() throws RemoteException {
    return message;
  }
}
```


Compilation

```
javac HelloInterface.java
javac Hello.java
```

(crée `HelloInterface.class` et `Hello.class`)
Création des stubs et squelettes:

```
rmic Hello
```

(crée `Hello_Stub.class` et `Hello_Skel.class`)

Client

```
import java.rmi.*;
public class HelloClient {
  public static void main(String[] argv) {
    try {
      HelloInterface hello =
      (HelloInterface) Naming.lookup
              ("//cher.polytechnique.fr/Hello");
      System.out.println(hello.say());
```

(le serveur est supposé toujours être sur `cher`, voir plus loin pour d'autres méthodes)

Client

```
    } catch(Exception e) {
        System.out.println("HelloClient exception: "+e);
    }
  }
}
```

Serveur

```
import java.rmi.*;

public class HelloServer {
  public static void main(String[] argv) {
    try {
      Naming.rebind("Hello",new Hello("Hello, world!"));
      System.out.println("Hello Server is ready.");
```


Serveur

```
    } catch(Exception e) {
        System.out.println("Hello Server failed: "+e);
    }
  }
}
```


Compilation et démarrage du serveur

Faire attention au `CLASSPATH` (doit au moins contenir `.` et/ou les répertoires contenant les `.class` nécessaires, accessibles de toutes les machines sous NFS)

```
javac HelloClient.java
javac HelloServer.java
```

Démarrer le serveur de noms:

```
rmiregistry &
```

(attendre un minimum)

Compilation et démarrage du serveur

Démarrer le serveur (Hello):

```
java HelloServer &
```

(attendre un peu)

Démarrage des clients et exécution

(ici en local)

```
> Hello Server is ready.
> java HelloClient
Hello, world!
```


Installation locale aux salles de TD

- `setenv PATH $PATH:/usr/java/j2sdk1.4.0_01/bin` (pour `rmiregistry` et `rmid`),
- démarrer `rmiregistry` dans le répertoire local où se trouvent les `.class` (ou vérifier le `CLASSPATH`).

Execution en salle TD

(récupérer les programmes java dans `~goubaul1/Cours03/RMI/*`)
Exemple: serveur sur cher,

```
[goubaul1@cher HelloNormal]$ java HelloServer
Hello Server is ready.
```

Client sur loire,

```
[goubaul1@loire HelloNormal]$ java HelloClient
Hello, world!
```


Callback

L'idée est la suivante (programmation "événementielle", typique d'interfaces graphique par exemple AWT):

- les "clients" vont s'enregistrer auprès d'un serveur,
- le "serveur" va les "rappeler" uniquement lorsque certains événements se produisent,
- le client n'a pas ainsi à faire de "l'active polling" (c'est à dire à demander des nouvelles continuellement au serveur) pour être mis au fait des événements.

Principe du "rappel"

Comment notifier un objet (distant) de l'apparition d'un événement?

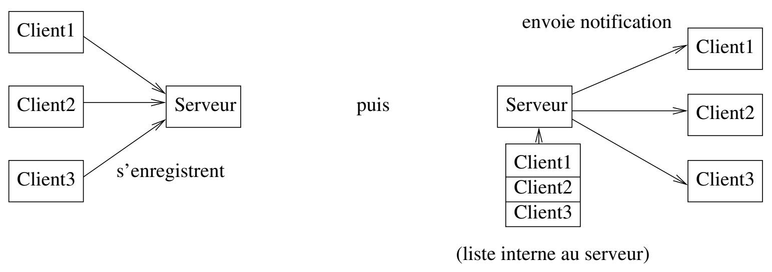
- on passe la référence de l'objet à rappeler, au serveur chargé de suivre (ou source des) les événements,
- à l'apparition de l'événement, le serveur va invoquer la méthode de notification du client.

Ainsi,

- pour chaque type d'événement, on crée une interface spécifique (pour le client qui veut en être notifié),
- les clients potentiels à notifier doivent s'enregistrer auprès d'une implémentation de cette interface.

Cela implique que "clients" et "serveurs" sont tous à leur tour "serveurs" et "clients".

C'est à dire...



Exemple - interface associée à un évévenement

... ici, changement de température:

```
interface TemperatureListener extends java.rmi.Remote {
    public void temperatureChanged(double temperature)
         throws java.rmi.RemoteException;
}
```

C'est la méthode de notification de tout client intéressé par cet événement. Forcément un objet `Remote`.

Exemple - l'interface du serveur d'événements

... doit au moins pouvoir permettre l'inscription et la désinscription de clients voulant être notifié:

```
interface TemperatureSensor extends java.rmi.Remote {
    public double getTemperature() throws
        java.rmi.RemoteException;
    public void addTemperatureListener
        (TemperatureListener listener)
        throws java.rmi.RemoteException;
    public void removeTemperatureListener
        (TemperatureListener listener)
        throws java.rmi.RemoteException; }
```

Exemple - l'implémentation du serveur

- doit être une sous-classe de `UnicastRemoteObject` (pour être un serveur...).
- doit implémenter l'interface `TemperatureListener` pour pouvoir rappeler les clients en attente,
- implémente également `Runnable` ici pour pouvoir avoir un thread indépendant qui simule les changements de température.

Exemple - implémentation serveur

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class TemperatureSensorServer
    extends UnicastRemoteObject
    implements TemperatureSensor, Runnable {
    private volatile double temp;
    private Vector list = new Vector();
```

(le vecteur `list` contiendra la liste des clients)

Exemple - implémentation serveur

Constructeur (température initiale) et méthode de récupération de la température:

```
public TemperatureSensorServer()
  throws java.rmi.RemoteException {
    temp = 98.0; }

public double getTemperature()
  throws java.rmi.RemoteException {
    return temp; }
```


Exemple - implémentation serveur

Méthodes d'ajout et de retrait de clients:

```
public void addTemperatureListener
              (TemperatureListener listener)
  throws java.rmi.RemoteException {
  System.out.println("adding listener -"+listener);
  list.add(listener); }

public void removeTemperatureListener
              (TemperatureListener listener)
  throws java.rmi.RemoteException {
  System.out.println("removing listener -"+listener);
  list.remove(listener); }
```

Exemple - implémentation serveur

Thread responsable du changement aléatoire de la température:

```
public void run()
{ Random r = new Random();
  for (;;)
    { try {
        int duration = r.nextInt() % 10000 +2000;
        if (duration < 0) duration = duration*(-1);
        Thread.sleep(duration); }
```


Exemple - implémentation serveur

```
      catch(InterruptedException ie) {}
      int num = r.nextInt();
      if (num < 0)
        temp += .5;
      else
        temp -= .5;
      notifyListeners(); } }
```

(`notifyListeners()` est la méthode suivante, chargée de broadcaster le changement d'événements à tous les clients enregistrés)

Exemple - implémentation du serveur

```
private void notifyListeners() {
  for (Enumeration e = list.elements(); e.hasMoreElements();)
  { TemperatureListener listener =
        (TemperatureListener) e.nextElement();
    try {
      listener.temperatureChanged(temp);
    } catch(RemoteException re) {
        System.out.println("removing listener -"+listener);
        list.remove(listener); } } }
```

(on fait simplement appel, pour chaque client, à la méthode de notification `temperatureChanged`)

Exemple - implémentation du serveur

Enregistrement du service auprès du `rmiregistry` (éventuellement fourni à la ligne de commande):

```
public static void main(String args[]) {
  System.out.println("Loading temperature service");
  try {
    TemperatureSensorServer sensor =
        new TemperatureSensorServer();
    String registry = "localhost";
    if (args.length >= 1)
      registry = args[0];
    String registration = "rmi://"+registry+
                          "/TemperatureSensor";
    Naming.rebind(registration,sensor);
```

Exemple

Démarrage du thread en charge de changer aléatoirement la température, et gestion des exceptions:

```
    Thread thread = new Thread(sensor);
    thread.start(); }
  catch (RemoteException re) {
    System.err.println("Remote Error - "+re); }
  catch (Exception e) {
    System.err.println("Error - "+e); } } }
```


Exemple - implémentation clients

```
import java.rmi.*;
import java.rmi.server.*;

public class TemperatureMonitor extends UnicastRemoteObject
    implements TemperatureListener {
    public TemperatureMonitor() throws RemoteException {}
```

(étend `UnicastRemoteObject` car serveur également! De même implémente `TemperatureListener`) Rq: constructeur vide (celui d'`Object` en fait).

Exemple - implémentation clients

Recherche du service serveur d'événements:

```
public static void main(String args[]) {
  System.out.println("Looking for temperature sensor");
  try {
    String registry = "localhost";
    if (args.length >= 1)
    registry = args[0];
    String registration = "rmi://"+registry+
                          "/TemperatureSensor";
    Remote remoteService = Naming.lookup(registration);
    TemperatureSensor sensor = (TemperatureSensor)
                               remoteService;
```


Exemple - implémentation clients

Création d'un moniteur et enregistrement auprès du serveur d'événements:

```
    double reading = sensor.getTemperature();
    System.out.println("Original temp : "+reading);
    TemperatureMonitor monitor = new TemperatureMonitor();
    sensor.addTemperatureListener(monitor);
```

Exemple - implémentation clients

Gestion des exceptions:

```
} catch(NotBoundException nbe) {
    System.out.println("No sensors available"); }
  catch (RemoteException re) {
    System.out.println("RMI Error - "+re); }
  catch (Exception e) {
    System.out.println("Error - "+e); } }
```


Exemple - implémentation clients

Implémentation de la méthode de rappel:

```
public void temperatureChanged(double temperature)
  throws java.rmi.RemoteException {
  System.out.println("Temperature change event : "
                     +temperature);
}
```


Compilation

```
[goubaul1@cher Ex3]$ javac *.java
[goubaul1@cher Ex3]$ rmic TemperatureMonitor
[goubaul1@cher Ex3]$ rmic TemperatureSensorServer
```


Exécution

```
[goubaul1@cher Ex3]$ rmiregistry &
[goubaul1@cher Ex3]$ java TemperatureSensorServer
Loading temperature service
```

Premier client (sur loire):

```
[goubaul1@loire Ex3]$ rmiregistry &
[goubaul1@loire Ex3]$ java TemperatureMonitor cher
Looking for temperature sensor
Original temp : 100.0
Temperature change event : 99.5
Temperature change event : 100.0
Temperature change event : 100.5
```

Exécution

```
Temperature change event : 100.0
Temperature change event : 100.5
Temperature change event : 101.0
Temperature change event : 100.5
Temperature change event : 100.0
Temperature change event : 100.5
Temperature change event : 101.0
Temperature change event : 101.5
```


Exécution

On voit alors sur la console de cher:

```
adding listener -TemperatureMonitor_Stub[RemoteStub
[ref: [endpoint:[129.104.254.64:3224](remote),
objID:[6e1408:f29e197d47:-8000, 0]]]]
```

Rajoutons un moniteur sur doubs:

```
[goubaul1@doubs Ex3]$ rmiregistry &
[goubaul1@doubs Ex3]$ java TemperatureMonitor cher
Looking for temperature sensor
Original temp : 101.5
Temperature change event : 102.0
```


Exécution

```
Temperature change event : 102.5
Temperature change event : 103.0
Temperature change event : 102.5
Temperature change event : 103.0
Temperature change event : 103.5
Temperature change event : 102.5
Temperature change event : 102.0
```


Exécution

Ce qui produit sur cher:

```
adding listener -TemperatureMonitor_Stub[RemoteStub
[ref: [endpoint:[129.104.254.57:3648](remote),
objID:[6e1408:f29de7882e:-8000, 0]]]]
```

On voit bien que les températures et événements sont synchronisés avec l'autre client sur loire:

```
Temperature change event : 102.0
```

Exécution

```
Temperature change event : 102.5
Temperature change event : 103.0
Temperature change event : 102.5
Temperature change event : 103.0
Temperature change event : 103.5
^C
[goubaul1@loire Ex3]$
```


Exécution

On a interrompu sur loire, du coup sur cher:

```
removing listener -TemperatureMonitor_Stub[RemoteStub
[ref: [endpoint:[129.104.254.64:3224](remote),
objID:[6e1408:f29e197d47:-8000, 0]]]]
```

On interrompt par Control-C sur doubs, du coup sur cher:

```
removing listener -TemperatureMonitor_Stub[RemoteStub
[ref: [endpoint:[129.104.254.57:3648](remote),
objID:[6e1408:f29de7882e:-8000, 0]]]]
```


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

```
> mkdir classes
> jpc -d classes HelloJP.java
> jpinvite HelloJP
Hello...
```

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

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.