

INF 560
Calcul Parallèle et Distribué
Cours 5

Eric Goubault

CEA, LIST & Ecole Polytechnique

10 février 2014

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)

Références: JAVA, Network Programming and Distributed Computing, D. Reilly et M. Reilly, Addison-Wesley.

- Classe qui implémente la méthode distante (serveur):
 - dont les méthodes renvoient ou recoivent des objets `Serializable` (sémantique par copie)
 - ou des objets appartenant à des classes `Remote` (sémantique par référence)
 - méthodes doivent pouvoir lancer l'exception `RemoteException`
- Client qui utilise les méthodes distantes
- Registre d'objets distants qui associe aux noms d'objets l'adresse des machines qui les contiennent

LES CLASSES IMPLÉMENTANT Serializable

- Objets instances peuvent être transcrits en “stream”, c'est-à-dire en flots d'octets.

- `writeObject(ObjectOutputStream aOutputStream)`
`readObject(ObjectInputStream aInputStream)`

responsables respectivement de décrire un objet sous forme de flot d'octets et de reconstituer l'état d'un objet à partir d'un flot d'octets.

- La plupart des classes (et de leurs sous-classes) de base `String`, `HashTable`, `Vector`, `HashSet`, `ArrayList`...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.).

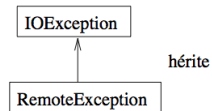
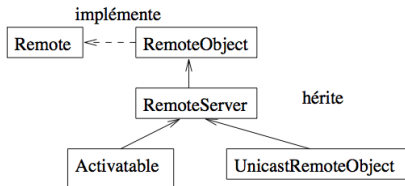
- leurs instances sont des objets ordinaires dans l'espace d'adressage de leur JVM
- des “pointeurs” sur ces objets peuvent être envoyés aux autres espaces d'adressage

POUR EN SAVOIR PLUS: PACKAGES RMI

- `java.rmi` définit l'interface `RemoteInterface`, et les exceptions,
- `java.rmi.activation` (depuis JAVA2): permet l'activation à distance des objets,
- `java.rmi.dgc`: s'occupe du ramassage de miettes dans un environnement distribué,
- `java.rmi.registry` fournit l'interface permettant de représenter un `rmiregistry`, d'en créer un, ou d'en trouver un,
- `java.rmi.server` fournit les classes et interfaces pour les serveurs RMI.

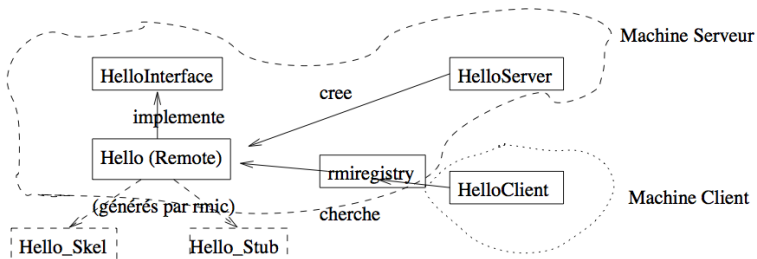
<http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html>

LES DIFFÉRENTES CLASSES ET INTERFACES



EXEMPLE - ON RETOURNE UN Serializable

“Hello World” distribué: on va construire les classes:



Un stub est une sorte de proxy pour le client (cache la sérialization et les communications bas niveau sur le réseau):

- c'est lui qui initie la connection avec la JVM distante contenant l'objet distant
- envoie les arguments à l'objet distant ("marshals")
- attend les résultats
- récupère les résultats ("unmarshals")

Un squelette est responsable (sur la demande du stub correspondant) d'appeler la méthode sur le serveur:

- lit les paramètres (“unmarshals”)
- appelle la méthode de l'objet serveur correspondant
- écrit les paramètres sur le réseau (“marshals”)

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; }

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

COMPILATION

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

(crée HelloInterface.class et Hello.class)

```
import java.rmi.*;
public class HelloClient {
    public static void main(String[] argv) {
        try {
            HelloInterface hello =
                (HelloInterface) Naming.lookup
                    ("//cher.polytechnique.fr/Service");
            System.out.println(hello.say());
        } catch (Exception e) {
            System.out.println("HelloClient_exception:_" + e);
        }
    }
}
```

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

```
import java.rmi.*;

public class HelloServer {
    public static void main(String[] argv) {
        try {
            Naming.rebind("Service",new Hello("Hello ,_world!"));
            System.out.println("Hello_Server_is_ready.");
        } catch(Exception e) {
            System.out.println("Hello_Server_failed :_" +e);
        }
    }
}
```

COMPILATION ET DÉMARRAGE DU SERVEUR

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

Démarrer le serveur de noms:

```
rmiregistry &
```

(attendre un minimum)

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!
```

- `rmiregistry` doit être démarré avec un numéro de port distinct pour plusieurs utilisateurs sur une même machine (numéros à partir de 1099), voir répartition sur fiche TD.
Exemple sur machine serveur:

```
rmiregistry 1100 &
```

- Dans ce cas, le serveur devra s'enregistrer par

```
Naming.rebind("rmi://localhost:1100/Service");
```

Et le client devra chercher sur le même port:

```
Naming.lookup("rmi://cher.polytechnique.fr:1100/Service");
```

(récupérer les programmes java sur la page web)

Exemple: serveur sur cher,

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

Client sur loire,

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

AUTRE MÉTHODE POUR LE SERVEUR

```
import java.rmi.*;
import java.rmi.registry.*;

public class HelloServer {
    public static void main(String[] argv) {
        try {
            LocateRegistry.createRegistry(1100);
            Registry r = LocateRegistry.getRegistry("oncidium",1100);
            Naming.rebind("//oncidium:1100/Service",new Hello("Hi!"));
            String[] name = r.list();

            for (int i=0; i<name.length; i++)
                System.out.println(name[i]);

            System.out.println("Hello_Server_is_ready.");
        } catch (Exception e) {
            System.out.println("Hello_Server_failed:_" + e);
        }
    }
}
```

- `LocateRegistry.createRegistry(port number)` crée le démon `rmiregistry` écoutant sur le port `port number`
- `LocateRegistry.getRegistry(server name, port number)` essaie de trouver un démon `rmiregistry` sur la machine `server name` et sur le port `port number`
- Cela donne un objet manipulable... par exemple, on parcourt la liste des services connus du `rmiregistry` par la boucle sur `r.list()`

OU ENCORE... (ACCÈS AU RMIREGISTRY)

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

public class HelloServer {
    public static void main(String[] argv) {
        try {
            LocateRegistry.createRegistry(1100);
            Registry r = LocateRegistry.getRegistry("localhost",1100);
            Hello obj = new Hello("Hello, _world!");
            r.bind("Service", obj);
            System.out.println("Hello_Server_is_ready.");
        } catch (Exception e) {
            System.out.println("Hello_Server_failed:_"+e);
        }
    }
}
```

OU ENCORE... (ACCÈS DIRECT AU STUB CRÉE DYNAMIQUEMENT)

(...)

```
public class HelloServer {  
    public static void main(String[] argv) {  
try {  
        LocateRegistry.createRegistry(1100);  
        Registry r = LocateRegistry.getRegistry("localhost",1100);  
        Hello obj = new Hello("Hello ,_world!");  
        HelloInterface stub=(HelloInterface)UnicastRemoteObject.  
                                exportObject(obj,1100);  
                                r.bind("Service", stub);  
        System.out.println("Hello_Server_is_ready.");  
    } catch (Exception e) {  
        System.out.println("Failed :_" +e);  
    }  
}
```

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

public class Hello implements HelloInterface , Remote {
    private String message;

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

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

ET POUR LE CLIENT: (ACCÈS AU RMIREGISTRY)

```
import java.rmi.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class HelloClient {
    public static void main(String[] argv) {
    try {
        Registry registry = LocateRegistry.getRegistry("localhost", 1100);
        HelloInterface hello = (HelloInterface) registry.lookup("Service");
        System.out.println(hello.say());
    } catch (Exception e) {
        System.out.println("HelloClient_exception: " + e);
    }
    }
```

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

- 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.

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

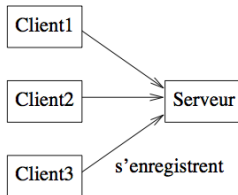
- on passe la référence de l'objet à rappeler, au serveur chargé de suivre (ou source des) les événements,
- à l'apparition de l'événement, le serveur va invoquer la méthode de notification du client.

Ainsi,

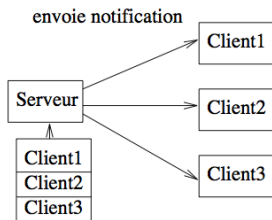
- pour chaque type d'événement, on crée une interface spécifique (pour le client qui veut en être notifié),
- les clients potentiels à notifier doivent s'enregistrer auprès d'une implémentation de cette interface.

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

C'EST À DIRE...



puis



(liste interne au serveur)

EXEMPLE - INTERFACE ASSOCIÉE À UN ÉVÉNEMENT

... 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 <TemperatureListener> list =
        new Vector <TemperatureListener> ();

    static final long serialVersionUID = 42L;
```

(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); }
    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);  
    }  
}
```

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).

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);
}
```

EXÉCUTION

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

```
[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

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  
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
```

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  
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]$
```

On a interrompu sur loire, du coup sur cher:

```
removing listener -TemperatureMonitor_Stub[RemoteStub  
[ref: [endpoint:[129.104.254.64:3224](remote),  
objID:[6 e1408: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:[6 e1408:f29de7882e:-8000, 0]]]]
```