Java RMI

Arnaud Labourel
Courriel: arnaud.labourel@lif.univ-mrs.fr

Université de Provence

8 mars 2011

Web services

La toile (world wide web $\neq$ internet) c'est, grossièrement l'ensemble des informations disponibles sur le port 80 via le protocole HTTP. Pour des raisons informationnelles et aussi pratiques, un certain nombre d'extensions se sont aussi mises en place sur le port 80.

Problématique de la Présentation des Données

Données au sens large :

- pages html statiques
- réponse à une requête dans une base de données
- structures de données dans un programme
- ...

Interfaces

Une solution portable : dialogue par l'intermédiaire d'interfaces

- Unix et RPC (Remote procedure call)
- Interface Java pour RMI
  (remote method invocation)
- Interface IDL (Interface Definition Language) pour Corba
- Descriptions en XML pour les web services

Systèmes Distribués et Programmation Orientée Objet

Principe

- Systèmes distribués et hétérogénes
- Communication de haut-niveau
- Programmation objet et réseau
- Manipuler des ‘objets distants” comme des objets standard
  - variables
  - paramètres de fonctions
  - valeur de retour

Architectures Distribuées

- RPC : Remote Procedure Call
- Corba (OMG) multi-langage
- Java RMI
- J2EE (Sun)
- DCOM (Microsoft) multi-langage
- .NET (Microsoft)
- Intergiciel
  - WebLogic
  - ObjectWeb
  - ...

Corba

Common Object Request Broker Architecture : Corba

Définition

Architecture logicielle, pour le développement de composants et d'Object Request Broker ou ORB.

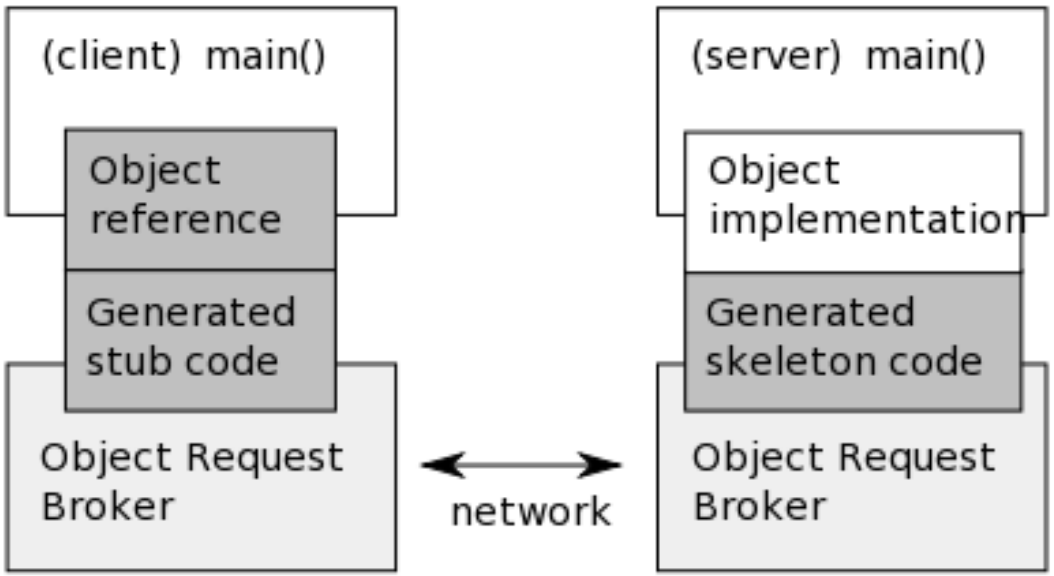
ORB : intergiciel permettant les échanges de messages entre objets.

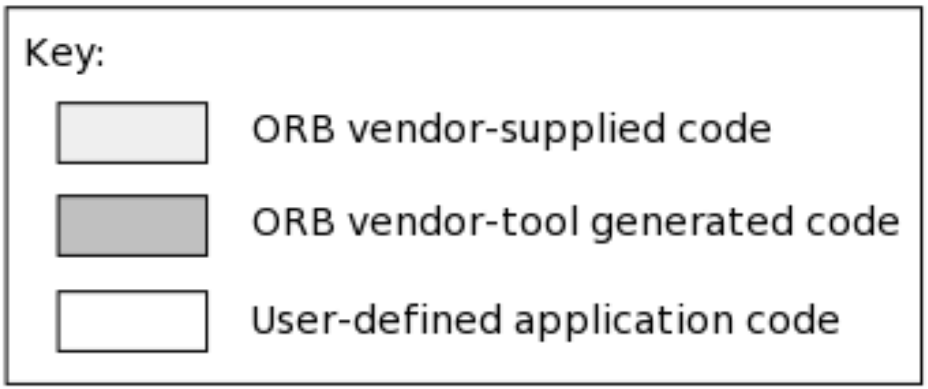
Mis en place par le consortium OMG (Object Management Group).

Interopérable avec Java RMI

Approche essentiellement orientée objet

Fonctionnement de corba





IDL : Interface Definition Language (I)

Description des interfaces distantes dans une syntaxe indépendante de tout langage de programmation.
Exemple:

```
// Contrat OMG IDL d'une application d'annuaire.
#include <date.idl>        // Réutilisation du service de dates.
module annuaire {
  typedef string Nom;                        // Nom d'une personne.
  typedef sequence<Nom> DesNoms;             // Ensemble de noms.
  struct Personne {                  // Description d'une personne.
     Nom nom;                        //       - son nom.
     string informations;            //       - données diverses.
     string telephone;               //       - numéro de téléphone.
     string email;                   //       - son adresse Email.
     string url;                     //       - son adresse WWW.
     ::date::Date date_naissance;
  };
```

IDL (II)

```
typedef sequence<Personne> DesPersonnes;

interface Repertoire {
  readonly attribute string libelle;
  exception ExisteDeja { Nom nom; };
  exception Inconnu { Nom nom; };
  void ajouterPersonne (in Personne personne)
                                          raises(ExisteDeja);
  void retirerPersonne (in Nom nom)       raises(Inconnu);
  void modifierPersonne (in Nom nom, in Personne p)
                                          raises(Inconnu);
  Personne obtenirPersonne (in Nom nom)   raises(Inconnu);
  DesNoms listerNoms ();
};
```

Langages Utilisés pour Corba

Les langages actuellement supportés :

- C (très dur)
- C++ (difficile)
- Java (facile)
- Smalltalk
- Ada,
- COBOL,

En java, il faut utiliser les packages `org.omg.CORBA` et `org.omg.CosNaming`.

Java RMI

Définition

Interface de programmation (API) pour le langage Java qui permet d'appeler des méthodes distantes.

Remote Method Invocation :

- Intégré au JDK
- Système distribué simplifié
- Uniquement pour des objets Java
- un objet distant (OD) est bien manipulable comme un objet standard :
  - variable
  - paramètre de fonctions
  - valeur de retour

Architecture Java RMI

Le fonctionnement repose initialement sur le modèle client/serveur :

- Le serveur présente des objets décrits par une *interface*
- Le *serveur* s'enregistre auprès d'un *annuaire* (registry)
- Chaque *client* obtient une référence distante auprès de l'annuaire.
- La communication haut-niveau s'appuie sur la communication des objets Java sous-jacents de JVM à JVM.

Classes implémentant `Serializable`

Comme les arguments d'appels de fonction vont être transmis par valeurs, il faut pouvoir transmettre une classe sous la forme d'une communication réseau, c'est-à-dire sous une forme "aplatie" (sérialisée).

- Les classes distantes doivent être `Serializable`,
- Transformation (automatique) des objets :
  - *pliage (marshalling)* avant transmission
  - *dépliage (unmarshalling)* après réception

L'interface Serializable

```
//Ã©crit un objet dans le stream out (sauvegarde)

private void writeObject(java.io.ObjectOutputStream out)
     throws IOException

// restaure l'objet Ã  son Ã©tat lors du dernier appel
de writeObject

private void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException;

// mÃ©thode appelÃ©e par read object lorsqu'il ne trouve
// pas d'instance de l'objet avec readObject

private void readObjectNoData()
     throws ObjectStreamException;
```

L'"interface `Remote`

Tous les objets distants héritent de l'interface `Remote`.

Les méthodes (distantes) définies dans une classe héritant de remote peuvent être appelées depuis une machine virtuelle distante.

Toute "méthode distante" peut engendrer l'exception `RemoteException` :

- serveur non trouvé
- connexion interrompue
- ...

La Classe `Naming`

Définition

Cette classe permet de manipuler des références sur des objets distants dans un registre d'objet distant (serveur)

Les `noms` qui sont des urls de la forme `//hote:port/nom`

- `hote` : nom du registre (serveur) contenant l'objet distant (optionnel : `localhost` par défaut)
- `port` : numéro de port sur lequel le registre accepte les requêtes (optionnel : `1099` par défaut)
- `name` : nom associé à l'objet

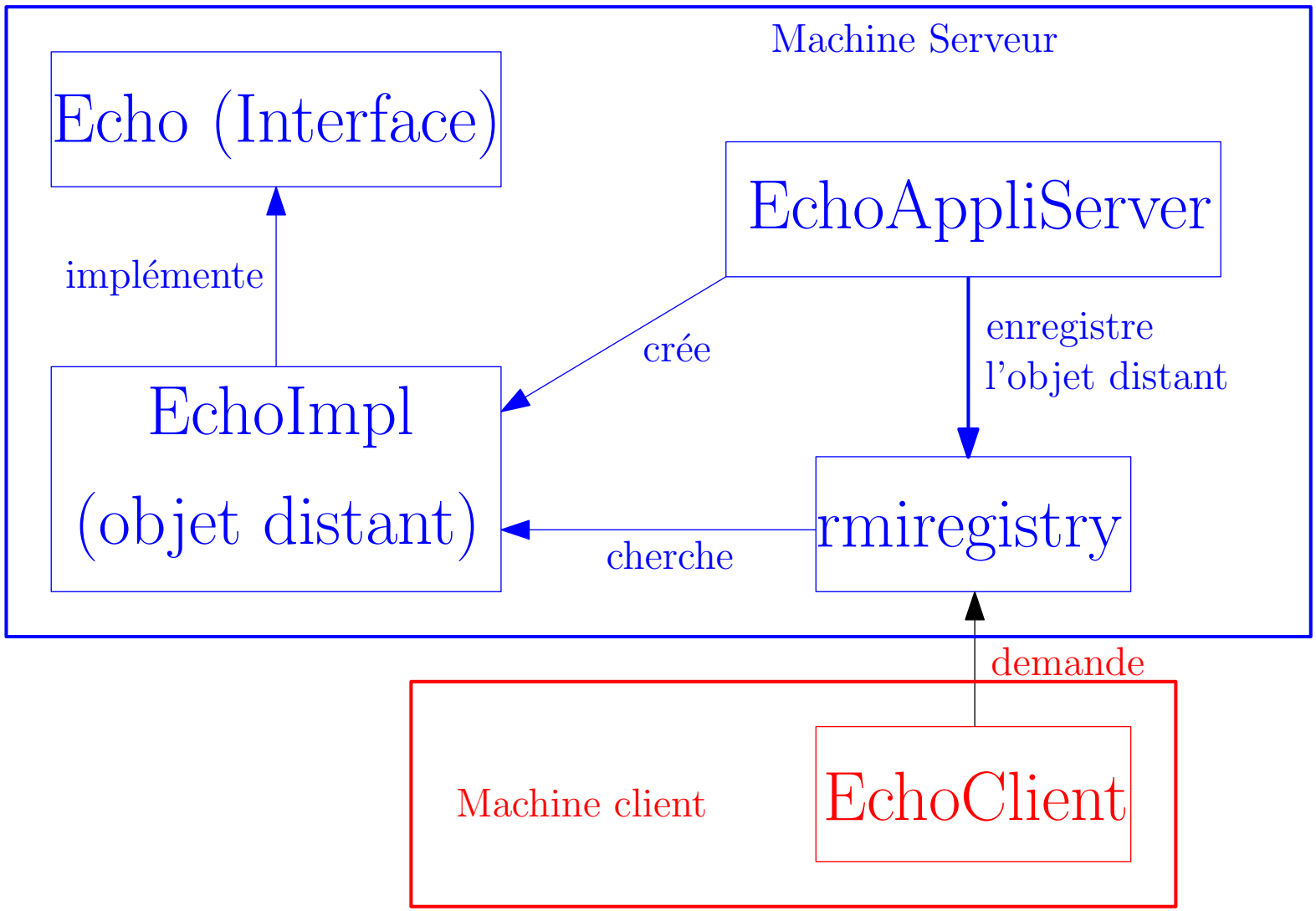
Les méthodes de `Naming`

- `static void bind(String nom, Remote obj)` : associe `nom` à l'objet `obj`.
- `static void rebind(String nom, Remote obj)` : réassocie `nom` à l'objet `obj`.
- `public static void unbind(String nom, Remote obj)` désassocie `nom` de l'objet `obj`.
- `static Remote lookup(String nom)` : renvoie l'objet `Remote` associé au nom `nom`.
- `static String[] list(String nom)` : renvoie le tableau des noms présent dans l'annuaire.

Exemple: Serveur d'écho

- Les fichiers:
  - `Echo.java` : interface
  - `EchoImpl.java` : implémentation (c'est-à-dire classe de l'objet distant)
  - `EchoAppliServer.java`, `EchoClient.java` : serveur et client.
- Les commandes préparatoires:
  - `javac *.java` : compilation
  - `rmic EchoImpl` : création squelette (skel) et amorce (stub) (optionnel depuis 1.5)
  - `rmiregistry&` (port 1099 par défaut)
- Exécution (attention à l'accessibilté des `.class` , via CLASSPATH de `rmiregistry`):
  - **Serveur** : `java EchoAppliServer`
  - **Client** : `java EchoClient (hote) msg`

Architecture du serveur d'écho



Echo.java

```
import java.rmi.*;

public interface Echo extends Remote {
    public String echo(String str)
        throws RemoteException;
}
```

EchoImpl.java

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

public class EchoImpl extends UnicastRemoteObject
                      implements Echo {
  public EchoImpl() throws RemoteException {
        super();
        }
  public String echo(String str)
    throws RemoteException {
    String ret;
    System.out.println("Recu: "+str);
    ret = "Echo: " + str;
    System.out.println("Renvoi: "+ret);
    return ret;
  }
}
```

EchoAppliServer.java

```
import java.rmi.*;

public class EchoAppliServer {
  public static void main(String args[]) {

    try {  // Création de l'OD
      EchoImpl od = new EchoImpl();
      // Enregistrement de l'OD dans RMI
      Naming.rebind("/echo", od);
      System.out.println("Enregistrement␣terminé␣...");
    }
    catch(Exception e) {
      System.out.println("Echec␣enregistrement\n"+e);
    }
  }
}
```

EchoClient.java I

```
import java.rmi.*;
import java.rmi.registry.*;
import java.net.*; // pour les url
public class EchoClient {
  public static void main(String args[]) {
    String registry="localhost";
    String msg = null;
    Echo od=null;
    if ( args.length >= 2 ) {
        registry = args[0];
        msg = args[1];
    }
    else {
      msg = args[0];
    }
}
```

EchoClient.java II

```
    iString url = "//"+registry + "/echo";
    System.out.println("URL: "+url);

    try{od = (Echo) Naming.lookup(url);}
    catch (NotBoundException e){}
    catch(MalformedURLException e){}
    catch(RemoteException e){}
    try{
    System.out.println("RÃ©sultat = "+od.echo(msg));
    }
    catch(RemoteException e){
      System.out.println("Exception "+e);
    }
  }
}
```

Exécution

```
serveur $ rmiregistry&
serveur $ java EchoAppliServer
Enregistrement terminé ...
Recu: coucou
Renvoi: Echo: coucou



-----------------------------------
client $ java EchoClient serveur coucou
URL: //serveur/echo
Résultat = Echo: coucou
```

Que se Passe-t-il?

- La JVM du client communique avec l'annuaire de la machine serveur
- Envoi des informations concernant l'objet distant
- Le client communique ensuite directement avec `EchoAppliServer`

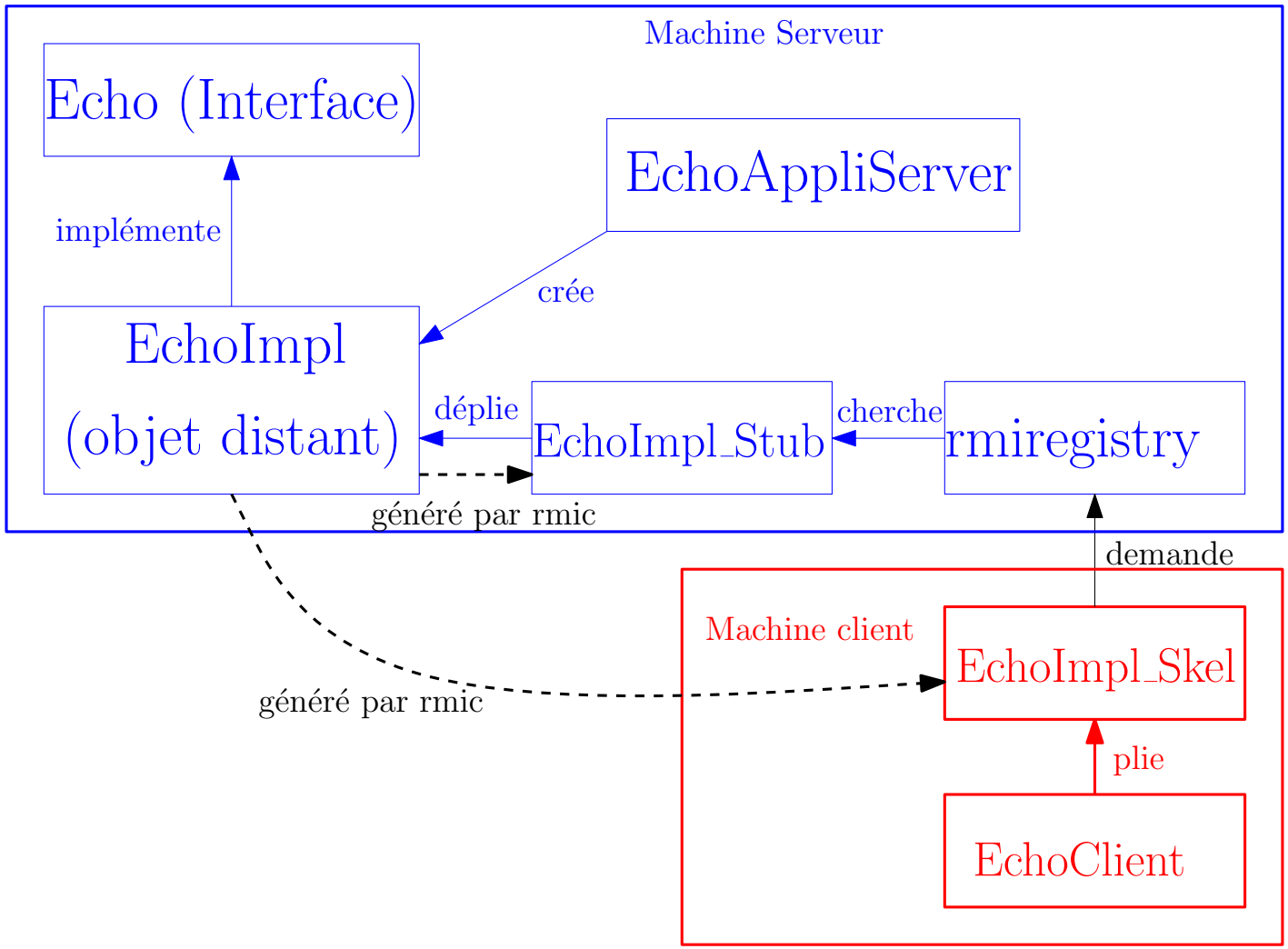
A noter : différence communication distante (socket TCP) et locale.

Amorces

A quoi sert la commande `rmic`?

Cette commande permet de compiler des “amorces” (stub et skeleton) à partir de l'implémentation de l'interface. Ces amorces assurent l'interface des JVMs vers la couche transport TCP/IP.

Architecture du serveur d'écho avec amorces



Amorces et Java 1.5

Java 1.5 permet de généraliser le mécanisme de `ClassLoader` aux ODs : si une classe n'est pas disponible localement, son amorce est compilée et récupérée automatiquement.

$\Longrightarrow$

`rmic` est désormais inutile (à partir de Java 1.5)
sauf si des clients de versions inférieures sont susceptibles de se connecter.

ClassLoader

Définition

Partie du JRE (Java Runtime Environment) qui charge dynamiquement les classes java dans la machine virtuelle java.

Les classes sont chargées à la demande.

Conclusion Provisoire

Pour utiliser un objet distant `od` via l'architecture RMI, il faut :

1. une interface et une implémentation de `od`,
2. (optionnel) compiler les amorces de l'implémentation,
3. serveur et client RMI :
   - un serveur qui enregistre `od` auprès de l'annuaire,
   - un client qui récupère `od`, par son nom, auprès de l'annuaire et le manipule de manière habituelle ,
4. démarrer le serveur d'annuaire,
5. démarrer le serveur (et attendre la fin de l'enregistrement),
6. lancer le client.

Communication entre Objets RMI

- A l'initiative du client
- sous la forme : Requête -> Réponse

Notification

Pour ne pas que les clients aient à faire de l'"écoute active" pour savoir si quelque chose a changé sur le serveur, on met en place une méthode de rappel :

- les clients s'enregistrent auprès du serveur
- le serveur maintient la liste de tous les clients à notifier
- lorsqu'un événement se produit, le serveur appelle la méthode de notification du client

NB : $\Longrightarrow$ programmation événementielle

Comment Faire en RMI?

Il faut que les clients deviennent “serveurs RMI” et que le serveur devienne ‘client RMI”.

- A chaque événement correspond à une interface
- Les clients et le serveur implémente l’interface
- Pour s’inscrire, le client passe au serveur l’objet `Remote` permettant de le notifier.

Exemple : TemperatureSensorServer.java et TemperatureMonitor.java

- **deux interfaces** : TemperatureSensor.java et TemperatureListener.java
- **deux implémentations de ces interfaces** : TemperatureSensorServer.java et TemperatureMonitor.java

TemperatureSensor.java

```
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;
}
```

TemperatureListener.java

```
interface TemperatureListener
  extends java.rmi.Remote {

  public void temperatureChanged
        (double temperature)
    throws java.rmi.RemoteException;
}
```

TemperatureMonitor.java I

```
import java.rmi.*;
import java.rmi.server.*;
public class TemperatureMonitor
    extends UnicastRemoteObject
    implements TemperatureListener {

    public TemperatureMonitor()
        throws RemoteException {} ;

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

TemperatureMonitor.java II

```
public static void main(String args[]) {
  try {
    String registry = "localhost";
    if (args.length >= 1)
       registry = args[0];
    System.out.println("Looking␣for␣temperature␣sensor␣@"+registry);
    String registration = "rmi://"+registry+
       "/TemperatureSensor";
    Remote remoteService = Naming.lookup(registration);
    TemperatureSensor sensor = (TemperatureSensor)
       remoteService;
    double reading = sensor.getTemperature();
    System.out.println("Original␣temp␣:␣"+reading);
    TemperatureMonitor monitor = new TemperatureMonitor();
    sensor.addTemperatureListener(monitor);
    } 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); }
    }
 }
```

TemperatureSensorSever.java I

```
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>();

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

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

TemperatureSensorSever.java II

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

TemperatureSensorSever.java III

```
public void addTemperatureListener
        (TemperatureListener listener)
        throws java.rmi.RemoteException {

        System.out.println("adding␣listener␣-"+listener);
        list.add(listener);
    }

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);
            }
            }
    }
```

TemperatureSensorServer.java IV

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

RMI avec réveil de serveur

Il reste un problème avec le serveur précédent.
$\Longrightarrow$ Le serveur doit rester actif même s'il n'y a pas de requêtes.

Solution

Un serveur qui ne se réveille que s'il y a une requête à résoudre.

$\Longrightarrow$ possible avec un serveur RMI.

Objets activables

Problématique

- Le nombre d'objets distribués par JVM est limité
- Dans des conditions réelles d'utilisation, plusieurs milliers d'objets distribuées peuvent être utilisés.
- À un instant donné, seule une partie des objets distribués est utilisée

$\Longrightarrow$ lancer des JVM (proposant des objets distribués) à la demande. C'est le rôle des objets activables

Outils/Package

- Un package java.rmi.activation
- Un démon rmid qui active les objets à la demande

Objets activables II

Paramètres

Quelles sont informations que le programme, qui prépare cette activation à la demande, doit fournir :

- Paramètres de la JVM associée à un ensemble d'objets distribués ?
  $\Longrightarrow$ Rôle de ActivationGroupDesc
- Identification d'une JVM associée à un ensemble d'objets distribués ?
  $\Longrightarrow$ Rôle de ActivationGroupID
- Informations associées à chaque objet distribué ?
  $\Longrightarrow$ Rôle de ActivationDesc

Objets activables III

Classe des objets distribués :

Les classes des objets distribués sont sous-classe de `Activatable`, ou passé en paramètre de la méthode statique `exportObject` d'`Activatable`

Objets activables IV

- Créer l'interface
- Créer une implémentation de l'objet
  - étend Activatable et non java.rmi.server.UnicastRemoteObject
  - possède le constructeur obligatoire XX(ActivationID id, MarshalledObject data)
- Créer le serveur :
  - Il crée les groupes d'activation
  - Il crée les objets et les associe au groupe d'activation
  - Il enregistre l'objet auprès du registre
- Créer un client standard
- Lancer le registre standard : rmiregistry
- Lancer le système d'activation : rmid
- Lancer le serveur

Objets activables V

Étapes principales

- Création de la description du groupe (ActivationGroupDesc) :
  - Propriétés + Paramètres de la JVM
- Obtention d'un identifiant de groupe (ActivationGroupID)
- Création de la description de l'objet activable (ActivationDesc) Nom de la classe + Position + Paramètre(s) aux constructeurs
- Enregistrement auprès du rmid
- Enregistrement auprès du rmiregistry

HelloImpl.java

```
public interface Hello
        extends java.rmi.Remote {

  public String sayHello(String nom)
    throws java.rmi.RemoteException;
}
```

HelloImpl.java

```
import java.rmi.*;
import java.rmi.activation.*;
public class HelloImpl extends Activatable
                    implements Hello {

  public HelloImpl
         (ActivationID id,MarshalledObject data)
         throws RemoteException{
    super(id, 0);
  }
  public String sayHello(String nom)
          throws RemoteException{
    return "Hello␣"+nom;
  }
}
```

Client.java

```
import java.rmi.Naming;

public class Client {
  public static void main(String args[]) {
    try {
      Hello obj = (Hello)
        Naming.lookup("//"+args[0]+"/Hello");
      String message = obj.sayHello(args[1]);
      System.out.println(message);
        } catch (Exception e) {
          System.out.println
                ("Client exception: " +e);
        }
  }
}
```

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class Server{
  public static void main(String args[]) throws Exception{
    try {
      System.setSecurityManager(new RMISecurityManager());
      Properties env = new Properties();
      env.put("java.security.policy",
        "file://home/labourel/ppd/RMI/helloworld.policy");
      ActivationGroupDesc mygroupdes =
        new ActivationGroupDesc(env, null);
      ActivationGroupID mygroupid =
        ActivationGroup.getSystem().registerGroup(mygroupdes);
      ActivationGroup.createGroup(mygroupid, mygroupdes, 0);
      ActivationDesc objectdesc = new ActivationDesc
        ("HelloImpl","file://home/labourel/ppd/RMI/", null);
      Hello myobject = (Hello)Activatable.register(objectdesc);
      Naming.rebind("/Hello", myobject);
    }
    catch(Exception e){
      System.out.println("Exception␣"+ e);
    }
  }
}
```

Execution

L'exécution :

- rmiregistry
- rmid -J -Djava.security.policy=helloworld.policy
- java -Djava.security.policy=helloworld.policy -Djava.rmi.server.codebase= file:/Users/arnaudlabourel/Desktop/ppd/RMI Server
- java Client

Remarques sur la Sécurité

Le fait de pouvoir transmettre du code exécutable à distance est, par principe, une faille de sécurité.

$\Longrightarrow$ `SecurityManager`

Mais

- difficile à gérer dans le code
- gestion externe possible : `.java.policy`
  - syntaxe assez complexe
  - commande `policytool`
- $\Longrightarrow$ souvent désactivé...

Exemple de .java.policy

```
grant { /* Autorise toutes connexions ... */
/* depuis localhost */
permission java.net.SocketPermission "localhost",
                "accept, connect, listen, resolve";
/* depuis le rÃ©seau privÃ© local */
permission java.net.SocketPermission "192.168.*:*",
                "accept, connect, listen, resolve";
/* depuis *.univ-mrs.fr */
permission java.net.SocketPermission "*.univ-mrs.fr:*",
                "accept, connect, listen, resolve";
 };
```

Java 1.5 : RMI avec SSL

Depuis Java 1.5, il est possible d'utiliser (notamment) SSL comme couche de transport pour RMI.

$\Longrightarrow$

- intégrité,
- confidentialité,
- authentification du serveur,
- optionnellement authentification du client.