Applications réparties en Java

Paquetage **`java.net`** – Sockets
Objets distants transparents – RMI

Yves Bekkers

Application répartie sur un réseau

- Deux applications communiquent via un réseau
  - L'application serveur
    - Créé un port de communication
    - Créé des services
    - Attend les demandes de service effectuées par les clients
    - Réponds aux demandes en envoyant des valeurs
  - L'application client
    - Se connecte au port de communication
    - Effectue des demandes de service par l'envoie de valeurs sur le port de communication
    - Attend les réponses sur le même port

Communication par message

- Soit un service correspondant à une procédure de la forme `TypeResultat NomDuService(Type1, ... Typek)`
- le client
  - envoit un message de la forme :
    `CodeService, p1, ... pk`
  - se met en attente du résultat,
  - après réception du résultat, décodage du résultat
- le serveur
  - reçoit le code du service demandé, le décode (au moyen d'un ***switch*** par exemple),
  - reçoit les *k* paramètres en les décodant et en restituant leur types respectifs,
  - appelle la procédure de ce service
  - code le résultat et le retourne au client par message.

Paquetage `java.net`

- Classes pour l'implémentation d'applications réseau
  - Identification des machines
    - **`java.net.InetAddress`**
      Classes pour représenter une adresse IP (protocole Internet)
    - **`java.net.URL`**
      Classes pour représenter une URL sur Internet
  - Ports de communication
    - **`java.net.Socket`**
      Extrémité d'une communication entre deux machines
      Port coté client
      Supporte des flux de données bidirectionels
    - **`java.net.ServerSocket`**
      Extrémité d'une communication entre deux machines
      Port coté serveur
      Attend les demandes de services arrivant sur le port

Sockets TCP

- Connection point à point
- Classe `Socket`
  - Connexion à une machine distante
  - Envoi/Réception de données
  - Fermeture d'une connexion
- Classe `SocketServer`
  - Attachement à un port
  - Acceptation d'une demande de connexion à un port local
  - Attente de demandes de connexion

Classe `Socket`

- Constructeurs

```
public Socket(String hote, int port) throws
  UnknownHostException, IOException
public Socket(InetAddress hote, int port)
  throws IOException
```

- Envoi/Réception de données

```
public InputStream getInputStream() throws
  IOException
public OutputStream getOutputStream() throws
  IOException
```

- Fermeture

```
public synchronized void close() throws
  IOException
```

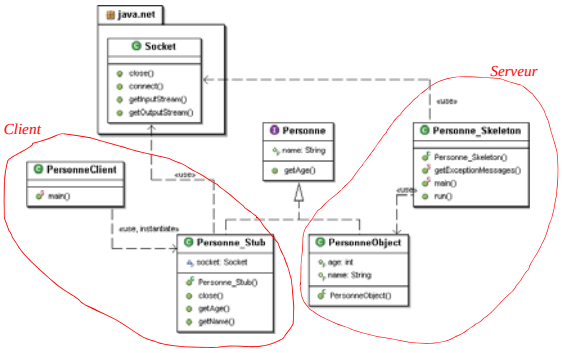
Appels de méthodes à distance

- Une interface commune entre les clients et le serveur

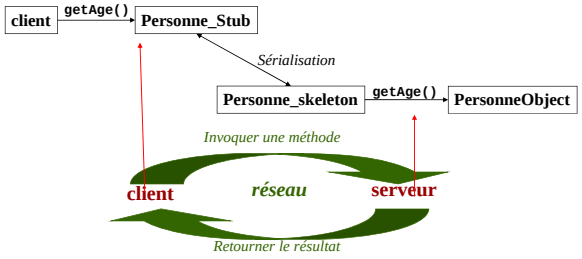
```
public interface Personne {
  public int getAge() throws Throwable;
  public String getName() throws Throwable;
}
```

- Deux implémentations de l'interface
  - Coté client `Personne_stub`
    - Connection au port de communcation
    - Sérialisation (codage) des demandes
    - Attente et décodage du résultat
  - Coté serveur `Personne_skel`
    - Mise en place d'un port de communcation
    - Attente et décodage des demandes de service
    - Codage et renvoie du résultat

Schéma de l'application



Simuler des appels de méthodes à distance



Classe `Personne_Stub`

- Constructeur

```
public Personne_Stub() throws Throwable {
   pSocket = new Socket("localhost",9000);
}
```

- Méthode `getAge()`

```
ObjectOutputStream outStream = new
  ObjectOutputStream(pSocket.getOutputStream());
outStream.writeObject("age");
outStream.flush();
ObjectInputStream inStream = new
  ObjectInputStream(pSocket.getInputStream());
return inStream.readInt();
```

- Méthode `close()`

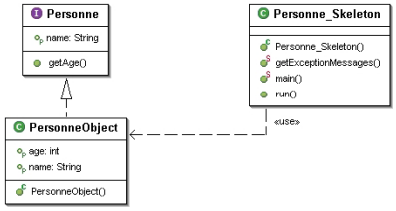
```
public void close() throws Throwable {
   pSocket.close();
}
```


Classe `Personne_Client`

- Méthode `main()`

```
Personne_Stub p = new Personne_Stub();
int age = p.getAge();
String nom = p.getName();
p.close();
System.out.println(nom+" est agé de "+age+"
  ans");
```


Serveur



Classe Personne_Skeleton

- Constructeur

```
public Personne_Skeleton(PersonneObject p) {
 this.myPersonne = p;
}
```

- Méthode main()

```
public static void main(String[] args) {
   Personne_Skeleton skel = new Personne_Skeleton(
       new PersonneObject("paul", 25));
   skel.start();
}
```


Classe Personne_Skeleton (1)

- Méthode run()

```
// Création d'un nouvel objet ServerSocket
ServerSocket serverSocket = new ServerSocket(9000);
while (true) {
   Socket socket = serverSocket.accept(); // attente
   while (socket != null) {
      ObjectInputStream inStream = null;
      try {
         inStream = new
  ObjectInputStream(socket.getInputStream());
      } catch (EOFException e) {// connection fermée
         socket.close();
         break; // fin de service pour cette connection
      }
   ...
   }
}
```


Traiter une demande de service (2)

- Méthode run() suite

```
String service = (String) inStream.readObject();
if (service.equals("age")) {
   int age = myPersonne.getAge();
   ObjectOutputStream outStream =
     new ObjectOutputStream(socket.getOutputStream());
   outStream.writeInt(age);
   outStream.flush();
} else if (service.equals("name")) {
   String name = myPersonne.getName();
   ObjectOutputStream outStream =
     new ObjectOutputStream(socket.getOutputStream());
   outStream.writeObject(name);
   outStream.flush();
}
```


RMI
Remote Method Invocation

RMI rendre transparents les échanges

- **Sans** RMI : Le concepteur de l'application fait tout le travail de transmission
  - Tâche de programmation lourde, dangereuse (nombreuses sources d'erreur)
- **Avec** RMI : mise en œuvre automatique du protocole de communication

Mise en place de services distants

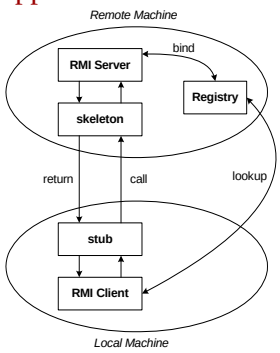
- Définition d'objets distants :
  - Objets résidents sur un site et utilisés sur un autre
- Deux technologies concurrentes
  - CORBA : applications réparties faisant intervenir divers langages (C++, Smalltalk, Eiffel, Java, ...)
  - RMI : une norme purement Java (***Remote Method Invocation***).

RMI – programmation distribuée

- Protocole client/serveur propre à java.
  - Package pour faire dialoguer des objets java par Internet.
    - Repose sur l'utilisation de la librairie **`java.net`**
- Un concurrent de Corba, avec quelques différences :
    - Simplicité de mise en oeuvre :
      - Pas de serveur spécial, le serveur est une classe java
      - transparence totale (intégration dans l'API Java)
    - RMI : échange d'instances d'objets facilité grâce à :
      - L'utilisation du **`Classloader`** de Java
      - La réflexivité native de Java
- Inconvénients : Java uniquement

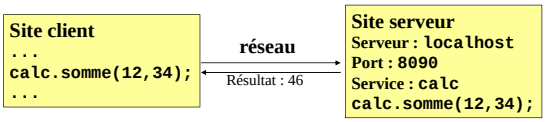
Structure d'une application RMI

- Le serveur lie son nom à un objet Registry
- The client recherche le nom du serveur et établie une connection.
- Le talon (Stub) sérialise les paramètres au squelette,
- Le squelette invoque la méthode distante et sérialise le résultat en retour vers le talon.



RMI – programmation distribuée

- Le code n'est pas localisé sur une unique machine,
- Il est déployé sur plusieurs machines en réseau.



Un exemple en 5 étapes

Un exemple – calculateur distant

- Un ***site serveur*** définit un objet **Convertisseur** doté d'une méthode **toEuro()** qui reçoit en paramètre une valeur (un float) et rend un résultat sous forme d'un float.
- Sur un ***site client*** on peut accéder à cet objet par un ***nom externe*** qui est une chaîne de caractères formée du nom du serveur, d'un numéro de port, et d'un identificateur d'objet, de la forme :

//localhost:8090/convertisseur

- Le client obtient la référence de l'objet distant puis l'utilise directement en appelant la méthode **toEuro()**.

(1) Interface d'objet distant

- Un convertisseur

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Conversion extends Remote {
  public float toEuro(float x)
    throws RemoteException;
}
```

- A noter
  - Extension de l'interface `Remote` (objet distant)
  - Prévoir que les méthodes sont susceptibles de lancer des exceptions `RemoteException`

(2) Implémentation coté serveur

```
public class ConvertisseurImpl
  extends UnicastRemoteObject
  implements Conversion {

  protected ConvertisseurImpl()
    throws RemoteException {
      super();
  }

  public float toEuro(float x)
    throws RemoteException {
      return x/6.55957f;
  }
}
```

Extension de `UnicastRemoteObject`

Implémentation de l'interface

Définir un constructeur

(3) Compilation des objets distants

- Java 1.4 Deux compilations sont nécessaires
  - Compilation des interfaces et des implémentations d'objets distants

    `javac –classpath . *.java`
  - Compilation des souches et des squelettes d'objets distants

    `rmic –classpath . MyClassDistant`
- Java 1.5 une seule compilation
  - les objets distants sont découverts à l'exécution par des « *proxies* »

Les souches et les squelettes

- Deux classes supplémentaires pour chaque objet distants
  - La souche (stub) coté client émet les paramètres et récupère les résultats
  - Le squelette (skeleton) coté serveur reçoit les paramètres retourne le résultat

(4) Initialisation du site serveur et enregistrement d'un objet distant

```
public class ServeurConvertisseur {
  public static void main(String[] args) {
    try {
      LocateRegistry.createRegistry(8090);
      ConvertisseurImpl euroConv = new
         ConvertisseurImpl();
      Naming.bind(
     "//e103c04.ifsic.univ-rennes1.fr::8090/testConv",
              euroConv);
    } catch (Exception e) {
      System.out.println("erreur rmi");
    }
  }
}
```


Étapes de mise en place du serveur

- lancement d'un processus d'enregistrement de service sur le port `8090` du « `localhost` »

  `LocateRegistry.createRegistry(8090);`
- Création de l'objet distant

  ```
  ServeurConvertisseur euroConv =
     new ServeurConvertisseur();
  ```
- Identification de l'objet

  ```
  Naming.bind(
     "//localhost:8090/testConv",
     euroConv);
  ```


Arrêt du service

- Détruire le lien

  ```
  Naming.unbind(
      "//localhost.irisa.fr:8090/testConv");
  ```

(5) Connexion d'un client à un objet distant connu par son nom externe

```
public class ClientEuroConv {
  public static void main(String[] args) {
    try {
      Conversion euroConv = (Conversion) Naming.lookup(
      "//localhost:8090/testConversion");
      float x = Float.parseFloat(args[0]);
      System.out.println("en entrée = " + x);
      System.out.println("En sortie = " +
           euroConv.toEuro(x));
    } catch (Exception e) {
      System.out.println("Erreur rmi");
    }
  }
}
```


Utilisation d'un objet distant

- Mise en correspondance avec l'objet distant

```
Conversion euroConv = (Conversion) Naming.lookup(
   "//localhost:8090/testConversion");
```

- Utilisation de l'objet distant

```
float x = Float.parseFloat(args[0]);
   System.out.println("en entrée = " + x);
   System.out.println("En sortie = " +
        euroConv.toEuro(x));
```


Type des arguments et des résultats

- Pratiquement tous les types d'arguments et de résultats sont acceptés
  - Types primitifs
  - Objets distants
    - ceux qui implémentent `java.rmi.Remote`
  - Types non-primitifs sérialisables
    - ceux qui implémentent `java.io.Serializable`

Mode de passage de paramètre et de résultat

- Objets Distants (Remote)
  - Passage par référence (stub)
    - Seules les méthodes de l'interface `Remote` sont disponibles
    - Les modifications sont faites sur l'objet original
- Objets locaux
  - Passage par copy (à l'aide de la sérialisation)
    - Les modifications sont faites sur la copie

Mise en place sur un serveur **http** classique

- Côté serveur :
  - `MyClassDistant_Skel.class`
  - mettre les **.class** des interfaces locales et distantes
- Côté client :
  - `MyClassDistant_Stub.class`
  - (si le serveur transmet des objets locaux particuliers, il doit les fournir, comme ferait un client)

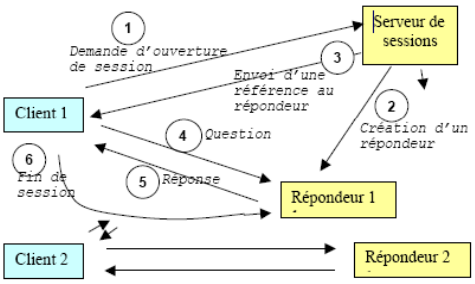
Dissémination des références

- En règle générale, dans une application répartie, seul un objet (ou quelques objets) est (sont) connu(s) par un nom externe, un objet "serveur central" de l'application répartie.
- Cet objet préexistant est le germe d'activités qui ensuite créent d'autres objets distants et se les communiquent par paramètres ou résultats de procédures, de façon tout à fait conventionnelle, comme si ces objets étaient situés sur le même site.

Références

- Tutorial Sun
  `http://java.sun.com/docs/books/tutorial/rmi/index.html`

Le tp



Deux interfaces

- ServeurSession

```
public interface ServeurSession extends Remote {
public Repondeur ouvreSession() throws
  RemoteException;
}
```

- Repondeur

```
public interface Repondeur extends Remote {
public void poseQuestion(String q) throws
  RemoteException;

public String obtientReponse() throws
  RemoteException;

public void fin() throws RemoteException;
}
```


Interfaces de saisie