Applications distribuées - 3

Java RMI

Module RCPI01

Sebastien.Kramm@univ-rouen.fr

IUT R&T Rouen

2018-2019

Sommaire

Introduction

Concepts et description

Exemple

Introduction

- Java RMI : une API "orientée objet" permettant la construction d'applications distribuées en masquant les aspects transport.
- Objectif de ce CM/TP : présentation rapide des possibilités de cette techno.

Problématique : Programmation Orientée Objet

- POO classique : on manipule des objets instanciés dans la mémoire de la machine.

```
Voiture v = new Voiture();
v.Demarrer();
v.Accelerer();
```

- Applications réparties : les objets sont sur d'autres machines
  ⇒ Comment accéder à ces objets ? ? ?
- Solution Java : RMI (*Remote Method Invocation*)
  https://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html

Sommaire

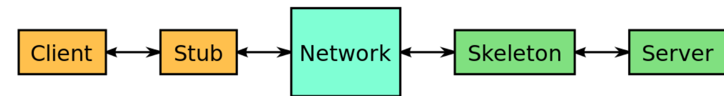
Introduction

Concepts et description

Exemple

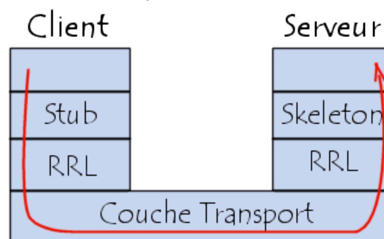
Java RMI

- API Java permettant de manipuler des objets distants de la même façon que sur la machine locale.
  - Un serveur instancie des objets en mémoire.
  - Un client (distant) accède aux objets via une **interface**, et peut invoquer des méthodes dessus.
- Solution "tout Java", contrairement à CORBA qui peut manipuler des objets à distance avec n'importe quel langage.
- Mais beaucoup plus simple à mettre en œuvre.



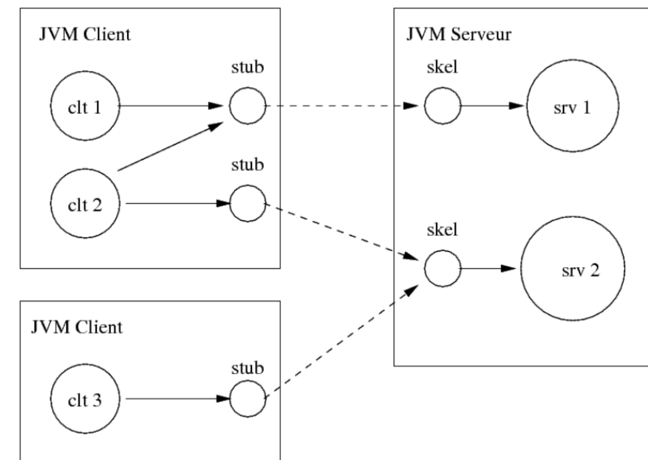
Vue en couches

- Les connexions et les transferts de données dans RMI sont effectués par Java sur TCP/IP grâce à un protocole normalisé :
  - JRMP (*Java Remote Method Protocol*)
  - ou RMI-IIOP (*Internet Inter-Orb Protocol*), protocole hérité de CORBA
- Une couche intermédiaire (RRL : *Remote Reference Layer*) permet de fournir un service de **nommage**, permettant de localiser les objets via un **nom**.
  (appellée aussi "registre RMI")



source :CCM

Organisation logique & vocabulaire



src : JL Massat

Ecrire une application répartie en RMI

- Pour chaque objet partagé, il faudra définir :
  - une classe "interface",
  - une classe d'implémentation de l'interface.
- Il faudra en sus écrire deux programmes (fonctions `main()`) :
  - le serveur, qui devra instancier les objets et les mettre à disposition,
  - le client.
- Compilation : tout sur machine de dev., deploiement à prévoir
- A l'éxécution, il faudra, dans l'ordre :
  1. lancer le service d'annuaire (commande `rmiregistry`),
  2. lancer le serveur,
  3. lancer le client.

Attributs publics ?

- L'accès aux attributs publics de l'OD n'est **pas possible** : ils ne sont pas décrits dans l'interface distante
  ⇒ Il faut écrire des méthodes ("getters" et "setters") pour consulter et/ou modifier les attributs d'un OD, et déclarer ces accesseurs dans l'interface.

```
public class MonMachin ...
{
    public int maValeur;

    public int getMaValeur() {
        return maValeur;
    }
}
```


Sommaire

Introduction

Concepts et description

Exemple

Exemple : Hello World

- Cahier des charges :
  Ecrire un programme qui exécute sur un objet sa méthode `getMessage()` (qui renvoie "Hello World") et afficher la valeur renvoyée.
- Version locale, non distribuée : tout sur la même machine, 2 fichiers (programme + classe de l'objet).
- Version distribuée avec Java RMI :
  - l'objet est crée sur une machine (serveur), qui est en attente
  - une autre machine (client) accède à l'objet sur le serveur et exécute la méthode sur l'objet
  - le client affiche la valeur renvoyée

  ⇒ Il faudra 4 fichiers source.

Version locale (SANS Java RMI)

- Le programme (inséré dans `main()`) :

```
Objet ob = new Objet();   // creation
String msg = ob.getMessage();   // appel de la
    méthode
System.out.println( msg ); // affichage
```

- La classe :

```
public class Objet
{
    public String getMessage()
    {
        return "Hello World";
    }
}
```

- Implique que la classe et le programme soient sur la même machine et compilés ensemble.

Version distribuée 1/4 : programme serveur

- Programme qui crée l'objet `RmiObjet` et le met à disposition via le service de nommage avec le nom `monOD`

```
public class RmiServer
{
   public static void main( String args[] ) throws
       Exception
   {
       RmiObjet obj = new RmiObjet();

       Naming.bind( "//localhost/monOD", obj );
   }
}
```

Rem : La dernière ligne lance un *thread* caché, qui va créer l'objet et le maintenir en vie : le programme ne va **pas** s'arrêter.

Version distribuée 2/4 : programme client

- Récupération d'un objet `obj` de type `ObjetIntf` via le service de nom, à partir du nom `monOD`
- Exécution de sa méthode `getMessage()` et affichage du résultat.

```
public class RmiClient {
  public static void main(String args[]) throws Exception
  {
    ObjetIntf obj
        = (ObjetIntf)Naming.lookup( "//localhost/monOD" );
    String msg = obj.getMessage();
    System.out.println( msg );
  }
}
```

Rem : les sauts de ligne sont ici juste pour la pagination !

Version distribuée 3/4 : **interface** de l'objet

Ne fait que **énumerer** ce que l'objet peut faire (liste des fonctions, sans le code).

- Interface de l'objet :
  - Ecriture d'une classe `ObjetIntf` de type `interface`
  - "étend" la classe Java `Remote` avec une méthode `getMessage()` (non définie ici !)

```
public interface ObjetIntf extends Remote
{
    public String getMessage() throws RemoteException;
}
```

Version distribuée 4/4 : **implémentation** de l'objet

- Doit fournir un constructeur (même vide), qui lancera l'exception `RemoteException()` en cas de problème.

```
public class RmiObjet
    extends UnicastRemoteObject // classe Java
    implements ObjetIntf      // mon interface
{
// constructeur (vide ici)
    public RmiObjet() throws RemoteException
    {}

// implementation de la fonction
    public String getMessage() {
        return "Hello World";
    }
}
```

Packages nécessaires

- En pratique : dans les sources, il faudra importer le package RMI via les lignes suivantes :

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

Lancement du service de nommage

- Avant de lancer le serveur, il faut que le service soit lancé en mémoire.
  Ceci se fait depuis le shell :
  - Sous Linux : `rmiregistry&`
  - Sous Windows : `start rmiregistry`
- Alternative : on peut aussi faire en sorte que ce soit le serveur qui le lance au démarrage, via le code suivant :
  (1099 = port par défaut du service RMI)

```
    try {
        LocateRegistry.createRegistry(1099);
        System.out.println( "RMI registry
            created" );
    }
    catch (RemoteException e) {
        System.out.println( "RMI registry
            already exists." );
    }
```

Rem : il faudra ajouter la ligne `import java.rmi.Registry.*;`

Service de nommage : compléments

La classe `java.rmi.Naming` fournit les méthodes suivantes :

- Coté serveur :
  - `void bind(String name, Remote obj)`
    Référencement de l'objet dans le registre RMI.
  - `void rebind(String name, Remote obj)`
    Remplacement d'objet dans le registre RMI (ne lance pas d'exception s'il existe déjà dans le registre)
  - `void unbind(String name)`
    Supprime l'association, et termine le *thread* associé à l'objet.
- Coté client :
  - `Remote lookup(String name)`
    Renvoie une référence sur un objet distant (un "stub") dans le registre RMI, à partir de son nom complet.
  - `String[] list(String name)`
    Renvoie un tableau contenant les noms de tous les objets distants contenus dans le registre RMI