Cours n°4

RMI
(Remote Method Invocation)

Safa YAHI
safa.yahi@univ-amu.fr

Introduction

Mécanismes de communication entre applications réparties :

- Sockets (bas niveau)
- RPC
- Corba
- **RMI**
- Services Web

Introduction

- Le système Java **RMI** (Remote Method Invocation) permet à un objet sur une machine virtuelle Java d'invoquer les méthodes d'un autre objet sur une autre machine virtuelle Java.
- Communication entre objets Java répartis.
- RMI se base sur les sockets TCP.

Introduction

Un application RMI comprend en général deux programmes : un serveur et un client.

- Un serveur typique crée un (des) objet(s) distant(s) et le (les) rend accessible (s) à des clients.
- Un client invoque les méthodes distantes des objets distants.

Objet distant

Objet distant

- Un objet distant est un objet qui propose des méthodes qui peuvent être appelées depuis une autre JVM.
- Un **objet distant** est défini par une classe qui implémente une interface distante.
- Une **interface distante** dérive de l'interface Remote et déclare un ensemble de méthodes distantes (juste les profils des méthodes sans leur corps).

Objet distant

- Chaque méthode distante doit déclarer l'exception RemoteException dans sa clause throws.
- On doit s'assurer que tous les arguments sont bien sérialisables.
- Un argument est sérialisable s'il est :
  - Soit de type de base (int, double, etc)
  - Soit d'une classe qui implémente l'interface Serializable.

UnicastRemoteObject

- En plus de l'implémentation de l'interface distante, un objet distant dérive de la classe **UnicastRemoteObject.**
- On doit définir un constructeur qui gère l'exception RemoteException.
- Notons qu'elle implémente Sérializable.

UnicastRemoteObject

Un objet distant peut définir une méthode qui n'apparait pas dans l'interface distante qu'il implémente mais cette méthode ne pourra pas être appelée à distance (elle sera utilisée uniquement en local).

Exemple Hello World

- Version distribuée de l'exemple “Hello World !”
- Un objet distant offre une méthode *afficherHello()* qui retourne la chaîne de caractères “Hello World !”
- Une client appelle cette méthode à distance.

Exemple Hello World (suite)

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote {

    String afficherHello() throws RemoteException;

}
```

Exemple Hello World (suite)

```
import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class HelloRemoteObject extends UnicastRemoteObject
implements HelloInterface {

    protected HelloRemoteObject() throws RemoteException {

        super();

    }

    public String afficherHello() throws RemoteException {

        return "Hello World !";

    }

}
```

Serveur RMI

Serveur RMI

- Créer une instance de l'objet distant
- Associer un nom logique à la référence de l'objet distant dans un annuaire : le registre java RMI.
- Le registre RMI doit être déjà lancé, soit en ligne de commandes, soit depuis le code du serveur.

Exemple Hello World (suite)

```
public class ServeurHello {
public static void main(String args[]) {
HelloRemoteObject obj;
try {
    obj = new HelloRemoteObject();
    Registry registre = LocateRegistry.getRegistry();
    registre.rebind("Hello", obj);
    System.out.println("Ready");
    } catch (RemoteException e) {
    e.printStackTrace();
    } } }
```

Là commence
l'attente des requêtes

RMI Registry

- Java RMI fournit un registre qui permet aux applications d'associer un nom (qui doit être publique) à la référence d'un objet.
- Il s'agit d'un service de nom simplifié.
- Ceci permet aux clients d'obtenir la référence d'un objet distant en le cherchant par son nom dans le registre RMI.

RMI Registry

- Registry registre = LocateRegistry.getRegistry()

  => permet d'avoir une référence vers le registre sur localhost en utilisant le port par défaut (1099)

- Registry registre = LocateRegistry.getRegistry(numPort)

  => permet d'avoir une référence vers le registre sur localhost en utilisant le port numPort.

- Pour désactiver un objet distant, on utilise la méthode unbind().

- Si on veut enregistrer un objet avec un nom pour lequel existe déjà une entrée dans le registre RMI, alors la rebind() écrase l'ancienne entrée à l'inverse de bind() qui genère une exception.

RMI Registry

- Le lancement du registre RMI depuis un terminal sur la machine du serveur se fait par : **rmiregistry &**
- Par défaut, il écoute sur le port 1099.
- On peut le vérifier via : netstat -ltn
- On peut utiliser un autre port (à part le port par défaut) comme suit : **rmiregistry *num_port* &**
- Il est possible de le lancer depuis le code du serveur avec la méthode createRegistry().

Le codebase du serveur

- On doit positionner le **codebase** (la propriété **java.rmi.server.codebase**) du serveur au nom du dossier contentant la définition de l'interface distante pour qu'elle puisse être téléchargée.
- Notons que si on lance le RMI Registry depuis le même répertoire qui contient la définition de l'interface distante, alors on n'a pas besoin de positionner le codebase de ce dernier.

Le codebase du serveur

Deux possibilités :

- Depuis la ligne de commande :

  java -Djava.rmi.server.codebase=file:classDir/ ServeurHello

- Depuis le source :

  ```
  Properties prop = System.getProperties();
  prop.put("java.rmi.server.codebase", "file:classDir/ ");
  ```

- ClassDir : le nom du répertoire qui contient la définition de l'interface distante.

Le hostname du serveur

Lorsque le client et le serveur se trouvent sur deux machines différentes, il faut positionner l'option java.rmi.server.hostname du serveur à son adresse IP (depuis la ligne de commande ou depuis le code).

Une autre méthode pour exporter un objet distant via exportObject()

- Il est possible que l'object distant ne dérive pas de UnicastRemoteObject.
- Dans ce cas, le serveur l'exporte en utilisant la méthode statique UnicastRemoteObject.exportObject(...) .

Exemple :

```
HelloInterface stub  = (HelloInterface)
UnicastRemoteObject.exportObject(obj, 0);
```

Serveur RMI multithread

- Par implémentation, les serveurs RMI sont multithread : plusieurs clients peuvent invoquer des méthodes distantes d'un même objet distant simultanément.
- On doit gérer les accès conccurents aux variables partagées.

Serveur RMI multithread

- Par implémentation, les serveurs RMI sont multithread : plusieurs clients peuvent invoquer des méthodes distantes d'un même objet distant simultanément.
- On doit gérer les accès conccurents aux variables partagées.

Synchronized pour l'exclusion mutuelle

- Le mécanisme “synchronized” permettent la gestion des accès conccurents à des ressources partagées.
- Quand un thread exécute une méthode déclarée “synchronized” d'un objet, tous les autres threads qui invoquent des méthodes synchronized du même objet se bloquent juqu'à la fin de la première méthode.
- Pour améliorer la conccurence, on peut synchroniser juste une partie d'une méthode en mettant un verou sur un objet particulier avec “synchronized statement” :

```
synchronized(objet){
    statements;}
```

Client RMI

Client

- Le client appelle une méthode distante d'un objet distant.
- Il n'y a pas de copie de l'objet distant du serveur vers le client.
- Par contre, le client a besoin d'un représentant en local de l'objet distant : on l'appelle **stub** (ou souche).
- Un stub implémente la (les) même(s) interface(s) distante(s) que l'objet distant associé.
- Le client appelle les méthodes distantes sur le stub local qui se charge de les invoquer sur l'objet distant (en masquant au client certains détails).

Client

- Le client obtient une référence vers le registre RMI du serveur en utilisant la méthode **LocateRegistry.getRegisetry** (en spécifiant l'adresse du serveur).
- Le client invoque la méthode **lookup** sur le registre RMI afin d'avoir une référence sur l'objet distant étant donné son nom.
- Le client invoque la méthode de l'objet distant sur le stub local.

Exemple Hello World (suite)

```
public class ClientHello {
public static void main(String args[]) {
try {
String adrServeur = “139.124.187.29”;
Registry registre = LocateRegistry.getRegistry(adrServeur);
HelloInterface stub = (HelloInterface) registre.lookup("Hello");
String msg = stub.afficherHello();
System.out.println(msg);
} catch (Exception e) {e.printStackTrace();
} } }
```

Passage de paramètres

Passage de paramètres

Deux cas :

- Un objet distant est passé par référence.
- Un objet “local” est passé par valeur.

Passage de paramètres

- La classe de l'objet passé comme paramètre par le client dans l'appel d'une méthode distante est un **sous type** du type du paramètre déclaré dans la définition de la méthode distante au niveau du serveur.
- Un sous type est :
  - Soit une implémentation de l'interface qui correspond au type du paramètre de la méthode
  - Soit une sous-classe de la classe qui correspond au type du paramètre de la méthode.
  - Ça peut être aussi la même classe.

Classe de paramètre inconnue

Si le serveur ne connait pas la classe d'un objet passé comme paramètre à une méthode distante, alors il télécharge le code de cette classe depuis le codebase du client en local (file:...) ou depuis un serveur FTP ou HTTP.

Classe de paramètre inconnue

Pour ce faire, il faut :

- Activer le gestionnaire de sécurité (sinon on obtient un message d'erreur : *“no security manager : RMI class cloader disabled”*)
- Définir les permissions
- Préciser le codebase du client

Gestionnaire de sécurité

Mise en place d'un gestionnaire de sécurité

- if (System.getSecurityManager()==null) System.setSecurityManager(new SecurityManager());
- Properties prop = System.getProperties();
- prop.put("java.security.policy",nom_fichier);

où *nom_fichier* est le nom du fichier décrivant la politique de sécurité à mettre en place (ensemble de SocketPermission).

SocketPermission

- Une permission de type **java.net.SocketPermission** comprend une machine et un ensemble d'actions spécifiant comment accéder à cette machine.
- **Machine :**
  - host = (hostname | IPaddress)[:portrange]
  - portrange = portnumber | -portnumber | portnumber-[portnumber]
- Parmi les **actions** possibles :
  - accept
  - connect

Exemple

```
grant {
permission java.net.SocketPermission "139.124.187.1:9999",
"connect, accept";
};
```

permet au code associé de se connecter à la machine 139.124.187.1 sur le port 9999 et aussi d'accépter les connexions depuis ce même port (depuis la même machine).

Exemple

```
grant {
permission java.net.SocketPermission "infodoc.iut.univ-aix.fr:1024-", "connect";
};
```

permet de se connecter à la machine infodoc sur tous les ports >=1024

Custum Socket Factories

Utilité

- Les “**custum sockets factories**” peuvent être utilisées pour personaliser la couche transport RMI.
- Par exemple, elles peuvent être utilisées pour :
  - modifier les options des sockets
  - gérer les points de terminaison
  - gérer l'authentification
  - coder les données (cryptage ou compression).

Définition

Lorsqu'un objet distant est exporté (par exemple avec les constructeurs ou les méthodes exportObject de UnicastRemoteObject), il est possible de spécifier un “custom client socket factory et un “custom server socket factory”.

Côté serveur

- Côté serveur, les custum socket factories implémentent l'interface **RMIServerSocketFactory**.
- On implémente en particulier la méthode **createServerSocket()** pour spécifier le type des Server Sockets à utiliser.

Côté client

- Côté client, les custum socket factories implémentent l'interface **RMIClientSocketFactory**.
- On implémente en particulier la méthode **createSocket()** pour spécifier le type des sockets à utiliser.

JRMP et IIOP

- RMI peut utiliser deux protocoles :
  - Le protocole natif JRMP (Java Remote Method Protocol) utrilisé dans ce cours.
  - Le protocol IIOP (Interoperable Internet Object Protocol) de CORBA
- RMI sur IIOP (RMI-IIOP) permet l'interopérabilité avec des objets CORBA (écrits en Java ou dans un autre langage).