**SI4 Polytech'Nice – UNS**

**UE « Applications Réparties », Partie « Objets » - Epreuve écrite individuelle**

23 Mai 2013, durée 1h

**Documents cours/TP autorisés. Comme d'habitude, il est conseillé de lire le sujet en entier avant de commencer !**

F. Baude

## Exercice 1 (barème approximatif : 10 points)

Le but de cet exercice est, grâce à une série de questions de synthèse, d'évaluer votre compréhension profonde du système Java RMI. Quelques lignes de réponse (4 / 5 maximum) par question.

1. Quelle est la nature d'une référence sur un objet serveur RMI, et comment (sous quelle forme) peut-on transmettre de telles références entre objets Java..

2. Quand on veut sécuriser une application RMI, il y a plusieurs aspects complémentaires à traiter. Comment les outils étudiés en cours/TP, que sont Jaas et les sockets RMI sur SSL se complètent et adressent certains de ces aspects ?

3. Expliquez pourquoi, dans le contexte d'une application répartie à objets (Java RMI, Corba), on a besoin d'un service de nommage. Expliquez le rôle d'un tel service. Peut-on s'en passer totalement ? Est-il pour autant nécessaire tout le temps ?

4. Si vous avez pris la peine de réaliser le TP CORBA jusqu'au bout, vous pouvez comparer la façon dont Java RMI d'une part et CORBA d'autre part permettent d'échanger des copies d'objets. Comparer cela au niveau de la spécification / définition des interfaces distantes, mais également concernant le support à runtime (ce qu'il se passe réellement) pour effectuer ces échanges.

5. Le téléchargement de code depuis des URLs http (et pas forcément depuis un répertoire du système de fichiers) est permis par Java. Sous quelles conditions néanmoins ?
   Ce mécanisme est donc utilisable par la plateforme Java RMI. Rappeler dans quelles circonstances ce mécanisme est profitable à une application Java RMI qui, par nature même, met en œuvre plus d'une JVM.

## Exercice 2 (barème approximatif : 5 points)

Dans cet exercice, l'objectif est de vous faire concevoir une application RMI (dit autrement, un objet serveur RMI) dont le rôle est de reproduire ce que réalise le rmiregistry. On supposera que le JDK a une version assez récente, donc, on n'est pas obligé de générer à la main, par rmic, les stubs.

1. Quelles fonctionnalités doivent être exposées par l'objet serveur « rmiregistry ». Quels sont les différents types de clients qui peuvent avoir besoin d'interagir avec cet objet serveur. Préciser l'interface RMI à exposer.
2. Comment est-il possible que cet objet serveur « rmiregistry » soit générique, c'est-à-dire, soit utilisable par des clients dont on n'a a priori aucune idée (pas de connaissance à l'avance des classes Java utilisées dans le code de ces clients). Indice : les objets que manipule (prend et rend en paramètre de retour) l'objet rmiregistry via les méthodes qu'il expose, implantent une interface qui elle-même étend Serializable : ces objets sont donc eux-mêmes sérialisables
3. Pour concrétiser les éléments de réponse fournis aux questions 1 et 2, expliquer via du pseudo code comment les fonctionnalités exposées par l'objet rmiregistry doivent être implantées. Puis, décrire ce qu'il va se passer concrètement lors de l'exécution lorsque les différents types de clients vont utiliser cet objet rmiregistry.

## Exercice 3 (barème approximatif : 5 points) : l'idée du Smart Proxy, un proxy(stub) RMI optimisé!

L'objectif de cet exercice est de réaliser une optimisation dans un cas particulier d'appels d'une méthode RMI. Supposons qu'un objet serveur expose deux méthodes, dans une seule interface appelée **IDistante**. La première méthode **m1** permet à l'appelant de déclencher un vrai travail coté serveur. Au contraire de la seconde méthode **m2** qui n'est en fait qu'un accesseur : cette méthode renvoie juste la valeur d'un attribut de l'objet serveur, et de plus, cet attribut est immuable. C'est-à-dire que si on invoque plusieurs fois cette seconde méthode, la valeur récupérée par l'appelant sera toujours la même.

L'idée de l'optimisation que l'on cherche à spécifier et implémenter est d'économiser de la bande passante réseau dans le cas où l'appelant appellerait une ou plusieurs fois la méthode m2. Il n'est en effet pas nécessaire à l'appelant de réellement déclencher un appel de la méthode m2 via le réseau, si on a fait en sorte que la réponse immuable de la méthode m2 soit déjà disponible localement dans l'objet appelant ! Mais attention, nous voulons que tout reste transparent du point de vue de l'objet appelant : si le code du client consiste à appeler la méthode m2 de IDistante sur l'objet serveur, et à récupérer une valeur de retour, il faut que le code du client invoque bien la méthode m2 de la même interface IDistante, sauf que cette invocation déclenchera une exécution locale de m2 et non remote. Pour que l'appel soit local et non distant, il faudra modifier certaines parties du code, pour que le résultat de l'appel à m2 soit disponible localement sur le client sans qu'il ait besoin de déclencher un appel RMI vers m2 sur le serveur.

Pour finaliser le codage de l'optimisation souhaitée, étudier avec soin les codes joints.

Voici les interfaces RMI nécessaires coté serveur et client

```
import java.rmi.RemoteException;

public interface IDistante extends java.rmi.Remote{
	Res m1() throws RemoteException; // declenche un vrai travail dont le resultat revient sous
forme d'objet serialisé res
	String m2() throws RemoteException;  // renvoie juste la chaine immutable
}
```

```
public interface IAcces extends java.rmi.Remote {
	IDistante getAcces () throws java.rmi.RemoteException;
}
```

```
public class Res implements java.io.Serializable {
	int num;
	public void incrRes() {
			num ++; // pour simuler un vrai travail, on invoquera incrRes
	}
	public String toString() {
		return "l entier est " + num;
	}
}
```

Voici le code du client, qui ne change pas, que l'optimisation soit utilisée ou non

```
public class Client {
	public static void main(String[] args) {
		try {
			IAcces acces = (IAcces)java.rmi.Naming.lookup("rmi://localhost/Serveur");
			IDistante dist=(IDistante) acces.getAcces();
			System.out.println(dist.m1()); // on doit voir s'afficher 1 sur console client
			System.out.println(dist.m1()); // on doit voir s'afficher 2 sur console client
			System.out.println(dist.m2()); // on doit doit coucou sur console client
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Voici le code du Serveur lorsque l'optimisation est activée. Vous remarquez qu'on ne renvoie pas directement une référence RMI vers ServiceMetier, mais, un objet sérialisé de la classe SmartProxyServiceMetier

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Serveur extends UnicastRemoteObject implements IAcces{

	ServiceMetier sm ;
	protected Serveur() throws RemoteException {
		super();
	}
	public static void main(String[] args) throws java.rmi.RemoteException {
		Serveur serv = new Serveur();
		System.out.println("Referencement dans le registry...");
		try {
			LocateRegistry.createRegistry(1099);
			Naming.rebind("rmi://localhost/Serveur", serv);
		} catch (MalformedURLException e) {
			e.printStackTrace();
		}
	}

	public IDistante getAcces() throws RemoteException {
		 sm = new ServiceMetier(); // demarre vraiment le service metier, qui saura repondre aux
requetes clientes
		 SmartProxyServiceMetier smartproxysm = new SmartProxyServiceMetier();
		 smartproxysm.intDistante=sm; //pour passer au client la bonne reference RMI sur le
service metier, afin qu'il puisse invoquer m1 en RMI
		 smartproxysm.reponseM2=sm.m2(); // met d'ores et deja la reponse immuable de la methode
m2 pour eviter les appels RMI de m2 et les faire en local
		return (IDistante) smartproxysm;
	}
}
```

Voici le code de ServiceMetier, qui reste inchangé que l'optimisation soit implantée ou non

```
import java.rmi.RemoteException;
public class ServiceMetier extends java.rmi.server.UnicastRemoteObject implements IDistante {
	Res res;

	protected ServiceMetier() throws RemoteException {
		super();
		res=new Res();
	}

	public Res m1() throws RemoteException {
		System.out.println("voici la thread qui exécute cette méthode m1 " +
Thread.currentThread().getName());
		res.incrRes(); // chaque fois que m1 est invoquée, on simule un travail coté
ServiceMetier, en incrémentant res.
		return res; // et on renvoie res comme réponse à l'appel de m1
	}

	public String m2() throws RemoteException  {
		System.out.println("voici la thread qui exécute cette méthode m2 " +
Thread.currentThread().getName());
		return "coucou"; // on remarque qu'on renvoie toujours la meme valeur (immuable)
	}
}
```

Question 1 : donner le code de getAcces() dans Serveur, dans le cas sans optimisation. Puis toujours dans ce cas sans optimisation, dites ce qui s'affiche dans la console Serveur et dans la console Client suite à l'exécution de leur méthode main respective : en particulier dites précisément ce qu'affichent les lignes `voici la thread qui exécute cette méthode mX " + Thread.currentThread().getName())` et sur quelles consoles. Pour cette question et pour la suivante, afin de donner les traces sur les consoles du

lancement du Serveur, vous supposerez que la thread déclenchée par le système RMI sur ce serveur a un nom de la forme « `RMI TCP Connection(4)- xxx.xxx.xxx.xxx` ». Par contre, si une méthode correspond à un appel en local, la thread s'appelle `main`

Question 2 : Il reste donc à remplir le code de la classe SmartProxyServiceMetier pour finaliser l'optimisation. Et vous redonnerez les traces qu'affichent le Serveur et le Client, suite au déclenchement de leur méthode main respective. En faisant ces traces, vous en profiterez pour expliquer, avec vos mots, ce que vous avez compris : quel est finalement le principe du smart proxy.

```
import java.io.Serializable;
import java.rmi.RemoteException;

public class SmartProxyServiceMetier implements IDistante, Serializable {

A COMPLETER

}
```