iUT *ORSAY*
Département Informatique

DUT Informatique
Module JAVA Apprentis
2008 / 2009

Travaux Pratiques n° 7 : RMI

Nom(s) :
Groupe :
Date :

Objectifs : savoir créer des applications client-serveur mettant en jeu des machines virtuelles distantes, les communications reposant sur les Remote Method Invocations (RMI) de Java.

1 Communications réseau en Java

Java a été conçu dès ses origines pour réaliser des applications communiquantes. Il offre pour cela diverses possibilités :

– Les **sockets** sont l'interface standard pour les communications avec le protocole TCP. Elles ont été introduites à l'origine dans le monde Unix et sont à présent un standard de fait. Dans ce mode de communication (au travers des classes `Socket` et `ServerSocket`), le protocole TCP assure la gestion de la connexion, l'envoi, la réception, le séquencement et l'assemblage des paquets transmis. L'établissement de la connexion et la mise en forme ou l'extraction des données envoyées ou reçues sont à la charge du programmeur (ces dernières opérations sont largement facilitées par les classes `ObjectInputStream` et `ObjectOutputStream`).

– Les **servlets** sont en quelque sorte des serveurs applets qui tirent parti du protocole HTTP en acceptant des requêtes `GET` ou `GET/POST` pour renvoyer en général leurs résultats sous la forme d'une page HTML, mais elles peuvent être utilisés pour renvoyer n'importe quel type de données.

– Trois différents protocoles permettent à des applications clientes d'invoquer des méthodes d'objets s'exécutant sur un serveur distant. CORBA (Common Object Request Broker Architecture) et SOAP (Simple Object Access Protocol) offrent une large interopérabilité grâce à des langages normalisés (des applications C++ exécutées depuis une plateforme Windows/x86 peuvent communiquer en toute transparence avec d'autres écrites en Java sur une plateforme Solaris/UltraSparc). Enfin, Java **RMI** est présenté plus en détail en Section 2.

2 Remote Method Invocation

Java RMI (pour Remote Method Invocation) permet comme CORBA ou SOAP l'exécution de méthodes sur des objets distants. À la différence de ces protocoles pensés pour l'interopérabilité entre langages, les RMI ont été développées spécifiquement pour Java et offrent ainsi le moyen le plus élégant de communiquer dans ce langage.

La complexité de la communication entre applications distances vient d'une part du problème de l'établissement de la communication et d'autre part de la mise en forme des informations à transmettre (par exemple les paramètres de méthodes peuvent être de types différents, en

nombre différent). Pour simplifier ces tâches, Java les automatise au maximum. Le travail d'établissement de connexion, d'envoi et de réception d'informations encodées est effectué par des objets jouant le rôle de proxy : le *stub* (ou *souche* en français) côté client et le *skeleton* (ou *squelette* en français) côté serveur. Le principe de communication par RMI est illustré en Figure 1. Ce processus assez complexe est transparent pour le programmeur, en particulier les objets stub et skeleton sont crées et gérés automatiquement. Au final, après avoir récupéré leurs références, on utilise les objets distants aussi aisément que s'ils étaient locaux.

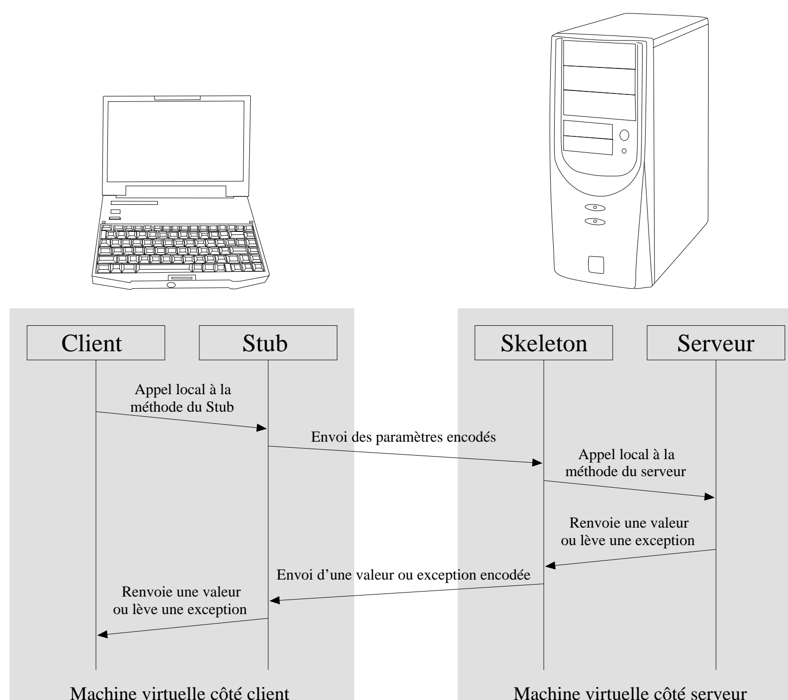

FIG. 1 – Communications lors d'un appel de méthode d'un objet distant avec Java RMI

2.1 Interface d'un objet communiquant

Toutes les méthodes qui pourront être invoquées de manière distante doivent être définies au préalable dans une interface qui hérite de `java.rmi.Remote`. Cet héritage oblige toutes les méthodes définies dans l'interface à déclarer qu'elles sont susceptibles de renvoyer l'exception `java.rmi.RemoteException`. Cette interface sera indispensable à l'objet exécuté sur le serveur qui devra l'implémenter. Elle est aussi nécessaire au stub et au skeleton, mais cet aspect est transparent pour le programmeur.
Le code suivant correspond à une interface déclarant une seule méthode `getMessage` qui pourra être invoquée de manière distante :

```
import java.rmi.*;

/**
 * Classe MessageInterface déclarant les prototypes de méthodes qui
 * pourront être invoquées à distance.
 */
public interface MessageInterface extends Remote {
        public String getMessage() throws RemoteException;
}
```

2.2 Objet communiquant

Pour réaliser un objet dont certaines méthodes pourront être invoquées à distance, il suffit d'une part que sa classe hérite de `java.rmi.server.UnicastRemoteObject` (il y a d'autres possibilités mais qui ne seront pas abordées dans ce TP) et d'autre part qu'il implémente une interface qui hérite de `java.rmi.Remote` où sont définies les méthodes invoquables à distance (voir Section 2.1). Son constructeur (ainsi que toutes les méthodes définies dans la ou les interfaces héritant de `java.rmi.Remote`) doit déclarer l'exception `java.rmi.RemoteException`.
Le code suivant correspond à un objet `Message` destiné à véhiculer une simple chaîne de caractères et possédant une seule méthode `getMessage` qui pourra être invoquée de manière distante :

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

/**
 * Classe Message destinée à véhiculer des chaînes de caractères. Sa
 * méthode getMessage peut être invoquée à distance.
 */
public class Message extends UnicastRemoteObject implements MessageInterface {
        private String message;

        public Message(String message) throws RemoteException {
                this.message = message;
        }

        public String getMessage() throws RemoteException {
                return message;
        }
}
```

2.3 Serveur

Le rôle d'une application serveur est tout d'abord de créer et d'installer un gestionnaire de sécurité par un appel à la méthode `System.setSecurityManager`. C'est un aspect fondamental car il est toujours délicat d'accepter une connexion et d'échanger des informations avec une machine distante. Le gestionnaire de sécurité tiendra compte d'un fichier de configuration qu'on lui précisera au moment de la compilation (voir Section 2.5). Nous utiliserons un fichier extrêmement laxiste puisqu'il autorise tout, on l'appelera *no.policy* et on le placera au même endroit que les codes source :

```
grant {
        permission java.security.AllPermission;
};
```

Pour une mise en production il sera nécessaire de consulter la documentation Java pour créer un fichier n'autorisant que le strict nécessaire, mais il conviendra dans le cadre de ce TP.
Pour mettre à disposition un objet à des clients, le serveur doit d'une part créer cet objet et d'autre part l'enregistrer auprès du serveur de nom Java. Cette dernière opération se réalise par un appel à la méthode `Naming.rebind`. Le premier argument de cette méthode est le nom symbolique de l'objet, de la forme `//machine:port/nom` où `machine` est le nom ou l'adresse IP du serveur, `port` est le port d'accès (optionnel, par défaut 1099) et `nom` est le nom qu'on choisit de donner à l'objet sur ce serveur. Le second argument est une référence à l'objet.
Le code suivant est celui d'un serveur créant et mettant à disposition un objet de type `Message` :

```
import java.rmi.*;

/**
 * Classe Serveur mettant en place des objets pour effectuer des RMI.
 */
public class Serveur {
	private static final String IP_SERVEUR = "192.168.0.1"; // À adapter !
	private static final String NOM_SERVICE = "SMessage";

	public static void main(String[] args) {
		// Création et installation du gestionnaire de sécurité.
		System.setSecurityManager(new RMISecurityManager());

		try {
			// Création et enregistrement d'un objet Message
			Naming.rebind("//" + IP_SERVEUR + "/"+ NOM_SERVICE,
					new Message("hello, world"));
			System.out.println("Serveur OK !");
		}
		catch(Exception e) {
			System.out.println("Serveur KO : " + e.getMessage());
		}
	}
}
```

2.4 Client

Le rôle d'une application cliente est tout d'abord de créer et d'installer un gestionnaire de sécurité de la même manière que pour un serveur (voir Section 2.3).
Pour travailler avec un objet distant, le client doit récupérer une référence à cet objet. Cela est réalisé grâce à la méthode `Naming.lookup` qui prend en argument le nom symbolique de l'objet (voir Section 2.3) et qui lance les communications nécessaires (transparentes pour le programmeur) entre le stub et le skeleton pour trouver cette référence. Le client ne connaissant pas la définition de l'objet distant (il ne dispose pas de son *.class*) mais seulement l'interface qu'il implémente, on utilisera le polymorphisme de Java pour ranger la référence de l'objet dans une référence à une classe implémentant l'interface.
Voici le code d'un client invoquant la méthode `getMessage` d'un objet `Message` distant :

```
import java.rmi.*;

/**
 * Classe Client utilisant des methodes d'un objet distant pour travailler.
 */
public class Client {
	private static final String IP_SERVEUR = "192.168.0.1"; // À adapter !
	private static final String NOM_SERVICE = "SMessage";

	public static void main(String[] args) {
		// Création et installation du gestionnaire de sécurité.
		System.setSecurityManager(new RMISecurityManager());

		try {
			// Récupération d'une référence à un Message distant
			MessageInterface m = (MessageInterface)Naming.lookup(
						"//"+IP_SERVEUR+"/"+NOM_SERVICE);
			// On peut à présent faire appel à ses méthodes.
			System.out.println(m.getMessage());
		}
		catch(Exception e) {
			System.out.println("Client KO : " + e.getMessage());
		}
	}
}
```

Les section suivantes décrivent la mise en place d'une application simple utilisant les Java RMI pour communiquer. Les codes source sont ceux des sections précédentes. Suivez avec une attention particulière l'ordre et les indications pour construire cette application. Eclipse a besoin d'un plugin pour gérer les RMI, ne l'utilisez que pour éditer votre code si vous le désirez, **mais pas pour l'exécuter**.

2.5 Compilation d'une application communiquante

2.5.1 Compilation du serveur

Le serveur possède tous les fichiers décrits précedemment (interfaces, objets distants, serveur, fichier de politique de sécurité) mais pas celui du client. On compile le serveur fourni en exemple dans les sections précéntes par la commande suivante, dans le répertoire où se trouvent les fichiers source (à adapter si vous avez fait des packages) :

```
javac MessageInterface.java Message.java Serveur.java
```

Utiliser `javac.exe` sous Windows, et si cette commande n'est pas dans le PATH, on utilisera son chemin complet de la forme `C:\Program Files\Java\jdkx.x.x\bin\javac.exe`.

De manière générale, et sauf indication contraire, ajoutez simplement le suffixe *.exe* aux commandes Linux si vous travaillez sous Windows et utilisez le chemin complet si nécessaire (toutes les commandes Java se trouvent dans le répertoire du JDK donné ci-avant).

Ensuite on va créer automatiquement les stub et skeleton par la commande :

```
rmic -keep Message
```

L'option `-keep` demande de conserver les sources Java que l'on pourra regarder. En particulier cette commande devrait créer un fichier *Message_Stub.class*. Attention si vous avez fait un package `monPackage`, il faudra exécuter cette commande depuis le répertoire au dessus du package et utiliser le nom complet de la classe : `rmic -keep monPackage.Message`.

2.5.2 Compilation du client

La compilation du programme client requiert les sources du client, un fichier de politique de sécurité, mais aussi la ou les interfaces implémentées par les objets distants dont il aura besoin et pour chacun d'entre eux le fichier *.class* du stub correspondant. Dans une utilisation propre, le fournisseur d'accès à un serveur met à disposition à une adresse accessible par Internet une archive *jar* (voir TP sur les applets) contenant les interfaces et les stubs des objets qu'on peut invoquer de manière distante. Pour ce TP, on se contentera de **copier les *.class* de l'interface et du stub auprès du code du client**. On compilera ensuite le client par la commande (à adapter si vous avez fait des packages) :

```
javac Client.java
```

2.6 Lancement d'une application communiquante

2.6.1 Lancement du serveur

Avant de lancer le serveur, deux opérations sont importantes. Tout d'abord il faut s'assurer que les classes de stub/skeleton soient dans le CLASSPATH (cette variable d'environnement indique à Java où chercher toutes les classes dont il aurait besoin) :

Sous Linux :
`export CLASSPATH=`chemin_vers_les_classes

Sous Windows :
`set CLASSPATH=`chemin_vers_les_classes

La seconde opération est de lancer le serveur de noms Java, qui permet aux clients d'obtenir la référence à un objet à l'aide de son nom symbolique, par la commande :

Sous Linux :
`rmiregistry &`

Sous Windows (attention à bien lancer la version de `rmiregistry` correspondant à votre dernier JDK et non une autre, s'il y a un problème par exemple *unmarshalling error*, **utilisez le nom complet jusqu'à l'exécutable correct**) :
`start rmiregistry`

Enfin, on peut lancer le serveur par la commande :

```
java -Djava.security.policy=no.policy Serveur
```

2.6.2 Lancement du client

Le lancement du client est plus simple et se fait par la commande :

```
java -Djava.security.policy=no.policy Client
```

Réalisez et exécutez cette application. Décrivez en quelques phrases et/ou par un schéma votre compréhension des échanges entre le client et le serveur au cours de l'exécution de cette application.

3 Estimation de π

Le but de ce problème est de construire une application client-serveur utilisant Java RMI comme moyen de communication pour trouver une approximation du nombre π.
La méthode utilisée pour calculer une valeur approchée de π, appelée méthode de Monte-Carlo, est d'effectuer des tirages aléatoires de nombres réels x et y tels qu'ils soient compris entre 0 et 1. Si N est le nombre de tirages de couples x et y, et M le nombre de tirages tels que $x^2 + y^2 \leq 1$, alors on a $\pi \simeq (4M)/N$.
Pour ce problème, un programme appelé *pi* va faire des appels à un autre programme appelé *tirage* s'exécutant sur une autre machine pour lui demander un nombre aléatoire entre 0 et 1. Pour cela, *pi* invoquera une méthode distante de *tirage*. Le programme *tirage* calculera alors le nombre et l'enverra en valeur de retour. Quand *pi* aura fait une demande et reçu les réponses pour x et pour y, il calculera si $x^2 + y^2 \leq 1$. Il affichera après chaque demande de x et y son évaluation de π qui devrait être de plus en plus précise.

Décrivez l'organisation des communications entre les deux programmes.

Dans ce schéma, qui est le client et qui est le serveur ?

Réalisez cette application, **vous joindrez le code source au compte-rendu de TP**.

Utilisez votre programme client pour qu'il puisse fonctionner avec le serveur d'un autre binôme. Qu'a-t-il fallu modifier ?

4 Communication bidirectionnelle : `talk`

Lorsqu'il est question d'invocation de méthodes à distance, la distinction entre client et serveur n'a lieu que pour une seule invocation : un client demande un service à un serveur, mais rien n'empêche ensuite les rôles de s'inverser.
La commande `talk` du monde Unix permet à deux utilisateurs de discuter entre eux en s'envoyant des messages textuels (en quelque sorte, il s'agit d'un *chat* limité à deux utilisateurs). On souhaite créer en Java une application *talk* qui peut à la fois envoyer des messages vers une autre application *talk* distante (par transmission d'un message en argument d'une méthode distante qui affichera ce message) et recevoir des messages (en proposant de la même façon un objet dont on pourra invoquer des méthodes à distance et qui affichera les messages reçus).

Réalisez cette application dont **vous joindrez le code source au compte-rendu de TP**. Commencez par une application purement en mode texte, puis utilisez le temps qu'il vous restera pour lui confectionner une interface graphique. Pensez à strictement décomposer moteur de communication et interface graphique.