# Programmation RMI

Ph. Truillet

Septembre 2018– v. 2.0

## 0. déroulement du TP

RMI (**R**emote **M**ethod **I**nvocation) est une technologie développée et fournie par Sun à partir du JDK 1.1 pour permettre de mettre en oeuvre facilement des objets distribués.

Dans ce TP, vous aurez à :

- exécuter un exemple illustrant l'architecture de RMI
- développer un service d'annuaire avec RMI

## 1. introduction

RMI permet l'appel d'une méthode appartenant à un objet distant, c'est à dire gérée par une autre JVM (Java Virtual Machine) que la JVM locale : il devient ainsi possible à des applications dites clientes (s'exécutant localement) d'invoquer des méthodes sur des objets distants localisés dans une application appelée « *serveur* »

Les objectifs de RMI sont de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être lancée sur une machine différente pourvu qu'elle soit accessible par le réseau.

L'appel coté client d'une méthode distant est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

***Principe*** :
Le mécanisme lié à RMI permet à une application s'exécutant sur une machine M1 de créer un objet et de le rendre accessible à d'autres applications exécutées dans une autre JVM : cette application et la machine M1 jouent donc le rôle de serveur.

Les autres applications manipulant un tel objet sont des clients. **Pour manipuler un objet distant, un client récupère sur sa machine une représentation de l'objet** appelé aussi talon ou souche (*stub*) : ce talon implémente l'interface (au sens Java du terme) de l'objet distant et c'est via ce talon que le client pourra invoquer des méthodes sur l'objet distant.
Une telle invocation sera transmise au serveur (grâce au protocole TCP) afin d'en réaliser l'exécution (cf. figure 1).

Du côté du serveur un *skeleton* (squelette) a en charge la réception des invocations distantes, leur réalisation et le renvoi des résultats.

La technologie RMI se charge ainsi de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

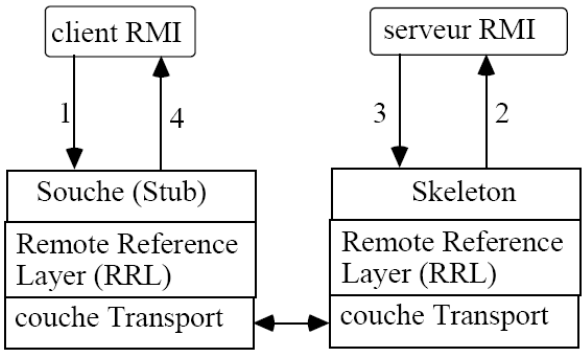



**Figure 1** : mode de fonctionnement de RMI

Les deux classes particulières, le *stub* et le *skeleton*, sont générées avec l'outil `rmic` fourni avec le JDK. Comme expliqué plus haut, le *stub* est une classe qui se situe côté *client* et le *skeleton* est son homologue coté *serveur*. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoi et de réception du résultat.

# 2. étapes pour créer un objet distant et l'appeler avec RMI

Le développement coté serveur se compose de :

- la définition d'une interface qui contient les méthodes qui peuvent être appelées à distance
- l'écriture d'une classe qui implémente cette interface
- l'écriture d'une classe qui instanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de nom RMI (RMI Registry)

Enfin, il faut générer les classes *stub* et *skeleton* en exécutant le programme **`rmic`** avec le fichier source de l'objet distant

Le développement côté client se compose de :

**l'obtention d'une référence sur l'objet distant à partir de son nom**
**et de l'appel à la méthode à partir de cette référence**

# 3. le développement coté serveur

## *3.1. définition d'une interface qui contient les méthodes de l'objet distant*

L'interface à définir doit hériter de l'interface `java.rmi.Remote`. Cette interface ne contient aucune méthode mais indique simplement que l'interface peut être appelée à distance.

L'interface doit contenir toutes les méthodes qui seront susceptibles d'être appelées à distance.
La communication entre le client et le serveur lors de l'invocation de la méthode distante peut échouer pour diverses raisons telles qu'un crash du serveur, une rupture de la liaison, etc.

Ainsi chaque méthode appelée à distance doit déclarer qu'elle est en mesure de lever l'exception `java.rmi.RemoteException`.

## *3.2. écriture d'une classe qui implémente cette interface*

Cette classe correspond à l'objet distant. Elle doit donc implémenter l'interface définie et contenir le code nécessaire.
Cette classe doit obligatoirement hériter de la classe `UnicastRemoteObject` qui contient les différents traitements élémentaires pour un objet distant dont l'appel par le stub du client est unique. Le stub ne peut obtenir qu'une seule référence sur un objet distant héritant de `UnicastRemoteObject`.

La hiérarchie de la classe `UnicastRemoteObject` est :

- `java.lang.Object`
- `java.rmi.Server.RemoteObject`
- `java.rmi.Server.RemoteServer`
- `java.rmi.Server.UnicastRemoteObject`

Comme indiqué dans l'interface, toutes les méthodes distantes doivent indiquer qu'elles peuvent lever l'exception `RemoteException` ainsi que le constructeur de la classe. Ainsi, même si le constructeur ne contient pas de code, il doit être redéfini pour inhiber la génération du constructeur par défaut qui ne lève pas cette exception.

## *3.3. écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre*

Ces opérations peuvent être effectuées dans la méthode `main` d'une classe dédiée ou dans la méthode main de la classe de l'objet distant. L'intérêt d'une classe dédiée et qu'elle permet de regrouper toutes ces opérations pour un ensemble d'objets distants.
La marche à suivre contient trois étapes :

- la mise en place d'un security manager dédié (facultatif)
- l'instanciation d'un objet de la classe distante
- l'enregistrement de la classe dans le registre de nom RMI en lui donnant un nom

### 3.3.1. mise en place d'un security manager

Cette opération n'est pas obligatoire mais elle est recommandée en particulier si le serveur doit charger des classes qui ne sont pas sur le serveur. Sans security manager, il faut obligatoirement mettre à la disposition du serveur toutes les classes dont il aura besoin (Elles doivent être dans le `CLASSPATH` du serveur). Avec un security manager, le serveur peut charger dynamiquement certaines classes.
Cependant, le chargement dynamique de ces classes peut poser des problèmes de sécurité car le serveur va exécuter du code d'une autre machine. Cet aspect peut ainsi conduire à ne pas utiliser de security manager.

Vous pouvez spécifier une politique de sécurité au lancement du serveur en utilisant l'option **-D** :

```
java -Djava.security.policy= policyfilename serveur
```

La syntaxe du fichier de sécurité est décrite dans **docs/technotes/guides/security/PolicyFiles.html** de la distribution JDK.
Voici un exemple de fichier de sécurité qui autorise toutes les actions !

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

On peut aussi n'autoriser que certaines opérations comme :

```
permission java.net.SocketPermission "*:8080", "connect, accept";
```

### 3.3.2. instanciation d'un objet de la classe distante

Cette opération est très simple puisqu'elle consiste simplement en la création d'un objet de la classe de l'objet distant.

### 3.3.3. enregistrement dans le registre de nom RMI en lui donnant un nom

La dernière opération consiste à enregistrer l'objet créé dans le registre de nom en lui affectant un nom. Ce nom est fourni au registre sous forme d'une URL constitué du préfixe `rmi://`, du nom du serveur (*hostname*) et du nom associé à l'objet précédé d'un slash.
Le nom du serveur peut être fourni « en dur » sous forme d'une constante chaîne de caractères ou peut être dynamiquement obtenu en utilisant la classe `InetAddress` pour une utilisation en local.

C'est ce nom qui sera utilisé dans une URL par le client pour obtenir une référence sur l'objet distant.
L'enregistrement se fait en utilisant la méthode `rebind` de la classe `Naming`. Elle attend en paramètre l'URL du nom de l'objet et l'objet lui même.

### 3.3.4. lancement dynamique du registre de nom RMI

Sur le serveur, le registre de nom RMI doit s'exécuter avant de pouvoir enregistrer un objet ou obtenir une référence.
Ce registre peut être lancé en tant qu'application fournie par SUN dans le JDK (**rmiregistry**) comme indiqué plus bas ou être lancé dynamiquement dans la classe qui enregistre l'objet (cf. ci-dessous).

```
/* Démarrer rmiRegistry au niveau du serveur */
try {
        java.rmi.registry.LocateRegistry.createRegistry(1099); // port 1099
        System.out.println("Le registre RMI est lancé.");
}
catch (Exception e) {
        System.out.println("Le registre RMI n'a pas démarré.");
}
```

Ce lancement ne doit avoir lieu qu'une seule et unique fois. Il peut être intéressant d'utiliser ce code si l'on crée une classe dédiée à l'enregistrement des objets distants. Le code pour exécuter le registre est la méthode `createRegistry` de la classe `java.rmi.registry.LocateRegistry`. Cette méthode attend en paramètre un numéro de port.

# 4. développement coté client

L'appel d'une méthode distante peut se faire dans une application ou dans une applet.

## 4.1. mise en place d'un security manager

Comme pour le coté serveur, cette opération est facultative. Le choix de la mise en place d'un security manager côté client suit des règles identiques à celui du côté serveur. Sans son utilisation, il est nécessaire de mettre dans le `CLASSPATH` du client toutes les classes nécessaires dont la classe *stub*.

## 4.2. obtention d'une référence sur l'objet distant à partir de son nom

Pour obtenir une référence sur l'objet distant à partir de son nom, il faut utiliser la méthode statique `lookup`() de la classe `Naming`.
Cette méthode attend en paramètre une URL indiquant le nom qui référence l'objet distant. Cette URL est composée de plusieurs éléments : le préfixe `rmi://`, le nom du serveur (*hostname*) et le nom de l'objet tel qu'il a été enregistré dans le registre précédé d'un slash.
Il est préférable de prévoir le nom du serveur sous forme de paramètres de l'application ou de l'applet pour plus de souplesse.

La méthode `lookup()` va rechercher dans le registre du serveur l'objet et retourner un objet stub. L'objet retourné est de la classe `Remote` (cette classe est la classe mère de tous les objets distants).

Si le nom fourni dans l'URL n'est pas référencé dans le registre, la méthode lève l'exception `NotBoundException`.

### 4.3. appel à la méthode à partir de la référence sur l'objet distant

L'objet retourné étant de type `Remote`, il faut réaliser un casting vers l'interface qui définit les méthodes de l'objet distant. Pour plus de sécurité, on vérifie que l'objet retourné est bien une instance de cette interface.
Un fois le casting réalisé, il suffit simplement d'appeler la méthode.

### 4.4. appel d'une méthode distante dans une applet

L'appel d'une méthode distante est la même dans une application et dans une applet.
Seule la mise en place d'un *security manager* dédié dans les applets est inutile car elles utilisent déjà un security manager (`AppletSecurityManager`) qui autorise le chargement de classes distantes.

## 5. génération des classes stub et skeleton

Pour générer ces classes, il suffit d'utiliser l'outil rmic fourni avec le JDK en lui donnant en paramètre le nom de la classe. La classe doit avoir été compilée ; rmic a besoin du fichier `.class`.

**Nota** : **sous Eclipse, cette étape est effectuée automatiquement** !

***Exemple*** :

```
rmic -cp . -keepgenerated HelloServeur
REM garde les fichiers stub et skeleton générés
```

## 6. mise en oeuvre des objets RMI

La mise en oeuvre et l'utilisation d'objet distant avec RMI nécessite plusieurs étapes :

- démarrer le registre RMI sur le serveur soit en utilisant le programme **`rmiregistry`** livré avec le JDK soit en exécutant une classe qui effectue le lancement.
- exécuter la classe qui instancie l'objet distant et l'enregistre dans le serveur de nom RMI
- lancer l'application ou l'applet pour tester.

### 6.1. lancement du registre RMI

La commande `rmiregistry` est fournie avec le JDK. Il faut la lancer en tâche de fond :

- Sous Unix : `rmiregistry&`
- Sous Windows : **`start`** `rmiregistry`

Ce registre permet de faire correspondre un objet à un nom et inversement. C'est lui qui est sollicité lors d'un appel aux méthodes `Naming.bind()` et `Naming.lookup()`. Le registre se lance sur le port `1099` par défaut.

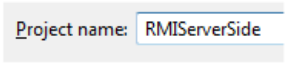
### 6.2. instanciation et l'enregistrement de l'objet distant

Il faut exécuter la classe qui va instancier l'objet distant et l'enregistrer sous son nom dans le registre précédemment lancé.
Pour ne pas avoir de problème, il faut s'assurer que toutes les classes utiles (la classe de l'objet distant, l'interface qui définit les méthodes, le *skeleton*) sont présentes dans un répertoire défini dans la variable **CLASSPATH**.

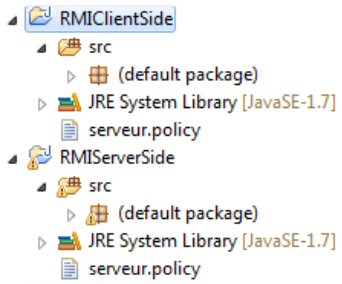
## 7. exercices

Récupérez le fichier RMI.zip à l'adresse :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/RMI.zip`

Créer un projet sous Eclipse (**File** | **New** | **Java Project**)




Ajouter les fichiers situés dans le dossier **`RMIServerSide/src`** de RMI.zip dans votre projet (répertoire **src**). Copier le fichier **`serveur.policy`** dans votre projet.




Lancer votre projet : le serveur est opérationnel.

Effectuer la même opération avec (le nouveau) projet **`RMIClientSide`**
Que se passe-t-il ? Analyser ensuite le code source des parties serveur et client.

**Nota** : Le port utilisé peut être déjà **utilisé** : pensez à le changer si nécessaire.

En s'inspirant de l'exemple précédent, développez un système d'appel de méthodes distantes qui **permette la gestion d'un annuaire type « Pages Jaunes » partagé** en utilisant RMI.

L'annuaire sera composé d'un ensemble de personnes.
Une personne sera définie au moins par un numéro, un nom/prénom et un numéro de téléphone.
Les opérations (*méthodes*) **d'initialisation, d'ajout**, de **suppression** et de **consultation** à partir du nom seront disponibles.

Ecrivez le fichier d'interface java ainsi que la partie serveur et un exemple de client qui utilise l'annuaire

***Nota*** : pensez à sérialiser les objets !

# 8. bibliographie


- http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
- http://cedric.cnam.fr/~farinone/IAGL/rmi.pdf
- http://www.ejbtutorial.com/java-rmi/a-step-by-step-implementation-tutorial-for-java-rmi