

# TD 2 – RMI

Kenneth VANHOEY

<https://dpt-info.u-strasbg.fr/~kvanhoey>

## 1 Rappels

RMI (Remote Method Invocation) permet de proposer à des applications clientes d'invoquer des méthodes sur des objets distants (c'est-à-dire localisés sur une machine distante ou tournant sur une autre JVM), et ce, de façon simple. RMI ne fonctionne qu'en langage Java et est un mécanisme propriétaire (Sun).

### 1.1 Fonctionnement

Pour faire fonctionner ce mécanisme, une interface d'objet est définie et connue par tous (clients et serveur).

Une implémentation de cette interface, que le client ne connaît pas, est placée sur le serveur.

Les mécanismes RMI permettent alors de générer depuis l'implémentation de cette classe un *proxy* ou *stub* (un fichier *.class*) qui sera transmis au client. Ce *stub* gèrera la liaison entre le client et le serveur, à savoir l'emballage (ou la *sérialisation*) des données à transmettre, la transmission sur le réseau (par protocole TCP) et la réception et désérialisation du résultat retourné.

De plus, une classe similaire, appelée *skeleton*, est également générée. Son but est analogue au *stub*, opérant du côté serveur.

**Remarque :** depuis la version 1.6 de Java, le *stub* et le *skeleton* ne sont plus générés explicitement (plus de fichiers créés). Ce mécanisme a été supprimé au profit d'une gestion transparente par les classes de gestion RMI (le *stub* est en fait téléchargé dynamiquement), qui intègrent désormais les fonctionnalités du *stub* et du *skeleton*. Le *stub* peut cependant encore être généré pour des raisons de rétro-compatibilité. Dans ce TD, nous verrons la procédure permettant de le générer.

### Déroulement d'un appel

Lorsqu'un client invoque une méthode sur le *stub* local, l'appel distant se déroule de la manière suivante :

1. le *stub* sérialise l'identifiant de la méthode à invoquer ainsi que les arguments. Cette opération est aussi appelée *marshalling* ;
2. la requête est transmise sur le réseau via le protocole TCP ;
3. sur le serveur, c'est le squelette qui reçoit le message et le désérialise ;
4. la méthode demandée est appelée localement sur le serveur avec les arguments donnés ;
5. le résultat est renvoyé au squelette qui sérialise ce dernier ;
6. le résultat sérialisé est envoyé au *stub* sur la machine cliente ;
7. le *stub* reçoit et désérialise ce message ;
8. finalement, le *stub* transmet le résultat au client.

Pour le client, ce mécanisme est totalement transparent.

## Nommage

Afin de pouvoir identifier un objet particulier, un mécanisme de nommage a été mis en place. Ainsi, un serveur de noms (*rmiregistry*) sera lancé dans un premier temps. Lorsqu'un serveur voudra mettre à disposition un objet, il devra l'enregistrer sur ce serveur de noms, en associant un nom (une chaîne de caractères) à l'objet en question.

Un client souhaitant exploiter l'objet donné, enverra ainsi sa demande au *rmiregistry* (lancé sur une machine et écoutant sur un numéro de port donné) en spécifiant le nom de l'objet qu'il souhaite utiliser. Le serveur de noms fera alors le lien avec le programme proposant cet objet.

## 1.2 Développement & compilation

Le développement et la compilation d'une application RMI se fait en plusieurs étapes. Tout le code qui suit forme un ensemble minimal. Toute classe importée, implémentée ou étendue l'est obligatoirement, toute exception gérée doit l'être.

1. **Développement** de l'interface de l'objet auquel on souhaite pouvoir accéder à distance. C'est l'interface que clients et serveur connaîtront et que le serveur implémentera afin de fournir les services correspondants.

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

public interface <nom_interface> extends Remote // Hérite de Remote
{
    public <type> maMethode([args]*)
        throws RemoteException ;
}
```

Ce fichier est placé sur le serveur et propagé aux clients (ou publié).

2. **Programmation** d'une implémentation des méthodes correspondant à cette interface.

```
import java.rmi.server.UnicastRemoteObject ;
import java.rmi.RemoteException ;
import <nom_interface> ;

public class <nom_impl>
    extends UnicastRemoteObject // Hérite de UnicastRemoteObject
    implements <nom_interface> // implémente l'interface
{
    public <nom_impl> () throws RemoteException
        // Rmq : Le client n'a pas accès au constructeur
    {
        super() ;
    } ;
    public <type> maMethode([args]) throws RemoteException
    {
        <code>
    }
}
```

Cette partie sera placée sur le serveur uniquement. Toute méthode peut lever l'exception *RemoteException* afin d'avertir l'appelant d'une erreur survenue lors d'un appel à distance.

3. **Compilation** de ces classes et **génération** du stub par la commande suivante :

```
rmic <nom_impl>
```

Ceci générera le fichier `<nom_impl_Stub.class>` qui devra être transmis aux clients. Cette étape peut être omise (voir « fonctionnement »).

4. **Développement** d'un programme serveur qui mettra à disposition l'objet.

```

import java.net.* ;
import java.rmi.* ;

public class Serveur
{
    public static void main(String [] args)
    {
        if (args.length != 1)
        {
            System.out.println("Usage : java Serveur <port du serveur de noms>") ;
            System.exit(0) ;
        }
        try
        {
            <nom_impl> objLocal = new <nom_impl> () ;
            Naming.rebind( "rmi://localhost:" + args[0] + "/"<nomObjet>" ,objLocal) ;
            System.out.println("Serveur pret") ;
        }
        catch (RemoteException re)      { System.out.println(re) ; }
        catch (MalformedURLException e) { System.out.println(e) ; }
    }
}

```

Ce programme devra donner un nom à l'objet (<nomObjet> dans le code) afin que le serveur de noms connaisse cet objet et lui associe un identifiant.

Remarquez que les adresses réseau se définissent en RMI par une URL de la forme suivante :

```
rmi://<nom_machine>:<no_port>/<nom_objet>
```

5. **Développement** d'un client qui utilise l'objet en question via des appels à distance.

```

import java.rmi.* ;
import java.net.MalformedURLException ;

public class Client
{
    public static void main(String [] args)
    {
        if (args.length != 2)
        {
            System.out.println(
                "Usage : java Client <machine du Serveur>:<port du rmiregistry>") ;
            System.exit(0) ;
        }
        try
        {
            <nom_interface> stub = (<nom_interface>) Naming.lookup(
                "rmi://" + args[0] + ":" + args[1] + "/"<nomObjet>") ;
            // stub est désormais utilisable comme un objet local.
            stub.maMethode() ;
        }
        catch (NotBoundException re)    { System.out.println(re) ; }
        catch (RemoteException re)     { System.out.println(re) ; }
        catch (MalformedURLException e) { System.out.println(e) ; }
    }
}

```

Le mécanisme RMI permet au client de télécharger le stub (ou de simplement l'utiliser s'il a été transmis manuellement) et d'utiliser celui-ci de façon transparente, c'est-à-dire d'invoquer des

méthodes sur cet objet comme s'il s'agissait de l'objet distant.

6. **Compilation** des programmes serveur et client dès que leur code est terminé. Le client doit avoir accès à l'interface lors de sa compilation. Le serveur doit avoir accès à l'implémentation de l'interface lors de la compilation.
7. Pour l'**exécution**, il faut d'abord lancer le serveur, et pour que le serveur puisse enregistrer le nom de l'objet, le serveur de nom doit préalablement fonctionner :

```
rmiregistry <portnum> &  
java Serveur <portnum>
```

Le serveur doit avoir accès à l'interface lors de l'exécution.

Le client peut alors être exécuté. Il devra spécifier le nom de la machine, le numéro de port ainsi que le nom de l'objet. Dans notre exemple, on l'exécutera avec les arguments suivants :

```
java Client <nom_machine> <num_port>
```

## 2 Exercice

Définissez pas à pas un code mettant à disposition une classe *Couleur* composée d'un triplet RVB (Rouge-Vert-Bleu), puis l'utilisant via un mécanisme RMI client/serveur. Cette classe aura (au moins) deux méthodes renvoyant une chaîne de caractères qui sera utilisée pour afficher la valeur du triplet RVB (utilisé pour la synthèse additive) correspondant à ladite couleur d'une part, la valeur du triplet CMJ (Cyan-Magenta-Jaune, utilisé pour la synthèse soustractive) correspondante<sup>1</sup> d'autre part.

1. Écrivez l'interface couleur, son implémentation, et les lignes de compilation permettant de les compiler et d'envoyer le stub au client ;
2. Écrivez un code d'une application serveur et d'une application cliente ainsi que les lignes de compilation associées ;
3. Écrivez les instructions à exécuter dans le bon ordre afin de faire fonctionner l'interaction RMI.

---

1. Le cyan est défini comme étant l'opposé du rouge, le magenta comme l'opposé du vert et le jaune comme l'opposé du bleu.