Programmation orientée objet en langage JAVA

Java RMI : Programmation répartie en JAVA

Claude Duvallet

Université du Havre
UFR Sciences et Techniques
25 rue Philippe Lebon - BP 540
76058 LE HAVRE CEDEX
Claude.Duvallet@gmail.com
http://litis.univ-lehavre.fr/∼duvallet/

Remote Method Invocation (RMI)

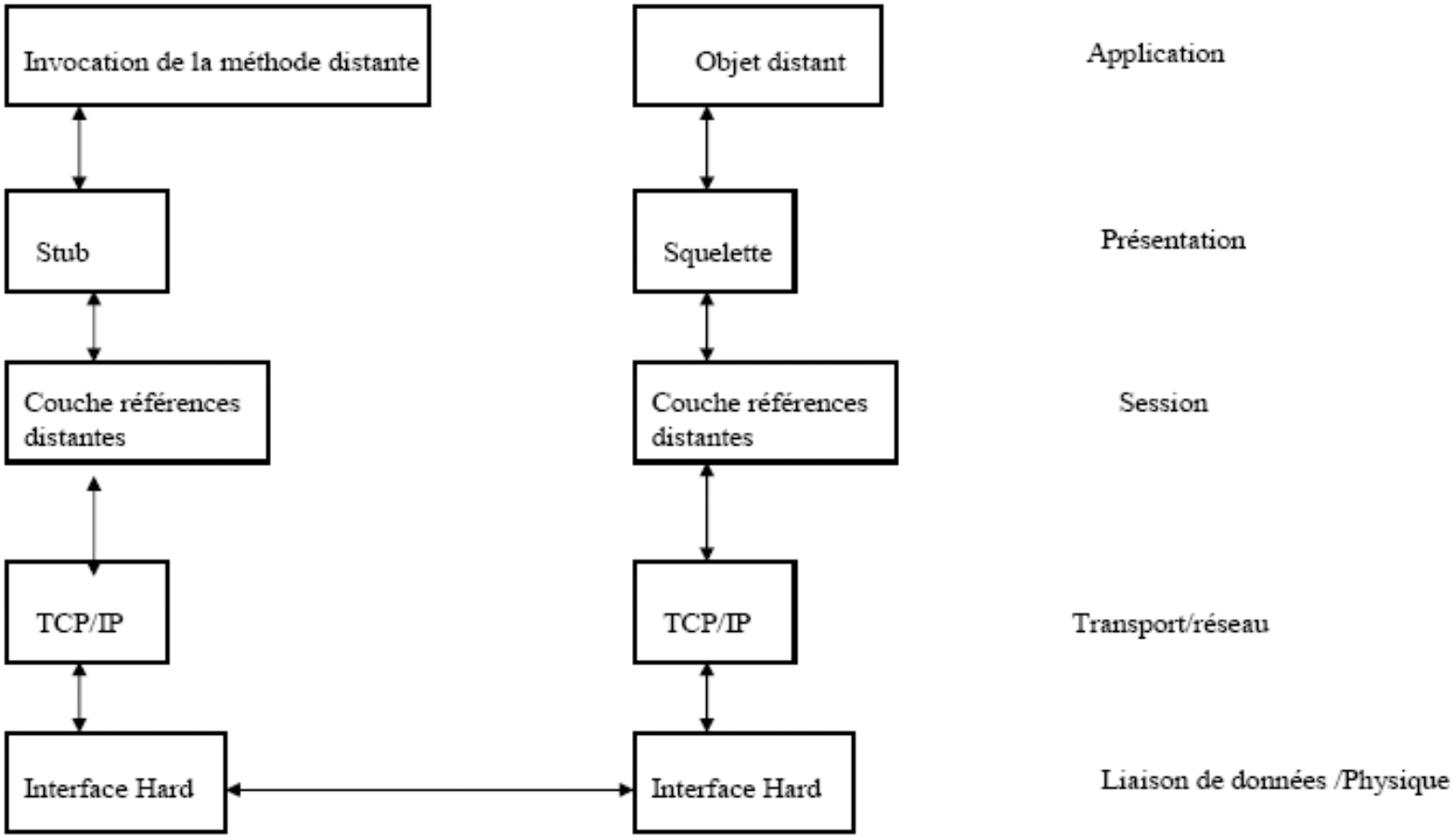
Introduction

- JAVA et les objets distribués.
  - Mise en place d'outils facilitant la distribution d'objets et leur utilisation dans le cadre d'architecture Client/Serveur : RMI.
  - Extension de la notion de programmation réseau avec les sockets UDP et TCP.
- RMI est un système d'objets distribués constitué uniquement d'objets JAVA.
- RMI est une API (Application Programming Interface) intégrée à JAVA depuis la version 1.1.

Principales caractéristiques de Java RMI

- Mécanisme permettant l'appel de méthodes entre des objets JAVA qui s'exécutent éventuellement sur des JVM (Java Virtual Machine) distinctes.
- L'appel peut se faire sur la même machine ou bien sur des machines connectées en réseau.
- Les échanges respectent un protocole propriétaire : Remote Method Protocol.
- RMI repose sur les classes de sérialisation.

Architecture de Java RMI



Les amorces (Stub/Skeleton)

- Elles assurent le rôle d'adaptateurs pour le transport des appels distants.
- Elles réalisent les appels sur la couche réseau.
- Elles réalisent l'assemblage et le désassemblage des paramètres (marshalling, unmarshalling).
- Une référence d'objets distribués correspond à une référence d'amorce.
- Les amorces sont créées par le générateur rmic.

Les stubs

- Représentants locaux de l'objet distribué.
- Initient une connexion avec la JVM distante en transmettant l'invocation distante à la couche des références d'objets.
- Assemblent les paramètres pour leur transfert à la JVM distante.
- Attendent les résultats de l'invocation distante.
- Désassemblent la valeur ou l'exception renvoyée.
- Renvoient la valeur à l'appelant.
- S'appuient sur la sérialisation.

Les squelettes

- Désassemblent les paramètres pour la méthode distante.
- Font appel à la méthode demandée.
- Assemblage du résultat (valeur renvoyée ou exception) à destination de l'appelant.

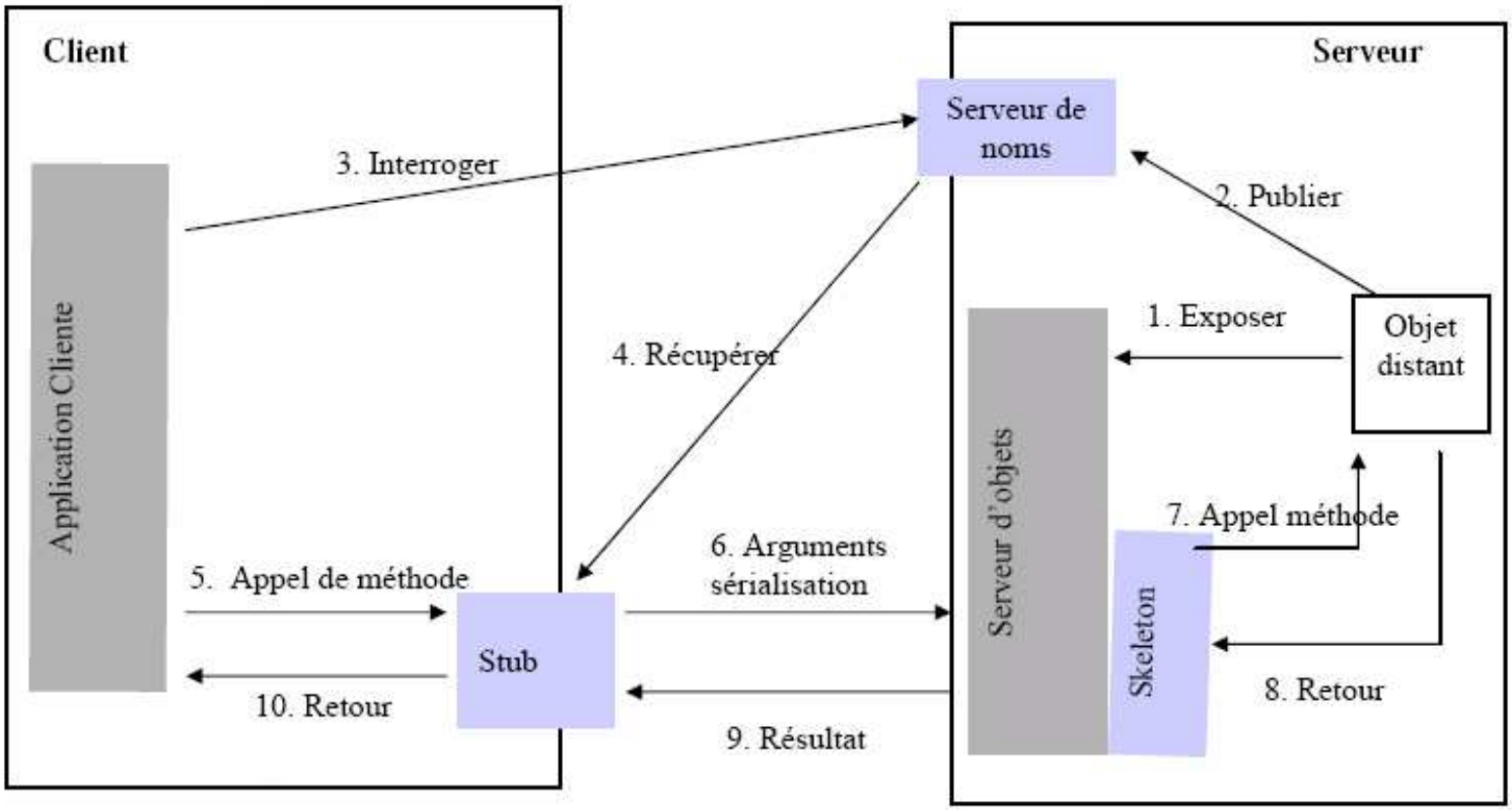
La couche des références d'objets Remote Reference Layer

- Permet d'obtenir une référence d'objet distribué à partir de la référence locale au stub.
- Cette fonction est assurée grâce à un service de noms `rmiregister` (qui possède une table de hachage dont les clés sont des noms et les valeurs sont des objets distants).
- Un unique `rmiregister` par JVM.
- `rmiregister` s'exécute sur chaque machine hébergeant des objets distants.
- `rmiregister` accepte des demandes de service sur le port 1099.

La couche transport

- Elle réalise les connexions réseaux basées sur les flux entre les JVM.
- Elle emploie un protocole de communication propriétaire (JRMP : Java Remote Method Invocation) basé sur TCP/IP.
- Le protocole JRMP a été modifié afin de supprimer la nécessité des squelettes car depuis la version 1.2 de Java, une même classe skeleton générique est partagée par tous les objets distants.

Étapes d'un appel de méthode distante



Développer une application avec RMI : Mise en œuvre

1. Définir une interface distante (`Xyy.java`).
2. Créer une classe implémentant cette interface (`XyyImpl.java`).
3. Compiler cette classe (`javac XyyImpl.java`).
4. Créer une application serveur (XyyServer.java).
5. Compiler l'application serveur.
6. Créer les classes stub et skeleton à l'aide de `rmic XyyImpl_Stub.java` et `XyyImpl_Skel.java` (`Skel` n'existe pas pour les versions >1.2).
7. Démarrage du registre avec `rmiregistry`.
8. Lancer le serveur pour la création d'objets et leur enregistrement dans `rmiregistry`.
9. Créer une classe cliente qui appelle des méthodes distantes de l'objet distribué (`XyyClient.java`).
10. Compiler cette classe et la lancer.

Inversion d'une chaîne de caractères à l'aide d'un objet distribué

Invocation distante de la méthode `reverseString()` d'un objet distribué qui inverse une chaîne de caractères fournie par l'appelant. On définit :

- `ReverseInterface.java` : interface qui décrit l'objet distribué
- `Reverse.java` : qui implémente l'objet distribué
- `ReverseServer.java` : le serveur RMI
- `ReverseClient.java` : le client qui utilise l'objet distribué

Fichiers nécessaires

Côté Client

- l'interface : `ReverseInterface`.
- le client : `ReverseClient`.

Côté Serveur

- l'interface : `ReverseInterface`.
- l'objet : `Reverse`.
- le serveur d'objets : `ReverseServer`.

Interface de l'objet distribué

- Elle est partagée par le client et le serveur.
- Elle décrit les caractéristiques de l'objet.
- Elle étend l'interface `Remote` définie dans `java.rmi`.
- Toutes les méthodes de cette interface peuvent déclencher une exception du type `RemoteException`.
- Cette exception est levée :
  - si connexion refusée à l'hôte distant
  - ou bien si l'objet n'existe plus,
  - ou encore s'il y a un problème lors de l'assemblage ou le désassemblage.

Interface de la classe distante

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface ReverseInterface extends Remote {
  String reverseString(String chaine)
                  throws RemoteException;
}
```

Implémentation de l'objet distribué (1/2)

- L'implémentation doit étendre la classe `RemoteServer` de `java.rmi.server`.
- `RemoteServer` est une classe abstraite.
- `UnicastRemoteObject` étend `RemoteServer`.
  - c'est une classe concrète.
  - une instance de cette classe réside sur un serveur et est disponible via le protocole TCP/IP.

Implémentation de l'objet distribué (2/2)

```
import java.rmi.*;
import java.rmi.server.*;
public class Reverse extends UnicastRemoteObject
                     implements ReverseInterface {
  public Reverse() throws RemoteException {
    super();
  }

  public String reverseString (String ChaineOrigine)
                               throws RemoteException {
    int longueur=ChaineOrigine.length();
    StringBuffer temp=new StringBuffer(longueur);
    for (int i=longueur; i>0; i--) {
      temp.append(ChaineOrigine.substring(i-1, i));
    }
    return temp.toString();
  }
}
```

Le serveur (1/2)

- Programme à l'écoute des clients.
- Enregistre l'objet distribué dans `rmiregistry`
  `Naming.rebind("rmi://hote:1099/Reverse", rev);`.
- On installe un gestionnaire de sécurité si le serveur est amené à charger des classes (inutile si les classes ne sont pas chargées dynamiquement) `System.setSecurityManager(new RMISecurityManager());`

Le serveur (2/2)

```
import java.rmi.*;
import java.rmi.server.*;
public class ReverseServer {
  public static void main(String[] args) {
    try {
      System.out.println( "Serveur : Construction de l'implémentation ");
      Reverse rev= new Reverse();
      System.out.println("Objet Reverse lié dans le RMIregistry");
      Naming.rebind("rmi://localhost:1099/MyReverse", rev);
      System.out.println("Attente des invocations des clients ...");
    }
    catch (Exception e) {
      System.out.println("Erreur de liaison de l'objet Reverse");
      System.out.println(e.toString());
    }
  } // fin du main
} // fin de la classe
```

Le client (1/4)

- Le client obtient un stub pour accéder à l'objet par une URL RMI

```
ReverseInterface ri = (ReverseInterface) Naming.lookup
("rmi://localhost:1099/MyReverse");
```

- Une URL RMI commence par `rmi://`, le nom de machine, un numéro de port optionnel et le nom de l'objet distant.

```
rmi://hote:2110/nomObjet
```

- Par défaut, le numéro de port est `1099` défini (ou à définir) dans `/etc/services` :

```
rmi 1099/tcp
```

Le client (2/4)

- Installe un gestionnaire de sécurité pour contrôler les stubs chargés dynamiquement :

```
System.setSecurityManager(new RMISecurityManager());
```

- Obtient une référence d'objet distribué :

```
ReverseInterface ri = (ReverseInterface) Naming.lookup
("rmi://localhost:1099/MyReverse");
```

- Exécute une méthode de l'objet :

```
String result = ri.reverseString ("Terre");
```

Le client (3/4)

```
import java.rmi.*;
public class ReverseClient {
  public static void main (String [] args) {
    System.setSecurityManager(new RMISecurityManager());
    try{
      ReverseInterface rev = (ReverseInterface) Naming.lookup
                              ("rmi://localhost:1099/MyReverse");
      String result = rev.reverseString (args [0]);
      System.out.println ("L'inverse de "+args[0]+" est "
                                              +result);
    }
    catch (Exception e) {
      System.out.println ("Erreur d'accès à l'objet distant.");
      System.out.println (e.toString());
    }
  }
}
```

Le client (4/4)

- Pour que le client puisse se connecter à rmiregistry, il faut lui fournir un fichier de règles de sécurité `client.policy`.

```
$more client.policy
grant {
  permission java.net.SocketPermission ":1024-65535", "connect";
  permission java.net.SocketPermission ":80", "connect";
};

$more client1.policy
grant {
  permission java.security.AllPermission;
};
```

Compilation et exécution (1/2)

- Compiler les sources (interface, implémentation de l'objet, le serveur et le client) :

```
mulder> javac *.java
```

- Lancer rmic sur la classe d'implémentation :

```
mulder>rmic -v1.2 Reverse
mulder>transférer *Stub.class et ReverseInterface.class
                                vers la machine scott
```

- Démarrer rmiregistry :

```
mulder>rmiregistry -J-Djava.security.policy=client1.policy &
```

Compilation et exécution (2/2)

- Lancer le serveur :

```
mulder>java ReverseServer &
Serveur : Construction de l'implémentation Objet Reverse
          lié dans le RMIregistry
Attente des invocations des clients ...
```

- Exécuter le client :

```
scott>java -Djava.security.policy=client1.policy ReverseClient Lille

L'inverse de Alice est ecilA
```

Charger des classes de manière dynamique

- Les définitions de classe sont hébergées sur un serveur Web.
- Les paramètres, les stubs sont envoyés au client via une connexion au serveur Web.
- Pour fonctionner, une application doit télécharger les fichiers de classe.

Chargement dynamique

- Cela évite de disposer localement de toutes les définitions de classe.
- Les mêmes fichiers de classe (même version) sont partagés par tous les clients.
- On ne charge que les classes dont on a besoin.

Fichiers nécessaires si pas de chargement dynamique

Côté Client

- l'interface : `ReverseInterface`.
- le stub : `Reverse_Stub`.
- le client : `ReverseClient`.

Côté Serveur

- l'interface : `ReverseInterface`.
- l'objet : `Reverse`.
- le serveur d'objets : `ReverseServer`.

Fichiers nécessaires si chargement dynamique

Côté Client

- le client : `ReverseClient`.

Côté Serveur

- le serveur d'objets : `ReverseServer`.

Récupérer les fichiers de classe à partir du serveur Web :

- L'interface : `ReverseInterface`.
- Le stub : `Reverse_Stub`.
- L'objet : `Reverse`.

Le client peut être lui même dynamique (1/2)

Côté Client

- le client : `DynamicClient`.

Côté Serveur

- le serveur d'objets : `ReverseServer`.

Récupérer les fichiers de classe à partir du serveur Web :

- L'interface : `ReverseInterface`.
- Le stub : `Reverse_Stub`.
- L'objet : `Reverse`.
- Le client : `ReverseClient`.

Le client peut être lui même dynamique (2/2)

Côté Client

Le client : `DynamicClient`.

- Chargement dynamique du client et de l'interface à partir d'un répertoire local. Si non disponibles localement, ils sont recherchés sur le serveur Web spécifié.
- L'exécution de `ReverseClient` permet d'obtenir la référence de l'objet `Reverse` et l'appel distant de sa méthode.

Côté Serveur

Le serveur d'objets : `ReverseServer`.

- Chargement dynamique de l'objet `Reverse` à partir du serveur Web.
- Enregistrement de l'objet dans RMIregistry (`bind`).
- Attente de requêtes des clients

Récupérer les fichiers de classe à partir du serveur Web :

- L'interface : `ReverseInterface`.
- Le stub : `Reverse_Stub`.
- L'objet : `Reverse`.
- Le client : `ReverseClient`.

Deux propriétés systèmes dans RMI

⟹ `java.rmi.server.codebase` : spécifie l'URL (file ://, ftp ://, http ://) où peuvent se trouver les classes.
Lorsque RMI sérialise l'objet (envoyé comme paramètre ou reçu comme résultat), il rajoute l'URL spécifiée par codebase.

⟹ `java.rmi.server.useCodebaseOnly` : informe le client que le chargement de classes est fait uniquement à partir du répertoire du codebase.

Principe du chargement dynamique

- $\Longrightarrow$ À l'enregistrement (dans rmiregistry) de l'objet distant, le codebase est spécifié par `java.rmi.server.codebase`.
- $\Longrightarrow$ À l'appel de `bind()`, le registre utilise ce codebase pour trouver les fichiers de classe associés à l'objet.
- $\Longrightarrow$ Le client recherche la définition de classe du stub dans son `classpath`. S'il ne la trouve pas, il essayera de la récupérer à partir du codebase.
- $\Longrightarrow$ Une fois que toutes les définitions de classe sont disponibles, la méthode proxy du stub appelle les objets sur le serveur.

Sécurité lors d'un chargement dynamique

⟹ Les classes `java.rmi.RMISecurityManager` et `java.rmi.server.RMIClassLoader` vérifient le contexte de sécurité avant de charger des classes à partir d'emplacements distants.

⟹ La méthode `LoadClass` de `RMIClassLoader` charge la classe à partir du codebase spécifié.

Les différentes étapes d'un chargement dynamique

- Écrire les classes correspondant respectivement à l'interface et à l'objet.
- Les compiler.
- Générer le Stub correspondant à l'objet.
- Installer tous les fichiers de classe sur un serveur Web.
- Écrire le serveur dynamique.
- Installer rmiregistry au niveau de la machine du serveur.
- Lancer le serveur en lui précisant l'URL des fichiers de classe afin qu'il puisse charger dynamiquement le fichier de classe correspondant à l'objet, l'instancier (le créer) et l'enregistrer auprès de rmiregistry.
- Sur la machine du client, écrire le code du client.
- Compiler le client statique et l'installer éventuellement sur le site Web.
- Compiler le client dynamique et le lancer en précisant l'URL des fichiers de classe.

Exemple avec chargement dynamique

```
// l'interface

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

public interface ReverseInterface extends Remote {
  String reverseString(String chaine) throws RemoteException;
}
```

L'objet Reverse

```
import java.rmi.*;
import java.rmi.server.*;
public class Reverse extends UnicastRemoteObject
                     implements ReverseInterface {
  public Reverse() throws RemoteException {
    super();
  }

  public String reverseString (String ChaineOrigine)
                          throws RemoteException {
    int longueur=ChaineOrigine.length();
    StringBuffer temp=new StringBuffer(longueur);
    for (int i=longueur; i>0; i--) {
      temp.append(ChaineOrigine.substring(i-1, i));
    }
    return temp.toString();
  }
}
```

Le serveur dynamique

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.util.Properties;

public class DynamicServer {
  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      Properties p= System.getProperties();
      String url=p.getProperty("java.rmi.server.codebase");
      Class ClasseServeur = RMIClassLoader.loadClass(url, "Reverse");
      Naming.rebind("rmi://localhost:1099/MyReverse",
                    (Remote)ClasseServeur.newInstance());
      System.out.println("Objet Reverse lié dans le RMIregistry");
      System.out.println("Attente des invocations des clients ...");
    }
    catch (Exception e) {
      System.out.println("Erreur de liaison de l'objet Reverse");
      System.out.println(e.toString());
    }
  } // fin du main
} // fin de la classe
```

Le client

```
import java.rmi.*;

public class ReverseClient {
  public ReverseClient () {
    String mot="Alice";
    try {
      ReverseInterface rev = (ReverseInterface)
                  Naming.lookup ("rmi://localhost:1099/MyReverse");
      String result = rev.reverseString(args[0]);
      System.out.println ("L'inverse de " + mot + " est "+result);
    }
    catch (Exception e) {
      System.out.println ("Erreur d'accès à l'objet distant ");
      System.out.println (e.toString());
    }
  }
}
```

Le client dynamique

```
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.util.Properties;

public class DynamicClient {
  public DynamicClient (String [] args) throws Exception {
    Properties p = System.getProperties();
    String url = p.getProperty("java.rmi.server.codebase");
    Class ClasseClient = RMIClassLoader.loadClass(url, "ReverseClient");
    // lancer le client
    Constructor [] C = ClasseClient.getConstructors();
    C[0].newInstance(new Object[]{args});
  } // vérifier le passage de paramètres

  public static void main (String [] args) {
    System.setSecurityManager(new RMISecurityManager());
    try{
      DynamicClient cli = new DynamicClient();
    }
    catch (Exception e) {
      System.out.println (e.toString());
    }
  }
}
```

Exécution...

```
mulder> ls
Reverse.java ReverseInterface.java DynamicServer.java
mulder> javac *.java
mulder> rmic -v1.2 Reverse
mulder> mv Reverse*.class /home/duvallet/public_html/rmi
Le répertoire destination des fichiers de classe doit être
accessible par http.
mulder> ls *.class DynamicServer.class
mulder>rmiregistry -J-Djava.security.policy=client1.policy &
mulder>java -Djava.security.policy=client1.policy
            -Djava.rmi.server.codebase= http://localhost/~duvallet/rmi
            DynamicServer
Objet lié
Attente des invocations des clients ...
-------------------------------
scott>ls *.java
ReverseClient.java DynamicClient.java
scott>javac *.java
scott>java -Djava.security.policy=client1.policy
           -Djava.rmi.server.codebase=http://localhost/~duvallet/rmi
           DynamicClient

L'inverse de Alice est ecilA
scott>
Le chargement de ReverseClient se fera à partir du répertoire local
alors que ReverseInterface et Reverse_Stub seront chargés à partir
du serveur Web spécifié dans la commande.
```