Programmation orientée objet en langage JAVA

Java RMI : Techniques et utilisations avancées de RMI

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/

Techniques et utilisations avancées de RMI

1. Sécurité et chargement dynamique de code
2. Applets et RMI
3. Polymorphisme et RMI
4. Callback, synchronisation multi-thread et RMI
5. Ramasse-miettes répartie

Sécurité et chargement dynamique de code

- Si le code du stub n'est pas présent sur le site local, le protocole RMI prévoit le chargement dynamique du stub en utilisant un serveur web et le protocole HTTP
  - `java -Djava.rmi.server.codebase=http ://hostname/... HelloServeur`
- Si on utilise le chargement dynamique alors
  - le chargeur dynamique utilisé par RMI (RMIClassLoader) regarde si la classe demandée correspond au niveau de sécurité requis
    - utilisation d'un SecurityManager.
  - il faut créer et installer le « gestionnaire de sécurité »
    - `System.setSecurityManager(new RMISecurityManager());`

Rappels et cadre

Codebase :

- Un codebase est un endroit depuis lequel charger du code dans une machine virtuelle. CLASSPATH est un codebase local, autre exemple de codebase :
  `http://serveur/~duvallet/classes/`
- Le codebase d'une applet est toujours relatif à l'URL depuis laquelle elle a été chargée.

Security Manager :

- Toute Jmachine virtuelle qui doit charger du code a besoin (Java 2) d'un Security Manager (SM).
- Principe d'un Security Manager :
  - si aucun security manager n'est installé, alors seules les classes accessibles depuis le CLASSPATH peuvent être chargées.
  - un security manager vérifie différents critères sur les classes au chargement.

Security Manager

- Un `SecurityManager` doit être utilisé avec RMI, sinon les clients et les serveurs RMI ne peuvent pas charger de code autre que directement par le CLASSPATH.
- Dans le cas d'une application générale (ni NFS, ni une applet), il faut un `SecurityManager` dans le client et le serveur.
- Les clients RMI charge du code : les stub.
- Dans le cas d'une applet : on peut se contenter d'un `SecurityManager`(SM) dans le serveur car le client dispose déjà d'un SM : celui du browser qui a chargé l'applet.

Fichier de sécurité et RMI

- Pour obtenir les sécurités de base nécessaire à RMI, il faut écrire un fichier `.java.policy`

```
grant {
  permission java.net.SocketPermission "*:1024-65535", "connect,accept,resolv
  permission java.net.SocketPermission "*:1-1023", "connect,resolve";
};
```

- La deuxième partie concerne les ports réservés.
- Ou toutes les permissions :

```
grant {
 // Permettre tout
  permission java.security.AllPermission;
};
```

Applets signées et RMI

- Par défaut, une applet ne peut communiquer (tcp, donc RMI) qu'avec le host qui l'a chargée.
- Pour obtenir son nom : `getCodeBase().getHost()`.
- Si l'on veut faire une communication vers une objet RMI se trouvant sur un autre host, il faut faire une applet signée.
- Une applet ne peut pas non-plus par défaut communiquer par RMI avec le host qui l'a chargée (celui sur lequel elle est en train de s'exécuter).

Codebase et RMI

- Avec RMI, propriété Java spéciale : `java.rmi.server.codebase`.
- Au lancement du client (mais aussi du seveur si nécessaire, voir plus loin) on va spécifier le java rmi server codebase, de sorte que l'on puisse télécharger du code (byte code de classes Java) si nécessaire.
- Par exemple : le code d'implémentation du stub.
- `codebase` est une propriété qui représente une ou plusieurs URL depuis lesquelles des classes peuvent être téléchargées.
- Exemple d'utilisation :

```
java -Djava.rmi.server.codebase=http://serveur/~duvallet/classes/ hello.HelloImpl
```

Codebase et RMI

- Ou plusieurs séparés par des blanc et entre “ ” :

```
java -Djava.rmi.server.codebase="http://serveur1/exemple1.jar
                                 http://serveur2/exemple2.jar
                                 http://serveur3/dir/"
```

- Il faut un / si c’est un répertoire et non un fichier.
- On peut aussi utiliser une URL vers un fichier : `file:/monRepertoire/chemin`.
- La technique n’est pas utilisable uniquement pour télécharger les stub, mais également pour récupérer le code (.class) de paramètres polymorphes.
- Ces techniques de polymorphisme et chargement dynamique permettent de faire des infrastructures client-serveurs complètement générique.

CLASSPATH et rmiregistry

- Avant de lancer le rmiregistry, on doit s'assurer de ne pas avoir de CLASSPATH qui permet de télécharger des classes que l'on veut télécharger automatiquement vers le client.
- En particulier le stub pour l'objet distant.
- Si rmiregistry peut accéder à des classes par le CLASSPATH, cela va être prioritaire sur le codebase, et un client ne pourra pas télécharger le stub, ou autres classes.
- Voir : `java.rmi.server.codebase property`
- Pour être absolument sûr, il est préférable de faire `unset CLASSPATH` avant de lancer le rmiregistry

Principe des applets

- Depuis une applet, on va faire un appel RMI vers un serveur qui se trouve sur le host depuis lequel on a chargé l'applet.
- Le résultat de l'appel est retourné dans le browser, et affiché par l'applet.
- Les différentes étapes :
  1. Chargement de l'applet
  2. Lookup dans le rmiregistry
  3. Retour du stub
  4. Appel à l'objet distant
  5. Retour du résultat

Code et principes des classes

- Attention à l'utilisation des packages : en Java, il y a un mapping entre le nom complet du package d'une classe et le chemin pour trouver cette classe.
- Cela permet au compilateur de trouver les classes.
- Dans l'exemple :
  - package hello
  - $HOME/rmi/hello
- Remote Interface Hello

```
package hello;

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

  public interface Hello extends Remote{
      String sayHello() throws RemoteException;
  }
```

Serveur Hello

```
package hello;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
        super();
    }

    public String sayHello() {
        return "Bonjour le monde !";
    }

    public static void main(String args[]) {
        // Crée et installe un Security Manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            HelloImpl obj = new HelloImpl();
            // Attache l'instance de l'objet avec le nom "HelloServer"
            Naming.rebind("HelloServer", obj);
            System.out.println("HelloServer liée dans le rmiregistry");
        }
        catch (Exception e) {
            System.out.println("HelloImpl err: " + e.getMessage()); e.printStackTrace();
        }
    }
}
```

Applet client

```
package hello;
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {
    String message = "blanc";

    // "obj" est l'identifiant de l'objet distant qui implément le "Hello"
    Hello obj = null;

    public void init() {
        try {
            obj = (Hello)Naming.lookup("//" + getCodeBase().getHost() + "/HelloServer");
            message = obj.sayHello();
        }
        catch (Exception e) {
            System.out.println("HelloApplet exception: " + e.getMessage());
            e.printStackTrace();
        }
    }

    public void paint(Graphics g) {
        g.drawString(message, 25, 50);
    }
}
```

Page HTML : hello.html

```
<HTML>
<TITLE>Hello World</TITLE>
<BODY>
<CENTER>
  <h1>Hello World</h1>
</CENTER>
Le message depuis HelloServer est :
<p>
  <applet codebase="classes/" code="hello.HelloApplet" width=500 height=120>
  </applet>
</p>
</BODY>
</HTML>
```

Compilation et déploiement (1/5)

- Placement des fichiers, serveur HTTP :
  - Compilation en spécifiant où les .class doivent être placés : `$HOME/public_html/classes`
  - Souvent les serveurs web permettent l'accès au répertoires des utilisateurs par : `http://host/~username/`, et `public_html` et égal à `www`.
  - Sinon, et dans le cas où le serveur de l'applet cliente et du serveur ont accès au même système de fichiers, alors on peut utiliser une URL de type file : `file:/home/login/public_html`.
  - Alternative : serveur HTTP minimal fourni par Sun :

`http://java.sun.com/products/jdk/rmi/class-server.zip`

Compilation et déploiement (2/5)

- Compilation :
  - Supposons que l'on place les .class dans $HOME/public_html/classes alors on va compiler de la façon suivante :
    `javac -d $HOME/public_html/classes hello/Hello.java hello/HelloImpl.java hello/HelloApplet.java`
  - Cette commande :
    - crée le répertoire hello (s'il n'existe pas déjà) dans le répertoire `$HOME/public_html/classes`.
    - et y met les `.class` Il faut que le répertoire `classes` existe.
    - Compile et y crée les 3 fichiers : Hello.class HelloImpl.class HelloApplet.class Puis, génération des stub et skeleton :
      `rmic -d $HOME/public_html/classes hello.HelloImpl`

Compilation et déploiement (3/5)

- Compilation (suite) :
  - La commande est exécutée sur la classe qui implémente des objets distants (HelloImpl) et va générer : `HelloImpl_Stub.class` et `HelloImpl_Skel.class` dans le même répertoire que les autres .class
- Placement de l'applet et CLASSPATH
  - Mettre le fichier HTML sous le répertoire accessible depuis le browser : `mv $HOME/src/hello/hello.html $HOME/public_html/`

Compilation et déploiement (4/5)

- Lancer le RMI registry :
  - Pas de CLASSPATH `unset CLASSPATH rmiregistry &` ou `rmiregistry 2001 &`.
  - On veut que le client qui va utiliser le rmiregistry download les .class par le serveur HTTP.
  - Si le rmiregistry trouve dans son CLASSPATH les .class, alors il va ignorer la propriété codebase du serveur qui lui indique comment trouver les classes, et le client ne marchera pas.

Compilation et déploiement (5/5)

- Lancer le serveur :
  - Pour lancer le serveur (HelloImpl), il faut que le répertoire $HOME/public_html/classes soit dans le CLASSPATH.
  - C'est la propriété Java (`java.rmi.server.codebase`) qui va permettre de spécifier comment transmettre le stub du host du serveur vers le client
  - Tout d'abord du serveur vers le rmiregistry :
    - Lorsque le rmiregistry a besoin du stub, il ne le trouve pas en local, car PATH à vide, il va donc chercher dans le codebase fourni par le serveur http.
    - Cette source de chargement est mémorisée dans l'objet, plus tard le client utilisera le même moyen pour charger le .class du stub reçu.

```
java -Djava.rmi.server.codebase=http://serveur/ duvallet/ classes/
      -Djava.security.policy=$HOME/src/policy hello.HelloImpl
```

Polymorphisme et RMI

- Mécanisme puissant pour la programmation répartie.
- Permet de faire, par exemple, du client-serveur générique, ou encore de gérer l'évolution des logiciels sans avoir besoin d'arrêter des systèmes complexes.

Principe et implications

- On envoie à distance un sous-type du type attendu.

$\Longrightarrow$ Technique classique du polymorphisme.

- Ici, en plus, on gère dynamiquement le fait que l'objet que l'on envoie n'est pas forcément connu dans le processus distant où il arrive :
  - l'objet n'est pas connu,
  - on ne dispose pas du byte code de sa classe.
- Comme pour les stub de l'exemple précédent, il va falloir faire du chargement dynamique de code d'une machine virtuelle à l'autre.

Retour de résultat

Du Serveur vers le client :

- Le client appelle une routine du type : `Resultat1 res = serveur.meth (...)`.
- Le serveur retourne un objet de type `Resultat2` qui dérive de `Resultat1`.
- Le client devra charger le code de `Resultat2` afin de pouvoir se servir (appeler les routines, avoir accès aux attributs) de l'objet reçu en résultat de son appel distant.
- Note : Le client devra également charger toutes les classes (ou interfaces) utilisées par `Resultat2` : ses attributs, les paramètres de ses routines, etc.

Passage de paramètres

Du Client vers le Serveur

- Le client appelle une routine du type :

  Resultat1 res = serveur.meth (ParamA1, ParamB1)

- mais ne passe pas en paramètres des objets de type ParamA1, ParamB1, mais de type ParamA2, ParamB2 qui sont des classes qui en dérivent.
- Le serveur devra charger le code de ParamA2, ParamB2 afin de pouvoir se servir (appeler les routines, avoir accès aux attributs) des objets reçu en paramètre pour exécuter l'appel de "meth".
- Remarque : Le serveur devra également charger toutes les classes (ou interfaces) utilisées par ses deux nouvelles classes : ses attributs, les paramètres de ses routines, etc.

Modèle

Que se passe-t-il si N clients font des appels RMI sur un serveur ?

- Ce n'est pas précisé dans la sémantique de RMI Au moins un thread, mais implicite.
- On ne sait pas si l'on va avoir plusieurs threads, ou si les appels sont sérialisés.
- S'il y a un seul thread, il y a forcément une mise en attente si plusieurs appels arrivent de façon proche.
- Mais on ne sait pas vraiment dans quel ordre ils seront exécutés :
  - FIFO : le plus simple et le plus logique.
  - Un pool de threads.
- On peut faire des implémentations de RMI qui créent un thread à chaque appel.

Appels asynchrones

Comment faire un appel asynchrone avec RMI ?

- Gérer l'asynchronisme explicitement avec un thread
- Dans le cas le plus simple (une routine qui retourne void), il n'y a pas grand chose de plus à faire.
- Le thread est utilisé simplement pour ne pas bloquer l'appelant.
- Si on veut faire un appel asynchrone avec une routine qui ne retourne pas void :
  - Il faut également un emplacement spécial (futur) pour mettre le résultat
  - Il faudra également avoir une technique pour tester si le résultat est revenu (ou notifier le thread qui pourrait attendre le résultat)
  - Il faudra aussi une technique de synchronisation pour faire attendre l'appelant si on le souhaite.
    - Soit on garde un thread bloquée chez l'appelant,
    - Soit on fait un Call Back depuis l'appelé.

Callback

Comment faire un Call back avec RMI ?

- Très utile, par exemple pour informer un client qu'une valeur à changée dans un serveur.
- Éviter de faire du "pooling" : faire sans cesse des appels au serveur pour voir si la valeur à changée.
- Pas de moyen simple est direct, il faut avoir un objet distant (Remote) du côté du client également afin de faire un appel RMI du serveur vers le client !
- Pattern au niveau des objets : Callback sur le client lui-même :
  - l'objet client est lui même un objet distant et il doit se protéger des call backs (synchronize).
  - un autre objet de la machine virtuelle du client est un objet distant et il y a une référence mutuelle entre celui-ci et le client.
- Attention à au moins deux choses :
  - accès concurrent,
  - possibilités de deadlock lors des callbacks.

Ramasse-miettes répartie

- Le DGC (Distributed Garbage Collector) interagit avec les GC locaux et utilise un mécanisme de *reference-counting*.
- Lorsqu'un OD est passé en paramètre à un autre OD $\rightarrow$ `ref_count++`.
- Lorsqu'un stub n'est plus référencé $\rightarrow$ *weak reference*.
- Lorsque le GC du client libère le *stub*, sa méthode `finalize` est appelée et informe le DGC $\rightarrow$ `ref_count--`.
- Lorsque le nombre de références d'un OD = 0 $\rightarrow$ *weak reference*.
- Le GC du serveur peut alors libérer l'OD.
- Le client doit régulierement renouveler son bail auprès du DGC.
- Si référence à un OD libérée $\rightarrow$ `RemoteException`.