Remote Method Invocation (RMI)

Introduction

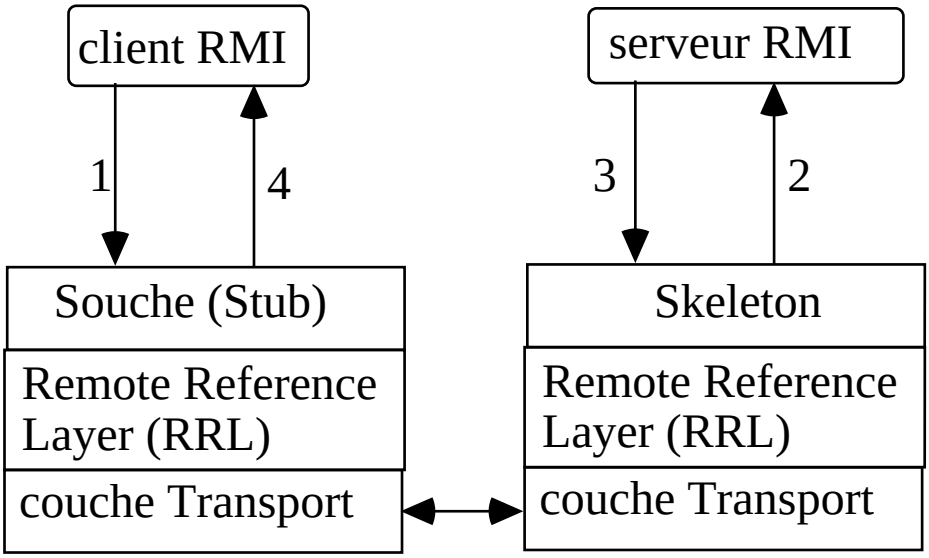
RMI est un ensemble de classes permettant de manipuler des objets sur des machines distantes (objets distants) de manière similaire aux objets sur la machine locale (objet locaux).
C'est un peu du "RPC orienté objet". Un objet local demande une fonctionnalité à un objet distant.
RMI apparaît avec Java 1.1 et est complètement intégré depuis Java 1.1 (mars 1997) donc est gratuit (• CORBA)

C'est du CORBA allégé avec ses avantages (c'est plus simple) et ses inconvénients (tout doit être Java coté client comme coté serveur) (mais JNI !).
Par contre RMI peut charger des classes à l'exécution, du serveur vers le client ou dans l'autre sens contrairement à CORBA.

Ces manipulations sont "relativement" transparentes. Pour cela RMI propose :
- un ramasse-miettes distribué.
- la gestion des représentants locaux d'objets distants et leur activation.
- la liaison avec les couches transport et l'ouverture de sockets appropriés.
- la syntaxe d'invocation, d'utilisation d'un objet distant est la même qu'un objet local.

RMI en pratique

fonctionne dans les applications Java 1.1 et dans appletviewer.



la couche souche contient des représentants locaux de références d'objets distants. Une invocation d'une fonctionnalité sur cet objet distant est traité par la couche RRL (ouverture de socket, passage des arguments, marshalling, ...). Les objets distants sont repérés par des références gérées par la couche skeleton.

Programmation avec RMI

Pour créer une application RMI, suivre les étapes suivantes.

1°) définir les spécifications du service RMI sous forme d'interfaces.

2°) Créer et compiler les implémentations de ces interfaces.

3°) Créer les classes pour la souche et le skeleton à l'aide de la commande `rmic`.

4°) Créer et compiler une application « lanceur du service RMI ».

5°) lancer le `rmiregister` et lancer l'application « lanceur du service RMI ».

6°) Créer et compiler un programme client qui accède à des objets distants : ce service RMI.

7°) lancer ce client.

Un exemple : calcul d'âge de personnes

Le client envoie un objet de classe `Personne` à un objet distant qui calcule son âge c'est à dire retourne des objets de classe `Age`.

On utilise donc deux classes :

```
package business;
import java.util.GregorianCalendar;
import java.util.Calendar;
import java.io.Serializable;
/*
* Personne implemente Serializable car devra etre
envoyé par RMI
*/
```

```
public class Personne implements Serializable {
private String nom;
private String prenom;
private GregorianCalendar dateNaissance;

public String getNom() {return nom;}
public String getPrenom() {return prenom;}
int getAnneeNaissance() {
   return dateNaissance.get(Calendar.YEAR);}

public Personne (String nom, String prenom,
GregorianCalendar dateNaissance) {
   this.nom = nom;
   this.prenom = prenom;
   this.dateNaissance = dateNaissance;
}
}
```

et :

```
package business;
import java.util.GregorianCalendar;
import java.util.Calendar;
import java.io.Serializable;
/*
 * Age implemente Serializable car devra etre retourné
par RMI
*/

public class Age extends GregorianCalendar
implements Serializable {
  public int getNbAnnees() {
    return this.get(Calendar.YEAR);
}
}
```

1°) définir les spécifications du service RMI à l'aide d'interface.

Les classes placées à distance sont spécifiées par des interfaces qui doivent

- hériter de `java.rmi.Remote` et
- - dont les méthodes lèvent une `java.rmi.RemoteException` exception. On définit alors l'interface `CalculAgeRMI`.

```
package calcul;
import java.rmi.*;
import business.*;

public interface CalculAgeRMI extends Remote {
  public Age calcul(Personne pers) throws
      java.rmi.RemoteException;
}
```

2°) Créer et compiler les implémentations de ces interfaces.

Il s'agit d'écrire une classe qui implémente l'interface ci dessus. C'est le service (classe) fourni à distance. En général ces classes ont pour nom *nomInterface*Impl.
Ces classes doivent dériver de `java.rmi.server.UnicastRemoteObject` qui est la classe qui permet de définir des objets distants interrogeables en point à point. Il est prévu de construire des objets distants interrogeables en diffusion mais ce n'est pas encore implanté.
De plus `java.rmi.server.UnicastRemoteObject` dérive de `java.rmi.server.RemoteObject` qui redéfinit les méthodes `equals()`, `hashcode()`, et les adaptent aux objets distribués.

```
package calcul;
import business.*;
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class CalculAgeRMIImpl extends
UnicastRemoteObject implements CalculAgeRMI {
  public CalculAgeRMIImpl() throws
java.rmi.RemoteException {}
  public Age calcul(Personne pers) throws
java.rmi.RemoteException {
  int anneeNaissance = pers.getAnneeNaissance();
  System.out.println("annee de naissance de " +
pers.getNom() + " : " + anneeNaissance);
  // faire la soustraction entre la date courante
  // et la date de naissance
```

```
  GregorianCalendar aujourdhui = new
GregorianCalendar();
  int anneeCourante = aujourdhui.get(Calendar.YEAR);
  System.out.println("annee courante : " +
anneeCourante);
  int ageEntier = anneeCourante - anneeNaissance;
  // on cree un calendrier et on le positionne
  // pour fabriquer l'age
  Age ageARetouner = new Age();
  ageARetouner.set(ageEntier + 1, 0 , 0);
// + 1 car la premiere annee a pour indice 0
  return ageARetouner;
  }
}
```

Pour compiler utiliser les commandes :

```
javac -d classes business\*.java
calcul\CalculAgeRMI*.java
```

Ces commandes vont créer les fichiers `.class` et les déposer dans `./classes`

3°) Créer les classes pour la souche et le skeleton (commande `rmic`)

Il faut maintenant créer la souche et le skeleton pour accéder aux classes ci dessus. Java donne un outil `rmic` qui utilisent les fichiers `.class`.
la syntaxe de `rmic` est

```
rmic [options]
paquetage.sousPaquetage..nomDeClasse1
paquetage.sousPaquetage.. nomClasse2...
```

On a donc ici :

```
rmic -d . calcul.CalculAgeRMIImpl
```

à lancer dans le répertoire `./classes`

En fait, seule la souche est créée. Le squeleton est devenu inutile en Java 1.4.

4°) Créer et compiler une application «lanceur de serveur RMI »

C'est le programme qui créé une instance de serveur RMI qui sera à l'écoute des demandes des clients. Ce programme crée un objet `CalculAgeRMIImpl` qui attend alors les requêtes.

```
import java.rmi.*;
import java.net.MalformedURLException;
import calcul.*;
public class LanceServeurRMI {
  public static void main(String args[]) {
    try {
      CalculAgeRMIImpl ca = new CalculAgeRMIImpl();
      System.out.println("l'objet RMI est construit");
      Naming.rebind(Constantes.NOM_OBJET_RMI,
ca);
      System.out.println("l'objet RMI est inscrit au
service de nommage");
    } catch (RemoteException exp) {
        System.out.println("Pb de RemoteException : " +
exp);
      } catch (MalformedURLException exp) {
          System.out.println("Pb de NotBoundException :
" + exp);
       }
   }
}
```

avec le fichier :

```
public interface Constantes {
       public static final String NOM_OBJET_RMI =
"objetRMI";
}
```

4°) Créer et compiler une application « lanceur de serveur RMI » (suite)

Si on veut que l'application indépendante (lanceur de serveur RMI) puisse télécharger des classes distantes (du futur client RMI), il faut installer un `SecurityManager` spécifique. Celui-ci est donné par Java et on l'installe par :

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

Pour rendre l'objet calcul de l'âge disponible, il faut l'enregistrer dans le "RMIregistry" (voir ci dessous). On lui donne pour cela un nom et on l'enregistre par la méthode statique

```
Naming.rebind(Constantes.NOM_OBJET_RMI, cmi);
```

Par la suite cet objet calculateur d'âge sera accessible par les autres machines en indiquant la machine sur laquelle est exécuté ce serveur RMI et le nom associé à cet objet (cf. ci dessous).
Le premier argument de `Naming.rebind()` doit être une `String` :

On compile ce serveur par :

```
javac -d classes LanceServeurRMI.java
```

5°) lancer `rmiregister` et l'application « lanceur de serveur rmi »

Il faut d'abord lancer le `rmiregister` puis le serveur.
Sous Unix, on lance `rmiregistry` par :

```
% rmiregister &
```

Sous Win32, on lance `rmiregistry` par :

```
start rmiregistry
```

Il faut lancer `rmiregistry` dans un répertoire tel qu'il pourra accédé à la souche lorsqu'on exécutera le « lanceur de serveur RMI ». En effet ce lanceur de serveur RMI demande en fait au service de nommage rmiregistry de connaître la souche (pour éventuellement la donner aux clients par http).

Puis on lance le serveur :

```
% java LanceServeurRMI
```

Ce qui rend maintenant le serveur disponible pour de futurs clients.
Les sorties du serveur sont :

```
l'objet RMI est construit
l'objet RMI est inscrit au service de nommage
```

6°) Créer et compiler un programme client qui accède au serveur RMI

Le programme doit être lancé par :

```
% java ClientRMI localhost
```

```
import java.rmi.*;
import java.util.*;
import java.net.MalformedURLException;
import business.*;
import calcul.*;

public class ClientRMI {
  public static void main(String args[ ]) {
    if (args.length != 1) {
       System.out.println("usage java ClientRMI
<NomMachineDistanteRMI>");
       System.exit(0);
   }
  GregorianCalendar gc =
     new GregorianCalendar(1915, 6, 11);
  Personne tryphonTournesol =
     new Personne("Tournesol", "Tryphon", gc);

  GregorianCalendar gc2 =
     new GregorianCalendar(1910, 6, 24);
   Personne haddock =
     new Personne("Haddock", "Archibald", gc2);
   // on recupere un objet distant CalculAgeRMI
   // on lui demande de faire le calcul
   try {
      CalculAgeRMI carmi =
         (CalculAgeRMI) Naming.lookup("rmi://" + args[0]
+ "/" + Constantes.NOM_OBJET_RMI);
       Age ag = carmi.calcul(tryphonTournesol);
```

```
    System.out.println("la personne " +
        tryphonTournesol.getPrenom() + " " +
        tryphonTournesol.getNom() + " a pour age : "  +
        ag.getNbAnnees());
    Age ag2 = carmi.calcul(haddock);
    System.out.println("la personne " +
       haddock.getPrenom() + " " +
       haddock.getNom() + " a pour age : " +
       ag2.getNbAnnees());
  } catch (RemoteException exp) {
       System.out.println("Pb de RemoteException : " +
exp);
   } catch (NotBoundException exp) {
       System.out.println("Pb de NotBoundException : "
+ exp);
    } catch (MalformedURLException exp) {
        System.out.println("Pb de NotBoundException : "
+ exp);
     }
  }
}
```

Remarques

Le code ci dessus manipule les objets distants comme s'ils étaient locaux.

Le client recherche le service de calcul d'âge distant par `Naming.lookup(url)` où `url` est une "URL rmi" de la forme

`rmi://`*`machineDuServeurRMI`*`:`*`port/nomServeurRMI`*

Pour tester ce programme sur une seule machine utiliser `localhost` comme *`machineDuServeurRMI`*.

`Naming.lookup()` retourne une référence `Remote` qu'il faut nécessairement convertir.

7°) lancer ce client

```
% java ClientRMI localhost
```

Notions avancées (1/2)

Un des plus gros intérêt de RMI est d'utiliser la particularité des machines virtuelles Java de pouvoir télécharger des classes à l'aide du protocole http donc en utilisant des serveurs Web.
Ainsi on a souvent une architecture où :
- la machine qui fournit le serveur RMI fournit aussi le stub pour les clients : c'est systématiquement le cas pour des applets.
- la machine qui lance l'exécution du client a rangé à un endroit connu, des classes pouvant être utile au serveur RMI qui seront chargées par la machine virtuelle du serveur par le protocole http. Le client possède alors un serveur http susceptible d'envoyer ces classes au serveur RMI.

On utilise pour cela la propriété `java.rmi.server.codebase`.
Le client est lancé par :

```
java -Djava.rmi.server.codebase=
http://une_machine/
repertoireOuSeTrouveDesClassesUtilesPourLeSe
rveur/ clientRMI
```

Le serveur est lancé par :

```
java -Djava.rmi.server.codebase=
http://une_autre_machine/repertoireOuSeTrouv
eDesClassesUtilesPourDesClientsExempleLeStub
/ LanceServeurRMI
```

Remarques :
1°) Par pitié ne pas oublier le /
2°) Mettre l'adresse IP de la machine et non pas localhost.

Notions avancées (2/2)

C'est parce qu'on a installé dans les 2 programmes `LanceServeurRMI` et `ClientRMI` un `RMISecurityManager`, que ces classes peuvent être chargées d'une machine externe à la machine d'exécution.
De plus, il faut autoriser que ces programmes accepte des des connexions et se connecte sur un serveur une machine distante. Utiliser l'outil policytool pour entrer l'autorisation :

```
permission java.net.SocketPermission
«machineDistante», «accept,connect»;
```

(ou une autorisation plus forte ;-)
ce qui donne à `machineDistante` l'autorisation d'une connexion sur la machine locale.
On peut aussi indiquer cette propriété sur la ligne de commande par :

```
java –Djava.security.policy=UnFichierDeSecurite
ProgrammeRMI
```

Rappelons, par contre, que les arguments des méthodes passés du client au serveur le sont par sérialisation.
Voir le tutorial de SUN à l'URL
`http://java.sun.com/docs/books/tutorial/rmi/index.html`

Exercice

1°) Ne pas mettre les stubs coté client.
2°) Lancer coté serveur, le serveur de noms RMI rmiregistry, de sorte qu'il ne connaisse pas les stubs
3°) Lancer sur la machine serveur un serveur web.
4°) Placer les stubs dans le répertoire du serveur web accessible par l'url
`http://la_machine/repertoireOuSeTrouveDesClassesUtilesPourDesClientsExempleLeStub`
5°) Placer dans le répertoire du serveur web accessible par l'url des classes qui peuvent être utiles pour le stub. Dans notre exemple ce sont, les classes `Age.class` et `Personne.class` du paquetage `business` et l'interface `CalcuAgeRMI.class` du paquetage `calcul`.
6°) Lancer le serveur par :

```
java -Djava.rmi.server.codebase=
http://la_machine/repertoireOuSeTrouveDesClassesUtilesPourDesClientsExempleLeStub/ LanceServeurRMI
```

Bien mettre l'adresse IP de la machine et pas localhost.
7°) Lancer le client par :

```
java ClientRMI numIPMachineServeur
```

8°) Ca fonctionne ! !

Bibliographie

Remote Method Invocation Specification à :
http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html
ou
http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

Le tutorial sur RMI :
http://java.sun.com/docs/books/tutorial/rmi/index.html