TD 9 - Cooking with RMI / Asynchronous RMI

P. Esling

3 décembre 2013

Exercice 1 – Homegrown RMI

Comme nous sommes au stade du hardcore coding tendance asociaux anarchiques, nous allons développer notre propre service RMI en Java, sans utiliser aucune des classes proposées par le JDK. Nous allons donc développer la couche faisant le lien entre le serveur et le client (`Remote Reference Layer`). Plus précisément, nous allons écrire le code permettant de déléguer l'appel fait depuis le client au travers d'un socket vers notre serveur maison. Nous allons devoir coder un équivalent au registre RMI basique ainsi que de quoi instancier nos stub et skeleton et enfin répondre aux appels distants.

Question 1

Faire la liste des modules et fonctions qui seront nécessaires pour effectuer notre implémentation maison du protocole RMI.

Solution:

Ici commencer par expliquer les différentes implémentations possibles d'un système RPC (par messages, par sérialisation ou par lightweight RPC). Plusieurs modules peuvent être acceptés mais de manière minimale (dans notre cas pour le RPC par sérialisation), il va nous falloir

- Les classes et interfaces qui seront appelées à distance (comme pour les `Remote` du RMI Sun).
- Un objet d'invocation (`InvocationContext`) permettant d'envoyer au serveur l'objet ciblé, la fonction et les paramètres utilisés. On devra ici utiliser un objet sérialisable et un flux d'objets.
- Un handler permettant de récuperer les appels de fonctions, les transformer en `InvocationContext` et les transmettre sur un flux d'objet.
- Une erreur spécifique aux appels distants (semblable á `RemoteException`).
- Un serveur de registre identique au `Registry` du RMI Sun. Celui-ci devra proposer les fonctions
  - Une boucle d'écoute infinie permettant de répondre aux requêtes client.
  - Une fonction pour enregistrer les objets (`rebind`)
  - Un thread spécifique á chaque requête permettant de traiter l'appel.
  - Une fonction `args2class` permettant de transformer les paramètres de l'invocation en classe et objet du bon type pour pouvoir effectuer l'appel.
  - Les méthodes permettant la création dynamique du stub et du skeleton.

Question 2

Nous allons publier le même service qu'avec l'implémentation RMI de Sun : nous allons donc garder nos interfaces de la séance précédente. Comme nous faisons tout nous-même, notre implémentation ne nécessitera pas d'utiliser l'interface `Remote` ni d'expliciter `RemoteException` comme exception propageable. Réecrire les classes et interfaces dont nous nous servirons pour les appels distants.

Solution:

Ici l'implémentation est identique au TD précédent, sauf qu'on a plus besoin des objets RMI (`Remote` et `RemoteException`

```
public interface IRemoteCalculette
{
 public Integer eval(Integer op1, Integer op2);
 public void cumuler(Integer val);
 public Integer getCumul();
 public void resetCumul();
}

public class RemoteCalculette implements IRemoteCalculette
{
 int cumul = 0;
 public RemoteCalculette() { super(); }
 public Integer eval(Integer op1, Integer op2) { return new Integer(op1.intValue() + op1.intValue()) ; }
 public void cumuler(Integer val) { cumul += val.intValue(); }
 public Integer getCumul() { return new Integer(cumul); }
 public void resetCumul() { cumul = 0; }
}
```

Question 3

Dans un premier temps, on se rends compte que avant de pouvoir effectuer l'appel distant, il va falloir d'abord faire traverser un socket à notre appel. Le principe est donc de regrouper tout ce dont nous avons besoin dans un objet chez le client pour réaliser notre invocation (nom du service, méthode à invoquer, et les valeurs des paramètres). Nous allons regrouper cela dans une classe sérialisable qui traversera le socket. Cette classe sera instanciée par notre stub, envoyée dans le socket, et traitée par le skeleton.

Solution:

```
public class InvocationContext implements Serializable
{
 private Object[] args;
 private String method;
 private String name;

 public InvocationContext() {     }

 public InvocationContext(String name, Object[] args, String method)
 {
  this.name = name;
  this.args = args;
  this.method = method;
 }

 // + fonctions get et set pour tous les arguments

}
```

Question 4

Lorsque le client effectue un appel sur un objet distant, nous devons d'une certaine manière attraper cet appel. Il nous faut donc définir une classe dérivant de `java.lang.reflect.InvocationHandler` (permettant des appels de fonctions sur objets de manière dynamique). Ceci produira une implémentation dynamique de notre stub, permettant ainsi de simuler un appel distant transparent pour l'utilisateur. Écrire un handler d'appel `RMIHandler`, créé sur un couple de flux d'objets (entrant et sortant) donné et qui instanciera simplement un `InvocationContext`, l'enverra

dans le socket, et retournera ce qui revient par le socket.
Solution:

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class RMIHandler implements InvocationHandler
{
 private ObjectInputStream ois;
 private ObjectOutputStream oos;
 private String name;

 public RMIHandler(String name, ObjectInputStream ois, ObjectOutputStream oos)
 {
  this.name = name;
  this.ois = ois;
  this.oos = oos;
 }

 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
 {
  oos.writeObject(new InvocationContext(name, args, method.getName()));
  return ois.readObject();
 }
}
```

Question 5

Lors des invocations, écriture des flux et sérialisation des objets, de multiples erreurs peuvent se produire (notamment au niveau des échanges résaux mais également des appels de fonctions). Comme nous sommes toujours dans l'optique de faire notre implémentation maison, nous allons donc créer notre propre exception spécifique à notre système d'appel distant (semblable à `RemoteException`), permettant de réunir toutes les erreurs. Quelle serait l'héritage le plus approprié pour cette exception ? Écrire la classe `RemoteInvocationException`.
Solution:

```
public class RMIInvocationException extends RuntimeException
{
 public RMIInvocationException() { super(); }
 public RMIInvocationException(String message) { super(message); }
 public RMIInvocationException(String message, Throwable cause)
 { super(message, cause); }
}
```

Question 6

Nous allons maintenant attaquer la couenne du problème, c'est à dire le registre. Comme l'écriture d'un registre RMI est relativement complexe, nous allons considérer ce problème fonction par fonction en supposant que toutes les fonctions feront finalement parties d'une classe `pc2r.Registry`. Écrire la fonction `register`, équivalente à la fonction RMI `bind`, en précisant bien quel type de structure de données vous comptez utiliser.
Solution:

```
// Insister sur l'utilisation d'une hashmap
private Map<String, Object> services = new HashMap<String, Object>();
public Registry register(String name, Object service)
{
 services.put(name, service);
 return this;
}
```

Question 7

Avant de passer à l'implémentation du serveur à proprement parler, il nous faut interpréter les arguments envoyés par les objets `InvocationContext`, notamment en termes de type (vu qu'on utilise ici une liste d'objets). On utilise pour ce faire la fonction `args2Class` permettant de construire la liste des types des arguments de l'invocation (le code vous est donné à la fin de la question). Ceci nous permettra d'utiliser les fonctionnalités d'appel dynamique JAVA (fonctions `getMethod()` et `invoke()`). Grâce à ces fonctions, écrire le code du serveur permettant de gérer les appels distants côté serveur (on considèrera qu'on réutilise le code d'un serveur précédent et on se contentera ici de modifier la fonction `run()` des threads clients)

```
private Class[] args2Class (Object[] objs)
{
 List<Class> classes = new ArrayList<Class>();
 for (Object o : objs)
  { classes.add(o.getClass());}
 return classes.toArray(new Class[]{});
}
```

Solution:

```
public void run()
{
 try
 {
  ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
  ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
  InvocationContext ic = (InvocationContext) ois.readObject();
  Object targetObject = services.get(ic.getName());
  Object result = targetObject.getClass()
  .getMethod(ic.getMethod(), args2Class(ic.getArgs()))
  .invoke(targetObject, ic.getArgs());
  oos.writeObject(result);
 } catch (Exception e)
 { throw new RMIInvocationException(e.getMessage(), e); }
}
```

Question 8

Il ne nous reste plus qu'à gérer la méthode `get` côté client qui lui permettra de récupérer le stub d'un objet de manière dynamique. Nous allons utiliser le `ClassLoader` Java pour permettre d'effectuer un lien dynamique avec notre `RMIHandler`. Écrire la fonction `get`. **Solution**:

```
public <T> T get(String name, Class<T> clazz)
{
```

```
 return (T) Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),
  new Class[]{clazz}, new RMIHandler(name, ois, oos));
}
```

Question 9

Après toutes ces péripéties, nous touchons au but et n'avons plus qu'à implémenter le serveur distant qui se servira de notre registre. Écrire la classe `RemoteServer`. Quelles différences faut-il prendre en compte vis-à-vis du RMI Sun ?

Solution:

Aucune différence (insister sur le fait qu'on est trop fort :)) ... mis à part que notre serveur Java restera bloqué en attente de requêtes.

```
import pc2r.Registry;
import service.implementation.RemoteCalculette;

public class Server
{
        public static void main(String[] argv)
        {
                Registry pc2rReg = new Registry();
                pc2rReg.register("Add", new RemoteCalculette());
                pc2rReg.publish(10000);
        }
}
```

Question 10

Nous allons enfin pouvoir tester notre système d'appel distant. Écrire la classe `RemoteClient`. Quelles différences faut-il prendre en compte vis-à-vis du RMI Sun ?

Solution:

Aucune différence (bien ré-insister sur le fait qu'on est trop fort :))

```
import pc2r.Registry;
import service.interf.IRemoteCalculette;

public class Client
{
        public static void main(String[] argv)
        {
                Registry pc2rReg = new Registry().connect("localhost", 10000);
                IRemoteCalculette cs = r.get("Add", IRemoteCalculette.class);
                System.out.println(cs.add(2, 3));
        }
}
```

TP - Remote Callback - Distant Process

Exercice 2 – Le Pattern `RemoteCallBack`

Les appels effectués par le biais de RMI peuvent etre long. Il est souhaitable pour un client de continuer à être réactif même en attendant la réponse d'une `Remote` méthode. Ainsi on souhaite qu'un serveur puisse appeller des `callback` sur le client pour le notifier d'évènements.

Afin d'implanter ce mécanisme de rappel(callback), il est nécessaire de définir un objet de rappel coté client dont une remote reference fournie au serveur lui permttra d'interagir avec le client. En d'autre termes cet objet de rappel est créé par le client comme objet distant et sert d'intermédiaire, *de messager*, portant la réponse du serveur au client. L'intérêt d'un tel mécanisme est de permettre de rendre la communication *asynchrone*. L'objet de rappel dispose typiquement de 5 méthodes :

- Deux pour le client : `boolean is_finished()` qui permet de savoir si le calcul est terminé et `TypeResultat getResult()` qui permet ensuite de récupérer le résultat.
- Trois pour le serveur : `void put(TypeResultat)` qui dépose le résultat à la fin du calcul `void finish()` qui indique que le résultat vient d'être déposé et prévient tous les threads en attente éventuelle du résultat `unreferenced` pour la gestion explicite des références d'objets distants, remplacant en quelque sorte le ramasse miette (GC) local L'interface `IRemoteCalculette` et la classe `RemoteCalculette` demeurent inchangées.

Question 1

La calculette est désormais considérée comme une sorte de caisse enregistreuse. A ce titre, tout nombre fourni à la méthode `cumuler` est stocké dans un vecteur (`class java.util.Vector`). Modifier la calculette pour que l'opération `getCumul` devienne *longue* (au sens de prendre beaucoup de temps).

Solution:

```
public class RemoteCalculette implements IRemoteCalculette
{
 int cumul = 0;
 public RemoteCalculette() { super(); }
 public Integer eval(Integer op1, Integer op2) { return new Integer(op1.intValue() + op1.intValue()) ; }
 public void cumuler(Integer val) { cumul += val.intValue(); }
 public Integer getCumul() { return new Integer(cumul); }
 public void resetCumul() { cumul = 0; }
}
```

A partir de maintenant nous allons définir un `Remote` objet sur lequel une `Remote` référence pourra être donnée à une autre JVM. Ca sera lui notre objet de callback.

Question 2

Ecrire la classe `RappelResRMI`, décrite plus haut, destinée à porter le résultat de la méthode `getCumul`, cette classe sera instanciée coté client et aura deux membres prives :

- `result`, un entier qui sera le resultat.
- `f` , un booleen qui sera positionné a vrai lorsque le calcul cote serveur sera terminé.

Écrire maintenant l'interface `IRappelResRMI` destinée à être transmise au serveur par le client (On prendra bien soin de n'exporter que les méthodes *vraiment nécessaires*)

Solution:

RappelResRMI

```
import java.rmi.*;
import java.rmi.server.*;
 public class RappelResRMI extends UnicastRemoteObject
     implements  IRappelResRMI, Unreferenced{

     private Integer result = null;
     private boolean f = false;

     public  RappelResRMI() throws RemoteException{}

     public void finish() {f=true;}

     public boolean isFinished() {return f;}

     public void put (Integer r)throws RemoteException{
         //allonger le temps de l'operation,
         //appele par TCalculCumul de terminer
         try {synchronized(this) {wait(10);}}
         catch (InterruptedException ie){};
         result=r;
         finish();
         System.out.println("rangement rsultat "+ result.intValue());
         synchronized(this){this.notifyAll();}
     }

     public void unreferenced(){
         try{boolean b = UnicastRemoteObject.unexportObject(this,true);}
         catch (NoSuchObjectException nsoe){}
     }

     public Integer getResult() {return result;}
 }
```

ET l'interface.

```
import java.rmi.*;
public interface IRappelResRMI extends Remote {
void put(Integer result) throws RemoteException;
}
```

Question 3

Créer une nouvelle interface `IRappelCalculette` ajoutant la methode `getCumulR (IRappelResRMI)` -pour passer l'objet temporaire de rappel- et créer aussi une nouvelle classe `RappelCalculette` implantant cette interface. Les clients distants de RappelCalculette appellent maintenant méthode `getCumulR (IRappelResRMI)` pour obtenir le résultat du cumul (et non plus la méthode `getCumul()`).

Solution:

L'interface

```
import java.rmi.*;

public interface IRappelCalculette extends IRemoteCalculette{
 Integer getCumulR(IRappelResRMI resCumul)  throws RemoteException;

}
```

La classe

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
//inchangee pour rappel
public class RappelCalculette  extends RemoteCalculette // Objets distants
    implements IRappelCalculette{

    Vector vals = new Vector(200,100);
    int k=0;

    public RappelCalculette()  throws RemoteException{super();}

    //redefinie : on ne calcule pas incrementalement, ajout valeur
    public void cumuler (Integer val) throws RemoteException{
        vals.add(k++,val);  }

    //redefinie pour simuler un calcul 'plus long'
    public Integer getCumul() throws RemoteException {
        int i; int r=0;
        for (i=0; i<k;i++){r += ((Integer)vals.elementAt(i)).intValue();}
        cumul= r;
        return new Integer(r);
    }

    //avec l'objet de rappel
    public Integer getCumulR( IRappelResRMI res)  throws RemoteException
    {//lance le calcul dans un thread spar
        new TCalculCumul(this,res).start();
        System.out.println("lancement Thread ");Thread.yield();
        return new Integer(0);
    }

    //redefinie pour coherence
    public void resetCumul()throws RemoteException{
        super.resetCumul();
        int i;
        for (i=0; i<k;i++) {vals.add(i, null);}
        k=0;
    }
}
```

Question 4

Pour que le client puisse, en théorie, réaliser d'autres tâches sans attendre la réponse calculée par le serveur d'objet, la méthode `getCumulR` va démarrer un *thread* `TcalculCumul` qui prend en charge le travail. Pour ce faire, il devra nécessairement avoir accès à la `calculette` ainsi qu'au messager `IRappelResRMI` (qui lui seront donc passés en paramètre). Ecrire la classe de `Thread TcalculCumul` dont la méthode `run` :

- s'endort un peu (pour rallonger artificiellement l'opération)
- appelle `getCumul()` sur l'objet réel du serveur
- définit le résultat et marque l'opération comme terminée dans le messager coté client (par l'intermédiaire de son représentant coté serveur : méthode `RappelResRMI.finish()`)

Solution:

```
public class TCalculCumul extends Thread{
```

```
    //s'execute cote serveur
    IRappelCalculette calculette;
    IRappelResRMI res;
    public TCalculCumul(  IRappelCalculette rc,  IRappelResRMI r){
        //reoit l'objet local calculette et
        //l'objet distant attendant le resultat
        calculette=rc;
        res=r;
    }

    public void run(){
        try{
            //appel methode de l'objet local
            Integer r = calculette.getCumul();//supposee "longue"...
            //appel de l'objet distant pour definir le resultat
            res.put(r);
        }
        catch (Exception e){e.printStackTrace();}
    }
}
```

Question 5

Quelle(s) modification(s) apporter dans le serveur ? Ecrire le serveur nommé `ServRappelCalculette.java`.
Solution:

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;

public class ServRappelCalculette { //classe ordinaire, cote serveur
    public static void main (String args[]) {
 String URL;

        if (args.length < 3)
            {URL="//localhost:1234/calculette";
                System.err.println("AlistRegister : usage host port object"
        + " par dfaut   " + URL);
            }


       else { URL = "rmi://"+args[0]+":"+args[1]+"/"+args[2];}
        //creation objet
        try {
            RemoteCalculette calculRappel = new RappelCalculette();
            Naming.rebind(URL,calculRappel);
        }
        catch(RemoteException e) {
            System.err.println(URL+":  serveur rmi indisponible" + "(" +
                               e.getMessage()+")");
            e.printStackTrace();
            System.exit(1);
        }
        /* en cas de bind
            catch(AlreadyBoundException e) {
```

```
            System.err.println(URL+": nom dj associ  un objet ");
            System.exit(1);
            }*/
        catch(MalformedURLException e) {
            System.err.println(URL+" :  URL syntaxiquement incorrecte");
            System.exit(1);
        }
    }
}
```

Question 6

Modifier le client pour appeler `getCumulR()` et attendre que le résultat soit disponible, c'est-à-dire que `isFinished()` du messager renvoie vrai. On portera une attention particulière au fait que l'attente active doit être évitée (sinon le mécanisme de rappel n'est pas utile).

Solution:

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
public class ClientPourServAutonomeCalc { //classe ordinaire, cot serveur
    public static void main (String args[]) {
 String URL="";
 int port;

        try {
            if (args.length>0){port=Integer.valueOf(args[0]).intValue();}
            else {port =1234;}

            Registry reg= LocateRegistry.getRegistry(port);


            IRemoteCalculette calc = (IRemoteCalculette)
                reg.lookup("/calc");
            System.out.println("apres lookup");
            System.out.println("apres eval " +
                               calc.eval(new Integer(5),new Integer(5)));


        }
        catch(RemoteException e) {
            System.err.println(URL+":  serveur rmi indisponible"+"("+e.getMessage()+")");
            System.exit(1);
        }

        catch(Exception e) {
            System.err.println(URL+":  serveur rmi indisponible"+"("+e.getMessage()+")");
            System.exit(1);
        }
    }
}
```