TP 1 Architectures Logicielles Distribuées :

JAVA RMI

Références distantes, Sérialisation et Téléchargement du code.

Année 2021 / 2022

Réalisé par :

- BENARAB Hanane

Encadré par :

- SERIAI Abdelhak-Djamel

Introduction :

Dans ce premier TP d'architectures logicielles distribués, nous allons nous intéresser à Java RMI (**Remote Methode Invocation**), a ses références distantes, à la sérialisation ainsi que le téléchargement du code.

Java RMI est un système qui permet à des objets s'exécutant dans une JVM d'invoquer des méthodes sur un objets s'exécutant dans une autre JVM, tout en en cachant au programmeur les détails de connexion et de transport.

C'est ce que nous allons mettre en œuvre, en nous plaçons dans le cadre d'un cabinet vétérinaire, ou chaque patient (les animaux) possède une fiche avec un dossier de suivi. Chacun des vétérinaires du cabinet peut accéder à ces fiches. Nous avons donc un serveur de fiches et des programmes client pour les vétérinaires.

Comme notre application sera distribuée sur la même machine, on a devisé notre projet en trois :

- un projet '**common**' contenant toutes les entités accessibles au serveur et client, telles que les interfaces des objets distants et éventuellement des classes désignant des objets non distants mais sérialisables (**implémentant l'interface java.io.Serializable**).

- un projet '**client**' contenant la classe Client et les classes qui devraient être reconnues par le serveur via le codebase.

- un projet '**serveur**' contenant la classe Serveur et toutes les classes des objets distants, étendant '**java.rmi.server.UnicastRemoteObject'** et implémentant chacune l'interface correspondante de son objet distant.

Objectifs du TP :

- Comprendre la notion d'objets distribuée.
- Utiliser le passage stub ou objet sérialisé.
- Utiliser le téléchargement de code via le codeBase.

1- Une première version simple :

Dans cette première partie du TP, nous avons mit en place une version de base sur laquelle on va pouvoir travailler.

1. Nous avons donc commencé par créer une interface distante publique « **InterfaceAnimal** » qui étend de l'interface 'Java.rmi.Remote', qui déclare des méthodes qui indique dans sa clause '**throws**' en plus de toute exception à l'application. Cette exception permet de capturer les différentes erreurs pouvant survenir lors de l'exécution dans un environnement distribué. Par exemple, une exception retournée lors de la coupure de la connexion entre les sockets utilisées pour établir une connexion réseau, ou lors de l'accès distant à une référence non existante.

Ces méthodes vont nous permettre d'afficher des informations relatives aux patients (nom, nom du maitre, l'espèce et la race), comme le démontre le code suivant :

```
package JavaRMI;

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

public interface InterfaceAnimal extends Remote {

    public String getNom() throws RemoteException;
    public void setNom(String nom) throws RemoteException;


    public String getNomDuMaitre() throws RemoteException;
    public void setNomMaitre(String nomMaitre) throws RemoteException;

    public String getEspeceAnim() throws RemoteException;
    public void setEspeceAnim(String especeAnim) throws RemoteException;

    public String getRace() throws RemoteException;
    public void setRace(String race) throws RemoteException;
```

Ensuite, nous avons créé une classe « **AnimalImp** » qui implémente l'interface « **InterfaceAnimal** », qui hérite de '**UnicastRemoteObject**'. Cette classe va implémenter toutes les méthodes que nous avons spécifier dans l'interface « **InterfaceAnimal** »

```java
package JavaRMI;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class AnimalImp extends UnicastRemoteObject implements InterfaceAnimal{


    public String nom;
    public String nomMaitre;
    public String especeAnim;
    public String race;
    public InterfaceDossierSuivie DossierSuivie;


    public InterfaceEspece Espece;

    public AnimalImp() throws RemoteException {

        this.nom = "";
        this.nomMaitre = "nomMaitre";
        this.especeAnim = "especeAnim";
        this.race = "race";

  public String getNom() {
      return nom;
  }

  public String getNomDuMaitre() {

      return nomMaitre;
  }

  public String getEspeceAnim() {
      return especeAnim;
  }

  public String getRace() {
      return race;
  }

  public void setNom(String nom) {
      this.nom = nom;
  }

  public void setNomMaitre(String nomMaitre) {
      this.nomMaitre = nomMaitre;
  }

  public void setEspeceAnim(String especeAnim) {
      this.especeAnim = especeAnim;
```

Après avoir créé la classe qui implémente l'interface et les méthodes distantes, nous avons écrit une classe serveur qui dispose d'une méthode principal '**main**' qui créé une instance de l'objet distant Animal qui a un nom, nom de son maitre, sa race et son espèce.

Nous savons qu'un client pour qu'il puisse invoquer une méthode, il doit obtenir un '**STUB**' pour l'objet distant Animal, pour cela nous avons fait appel a la méthode statique '**LocateRegistry.createRegistry'** (qui crée et exporte un registre a la machine locale qui accepte des appels dans le port spécifié (1099 par défaut)).

Ensuite, nous avons lié un nom unique au stub de l'objet distant 'Animal' avec la méthode de liaison '**Registry.bind**', qui sera enregistré dans un registre RMI.

Si aucun registre n'est exécuté sur le port TCP 1099 de l'hôte local lorsque la méthode de liaison est invoquée, le serveur échouera avec une RemoteException et renvoie '**Registry not found'**.

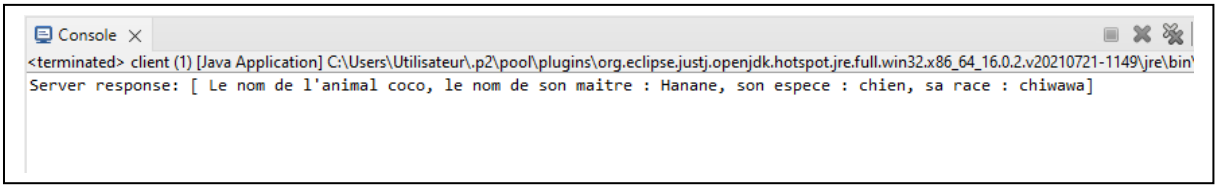
```
package JavaRMI;

import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class serveur {

public serveur() {}

    public static void main(String[] args) {

        try {
            AnimalImp Animal =new AnimalImp("coco","Hanane", "chien", "chiwawa");
            Registry registry = LocateRegistry.createRegistry(1099);

            if (registry == null)

                System.err.println("Registry not found");
                else {

                registry.bind("ObjAnimal", Animal);

                System.err.println("Server is ready");}
        }
            catch (RemoteException e) {

            e.printStackTrace();
        } catch (AlreadyBoundException e) {

                e.printStackTrace();
            }
    }
}
```

en exécutant le programme :

Nous avons créé une classe client qui va récupérer l'animal en question, en commençant par obtenir un stub pour le registre sur l'hôte du serveur avec la méthode '**LocateRegistrygetRegidtry(host)**'. Il va ensuite appeler la méthode '**lookup**' sur le stub de registre pour obtenir le stub de l'objet distant à partir du registre du serveur.

```
package JavaRMI;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class client {
    private client() {}

    public static void main(String[] args) {

        String host = (args.length < 1)? null : args[0];

        try {

            Registry registry = LocateRegistry.getRegistry(host);
            InterfaceAnimal stub = (InterfaceAnimal) registry.lookup("ObjAnimal");

            System.out.println("Server response: [ Le nom de l'animal " +stub.getNom()+", le nom de son maitre : "
                      +stub.getNomDuMaitre() + ", son espece : "+stub.getEspece() + ", sa race : "+stub.getRace()+"]");}

        catch (RemoteException | NotBoundException e) {
            e.printStackTrace();

            }
    }
}
```

En exécutant le programme, nous avons eu le résultat suivant :



2. Nous avons ajouté un gestionnaire de sécurité, qui permet de définir à notre classe serveur une politique de sécurité et de vérifier si cette politique de sécurité est bien respectée. Cette politique permet de contrôler les interactions possibles avec les autres classes en spécifiant des permissions. Toute interaction non autorisée par la politique de sécurité engendre une exception de type '**java.security.SecurityException**'. Cela ce fait, en créant un fichier '**Security.policy**' qui autorise l'usage du accept et du connect sur les sockets java :

```
grant{
permission  java.net.SocketPermission "*:1024-65535","connect,accept";
permission  java.net.SocketPermission ":80","connect";
//  permission  java.security.AllPermission;
}
```

Et dans la partie serveur, nous avons rajouté le gestionnaire de sécurité :

```
if (System.getSecurityManager()==null)     // Creer un gestionnaire de sécurité
System.setSecurityManager(new SecurityManager());
```

3. Nous avons rajouté à notre objet distant « animal » un dossier de suivi pour que le client puisse récupérer et modifier ce dernier, pour cela nous avons exactement fait comme précédemment, créer une interface « InterfaceDossierSuivie », ou on a déclaré 2 méthodes, 'getSuivi' et 'setSuivi'.

```
package JavaRMI;

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

public interface InterfaceDossierSuivie extends Remote{

    public String getSuivi() throws RemoteException;

    public void setSuivi(String suivi) throws RemoteException;


}
```

Ensuite, nous avons créé une classe « DossierSuivieImpl » qui va implémenter l'interface « InterfaceDossierSuivie » et qui spécifie les deux méthodes qu'on a déclaré dans l'interface.

```
package JavaRMI;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class DossierSuivieImpl extends UnicastRemoteObject implements InterfaceDossierSuivie{
    public String suivi;


    public DossierSuivieImpl() throws RemoteException{}


    public String getSuivi() throws RemoteException {

        return this.suivi;
    }

    public void setSuivi(String suivi) throws RemoteException {
        this.suivi=suivi;
    }
```

Après avoir créé l'interface et l'implémentation de « DossierSuivi », nous avons réalisé une association avec la classe « AnimalImpl » en rajoutant un attribut DossierSuivi de type « InterfaceDossierSuivi » :

```
package JavaRMI;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class AnimalImp extends UnicastRemoteObject implements InterfaceAnima


    public String nom;
    public String nomMaitre;
    public String especeAnim;
    public String race;
    public InterfaceDossierSuivie DossierSuivie;
```

Nous avons ainsi rajouté les méthodes distantes dans l'interface « InterfaceAnimal » pour que le client puisse utiliser ces méthodes distantes :

```
public String getNomDuMaitre() throws RemoteException;
public void setNomMaitre(String nomMaitre) throws RemoteException;

public String getEspeceAnim() throws RemoteException;
public void setEspeceAnim(String especeAnim) throws RemoteException;

public String getRace() throws RemoteException;
public void setRace(String race) throws RemoteException;

public InterfaceDossierSuivie getDossierSuivi() throws RemoteException;
public void setDossierSuivi(InterfaceDossierSuivie DossierSuivi) throws RemoteException;

public String getSuivi() throws RemoteException;
public void setSuivi(String suivi) throws RemoteException;

public InterfaceEspece getEspece() throws RemoteException ;
```

4. Nous avons rendu l'espèce de l'animal en une classe qui contient le nom de l'espèce et sa durée de vie comme attributs, nous avons mis en œuvre cette application de façon que le client puisse consulter l'espèce d'un animal sans pouvoir le modifier sachant qu'on transmettra des copies d'espèces et pas des références distantes. Cela veut dire que cet objet, doit être sérialisable et ne sera associé à aucun proxy le référençant.

```
package JavaRMI;

import java.io.Serializable;


public interface InterfaceEspece extends Serializable{


    public String getNomEspece();
    public int getDureeVie();
    public void setNom(String nomEspece);


    }
```

Pour chaque interface, il faut une classe qui l'implémente, de ce fait, nous avons créé une classe « especeImp » avec une méthode 'get' qui permet de consulter l'espèce d'un animal sans pouvoir le modifier.

```java
package common;

public class EspeceImpl implements InterfaceEspece {


    private String nomEspece;
    private int DureeVie;


    public EspeceImpl () {}


    public EspeceImpl(String nomEspece, int DureeVie) {
        this.DureeVie=DureeVie;
        this.nomEspece=nomEspece;
    }


    public String getNomEspece() {
        return nomEspece;
    }


    public int getDureeVie() {
        return DureeVie;
    }
}
```

2- Classe Cabinet Vétérinaire :

Dans cette deuxième partie du TP, nous avons modifié notre application de sorte que la classe Cabinet Vétérinaire que nous avons créé, qui connait sa collection d'animaux soit distribuée.

```java
package JavaRMI;

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

public interface IntCabinet extends Remote{

    public InterfaceAnimal recherche(String nomAnimal) throws RemoteException;

}
```

Ensuite, nous avons créé une classe « CabinetImpl » qui va implémenter l'interface « IntCabinet » et qui spécifie une méthode qu'on a déclaré dans l'interface, qui va permettre au client de chercher un patient, qui renvoie le nom de l'animal s'il figure parmi les patients du cabinet, sinon il renvoie une chaine de caractères disant que l'animal est introuvable.

```java
package JavaRMI;

import java.rmi.RemoteException;

public class CabinetImpl extends UnicastRemoteObject implements IntCabinet{

    private List<InterfaceAnimal> listAnimals;

    public CabinetImpl() throws RemoteException {

            this.listAnimals = new ArrayList<InterfaceAnimal>();
        }

    public InterfaceAnimal recherche(String nomAnimal) throws RemoteException {
        for (InterfaceAnimal animal : listAnimals) {
            if (animal.Verification(nomAnimal)) {
                System.out.println(nomAnimal + " Trouvé ! ");
                return animal;
            }
        }
           System.out.println("Animal introuvable");
        return new AnimalImp();
    }
    }
```

.

3- Création de patient :

Dans cette partie, nous avons mit en place un processus qui va nous permettre de rajouter des nouveaux patients au cabinet vétérinaire. Pour cela, nous avons d'abord commencé par rajouter une méthode '**addAnimal**' dans l'interface distante « **IntCabinet** » :

```java
package common;

import java.rmi.Remote;

public interface IntCabinet extends Remote{

    public InterfaceAnimal recherche(String nomAnimal) throws RemoteException;

    public boolean addAnimal(String nom, String NomMaitre, InterfaceEspece espece,String race, String suivi) throws RemoteException;
    public boolean addAnimal(String nom, String NomMaitre, String NomEspece, int DureeVie, String race, String suivi) throws RemoteException;
}
```

Et dans « CabinetImpl », nous avons mis en place deux méthodes '**AddAnimal**', La première permet d'ajouter un animal sans besoin de spécifier son espèce. La deuxième version de la méthode permet d'ajouter un animal en spécifiant son espèce via une instance de la classe '**Espece**'. Ceci est possible parce que la classe '**Espece**' est dans le projet commons, et donc le client et le serveur la reconnaissent. Ainsi l'objet '**Espece**' est créé et initialisé du côté client avant d'être passé à la méthode addAnimal (). Le constructeur de l'animal ajouté se contentera ainsi d'affecter l'instance '**Espece**'.

```java
public boolean addAnimal(String nom, String NomMaitre, String NomEspece, int DureeVie, String race, String suivi) throws RemoteException {
        InterfaceAnimal patient = new AnimalImp(nom, NomMaitre, NomEspece ,race, DureeVie, suivi);
        return listAnimals.add(patient);
        }
public boolean addAnimal(String nom, String NomMaitre, InterfaceEspece espece,String race, String suivi) throws RemoteException {
        InterfaceAnimal patient = new AnimalImp(nom, NomMaitre, espece, race, suivi);
        return listAnimals.add(patient);
        }
}
```

Pour tester ce que l'on vient de faire, nous avons d'abord créé une instance de l'objet distant Cabinet, nous avons obtenu un '**STUB**' pour cet OD, nous avons lié un nom unique au stub 'ObjCabinet' avec la méthode de liaison '**Registry.bind**', qui sera enregistré dans un registre RMI.

nous avons évoqué la méthode « **addAnimal** », pour ajouter un patient, en précisant son nom, nom du maitre, race, suivi, durée de vie et son espèce.

Ensuite, nous avons essayé de rechercher ce patient grâce à la méthode « **Recherche** ».

```java
IntCabinet stub=(IntCabinet) registry.lookup("objCabinet");

stub.addAnimal("COCO","Hanane","chien", 5, "Chiwawa", "Parfaite Santé");
IntAnimal Anim= (IntAnimal) stub.recherche("coco");
System.out.println(Anim.getNom());
```

4 – Téléchargement de code :

Nous savons qu'en réalité, le client et le serveur ne s'exécutent pas sur la même machine comme nous l'avons mis en œuvre durant ce TP. Quand le serveur demande auprès d'une classe qu'il ne connait pas, on obtient une exception de type ' **ClasseNotFoundException**'.

C'est pour cette raison qu'on a eu recours a ce qu'on appelle 'Codebase' qui désigne un mécanisme de classpath distribué. En effet, le codebase est un dossier qui contient des fichiers bytecodes de classes.

Notre codebase est localisé dans un système de fichiers dans notre machine, il est consulté par le classpath qu'on a définit comme désigné dans la figure suivante :

```java
// Mise en place du CODEBASE :
String classpath="C:\\Users\\Utilisateur\\eclipse-workspace\\server\\bin";
System.setProperty("java.rmi.server.codebase",classpath);
```

Nous avons défini cette propriété pour le serveur, pour qu'il puisse accéder aux implémentations de classes depuis le codebase du client.

5- Alertes :

Dans cette dernière partie du TP, nous avons intégré à notre application un mécanisme qui alerte automatiquement tous les clients du même cabinet vétérinaire quand on franchit à la hausse ou à la baisse les seuils de 100,200, 500 et 1000 patients.

Pour mettre en œuvre cela, nous avons créé une méthode « Alertes » qui nous renvoie un String quand on franchit les caps soulignés ci-dessus :

```
public String alertes() {
    if (listAnimals.size()==500 )
        return "vous avez atteint les 500 patients" ;
    else if (listAnimals.size()==1000)
        return "vous avez atteint les 1000 Patients";
    else if (listAnimals.size()==100)
        return "vous avez atteint les 100 patients";
    else if (listAnimals.size()==200)
        return "vous avez atteint les 200 patients";
return null;
```

Dés qu'un patient est rajouté à la liste des patients, la méthode « alertes » va calculer la longueur de cette liste et va émettre une chaine de caractères si le nombre de patients a franchit les 100,200 ou 100 patients.

Conclusion :

Grace à ce TP, nous avons pu comprendre au mieux différentes notions vues en cours concernant JAVA RMI et les objets distants distribués.

L'application que nous avons mis œuvre, qui représente un cabinet vétérinaire, nous a permis d'utiliser les composants nécessaires à une application JAVA RMI tel que le client, le serveur et son squelette, le registre RMI, les objets distribués et leurs stubs.

Finalement, la mise en place du codebase nous a permit de voir son utilité, dans le cas où on n'est pas sur la même machine locale et qu'un serveur souhaite obtenir des classes qu'il ne connait pas.