Architectures Distribué - Java RMI

GARCIA PENA Loris

16 octobre 2023

Résumé

Ce rapport présente mon travail réalisé dans le cadre du TP1 portant sur Java RMI (Remote Method Invocation).

Table des matières

1 Introduction

La technologie Java Remote Method Invocation (RMI) est un système puissant qui permet à un objet s'exécutant dans une machine virtuelle Java d'invoquer des méthodes sur un objet s'exécutant dans une autre machine virtuelle Java distante. Cette capacité de communication à distance entre des programmes repose sur l'invocation de méthodes sur des objets distribués appellés stub. En d'autres termes, elle offre la possibilité d'interagir avec un objet distant comme s'il était local. Cela favorise la construction d'applications réparties en utilisant des appels de méthode au lieu d'appels de procédure, simplifiant ainsi le développement d'applications distribuées.

Dans le cadre de ce projet, nous nous plaçons dans le contexte d'un cabinet vétérinaire. Chaque patient du cabinet, c'est-à-dire chaque animal, possède une fiche individuelle avec un dossier de suivi médical. L'objectif est de créer un système où chaque vétérinaire du cabinet peut accéder aux fiches des patients à distance. Nous mettrons en place un serveur et développerons un client pour les vétérinaires.

2 Architecture globale

L'architecture globale de notre système repose sur trois composants principaux :

- Une partie Serveur, responsable de la gestion du cabinet medical.
- Une partie Client, utilisée par les vétérinaires pour accéder au cabinet medical.
- Une partie commune (*Common*), qui permet de structurer les composants communs.

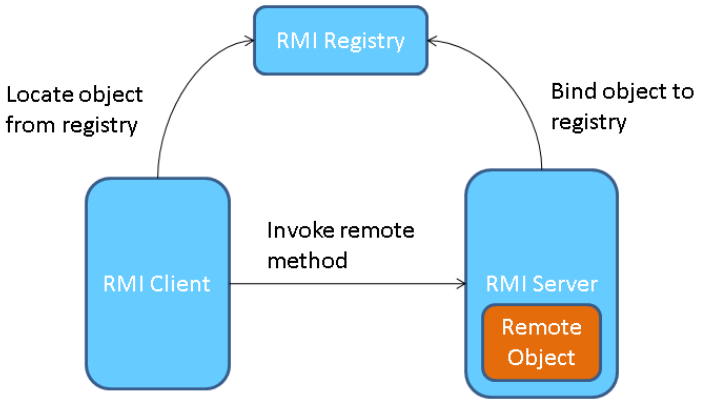

FIGURE 1 – JAVA RMI

Cette architecture permet la création d'une application distribuée utilisant Java RMI pour la communication à distance. Chaque composant joue un rôle essentiel dans le fonctionnement de notre application vétérinaire. Dans notre application, chaque composant est implémenté en tant que projet distinct.

2.1 Partie Serveur

Le serveur est responsable de la création d'objets, dans notre cas, le cabinet médical, qui sont rendus disponibles pour l'accès distant via Java RMI. Le serveur crée ces objets et les publie dans un registre RMI. Le registre RMI agit comme un annuaire central permettant aux clients d'accéder aux objets distants.

Ainsi, le serveur joue un rôle essentiel dans la mise en place de l'infrastructure RMI, permettant aux différentes clients (vétérinaire) d'accéder aux différentes fonctions disponibles dans le cabinet médical. Il met en place une communication sécurisée entre les clients et les objets distants (stub). Le serveur gère également la gestion des RMI Callback pour alerter les clients lorsque certains seuils sont atteints, améliorant ainsi la réactivité et la gestion du cabinet médical.

2.2 Partie Clients

Le composant client de notre projet est destiné aux vétérinaires du cabinet. Chaque vétérinaire utilise un client pour interagir avec le cabinet vétérinaire à distance. Lorsqu'un vétérinaire effectue une opération, le client communique avec le serveur en utilisant Java RMI.

Chaque client utilise un objet stub pour représenter l'interface distante de notre cabinet médical.

L'objet stub agit comme un proxy local pour l'objet distant du serveur. Lorsqu'un client appelle une méthode sur l'objet stub, la requête est automatiquement acheminée vers le serveur.

Les clients utilise une interface en ligne de commande (CLI) pour effectuer diverses actions, telles que la recherche de patients, l'ajout de nouveaux animaux, la consultation des dossiers médicaux, etc.

De plus, les clients utilisent des méthodes qui déclenchent des affichages dans le terminal du serveur, permettant aux vétérinaires de rester informés en temps réel sur les opérations effectuées sur le cabinet médical.

Enfin, les clients déposent les fichiers de classes ('.class') dans un répertoire 'codebase', permettant ainsi au serveur de les télécharger et de les rendre accessibles (voir section Codebase en Java RMI).

2.3 Partie commune (*Common*)

Le module commun, également appelé *Common*, est utilisé en tant que composant partagé entre le serveur et les clients. Il contient les classes et interfaces nécessaires à la communication entre le serveur et les clients. En utilisant ce module, nous garantissons la cohérence des données partagées et simplifions le développement en évitant la duplication de code.

Le module commun contient des définitions d'objets partagés, telles que les interfaces des différents objets distribués, mais aussi des classes partagées.

Conclusion

L'architecture de notre projet basé sur Java RMI offre donc une solution robuste pour la gestion d'un cabinet vétérinaire à distance. Elle se compose de trois parties, chacune de ces parties remplit un rôle distinct, contribuant ainsi à la création d'une application distribuée fonctionnelle.

Le serveur centralise la création et la gestion des objets distants, tandis que les clients utilisent des objets stub pour interagir avec l'interface distante. L'architecture garantit une communication à distance fluide et sécurisée.

En résumé, l'architecture globale de notre projet simplifie la gestion des opérations vétérinaires à distance.

3 Diagrammes de Conception

Le diagramme de classes présenté ci-dessous (Figure 2), offre une vue globale des relations entre les composants clés de notre projet, notamment les classes, les interfaces, et les flux de données.

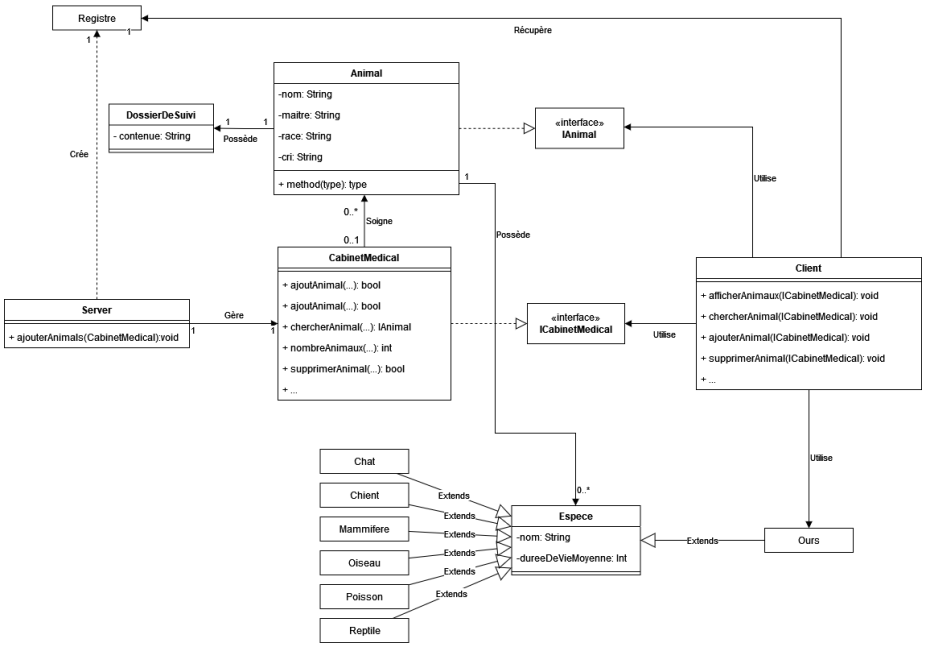

FIGURE 2 – Diagramme de classes

Ce diagramme de classe met en évidence les composants qui interagissent pour fournir une solution distribuée. Au cœur de notre système se trouve la classe `CabinetMedical`, qui joue un rôle central dans la gestion des dossiers médicaux des patients. C'est dans cette classe que l'on retrouve notre liste de patients.

Les classes `CabinetMedical` et `Animal` implémentent les interfaces définies dans la partie commune. Cela permet aux clients de manipuler des objets distants via des objets stub générés à partir des interfaces communes.

Le `Client` s'appuie sur ces interfaces pour interagir avec le serveur et les objets distants (Cf. Composants clés).

De plus, le diagramme de classes met en évidence un élément essentiel de notre architecture Java RMI : le registre RMI.
Le registre RMI joue un rôle crucial en servant de répertoire central pour la publication des objets distants :

```
1    registry.bind("Cabinet", cabinet);
```

Cela permet aux clients d'accéder facilement aux objets distants en utilisant un mécanisme de recherche :

```
1     ICabinetMedical cabinet = (ICabinetMedical) registry.lookup("
      Cabinet");
```

La présence du registre RMI dans notre diagramme souligne son rôle fondamental dans la mise en place de la communication à distance au sein de notre application, renforçant ainsi la gestion des objets distants et des clients.

4 Communication

La communication dans notre application Java RMI repose sur le mécanisme de communication à distance offert par RMI. Elle permet aux vétérinaires (clients) d'interagir avec le cabinet médical (serveur) de manière transparente, comme s'ils travaillaient en local.

4.1 Invocation de Méthodes à Distance

L'invocation de méthodes à distance est la base de la communication dans notre application. Grâce à Java RMI, les vétérinaires peuvent invoquer des méthodes définies dans l'interface `ICabinetMedical` sur des objets distants situés côté serveur. Ces méthodes permettent d'ajouter, chercher, supprimer des animaux, etc.

Lorsqu'un vétérinaire appelle l'une de ces méthodes, RMI utilise automatiquement un objet stub pour gérer la communication à distance. Les paramètres sont sérialisés, les appels sont acheminés vers le serveur, et les résultats sont renvoyés au client :

```
1 ICabinetMedical cabinet = (ICabinetMedical) registry.lookup("
    Cabinet");
2 cabinet.ajoutAnimal(nom, maitre, race, nomEspece, dureeDeVieMoyenne
    , cri);
```

Cette approche rend la communication à distance transparente et simplifie la gestion des opérations à travers des objets proxy locaux.

4.2 Sérialisation des objets dans l'application RMI

L'un des aspects clés de notre application Java RMI réside dans la capacité à transmettre des objets d'une machine virtuelle Java à une autre de manière transparente. Cela est rendu possible grâce au mécanisme de sérialisation de Java.

4.3 Mécanisme de sérialisation de Java

La sérialisation est un processus par lequel un objet Java est converti en un flux d'octets, ce qui permet de le transférer sur un réseau ou de le stocker dans un fichier. La désérialisation est le processus inverse, où un flux d'octets est converti en un objet Java. Java fournit un mécanisme de sérialisation intégré qui facilite grandement la communication entre les machines virtuelles.

Dans notre application, lorsque nous passons des objets tels que des patients ou des espèces entre le serveur et les clients, Java gère automatiquement la sérialisation et la désérialisation. Cela simplifie considérablement le processus de développement, car les détails de la sérialisation ne nécessitent pas d'intervention directe de notre part.

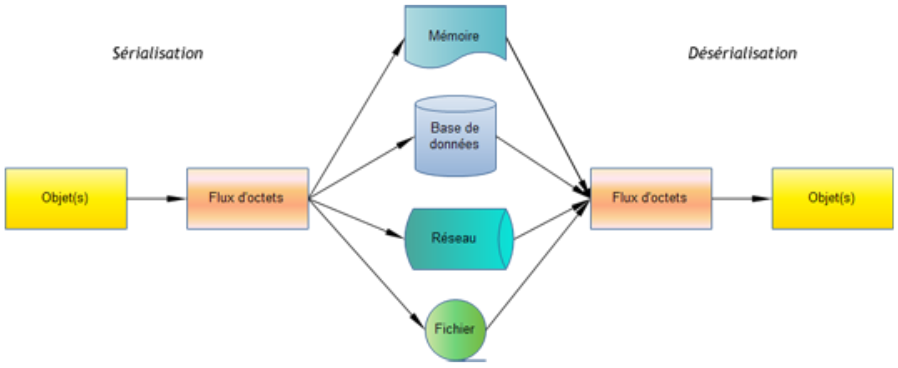

FIGURE 3 – Sérialisation dans une application

4.4 Sérialisation implicite

Dans notre application, la classe `CabinetMedical` est un composant central qui gère les dossiers des patients. Bien que cette classe n'implémente pas directement l'interface `Serializable`, elle bénéficie de la sérialisation grâce à l'extension de la classe `UnicastRemoteObject`. Cette extension rend les objets de la classe `CabinetMedical` sérialisables :

```
1    public class CabinetMedical extends UnicastRemoteObject
     implements ICabinetMedical {}
```

Ainsi, lorsqu'un objet `CabinetMedical` est transmis entre le serveur et les clients, il est automatiquement sérialisé et désérialisé. Nous avons également implémenté une classe `Espece` qui étend directement `Serializable` :

```
1    public class Espece implements Serializable {}
```

Permettant au serveur de sérialiser et de désérialiser les objets de cette classe.

5 Sécurité et codebase

La sécurité est un aspect crucial de toute application distribuée, et Java RMI offre des mécanismes intégrés pour la gestion de la sécurité. Dans notre projet, nous avons commencé à mettre en place des mesures de sécurité de base.

5.1 Politique de Sécurité

Nous avons configuré une politique de sécurité en utilisant le fichier `security.policy`. Cette politique définit les autorisations et les restrictions qui seront appliquées aux composants de notre application. La politique de sécurité est chargée via la ligne de code suivante :

```
System.setProperty("java.security.policy", "security/security.
    policy");
```

Dans le fichier de politique de sécurité, situé coté serveur dans un sous dossier security, nous avons actuellement octroyé toutes les autorisations à notre application en utilisant la règle suivante :

```
grant {
    permission java.security.AllPermission;
};
```

Cela signifie que notre application a la permission d'accéder à toutes les ressources et d'effectuer toutes les actions, ce qui est une configuration très permissive et ne devrait être utilisée que pour des besoins de développement et de test.

Dans un environnement de production, il est fortement recommandé de définir des politiques de sécurité plus restrictives pour protéger les ressources sensibles et garantir l'intégrité de l'application.

5.2 Gestionnaire de Sécurité

Nous avons également mis en place un gestionnaire de sécurité avec la ligne de code suivante :

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

Le gestionnaire de sécurité est responsable de l'application des politiques de sécurité définies. Il veille à ce que les actions de l'application soient conformes aux autorisations spécifiées dans la politique de sécurité.

Bien que notre configuration actuelle soit permissive pour faciliter le développement, il est essentiel de noter que dans un environnement de production, des politiques de sécurité appropriées doivent être définies pour protéger l'application contre les menaces potentielles.

5.3 Codebase en Java RMI

La *codebase* en Java RMI est un mécanisme essentiel pour le téléchargement dynamique de classes depuis des emplacements distants. Lorsqu'un serveur RMI

souhaite utiliser une classe distante qu'il ne possède pas localement, il peut la télécharger depuis un emplacement spécifié, appelé la *codebase*.

Dans notre application, nous avons placé nos classes téléchargeables dans un répertoire situé côté client. Plus précisément dans le sous-répertoire `codebase/src/com/cabinet/client/rmi` de notre projet :

```
1     System.setProperty("java.rmi.server.codebase",
2   "file:///chemon_vers_application/TP1_Cabinet/RMI_client/codebase/
      ");
```

Cette structure permet au serveur RMI de télécharger ces classes dynamiquement lorsque les clients en ont besoin. Nous avons par exemple utilisé cette fonctionnalité pour télécharger la classe `Especе`. En effet, coté client nous avons une classe *Ours* qui extends *Espece*, cette classe n'est donc pas accessibles coté serveur, nous devons donc la mettre dans notre codebase.

```
1 import com.cabinet.common.rmi.Especе;
2
3 public class Ours extends Especе {
4     public Ours() {
5         super("Ours", 50);
6     }
7 }
```

Lorsque le client émet une requête nécessitant l'utilisation d'une classe distante, le serveur RMI vérifie la *codebase* spécifiée pour cette classe. Si la classe n'est pas présente localement sur le serveur, il la télécharge automatiquement depuis la *codebase* du client. Cela garantit que les classes requises sont accessibles et utilisables, même si elles ne sont pas distribuées avec le serveur RMI lui-même.

6 Composants clés

Pour comprendre en détail les composants clés de notre application Java RMI, examinons les principales classes et interfaces qui les composent.

6.1 Composants communs

Comme sité précédemment, les classes et interfaces communes sont utilisées par le serveur et les clients. Elles permettent de définir une structure commune pour les objets distants (stub).

· Interface IAnimal

L'interface `IAnimal` est une interface qui définit les méthodes que tout animal doit mettre en œuvre. Elle étend l'interface `Remote`, ce qui permet aux objets qui la mettent en œuvre d'être distribués via Java RMI. Cette interface définit des méthodes pour obtenir le nom de l'animal, afficher des informations sur l'animal, émettre un cri, consulter le dossier de suivi, obtenir des informations sur l'espèce de l'animal, etc :

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3
4  public interface IAnimal extends Remote {
5    /* METHODS */
6    String getNom() throws RemoteException;
7    String afficherAnimal() throws RemoteException;
8    String afficherAnimalComplet() throws RemoteException;
9    void crier() throws RemoteException;
10   String afficherDossierDeSuivi() throws RemoteException;
11   void modifierDossierDeSuivi(String contenue) throws
       RemoteException;
12   Espece getEspece() throws RemoteException;
13 }
```

Chacun des méthodes de nos interfaces étend l'interface `RemoteException`, qui est une exception qui doit être déclarée par toutes les méthodes distantes afin de signaler les problèmes de communication à distance.

· Interface ICabinetMedical

De manière similaire, l'interface `ICabinetMedical` est une interface qui définit les méthodes que tout cabinet médical doit mettre en œuvre. Elle étend également l'interface `Remote`. Cette interface définit des méthodes pour ajouter des animaux, chercher des animaux, supprimer des animaux, obtenir la liste des patients, etc :

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  import java.util.ArrayList;
4
5  public interface ICabinetMedical extends Remote {
```

```
6
7           boolean ajoutAnimal(String nom, String maitre, String race,
        Especе especе, String cri) throws RemoteException;
8           boolean ajoutAnimal(String nom, String maitre, String race,
        Especе especе, String cri, String etat) throws RemoteException
        ;
9           boolean ajoutAnimal(String nom, String maitre, String race,
        String nomEspece, int dureeDeVieMoyenne, String cri) throws
        RemoteException;
10          boolean ajoutAnimal(String nom, String maitre, String race,
        String nomEspece, int dureeDeVieMoyenne, String cri, String
        etat) throws RemoteException;
11          int nombreAnimaux() throws RemoteException;
12          IAnimal chercherAnimal(String nom) throws RemoteException;
13          boolean supprimerAnimal(String nom) throws RemoteException;
14          ArrayList<IAnimal> getPatients() throws RemoteException;
15  }
```

Nous pouvons voir que cette interface définit plusieur méthodes pour ajouter des animaux, en fonction des paramètres fournis. Cela permet de simplifier l'ajout d'animaux en fournissant des méthodes surchargées.

- **Classe Especе**

La classe `Especе` est un composant commun qui définit une espèce d'animal. Elle implémente l'interface `Serializable` pour permettre la sérialisation des objets lors de la communication à distance. Cette classe contient des propriétés telles que le nom de l'espèce et la durée de vie moyenne :

```
1  public class Especе implements Serializable {
2      private String nom;
3      private int dureeDeVieMoyenne;
4
5      public Especе(String nom, int dureeDeVieMoyenne) {
6          this.nom = nom;
7          this.dureeDeVieMoyenne = dureeDeVieMoyenne;
8      }
9
10     public String getNom() {
11         return nom;
12     }
13
14     public int getDureeDeVieMoyenne() {
15         return dureeDeVieMoyenne;
16     }
17 }
```

Cette classe nous permet d'aborder un aspect important de notre application, l'envois par copie. Lorsqu'un client appelle une méthode sur un animal distant, sont espèce lui est envoyés par copie. Cela signifie qu'un client ne peut pas modifier l'especе d'un aniaml directement.

Elle nous permet également d'illustrer le mécanisme de téléchargement de code via l'utilisation de la codebase (Cf. Codebase en Java RMI), avec l'utilisation de la classe `Ours`.

6.2 Composants serveur

Les classes présentes dans le package `com.cabinet.server.rmi` sont utilisées par le serveur pour gérer les patients. Elles orbite autour de la classe `CabinetMedical`. Un server est lui représenté par une classe `Server`. Cette classe s'occupe de générer un registre RMI, de créer un cabinet médical et de le publier dans le registre RMI. Il est également responsable de l'ajout préliminaire de patients au cabinet médical :

```
CabinetMedical cabinet = new CabinetMedical();
ajouterAnimals(cabinet);
Registry registry = LocateRegistry.createRegistry(1099);
registry.bind("Cabinet", cabinet);
```

· Classe DossierDeSuivi

La classe `DossierDeSuivi` est une classe qui représente le dossier de suivi d'un animal. Elle contient simplement une chaîne de caractères qui représente le contenu du dossier. Cette classe nous permet de montrer qu'un client peut modifier un dossier de suivi d'un animal distant en appelant la méthode `modifierDossierDeSuivi`.

· Classe Animal

La classe `Animal` est une implémentation de l'interface `IAnimal`. Elle représente un patient du cabinet vétérinaire et stocke des informations telles que le nom, le maître, l'espèce, la race, le cri et le dossier de suivi de l'animal :

```
public class Animal extends UnicastRemoteObject implements IAnimal
    {

  private String nom;
  private String maitre;
  private Espece espece;
  private String race;
  private String cri;
  private DossierDeSuivi dossierDeSuivi;
    ...
}
```

Cette classe est utilisée côté serveur pour créer et gérer les fiches médicales des animaux. Elle possède plusieurs contructeurs paramétrés différents pour permettre la création d'animaux selon différents paramètres :

```
protected Animal(String nom, String maitre, String race, Especе
    espece, String cri) {...}
protected Animal(String nom, String maitre, String race, Especе
    espece, String cri, String etat) {...}
protected Animal(String nom, String maitre, String race, String
    nomEspece, int dureeDeVieMoyenne, String cri) {...}
protected Animal(String nom, String maitre, String race, String
    nomEspece, int dureeDeVieMoyenne, String cri, String etat)
    {...}
```

Ansi, un client peut créer un animal utilsant un espece prédéfinis appartenant à la classe `Espece` ou il peut créer un animal avec une espece personnalisé en fournissant le nom de l'espece et sa durée de vie moyenne. Il a également la possibilité de fournir un état à l'animal, qui peut être "malade", "blessé" ou autre.

· Classe CabinetMedical

La classe `CabinetMedical` est le cœur du côté serveur. Elle gère une liste d'animaux (patients) (liste d'*Animal*) au sein du cabinet médical. Cette classe expose les méthodes définies dans l'interface `ICabinetMedical` pour gérer les patients, rechercher des animaux, supprimer des patients, etc :

```
public class CabinetMedical extends UnicastRemoteObject implements
    ICabinetMedical {

    private ArrayList<IAnimal> patients;
    ...
}
```

Cette classe est également responsable de la gestion des RMI Callback.

· Différentes espece

Dans le serveur on retrouve également de nombreuses classes qui représentent des espèces d'animaux. Ces classe implements toutes la classe commune `Espece` :

```
public class Oiseau extends Espece {
    public Oiseau() {
        super("Oiseau", 5);
    }
}
```

6.3 Composants client

Les classes présentes dans le package `com.cabinet.client.rmi` sont utilisées par les clients pour interagir avec le cabinet médical. Elles orbite autour de la classe `Client`. Cette classe est responsable de la gestion des interactions entre les vétérinaires et le cabinet médical. Nous utilisons un CLI (Command Line Interface) pour permettre aux vétérinaires (user) d'interagir avec le cabinet médical. Un client s'occupe donc de récupérer le cabinet médical depuis le registre RMI de gérer les interactions avec le cabinet médical, et d'afficher les résultats dans le terminal :

```
ICabinetMedical cabinet = (ICabinetMedical) registry.lookup("
    Cabinet");
System.out.println("\nChoisir une action : ");
            System.out.println("1. Afficher tous les animaux (court
    )");
            System.out.println("2. Afficher tous les animaux (
    complet)");
```

```
5               System.out.println("3. Chercher un animal");
6               System.out.println("4. Ajouter un animal");
7               System.out.println("5. Supprimer un animal");
8               System.out.println("6. Ajouter un Ours");
9               System.out.println("7. Ajouter beaucoup d'animaux");
10              System.out.println("8. Quitter");
```

Classes Ours

Côté client, nous trouvons la classe `Ours` qui hérite de la classe `Espece`. Elle représente une espèce spécifique (l'ours) et est utilisée pour créer des animaux de cette espèce. Cette classe est utilisée pour illustrer le mécanisme de téléchargement de code (codebase). En effet, cette classe n'est pas présente côté serveur, elle est donc téléchargée depuis la codebase du client.

7 RMI Callback

Dans notre application Java RMI, nous utilisons le mécanisme de callback pour informer les clients des seuils atteints en matière de nombre de patients dans le cabinet médical. Cette fonctionnalité est utilisé pour alerter les vétérinaires lorsqu'un certain nombre de patients est atteint.

· Côté client

Dans le package `com.cabinet.common.rmi`, nous avons défini l'interface `IClientCallback`, qui étend l'interface `Remote`. Cette interface contient une méthode, `notifierSeuilAtteint`, qui est appelée par le serveur pour notifier un client de l'atteinte d'un seuil :

```
1 public interface IClientCallback extends Remote {
2     void notifierSeuilAtteint(int nombrePatients) throws
      RemoteException;
3 }
```

· Côté serveur (dans CabinetMedical)

Nous avons une liste, `listeClients`, qui stocke les clients enregistrés pour les notifications de seuil. Le serveur offre une méthode, `enregistrerAlertCallback`, qui permet d'enregistrer un client pour les alertes. Si un client n'est pas déjà enregistré, il est ajouté à la liste :

```
1 private ArrayList<IClientCallback> listeClients;
2
3 public void enregistrerAlertCallback(IClientCallback client) throws
       RemoteException {
4     if (!listeClients.contains(client)) {
5         listeClients.add(client);
6     }
7 }
8
```

```
9  public void supprimerAlertCallback(IClientCallback client) throws
       RemoteException {
10     if (!listeClients.contains(client)) {
11         listeClients.remove(client);
12     }
13 }
14
15 private void notifierSeuilAtteint(int nombrePatients) throws
       RemoteException {
16     for (IClientCallback client : listeClients) {
17         try {
18             client.notifierSeuilAtteint(nombrePatients);
19         } catch (RemoteException e) {
20             supprimerAlertCallback(client);
21         }
22     }
23 }
24
25 private void verifierSeuilAtteint() throws RemoteException {
26     int nombrePatients = patients.size();
27     if (nombrePatients == 100 || nombrePatients == 500 ||
       nombrePatients == 1000) {
28         notifierSeuilAtteint(nombrePatients);
29     }
30 }
```

C'est également dans cette classe que nous vérifions si un seuil est atteint. Si un seuil est atteint, nous appelons la méthode `notifierSeuilAtteint` pour informer les clients. Cela va donc appeller la méthode `notifierSeuilAtteint` de chaque client enregistré.

- **Côté client**

Dans le package `com.cabinet.client.rmi`, nous avons implémenté la classe `ClientCallbackImpl` qui étend `UnicastRemoteObject` et implémente l'interface `IClientCallback`. Cette classe fournit l'implémentation de la méthode `notifierSeuilAtteint`, qui affiche une alerte sur la console client indiquant le nombre de patients atteint :

```
1  public class ClientCallbackImpl extends UnicastRemoteObject
       implements IClientCallback {
2
3  public void notifierSeuilAtteint(int nombrePatients) throws
       RemoteException {
4      System.out.println("Alerte : Le nombre de patients est de " +
       nombrePatients);
5  }
6  }
```

Dans `Client.java`, nous créons une instance de `ClientCallbackImpl` et l'enregistrons auprès du serveur en appelant la méthode `enregistrerAlertCallback` du cabinet médical :

```
1 ClientCallbackImpl clientCallback = new ClientCallbackImpl();
2 cabinet.enregistrerAlertCallback(clientCallback);
```

Ce mécanisme de callback permet aux clients d'être informés en temps réel des seuils atteints.

8 Read-Me

Pour executer et tester le projet, il faut suivre les étapes suivantes (avec l'IDE eclipse) :

- Ouvrir le dossier TP1_Cabinet dans eclipse.
- Ajouter RMI_common au build path si ce n'est pas déja fait.
- Modifier le chemin du gestionnaire de sécurité en fonction de sa machine : System.setProperty("java.rmi.server.codebase", "file :///chemin_vers_projet/TP1_Cabinet/RMI_client/codebase/").
- Lancer le serveur (Server.java)
- Lancer autant de client que l'on souhaite (Client.java)
- Utiliser le CLI pour interagir avec le cabinet médical. (La choix 'ajouter beaucoup d'animaux' permet de voir le callback en action).