UNIVERSITÉ DE BOURGOGNE
UFR Sciences et Techniques

Cours: BD et Environnement Distribuées

TP 1 - Illustration des concepts objets distribués avec Java-RMI

Préparé par:

MATTA Elie et al.

Table des matières

Exercices 2: Programme de simulation de gestion de compte bancaires

On va développer dans cet exercice, une application qui gère la manipulation d'un compte bancaire avec plusieurs clients en même temps.

Ainsi, on a diviser notre architecture en de la façon suivante:

Interfaces:

ICompteFonctions qui définit les fonctions que le client peut utiliser.

Classes:

1. Client, qui consiste a créer un instance a l'interface et de manipuler le compte courant.
2. Serveur, qui établie la connexion pour que le client puisse se connecté.

Il est utile de mentionner les package les plus utiliser dans cet exercice:

- java.rmi : pour accéder a des objets distants
- java.rmi.server : pour créer des objets distants
- java.rmi.registry : lié a la localisation et au nommage des objets distants

On peut mentionner les classes utiles fournies par java.rmi [(1)]:

- Naming : sert de représentant local du serveur de noms. Permet d'utiliser les méthodes bind(), rebind(), lookup(), unbind(), list()
- LocateRegistry : permet de localiser un serveur de noms (rmiregistry) et éventuellement d'en créer un.

- 1ère Question : Problème de concurrence

Apres avoir exécuter le programme, on a essayer de créditer le solde de 100 en utilisant le client 1, puis créditer de nouveau le solde de 50 en utilisant le client 2. On a obtenu le résultat montré ci-dessous.

```
Serveur lancé
Une somme de 100.0 a été credité
Une somme de 50.0 a été credité
Le solde courant est: 50.0
```

Figure 1 - Crediter le solde

Donc on remarque que le solde courant est de 50, ce qui est faux car notre programme n'a pas pris en compte la cohérence des données.

Alors le solde étant accédé simultanément par plusieurs client en même temps pose un problème de synchronisation car Java RMI ne prend pas initialement charge de la concurrence.

Une solution pour ce problème est d'ajouter "*synchronized*" pour permettre au client de manipuler l'état de l'objet créer par le serveur[(2)], alors l'objet sera verrouillé avant qu'un autre client puisse changer cet objet.

Alors on ajoute "synchronized" au fonction créditer, débiter et afficher comme montré ci-dessous, le reste du code est inclus dans la **section 1** de l'annexe:

```
public synchronized void crediter(double newSolde) throws
java.rmi.RemoteException {

      solde = solde + newSolde;

      System.out.println("Une somme de " +newSolde+ "a été credité"); }
```

Figure 2 - Fonction crediter avec "synchronized"

- 2ème Question : "ObjectFactory"

Pour réaliser une application qui gère plusieurs instances de compte en même temps, on doit trouver une solution pour la duplication pour un même objet.

Une solution pour ce problème est d'utiliser le paterne de développement "ObjectFactory" qui nous permet de créer des instances d'une classe sur le serveur.

Pour implémenter l'ObjectFactory, on doit au début créer la classe avec l'interface qui doit nous fabriqué cet objet :

A) Classe Interface ICompte

```
import java.rmi.*;

public interface ICompte extends Remote{

    public ICompteFonctions newCompte(double solde) throws RemoteException
;
```

Figure 3 - Interface ICompte

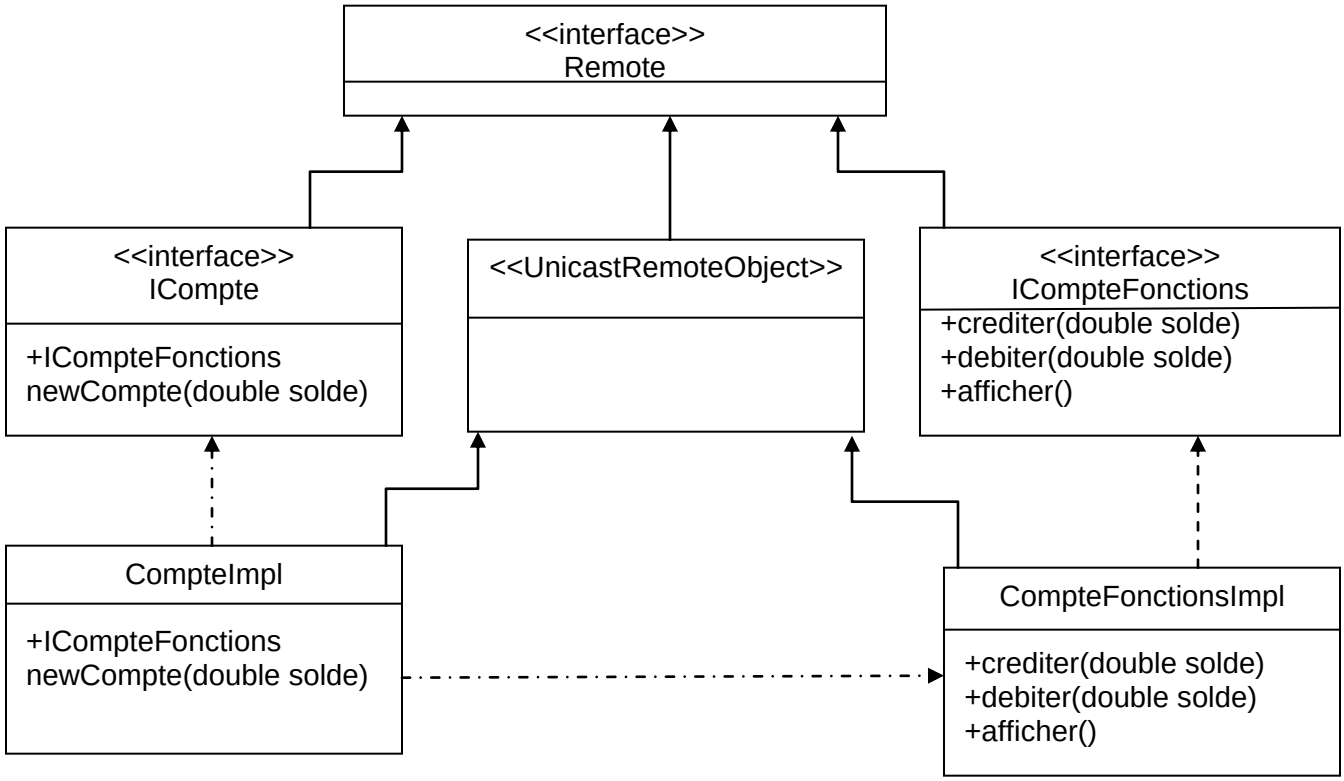
B) Class CompteImpl

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

public class CompteImpl  extends UnicastRemoteObject implements ICompte  {
      private double solde;
            public CompteImpl() throws RemoteException{     }

            public ICompteFonctions newCompte(solde) throws RemoteException{
                  return new CompteFonctionsImpl(solde);     } }
```

Figure 4 - Class CompteIml

C'est utile de mentionner que l'utilisation de "new" n'est pas efficace chez l'utilisateur car elle crée l'instance de la classe dans la mémoire locale du client.
C'est pourquoi on a explicitement créer une interface contenant la fonction newCompte et l'implémenter dans la classe CompteImpl pour l'avoir retourner une nouvelle instance (qui sera crée sur le serveur) de la class CompteFonctionsImpl qui contient les fonctions principales que le client peut utiliser, d'où l'architecture suivante:

C) Architecture de l'implémentation de l'ObjectFactory



Les classes distantes CompteImpl et CompteFonctionImpl implementent les interfaces ICompte et ICompteFonctions qui étendent l'interface "Remote" distante.

Les classes distantes CompteImpl et CompteFonctionsImpl doivent étendre la classe UnicastRemoteObject.

Par suite il nous reste de lancer le Serveur pour faire connecter le client:

D) Class Serveur

```
import java.rmi.*;

public class Serveur {
    public static void main (String [ ] argv) {
System.setSecurityManager (new RMISecurityManager ()) ; //Initialiser
SecurityManager
    try {
        String objet = "tpexdeux";
        CompteImpl ci= new CompteImpl();
        Naming.rebind (objet, ci) ;
        System.out.println ("Serveur lancé") ;
    }
    catch (Exception e) {
    System.out.println ("Erreur du serveur : " + e) ; } } }
```

Figure 4 - Class Serveur

E) Class Client

```
import java.rmi.*;
public class Client {
    public static void main (String [ ] argv) {
   System.setSecurityManager (new RMISecurityManager ()); //Initialiser
SecurityManager
    try {
ICompte InterfaceCompte =(ICompte)
Naming.lookup("rmi://localhost/tpexdeux");
        client1 = InterfaceCompte.newCompte(100);
        client2= InterfaceCompte.newCompte(200);
        client1.crediter(100); //Ajout de 100 au compte
        client2.debiter(50); //Retrait 50 du compte
        client1.afficher();
 } catch (Exception e) {
    System.out.println ("Erreur client : " + e) ; } } }
```

Figure 4 - Class Client

Interfaces:

1. ICompte: Définit la fonction newCompte.
2. ICompteFonctions : Définit les fonctions créditer, débiter et afficher.

Classes:

1. Client : Créer des instances des comptes pour les gérer.
2. Serveur : Créer l'objet et l'enregistré dans RMIRegistry.
3. CompteFonctionsImpl : Implémente l'interface ICompteFonctions et ainsi redéfinit les fonctions que le client peut utiliser.
3. CompteImpl : Implemente l'interface ICompte et redéfinit la fonction principal newCompte.

Le reste du code en mentionner dans la **section 2** du l'annexe.

En implémentant notre code on peut ainsi obtenir le workflow suivant[(1)]:

F) Workflow de l'implementation de l'ObjectFactory

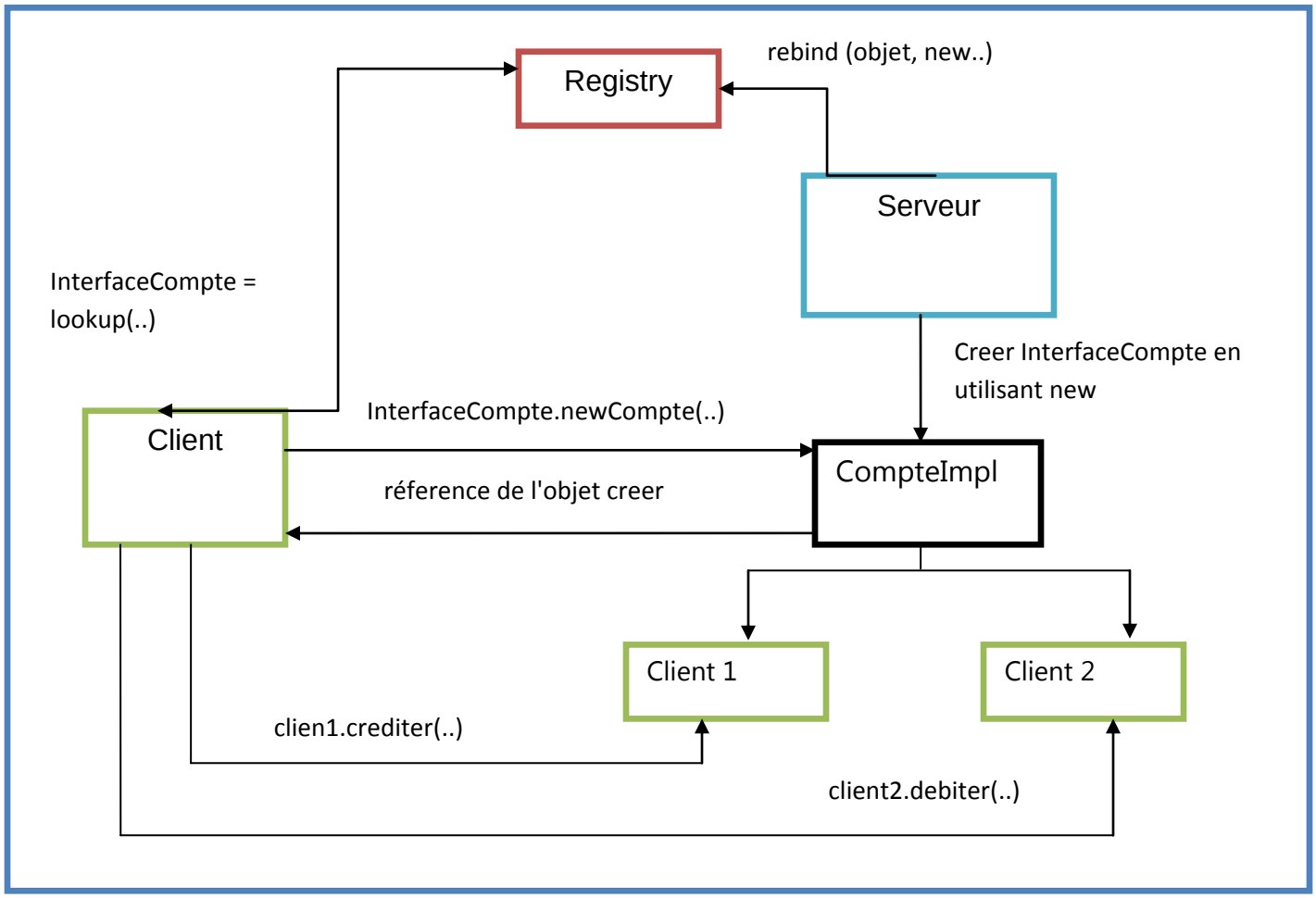


Figure 4 - WorkFlow de l'implementation de l'ObjectFactory

Donc le Client demande de creer un nouveau compte, alors une instance d'un nouveau objet sera creer et sera enregistré dans le rmiregistry, puis le client recevera la reponse par la methode de lookup.

- 3ème Question : Serveur arrêté

Lors de l'arrêt du serveur, tous les instances des objets créer seront détruites alors les comptes seront mis à zéro de nouveau.

Une solution pour ce problème est de lancer un trigger qui sauvegarde les changements qui sont émis par le client, et le solde courant.

- 4ème Question : Utilisation du package "activation"

Le package java.rmi.activation contient les interfaces, classes, et les exceptions qui représentent le système d'activation RMI, introduites dans Java 2.

Ce service RMI nous permet de définir des objets distants qui ne sont pas instanciés sur le serveur jusqu'à ce qu'une demande du client déclenche leur activation.
Le système d'activation prend charge de spécifier la manière dont un objet distant est activé/désactivé et comment les objets activé sont regroupés dans les JVM.

L'activation prend également en charge la persistance des références à distance, ou en d'autres termes, les références à des objets distants qui peuvent persister au-delà de la durée de vie d'un objet serveur individuel[3]. D'ou l'hiérarchie suivante:

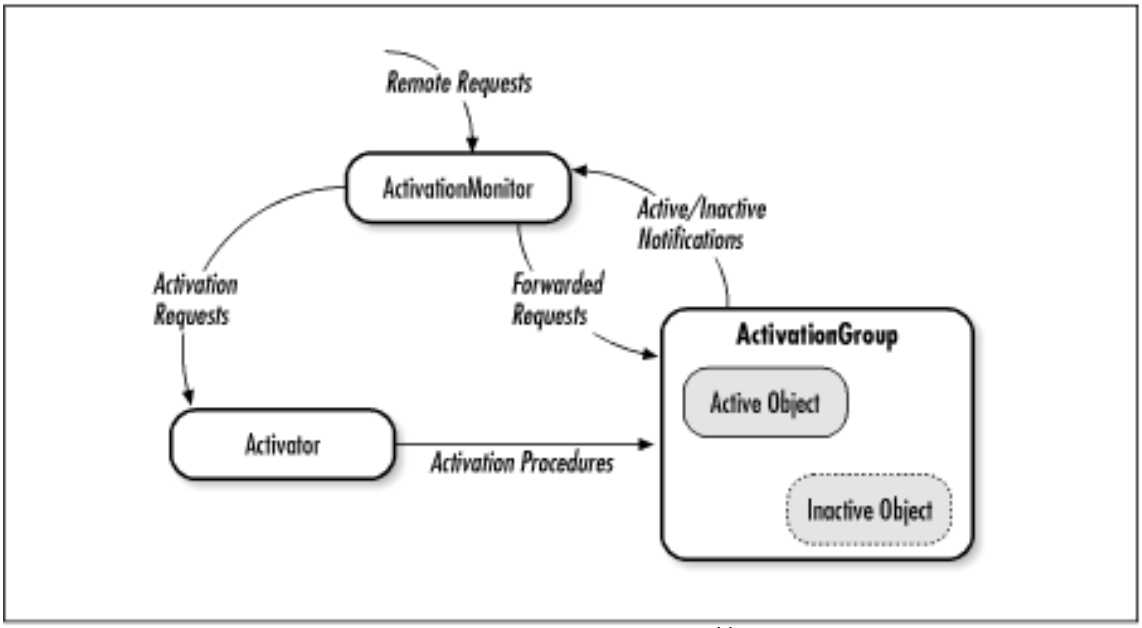


Figure 5 - Activation Workflow [4]

Depuis JDK1.2, le démon **rmid** active les objets à la demande ou au reboot de la machine.

Implementation:

A) Implementation de l'objet activable à distance

D'abord, on doit suivre les etapes suivantes pour l'utilisation du package java.rmi.actionation[5]:

- La classe d'un objet activable doit dériver de Activatable.
- La classe doit déclarer un constructeur à 2 arguments :
  – java.rmi.activation.ActivationId : C'est un identificateur pour les objets activable a distances. Lorsqu'une application enregistre un descripteur d'activation avec rmid, rmid lui attribue un ID d'activation, qui se réfère à l'information associée avec le descripteur. Cet ID d'activation même (également contenus dans stub de l'objet distant) est passée au constructeur lors de l'activation de l'objet distant
  – java.rmi.MarshalledObject : contient les données d'initialisation qui sont déjà enregistrés avec des rmid. Notre programme ne nécessite pas de données d'initialisation pour construire l'objet distant.

Alors on crée une nouvelle classe "ICompteActivatableImpl" qui dérive de la classe "Activatable" et qui implémente l'interface ICompteFonctions.

Puis on a créer u constructeur qui prend deux arguments, cette classe va nous aider à déclarer une nouvelle instance durant le processus de l'activation, elle vas appeler un constructeur de la superclasse (activatable) pour exporter l'objet, d'où la figure ci-dessous:

```
import java.rmi.*;
import java.rmi.activation.*;

public class ICompteActivatableImpl extends Activatable implements
ICompteFonctions {
        public ICompteActivatableImpl(ActivationID id, MarshalledObject mo)
throws java.rmi.RemoteException {
                      super(id, 0);
                       double solde;
                        System.out.println("ICompteActivatableImpl est
activé comme ayant l'id = "+id);
    } }
```

Figure 6 - Class ICompteActivatableImpl

B) Implémentation du programme d'enregistrement (Setup)

Le rôle du setup c'est de passer l'information nécessaire à l'activation de l'objet activable au démon rmid (qui active une JVM par groupe) puis enregistrer l'objet auprès de rmiregistry. Cette classe va aussi enregistrer un descripteur "activation descriptor" avec rmid pour permettre aux objet suivantes d'être activés en utilisant ce dernier.

- Après que rmid a activé une JVM, on aura intérêt a utilisé les objets suivants: