Programmation Client/Serveur
Java & RMI

Dernière minute...

Une petite remarque concernant la politique de sécurité en Java pour le lancement des applications serveur et cliente.

Vous remarquerez que l'une des premières lignes exécutées dans la méthode `main` des ces applications est `setSecurityManager(...)`. Cette instruction impose à la JVM d'installer un gestionnaire de sécurité pour limiter le champ d'action de vos applications. Du coup, pour qu'elles puissent s'exécuter "normalement", il est nécessaire d'indiquer à ce gestionnaire de sécurité que vos applications serveur et cliente ont la permission d'établir des connexions réseaux pour dialoguer. C'est quand même l'objectif de ce TP...;-) D'où la nécessité du fichier `java.policy` (page 7) et de l'option `-Djava.security.policy=java.policy` de la commande `java`.

Si vous ne souhaitez pas être confrontés aux problèmes de permissions inhérents à la mise en place d'un gestionnaire de sécurité en Java, il vous suffit simplement de mettre en commentaire (ou retirer) l'instruction `setSecurityManager(...)` dans la méthode `main` de vos applications serveur et cliente. Peut-être aurez-vous l'occasion de revenir un jour sur ces problèmes de permissions en licence ASUR...

Programmation Client/Serveur
Java & RMI

Résumé

L'objectif de ce premier TP est d'assurer la transition entre la programmation d'applications monoposte (un programme s'exécutant sur une machine) et la programmation d'applications distribuées (plusieurs programmes s'exécutant sur des machines différentes et interconnectés via le réseau). Ce TP vous permettra de vous familiariser avec le modèle client/serveur puis de le mettre en pratique sans vous soucier des problèmes inhérents à la couche réseau (programmation des sockets notamment). Nous utiliserons pour cela les RMI (*Remote Method Invocation*) du langage Java[1].

1 Compléments de Cours

En POO et en Java vous avez déjà vu de quelle manière un objet (**client**) pouvait invoquer une méthode sur un autre objet (**serveur**). Il s'agit de la version la plus simple du modèle **client/serveur** où les deux objets s'exécutent sur la même machine. Le mécanisme d'invocation de méthodes à distance de Java (les Java RMI) permet à un programme s'exécutant sur un ordinateur (client) d'effectuer des appels de méthodes sur un objet se trouvant sur un ordinateur distant (serveur). Les RMI offrent ainsi aux programmeurs Java la possibilité de distribuer les tâches exécutées par leurs programmes dans un environnement réseau. En conception orientée objet, l'objectif est que chaque tâche soit exécutée par l'**objet** le plus approprié pour cet tâche. Les RMI vont encore plus loin dans cette démarche en permettant l'exécution d'une tâche sur l'**ordinateur** le plus approprié à cette tâche.

L'architecture RMI définit un ensemble d'interfaces spécifiques destinées à créer et manipuler des objets distants. Un client peut ainsi invoquer des méthodes sur un objet distant en utilisant la même syntaxe que pour l'invocation de méthodes sur un objet local. L'API des RMI fournit les classes et les méthodes qui vont gérer tout ce qui concerne la couche communication sous-jacente et les références sur les paramètres lors des invocations de méthodes à distance. Les RMI s'occupent également de la sérialisation des objets passés en arguments des méthodes sur les objets distants.

1.1 L'architecture RMI

Les paquettages `java.rmi` et `java.rmi.server` contiennent les interfaces et les classes qui implémentent toutes ces fonctionnalités de l'architecture RMI. Ils contiennent les briques de base nécessaires pour développer les objets du côté serveur et les objets *stubs* du côté client. Un stub[2] est une représentation locale de l'objet distant. Le client effectue ses invocations de méthodes sur ce stub (objet local), lequel s'occupe automatiquement de communiquer avec la partie serveur. Ceci se fait de manière totalement transparente pour le client. Du côté serveur, c'est le *skeleton*[3] qui reçoit les appels effectués par

[1] Les RMI (*Remote Method Invocation*) sont similaires aux RPC (*Remote Procedure Call*) introduits par SUN en 1985. Les RPC nécessitaient toutefois de sérialiser manuellement les paramètres et les valeurs de retour des méthodes. SUN avait d'ailleurs développé une bibliothèque à cet effet : XDR (*eXternal Data Representation*). Une des principales différences entre les RMI et les RPC est que les RMI se basent sur le mécanisme de sérialisation offert par le langage Java. Les RMI permettent également de sérialiser les exceptions pouvant être levées par une méthode distante pour les transmettre aux programmes clients.

[2] La traduction française de "stub" est "talon", ce qui est moins parlant. Du coup, vous rencontrerez la plupart du temp le mot stub, même en français.

[3] "Skeleton" peut se traduire par "squelette" en français, si cela vous tente...

des appels distants et qui les transmet au serveur comme s'il s'agissait d'une invocation locale. Là encore, que l'invocation soit locale ou distante est totalement transparent pour le serveur. Le principe de fonctionnement peut se résumer ainsi (cf. figure 1) :

- L'objet client ne communique pas directement avec l'objet serveur, mais avec un proxy local : le **stub**. Celui-ci est en charge de :
  - la sérialisation des objets passés en paramètres (*marshaling*)
  - la construction de la requête réseau
  - l'attente du résultats et/ou des exceptions
- De son côté, l'objet serveur est relié au réseau via un skeleton dont le rôle est de :
  - désérialiser les paramètres (*unmarshaling*)
  - d'invoquer la méthode sur l'objet serveur
  - de sérialiser le résultats et/ou les exceptions et de les renvoyer au client via le stub

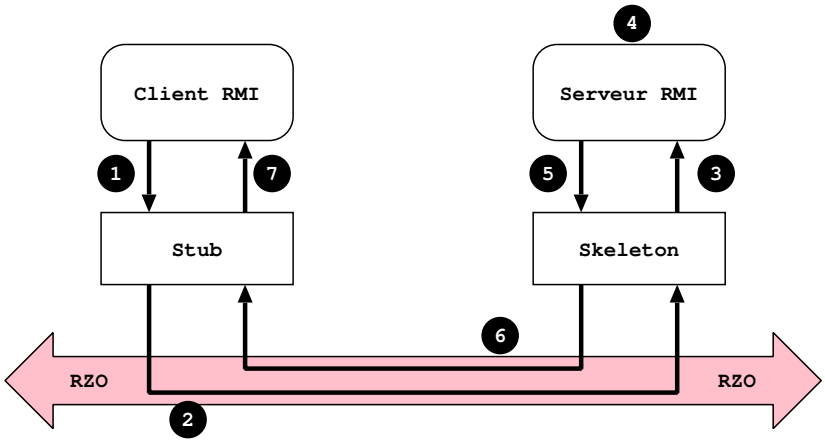

FIG. 1 – Architecture RMI

Comme cela est illustré par la figure 1, lorsque qu'un objet client invoque une méthode sur un objet serveur distant, cette invocation de méthode distante se déroule en plusieurs étapes :

1. l'objet client invoque (normalement) la méthode sur le stub
2. le stub sérialise les objets passés en paramètres de l'appel (*marshaling*) puis envoie la requête sur le réseau
3. le skeleton reçoit la requête, désérialise les paramètres (*unmarshaling*), puis invoque la "vraie" méthode sur l'objet serveur
4. l'objet serveur exécute (normalement) la méthode
5. l'objet serveur retourne le résultat et/ou les exceptions à l'objet ayant invoqué la méthode, c'est-à-dire le skeleton (et non directement à l'objet client)
6. le skeleton sérialise le résultat et/ou les exceptions, puis envoie le tout via le réseau au stub qui avait effectué la requête
7. le stub reçoit le résultat de l'invocation, désérialise les données, puis les transmet à l'objet, **comme si l'invocation avait été réalisée localement**

Le stub n'est donc qu'un proxy permettant à l'objet client d'accéder via le réseau, et ce de manière totalement transparente, aux services (méthodes) proposés par l'objet serveur. Pour construire ce stub, seule l'interface du serveur est nécessaire. C'est dans cette interface que le serveur déclare tous les services qu'il propose. Il n'est pas nécessaire de connaître son implémentation. Quand on programme en client/serveur il est donc indispensable de bien distinguer l'interface et l'implémentation du serveur.

1.2 Un exemple RMI pas à pas

Cette section va vous guider pas à pas dans la conception, la compilation et l'exécution d'un exemple d'application utilisant les RMI. Pour créer une application qui soit accessible à des clients distants, vous

devez suivre un certain nombre d'étapes :

1. Définir les interfaces des classes distantes.
2. Ecrire et compiler l'implémentation de ces classes distantes.
3. Créer les classes stub et skeleton en utilisant la commande `rmic`.
4. Ecrire et compiler l'application serveur.
5. Démarrer le RMI registry et l'application serveur.
6. Ecrire et compiler le programme client accédant aux objets distants.
7. Tester le client.

Nous allons détailler notre exemple en suivant les étapes indiquées ci-dessus. L'exemple utilisé est relativement simple. L'application serveur est chargée de gérer plusieurs comptes bancaires et de permettre à des programmes clients d'effectuer différentes opérations de base sur ces comptes. Cet exemple a été simplifié au maximum afin de se concentrer uniquement sur les RMI.

1.2.1 Interfaces des classes distantes

La première classe que nous écrivons est celle représentant un compte bancaire. Les programmes clients ne manipuleront pas directement les objets de cette classe. Ils devront le faire via la banque. Or cette dernière n'effectuera que des invocations de méthodes locales. La classe "compte bancaire" ne sera donc jamais utilisée à distance. Nous pouvons donc écrire l'interface `Compte` et la classe `CompteImpl` comme d'habitude.

Compte.java

```
import java.lang.*;

interface Compte {

    String nom();
    double solde();
    void deposer(double montant);
    void retirer(double montant);

}
```

CompteImpl.java

```
import java.lang.*;

public class CompteImpl implements Compte {

    // Attributs

    private String _nom   = null;
    private double _solde = 0.0;

    // Constructeurs

    public CompteImpl()
    {
        _nom   = "unknown";
        _solde = 0.0;
    }
```

```
    public CompteImpl(String n)
    {
        this();
        _nom = n;
    }

    // Méthodes de l'interface

    public String nom()
    {
        return _nom;
    }

    public double solde()
    {
        return _solde;
    }

    public void deposer(double montant)
    {
        _solde += montant;
    }

    public void retirer(double montant)
    {
        _solde -= montant;
    }

}
```

Quand un programme client voudra réaliser une opération sur un compte bancaire, il devra s'adresser à la banque en indiquant le compte qu'il veut manipuler. C'est donc sur cette classe "banque" que les client invoqueront, à distance, des méthodes. Lors de la définition de l'interface `Banque` nous devons donc indiquer qu'il s'agit d'une interface dont les méthodes pourront être invoquées à distance. Il suffit pour cela qu'elle étende l'interface `java.rmi.Remote` et que toutes ses méthodes signalent qu'elles peuvent éventuellement lever une exception `java.rmi.RemoteException` lors d'une invocation à distance. Ces deux conditions sont imposées par l'utilitaire `rmic` qui se chargera de créer les classes stub du côté des clients. Ces stubs implémenteront cette interface.

Banque.java

```
import java.lang.*;

interface Banque extends java.rmi.Remote {

    boolean existe(String nom) throws java.rmi.RemoteException;
    void ouvrir(String nom) throws java.rmi.RemoteException;
    double solde(String compte) throws java.rmi.RemoteException;
    void deposer(String compte, double montant) throws java.rmi.RemoteException;
    void retirer(String compte, double montant) throws java.rmi.RemoteException;
    void transferer(String compte1, String compte2, double montant)
        throws java.rmi.RemoteException;

}
```

1.2.2 Implémentation des classes distantes

Nous pouvons maintenant écrire l'implémentation de nos objets serveur. La classe `BanqueImpl` implémente tout naturellement l'interface `Banque` définie précédemment. Comme vous pouvez le remarquer, nous indiquons que la classe `BanqueImpl` étend la classe `UnicastRemoteObject`, ce qui signifie que les objets intanciés à partir de la classe `BanqueImpl` ne pourront traiter qu'une seule requête à la fois. Si plusieurs clients tentent d'accéder en même temps à un de ces objets distants, ils seront mis dans une file d'attente et ne recevront la référence à cet objet distant qu'une fois cette référence relâchée par le client précédent.

BanqueImpl.java

```
import java.lang.*;
import java.util.*;

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

// La classe UnicastRemoteObject indique que l'objet
// distant sera "unicast", i.e. qu'il ne pourra répondre
// qu'à une seule requête à la fois.

public class BanqueImpl extends UnicastRemoteObject implements Banque {

    // Attributs

    private Hashtable _comptes = null;

    // Constructeurs

    public BanqueImpl() throws java.rmi.RemoteException
    {
        _comptes = new Hashtable();
    }

    // Méthodes de l'interface

    public boolean existe(String nom) throws java.rmi.RemoteException
    {
        return ( (Compte)_comptes.get(nom) != null );
    }

    public void ouvrir(String nom) throws java.rmi.RemoteException
    {
        _comptes.put(nom,new CompteImpl(nom));
    }

    public double solde(String compte) throws java.rmi.RemoteException
    {
        return ( (Compte)_comptes.get(compte) ).solde();
    }

    public void deposer(String compte, double montant) throws java.rmi.RemoteException
    {
        ( (Compte)_comptes.get(compte) ).deposer(montant);
    }
```

```
    public void retirer(String compte, double montant) throws java.rmi.RemoteException
    {
        ( (Compte)_comptes.get(compte) ).retirer(montant);
    }

    public void transferer(String compte1, String compte2, double montant)
        throws java.rmi.RemoteException
    {
        ( (Compte)_comptes.get(compte1) ).retirer(montant);
        ( (Compte)_comptes.get(compte2) ).deposer(montant);
    }

}
```

La classe `BanqueImpl` est responsable de la création et du stockage d'objets `Compte`. Elle utilise pour cela une table de hashage dont la clé est le nom du compte.

1.2.3 Création des classes stub et skeleton

Une fois que l'imlémentation des classes distantes sont compilées, l'étape suivante est la génération des classes stub et skeleton permettant d'accéder à distance à ces distance. Les classes stub seront utilisées par le code du client pour communiquer avec le code du skeleton du serveur concerné.

La commande `rmic` permet de créer automatiquement le code des stubs et skeletons à partir de l'interface et de l'implémentation des classes des serveurs. Par exemple, pour générer le stub et le skeleton de notre classe distante `Banque`, il faut exécuter :

```
rmic BanqueImpl
```

Cette commande `rmic` crée dans le répertoire deux nouveaux fichiers de classes, `BanqueImpl_Skel.class` et `BanqueImpl_Stub.class`, correspondant respectivement au skeleton et au stub de la classe `BanqueImpl`. Les stubs et les skeletons étant créés, l'étape suivante consiste à écrire l'application serveur mettant ces classes à disposition des clients désirant y invoquer des méthodes à distance.

1.2.4 Création et compilation de l'application serveur

Tout est maintenant en place pour le développement de l'application côté serveur. Nous allons pour cela créer une classe `Serveur` dont le seul travail sera d'instancier un objet à partir de la classe `BanqueImpl` puis de le publier dans le RMI registry afin qu'il soit accessible par les clients.

Par défaut, les applications s'exécutent sans security manager. L'appel `setSecurityManager` impose l'utilisation d'un RMI security manager. Nous n'irons pas plus loin sur cette notion. Sachez simplement que ce mécanisme permet de sécuriser les connexions client/serveur en vérifiant notamment que les interfaces utilisées par le client sont bien celles publiées par le serveur. Ce mécanisme permet également l'envoi de ces interfaces à l'application cliente si nécessaire.

Le serveur publie ensuite l'objet instancié en l'enregistrant dans le RMI registry sous un certain nom. Nous disposons pour cela de deux méthodes : `bind` et `rebind`. Ces deux méthodes fonctionnent de manière identique, sauf quand le nom est déjà lié à un autre objet. Dans ce cas, la méthode `bind` lève une exception `AlreadyBoundException` alors que la méthode `rebind` déréférence l'ancien objet et force le lien vers le nouvel objet. Dans les deux cas, le nom utilisé est une chaîne de caractères de la forme d'une URL, c'est-à-dire `protocol ://host :port/bindingName`. Dans le cas présent, `protocol` correspond à `rmi`, `host` est le nom de la machine sur laquelle s'exécute le serveur RMI, `port` est le numéro de port sur lequel le serveur est à l'écoute, et `bindingName` correspond au nom exact qui sera utilisé par les clients pour obtenir l'accès au serveur. Si seul le `bindingName` est fourni, les valeurs par défaut sont utilisées pour les autres paramètres, à savoir : `rmi` pour `protocol`, `localhost` pour `host` et 1099 pour `port`.

Serveur.java

```
import java.rmi.*;

public class Serveur {

    public static void main(String args[])
    {
        // Création et installation d'un "security manager".
        System.setSecurityManager(new RMISecurityManager());

        try {
            // Création d'une instance de notre classe Banque
            Banque bank = new BanqueImpl();
            System.out.println("Banque créée");

            // "Bind" de cette instance dans le RMI registry
            Naming.rebind("banqueGTR",bank);
            System.out.println("Banque enregistrée dans le RMI registry");

            System.out.println("Le serveur est prêt...");
        }
        catch (Exception e)
            {
                System.out.println("Il y a eu une erreur !!!");
                e.printStackTrace();
                System.out.println(e.getMessage());
            }
    }

}
```

1.2.5 Lancement du RMI registry et de l'application serveur

Le RMI registry est un programme qui permet de faire la correspondance entre un nom symbolique et une référence vers un objet distant (un objet serveur). Comme nous l'avons vu précédemment, l'enregistrement d'un nouvel objet se fait via les méthodes `bind` et `rebind`. Les applications clientes pourront alors se connecter au RMI registry et y faire un lookup qui, à partir d'un nom symbolique, instanciera un stub et un skeleton pour le serveur indiqué, les mettra en contact, puis retournera au client une référence vers le stub.

Les deux lignes de commandes suivantes permettent d'intaller le RMI registry puis de démarrer le serveur bancaire. Pour des raisons de simplification, nous nous abstiendrons de télécharger les interfaces via HTTP. Du coup, il est nécessaire d'exécuter le RMI registry dans le répertoire où se trouvent toutes les classes que vous avez écrites, de manière à ce qu'il puisse y avoir accès.

```
rmiregistry
java -Djava.security.policy=java.policy Serveur
```

Comme mentionné précédemment, nous ne nous occupons pas pour le moment des problèmes de sécurité entre le serveur et le client. C'est pour cette raison que la politique de sécurité uilisée est extrêmenent simple et permissive puisqu'elle autorise toutes les connexions aux ports situés entre `1024` et `65535`, ainsi qu'au port `80` (protocole HTTP). Cette politique est enregistrée dans le fichier `java.policy`.

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

L'application serveur affiche plusieurs messages lors de son exécution. S'il n'y a aucune erreur, vous devriez obtenir ceci :

```
Banque créée
Banque enregistrée dans le RMI registry
Le serveur est prêt...
```

A noter que vous pouvez démarrer le `rmiregistry` sur un port différent, ce qui est utile si plusieurs `rmiregistry` doivent tourner simultanément sur la même machine. Il suffit pour cela d'indiquer en paramètre le numéro du port (exemple : `rmiregistry 2048`). Ensuite, il vous faudra bien évidemment signaler aux programmes serveurs et clients qu'ils doivent utiliser ce numéro de port pour contacter le `rmiregistry` et non le port par défaut (`1099`).

1.2.6 Création et compilation du programme client

Lorsque l'application cliente démarre, elle doit trouver un objet `Banque` sur le serveur distant. Ce programme suppose que le nom de la machine sur laquelle s'exécute l'application serveur est passé en premier argument sur la ligne de commandes. Ce nom est utilisé pour créer une URL de la forme `rmi ://<hostname>/banqueGTR`. Cette URL est passée à la méthode `lookup` invoquée statiquement sur la classe `Naming`. Comme indiqué précédemment, cette méthode renvoie au programme client une référence vers le stub qui se chargera de communiquer avec l'objet serveur. A noter que le type de retour de la méthode `lookup` est `Remote` qui est l'interface mère de toutes les interfaces stubs.

Le programme client utilise le second argument de sa ligne de commandes comme nom de compte bancaire. Il contacte alors le serveur bancaire pour effectuer plusieurs opérations sur ce compte.

Client.java

```
import java.rmi.*;

public class Client {

    public static void main(String args[])
    {
        Banque bank = null;

        // Analyse des paramètres de la ligne de commande
        if (args.length < 2)
            {
                System.err.println("Usage:");
                System.err.println("java Client <serveur> <nom compte>");
                System.exit(1);
            }

        // Création et installation d'un "security manager".
        System.setSecurityManager(new RMISecurityManager());

        // Obtention d'une référence de la banque (le serveur)
        try {
            String url = new String("//"+args[0]+"/banqueGTR");
            System.out.println("Client: lookup serveur, url = "+url);
            bank = (Banque)Naming.lookup(url);
        }
        catch (Exception e)
            {
                System.out.println("Erreur dans l'obtention du serveur"+e);
            }
```

```
        // On effectue quelques transactions bancaires
        try {
            // Si le compte n'existe pas, on l'ouvre
            if (!bank.existe(args[1]))
                {
                    bank.ouvrir(args[1]);
                    System.out.println("Compte \""+args[1]+"\" créé");
                }

            System.out.println("Le solde du compte \""+args[1]+"\" est de "+
                               bank.solde(args[1])+" FF");

            bank.deposer(args[1],5000);
            System.out.println("Ajout de 5000 FF sur le compte \""+args[1]+"\"");
            System.out.println("Le solde du compte \""+args[1]+"\" est de "+
                               bank.solde(args[1])+" FF");

            bank.retirer(args[1],400);
            System.out.println("Retrait de 400 FF depuis le compte \""+args[1]+"\"");
            System.out.println("Le solde du compte \""+args[1]+"\" est de "+
                               bank.solde(args[1])+" FF");
        }
        catch (Exception e)
            {
                System.out.println("Erreur de transaction pour "+args[1]);
            }

        System.exit(0);
    }

}
```

1.2.7 Test du programme client

La dernière étape de notre exemple consiste à tester le bon fonctionnement de notre programme client (et donc également de notre application serveur). Ce programme peut être exécuté depuis n'importe quelle machine ayant accès au serveur et aux classes nécessaires (nous utilisons pour le moment un RMI security manager minimal). Voici un exemple d'exécution de ce programme avec le serveur distant tournant sur la machine `urizen` (la première ligne correspond à la commande exécutée) :

```
munier@chryseis:remote > java -Djava.security.policy=java.policy Client urizen munier
Client: lookup serveur, url = //urizen/banqueGTR
Compte "munier" créé
Le solde du compte "munier" est de 0.0 FF
Ajout de 5000 FF sur le compte "munier"
Le solde du compte "munier" est de 5000.0 FF
Retrait de 400 FF depuis le compte "munier"
Le solde du compte "munier" est de 4600.0 FF
```

Une fois que le programme client a terminé son exécution, les objets distants existent toujours sur le serveur. L'exécution ci-dessus a créé le compte bancaire pour l'utilisateur `munier`. Si nous exécutons à nouveau ce programme avec les mêmes paramètres, le programme va utiliser le compte existant. La trace ci-dessous montre bien que les opérations ont été exécutées sur le compte dans l'état où il était après la première exécution du programme client.

```
munier@chryseis:remote > java -Djava.security.policy=java.policy Client urizen munier
Client: lookup serveur, url = //urizen/banqueGTR
Le solde du compte "munier" est de 4600.0 FF
Ajout de 5000 FF sur le compte "munier"
Le solde du compte "munier" est de 9600.0 FF
Retrait de 400 FF depuis le compte "munier"
Le solde du compte "munier" est de 9200.0 FF
```

1.3 Sérialisation des paramètres

Dans l'exemple que nous venons de développer, les paramètres transmis lors des invocations de méthodes à distance sont de types relativement simples : `double`, `String`,... Imaginez maintenant que vous vouliez transmettre des paramètres plus compliqués tels que des objets instances de classes que vous avez définies. Prenons par exemples les classes représentant des expressions numériques (cf. figure 2).

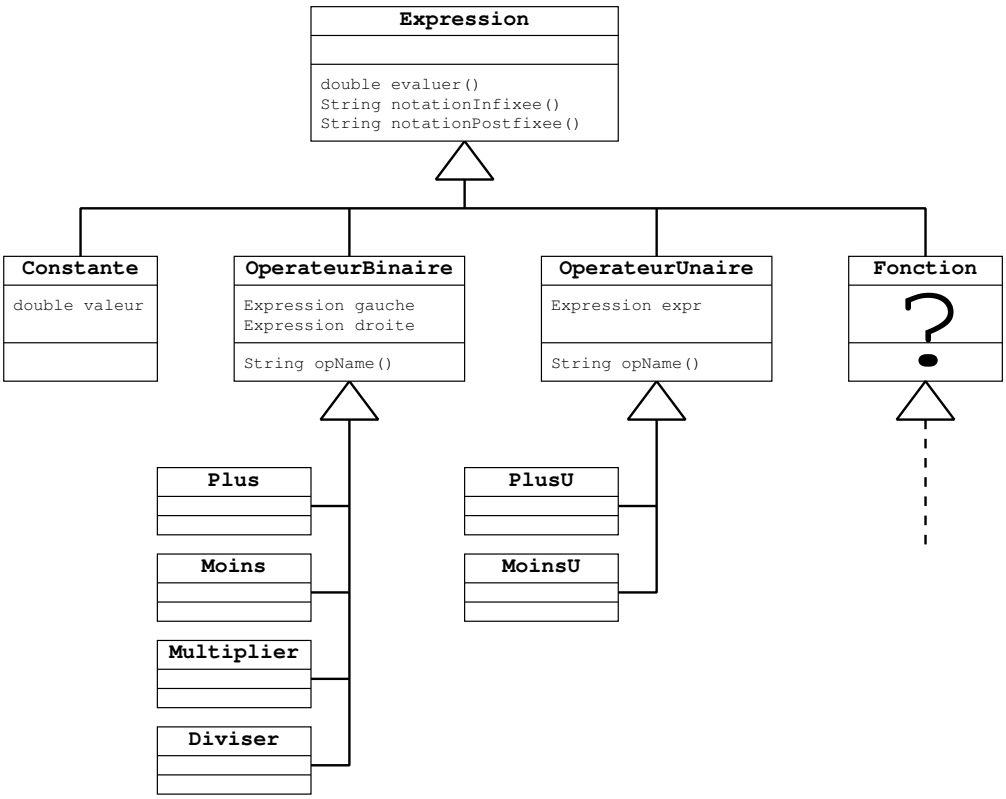

FIG. 2 – Graphe de classes des expressions numériques

Le problème est que pour pouvoir passer de tels objets complexes en paramètres (nous parlons ici d'arbres binaires dont les nœuds peuvent être de classes différentes), il est nécessaire d'être capable de les sérialiser de manière à les découper en paquets au niveau de la couche réseau, puis de reconstruire ces paramètres sur la machine distante ! Essayez de le faire manuellement pour voir... et vous comprendrez que ce n'est pas si simple `;-)`

Heureusement, Java propose en standard un mécanisme de sérialisation automatique des classes. Il vous suffit simplement d'indiquer que l'interface mère de vos paramètres étend l'interface `java.io.Serializable` et le tour est joué. Par exemple : `interface Expression extends java.io.Serializable {...}` Vous pouvez bien évidemment surcharger les méthodes `writeObject` et `readObject` utilisées lors de la sérialisation, mais celles disponibles en standard sont très satisfaisantes.

2 Exercices

Il est maintenant temps pour vous de mettre toutes ces notions en pratique. Voici donc deux exercices (distributeur de tickets, serveur DNS) qui, d'un point de vue algorithmique sont relativement simples, mais qui vous permettront justement de n'être confrontés qu'à des problèmes d'invocation de méthodes sur des objets distants (par rapport aux invocations de méthodes sur des objets locaux qui n'ont quasiment plus aucun secret pour vous).

2.1 Distributeur de tickets

Pour ce premier exercice, il serait difficile de faire plus simple. Il s'agit de réaliser un serveur ne disposant que d'une seule méthode : `int prendreTicket()`. C'est le type de "serveur" que l'on trouve dans certains supermarchés quand vous devez faire la queue pour être servi(e) : il donne un ticket différent à chaque fois qu'on le sollicite. Ce serveur sera initialisé lors de sa mise en service, puis distribuera le ticket n°1 au premier client, le ticket n°2 au suivant, et ainsi de suite.

Outre l'interface `Distributeur` et la classe `DistributeurImpl`, vous devrez également écrire un programme permettant de créer un nouveau serveur et de l'enregistrer dans le RMI registry ainsi qu'un programme client qui se connectera à ce serveur pour prendre un ticket. Il vous faudra bien évidemment vérifier que vous obtenez un numéro de ticket différent à chaque appel.

2.2 Serveur DNS

Dans cet exercice, on se propose pour cela de réaliser l'interface puis l'implémentation d'un serveur DNS (notées respectivement `DNS` et `DNSImpl`). Ce serveur permettra de faire la correspondance entre un nom de machine (une `String`) et une adresse IP (pour le moment, une `String` également). Il propose à cet effet deux services :

- La méthode `void enregistrer(String, String)` permet au client d'ajouter une nouvelle association nom de machine/adresse IP dans le DNS. Le premier paramètre est l'adresse IP, le second est le nom de la machine.
- La méthode `String rechercher(String)` recherche l'adresse IP d'une machine à partir de son nom.

2.2.1 Objets locaux

Cette première partie est facultative. Elle consiste à écrire l'interface `DNS`, la classe `DNSImpl` ainsi qu'un programme de test en considérant que tous les objets s'exécutent sur la même machine. Cette première étape correspond à la programmation orientée objets classique.

2.2.2 Objets distants

Dans cette seconde partie vous allez tester vos (nouvelles) compétences en programmation orientée objets distribués. Il s'agit de modifier les interfaces et les classes écrites dans la première étape de manière à ce que le serveur DNS puisse être utilisé par des programmes clients s'exécutant sur des machines différentes. En d'autres termes, il vous est demandé de mettre en œuvre et de tester les mécanismes RMI pour les méthodes `enregistrer` et `rechercher` du serveur DNS.

2.2.3 Objets distants & sérialisation

Pour conclure cet exercice sur les RMI, nous voulons changer la représentation des adresses IP. Au lieu de stocker sous forme de chaînes de caractères nous allons définir une interface `AdresseIP` et une classe `AdresseIPImpl` permettant de représenter une adresse IP sous la forme d'un tableau de quatre entiers, avec bien évidemment les méthodes adéquates. Les signatures des méthodes du DNS doivent également être modifiées :

- `void enregistrer(AdresseIP, String)`
- `AdresseIP rechercher(String)`

2.3 Forum de discussion

Maintenant que vous avez bien compris le mécanisme des RMI, nous allons passer à un exercice où les invocations de méthodes à distance entre le "client" et le "serveur" s'enchaînent les unes après les autres. Vous allez pour cela développer un serveur *chat* auquel pourront se connecter plusieurs clients pour discuter. Le principe de fonctionnement est le suivant : une fois le serveur démarré, un client peut prendre part à la discussion en s'enregistrant sur le serveur, ceci afin de signaler sa présence. Ensuite, lorsqu'un client veut émettre un message, il l'envoie au serveur. A réception de ce message, le serveur se charge de le retransmettre à tous les clients qui se sont enregistrés, qui s'occupent alors de l'afficher sur la console de l'utilisateur. Afin que vous puissiez utiliser votre programme client avec le programme serveur d'un autre binôme, le mieux est que vous vous basiez tous sur les mêmes interfaces (et ce au caractère prêt, y compris les majuscules/minuscules !).

ChatRoom.java

```
public interface ChatRoom extends java.rmi.Remote {
    void enregistrerClient(ChatClient client) throws java.rmi.RemoteException;
    void envoyerMessage(Message msg) throws java.rmi.RemoteException;
}
```

ChatClient.java

```
import java.lang.*;

public interface ChatClient extends java.rmi.Remote {
    String nom() throws java.rmi.RemoteException;
    void afficherMessage(Message msg) throws java.rmi.RemoteException;
}
```

Message.java

```
import java.lang.*;

public interface Message extends java.io.Serializable {
    String origine();
    String toString();
}
```

Parlons un peu plus en détails de la manière dont tout ceci va fonctionner. En effet, comme vous allez le constater rapidement, il n'existe plus dans cet exercice de "client" ou de "serveur". Les objets que nous allons utiliser vont être tour à tour client ou serveur ;-)

1. On démarre, comme d'habitude, le RMI registry.
2. On lance l'application serveur qui instancie un objet `ChatRoom`[4] puis l'enregistre, toujours comme d'habitude, dans le RMI registry.
3. On peut alors démarrer une application cliente sur une autre machine. Celle-ci se connecte au RMI registry pour récupérer une référence vers l'objet `ChatRoom`. Jusque là, il n'y a rien de nouveau par rapport aux exercices précédents.
4. C'est maintenant que ça change un peu. L'application cliente peut alors instancier un objet `ChatClient`. Elle effectue ensuite une RMI pour invoquer la méthode `enregistrerClient` sur l'objet `ChatRoom` en lui passant comme argument la référence vers l'objet `ChatClient` qu'elle vient de créer.
5. L'objet `ChatRoom` stocke cette référence vers l'objet `ChatClient` dans un de ses attributs (un `Vector` par exemple).
6. L'application cliente entre alors dans un boucle qui lit une chaîne de caractères au clavier, construit un objet `Message` en précisant son nom comme origine, puis envoie ce message à l'objet `ChatRoom` via une RMI sur la méthode `envoyerMessage`.

[4] Ou plus exactement un objet issu d'une classe implémentant l'interface `ChatRoom`.

7. Quand la méthode `envoyerMessage` est exécutée par l'objet `ChatRoom`, elle parcourt la liste des objets `ChatClient` enregistrés et, pour chacun d'entre eux, effectue à son tour une RMI sur leur méthode `afficherMessage` en utilisant tout simplement la référence `ChatClient` stockée. Il lui est inutile d'interroger le RMI registry pour faire un lookup puisque l'objet `ChatRoom` possède déjà une référence vers l'objet `ChatClient` qu'il veut atteindre (cf. méthode `enregistrerClient`).
8. Chaque objet `ChatClient` exécutera donc sa méthode `afficherMessage` qui aura pour effet d'afficher l'origine et le contenu du message sur l'écran de la machine sur laquelle tourne l'application cliente correspondante.

Dans ce scénario, vous pouvez effectivement constater qu'à l'étape 6, c'est l'objet `ChatClient` qui effectue une RMI vers l'objet `ChatRoom` pour invoquer sa méthode `envoyerMessage`. L'objet `ChatClient` joue le rôle de client et l'objet `ChatRoom` joue le rôle de serveur. Par contre, à l'étape 7, c'est l'objet `ChatRoom` qui effectue une RMI vers l'objet `ChatClient` pour invoquer sa méthode `afficherMessage`. Dans ce cas, l'objet `ChatRoom` joue le rôle de client et l'objet `ChatClient` joue le rôle de serveur !

Une conséquence de ceci set qu'il vous sera nécessaire de générer les stubs et les skeletons non seulement pour les classes implémentant l'interface `ChatRoom`, mais également pour celles implémentant l'interface `ChatClient`, de manière à ce que l'objets `ChatRoom` puisse invoquer la méthode `afficherMessage` sur des objets `ChatClient` distants.