Remote Method Invocation

Invocation de méthodes distantes (RMI)

Stéphane NICOLAS

Hiver 2002

Résumé

Ce document est destiné à donner au lecteur une connaissance à la fois pratique et théorique du modèle de communication de l'API RMI. Il permet de s'initier à l'informatique distribuée à travers la mise en oeuvre d'un système général de calcul distribué généraliste et à chargement dynamique de tâches (cet exemple est une adaptation libre d'un tutoriel de SUN [13]).

1 Introduction

L'API Remote Method Invocation (RMI) permet la communication entre des objets java exécutés dans des Java Virtual Machine (JVM) différentes ; ces JVMs pouvant être situées sur des machines distantes l'une de l'autre. Cette API est incluse par défaut dans la jdk et repose sur le package `java.rmi` et ses sous-packages. Sun maintient un site web dédié à l'API disponible à l'emplacement suivant : http://java.sun.com/products/jdk/rmi/index.html.

RMI est un modèle de communication permettant l'interopérabilité entre plusieurs Java Virtual Machine (JVM). Il existe de nombreuses autres techniques permettant de réaliser une certaine forme d'intéropérabilité entre des systèmes distants tels CORBA, dcop, SOAP, RPC, DCOM, …RMI possède quelques spécificités qui la distinguent de ces systèmes : 1) l'interopérabilité est limitée aux seul langage java 2) RMI est fondamentalement orienté objet 3) RMI ne requiert pas l'utilisation d'un langage de description des contrats clients-serveurs 4) il permet le téléchargement automatique de code 5) il autorise l'activation automatique des processus serveurs.

2 Architecture générale d'un système RMI

La figure 1 illustre l'architecture général d'un système RMI. Dans cette architecture, un serveur RMI désire rendre accessible un certain nombre de ses méthodes à des clients RMI. Le client et le serveur RMI sont tous les deux des objets java qui peuvent être exécutés sur des machines différentes. Une troisième composante agît comme un "service d'annuaire" entre le client et le serveur : la RMI registry. Elle permet au client de trouver un serveur distant qui pourra lui rendre certains services. *La notion de service est fondamentale en RMI et plus généralement en informatique distribuée et rejoint la notion de contrat en programmation orientée objet (POO).*
En RMI, les services sont réalisés par l'invocation de méthodes du serveur RMI.

Minimalement, un système RMI repose sur trois phases :

1. opération de bind/rebind :
   durant cette phase, le serveur RMI demande à la RMI registry de créer une nouvelle entrée dans son "annuaire" afin de rendre ces méthodes visibles aux clients RMI. La nouvelle entrée de l'annuaire associera un nom au serveur RMI.

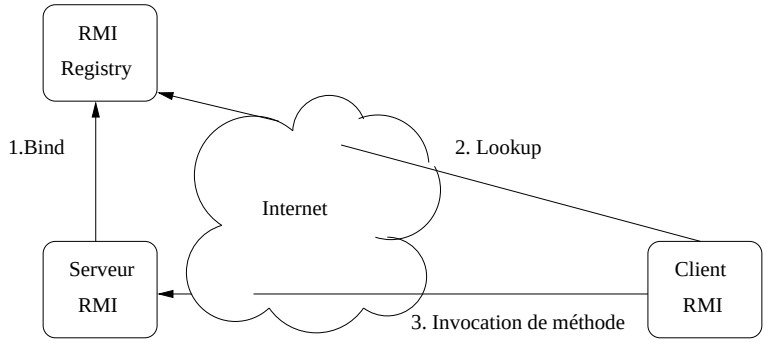

FIG. 1 – Architecture générale d'un système RMI.

2. opération de lookup :

   durant cette phase, le client RMI demande à la RMI registry de lui donner le serveur RMI associé à un certain nom dans son annuaire. Il est donc nécessaire que le client connaisse le nom sous lequel le serveur a été inscrit dans l'annuaire de la registry.

3. invocation de méthodes distantes :

   maintenant le client peut invoquer les méthodes du serveur. Les appels de méthodes distantes sont presque aussi simples que les appels de méthodes locales.

3 Composants d'un système RMI

Nous allons maintenant étudier en détails les différents composants d'un système RMI. Afin de mieux comprendre les différents concepts et leur réalisation en Java, ce document a été conçu comme un tutoriel permettant de créer entièrement un petit système distribué : un serveur de calcul capable de télécharger des tâches à effectuer et de renvoyer le résultat à un client désirant utiliser ce service. Cet exemple a été tiré d'un tutoriel de SUN [13].

Nous allons créer un serveur RMI qui offre une méthode (`executeTask`), cette méthode permettra de réaliser n'importe quelle tâche de calcul. Côté client, nous créerons une tâche de calcul et construirosn une application cliente qui invoquera, à distance, la méthode `executeTask` en passant en paramètre la tâche que nous avons créée. *Le serveur RMI téléchargera la tâche dynamiquement et l'exécutera puis retournera le résultat à l'application cliente. Le serveur RMI sera donc un véritable "serveur de calculs"*. La figure 2 présente une ue schématisée de l'application.

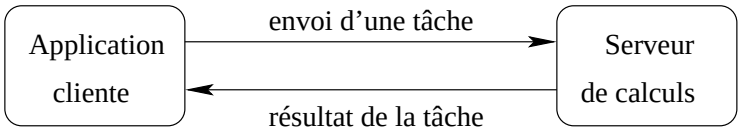

FIG. 2 – Schéma simplifié de l'application distribuée.

3.1 Configuration pour le tutoriel

Pour réaliser ce tutoriel, vous devez disposer d'une version 1.2 ou supérieure de la jdk. Cette jdk doit être fonctionnelle, c'est à dire que vous devez pouvoir appelez "java -version" dans une console sans voir d'erreur.

Nous avons réaliser le tutoriel en plaçant tous les fichiers dans le répertoire `/tmp` (sous Unix) ou `c:/temp` (sous windows). Ce répertoire devra comprendre les répertoires `compute`, `engine` et `client` qui seront les répertoires principaux du projet. Les fichiers `.java` et les fichiers `.class` d'une même classe seront placés dans un même répertoire dont le nom reflète le nom complet de la classe.

Par exemle, la classe `engine.ComputeEngine` sera stockée dans le fichier `/tmp/engine/ComputeEngine.java` (ou `c:/temp/engine/ComputeEngine.java`) et sera compilée dans le fichier `/tmp/engine/ComputeEngine.class` (ou `c:/temp/engine/ComputeEngine.class`).

3.2 Le serveur RMI

Le serveur RMI est le composant qui nécessite le plus gros effort de programmation dans un système RMI. Malgré tout, les étapes permettant de construire un serveur sont peu nombreuses et simples : 1) créer une interface de serveur RMI 2) créer une classe de serveur RMI 3) enregistrer la classe de serveur auprès de la RMI registry.

Comme nous l'avons mentionné précédemment, l'architecture RMI repose sur la notion de service : un client RMI demande à un serveur RMI de réaliser un service en appelant certaines de ses méthodes. *Ainsi, le client et le serveur sont liés par une sorte de contrat dans lequel sont spécifiés les services qu'un serveur RMI est suceptible d'offrir. En réalité, le client ne connaîtra jamais véritablement le serveur*

durant la réalisation du service, il n'intéragît pas directement avec un serveur RMI, il n'intéragît qu'avec ce contrat, avec l'interface du serveur tel qu'illustré à la figure 3.

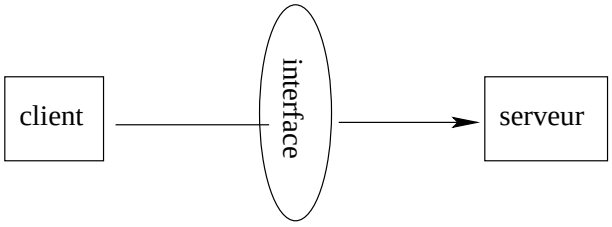

FIG. 3 – Le client intéragit avec le serveur distant via une interface qui décrit les services disponibles.

3.2.1 Création d'une interface de serveur RMI

Le contrat est une déclaration de tous les services que peut rendre un serveur. En RMI, les contrats sont représentés par une interface java (interface au sens de "classes sans implémentation" et non au sens d'interface graphique). Cette interface contient l'ensemble des signatures des méthodes qui sont invocables à distance.

Les interfaces qui décrivent des contrats RMI sont soumises à un certain nombre de contraintes :

- elles doivent étendre l'interface `java.rmi.Remote`,
- toutes les méthodes de l'interface doivent déclarer qu'elles peuvent envoyer une `java.rmi.RemoteException` dans leur clause throws,
- les paramètres et les types de retour d'une méthode de l'interface doivent être de l'un des types suivants : 1) un serveur RMI 2) une donnée de type primitif 3) un objet qui implémente l'interface `java.io.Serializable`.

La figure 4 donne un exemple d'interface RMI.

```
package compute;
import java.rmi.*;
public interface Compute extends Remote
{
  Object executeTask(Task t)
                  throws RemoteException;
}//interface
```

FIG. 4 – Exemple d'interface de serveur RMI.(/tmp/compute/Compute.java)

La méthode executeTask de l'interface `compute.Compute` accepte un paramètre de type

`compute.Task`. L'interface `compute.Task`, telle que définie à la figure 5 dérive de `java.io.Serializable`. Elle peut donc être utilisée comme type de paramètre d'une méthode distante conformément aux contraintes que nous avons énoncées plus haut.

```
package compute;
import java.io.Serializable;
public interface Task extends Serializable
{
  Object execute();
}//class
```

FIG. 5 – En dérivant de java.io.Serializable, les instances de compute.Task peuvent légitimement être utilisées comme paramètre d'une méthode distante. (/tmp/compute/Task.java)

3.2.2 Création d'une classe de serveur RMI

Les serveurs RMI sont des instances de classes java qui implémentent un certain type d'interface "contrat". Les classes de serveurs RMI sont des classes concrètes qui doivent fournir une implémentation de toutes les méthodes déclarées dans l'interface. Seules les méthodes de la classe de serveur RMI dont la signature est spécifiée par l'interface seront accessibles à distance. *Les éventuelles autres méthodes de cette classe ne faisant pas partie du contrat ne sont donc pas invocables à distance.*

Les classes de serveurs RMI sont également soumises à certaines contraintes :

- elles doivent dériver de la classe `java.rmi.server.UnicastRemoteObject`,
- elles doivent étendre une interface de serveur RMI, elles doivent donc implémenter *toutes* les méthodes définies dans l'interface,
- le constructeur sans paramètre de ces classes doivent déclarer qu'ils peuvent envoyer une `java.rmi.RemoteException` dans leur clause throws.

La figure 6 illustre les relations d'héritage du côté serveur.

La classe abstraite `java.rmi.server.UnicastRemoteObject` définit un ensemble de méthodes permettant de répondre à des appels de méthodes distants. En dérivant de la classe `java.rmi.server.UnicastRemoteObject`, les classes acquièrent le statut de pouvoir publier certaines de leurs méthodes, de sorte qu'elles soient invocables à distance. Les classes de serveurs RMI doivent implémenter une interface de serveur RMI comme celle que avons créée à l'étape précédente. Afin d'implémenter l'interface de serveur RMI, les classes de serveur sont tenues de fournir une implé-

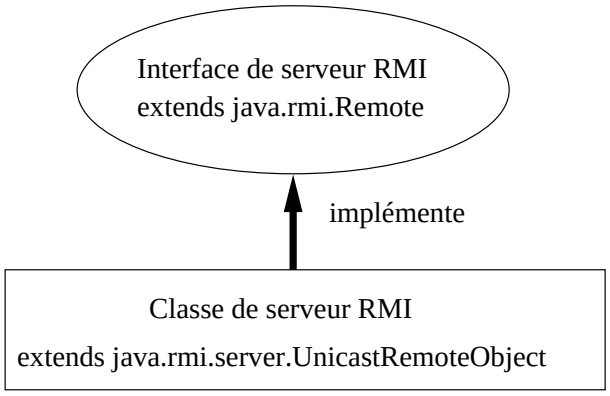

FIG. 6 – L'héritage dans la création d'un serveur RMI.

mentation de toutes les méthodes de l'interface. La figure 7 nous donne un exemple de classe de serveur RMI qui peut rendre les services spécifié par l'interface `compute.Compute` que nous avons définie à la figure 4.

```
package engine;
import java.rmi.*;
import java.rmi.server.*;
import compute.*;
public class ComputeEngine
             extends UnicastRemoteObject
             implements Compute
{
  public ComputeEngine()
                  throws RemoteException
  {
    super();
  }//cons
  public Object executeTask(Task t)
                  throws RemoteException
  {
    return t.execute();
  }//met
}//class
```

FIG. 7 – Exemple de serveur RMI. (/tmp/engine/ComputeEngine.java)

Ici, la classe de serveur RMI ne comporte qu'une seule méthode : `executeTask`. Cette méthode va simplement retourner le résultat de l'exécution de la tâche qui lui est passée en paramètre. Les méthodes qui implémentent une des méthodes de l'interface ne sont pas tenues de spécifier qu'elles peuvent lancer une exception dans leur clause throws. Nous avons choisi de conserver le niveau d'exception défini dans l'interface pour indiquer leur caractère "distant".

Nous sommes également obligés de définir un constructeur sans paramètre pour cette classe. En ef-

fet, la classe `java.rmi.server.UnicastRemoteObject` définit un constructeur sans paramètre pouvant lancer une `java.rmi.RemoteException`. Comme nous le savons, si nous ne définissons pas de constructeur sans paramètre pour une classe, Java crée un constructeur par défaut qui appelle simplement le constructeur de la classe mère. Ce constructeur ne peut malheureusement pas être créé ici pusqu'il violerait une règle de programmation Java : il invoquerait une méthode pouvant lancer une exception sans la traiter. Nous sommes donc dans l'obligation de fournir un constructeur sans paramètre explicite traitant l'exception. Le traitement choisi pour l'exemple est très simple : nous déléguons à l'appelant de notre constructeur le traitement d'erreur en indiquant une `RemoteException` dans la clause throws du constructeur de `engine.ComputeEngine`.

3.2.3 Inscription du serveur RMI auprès de la RMI registry

Comme nous l'avons mentionné dans la section 2, un serveur RMI doit être inscrit auprès du service d'annuaire de la RMI registry. Ici, nous avons choisi de réaliser cette opération à l'intérieur de la classe du serveur : dans la méthode main de la classe `engine.ComputeEngine`. La figure 8 illustre cette technique. L'inscription d'un serveur RMI se fait grâce à la méthode statique `java.rmi.Naming.rebind( String url, Remote serveurRmi)`.

L'invocation de cette méthode est suceptible de lancer des exceptions de type `MalformedURLException, RemoteException` ou `AccessException`, c'est la raison pour laquelle nous l'avons placée dans un bloc try/catch (ici nous avons choisi d'attraper toutes les exceptions indifféremment dans un seul bloc catch généraliste).

Le premier paramètre de la méthode rebind est une URL devant répondre au schéma suivant : `rmi://<hôte>:[port]/<nom>` où `<hôte>` fait référence au nom d'hôte ou à l'adresse IP de l'ordinateur hébergeant la RMIregistry, `<port>` fait référence au numéro de port de la registry (par défaut 1099) et `<nom>` au nom donné au serveur RMI dans l'annuaire de la registry (cf section 5.1 pour plus de détails sur la partie `<nom>` de l'URL). Notez également que le protocole `rmi` est optionnel, une URL RMI peut s'écrire : `//<hôte>:[port]/<nom>`.

Le second paramètre de cette méthode est l'instance de service accessible à distance que nous désirons placé dans la registry. Cet objet sera désormais accessible depuis la registry par le `<nom>` que nous avons spécifié dans le premier paramètre.

```
public static void main( String[] args )
{
  if ( System.getSecurityManager() == null )
    System.setSecurityManager( new RMISecurityManager() );
  try
  {
    ComputeEngine engine = new ComputeEngine();
    Naming.rebind( "//localhost/serveurDeCalcul", engine );
  }//try
  catch( Exception ex )
  {
     ex.printStackTrace();
  }//catch
}//met
```

FIG. 8 – Enregistrement d'un serveur RMI auprès de la registry.

On notera que, dans les versions 1.4 et antérieures de la jdk, il est impossible, pour des raisons de sécurité, d'exécuter le serveur RMI et la RMI registry sur des ordinateurs différents. Autrement dit, l'URL utilisée est toujours de la forme "//localhost/<nom>". Il est, en effet, impossible d'enregistrer un objet depuis une machine différente de celle qui habrite la RMI registry.

Après l'exécution de cette méthode, le serveur RMI sera enregistré auprès de la RMI registry et prêt à être utilisé par un client distant.

Nous verrons dans la section 4 comment procéder à la compilation et au déploiement de la classe du serveur RMI.

3.3 Le client RMI

Si la création d'un serveur RMI peut paraître difficile au début de l'apprentissage du RMI, en revanche, la création d'un client RMI est triviale. Il n'existe aucune contrainte sur un client RMI, toute classe Java peut faire appel au services d'un serveur RMI. La seule différence entre un appel local de méthode et un appel distant de méthode est que ce dernier doit avoir lieu à l'intérieur d'un bloc try/catch. En effet, un appel de méthode distante peut echouer pour toutes sortes de raisons comme une panne de réseau, un bris sur l'ordinateur distant, etc.

Un appel à une méthode distante s'effectue toujours en deux temps : 1) obtenir une référence sur le serveur distant de la part de la RMI registry 2) invoquer une méthode de cette référence. La figure 9 représente un client RMI minimaliste pour notre serveur RMI. Comme on peut le constater, il n'existe

aucune contrainte d'héritage sur le client : il ne doit étendre aucune classe ou implémenter aucune interface particulière. Ici, tout le code permettant de réaliser un appel distant à été placé dans la méthode `main` de la classe mais nous aurions pu réaliser cette opération depuis n'importe quelle autre méthode de la classe.

```
package client;
import java.rmi.*;
import java.math.*;
import compute.*;
public class ComputePi
{
  public static void main( String args[] )
  {
    if ( System.getSecurityManager() == null )
      System.setSecurityManager( new RMISecurityManager() );
    try
    {
      Compute comp = ( Compute ) Naming.lookup(
                              "//serveurA/serveurDeCalcul" );
      Task task = new TaskAdd(Integer.parseInt(args[0]),
                              Integer.parseInt(args[1]));
      Object res =  (comp.executeTask(task));
      System.out.println( res );
    }//try
    catch( Exception ex )
    {
      ex.printStackTrace();
    }//catch
  }//met
}//class
```

FIG. 9 – Utilisation d'un serveur RMI.(/tmp/client/ComputePi.java)

La première chose effectuée par le client RMI est d'installer un RMISecurityManager à l'aide de la méthode statique System.setSecurityManager. Nous reviendrons, à la section 6.5, sur le pourquoi de cette instruction.

Ensuite, le client RMI essaye d'obtenir une instance du serveur RMI. Cette opération se fait par l'intermédiaire de la RMI registry. Le client demande à la RMI registry de lui envoyer une référence sur l'instance de serveur RMI que nous avons enregistré au préalable. Pour ce faire, il utilise la méthode statique `Naming.lookup` et lui passe en paramètre l'URL à laquelle le serveur est enregistré. Cette URL est équivalente à celle que nous avons utilisée à la figure 8 pour inscrire le serveur dans l'annuaire. Notez bien que nous ne pouvons plus utiliser une URL avec l'hôte "localhost", car nous supposons que le client et le serveur RMI s'exécutent sur des machines différentes. Il nous faut donc donner le nom DNS ou l'adresse IP de l'ordinateur abritant la RMI registry du serveur RMI.

Il est essentiel de remarquer ici que le résultat de cette instruction est converti (typeCasting) en "le type de l'interface du serveur RMI" et non pas en "le type de la classe du serveur RMI". Si nous avions converti le résultat de la méthode `lookup` en `engine.ComputeEngine`, nous aurions immanquablement reçu une exception durant la phase d'exécution du client. Il faut que vous gardiez à l'esprit le fait que le serveur RMI n'est *jamais* téléchargé sur le poste client, la référence renvoyéz par l'opération de lookup n'est donc pas une référence sur un `engine.ComputeEngine` mais simplement une référence sur quelque chose qui permet d'appeler les méthodes de l'objet distant. La section 5.2 donnera de plus amples explications sur cette subtilité de RMI.

Finalement, après avoir obtenu une référence vers le serveur RMI, il ne nous reste plus qu'à l'utiliser exactement comme nous le ferions avec une référence sur un objet local : en appelant une de ses méthodes. Malgré tout, il est important de noter que l'appel d'une méthode distante, doit se faire à l'intérieur d'un bloc try/catch. Cette situation est une conséquence logique du fait d'avoir défini les méthodes du serveur RMI (figure 7) comme étant suceptibles de lancer des `java.rmi.RemoteException`. On remarquera que la gestion des exceptions distantes est tout à fait homogène avec la gestion classique des exceptions dans le langage java.

L'opération de lookup peut lancer des exceptions de type : `NotBoundException`, `MalformedURLException` ou `RemoteException`. Ici, nous avons choisi d'intercepter ces exceptions indifféremment et avons uniquement écrit un block catch généraliste capable d'attraper toutes les exceptions possibles.

Comme vous pouvez le voir, l'application cliente utilise un objet de type `compute.TaskAdd`. La classe `client.TaskAdd`, dont le code est présenté à la figure 10, est une tâche qui permet simplement d'ajouter deux entiers. La classe `client.TaskAdd` implémente l'interface `compute.Task` afin de pouvoir être passé en paramètre à la méthode `execute` du serveur RMI. Cette tâche n'est pas très intéressante et a été simplifiée pour vous permettre de vous concentrer sur RMI. Vous trouverez en annexe (section 9) une tâche plus consistante capable de renvoyer la valeur de π avec un très grande précision.

4 Compilation, déploiement et exécution

Cette section complète le tutoriel sur RMI. Elle explique comment compiler, lancer et déployer l'application distribuée. Le lecteur trouvera dans les sections suivantes de plus amples explications sur les

```
package client;

import compute.*;
import java.math.*;

public class TaskAdd implements Task
{
  private int a;
  private int b;

  /** Construit une tâche. */
  public TaskAdd(int a, int b)
  {
    this.a = a;
    this.b = b;
  }//cons

 /**
  * Cette méthode renvoie la valeur de l'addition
  * des deux int passés en paramètre au constructeur.
  */
  public Object execute()
  {
    return new Integer( a+b );
  }//met
```

FIG. 10 – Une tâche d'addition de 2 entiers.(/tmp/client/TaskAdd.java)

commandes utilisées ici.

4.1 Compilation

Désormais, nous avons créé toutes les classes nécessaires au fonctionnement de l'application. La première étape consiste à compiler toutes les classes du projet.

Pour compiler tous les fichiers du projet, il suffit de taper :

```
javac -classpath /tmp /tmp/compute/*.java
javac -classpath /tmp /tmp/engine/*.java
javac -classpath /tmp /tmp/client/*.java
```

Maintenant que toutes les classses sont compilées, il nous faut générer les stubs de notre serveur RMI (cf section 5.2) :

```
rmic -v1.2 -classpath /tmp/ engine.ComputeEngine
```

Cette commande aura pour effet de générer le fichier `/tmp/engine/ComputeEngine_Stub.class`

4.2 Déploiement

Le déploiement d'une application RMI est la phase la plus complexe. La difficulté tient au fait qu'il faut utiliser plusieurs ordinateurs pour que l'application soit distribuée, il nous faut donc toujours penser

à la phase de déploiement lorsque l'on programme une application RMI afin de bien différencier ce qui doit résider du côté du serveur RMI et du côté de l'application cliente. Il existe une grande variété de livres et de tutoriaux sur RMI, un certain nombre d'entre eux cachent une partie de la complexité du déploiement en faisant exécuter le client et le serveur sur une même machine, d'autres encore n'utilisent pas le téléchargement dynamique de code (cf sections 6.3 et 6.4).

Nous vous présentons, ici, la technique de déploiement la plus générique, celle qui utilise au mieux les fonctionnalités offertes par RMI. Mais il ne faut pas le cacher, cette technique est aussi la plus complexe car elle requiert l'utilisation de plusieurs ordinateurs et d'un serveur web. TouTEs les étudiantEs du département peuvent utiliser le serveur web du département et également utiliser certaines machines à distance (comme ift-linux1.ift.ulaval.ca et ift-linux2.ift.ulaval.ca), mais peu savent utiliser ces technologies.

Nous vous présenterons donc trois techniques de déploiement. La première sera appelée "déploiement simpliste" (section 4.3) et vous permettra d'exécuter rapidement le tutoriel mais ne représente pas un déployement réaliste. La deuxième sera nommée "déploiement non distribué" (section 4.4), elle n'utilisera qu'un seul ordinateur et ne nécessitera pas de serveur web. La troisième sera nommée "déploiement distribué" (section 4.5), elle utilisera 2 ordinateurs et un serveur web. *Nous vous conseillons très fortement de ne pas vous limiter à la première technique puisqu'elle est simpliste et ne reflète pas la réalité. L'application peut très bien fonctionner sur une seule et même machine mais il est préférable d'en utiliser plusieurs pour bien percevoir le caractère réparti de l'application.*

Vous devrez donc suivre l'une ou l'autre de ces deux techniques durant le tutorial, pas les trois en même temps !

Maintenant que les fichiers sont compilés, nous allons créer un fichier jar pour déployer plus facilement les classes qui devront transiter par le réseau. RMI n'impose pas l'utilisation de fichiers jar pour le déploiement. Il nous suffirait de déployer les fichiers `.class` directement sur la racine d'un serveur web. Malgré tout, l'utilisation de fichiers jar est vivement recommandée car elle accélère les temps de transfert et permet de fournir un bien livrable facilement déployable.

Pour créer les fichiers jar, il suffit de taper la commande :

```
jar cvf deploy-server.jar -C /tmp engine/ComputeEngine_Stub.class
                          -C /tmp compute/Compute.class
                          -C /tmp compute/Task.class

jar cvf deploy-client.jar -C /tmp client/TaskPi.class
```

Si tout va bien, vous devriez voir apparaître ces deux messages :

```
manifest ajouté
ajout : compute/Compute.class(entrée = 238) (sortie = 168)(29% compressés)
ajout : compute/Task.class(entrée = 166) (sortie = 138)(16% compressés)
ajout : engine/ComputeEngine_Stub.class(entrée = 1814) (sortie = 915)

manifest ajouté
ajout : client/TaskPi.class(entrée = 1210) (sortie = 722)(40% compressés)
```

Une fois les fichiers de déploiement créés, il nous faut maintenant les déployer sur les différentes machines participant à l'application distribuée. Vous aurez également besoin d'un fichier de sécurité pour faire fonctionner l'application. Vous trouverez un exemple de ce genre de fichier à la figure 17.

4.3 Déploiement simpliste

Pour exécuter rapidement l'application, nous vous proposons un mode de déployement très simple. Ce mode de déployement est en fait *trop simple* et ne figure dans ce document que pour vous montrer que l'exécution d'un programme RMI n'est pas complexe en soit.

Pour exécuter rapidement l'exemple, placez vous dans le répertoire temp et tapez :

```
rmiregistry &

java -Djava.security.policy=security.policy
     -Djava.rmi.server.codebase=file:///tmp/
     -cp . engine.ComputeEngine &

java -Djava.security.policy=security.policy
     -Djava.rmi.server.codebase=file:///tmp/
     -cp . client.ComputePi 10
```

Sous windows, n'utilisez pas le symbole '&', utilisez plutôt trois consoles DOS différentes.

4.4 Déploiement non distribué

Nous supposerons donc maintenant que nous disposons d'un seul ordinateur. Sur cet ordinateur, nous allons simuler deux ordinateurs A et B. L'ordinateur A hébergera l'objet serveur RMI tandis que l'ordinateur B hébergera l'application cliente. Chacun de ces ordinateurs sera simulé par un répertoire différent sur l'ordinateur dont nous disposons. Par exemple, le répertoire `/tmp/A/` contiendra les fichiers de la machine A et le répertoire `/tmp/B/` contiendra les fichiers de la machine B.

Sur l'ordinateur A, nous déposerons les fichiers `.class` du serveur soient les packages `compute` et `engine` mais pas celles du package `client`. Sur l'ordinateur B, nous déposerons les fichiers `.class`

du client soient les packages `compute` et `client` mais pas celles du package `engine`. Soit, plus précisément :

répoitre `/tmp/A/`	sur répertoire `/tmp/B/`
compute.compute	compute.compute
compute.Task	compute.Task
engine.ComputeEngine	client.ComputePi
engine.ComputeEngine_Stub	
security.policy	security.policy

La distribution des fichiers `class` *est très importante : si vous ne donnez pas toutes les classes à une des deux applications (client ou serveur), le logiciel ne fonctionnera pas. Si, en revanche, vous lui en donnez trop, l'application s'exécutera mais sans utiliser le téléchargement dynamique de code, qui est une des principales fonctionnalités de RMI.*

Une fois les fichiers des deux applications répartis dans les répertoires A et B, vous devez maintenant placer les fichiers jar sur une serveur web. Comme nous ne disposons pas de serveur web, nous allons le simuler également, dans le répertoire `/tmp/web`. Vous devrez donc copier les deux fichiers jars créés précédement dans le répertoire `/tmp/web`. Assurez que les deux fichiers et le répertoire du serveur web sont bien accessibles à tout le monde en lecture. Les URLs que vous utiliserez pour le téléchargement de code seront de la forme : file:///tmp/web/deploy-client.jar et file:///tmp/web/deploy-server.jar. Ces URLs fonctionneront uniquement localement, votre application ne pourra donc pas s'exécuter sur plusieurs machines.

4.5 Déploiement distribué

Les étudiantEs du département peuvent se connecter, en ssh/sftp, sur les machines `ift-linux1.ift.ulaval.ca` et `ift-linux2.ift.ulaval.ca`. Vous devez utiliser le même mot de passe et le même mot d'usager que sur les ordinateurs des laboratoires du département. Si vous ne disposez pas de client ssh/sftp sur votre ordinateur, vous pouvez en télécharger un sur le site ftp du département ftp://ftp:toto@ftp.ift.ulaval.ca/pub/SSHWin-2.3.0.exe. Vous aurez ainsi accès à deux ordinateurs (le vôtre et celui du département).

De la même manière, touTEs les étudiantEs du département disposent d'un répertoire accessible par le web. Pour utiliser ce répertoire, loggez-vous sur une station Unix du département

comme `ift-linux1.ift.ulaval.ca` et placez les fichiers jar dans votre répertoire public_html : `~/public_html`. Ensuite, vous devrez changer les permissions de ce répertoire de manière à le rendre accessible à touTEs avec la commande `chmod -R 755 ~/public_html`. Notez qu'il est également très facile d'installer un serveur web sur votre propre ordinateur. Le meilleur serveur web est disponible sur le site de la fondation apache (pour windows et linux) : www.apache.org.

Nous supposerons donc maintenant que nous disposons de deux ordinateurs A et B. L'ordinateur A hébergera l'objet serveur RMI tandis que l'ordinateur B hébergera l'application cliente. Sur l'ordinateur A, nous déposerons les fichiers `.class` du serveur soient les packages `compute` et `engine` mais pas celles du package `client`. Sur l'ordinateur B, nous déposerons les fichiers `.class` du client soient les packages `compute` et `client` mais pas celles du package `engine`. Soit, plus précisément :

sur l'ordinateur A	sur l'ordinateur B
compute.compute	compute.compute
compute.Task	compute.Task
engine.ComputeEngine	client.ComputePi
engine.ComputeEngine_Stub	
security.policy	security.policy

La distribution des fichiers `class` *est très importante : si vous ne donnez pas toutes les classes à une des deux applications (client ou serveur), le logiciel ne fonctionnera pas. Si, en revanche, vous lui en donnez trop, l'application s'exécutera mais sans utiliser le téléchargement dynamique de code, qui est une des principales fonctionnalités de RMI.*

Une fois les fichiers des deux applications répartis sur les ordinateurs A et B, vous devez maintenant placer les fichiers jar sur un serveur web. Il vous suffit de copier les deux fichiers jar dans le répertoire `~/public_html` de votre compte Unix. Vous devez désormais être capable de télécharger vos fichiers par le web, avec un navigateur, aux adresses <votre URL>/deploy-client.jar et <votre URL>/deploy-server.jar.Les URLs seront de la forme : www.ift.ulaval.ca/~⟨IDauDepartement⟩/deploy-client.jar et www.ift.ulaval.ca/~⟨IDauDepartement⟩/deploy-server.jar.

4.6 Exécution

Maintenant il ne reste plus qu'à exécuter les 2 applications. Sur l'ordinateur A, commencez par lancer une RMI registry avec la commande :

`rmiregistry &` (sous Unix) et `start rmiregistry` (sous windows).[1]

Ensuite, placez-vous dans le répertoire parent de `compute` et tapez :

```
java -cp . -Djava.security.policy=security.policy
           -Djava.rmi.server.codebase=<votre URL>/deploy-server.jar
           engine.ComputeEngine
```

Du côté client, il suffit de taper :

```
java -cp . -Djava.security.policy=security.policy
           -Djava.rmi.server.codebase=<votre URL>/deploy-client.jar
           client.ComputePi 10 5
```

Les deux derniers paramètres de la ligne de commande sont les entiers à ajouter.

Vous venez de terminer le tutoriel ! Ouf ;)

L'application déployée est représentée par la figure 11. Nous aurions pu, utiliser deux serveurs web différents pour le client et pour le serveur puisque ces deux applications n'ont absolument aucune raison de partager leur site de téléchargement de code. Mais ce cas de figure aurait compliqué encore l'exemple...

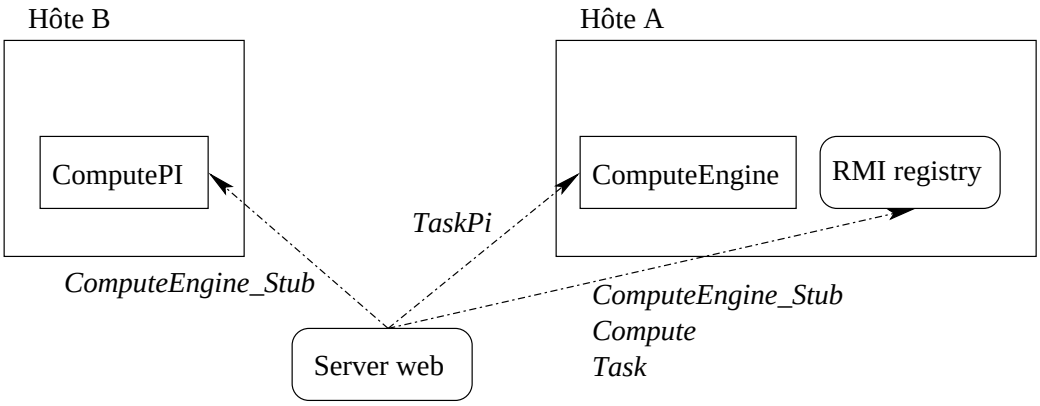

FIG. 11 – Schéma de déploiement de l'application exemple.

L'exécution va se dérouler comme suit : le serveur RMI va s'enregistrer auprès de la RMI registry. Ensuite, le client va demander à la registry de lui fournir une référence sur le serveur RMI. Comme le client ne possède pas le stub du serveur, celui-ci sera téléchargé depuis l'URL ⟨votreURL⟩/deploy-server.jar. Une fois le stub acquis, le client va invoquer la méthode distante du serveur en lui passant en paramètre l'objet `TaskAdd`.

[1] Faîtes bien attention lorsque vous lancer la registry : la registry ne doit pas trouver le stub du serveur RMI dans son classpath, sinon, il serait impossible de le télécharger pour le client. Assurez vous que la variable d'environnement CLASSPATH ne contient aucune entrée permettant d'accéder au stub. La registry doit rester active durant jusqu'à la fin de l'exécution du client, ne la fermez pas.

Cette fois, c'est le serveur RMI qui ne connait pas la classe `client.TaskAdd`, il va donc la télécharger depuis l'URL ⟨votreURL⟩/deploy-client.jar. Ensuite, le serveur RMI va exécuter la tâche et renvoyer son résultat par le réseau au client qui pourra l'afficher dans la console.

4.7 Erreurs courantes à l'exécution

Si vous obtenez, dans le client ou le serveur, l'erreur suivante :

```
java.security.AccessControlException: access denied
  (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
  at ...
```

Cela signifie que vous n'avez correctement donner le chemin qui permet d'accéder au fichier de security.policy, ce qui empêche de contacter la registry. Vérifiez également que vous bien orthographié l'option `-Djava.security.policy`

Si vous obtenez, dans le client, l'erreur suivante :

```
java.rmi.UnmarshalException: error unmarshalling return;
  nested exception is:
  java.lang.ClassNotFoundException: engine.ComputeEngine_Stub
  at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
  at java.rmi.Naming.lookup(Naming.java:83)
  at ...
```

Cela signifie que vous votre client ne peut pas trouver la classe de stub du serveur. Cette erreur peut avoir essentiellement 3 causes :

- vous avez mal tapé l'option `-Djava.rmi.server.codebase dans le serveur`, vérifiez également que votre URL est correcte.
- le fichier jar du serveur n'est pas accessible depuis internet. Vérifiez que le fichier est bien accessible en utilisant un navigateur web.
- sur certaines plates-formes[2], la résolution des noms d'hôtes laisse à désirer, indiquez alors l'adresse IP de votre serveur web dans <votre URL> aussi bien côté client que serveur plutôt que le nom d'hôte du serveur.

Finalement, si vous obtenez, en exécutant le client, l'erreur suivante :

```
java.rmi.ServerException: RemoteException occurred in server thread;
  nested exception is:
  java.rmi.UnmarshalException: error unmarshalling arguments;
  nested exception is:
  java.lang.ClassNotFoundException: client.TaskPi
  at ....
```

[2] Sans commentaires.

Cela signifie que le fichier jar contenant votre classe de déploiement côté serveur n'est pas accessible. Reportez vous à l'erreur précédente pour avoir des explications sur la cause de cette erreur.

5 Fondements théoriques de l'informatique distribuée

L'API RMI permet de réaliser facilement des systèmes distribués en java en déchargeant le programmeur d'une multitude de contraintes liées à l'exécution de méthodes distantes. RMI simplifie la tâche de programmation en cachant un certain nombre de difficultés au créateur du client et du serveur RMI. Dans cette section, nous allons étudier plus en détails les mécanismes mis en oeuvre par RMI et étudier les fondements théoriques sur lesquels repose RMI.

5.1 La RMI registry

La RMI registry est un composant logiciel dont les méthodes, comme `lookup` et `rebind`, sont invocables à distance. En fait, il s'agit d'un serveur RMI au sens où nous l'avons défini à la section 3.2.2 ; à ceci près qu'il n'est, bien sûr, pas enregistré auprès d'une RMI registry, sans quoi nous ne disposerions pas d'une séquence de bootstrap permettant d'initialiser le processus...

L'interface qui décrit les services distants rendus par la RMI registry est définie par l'interface `java.rmi.registry.Registry`. Cette interface offre les méthodes :

- `public void bind( String name, Remote obj)`
  qui permet d'inscrire un serveur RMI dans la structure d'annuaire de la RMI registry.
- `public void rebind( String name, Remote obj)`
  qui permet d'inscrire un serveur RMI dans la structure d'annuaire de la RMI registry. La différence entre bind et rebind est que si un objet à déjà été enregistré auprès de la RMI registry sous le nom *name*, bind lancera une exception tandis que rebind remplacera le premier serveur RMI par le second pour cette entrée dans la structure d'annuaire.
- `public void unbind( String name )`
  qui permet de supprimer un serveur RMI de la structure d'annuaire de la RMI registry.
- `public Remote lookup( String name)`
  qui permet d'obtenir une référence sur un objet enregistré dans la structure d'annuaire de la RMI registry.

– `public String[] list()`

qui permet d'obtenir la liste des entrées de la structure d'annuaire auxquelles des serveurs RMI sont attachés.

Étant donné que la RMI registry est un serveur RMI et que ce serveur RMI ne peut pas être inscrit auprès d'une RMI registry, toute la problématique est de pouvoir obtenir une instance de la RMI registry : "Comment trouver un serveur RMI". Ce problème est résolu par l'intermédiaire de la classe `java.rmi.Naming`. La classe `Naming` permet de trouver une registry sur hôte et un port connu, ensuite, elle lui enverra le *name* sous lequel les serveurs RMI sont accessibles. Pour cette raison, les URL utilisées par Naming contiennent un nom d'hôte, un numéro de port et le nom associé au serveur RMI désiré.

La structure "d'annuaire" d'une RMI registry est une structure hiérarchique relativement comparable à la structure d'un disque dur sous UNIX. Elle possède une racine (/) à partir de laquelle on peut préciser des noms de "répertoires", de "sous-répertoires" ou de "fichiers". La figure 12 illustre e à quoi peut ressembler la structure d'annuaire d'une RMI registry [3]

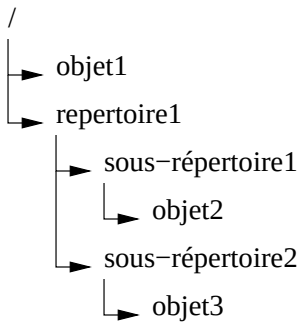

FIG. 12 – Structure d'annuaire d'une RMI registry.

5.2 Les stubs et les skeletons

Comme nous l'avons vu dans la figure 9, les appels de méthodes sur un objet distant sont pratiquement aussi simples que des appels locaux. Malgré tout, il semble évident que certains paramètres inhérents à la programmation distribuée doivent être pris en compte comme l'ouverture d'une connexion réseau, la mise en place d'un "timeout" après lequel un appel est considéré comme échoué, la gestion des pannes

[3] Une RMI registry ne permet pas que plusieurs serveurs RMI soient enregistrés sous le même nom. En revanche, rien n'interdit qu'un même serveur RMI soit enregistré sous plusieurs chemins différents dans la structure de répertoires de la registry. Même si cette possibilité n'est jamais utilisée dans la pratique puisqu'elle peut rapidement prêter à confusion, il serait ainsi plus exact de parler de structure de graphe que de structure arborescente pour qualifier la structure de la registry.

réseaux, la transmission des paramètres de la méthode, la réception des résultats et bien d'autres choses encore.

D'une manière générale en informatique distribuée, l'approche proposée pour résoudre ce problème est d'utiliser des objets proxy (prestataires) qui servent d'intermédiaire entre la logique représentée par le contenu des classes distribuées et le médium de communication. Ainsi, il existe une classe, côté serveur, nommée *skeleton* (squelette) et une autre classe côté client nommée *stub* (talon/souche) qui prennent un charge tout le protocole de communication RMI. Ces classes effectuent le véritable travail d'invocation des méthodes à distance. Elles agissent comme des prestataires des classes distribuées sur le réseau.

Ainsi tous les appels distants de méthodes sont en réalités des appels locaux des méthodes du Stub qui se charge d'effectuer véritablement les appels sur le réseau. Les appels distants sont de simples échanges d'information sur des sockets entre le stub et le skeleton. Les informations échangées sont principalement le nom de la méthode à invoquer, les paramètres, les types de retour ou les exceptions résultant de l'appel de méthode. Le skeleton, de son côté, décode ces informations et effectue un appel local vers la classe de serveur RMI. On peut donc considérer que les stubs et les skeletons sont des wrappers (coquiles/enveloppes) qui agissent comme intermédiaires entre le réseau et la logique de traitement incluse dans le serveur et le client RMI comme illustré à la figure 13.

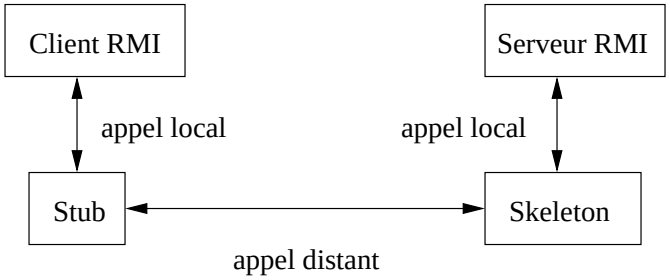

FIG. 13 – Protocole de communication entre un client et un serveur par l'intermédiaire du Stub et du Skeleton.

Le but d'une API comme RMI ou CORBA est essentiellement de procurer un framework (cadre) standard dans lequel s'effectue la programmation distribuée. Afin d'éviter au programmeur d'avoir à prendre en charge lui-même la programmation des classes Stubs et Skeletons, celles-ci sont générées automatiquement à partir des classes que le programmeur désire distribuer. La création automatique de ces classes de proxy permet de libérer le programmeur des contraintes de la programation réseau tout en standardisant le protocole utilisé pour la communcation entre les clients et les serveurs distribués. Ainsi, le protocole utilisé ne varie pas d'un programmeur à l'autre. Cette standardisation est la clef de

l'intéropérabilité entre les systèmes distribués : elle assure que tous les clients peuvent invoquer les méthodes de tous les serveurs.

En RMI, les classes de Stubs et de Skeletons sont créées à partir de la classe du serveur. Le Skeleton est déployé sur la machine du serveur tandis que le Stub est déployé côté client. Lorsque le client demande à la RMI registry de lui donner une référence sur le serveur RMI, celle-ci lui retourne en fait une référence sur le Stub du serveur. Tout comme la classe de serveur, le Stub implémente l'interface du serveur. Il dispose donc de toutes les méthodes que le client désire appeler sur le serveur (puisqu'implémenter une interface en java implique de fournir une implantation de toutes les méthodes de l'interface).

C'est cette petite subitilité du RMI qui explique pourquoi , à la figure 9, la référence renvoyée par la RMI registry est convertie en l'interface du serveur : `Compute comp = (Compute) Naming.lookup( url );` et non pas `ComputeEngine comp = (ComputeEngine) Naming.lookup( url );`. La deuxième instruction ne pourrait pas fonctionner car ce que reçoit le client RMI de la part de la RMI registry est une référence vers quelque chose qui implémente l'interface de serveur RMI (le Stub) et non pas le serveur RMI lui-même. Le serveur RMI n'est donc jamais téléchargé sur le poste client. La figure 14 propose une modélisation UML du lien d'héritage entre le serveur RMI, l'interface du serveur RMI et le stub.

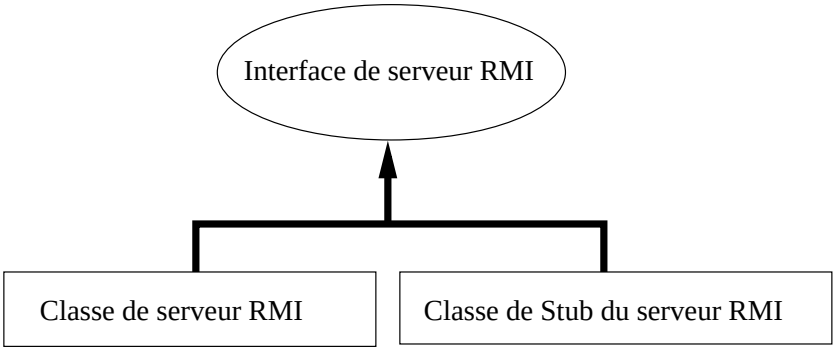

FIG. 14 – Lien d'héritage entre la classe de serveur, l'interface et le stub en RMI.

5.3 rmic (rmi compiler)

rmic (rmi compiler) est le compilateur de stubs et de skeletons pour rmi, c'est grace à lui que l'on peut générer automatiquement les proxy d'une serveur RMI. rmic est inclus par défaut dans la jdk. Il doit être invoqué de la façon suivante : rmic <options> <noms des classes de serveurs RMI> où les options

peuvent être :

- -keep ou -keepgenerated
  conserve les fichiers sources intermédiaires générés pour les stubs et les skeletons,
- -g
  génère de l'information de débogage,
- -depend
  recompile récursivement les fichiers qui ne sont pas à jour,
- -nowarn
  Ne génère pas de messages d'avertissement,
- -verbose
  compilation bavarde,
- -classpath <path>
  permet de spécifier le classpath à utiliser pour trouver les classes à compiler.
- -d <directory>
  indique où placer les fichiers class générés,
- -J<runtime flag>
  permet de donner un argument à la jvm avant la compilation
- -v1.1
  crée des stubs et des skeletons pour le protocole de la jdk 1.1
- -vcompat (defaut)
  crée des stubs et des skeletons compatible à la fois avec la jdk 1.1 et la plateforme java 2.
- -v1.2
  crée uniquement des stubs et PAS de skeletons pour le protocole de la plateforme java 2 seulement.

rmic génère les stubs à partir des fichiers class compilés d'un serveur RMI et non pas à partir de son code source. Ainsi, dans l'exemple que nous avons utilisé jusqu'à maintenant, nous utiliserons d'abord javac pour compiler le serveur `engine.ComputeEngine` puis, ensuite, rmic pour générer ses stubs avec la commande :

```
rmic -classpath tmp -d /tmp -v1.2 engine.ComputeEngine
```

Cette commande produira le fichier `engine.ComputeEngine_Stub` que nous pourrons déployer sur le poste client ou bien transférer automatiquement sur le poste client (cf section 6.3).

Les dernières versions de la jdk (1.2 et supérieures) ne requièrent plus l'utilisation de skeletons. Cette particularité tient au fait que RMI est une architecture distribuée pour le langage java uniquement, il a donc été possible de transférer dans certaines classes du langage (comme `java.rmi.UnicastRemoteObject`) la faculté de recevoir les messages des stubs et d'y répondre directement. Il n'y a donc plus de Skeletons dans la version du protocole RMI pour la plate-forme java 2.

6 Les fonctionnalités avancées de RMI

Comme nous venons de le voir, RMI ne requiert plus de Skeletons dans son protocole de communication. RMI possède également bien d'autres fonctionnalités qui le distinguent des architectures distribuées classiques. Cette section en couvrira quelques unes.

6.1 Distributed garbage collecting (ramasse-miettes distribué)

Toute comme dans le cas d'une application s'exécutant sur une JVM unique, les applications RMI disposent d'un mécanisme qui gère la mémoire et permet de ne plus allouer de ressources à un objet java devenu inutilisable. Ce mécanisme est le Distributed Garbage Collection (DGC). Dans la pratique, les programmeurs d'application RMI n'ont pas à se soucier du fonctionnement du ramasse-miettes distribué, mais il est intéressant de connaître les grands principes de fonctionnement de cet outil de RMI.

Lors de son cycle de vie, un serveur RMI peut créer un certain nombre d'objets et donner aux clients RMI des références permettants d'utiliser ces objets à ditance. Les objets côté-serveur peuvent alors être utilisés de la même manière que des objets java locaux. Lorsque plus aucune JVM distante ne maintient de référence sur cet objet distant, la ramasse-miettes distribué s'active et supprimer les ressources allouées aux objets côté-serveurs. De cette manière ni le client ni le serveur n'ont à se préoccuper de la gestion de la mémoire.

Lorsqu'un client acquiert une référence sur un objet côté-serveur, le client envoie à la JVM serveur un indentifiant de JVM : JVMID. De son côté la JVM cliente reçoit la référence sur l'objet distant ainsi qu'un objet de lease (réservation) associé à l'objet distant. Les leases expirent au bout d'un certain temps, et c'est à la JVM cliente de demander à la JVM serveur de prolonger son lease si elle désire utiliser plus longtemps l'objet côté-serveur. Un client peut également révoquer directement sa réservation auprès de la JVM serveur. Lorsque toutes les réservations sur un objet côté-serveur ont été révoquées ou ont expiré,

le DGC s'active et supprimer l'objet côté-serveur puisqu'aucune JVM ne peut plus l'utiliser.

L'interface `java.rmi.dgc.DGC` définit deux méthodes permettant la gestion des objets côté-serveur :

- `public void clean(ObjID[] ids, long sn, VMID vmid, boolean b) throws RemoteException` Cette méthode permet de supprimer les réservations associées à des objets côté-serveur. Elle peut être invoquée par les JVM clientes pour faciliter le travail du DGC et l'aviser qu'elles n'utilisent plus les objets côté-serveur.
- `public Lease dirty(ObjID[] ids, long sn, Lease lease) throws RemoteException` La méthode dirty permet de demander à une JVM serveur une réservation pour un certain groupe d'objets côté-serveur. Elle permet également de prolonger un lease déjà obtenu afin de maintenir la réservation antérieure.

L'interface `java.rmi.dgc.DGC` est une interface de serveur distant. Elle décrit les services d'un objet (le DGC) qui réside sur la JVM serveur et qui doivent être invoqués depuis une JVM cliente. À ce titre, elle hérite de `java.rmi.Remote` et toutes ses méthodes sont déclarées comme pouvant lancer une exception d'invocation à distance. Tous les paramètres et les types de retour sont des types primitifs ou bien des objets distants ou encore des objets serialisables.

6.2 La sérialisation des objets

Nous avons vu dans la section 3.2.1 que les paramètres et les types de retour d'une méthode de l'interface d'un serveur RMI doivent être de l'un des types suivants : 1) un serveur RMI 2) une donnée de type primitif 3) un objet qui implémente l'interface `java.io.Serializable`. Dans cette section, nous étudierons plus particulièrement la 3ème possibilié : utiliser des objets sérialisés.

La sérialisation est le procédé par lequel un objet sauvegarde son état dans un flux binaire. À l'inverse, la désérialisation est le procédé qui permet de reconstruire cet objet depuis le flux binaire dans lequel il a été stocké. Un objet sérialisé peut donc être stocké dans un fichier ou bien encore transiter sur le réseau par l'intermédiaire d'une socket TCP, voire d'un paquet UDP.

La jdk 1.4 introduit la différence entre sérialisation à court terme (utilisé en RMI par exemple) et la sérialisation à long terme (qui transforme un objet en XML présentement). Cette section ne traite que de la sérialisation à court terme.

L'interface `java.io.Serializable` permet de transformer virtuellement toutes les classes de java en classes sérialisables, i.e. en classes dont les instances peuvent être sérialisées. L'interface `java.io.Serializable` ne contient aucune méthode et aucune constante, il s'agit simplement d'une interface marqueuse qui indique à la JVM qu'elle a été conçue pour la sérialisation. Pour rendre une classe sérialisable, il suffit d'ajouter `implements Serializable` dans sa définition.

Malheureusement, les choses deviennent plus compliquées avec l'héritage, une classe devant assumer de sérialiser également les données membres dont elle hérite de la part de ses classes mères. Un tel contrôle n'est pas toujours possible car il implique que la classe mère a été prévue à cette effet. Il est donc conseillé, dans la mesure du possible, de ne transformer que les classes dérivant directement de `java.lang.Object` en classes sérialisables pour éviter ce genre de désagrément.

Il existe un mécanisme par défaut de sérialisation offert par la JVM, de sorte que si le programmeur l'accepte, tous les champs de l'instance (quelque soit leur niveau de visibilié), à l'exception des champs déclarés `transient`, seront à leur tour sérialisés. S' il s'agit d'une champ de type primitif, alors il est encodé directement ; s'il s'agit d'un objet, le processus de sérialisation s'applique récursivement. Ainsi, les objets agrégés sont transformés en un graphe d'objets sérialisés. Il est important de comprendre, que malgré les apparences, si l'objet est envoyé par sérialisation entre deux JVM distinctes, nous aurons deux instances distinctes de cet objet sur chaque JVM, sauf si le programmeur redéfinit le processus de sérialisation. On peut donc qualifier la désérialisation de "processus extra-linguistique (pour le langage Java) permettant la création d'objets".

Un programmeur peut en effet, choisir de ne pas adopter le processus standard de sérialisation offerts par la JDK. Dans ce cas, il devra redéfinir une des méthodes suivantes : private void readObject, private void writeObject, private void readResolve (jdk 1.4 uniquement). Ces méthodes, bien que privées, seront appelées par la JVM lors du processus de sérialisation et de désérialisation. La documentation javadoc de ces API donnera au lecteur plus de précisions à ce sujet.

Lors de la sérialisation, la JVM "emballe" l'objet dans un certain encodage binaire. En anglais, cette opération est appelée "marshalling", l'opération de déballage de l'objet est appelée "unmarshalling".

La sérialisation est donc un procédé par lequel il est possible de transférer des objets d'une JVM à l'autre. Elle permet non seulement l'échange de structures de données entre les JVM (ce que certains ont appelé la fin des protocoles), mais également le transfert de code entre différentes JVM. La section 6.4

s'intéressera plus particulièrement à cette fonctionnalité de RMI.

6.3 Le téléchargement dynamique des stubs

Si vous avez suivi avec attention la préparation des fichiers de déploiement à la 4.2, vous avez sans doute remarqué que le Stub du serveur RMI n'est pas déployé sur la machine du client RMI. Pourtant, comme l'indique la figure 13, un stub doit bel et bien résider sur la machine du client. Rappelez-vous que les appels à un serveur distant ne sont pas des appels distants mais bien des appels locaux des méthodes du Stub. C'est le Stub qui réalisera véritablement l'appel de méthode vers l'objet distant.

Avec RMI, contrairement aux autres méthodes d'appels distants, il est possible de ne plus déployer le stub du serveur avec le serveur lui-même : le Stub peut être téléchargé, au besoin, à partir d'un serveur web ou ftp durant l'exécution d'un appel distant. Cela signifie que c'est non seulement l'objet Stub mais aussi *la classe* de Stub elle-même qui sera intégrée de manière dynamique à la JVM cliente durant la phase d'exécution.

Lorsqu'un serveur RMI s'inscrit auprès d'une RMI registry, et qu'il désire que les clients RMI téléchargent son Stub, il doit préciser à sa JVM (*la JVM serveur*) l'emplacement à partir duquel les clients RMI pourront trouver le Stub. Le serveur indique cette emplacement grâce à l'option `-Djava.rmi.server.codebase` de la ligne de commande qui lance la JVM. *C'est la responsabilité des serveurs RMI que d'indiquer où leur Stub se trouve. Ce ne sont pas les clients qui indiquent où trouver les Stubs qu'ils désirent télécharger. Ceci est une erreur de compréhension courante, y compris chez les programmeurs chevronnés*.

Le téléchargement dynamique des stubs a un très grand avantage sur un déploiement plus statique : si l'on désire modifier le serveur durant la vie d'un logiciel, il n'est pas nécessaire de modifier les applications clientes ni de les déployer de nouveau. Le téléchargement permet ainsi une meilleure maintenance de l'application. Mais nous allons voir que ce concept peut être poussé plus loin encore dans la section suicante.

6.4 Le téléchargement dynamique de code

Comme nous l'avons vu, RMI permet le téléchargement dynamique des stubs des serveurs RMI sur la JVM du client RMI. Cette opération, entièrement transparente pour le développeur, télécharge du code

exécutable depuis une source externe à la JVM sur laquelle l'application s'exécute.

La sérialisation permet également de télécharger automatiquement du code exécutable entre deux JVM. La logique de calcul contenue dans les méthodes d'une classe sérialisable est ainsi transférée entre deux JVM. Il est ainsi possible de transmettre du code exécutable, de manière dynamique, à une autre JVM.

Le téléchargement de code est étroitement lié à la notion de polymorphisme et ses implications dans un environnement distribué. Supposons, par exemple, qu'un serveur RMI possède une méthode ayant la signature `public TypeRetour method(TypeParam param)`.

Supposons maintenant qu'un client RMI connaisse une classe fille de `TypeParam` et que cette classe fille soit inconnue du serveur RMI. Le client peut, en toute légitimité, invoquer la méthode du serveur en lui passant une instance de la classe fille de TypeParam. Dans ce cas, la JVM du serveur RMI va être "consommatrice" d'un code "produit" par la JVM cliente. La JVM serveur va devoir télécharger du code appartenant à la JVM cliente pour répondre à l'appel de la méthode. Pour cette raison, la JVM cliente (la "productrice") est lancée avec le paramètre `-Djava.rmi.server.codebase`. Ce paramètre indique aux JVMs consommatrices (ici la JVM serveur) où se trouve le code à télécharger.

De manière symétrique, nous pouvons supposer que la JVM peut répondre à l'appel de sa méthode par une classe qui lui seul connaît, qui serait une classe fille de `TypeRetour`. Dans ce cas, la JVM serveur sera "productrice" de code qui sera téléchargé par la JVM "consommatrice" : la JVM cliente. Alors, la JVM serveur devra utiliser l'option `-Djava.rmi.server.codebase` pour indiquer à la JVM cliente où trouver la classe fille de `TypeRetour`.

Lorsqu'une application désire rendre téléchargeable certaines de ses classes, elle doit préciser à la JVM qui va l'exécuter un emplacement à partir duquel le code poura être téléchargé par la JVM distante. Lors du lancement d'une application, l'option `-Djava.rmi.server.codebase` permet de spécifier cet emplacement à Java. Il est bien important de noter que c'est à la jvm "productrice de classes" que le codebase doit être spécifié et non à la JVM qui va les "consommer" (télécharger). Un exemple de ligne de commande pourrait être :

```
java -Djava.rmi.server.codebase
     <votre URL>/deploy-server.jar
     -classpath .
     engine.ComputeEngine!
```

L'option `-Djava.rmi.server.codebase` est suivi d'une URL indiquant l'emplacement de téléchargement. Cet emplacement peut pointer soit vers un répertoire et *dans ce cas, il doit impérativement se terminer par un '/'* soit vers un fichier jar. Dans les deux cas, l'emplacement indiqué doit permettre de trouver les classes à télécharger à distance exactement de la même manière que si l'on indiquait le classpath : il doit posséder la structure de fichiers d'un ensemble de classes Java dans leurs répertoires respectifs. Les URLs supportées par cette option peuvent utiliser les protocoles http, ftp et file. Mais, dans le dernier cas, l'accès aux classes téléchargeables est restreint aux JVM s'exécutant sur le même ordinateur.

Le codebase est donc une sorte de "classpath distant" : il permet à une JVM de charger des classes à distance tout comme le classpath le permet localement.

Le téléchargement dynamique de code ouvre de vastes perspectives comme la réalisation d'agents mobiles capables de se déplacer pour effectuer une tâche ou encore la création de niches "écologiques" d'agents mobiles en passant par la réalisation de tâches distribuées téléchargées dynamiquement par une JVM ou encore la mise à jour automatique d'applications pendant leur exécution (mettre à jour un serveur http sans même l'éteindre, par exemple). . .Mais elle impose en contre-partie des très fortes contraintes au niveau de la sécurité puisque l'on donne l'autorisation à du code "étranger" de s'accaparer des ressources du système telles que des connexions réseaux, des fichiers, etc. Par exemple, le Stub devra ouvrir une socket pour se connecter au serveur RMI.

6.5 La sécurité en RMI

L'architecture de la plate-forme Java 2 est constituée de plusieurs éléments et permet d'obtenir une très fine granularité de contrôle de la sécurité. Elle repose sur différents éléments comme l'authentification par certificats, l'encryption des données, l'utilisation de canaux sécurisés de communication, la personnalisation de protocoles, l'installation de gestionnaires de sécurité ou encore la définition de politiques de sécurité personnalisées.

Nous n'aborderons ici que les gestionnaires de sécurité et les politiques de sécurité. Ces deux éléments permettent, à eux seuls, de définir exactement les droits qui sont accordés à du code selon sa provenance. Ainsi, on pourra donner des droits sur des fichiers à du code local, accorder une connexion réseau à du code téléchargé depuis un emplacement connu, ou encore permettre à du code authentifié par un certificat électronique de faire tout ce qu'il désire.

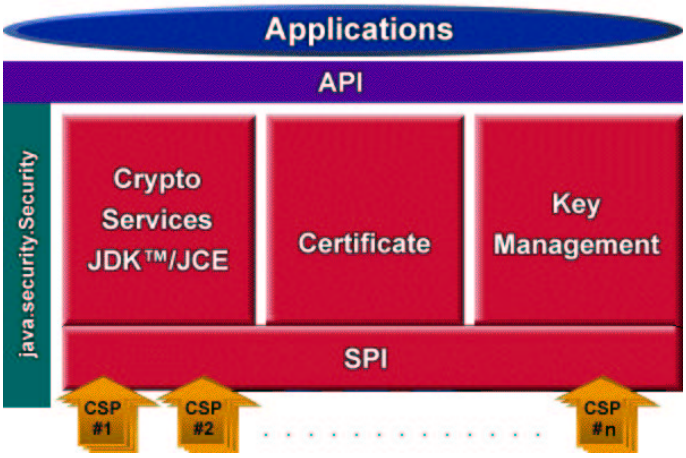

FIG. 15 – Sécurité dans la plateforme java 2.

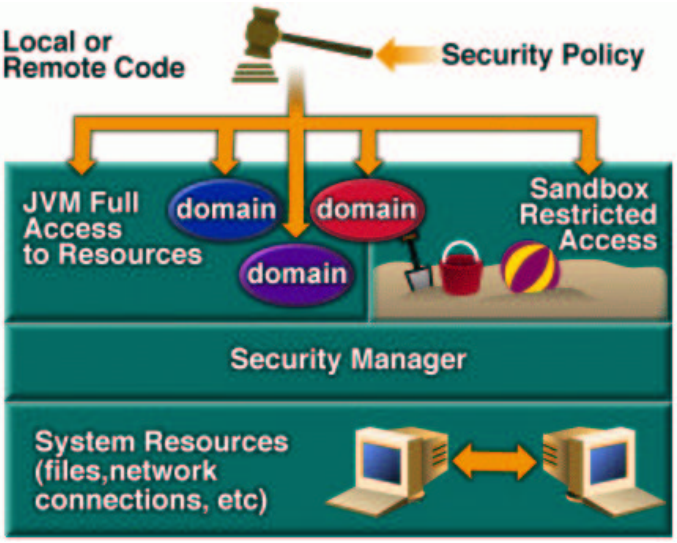

FIG. 16 – Politiques de sécurité et gestionnaire de sécurité dans la la plateforme java 2.

Un gestionnaire de sécurité est représenté par une instance de `java.lang.SecurityManager`. Il est possible de vérifier qu'un gestionnaire de sécurité est installé grâce à la méthode `System.getSecurityManager` et il est possible d'en installer un avec `System.setSecurityManager`. Un gestionnaire de sécurité à pour but de restreindre l'accès aux ressources du système en appliquant une politique de sécurité. Les méthodes de la classe `java.lang.SecurityManager` autorisent ou refusent l'accès à des ressources du système et sont appelées en interne par les librairies de la JVM. Le SecurityManager vérifie alors si la politique de sécurité de la JVM autorise cette action.

Les politiques de sécurité peuvent être paramétrées grâce à des fichiers texte dont l'emplacement doit être indiqué au démarrage de la JVM. Par exemple si on utilise le fichier `/tmp/security.policy`, on lancera la JVM à l'aide de la commande : `java -Djava.security.policy=/tmp/security.policy...`

En RMI, le télechargement dynamique de code (y compris pour les stubs) ne fonctionne pas sans qu'un gestionnaire de sécurité ne soit installé. Ce qui implique d'utiliser un fichier de sécuirté minimal permettant au code situé sur le codebase du serveur la permission de se connecter au serveur RMI. Généralement, on octroie à ce code le droit de se connecter n'importe où sur le port utilisé pour le RMI (1099 par défaut et 1100 pour rmid, cf section 7).

On peut alors utiliser un fichier de politique de sécurité qui ressemble à celui de la figure 17. Un fichier de sécurité a toujours la structure suivante : des blocs de permissions qui sont accordées (granted) à du code provenant d'une source identifiée. Il est possible d'autentifier le code soit par le codebase depuis lequel il est chargé, soit par un certificat, soit par une combinaison des deux méthodes. On peut également omettre la provenance du code étranger mais ce cas présente de sérieuses failles de sécurité puisque n'importe quel emplacement pourra être utilisé pour le téléchargement de code dynamique.

```
grant  {
  permission java.net.SocketPermission "*",
             "accept, connect, listen, resolve";
};
```

FIG. 17 – Fichier de démonstration pour les JVM qui téléchargent du code. Ce fichier ne doit pas êre utilisé dans un environnement de production.(/tmp/security.policy).

Les permissions sont accordées en spécifiant les noms des classes de permisions de la jdk qui sont octroyées au code étranger. Le lecteur trouvera plus d'informations sur les permissions de la jdk dans

la documentation de SUN. Il est possible de générer ces fichiers sans en connaître exactement la sytaxe grâce à l'outil policytool, inclus dans la jdk. Cet outil permet de générer les fichiers de politiques de sécurité avec une interface graphique. Vousd pourrez également trouver de bonnes explications sur les fichiers de sécurité sur le site de SUN [4, 10].

Ainsi, toutes les JVM utilisant le téléchargement dynamique de code doivent exécuter l'instruction de la figure 18.

```
if (System.getSecurityManager() == null)
  System.setSecurityManager(new RMISecurityManager());
```

FIG. 18 – Instruction permettant d'installer un gestionnaire de sécurité.

Étant donné que les applets s'exécutent dans un contexte de sécurité (fourni par le navigateur), il est, en théorie, posssible d'utiliser RMI dans les applets java. Malheureusement, en réalité, tout dépend de la politique de sécurité du navigateur. Seules les navigateurs incluant une Open JVM comme Konqueror [4] et Mozilla[5] peuvent utiliser des applets avec RMI. Les autres imposent de fortes restrictions aux applets, sans doute trop fortes puisque ces restrictions l'empêchent d'établir une sockets pour communiquer avec les machines serveurs RMI ou la RMI registry.

7 L'activation

Comme nous l'avons vu, lorsqu'un serveur RMI est créé, il doit s'enregistrer auprès de la RMI regisrtry. Le service devient alors accessible aussi longtemps que l'objet serveur est en vie. Pour cette raison, nous devons laisser s'exécuter le serveur RMI aussi longtemps que l'on désire l'exécuter.

Ainsi, même si le serveur RMI ne travaille que très rarement, il nous faut malgré tout le maintenir en vie en permanence pour qu'il puisse répondre aux appels des clients RMI, ce qui implique une perte de l'utilisabilité des ressources. De plus, lorsque l'on désire utiliser un grand nombre de serveurs RMI sur une même machine, le coût de maintenir tout ses objets en vie peut être prohibitif et dégrader de façon significative les performances des serveurs.

L'activation est un mécanisme construit au-dessus de RMI qui permet d'activer sur demande des objets serveurs RMI. Cette technique permet de palier aux deux inconvénients que nous venons d'évoquer.

[4] www.konqueror.org
[5] www.mozilla.org

Avec l'activation, il est désormais possible d'activer un objet serveur RMI uniquement lorsque ses services sont requis et de le désactiver au besoin afin qu'il ne consommme pas inutilement de ressources. *Le framework d'activation est ainsi capable de "réveiller" des objets serveurs sur demande. Cette stratégie lui confère un avantage supplémentaire : l'activation peut être utilisée afin de s'assurer que les serveurs RMI seront automatiquement redémarrer en cas de crash du service ou de la JVM qui l'abrite, ou encore en cas de panne de la machine sur laquelle il s'exécute.*

7.1 Les acteurs de l'activation

Le framework de l'activation repose sur la collaboration de 4 entités logicielles ayant des rôles distincts :

- l'objet activable

  C'est l'objet que nous voulons rendre activable, qui devra être réveillé au besoin pour répondre aux invocations de méthodes distantes.
- l'application de démarrage de l'objet activable

  Cette application a une durée de vie très courte, c'est elle qui est responsable d'intégrer l'objet activable dans le framework d'activation.
- le démon d'activation

  le démon d'activation est un petit programme inclus dans la jdk : rmid. C'est lui qui est responsable de déclencher l'activation des objets activables.
- le groupe d'activation

  un groupe d'activation est grossièrement comparable à une JVM. Lorsque rmid active un objet, il crée un groupe d'activation (une nouvelle JVM indépendante) qui va héberger l'objet activé.

Pour qu'un objet activable puisse être intégré au framework d'activation, une application de démarrage doit se charger de créer cet objet et de l'enregistrer auprès du démon d'activation. Étant donné que le démon d'activation devra créer une nouvelle instance de l'objet activable, il est nécessaire que l'application lui fournisse toutes les informations nécessaires à la création d'une nouvelle instance de l'objet activable. L'application de démarrage effectue ces opérations et se termine tout simplement. Il n'est plus nécessaire de maintenir cette JVM.

Lorsque rmid, le démon d'activation, réveille un objet activable, il utilise ces informations et crée une nouvelle JVM indépendante. Cette JVM est représentée, dans le framework d'activation, par un groupe

d'activation. Un groupe d'activation permet de grouper plusieurs objets activables à l'intérieur d'une même JVM. La JVM créée par rmid devra s'exécuter dans un certain environnement (variables d'environnnement) et avec une certaine politique de sécurité. Nous devrons donc spécifier tous ces paramètres au groupe d'activation avant de l'enregistrer auprès de rmid. La figure 19 illustre l'intéraction de ces différentes composantes.

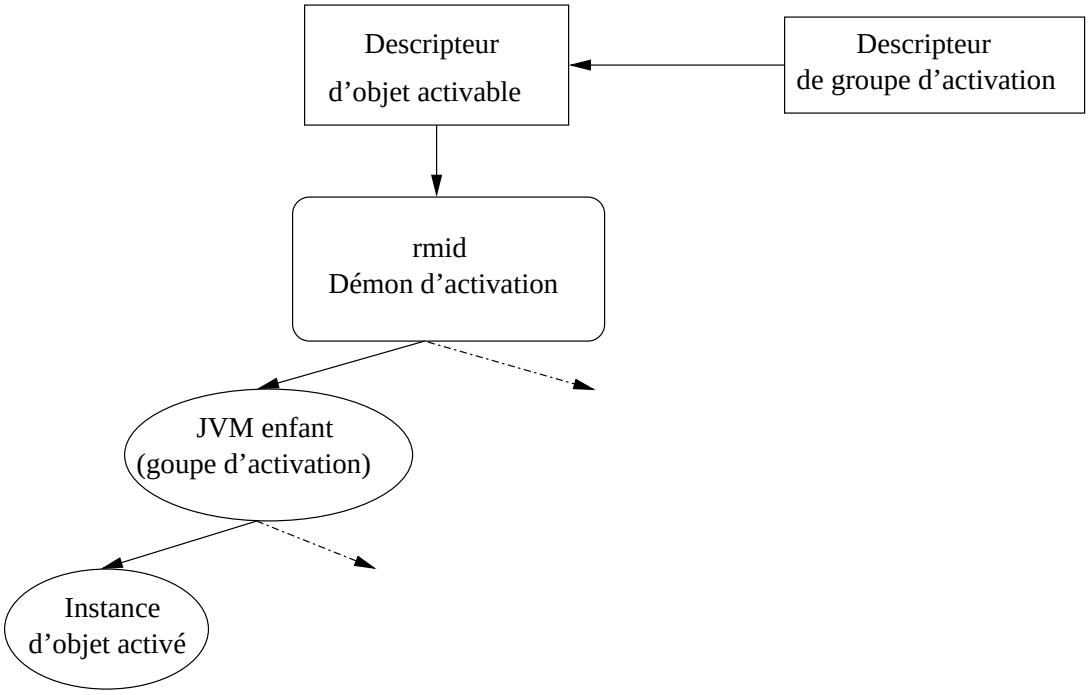

FIG. 19 – Le framework d'activation.

7.2 Création d'un objet activable

Pour créer un objet activable, il suffit de suivre les étapes suivantes :

- s'assurer qu'un démon d'activation est présent sur le système et qu'il fonctionne,
- créer la classe de l'objet Activable,

  Cette tâche peut être décomposée comme suit :
  - faire dériver la classe de l'objet de `java.rmi.Activatable` et étendre une interface distante,
  - donner à cette classe un constructeur admettant 2 paramètres (constructeur utilisé pour activer l'objet),
  - fournir une implémentation des méthodes définies dans l'interface distante comme dans le cas d'un serveur RMI.

– créer l'application de démarrage

  Cette tâche peut être décomposée comme suit :

  – installer un gestionnaire de sécurité,
  – créer un groupe d'activation (`java.rmi.activation.ActivationGroup`)qui contiendra l'information nécessaire à la construction d'une nouvelle JVM qui va héberger l'objet Activable,
  – créer un descripteur d'activation (`java.rmi.activation.ActivationDesc`) qui permettra à l'activationGroup d'instancier convenablement notre objet activable,
  – enregistrer le descripteur d'activation auprès du démon RMI (rmid),
  – l'enregistrement nous renverra un stub de l'objet activable. Ce Stub peut être , par la suite, placé dans une RMI registry ou bien encore être donné à toutes les parties intéressées par les services de notre serveur distant.

Ainsi, la création d'objet activable ne diffère que peu de la création d'objet distants "normaux". Les deux seules différences sont que notre objet ne doit plus étendre `java.rmi.UnicastRemoteObject` mais `java.rmi.Activatable` et qu'il doit posséder un constructeur acceptant deux paramètres : un `java.rmi.activation.ActivationID` et un `java.rmi.MarshalledObject`. C'est ce constructeur qui sera appelé à l'intérieur de la nouvelle JVM pour instancier un nouvel objet activable lorsqu'une requête qui lui est destinée est captée par le framework d'activation.

Si nous voulions convertir notre objet serveur de calcul du premier chapitre en objet activable, nous n'aurions pas beaucoup de transformations à effectuer par rapport au serveur de calcul non activable comme l'indique la figure 20.

Maintenant que nous venons de créer un objet activable, nous devons concevoir une application de démarrage qui sera responsable d'intégrer l'objet activable dans le framework d'activation. Ici, nous avons choisi une solution très simple : c'est la classe de l'objet activable elle-même qui va disposer d' une méthode main et constituer l'application de démarrage. Nous pouvons donc ajouter la méthode de la figure 21 au code de la figure 20

La première étape réalisée par cette application est d'installer un SecurityManager. Le gestionnaire de sécurité permet de mettre en place une politique de sécurité qui autorisera le téléchargement de code dynamique. Tout comme la RMI registry, rmid est un serveur RMI (au sens où ces méthodes sont invo-

```
package engine;
import java.rmi.*;
import java.rmi.activation.*;
import java.rmi.server.*;
import java.util.*;
import compute.*;
public class ComputeEngine
             extends Activatable
             implements Compute
{
  public ComputeEngine(ActivationID ID,
                       MarshalledObect obj)
             throws RemoteException
  {
    super( ID,0 );
  }//cons
  public Object executeTask(Task t)
                   throws RemoteException
  {
    return t.execute();
  }//met
}//class
```

FIG. 20 – Exemple de serveur RMI activable. (/tmp/engine/ComputeEngine.java)

cables à distance), il est donc nécessaire que l'application de démarrage puisse communiquer avec rmid. Rmid écoute les connexions sur le port 1100 par défaut.

Ensuite, l'application construit un descripteur de groupe d'activation. Ce descripteur contient toute l'information nécessaire à rmid pour pouvoir créer une nouvelle JVM qui devra abriter l'objet activé. Ces informations sont essentiellement des paramètres et des propriétés Java qui devront être passés à la future JVM. Ici, ils sont passés sous la forme d'un objet `java.util.Properties`. Ce groupe est par la suite enregistré auprès du système d'activation, lequel nous renvoie une ID identifiant le groupe.

Finalement l'application de démarrage doit construire un descripteur d'activation pour l'objet à activer. Ce descripteur doit contenir les informations suivantes :

- l'ID du groupe dans lequel l'objet devra être activé,
- la classe de l'objet activable,
- l'emplacement à partir duquel les serveurs pourront acéder au code de l'objet (pusique c'est un serveur RMI),
- un `java.rmi.MarshalledObject` qui sera passé au constructeur de l'objet lorsqu'il sera activé.

  Cet objet permet donc de fournir de l'information à toutes les nouvelles instances de l'objet acti-

```
public static void  main(String[] args)
{
  try
  {
    if ( System.getSecurityManager() == null )
      System.setSecurityManager( new RMISecurityManager() );

    //réglage des propriétés pour le démarrage d'une nouvelle
    //JVM
    Properties prop = new Properties();
    prop.put( "java.security.policy",
              "/tmp/security.policy" );
    ActivationGroupDesc group = new ActivationGroupDesc(
                                      prop, null );
    ActivationGroupID gid =
       ActivationGroup.getSystem().registerGroup( group );

    //on donne tous les paramètres nécessaires à la création
    //d'une nouvelle instance de l'objet activable
    String codebase = "<votre URL>/deploy-server.jar";
    MarshalledObject obj = null;
    ActivationDesc desc = new ActivationDesc( gid,
                                    "engine.ComputeEngine",
                                          location, obj );
     Compute engine = (Compute) Activatable.register( desc );

    //on rend publique le stub obtenu dans une registry
    Naming.rebind( "//localhost/serveurDeCalcul, engine" );
  }//try
  catch(Exception ex)
  {
     System.out.println("une exception est survenue durant
                         le démarrage de l'activation");
     ex.printStackTrace();
  }//catch
}//main
```

FIG. 21 – Application de démarrage. (/tmp/engine/ComputeEngine.java)

vable. Il est souvent utilisé pour transmettre des informations persistentes aux différentes instances d'un objet activable pour que cette information survive à un crash de l'objet activé.

En dernier lieu, le descripteur est enregistré dans le framework d'activation lequel retourne un Stub de l'objet activable.

7.3 Compilation et exécution

Le marche à suivre pour la compilation et le déploiement des applications cliente et serveur sont exactement les mêmes qu'à la section 4.

En revanche, l'exécution est légèrement différente. Il vous faut tout d'abord lancer une instance de rmid sur l'ordinateur hôte du serveur (A) :

`rmid &` ou bien `start rmid`

puis lancer la RMI registry :

`rmiregistry &` ou bien `start rmiregistry`

(Ces deux étapes peuvent avoir lieu dans l'ordre inverse).

Ensuite, pour lancer l'application de démarrage, il vous faut taper l'instruction :

```
java  -Djava.security.policy=/tmp/security.policy
       -Djava.rmi.server.codebase=<votre URL>/deploy-server.jar
       -cp /tmp
       engine.ComputeEngine
```

Le fichier de sécurité est utilisé pour avoir accès au stub de rmid. Le codebase, lui, ne sert que dans le cas où vous désirez passer le stub de votre objet activable (obtenu auprès du système d'activation) à une RMI registry comme c'est le cas dans l'exemple.

Et voilà, vous diposez désormais d'un serveur RMI à toute épreuve : il sera réactivé après un crash ou si une panne de machine survient. Pour le relancer, il suffit de réactiver rmid depuis le même répertoire que celui qui a servi à le lancer initialement.

8 Jini, ubiquitous computing (l'informatique ubiquitaire)

Nous avons assisté ces dernières années à deux révolutions comparables aux révolutions industrielles du 20ème siècle. La première a été l'avénement de l'ordinateur et son utilisation massive dans la société civile. La seconde a été l'interconnexion des ordinateurs au niveau mondial : Internet. Il semble qu'une

nouvelle tendance se dessine peu à peu à l'horizon : l'informatique ubiquitaire, l'informatique présente partout et tout le temps. Plusieurs projets visant à intégrer l'informatique dans tous les aspects de notre vie sont actuellement en cours de développement. Le but est simple : rendre l'interconnexion de tous les appareils (ordinateurs, électro-ménager, PDA, vêtements, etc…) aussi simple que la connexion d'un téléphone : il suffit de brancher un appareil pour qu'il puisse interagir instantanément avec n'importe quel autre sur le réseau.

Pour mieux comprendre, prenons un petit exemple. Vous vous promenez à la recherche d'une imprimante dans un centre d'achats. Vous sortez votre PDA et lui demandez de vous indiquer où se trouvent les magasins d'imprimante. Après quelques instants, un trajet menant à chaque magasin apparaît, avec une liste des prix. Vous décidez d'entrer dans un magasin et achetez l'imprimante. De retour chez vous, vos vêtements envoient un message à la chaîne hifi et une douce musique se fait entendre tandis que le chauffage est au bon thermostat, prévenu de votre arrivé dix minutes plus tôt. Vous branchez votre imprimante, sans la moindre configuration, et décidez de prendre une photo. L'appareil photo détecte la présence de la nouvelle imprimante, et lui envoie la photo à imprimer qui sort dans le bon format presque aussitôt. Ce genre de scénario serait aujourd'hui tout à fait envisageable (mais peut-être pas désirable…) du point de vue technologique.

JINI offre le support technologique pour ce genre d'interconnexion. JINI repose sur RMI et l'encapsule totalement, de telle sorte que RMI devient parfois difficilement perceptible. Bien que cela puisse sembler trop avancé, de très nombreuses compagnies ont décidé de s'engouffrer dans cette technologie et un très grand nombre de projet basés sur JINI commencent à voir le jour. Des étudiantEs du département ont d'ailleurs amorcé l'un d'entre eux : le PNC.

8.1 Le projet PNC au département

Les Travaux d'Initiatives Personnelles (TIP) initiés par le département d'informatique permettent aux étudiantEs de se confronter à des projets réalistes. L'un des premiers TIP a été de créer un super ordinateur capable d'utiliser toute la puissance de calcul des ordinateurs du département pour effectuer un même calcul de manière distribuée, massivement parallèle. Vous trouverez ci-dessous un extrait de la charte pédagogique du projet qui s'appuie entièrement sur Java et JINI.

8.2 Objectifs pédagogiques

Le développement du projet est en soit un projet éducatif destiné à permettre aux étudiantEs du département d'informatique de participer à un cycle de développement logiciel complet : de la cueillette des besoins jusqu'à la livraison d'un logiciel fonctionnel en passant par l'analyse, la conception, l'implémentation, le déploiement et la maintenance.

Le développement sera axé sur l'exploration et l'utilisation de nouvelles technologies afin d'y familiariser les étudiantEs impliquéEs. Il vise à leur permettre d'acquérir une expertise pertinente avant d'entrer dans le monde professionel et de les initier aux technologies et à la philosophie du modèle de développement Open Source.

8.3 Description du projet

Le projet de noeud de calcul (PNC) vise à concevoir et déployer un système informatique distribué permettant de répartir des tâches de calcul sur plusieurs ordinateurs reposant sur la technologie JINI. Le système doit souscrire aux impératifs suivants :

1. généraliste : les tâches pouvant être exécutées par le système pourront être de natures diverses. Nous ne cherchons pas à effectuer une seule et unique tâche (comme seti@home par exemple) mais à réaliser n'importe quelle tâche,
2. dynamique : le nombre d'ordinateurs participant au noeud de calcul doit pouvoir évoluer dans le temps. De même, la nature des tâches pourra également varier dans le temps sans qu'il ne soit nécessaire de recompiler et/ou redéployer le noeud de calcul,
3. fédératif : lorsqu'un nouvel ordinateur est disponible, il devra pouvoir rejoindre la fédération des ordinateurs oeuvrant dans le noeud de calcul sans que la moindre information ne soit codée "en dur" dans aucun ordinateur,
4. ergonomique : le système devra proposer une interface utilisateur permettant de contrôler l'activité des ordinateurs participant au noeud de calcul et l'évolution du calcul,
5. robuste : le système doit être résistant aux pannes et continuer de fonctionner malgré la perte de certaines composantes distribuées,
6. maintenable et modulaire : le système devrait être facilement maintenable et son architecture doit permettre des évolutions importantes. Il devra nécessiter le plus petit effort possible de maintenance

et de déploiement. Ce critère renvoie évidemment à la qualité de la documentation que devra fournir l'équipe de développement.

A l'heure actuelle ce projet est arrivé à maturité, une première version officielle devrait être disponible vers le mois de mai 2002. Vous pouvez trouvez plus de renseignements sur le site web du projet : http://s-java.ift.ulaval.ca/\~pnc.

9 Annexe A

La figure22 propose une tâche capable de calculer avec une grande précision la valeur de π. Cette tâche est un peu complexe mais la complexité ne vient pas de RMI mais de la précision du calcul de π. Si vous utilisez cette tâche dans l'application cliente `compute.ComputePi`, n'oubliez pas de donner un paramètre à la ligne de commande lançan l'application. Ce paramètre devra être un entier indiquant le nombre de décimales de la valeur de π.

```
package client;

import compute.*;
import java.math.*;

public class TaskPi implements Task
{
  private static final BigDecimal ZERO = BigDecimal.valueOf(0);
  private static final BigDecimal  ONE = BigDecimal.valueOf(1);
  private static final BigDecimal FOUR = BigDecimal.valueOf(4);
  private static final int roundingMode =
                                    BigDecimal.ROUND_HALF_EVEN;
  private int digits;

  /** Construit une tâche pour calculer pi avec une précision
      variable. */
  public TaskPi(int digits)
  {
    this.digits = digits;
  }//cons
```

```
 /**
  * La valeur est calulée par la formule de Machin :
  * pi/4 = 4*arctan(1/5) - arctan(1/239)
  */
  public Object execute()
  {
    BigDecimal arctan1_5 =
      arctan(5, digits + 5).multiply(FOUR);
    BigDecimal arctan1_239 = arctan(239, digits +5 );
    BigDecimal pi =
       arctan1_5.subtract( arctan1_239).multiply(FOUR);
    return pi.setScale(digits,
                       BigDecimal.ROUND_HALF_UP);
  }//met
  /**
   * Calcul de Arctan en BigDecimal. Retourne une valeur
   * en radian de l'inverse de l'arctan du premier paramètre.
   * Le second paramètre fixe la précision du calcul. La
   * valeur est obtenue par la formule :
   * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7...
   */
  public static BigDecimal arctan(int inverseX, int scale)
  {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
    numer = ONE.divide(invX, scale, roundingMode);
    result = numer;
    int i = 1;
    do
    {
      numer = numer.divide(invX2, scale, roundingMode);
      int denom = 2 * i + 1;
      term = numer.divide(BigDecimal.valueOf(denom), scale,
                                                  roundingMode);
      if ((i % 2) != 0)
        result = result.subtract(term);
      else
        result = result.add(term);
      i++;
    }//do
    while (term.compareTo(ZERO) != 0);
    return result;
  }//met
}//class TaskPi
```

FIG. 22 – Une tâche calculant la valeur de π.(/tmp/client/TaskPi.java)

Références

[1] L'équipe du pnc. Site web du pnc.
http://s-java.ift.ulaval.ca/~pnc.

[2] JGuru. Fundamentals of RMI.
http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html.

[3] SUN. Creating a Custom RMI Socket Factory.
http://java.sun.com/j2se/1.4/docs/guide/rmi/rmisocketfactory.doc.html.

[4] SUN. Default policy implementation and policy file syntax.
http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html.

[5] SUN. Dynamic code downloading using RMI.
http://java.sun.com/j2se/1.4/docs/guide/rmi/codebase.html.

[6] SUN. Getting started with RMI.
http://java.sun.com/j2se/1.4/docs/guide/rmi/getstart.doc.html.

[7] SUN. $java^{TM}$ remote method invocation (RMI).
http://java.sun.com/products/jdk/rmi/index.html.

[8] SUN. Jini home page.
www.jini.org.

[9] SUN. An overview of RMI Applications.
http://java.sun.com/docs/books/tutorial/rmi/overview.html.

[10] SUN. Permissions in the javatm 2 sdk.
http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html.

[11] SUN. Remote Object Activation.
http://java.sun.com/j2se/1.4/docs/guide/rmi/activation.html.

[12] SUN. RMI 1.4 specifications.
http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html.

[13] SUN. RMI tutorial : a distributed compute engine.
http://java.sun.com/docs/books/tutorial/rmi/.

[14] SUN. Security in java 2 sdk 1.2.
http://java.sun.com/docs/books/tutorial/security1.2/index.html.

Table des matières

Table des figures