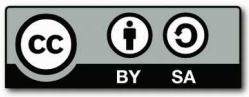
Remote Method Invocation
Invocation de méthodes distantes (RMI)



Résumé
Ce document résume la technologie RMI, la programmation distribuée et donne une connaissance à la fois pratique et théorique du modèle de communication de l'API RMI.
Ceci permet de s'initier à l'informatique distribuée au travers de la mise en œuvre d'un système général de calcul distribué généraliste et à chargement dynamique de tâches.

1 Introduction

L'API Remote Method Invocation (RMI) permet la communication entre des objets java exécutés sur des Machine Virtuelle Java (JVM) différentes ; ces JVMs pouvant être situées sur des machines (serveur ou client) distantes l'une de l'autre. Cette API est incluse par défaut dans le jdk et repose sur le package java.rmi et ses sous-packages.
Vous trouverez une documentation de l'API rmi dans la documentation maintenu par Oracle à l'adresse suivante pour le JDK 8 :

http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html
http://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmiTOC.html

Et tous les liens qui vont bien pour accéder à l'ensemble de la documentation attenante.

RMI est un modèle de communication permettant l'interopérabilité entre plusieurs JVM. Il existe de nombreuses autres techniques permettant de réaliser une certaine forme d'interopérabilité entre des systèmes distants tels CORBA, D-BUS, WEB-WS, SOAP, REST, RPC, DCOM, . . .RMI possède quelques spécificités qui la distinguent de ces systèmes :

1) Avec RMI, l'interopérabilité est limitée aux seul langage java (sauf avec IIOP).
2) RMI est fondamentalement orienté objet.
3) RMI ne requiert pas l'utilisation d'un langage de description des contrats clients-serveurs comme le demande CORBA.
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 ci dessous, 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.
Avec RMI, les services sont réalisés par l’invocation de méthodes du serveur RMI.

De manière minimaliste, 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 ses méthodes visibles aux clients RMI. La nouvelle entrée de l’annuaire associera un nom au serveur 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.

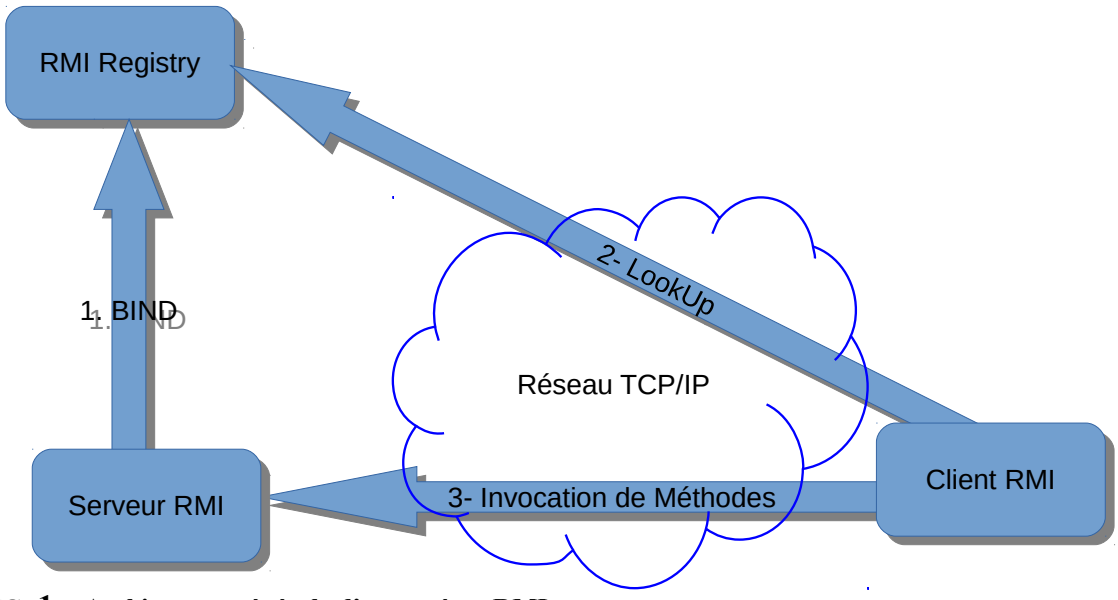
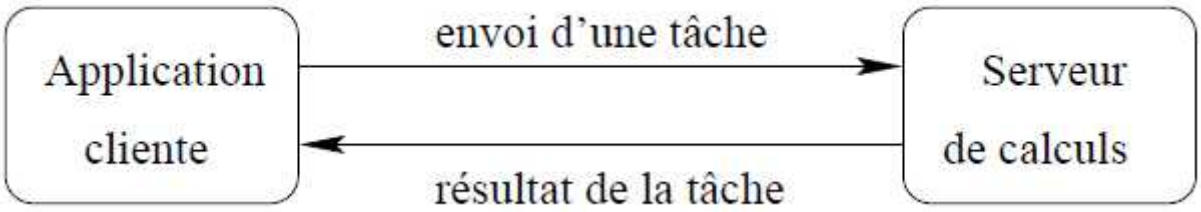


FIG. 1 – Architecture générale d’un système RMI.

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.
Je vais créer un serveur RMI qui offre une méthode (executeTask), cette méthode permettra de réaliser n’importe quelle tâche de calcul. Côté client, je créerai une tâche de calcul et construirai une application cliente qui invoquera, à distance, la méthode executeTask en passant en paramètre la tâche que j’ai 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 vue schématisée de l'application.

3.1 Configuration pour le tutoriel

Pour réaliser ce tutoriel, vous devez disposer d'une version 8 ou supérieure du JDK. Ce JDK doit être fonctionnel, c'est à dire que vous devez pouvoir appelez "java -version" dans une console sans voir d'erreur, ainsi que « javac -version ».
J'ai réalisé le tutoriel en plaçant tous les fichiers dans le répertoire */tmp* (sous UNIX) ou *c:/temp* (sous Windows). Ce répertoire devra contenir 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 exemple, 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 je l'ai 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 susceptible d'offrir. En réalité, le client ne connaîtra jamais véritablement le serveur durant la réalisation du service, il n'interagit pas directement avec un serveur RMI, il n'interagit qu'avec ce contrat, avec l'interface du serveur tel qu'illustré à la figure 3.

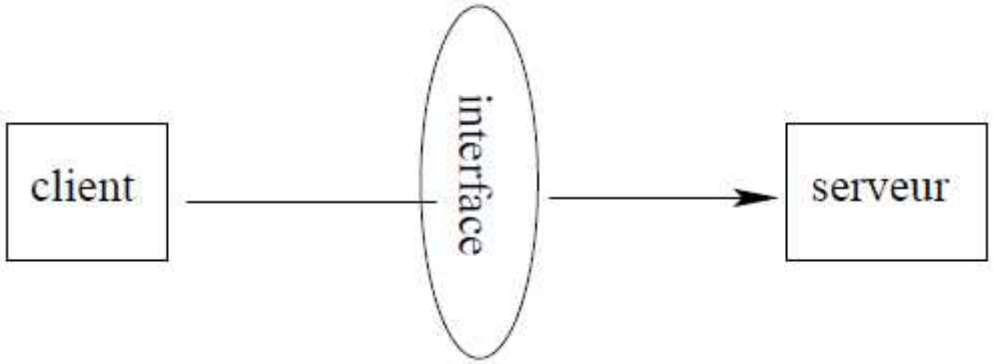


FIG. 3 – Le client interagit avec le serveur distant via une interface qui décrit les services disponibles.

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 (GUI)). 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 code source ci dessous 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 j'ai 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 exception « *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.

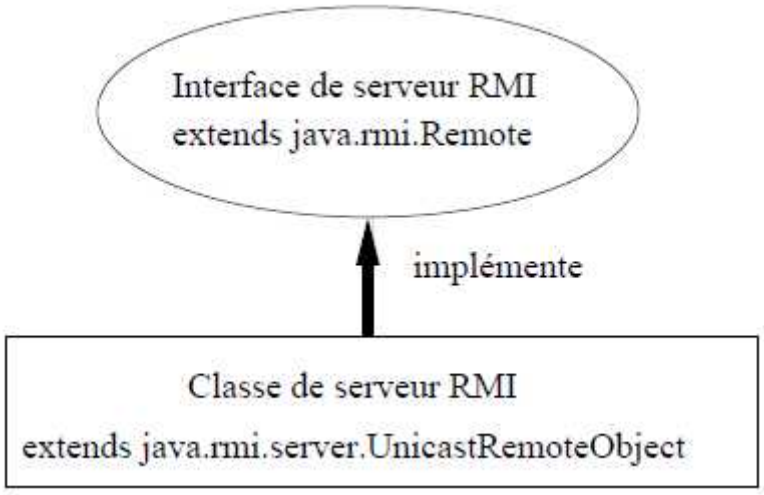


FIG. 6 – L'héritage dans la création d'un serveur RMI.

Afin d'implémenter l'interface de serveur RMI, les classes de serveur sont tenues de fournir une implémentation 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 j'ai 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* ». J'ai choisi de conserver le niveau d'exception défini dans l'interface pour indiquer leur caractère "distant".

Je suis également obligés de définir un constructeur sans paramètre pour cette classe. En effet, la classe « *java.rmi.server.UnicastRemoteObject* » définit un constructeur sans paramètre pouvant lancer une « *java.rmi.RemoteException* ». Comme nous le savons, si je ne défini 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 puisqu'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 :
Je délégue 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 je l'ai mentionné dans la section 2, un serveur RMI doit être inscrit auprès du service d'annuaire de la RMI registry.
Ici, j'ai 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 susceptible de lancer des exceptions de type « *MalformedURLException* », « *RemoteException* » ou « *AccessException* », c'est la raison pour laquelle je l'ai placé dans un bloc try/catch (ici les exceptions sont 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 défaut1099) 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 je désire placer 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
}
```

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

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 échouer 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 j'aurais 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* ». Je reviendrai, 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 du serveur RMI que j'ai 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 j'ai utilisée à la figure 8 pour inscrire le serveur dans l'annuaire.

Notez bien que je ne peu plus utiliser une URL avec l'hôte "localhost", car je suppose que le client et le serveur RMI s'exécutent sur des machines différentes. Il me 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 j'avais converti le résultat de la méthode lookup en « *engine.ComputeEngine* », j'aurais 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 renvoyée 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 me reste plus qu'à l'utiliser exactement comme je le ferais 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 susceptibles 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, j'ai choisi d'intercepter ces exceptions indifféremment et j'ais uniquement écrit un bloc 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 permettre de se concentrer sur RMI. Vous trouverez en annexe (section 9) une tâche plus consistante capable de renvoyer la valeur de retour 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.

```
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
   }//class
```

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

4.1 Compilation

Désormais, j'ais 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 classes 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 documents 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).
Je vous présente, 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.

Je vous présenterai 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 déploiement 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.
Je vous conseille 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 tutoriel, pas les trois en même temps !
Maintenant que les fichiers sont compilés, je vais 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, je vous propose un mode de déploiement très simple. Ce mode de déploiement 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

Je suppose 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, je dépose les fichiers .class du serveur soient les packages « *compute* » et « *engine* » mais pas celles du package client. Sur l'ordinateur B, je dépose 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 précédemment dans le répertoire « */tmp/web* ». Assurez vous 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

Je suppose 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, je dépose les fichiers .class du serveur soient les packages « *compute* » et « *engine* » mais pas celles du package client. Sur l'ordinateur B, je dépose 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 <votreURL>/deployserver.jar.

Les URLs seront de la forme :
www.monserveur/monRepertoire/deploy-client.jar et www.monserveur/monRepertoire/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]

(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 jusqu'à la fin de l'exécution du client, ne la fermez pas.

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.

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. . .

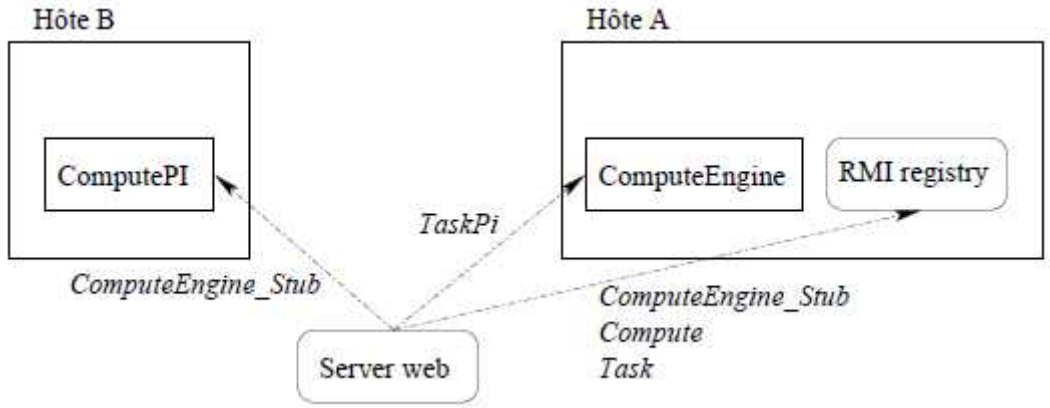


FIG. 11 – Schéma de déploiement de l'application exemple.

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.

Cette fois, c'est le serveur RMI qui ne connaît 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 avez 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 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, 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 .…
```

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.