On peut aussi utiliser une URL vers un fichier:

file:/myDirectory/location

La technique n'est pas utilisable uniquement pour télécharger les stub, mais également pour récupérer le code (.class) de paramètres polymorphes.

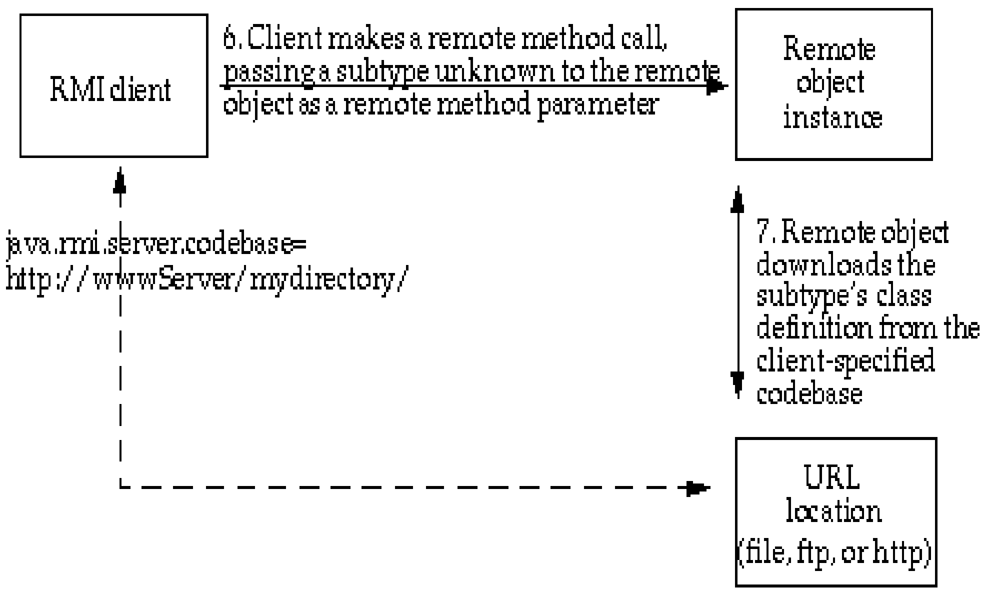

FIGURE 17 Téléchargement due au polymorphisme

Attention:

“URL location (http serveur, etc.)” devrait être à gauche: côté client dans ce cas !

Ces techniques de polymorphisme et chargement dynamique permettent de faire des :

infrastructures client-serveurs complètement générique.

Voir par exemple:

http://java.sun.com/docs/books/tutorial/rmi/

"Building a Generic Compute Engine"

qui donne un tel exemple, et suite du cours, section 4.3

"Exemple RMI avec polymorphisme"

Notes:

- **codebase** est une sorte de pointeur vers du code, c'est une sorte d'annotation qui est ajouté à un objet

- les objets sérialisés et le code prennent des chemins différents :

objets: sockets (TCP/IP)

.class: HTTP (au dessus de TCP/IP)

Voir sur dessin au tableau ...

4.1.6 CLASSPATH et rmiregistry

Avant de lancer le rmiregistry, on doit s'assurer de ne pas avoir de CLASSPATH qui permet de télécharger des classes que l'on veut télécharger automatiquement vers le client

En particulier le stub pour l'objet distant.

Si rmiregistry peut accéder à des classes par le CLASSPATH, cela va être prioritaire sur le codebase,

et un client ne pourra pas télécharger le stub, ou autres classes.

Voir:

java.rmi.server.codebase property

Le plus sure :

unset CLASSPATH

avant de lancer le rmiregistry

Notes:

Tous les objets accessible à distance sont créés et enregistrés en local.

Jini permettra de lever cette contrainte
+ autres fonctionnalités:

- lookup sur ne interface
- broadcast afin de trouver un serveur sans connaitre la machine où il se trouve
- notion de “lease” : proxy à duré de vie finie, et pré-définie

4.2 Applets et RMI

4.2.1 Principes

Depuis une applet, on va faire un appel RMI vers un serveur qui se trouve sur le host depuis lequel on a chargé l'applet.

Le résultat de l'appel est retourné dans le browser, et affiché par l'applet.

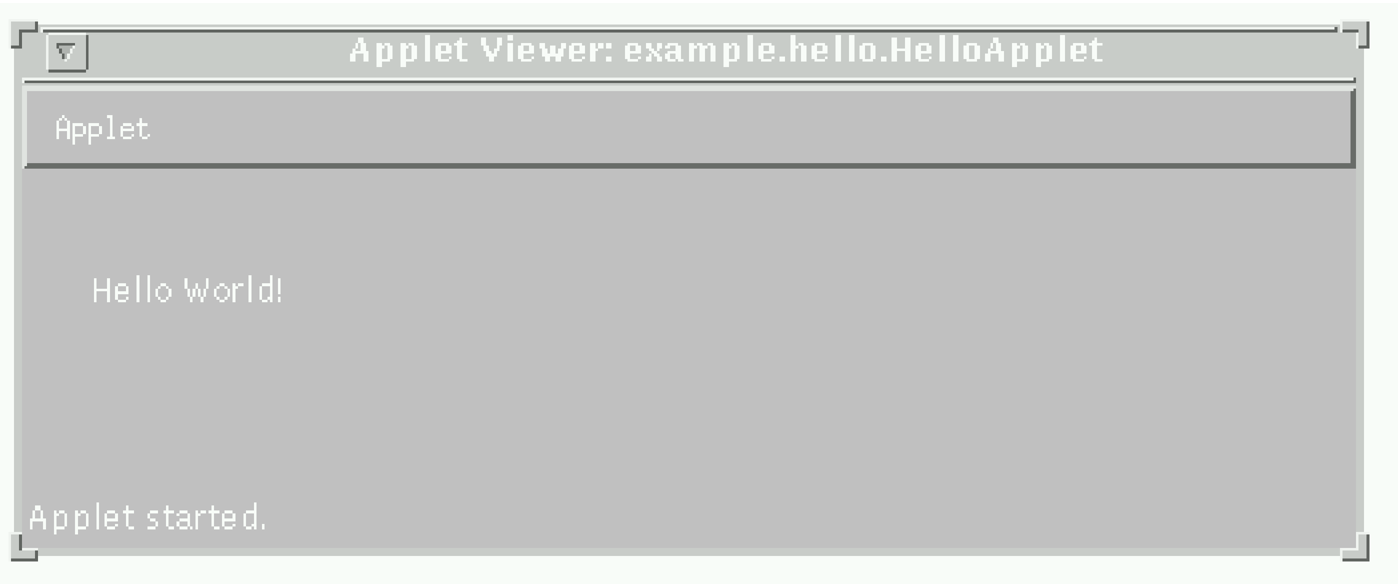

FIGURE 18 Exécution de l'applet Hello World

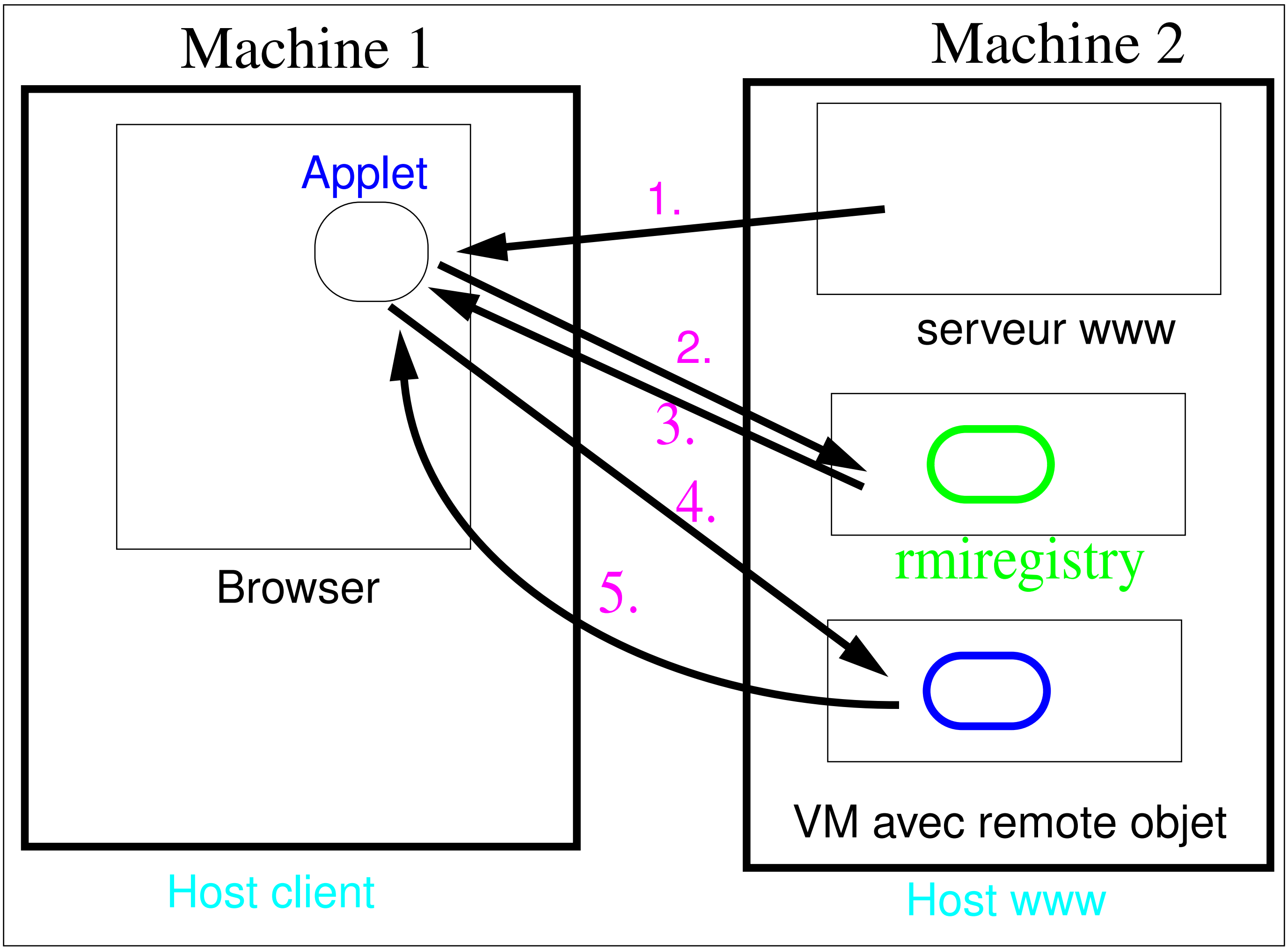

FIGURE 19 Architecture d'un RPC avec une applet RMI

1. Chargement de l'applet
2. Lookup dans le rmiregistry
3. retour du stub
4. Appel à l'objet distant
5. Retour du résultat

4.2.2 Code et principes des classes

Attention à l'utilisation des packages:

en Java mapping entre

le nom complet du package d'une classe

et

le chemin pour touver cette classe

Cela permet au compilateur de trouver les classes

Dans l'exemple:

- package examples.hello
- sources dans $HOME/mysrc/examples/hello

^ répertoire ^ Package ^

de base

a) Remote interface Hello

Remote interface classique, pas de changement par rapport à l'exemple précédent.

```
package examples.hello;

   import java.rmi.Remote;
   import java.rmi.RemoteException;

   public interface Hello extends Remote {
      String sayHello() throws RemoteException;
   }
```

b) Serveur classe (HelloImpl) et main

```
package examples.hello;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject
                           implements Hello {

   public HelloImpl() throws RemoteException {
      super();
   }
public String sayHello() {
      return  "Hello World!";
   }
public static void main(String args[]) {
// Create and install a security manager
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         HelloImpl obj = new HelloImpl();
         // Bind this object instance to the name "HelloServer"
         Naming.rebind("HelloServer", obj); // eventuellement clio.unice.fr:2001
         System.out.println("HelloServer bound in registry");
      } catch (Exception e) {
         System.out.println("HelloImpl err: " + e.getMessage());
         e.printStackTrace();
} } }
```

UnicastRemoteObject et Hello

Implémentation des routines

Un security manager est nécessaire

Instanciation de l'objet distant

Enregistrement dans le rmiregistry

c) Applet cliente: HelloApplet

```
package examples.hello;

import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {

   String message = "blank";

   // "obj" is the identifier that we'll use to refer
   // to the remote object that implements the "Hello"
   // interface
   Hello obj = null;

   public void init() {
      try {
         obj = (Hello)Naming.lookup("//" +
                  getCodeBase().getHost() + "/HelloServer");
         message = obj.sayHello();
      } catch (Exception e) {
         System.out.println("HelloApplet exception: " +
                           e.getMessage());
         e.printStackTrace();
     } }
public void paint(Graphics g) {
      g.drawString(message, 25, 50);
} }
```

getCodeBase().getHost() pour touver le host

pas de securityManager: on a celui de l'applet

Le lookup envoie le stub, et le code du stub va être également envoyé vers le client (serveur HTTP).

d) Page HTML: hello.html

```
<HTML>
<title>Hello World</title>
<center> <h1>Hello World</h1> </center>

The message from the HelloServer is:
<p>
<applet codebase="myclasses/"
     code="examples.hello.HelloApplet"
     width=500 height=120>
</applet>
</HTML>
```

Notes:

- Il faut un serveur HTTP sur la machine depuis laquelle on veut récupérer (downloader) les classes:

    - l'applet tout d'abord,

    - mais également le stub vers le Rem. Object

- codebase est le répertoire où se trouve les .class de l'applet. Il spécifie un répertoire sous celui utilisé pour charger l'applet:

chemin relatif

ce qui est en général une bonne solution (utiliser ../ si le codebase est au dessus du répertoire HTML)

- L'attribut code de l'applet spécifie le nom complet du package: examples.hello.HelloApplet

4.2.3 Compilation et déploiement

Attention, quelques points spécifiques

a) Placement des fichiers, serveurs HTTP

Rappel des fichiers sources:

- Hello.java
- HelloImpl.java
- HelloApplet.java
- hello.html

Compilation en spécifiant où les .class doivent être placés:

ici $HOME/public_html/myclasses

Souvent les serveurs web permettent l'accès au répertoires des utilisateurs par:

http://host/~username/ , *et*

public_html et égal à www

Sinon, et dans le cas où le host de l'applet cliente et l'host du serveur ont accès au même système de fichiers, alors on peut utiliser une URL de type file:

file:/home/username/public_html

Alternative: serveur HTTP minimal fourni par Sun:

http://java.sun.com/products/jdk/rmi/class-server.zip

Voir utilisation en TD.

b) Compilation

Supposons que l'on place les .class dans

$HOME/public_html/myclasses

alors on va compiler de la façon suivante:

javac -d $HOME/public_html/myclasses examples/hello/Hello.java examples/hello/HelloImpl.java examples/hello/HelloApplet.java

Cette commande:

- crée le répertoire examples/hello (s'il n'existe pas déjà) dans le répertoire $HOME/public_html/myclasses
- et y met les .class

Il faut que le répertoire myclasses existe

- Compile et y crée les 3 fichiers:

Hello.class HelloImpl.class
HelloApplet.class

Puis, génération des stub et skeleton:

rmic -d $HOME/public_html/myclasses examples.hello.HelloImpl

La commande est exécutée sur la classe qui implémente des objets distants (HelloImpl) et va générer:

HelloImpl_Stub.class et HelloImpl_Skel.clas

dans le même répertoire que les autres .class

c) PLacement de l'applet et CLASSPATH

Mettre le fichier HTML sous le répertoire accessible depuis le browser:

mv $HOME/mysrc/examples/hello/hello.html $HOME/public_html/

d) RMI registry, serveur et applet

Lancer le RMI registry:

Pas de CLASSPATH

unset CLASSPATH
rmiregistry & .... *ou* ...... **rmiregistry 2001 &**

car on veut que le client qui va utiliser le rmiregistry download les .class par le serveur HTTP

Si le rmiregistry trouve dans son CLASSPATH les .class, alors il va ignorer la propriété codebase du serveur qui lui indique comment trouver les classes, et le client ne marchera pas.

Bug ou feature ???????

Lancer le serveur:

Pour lancer le serveur (HelloImpl), il faut que le répertoire $HOME/public_html/myclasses soit dans le CLASSPATH.

C'est la propriété Java

java.rmi.server.codebase

qui va permettre de spécifier comment transmettre le stub du host du serveur vers le client

Tout d'abord du serveur vers le rmiregistry:

lorsque le rmiregistry a besoin du stub, il ne le trouve pas en local, car PATH à vide, il va donc chercher dans le codebase fourni:

par le serveur http,

cette source de charchement est mémorisée dans l'objet, plus tard le client utilisera le même moyen pour charger le .class du stub reçu).

----> Dessin au tableau

```
java
    -Djava.rmi.server.codebase=http://myhost/~myusrname/
myclasses/
     -Djava.security.policy=$HOME/mysrc/policy
     examples.hello.HelloImpl
```

Rappel : le denier / du codebase est indispensable!

On spécifie également un policy file spécifique: $HOME/mysrc/policy

Normalement: "HelloServer bound in registry"

e) Exécution de l'applet

http://myhost/~myusrname/hello.html

Depuis un browser ou l'applet viewer:

appletviewer http://myhost/~myusrname/hello.html

4.3 Polymorphisme et RMI

Mécanisme puissant pour la programmation répartie

Permet de faire, par exemple, du client-serveur générique, ou encore de gérer l'évolution des logiciels sans avoir besoin d'arrêter des systèmes complexes.

4.3.1 Principe et implications

On envoie à distance un sous-type du type attendu.

Technique classique du polymorphisme.

Ici, en plus, on gère dynamiquement le fait que l'objet que l'on envoie n'est pas forcément connu dans le processus distant où il arrive:

> l'objet n'est pas connu =
> on ne dispose pas du byte code de sa classe

Comme pour les stub de l'exemple précédent, il va faloir faire du chargement dynamique de code d'une VM à l'autre.

4.3.2 Retour de résultat

Du Serveur vers le client:

Le client appelle une routine du type:

Result1 res = serveur.foo (...)

le serveur retourne un objet de type Result2 qui dérive de Result1

Le client devra downloader le code de Result2 afin de pouvoir se servir (appeller les routines, avoir accès aux attributs) de l'objet reçu en résultat de son appel distant.

Note:

Le client devra également downloader toutes les classes (ou interfaces) utilisées par Result2:

ses attributs,
les paramètres de ses routines,
etc.

4.3.3 Passage de paramètres

Du Client vers le Serveur:

Le client appelle une routine du type:

Result1 res = serveur.foo (ParmA1, ParamB1)

mais ne passe pas en paramètres des objets de type ParmA1, ParamB, mais de type ParmA2, ParamB2 qui sont des classes qui en dérivent.

Le serveur devra downloader le code de ParmA2, ParamB2 afin de pouvoir se servir (appeller les routines, avoir accès aux attributs) des objets reçu en paramètre pour exécuter l'appel de foo

Note:

Le serveur devra également downloader toutes les classes (ou interfaces) utilisées par ses deux nouvelles classes:

ses attributs,
les paramètres de ses routines,
etc.

4.4 Exemple RMI avec polymorphisme

4.4.1 Principe des 2 interfaces

Un serveur générique:

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

Une tâche à exécuter:

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

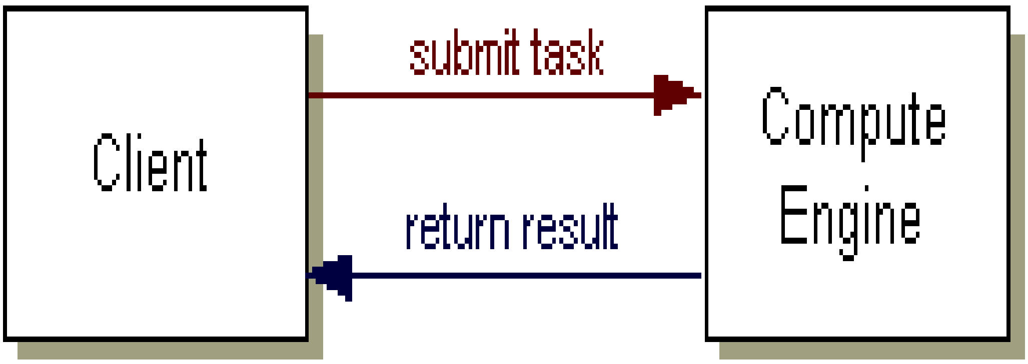

FIGURE 20 Client-Serveur générique

Un objet représente la tâche à exécuter :

réification classique en program. à objets

Task est Interface mais pourrait être une Classe

4.4.2 Implémentation du serveur

Le serveur ne fait qu'exécuter les tâches qu'on lui soumet:

executeTask appelle execute sur les objets de type Task qu'on lui transmet.

```
package engine;

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

    public class ComputeEngine extends UnicastRemoteObject
                          implements Compute
    {
        public ComputeEngine() throws RemoteException {
            super();
        }

        public Object executeTask(Task t) {
            return t.execute(); // Ici: POLYMORPHISME, ST vs. DT
        }

        public static void main(String[] args) {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            String name = "//host/Compute";
            try {
                Compute engine = new ComputeEngine(); // Hidden Multi-Threading
                Naming.rebind(name, engine);
                System.out.println("ComputeEngine bound");
            } catch (Exception e) {
                System.err.println("ComputeEngine exception: " +
                                e.getMessage());
                e.printStackTrace();
            }
        }
    }
```

4.4.3 Un client: compute Pi

Principe classique de connexion au serveur:

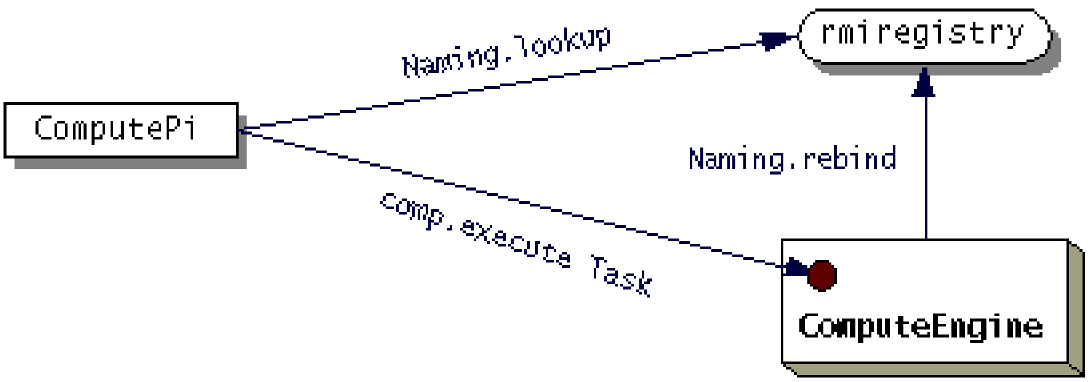

FIGURE 21 Un client : compute Pi

Puis on crée en local (client) un objet du type de la tâche que l'on souhaite exécuter, on le passe au serveur, puis on récupère le résultat.

```
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 {
                String name = "//" + args[0] + "/Compute";
                Compute comp = (Compute) Naming.lookup(name);
                Pi task = new Pi(Integer.parseInt(args[1]));
                BigDecimal pi = (BigDecimal) (comp.executeTask(task));
                System.out.println(pi);
            } catch (Exception e) {
                System.err.println("ComputePi exception: " +
                                e.getMessage());
                e.printStackTrace();
            }
        }
    }
```

L'implémentation de la classe Pi est assez classique, elle doit juste implémenter deux choses:

- Task
- être Serializable (les attributs, car Task l'est déjà))

On doit y définir la routine execute qui représente le travail à réaliser.

```
package client;
import compute.*;
    import java.math.*;

    public class Pi implements Task {

      /** constants used in pi computation */
      private static final BigDecimal ZERO =
        BigDecimal.valueOf(0);
      private static final BigDecimal  ONE =
        BigDecimal.valueOf(1);
      private static final BigDecimal FOUR =
        BigDecimal.valueOf(4);

      /** rounding mode to use during pi computation */
      private static final int roundingMode =
        BigDecimal.ROUND_HALF_EVEN;

      /** digits of precision after the decimal point */
      private int digits;

      /**
       * Construct a task to calculate pi to the specified
       * precision.
       */
      public Pi(int digits) {
        this.digits = digits;
      }
```

```
 /**
 * Calculate pi.
 */
                    /* Code développé par le client,
                     * mais qui va être exécuté par le serveur ! */
public Object execute() {
  return computePi(digits);
}

/**
 * Compute the value of pi to the specified number of
 * digits after the decimal point.  The value is
 * computed using Machin's formula:
 *
 *        pi/4 = 4*arctan(1/5) - arctan(1/239)
 *
 * and a power series expansion of arctan(x) to
 * sufficient precision.
 */
public static BigDecimal computePi(int digits) {
   int scale = digits + 5;
   BigDecimal arctan1_5 = arctan(5, scale);
   BigDecimal arctan1_239 = arctan(239, scale);
   BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                          arctan1_239).multiply(FOUR);
   return pi.setScale(digits,
                    BigDecimal.ROUND_HALF_UP);
}
/**
 * Compute the value, in radians, of the arctangent of
 * the inverse of the supplied integer to the speficied
 * number of digits after the decimal point.  The value
 * is computed using the power series expansion for the
 * arc tangent:
 *
 * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
 *     (x^9)/9 ...
 */
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++;
        } while (term.compareTo(ZERO) != 0);
        return result;
    }
}
```

Code source et tutorial détaillé (en anglais) à :

http://java.sun.com/docs/books/tutorial/rmi/index.html

Dessin tableau :

JVM A.	JVM B.	JVM C.
Serv. HTTP	1st. download	2 nd download depuis le serveur de le JVM A.

4.5 Callback, synchro, multithread, et RMI

4.5.1 Model

Que se passe-t-il si N clients font des appels RMI sur un serveur ?

Ce n'est pas précisé dans la sémantique de RMI

Au moins un thread, mais implicite.

On ne sait pas si l'on va avoir plusieurs threads, ou si les appels sont sérialisés.

Si il y a un seul thread, il y a forcément une mise en attente si plusieurs appels arrivent de façon proche.

Mais on ne sait pas vraiment dans quel ordre ils seront exécutés:

- FIFO : le plus simple et le plus logique

- Un pool de threads

On peut faire des implémentations de RMI qui créent un thread à chaque appel.

Implémentation actuelle de RMI jdk:

RMI ouvre une socket TCP/IP par client.

Un thread du cote serveur est dédié à la lecture de l’input stream de la socket.
C’est apparemment ce thread qui réalise l’appel de la méthode du serveur.

Donc, cela revient à une thread par client

4.5.2 Appel asynchrones

Comment faire un appel asynchrone avec RMI ?

- Gérer l’asynchronisme explicitement avec une thread

Dans le cas le plus simple:

une routine qui retourne void

il n'y a pas grand chose de plus à faire.

Le thread est utilisé simplement pour ne pas bloquer l’appelant.

Si on veut faire un appel asynchrone avec une routine qui ne retourne pas void:

Il faut également un emplacement spécial (futur) pour mettre le résultat

Il faudra également avoir une technique pour tester si le résultat est revenu (ou notifier le thread qui pourrait attendre le résultat)

Ainsi qu’une technique de synchronisation pour faire attendre l’appelant si on le souhaite.

----> Schéma, exemple

Soit on garde un thread bloquée chez l’appelant, soit on fait un Call Back depuis l’appelé.

4.5.3 Call back

Comment faire un Call back avec RMI ?

Très utile, par exemple pour informer un client qu'une valeur à changée dans un serveur.

Evite de faire du “pooling“ :

faire sans cesse des appels au serveur pour voir si la valeur à changée.

Pas de moyen simple est direct, il faut avoir un objet distant (Remote) du côté du client également afin de faire un appel RMI du serveur vers le client!

Deux solutions:

- Pattern au niveau des objets :

Callback sur le client lui-même:

- l'objet client est lui même un objet remote

    Il doit se protéger des call backs (synchronize).

- un autre objet de la VM du client est un objet remote,

    Il y a une référence mutuel entre celui-ci et le client

    ----> Schéma

Attention à au moins deux choses:

--> • accès concurrent

--> • possibilités de deadlock lors des call backs (car sémantique synchrone!).