RMI

VÍCTOR CUSTODIO

Introducción

- La invocación remota de métodos de Java es un modelo de objetos distribuidos, diseñado específicamente para el lenguaje Java, por lo que mantiene la semántica del modelo de objetos locales de Java, facilitando de esta manera la implantación y el uso de objetos distribuidos

Introducción

- Un objeto remoto es aquel cuyos métodos pueden ser invocados por objetos que se encuentran en una máquina virtual diferente.

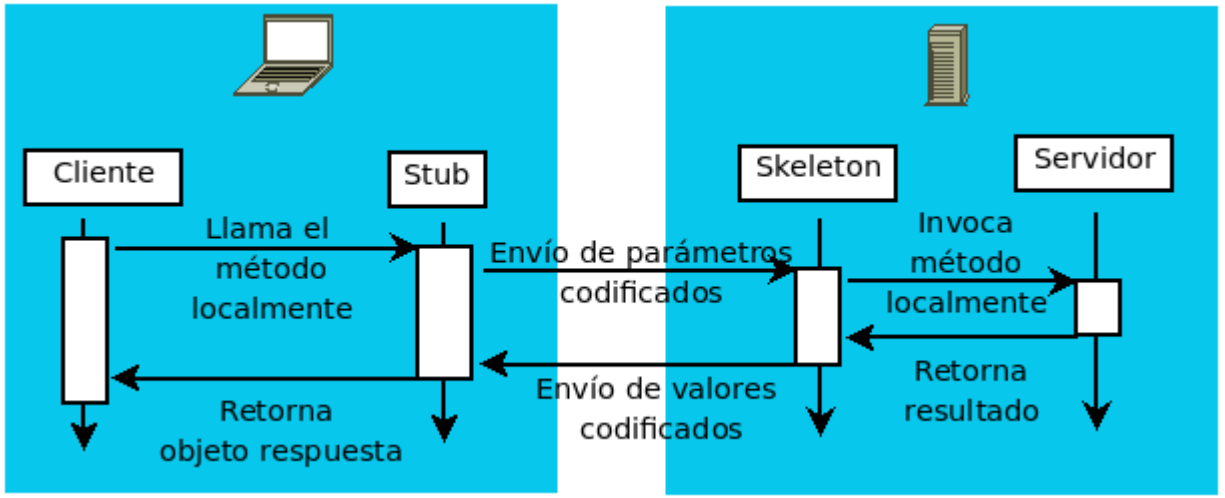
- Los objetos de este tipo se describen por una o más interfaces remotas que contienen la definición de los métodos del objeto que es posible invocar remotamente.

Definición

- RMI es un paquete de JAVA que permite manejar objetos (y sus respectivos métodos) de manera remota, para utilizar los recursos de un servidor de manera transparente para el usuario local.

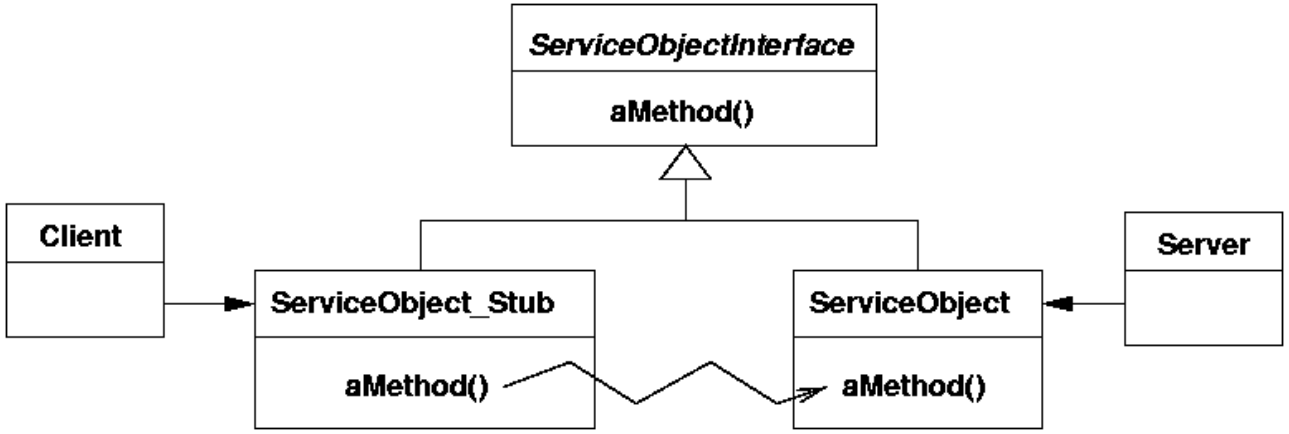
Definición

- La manera en que RMI logra hacer esto, es por medio de lo que se conoce como STUBs. En el caso del STUB servidor, se conoce como SKELETON. Estos Stubs y Skeletons permiten que al momento de ser invocada la función remota esta pueda ser simulada localmente



Skeletons y Stubs

- **Los stubs** forman parte de las referencias y actúan como representantes de los objetos remotos ante sus clientes. En el cliente se invocan los métodos del cabo, quien es el responsable de invocar de manera remota al código que implementa al objeto remoto.
- En RMI un stub de un objeto remoto implementa el mismo conjunto de interfaces remotas que el objeto remoto al cual representa.



Skeletons y Stubs (funcionamiento)

- Un cliente invoca a un método remoto:
  - La invocación es redirigida primero al stub.
  - El stub es el responsable de "transmitir" la invocación remota hacia el skeleton que está en el lado del servidor.
  - Para ello el stub abre una conexión (mediante un socket) con el servidor remoto, empaqueta los parámetros de la invocación y la redirige a través del flujo de datos hacia el skeleton.
  - El skeleton posee un método que recibe las llamadas remotas, desempaqueta los parámetros e invoca a la implementación real del objeto remoto.
  - El skeleton recibe el return de la implementación real y realiza el mismo camino que con los parámetros de entrada pero en sentido contrario.
- Finalmente el cliente recibe el resultado de la invocación al método remoto

Invisible al programador

Skeletons y stubs

- Los stubs se encargan de ocultar los mecanismos de comunicación empleados al programador.
- En el cliente se trabaja con el stub como si del skeleton se tratase. (como si tuviéramos el objeto remoto en nuestro propio sistema)
- En la MV remota, cada objeto debe poseer su skeleton correspondiente. El Skeleton es responsable de despachar la invocación al objeto remoto.

Como generar un Skeleton(stub en servidor)

- Requisitos para generar un Skeleton:
  - El objeto que queremos hacer remotamente accesible debe **implementar una interfaz**
  - Ésta interfaz debe **heredar de la interfaz Remote**
    - **(obliga a que sus métodos lancen una remoteException)**
  - Tanto los **parámetros** de entrada como lo valores de retorno deben ser **serializables** (implemetar la interfaz Serializable)

Como generar un Skeleton(stub en servidor)

- Ejemplo: Queremos hacer remotamente accesible la clase Addition, que simplemente realiza una suma

Interfaz:

```
import java.rmi.*;

public interface AdditionInterface extends Remote {
public int Add(int a,int b) throws RemoteException;
}
```

Implementación

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

public class Addition implements AdditionInterface {

    public Addition () throws RemoteException {  }

    public int Add(int a, int b) throws RemoteException {
    int result=a+b;
    return result;
    }
}
```

Como generar un Skeleton(stub en servidor)

- Para generar el Skeleton:

Addition add = **new Addition();**

AdditionInterface skeleton = (AdditionInterface) UnicastRemoteObject.*exportObject(add,0);*

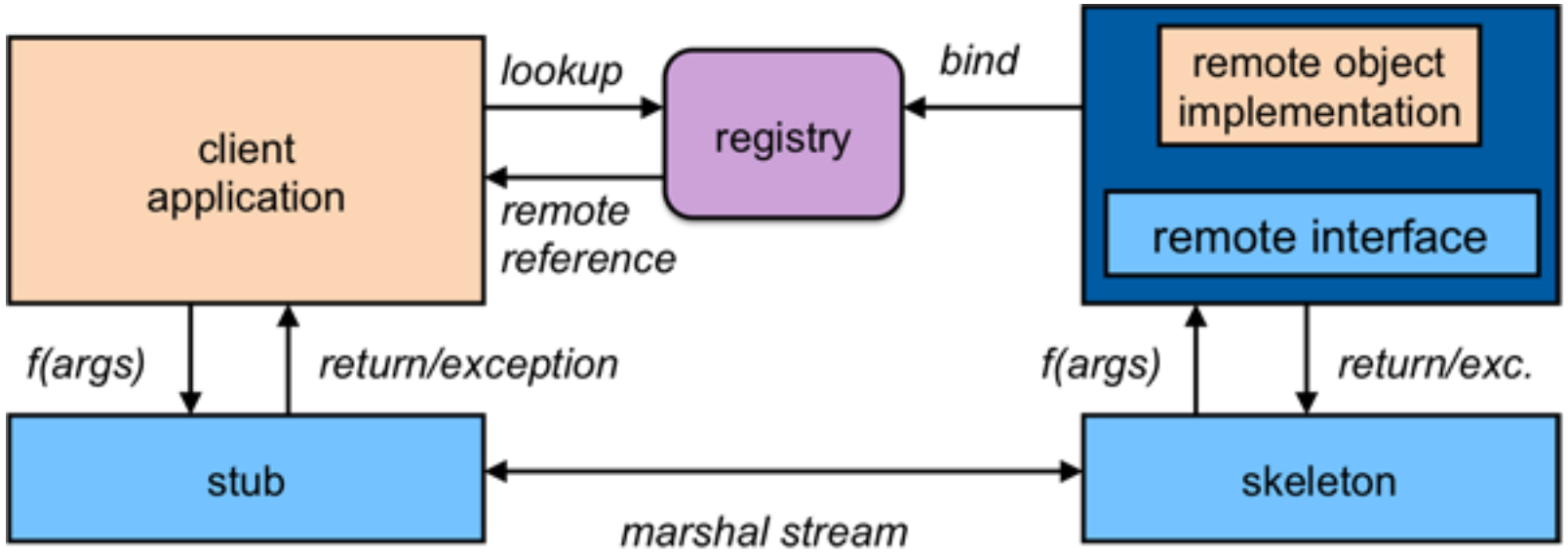
- ***UnicastRemoteObject.exportObject nos devuelve un skeleton que proporcionará los métodos especificados en la interfaz AdditionInterface a los stubs de los clientes. Para ello invocará los métodos a la instancia add de la clase addition.***
  - *EL primer parámetro, es el objeto remoto al que se invocará*
  - *El segundo parámetro es el puerto TCP/IP en el que se servirá este objeto (el valor cero indica un puerto anónimo y el S.O o la MV le asignarán cualquier puerto libre.*

¿Cómo obtenemos el stub de un objeto remoto en cliente?

- **Servicio de nombres.**
- Asocia nombres lógicos a los objetos remotos exportados (skeletons)
  - Servidor: asocia un nombre al objeto remoto (**registro**) [bind()]
  - Cliente: obtiene una referencia al objeto remoto (**stub**) a partir de ese nombre [lookup()]
- Objetivo: ofrecer transparencia de ubicación:
  - Evitar que el cliente tenga que ser recompilado si el objeto remoto pasa a ejecutarse en otra máquina virtual (JVM)
  - Almacena y mantiene asociación entre nombre de objeto remoto y su referencia remota serializada (stub)

RMI Registry

Es el servicio de nombres de objetos remotos en Java



RM Registry – Lado del Servidor

- Para registrar un objeto, se debe hacer un bind de su skeleton:

```
Addition add = new Addition();
AdditionInterface skeleton = (AdditionInterface)UnicastRemoteObject.exportObject(add,0);
Registry registry = LocateRegistry.getRegistry();
registry.rebind("Add", skeleton);
```

- Se utiliza LocateRegistry.getRegistry() para localizar el Servidor de nombres. Si no se le pasa ningún parámetro localiza el servidor de nombres en el localhost en el puerto 1099.
- registry.rebind("Add", skeleton), asigna un nombre lógico (en este caso "Add") a un skeleton (en este caso un skeleton que ofrece una interfaz remota al objeto add) . A diferencia de bind, si el nombre ya esta registrado, rebind lo machaca, bind lanza excepción.

RM Registry – Lado del Cliente

- Para utilizar un objeto remoto:

```
Registry registry = LocateRegistry.getRegistry("192.168.0.1", 1099);
AdditionInterface stub= (AdditionInterface)registry.lookup("Add");
int result=stub.add(9,11);
```

- Se utiliza **LocateRegistry.getRegistry()** para localizar el Servidor de nombres. Esta vez le pasamos como parámetro la dirección del registry porque no esta en local y el puerto donde esta escuchando..
- **registry.lookup("Add")**, nos devuelve un stub asignado al nombre que le pasemos por parametro (en este caso el stub del anterior diapositiva). A partir de ahí actuaremos como si de un objeto local se tratase.
- **Nota: es necesario que la MV de cliente tenga la clase AdditionInterface para poder tratar el stub, pero no es necesario que tenga la clase que lo implementa(class Addition)**

RMI Registry

- Podemos iniciar un Servicio de Nombres (Registry) de dos formas distintas:
  - Escribiendo el siguiente comando en la consola de comandos de Windows:
    - start rmiregistry

      *Opcionalmente se le puede indicar el puerto donde trabajará: start rmiregistry 1099
  - Programáticamente usando:
    - Registry registry = LocateRegistry.*createRegistry(port);*
      - **El servidor de nombres(registry se destruirá al finalizar la ejecución)*

Nota: Si queremos tener el servidor de nombres en una máquina distinta al servidor de objetos remotos la primera opción es la única valida.

RMI Registry

- Para indicarle al RMI Registry donde se encuentran las clases (codeBase) que tendrá que serializar (AdditionInterface.class en nuetro caso) tenemos varias opciones:
- DIRECTORIO FIJO:
  - CLASSPATH, indicamos en la variable CLASSPATH de la máquina donde se ejecuta el rmiregistry donde se encuentran las clases necesarias (solo si están localmente accesibles)
  - Se puede indicar un codeBase por defecto al rmiregistry al momento de iniciarlo con el siguiente formato:

```
Start rmiregistry -J-Djava.rmi.server.codebase=file:///C:/Users/Victor/workspace/ServerSide/bin/
```

Si quisiéramos descargar las clases de otra máquina o de un servidor http o ftp el formato de dirección cambiaría, por ejemplo

```
Start rmiregistry -J-Djava.rmi.server.codebase=http://www.victor.com/classes/
```

RMI Registry

- DIRECTORIO Dinámico. Los programas que se conectan al RMIRegistry le indican donde debe buscar las clases necesarias
  - Al ejecutar nuestro programa añadimos la siguiente opción:

```
-Djava.rmi.server.codebase=file:/C:/Users/Victor/workspace/ServerSide/bin/
```

  - Programaticamente añadimos lo siguiente:

```
System.setProperty("java.rmi.server.codebase","file:/C:/Users/Victor/workspace/ServerSide/bin/");
```

IMPORTANTE : En las últimas versiones de java por defecto esta deshabilitado el uso de codebase dinámicos por motivos de seguridad, para que rmiregistry tenga en cuenta los codebase dinámicos se debe iniciar de la siguiente forma:

```
Start rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

Seguridad

- Al utilizar RMI nos descargaremos código (parámetros de entrada y salida a los métodos remotos)de otras máquinas que ejecutaremos en nuestra máquina.
- Es necesario definir una política de seguridad para limitar las acciones que pueden realizar estos objetos. Esta política de seguridad se define en el fichero java.policy
- También es necesario activar un gestor de seguridad que aplique la política definida

Seguridad – java.policy

- Fichero de seguridad para el servidor

```
 grant codeBase "file:///C:/Users/Victor/workspace/ServerSide/bin/"
{
      permission java.security.AllPermission;
  };
```

- Este fichero se debe almacenar en alguna carpeta del proyecto
- Asigna todos los permisos a las clases que se encuentren en el codeBase definido. Las clases que no se encuentren en esta url (las descargadas por RMI no tendrán permisos.

Seguridad – java.policy

- Se puede gestionar la seguridad de forma más completa, por ejemplo

```
grant signedBy "Duke" {
    permission java.io.FilePermission "/tmp/*", "read,write";
};
```

- Permite leer y escribir ficheros a todas las clases firmadas por Duke. Para crear completas políticas de seguridad mirar:

http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html

Seguridad – Gestor de Seguridad

- Se encarga de aplicar la política definida en java.policy
- Para gestionar la seguridad en nuestra aplicación podemos añadir el siguiente código:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
    }
```

- De forma transparente al programador el secutrityManager se encargará de manejar los permisos.

Seguridad – Gestor de Seguridad

- Hay que indicar al gestor de seguridad donde esta nuestro fichero java.policy
- Tenemos dos opciones para hacerlo
  - Al ejecutar nuestro programa añadimos la siguiente opción:

```
-Djava.security.policy=file:/C:/Users/Victor/workspace/ServerSide/java.policy
```

  - Programaticamente añadimos lo siguiente:

```
System.setProperty("java.security.policy","file:/C:/Users/Victor/workspace/ServerSide/java.policy" );
```

Servidor RMI

```
public class AdditionServer {
   public static void main (String[] argv) {
   System.setProperty("java.rmi.server.codebase","file:///C:/Users/Victor/workspace/ServerSide/bin/");
   System.setProperty("java.security.policy","file:///C:/Users/Victor/workspace/ServerSide/java.policy" );
   if (System.getSecurityManager() == null) {
   System.setSecurityManager(new SecurityManager());
   }
   try {
   String name="Addition";
   Addition add = new Addition();
   AdditionInterface skeleton = (AdditionInterface) UnicastRemoteObject.exportObject(add,0);
   Registry registry = LocateRegistry.createRegistry(1059);
   registry.rebind(name, skeleton);
   System.out.println("Addition Server is ready.");
   }catch (Exception e) {
   System.out.println("Addition Server failed: " + e);
}
   }
}
```

Cliente RMI

```
public class AdditionClient {
public static void main (String[] args) {
System.setProperty("java.security.policy","file:///C:/Users/Victor/workspace/ClientSide/securit
y.policy" );

if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
try {
String name = "Addition";
Registry registry = LocateRegistry.getRegistry(args[0],1059);
AdditionInterface stub = (AdditionInterface)registry.lookup(name);
int result=stub.add(9,10);
System.out.println("Result is :"+result);

}catch (Exception e) {
System.out.println("Addition exception: " + e);
}
}
}
```

Ejemplo: Servidor ejecutor de tareas genéricas

- Objetivo: Crear un servidor que sea capaz de realizar tareas genéricas: Será capaz de realizar distintas tareas sin necesidad de modificar su código o parar su ejecución por cada tarea distinta que deba realizar.
  - Uso de tipos genéricos en Java
  - RMI con carga dinámica de clases
  - Publicación de clases.

1° Crear interfaces de los métodos remotos

Creamos el paquete compute, donde crearemos las siguientes interfaces

- Interfaz Compute

```
package compute;

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

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Interfaz Task

```
package compute;

public interface Task<T> {
    T execute();
}
```

Servidor :Creamos la implementación de la interfaz compute

- Creamos el paquete engine, donde creamos la clase ComputeEngine.

```
package engine;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
}
```

La implementación de la interfaz Task, la realizará cada cliente con el código que le interese que el servidor realice por ellos

Cliente: Creamos una implementación de TASK<T> : Pi.java

```
public class Pi implements Task<BigDecimal>, Serializable {

private static final long serialVersionUID = 227L;

private static final BigDecimal FOUR =
        BigDecimal.valueOf(4);

private static final int roundingMode =
        BigDecimal.ROUND_HALF_EVEN;

private final int digits;

public Pi(int digits) {
        this.digits = digits;
    }

public BigDecimal execute() {
        return computePi(digits);
    }

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);
    }
```

Cliente: Creamos la implementación de TASK<T>

```
public static BigDecimal arctan(int inverseX,
                                int scale)
    {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2 =
            BigDecimal.valueOf(inverseX * inverseX);

        numer = BigDecimal.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(BigDecimal.ZERO) != 0);
        return result;
    }
}
```

Creamos Servidor

- Aprovechamos la clase ComputeEngine para introducir el código del servidor en el método main:

```
public static void main(String[] args) {

  if (System.getSecurityManager() == null) {
          System.setSecurityManager(new SecurityManager());
      }
      try {
          String name = "Compute";
          Compute engine = new ComputeEngine();
          Compute stub =
              (Compute) UnicastRemoteObject.exportObject(engine, 0);
          Registry registry = LocateRegistry.getRegistry(1500);
          registry.rebind(name, stub);
          System.out.println("ComputeEngine bound");
      } catch (Exception e) {
          System.err.println("ComputeEngine exception:");
          System.exit(0);
          e.printStackTrace();
      }
  }
```

Creamos el Cliente

```
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;

import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {

if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0],args[1]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(args[2]);
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Publicamos las interfaces

- Generamos compute.jar
  - En eclipse, mediante exportación del paquete
  - En comandos de Windows no situamos en la carpe bin de nuestro proyecto :
    - jar cvf compute.jar compute/*.class
- Publicamos compute.jar
  - Instalamos un servidor web (Ej:apache2)
  - Creamos una carpeta llamada classes en la carpeta publica
  - Pegamos compute.jar en la carpeta classes

Iniciamos RMIRegistry

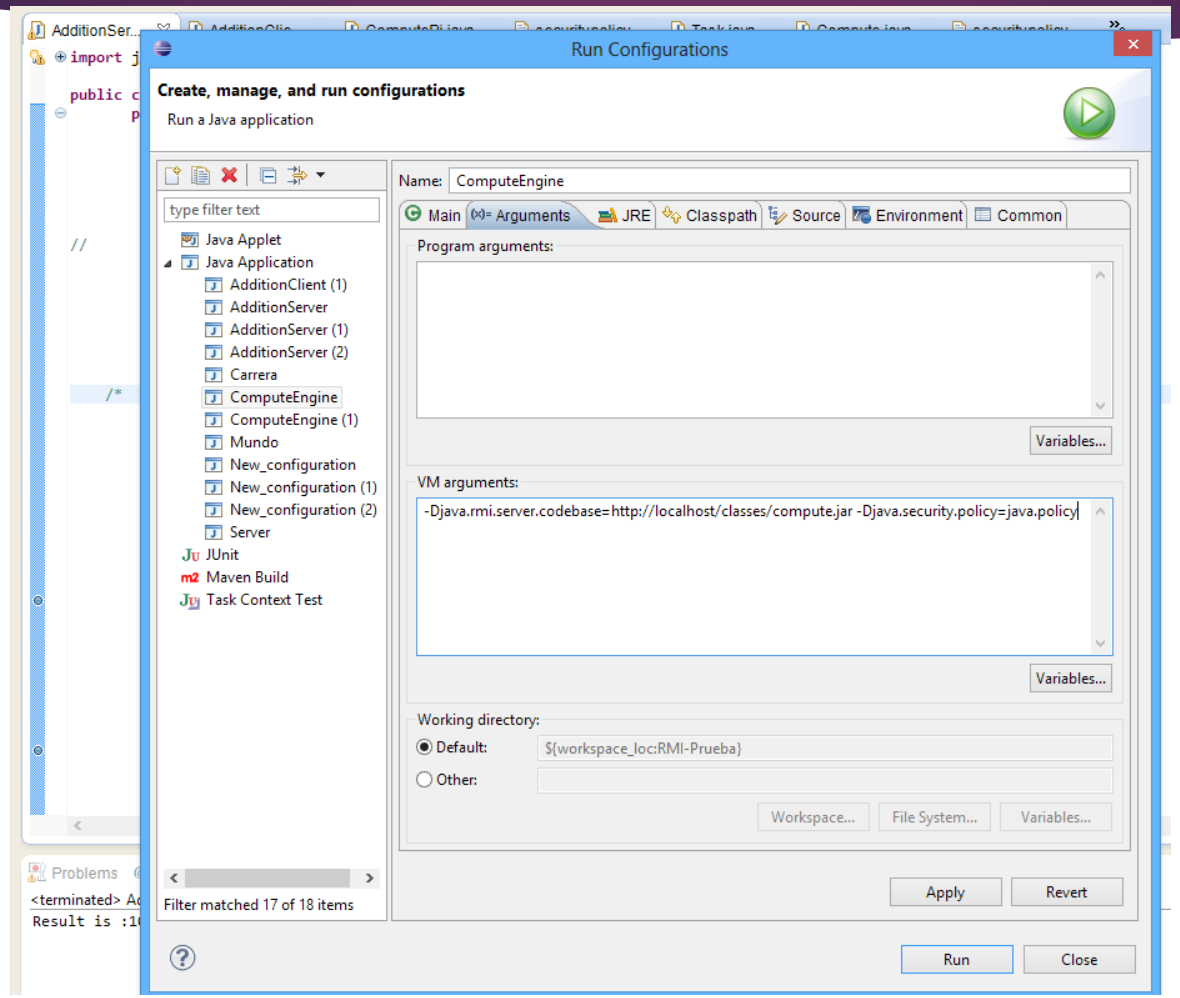
- `start rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false`

Iniciamos el servidor: crear el java.policy

Fichero : java.policy

```
grant codeBase "file:///C:/Users/Victor/workspace/RMI-
Prueba/bin/engine/" {
      permission java.security.AllPermission;
  };
```

Iniciamos el servidor: ejecutar el servidor



Cliente : publicar las clases necesarias

- En nuestro ejemplo será necesario serializar la clase Pi, por lo que debemos publicarla en el codebase para que RMI tenga acceso a deserializarla (unmarshal).Es suficiente con publicar el .class dentro de una carpeta con el nombre del paquete.
  - http://localhost/clases/client/ComputePi.class

Nota: el codebase será el mismo para Servidor y los clientes de un mismo registro de (nombre, objeto remoto)

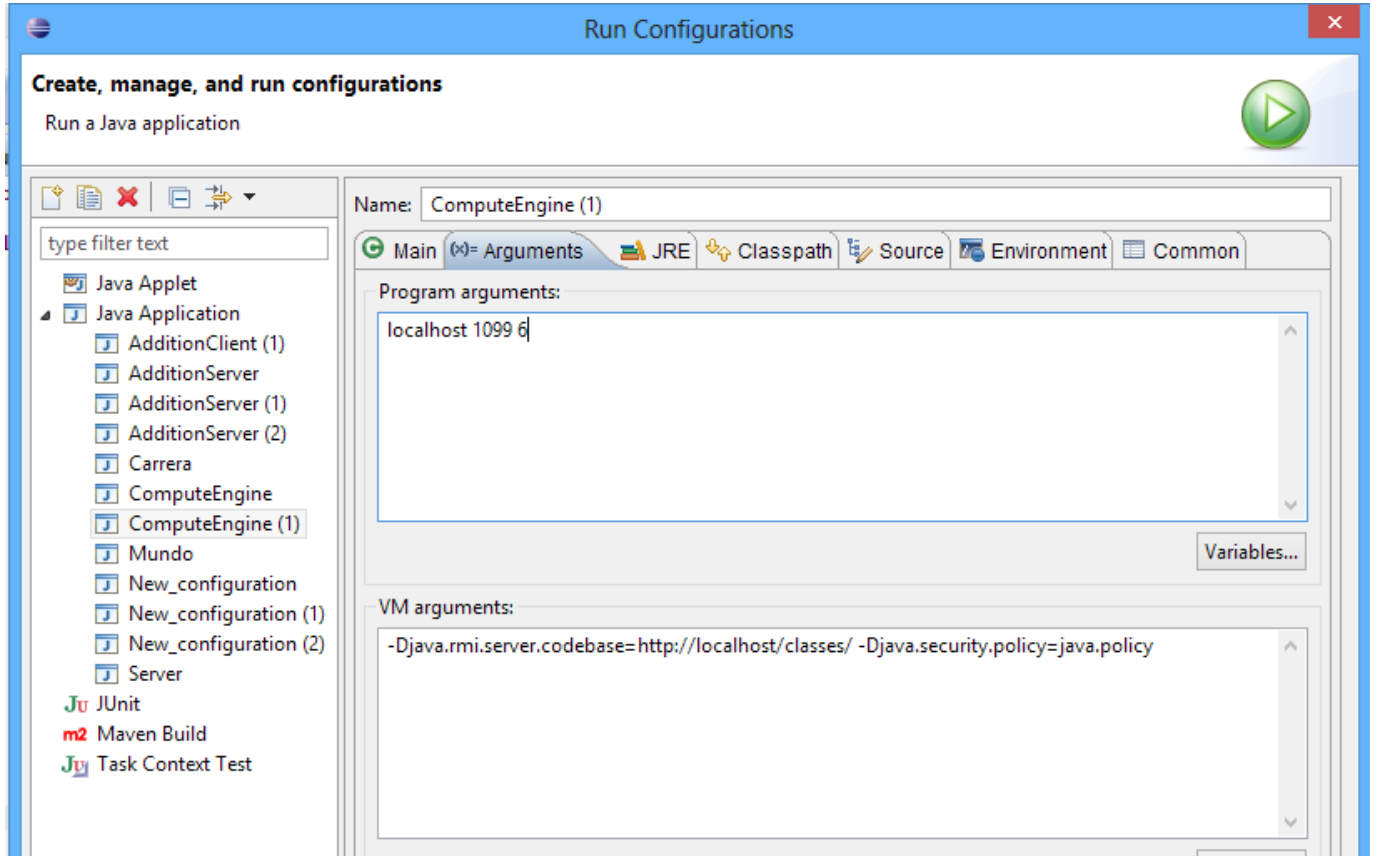
Iniciamos el Cliente: crear el java.policy

Fichero : java.policy

```
grant codeBase "file:///C:/Users/Victor/workspace/RMI-
Prueba/bin/client/" {
      permission java.security.AllPermission;
  };
```

Ejecutar Cliente

- Hay que pasarle los parámetros y Al igual que Al servidor añadir las opciones a la MV:



Ejercicio 1: Un servicio básico: servicio de eco

- El cliente enviará una cadena de caractereres recibida por argumentos en el main y el servidor le devolverá la misma cadena repetida y en mayúsculas. El cliente lo mostrará por pantalla.

Concurrencia RMI

- La invocación concurrente de métodos remotos se ejecuta de manera automática en hilos de control separados, según sea necesario.
- RMI internamente crea un Hilo por cada cliente.
- Cuando en un método se acceda a recursos no temporales que puedan ser modificados es necesario controlar la concurrencia (Syncronized) y wait and notify si fuera necesario.

Ejercicio 2 : Servicio de Log

- Crear un servicio de log, que conste de dos clientes que envían cadenas de texto constantemente a un servidor, y este debe escribir las cadenas en un fichero de log en orden.

Referencias remotas como parámetros

- En los ejemplos previos, los clientes obtenían las referencias remotas de servicios a través del rmiregistry. Sin embargo, teniendo en cuenta que estas referencias son objetos Java convencionales, éstas se pueden recibir también como parámetros de un método o como valor de retorno del mismo.
- Se podría decir que el rmiregistry sirve como punto de contacto inicial para obtener la primera referencia remota, pero que, a continuación, los procesos implicados pueden pasarse referencias remotas adicionales entre sí.

Ejercicio 3: Servicio de Chat

- Este servicio permitirá que cuando un usuario, identificado por un apodo, se conecte al mismo, reciba todo lo que escriben el resto de los usuarios conectados y, a su vez, éstos reciban todo lo que escribe el mismo.
- Esta aplicación se va a organizar con procesos clientes que atienden a los usuarios y un servidor que gestiona la información sobre los clientes/usuarios conectados.
- El servidor ofrecerá un servicio remoto para darse de alta y de baja, así como para enviarle la información que escribe cada usuario.
- En este caso, se requiere, además, que los clientes ofrezcan una interfaz remota para ser notificados de lo que escriben los otros clientes.

Ejercicio 3: Pista

Interfaz en el Servidor del servicio de chat:

```
import java.rmi.*;
interface ServicioChat extends Remote {
    void alta(Cliente c) throws RemoteException;
    void baja(Cliente c) throws RemoteException;
    void envio(Cliente c, String apodo, String m) throws RemoteException;
}
```

Interfaz en el Cliente, para recibir notificaciones:

```
import java.rmi.*;

interface Cliente extends Remote {
    void notificacion(String apodo, String m) throws RemoteException;
}
```