Programación Concurrente y de Tiempo Real

Guión de prácticas 10: Introducción a la programación distribuida: RMI a nivel básico

Natalia Partera Jaime
Alumna colaboradora de la asignatura

Índice

1. Introducción

Hasta el momento hemos visto cómo conseguir que varios procesos se ejecuten concurrentemente en una misma máquina. En este guión comenzaremos con la programación distribuida en tiempo real, donde un proceso servidor será ejecutado en una máquina y otros procesos clientes que se ejecuten en otras máquinas podrán interactuar con él.

En este guión veremos una introducción a RMI. RMI es una herramienta basada en el paradigma de objetos distribuidos. En este paradigma basado en objetos, los objetos se encuentran repartidos en distintos sistemas. Por lo que a veces se hace necesario acceder a algún objeto que reside fuera de nuestro sistema.

2. Conceptos previos

RMI es una API para programas Java que nos permite trabajar de manera distribuida. Se basa en el modelo de llamada a procedimientos remotos o RPC (Remote Procedure Call). El modelo de llamada a procedimientos remotos se basa, a su vez, en la programación procedimental.

La programación procedimental es aquella centrada en la ejecución de procedimientos, o funciones. Los procedimientos o funciones proporcionan abstracción sobre las acciones que realiza el sistema. Al llamar a una función, hacemos que su acción se ejecute. Normalmente, estas llamadas son a procedimientos que ya existen en el mismo sistema desde el que se invoca.

En el modelo de llamadas a procedimientos remotos, en cambio, los procesos a los que se llama no se encuentran en el mismo sistema desde el que se produce la llamada.

3. RMI (Remote Method Invocation)

RMI es una implementación orientada a objetos del modelo de llamada a procedimientos remotos. Con RMI podemos utilizar objetos remotos (objetos que se encuentran en otra JVM) como si se encontrasen en la misma JVM (la JVM local). Como RMI es orientado a objetos, nos permite disponer en Java de objetos distribuidos. Aunque también puede generalizarse a otros lenguajes usando JNI.

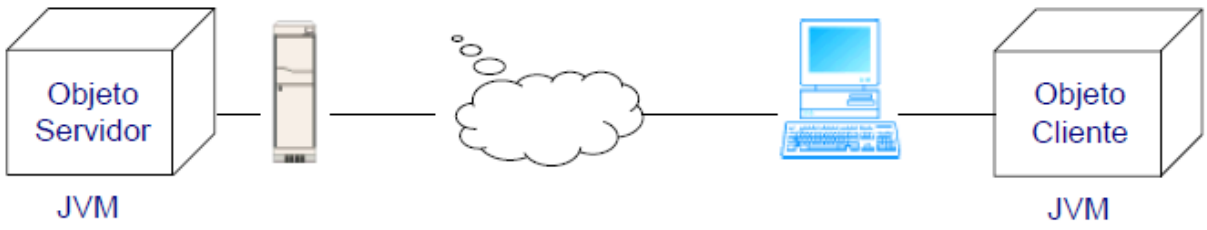

Figura 1: Diagrama del uso distribuido.

Una de las principales diferencias entre RMI y RPC es que RMI se basa en objetos. Además, RMI proporciona mayor abstracción que RPC.

Para lograr una correcta ejecución, RMI pasa los parámetros y los valores de retorno utilizando la serialización de objetos.

Veamos cómo se realizan llamadas a métodos de objetos locales y de objetos remotos:

```
//Ejemplo de llamada a método local (o de un objeto local)
int vAbs;
vAbs = ValorAbs(x);

//Ejemplo de llamada a método remoto (o de un objeto remoto)
InterfazRemota objRem = (InterfazRemota) Naming.lookup("Server");
objRem.ValorAbs(x);
```

3.1. Arquitectura RMI

Antes hemos mencionado que el API RMI de Java permite que podamos invocar a objetos remotos como si fueran objetos locales. En realidad, la arquitectura de Java RMI utiliza ciertos módulos de software para enmascarar los detalles de la comunicación entre estos procesos. El módulo software que realiza esta función suele ser llamado *resguardo* (*stub*) o *proxy*.

El proxy del cliente se llama *stub*, mientras que el del servidor se llama *skeleton*. A partir de la versión 1.2 del `jdk` se introducen algunos cambios, y el *skeleton* deja de ser necesario.

Veamos qué debe hacerse con el servidor:

- El servidor debe extender a la clase `RemoteObject`.
- El servidor debe implementar una interfaz diseñada previamente.
- El servidor debe tener, al menos, un constructor nulo que lanzará la excepción `RemoteException`.
- El método `main` del servidor debe lanzar un gestor de seguridad.
- El método `main` del servidor crea los objetos remotos.

Ahora veamos qué precauciones tener con los clientes:

- Los clientes de objetos remotos se comunican con interfaces remotas diseñadas previamente.
- Los objetos remotos son pasados por referencia.
- Los clientes que llaman a métodos remotos deben manejar excepciones.

Además, hay que tener en cuenta que el compilador de RMI (`rmic`) es quien genera el *stub* y el *skeleton*.

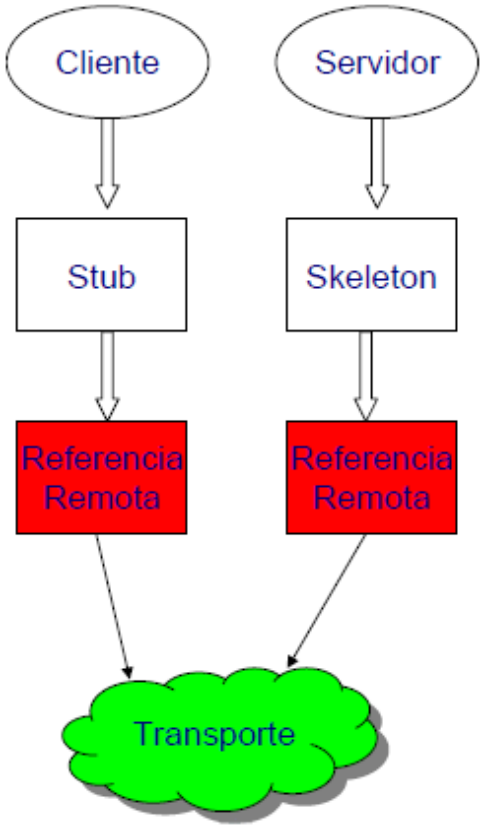

Figura 2: Diagrama de la arquitectura RMI.

3.2. Paquete RMI

Antes de empezar a desarrollar una aplicación RMI, veamos algunos elementos del paquete RMI en Java. Con la clase `java.rmi.Naming` podemos almacenar y recuperar referencias a objetos remotos en un registro de objetos remotos. Veamos los métodos de esta clase, son todos estáticos, por los que hay que invocarlos precediéndolos del nombre de su clase. Además, todos reciben una URL definida como `String` en uno de sus argumentos.

- `void bind(String name, Remote obj)`: asocia al objeto remoto el nombre que se indica como primer argumento.
- `String[] list(String name)`: devuelve una lista con los nombres existentes en el registro.
- `Remote lookup(String name)`: devuelve una referencia para el objeto remoto asociado con el nombre que se indica como argumento.
- `void rebind(String name, Remote obj)`: vuelve a asociar el nombre indicado con un nuevo objeto remoto.
- `void unbind(String name)`: destruye la asociación para el nombre indicado con el objeto remoto al que referenciaba.

El uso habitual de estos métodos es el siguiente:

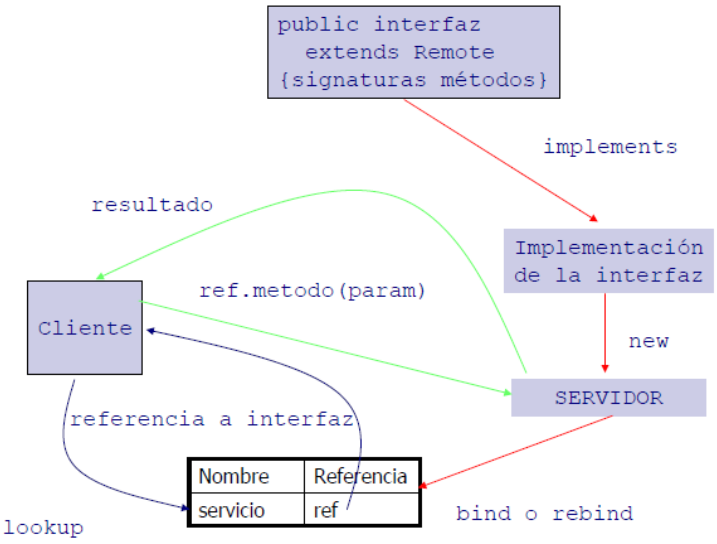

Figura 3: Uso de los métodos de la clase Naming.

4. Fases de diseño RMI

Ya podemos comenzar a desarrollar una aplicación en Java usando RMI. Para poner en marcha nuestra aplicación distribuida, tendremos que seguir los pasos que se detallan en los siguientes subapartados.

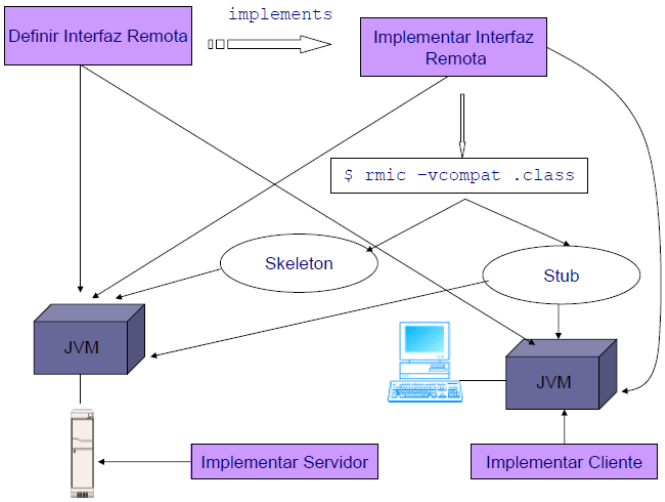

Figura 4: Diagrama resumen de las fases del diseño RMI.

4.1. Diseño de las interfaces

Lo primero es escribir el fichero de la interfaz remota. El propósito de la interfaz es ocultar la implementación de los aspectos relativos a los métodos remotos. Así, cuando el cliente logra una referencia a un objeto remoto, en realidad obtiene una referencia a una interfaz. Durante la ejecución, los clientes envían sus mensajes a los métodos de la interfaz.

- Debe ser `public` y extender a `Remote`.
- Declara todos los métodos que el servidor remoto ofrece, pero no los implementa. De cada método se indica el nombre, los parámetros y el tipo de devolución.
- Todos los métodos de la interfaz remota lanzan obligatoriamente la excepción `RemoteException`.

A continuación puede ver el ejemplo de la interfaz remota de la clase `Servidor` que provee de algunas operaciones matemáticas:

```
/**
 * Interfaz remota.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.rmi.*;

public interface InterfazRemota extends Remote
{
  int Potencia(int base, int exp) throws RemoteException;
  float RaizCuadrada(int rad) throws RemoteException;
  int ValorAbs(int x) throws RemoteException;
}
```

Ejercicio 1 Desarrolle una interfaz remota que tenga, al menos, 2 o 3 métodos. Tenga en cuenta que en los próximos ejercicios tendrá que implementarla. Cuando termine, puede comparar su interfaz con la que aparece en el apartado 5.1.

4.2. Implementación del servidor

La implementación del servidor es un fichero que realiza la implementación de la interfaz definida previamente.

- El servidor debe contener una clase que extienda a `UnicastRemoteObject` y que implemente a la interfaz remota.
- Debe tener un constructor que lance la excepción `RemoteException`.
- El método `main` debe lanzar un gestor de seguridad.
- El método `main` debe crear los objetos remotos deseados.
- El método `main` debe registrar al menos uno de los objetos remotos.

A continuación puede ver el ejemplo de la implementación del servidor correspondiente a la interfaz del ejemplo anterior:

```
/**
 * Servidor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.rmi.*;
import java.rmi.server.*;
import java.lang.Math.*;

public class Servidor extends UnicastRemoteObject implements InterfazRemota {
  public int Potencia(int base, int exp) throws RemoteException {
    int res = 1;

    if (base == 0)
      return 0;
    for (int i = 0; i < exp; ++i) {
      res = res * base;
    }
    return res;
  }


  public double RaizCuadrada(int rad) throws RemoteException {
    return Math.sqrt(new Double(rad));
  }


  public int ValorAbs(int x) throws RemoteException {
    if(x >= 0)
      return x;
    else
      return (-1) * x;
  }


  public Servidor() throws RemoteException {
    super();
  }


  public static void main (String[] args) throws Exception {
    //Creación de objetos remotos
    Servidor objRem1 = new Servidor();
    //Registro del objeto en la máquina remota. Si no se especifica otro puerto, se
    //asume el 1099
    Naming.bind("Server", objRem1);
    System.out.println("Servidor Remoto Preparado.");
```

```
  }
}
```

Ejercicio 2 Desarrolle el servidor correspondiente a la interfaz del ejercicio anterior. Cuando termine, puede comparar su resultado con el código que aparece en el apartado 5.2.

4.3. Generación de los ficheros *stub* y *skeleton*

Para generar los ficheros de *stub* y de *skeleton* es necesario que hayamos compilado ya la implementación de la interfaz. Tras ello, se compila con el compilador de RMI, y éste genera los dos ficheros: *stub* y *skeleton*.

Figura 5: Generación del *stub* y del *skeleton*.

Para generar los ficheros *stub* y *skeleton*, compilamos con `rmic` el servidor:

```
$ rmic ClaseServidor
```

Una vez que los tengamos, situamos los ficheros de *stub* y *skeleton* también en la máquina remota donde se aloja el servidor. Una vez hecho, se lanza el servidor llamando a la JVM del host remoto:

```
$ java ClaseServidor &
```

```
//Secuencia de comandos
$ javac InterfazRemota.java
$ javac Servidor.java
$ rmic Servidor
```

4.4. Activación del servicio de nombres

Para activar el servicio de nombres, debemos elegir un puerto. Por defecto, el puerto que se utiliza es el 1099. Activamos el servicio de nombres con `start rmiregistry [puerto]`.

Atención: si estamos trabajando en un sistema operativo Linux, tras la orden anterior deberemos introducir `&`.

```
//Activación en Windows:
$ start rmiregistry [puerto]

//Activación en Linux:
$ start rmiregistry [puerto] &
```

4.5. Registro de objetos

Para que se lleve a cabo el registro de objetos, es necesario que el servidor de nombres esté activo. El registro de objetos se lleva a cabo mediante las llamadas al método `Naming.bind()` que aparezcan en el código del servidor cuando se ejecute. El nombre que se le asigne a un objeto remoto puede ser el nombre de un equipo, por ejemplo:

```
Naming.bind("//sargo.uca.es:2005/Servidor", ORemoto);
```

```
//Los comandos ejecutados hasta el momento han sido:
$ javac InterfazRemota.java
$ javac Servidor.java
$ rmic Servidor
$ start rmiregistry
$ java Servidor
```

4.6. Implementación del cliente

El objeto cliente siempre crea un objeto de interfaz remota al que le asigna el resultado de llamar a `Naming.lookup`. Esta llamada recibe como parámetros el nombre del equipo, el puerto del equipo y el nombre del servidor con forma de URL. La llamada a `Naming.lookup` devuelve una referencia que se convierte a una referencia a la interfaz remota. A partir de ese momento, a través de esa referencia el programador puede invocar todos los métodos que desee de esa interfaz remota como si fueran referencias a objetos en la JVM local.

```
/**
 * Cliente.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.rmi.*;
import java.rmi.registry.*;

public class Cliente
{
  public static void main (String[] args) throws Exception
  {
    int a = 5;
    int b = 3;
    int c = -1;
```

```
    int d = 16;

    //Obtener la referencia al objeto remoto usando su interfaz
    InterfazRemota refObjRem = (InterfazRemota) Naming.lookup("Server");

    //Llamadas a los métodos de la interfaz remota
    System.out.println("Potencia de " + a + " elevado a " + b + " = " + refObjRem.Potencia(a, b));
    System.out.println("Raiz cuadrada de " + d + " = " + refObjRem.RaizCuadrada(d));
    System.out.println("Valor absoluto de " + c + " = " + refObjRem.ValorAbs(c));
  }
}
```

Ejercicio 3 Desarrolle un cliente para la interfaz remota y el servidor de los ejercicios anteriores. Cuando finalice, puede comparar su cliente con el que encontrará en el apartado 5.3.

4.7. Resultado final

Finalmente, los ficheros que deben encontrarse en cada equipo son los siguientes:

- Cliente:
  - Fichero objeto resultante de compilar la interfaz remota.
  - Fichero objeto resultante de compilar la clase cliente.
  - Fichero objeto de *stub*.
- Servidor:
  - Fichero objeto resultante de compilar la interfaz remota.
  - Fichero objeto resultante de compilar la clase servidor.
  - Fichero objeto de *stub*.
  - Fichero objeto de *skeleton*.

Ejercicio 4 Compile los ejemplos de la fase de diseño y ejecútelos. Lance el cliente desde una terminal y cada cliente desde otra terminal distinta.

Ejercicio 5 Compile el resultado de los ejercicios propuestos durante la explicación de la fase de diseño y ejecútelos. Compruebe que funcionan como esperaba.

---

5. Soluciones de los ejercicios

Compruebe los resultados de sus ejercicios con estas soluciones.

5.1. Ejercicio 1

Esta es una posible interfaz:

```
/**
 * Interfaz remota.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.rmi.*;

public interface InterRem extends Remote
{
  String Concatena(String s1, String s2) throws RemoteException;
  boolean Contiene(String cadena, String s) throws RemoteException;
  int Longitud(String s) throws RemoteException;
}
```

5.2. Ejercicio 2

Esta es la implementación del servidor correspondiente a la interfaz del ejercicio 1:

```
/**
 * Servidor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.rmi.*;
import java.rmi.server.*;
import java.lang.Math.*;

public class Serv extends UnicastRemoteObject implements InterRem {
  public String Concatena(String s1, String s2) throws RemoteException {
    return (s1 + s2);
  }


  public boolean Contiene(String cadena, String s) throws RemoteException {
    return cadena.contains(s);
  }


  public int Longitud(String s) throws RemoteException {
```

```
    return s.length();
  }


  public Serv() throws RemoteException {
    super();
  }


  public static void main (String[] args) throws Exception {
    //Creación de objetos remotos
    Serv objRem = new Serv();
    //Registro del objeto en la máquina remota. Si no se especifica otro puerto, se
    //asume el 1099
    Naming.bind("Server", objRem);
    System.out.println("Servidor Remoto Preparado.");
  }
}
```

5.3. Ejercicio 3

Esta es la clase que representa a los objetos clientes del servidor:

```
/**
 * Cliente.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.rmi.*;
import java.rmi.registry.*;

public class Cliente
{
  public static void main (String[] args) throws Exception
  {
    String s1 = "En un lugar ";
    String s2 = "de la Mancha";
    String s3 = "Andalucia";

    //Obtener la referencia al objeto remoto usando su interfaz
    InterRem refObjRem = (InterRem) Naming.lookup("Server");

    //Llamadas a los métodos de la interfaz remota
    String quijote = refObjRem.Concatena(s1, s2);
    System.out.println(quijote);
    System.out.println("La cadena " + s3 + " tiene longitud " +
      refObjRem.Longitud(s3));
    if (refObjRem.Contiene(quijote, s3))
      System.out.println("La cadena del quijote contiene " + s3);
```

```
    else
      System.out.println("La cadena del quijote no contiene " + s3);
  }
}
```