Tema 9

Llamada a métodos remotos (RMI).

Departament d'Informàtica. Universitat de València

Índice

1. Introducción **2**

1.1. ¿Cómo funciona RMI? 2

2. Usando RMI **4**

2.1. Fase de desarrollo: el servidor 4

2.1.1. Creación de una interface que describe el servicio 4

2.1.2. Implementación de la interface 5

2.1.3. Generación de stub (SDK 1.2) o stub y skeleton (SDK 1.1) 6

2.1.4. Clase que crea una instancia del servicio y la registra 6

2.2. Fase de ejecución: el servidor 7

2.2.1. Iniciar rmiregistry 7

2.2.2. Iniciar el servidor 7

2.3. Fase de desarrollo: el cliente 8

2.3.1. Creación del cliente 8

2.4. Fase de ejecución: el cliente 9

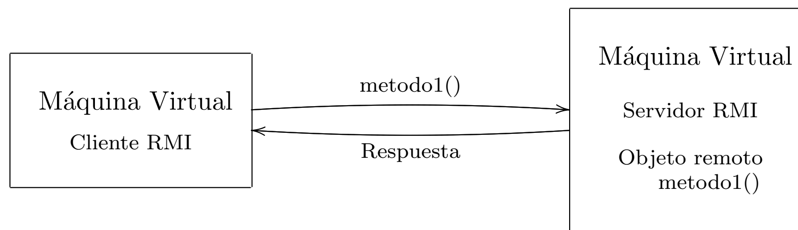
2.4.1. Iniciar el cliente 9

2.5. Una situación realista: carga dinámica clases 9

1. Introducción

¿Qué es RMI (*Remote Method Invocation*)? Es una tecnología Java que permite enviar mensajes a objetos situados en otra máquina virtual desde una aplicación que esté ejecutándose en una máquina virtual.

En su versión básica requiere que toda la aplicación (tanto el cliente como el servidor) estén desarrollados en Java.



Esto permite tener los objetos distribuidos en diversas máquinas.

Esta tecnología permite que se puedan pasar argumentos al método remoto y recibir los datos que devuelve (en ambos casos pueden ser tipos primitivos u objetos de clases que sean serializables).

Cada servicio RMI (objeto remoto) se define mediante una interface, que fija los métodos que se pueden invocar en el objeto remoto.

Esta interface debe estar disponible en el cliente y en el servidor.

La tecnología RMI no es un concepto nuevo. Antes de la programación orientada a objetos existía tecnología que permitía realizar llamadas a funciones y procedimientos remotos (RPC, *Remote Procedure Calls*).

En sistemas orientados a objetos también se puede utilizar CORBA (*Common Object Request Broker Arquitecture* o Arquitectura Común de Agente de Solicitud de Objetos) que soportan la utilización de diferentes lenguajes.

Java soporta RMI sobre IIOP (*Internet Inter-ORB*) con lo cual es posible integrar RMI con CORBA.

1.1. ¿Cómo funciona RMI?

Los sistemas que utilizan RMI se dividen en dos categorías: clientes y servidores.

El servidor proporciona un **servicio RMI** y el cliente llama a métodos del objeto ofrecido por el servicio.

El servicio RMI se debe registrar en un servicio de consulta para permitir a los clientes encontrar el servicio.

El J2SE incluye una aplicación llamada `rmiregistry`, que lanza un proceso que permite registrar servicios RMI mediante un nombre. Este nombre identifica al servicio, esto se hace así ya que en una máquina puede haber diferentes servicios.

Una vez que el servicio se ha registrado, el servidor esperará a que lleguen peticiones RMI desde los clientes.

El cliente solicita el servicio mediante el nombre con el que fue registrado y obtiene una referencia al objeto remoto.

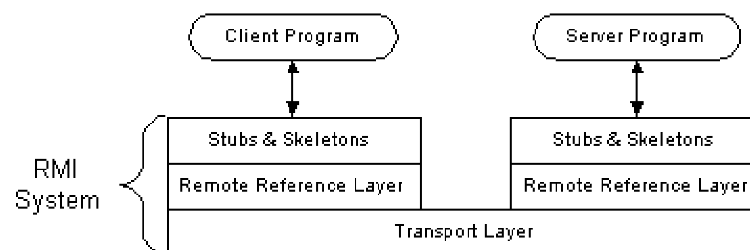
El formato utilizado por RMI para representar una referencia al objeto remoto es el siguiente

rmi://hostname:puerto/nombreServicio

Una vez obtenida la referencia remota (ya sea mediante rmiregistry, leyendo el URL de un archivo,...) los clientes pueden enviar mensajes como si se tratase de objetos ejecutándose en la misma máquina virtual.

Los detalles de red de las peticiones y las repuestas son transparentes para el desarrollador. Esto se consigue mediante el uso de *stub* (a partir de la versión 1.2, ya que en la versión 1.1 se requería generar un *skeleton*).

Las capas de la arquitectura RMI son las siguientes:



Capa de stub y skeletons

La capa de stub y skeletons se sitúa bajo la aplicación. En esta capa, RMI utiliza el patrón de diseño Proxy. En el patrón Proxy, un objeto en un contexto es representado por otro (el proxy) en un contexto separado.

En el uso que se hace en RMI del patrón proxy, la clase stub realiza el papel del proxy de la implementación del servicio remoto.

Mediante la utilización de este patrón es posible acceder al servicio como si se tratase de objetos locales.

Antes de que el stub pase una petición al servicio remoto debe agrupar los parámetros (ya sean tipos primitivos, objetos o ambos) para la transmisión, operación conocida como *marshalling*.

La clase skeleton es una clase generada por RMI. Esta clase es la responsable de comunicarse con el stub durante la comunicación RMI. Debe reconstruir los parámetros para formar los tipos primitivos y objetos, lo que es conocido como *unmarshalling*.

A partir de la versión 1.2 del SDK, el nuevo protocolo utilizado para RMI no necesita la clase skeleton.

Capa de Referencia Remota

Esta capa es utilizada por la capa de stub y skeleton. Es la capa responsable del funcionamiento independientemente de la la capa de transporte que se esté utilizando.

Esta capa proporciona un objeto RemoteRef que representa la conexión con el objeto remoto.

El objeto stub utiliza el método invoke(·) de RemoteRef para enviar las llamadas a los métodos del objeto remoto.

Capa de Transporte

La capa de transporte realiza la conexión entre las máquinas virtuales. Todas las conexiones son conexiones de red basadas en flujos que utilizan TCP/IP.

Sobre TCP/IP, RMI utiliza un protocolo llamado Java Remote Method Protocol (JRMP).

Sun e IBM han trabajado de forma conjunta para obtener la nueva versión de RMI llamada RMI-IIOP, en la que se puede trabajar con el protocolo IIOP en lugar de JRMP para comunicar clientes y servidores.

2. Usando RMI

2.1. Fase de desarrollo: el servidor

2.1.1. Creación de una interface que describe el servicio

En este ejemplo, el objeto remoto va a ofrecer un único método getMensaje() que devuelve un String.

La interface que describe el servicio es la siguiente (debe extender a la interface Remote).

```
import java.rmi.*;

/** Interface que describe el servicio RMI
 * Define los métodos que se pueden llamar desde el cliente
 */
public interface Saludo extends Remote {

   /** Metodo que se debe implementar
    * @return una String
    * @throws RemoteException
    */
   public String getMensaje() throws RemoteException;
}
```

Compilamos:

```
javac Saludo.java
```

2.1.2. Implementación de la interface

El siguiente paso a realizar es crear una clase que implemente a la interface anterior.

Esta clase puede extender a una de las siguientes clases:

- UnicastRemoteObject si el objeto va a estar ejecutándose en la máquina virtual remota.
- Activatable si el objeto se cargará en la máquina virtual remota cuando algún cliente envíe un mensaje.

En el ejemplo mostrado optamos por la primera opción.

Esta clase puede tener otros métodos aparte de los especificados en la interface pero no podrán ser llamados por el cliente.

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

/** Clase que implementa a la interface que describe el servicio
 */
public class SaludoImpl extends UnicastRemoteObject
                        implements Saludo {
   private String cadena;

   public SaludoImpl() throws RemoteException{
      cadena = "Mensaje desde el servidor RMI";
   }

   /** Metodo definido en la interface Saludo */
   public String getMensaje() throws RemoteException{
      return cadena;
   }
}
```

Compilamos:

```
javac SaludoImpl.java
```

2.1.3. Generación de stub (SDK 1.2) o stub y skeleton (SDK 1.1)

Si tanto el servidor como los servidores son Java, podemos trabajar con RMI sobre el protocolo JRMP.

Si vamos a trabajar con el SDK 1.2 podemos hacer lo siguiente:

```
rmic -v1.2 SaludoImpl
```

Esto genera la clase `SaludoImpl_Stub.class`

Si vamos a trabajar con el SDK 1.1 (quizá por compatibilidad con software existente) realizamos lo siguiente:

```
rmic -v1.1 SaludoImpl
```

Lo cual genera las clases `SaludoImpl_Stub.class` (para el cliente) y la clase `SaludoImpl_Skeleton.class` (para el servidor).

2.1.4. Clase que crea una instancia del servicio y la registra

Vamos a crear ahora el servidor RMI.

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;


/**
 * Servidor RMI
 */
public class SaludoServidor {

   public static void main(String[] args) {

      // Instalamos un RMISecurityManager()
      System.setSecurityManager(new RMISecurityManager());

      try{
         // Creamos el objeto que ofrecemos como servicio
         Saludo sal = new SaludoImpl();

         // ... y lo registramos
         // Puesto que no estamos especificando el host ni el puerto
         // tendrán los valores por defecto: //localhost y 1099
         Naming.rebind("Saludo",sal);
```

```
        }catch(RemoteException e){
            e.printStackTrace();
        }catch(MalformedURLException e){
            e.printStackTrace();
        }
    }
}
```

2.2. Fase de ejecución: el servidor

2.2.1. Iniciar rmiregistry

`rmiregistry` es una utilidad ofrecida por Sun para registrar los servicios RMI.

`rmiregistry` crea e inicia un registro de objetos remotos en un determinado puerto de la máquina local. Si se omite el puerto, el registro se inicia en el puerto 1099.

Un registro de objetos remotos es un servicio de nombres que es utilizado por los servidores RMI de la máquina para asociar objetos remotos con nombres. Los clientes en la máquina local o en máquinas remotas pueden buscar objetos remotos a través del nombre con el que se ha registrado el servicio.

Desde el directorio en el que se encuentran todas las clases se lanza el registro:

```
rmiregistry
```

2.2.2. Iniciar el servidor

Puesto que se ha instalado un gestor de seguridad RMI hemos de crear un archivo en el que se especifiquen lo permisos otorgados a la aplicación (en este caso serán permisos de conexión para los sockets). El archivo `servidor.policy` contiene lo siguiente:

```
grant{
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission "*:80", "connect,accept";
};
```

Desde el directorio donde se encuentran las clases compiladas lanzamos el servidor:

```
java -Djava.security.policy=servidor.policy SaludoServidor
```

2.3. Fase de desarrollo: el cliente

2.3.1. Creación del cliente

El cliente del servicio RMI será:

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;

/**
 * Cliente RMI
 */
public class SaludoCliente {

   public static void main(String[] args) {

      // El argumento será el host donde está el servicio
      String url = "//localhost/";


      System.setSecurityManager(new RMISecurityManager());

      try {

         // Obtencion de los servicios RMI disponibles

         String[] servicios = Naming.list(url);

         // Se muestran los servicios RMI
         for (int i=0;i<servicios.length;i++)
            System.out.println(servicios[i]);

         // Obtenemos la referencia al objeto remoto
         Saludo sal = (Saludo) Naming.lookup(url + "Saludo");

         // Ahora la utilizamos como si se tratase de un objeto
         // local

         String mensaje = sal.getMensaje();

         System.out.println(mensaje);

      } catch (MalformedURLException e) {
         e.printStackTrace();
      } catch (RemoteException e) {
         e.printStackTrace();
      } catch (NotBoundException e) {
         e.printStackTrace();
      }
   }

}
```

En el cliente deben estar visibles (en el mismo directorio o en el classpath) la interface que describe el servicio remoto y la clase stub generada con rmic.

2.4. Fase de ejecución: el cliente

2.4.1. Iniciar el cliente

Igual que en el caso anterior hemos de dar permisos a los sockets. El archivo `cliente.policy` contiene lo siguiente:

```
grant{
   permission java.net.SocketPermission "*:1024-65535", "connect,accept";
   permission java.net.SocketPermission "*:80", "connect,accept";
};
```

Desde el directorio donde se encuentran las clases compiladas lanzamos el cliente:

```
java -Djava.security.policy=cliente.policy SaludoCliente
```

2.5. Una situación realista: carga dinámica clases

En la mayoría de situaciones reales, el servidor y el cliente no están en la misma máquina.

En este caso voy a suponer que en la máquina que va a actuar de servidor RMI se encuentra el directorio `Servidor` con los siguientes ficheros:

```
Servidor
   |  SaludoServidor.class
   |  Saludo.class
   |  SaludoImpl.class
   |  SaludoImpl_Stub.class
   |  servidor.policy
```

Y en la máquina que va a actuar como cliente se encuentra el directorio `Cliente` con los siguientes ficheros:

```
Cliente
   |  SaludoCliente.class
   |  Saludo.class
   |  cliente.policy
```

En ésta situación ¿cómo puede acceder el cliente a la clase `SaludoImpl_Stub.class`?

La solución es que las clases necesarias se carguen de forma dinámica utilizando la capacidad que ofrece Java para descargar clases desde un URL.

Esto se puede conseguir utilizando un *codebase* que es una localización desde la cual la máquina virtual puede obtener las clases que necesite.

Esta localización se puede establecer utilizando la propiedad java.rmi.server.codebase que representa uno o más URL's desde donde se pueden descargar los stubs (y otras clases que sean necesarias).

En este caso, el proceso es el siguiente:

- Se lanza `rmiregistry` desde un directorio en el que no esté ninguna de las clases del servidor
- Se llevan las clases `SaludoImpl_Stub.class` y `Saludo.class` a un directorio (voy a suponer que se llama `RMI`) del servidor, que voy a suponer es un servidor HTTP al que se puede acceder como http://host

  Es decir, que las clases se pueden encontrar en http://host/RMI/
- Se lanza el servidor RMI especificando en la propiedad `java.rmi.server.codebase` el valor http://host/RMI/. El servidor registra el objeto remoto, asociado a un nombre en el registro. El registro (rmiregistry) carga las clases `Saludo.class` y `SaludoImpl_Stub.class` del codebase especificado por el servidor.
- Se lanza el cliente RMI. Solicita una referencia del objeto al registro.
- El registro devuelve una refencia (el stub) y si el cliente no encuentra el fichero con la clase intentará obtenerlo en el `codebase` especificado.
- El cliente pide el fichero con la clase. El codebase que utiliza el cliente es el que quedó asociado al stub cuando se registró el objeto remoto.
- El cliente descarga la clase `SaludoImpl_Stub.class` de `http://host/RMI/`.

La instrucción que se ha utilizado para ejecutar el servidor es:

```
java -Djava.security.policy=servidor.policy -Djava.rmi.server.codebase=http://host/RMI/
    SaludoServidor
```

Y la que se ha utilizado para ejecutar el cliente es:

```
java -Djava.security.policy=cliente.policy  SaludoCliente
```

Si se desea ver cuales son las llamadas que se están realizando desde el cliente se pueden establecer las siguientes propiedades al lanzar el servidor:

```
-Djava.rmi.server.logCalls=true  -Djava.rmi.server.logLevel=VERBOSE  -Djava.rmi.loader.
    logLevel=VERBOSE
```

Si no se dispone de un servidor HTTP o FTP se pueden obtener dos clases que se pueden utilizar como servidor de ficheros en la dirección:

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

Tras descomprimir y compilar las clases que contiene este fichero zip se puede ejecutar indicando el número de puerto y el directorio donde se encuentran los ficheros con las clases:

```
java ClassFileServer 80 ruta
```

Y ejecutar el servidor del siguiente modo:

```
java -Djava.security.policy=servidor.policy -Djava.rmi.server.codebase=http://localhost/
    SaludoServidor
```