RMI (*Java Remote Method Invocation*) es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y proporciona un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java. Si se requiere comunicación entre otras tecnologías debe utilizarse CORBAo SOAP en lugar de RMI.

RMI se caracteriza por la facilidad de su uso en la programación por estar específicamente diseñado para Java; proporciona paso de objetos por referencia (no permitido por SOAP), recolección de basura distribuida (Garbage Collector distribuido) y paso de tipos arbitrarios (funcionalidad no provista por CORBA).

A través de RMI, un programa Java puede exportar un objeto, con lo que dicho objeto estará accesible a través de la red y el programa permanece a la espera de peticiones en un puerto TCP. A partir de ese momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto.

La invocación se compone de los siguientes pasos:

- Encapsulado (marshalling) de los parámetros (utilizando la funcionalidad de serialización de Java).
- Invocación del método (del cliente sobre el servidor). El invocador se queda esperando una respuesta.
- Al terminar la ejecución, el servidor serializa el valor de retorno (si lo hay) y lo envía al cliente.
- El código cliente recibe la respuesta y continúa como si la invocación hubiera sido local.

Desde la versión 1.1 de JDK, Java tiene su propio ORB: RMI (Remote Method Invocation). A pesar de que RMI es un ORB en el sentido general, no es un modelo compatible con CORBA. RMI es nativo de Java, es decir, es una extensión al núcleo del lenguaje. RMI depende totalmente del núcleo de la Serialización de Objetos de Java, así como de la implementación tanto de la portabilidad como de los mecanismos de carga y descarga de objetos en otros

sistemas, etc.

El uso de RMI resulta muy natural para todo aquel programador de Java ya que éste no tiene que aprender una nueva tecnología completamente distinta de aquella con la cual desarrollará. Sin embargo, RMI tiene algunas limitaciones debido a su estrecha integración con Java, la principal de ellas es que esta tecnología no permite la interacción con aplicaciones escritas en otro lenguaje.

RMI como extensión de Java, es una tecnología de programación, fue diseñada para resolver problemas escribiendo y organizando código ejecutable. Así RMI constituye un punto específico en el espacio de las tecnologías de programación junto con C, C++, Smalltalk, etc.

La principal diferencia de utilizar RPC o RMI, es que RMI es un mecanismo para invocación remota de procedmientos basado en el lenguaje de programación Java que soporta interacción entre objetos, mientras que RPC no soporta esta característica.

Arquitectura

La arquitectura RMI puede verse como un modelo de cuatro capas.

Primera capa

La primera capa es la de aplicación y se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda java.rmi.Remote. Dicha interfaz se usa básicamente para «marcar» un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase UnicastRemoteObject (paquete java.rmi.server), o puede hacerse de forma explícita con una llamada al método exportObject() del mismo paquete.

Segunda capa

La capa 2 es la capa proxy, o capa stub-skeleton. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

Tercera capa

La capa 3 es la de referencia remota, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo stream (stream-oriented connection) desde la capa de transporte.

Cuarta Capa

La capa 4 es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (Java Remote Method Protocol), que solamente es «comprendido» por programas Java.

Elementos

Toda aplicación RMI normalmente se descompone en 2 partes:

- Un servidor, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- Un cliente, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

Ejemplo

Un servidor RMI consiste en definir un objeto remoto que va a ser utilizado por los clientes. Para crear un objeto remoto, se define una interfaz, y el objeto remoto será una clase que implemente dicha interfaz. Veamos como crear un servidor de ejemplo mediante 3 pasos:

- Definir la interfaz remota. Cuando se crea una interfaz remota:
    - La interfaz debe ser pública.
    - Debe heredar de la interfaz java.rmi.Remote, para indicar que puede llamarse desde cualquier máquina virtual Java.
    - Cada método remoto debe lanzar la excepción java.rmi.RemoteException en su cláusula throws, además de las excepciones que pueda manejar.

```
public interface MiInterfazRemota extends java.rmi.Remote
{
  public void miMetodo1() throws java.rmi.RemoteException;
  public int miMetodo2() throws java.rmi.RemoteException;
}
```

- Implementar la interfaz remota

```
public class MiClaseRemota
 extends java.rmi.server.UnicastRemoteObject
 implements MiInterfazRemota
{
```

```
    public MiClaseRemota() throws java.rmi.RemoteException
    {
      // Código del constructor
    }

    public void miMetodo1() throws java.rmi.RemoteException
    {
      // Aquí ponemos el código que queramos
      System.out.println("Estoy en miMetodo1()");
    }

    public int miMetodo2() throws java.rmi.RemoteException
    {
      return 5; // Aquí ponemos el código que queramos
    }

    public void otroMetodo()
    {
      // Si definimos otro método, éste no podría llamarse
      // remotamente al no ser de la interfaz remota
    }

    public static void main(String[] args)
    {
      try
      {
        MiInterfazRemota mir = new MiClaseRemota();
                        java.rmi.Naming.rebind("//"    +
  java.net.InetAddress.getLocalHost().getHostAddress() +
                              ":" + args[0] + "/PruebaRMI",
```

```
    mir);
       }
       catch (Exception e)
       {
       }
     }
    }
```

- Como se puede observar, la clase MiClaseRemota implementa la interfaz MiInterfazRemota que hemos definido previamente. Además, hereda de UnicastRemoteObject, que es una clase de Java que podemos utilizar como superclase para implementar objetos remotos.
- Luego, dentro de la clase, definimos un constructor (que lanza la excepción RemoteException porque también la lanza la superclase UnicastRemoteObject), y los métodos de la/las interfaz/interfaces que implemente.
- Finalmente, en el método main, definimos el código para crear el objeto remoto que se quiere compartir y hacer el objeto remoto visible para los clientes, mediante la clase Naming y su método rebind(…).

Nota: Hemos puesto el método main() dentro de la misma clase por comodidad. Podría definirse otra clase aparte que fuera la encargada de registrar el objeto remoto.

- Compilar y ejecutar el servidor

Ya tenemos definido el servidor. Ahora tenemos que compilar sus clases mediante los siguientes pasos:

- Compilamos la interfaz remota. Además lo agrupamos en un fichero JAR para tenerlo presente tanto en el cliente como en el servidor:

```
javac MiInterfazRemota.java
jar cvf objRemotos.jar MiInterfazRemota.class
```

- Luego, compilamos las clases que implementen las interfaces. Y para cada una de ellas generamos los ficheros Stub y Skeleton para mantener la referencia con el objeto remoto, mediante el comando rmic:

```
set CLASSPATH=%CLASSPATH%;.\objRemotos.jar;.
javac MiClaseRemota.java
rmic -d . MiClaseRemota
```

Observamos en nuestro directorio de trabajo que se han generado automáticamente dos ficheros.class (MiClaseRemota_Skel.class y MiClaseRemota_Stub.class) correspondientes a la capa stub-skeleton de la arquitectura RMI.

- Para ejecutar el servidor, seguimos los siguientes pasos:
  - Se arranca el registro de RMI para permitir registrar y buscar objetos remotos. El registro se encarga de gestionar un conjunto de objetos remotos a compartir, y buscarlos ante las peticiones de los clientes. Se ejecuta con la aplicación rmiregistry distribuida con Java, a la que le podemos pasar opcionalmente el puerto por el que conectar (por defecto, el 1099).

En el caso de Windows, se debe ejecutar:

```
start rmiregistry 1234
```

Y en el caso de Linux:

```
rmiregistry &
```

- Por último, se lanza el servidor:

```
java -Djava.rmi.server.hostname=127.0.0.1 MiClaseRemota 1234
```

- Crear un cliente RMI

Vamos ahora a definir un cliente que accederá a el/los objeto/s remoto/s que creemos. Para ello seguimos los siguientes pasos:

- Definir la clase para obtener los objetos remotos necesarios

La siguiente clase obtiene un objeto de tipo MiInterfazRemota, implementado en nuestro servidor:

```
public class MiClienteRMI
{

 private MiClienteRMI(){};

 public static void main(String[] args)
 {
   try
   {
                        MiInterfazRemota     mir     =
(MiInterfazRemota)java.rmi.Naming.lookup("//" +
                             args[0] + ":" + args[1] +
"/PruebaRMI");

      // Imprimimos miMetodo1() tantas veces como devuelva
miMetodo2()
     for (int i=1;i<=mir.miMetodo2();i++)
         mir.miMetodo1();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }
}
```

Como se puede observar, simplemente consiste en buscar el objeto remoto en el registro RMI de la máquina remota. Para ello usamos la clase Naming y su método lookup(…).

- Compilar y ejecutar el cliente

Una vez que ya tenemos definido el cliente, para compilarlo hacemos:

```
set CLASSPATH=%CLASSPATH%;.\objRemotos.jar;.
javac MiClienteRMI.java
```

Luego, para ejecutar el cliente hacemos:

```
java MiClienteRMI 127.0.0.1 1234
```

Se debe poder acceder al fichero Stub de la clase remota. Para ello, o bien lo copiamos al cliente y lo incluimos en su CLASSPATH, o lo eliminamos del CLASSPATH del servidor e incluimos su ruta en el java.rmi.codebase del servidor (si no se elimina del CLASSPATH del servidor, se ignorará la opción java.rmi.codebase, y el cliente no podrá acceder al Stub).. Si echamos un vistazo a la ventana donde está ejecutándose el servidor RMI, veremos como se ha encontrado el objeto remoto y ejecutado sus métodos.