Un ejemplo tonto de RMI (Remote Method Invocation o Invocación de Métodos Remotos)

Al igual que con los RPC (Remote Procedure Call o Llamada a Procedimientos Remotos) de C en linux, es posible hacer que un programa en **java** llame a métodos de objetos que están instanciados en otro programa distinto, incluso que estén corriendo en otra máquina conectada en red. Estos métodos, aunque los llamemos desde este ordenador, se ejecutan en el otro. Como se comentó en el RPC, este sistema tiene la ventaja de que si, por ejemplo, tenemos un ordenador capaz de realizar cuentas muy deprisa y otro capaz de dibujar gráficos maravillosos y además necesitamos hacer un programa con muchas cuentas y muchos gráficos, podemos implementar en el ordenador de las cuentas aquellas clases que echan cuentas, en el ordenador de gráficos aquellas clases de pintado y hacer que cada ordenador haga lo que mejor sabe hacer. Cuando al hacer un gráfico necesitemos echar cuentas, llamaremos a la clase remota que echa las cuentas, y estas se harán en el ordenador de las cuentas, devolviendo el resultado al de los gráficos.

Vamos a hacer un ejemplo muy tonto de **rmi**, para introducirnos en cómo funciona. Harémos una clase

capaz de sumar ni más ni menos que dos enteros. Haremos que esta clase quede registrada como objeto remoto, por lo que podrá ser llamada desde otros ordenadores. Harémos un cliente capaz de llamar a esa clase para sumar 2 + 3.

¿Qué cosas tenemos que hacer?

Vamos a ver qué clases necesitamos para hacer nuestros clientes y servidores de **rmi**, sin extendernos demasiado. Luego veremos cada uno de estos puntos por detallado.

- **InterfaceRemota**. Es una interface java con todos los métodos que queramos poder invocar de forma remota, es decir, los métodos que queremos llamar desde el cliente, pero que se ejecutarán en el servidor.
- **ObjetoRemoto**. Es una clase con la implementación de los métodos de **InterfaceRemota**. A esta clase sólo la ve el servidor de **rmi**.
- **ObjetoRemoto_Stub**. Es una clase que implementa **InterfaceRemota**, pero cada método se encarga de hacer una llamada a través de red al **ObjetoRemoto** del servidor, esperar el resultado y devolverlo. Esta clase es la que ve el cliente y no necesitamos codificarla, java lo hace automáticamente para nosotros a partir de **ObjetoRemoto**.

En el pc/máquina servidor de rmi deben correr dos programas

- **rmiregistry**. Este es un programa que nos proporciona java (está en JAVA_HOME/bin, siendo JAVA_HOME el directorio en el que tengamos instalado java). Una vez arrancado, admite que registremos en él objetos para que puedan ser invocados remotamente y admite peticiones de clientes para ejecutar métodos de estos objetos.
- **Servidor**. Este es un programa que debemos hacernos nosotros. Debe instanciar el **ObjetoRemoto** y registrarlo en el **rmiregistry**. Una vez registrado el **ObjetoRemoto**, el servidor no muere, sino que queda vivo. Cuando un cliente llame a un método de **ObjetoRemoto**, el código de ese método se ejecutará en este proceso.

En el pc/máquina del cliente debe correr el programa

- **Cliente**. Este programa debemos hacerlo nosotros. Pide al **rmiregistry** del pc/máquina servidor una referencia remota al **ObjetoRemoto**. Una vez que la consigue (en realidad obtiene un **ObjetoRemoto_Stub**), puede hacer las llamadas a sus métodos. Los métodos se ejecutarán en el Servidor, pero Cliente quedará bloqueado en cada llamada hasta que Servidor termine de ejecutar el método.

La interface de la clase remota

Lo primero que tenemos que hacer es una interface con

los métodos que queremos que se puedan llamar remotamente. Esta interface sería como la siguiente:

```
import java.rmi.Remote;
public interface InterfaceRemota extends Remote
{
   public int suma (int a, int b) throws
java.rmi.RemoteException;
}
```

Tiene que heredar de la interface **Remote** de java, si no el objeto no será remoto. Añade además los métodos que queramos, pero todos ellos deben lanzar la excepción **java.rmi.RemoteException**, que se producirá si hay algún problema con la comunicación entre los dos ordenadores o cualquier otro problema interno de rmi o en el servidor.

Todos los parámetros y valores devueltos de estos métodos deben ser tipos primitivos de java o bien clases que implementen la interface **Serializable** de java. De esta forma, tanto los parámetros como el resultado podrán "viajar" por red del cliente al servidor y al revés. También se admiten los que implementen la interface **Remote**, pero ya lo veremos más adelante. De momento, para este ejemplo, nos conformamos con tipos primitivos, en concreto, unos int.

El objeto remoto

Hay que hacer una clase que implemente esa

InterfaceRemota, es decir, que tenga los métodos que queremos llamar desde un cliente **rmi**. El servidor de **rmi** se encargará de instanciar esta clase y de ponerla a disposición de los clientes. Esa clase es la que llamamos "**objeto remoto**".

Esta clase remota debe implementar la interface remota que hemos definido (y por tanto implementará también la interface **Remote** de java), pero también debe hacer ciertas cosas bien. Debe definir métodos como **hashCode()**, **toString()**, **equals()**, etc de forma adecuada a un objeto remoto. También debe tener métodos que permita obtener referencias remotas, etc, etc. Es decir, una serie de cosas más o menos complejas y de las que afortunadamente no tenemos que preocuparnos. La forma sencilla de hacer esto es hacer que la clase herede de **UnicastRemoteObject**. Sería más o menos la siguiente:

```
import java.io.Serializable;
public class ObjetoRemoto extends UnicastRemoteObject
implements InterfaceRemota
{
   public int suma(int a, int b)
   {
      System.out.println ("sumando " + a + " + " + b + "...");
      return a+b;
   }
}
```

Otra opción es no hacerlo heredar de **UnicastRemoteObject**, pero luego la forma de registrarlo varía un poco y además debemos encargarnos de implementar adecuadamente todos los métodos indicados (y algunos más).

La clase de stubs

Una vez compilado y que obtenemos nuestro fichero **ObjetoRemoto.class**, necesitamos crear la "clase de stubs". ¿Qué es esto?. Para que desde un programa en un ordenador se pueda llamar a un método de una clase que está en otro ordenador, está claro que de alguna manera se debe enviar un mensaje por red de un ordenador a otro, indicando que se quiere llamar a determinado método de determinada clase y además pasar los parámetros de dicha llamada. Una vez ejecutado el método, el ordenador que lo ha ejecutado debe enviar un mensaje con el resultado. La clase de stubs es una clase con los mismos métodos que nuestro **ObjetoRemoto**, pero en cada uno de esos métodos está codificado todo el tema del envio del mensaje por red y recepción de la respuesta.

Afortunadamente para nosotros, no tenemos que codificar todo este rollo de mensajes de ida y vuelta. java nos proporciona una herramienta llamada **rmic**, también en JAVA_HOME/bin, a la que le pasamos la clase **ObjetoRemoto** y nos devuelve la clase de stubs **ObjetoRemoto_Stub**. Sólo tenemos que poner en la

variable de entorno **CLASSPATH** el directorio en el que está nuestra clase **ObjetoRemoto** y ejecutar **rmic**

```
$ set CLASSPATH=C:\Documents and Settings\Chuidiang
$ rmic chuidiang.ejemplos.rmi.suma.ObjetoRemoto
```

Esto generarará un **ObjetoRemoto_stubs.class** y, en versiones antiguas de java, un **ObjetoRemoto_Skel.class**. El primero debe estar visible tanto por el cliente como por el servidor, es decir, deben aparecer en el **CLASSPATH** de ambos. Eso implica que debe estar situado en el servidor en un sitio público al que el cliente tenga acceso o que se debe suministar una copia al cliente. El **ObjetoRemoto_Skel.class** se generá por defecto y sólo es útil para clientes con java anterior a la versión 1.2. Para los java más modernos no tiene utilidad.

Lanzar rmiregistry

Antes de registrar el objeto remoto, debemos lanzar, desde una ventana de ms-dos o una shell de linux el programa **rmiregistry**. Este programa viene con java y está en el directorio **bin** de donde tengamos instalado java y símplemente tenemos que arrancarlo sin parámetros. Es importante al arrancarlo que la variable **CLASSPATH** no tenga ningún valor que permita encontrar nuestros objetos remotos, por lo que se aconseja borrarla antes de lanzar **rmiregistry**. Si **rmiregistry** esta en el **PATH** de búsqueda de ejecutables

(o nos hemos situado en el directorio en el que está), se lanzaría de la siguiente manera.

```
c:\> set CLASSPATH=
c:\> rmiregistry

$ unset CLASSPATH
$ rmiregistry
```

Una vez arrancado **rmiregistry**, podemos registrar en él nuestros objetos remotos.

Nuestro Servidor de rmi

Lo siguiente que tenemos que hacer es ejecutar un programa java que instancie y registre el objeto remoto. Este programa java debe hacer lo siguiente

- Indicar cual es el path en el que rmiregistry puede encontrar la clase correspondiente al objeto remoto. En nuestro caso, el path en el que se puede encontrar **ObjetoRemoto_Stub.class**. Dicho path se da en formato URL, por lo que no admite espacios ni caracteres extraños (Quizás admita los típicos %20% que se ven en el navegador cuando hay espacios en la url, pero no he probado). El path, en formato URL, puede ser algo como esto "**file:/D:/users/javier/prueba_servidor**/".

  El código para indicar esto es el siguiente:

  ```
  System.setProperty ("java.rmi.server.codebase",
  ```

"file:/D:/users/javier/prueba_servidor/");

Consiste en fijar ua propiedad de nombre **java.rmi.codebase** con el path donde se encuentran los ficheros **.class** remotos. Es imortante la barra del final, porque si no, fallará luego el servidor.

Aquí también podemos poner "**http://un_host/un_path/**" si queremos que nuestra clase **ObjetoRemoto_Stub.class** esté accesible desde un servidor web. En cualquier caso, el path que pongamos aquí debe estar accesible para el programa **rmiregistry** cuando lo lancemos.

- Instanciar una clase remota y luego registrarla en **rmiregistry**. Eso es sencillo.

  ObjetoRemoto objetoRemoto = new ObjetoRemoto();
  Naming.rebind ("ObjetoRemoto", objetoRemoto);

  La instanciación no tiene problemas. Para registrarla hay que llamar al método estático **rebind()** de la clase **Naming**. Se le pasan dos parámetros. Un nombre para poder identificar el objeto y una instancia del objeto. El nombre que hemos dado debe conocerlo el cliente, para luego poder pedir la instancia por el nombre. El método **rebind()** registra el objeto en **rmiregistry**. Si ya estuviera registrado, lo sustituye por el que acabmos de pasarle.

  En donde pone "ObjetoRemoto" podría ponerse

"//localhost/ObjetoRemoto" o bien "//un_host/ObjetoRemoto". Si no ponemos nada se sobreentiende localhost. Si ponemos algo, es el host donde se buscará rmiregistry para registrar el objeto. El host que pongamos aquí debe ser el host en el que arranquemos **rmiregistry**.

Ya está todo lo que tiene que hacer el programa que registra el objeto.

El Cliente de RMI

Ahora sólo tenemos que hacer el programa que utilice este objeto de forma remota. Los pasos que debe realizar este programa son los siguientes:

- Pedir el objeto remoto al servidor de rmi. El código para ello es sencillo

  InterfaceRemota objetoRemoto = (InterfaceRemota)Naming.lookup ("//host_servidor/ObjetoRemoto");

  Simplemente se llama al método estático **lookup()** de la clase **Naming**. Se le pasa a este método la URL del objeto. Esa URL es el nombre (o IP) del host donde está arrancado **rmiregistry** y por último el nombre con el que se registró anteriormente el objeto (en mi caso **ObjetoRemoto**).

  Este método devuelve un **Remote**, así que debemos

hacer un "cast" a **InterfaceRemota** para poder utilizarlo. El objeto que recibimos aquí es realmente un **ObjetoRemoto_Stub**. Por ello, **ObjetoRemoto_Stub.class** debe estar en el CLASSPATH del cliente.

- Usarlo. Ya podemos llamar al método de **suma()**.

  System.out.print ("2 + 3 = ");
  System.out.println (objetoRemoto.suma(2, 3));

Para que el código del cliente compile necesita ver en su classpath a **InterfaceRemota.class**. Para que además se ejecute sin problemas necesita además ver a **ObjetoRemoto_Stubs.class**, por lo que estas clases deben estar accesibles desde el servidor o bien tener copias locales de ellas.

El código de ejemplo

En RMI_SIMPLE.zip tienes todos los fuentes y clases de cliente y servidor compilados. Para que te funcione tal cual, tendrás que crear en la unidad D: el directorio D:\users\javier\JAVA\RMI_SIMPLE y desempaquetar ahí. Se crearán dos directorios: src_cliente y src_servidor.

Si lo desempaquetas en otro directorio, en el lado del servidor tendrás que editar **Servidor.java** y cambiar el path de la propiedad "**java.rmi.server.codebase**" para que coincida donde tengas tu directorio src_servidor. Luego debes compilar este **Servidor.java**. Recuerda que,

para evitar problemas, es mejor que lo pongas todo en directorios que no tengan espacios en blanco en los nombres, como el dichoso "Documents and Settings" de windows.

Si no arrancas Servidor, Cliente y **rmiregistry** en el mismo ordenador, tendrás que cambiar también los "localhost" que encuentres en Servidor.java y Cliente.java para que indiquen el host donde está arrancado **rmiregistry**.

Una vez preparado todo, abre tres ventanas de ms-dos. En cada una de ellas arrancas para el servidor

c:\> rmiregistry

c:\> cd path_desempaquetado\src_servidor
c:\> java chuidiang.ejemplos.rmi.suma.Servidor

y en el cliente

c:\> cd path_desempaquetado\src_cliente
c:\> java chuidiang.ejemplos.rmi.suma.Cliente

Al hacer esto, verás que en la ventana de servidor sale un "Sumando 2 + 3 = ..." y en la del cliente "2 + 3 = 5".

Ahora podemos seguir aprendiendo más cosas sobre **rmi**.

- Pasar objetos Serializables y Remote como parámetros en rmi.
- Carga dinámica de clases. Seguridad en rmi con RMISecurityManager y java.policy.

Estadísticas y comentarios

Numero de visitas desde el 4 Feb 2007:

- Esta pagina este mes: 406
- Total de esta pagina: 185580
- Total del sitio: 21334194