RMI, ¿que és? + Un ejemplo sencillo con RMI

Después de unas no tan merecidas vacaciones de fin de año, retomaré la costumbre de andar posteando cosillas en el blog. Iniciemos el año con un pequeño calentamiento en Java, tratando un tema muy sencillo pero muy útil: RMI o Invocación de Métodos Remotos (por sus siglas en inglés Remote Method Invocation).

Una definición friki podría ver al RMI como una forma de utilizar funciones de un programa, desde otro programa residente en otro equipo. Es decir, permitir a un programa (A) utilizar las funciones de otro programa (B), solo que el programa B está en otro equipo. Es ejecutar algo por allí, pero que parezca que esté por acá.
Pero, ¿que ventajas trae esto? umm, a simple vista las ventajas son obvias: se puede distribuir un programa en varios equipos; así podemos dividir la carga de procesos en varios hosts, y haciendo el programa más rápido. Esto es útil si el equipo "cliente" no tiene una suficiente capacidad de procesamiento, por ejemplo.

Ahora, una definición ~~wiki~~ más técnica podría ser está (utilicemos las ventajas del GFDL):

RMI es un mecanismo ofrecido en Java para invocar un método remotamente. Al ser RMI parte estándar del entorno de ejecución Java usarlo provee un mecanismo simple en una aplicación distribuida que solamente necesita comunicar servidores codificados para Java. Si se requiere comunicarse con otras tecnologías debe usarse CORBA o SOAP en lugar de RMI.

Al estar específicamente diseñado para Java RMI provee pasaje por referencia de objetos (cosa que no hace SOAP), "recolección de basura" distribuida y pasaje de tipos arbitrarios (funcionalidad no provista por CORBA).

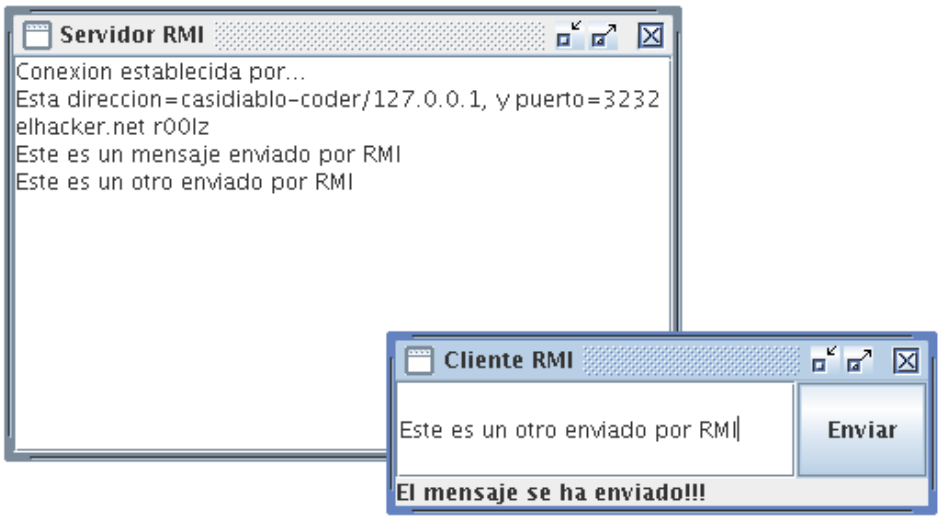
Por medio de RMI, un programa Java puede exportar un objeto. A partir de esa operación este objeto está disponible en la red, esperando conexiones en un puerto TCP. Un cliente puede entonces conectarse e invocar métodos. La invocación consiste en el "marshaling" de los parámetros (utilizando la funcionalidad de "serialización" que provee Java), luego se sigue con la invocación del método (cosa que sucede en el servidor). Mientras esto sucede el llamador se queda esperando por una respuesta. Una vez que termina la ejecución el valor de retorno (si lo hay) es serializado y enviado al cliente. El código cliente recibe este valor como si la invocación hubiera sido local.

> *Este fragmento de texto fue extraido del artículo RMI de la Wikipedia; por lo tanto está bajo los terminos de la Licencia de Documentación Libre de GNU (véase su uso).*

Ahora bien, vamos a ver un ejemplo explicado de RMI. Lo primero es implementar el servidor. Para ello se debe crear una "interfaz" que enumere las funciones provistas por el objeto a ser exportado por la red. Esta "interfaz" (en este ejemplo la interfaz se llama **InterfazReceptorMensajes.java**) es compartida entre cliente y servidor. Del lado servidor es necesario que una de las clases (que hace de servidor como tal) implemente dicha interfaz. Del lado cliente el llamador accede a la misma interfaz, pero debajo hay un "stub" que da comienzo a la llamada vía la red. Para crear este "stub" es necesario utilizar el "compilador" rmi llamado **rmic**.

> *El lado cliente y el lado servidor deberán tener ambos acceso a la definición de las clases utilizadas para el intercambio de parámetros y de valores de retorno. Para esto Java tiene la capacidad de dinámicamente obtener las clases que se necesiten desde un servidor.*

El programa a explicar consta de dos GUI's, una para el cliente y una para el servidor. El cliente tiene una caja de texto donde se escriben los mensajes a enviar; dichos mensajes se envian utilizando un método remoto. Mientras, el servidor posee un area de texto donde se muestran los mensajes que recibe de los posibles clientes. Esta es la salida del los programas Servidor/Cliente:



Expliquemos entonces, paso a paso, el código fuente de la aplicación. Comencemos con la interfaz **InterfazReceptorMensajes.java**. Fíjate que no se declara una clase como tal (public class), sino una interfaz; y que dicha interfaz hereda de la clase Remote del paquete java.rmi. Además se declara el método recibirMensaje, más no se define. Y es necesario indicar que dicho método puede lanzar una excepción RemoteException.

```
import java.rmi.*;

public interface InterfazReceptorMensajes extends Remote
{
      //Este es el metodo que implementará el servidor
      void recibirMensaje(String texto) throws RemoteException;
}
```

Antes de ver el mecanismo de funcionamiento del servidor, veamos como funciona la clase GUIServidor, que simplemente nos sirve como salida para los procesos que realice el servidor. Lo único a resaltar en esta clase es el método anadirEntradas que nos permite añadir texto en el area de texto.

```
import javax.swing.*;

public class GUIServidor extends JFrame {

	private JTextArea areaTexto;

	public GUIServidor() {
		super("Servidor RMI");
		areaTexto = new JTextArea();
		areaTexto.setEditable(false);
		getContentPane().add(new JScrollPane(areaTexto));

		setSize(600, 400);
		setVisible(true);
	}

	public void anadirEntradas(String texto) {
		areaTexto.append(texto + "\n");
	}
}
```

Ahora veamos la clase **RmiServidor.java** que, como su nombre lo indica, maneja la parte del servidor. Es necesario importar ciertos paquetes y clases. Por ejemplo la clase InetAddress del paquete java.net, que sirve para capturar la dirección IP de la máquina local. Todo el paquete java.rmi, java.rmi.registry y java.rmi.server.

```
import java.net.InetAddress;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import javax.swing.JFrame;
```

En la declaración de la clase es necesario heredar de la clase UnicastRemoteObject del paquete java.rmi.server, e implementar la interfaz InterfazReceptorMensajes.

```
public class RmiServidor extends UnicastRemoteObject implements
		InterfazReceptorMensajes {
```

Ahora declaramos los objetos que vayamos a utilizar. Declaramos el objeto ventana de la clase GUIServidor, para la parte visual. Una variable entera para almacenar el valor del puerto de escucha. Un string que almacenará la dirección IP del equipo servidor. Y un objeto Registry, del paquete java.rmi.registry.

```
	private static GUIServidor ventana;

	private int estePuerto;

	private String estaIP;

	private Registry registro;
```

El constructor de la clase debe especificar el tipo de excepciones que puede lanzar, en este caso RemoteException. Después obtenemos la dirección IP del servidor utilizando el método estático getLocalHost() de la clase InetAddress; y la almacenamos en el objeto estaIP. Se define el puerto de escucha en 3232. Y se despliega información en la GUI del servidor, acerca de la dirección IP y el puerto de escucha:

```
public RmiServidor() throws RemoteException {
	try {
		// obtener la direccion de este host.
		estaIP = (InetAddress.getLocalHost()).toString();
	} catch (Exception e) {

		throw new RemoteException("No se puede obtener la direccion IP.");
	}
	estePuerto = 3232; // asignar el puerto que se registra
	ventana.anadirEntradas("Conexion establecida por...\nEsta direccion="
			+ estaIP + ", y puerto=" + estePuerto);
```

Luego se inicializa el objeto registro, utilizando la instrucción LocateRegistry.createRegistry(estePuerto); que indica al registro el puerto por el cual va a recibir conexiones, de la siguiente forma:

```
	try {

		// crear el registro y ligar el nombre y objeto.
		registro = LocateRegistry.createRegistry(estePuerto);
		registro.rebind("rmiServidor", this);
	} catch (RemoteException e) {
		throw e;
	}
}
```

El método recibir mensajes aquí definido, es el mismo que se declaró en la interfaz **InterfazReceptorMensajes.java**. Este método recibirá un string y lo enviará al método anadirEntradas de la GUI, para desplegar dicho mensaje en pantalla.

```
public void recibirMensaje(String texto) throws RemoteException {
	ventana.anadirEntradas(texto);
}
```

En el método main se crea una instancia de la clase RmiServidor, lo que lanza el programa servidor, y lo pone en escucha:

```
public static void main(String[] args) {
	JFrame.setDefaultLookAndFeelDecorated(true);
	ventana = new GUIServidor();
	ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	try {
		new RmiServidor();
	} catch (Exception e) {
		e.printStackTrace();
		System.exit(1);
	}
}
```

Para compilar adecuadamente la parte del servidor y crear el “stub”, es decir: la

interfaz gráfica (**GUIServidor.java**), la interfaz receptora de mensajes (**InterfazReceptorMensajes.java**) y el servidor (**RmiServidor.java**), debes utilizar los siguientes comandos:

```
javac *.java
rmic RmiServidor
```

> *Nota: El uso del comando* **rmic** *es necesario para crear el stub, aunque esto solo es en algunas versiones de Java. He probado con la version 1.5 y no he tenido que utilizarlo para que el programa funcione correctamente. Pero es recomendable usarlo en cualquiera de los casos, para evitar posibles errores.*
> *Es de notar también que al utilizar el comando* **rmic***, este crea un archivo de clase aparte del convencional (que se gunera con el comando* **javac***), este archivo (para este ejemplo) toma el nombre de:* **RmiServidor_Stub.class***, y es el que Java utiliza para manipular el stub del servidor.*

Ahora veamos la parte cliente. Como ya hemos dicho antes la interfaz **InterfazReceptorMensajes.java** se comparte entre cliente y servidor; por lo cual solo nos queda explicar la clase RmiCliente.java. En dicha clase es necesario importar los paquetes java.rmi y java.rmi.registry, además de los necesarios para la GUI:

```
import java.rmi.*;
import java.rmi.registry.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

La clase RmiCliente, aparte de los necesarios para la interfaz gráfica, declara: un objeto de la interfaz InterfazReceptorMensajes, un objeto de la clase Registry y dos objetos String en donde colocaremos la dirección IP del servidor a conectarnos y el puerto de escuha. En este caso, en la dirección IP del servidor coloco la loop IP (127.0.0.1), ya que las pruebas las realizo en el mismo equipo, es decir, servidor y cliente corriendo en la misma máquina:

```
public class RmiCliente extends JFrame implements ActionListener {
	private JTextField cajaEnviar;
	private JButton botonEnviar;
	private JLabel estado;
	private static InterfazReceptorMensajes rmiServidor;
	private static Registry registro;
	private static String direccionServidor = "127.0.0.1";
	private static String puertoServidor = "3232";
```

El constructor inicializa y configura los objetos de la GUI:

```
	public RmiCliente() {
		super("Cliente RMI");
		getContentPane().setLayout(new BorderLayout());
		cajaEnviar = new JTextField();
		cajaEnviar.addActionListener(this);
		botonEnviar = new JButton("Enviar");
		botonEnviar.addActionListener(this);
```

```
        estado = new JLabel("Estado...");

        getContentPane().add(cajaEnviar);
        getContentPane().add(botonEnviar, BorderLayout.EAST);
        getContentPane().add(estado, BorderLayout.SOUTH);

        setSize(300, 100);
        setVisible(true);
    }
```

El método actionPerformed captura cualquier evento, ya sea de la caja de texto o del boton, y envía el mensaje como parámetro al método enviarMensaje.

```
    public void actionPerformed(ActionEvent e) {
        if (!cajaEnviar.getText().equals("")) {
            enviarMensaje(cajaEnviar.getText());
            cajaEnviar.setText("");
        }
    }
```

El método conectarseAlServidor inicializa el objeto registro (de la clase Registry), utilizando para ello la instrucción LocateRegistry.getRegistry(), que recibe como argumentos la dirección IP del servidor y el puerto de escucha. Luego inicializa el objeto rmiServidor (de la clase InterfazReceptorMensajes) utilizando el método lookup de de la clase Registry. Fíjate que al utilizar el método lookup se le pasa como argumento el string “rmiServidor”, pero ¿porqué “rmiServidor”?; la respuesta es sencilla: cuando implementamos el servidor y lo ponemos a la escucha, utilizamos el comando registro.rebind("rmiServidor", this); esto es como colocarle nombre al registro de escucha, y por ello al crear el registro cliente es necesario indicar dicho nombre.

```
    private static void conectarseAlServidor() {
        try {
            // obtener el registro
            registro = LocateRegistry.getRegistry(direccionServidor,
                        (new Integer(puertoServidor)).intValue());
            // creando el objeto remoto
            rmiServidor = (InterfazReceptorMensajes) (registro
                        .lookup("rmiServidor"));
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (NotBoundException e) {
            e.printStackTrace();
        }
    }
```

El método enviarMensaje es el que realmente utiliza RMI. Dicho método utiliza la función recibirMensaje del objeto rmiServidor, es decir: utiliza el método remoto del servidor a través de la interfaz.

```
    private void enviarMensaje(String mensaje) {
        estado.setText("Enviando " + mensaje + " a " + direccionServidor + ":"
                    + puertoServidor);
```

```
		try {
			// llamando el metodo remoto
			rmiServidor.recibirMensaje(mensaje);
			estado.setText("El mensaje se ha enviado!!!");
		} catch (RemoteException re) {
			re.printStackTrace();
		}
	}
```

El método main utiliza la función conectarseAlServidor para crear la conexión, y luego crea el objeto de la GUI:

```
	static public void main(String args[]) {
		JFrame.setDefaultLookAndFeelDecorated(true);
		conectarseAlServidor();
		RmiCliente ventana = new RmiCliente();
		ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
```

El proceso de compilación del cliente es normal, no tienes que utilizar otros comandos. Por cuestiones de comodidad he realizado las pruebas en un solo equipo, pero lo ideal es hacerlo en equipos distintos, y mas aún: que tengan SO diferentes. Por ejemplo correr el servidor en un equipo con Linux, y utilizar sus métodos remotos desde uno con Windows; ahí se centra el verdadero potencial del Java y RMI: la multiplataforma.

Bien eso es todo. Imagino que ya descargaste el código fuente de los ejemplos. Dudas, comentarios, sugerencias e insultos: castidiablo[at]gmail.com.