PROGRAMACIÓN CONCURRENTE Y DE TIEMPO REAL

GUIÓN DE PRÁCTICAS CON REMOTE METHODS INVOCATION

PROGRAMACIÓN AVANZADA DE RMI EN JAVA

Nota: Para su comodidad, todos los códigos objeto de anáisis en este guión se encuentran disponibles para descarga en las carpeta de códigos correspondientes (rmi/callback).

A continuación se desarrolla en forma de guiónel conjunto de tareas necesarias para realizar la programación de aplicaciones RMI completas utilizando la tecnología RMI de Java que incorporan características algo más avanzadas, tales como *callback* de cliente, gestión de la seguridad y descarga dinámica de clases. Siga la secuencia de instrucciones que se le proporcionan.

1. *Call-Back* de Cliente

Tal y como ha sido presentada hasta ahora, la tecnología RMI tiene carácter unidireccional, en el sentido de que es el cliente quien realiza una peticiïón al servidor, que la procesa y devuelve el resultado. Actualmente, y en entornos distribuidos, la diferencia entre objetos clientes y objetos servidores es cada vez más difusa, y todos los objetos actúan con caráccter de cliente o de servidor según les es necesario. En otras palabras, en ocasiones serán un "servidor" quien tras procesar una llamada a método de un "cliente", efectúen a su vez una llamada sobre uno de los métodos del "cliente". A esto se le denomina técnicamente *callback* de cliente.

Como es natural, para poder desarrollar esto son necesarias dos cosas:

a) Que el objeto servidor lleve una bitácora de los clientes que eventualmente pueden estar esperando un *callback* como consecuencia de algún cálculo que debe desarrollar.

b) El cliente debe ofrecer métodos para *callback* que un servidor pueda invocar.

Todo ello pasa por desarrollar un soporte semántico desde el lado del cliente idéntico al que ya conocemos y sabemos desarrollar desde el lado del servidor. De esta forma, definiremos una interfaz del lado del cliente que ofrezca los métodos sobre los cuales puede un objeto servidor realizar un *callback*. Implementaremos esa interfaz para darle el contenido semántico concreto que nos pueda interesar, y realizaremos una aplicación cliente que registre objetos cliente en el objeto servidor que luego debe darles el *callback*.

Comenzamos por escribir la interfaz del cliente:

```
/*@author Antonio Tomeu
*@version 1.0
*Clase que define el metodo que via callback un objeto servidor puede
invocar
*sobre un objeto cliente
*Nota: Adaptado de Liu, M. Computacion Distribuida
**/

import java.rmi.*;
```

```
    public interface InterfazCliente
        extends java.rmi.Remote
    {
        public String Senial(String mensajito) throws
java.rmi.RemoteException;
    }
```

Observe que lo único que hacemos es especificar una interfaz que permitirá, una vez implementada y a través de `rmic`, generar el soporte de *proxies* necesario (tanto stub como skeleton) para que el servidor puede invocar a través de *callback* un método de un objeto cliente. En el caso que nos ocupa, cuando el objeto servidor ejecute el *callback*, lo hará enviado un `String` al objeto cliente, el cual devolverá a su vez otro `String` al servidor.

```
public String Senial(String mensajito)
```

Corresponde a continuación implementar la anterior interfaz:

```
/*@author Antonio Tomeu
*@version 1.0
*Clase que implementa el metodo que via callback un objeto servidor
puede invocar
*sobre un objeto cliente
*Nota: Adaptado de Liu, M. Computacion Distribuida
**/

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

public class ImpInterfazCliente
    extends UnicastRemoteObject
        implements InterfazCliente
{
    public ImpInterfazCliente() throws RemoteException
    {
        super();
    }

    public String Senial(String mensajito)
    {
        String Respuesta = "Senial recibida: "+mensajito;
        System.out.println(Respuesta);
        return (Respuesta);
    }
}
```

En la implementación, el objeto cliente hace algo tan simple como tomar la cadena enviada desde el servidor a través del *callback*, y utilizarla para componer una cadena de mensaje que retorna al servidor. Por fin, desarrollamos una aplicación cliente que creará un objeto cliente y lo registrará para callback en un objeto servidor.

```
/*@author Antonio Tomeu
*@version 1.0
*Clase que implementa un cliente
*Nota: Adaptado de Liu, M. Computacion Distribuida
**/

import java.io.*;
```

```
import java.rmi.*;

public class Cliente
{
    public static void main(String[] args)
    {
        //contacto al servidor
        try
        {
            InterfazServidor L =
(InterfazServidor)Naming.lookup("rmi://localhost:2001/callback");
            System.out.println("El servidor contesta "+L.Hola());
            InterfazCliente ObjetodeCallBack = new
ImpInterfazCliente();
            System.out.println("objeto callback cliente creado...");
            //registro de callback
            L.RegistroCallBack(ObjetodeCallBack);
            System.out.println("CallBack registrado...");
            try{
                Thread.sleep(1000);
               } catch (InterruptedException e){}

            L.QuitarCallBack(ObjetodeCallBack);
            System.out.println("CallBack eliminado de registro...");

        }catch (Exception e) {System.out.println("problema en
cliente...");}
        System.out.println("Cliente finalizando...");
    }//main
}
```

En este caso, el cliente comienza por localizar a un objeto servidor en la máquina y puerto especificado como parámetro al método `lookup`. Inmediatamente se invoca a un método de la interfaz de tal objeto servidor, a efectos de validar la conexión de cara al usuario del objeto cliente mediante la instrucción siguiente:

```
System.out.println("El servidor contesta "+L.Hola());
```

Posteriormente se crea un objeto cliente denominado `ObjetodeCallBack`, que deberáa ser registrado en el objeto servidor, a efectos de que este puede efectuar *callbacks* sobre la interfaz compuesta por los métodos del objeto `ObjetodeCallBack`. El punto principal de interés en lo que nos resta por analizar del código del cliente es precisamente la realización de tal registro en el objeto servidor, utilizando un método específicamente dispuesto para ello en la interfaz del mismo: `RegistroCallBack()`.

Desde el punto de vista del cliente, no nos interesa como se realiza ese registro. Lo único importante es que a partir de la instrucción en cuestón, el programador sabe que el objeto servidor dispone de una referencia a un objeto que implementa la clase `ImpInterfazCliente`, y que los métodos que conforman la misma puede ser invocados vía *callback* desde ese objeto servidor, a los efectos que el programador desee: sincronizaciïón retroalimentación de información, etc. La instrucción que efectúa el registro en detalle es

```
L.RegistroCallBack(ObjetodeCallBack);
```

El resto del código cliente no tiene demasiado interés; duerme al programa principal durante unos instantes, durante los cuales puede llegar o no un *callback*, elimina el objeto *callback* registrado en el servidor, y termina.

Es necesario desarrollar ahora el lado del servidor. Comenzamos por la escritura de la interfaz:

```
/*@author Antonio Tomeu
*@version 1.0
*Clase que define la interfaz de objetos servidores remotos
*Nota: Adaptado de Liu, M. Computacion Distribuida
**/

import java.rmi.*;

    public interface InterfazServidor
        extends Remote
    {
    public String Hola() throws RemoteException;
    public void RegistroCallBack(InterfazCliente ObjetoCallBack)
throws RemoteException;
    public void QuitarCallBack(InterfazCliente ObjetoCallBack) throws
RemoteException;
    }
```

la cual ofrece una interfaz de cuatro métodos, de los cuales son los dos últimos los que revisten un especial interés:

```
       public void RegistroCallBack(InterfazCliente ObjetoCallBack)
                        throws RemoteException;
     public void QuitarCallBack(InterfazCliente ObjetoCallBack) throws
                             RemoteException;
```

Observe que ambos están parametrizados por un objeto de clase `InterfazCliente`, o lo que es lo mismo, proporcionan a aquél objeto que implemente a `InterfazServidor` una referencia a objetos que a su vez implementarán a la interfaz `InterfazCliente`. Es decir, podremos hacer *callback,* utilizando esa referencia a métodos como `String Senial(String mensajito)`. Adicionalmente, un potencial objeto cliente que contacte con un objeto servidor que implemente a `InterfazServidor`, podrá registrar y eliminar objetos para *callback*. Veamos cómo, implementando la interfaz:

```
/*@author Antonio Tomeu
*@version 1.0
*Clase que implementa la interfaz de objetos servidores remotos
*Nota: Adaptado de Liu, M. Computacion Distribuida
**/

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class ImpInterfazServidor
    extends UnicastRemoteObject
        implements InterfazServidor
{
    private Vector ListaClientes;
```

```
    public ImpInterfazServidor() throws RemoteException
    {
        super();
        ListaClientes = new Vector();
    }

    public String Hola() throws RemoteException
    {return("Hola capullo...");}

    public synchronized void RegistroCallBack(InterfazCliente
ObjetoCallBack) throws RemoteException
    {

if(!(ListaClientes.contains(ObjetoCallBack))){ListaClientes.addElement
(ObjetoCallBack);
            System.out.println("Nuevo cliente de callback
registrado...");
            EfectuarCallBack();}
    }

    public synchronized void QuitarCallBack(InterfazCliente
ObjetoCallBack) throws RemoteException
    {
        if(ListaClientes.removeElement(ObjetoCallBack))
            System.out.println("Cliente eliminado...");
            else System.out.println("Cliente no registrado...");
    }

    private synchronized void EfectuarCallBack()throws RemoteException
    {
        for(int i=0; i<ListaClientes.size(); i++)
            {
                InterfazCliente Cliente = (InterfazCliente)
ListaClientes.elementAt(i);
                Cliente.Senial ("Eres el cliente nï¿½mero: " + i);
            }//for
    }
}
```

Vemos que un objeto servidor que implementa a `InterfazServidor` comienza por preparar el espacio de almacenamiento donde deberá guardar informaciï¿½n sobre aquellos clientes que deseen registrase en él para *callback*. En este caso, se trata de sencillo objeto de la clase `Vector`. Nota: si lo desea, puede hacer el tipado de la clase vector.

```
                    private Vector ListaClientes;
```

Tras ello, se implementan los métodos que conforman a la interfaz `InterfazServidor`. El primero de ellos lo único que hace es enviar un mensaje a un objeto cliente bajo el modelo rmi estándar, y usted ya conoce cómo funciona todo esto. El segundo de ellos, se encarga de registrar un objeto para *callback* en el espacio de memoria (el contenedor de clase `Vector`) habilitado para ello:

```
  public void synchronized RegistroCallBack(InterfazCliente
ObjetoCallBack) throws RemoteException
    {

if(!(ListaClientes.contains(ObjetoCallBack))){ListaClientes.addElement
(ObjetoCallBack);
```

```
            System.out.println("Nuevo cliente de callback
registrado...");
            EfectuarCallBack();}
    }
```

Vemos que el método es `synchronized`, al igual que `QuitarCallBack` e incluso que `EfectuarCallBack()`. Esto se hace así puesto que varios clientes pueden estar intentando registrarse o darse de baja para *callback* al mismo tiempo.

Este método recibe como parámetro (desde un cliente) una referencia a un objeto que implementa a `InterfazCliente` y lo registra -si no estaba ya registrado- en la memoria habilitada para ello con `ListaClientes.addElement(ObjetoCallBack)`. Se informa al usuario del objeto servidor de que un cliente se registró para *callback*, y se invoca al método `EfectuarCallBack()`.

Es este un método interno de la clase `ImpInterfazServidor`, que describimos a continuación:

```
private synchronized void EfectuarCallBack()throws RemoteException
    {
        for(int i=0; i<ListaClientes.size(); i++)
            {
                InterfazCliente Cliente = (InterfazCliente)
ListaClientes.elementAt(i);
                Cliente.Senial ("Eres el cliente nï¿½mero: " + i);
            }//for
    }
```

Es un méodo simple en sí mismo; lo único que hace es ir recorriendo la lista que guarda las referencias a los objetos cliente que se registraron para *callback*, e invocando **vía *callback*** al mñetodo que forma parte de la interfaz de esos clientes:

```
                InterfazCliente Cliente = (InterfazCliente)
ListaClientes.elementAt(i);
                Cliente.Senial ("Eres el cliente numero: " + i);
```

Finalmente, el método

```
public synchronized void QuitarCallBack(InterfazCliente
ObjetoCallBack) throws RemoteException
```

sirve a los clientes para darse de baja *callbacks* en la lista de del objeto servidor donde se registraron cuando lo consideran oportuno.

Desarrollemos por úlltimo el código del servidor:

```
/*@author Antonio Tomeu
*@version 1.0
*Clase que implementa al servidor
*Nota: Adaptado de Liu, M. Computacion Distribuida
**/

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
```

```
import java.io.*;

public class Servidor
{
    public static void main (String[] args)
    {
        try
            {
                ImpInterfazServidor ObjServidor = new
ImpInterfazServidor();
                Naming.rebind("rmi://localhost:2001/callback",
ObjServidor);
                System.out.println("Servidor Activo...");
            }catch (Exception e) {System.out.println("Problema en
servidor...");}
    }
}
```

En esta ocasión lo hemos simplificado al máximo, y todo lo que hace es ya conocido por usted; se crea un objeto servidor, se registra y el mismo queda a la espera de recibir peticiones de clientes a travïés del puerto indicado.

Ejecución del Ejemplo.

Como siempre, lo haremos todo localmente la primera vez. Descargue desde la carpeta correspondiente los seis códigos que ha sido objeto de análisis en los párrafos anteriores. Guárdelos en una carpeta llamada `rmicallback`, o como usted quiera. Abra ahora una ventana de sistema, navegue a la carpeta indicada y realice las siguientes acciones:

a) Compile todos los códigos: `javac *.java` Nota: obtendra um mensaje Del compilador indicado que "ImpInterfazServidor.java uses unchecked or unsafe operations", debido a que estamos utilizando un contendor polimórfico sin tipos, aunque la compilación se habrá efectuado normalmente. Si lo desea puede efectuar el tipado del contenedor Vector, indicando <tipo> junto a él.

b) Genere los resguardos desde el lado del cliente: `rmic –vcompat ImpInterfazCliente`

c) Genere los resguardos desde el lado del servidor: `rmic –vcompat ImpInterfazServidor`

d) Active el registro de servicios: `start rmiregistry 2001`

f) Active un servidor: `java Servidor`

g) Abra otra venta de sistema y active un cliente: `java Cliente`

h) Observe lo que ocurre y analice la lógica del asunto. Y claro, la verdadera gracia está en activar varios clientes a la vez. Haga los arreglos necesarios para ver qué ocurre en este caso.

2) Comunicación Segura con RMI

En general la transferencia de informacón a través de RMI puede provocar inseguridad en el sistema. Para evitarlo, la arquitectura proporciona la clase

`RMISecurityManager`, cuyos objetos pueden ser instanciados desde cualquier programa, quedando a partir de entonces bajo control del mismo todos los aspectos que puedan suponer violaciones de seguridad en los objetos. Para funcionar en combinación con RMI, el proceso servidor debe instalar un gestor de seguridad antes de hacer cualquier otra cosa. En particular, utilizaremos el gestor por defecto proporcionado por el jdk, si bien otros son posibles. Este es tan restrictivo en sus condiciones iniciales por defecto, que apenas podremos lanzar un servidor sin tener dificultades. Es posible graduar el nivel de seguridad requerido parametrizando al gestor a través de un fichero de políticas de seguridad, generalmente de texto plano. He aquí un ejemplo, llamado `tomeu.policy`.

```
//sencillo fichero de politica de seguridad para pruebas
grant
{
//rango de puertos de escucha permitidos
permission java.net.SocketPermission "*:1099-2002", "connect, accept,
resolve";
};
```

Descargue el fichero anterior, y sitúelo en su carpeta de trabajo. Cuando la aplicación está distribuida, una copia del mismo debe estar junto al servidor y otra junto al cliente. Descargue ahora los ficheros `IEjemploRMI1.java`, `EjemploRMI1.java` y `ClienteEjemploRMI.java`. Recuerde dejar activas en cliente y servidor las líneas que habilitan el gestor de seguridad:

```
System.setSecurityManager(new RMISecurityManager());
```

Proceda a compilarlos y genere mediante `rmic` el *stub* y el *skeleton* de la manera habitual.

Abra ahora un shell de sistema, sitúese en la carpeta de trabajo, active el registro de objetos servidores y lance el servidor, de acuerdo a la siguiente orden:

```
java -Djava.security.policy=tomeu.policy EjemploRMI1
```

que lanza el servidor activando el gestor de seguridad de acuerdo a la política de seguridad que hemos fijado en el fichero `tomeu.policy`, y que restringe las comunicaciones entre servidores y clientes a los puertos indicados. Naturalmente, hay mucho más sobre la gestión de la seguridad en java, y remitimos al lector al documento disponible en **http://docs.oracle.com/javase/1.4.2/docs/tooldocs/windows/policytool.html** para más detalles. Es necesario aclarar también que esto no provee comunicaciones seguras cifradas, ya que de hecho, la arquitectura RMI no se diseño teniendo en cuenta este detalle. No obstante, es todavía posible lograr comunicaciones cifradas con RMI tunelizando las invocaciones remotas sobre sockets seguro. Puede ver este documento si http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/socketfactory/SSLInfo.html desea obtener detalles adicionales sobre cómo hacerlo.

De la misma forma ejecutamos un cliente:

```
java -Djava.security.policy=tomeu.policy ClienteEjemploRMI1
```

Es también posible ajustar la política de seguridad desde el propio código utilizando instrucciones de la forma `System.setSecurityManager(new RMISecurityManager());` que permiten activar un gestor de seguridad en base a una política de seguridad dada, pero nosotros no lo haremos en esta ocasión.

Para escribir una política de seguridad, puede editar un fichero de texto .policy, o utilizar el asistente policytool, tal y como se la habrá explicado en clase de teoría, generar las políticas de seguridad mediante un asistente que forma parte de los binarios del jdk, y que puede activarse mediante el comando de sistema `policytool`.