Departamento de Ingeniería Telemática
Escuela Superior de Ingenieros
Universidad de Sevilla

PRÁCTICA 3.
INVOCACIÓN DE MÉTODOS REMOTOS (RMI, REMOTE METHOD INVOCATION)

1. OBJETIVO

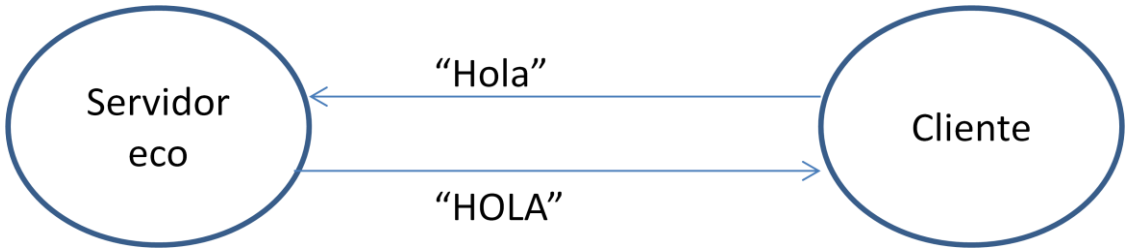
El objetivo es mostrar el funcionamiento de RMI para la invocación remota de métodos en java mediante la prueba de un ejemplo simple.

2. DESCRIPCIÓN

RMI permite programar aplicaciones distribuidas en java de una forma rápida. Para el desarrollo de un servicio remoto se van a seguir los siguientes pasos:

1. Definición del servicio
2. Implementación del servicio
3. Desarrollo del servidor
4. Desarrollo del cliente
5. Compilación
6. Ejecución

Estos pasos se explican en los siguientes apartados para un servicio de eco. El servidor de eco devuelve al cliente la cadena en mayúsculas que el cliente le ha enviado.



2.1 DEFINICIÓN DEL SERVICIO

Para crear un servicio remoto en RMI, hay que definir una interfaz que derive de la interfaz `Remote` y que contenga los métodos requeridos por ese servicio, especificando en cada uno de ellos que pueden activar la excepción `RemoteException`, usada por RMI para notificar errores relacionados con la comunicación.

Este servicio ofrece únicamente un método remoto que retorna la cadena de caracteres recibida como argumento pero pasándola a mayúsculas.

A continuación, se muestra el código de esta definición de servicio:

```
import java.rmi.*;

interface ServicioEco extends Remote {
        String eco (String s) throws RemoteException;
}
```

ServicioEco.java

2.2 IMPLEMENTACIÓN DEL SERVICIO

A continuación es necesario desarrollar el código que implementa cada uno de los servicios remotos. Ese código debe estar incluido en una clase que implemente la interfaz de servicio definida en la etapa previa. Para permitir que los métodos remotos de esta clase puedan ser invocados externamente, la opción más sencilla es definir esta clase como derivada de la clase `UnicastRemoteObject`. Una limitación de esta alternativa es que, debido al modelo de herencia simple de Java, nos impide que esta clase pueda derivar de otra relacionada con la propia esencia de la aplicación (otra alternativa sería usar el método estático `exportObject` de `UnicastRemoteObject`).

Por lo tanto, para la implementación del servicio, desarrollaremos una clase derivada de `UnicastRemoteObject` y que implemente la interfaz remota ServicioEco:

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

class ServicioEcoImpl extends UnicastRemoteObject implements
ServicioEco {
    ServicioEcoImpl() throws RemoteException {
    }
    public String eco(String s) throws RemoteException {
        return s.toUpperCase();
    }
}
```

ServicioEcoImpl.java

Observe la necesidad de hacer explícito el constructor para poder declarar que éste puede generar la excepción `RemoteException`.

Es importante entender que todos los objetos especificados como parámetros de un método remoto, así como el retornado por el mismo, se pasan por valor, y no por referencia como ocurre cuando se realiza una invocación a un método local. Esta característica tiene como consecuencia que cualquier cambio que se haga en el servidor sobre un objeto recibido como parámetro no afecta al objeto original en el cliente. Por ejemplo, este método remoto no llevará a cabo la labor que se le supone, aunque sí lo haría en caso de haber usado ese mismo código (sin la excepción `RemoteException`) para definir un método local.

```
public void vuelta(StringBuffer s) throws RemoteException {
    s.reverse();
}
```

Un último aspecto que conviene resaltar es que la clase que implementa la interfaz remota es a todos los efectos una clase convencional y, por tanto, puede incluir otros métodos, además de los especificados en la interfaz. Sin embargo, esos métodos no podrán ser invocados directamente por los clientes del servicio.

2.2 DESARROLLO DEL SERVIDOR

El programa que actúe como servidor debe iniciar el servicio remoto y hacerlo públicamente accesible usando el registro de RMI (rmiregistry) que es el proceso que hace de enlazador.

A continuación, se muestra el código del servidor:

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

class ServidorEco  {
    static public void main (String args[]) {
       if (args.length!=1) {
            System.err.println("Uso:                ServidorEco
numPuertoRegistro");
            return;
        }
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
RMISecurityManager());
        }
        try {
            ServicioEcoImpl srv = new ServicioEcoImpl();
            Naming.rebind("rmi://localhost:"   +   args[0]   +
"/Eco", srv);
        }
        catch (RemoteException e) {
            System.err.println("Error  de  comunicacion:  "  +
e.toString());
            System.exit(1);
        }
```

```
            catch (Exception e) {
                System.err.println("Excepcion en ServidorEco:");
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
```

ServidorEco.java

A la hora de ejecutar el programa el registro de RMI debe estar arrancado previamente. Se explica cómo arrancarlo en la sección correspondiente a la ejecución.

El programa espera recibir como único argumento el número de puerto por que el que está escuchando el proceso `rmiregistry`.
En el código del servidor se puede apreciar cómo éste instancia un gestor de seguridad, que da soporte a la seguridad pero que no vamos a entrar en detalle. La parte principal de este programa está incluida en la sentencia `try` y consiste en crear un objeto de la clase que implementa el servicio remoto y darle de alta en el `rmiregistry` usando el método estático `rebind` que permite especificar la operación usando un formato de tipo URL. El `rmiregistry` sólo permite que se den de alta servicios que ejecutan en su misma máquina.

2.3 DESARROLLO DEL CLIENTE

El cliente debe obtener una referencia remota (es decir, una referencia que corresponda a un objeto remoto) asociada al servicio para luego invocar de forma convencional sus métodos, aunque teniendo en cuenta que pueden generar la excepción `RemoteException`. En este ejemplo, la referencia la obtiene a través del `rmiregistry`.

A continuación, se muestra el código del cliente:

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

class ClienteEco {
    static public void main (String args[]) {
        if (args.length<2) {
            System.err.println("Uso:  ClienteEco  hostregistro
numPuertoRegistro ...");
            return;
        }

        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try {
```

```
            ServicioEco srv = (ServicioEco) Naming.lookup("//"
+ args[0] + ":" + args[1] + "/Eco");

            for (int i=2; i<args.length; i++)
                System.out.println(srv.eco(args[i]));
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " +
e.toString());
        }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteEco:");
            e.printStackTrace();
        }
    }
}
```

ClienteEco.java

El programa espera recibir como argumentos la máquina donde ejecuta rmiregistry, así como el puerto por el que escucha. El resto de los argumentos recibidos por el programa son las cadenas de caracteres que se quieren pasar a mayúsculas.
Como ocurre con el servidor, se realiza la activación de un gestor de seguridad.
El programa usa el método estático lookup para obtener del rmiregistry una referencia remota del servicio. Observe el uso de la operación de cast para adaptar la referencia devuelta por lookup, que corresponde a la interfaz Remote, al tipo de interfaz concreto, derivado de Remote, requerido por el programa (ServicioEco).
Una vez obtenida la referencia remota, la invocación del método es convencional, requiriendo el tratamiento de las excepciones que puede generar.

2.4 COMPILACIÓN

El proceso de compilación tanto del cliente como del servidor es el habitual en Java. El único punto que conviene resaltar es que para generar el programa cliente, además de la(s) clase(s) requerida(s) por la funcionalidad del mismo, se debe disponer del fichero class que define la interfaz (en este caso, ServicioEco.class), tanto para la compilación como para la ejecución del cliente. Para ello deberá copiar el fichero class del ServicioEco al directorio donde está el cliente. Obsérvese que no es necesario, ni incluso conveniente, disponer en el cliente de las clases que implementan el servicio.
Hay que resaltar que en la versión actual de Java (realmente, desde la versión 1.5) no es necesario usar ninguna herramienta para generar soportes ni para el cliente (proxy) ni para el servidor (skeleton). En versiones anteriores, había que utilizar la herramienta rmic para generar las clases que realizan esta labor, pero gracias a la capacidad de reflexión de Java, este proceso ya no es necesario.

En el ejemplo que nos ocupa, dado que, por simplicidad, no se han definido paquetes ni se usan ficheros JAR, para generar el programa cliente y el servidor, basta con entrar en los directorios respectivos y ejecutar directamente:

```
javac *.java
```

2.5 EJECUCIÓN

Antes de ejecutar el programa, hay que arrancar el registro de Java RMI (rmiregistry). Este proceso ejecuta por defecto usando el puerto 1099, pero puede especificarse como argumento al arrancarlo otro número de puerto, lo que puede ser lo más conveniente para evitar colisiones en un entorno donde puede haber varias personas probando aplicaciones Java RMI en la misma máquina.
Hay que tener en cuenta que el rmiregistry tiene que conocer la ubicación de las clases de servicio. Para ello, puede necesitarse definir la variable de entorno CLASSPATH para el rmiregistry de manera que haga referencia a la localización de dichas clases. Esto se hace usando export CLASSPATH=directorio_del_servidor en el terminal donde se ejecuta el registro. En cualquier caso, si el rmiregistry se arranca en el mismo directorio donde ejecutará posteriormente el servidor y en la programación del mismo no se han definido nuevos paquetes (todas las clases involucradas se han definido en el paquete por defecto), no es necesario definir esa variable de entorno. Así ocurre en este ejemplo:

```
cd servidor
rmiregistry 54321 &
```

Queda un último aspecto vinculado con la seguridad. Dado que tanto en el cliente como en el servidor se ha activado un gestor de seguridad, si queremos poder definir nuestra propia política de seguridad, con independencia de la que haya definida por defecto en el sistema, debemos crear nuestros propios ficheros de políticas de seguridad.
Dado que estamos trabajando en un entorno de pruebas, lo más razonable es crear ficheros de políticas de seguridad que otorguen todos los permisos posibles tanto para el cliente (fichero que hemos llamado cliente.permisos) como para el servidor (fichero servidor.permisos):

```
grant  {
   permission java.security.AllPermission;
};
```

NOTA: Como no vamos a tratar este tema y dada su importancia, si está interesado, se recomienda revisar las numerosas fuentes disponibles en Internet que tratan acerca de ello.

Procedemos finalmente a la ejecución del servidor:

```
cd servidor
java -Djava.security.policy=servidor.permisos  ServidorEco 54321
```

Y la del cliente:

```
cd cliente
java -Djava.security.policy=cliente.permisos  ClienteEco localhost 54321 hola adios
HOLA
ADIOS
```

⇒ 1. Descargue el fichero p3RMIcodigo.zip de la plataforma de enseñanza virtual y descomprímalo.
⇒ 2. Siga los pasos que se han explicado para la ejecución del servidor y del cliente.
NOTA: Por la configuración propia del centro de cálculo, la ejecución debe ser la siguiente:

Para la ejecución del registro:
rmiregistry –J-Djava.net.preferIPv4Stack=true 54321 &

Para la ejecución del servidor:
java –Djava.security.policy=servidor.permisos –Djava.net.preferIPv4Stack=true
–Djava.rmi.server.hostname=dirección_IP_de_la_máquina ServidorEco 54321

Obteniendo la dirección IP con ifconfig.

Para la ejecución del cliente:
java –Djava.security.policy=cliente.permisos –Djava.net.preferIPv4Stack=true ClienteEco localhost 54321 hola adios

⇒ 3. Cree un servicio de eco que además al hacer eco, invierta la cadena de caracteres que se le da, partiendo del ejemplo. Póngalo en el directorio ecoInv dentro de p3RMIcodigo. Llame a la interfaz, al servidor y al cliente, igual que en el caso anterior pero poniendo siempre EcoInv en vez de Eco. Llámele al método ecoInv. Para ello puede utilizar el método reverse de la clase StringBuffer (cree un objeto del tipo StringBuffer, dando en el constructor la cadena que se quiere invertir, una vez invertida, se puede obtener del StringBuffer la cadena mediante el método toString).

2.6 CONTROL DE LA CONCURRENCIA: SERVICIO DE LOG

El mecanismo de invocación remota no sólo libera al programador de todos los aspectos relacionados con el intercambio de mensajes, sino también de todas las cuestiones vinculadas con el diseño de un servicio concurrente.

Java RMI se encarga automáticamente de desplegar los threads requeridos para dotar de concurrencia a un servicio implementado usando esta tecnología. Aunque esta característica es beneficiosa, el programador debe ser consciente de que los métodos remotos en el servidor se ejecutan de manera concurrente, debiendo establecer mecanismos de sincronización en caso de que sea necesario.

Esta concurrencia automática hace que, como puede observarse en el ejemplo previo, el programa servidor no termine cuando completa su código, sino que se quede esperando indefinidamente la llegada de peticiones. Nótese cómo en el tratamiento de las excepciones se usa una llamada a System.exit para completar explícitamente su ejecución.

Este ejemplo intenta ilustrar esta cuestión creando un hipotético servicio de log que ofrece un método que permite al cliente enviar un mensaje al servidor para que lo almacene de alguna forma. La interfaz del ServicioLog es el siguiente:

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

```
interface ServicioLog extends Remote {
        void log (String m) throws RemoteException;
}
```

ServicioLog.java

Para ilustrar la cuestión que nos ocupa, este método va a enviar el mensaje a dos destinos: a la salida estándar del programa servidor y a un fichero especificado como argumento del programa servidor. Esta duplicidad un poco artificial pretende precisamente mostrar la no atomicidad en la ejecución de los servicios remotos.

A continuación, se muestra la clase que implementa esta interfaz remota:

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

class ServicioLogImpl extends UnicastRemoteObject implements ServicioLog {
    PrintWriter fd;
    ServicioLogImpl(String f) throws RemoteException {
        try {
             fd = new PrintWriter(f);
        }
        catch (FileNotFoundException e) {
            System.err.println(e);
            System.exit(1);
        }
    }
    public void log(String m) throws RemoteException {
        System.out.println(m);
        fd.println(m);
        fd.flush();  // para asegurarnos de que no hay buffering
    }
}
```

ServicioLogImpl.java

Se utiliza la clase PrintWriter para escribir en un fichero mediante la función println. El método flush se encarga de enviar directamente la información, que se ha escrito, al fichero en el disco y así nos aseguramos de que no se ha quedado en el buffer de memoria.

No se incluye el código del servidor ni del cliente puesto que no aportan información adicional relevante pero se pueden ver en los ficheros descargados de la plataforma de enseñanza virtual.

Para ilustrar el carácter concurrente del servicio de Java RMI, se propone arrancar simultáneamente dos clientes que envíen un número elevado de mensajes (se muestra un extracto del fichero ClienteLog.java):

```
            for (int i=0; i<10000; i++)
                srv.log(args[2] + " " + i);
```

Se pretende comprobar que los mensajes pueden quedar en orden diferente en la salida estándar y en el fichero precisamente por la ejecución concurrente del método log.

Para la ejecución lo que hay que hacer es lo siguiente:

```
cd servidor
rmiregistry 54321 &
java -Djava.security.policy=servidor.permisos  ServidorLog 54321 fichero >
salida &
cd ../cliente
java -Djava.security.policy=cliente.permisos  ClienteLog localhost 54321 yo &
java -Djava.security.policy=cliente.permisos  ClienteLog localhost 54321 tu
```

NOTA: Debe incluir también las opciones que ya se han indicado en el ejemplo de eco para el funcionamiento en el centro de cálculo.

Se puede apreciar la diferencia entre el fichero y la salida haciendo:

```
diff fichero salida
2544d2543
< tu 714
2545a2545
> tu 714
9985d9984
< yo 5997
9986a9986
> yo 5997
15325a15326
> yo 8469
15444d15444
< yo 8469
17708a17709
> tu 8229
17985d17985
< tu 8229
```

La solución en este caso es la habitual en Java: marcar el método como sincronizado:

```
    public synchronized void log(String m) throws RemoteException {
```

Al marcar el método como sincronizado, se asegura que si un hilo lo está ejecutando, otro hilo que intente ejecutarlo se esperará a que el primero termine. Con esto se asegura exclusión mutua en el acceso al método.

$\Rightarrow$ 4. Pruebe a ejecutar el cliente y servidor de log para ver el resultado indicado. En el caso de que no se produzca la diferencia entre la salida en la salida estándar y el fichero, se puede probar a incluir en la implementación un sleep entre la impresión en la salida estándar y la impresión en el fichero. Para ello se puede poner en el método log lo siguiente:

```
    public void log(String m) throws RemoteException {
        System.out.println(m);
```

```
        try {
           Thread.sleep(1);
        }
        catch( InterruptedException e)
        {
        }

        fd.println(m);
        fd.flush();  // para asegurarnos de que no hay buffering
    }
```

Método log de ServicioLogImpl.java

⇒ 5. Modifique el método log para que sea sincronizado y pruebe de nuevo el ejemplo.

⇒ 6. Cree un servicio que devuelva la fecha y la hora actual. Para ello utilice el directorio llamado marcaTiempo dentro del directorio p3RMIcodigo e incluya en este directorio el cliente y el servidor. Llame a los ficheros del servidor ServicioMarcaTiempo.java, ServicioMarcaTiempoImpl.java y ServidorMarcaTiempo.java y al fichero del cliente clienteMarcaTiempo.java. El servicio debe tener un único método que será:

Date marcaTiempo() throws RemoteException;

Para utilizar la clase Date hay que importar java.util.Date.

⇒ 7. En la tarea habilitada para la práctica, entregue un fichero comprimido (.zip) con nombre su login que sea el directorio de nombre su login comprimido (que contiene el directorio p3RMIcodigo con todo su trabajo).