Arquitecturas Distribuidas
Práctica 6. Introducción a CORBA

1. Objetivos

- Entender el funcionamiento básico de la Llamada a Procedimiento Remoto (LPR).
- Desarrollar aplicaciones distribuidas sencillas basadas en CORBA.

2. Introducción

El *middleware* debe proporcionar a los programadores métodos eficaces para la construcción de aplicaciones distribuidas. La base del *middleware* actual es un sistema de LPR, que permite con una sintaxis similar a la de la función local, invocar funciones que se ejecutan realmente en una máquina servidora.

Para lograr esta transparencia, cada vez que se llama a un PR, el ordenador cliente ejecuta en realidad un "delegado del cliente" o *stub*, cuya misión es contactar con el servidor, y solicitar a un "delegado en el servidor" o *skeleton* la ejecución del procedimiento. El "delegado en el servidor" llama a la implementación del procedimiento en el servidor, recoge los resultados, y los transfiere de regreso al "delegado del cliente". Finalmente, el "delegado del cliente" devuelve el resultado al programa en ejecución. Un sistema de LPR es tanto mejor, cuanto más se parezca la sintaxis de la llamada local a la de una llamada remota equivalente.

En esta práctica se ilustra como trabajar usando CORBA y Java como lenguaje soporte. Para ello, se realizará una presentación de las características del Lenguaje de Definición de Interfaces (IDL, *Interface Description Language*) de CORBA, y se verán dos ejemplos de aplicaciones desarrolladas con CORBA/Java. Finalmente, se proponen tres ejercicios.

3. "Hola Mundo" en CORBA

El primer ejemplo muestra el clásico "Hola Mundo" realizado con CORBA, donde se declara e implementa una interfaz de CORBA con un sólo método: `saludo`. Cada vez que sea invocado desde un cliente, `saludo` imprime un mensaje `"Hola Mundo!"` en la pantalla del servidor. El apéndice A muestra la estructura del IDL para esta aplicación, junto con el código del cliente y servidor. Observe que el fichero que contiene la clase

`Servidor` incluye también la clase `holamundoImplementacion`, que es el código que se ejecuta en la llamada al LPR.

Para probar el ejemplo debe seguir estos pasos:

1. Crear los delegados Java, mediante el compilador `idlj`:

   ```
   idlj -fall holamundo.idl
   ```

   Esta llamada crea un directorio `hola` (mismo nombre que el módulo declarado en el IDL), y dentro de este **para cada interfaz declarada** crea diversas clases de Java, todas ellas asignadas al *package hola* (mismo nombre que el módulo declarado en el IDL).

2. Compilar el código:

   ```
   javac -d hola *.java hola/*.java
   ```

   El parámetro `-d` indica al compilador que todos los `.class` generados deben crearse dentro del directorio `hola`. En el resto de la llamada, se dice a `javac` que debe usar en la compilación todos los `.java` del directorio actual, y todos los del directorio `hola`.

3. Probar el ejemplo. Para ello, desde un *shell*, debe ejecutar el servidor.

   ```
   java -cp .:hola hola.Servidor
   ```

   Esta llamada lanza el servidor. Observe que se generá el archivo `hola.ref` que contiene el IOR del objeto, usando la llamada al método `object_to_string`. Este archivo sirve posteriormente al cliente para contactar con el servidor.

   A continuación, desde otro *shell* `en otro ordenador`, debe lanzar el cliente:

   ```
   java -cp .:hola hola.Cliente
   ```

   El parámetro `-cp .:hola` redefine el `PATH` para poder ejecutar correctamente el código Java. Tras iniciarse, el cliente lee `hola.ref`[*], y crea a través del método `string_to_object` una referencia a un objeto remoto generalizado, que se convierte en un objeto del tipo `holamundo` con el método a través `narrow`. Una vez obtenido un objeto `holamundo` específico, se puede llamar a su método remoto `saludo`. Una vez ejecutado el cliente, compruebe que el mensaje `"Hola Mundo!"` aparece en la ventana del servidor.

[*] Notese que cada usuario observa los mismos archivos desde cualquier ordenador, puesto que se está utilizando el sistema de ficheros distribuido NFS (que curiosamente, también se implementa con un mecanismo de LPR).

3.1. Cuestiones

- ¿Qué sucedería si se lanzase el cliente antes que el servidor?
- ¿Qué sucedería si se lanzase el cliente antes del servidor y existiese un `hola.ref` de un servidor previo?
- ¿Qué sucedería si se lanzasen varios servidores y un solo cliente?
- Supongamos que se desea realizar una modificación de este ejemplo en la cual no sea el servidor el que imprime un mensaje de bienvenida, sino el cliente el que muestra en pantalla un mensaje de saludo, que le es transferido desde el servidor. Realice las modificaciones adecuadas al IDL y a las clases `Cliente` y `Servidor` para ello.

4. Calculadora remota con CORBA

En este ejercicio se muestra como crear una calculadora remota usando CORBA. Se pretende crear dos modelos diferentes, uno para cálculo entero, y otro para cálculo real. En CORBA, pueden usarse para este tipo de definición dos interfaces pertenienciente al mismo módulo. En el apéndice B se muestra el IDL, y el cliente y servidor correspondientes a este ejemplo. Como novedades, puede observar:

- Para tratar el caso de la división por 0, la interfaz declara que los métodos de división pueden lanzar una excepción CORBA, declarada como `NumeroIncorrecto`. Como puede observar en la implementación, una excepción CORBA se traduce directamente a una excepción Java*, y se lanza del mismo modo (cláusula `throw`). La generación de la excepción provoca la finalización inmediata del PR, y la recepción de la misma en el cliente.
- En la calculadora entera, la llamada al método de división, debe devolver junto con el cociente el resto de la división. Para ello, se declara un parámetro adicional de tipo `out`. Puesto que en Java no existe de modo nativo el concepto de copia/restauración, es preciso manejar los parámetros de tipo `out` e `inout` a través de clases especiales *Holder*. El ejemplo le muestra como usarlas.

4.1. Cuestiones

- Pruebe el ejemplo anterior siguiendo los pasos del ejemplo del “Hola Mundo”.
- Existen, además del presentado, otros métodos para devolver el resto en el ejemplo de la calculadora. En concreto, debe cambiar el ejemplo para hacerlo de los dos modos siguientes:
    - Devolviendo el resto en un parámetro de entrada, de tipo `inout`.

* En otros lenguajes sin concepto de excepción, como C, la generación y recepción de excepciones se complica.

    - Devolviendo el resto y el cociente en una estructura, en el resultado.

- Por último, amplíe la calculadora con otra interfaz para cálculo vectorial, con operaciones de suma y resta de vectores y multiplicación escalar. En esta cuestión, debe usar parámetros de tipo `sequence` (arrays en IDL).

5. Ejercicio: Acceso a archivos remotos con CORBA

En este ejercicio debe implementar cuatro PR para manejar archivos remotos, análogos a sus equivalentes locales. Estas cuatro funciones deben permitir `abrir`, `cerrar`, `leer` y `escribir` archivos que se encuentran físicamente en el ordenador remoto, desde el equipo local.

Una vez implementados estos PR, cree con ellos los siguientes programas:

1. **rcat**: Cat remoto, análogo en funcionamiento al cat en `UNIX`.
2. **rpipe**: Redirección de contenido a un archivo remoto. En `UNIX` la llamada:

   ```
   programa > fichero
   ```

   vuelca al archivo local `fichero` el contenido generado en *stdout* por `programa`.

   Entonces, se pretende que:

   ```
   programa | rpipe fichero_remoto
   ```

   vuelque la salida del programa al `fichero_remoto`.

6. Ejercicio: Computación en paralelo con CORBA

Uno de los usos comunes de las AD es realizar cómputos en paralelo en M ordenadores. De este modo, se logra reducir el tiempo de ejecución a $\frac{T_{local}}{M}$.

En este ejercicio realizará una implementación en paralelo del *problema del viajante*: Considere una lista de ciudades, que un viajante debe recorrer, partiendo de una ciudad origen a la que debe regresar. La distancia entre cada dos ciudades también se supone conocida. Se desea determinar cual es la ruta óptima (aquella en la que la distancia recorrida es menor). Puesto que el número de posibles rutas es $(N-1)!$ [*], es preciso entonces limitar a un número K de iteraciones al algoritmo. Así, cada uno de los M computadores realiza $\frac{K}{M}$ iteraciones.

Aunque existen técnicas muy elaboradas que intentan resolver este problema, es posible intentar una busqueda "al azar" entre todas las soluciones posibles. Es decir, partiendo del conjunto de ciudades $\{C_1, C_2, ..., C_N\}$, se eligen rutas aleatorias:

$$C_1 \rightarrow C_i \rightarrow \ldots \rightarrow C_j \rightarrow C_1 \tag{1}$$

[*] Ejemplo) $N = 32$, $Rutas = 8{,}2228e+33$

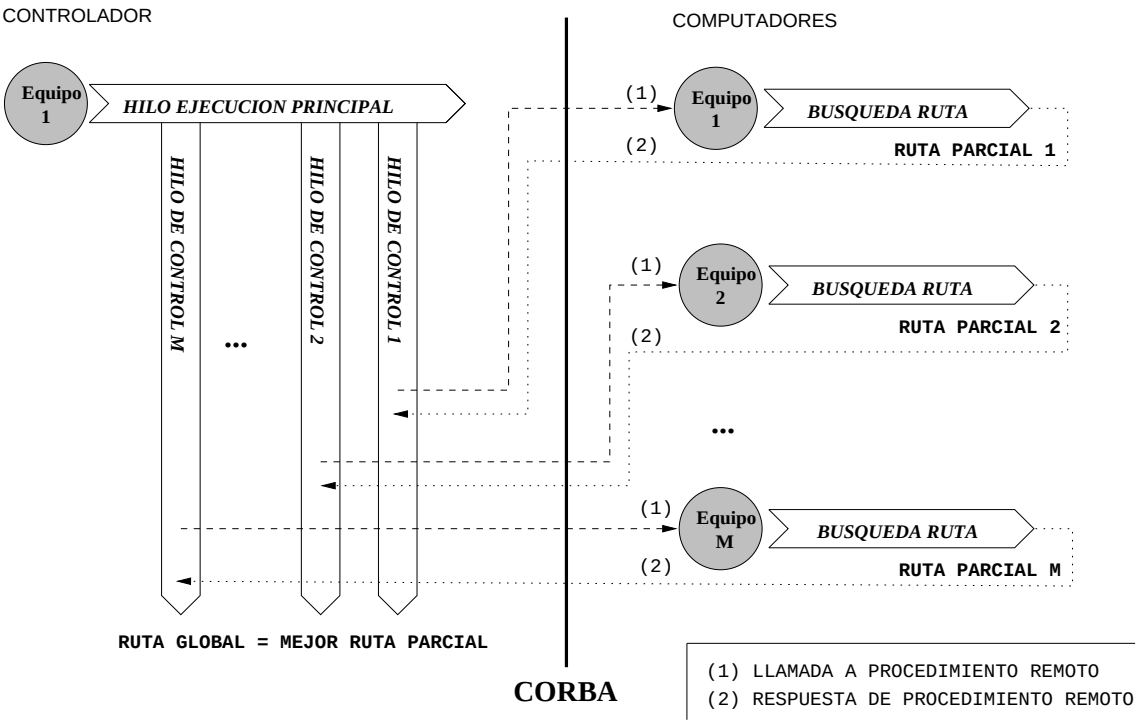

Figura 1: Computo en paralelo del problema del viajante

y se calcula su coste asociado:

$$coste = dist(C_1, C_i) + ... + dist(C_j, C_1) \quad (2)$$

Si el coste es menor que el mínimo hallado hasta el momento, se actualiza el valor del mínimo, y se prosigue. Obviamente, este tipo de algoritmos da una solución subóptima. La ventaja es que este método es fácilmente paralelizable, basta con que cada ordenador ejecute exactamente el mismo algoritmo y obtenga mínimos parciales. Al final de la ejecución, se elige la mejor solución entre todas las soluciones parciales.

Para desarrollar este ejercicio en Java puede seguir un esquema similar al mostrado en la figura 1. En ella, puede observar que el algoritmo está gobernado por un programa "controlador", que se divide en M hilos de ejecución, y desde cada uno de esos hilos realiza invocaciones a otros procesos "computadores", uno en cada máquina que colabora en la resolución del algoritmo. En cada LPR, una máquina "computadora" recibe como parámetros la lista de ciudades y el número de rutas que debe intentar, y como respuesta debe devolver la mejor ruta encontrada y el coste de ese viaje. Finalmente, observe en la figura 1 que el ordenador "controlador" también ejecuta un proceso de busqueda.

7. Ejercicio: Servicio de mensajería instantanea sobre WWW

Este ejercicio consiste en desarrollar un servicio de mensajería instantánea sobre WWW, cuyo funcionamiento sea similar al *messenguer* o al *ICQ*. Para desarrollarlo debe utilizar CORBA sobre *applets* Java, usando como guía una arquitectura como la mostrada en la

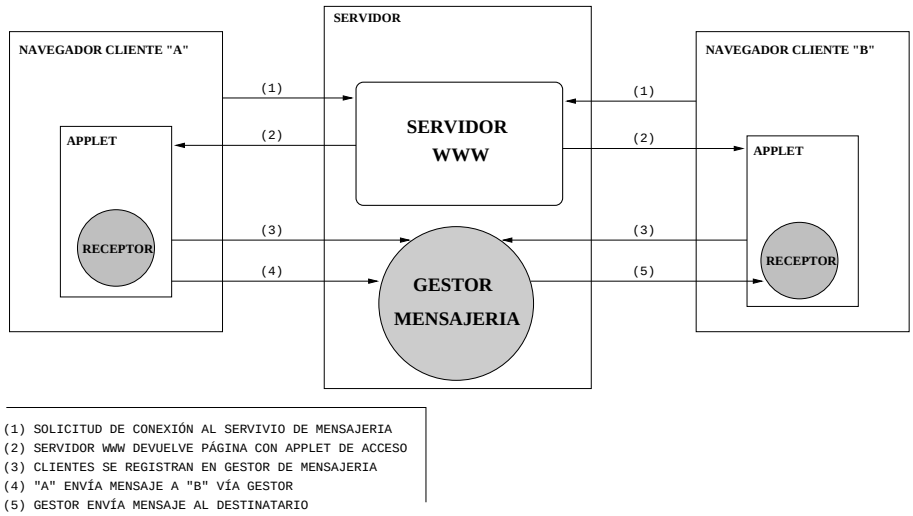

Figura 2: Servicio de mensajeria con CORBA

figura 2. Observe que la novedad en esta arquitectura, respecto a los ejercicios anteriores, es que un objeto de CORBA debe funcionar como cliente de otro objeto. Además, todas las comunicaciones deben pasar por el gestor, ya que un *applet* Java no puede abrir conexiones con una IP diferente de la que se descargó [*].

[*] A no ser que se disponga de un certificado digital para el *applet*.

8. Ejercicio: Distribución de noticias simple.

Se desea crear un servicio de distribución de noticias empleando CORBA. Para ello, los clientes pueden acceder a un servidor centralizado para ESCRIBIR (una noticia) o LEER (el bloque de todas las noticias enviadas). Asimismo, tras haberlas leído, los clientes pueden PUNTUAR individualmente las noticias entre 0 y 10 puntos. Cuando un cliente lee el bloque de noticias se le indican, para cada una, los votos de la misma y su puntuación máxima (si los hay). Nota: Considere una noticia como una cadena de texto.

1. Escriba un IDL para el objeto servidor de tal aplicación.
2. Codifique la implementación de los métodos remotos correspondientes al IDL anterior.

A. Ejemplo “Hola Mundo”

A.1. IDL

```
//
// holamundo.idl
//

module hola {
        interface holamundo {
                void saludo();
        };
};
```

A.2. Servidor

```
//
// Servidor.java
//
package hola;

// IMPORTAMOS LAS CLASES DE CORBA NECESARIAS
import org.omg.CORBA.ORB;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
                                                                    10



// ****************************************************
// IMPLEMENTACION DE LOS METODOS REMOTOS
// SE REALIZA A TRAVES DE HERENCIA
// ****************************************************
class holamundoImplementacion extends holamundoPOA
{
    public void saludo()                                            20
    {
        System.out.println("Hola Mundo!!\n");
    }
}



// ********************************************************************************
// PROGRAMA SERVIDOR: RECIBE PETICIONES Y LAS ENVÍA A LOS DELEGADOS    30
// SIMILAR PARA CUALQUIER PROGRAMA SERVIDOR QUE USE CORBA
// ********************************************************************************
public class Servidor {

public static void main (String args[]) {
```

```
        try {
                // INICIAMOS ORB Y POA
                ORB orb = ORB.init(args, null);
                POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
                rootPOA.the_POAManager().activate();

                // CREAMOS LOS OBJETOS QUE DARAN SERVICIO
                // Y LOS CONECTAMOS AL ORB
                holamundoImplementacion holaIMPL = new holamundoImplementacion();
                holamundo hh = holaIMPL._this(orb);

                // CREAMOS UNA CADENA CON LA IDENTIFICACION DEL OBJETO
                String ref = orb.object_to_string(hh);

                // VOLCAMOS LA IDENTIFICACION A UN ARCHIVO PARA
                // PERMITIR AL CLIENTE HALLAR EL OBJETO REMOTO
                java.io.FileOutputStream file = new java.io.FileOutputStream("hola.ref");
                java.io.PrintWriter out = new java.io.PrintWriter(file);
                out.println(ref);
                out.flush();
                file.close();

                // ESPERAMOS INVOCACIONES DE LOS CLIENTES
                orb.run();

        } catch(Exception e) {
                System.out.println("ERROR : " + e);
                e.printStackTrace(System.out);
        }
}

}
```

A.3. Cliente

```
//
// Cliente.java
//

package hola;

// Importamos las clases de CORBA necesarias
import org.omg.CORBA.ORB;

// Importamos otras clases necesarias de Java
import java.io.*;

public class Cliente {

public static void main (String args[]) {
```

```
        try {
                // INICIAMOS ORB
                ORB orb = ORB.init(args, null);

                // LEEMOS EN ARCHIVO UBICACION DEL OBJETO REMOTO
                FileInputStream file = new FileInputStream("hola.ref");
                BufferedReader in = new BufferedReader(new InputStreamReader(file));
                String ref = in.readLine();

                // CREAMOS REFERENCIA AL OBJETO REMOTO
                holamundo h = holamundoHelper.narrow(orb.string_to_object(ref));

                // UNA VEZ OBTENIDA LA REFERENCIA PODEMOS LLAMAR
                // A LOS METODOS DEL OBJETO REMOTO
                h.saludo();

        } catch(Exception e) {
                System.out.println("ERROR : " + e);
                e.printStackTrace(System.out);
        }
}

}
```

B. Ejemplo “Calculadora”

B.1. IDL

```
//
// calculadora.idl
//

module calc {
        exception NumeroIncorrecto {};


        interface entero {
                long suma(in long a, in long b);
                long resta(in long a, in long b);
                long multiplica(in long a, in long b);
                long divide(in long a, in long b, out long resto) raises(NumeroIncorrecto);
        };

        interface real {
                double suma(in double a, in double b);
                double resta(in double a, in double b);
                double multiplica(in double a, in double b);
                double divide(in double a, in double b) raises(NumeroIncorrecto);
        };
};
```

B.2. Servidor

```
//
// Servidor.java
//

package calc;

// Importamos las clases de CORBA necesarias
import org.omg.CORBA.*;
import org.omg.CORBA.ORB;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;




// ******************************************************
// IMPLEMENTACION DE LOS METODOS REMOTOS
// SE REALIZA A TRAVES DE HERENCIA
// ******************************************************
class enteroImplementacion extends enteroPOA
{
    public int suma(int a, int b) {
        return a+b;
```

```
    }

    public int resta(int a, int b) {
        return a-b;
    }

    public int multiplica(int a, int b) {
        return a*b;
    }

    public int divide(int a, int b, IntHolder resto) throws calc.NumeroIncorrecto {
        if (b==0) {
                throw new calc.NumeroIncorrecto();
        }

        resto.value = a % b;
        return a/b;
    }
}

class realImplementacion extends realPOA
{
    public double suma(double a, double b) {
        return a+b;
    }

    public double resta(double a, double b) {
        return a-b;
    }

    public double multiplica(double a, double b) {
        return a*b;
    }

    public double divide(double a, double b) throws calc.NumeroIncorrecto {
        if (b==0) {
                throw new calc.NumeroIncorrecto();
        }
        return a/b;
    }
}




// ***************************************************************
// PROGRAMA SERVIDOR: RECIBE PETICIONES Y LAS ENVÍA A LOS DELEGADOS
// SIMILAR PARA CUALQUIER PROGRAMA SERVIDOR QUE USE CORBA
// ***************************************************************
public class Servidor {

public static void main (String args[]) {

        try {
```

```
                // INICIAMOS ORB Y POA
                ORB orb = ORB.init(args, null);
                POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
                rootPOA.the_POAManager().activate();

                // CREAMOS LOS OBJETOS QUE DARAN SERVICIO Y LOS CONECTAMOS AL ORB
                // UNO POR CADA INTERFAZ
                enteroImplementacion enteroIMPL = new enteroImplementacion();
                entero e = enteroIMPL._this(orb);
                realImplementacion realIMPL = new realImplementacion();
                real r = realIMPL._this(orb);

                // CREAMOS UNA CADENA CON LA IDENTIFICACION DE CADA OBJETO
                String refe = orb.object_to_string(e);
                String refr = orb.object_to_string(r);

                // VOLCAMOS LA IDENTIFICACION A SENDOS ARCHIVOS PARA
                // PERMITIR AL CLIENTE HALLAR LOS OBJETOS REMOTOS
                java.io.FileOutputStream file = new java.io.FileOutputStream("entero.ref");
                java.io.PrintWriter out = new java.io.PrintWriter(file);
                out.println(refe);
                out.flush();
                file.close();

                file = new java.io.FileOutputStream("real.ref");
                out = new java.io.PrintWriter(file);
                out.println(refr);
                out.flush();
                file.close();

                // ESPERAMOS INVOCACIONES DE LOS CLIENTES
                orb.run();

        } catch(Exception e) {
                System.out.println("ERROR : " + e);
                e.printStackTrace(System.out);
        }
}

}
```

B.3. Cliente

```
//
// Cliente.java
//

package calc;

// Importamos clases de CORBA necesarias
import org.omg.CORBA.*;
import org.omg.CORBA.ORB;
```

```
                                                                    10
// Importamos clases Java necesarias
import java.io.*;

public class Cliente {

public static void main (String args[]) {


    try {
        // INICIAMOS ORB                                            20
        ORB orb = ORB.init(args, null);

        // LEEMOS EN ARCHIVO UBICACION DEL OBJETO REMOTO
        // PARA CALCULO ENTERO
        FileInputStream file = new FileInputStream("entero.ref");
        BufferedReader in = new BufferedReader(new InputStreamReader(file));
        String ref = in.readLine();
        file.close();

        // CREAMOS REFERENCIA AL OBJETO REMOTO PARA CALCULO ENTERO   30
        entero e = enteroHelper.narrow(orb.string_to_object(ref));


        // LEEMOS EN ARCHIVO UBICACION DEL OBJETO REMOTO
        // PARA CALCULO REAL
        file = new FileInputStream("real.ref");
        in = new BufferedReader(new InputStreamReader(file));
        ref = in.readLine();

        // CREAMOS REFERENCIA AL OBJETO REMOTO PARA CALCULO REAL     40
        real r = realHelper.narrow(orb.string_to_object(ref));


        // UNA VEZ OBTENIDA LA REFERENCIA PODEMOS LLAMAR
        // A LOS METODOS DEL OBJETO REMOTO
        System.out.println("PRUEBAS CON ENTEROS:");
        System.out.println("--------------------");
        System.out.println();
        System.out.println("Resultado suma remota: " + e.suma(4,7));
        System.out.println("Resultado resta remota: " + e.resta(4,7));   50
        System.out.println("Resultado multiplicación remota: " + e.multiplica(4,7));

        // LA DIVISION REQUIERE USAR UNA CLASE HOLDER
        IntHolder resto = new IntHolder();
        System.out.println("Resultado división remota: " + e.divide(3,7,resto));
        System.out.println("Resto división remota: " + resto.value);
        System.out.println();
        System.out.println();

        System.out.println("PRUEBAS CON REALES:");                   60
        System.out.println("--------------------");
        System.out.println();
        System.out.println("Resultado suma remota: " + r.suma(4,7));
```

```
                System.out.println("Resultado resta remota: " + r.resta(4,7));
                System.out.println("Resultado multiplicación remota: " + r.multiplica(4,7));
                System.out.println("Resultado división remota: " + r.divide(3,7));
                try {
                        System.out.println("Prueba división por 0 remota: ");
                        System.out.println(r.divide(3,0));
                } catch (calc.NumeroIncorrecto ni) {                                    70
                        System.out.println("Excepción de NumeroIncorrecto capturada");
                }
                System.out.println();
                System.out.println();

        } catch(Exception e) {
                System.out.println("ERROR : " + e);
                e.printStackTrace(System.out);
        }
}                                                                                       80

}
```