PRÁCTICA 2

CONTROL PARKING SERVIDOR HTTP, SOCKETS Y RMI.

- INDEX -

INTRODUCCIÓN

Esta práctica tiene como objetivo adquirir conocimientos sobre el uso de herramientas de comunicación básicas, tales como RMI y sockets.

Los **sockets** son un mecanismo de comunicación punto a punto entre dos procesos que utilizan el protocolo TCP/IP. Programar sockets es una manera de establecer un canal de comunicación bidireccional entre dos nodos de una red.

Un socket servidor escucha en un puerto en particular, en una IP concreta, mientras que el otro socket cliente hace una petición al socket cliente para que le de un servicio.

Por otro lado, la tecnología **RMI** es una API del lenguaje Java que nos proporcionan un nivel de abstracción mayor. Su mayor ventaja es que nos permite llamar a métodos de objetos de manera remota, es decir que se pueden encontrar en otra máquina.

Por último para el servidor Web implementaremos el protocolo de comunicación **HTTP**, que esta basado en el modelo cliente-servidor y no tiene estado de respuesta/petición, simplemente funciona intercambiando mensajes mediante una conexión TCP/IP. El cliente en este caso el el navegador, el cual establece una conexión hacia el servidor utilizando el formato de mensaje http. EL HTTPServer es un programa en lenguaje Java, que se encarga de aceptar peticiones de los clientes y crear hilos para que sirvan a los clientes.

El sistema que se va a implementar es un controlador de plazas de un parking. El parking dispone de sensores en cada plaza de aparcamiento, que nos da la siguiente información:

-Volumen: El volumen detectado en la plaza de garaje.

-Fecha: La fecha que se realizó la última modificación.

-Led: Indicará (1) si la plaza está ocupada, o (0) si la plaza está libre.

Estos sensores se registrarán mediante RMI, para que desde un centro de control podamos acceder a su información y si hace falta a modificar su estado.

El controlador o controller se encargará de recibir las peticiones del servidor HTTP, previamente echas por el cliente, procesar dichas peticiones y si son aceptadas, consultarlas con los sensores correspondientes. Una vez recibe el la respuesta de los sensores, procede a comunicarla al servidor.

Por último el servidor HTTP, se encarga de proporcionar una interfaz gráfica, para que al usuario le sea más fácil hacer sus peticiones dependiendo de su necesidad. Una vez capta la petición del usuario final, procesa estas para que se adapten al protocolo HTTP y la comunicación entre navegador y servidor se haga correctamente. Envía las peticiones al controller mediante sockets y cuando recibe la respuesta, la vuelve a enviar al navegador, así el cliente recibe la información de manera entendible y se cierra el ciclo de comunicación de este sistema.

RMI

Registro

Se empieza definiendo la interfaz, la cual va a definir que métodos de nuestro objeto el cliente va a tener acceso. Son los servicios que el 'Servidor' ofrece al 'Cliente'.

Por supuesto extiende la interfaz java.rmi.Remote.

```
1  package Sensor;
2  import java.rmi.Remote;
4
5  public interface RemoteInterface extends Remote{
6
7      String getNombre() throws RemoteException;
8      String getFecha() throws RemoteException;
9      int getVolumen() throws RemoteException;
10     int getLed() throws RemoteException;
11     void setLed(int led) throws RemoteException;
12
13 }
14
```

Para que registrar los sensores creamos una clase a parte llamada Registrador, que por una parte implementará una interfaz propia RegistradorInterfaz que tiene dos métodos: registrarSensor y desregistrarSensor. A su vez va a extender RemoteInterface.

He creado una clase Registrador, siguiendo la guía de Oracle al registrar objetos RMI.

```
1  package Sensor;
2
3  import java.rmi.NotBoundException;
5
6  public interface RegistradorInterface extends RemoteInterface{
7
8      public void registrarSensor(RemoteInterface sensor) throws RemoteException;
9      public void desregistrardoSensor(RemoteInterface sensor) throws RemoteException;
10
11 }
12
```

Esta es el método main de la clase Registrador. Recibe dos parámetros la IP y el puerto donde hayamos ejecutado el rmiregistry. Crea el registro y le asigna un nombre. De está manera el registro es localizable para los demás objetos.

```
47     public static void main(String[] args) throws Exception {
48
49         if(args.length >= 2)
50         {
51             Registry registry = LocateRegistry.getRegistry(args[0], Integer.parseInt(args[1]));
52             Registrador master = new Registrador(registry);
53             registry.rebind(Registrador.NOMBRE, master);
54             System.out.println("REGISTRO NOMBRE: " + Registrador.NOMBRE);
55             System.out.println("REGISTRADOR OK -> " + args[0] + " : " + args[1]);
56
57         }else {
58
59             System.out.println("ERROR ARGUMENTOS : <IP> <PORT>");
60         }
61
62     }
```

Sensor

La clase Sensor implementará la interfaz RemoteInterface, es decir, que a cada método le proporcionará una funcionalidad. Se va a encargar de leer el fichero donde están sus propiedades, instanciar dicha clase, localizar el registro y registrar el objeto.

```
1  package Sensor;
2  import java.rmi.Remote;
4
5  public interface RemoteInterface extends Remote{
6
7      String getNombre() throws RemoteException;
8      String getFecha() throws RemoteException;
9      int getVolumen() throws RemoteException;
10     int getLed() throws RemoteException;
11     void setLed(int led) throws RemoteException;
12
13 }
14
```

Se le pasan 3 argumentos a esta clase. La IP del Registro, el puerto del Registro y el archivo de texto, de donde se inicializarán los valores del Sensor.

Una vez localizado el Registro, se procede a utilizar el método de la clase Registro, registrarSensor, que tiene como parámetro el propio Sensor y se registra. Si todo va bien, deberíamos de ver un mensaje de éxito.

Cuando se desee desregistrar el Sensor del Registro, solo hay que pulsar enter. Se llamará al método del Registro, desregistrarSensor para que se encargue.

```
103
104    public static void main(String[] args) throws Exception{
105
106        if(args.length >= 3)
107        {
108            registry = LocateRegistry.getRegistry(args[0], Integer.parseInt(args[1]));
109            Sensor sensor = new Sensor(args[2]);
110            registroInterface = (RegistradorInterface) registry.lookup(Registrador.NOMBRE);
111            registroInterface.registrarSensor(sensor);
112            System.out.println(">> " + sensor.getNombre() + " registrado.");
113            System.out.print("Pulsa ENTER para desregistrar " + sensor.getNombre());
114            new BufferedReader(new InputStreamReader(System.in)).readLine();
115            System.out.println("...");
116            registroInterface.desregistrardoSensor(sensor);
117            System.exit(0);
118        }
119        else
120        {
121            System.out.println("ERROR ARGUMENTOS: <IP> <PORT> <FILENAME>");
122        }
123
124    }
125
```

Controller

Empezando por el main, el Controller lee 2 parámetros importantes que se encuentran en el archivo 'settings.txt'. La IP y el puerto de la máquina donde se ejecuta el Controller. Crea el socket server con estos parámetros.

A continuación se pone a la escucha de peticiones de clientes. En este caso los clientes vienen del servidor HTTP, mediante sockets. Acepta las peticiones y les proporciona un hilo para que se sirvan.

```
27  public class Controller extends Thread implements Runnable{
28
29      private static Registry registry;
30      private Socket connect;
31
32      Controller(Socket c){
33          this.connect = c;
34      }
35
36      public static void main(String[] args) {
37          try
38          {
39              ServerSocket c_Server = new ServerSocket(HTTPServer.CONTROLLER_PORT);
40              System.out.println("Controller active in: " + HTTPServer.CONTROLLER_IP + " : " + HTTPServer.CONTROLLER_PORT);
41
42              while(true)
43              {
44                  Socket c_Client = c_Server.accept();
45                  Controller controller = new Controller(c_Client);
46                  Thread thread = new Thread(controller);
47                  thread.start();
48              }
49          }
50          catch(Exception e)
51          {
52              System.err.println("ERROR CONTROLLER : " + e.getMessage());
53          }
54      }
55
```

En la siguiente imagen se puede observar como lee el socket, recibe la petición y si en la petición se especifica que se quiere la información de todos los sensores, la procesa, localizando el Registro, y llamando a los métodos correspondientes. Después se moldea la respuesta para que al escribirla de vuelta en el socket sea interpretada por el servidor http y el navegador. Este es solo un ejemplo de como se procesa las peticiones mediante el controller.

```
57      @Override
58      public void run() {
59
60          BufferedReader br = null;
61          String ask = "";
62          String ans = "";
63
64          try
65          {
66              ask = readSocketServer();
67              registry = LocateRegistry.getRegistry(HTTPServer.REGISTRY_IP, HTTPServer.REGISTRY_PORT);
68              String[] s = ask.split("&");
69              String respuesta = "";
70
71              if(s[1].contains("all"))
72              {
73                  int i = 0;
74                  while(i <= 4)
75                  {
76                      i++;
77                      try
78                      {
79                          String sensorNombre = "Sensor/Sensor"+String.valueOf(i);
80                          System.out.println("NOMBRE SENSOR: " + sensorNombre);
81                          Object remoteObject = registry.lookup(sensorNombre);
82                          if(remoteObject instanceof RemoteInterface)
83                          {
84                              RemoteInterface sensor = (RemoteInterface) remoteObject;
85                              respuesta += "Sensor"+i+"<br>";
86                              respuesta += String.valueOf(sensor.getVolumen()) + "<br>" ;
87                              respuesta += sensor.getFecha() + "<br>";
88                              if(sensor.getLed() == 0) respuesta += "LIBRE";
89                              else respuesta += "OCUPADA";
90                              respuesta += "<br><br>";
91
92                              System.out.println(".....................................");
93                              System.out.println("RESPUESTA: " + respuesta);
94                              System.out.println(".....................................");
95
96                          }
97                      }
98                      catch(NotBoundException e)
99                      {
100                         respuesta = "error";
101                     }
102             }
```

HTTPServer

Primero lee las especificaciones desde un fichero de texto. Estas se guardan en variables estáticas, dentro del HTTPServer.

settings.txt

```
MAX_CONNECTIONS=3
HTTP_PORT=1025
CONTROLLER_IP=127.0.0.1
CONTROLLER_PORT=1026
REGISTRY_IP=127.0.0.1
REGISTRY_PORT=1099
```

Este servidor actúa de la misma manera que el controller, crea un socket servidor y se mantiene a la espera de sockets clientes que quieran conectarse y les proporciona un hilo para su servicio.

```
67    public static void main(String[] args) throws InterruptedException{
68
69        if(args.length <= 2)
70        {
71            try
72            {
73                readSettings("src/MyHTTPServer/settings.txt");
74                ServerSocket socketServidor = new ServerSocket(HTTPServer.HTTP_PORT);
75                System.out.println("Server started. \nListening for connections on port: " + HTTPServer.HTTP_PORT);
76                Thread thread = null;
77                while(true)
78                {
79                    if(Thread.activeCount() <= MAX_CONNECTIONS)
80                    {
81                        Socket clientSocket = socketServidor.accept();
82                        HTTPServer myServer = new HTTPServer(clientSocket);
83                        thread = new Thread(myServer);
84                        thread.start();
85                    }
86                }
87            }
88            catch(Exception e)
89            {
90                System.err.println("SERVER CONNECTION ERROR: " + e.getMessage());
91            }
92        }
93        else
94        {
95            System.err.println("ERROR ARGS: /.class <PORT> <MAX_CONNECTIONS> ");
96        }
97    }
98
```

El método run se encarga del flujo del programa. El servidor solo atiende a métodos GET y SET. El servidor se encarga de leer lo que el cliente desea, es decir, leer el socket con el navegador, procesar el mensaje, establecer conexión con el controller, hacer una petición al controller, recibir la respuesta y transmitirla al navegador.

```
100    @Override
101    public void run() {
102
103        BufferedReader br = null;           //BufferReader
104        try
105        {
106            br = new BufferedReader(new InputStreamReader(connect.getInputStream()));
107            String[] input = new String(br.readLine()).split(" ");
108
109            if(input[0].equals("GET") || input[0].equals("SET") )
110            {
111                if(input[1].equals("/")) input[1] += DEFAULT_FILE;
112                if(input[1].contains("/controlSD"))servicioDinamico(input[1]);
113                else servicioEstatico(input[1]);
114
115            }
116            else
117            {
118                sendMessageHTTPToClient(0, 405, null); //ERROR 404 FILE NOT FOUND
119
120            }
121        }
122        catch(FileNotFoundException f)
123        {
124            String s = "/";
125            s += NOT_FOUND;
126            File file = new File(ROOT, s);
127            int fileLength = (int)file.length();
128            byte[] fileData;
129            try
130            {
131                fileData = readFileData(file, fileLength);
132                sendMessageHTTPToClient(fileLength, 404, fileData);     // 404 FILE NOT FOUND
133
134            } catch (IOException e) {
135                System.err.println("ERROR IO EXCEPTION: " + e.getMessage());
136            }
137
138            System.err.println("ERROR FILE NOT FOUND: " + f.getMessage());
139        }
140        catch(IOException e)
141        {
142            System.err.println("ERROR IO EXCEPTION: " + e.getMessage());
143        }
144    }
```

Este método se encarga de hacer los mensajes atendiendo al protocolo HTTP, según el estado de la respuesta.

```
207    private void sendMessageHTTPToClient(int fileLength, int cod, byte[] fileData) throws IOException {
208
209        PrintWriter out = null;
210        DataOutputStream bw = null;
211        try
212        {
213            bw = new DataOutputStream(connect.getOutputStream());
214            out = new PrintWriter(connect.getOutputStream());
215
216            if(cod == 200)//OK
217            {
218                out.println("HTTP/1.1 200 OK");     //We send HTTP Headers with data to client
219                out.println("Server: HTTPServer by STALYN ALEJNDRO : 1.0");
220                out.println("Date: " + new Date());
221                out.println("Content-type: " + "text/html");
222                out.println("Content-length: " + fileLength);
223            }
224            else if(cod == 404)//FILE NOT FOUND
225            {
226                out.println("HTTP/1.1 404 File Not Found");
227                out.println("Server: HTTPServer by STALYN ALEJNDRO : 1.0");
228                out.println("Date: " + new Date());
229                out.println("Content-type: " + "text/html");
230                out.println("Content-length: " + fileLength);
231            }
232            else if(cod == 405)//METHOD NOT ALLOWED
233            {
234                out.println("HTTP/1.1 405 METHOD_NOT_ALLOWED");
235                out.println("Server: HTTPServer by STALYN ALEJNDRO : 1.0");
236                out.println("Date: " + new Date());
237                out.println("Content-type: " + "text/html");
238                out.println("Content-length: " + fileLength);
239            }
240            else if(cod == 409)//CONTROLLER CONEXIÓN LOST
241            {
242                out.println("HTTP/1.1 409 CONTROLLER_CONEXION_LOST");
243                out.println("Server: HTTPServer by STALYN ALEJNDRO : 1.0");
244                out.println("Date: " + new Date());
245                out.println("Content-type: " + "text/html");
246                out.println("Content-length: " + fileLength);
247            }
248            out.println();  // Blank line between headers and content, very important !
249            out.flush();    // Flush character output stream buffer
250            bw.write(fileData, 0, fileLength);
251            bw.flush();
252
```

Hay dos tipos de servicios, estáticos y dinámicos. Estos son los métodos que los implementan.

```
146    private void servicioEstatico(String request) throws IOException {
147
148        File file = new File(ROOT, request);
149        int fileLength = (int)file.length();
150        byte[] fileData = readFileData(file, fileLength);
151        sendMessageHTTPToClient(fileLength, 200, fileData); //OK
152    }
153
154
155    private void servicioDinamico(String request) throws IOException {
156
157        Socket controllerSocket = null;
158        try
159        {
160            controllerSocket = new Socket(HTTPServer.CONTROLLER_IP, HTTPServer.CONTROLLER_PORT);
161            messageToController(controllerSocket, request);
162            System.out.println("servicioDinámico: " + request);
163            String ans = receiveMessageFromController(controllerSocket);
164
165             File file = new File(ROOT, DEFAULT_FILE);
166             BufferedReader br = new BufferedReader(new FileReader(file));
167
168             char h = '"';
169             String st;
170             String res = "";
171             String stt = "<label id="+h+"respuesta"+h+">"+ans+"</label>";
172             while ((st = br.readLine()) != null)
173             {
174                 if(st.contains("respuesta"))res += stt;
175                 else res += st;
176             }
177
178            byte[] b = res.getBytes();
179            sendMessageHTTPToClient(res.length(), 200, b);
180            if(ans.contains("error"))sendMessageHTTPToClient(0, 404, null); //404 : NOT FOUND
181            controllerSocket.close();
182        }
183        catch(IOException e)
184        {
185            System.out.println("ERROR 409");
186            sendMessageHTTPToClient(0, 409, null);  //409 : CONTROLLER CONEXION FAILED
187        }
188    }
```

Guia de despliegue.

0. Primero compilamos las classes.

javac *.java

1. Para el rmiregistry. Debe de ejecutarse en el directorio src. Si no especificamos puerto, por defecto es el 1099 de la máquina en la que estamos.

rmiregistry <puerto> &

2. Para el registro. Es importante que se ejecute en el mismo directorio que rmiregistry. La ip de hostname es donde se encuentra rmiregistry.

java -cp <directorio> -Djava.rmi.server.hostname=<ip> Registrador <ip> <puerto>

3. Para el Sensor. El ip hostname es la ip y el puerto donde se encuentra el Registro. El archivo es de tipo txt.

java -cp <directorio> -Djava.rmi.server.hostname=<ip> Sensor <ip> <puerto> <archivo>

4. Para el servidor. IP y puerto donde se ejecuta el servidor. Los demás parámetros los lee en un fichero de texto ‘settings.txt’.

java -cp <directorio> HTTPServer <ip> <puerto> <MaxConexiones>

5. Para el Controller. Los parámetros del controller como la ip y el puerto, las he especificado en el servidor http. Esa es la razón por la que ejecuto primero el Servidor.

j**ava -cp <directorio> Controller**