**Daniel Martí Santos**

**46080194Q**

**Viernes 09:00-11:00**

**Año 2019-2020**

# Práctica sockets y RMI

# Código común a varias subclases.

*Éste código se usa en varias clases para mandar y recibir mensajes.*

```
private String recibirMensaje(Socket s) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(rs.getInputStream()));
    StringBuilder sb = new StringBuilder();
    String st;
    while ((st = br.readLine()) != null && !st.equals(""))
        sb.append(st).append("\n");
    return sb.toString();
}

private void enviarMensaje (Socket s, String mensaje) throws IOException {
    DataOutputStream os = new DataOutputStream(s.getOutputStream());
    os.writeBytes(mensaje);
}
```

# Servidor HTTP

***El servidor HTTP se encarga de recibir peticiones, cada vez que se recibe una petición se genera un hilo que la procesa.***

***El servidor debe comprobar las siguientes cosas:***

***1. Comprobar que el método solicitado es GET, si no, mandar un error 405.***

***2. Si se pide un recurso estático, se abrirá el archivo y se mostrará, en el caso de que no exista el archivo, se mostrará el error 404.***

***3. Si se pide un recurso dinámico se contactará con el controlador para que nos dé la información, si no se puede contactar con el controlador se manda un error 409.***

***Se usan varios archivos para implementar el servidor HTTP:***

## MyHTTPSettings.

***Es una clase que usarán las demás clases (MyHttpServer, HttpThread, Controller y ControllerThread) para guardar los datos de la conexión (número de conexiones, ip's y puertos).***

```
public class MyHttpSettings {
```

```
    public String ip_servidor;
    public int puerto_servidor;
    public int conexiones_maximas;
    public int puerto_controlador;
    public int puerto_registro;
    public String ip_controlador;
    public String ip_registro;
    public final String contenido_dinamico = "controladorSD";
    public final String nombre_sondas = "sonda";
}
```

## MyHttpServer

**Es la clase que se ejecuta y a la que se le pasan los parámetros necesarios para la ejecución.**

**Se encarga de iniciar el valor de las variables.**

```
    sett.puerto_servidor = Integer.parseInt(args[0]);
    sett.conexiones_maximas = Integer.parseInt(args[1]);
    sett.ip_controlador = args[2];
    sett.puerto_controlador = Integer.parseInt(args[3]);
    sett.ip_registro = args[4];
    sett.puerto_registro = Integer.parseInt(args[5]);
    sett.ip_servidor = args[6];
```

**También se encarga de generar el ServerSocket y de crear los procesos hijo.**

```
    ServerSocket ss = new ServerSocket();
    ss.bind(new InetSocketAddress(sett.ip_servidor, sett.puerto_servidor));
    System.out.println("Servidor abierto en la IP: " + ss.getInetAddress());
    while(verdad == true){
        rs = ss.accept();
        System.out.println("Conectado el cliente con la IP: " + rs.getRemoteSocketAddress());
        if(Thread.activeCount() <= sett.conexiones_maximas) {
            Thread t = new HttpThread(sett, rs);
            t.start();
        }
        else {
            System.out.println("Ya se han llegado al máximo de conexiones");
```

```
            rs.close();
        }
```

## HttpThread

Es la clase principal del servidor. Se encargar de gestionar las peticiones y mostrar el resultado a los usuarios.

Esta parte del código se encarga de comprobar que el método pedido es GET y de comprobar si el recurso pedido es dinámico o estática.

```
String peticion = recibirMensaje(rs);
    String[] partesPeticion = peticion.split(" ");
    if (partesPeticion[0].equals("GET")){
        String URL = partesPeticion[1];
        if (URL.equals("/")){
            URL += "index.html";
        }
        String[] routeParts = URL.split("/");
        if (routeParts[1].equals(sett.contenido_dinamico)){
            if(routeParts.length == 3){
                serveDynamicRequest(routeParts[2]);
            }


            else
                serveDynamicRequest("");
        }
        else
            serveStaticRequest(routeParts[1]);
    }
    else {
        System.err.println("El cliente con la IP " + rs.getRemoteSocketAddress() + "ha recibido un error por intentar usar el método: " + partesPeticion[0]);
        mandarRespuestaHTTP(HTMLEntityEncode(conseguirErrorHTML(405)),405, "Method Not Allowed");
    }
```

Cuando se comprueba si solicita un recurso estático o dinámico se ejecuta otro módulo dependiendo del tipo de recurso.

```
   private void serveDynamicRequest(String query) throws IOException {
      try {
         String respuesta;
         Socket s = new Socket(sett.ip_controlador, sett.puerto_controlador);
         enviarMensaje(s, query + "\n\r");
         respuesta = recibirMensaje(s);
         s.close();
         if(!respuesta.replaceAll("\\n","").equals("null")){
            mandarRespuestaHTTP(respuesta, 200, "OK");
         }
         else
            throw new IOException();
      } catch (IOException e){
         mandarRespuestaHTTP(HTMLEntityEncode(conseguirErrorHTML(409)), 409,
"Conflict");
      }
   }


   private void serveStaticRequest(String path) throws IOException {
      try{
         mandarRespuestaHTTP(new String(Files.readAllBytes(Paths.get(path))) + "\n",
200,"OK");
      } catch (IOException e) {
         try {
            mandarRespuestaHTTP(HTMLEntityEncode(conseguirErrorHTML(404)), 404, "Not
Found");
         } catch (IOException e2) {
            e2.printStackTrace();
         }


         System.err.println("El cliente con la IP" + rs.getRemoteSocketAddress() + " ha
conseguido el error 404 por no encontrar el archivo:  " + path);
      }
   }
```

Éstos tres módulos se encargan de dar el formato necesario a los archivos y al contenido (caracteres especiales) para realizar la petición HTTP.

```
    private void mandarRespuestaHTTP(String fileContent, int statusCode, String descr) throws
IOException {
        String response = "HTTP/1.1 " + statusCode + " " + descr + "\n" +
                "Connection: close\n" +
                "Content-Length: " + fileContent.length() + "\n" +
                "Content-Type: text/html\n" +
                "Server: practicas-sd\n" +
                "\n" + //Headers end with an empty line
                fileContent;
        enviarMensaje(rs, response);
    }

    public  static String HTMLEntityEncode(String s) {
        StringBuilder builder = new StringBuilder ();
        for (char c : s.toCharArray())
            builder.append((int)c < 128 ? c : "&#" + (int) c + ";");
        return builder.toString();
    }

    public String conseguirErrorHTML(int codigo) throws IOException{
        return HTMLEntityEncode(new String(Files.readAllBytes(Paths.get(codigo + ".html"))) +
"\n\r");
    }
}
```

# Controlador

Se encarga de realizar las peticiones y mandar la información entre el servidor HTTP y las sondas.

## Controller

Su comportamiento es similar al MyHttpServer, se encarga de inicializar los sockets y las variables necesarias para la comunicación entre las distintas partes

## ControllerThread

Se encarga de la comunicación entre el servidor HTTP y el registrador.

Primero se encarga de ver la petición que hay, si la petición está en blanco devuelve una lista de las sondas.

```
public void run() {
    try{
        String respuesta;
        registry = LocateRegistry.getRegistry(sett.ip_registro, sett.puerto_registro);
        String consulta = recibirMensaje(rs).replaceAll("\\n","");
        System.out.println("Consulta = " + consulta);
        if (consulta.equals("")){
            respuesta = mandarSondasDisponibles();
        }
        else{
            respuesta = procesarConsulta(consulta);
        }
        enviarMensaje(rs,respuesta + "\n");
        rs.close();
    }catch (IOException e){
        e.printStackTrace();
    }
}
```

Hay un método encargado de comprobar que clase de consulta es, y se encarga de mandar la respuesta correcta.

```
private String procesarConsulta(String query) {
    StringBuilder respuesta = new StringBuilder();
    String[] partes = query.split("\\?");
    if (partes.length == 2) {
        String recurso = partes[0];
        String[] parametros = partes[1].split("&");
        for (String parametro : parametros) {
            String[] partesParametros = parametro.split("=");
            if (partesParametros.length == 2) {
                String variable = partesParametros[0];
```

```
            String valor = partesParametros[1];
            if (variable.equals(sett.nombre_sondas)){
               respuesta.append("<p>[").append(valor).append(",").append(recurso).append("]
= ").append(conseguirDatosSonda(valor, parametro)).append("</p>\n");
            }
         }
      }
      respuesta.append("<a href=\"../\">Inicio</a>");
      return respuesta.toString();
   }
   return null;
}
```

Otro método se encarga de conseguir los datos de la sonda en caso de ser necesarios en la petición.

```
   private String conseguirDatosSonda(String sonda, String recurso) {
   int parametro;

   try{
      Object valor;
      Object objetoRemoto = registry.lookup(sonda);
      ServiciosSonda so = (ServiciosSonda) objetoRemoto;
      if (objetoRemoto instanceof Sonda) {
         if (recurso.contains("=")) {
            String[] partesRecurso = recurso.split("=");
            try {
               parametro = Math.max(0, Integer.parseInt(partesRecurso[1]));
            } catch (NumberFormatException e) {
               parametro = 0;
            }
            so.getClass().getMethod(partesRecurso[0], int.class).invoke(sonda,parametro);
            valor = parametro;
         }
         else {
            valor = sonda.getClass().getMethod(recurso).invoke(sonda);
         }
         return valor.toString();
```

```
        }
```

Hay un método que se encarga de mandar la lista de sondas que hay disponibles:

```
    private String mandarSondasDisponibles() {
        try {
            StringBuilder nombres = new StringBuilder();
            nombres.append("<p>Lista de nombres de sensores disponibles:</p>\n");
            for (String remoteName : registry.list()){
                if(registry.lookup(remoteName) instanceof Sonda){
                    nombres.append("<p>").append(remoteName).append("</p>\n");
                }
            }
                nombres.append("<p><a href=\"../\">Inicio</a></p>\n");
                return nombres.toString();
            } catch (RemoteException | NotBoundException e) {
                return "<h1>El controlador no se ha podido conectar con los sensores </h1>" +
"<p><a href=\"../\">Inicio</a></p>\n";
            }
        }
    }
```

# Registros y sensores

Los registros se realizan a través de RMI. Los sensores están controlados por éste protocolo, a la vez que están conectados al controlador.

Hay unos métodos que usarán tanto los sensores como el registrador.

```
public interface RMIServices extends Remote {
    int volumen() throws RemoteException;
    String fecha() throws RemoteException;
    String ultimafecha() throws RemoteException;
    int luz() throws RemoteException;
    void setluz(int valor) throws RemoteException;
}
```

## Register

Se encarga de registrar las sondas (tiene que estar en el mismo ordenador que el rmiregistry).

Al iniciar ésta clase se le pasa la IP del propio ordenador y el puerto por el que debe escuchar.

```
public static void main(String[] args) throws Exception {
    if (args.length >= 2) {
        Registry registry = LocateRegistry.getRegistry(args[0], Integer.parseInt(args[1]));
        Register master = new Register(registry);
        registry.rebind(Register.nombreRMI, master);
    }
    else
        System.out.println("Argumentos: IP Puerto");
}
```

Además también tiene un método encargado de registrar las sondas.

```
public void registrarSonda(ServiciosSonda ss) throws RemoteException {
    try {
        registry.rebind(ss.conseguirNombreRMI(), ss);
    } catch (RemoteException e) {
        e.printStackTrace();
                throw e;
    }
}
```

## Sonda

Las sondas se encargan de cualquier cosa relacionada con los atributos asociados a estas. Cada sonda tiene asociado un fichero desde el que lee y modifica las variables. Además de solicitar la conexión al register.

Código para registrarse:

```
public static void main(String[] args) throws Exception {
    if (args.length >= 3) {
            registry = LocateRegistry.getRegistry(args[0], Integer.parseInt(args[1]));
            sonda = new Sonda(args[2]);
            registro = (RegisterServices) registry.lookup(Register.nombreRMI);
            registro.registrarSonda(sonda);
        }
    else{
```

```
            System.out.println("Uso de los argumentos: IP Puerto Nombre_de_archivo");
        }
    }
```

Código para leer el archivo :

```
    private void readFile (String fileName) throws IOException, NoSuchFieldException,
IllegalAccessException{
        for (String s : Files.readAllLines(Paths.get(fileName))) {
            String [] campos = s.split("=");
            Field f = this.getClass().getDeclaredField(campos[0]);
            if (f.getType().isAssignableFrom(String.class)){
                f.set(this, campos[1]);
            }
            else if (f.getType().isAssignableFrom(int.class)){
                f.set(this, Integer.parseInt(campos[1]));
            }
        }
    }
```

Código para modificar el archivo

```
    private void saveFile() {
        try (BufferedWriter bw = Files.newBufferedWriter(Paths.get(nombreRMI + ".txt"))) {
            for (Field f : this.getClass().getDeclaredFields()){
                if(!Modifier.isPublic(f.getModifiers())){
                    bw.write(f.getName() + "=" + f.get(this) + System.lineSeparator());
                }
            }
        }catch (IOException | IllegalAccessException e){
                    e.printStackTrace();
        }
    }
```

:

También tiene los métodos Set y Get de las distintas variables. El nombre de la sonda se consigue a través del nombre del archivo que contiene los datos sin la extensión.

# Guía de despliegue:

Tanto el controlador como el servidor se pueden ejecutar en cualquier momento del proceso, ya que no afecta al resultado.

Para una mayor comodidad, y no tener que indicar que comando se tiene que ejecutar en la carpeta con los código fuente y cual no, se recomienda que se ejecuten todos los comandos en la carpeta en la que estén los archivos con el código fuente.

## Register

Si no se tiene el archivo Register.class, habrá que compilar el archivo Register.java con el siguiente comando:

**javac Register.java**

Para ejecutar el register tenemos que ejecutar el rmiregistry, el puerto que le pasemos tiene que ser el mismo que indicamos a la hora de ejecutar el servidor HTTP y el controlador.

El código useCodeBaseOnly=false evita que se tengan que generar los stubs.

Comando para ejecutar el rmiregistry: **rmiregistry (puerto) -J-Djava.rmi.server.useCodeBaseOnly=false**

Ahora se ejecuta el registrado con el siguiente comando:

**java -cp . -Djava.rmi.server.hostname=(IP) Register (IP) (Puerto)**

## Sonda

Al igual que con el registrador, si no se tiene el archivo Sonda.class se compila con el comando:

**javac Sonda.java**

Después de ejecutar el registrador, se registran las sondas con el siguiente comando:

**java -cp . -Djava.rmi.server.hostname=(IP) Sonda (IP) (Puerto) (Archivo_sonda)**

## Controlador y servidor

Para ejecutar el servidor y el controlador, si no se tienen los archivos MyHttpServer.class y Controler.class respectivamente, habrá que ejecutar los siguientes comandos:

**javac Controler.java**

**javac MyHttpServer.java**

Para ejecutar el controlador es necesario el siguiente comando:

**java Controler (Puerto_servidor) (Máximo_de_conexiones) (IP_controlador) (Puerto_controlador) (IP_registro) (Puerto_registro)**

Para ejecutar el servidor se usa un comando muy parecido:

**java MyHttpServer (Puerto_servidor) (Máximo_de_conexiones) (IP_controlador) (Puerto_controlador) (IP_registro) (Puerto_registro) (IP_servidor)**