



CURSO 2016-2017

SISTEMAS DISTRIBUIDOS – PRÁCTICA 1

CONTROL PARKING SOCKETS Y RMI

PAVEL RAZGOVOROV

Y0888160-Y | pr18@alu.ua.es

Turno de lunes 11:00-13:00

Contenido

Introducción	2
Servidor HTTP	2
MyHTTPSettings	3
SocketUtils.....	3
MyHTTPCodeStatus.....	4
MyHTTPServer.....	5
MyHTTPThread.....	6
Controlador	8
Controller	8
ControllerThread	8
RMI – Register y Sensores	11
Register	11
La IP que recibe debe de ser la propia; rmiregistry y Register deben de estar en la misma máquina (por razones de seguridad).Sensor	11
Guía de despliegue.....	14
Register	14
Sensor	14
Controlador y Servidor HTTP.....	15

Introducción

Esta es la implantación de un sistema de sensores para el control de plazas de un parking. Los sensores medirán una serie de datos de su entorno y dispondrán de atributos que informen de su estado:

- Luz: El código del LED que indicará si una plaza esté libre, ocupada o deshabilitada.
- Fecha: Fecha en la que se realizó una modificación.
- Volumen: volumen detectado en la plaza de garaje.

Estos sensores estarán conectados a una estación central, Controller, la cual podrá acceder a éstos para consultar su información, así como modificarla.

Por otra parte, existirá un mini-servidor HTTP que alojará una interfaz para contactar con los sensores la cual será transmitida a través de un navegador web conectado al servidor.

El modo de comunicación entre un componente y otro es el siguiente:

- Navegador-Servidor: Sockets HTTP
- Servidor-Controlador: Sockets
- Control-Sensores: RMI

Todas las comunicaciones serán concurrentes y se podrá parametrizar el número de conexiones máximas entre el navegador y el servidor, así como el servidor y el controlador.

Servidor HTTP

Tendremos que realizar un servidor parecido al que aparece en la práctica guiada de sockets (concurrente, en el que se especifica el puerto de escucha), excepto por que en este debemos de controlar el máximo número de conexiones simultáneas. En la lógica de cada hilo deberemos de recibir una petición HTTP y procesarla. Los pasos del proceso son estos:

1. Comprobar que el método que pide es GET; si no, enviar un error HTTP 405 Method Not Allowed.
2. Comprobar qué recurso pide, si es dinámico (información de los sensores) o estático (un archivo). Si no hay ningún recurso solicitado, por defecto mandaremos una página de índice.
 - a. Si el recurso es dinámico, enviar la consulta al Controlador para que la procese y nos envíe una respuesta que nosotros transmitiremos al navegador. Si el controlador no está conectado, enviar un error HTTP 409 Conflict
 - b. Si es estático, abrir el archivo que solicita, leerlo y transmitirlo. Si no existe, enviar un error HTTP 404 Not Found.

La implementación que hemos realizado es la siguiente:

MyHTTPSettings

Ésta es una clase de utilidad que utiliza tanto el servidor HTTP como el Controlador para tener a mano todos los parámetros de configuración que debe de conocer para su puesta en marcha. Tiene un método que lee la configuración de un fichero JSON (utilizando la librería gson) y asigna sus valores a las variables cuyo nombre coincida con la clave del elemento JSON.

```

1. public int MAX_SERVER_CONNECTIONS;
2. public int SERVER_PORT;
3. public int CONTROLLER_PORT;
4. public int REGISTRY_PORT;
5. public String CONTROLLER_IP;
6. public String REGISTRY_IP;
7. public String DYNAMIC_CONTENT_KEYWORD;
8. public String DEFAULT_INDEX_PAGE;
9. public String SENSOR_KEYWORD;

1. public static MyHTTPSettings readConfig() throws FileNotFoundException {
2.     Gson gson = new Gson();
3.     JsonReader reader = new JsonReader(new FileReader("server-config.json"));
4.     return gson.fromJson(reader, MyHTTPSettings.class);
5. }
```

NOTA: UTILIZAMOS LA LIBRERÍA DE GOOGLE GSON PARA LEER EL JSON

También tenemos un método toString() para visualizar la configuración leída.

```

1. public String toString() {
2.     return "MyHTTPSettings{" +
3.         "MAX_SERVER_CONNECTIONS=" + MAX_SERVER_CONNECTIONS +
4.         ", SERVER_PORT=" + SERVER_PORT +
5.         ", CONTROLLER_PORT=" + CONTROLLER_PORT +
6.         ", CONTROLLER_IP=" + CONTROLLER_IP + '\n' +
7.         ", DYNAMIC_CONTENT_KEYWORD=" + DYNAMIC_CONTENT_KEYWORD + '\n' +
8.         ", DEFAULT_INDEX_PAGE=" + DEFAULT_INDEX_PAGE + '\n' +
9.         ", SENSOR_KEYWORD=" + SENSOR_KEYWORD + '\n' +
10.        '}';
11. };
```

SocketUtils

Ésta también es una clase de utilidad que usan los dos componentes anteriormente mencionados. Tiene tres métodos: mostrar las IP's del equipo, recibir mensaje y enviar mensaje:

```

1. public static void displayIPAddresses() {
2.     try {
3.         System.out.println("Available IP's at this machine");
4.         for (NetworkInterface network : Collections.list(NetworkInterface.getNetworkInterfaces()))
5.             Collections
6.                 .list(network.getInetAddresses()).stream()
7.                 .filter(adress -> adress instanceof Inet4Address)
8.                 .forEach(adress -> System.out.println("\t" + adress));
9.     } catch (SocketException e) {
10.        System.err.println("Error: unable to display available IP's");
11.    }
12. }
```

```

1. public static String receiveMessage(Socket socket) throws IOException {
2.     BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
3.     StringBuilder sb = new StringBuilder();
4.     String s;
5.     while ((s = br.readLine()) != null && !s.equals(""))
6.         sb.append(s).append("\n");
7.     return sb.toString();
8. }

```

```

1. public static void sendMessage(Socket socket, String message) throws IOException {
2.     DataOutputStream outputStream = new DataOutputStream(socket.getOutputStream());
3.     outputStream.writeBytes(message);
4. }

```

Estos métodos son esenciales en la comunicación entre sockets (en especial los dos últimos).

MyHTTPCodeStatus

También se trata de una clase de apoyo. En este caso, es una enumeración de códigos de HTTP con los que podemos disponer de una interfaz para trabajar con ellos de una manera cómoda disponiendo de un nombre en clave para cada error del que podemos extraer su código, su descripción y su página HTML asociada que deberemos de enviar si ese error:

```

1. enum MyHTTPStatusCode {
2.     OK (200, "OK"),
3.     NOT_FOUND (404, "Not Found"),
4.     METHOD_NOT_ALLOWED (405, "Method Not Allowed"),
5.     CONFLICT (409, "Conflict"),
6.     INTERNAL_SERVER_ERROR (500, "Internal Server Error");
7.
8.     private final int code;
9.     private final String description;
10.
11.     MyHTTPStatusCode(int code, String description) {
12.         this.code = code;
13.         this.description = description;
14.     }
15.
16.     public int getCode() { return code; }
17.     public String getDescription() { return description; }
18.
19.     @NotNull
20.     public String getHTMLError() throws IOException {
21.         return MyHTTPThread.HTMLEntityEncode(new String(Files.readAllBytes(Paths.get(code + ".html")))) + "\n\r");
22.     }
23. }

```

MyHTTPServer

Es la clase que recibe la configuración del servidor HTTP y lo inicia.

```
1. public static void main(String args[]) {
2.     try {
3.         MyHTTPSettings settings = MyHTTPSettings.readConfig();
4.         System.out.println("MyHTTP-
   Server setup complete. Server info: " + settings);
5.         init(settings);
6.     } catch (FileNotFoundException e) {
7.         System.err.println("Server's config file not found");
8.     }
9. }
```

A la hora de iniciarlo, muestra las IP's disponibles en el ordenador a través de las que nos podemos conectar, abre el Socket del servidor y espera a que alguien se conecte. Cuando lo hace, comprueba que no se haya superado el número de conexiones simultáneas y abre un nuevo hilo que es donde se procesará la petición.

```
1. private static void init(MyHTTPSettings settings) {
2.     try {
3.         SocketUtils.displayIPAddresses(); //Shows all available IP's that can connect to the settings
4.         ServerSocket serverSocket = new ServerSocket(settings.SERVER_PORT);
5.         System.out.println("Socket opened at port " + serverSocket.getLocalPort());
6.     }
7.     while (true) { //Server is stopped with ^C
8.         Socket requestSocket = serverSocket.accept();
9.         System.out.println("Serving to " + requestSocket.getRemoteSocketAddress());
10.        if (Thread.activeCount() <= settings.MAX_SERVER_CONNECTIONS) {
11.            Thread t = new MyHTTPThread(settings, requestSocket);
12.            t.start();
13.        }
14.        else {
15.            System.err.println("Connection with " + requestSocket.getRemoteSocketAddress() + " closed due to connection limit");
16.            requestSocket.close();
17.        }
18.    }
19. } catch (IOException e) {
20.     System.err.println("Unable to open socket");
21. }
22. }
```

MyHTTPThread

La clase principal del servidor HTTP, implementa la lógica de negocio de cada petición que se debe de manejar (su funcionamiento lo hemos explicado anteriormente).

```

1. @Override
2. public void run() {
3.     try {
4.         String request = receiveMessage(requestSocket);
5.         String[] requestParts = request.split(" ");
6.         if (requestParts[0].equals("GET")) {
7.             String URL = requestParts[1];
8.             if (URL.equals("/")) URL += (settings.DEFAULT_INDEX_PAGE);
9.             String[] routeParts = URL.split("/");
10.            if (routeParts[1].equals(settings.DYNAMIC_CONTENT_KEYWORD))
11.                serveDynamicRequest(routeParts.length == 3 ? routeParts[2] : "");
12.            else serveStaticRequest(routeParts[1]);
13.        } else {
14.            System.err.println("ERROR 405 for client " + requestSocket.getRemoteSocketAddress() + ". Requested method: " + requestParts[0]);
15.            sendHTTPResponse(MyHTTPStatusCode.METHOD_NOT_ALLOWED.getHTMLError(), MyHTTPStatusCode.METHOD_NOT_ALLOWED);
16.        }
17.        requestSocket.close();
18.    } catch (IOException e) {
19.        System.err.println("ERROR: Unable to open inputStream from socket");
20.        e.printStackTrace();
21.    }
22. }

```

El método para servir la petición dinámica se conecta con el controlador, le envía la petición, recibe la respuesta y la transmite por el socket HTTP. Si no se puede establecer contacto o bien éste envía una respuesta vacía, se enviará un error HTTP 409 Conflict.

```

1. private void serveDynamicRequest(String query) throws IOException {
2.     try {
3.         Socket s = new Socket(settings.CONTROLLER_IP, settings.CONTROLLER_PORT);
4.         sendMessage(s, query + "\n\r");
5.         String response = receiveMessage(s);
6.         s.close();
7.         if (!response.replaceAll("\\n", "").equals("null")) sendHTTPResponse(response, MyHTTPStatusCode.OK);
8.         else throw new IOException();
9.     } catch (IOException e) {
10.        sendHTTPResponse(MyHTTPStatusCode.CONFLICT.getHTMLError(), MyHTTPStatusCode.CONFLICT);
11.    }
12. }

```

El método para servir la petición estática simplemente lee el fichero solicitado, le aplica un filtro que codifica los caracteres especiales del HTML y llama al método auxiliar `sendHTTPResponse`:

```
1. private void serveStaticRequest(String path) throws IOException {
2.     try {
3.         sendHTTPResponse(HTMLEntityEncode(new String(Files.readAllBytes(Paths.get(
4.             path)))) + "\n", MyHTTPStatusCode.OK);
5.     } catch (IOException e) {
6.         try {
7.             sendHTTPResponse(MyHTTPStatusCode.NOT_FOUND.getHTMLError(), MyHTTPStatus
8.                 Code.NOT_FOUND);
9.         } catch (IOException e1) { e1.printStackTrace(); }
10.        System.err.println("Client " + requestSocket.getRemoteSocketAddress() + "
11.            got ERROR 404 for file " + path);
12.    }
13. }
```

El método auxiliar `HTMLEntityEncode` es quien codifica los caracteres especiales del fichero leído:

```
1. static String HTMLEntityEncode(String s) {
2.     StringBuilder builder = new StringBuilder ();
3.     for (char c : s.toCharArray()) builder.append(((int)c < 128 ? c : "&#" + (int)
4.         c + ";");
5.     return builder.toString();
6. }
```

Y, por último, `SendHTTPResponse` es quien envía la respuesta que se ha procesado la lógica de negocio:

```
1. private void sendHTTPResponse(String fileContent, MyHTTPStatusCode statusCode)
2.     throws IOException {
3.     String response = "HTTP/1.1 " + statusCode.getCode() + " " + statusCode.getDe
4.         scription() + "\n" +
5.         "Connection: close\n" +
6.         "Content-Length: " + fileContent.length() + "\n" +
7.         "Content-Type: text/html\n" +
8.         "Server: practicas-sd\n" +
9.         "\n" + //Headers end with an empty line
10.        fileContent;
11.     sendMessage(requestSocket, response);
12. }
```


Controlador

Este conjunto de clases realiza la función de intermediario entre el servidor HTTP y los sensores. Reutiliza las clases MyHTTPSettings y SocketUtils del anterior paquete.

Controller

Es un análogo de la clase MyHTTPServer, pero un par de cambios ligeros para aplicarle la configuración propia del controlador:

```

1. private static void init(MyHTTPSettings settings) {
2.     try {
3.         SocketUtils.displayIPAddresses(); //Shows all available IP's that can connect to the server
4.         ServerSocket serverSocket = new ServerSocket(settings.CONTROLLER_PORT);
5.         System.out.println("Socket opened at port " + serverSocket.getLocalPort());
6.     };
7.     //noinspection InfiniteLoopStatement
8.     while (true) { //Server is stopped with ^C
9.         Socket requestSocket = serverSocket.accept();
10.        System.out.println("Serving to " + requestSocket.getRemoteSocketAddress());
11.        if (Thread.activeCount() <= settings.MAX_SERVER_CONNECTIONS) {
12.            Thread t = new ControllerThread(settings, requestSocket);
13.            t.start();
14.        } else {
15.            System.err.println("Connection with " + requestSocket.getRemoteSocketAddress() + " closed due to connection limit");
16.            requestSocket.close();
17.        }
18.    }
19. } catch (IOException e) {
20.     System.err.println("Unable to open socket");
21. }
22. }
```

ControllerThread

Implementa la lógica de comunicación entre los componentes con los que interactúa. El método principal se conecta con el registrador, lee la petición, la procesa y devuelve una respuesta. Si no hay petición, devuelve un listado con los sensores disponibles:

```

1. @Override
2. public void run() {
3.     try {
4.         String response;
5.         registry = LocateRegistry.getRegistry(settings.REGISTRY_IP, settings.REGISTRY_PORT);
6.         String query = SocketUtils.receiveMessage(requestSocket).replaceAll("\\n", "");
7.         if (query.equals("")) response = sendAvailableSensors();
8.         else response = processQuery(query);
9.         SocketUtils.sendMessage(requestSocket, response + "\n");
10.        requestSocket.close();
11.    } catch (IOException e) {
12.        e.printStackTrace();
13.    }
14. }
```

El método que envía los sensores disponibles lo hace listando los nombres que tiene registrados y filtra por aquellos que efectivamente tengan la interfaz de un sensor:

```

1. @Nullable
2. private String sendAvailableSensors() {
3.     try {
4.         StringBuilder names = new StringBuilder();
5.         names.append("<p>Lista de nombres de sensores disponibles:</p>\n");
6.         for (String remoteName : registry.list())
7.             if (registry.lookup(remoteName) instanceof SensorServices) names.append
8.                 ("<p>").append(remoteName).append("</p>\n");
9.         names.append("<p><a href=\"../\">Inicio</a></p>\n");
10.        return names.toString();
11.    } catch (RemoteException | NotBoundException e) {
12.        return "<h1>El controlador no se ha podido conectar con los sensores</h1>"
13.            +
14.            "<p><a href=\"../\">Inicio</a></p>\n";
15.    }
16. }

```

El método que procesa la petición primero la lee y muestra en un listado todos aquellos sensores que se han solicitado y el recurso solicitado:

```

1. @Nullable
2. private String processQuery(String query) {
3.     StringBuilder response = new StringBuilder();
4.     String[] queryParts = query.split("\\?");
5.     if (queryParts.length == 2) {
6.         String resource = queryParts[0];
7.         String[] parameters = queryParts[1].split("&");
8.         for (String parameter : parameters) {
9.             String[] parameterParts = parameter.split("=");
10.            if (parameterParts.length == 2) {
11.                String key = parameterParts[0];
12.                String value = parameterParts[1];
13.                if (key.equals(settings.SENSOR_KEYWORD))
14.                    response.append("<p>[").append(value).append(",").append(resource).a
15.                        ppend("] = ")
16.                        .append(getSensorProperty(value, resource)).append("</p>\n");
17.            }
18.        }
19.        response.append("<a href=\"../\">Inicio</a>");
20.        return response.toString();
21.    }
22.    return null;
23. }

```

Por último, para obtener la propiedad de un objeto remoto, lo busco por su nombre e invoco el método que necesito con el nombre del recurso que estoy pidiendo a través de reflexión. Si se trata del método setLuz, separo su nombre del valor a introducir (el cual me aseguro de que sea un número positivo; si no, será 0) y lo invoco.

```
1. @Nullable
2. private String getSensorProperty(String sensorName, String resource) {
3.     try {
4.         Object returnedValue, remoteObject = registry.lookup(sensorName);
5.         if (remoteObject instanceof SensorServices) {
6.             SensorServices sensor = (SensorServices) remoteObject;
7.             if (resource.contains("=")) {
8.                 String[] resourceParts = resource.split("=");
9.                 int param;
10.                try {
11.                    param = Math.max(0, Integer.parseInt(resourceParts[1]));
12.                } catch (NumberFormatException e) { param = 0; }
13.                sensor.getClass().getMethod(resourceParts[0], int.class).invoke(sensor
14.                    , param);
15.                returnedValue = param;
16.            } else returnedValue = sensor.getClass().getMethod(resource).invoke(sensor);
17.            return returnedValue.toString();
18.        } catch (NotBoundException e) {
19.            //No existe el elemento en remoto
20.            return "No existe el sensor " + sensorName;
21.        } catch (NoSuchMethodException e) {
22.            //No existe el recurso solicitado
23.            return "No existe el recurso " + (resource.contains("=") ? resource.split(
24.                "=")[0] : resource) + "\n";
25.        } catch (RemoteException | IllegalAccessException | InvocationTargetException e) {
26.            return "El método con parámetro " + resource + " ha tenido un error y no se ha podido procesar\n";
27.        }
28.    }
29.    return null;
30. }
```

RMI – Register y Sensores

Los sensores a los que se conecta el Controlador están manejados por el protocolo RMI de Java. Para ello, debemos de manejar una estación central que actuará de registrador de sensores. Nos hemos basado [en esta página](#) para implementarlo.

La interfaz básica de los servicios que ofrecerá el RMI, que heredarán tanto la parte del registrador como la del sensor, es la siguiente:

```
1. public interface RMIServices extends Remote {
2.     int volumen() throws RemoteException;
3.     String fecha() throws RemoteException;
4.     String ultimafecha() throws RemoteException;
5.     int luz() throws RemoteException;
6.     void setluz(int value) throws RemoteException;
7. }
```

Register

Register es quien, estando en la misma máquina que el rmiregistry, se encarga de recibir peticiones de registro de los sensores para registrarlos (valga la redundancia). Tiene dos métodos esenciales: registrar y desregistrar.

```
1. @Override
2. public void registerSensor(SensorServices sensor) throws RemoteException {
3.     try {
4.         registry.rebind(sensor.getRMIName(), sensor);
5.         System.out.println("Registered: " + sensor.getRMIName());
6.     } catch (RemoteException e) {
7.         e.printStackTrace();
8.         throw e;
9.     }
10. }
```

```
1. @Override
2. public void unregisterSensor(SensorServices sensor) throws RemoteException {
3.     try {
4.         registry.unbind(sensor.getRMIName());
5.         System.out.println("Unregistered: " + sensor.getRMIName());
6.     } catch (NotBoundException e) {
7.         e.printStackTrace();
8.     }
9. }
```

Después, para iniciarlo, simplemente se conecta al rmiregistry y se autoregistra:

```
1. public static void main(String[] args) throws Exception {
2.     if (args.length >= 2) {
3.         final Registry registry = LocateRegistry.getRegistry(args[0], Integer.parseInt(args[1]));
4.         Register master = new Register(registry);
5.         registry.rebind(Register.RMI_NAME, master);
6.         System.out.println("Register OK -> " + args[0] + ":" + args[1]);
7.     } else System.out.println("Wrong args: <IP> <PORT>");
8. }
```

La IP que recibe debe de ser la propia; rmiregistry y Register deben de estar en la misma máquina (por razones de seguridad).

Sensor

Esta clase implementa la lógica de negocio del manejo de sensores. Su funcionamiento principal es el de crear el objeto (leyendo un fichero de configuración), pidiendo registrarse al Register y esperando una orden de desconexión. Cada vez que se pida un recurso, éste se proveerá del fichero. Si no puede acceder al fichero, muestra un error y devuelve el último valor cargado en memoria. Cuando reciba la orden de desconexión (un ENTER por teclado), pedirá desregistrarse y saldrá del programa.

```

1. public static void main(String[] args) throws Exception {
2.     if (args.length >= 3) {
3.         registry = LocateRegistry.getRegistry(args[0], Integer.parseInt(args[1]));
4.         try {
5.             System.out.println("Before reading");
6.             sensor = new Sensor(args[2]);
7.             System.out.println("Binding " + sensor.getRMName() + " through " + args
8. [0] + ":" + args[1]);
9.             registerServices = (RegisterServices) registry.lookup(Register.RMI_NAME)
10. ;
11.             registerServices.registerSensor(sensor);
12.             System.out.print("Press ENTER to disconnect ");
13.             new BufferedReader(new InputStreamReader(System.in)).readLine();
14.             System.out.println("Unregistering...");
15.             registerServices.unregisterSensor(sensor);
16.             System.out.println("Success");
17.             System.exit(0);
18.         } catch (IOException e) {
19.             System.err.println("Error reading file");
20.         }
21.     } else System.out.println("Wrong args: <IP> <PORT> <FILENAME>");
22.     System.exit(0);
23. }

```

Para crear el sensor, su nombre de RMI para el registro será el nombre del archivo, pero sin la extensión. Leerá los datos del fichero y asignará sus valores.

```

1. Sensor(String fileName) throws RemoteException {
2.     this.RMName = fileName.substring(0, fileName.lastIndexOf('.'));
3.     try {
4.         String s = new String(Files.readAllBytes(Paths.get(fileName)));
5.         readFile(fileName);
6.     } catch (IOException | NoSuchFieldException | IllegalAccessException e) {
7.         e.printStackTrace();
8.     }
9. }

```

Para consultar información del fichero, lo hace usando el fichero que leyó en un principio:

```
1. @Override
2. public int luz() throws RemoteException {
3.     try {
4.         readFile(RMName + ".txt");
5.     } catch (IOException | IllegalAccessException | NoSuchFieldException e) {
6.         e.printStackTrace();
7.     }
8.     return Led;
9. }
10.
11. @Override
12. public void setluz(int value) throws RemoteException {
13.     Led = value;
14.     saveFile();
15. }
```

Los métodos de escribir y leer se han implementado mediante reflexión:

```
1. private void readFile(String fileName) throws IOException, NoSuchFieldException, IllegalAccessException {
2.     for (String a : Files.readAllLines(Paths.get(fileName))) {
3.         String[] campos = a.split("=");
4.         Field f = this.getClass().getDeclaredField(campos[0]);
5.         if (f.getType().isAssignableFrom(String.class)) f.set(this, campos[1]);
6.         else if (f.getType().isAssignableFrom(int.class)) f.set(this, Integer.parseInt(campos[1]));
7.     }
8. }
9.
10. private void saveFile() {
11.     try (BufferedWriter bw = Files.newBufferedWriter(Paths.get(RMName + ".txt"))) {
12.         for (Field f : this.getClass().getDeclaredFields())
13.             if (!Modifier.isPublic(f.getModifiers())) bw.write(f.getName() + "=" + f.get(this) + System.lineSeparator());
14.     } catch (IOException | IllegalAccessException e) {
15.         e.printStackTrace();
16.     }
17. }
```

Guía de despliegue

Tanto el componente del servidor HTTP como el del Controlador son totalmente independientes el uno del otro. Pueden estar en marcha sin que el resto de componentes esté activo. Sin embargo, el registrador depende del rmiregistry (pues necesita un lugar donde registrarse) así como los sensores dependen del registrador para registrarse. Por tanto, el despliegue en orden será el que seguiremos punto tras punto:

Register

Primero deberemos de encender el rmiregistry pasándole como argumento el puerto (si no se lo pasamos, por defecto es el 1099) y el argumento de Java RMI useCodeBaseOnly=false para evitar tener que generar los stubs:

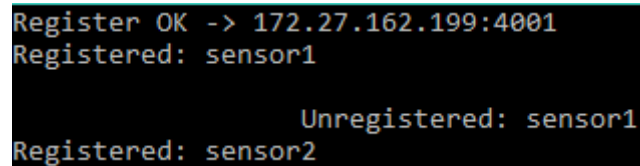
```
rmiregistry <puerto> -J-Djava.rmi.server.useCodeBaseOnly=false &
```

Es importante ejecutarlo desde el directorio raíz de la capreta en donde esté el código el proyecto (src).

Luego, ejecutamos el Register pasándole como argumento la IP y el puerto y como argumento de Java RMI la propia IP del host:

```
java -cp . -Djava.rmi.server.hostname=<mi IP>  
Parking.Registry.Register <IP> <Puerto>
```

El registrador estará listo y preparado para escuchar peticiones.



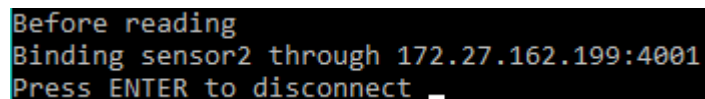
```
Register OK -> 172.27.162.199:4001  
Registered: sensor1  
  
Unregistered: sensor1  
Registered: sensor2
```

Sensor

Una vez tengamos el Registrador listo, pasaremos a registrar los sensores. Simplemente ejecutamos la clase pasándole como argumento la IP, el puerto del Registrador y el archivo del sensor y como argumento de Java RMI la propia IP del host:

```
java -cp . -Djava.rmi.server.hostname=<mi IP> Parking.Sensor.Sensor  
<IP> <Puerto> <Archivo>
```

Cuando queramos, pulsamos ENTER y pedirá una orden de desregistro al Registrador.



```
Before reading  
Binding sensor2 through 172.27.162.199:4001  
Press ENTER to disconnect _
```

Controlador y Servidor HTTP

Primero tenemos que asegurarnos de que el fichero server-config.json tenga todos sus parámetros correctamente indicados:

```
{
  "MAX_SERVER_CONNECTIONS" : 2,
  "SERVER_PORT" : 8081,
  "CONTROLLER_IP" : "172.20.42.7",
  "CONTROLLER_PORT" : 3001,
  "REGISTRY_IP" : "172.20.42.9",
  "REGISTRY_PORT" : 4001,
  "DYNAMIC_CONTENT_KEYWORD" : "controladorSD",
  "DEFAULT_INDEX_PAGE" : "index.html",
  "SENSOR_KEYWORD" : "sensor"
}
```

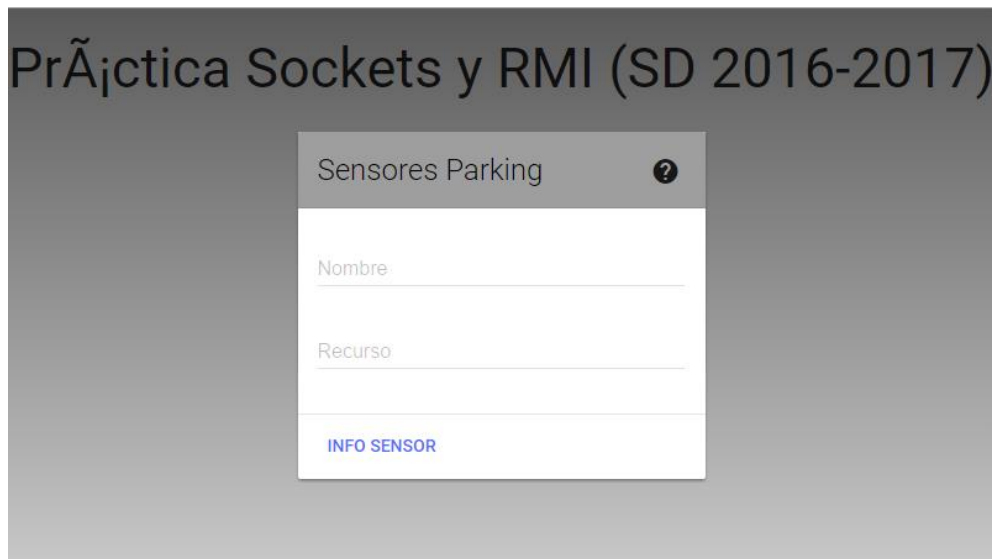
Después, es tan fácil como ejecutar el .jar correspondiente (que habremos generado desde nuestro entorno de desarrollo) y ya estaría en marcha:

```
java -jar Server.jar
java -jar Controller.jar
```

```
MyHTTP-Server setup complete. Server info: MyHTTPSettings{MAX_SERVER_CONNECTIONS=2,
SERVER_PORT=8081, CONTROLLER_PORT=3001, CONTROLLER_IP='172.20.42.7', DYNAMIC_CONTENT_KEYWORD='controladorSD', DEFAULT_INDEX_PAGE='index.html', SENSOR_KEYWORD='sensor'}
Available IP's at this machine
    /127.0.0.1
    /172.27.162.199
Socket opened at port 8081
```

```
Controller-Server setup complete. Server info: MyHTTPSettings{MAX_SERVER_CONNECTIONS=2, SERVER_PORT=8081, CONTROLLER_PORT=3001, CONTROLLER_IP='172.20.42.7', DYNAMIC_CONTENT_KEYWORD='controladorSD', DEFAULT_INDEX_PAGE='index.html', SENSOR_KEYWORD='sensor'}
Available IP's at this machine
    /127.0.0.1
    /172.27.162.199
Socket opened at port 3001
```


Con esto, ya podremos iniciar el cliente web:



Lista de nombres de sensores disponibles:

sensor1

[Inicio](#)

[sensor1,ultimafecha] = 31/10/2016 11:50:21

[Inicio](#)

[jhg,jkh] = No existe el sensor jhg

[Inicio](#)

Error 409: Conflict

Parece ser que el controlador no está online

Prueba volver al [inicio](#)

Error 404: Page Not Found

No hemos encontrado la página que pides

Prueba a volver al [inicio](#)