Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Monitorización del pulso de usuario mediante la lectura de sensores Bluetooth LE usando Eclipse Kura y AWS IoT

Autor: Francisco Javier Ortiz Bonilla

Tutora: María Teresa Ariza Gómez

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Monitorización del pulso de usuario mediante la lectura de sensores Bluetooth LE usando Eclipse Kura y AWS IoT

Autor:
Francisco Javier Ortiz Bonilla

Tutora:
María Teresa Ariza Gómez
Profesora titular

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Monitorización del pulso de usuario mediante la lectura de sensores Bluetooth LE usando Eclipse Kura y AWS IoT

Autor: Francisco Javier Ortiz Bonilla

Tutora: María Teresa Ariza Gómez

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis amigos

A mis maestros

Agradecimientos

Con este trabajo concluyo mis estudios como graduado en Ingeniería de las Tecnologías de Telecomunicación. Esto supone un importante hito para mí y un punto de inflexión en mi vida, y sin duda no podría haber llegado hasta aquí yo solo.

Sobre todo, me gustaría agradecer a mis padres, los cuales han vivido por y para mí y me lo han dado todo, siempre preocupados por que yo recibiera la mejor educación posible, a pesar de las diferentes adversidades por las que han pasado. Valoro muchísimo todo lo que habéis hecho –y hacéis– por mí, y aunque no os lo diga a menudo, sabéis que os quiero mucho.

Gracias también a mis colegas por haber estado siempre ahí y haberme echado una mano cada vez que la he necesitado. Tanto a los cercanos como a los lejanos.

Por último, me gustaría agradecer a todos los profesores que he tenido, tanto a los mejores como a los peores, pues de todos ellos se aprende algo. Me gustaría agradecer también de forma particular a mí tutora Teresa por haberme permito encargame de este trabajo y haberme atendido en tutoría cada vez que lo he necesitado.

Muchas gracias a todos.

Francisco Javier Ortiz Bonilla

Sevilla, 2017

Resumen

El uso de wearables (dispositivos vestibles, prendas inteligentes ó gadgetoprendas) es algo cada vez más habitual, pues proporcionan a sus usuarios una serie de funcionalidades concretas, como puede ser la monitorización de actividad (distancia recorrida, nivel de pulso cardíaco, energía gastada, etc.).

La forma tradicional que tiene un usuario de acceder a la información proporcionada por estos dispositivos es mediante una aplicación móvil que se comunica de forma inalámbrica con el dispositivo (mediante el uso de Bluetooth LE ó ANT+, por ejemplo) ó interactuando él mismo con el dispositivo de forma directa si este tiene una pantalla que le muestre la información.

Mediante este proyecto, se pretende proporcionar a los usuarios otra forma de interactuar y obtener información de pulso cardíaco y energía gastada de dispositivos que se comuniquen mediante Bluetooth LE, que permita a más de un usuario recibir información en tiempo real de un mismo dispositivo así como de varios a la vez.

Abstract

The use of wearables is currently becoming more and more common, since they provide its users with a specific number of functionalities, such as activity tracking (distance, heart rate measurement, expended energy, etc.).

The traditional way for a user to get all the information provided by these devices is through a mobile application that communicates wirelessly with the device (e.g., using Bluetooth LE or ANT+) or by directly interacting with the device itself if it has a screen that shows the information.

With this project, it is intended to provide users with another way of interacting and obtaining information about heart rate measurements and expended energy from devices that communicate via Bluetooth LE, allowing more than a user to receive real-time information from the same device as well as several at a time.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

En este primer capítulo se van a exponer los objetivos y motivación que han dado pie a este proyecto, así como la descripción de la solución adoptada. Además, al final del capítulo, se indicará cuál va a ser la estructura de la memoria.

1.1. Motivación

Los wearables están ganando cada vez más presencia, especialmente en el ámbito de la salud y el fitness. Es cada vez más habitual ver a los usuarios de un gimnasio entrenar con pulseras inteligentes que monitorizan su actividad, permitiéndoles consultar información como puede ser el pulso cardíaco, las calorías quemadas o la distancia recorrida.

Sin embargo, esta información fluye únicamente entre el propio dispositivo y su usuario: siempre con cardinalidad uno a uno. En esta situación, carecemos de una solución para que un monitor de gimnasio que está impartiendo una clase (ya sea de spinning, crossfit, zumba…) pueda monitorizar el pulso de cada usuario que reciben la clase (y que, evidentemente, lleva puesto un dispositivo que monitoriza su actividad).

Figura 1: Interacción tradicional entre un usuario y un wearable.

Además, aunque existen distintos protocolos para la comunicación inalámbrica, el más extendido y usado por estos dispositivos es Bluetooth LE (Bluetooth Smart), tecnología diseñada y comercializada por el *Grupo con especial interés en Bluetooth* (Bluetooth SIG). Adicionalmente, existe el protocolo ANT+, bastante similar a Bluetooth LE, diseñado y mantenido por Garmin[1].

Es por esto que, mediante la ejecución de este proyecto, se pretende proporcionar a los usuarios otra forma de interactuar y obtener la información de pulso cardíaco y energía gastada proporcionada por dispositivos que se comunican mediante Bluetooth LE, permitiendo a más de un usuario recibir información en tiempo real de un

[1] Garmin es una multinacional tecnológica americana especializada en en tecnología GPS y que también ha desarrollado importantes dispositivos wearables, que, como no iba a ser de otro modo, hacen uso de la tecnología ANT+ que ellos mismos desarrollan [1].

mismo dispositivo así como de varios a la vez.

De esta forma, una persona como puede ser un monitor de un gimnasio puede monitorizar el pulso de los asistentes a una clase, y al mismo tiempo cada uno de ellos, si lo desea, puede también recibir la información relativa a él.

1.2. Objetivos

Nuestros objetivos van a ser:

- En primer lugar, desarrollar una aplicación que permita recolectar la información de pulso cardíaco y energía gastada proporcionada por los distintos dispositivos BLE que se encuentren alrededor.
- En segundo lugar, hacer esta información accesible a los usuarios interesados, así como almacenarla de alguna manera para que sea posible consultarla posteriormente.
- Y, por último, realizar una aplicación que obtenga la información en tiempo real y la muestre a los usuarios.

1.3. Descripción de la solución

A continuación se muestra la arquitectura de la solución adoptada:

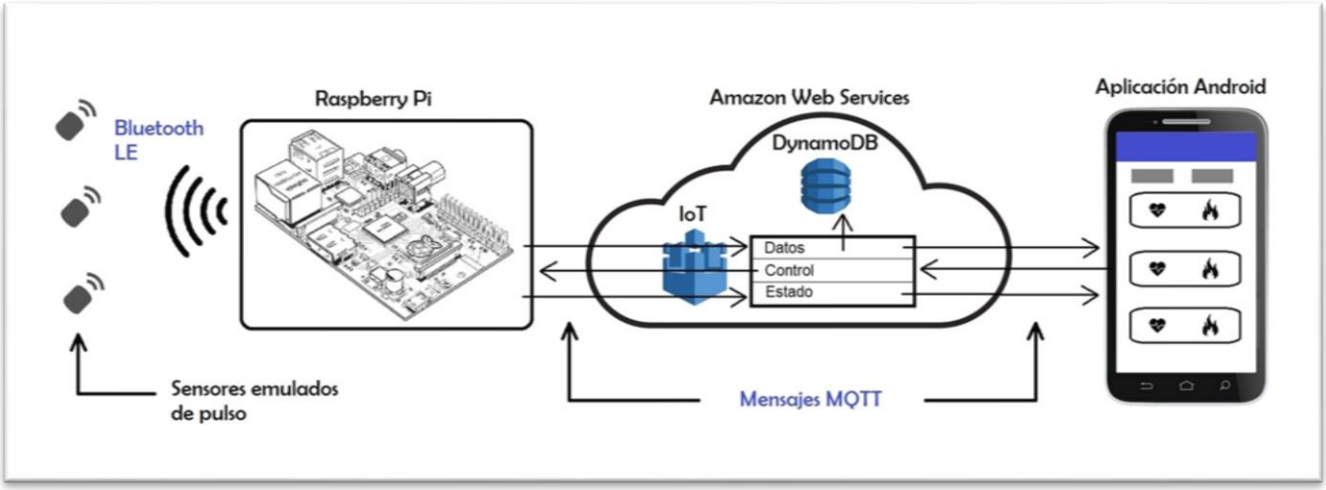

Figura 2: Arquitectura de la solución adoptada.

El funcionamiento del sistema, grosso modo, es el siguiente:

1. Haciendo uso de una Raspberry Pi que dispone de un adaptador Bluetooth 4.0 o superior conectado, se realiza la lectura de datos de pulso cardíaco y energía gastada proporcionada por los dispositivos BLE de alrededor mediante un programa desarrollado en Java.
2. Esta información se publica en un topic de la nube, y es recibida por el bróker de Amazon (en concreto, por el componente **Message broker** de la herramienta/servicio **AWS IoT**[2]) que se encarga de reenviarla a aquellos que se hubiesen suscrito al topic en cuestión. Además, al llegar los mensajes de datos, se va almacenando la información en una base de datos.
3. El usuario recibe los datos en tiempo real mediante una aplicación Android que se suscribe al topic de

[2] La nube de Amazon dispone de una serie de servicios, entre los cuales se encuentra AWS IoT. AWS IoT proporciona una comunicación segura y bidireccional entre dispositivos IoT (en nuestro caso, la Raspberry Pi) y la propia nube de Amazon. Para ello se envían y reciben mensajes MQTT (protocolo ligero para la comunicación entre máquinas (M2M) que usa el patrón publicador-suscriptor) que son procesados por un bróker (intermediario entre los publicadores y los suscriptores).

datos. A su vez, puede controlar de forma remota la Raspberry mediante la publicación de órdenes en el topic de control.

En los siguientes subapartados se va a explicar con un poco más de detalle cada una de las partes del sistema.

1.3.1. Bundle para la lectura de dispositivos BLE

Para desarrollar el programa que va a conectarse y obtener información de los dispositivos BLE, así como enviar y recibir información de la nube, se ha usado Eclipse Kura.

Eclipse Kura es un framework basado en Java y la arquitectura OSGI que nos permite transformar nuestra Raspberry en una **IoT Gateway**, esto es, un dispositivo IoT en el que vamos a poder instalar aplicaciones o plugins que reciben el nombre de **bundle,** y con una interfaz web para su configuración.

Más adelante, en el capítulo de tecnologías usadas, se va a hablar más en profundidad sobre Eclipse Kura. De momento, vamos a quedarnos con la idea de que sirve para poder instalar bundles en nuestro dispositivo Kura[3], que en este caso es la Raspberry.

La aplicación para la lectura de dispositivos BLE es, por tanto, un bundle. Este bundle se ha programado en Java, y para ello ha sido preciso usar el entorno de desarrollo Eclipse con las bibliotecas de Kura y los plugins para el desarrollo de bundles[4]. Una vez desarrollado, desde el propio Eclipse creamos un archivo de paquete que podemos instalar en nuestro dispositivo Kura.

El bundle lleva a cabo escaneos de dispositivos BLE. Conforme encuentra nuevos dispositivos que ofrecen el servicio de pulso cardíaco[5], se conecta a ellos para recibir los datos de pulso y energía gastada. Cada vez que se reciben datos nuevos, se publican en la nube.

El bundle puede ser controlado de manera remota ya que está suscrito a un topic de control, a través del cual recibirá las órdenes de comenzar a funcionar y de parar.

1.3.2. Uso de Amazon Web Services

Se hace uso de la nube de Amazon para:

1. Comunicar los dispositivos Kura (es decir, las Raspberry) con los usuarios finales que están interesados en los datos de pulso y energía gastada. Para ello, como se ha explicado anteriormente, se usa el servicio **AWS IoT**.
2. Hacer está información persistente. Para ello se usa el servicio **DynamoDB,** que proporciona una base de datos NoSQL rápida y que escala sin problemas, permitiendo en el futuro manejar grandes cantidades de datos y una alta concurrencia.

Además, en el futuro se podrá hacer uso de más servicios para dar una funcionalidad más completa al sistema y a sus usuarios. Este tema se tratará con mayor extensión en el capítulo de líneas futuras.

1.3.3. Aplicación Android

La aplicación Android se ha desarrollado pensando como usuario final en el monitor de un gimnasio que

[3] Un dispositivo Kura es un dispositivo en el que se ha instalado Eclipse Kura para transformarlo en una IoT Gateway, no confundir Eclipse Kura con el entorno de desarrollo Eclipse. En nuestro caso, el dispositivo Kura es la Raspberry.

[4] A la hora de desarrollar los bundles, se trabaja con el entorno de desarrollo Eclipse, en el que ha sido necesario instalar plugins para el desarrollo de bundles. Se programa el bundle en cualquier sitio, y luego se instala en el dispositivo Kura.

[5] Los dispositivos Bluetooth LE ofrecen servicios. En este proyecto el servicio con el que vamos a trabajar es el de pulso cardíaco (Heart Rate Service). Hay una serie de servicios estandarizados por Bluetooth, y este es uno de ellos. Más adelante, en el capítulo de tecnologías usadas, se va a hablar más en profundidad sobre Bluetooth Low Energy.

imparte una clase de deporte (ya sea spinning, crossfit, zumba…).

La idea es que en la sala donde se lleva a cabo la clase, se encuentra una Raspberry con Kura instalado y con el bundle visto anteriormente instalado y configurado. La aplicación se conecta a la nube de Amazon, y puede recibir los datos que el bundle está publicando (suscribiéndose al topic donde se publican), así como activarlo o desactivarlo (publicando órdenes en el topic de control al que está suscrito el bundle).

Los datos que se muestan son las medidas de pulso y energía gastada que se van recibiendo de cada uno de los dispositivos BLE (identificados por su dirección MAC) y para cada uno de ellos se puede consultar una gráfica con el histórico de medidas recibidas durante la clase.

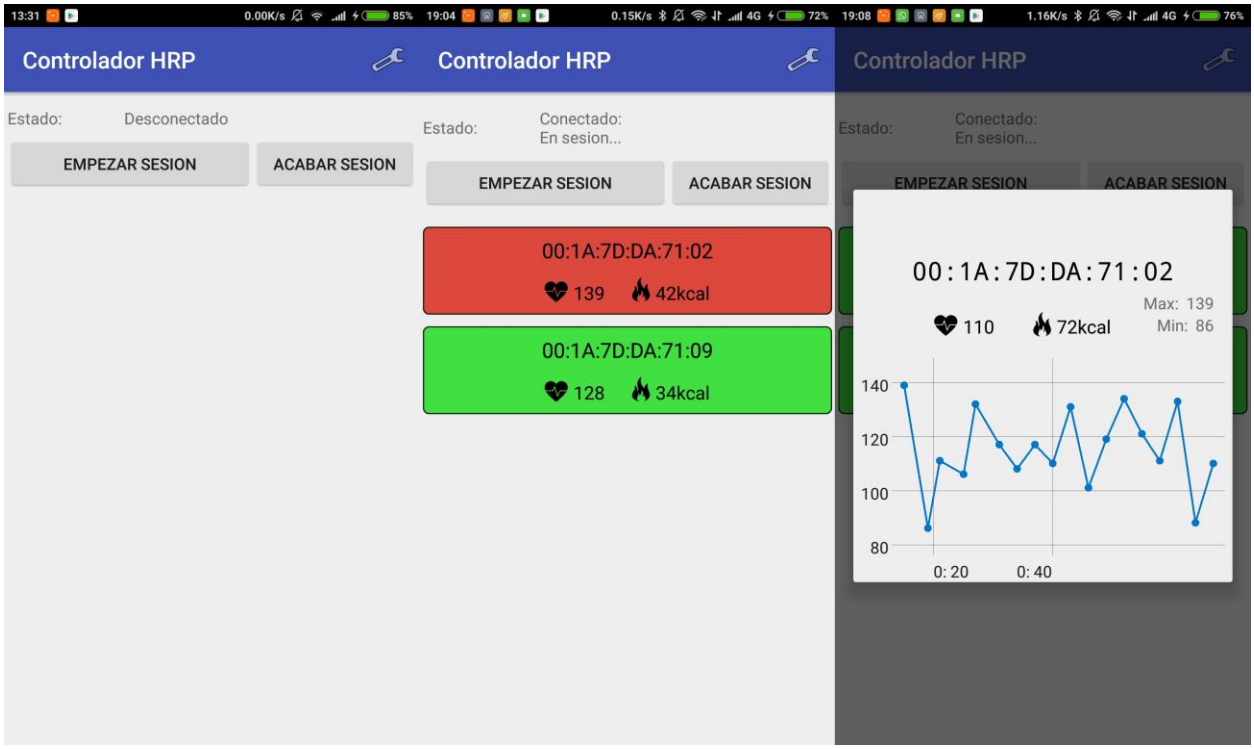

Figura 3: Presentación de los datos en la aplicación Android.

1.3.4. Sensores emulados de pulso

Para los sensores emulados de pulso, se ha usado un módulo de Node.js llamado **bleno**, que permite implementar periféricos Bluetooth LE.

Los sensores emulados de pulso son, pues, un programa escrito en Node.js. Al ejecutarse, toma una interfaz Bluetooth LE (adaptador Bluetooth 4.0 ó superior conectado a la máquina) y la convierte en un periférico Bluetooth LE, que en nuestro caso se ha programado para que ofrezca un servicio de pulso cardíaco (siguiendo la especificación de este servicio proporcionada por el Bluetooth SIG) y que proporcione valores aleatorios dentro de un rango conocido.

Estos sensores se ejecutan en un portátil desde una terminal Linux y se necesita de un adaptador Bluetooth 4.0 o superior para cada sensor emulado que se quiera poner en funcionamiento.

1.4. Estructura de la memoria

La memoria de este trabajo se va a organizar de la siguiente manera:

- **Capítulo 1: Introducción.**

  En este capítulo se ha hablado sobre los objetivos y motivación que han dado pie al proyecto, y se ha presentado la solución adoptada sin entrar en muchos detalles.

- **Capítulo 2: Tecnologías utilizadas.**

  En este capítulo se va a hablar sobre las distintas tecnologías utilizadas, explicando con más profundidad aquellas en las que esto se considere necesario, como puede ocurrir con Bluetooth LE, Eclipse Kura y los servicios de AWS utilizados: AWS IoT, DynamoDB y Amazon Cognito.

- **Capítulo 3: Arquitectura y análisis del sistema.**

  En este capítulo se va a explicar con mayor detalle el funcionamiento del sistema completo y como interaccionan las distintas partes entre sí.

- **Capítulo 4: Sensores emulados de pulso.**

  En este capítulo se va a explicar el funcionamiento y el desarrollo de los sensores emulados de pulso usados en este trabajo.

- **Capítulo 5: Bundle HRPClient.**

  En este capítulo se va a explicar el funcionamiento del bundle programado para la lectura de la información relativa al pulso cardíaco y la energía gastada proporcionada por los dispositivos BLE.

- **Capítulo 6: Aplicación Android.**

  En este capítulo se va a explicar el diseño y funcionamiento de la aplicación Android realizada para controlar el bundle y para recibir y mostrar la información de pulso cardíaco y energía gastada.

- **Capítulo 7: Conclusiones y líneas futuras.**

  En este capítulo se van a exponer las líneas para la continuación de este proyecto y las conclusiones a las que he llegado tras haber realizado este trabajo.

Anexos:

- **Anexo A: Manual de instalación y uso de la pila Bluetooth para Linux, BlueZ.**

  En este anexo se va a exponer el proceso de instalación de BlueZ, la pila Bluetooth para Linux, y se van a ver sus distintas herramientas y posibilidades.

- **Anexo B: Manual de instalación y configuración de Eclipse Kura en una Raspberry Pi.**

  En este anexo se va a ver cómo instalar Eclipse Kura en una Raspberry Pi y cómo configurarla usando la interfaz web que proporciona Kura.

2 TECNOLOGÍAS UTILIZADAS

En este capítulo se va a hablar sobre las distintas tecnologías utilizadas para la realización de este trabajo, explicando con más profundidad aquellas en las que esto se considere necesario.

2.1 Bluetooth Low Energy

Bluetooth es una especificación tecnológica que permite la interconexión inalámbrica entre dispositivos y el intercambio de información entre estos haciendo uso de la banda ISM de 2.4 GHz. Las especificaciones de las distintas versiones de Bluetooth son realizadas por el grupo con especial interés en Bluetooth (Bluetooth SIG).

Figura 4: Logo de Bluetooth SIG.

Se denomina Bluetooth Low Energy (**Bluetooth LE**, **BLE**, y comercializado como **Bluetooth Smart**) a la implementación adoptada a partir de la versión 4.0 de Bluetooth, en Junio de 2010. Esto supuso el nacimiento de una nueva forma de Bluetooth distinta a la anterior, que denominaremos Bluetooth clásico. De esta forma, conviven los dos tipos de tecnología Bluetooth, siendo usada una u otra según el ámbito en el que se necesite.

2.1.1 Diferencia con el Bluetooth clásico

Bluetooth clásico

Se establece una conexión inalámbrica de corto rango para el envío de datos continuo que lo hace ideal para ser usado en casos como el streaming de audio.

Bluetooth LE

Se establece una conexión inalámbrica de rango más largo, pero no para el envío de datos continuo, con un consumo de energía mucho más reducido, haciéndolo esto ideal para aplicaciones IoT que no requieren envío continuo y necesitan de una larga vida de batería [2].

De manera adicional, existe la posibilidad de incluir información en los mensajes de beacon (esto es, los mensajes que difunde un dispositivo Bluetooth cuando se anuncia) haciendo esta información visible a cualquier otro dispositivo Bluetooth que esté dentro de su rango y sin necesidad de establecer una conexión. Un ejemplo de esto es **iBeacon**, un sistema de posicionamiento de interiores desarrollado por Apple.

2.1.2 GAP

GAP son las siglas de Generic Access Profile, y es lo que controla las conexiones y el anuncio en Bluetooth LE. Podemos decir que es lo que hace a nuestro dispositivo visible hacia el exterior, y determina cómo un par de dispositivos pueden (o no) interaccionar el uno con el otro.

Roles

GAP define fundamentalmente dos roles para los dispositivos:

- **Periféricos**: Podemos definir los dispositivos periféricos como aquellos que son fuente de datos. Por ejemplo, sensores de temperatura, sensores de luz, dispositivos wearables para el seguimiento de la actividad…
- **Central**: Aquellos dispositivos que se conectan a los periféricos para obtener información de ellos.

Mecanismo para anunciarse

Un periférico que se anuncia transmite un paquete con payload de tipo *Advertising Data* de forma periódica (cada periférico tiene un intervalo de anuncio específico, y cada vez que pasa este intervalo, se retransmite el paquete de beacon).

De manera opcional, existe un segundo paquete con payload de tipo *Scan Response Data* que puede ser transmitido en caso de que un dispositivo central lo pida, y de esta manera se permite a los diseñadores de los periféricos proporcionar un poco más de información durante el anuncio.

Ambos payloads pueden contener hasta 31 bytes de datos. En la siguiente figura se puede observar el comportamiento que se acaba de explicar.

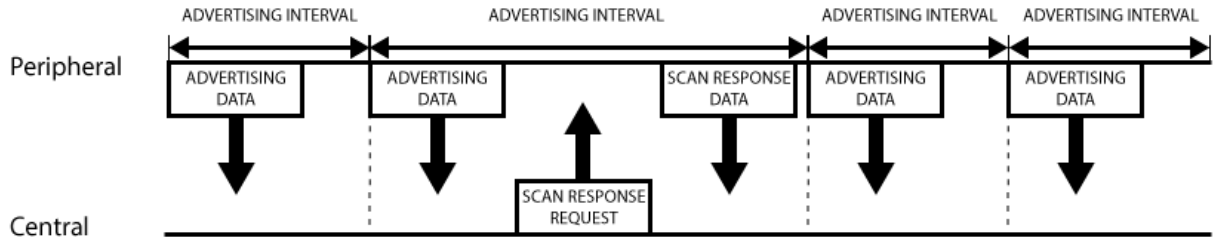

Figura 5: Paquetes transmitidos durante el anuncio de un periférico.

Además, como se comentó anteriormente, puede usarse el payload del mensaje de anuncio para incluir información que quiere difundirse, de esta manera se puede enviar esta información a todos los dispositivos que estén escuchando dentro de nuestro rango.

Es importante mencionar que un dispositivo BLE al aceptar una conexión deja de anunciarse, y no podrán enviarse los mensajes de anuncio. Es decir, si un periférico BLE se encuentra en una conexión con otro dispositivo, no se puede anunciar [3].

2.1.3 GATT

GATT es un acrónimo de Generic Attribute Profile, y define la manera en la que dos dispositivos BLE intercambian datos, usando lo que se denominan Servicios y Características. A su vez, hace uso de un protocolo genérico de datos llamado Attribute Protocol (ATT) y que sirve para almacenar Servicios.

Una vez que se ha establecido una conexión entre un periférico y un dispositivo central, se usarán los servicios y características GATT para la comunicación en ambas direcciones.

GATT se basa en objetos anidados: Perfiles, servicios y características [4].

Perfiles

Un perfil se compone de un conjunto de servicios. Aunque en los periféricos BLE no existen los perfiles –sino sólo una colección de servicios y sus características que han sido compilados por los diseñadores del periférico–, los perfiles son útiles para especificar qué servicios son necesarios implementar para poder proporcionar una funcionalidad concreta.

De este modo, existen una serie de perfiles que han sido especificados por el Bluetooth SIG. Estos pueden consultarse aquí: https://www.bluetooth.com/specifications/adopted-specifications

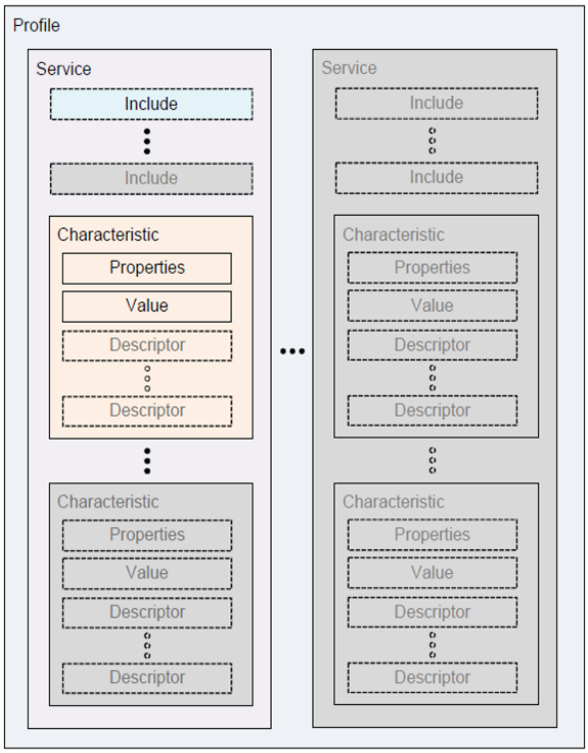

Figura 6: Jerarquía ATT.

Servicios

Los servicios son unidades independientes dentro de un perfil, y se usan para agrupar en entidades lógicas datos relacionados entre sí, y que reciben el nombre de características. Un servicio se diferencia de otro por su UUID, un identificador universal que puede ser de 16 bits para servicios adoptados oficialmente por el Bluetooth SIG ó de 128 bits si es un servicio personalizado.

La lista de los servicios adoptados por el Bluetooth SIG puede consultarse aquí:
https://www.bluetooth.com/specifications/gatt/services

Características

Cada característica contiene el valor de un dato concreto (ya sea un byte, una cadena de caracteres o un array como podrían ser los valores X,Y,Z proporcionados por un acelerómetro). Además de para leer el valor de una característica y obtener así información del periférico en un dispositivo central, pueden usarse para enviar información del dispositivo central al periférico, ya que también puede escribirse en una característica.

Las características poseen propiedades que definen qué puede hacer otro dispositivo BLE con ellas, estas son:

- READ: Si una característica tiene la propiedad READ, entonces otro dispositivo BLE puede leer su valor.
- WRITE: Si una característica tiene la propiedad WRITE, entonces otro dispositivo BLE puede

escribir su valor.

- NOTIFY: Si una característica tiene la propiedad NOTIFIY, entonces otro dispositivo BLE puede activar las notificaciones para esta característica, así cada vez que el periférico disponga de un nuevo valor actualizado, enviará al dispositivo central este nuevo valor.

Al igual que los servicios, cada característica posee un UUID que la diferencia del resto, y puede ser de 16 o 128 bits según si es una característica adoptada oficialmente por el Bluetooth SIG o bien es una característica personalizada.

La lista de las características adoptadas por el Bluetooth SIG puede consultarse aquí:
https://www.bluetooth.com/specifications/gatt/characteristics

2.2 Eclipse Kura

Eclipse Kura es un proyecto de Eclipse IoT[6] que proporciona una plataforma para construir lo que se denominan '**IoT gateways**' (pasarela para internet de las cosas). Es un contenedor de aplicaciones que permite la gestión remota de estos gateways y además proporciona una serie de APIs útiles para el desarrollo y despliegue de aplicaciones IoT.

Figura 7: Logo de Eclipse Kura.

Un dispositivo 'IoT gateway' se encarga de la comunicación entre dispositivos IoT, sensores, equipos y la nube. Pueden conectarse de forma sistemática a la nube y ofrecen soluciones para el procesamiento y almacenamiento local de datos.

Eclipse Kura proporciona un contenedor para aplicaciones que corren en un dispositivo IoT basado en Java y la tecnología OSGi[7] [5]. Estas aplicaciones van a recibir el nombre de **bundles**.

Vamos a diferenciar entonces dos aspectos de Eclipse Kura:

1. Instalación de Eclipse Kura en un dispositivo para convertirlo en una pasarela IoT. A partir de ahora, a los dispositivos en los que se ha instalado Eclipse Kura para convertirlos en una pasarela IoT vamos a llamarlos '**dispositivos Kura**'.
2. Desarrollo de bundles para ser instalados en un dispositivo Kura usando el entorno de desarrollo Eclipse.

2.2.1 OSGi

OSGi son las siglas de Open Services Gateway initiative. La especificación OSGi describe un sistema modular y una plataforma de servicios para el lenguaje de programación Java, que implementa un modelo de

[6] https://iot.eclipse.org/

[7] https://www.osgi.org/

componentes completo y dinámico, lo cual no existe en entornos autónomos de Java [6].

OSGi comenzó a desarrollarse en 1999, cuando los vendedores de sistemas embebidos y los proveedores de red se unieron para crear un conjunto de estándares para un marco de servicios basado en Java y que pudiera gestionarse de manera remota. Al principio se concibió como una pasarela para controlar pequeños aparatos y dispositivos del hogar con acceso a Internet, y consistía de un framework Java embebido en alguna plataforma hardware como podía ser la de un módem de cable. La pasarela actuaba entonces como el bróker central de los mensajes enviados por los dispositivos en la red local del hogar [7].

Fundamentalmente, OSGi tiene dos partes:

1. La especificación de los componentes modulares, llamados **bundles**, y que son referidos comúnmente también como 'plugins'.
2. Una máquina virtual Java (JVM) que los bundles usan para publicar, descubrir y enlazarse a servicios en una arquitectura orientada a servicios (SOA).

2.2.1.1 Bundles

Un bundle es, grosso modo, un componente JAR[8] con un manifiesto. Es un conjunto de clases Java y otros recursos adicionales acompañados de un archivo manifiesto, `MANIFEST.MF`, en el que se describe su contenido y características, así como los servicios que usa, los servicios exportados y los permisos necesarios para poder ser utilizados por otros componentes [8].

En un contenedor de bundles, como puede ser un dispositivo Kura, un bundle puede encontrarse en los siguientes estados [9]:

- INSTALLED: El bundle se ha instalado correctamente.
- RESOLVED: El bundle está instalado y todas sus clases y dependencias están disponibles, pero no está activo.
- STARTING: El bundle se está activando. Para activar un bundle, se llama al método que se haya indicado como método para activarlo. Tras activarse pasará al estado ACTIVE.
- ACTIVE: El bundle se ha activado correctamente y está funcionando.
- STOPPING: El bundle se está desactivando. Para parar un bundle, se llama al método que se haya indicado como método para pararlo. Tras desactivarse pasará al estado RESOLVED.
- UNINSTALLED: El bundle se ha desinstalado correctamente.

Como hemos visto, los bundles se activan y desactivan llamando a sus respectivos métodos de activación y desactivación, por lo que **no tienen un método main**.

2.2.1.2 Servicios

Los servicios son especificados por interfaces Java, y un bundle puede implementar la interfaz de un servicio para posteriormente exportarlo y registrarlo en el registro de servicios. De este modo, un bundle puede hacer uso de un servicio ofrecido por otro bundle.

En el siguiente apartado se van a ver un par de servicios implementados por componentes de Eclipse Kura que se instalan por defecto al instalar Eclipse Kura en un dispositivo.

[8] JAR: **J**ava **AR**chive. Es un formato de archivos de empaquetado. Se usa para agregar clases Java y recursos asociados.

2.2.2 Dispositivo Kura

Como se dijo anteriormente, un dispositivo Kura es aquel en el que se ha instalado Eclipse Kura para convertirlo en una pasarela IoT.

La instalación de Eclipse Kura en una Raspberry Pi es sencilla y está explicada de manera detallada en el Anexo B: Manual de instalación y configuración de Eclipse Kura en una Raspberry Pi.

Una vez que se ha instalado Eclipse Kura en nuestra Raspberry, podemos acceder a través de un navegador (si tenemos la Raspberry accesible a nivel de red, claro) a la interfaz web.

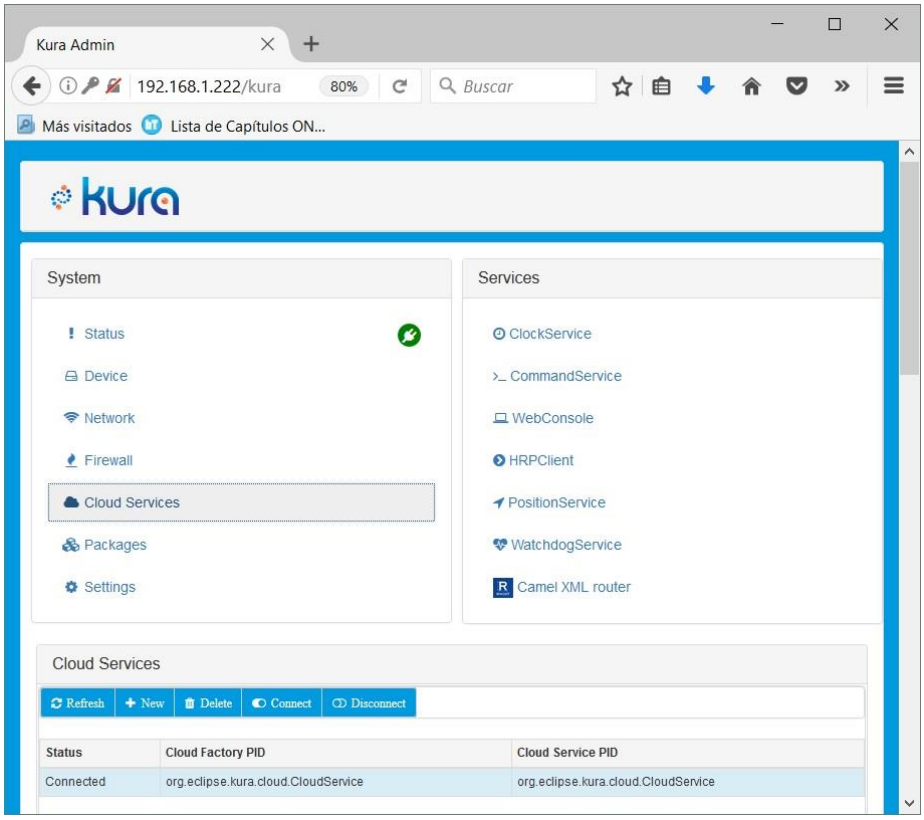

Figura 8: Interfaz web de Eclipse Kura.

Como podemos ver en la figura 8, Eclipse Kura viene con una serie de servicios preparados y que podemos usar. Nuestro bundle (HRPClient, podemos ver que aparece también en la interfaz web[9]), en realidad, hace uso de dos de los servicios proporcionados por Kura: Un servicio para el uso de Bluetooth (`org.eclipse.kura.bluetooth.BluetoothService`) y otro para el uso de la nube (`org.eclipse.kura.data.DataService`).

2.2.2.1 Kura BluetoothService

Eclipse Kura dispone de una librería para manejar la pila Bluetooth de Linux y que nosotros vamos a usar en nuestro bundle para poder escanear dispositivos BLE, conectarnos e interactuar con ellos.

Hay que decir que de manera interna, las clases de Eclipse Kura para el manejo de Bluetooth están haciendo a

[9] HRPClient es el bundle que se ha desarrollado para interactuar con los dispositivos BLE y enviar los datos a la nube. El nombre viene de Heart Rate Peripheal Client, es decir, cliente de periféricos de pulso cardíaco.

su vez uso, mediante JNI[10], de `gatttool`, una herramienta de BlueZ[11] que nos permite conectarnos a dispositivos BLE e interactuar con ellos. Es por esto que si queremos hacer uso de estas clases, será necesario tener instalado BlueZ en el dispositivo donde se vayan a ejecutar.

2.2.2.2 Kura DataService

Eclipse Kura también incluye un componente ya preparado que implementa la interfaz del servicio DataService. Mediante el uso de este servicio, podemos conectarnos a la nube y enviar mensajes a topics y recibir mensajes de los topics a los que nos hayamos suscrito.

La configuración de este componente que implementa el servicio DataService se hace desde la interfaz web. Como puede verse en la Figura 8: Interfaz web de Eclipse Kura., el servicio aparece en la pestaña Cloud Services, y ahí nos aparecen una serie de parámetros a configurar como pueden ser el punto final (la dirección de nuestra nube), el id del cliente con el que se conecta a la nube, el certificado a usar… El tema de la configuración de la nube en Eclipse Kura se va a tratar con más profundidad en el Anexo B: Manual de instalación y configuración de Eclipse Kura en una Raspberry Pi.

Aparte del servicio DataService, existe otro llamado CloudService, y ambos sirven para lo mismo. Sin embargo, el CloudService no te permite trabajar directamente con los topics, si no que los forma a partir de unos parámetros configurados en la interfaz web.

2.2.3 Desarrollo de bundles con Eclipse

Para desarrollar y crear bundles que puedan ser desplegados en dispositivos Kura, necesitaremos el entorno de desarrollo Eclipse y el espacio de trabajo de Kura, que contiene las APIs de Kura y una serie de bundles necesarios que son dependencias de Kura. Por supuesto también es necesario tener instalada una máquina virtual de Java.

Una vez tenemos listo Eclipse, podemos comenzar a desarrollar bundles para nuestros dispositivos Kura. Además, el workspace de Kura permite emular un dispositivo Kura y poder probar en él los bundles que desarrollemos.

2.3 La nube: Amazon Web Services

El tema del cloud computing hace ya tiempo que dejó de ser algo del futuro para convertirse en el propio presente. No es nada nuevo o desconocido, por lo que no vamos a extendernos mucho en este apartado.

Según la definición de la Real Academia de la Ingeniería, la computación en la nube es la *utilización de las instalaciones propias de un servidor web albergadas por un proveedor de Internet para almacenar, desplegar y ejecutar aplicaciones a petición de los usuarios demandantes de las mismas* [10].

Es decir, mediante el uso de la nube podemos disponer de recursos computacionales sin necesidad de tener que manipular los dispositivos físicos en los que se encuentran ni tener que preocuparnos por su gestión o despliegue.

En este trabajo hemos hecho uso de la nube de Amazon, que ofrece una larga serie de servicios que dan solución prácticamente a cualquier problema que se nos pueda presentar. Para nuestro sistema hemos utilizado tres servicios: **AWS IoT**, **DynamoDB** y **Amazon Cognito**. No obstante, nuestro sistema puede aún mejorarse e incorporar más funcionalidades, y para ello sin duda se podría hacer uso de otros servicios de la nube de Amazon muy interesantes. Este tema se tratará más a fondo en el capítulo de líneas futuras.

[10] Java Native Interface: Sirve para poder interactuar desde un programa escrito en Java y que corre en la máquina virtual de Java con programas escritos en otros lenguajes como C ó C++.

[11] BlueZ es una pila de Bluetooth para Linux. Se hablará de ella y se verá como instalarla en el Anexo A: Manual de instalación y uso de la pila Bluetooth para Linux, BlueZ.

2.3.1 AWS IoT

AWS IoT proporciona una comunicación segura y bidireccional entre dispositivos IoT (en nuestro caso, la Raspberry Pi) y la propia nube de Amazon. Para ello se envían y reciben mensajes MQTT (protocolo ligero para la comunicación entre máquinas (M2M) que usa el patrón publicador-suscriptor) que son procesados por un bróker (intermediario entre los publicadores y los suscriptores) [11].

AWS IoT consiste de los siguientes componentes:

- **Device gateway (Pasarela para dispositivos)**
  Permite a dispositivos comunicarse de manera segura y eficiente con AWS IoT.

- **Things[12] registry (Registro de 'cosas')**
  Organiza los recursos asociados con cada 'thing'. A ellas se pueden asociar atributos, certificados e id de clientes MQTT.

- **Things shadow (Sombra de las 'things')**
  La sombra de una 'thing' es un documento JSON que contiene información acerca del estado de la 'thing'.

- **Things Shadows service (Servicio para las sombras de las 'things')**
  Proporciona representaciones persistentes de los estados de nuestras 'things'. De este modo, se puede publicar información actualizada del estado de una 'thing', y esta cuando se conecte a la nube puede sincronizar su estado.

- **Message broker (Bróker para los mensajes)**
  Proporciona un mecanismo seguro para que las 'things' y las aplicaciones AWS IoT puedan publicar y recibir mensajes entre ellas.

- **Rules engine (Motor de reglas)**
  Proporciona un mecanismo para procesar mensajes e integrar AWS IoT con otros servicios de la plataforma AWS. Un ejemplo es que cada mensaje de datos que llegue haga saltar una regla que haga que determinada información del mensaje se almacene en una tabla de la base de datos DynamoDB.

- **Security and Identity service (Seguridad y servicio de identidad)**
  Proporciona una responsabilidad compartida para la seguridad en la nube de Amazon. Es decir, nuestras 'things' deben mantener sus credenciales seguras para que el envío de datos al bróker de los mensajes se haga de forma segura.

En nuestro sistema se hace uso del **bróker para los mensajes** para intercambiar mensajes de datos y control entre los dispositivos Kura (las Raspberry Pi) y los usuarios (haciendo uso de la aplicación Android), y del **motor de reglas** para almacenar en DynamoDB información de cada mensaje de datos que llegue.

Además, nuestro sistema está pensado para que pueda ser usado por multitud de gimnasios, por lo que cada uno se refleja como una 'thing' en la nube de Amazon, con su certificado y política correspondiente. Cada política permite a un gimnasio concreto publicar y suscribirse en sus topics, pero no en los de otros.

2.3.2 DynamoDB

DynamoDB es una base de datos NoSQL que proporciona un rendimiento rápido y predecible y que escala sin problemas. Se puede usar para crear una tabla para almacenar y devolver cualquier cantidad de datos y servir

[12] En AWS IoT, una 'thing' (cosa) es algo que identifica a una entidad (comúnmente un dispositivo) que según qué caso representará a una cosa diferente. Por ejemplo, en el sistema que hemos desarrollado, una 'thing' en nuestra nube representa a un gimnasio, pero dentro de cada gimnasio existen varios dispositivos (Raspberry Pi), por lo que en este escenario, una 'thing' no representa a un dispositivo en concreto, sino a un gimnasio particular.

cualquier nivel de tráfico de solicitudes [12].

Se ha elegido DynamoDB frente a una base de datos relacional tradicional por la facilidad con la que escala y porque, según Amazon, puede manejar una altísima concurrencia en la escritura de datos. Estos son requisitos fundamentales para nuestro sistema, pues en un futuro podría hacerse muy popular y existir muchísimos gimnasios con varios dispositivos enviando a la vez información que debe ser almacenada en nuestra base de datos.

En DynamoDB, una tabla contiene una serie de atributos, al igual que las tablas de las bases de datos tradicionales, sin embargo, es obligatorio especificar para la tabla una clave primaria[13]. La clave primaria puede ser de dos tipos:

- **Partition key**: Una clave primaria simple compuesta por un único atributo, llamado **partition key**.
- **Partition key and sort key**: Una clave primaria compuesta por un par de atributos, el atributo **partition key** y el atributo **sort key**.

2.3.3 Amazon Cognito

Amazon Cognito es un servicio que nos permite crear identidades para los usuarios de nuestras aplicaciones y autenticarlos [13].

Nuestra aplicación Android, para poder publicar y recibir datos de AWS IoT necesitaría de una identidad (certificado y clave privada). La aplicación hace uso de Amazon Cognito para crear de forma local y automática una identidad y la usa para autenticarse con AWS IoT. A esta identidad se le asocia siempre una política que le permite publicar y suscribirse a los topics de los demás gimnasios.

2.4 Android

Se ha elegido Android para crear la aplicación que usaría un monitor de gimnasio para recibir los datos de pulso y energía gastada de sus alumnos por ser este el sistema operativo más extendido entre smartphones (y tablets) con diferencia, como podemos observar en la figura 9 [14].

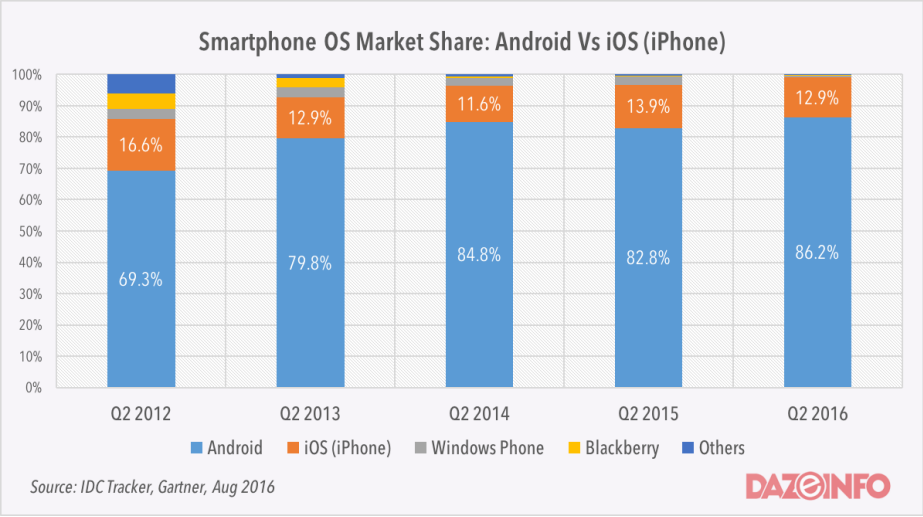

Figura 9: Porcentaje de smartphones según sistema operativo

Para desarrollar la aplicación se ha hecho uso del entorno de desarrollo Android Studio, y ha sido necesario incluir como dependencias, entre otras, el sdk de AWS para IoT.

[13] Una clave primaria en una tabla permite identificar de manera única a un elemento de esta.

3 ARQUITECTURA Y ANÁLISIS DEL SISTEMA

En este capítulo se va a explicar la arquitectura del sistema. Se va a ver la funcionalidad que cumple cada componente y cómo estos interaccionan los unos con los otros.

3.1 Arquitectura del sistema

Tal y como vimos en la introducción, la arquitectura del sistema es la siguiente:

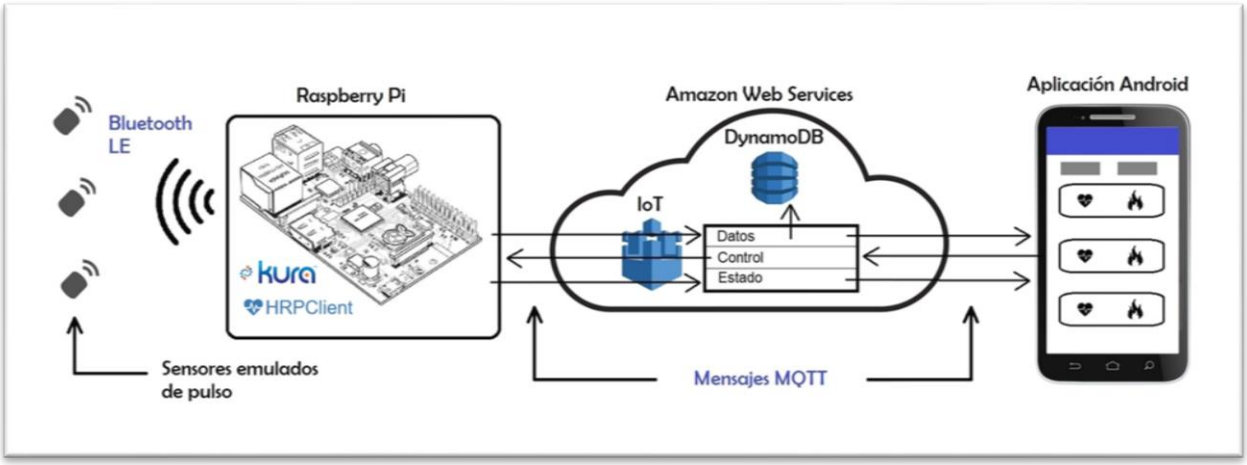

Figura 10: Arquitectura del sistema.

El sistema presenta tres entidades diferenciadas, que son:

1. Una Raspberry Pi con Eclipse Kura instalado y nuestro **bundle HRPClient** activo, que se encarga de obtener información de pulso y energía de los dispositivos BLE y de publicarla en la nube.

2. La **nube**. Se usa Amazon Web Services, y para gestionar la información se utilizan los servicios AWS IoT y DynamoDB.

3. Una **aplicación Android**, que se conecta a la nube y recibe a través de esta la información en tiempo real.

3.1.1 Bundle HRPClient: Lectura de dispositivos BLE y envío de la información a la nube

Por un lado tenemos los sensores de pulso emulado, que van a simular ser dispositivos que miden el pulso y la energía gastada de un usuario. Los datos que proporcionan son accesibles vía Bluetooth LE del mismo modo que si fueran dispositivos reales, pues implementan el servicio de pulso cardíaco tal y como se describe en la especificación dada por el Bluetooth SIG.

Luego tenemos la Raspberry Pi con Eclipse Kura instalado, en la que corre nuestro bundle **HRPClient**. La idea es que en un mismo gimnasio puede haber distintas salas en las que se dan clases de deporte, y en cada

una de ellas hay una Raspberry Pi. De este modo, cada monitor puede hacer uso de la Raspberry Pi que se encuentra en su sala para obtener la información de pulso y energía gastada de sus alumnos.

El bundle **HRPClient** se activa al encenderse la Raspberry y se mantiene siempre activo, pero en estado idle si no se está usando[14]. Su funcionamiento se controla enviándole órdenes a través de la nube, ya que en principio sólo nos interesa que reciba datos de los dispositivos BLE cuando se está dando una clase.

Como se verá más adelante, la característica 'medida de pulso' (**Heart Rate Measurement**) del servicio de pulso cardíaco es notificable, es decir, el dispositivo central que esté conectado al periférico BLE activa las notificaciones para esta característica, y cada vez que el periférico disponga de un nuevo valor para la característica, notificará al dispositivo central con este nuevo valor.

Entonces, al recibir el bundle **HRPClient** de una Raspberry Pi la orden de comenzar a funcionar (cada vez que comience a funcionar diremos que ha iniciado una nueva sesión, y cuando se reciba la orden de parar, se da la sesión por finalizada), este comienza a realizar escaneos Bluetooth LE de forma periódica para descubrir los dispositivos BLE de alrededor, se conecta a ellos y activa las notificaciones para la característica 'medida de pulso'. Cada vez que es notificado con un nuevo valor de pulso, envía a la nube esta información.

Para que el bundle pueda enviar la información a la nube, ha debido configurarse previamente en la Raspberry Pi, mediante la interfaz web de Kura, los distintos parámetros del CloudService para la conexión de la Raspberry Pi con nuestra nube (nuestro endpoint, el id de cliente, el certificado, etc.). Cómo realizar esta configuración se explica en el Anexo B: Manual de instalación y configuración de Eclipse Kura en una Raspberry Pi.

3.1.2 La nube

Nuestro sistema está pensado para que pueda ser usado por múltiples gimnasios que disponen de diferentes dispositivos (Raspberry Pi) que pueden estar funcionando a la vez. Es por esto que es de vital importancia separar la información de cada gimnasio, incluso de los diferentes dispositivos (Raspberry Pi) dentro de un gimnasio, de manera que no se mezcle. Del mismo modo, es necesario que cada gimnasio tenga una identidad y que esto le permita manejar su información de manera segura, es decir, que nadie que no sea el propio gimnasio pueda enviar información relativa a él.

Otra característica que queremos para nuestro sistema es que la información se almacene de alguna manera para poder ser consultada posteriormente.

3.1.2.1 Identidad de un gimnasio

En nuestro sistema, cada gimnasio se va a identificar mediante un código.

Además, cada gimnasio posee una 'thing' en el registro de 'cosas' (**Thing registry**) de AWS IoT. Para esa 'thing' se crea y activa un certificado y se obtiene su clave privada. Mediante este certificado y esta clave

[14] Es importante entender esto. El bundle siempre está activo, pero sin hacer nada, en estado idle, a la espera de recibir órdenes. Si recibe la orden de empezar a funcionar, entonces comenzará a realizar escaneos Bluetooth LE para encontrar dispositivos de pulso cardíaco, se conectará a ellos y conforme reciba información de pulso y energía gastada, la enviará a la nube. Estará funcionando hasta que reciba la orden de parar. Podríamos decir que el bundle se comporta como una televisión que está enchufada a la corriente y se enciende o apaga (¡pero se mantiene en stand by!).

privada, el gimnasio obtiene su identidad, necesaria para que cualquiera de sus dispositivos (Raspberry Pi) pueda conectarse a nuestra nube en su nombre[15].

Luego, al certificado del gimnasio se asocia una política[16] personalizada para él, que permite al gimnasio conectarse a nuestra nube usando una serie de ids de cliente (de esta forma limitamos el número de dispositivos de un gimnasio que pueden conectarse a la nube) y publicar y suscribirse **sólo a sus topics.**

A continuación, las figuras 11 y 12 muestran un par de capturas en las que se pueden ver las distintas 'things' y políticas que he ido creando en mi nube para hacer pruebas.

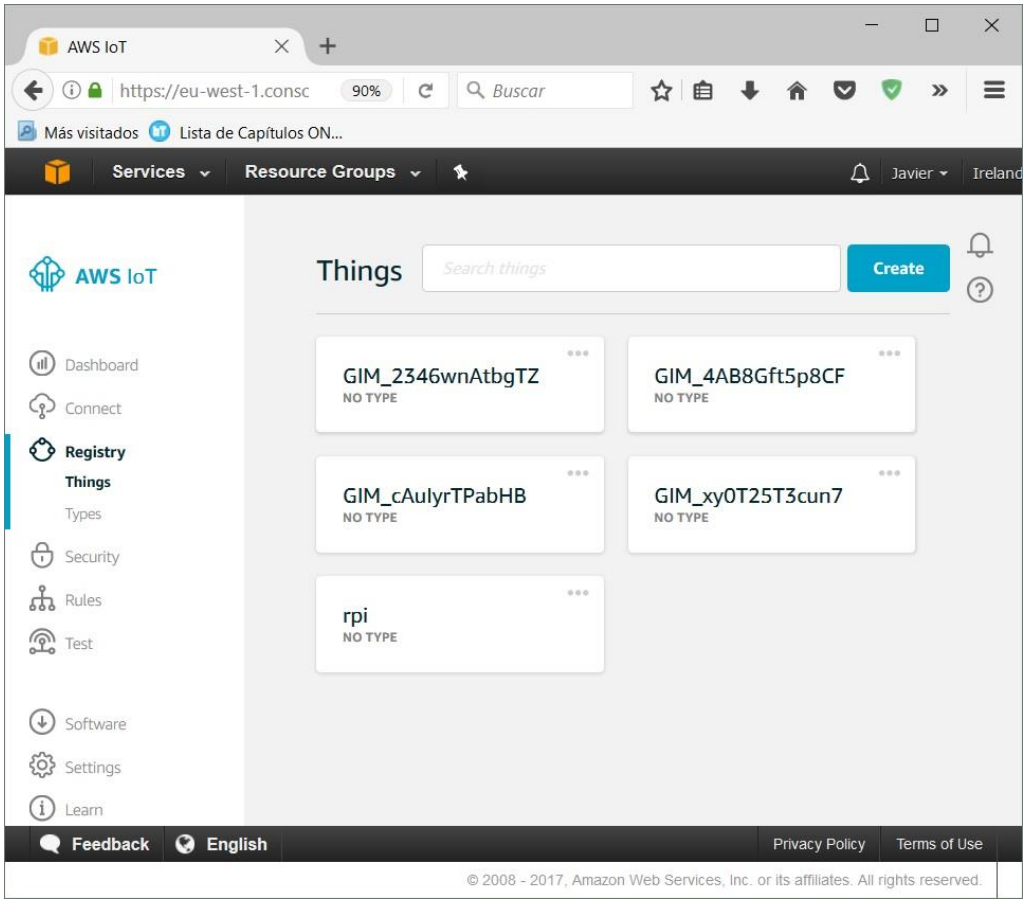

Figura 11: Things en AWS IoT.

[15] Se pueden conectar múltiples dispositivos a la vez usando la misma identidad sin ningún problema. Sin embargo, es preciso que cada uno lo haga con un id de cliente distinto, ya que en una misma nube de Amazon (un endpoint) no pueden existir dos clientes con el mismo id de cliente. Es decir, cualquier dispositivo que se conecte a nuestra nube debe hacerlo con un id de cliente único, si no, en caso de existir un dispositivo a la nube con el mismo id, hará que se desconecte.

[16] En AWS IoT, las políticas definen un conjunto de reglas que definen qué puede y qué no puede hacer 'algo' (un certificado) a lo que se haya vinculado la política.

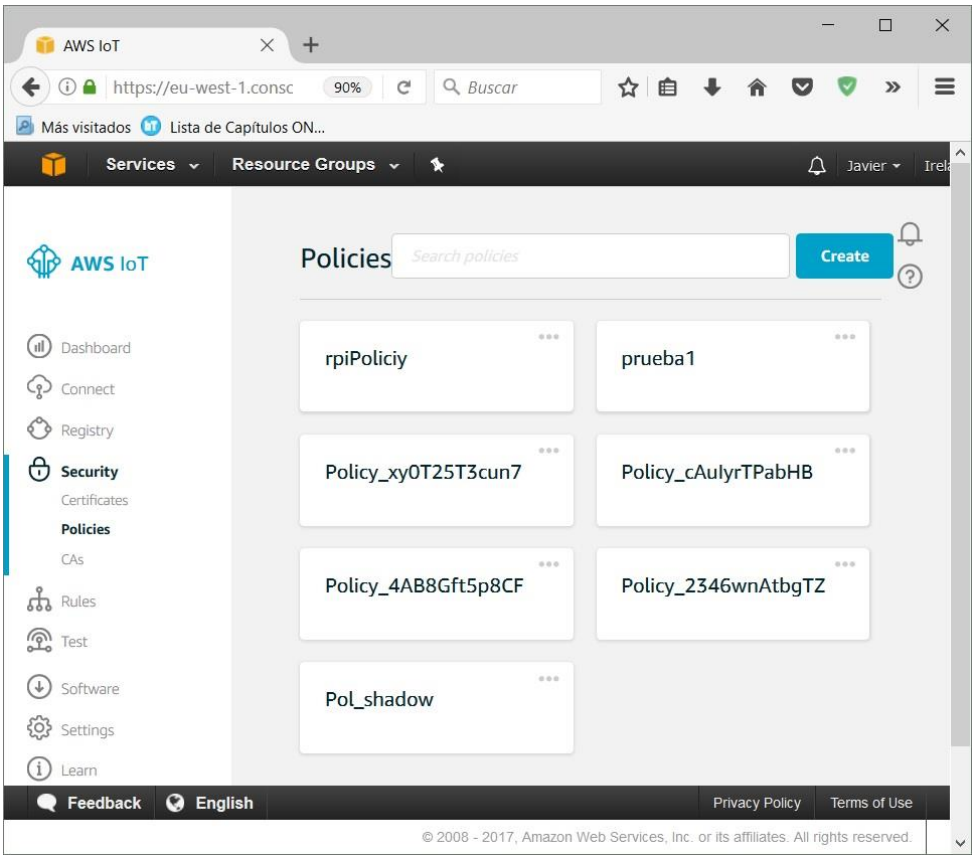

Figura 12: Políticas en AWS IoT.

Por ejemplo, la política del gimnasio 4AB8Gft5p8CF es:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": [
        "arn:aws:iot:eu-west-1:394244280836:client/4AB8Gft5p8CF_1",
        "arn:aws:iot:eu-west-1:394244280836:client/4AB8Gft5p8CF_2",
        "arn:aws:iot:eu-west-1:394244280836:client/4AB8Gft5p8CF_3",
        "arn:aws:iot:eu-west-1:394244280836:client/4AB8Gft5p8CF_4"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": [
        "arn:aws:iot:eu-west-
1:394244280836:topic/fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}",
```

```
        "arn:aws:iot:eu-west-
1:394244280836:topic/fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource":                                                    "arn:aws:iot:eu-west-
1:394244280836:topicfilter/fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/control"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource":                                                    "arn:aws:iot:eu-west-
1:394244280836:topic/fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/control"
    }
  ]
}
```

Vemos que el documento de una política se escribe en formato JSON, y su objeto **Statement** contiene una serie de reglas en las que se especifica si se permite o no (**Effect**) una operación (**Action**) sobre un recurso (**Resource**) [15].

Como vemos, la política de este gimnasio permite a sus dispositivos:

- Conectarse a la nube usando los siguientes id de cliente: `4AB8Gft5p8CF_1`, `4AB8Gft5p8CF_2`, `4AB8Gft5p8CF_3`, `4AB8Gft5p8CF_4`. De esta forma, este gimnasio sólo puede tener 4 dispositivos conectados a la nube de forma simultánea.
- Publicar en los siguientes topics:

  Para datos: `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}`

  `'${iot:ClienteId}'` es una variable cuyo valor es el id de cliente que ha usado el dispositivo al conectarse. Por lo que un dispositivo que se haya conectado usando como id de cliente `4AB8Gft5p8CF_3` podrá publicar en su topic de datos correspondiente, es decir, en `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/4AB8Gft5p8CF_3`

  Para estado: `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/*`

  `'*'` se expande a cualquier grupo de carácteres, por lo que se permite publicar en cualquier topic que "cuelgue" del anterior, como es por ejemplo el topic de estado: `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/status`

- Suscribirse a su topic de control: `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/control`

- Recibir datos de su topic de control: `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/${iot:ClientId}/control`

3.1.2.2 Separación de la información: topics

Los topics no se crean en ningún momento, simplemente cada vez que se envía un mensaje, se indica el topic. Es como una etiqueta que sirve de mecanismo para separar y organizar los mensajes.

El formato escogido para los topics es el siguiente:

- **Topic de datos**

  Cada bundle (corriendo en una Raspberry Pi) dentro de un gimnasio envía la información que recibe de los dispositivos BLE a un topic relativo a él, siguiendo el siguiente patrón:
  `fcoortbon/centralhrp/GIM_{Codigo}/{Codigo}_{Numero}`
  Por ejemplo, el dispositivo número 2 del gimnasio con código 1D23F, publicaría datos de pulso y energía gastada en: `fcoortbon/centralhrp/GIM_1D23F/1D23F_2`

- **Topic de estado**

  Cada bundle publica su estado en un topic que sigue el siguiente patrón:
  `fcoortbon/centralhrp/GIM_{Codigo}/{Codigo}_{Numero}/status`
  Siguiendo el ejemplo anterior: `fcoortbon/centralhrp/GIM_1D23F/1D23F_2 /status`

- **Topic de control**

  Cada bundle se suscribe a un topic de control a través del cuál recibe órdenes. Este topic sigue el siguiente patrón:
  `fcoortbon/centralhrp/GIM_{Codigo}/{Codigo}_{Numero}/control`
  Siguiendo el ejemplo anterior: `fcoortbon/centralhrp/GIM_1D23F/1D23F_2/control`

De este modo, cada dispositivo (Raspberry Pi) diferente envía y recibe datos en topics específicos para él, y no puede enviar ni recibir datos de otros topics, ya que la política asociada a su gimnasio no se lo permite.

3.1.2.3 Persistencia de la información: DynamoDB

Para almacenar la información, como se comentó anteriormente, hacemos uso de la base de datos NoSQL DynamoDB, que es otro de los servicios proporcionados por la nube de Amazon.

Para cada gimnasio se crea una tabla en DynamoDB con el nombre `GIM_{codigo}`. Por ejemplo, un gimnasio con código 1D23F tendría una tabla en la base de datos DynamoDB con el nombre `GIM_1D23F`.

Estas tablas tienen 3 atributos:

1. **Address**: Dirección MAC del dispositivo BLE que proporcionó el dato. Hace de **partition key**.
2. **TimeStamp**: Fecha en la que se creó el dato. En la franja horaria UTC. Hace de **sort key**.
3. **payload**: Contenido completo de un mensaje.

Y para insertar en la tabla los datos de pulso que se vayan recibiendo, se hace uso del motor de reglas de AWS IoT. Por cada gimnasio se crea una regla que se activa con cada mensaje perteneciente a uno de los topic de datos que llegue al **Message broker**. Y cuando se activa la regla, la acción que se lleva a cabo es almacenar la información de este mensaje en la tabla del gimnasio en DynamoDB.

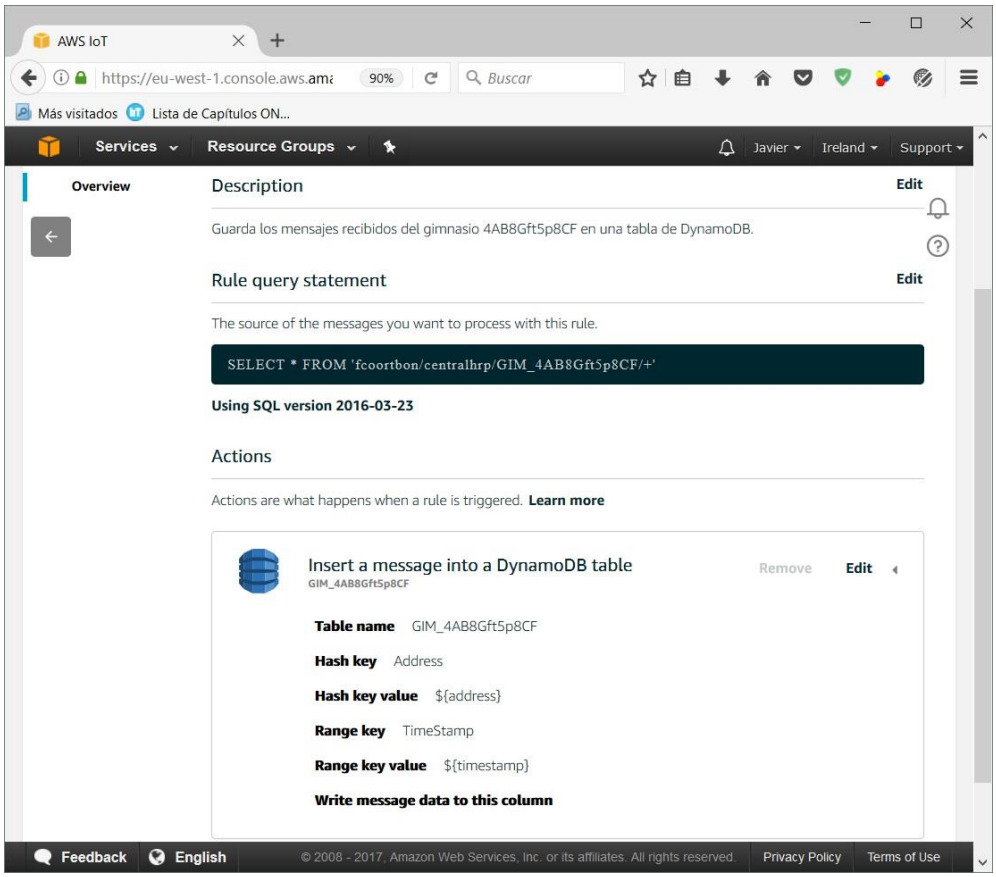

Figura 13: Ejemplo de regla en AWS IoT.

En la figura 13 podemos observar la regla para el gimnasio 4AB8Gft5p8CF que va a dispararse con los mensajes enviados a los topic que cumplan el patrón `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/+`. El carácter '+' está indicando que el topic objetivo está justo un nivel por debajo de `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF`, es decir, un topic como `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/foo` pero no `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/foo/bar`.

Y los únicos topics que coinciden con este patrón son los de datos, que como vimos anteriormente, para el gimnasio 4AB8Gft5p8CF eran tales que así:
`fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/4AB8Gft5p8CF_{Numero}`

Además, como también se observa en la imagen anterior, la acción que se lleva a cabo al dispararse la regla es insertar el contenido del mensaje en la tabla GIM_4AB8Gft5p8CF de DynamoDB:

- En la columna Address se va a insertar el campo **address** del mensaje.
- En la columna TimeStamp se va a insertar el campo **timestamp** del mensaje.
- Adicionalmente, se va a insertar el mensaje entero en una tercera columna, que por defecto se llama payload.

Los mensajes, como se va a ver en un apartado posterior, están en formato JSON.

A continuación, la figura 14 muestra una captura de la tabla del gimnasio 4AB8Gft5p8CF, que podemos observar que no está vacía, pues contiene muchos mensajes de cuando se ha estado probando el sistema.

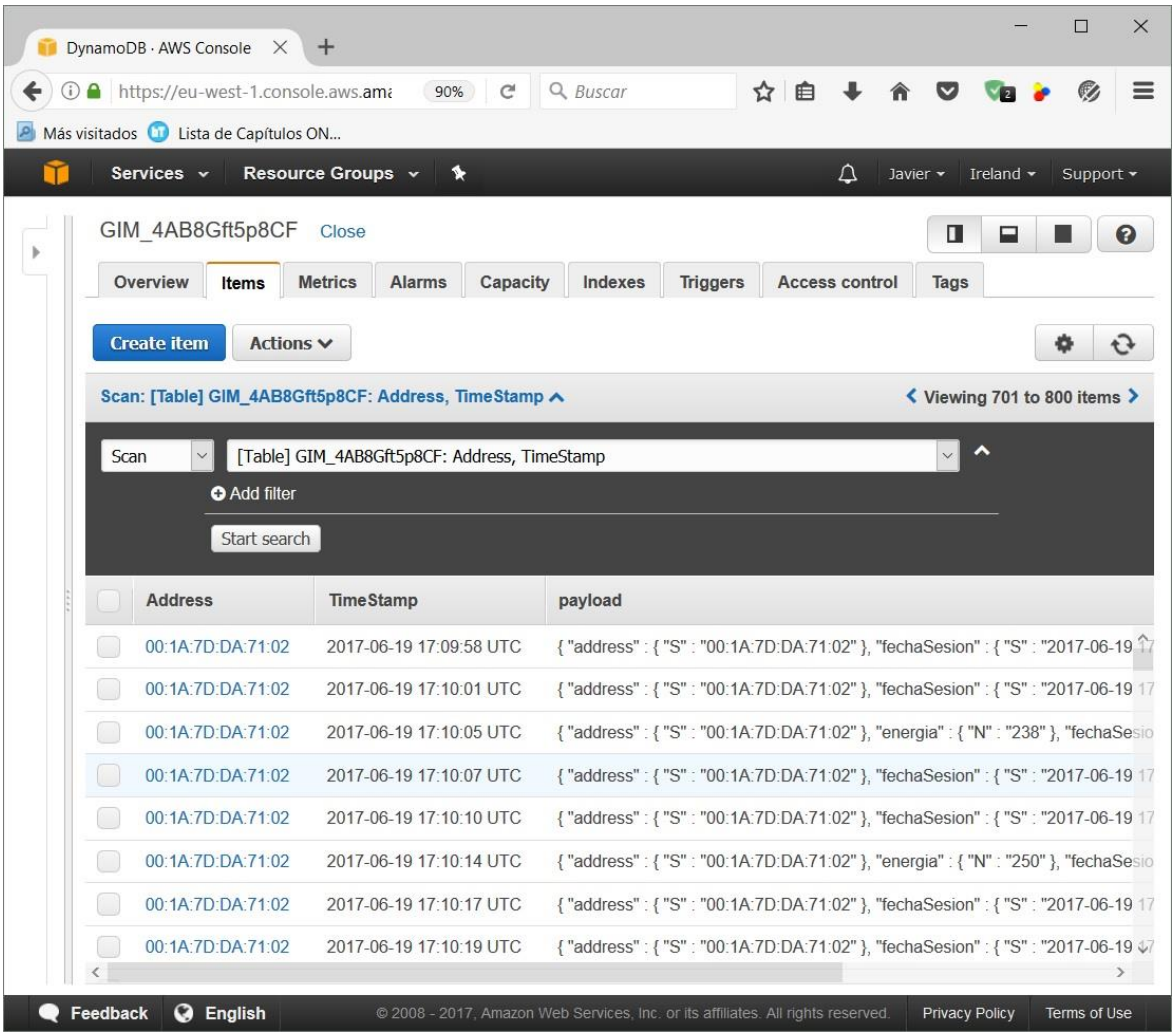

Figura 14: Tabla de un gimnasio en DynamoDB.

3.1.2.4 Configuración de la nube en el dispositivo Kura

Cada Raspberry Pi que se quiera utilizar dentro de un gimnasio debe ser previamente configurada para que se conecte a nuestra nube (que use nuestro endpoint) usando el certificado creado para el gimnasio y con un id de cliente distinto al de las demás.

Todo esto se realiza de forma sencilla desde la interfaz web de Kura, y para ello hay que:

1. Configurar el apartado SSL: aquí añadimos el certificado del gimnasio.

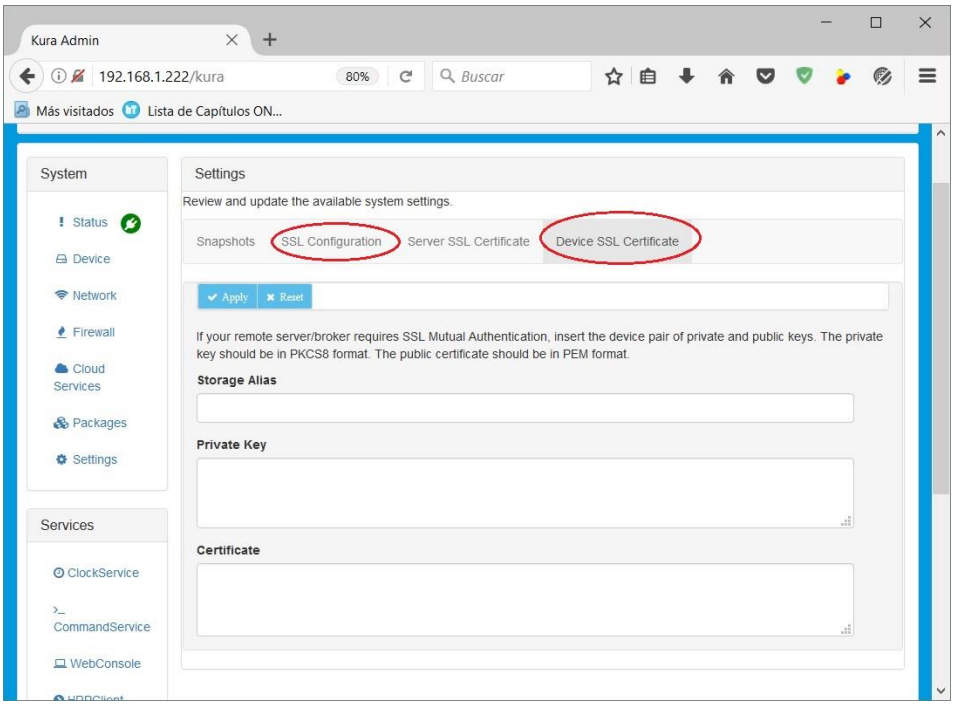

Figura 15: Interfaz web de Kura: Apartado de configuración SSL

2. Configurar el Cloud Service con los parámetros de nuestra nube.

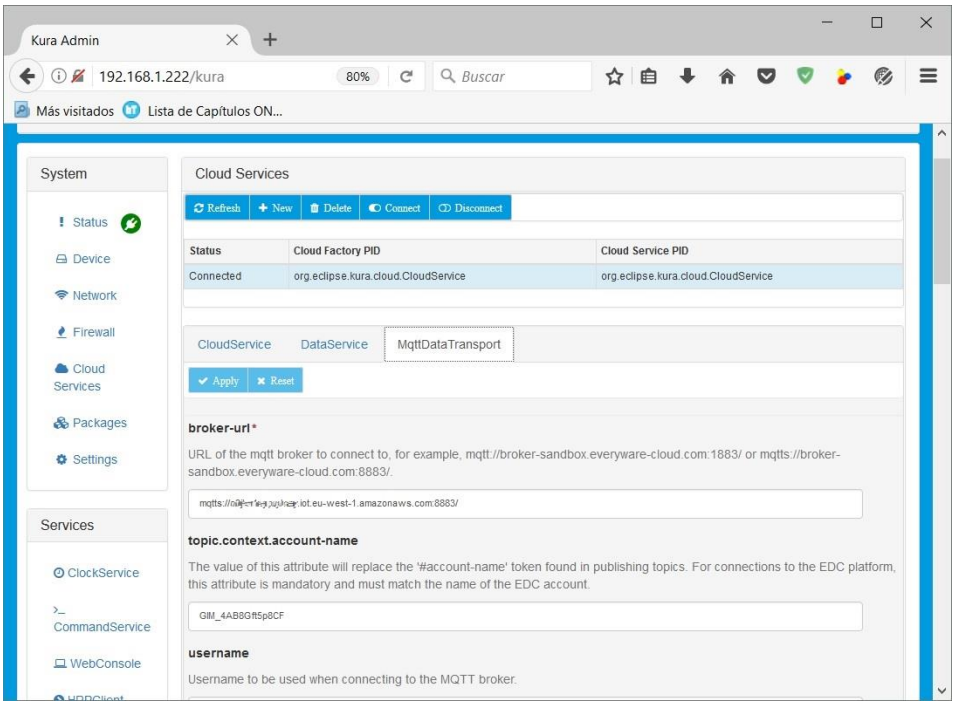

Figura 16: Interfaz web de Kura: Apartado de configuración de un CloudService.

3. Configurar los parámetros del bundle HRPClient

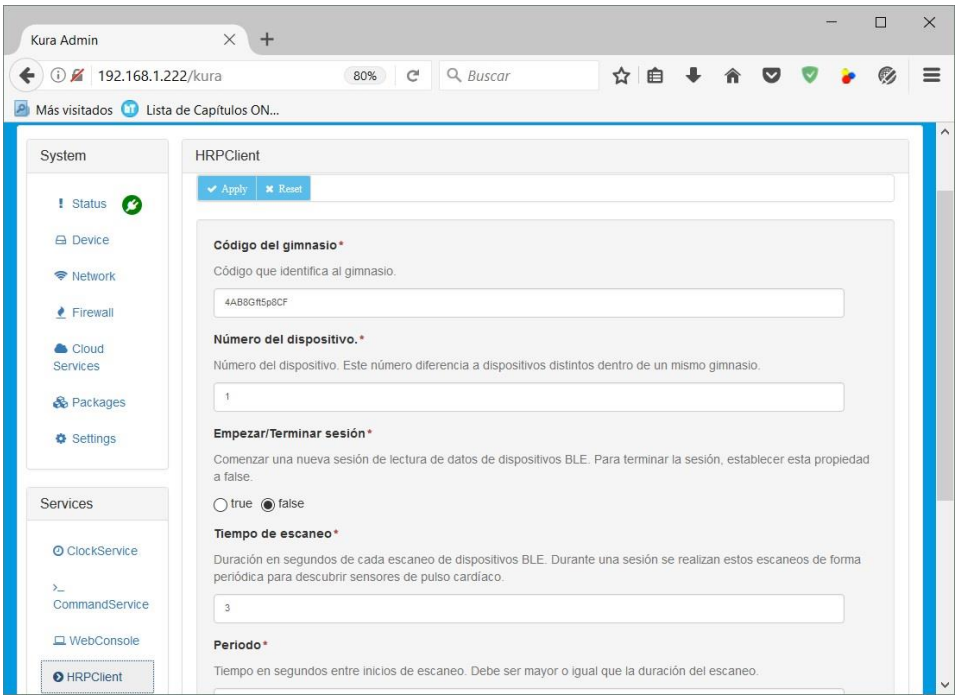

Figura 17: Interfaz web de Kura: Apartado de configuración del bundle HRPClient.

El tema de la configuración de Kura se va a tratar con detalle en el Anexo B: Manual de instalación y configuración de Eclipse Kura en una Raspberry Pi.

3.1.3 Aplicación Android: Control de los bundles y obtención en tiempo real de la información proporcionada por los dispositivos BLE

Se ha explicado cómo un dispositivo Kura (la Raspberry Pi) se conecta y recibe información de los dispositivos BLE que encuentra a su alrededor, y tal y como la recibe, la publica en la nube. Los mensajes que llegan a la nube son recibidos por el **Message broker** de AWS IoT, que como se explicó anteriormente, es el encargado de reenviar mensajes entre publicadores y suscriptores. Así mismo, gracias al uso del motor de reglas de AWS IoT, puede insertarse la información de cada mensaje de datos en la tabla del gimnasio correspondiente.

Sin embargo, ¿cómo se controla el bundle para que empiece y deje de funcionar?, ¿cómo el monitor recibe los datos de pulso y energía gastada de sus alumnos en tiempo real?

Bien, todo esto se hace a través de la aplicación Android desarrollada. Al igual que la idea era que en cada sala del gimnasio donde se dan clases de ejercicio hubiese una Raspberry Pi con Kura y el bundle desarrollado instalado, también es lo ideal que el gimnasio disponga de tablets con la aplicación Android. El motivo por el que se prefiere una tablet frente a un simple smartphone es que la tablet tiene una pantalla mucho más grande, y esto es conveniente ya que la aplicación muestra los datos de cada usuario en una cuadrícula y elabora una gráfica con la evolución del pulso de cada usuario a lo largo del tiempo.

3.1.3.1 Control del bundle HRPClient

Como se ha visto, el bundle **HRPClient** publica los datos proporcionados por los dispositivos BLE en un topic de datos. Adicionalmente, hemos visto que usa otros dos topics, uno para publicar su estado y otro de control al que se suscribe para recibir órdenes.

Las órdenes que puede recibir el bundle son tres:

- `PUBLICAR_ESTADO`: Se ordena al bundle que publique su estado. Al recibir esta orden, el bundle publica su estado en el topic de estado. Su estado puede ser `ESTADO_IDLE` ó `ESTADO_EN_SESION`.
- `COMENZAR_SESIÓN`: Se ordena al bundle que comience una sesión. Al recibir esta orden, si no estaba ya en sesión, el bundle comenzará a funcionar y su estado pasará de `ESTADO_IDLE` a `ESTADO_EN_SESION`.
- `TERMINAR SESIÓN`: Se ordena al bundle que termine una sesión. Al recibir esta orden, si está en el estado `ESTADO_EN_SESION`, es decir, si está funcionando, termina de funcionar y pasa a estado `ESTADO_IDLE`.

Estos estados y órdenes se recogen en la clase `ControlHRPClient`:

```
package fcoortbon.ble.central_hrp;

public class ControlHRPClient
{
    public static final int COMENZAR_SESION = 1;
    public static final int TERMINAR_SESION = 2;
    public static final int PUBLICAR_ESTADO = 3;

    public static final int ESTADO_IDLE = 1;
    public static final int ESTADO_EN_SESION = 2;
}
```

La idea es, entonces, que un monitor cuando vaya a empezar una clase, envíe la orden `COMENZAR_SESION` a la Raspberry Pi de la sala, y que cuando acabe la clase, le envíe la orden `TERMINAR_SESION`.

Para ello, la aplicación Android dispone de un par de botones que permiten iniciar y acabar sesiones en un bundle HRPClient. Para ello, la aplicación envía las órdenes anteriores al dispositivo, y si consigue iniciarse una sesión, la aplicación comenzará a recibir los datos publicados a la nube y a mostrarlos en la pantalla.

Figura 18: Detalle de la aplicación: botones.

Previamente, en la aplicación se ha debido de configurar con qué dispositivo HRPClient (Raspberry Pi) se quiere comunicar, para lo que es necesario indicar el código del gimnasio y el número del dispositivo.

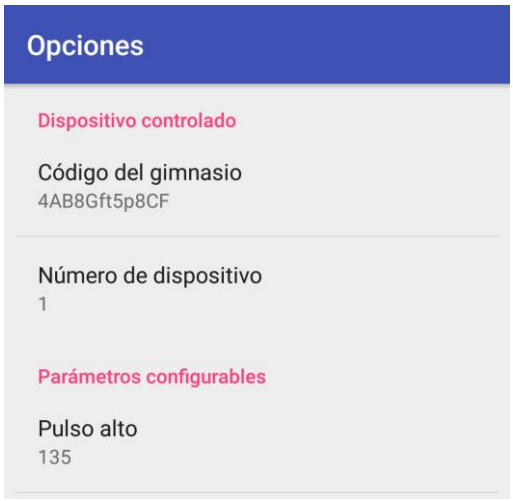

Figura 19: Detalle de la aplicación: configuración.

A continuación, la figura 20 muestra un diagrama en el que se ve el intercambio de mensajes que se ha producido en el caso de haber iniciado una sesión con éxito:

Figura 20: Intercambio de mensajes dado cuando se ha iniciado una sesión con éxito.

3.1.3.2 Obtención y presentación de los datos en tiempo real

Una vez se ha iniciado una nueva sesión en un dispositivo HRPClient, la aplicación comenzará a recibir los datos que publique el dispositivo y los irá mostrando en la pantalla.

La presentación de los datos se realiza de la siguiente manera: por cada dispositivo BLE que esté enviando datos, aparece un rectángulo en la pantalla relativo a él con los últimos datos recibidos de pulso y energía gastada. Adicionalmente, si se hace clic en uno de ellos, se muestra una gráfica con la evolución del pulso a lo largo del tiempo.

A continuación se muestran algunas imágenes de la aplicación recibiendo datos y mostrándolos en la pantalla.

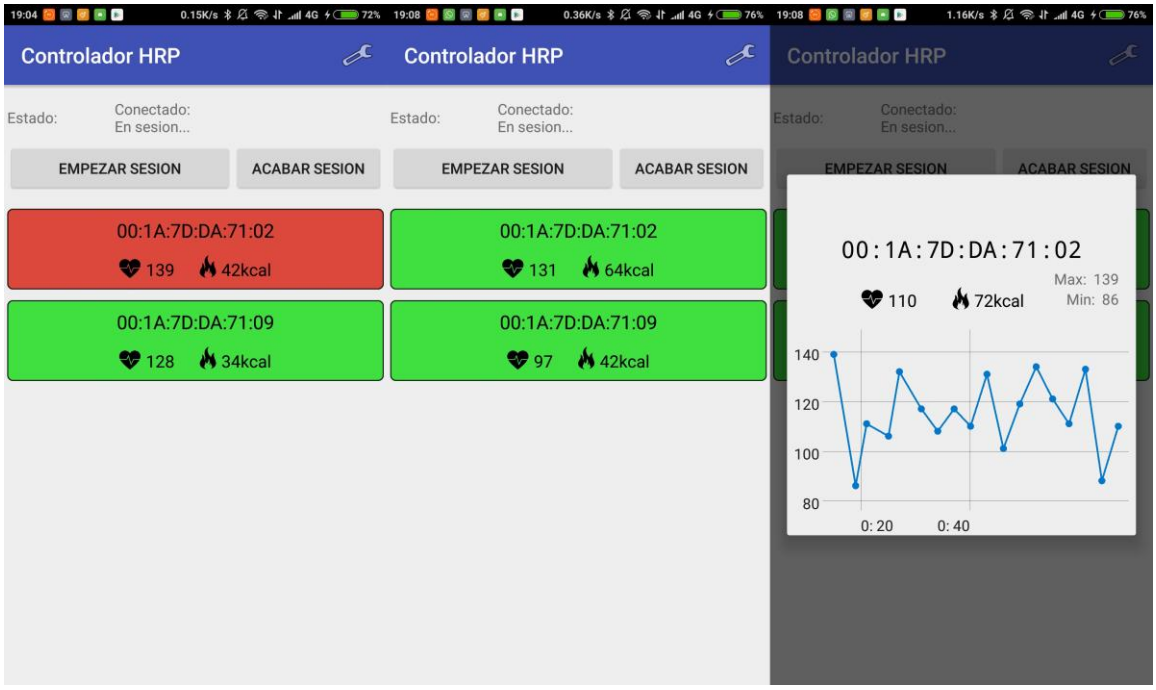

Figura 21: Aplicación en funcionamiento (vertical).

Figura 22: Aplicación en funcionamiento (horizontal).

En este caso, se estaba haciendo uso de un par de sensores emulados, que son los que aparecen en la aplicación. Como puede observarse, el color de los rectángulos puede variar, ya que:

- Si la medida de pulso que se muestra de un dispositivo BLE es antigua (se recibió hace más de 12s), se colorea en gris.
- Si la medida de pulso es reciente y está por encima del valor configurado como pulso alto, se colorea en rojo.

- Si la medida de pulso es reciente y está por debajo del valor configurado como pulso alto, se colorea en verde.

3.2 Formato de la información

Para el intercambio de información entre los distintos componentes del sistema siempre se hace uso de AWS IoT y su **Message broker**. De esta manera, la información que se envía se hace mediante la publicación de un mensaje en un topic determinado, y para que un componente reciba información, debe haberse suscrito previamente a algún topic.

Como ya se ha explicado antes, el **Message broker** es el intermediario entre publicadores y suscriptores: Recibe los mensajes de todos los topics y los hace llegar a los suscriptores que estén suscritos a cada topic.

Los mensajes se envían siempre en formato JSON, y por ahora, existen tres tipos de mensajes:

1. Mensaje de datos.
2. Mensaje de estado de dispositivo.
3. Mensaje de orden hacia el dispositivo.

3.2.1 Mensaje de datos

Los mensajes de datos son publicados por los dispositivos HRPClient en sus respectivos topics de datos cada vez que estos reciben un nuevo dato por parte de un dispositivo BLE. Un mensaje de datos consta de los siguientes campos:

- `address`: Dirección MAC del dispositivo BLE que ha proporcionado el dato.
- `fechaSesion`: Fecha del inicio de la sesión en el bundle HRPClient. En la franja horaria UTC.
- `pulso`: Valor en **bpm** (pulsaciones por minuto) de la medida de pulso recibida.
- `energia`: Valor recibido de energía gastada acumulada. En kilojulios. Este campo no se incluye siempre ya que el valor de energía gastada acumulada puede estar o no en los datos de pulso recibidos por un dispositivo BLE.
- `timestamp`: Fecha del momento en el que el bundle HRPClient recibió estos datos.

Un ejemplo de un mensaje de datos es el siguiente:

```
{
    "address": "00:1A:7D:DA:71:02",
    "energia": 555,
    "fechaSesion": "2017-06-20 10:45:05 UTC",
    "pulso": 115,
    "timestamp": "2017-06-20 10:52:03 UTC"
}
```

3.2.2 Mensaje de estado

Los mensajes de estado son publicados por los dispositivos HRPClient en sus respectivos topics de estado. Un dispositivo HRPClient publica su estado cuando:

- Comienza una nueva sesión, es decir, empieza a funcionar.
- Acaba una sesión, es decir, pasa de estar funcionando a estado `ESTADO_IDLE`.

- Recibe la orden `PUBLICAR_ESTADO`.

Un mensaje de estado consta de los siguientes campos:

- `estado`: Indica el estado del dispositivo. Recordemos que este podía ser `ESTADO_IDLE` (se encuentra a la espera de órdenes) o `ESTADO_EN_SESION` (se encuentra actualmente funcionando).
- `fechaIni`: Indica la fecha en la que se inició la sesión que se está dando actualmente (en caso de que se encuentre en estado `ESTADO_EN_SESION`) ó la fecha en la que se inició la última sesión (en caso de que se encuentre en estado `ESTADO_EN_IDLE`).
- `fechaFin`: Indica la fecha en la que acabó la última sesión (en caso de que se encuentre en estado `ESTADO_EN_IDLE`). Si se encuentra en estado `ESTADO_EN_SESION`, el campo estará vacío.

Un ejemplo de un mensaje de estado de un dispositivo que se encuentra en sesión es el siguiente:

```
{
    "estado": 2,
    "fechaIni": "2017-06-20 10:45:05 UTC",
    "fechaFin": ""
}
```

3.2.3 Mensajes de orden para el dispositivo

Los mensajes de órdenes son publicados por la aplicación Android en los topics de control de los dispositivos, de acuerdo a los deseos del usuario. Un mensaje de orden contiene un único campo:

- `action`: Indica el código de la orden para el dispositivo. Recordemos que podía ser `COMENZAR_SESION`, `TERMINAR_SESION` ó `PUBLICAR_ESTADO`.

Por ejemplo, si un usuario quiere iniciar una nueva sesión en el dispositivo HRPClient número 3 del gimnasio con código 4AB8Gft5p8CF, publicaría en el topic de control de ese dispositivo, es decir en `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/4AB8Gft5p8CF_3/control`, el siguiente mensaje:

```
{
    "action": 1
}
```

4 SENSORES EMULADOS DE PULSO

Para disponer de dispositivos BLE que ofrecieran el servicio de pulso cardíaco, se han programado unos sensores emulados de pulso.

Cada sensor no es más que un programa escrito en Node.js que hace uso de una interfaz Bluetooth LE (4.0 ó superior) para emular un periférico BLE. Para ello se ha usado un módulo de Node.js llamado **bleno**, que sirve para implementar periféricos BLE.

El periférico implementado ofrece un único servicio, que es el de pulso cardíaco, y para implementarlo se ha seguido la especificación de este servicio proporcionada por el Bluetooth SIG.

Además, necesitamos un adaptador Bluetooth LE por cada sensor emulado de pulso que queramos ejecutar, puesto que necesita "adueñarse" de una interfaz Bluetooth LE, la cual convierte en un periférico BLE.

4.1 Servicio de pulso cardíaco

Como vimos anteriormente, los periféricos BLE ofrecen servicios, y estos a su vez se componen de una serie de características.

La especificación ofrecida por el Bluetooth SIG del servicio cardíaco puede encontrarse aquí:

https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.service.heart_rate.xml

Vemos que el servicio contiene tres características:

- Heart Rate Measurement
- Body Sensor Location
- Heart Rate Control Point

Y que el número asignado por el Bluetooth SIG para este servicio es **180d** (en hexadecimal), que como se explicó en el capítulo de tecnologías utilizadas, es un UUID de 16 bits.

4.1.1 Característica Heart Rate Measurement

El número asignado para esta característica es **2a37**.

Esta característica es usada para enviar una medida de pulso y es obligatorio incluirla en el servicio de pulso cardíaco. La única propiedad que puede –y debe también– implementar es **Notify**, es decir, que puedan activarse o desactivarse las notificaciones para esta característica.

A su vez, cada valor de la característica Heart Rate Measurement contiene una serie de campos, indicados en la siguiente tabla.

Tabla 1: Campos de la característica Heart Rate Measurement

Campo	Requerimiento	Formato
Flags	Obligatorio	8 bits
Heart Rate Measurement Value	Obligatorio	uint8 ó uint16: indicado en flags.
Energy Expended	Opcional. Indicado en flags.	uint16
RR-Interval	Opcional. Indicado en flags.	uint16

Donde **flags** es un campo de opciones, **Heart Rate Measurement Value** es el valor de pulso, **Energy Expended** es el valor de energía total gastada y **RR-interval** el valor del intervalo entre latidos consecutivos del corazón.

El campo flags contiene 8 bits:

- **bit 0**

  Indica el formato del valor de pulso (Heart Rate Measurement Value).

  0: uint8.

  1: uint16.

- **bits 1, 2**

  Indican si la funcionalidad de sensor de contacto está o no soportada:

  00: No soportada en la conexión actual.

  01: No soportada en la conexión actual.

  10: Soportada, pero no se detecta contacto.

  11: Soportada y con contacto detectado.

- **bit 3**

  Indica si se encuentra o no presente el campo de energía gastada (Energy Expended).

  0: No.

  1: Sí. Unidad kilojulios.

- **bit 4**

  Indica si se encuentran presentes valores de RR-Interval.

  0: No.

  1: Uno o más valores de RR-Interval se encuentran presentes.

- **bits 5, 6, 7**

  Reservados para uso futuro.

Entonces, la característica Heart Rate Measurement debe contener siempre dos valores: un campo de opciones y un valor de pulso cardíaco. De forma opcional, pueden incluirse un par de campos más: uno con el valor de la energía gastada total y otro con el valor de los intervalos entre latidos del corazón.

La especificación de esta característica puede consultarse aquí:

https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml

4.1.2 Característica Body Sensor Location

Su número asignado es **2a38**.

Esta característica es más simple que la anterior, pues sólo contiene un campo que es Body Sensor Location, de 8 bits, y que representa el lugar del cuerpo en el que debería colocarse el dispositivo BLE con el sensor que mide el pulso cardíaco.

La única propiedad que puede –y debe también– implementar es **Read**, es decir, que permita poder leer su valor a otro dispositivo BLE con el que estamos conectados.

Esta característica, al contrario que la anterior, es opcional. Es decir, no es obligatorio incluirla en el servicio de pulso cardíaco.

Los posibles valores que puede tener son los siguientes:

0: Otro.

1: Pecho.

2: Muñeca.

3: Dedo.

4: Mano.

5: Lóbulo de la oreja.

6: Pie.

7-255: Reservado para uso futuro.

La especificación de esta característica puede consultarse aquí:

https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml

4.1.3 Característica Heart Rate Control Point

Su número asignado es **2a39**.

Al igual que la característica anterior, contiene un único campo de 8 bits, llamado Heart Rate Control Point, que representa una orden enviada hacia el dispositivo.

La única propiedad que puede –y debe también– implementar es **Write**, es decir, que permita poder escribir su valor a otro dispositivo BLE con el que estamos conectados, y así que este pueda enviarle órdenes.

Esta característica, es opcional, pero si la funcionalidad de medir la energía gastada está soportada por el dispositivo, entonces es obligatoria incluirla en el servicio de pulso cardíaco.

Los posibles valores que puede tener son los siguientes:

0: Reservado.

1: Orden de reiniciar el contador de energía gastada.

Como se vio antes, uno de los campos de la característica Heart Rate Measurement era Energy Expended (energía gastada). Este campo proporcionaba el total de la energía gastada por el usuario del dispositivo, en kilojulios, y tenía el formato uint16. Esto quiere decir que lo máximo que podía medir eran 2^16-1 = 65535 kilojulios de energía gastada. En caso de llegar a ese valor, ya no se incrementa más y se continúa enviando 65535.

Mediante el envío de esta orden, el dispositivo central que recibe esta información tiene la posibilidad de reiniciar el contador para poder así seguir midiendo la energía gastada por el usuario.

2-255: Reservado para uso futuro.

4.2 Módulo bleno

Bleno es un módulo para Node.js que permite implementar periféricos BLE de manera sencilla.

Su documentación puede encontrarse aquí: https://github.com/sandeepmistry/bleno

Vemos que para poder usarlo en Linux, previamente necesitamos una serie de dependencias, como puede ser la pila Bluetooth, BlueZ.

Para comenzar a usarlo en nuestro programa Node.js, lo incluimos de la siguiente manera

```
var bleno = require.('bleno');
```

A partir de aquí podemos programar que se anuncie y los distintos servicios que queremos que ofrezca.

Podemos ver ejemplos aquí: https://github.com/sandeepmistry/bleno/tree/master/examples

4.3 Implementación

El servicio de pulso cardíaco que vamos a implementar dispone de la funcionalidad para medir la energía gastada, pero no dispone de la funcionalidad de sensor de contacto con la piel, ni incluirá valores de RR-interval en los valores de la característica Heart Rate Measurement. También incluye la característica Body Sensor Location, indicando que el sensor debe colocarse en la muñeca.

De este modo, el servicio que implementamos:

1. Usa el UUID del servicio de pulso cardíaco, **180d.**
2. Contiene la característica Heart Rate Measurement.
   - UUID **2a37**.
   - Propiedad notify.
   - Se manejan los eventos de suscripción y cancelación de suscripción de esta característica. Cada vez que un dispositivo se suscribe, se le envían valores aleatorios cada X tiempo, y se dejan de enviar cuando se desuscribe o desconecta.
3. Contiene la característica Body Sensor Location.
   - UUID **2a38.**
   - Propiedad read.
   - Valor: 2 (Muñeca).
   - Se maneja el evento de lectura. Cuando un dispositivo lleva a cabo la operación de leer sobre esta característica, se devuelve su valor.
4. Contiene la característica Heart Rate Control Point.
   - UUID **2a39**.
   - Propiedad write.
   - Se maneja el evento de escritura. Cuando un dispositivo escribe un nuevo valor, se interpreta este valor como una orden. Sólo se conoce la orden de reinicar el contador de energía gastada acumulada, cuyo código es 1. Si se recibe esta orden, se pone a cero la variable que lleva el conteo de energía gastada.

4.3.1 Código

- Primero, declaramos unas constantes con los números asignados por el Bluetooth SIG.

```
const SERVICIO_HEART_RATE = '180d';
const CARACT_HEART_RATE_MEASUREMENT = '2a37';
const CARACT_BODY_SENSOR_LOCATION = '2a38';
const CARACT_HEART_RATE_CONTROL_POINT = '2a39';

const CODIGO_RESET_ENERGY = 0x01;
const CODIGO_SENSOR_MUÑECA = "02";
```

- Declaramos un par de variables que indican cada cuanto tiempo (en ms) se notifica una nueva medida de pulso, y con qué frecuencia (una de cuántas veces) se incluye valor de energía gastada.

```
var periodoEnvio = 3000;
var freqEnergia = 3;
```

- Declaramos un par de variables de control y otra para llevar el conteo de la energía gastada.

```
var index = 1;
var expendedEnergy = 0;
var subscribed = false;
```

La variable index determina si se incluye o no valor de energía gastada. Con cada medida de pulso se incrementa en uno, y si es igual a freqEnergia se incluye medida de energía y se establece su valor a 1.

La variable subscribed indica si hay o no un dispositivo suscrito a la característica Heart Rate Measurement.

- Declaramos un par de funciones auxiliares. Una para generar un número aleatorio entre dos valores proporcionados y otra para construir un valor de la característica Heart Rate Measurement, con valores de pulso y energía aleatorios.

```
function randomInt (low, high)
{
    return Math.floor(Math.random() * (high - low + 1) + low);
}

function makeHRM(iBPM, sBPM, iEE, sEE, includeEE)
{
  var pulso = randomInt(iBPM, sBPM);
  if (includeEE)
  {
    expendedEnergy += randomInt(iEE, sEE);
    if (expendedEnergy > 65535)
```

```
      expendedEnergy = 65535;
    var HRM = new Buffer(4);
    HRM.writeUInt16LE(expendedEnergy, 2);
    console.log("Pulso:  "+pulso+"bpm,  Energia:  "+expendedEnergy+"kJ
!");
    HRM[0] = 0b00110000;
  }
  else
  {
    var HRM = new Buffer(2);
    console.log("Pulso: "+pulso+"bpm");
    HRM[0] = 0b00100000;
  }
  HRM.writeUInt8(pulso, 1);
  return HRM;
}
```

- Usamos bleno para manejar los eventos de conexión y desconexión con un dispositivo y para que nuestro periférico se anuncie.

```
var bleno = require('bleno');

bleno.on('stateChange', function(state) {
    console.log('State change: ' + state);
    if (state === 'poweredOn') {
        bleno.startAdvertising('SensorEmulado',['180d']);
    } else {
        bleno.stopAdvertising();
        clearInterval(this.intervalId);
    }
});

bleno.on('accept', function(clientAddress) {
    console.log("Aceptada conexion de: " + clientAddress);
});

bleno.on('disconnect', function(clientAddress) {
    console.log("Desconectado de: " + clientAddress);
});
```

- Cuando el periférico comience a anunciarse, añadimos el servicio de pulso cardíaco, creándolo como se ha descrito al principio de este apartado 4.3.

```
bleno.on('advertisingStart', function(error) {
  if (error) {
    console.log("Error al empezar a anunciarse:" + error);
  }
  else {
    console.log("Anunciandose...");
    // Servicios que ofrecemos
    bleno.setServices([
    // Servicio de pulso cardíaco
    new bleno.PrimaryService({
      uuid : SERVICIO_HEART_RATE,
      // Características dentro del servicio
      characteristics : [
        // Característica Heart Rate Measurement
        new bleno.Characteristic({
          value : null,
          uuid : CARACT_HEART_RATE_MEASUREMENT,
          properties : ['notify'],

          // Si se suscriben a esta característica, cada periodoEnvio ms
          // enviamos una nueva medida de pulso.
          onSubscribe : function(maxValueSize, updateValueCallback) {
            clearInterval(this.intervalId);
            console.log("Cliente suscrito a caracteristica Heart Rate Measurement
!");
            this.intervalId = setInterval(function() {
              var includeEE = false;
              // Incluimos medida de energia acumulada gastada una vez cada
              // freqEnergia veces.
              if (index == freqEnergia)
              {
                includeEE = true;
                index = 0;
              }
              index++;
              var HeartRateMeasurement = makeHRM(80,140,10,15,includeEE)
              updateValueCallback(HeartRateMeasurement);
            }, periodoEnvio);
          },
```

```
// Si el cliente se desuscribe, dejamos de enviar medidas de pulso.
          onUnsubscribe : function() {
            console.log("Cliente desuscrito a caracteristica Heart Rate
Measurement.");
            clearInterval(this.intervalId);
          }
         }),
        // Característica Body Sensor Location
        new bleno.Characteristic({
          value : new Buffer(CODIGO_SENSOR_MUÑECA, "hex"),
          uuid : CARACT_BODY_SENSOR_LOCATION,
          properties : ['read'],

          onReadRequest : function(offset, callback) {
            console.log("Peticion de lectura de caracteristica Body Sensor
Location");
            callback(this.RESULT_SUCCESS, this.value);
          }
        }),
        // Característica Heart Rate Control Point
        new bleno.Characteristic({
          value : null,
          uuid : CARACT_HEART_RATE_CONTROL_POINT,
          properties : ['write'],

          // Petición de escritura
          onWriteRequest : function(data, offset, withoutResponse, callback) {
          // this.value = data;
            if (data.toString("hex") == CODIGO_RESET_ENERGY)
            {
              console.log('Recibida orden de reiniciar el contador de energia...');
              expendedEnergy = 0;
  console.log('Contador de energia gastada acumulada reiniciado !');
              callback(this.RESULT_SUCCESS);
            }
            else
            {
              console.log('Heart Rate Control Point: Recibido codigo desconocido: ' +
data.toString("hex"));
              callback(0x80);
            }
          }
```

```
          })
        ]
      })
    ]);
  }
});
```

4.6.1 Ejecución

En esta sección se va a explicar cómo ejecutar el sensor emulado en un equipo Linux.

Para ejecutarlo, necesitamos disponer de un adaptador Bluetooth LE (4.0 ó superior) conectado al equipo. Vamos a suponer que el nombre de su interfaz es **`hci0`**.

Para que se ejecute bien, primero debemos asegurarnos de que el servicio bluetoothd no está activo. Desde la terminal:

```
$ sudo kill -9 `pidof bluetoothd`
```

También debemos activar el adaptador bluetooth en caso de que estuviera desactivado:

```
$ sudo hciconfig hci0 up
```

Nos situamos en el directorio donde se encuentra el fichero Node.js y lo ejecutamos de la siguiente manera:

```
$ sudo BLENO_HCI_DEVICE_ID=0 node sensor_emulado.js
```

Donde mediante el parámetro `BLENO_HCI_DEVICE_ID` indicamos la interfaz del adaptador bluetooth. En este caso indicamos la 0, puesto que hemos dicho que la interfaz era **`hci0`**.

4.6.2 Prueba

Para probar el sensor, vamos a ejecutarlo y desde nuestro smartphone, usando la aplicación **BLE Scanner**[17]**,** vamos a conectarnos al sensor y a activar las notificaciones de pulso.

A continuación, en las figuras 23 y 24 se muestran los resultados en una terminal Linux y en un smartphone al realizar los siguientes pasos:

1. Ejecutamos el sensor emulado de pulso.
2. Escaneamos con la aplicación BLE Scanner.
3. Nos conectamos al sensor desde la aplicación.
4. Leemos la característica Body Sensor Location.
5. Activamos las notificaciones para la característica Heart Rate Measurement. Permanecemos suscritos unos 10 segundos.
6. Desactivamos las notificaciones.
7. Nos desconectamos.

[17] https://play.google.com/store/apps/details?id=com.macdom.ble.blescanner&hl=es

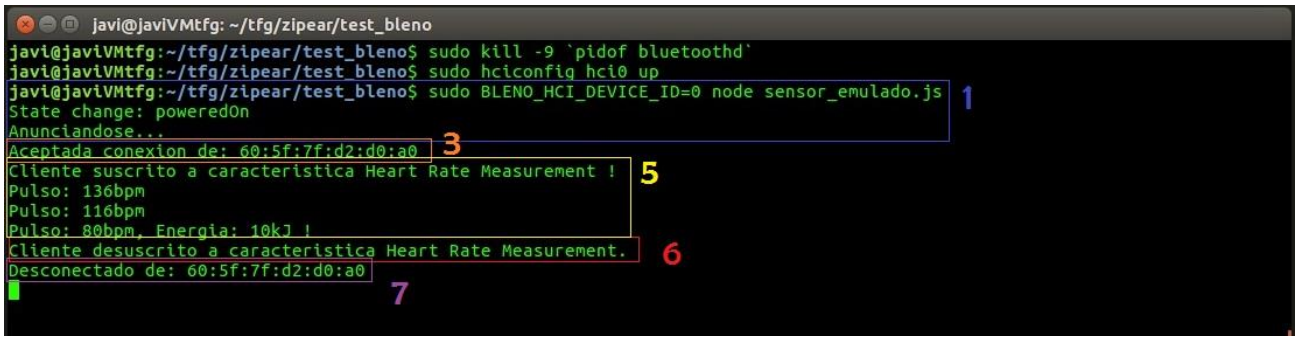

Figura 23: Prueba del sensor emulado. Terminal de Linux.

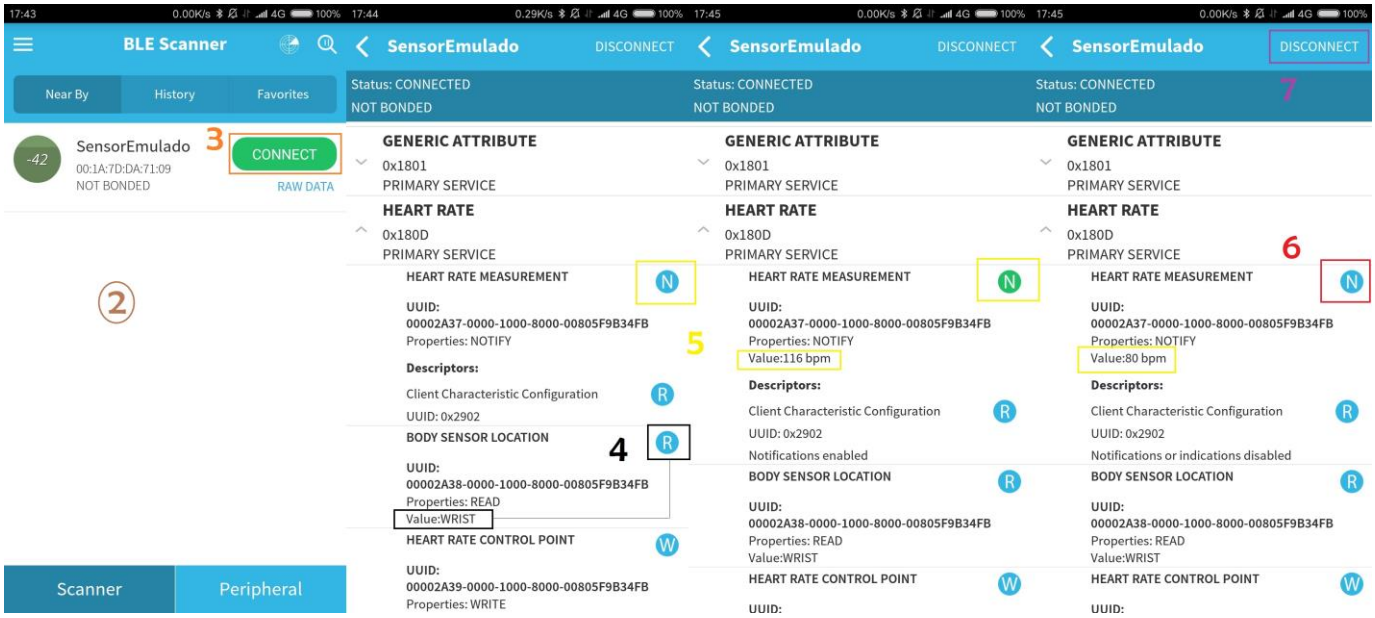

Figura 24: Prueba del sensor emulado. Aplicación Android BLE Scanner.

5 BUNDLE HRPCLIENT

A lo largo de la memoria se ha hablado largo y tendido sobre el bundle HRPClient. Se ha visto que su funcionamiento se puede controlar de forma remota, que es el encargado de conectarse y recibir información de dispositivos BLE, la cual publica en la nube, y que corre en los dispositivos Kura, que en nuestro caso son las Raspberry Pi.

Sin embargo, no se ha hablado sobre su implementación y desarrollo, y eso lo que se va a tratar en este capítulo.

5.1 Introducción

Como ya se comentó en la sección de OSGi del capítulo 2, los bundles son básicamente un JAR y un manifiesto. Y para desarrollar bundles para dispositivos Kura, debemos usar el entorno de desarrollo Eclipse con el workspace de Kura incorporado.

En la documentación de Eclipse Kura hay una serie de ejemplos de bundles explicados a modo tutorial que han sido muy útiles a la hora de desarrollar nuestro bundle HRPClient. Los podemos encontrar aquí: http://eclipse.github.io/kura/ en el apartado Development.

En la figura 25 se muestran las dependencias, clases y recursos del bundle, desde el proyecto de Eclipse.

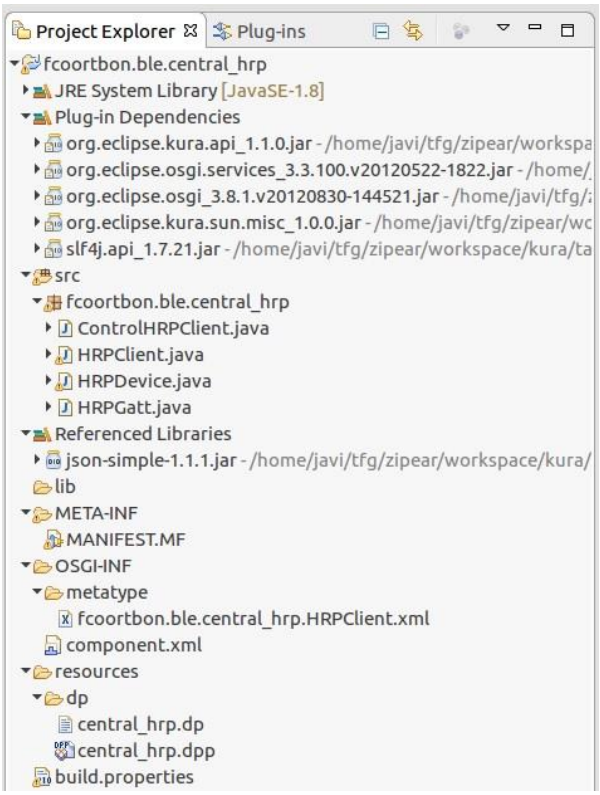

Figura 25: Proyecto en Eclipse para el bundle HRPClient.

En los siguientes apartados se van a tratar los ficheros de metadatos del bundle, los servicios que usa y los que exporta (al estilo OSGi), las clases en las que consiste y su funcionamiento.

5.2 Ficheros de metadatos

Los ficheros que proporcionan metainformación acerca de nuestro bundle son MANIFEST.MF y component.xml.

5.2.1 MANIFEST.MF

El fichero manifiesto contiene una serie de cabeceras, algunas específicas de OSGi. Además, Eclipse dispone de una herramienta para editar los archivos de manifiesto de forma sencilla, sin tener que escribirlos a mano. El fichero MANIFEST.MF de nuestro proyecto es el siguiente:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Cliente central BLE
Bundle-SymbolicName: fcoortbon.ble.central_hrp
Bundle-Version: 1.0.0.qualifier
Bundle-ClassPath: .,
 lib/json-simple-1.1.1.jar
Import-Package: org.eclipse.kura.bluetooth;version="1.3.0",
 org.eclipse.kura.configuration;version="1.1.1",
 org.eclipse.kura.message;version="1.1.0",
 org.osgi.service.component;version="1.2.0",
 org.slf4j;version="1.7.21"
Require-Bundle: org.eclipse.kura.api,
 org.eclipse.osgi.services;bundle-version="3.3.100"
Service-Component: OSGI-INF/component.xml
```

La cabecera **`Manifest-Version`** indica la versión de la especificación para ficheros manifiesto (Manifest Specification) que sigue el fichero.

Las cabeceras que empiezan por Bundle- son las específicas de OSGi, y los metadatos que se especifican en estas cabeceras permiten que la infraestructura OSGi procese los aspectos modulares del paquete [16].

- **`Bundle-ManifestVersion: 2`**

  Indica que el bundle sigue las reglas de la especificación 2.

- **`Bundle-Name`**: Indica el nombre del bundle.
- **`Bundle-SymbolicName`**: Indica el nombre que identifica al paquete de forma exclusiva.
- **`Bundle-Version`**: Indica la versión del paquete. De esta forma, varias versiones de un paquete pueden estar activas simultáneamente en la misma instancia de la infraestructura.
- **`Bundle-ClassPath`**: Indica dónde buscar las clases necesarias para la ejecución del bundle, y que están incluidas en el proyecto.

Ha sido necesario especificar el **classpath** para nuestro bundle porque además de librerías de Kura, hemos hecho uso de un JAR para el uso de JSON, llamado **JSON.simple**[18], y debemos indicar dónde se encuentra (en nuestro caso, se sitúa en la carpeta lib del proyecto).

Luego, mediante las cabeceras **`Import-Package`** y **`Require-Bundle`** indicamos las dependencias y librerías del workspace de Kura usadas por nuestro bundle. Mediante la cabecera **`Require-Bundle`** indicamos en qué bundles buscar paquetes o clases. Por ejemplo, las clases utilizadas para el manejo de Bluetooth LE se encuentran en el bundle **`org.eclipse.kura.api`**.

Por último, en la cabecera **`Service-Component`** especificamos la ruta (local al proyecto) donde se sitúa un fichero de "definición de componente" (En Eclipse ***`New > Other > Plugin Development > Component Definition`***).

5.2.2 component.xml

Mediante este fichero indicamos cuál es la clase principal del bundle, es decir, la clase componente, y también los servicios que implementa y exporta y los que utiliza [17]. El fichero component.xml de nuestro bundle es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
        activate="activate"
        configuration-policy="require"
        deactivate="deactivate"
        enabled="true"
        immediate="true"
        modified="updated"
        name="fcoortbon.ble.central_hrp.HRPClient">

  <implementation class="fcoortbon.ble.central_hrp.HRPClient"/>

  <service>
                <provide interface="org.eclipse.kura.configuration.ConfigurableComponent"/>
  </service>

  <property name="service.pid" value="fcoortbon.ble.central_hrp.HRPClient"/>

  <reference bind="setBluetoothService"
                cardinality="1..1"
                interface="org.eclipse.kura.bluetooth.BluetoothService"
                name="BluetoothService"
                policy="static"
                unbind="unsetBluetoothService"/>

  <reference bind="setDataService"
                cardinality="1..1"
                interface="org.eclipse.kura.data.DataService"
                name="DataService"
                policy="static"
                unbind="unsetDataService"/>
</scr:component>
```

[18] https://code.google.com/archive/p/json-simple/

Se han destacado en negrita aquellos campos más relevantes.

El elemento raíz es **`component`**, y contiene un elemento **`service`** por cada servicio que implementa y exporta, y un elemento **`reference`** por cada servicio que utiliza. Además, mediante el elemento **`implementation`** indicamos qué clase es nuestra clase componente, que implementará los métodos de activación y desactivación y los servicios exportados. En nuestro bundle, esta es la clase **`HRPClient`**.

En el elemento **`component`**, indicamos cuáles son los métodos de activación, desactivación y modificación (cuando se modifica la configuración del bundle, se llamará a este método) mediante los atributos **`activate`**, **`deactivate`** y **`modified`**, respectivamente. En nuestro caso, estos métodos son **`activate`**, **`deactivate`** y **`update`**.

En cada elemento `service`, indicamos mediante el elemento provide cuál es la interfaz del servicio que se implementa. En nuestro caso, sólo implementamos y exportamos el servicio **`ConfigurableComponent`**, cuya interfaz se encuentra en la clase **`org.eclipse.kura.configuration.ConfigurableComponent`**.

En cada elemento reference, indicamos:

- Mediante el atributo **`name`**: El nombre del servicio.
- Mediante el atributo **`interface`**: La interfaz del servicio.
- Mediante el atributo **`bind`**: El método de nuestra clase component llamado para obtener la referencia del servicio.
- Mediante el atributo **`unbind`**: El método de nuestra clase component llamado para eliminar la referencia del servicio.

Como vemos, en nuestro fichero component.xml hay dos elementos `reference` que indican que nuestro bundle utiliza los servicios **`BluetoothService`** y **`DataService`**.

Visto esto, la clase **`HRPClient`** debe:

- Implementar la interfaz `ConfigurableComponent`.
- Implementar los métodos `activate`, `deactivate` y `update`.
- Implementar los métodos `setBluetoothService` y `unsetBluetoothService`.
- Implementar los métodos `setDataService` y `unsetDataService`.

5.3 Servicios implementados: ConfigurableComponent

Como ya hemos visto, nuestro bundle implementa y exporta un único servicio, que es el servicio `ConfigurableComponent`.

Podemos encontrar un tutorial de Eclipse Kura donde se explica cómo crear un bundle que implementa la interfaz ConfigurableComponent aquí: https://eclipse.github.io/kura/dev/configurable_component.html

En Eclipse Kura, al implementar la interfaz `ConfigurableComponent`, damos la posibilidad a nuestro bundle de exponer su configuración a través del servicio de configuración **`ConfigurationService`**.

El servicio de configuración monitoriza todos los componentes que implementan la interfaz `ConfigurableComponent`, y cuando un nuevo componente configurable se registra, el servicio de configuración llama a su método de actualización (recordemos, atributo `modified` en el fichero component.xml) con la última configuración guardada para este componente, ó, en caso de no haberla, con la configuración por defecto.

Además, el servicio de configuración `ConfigurationService` asume que para cada componente configurable existe un recurso xml con metainformación acerca de su configuración en la carpeta OSGI-INF/metatype/ del componente y con el nombre del componente (nombre de la clase componente con el paquete incluyendo el paquete en el que se encuentra) [18].

Es decir, para nuestro bundle HRPClient es necesario el fichero **fcoortbon.ble.central_hrp.HRPClient.xml**.

Una vez que el componente configurable está activo en el dispositivo Kura, si se ha configurado todo correctamente, podremos ver un nuevo apartado correspondiente a nuestro bundle en la interfaz web de Kura, a través del cual podremos actualizar el valor de los parámetros que hayamos indicado en el fichero xml de metainformación.

5.3.1 Configuración del bundle HRPClient

Nuestro bundle tiene los siguientes parámetros de configuración:

- **Código de gimnasio**

  Este parámetro indica el código del gimnasio en el que se encuentra el dispositivo.

- **Número del dispositivo**

  Este parámetro indica el número asignado al dispositivo dentro del gimnasio.

  Recordemos que cada gimnasio en nuestro sistema se identifica con un código, y cada uno tiene una identidad (certificado y clave privada). Sin embargo, dentro de un gimnasio puede haber varios dispositivos Kura, y por lo tanto varios bundles HRPClient activos. Entonces, un dispositivo Kura se identifica de manera universal mediante su número (local al gimnasio en el que se encuentra) y el código del gimnasio.

- **Empezar/Terminar sesión**

  El bundle se puede controlar de manera remota mediante órdenes para finalizar o comenzar sesión. Adicionalmente, mediante el uso de este parámetro se permite también empezar y acabar sesiones: Si se establece su valor a verdadero, se inicia una sesión, y si se establece a falso, se acaba.

- **Tiempo de escaneo**

  Indica la duración en segundos de cada escaneo de dispositivos BLE. Recordemos que el bundle cuando está en sesión realiza escaneos de dispositivos BLE de forma periódica.

- **Periodo**

  Indica el tiempo en segundos entre inicios de escaneos. Evidentemente este tiempo debe ser mayor que el tiempo de escaneo.

- **Adaptador bluetooth**

  Indica el nombre de la interfaz bluetooth del dispositivo en el que se encuentra el bundle.

El valor de estos parámetros se puede cambiar desde la interfaz web. Para ello es necesario añadirlos al fichero fcoortbon.ble.central_hrp.HRPClient.xml.

5.3.2 Fichero fcoortbon.ble.central_hrp.HRPClient.xml

El fichero xml de metainformación para el servicio de configuración sigue la especificación de **OSGi Metatype**[19] y consiste de un elemento **`MetaData`** que debe contener un elemento **`Designate`**, y adicionalmente más elementos de diversos tipos. En nuestro caso elementos **`OCD`** y **`AD`** [19].

Un elemento `AD` (Attribute Definition) describe un atributo. Es decir, un parámetro. Para cada parámetro configurable de los que hemos visto antes existe un elemento `AD`.

Un elemento `OCD` (Object Class Definition) contiene información sobre un conjunto de atributos y varios elementos `AD`.

Un elemento `Designate` indica una asociación entre un elemento `OCD` y un pid (en nuestro caso referencia a la clase componente).

A continuación, se muestra el contenido del fichero fcoortbon.ble.central_hrp.HRPClient.xml:

[19] https://osgi.org/xmlns/metatype/v1.0.0/metatype.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaData xmlns="http://www.osgi.org/xmlns/metatype/v1.2.0" localization="es_es">

   <OCD id="fcoortbon.ble.central_hrp.HRPClient"
      name="HRPClient"
      description="Cliente BLE para el servicio de pulso cardíaco">

      <AD id="codigoGim"
         name="Código del gimnasio"
         type="String"
         cardinality="0"
         required="true"
         default="123456789012"
         description="Código que identifica al gimnasio."/>
      <AD id="numDisp"
         name="Número del dispositivo."
         type="Integer"
         cardinality="0"
         required="true"
         default="1"
         description="Número del dispositivo. Este número diferencia a dispositivos distintos dentro de un
mismo gimnasio."/>
      <AD id="client_enable"
         name="Empezar/Terminar sesión"
         type="Boolean"
         cardinality="0"
         required="true"
         default="false"
         description="Comenzar una nueva sesión de lectura de datos de dispositivos BLE.
         Para terminar la sesión, establecer esta propiedad a false."/>
      <AD id="scan_time"
         name="Tiempo de escaneo"
         type="Integer"
         cardinality="0"
         required="true"
         default="3"
         min="2"
         max="10"
         description="Duración en segundos de cada escaneo de dispositivos BLE. Durante una sesión
         se realizan estos escaneos de forma periódica para descubrir sensores de pulso cardíaco."/>
      <AD id="period"
         name="Periodo"
         type="Integer"
         cardinality="0"
         required="true"
         default="10"
         min="10"
         max="60"
         description="Tiempo en segundos entre inicios de escaneo. Debe ser mayor o igual que la duración
del escaneo."/>
      <AD id="iname"
         name="Adaptador bluetooth"
         type="String"
         cardinality="0"
         required="true"
         default="hci0"
         description="Nombre de la interfaz del adaptador bluetooth del dispositivo."/>
   </OCD>

   <Designate pid="fcoortbon.ble.central_hrp.HRPClient">
      <Object ocdref="fcoortbon.ble.central_hrp.HRPClient"/>
   </Designate>
</MetaData>
```

Como vemos, cada elemento `AD` tiene un id, y recordemos que cuando cambiamos el valor de alguno de estos parámetros se invoca al método `update` de `HRPClient`. Bien, pues al llamarse al método de actualización, se le pasa como parámetro un mapa con parejas atributo-valor, y cada atributo es identificado por su id. Es decir, nuestra clase HRPClient debe conocer los id de cada parámetro para poder leer los nuevos valores.

A continuación se muestra el código de la clase `HRPClient` relacionado con esto.

- Declaración de constantes con los id de cada parámetro configurable (elementos `AD`).

```
private final String PROPERTY_CODIGO_GIMNASIO = "codigoGim";
private final String PROPERTY_NUM_DISPOSITIVO = "numDisp";
private final String PROPERTY_ENABLE = "client_enable";
private final String PROPERTY_SCANTIME = "scan_time";
private final String PROPERTY_PERIOD = "period";
private final String PROPERTY_INAME = "iname";
```

- Variables para cada parámetro. Inicializadas con valor por defecto.

```
private String codigoGim = "";
private int numDisp = 1;
private boolean client_enable = false;
private int scantime = 3;
private int period = 10;
private String iname = "hci0";
```

- Método de actualización.

```
protected void activate(ComponentContext context, Map<String, Object>
properties)
{
        …

        readProperties(properties);

        …
}
```

- Método auxiliar readProperties para leer los nuevos valores de los parámetros configurables y asignarlos a las variables.

```
private void readProperties(Map<String, Object> properties)
{
  if (properties != null)
  {
    if (properties.get(this.PROPERTY_CODIGO_GIMNASIO) != null)
      this.codigoGim = (String) properties.get(this.PROPERTY_CODIGO_GIMNASIO);
    if (properties.get(this.PROPERTY_NUM_DISPOSITIVO) != null)
      this.numDisp = (Integer) properties.get(this.PROPERTY_NUM_DISPOSITIVO);
    if (properties.get(this.PROPERTY_ENABLE) != null)
      this.client_enable = (Boolean) properties.get(this.PROPERTY_ENABLE);
    if (properties.get(this.PROPERTY_SCANTIME) != null)
      this.scantime = (Integer) properties.get(this.PROPERTY_SCANTIME);
    if (properties.get(this.PROPERTY_PERIOD) != null)
      this.period = (Integer) properties.get(this.PROPERTY_PERIOD);
    if (properties.get(this.PROPERTY_INAME) != null)
      this.iname = (String) properties.get(this.PROPERTY_INAME);

  }

  …
}
```

La figura 26 muestra una captura del apartado de nuestro bundle HRPClient en la interfaz web de Kura, que aparece gracias a la implementación de la interfaz `ConfigurableComponent` y con el contenido indicado en el fichero fcoortbon.ble.central_hrp.HRPClient.xml.

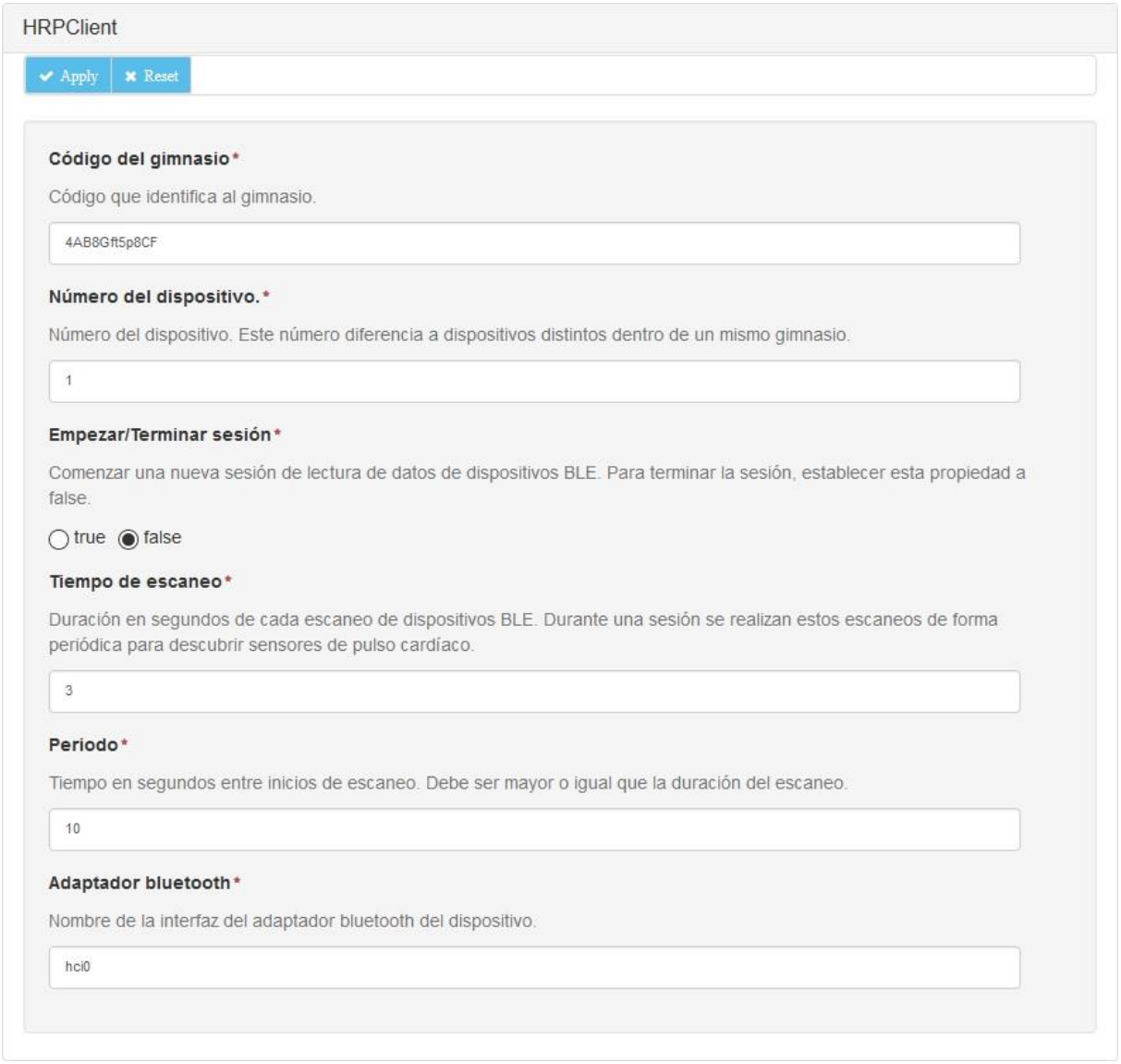

Figura 26: Apartado HRPClient en la interfaz web de Kura.

5.4 Servicios utilizados

Nuestro bundle utiliza dos servicios que vienen ya implementados por defecto al instalar Eclipse Kura en un dispositivo, por lo que no tenemos que preocuparnos nosotros por hacer una implementación de ellos.

Estos son un servicio para el uso de Bluetooth (`BluetoothService`) y otro para la publicación de datos a la nube y demás operaciones relacionadas con esta y el protocolo MQTT (`DataService`).

5.4.1 BluetoothService

Mediante el uso de este servicio podemos obtener un objeto de tipo **`BluetoothAdapter`**, que representa una

interfaz bluetooth del dispositivo. Además, se obtiene a partir del nombre de interfaz que proporcionemos, y en nuestro caso recordemos que este era un párametro configurable, con "hci0" como valor por defecto.

Una vez tenemos nuestro objeto `BluetoothAdapter`, podemos llamar a sus métodos para habilitarlo o deshabilitarlo, comenzar y parar un escaneo, etc. Podemos encontrar la documentación de la clase `BluetoothAdapter` aquí:
http://download.eclipse.org/kura/docs/api/2.1.0/apidocs/org/eclipse/kura/bluetooth/BluetoothAdapter.html

A continuación se muestran fragmentos de código de la clase `HRPClient` relacionado con esto.

- Declaración de variables para guardar la referencia al servicio bluetooth y al objeto BluetoothAdapter.

```
private BluetoothService bluetoothService;
private BluetoothAdapter bluetoothAdapter;
```

- Métodos `setBluetoothService` y `unsetBluetoothService`.

```
private void setBluetoothService (BluetoothService bluetoothService)
{
  this.bluetoothService = bluetoothService;
}

private void setBluetoothService (BluetoothService bluetoothService)
{
  this.bluetoothService = null;
}
```

- Método `start`, llamado al empezar a funcionar (nueva sesión).

```
private void start()
{
  …

  // Obtener el adaptador bluetooth y comprobar que está activado
  this.bluetoothAdapter = this.bluetoothService.getBluetoothAdapter(this.iname);
  if (this.bluetoothAdapter != null)
  {
    if (!this.bluetoothAdapter.isEnabled())
    {
      logger.info("Habilitando adaptador...");
      this.bluetoothAdapter.enable();
    }
    …
  }
  …
}
```

- Método `checkScan`, ejecutado de forma periódica cuando el bundle está en sesión.

```
void checkScan()
{
  if (this.bluetoothAdapter.isScanning())
  {
    …
  }
  else
  {
    // Empezamos a escanear si, desde el último escaneo, ha pasado tiempo igual
    // o mayor al periodo entre escaneos.
    if (System.currentTimeMillis() - this.startTime >= this.period * 1000)
    {
      this.bluetoothAdapter.startLeScan(this);
      this.startTime = System.currentTimeMillis();
    }
  }
}
```

Como hemos visto, realizamos y finalizamos los escaneos usando métodos del objeto BluetoothAdapter, pero cuándo finaliza el escaneo, ¿dónde se reciben los resultados de este?

Bien, para ello la clase HRPClient implementa la interfaz `BluetoothLeScanListener`. La documentación de esta interfaz la podemos encontrar aquí:
http://download.eclipse.org/kura/docs/api/2.1.0/apidocs/org/eclipse/kura/bluetooth/BluetoothLeScanListener.html

Sólo contiene dos métodos:

- `void onScanFailed(int errorCode)`
  Método llamado cuando se produce un error al realizar un escaneo Bluetooth LE.

- `void onScanResults(List<BluetoothDevice> devices)`
  Método llamado cuando finaliza un escaneo Bluetooth LE. Recibe como parámetro una lista de dispositivos BLE, que son los que se han descubierto durante el escaneo.

Los cuales sobreescribimos en nuestra clase `HRPClient` para manejar estos eventos como queramos.

A continuación se muestra un fragmento del código de estos métodos:

```
@Override
public void onScanFailed(int errorCode)
{
  logger.error("Error durante el escaneo. Codigo: " + errorCode);
}

@Override
public void onScanResults(List<BluetoothDevice> scanResults)
{
  …

  for (BluetoothDevice bluetoothDevice : scanResults)
  {
    String address = bluetoothDevice.getAdress();
    String name = bluetoothDevice.getName();
    logger.info("Encontrado dispositivo " + address + " Name " + name);

    …
  }
  …
}
```

5.4.2 DataService

Mediante el uso de este servicio podemos conectarnos a la nube, suscribirnos a topics, publicar y recibir mensajes, etc. Recordemos que por defecto Eclipse Kura trae implementado un **`Cloud Service`** (como ya comentamos anteriormente, `CloudService` y `DataService` sirven para lo mismo, pero hay algunas diferencias a la hora de operar con ellos) y debemos configurarlo previamente en la interfaz web de Kura.

Al obtener la referencia al servicio a través del método `setDataService`, podemos comenzar a llamar a los métodos de nuestro objeto `DataService` para conectarnos a la nube, suscribirnos a un topic, publicar mensajes en un topic…

La documentación de la interfaz `DataService` se puede encontrar aquí:
http://download.eclipse.org/kura/docs/api/2.1.0/apidocs/org/eclipse/kura/data/DataService.html

Además, para recibir notificaciones de eventos relacionados con la conexión a la nube (conexión establecida, llegada de un nuevo mensaje…) debemos implementar la interfaz `DataServiceListener` e indicar al servicio `DataService` que nuestra clase quiere recibir estas notificaciones mediante el método `addDataServiceListener`. La documentación de la interfaz `DataServiceListener` se puede encontrar aquí: http://download.eclipse.org/kura/docs/api/2.1.0/apidocs/org/eclipse/kura/data/DataService.html

La conexión con la nube se ha configurado en la interfaz web para que se haga de manera automática.

A continuación se muestran algunos fragmentos de código relacionados con el servicio.

- Declaración de la variable para guardar la referencia al servicio DataService.

```
private DataService dataService;
```

- Métodos `setDataService` y `unsetDataService`.

```
private void setDataService (DataService dataService)
{
  this.dataService = dataService;
}

private void unsetDataService (DataService dataService)
{
  this.dataService = null;
}
```

- Llamada a `addDataServiceListener` al activar el bundle.

```
protected void activate(ComponentContext context, Map<String, Object>
properties)
{
  …

  dataService.addDataServiceListener(this);

  …
}
```

- Suscripción al topic de control cuando se establece conexión con la nube.

```
@Override
public void onConnectionEstablished()
{
  suscribirTopic(controlTopic);
}
```

- Métodos auxiliares para suscribirse a un topic y para publicar datos en un topic.

```
private void suscribirTopic(String topic)
{
  int qos = 1;
  try
  {
    logger.info("Suscribir a "+ topic + "...");
    dataService.subscribe(topic, qos);
  }
  catch (…) {
    …
  }
}

protected static void publicarDatos(String topic, int pulso, int energia)
{
  …
  JSONObject datos = new JSONObject();
  if (energia != -1)
    datos.put("energia", energia);
  datos.put("pulso", pulso);

  …

  byte[] payload = datos.toJSONString().getBytes();

  if (publish(topic, payload))
    logger.info("Datos publicados!");
}
```

- Método `onMessageArrived` para manejar los mensajes de órdenes recibidos desde la nube.

```
@Override
public void onMessageArrived(String topic, byte[] payload, int qos,
                    boolean retained)
{
  JSONParser parser = new JSONParser();

  if (topic.equals(controlTopic))
  {
    try
    {
      JSONObject json = (JSONObject) parser.parse(new String(payload));
      int action = (int) (long) json.get("action");

      switch (action)
      {
        case ControlHRPClient.COMENZAR_SESION:
          …
          break;
        case ControlHRPClient.TERMINAR_SESION:
          …
          break;
        case ControlHRPClient.PUBLICAR_ESTADO:
          …
          break;
        default:
          …
          break;
      }
    }
    catch(…) {
      …
    }
  }
}
```

5.5 Clases

El bundle se compone de las siguientes clases:

- **HRPControlClient**

  Esta clase contiene únicamente una serie de constantes que identifican las distintas órdenes que se pueden enviar al bundle y los estados en los que se puede encontrar.

- **HRPGatt**

  Esta clase contiene sólo una serie de constantes para los UUID y los handle del servicio de pulso cardíaco.

- **HRPDevice**

  Esta clase representa a un dispositivo BLE real que se ha descubierto. Proporciona métodos para interactuar con él y es quién recibe los datos enviados (notificaciones) por el dispositivo BLE al que está asociado. Usa las constantes definidas en la clase HRPGatt.

- **HRPClient**

  Esta es la clase principal del bundle. Es la encargada de recibir órdenes desde la nube y ejecutarlas (empezar o acabar una sesión, publicar el estado), de realizar los escaneos Bluetooth LE y de controlar a los dispositivos HRPDevice. Además proporciona un método para publicar datos en la nube que es llamado por los HRPDevice cuando estos reciben información de los dispositivos BLE. Usa las constantes definidas en la clase HRPControlClient.

La figura 27 muestra el diagrama de clases del bundle.

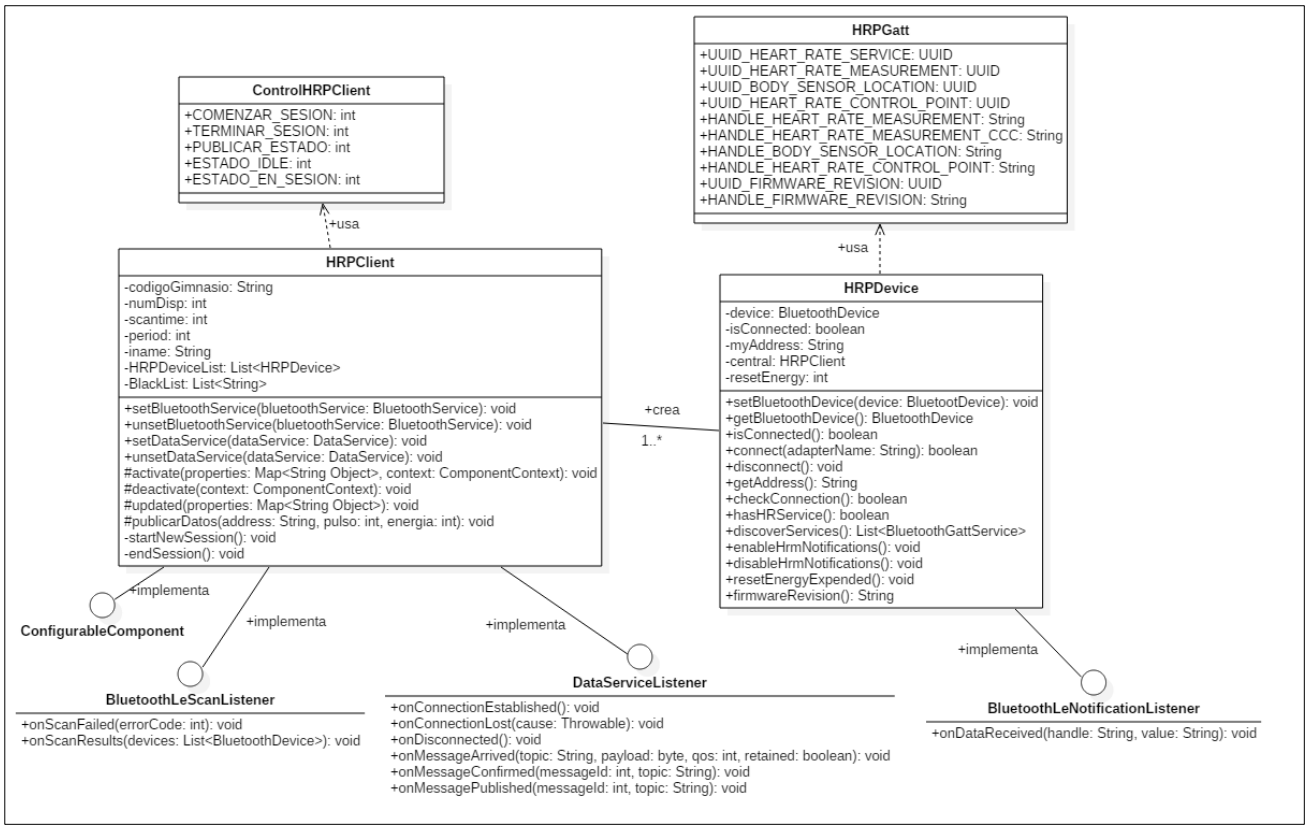

Figura 27: Diagrama de clases del bundle HRPClient.

5.6 Funcionamiento

5.6.1 Inicio del bundle y estado idle

Como ya se ha comentado anteriormente, el bundle se activa al iniciarse el dispositivo Kura pero se mantiene en estado idle, a la espera de órdenes. Para activar el bundle, el método `activate` de la clase `HRPClient` es llamado.

Método `activate`

- Procesa el valor de los parámetros de configuración.
- Crea la lista para los dispositivos BLE de pulso con los que nos conectamos e interactuamos.
- Crea un mapa para apuntar los intentos de conexión fallidos para cada dispositivo (dirección MAC).
- Crea la lista negra[20].
- Intenta conectarse a la nube si el bundle no está conectado ya.
- Establece el estado del bundle a `ESTADO_IDLE`.
- Añade la instancia de la clase HRPClient a los "listeners" del servicio DataService.

[20] En la lista negra se guardan las direcciones de los dispositivos a los cuales el bundle se ha intentado conectar sin éxito 3 veces seguidas y también las direcciones de aquellos que no ofrecen el servicio de pulso cardíaco. Cuando se reciben los resultados de los escaneos Bluetooth LE, se ignoran aquellas direcciones que están en la lista negra.

Al recibir la notificación de que se ha establecido una conexión con la nube, se suscribe al topic de control (recordemos, esto ocurre en el método `onConnectionEstablished`). De esta forma, el bundle está en estado idle pero recibe las órdenes que se le envíen a través de la nube.

A partir de aquí, el bundle actuará en consecuencia cada vez que reciba un nuevo mensaje con una orden (recordemos, esto ocurre en el método `onMessageArrived`).

A continuación, la figura 28 muestra un diagrama de flujo que explica la lógica del método `onMessageArrived`.

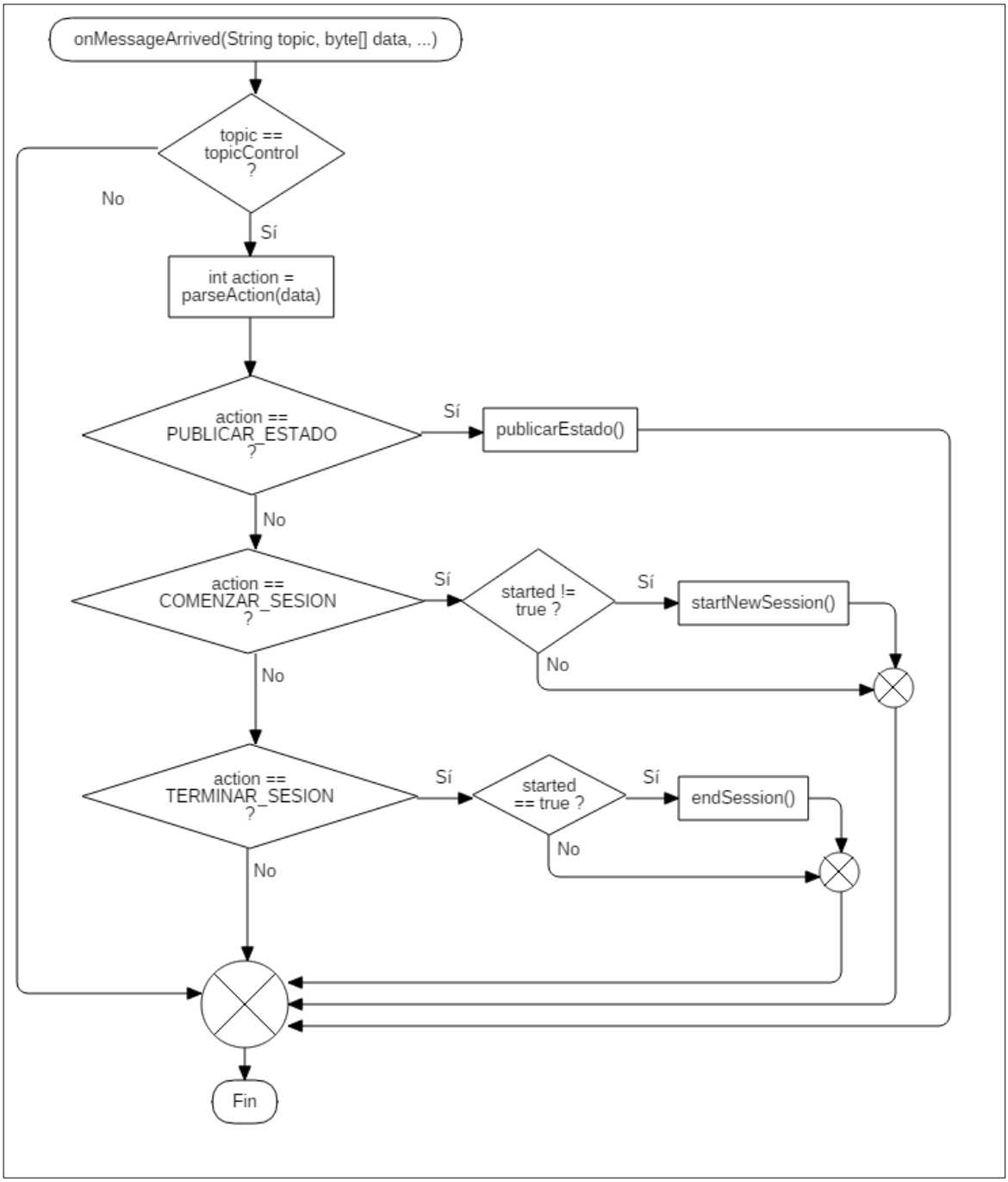

Figura 28: Diagrama de flujo del método onMessageArrived.

5.6.2 Nueva sesión en el bundle

A continuación, la figura 29 muestra un diagrama de secuencia con los mensajes intercambiados al iniciar una nueva sesión en el bundle (se encuentra en estado idle).

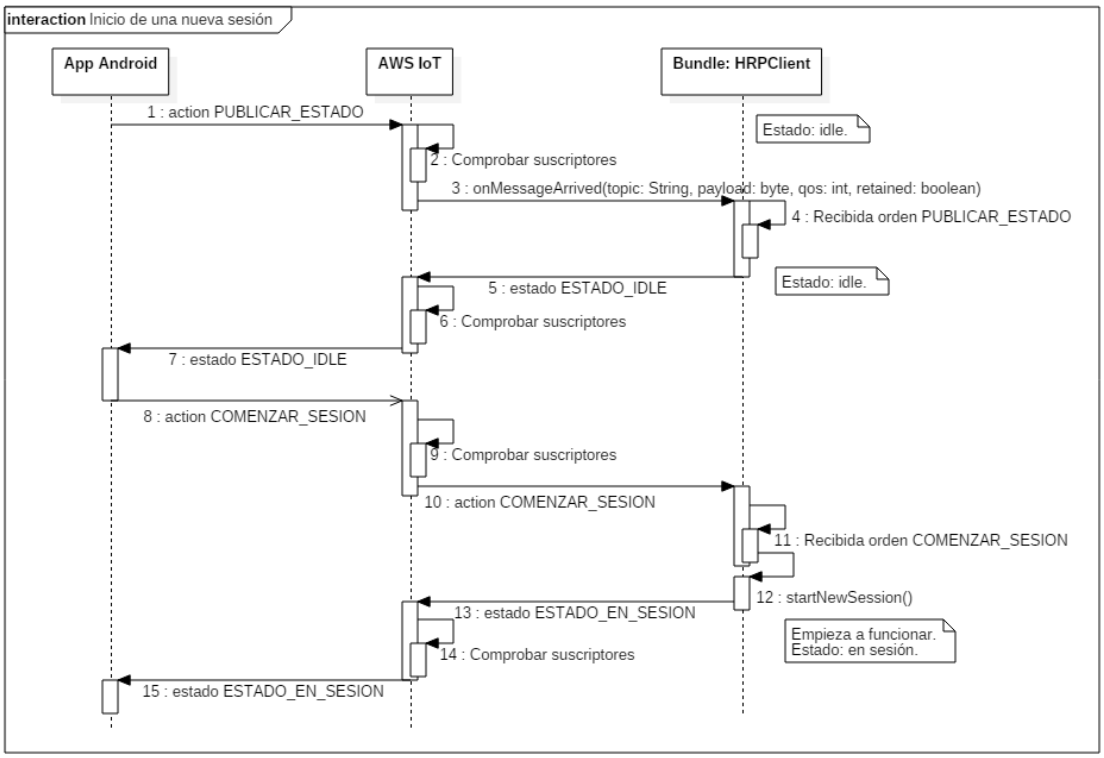

Figura 29: Diagrama de secuencia del inicio de una nueva sesión en el bundle.

Como se observa en el diagrama anterior, el método llamado para poner a funcionar el bundle es `startNewSession`. A continuación se muestra el código de este método.

```
private void startNewSession() throws ComponentException
{
  inicioSesion = new Date();

  logger.info("Cliente HRP: Comenzando nueva sesion en dispositivo "+numDisp);

  start();

  if (started)
  {
    String _inicioSesion = formatoFecha.format(inicioSesion);
    actualizarEstado(ControlHRPClient.ESTADO_EN_SESION, _inicioSesion, "");
  }
  else
  {
    logger.error("Error al iniciar nueva sesion");
  }
}
```

Como vemos, este método a su vez llama al método `start`, y comprueba a través de la variable `started` si se ha conseguido poner a funcionar el bundle de manera correcta o no.

A continuación, la figura 30 muestra un diagrama de flujo del método `start` para explicar su lógica y cómo pone a funcionar el bundle.

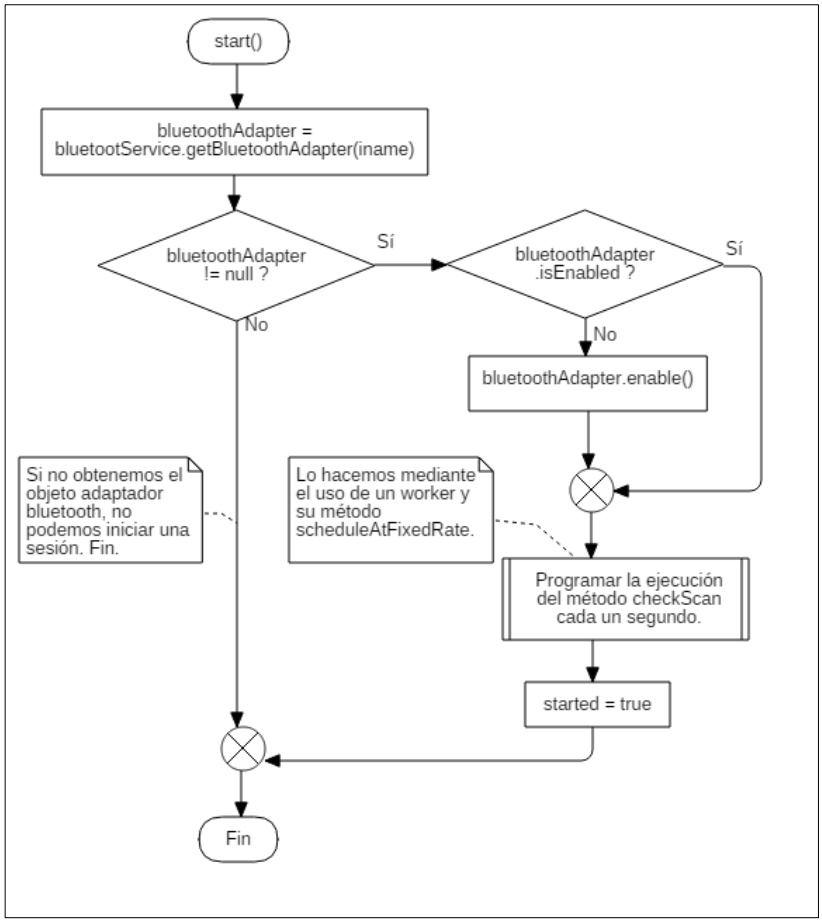

Figura 30: Diagrama de flujo del método start.

Como vemos en el diagrama anterior, el funcionamiento del bundle consiste en que el método checkScan se llame cada segundo. Este método es muy simple, lo único que hace es comprobar si se está llevando a cabo un escaneo o no:

- En caso verdadero, el escaneo se parará si se lleva escaneando más tiempo que el indicado por el parámetro `scantime` (en segundos).
- En caso contrario, se deberá comenzar un nuevo escaneo si desde el inicio del último escaneo ha pasado ya el tiempo indicado por el parámetro `period` (en segundos).

La clave ahora está en que cada vez que se realiza un escaneo, al acabarlo se llama al método `onScanResults` y se le pasa una lista con los dispositivos BLE descubiertos.

Método onScanResults

- Recorre la lista de dispositivos BLE descubiertos. Por cada dispositivo:

    1. Se comprueba que no está en la lista negra.

    2. Se comprueba que no está ya en nuestra lista `HRPDeviceList`: Recordemos que un dispositivo BLE no se anuncia si está conectado con otro dispositivo, así que si ya estaba en nuestra lista es porque se desconectó o rechazó la conexión anteriormente.

    3. Se añade a la lista `HRPDeviceList` si se cumplen las condiciones de los puntos 1 y 2.

- Recorre la lista `HRPDeviceList`. Por cada dispositivo:

1. Se intenta conectar al dispositivo si no está conectado ya a él.
2. Una vez se conecta, comprueba si el dispositivo ofrece el servicio de pulso cardíaco.
    - En caso de ofrecerlo, activa las notificaciones para la característica Heart Rate Measurement.
    - En caso de no ofrecerlo, se desconecta de él, se borra su entrada de la lista `HRPDeviceList` y se añade su dirección MAC a la lista negra.
3. En caso de haber fallado el intento de conexión, se incrementa en 1 el número de intentos de conexión fallidos seguidos ocurridos con ese dispositivo. Si el número de intentos resultante es igual a 3, se borra su entrada de la lista `HRPDeviceList` y se añade su dirección MAC a la lista negra.

Es decir, en un caso exitoso, el bundle consigue conectarse a un dispositivo BLE que ofrece el servicio de pulso cardíaco y activa las notificaciones para las medidas de pulso y energía. A partir de este momento, cada vez que el dispositivo disponga de una nueva medida de pulso, notificará al bundle.

Quién recibe la notificación es el objeto `HRPDevice` asociado al dispositivo BLE, mediante el método `onDataReceived`. Al recibir los datos, extrae la información de pulso y energía y llama al método `publicarDatos` del objeto `HRPClient` para publicarla en la nube.

A continuación, las figuras 31 y 32 muestran unos diagramas de secuencia que explican la interacción entre el bundle y los dispositivos BLE.

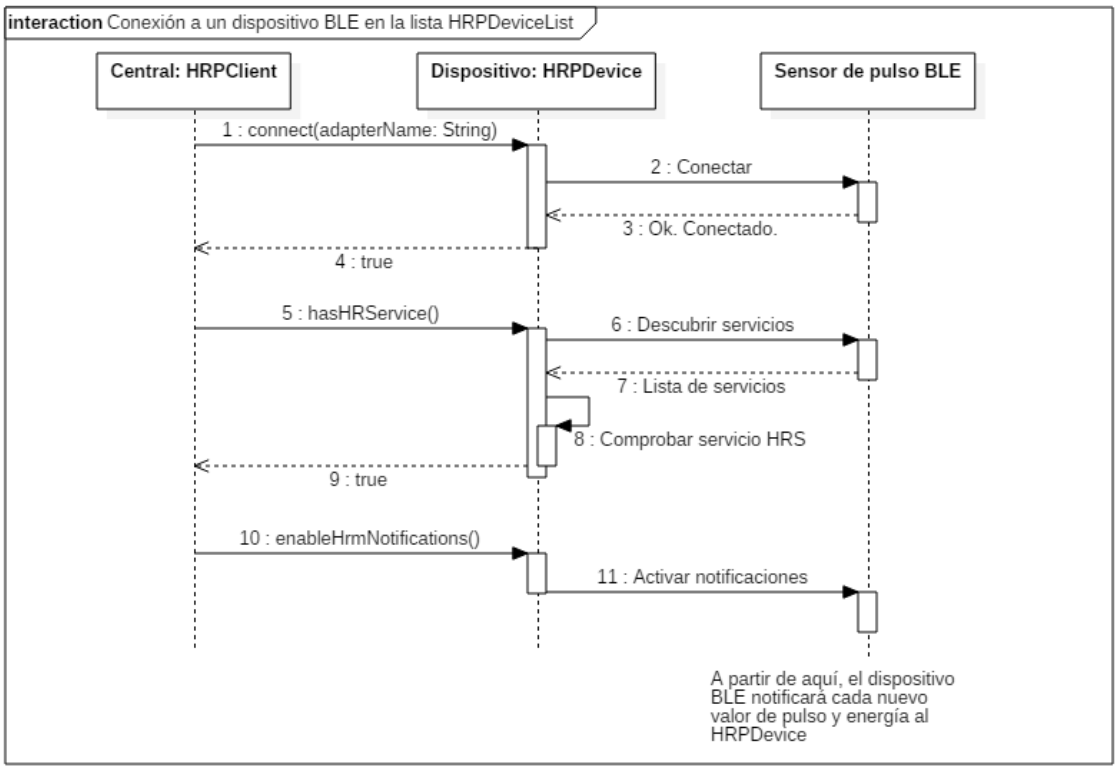

Figura 31: Diagrama de secuencia de una conexión con éxito a un dispositivo BLE que ofrece el servicio de pulso cardíaco.

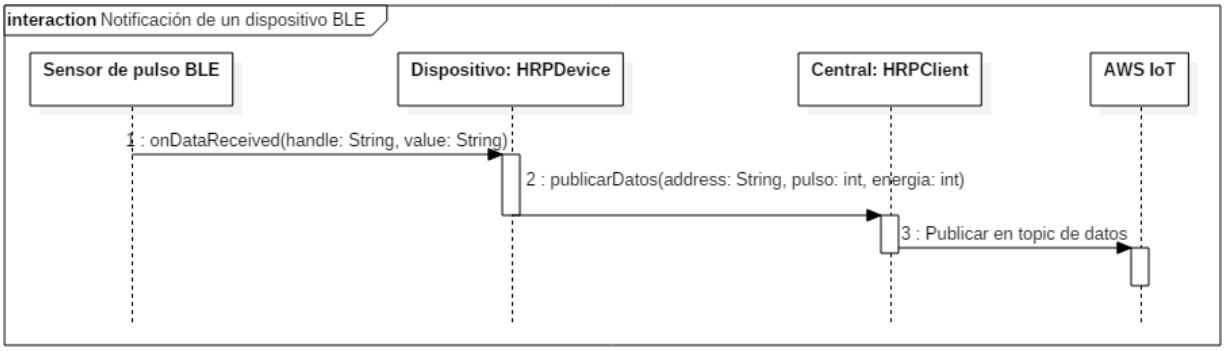

Figura 32: Diagrama de secuencia de la notificación de datos por parte de un dispositivo BLE y el envío de estos a la nube.

5.6.3 Fin de una sesión en el bundle

Al igual que para iniciar una nueva sesión en el bundle, para finalizarla se le debe enviar una orden, en este caso la de finalizar sesión.

A continuación, la figura 33 muestra un diagrama de secuencia con los mensajes intercambiados al finalizar una sesión en el bundle (suponemos que está en sesión, en caso de no estarlo simplemente ignorará la orden, ver Figura 28: Diagrama de flujo del método onMessageArrived.).

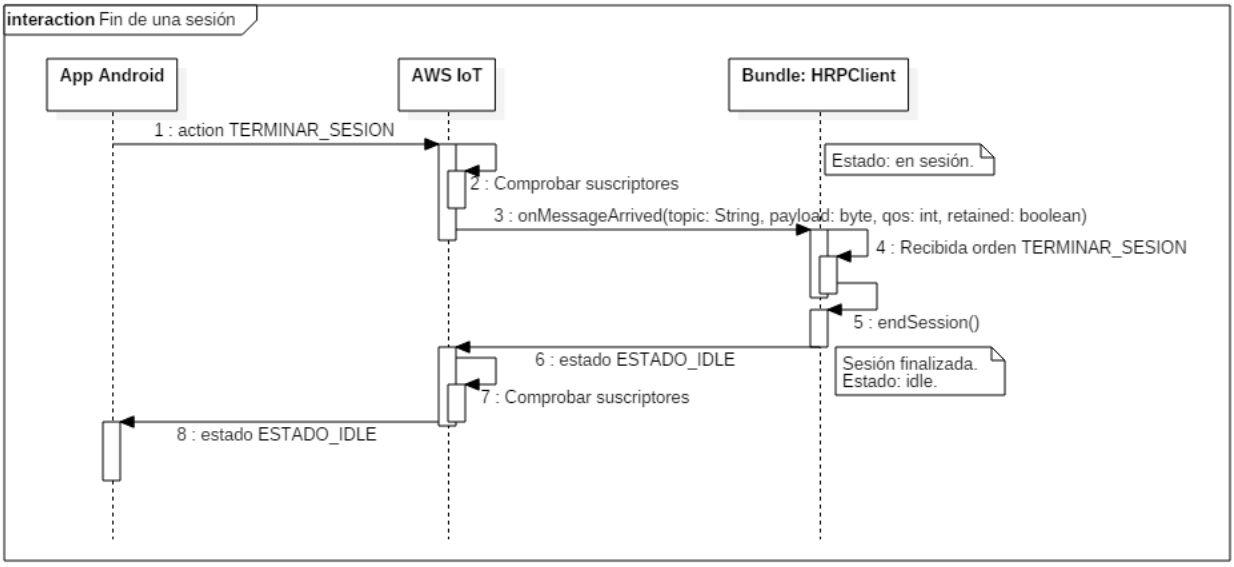

Figura 33: Diagrama de secuencia del inicio de una nueva sesión en el bundle.

El método `endSession` llama a su vez al método `stop`, que cancela los escaneos periódicos, se desconecta de todos los dispositivos BLE a los que estaba conectado y por último publica su nuevo estado: idle.

5.7 Despliegue del bundle en la Raspberry Pi

Una vez tenemos el bundle programado y listo para usar, podemos instalarlo en nuestro dispositivo Kura, que en este caso es una Raspberry Pi. Se puede hacer de varias formas, pero la que nos interesa es aquella que deja el bundle instalado en el dispositivo de forma permanente.

Podemos encontrar cómo hacerlo de cada manera en la documentación de Eclipse Kura. Aquí: https://eclipse.github.io/kura/dev/deploying-bundles.html

En nuestro caso, el apartado que nos interesa es **Making Deployment Permanent**. Previamente, debemos haber creado un paquete para el despliegue de nuestro bundle (Deployment Package).

5.7.1 Deployment Package

Un paquete de despliegue (Deployment Package) consiste en una serie de recursos agrupados en un único archivo de tipo paquete que puede ser desplegado en un framework OSGi a través del servicio 'Deployment Admin' (administrador de despliegue). Puede contener uno o más bundles, objetos de configuración,…

En nuestro caso, el paquete de despliegue contendrá nuestro bundle únicamente y es una tarea muy simple generarlo:

1. Creamos el JAR de nuestro bundle.
2. Creamos un fichero de tipo 'Deployment Package Definition' en la carpeta resources/dp/ de nuestro proyecto.
3. Indicamos en el fichero anterior los bundles que se van a empaquetar. En nuestro caso, sólo el JAR de nuestro bundle.
4. Guardamos los cambios en el fichero, hacemos clic derecho en él y seleccionamos 'Quick Build'.

Para ver los pasos con más detalle: https://eclipse.github.io/kura/dev/hello-example.html#deploying-the-plug-in

5.7.2 Despliegue permanente

Ahora, para desplegar el bundle en la Raspberry Pi de forma permanente, tenemos que:

1. Copiar el paquete de despliegue del bundle (central_hrp.dp) en la Raspberry Pi, en el directorio /opt/eclipse/kura/kura/packages
2. Editar el fichero dpa.properties (/opt/eclipse/kura/kura/dpa.properties) y añadir la siguiente línea:

```
central_hrp=file\:/opt/eclipse/kura/kura/packages/central_hrp.dp
```

Una vez hecho esto, en el próximo inicio de Kura el paquete se instalará y quedará en el dispositivo de forma permanente.

Nota

Al instalar Eclipse Kura en la Raspberry Pi, por defecto vienen incorporados y activados una serie de bundles. Para que funcione bien nuestro sistema, es preciso desactivar uno en concreto, `org.eclipse.kura.camel.xml_1.0.0`.

El motivo es que cuando nuestro dispositivo logra establecer conexión con la nube, este bundle intenta suscribirse a un topic, pero recordemos que en AWS IoT la política asociada al gimnasio sólo le permite suscribirse a sus topics de control y no a otros. Esto hace que cuando el bundle se intenta suscribir a un topic al que no tiene permiso se caiga la conexión con la nube.

Para evitar que este bundle se active de forma automática, debemos quitar la referencia a él en el fichero `/opt/eclipse/kura/kura/config.ini`

5.8 Ejemplo de ejecución

Para mostar el funcionamiento del bundle:

- Se van a usar un par de sensores emulados ejecutados en un portátil.
- Se va a usar una Raspberry Pi con el bundle instalado y configurada como el dispositivo número uno del gimnasio 4AB8Gft5p8CF. Esto quiere decir que:

- Su topic de control es:
  `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/4AB8Gft5p8CF _1/control`
- Su topic de estado es:
  `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/4AB8Gft5p8CF _1/satus`
- Su topic de datos es:
  `fcoortbon/centralhrp/GIM_4AB8Gft5p8CF/4AB8Gft5p8CF _1`

- Se va a usar el cliente MQTT de AWS IoT para enviar los mensajes de órdenes al topic de control del bundle y para suscribirse y ver los mensajes en los topics de estado y datos.

5.8.1 Bundle activado y en estado idle

En la Raspberry podemos:

- Conectarnos al framework OSGi.

```
> telnet localhost 5002
```

- Ver el log de Kura.

```
> tail –f /var/log/kura.log
```

Nos vamos a conectar mediante SSH a la Raspberry desde Windows, usando PuTTY[21].

En la figura 34 se muestra una captura del resultado de ejecutar el comando `ss` en el framework OSGi. Este comando lista los bundles instalados en el dispositivo.

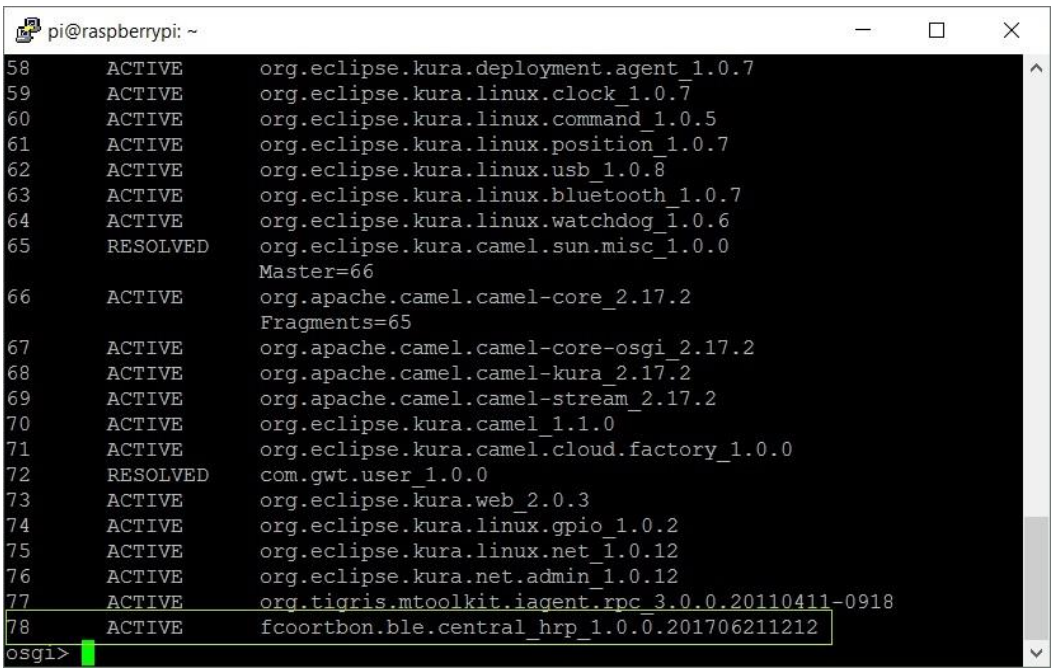

Figura 34: Salida del comando `ss` en el framework OSGi de la Raspberry Pi.

Observamos que aparece nuestro bundle y que se encuentra activo. Además, se encuentra en estado idle ya que no ha recibido aún la orden de empezar sesión.

[21] http://www.putty.org/

Desde el cliente MQTT de AWS IoT vamos a suscribirnos a los topics de estado y de datos del bundle, y además vamos a enviar la orden `PUBLICAR_ESTADO`.

Recordemos que el contenido de la clase `HRPControlClient` era:

```
public class ControlHRPClient
{
    public static final int COMENZAR_SESION = 1;
    public static final int TERMINAR_SESION = 2;
    public static final int PUBLICAR_ESTADO = 3;

    public static final int ESTADO_IDLE = 1;
    public static final int ESTADO_EN_SESION = 2;
}
```

A continuación, en las figuras 35 y 36 se muestran un par de capturas en las que podemos observar cómo desde el cliente MQTT de AWS IoT hemos enviado la orden `PUBLICAR_ESTADO` (3) al topic de control del bundle y cómo este ha contestado publicando su estado, `ESTADO_IDLE` (1).

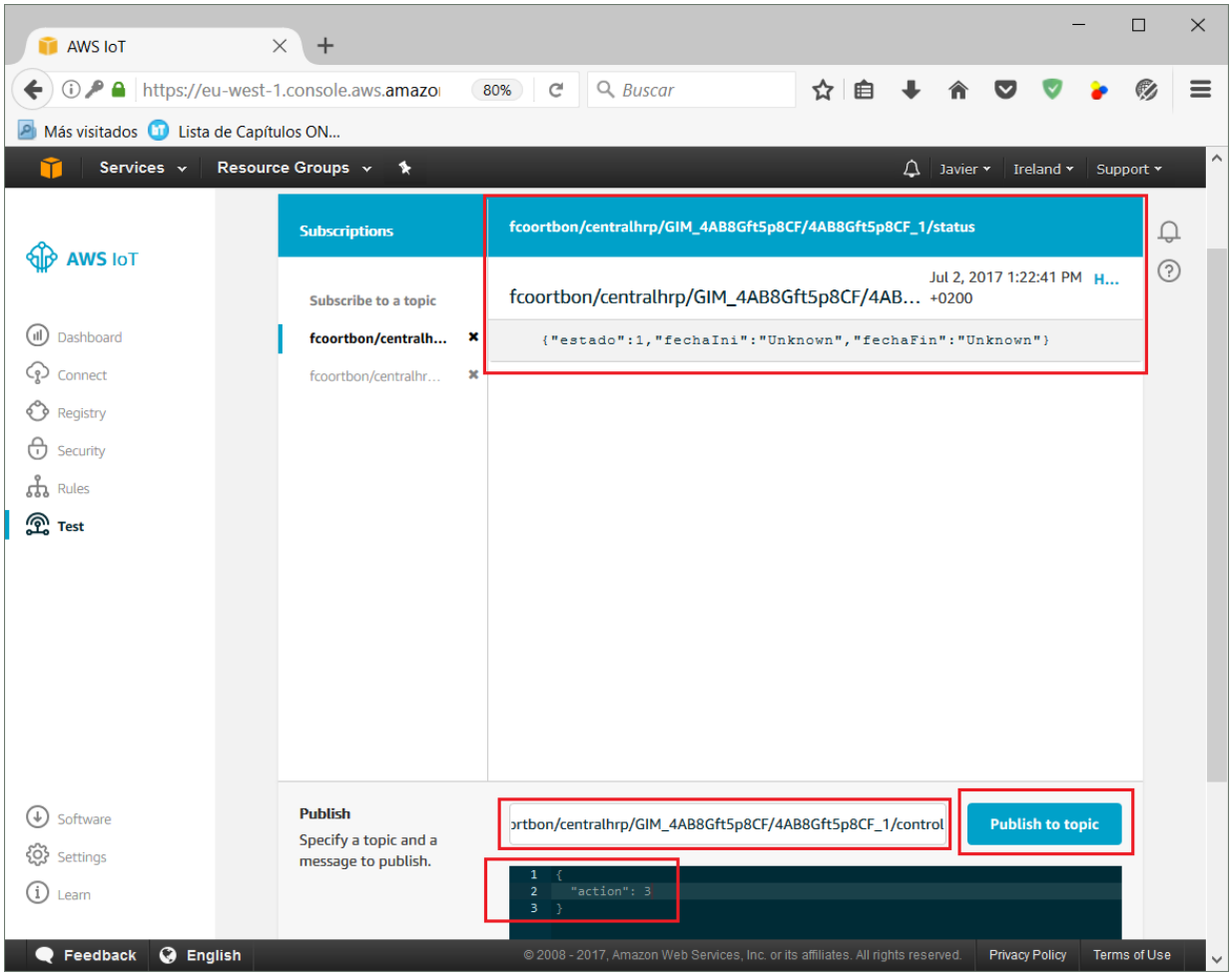

Figura 35: Cliente MQTT de AWS IoT: Envío de orden PUBLICAR_ESTADO y recepción del estado publicado por el bundle.

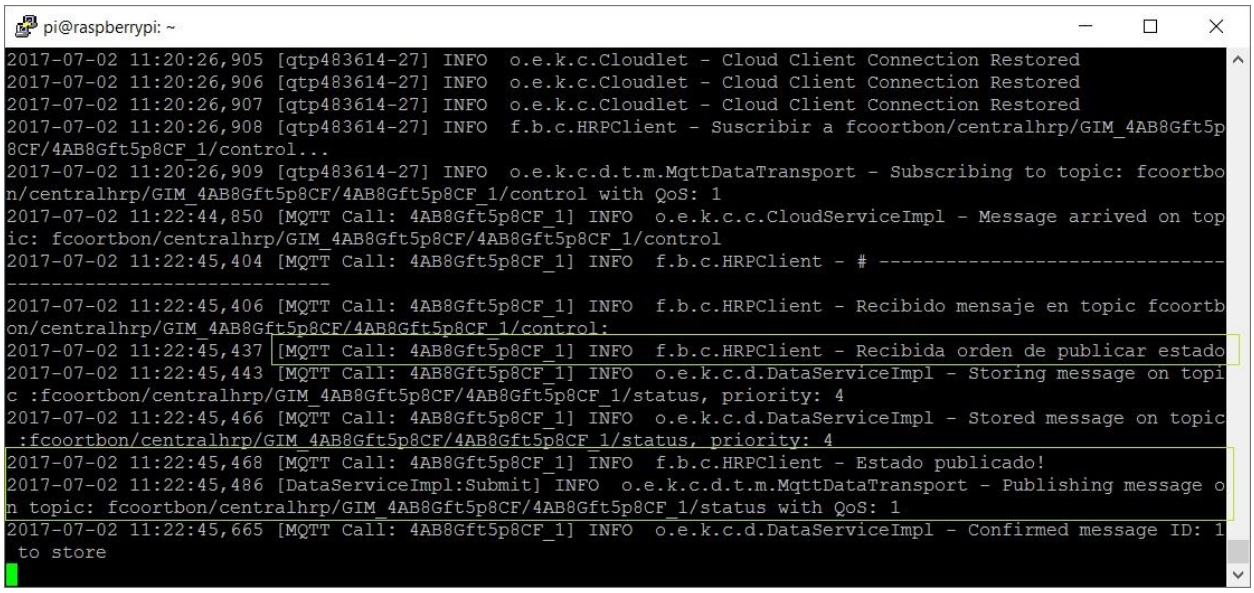

Figura 36: Logs del bundle HRPClient al recibir la orden PUBLICAR_ESTADO.

5.8.2 Inicio de una nueva sesión en el bundle

Ahora vamos a enviar desde el cliente MQTT de AWS IoT la orden `COMENZAR_SESION` (1) y vamos a suscribirnos al topic de datos y a ver los mensajes que publica el bundle.

Primero, desde Ubuntu vamos a poner a funcionar un par de sensores emulados. En la figura 37 se muestra una captura del arranque de los sensores.

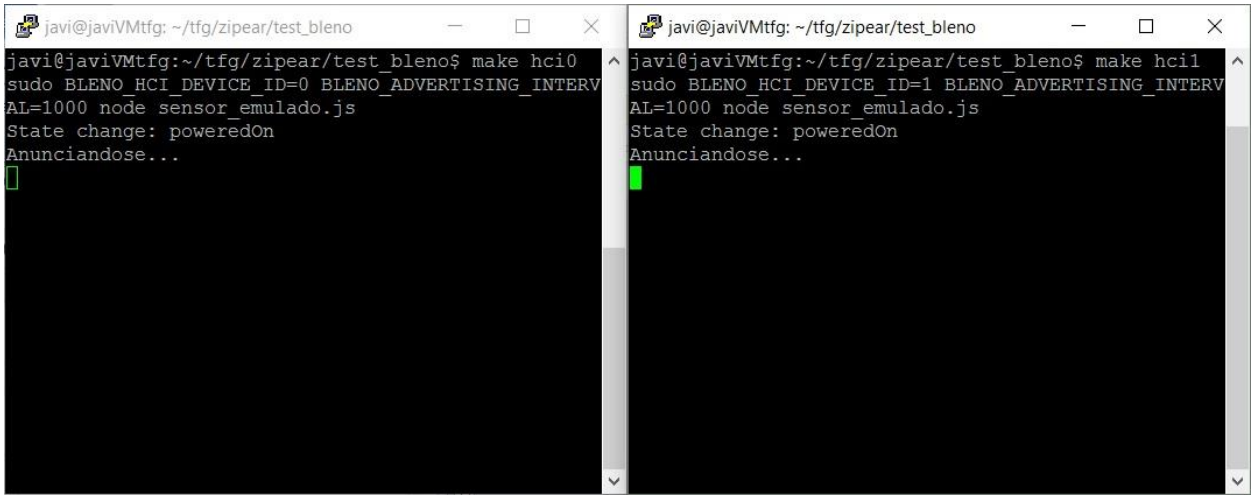

Figura 37: Sensores emulados de pulso en funcionamiento.

Desde el cliente MQTT de AWS IoT publicamos la orden `COMENZAR_SESION` (1) y observamos el topic de estado para ver cómo el bundle publica su estado `ESTADO_EN_SESION` (2) al comenzar a funcionar. En la figura 38 se muestra una captura del cliente MQTT de AWS IoT al realizar esto.

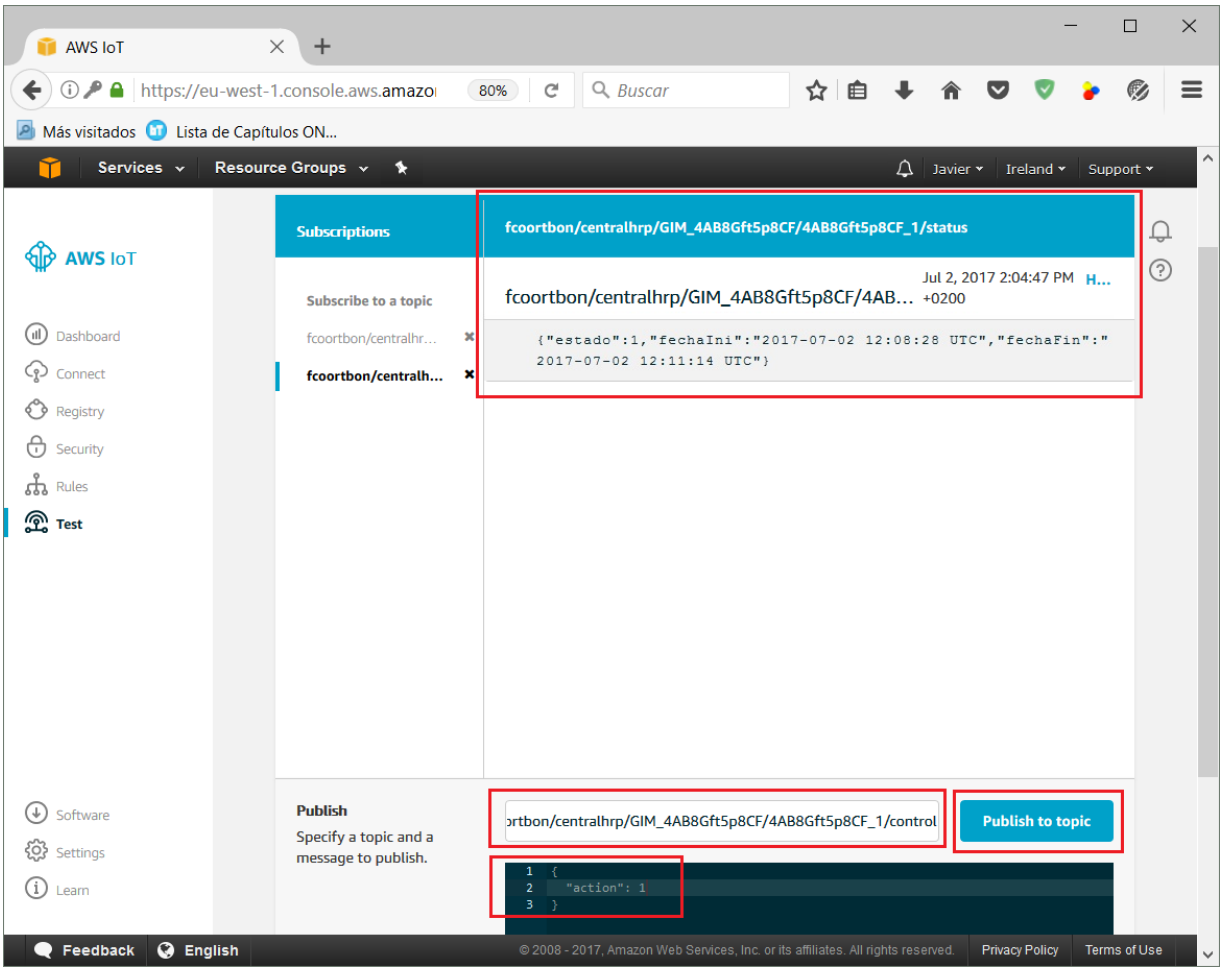

Figura 38: Cliente MQTT de AWS IoT: Envío de orden COMENZAR_SESION y recepción del estado publicado por el bundle.

Dejamos transcurrir unos 30 segundos para dar tiempo al bundle a descubrir a los sensores y a conectarse a ellos.

A continuación, en las figuras 39 y 40 se muestran un par de capturas en las que podemos observar cómo los sensores han recibido una conexión desde el bundle y la orden de activar las notificaciones de pulso cardíaco, y cómo se ha ido notificando al bundle con nuevos valores.

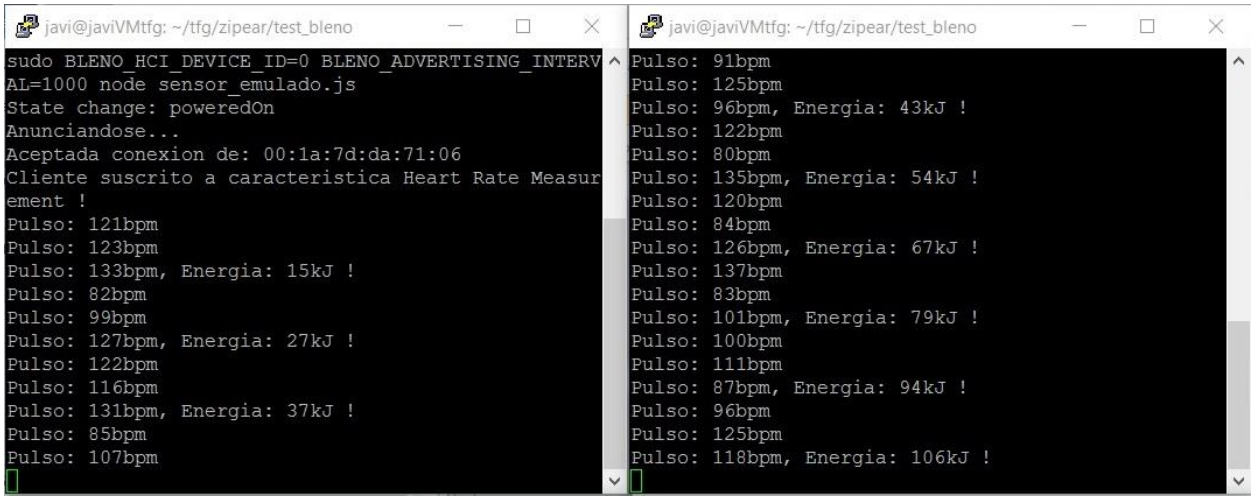

Figura 39: Sensores emulados enviando datos al bundle.

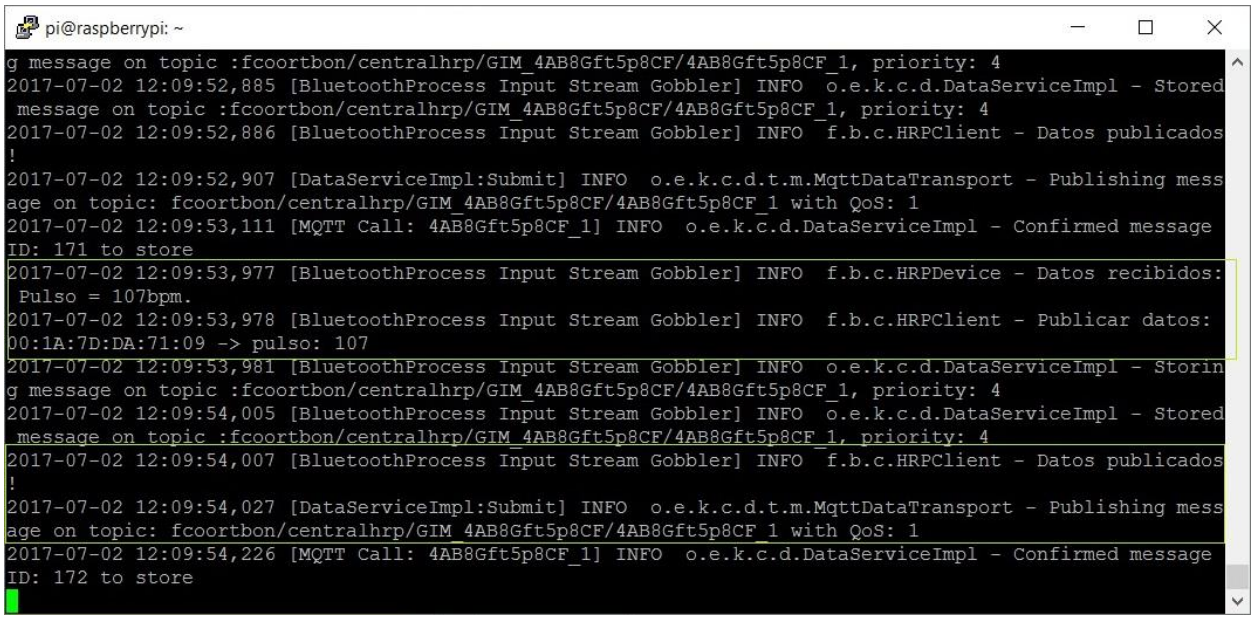

Figura 40: Logs del bundle HRPClient al recibir datos de los dispositivos BLE.

Desde el cliente MQTT de AWS IoT nos habíamos suscrito al topic de datos, en el que podemos ir viendo la información publicada por el bundle. En la figura 41 se muestra una captura de esto.

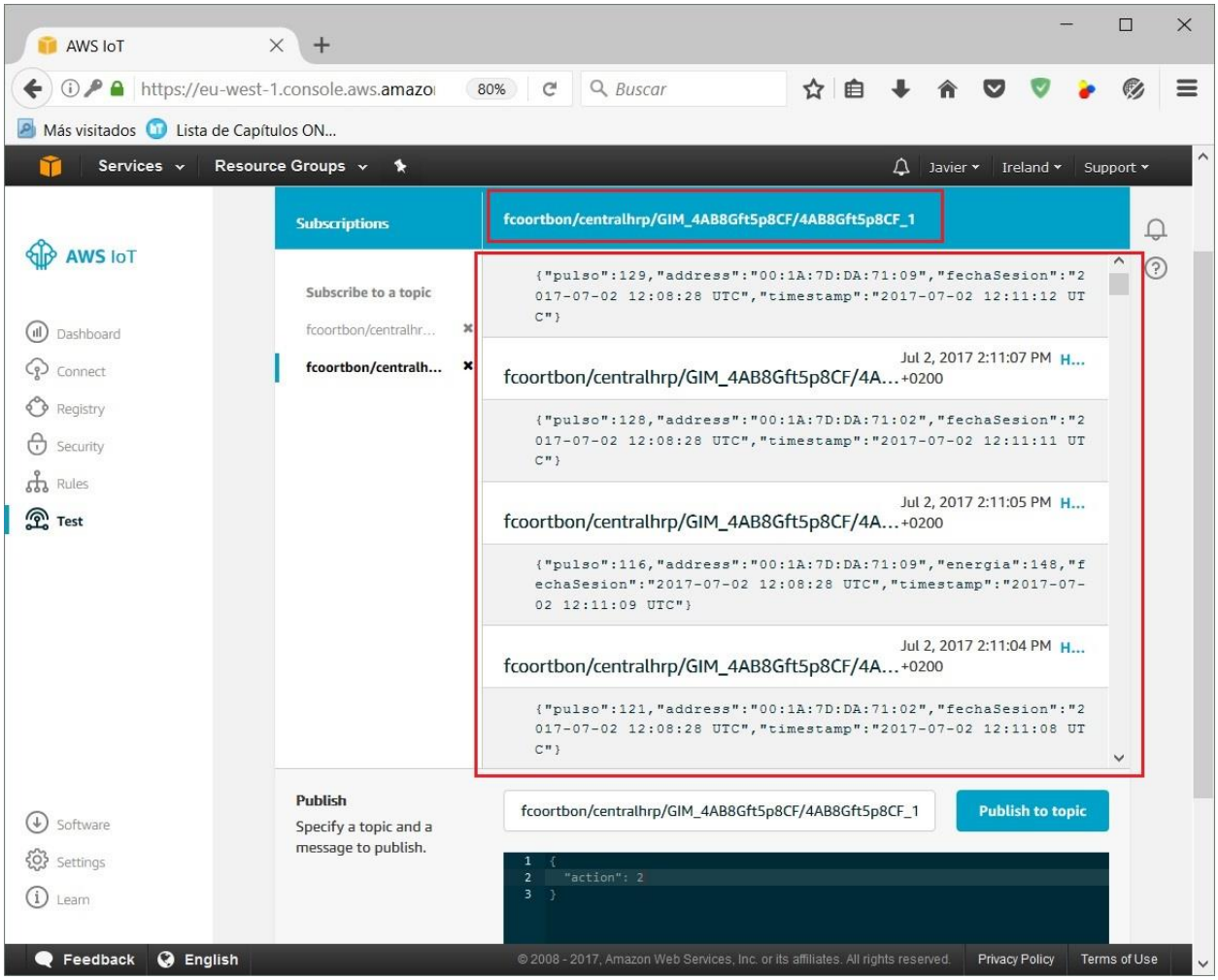

Figura 41: Cliente MQTT de AWS IoT: Recepción de los datos publicados por el bundle.

5.8.3 Fin de la sesión

Para finalizar la sesión que empezamos en la sección anterior, vamos a enviar desde el cliente MQTT de AWS IoT la orden `TERMINAR_SESION` (2).

A continuación, en la figura 42 se muestra una captura del envío de la orden para terminar la sesión y de cómo el dispositivo ha publicado su nuevo estado: `ESTADO_IDLE` (1).

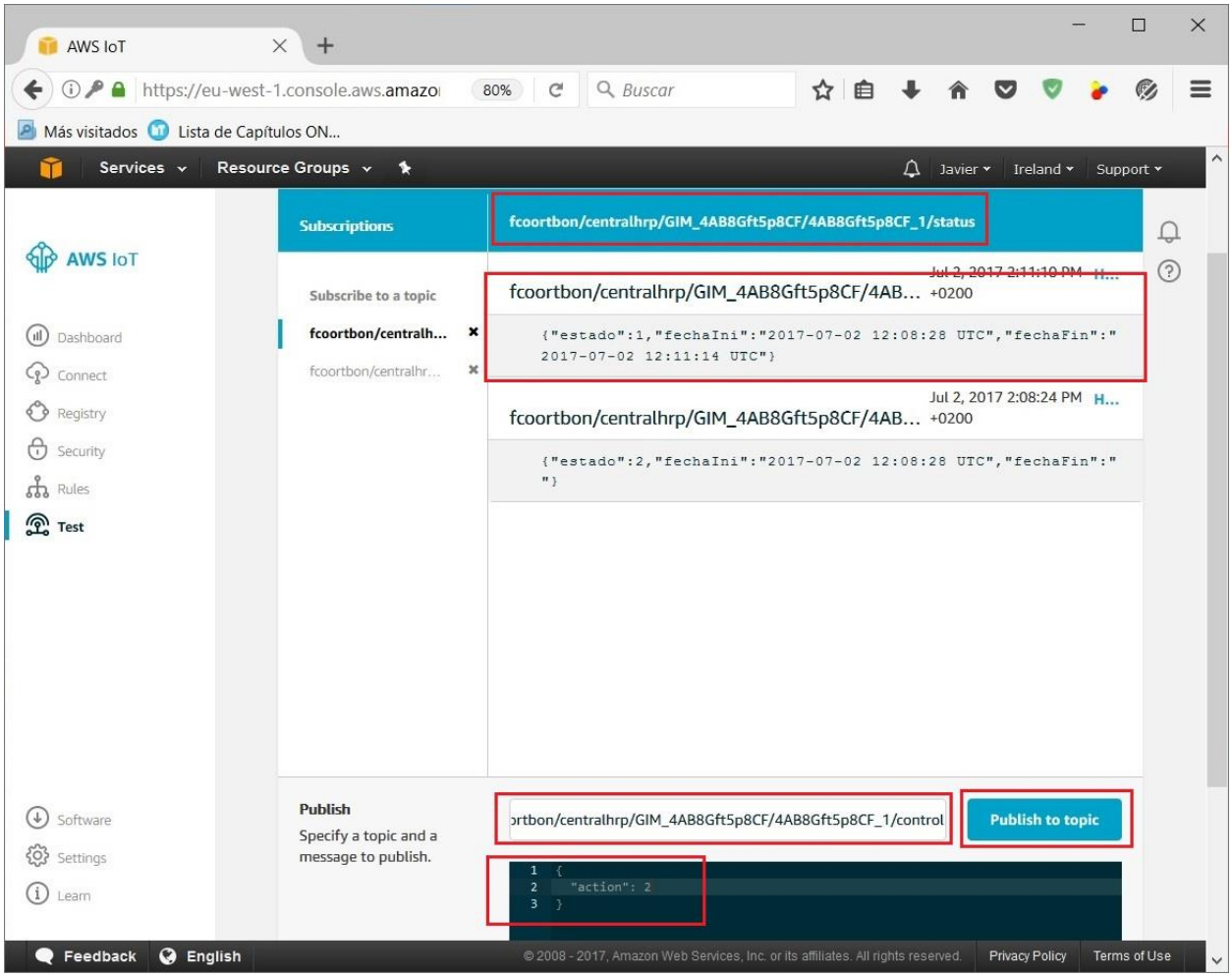

Figura 42: Cliente MQTT de AWS IoT: Envío de orden TERMINAR_SESION y recepción del estado publicado por el bundle.

A continuación, en las figuras 43 y 44 se muestran capturas de los sensores emulados de pulso y de logs del bundle al acabar la sesión.

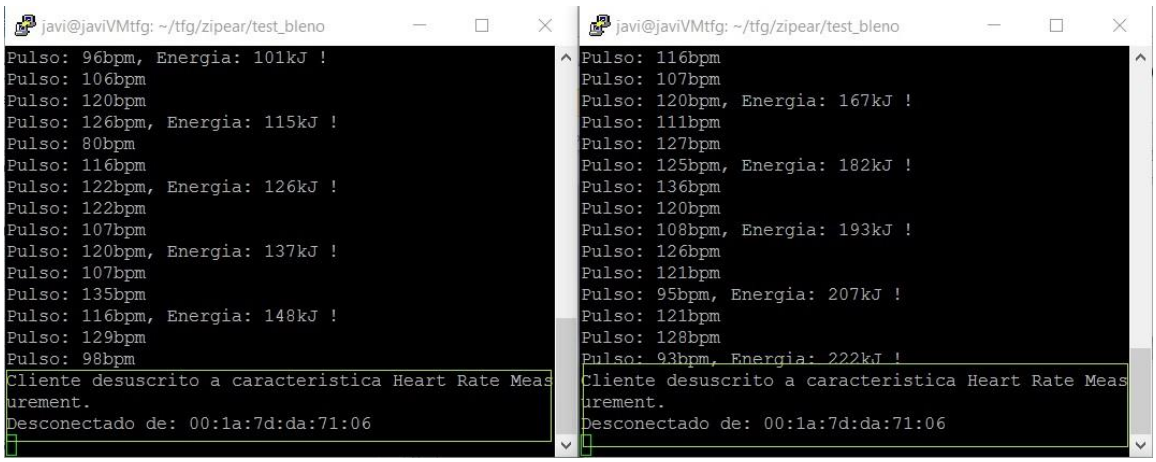

Figura 43: Sensores emulados desconectados del bundle al acabar la sesión.

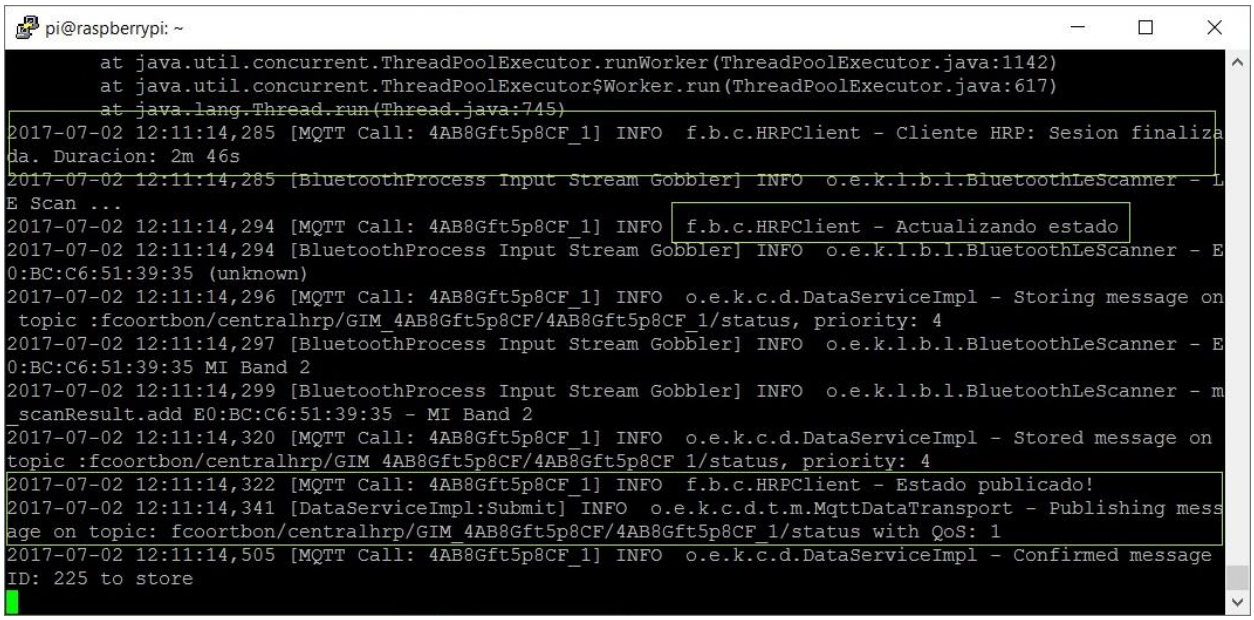

Figura 44: Logs del bundle HRPClient al acabar una sesión.

6 APLICACIÓN ANDROID

Como hemos visto anteriormente, mediante la aplicación Android desarrollada un monitor puede controlar un bundle HRPClient, con la idea de que lo ponga a funcionar (inicie una nueva sesión de funcionamiento en él) cuando vaya a dar comienzo una clase de deporte y que lo pare (termine la sesión iniciada previamente) cuando la clase acabe, y de esta forma monitorizar durante la clase el pulso y energía gastada de sus alumnos.

En este capítulo se va a ver cómo se ha desarrollado la aplicación y su funcionamiento.

6.1 Introducción

Para desarrollar la aplicación Android se ha hecho uso del entorno de desarollo Android Studio. Además, se han usado las siguientes librerías:

- **AWS SDK para Android**[22]

  Se ha usado el SDK de Amazon Web Services, en concreto las librerías para AWS IoT, para poder conectarnos a la nube y publicar y recibir información desde ella.

  Para incluir las librerías de AWS IoT, se añade en el fichero `build.gradle`, dentro del elemento dependencies, lo siguiente:

  ```
  compile 'com.amazonaws:aws-android-sdk-iot:2.2.+'
  ```

  Podemos encontrar ejemplos de código y aplicaciones usando el SDK de AWS aquí: https://github.com/awslabs/aws-sdk-android-samples

  En concreto, para el uso de AWS IoT ha sido muy útil el siguiente ejemplo: https://github.com/awslabs/aws-sdk-android-samples/tree/master/AndroidPubSub

- **Android GraphView**[23]

  Se ha usado esta librería para construir y mostrar gráficas con los datos de pulso en función del tiempo. Se ha incluido la librería del mismo modo que antes, pero añadiendo la siguiente línea:

  ```
  compile 'com.jjoe64:graphview:4.2.1'
  ```

A continuación, la figura 45 muestra una captura en la que se ven los distintos ficheros de clases y recursos usados en la aplicación Android.

[22] https://github.com/aws/aws-sdk-android

[23] http://www.android-graphview.org/

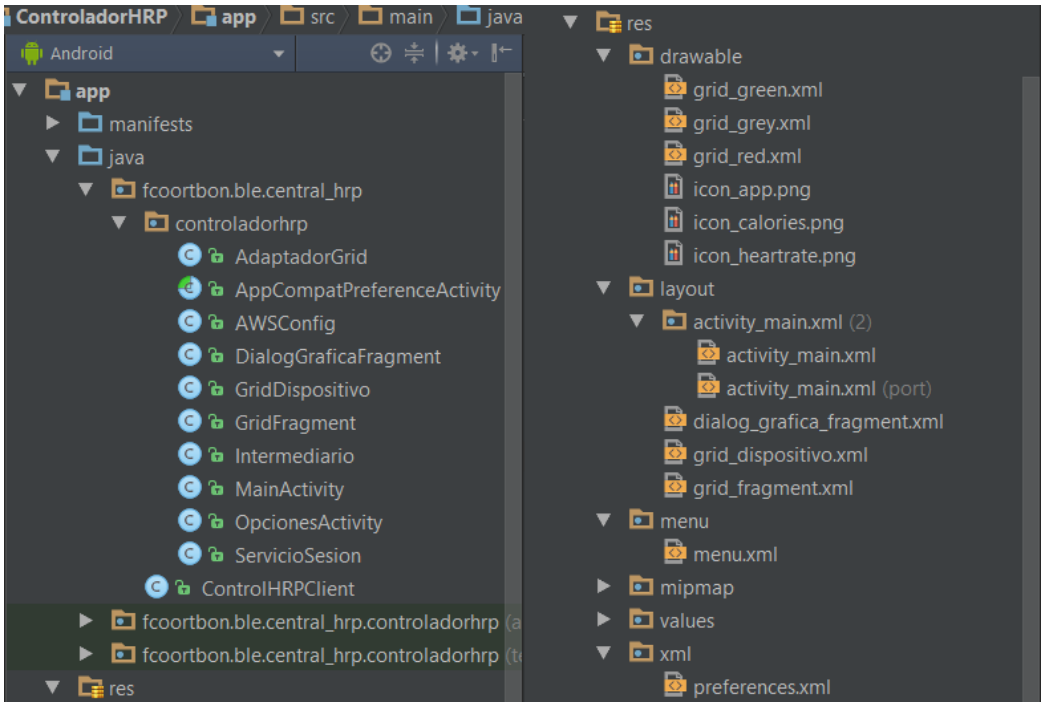

Figura 45: Proyecto en Android Studio.

6.2 Pantallas de la aplicación

La aplicación consta sólo de un par de pantallas: la pantalla principal y la pantalla de opciones.

6.2.1 Pantalla principal

En la pantalla principal se ubican un par de botones para iniciar y acabar una sesión en un bundle HRPClient. Previamente, hemos debido de haber configurado el dispositivo que queremos controlar en la pantalla de opciones.

Además de los botones, hay un cuadro de texto que indica el estado de la aplicación respecto a la nube y al dispositivo. Los códigos de los posibles estados se recogen en la clase ServicioSesion, de la siguiente forma:

```
public static final int CODIGO_DESCONECTADO = 0;

public static final int CODIGO_ERROR_CONECTAR = 1;
public static final int CODIGO_ERROR_SUSCRIBIR = 2;
public static final int CODIGO_ERROR_DISPOSITIVO_TIMEOUT = 3;
public static final int CODIGO_ERROR_DISPOSITIVO_OCUPADO = 4;
public static final int CODIGO_ERROR_PUBLICAR = 5;
public static final int CODIGO_ERROR_INICIAR_SESION = 6;
public static final int CODIGO_ERROR_FINALIZAR_SESION = 7;
public static final int CODIGO_ERROR_DISPOSITIVO_TIMEOUT_FINALIZAR_SESION = 8;

public static final int CODIGO_CONECTANDO = 30;
public static final int CODIGO_CONEXION_ESTABLECIDA = 31;
public static final int CODIGO_DISPOSITIVO_DISPONIBLE = 32;
public static final int CODIGO_SESION_INICIADA = 33;
public static final int CODIGO_FINALIZANDO_SESION = 34;

public static final int CODIGO_SESION_FINALIZADA = 35;
```

Por último, aparte del cuadro de texto para el estado y de los botones, existe una cuadrícula (`GridView`) en la que cada elemento representa un dispositivo BLE del que se está recibiendo información. Además, al hacer clic en uno de ellos, se muestra una gráfica con las medidas de pulso recibidas de ese dispositivo en función

del tiempo.

A continuación, las figuras 46, 47, 48 y 49 muestran una serie de capturas de la pantalla principal:

1. Al abrir la aplicación:

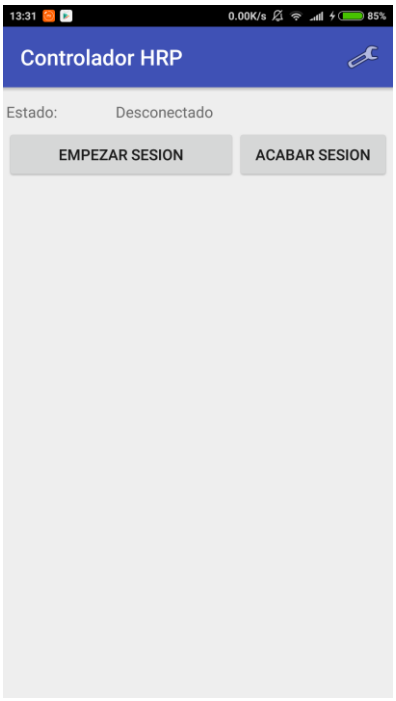

Figura 46: Pantalla principal de la aplicación Android: al abrir la aplicación.

2. Durante una sesión en la cual el bundle HRPClient está conectado a dos sensores emulados de pulso:

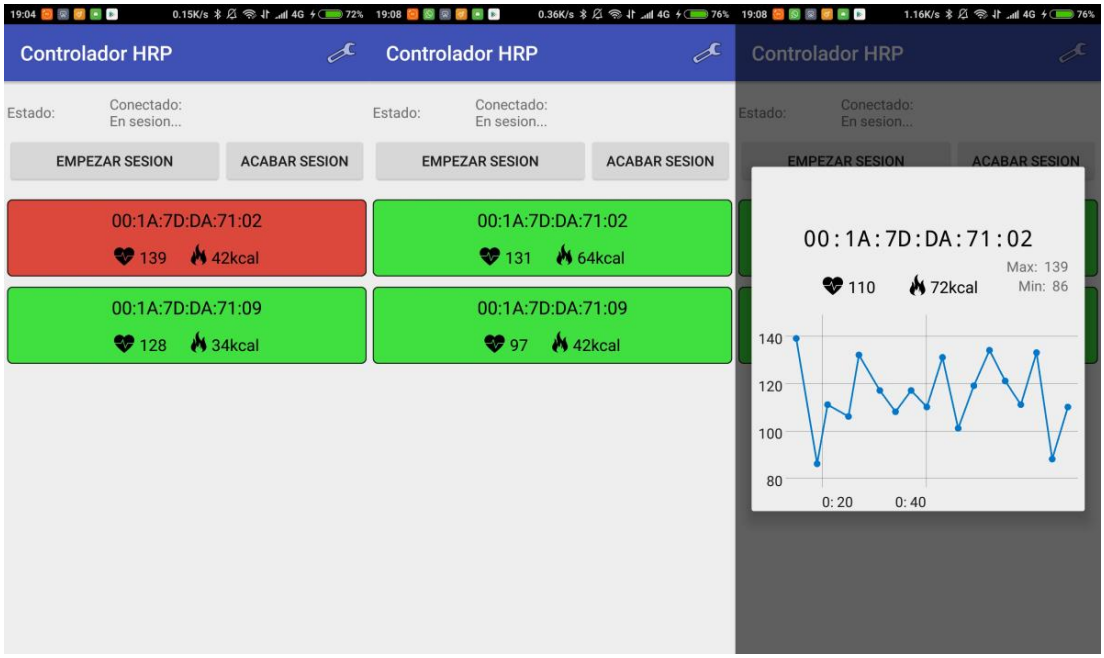

Figura 47: Pantalla principal de la aplicación: durante una sesión (vertical).

Podemos girar sin problema la pantalla y no perder la información que se está mostrando:

Figura 48: Pantalla principal de la aplicación: durante una sesión (horizontal).

3. Diferentes errores al intentar iniciar sesión:
   - Se intenta iniciar sesión pero no se ha configurado el código del gimnasio.
   - Se intenta iniciar sesión pero el dispositivo configurado no está conectado (no ha respondido a la orden `PUBLICAR_ESTADO`).
   - Se intenta iniciar sesión en un dispositivo que ya está en sesión (ha respondido `ESTADO_EN_SESION` a la orden `PUBLICAR_ESTADO`).

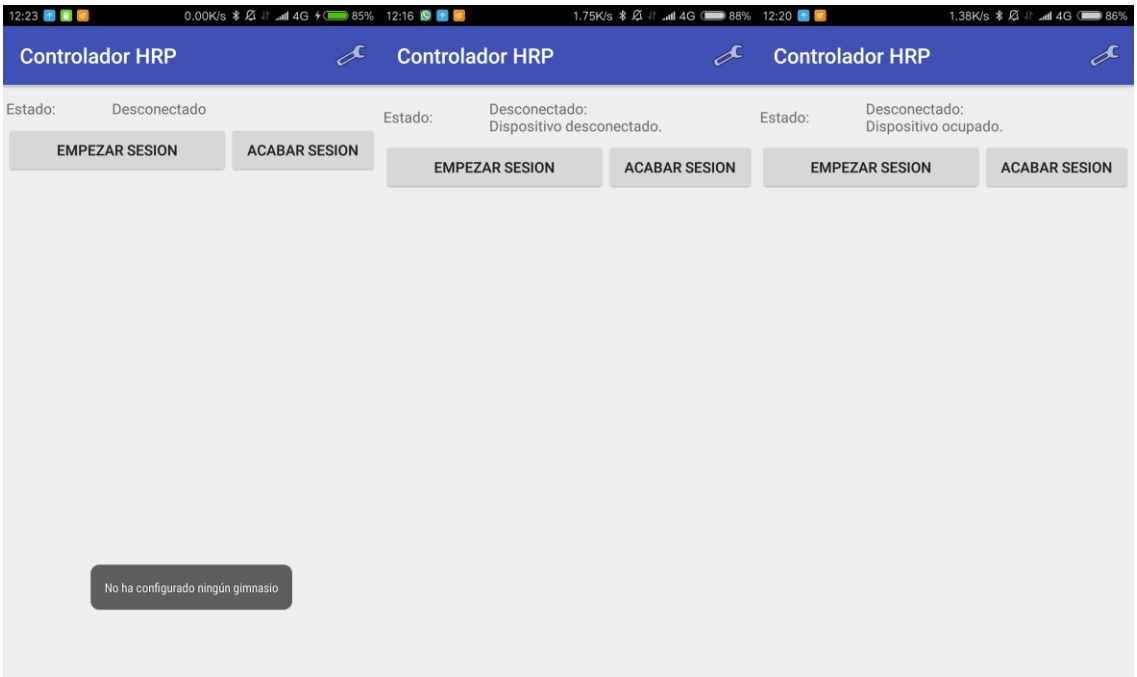

Figura 49: Pantalla principal de la aplicación: diferentes errores al iniciar una sesión.

6.2.2 Pantalla de opciones

Mediante esta pantalla se configura cuál es el dispositivo que se desea controlar (mediante el código del gimnasio y el número del dispositivo) y también el parámetro de pulso alto. La aplicación considerará cualquier medida de pulso recibida con valor igual o mayor al de este parámetro como pulso alto y la mostrará en rojo.

A continuación, la figura 50 muestra unas capturas de esta pantalla.

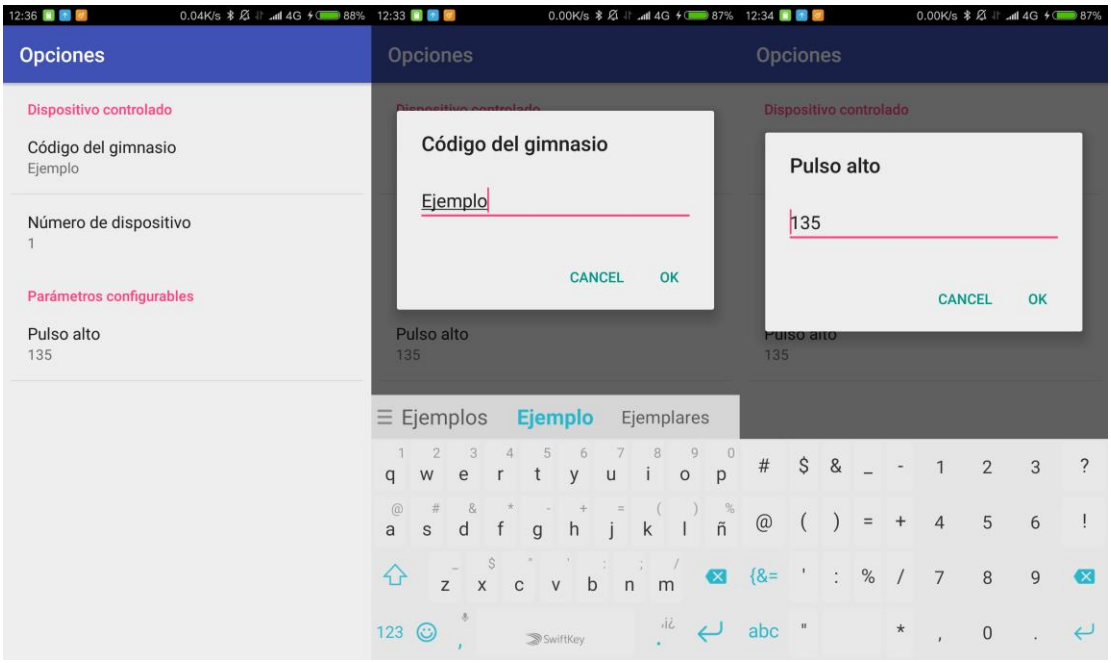

Figura 50: Pantalla de opciones de la aplicación.

6.3 Funcionamiento

Para explicar el funcionamiento de la aplicación, vamos a diferenciar las siguientes partes:

- Clase **AWSConfig**
- Actividad **MainActivity**
- Fragmento **GridFragment**
- Diálogo **DialogGraficaFragment**
- Servicio **ServicioSesion**
- Objeto Application **Intermediario**
- Actividad **OpcionesActivity**

Básicamente, en la aplicación se hace uso de un objeto singleton[24] **Intermediario**, que se encarga de guardar en memoria todos los datos que se reciben y de actualizar la información que se muestra en las diferentes vistas (**MainActivity**, **GridFragment, DialogGraficaFragment**).

El objeto **Intermediario** obtiene los datos a través del servicio **ServicioSesion**. Este servicio se inicia al hacer clic en el botón de iniciar sesión de la pantalla principal (desde **MainActivity**) y se encarga de manejar la conexión con la nube: publica mensajes de órdenes y recibe los mensajes de datos. Al recibir datos, se los pasa al objeto **Intermediario**.

6.3.1 Clase AWSConfig

En esta clase se recoge la configuración relacionada con la nube de Amazon, declarando constantes como el endpoint a usar, la región ó la política que se va a aplicar al certificado de la aplicación[25].

[24] Singleton: Sólo existe una instancia de la clase y se puede obtener su referencia mediante el método getIstance (en Android para obtener la referencia del objeto Application se usa getApplication).

[25] Para el certificado y la política de la aplicación Android se hace uso de Amazon Cognito. Ver Sección 2.3.3.

Para que funcione bien la aplicación con la nube, se ha dado valor a estas constantes de acuerdo a mi endpoint de AWS, no obstante, para la entrega del código estas constantes estarán vacías.

Las clases **MainActivity** y **ServicioSesion** son las que usan las constantes definidas en esta clase.

6.3.2 Actividad MainActivity

Esta actividad se encarga de presentar la pantalla principal, que como vimos constaba de un cuadro de texto que indicaba el estado respecto al dispositivo que se quiere controlar, un par de botones para iniciar y finalizar una sesión en él y de un fragmento con una vista `GridView`.

Al crearse la actividad (método onCreate):

1. Se pasa al objeto `Intermediario` la referencia de la actividad, de esta forma, el objeto Intermediario podrá invocar el método `actualizarEstado` de la actividad.

```
intermediario = (Intermediario) getApplication();
intermediario.setMainActivity(this);
```

2. Comprobamos si el servicio ServicioSesion ya está en funcionamiento. Si estaba en funcionamiento, es porque habíamos iniciado una sesión anteriormente pero la actividad se destruyó y ahora se ha vuelto a crear (por ejemplo, si giramos el teléfono, ocurre esto). En caso de ser así, nos enlazamos con el servicio.

```
if (isMyServiceRunning(ServicioSesion.class))
{
  Log.d(LOG_TAG, "Bind al servicio...");
  bindService(new Intent(this, ServicioSesion.class), mConnection, 0);
}
```

   Al enlazarnos al servicio, obtenemos a través del objeto `mConnection` un objeto de tipo `messenger` que podemos utilizar para enviar mensajes al servicio. Útil para enviarle el mensaje de terminar sesión al hacer clic en el botón de acabar sesión.

3. Comprobamos si la aplicación ya tiene una identidad creada (pareja certificado-clave privada). Si no la tiene, la creamos y la guardamos para no tener que crearla de nuevo posteriormente.

Una vez la actividad ha sido creada, queda a la espera de los eventos de clic en los botones.

Al hacer clic en el botón de empezar sesión, se llama al método `onClick` del 'listener' del botón. La figura 51 muestra un diagrama de flujo que explica la lógica de este método.

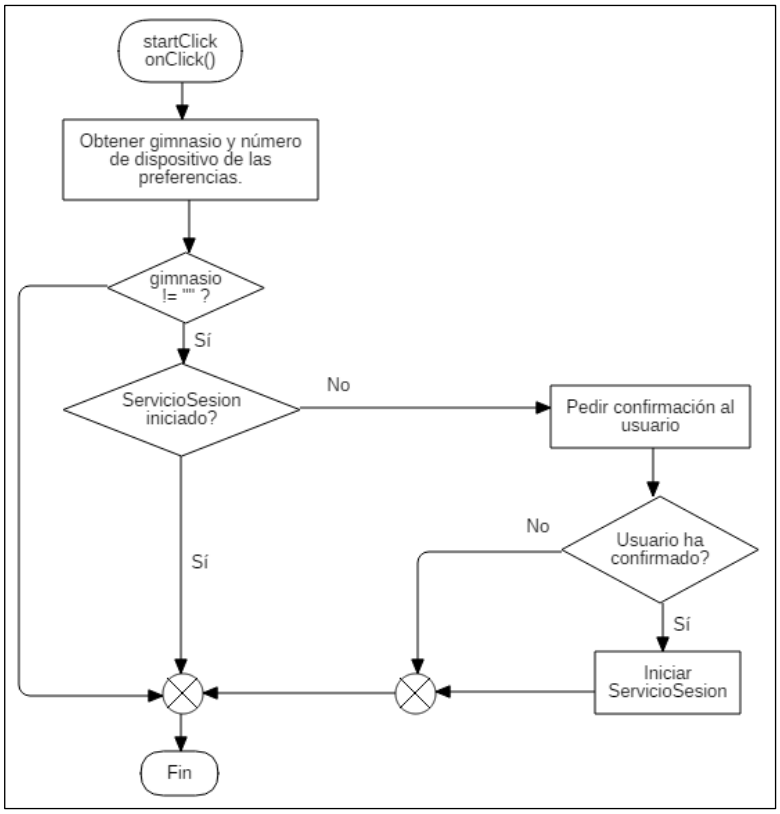

Figura 51: Diagrama de flujo del método onClick el botón de empezar sesión.

Al hacer clic en el botón de acabar sesión, se llama al método `onClick` del 'listener' del botón. La figura 52 muestra un diagrama de flujo que explica la lógica de este método.

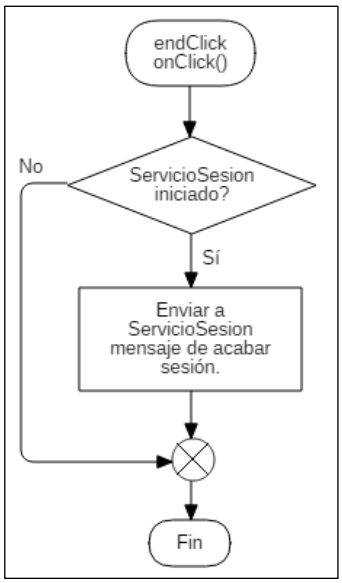

Figura 52: Diagrama de flujo del método onClick el botón de acabar sesión.

Como vemos, quién se encarga de la interacción con la nube y el bundle es el servicio **`ServicioSesion`**.

6.3.3 Fragmento GridFragment

Este fragmento contiene únicamente una vista `GridView` en la que cada elemento representa un dispositivo BLE del que se está recibiendo información.

Para construir la vista se utiliza la clase `AdaptadorGrid`, que recibe como parámetro la vista `GridView` y una lista de objetos `GridDispositivo`.

Clase GridDispositivo

Un objeto GridDispositivo representa un dispositivo BLE del que se ha recibido información y sirve para encapsular la última información recibida de él (dirección MAC, pulso y energía).

Además, contiene un método estático, `alerta`, que recibe como parámetro una medida de pulso y devuelve **`false`** si esta medida está por debajo del valor del parámetro pulso alto ó **`true`** en caso contrario.

Clase AdaptadorGrid

Por cada objeto `GridDispositivo` crea un nuevo elemento en la vista `GridView`. Este elemento contiene la siguiente información: dirección MAC del dispositivo, pulso y energía gastada.

Además, el color que da a cada elemento varía:

- Si el dispositivo BLE al que hace referencia el objeto `GridDispositivo` está inactivo (método `isActive` devuelve `false`), el elemento se cólera en gris.
- Si el dispositivo está activo y el valor de la medida de pulso es igual o mayor al valor configurado como pulso alto, se colorea en rojo.
- Si el dispositivo está activo y el valor de la medida de pulso está por debajo del valor configurado como pulso alto, se colorea en verde.

La clase GridFragment dispone de un método para actualizar el contenido de la vista `GridView`, que es llamado por el objeto Intermediario cada vez que le llegan nuevos datos. A continuación se muestra el código de este método.

```
public void updateGrid(ArrayList<GridDispositivo> dispositivos)
{
  Log.d(LOG_TAG, "#updateGrid: "+dispositivos.size());
  final ArrayList<GridDispositivo> disps = dispositivos;
  getActivity().runOnUiThread(new Runnable() {
    @Override
    public void run() {
      adaptador = new AdaptadorGrid(getActivity(), R.layout.grid_dispositivo,
                                                                    disps);
      gridView.setAdapter(adaptador);
    }
  });
}
```

Además, la vista `GridView` dispone de un 'listener' para el evento de clic en uno de sus elementos, y cuando se hace clic en uno de ellos, se muestra una gráfica en función del tiempo con las medidas de pulso recibidas por el dispositivo BLE en cuyo elemento se ha hecho clic. Recordemos que cada elemento representa un dispositivo BLE del que se está recibiendo información.

Para ello se crea un diálogo de tipo `DialogGraficaFragment`, se le pasa como argumento la dirección del dispositivo BLE y se muestra. A continuación se muestra el código del método `onItemClick`, llamado al hacerse clic en uno de los elementos.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id)
{
  GridDispositivo dispositivo = adaptador.getItem(position);
  Log.d(LOG_TAG, "Clic en "+dispositivo.getAddress());

  DialogGraficaFragment grafica = new DialogGraficaFragment();
  Bundle argumentos = new Bundle();
  argumentos.putString("address", dispositivo.getAddress());
  grafica.setArguments(argumentos);
  grafica.show(getFragmentManager(), "Fragment grafica");

}
```

6.3.4 Diálogo DialogGraficaFragment

Como hemos visto en la sección anterior, al hacer clic en un elemento de la vista `GridView` se crea y muestra un nuevo diálogo `DialogGraficaFragment`, al que se le pasa como argumento la dirección MAC del dispositivo BLE cuya gráfica de pulso quiere construirse.

Para crear la gráfica, el `DialogGraficaFragment` llama al método `getPuntos` del objeto `Intermediario`, pasándole como parámetro la dirección MAC del dispositivo BLE, y este le retorna una tabla de puntos (`DataPoint[]`) con la que construye la gráfica. Para construir la gráfica se hace uso de la librería GraphView, como se comentó en la introducción de este capítulo.

Además, al igual que la clase `GridFragment` tenía el método `updateGrid`, la clase `DialogGraficaFragment` tiene el método `updateGraph`, el cual es llamado por el objeto `Intermediario` cada vez que llegan datos del dispositivo BLE cuya información se está mostrando en la gráfica, y de esta forma la gráfica se actualiza en tiempo real.

Además de la gráfica, se muestran otros datos: últimas medidas recibidas de pulso y energía gastada, pulso máximo y pulso mínimo. En la figura 53 se muestra una imagen de uno de estos diálogos.

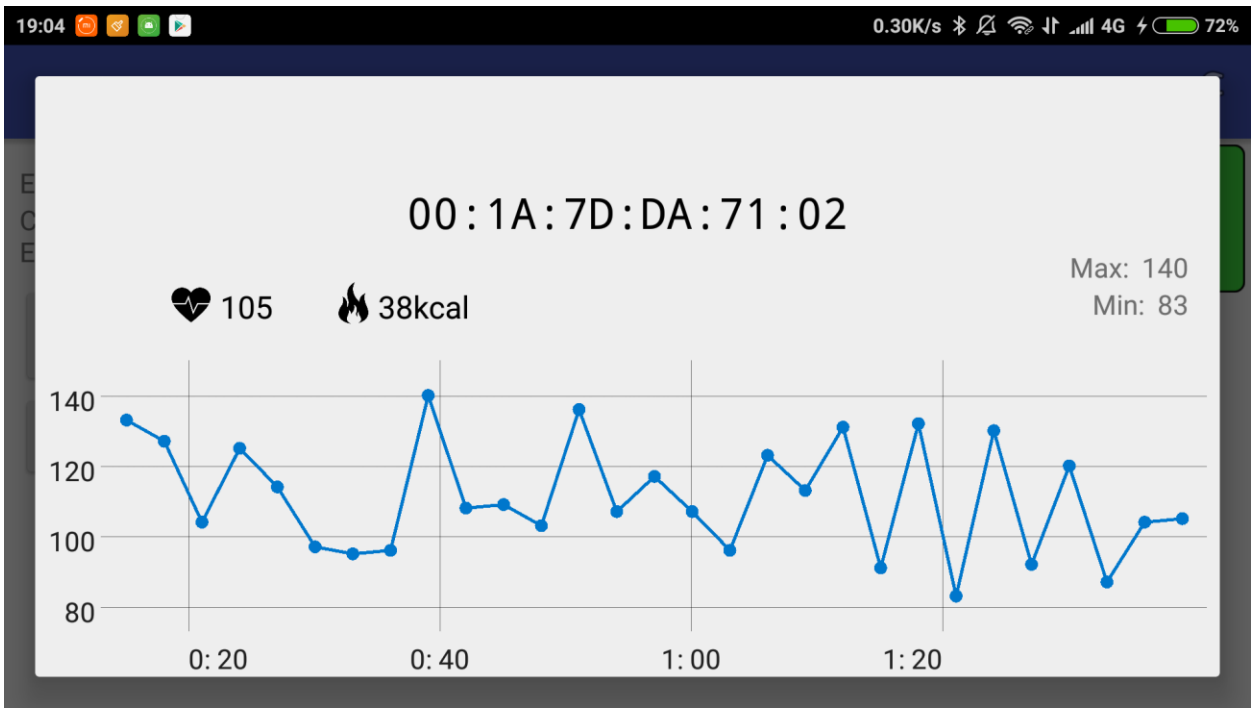

Figura 53: Ejemplo de un DialogGraficaFragment.

6.3.5 Servicio ServicioSesion

Este servicio se inicia cuando se quiere comenzar una nueva sesión en un bundle HRPClient, permanece activo durante la sesión y se destruye al finalizar la sesión.

Inicio del servicio

Recordemos que el servicio se inicia al hacer clic en el botón de empezar sesión de la pantalla principal (es decir, desde la actividad MainActivity).

Al iniciarse el servicio, este se suscribe al topic de estado del dispositivo, publica en el topic de control la orden `PUBLICAR_ESTADO` y programa un timeout de 3 segundos. Si vence el timeout o el dispositivo responde con `ESTADO_EN_SESION`, el servicio se destruye. Si, en caso contrario, el dispositivo responde `ESTADO_IDLE`, el servicio se suscribe al topic de datos, envía la orden `COMENZAR_SESION` y programa un nuevo timeout.

Si vence el timeout, el servicio se destruye, pero si el dispositivo responde con `ESTADO_EN_SESION`, el servicio quedará funcionando e irá recibiendo los mensajes de datos publicados por el dispositivo.

Para ver un diagrama de lo que se acaba de explicar, ver: Figura 20: Intercambio de mensajes dado cuando se ha iniciado una sesión con éxito.

Servicio en sesión

Una vez el servicio ha conseguido iniciar una nueva sesión en el dispositivo (en el bundle HRPClient), no hace nada salvo procesar los mensajes de datos que vayan llegando, los cuales se reciben a través del método `onMessageArrived`.

Al recibir un mensaje de datos, se extrae del mensaje la dirección del dispositivo BLE, la medida de pulso y la energía gastada (en caso de estar presente) y se pasan estos datos al objeto `Intermediario`, que los mantiene en memoria.

A continuación se muestra al código del método auxiliar `procesarMensajeDatos`.

```
private void procesarMensajeDatos(String message)
{
  try
  {
    JSONObject messageJSON = new JSONObject(message);

    String address = messageJSON.getString("address");
    int pulso = messageJSON.getInt("pulso");
    int energia = -1;
    if (messageJSON.has("energia"))
      energia = kJtokcal(messageJSON.getInt("energia"));

    Date now = new Date();
    int time = (int) (now.getTime() - fechaInicio.getTime())/1000;
    DataPoint puntoPulso = new DataPoint(time, pulso);
    intermediario.nuevoDatoPulso(address, puntoPulso, energia);
  }
  catch (JSONException e)
  {
    Log.e(LOG_TAG, "Excepcion JSON", e);
  }
}
```

Fin de la sesión

El servicio envía la orden `TERMINAR_SESION` cuando en la pantalla principal se hace clic en el botón de acabar sesión. Al publicar la orden, activa un timeout. Si vence el timeout, no se da la sesión por finalizada y el servicio no se destruye. Si, por el contrario, se recibe como respuesta `ESTADO_IDLE`, la sesión se da por finalizada y el servicio se destruye.

6.3.6 Objeto Application Intermediario

6.3.6.1 Clase Application

En Android, toda aplicación dispone de la clase Application, la cual es la base de la aplicación y puede usarse para mantener un estado global [20].

Se puede realizar una implementación propia de esta clase: sólo tenemos que darle nombre en el manifiesto (fichero AndroidManifest.xml) e implementarla heredando de la clase Application. De esta forma, al crearse la aplicación se creará una instancia de esta clase. En nuetro caso, esta es la clase Intermediario.

En el fichero AndroidManifest.xml:

```
<application
      android:name=".Intermediario"
      android:icon="@drawable/icon_app"
      android:label="@string/app_name" >
```

Clase Intermediario:

```
public class Intermediario extends Application
{
  …
}
```

6.3.6.2 Objeto Intermediario

Este objeto se encarga de mantener los datos de la sesión en memoria e interacciona con los demás componentes para mostrar la información correcta en el momento adecuado. Si consideramos las entidades del patrón Modelo-Vista-Controlador, este objeto sería tanto Modelo como Controlador.

1. Respecto a la actividad principal (`MainActivity`).

   Actualiza su estado cada vez que el servicio ServicioSesion invoca al método actualizarEstado.

   1.1. Respecto a `GridFragment`

   Recordemos que la actividad principal contiene un fragmento con un `GridView` en el que cada elemento representa la última medida recibida de un dispositivo.

   El objeto `Intermediario` se encarga de refrescar la información que se debe mostrar en esta vista cada vez que recibe nuevos datos. Para ello invoca al método `updateGrid` del `GridFragment`.

2. Respecto a la gráfica de medidas de pulso de un dispositivo (`DialogGraficaFragment`)

   Recordemos que al hacer clic en un elemento del `GridView`, se muestra un diálogo con una gráfica en la que aparecen las medidas de pulso recibidas de ese dispositivo en concreto así como otros estadísticos.

   El objeto `Intermediario` se encarga de proporcionar al diálogo de la gráfica, es decir, al `DialogGraficaFragment` en cuestión, los datos del dispositivo adecuado (el objeto `DialogGraficaFragment` invoca el método `getPuntos` del objeto `Intermediario`) así como de ir pasándole los nuevos datos relativos al dispositivo cuya gráfica se está mostrando para que la gráfica se actualice en tiempo real (para ello el objeto `Intermediario` invoca al método `updateGraph` del objeto `DialogGraficaFragment`.

A continuación, la figura 54 muestra un diagrama de flujo para el método `nuevoDatoPulso` de la clase `Intermediario`.

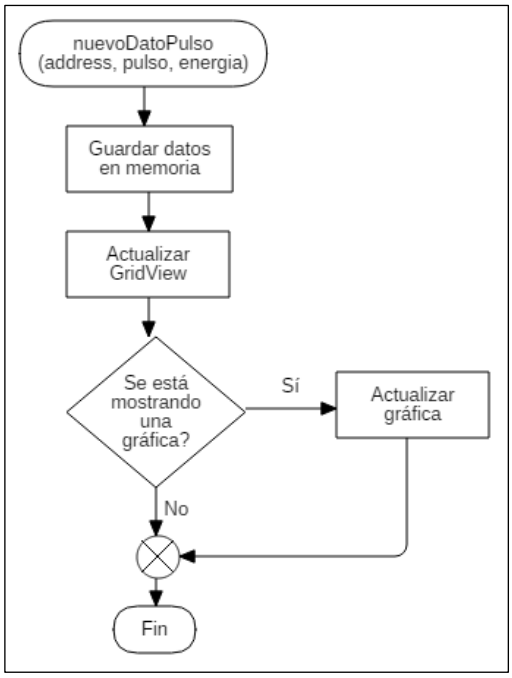

Figura 54: Diagrama de flujo del método nuevoDatoPulso.

6.3.7 OpcionesActivity

Esta actividad se encarga de presentar la pantalla de opciones, a través de la cual el usuario configura el dispositivo que va a controlar, a partir del código del gimnasio y el número de dispositivo, y otros parámetros como el de pulso alto.

6.3.8 Diagramas de secuencia

A continuación se van a incluir una serie de diagramas de secuencia para que quede más claro la lógica de la aplicación y cómo interaccionan los distintos componentes entre sí.

6.3.8.1 Inicio de una nueva sesión en el dispositivo controlado

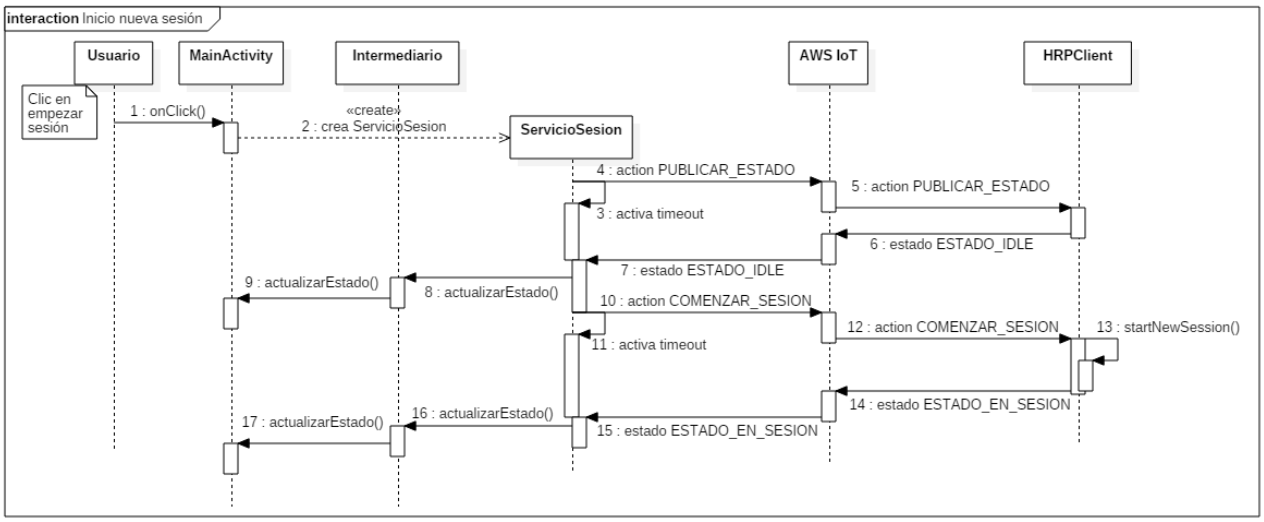

Figura 55: Diagrama de secuencia para el inicio de una sesión en un dispositivo HRPClient desde la aplicación Android.

6.3.8.2 Recepción de datos durante una sesión

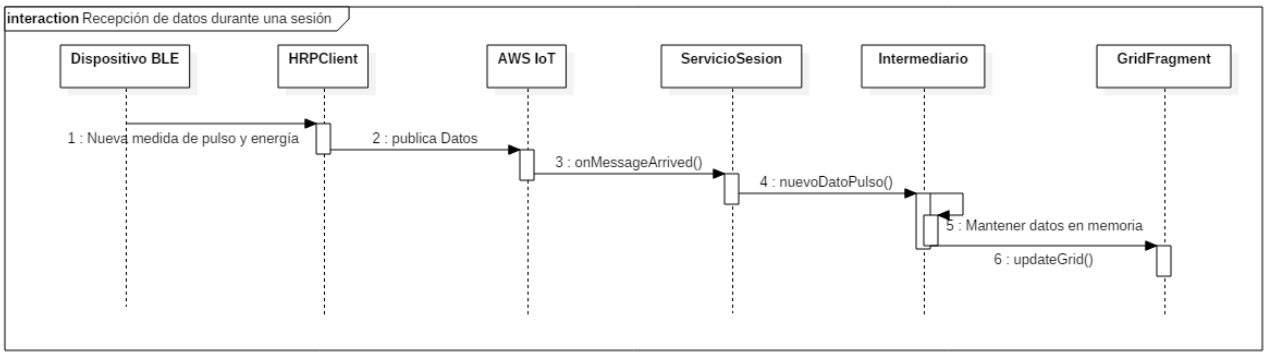

Figura 56: Diagrama de secuencia de la recepción de datos de pulso y energía por parte de la aplicación Android.

6.3.8.3 Creación de gráfica de pulso (DialogGraficaFragment)

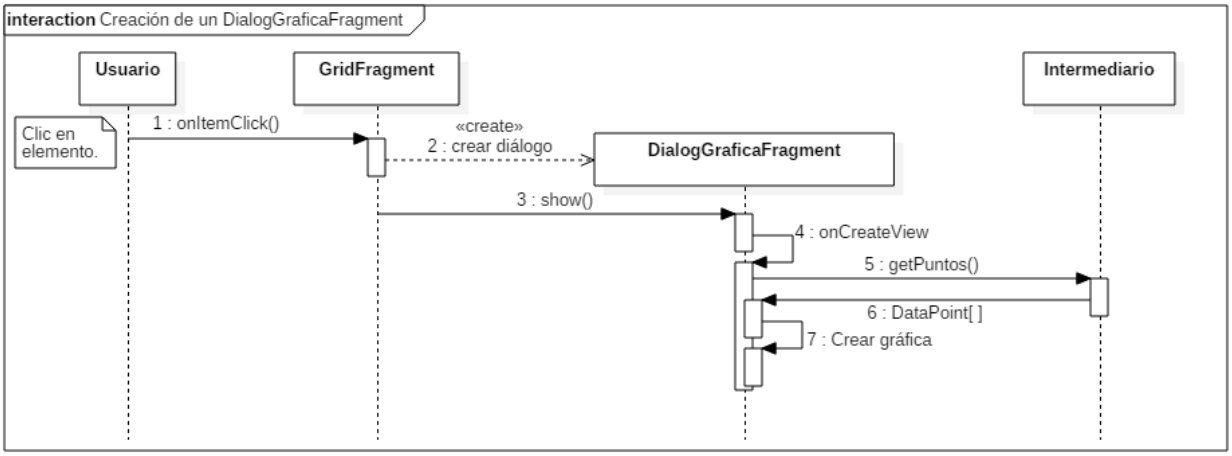

Figura 57: Diagrama de secuencia de la creación de un DialogGraficaFragment.

6.3.8.4 Actualización de la gráfica en tiempo real

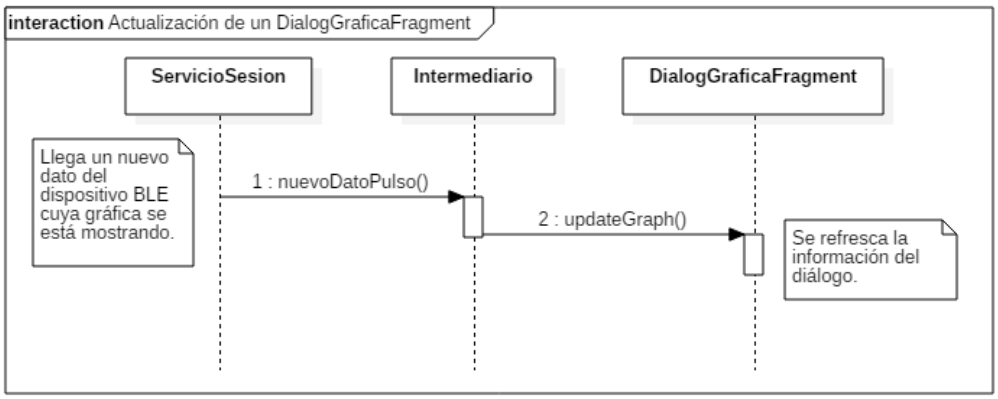

Figura 58: Diagrama de secuencia para la actualización en tiempo real de un DialogGraficaFragment.

6.3.8.5 Fin de una sesión en el dispositivo controlado

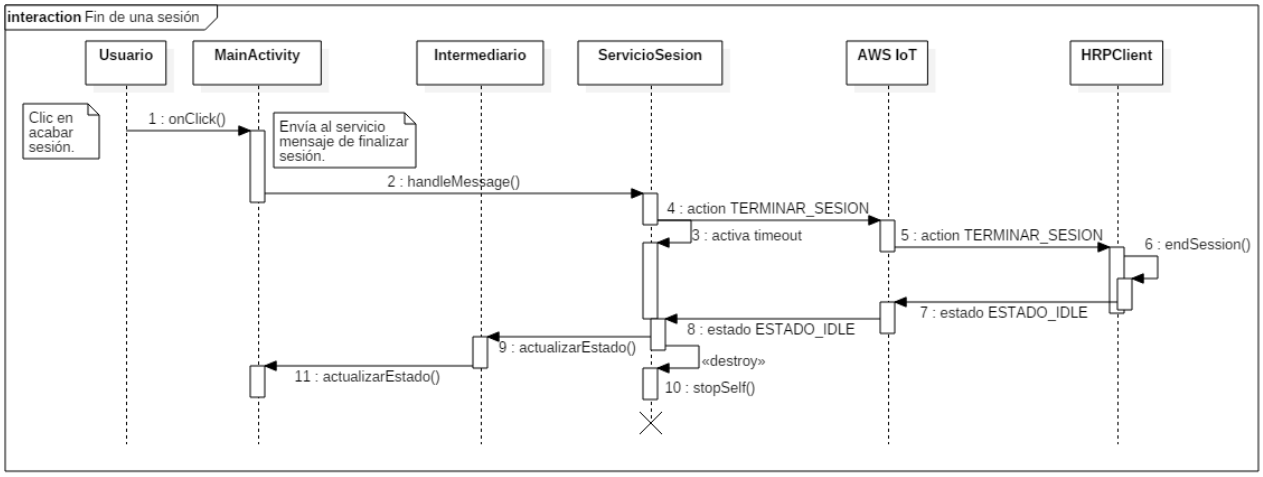

Figura 59: Diagrama de secuencia para la finalización de una sesión en un dispositivo HRPClient desde la aplicación Android.

7 CONCLUSIONES Y LÍNEAS FUTURAS

Mediante la ejecución de este proyecto se ha conseguido aportar una solución extra a la hora de interactuar y leer múltiples dispositivos BLE a la vez, centrándose en el ámbito de la salud y la monitorización del pulso de usuario, que era el objetivo original del trabajo.

Además, haber realizado este trabajo ha supuesto también un gran aprendizaje para mí, pues aparte de los conocimientos adquiridos durante la carrera, he necesitado hacer uso y aprender sobre tecnologías actuales cuyo funcionamiento desconocía por completo, como son Bluetooth Low Energy, Cloud Computing e IoT, lo cual hace que me alegre mucho de haber trabajado en este proyecto.

No obstante, si bien se ha conseguido lograr el objetivo del trabajo, considero que el sistema construido aún es un "Hola Mundo" de lo que podría ser potencialmente. Una de las cosas que me ha preocupado más durante la realización de este trabajo ha sido saber hasta dónde llegar, pues a medida que avanzaba, se me iban ocurriendo funcionalidades y características para el sistema, no ya opcionales, sino necesarias, pero por temas de extensión era preciso acotarlo.

A continuación se van a exponer los aspectos en los que yo creo que se debería trabajar y que serían las líneas futuras del proyecto:

- **Estudiar la compatibilidad con dispositivos BLE reales.**

  Actualmente, los dispositivos BLE que hemos usado para probar nuestro sistema han sido sensores de pulso emulado. Estos sensores implementan el servicio de pulso cardíaco como se especifica en su especificación otorgada por el Bluetooth SIG, por lo que la interacción con un sensor de pulso real debe ser muy parecida, por no decir igual. No obstante, esto no se ha estudiado aún y yo creo que es el aspecto más importante a la hora de continuar con este proyecto.

- **Automatizar la creación de gimnasios en la nube.**

  El sistema está pensado para que sea usado por gimnasios, los cuales se identifican por un código. Sin embargo, actualmente si se quiere añadir un nuevo gimnasio sólo se dispone de un script que crea en la nube una nueva 'thing' correspondiente al gimnasio junto con su certificado y claves pública y privada y una política personalizada para ese gimnasio, que se asocia al certificado del gimnasio.

  Es necesario crear de forma manual una tabla en DynamoDB para el gimnasio y una regla en el motor de reglas de AWS IoT para que se almacenen los datos enviados desde el gimnasio.

- **Añadir registro de gimnasios y usuarios.**

  Actualmente, si se quiere dar de alta un nuevo gimnasio, hay que hacer lo comentado en el punto anterior: ejecutar el script para el nuevo gimnasio y luego crear su tabla en DynamoDB y la regla en el motor de reglas de AWS IoT.

  Sería necesario crear algún servicio web que permitiera a un gimnasio cliente potencial registrarse usando la API que ofrece y obtener su código.

  Además de la posibilidad de registrar gimnasios mediante una API, también sería necesario disponer de una base de datos de usuarios, de forma que no cualquiera pueda controlar un dispositivo de un gimnasio, sino sólo aquellos usuarios con permiso.

- **Identificar a los dispositivos BLE mediante un número o mediante su usuario.**

  Actualmente, cuando se monitorizan los datos proporcionados por los dispositivos BLE, estos se identifican por su dirección MAC. Sirve, pero se puede mejorar si disponemos de una base de datos en la que relacionamos cada dispositivo BLE con algún tipo de identificador, como puede ser un número o el usuario que la está usando.

ANEXO A:
MANUAL DE INSTALACIÓN Y USO DE LA PILA BLUETOOTH PARA LINUX, BLUEZ.

Como se ha visto a lo largo de la memoria, es necesario disponer de BlueZ instalado en:

- Por un lado, nuestro dispostivo Kura en el que corre el bundle HRPClient, ya que como se comentó anteriormente las librerías de Kura para el manejo de Bluetooth hacen uso mediante JNI de herramientas de BlueZ.
- Por otro lado, en el ordenador o dispositivo en el que vayamos a ejecutar los sensores emulado. Como ya vimos anteriormente, una de las dependencias del módulo **`bleno`** era BlueZ.

A.1 Instalación de BlueZ 5.44 en una Raspberry Pi 3 con Raspbian Jessie Lite

Raspian Jessie viene con BlueZ ya instalado, sin embargo es una versión antigua (5.23) que no funciona realmente bien y tiene varios bugs.

Vamos a detallar la instalación para la versión 5.44 de BlueZ, pues era la última hasta la fecha cuando empezamos a trabajar en este proyecto, no obstante para instalar otra versión más reciente se seguiría el mismo procedimiento.

1. Primero, debemos desinstalar y eliminar la versión de BlueZ que viene ya instalada. Para ello, ejecutamos lo siguiente en una terminal:

```
$ sudo apt-get --purge remove bluez
```

A continuación, la figura 60 muestra una captura de la salida de este comando.

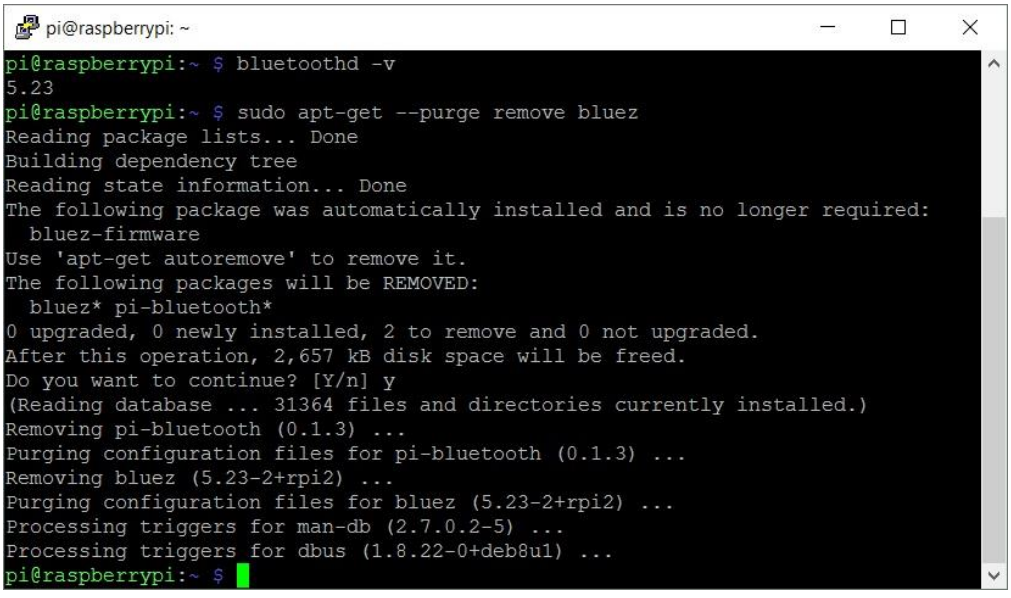

Figura 60: Eliminación de BlueZ.

2. Instalamos una serie de depencias para BlueZ. Desde la terminal, ejecutamos lo siguiente:

```
$ sudo apt-get update
$ sudo apt-get install -y libusb-dev libdbus-1-dev libglib2.0-dev libudev-dev libical-dev libreadline-dev
```

3. Descargamos la última versión de BlueZ en formato .tar.gz, en nuestro caso la 5.44. Los links de descarga de todas las versiones pueden encontrarse aquí: https://www.kernel.org/pub/linux/bluetooth/

   Para ello, desde la terminal ejecutamos lo siguiente:

```
$ cd /tmp/
$ wget https://www.kernel.org/pub/linux/bluetooth/bluez-5.44.tar.gz
$ tar xf bluez-5.44.tar.gz
$ cd bluez-5.44/
```

   A continuación, la figura 61 muestra una captura con la salida de estos comandos.

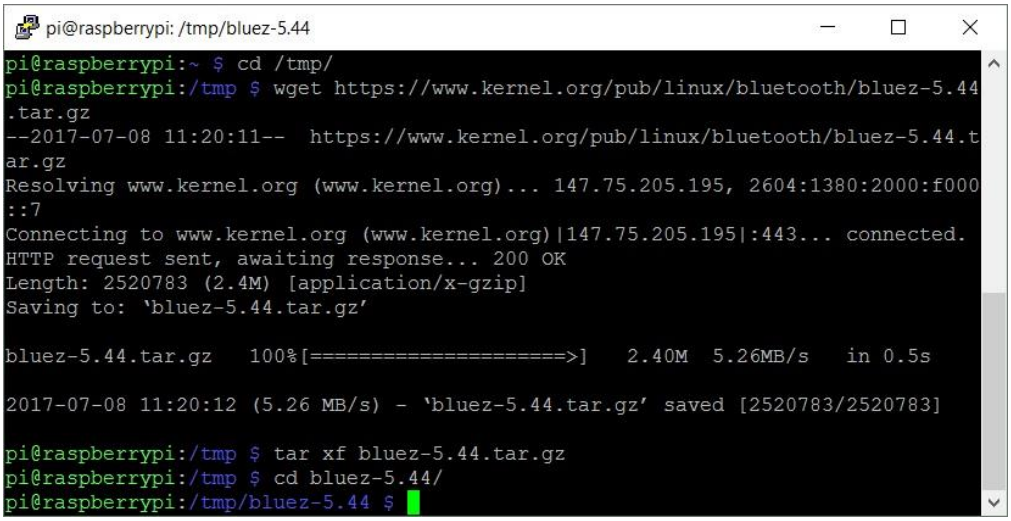

Figura 61: Descarga de BlueZ 5.44.

4. Instalamos BlueZ a partir de los ficheros que acabamos de descargar, indicando que no queremos que sea gestionado por **`systemd`**[26]. Para ello, desde la terminal:

```
$ exports LDFLAGS=-lrt
$ ./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var --enable-library -disable-systemd
$ make
$ sudo make install
```

5. Ubicamos las herramientas **`bluetoothd`** y **`gatttool`** en directorios apropiados del **`PATH`**.

```
$ sudo cp attrib/gatttool /usr/bin/
$ sudo cp ./src/bluetoothd /usr/local/bin/
```

6. Programamos el inicio del servicio bluetooth (ejecutable **`bluetoothd`**) cuando inicie el equipo. Para ello, añadimos la siguiente línea al final del fichero, antes de la línea `'exit 0'` **`/etc/rc.local`**.

[26] https://es.wikipedia.org/wiki/Systemd

```
/usr/local/bin/bluetoothd --experimental &
```

A continuación, la figura 62 muestra una captura del contenido del fichero rc.local

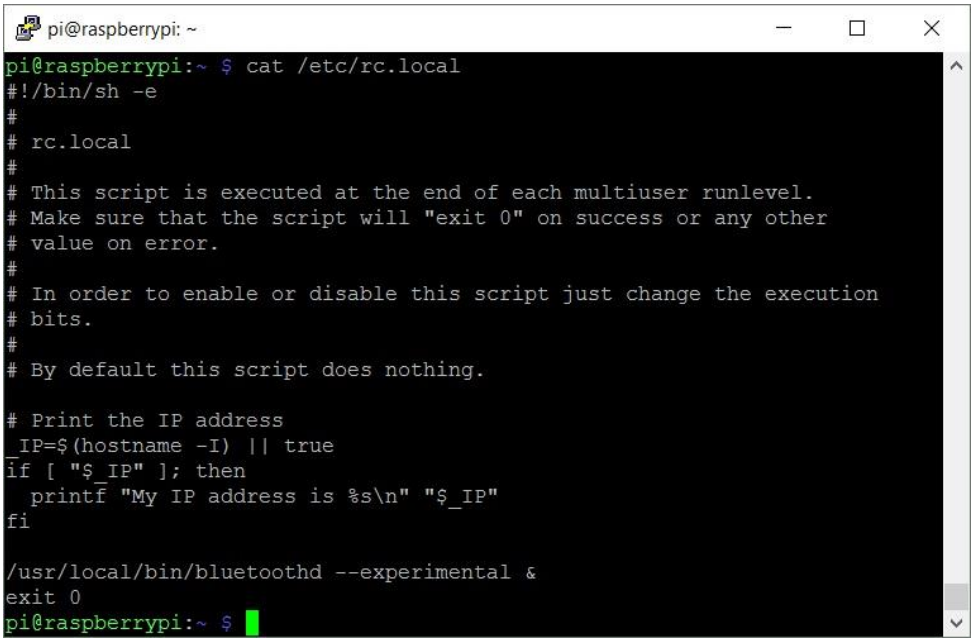

Figura 62: Fichero rc.local en la Raspberry Pi

De este modo ya tenemos instalado BlueZ en nuestra Raspberry Pi y podemos hacer uso de las distintas herramientas que trae, como **`gatttool`**.

Además, cada vez que la Raspberry Pi inicie, se pondrá en marcha el servicio de bluetooth debido a lo que pusimos en el fichero **`rc.local`**. Si en algún momento se desea terminar el servicio, basta con matar el proceso de bluetoothd:

```
$ sudo kill -9 `pidof bluetoothd`
```

A.2 Instalación de BlueZ en Ubuntu

El procedimiento para instalar BlueZ en cualquier otra distribución Linux como puede ser Ubuntu es idéntico al expuesto en la sección anterior. La única diferencia que puede haber es que no sea necesario hacer el paso 1 porque no se tenga ninguna versión de BlueZ instalada previamente.

ANEXO B:
MANUAL DE INSTALACIÓN Y CONFIGURACIÓN DE ECLIPSE KURA EN UNA RASPBERRY PI.

En este anexo se va a explicar cómo instalar Eclipse Kura 2.1.0 en una Rasperry Pi 3 y los distintos aspectos de configuración.

B.1 Instalación de Eclipse Kura 2.1.0 en una Raspberry Pi 3 con Raspbian Jessie Lite

Instalar Eclipse Kura para convertir un dispositivo Linux en una pasarela IoT es una tarea muy simple. En la documentación de Eclipse Kura podemos encontrar un tutorial de cómo realizar la instalación de Eclipse Kura en una Raspberry Pi, en concreto aquí: https://eclipse.github.io/kura/intro/raspberry-pi-quick-start.html

Desde una terminal en la Raspberry Pi hacemos lo siguiente:

1. Nos deshacemos de un par de paquetes que no son compatibles con Kura.

```
$ sudo apt-get purge dhcpcd5
$ sudo apt-get remove network-manager
```

2. Instalamos la herramienta **`gdebi`**.

```
$ sudo apt-get update
$ sudo apt-get install gdebi-core
```

3. Instalamos Java 8 en caso de no tenerlo.

```
$ sudo apt-get install openjdk-8-jre-headless
```

4. Descargamos el paquete de Kura y lo instalamos

```
$ cd /tmp/
$ wget download.eclipse.org/kura/releases/2.1.0/kura_2.1.0_raspberry-pi-2-3_installer.deb
$ sudo gdebi kura_2.1.0_raspberry-pi-2-3_installer.deb
```

De esta forma ya tenemos instalado Eclipse Kura en nuestra Raspberry Pi. Podemos reiniciarla ahora y acceder a la interfaz web de Kura desde un navegador, poniendo la dirección IP de nuestra Raspberry. Nos pedirá usuari y contraseña, que por defecto son `admin` y `admin`.

B.2 Configuración de Eclipse Kura

A través de la interfaz web de administración de Kura podemos configurar diferentes aspectos de lo que es ya nuestra pasarela IoT como la configuración de red, la configuración para conectarse a la nube, la configuración de los bundles configurables…

B.2.1 Configuración de red

Podemos configurar los parámetros de red de nuestro dispositivo Kura de la forma tradicional en la que lo hacemos en un sistema Linux o haciendo uso de la interfaz web de Kura.

En la interfaz web de Kura encontramos un apartado llamado **Network** en el que podemos configurar las diferentes interfaces de red nuestro dispositivo (ethernet, wlan…).

A continuación, las figuras 63 y 64 muestran un par de capturas de la configuración de red para la interfaz WiFi del dispositivo.

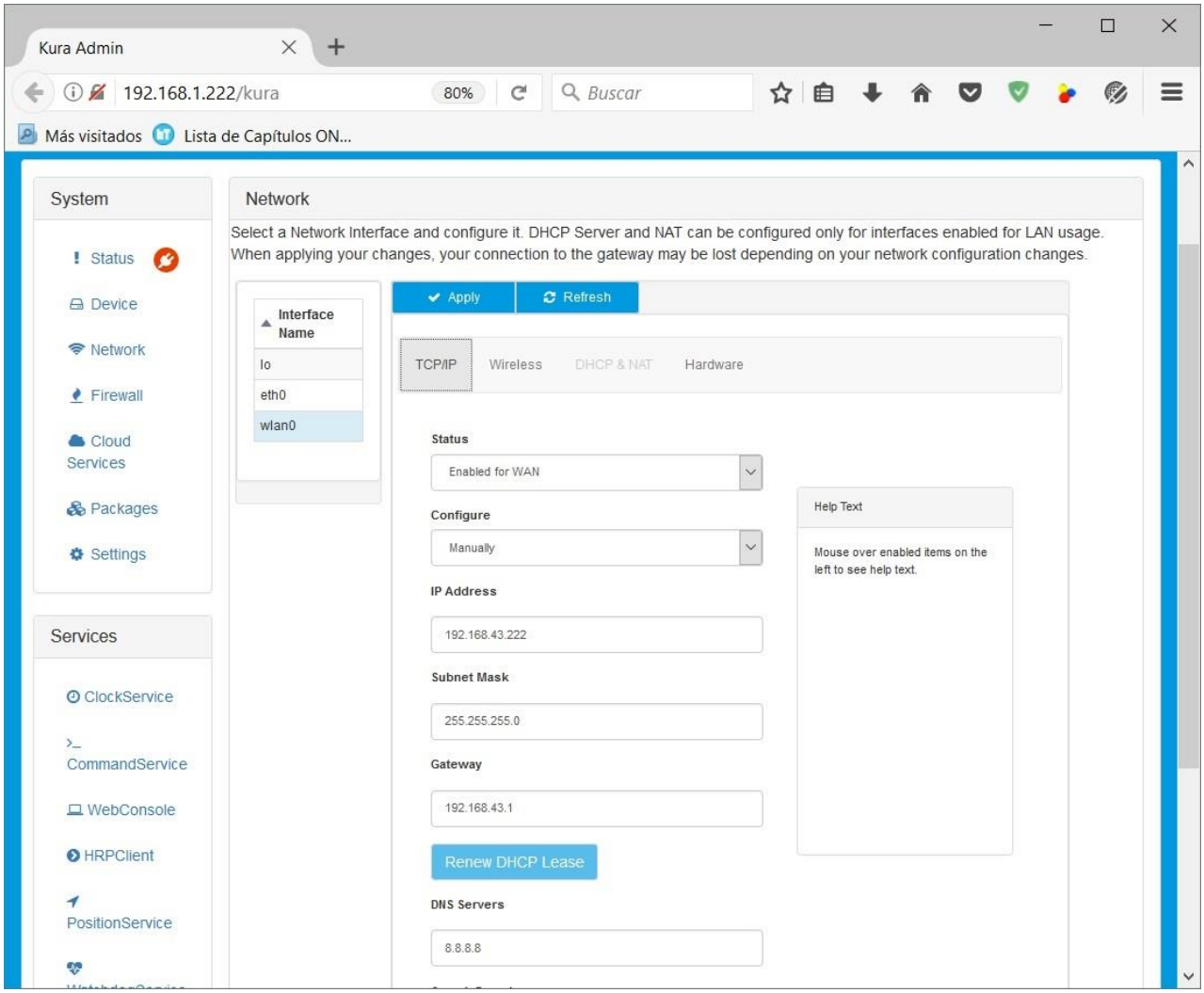

Figura 63: Interfaz web de Kura: Configuración de red (I).

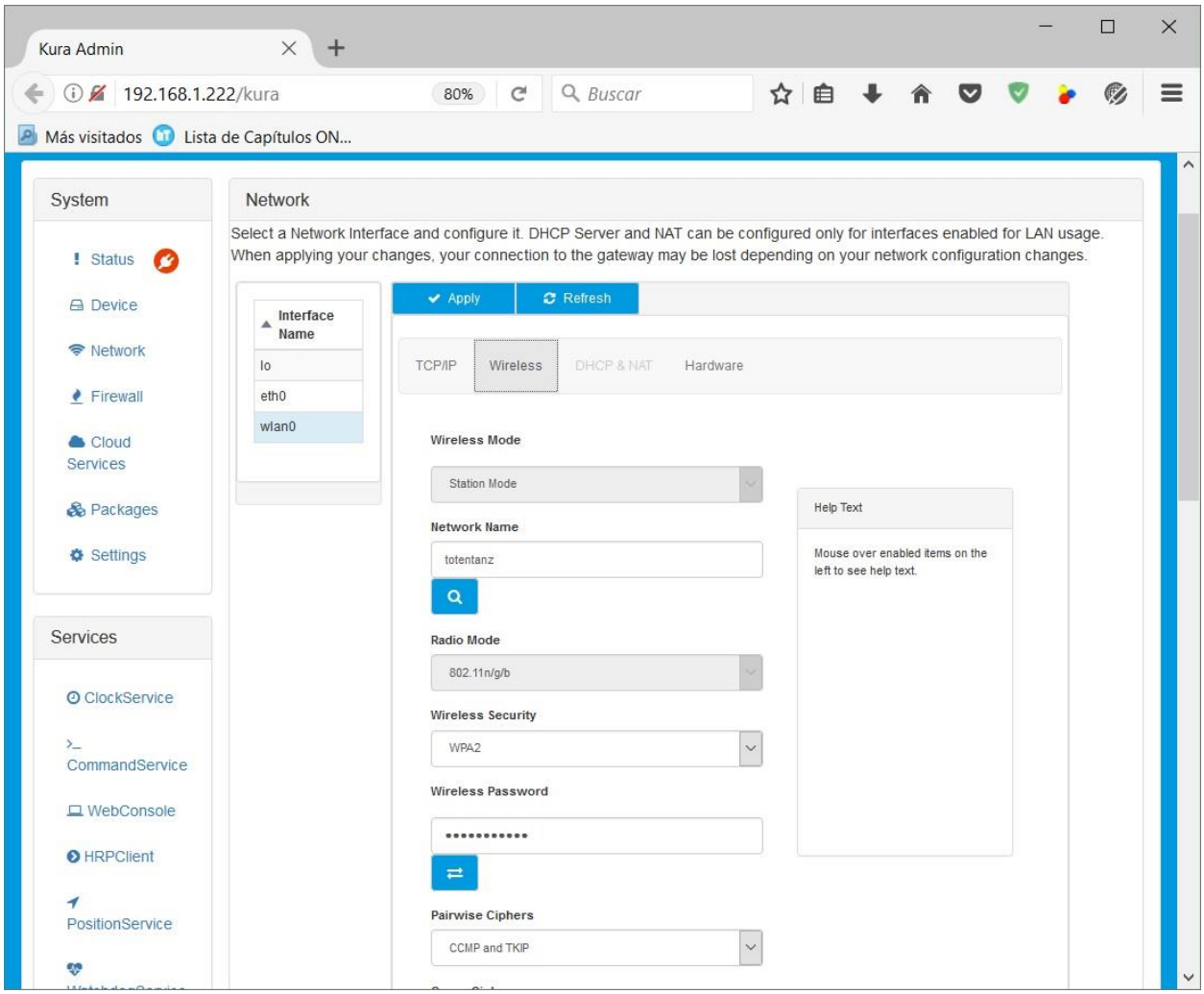

Figura 64: Interfaz web de Kura: Configuración de red (II).

Mediante esta configuración hemos indicado:

- Que queremos que nuestro dispositivo se conecte a una red WiFi cuyo SSID es totentanz (indicamos la contraseña también, por supuesto).
- Que queremos que la dirección IP de la interfaz WiFi sea una configurada manualmente (192.168.43.222) y no obtenida por dhcp.
- La dirección del DNS que vamos a usar (8.8.8.8).
- La dirección de la pasarela (192.168.43.1) y la máscara de subred (255.255.255.0).

B.2.1 Configuración de la nube AWS

Como se ha mencionado durante la memoria, Eclipse Kura nos provee con un servicio CloudService para poder interactuar con la nube y que podemos usar desde nuestro bundles. No obstante, es necesario haberlo configurado antes, para indicar cuál es el endpoint, qué certificado usa el dispositivo, etc.

Esta configuración se realiza desde el apartado **Cloud Services**, y podemos encontrar en la documentación de Eclipse Kura diferentes tutoriales sobre cómo configurarlo para diferentes plataformas cloud, como por ejemplo AWS: https://eclipse.github.io/kura/cloud/kura-aws-cloud.html

A continuación se va a explicar cómo configurar nuestro dispositivo Kura para que se conecte con nuestro endpoint en la nube de Amazon.

Recordemos que en nuestro sistema cada gimnasio en la nube se representa por medio de una 'thing', que tiene su propia identidad (certificado y clave privada) y una política asociada. Y cada dispositivo Kura dentro de un mismo gimnasio se conectaba a la nube con un id de cliente diferente, pero todos haciendo uso de la identidad del gimnasio.

Bien, entonces, para configurar nuestro dispositivo Kura (nuestra Raspberry Pi) para que se conecte con nuestro endpoint en la nube de Amazon usando una identidad concreta, debemos:

1. Crear un almacén Java de claves (Java keystore) en el dispositivo e incluir en él el certificado raíz usado por la plataforma IoT de Amazon. Este certificado no forma parte de la identidad de nuestro dispositivo, sino de la de Amazon. Para ello, desde una terminal en nuestra Raspberry Pi:

   - Obtenemos el certificado raíz de AWS IoT.

```
$ curl https://www.symantec.com/content/en/us/enterprise/verisign/roots/VeriSign-Class%203-Public-Primary-Certification-Authority-G5.pem > /tmp/root-CA.pem
```

   - Creamos el Java keystore con nombre **cacerts** y contraseña **password** y le añadimos el certificado que acabamos de descargar.

```
$ sudo mkdir /opt/eclipse/security

$ cd /opt/eclipse/security

$ sudo keytool -import -trustcacerts -alias verisign -file /tmp/root-CA.pem -keystore cacerts -storepass password
```

2. Configurar el apartado SSL y la identidad de nuestro dispositivo. Desde el apartado **Settings**:

   2.2. En la pestaña **SSL Configuration** indicamos que vamos a usar la versión TLSv1.2, la ruta de nuestro Java keystore y su contraseña. A continuación, en la figura 65 se muestra una captura de esta configuración.

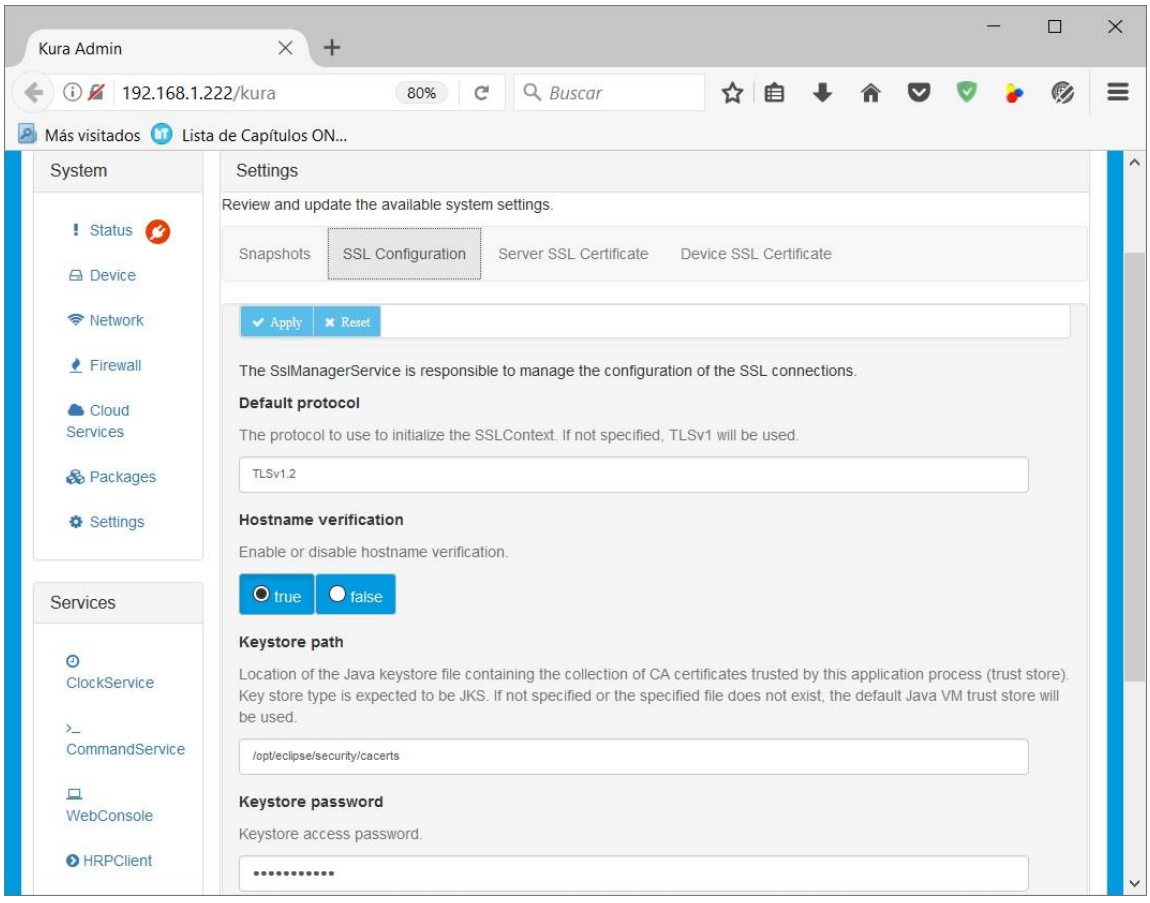

Figura 65: Interfaz web de Kura: Configuración SSL.

2.3. En la pestaña **Device SSL Certificate** añadimos una identidad para nuestro dispositivo. Esto es, el certificado y clave privada de nuestra 'thing' en Amazon, que en nuestro sistema representa a un gimnasio.

A continuación, la figura 66 muestra una captura de esta pestaña.

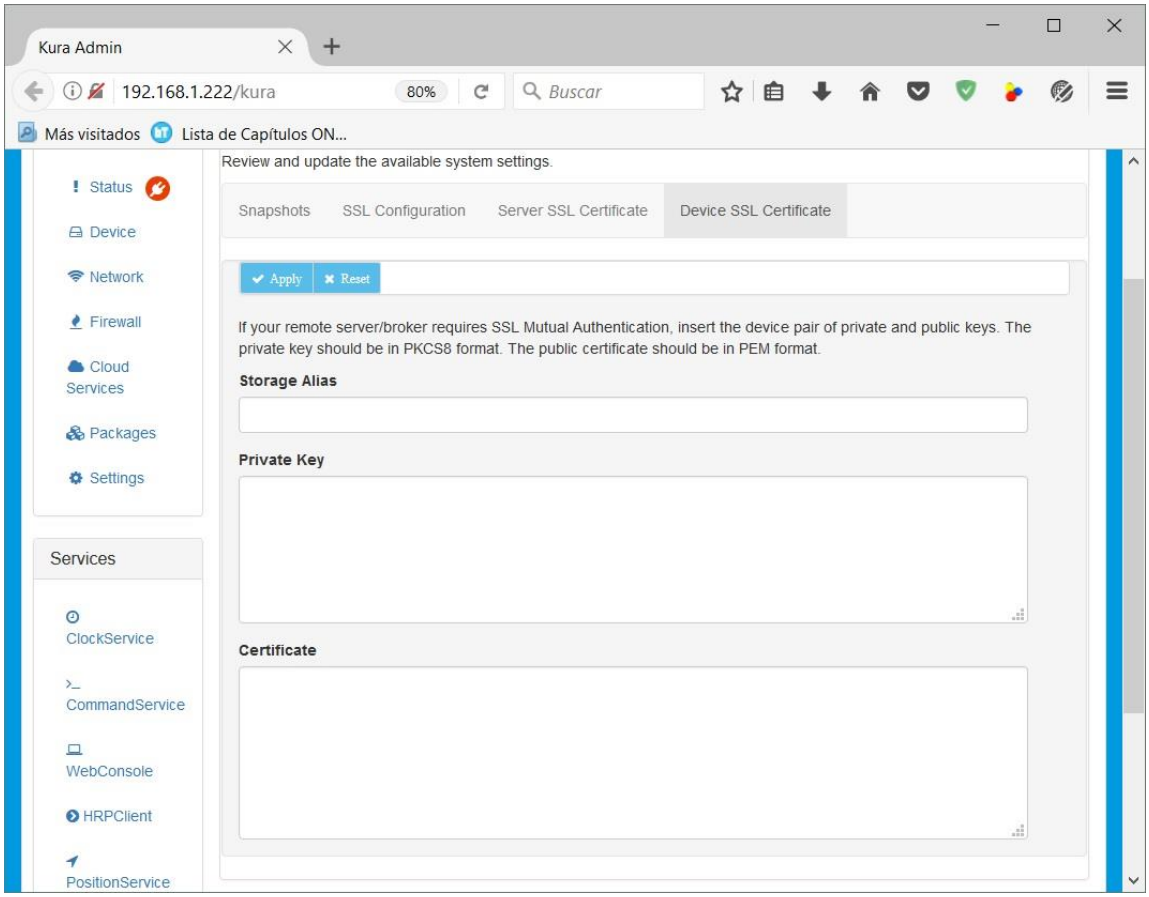

Figura 66: Interfaz web de Kura: Configuración SSL.

En el campo **Storage Alias** insertamos un alias para la identidad que vamos a añadir, como por ejemplo mi_identidad. En el campo **Certificate** pegamos el contenido de nuestro certificado y en el campo **Private Key** debemos pegar el contenido de nuestra clave privada pero en formato PKCS8.

Para pasar nuestra clave a formato PKCS8 podemos ejecutar desde una terminal Linux lo siguiente:

```
$ openssl pkcs8 -topk8 -inform PEM -outform PEM -in MiClavePrivada.pem.key
-out MiClavePrivadaPKCS8.pem -nocrypt
```

3. Una vez hemos configurado una identidad para nuestro dispositivo, toca configurar el apartado de conexión con la nube: **Cloud Services**.

   En este apartado existe una lista de los servicios cloud implementados. Como ya dijimos, Eclipse Kura implementa uno y nos permite usarlo desde nuestros bundles, por lo que este es el único que aparece en la lista. Lo seleccionamos y vemos que existen tres pestañas: **CloudService**, **DataService** y **MqttDataTransport**.

   3.1. Pestaña CloudService

   Recordemos que en nuetro bundle hacemos uso del servicio **`DataService`** y no del **`CloudService`**, por lo que en nuestro caso sólo nos va a interesar que los parámetros mostrados en la figura 67 estén configurados como en ella se aprecia.

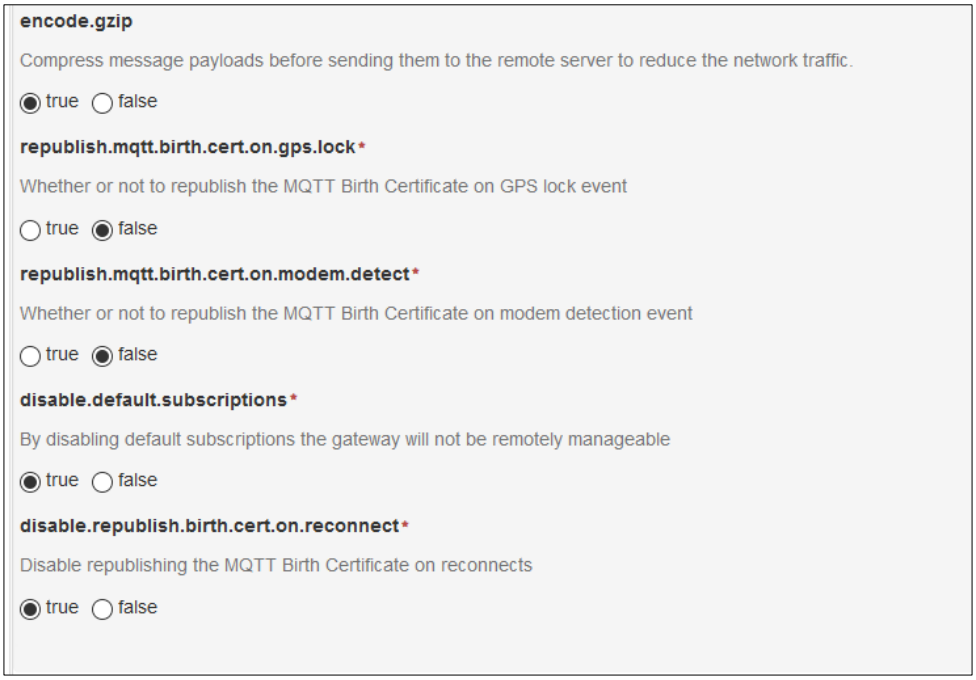

Figura 67: Interfaz web de Kura: Configuración Cloud Service, pestaña CloudService.

3.2. Pestaña DataService

En esta pestaña podemos dejar los valores por defecto, excepto el parámetro que determina si el dispositivo se conecta o no automáticamente a la nube al iniciarse. Este lo pondremos a verdadero porque sí queremos que se conecte a la nube de forma automática.

3.3. Pestaña MqttDataTransportProtocol

Es aquí donde vamos a configurar el endpoint de nuestra nube de Amazon, el id de cliente y el certificado que va a usar el dispositivo al conectarse. Para ello, configuramos los siguientes parámetros:

- **broker-url**

  Aquí indicamos la dirección de nuestro endpoint, la cual podemos obtener desde la consola AWS IoT de nuestra cuenta de AWS.

  El formato será el siguiente: `mqtts://xxxxxxxxxx.iot.<region>.amazonaws.com:8883/`

- **client-id**

  Aquí indicamos cuál va a ser el id de cliente que va a usar el dispositivo al conectarse a la nube. Recordemos que debe ser distinto para cada dispositivo.

- **protocol-version**

  Indicamos la versión 3.1.1 de MQTT.

- **ssl.certificate.alias**

  Aquí indicamos el alias de la identidad que vamos a usar y que añadimos anteriormente en el paso 2.3.

El resto de parámetros pueden quedar vacíos o tomar su valor por defecto.

Una vez hecho esto, ya tenemos la configuración de la nube lista para nuestro sistema.

REFERENCIAS

[1] Garmin, «ANT+ in the Gym». [En línea]. Disponible: http://www8.garmin.com/intosports/antplus.html

[2] Bluetooth SIG, «Bluetooth Core Specification». [En línea]. Disponible: https://www.bluetooth.com/specifications/bluetooth-core-specification

[3] Adafruit, «Introduction to Bluetooth Low Energy», 2014. [En línea]. Disponible: https://cdn-learn.adafruit.com/downloads/pdf/introduction-to-bluetooth-low-energy.pdf

[4] Bluetooth SIG, «Generic Attribute Profile (GATT) Specification». [En línea]. Disponible: https://www.bluetooth.com/specifications/generic-attributes-overview

[5] Eclipse Kura Documentation, «Introduction > Overview». [En línea] .Disponible: http://eclipse.github.io/kura/intro/intro.html

[6] OSGi Alliance, «Developer». [En línea]. Disponible: https://www.osgi.org/developer/

[7] Margaret Rouse, TechTarget, «OSGi (Open Service Gateway Initiative) ». [En línea]. Disponible: http://searchnetworking.techtarget.com/definition/OSGi

[8] Wikipedia, «OSGi». [En línea]. Disponible: https://en.wikipedia.org/wiki/OSGi

[9] OSGi Alliance, «OSGi Core Release 6». [En línea]. Disponible: https://osgi.org/javadoc/r6/core/index.html

[10] Diccionario de la Real Academia de Ingeniería, «Computación en la nube». [En línea]. Disponible: http://diccionario.raing.es/es/lema/computaci%C3%B3n-en-la-nube

[11] Amazon Web Services Documentation, «What Is AWS IoT?». [En línea]. Disponible: http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[12] Amazon Web Services Documentation, «What Is Amazon DynamoDB?». [En línea]. Disponible: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

[13] Amazon Web Services Documentation, «What Is Amazon Cognito?». [En línea]. Disponible: http://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html

[14] Moon Technolabs, «Apple Vs Android – A comparative study 2017». [En línea]. Disponible: https://www.moontechnolabs.com/apple-vs-android-comparative-study-2017/

[15] Amazon Web Services Documentation, «AWS IoT Policies». [En línea]. Disponible: http://docs.aws.amazon.com/iot/latest/developerguide/iot-policies.html

[16] IBM, «Example: OSGi bundle manifest file». [En línea]. Disponible: https://www.ibm.com/support/knowledgecenter/SSEQTP_8.5.5/com.ibm.websphere.osgi.doc/ae/ra_bundle_mf.html

[17] Eclipse Kura Documentation, «Development > Hello Word Example». [En línea] .Disponible: http://eclipse.github.io/kura/dev/hello-example.html#create-component-class

[18] Eclipse Kura 2.1.0 API Documentation, «Interface ConfigurableComponent». [En línea]. Disponible: http://download.eclipse.org/kura/docs/api/2.1.0/apidocs/org/eclipse/kura/configuration/ConfigurableComponent.html

[19] Apache Felix Documentation, «Apache Felix Metatype Service». [En línea]. Disponible: http://felix.apache.org/documentation/subprojects/apache-felix-metatype-service.html

[20] Android Developers, «Application». [En línea]. Disponible: https://developer.android.com/reference/android/app/Application.html

GLOSARIO

AWS: Amazon Web Services.

BLE: Bluetooth Low Energy.

Bluetooth LE: Bluetooth Low Energy.

Dispositivo Kura: Dispositivo en el que se ha instalado Eclipse Kura.

IoT: Internet of Things (Internet de las cosas).