ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Universidad Politécnica de Cartagena

Trabajo Fin de Máster

Diseño y desarrollo de una aplicación Android para calibrar un sistema de localización de interiores

AUTOR: Pablo Marco Jornet

DIRECTOR: Alejandro Martínez Sala

Septiembre / 2017

Autor	Pablo Marco Jornet
E-mail del autor	pablomarcojornet@gmail.es
Director	Alejando Martínez Sala
E-mail del director	alejandros.martinez@upct.es
Título del TFM	Diseño y desarrollo de una app Android para un sistema de localización de interiores.
Resumen	Los sistemas de localización de smartphone son de interés para el desarrollo de servicios de localización (navegación, guiado, etc.) en entornos de interior (hospital, supermercado, aeropuerto, universidad, etc.). En los entornos de interior no funciona la tecnología GPS por lo que se necesitan otras tecnologías. Un sistema de localización de interiores de smartphone puede usar las señales wifi y/o bluetooth que recopilan dichos dispositivos para su procesamiento y estimación de la posición en un servidor. Un Servidor de localización debe poder recibir datos en cantidad y calidad de forma estructurada y ordenada de diversos usuarios que llevan la app instalada en su smartphone Android. El objetivo del presente proyecto es diseñar e implementar una app Android que recopile datos en cantidad y calidad de las señales wifi y/o bluetooth 4.0 y las envíe a un servidor central. Esta app debe servir para calibrar y testear el desempeño de un sistema de localización.
Titulación	Máster Universitario en Ingeniería de Telecomunicación.
Departamento	Tecnología de la Información y las Comunicaciones.
Fecha de presentación	Septiembre - 2017

Índice

1. Introducción y objetivos 6
1.1 Introducción 6
1.2 Objetivos 7
2. Tecnologías Empleadas 8
2.1 Android 8
2.2 Bases de datos 9
2.2.1 SQLite 9
2.3 Protocolo MQTT 9
2.4 Bluetooth 11
2.4.1 Bluetooth Low Energy (BLE) 12
2.4.2 Método de escaneo Bluetooth en Android 15
2.5 Wifi 16
2.5.1 Método de escaneo wifi en Android 18
3. Implementación 20
3.1 Arquitectura del sistema 20
3.2 Estructura de BBDD empleada 20
3.3 Implementación de la aplicación Android de calibración 22
3.3.1 Clases 24
3.4 Implementación del servidor 34
4. Aplicación de calibración y simulación de rutas 36
4.1 Inicio 36
4.2 Calibración 37
4.3 Rutas 38
5. Servidor para la recepción de mensajes MQTT 41
5.1 Puesta en marcha 41
6. Pruebas 42
6.1 Terminales 42
6.2 Prueba con la interfaz wifi 43
6.2.1 Resultados 43
6.2.2 Conclusiones 43
6.3 Prueba Bluetooth 44
6.3.1 Resultados 44
6.3.2 Conclusiones 44
6.4 Prueba utilizando wifi y bluetooth 45
6.4.1 Resultados 45

6.4.2 Comparación de resultados de las tres pruebas 45
6.4.3 Conclusiones.. 47
7. Conclusiones y líneas futuras 48
7.1 Conclusiones.. 48
7.2 Líneas futuras 48
Bibliografía 50

Índice de figuras

Figura 1. Arquitectura de Android 9
Figura 2. Diagrama MQTT 10
Figura 3. Ejemplo de estructura de topics en MQTT........ 10
Figura 4 Formato de trama bluetooth 11
Figura 5 Canales Bluetooth 12
Figura 6 Canales Bluetooth Low Energy........ 13
Figura 7 Beacon BLE utilizado por el sistema........ 13
Figura 8 Captura de la aplicación para configurar los beacon 14
Figura 9 Pantalla de configuración de uno de los modos del beacon........ 15
Figura 10 Cronograma del método de escaneo BLE en Android 16
Figura 11 Canales wifi en 2,4 GHz 17
Figura 12 Canales wifi en 5 GHz 17
Figura 13 Estructura de la trama wifi 18
Figura 14 Cronograma del método de escaneo wifi en Android 19
Figura 15 Arquitectura del sistema 20
Figura 16 Diagrama de la estructura de la base de datos........ 21
Figura 17 Flujograma de la aplicación........ 23
Figura 18 Primera pantalla de la aplicación 36
Figura 19 Pantalla de introducción de datos de usuario........ 37
Figura 20 Pantalla de listado de zonas........ 38
Figura 21 Pantalla con el listado de rutas 39
Figura 22 Pantalla de creación de rutas........ 39
Figura 23 Pantalla que muestra el listado de zonas de una ruta 40
Figura 24 Información mostrada por el servidor 41
Figura 25 Xiaomi Redmi Note 3 Pro 42
Figura 26 Bq Aquaris M8........ 42

Índice de gráficos

Gráfico 1 Comparativa del número de muestras wifi 46
Gráfico 2 Comparación del comportamiento de la interfaz wifi en la tablet BQ........ 46
Gráfico 3 Comparativa del número de muestras BLE obtenidas 47

Índice de tablas

Tabla 1 Resultados de la prueba wifi 43
Tabla 2 Resultados de la prueba bluetooth 44
Tabla 3 Resultados de la prueba conjunta wifi BLE3........ 45

1. Introducción y objetivos

1.1 Introducción

Actualmente hay varios sistemas de posicionamiento y guiado outdoor ampliamente extendidos y empleados a diario en multitud de ocasiones como son GPS, GLONASS, etc. Estos sistemas llevan años siendo utilizados y su funcionamiento y precisión están más que contrastados. Sin embargo, pasamos mucho más tiempo en entornos indoor y es en el interior de los edificios donde reside el reto aún no resuelto de los sistemas de localización ya que en estos espacios interiores los sistemas mencionados anteriormente no funcionan correctamente.

Debido a esta problemática enunciada en el párrafo anterior, en los últimos años proliferan los estudios e implementaciones de IPS (Indoor Positioning System) para resolver la cuestión de la localización en espacios interiores, no existiendo un estándar definido para el desarrollo de estos sistemas. Se han desarrollado y testeado una amplia variedad de tecnologías y no existe un sistema que prevalezca sobre el resto.

Los algoritmos de los sistemas más extendidos que dan servicios IPS suelen estar basados en fingerprinting wifi, es decir, usan la potencia RSSI de las señales recibidas para modelar el espacio interior y realizar el posicionamiento a partir de las señales wifi que reciba un dispositivo en cada momento.

El auge de estos sistemas basados en la información de las señales wifi se debe principalmente a que no necesitan la instalación de ningún tipo de infraestructura adicional, sino que utilizan la información de los puntos de acceso que ya se encuentran instalados para dar servicio a los usuarios, esta característica hace que la implementación de este tipo de sistemas sea mucho más barata que la del resto de tecnologías, las cuales necesitan la instalación de dispositivos en todo el edificio.

Para la obtención del fingerprinting wifi empleado en los algoritmos IPS se necesita adquirir la RSSI en las distintas ubicaciones a localizar, para ello, se debe realizar una calibración de todo el espacio interior donde se va a utilizar el sistema, y estos datos generados durante la calibración serán utilizados para modelar el edificio y posteriormente, calcular la ubicación de los usuarios que envíen la información wifi captada por sus dispositivos.

Consideramos que la mejor forma de adquirir el fingerprinting wifi es mediante lo que se conoce como calibración “crowsorced”, es decir, una calibración realizada de forma natural que no requiere de acciones específicas ni calibraciones intensivas. Para ello se va desarrollar una aplicación Android que va a recoger esta información de las señales wifi de forma ordenada y estructurada con el fin de poder ser empleada en los distintos algoritmos IPS, además se ha añadido la funcionalidad de captar señales bluetooth que puedan complementar el sistema para mejorar su precisión. La aplicación permitirá adquirir el fingerprinting wifi simplemente andando de forma natural por la ubicación objetivo.

1.2 Objetivos

En este proyecto se pretende desarrollar una aplicación Android capaz de generar datos de las señales wifi y/o bluetooth recibidas de forma estructurada y ordenada que serán utilizados para entrenar a un sistema de localización en interiores (IPS). Para ello se debe desarrollar los siguientes apartados que se pueden considerar como objetivos principales del proyecto:

- Aprender sobre el sistema operativo Android, programación en ese sistema operativo y su entorno de desarrollo.
- Adquirir conocimientos sobre los distintos estándares wifi, la información que proporciona el sistema Android al escanear las frecuencias utilizadas por esta tecnología y conocer el método de escaneo wifi en detalle para entender los procesos de generación y ordenación de los datos recibidos.
- Adquirir conocimientos sobre la versión 4.0 del estándar bluetooth y las funcionalidades relacionadas con esta versión que proporciona el sistema Android
- Desarrollo de aplicación Android para obtener datos de las interfaces wifi y/o bluetooth que puedan ser utilizados para entrenar y testear un sistema de localización en interiores
- Estudio del protocolo MQTT para el envío de datos en tiempo real
- Aprender el funcionamiento de las bases de datos en general y como se usan en Android en particular

2. Tecnologías Empleadas

Para el desarrollo de esta aplicación ha sido necesaria la utilización de diversas tecnologías distintas que nos han permitido alcanzar el objetivo final de este trabajo. En este capítulo vamos a presentar y especificar los detalles y las peculiaridades de estas.

2.1 Android

Android es un sistema operativo diseñado para dispositivos móviles con pantalla táctil, en un primer momento se diseñó para ser utilizado en teléfono móviles, el primer dispositivo que lo incorporó fue el *HTC Dream* (octubre de 2008), pero con el tiempo ha experimentado una gran evolución y actualmente se despliega en una gran variedad de dispositivos móviles: *smartphones, tablets, netbooks, e-books,* reproductores multimedia, televisores e incluso lo podemos encontrar en automóviles.

Es el sistema operativo para dispositivos móviles más extendido en el mercado. Uno de los secretos de su éxito es que Android está basado en Linux, un núcleo de sistema operativo libre, gratuito y multiplataforma lo que permite que cualquier desarrollador pueda crear sus aplicaciones de una manera muy rápida y sencilla utilizando una variación de Java llamada Dalvik. El sistema operativo proporciona todas las interfaces necesarias para desarrollar aplicaciones que accedan a las funciones del teléfono (como el *GPS*, las llamadas, la agenda, etc.) de una forma muy simple y en un lenguaje de programación muy conocido como es Java. De esta forma permite que los costes para lanzar un teléfono o una aplicación son muy bajos.

En la Figura 1 se muestra la arquitectura global del sistema Android.

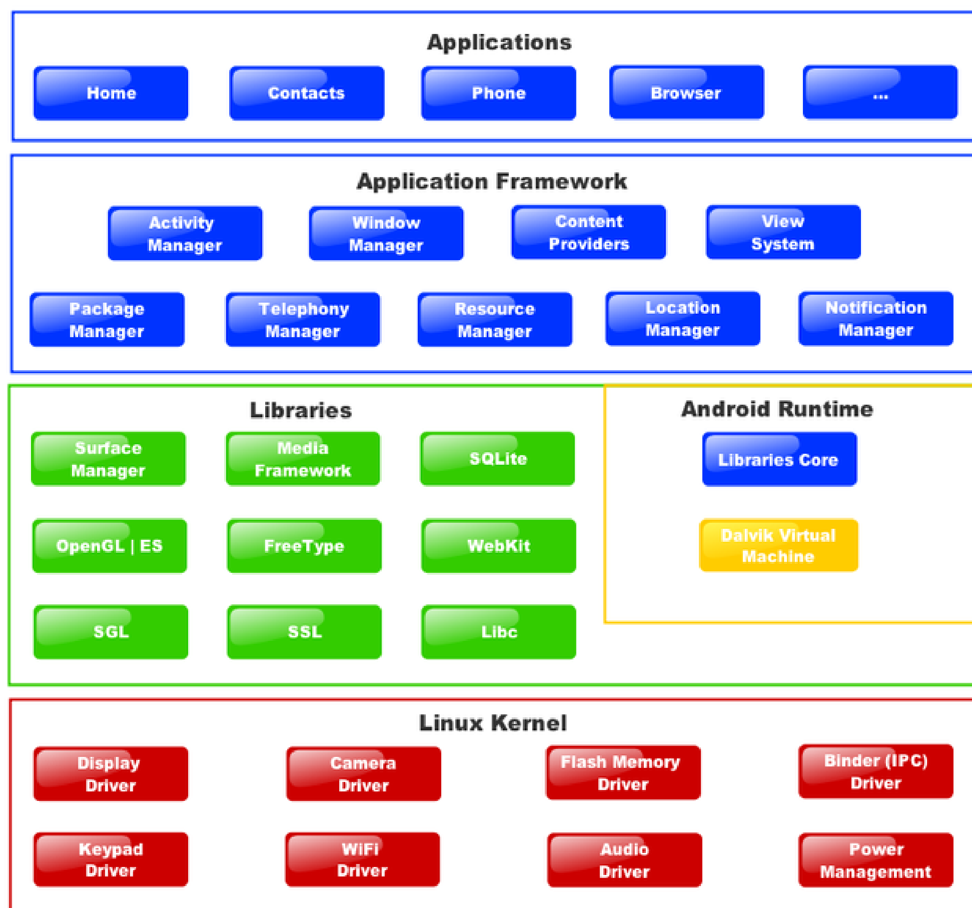


Figura 1. Arquitectura de Android

2.2 Bases de datos

Para implementar la parte de la aplicación encargada de almacenar la información nos hemos decantado por el empleo de base de datos. Concretamente hemos usado *SQLite*, el gestor de bases de datos que incorpora el sistema Android y que introduciremos a continuación.

2.2.1 SQLite

SQLite es una librería escrita en lenguaje C que implementa un sistema de gestión de bases de datos transaccionales SQL y no necesita un proceso separado funcionando como servidor ya que lee y escribe directamente sobre archivos que se encuentran en el disco duro. Además, la base de datos se almacena en un único fichero a diferencia de otros sistemas de gestión de bases de datos que hacen uso de varios archivos. SQLite emplea registros de tamaño variable de forma tal que se utiliza el espacio en disco que es realmente necesario en cada momento. El código de SQLite es de dominio público y libre para cualquier uso, ya sea comercial o privado.

2.3 Protocolo MQTT

El Message Queue Telemetry Transport (MQTT) es un protocolo de comunicaciones *machine-to-machine* (M2M) utilizado en *"Internet of Things"* (IoT). Este protocolo está diseñado para el envío de la información generada por los sensores en el *IoT*, por lo tanto, ocupa muy poco ancho de banda y envía en tiempo real. Por esta característica hemos elegido este protocolo debido a que la aplicación precisa de este envío de los datos en tiempo real.

La arquitectura de MQTT sigue una topología de estrella, con un nodo central que hace de servidor o *broker* con una capacidad de hasta 10000 clientes. El *broker* es el encargado de gestionar la red y de transmitir los mensajes, para mantener activo el canal, los clientes mandan periódicamente un paquete (PINGREQ) y esperan la respuesta del *broker* (PINGRESP).

Se muestra a continuación, en Figura 2, el esquema de empleo de MQTT para un sensor de temperatura típico donde los datos son recibidos en tiempo real en dos sistemas en paralelo a través de un *broker* MQTT.

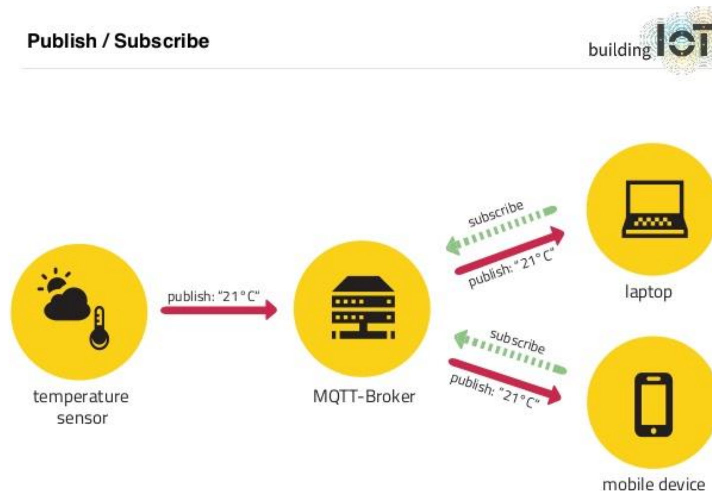


Figura 2. Diagrama MQTT

La comunicación se basa en unos *topics* (temas), que el cliente que publica el mensaje crea y los nodos que deseen recibirlo deben subscribirse a él. La comunicación puede ser de uno a uno, o de uno a muchos. Un *topic* se representa mediante una cadena y tiene una estructura jerárquica. Los niveles de jerarquía se separan por '/'. En la Figura 3 podemos ver un ejemplo de los niveles de jerarquía de los *topics*:

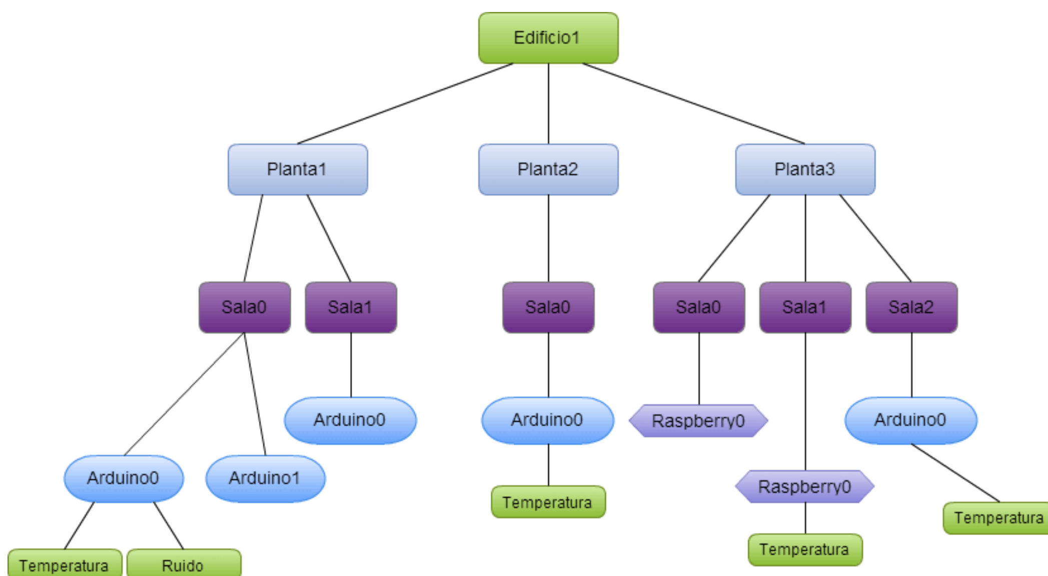


Figura 3. Ejemplo de estructura de topics en MQTT

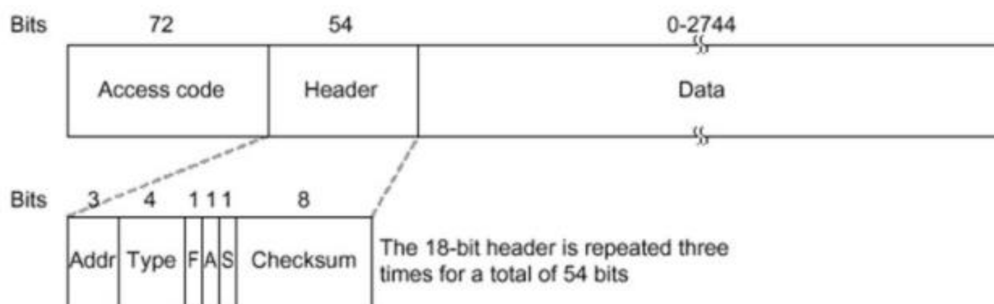
Un cliente puede estar suscrito al *topic* “edificio1/planta1/sala0/Arduino0/Temperatura” y recibirá los mensajes que emita este cliente, pero si se suscribe al *topic “edificio1/Planta1”* recibirá todos los mensajes de los *topics* que estén por debajo en la jerarquía.

Para la utilización del protocolo MQTT en nuestro sistema hemos desplegado el servidor *mosquitto* (https://mosquitto.org/) que realiza el rol de *broker* y recibe los mensajes de todos los clientes que emiten con el *topic* asociado a nuestra calibración. Por otro lado, hemos usado la librería *Paho Android Service* (https://www.eclipse.org/paho/clients/android/) para implementar el cliente MQTT en la aplicación. En el siguiente capítulo de la memoria se detalla el despliegue y la implementación tanto del servidor *mosquitto* como del cliente *Paho* para Android.

2.4 Bluetooth

El término Bluetooth es la denominación comercial y popular del estándar de comunicación inalámbrica IEEE 802.15.1. La primera empresa en investigar esta tecnología fue Ericsson, encargada de liderar un grupo que, con el tiempo, sumó a IBM, Nokia, Microsoft, Motorola y otras compañías que apoyaron el estándar.

Bluetooth es una especificación tecnológica para redes inalámbricas que permite la transmisión de voz y datos entre distintos dispositivos mediante una radiofrecuencia segura (2,4 GHz). Esta tecnología, por lo tanto, permite las comunicaciones sin cables ni conectores y la posibilidad de crear redes inalámbricas domésticas para sincronizar y compartir la información que se encuentra almacenada en diversos equipos. En la la Figura 4 se muestra el formato de las tramas bluetooth.



Access Code: identifica al maestro (puede haber más de uno accesible para el esclavo)

Addr: Dirección (máximo 8 estaciones)
Type: Tipo de trama, corrección de errores y longitud
F: Control de flujo
A: Acknowledgment
S: Num. Secuencia (protocolo de parada y espera)

Figura 4 Formato de trama bluetooth

Emplea ondas de radio de corto alcance, de 2,4 a 2,48 GHz de la banda ISM, lo que le da una compatibilidad universal entre dispositivos Bluetooth, puesto que la banda ISM está disponible

a nivel mundial. Puede alcanzar distancias de 10 metros manteniendo un consumo bajo, pero puede llegar hasta los 100 m si se incrementa el consumo de batería.

Para evitar interferencias entre dispositivos y protocolos que operan en la misma banda de frecuencias, Bluetooth emplea la técnica de salto de frecuencias (FHSS) consistente en dividir la banda en 79 canales de 1 MHz y llevar a cabo 1600 saltos por segundo. En la Figura 5 se muestran los canales en los que se divide el rango de frecuencias en el que se utiliza bluetooth.

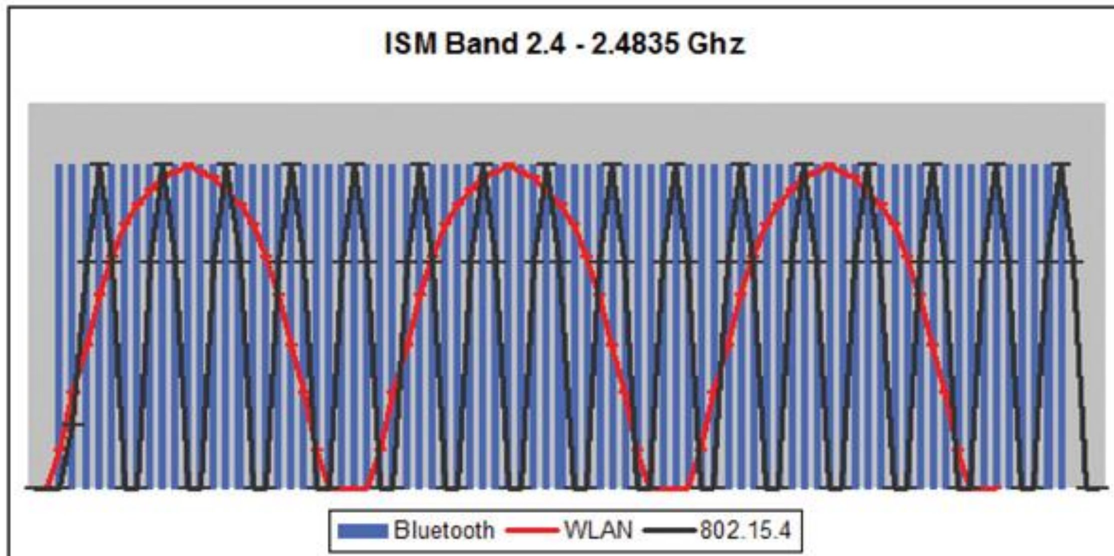


Figura 5 Canales Bluetooth

2.4.1 Bluetooth Low Energy (BLE)

El Bluetooth Low Energy es el nuevo sistema Bluetooth, conocido comercialmente como Bluetooth 4.0, se incorporó en los smartphones en 2013. La principal mejora de esta versión del estándar es la capacidad de enviar datos con un consumo de energía muy bajo, además de mejorar la velocidad de transferencia a 1 Mbps.

El estándar Bluetooth clásico tiene 79 canales mientras BLE tiene 40 (3 de *advertising* y 37 para envío de datos). La separación entre canales también es diferente, debido a estas dos diferencias entre BLE y Bluetooth clásico, éstos son incompatibles entre sí, por lo tanto, no se pueden comunicar. Sin embargo, existen dispositivos *Dual Mode* que soportan las dos tecnologías conmutando los parámetros de modulación y los canales donde se está radiando. En la Figura 6 se muestra la división de canales mencionada anteriormente, los canales verdes corresponden a los tres canales de *advertising*, mientras que los azules son los utilizados para el envío de datos.

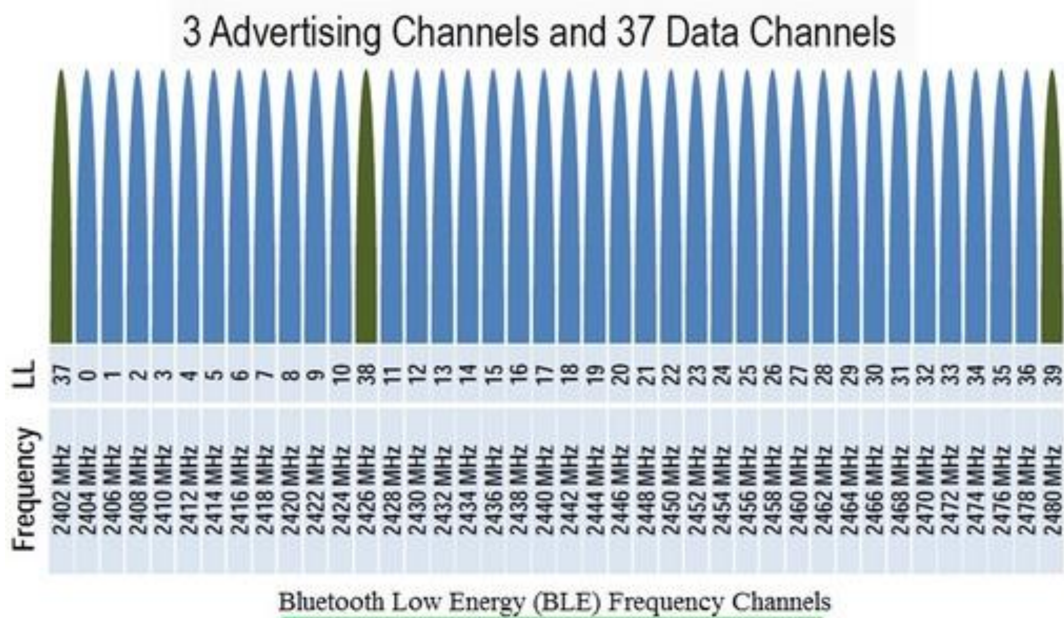


Figura 6 Canales Bluetooth Low Energy

Este nuevo funcionamiento del Bluetooth permite la conexión de miles de dispositivos y con las características antes mencionadas de bajo consumo y velocidad de transmisión, haciendo que la tecnología Bluetooth juegue un papel muy importante en el "Internet de las cosas".

2.4.1.1 Beacon BLE

Un *beacon* es un dispositivo de bajo consumo que emite una señal broadcast, en la Figura 7 se muestra el tipo de *beacon* utilizado en el sistema. Por lo general son suficientemente pequeños para fijarse en una pared o mostrador. Utiliza conexión bluetooth de bajo consumo (BLE) enunciado anteriormente para transmitir mensajes o avisos directamente a un dispositivo móvil sin necesidad de una sincronización de los aparatos. En su modo habitual de funcionamiento, la señal es captada por estos dispositivos y se transmite a un servidor en la nube a través de Internet donde se almacena y procesa la información para llevar a cabo análisis más detallados.

Figura 7 Beacon BLE utilizado por el sistema

Aunque nuestro sistema de localización en espacios interiores se basa en las señales wifi para estimar la posición, vamos a utilizar este tipo de *beacon* para proporcionar una información

extra en ciertas zonas que sean especiales o donde la señal wifi no sea suficiente para estimar la posición.

Además, estos dispositivos permiten ser configurados en varios modos, con distintos intervalos de anuncio y diferentes potencias de trasmisión, lo que nos proporciona una gran versatilidad estableciendo varios radios de la acción del *beacon* e incluso nos ofrecen la opción de enviar una pequeña porción de datos que pueden ser leídos y utilizados gracias a unas librerías específicas para ello.

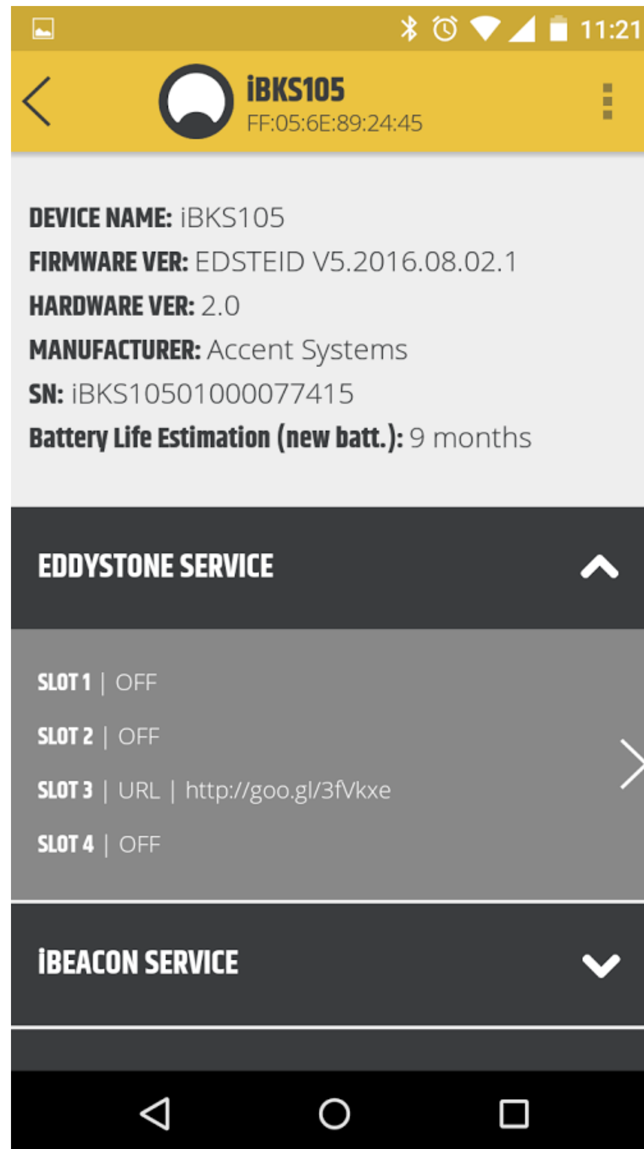


Figura 8 Captura de la aplicación para configurar los beacon

Como podemos observar en la Figura 8, donde se muestra la aplicación de configuración de los *beacon*, éstos soportan dos estándares: *Eddystone Service* (Google) e *IBeacon* (Apple). El primer estándar permite el despliegue de cuatro modos distintos mientas que *IBeacon* solo soporta dos modos. En la siguiente figura (Figura 9) se muestra la pantalla de configuración de un *beacon* en un modo, con todos los parámetros disponibles para ser ajustados.

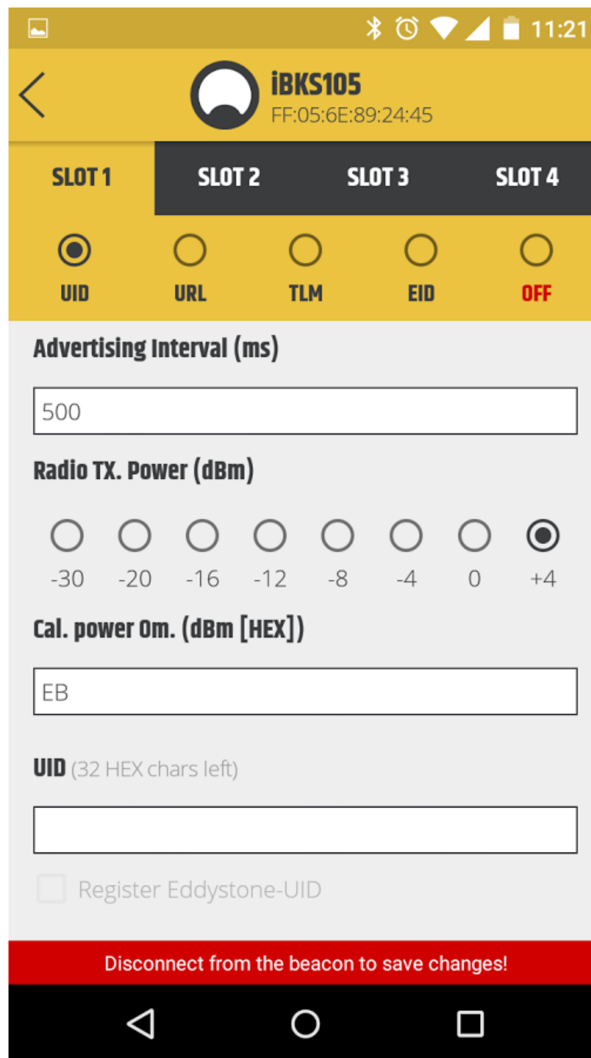


Figura 9 Pantalla de configuración de uno de los modos del beacon

Android proporciona los métodos y las clases necesarias para la utilización de la tecnología Bluetooth Low Energy que nuestra aplicación usará para escanear e identificar los *beacons.* Hay que señalar que hay muchos terminales que, aunque utilicen las últimas versiones de Android, no disponen del hardware compatible con BLE por lo que esta tecnología solo la podremos usar en *smartphones* de gama media-alta.

2.4.2 Método de escaneo Bluetooth en Android

En el contexto de nuestra aplicación de calibración es importante analizar como este sistema operativo detecta los dispositivos bluetooth cercanos a nuestro terminal y que diferencias existen con la forma que tiene de escanear los puntos de acceso wifi que se detectan. En la Figura 10 se muestra el funcionamiento del escaneo BLE en terminales Android.

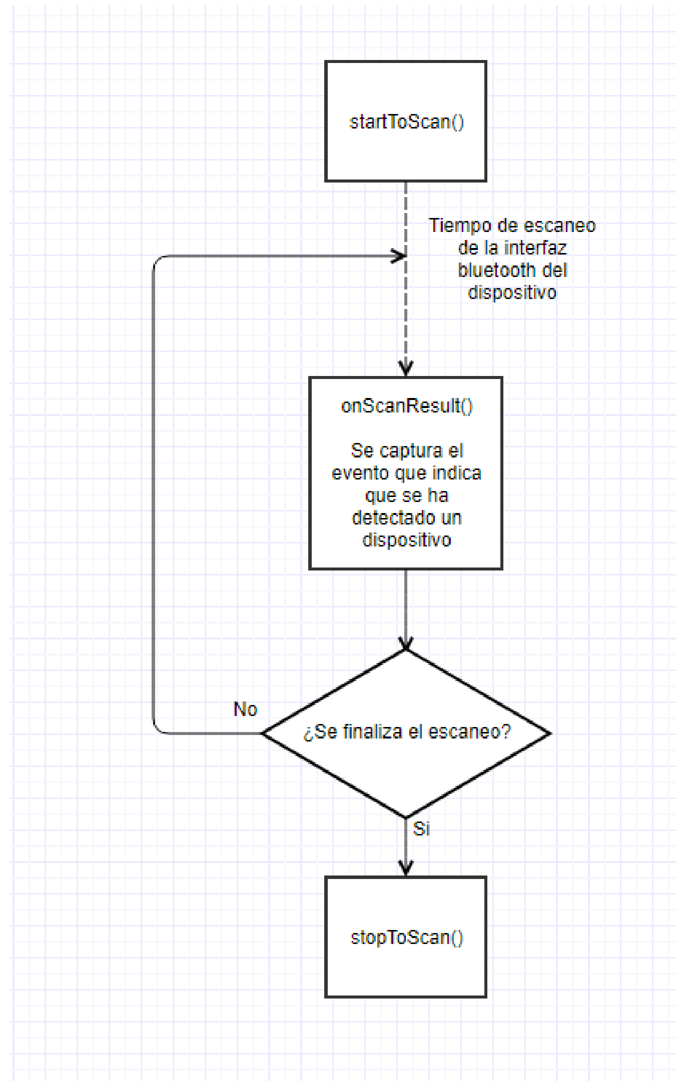


Figura 10 Cronograma del método de escaneo BLE en Android

Para el escaneo *BLE* existe un *callback* dedicado a recoger los eventos generados por la interfaz bluetooth. Para comenzar el escaneo se realiza una llamada al método *startScan* y cuando detecte algún resultado se ejecutará el método *onScanResult* donde se han implementado las tareas correspondientes a la gestión y almacenamiento de la información obtenida. Una vez finalizadas estas tareas el sistema sigue escaneando y ejecutará el método *onScanResult* cada vez que encuentre otro resultado hasta que se realice una llamada al método *stopScan*, el cual detiene el escaneo. Por lo tanto, desde nuestra aplicación podemos controlar el tiempo de escaneo de la interfaz.

Además, Android permite añadir filtros por nombre o por dirección para que obvie los resultados que no nos interesan y solo se ejecute el método *onScanResult* cuando la información del dispositivo detectado sea útil para nuestro propósito.

En el siguiente apartado de la memoria vamos a detallar el funcionamiento del método de escaneo wifi para observar las diferencias que existen entre los dos tipos de interfaces.

2.5 Wifi

Existen diversos tipos de wifi, basado cada uno de ellos en una estándar IEEE 802.11 aprobado:

- Los estándares IEEE 802.11b, IEEE 802.11g e IEEE 802.11n disfrutan de una aceptación internacional debido a que la banda de 2,4 GHz está disponible casi universalmente, con una velocidad de hasta 11 Mbit/s, 54 Mbit/s y 300 Mbit/s, respectivamente. En la Figura 11 se detalla la distribución de los canales wifi en esta frecuencia.

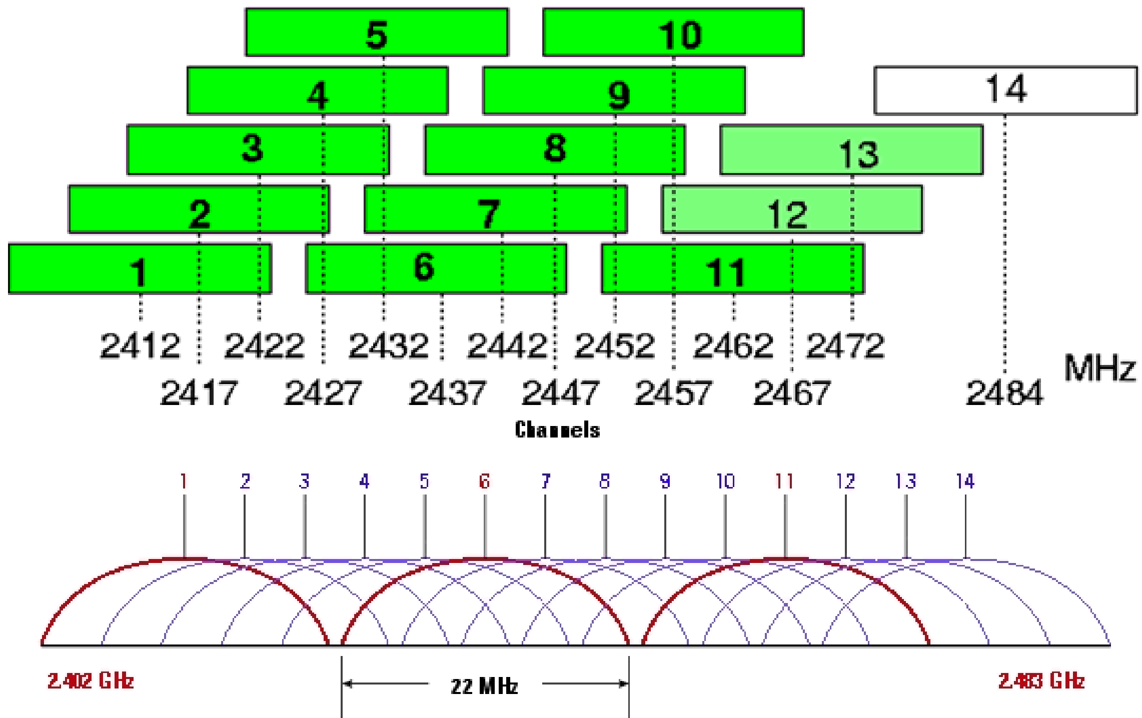


Figura 11 Canales wifi en 2,4 GHz

- En la actualidad ya se maneja también el estándar IEEE 802.11ac que opera en la banda de 5 GHz y que disfruta de una operatividad con canales relativamente limpios. La banda de 5 GHz ha sido recientemente habilitada y, además, no existen otras tecnologías (Bluetooth, microondas, ZigBee, WUSB) que la estén utilizando, por lo tanto, existen muy pocas interferencias. En la Figura 12 se muestran los canales establecidos en la banda de 5 GHz.

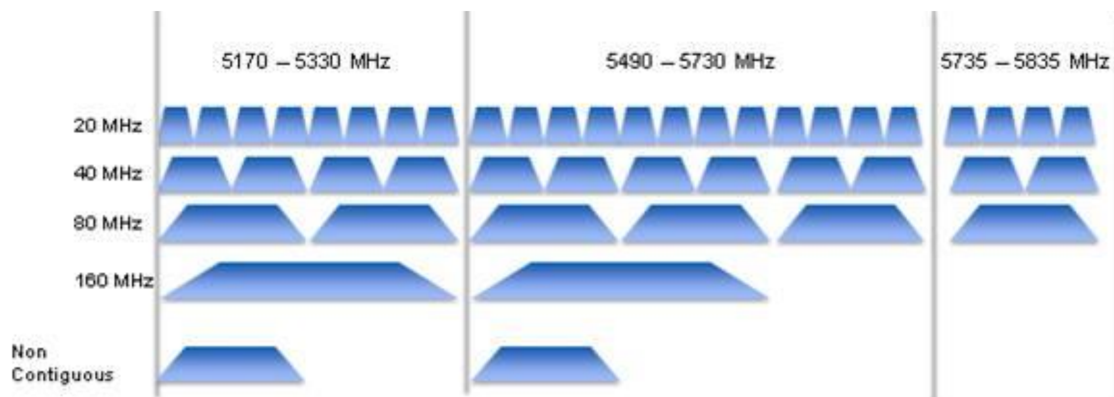


Figura 12 Canales wifi en 5 GHz

Es importante conocer los diversos estándares que existen en la tecnología wifi ya que a la hora de utilizar la aplicación para calibrar un edificio la diferencia de estándares va a influir en gran

medida en los datos obtenidos. Por ejemplo, si usamos un terminal que solo es compatible con los estándares que operan en 2.4 GHz la cantidad de muestras va a ser mucho menor que si utilizamos un dispositivo que pueda escanear tanto la banda de 2.4 GHz como la de 5 GHz. Incluso con terminales que solo trabajen en la banda de 2.4 GHz podemos encontrar diferencias notables ya que es posible que no sean compatibles con los diferentes estándares que coexisten en esa frecuencia.

Por lo tanto, a la hora de calibrar un edifico es importante tener en cuenta el hardware del que disponen los dispositivos que se vayan a utilizar ya que las diferencias entre ellos pueden interferir notablemente en los resultados obtenidos.

2.5.1 Método de escaneo wifi en Android

Como hemos comentado anteriormente en la sección de bluetooth, es importante para nuestra aplicación analizar la forma en la que el sistema operativo Android escanea el espectro wifi y detecta los puntos de acceso cercanos al dispositivo al recibir las tramas *beacon* que envían periódicamente con el objetivo de anunciar su presencia. En la Figura 13 se muestra la estructura del formato de una trama wifi.

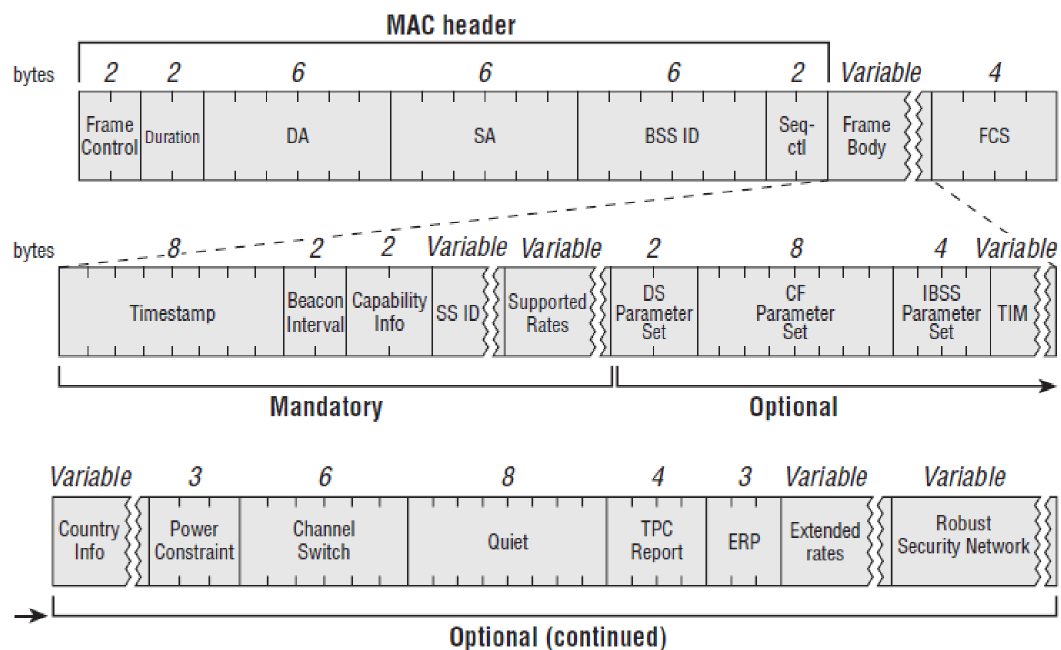


Figura 13 Estructura de la trama wifi

El sistema operativo Android dispone de un objeto *WifiManager* que se encarga de gestionar todas las acciones relacionadas con la interfaz wifi, además, debemos implementar un *BroadcastReceiver* que incluye el método *onReceive* que será ejecutado cuando el sistema tenga la lista de resultado.

Para comenzar el escaneo, la aplicación ejecutará la llamada al método *WifiManager.startScan* y la interfaz wifi comenzará a escanear el espacio radioeléctrico, una vez se haya finalizado el escaneo, se ejecutará el método *onReceive* dónde a través del método *WifiManager.getScanResult* se obtendrán los resultados obtenidos durante ese escaneo. En este caso, la aplicación no puede controlar el tiempo de escaneo, que dependerá del hardware del que disponga el terminal utilizado. El desarrollador solo puede controlar cuando lanza un escaneo, pero no cuánto tiempo durará, a diferencia de la interfaz bluetooth que nos permite

ajustar estos dos parámetros. En la Figura 14 aparece un pequeño cronograma con la secuencia de acciones realizadas en el escaneo wifi explicada anteriormente.

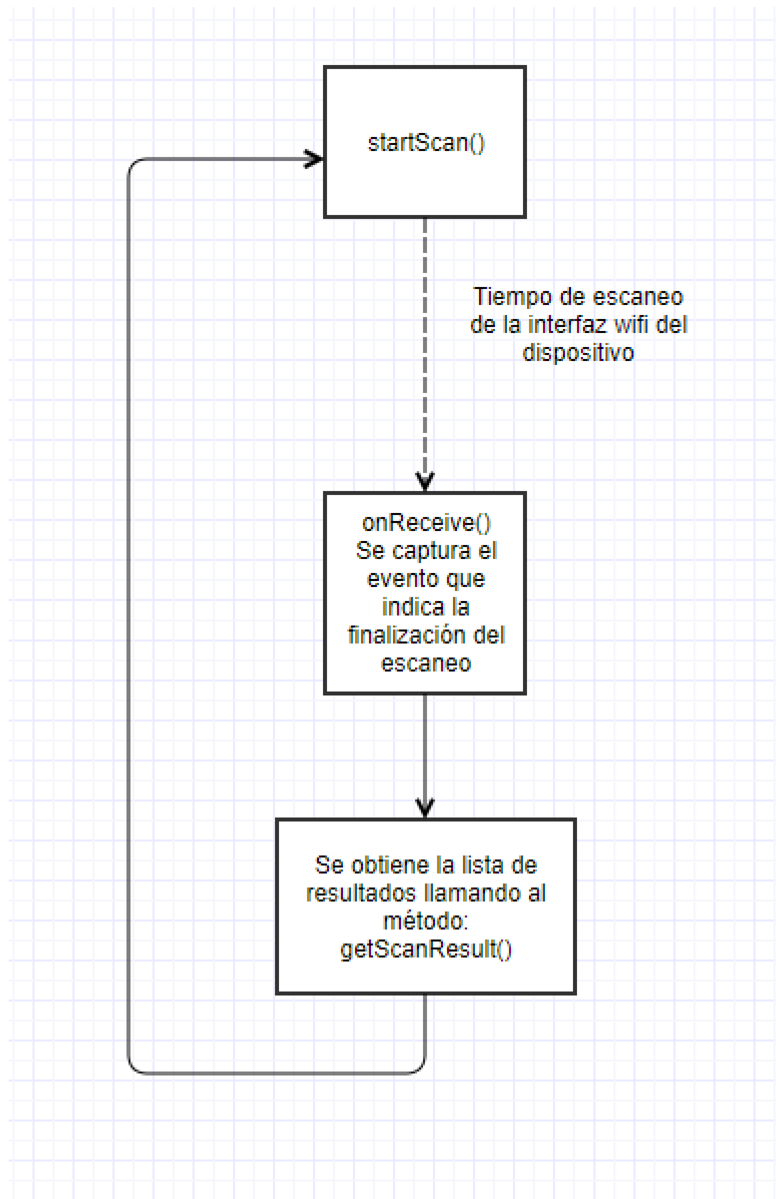


Figura 14 Cronograma del método de escaneo wifi en Android

La otra gran diferencia con la forma de escanear de bluetooth es el acceso a los resultados, mientras que utilizando bluetooth accedemos a los resultados uno a uno según se vayan detectando, en la tecnología wifi el sistema nos devuelve una lista de resultados cuando el escaneo ha finalizado.

3. Implementación

3.1 Arquitectura del sistema

Como se ha comentado en capítulos anteriores, los sistemas IPS de *smartphones* usan las señales wifi y/o bluetooth que reciben en cada instante para enviarlas a un servidor donde el motor de localización estimará la posición mediante el procesamiento de los datos enviados. Por lo tanto, nuestro sistema sigue una arquitectura cliente-servidor, donde los clientes son los terminales móviles que envían los datos recibidos por sus interfaces wifi y bluetooth a un servidor que registra todas las muestras generadas correctamente etiquetadas y las almacena en su base de datos. En la Figura 15 se muestra un esquema de esta arquitectura cliente-servidor.

Cuando la aplicación se inicia, el usuario elegirá el archivo con la lista de zonas del edificio que va a calibra. Dicho fichero tendrá que estar cargado previamente en el directorio "/geoda/building properties/". Una vez definidos los datos del usuario y el fichero de zonas correspondientes se elige el modo en el que queremos adquirir los datos del fingerprinting del edificio:

- **Online:** los datos se envían al servidor en tiempo real.
- **Offline:** los datos se van guardando en una base de datos dentro del terminal móvil. Esta base de datos será enviada al servidor una vez finalizada la sesión de calibración.
- **Route:** se selecciona una ruta que simule el recorrido real de un usuario por el edificio, estas muestras son utilizadas para testear el sistema y se envían una vez finalizada la ruta.

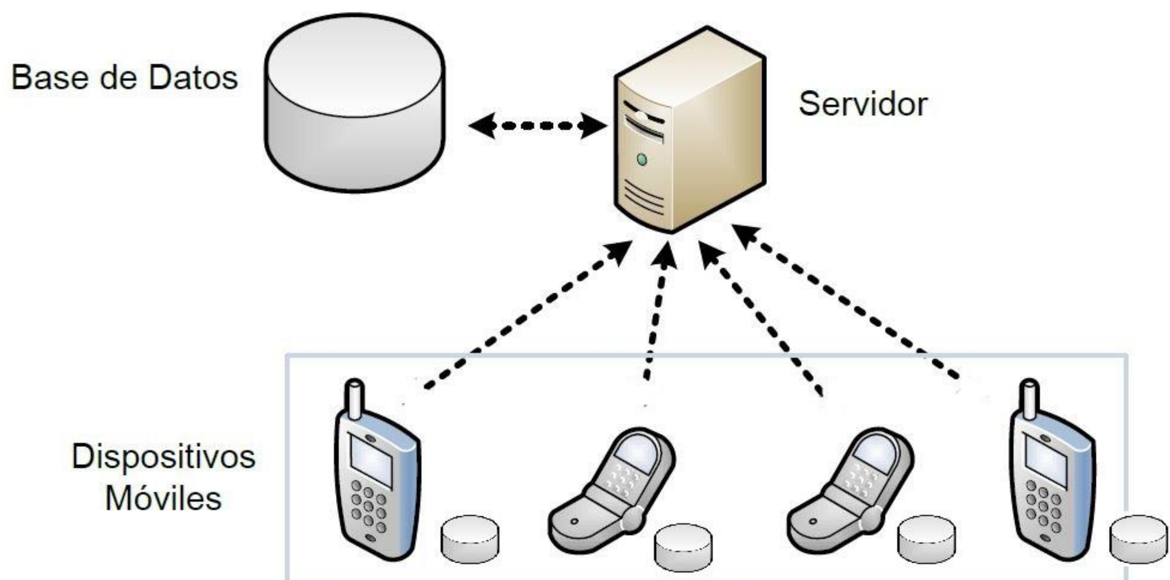


Figura 15 Arquitectura del sistema

3.2 Estructura de BBDD empleada

Como se ha comentado anteriormente, la BBDD empleada en el terminal es SQLite debido a que nos la proporciona el propio sistema Android, por lo que es muy rápida la puesta en marcha de la misma.

Se ha determinado el empleo de BBDD debido al objetivo de esta aplicación, la generación y el envío de datos en cantidad y calidad de forma estructurada y ordenada de diversos usuarios para calibrar y testear un sistema de localización en interiores, se ha diseñado una estructura de tablas que nos permite modelar los edificios que van a componer nuestro sistema y etiquetar todos los datos para satisfacer todas las necesidades del mismo.

Un esquema global de las tablas empleadas se muestra en la Figura 16:

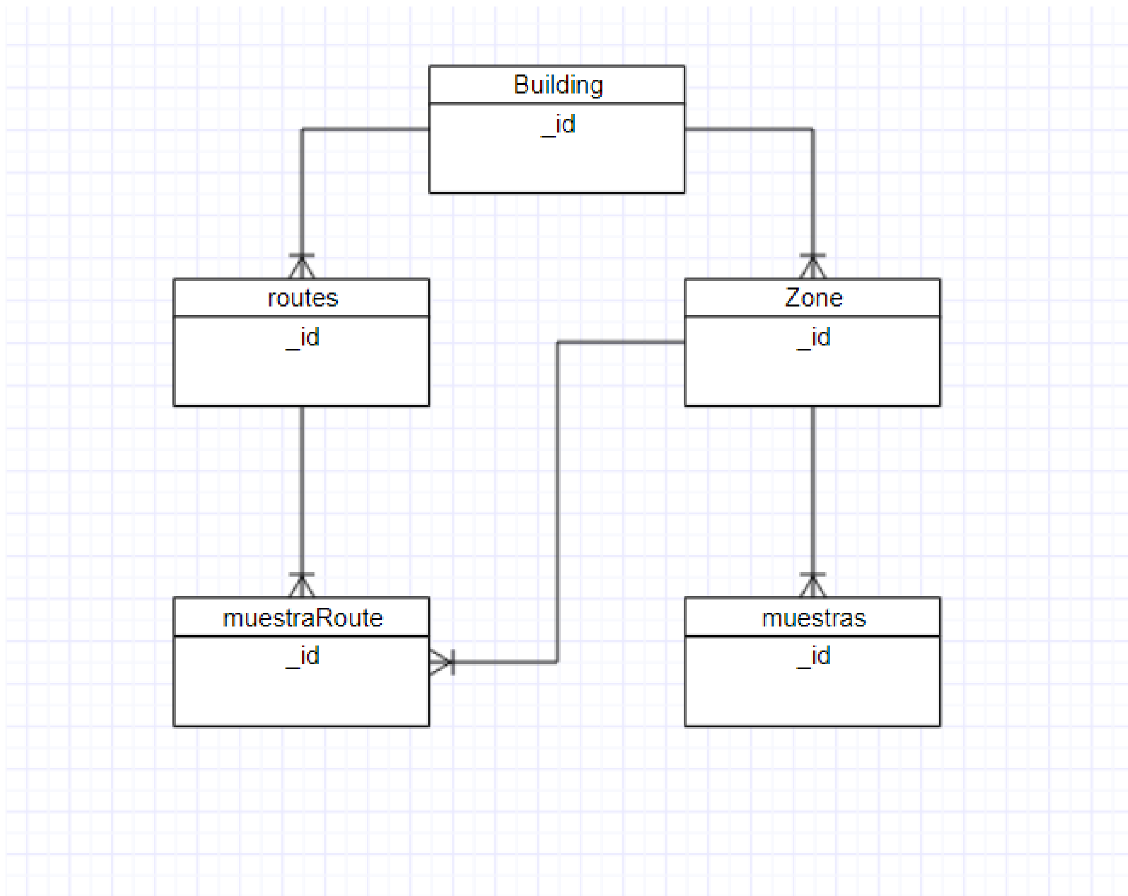


Figura 16 Diagrama de la estructura de la base de datos

En primer lugar, vamos a detallar la tabla *building*, en la que vamos a tener un registro por cada edificio donde se despliega el sistema. Esta tabla consta de los siguientes campos:

- *_id*: identificador del edificio. Este identificador es generado automáticamente por la base de datos.
- *name*: nombre del edificio.
- *timestamp*: timestamp del momento en el que se registra el edificio en la base de datos.

Para llevar a cabo la localización dentro de un edificio necesitamos dividir éste en zonas, en la tabla *zones* vamos a registrar todas las zonas que componen los diferentes edificios que tenemos en la tabla anterior. Los campos de la tabla *zones* son:

- *_id*: identificador de la zona. Este identificador es generado automáticamente por la base de datos.
- *idbuilding*: identificador del edificio al que pertenece la zona.
- *name*: nombre de la zona.

- *timestamp*: timestamp del momento en el que se registra la zona en la base de datos.
- *code*: código identificador de la zona.
- *count*: número de muestras obtenidos en esta zona.
- *time*: duración del período de obtención de muestras en esta zona.

Para almacenar las muestras que se obtienen en las sesiones de calibración utilizaremos la tabla muestras, que consta de los siguientes campos:

- *_id*: identificador de la muestra. Este identificador es generado automáticamente por la base de datos.
- *idzona*: identificador de la zona donde se ha obtenido la muestra.
- *timestamp*: timestamp del momento en el que se captura la muestra.
- *mac*: dirección física del punto de acceso wifi o del BLE captado.
- *rssi*: rssi de la señal recibida.
- *freq*: frecuencia de la señal captada, en el caso de BLE se guarda el valor -1 para indicar que hemos captado la señal de un BLE.

Para llevar a cabo la funcionalidad de rutas de testeo vamos a definir la tabla *routes* con los siguientes campos:

- *_id*: identificador de la ruta. Este identificador es generado automáticamente por la base de datos.
- *zones*: lista de caracteres con los identificadores, separados por comas, de las zonas que componen la secuencia de zonas de la ruta.
- *user*: identificador del usuario que lleva a cabo la ruta.
- *phone*: identificador del dispositivo utilizado para realizar la ruta.
- *mac*: dirección física del dispositivo utilizado para realizar la ruta.

Por último, vamos a especificar los detalles de la tabla *muestrasRoute* utilizada para guardar las muestras generadas durante las rutas de testeo. Los campos de la tabla son:

- *_id*: identificador de la muestra. Este identificador es generado automáticamente por la base de datos.
- *idroute*: identificador de la ruta a la que pertenece esta muestra.
- *sequency*: número de secuencia que permite realizar una ruta varias veces y tener las muestras bien diferenciadas.
- *idzona*: identificador de la zona donde se ha obtenido la muestra.
- *code*: código de la zona donde se ha obtenido la muestra.
- *timestamp*: timestamp del momento en el que se captura la muestra.
- *mac*: dirección física del punto de acceso wifi o del BLE captado.
- *rssi*: rssi de la señal recibida.
- *freq*: frecuencia de la señal captada, en el caso de BLE se guarda el valor -1 para indicar que hemos captado la señal de un BLE.

3.3 Implementación de la aplicación Android de calibración

Para la implementación de la aplicación Android se han creado 21 clases, cuyo flujograma se indica a continuación en la Figura 17:

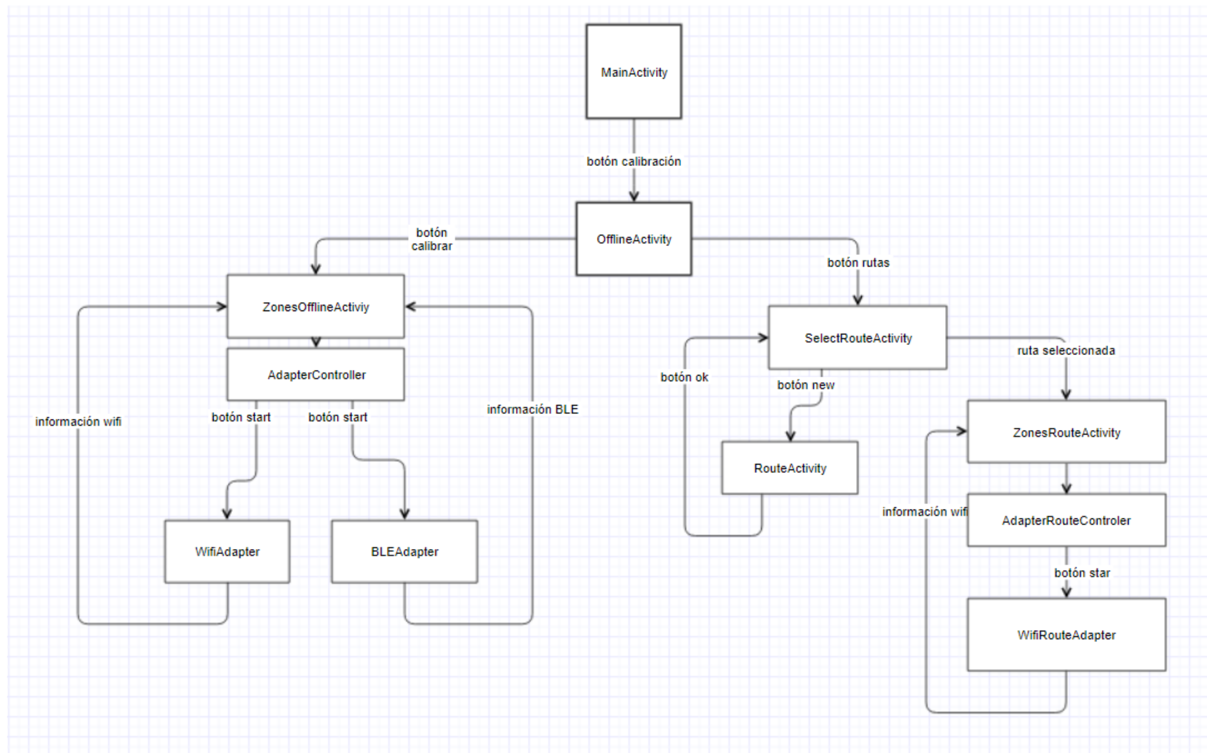


Figura 17 Flujograma de la aplicación

Para el desarrollo e implementación de la aplicación se han utilizado de los patrones de diseño software *singleton* y *MVC* (modelo-vista-controlador), buscando el funcionamiento óptimo de la misma y una estructura clara y ordenada que permita un desarrollo ágil y dinámico.

El patrón *singleton* consiste en un diseño que restringe la creación de objetos a una clase a un único objeto con la intención de garantizar que una clase solo tenga una instancia y, de esta manera, proporcionar un punto de acceso global a ella. El patrón *singleton* se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe ninguna, en el caso de que existiera una instancia previa, el método no creará otra, sino que devolverá ésta.

Por otro lado, el patrón *MVC* consiste en la separación de las clases en tres componentes según su función:

- **modelo**: es la representación de la información con la cual el sistema opera, por lo tanto, gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación. Envía a la vista aquella parte de la información que en cada momento se le solicita para que sea mostrada. Las peticiones de acceso o manipulación de información llegan al modelo a través del controlador.
- **controlador**: responde a eventos (usualmente acciones del usuario) e invoca peticiones al modelo cuando se hace alguna solicitud sobre la información (consultas a la base de datos). También puede enviar comandos a su vista asociada si se solicita un cambio en la forma en que se presenta el modelo, por tanto, se podría decir que el controlador hace de intermediario entre la vista y el modelo
- **vista**: presenta el modelo en un formato adecuado para interactuar (usualmente la interfaz de usuario) por tanto requiere de dicho modelo la información que debe representar como salida.

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo de control que se sigue generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, pulsando un botón).
2. El controlador recibe la notificación de la acción solicitada por el usuario y gestiona el evento que llega, frecuentemente a través de un gestor de eventos (*handler*) o *callback*.
3. El controlador accede al modelo, actualizándolo o modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, añadir una nueva entrada a la base de datos).
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado de las entradas almacenadas en la base de datos).
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

Se dedica el capítulo siguiente a explicar en detalle cada una de estas clases.

3.3.1 Clases

La aplicación se estructura en tres paquetes, cada uno de ellos alberga un tipo de clases totalmente diferenciado:

- **Paquete bbdd:** se corresponde con el modelo dentro del patrón MVC. En este paquete están almacenadas las clases que definen los objetos que guardamos en la base de datos, además de una clase llamada *DataBase.java* en la que creamos todas las tablas de la base de datos. Para cada tabla tenemos una clase que modela los objetos que hemos creado para guardar la información, por lo tanto, tenemos 5 clases asociadas a las tablas que hemos detallado en el capítulo anterior de esta memoria: *BuildingModel.java, MuestraModel.java, MuestraRouteModel.java, RouteModel.java y ZoneModel.java.* Estas clases son muy sencillas, están formadas por las variables que recogen los atributos de cada tabla, un constructor que recibe esos atributos y los asigna a las variables y los métodos *get* y *set* para acceder o modificar estos atributos.
- **Paquete controller:** se corresponde con el controlador dentro del patrón MVC**.** En este paquete se incluyen las clases que podemos definir como controladores o adaptadores. Podemos encontrar las clases *BLEAdapter.java, WifiAdapter.java*, que son las encargadas de configurar y controlar los métodos de escaneo de BLE y de Wifi, así como la obtención de los resultados recogidos en esos escaneos y su correcta ordenación antes de ser almacenados o enviados. Por otro lado, tenemos las clases *AdapterController.java* y *AdapterRouteController.java* que se encargan de modelar y definir los ítems personalizados que utilizaremos para mostrar la lista de zonas de un edificio y su información asociada. Para terminar, tenemos la clase *MqttManager.java*, que utilizamos para gestionar la conexión y el envío de datos mediante el protocolo MQTT, y la clase *DataBaseController.java,* en la cual tenemos implementados todos los métodos que se encargan de leer, extraer o modificar la información almacenada en la base de datos.
- **Paquete geodacalibracion:** se corresponde con la vista dentro del patrón MVC**.** En este paquete se almacenan las clases que se corresponden a cada *Activity* de la aplicación, es decir, que implementan las funcionalidades de cada una de las distintas pantallas o

vistas que componen nuestra aplicación para calibrar un sistema de localización en interiores.

2.2.1.1 Paquete Controller

Clase DatabaseController.java

Siguiendo los estándares de buena programación, para la gestión de SQLite en Android se tiene que emplear un objeto del tipo *SQLiteDatabase* que desempeña el rol de controlador para acceder a la base de datos y realizar todas las operaciones necesarias que van a ser detalladas posteriormente.

En primer lugar, tenemos el constructor que recibe como argumento el contexto de la aplicación (objeto tipo *Context*) que utilizaremos para inicializar el objeto *SQLiteDatabase* y crear la conexión a la base de datos.

Los métodos que componen esta clase son:

- *void open()*: abre la conexión con la base de datos.
- *void close()*: cierra la conexión con la base de datos.
- *long insertBuilding(BuildingModel)*: recibe como argumento un objeto asociado a la tabla de edificios y lo guarda en la base de datos. Además, devuelve el id que la base de datos ha asignado automáticamente a ese registro.
- *long insertZone(ZoneModel)*: recibe como argumento un objeto asociado a la tabla de zonas y lo guarda en la base de datos. Además, devuelve el id que la base de datos ha asignado automáticamente a ese registro.
- *long insertRoute(RouteModel route)*: recibe como argumento un objeto asociado a la tabla de rutas y lo guarda en la base de datos. Además, devuelve el id que la base de datos ha asignado automáticamente a ese registro.
- *long insertMuestra(MuestraModel muestra)*: recibe como argumento un objeto asociado a la tabla de muestras y lo guarda en la base de datos. Además, devuelve el id que la base de datos ha asignado automáticamente a ese registro.
- *long insertMuestraRoute(MuestraRouteModel muestra)*: recibe como argumento un objeto asociado a la tabla de muestras de ruta y lo guarda en la base de datos. Además, devuelve el id que la base de datos ha asignado automáticamente a ese registro.
- *long deleteMuestrasByZone(long idzona)*: elimina las muestras de calibración obtenidas de la zona asociada al id que se recibe como argumento.
- *long deleteMuestras()*: elimina todas las muestras de calibración de la base de datos.
- *void deleteRoute(long idroute):* elimina la ruta asociada al id recibido como argumento de la base de datos.
- *void deleteMuestrasByRoute(long idroute):* elimina de la base datos todas las muestras que pertenecen a la ruta asociada al id que se recibe como argumento.
- *String translateSelectedZones(RouteModel route)*: este método se encarga de extraer la lista de zonas que componen la ruta a la que hace referencia el objeto *RouteModel* recibido como argumento y transforma esa lista de zonas en un *String* con los id de las zonas separados por comas.
- *ArrayList<String> getListZones(String str)*: en este método se recibe un *String* con el listado de los id de las zonas que componen una ruta y a partir de éste, se compone la lista de zonas.
- *long getTime(long id)*: este método obtiene las muestras de calibración de la zona asociada al id recibido como argumento y según los tiempos de la primera y la última

muestra se calcula cuanto tiempo se ha dedicado a obtener muestras en esa zona mientras se realizaba la ruta.

- *int getCountByRoute(long idroute, ZoneModel zone):* la función de este método es la de contar el número de muestra que tenemos de una de las zonas contenida en una ruta determinada mediante una consulta a la base datos. Los argumentos de entrada definen la zona y la ruta que debemos consultar.
- *long getTimeByRoute(long idroute, ZoneModel zone)*: de la misma forma que el anterior, este método obtiene las muestras de una zona asociadas a un ruta concreta y según los tiempos de la primera y la última muestra calcula cuanto tiempo se ha dedicado a obtener muestras en esa zona mientras se realizaba la ruta.
- *Cursor getZonesByBuilding(BuildingModel building)*: mediante una consulta a la base de datos, obtiene la lista de zonas que componen el edificio recibido como argumento.
- *ZoneModel getZoneById(long id)*: devuelve la zona que esta almacenada en la base datos con el id que se recibe como argumento.
- *Cursor getMuestrasByZone(long id)*: devuelve todas las muestras de calibración obtenidas en la zona asociada al id que se recibe como argumento.
- *RouteModel getRoute(Long id):* devuelve la ruta asociada a ese id de la tabla *routes*.
- *BuildingModel getBuildingById(long _id)*: devuelve el edificio asociado a ese id recibido como argumento.
- *Cursor getRoutes()*: devuelve todas las rutas definidas en la base de datos.
- *boolean muestrasIsEmpty()*: comprueba si hay alguna muestra de calibración almacenada en la base de datos.
- *boolean isBuildingCreated(BuildingModel building)*: comprueba si el edificio recibido como argumento esta almacenado en la base de datos.
- *BuildingModel whatBuildingIs(String name)*: este método recibe un *String* con el nombre del edificio que queremos obtener y, mediante una consulta a la base de datos, se recupera el objeto de *BuildingModel* que tiene ese nombre.
- *long updateZone(long id,long time)*: este método es utilizado para actualizar el tiempo de calibración de una zona después de añadir nuevas muestras. En los argumentos del método se definen el id de la zona afectada y el nuevo tiempo de calibración.
- *int getRouteSequency(long idroute)*: en este método se devuelve el número de secuencia que tiene la ruta asociada al id que se recibe como argumento en el instante actual.
- *boolean isRouteStarted(long idroute):* este método se encarga de comprobar si la ruta asociada al *id* que se recibe como argumento ha sido iniciada.

Clase WifiAdapter.java

Para la implementación de esta clase, que se va a encargar de controlar el acceso y la gestión de la interfaz wifi, hemos usado un patrón *singleton* para evitar que durante la ejecución de la aplicación se creen varias instancias de este objeto.

El constructor va a recibir el contexto de la aplicación, que será usado para inicializar una instancia del objeto *DataBaseController* que posteriormente será utilizado para almacenar todas las muestras obtenidas desde la interfaz Wifi.

Los métodos implementados en esta clase son:

- *setWifi()*: este es el método principal del controlador de la interfaz Wifi, se encarga de abrir una instancia de la base de datos (necesaria para guardar las muestras), de activar

el Wifi si estuviera desconectado y de configurar la interfaz WiFi para empezar a escanear. Dentro de este método se encuentra la implementación del objeto *BroadcastReceiver*, cuyas tareas son: recoger le evento del sistema que indica que la información escaneada está lista y ordenarla para ser enviada o almacenada en la base datos.

- *starToScan()*: inicia los escaneos de la interfaz wifi.
- *stopToScan()*: interrumpe los escaneos de la interfaz wifi.

Clase WifiRouteAdapter.java

Esta clase es la encargada de controlar la interfaz Wifi cuando se está realizando la simulación de una ruta. La estructura y la implementación de este controlador es la misma que la explicada anteriormente, la diferencia entre estas dos clases reside en el tratamiento de los datos escaneados por la interfaz wifi. En esta clase *WifiRouteAdapter* los datos se organizan para ser almacenados en la tabla *muestrasRoute*, por la tanto, están relacionados a las rutas ubicadas en la tabla *routes* y son distintos a los datos usados para la calibración, que son los recogidos por el controlador *WifiAdapter*.

Clase BLEAdapter.java

En esta clase hemos implementado el controlador de la interfaz bluetooth, de la misma forma que en los controladores anteriores, el desarrollo sigue el patrón *singleton* para evitar que se creen varias instancias de este controlador durante la ejecución de la aplicación.

Del mismo modo que en el *WifiAdapter*, el constructor va a recibir como argumento el contexto de la aplicación, el cual será usado para crear una instancia del controlador de la base de datos, que será utilizado en el proceso de almacenamiento de las muestras.

Los métodos implementados en esta clase son:

- *setBluetooth()*: este método se encarga de abrir la instancia del controlador de la base de datos, de activar la interfaz bluetooth del *smartphone* y de comprobar si el terminal es compatible con la tecnología *BLE*. Si el dispositivo no fuera compatible con esta versión del protocolo bluetooth, no se recogerán muestras de esta interfaz durante la calibración. Además, añadimos un filtro a los resultados que se obtienen para recibir solo los que corresponden a los *beacons* utilizados por el sistema.
- *starToScanBLE()*: inicia el escaneo de la interfaz bluetooth.
- *stopToScanBLE()*: interrumpe el escaneo de la interfaz bluetooth.
- *onScanResult*: en este controlador implementamos el objeto *ScanCallBack*, este tipo de objeto se encarga de gestionar los eventos generados por la interfaz bluetooth. En nuestro caso, es el método *onScanResult,* que se ejecuta cada vez que se detecta un dispositivo BLE, el que vamos a implementar. Cuando el escaneo devuelva algún resultado, se procesa para obtener la información necesaria y ésta se almacena o se envía en tiempo real.

Clase AdapterController.java

Esta clase se encarga de obtener y mostrar la información que el usuario necesita durante la calibración. Cuando nos disponemos a calibrar un edificio, la aplicación muestra una lista con todas las zonas que componen el edificio, en cada ítem de esa lista se muestra el nombre de la zona, su código identificativo, el número de muestras almacenadas que han sido tomadas en esa zona y el tiempo durante el que se han tomado muestras en la zona. Además, se

implementan dos botones, uno para empezar a calibrar en la zona y otro para borrar las muestras obtenidas en esa zona.

Este controlador se encarga de hacer todas las consultas a la base de datos que son necesarias mediante los métodos implementados en la clase *DataBaseController* que han sido detallados anteriormente.

Por otro lado, también se llama a los métodos implementados en las clases *WifiAdapter* y *BLEAdapter* cuando se gestionan los eventos relacionados con el evento de pulsación sobre el botón que se encarga de comenzar la calibración.

Clase AdapterRouteController.java

Como hemos visto anteriormente en los controladores de la interfaz wifi *WifiAdapter y WifiRouteAdapter*, necesitamos dos implementaciones muy parecidas del mismo controlador debido a que nuestra aplicación genera dos tipos de muestras: calibración y simulación de rutas.

En este controlador se realizan las mismas funciones que en el controlador anterior (*AdapterController*) con la diferencia de que la información que muestra y las consultas a la base de datos que realiza son las relacionadas con la generación de muestras durante la simulación de rutas dentro de un edificio. Además, no se implementa el botón que borra las muestras ya que en las rutas se recorren todas las zonas de forma secuencial y no se contempla el borrado de una de las zonas que componen la ruta.

Clase MqttManager.java

Este último controlador es el que se encarga de controlar y realizar todas las funciones del protocolo *MQTT*, la implementación de esta clase, de la misma manera que los controladores anteriores, sigue un patrón *singleton* para evitar que no se creen varias instancias del cliente *MQTT*, en este caso, esta manera de desarrollar el controlador es muy importante ya que si durante la ejecución se crean varias instancias del cliente *MQTT* surgirán problemas debido a que se enviarán datos duplicados y la calibración no tendría valor.

Los métodos que se implementan en esta clase son:

- *boolean isInit()*: devuelve un valor *boolean* indicando si el cliente *MQTT* ha sido inicializado.
- *init(Context c,String url, String clientId, String topic)*: este método se encarga de inicializar el cliente *MQTT*, con los parámetros de entrada que recibe se configuran las opciones de la conexión.
- *void connect()*: en este método se realiza la conexión con el *broker*. Además, dentro de este método se implementa un objeto del tipo *Callback* que se encarga de escuchar el evento que se genera como resultado del intento de conexión. Si la conexión es exitosa, la aplicación entra en el método *onSuccess()* y el cliente se suscribe al *topic* correspondiente, en caso contrario, entra en el método *onFailure()*, se intenta la reconexión y se comunica al usuario que no ha sido posible la conexión.
- *subscribe()*: el cliente *MQTT* se subscribe al topic del sistema.
- *boolean isConnected()*: devuelve un dato del tipo *boolean* indicando si el cliente se ha conectado al *broker*.
- *boolean isReconnectAvailable()*: devuelve un dato del tipo boolean indicando si es posible la reconexión al broker.
- *void publish()*: método encargado de publicar los mensajes *MQTT*.
- *void close()*: método utilizado para cerrar la conexión.

2.2.1.2 Paquete geodacalibracion

Clase MainActivity.java

Esta clase se corresponde con la primera pantalla de la aplicación, la interfaz gráfica asociada a esta clase es bastante sencilla y está compuesta por los siguientes elementos:

- Botón *Export*: la funcionalidad de este botón es la de exportar la base de datos que recoge toda la información de las calibraciones o las rutas realizadas. Se implementa un objeto del tipo *View.OnClickListener* para capturar el evento de clic sobre este botón. Cuando el botón es pulsado, recogemos la ruta del fichero *.db* donde se encuentra la base de datos, y configuramos un objeto *Intent* que se encarga de obtener este fichero y enviarlo a cualquiera de las aplicaciones de nuestro *smartphone* que permitan el envío de datos.
- Botón *Calibrar*: este es el botón utilizado para comenzar el proceso de calibración. De la misma forma que en el botón *Export*, implementamos el objeto que recoge el evento de pulsación del botón y realiza las acciones correspondientes. En este caso, cuando pulsamos este botón lo primero que hace el método es comprobar si está activo el modo online y en caso afirmativo obtiene la ip y el puerto del *broker MQTT* introducidos por el usuario y guarda la *URL* del mismo en las preferencias de la aplicación para que sea accesible por todas las clases de la aplicación. Por último, se crea un objeto del tipo *Intent* que se encarga de lanzar la siguiente pantalla de la aplicación.
- Botón tipo *switch*: este botón se emplea para activar o desactivar el modo online de la aplicación. Cuando este botón es activado, aparecen dos *EditText* donde el usuario deberá introducir la ip y el puerto del servidor *MQTT* que se va a encargar de recibir los datos. Una vez introducidos estos datos, pulsamos el botón *Calibrar*, que configura la *URL* del servidor para que los controladores *WifiAdapter, WifiRouteAdapter y BLEAdapter* puedan gestionar el envío de datos en tiempo real.

Clase OfflineActivity.java

La función principal de esta pantalla es la de obtener los datos del usuario y la configuración de la calibración o simulación de ruta que va a realizarse. La interfaz gráfica asociada a esta clase está compuesta por varios elementos encargados de recoger los datos:

- *userID*: de este cuadro se recogerá el identificador del usuario que realiza la toma de muestras.
- *phoneID*: en este campo se obtiene el identificador del terminal que se va a utilizar para tomar las muestras.
- *tiempo de escaneo*: este valor solo se utilizará en la simulación de rutas dentro del edificio y define el tiempo en segundos que separa los escaneos y envíos de datos durante la realización de la ruta.
- *ráfagas por escaneo*: este valor, igual que el anterior, solo funciona en la simulación de rutas y se utiliza para indicar cuantas recogidas de datos se van a realizar en cada escaneo.
- *mode:* este elemento es una lista desplegable que muestra tres modos distintos: *Wifi*, *BLE*, *Both*. En este campo se define que tecnologías se van a utilizar para la obtención de datos, interfaz wifi, bluetooth o ambas. Si el terminal no es compatible con la versión bluetooth 4.0, es decir, que no soporta *BLE*, esta lista desplegable solo mostrará la opción de wifi.
- *zones*: este elemento es otra lista desplegable donde vamos a elegir que edificio queremos calibrar. La implementación de esta clase se encarga de acceder al directorio

/geoda/building properties/ del terminal. Este directorio se crea la primera vez que se ejecuta la aplicación y es donde el usuario debe almacenar los ficheros con la lista de zonas que componen cada edificio, mediante esta lista desplegable definimos el edificio que vamos a calibrar.
- *botón calibrar*: este botón se utiliza para elegir las funciones de calibración.
- *botón route:* este botón se utiliza para elegir las funciones de simulación de rutas dentro del edificio.

Cuando el usuario ha proporcionado toda la información necesaria para llevar a cabo todas las funciones de la aplicación, esta clase se encarga de obtener estos datos y de la gestión de los mismos con los siguientes métodos:

- *setLayout()*: este método es el encargado de inicializar todos los objetos que se asocian a los elementos de la interfaz gráfica y de obtener el listado de ficheros de zonas que hay almacenados en el terminal para rellenar la lista desplegable *zones*. Además, este método se encarga de identificar si se ha producido un cambio de edificio en la aplicación.
- *void getSharedPreferences()*: las funciones de este método son: obtener los datos almacenados en las preferencias de la aplicación y rellenar los elementos de la interfaz gráfica con los valores que se establecieron la última vez que se utilizó la aplicación.
- *File createFolder()*: este método devuelve un objeto *File* con la referencia al directorio */geoda/building properties/* donde se encuentran los ficheros con los listados de zonas de los edificios. Si el directorio no existiera, el método se encarga de crearlo.
- *void getUserInput():* este método se encarga de leer los datos introducidos por el usuario y actualiza las variables de la clase asociadas a estos valores.
- *void setSharedPreferences()*: en este método la aplicación obtiene los valores introducidos por el usuario mediante el método *getUserInput()* y los guarda en las preferencias para que sean accesibles por otras clases o en sucesivas ejecuciones de la aplicación.
- *botón calibrar*: en este punto se detalla la implementación del método que captura el evento de pulsación sobre el botón calibrar. En este método, en primer lugar, se llama a los métodos *getUserInput* y *setSharedPreferences*, después, se comprueba si está activado el modo online y en caso afirmativo, se inicializa el cliente *MQTT* mediante el objeto *MqttManager* y se realiza la conexión con el *broker*. Por último, se comprueba el fichero de zonas seleccionado y se arranca la siguiente pantalla.
- *botón rutas*: en este punto se detalla la implementación del método que captura el evento de pulsación sobre el botón rutas. Este método sigue la misma estructura que el anterior, primero se llama a los métodos *getUserInput* y *setSharedPreferences*, después se comprueba el fichero de zonas seleccionado y, por último, se arranca la siguiente pantalla.

Clase ZonesOfflineActivity.java

La función de esta pantalla es la de mostrar la lista de zonas que componen el edificio seleccionado anteriormente, la lista sigue el formato especificado en la clase *AdapterController*. Como hemos explicado anteriormente, en cada ítem de la lista se muestra la información asociada a esa zona (nombre, código, numero de muestras obtenidos, tiempo de calibrado de la zona) y dos botones que permiten calibrar la zona y borrar las muestras almacenadas asociadas a esa zona. Durante el proceso de obtención de muestras, la lista de zonas quedará deshabilitada

para evitar que se generen eventos de pulsación no deseados y aparecerá un botón *stop* que detendrá la obtención de muestras en el momento que el usuario considere oportuno.

Los métodos que componen esta clase son:

- *void buildingChanged()*: este método es ejecutado cuando se detecta que se ha producido un cambio de edifico respecto a la calibración anterior y se encarga de mostrar un cuadro de diálogo donde se informa al usuario de que si se cambia de edificio se eliminarán las muestras de la calibración anterior y dos botones que determinan si se inicia una nueva calibración o si finalmente se realiza el cambio de edificio.
- *void borrarCalibracion()*: este método es ejecutado cuando se detecta que hay una calibración en curso y se encarga de mostrar un cuadro de diálogo donde se informa al usuario de que puede continuar con la calibración anterior o iniciar un nueva, lo que implica el borrado de las muestras obtenidas anteriormente, además este cuadro de diálogo tiene dos botones utilizados para elegir una de las opciones anteriormente presentadas.
- *void checkBuildingAndZones()*: la función de este método es comprobar si el edificio que queremos calibrar está guardado en la base de datos o si por el contrario, aún no está almacenado. Si el edificio aún no ha sido guardado, se registra en la base de datos y lee el fichero de zonas para almacenarlas. Por otro lado, si el edificio ya fue introducido anteriormente en la base de datos, directamente se obtienen las zonas asociadas a ese edificio mediante una consulta a la base de datos. Por último, se rellena la lista mostrada en la pantalla con el listado de zonas correspondiente.
- *BuildingModel createBuilding()*: este método es llamado cuando se detecta que el edificio elegido aún no ha sido almacenado en la base de datos y se encarga de crear el objeto *BuildingModel* asociado a ese edificio y de almacenarlo.
- *boolean isBuildingCreated()*: este método comprueba si el edificio que queremos calibrar ha sido guardado en la base de datos.
- *BuildingModel getBuildingInUse()*: la función de este método es la de devolver el objeto asociado al edificio que se está calibrando.
- *void setListView(Cursor cursor)*: este método recibe como argumento un *Cursor*, recibido después de la consulta a la base de datos, con la lista de zonas del edificio que queremos calibrar. Para rellenar la lista, creamos un objeto del tipo *AdapterController* al que le pasamos el *Cursor* y establecemos este objeto como *adapter* de la lista.
- *void createZonesFromBuilding(BuildingModel buildingModel)*: este método es ejecutado cuando se detecta que el edificio que queremos calibrar no está almacenado en la base de datos. Una vez que el edificio es introducido en la base de datos mediante el método *createBuilding()*, se utiliza este método para leer el fichero de zonas y almacenarlas en la base de datos. En primer lugar, creamos el objeto *File* asociado al fichero de zonas y se lo pasamos a un objeto *BufferedReader* que nos permitirá leer el texto de este fichero línea a línea. Una vez tenemos lista esta configuración, se recorre el fichero línea a línea y se separan los datos de la zona y se insertan en la base de datos. En posteriores capítulos se detallará el formato que debe cumplir este fichero de zonas que es de vital importancia para el correcto funcionamiento de la aplicación.
- *onCreate()*: este es el método que se ejecuta al arrancar esta pantalla de la aplicación, la secuencia de las acciones es la siguiente. En primer lugar, obtenemos los datos almacenados en las preferencias de la aplicación y creamos el objeto *DataBaseController* que nos dará acceso a la base de datos. En segundo lugar,

comprobamos si el edificio que queremos calibrar está almacenado en la base de datos y rellenamos la lista según las indicaciones que el usuario haya elegido en los cuadros de diálogo de los métodos *buildingChanged*, *borrarCalibracion* y *checkBuildingAndZones.* Por último, creamos los objetos *WifiAdapter* y *BLEAdapter* que van a configurar las interfaces wifi y bluetooth que usaremos para tomar las muestras.

Clase SelectRouteActivity.java

Esta clase se ejecutará después de que el usuario pulse el botón de *rutas* que aparece en la pantalla asociada a la clase *OfflineActivity.java*. La función de esta clase es mostrar una lista con las rutas que hay configuradas en la base de datos y que están asociadas al edificio que hemos elegido, cuando se pulse sobre una ruta, se lanzará la pantalla en la que podremos empezar la obtención de muestras de esa ruta. En la interfaz gráfica de esta pantalla aparecen dos elementos: una lista de las rutas disponibles y un botón que nos llevará a la pantalla para crear nuevas rutas.

Los métodos que componen esta clase son:

- *void setButton()*: este método se encarga de gestionar el evento de pulsación del botón de crear nuevas rutas. Cuando este botón es pulsado, se crea un objeto del tipo *Intent* que se utilizará para lanzar la pantalla que crea una nueva ruta.
- *void setListView(Cursor c)*: En el objeto *Cursor* se recibe la lista de rutas configuradas para ese edificio, esta lista será la que le pasemos al *adapter* del elemento lista de la interfaz gráfica. Además, dentro de este método están implementados los eventos que recogen tanto la pulsación simple como la larga sobre un elemento de la lista:
    - *setOnItemClickListener()*: este método se ejecutará cuando se produzca una pulsación sobre cualquier elemento de la lista. Cuando este evento ocurre, el primer paso es comprobar si ya hay muestras de esta ruta, si no las hubiera la aplicación pasa a la siguiente pantalla donde aparecerán las zonas que componen esta ruta. En el caso de que si tengamos muestras de esa ruta se mostrará un cuadro diálogo que le preguntará al usuario si quiere seguir con la calibración anterior o si desea empezar una nueva, ajustará los parámetros según la decisión tomada por el usuario y lanzará la pantalla siguiente de la aplicación.
    - *setOnItemLongClickListener()*: este método se ejecutará cuando se produzca una pulsación larga sobre cualquier elemento de la lista y se encarga de borrar las muestras asociadas a la ruta que están almacenadas en la base de datos. Cuando este evento ocurra se mostrará un cuadro de diálogo que pregunta al usuario si está seguro de querer borrar las muestras y, si el usuario lo confirma, se eliminarán de la base de datos.
- *onCreate()*: este es el método que se ejecuta al arrancar esta pantalla de la aplicación, la secuencia de las acciones es la siguiente. En primer lugar, se inicializan los objetos que gestionan los elementos de la interfaz gráfica, se crea un objeto del tipo *DataBaseController* y se abre la conexión a la base de datos. Por último, realizamos la consulta a la base de datos que devuelve el *Cursor* con la lista de rutas del edificio con la que rellenaremos la lista mostrada en pantalla y llamamos a los métodos *setButton()* y *setListView(Cursor c)*.

Clase RouteActivity.java

El propósito de esta clase es crear nuevas rutas para generar datos que simulen el recorrido de un usuario dentro del edificio. La interfaz gráfica de esta pantalla tiene 4 elementos: una lista

con todas las zonas del edificio, un cuadro de texto que irá mostrando los identificadores de las zonas que vamos añadiendo a esa ruta, un botón para borrar la última zona añadida a la ruta y un botón para crear la ruta.

Los métodos que forman esta clase son:

- *void checkBuildingAndZones()*: la función de este método es la de obtener la lista de zonas del edificio y rellenar la lista que se muestra en pantalla. En primer lugar, se comprueba si el edificio ya ha sido introducido en la base de datos, y en el caso de que no sea así, se almacena el edificio y sus zonas correspondientes, si el edificio ya había sido almacenado anteriormente, directamente se recogen las zonas y se rellena la lista.
- *boolean isBuildingCreated()*: el método comprueba si el edificio ha sido ya almacenado en la base de datos.
- *BuildingModel createBuilding()*: este método es llamado cuando se identifica que el edificio es de nueva creación en la base de datos y se encarga de crear el objeto *BuildingModel* y almacenarlo en la base de datos.
- *void createZonesFromBuilding(BuildingModel buildingModel)*: este método será llamado después de haber creado el edificio, la función de este método es la de leer el fichero de zonas del edificio e introducirlas en la base de datos. El funcionamiento de este método es el mismo que el del método que tiene el mismo nombre y que se encuentra en la clase *ZonesOfflineActivity*.
- *BuildingModel getBuildingInUse()*: devuelve el objeto *BuildingModel* asociado al edificio en el que se está trabajando.
- *void setListView(Cursor cursor)*: igual que en los demás métodos *setListView* encontrados en otras clases, se recibe un *Cursor* con la lista de zonas del edificio que va a ser utilizado en el *adapter* que gestiona la lista que muestra las zonas del edificio. Además, implementa el método que se ejecutará cuando se produzca el evento de pulsar una zona de la lista, este método añade la zona sobre la que se ha pulsado a la lista de zonas seleccionadas para formar la ruta y actualiza el cuadro de texto que informa al usuario de la secuencia de zonas que se han seleccionado hasta el momento.
- *void updateTextView()*: este método es el que se encarga de actualizar la información mostrada en el cuadro de texto, cada vez que se añada una zona a la ruta, este método se encarga de mostrar el nuevo identificador.
- *void getSharedPreferences()*: la función de este método es obtener la información almacenada en las preferencias de la aplicación y que es necesaria para generar la nueva ruta.
- *onCreate()*: este método se ejecutará al lanzar la *Activity* asociada a esta clase y es el que ejecuta las funciones de la clase apoyándose en los métodos detallados anteriormente. En primer lugar, inicializa los objetos asociados a los elementos de la interfaz gráfica, llama al método *getSharedPreferences* para leer los datos proporcionados por el usuario en anteriores pantallas y abre una conexión a la base de datos. En segundo lugar, llama al método *checkBuildingAndZones* para comprobar si el edificio ha sido creado anteriormente y rellenar la lista de la interfaz gráfica con todas las zonas que componen el edificio. Por último, implementa los métodos que se ejecutarán cuando se pulse sobre los botones *borrar última zona última zona* y *nueva ruta*:
    - *botón nueva ruta*: la primera función de este método es comprobar si hay alguna zona seleccionada para generar la ruta, en el caso de que no sea así, se

informa al usuario de que no es posible crear la ruta. Una vez finalizada la comprobación de que la ruta no está vacía, se crea el objeto *RouteModel* asociado a la ruta que queremos crear, se almacena la ruta en la base de datos y, mediante un objeto del tipo *Intent*, la aplicación vuelve a la pantalla anterior donde en la lista de rutas del edificio podremos observar la ruta que acabamos de configurar.

  - *botón borrar última zona*: este método borra la última zona seleccionada para formar la ruta y actualiza el cuadro de texto que informa al usuario de la secuencia de zonas seleccionadas anteriormente.

Clase ZonesRouteActivity.java

Esta es la última clase de la aplicación, su función es la de mostrar la lista de zonas que forman la ruta seleccionada previamente con el objetivo de poder generar las muestras que simulan el recorrido de un usuario dentro del edificio, esta lista de zonas sigue el orden secuencial definido en la creación de la ruta, si en este orden una zona aparece varias veces en la ruta, se mostrará tantas veces como aparezca siguiendo el orden secuencial establecido para facilitar al usuario la simulación de la ruta.

Los ítems de la lista siguen el formato definido en la clase *AdapterRouteController* mostrando la información necesaria para que el usuario se consciente del progreso realizado en la obtención de muestras. Por lo tanto, en cada ítem el usuario va a poder identificar el nombre de la zona, su código identificativo, el número de muestras obtenidas hasta el momento y el tiempo que ha transcurrido durante la obtención de esas muestras, además de el botón *start* que es utilizado para comenzar el proceso de obtención de muestras en la zona correspondiente. Durante la ejecución de este proceso de escaneo realizado por las interfaces Wifi y/o Bluetooth, la lista de zonas será deshabilitada para evitar eventos de pulsación no deseados y aparecerá un botón *stop* que permite interrumpir este proceso de obtención de muestras cuando el usuario estime oportuno.

La implementación de esta clase es bastante sencilla debido a que es el controlador *WifiRouteAdapter* el que se encarga de realizar y gestionar todo el proceso de obtención y almacenamiento de datos de forma ordenada, la función de esta clase es solo la de mostrar la secuencia de zonas correspondientes y permitir que el usuario seleccione en que zona se encuentra. Los métodos que se implementan en esta clase son:

- *setListView()*: este método es el encargado de leer los identificadores de la secuencia de zonas de la ruta, hacer las consultas necesarias a la base de datos para obtener las zonas a partir de estos identificadores, crear el objeto del tipo *AdapterRouteController* que se encargará de realizar la toma de muestras y configurar este último como *adapter* de la lista.
- *onCreate()*: como hemos explicado anteriormente en otras clases, este es el método que se ejecuta cuando arranca la *Activity*, las funciones realizadas en este caso son: crear los objetos asociados a los elementos de la interfaz gráfica, abrir una conexión a la base de datos, crear y configurar el objeto controlador del tipo *WifiRouteAdapter*, y, por último, llamar al método *setListView* para rellenar la lista con la secuencia de zonas que componen la ruta seleccionada por el usuario.

3.4 Implementación del servidor

El servidor encargado de recibir los datos enviados mediante el protocolo *MQTT* está compuesto únicamente por una sola clase (*Main.java)* que carece de interfaz gráfica que

implementa la interfaz *MqttCallback* lo que le proporciona la funcionalidad para leer los mensajes *MQTT* recibidos. Cuando el servidor es ejecutado, inicia la conexión al *broker* del protocolo *MQTT* que estará ejecutándose en la misma máquina. Cuando la conexión se haya realizado correctamente, se mostrará un mensaje por pantalla informando al usuario de este hecho y ya se puede comenzar el envío de datos. Este servidor se quedará escuchando los mensajes *MQTT* que reciba y almacenará los datos en el fichero correspondiente mediante un objeto del tipo *BufferedReader*. El mensaje *MQTT* tiene el siguiente formato: "userX_" + información de la muestra obtenida. De esta forma se distinguen dos partes separadas por el símbolo "_", la primera parte índica que usuario envía el mensaje, y es utilizada para identificar en que fichero guardar la información, y la segunda parte corresponde a la información de una muestra obtenida durante el proceso de calibración. Todo este procedimiento de recepción del mensaje y guardado de datos va a ser explicado en detalle en la especificación de la implementación de los métodos que componen la clase:

- *static void writeFile(String message)*: este método recibe una cadena de caracteres con la información recibida en el mensaje y la escribe en el fichero, cada línea del fichero corresponde a una muestra obtenida por la aplicación de calibración.
- *static void openFile(String user)*: este método recibe como argumento un *String* que se corresponde con la primera parte de un mensaje *MQTT* recibido y que será utilizado para identificar el fichero correspondiente a ese usuario mediante el método *findFile(String user)*. Si el fichero ya existe, configura el *BufferedReader* para que escriba en ese fichero e informa al usuario con un mensaje por pantalla. En el caso contrario, crea el fichero, configura el *BufferedReader* e informa al usuario de que el fichero ha sido creado.
- *static void closeFile()*: cierra el objeto del tipo *BufferedReader*.
- *static boolean findFile(String user)*: este método recibe como argumento un String que se corresponde con la primera parte de un mensaje MQTT recibido y que será utilizado para identificar el fichero correspondiente a ese usuario. Devuelve *true* si el fichero ha sido encontrado y *false* si la búsqueda del fichero no ha proporcionado ningún resultado.
- *void messageArrived(String arg0, MqttMessage arg1)*: este método se ejecutará cuando se reciba un mensaje *MQTT*, recibe como argumentos el *topic* del mensaje y el contenido del mismo. El primer paso que se realiza es separar el contenido del mensaje en las dos partes que hemos explicado anteriormente (nombre de usuario y datos de la muestra obtenida). A continuación, se llama al método *openFile(String user)* para identificar o crear el fichero asociado a ese usuario y al método *writeFile(String message)* para escribir la información recibida en una línea de este fichero. Para finalizar, se llama al método *closeFile()* para cerrar el acceso al fichero.
- *main(String[] args)*: este método se ejecuta al arrancar el servidor y en él se definen los parámetros de la conexión *MQTT* (*topic* y url). Una vez definidos estos datos se procede a configurar los objetos *MqttAsyncClient* y *MqttConnectOptions* que van a gestionar la conexión y se llama al método *MqttAsyncClient.connect()* para conectarse con el *broker*. Este método informará al usuario de que la conexión se ha realizado correctamente mediante un mensaje mostrado por pantalla.

4. Aplicación de calibración y simulación de rutas

Una vez que ha sido detallada toda la implementación de la aplicación desarrollada, en este capítulo vamos a explicar el funcionamiento de la misma desde el punto de vista del usuario, mostrando ejemplos de las pantallas y especificando como se realiza la navegación entre ellas.

4.1 Inicio

Cuando se arranca la aplicación, la primera pantalla que se observa es la mostrada en la Figura 18:

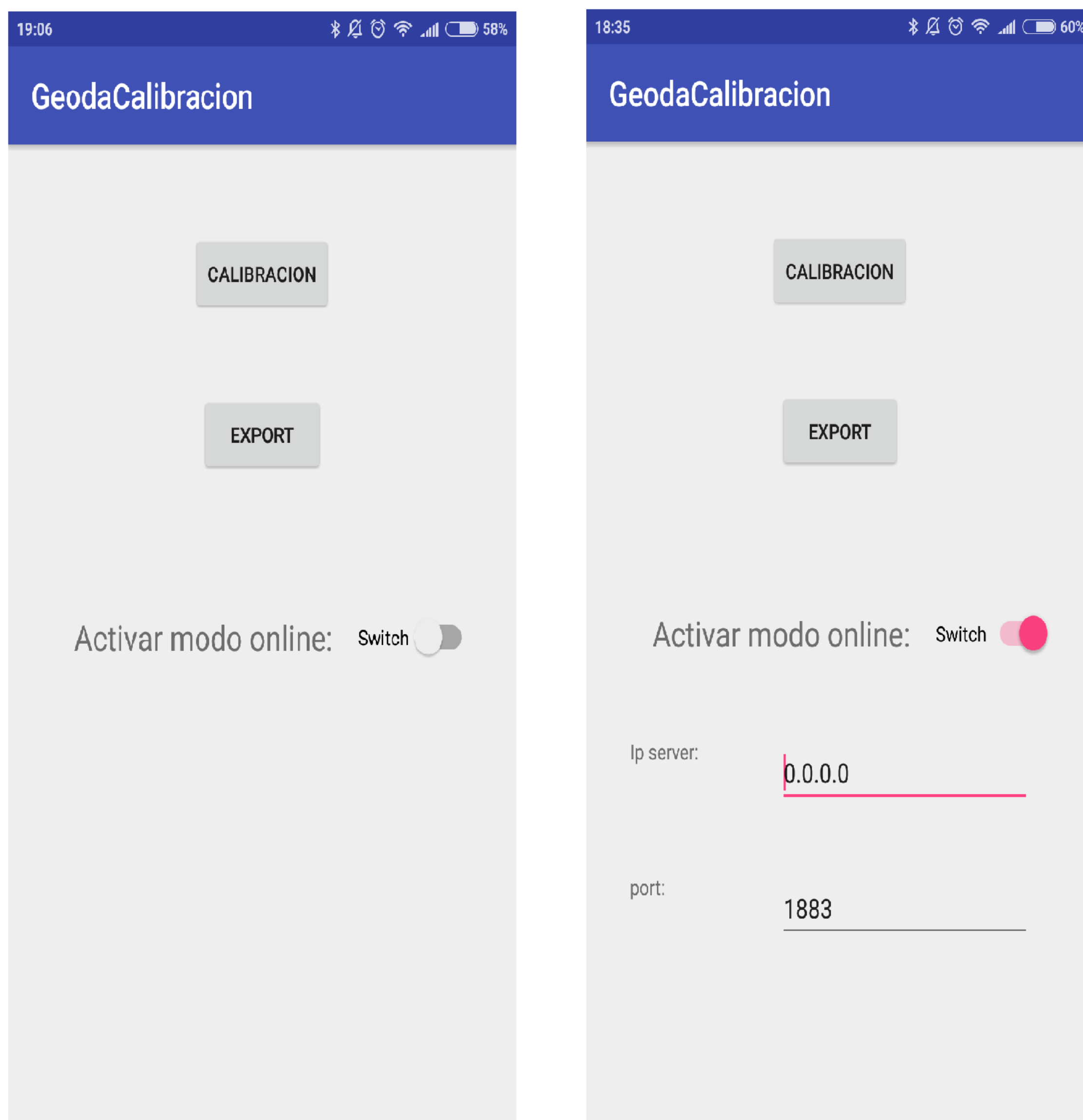


Figura 18 Primera pantalla de la aplicación

Como podemos observar en la figura anterior (Figura 18) la interfaz gráfica es bastante sencilla, el botón *'export'* se utiliza para exportar la base de datos, al pulsar sobre él se abre un menú con un listado de las aplicaciones externas disponibles para exportar el fichero .db de la base datos.

Con el botón del tipo *switch* seleccionamos si queremos activar el modo online de la aplicación, en caso afirmativo el usuario deberá introducir la dirección del ip del servidor y el puerto en el

que está escuchando. Una vez introducidos los datos necesarios, el botón 'calibración' nos lleva a la siguiente pantalla (Figura 19) donde el usuario deberá introducir los datos necesarios para iniciar la calibración.

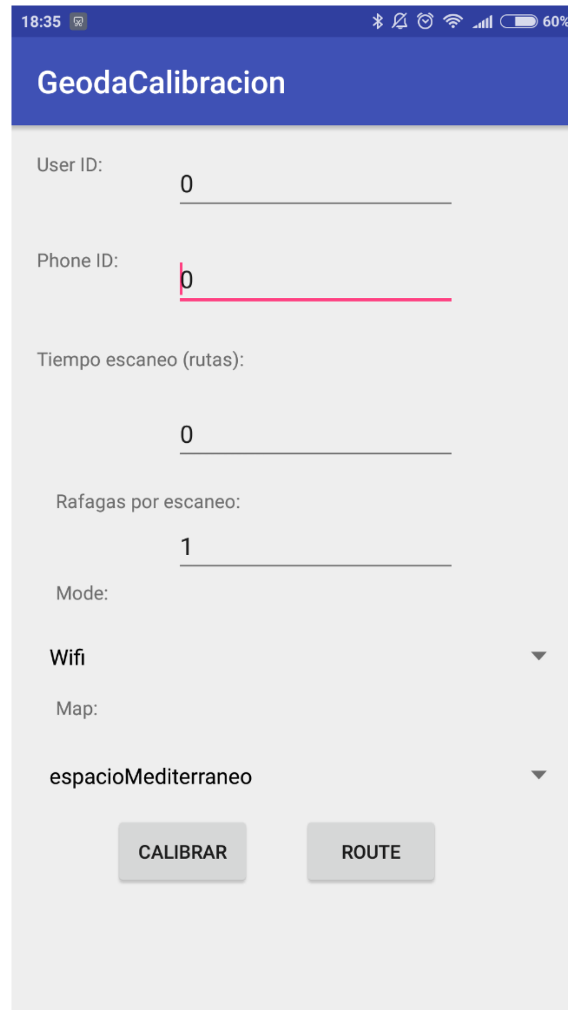


Figura 19 Pantalla de introducción de datos de usuario

En el primer campo de introducción de datos se debe introducir el número del usuario que va a realizar la calibración, seguidamente se introduce el número de terminal que se va a utilizar. Los siguientes datos por introducir se utilizan para definir el tiempo que transcurre entre escaneos y las ráfagas de resultados que queremos enviar por cada escaneo durante la simulación de rutas. Por último, se encuentran dos listas desplegables utilizadas para definir el modo de obtención de datos (wifi, *BLE* o ambos) y el edifico que queremos calibrar. Los edificios que estén disponibles para su elección dependerán de los ficheros de zonas que previamente hayamos almacenado el en directorio */geoda/building properties/* de nuestro dispositivo.

Los botones de la parte baja de la pantalla son para seleccionar que tipo de muestras queremos obtener: muestras de calibración (botón 'calibrar') o muestras de simulación de rutas (botón 'rutas').

4.2 Calibración

La pantalla de calibración, mostrada en la Figura 20, se encarga de mostrar la lista completa de zonas que modelan el edificio que hemos elegido previamente

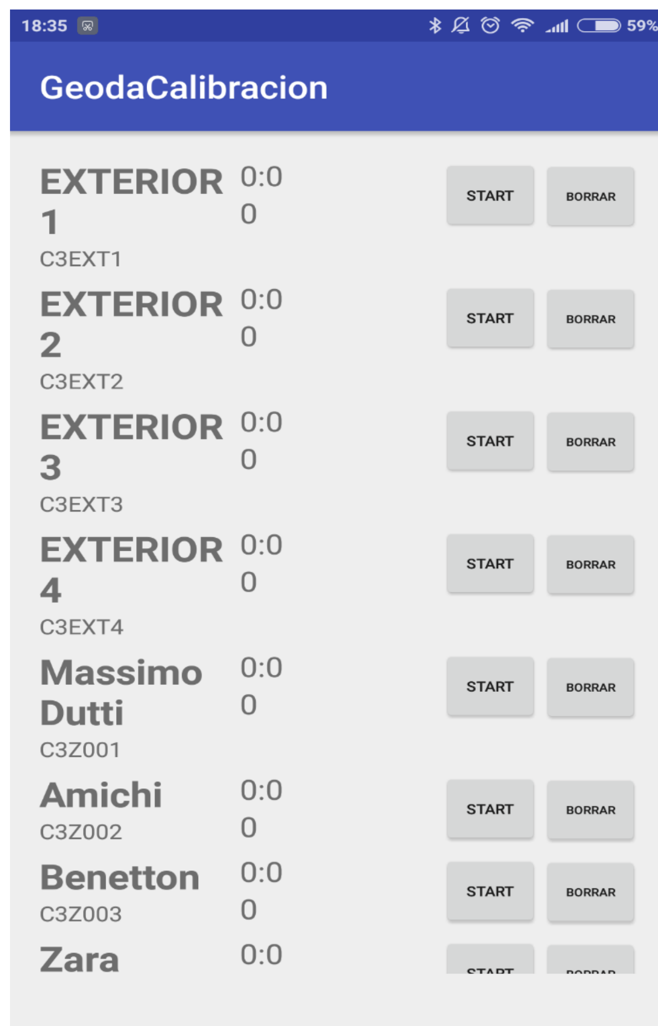


Figura 20 Pantalla de listado de zonas

Para cada zona se muestra su nombre y su código identificador, además de varios valores que aportan información sobre el progreso de calibración de cada una de ellas. En el valor superior se índice el tiempo durante el que se ha calibrado la zona con el formato “min:seg” y el valor inferior muestra el número de muestras que se han obtenido en esa zona. El botón ‘*start’* sirve para comenzar la obtención de muestras y el botón ‘borrar’ para eliminar todas las muestras que han sido obtenidas en esa zona.

4.3 Rutas

Si se elige el modo de simulación de rutas, la primera pantalla (Figura 21) muestra el listado de rutas que han sido configuradas para el edificio escogido. Si pulsamos sobre alguna de estas rutas, la aplicación nos llevará a la siguiente pantalla (Figura 23), donde se muestra la secuencia de zonas que definen la ruta seleccionada. Por último, si se realiza una pulsación prolongada sobre una ruta, aparecerá un cuadro de diálogo que permite el borrado de las muestras asociadas a la zona seleccionada.

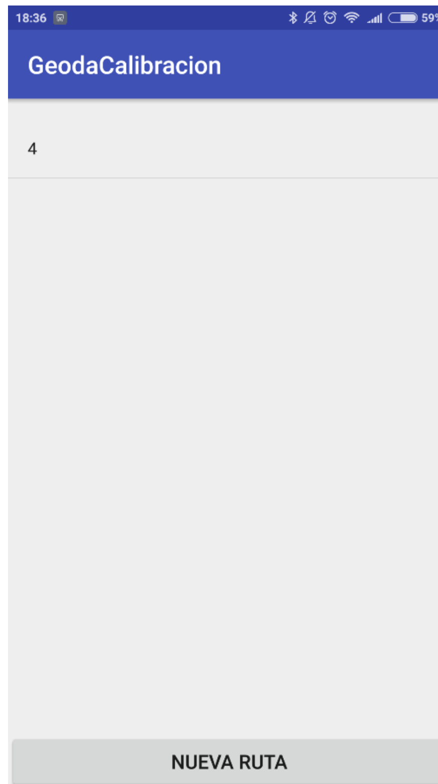


Figura 21 Pantalla con el listado de rutas

El botón ‘nueva ruta’ se utiliza para configurar nuevas rutas de simulación al edificio, cuando se pulsa sobre él nos lleva a la pantalla de creación de rutas mostrada en la Figura 22.

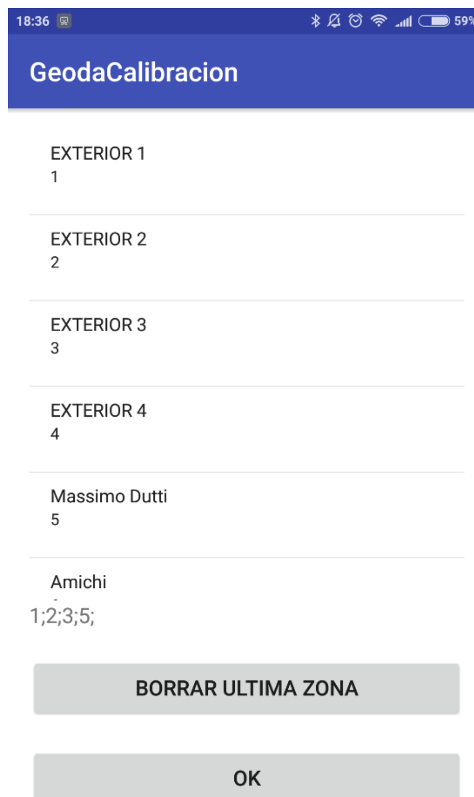


Figura 22 Pantalla de creación de rutas

En esta pantalla (Figura 22) se muestra el listado completo de las zonas pertenecientes al edificio sobre el que se va a calibrar. Para crear la secuencia de zonas que definen la nueva ruta el usuario deberá seleccionar las zonas en el orden deseado y pulsar sobre el botón 'ok' para volver a la pantalla del listado de rutas, donde podrá observar la nueva ruta creada. Durante la creación de rutas aparece un cuadro de texto informando al usuario de los identificadores de las zonas que ha añadido a la ruta y el botón 'borrar última zona' que le permite eliminar de la ruta la última zona añadida.

Por último, se va a mostrar la pantalla que contiene muestra la lista de zonas de una ruta y permite realizar la simulación de la misma.

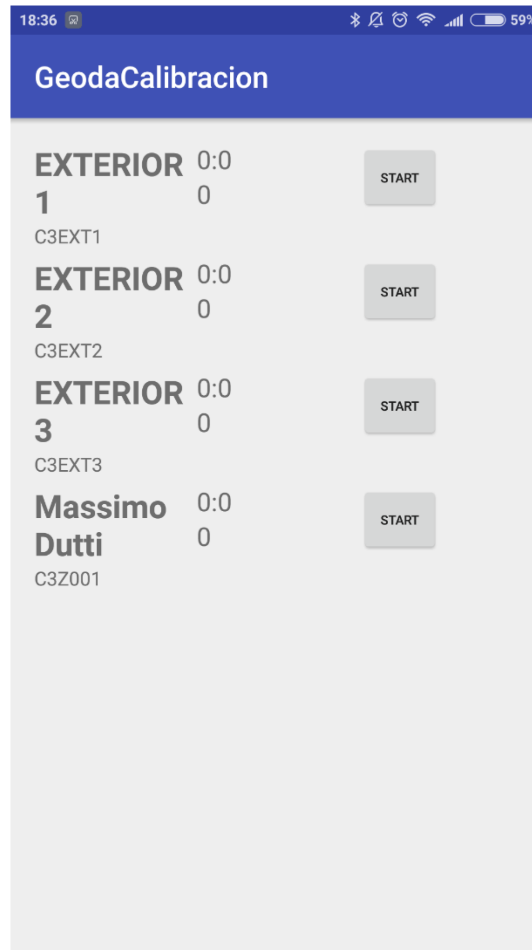


Figura 23 Pantalla que muestra el listado de zonas de una ruta

Como podemos observar en la Figura 23 Pantalla que muestra el listado de zonas de una ruta, el formato en el que se muestran las zonas es el mismo que en la pantalla de calibración, se indican el nombre y el código de la zona, además de los valores de tiempo de calibración y número de muestras obtenidas, con la diferencia de que en esta pantalla los dos valores están asociados a las muestras de esta zona asociadas exclusivamente a la ruta seleccionada. El botón 'start', de la misma manera que en la parte de calibración, se utiliza para comenzar la toma de muestras.

5. Servidor para la recepción de mensajes MQTT

Como hemos explicado en el capítulo 3, el servidor no tiene interfaz gráfica y tampoco se requiere la interacción del usuario en él, pero si muestra cierta información que es muy útil para informar tanto del estado de la conexión, como de los mensajes que han sido recibidos. Además, hay algunos pasos previos a la ejecución del servidor que debemos seguir para que el funcionamiento del mismo sea óptimo.

5.1 Puesta en marcha

En primer lugar, en la máquina donde se ejecute el servidor debe de haber instalado un *broker MQTT* que se encargará, como se explica en el capítulo 2, de recibir los mensajes del cliente que los publica y enviarlos al cliente que desea recibirlos, que en nuestro caso será este servidor. Antes de arrancar nuestro servidor, el *broker* mencionado anteriormente debe de estar en funcionamiento.

```
"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...
Connected to the target VM, address: '127.0.0.1:65299', transport: 'socket'
Connected
topic:geodacalibracion
Fichero creado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;ec:08:6b:b5:54:94;-55;2462;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;00:9a:cd:c2:4a:fc;-71;2432;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;74:b5:7e:f5:53:e2;-89;5240;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;18:d6:c7:90:22:ec;-89;2452;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;30:b4:9e:08:bf:01;-94;5200;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;74:b5:7e:f5:53:e1;-88;2462;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;60:e3:27:5a:2a:e6;-93;2442;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011443;a8:a7:95:7e:a8:74;-77;2437;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011664;ec:08:6b:b5:54:94;-54;2462;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011664;74:b5:7e:f5:53:e2;-89;5240;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011722;ec:08:6b:b5:54:94;-54;2462;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011722;74:b5:7e:f5:53:e2;-89;5240;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011771;ec:08:6b:b5:54:94;-54;2462;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011771;74:b5:7e:f5:53:e2;-89;5240;
Fichero encontrado C:\Users\pablo\Desktop\Datos FTP\MQTT\
1;1505935011818;ec:08:6b:b5:54:94;-54;2462;
```

Figura 24 Información mostrada por el servidor

Como podemos observar en la Figura 24, el servidor informa del estado de la conexión, el *topic* asociado al sistema y la información de los mensajes recibidos.

6. Pruebas

En este capítulo vamos a realizar una serie pruebas que nos permitan observar que circunstancias influyen en el proceso de obtención de datos. Con estas pruebas se pretende observar la heterogeneidad del hardware con respecto a la adquisición de datos. Es decir, cómo cambia la calibración al usar dos dispositivos diferentes Además, también queremos observar cómo afecta a las interfaces wifi y bluetooth que estén escaneando simultáneamente, ya estas tareas son muy costosas desde el punto de vista computacional.

6.1 Terminales

Los dispositivos que vamos a usar para la realización de las pruebas son:

- **Xiaomi Redmi Note 3 Pro.** Mostrado en la Figura 25. Soporta todos los estándares de wifi, incluido el estándar IEEE 802.11ac (5 GHz). Además, soporta la tecnología *BLE*.

Figura 25 Xiaomi Redmi Note 3 Pro

- **Bq Aquaris M8**. Mostrado en la Figura 26. Soporta todos los estándares de wifi, incluido el estándar IEEE 802.11ac (5 GHz). Además, soporta la tecnología BLE.

Figura 26 Bq Aquaris M8

6.2 Prueba con la interfaz wifi

Para medir las diferencias entre dos terminales se ha utilizado la aplicación para tomar muestras en el modo 'Wifi' durante un 1 minuto y en la misma ubicación.

6.2.1 Resultados

Los resultados se muestran en la Tabla 1:

	Bq Aquaris M8	*Xiaomi Redmi Note 3 Pro*
Número de escaneos	16	193
Número total de muestras	170	1737
Número medio de muestras por escaneo	11	9
Tiempo medio de escaneo	3.74 s	0.31 s

Tabla 1 Resultados de la prueba wifi

Cómo podemos observar, los resultados obtenidos son muy dispares, recogiendo el dispositivo de la marca *Xiaomi* ha recogido 10 veces más muestras que la *tablet Bq.* Al trabajar los dos en los mismos estándares, la velocidad de escaneo del *smartphone* ha sido aproximadamente 10 veces más alta que la de la *tablet*. Analizados los AP que se han adquirido en estas pruebas, se observa que el número de punto de acceso wifi es prácticamente igual en los dos terminales.

6.2.2 Conclusiones

En primer lugar, se observa que el número de puntos de acceso detectados es muy similar, esto se debe a que el hardware de los dos terminales soporta los mismos estándares de wifi, lo que provoca que no haya diferencias a la hora de escanear el espacio radioeléctrico, si alguno de los dos terminales no soportara todos los estándares wifi, podríamos observar que el número de muestras por escaneo sería diferente.

En segundo lugar, vamos a analizar la sorprendente diferencia que se observa en la velocidad de escaneo de los dos terminales. Mientras que la *tablet Bq* emplea de media 3.74 segundos para cada escaneo, el *smartphone Xiaomi* genera resultados cada 0.31 segundos. El tiempo mínimo de escaneo que soportan las interfaces wifi de los dispositivos Android se encuentra en el intervalo de 2 a 6 segundos, teniendo en cuenta este dato, la *tablet Bq* ha mostrado unos resultados razonables, mientras que es el dispositivo *Xiaomi* el que presenta anomalías en el proceso de obtención de datos. Esta inusual velocidad de escaneo se debe a que el dispositivo está devolviendo datos almacenados en la caché, por lo tanto, muchos de los resultados que nos proporciona no se corresponden con la realidad si no que son resultados obtenidos en escaneos anteriores. La diferencia de comportamiento del sistema Android de los dos terminales se debe a que el *smartphone* incorpora una *ROM* de Android propia de Xiaomi bautizada como *MIUI*.

6.3 Prueba Bluetooth

Para medir las diferencias entre dos terminales se ha utilizado la aplicación para tomar muestras en el modo 'BLE' durante un 1 minuto y en la misma ubicación.

Para la creación de muestra hemos utilizado dos *beacons BLE* configurados con un intervalo de anuncio de 100 milisegundos y los dos a la misma distancia del terminal.

6.3.1 Resultados

Los resultados obtenidos se muestran en la Tabla 2:

	Bq Aquaris M8	*Xiaomi Redmi Note 3 Pro*
Número total de muestras	467	526
Número de muestras por segundo	7.78	8.77
Muestras del beacon 1	200	280
Muestras del beacon 2	267	246

Tabla 2 Resultados de la prueba bluetooth

Se observa que los resultados obtenidos no presentan grandes diferencias entre los dos dispositivos, solamente se observa que el *smartphone* es ligeramente más rápido a la hora obtener las muestras *BLE*.

6.3.2 Conclusiones

En el caso de bluetooth, la diferencia de versión de Android no influye en la detección y obtención de datos ya que como podemos observar en la Tabla 2, los resultados obtenidos son muy similares, con una pequeña diferencia de velocidad de la interfaz debida a que los dos dispositivos utilizados no tienen el mismo hardware.

6.4 Prueba utilizando wifi y bluetooth

Para la realización de esta prueba se han combinado las condiciones de las dos anteriores: los dos terminales en la misma ubicación tomando muestras durante un minuto y dos *beacon BLE* configurados con un intervalo de anuncio de 100 milisegundos.

6.4.1 Resultados

Los resultados obtenidos durante la realización de la prueba se muestran en la Tabla 3:

	Bq Aquaris M8	*Xiaomi Redmi Note 3 Pro*
Número total de muestra	651	1533
Número de muestras wifi	140	1190
Número de muestras BLE	511	343
Número de escaneos wifi realizados	3.75s	2.2s
Muestras wifi por escaneo	9	9
Muestras BLE por segundo	8.51	5.7

Tabla 3 Resultados de la prueba conjunta wifi BLE3

Tras un análisis de los datos se puede afirmar que el comportamiento de las interfaces wifi y bluetooth cuando escanean simultáneamente es similar a cuando lo hacen de forma individual, observando además que las muestras obtenidas siguen la misma tendencia que en las pruebas anteriores (Tabla 1 y Tabla 2). En el siguiente apartado vamos a observar una comparativa con los datos de las tres pruebas realizadas.

6.4.2 Comparación de resultados de las tres pruebas

En esta sección vamos a presentar las gráficas comparativas de los resultados obtenidos.

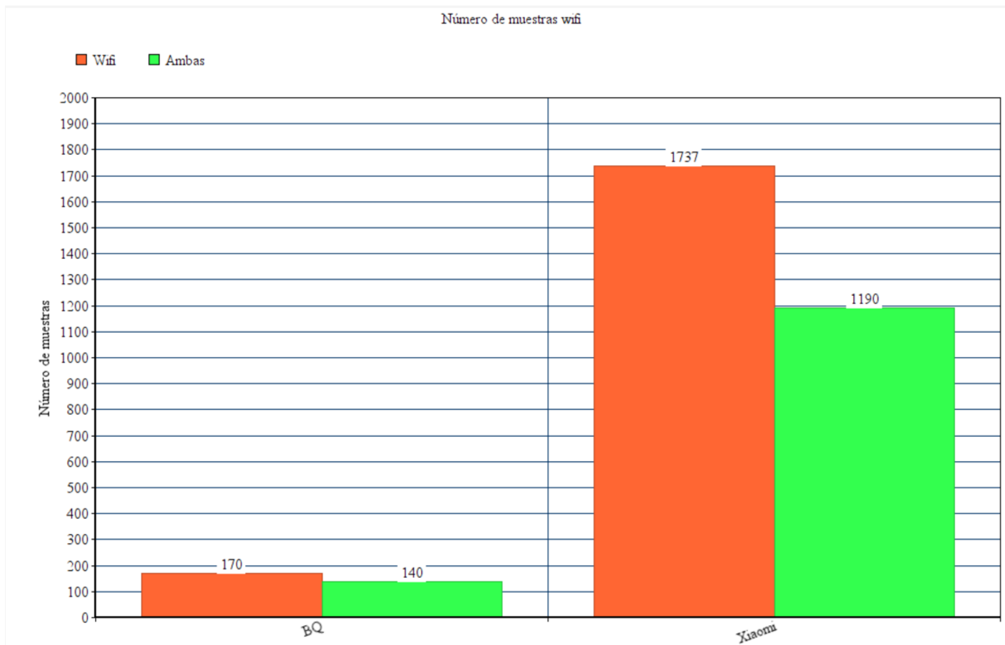


Gráfico 1 Comparativa del número de muestras wifi

En la imagen anterior (Gráfico 1) podemos observar que la utilización de la interfaz bluetooth afecta ligeramente en el número de muestras wifi obtenidas durante la calibración con la *tablet BQ*, mientras que en el caso del dispositivo de la marca *Xiaomi* el descenso en el número de muestras es muy notable, pero como hemos comentado anteriormente, estas muestras no son totalmente fiables debido a que todas las muestras no se corresponden con escaneos reales así que para medir la influencia del bluetooth sobre la obtención de muestras wifi vamos a centrarnos en la comparativa del dispositivo *BQ*.

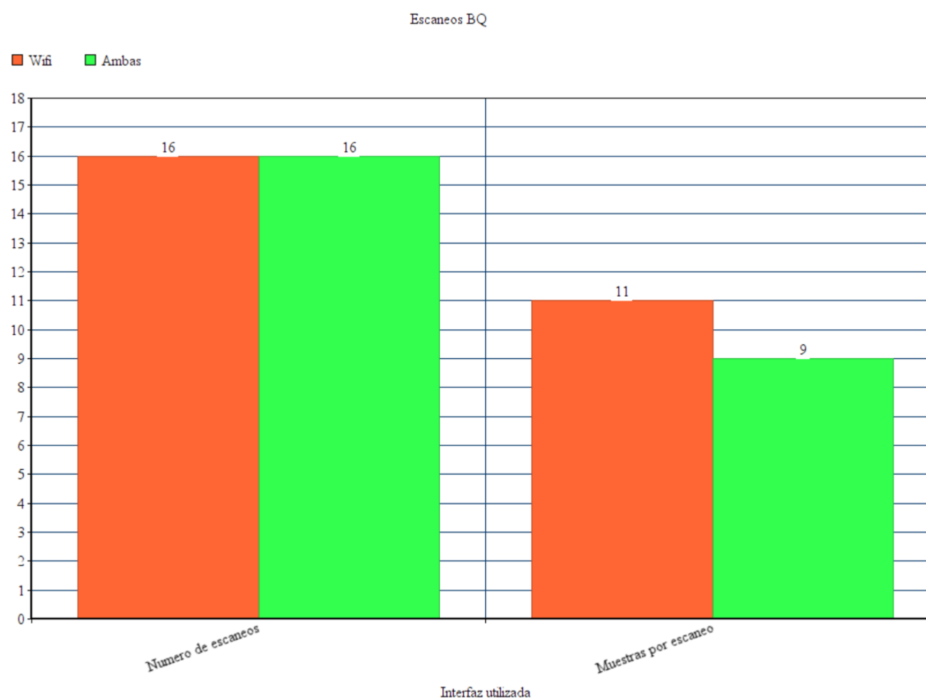


Gráfico 2 Comparación del comportamiento de la interfaz wifi en la tablet BQ

En la comparativa anterior (**¡Error! No se encuentra el origen de la referencia.**) podemos comprobar que, aunque el número demuestras descienda ligeramente al utilizar las dos interfaces simultáneamente, el comportamiento de la interfaz wifi no varía ya que sigue realizando es mismo número de escaneos, con la única diferencia de que detecta menos puntos de acceso en cada escaneo.

En los siguientes gráficos se mostrarán los datos referidos a la interfaz bluetooth.

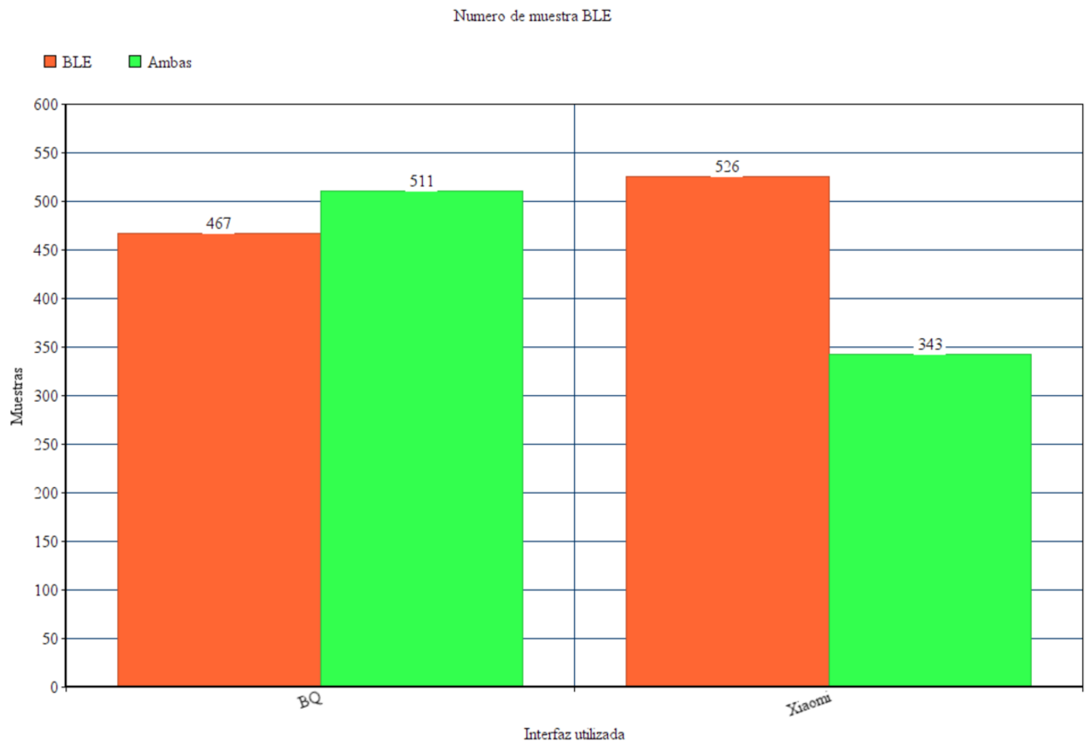


Gráfico 3 Comparativa del número de muestras BLE obtenidas

Esta imagen (Gráfico 3) muestra la diferencia en el número de muestras BLE obtenidas cuando se utilizada la interfaz BLE y cuando las dos interfaces (wifi y BLE) actúan simultáneamente. Para el dispositivo *BQ* no hay diferencia significativa entre las dos situaciones, incluso captura más muestras en el segundo escenario. En el caso del dispositivo *Xiaomi*, si se observa un notable descenso en el número de muestras recogidos.

6.4.3 Conclusiones

Después de los resultados adquiridos durante las distintas pruebas podemos confirmar que el uso simultáneo de las dos tecnologías utilizadas para la obtención de muestras no influye en el correcto funcionamiento, aunque si produce un ligero empeoramiento del número de muestras obtenido. Además, teniendo en cuenta el comportamiento del dispositivo *Xiaomi*, hemos comprobado que es importante la elección del terminal utilizado para realizar la calibración ya que, según el fabricante, el hardware o la versión de Android, los datos pueden ser muy diferentes entre dos terminales con la misma tecnología.

7. Conclusiones y líneas futuras

7.1 Conclusiones

Ha continuación se enumeran los principales conocimientos adquiridos durante el desarrollo de este proyecto, así como las conclusiones que podemos extraer del mismo:

- Se ha alcanzado un amplio conocimiento del sistema operativo Android a nivel global y especialmente en la utilización y gestión de las interfaces wifi y bluetooth.
- Se han conocido y aplicado alguna de las técnicas de buena programación, así como patrones de diseño.
- Se han adquirido conocimientos sobre las necesidades, el funcionamiento y la implementación de un sistema de localización en interiores.
- Se ha realizado un estudio exhaustivo sobre los métodos de escaneo wifi y BLE en Android, los cambios que se experimentan al utilizar distintos tipos de dispositivos y las interferencias que se pueden producir al ser utilizadas simultáneamente.
- Se ha recibido una formación adicional en la gestión y el trabajo con bases de datos.
- Se han adquirido conocimientos sobre los protocolos de envío de datos en tiempo real.

El desarrollo e implementación de esta aplicación ha proporcionado una herramienta que permite ser utilizada en un sistema real de localización en interiores. Además, la experiencia de trabajar en este campo me ha enseñado que antes de calibrar un edificio hay que tener en cuenta ciertas consideraciones que complementan las funciones realizadas por esta aplicación:

- Modelar correctamente el edificio sobre el que se va a trabajar, es importante una buena distribución de zonas que facilite el cálculo de la localización.
- Una buena planificación de las sesiones de calibración teniendo en cuenta el tamaño de las zonas y el número de muestras necesario para caracterizarlas.
- Un estudio de los dispositivos que se van a utilizar para tomar los datos ya que pueden proporcionar datos muy distintos además de que, como hemos visto en el capítulo de pruebas, algunos dispositivos necesiten tareas de procesamiento de las muestras antes de entrenar al sistema.
- La elección del terminal Android requiere de un estudio previo del comportamiento debido a que pueden almacenarse datos en caché que arrojen datos que no son reales

7.2 Líneas futuras

En cuanto a posibles líneas futuras de investigación y mejoras en la aplicación destacan las siguientes:

- Mejoras en la interfaz gráfica y en el aspecto visual de la aplicación.
- Implementar una funcionalidad que permita obtener el fichero de zonas de un edifico de manera automática a partir de un mecanismo de entrada.
- Implementar una funcionalidad que a partir del fichero de zonas la aplicación pueda crear unas rutas de simulación predeterminadas.
- Cambiar la pantalla que muestra la lista de zonas para que muestre un plano del edificio y según el lugar donde pulse el usuario, reconozca la zona y obtenga las muestras

En cuanto a posibles mejoras en el servidor y futuras líneas de investigación:

- Implementación de una interfaz gráfica que facilite la visualización de la información almacenada y aquella que se va recibiendo en tiempo real..

- Mejora del almacenamiento de datos con el uso de una base de datos en el servidor.

Bibliografía

[1] https://developer.android.com/reference/packages.html

[2] https://mosquitto.org

[3] https://www.bluetooth.com

[4] https://www.eclipse.org/paho/files/javadoc/index.html

[5] https://accent-systems.com/support/knowledge/ibks-config-tool-user-manual/

[6] MEIER, Reto. Professional Android 4 Application Development. ISBN: 978-1-118-10227-5