ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Trabajo Fin de Grado

GRADO EN INGENIERÍA TELEMÁTICA

Diseño y desarrollo de un App en React Native y Back-end para la gestión de presencia de trabajadores

AUTOR: Marta Terrones Albarracín

DIRECTOR: Alejandro S. Martínez Sala

Codirector: Juan José Franco Peñaranda

Abril / 2022

Autor	Marta Terrones Albarracín
E-mail del autor	marta22terrones@gmail.com
Director	Alejando S. Martínez Sala
E-mail del director	alejandros.martinez@upct.es
Co-director/mail	Juan José Franco Peñaranda, juanjo.franco@digio.es Empresa DIGIO S.L.
Título del TFE	Diseño y desarrollo de una App en React Native y Back-end para la gestión de presencia de trabajadores.
Resumen	La automatización de la gestión de la presencia de trabajadores en sus puestos de trabajo puede ayudar a reducir la carga de trabajo y burocracia para el cumplimiento de la reciente normativa de registro de la jornada laboral de obligado cumplimiento en las empresas. Se propone diseñar la arquitectura de un sistema sencillo y eficiente de gestión de la presencia mediante un app que monitoriza datos GPS y Bluetooth para detectar la ubicación y enviar estos datos de telemetría a un servidor backend. Se va a desarrollar un caso de uso contando con una empresa y, dadas sus necesidades. identificar y diseñar un servicio útil, sencillo y que tenga potencial de convertirse en un producto comercial. Para el desarrollo del app se va a usar tecnología React Native y para el backend NodeJS. La comunicación entre el app y el backend se realizará usando el estándar MQTT.
Titulación	Grado en Ingeniería Telemática.
Departamento	Tecnología de la Información y las Comunicaciones.
Fecha de presentación	Abril - 2022

Agradecimientos

Quiero agradecer a mis docentes, en especial a mi tutor Alejandro S. Martínez Sala y a mi co-director de TFG Juan José Franco Peñaranda. Gracias por guiarme en este proyecto, por la ayuda en la planificación, información y organización.

También, expresar mi agradecimiento a mis padres Antonio e Isabel Mª, y hermanos Ana y Antonio. Gracias de corazón por el cariño y el apoyo incondicional en todos estos años, por haber creído en mí y por no soltarme nunca de la mano.

Gracias a mis amigos, Marisa, Mª del Mar, Inma, Álvaro y Eugenio. Por estar siempre.

A mi compañero de universidad Diego, por los cafés de antes de clase y las cervezas de después.

Para finalizar, agradecer a mi tío Miguel. Ahí donde estés, gracias por ser la base de la familia Terrones, por haber hecho el camino más fácil.

A todos ellos, mil gracias.

Índice

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

1.1 Introducción

El Real Decreto 8/2019[1], de 8 de marzo de 2019, redactó la obligación legal de todas las empresas de registrar la jornada de trabajo de sus trabajadores de manera individual. Se busca evitar las horas extraordinarias por parte del trabajador, y en el caso de que se lleven a cabo, que sean abonadas por la empresa y bien acreditadas. Las jornadas se anotarán día a día y a final de mes se sumará el total. Cada empleado recibirá junto con el salario el cómputo de las horas trabajadas. Además, los resúmenes de las horas deberán conservarse durante un mínimo de 4 años por parte de la empresa.

Se ha preguntado entre varias empresas conocidas por su normativa en cuanto a la gestión del control del horario, y se ha destacado el caso de *Imprenta Grafidemar*. Se trata de un negocio familiar ubicado en Águilas (Murcia) que hace uso de un método de registro conservador: fichaje en papel individual de cada trabajador. Es una de las soluciones más tradicionales y sencillas pero puede presentar ciertas dificultades tanto en el día a día como a largo plazo:

- Control exhaustivo de que se rellenan adecuadamente.
- Asegurarnos de que se almacenan correctamente durante el periodo que obliga la ley (4 años mínimo)
- Posibles pérdidas y descuadres

Teniendo en cuenta esto, se propone el diseño de una arquitectura eficiente para la gestión de presencialidad de empleados mediante una App android que monitoriza datos Bluetooth y GPS. El objetivo será el envío de información mediante MQTT a un servidor Back-end y, haciendo uso de un Front-end poder filtrar entre los datos almacenados que se consideren útiles.

Por un lado, Bluetooth es un protocolo inalámbrico muy popular, disponible en teléfonos inteligentes, ordenadores y muchos otros dispositivos. El crecimiento de esta tecnología hizo que Bluetooth SIG, entre otras empresas, se dieran cuenta del exceso de energía que consumía lo que derivó a la implementación de BLE[2] (Bluetooth Low Energy)

Por otro lado, la tecnología GPS[3] se trata de un sistema de posicionamiento global por satélite que ofrece información de la ubicación. No funciona correctamente en interiores, aunque sí que mantiene la información de ubicación.

Usaremos la información BLE para ubicar a los trabajadores dentro de la empresa gracias a los dispositivos beacons instalados en esta mientras que GPS será necesario para confirmar que no están dentro de esta.

Siendo conscientes de ambos puntos, usaremos ambas tecnologías para saber si un trabajador está dentro de la empresa o no. Introducimos una nueva técnica conocida como *Geofencing*[4]. Permite establecer perímetros para una empresa haciendo uso de ubicación GPS y datos de un dispositivo, en nuestro caso un beacon. Se fija un perímetro invisible que delimita una área geográfica, como una especie de valla virtual. Esta limitación se conoce como *geofence* y será establecida por un listado de MACs de los dispositivos BLE asociados en el interior de la empresa.

Se pretende el desarrollo de los medios necesarios para poder detectar dispositivos BLE cercanos junto con la localización del dispositivo que lo lleve a cabo. De esta manera, se llevará a cabo la automatización de una gestión de presencia de trabajadores en sus puestos de trabajo.

Los medios que se van a necesitar para el desarrollo será una App android que simulará un escanear BLE y un rastreo de localización GPS junto con un Back-end y una Interfaz de Usuario. Esta App se encargará de enviar los datos sin filtrar a un servidor vía MQTT, más adelante se escogerá entre toda la información la necesaria para la monitorización.

A continuación vemos un esquema simplificado para entender el contexto del sistema y lo explicado anteriormente.

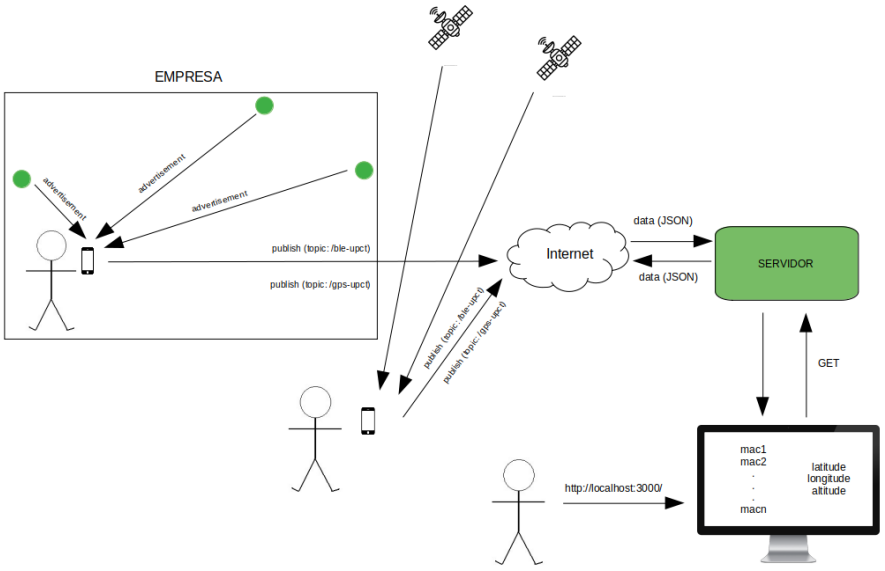

Figura 1. Esquema simplificado proyecto

1.2 Objetivos

Involucrar todos estos conceptos tiene sus objetivos, como son:

- Análisis del problema y diseño de la arquitectura.
- Familiarización en entorno NodeJS y React Native e implementación App android
- Conocimiento de tecnologías Bluetooth Low Energy, GPS y MQTT. Desarrollo de estas en conjunto.
- Monitorización de datos a través de inteligencia de geolocalización y dispositivos BLE beacons.
- Implementación Back-end para desarrollo y almacenamiento de datos.
- Implementación Front-end para filtrado de datos y consulta de información por parte de responsable de empresa.
- Pruebas de rendimiento y de comprobación del proyecto.

1.3 Herramientas software

Para el desarrollo de este trabajo se han utilizado los siguientes programas y librerías:

- Galaxy A50.

Dispositivo móvil donde se harán las pruebas del entorno de desarrollo de la App android.

- App iBKS Config Tool.

Aplicación que nos permite configurar las balizas iBKS.

- App nRF Connect Nordic.

Aplicación que escanea las tramas BLE de tu entorno por medio de un Smartphone.

- MQTTX v1.5.2.

Aplicación de escritorio que se conecta a un servidor MQTT y permite el envío de tramas a los topics que se desee, además de corroborar el funcionamiento del sistema conectándose a los topic que queramos.

- NodeJS 14.17.3 LTS.

Framework del lenguaje de programación Javascript que se especializa en la creación de backends, se ha utilizado para dar forma al servidor que escucha las tramas MQTT y las guarda en BBDD para su uso futuro, a continuación, se listan las librerías usadas

  - express v4.17.3
  - typescript v4.1.2

- React Native.

Framework de programación de aplicaciones nativas multiplataforma basado en JavaScript y ReactJS. Las librerías que se han utilizado son:

  - react-native-ble-plx
  - react-native-location
  - sp-react-native-mqtt
  - react-native-device-info

- Typescript v4.1.2

Lenguaje de programación libre y de código abierto (superconjunto de Javascript) Se utiliza en el desarrollo de todos los script del proyecto.

- Postman 8.6.2.

Software para realizar peticiones de prueba al Back-end.

- MongoDB Atlas.

Base de datos NoSQL de código abierto y orientada a documentos. MongoDB no almacena datos en tablas como se hace en bases de datos relaciones, sino en un esquema dinámico conocido como colecciones.

- Linux Mint 20.1 Cinnamon

Sistema operativo de libre uso.

- Visual Studio Code.

Editor de código fuente desarrollado por Microsoft.

- GitHub.

Desarrollador de software para control de versiones de código.

1.4 Estructura y organización del proyecto

En este proyecto se va a hacer uso de diferentes conceptos, protocolos y tecnologías que nos permitirán llegar a conclusiones sobre Bluetooth Low Energy y GPS, entre otras ideas.

Se estudiarán todas las tecnologías implementadas: NodeJS, React Native, BLE, GPS, MQTT y MongoDB Atlas, profundizando en conceptos que nos serán más útiles.

A continuación, llevaremos a cabo el desarrollo de la App android, que será la base de nuestro proyecto. La App recopila la información que posteriormente usaremos.

Se estudiarán las tramas *advertisement* recibidas y la localización del dispositivo del que se esté haciendo uso. Esta información será procesada por el Back-end (gestión y

almacenamiento) y posteriormente enviada al Front-end (filtrado y consulta) dependiendo de lo que solicite el usuario.

Finalmente, haremos pruebas para evaluar el desempeño del proyecto. Se llevarán a cabo en entornos reales y coherentes con el objetivo final del sistema como puede ser una habitación y una empresa con varias oficinas y empleados.

Se hará un análisis de los datos obtenidos y se filtra de acuerdo a la conveniencia del usuario. Lo veremos en los próximos capítulos.

Capítulo 2. Tecnologías Empleadas

2.1 NodeJS

2.1.1 ¿Qué es NodeJS? ¿Cómo funciona?

NodeJS[5] es un entorno de tiempo de ejecución basado en JavaScript basado en eventos asincrónicos diseñado para crear aplicaciones web escalables. Permite establecer y administrar múltiples conexiones al mismo tiempo sin preocuparse por posibles bloqueos de procesos.

Está inspirado en sistemas como *Event Machine* (de Ruby) o *Twisted* (de Python). A diferencia de estos, este entorno presenta el bucle de eventos como una construcción en tiempo de ejecución en lugar de una biblioteca. No será visible para el usuario.

NodeJS consta de características muy favorables a nivel de usuario como son:

- **Velocidad**. La biblioteca que presenta NodeJS es muy rápida en ejecución de código debido a su motor de JavaScript V8 de Google Chrome.
- **No búfer**. NodeJS no almacena en búfer, sino que genera datos en trozos (conocidos como *chunks*)
- **Asincronismo y control por eventos**. Las bibliotecas que proporciona NodeJS son asíncronas, sin bloqueo. Un servidor basado en el entorno NodeJS no se va a quedar esperando a que una API devuelva datos, un mecanismo de notificación de eventos ayuda al servidor a obtener una respuesta de la llamada a la API anterior.
- **Subproceso escalable**. NodeJS hace uso de un único modelo con un subproceso el cual se compone de un bucle con eventos. Este mecanismo hace que el servidor pueda responder sin bloqueos.

2.1.2 ¿Por qué usar NodeJS?

Es recomendable el uso de este entorno porque puede ejecutarse en una gran variedad de servidores, gran escalabilidad en plataformas que así lo requieren, un alto rendimiento, lenguaje de fácil comprensión, una de las mejores opciones para aplicaciones en tiempo real, etc.

2.1.3 Ejemplo entorno NodeJS

En el siguiente ejemplo, donde enseñamos 'Hola mundo' al cliente, se podrían mantener muchas conexiones simultáneamente. Por cada conexión al servidor, se activará un callback. En el caso de que no haya trabajo para NodeJS, este se dormirá.

Esto sería un ejemplo básico que contrasta con el modelo de concurrencia más comun de hoy en día: el uso de hilos. Esto es realmente ineficiente y difícil de usar.

```
const http = require('http');


const hostname = '127.0.0.1';

const port = 3000;


const server = http.createServer((req, res) => {

 res.statusCode = 200;

 res.setHeader('Content-Type', 'text/plain');

 res.end('Hola Mundo');

});


server.listen(port, hostname, () => {

    console.log(`El servidor se está ejecutando en
http://${hostname}:${port}/`);

});
```

Figura 2. Ejemplo entorno NodeJS

2.2 React

2.2.1 ¿Qué es React? ¿Cómo funciona?

React[6] es una biblioteca JavaScript de código abierto para crear interfaces de usuario diseñadas para facilitar el desarrollo de aplicaciones de una sola página. Fue lanzado en 2013 y desarrollado por Facebook.

Es una de las tecnologías front-end más usadas a día de hoy. Su elemento más importante es el componente, que es, en esencia, una pieza de la interfaz de usuario. Cuando se diseña una aplicación bajo esta biblioteca lo que se hace realmente es crear componentes independientes y reusables para, poco a poco, crear interfaces de usuario un poco más complejas.

2.1.2 Arquitectura de React

Toda aplicación en React tiene como mínimo un componente, que normalmente es conocido como el componente "raíz". Este elemento tiene otros componentes que serán los componentes "hijos", y estos a su vez otros, y así sucesivamente. La vista de la aplicación será un árbol de componentes como el de la Figura 3.

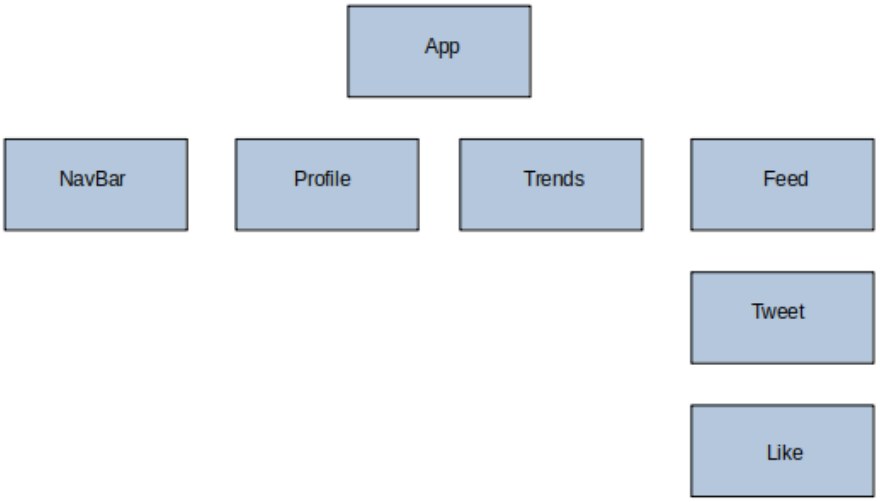

Figura 3. Diagrama ejemplo árbol componentes Twitter

En React existe una representación llamada *VirtualDOM*, donde nuestro JSX (donde definimos los documentos HTML) transforma los documentos en componentes del navegador a través del lenguaje JavaScript.

Un componente puede tener un método render que será el responsable de describir cómo será la vista de la aplicación. En este caso, React rompe las buenas prácticas que se han ido utilizando a lo largo de los últimos años tras juntas vista con lógica. Aún así, será a pequeña escala y se ve, dentro de esta tecnología, más como una ventaja.

2.3 React Native

2.3.1 ¿Qué es React Native? ¿Cómo funciona?

React Native[7] es un framework de programación de aplicaciones nativas multiplataforma basado en JavaScript y ReactJS.

En React Native tenemos nuestros componentes JSX distintos a los componentes HTML, tendrán tags y nombres diferentes. El compilador de React Native va a convertir

los elementos nativos de la interfaz para Android, en nuestro caso (también se puede para iOS).

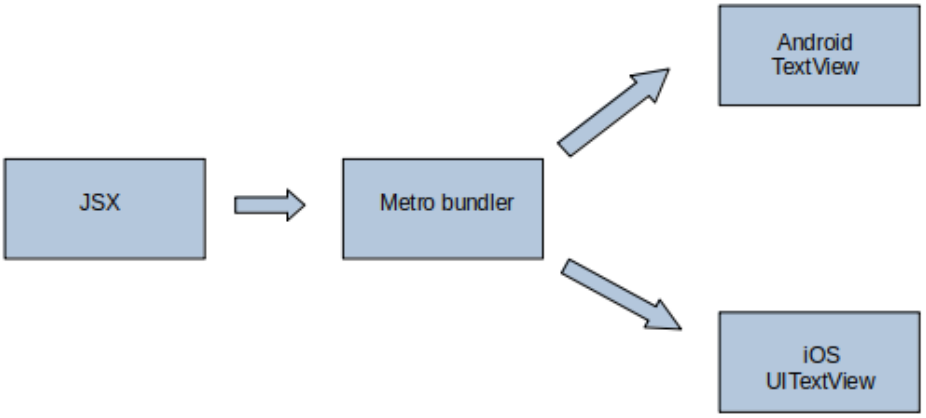

Figura 4. "VirtualDOM" en React Native

Metro bundler es una especie de *pack* que agrupa todos los ficheros de JavaScript en un solo.

2.3.2 React vs React Native

La gran diferencia destacable entre ambas tecnologías es que React se trata de una biblioteca de código abierto, mientras que React Native es un framework. React Native está basado en React.

La pregunta del millón, ¿cuál de los 2 es mejor? Eso dependerá del tipo de aplicación de software en la que se esté trabajando. Si por ejemplo se está pensando en crear una aplicación de software web simple, ReactJS será la tecnología ideal. Por otro lado, si se desea desarrollar una aplicación móvil native real que funciona como cualquier otra aplicación móvil desarrollada con un SDK, React Native será la mejor opción. En nuestro caso, usamos React Native pero es necesario tener unos conocimientos base de React ya que depende de esta biblioteca.

2.4 Bluetooth Low Energy

2.4.1 ¿Qué es BLE? ¿Cómo funciona?

BLE (Bluetooth Low Energy) es una variante del Bluetooth clásico que conocemos. Una puesta en marcha con pros y contras pero que, poco a poco, está haciéndose un hueco en la tecnología.

Usa un cifrado de 128 bits más la autenticación. A la hora de la transmisión ofrece más seguridad gracias al AFH (salto de frecuencia adaptable) que incluye corrección de errores hacia adelante. Esto se conoce como FEC y permite que el receptor pueda corregir los errores sin necesidad de reenvíos. Además, también ofrece canales con ancho de banda estrecho. En resumen, cuando un dispositivo establece una conexión obtenemos seguridad. Pero, ¿cómo se conectan?

En la tecnología BLE esta conexión se conoce como emparejamiento. Los dispositivos BLE mandan tramas de información de datos conocidas como *advertisements* cuya transmisión es obligatoria y su función será hacer que los centrales conozcan de su existencia. En este caso, el central será la aplicación móvil basada en React Native. Estos *advertisements* ofrecen información como qué dispositivo es, qué hace, cuáles son sus características, etc. Este intercambio de datos NO está encriptado.

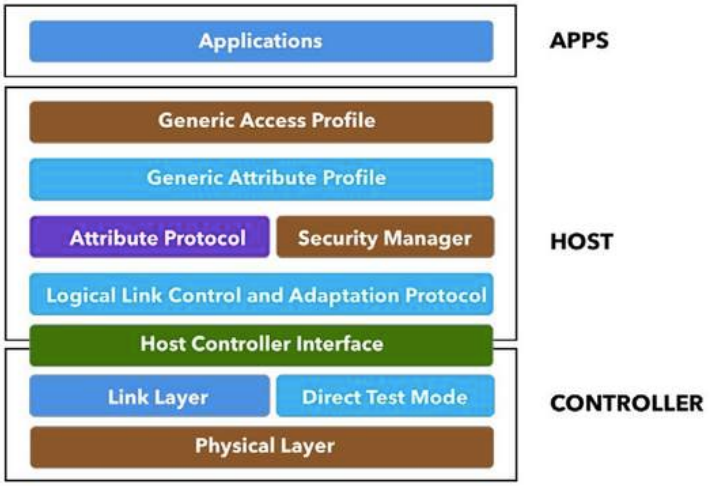

Figura 5. Arquitectura Bluetooth Low Energy

Para llegar a comprender del todo estas tramas, es recomendable dar un paso hacia atrás y aprender sobre una de las capas de la arquitectura BLE: GAP (Generic Access Profile)

Esta capa proporciona un framework donde se define cómo los dispositivos BLE interactúan entre ellos: el rol que tendrán, las tramas advertisement (broadcast, parámetros, datos, etc), establecimiento de conexión y seguridad.

4 roles principales de un dispositivo BLE			
Central	Periférico	Broadcaster	Observador
Descubre periféricos y broadcasters. Tiene la opción de conectarse a los periféricos	Alerta de su presencia y acepta conexiones con un central	Envía paquetes de tramas advertisement. No tiene la opción de conexión	Descubre periféricos y broadcasters. No tiene la opción de aceptar conexiones de un central

Tabla 1. Roles dispositivos BLE

Un periférico y un broadcaster siempre comienzan con las tramas de alerta antes que con una conexión. De hecho, la única forma para que un central sepa de la existencia de estos es a través de los *advertisements*.

Para saber si un dispositivo BLE está en modo conexión o en modo descubrimiento tendremos que ver si la transferencia de datos es bidireccional o no. Es decir, si un periférico o un broadcaster no es capaz de recibir ningún dato del central, se encuentra en modo advertisements. Por otro lado, si se permite la admisión de datos en ambos sentidos querrá decir que se encuentra en modo conexión.

Adentrándonos en el modo descubrimiento (*advertisement*), un dispositivo será el encargado de enviar paquetes que contengan los datos útiles. Estos paquetes se envían en un intervalo fijo llamado Advertising Interval. Hay 40 canales de RF en esta tecnología, cada uno lo separa unos 2 MHz (centro a centro)

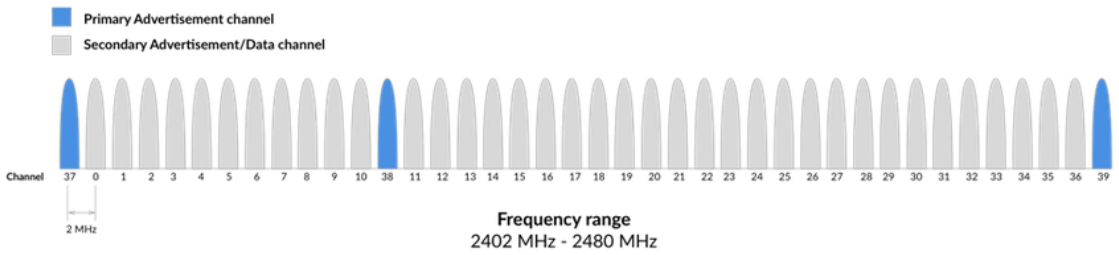

Figura 6. Canales Advertisement en BLE

Como se observa en la Figura 6, hay 2 tipos de canales: Canales Advertisement Primarios y Canales Advertisement Secundarios, pintados en azul y sin pintar, respectivamente.

Los Canales Advertisement Secundarios serán canales auxiliares, es decir, siempre que los dispositivos puedan tendrán que usar los primarios.

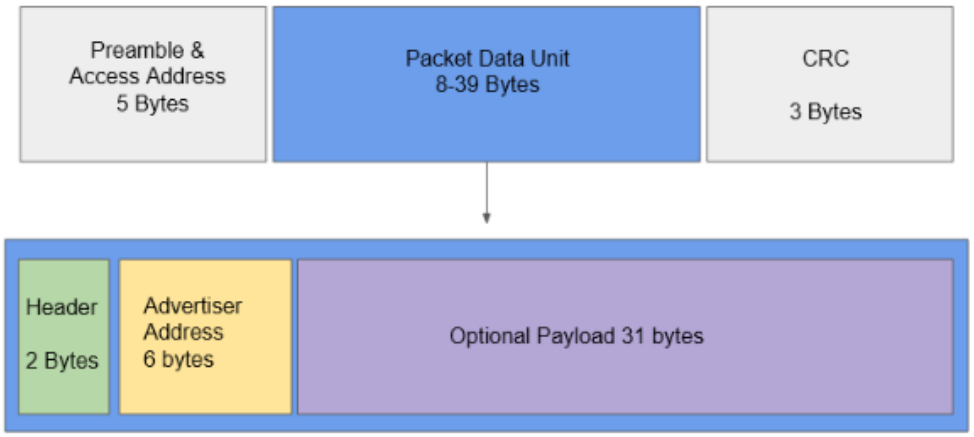

Figura 7. Formato de trama beacon

La Figura 7 representa el formato de paquete de una trama beacon. El campo PDU es el campo que más nos interesa ya que es el campo que cambia respecto al tipo de paquete que se esté transmitiendo: datos o alertas.

Se pueden transmitir 7 tipos distintos de mensajes que varían dependiendo del formato del payload y su función y se definen en el primer byte de la cabecera PDU.

ADV_IND	0X0000
ADV_DIRECT_IND	0X0001
ADV_NONCONN_IND	0X0010
ADV_SCAN_IND	0X0110
SCAN_REQ	0X0011
SCAN_RSP	0X0100
CONNECT_IND	0X0101

Tabla 2. Tipo de mensaje de acuerdo con primer byte del campo PDU

Los campos que siguen a la cabecera PDU son: bit RFU, MAC Type y 1 byte restante que denotará la longitud total entre el campo Optional Payload y el campo Advertiser Address (se observan en la Figura 7)

La segunda fase de emparejamiento se encarga de generar e intercambiar claves. Esta es la fase más vulnerable ya que, si las conexiones BLE no están aseguradas correctamente los atacantes pueden tomar el control de dichos dispositivos y de los datos que estos transmiten.

En el momento de la conexión se definen dos conceptos nuevos en BLE: el cliente y el servidor. En nuestro caso, los dispositivos BLE escaneables actuarán como servidores mientras que la aplicación móvil actuará como cliente.

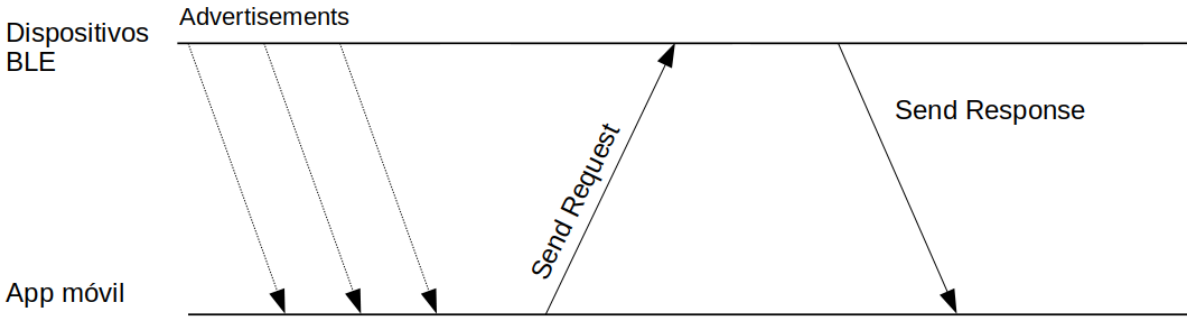

Figura 8. Diagrama emparejamiento BLE

Ahora ocurre la vinculación. El central ha almacenado los datos de autenticación que se intercambiaron anteriormente, lo que permite recordarse mutuamente.

2.4.2 Beacons iBKS comerciales de Accent Systems

Un beacon iBKS105[8] se trata de un beacon Bluetooth Low Energy (BLE) que hace uso de una pila de botón. Está basado en un chipset Nordic Semiconductors nrf51822. Hardware transmisor que se utiliza para el envío de una señal bluetooth de baja energía sin necesidad de sincronización previa.

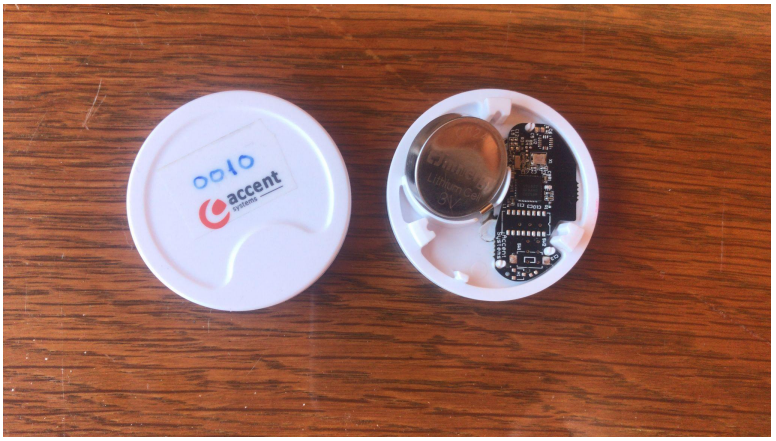

Figura 9. Imagen Beacon iBKS 105

Las pruebas se han llevado a cabo desde la app de escaneo de dispositivos Bluetooth desde Android: nRF Connect for Mobile. Desde ahí estudiaremos los campos informativos del dispositivo iBeacon iBKS105.

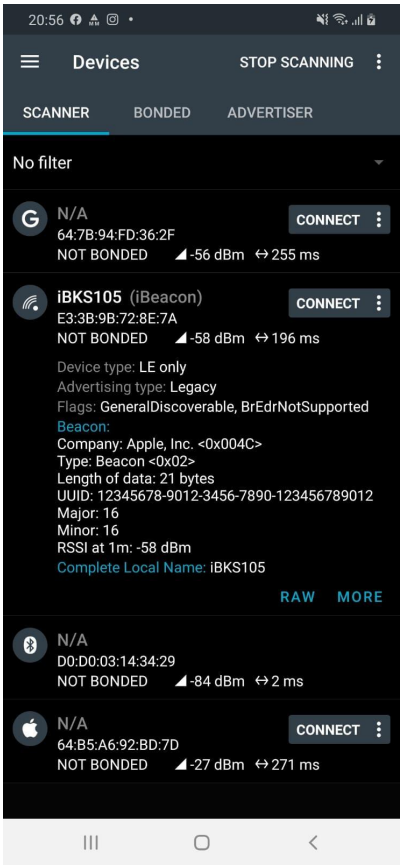

Figura 10. Captura 1 App nRF Connect for Mobile

De todos los dispositivos que encuentra la aplicación nos encontramos con nuestro iBKS105. Para ver los datos en bruto y poder estudiar la trama seleccionamos el dispositivo.

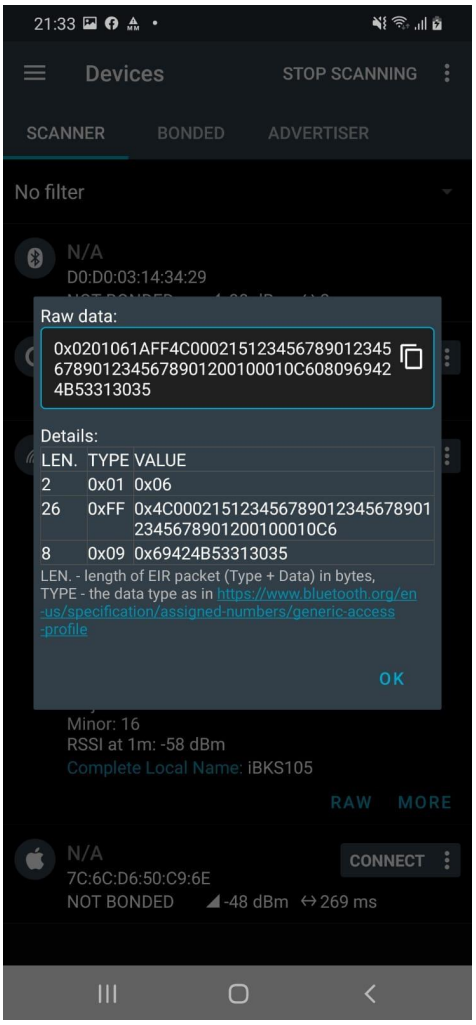

Figura 11. Captura 2 App nRF Connect for Mobile

2.5 GPS

2.5.1 ¿Qué es GPS? ¿Cómo funciona?

El propósito de utilizar la tecnología GPS[9] es conocer la ubicación en coordenadas de latitud, longitud y altitud a medida que escanea nuestro dispositivo. Consta de tres sectores: sector espacial, sector terrestre y sector usuario.

Cada uno de los satélites en órbita cuenta con cuatro relojes atómicos. Se tratan de los relojes más exactos que existen, de hecho, tienen un retraso de 1 segundo cada 3 millones de años. El tiempo es fundamental para poder conocer la posición.

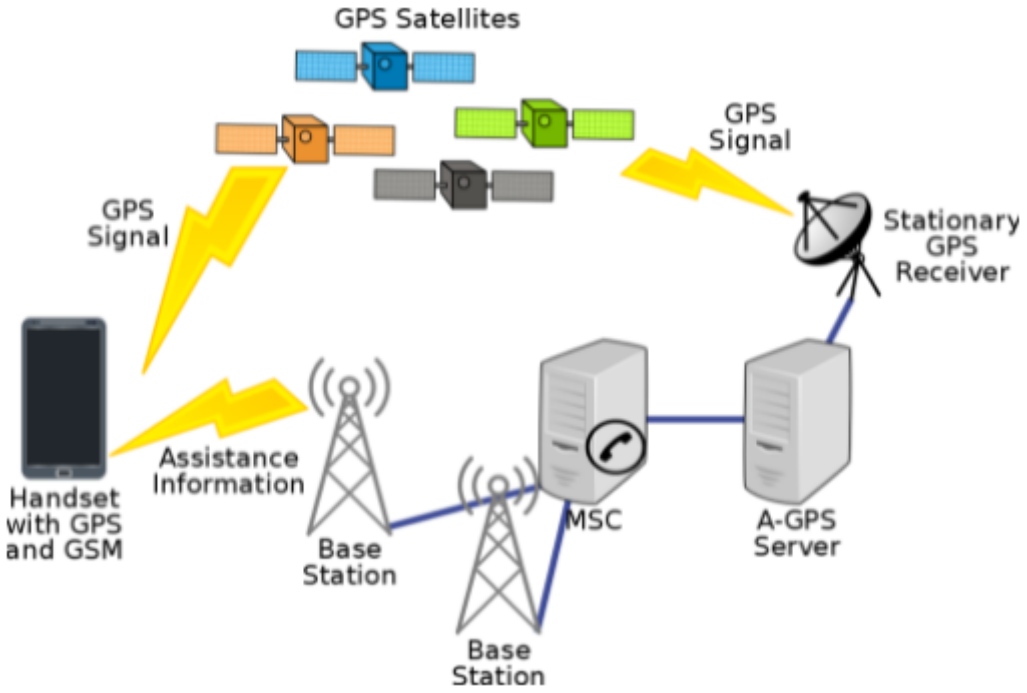

Figura 12. Esquema funcionamiento GPS

La tecnología GPS requiere 4 satélites: tres para calcular la posición y uno para calcular la altitud. Hay un total de 27 satélites en la red GPS, 24 de los cuales están en uso y los 3 restantes se utilizan como respaldo en caso de que uno falle.

Cada satélite transmite 2 señales: una se usa como matriz y la otra se usa para corregir por la ionosfera (la parte de la atmósfera que ayuda a reflejar las ondas de radio emitidas desde la superficie terrestre) El equipo del usuario mide el tiempo que tarda en ello para viajar desde el satélite a la señal de la antena de recepción. Por esta razón, la sincronización es esencial en ambos extremos.

Este tiempo se multiplica por la velocidad de las ondas en la atmósfera para calcular la distancia a cada satélite.

Si hay perturbaciones atmosféricas y cambia la ubicación, está el DGPS, que es el sistema encargado de corregirlas.

Enfocándonos en nuestra App móvil android, nuestro dispositivo móvil tiene un receptor GPS que se conecta con los 4 satélites mencionados anteriormente.

Estar constantemente conectado con los satélites necesita del uso de bastante batería. Para evitar obstrucciones (mala cobertura en zonas por edificios altos u orografía compleja) y ahorrar batería se combina con otra tecnología llamada A-GPS. Es un sistema de asistencia al GPS que permite al usuario combinar la recepción de datos

desde los satélites con la que recogen las antenas de telefonía a las que está conectado el smartphone.

Con Google Maps se ve muy claro: inicialmente cuando ves tu posición aparece un amplio círculo azul al alrededor utilizando los datos del A-GPS. Más adelante, el círculo se va reduciendo según se reciben datos de los satélites GPS.

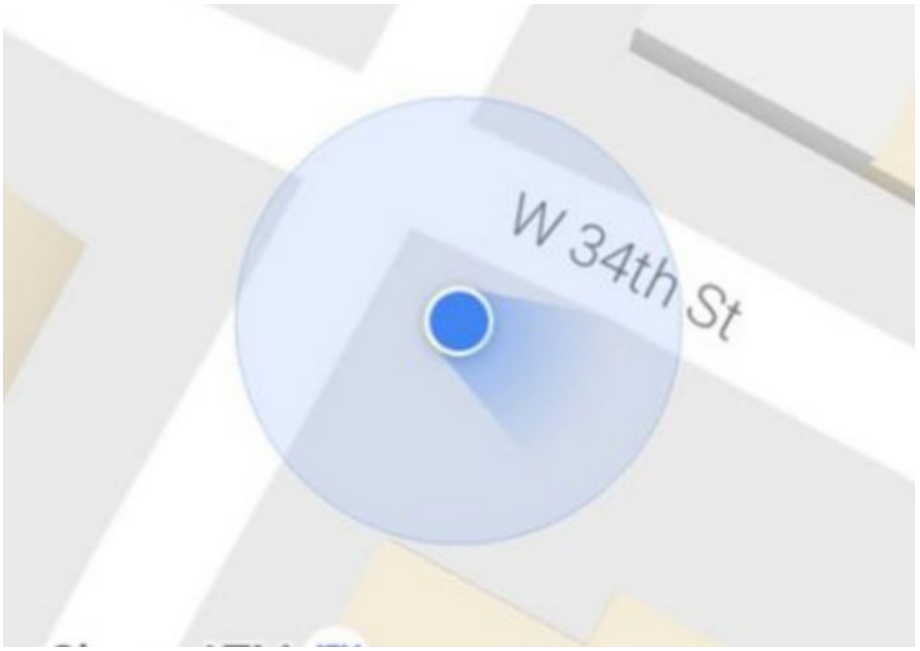

Figura 13. A-GPS en Google Maps

2.6 MQTT

2.6.1 ¿Qué es MQTT? ¿Cómo funciona?

MQTT[10] se trata de un protocolo ligero de comunicación M2M (Machine to Machine) de tipo *message queue* de mucha importancia entre el mundo IoT (Internet of Things). Esto se debe a que los dispositivos usados en este nuevo concepto tienen limitaciones en el uso de energía y ancho de banda, por eso MQTT sería el protocolo idóneo.

Está basado en la pila TCP/IP como base para dicha comunicación. En MQTT, cada conexión se mantiene abierta y se puede reutilizar en cada comunicación.
La idea principal de este protocolo es aportar un servicio de mensajería con patrón publicador/suscriptor.

Los clientes se conectan a un servidor central que denominaremos *BROKER*. El broker se encarga de distribuir la información a los clientes conectados. Todos los clientes pueden ser suscriptores, publicadores, o ambas cosas.

Para poder filtrar los mensajes que son enviados a cada cliente, podrán estar organizados por topics. Un cliente podrá publicar un mensaje en un topic. También los clientes podrán suscribirse a dicho topic, donde el broker les hará llegar los mensajes suscritos.

Los clientes inician una conexión TCP/IP con el broker, la cual se mantendrá abierta hasta que el propio cliente lo desee. El cliente enviará un mensaje *CONNECT* con la información necesaria para poder hacer la conexión. Esta conexión se realizará a través de los puertos 1883 o 8883. El broker responderá con un mensaje *CONNACK* que dirá si es una conexión aceptada o rechazada.

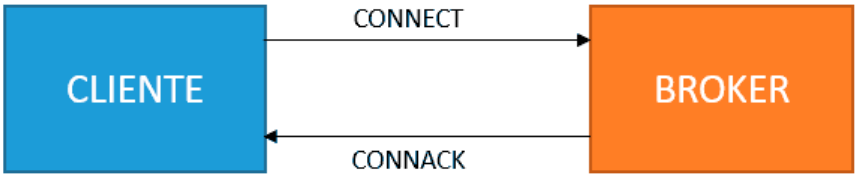

Figura 14. Esquema conexión MQTT Cliente-Broker

El cliente que quiera enviar mensajes deberá enviar un *PUBLISH* que contenga el topic y el payload.

Figura 15. Esquema publicación MQTT Cliente-Broker

En el caso de que un cliente desee suscribirse a un topic, deberá enviar un mensaje SUSCRIBE que será respondido por el broker con un mensaje SUBACK

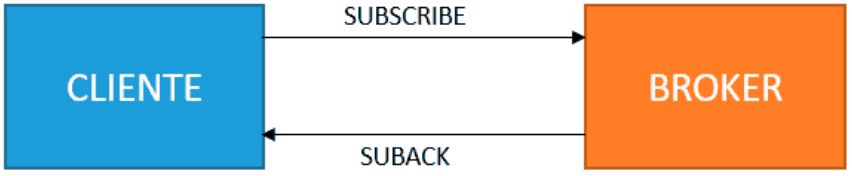

Figura 16. Esquema suscripción MQTT Cliente-Broker

Estos son los mensajes que más veremos en el proyecto. Existen otros como serían PINGREQ y PINGRESP para saber si hay conexión, DISCONNECT para desconectar de la comunicación, etc.

MQTT tiene un mecanismo QoS que elegirá la calidad de servicio al broker en cada conexión. Esta calidad no va a afectar al manejo de las transmisiones de datos, sólo afectará a los clientes MQTT.

Qos 0	QoS 1	QoS 2
Solo se envía una vez. No hay una seguridad 100% de que haya llegado correctamente	Se envía el mismo mensaje hasta que se recibe una garantización de la entrega	Se asegura cada mensaje de que ha llegado correctamente

Tabla 3. QoS MQTT

Decimos que MQTT es un protocolo ligero porque cada mensaje consiste en un encabezado fijo de 2 bytes, un encabezado opcional, una carga útil del mensaje de información (limitada a 256 MB) y el nivel de calidad de servicio (QoS)

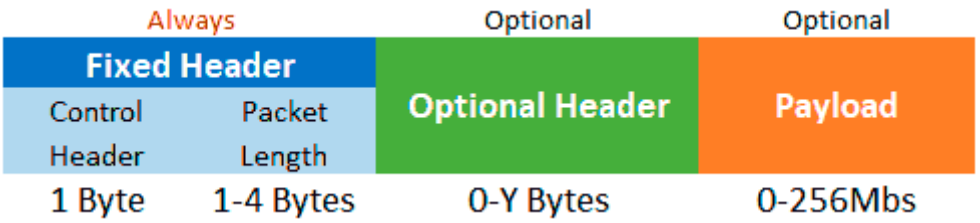

Figura 17. Campos mensaje MQTT

2.6.2 Broker

Los dispositivos IoT pueden soportar mucha información de una vez, por tanto, para poder gestionar bien dicha información necesitamos un elemento para ello que se conocerá como Broker. De esta manera se puede garantizar toda la comunicación dentro del sistema.

Como funciones tiene:

- Organización de la información
- Garantizar comunicación entre todas las entidades del sistema
- Obtener información de todos los datos cliente - sistema
- Recibir mensajes enviados por clientes y distribuirlos en el sistema Publisher/Subscriber

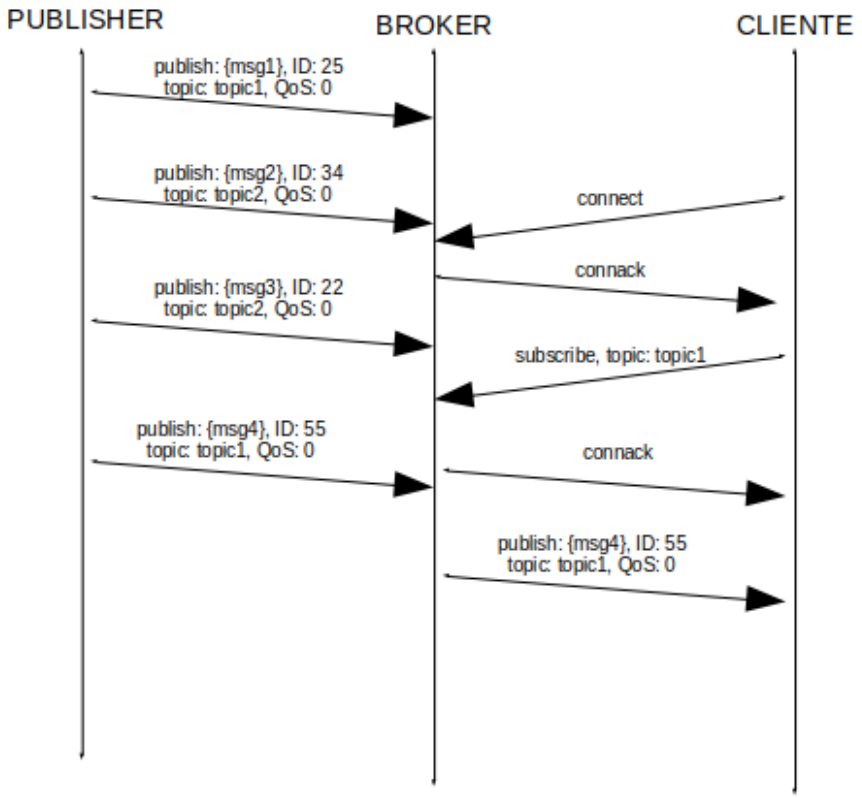

Figura 18. Cronograma MQTT

2.6.3 MQTT vs HTTP

Para entender mejor el protocolo MQTT haremos una comparativa con HTTP. HTTP es un protocolo con el que llevamos familiarizados toda la carrera, es por esto que el concepto de MQTT puede entenderse mejor de esta manera.

MQTT	HTTP
Modelo publicador/suscriptor Mantenimiento de la conexión Conexión TCP full-duplex Datos a nivel de byte Sencillo y ligero (Figura 17) 3 tipos de QoS	Modelo cliente/servidor Conexión por petición Conexión TCP half-duplex Transmisión a nivel de documentos Complejo pero mejor para mayor control de comunicación Sin QoS

Tabla 4. Campos mensaje MQTT

2.6.4 Entorno MQTT Ejemplo en NodeJS

Este ejemplo está basado en lenguaje Javascript

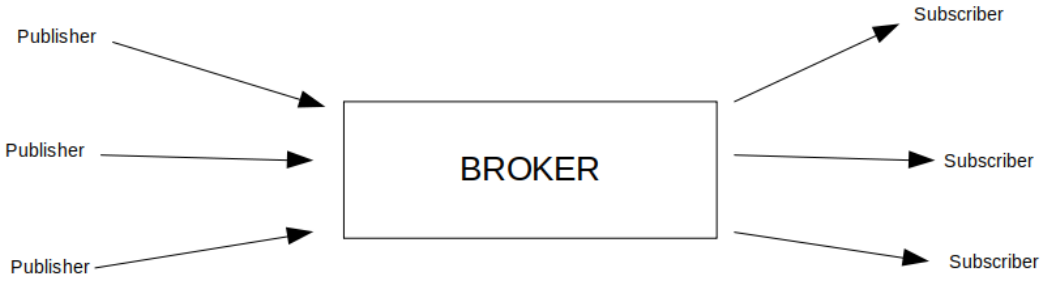

Figura 19. Esquema ejemplo MQTT

```
// MQTT broker

var mosca = require('mosca')
var settings = {port: 1883}
var broker = new mosca.Server(settings)


broker.on('ready', () => {
   console.log('Broker is ready!')
})
```

Figura 20. Ejemplo MQTT Broker

```
// MQTT publisher

const mqtt = require('mqtt');
const client = mqtt.connect('mqtt://localhost:1883');
const topic = 'test'
const message = '1'

client.on('connect', () => {
   setInterval(() => {
       client.publish(topic, message)
       console.log('Message sent!', message)
   }, 5000);
})
```

Figura 21. Ejemplo MQTT Publisher

```
// MQTT subscriber

var mqtt = require('mqtt')
var client = mqtt.connect('mqtt://localhost:1883')
var topic = 'test';

client.on('message', (topic, message) => {
   message = message.toString()
   console.log(message)
})

client.on('connect', () => {
   client.subscribe(topic)
})
```

Figura 22. Ejemplo MQTT Subscriber

Cada 5 segundos un Publisher se encargará de enviar el mensaje *1* al topic *test*. Cuando se envíe el mensaje se mostrará por la terminal del Publisher '*Message sent!*', de esta manera sabemos que se está realizando todo correctamente.

El Broker recibe la trama. La enruta hacia un Subscriber que esté suscrito al topic *test*.

El Subscriber tiene 2 párrafos en el código:

- Se encarga de recibir el mensaje del *Publisher* y además, convertirlo a string para poder mostrarlo por consola (es una manera de corroborar que todo funciona correctamente)
- Se suscribe al topic que desee, en este caso será *test*

2.7 MongoDB

2.7.1 ¿Qué es MongoDB? ¿Cómo funciona?

MongoDB es una de las bases de datos NoSQL (Not only SQL) más representativas. Almacena documentos JSON a diferencia de las tablas de base de datos relaciones donde encontramos registros.

- Velocidad. MongoDB, al ser una BBDD documental, es capaz de ser mucho más rápida. Puede atender clientes que necesiten realizar muchas operaciones por segundo.
- Volumen. Una BBDD relacional, al depender unas tablas de otras, funcionarán por lo general más lentamente y las cantidad de registros serán mayores. Esto

conlleva a buscar soluciones paralelas como son la división de las tablas en segmentos, por ejemplo. Con MongoDB no pasa, es capaz de administrar volúmenes muy grandes de datos en sus entidades.

- Variabilidad. Con MongoDB no hay problema en que cada documento almacene campos distintos. Hay mucha flexibilidad en cuanto al esquema de la información.

2.7.2 MongoDB Atlas

MongoDB Atlas[11] es el servicio de MongoDB en la nube. Permite crear y administrar una base de datos MongoDB desde cualquier lugar del mundo a través de su plataforma. La ventaja que ofrece es el no tener que administrar la BBDD al completo como sería llevar a cabo instalaciones, actualizaciones, administración de usuarios, etc. Nosotros, como cliente del servicio, nos olvidamos de cómo funciona la infraestructura de MongoDB y nos enfocamos en crear la gestión de la BBDD.

Atlas está disponible en más de 70 regiones de AWS, GCP y Azure.

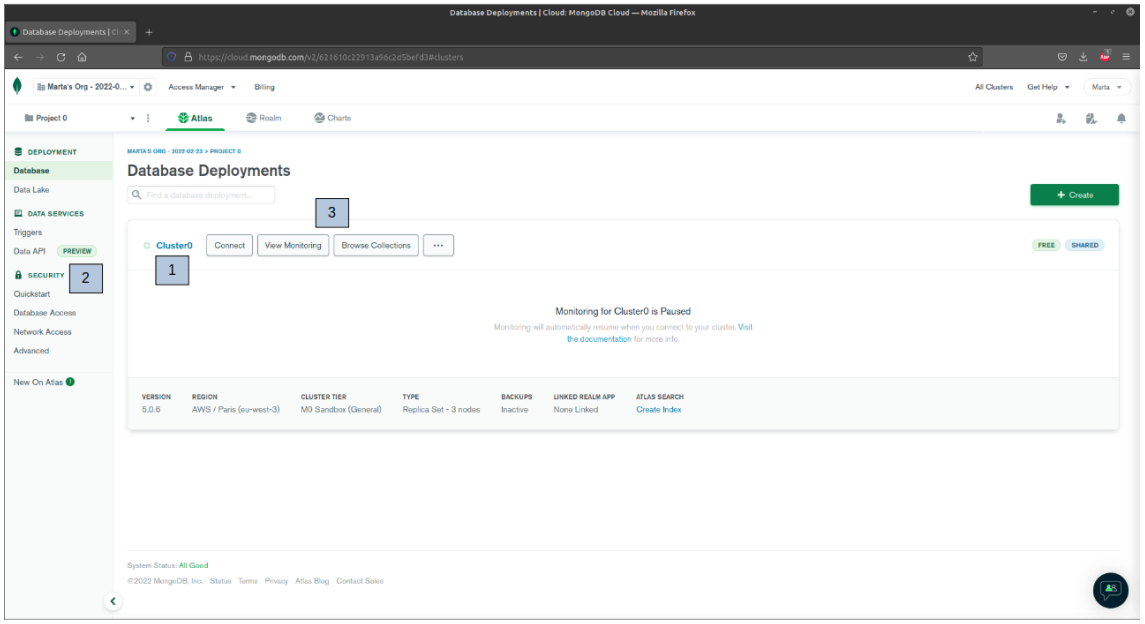

Figura 23. Interfaz usuario MongoDB Atlas

MongoDB Atlas trabaja a través de Clusters(1). Un Cluster es un conjunto de BBDD, y para comenzar a trabajar con este servicio debemos crear uno (a raíz de un proyecto nuevo).
Para construir nuestro primer Cluster, tendremos que rellenar algunos datos además de indicar en qué lenguaje de programación nos conectaremos a la base de datos.

MongoDB Atlas proporciona bastantes medidas de seguridad(2) integradas para evitar accesos no deseados a nuestra BBDD. Una de ellas (quizás la más destacable) es el

poder dar permiso a una dirección IP en concreto. Esta opción la tenemos en nuestro dashboard.

En la interfaz usuario tenemos una serie de botones(3) que nos ayudarán a conseguir la URL para llevar a cabo la conexión, visualizar las colecciones de nuestro cluster, etc.

2.7.3 Ejemplo almacenamiento en MongoDB Atlas con base Nodejs

Para terminar de enlazar los conceptos teóricos con la práctica, he creado un script ejemplo que añada una colección a una BBDD en MongoDB Atlas. El lenguaje usado ha sido Typescript.
Con un await, cuando el cliente creado en MongoDB se conecte a la URL proporcionada este mismo creará una BBDD nueva donde insertará una colección que sigue la distribución del *interface* declarado previamente.

```
try {
   await client.connect();

   const database = client.db("DB_Ejemplo");
   const ble = database.collection<EJEMPLO>("ejemplo");
   const result = await ble.insertOne({
     name: "Ejemplo_1",
     content:"Hola mundo",
   });
   console.log('A document was inserted');
 } finally {
   await client.close();
 }
}
```

Figura 24. Almacenamiento en MongoDB Atlas

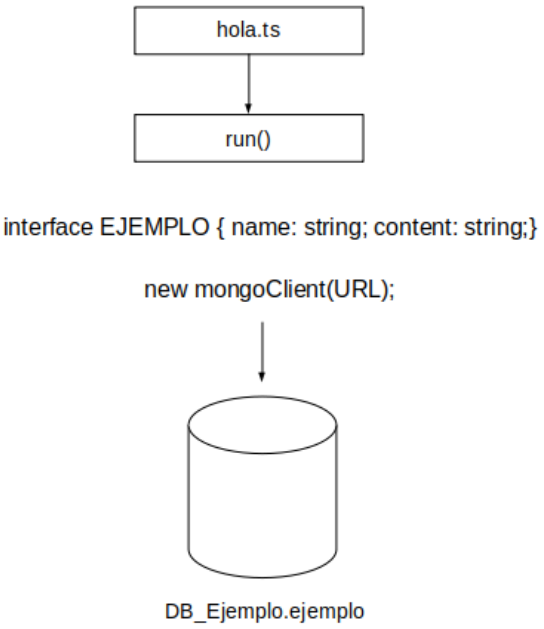

Figura 25. Diagrama flujo almacenamiento MongoDB Atlas

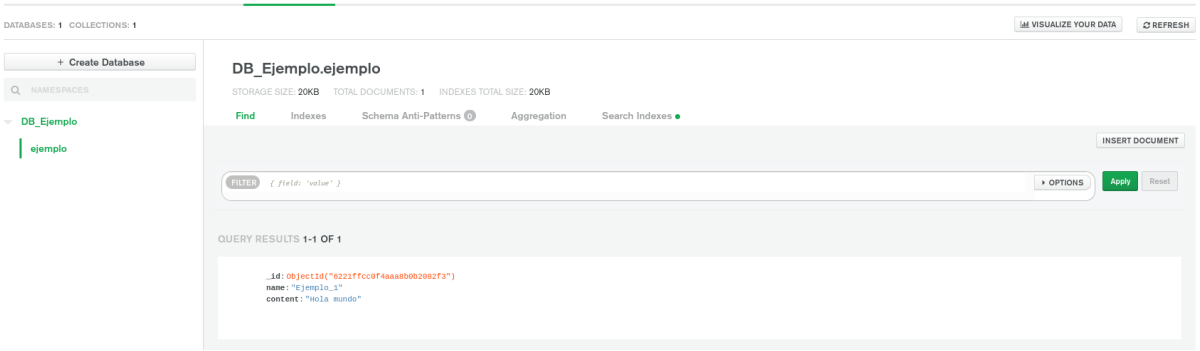

Figura 26. Comprobación interfaz usuario ejemplo

Capítulo 3. Diseño e implementación de App React Native

3.1 Estructurado del proyecto

La estructura del proyecto se ha separado en 2 bloques principales: Vista y Controlador, influenciado por el diseño MVC que ya conocemos.

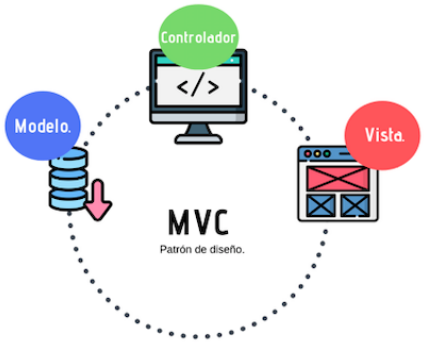

Figura 27. Diseño MVC

El componente raíz de nuestro proyecto será el componente *index.tsx* ubicado en la carpeta raíz del proyecto *src*. En este fichero importamos todos elementos de la aplicación que serán usados en su desarrollo: vista de la parte de escaneo de dispositivos BLE, vista de la parte de localización GPS y vista del módulo MQTT.

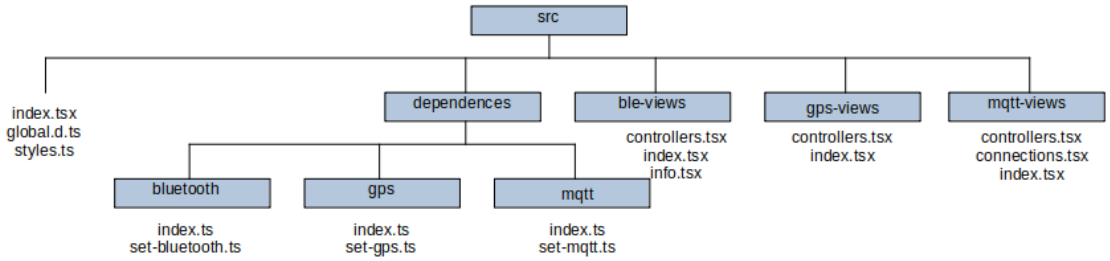

Figura 28. Esquema de la estructura de la App React Native

La carpeta *dependences* estará compuesta por ficheros donde se declaran objetos tipo para los elementos que queremos obtener a lo largo del proyecto. En el caso de *bluetooth*, se crea un objeto tipo *BluetoothDevice* compuesto por dos string (*id* y *name*). Por otro lado, en gps se crea un objeto tipo *GPSInformation* compuesto por dos string y un number (*id*, *valor* y *device*)

```
export type BluetoohDevice = {

   id: string;

   name: string;

};
```

Figura 29. Objeto tipo BluetoohDevice

```
export type GPSInformation = {

   id: string;

   valor: number;

   device: string;

};
```

Figura 30. Objeto tipo GPSInformation

Por otro lado, tenemos las carpetas *ble-views*, *gps-views* y *mqtt-views*. Sus ficheros se encargan de integrar la parte de gestión de las dependencias con el controlador y la vista.

La base del proyecto se encuentra en la aplicación React Native. Esta se resume en 3 elementos principales:

- Scanner BLE
- Searcher GPS
- MQTT

Cuando la aplicación se ejecuta, de manera invisible para el usuario, se estará escaneando dispositivos BLE y buscando localización GPS en bucle. Como se ha mencionado previamente, las tecnologías de las que se está haciendo uso son de bajo consumo lo que no supondrá un gasto excesivo de batería en nuestro dispositivo.

Si el usuario lo desea, podrá visualizar los resultados de ambas búsquedas a través de unos botones ubicados en pantalla.

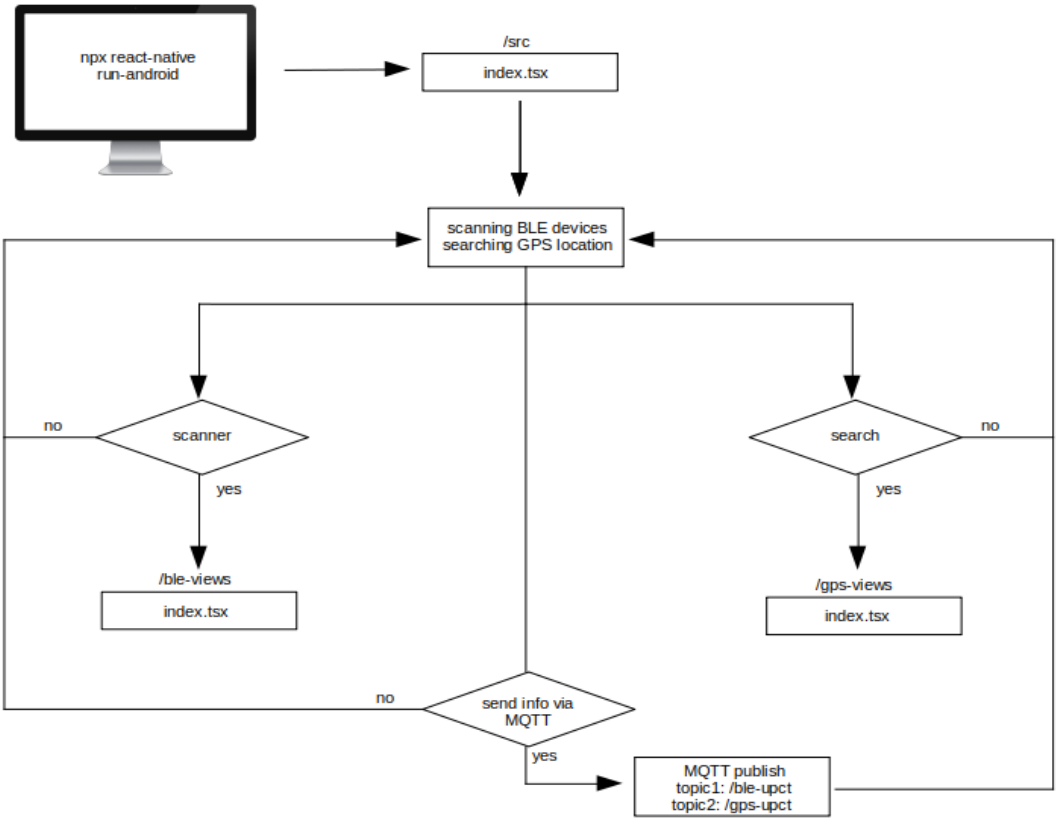

Figura 31. Diagrama de flujo funcionamiento app React Native

La información se enviará de forma paralela vía MQTT en 2 topics diferentes:

- Topic para Scanner BLE: */ble-upct*
- Topic para Searcher GPS: */gps-upct*

El servidor recibe los datos en bruto.

3.2 Scanner BLE

Este módulo se encargará de hacer el escaneo de dispositivos BLE.

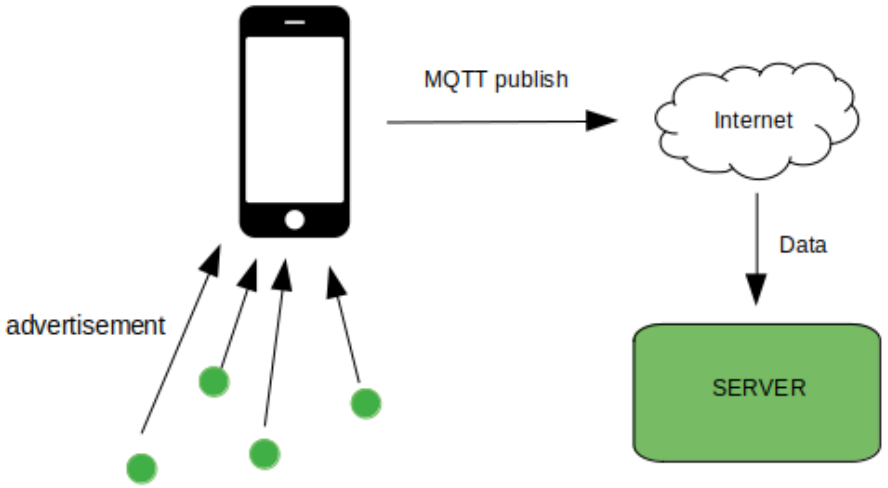

Figura 32. Esquema comunicación BLE

Los atributos de la función de escaneo que ofrece la librería BLE[12] son: UUIDs, ScanOptions y listener.

- UUIDs. Array que contiene un string con los UUID de los dispositivos escaneados.
- ScanOptions. Configuración opcional para la operación de escaneo.
- Listener. Función que será llamada por cada dispositivo escaneado, teniendo en cuenta que un dispositivo puede ser escaneado varias veces.

bleManager.startDeviceScan(UUIDs, options, listener)

```
bleManager.startDeviceScan(

   UUIDs: ?Array<UUID>,

   options: ?ScanOptions,

   listener: (error: ?Error, scannedDevice: ?Device) => void

);
```

Figura 33. Ejemplo búsqueda dispositivos BLE con startDeviceScan()

Se puede obtener información del dispositivo escaneado.

id	Identificador del dispositivo. Dirección MAC (DeviceId)
name	Nombre del dispositivo (String)
rrsi	Intensidad de la señal recibida (Number)
mtu	Unidad de transmisión máxima para el dispositivo (Number)

Tabla 5. Algunas características BLE escaneado

Una vez el protocolo MQTT forma el mensaje JSON, lo envía a través de un *publish* con una QoS configurada y un *topic* al servidor. El servidor se trata de un Broker EMQ.

EMQ es una plataforma Open Source que sirve para crear un servidor MQTT. EMQ X es un servidor de mensajes IoT MQTT desarrollado en base a la plataforma Erlang / OTP (plataforma de lenguaje en tiempo real suave). Tiene como objetivo lograr una alta confiabilidad y admisión de conexiones MQTT que transportan mensajes de baja latencia entre dispositivos IoT.

El servidor podrá actuar también como suscriptor para poder escuchar un topic en particular, en este caso, */ble-upct*.

3.3 Searcher GPS

Este módulo se encargará de llevar a cabo la búsqueda de localización de nuestro dispositivo.

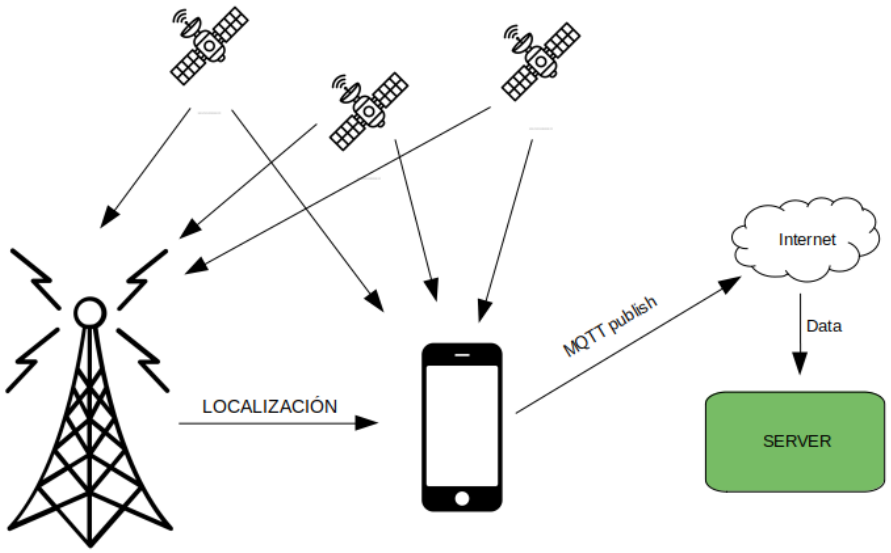

Figura 34. Esquema comunicación GPS

La librería GPS[13] usada para esta parte del proyecto presenta diferentes métodos para realizar la búsqueda de localización del dispositivo en uso. Para obtener la información tenemos 2 métodos específicos: *getCurrentPosition()* y *watchPosition()*. En este proyecto haremos uso del último. La razón es que este devolverá un objeto tipo number conocido como *watchID* que nos será necesario más adelante.

geolocation.watchPosition(success, [error], [options]);

```
Geolocation.watchPosition(
       async (position) => {

           console.log(position.coords.altitude,
position.coords.latitude, position.coords.longitude);
       },

       (error) => {
           console.log(error);
       },
       {enableHighAccuracy: true, interval: 20000, fastestInterval:
0, distanceFilter: 1}
   );
```

Figura 35. Ejemplo localización con método watchPosition()

Este método invoca la devolución de una llamada exitosa cada vez que se cambia la ubicación. Nos mantiene constantemente informados de la localización exacta.

Nombre	Tipo	Obligatorio	Descripción
success	function	Sí	Función invocada cuando la localización cambia
error	function	No	Función invocada cuando ocurre un error
options	object	No	*

Tabla 6. Parámetros método watchPosition()

*El parámetro *options* soporta varias opciones:

- timeout (ms). Representa la longitud máxima de tiempo en milisegundos que el dispositivo tardará en devolver la posición.

- maximumAge (ms). Indica la edad máxima en milisegundos de una posición almacenada que podrá devolverse al usuario.

- enableHighAccuracy (bool). Representa si se va a hacer uso del GPS o no.

- distanceFilter (m). Distancia mínima desde la localización previa que se puede superar antes de devolver una nueva ubicación.

- useSignificantChanges (bool). Indica si se hace uso de APIs nativas para ahorrar batería.

Este método devuelve un número de identificación *watchID* que se puede utilizar para identificar de forma única al observador de la posición solicitada. En el caso de que el usuario así desee dejar de observar la ubicación se podrá usar el método *clearWatch()* que recibirá como atributo este número.

3.4 Módulo MQTT

El servidor MQTT con el que se comunica nuestra App será EMQ X. Su función principal será la de escuchar los eventos *publish* y reenviar los datos que reciba de estos a todos sus suscriptores. El back-end estará suscrito a los 2 topic del proyecto: */ble-upct* y */gps-upct*, por tanto, escuchará a cada momento todas las tramas del protocolo MQTT.

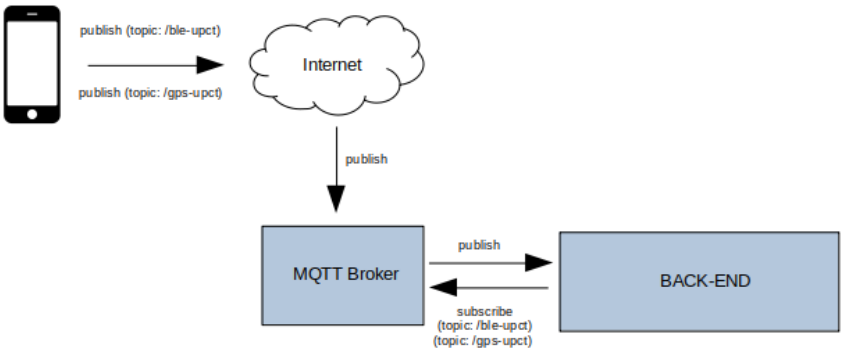

Figura 36. Esquema módulo MQTT

A continuación, vemos una serie de ejemplos de código de la librería MQTT[14] con la que se va a desarrollar este módulo.

1º Creamos cliente MQTT.
Usamos el método *createClient()* que devuelve un objeto tipo mqttClient. Se pueden crear tantos clientes como la memoria nos permita, sin embargo, solo nos podemos conectar a uno en el momento de ejecución. Si intentamos la conexión con un segundo cliente, el programa nos lanzará una excepción cuando se haga llamada al método *connect()*

```
MQTT.createClient({
   uri: 'mqtt://test.mosquitto.org:1883',
   clientId: 'your_client_id'
})
```

Figura 37. Creación cliente MQTT

2º Añadimos escuchador de eventos.
Como eventos tenemos: *connect*, *closed*, *error*, *message*. Se hace uso del método on(event, callback) donde event corresponde a uno de los eventos mencionados previamente y callback será la función a ejecutar cuando el evento salte.

```
client.on('closed', function() {
   console.log('mqtt.event.closed');
 });

 client.on('error', function(msg) {
   console.log('mqtt.event.error', msg);
 });

 client.on('message', function(msg) {
   console.log('mqtt.event.message', msg);
 });

 client.on('connect', function() {
   console.log('connected');
   client.subscribe('/data', 0);
   client.publish('/data', "test", 0, false);
 });
```

Figura 38. Escuchador eventos MQTT

3º Conectamos el cliente.
Se usa el método *connect()*, que pertenece al objeto cliente creado previamente.

4º Suscribimos el cliente.
Se usa el método *subscribe(topic, qos)* si deseamos que actúe como suscriptor de cierto topic.

5º Publicamos con el cliente.
Se usa el método *publish(topic, payload, qos, retain)* si deseamos que el cliente publique algo a cierto topic.

6º Desconectamos el cliente.
Se usa el método *disconnect()*, que pertenece al objeto cliente.

3.5 Resultado final App

La App final quedará como se ve en la Figura 39. Presenta 3 secciones fácilmente diferenciables:

- Envío de datos MQTT

Cuando el empleado quiera mandar su información vía protocolo MQTT al Servidor lo podrá realizar mediante el botón *SENDING*.

- Escáner de dispositivos BLE

Cuando el empleado quiera visualizar los dispositivos que está rastreando a su alrededor podrá hacerlo con el botón *OFF*, que pasará a ponerse en *ON*.

- Localización coordenadas GPS

Cuando el empleado quiera visualizar la localización que está teniendo en ese momento podrá hacerlo pulsando el botón *NOT SEARCHING*.

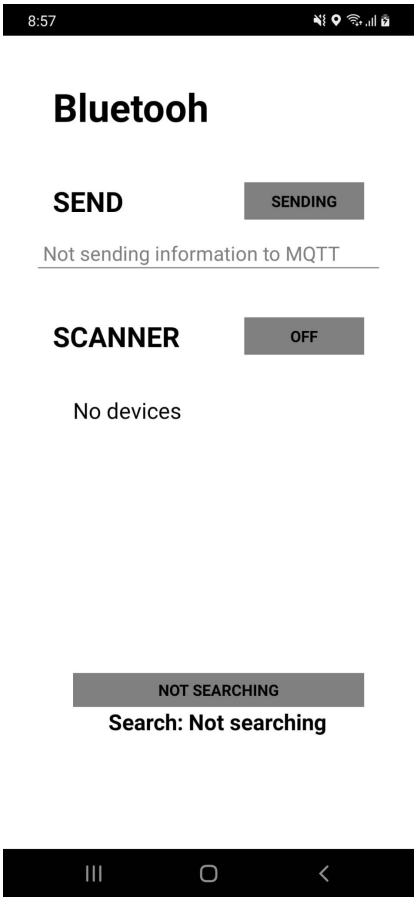

Figura 39. Resultado Final App

Como prueba de funcionamiento de la App tenemos la Figura 40. Si hacemos una comparativa visual entre la Figura 39 y la Figura 40, vemos que el título de los 3 botones ha cambiado. Además, como se ha mencionado, el usuario estará viendo gráficamente la información que se está transmitiendo al Servidor.

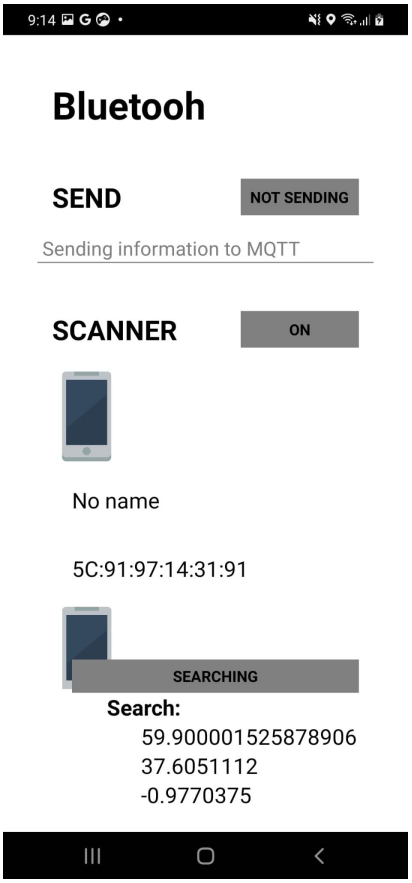

Figura 40. Resultado Final App

Capítulo 4. Diseño e implementación de la arquitectura del sistema

4.1 Arquitectura global del sistema

La arquitectura global del sistema se expone en la Figura 41, pero como elementos principales tenemos: App React Native, Back-end y Front-end.

- App React Native. Elemento clave del proyecto. Se utiliza para escanear dispositivos BLE y buscar su localización. Escucha tramas advertisement y, a través del protocolo MQTT, enviará la información recopilada a un servidor.

Servidor del sistema formado por:

- Back-end. Elemento que cumple con varias funcionalidades. Por un lado, será suscriptor de los 2 topic MQTT que están involucrados con el envío de la información necesaria. Este volcado de información se llevará a cabo gracias al servidor MQTT EMQ X. Es el que se comunica con la App y reenvía los datos a los *suscribers* tras escuchar las tramas de los *publishers*. Por otro lado, será el encargado de la API REST que haga posible la comunicación con el Front-end. Espera peticiones HTML de manera que cuando un usuario desee solicitar información en la interfaz gráfica se pueda llevar a cabo.

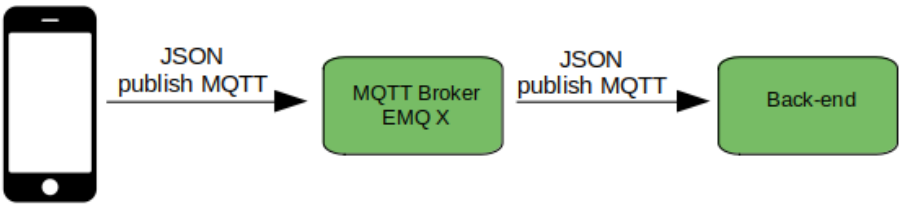

Figura 41. Esquema comunicación servidor MQTT EMQ X

- Front-end. Elemento que lleva a cabo las peticiones para recordar la página que el usuario esté visualizando (ya sea por decisión de este o bien por actualizaciones del sistema)

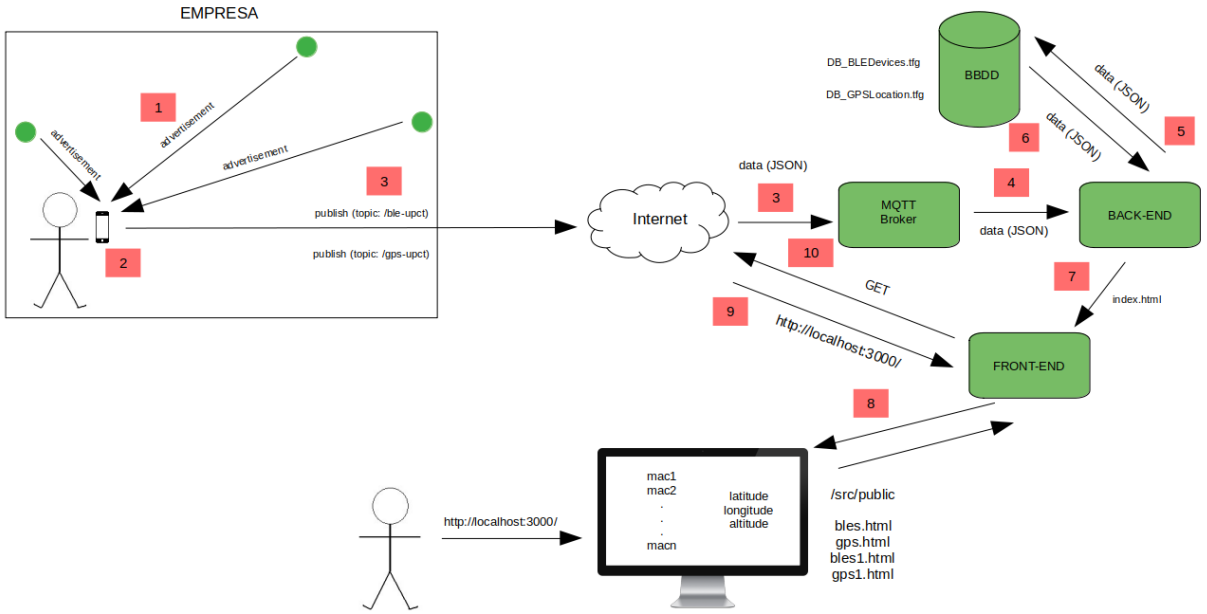

Figura 42. Arquitectura global del sistema

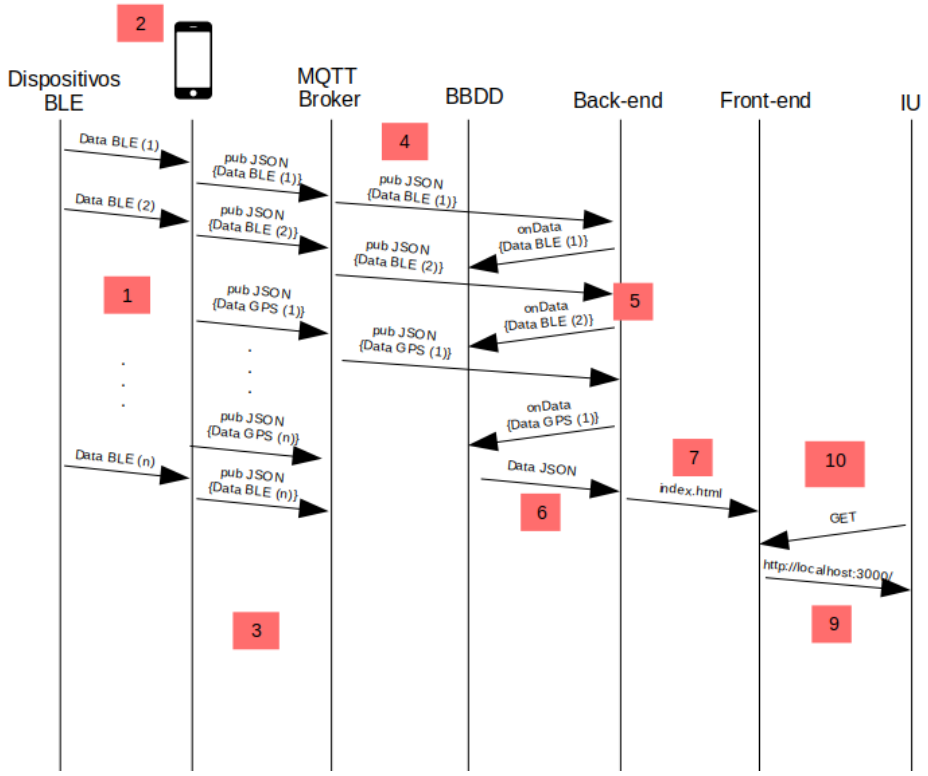

Figura 43. Cronograma global del sistema

A continuación, se explican los pasos indicados en la Figura 42 y Figura 43 para comprender cómo interactúa la App con el sistema y como se cohesionan todas las tecnologías.

1. Los dispositivos BLE envían continuamente tramas *advertisement* (transmisión obligatoria) que serán escuchadas por la App.

2. La App android desarrollada en React Native se encarga de escuchar las tramas del punto 1 y la localización GPS de los satélites. La información necesaria recopilada se mandará vía MQTT

3. La información necesaria recopilada se enviará vía MQTT al Broker. Se enviará en formato JSON e irá dividida en 2 topic diferentes: */ble-upct* y */gps-upct*.

4. MQTT Broker redirigirá esta información a los terminales suscritos a los topic mencionados previamente. Back-end es suscriptor.

5. Esta información, para que no se pierda por la red, será almacenada en una BBDD MongoDB. Será información accesible para cualquier uso, el Back-end podrá solicitarla en cualquier momento.

6. La base de datos mandará dicha información al Back-end cuando ésta lo solicite. Puede ser tanto información de tramas específicas o sobre las últimas actualizaciones.

7. El Front-end hace peticiones cada x tiempo para recargar la página y así mostrar la información más reciente. Cada 5 minutos se actualiza la página.

8. 9. 10. En estos puntos se establece la “comunicación” entre el cliente y el sistema. Cada vez que un usuario hace uso de este, se enviará una petición al Back-end devolviendo la página que así se desee. Esto se llevará a cabo gracias a un servicio REST que hará posible la conexión Back-end - Front-end.

4.2 Diseño Back-end y BBDD

A continuación, en la Figura 44 y Figura 45 vamos a observar los elementos del código de la App android y del Back-end, respectivamente. Servirá para ver de manera superficial las funciones principales de las que se hará uso para la comunicación entre ambas partes del proyecto.

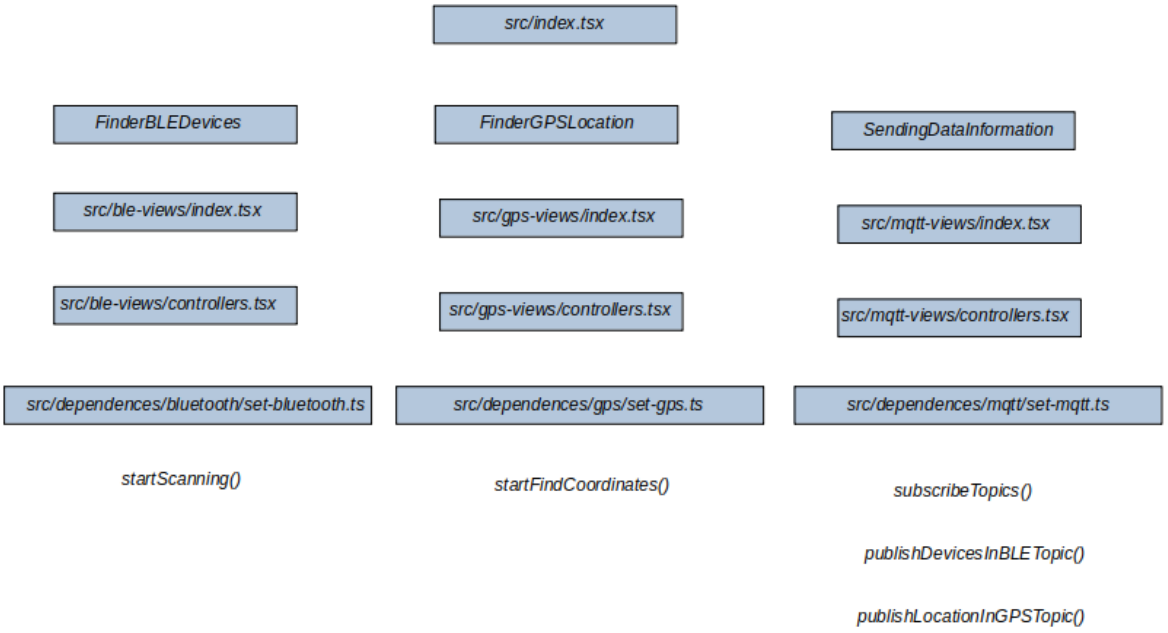

Figura 44. Elementos parte código App android

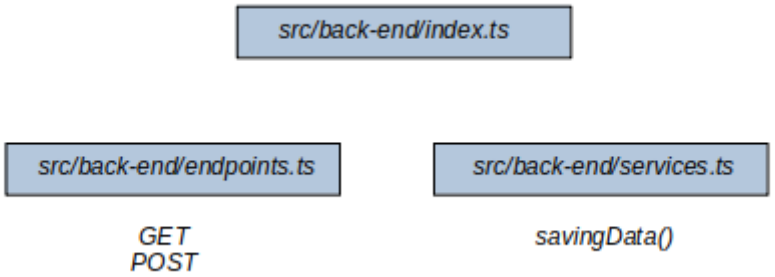

Figura 45. Elementos parte código Back-end

Describimos el comportamiento entre módulos del Back-end y de la App.

La App android se pondrá en funcionamiento mediante las funciones *startScanning()* y *startFindCoordinates()*. Por medio del protocolo MQTT manda tramas *publish* mediante *publishLocationInGPSTopic()* y *publsihDevicesInBLETopic()*. Contendrán la información recopilada por ambas ejecuciones e irá separada en 2 topic diferentes. El Broker reenviará los datos a los suscriptores de los 2 topic. El Back-end es suscriptor y el script *index.ts* ubicado en el directorio *src/back-end* es el que actúa como tal. Escuchará las tramas *publish* gracias a *subscribeTopics()*. Con la librería MongoDB y la función *savingData()* guardaremos la información en una BBDD sin que se pierda por la red. Cada vez que llegue un mensaje se llamará a la función *savingData()*.

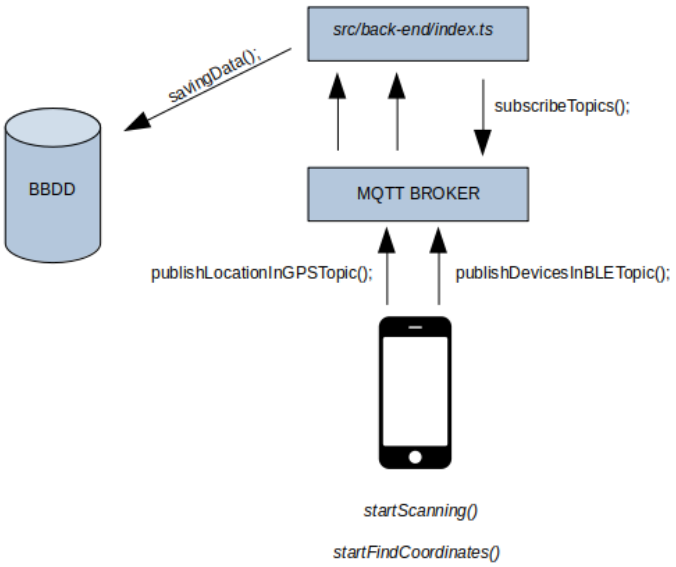

Figura 46. Interconexión módulos Back-end - App

Una vez el usuario tiene acceso a la Interfaz Gráfica, podrá navegar en ella con 4 posibles opciones: Información BLE en tiempo real, información GPS en tiempo real, información BLE filtrada e información GPS filtrada. Poder acceder a esta información se llevará a cabo gracias a operaciones GET a través de una API REST. Estos GET accederán a la BBDD de MongoDB y recolectarán los datos.

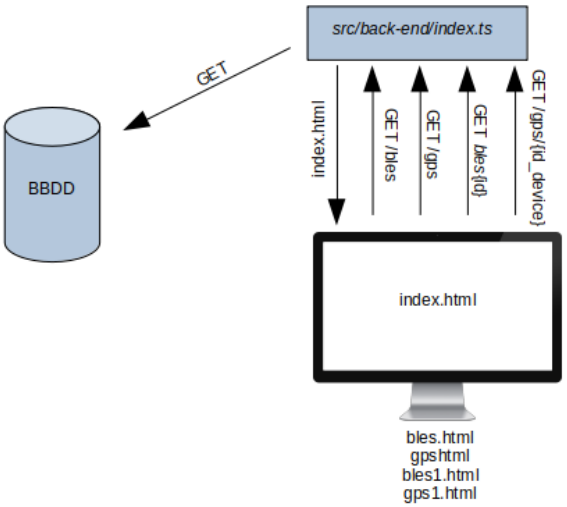

Figura 47. Interconexión módulos BBDD - Front-end

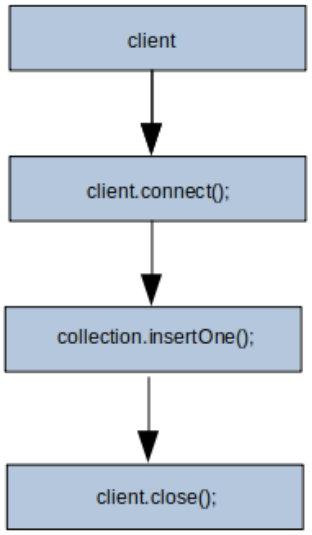

Figura 48. Diagrama flujo INSERT MongoDB Atlas

4.3 Diseño Front-end e Interfaz de Usuario

Las solicitudes que el usuario puede hacer son:

- *GET /bles*

Petición que contiene el contenido actualizado de búsqueda de dispositivos BLE. Devuelve el campo MAC y el campo Name de cada uno de ellos. Esta página se recargará cada 60 segundos.

- *GET /bles/{id}*

Petición que solicita buscar si está escaneado un dispositivo BLE en concreto.

- *GET /bles/{timestamp}*

Petición que solicita buscar los dispositivos BLE escaneados en un timestamp concreto.

- *GET /gps*

Petición que contiene el contenido actualizado de búsqueda de localización. Devuelve latitud, longitud y altitud. Esta página se recargará cada 60 segundos.

- *GET /gps/{id_device}*

Petición que solicita los datos de localización GPS de un dispositivo en concreto.

A continuación se va a explicar cómo funciona el Front-end y cómo realiza las peticiones para obtener información desde el back-end

El Front-end es la parte del desarrollo web que consiste en la conversión de datos en una interfaz gráfica para que así el usuario pueda ver e incluso interactuar con la información.

Lo primero que ejecutará el navegador será el script html *index.html*, se trata del archivo que consiste en el esqueleto de la página web. Dentro de este tenemos 4 ficheros internos que redirigirá al usuario a diferentes páginas web dependiendo de la información que este desee consultar. Esta acción se llevará a cabo bajo una etiqueta script con código Javascript:

```
var xhr = new XMLHttpRequest();

function abrirNuevoTab1(){

    var win = window.open('./bles.html');

    win.focus();

}
```

```
function abrirNuevoTab2(){

    var win = window.open('./gps.html');

    win.focus();

}

function abrirNuevoTab3(){

    var win = window.open('./bles1.html');

    win.focus();

}

function abrirNuevoTab4(){

    var win = window.open('./gps1.html');

    win.focus();

}
```

Figura 49. Código Javascript para redirigir ficheros html

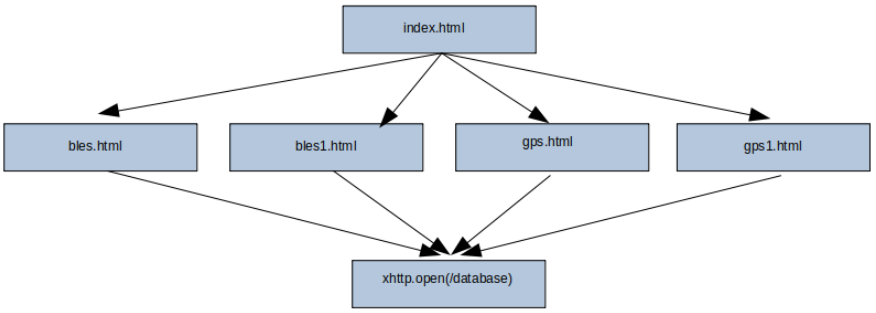

Figura 50. Diagrama flujo Front-end

Cada fichero html contendrá un script interno que se encargará de realizar las peticiones REST al Back-end para así obtener la información que cada uno requiere.

Dichas consultas se llevarán a cabo con una variable *xhttp*. Se hace uso de una interfaz llamada XMLHttpRequest para realizar las peticiones HTTP al servidor Web. Antes de mostrar ninguna información, comprobaremos que los atributos readyState y status estarán en 4 y 200 respectivamente.

Posibles estados de la interfaz XMLHttpRequest:

- readyState. **0**: Petición sin inicializar. **1**: Conexión del servidor establecida. **2**: Petición recibida. **3**: Procesando la petición. **4**: Petición terminada y respuesta lista.
- status. **500 -599**: El servidor tiene un error. **400 - 499**: Error del cliente. **300 - 399**: Redirige la página web. **200 - 299**: La respuesta del servidor es correcta. 100 - 199: Mensaje informativo.

Con el método *JSON.parse()* analizamos la cadena de texto recibida como respuesta como JSON, y pintamos los datos en la página web.

Figura 51. Imagen página inicial

En la Figura 51 se pueden observar las 4 opciones válidas para la interfaz gráfica de usuario.

- Opción 1. LOOK BLE DEVICES REAL TIME

El usuario podrá observar en tiempo real todos los dispositivos BLE que la App estará escaneando. La página se refrescará sola cada 60 segundos usando la función setTimeOut().

Figura 52. Imagen LOOK BLE DEVICES REAL TIME

- Opción 2. LOOK GPS LOCATION REAL TIME

El usuario podrá observar en tiempo real la ubicación en cada momento junto con el ID del dispositivo que está siendo localizado.

- Opción 3. LOOK BLE DEVICES

En este apartado, el usuario podrá filtrar la búsqueda de los dispositivos BLE de 2 maneras distintas: por fecha o por ID del aparato.

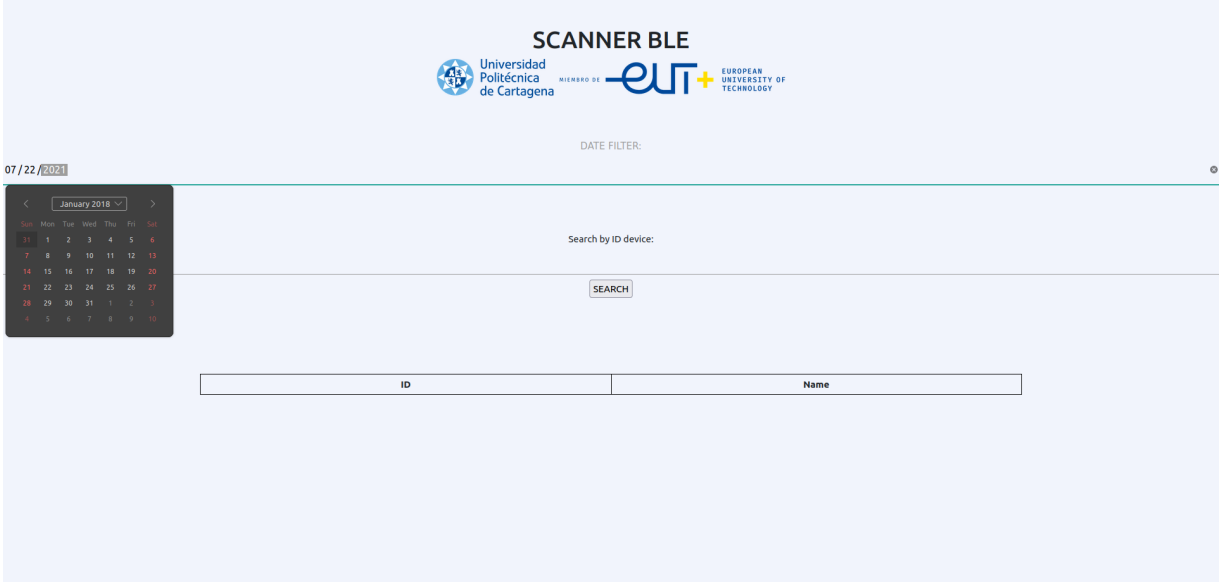

Figura 53. Imagen LOOK BLE DEVICES

- Opción 4. LOOK GPS LOCATION

En este apartado, el usuario podrá filtrar la búsqueda de la localización GPS dependiendo del dispositivo que está haciendo uso de la App gracias a una librería[15] que nos permitirá saber el ID del dispositivo. Para ello habrá una lista de dispositivos ya usados previamente para poder seleccionar el que se desee.

Figura 54. Imagen LOOK GPS LOCATION

Capítulo 5. Pruebas y resultados

5.1 Toma de datos y análisis

En una habitación de una casa familiar se ha mantenido la App android trabajando durante 1 hora aproximadamente. Tras recopilar los datos, se ha forzado el cierre de la App y se hace un análisis de estos.

El servicio de Cloud de la BBDD MongoDB pone a nuestra disposición una serie de gráficas de control de estado del Cluster gracias a RTPP (*Real-Time Performance Panel*) Monitoriza las métricas específicas de la BBDD, incluyendo fallos de la página, contadores de operaciones, colas, conexiones y el estado del conjunto de réplicas.

- *Ops counters*

Estas líneas de base de utilización para la App ayudan a llevar un recuento de las operaciones. Si este recuento se desvía de manera sospechosa podría ser un indicador de que algo ha cambiado en la aplicación o de que se está produciendo un ataque malicioso.

En nuestro caso, durante la hora de escaneo se producen picos en la gráfica. Estos se traduce en las operaciones INSERT (***command***) que introducen la información de los dispositivos BLE en la BBDD.

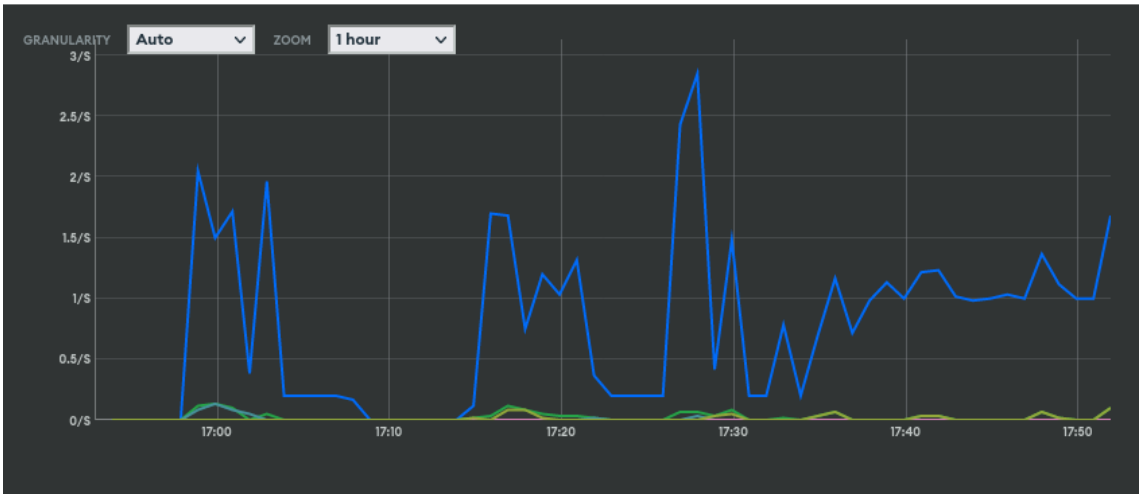

Figura 55. Gráfica Ops counters

- *Connections*

Los controladores de MongoDB implementan la agrupación de conexiones para facilitar el uso eficiente de los recursos. Cada conexión consume 1 MB de RAM, así que será importante el control de la cantidad total de conexiones para así no abrumar la RAM. Esto suele pasar cuando las aplicaciones no cierran correctamente sus conexiones.

En este caso en particular, cada vez que se hace un insert en la BBDD se abre una conexión MongoDB (***current***) pero siempre con cierre de esta al acabar la acción, llevando un control del consumo de RAM.

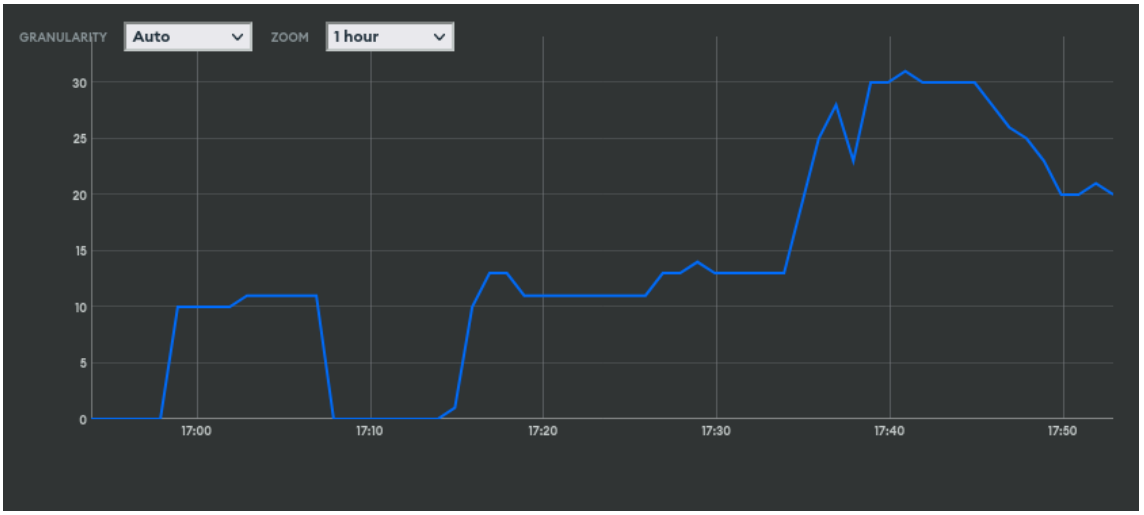

Figura 56. Gráfica Connections

- *Network*

Esta gráfica muestra la tasa media de bytes enviados al servidor de BBDD por segundo (***bytesIn***), la tasa media de bytes enviados desde este servidor de BBDD por segundo (***bytesOut***) y la tasa media de solicitudes enviadas al servidor de BBDD (***numRequests***)

La herramienta que lleva a cabo la toma de estos datos es *mongstat*.

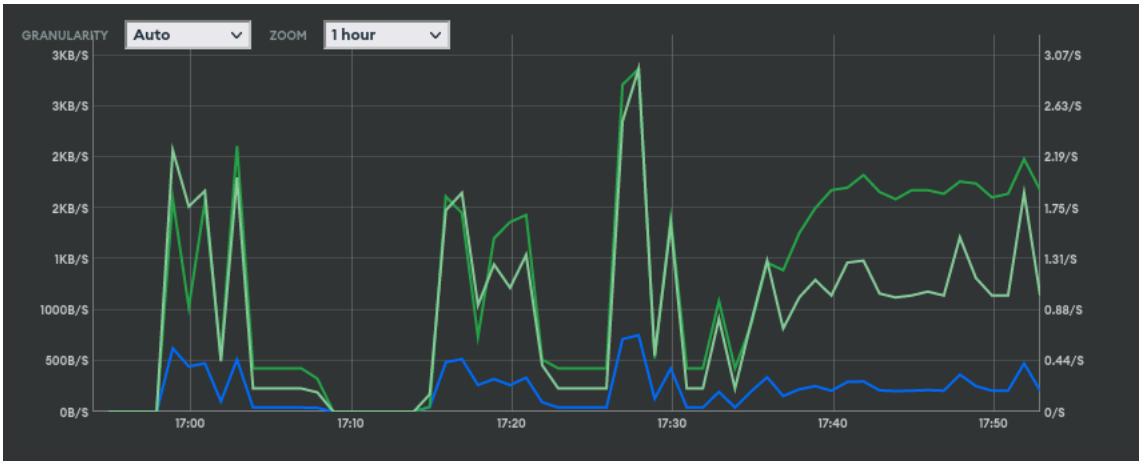

Figura 57. Gráfica Network

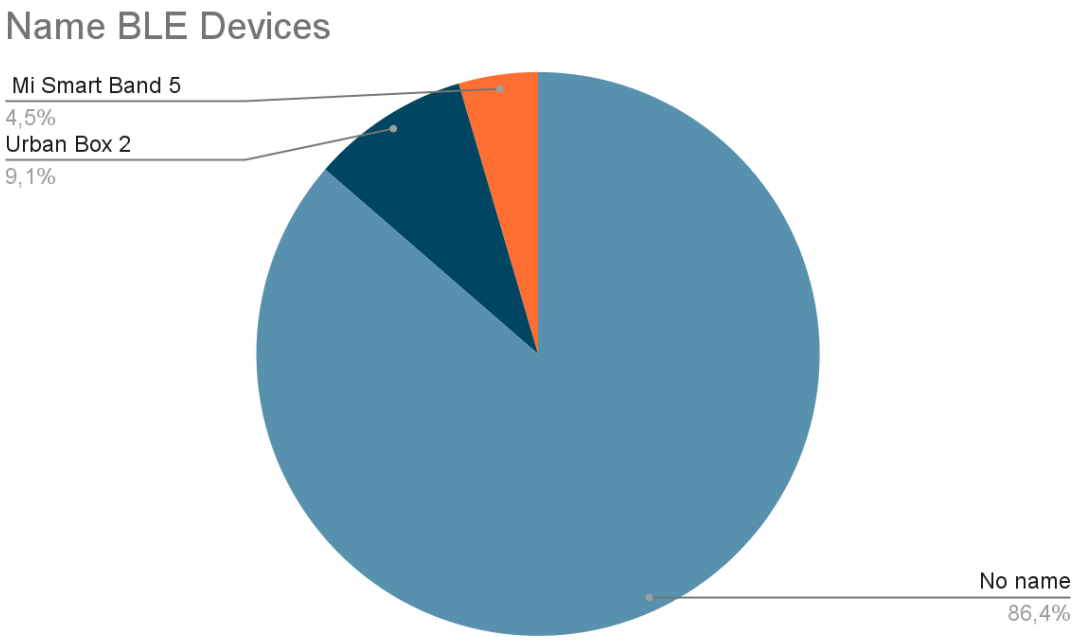

Figura 58. Gráfica Name BLE Devices

En la Figura 58 tenemos, de todos los datos recopilados, una gráfica donde vemos que la mayoría de dispositivos escaneados son sin nombre reconocido. Esto es porque los dispositivos no están obligados a anunciar su nombre según la especificación Core Bluetooth. Además, no es fácil usar el campo Name para filtrar ya que se trata de una cadena y es difícil hacerlo legible y único. Se recomienda la identificación de los beacons por su ID, teniendo en cuenta que puede ser diferente en iOS y en Android.

5.2 Setup de pruebas en escenario real

Para la realización de las pruebas en un escenario real hemos escogido una oficina de la empresa *Imprenta Grafidemar*. En la Figura 61 observamos los dispositivos BLE que se están escaneando en tiempo real, nosotros como usuario haremos una búsqueda filtrada.

Trabajador 1	Trabajador 2	Trabajador 3
EXYNOS9610	GOLDFISH	EXYNOS7904

Tabla 7. Tabla MAC trabajadores

En la Figura 59 y Figura 60 vemos la oficina que vamos a usar como escenario. Tenemos 3 habitaciones y 3 trabajadores.

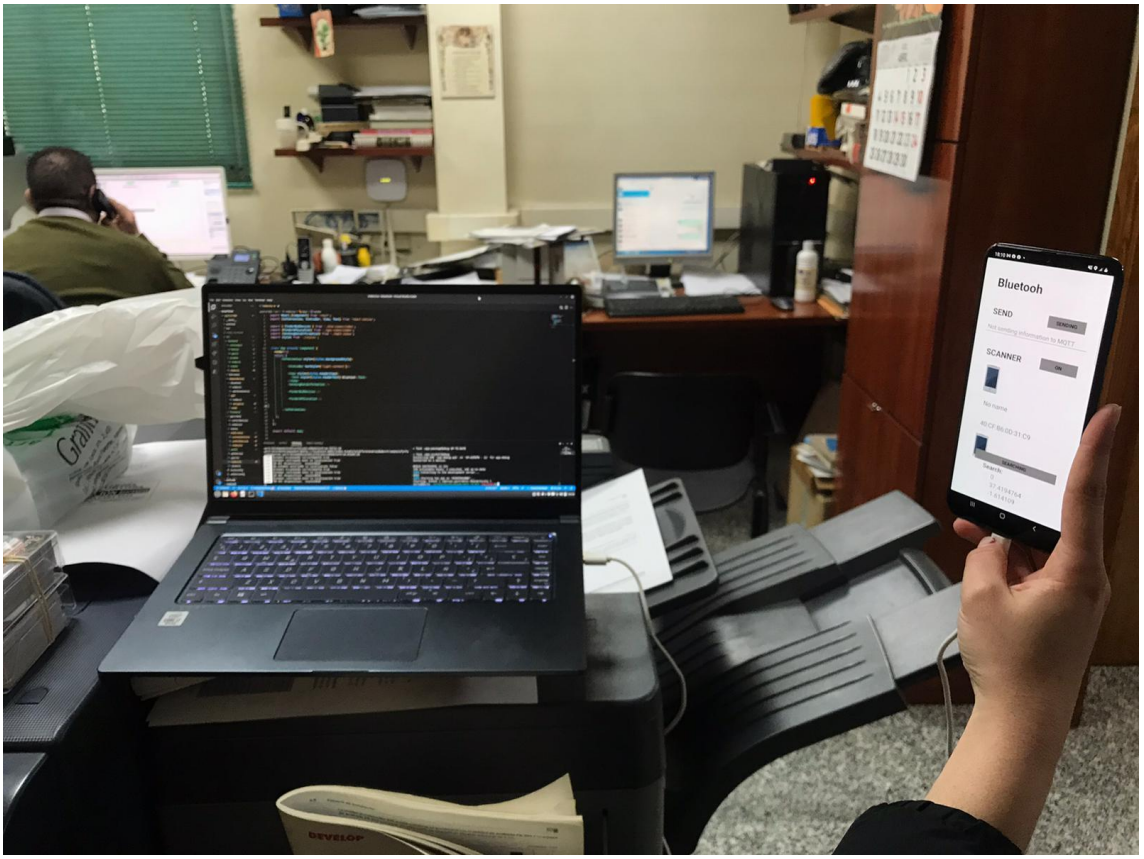

Figura 59. Escenario 1 real: oficina

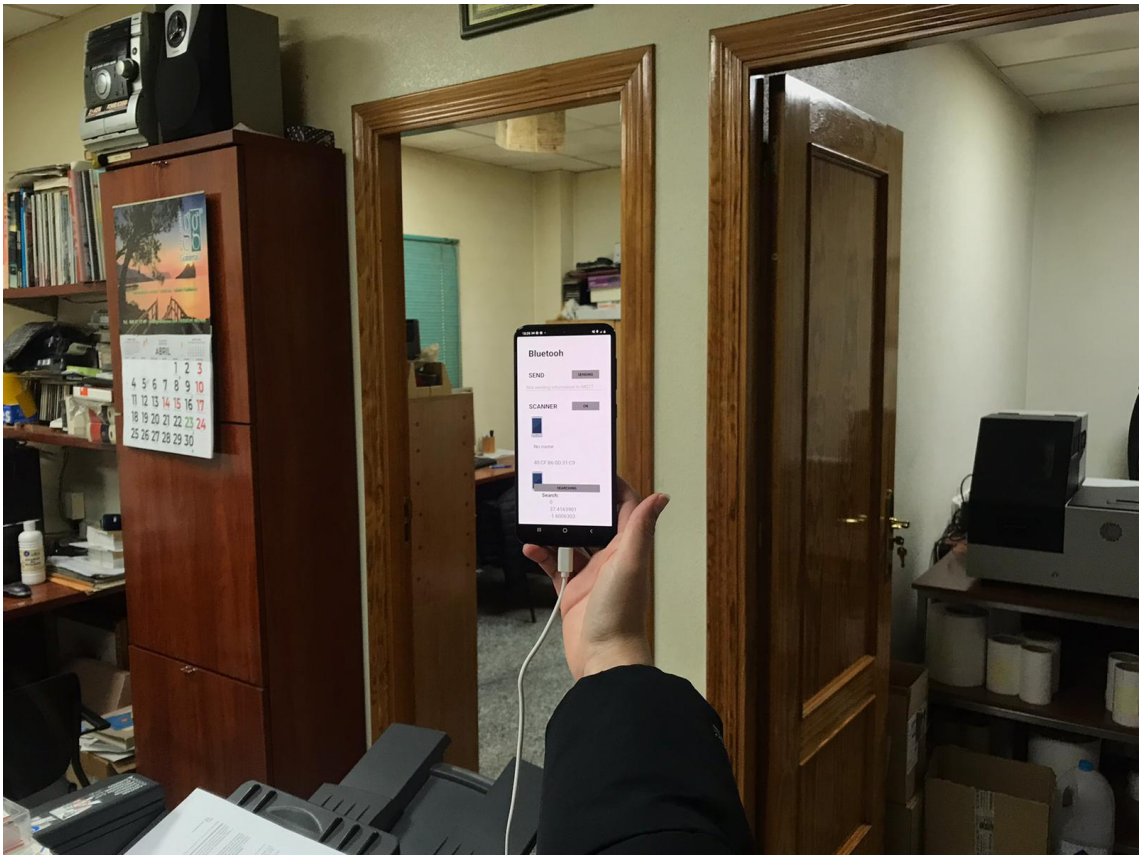

Figura 60. Escenario 2 real: oficina

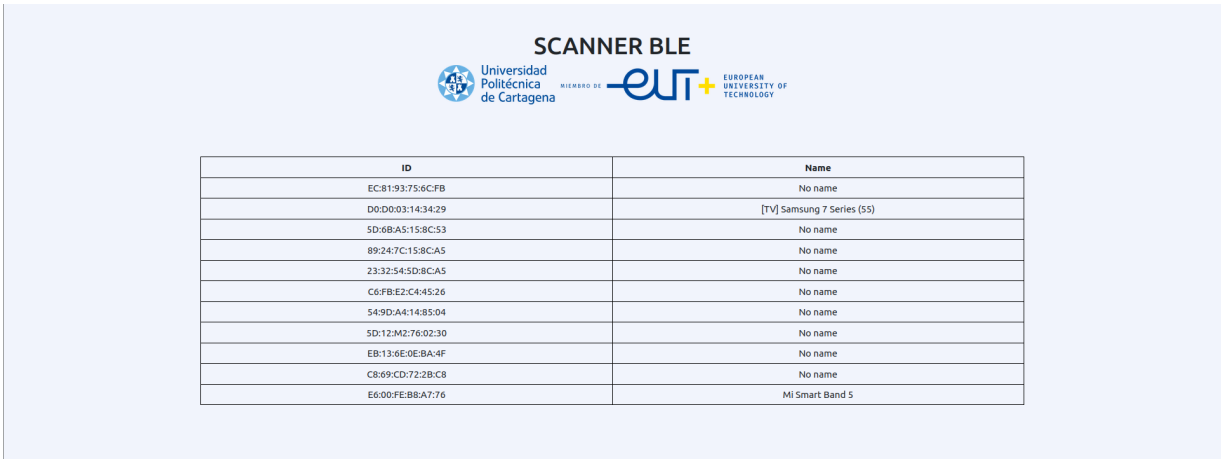

Figura 61. IU escáner tiempo real en oficina

Escogemos el dispositivo BLE con ID *5D:6B:A5:15:BC:53* para filtrar la información en la sección de *SCANNER BLE*

Figura 62. IU escáner filtrado trabajador 1

Por otro lado, en la sección *SEARCHER GPS* seleccionamos el dispositivo *EXYNOS9610* (que será el dispositivo que hará uso del APK) y vemos la localización que ha encontrado.

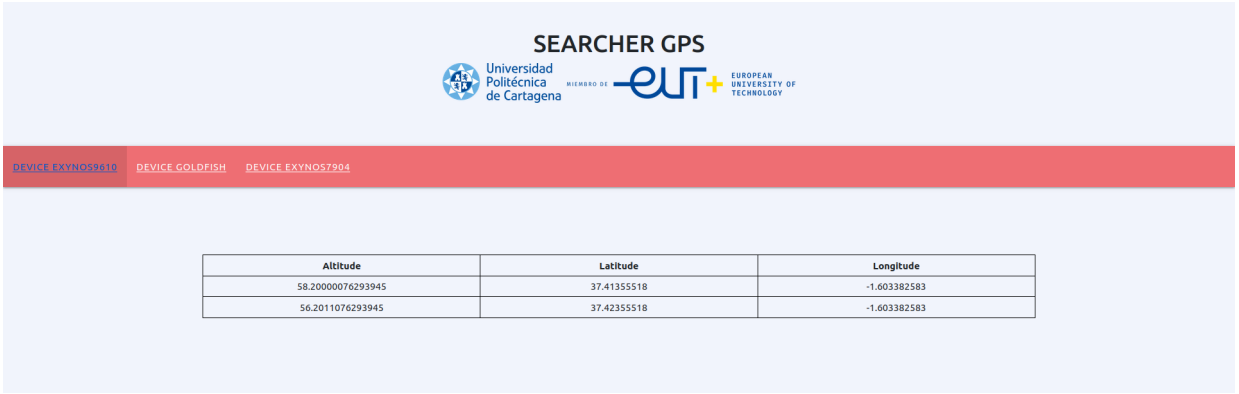

Figura 63. IU searcher filtrado por device EXYNOS9610

Seleccionamos la latitud y longitud (altitud no es necesario) y hacemos una búsqueda con Google Maps para corroborar que la localización es la correcta.

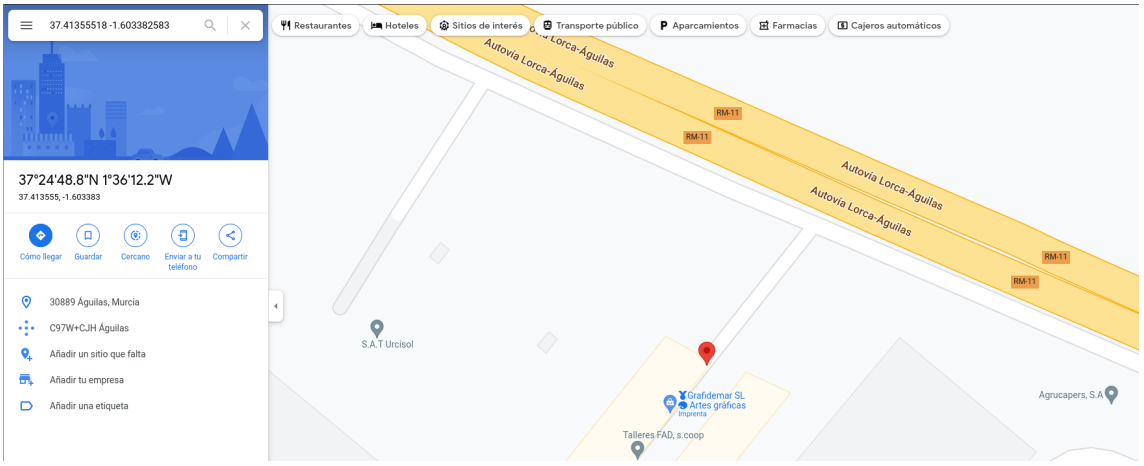

Figura 64. Comprobación searcher GPS

Además, como vemos en la Figura 65, el empleado está observando en tiempo real la información que manda al Servidor vía MQTT (porque este ha decidido que así sea)

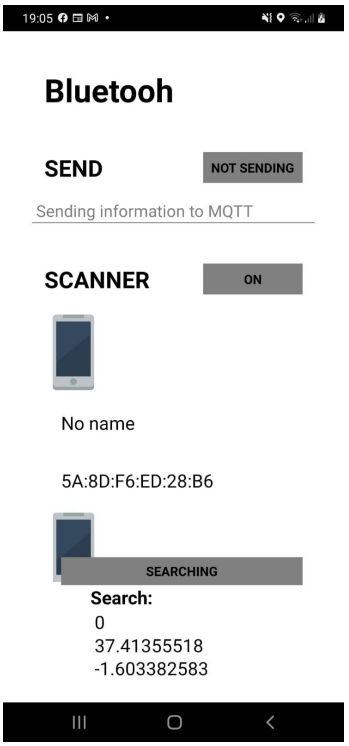

Figura 65. Captura App prueba escenario real

Capítulo 6. Conclusiones

6.1 Conclusiones

A modo de cierre de este proyecto, podemos señalar que la monitorización de datos desarrollada a lo largo del trabajo ha sido exitosa.

El resultado más relevante de esta investigación ha sido el poder llevar a ejecución la App android en una oficina real. Como se ha mostrado en las figuras Figura 52, Figura 53 y Figura 54, se ha recopilado información real y se ha podido filtrar aquella que nos ha interesado en función de tener conocimiento previo de los trabajadores y del dispositivo en uso.

Hemos podido supervisar métricas específicas en la BBDD como son: posibles fallos del desarrollo de la App, contador de operaciones, colas, conexiones y un estado del conjunto de réplicas. Además, con un control de la infraestructura subyacente asegurándonos de que siempre esté en buen estado. Todo esto gracias a herramientas como *mongodump*[16] y *mongostat*[17].

Cabe destacar además la agrupación de conocimientos de nuevas tecnologías como han sido BLE, MQTT y GPS, entre otras y la comunicación efectiva Back-end - Front-end bajo el framework NodeJS.

6.2 Objetivos logrados

Los objetivos logrados en este proyecto han sido:

- Análisis del problema y diseño de la arquitectura.

Se han realizado diagramas y esquemas introductorios de acuerdo con lo esperado del proyecto en busca de hacerlo más simple y legible.

- Familiarización en entorno NodeJS y React Native e implementación App android.

Ejemplos previos para familiarizarnos con el entorno.

- Conocimiento de tecnologías Bluetooth Low Energy, GPS y MQTT. Desarrollo de estas en conjunto.

Estudio exhaustivo anticipado de las tecnologías usadas e implementación correcta de estas.

- Monitorización de datos a través de inteligencia de geolocalización y dispositivos BLE beacons.

Recopilación de datos.

- Implementación Back-end para desarrollo y almacenamiento de datos.

MQTT Broker, BBDD y scripts que dan forma a un Back-end eficiente.

- Implementación Front-end para filtrado de datos y consulta de información por parte de responsable de empresa.

Front-end eficiente que se comunica con Back-end e interfaz de usuario.

- Pruebas de rendimiento y de comprobación del proyecto.

Monitorización y filtrado de datos exitosa.

6.3 Trabajos futuros

Como cualquier proyecto que se lleva a cabo, este tiene una serie de aspectos a mejorar. Se tratarán para trabajos futuros:

- Implementación de un servidor en Cloud, es decir, pasar de local a la nube. Levantar docker para dar Devops al proyecto. Posibles opciones: Azure o AWS. De esta manera no tendremos que realizar mantenimiento del hardware del servidor además de no depender de estar bajo la misma red.
- Relación entre BBDD de localización GPS y dispositivos BLE para relacionar de manera más visual en la IU todos los datos.
- Uso de sockets en Back-end. Evitar el uso de funciones como setTimeOut de NodeJS. Los websockets son una tecnología que permite la comunicación bidireccional entre cliente y servidor sobre un único socket TCP.
- Mejora en Back-end y Front-end para desarrollo de una App que complete gestión de presencialidad de trabajadores.

Bibliografía y referencias

[1] Real Decreto-Ley 8/2019 https://www.mites.gob.es/ficheros/ministerio/GuiaRegistroJornada.pdf

[2] BLE Bluetooth Low Energy https://developer.android.com/guide/topics/connectivity/bluetooth-le?hl=es-419

[3] GPS https://picoloro.co/gps-que-es-como-funciona-aplicaciones/

[4] Geofencing https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiAg9bWoKD3AhVP1BoKHXEHBiQQFnoECAoQAQ&url=https%3A%2F%2Fwww.useit.es%2Fblog%2Fgeofencing-que-es-y-como-funciona&usg=AOvVaw2UG1b3VzchFU9cs2Jc7vqV

[5] NodeJS https://nodejs.org/es/about/

[6] React https://es.reactjs.org/

[7] React Native https://reactnative.dev/docs/tutorial

[8] iBKS 105 https://accent-systems.com/es/producto/ibks-105/?gclid=Cj0KCQjwxtSSBhDYARIsAEn0thSCCFDFYWNXvEt0Wms1qCEf5UnAZxDDy64l-Ykp3Gc8R01uX7aVggQaAoZIEALw_wcB

[9] GPS

[10] MQTT https://mqtt.org/

[11] MongoDB Atlas https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE4HkJP

[12] Librería BLE https://www.npmjs.com/package/react-native-ble-plx

[13] Librería GPS https://www.npmjs.com/package/react-native-location

[14] Librería MQTT https://www.npmjs.com/package/sp-react-native-mqtt

[15] Librería Device Info https://www.npmjs.com/package/react-native-device-info

[16] Herramienta mongodump https://www.mongodb.com/docs/database-tools/mongodump/

[17] Herramienta mongostat https://www.mongodb.com/docs/database-tools/mongostat/

Anexos

El código del desarrollo tanto de la App, como del servidor se encuentra subido a Github:

https://github.com/digio-practicas/upct-rn-ble

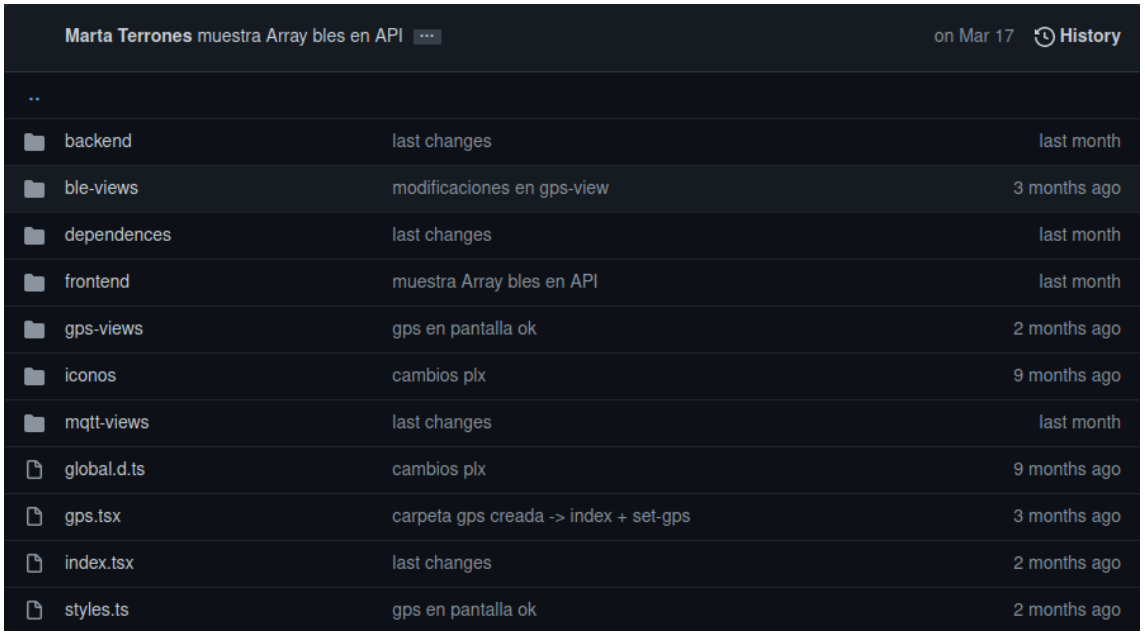

Figura 66. Captura Github