UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Mi Maps: Aplicación basada en el envío de direcciones de Google Maps

Salvador Peris Gimeno

Tutor: Francisco Martínez Zaldívar

Trabajo Fin de Máster presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la UniversitatPolitècnica de València, para la obtención del Título de Máster en Ingeniería Telecomunicación

Curso 2019

Valencia, 20 de junio de 2019

Escuela Técnica Superior de Ingeniería de Telecomunicación
UniversitatPolitècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada REG.NO. ES-CV- 000030

Resumen

En este documento se va a explicar las características fundamentales de la tecnología Bluetooth Low Energy, así como también, la realización de un proyecto basado en la creación de una aplicación iOS y Android, realizada en Swift y Java respectivamente, para el envío de direcciones desde Google Maps, en el caso de Android, o Maps, en el caso de iOS, a la pulsera de actividad Mi Band 3.

Resum

En aquest document s'explicarà les característiques fonamentals de la tecnología Bluetooth Low Energy, així com també, la realització d'un projecte basar en la creació d'una aplicación iOS i Android, realitzada amb els lleguatjes Swift i Java respectivament, per l'enviament de direccions des de Google Maps, en el cas d'Android, o Maps, en el cas d'iOS, a la polsera Mi Band 3.

Abstract

This document will explain the fundamental characteristics of Bluetooth Low Energy technology, as well as the realization of a project based on the creation of an iOS and Android application, coded in Swift and Java respectively, for sending addresses from Google Maps, in the case of Android, or Maps, in the case of iOS, to Mi Band 3 activity bracelet.

Índice

Capítulo 1. Introducción

En los últimos 10 años, las tecnologías sin cables han evolucionado exponencialmente y Bluetooth ha tenido un papel fundamental.

Esto es debido a que la sociedad se ha acostumbrado a estar en continuo movimiento además de tener todo conectado sin ningún cable que interfiera en los movimientos. Para esta demanda, la mejor tecnología que existe es Bluetooth, y en la actualidad, una versión más reciente, Bluetooth Low Energy.

Actualmente, la versión más extendida en los dispositivos es la 4.2.La versión Bluetooth 4.2 es una importante actualización de Bluetooth, ya que incorpora nuevas características y beneficios entregando una experiencia de usuario inmejorable. Las características principales de esta versión es que cuenta con opciones de conexión a Internet, a través de IPv6/6LoWPAN y tiene un consumo de energía mucho más eficiente, por este motivo se le llama Bluetooth Low Energy.

Bien sabido es que la tecnología Bluetooth Low Energy está presente en la mayoría de los dispositivos como pueden ser tablets, ordenadores o *smartphones*. Aproximadamente se venden cerca de tres billones de dispositivos cada año. BLE será la tecnología base para IoT (*Internet of Things*).

"Bluetooth was initially complementary to platforms like WiFi and ZigBee, but Bluetooth is starting to pick up features found in these other platforms. For example, one of ZigBee's advantages has been its mesh network support." [1]

Capítulo 2. Objetivo

El objetivo de este proyecto es desarrollar una aplicación para *smartphones*, en este caso, Android y iOS, donde esta se encargue de conectarse a la pulsera Mi Band 3 y después de obtener toda la información de actividades, sueños, etc. sea capaz también de recibir las instrucciones de guía de Google Maps, procesarlas y enviarlas a la pulsera para que sea consultada por el usuario de forma visual o vibratoria.

Con esto, se estudiará el funcionamiento y la estructura de la tecnología inalámbrica Bluetooth Low Energy con el objetivo de aprender a saber cómo funciona en profundidad para poder abordar este proyecto.

Figura 1. Pulsera Mi Band 3

Capítulo 3. La tecnología Bluetooth

3.1 Los orígenes de Bluetooth

En 1994, Ericsson Mobile Communications, la compañía global de telecomunicaciones con base en Suecia, comenzó a investigar sobre una interfaz de radio de baja potencia y asequible económicamente. El objetivo principal era encontrar la manera de eliminar los cables entre teléfonos móviles y sus periféricos. La compañía determinó que esta tecnología se desarrollaría para crear enlaces de radio de corto alcance.

El trabajo de Ericsson atrajo la atención de IBM, Intel, Nokia y Toshiba, en ese momento, los gigantes tecnológicos. Estas compañías formaron el conocido SIG Bluetooth en 1998, grupo que creció de manera exponencial hasta las más de 2000 que hay en la actualidad. Estas compañías crearon conjuntamente la especificación Bluetooth 1.0, que vio la luz en 1999. La especificación consistía en dos documentos: el núcleo fundamental, donde se especificaba la radio, la banda base, el gestor de enlace, el protocolo de descubrimiento de servicios, el nivel de transporte y la interoperabilidad con diferentes protocolos; y el perfil, donde se especificaba los protocolos y los procedimientos requeridos para los distintos tipos de aplicaciones Bluetooth.

3.2 ¿De dónde viene el nombre?

Los ingenieros de Ericsson denominaron Bluetooth a la nueva tecnología en honor a un rey vikingo danés del siglo X. Harald Bluetooth reinó desde 940 a 985 y se le atribuye la unificación de ese país y la adopción del cristianismo.

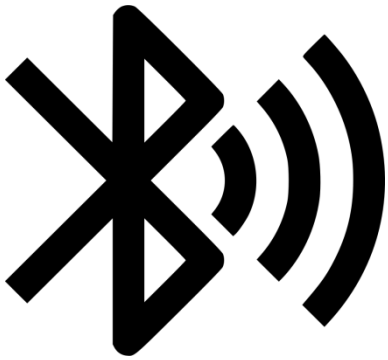


Figura 2.Símbolo Bluetooth

En esa época, los daneses vivían en comunidades bajo el mandato de jefes locales, algunos de estos jefes aterrorizaron las ciudades costeras de Europa con sus incursiones piratas vikingas para conseguir esclavos y botín. Durante mucho tiempo, los daneses habían venerado a los dioses Thor y Odin.

La historia dice que Harald era el hijo del rey Gorm el Viejo de Dinamarca y de Thyra, que se decía que era la hija de un noble inglés. Cuando llevaba 25 años de reinado, el sacerdote Poppos impresionó a Harald sujetando una pieza

de metal al rojo vivo sin producirse ninguna herida. Poppos le explicó que esto pasaba por su fe en Dios y con esto, convenció a Harald de los poderes del cristianismo. La aceptación del cristianismo por el rey Harald y su subsiguiente bautismo hizo mucho para aliviar las luchas religiosas en Dinamarca.

Los objetivos de la tecnología inalámbrica Bluetooth son también la unificación y la armonía: el permitir a diferentes dispositivos que se comuniquen a través de un estándar aceptado por todos.

Capítulo 4. Estado del arte

4.1 ¿Qué es Bluetooth Low Energy?

Bluetooth Low Energy y también denominado *Smart Bluetooth*, (desde ahora en adelante BLE) es la característica principal de la versión 4.0 del núcleo de especificaciones Bluetooth. Fue introducida a principios de junio de 2010 por SIG, así llamado al Grupo Especial de Bluetooth. Es una continuación del Bluetooth clásico, pero con finalidades un poco distintas. BLE es una tecnología inalámbrica diseñada para tener un menor consumo de energía que su antecesor, para que también tuviera un bajo coste, baja potencia y fácil de implementar.

BLE facilita con creces las comunicaciones en distancias cortas entre dispositivos que no requieren grandes transferencias de datos, implementar una tecnología que sea eficiente para el monitoreo y el control de aplicaciones donde la cantidad de datos a transferir suelen ser bajas.

Las configuraciones clásicas Bluetooth (BR/EDR) y BLE no son compatibles. La manera de llegar a entenderse entre los dos estándares es mediante un módulo doble. Entonces, aquí se encuentran los tres tipos de configuraciones posibles:

- **Bluetooth clásico (BR/EDR):** el clásico Bluetooth.
- **Monomodo (BLE):** implementa BLE y se puede comunicar con otros módulos monomodo y con los de modo dual.
- **Modo dual (BR/EDR/LE):** implementa ambos estándares y es capaz de conectarse con cualquier dispositivo de esta tecnología.

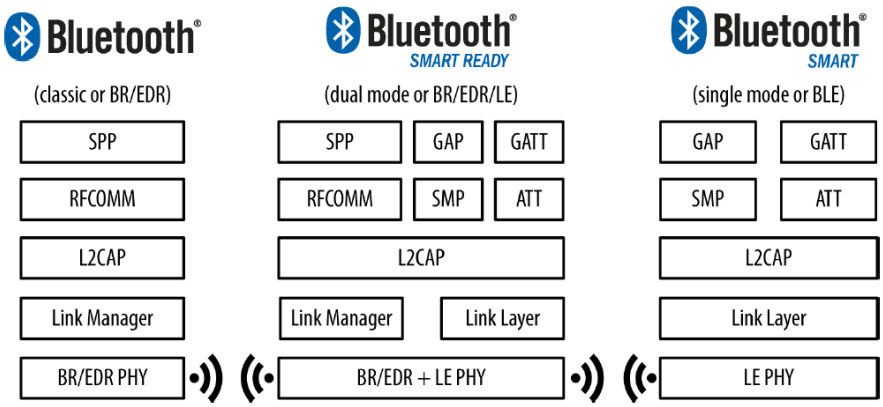


Figura 3. Configuraciones entre Bluetooth

En la figura 3 se puede observar todas las configuraciones posibles de los dispositivos Bluetooth. Los dispositivos, como la pulsera de actividad Mi Band 3, llevan integrado la nueva versión de Bluetooth, BLE.

Desde la primera versión de BLE, siendo ésta la 4.0, se han añadido la 4.1 y la 4.2. Todas estas versiones introducen mejoras y nuevas funcionalidades. Las diferencias más importantes entre las versiones 4.1 y 4.2 son las siguientes:

- Los dispositivos con Bluetooth 4.2 serán compatibles con dispositivos con versiones 4.0 y 4.1.
- Las características de Bluetooth 4.1 son compatibles con dispositivos Bluetooth 4.2.
- En la versión 4.1 se introdujeron canales en la capa L2CAP para la conexión a Internet, así pues, con los nuevos servicios y perfiles de la versión 4.2, facilita a los desarrolladores la conexión a Internet.
- La versión 4.2 extiende la longitud de datos de los 27 bytes a los 251 bytes.
- Esta es la característica más importante que cabe destacar. En la versión 4.2 se añade un protocolo de seguridad llamado ECDH (*Elliptic Curve Diffie-Hellman*) que permite que dos dispositivos se intercambien información de forma totalmente segura.

Por otra parte, la pila de protocolos de BLE se divide en tres partes bien diferenciadas:

- **Aplicación:** Aplicación de usuario
- **Host:** Capas superiores de la pila de protocolos.
- **Controlador:** Capas inferiores de la pila de protocolos.

"El host y el controlador interactúan entre ellos mediante la interfaz de control de host (HCI), así pues, pueden operar host y controladores de diferentes compañías." [2]

Con esto, se permite distintas configuraciones de hardware. Las distintas capas se pueden incluir en un único chip o circuito integrado, o directamente separados en varios IC (*Integrated Circuit*).

4.2 BLE, la tecnología futura

Los grandes vendedores tecnológicos están invirtiendo todos sus esfuerzos en facilitar el trabajo a los desarrolladores de aplicaciones móviles dando la libertad de usar el marco BLE en todo lo necesario.

En comparación con otras tecnologías inalámbricas, debemos tener en cuenta varios parámetros importantes como el rango de cobertura, el consumo de energía, así como las tasas de transmisión de datos.

BLE es una tecnología WPAN (*Wireless Personal Area Network*), es decir, un área inalámbrica cercana. Su cobertura típica es de 10 metros aproximadamente. Una red de área personal se suele conseguir siempre cuando existe un teléfono móvil o Smartphone conectado mediante Bluetooth a un dispositivo como puede ser una pulsera de actividad física. Cabe destacar que estos dispositivos tienen una potencia de transmisión baja, ya que son diseñados para un rango de cobertura bajo, así pues, las baterías que suelen llevar pueden ser de tamaño muy pequeño.

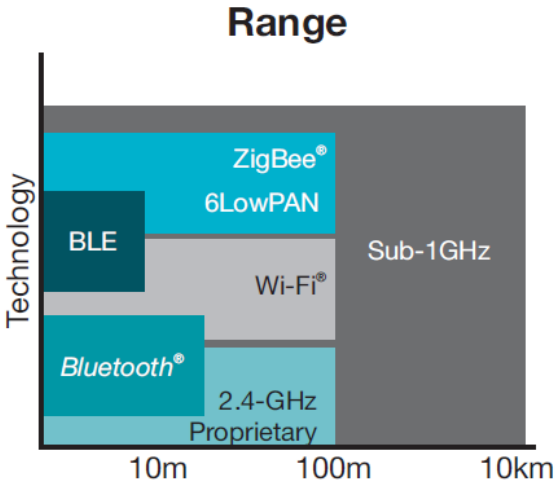


Figura 4. Comparación de cobertura

En la figura anterior, se puede observar como para las cortas distancias se encuentra tanto Bluetooth clásico como el BLE. Incluso el BLE todavía está diseñado para distancias más cortas que el clásico, ya que, como su nombre indica, pretende ser la tecnología que menos consumo de energía tenga.

Las WLAN, en cambio, tienen un rango de cobertura de 100 metros y la tecnología más conocida que trabaja en estos rangos es el Wi-Fi, aunque existen otras como ZigBee que también son conocidas en el ámbito industrial.

Las topologías más conocidas en estas tecnologías son estrella y P2P. En estrella, todos los nodos están conectados a un nodo central, que normalmente se utiliza como puerta de enlace o Gateway a Internet. Un ejemplo muy conocido de esta topología son las redes Wi-Fi, donde el punto de acceso es el encargado de ser la puerta de enlace a Internet a todos los nodos conectados.

En este proyecto, la topología de red que se utiliza es P2P, donde existe un nodo maestro o servidor y un esclavo que actúa de cliente. En este caso, la pulsera Mi Band 3 es la encargada de dar a conocer los servicios que tiene disponibles y por otra parte, el teléfono móvil o *smartphone*, encargado de acceder a las características de dichos servicios.

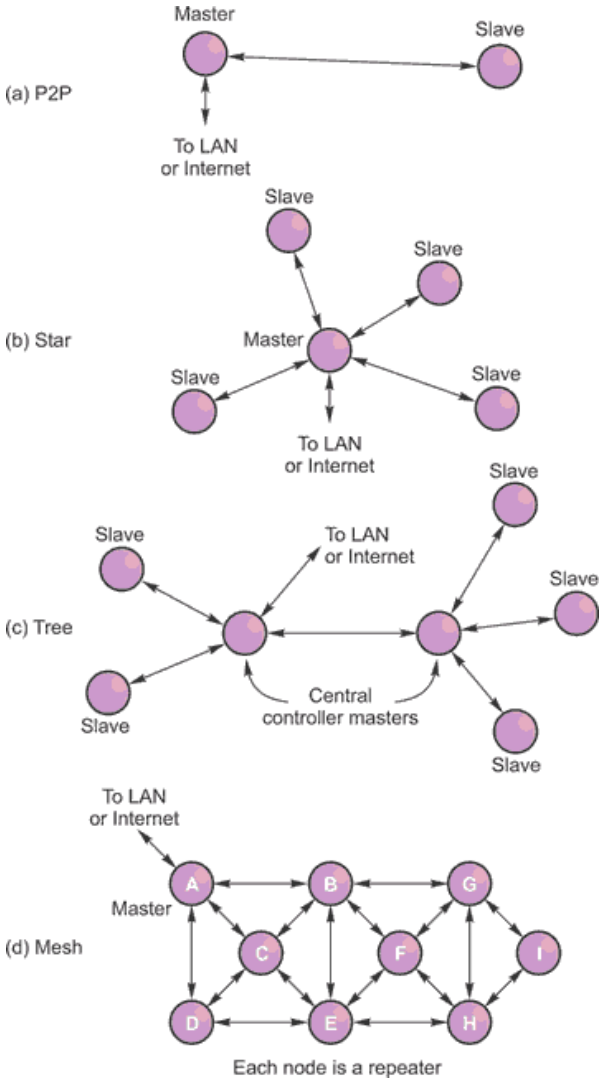


Figura 5. Tipos de tipologías

En la figura anterior se pueden observar distintas topologías. En este proyecto, la topología (a) P2P es la que se utiliza.

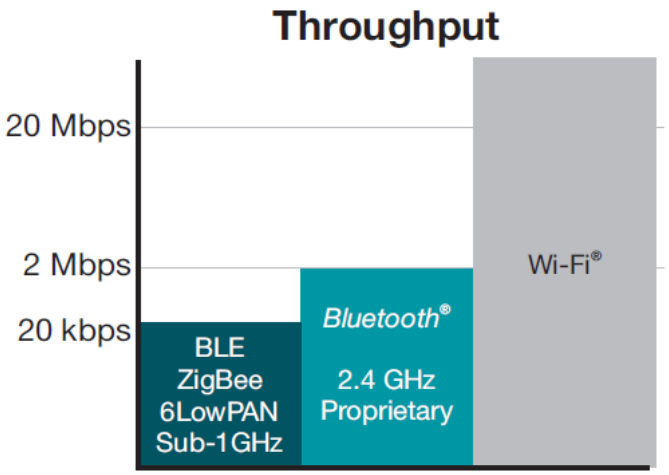


Figura 6. Anchos de banda

En cuanto a la transmisión de datos se refiere, BLE está en la sección de mínimo ancho de banda, esto es porque BLE fue diseñado para transmitir poca cantidad de datos. En el caso de BLE, se puede observar en la figura anterior, 20 kbps es más que suficiente, mientras que Bluetooth tiene un ancho de banda de 2 Mbps, demasiado para la mayoría de los accesorios BLE que se utilizan hoy en día.

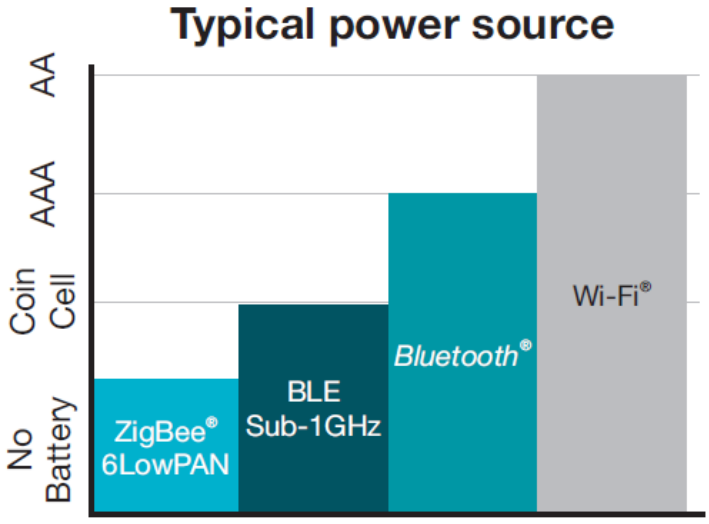


Figura 7. Tipo de batería utilizada

Independientemente de las baterías que se utilicen, en la figura anterior queda claro la diferencia de consumo energético entre la tecnología Bluetooth clásica y BLE.

4.3 Arquitectura Bluetooth Low Energy

Como anteriormente se ha comentado, la pila de protocolos se divide en tres partes bien diferenciadas: el controlador, el host y la aplicación. Además, la pila de protocolos BLE está formada por diferentes capas y éstas interactúan entre sí.

4.3.1 Estructura

Vamos a resumir la estructura de BLE:

- **Aplicación**
  - Esta es la capa superior y es la que se encarga de manipular los datos.

- **Host**
  - **GAP:** Perfil genérico de acceso
  - **GATT:** Perfil genérico de atributo
  - **ATT:** Protocolo de atributo
  - **SMP:** Protocolo de gestión de seguridad
  - **L2CAP:** Protocolo de control lógico y adaptación de enlace
  - **HCI:** Interfaz de control del host, parte del host

- **Controlador**
  - **HCI:** Interfaz de control del host, parte del controlador
  - **LL:** Capa de enlace
  - **PHY:** Capa física

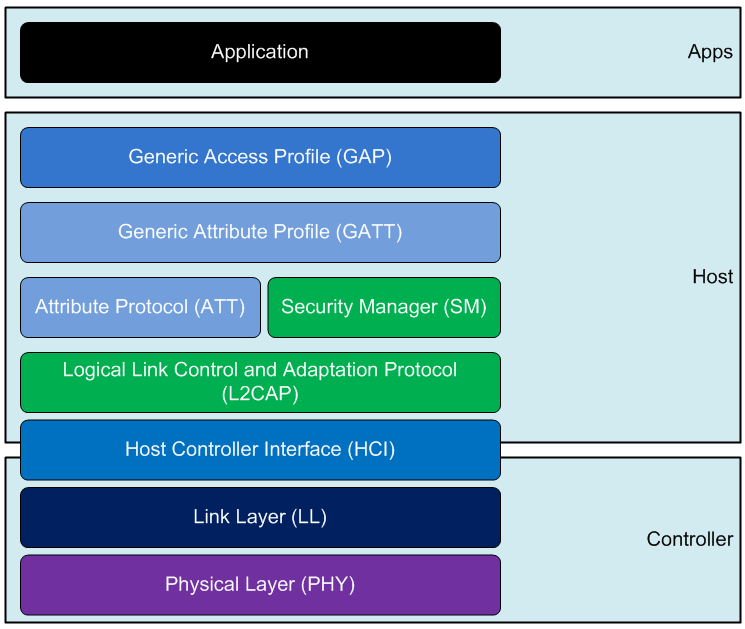


Figura 8. Capa de protocolos

4.3.2 Controlador

4.3.2.1 La capa física (PHY)

Expansión de espectro con saltos de frecuencia

La expansión de espectro es una técnica de codificación digital en la que se toma una señal de banda estrecha y se expande sobre un espectro de frecuencias. La operación de codificación hace que el número de bits aumente artificiosamente y expanda el ancho de banda utilizado. Utilizando el mismo código de expansión que el transmisor, el receptor correlaciona y devuelve la señal expandida a su forma original. El resultado de todo esto es una tecnología de transmisión de datos muy robusta y ofrece considerables ventajas respecto a sistemas de radio convencionales de banda estrecha.

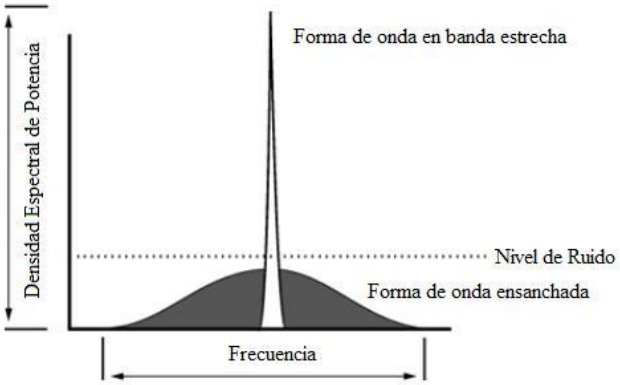


Figura 9. Expansión de espectro

Una de las ventajas de la expansión es que la señal expandida tiene una densidad espectral de potencia mucho menor. Esta baja densidad espectral de potencia proporciona resistencia a problemas creados por otros sistemas de radio, incluyendo:

Por una parte, interferencias. Condición en la que se interrumpe una transmisión a causa de fuentes externas, como el ruido emitido por varios dispositivos electrónicos.

Por otra, la interferencia intencionada. Condición en la que una señal más fuerte se sobrepone a una señal más débil.

Por otra, trayecto múltiple. Condición en la que la señal original se distorsiona tras reflejarse en un objeto sólido.

Y por la última parte, intercepción. Condición en la que usuarios no autorizados capturan señales intentando averiguar su contenido.

Los sistemas de radio convencionales de banda estrecha transmiten y reciben en un ancho de banda alrededor de cierta portadora, que es sólo lo suficientemente amplio como para transmitir la información.

La expansión de espectro tiene dos modos de funcionamiento:

- Los *saltos de frecuencias* expanden la señal haciendo saltar la señal de banda estrecha sobre toda la banda de radio en función del tiempo.

- La *secuencia directa* expande toda la señal a la vez sobre la banda de radio al completo.

Aunque la tecnología Bluetooth utiliza el modo de expansión de espectro mediante saltos de frecuencia, es interesante contrastarlo con el mecanismo de secuencia directa para observar sus ventajas.

La tecnología Bluetooth utiliza la técnica de saltos de frecuencia, esto significa que, el transmisor da saltos de frecuencia a la señal con una velocidad específica y siguiendo una secuencia que conoce tanto el emisor como el receptor. Las frecuencias seleccionadas se adquieren de un grupo de frecuencias predeterminado. Entonces, como sólo el receptor conoce cuál es la secuencia exacta de los saltos de frecuencia, él solo entenderá los datos transmitidos.

Las autoridades de EE. UU. obligan que los sistemas con expansión de espectro que utilicen los saltos de frecuencia no permanezcan más de 0,4 segundos en cualquier canal en la banda de 2,4 GHz. Entonces, en la banda de 900 MHz deben saltar entre 50 canales y en la banda de 2,4 GHz deben saltar entre 75 canales. Con esta normativa, se permite reducir las colisiones de paquetes en lugares donde hay múltiples transmisores.

"El estándar Bluetooth determina una velocidad de 1.600 saltos por segundo entre 79 frecuencias." [2]

El principal problema que tiene la tecnología Bluetooth es que existen en la banda de 2,4 GHz otras tecnologías, entonces, ya han creado directrices para que las interferencias que se produzcan sean las mínimas.

GFSK

"La modulación por desplazamiento de frecuencia gausiana (Gaussian Frequency Shift Keying) es un tipo de modulación donde un 1 lógico es representado mediante una desviación positiva de la frecuencia de la onda portadora y, en cambio, un 0 mediante una desviación negativa de la misma.

*GFSK es una versión mejorada de la modulación por desplazamiento de frecuencia (**FSK**). En **GFSK**, la información*
es pasada por un filtro gausiano antes de modular la señal. Esto se traduce en un espectro de e
nergía más estrechode la señal modulada, lo cual permite mayores velocidades de transferencia sobre un mismo canal." [9]

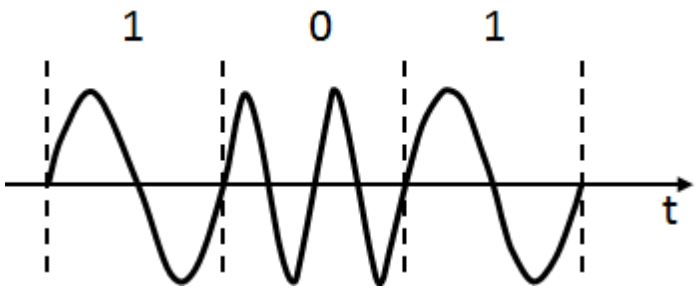


Figura 10. Ejemplo de GFSK

En la figura anterior se puede observar que depende el cambio de frecuencia que realice la señal, en el receptor se interpretará como un "0" o como un "1".

4.3.2.2 Funcionamiento de la capa física

La capa física está formada por los elementos que hacen posible la comunicación, por ejemplo, la modulación y demodulación de señal, la transformación de señal analógica a señal digital y viceversa.

Esta capa funciona en la banda ISM (*Industrial, Scientific and Medical*) desde los 2.4 GHz hasta los 2.4835 GHz y se definen 40 canales de 2 MHz de ancho de banda cada uno. Existen dos tipos de canales: unos canales se dedican a los anuncios y otros a los datos. Los canales de anuncios se encargan de descubrir dispositivos, emitir mensajes de *broadcast*, etc.

Los dispositivos que utilizan la tecnología BLE y están en modo de anuncio, transmiten en tres canales que suelen ser 37, 38 y 39. Estos canales no han sido elegidos al azar, sino que son los tres que minimizan las interferencias con los canales Wi-Fi que más se utilizan, estos son los canales 1, 6 y 11. [4]

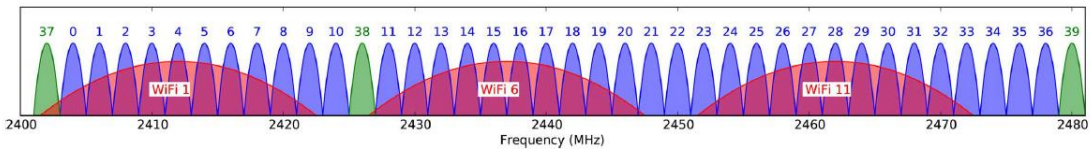


Figura 11. Espectro de frecuencias

Los dispositivos BLE activan la radio de 0.6 a 1.2 ms para buscar otros dispositivos utilizando los canales de anuncio 37, 38 y 39. [4]

En cambio, los canales de datos se utilizan en la comunicación bidireccional y cuando los dispositivos están conectados. Como la banda de frecuencias en la que trabajan los dispositivos BLE sufre muchas interferencias se utiliza la técnica explicada anteriormente, expansión de espectro con saltos de frecuencia. Esta técnica divide la banda de frecuencia en varios canales y, durante la comunicación, va cambiando de un canal a otro. Para ello, se utiliza la siguiente fórmula:

"canal = (canal actual + salto) · mod 37" [4]

Cuando se realiza la conexión, el valor del salto se comunica y cada vez que se crea una nueva conexión, el valor del salto cambia. Con esta técnica es muy poco probable que dos dispositivos transmitan en la misma frecuencia. [4]

Los canales utilizan la modulación expuesta anteriormente llamada GFSK (*Gaussian Frecuency Shift Keying*).

Como también hemos visto anteriormente, el máximo rango de cobertura de BLE está en torno a los 100 m y la tasa de modulación es de 1 Mbps.

4.3.3 La capa de enlace (LL)

Esta capa es la que interactúa con la capa física. Ésta es la encargada de que los tiempos que define la especificación Bluetooth se cumplan, también es la que más costo computacional ocasiona.

La capa de enlace gestiona cómo se establecen las conexiones entre los dispositivos. Un dispositivo BLE puede ser maestro, esclavo o ambos. BLE utiliza más recursos del dispositivo si éste es maestro.

Se definen 4 tipos de rol en la capa de enlace:

- **Anunciante:** el dispositivo está en modo de anuncio
- **Escaneador o escáner:** el dispositivo está en modo de escaneo
- **Maestro:** es el dispositivo que inicia la conexión y más tarde es el encargado que la gestiona.
- **Esclavo:** es el dispositivo que acepta las peticiones de conexión.

Para conocer mejor cómo funciona la capa de enlace, vamos a conocer cinco estados en los que puede estar un dispositivo BLE:

- **Estado de espera o Standby:** el dispositivo no transmite o recibe paquetes. Desde cualquier estado se puede acceder a este estado.

- **Estado de anuncio o Advertising:** el dispositivo está transmitiendo paquetes de anuncio y escucha y contesta a los mensajes de petición y respuesta de escaneo. Desde el estado de espera se puede acceder a este estado.

- **Estado de escaneo o Scanning:** el dispositivo se encuentra escuchando a los paquetes de anuncio en los canales de anuncio. Sería un dispositivo "escaneador", llamado así por estar en el "estado de escaneo". Desde el estado de espera se puede acceder a este estado.

- **Estado de inicio o Initiating:** el dispositivo escucha paquetes de anuncio y responde para iniciar una conexión con otro dispositivo. Se puede acceder a este estado desde el estado de espera.

- **Estado de conexión o Connection:** el dispositivo puede ser maestro si se accede desde el estado de inicio o esclavo si el dispositivo accede desde el estado de anunciado.

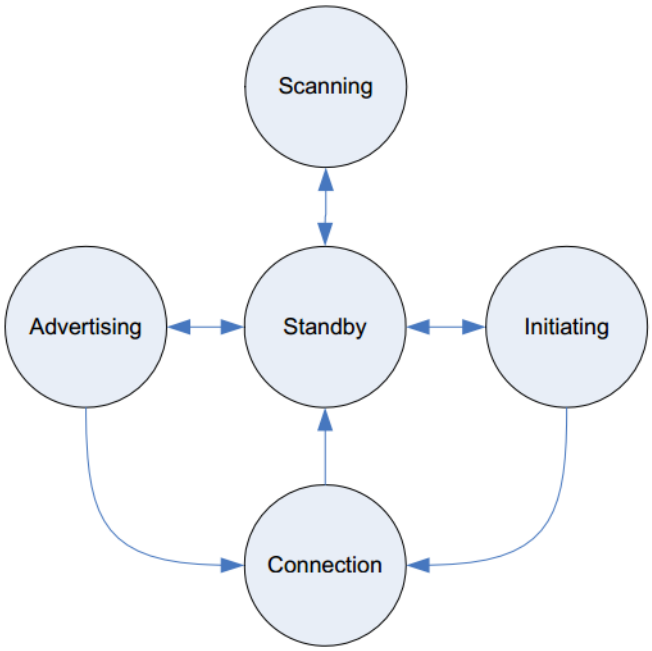
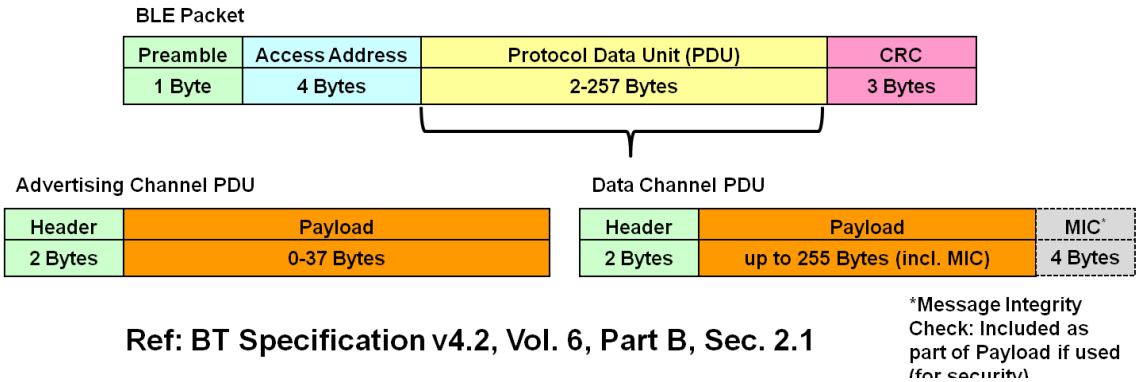


Figura 12. Estados BLE

En los procesos de conexión existen dos direcciones diferentes. Cada transmisión en un canal físico se establece al inicio con una dirección de acceso que se utiliza como un código de correlación por parte de los dispositivos conectados al canal físico. Este código está al principio de cada paquete que se transmite. Tiene un tamaño de 4 bytes y en el caso de que sea un paquete de anuncio, la dirección será fija y tendrá el valor 0x8E89BED6. Para conexiones en canales de datos, la dirección debe ser diferente cada vez y será un valor aleatorio de 5 bytes y esta dirección de acceso es enviada en el paquete de la petición de conexión: CONNECT_REQ.



La segunda dirección que se utiliza es aquella que identifica al dispositivo Bluetooth en cuestión. Estas direcciones tienen un tamaño de 6 bytes y se encuentran en la PDU después de los 2 bytes de cabecera. Cabe destacar que existen dos tipos de direcciones: una fija y otra dinámica. La fija es la que le suministra el fabricante y será la misma para toda la vida útil del dispositivo, y la otra es la aleatoria, que puede ser reprogramada o generada dinámicamente durante el tiempo de ejecución.

En la figura anterior se puede observar el paquete BLE, entonces, en la capa de enlace se comprueba toda la información recibida en un paquete BLE con su CRC y demanda la retransmisión cuando la comprobación de errores detecta un error. Cabe destacar que no hay límite de retransmisión, es decir, hasta que la retransmisión no sea satisfactoria, ésta no cesará. [4]

También es fácil observar que sólo existe un paquete BLE y dos tipos de PDU: uno dedicado a los anuncios y otro a los datos.

4.3.3.1 Paquetes de anuncio

Los paquetes de anuncio tienen dos tareas fundamentales que realizar. La primera tarea es transmitir paquetes para aplicaciones que no requieren ninguna conexión y la segunda es descubrir esclavos para, más tarde, conectarse a ellos. Cuando estos paquetes se envían, no se conoce si hay algún dispositivo escaneando.

La carga útil de los paquetes de anuncio suele ser de 31 bytes. A continuación se va a mostrar en la figura la PDU de un paquete de anuncio:

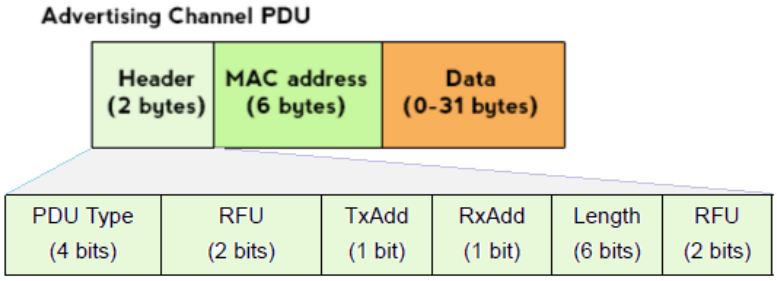


Figura 13. Cabecera de PDU de anuncio

El campo *PDU type* puede contener distintos valores, entre ellos destacamos:

- 0011: SCAN_REQ
- 0100: SCAN_RSP
- 0101: CONNECT_REQ

El campo TXADD indica si el campo *MAC address* contiene una dirección pública, identificado con 0, o aleatoria, identificado con un 1. Por otra parte, el campo RXADD indica si la dirección del iniciador del campo data es pública, identificado con 0, o es aleatoria, identificado con 1.

En el interior del tipo de paquete de anuncio existen diferentes PDU y con diferentes propósitos. Antes de continuar, vamos a definir tres características de los paquetes de anuncio.

- **Conectabilidad:** puede ser conectable, cuando un escaneador puede establecer una conexión al recibir un paquete de anuncio y no conectable, cuando un escaneador no puede establecer conexión al recibir un paquete de anuncio.

- **Escaneabilidad:** puede ser escaneable, cuando un escaneador puede emitir una petición de escaneado, o no escaneable, no puede emitirlo.

- **Directividad:** si es directivo, un paquete de esta naturaleza contiene en su carga útil solamente la dirección del dispositivo anunciador y del dispositivo escaneador objetivo, de otra forma, ya no está dirigido a ningún dispositivo.

4.3.3.2 PDU de anuncio

Las PDU de anuncio se utiliza para anunciarse y esto ocurre en la capa de enlace. El campo *AdvA* contiene la dirección pública o aleatoria, según corresponda, y el campo *InitA*, contiene la dirección del iniciador en el caso que sea pública o aleatoria.

Existen diferentes variables que tienen su significado, entre ellas, las más destacables son las siguientes:

- ADV_IND: el dispositivo tendrá las propiedades de ser conectable, escaneable y no directivo.
- ADV_DIRECT_IND: conectable, no escaneable y directivo.
- ADV_NONCONN_IND: no conectable, no escaneable y no directivo.
- ADV:SCAN_IND: no conectable, escaneable y no directivo.

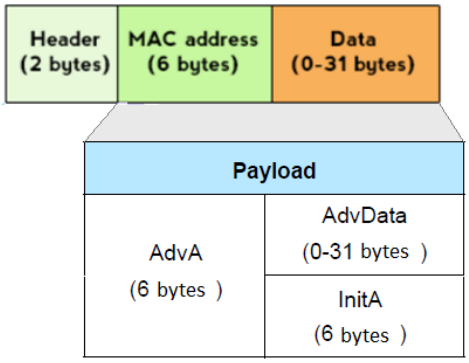


Figura 14. PDU de anuncio

4.3.3.3 PDU de escaneo

La estructura de las PDU de escaneo es la mostrada en la siguiente figura:

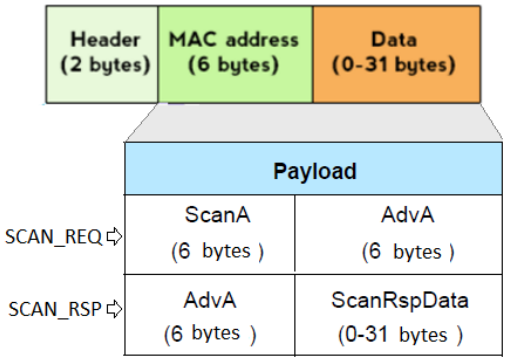


Figura 15. PDU de escaneo

Las PDU de escaneo tienen algunas variables que también son dignas de ser conocidas:

- SCAN_REQ: se envía desde la capa de enlace del dispositivo escaneador y envía esta solicitud porque ha recibido un paquete que en su PDU está la propiedad escaneable. Esta solicitud la recibe el dispositivo que envió el paquete de anuncio escaneable.

- SCAN_RSP: el dispositivo anunciador enviará este paquete como respuesta a una solicitud SCAN_REQ. Todo lo relativo al host se encontrará en el campo *ScanRspData*.

4.3.3.4 PDU de inicio de conexión

Las PDU de inicio de conexión son enviadas por el dispositivo que está en el estado de inicio y recibido por el que está en estado de anuncio. La estructura de las PDU de inicio de conexión es la siguiente:

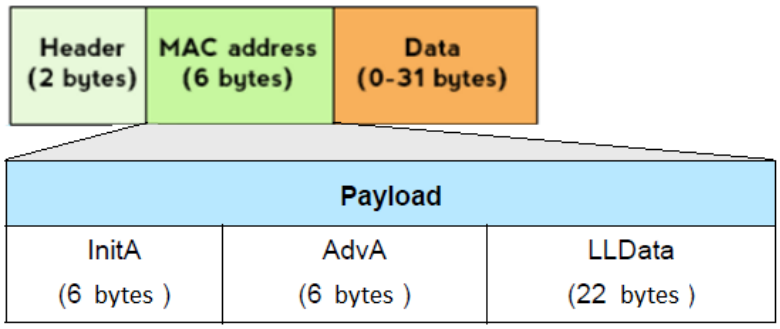


Figura 16. PDU de inicio de conexión

Como esta PDU lo que pretende es iniciar la conexión, la variable más destacable es la siguiente:

- CONNECT_REQ: la carga útil que contiene está formada por *InitA*, *AdvA*, explicadas anteriormente, y un campo llamado *LLData* (*Link Layer Data*), donde se encuentran 10 campos que configuran la conexión, así como le da un valor a la variable *salto* para conocer cuál será el salto de frecuencia.

4.3.3.5 Paquetes de datos

En el momento que se establece la conexión, existe por una parte, el maestro, encargado de configurar la conexión y, por otra, el esclavo.

En los paquetes de datos, vamos a centrarnos en la cabecera. Está compuesta por una serie de campos y son los siguientes:

- LLID: identificador de enlace lógico.
- NESN: siguiente número de secuencia esperado.
- SN: número de secuencia.
- MD: campo para indicar si hay más datos.
- RFU: reservado para uso futuro.
- Length: indica el tamaño en bytes de la carga útil.

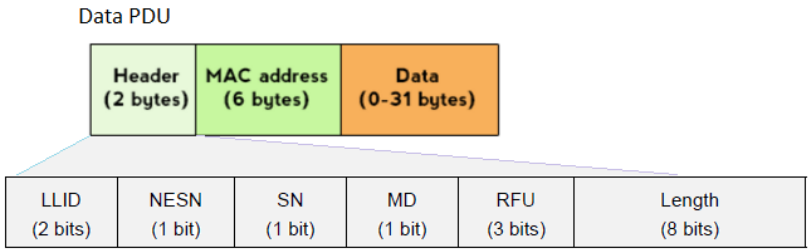


Figura 17. PDU de datos

4.3.3.6 Anuncio

El tiempo de envío debe ser un múltiplo entero de 0.625 ms y existe un mínimo de 20 ms y un máximo de 10.24 segundos. Cuando un dispositivo pretende enviar mensajes de anuncio, éste puede enviar un mensaje por cada uno de los tres canales disponibles, es decir, uno por el canal 37, otro por el canal 38 y otro por el canal 39.

No importa el tipo de paquete de anuncio que se ha enviado, siempre y cuando el tiempo de inicio entre dos paquetes de anuncio sea menor o igual a 10 ms. En la figura siguiente podemos observar un esquema donde queda claro cómo funciona el proceso de un evento de anuncio:

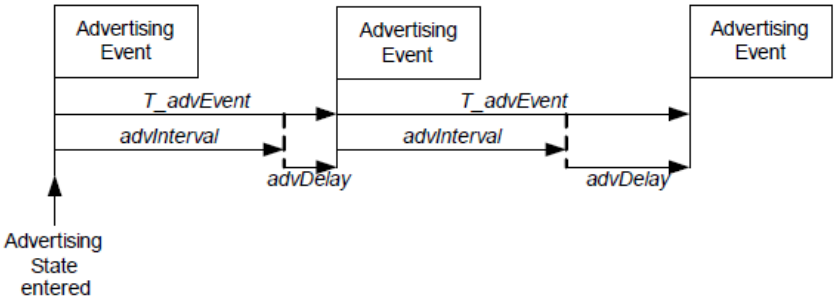


Figura 18. Evento de anuncio

Si el tipo de paquete de anuncio es escaneable o conectable, el dispositivo que lo reciba puede contestar con un SCAN_REQ o con un INITIATION_REQ. Estas solicitudes deberán ser mandadas por el escaneador en el mismo momento que reciba el paquete de anuncio y, obviamente, por el mismo canal. A continuación se puede observar en la figura el proceso de anuncio, solicitud y respuesta por el canal 38, además, este proceso no debe durar más de 10 ms:

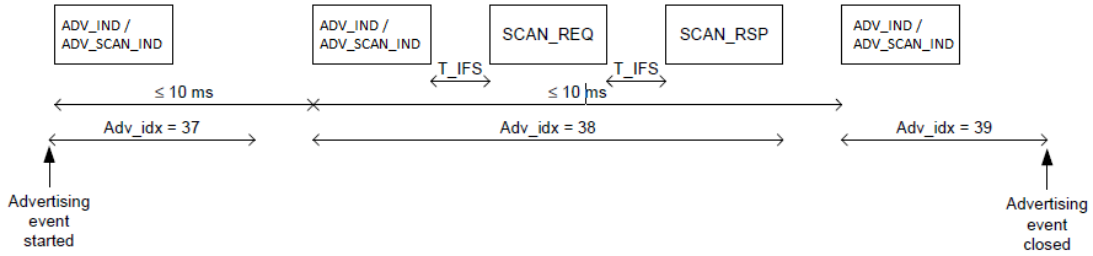


Figura 19. Anuncio y escaneo

En la siguiente figura se puede observar que, para una petición de conexión sucede que, cuando se envía dicha petición, el evento de anuncio finaliza.

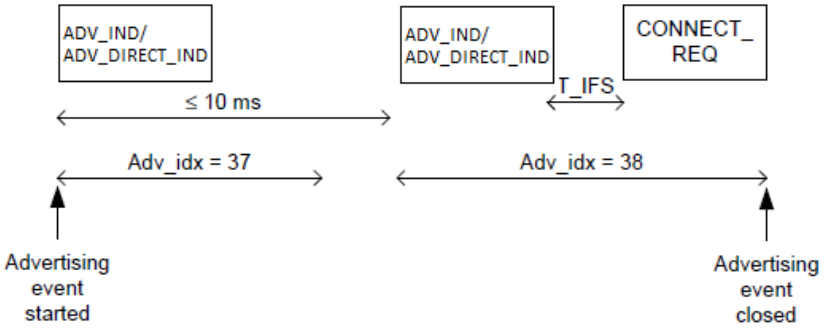


Figura 20. Proceso de anuncio completo

4.3.3.7 Escaneo

El proceso de escaneo tiene dos parámetros destacables. El primero de ellos es la ventana de escaneo y el segundo, el intervalo de escaneo.

- La ventana de escaneo es la cantidad de tiempo utilizado en modo escucha.
- El intervalo de escaneo es la frecuencia en la que sucede la escucha.

Cabe destacar que estos detalles influyen en el consumo energético, así que son sumamente importantes.

Existen dos tipos de escaneo, por una parte, el escaneo pasivo y, por otra, el escaneo activo. A continuación se explica detalladamente cada uno:

- **Escaneo pasivo:** solamente se escuchan los paquetes de anuncio, por este motivo, la capa de enlace sólo puede recibir paquetes, no puede transmitir ninguno. En el caso de que sea objetivo de un ADV_DIRECT_IND, podrá contestar con un CONNECT_REQ.
- **Escaneo activo:** se pueden escuchar todos los paquetes y puede transmitir paquetes de solicitud de escaneo para que el anunciador mande un paquete de respuesta de escaneo donde se pueda mandar información sobre el anunciador.

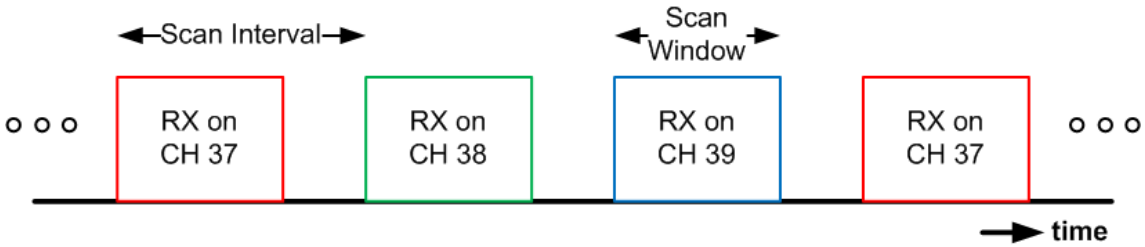


Figura 21. Escaneo

4.3.3.8 Conexión

Una vez visto los procesos de anuncio y escaneo, procedemos a explicar cómo funciona el proceso más importante, el de la conexión.

Cuando un dispositivo maestro escanea y recibe un paquete de anuncio con la propiedad conectable, éste envía una paquete de solicitud de conexión, CONNECT_REQ. Entonces, el esclavo responde, se inicia una conexión y se intercambian información de ésta como, por ejemplo, el incremento de los saltos de frecuencia.

La conexión es un intercambio de información entre maestro y esclavo, y cada intercambio es llamado evento de conexión.

En el proceso de conexión existen tres parámetros que hay que destacar. Ellos son el intervalo de conexión, la latencia del esclavo y el tiempo de supervisión de la conexión.

- El intervalo de conexión es el tiempo transcurrido entre los inicios de dos eventos de conexión. Este puede estar entre los 7.5 ms y los 4 s.
- La latencia del esclavo son el máximo número de eventos de conexión que el esclavo puede saltarse sin que exista desconexión.

- Tiempo de supervisión de conexión es el tiempo máximo entre dos paquetes de datos recibidos satisfactoriamente antes de que se produzca la desconexión.

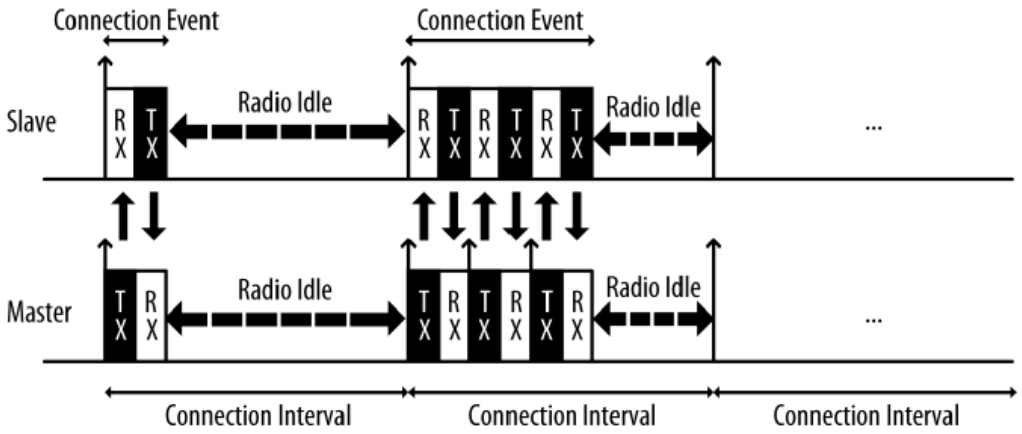


Figura 22. Eventos de conexión

En todas las conexiones existen procesos de control. En estas conexiones también existen estos procesos, por tanto, vamos a explicar los dos más importantes.

- **Cambio de los parámetros de la conexión:** los parámetros de una conexión son establecidos por el maestro, pero las condiciones pueden variar y entonces es necesario un cambio en esto parámetros. La capa de enlace permite que, tanto el maestro como el esclavo, soliciten nuevos parámetros. Sin embargo, si es el maestro el solicitante, lo puede hacer directamente. Para el caso del esclavo, necesita aprobación del maestro. Todo este proceso se produce para encontrar el mejor equilibrio entre la tasa de transmisión y consumo energético.

- **Encriptado del enlace para intercambio de información:** las claves son generadas por el host, por este motivo, la capa de enlace realiza el cifrado y descifrado de datos.

4.4 Host

4.4.1 La capa de interfaz de control del host (HCI)

La capa de interfaz de control del host permite la comunicación entre el host y el controlador. El host y la aplicación se ejecutan en la CPU y el controlador está en un chip. Suelen haber sensores en un solo chip donde se ejecutan las tres capas simultáneamente con una CPU de bajo consumo.

Los dos tipos de paquetes que hemos visto anteriormente: los de datos y los de anuncios se diferencian en que el paquete de anuncio tiene una carga útil de 31 bytes y el paquete de datos, 27 bytes, aunque mucha información sea de protocolos superiores.

4.4.2 El protocolo de control lógico y adaptación de enlace (L2CAP)

Este protocolo realiza una multiplexación de los protocolos de las capas superiores y lo implementa en los paquetes BLE y al contrario.

Los protocolos superiores son el protocolo de atributos (*ATT*) y el protocolo de gestión de seguridad (*SMP*). El protocolo de atributos es el encargado del intercambio de información en BLE y el protocolo de gestión de seguridad se encarga de generar las claves de seguridad en las conexiones.

La cabecera del paquete L2CAP ocupa 4 bytes, así pues, el tamaño de la carga útil es reducido en 4 bytes.

4.4.3 El protocolo de atributo (ATT)

El protocolo de atributo es un protocolo cliente-servidor. Este protocolo es el encargado de solicitar y responder sobre cuestiones relacionadas con los atributos que contiene cada servidor.

Cada atributo tiene un driver de atributo de 16 bits que es un identificador que sirve para acceder al valor del atributo, un identificador único universal cuyo número identifica el tipo de datos contenidos en el valor del atributo, los permisos de escritura y lectura y un valor.

Si un dispositivo necesita leer o escribir el valor de un atributo, el dispositivo envía una petición de lectura o escritura al servidor. El servidor contesta o bien con el valor del atributo en cuestión o con una aprobación. Si la operación es de escritura, el servidor está a la espera de que la información que se vaya a escribir sea acorde al atributo, si por el contrario no lo es, la operación será rechazada.

4.4.4 El protocolo de gestión de seguridad (SMP)

El protocolo de gestión de seguridad está formado por algoritmos de seguridad que facilitan la generación de claves de seguridad a la pila de protocolos, además de añadir seguridad a la tecnología BLE, como por ejemplo, ocultar la dirección pública Bluetooth para evitar comportamientos inoportunos.

Se distinguen dos tipos de roles:

- **Iniciador:** es el maestro de la capa de enlace.
- **Respondedor:** es el esclavo de la capa de enlace.

El protocolo de gestión de seguridad se encarga de vigilar los siguientes tres procedimientos:

- **Emparejamiento:** en este proceso se genera una clave de seguridad temporal para convertir la conexión en una conexión segura. La clave es aleatoria y no se guarda, y además no será utilizada en ninguna conexión posterior.

- **Vinculación:** en este proceso se generan e intercambian claves de seguridad permanentes y se almacenan en memoria no volátil, así pues, siempre que estén en sus rangos de cobertura se vincularán de manera automática. Con esto, el proceso de emparejamiento sólo será necesario la primera vez, las siguientes será un proceso automático.

- **Cifrado de restablecimiento:** cuando se ha vinculado con éxito, este proceso se activa para definir cómo usar las claves de seguridad en posteriores conexiones para no tener que volver a realizar el procedimiento de emparejamiento de nuevo.

4.4.5 El perfil genérico de atributo (GATT)

El perfil genérico de atributo está basado en el protocolo de atributo (*ATT*) y crea las directrices para organizar e intercambiar la información entre aplicaciones. La arquitectura que se utiliza es la de cliente-servidor, pero los datos son ordenados en servicios y que, a su vez, éstos están formados por características que están compuestas por atributos con diferentes funciones.

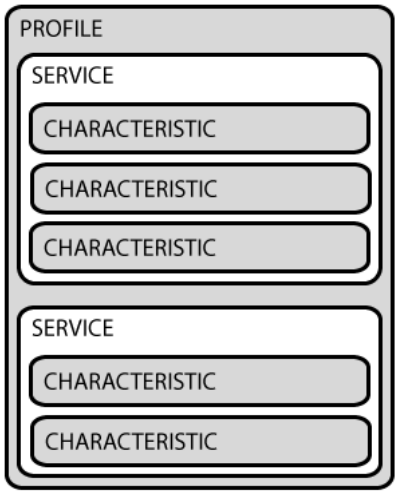


Figura 23. Perfil Genérico de Atributo

En Bluetooth clásico también existen perfiles de uso, en BLE, los perfiles son sencillos y permite un ajuste óptimo a casos específicos y así pues, se puede llegar a lograr una mejoría en el rendimiento y bajo consumo de energía.

A continuación se van a detallar algunos de los perfiles GATT aprobados por el SIG, extraído de las especificaciones de BLE:

- Find me Profile
- Proximity Profile
- Glucose Profile

Existen dos tipos de roles para los dispositivos:

- **Cliente:** es el cliente ATT. Envía solicitudes al servidor. El cliente GATT no conoce ningún servicio del servidor, para ello se ejecuta el descubrimiento de servicios en el cual todos los servicios se darán a conocer.

- **Servidor:** es el servidor ATT. Recibe todas las solicitudes del cliente. Se encarga de almacenar la información, organizarla según sus atributos y hacer posible su uso por el cliente que lo solicite.

Como anteriormente se ha explicado, en ATT cada atributo tiene un driver de atributo de 16 bits, unos permisos que pueden ser de lectura y escritura y un valor. Los atributos son los que componen las características de un servicio y cada servicio tiene asignado un UUID (*Universally Unique IDentifier*).

Los UUIDs son utilizados por muchos protocolos y tienen un tamaño de 16 bytes. También en las últimas versiones se han añadido otros formatos de UUID, de 2 bytes y de 4 bytes.

Para volver a conseguir el UUID de 16 bytes, se insertan los UUIDs acortados en la base UUID siguiente:

XXXXXXXX-0000-1000-8000-00805F9B34FB

Donde en XXXXXXXX se añadirían los UUIDs acortados.

Glucose	org.bluetooth.service.glucose	0x1808	GSS
Health Thermometer	org.bluetooth.service.health_thermometer	0x1809	GSS
Heart Rate	org.bluetooth.service.heart_rate	0x180D	GSS
HTTP Proxy	org.bluetooth.service.http_proxy	0x1823	GSS

Figura 24. Tipos de servicios

En la figura anterior, podemos observar el UUID aprobado por el SIG de Bluetooth para el Heart Rate, que en este caso es 0x180D. Toda esta información está disponible en https://www.bluetooth.com/specifications/gatt/services/.

Cada atributo está compuesto por:

Driver	Tipo	Permisos	Valor

El driver tiene un identificador de 2 bytes. El tipo de atributo es un UUID que suele ser de 2, 4, 16 bytes, y determina el tipo de datos que se va a presentar en el valor del atributo.

El campo permisos indica qué tipo de operaciones se pueden realizar y con qué medidas de seguridad.

A continuación, se listan los permisos de acceso:

- **Ninguno:** el atributo no puede ser ni leído ni escrito.
- **Lectura:** el atributo puede ser leído.
- **Escritura:** el atributo puede ser escrito.
- **Lectura y escritura:** el atributo puede ser leído y escrito.

En estos permisos también hay unos parámetros de seguridad y que a veces son necesarios para que puedan ser accesibles por el cliente:

- **No se requiere encriptado:** el atributo es accesible siempre.
- **Autentificación de encriptado no requerida:** el atributo es accesible siempre y cuando la conexión esté encriptada, aunque las claves no estén autentificadas.
- **Autentificación de encriptado requerida:** el atributo es accesible cuando la conexión esté encriptada con claves autentificadas.

Existen atributos que necesitan autorización para poder acceder a ellos. Si se intenta acceder a un atributo que requiere autorización y el cliente no la posee, éste recibirá un error de permiso denegado.

El último campo es el del valor. En este campo es donde está almacenada toda la información útil y la especificación define que el valor debe ser como máximo de 512 bytes.

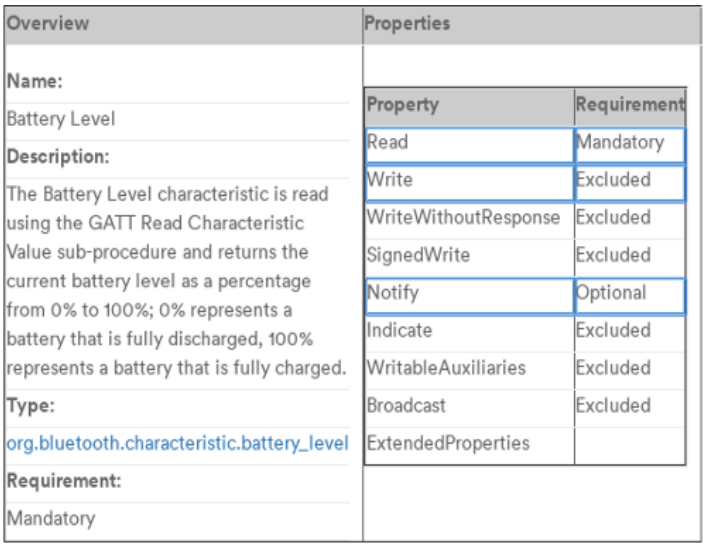


Figura 25. Ejemplo de servicio

En la figura anterior se puede observar la característica para acceder al nivel de la batería. Este es un ejemplo de cómo funcionan los servicios y características.

4.4.6 El perfil genérico de acceso (GAP)

El perfil genérico de acceso define cómo los dispositivos BLE pueden controlar la detección de otros dispositivos, la conexión, etc. para garantizar el buen funcionamiento entre todos los dispositivos aunque sean de diferentes fabricantes. Se establecen un conjunto de normas para regular a nivel bajo el funcionamiento de todos los dispositivos que utilicen la tecnología BLE.

Los perfiles Bluetooth establecen la iteración vertical desde la capa física hasta L2CAP así como también las interacciones entre capas de dos dispositivos conectados. Todos los perfiles establecen las normas para que el descubrimiento de servicios se realice correctamente y poder encontrar servicios de aplicaciones e información de conexión para que dos dispositivos puedan interactuar a nivel de aplicación.

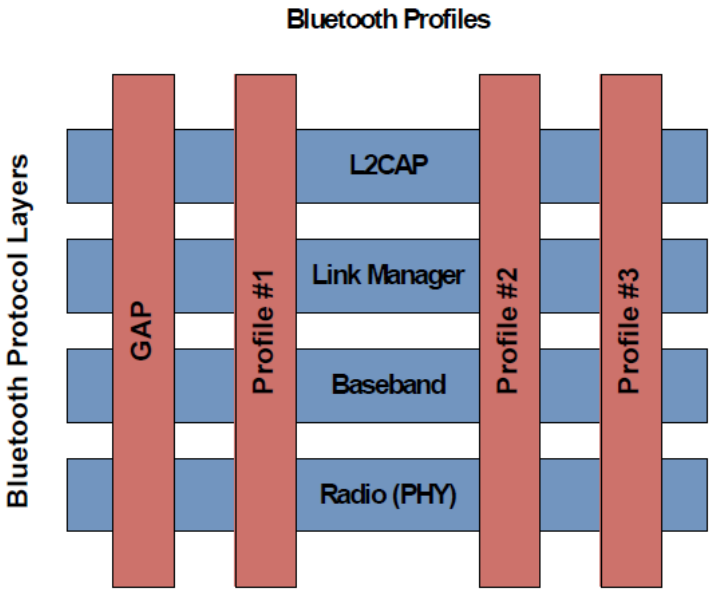


Figura 26. Perfiles de Bluetooth

Un dispositivo puede tener uno o más roles. En la siguiente lista se muestran los cuatro roles que un dispositivo puede tener:

- **Emisor:** el dispositivo envía paquetes de anuncio. Es el anunciante en la capa de enlace.
- **Observador:** el dispositivo escucha los datos de los mensajes de anuncio. Sería el escaneador en la capa de enlace.

- **Central:** el dispositivo es el iniciador y suele tener varias conexiones con otros dispositivos. Sería el maestro en la capa de enlace. Empieza escuchando los paquetes de anuncio de otros dispositivos y es cuando establece una conexión con el dispositivo que ha mandado el paquete de anuncio. El coste computacional en este rol es mayor que en los otros roles, por este motivo, este rol lo suelen tener los teléfonos móviles o *smartphones*.

- **Periférico:** Sería el esclavo en la capa de enlace. Usa los paquetes de anuncio para permitir que el maestro lo encuentre y establecer una conexión. No requiere gran coste computacional, de ahí que, estos dispositivos tengan un coste de fabricación bajo.

Cabe destacar que un dispositivo puede ser al mismo tiempo cliente GATT y servidor GATT.

4.4.6.1 Descubrimiento

Para entender el descubrimiento en BLE, existen modos de detectabilidad y procedimientos de descubrimiento.

Los modos de detectabilidad otorgan cierta libertad a los diseñadores de periféricos para que puedan diseñar sus productos en consonancia a sus funciones:

- **Modo no detectable:** no será detectado por ningún periférico.

- **Modo de descubrimiento limitado:** permite que el dispositivo sea detectado por un tiempo limitado pero será detectado si el dispositivo central está en procedimiento de descubrimiento limitado.

- **Modo de descubrimiento general:** permite que el dispositivo sea detectado por un tiempo limitado pero será detectado si el dispositivo central está en procedimiento de descubrimiento general.

Por otra parte, existen los procedimientos de descubrimiento, según la especificación, esto son los siguientes:

- **Procedimiento de descubrimiento limitado:** el dispositivo central está en modo escaneo. Si recibe un paquete con el modo de descubrimiento limitado habilitado, éste lo pasa a la aplicación para que siga el proceso correspondiente.

- **Procedimiento de descubrimiento general:** el dispositivo central está en modo escaneo. Si recibe un paquete con el modo de descubrimiento general habilitado, éste lo pasa a la aplicación para que siga el proceso correspondiente.

4.4.6.2 Establecimiento de conexión

Para establecer una conexión, el periférico debe estar en modo conectable. Existen algunos modos y procedimientos que pueden controlar cómo es la interacción entre los dispositivos.

Los modos de establecimiento de conexión son los siguientes:

- **Modo no conectable:** un dispositivo central no podrá en ningún caso establecer conexión con el periférico.
- **Modo de conexión directo:** un periférico envía paquetes ADV_DIRECT_IND. Con estos paquetes se puede conseguir rápidas reconexiones y solamente será recibido por el dispositivo central.
- **Modo de conexión indirecto:** un dispositivo envía paquetes de anuncio ADV_IND. Con este modo, el periférico será conectable durante un tiempo.

Finalmente, vamos a listar los procedimientos de establecimiento de conexión que dependen del tipo de filtrado que el dispositivo central ejecuta.

- **Procedimiento de establecimiento de conexión automática:** se añaden en una lista todos los dispositivos conocidos y se conecta al primero que se detecte. Este procedimiento se utiliza cuando el dispositivo central conoce un conjunto de dispositivos y no tiene preferencia por ninguno.
- **Procedimiento de establecimiento de conexión general:** es el usado para conectarse a un periférico nuevo y desconocido. El dispositivo central escucha paquetes de anuncio entrantes y decide si se conecta o no. En el momento que el periférico es el elegido, el dispositivo central se conecta con el periférico mediante el procedimiento de establecimiento de conexión directo.
- **Procedimiento de establecimiento de conexión selectiva**: este procedimiento se utiliza cuando el dispositivo central sólo quiere conectarse a ciertos periféricos conocidos.
- **Procedimiento de establecimiento de conexión directo:** En este procedimiento, el host usa la capa de enlace para establecer una conexión con un dispositivo por su dirección Bluetooth. Este procedimiento puede fallar porque el dispositivo puede estar disponible o en modo no conectable.

El dispositivo central tiene dos métodos diferentes de establecer una conexión. El primer método se divide en dos pasos: el primero, escanear y el segundo, conexión directa a un dispositivo, con dirección Bluetooth conocida, detectado. El segundo método usa el controlador para seleccionar el dispositivo sin conocer si están en el rango de cobertura.

Capítulo 5. Mi Band 3

En este capítulo, vamos a tratar de dar a conocer (si no se conoce todavía), qué es la pulsera de actividad Mi Band 3, es decir, qué fabricante la ha creado, qué características tiene, etc.

Figura 27. Pulsera de actividad Mi Band 3

La pulsera de actividad Mi Band 3 ha sido fabricada por la empresa internacional Xiaomi. Es la tercera generación de las pulseras de actividad más vendida. La primera pulsera que vio la luz fue Mi Band 1S, para más tarde presentar la pulsera Mi Band 2. La diferencia principal entre estos dos modelos es que la pulsera Mi Band 2, ya poseía pantalla. Además, como curiosidad, los UUID de las características de la pulsera Mi Band 1S eran diferentes a la de la segunda generación. En la tercera generación han mantenido los mismos UUID que utilizaba la pulsera Mi Band 2.

5.1 Especificaciones de Mi Band 3

Nombre completo: Xiaomi Mi Band 3 Activity bracelet

Pantalla: 078" OLED *Touch Screen* con una resolución de 128 x 80 px

Peso: 8.5 gramos

Tamaño: 17.9 x 46.9 x 12 mm

Material de la pulsera: Elastómero termoplástico

Longitud ajustable de la pulsera: 155 -216 mm

Sistemas operativos soportados: Android 4.4 o iOS 9.0 y superiores

Capacidad de la batería: 110 mAh

Tipo de batería: Batería Ion Litio

Tiempo duración batería en *stand-by*: Alrededor de 20 días

Tiempo total de carga: 2 horas

Voltaje de entrada: 5V DC

Intensidad de entrada: 250 mA máx.

Nivel de resistencia: 5 ATM

Temperatura de operación: -10ºC – 50ºC

Conexión: Bluetooth 4.2 BLE

5.2 Precauciones

En este apartado se añaden algunas observaciones que el fabricante ha decidido incluir en los manuales. Pueden ser importantes o no, pero como información, nunca está de más saber acerca de la pulsera Mi Band 3. A continuación, se muestran estas precauciones:

- Cuando la pulsera de actividad está en el modo de medir el pulso cardíaco, mantener la pulsera lo más fija posible.
- La pulsera Xiaomi Mi Band 3 tiene un ratio de resistencia al agua de 5 ATM, es decir, puede ser mojada, pero no ser utilizada en saunas y en buceo.
- El botón y la pantalla de la pulsera Mi Band 3 no soporta la utilización debajo del agua. Cuando el dispositivo esté mojado, para que funcione, seque la superficie de la pulsera, ya que si está mojada no funcionará.

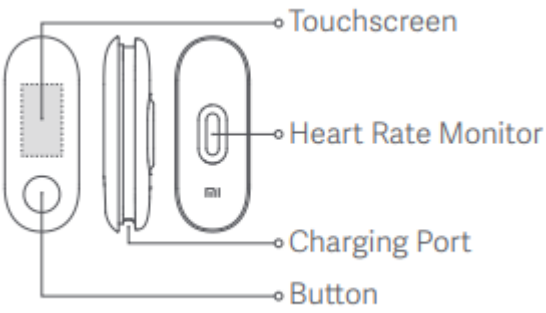


Figura 28. Elementos de la pulsera Mi Band 3

Capítulo 6. Mi Maps

6.1 Investigaciones previas

Antes de empezar a programar la aplicación, hace falta realizar un estudio previo de la tecnología BLE en la pulsera Mi Band 3.

Lo primero que necesitamos conocer son los servicios y las características de la pulsera, para ello, investigué en el foro archiconocido como es www.stackoverflow.com. Allí encontré alguna de las características más importantes:

- La característica con UUID "*00000009-0000-3512-2118-0009af100700*" sirve para la autenticación, que más adelante explicaré el procedimiento.
- La característica con UUID "*00000010-0000-3512-2118-0009af100700*" envía una señal cuando se pulsa el botón en la pulsera Mi Band 2, pero en este caso, tendremos algún problema menor, ya que la pulsera es la Mi Band 3.
- La característica con UUID "*0x2A46"* se encarga de recibir texto en la pulsera Mi Band 3.

Con estas características, aunque no son muchas, ya son suficientes para llevar a cabo el proyecto "Mi Maps" o, por lo menos, lo relativo a la pulsera está más claro.

Una vez conocidas ciertas características, necesito saber más acerca de la pulsera. Para ello, conseguí la aplicación oficial Mi Fit en formato .apk y conseguí decompilarla desde este enlace http://www.javadecompilers.com/apk. Con esta aplicación decompilada, aunque no pude ver grandes cosas, sí que obtuve algunas características más.

- La primera característica que conseguí fue "*0x2A06*", ésta se encarga de enviar vibraciones a la pulsera.
- La segunda característica que conseguí fue "*00000007-0000-3512-2118-0009af100700*", con ésta puedo conseguir los pasos, las calorías y la distancia recorrida.
- Las últimas características que puedo conseguir de la aplicación decompilada fue "*00000005-0000-3512-2118-0009af100700*", con esta característica se conseguiría toda la actividad en cada minuto guardada en la memoria de la pulsera Mi Band 3.

Conocemos todo lo necesario referente a la pulsera para extraer toda la información guardada en la pulsera Mi Band 3.

6.2 Pasos previos a la programación de la aplicación

El problema principal que existe es,

... ¿Para qué plataforma creo la aplicación? ¿iOS? ¿Android? ¿iOS y Android?...

6.2.1 ¿Aplicaciones nativas vs Aplicaciones Híbridas?

Software Aplicaciones híbridas

Después de estudiar qué sería mejor si hacer una aplicación híbrida, es decir, un mismo código en C# para las plataformas iOS y Android o crear aplicaciones nativas, explico mis conclusiones y mi decisión:

- En las aplicaciones híbridas no se soportan los servicios del sistema, con lo cual iba a tener problemas desde el primer momento porque para que funcione el bluetooth de la aplicación, necesito acceder a los servicios del sistema.
- En las aplicaciones híbridas se soportan las tareas en segundo plano pero limitadas a sólo acceso a la red y en el mejor de los casos.
- En la web, existen muchas más soluciones tanto para Android utilizando Android Studio como para iOS utilizando Xcode.
- En aplicaciones de iOS, al utilizar aplicaciones nativas, se pueden acceder a funciones de accesibilidad, lo cual es imposible al realizarlo con programas como Xamarin.

Por lo tanto, estas conclusiones decantan la balanza a crear la aplicación, por una parte, en Java y, por la otra, en Swift (lenguaje exclusivo de la familia Apple).

Figura 29. Logotipos de Android Studio y XCode respectivamente

6.3 Empezando a programar la aplicación

A partir de ahora, todo lo relativo a la aplicación será explicado con el lenguaje Java para la versión en Android, pero si lo considero oportuno, nombraré algunas diferencias con Swift, ya que la mayoría del código será el mismo para Java como para Swift, sólo cambiaría la sintaxis y el nombre de las funciones de cada lenguaje. Por otra parte, las imágenes sólo serán de la aplicación Android, ya que la mayoría son idénticas en iOS.

A continuación se mostrará la primera imagen de la aplicación, la cual conduce a la búsqueda de todos los dispositivos Bluetooth.

6.3.1 La búsqueda del dispositivo Bluetooth

La aplicación empieza por la búsqueda del dispositivo, en nuestro caso, la pulsera Mi Band 3. Para ello, he creado una clase llamada **searchActivity**, cuyo propósito será el de mostrar al usuario los dispositivos en una lista. A continuación, explico detalladamente cómo lo he realizado.

Figura 30.Primera pantalla de Mi Maps

Para empezar, se comprueba el bluetooth del teléfono. Para ello, se utiliza **BluetoothAdapter**, ya que se trata de una clase que se encarga de todo lo relativo a la tecnología Bluetooth. Con este BluetoothAdapter puedes saber si el teléfono tiene bluetooth, si éste está activo o no, en el último caso, se mandará una alerta a la pantalla para avisar al usuario que el bluetooth está desactivado.

En cambio, en iOS se utiliza **CBCentralManager**, encargado de realizar las funciones relativas al bluetooth en este caso. Para comprobar cómo está el bluetooth se recurre al método **centralManagerDidUpdateState**.

Una vez comprobado el estado de bluetooth, se procede a comprobar si existe una MAC de la pulsera Mi Band 3 en las preferencias del usuario. Para ello, se utiliza la clase **SharedPreferences** en Android y **UserDefaults** en iOS.

Se abren dos posibilidades:

- Si existe una MAC de una pulsera Mi Band 3, se entiende que la búsqueda no hace falta ser inicializada, ya que la pulsera ya ha sido vinculada anteriormente. Por este motivo, se procede a cerrar esta actividad y abrir la siguiente que será explicada.

- Si no existe tal MAC, entonces podemos empezar la búsqueda con **bluetoothAdapter.startDiscovery()**. Con esta función, cada vez que se encuentre un elemento bluetooth, mediante BroadcastReceiver (acción o variable que se manda por

broadcast), se incluirá en un array llamado arrayFoundBtDevices y éste, a su vez, se añadirá a una lista que se visualizará mediante el objeto ListView con la ayuda de un ArrayAdapter de objetos BluetoothDevice. En Swift, sin embargo, no hace falta un adaptador bluetooth como en Android, en este caso, es **CBCentralManager**, objeto encargado de buscar con el método propio **didDiscover()** y conectar con el método propio **didConnect()**.

Una vez mostrada la lista con todos los ítems, se puede seleccionar la pulsera Mi Band 3. La aplicación sólo permite avanzar si el nombre es "Mi Band 3" porque se comprueba con el método item.getName(), siendo ítem el objeto BluetoothDevice.

Cabe destacar que, antes de seguir a la siguiente pantalla, cuando es seleccionada la pulsera Mi Band 3, se debe guardar en las preferencias del usuario la MAC seleccionada para que, en posteriores accesos, directamente pase a la siguiente pantalla.

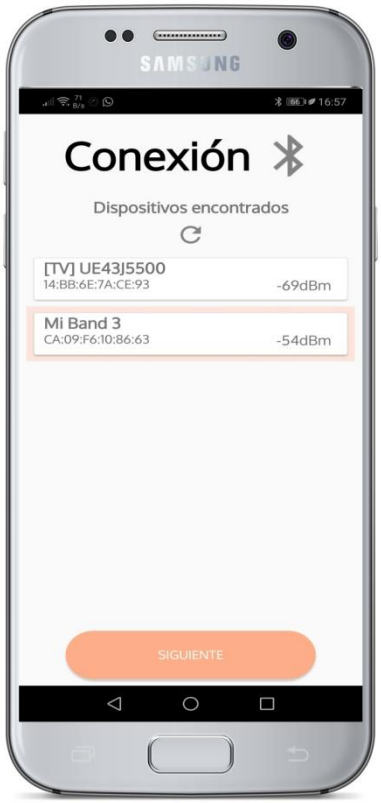


Figura 31. Lista de dispositivos

6.3.2 Cuando ya está identificado el dispositivo Bluetooth

Una vez hemos pasado el proceso de búsqueda del dispositivo o ya lo teníamos guardado en las preferencias del usuario, vamos a ejecutar un servicio llamado **BTService**.

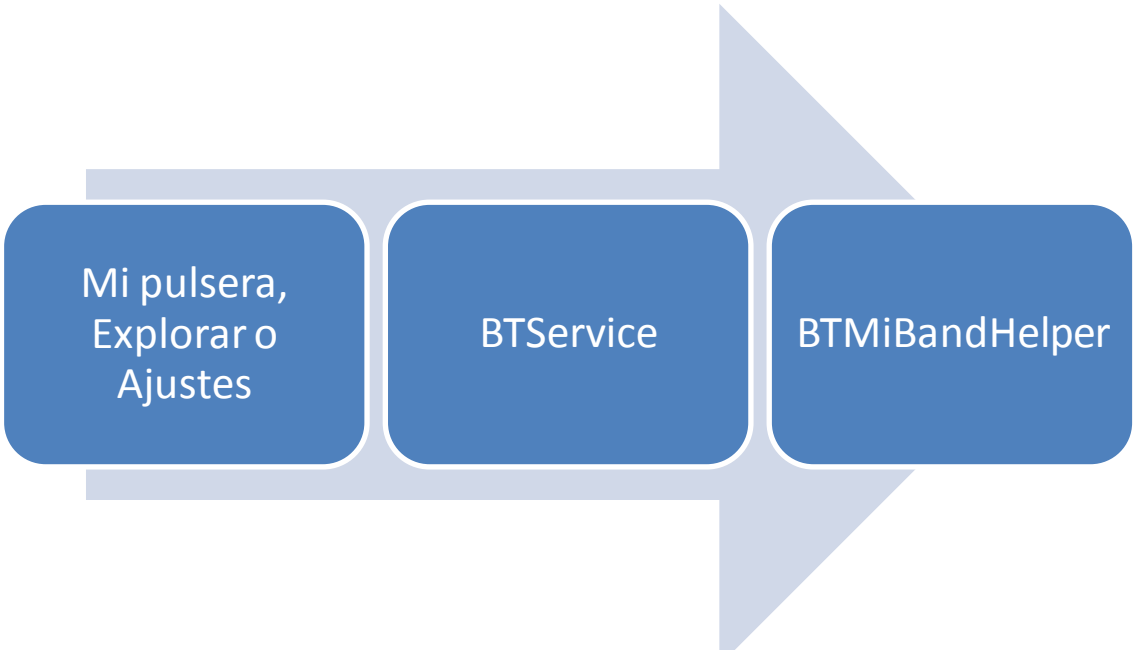


Figura 32. Clases utilizadas en la conexión Mi Band 3

Aunque más tarde va a ser explicado con más detalle, voy a aclarar qué es Mi Pulsera, Explorar y Ajustes, para poder explicar qué función tiene BTService.

Mi Pulsera es la vista principal. Allí se encontrará toda la información relativa a los pasos, calorías, actividades, sueño, etc. Explorar es la vista encargada de enviar las direcciones a la pulsera. Y, por último, en Ajustes puedes desde borrar todos los trayectos hasta comprobar si las vibraciones funcionan, por ejemplo.

Con este gráfico intento explicar qué función tiene BTService. La principal función de BTService es servir de puente entre las vistas, que en nuestro caso son Mi Pulsera, Explorar o Ajustes, y el objeto que interactúa directamente con la pulsera Mi Band 3, llamado **BTMiBandHelper**.

"BTMiBandHelper es la clase que más cerca va a estar del dispositivo Bluetooth."

Entonces,

¿Cómo se comunica, por ejemplo, Mi Pulsera con BTMiBandHelper?

Muy sencillo, existe dos funciones imprescindibles para enviar variables desde un servicio a una clase o entre dos clases, etc. Estas funciones son, por una parte, **BroadcastReceiver**, y al crear esta clase añades las palabras claves para saber qué variables te interesan y recibirlas cuando se envíen desde otra clase o servicio; y, por otra parte, **BroadcastIntent**, que se utiliza para mandar alguna variable.

El procedimiento es así, por ejemplo, desde Ajustes se envía la orden de vibrar con la palabra clave "*vibrate*" por medio de un broadcastIntent. BTService lo recibe porque tenía registrado "*vibrate*" como palabra de interés. BTService, al tener acceso directo a BTMiBandHelper, puede mandar la orden de vibrar directamente a la pulsera. Si, en el caso contrario, es la pulsera

la que quiere enviar información, BTService es el que manda mediante un broadcastIntent diferente al anterior y desde Mi Pulsera, Ajustes o Explorar se recibe la variable mediante la clase BroadcastReceiver.

Antes de seguir, quería hacer un inciso, para explicar de qué manera lo he realizado en iOS. En iOS, en la clase MiPulsera estaba lo que en Android sería BTMiBandHelper y utilizar instancias desde toda la aplicación, porque en iOS no se permiten los servicios, entendiendo servicio por un proceso que se ejecuta en segundo plano.

Volviendo a Android, con la clase BTMiBandHelper, intentaré resumir cerca de 2.000 líneas, de manera que queden claras cada una de las funciones:

- Primero, volvemos a comprobar la conexión bluetooth con BluetoothAdapter y, una vez hecho esto, procedemos a conectar el dispositivo cuya MAC estaba guardada en las preferencias de usuario con la función de BluetoothAdapter llamada **getRemoteDevice()**, donde se introduce como parámetro la MAC guardada.

- Segundo, mediante la función **connectGatt()**, procedemos a crear una conexión GATT. GATT es el acrónimo de Generic Attribute Profile y define la manera en que dos dispositivos BLE pueden comunicarse usando los servicios y características explicados anteriormente.

"La comunicación se realiza mediante un protocolo llamado ATT (Attribute Protocol), cuya función es almacenar servicios, características y datos relacionados en una tabla usando identificadores UUID de 16-bit para cada entrada en la tabla." [5]

Lo más importante de GATT es que las conexiones son exclusivas, un periférico Bluetooth LowEnergy sólo puede ser conectado a un dispositivo, en nuestro caso, un teléfono móvil, a la vez.

La función connectGatt() que se utiliza en Android posee retorno de llamadas o retrollamadas en algunos métodos. Los métodos que tienen retrollamadas más importantes son:

- **OnServicesDiscovered:** Este método, propio de BluetoothGatt, se encarga de encontrar todas las características, ya que recorre cada uno de los servicios que el dispositivo posee. Lo que he añadido a este método es el activar el descriptor cuyo UUID es "*00002902-0000-1000-8000-00805f9b34fb*" a todas las características que he considerado oportunas. Como anteriormente se ha explicado, cada servicio tiene características y cada característica tiene un descriptor. Al activar el descriptor, cuando haya algún cambio en la característica especificada, ésta enviará a la retrollamada onCharacteristicChanged que a continuación la explicaré.

- **OnCharacteristicRead:** Este método, propio de BluetoothGatt, se encarga de recopilar la información que brinda la pulsera Mi Band en el momento de la conexión. Por ejemplo, cuando todavía no está realizada la autenticación, la pulsera envía la información de la batería utilizando la característica de ésta cuyo UUID es "*00000006-*

0000-3512-2118-0009af100700", más adelante se explicará cómo se extrae la información de la batería, hardware y firmware, entre otros.

- **OnCharacteristicChanged:** Como se había comentado antes en el método OnServicesDiscovered, cuando habilitas las notificaciones en el descriptor de una característica determinada, cuando ésta cambie, se ejecutará este método con la característica como parámetro y se podrá acceder al valor con la función characteristic.getValue(). Esto sirve, por ejemplo, por si el valor de la batería desciende en un 1%, automáticamente se ejecuta este método y se procede a actualizar el valor en nuestra aplicación. Más tarde se explicará el método de autenticación, la función OnCharacteristicChanged se vuelve imprescindible, ya que te va informando en qué paso estás de la autenticación, así como las contestaciones por parte de la pulsera Mi Band 3.

- **OnCharacteristicWrite:** Este método se utiliza para escribir directamente sobre la característica deseada, pero en mi caso no la he utilizado porque veía más práctico crear una función creada para mis necesidades.

Como simple curiosidad, también existe un método llamado **onReadRemoteRssi**, utilizado para saber en todo momento la fuerza de la señal recibida, RSSI según sus siglas en inglés, *Received Signal Strength Indicator*, y es una escala de referencia, en relación a 1 mW. La variable está en dBm, es decir, si recibes un 0 que en este caso es el máximo, quiere decir que son 0 dBm que es igual a 1 mW. El mínimo son -113 dBm, prácticamente el periférico está sin cobertura.

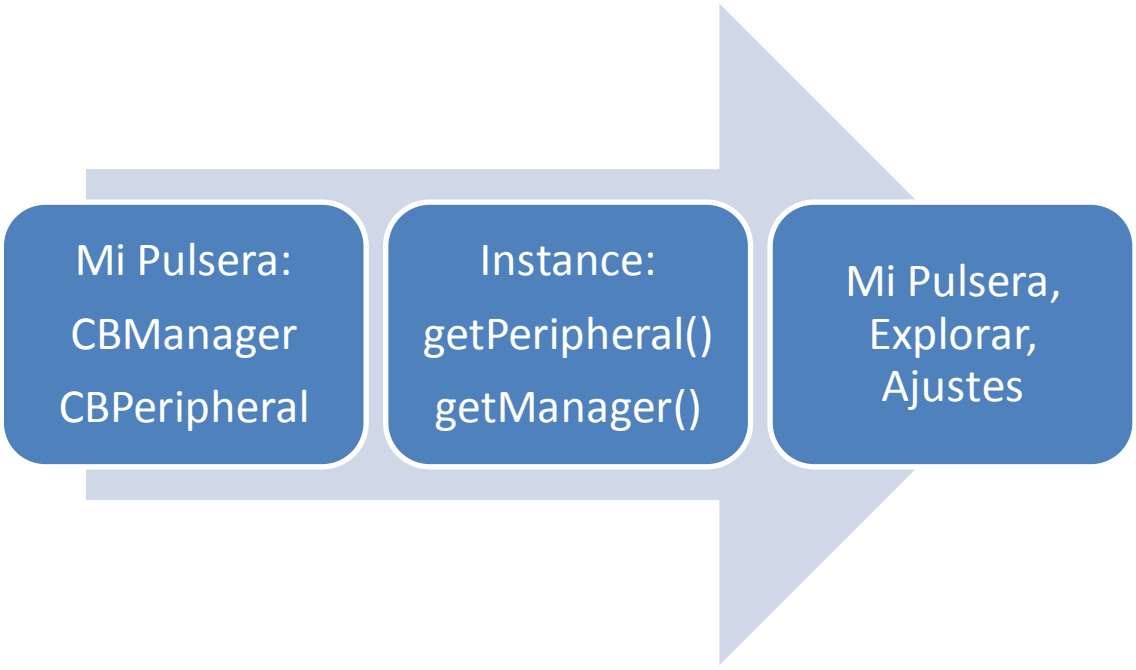


Figura 33. Conexión con Mi Band 3 en Swift

Como se puede observar en el gráfico superior, la instancia se crea en Mi Pulsera, donde la conexión se crea, y una vez está hecha la conexión, ya se puede acceder a todo lo relativo al periférico bluetooth desde todas las ventanas de la aplicación, con el simple método getPeripheral() para conseguir el objeto CBPeripheral, que en nuestro caso, resulta ser la pulsera Mi Band 3.

En la parte de iOS, una vez tenemos instaciado el CBCentralManager y el CBPeripheral. Una vez CBCentralManager ya ha logrado la conexión con el periférico bluetooth, CBPeripheral es el que dirige todo lo relativo a la conexión. Los métodos más importantes de CBPeripheral se explican a continuación:

- **DidDiscoverServices:** Este método es el análogo a OnServicesDiscovered en iOS. Este método te muestra cada servicio como un objeto CBService, el cual almacena las características del propio servicio, pero para ello está el siguiente método: didDiscoverCharacteristicsFor.

- **DidDiscoverCharacteristicsFor:** Este método se ejecuta cada vez que se descubre un servicio y a cada servicio se le extraen todas las características como objeto CBCharacteristic. Dentro de este método se puede saber si la característica es de lectura con el método characteristic.properties.contains(.read) y si la característica es de escritura con el método characteristic.properties.contains(.write). En este método, cuando se necesita habilitar las notificaciones de una característica determinada se utiliza la función .setNotifyValue(true), así pues, se activa las notificaciones para que cuando cambie alguna característica, se llame al siguiente método: didUpdateValueFor.

- **DidUpdateValueFor:** Con la característica como parámetro, este método es análogo a OnCharacteristicChanged en Android y sirve para poder acceder al valor que ha cambiado con la función characteristic.value. Como ha pasado en Android, en iOS también resulta imprescindible este método para saber en qué punto de la autenticación se está.

- **DidWriteValueFor:** Este método es análogo a OnCharacteristicWrite en iOS y tampoco lo utilizo por la misma razón que en Android. En este caso también es más cómodo crear una función propia para escribir en una determinada característica que se ajuste a tus necesidades que utilizar este método.

Una vez explicado cómo funciona tanto en iOS como en Android las conexiones, transacciones, etc. en los periféricos bluetooth, voy a explicar la estructura de los servicios y características de la pulsera Mi Band 3. Esto facilitará la comprensión de las posteriores explicaciones.

Antes de mostrar el gráfico donde se muestran dónde están los servicios, características y los descriptores. La estructura principal es la siguiente:

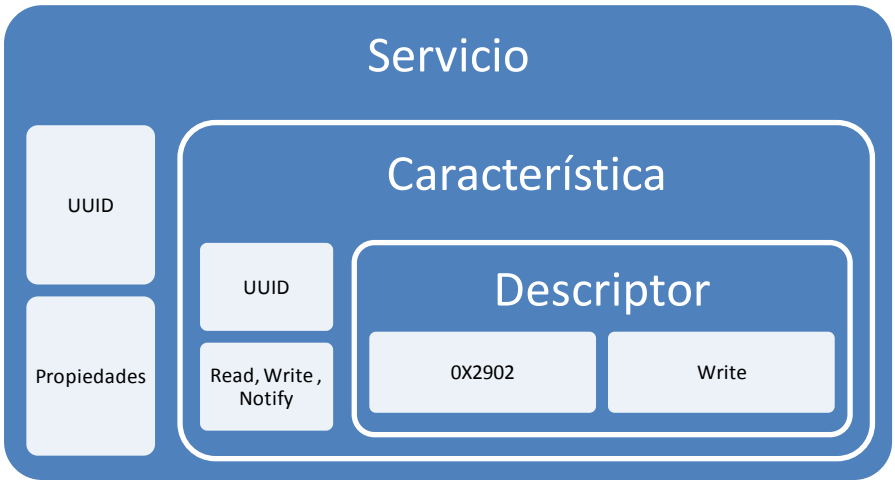
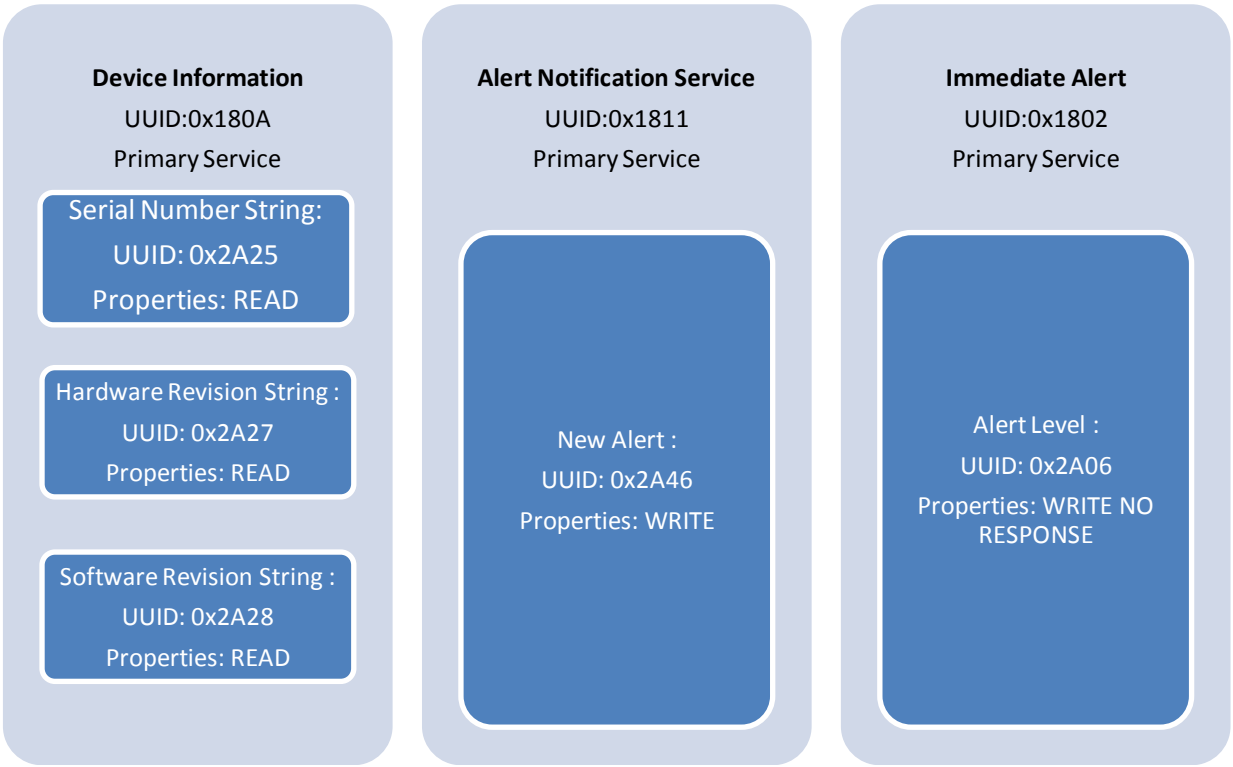


Figura 34. Estructura de un servicio

Cabe destacar que si la característica no tiene en las propiedades la opción de notificar (**Notify**), el descriptor no existirá para esa característica, ya que el descriptor sólo sirve para habilitar las notificaciones de la característica en cuestión. He llegado a esta conclusión porque en algunas características sucede esto, que activas la notificación y ésta nunca llega.

Después de estas aclaraciones, es el momento de mostrar cómo se estructuran los servicios, características, etc. en la pulsera Mi Band 3, además de destacar que en cada característica, está su descriptor cuya UUID es *0x2902*:



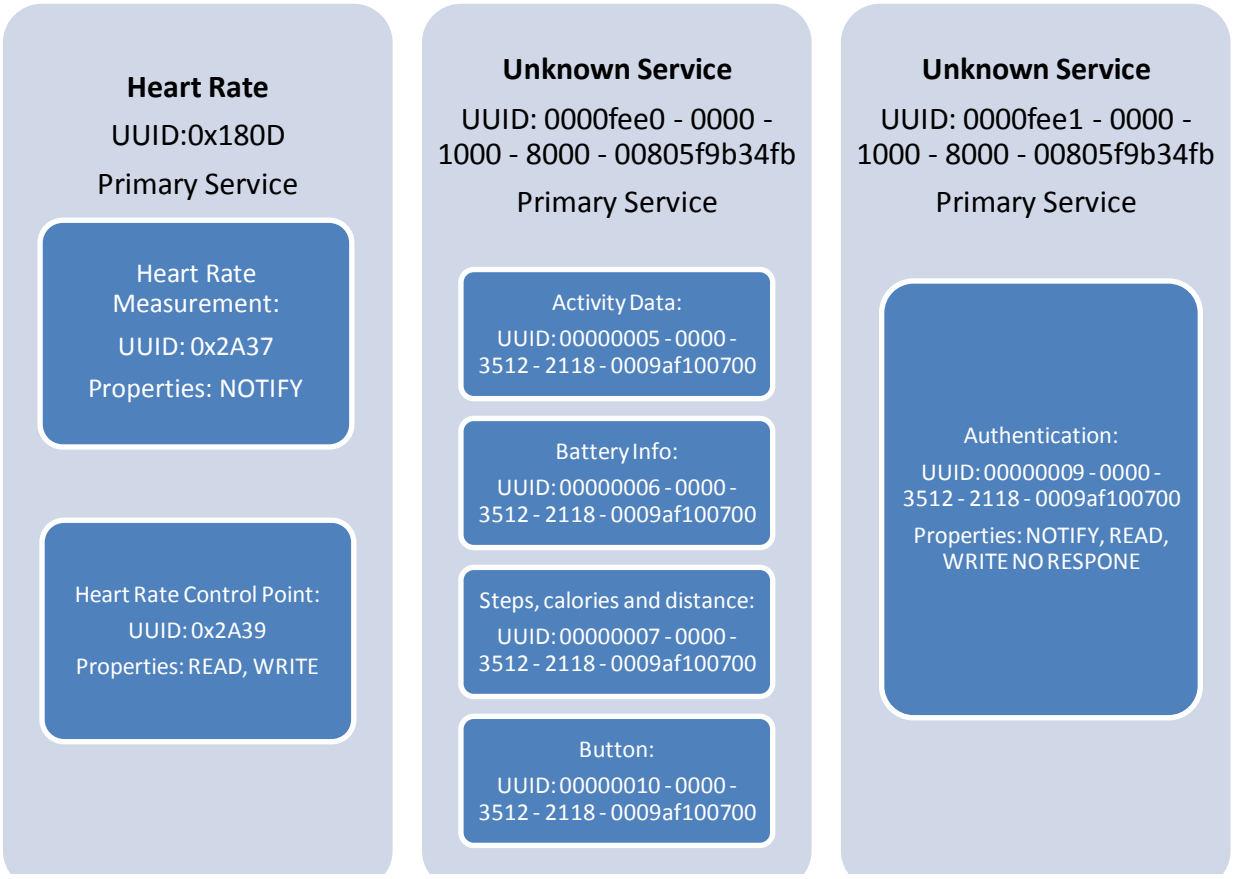


Figura 35. Estructura de los servicios

6.3.3 Autenticación

A continuación, voy a explicar cómo se realiza el proceso de autenticación y cómo proceder en las siguientes conexiones, porque una vez completado el proceso de autenticación, las siguientes conexiones serán mucho más rápidas:

Antes de empezar, la característica de autenticación es "00000009-0000-3512-2118-0009af100700", y un aspecto a tener en cuenta antes de enviar ningún dato a esta característica es habilitar las notificaciones como anteriormente se ha descrito, es decir, enviar al descriptor la habilitación de notificaciones.

Entonces, como ha quedado claro lo explicado anteriormente, procedemos a describir el proceso de autenticación. Cabe destacar que, Mi Maps está entre el 5% de las aplicaciones que utilizan la Mi Band y que, además, utiliza el proceso de autenticación. El 95% restante de las aplicaciones precisa de Mi Fit, la aplicación oficial, para poder ser utilizada correctamente.

En primer lugar, una vez habilitadas las notificaciones, enviamos a la característica de autenticación, un array encabezado con un 0x01, 0x08; seguido de 16 números que conforman la **secretkey**, o lo que es lo mismo, 128 bits, en este caso en particular, se ha enviado lo siguiente:

"[0x01, 0x08, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45]"

Esperamos la contestación, que esta vez, debe ser: *[0x10, 0x01, 0x01]*, estos números se refiere que ha tenido éxito el primer paso. Se compone de 3 dígitos en hexadecimal: el primero, 16 en decimal, se refiere a que es una contestación; el segundo, 1 en decimal, se refiere a que es una contestación referente al primer paso de la autenticación; y el último, 1 en decimal, se refiere a que el primer paso se ha realizado con éxito.

Una vez que hemos recibido la primera contestación satisfactoria, procedemos a realizar el segundo paso en la autenticación. El segundo paso se trata de hacerle una petición de la clave aleatoria que genere la pulsera. Para ello se debe enviar los siguiente: *[0x02, 0x08]*.

Esperamos la contestación, esta vez del segundo paso, que se compone por:

"[0x10, 0x02, 0x01, (16 bytes of Band secret key)]"

Se compone por los 3 primeros dígitos en hexadecimal: el primero, 16 en decimal, se refiere a que es una contestación; el segundo, 2 en decimal, se refiere a que es una contestación referente al segundo paso de la autenticación; y el último de los tres primeros dígitos, 1 en decimal, se refiere a que el segundo paso se ha realizado con éxito. Estos tres dígitos vienen acompañados por la clave secreta de la pulsera Mi Band 3.

Una vez tenemos la clave secreta enviada por la pulsera, procedemos a realizar el tercer y último paso de la autenticación. Este último paso consiste en enviar a la característica de autenticación un array con los siguientes elementos:

"[0x03, 0x08, (encrypted generated key)]"

Donde *encrypted generated key* es la clave que se mandó en el primer paso de la autenticación, en este caso, la anteriormente descrita *secretkey* de 128 bits cifrada junto a la Band *secretkey*, que mandó la pulsera en el segundo paso.

"El cifrado es de tipo AES (Advanced Encryption Standard), que utiliza un algoritmo basado en sustituir, permutar y transformar linealmente bloques de datos de 16 bytes, llamado blockcipher, repitiéndose varias veces. En cada ronda, una clave circular única se calcula a partir de la clave de cifrado y se incorpora en los cálculos. Para este caso en concreto se utiliza un modo de operación llamado ECB (Electroni Code-Book), en el cual los mensajes se dividen en bloques y cada uno de ellos es cifrado por separado utilizando una misma clave K." [6]

Una vez realizado el cifrado, se procede a enviar el resultado junto a los 2 bytes primeros, 0x03 y 0x08.

Esperamos la última contestación, que esta vez, debe ser: *[0x10, 0x03, 0x01]*, estos números se refiere que ha tenido éxito el último paso. Se compone de 3 dígitos en hexadecimal: el primero, 16 en decimal, se refiere a que es una contestación; el segundo, 3 en decimal, se refiere a que es

una contestación referente al tercer paso de la autenticación; y el último, 1 en decimal, se refiere a que el último paso se ha realizado con éxito y la pulsera ya se ha vinculado.

Cabe destacar que en el proceso de autenticación, la pulsera te solicita que pulses el botón para dar el consentimiento en la vinculación, en el caso de que no se pulsara, el proceso de autenticación no se realizaría correctamente.

Una vez realizado el proceso de autenticación correctamente y guardada la dirección MAC en Android y el identificador de CBPeripheral en iOS, me surgieron varios problemas…

¿Cómo se podría conectar directamente la pulsera sin realizar todos los pasos anteriormente descritos?

El problema era bastante grave, no por el hecho de realizar todos los pasos de autenticación, sino porque de dejarlo así, el usuario debería dar su consentimiento pulsando el botón cada vez que se iniciara la aplicación, cosa que no es útil. La solución es sencilla. A continuación explico lo que hice.

La primera vez que se conectara la pulsera, ésta se guardaría la *secretkey*. Bien pues, en las siguientes conexiones sólo tendrías que pedirle la clave de la pulsera, cifrarla con la *secretkey* y devolvérsela.

Así pues, la pulsera contesta con: *[0x10, 0x03, 0x01]*. Entonces, ya está la pulsera vinculada sin la necesidad de pulsar el botón.

Antes de avanzar tengo que añadir que en iOS fue de lo más sencillo, ya que seguí el mismo patrón que en Android, pero sin olvidarme de la gran ayuda que recibí por parte de una librería llamada CryptoSwift y su método llamado AES, sin el cual, el proceso de autenticación hubiese sido muchísimo más tedioso. Y ya que estoy hablando de una librería, decir que, en Swift, las librerías se pueden añadir con Cocoapods, entonces…

¿Qué hace Cocoapods para poder tener librerías en nuestros proyectos?

"Básicamente Cocoapods nos proporciona inyección de dependencias para iOS y se trata de de un archivo llamado Podfile donde introduces qué librerías quieres añadir, ya que los nombres de las librerías se proporcionan en GitHub y a su vez, Podfile, al ejecutarse, añade un proyecto llamado Pods donde se insertan las librerías, entonces, al añadir este proyecto al nuestro también se comparten las librerías."[7]

En Android, la librería de cifrado es javax.crypto.

6.3.4 Bases de datos

A continuación se va a mostrar cómo funciona la base de datos ahora, para que, a partir de ahora se entienda todo cuando hable de guardar datos, etc.

Bueno, llegamos aquí con la pulsera conectada y vinculada correctamente. Más tarde seguiré comentando cómo intercambiamos información con la pulsera. Mientras, vamos a extendernos en las bases de datos.

SQLite es una base de datos perfecta para las necesidades de la aplicación. A continuación explicaré las ventajas de SQLite [8]:

- **Tamaño:** SQLite tiene una pequeña memoria y una única biblioteca es necesaria para acceder a bases de datos, lo que lo hace ideal para aplicaciones de bases de datos incorporadas.

- **Rendimiento:** SQLite realiza operaciones de manera eficiente y es más rápido que MySQL y PostgreSQL. Como curiosidad, al volcar datos desde la pulsera a la aplicación, se suelen pasar alrededor de 1500 registros por día, entonces, que el rendimiento sea bueno es favorable, ya que si fuera peor, el volcado de datos tardaría mucho más. Más adelante, lo explicaré con más detalle.

- **Portabilidad:** se ejecuta en muchas plataformas y sus bases de datos pueden ser fácilmente portadas sin ninguna configuración o administración. En iOS también utilizo SQLite como base de datos.

- **Estabilidad:** SQLite es compatible con ACID, reunión de los cuatro criterios de Atomicidad, Consistencia, Aislamiento y Durabilidad.

- **Precio:** SQLite es de dominio público y, por tanto, es libre de utilizar para cualquier propósito sin costo y se puede redistribuir libremente.

Con todas estas ventajas, es difícil no decantarse por SQLite. Ahora que ya sabemos qué base de datos tenemos, vamos a mostrar cómo son sus tablas, sin embargo, las consultas se explicarán a medida que se vayan utilizando.

Empezamos por la primera tabla:

ACTIVITYDATA
id_activity
type
intensity
hr
steps
datetime

Esta tabla sirve para guardar toda la actividad de cada minuto en que has llevado la pulsera. El *id_activity* es un entero con autoincremento, *intensity* es la media de la intensidad del minuto, *type* es el tipo de actividad, *hr* es el pulso cardíaco si se ha tomado en ese minuto, steps son la suma de los pasos en ese minuto y *datetime* es la fecha guardada con el formato '*yyyyMMddHHmm*'. En iOS, esta tabla se ha guardado exactamente igual que en Android.

WIRSTHRDATA
id_hr
pulse
datetime

Esta tabla sirve para guardar todos los pulsos. El *id_hr* es un entero con autoincremento, *pulse* es el pulso cardíaco y *datetime* es la fecha guardada con el formato '*yyyyMMddHHmm*'. En iOS, esta tabla se ha guardado exactamente igual que en Android.

ROUTE
id_route
distance
time
name_orig
name_dest
datetime

Esta tabla sirve para guardar toda la actividad de cada minuto en que has llevado la pulsera. El *id_route* es un entero con autoincremento, *distance* es la distancia del trayecto, *time* es el tiempo que se ha tardado en realizar el trayecto, *name_orig* es el nombre del origen, *name_dest* es el nombre del destino y *datetime* es la fecha guardada con el formato '*yyyyMMddHHmm*'. En iOS, esta tabla se ha guardado exactamente igual que en Android.

DIRECTION
id_direction
road
distance
time
arrow
lat
lon

id_route

Esta tabla sirve para guardar todas las direcciones de cada ruta. El *id_direction* es un entero con autoincremento, *road* es el nombre de la calle, *distance* es la distancia que tienes que recorrer hasta tomar la siguiente dirección, *time* es el tiempo que queda hasta la nueva dirección, *arrow* es la flecha de la dirección a tomar, *lat* es la latitud y *lon* la longitud. Para saber en qué ruta está esta dirección, he creado la relación entre ROUTE y DIRECTION con id_route en DIRECTION, ya que una ruta o trayecto puede tener varias direcciones pero una dirección sólo pertenece a una ruta o trayecto. En iOS, esta tabla no ha sido necesario crear, porque cada vez que accedes a una ruta o trayecto guardado, toda la información necesaria se obtiene de la API de MapKit, que es la librería que Swift utiliza en todo lo relativo a los mapas. En Android, no puede ser así porque Google cobra por cada petición, pero este tema lo dejaremos para más adelante.

USERSETTINGS
visual
vibrate
destiny
time_arrival
distance
direction
road
total_distance
touch

Esta tabla sirve para guardar toda la configuración que el usuario necesita. Todas las variables son booleanas. En el caso de visual y vibrate, no pueden ser true a la vez, ya que si eliges que se muestre por pantalla (‘visual’ a true), ‘vibrate’ debe ser false, porque no se puede mostrar por pantalla y vibrar según la dirección a tomar, porque al mostrar por pantalla, la pulsera envía una vibración para avisar que hay una alerta en pantalla, entonces entrarían en conflicto las dos vibraciones. Las siguientes variables son para saber qué opciones necesita el usuario que se muestren por pantalla, pero esto se ampliará más adelante.

Bien pues, la base de datos ya ha sido explicada. En el siguiente capítulo, se va a explicar cómo son los instantes siguientes a la autenticación.

6.3.5 Después de la autenticación…

Una vez que ha finalizado la autenticación, la pulsera empezará a mandarnos información. Al principio leemos, el hardware en la característica *0x2A27*, el software en la característica

0x2A28, a partir de aquí vamos a centrarnos en las características de la información de la batería, los pasos, las calorías y la distancia recorrida.

6.3.6 Información de la batería

La información de la batería nos informa de todo lo relativo a esta y lo recibiremos, como anteriormente se ha explicado, por la característica cuyo UUID es "*00000006-0000-3512-2118-0009af100700*". Lo voy a explicar con un ejemplo, porque así se entenderá mejor.

Recibimos lo siguiente:

0F-4F-00-E3-07-04-0E-10-27-11-0C-E3-07-04-0D-13-19-16-0C-64

Donde **0F** es algún tipo de cabecera.

Donde **4F**, significa que la batería tiene un 79% de carga en el momento de recibir esta información.

Los siguientes **00**, significan que se está utilizando la batería, si estuviera cargando, sería 01 en vez de 00.

Los siguientes **E3-07-04-0E-10-27-11** se refieren a la fecha actual, por un lado E3-07, se deben cambiar las posiciones de manera que quede 07E3, refiriéndose en hexadecimal al año 2019. Después, 04 es el cuarto mes y 0E, el día 14. Lo siguiente es la hora que en este caso es 10-27-11, las 16:39 y 17 segundos.

El siguiente es **0C**, cuyo valor se refieren al número de cargas o ciclos, que en este caso son 12.

Los siguientes **E3-07-04-0D-13-19-16** se refieren a la fecha de la última carga de la batería, por un lado E3-07, se deben cambiar las posiciones de manera que quede 07E3, refiriéndose en hexadecimal al año 2019. Después, 04 es el cuarto mes y 0D, el día 13. Lo siguiente es la hora que en este caso es 13-19-16, las 19:25 y 22 segundos.

El siguiente es **0C**, cuyo valor se refieren al número de cargas o ciclos, que en este caso son 12.

El siguiente es **64**, cuyo valor se refieren a cuánto se cargó la última vez, es decir, hasta qué porcentaje llegó la última vez que se cargó. En este caso, se cargó al 100%.

6.3.7 Pasos, calorías y distancia

La información de los pasos, la distancia y las calorías nos informa de todo lo relativo a la actividad que hemos realizado en el mismo día. A partir de las 00.00 horas de cada día, el contador de los tres valores se inicializan a 0. Toda esta información la recibiremos por la característica cuyo UUID es "*00000007-0000-3512-2118-0009af100700*".

Al igual que hemos hecho con la información de la batería, vamos a hacer lo mismo con los pasos, distancia y calorías, explicar cómo funciona con la ayuda de un ejemplo.

Recibimos:

0C 4C 00 00 00 1F 00 00 00 01 00 00 00

Donde **OC**, es un tipo de cabecera.

Los siguientes 4 bytes, **4C-00-00-00**, se refieren a los pasos. En este caso, significa que tiene contados 76 pasos. Destacar que el LSB (*Less Significant Bit*) es en este caso el **4C**. En los siguientes casos ocurre de la misma forma.

Los siguientes **1F-00-00-00**, significa que ha recorrido 31 metros.Y, para finalizar, los últimos 4 bytes, **01-00-00-00**, significa que se han quemado 1 caloría.

El procedimiento para calcular la distancia a partir de los pasos, así como el cálculo de las calorías a partir de los pasos, no se puede determinar, ya que al ser privativa la pulsera no se puede acceder al algoritmo del cálculo. También destacar, que dependiendo el tipo de actividad, las distancias y las calorías quemadas pueden variar siendo los mismos pasos.

6.3.8 Pulso cardíaco

Para conseguir la pulsación cardíaca se tiene que pedir a la pulsera mediante un procedimiento muy sencillo.

En primer lugar, dejar bien claro que existen dos características con funciones distintas en la medición del pulso. Está la característica *0x2A39*, siendo su UUID largo: "*00002a39-0000-1000-8000-00805f9b34fb*". Bien pues, esta característica se dedica a recibir las órdenes de control. Por ejemplo, para sólo realizar una medición, se envía a esta característica los siguientes valores: *[0x21, 0x02, 0x01]*.

Si se necesita realizar una medición del pulso continuada, sería diferente. Con unos cuantos intentos se encontraría la solución.

Una vez se ha enviado los 3 bytes anteriores, la pulsera enviará cuando acabe la monitorización del pulso los datos de la medición mediante la característica *0x2A37*. Este es el primer valor que ya se puede guardar en la base de datos, por tanto, vamos a explicar el proceso a seguir para registrarlo.

Como la tabla que va a almacenar el pulso cardíaco se llama WIRSTHRDATA, hemos creado una función en BTMiBandHelper llamada saveWirstHRData() y como parámetro se le pasa el pulso. Desde aquí se mandará a otra función llamada extractWirstHRData() donde se extrae del array de bytes el valor del pulso y se crea un objeto llamado WirstHRData que contiene el valor del pulso y la fecha con el siguiente formato "*yyyyMMddkkmm*".

Una vez tenemos el objeto WirstHRData listo, procedemos a enviárselo al WirstDataDBHelper. WirstDataDBHelper es un objeto encargado de realizar todas las funciones de inserción, borrado y consulta. Con WirstDataDBHelper se puede acceder a la función insertWirstHRData() para que abra la base de datos y habilite la escritura con el método **.getWritableDatabase()**, inserte el objeto en la base de datos con **.insert()** y cierre la base de datos con **.close()**.

En iOS, sólo cambia en la manera de insertar, que no tiene una función concreta sino que se crea la query, que en este caso es así:

"INSERT INTO WIRSTHRDATA (pulse,date) VALUES (\(pulse),?)"

Con el carácter "?", lo que se pretende decir es que se añade después. La función para añadir el parámetro es sqlite3_bind_text().

Más adelante, ya se explicará cómo consultar los pulsos y cómo añadirlos a una gráfica. Eso se explicará en el apartado Pulso que está en la vista Mi Pulsera.

6.3.9 Botón

Igual que en el pulso cardíaco, para tener una notificación del botón en este caso, hace falta pulsarlo. Para ello se debe Ir al menú de la pulsera y aparecerá un apartado que se llama *Más*, pues dos posiciones a la derecha y estaremos ante *Buscar dispositivo*. En el momento que mantengamos el botón pulsado, la pulsera enviará una notificación con la característica "*00000010-0000-3512-2118-0009af100700"*. No importa el valor que envíe, lo importante es que se está pulsando el botón y así lo podemos saber.

La manera de acceder a esa "*pulsación*" se convierte en algo engorrosa, pero la solución que se ha intentado de la manera que voy a explicar a continuación no es viable, ya que se consume rápidamente la batería de los dos dispositivos.

Existe un parámetro en todos los dispositivos bluetooth que sirve para controlar el tiempo de respuesta en ms. Bien pues, lo que hice es crear un bucle donde continuamente preguntaba a la pulsera algo parecido a un *¿estás ahí?... 120 ms… "Sí, aquí estoy"*. Bien pues, en el momento en que ese "*Sí, aquí estoy"* tardara más de 200 ms, por ejemplo, significaba que el usuario había pulsado el botón. A veces funcionaba bien, a veces fallaba, pero al final, la batería se consumía muy rápido. Al no ser tan imprescindible el botón, esta solución la descarté de inmediato, ya que la otra solución era más tediosa pero no requería tanto esfuerzo energético.

6.3.10 Datos de actividad

Este es el punto donde más me voy a extender, ya que es el principal encargado de guardar información sobre toda la actividad de la pulsera y que se quede registrado correctamente sin repetición y ordenado cronológicamente.

Empezamos por cómo se realiza el volcado de datos y luego cómo se organiza y cómo se guarda, porque hay aspectos que hay que tener en cuenta.

El proceso de descargar todos los registros de las actividades se realiza inmediatamente después de haberse completado el proceso de autenticación. Esto es, una vez autenticado se procede a llamar a **startActivityData()**, esta función se encarga de enviar a la característica "*00000004-0000-3512-2118-0009af100700*" la última fecha que se guardó en la base de datos, o si, por el contrario, no hay última fecha, crearla. Si hay que crearla, la fecha que se debe guardar es la de un día antes de la actual.

Antes de seguir, he de hacer una aclaración. Hay dos características que se dedican a los datos de las actividades. El primero es la característica "*00000004-0000-3512-2118-0009af100700*" que se encarga del control de las fechas de las actividades, etc. y el segundo es la característica

cuyo UUID es "*00000005-0000-3512-2118-0009af100700*" que se encarga de enviar todos los datos de actividades desde la fecha mandada a la característica anterior a ésta.

Bien, pues entonces, con la última fecha conseguida con **getLastDateString()**, que se consigue de las preferencias de usuario, se le da un formato conveniente y se añade a lo siguiente:

"[0x01, 0x01, 0x227, última fecha, 0x0, 0x8]"

En total, son 10 bytes los que se mandan, siendo 5 los bytes de la fecha: año, mes, día, hora y minutos. Cabe recordar, como se comentó anteriormente, que la actividad se registra por minutos.

El primer paso que es el envío de la última fecha guardada ya está realizado. El segundo y tercero son habilitar las notificaciones tanto para la característica "*00000004*" como para la característica "*00000005*", ya que si no se habilitan, no recibiremos ninguna información.

El cuarto y último paso ha sido el más costoso de encontrar, he leído en cientos de foros hasta que he encontrado la solución. Si no se manda lo que a continuación voy a explicar, nada funciona. Una vez están realizados los pasos anteriores, se procede a enviar el byte que hace que todo funcione. Este byte es *0x02* y se manda a la característica "*00000004*".

A partir de ahí, se recibe de la pulsera una cabecera que contiene el éxito de la petición, seguido de la última fecha recibida. Una vez recibido esto, ya empieza a enviar toda la información en arrays.

Los arrays pueden contener diferentes longitudes de 5, 9, 13 y 17.

Vamos a poner un ejemplo para que se entienda mejor. Imagínese una persona sentada escribiendo en el ordenador como ahora mismo lo estoy haciendo yo durante 2 minutos. Lo recibido sería tal que así:

"[0, 80, 12, 0, -1, 80, 16, 0, -1]"

Esto es a modo de ejemplo, pero podría ser perfectamente válido. El primer 0, se refiere al número de array, en este caso, como sólo son dos minutos, sólo necesita un array. Como máximo, sólo 4 minutos por array, por eso la longitud máxima es de 17. El 80 se refiere al tipo de actividad, estar sentado. El 12 es la intensidad de la actividad. El siguiente 0 son los pasos realizados en ese minuto. Y el -1, es el valor null al pulso cardíaco, es decir, que no hay ningún valor porque no se ha tomado el pulso en ese minuto.

Una vez recibimos estos dos minutos, se llama a la función **parseActivityData()**, **arrayOrganisator()** y, por último, **saveActivityData()**.

Y, para que quede más claro, mostraré cómo se guardarían estos dos minutos después de ejecutar las tres funciones anteriores:

"[id_activity 80 12 0 -1 201905311007]"
"[id_activity 80 16 0 -1 201905311008]"

Y la última fecha guardada en las preferencias del usuario sería: 201905311008. Entonces, la siguiente ejecución de Mi Maps, la pulsera enviaría a partir de la fecha anterior.

6.3.11 Mi Pulsera

A continuación voy a explicar la primera vista que nos encontramos cuando ejecutamos la aplicación y acaba la sincronización de los datos. A partir de ahora, llamaré vista a las distintas pantallas que el usuario tendrá a su disposición, pero en realidad, tanto Mi Pulsera, como Explorar y Ajustes, se llaman *Fragments*.

Los datos de la característica "*00000007*", que contenía los pasos, calorías y distancia, se muestran al principio. Los pasos se muestran en un TextView, pero también en un gráfico redondo, gracias a la librería llamada **FitChart()**. El relleno del gráfico redondo es porcentual, y para que sea éste porcentual debe haber un máximo. En este caso, como máximo, he decidido escoger los 10.000 pasos, ya que son los pasos que la OMS (Organización Mundial de la Salud) recomienda realizar todos los días.

Figura 36. Fragment Mi Pulsera

6.3.12 Actividad

Lo siguiente que el usuario se encontrará es un apartado dedicado a la actividad guardada en la base de datos donde puedes revisar las actividades realizadas en días anteriores. Esta vista muestra un gráfico de barras y más abajo, un ListView donde se muestra la lista de actividades realizadas. Para poder implementar el gráfico, se ha necesitado una librería llamada **Charts** de MikePhil, y elegí esta librería porque es de las pocas librerías que existen tanto para Android como para iOS.

Figura 36. Actividad

La implementación del gráfico fue fácil, ya que en el eje X, siempre necesita 1440 elementos, ya que son los minutos de un día. Entonces se crea un array de 1440 elementos y lo inicio a 0. Y, finalmente, añado los pasos registrados en cada minuto.

Y para el ListView hay que agrupar las actividades cuyo tipo de actividad sea el mismo y los minutos adyacentes. Entonces se clasificaban según el tipo de actividad acorde a los siguientes valores:

1: Caminando

16, 17, 26: Caminando lento

80: Sentado

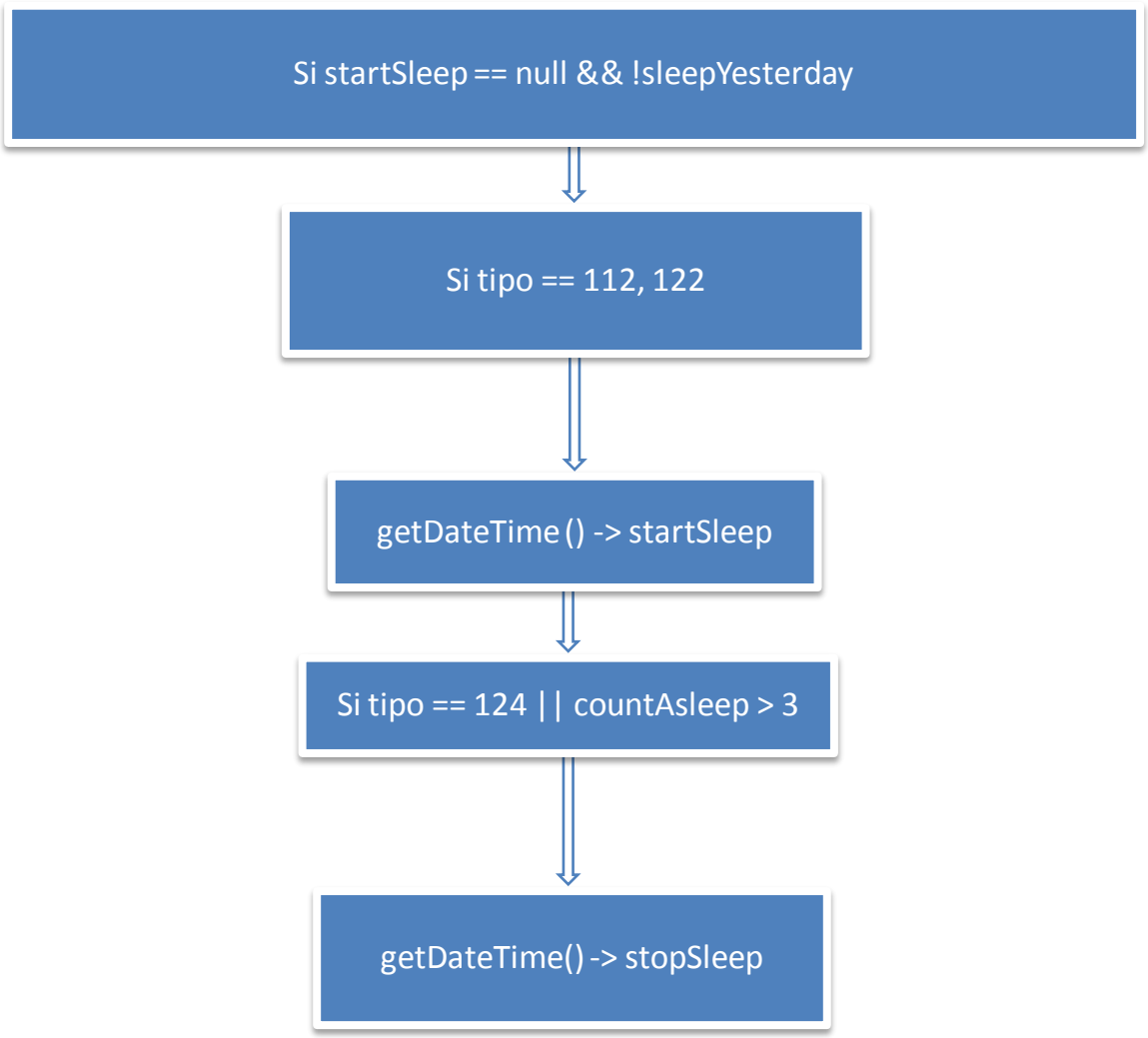
96, 90: Sentado con movimientos

98, 82, 50, 66: Corriendo

Si se trata de un tipo de actividad diferente a lo descrito anteriormente, se clasifica como acción no reconocida.

6.3.13 Sueño

La siguiente vista que vamos a analizar es la del sueño. En definitiva, la vista es muy parecida a la de actividad pero vamos a indagar sobre el algoritmo que he utilizado para extraer las horas de sueño, así como el sueño profundo y el sueño ligero.



El gráfico anterior resume a grandes rasgos cómo se obtiene la información del sueño. A continuación, voy a explicar con todo detalle cómo obtener también los datos de sueño ligero y sueño profundo.

Empezamos con recorrer un array con todos los registros desde una función de WirstDataDBHelper llamada **getStepsActivityEveryTenByDay()** donde pasamos como parámetro el día que queremos consultar con el formato '*yyyyMMdd*' y nos devuelve un array con todos los registros de la actividad de la pulsera en ese día.

Recorremos el array y en cada registro se comprueba si el tipo de actividad es 112 o 122, que significa que el usuario se acuesta. Si en los 10 primeros registros, es decir, desde las 00.00 horas hasta las 00.10 horas, existen 3 registros seguidos con intensidad 0, activamos una variable booleana llamada *sleepYesterday* y esto significa que el usuario ha empezado a dormir antes de las 00.00 horas.

En este caso, volvemos a consultar la base de datos para que nos proporcione los datos de la actividad del día anterior pero en vez de recorrer el array desde las 00.00 horas hasta las 23.59 horas, lo recorre de manera descendente, es decir, desde las 23.59 horas hasta las 00.00 horas. Esta búsqueda es la misma que la anterior con diferencias ínfimas.

Como a veces no hay actividad cuyo tipo sea 112 o 122, debemos tener un contador por si existe una actividad con intensidad 0 cuya duración sea más de 5 minutos. En este caso, escogeremos esta hora como la hora de dormir. He llegado a la conclusión que son 5 minutos al comparar la hora de acostarse con la de la aplicación oficial, Mi Fit. Aunque no es exactamente la misma, es muy parecida.

Una vez tenemos la hora de acostarse o **startSleep**, vamos a buscar la de levantarse, **stopSleep**.

Para consultar un array donde sólo esté la hora de levantar se crea una función llamada **getNextSleepActivityData()**, donde se le pasa como parámetro la hora de acostarse, así pues, devuelve la actividad a partir de la hora de acostarse con 1440 registros que significa que son 24 horas. A partir del registro 180, se empieza a buscar la hora de levantarse. ¿ Por qué 180 y no 200? Porque he llegado a esta conclusión después de probar y probar con mis horas de sueño. 180 se refiere a que a partir de tres horas o 180 minutos, se considera que has dormido. Al principio, tenía programado para que a la hora ya buscara la hora de levantarse. No era factible, ya que una siesta de más de una hora la registraba como descanso normal.

Figura 37. Sueño

Una vez rebasados los 180, comprueba si hay pasos, es decir, está despierto; si el tipo de actividad es de 124, 122, 121, 123 o 112, quiere decir que el usuario está levantado; si la intensidad es superior a 50, en los casos de sueño, ninguna intensidad rebasa los 50. Cuando alguna de las condiciones anteriores se confirma, la hora en cuestión pasa a ser el llamado **stopSleepCandidate**, y si los siguientes registros, a partir de 5 registros, tienen pasos, intensidad mayor de 50; en definitiva, movimiento, el candidato pasa a ser la hora de levantarse.

Una vez hemos detectado startSleep y stopSleep, queda mediante una resta averiguar cuántas horas se ha dormido y cuánto tiempo ha invertido en sueño profundo y cuánto tiempo ha invertido en sueño ligero.

Para saber cuánto tiempo de sueño profundo ha dormido el usuario, he creado una manera eficaz, en comparación con la aplicación oficial Mi Fit. Esto es sencillo, a continuación se explica.

Si los cuatro registros anteriores y los cuatro registros posteriores al actual y, por supuesto, el actual, tienen intensidades igual a 0, podremos asegurar que el registro actual se guarde como

minuto con sueño profundo. Todo lo que sea diferente a lo anterior, se guarda como sueño ligero.

Para afianzar que lo que se muestra por pantalla relativo al sueño sea veraz, he optado por añadir otra restricción. Ésta es, que si por alguna razón, el sueño profundo es inferior a 30 minutos, todo lo calculado anteriormente no valga y no se muestre ni gráficos ni ningún TextView.

Bueno, pues ya está explicado lo más importante relativo al sueño. Destacar que para obtener los mismos resultados en iOS he tenido que modificar algunos tipos de variables, ya que en los condicionales no funcionaban, pero aparte de este mínimo problema, nada más.

6.3.14 Pulso cardíaco

Para conseguir la información de los pulsos medidos, simplemente se consulta en WirstDataDBHelper en una función llamada **getHRDataByDay()** donde se le pasa como parámetro el día y se recibe un array con todas las mediciones de pulso de ese día en concreto.

Después, en la ListView se muestra la hora de la medición, así como el valor del pulso y se muestran las mediciones en el gráfico de línea.

En iOS, no he tenido ningún problema debido a la sencillez de esta vista.

En el siguiente capítulo, voy a explicar detalladamente la siguiente vista que se va a encontrar el usuario al abrir la aplicación Mi Maps: Explorar. Aquí mostraré cómo funciona el envío de las direcciones a la pulsera y todos los problemas que he tenido que ir esquivando para que funcione todo sin ningún problema.

Figura 38. Pulso cardíaco

6.3.15 Explorar

En esta parte del proyecto, sí que hay una diferencia bastante grande entre Android y iOS. En un principio, pretendía seguir el mismo patrón en las dos plataformas, pero por un problema, que más tarde explicaré, no pude acabarlo como a mí me hubiese gustado.

Empiezo con iOS que no tiene mucha complicación y, a continuación abordaré la parte de Android, donde me explayaré un poco más.

Swift brinda a tu disposición una librería llamada MapKit, desde donde puedes obtener toda la información relativa a los mapas. Desde rutas hasta distancias, etc. Con esto, el método que he seguido es muy fácil.

Figura 39. Explorar

Primero, he añadido una barra de búsqueda cuyo nombre es **UISearchBar** implementada en otro objeto llamado **MKLocalSearchCompleter()**.

¿Qué hace MKLocalSearchCompleter?

Bueno, pues, como su nombre indica, la función principal de este objeto es completar los nombres basándose en la localización. Por ejemplo, si el usuario se encuentra cerca de Valencia, al buscar "*va*", el objeto MKLocalSearchCompleter le devolverá "*Valencia*". Esto evita que el usuario tenga que introducir el nombre completo de la dirección donde desea ir.

Una vez ya tenemos la dirección donde el usuario quiere ir, creamos la ruta. En resumen, la ruta se crea de la siguiente manera. El destino se convierte en un objeto llamado **MKPlacemark**, así como el origen también se convierte en el objeto anterior.

Cuando ya tenemos estos dos objetos, procedemos a crear otro objeto llamado **MKDirectionsRequest**, donde se recoge el origen, el destino y el tipo de transporte; y nos devuelve todo lo relativo a esta ruta.

Con la información que ha devuelto, nosotros ya podemos dibujar la ruta en el mapa mediante **mapkitView.add(route.polyline, level: .aboveRoads)**, y utilizar las instrucciones para enviarlas a la pulsera.

Esto es, una vez tenemos la primera instrucción, hay varios procesos que hay que pasar, ya que las instrucciones que los servidores de Apple nos envían no son intuitivas. El problema está en que, por ejemplo, una instrucción es un String como este: "Gire a la derecha hacia la calle X", acompañado de la distancia que hay que recorrer hasta realizar este giro y la distancia total restante al destino.

¿Qué problema hay en estas instrucciones? Pues que no ayudan al desarrollador a la hora de añadir las flechas, es decir, sería mejor mandar una variable llamada *arrow* o *maneuver*, por ejemplo, así pues, quedaría de la siguiente manera: "*maneuver:turn-right"*. De esta forma, el desarrollador no tendría ningún problema al añadir las flechas.

¿Cuál ha sido mi solución? Mi solución a este problema ha sido el de extraer toda la información posible de cada instrucción y con esa información, elegir una flecha. Por ejemplo, "*Tome una curva cerrada a la derecha",* pues la solución es que si la instrucción contiene la palabra *izquierda, curva* y *cerrada,* el icono a mostrar es *turnsharpleft* y la flecha a mostrar en la pulsera se conseguirá mandando tres bytes: *[226, 172, 139]*, más adelante, en la parte de Android, se explicará el proceso de añadir la flecha para el envío a la pulsera.

En el caso de las rotondas, tampoco era complicado, ya que te dice que tomes la primera, segunda, tercera, etc. salida.

Con el problema de la flecha solucionado, ya sólo queda el añadir lo que el usuario haya configurado en los ajustes y mediante la función **messageToTextNotification()**, mandarlo a la pulsera para que se muestre o vibre.

Una vez explicado el procedimiento para enviar las direcciones a la pulsera desde la aplicación de Mi Maps en iOS, vamos a proceder a explicar el método que he utilizado para conseguirlo en Android, ya que no ha sido tarea fácil.

Desde un primer momento, empecé a programar una clase llamada **HttpHandler** y otra llamada **JsonParser**. La primera trataba de conseguir desde la API (*Application Programming Interface*) de *Google Directions* toda la información relativa a la ruta y la segunda, de extraer esta información en formato *json* a un objeto donde fuera más fácil conseguir las variables.

Todo esto que ya tenía hecho, lo tuve que descartar.

¿*Cuál fue el problema*? No me imaginaba que Google cobraría por cada solicitud. El plan de facturación de Google es el siguiente: a principios de cada mes Google te permite gastar 200 dólares, pero si dentro de ese mes superas los 200 dólares, entonces es cuando empieza a cobrarte.

Directions API Google Maps
100.001 – 500.000
0.004 USD per each

Google cobra 0.004 USD por cada solicitud. No es mucho, pero si tenemos en cuenta que cada paso o cada 2 metros, la aplicación Mi Maps realiza un solicitud. También si el usuario cambia

la ruta, la aplicación manda una solicitud. De esta forma, esto es inviable, ya que cada ruta, como mínimo usará 10 solicitudes por cada usuario. Si hay 100 usuarios y estos 100 usuarios lo utilizan 10 días, ya son 10.000 solicitudes, que equivalen a 40 USD. Aunque no llegue a los 200 USD, no quiero arriesgarme, ya que la aplicación no tiene ningún coste y no quiero crear costes que se puedan evitar.

A continuación explico cómo realicé el *workaround* para solucionar el problema de los costes de los requests.

Para solucionar el problema creé una clase llamada **NotificationService()**. Esta clase se extiende con otra llamada **NotificationListenerService**, cuya función principal es la de conseguir todas las notificaciones push o locales que se muestran en el teléfono. Desde alertas porque la batería se está agotando hasta notificaciones push de mensajes de Whatsapp.

Con este servicio, ya se pueden conseguir las notificaciones de Google Maps. Con este servicio activo, se abre Google Maps y todas las direcciones que son mandadas desde los servidores de Google, son recogidas en la aplicación Mi Maps. Para extraer sólo las notificaciones que provienen de Google Maps se utiliza la condición siguiente:

"If pack.equals("com.google.android.apps.maps")"

Una vez nos hemos desprendido de todas las notificaciones menos las de Google Maps, tenemos la tarea ardua de extraer la información convenientemente, ya que se recibe todo desordenado así como también se recibe la flecha de la dirección como una imagen que hay que procesarla, pero este proceso se explicará más adelante.

La notificación de Google Maps que recibimos contiene tres elementos: title, subtitle y subtext. Con estos tres elementos, se pueden extraer los datos básicos para, más tarde, ordenarlos de manera que se muestren en la pulsera. Todos los datos son extraídos menos la flecha de dirección. La flecha de dirección se extrae de la notificación mediante **Notificacion.EXTRA_SMALL_ICON**.

El proceso para averiguar de qué flecha se trata es de la siguiente manera:

Extraes la imagen en formato *bitmap* y la conviertes en *String*. Se tienen que extraer todas las imágenes que existan y guardarlas en una especie de biblioteca creada por un array de objetos llamados **ArrowBytesPair**.

Una vez se ha creado esta "biblioteca", cada vez que recibamos esta imagen en String, la mandamos a una función llamada **getArrow()** y recibiremos una respuesta tal que así: "*turnsharpright*".

Ya tenemos todo lo necesario para saber qué flecha debemos enviar a la pulsera. Entonces, ahora hace falta un "traductor" que convierta "*turnsharpright"* en un lenguaje entendible por la pulsera para que ésta muestre la flecha en la pantalla.

Para construir este "traductor" tuve que buscar información en Internet para poder crear mi propia tabla de codificación. Una página que me ayudó mucho fue *www.utf8-chartable.de*. En esta página encontré todos los símbolos necesarios para todas las rutas y sus respectivos códigos en hexadecimal.

Entonces, vamos a explicarlo con el anterior ejemplo:

Turnleft: la dirección a tomar. No sé qué tengo que mandar a la pulsera para que me muestre un símbolo que signifique esa palabra.

Bueno, pues buscamos una flecha parecida a la siguiente:

←

Ya la he encontrado en la página anteriormente descrita cuyo código son tres bytes compuestos por:

[226, 172, 133]

Como ya tenemos el código que tenemos que enviar a la pulsera, sólo nos queda añadir este array de bytes al array del mensaje que se va a mandar a la pulsera.

Pues así es cómo se envía a la pulsera una flecha de dirección. Para poder mostrar todas se tienen que guardar todos los códigos de todas las flechas de dirección posibles. Todas tienen una longitud de 3 bytes, pero el problema es que para las rotondas utilizo 6 bytes, es decir, 3 bytes para el símbolo de la rotonda y los otros bytes para la dirección a tomar. Hay casos en que no puede ser muy preciso porque el símbolo muestra que es en la rotonda a la izquierda y en realidad es en la rotonda ligeramente a la izquierda por ejemplo. Pero en estos casos, la distancia restante al cambio de dirección es sumamente importante.

Destacar que como la aplicación Google Maps envía la misma notificación cada 5 segundos, en Mi Maps se tiene que comprobar que si no cambia ningún parámetro de esta información, como por ejemplo la distancia restante, ésta no se debe mostrar, porque entonces siempre estaría mandando notificaciones a la pulsera. En el caso de las vibraciones, la pulsera sólo recibirá las vibraciones cuando la dirección a tomar esté a 10 o 20 metros.

6.3.16 Ajustes

Figura 40. Ajustes 1/2

En este capítulo vamos a profundizar más en la plataforma Android, ya que tiene más configuraciones que en iOS. Todo es igual en ambas plataformas menos la parte del funcionamiento de Google Maps.

Al principio de esta vista llamada Ajustes está la pulsera sincronizada, desde donde puedes eliminarla. Después está un apartado llamado "Comprobación del funcionamiento", donde existen dos apartados: el primero sirve para configurar las notificaciones de Google Maps con el teléfono, para que así, Mi Maps pueda conseguir las indicaciones que Google Maps manda y se abre Google Maps desde una función llamada: **Settings.ACTION_APPLICATION_DETAILS_SETTINGS**; y el segundo sirve para activar las notificaciones de Mi Maps en el teléfono abriendo la configuración de notificaciones de Android mediante la siguiente llamada: **android.settings.ACTION_NOTIFICATION_LISTENER_SETTINGS**.

Figura 41. Cuadro de diálogo

En la figura de la izquierda se puede observar un cuadro de diálogo que aparece al comprobar la conexión con la pulsera, en este caso, con notificaciones, pero además se puede comprobar si funciona en modo vibración. En el caso de que no funcionase, quiere decir que hay algún problema en la conexión, se debería eliminar la pulsera desde Ajustes y volverla a vincular.

¿Por qué son necesarias las notificaciones en Mi Maps?

Las notificaciones en Mi Maps son necesarias porque al abrir la aplicación se crea un servicio llamado BTService, que fue explicado en su momento y también NotificationService, para conseguir todas las notificaciones del teléfono. Estos servicios se ejecutan en segundo plano. Entonces se necesita una notificación para saber si el servicio está en ejecución o no.

A continuación está el Modo de guía. Aquí se configura el modo en que el usuario va a recibir las instrucciones: en modo vibración o en modo visual. Siguiendo los modos, se encuentra la información para la Guía por vibraciones. Esto es, una especie de diccionario donde están todas las equivalencias de direcciones en formato vibración. Así pues, para utilizar el modo vibración, primero debes aprenderte qué significan todos los tipos de notificaciones.

Figura 42. Diccionario de vibraciones

Lo más importante que hay que saber para entender cómo funcionan las vibraciones es lo siguiente:

"El número de vibraciones indica la dirección a tomar: una, recto; dos, derecha; y tres, izquierda.

La duración de las direcciones indican si ésta es brusca o suave, es decir, si la duración de las vibraciones es muy corta, la brusquedad del cambio será pequeña."

Una vez hemos repasado el diccionario de vibraciones, pasamos a los ajustes de la Guía visual. En estos ajustes se pueden configurar si quieres que se muestre el destino, así como la hora de

llegada, la dirección, el nombre de calle, la distancia restante para tomar la dirección indicada, y por último, la distancia restante al destino.

Figura 43. Información para mostrar en Guía Visual

En el siguiente apartado se puede elegir la capacidad de uso del botón de la pulsera. Esta opción está en fase de desarrollo. Está pensado para añadir más funcionalidades en un futuro. En el siguiente capítulo se hablará sobre el futuro de Mi Maps.

Más abajo están las opciones de borrado, con la primera sólo se borrarán los trayectos del historial y en la segunda se borrará absolutamente todo.

Figura 44. Botón y borrado

6.4 Funcionamiento de la aplicación

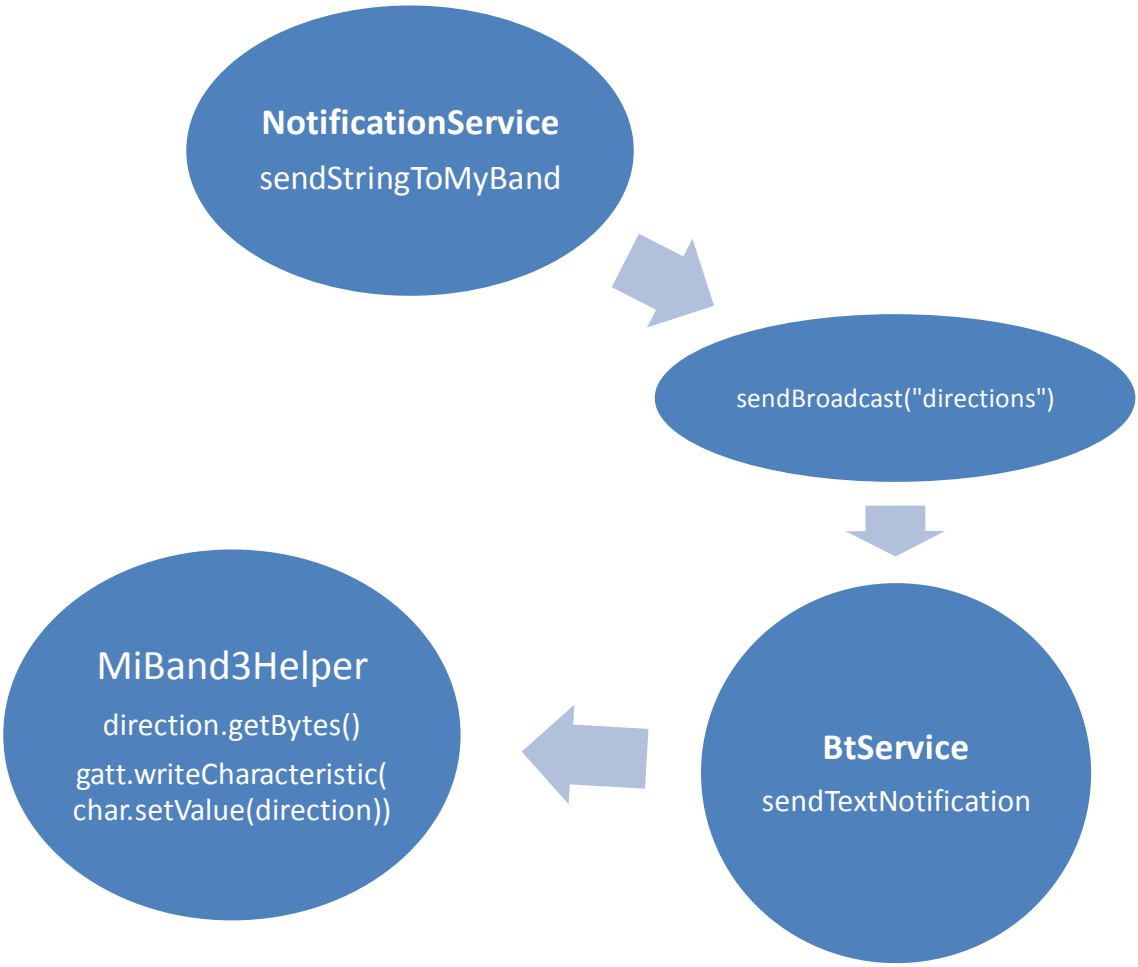
Mi Maps es bastante sencilla de utilizar, para ello vamos a explicar los pasos para utilizarla, estos pasos se describen a continuación:

- Abre la aplicación
- Se busca el dispositivo Bluetooth
- Se autentifica el dispositivo
- Ir a la pantalla Ajustes y comprobar si todas las configuraciones del teléfono móvil y de Google Maps son las correctas.
- Ir a la pantalla Explorar y pulsar en el botón Empezar.

- Al abrirse Google Maps, añadir la dirección y esperar a que la pulsera reciba la primera notificación.
    - Para iOS, directamente añadir la dirección y pulsar el botón de “Ir”.

Notificación de Guía Visual

6.5 Procesos internos al mandar direcciones a Mi Band 3



Al abrir Google Maps e introducir una dirección, si está bien configurado tanto la aplicación Mi Maps como la aplicación Google Maps, el servicio ejecutado en segundo plano NotificationService estará recibiendo todos los mensajes de Google Maps. Entonces, si es una información que no se repite, procedemos a mandarla a **sendStringToMyBand**.

Desde **sendStringToMyBand** se mandará el string de la imagen bitmap de la flecha de dirección a una clase llamada getNameArrowImg que se encargará de cotejar el string con toda la "biblioteca" de todos los strings de imágenes de flechas. Una vez mandada el string, se devolverá el nombre de la dirección tipo "*turnleft"* y se guardará con los otros parámetros en saveDirection, o si es la primera dirección, también se guardará en saveRoute, creando así un trayecto nuevo.

Con toda esta información guardada en la base de datos, se procederá a enviarla a **sendBroadcast(**"directions"**)**, como ha sido comentado antes, al ejecutar sendBroadcast, esta información se enviará a todos los broadcastIntent que hayan sido creados y tengan "directions" como variable de interés. Como BTService tendrá "directions" como variable de interés, éste recibirá toda la información y la mandará a una función llamada **sendTextNotification** que pertenece al objeto que maneja la pulsera Mi Band 3: *bthelper*.

Una vez sea mandada la información a la función sendTextNotification, ésta será ordenada según la configuración que haya elegido el usuario y será "empaquetada" en una única trama de bytes, para eso se utilizará la función **getBytes**, encargada de convertir los strings en array de bytes, y, una vez hecho esto, se escribirá en la característica *0x2A46* con la función, **char.setValue(**array de bytes**)**, para, más tarde, enviarla a **bluetoothGatt.writeCharacteristic(**char**)**. Con esto, la pulsera recibirá la información enviada y acabará el proceso de envío de dirección.

Cuando vuelva a cambiar alguna información enviada desde Google Maps, se volverá a repetir el mismo proceso que se acaba de explicar.

6.6 Futuro

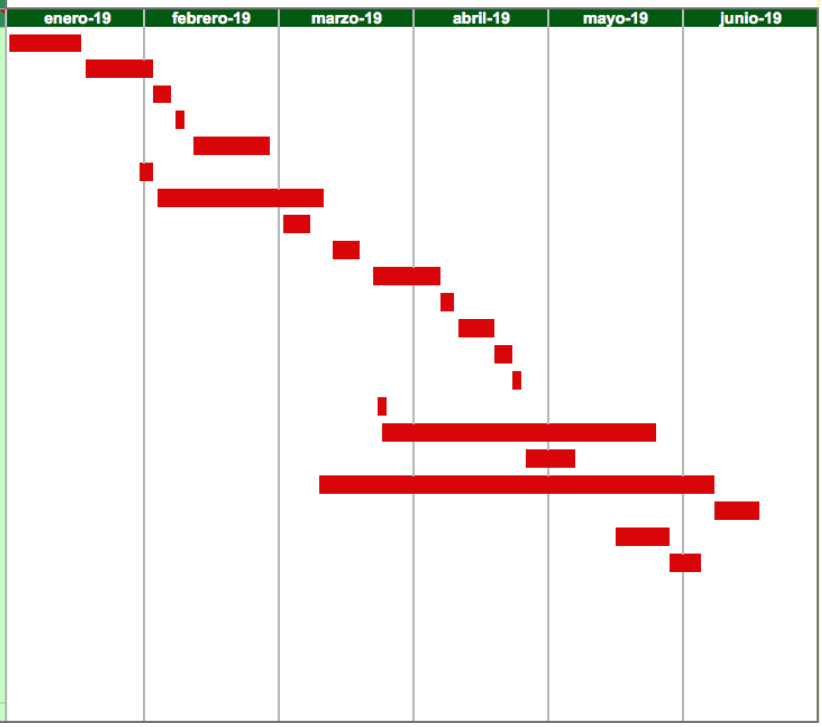
El futuro de Mi Maps pasa por la intención de crear una aplicación con más funcionalidades que las que tiene ahora.

En la actualidad, la comunidad tecnológica está a la espera para poder probar el funcionamiento de la nueva Mi Band 4. Esto abrirá muchas posibilidades a Mi Maps.

Hay que ser sinceros. Esto es un proyecto que requiere muchísimo esfuerzo. Y Mi Maps ya ha nacido, pero para poder desarrollarse como se merece, necesitaría más personas involucradas. Como ejemplo está Mi Fit, tienen desarrolladores para Android y desarrolladores para iOS, a parte de un equipo de diseño. Todo esto en Mi Maps lo ha creado una sola persona y espero que en un futuro haya más personas involucradas.

Capítulo 7. Planificación del proyecto

Mi Maps					
Etapa	Descripción	Inicio	Fin	Días N	Días L
Investigaciones	Investigaciones previas	5-1-19	25-1-19	21	15
Android	Conexión y Mi Pulsera	27-1-19	10-2-19	15	10
Android	Sueño	11-2-19	14-2-19	4	4
Android	Actividad y Pulso	16-2-19	17-2-19	2	
Android	Explorar	20-2-19	8-3-19	17	13
Android	Creación de bases de datos	8-2-19	10-2-19	3	1
Android	Creación de todas las consultas	12-2-19	20-3-19	37	27
Android	Ajustes	12-3-19	17-3-19	6	4
iOS	Investigación Bluetooth Swift	23-3-19	28-3-19	6	4
iOS	Conexión y Mi Pulsera	1-4-19	15-4-19	15	11
iOS	Actividad	16-4-19	18-4-19	3	3
iOS	Explorar	20-4-19	27-4-19	8	5
iOS	Sueño	28-4-19	1-5-19	4	3
iOS	Pulso	2-5-19	3-5-19	2	2
iOS	Creación de bases de datos	2-4-19	3-4-19	2	2
iOS	Creación de consultas	3-4-19	2-6-19	61	43
iOS	Ajustes	5-5-19	15-5-19	11	8
iOS y Android	Pruebas preAlfa	20-3-19	15-6-19	88	63
iOS y Android	Pruebas Alfa	16-6-19	25-6-19	10	7
Memoria	Borrador Memoria TFM	25-5-19	5-6-19	12	8
Memoria	Memoria TFM	6-6-19	12-6-19	7	5
			Total	334	238



El primer paso que tuve que hacer fue buscar información, este proceso empezó alrededor del 5 de enero de 2019 y finalizó sobre el 25 de enero. Obviamente, no me dediqué plenamente al proyecto, ya que todavía tenía que acabar el primer semestre. Aún así, yo empecé para adelantar lo máximo posible.

Una vez tuve las investigaciones concluidas, porque tenía todo lo necesario para el proyecto, empecé a programar en Java la conexión en Bluetooth, con sus respectivos dispositivos Bluetooth y la pantalla Mi Pulsera. Desde el 27 de enero hasta el 10 de febrero estuve realizando este proceso.

El sueño, las actividades y el pulso lo pude realizar entre el 11 de febrero y el 17 de febrero, aunque la mayoría de días se utilizaron para realizar ajustes sobretodo en el algoritmo del sueño, ya que era lo más costoso.

Entre el 20 de febrero y principios de marzo se creó la pantalla Explorar. En esta pantalla se demoró tanto tiempo porque en un principio de iba a utilizar la API de *Google Directions* pero por lo comentado en capítulos anteriores no se pudo conseguir porque económicamente era inviable, entonces la opción que quedaba era costosa, pero se consiguió como se esperaba.

La creación de las bases de datos se hizo en el momento que se creaba el sueño, actividad y pulso, ya que fue en ese mismo momento cuando se necesitaban las bases de datos.

Una vez creadas las bases de datos, era el momento de crear las consultas o '*queries*'. Éstas iban creándose según se iban necesitando, por este motivo el proceso de creación de consultas de las bases de datos duró desde el 12 de febrero al 20 de marzo. En cambio, la pantalla Ajustes no demoró tanto tiempo, en este caso, aproximadamente unos 5 días, desde el 12 de marzo hasta el 17 de marzo.

Una vez estaba la aplicación en Android prácticamente acabada, lo próximo era hacer lo mismo en iOS. La principal ventaja de todo esto es que en muchas funciones sólo tenía que transcribir de un lenguaje a otro, en este caso, de Java a Swift.

Se empezó realizando investigaciones de Bluetooth en el lenguaje Swift, esto duró alrededor de 5 días, es decir, desde el 23 de marzo al 28 de ese mismo mes. Con lo investigado, se podría hacer la pantalla conexión y Mi Pulsera pero esto se explica a continuación.

En 15 días, Conexión y Mi Pulsera se crearon. Esto es, desde el día 2 de abril hasta aproximadamente el día 15 de abril. Más tarde se crearían la pantalla de actividad que tardaría 2 días, desde el 16 hasta el 18 de abril. La pantalla Explorar sí que fue un proceso que duró unos 6 días, desde el 20 de abril hasta el 27.

Como había ocurrido en Android, las bases de datos se crearon al principio y más tarde, desde el 3 de abril hasta principios de junio. La pantalla Ajustes duró desde el 5 de mayo al 15 del mismo mes, esto es debido a que tuve algunos problemas personales y durante unos días no pude dedicarle tiempo como me hubiese gustado. Y las pruebas de la aplicación empezaron sobre el 20 de marzo y acabaron a mediados de junio. Finalmente, el borrador del TFM demoró unos 10 días en realizarse, ya que me dedicaba a escribir, para más tarde, dejar este documento finalizado.

Capítulo 8. Conclusiones

Ante todo, debo agradecer la confianza que Paco ha demostrado tener en mí, porque desde el primer día me dejó total libertad para realizar el proyecto a mi manera con mis gustos y mis ambiciones.

Es verdad que cuando le conté mi proyecto, él me dijo que era muy ambicioso, cosa que me dio más ánimos para completarlo como yo quería.

Han sido muchas horas hasta las tantas de la madrugada para ir alcanzando pequeños retos y eso es como la vida, ir construyéndolo todo, poco a poco, ladrillo a ladrillo.

Estoy orgulloso de haber realizado este proyecto porque a priori parecía inalcanzable pero al final lo he conseguido.

He de confesar que al principio no tenía pensado añadir a la aplicación las pantallas de Actividad, Sueño y Pulso, ya que no creía que podría hacer algo parecido. Con la paciencia y la tenacidad lo he conseguido y una cosa más importante que he aprendido es:

"Sea lo que sea, inténtalo. Lo peor que te puede pasar es fracasar en el intento, pero mucho peor es no intentarlo."

Espero sinceramente que haya disfrutado leyendo este proyecto, pues yo lo he disfrutado programando, escribiendo la memoria y sobre todas las cosas, aprendiendo.

Bibliografía

Sitios Web

https://stackoverflow.com/

http://nilhcem.com/android-things/bluetooth-low-energy

https://developer.android.com/guide/topics/connectivity/bluetooth-le

https://source.android.com/devices/bluetooth/ble

https://www.truiton.com/2015/04/android-bluetooth-low-energy-ble-example/

https://medium.com/@avigezerit/bluetooth-low-energy-on-android-22bc7310387a

https://github.com/kshtj24/mi-band-2-connect

https://www.appcoda.com/core-bluetooth/

https://codeburst.io/getting-started-with-bluetooth-low-energy-on-ios-ada3090fc9cc

https://stackoverflow.com/questions/43358925/how-to-connect-with-bluetooth-low-energy-in-ios-swift

https://scribles.net/creating-a-simple-ble-scanner-on-iphone/

https://swifting.io/blog/2017/07/05/44-watch-your-bluetooth/

https://github.com/danielweber90/MiBand-for-Swift

https://github.com/inFullMobile/inFullBand

https://medium.com/machine-learning-world/how-i-hacked-xiaomi-miband-2-to-control-it-from-linux-a5bd2f36d3ad

https://blog.infullmobile.com/introduction-to-bluetooth-le-on-ios-mi-band-2-case-study-343153921877

http://allmydroids.blogspot.com/2014/12/xiaomi-mi-band-ble-protocol-reverse.html

https://github.com/paulgavrikov/xiaomi-miband-android

https://github.com/paulgavrikov/xiaomi-miband-cocoa

Libros y trabajos

- **Bluetooth 1.1 – Connect without cables.** Second Edition. *Jennifer Bray and Charles F Sturnan.* Prentice Hall. The Service Discover Protocol

- **Tecnología Bluetooth.** Nathan J. Muller. McGraw-Hill Profesional. Protocolos, conceptos básicos y detalles relacionados con la tecnología Bluetooth.

- **Diseño y estudio de un sistema de comunicación inalámbrico basado en tecnología Bluetooth Low Energy. Trabajo Fin de Grado**. Luis Garijo Gutierrez. [En línea]. http://academica-e.unavarra.es/xmlui/bitstream/handle/2454/21877/TFG%20Protocolo%20de%20Enrutamiento%20-%20Luis%20Garijo.pdf?sequence=1

Referencias

[1] W. Wong, «Bluetooth Mesh, Far-field Voice Processing, and More at CES 2016,» electronic design, 1 Febrero 2016. [En línea]. Available: http://electronicdesign.com/embedded/bluetooth-mesh-far-field-voice-processing-and-more-ces-2016.

[2] Specification of the Bluetooth System. Version 1.1, February 22, 2001. https://inf.ethz.ch/personal/hvogt/proj/btmp3/Datasheets/Bluetooth_11_Specifications_Book.pdf

[3] Simulation multi-moteurs multi-niveaux pour la validation des spécifications système et optimisation de la consummation. [En línea]. www.theses.fr/2016NICE4008.pdf, Fangyan LI

[4] C. C. A. R. D. Kevin Townsend, Getting Started with Bluetooth Low Energy, O'Reilly Media, 2014-04-29.

[5] www.solidgeargroup.com

[6] https://techlandia.com/funciona-aes-info_215975/

[7] https://kodigoswift.com/cocoapods-instalacion-y-gestion-de-dependencias/

[8] https://www.ecured.cu/SQLite

[9] https://esacademic.com/dic.nsf/eswiki/512434

Fin.