ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

DESCUBRIENDOS DISPOSITIVOS IoT

DISCOVERING IoT DEVICES

Realizado por

Steven Montoya Vargas

Tutorizado por

Mercedes Amor Pinilla

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Junio 2018

Fecha defensa:

El Secretario del Tribunal

RESUMEN:

Este proyecto consiste en la realización de una API (Application Programming Interface) que permita la conexión e intercambio de datos entre dispositivos de monitorización y dispositivos móviles (Smartphone, Tablet, etc.) que empleen la tecnología Bluetooth Low Energy y utilicen perfiles GATT (Generic Attribute Profile) oficialmente adaptados.

La singularidad de dicha API es facilitar la codificación de cualquier aplicación móvil desarrollada en el sistema operativo Android, cuyo principio sea el uso de la tecnología Bluetooth Low Energy (Bluetooth LE). Aportando una colección de funciones y procedimientos para el descubrimiento de dispositivos, establecimiento de conexión y recepción y envío de datos. Está centrada en perfiles GATT oficialmente adaptados.

PALABRAS CLAVE:

- API
- Bluetooth Low Energy
- Android

SUMARY:

This project consists in the realization of an API that allows the connection and exchange of data between monitoring devices and mobile devices (Smartphone, Tablet, etc.) that use Bluetooth Low Energy technology and use GATT profiles officially adapted.

The singularity of this API is to facilitate the development of any Android mobile requiring the use of Bluetooth Low Energy technology for communication purposes. The Object-Oriented API facilitates the programming of Bluetooth-based communication activities by providing a set of constants, attributes classes, functions and techniques for devices discovery, connection establishment, and the reception and sending of data.

KEYWORDS:

- API
- Bluetooth Low Energy
- Android

INDICE

1 INTRODUCCIÓN _____ *1*
2 OBJETIVOS _____ *3*
3 TECNOLOGÍA BLUETOOTH _____ *7*
3.1 BLUETOOTH LOW ENERGY _____ **8**
4 ARQUITECTURA DE BLUETOOTH LOW ENERGY _____ *11*
4.1 CAPA DE APLICACIÓN _____ **12**
4.2 CAPA HOST _____ **12**
4.2.1 Protocolo de Control y Adaptación del Enlace Lógico _____ 12
4.2.2.1 Handler _____ 13
4.2.2.2 Tipos de atributos _____ 13
4.2.2.3 Permisos de Atributo _____ 14
4.2.2.4 Valores de Atributo _____ 14
4.2.3 Perfil de Atributo Genérico _____ 15
4.2.3.1 Servicios _____ 15
4.2.3.2 Características _____ 16
4.2.4 Perfil de Acceso Genérico _____ 18
4.2.4.1 Roles _____ 18
4.2.4.2 Modos y procedimientos _____ 20
4.2.4.3 Emisión y observación _____ 21
4.2.4.4 Descubrimiento _____ 21
4.2.4.5 Establecimiento de conexión _____ 23
4.2.4.6.1 Tipos de direcciones. _____ 26
4.2.4.6.2 Modos y procedimientos de seguridad. _____ 26
4.2.4.6.3 Servicio GAP _____ 27
4.3 CAPA DEL CONTROLADOR _____ **28**
4.3.1 Capa física _____ 28
4.3.1.1 Canales de radio _____ 29
4.3.1.2 Modulación _____ 30

4.3.1.3 Potencia y rango de trasmisión. 31
4.3.2 Capa de enlace 31
4.3.2.1 Máquina de estados 32
4.3.2.1.1 Estado de espera (Standby) 32
4.3.2.1.2 Estado de anuncio/difusión 32
4.3.2.1.3 Estado de escaneo 33
4.3.2.1.4 Estado de iniciación 33
4.3.2.1.5 Estado de conexión 33
4.3.2.5 Direcciones del dispositivo 34
4.3.2.5.1 Dirección aleatoria 34
4.3.2.5.2 Dirección pública 35
4.3.2.6 Canales 35
4.3.2.7 Frecuencia de salto adaptable 36
4.3.2.8 Formato de paquete 37
4.3.2.8.1 Preámbulo 38
4.3.2.8.2 Dirección de acceso 38
4.3.2.8.3 Cabecera 38
4.3.2.8.4 Longitud 39
4.3.2.8.5 Carga útil 39
4.3.2.8.6 Verificación de redundancia cíclica 40
4.3.2.9 Topología 41
4.3.2.9 Cifrado 43
4.3.2.10 Escáner y conexión desde el punto gráfico 43
5 HERRAMIENTAS EMPLEADAS EN EL DESARROLLO ***47***
5.1 TECNOLOGÍAS SOFTWARE **47**
5.2 TECNOLOGÍAS HARDWARE **49**
6 IMPLEMENTACIÓN DEL PROYECTO ***51***
6.1 SISTEMA OPERATIVO MÓVIL ANDROID **51**
6.2 ANDROID Y BLUETOOTH LOW ENERGY **51**
6.2.1 Problemática y objetivos 52
6.2.2 Solución propuesta 54

6.2.3 Estructura de la API-B desarrollada 55
6.2.3.1 Patrón de diseño 55
6.2.3.2 ListAdapter 56
6.2.3.3 BluetoothLowEnergyScanner 57
6.2.3.4 Clases de tipo interfaz y de constantes 57
6.2.3.5 AbstractBluetoothLowEnergy 58
6.2.4 Modelo de uso 60
6.2.4.1 Diagrama de secuencia 60
6.2.4.2 Ejemplificación de uso mediante App. 62
7 CONCLUSIONES 67
8 REFERENCIAS 69
9 BIBLIOGRAFÍA 71
ANEXO I 73
1. EXPORTACIÓN DE LIBRERÍA 73
2. IMPORTACIÓN DE LIBRERÍA 74
ANEXO II 79
1. DOCUMENTACIÓN API 79

1 INTRODUCCIÓN

En los últimos años las tecnologías inalámbricas han ido creciendo de manera exponencial hasta el punto de introducirse por completo en todos los hogares, donde han ido sustituyendo a las conexiones cableadas, como el teléfono fijo, el internet vía ethernet, etc. Y es que el continuo movimiento de las personas ha generado la necesidad de que las antiguas tecnologías se hayan renovado de forma tan satisfactoria. La que ha presentado un mayor avance en este campo es la tecnología móvil, la cual, gracias a los avances en investigación y desarrollo ha conseguido reducir los costes del producto, para hacerlo llegar a un gran número de consumidores finales.

Hoy en día los teléfonos inteligentes permiten al usuario realizar casi cualquier actividad diaria de forma rápida y sencilla, llegando en algunos casos a sustituir al ordenador personal, es por esto por lo que han alcanzado tanta relevancia en el mundo digital. Pero es importante preguntarse en qué se diferencian estos dispositivos de sus antecesores que los hace tan necesarios en la actualidad. Podemos enumerar una gran lista, ya que el avance de esta tecnología ha sufrido una innovación sinigual, pero dada la naturalidad de este trabajo, es menester que nos centremos en las aplicaciones móviles que dotan a estos dispositivos de vida. Entendiéndose aplicación móvil como un servicio informático debidamente pensado para ser ejecutado en teléfonos inteligentes, tabletas, entre otros.

La revolución de las aplicaciones móviles llegó con el avance en la tecnología WAP, el cual era un protocolo que actuaba como pasarela entre el dispositivo móvil e internet, simplificando el acceso a la web que se quería consultar. Las webs a las que se accedía estaban preparadas para ser leídas en móviles, escritas en ficheros de texto WLM (Wireles Markup Language). Posteriormente el gran paso fue dado por iOS de Apple y Android de Google, los dos sistemas operativos móviles por excelencia, que dieron lugar a un sinfín de aplicaciones aportadas por millones de desarrolladores a través del todo el mundo.

El siguiente adelanto de la tecnología inalámbrica es el Internet de las cosas (Internet of Things o IoT por sus siglas en inglés), un concepto surgido en el Instituto de Tecnología de Massachussetts (MIT). La IoT propone una total revolución en la interacción de las personas con los objetos y de los objetos entre

sí, ya que estos últimos se conectarán entre ellos y con la red para obtener diversos datos en tiempo real, digitalizando el mundo físico y cambiando la sociedad que hoy en día conocemos.

La Internet de las Cosas está llamada a revolucionar el futuro de los hogares, poniendo a disposición de cualquier persona un ecosistema tecnológico inmenso, como neveras inteligentes capaces de realizar la compra, puertas capaces de abrirse y cerrar mediante una aplicación, cepillos de dientes que te notifican de caries y te pide una consulta al dentista de forma inmediata, etc.

Fuera de los hogares, la IoT es si cabe aún más importante, ya que da lugar a las llamadas ciudades inteligentes (*SmartCities*). En estas, la interacción y comunicación con los usuarios será total, sirviéndose mutuamente. Las primeras obtendrán información en forma de producto desde las redes sociales, los blogs, o los servicios móviles, mientras que, del otro lado, serán los propios usuarios los que, además de mejorar la comunicación entre ellos, tendrán a su disposición todo tipo de información para ser consumida ya sea de servicios públicos, eventos sociales, información de transporte, etc.

Ligado a este flujo de continuo avance, el ámbito de la seguridad tomará un papel fundamental y completamente activo en las tecnologías ligadas a la IoT, comprometiéndose a la difícil tarea de proporcionar tranquilidad y sostenibilidad en un entorno tecnológico, pero totalmente cosificado.

Las tecnologías que permiten las conexiones inalámbricas han construido y seguirán construyendo, un futuro que está aún por inventar, generando un progreso vertical, de cero a uno. Una de estas es la tecnología Bluetooth, la cual está muy presente en la mayoría de dispositivos que se manejan hoy en día, Smartphone, Tablet, Pc, Smartwatch. Más de 1800 marcas utilizan este sistema para sus dispositivos, bajo la autorización de SIG (*Bluetooth Special Interest Group*). Dado su amplio uso, la tecnología Bluetooth se ha ido adaptando, en sus sucesivas versiones, a las necesidades de la IoT y a los dispositivos personales actuales, así como a los diferentes escenarios de comunicación. A partir de su versión 4, Bluetooth incorpora mejoras para un menor consumo de energía (Bluetooth Low Energy – BLE) y facilita la caracterización y difusión de datos.

"En 4 años habrá unos 48 mil millones de dispositivos inteligentes de los cuales alrededor de un tercio tendrán conexión Bluetooth, es decir, varios miles de millones de dispositivos para conectarnos con nuestro smartphone. Desde luego el estándar Bluetooth será uno de los protagonistas del internet de las cosas (IoT)" [1].

2 OBJETIVOS

El objetivo de este proyecto es diseñar una librería de aplicaciones o API basada en Bluetooth Low Energy (Bluetooth LE). Su principal atractivo es facilitar el desarrollo de aplicaciones móviles para el sistema operativo (SO) Android de Google, que hagan uso de la tecnología Bluetooth LE, proporcionando al desarrollador, una serie de funciones y procedimientos para facilitar la tarea de este a la hora de descubrir dispositivos, el establecimiento de conexión y la trasferencia de datos para ciertos perfiles GATT.

Más concretamente, esta API actuará como capa intermedia entre las librerías propias que presta Android 4.3 (API level 18), la cual introduce soporte para Bluetooth LE, (añadiendo características que son propias de esta tecnología, diferenciándose así de la librería que ya existía para el Bluetooth clásico) y el desarrollador. Librando a este último de la compleja tarea de conocer el funcionamiento interno de Bluetooth LE y cómo Android interacciona y trabaja con esta tecnología.

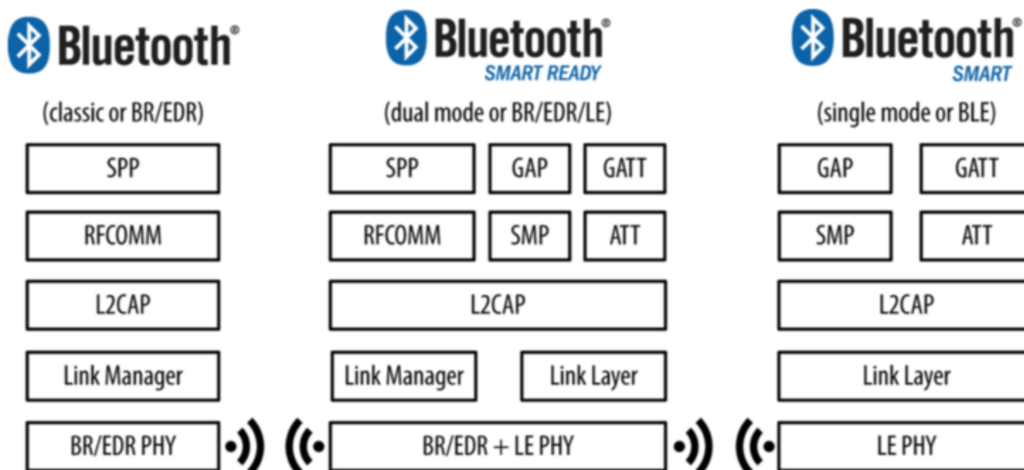


Figura 2.1. Comparación de la pila de protocolos de las versiones de Bluetooth [2].

Bluetooth Clásico es el usado por dispositivos que necesitan trasmitir grandes cantidades de datos, lo que conlleva un gran consumo de batería del dispositivo. Bluetooth Smart está pensado para dispositivos que no necesitan un amplio intercambio de datos, por lo que su consumo es mínimo. Haciendo referencia a la comunicación entre dispositivos, también encontramos Bluetooth Smart Ready el cual propone un modo dual, permitiendo así que los dispositivos que lo integren puedan establecer conexión tanto con Bluetooth clásico como Low Energy.

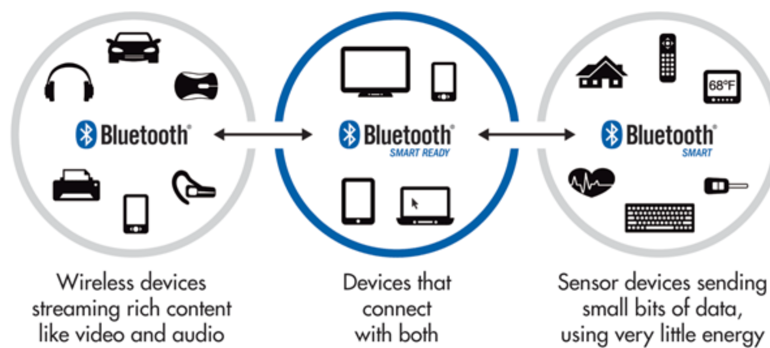


Figura 2.2. Distinción de los dispositivos que usan cada versión de Bluetooth [3].

If your product bears this logo…	It's compatible with products bearing any of these logos…		
Bluetooth® SMART READY	Bluetooth® SMART READY	Bluetooth®	Bluetooth® SMART
Bluetooth®	Bluetooth® SMART READY	Bluetooth®	
Bluetooth® SMART	Bluetooth® SMART READY		

Figura 2.3. Logos Bluetooth en relación con la versión y sus compatibilidades [4].

A continuación, se enumeran los términos y conceptos (en inglés) que son necesarios tener en cuenta a la hora de trabajar con Bluetooth Smart:

- Generic Attribute Profile (GATT).
- Attribute Porotocol (ATT).
- Characteristic.
- Descriptor.
- Service.

Se comentarán más detalladamente en los próximos capítulos, con la intención de que se comprendan en profundidad. Se analizará también cómo la API que da lugar a este trabajo intermedia para que estos conceptos sean prácticamente trasparentes al programador, que necesita de ellos para la codificación de su aplicación móvil.

3 TECNOLOGÍA BLUETOOTH

La tecnología Bluetooth, es una tecnología inalámbrica de corto alcance concebida para redes de área personal (WPAN) que facilitan la conexión entre dos dispositivos y que fue creado por Bluetooth Special Interest Group, Inc. Esta especificación permite la trasmisión tanto de datos como de voz mediante el uso de radio frecuencia y con un ancho de banda de 2.4 GHz.

Los dispositivos que hacen uso de este protocolo necesitan estar dentro de alcance para que sea posible la trasferencia de información. Existe una clasificación en función de la potencia y el alcance de los dispositivos.

Clase	Potencia máxima permitida (mW)	Potencia máxima permitida (dBm)	Alcance (aproximado)
Clase 1	100 mW	20 dBm	~100 metros
Clase 2	2.5 mW	4 dBm	~5-10 metros
Clase 3	1 mW	0 dBm	~1 metro

Figura 3.1. Clasificación de dispositivos en función de su potencia de trasmisión. Fuente: [5].

Estos dispositivos también pueden ser clasificados de acuerdo con su capacidad de canal:

Versión	Ancho de banda (BW)
Versión 1.2	1 Mbit/s
Versión 2.0 + EDR	3 Mbit/s
Versión 3.0 + HS	24 Mbit/s
Versión 4.0	32 Mbit/s

Figura 3.2. Clasificación de dispositivos en función de su capacidad de canal. Fuente: [5].

Actualmente hay cinco versiones de Bluetooth, de las cuales podemos extraer las siguientes distinciones principales. Bluetooth clásico, Bluetooth Smart Ready y Bluetooth Smart.

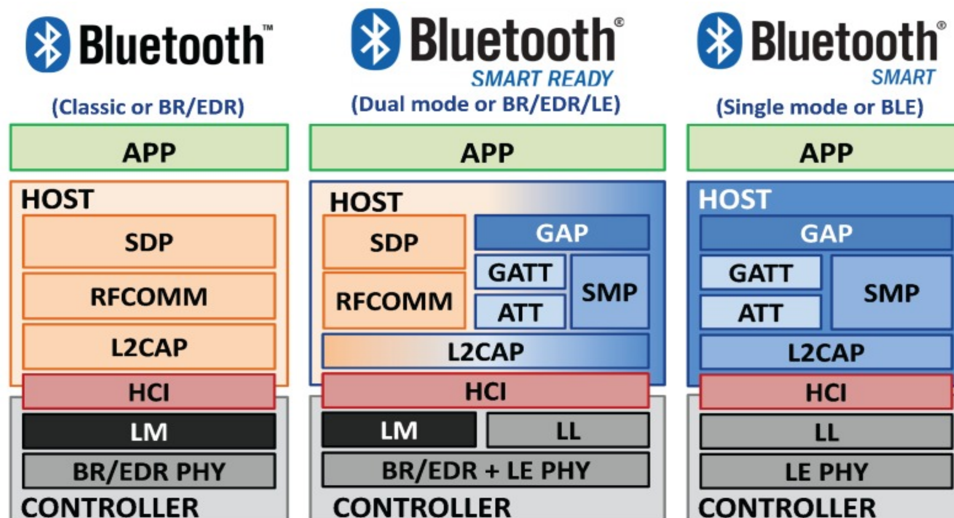


Figura 3.3. Configuración entre versiones Bluetooth y tipos de dispositivos. Fuente: [6].

3.1 BLUETOOTH LOW ENERGY

Una vez hecha la introducción de Bluetooth de forma general, este trabajo se centrará en Bluetooth Low Energy o Bluetooth Smart, en adelante BLE.

BLE se introdujo en la versión 4.0, con el fin de dar soporte a la gran cantidad de wearables (entiéndase como dispositivos que se incorporan como “vestimenta” en alguna parte del cuerpo) que se están comercializando en el mercado. Con una gran reducción del consumo y por consiguiente un aumento de la autonomía, este nuevo estándar de Bluetooth permite que dispositivos que son alimentados por una “pila de botón” y que trabajan con esta tecnología consigan extender su función durante días, incluso meses sin reemplazar su batería.

BLE utiliza la misma banda (2.4GHz) que usa Bluetooth clásico, como ya se indicó más arriba, con la salvedad de que el protocolo ha sido rediseñado para evitar colisiones, consiguiendo así una reducción del consumo de energía [5]. La versión 4.1 incluye soporte Mesh, topología que permite establecer comunicaciones de muchos a muchos, mejorando la conexión punto a punto que incluía de base.

BLE establece dos distinciones en función del papel que interpreta cada dispositivo.

Central:

- Mantiene la escucha de forma continua, descubriendo que dispositivos están disponibles.
- Tiene la capacidad de comenzar la conexión con otro dispositivo.

Periférico:

- Manifiesta que está disponible mediante el envío de mensajes.
- Puede aceptar peticiones de conexión provenientes de un dispositivo central.

Por lo general un dispositivo asume un rol u otro, pero no ambos a de forma simultánea. El dispositivo que toma el papel de periférico anuncia su presencia hasta que un dispositivo central decide enlazarse con él. Cabe decir que un dispositivo central pude conectarse con varios periféricos, pero este último solo puede tomar conexión con un central.

Es importante destacar la versión 4.2, ya que además de incluir mejoras en la seguridad y la velocidad, permite que los chips Bluetooth puedan usarse bajo el Protocolo de Internet Versión 6 (IPv6). De esta forma los sensores que usen BLE 4.2 pueden trasmitir datos sobre internet. Para esto fue necesario incluir el Perfil de Soporte de Protocolo de Internet (IPSP) en la pila de BLE, este protocolo permite que los dispositivos puedan comunicarse con otros que también soporten IPSP, así como con otros dispositivos que soporten IPv6, la conexión es poco costosa y sencilla, ya que los enrutadores que se usan para dicha conexión confían en los paquetes IPv6 recibidos, sin llevar a cabo ninguna modificación o análisis.

En 2016 se anunció la última versión hasta la fecha, Bluetooth 5, el cual introdujo mejoras en el alcance, hasta 4 veces mayor que en la versión 4.2, permitiendo nuevos usos en la rama de IoT. Se puede observar claramente que el futuro de la IoT está relacionado casi de forma directa con BLE, ya que su gran popularidad y utilidad permitirán que se lleven a cabo nuevas aplicaciones, proyectos o servicios que necesiten de conectividad personal con un bajo consumo. Esta tecnología también permitirá que dispositivos más simples que ya se encuentran en el mercado y a los que le damos uso cada día se actualicen y sean dotados con nuevas funcionalidades.

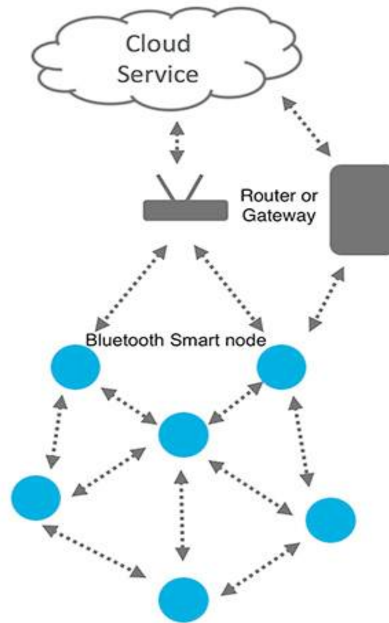
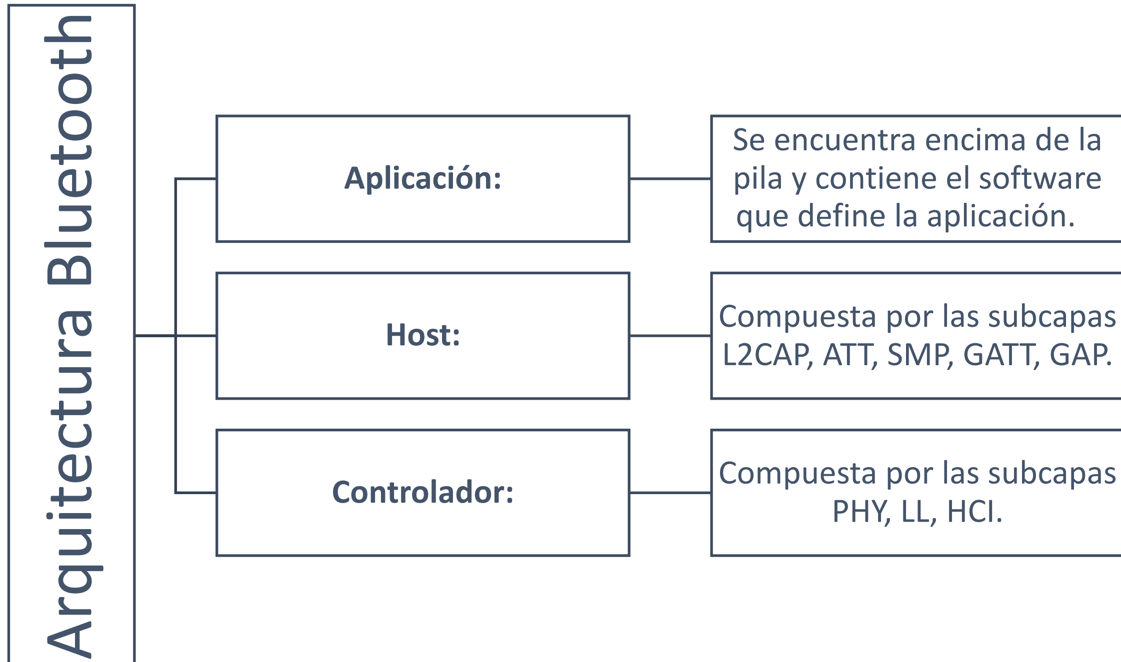


Figura 3.4. Versión 4.2 que posibilita la conexión directa con Internet. Fuente: [7].

4 ARQUITECTURA DE BLUETOOTH LOW ENERGY

La pila de protocolos de BLE está diferenciada en tres capas principales:



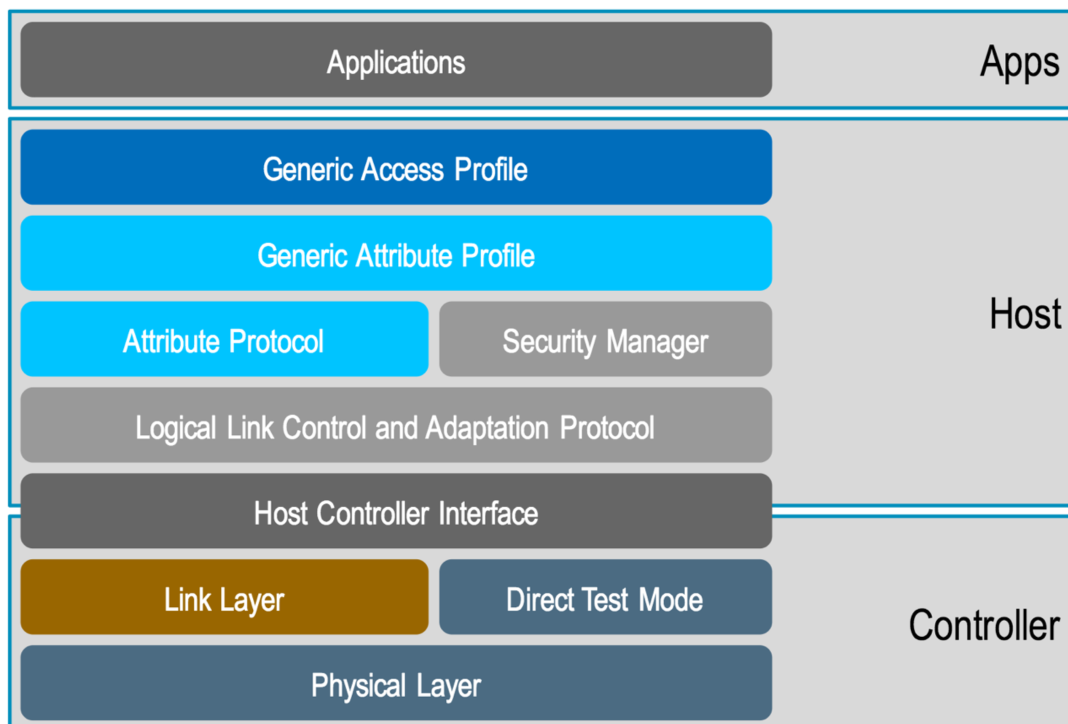


Figura 4.1. Pila de protocolo Bluetooh Low Energy. Fuente: [8].

4.1 CAPA DE APLICACIÓN

La capa de aplicación es donde se alojan la interfaz de usuario, la lógica y estructura general de la aplicación desarrollada para dispositivos compatibles con BLE. Es necesario tener un conocimiento teórico y práctico de esta capa a la hora de comenzar a trabajar con BLE, pero igual de importante es saber cómo funciona la parte de más bajo nivel de la aplicación y cómo esta se relaciona con el hardware y el chip Bluetooth del dispositivo, estas funcionalidades serán vistas en la capa de Host y Controlador.

4.2 CAPA HOST

4.2.1 Protocolo de Control y Adaptación del Enlace Lógico

Protocolo de Control y Adaptación del Enlace Lógico o su nombre en inglés Logical Link Control and Adaptation Protocol (L2CAP), cuya función es transmitir paquetes con y sin orientación a la conexión a sus capas superiores, para ello realiza segmentación y ensamblado de paquetes, así como multiplexación, adaptando el protocolo de las capas superiores antes del reensamblaje. Proporciona también canales de lógicas que son multiplexadas sobre uno o más enlaces lógicos. Todos los paquetes de datos de la aplicación son enviados usando paquetes L2CAP.

El paquete L2CAP está compuesto por la longitud del mismo, el destino de canal lógico y la carga útil.

El destino (CID) pueden ser canales fijos que son orientados a la conexión. Los CIDs desde 0x0001 hasta 0x003F son canales fijos y desde 0x0040 hasta 0xFFFF son dinámicamente asignados.

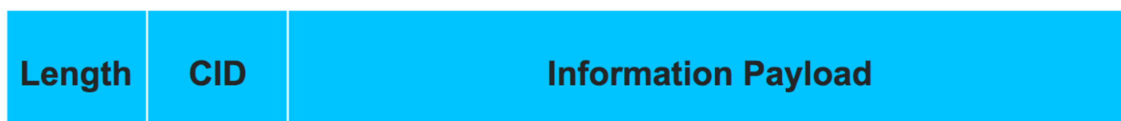


Figura 4.2. Paquete L2CAP de la capa de lógica de enlace. Fuente: [8].

La capa Host también incluye el Protocolo de Atributo (ATT), este protocolo sirve para descubrir, leer y escribir mediante atributos en un dispositivo emparejado, se puede entender como una arquitectura de cliente servidor, donde el cliente reclama información del servicio del servidor. Estos atributos vienen representados en arrays de 0 a 512 octetos de longitud que puede ser fija o variable.

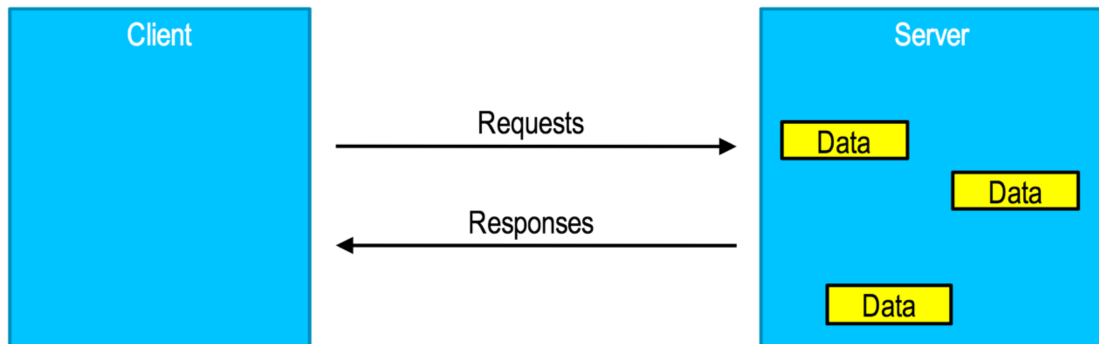


Figura 4.3. Arquitectura de cliente-servidor que define al Protocolo de Atributo. Fuente [8].

4.2.2.1 Handler

Cada atributo viene referenciado por un “handle” de forma única en el servidor, que permite al cliente identificarlo a la hora de realizar cualquier petición de lectura o escritura. Los identificadores son números de 16 bits que no se presentan siempre de forma secuencial, por lo que es necesario realizar una función de descubrimiento con el fin de obtener los atributos que se desean. Para un programador esta estructura da facilidad a la hora de mantener un seguimiento de los atributos con los que se está trabajando.

4.2.2.2 Tipos de atributos

El tipo de cada atributo viene definido por un identificador único universal (UUID) que puede ser de 16, 32 o 128 bits. El tipo permite identificar la naturaleza del valor del atributo y por ende la forma más efectiva de procesar dicho valor.

Handle	Tipo de atributo (UUID)	Valor del atributo.
0x000C	Service Declaration 0x2800	Heart Rate Service 0x180D

Tabla 4.1. Ejemplificación de handle, tipo de atributo y valor de atributo.

Como se puede observar en la tabla 4.1, se descubre un atributo, que, por medio de su UUID, se conoce su tipo, en este caso se trata de un atributo de declaración de servicio, el cual tiene como valor el servicio HeartRate, que viene identificado también por su correspondiente UUID (0X180D). El conocer el tipo del atributo ayuda al desarrollador a la hora de tratar el valor de este, en el caso anterior se trataba de un servicio, por lo que se emplearía un descubrimiento de servicio para conocer su valor. Esto se verá con más detalle más adelante.

4.2.2.3 Permisos de Atributo

Sobre el atributo se definen algunas reglas que establecen como se puede interactuar en cuanto a la permisividad de lectura y escritura. Destacar que estos permisos solo se definen sobre el valor del atributo, no sobre el handle ni el tipo del atributo, es decir, para el caso que se trató en el anterior apartado, el cliente podría descubrir los atributos existentes en el servidor (Service Declaration), pero no poder leer o escribir sus valores. En concreto Heart Rate Sevice tiene establecidos permisos de solo lectura. También es posible encontrarse valores de atributos que requieran autorización o un emparejamiento con suficiente fuerza de señal para poder leer o escribir.

4.2.2.4 Valores de Atributo

El valor del atributo se puede entender como la información que este contiene. Recurriendo de nuevo a la tabla 4.1, se puede extraer como información útil, que el servicio encontrado en el servidor del dispositivo es Heart Rate Service (con UUID 0x180D), este servicio es el estándar establecido por Bluetooth SIG para que aquellos dispositivos que realicen la función de medir el ritmo cardiaco tengan la capacidad para notificarlo.

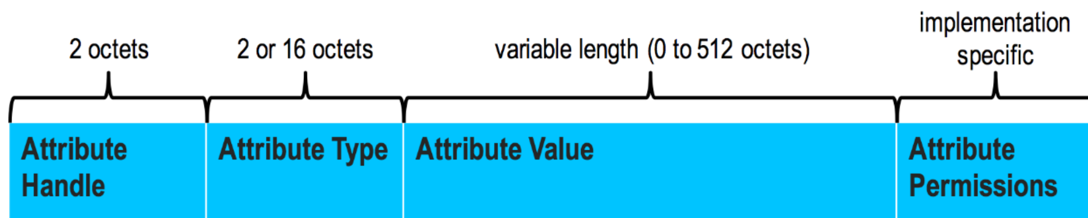


Figura 4.5. Representación lógica de un atributo. Fuente [8].

4.2.3 Perfil de Atributo Genérico

El Perfil de Atributo Genérico (GATT) define un marco de referencia basado en ATT que detalla los mecanismos con los cuales dos dispositivos pueden comunicarse e intercambiar datos de perfil y de usuario mediante el uso de servicios y características.

Mantiene la arquitectura cliente-servidor predispuesta por ATT con la salvedad de que en GATT los datos del servidor se encapsulan en servicios y vienen presentados como características.

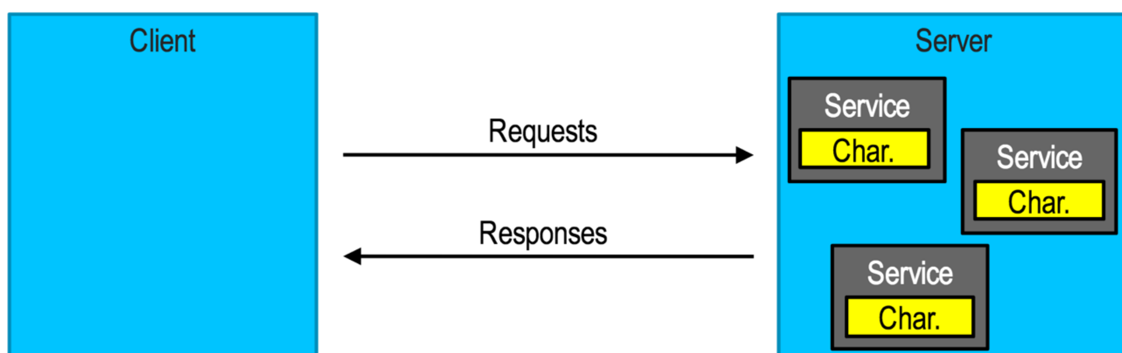


Figura 4.6. Descripción cliente, servidor GATT. Fuente: [8].

4.2.3.1 Servicios

Como ya se ha dicho los atributos de un servidor GATT son una sucesión de servicios, pero, qué es un servicio. Un servicio se puede entender a su vez como una colección de características o referencias hacia otros servicios, que definen la conducta de un dispositivo. El tipo de atributo para los servicios es siempre el

mismo "*Service Declaration*" con UUID 0x2800, siendo este el estándar designado por Bluetooth SIG, aunque también puede ser 0x2801, en función del tipo del servicio.

Se habla de tipo de servicio porque BLE define dos tipos de servicios. El servicio primario, que hace referencia a la principal funcionalidad que ofrece el dispositivo, y el servicio secundario cuya relevancia está sujeta a la participación de otro servicio, ya sea primario o secundario.

4.2.3.2 Características

Las características se pueden interpretar como un conjunto de datos del cliente donde cada una de ellas presenta al menos dos atributos importantes, el primero, cuyo tipo viene definido por Bluetooth SIG como "*Characteristic Declaration*" con UUID 0x2803, proporciona metadatos sobre la característica que se está tratando, sus permisos son siempre de solo lectura y sin autenticación. Y como segundo, el valor de la característica, que aporta los datos reales que proporciona el dispositivo (nombre del dispositivo, dirección, etc).

Así mismo, puede que el valor de la característica venga seguido de descriptores, los cuales añaden más metadatos a los ya leídos en la declaración de la característica. También pueden ser específicos del comerciante.

Obsérvese en la siguiente página una tabla como apoyo visual, donde se resumen los dos puntos anteriores

Servicio

Tipo del atributo	Permisos del atributo	Valor del atributo
Declaración de Servicio **UUID → 0x2800**	-Solo lectura -Sin identificación -Sin autorización	Heart Rate Service UUID → 0x180D

Característica

Tipo del atributo	Permisos del atributo	Valor del atributo
Declaración de Característica **UUID → 0x2803**	-Solo lectura -Sin identificación -Sin autorización	Propiedades (notificación) Valor del handle (0xNNNN) UUID de Característica de medición de frecuencia cardiaca (0x2A37)
Valor de Característica **Característica de medición de frecuencia cardiaca** **UUID → 0x2A37**	Especificación en capa superior.	Latidos por minuto (lpm)
Descriptor **Característica de configuración del cliente** **UUID → 2902**	Lectura sin autenticación o autorización. Para escritura es necesario autorización e identificación.	Notificaciones activadas o desactivadas

.

.

.

Característica

4.2.4 Perfil de Acceso Genérico

Un dispositivo que use la tecnología Bluetooth LE tiene a su disposición distintas opciones a la hora de relacionarse con su entorno, la difusión de datos (broadcasting) o descubrir y conectar con otro dispositivo son un ejemplo de ello. Independientemente del mecanismo escogido, estos vienen regidos por las directrices que marca GAP.

Se continuará el apartado de GAP definiendo diversos aspectos importantes que se definen del perfil sobre Bluetooth LE.

4.2.4.1 Roles

Cuando dos dispositivos se comunican entre sí, pueden estar obrando bajo uno o varios roles que les permiten esta acción, respetando ciertos requerimientos. GAP se encarga de que las iteraciones entre los roles mencionados se realicen de forma precisa, por norma general, aunque no exclusiva, los roles suelen vincularse con ciertos tipos de dispositivos en concreto, inclusive para un gran número de implementaciones se asocian de forma directa con su usabilidad.

Son cuatro los roles que define GAP y que puede seguir cualquier dispositivo que trabaje con tecnología BLE y desee entablar conexión con su entorno.

-**Difusor (broadcaster):**

Este rol está pensado para dispositivos que trabajen distribuyendo información de forma continuada hacia cualquier dispositivo que se encuentre “escuchando”, un dispositivo que adopte este rol trasmite paquetes de datos de forma pública. Un termómetro público que trasmita las lecturas de temperatura leídas, podría ser un ejemplo de uso de este rol [9]. El emisor se vale del rol de notificación de la capa de enlace para llevar a cabo su función.

-**Observador:**

Hacen uso de este rol los dispositivos que recaban los datos emitidos por aquellos que se adhieren al rol de broadcaster. Un dispositivo con una pantalla, en la cual muestre datos sobre mediciones echas por un sensor que emite la trama de datos, sería un ejemplo de uso de este rol [9]. El rol de observador usa el modo de escáner de la capa de enlace.

-Central:

El rol central lo toman los dispositivos que pueden ofrecer un alto rendimiento, tales como smartphones o tablets, los cuales pueden establecer conexión con múltiples dispositivos de forma simultánea. Esto es debido a que BLE usa un protocolo de carácter asimétrico, lo que conlleva una mayor carga de computación en la capa de enlace del dispositivo maestro. El dispositivo maestro es el encargado de iniciar la conexión con el dispositivo esclavo.

-**Periférico:**

Este rol, hace referencia al dispositivo que actúa como esclavo. Usa paquetes de notificación, los cuales permiten que el dispositivo que toma el papel de central, pueda encontrarlo, para luego establecer conexión con él. BLE está diseñado para que los recursos necesitados por el periférico sean los mínimos posibles, en cuanto a potencia, procesamiento, memoria y autonomía, permitiendo así la comercialización de dispositivos con un bajo coste.

Un error común es confundir los roles de cliente servidor, que se vieron con anterioridad en el apartado de GATT, con los roles definidos por GAP. Entrando en detalle, un dispositivo puede actuar tanto de servidor como cliente, ya que la naturaleza de esta definición se basa en el sentido de trasmisión de los datos, es decir, si el periférico esta enviado trama de datos hacia el dispositivo central, es el primero el que actúa como servidor, sin embargo, se hablaría de cliente en el caso de ser este mismo quién realiza esta vez la petición de datos.

Por otro lado, los roles definidos por GAP permanecen inmutables. Una pulsera de monitorización cardiaca actuará siempre como periférico frente al smartphone, el cual será el dispositivo central, no obstante, la pulsera puede tomar tanto rol de cliente como de servidor según su situación en la trasmisión de datos.

En la figura 4.7 y 4.8, puede observarse el diagrama de estados para cada uno de los roles antes definidos. Dichos estados hacen referencia a la máquina de estados de la capa de enlace, que se verá posteriormente en el apartado 4.3.2.

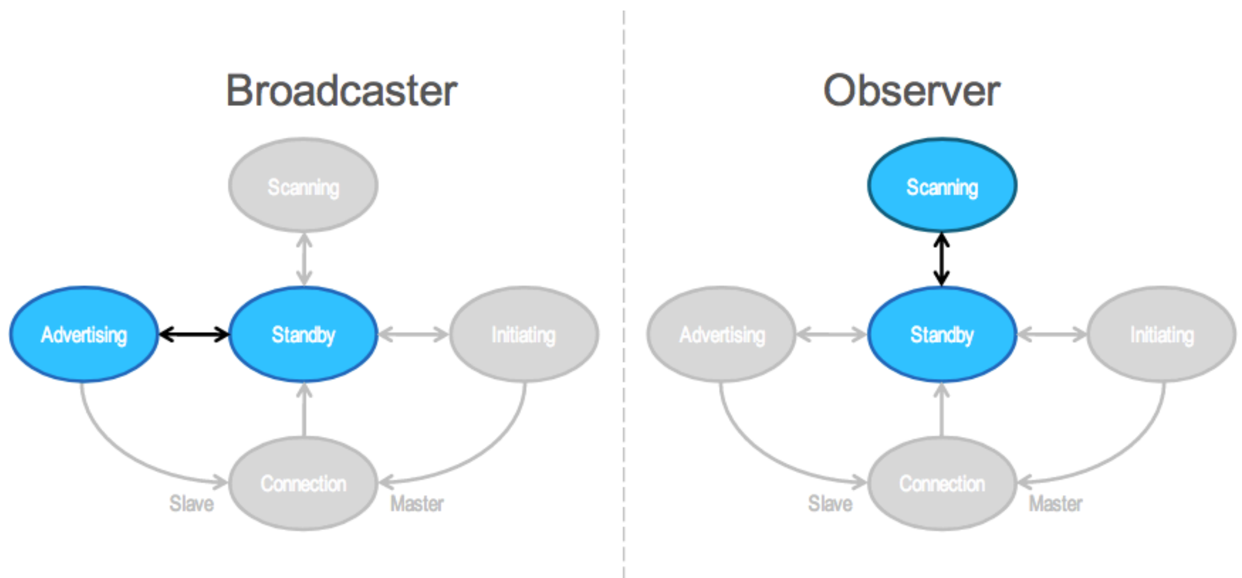


Figura 4.7. Diagrama de estados de los roles de difusor y observador. Fuente: [8]

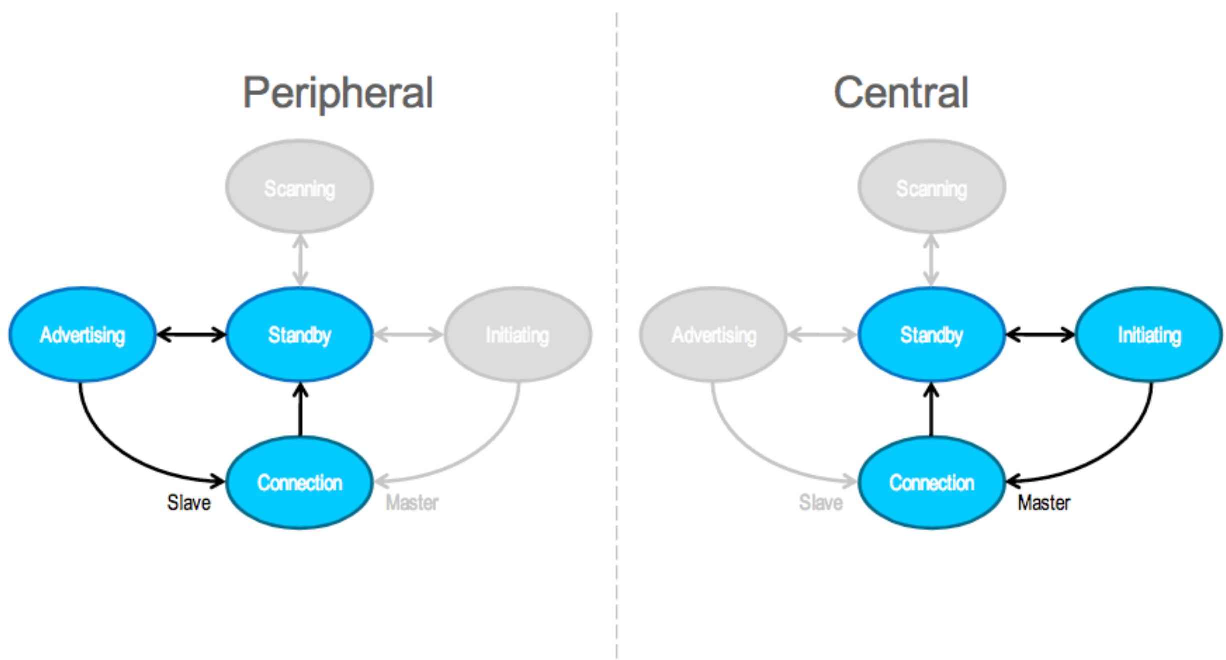


Figura 4.8. Diagrama de estados de los roles de periférico y central. Fuente: [8].

4.2.4.2 Modos y procedimientos

Se entiende como modo un estado que el dispositivo puede experimentar en un determinado momento, con el fin de que este realice un procedimiento concreto. Dichos modos pueden activarse de forma automática cuando sea conveniente o mediante interacción con la interfaz de usuario.

Por otro lado, se encuentran los procedimientos. Estos permiten que un dispositivo alcance una meta, mediante una sucesión de acciones. Habitualmente un procedimiento está directamente relacionado con un modo que los une de una forma determinada.

Se verán en detalle algunos de estos modos y procedimientos en los siguientes apartados.

4.2.4.3 Emisión y observación

El modo de emisión, que se encuentra definido en GAP, establece un marco de trabajo por el cual los dispositivos que lo siguen pueden emitir datos en una sola dirección, actuando como emisor, y enviando estos hacía otros dispositivos que se mantienen en escucha, actuando como observadores (procedimiento de observación). Ni los emisores tienen la certeza de que los datos van a llegar a otro dispositivo ni los observadores de que van a recibir datos ya que no existe ningún tipo de confirmación.

Los paquetes que son enviados por el dispositivo que sigue el modo de emisión, contienen información actual del usuario, así como metadatos (nombre del dispositivo, por ejemplo), los cuales son insertados por la capa de enlace.

4.2.4.4 Descubrimiento

Un periférico tiene varias formas de anunciar su presencia, y su información también puede ser tratada de distintas maneras y para diversos fines. Esto hace que sea necesario indicar qué modo va a seguir para anunciarse e indicar que uso se puede dar a la información que emita.

La solución a esto viene dada por SIG, el cual incluyó un campo en los paquetes de datos de notificación llamados Flags, que indican precisamente el modo en el que se encuentra el dispositivo. El modo descubrible es usado únicamente por periféricos, cuyo fin último es permitir que los dispositivos que actúen como centrales, puedan descubrir los periféricos que se encuentren dentro de su rango.

El proceso de descubrimiento usualmente se centra en advertir la presencia de un dispositivo próximo y una cierta información básica del mismo, sin que esto obligue a comenzar una conexión. Por lo general el descubrimiento suele usarse

para listar (generalmente en una pantalla de Smartphone o Tablet) los diversos dispositivos que se encuentran en las proximidades, con la intención última de que sea el usuario el que seleccione a cuál quiere conectarse.

En GAP se incluyen **tres** modos que puede experimentar el periférico a la hora de anunciarse:

Modo No detectable

- Este modo establece que un dispositivo no puede ser descubierto por ningún dispositivo central.

Modo Limitado-Descubrible

- Este modo es usado por el periférico, cuando su deseo es permanecer descubrible por un corto periodo de tiempo. Los paquetes que son enviados por dispositivos bajo este modo definen su campo “Flags” como “Limited Discoverable”.

Modo General-Descubrible

- Es el modo que se usa generalmente, en este estado, un dispositivo indica su deseo de ser descubierto, habitualmente con la intención de establecer conexión con los dispositivos centrales que puedan descubrirle. Para este modo, el “Flag” indicado en el paquete debe ser “General Discoverable”.

Los modos anteriormente descritos son utilizados por los periféricos. Conjuntamente a ellos, los dispositivos centrales manejan una serie de procedimientos concordantes con el modo en el que se encuentra el periférico para su descubrimiento, permitiendo así que tanto centrales como periféricos establezcan sus preferencias en este proceso.

En GAP se definen **dos** tipos de procedimientos para estos modos.

Procedimiento de Descubrimiento General:

- Este procedimiento es usado por el dispositivo central, cuando no se desea realizar ningún filtro a la hora de realizar el escaneo de dispositivos, siendo indiferente si el estado del periférico es Limitado-Descubrible o Limitado-General. Para cualquiera de los dos casos se informa a la capa de aplicación.

Procedimiento de Descubrimiento Limitado:

- En este caso el dispositivo central no establece ningún filtro al igual que en el procedimiento anterior, con la salvedad de que, en este modo, analiza todos los paquetes que descubre durante el escaneo y solo notifica a la capa de aplicación cuando este incluye en su campo “Flag” “Limited Discoverable”.

4.2.4.5 Establecimiento de conexión

Como ya se ha explicado en puntos anteriores de este trabajo, el dispositivo que actúa como central es el que tiene la potestad de iniciar la conexión con el periférico que él desee. Pero para poder que se de dicha acción, es necesario que el periférico en cuestión se encuentre en modo conectable.

Al igual que ocurre con el descubrimiento, para la conexión también existen una serie de modos aplicables a los periféricos, que a su vez están relacionados con diversos procedimientos que son llevados a cabo por el dispositivo central y son definidos de forma estándar por GAP.

Modo No Conectable:

- Tal como lo de define su nombre, un periférico que se encuentre en este estado no está disponible para establecer conexión, para este caso el dispositivo puede optar por no enviar paquetes de anuncio o en caso de hacerlo, estos paquetes indican que su naturaleza es de no conexión.

Modo Conectable sin dirección:

- Un periférico en este modo de conexión envía paquetes no dirigidos, es decir, hacia cualquier dispositivo central, en busca de establecer una conexión, este modo de conexión suele ser en el que se encuentran generalmente los periféricos. Cabe decir que se establece de manera automática cuando el modo de descubrimiento es "General Discoverable" que fue analizado anteriormente.

Modo Conectable direccionado:

- A diferencia del modo anterior, en este estado, el periférico no envía paquetes de datos de forma indiscriminada y de manera promiscua, sino que estos incluyen la dirección Bluetooth de un determinado dispositivo central, usualmente este modo hace referencia a un intento de reconexión, de forma que, al realizarse el filtro por la dirección Bluetooth, esta se realiza de manera más rápida.

Son cuatro los procedimientos que pueden llevar a cabo los dispositivos centrales para trabajar con los modos antes descritos.

Procedimiento de establecimiento de conexión directa:

- Procedimiento que sigue el dispositivo central para conectarse con un periférico concreto, usando su dirección Bluetooth.
- Este procedimiento puede llevarse a cabo tanto con dispositivos en modo "Conectable direccionado" o "Conectable sin dirección", pero viendo su proceder es fácil afirmar que es más efectivo cuando el periférico se encuentra en el primer modo.
- En caso del modo "No conectable" este procedimiento concurriría en un error.

Procedimiento de establecimiento de conexión selectiva:

- Este procedimiento permite conectarse con cualquier dispositivo, con la salvedad de que los paquetes emitidos por el periférico son sometidos a un filtro por parte del dispositivo central, quedándose únicamente con los paquetes provenientes de dispositivos ya conocidos.
- Generalmente se listan estos dispositivos seleccionados dejando al usuario que decida cuál es el dispositivo con el que desea conectarse.

Procedimiento de establecimiento de conexión automática:

- Para este procedimiento también se seleccionan una serie de dispositivos conocidos, con un modo de operación similar al procedimiento de conexión selectiva, diferenciándose en que en este caso se le indica al dispositivo central que se conecte con el primer dispositivo de dicha lista que sea descubierto.

Procedimiento de establecimiento de conexión general:

- Procedimiento que sigue un dispositivo central cuando quiere conectarse a un nuevo dispositivo descubierto.

Tiene una técnica de actuación de dos pasos:

Paso 1

El primer paso se centra en aceptar todos los paquetes de anuncio recibidos, sin realizar ningún tipo de filtro a la hora de aceptarlo.

En este mismo paso realiza un filtro para los datos recibidos tales como nombre, dirección, servicios, etc.

El resultado de este filtro genera la dirección MAC que se usará para conectarse con el dispositivo pretendido.

Paso 2

En el segundo paso, el dispositivo central detiene el escaneo y se conecta con el dispositivo que le sea indicado.

Para ello se basa en el procedimiento de establecimiento de conexión directa.

4.2.4.6 Seguridad y Servicio GAP

Bluetooth LE también ha hecho hincapié en el ámbito de la seguridad, definiendo diversos protocolos que ayudan a que las conexiones entre dispositivos sean seguras y privadas.

Security Manager desempeña un papel importante en la seguridad de BLE ya que tanto su protocolo como sus diversos algoritmos de seguridad, proporcionan la capacidad de generar claves y que sean intercambiadas por todas sus capas, lo que finalmente permite que dos pares de dispositivos puedan autenticarse y trabajar con un enlace encriptado.

Por otro lado, GAP proporciona un conjunto de utilidades que son usados para extender las funcionalidades que proporciona Security Manager.

4.2.4.6.1 Tipos de direcciones.

GAP hace uso de algunas características propias de la capa más baja de Bluetooth en relación con las direcciones Bluetooth del dispositivo. Definiendo tres tipos de direcciones:

Dirección privada resuelta:

- Usa una clave de resolución de identidad (*Identity Resolving Key*) y un valor aleatorio para generar una dirección privada resuelta, que permiten que el dispositivo no sea rastreado por otro desconocido.

Dirección privada no resuelta:

- Esta es una dirección generada aleatoriamente que cambia en cada conexión.

Dirección estática:

- Dirección que se establece generalmente cuando se inicia el dispositivo, se genera de forma aleatoria.

4.2.4.6.2 Modos y procedimientos de seguridad.

Al igual que los modos y procedimientos de descubrimiento y conexión explicados en anteriores apartados, GAP define los propios para la seguridad.

Modo no vinculable:

- Como indica su nombre, cuando un dispositivo se encuentra en este modo, no es posible realizar ninguna acción de vinculación.

Modo vinculable:

- Posibilita la vinculación de un dispositivo con otro. Como resultado permite almacenar las claves de seguridad, aunque haya terminado la vinculación.

Procedimiento de autorización:

- Este procedimiento implica que se obtenga una validación por parte del usuario para continuar con el proceso en el que se esté trabajando.

Procedimiento de autenticación:

- Se inicia acto de seguido de que se haya establecido la conexión.
- Se lleva a cabo un proceso de petición de servicio por parte del cliente, que conlleva una verificación de análisis de los niveles de seguridad y tratamiento de las claves de seguridad generadas.

Procedimiento de encriptación:

- Procedimiento que se encarga de cifrar los datos que van a ser trasmitidos entre el dispositivo central y el periférico.
- La encriptación solo puede ser iniciada por el dispositivo central, aunque el periférico puede enviar una solicitud para que el primero inicie la acción.

4.2.4.6.3 Servicio GAP

Por último, GAP incluye en sus especificaciones su propio servicio, el cual es obligatorio y debe ser introducido por cada dispositivo en su lista de atributos.

Como ya se comentó más detalladamente en el apartado de GATT, los servicios son poseedores de características.

Característica de nombre de dispositivo	Define el nombre local del dispositivo.
Característica de apariencia	Determina cuál es la categoría del dispositivo en cuestión.
	Esta característica viene definida por una variable de 16 bits.
Parámetros de conexión preferidos del periférico	Esta característica define las preferencias que tiene el dispositivo periférico cuando establece conexión con el dispositivo central.
	Estos parámetros pueden ser leídos por el central y adaptar la conexión conforme a ellos.
Dirección de reconexión	Se usa esta dirección al querer reconectar con un dispositivo privado.
Bandera de privacidad periférica	Controla la privacidad del dispositivo.

4.3 CAPA DEL CONTROLADOR

4.3.1 Capa física

La capa física, es la capa que se encuentra a más bajo nivel en la pila de protocolos de BLE, como se pudo observar en la figura 4.1. Su función consiste en el envío y recepción de datos mediante el uso de radiofrecuencia.

Opera en una banda de 2,4 GHz ISM (Indrustrial, Scientifc y Medical). Esta banda es adoptada por numerosas tecnologías como la inalámbrica (en inglés Wireless), la comunicación de campo cercano o NFC (del inglés Near Field Communication), dispositivos como juguetes a radio control y otras muchas más, lo que lleva inevitablemente a la existencia de interferencias. Para prevenir o disminuirlas, se produce un continuo intercambio de frecuencia, la norma con la

que se produce el salto de frecuencia está predeterminada, de manera que los dispositivos que participen en el intercambio de datos conozcan cual es la frecuencia a la que deben trasladarse.

El rango utilizado por BLE oscila entre 2.400 a 2.483,5 MHz, dividiéndose la franja en 40 canales, comenzando por el 2402 MHz y numerándose del 0 al 39 [10].

4.3.1.1 Canales de radio

Los 40 canales en los que se divide la banda de BLE (a diferencia del Bluetooth clásico el cual se dividía en 79 canales [10], muestran una separación de 2 MHz en lo borde inferior de la banda, y de 3,5 MHz en el borde superior de la misma, este espacio separa los canales de cualquier elemento que se encuentre usando la siguiente frecuencia por la parte inferior o superior. Véase un ejemplo en la figura 4.9.

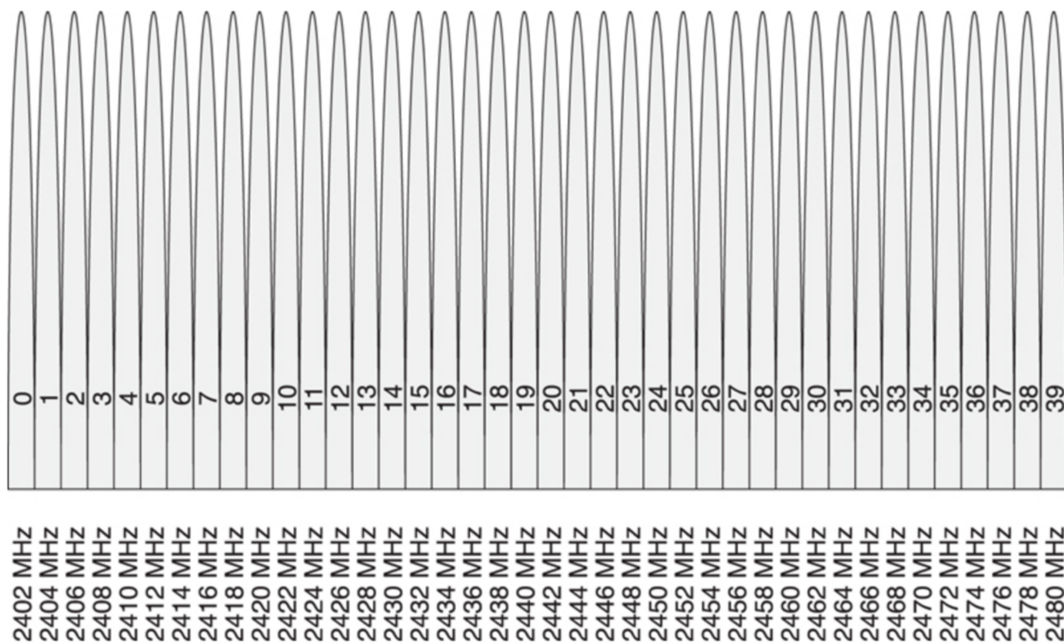


Figura 4.9. Canales de radio en los que se divide la banda de BLE [10].

La frecuencia central de cada canal viene dada por la siguiente ecuación.

$$f(k) = 2{,}402 + k * 2 \text{ MHz},\ k = 0, \ldots, 39 \qquad (4.1)$$

Dónde f, representa la frecuencia central.

4.3.1.2 Modulación

Comenzaremos definiendo a que se hace referencia cuando se habla de modulación, con el fin de ir encuadrando los términos que se expondrán en este apartado.

Cuando se quiere trasmitir información sobre una onda (generalmente, sinodal), se pueden emplear diversas técnicas, las cuales modifican algún parámetro de la propia onda, mediante una señal de entrada calificada como moduladora. Dicho lo anterior, el término modulación engloba el conjunto de técnicas que usan para el mencionado fin, posibilitando que se pueda trasmitir más información de forma síncrona, disminuyendo también el ruido e interferencias que puedan llegar a producirse.

Bluetooth Low Energy, usa la técnica GFSK (Gussian Frecuency Shift Keying), que aúna la técnica FSK, y aplica un filtro Gaussiano para el modulado de la señal.

La modulación por desplazamiento de frecuencia o FSK utiliza dos frecuencias diferentes para cada símbolo que desea trasmitir, variando entre solo dos posibles valores de tensión discretos, formando un tren de pulsos, en el que uno simboliza un “1” o “marca” y otro un “0” o “espacio” [11].

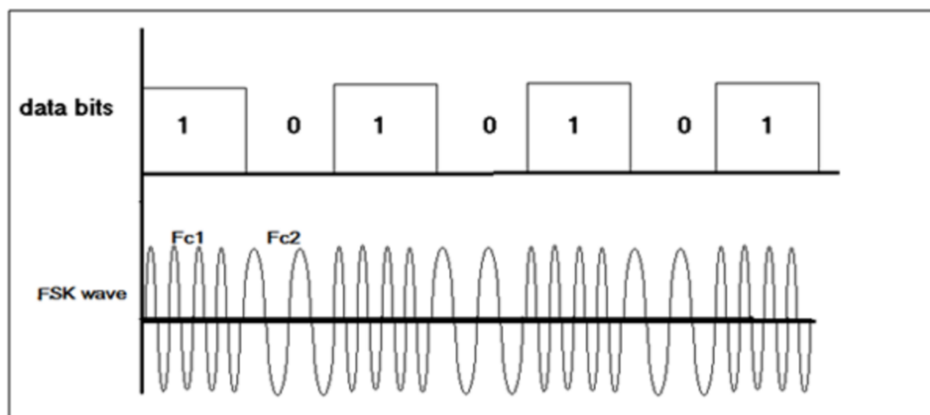


Figura 4.10. Representación de modulación FSK [12].

Como se puede comprobar, la onda modulada, que varía entre -1 y +1 realiza este cambio de forma rápida y drástica, lo que conlleva que haya grandes espectros fuera de banda. Es aquí donde se introduce el filtro gaussiano, el cual permite que las transiciones sean más uniformes, limitando el ancho espectral, para ello, el pulso es modificado, pasando de -1 a +1 de forma progresiva, -1, -97, -92, ..., +92, +97, +1, dónde un 1 lógico es presentado por una desviación positiva, y un 0 mediante una desviación negativa. Usando este nuevo pulso para calcular la frecuencia portadora y reduciendo el espectro fuera de banda [11].

Véase en las figuras 4.10 y 4.11, donde se puede observar de forma gráfica lo antes comentado.

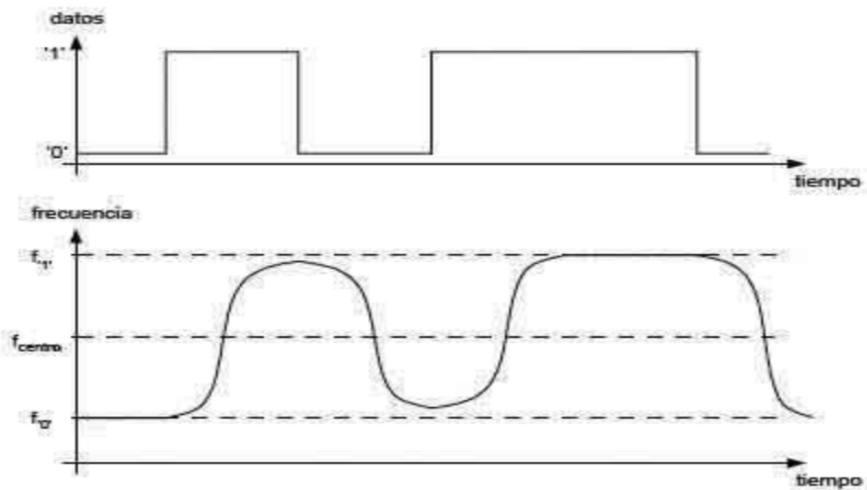


Figura 4.11. Representación de modulación GFSK [13].

4.3.1.3 Potencia y rango de trasmisión.

Recordando que BLE trabaja en la banda ISM de 2.4 GHz, se define un rango de potencia de trasmisión máximo para asegurarse el cumplimiento de la reglamentación que la propia banda impone, siendo este de +10dBm equivalente a 10mW. Del lado contrario, se impone también una potencia mínima, esta vez definida por BLE, con el objetivo de que cualquier dispositivo pueda ser descubierto, se define un valor mínimo de -20dBm, equivalente a 0.01 mW.

Directamente relacionado con la potencia, se encuentra el rango de alcance de la trasmisión, que puede variar de 30m a 100m en función de la intensidad de la potencia.

4.3.2 Capa de enlace

La capa de enlace, situada por encima de la capa física se encarga de diversas labores tales como la negociación y establecimiento de conexión, selecciona la frecuencia de trasferencia de datos, define como dos dispositivos pueden trasferir información entre ellos mediante los canales de radio, proporciona servicios a la capa L2CAP, etc.

El funcionamiento de la capa de enlace viene definido por su máquina de estados, la cual se desgranará y explicará de forma detallada en los aparatados que acontecen.

4.3.2.1 Máquina de estados

La figura 4.12 representa los estados por los que pasa un dispositivo Bluetooth LE.

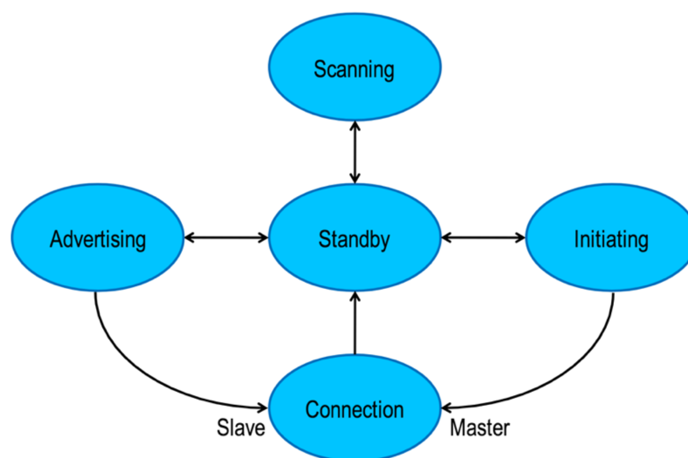


Figura 4.12. Máquina de estados de la capa de enlace [8].

4.3.2.1.1 Estado de espera (Standby)

Como se puede observar en la figura 4.12, el estado de espera es el estado central, en el cual se inicia la capa de enlace de cualquier dispositivo cuando es activada, en este no se trasmiten ni se reciben paquetes y puede ser accedido desde cualquiera de los demás estados.

4.3.2.1.2 Estado de anuncio/difusión

La capa de enlace se encuentra en este estado cuando un dispositivo comienza el proceso de anunciarse, con el fin de ser descubrible para establecer o no conexión. Mientras se encuentre en este estado, el dispositivo emitirá constantemente paquetes de anuncio.

Si un dispositivo quiere enviar paquetes de datos de difusión (broadcast), debe establecerse en este estado. Además de trasmitir, también se mantiene a la escucha, pudiendo responder solicitudes de escaneo provenientes de otros dispositivos. Si se desea finalizar el proceso, tan solo es necesario transitar al estado de espera.

4.3.2.1.3 Estado de escaneo

Se llama escaneo al proceso que inicia un dispositivo central cuando comienza la búsqueda de periféricos, que se encuentren cerca de su área de alcance. Cuando se requiere iniciar dicho proceso, la capa de enlace debe encontrarse en este estado.

Pueden definirse dos sub-estados a partir del principal. El primero de ellos es el escaneo pasivo, cuyo cometido es únicamente recibir paquetes de anuncio provenientes de los distintos dispositivos que se encuentran en su sector. El segundo es el escaneo activo, que además de recibir, envía peticiones de escaneo a cada dispositivo que se vaya descubriendo, para la que también espera una respuesta.

4.3.2.1.4 Estado de iniciación

Cuando se quiere establecer conexión con otro dispositivo, primero debe realizarse cierta formalidad, la cual consiste en transitar al estado de iniciación, en el que se escucha al dispositivo al que se desea conectar, en el momento que se recibe un paquete de anuncio proveniente de este dispositivo, se envía una solicitud de conexión y es entonces cuando se puede avanzar al estado de conexión. Al igual que los anteriores estados, es posible finalizar el proceso, accediendo al estado de espera.

4.3.2.1.5 Estado de conexión

El estado de conexión se puede entender como el estado final al que se accede una vez se ha transitado por los anteriores estados, la finalidad de este es establecer el enlace de trasmisión de datos entre dos dispositivos, haciendo uso

del canal de datos, a diferencia de los anteriores, que se valen del canal de notificación. En la figura 4.12 se contempla que hay dos posibles caminos de acceso. Si se alcanza desde el estado de anuncio, es el periférico el que realiza la transición, siguiendo su rol de esclavo, de lo contrario la acción es ejecutada por el dispositivo central, accediéndose desde el estado de iniciación.

Un dispositivo puede establecer que su capa de enlace disponga de varias máquinas de estado, pero que actúan de forma separada, posibilitando al mismo realizar distintas acciones a la vez, como actuar de dispositivo central, iniciar el escáner de forma pasiva y establecer una conexión, a la vez que envía paquetes de anuncio. Es importante puntualizar que, aunque un dispositivo pueda encontrarse en diferentes estados de forma paralela, nunca podrá interpretar el papel de esclavo y central simultáneamente. Esta delimitación que establece BLE, proporciona cierta sencillez en su implementación, dando lugar a una tecnología muy eficiente.

4.3.2.5 Direcciones del dispositivo

Bluetooth Low Energy, proporciona en su implementación dos tipos de direcciones a usar por los dispositivos que emplean esta tecnología, la dirección pública y la dirección aleatoria, siendo necesario usar al menos una de ellas, para que el dispositivo en cuestión pueda ser identificado.

Añádase esta como una diferencia más entre BLE y el Bluetooth clásico, que tan solo define un tipo de dirección para sus dispositivos.

4.3.2.5.1 Dirección aleatoria

Los principales dispositivos que implementan BLE hoy en día, son aquellos que están en continua iteración con el usuario, son los llamados “wearables” como pueden ser las pulseras de actividad. Esta trasmisión de datos permanente pone a disposición de usuarios malintencionados la posibilidad de rastrear estos dispositivos.

La dirección aleatoria, permite al dispositivo usar una dirección que no es la suya para trasmitir los paquetes de datos, generando una nueva cada cierto tiempo e impidiendo ser rastreado. Solo el receptor seleccionado puede relacionar cada dirección aleatoria con el dispositivo que conectó originalmente y continuar recibiendo datos.

Está compuesta por dos campos, de 48 bits, donde los 24 menos significativos hacen referencia al campo hash, y los 24 más significativos corresponden al campo aleatorio. En el apartado de GAP, que fue explicado anteriormente, se definieron los dos tipos en los que es a su vez diferenciada la dirección aleatoria.

4.3.2.5.2 Dirección pública

La dirección pública del dispositivo hace referencia a una única dirección compuesta por 48 bits y que se encuentra bajo el estándar IEEE. Al igual que la dirección aleatoria, esta muestra dos campos diferenciados, los 24 bits más significativos, pertenecen a la empresa y asignada por la entidad IEEE, para asegurarse que sea diferente para cada empresa, los otros 24 bits designan un identificador único que es impuesto por la empresa y diferente para cada dispositivo.

4.3.2.6 Canales

Tal y como se detalló en el apartado 4.3.1, "*Capa física",* Bluetooth LE presenta 40 canales de radio en la banda 2,4 GHz ISM, pero no todos ellos realizan la misma función, de hecho, la capa de enlace expone dos funcionalidades claras, en las que son asignados.

Defínanse primero los canales de anuncio, cuyo uso radica en el descubrimiento de dispositivos, difusión y recepción de datos e iniciación de conexión.

Son tres los canales que se destinan a esta funcionalidad, véanse en la figura 4.13, los cuales son escogidos por BLE meticulosamente, situándolos a una distancia mínima de 24MHz, con el propósito de que si una zona de la banda ISM sufre interferencias (con radio frecuencias Wifi, por ejemplo), no se vean afectados todos los canales de anuncio.

Por otro lado, los 37 canales restantes, son usados para la trasmisión de datos durante la conexión entre dos dispositivos.

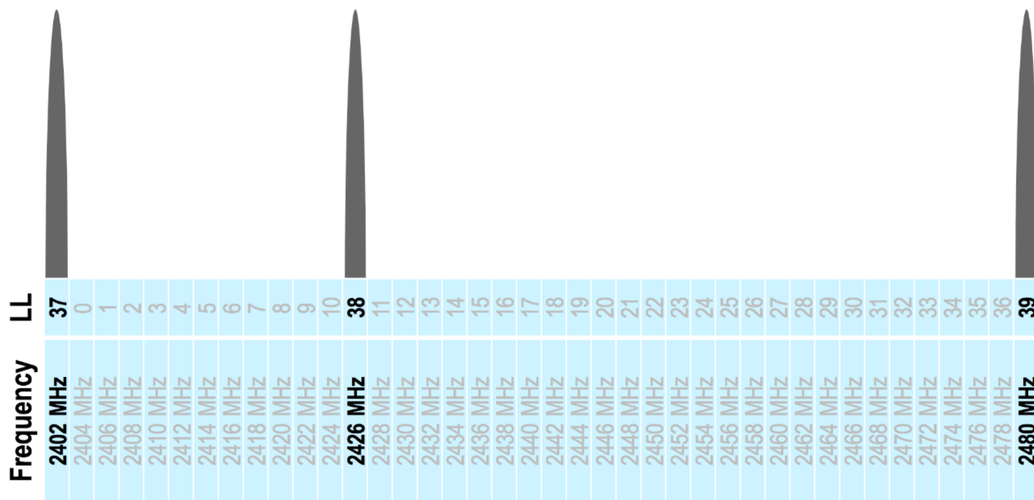


Figura 4.13. Tres canales de anuncio escogidos [8].

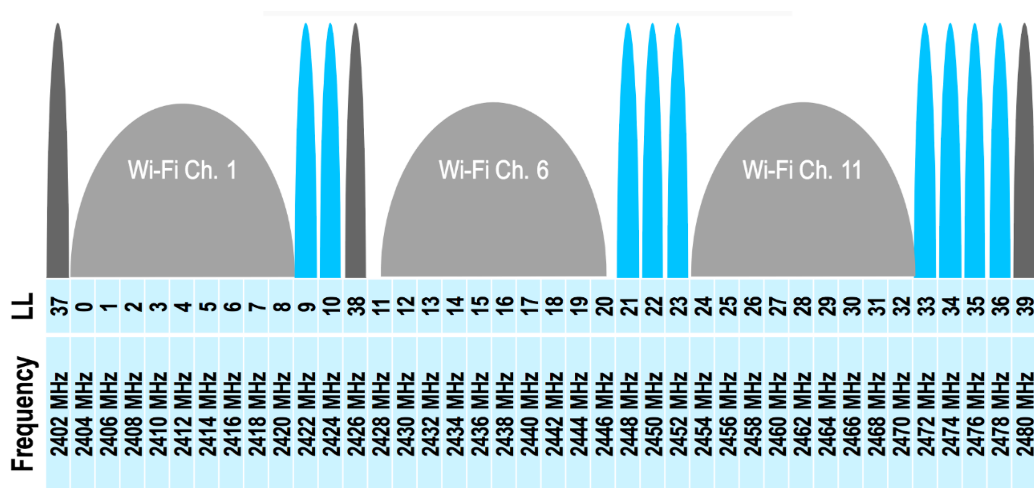


Figura 4.14. Convivencia entre canales de datos, anuncio y wifi [8].

4.3.2.7 Frecuencia de salto adaptable

Bluetooth Low energy define un algoritmo de salto de frecuencia, usado cuando dos dispositivos que se encuentran en una conexión de trasferencia de datos quieren recalcular la próxima frecuencia sobre la que enviar la siguiente trama.

$$f_{n+1} = (f_n + \text{hopIncrement}) \bmod 37 \quad (4.2)$$

Como se observa en la ecuación, se define esta sobre módulo 37, permitiendo que cada frecuencia tenga la misma prioridad de ser empleada. La variable *"hopIncrement"* puede tomar valores entre 5-16 y es empleada para evitar colisiones con otros dispositivos centrales, en los cuales sus frecuencias han

sido calculadas a partir de este valor aleatorio, garantizando que estas difieran en su mayor parte.

El fin último de realizar este salto, es disminuir las interferencias que puedan ocasionarse sobre el canal que se está trabajando. Para que el proceso completo se desarrolle de forma satisfactoria, los dispositivos implicados en la conexión poseen un mapa de canales que han sido filtrados como “buenos” o “utilizables” y “malos” o “inutilizables”, haciendo referencia a las interferencias que presentan. Si alguno de los canales escogidos a partir de la ecuación 4.2 viene clasificado como inutilizable, estos son recalculados por el dispositivo maestro, el cual puede marcar como malos todos aquellos que intuya que muestran interferencias, siendo imperativo que designe al menos dos canales como usables.

4.3.2.8 Formato de paquete

La capa de enlace que presenta BLE, simplifica en gran medida el formato definido para los paquetes, tanto de datos como de anuncio, especificando tan solo un formato que es usado para ambos canales.

Los paquetes, compuestos por bytes de datos, son trasmitidos bit a bit, y presentan seis campos diferenciados. Preámbulo, dirección de acceso, cabecera, longitud, datos y verificación de redundancia cíclica, obsérvese la figura 4.15.

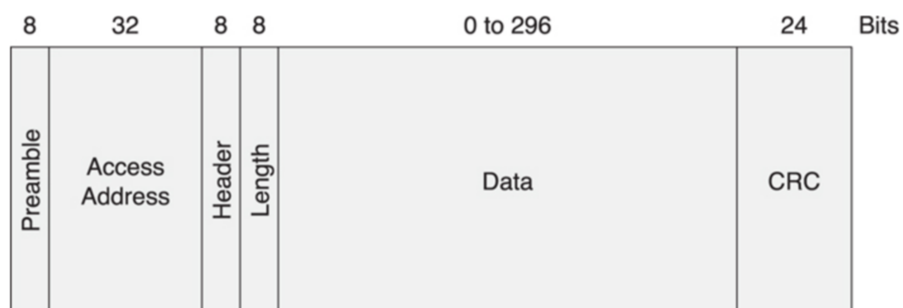


Figura 4.15. Formato de paquete [10].

4.3.2.8.1 Preámbulo

El primer campo que se puede identificar en este formato de paquete es conocido como preámbulo. Consta de 8 bits, los cuales hacen referencia a una cadena alterna de ceros y unos (10101010 ó 01010101), que sirven al receptor para que ajuste su radio de frecuencia, así como su control de ganancia, con el fin de que el resto de campos del paquete sean obtenidos de forma satisfactoria.

4.3.2.8.2 Dirección de acceso

Como se detalló anteriormente, BLE utiliza una cantidad restringida de canales, lo que puede dar lugar a que dos dispositivos que sean indiferentes el uno al otro, se encuentren usando el mismo canal de radiofrecuencia de forma simultánea, es aquí donde juega un papel importante esta dirección, posibilitando que la trasmisión se designe al dispositivo que realmente la está recibiendo. En caso haber varias capas de enlace, la dirección será diferente para cada una de ellas.

El tamaño definido es de 32 bits y cabe la diferenciación entre dirección de acceso de anuncio o de datos. Por otra parte, y con el propósito de que esta medida sea más robusta, la capa de enlace sigue un procedimiento por el cual mantiene una copia de los últimos bits recibidos que le sirven para comprobar si la nueva secuencia de bits que llegan concuerda con la dirección de acceso o con el preámbulo que se esperaban.

Cualquier dispositivo que se encuentre escuchando, puede recibir paquetes de anuncio de diferentes destinatarios, debido a esto la dirección de acceso para los canales de anuncio está predefinida siendo esta 0x8E89BED6. Todo lo contario ocurre con los canales de datos, para ellos la dirección de acceso se genera de forma aleatoria y cambia continuamente para cada nueva conexión entre dispositivos.

4.3.2.8.3 Cabecera

La cabecera de un paquete, por definición general, suele contener la información necesaria para llevar un paquete desde el emisor al receptor de forma

satisfactoria. Formada por un total de 8 bits, la cabecera de la capa de enlace incluye información variada, la cual es dependiente del tipo de paquete (paquete de anuncio o de datos).

En el caso de que el paquete sea de tipo anuncio, se pueden encontrar subtipos dentro de este, que vienen debidamente indicados en la cabecera de este.

Estos subtipos guardan relación directa con los modos de conexión que fueron explicados en el punto 4.2.3. Para cada uno existe una abreviatura estándar impuesta por BLE:

1. Indicación de anuncio general: AVD_IND.
2. Indicación de conexión directa: AVD_DIRECT_IND.
3. Indicación de no conectable: AVD_NONCONN_IND.
4. Indicación de escaneable: AVD_SCAN_IND.
5. Solicitud de escaneo activo: SCAN_REQ.
6. Respuesta de escaneo activo: SCAN_RSP.
7. Solicitud de conexión: CONNECT_REQ.

4.3.2.8.4 Longitud

Determina la longitud de los datos útiles que lleva el paquete, la longitud de este campo varía entre 6 bits cuando el paquete es de anuncio y 5 bits cuando se trata de un paquete de datos.

4.3.2.8.5 Carga útil

Todos los campos que conforman el paquete a enviar son importantes, pero sus funcionalidades son impuestas con el fin último de que la información que porta el campo que da nombre a este apartado se envíe correctamente, esto es debido a que los bits que lo constituyen portan los datos que son realmente provechosos para el dispositivo que actúa como receptor.

4.3.2.8.6 Verificación de redundancia cíclica

Es el último campo que viene en cada paquete y cuyo cometido es la verificación de redundancia, es aplicado sobre el campo de cabecera, longitud y carga útil, con un tamaño de 24 bits es capaz de detectar los errores de bit de número impar de igual modo que los de 2 y 4 bits. Se escoge el tamaño de 24 bits por la gran relación de fortaleza y ahorro de energía que presenta.

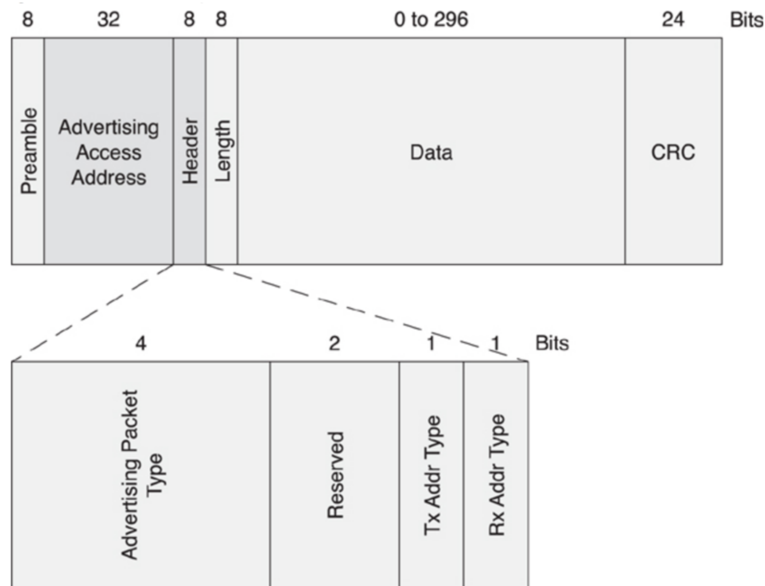


Figura 4.16. Formato de cabecera paquete de anuncio [10].

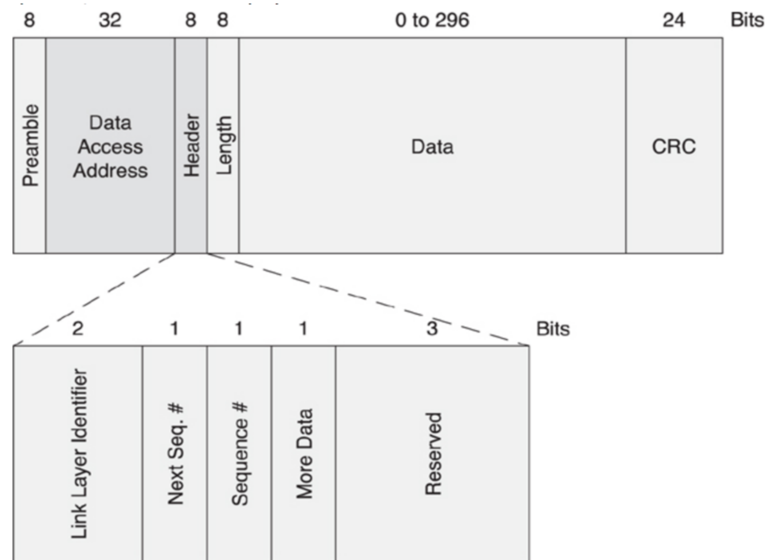


Figura 4.17. Formato de cabecera paquete de datos [10].

4.3.2.9 Topología

Se entiende como topología la manera en la que está diseñada una red de intercambio de datos, para el caso de BLE pueden distinguirse dos marcos distintos.

El primero de ellos, figura 4.18 define una piconet en la que un dispositivo que actúa como central (maestro) se encuentra manteniendo una conexión con varios dispositivos periféricos (esclavos). A diferencia de Bluetooth clásico, BLE no impone una restricción en el número de dispositivos al que puede conectarse el central (siendo siete para Bluetooth clásico), permitiendo su libre gestión, limitada tan solo por la propia capacidad del maestro.

La figura 4.19 referencia el otro escenario posible, en el cual dos dispositivos se encuentran enviando paquetes de anuncio a otro que se encuentra escuchando, en estado de escáner. Este último puede decidir entre reclamar más información a los dispositivos que se anuncian o enviar una petición de conexión.

Nótese que el dispositivo central puede verse inmerso en ambos escenarios. Actuando como central, manteniendo diversas conexiones, así como escáner, escuchando y respondiendo a los distintos paquetes.

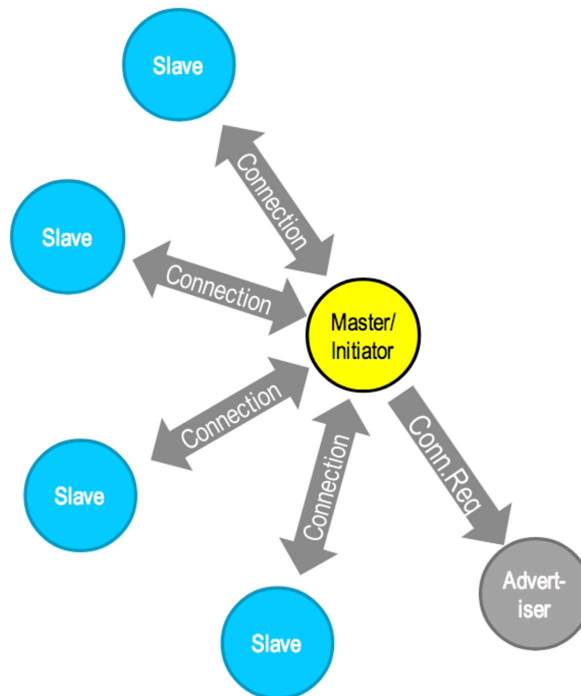


Figura 4.18 Piconet definida por dispositivo central conectado a cuatro periféricos [8].

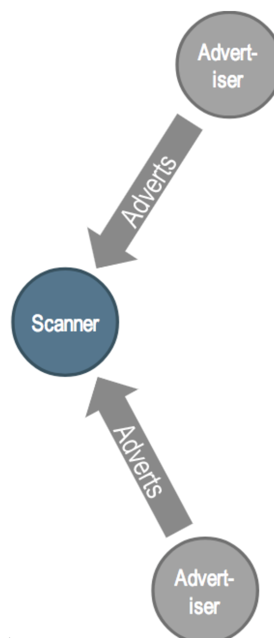


Figura 4.19. Dispositivos anunciándose ante otro que se encuentra escuchando [8].

4.3.2.9 Cifrado

Bluetooth Low Energy también tiene la capacidad de encriptar los datos que se incluyen en la carga útil de los paquetes, permitiendo que individuos malintencionados no puedan intersectar los envíos y tener acceso a su información.

Los paquetes cifrados incorporan un contador de paquete, la cual es una medida usual para impedir los ataques por repetición, también incluyen un indicador de integridad, cuya función es asegurar que los datos provienen del emisor esperado (autenticación).

El algoritmo de cifrado empleado por BLE es AES (del inglés Advance Encryption System), usando 128 bits tanto para longitud de clave como para el tamaño de bloque. El formato de paquete de datos que se vio anteriormente adopta una ligera modificación cuando se quiere añadir el indicador de integridad, añadiendo un nuevo campo entre la carga útil y el CRC.

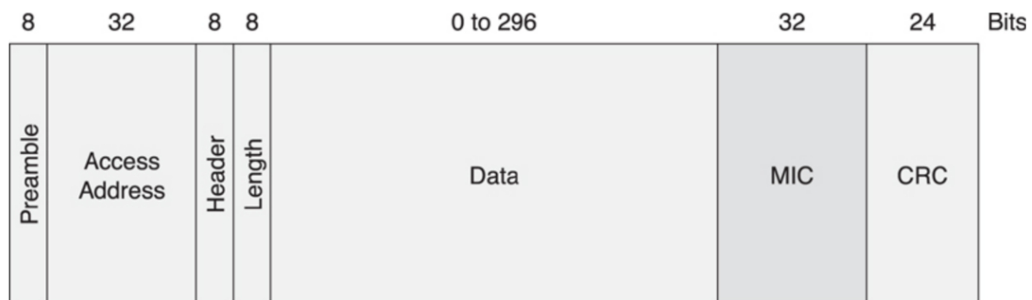


Figura 4.20. Formato de paquete con campo MIC (Message Integriry Check) [10].

4.3.2.10 Escáner y conexión desde el punto gráfico

En este apartado se dispondrán cuatro figuras que ilustran de forma clara cuales son los paquetes intercambiados entre dos dispositivos tanto en el proceso de descubrimiento como de conexión, indicándose el tipo de trama que se envía en cada momento.

-Escaneado pasivo:

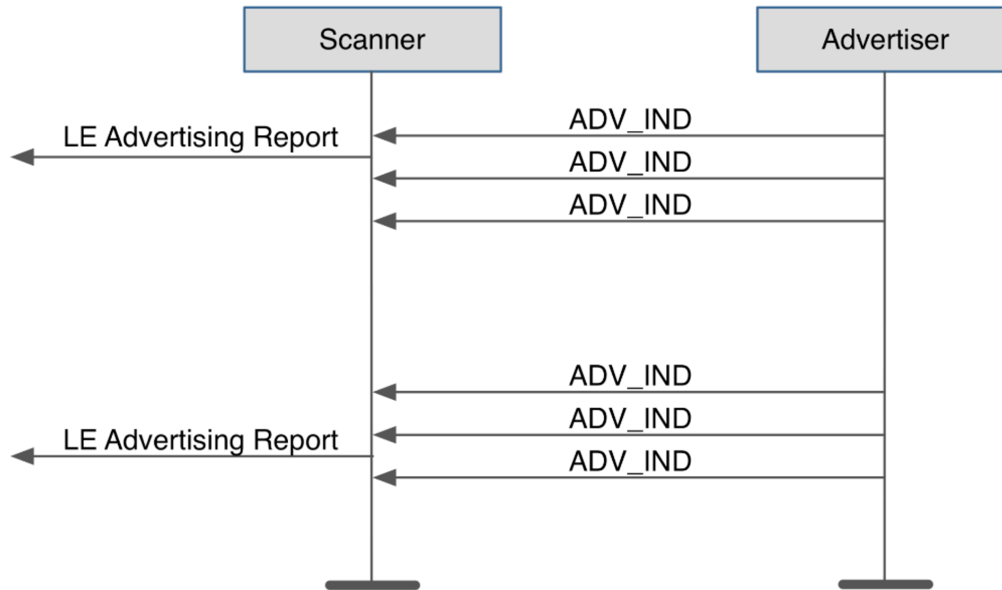


Figura 4.21. Escaneado pasivo, un dispositivo central como escáner y periférico se anuncia [10].

-Escaneado activo:

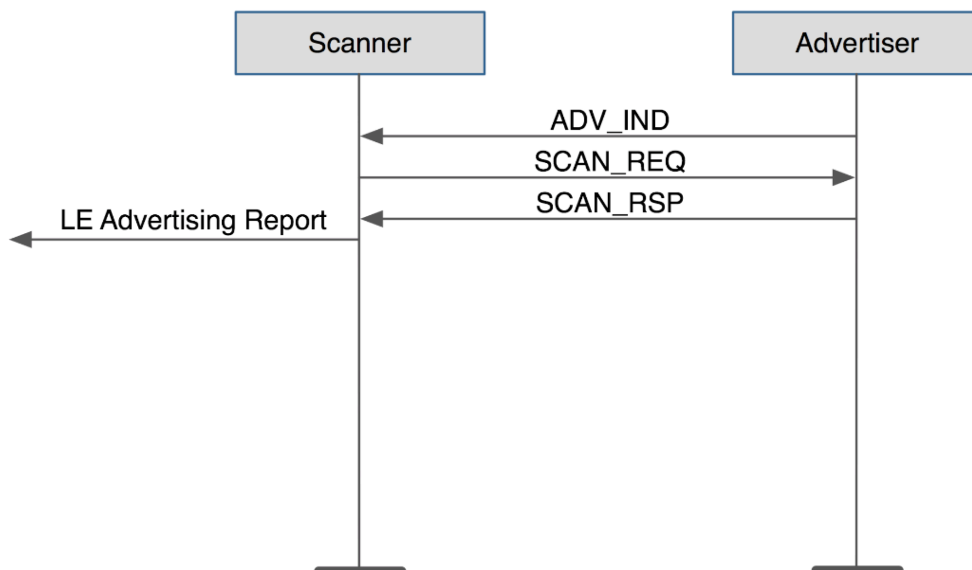


Figura 4.22. Escaneado activo, dispositivo central como escáner y periférico se anuncia [10].

-Iniciación de conexión:

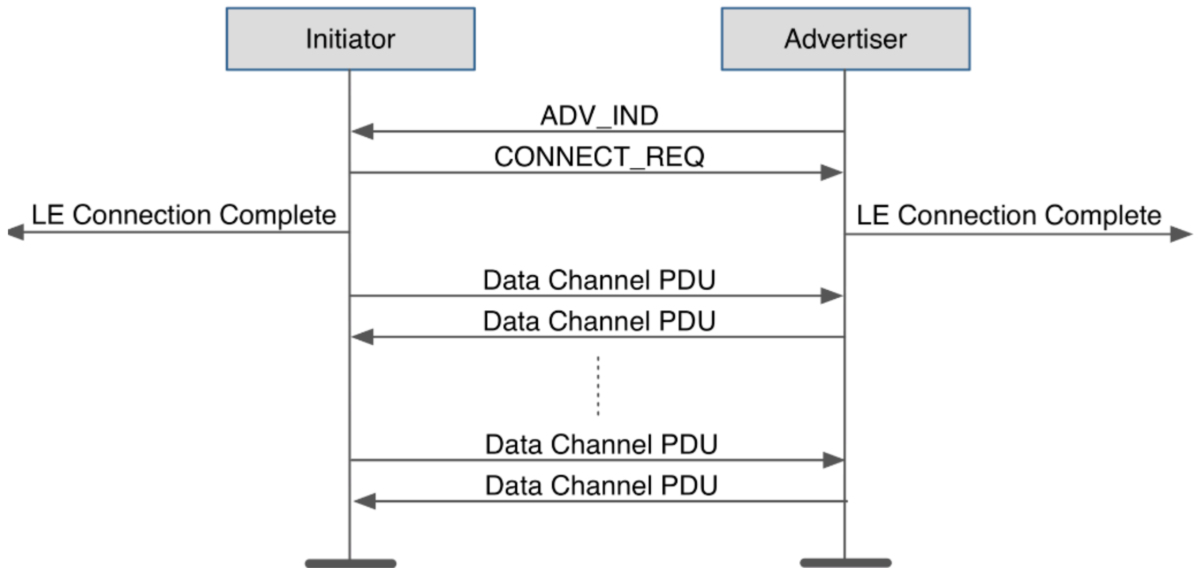


Figura 4.23. Anuncio, petición de conexión y trasferencia de datos entre periférico y central [10].

-Conexión directa:

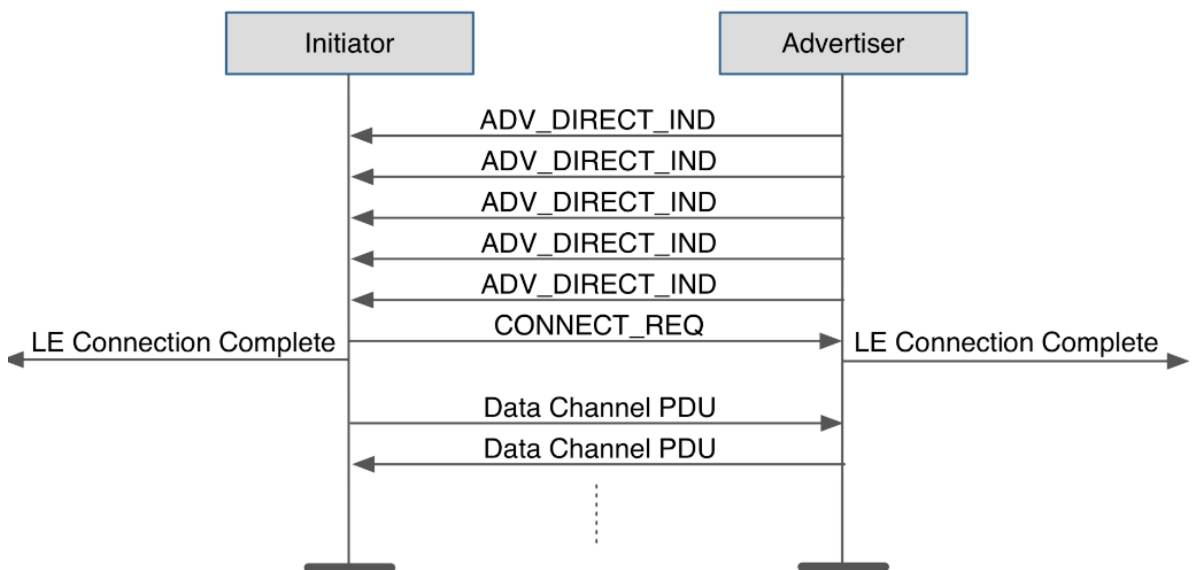


Figura 4.24. Anuncio de conexión, petición e intercambio entre central y periférico [10].

5 HERRAMIENTAS EMPLEADAS EN EL DESARROLLO

Ha sido necesario el uso de una serie de tecnologías tanto software como hardware para la elaboración de este proyecto. Siendo algunas de ellas de uso obligado dada la naturaleza del mismo, mientras otras fueron escogidas por la comodidad y facilidades que aportaban.

5.1 TECNOLOGÍAS SOFTWARE

-Android Studio:

Android Studio es el nombre dado al entorno de desarrollo o IDE (del inglés Integrated Development Environment) proporcionado por Google para llevar a cabo el desarrollo de cualquier aplicación móvil del sistema operativo Android. Se entiende como entorno de desarrollo a aquella aplicación informática que aporta ciertas prestaciones, como un editor de código, depurador, compilador, intérprete entre otras, con la finalidad de facilitar la labor del desarrollador.

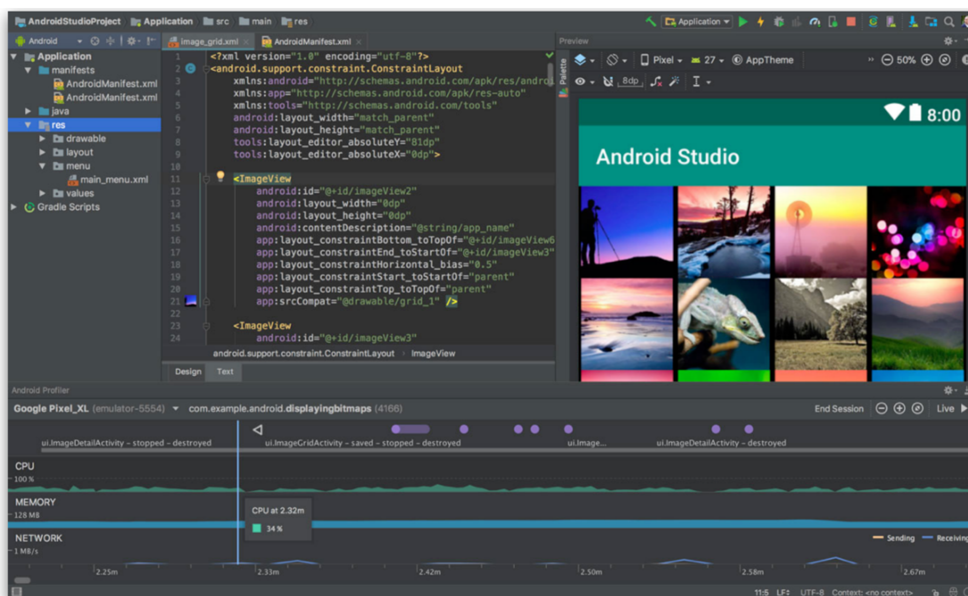


Figura 5.1. Android Studio, IDE de la plataforma Android [14].

Anunciado por Google el 16 de mayo de 2013 y presentado posteriormente en diciembre de 2014, sustituyó a eclipse como IDE principal para el desarrollo de aplicaciones sobre el SO que le da nombre. Comercializado de forma gratuita está presente en las tres principales plataformas como son macOS, Microsoft Windows y Linux.

Se ha empleado este IDE para confeccionar la principal funcionalidad de este proyecto, el código fuente, encargado de dotar de funcionalidad y usabilidad a la API desarrollada. Es un entorno muy manejable, prestando cómodamente las instalaciones que aportan las distintas librerías necesarias para confeccionar cualquier aplicación.

-Visual Paradigm:

Esta es una herramienta usada en este proyecto para la elaboración de diagramas UML (Lenguaje Unificado de Modelado). Puede entenderse como un lenguaje visual usado a manera de plano que facilita el entendimiento de estructura y comportamiento de un sistema software.

Es de uso común elaborar estos diagramas antes, durante y después de la implementación de un proyecto software, aportando una visión general de lo elaborado.

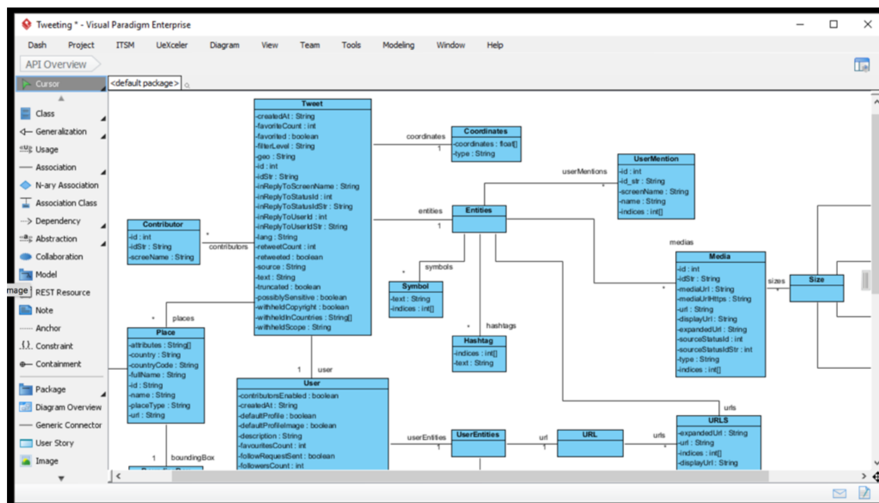


Figura 5.2. Visual Paradigm, herramienta para elaboración de diagramas UML [15].

-Source tree:

Cliente de administración de repositorios Git desarrollado por Atlassian, sencillo de usar gracias a su limpia e intuitiva interfaz, aportando ventajas frente al uso

de la línea de comandos en lo que, al uso de ramas, gestión de commits y push se refiere.

Esta aplicación contribuyó en la gestión de las distintas versiones que constituyeron el trabajo final.

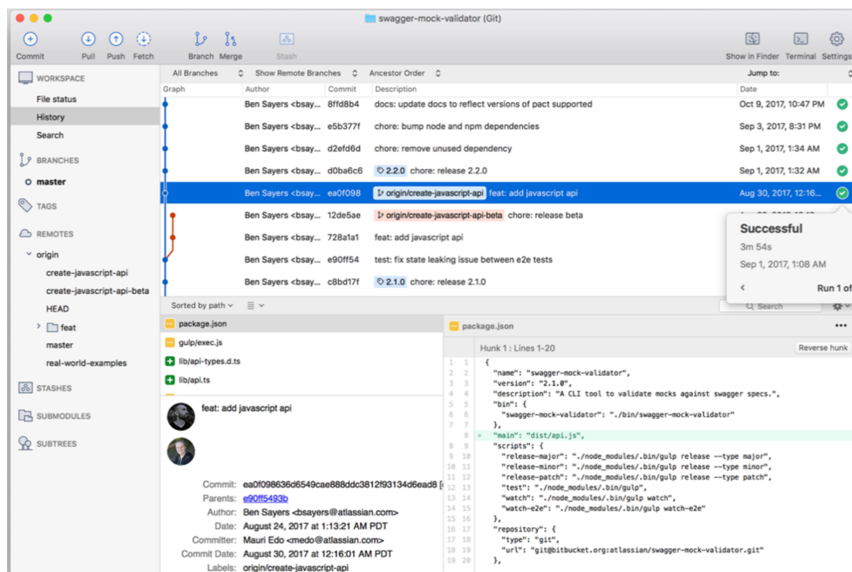


Figura 5.3. Interfaz del control de versiones Source Tree [16].

5.2 TECNOLOGÍAS HARDWARE

-Dispositivo móvil Android:

Para realizar las distintas pruebas de funcionamiento, fue necesario emplear un Smartphone con SO Android (de la marca One Plus) actualizado a la versión necesaria para llevar a cabo dichas comprobaciones, actuando este como dispositivo central.

-Pulsera de actividad:

Como dispositivo periférico se escogió la pulsera de actividad marca nueboo, prestando los servicios de monitorización cardiaca (heart rate) y presión arterial (blood pressure) propios de BLE. Los servicios y características que porta este dispositivo son identificados por UUIDs propios del fabricante, lo que implica un estudio previo para poder trabajar con ellos.

Figura 5.4. Pulsera de actividad nueboo, con tecnología Bluetooth 4.0 [17].

-Banda de actividad:

Banda de actividad Polar H7, como segundo dispositivo esclavo, ayudando a completar el plan de pruebas. Incorpora servicios y características oficialmente adaptados por Bluetooth SIG, con UUIDs estándar facilitados por Bluetooth en su portal web.

Figura 5.5. Sensor de frecuencia cardiaca Polar H7 [18].

6 IMPLEMENTACIÓN DEL PROYECTO

6.1 SISTEMA OPERATIVO MÓVIL ANDROID

Android es el SO móvil más presente en smartphones a nivel mundial, con una cuota de mercado por encima del 80%, dejando muy atrás a su principal competidor, el sistema iOS de Apple. Comenzó siendo desarrollado por la empresa Android Inc, fundada en 2003 por Andy Rubin, Rich Miner, Chris White y Nick Sears en Palo Alto, California, para después ser adquirida por la multinacional Google en 2005.

Presentado en noviembre de 2007, ha recibido numerosas actualizaciones, situándose en la actual 8.0. Su tienda virtual, Google Play, cuenta con más de 2 millones de aplicaciones, demostrando así que hay detrás una gran comunidad de desarrolladores para este SO.

Escrito en java, C y C++ es distribuido bajo licencia Apache, la cual es libre y de código abierto. En el año 2017, Google lanzó una iniciativa llamada Android GO, el cual se centra en adaptar este sistema a dispositivos de gama baja y poder llegar así a más usuarios a lo largo del mundo.

6.2 ANDROID Y BLUETOOTH LOW ENERGY

Android introdujo soporte para Bluetooth Low Energy en su versión 4.3, concretamente en la API 18, significando así que es esta la mínima versión posible para hacer uso de BLE en un dispositivo móvil con SO Android.

Actualmente su API proporciona una larga lista de procedimientos y funciones, que se han ido incrementando en las distintas actualizaciones del sistema. Estas sirven al desarrollador de lo necesario para llevar a cabo todo tipo de implementaciones en las que esté involucrado BLE.

6.2.1 Problemática y objetivos

Como se ha comentado, Android facilita a los desarrolladores multitud de herramientas e información para que puedan trabajar con esta tecnología, además, como es conocido, existe una gran comunidad en torno a este sistema, lo cual repercute en que existan infinidad de portales web, ajenos al oficial, que aportan nuevos conocimientos desde la experiencia.

Sin embargo, existe un inconveniente generalizado en todos estos dominios web, y que no es solventado por ninguno de ellos de forma satisfactoria.

El problema en cuestión se centra en la complejidad presente a la hora de entender cómo funciona BLE, cómo se relaciona con dispositivos móviles, y lo más importante, cómo dar uso a las más de veinte clases proporcionadas por la API de Android, concretamente en el paquete *android.bluetooth.le*, el cual, de partida, puede dar lugar a confusión, pues en la mencionada librería también se encuentra el paquete que da soporte a Bluetooth clásico, con el nombre *android.bluetooth*. El segundo paquete no es excluyente, pues contiene clases importantes y necesarias a la hora del desarrollo, aunque este esté centrado en low energy.

Las figuras 6.1 y 6.2 presentan las principales clases que son necesarias para el desarrollo de una aplicación sobre BLE. Rápidamente, se puede observar la problemática que se intenta solucionar y cuáles son los objetivos a conseguir. Compruébese que la suma de clases que muestran las dos figuras, hacen un total de catorce, perteneciendo las primeras nueve al paquete *android.bluetooth.le*. Véase a continuación una breve explicación de alguna de ellas:

- La clase **Bluetooth Manager** es usada para obtener acceso al adaptador del dispositivo móvil y comprobar que se encuentre activo, **BluetoothAdapter** como se observa en la figura, se relaciona con **BluetoothLeScanner**, el acceso a dicho objeto por medo de esta relación permite que sea usado para buscar dispositivos que se encuentran en el entorno y que se están anunciando**.**
- **ScanFilter** y **ScanSettings** son usadas para añadir filtros de búsqueda, puede filtrarse por UUID de servicios, dirección y nombre del dispositivo, etc.
- Finalmente, **ScanCallBack,** la cual es una clase abstracta que es usada por BluetoothLeScanner para devolver los datos resultantes del escaneo, pudiéndose consultar estos mediante la clase **ScanResult.**

Por otro lado, en la figura 6.2, pueden observarse otras cinco clases, estas pertenecientes al paquete *android.bluetooth*. Las clases **BluetoothGattService, BluetoothGattCharacteristic**, **BluetoothGattDescriptor** y **BluetoothGatt** están directamente relacionadas con el perfil GATT de la pila de protocolos, visto en el punto 4.2.3. El propio nombre de las clases indica con que parámetros de la estructura de BLE se relacionan (Service, characteristic, Descriptor, GATT, …). Aunque son fácilmente reconocibles, ya que han sido explicados detalladamente con anterioridad en este documento, no tiene que ser así para un desarrollador que se expone a trabajar con BLE por primera vez.

Resumiendo, y teniendo en cuenta lo visto en los anteriores párrafos, cualquier desarrollador que desee implementar una aplicación móvil en la que se incluya tecnología BLE, debe primeramente realizar un proceso de estudio y aprendizaje, el cual puede resultar bastante tedioso si esta tecnología representa una funcionalidad secundaria, y es tan solo usada como medio para el objetivo final de la aplicación móvil en cuestión. El objetivo principal de este trabajo será por tanto ofrecer a los desarrolladores Android una API que facilite el desarrollo de la comunicación basada Bluetooth Low Energy

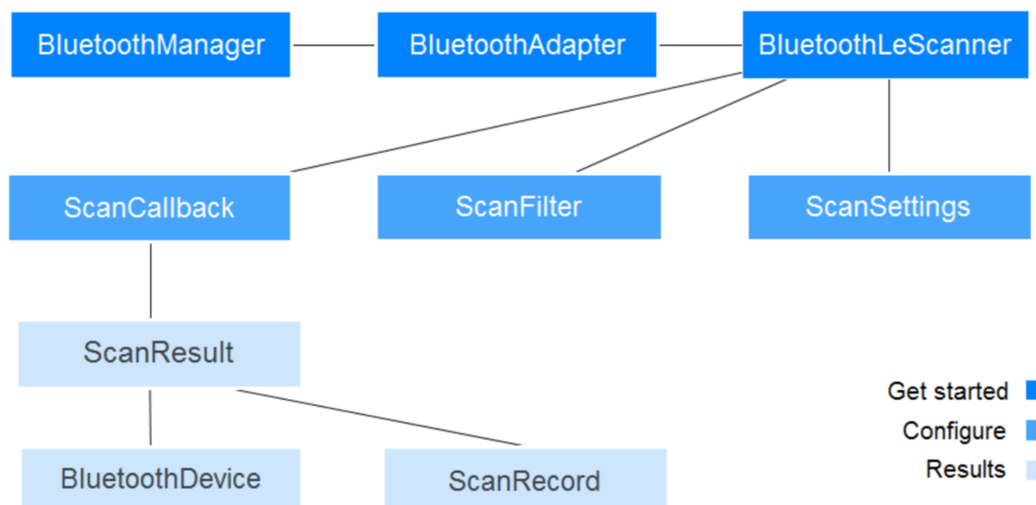


Figura 6.1. Relaciones entre clases del paquete Bluetooth Low Energy en Android [19]

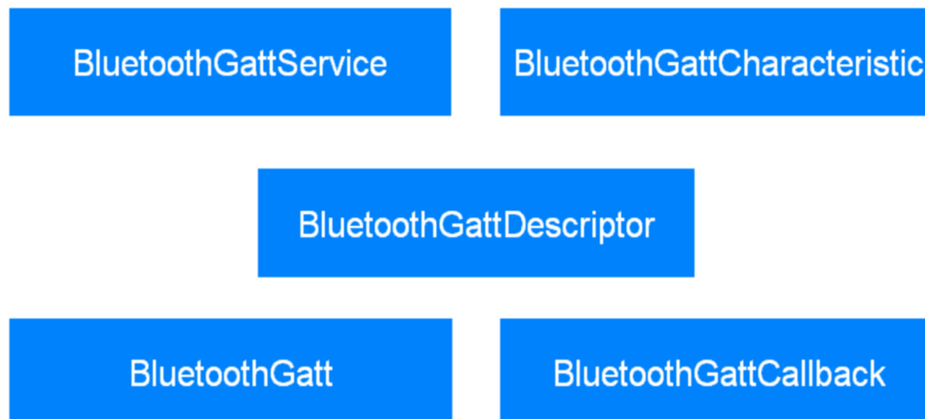


Figura 6.2. Clases del paquete Bluetooth que son útiles para Bluetooth Low Energy [19].

6.2.2 Solución propuesta

A lo largo de este documento, se ha podido comprobar el amplio y complejo funcionamiento que tiene Bluetooth Low Energy, fundamentando así los inconvenientes ya descritos y dejando sin lugar a duda que es necesario aportar una solución al problema.

La solución escogida es la elaboración de una librería de utilidades (API), cuya distinción sea su facilidad de uso (referenciándose en adelante como API-B), intentando solventar los problemas localizados en la API propuesta por Android para la implementación de comunicaciones basadas en Bluetooth Low Energy.

En primera instancia, es posible pensar que, si ya existe una API oficialmente adaptada, es un error continuar por ese camino y que no se llegará a una solución óptima. Este no es un pensamiento erróneo, efectivamente repetir algo existente y que no cubre algunas necesidades requeridas es una equivocación, no obstante, es en la repetición donde reside la parte fundamental de la librería propuesta en este proyecto, precisamente porque se ha realizado todo lo contrario.

El procedimiento seguido no se ha centrado en imitar y mejorar sino en servirse de lo existente para prestar la misma función, pero de distinta forma, más sencilla. La API-B desarrollada actúa como una capa intermedia pensada para ser usada, es decir, proporciona métodos y funciones que se valen de los ya estandarizados, proporcionando al desarrollador resultados finales fáciles de tratar. Esta capa, oculta toda la lógica y complejidad de BLE al programador,

evitando así el estudio y comprensión que es necesario para dar uso a la librería oficial.

En los siguientes apartados se explicará su proceder de forma más técnica y detallada, a fin de que sea entendida en gran medida.

6.2.3 Estructura de la API-B desarrollada

Este apartado se ocupa de describir la estructura de clases y sus relaciones mediante diagramas UML. Comience contemplando la figura 6.3, la cual define la estructura de clases de la capa principal de la API-B. Como puede apreciarse esta cuenta tan solo con 6 clases, de las cuales tres de ellas están definidas como interfaces, definiendo estas el prototipo de las funciones y procedimientos implementados en las clases AbstractBluetoothLowEnergy y BluetoothLow-EnergyScanner. Es importante tener presente que debajo de esta capa sí se implementa la estructura de los paquetes Android vistos en las figuras 6.1 y 6.2, pero en este caso es trasparente al desarrollador, solventando así el problema de estudio y aprendizaje que suponía el esquema inicial.

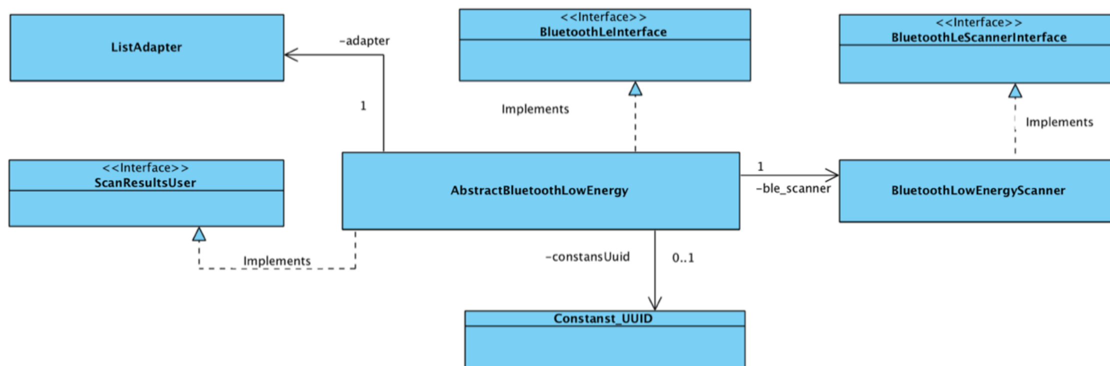


Figura 6.3. Estructura de la capa principal de la API-B.

6.2.3.1 Patrón de diseño

Se entiende como patrón de diseño, aquella técnica software que permite resolver problemas en lo referente al diseño del sistema en el que se trabaja.

Concretamente la API desarrollada se basa, aunque con algunas variaciones, en el patrón de comportamiento llamado ***Command***, cuya función consiste en

encapsular cierta acción en un objeto, dando la posibilidad de que se lleve a cabo dicha acción no siendo necesario que se conozca el contenido de la misma [20].

Esta idea de diseño ha sido utilizada en aquellos métodos que se prestan a uso del desarrollador, sin que este deba tener los conocimientos sobre las clases usadas en su desarrollo.

En cuanto a la estructura en sí, se ha usado como referencia el patrón creacional Abstract Factory, el cual permite crear distintas familias de objetos sin que estas se mezclen entre sí [20]. El concepto se puede ver patente en el uso de interfaces, incluidas con el pensamiento de que la API sea escalable, atendiendo a futuras mejoras como pueden ser añadir formas alternativas al proceso de descubrimiento, para ello se crearía una nueva clase llamada por ejemplo BluetoothLowEnergy2, la cual extiende de la misma interfaz que la primera, pero en la que los métodos realizan sus funcionalidades de distinta manera.

En los siguientes puntos se comentará cada clase con más detalle.

6.2.3.2 ListAdapter

Esta clase es quizás las más independiente de las 6 que se presentan en el diagrama anterior, y es usada para almacenar en un *ArrayList* los dispositivos que son descubiertos cuando se inicia el proceso de escaneo. También incluye procedimientos que pueden realizarse sobre la lista como borrar, añadir, comprobar si contiene un elemento concreto, etc. Mantiene una relación uno a uno con *AbstractBluetoothLowEnergy*.

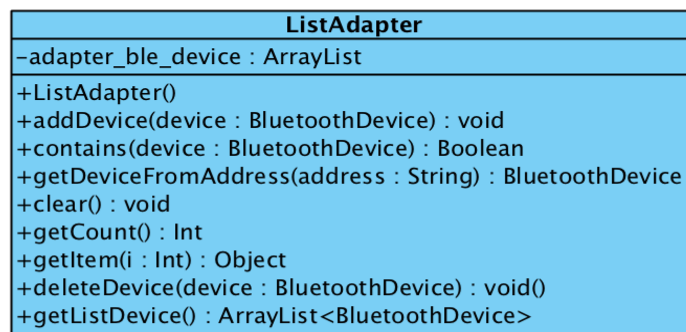


Figura 6.4. Detalle de clase ListAdapter.

6.2.3.3 BluetoothLowEnergyScanner

Es una de las clases principales, encargada de realizar el proceso principal de escaneo de dispositivos (Figura 6.5). Obsérvese que define tres objetos, cada uno de ellos de distinto tipo y que hacen referencia a las clases BluetoothLeScanner, BluetoothAdapter y ScanCallBack. Decir que el desarrollador no interactúa directamente con esta clase, esta es usada de forma intermedia e interna, lo que indica que la lógica de uso necesaria para trabajar con las tres clases mencionadas en este párrafo es ajena a él.

Implementa la clase BluetoothLowEnergyInterface, que facilita y guía su uso, y está relacionada con AbstractBluetoothLowEnergy.

El principal método de esta clase es scannerBleDevice, en cuya implementación se encarga de hacer uso del procedimiento startScan perteneciente a BluetoothLeScanner.

BluetoothLowEnergyScanner
-scanner : BluetoothLeScanner -bluetoohAdapther : BluetoothAdapter -context : Context -scanning : boolean -deviceName : String -scanCallBack : ScanCallback -scanResultsDevices : ScanResultsUser -handler : Handler -scanOn : boolean -TIME_SCAN : long -TAG : String
+BluetoothLowEnergyScanner(context : Context) +scannerBleDevice(scanResult : ScanResultDevices, timeSca... +stopScanDevice() : void() +isScanOn() : boolean +setScanOn() : void

Figura 6.5. Detalle de la clase BluetoothLowEnergyScanner.

6.2.3.4 Clases de tipo interfaz y de constantes

Las clases BluetoothLowEnergyInterface, BluetoothLeScannerInterface y ScanResultsUser conceden una visión limpia, documentada y generalizada de todos los sistemas a los que se le puede dar uso, es decir, funciona a modo de catálogo, ya que la ausencia del código explicito permite una fácil lectura, véase

la Figura 6.8. Para utilizarlas tan solo es necesario implementarlas en las clases correspondientes.

Además de las anteriores, se puede contemplar una clase de constantes, que contiene una colección de valores de tipo String, referenciando los principales identificadores únicos (UUIDS) de servicios y características adaptados oficialmente por GATT.

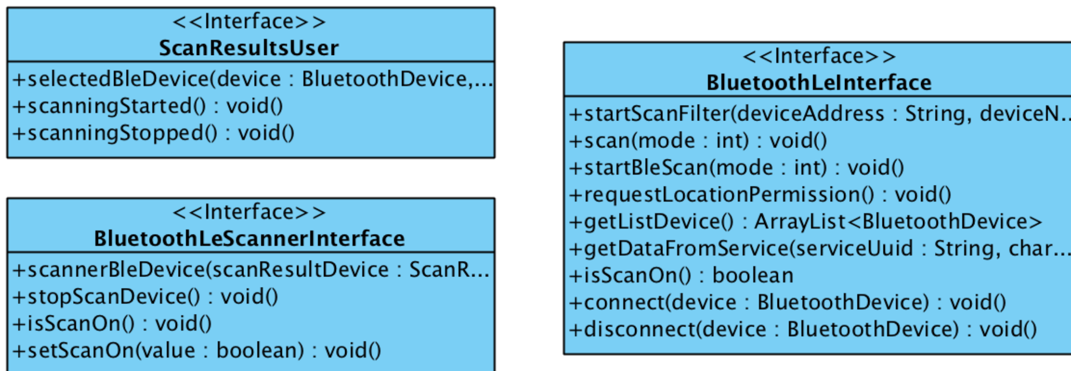


Figura 6.6. Detalle de clases de tipo interfaz.

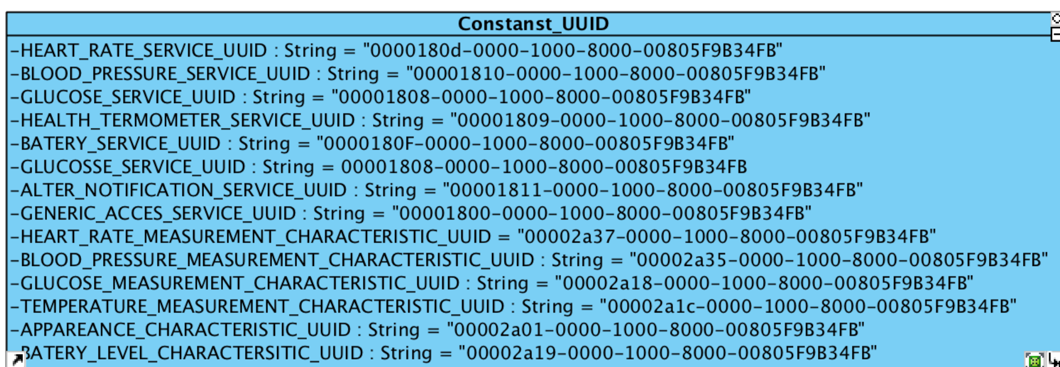


Figura 6.7. Detalle de clase Constans_UUID. Identificadores de servicios y características.

6.2.3.5 AbstractBluetoothLowEnergy

Para finalizar, véase la que puede definirse como la clase principal, ya que es esta con la que interactúa directamente el desarrollador. Presenta numerosos procedimientos que realizan todo el trabajo lógico necesario, dejando de lado del programador la simple decisión de cuándo y dónde usar los mismos.

Se decidió que fuera una clase abstracta debido a que incluye cuatro métodos los cuales no presentan codificación, permitiendo que sea el desarrollador el que decida como desea que se comporten. Estos métodos son:

selectedBleDevices

- Recibe información de un dispositivo cuando es descubierto, permitiendo que se puedan listarse sus datos a la par que se va descubriendo o por lo contrario no realizar ninguna acción. Como se ha comentado, se deja a libre decisión.

parseCharacteristicValue:

- Este método recibe los valores de datos que son enviados por el periférico. Se presentan en array de bytes, legando al programador la tarea de tratarlos en conveniencia.

informationDeviceRead:

- Obtiene un listado de los servicios y características que posee un dispositivo cuando se establece conexión con él.

readRssi:

- Lee la intensidad de la señal, necesaria a veces para conocer si el establecimiento de conexión será exitoso.

Estos cuatro métodos son llamados cuando su respectivo Callback es activado, devolviendo en ellos la información que se ha indicado.

Esta clase implementa dos interfaces y está relacionada con BluetoothLow-EnergyScanner y con ListAdapter, clases que proporcionan distintas funciones que son necesarias para que esta clase abstracta pueda cumplir con sus funcionalidades.

Como se ha podido contemplar, la API-B permite al desarrollador llevar a cabo diversas funcionalidades para trabajar con la tecnología BLE tales como descubrir, conectar, desconectar, trasferir datos, etc. Pero siempre desde la comodidad de ofrecer simples métodos para cada una de estas acciones. Es una librería usada como fin, es decir, su uso está centrado en obtener resultados de una forma rápida y directa. Todo lo contrario, ocurre con la librería proporcionada por Android, la cual permite comenzar el desarrollo desde muy bajo nivel, usándose como medio para desarrollar métodos más personalizados. Es ideal para desarrolladores con amplios conocimientos en la estructura y

funcionamiento de BLE o para aquellos que deseen implementar una compleja aplicación cuyos cimientos estén sustentados por esta tecnología.

AbstractBluetoothLowEnergy

-ble_scanning : Boolean
-ble_scanner : Bluetooth_LE_Scanner
-adapter : ListAdapter
-SCAN_TIMEOUT : Long
-REQUEST_LOCATION : Long
-permissionGranted : Boolean
-deviceCounter : Int
-values : BluetoothDevice[]
-contextActivity : Context
-bleScanner : BluetoothLowEnergyScanner
-gatt : BluetoothGatt
-mainActivity : Activity
-deviceAddress : String
-deviceName : String
-TAG : String
-deviceServiceUuid : ParcelUuid
-constantsUuid : Constants_UUID
-serviceUuid : UUID
-characteristicUuid : UUID
-listServices : List<BluetoothGattService>
-listCharacteristic : List<BluetoothGattCharacteristic>
-mapServiceCharacteristic : Map<BluetoothGattService, List<BluetoothGatt...

+selectedBleDevice(device : BluetoothDevice, scan_record : byte[], rssi : int...
+startScanning() : void
+scanningStarted() : void
+scanningStopped() : void
+setScanState(value : boolean) : void
+requestLocationPermission() : void
+onRquestPermissionsResult(requestCode : int, permissions : String[], gra...
+simpleToast(message : String, duration : int)
+AbstractBluetoothLowEnergy(context : Context, activity : Activity)
+startScanFilter(deviceAddress : String, deviceName : String, serviceUuid : ...
+scan(mode : int) : void()
+startBleScan(mode : int) : void()
+getListDevice() : ArrayList<BluetoothDevice>
+getDataFromService(serviceUuid : String, charactertisticUuid : String, devi...
+parseCharacteristicValue(arrayBytesValue[]) : void()
+InformationDeviceRead(listServices : List<BluetoothGattService>, mapSer...
+readPhy(gatt : BluetoothGatt, txPhy : int, rxPhy : int, status : int) : void()
+connect(device : BluetoothDevice) : void()
+disconnect(device : BluetoothDevice) : void()
+isScanOn() : boolean
+requesrLocationPermission() : void()
+onRequestPermissionResult(requestCode : int, permissions : String[], gra...
+sendNewData(serviceUuid : String, characteristicUuid : String, value : boo...
+getCharacteristicProperties(characteristic : BluetoothGattCharacteristic)
+selectedBleDevices(device : BluetoothDevice, scan_record : byte[], rssi : i...

Figura 6.8. Detalle de clase AbstractBluetoothLowEnergy.

6.2.4 Modelo de uso

Véase en este apartado una ejemplificación de uso de la API desarrollada, con el objetivo de que se entienda correctamente su finalidad y se emplee en consecuencia.

6.2.4.1 Diagrama de secuencia

Comiéncese por observar el diagrama de casos de uso que se dispone a continuación (figura 6.7), dónde se explica un cierto proceder para el uso de

algunas de sus funcionalidades, sirviendo de guía para la implementación de cualquier aplicación que haga uso de esta librería.

Se ha optado por el proceso referente al descubrimiento y conexión, ya que son los procedimientos iniciales para cualquier implementación.

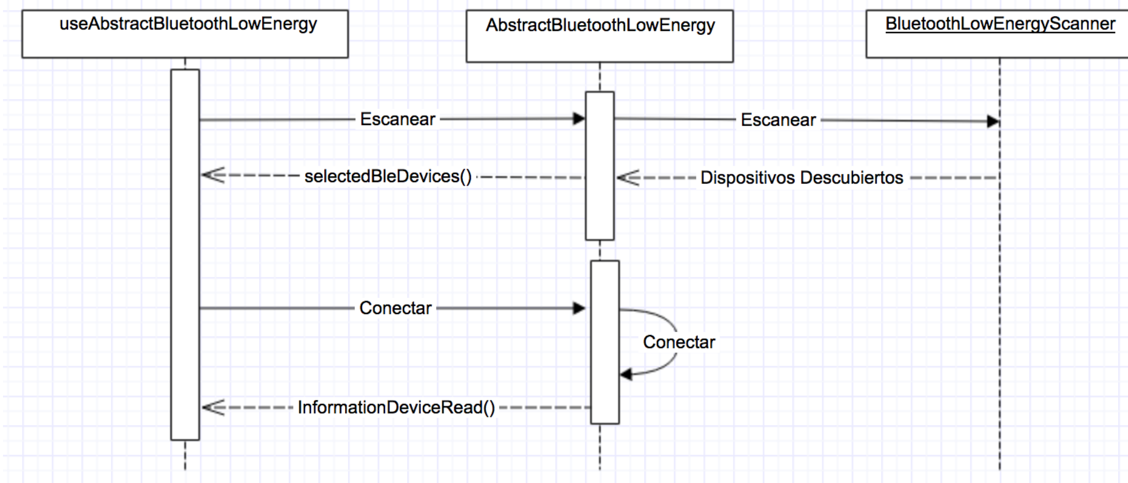


Figura 6.9. Diagrama de secuencia para el descubrimiento y conexión de dispositivos.

En la figura 6.7, se puede contemplar en primera instancia el uso de una clase que no ha sido especificada previamente (useAbstractBluetoothLowEnergy). Esto es así porque la clase mencionada, no es propia de la API implementada, aunque su uso viene generado por esta de forma indirecta.

La clase useAbstractBluetoothLowEnergy, como su propio nombre indica, se utiliza para dar uso a los métodos propios que proporciona la clase Abstract-BluetoothLowEnergy, es decir, en términos más técnicos, la primera extiende de la segunda. Esta acción es necesaria ya que como se comentó en el apartado 6.2.3.4, la segunda clase es de tipo abstracta, dejando a disposición del desarrollador una serie de métodos para su libre implementación.

En el proceso de escaneo, la nueva clase hace uso del método desarrollado de la clase que extiende y esta a su vez de forma interna, inicia el proceso de escaneo disponible en la clase BluetoothLowEnergyScanner. Una vez completado este flujo, los dispositivos que van siendo descubiertos pueden obtenerse en el método selectedBleDevices.

Para realizar la conexión con un dispositivo, se sigue un procedimiento análogo, con la salvedad de que en este caso no es necesario que intervenga la tercera clase vista en la figura 6.7. De la misma manera, cuando se ha establecido

conexión, se retornan una serie de datos del dispositivo, accesibles desde el método InformationDeviceRead.

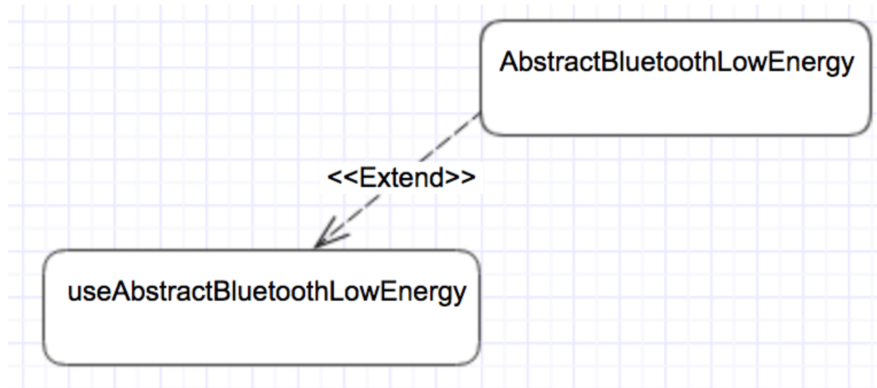


Figura 6.10. Es necesaria una nueva clase que extienda de AbstractBluetoothLowEnergy.

6.2.4.2 Ejemplificación de uso mediante App.

Para especificar aún más las pautas a seguir en el uso de la API, se ha desarrollado una simple aplicación que realiza las dos funciones comentadas previamente, también se incluirán en este apartado pequeños fragmentos de código para completar el modelo de forma más técnica.

La primera vista tiene un análisis simple, pues en esta primera no se realiza ninguna acción en primera instancia, tan solo se crean los distintos objetos de clase y se inicializan adecuadamente, además de crear la clase auxiliar necesaria, la cual se encargará de llevar a cabo los procedimientos principales, así como de recoger los distintos resultados que estos generan.

Figura 6.11. Vista inicial de aplicación de ejemplo.

Debe puntualizarse que son necesarios más fragmentos de código para realizar la aplicación que se verá en este punto, pero son ajenos a la funcionalidad propia de Bluetooth Low Energy, por lo que no incluirán estos en la explicación que sigue a continuación.

```
/* Código de API */
private useAbstractBluetoothLowEnergy bluetoothLowEnergy;
private Constants_UUID constants_uuid;
```

Figura 6.12. Se declaran los objetos de clase que serán necesarios.

```
/* Codigo API */
bluetoothLowEnergy = new useAbstractBluetoothLowEnergy(context, activity: this);
constants_uuid = new Constants_UUID();
```

Figura 6.13. Se inicializan los objetos de forma adecuada.

```java
public class useAbstractBluetoothLowEnergy extends AbstractBluetoothLowEnergy {

    private MainActivity activity;

    public useAbstractBluetoothLowEnergy(Context context, Activity activity) {...}

    @Override
    public void parseCharacteristicValue(byte[] arrayBytesValue) {}

    @Override
    public void selectedBleDevice(BluetoothDevice device, int rssi, byte[] scan_record) {...}

    @Override
    public void informationDeviceRead(List<BluetoothGattService> listServices, Map<BluetoothGattService,
            List<BluetoothGattCharacteristic>> mapServiceCharacteristic) {...}
```

Figura 6.14. Creación de la clase useAbstractBluetoothLowEnergy.

El siguiente paso lógico es pulsar sobre el botón “SCAN” que está posicionado en la barra de tareas. Este iniciará la fase de escaneo de dispositivos, valiéndose de uno de los dos métodos proporcionados para esta tarea (uno de ellos permite filtrar por nombre y/o dirección del dispositivo y por UUID del servicio principal).

A medida que van siendo descubiertos, los dispositivos son recogidos por el método *selectedDevice*, que además del dispositivo, aporta la intensidad de la señal (rssi) y un array de datos en formato de bytes. Este método, como ya se ha explicado, es de libre implementación, eligiéndose en el caso de la aplicación prestada como ejemplo, añadir cada dispositivo a una lista, usada posteriormente para exponerlos en la vista principal. Véase esto en las figuras 6.13 y 6.14.

```java
//Si se pulsa en scan, se inicia el escaneo.
bluetoothLowEnergy.scan( mode: 0);
bluetoothLowEnergy.startScanFilter( deviceAddress: "BL-P1", deviceName: null, serviceUuid: null, mode: 0);
```

Figura 6.15. El procedimiento scan y startScanFilter inician el proceso para descubrir dispositivos.

```java
public void selectedBleDevice(BluetoothDevice device, int rssi, byte[] scan_record) {
    /* Dispositivos que van siendo descubiertos */
    if(!activity.fragmentPrincipal.listDevices.contains(device)){
        activity.fragmentPrincipal.listDevices.add(device);
        activity.fragmentPrincipal.adapter.notifyDataSetChanged();
    }
}
```

Figura 6.16. Implementación del método selectedBleDevice.

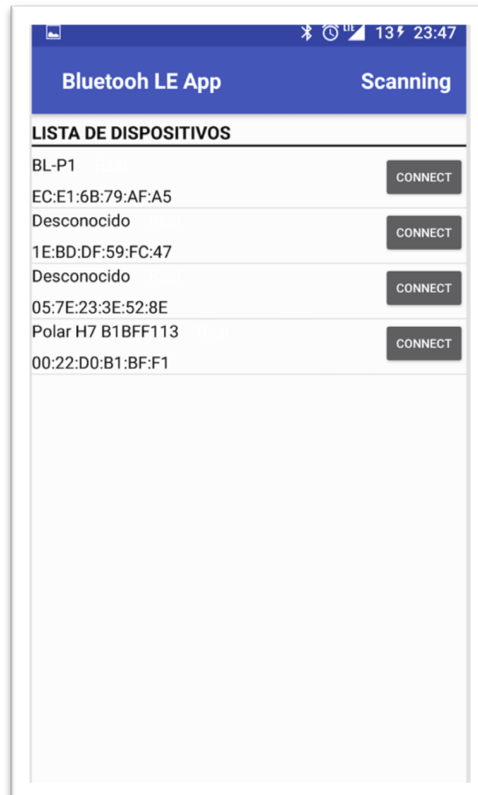


Figura 6.17. Vista que lista los dispositivos descubiertos.

Se listan una serie de dispositivos, mostrando su nombre y dirección, dejando como desconocidos aquellos dispositivos que no permiten leer o no tienen asignado un nombre específico.

Una vez se tenga en la colección el dispositivo al que se desea conectar, tan solo es necesario pulsar sobre el botón “connect”, dando comienzo al proceso de conexión entre el dispositivo central y el periférico, en este caso, el periférico escogido es “Polar H7” (sensor de ritmo cardiaco).

```
/* Codigo API */
bluetoothLowEnergy.connect(device);
```

Figura 6.18. El procedimiento connect comienza con el proceso de conexión.

Cuando se establece la conexión con el dispositivo, pueden leerse una serie de datos de este, que son recogidos en el método *informationDeviceRead*. Para la aplicación en cuestión, se obtienen la lista de servicios del dispositivo y se presentan en pantalla.

```java
public void informationDeviceRead(List<BluetoothGattService> listServices, Map<BluetoothGattService,
        List<BluetoothGattCharacteristic>> mapServiceCharacteristic) {
    /* Cuando connectamos leemos datos del device */
    activity.listServices = listServices;
    activity.gotoFragmentServices();
}
```

Figura 6.19. Implementación del método informationDeviceRead.

En resumen, el apartado 6.2.4 deja patente la funcionalidad real que puede darse a la librería desarrollada, puesto que si realiza una revisión detenida de los fragmentos de código que han sido incluidos, se verá de forma clara como en ningún momento es necesario conocer el funcionamiento y estructura de BLE, para poder realizar una aplicación que le de uso.

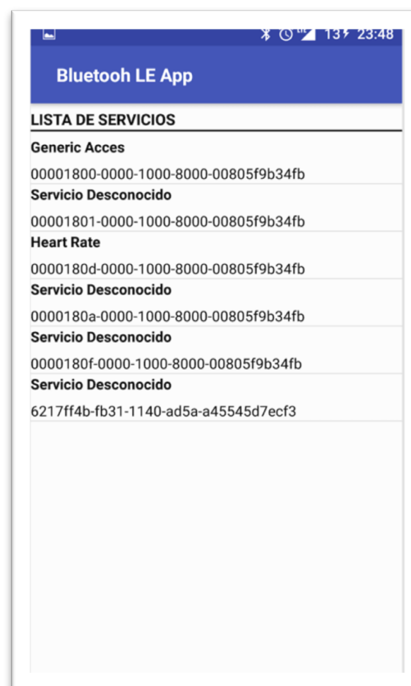


Figura 6.20. Vista que lista los servicios del dispositivo al que se ha conectado.

De igual forma en algunos de estos fragmentos, puede observarse cómo se hace uso de clases propias del paquete de Bluetooth, aportado por la librería nativa de Android, tales como BluetoothGattService, BluetoothGattCharacteristic, BluetoothDevice, no siendo necesario conocer para qué ni cómo se usan, pues se aplican de manera interna y ajenas al desarrollador (véase el ANEXO II. Documentación, donde se dispone la documentación Javadoc de la librería, para obtener una visión más técnica y que ayude aún más a la comprensión de todas las clases explicadas en los anteriores puntos).

7 CONCLUSIONES

El desarrollo de aplicaciones móviles ha supuesto una revolución tecnológica de una escala inmensurable, recordando a la revolución industrial que surgió en Gran Bretaña en el siglo XVIII, concretamente en el ámbito social, en el que las aplicaciones móviles han transformado la vida cotidiana de millones de personas, permitiéndoles realizar una infinidad de actividades tanto de forma individual como en comunidad, y todo esto mediante dispositivos como los smartphone, los cuales no son de un tamaño mayor que la palma de nuestra mano.

Esta nueva revolución sufrirá en los años venideros su propia revolución interna, por la cual, su escala de alcance se incrementará tanto en lo social como en el ámbito de los objetos. El culpable de estos acontecimientos será la Internet de Las Cosas, cuya tecnología unificará lo personal con lo material hasta puntos insospechados, generando un ecosistema totalmente intrusivo, pero a su vez útil y necesario para fomentar el continuo desarrollo social, tecnológico y económico.

Es aquí donde juega un papel fundamental la tecnología Bluetooth Low Energy, pues es hoy en día la más implementada para las relaciones dispositivo-persona. Tendrá una gran importancia en las tecnologías futuras y es por ello por lo que se debe de una u otra manera fomentar su uso en las distintas aplicaciones móviles, conllevando de forma inevitable a que su desarrollo continúe. Un buen ejemplo de esto puede ser los Beacons, dispositivos pequeños y dotados de la tecnología BLE, utilizados para la trasferencia de datos por proximidad, tal como ocurren en centros comerciales dónde estos dispositivos envían ofertas personalizadas a cada cliente directamente a sus dispositivos móviles.

Se espera que la API desarrollada en este proyecto, pueda contribuir en este progreso, acercando y facilitando su implementación, permitiendo a desarrolladores nobeles introducirse en este campo.

Como línea futura, las mejoras aplicables se centran en la actualización, es decir, es importante mantener la librería en concordancia con las versiones venideras tanto de Bluetooth como de Android, añadiendo y mejorando sus procedimientos en función de las nuevas y distintas posibilidades que se presten.

8 REFERENCIAS

[1] L. Ortega, «Bluetooth 5: Características, funciones y dispositivos,» [En línea]. Available: https://www.androidpit.es/bluetooth-5-caracteristicas-funciones-dispositivos. [Último acceso: junio 2018].

[2] Developex, «What is Bluetooth Low Energy (BLE)?,» [En línea]. Available: https://developex.com/blog/what-is-bluetooth-low-energy-ble/. [Último acceso: junio 2018].

[3] L. Looper, «A Short History of the Bluetooth BLE standard, BLE Beacons, and GATT,» [En línea]. Available: https://www.silabs.com/community/blog.entry.html/2016/07/13/a_short_history_oft-mjXj. [Último acceso: junio 2018].

[4] J. Pollicino, «Bluetooth SIG unveils Smart Marks, explains v4.0 compatibility with unnecessary complexity,» [En línea]. Available: https://www.engadget.com/2011/10/25/bluetooth-sig-unveils-smart-marks-explains-v4-0-compatibility-w/. [Último acceso: junio 2018].

[5] Wikipedia, «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Bluetooth. [Último acceso: junio 2018].

[6] J. Tosi, F. Taffoni, M. Santacatterina, R. Saninno y D. Fornica, «Perfomance Evaluation of Bluetooth Low Energy: A Systematic Review,» Sensors(Basel), Switzerland, 2017.

[7] Bluetooth Low Energy, «makespacemadrid,» [En línea]. Available: http://wiki.makespacemadrid.org/index.php?title=Bluetooth_Low_Energy. [Último acceso: junio 2018].

[8] R. Heydon, «An Introduction to Bluetooth low energy,» [En línea]. Available: https://datatracker.ietf.org/meeting/interim-2016-t2trg-02/materials/slides-interim-2016-t2trg-2-7. [Último acceso: junio 2018].

[9] R. Davidson, Akiba, C. Kubi y K. Townsend, Guetting Started with Bluetooth Low Energy, Sebastopol: O'Reilly Media, Inc, 2014.

[10] R. Heydon, Bluetooth Low Energy: The Developer's Handbook, Crawfordsville: Prentice Hall, 2012.

[11] W3eacademy, «Modulación de frecuencia,» [En línea]. Available: http://www.es.w3eacademy.com/wiki/Frequency-shift_keying . [Último acceso: junio 2018].

[12] T. Agarwal, «Know About FSK Modulation and Demodulation with Circuit Diagram,» [En línea]. Available: https://www.elprocus.com/fsk-modulation-demodulation-circuit-diagram/. [Último acceso: junio 2018].

[13] Wikipedia, «Modulación por desplazamiento de frecuencia gausiana,» [En línea]. Available: https://es.wikipedia.org/wiki/Modulaci%C3%B3n_por_desplazamiento_de_frecuencia_gausiana. [Último acceso: junio 2018].

[14] Android, «Developers, Android Studio,» [En línea]. Available: https://developer.android.com/studio/. [Último acceso: junio 2018].

[15] Visual Paradigm, «Visual Paradigm Software,» [En línea]. Available: https://www.visual-paradigm.com/. [Último acceso: junio 2018].

[16] SourceTree, «SourceTree Blog App,» [En línea]. Available: https://blog.sourcetreeapp.com/files/2017/11/Multiple-Runs.png. [Último acceso: junio 2018].

[17] nueboo, «nueboo, pulsera de actividad,» [En línea]. Available: http://nueboo.com/index.php/lifestyle/smartbands/nueboo-pulsera-de-. [Último acceso: junio 2018].

[18] Polar, «Polar, sensor de frecuencia cardíaca,» [En línea]. Available: https://www.polar.com/es/productos/accesorios/sensor_de_frecuencia_card_aca_h10. [Último acceso: junio 2018].

[19] Bluetooh SIG, «Bluetooth Developer Startet Kit,» [En línea]. Available: https://www.bluetooth.com/develop-with-bluetooth/build/developer-kits/bluetooth-starter-kit. [Último acceso: junio 2018].

[20] E. Gamma, R. Helm, J. Ralph y V. John, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.

9 BIBLIOGRAFÍA

- R. Davidson, Akiba, C. Kubi y K. Townsend, Guetting Started with Bluetooth Low Energy, Sebastopol: O'Reilly Media, Inc, 2014.
- R. Heydon, Bluetooth Low Energy: The Developer's Handbook, Crawfordsville: Prentice Hall, 2012.
- N. Gupta, Inside Bluetooth Low Energy, Artech House, 2013.

ANEXO I

1. EXPORTACIÓN DE LIBRERÍA

El lenguaje base sobre el que se trabaja en Android, es JAVA, al cual se le añaden ciertas modificaciones, para acércalo más a las necesidades que el sistema operativo requiere.

Por lo general, las librerías software desarrolladas para el lenguaje antes mencionado, suelen ser de tipo JAR, en el caso de Android, este tipo de archivos son totalmente válidos para ser usados como librerías, cumpliendo su función sin ninguna complicación. Pero como se ha comentado, el lenguaje propiedad de Google posee recursos y archivos propios, como puede ser el archivo de manifiesto, que aporta además de los métodos java, diseños o elementos de diseño que no pueden incluirse en estos elementos de tipo JAR.

Para poder incluir los recursos propios de Android, se opta por exportar la librería en formato AAR, este tipo de ficheros, pueden contener todo lo necesario para llevar a cabo la compilación de una aplicación o usarse como es el caso en forma de dependencia para módulo de tipo aplicación.

La cuestión que hay que argumentar es por qué es necesario para la nueva librería, usar el formato AAR y no el clásico JAR. Esto se debe a los permisos que son necesarios incluir en el manifiesto de la aplicación, siendo requisitos indispensables para el correcto funcionamiento de la misma, ya que son los encargados de hacer saber al usuario la necesidad de que este permita a la aplicación en cuestión hacer uso de la localización del dispositivo, activar o desactivar el Bluetooth del dispositivo, así como de ponerlo en funcionamiento. Al incluirlos en el manifiesto de la propia librería, se exime al desarrollador de realizar esta tarea, añadiendo así más facilidades para el uso la tecnología BLE en la aplicación que se desee.

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Figura 10.1 Permisos necesarios que son incluidos en el manifiesto de la librería.

2. IMPORTACIÓN DE LIBRERÍA

A continuación, se desarrollará una pequeña explicación, dónde se verá cómo debe hacerse la importación de la librería en Android Studio.

En primer lugar, será necesario tener localizada la biblioteca en formato AAR que contiene todos los métodos de la librería además de los recursos propios de Android que son necesarios. Una vez realizado lo anterior, se deberá proceder como sigue:

1. Es necesario hacer clic en **File > New Module.**

2. Hacer clic en **Import .JAR/.AAA Package** y a continuación clic en **Next.**

3. Por último, se debe insertar la ubicación donde se encuentra el módulo AAR. Después hacer clic en **Finish.**

Cuando se hayan completado los tres pasos anteriores, será necesario incluir la biblioteca en las dependencias del módulo de aplicación, para ello realizar las siguientes operaciones:

1. Hacer clic sobre **File > Project Structure**.

2. Luego, situándose sobre el módulo de **app > Dependencies**.

3. Por último, hacer clic sobre el símbolo **+ > Module dependency** y elegir el módulo correspondiente.

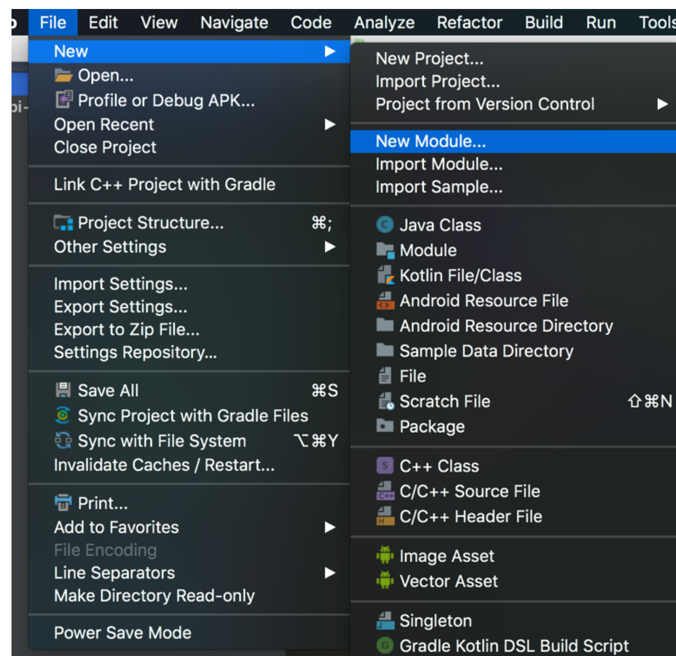


Figura 10.2 Primer paso de importación de la biblioteca AAR.

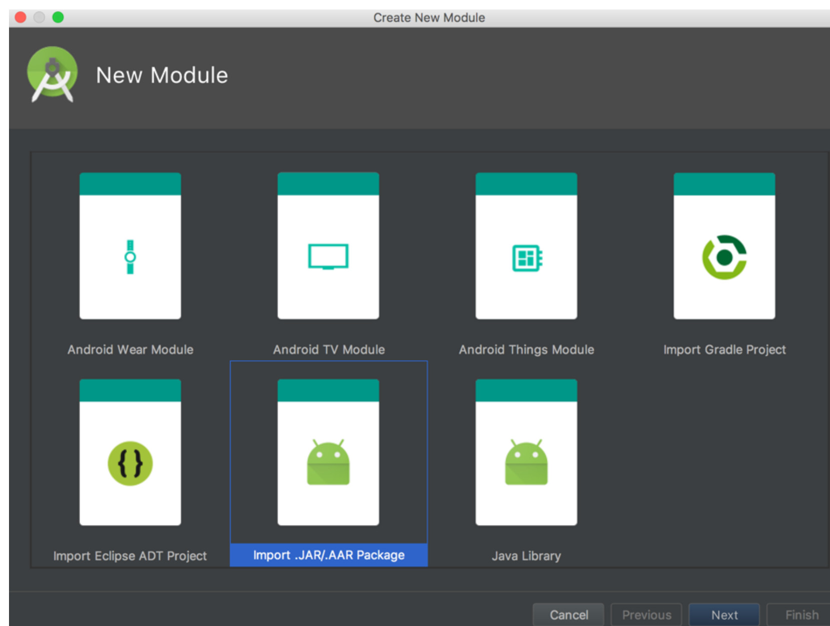


Figura 10.3. Segundo paso de importación de biblioteca AAR.

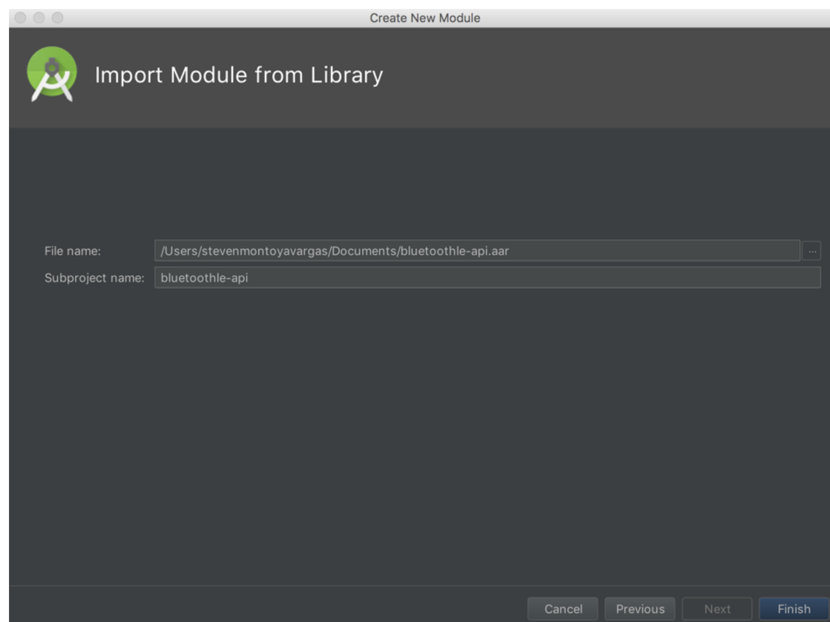


Figura 10.4. Tercer paso de importación de biblioteca AAR.

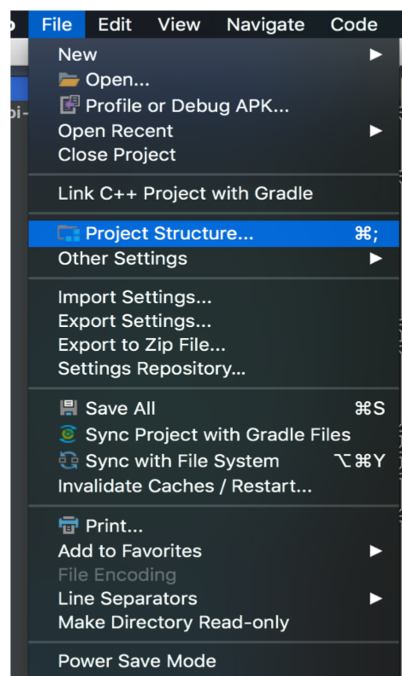


Figura 10.5. Primer paso de configuración de biblioteca AAR.

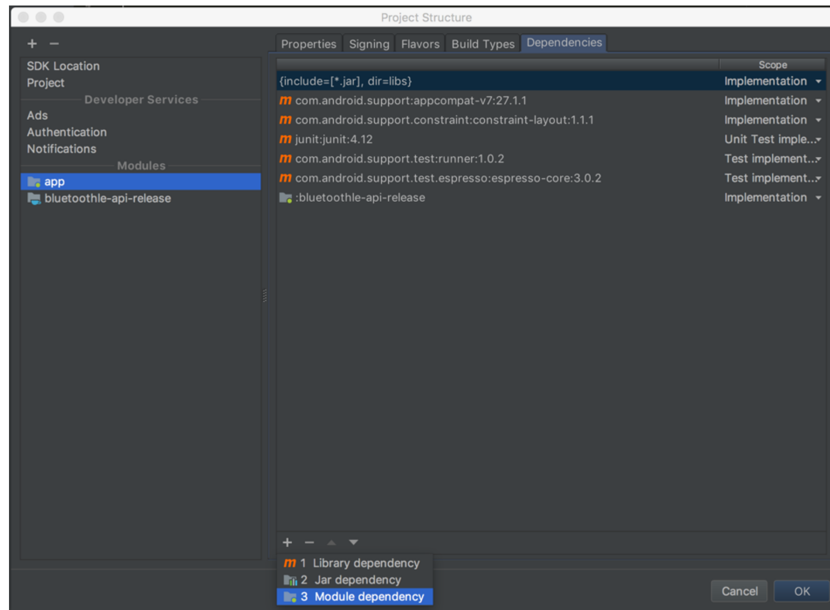


Figura 10.6. Último paso de configuración de biblioteca AAR.

ANEXO II

1. DOCUMENTACIÓN API

En las siguientes páginas de este anexo se incluirá la documentación de la librería desarrollada, siguiendo un formato 'Javadoc', que permitirá una navegación simple entre las distintas clases que se presentan.

Interface BluetoothLeInterface

All Known Implementing Classes:
AbstractBluetoothLowEnergy

`public interface` **`BluetoothLeInterface`**

La interfaz `BluetoothLeInterface` define métodos para ser implementados por la clase AbstractBluetoothLowEnergy

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`void`	**`connect`**`(.BluetoothDevice device)` El método `connect` permite conectarse con un dispositivo y obtener información de este dispositivo.
`void`	**`disconnect`**`(.BluetoothDevice device)` El método `disconnect` permite desconectarse de un dipositivo.
`boolean[]`	**`getCharacteristicProperties`**`(.BluetoothGattCharacteristic characteristic)` La función `getCharacteristicProperties` devuelve un array de tipo boolean, indicando las propiedades de esa característica.
`void`	**`getDataFromService`**`(String service_uuid, String characterisctic_uuid, .BluetoothDevice device)` El método `getDataFromService` obtiene el valor que proporcionan las carcaterísticas del dispositivo.
`ArrayList<BluetoothDevice>`	**`getListDevice`**`()` El método `getListDevice` permite obtener los dispositivos que han sido descubiertos de forma directa.
`boolean`	**`isScanOn`**`()` El método isScanOn devuelve falso si el dispositivo no está escaneando y true en caso contario.
`void`	**`onRequestPermissionsResult`**`(int requestCode, String[] permissions, int[] grantResults)` Este método se llama cuando el usuario o acepta o rechaza los permisos necesarios.
`void`	**`requestLocationPermission`**`()` EL método `requestLocationPermission` permite obtener los permisos necesarios por parte del usuario.
`void`	**`scan`**`(int mode)` El método `scan` inicia el proceso de descubrimiento de dispositivos sin añadir ningún filtro

void	**sendNewData**(String service_uuid, String characteristic_uuid, byte[] value) El método `sendNewData` permite enviar datos al prefiférico, escribiendo nuevos valores en las caraterísticas que lo permitan.
void	**setScanState**(boolean state) El método `setScanState` permite el cambio de estado de la conexión
void	**startBleScan**(int mode) EL método `startBleScan` comienza el escaneo de dispositivos de forma interna, este método realiza una llamada al método scannerBleDevice, que se encuentra en la clase BluetoothLowEnergyScanner.
void	**startScanFilter**(String deviceAddress, String deviceName, String serviceUuid, int mode) El método `startScanFilter` inicia el descubrimiento de dispositivos, permitiendo filtrar por nombre, dirección o UUID del servicio principal

Method Detail

startScanFilter

```
void startScanFilter(String deviceAddress,
                     String deviceName,
                     String serviceUuid,
                     int mode)
```

El método `startScanFilter` inicia el descubrimiento de dispositivos, permitiendo filtrar por nombre, dirección o UUID del servicio principal

Parameters:

`deviceAddress` - Cadena que indica la dirección del dispositivo, generalemnte es de 48 bits.

`deviceName` - Cadena que indica el nombre del dispositivo

`serviceUuid` - Cadena que indica el UUID del servicio principal

`mode` - Variable de tipo Integer que indica el modo en el que se va a escanear

scan

```
void scan(int mode)
```

El método `scan` inicia el proceso de descubrimiento de dispositivos sin añadir ningún filtro

Parameters:

`mode` - Variable de tipo Integer que indica el modo en el que se va a escanear

startBleScan

```
void startBleScan(int mode)
```

EL método `startBleScan` comienza el escaneo de dispositivos de forma interna, este método realiza una llamada al método scannerBleDevice, que se encuentra en la clase BluetoothLowEnergyScanner.

Parameters:

`mode` - Variable de tipo Integer que indica el modo en el que se va a escanear

See Also:

`BluetoothLeScannerInterface`

setScanState

```
void setScanState(boolean state)
```

El método `setScanState` permite el cambio de estado de la conexión

Parameters:

`state` - valor booleano del estado, true si está activado or false si está desactivado.

requestLocationPermission

```
void requestLocationPermission()
```

EL método `requestLocationPermission` permite obtener los permisos necesarios por parte del usuario.

onRequestPermissionsResult

```
void onRequestPermissionsResult(int requestCode,
                                @NonNull
                                String[] permissions,
                                @NonNull
                                int[] grantResults)
```

Este método se llama cuando el usuario o acepta o rechaza los permisos necesarios.

Parameters:

`requestCode` - Entero que indica cual ha sido la respuesta a la petición de permisos.

`permissions` - Array con los permisos.

`grantResults` - Array de enteros con el resultado de la petición.

getListDevice

```
ArrayList<.BluetoothDevice> getListDevice()
```

El método `getListDevice` permite obtener los dispositivos que han sido descubiertos de forma directa. Es necesario inlcuir un postDelayed para llamar a este procedimiento, ya que es necesario asegurarse de que se han descubierto y añadido dispositivos antes de obtenerlos.

Returns:

`BluetoothDevice ArrayList`

getDataFromService

```
void getDataFromService(String service_uuid,
                        String characterisctic_uuid,
                        .BluetoothDevice device)
```

El método `getDataFromService` obtiene el valor que proporcionan las carcaterísticas del dispositivo. Estos valores son devueltos en formato de bytes en la clase parseCharacteristicValue.

Parameters:

`service_uuid` - Cadena que indica el servicio del que se quiere obtener su valor.

`characterisctic_uuid` - Cadena que indica la característica que posee el valor que quiere ser leído.

`device` - Dispositivo del que se están leyendo los datos.

isScanOn

```
boolean isScanOn()
```

El método isScanOn devuelve falso si el dispositivo no está escaneando y true en caso contario.

Returns:

return true or false

See Also:

`BluetoothGattService`, `BluetoothGattCharacteristic`

connect

```
void connect(.BluetoothDevice device)
```

El método `connect` permite conectarse con un dispositivo y obtener información de este dispositivo.

Parameters:

`device` - Dispositivo con el que se desea entablar conexión.

disconnect

```
void disconnect(.BluetoothDevice device)
```

El método `disconnect` permite desconectarse de un dipositivo.

Parameters:

`device` - Dispositivo del que se desea desconectarse.

sendNewData

```
void sendNewData(String service_uuid,
                 String characteristic_uuid,
                 byte[] value)
```

El método `sendNewData` permite enviar datos al prefiférico, escribiendo nuevos valores en las caraterísticas que lo permitan.

Parameters:

`service_uuid` - Cadena que indica el UUID del servicio que posse la característica a la cual queremos cambiar o introducir un nuevo valor.

`characteristic_uuid` - Cadena que indica el UUID de la característica a la que se quiere modificar el valor.

`value -`

getCharacteristicProperties

```
boolean[] getCharacteristicProperties(.BluetoothGattCharacteristic characteristic)
```

La función `getCharacteristicProperties` devuelve un array de tipo boolean, indicando las propiedades de esa característica. Indicando true si la propiedad está presente en dicha característica. Propiedades por posición en el array: 0: PROPERTY_INDICATE, 1: PROPERTY_READ, 2: PROPERTY_BROADCAST, 3: PROPERTY_EXTENDED_PROPS, 4: PROPERTY_NOTIFY, 5: PROPERTY_SIGNED_WRITE, 6: PROPERTY_WRITE, 7: PROPERTY_WRITE_NO_RESPONSE.

Parameters:

`characteristic - Caracteristica de la que se desea conocer sus propiedades.`

Returns:

`boolean [] array`

Interface BluetoothLeScannerInterface

All Known Implementing Classes:

`BluetoothLowEnergyScanner`

`public interface BluetoothLeScannerInterface`

La interfaz `BluetoothLeScannerInterface` define métodos que permiten descubrir los dispositivos cercanos.

See Also:

`BluetoothLeScanner`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`boolean`	`isScanOn()` La función `isScanOn` devuelve un valor boolean que nos indica si el escáner está activado o apagado.
`void`	`scannerBleDevice(ScanResultsDevices scanResultsDevices, long time_scan, String device_address, String device_name, ParcelUuid filter_uuid, int mode)` El método `scannerBleDevice` inicia el escaneo de dispositivos.
`void`	`setScanOn(boolean scan_on)` El método `setSanOn` cambiar el estado del escáner.
`void`	`stopScanDevice()` El método `stopScanDevice` permite parar el descubrimiento de dispositivos.

Method Detail

scannerBleDevice

```
void scannerBleDevice(ScanResultsDevices scanResultsDevices,
                      long time_scan,
                      String device_address,
```

```
                       String device_name,
                       ParcelUuid filter_uuid,
                       int mode)
```

El método `scannerBleDevice` inicia el escaneo de dispositivos. Hay tres modos de escaneo disponibles.

Parameters:

`scanResultsDevices` - scanResultDevices devuelve los dispositivos descubiertos.

`time_scan` - Variable entera que indica el tiempo que se mantendrá activo el escáner.

`device_address` - Cadena usada para filtrar los dispositivos por dirección.

`device_name` - Cadena usada para filtrar los dispositivos por nombre.

`filter_uuid` - Variable de tipo ParcelUuid usada para filtrar los dispositivos por el UUID del servicio principal.

`mode` - Entero que indica el modo seleccionado para escanear. 0 == SCAN_MODE_BALANCED. 1 == SCAN_MODE_LOW_LATENCY. 2 == SCAN_MODE_LOW_POWER. 3 == SCAN_MODE_OPPORTUNISTIC.

See Also:

ScanResultsDevices

stopScanDevice

```
void stopScanDevice()
```

El método `stopScanDevice` permite parar el descubrimiento de dispositivos.

isScanOn

```
boolean isScanOn()
```

La función `isScanOn` devuelve un valor boolean que nos indica si el escáner está activado o apagado.

Returns:

true si el escáner está activado, false si está parado.

setScanOn

```
void setScanOn(boolean scan_on)
```

El método `setSanOn` cambiar el estado del escáner.

Parameters:

`scan_on` - Variable de tipo boolean. true si está encendido y false si está apagado.

Interface ScanResultsDevices

All Known Implementing Classes:

`AbstractBluetoothLowEnergy`

```
public interface ScanResultsDevices
```

La interfaz `ScanResultsDevices` procesa los dispositivos descubiertos y maneja el estado del escáner.

See Also:

`BluetoothDevice`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`void`	**`scanningStarted`**`()` El método `scanningStarted` notifica que el escáner se encuentra iniciado.
`void`	**`scanningStopped`**`()` El método `scanningStopped` notifica que el escáner se encuentra parado.
`void`	**`selectedBleDevice`**`(BluetoothDevice device, byte[] scan_record, int rssi)` Este método `selectedBleDevice` es llamado cuando un dispositivo es descubierto, es usado por la clase BluetoothLowEnergyScanner

Method Detail

selectedBleDevice

```
void selectedBleDevice(BluetoothDevice device,
                       byte[] scan_record,
                       int rssi)
```

Este método `selectedBleDevice` es llamado cuando un dispositivo es descubierto, es usado por la clase BluetoothLowEnergyScanner

Parameters:

`device` - Dispositivo descubierto por el escáner

`scan_record` - Array de bytes con información.

`rssi` - Variable de tipo entero que indica la intensidad de la señal

scanningStarted

```
void scanningStarted()
```

El método `scanningStarted` notifica que el escáner se encuentra iniciado.

scanningStopped

```
void scanningStopped()
```

El método `scanningStopped` notifica que el escáner se encuentra parado.

Class AbstractBluetoothLowEnergy

All Implemented Interfaces:

`BluetoothLeInterface`, `ScanResultsDevices`

```
public abstract class AbstractBluetoothLowEnergy
extends Object
implements ScanResultsDevices, BluetoothLeInterface
```

La clase `AbstractBluetoothLowEnergy` incluye métodos que permiten el escaneo de dispositivos, establecer conexión, lectura de servicios y características, así como el valor de estas últimas.

Constructor Summary

Constructors

Constructor and Description
AbstractBluetoothLowEnergy(Context context, Activity activity) Constructor de la clase

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
void	**connect**(BluetoothDevice device) El método `connect` permite conectarse con un dispositivo y obtener información de este dispositivo.
void	**disconnect**(BluetoothDevice device) El método `disconnect` permite desconecctase de un dipositivo.
boolean[]	**getCharacteristicProperties**(BluetoothGattCharacteristic characteristic) La función `getCharacteristicProperties` devuelve un array de tipo boolean, indicando las propiedades de esa característica.
void	**getDataFromService**(String service_uuid, String characterisctic_uuid, BluetoothDevice device) El método `getDataFromService` obtiene el valor que proporciona las carcaterísticas del dispositivo.
ArrayList<BluetoothDevice>	**getListDevice**() El método `getListDevice` permite obtener los dispositivos que han sido descubiertos de forma directa.
void	**informationDeviceRead**(List<BluetoothGattService> listServices, Map<BluetoothGattService,List<BluetoothGattCharacteristic>> mapServiceCharacteristic) El método `informationDeviceRead` recibe una lista de servicios y un mapa que relaciona cada servicio con sus características.
boolean	**isScanOn**() El método `isScanOn` devuelve falso si el dispositivo no está escaneando y true en caso contario.

void	**onRequestPermissionsResult**(int requestCode, String[] permissions, int[] grantResults) El método `onRequestPermissionsResult` se llama cuando el usuario o acepta o rechaza los permisos necesarios.
void	**parseCharacteristicValue**(byte[] arrayBytesValue) El método `parseCharacteristicValue` recibe un array de valores que indican el valor de la característica leída.
void	**readPhy**(BluetoothGatt gatt, int txPhy, int rxPhy, int status) El metodo `readPhy` recibe el valor del Phy
void	**requestLocationPermission**() EL método `requestLocationPermission` permite obtener los permisos necesarios por parte del usuario.
void	**scan**(int mode) El método `scan` inicia el proceso de descubrimiento de dispositivos sin añadir ningún filtro
void	**scanningStarted**() EL método `scanningStarted` introduce un valor boolean con el estado de la conexión, true si está activa y false en caso contrario.
void	**scanningStopped**() El método `scanningStopped` para el proceso de escaneo.
void	**selectedBleDevice**(BluetoothDevice device, byte[] scan_record, int rssi) Este método `selectedBleDevice` es llamado cuando un dispositivo es descubierto, se llama desde la clase BluetoothLowEnergy.
void	**selectedBleDevices**(BluetoothDevice device, int rssi, byte[] scan_record) El método `selectedBleDevices` permite contemplar los dispositivos que estan siendo descubiertos de forma gradual.
void	**sendNewData**(String service_uuid, String characteristic_uuid, byte[] value) El método `sendNewData` permite enviar datos al prefiférico, escribiendo nuevos valores en las caraterísticas que lo permitan.
void	**setScanState**(boolean state) El método `setScanState` permite el cambio de estado de la conexión
void	**startBleScan**(int mode) EL método `startBleScan` comienza el escaneo de dispositivos de forma interna.
void	**startScanFilter**(String deviceAddress, String deviceName, String serviceUuid, int mode) El método `startScanFilter` inicia el descubrimiento de dispositivos, permitiendo filtrar por nombre, dirección o UUID del servicio principal

Methods inherited from class Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

AbstractBluetoothLowEnergy

```
public AbstractBluetoothLowEnergy(Context context,
                                  Activity activity)
```

Constructor de la clase

Parameters:

`context - Activity context`

activity - Actividad principal de la aplicación

Method Detail

startScanFilter

```
public void startScanFilter(String deviceAddress,
                            String deviceName,
                            String serviceUuid,
                            int mode)
```

El método `startScanFilter` inicia el descubrimiento de dispositivos, permitiendo filtrar por nombre, dirección o UUID del servicio principal

Specified by:

startScanFilter in interface BluetoothLeInterface

Parameters:

deviceAddress - Cadena que indica la dirección del dispositivo, generalemnte es de 48 bits.

deviceName - Cadena que indica el nombre del dispositivo

serviceUuid - Cadena que indica el UUID del servicio principal

mode - Variable de tipo Integer que indica el modo en el que se va a escanear

scan

```
public void scan(int mode)
```

El método `scan` inicia el proceso de descubrimiento de dispositivos sin añadir ningún filtro

Specified by:

scan in interface BluetoothLeInterface

Parameters:

mode - Variable de tipo Integer que indica el modo en el que se va a escanear

startBleScan

```
public void startBleScan(int mode)
```

EL método `startBleScan` comienza el escaneo de dispositivos de forma interna. Este método realiza una llamada al método scannerBleDevice.

Specified by:

startBleScan in interface BluetoothLeInterface

Parameters:

mode - Variable de tipo Integer que indica el modo en el que se va a escanear

See Also:

BluetoothLeScannerInterface

getListDevice

```
public ArrayList<BluetoothDevice> getListDevice()
```

El método `getListDevice` permite obtener los dispositivos que han sido descubiertos de forma directa. Es necesario inlcuir un postDelayed para llamar a este procedimiento, ya que es necesario asegurarse de que se han descubierto y añadido dispositivos antes de obtenerlos.

Specified by:

getListDevice in interface BluetoothLeInterface

Returns:

`BluetoothDevice ArrayList`

getDataFromService

```
public void getDataFromService(String service_uuid,
                               String characterisctic_uuid,
                               BluetoothDevice device)
```

El método `getDataFromService` obtiene el valor que proporciona las carcaterísticas del dispositivo. Estos valores son devueltos en formato de bytes en el método `parseCharacteristicValue` .

Specified by:

`getDataFromService` in interface `BluetoothLeInterface`

Parameters:

`service_uuid` - Cadena que indica el servicio del que se quiere obtener su valor.

`characterisctic_uuid` - Cadena que indica la característica que posee el valor que quiere ser leído.

`device` - Dispositivo del que se están leyendo los datos.

selectedBleDevice

```
public void selectedBleDevice(BluetoothDevice device,
                              byte[] scan_record,
                              int rssi)
```

Este método `selectedBleDevice` es llamado cuando un dispositivo es descubierto, se llama desde la clase BluetoothLowEnergy. A su vez en este se devuevle la información mediante el método selectedBleDevice.

Specified by:

`selectedBleDevice` in interface `ScanResultsDevices`

Parameters:

`device` - Dispositivo que ha sido descubierto

`scan_record` - Array con información de la conexión, viene en formato de array de bytes.

`rssi` - Entero que indica la intensidad de la señal.

scanningStarted

```
public void scanningStarted()
```

EL método `scanningStarted` introduce un valor boolean con el estado de la conexión, true si está activa y false en caso contrario.

Specified by:

`scanningStarted` in interface `ScanResultsDevices`

scanningStopped

```
public void scanningStopped()
```

El método `scanningStopped` para el proceso de escaneo.

Specified by:

`scanningStopped` in interface `ScanResultsDevices`

parseCharacteristicValue

```
public void parseCharacteristicValue(byte[] arrayBytesValue)
```

El método `parseCharacteristicValue` recibe un array de valores que indican el valor de la característica leída.

selectedBleDevices

```
public void selectedBleDevices(BluetoothDevice device,
                               int rssi,
                               byte[] scan_record)
```

El método `selectedBleDevices` permite contemplar los dispositivos que estan siendo descubiertos de forma gradual.

informationDeviceRead

```
public void informationDeviceRead(List<BluetoothGattService> listServices,

Map<BluetoothGattService,List<BluetoothGattCharacteristic>> mapServiceCharacteristic)
```

El método `informationDeviceRead` recibe una lista de servicios y un mapa que relaciona cada servicio con sus características. Este método es llamado cuando se establece la conexión con un dispositivo.

readPhy

```
public void readPhy(BluetoothGatt gatt,
                    int txPhy,
                    int rxPhy,
                    int status)
```

El metodo `readPhy` recibe el valor del Phy

connect

```
public void connect(BluetoothDevice device)
```

El método `connect` permite conectarse con un dispositivo y obtener información de este dispositivo.

Specified by:

`connect` in interface `BluetoothLeInterface`

Parameters:

`device` - Dispositivo con el que se desea entablar conexión.

sendNewData

```
public void sendNewData(String service_uuid,
                        String characteristic_uuid,
                        byte[] value)
```

El método `sendNewData` permite enviar datos al prefiférico, escribiendo nuevos valores en las caraterísticas que lo permitan.

Specified by:

`sendNewData` in interface `BluetoothLeInterface`

Parameters:

`service_uuid` - Cadena que indica el UUID del servicio que posse la característica a la cual queremos cambiar o introducir un nuevo valor.

`characteristic_uuid` - Cadena que indica el UUID de la característica a la que se quiere modificar el valor.

`value` -

getCharacteristicProperties

```
public boolean[] getCharacteristicProperties(BluetoothGattCharacteristic characteristic)
```

La función `getCharacteristicProperties` devuelve un array de tipo boolean, indicando las propiedades de esa característica. Indicando true si la propiedad está presente en dicha característica. Propiedades por posición: 0: PROPERTY_INDICATE, 1: PROPERTY_READ, 2: PROPERTY_BROADCAST, 3: PROPERTY_EXTENDED_PROPS, 4: PROPERTY_NOTIFY, 5: PROPERTY_SIGNED_WRITE, 6: PROPERTY_WRITE, 7: PROPERTY_WRITE_NO_RESPONSE.

Specified by:

`getCharacteristicProperties` in interface `BluetoothLeInterface`

Parameters:

`characteristic` - Caracteristica de la que se desea conocer sus propiedades.

Returns:

boolean [] array

disconnect

```
public void disconnect(BluetoothDevice device)
```

El método `disconnect` permite desconecctase de un dipositivo.

Specified by:

`disconnect` in interface `BluetoothLeInterface`

Parameters:

`device` - Dispositivo del que se desea desconectar.

setScanState

```
public void setScanState(boolean state)
```

El método `setScanState` permite el cambio de estado de la conexión

Specified by:

`setScanState` in interface `BluetoothLeInterface`

Parameters:

`state` - valor booleano del estado, true si está activado or false si está desactivado.

isScanOn

```
public boolean isScanOn()
```

El método `isScanOn` devuelve falso si el dispositivo no está escaneando y true en caso contario.

Specified by:

`isScanOn` in interface `BluetoothLeInterface`

Returns:

return true or false

See Also:

`BluetoothGattService`, `BluetoothGattCharacteristic`

requestLocationPermission

```
public void requestLocationPermission()
```

EL método `requestLocationPermission` permite obtener los permisos necesarios por parte del usuario.

Specified by:

requestLocationPermission in interface BluetoothLeInterface

onRequestPermissionsResult

```
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull
                                       String[] permissions,
                                       @NonNull
                                       int[] grantResults)
```

El método `onRequestPermissionsResult` se llama cuando el usuario o acepta o rechaza los permisos necesarios.

Specified by:

onRequestPermissionsResult in interface BluetoothLeInterface

Parameters:

requestCode - Entero que indica cual ha sido la respuesta a la petición de permisos.

permissions - array con los permisos.

grantResults - array de enteros con el resultado de la petición.

Class BluetoothLowEnergyScanner

All Implemented Interfaces:

`BluetoothLeScannerInterface`

```
public class BluetoothLowEnergyScanner
extends Object
implements BluetoothLeScannerInterface
```

La clase `BluetoothLowEnergyScanner` se encarga de realizar el descubrimiento de dispositivos. Utiliza las clase ScanCallBack que recibe los dispositivos que van siendo descubiertos.

See Also:

`BluetoothLeScanner, ScanCallback`

Constructor Summary

Constructors

Constructor and Description
`BluetoothLowEnergyScanner(Context context)` Constructor de clase.

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method and Description
`boolean`	`isScanOn()` La función `isScanOn` devuelve un valor boolean que nos indica si el escáner está activado o apagado.
`void`	`scannerBleDevice(ScanResultsDevices scanResultsDevices, long time_scan, String device_address, String device_name, android.os.ParcelUuid filter_uuid, int mode)` El método `scannerBleDevice` inicia el escaneo de dispositivos.

void	**setScanOn**(boolean scan_on) El método setScanOn cambiar el estado del escáner.
void	**stopScanDevice**() El método stopScanDevice permite parar el descubrimiento de dispositivos.

Methods inherited from class Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

BluetoothLowEnergyScanner

```
public BluetoothLowEnergyScanner(Context context)
```

Constructor de clase.

Parameters:

context - Activity context.

Method Detail

scannerBleDevice

```
public void scannerBleDevice(ScanResultsDevices scanResultsDevices,
                             long time_scan,
                             String device_address,
                             String device_name,
                             android.os.ParcelUuid filter_uuid,
                             int mode)
```

El método scannerBleDevice inicia el escaneo de dispositivos. Hay tres modos de escaneo disponibles.

Specified by:

scannerBleDevice in interface BluetoothLeScannerInterface

Parameters:

scanResultsDevices - scanResultDevices devuelve los dispositivos descubiertos.

time_scan - Variable entera que indica el tiempo que se mantendrá activo el escáner.

device_address - Cadena usada para filtrar los dispositivos por dirección.

device_name - Cadena usada para filtrar los dispositivos por nombre.

filter_uuid - Variable de tipo ParcelUuid usada para filtrar los dispositivos por el uuid del servicio principal.

mode - Entero que indica el modo seleccionado para escanear. 0 == SCAN_MODE_BALANCED. 1 == SCAN_MODE_LOW_LATENCY. 2 == SCAN_MODE_LOW_POWER. 3 == SCAN_MODE_OPPORTUNISTIC.

See Also:

ScanResultsDevices

stopScanDevice

```
public void stopScanDevice()
```

El método `stopScanDevice` permite parar el descubrimiento de dispositivos.

Specified by:

`stopScanDevice` in interface `BluetoothLeScannerInterface`

isScanOn

```
public boolean isScanOn()
```

La función `isScanOn` devuelve un valor boolean que nos indica si el escáner está activado o apagado.

Specified by:

`isScanOn` in interface `BluetoothLeScannerInterface`

Returns:

Devuelve un boolean que indica el estado del escáner.

setScanOn

```
public void setScanOn(boolean scan_on)
```

El método `setScanOn` cambiar el estado del escáner.

Specified by:

`setScanOn` in interface `BluetoothLeScannerInterface`

Parameters:

scan_on - Variable de tipo boolean. true si está encendido y false si está apagado.

Class List_Adapter

```
public class List_Adapter
extends Object
```

La clase `ListAdapter` es utilizada para gestionar la lista de dispositivos que son descubiertos en el proceso de escaneo

Constructor Summary

Constructors

Constructor and Description
`List_Adapter()` Constructor de clase

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`void`	`addDevice(BluetoothDevice device)` El método `addDevice` permite añadir un nuevo dispositivo a la lista.
`void`	`clear()` El método `clear` permite limpiar por completo la lista de dispositivos.
`boolean`	`contains(BluetoothDevice device)` El método `contains` permite conocer si un dispositivo está o no en la lista
`void`	`deleteDevice(BluetoothDevice device)` El método `deleteDevice` permite eliminar dispositivos de la lista.
`int`	`getCount()` El método `getCount` permite conocer el tamaño de la lista.
`BluetoothDevice`	`getDevice(int posistion)` El método `getDevice` permite obtener un dispositivo de la lista.

BluetoothDevice	**getDeviceFromAddress**(String address) El método `getDeviceFromAddress` permite obtener un dispositivo filtrando por su dirección.
ArrayList<BluetoothDevice>	**getListDevices**() El método `getListDevices` devuelve todos los dispositivos que se encuentran en la lista.

Methods inherited from class Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

List_Adapter

`public List_Adapter()`

Constructor de clase

Method Detail

addDevice

`public void addDevice(BluetoothDevice device)`

El método `addDevice` permite añadir un nuevo dispositivo a la lista.

Parameters:

`device` - Dispositivo que se quiere añadir a la lista.

deleteDevice

`public void deleteDevice(BluetoothDevice device)`

El método `deleteDevice` permite eliminar dispositivos de la lista.

Parameters:

`device` - Dispositivo que se quiere eliminar de la lista.

contains

`public boolean contains(BluetoothDevice device)`

El método `contains` permite conocer si un dispositivo está o no en la lista

Parameters:

`device - Dispositivo que se quiere comprobar si se encuentra o no en la lista.`

Returns:

`true si está en la lista, false si no lo está.`

getDevice

```
public BluetoothDevice getDevice(int posistion)
```

El método `getDevice` permite obtener un dispositivo de la lista.

Returns:

`BluetoothDevice.`

clear

```
public void clear()
```

El método `clear` permite limpiar por completo la lista de dispositivos.

getCount

```
public int getCount()
```

El método `getCount` permite conocer el tamaño de la lista.

Returns:

`Devuelve un valor entero que indica el tamaño.`

getDeviceFromAddress

```
public BluetoothDevice getDeviceFromAddress(String address)
```

El método `getDeviceFromAddress` permite obtener un dispositivo filtrando por su dirección.

Parameters:

`address - Cadena que indica la dirección con la que se quiere filtrar.`

Returns:

`BluetoothDevice`

getListDevices

```
public ArrayList<BluetoothDevice> getListDevices()
```

El método `getListDevices` devuelve todos los dispositivos que se encuentran en la lista.

Returns:

`Devuelve un un array de tipo BluetoothDevice.`

Class Constants_UUID

```
public class Constants_UUID
extends Object
```

La clase `Constants_UUID` define cadenas constantes y toman como valor los UUID de servicios y características oficialmente adaptados por SIG.

Field Summary

Fields

Modifier and Type	Field and Description
String	ALERT_NOTIFICATION_SERVICE_UUID
String	APPAREANCE_CHARACTERISTIC_UUID
String	BATERY_SERVICE_UUID
String	BATTERY_LEVEL_CHARACTERISTIC_UUID
int	BLOOD_PRESSURE_ARM
String	BLOOD_PRESSURE_MEASUREMENT_CHARACTERISTIC_UUID
String	BLOOD_PRESSURE_SERVICE_UUID
int	BLOOD_PRESSURE_WRIST
String	CLIENT_CHARACTERISTIC_CONFIG_UUID
String	CLIENT_CHARACTERISTIC_CONFIG_UUID_2
String	DEVICE_NAME_CHARACTERISTIC_UUID
String	GENERIC_ACCESS_SERVICE_UUID
int	GENERIC_BLOOD_PRESURE
int	GENERIC_HEART_RATE_SENSOR
int	GENERIC_INSULIN_PUMP
int	GENERIC_THERMOMETER

String	GLUCOSE_MEASUREMENT_CHARACTERISTIC_UUID
String	GLUCOSE_SERVICE_UUID
String	GLUCOSSE_SERVICE
String	HEALTH_THERMOMETER_SERVICE_UUID
String	HEART_RATE_MEASUREMENT_CHARACTERISTIC_UUID
int	HEART_RATE_SENSOR_HEART_BELT
String	HEART_RATE_SERVICE_UUID
int	SCAN_MODE_BALANCED_
int	SCAN_MODE_LOW_LATENCY
int	SCAN_MODE_LOW_POWER
int	SCAN_MODE_OPORTUNIST
String	TEMPERATURE_MEASUREMENT_CHARACTERISTIC_UUID

Constructor Summary

Constructors

Constructor and Description
Constants_UUID()

Method Summary

Methods inherited from class Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

HEART_RATE_SERVICE_UUID

```
public final String HEART_RATE_SERVICE_UUID
```

See Also:

Constant Field Values

BLOOD_PRESSURE_SERVICE_UUID

```
public final String BLOOD_PRESSURE_SERVICE_UUID
```

See Also:

Constant Field Values

GLUCOSE_SERVICE_UUID

```
public final String GLUCOSE_SERVICE_UUID
```

See Also:

Constant Field Values

HEALTH_THERMOMETER_SERVICE_UUID

```
public final String HEALTH_THERMOMETER_SERVICE_UUID
```

See Also:

Constant Field Values

CLIENT_CHARACTERISTIC_CONFIG_UUID

```
public final String CLIENT_CHARACTERISTIC_CONFIG_UUID
```

See Also:

Constant Field Values

CLIENT_CHARACTERISTIC_CONFIG_UUID_2

```
public final String CLIENT_CHARACTERISTIC_CONFIG_UUID_2
```

See Also:

Constant Field Values

GENERIC_ACCESS_SERVICE_UUID

```
public final String GENERIC_ACCESS_SERVICE_UUID
```

See Also:

Constant Field Values

ALERT_NOTIFICATION_SERVICE_UUID

```
public final String ALERT_NOTIFICATION_SERVICE_UUID
```

See Also:

Constant Field Values

BATERY_SERVICE_UUID

```
public final String BATERY_SERVICE_UUID
```

See Also:

Constant Field Values

GLUCOSSE_SERVICE

```
public final String GLUCOSSE_SERVICE
```

See Also:

Constant Field Values

HEART_RATE_MEASUREMENT_CHARACTERISTIC_UUID

```
public final String HEART_RATE_MEASUREMENT_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

BLOOD_PRESSURE_MEASUREMENT_CHARACTERISTIC_UUID

```
public final String BLOOD_PRESSURE_MEASUREMENT_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

GLUCOSE_MEASUREMENT_CHARACTERISTIC_UUID

```
public final String GLUCOSE_MEASUREMENT_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

TEMPERATURE_MEASUREMENT_CHARACTERISTIC_UUID

```
public final String TEMPERATURE_MEASUREMENT_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

APPAREANCE_CHARACTERISTIC_UUID

```
public final String APPAREANCE_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

BATTERY_LEVEL_CHARACTERISTIC_UUID

```
public final String BATTERY_LEVEL_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

DEVICE_NAME_CHARACTERISTIC_UUID

```
public final String DEVICE_NAME_CHARACTERISTIC_UUID
```

See Also:

Constant Field Values

HEART_RATE_SENSOR_HEART_BELT

```
public final int HEART_RATE_SENSOR_HEART_BELT
```

See Also:

Constant Field Values

GENERIC_HEART_RATE_SENSOR

```
public final int GENERIC_HEART_RATE_SENSOR
```

See Also:

Constant Field Values

BLOOD_PRESSURE_ARM

```
public final int BLOOD_PRESSURE_ARM
```

See Also:

Constant Field Values

BLOOD_PRESSURE_WRIST

```
public final int BLOOD_PRESSURE_WRIST
```

See Also:

Constant Field Values

GENERIC_BLOOD_PRESURE

```
public final int GENERIC_BLOOD_PRESURE
```

See Also:

Constant Field Values

GENERIC_INSULIN_PUMP

```
public final int GENERIC_INSULIN_PUMP
```

See Also:

Constant Field Values

GENERIC_THERMOMETER

```
public final int GENERIC_THERMOMETER
```

See Also:

Constant Field Values

SCAN_MODE_BALANCED_

```
public final int SCAN_MODE_BALANCED_
```

See Also:

Constant Field Values

SCAN_MODE_LOW_LATENCY

```
public final int SCAN_MODE_LOW_LATENCY
```

See Also:

Constant Field Values

SCAN_MODE_LOW_POWER

```
public final int SCAN_MODE_LOW_POWER
```

See Also:

Constant Field Values

SCAN_MODE_OPORTUNIST

```
public final int SCAN_MODE_OPORTUNIST
```

See Also:

Constant Field Values

Constructor Detail

Constants_UUID

```
public Constants_UUID()
```