Desarrollo de un despliegue completo de sensorización de edificios mediante BLE Mesh

José Ángel Garrido Montoya

Máster en Internet de las Cosas

Trabajo Fin de Máster en Internet de las Cosas

27 de septiembre de 2021

Director: Francisco Daniel Igual Peña
Calificación: *9*

Autorización de difusión

José Ángel Garrido Montoya

27 de septiembre de 2021

El/la abajo firmante, matriculado/a en el Máster en Internet de las Cosas la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: *"Desarrollo de un despliegue completo de sensorización de edificios mediante BLE Mesh"*, realizado durante el curso académico 2020-2021 bajo la dirección de Francisco Daniel Igual Peña en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

Con el paso de los años, el campo de IoT (Internet de las Cosas) ha ganado aceptación de forma progresiva. De hecho, esta revolución ha llevado a cabo el desarrollo de placas de desarrollo capaces de conectarse y comunicarse mediante multitud de protocolos; tanto es así, que tenemos por ejemplo una placa como la Espressif Dev_kitc_v4 a un precio de unos 5 dolares con capacidades WiFi y Bluetooth. Debido a esto, se abren multitud de usos y escenarios diferentes para los que antes, con la tecnología disponible, era difícil desarrollar soluciones factibles. Tanto es así que, debido a su bajo precio, IoT se ha implantado en ámbitos como por ejemplo industria, en una planta de montaje, en ganadería para controlar las reses de manera remota, en edificios para el control de la temperatura en distintas salas, control de iluminación o lo que llamamos comúnmente como domótica.

Cuando nos enfrentamos a un proyecto de IoT, una de las partes más determinantes es el tipo de arquitectura de red que debemos usar. Esto viene prácticamente impuesto por el tipo de problema que el proyecto quiera resolver. Una arquitectura que se ha visto impulsada por este campo ha sido la de tipo *mesh* mediante el uso de Bluetooth Low Energy. Este tipo de arquitectura y tecnología se utiliza en ámbitos como la domótica, aplicaciones industriales, edificios inteligentes, etc. El punto principal por el que BLE Mesh se ha revelado como una tecnología interesante en IoT es que provee, desde su especificación, de *modelos* que encapsulan mensajes y un comportamiento determinado. Concretamente, este Trabajo de Fin de Máster ofrece una descripción detallada del proceso de desarrollo de un *firmware*, desarrollado bajo el *framework* de Espressif, para un despliegue de una red de sensores mediante el uso de BLE Mesh y el modelo *sensor*. Juntos, los elementos desarrollados proporcionan toda la funcionalidad necesaria para realizar un despliegue fiable que permita la sensorización de distintos parámetros ambientales en un entorno cerrado.

Palabras clave

IoT, Internet of Things, Bluetooth, BLE Mesh, ESP32, Espressif ESP IDF, Raspberry Pi, Modelo Sensor, WiFi.

Abstract

Over the years, the IoT (Internet of Things) field has progressively gained acceptance. In fact, this revolution has led to the development of development boards capable of connecting and communicating through a multitude of protocols; so much so, that we have for example a board like the Espressif Dev_kitc_v4 at a price of about 5 dollars with WiFi and Bluetooth capabilities. Because of this, a multitude of uses and different scenarios are opened for which before, with the available technology, it was difficult to develop feasible solutions. So much so that, due to its low price, IoT has been implemented in areas such as industry, in an assembly plant, in livestock to control cattle remotely, in buildings to control the temperature in different rooms, lighting control or what we commonly call home automation.

When we face an IoT project, one of the most determining parts is the type of network architecture to use. This is practically imposed by the type of problem the project wants to solve. One architecture that has been driven by this field has been the mesh type through the use of Bluetooth Low Energy. This type of architecture and technology is used in fields such as home automation, industrial applications, intelligent buildings, etc. The main point why BLE Mesh has been revealed as an interesting technology in IoT is that it provides, from its specification, em models that encapsulate messages and a certain behavior. Specifically, this Master Thesis provides a detailed description of the development process of a firmware, developed under the Espressif framework, for a sensor network deployment using BLE Mesh and the sensor model. Together, the developed elements provide all the necessary functionality to perform a reliable deployment that allows the sensing of different environmental parameters in a closed environment.

Keywords

IoT, Internet of Things, Bluetooth, BLE Mesh, ESP32, Espressif ESP IDF, Raspberry Pi, Sensor Model, WiFi.

Índice general

Agradecimientos

Agradecimientos primeramente a mi profesor Francisco Daniel Igual Peña por la paciencia y los conocimientos que me ha ido enseñando, tanto en la carrera como en el máster, para la realización de este TFM. Agradecimientos también a mis padres por todo su apoyo ya que sin ellos no habría sido posible estudiar Ingeniería Informática, y mucho menos el Máster en Internet de las Cosas. Agradecimientos también a mi hermano y hermana que, aunque son pequeños, me han dado mucha alegría en todo momento y, este trabajo, es una manera de demostrarles que se puede conseguir lo que se propone. Por ultimo, agradecimientos a mi novia Alicia, futura óptica y optometrista, ya que me ha ayudado mucho mentalmente a seguir adelante con los estudios.

A todos ellos, gracias porque siempre han creido en mí y en mis capacidades.

Dedicatoria

Este trabajo final esta dedicado a mi. Al esfuerzo realizado para sacarme la carrera y el máster, a mi para demostrarme que puedo hacerlo y que lo he conseguido. A ese chaval que con 6 años sabia que queria estar frente a un ordenador. A todas esas personas que no se ven capaces. A ti, un lector que por casualidad ha acabado leyendo esta memoria pero que nos une el gusto por este campo. Por último, se lo dedico a las personas mencionadas en los agradecimiento, tanto a mi familia y profesor que se ha esforzado en hacer este trabajo conmigo.

Capítulo 1

Introducción

Los datos representados en la Figura 8.1 revelan la importancia que IoT (*Internet of Things*, Internet de las Cosas) ha tomado en los últimos años. El motivo, en parte, radica en la adopción de las redes móviles como 2G/3G/4G y ahora 5G, que permiten transmitir cada vez a mayor velocidad desde cualquier rincón del planeta. También han crecido el número de dispositivos que están conectados, como por ejemplo los *smartwatch*, pulseras inteligentes, electrodomésticos en general, asistentes, televisiones, vehículos, edificios etc. En general, disponemos de una amplia variedad de dispositivos diferentes que pueden conectarse entre sí e intercambiar información, fenómeno que conlleva un nuevo reto para la adopción de nuevas tecnologías de comunicación y topologías de red diferentes que puedan adecuarse a cada escenario.

Las definiciones para IoT son muy diversas y dispares, aunque todas ellas tienen un punto en común: la *interconexión* de diferentes tipos de dispositivos para intercambiar información. Un tipo de arquitectura de red muy usada en IoT son las basadas en topologías de tipo malla (*mesh*). Este tipo de arquitectura permite conectar multitud de nodos entre sí, con conexiones de tipo "muchos a muchos"; se caracterizan, además, porque permiten a los nodos configurarse de manera dinámica, por lo que si un nodo cae o se añade a la red, no interfiere en ninguna comunicación (o si lo hace, lo hará de forma transitoria). De hecho, esta es la característica más importante, ya que en términos de confiabilidad es más ventajosa frente a otro tipo de topologías. Por ejemplo, en una red doméstica toda la información suele

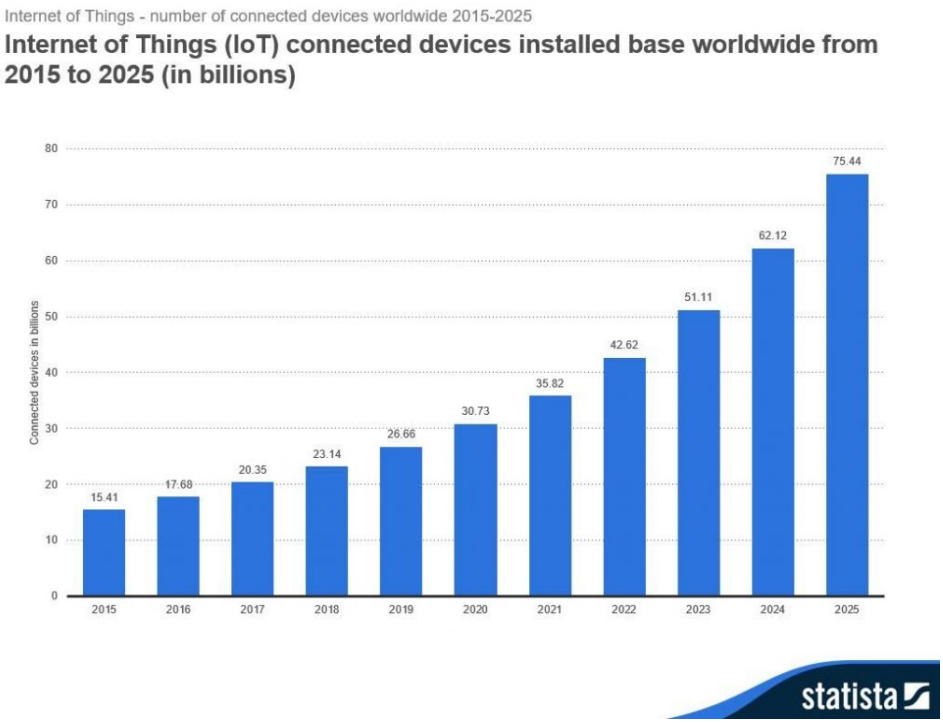

Figura 1.1: *Numero de dispositivos conectados por año [7].*

atravesar un punto central o router y, si este se cae, se corta toda comunicación; en cambio, en una red *mesh*, la información puede tomar caminos diversos hasta llegar a su destino ya que no es una red centralizada.

La arquitectura de red *mesh* se puede montar sobre diversas tecnologías de red como puede ser WiFi, 6LowPan, LoRa y BLE. WiFi Mesh básicamente nos permite tener una red *mesh* donde la capa de red viene dada por WiFi, 6LowPan es un estándar que nos permite usar IPv6 sobre redes que funcionen bajo el estándar IEEE 802.15.4, LoRa es una tecnología inalámbrica que utiliza un tipo de modulación patentado por Semtech la cual permite comunicación a grandes distancias.

En términos generales, un proyecto de IoT se caracteriza por la obtención de datos, ingesta y tratamiento de los datos, monitorización y configuración. Teniendo en cuenta esto, lo que se pretende con este trabajo es ofrecer una aplicación para un despliegue de dispositivos que funcionen con tecnología BLE Mesh, junto con un panel de control y una

herramienta que nos permita configurarlos.

1.1. Motivación

BLE Mesh es una tecnología reciente, de gran aplicación en diferentes entornos donde no se puede encontrar mucha información. La primera razón por la que realizar este trabajo es la de intentar profundizar más en como trabaja esta tecnología, que posibilidades ofrece, que requisitos necesitamos o que escenarios son las indicados.

Ante un desarrollo de despliegue BLE Mesh, no solo es necesario conocer el protocolo en si, sino que resulta imprescindible otorgarle funcionalidad: en última instancia, el protocolo es el medio por el que se envían los datos o la manera en la que se envían. Por ello, gran parte de este trabajo se centra en el desarrollo de un *firmware* capaz de funcionar con BLE Mesh sobre un dispositivo de bajo consumo.

Dentro de BLE Mesh, concretamente vamos a hacer uso del modelo sensor. Este modelo, sin entrar en detalles de lo que implica, nos permite tener una configuración y hacer uso de un sensor de manera genérica, es decir, independientemente del sensor que usemos el modelo se adecua a cualquiera independientemente del tipo de sensorización que hagamos o la configuración que requiramos. El hecho de crear un firmware de propósito general que funcione con el modelo sensor es motivador ya que no se ha encontrado ningún ejemplo completo de código abierto, salvo fragmentos de código que podemos encontrar por la red. Esto implica un esfuerzo para el desarrollo, ya que hay que separar y organizar bien el código, intentar cumplir unos requisitos mínimos para un tipo de arquitectura *mesh* etc. Esto tiene que ver más con la ingeniería o arquitectura de un proyecto, y es aquí donde se encuentra el tercer pilar del trabajo: organizar el desarrollo de un proyecto *hardware/software* de envergadura.

En cualquier aplicación donde se pueda configurar o controlar ciertos dispositivos, siempre encontraremos un panel de control. Si además tenemos los conocimientos técnicos necesarios, podremos hacerlo nosotros mismos con HTML, CSS y Javascript/jQuery o usando

algún *framework* CSS, aunque en el mercado existen multitud de herramientas que nos permiten desarrollar *dashboards* mediante herramientas basadas en flujos, como puede ser Node-Red. Este es otro motivo por el que enfocar este trabajo y es la creación de un *dashboard* junto con una herramienta de linea de comandos desde la cual poder controlar nuestra red *mesh*, monitorizarla, y sacarle el mayor provecho dentro del marco de un despliegue de una red BLE Mesh. Esto nos permitirá tener un proyecto completo con todas las partes importantes desarrolladas.

1.2. Objetivos

El objetivo de este trabajo es doble. Por un lado, profundizar y documentar en la memoria el protocolo BLE Mesh, concretamente todos aquellos aspectos relevantes para entender mejor el *firmware* desarrollado y todos los elementos que forman parte de un despliegue BLE Mesh típico. Por otro lado, desarrollar un *firmware* que funcione bajo el *framework* ESP-IDF [5] capaz de funcionar en una red BLE Mesh junto con herramientas para monitorizar e interactuar con dicha red. Estos objetivos generales se pueden dividir en los siguientes objetivos específicos:

- Documentar el protocolo Bluetooth Low Energy sobre el cual está construido Bluetooth Mesh. Esto nos permitirá conocer mejor el protocolo para dar paso a los modelos, pieza clave en la parte de BLE de nuestro firmware.
- Comparar BLE Mesh con otras tecnologías similares para dar argumentos de porque se ha elegido dicho protocolo.
- Explicar que material es el que se ha usado para este proyecto. Sera importante de cara a tener una visión global del despliegue propuesto y que tipo de dispositivos necesitamos.
- Documentar y explicar el desarrollo propuesto para un despliegue de una red de dispositivos que trabajen bajo BLE Mesh.

- Documentar y explicar el desarrollo propuesto para un panel de control donde monitoricemos la información recibida por parte de la red BLE Mesh.
- Documentar y explicar el desarrollo propuesto para una herramienta por linea de comandos (CLI) que nos permitirá interactuar con nuestra red.
- Documentar y explicar la configuración propuesta para el proyecto de todos los componentes del despliegue.

Por tanto, la memoria se dividirá en una serie de capítulos donde trataremos los temas mencionados. El capitulo 1 se presenta, a alto nivel, una visión general del trabajo desarrollado. Después, continuaremos con el capitulo 2 donde veremos de manera teórica aspectos relacionados con Bluetooth Mesh en relación al trabajo desarrollado, para pasar al capitulo 3 donde veremos el desarrollo puramente dicho, tanto del *firmware* como de los demás componentes implicados. Después, en el capitulo 4 veremos lo relacionado con el provisionador y el proceso de provisionamiento en relación con el protocolo BLE Mesh. Por último, veremos el capitulo 5 donde hablaremos de la CLI y del *dashboard*.

1.3. Arquitectura del despliegue

La idea de este proyecto es ofrecer un *firmware*, bajo el *framework* de Espressif, con capacidad para funcionar dentro de una red BLE Mesh. Por otro lado, al trabajar con el modelo sensor, debemos poder obtener las mediciones de los nodos u otra información interesante y poder cambiar parámetros del modelo.

Una arquitectura de red *mesh*, es una arquitectura de malla donde, concretamente para el caso de BLE, tenemos dos tipos de nodos: cliente y servidor. El servidor se encarga de tomar mediciones, proporcionar información sobre sus características o modificar alguna de ellas. Por otro lado, el cliente sera el encargado de ejecutar acciones en forma de mensajes BLE Mesh. Esto significa que pedirá un valor de medición, información sobre alguna característica o modificara alguna de ellas para algún servidor o servidores concretos. Por otra parte,

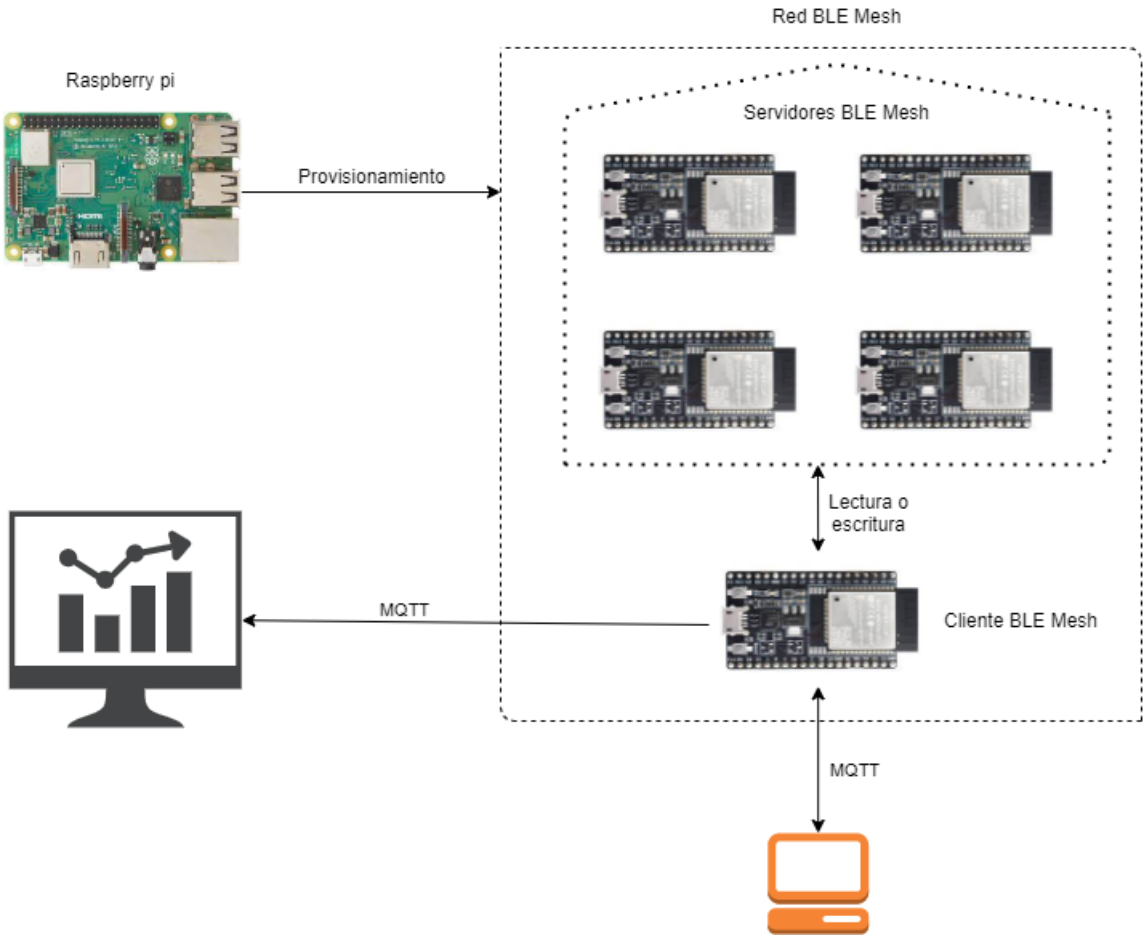

Figura 1.2: *Arquitectura del despliegue.*

necesitamos provisionar todos los nodos con claves de red y aplicación para que puedan comunicarse entre si y que la información viaje de manera segura. Reuniendo estas ideas, lo que se plantea es una arquitectura de despliegue como la mostrada en la figura 8.2. Si observamos el esquema, tenemos por un lado la red BLE Mesh, la cual está compuesta por las placas de desarrollo ESP32 que funcionaran como servidores salvo una de ellas que sera el cliente. Tenemos una Raspbery pi que se encargara de provisionar los nodos de la red. El dashboard tendrá como función pintar cualquier tipo de información interesante de manera más visual, como puede ser las mediciones que se vayan pidiendo a los servidores. Por último, tenemos una pequeña herramienta por linea de comandos (CLI) que usaremos para interactuar con la red.

Esta arquitectura es la propuesta en este trabajo ya que se adecua al tipo de arquitectura de red que usamos y es un modelo genérico para cualquier tipo de aplicación BLE Mesh. Es posible que pueda variar en función de la casuística que se tenga pero, de manera general, tendremos los elementos mencionados con esta disposición. Además, potencialmente, tanto el *dashboard* como la herramienta de comandos, se pueden desplegar en la propia Raspberry Pi.

1.4. Estructura del proyecto

Como todo proyecto de programación, se busca que ejecute una funcionalidad concreta o que resuelva un problema dado pero, y no menos importante, también hay que buscar que este bien organizado desde el punto de vista del diseño. Esta ha sido la idea principal a la hora de crear los archivos necesarios para el proyecto, no solo que podamos obtener alguna medición de los sensores, sino que si alguna persona quiere hacer alguna modificación o mejora, le sea lo más fácil posible sin romper algún otro punto del firmware. También, se ha hecho una implementación en cascada, es decir, que cuando un componente se inicializa, también inicialice aquellos que deban estar preparados o que necesite el modulo que se este inicializando en este momento. Por otra parte, como el *framework* utilizado ha sido ESP-IDF, todo el código se ha desarrollado en lenguaje C con programación modular para separar el código en archivos .c y .h y así tener mejor encapsulación y organización.

1.4.1. Componentes del proyecto

Conforme a lo visto en la figura 8.2 donde hablábamos de la arquitectura, tenemos los siguientes componentes:

- **Raspberry Pi:** Sera nuestro provisionador para los dispositivos que formen parte de la red *mesh*. Necesitaremos habilitarle opciones del kernel específicas para que tenga las funcionalidades requeridas por BLE Mesh para el intercambio de claves seguro en el proceso de provisionamiento. Además, potencialmente, no solo puede funcionar como

provisionador sino que también podríamos tener alojados tanto el dashboard como la CLI.

- **ESP32 como servidor BLE Mesh:** Al funcionar como servidor, tiene que estar a la escucha de cualquier mensaje BLE dirigido por parte del cliente.

- **ESP32 como cliente BLE Mesh:** En caso de ser cliente, debe tener comunicación con el exterior y con la red BLE Mesh. Por la parte exterior, debe interpretar la información recibida para ejecutarla o transformarla en un mensaje BLE Mesh para lanzarlo a la red, ya sea de manera puntual o periódica.

- **CLI:** Debe ser capaz de mandar mensajes al cliente BLE Mesh para ejecutar una acción concreta. También debe ser capaz de recibir *feedback* del cliente BLE Mesh para saber si la acción se ha ejecutado con éxito o no.

- ***Dashboard:*** Ya que tenemos un numero de servidores tomando mediciones de un sensor, tenemos que ser capaces de visualizarlos en un panel que sea fácil de leer y pueda interpretarse correctamente para su monitorización. En este sentido, el client preguntara a los servidores por dichas mediciones para que el cliente las exporte al exterior y el *dashboard* las pueda ilustrar.

1.4.2. Hardware/Software utilizado

Las placas de desarrollo que utilizaremos son las placas ESP32-DevKitC V4 [4]. Estos dispositivos son de dimensiones bastante reducidas, ofrecen WiFi y BLE, tiene una interfaz GPIO bastante amplia para usar ADCs, comunicación mediante I2C etc. A estas placas uniremos mediante la interfaz GPIO sensores Adafruit si7072 [1] los cuales ofrecen mediciones de temperatura y humedad. Estos sensores ofrecen una medición de humedad relativa de $\pm 3\,\%$ con un rango de 0-80 % RH. Por otro lado, ofrece una precisión en la medición de temperatura de $\pm$ 0.4°C. Para la parte de provision de dispositivos, se dispone de una Raspberry Pi 3b+ [9] bastante conocida donde tenemos un procesador Cortex-A53 (ARMv8) 64-bits a

una frecuencia de 1.4 GHz. Dispone de 1GB LPDDR2 de memoria SDRAM y conectividad Wifi de doble banda además de Bluetooth 4.2 y BLE.

1.4.3. Metodología y planificación

El desarrollo es la parte más importante de este trabajo por lo que es necesario tener especial cuidado a la hora de diseñar e implementar el código a desarrollar; por ello, se utilizó la plataforma GitHub para subir todo el código de este proyecto [18]. En los dispositivos ESP32, el desarrollo se ha llevado a cabo bajo el *framework* de Espressif, el cual está escrito en C. Por este motivo, y para tener una mayor organización, se ha escrito el código con programación modular para así no afectar en varios sitios del código sino que cada modulo tenga un cometido claro y, si se desea realizar futuras modificaciones, no afecte a otras partes del firmware. La herramienta por linea de comandos también puede estar sujeta a modificaciones o nuevas características por lo que también debe ser desarrollada pensando en estas circunstancias. Por ello, ha sido desarrollada en Ruby, que nos ofrece la flexibilidad de los lenguajes interpretados además de ofrecer una biblioteca bastante potente para lectura de parámetros por linea de comandos.

Capítulo 2

BLE Mesh

2.1. Bluetooth Low Energy

Bluetooth Low Energy (BLE) [2] es una tecnología de comunicación inalámbrica que funciona en la banda 2.4 GHz, utilizada típicamente en aplicaciones donde se necesite un consumo energético reducido, por ejemplo porque se dispone de dispositivos conectados a baterías que deben funcionar durante meses o años.

La primera versión de Bluetooth fue lanzada en 1994. Comenzó como una tecnología para sustituir comunicaciones cableadas a corta distancia para periféricos como ratones o teclados inalámbricos para uso domestico, en el uso de PDAs (populares en los 2000) o incluso sustituyendo a la tecnología de infrarrojos en los teléfonos móviles para transferir información.

Aunque Bluetooth apareció en 1994, Bluetooth Low Energy (BLE) no surgió hasta la especificación 4.0 de Bluetooth en 2010 pasando, a partir de ese momento, a denominarse Bluetooth Classic a la anterior especificación. BLE, de hecho, no fue una actualización de Bluetooth sino que nació como una tecnología que utiliza Bluetooth enfocada en aplicaciones IoT donde se envían pequeñas cantidades de información a velocidades bajas. Para entender mejor cual es la diferencia entre Bluetooth Classic y BLE, se pueden citar las siguientes diferencias:

- Bluetooth Classic se usa para aplicaciones de streaming, como puede ser el sistema de

audio de los vehículos actuales o transferencia de archivos.

- BLE se usa para datos de sensores, control de dispositivos o aplicaciones con bajos requerimientos de ancho de banda.
- BLE se caracteriza por un consumo energéticos bajo desde la propia especificación.
- Bluetooth Classic no está optimizado para tener un consumo energético reducido pero tiene una mayor velocidad de transferencia.
- BLE opera a través de 40 canales frente a los 79 de Bluetooth Classic.
- Las conexiones en BLE son más rápidas (tiene 3 canales de descubrimiento) frente a los 32 canales para Bluetooth Classic, lo que se resume en conexiones más lentas.

Principalmente, estas son las diferencias que hay entre Bluetooth Classic y BLE. Centrándonos más en BLE, también es necesario destacar que existen ventajas y desventajas en el uso de esta tecnología. Una de las principales ventajas es que, por diseño, presenta un consumo energético menor, por lo que si la distancia entre nodos de nuestra aplicación no es muy grande (de unos 6-8 metros), es una tecnología conveniente: podemos disponer de una batería con la que el nodo pueda operar durante meses o años. Otra ventaja, en cuanto al desarrollo, documentación, o simplemente curiosidad es que no tiene coste alguno acceder a la información oficial del protocolo (podemos por ejemplo, descargar especificaciones y manuales desde la página de Bluetooth SIG [12]). De cara al coste de producción también existen ventajas, ya que es más barato fabricar un chip para BLE que para otras tecnologías similares. Como última ventaja, cabe se señalar que hoy en día muchos productos incorporan BLE, por lo que cualquier aplicación bajo esta tecnología es capaz de funcionar con casi cualquier dispositivo del mercado. Las principales desventajas tienen que ver con el diseño inicial de Bluetooth. La primera de las desventajas es el flujo de datos, ya que está limitado por la capa física del protocolo (PHY). Está sujeto a la transmisión de radio, concretamente al ratio de transmisión. Además, también depende de la versión de Bluetooth

que usemos. Por otra parte nos encontramos con que Bluetooth fue diseñado para funcionar en distancias cortas, las cuales se ven aun más limitadas por: la banda 2.4 GHz que se ve afectada por obstáculos como pueden ser el metal, las paredes, agua y más con las personas; rendimiento y diseño de las antenas BLE, caja física donde tengamos el dispositivo encerrado o la orientación del dispositivo. Por otra parte, tenemos una tecnología o protocolo que nos permite alejarnos de Wifi, aunque sin Wifi no podríamos exportar hacia Internet los datos que manejemos con BLE.

Resumiendo, estos son los principales parámetros a tener en cuenta para adoptar o no un despliegue basado en BLE:

- **Tiempo de vida de la batería:** si nuestros dispositivos van a estar en entornos donde tengamos difícil acceso.
- **Transmisión de datos:** si necesitamos transmitir grandes cantidades de información o pequeños paquetes de datos.
- **Área a cubrir por numero de nodos:** si necesitamos un área extensa o pequeña o si disponemos de muchos nodos BLE para usar.

Dentro del marco de este proyecto encaja perfectamente esta tecnología ya que los datos a transmitir van a ser datos de sensores. Tampoco es un requisito el envío constante de información, sino que éste será periódico (cada ciertos segundos) por lo que los nodos permanecerán la mayor parte del tiempo apagado, con un consumo energético menor). Además, como veremos más adelante, BLE Mesh proporciona una pieza muy interesante llamada *modelo* del que no disponemos en Wifi Mesh, por ejemplo, cuya funcionalidad es muy práctica en el ámbito de nuestro proyecto.

2.1.1. Arquitectura Bluetooth Low Energy

La arquitectura de Bluetooth Low Energy se divide en 3 capas, y estas a su vez en diferentes partes como vemos en la Figura 2.1:

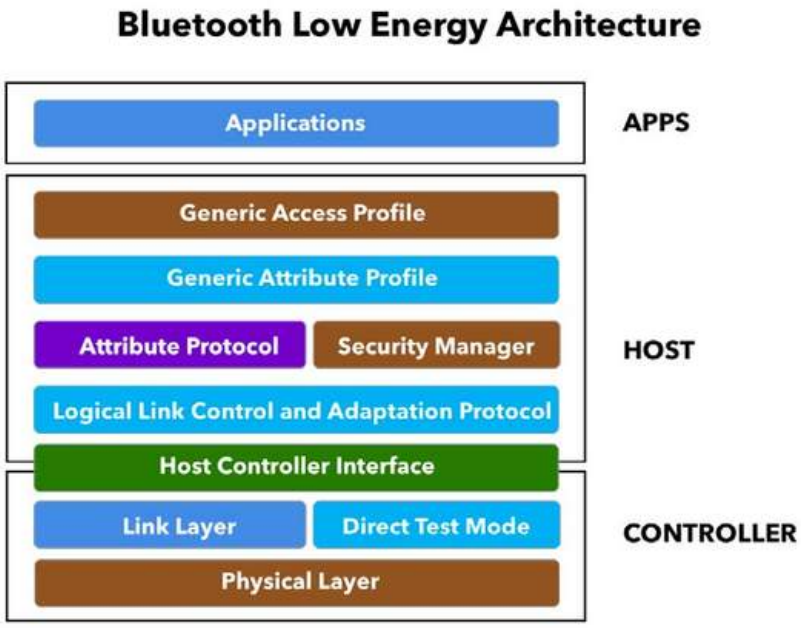

Figura 2.1: *Arquitectura Bluetooth Low Energy [2].*

- **Capa física (PHY):** Esta capa usa la radiofrecuencia para comunicarse. Se encarga de modular o demodular los datos para las capas superiores. Trabaja en la banda de 2.4GHz.

- **Capa de enlace:** Es la capa que interconecta con la capa física, provee abstracción a las capas superiores y actúa como interfaz de tal manera que capas superiores puedan interactuar con la radio. Se encarga de gestionar el estado de la radio, así como los requisitos de sincronización para cumplir con la especificación Bluetooth Low Energy.

- **Direct Test Mode:** Tiene como propósito testear aspectos relacionados con la radio (potencia de transmisión, por ejemplo).

- **Host Controller Interface (HCI):** Es un protocolo estándar definido en la especificación Bluetooth que permite a la capa *host* comunicarse con la capa *controller*. La capa *controller* tiene como objetivo encargarse de la parte más física del protocolo, es decir, esta capa está implementada en el propio chip Bluetooth. Por otro lado, la capa

host funciona como una interfaz de abstracción para el desarrollador.

- **Logical Link Control and Adaptation Protocol (L2CAP):** Es una capa de multiplexación de protocolos: toma múltiples protocolos de las capas superiores y los traduce en paquetes dentro del estándar BLE para pasarlos a capas inferiores.

General Access Profile (GAP)

General Access Profile es un framework que define cómo los dispositivos BLE se comunican entre si. Esto no solo incluye tipos de mensajes o maneras en las que se comunican (*advertisements*, iniciar conexión, establecimiento de la conexión, seguridad) sino también los roles que pueden desempeñar los dispositivos BLE. Estos roles son:

- ***Broadcaster:*** El dispositivo envía mensajes de anuncio y no recibe paquetes o permite conexión con otros dispositivos.
- ***Observer:*** El dispositivo permanece a la escucha de mensajes de anuncio pero no inicia la conexión con el dispositivo que emite dichos paquetes.
- ***Central:*** El dispositivo está descubriendo y escuchando a otros dispositivos que envían mensajes de anuncio. Tiene la capacidad de conectarse con dispositivos que envían dichos mensajes de anuncio.
- ***Peripheral:*** El dispositivo envía mensajes de anuncio y acepta conexiones desde dispositivos *Central*.

Generic Attribute Profile (GATT)

Una vez que tenemos una conexión entre dos dispositivos BLE, con el objetivo de intercambiar información, necesitamos información sobre cómo es el formato de los datos expuestos o los procedimientos necesarios para acceder a dichos datos. De esto se encarga GATT, a través de dos roles: servidor y cliente. El servidor es el dispositivo que expone

los datos que contiene y maneja algún aspecto de su comportamiento que otros dispositivos pueden ser capaces de cambiar o controlar. Un cliente es el dispositivo que interactúa con el servidor con el propósito de leer los datos expuestos por el servidor y/o controlar su comportamiento. Todos estos datos que se exponen o las acciones relacionadas con algún cambio en el comportamiento del servidor se realizan a través de los siguientes conceptos de GATT:

- ***Attributes:*** se refiere a cualquier tipo de dato expuesto por el servidor y define la estructura de ese dato. Los servicios y características son tipos de atributos. Cada uno se identifica mediante un identificador único universal (UUID).
- ***Services:*** es un grupo de uno o más atributos donde alguno de ellos son características. El cometido es agrupar atributos relacionados para una determinada funcionalidad del servidor.
- ***Characteristics:*** las características son elementos de datos que se refieren a un estado interno concreto del dispositivo o quizás a algún estado del entorno que el dispositivo puede medir mediante un sensor. También, las características pueden representar datos de configuración como la frecuencia con la que se quiere medir algún parámetro. En cualquiera de los casos, la forma en que un dispositivo puede exponer dichos datos a otros dispositivos para que los utilicen a través de Bluetooth es poniéndolos a disposición como una característica. Una característica siempre forma parte de un servicio y representa el dato o los datos que el servidor quiere exponer al cliente.
- **Profile:** definen el comportamiento, tanto del cliente como del servidor, en lo que respecta a los servicios, características e incluso conexiones.

2.2. BLE Mesh

La tecnología BLE resulta muy prometedora dentro del mundo de IoT, ya que exhibe un reducido consumo energético, que en último término es un requisito casi imprescindible en

este campo. Pero si se desea gestionar un conjunto de nodos, que estos nodos se comuniquen de manera lógica y segura entre sí, y además disponemos de cientos o incluso miles de dispositivos, debemos poner el foco en el siguiente escalón: BLE Mesh [11].

2.2.1. Nodos y elementos

BLE Mesh es una arquitectura de red donde todos los dispositivos que forman parte de la red se denominan nodos. Además, estos nodos usan BLE para comunicarse y cada uno puede enviar y recibir mensajes: es posible que un nodo envíe un mensaje a otro nodo directamente porque está en el rango de actuación de BLE, o puede que se reenvíe el mensaje a través de uno o varios nodos intermedios hasta llegar al nodo destino. Además, cada nodo puede desempeñar más de un rol concreto que se definen como *features* que pueden realizar:

- **Low-power feature:** Algunos nodos necesitan estar apagados la mayor parte del tiempo ya que, por ejemplo, están conectados a una batería por lo que necesitan apagar la radio y ahorrar consumo energético. Aquellos nodos que tenga esta característica, suelen trabajar en conjunto con nodos *friend*.

- **Friend feature:** Aquellos nodos que no tienen ninguna restricción de consumo eléctrico, son buena elección para trabajar en modo *friend*. Estos nodos tienen como finalidad básica almacenar cualquier mensaje para nodos receptores *low power*. Una vez que los nodos *low power* solicitan dichos mensajes, los nodos *friend* reenvían los mensajes almacenados.

- **Relay feature:** El nodo tiene la capacidad de recibir y reenviar mensajes. Esto da la capacidad de tener una red extensa de nodos y controlar el *flooding*. Es recomendable activarlo únicamente en nodos que no estén sometidos a limitaciones energéticas.

- **Proxy feature:** Permite transmitir y recibir mensajes *mesh* entre GATT y nodos BLE Mesh. Al igual que con *relay*, se recomienda en nodos con una fuente constante de energía.

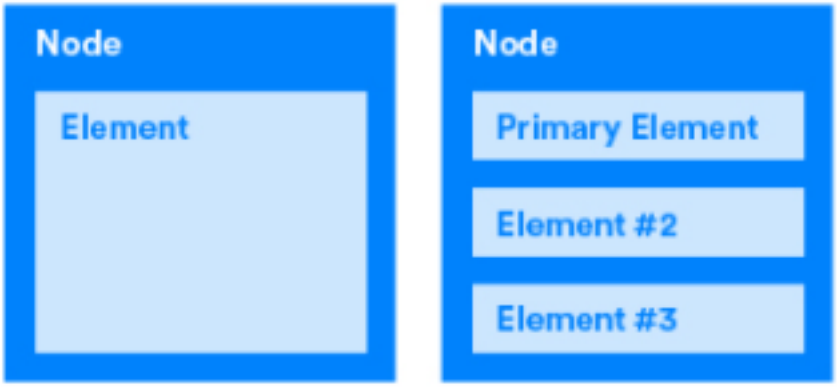

Figura 2.2: *Elementos en los nodos [11].*

Los nodos también están compuestos de *elementos*, donde siempre un nodo tiene un elemento principal denominado elemento primario. Estos elementos están compuestos de entidades que definen la funcionalidad del nodo y la condición de un elemento.

Cada elemento, al ofrecer una funcionalidad concreta, estará sometido a diferentes mensajes BLE Mesh, y además dispone de una dirección *unicast*.

2.2.2. Modelos y estados

Como vimos en el apartado anterior, los elementos implementan la funcionalidad del nodo. Pero realmente, los elementos contienen uno o más modelos que son realmente los que definen la funcionalidad y comportamiento del nodo a través de *estados*.

En el caso del modelo *Sensor*, que es el que se utilizará en este trabajo, el estado es el valor de medición que tengamos en el momento que hacemos una petición. Realmente define la condición en la que se encuentra el dispositivo. Otro ejemplo podría ser el modelo *on/off* (típicamente utilizado en el control de interruptores de iluminación) donde los estados son *on* y *off*, es decir, es la condición en la que se puede encontrar la luz en un instante determinado. Estos estados se intercambian mediante mensajes que sólo entiende el modelo en cuestión: por ejemplo, el modelo *on/off* no puede entender un mensaje `GET_STATUS` que solo entiende el modelo *sensor* para el cual se devuelve la medición. Los modelos se pueden identificar entre

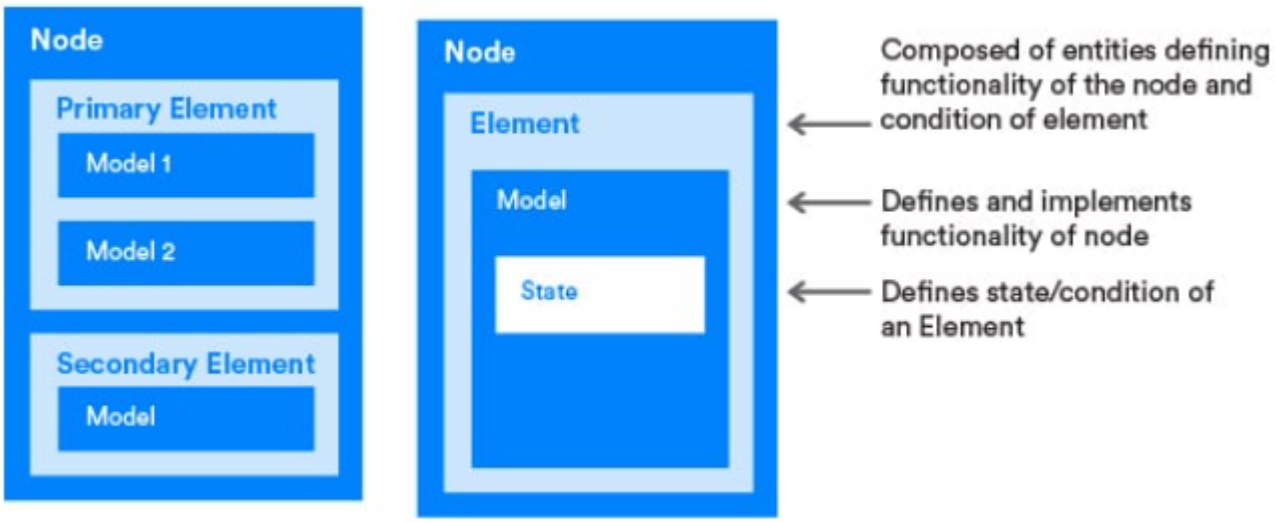

Figura 2.3: *Elementos, modelos y estados [11].*

si por un identificador de 16 bits definido por Bluetooth SIG; por ejemplo, el modelo sensor para el servidor tiene como identificador 0x1100 y el modelo sensor para el cliente tiene el identificador 0x1102. Por otra parte, un proveedor puede definir su propio identificador de modelo aunque en este caso debe ser de 32 bits, donde los primeros 16 corresponden con el identificador de la compañía y los otros 16 vienen definidos por el proveedor.

La comunicación de los nodos se realiza mediante un modelo cliente/servidor. El servidor expone los estados del elemento; el cliente, por su parte, solicita estos estados para consumirlos. Como vemos, el patrón de comunicación es el clásico cliente/servidor donde podemos tener 3 tipos de modelos:

- **Server model:** Compuesto por uno o más estados distribuidos en varios elementos. En nuestro caso, el modelo sensor en la parte del servidor expone las mediciones de temperatura y humedad.
- **Client model:** Define el conjunto de mensajes que entienden los modelos que tengamos en el servidor. Por ejemplo, el cliente del modelo sensor envía un mensaje `GET_STATUS` para obtener las mediciones de los sensores.

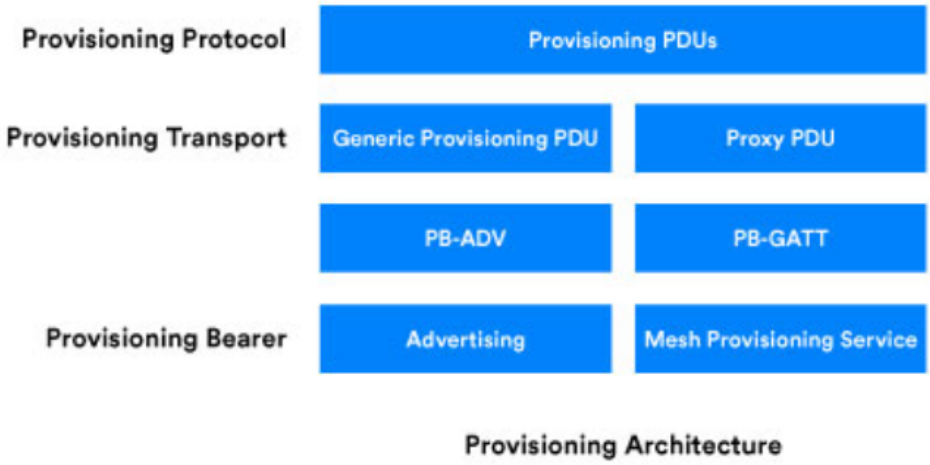

Figura 2.4: *Pila de provisionamiento junto a la pila BLE Mesh [10].*

- **Control model:** Combinan la funcionalidad del cliente y del servidor. Por ejemplo, podríamos tener el modelo sensor servidor para que, si la temperatura supera un umbral, envíe un mensaje a otro dispositivo que sea servidor para encender el aire acondicionado.

2.2.3. Provisionamiento BLE Mesh

El provisionamiento [10] es el proceso por el cual añadimos un nuevo dispositivo a la red *mesh*. Este proceso se realiza mediante el intercambio de PDUs, que no son más que paquetes de información pero con un formato concreto determinado por el protocolo, entre el provisionador y los dispositivos que no están provisionados. El provisionamiento tiene su propia pila, como se puede observar en la Figura 2.4 junto con la pila BLE Mesh.

La pila se divide en tres capas:

- **Provision Bearer:** Sirve para el transporte de PDUs referentes al proceso de provisionamiento entre el provisionador y un dispositivo no provisionado. Esta capa puede servir mediante:
  - **PB-ADV:** Usa los canales de anuncio para provisionar. PB-ADV se usa para

transmitir PDUs genéricas de provisionamiento. Un dispositivo compatible con PB-ADV debería realizar un escaneo pasivo con un ciclo de trabajo lo más cercano al 100 % para no perder ningún PDUs genérico de provisionamiento.

- **PB-GATT:** PB-GATT encapsula PDUs de provisionamiento dentro de operaciones GATT, involucrando el servicio de provisionamiento GATT. Se proporciona cuando no se soporta PB-ADV.

- **Provisioning protocol:** Define los requisitos para PDUs de provisionamiento, comportamiento y seguridad.

2.2.4. Proceso de provisionamiento

El proceso de provisionamiento es clave a la hora de crear nuestra red *mesh* ya que nos permite ir añadiendo nodos y hacer que los nodos se comuniquen entre si de manera segura. Este proceso se divide en varias fases que veremos a continuación en detalle.

Beaconing

Es el inicio del proceso de provisionamiento, en el que un dispositivo se anuncia a la espera que un provisionador desee añadirlo a la red. Si un dispositivo no provisionado soporta PB-ADV, se anuncia como un *beacon de dispositivo no provisionado*. Esto requiere un formato de paquete especifico usado por el dispositivo para que el provisionador pueda descubrirlo.

Cuando se usa PB-GATT, un servicio GATT llamado *servicio de provisionamiento mesh*, soporta la gran parte del procedimiento de provisionamiento e interactúa con el provisionador. El la fase de *beacon*, el dispositivo no provisionado envía paquetes de anuncio a *broadcast* que incluyen el UUID del servicio de provisionamiento mesh. Este servicio es descubierto por el provisionador a través del proceso de escaneo estándar de BLE.

Invitation

Después de la fase de *beaconing*, el provisionador y el dispositivo a provisionar establecen un *bearer*[1] de provision, ya sea PB-ADV o PB-GATT en función de la interfaz que el dispositivo no provisionado tenga disponible. Una vez establecido el *bearer*, el provisionador envía un paquete *Provisioning Invite PDU* donde se incluye un campo que indica cuanto tiempo debe el dispositivo avisar al usuario de que está siendo provisionado. Por otra parte, el dispositivo responde con un *Provision Capabilites PDU*, donde se incluye la información de:

- El número de elementos que el dispositivo soporta.
- El conjunto de algoritmos de seguridad que soporta.
- La disponibilidad de su clave pública mediante la tecnología *out-of-band* (OOB). Esto significa que la clave se puede transmitir, no solo a través de Bluetooth, sino por ejemplo a través de NFC.
- La capacidad de emitir un valor al usuario.
- Si permite que el usuario introduzca un valor.

Este proceso lo podemos ver resumido en la Figura 2.5, donde el objetivo principal es que el dispositivo provea al provisionador con la información sobre el mismo para que luego, con esa información, el provisionador decida qué hacer en el siguiente paso.

Exchange Public Keys

El uso de BLE Mesh se aplica a casos de uso donde, principalmente, los dispositivos tienen restricciones en el cómputo por lo que tener que encriptar y desencriptar los mensajes no es viable. Dentro de la criptografía tenemos dos tipos de maneras de securizar la comunicación: de manera asimétrica o simétrica. Debemos tener en cuenta que el más adecuado para

[1]Portador. Medio de transmisión utilizado para construir uno o varios canales de comunicación

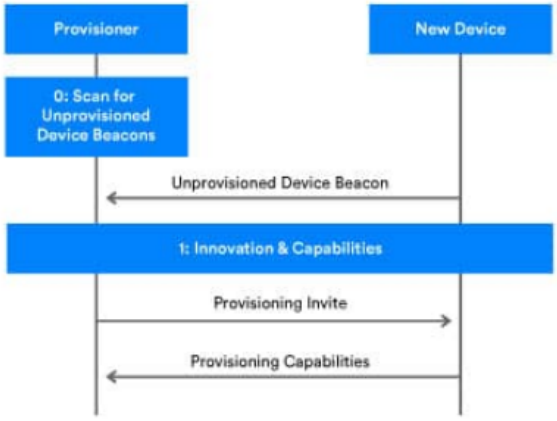

Figura 2.5: *Paso de invitación en el proceso de provisionamiento [10].*

entornos IoT es el simétrico debido a que requiere menos cómputo. En cambio, al encriptar y desencriptar con la misma clave, puede desembocar en que un atacante pudiera leer los mensajes. En el caso de BLE Mesh se usa una combinación de ambas técnicas:

- **Cifrado asimétrico:** En esta parte se una el conocido protocolo de curvas elípticas de Diffie-Hellman para establecer un enlace seguro entre el provisionador y el dispositivo. Hace uso de claves privadas y publicas para distribuir una clave simétrica para que ambos dispositivos puedan a partir de ahora encriptar y desencriptar los mensajes.

- **Cifrado simétrico:** Cualquier mensaje que se transmite a través de BLE Mesh es encriptado mediante AES-128.

Existen dos mecanismos para desarrollar la fase de intercambio de claves publicas: mediante un enlace Bluetooth o mediante un túnel OOB. En la fase de invitación, el dispositivo a provisionar, reporta si soporta enviar su clave publica a través de un túnel OOB. Si soporta OOB, el provisionador también hace uso del mismo para seguir con la comunicación con el dispositivo y envía, mediante el túnel OOB, un PDU de comienzo del provisionamiento. Si la clave publica del dispositivo a provisionar está disponible mediante el túnel OOB, entonces el provisionador comparte su clave publica con el dispositivo, se transmite una clave publica

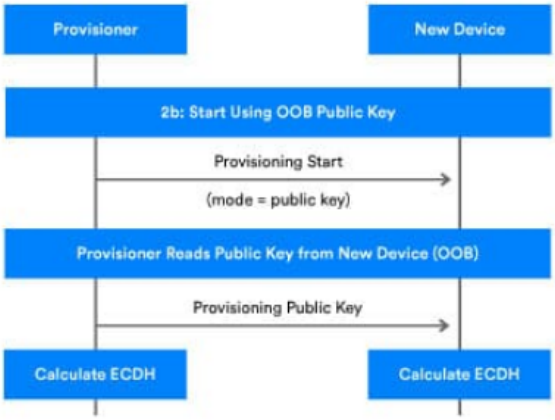

Figura 2.6: *Intercambio de claves mediante túnel OOB [10].*

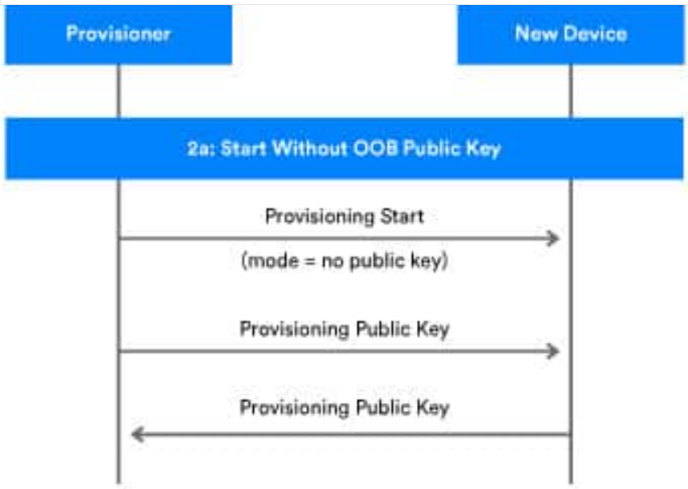

Figura 2.7: *Intercambio de claves a través de link Bluetooth túnel OOB [10].*

efímera por parte del provisionador al dispositivo, y el dispositivo no provisionado lee una clave publica estática usando la tecnología OOB apropiada, como si de un código QR se tratara 2.6.

En el caso de que no sea posible el uso del túnel OOB, ambas claves publicas se intercambian a través de un enlace Bluetooth como vemos en la Figura 2.7

Authentication

El siguiente paso es autenticar al dispositivo no provisionado mediante el método que haya elegido el provisionador. Estos métodos son: Output OOB, Input OOB, static OOB o No OOB.

Output OOB El dispositivo a provisionar elige un numero aleatorio y lo muestra a través de una secuencia con leds o por algun display por ejemplo. Después, el usuario lo introduce en el provisionador.

Input OOB Tiene el mismo funcionamiento que el caso anterior solo que quien elije el numero aleatorio es el provisionador para que luego el usuario lo introduzca en el dispositivo a provisionar. Después, el dispositivo envía un mensaje PDU que informa de que el numero aleatorio ha sido introducido.

Static OOB o No OOB En este caso, tanto el provisionador como el dispositivo a provisionar generar un numero aleatorio para que luego ambos comprueben dicho numero.

Distribución de datos de provisionamiento

Una vez que se ha completado el paso de autenticación, tenemos un canal de comunicación seguro entre el provisionador y el dispositivo no provisionado. El siguiente paso es que el provisionador genere los datos de provisionamiento y se los envíe al dispositivo. Estos datos de provisionamiento son:

- **Network Key:** Securiza la comunicación a nivel de red y se comparte por todos los nodos de la red. Se utiliza para crear subredes, es decir, determina a que subred pertenece un dispositivo.
- **Device Key:** Clave única que posee tanto el provisionador como el dispositivo que está siendo provisionado.

- **Key Index:** Con el objetivo de hacer una comunicación optimizada, las claves tienen un identificador global de 12 bits conocido como *key index*. Básicamente, este índice actúa como un identificador global.
- **IV index:** Es un valor de 32 bits que es compartido por todos los nodos con el objetivo de introducir aleatoriedad en los mensajes.
- **Unicast address:** Dirección unicast del elemento primario del nodo provisionado.

Una vez que se completa este paso, nuestro dispositivo ha sido provisionado y pertenece a nuestra red *mesh*.

2.2.5. Modelo *sensor* cliente, sensor servidor y *setup*

El modelo *Sensor* [16] ofrece funcionalidad genérica para operar con un sensor dentro de una red BLE Mesh, permitiendo a cualquier sensor exportar las lecturas que se vayan realizando hacia otros nodos de la red. Por otra parte, el *sensor setup server* permite configurar el sensor y el formato de sus datos.

Relación entre modelos

Los modelos sensor hacen uso de multitud de propiedades dentro de un pequeño numero de estados. Una propiedad se diferencia de un estado en que una propiedad contiene un identificador y un valor. El identificador nos informa de qué tipo de dato contiene la propiedad. Aprovechando estas propiedades, tenemos que estos modelos se pueden adecuar a cualquier tipo de sensor y datos en vez de tener varios modelos, mensajes y estados para cada tipo de sensor.

Si un elemento implementa el modelo *sensor server*, debe también tener el modelo *sensor setup server*, el cual lo extiende. Por otra parte, el modelo *sensor client* no está relacionado con los otros dos modelos y se puede usar de manera independiente. Esta relación puede observarse en la Figura 2.8.

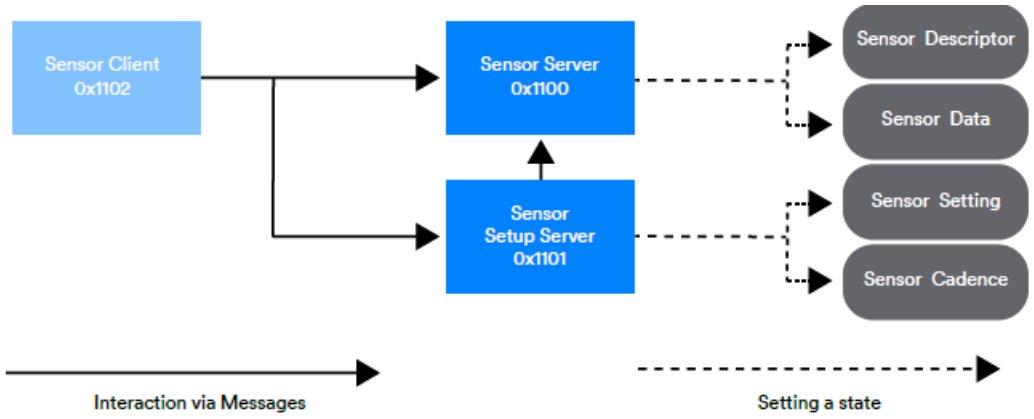

Figura 2.8: *Relación entre los modelos [16].*

Sensor states

Los modelos sensor se definen bajo un único estado compuesto denominado *sensor state*. Esto lo podemos ver en la definición de los estados dentro del framework de Espressif en el código 6, donde observamos dos estados que definen las dos mediciones que manejamos y dentro tenemos las propiedades. El resumen completo de lo que podemos tener en *sensor state* puede observarse en la Figura 2.9.

La información dentro de **sensor data** básicamente es un array de clave/valor donde la clave es un identificador de propiedad único y un valor en crudo. Según la especificación de BLE Mesh sobre las propiedades, algunas veces contienen un simple valor que puede actuar en la recepción de un mensaje, por ejemplo un mensaje `SENSOR_GET`. Otras propiedades definen arrays de datos para crear histogramas donde podemos acceder a un conjunto de ellos mediante un mensaje `SENSOR_COLUMN_GET`. Dentro de **sensor descriptor** disponemos de información que describe los datos del sensor disponibles y que son inmutables durante la vida útil del dispositivo: la *tolerancia* nos indica la magnitud de los posibles errores en la medición; *sensor sampling* indica que tipo de función se está usando para obtener las mediciones del sensor; el campo *sensor measurement* indica cada cuanto tiempo se está haciendo la media en el caso de que la función de sampling se base en la media; y el campo *sensor update interval* indica la frecuencia en la que se está tomando las mediciones del

Figura 2.9: *Información dentro de Sensor state [16].*

sensor. El campo *sensor setting* contiene propiedades configurables como por ejemplo la sensibilidad de un sensor de movimiento. Cada miembro de la lista está compuesto por un identificador de la propiedad a la que aplica la configuración concreta, el identificador de la propiedad que identifica a la configuración, un indicador de los permisos de la configuración (lectura, escritura) y el valor de la configuración en si. Por último, el campo *sensor cadence state* permite configurar la frecuencia a la que el sensor publica los datos de medición identificados con el *property id* que tengan. La frecuencia a la que se publican los datos se puede configurar en función de ciertas condiciones. Si el valor de medición cae por debajo de un rango, la frecuencia de publicación se puede cambiar de manera automática. En este sentido, el campo *fast cadence period divisor* indica cuando debería incrementarse la frecuencia de publicación cuando, como hemos dicho, el valor de medición cae por debajo de un rango.

Capítulo 3

Desarrollo del firmware

En este capitulo se presenta el *firmware* desarrollado, tanto para el cliente como para el servidor. Además, se profundizara en las partes que componen ambos firmware para entender mejor su estructura y funcionamiento.

3.1. ESP32 como servidor BLE Mesh

El servidor será el encargado de ir tomando mediciones del sensor para, cuando el cliente lo requiera, el servidor le devuelva el o los valores de sensorización requeridos. Esta petición donde el cliente pide una medición para el modelo sensor, que es con el que estamos trabajando, se realiza mediante un mensaje GET_STATUS, cuyo opcode se puede obtener mediante la macro de ESP-IDF llamada *ESP_BLE_MESH_MODEL_OP_SENSOR_GET*.

El firmware del servidor, en comparación con el del cliente, es bastante sencillo:

- **humidity_sensor:** Este módulo es el encargado de obtener mediciones de humedad del sensor. También ofrece una función para ser lanzada como tarea y que se vayan obteniendo automáticamente los datos.
- **temperature_sensor:** Este módulo tiene el mismo cometido que el módulo humidity_sensor pero para la obtención de la temperatura.
- **si7021_i2c:** Este módulo se encarga de encapsular las funcionalidades de los módulos

humidity_sensor y temperature_sensor para manejar toda la funcionalidad del sensor desde un único punto. Además, es el encargado de lanzar las tareas de sensorización.

- **sensor_model_server:** Este módulo es el encargado de gestionar todo lo relacionado con BLE Mesh.

Por último, se ha creado también un archivo de cabecera llamado si7021_utils.h con macros de uso genérico para el sensor. Por ejemplo, en este módulo tenemos la dirección del esclavo a la hora de hacer alguna petición por i2c al sensor.

3.1.1. Main en servidor BLE Mesh

Conforme a lo explicado en el apartado 1.4, tenemos una inicialización en cascada. Como podemos ver en la figura 3.1, la función main se encarga de inicializar los componentes necesarios para que el servidor funcione correctamente: por un lado inicializa la parte de BLE MEsh y por otra parte lanza la inicialización del sensor.

Lo primero que hace el main es inicializar el almacenamiento no volátil como vemos en el código 1 para luego llamar a la función de inicialización del módulo sensor i2c 3.1.2. Esta a su vez, llama a las funciones de inicialización de humidity_sensor y temperature_sensor para que creen sus buffers circulares y lancen sus tareas de sensorizacion. Después, se inicializa la parte de BLE para que este operativa y se puedan recibir mensajes.

3.1.2. Módulo del sensor si7021_i2c.c

Esta parte del código está desarrollada en el archivo si7021_i2c.c. El nombre proviene del sensor usado para este proyecto [1] capaz de obtener la temperatura y la humedad. La función más importante donde se inicializa el módulo es la función *si7021_init* como vemos en el código 2.

Esta función tiene dos objetivos: inicializa i2c de manera común para ambos datos de sensorización llamando a la función *initialize_i2c*. Después se inicializa la parte dedicada a pedir la temperatura al sensor mediante la función *si7021_init_temp*, donde se crea

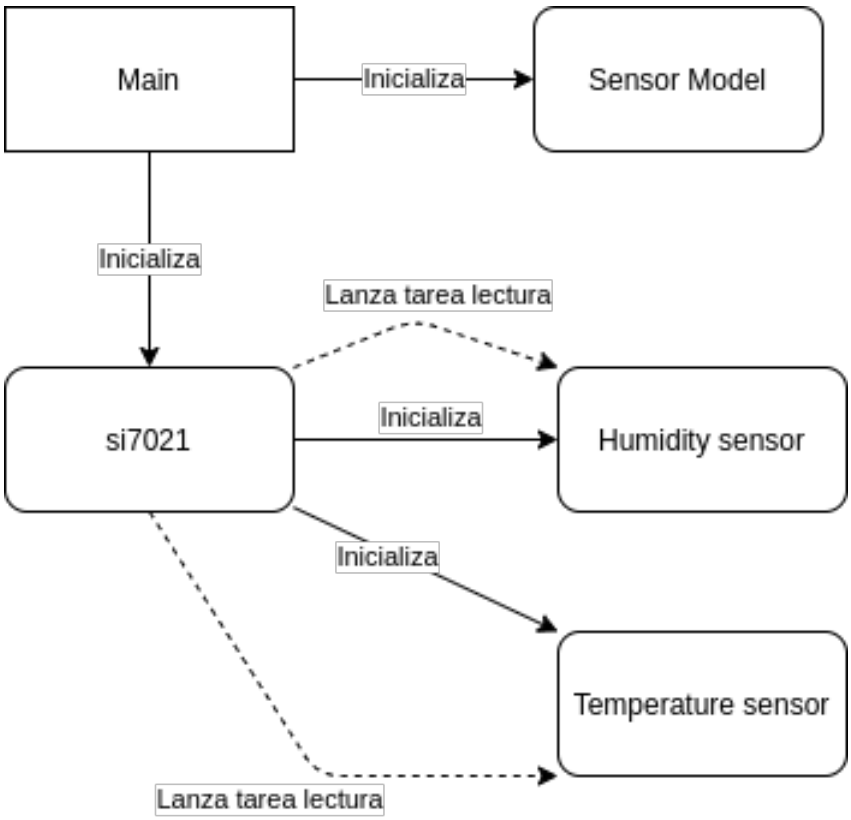

Figura 3.1: *Arquitectura Servidor BLE Mesh.*

```
void app_main(void){

    /* NVS */
    ESP_LOGI(TAG, "Initialising NVS");

    esp_err_t err = nvs_flash_init();
    if (err == ESP_ERR_NVS_NO_FREE_PAGES) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK(err);


    ESP_LOGI(TAG, "Initialising sensor si7021");
    ESP_ERROR_CHECK(si7021_init());

    ESP_LOGI(TAG, "Initialising Ble and Bluetooth with sensor model -> server");
    ESP_ERROR_CHECK(ble_mesh_init());

    vTaskDelete(NULL);
}
```

Listing 1: Main en el servidor.

un semáforo para acceder al buffer circular, tanto para leer como para escribir. Una vez creado, se crea la tarea de lectura y se le pasa la función *task_read_temperature* que vemos en el código 3. Esta función, primeramente llama a *get_temperature* donde se construye la secuencia i2c marcada por el fabricante del sensor para recuperar la temperatura y se devuelve el resultado en las variables *sensor_data_h* y *sensor_data_l* donde se almacenan los bytes más y menos significativos del valor de la medición respectivamente. Si todo sale bien, se vuelcan *sensor_data_h* y *sensor_data_l* en un uint16_t para que regularice la medición llamando a la función *regularize_temperature*. Este paso es necesario ya que el valor devuelto por el sensor no es un valor legible, sino que tenemos que pasar el dato por una formula marcada por el datasheet del fabricante para tener la temperatura real. Una vez que se ha regularizado la temperatura, se llama por último a la función *insert_data* para insertar la medición en el buffer circular donde se espera a obtener el semáforo para insertar dicha medición. Más tarde, la tarea duerme un numero de segundos dados por la macro `DELAY_TIME_ITEMS`. Como vemos es un flujo bastante sencillo donde obtenemos

```c
static esp_err_t initialize_i2c()
{
    i2c_config_t conf;
    conf.mode = I2C_MODE_MASTER;
    conf.sda_io_num = I2C_MASTER_SDA_IO;
    conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
    conf.scl_io_num = I2C_MASTER_SCL_IO;
    conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
    conf.master.clk_speed = I2C_MASTER_FREQ_HZ;

    i2c_param_config(I2C_MASTER_NUM, &conf);

    return i2c_driver_install(I2C_MASTER_NUM, conf.mode,
                              I2C_MASTER_RX_BUF_DISABLE,
                              I2C_MASTER_TX_BUF_DISABLE, 0);
}

esp_err_t si7021_init()
{

    ESP_ERROR_CHECK(initialize_i2c());

    si7021_init_temp(); // initialize temperature measure
    si7021_init_hum();  // initialize humidity measure

    return ESP_OK;
}
```

Listing 2: Inicialización del sensor.

una medición y la guardamos. Por último, para obtener una medición, debemos llamar a la función *get_mean_temperature_data*, del módulo si7021_i2c, donde nos devolverá la media aritmética de los elementos dados por la macro WINDOW_SIZE. Esta función se llama desde la parte de BLE Mesh cuando el servidor recibe un mensaje de tipo GET_STATUS. Una vez inicializada la parte de la temperatura, se inicializa la parte del la humedad mediante la función *si7021_init_hum*. El flujo es exactamente igual que con el caso de la temperatura como vemos en el código 4, donde se leen los datos de la misma manera y se inserta en un buffer circular.

```c
static void task_read_temperature(void* params)
{
    int ret;
    uint8_t sensor_data_h, sensor_data_l;
    uint16_t bytes = 0;

    for(;;)
    {
        ret = get_temperature(I2C_MASTER_NUM, &sensor_data_h, &sensor_data_l);
        if (ret == ESP_ERR_TIMEOUT)
        {
            ESP_LOGE(TAG, "I2C Timeout");
        }
        else if (ret == ESP_OK)
        {
            ESP_LOGD(TAG, "data_h: %02x, data_l: %02x", sensor_data_h, sensor_data_l);

            bytes = (sensor_data_h << 8 | sensor_data_l);
            float temp = regularize_temperature(bytes);
            ESP_LOGI(TAG, "Sensor temp: %.02f [ºC]", temp);

            insert_data(temp);
        }
        else
        {
            ESP_LOGW(TAG, "%s: No ack, sensor not connected...skip...", esp_err_to_name(ret));
        }
        vTaskDelay(DELAY_TIME_ITEMS * 1000 / portTICK_RATE_MS);
    }
    vTaskDelete(NULL);
}
```

Listing 3: Tarea para leer la temperatura.

3.1.3. Módulo BLE Mesh sensor_model_server.c

Esta parte del servidor es la encargada de gestionar la parte de BLE Mesh que estará a cargo de responder a lo mensajes que el cliente le envié. Tendiendo en cuenta esto, su desarrollo es sencillo salvo en las partes donde tratamos con las mediciones.

Una parte importante en el modelo que usamos en este proyecto, es que podríamos tener más de un sensor conectado, o como es en nuestro caso, dos mediciones diferentes. Esto significa que a través de BLE Mesh podríamos pedir información de distintos tipos de sensorizaciones. Para contener la información de cada medición, debemos inicializar correc-

```c
static void task_read_humidity(void* params)
{
    int ret;
    uint8_t sensor_data_h, sensor_data_l;
    uint16_t bytes = 0;

    for(;;)
    {
        ret = get_humidity(I2C_MASTER_NUM, &sensor_data_h, &sensor_data_l);
        if (ret == ESP_ERR_TIMEOUT)
        {
            ESP_LOGE(TAG, "I2C Timeout");
        }
        else if (ret == ESP_OK)
        {
            ESP_LOGD(TAG, "data_h: %02x, data_l: %02x", sensor_data_h, sensor_data_l);

            bytes = (sensor_data_h << 8 | sensor_data_l);
            float hum = regularize_humidity(bytes);
            ESP_LOGI(TAG, "Sensor hum: %.02f %%", hum);

            insert_data(hum);
        }
        else
        {
            ESP_LOGW(TAG, "%s: No ack, sensor not connected...skip...", esp_err_to_name(ret));
        }
        vTaskDelay(DELAY_TIME_ITEMS * 1000 / portTICK_RATE_MS);
    }
    vTaskDelete(NULL);
}
```

Listing 4: Tarea para leer la humedad.

tamente la estructura proporcionada por Espressif del tipo *esp_ble_mesh_sensor_state_t* donde, como vemos en el código 5, tenemos información de cada sensor. Además, como veremos a continuación, estos parámetros son genéricos al uso de cualquier tipo de sensor:

- **sensor_property_id:** identificador único por cada sensor que estemos definiendo. En nuestro caso tiene como valor 0x0056 para la temperatura y 0x0080 para la humedad. Este dato sirve para pedir información concreta de un sensor por ejemplo mediante los mensajes GET_CADENCE, GET_SETTINGS o GET_SERIES.

- **descriptor.positive_tolerance:** es un indicador de la magnitud de un posible error

en la medición por el limite superior.

- **descriptor.negative_tolerance:** es un indicador de la magnitud de un posible error en la medición por el limite inferior.

- **descriptor.sampling_function:** indica el tipo de función que estamos usando para obtener las mediciones del sensor. En nuestro caso, como comentamos en la parte del sensor si7021 3.1.2, obtenemos la media aritmética. Este tipo de función se define con la macro ESP_BLE_MESH_SAMPLE_FUNC_ARITHMETIC_MEAN.

- **descriptor.measure_period:** indica cada cuando tiempo estamos, en nuestro caso, calculando la media aritmética. Entonces, este valor solo tiene sentido si usamos una función que haga algun tipo de media. En nuestro caso solo hacemos la media cuando se pide una medición, por tanto, este dato tendrá como valor la macro ESP_BLE_MESH_SENSOR_NOT_APPL_MEASURE_PERIOD ya que no aplica en nuestro proyecto.

- **descriptor.update_interval:** indica cada cuando tiempo estamos obteniendo mediciones del sensor. Como comentamos en la sección 3.1.2, tendrá como valor DELAY_TIME_ITEMS.

- **sensor_data.format:** Formato de datos que manejaremos del sensor. En este caso, es el formato ESP_BLE_MESH_SENSOR_DATA_FORMAT_A.

- **sensor_data.length:** tamaño del dato de medición.

- **sensor_data.raw_value:** referencia a una estructura que se maneja en capas más bajas del protocolo BLE creada por Zephyr. Esta estructura debemos inicializarla previamente con una llamada a NET_BUF_SIMPLE_DEFINE_STATIC.

Estos datos serán devueltos en respuesta a mensajes GET_DESCRIPTOR definido bajo la macro ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET. Se construu-

ye un buffer hexadecimal que contiene: la información de todos los sensores o la información del sensor cuyo *sensor_property_id* coincida con el *sensor_property_id recibido*.

```
#define SENSOR_PROPERTY_TEMP 0x0056  /* Sensor temperature */
#define SENSOR_PROPERTY_HUM  0X0080  /* Sensor humidity */

#define SENSOR_POSITIVE_TOLERANCE   ESP_BLE_MESH_SENSOR_UNSPECIFIED_POS_TOLERANCE
#define SENSOR_NEGATIVE_TOLERANCE   ESP_BLE_MESH_SENSOR_UNSPECIFIED_NEG_TOLERANCE
#define SENSOR_SAMPLE_FUNCTION      ESP_BLE_MESH_SAMPLE_FUNC_ARITHMETIC_MEAN
#define SENSOR_MEASURE_PERIOD       ESP_BLE_MESH_SENSOR_NOT_APPL_MEASURE_PERIOD
#define SENSOR_UPDATE_INTERVAL      CONFIG_DELAY_TIME_ITEMS

static esp_ble_mesh_sensor_state_t sensor_states[2] = {
    [0] = {
        .sensor_property_id = SENSOR_PROPERTY_TEMP,
        .descriptor.positive_tolerance = SENSOR_POSITIVE_TOLERANCE,
        .descriptor.negative_tolerance = SENSOR_NEGATIVE_TOLERANCE,
        .descriptor.sampling_function = SENSOR_SAMPLE_FUNCTION,
        .descriptor.measure_period = SENSOR_MEASURE_PERIOD,
        .descriptor.update_interval = (uint8_t) SENSOR_UPDATE_INTERVAL,
        .sensor_data.format = ESP_BLE_MESH_SENSOR_DATA_FORMAT_A,
        .sensor_data.length = 0, /* 0 represents the length is 1 */
        .sensor_data.raw_value = &temp_sensor_data,
    },
    [1] = {
        .sensor_property_id = SENSOR_PROPERTY_HUM,
        .descriptor.positive_tolerance = SENSOR_POSITIVE_TOLERANCE,
        .descriptor.negative_tolerance = SENSOR_NEGATIVE_TOLERANCE,
        .descriptor.sampling_function = SENSOR_SAMPLE_FUNCTION,
        .descriptor.measure_period = SENSOR_MEASURE_PERIOD,
        .descriptor.update_interval = (uint8_t) SENSOR_UPDATE_INTERVAL,
        .sensor_data.format = ESP_BLE_MESH_SENSOR_DATA_FORMAT_A,
        .sensor_data.length = 0, /* 0 represents the length is 1 */
        .sensor_data.raw_value = &hum_sensor_data,
    }
};
```

Listing 5: Estados del modelo sensor en el servidor.

Una vez visto cuales son los estados del dispositivo BLE Mesh, vamos a ver que modelos conforman nuestro servidor. No solo tendremos el modelo sensor, el cual juega el papel principal, sino que tenemos otros modelos básicos y necesarios en cualquier dispositivo BLE Mesh. Estos modelos los vemos definidos en el código 6 donde nos van a dar funcionalidad básica en cualquier red BLE Mesh. Por ejemplo, el primer modelo llamado *config* se encarga-

ra de almacenar las claves de red o aplicación en el proceso de provisionamiento. También se encargara de asociar el dispositivo a un grupo, es decir, que tenga una dirección de grupo. Por otro lado, el modelo *sensor setup* se encarga de configurar el sensor y el formato de sus datos. Todos estos modelos están bajo una estructura del tipo *esp_ble_mesh_model_t* donde se definen los modelos root. Estos modelos se caracterizan porque no son extendidos en funcionalidad mediante otro modelo. Por otra parte, el numero de elementos creados ha sido solo uno, donde tenemos los tres modelos utilizados en el servidor.

```
static esp_ble_mesh_model_t root_models[] = {
    ESP_BLE_MESH_MODEL_CFG_SRV(&config_server),
    ESP_BLE_MESH_MODEL_SENSOR_SRV(&sensor_pub, &sensor_server),
    ESP_BLE_MESH_MODEL_SENSOR_SETUP_SRV(&sensor_setup_pub, &sensor_setup_server),
};

static esp_ble_mesh_elem_t elements[] = {
    ESP_BLE_MESH_ELEMENT(0, root_models, ESP_BLE_MESH_MODEL_NONE),
};
```

Listing 6: Modelos usados en el servidor.

A la hora de tomar una medición, debemos preparar los datos de tal manera que se envíen de manera correcta a través de BLE Mesh. La pila de BLE Mesh que usa ESPIDF es la creada por Zephyr [17], por lo que en el desarrollo nos toca interactuar con esta implementación de manera superficial. Concretamente, hay una función llamada *ble_mesh_get_sensor_data* encargada de obtener la medición del sensor mediante el uso de buffers por parte de Zephyr. Estos buffers los usamos con funciones que ya nos proporcionan como se puede ver en el código 7. Primeramente se obtiene la medición en función del dato *sensor_prop_id* que nos envíen, para después limpiar el buffer con la función *net_buf_simple_pull_u8* y rellenar dicho buffer con el valor obtenido del sensor mediante la llamada a *net_buf_simple_push_u8*.

El resto de código del servidor ha sido reutilizado totalmente. El flujo básico cuando el dispositivo está provisionado, pasa por la función de callback *ble_mesh_sensor_server_cb* donde se analiza el evento recibido para enviar una información u otra. Además, para responder a los mensajes hay creadas funciones wrapper para no llenar de código la función de

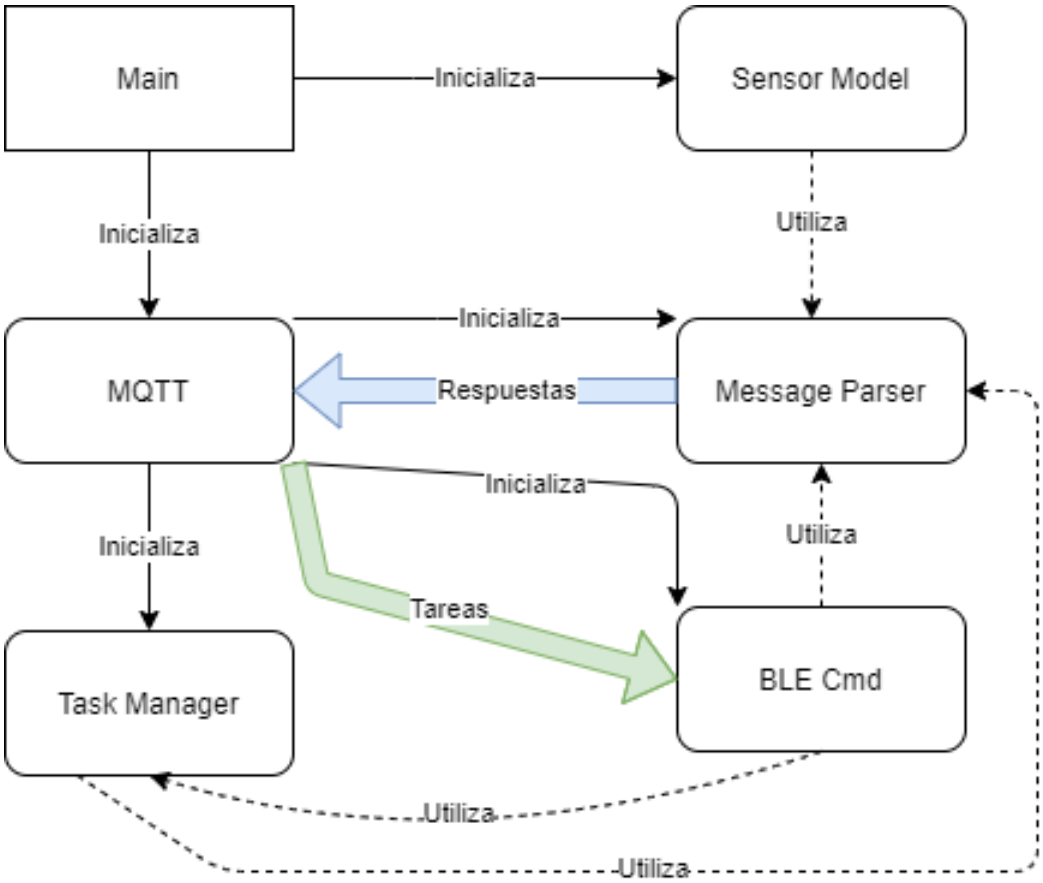

Figura 3.2: *Arquitectura Cliente BLE Mesh.*

callback.

3.2. ESP32 como cliente BLE Mesh

El cliente es la parte más potente de este trabajo. Ha sido donde más horas se han invertido en el desarrollo de este trabajo ya que es donde se centraliza la operativa de la red. Conforme se avanzaba en el desarrollo teniendo en mente los puntos descritos en la estructura del proyecto 1.4, se han ido creando diferentes módulos enfocados en una funcionalidad concreta. Además, para comunicarlos entre si, se han utilizado colas de mensajes. Todo esto se puede ver en la figura 3.2 donde vemos cual es la inicialización en cascada, la comunicación de los componentes y, con flechas de colores, donde están situadas las colas de mensajes.

3.2.1. Main en cliente BLE Mesh

El punto de entrada del firmware del cliente tiene la misma funcionalidad que el del servidor. Es el encargado de inicializar los componentes necesarios. En este caso, realiza las llamadas que vemos en el código 8 donde primero se inicializa el almacenamiento no volátil, después se llama a la función *wifi_init_sta* para que se conecte a la red wifi indicada en menuconfig 3.8. Si se conecta correctamente a la red Wifi, se sigue con la ejecución de la función donde inicializamos la parte de BLE Mesh y finalmente la parte de MQTT. La llamada *ble_mesh_init* simplemente levanta la posibilidad de provisionar el nodo e inicializa la pila BLE. La inicialización MQTT tiene algo más de complejidad como vemos en el código 9. En esta parte se crean dos colas: la primera dedicada a las acciones que queramos realizar (crear o eliminar tareas) que pasaremos a una tarea llamada *task_parse_json* para que, en el módulo *ble_cmd*. La segunda cola está dedicada a los mensajes, la cual se la pasamos a la tarea *task_send_response_mqtt* para que cuando le llegue un mensaje, esta tarea lo traduzca en un json para enviarlo por mqtt y también se le envía esta cola al módulo *messages*. A continuación se inicializa el *task manager*, donde guardaremos las tareas en ejecución y finalmente se inicializa el client mqtt. Cabe mencionar que el módulo *ble_cmd* no tiene función de inicialización, donde se podría haber creado la tarea *task_parse_json* e inicializado el *task manager*. La problemática era que se necesitaba una cola para comunicar el módulo *mqtt* con *ble_cmd* para que pudiera procesar los json relacionados con las tareas. Por este motivo, se decidió crear la tarea *task_parse_json* en la inicialización de MQTT, a parte de inicializar el *task manager*.

3.2.2. Módulo MQTT

Esta parte del firmware, a grandes rasgos, se encarga de traducir el JSON que recibe en estructuras internas para lanzar tareas o realizar alguna acción y viceversa, es decir, de transformar cualquier mensaje recibido por un servidor BLE Mesh y enviarlo por MQTT. Una parte fundamental en MQTT es crear los topics que necesitamos o más se adecuen

a nuestro propósito. En este proyecto tenemos datos que van a enviarse al dashboard, en este caso mediciones, y otros datos que son de consulta, por ejemplo mensajes de tipo GET_DESCRIPTOR. Por esta razón, se han creado 4 *topics*:

- **/sensors/results/dashboard:** Este *topic* se utilizara para publicar las mediciones.
- **/sensors/results/cli:** Este topic se utilizada para dar feedback al usuario o mandar información de algun tipo de mensaje que no sea automático. Por ejemplo el resultado de un mensaje GET_DESCRIPTOR se mandara a la CLI para que el administrador pueda luego lanzar tareas dirigidas a un estado concreto.
- **/sensors/actions/ble:** A este topic llegan las acciones que lancemos desde la CLI, es decir, todo lo que sea crear o eliminar tareas.
- **/sensors/commands:** A este topic llegan los mensajes referentes a acciones que no son de BLE. En el caso de este proyecto llegaran a este topic las peticiones de listar el numero de tareas en ejecución.

En función de a que topic llegue la información, se procesara la información como vemos en el código 10 donde, si el mensaje nos llega al topic */sensors/actions/ble* construimos un struct *mqtt_json* el cual tiene solo dos campos: json donde guardaremos el string que representa dicho json y un campo size donde guardaremos el tamaño del string recibido. En cambio, si el mensaje nos llega al topic */sensors/commands* ejecutaremos el comando que nos envíen. En nuestro caso solo soportamos devolver la lista de tareas.

Por último, la tarea que creamos en la inicialización de MQTT 9 llamada *task_send_response_mqtt* tiene como objetivo transformar un mensaje en un json. La definición de mensaje la veremos más adelante, en este punto basta con saber que es una estructura con varios campos que se vuelcan en un json. Esta tarea realiza los pasos que vemos en el código 11. Esta tarea es un bucle de tal manera que se espera a recibir un mensaje por la cola. Una vez recibido un mensaje, se hace uso del propio módulo *messages_parser* para que nos devuelva el

JSON asociado a la estructura que hemos recibido. Si el mensaje es de tipo GET_STATUS, entonces lo enviamos al dashboard, sino, lo enviamos a la CLI.

3.2.3. Módulo ble_cmd

Este módulo se encarga de recibir los JSON por parte del módulo de MQTT y procesarlos para realizar las operaciones requeridas. Esto se traduce en crear tareas automáticas o puntuales, además de eliminar tareas. El punto de entrada a este módulo lo tenemos en la tarea que vimos en el módulo de MQTT 9 llamada *task_parse_json*. Esta tarea consume elementos de la cola que MQTT va insertando cuando se envía algun mensaje al topic */sensors/actions/ble* para procesarlos como vemos en el código 12. Como vemos, si se obtiene de la cola un json, se crea una estructura *message_t* de tipo *PLAIN_TEXT*. Esto quiere decir que la estructura se va a usar para guardar mensajes que informen al usuario de lo sucedido, para dar feedback de como se ha ejecutado. Una vez creada la estructura, volcamos el json que hemos recibido en un string pasándole el tamaño para construirlo con el carácter fin de string al final. A continuación se hace una llamada a *parse_build_task* pasándose el string que representa el json y la estructura message_t para que nos devuelva una lista de acciones a realizar y el numero de acciones que tenemos que ejecutar. Las acciones es una array de estructuras que contienen los campos que cualquier tarea puede tener 5.3, además de un campo adicional para saber si dicha tarea tenemos que crearla o eliminarla. Una vez se ha procesado, recorremos el array de acciones y creamos la tarea o la eliminamos llamando a funciones que desempeñan tal papel.

La función *parse_build_task* encargada de parsear el json, comprueba que el json venga como una lista de json donde la clave sea *actions* como vemos en el código 13. Se encarga de revisar que el json tenga un formato especifico donde el par clave-valor tenga como clave el string *actions* y como valor tenga una lista de acciones. Si todo está correcto, nos recorremos la lista de acciones haciendo una llamada a la función *build_task* que se encargara de inferir si debemos crear una tarea o eliminarla como vemos en el código 14, además de reportar

si se ha podido parsear correctamente para aumentar el contador de acciones que se van a procesar en la tarea *task_parse_json*. Si solo nos mandan el campo nombre, es que debemos eliminar una tarea con dicho nombre, sino entonces debemos crear una tarea. A la hora de crear dicha tarea, se comprueba que el campo auto sea true y que el opcode requiere de dicho campo. Por ejemplo, lanzar repetidamente un mensaje GET_DESCRIPTOR no tendría sentido pero si un GET_STATUS. Debido a esto, la función *is_auto_required* comprueba que el opcode permita que se lance una tarea automática. Si es así, entonces el campo auto es true, sino false.

Cuando volvemos de la llamada a la función *parse_build_task*, ejecutamos la acción que nos devuelva dicha función: crear o eliminar tareas. En el caso de eliminar la tarea, nos iríamos a la función *delete_task* donde se intenta eliminar dicha tarea como vemos en el código 15. Básicamente, se va a buscar la tarea por el nombre al *task_manager* para comprobar que de verdad esa tarea está en ejecución. Si existe, se devuelve la tarea con el handler para poder eliminarla, tanto del scheduler como del *task_manager*. Si no está en ejecución, se comunica al usuario comunicándole que dicha tarea no existe.

Si por el contrario se quiere crear la tarea, se llama a la función *create_task* donde se realiza la lógica de la imagen 16. Si la tarea es automática, se consulta al *task_manager* por si existe. Si no existe se crea y se lanza la tarea, pero si existe, se informa al usuario y no se realiza ninguna acción. Por el contrario, si la tarea no es automática directamente se lanza y no se registra en el *task_manager* ya que se va a ejecutar una vez y se va a eliminar.

Cuando se lanza una tarea ya sea automática o no, se hace uso de la función 17, donde se crea un bucle en el caso de que sea automática y se duerme los segundos que le hayamos indicado. Si por el contrario no lo es, se elimina la tarea y se liberan recursos. Por último, para mandar el mensaje propiamente de BLE Mesh se hace uso de la función *ble_mesh_send_sensor_message* 20 declarada en el módulo sensor_model_client 3.2.5.

3.2.4. Módulo tasks_manager

Ya hemos hablado del el módulo *task_manager* en módulos anteriores como en el módulo *MQTT* cuando lo inicializábamos 3.2.2 o cuando hacíamos uso de el en el módulo *ble_cmd* 3.2.3. Pero realmente, hasta ahora, no hemos hablado de como está implementado.

El objetivo del módulo *task_manager* es ser una lista de tareas en ejecución donde, para cada tarea, guardemos la información que consideremos oportuna. Tal y como está desarrollado, de cada tarea se guarda el nombre y el handler que nos permite eliminarla. Para ello, se ha creado como una lista doblemente enlazada de estructuras como la que vemos en el código 18. Cuando se llama a la función *init_task_manager* se crea un puntero a una estructura de tipo *tasks_t* para poner los punteros a NULL y el numero de tareas a cero.

Repasar la funcionalidad de este módulo realmente no tiene mucho que aportar al trabajo ya que las operaciones son las que nos podemos encontrar en cualquier lista doblemente enlazada. Solo cabe mencionar que desde el módulo *task_manager* se proporciona una función llamada *queue_list_task* 19, la cual se ejecuta cuando en MQTT recibimos como comando la clave *tasks*, que se recorre la lista para crear una estructura *message_t* y posteriormente el módulo *message_parser* la encola para que el módulo *MQTT* la obtenga y la mande en formato JSON.

3.2.5. Módulo sensor_model_client

Tal y como paso con el servidor BLE Mesh 3.1, el cliente también se ha reutilizado en gran parte. Lo primero que se realizo fue quitar las capacidades de provisionador al cliente ya que la Raspberry pi es quien debe realizar ese papel. Una vez eliminado el código, había que adecuar el envió de mensajes BLE Mesh a los opcodes GET_STATUS y GET_DESCRIPTOR. Esto quiere decir que si le enviamos el dato *sensor_prop_id*, deberíamos mandárselo al servidor. Como se puede ver en el código 20, cuando se manda un mensaje GET_STATUS o GET_DESCRIPTOR, se rellena en la correspondiente estruc-

tura tanto el campo *sensor_prop_id* como el campo *addr*. Este último campo se rellena cuando se hace uso de la función *ble_mesh_set_msg_common* que prepara una estructura de tipo *esp_ble_mesh_client_common_param_t* que contiene parámetros genéricos para cualquier mensaje como puede ser el modelo que se usa, el timeout del mensaje, la dirección a la que va dirigido o el opcode del mensaje.

Todos las respuestas a los mensajes se recogen en el callback *ble_mesh_sensor_client_cb*. Inicialmente se declara una variable del tipo *message_t* del módulo *message_parser* para poder exportar esa información al exterior. Para el caso del mensaje con opcode GET_DESCRIPTOR se comprueba que el tamaño recibido sea de 8 bytes como vemos en el código 21. Esta comprobación tiene sentido ya que el descriptor para un estado es de 8 bytes, incluso aunque pidamos el descriptor para más de un estado tenemos que tener en cuenta que se recibirán tantos mensajes GET_DESCRIPTOR como estados pidamos. Si no tiene el tamaño solicitado, se devuelve un error informando al usuario.

En el caso de recibir una respuesta a un mensaje GET_STATUS, se llama a la función *publish_measure* que podemos ver en el código 22. Primeramente se hace una comprobación del primer byte que sea igual a 0xFF, lo que significa que ha habido algun error porque el *sensor_prop_id* que enviamos no existe. Sino, se procesan los datos para sacar las mediciones del buffer recibido y publicamos las mediciones a través de MQTT mediante la llamada a la función *send_message_queue* del módulo *messages_parser*.

Cuando enviamos un mensaje a una dirección de grupo, el evento que recibimos no es el opcode del mensaje sino que recibimos como evento principal ESP_BLE_MESH_SENSOR_CLIENT_PU
En el propio callback *ble_mesh_sensor_client_cb*, se ha hecho un apartado para dicho evento donde miramos el opcode real que se envió a la dirección de grupo como vemos en el código 23.

3.2.6. Módulo messages_parser

El módulo *messages_parser* comparte una cola de mensajes con el módulo MQTT donde este es el consumidor. El cometido es el de ofrecer estructuras para ir rellenándolas con la información pertinente para luego, en MQTT, se devuelva el JSON asociado. Para este módulo, es importante entender las estructuras que se han creado las cuales podemos verlas en el código 24. La estructura *message_t* tiene un campo que define el tipo de mensaje y un campo *message_content_t* que define el contenido que se va a manejar en función del tipo de mensaje mediante un union.

Cuando se crea una estructura *message_t*, se debe hacer mediante la llamada a la función *create_message* que vemos en el código 25 la cual nos prepara los campos en base al tipo de mensaje que le pedimos. También se proporcionan funciones helper para rellenar el contenido del mensaje. Estas funciones son las que vemos en el código 26 donde existe una por cada tipo de información que tratamos.

Las funciones *helper* que rellenan el contenido del mensaje tienen como objetivo no realizar el mismo código en varios puntos. En un principio podríamos tener una por cada tipo de mensaje pero en realidad esto no es así ya que:

- **add_message_text_plain(message_t* m, bool error_message, const char* message, ...):** Acepta numero de parámetros variable para construir el mensaje y un boolean para marcar el mensaje como error o no. Se usa para tipos de mensaje PLAIN_TEXT, TIMEOUT, ERROR y TASKS.
- **add_measure_to_message(message_t* m, uint16_t addr, uint16_t sensor_prop_id, int measure):** Se usa exclusivamente con mensajes de tipo GET_STATUS.
- **add_hex_buffer(message_t* m, uint8_t* data, uint16_t len):** Se usa con los tipos de mensaje GET_DESCRIPTOR y HEX_BUFFER.

Viendo esto, podría llevar a duda porque hay tantos tipos de mensajes. La clave está en que dos tipos de mensaje se pueden rellenar de la misma manera pero el JSON que se

va a formar es diferente. Por ejemplo, los tipos GET_DESCRIPTOR y HEX_BUFFER se rellenan igual pero el tipo HEX_BUFFER pinta el buffer como un string de manera literal. En cambio, el tipo GET_DESCRIPTOR se separan los datos del buffer en varios campos del JSON. Esto lo podemos ver en la función *message_to_json* 27 que se usa desde el módulo MQTT para transformar una estructura message_t en un JSON, la cual según el tipo de mensaje se llama a una función distinta o a la misma con diferentes parámetros como vemos en la imagen.

3.2.7. Módulo data_utils

Este último módulo agrupa funciones para cambiar entre tipos de datos 28. Por ejemplo, cuando recibimos un json para crear una tarea, el campo addr llega como un string pero en el firmware se necesita que sea de tipo uint16_t que es el tipo de dato que se maneja en BLE. También se usan en el módulo messages_parser por la misma razón ya que cJSON no acepta tipos que se manejan en BLE.

3.3. Menuconfig

El menuconfig es una herramienta que nos permite definir macros que luego podemos usar en el código. Ya hemos hablado de alguna de ellas, sobretodo las que nos ofrece el framework espresiff. Ahora, vamos a ver cuales han sido las macros definidas para el firmware que son parte del desarrollo realizado y así poder ofrecer una configuración un poco más flexible a este nivel. Todas las macros que queramos definir o entradas de menú nuevas, deben crearse en el archivo *Kconfig.probuild* dentro del proyecto. Una vez creadas, debemos ejecutar *idf.py fullclean* para limpiar el build hecho anteriormente, si es que se ha realizado, y luego ejecutamos *idf.py menuconfig* para entrar en dicho menú. Una vez entramos, veremos una entrada dedicada al proyecto llamada **TFM Configuration** como vemos en la imagen 3.3. Además, cabe mencionar que como tenemos dos firmware, uno para el cliente y otro para el servidor, tendremos dos menuconfig para ambos firmware.

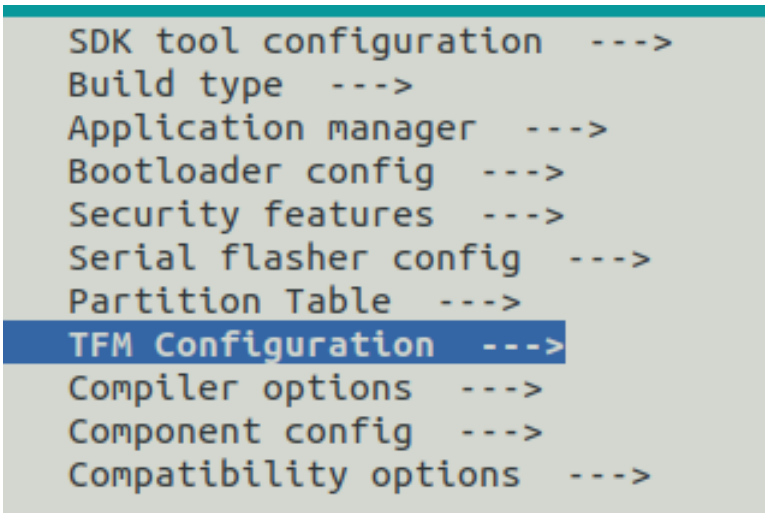

Figura 3.3: *Entrada de menuconfig dedicada al proyecto.*

En el lado del servidor tenemos las dos entradas mostradas en la imagen 3.4 donde la primera está dedicada a parámetros propios del bus i2c como vemos en la imagen 3.5. La segunda entrada está dedicada a la obtención de mediciones como vemos en la imagen 3.6 donde:

- El primer parámetro es el tamaño del buffer donde vamos guardando las mediciones.
- El segundo parámetro es el numero de mediciones que vamos a coger para calcular la media aritmética.
- El tercer parámetro es el numero de segundos que va a dormir la tarea que se encarga de obtener mediciones.

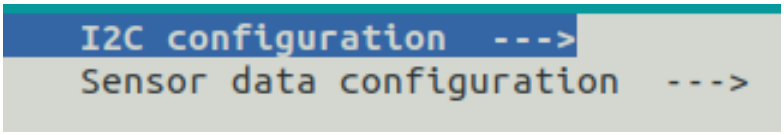

Figura 3.4: *Entradas para el servidor en menuconfig.*

El cliente por su parte en el menuconfig tiene dos entradas pero enfocadas en MQTT y Wifi como vemos en la imagen 3.7. Dentro de la configuración para el Wifi tenemos tres

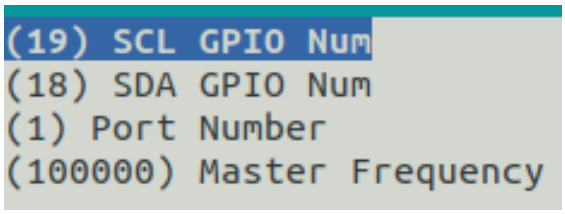

Figura 3.5: *Parámetros configurables para i2c.*

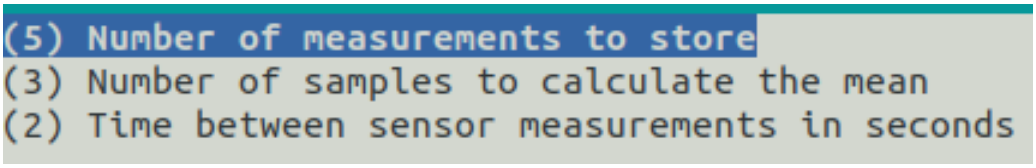

Figura 3.6: *Parámetros configurables para el sensor.*

parámetros básicos que son: el SSID, la contraseña de la red Wifi y cuantos reintentos se hacen antes de dejar de intentar conectarse 3.8. Por la parte de MQTT, tenemos en dominio o ip donde reside el broker 3.9.

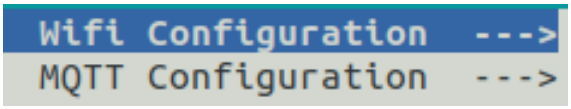

Figura 3.7: *Entradas para el cliente en menuconfig.*

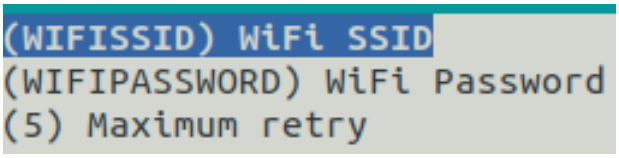

Figura 3.8: *Parámetros configurables Wifi en el cliente.*

Figura 3.9: *Parámetros configurables MQTT en el cliente.*

```
static uint16_t ble_mesh_get_sensor_data(esp_ble_mesh_sensor_state_t *state, uint8_t *data){
    uint8_t mpid_len = 0, data_len = 0;
    uint32_t mpid = 0;

    if (state == NULL || data == NULL) {
        ESP_LOGE(TAG, "%s, Invalid parameter", __func__);
        return 0;
    }

    uint8_t sensor_data = 0;
    if(state->sensor_property_id == SENSOR_PROPERTY_TEMP)
    {
        sensor_data = si7021_get_mean_temperature_data();
        ESP_LOGW(TAG, "Temperature: %d ªC", sensor_data);
    }
    else
    {
        sensor_data = si7021_get_mean_humidity_data();
        ESP_LOGW(TAG, "Humidity: %d %%", sensor_data);
    }

    // store sensor data into net_buffer
    net_buf_simple_pull_u8(state->sensor_data.raw_value);
    net_buf_simple_push_u8(state->sensor_data.raw_value, sensor_data);

    if (state->sensor_data.length == ESP_BLE_MESH_SENSOR_DATA_ZERO_LEN) {
        /* For zero-length sensor data, the length is 0x7F, and the format is Format B. */
        mpid = ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID(state->sensor_data.length,
            state->sensor_property_id);
        mpid_len = ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID_LEN;
        data_len = 0;
    } else {
        if (state->sensor_data.format == ESP_BLE_MESH_SENSOR_DATA_FORMAT_A) {
            mpid = ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID(state->sensor_data.length,
                state->sensor_property_id);
            mpid_len = ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID_LEN;
        } else {
            mpid = ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID(state->sensor_data.length,
                state->sensor_property_id);
            mpid_len = ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID_LEN;
        }
        /* Use "state->sensor_data.length + 1" because the length of sensor data is zero-based. */
        data_len = state->sensor_data.length + 1;
    }

    memcpy(data, &mpid, mpid_len);
    memcpy(data + mpid_len, state->sensor_data.raw_value->data, data_len);

    return (mpid_len + data_len);
}
```

Listing 7: Función que obtiene una medición del sensor.

```c
void app_main(void)
{
    /* NVS */
    ESP_LOGI(TAG, "Initialising NVS");

    esp_err_t err = nvs_flash_init();
    if (err == ESP_ERR_NVS_NO_FREE_PAGES) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK(err);

    /* Wifi */
    xSem_connected = xSemaphoreCreateMutex();
    wifi_init_sta();
    ESP_LOGI(TAG, "Waiting for wifi connection...");
    while(!get_connected());

    // BLE
    ESP_LOGI(TAG, "Initialising Ble and Bluetooth with sensor model -> client");
    ESP_ERROR_CHECK(ble_mesh_init());

    // MQTT
    ESP_ERROR_CHECK(init_mqtt());
    /*****************************/

    vTaskDelete(NULL);
}
```

Listing 8: Main del cliente BLE Mesh

```
esp_err_t init_mqtt()
{
    esp_mqtt_client_config_t mqtt_cfg = {
        .uri = CONFIG_BROKER_URL,
    };

    queue_receive  = xQueueCreate(4, sizeof(mqtt_json));
    QueueHandle_t queue_messages = xQueueCreate(25, sizeof(message_t *));

    // Initialize queue message parser
    initialize_messages_parser_queue(queue_messages);

    // ble cmd task
    xTaskCreate(&task_parse_json, "task_parse_json", 4098, (void *) &queue_receive, 4, NULL);

    // Task to send responses to dashboard or cli
    xTaskCreate(&task_send_response_mqtt, "task_send_response_mqtt", 4098,
        (void *) &queue_messages, 4, NULL);

    // Initialize task manager
    init_tasks_manager();

    client_mqtt = esp_mqtt_client_init(&mqtt_cfg);
    mqtt_app_start(client_mqtt);

    return ESP_OK;
}
```

Listing 9: Inicialización MQTT.

```
// static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event)

case MQTT_EVENT_DATA:
    ESP_LOGI(TAG, "MQTT_EVENT_DATA");

    // Get topic
    char *topic = malloc(event->topic_len * sizeof(char) + 1);
    memset(topic, '\0', event->topic_len * sizeof(char) + 1);
    strncpy(topic, event->topic, event->topic_len);

    // Get data
    char *data = malloc(event->data_len * sizeof(char) + 1);
    memset(data, '\0', event->data_len * sizeof(char) + 1);
    strncpy(data, event->data, event->data_len);

    ESP_LOGI(TAG, "Topic: %s, data: %s", topic, data);

    // Receive a request to create a task
    if(strcmp(topic, SUB_TOPIC_BLE) == 0)
    {
        mqtt_json json = {
            .json = event->data,
            .size = event->data_len,
        };
        xQueueSendToBack(queue_receive, &json, 0);
    }
    else if(strcmp(topic, SUB_TOPIC_CMD) == 0)
    {
        cJSON *root = cJSON_Parse(data);
        const cJSON *cmd = cJSON_GetObjectItem(root, "cmd");

        if(cmd != NULL)
        {
            if(strcmp(cmd->valuestring, "tasks") == 0)
            {
                queue_list_task();
            }
        }
    }

    free(topic);
    free(data);
```

Listing 10: Procesamiento de json en módulo MQTT.

```c
static void task_send_response_mqtt(void* params)
{
    QueueHandle_t queue = (*(QueueHandle_t *) params);
    BaseType_t xStatus;
    message_t *message = NULL; // all data is copied to queue area
    char* json = NULL;

    for(;;)
    {
        xStatus = xQueueReceive(queue, &(message), portMAX_DELAY);
        if(xStatus == pdTRUE)
        {
            ESP_LOGI(TAG, "Message to mqtt of type %d", message->type);
            json = message_to_json(message);
            if(json != NULL)
            {
                if(message->type == GET_STATUS)
                {
                    // send to dashboard
                    esp_mqtt_client_publish(client_mqtt, PUB_TOPIC_DASH, json, 0, 0, 0);
                }
                else{
                    // send to cli
                    esp_mqtt_client_publish(client_mqtt, PUB_TOPIC_CLI, json, 0, 0, 0);
                }
                free(json);
            }
            else
            {
                ESP_LOGE(TAG, "Json is null!");
            }
            free_message(message);
        }
        vTaskDelay(2000 / portTICK_PERIOD_MS);
    }
}
```

Listing 11: Tarea que procesa mensajes para enviar un JSON.

```
// void task_parse_json(void *params)

for(;;)
{
    xStatus = xQueueReceive(queue, &json_received, portMAX_DELAY);
    if(xStatus == pdTRUE)
    {
        message_t* messages = create_message(PLAIN_TEXT);

        json_string = (char *) malloc(json_received.size * sizeof(char) + 1);
        memset(json_string, '\0',  json_received.size * sizeof(char) + 1);
        strncpy(json_string, json_received.json, json_received.size);
        ESP_LOGI(TAG, "Json received %s", json_string);

        // Check if json is correct
        size_actions = 0;
        actions = parse_build_task(messages, json_string, &size_actions); free(json_string);
        if(size_actions != 0 && actions != NULL)
        {
            for(int i = 0; i < size_actions; i++)
            {
                if(actions[i].opmode == REMOVE) // remove task
                {
                    delete_task(&actions[i].task, messages);
                }
                else if(actions[i].opmode == CREATE)
                {
                    create_task(&actions[i].task, messages);
                }
                else
                {
                    ESP_LOGE(TAG, "opmode couldn't be recognize!");
                }
            }
        }
        else
        {
            ESP_LOGE(TAG, "Json could be processed or size task equals zero!");
            add_message_text_plain(messages, false, "Json could be processed or
                size task equals zero!");
        }
        send_message_queue(messages);
        free(actions);
    }
    else
    {
        ESP_LOGE(TAG, "Error in queue -> task_receive_json");
    }
    vTaskDelay(2000 / portTICK_PERIOD_MS);
}
vTaskDelete(NULL);
```

Listing 12: Tarea que procesa JSON para manejar tareas.

```c
static action_t* parse_build_task(message_t* messages, char *json, int* size)
{
    action_t *acts = NULL;

    cJSON *root = cJSON_Parse(json);
    const cJSON *actions = cJSON_GetObjectItem(root, "actions");

    if(actions != NULL)
    {
        const cJSON *action = NULL;
        int size_actions = cJSON_GetArraySize(actions);

        if(size_actions != 0)
        {
            int index = 0;
            acts = (action_t *) malloc(size_actions * sizeof(action_t));
            memset(acts, 0, size_actions * sizeof(action_t));

            cJSON_ArrayForEach(action, actions)
            {
                if(build_task(&acts[index], action, messages))
                    index++;
            }
            *size = index;
        }
        else{
            ESP_LOGE(TAG, "Action size is zero");
            add_message_text_plain(messages, true, "Action size is zero");
        }
    }
    else
    {
        ESP_LOGE(TAG, "Action list required");
        add_message_text_plain(messages, true, "Action list required");
    }
    return acts;
}
```

Listing 13: Función que devuelve una lista de acciones.

```c
static bool build_task(action_t *ble_task, const cJSON* action, message_t *messages)
{
    bool build = true;

    const cJSON *auto_task      = cJSON_GetObjectItem(action, "auto");
    const cJSON *opcode         = cJSON_GetObjectItem(action, "opcode");
    const cJSON *delay          = cJSON_GetObjectItem(action, "delay");
    const cJSON *name           = cJSON_GetObjectItem(action, "name");
    const cJSON *addr           = cJSON_GetObjectItem(action, "addr");
    const cJSON *sensor_prop_id = cJSON_GetObjectItem(action, "sensor_prop_id");

    // Task to delete
    if(opcode == NULL && delay == NULL
        && auto_task == NULL && addr == NULL
        && name != NULL)
    {
        ble_task->opmode = REMOVE;
        ble_task->task.name = name->valuestring;
    }
    // Task to create -> one time
    else if(opcode != NULL && addr != NULL)
    {
        ble_task->opmode      = CREATE;
        ble_task->task.opcode = get_opcode(opcode->valuestring);
        ble_task->task.addr   = string_to_hex_uint16_t(addr->valuestring);

        if(sensor_prop_id != NULL)
        {
            ble_task->task.sensor_prop_id = string_to_hex_uint16_t(sensor_prop_id->valuestring);
        }
        else
        {
            ble_task->task.sensor_prop_id = 0x0000;
        }

        if(auto_task != NULL) // task to create periodically
        {
            if(cJSON_IsTrue(auto_task) && is_auto_required(ble_task->task.opcode))
            {
                ble_task->task.auto_task = true;
                ble_task->task.name   = sanitize_string(name->valuestring);
                ble_task->task.delay  = delay->valueint;
            }
            else
            {
                ble_task->task.auto_task = false;
            }
        }
    }
    else
    {
        ESP_LOGE(TAG, "Error processing a task!");
        add_message_text_plain(messages, true, "Error processing a task!");
        build = false;
    }
    return build;
}
```

Listing 14: Función que traduce una acción en un elemento action_t.

```c
static void delete_task(ble_task_t *ble_task, message_t *messages)
{
    ESP_LOGI(TAG, "Deleting task %s", ble_task->name);

    task_t found = {
        .name = ble_task->name,
    };
    task_t *task = obtain_task(&found);

    if(task != NULL)
    {
        if(task->task_handler != NULL)
        {
            vTaskDelete(task->task_handler);
            remove_task(task);

            ESP_LOGI(TAG, "Task %s removed", ble_task->name);
            add_message_text_plain(messages, false, "Task %s deleted", ble_task->name);
        }
    }
    else
    {
        ESP_LOGE(TAG, "Task %s doesn't exists. Remove failed", ble_task->name);
        add_message_text_plain(messages, true, "Task %s
            doesn't exists. Remove failed", ble_task->name);
    }
}
```

Listing 15: Función que elimina una tarea

```c
static void create_task(ble_task_t *ble_task, message_t *messages)
{
    // If it is auto, it will be register into task_manager
    if(ble_task->auto_task)
    {
        // Check if the tasks exists
        TaskHandle_t TaskHandle = NULL;
        task_t *new_task = (task_t *)malloc(sizeof(task_t));
        new_task->name = ble_task->name;

        status_t status = add_new_task_if_not_exists(new_task);
        if(status == CREATED)
        {
            ble_task_t aux;
            memcpy(&aux, ble_task, sizeof(ble_task_t));
            xTaskCreate(&task_ble_cmd, aux.name, 2048, (void *) &aux, 5, &TaskHandle);
            new_task->task_handler = TaskHandle;

            ESP_LOGI(TAG, "Task %s created", ble_task->name);
            add_message_text_plain(messages, false, "Task %s created", ble_task->name);
        }
        else
        {
            free(new_task);
            ESP_LOGE(TAG, "Task - %s - exists!", ble_task->name);
            add_message_text_plain(messages, true, "Task %s exists", ble_task->name);
        }
    }
    // If it is not auto task, just create it.
    else
    {
        ble_task_t aux;
        memcpy(&aux, ble_task, sizeof(ble_task_t));

        // Create a name for one time task
        char buff[80];
        sprintf(buff, "Task 0x%04x, addr 0x%04x", aux.opcode, aux.addr);

        xTaskCreate(&task_ble_cmd, buff, 2048, (void *) &aux, 5, NULL);
        add_message_text_plain(messages, false, "One-time Task with
            opcode 0x%04x, addr 0x%04x launched", ble_task->opcode, ble_task->addr);
    }
}
```

Listing 16: Función que crea una tarea.

```c
static void task_ble_cmd(void *params)
{
    ble_task_t ble_task = (*(ble_task_t *) params);

    if(ble_task.auto_task)
    {
        ESP_LOGI(TAG, "[%s] auto = %d, opcode = 0x%04X, delay = %d, addr = 0x%04X,
            sensor_prop_id = 0x%04X", ble_task.name, (int)ble_task.auto_task, ble_task.opcode,
            ble_task.delay, ble_task.addr, ble_task.sensor_prop_id);

        for(;;)
        {
            ble_mesh_send_sensor_message(ble_task.opcode, ble_task.addr, ble_task.sensor_prop_id);
            vTaskDelay(ble_task.delay * 1000 / portTICK_PERIOD_MS);
        }
    }
    else
    {
        ESP_LOGI(TAG, "[One-time task] opcode = 0x%04X, addr = 0x%04X, sensor_prop_id = 0x%04X",
            ble_task.opcode, ble_task.addr, ble_task.sensor_prop_id);
        ble_mesh_send_sensor_message(ble_task.opcode, ble_task.addr, ble_task.sensor_prop_id);
    }

    vTaskDelete(NULL);
}
```

Listing 17: Tarea que manda mensajes BLE Mesh.

```c
typedef struct task_t {
    char *name;
    TaskHandle_t task_handler;
} task_t;

typedef struct node_t {
    task_t *task;
    struct node_t *next;
    struct node_t *prev;
} node_t;

typedef struct tasks_t {
    node_t *first;
    node_t *last;
    unsigned int num_tasks;
} tasks_t;
```

Listing 18: Estructuras del *task manager*.

```
void queue_list_task()
{
    message_t* tasks_info = create_message(TASKS);

    if(task_manager->num_tasks > 0)
    {
        for(node_t *temp = task_manager->first; temp != NULL; temp = temp->next)
        {
            add_message_text_plain(tasks_info, false, "Task: Name -> %s", temp->task->name);
        }
    }
    else
    {
        add_message_text_plain(tasks_info, false, "There are not tasks running...");
    }
    send_message_queue(tasks_info);
}
```

Listing 19: Función que proporciona la lista de tareas.

```
void ble_mesh_send_sensor_message(uint32_t opcode, uint16_t addr, uint16_t sensor_prop_id)
{
    ESP_LOGI(TAG, "ble_mesh_send_sensor_message: 0x%04x. Addr = 0x%04x", opcode, addr);

    esp_ble_mesh_sensor_client_get_state_t get = {0};
    esp_ble_mesh_client_common_param_t common = {0};
    esp_err_t err = ESP_OK;

    ble_mesh_set_msg_common(&common, sensor_client.model, opcode, addr);
    switch (opcode) {
    case ESP_BLE_MESH_MODEL_OP_SENSOR_GET:
        if(sensor_prop_id != 0x0000)
        {
            // indicate that we send optional parameters. In this case is sensor_prop_id
            get.sensor_get.property_id = sensor_prop_id;
            get.sensor_get.op_en = true;
        }
        else
        {
            get.sensor_get.op_en = false;
        }
        break;
    case ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET:
        if(sensor_prop_id != 0x0000)
        {
            get.descriptor_get.property_id = sensor_prop_id;
            get.descriptor_get.op_en = true;
        }
        else
        {
            get.descriptor_get.op_en = false;
        }
        break;
    default:
        break;
    }

    err = esp_ble_mesh_sensor_client_get_state(&common, &get);
    if (err != ESP_OK) {
        ESP_LOGE(TAG, "Failed to send sensor message 0x%04x", opcode);
    }
}
```

Listing 20: Relleno de campos para mandar mensaje BLE Mesh.

```
// static void ble_mesh_sensor_client_cb(esp_ble_mesh_sensor_client_cb_event_t event,
//                                       esp_ble_mesh_sensor_client_cb_param_t *param)

case ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET:
    ESP_LOGI(TAG, "Sensor Descriptor Status, opcode 0x%04x", param->params->ctx.recv_op);
    if (param->status_cb.descriptor_status.descriptor->len !=
        ESP_BLE_MESH_SENSOR_SETTING_PROPERTY_ID_LEN &&
        param->status_cb.descriptor_status.descriptor->len % ESP_BLE_MESH_SENSOR_DESCRIPTOR_LEN) {
        ESP_LOGE(TAG, "Invalid Sensor Descriptor Status length %d",
            param->status_cb.descriptor_status.descriptor->len);
        return;
    }

    if (param->status_cb.descriptor_status.descriptor->len)
    {
        if(param->status_cb.descriptor_status.descriptor->len % 8 == 0)
        {
            ESP_LOG_BUFFER_HEX("Sensor Descriptor",
                param->status_cb.descriptor_status.descriptor->data,
            param->status_cb.descriptor_status.descriptor->len);

            messages = create_message(GET_DESCRIPTOR);
            add_hex_buffer(messages,
                param->status_cb.descriptor_status.descriptor->data,
                param->status_cb.descriptor_status.descriptor->len
            );
            send_message_queue(messages);
        }
        else
        {
            messages = create_message(PLAIN_TEXT);
            add_message_text_plain(messages, true, "Incorrect sensor prop id for device 0x%04x",
                param->params->ctx.addr);
            send_message_queue(messages);
        }

    }
    break;
```

Listing 21: Evento para respuestas de mensajes GET_DESCRIPTOR.

```
static void publish_measure(esp_ble_mesh_sensor_client_cb_param_t *param)
{
    ESP_LOGI(TAG, "Sensor Status, opcode 0x%04x", param->params->ctx.recv_op);

    if (param->status_cb.sensor_status.marshalled_sensor_data->len)
    {
        ESP_LOG_BUFFER_HEX("Sensor Data", param->status_cb.sensor_status.marshalled_sensor_data->data,
            param->status_cb.sensor_status.marshalled_sensor_data->len);
        uint8_t *data = param->status_cb.sensor_status.marshalled_sensor_data->data;
        uint16_t length = 0;

        if(data[0] == 0xFF) // prop id doesnt exists. Error.
        {
            uint8_t fmt      = ESP_BLE_MESH_GET_SENSOR_DATA_FORMAT(data);
            uint16_t prop_id = ESP_BLE_MESH_GET_SENSOR_DATA_PROPERTY_ID(data, fmt);

            message_t* message = create_message(PLAIN_TEXT);
            add_message_text_plain(message, true, "Sensor prop id 0x%04x doesnt exists
                in sensor addr 0x%04x", prop_id, param->params->ctx.addr);
            send_message_queue(message);

        }
        else
        {
            for (; length < param->status_cb.sensor_status.marshalled_sensor_data->len; )
            {
                uint8_t fmt      = ESP_BLE_MESH_GET_SENSOR_DATA_FORMAT(data);
                uint8_t data_len = ESP_BLE_MESH_GET_SENSOR_DATA_LENGTH(data, fmt);
                uint16_t prop_id = ESP_BLE_MESH_GET_SENSOR_DATA_PROPERTY_ID(data, fmt);
                uint8_t mpid_len = (fmt == ESP_BLE_MESH_SENSOR_DATA_FORMAT_A ?
                        ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID_LEN
                            : ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID_LEN);

                ESP_LOGI(TAG, "Format %s, length 0x%02x, Sensor Property ID 0x%04x",
                    fmt == ESP_BLE_MESH_SENSOR_DATA_FORMAT_A ? "A" : "B", data_len, prop_id);

                if (data_len != ESP_BLE_MESH_SENSOR_DATA_ZERO_LEN)
                {
                    ESP_LOG_BUFFER_HEX("Sensor Data", data + mpid_len, data_len + 1);

                    int measure = *(data + mpid_len);
                    ESP_LOGW(TAG, "Measure %d", measure);

                    message_t* message = create_message(GET_STATUS);
                    add_measure_to_message(message, param->params->ctx.addr, prop_id, measure);
                    send_message_queue(message);

                    length += mpid_len + data_len + 1;
                    data += mpid_len + data_len + 1;
                }
                else
                {
                    length += mpid_len;
                    data += mpid_len;
                }
            }
        }
    }
}
```

```
//static void ble_mesh_sensor_client_cb(esp_ble_mesh_sensor_client_cb_event_t event,
//                                      esp_ble_mesh_sensor_client_cb_param_t *param)

case ESP_BLE_MESH_SENSOR_CLIENT_PUBLISH_EVT:
    ESP_LOGI(TAG, "Receive message from group. opcode 0x%04x", param->params->opcode);
    switch(param->params->opcode)
    {
        case ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS:
            publish_measure(param);
            break;
        case ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET:
            if(param->status_cb.descriptor_status.descriptor->len == 8)
            {
                ESP_LOG_BUFFER_HEX("Sensor Descriptor",
                    param->status_cb.descriptor_status.descriptor->data,
                param->status_cb.descriptor_status.descriptor->len);

                messages = create_message(GET_DESCRIPTOR);
                add_hex_buffer(messages,
                        param->status_cb.descriptor_status.descriptor->data,
                        param->status_cb.descriptor_status.descriptor->len
                );
                send_message_queue(messages);
            }
            else
            {
                messages = create_message(PLAIN_TEXT);
                add_message_text_plain(messages, true,
                    "Incorrect sensor prop id for device 0x%04x", param->params->ctx.addr);
                send_message_queue(messages);
            }
            break;
    }
    break;
```

Listing 23: Evento cuando el mensaje fue enviado a un grupo.

```c
typedef enum {
    PLAIN_TEXT, // simple message with info, errors
    TASKS, // tasks list
    GET_STATUS,
    GET_DESCRIPTOR,
    HEX_BUFFER
} message_type_t;

/*********** Types of messages *****************/
// plain text: errors, info messages, tasks list
typedef struct text_t {
    bool error_message;
    int num_messages;
    char messages[MAX_NUM_MESSAGES][MAX_LENGHT_MESSAGE];
} text_t;

typedef struct measure_t {
    uint16_t sensor_prop_id;
    uint16_t addr;
    int value;
} measure_t;

typedef struct hex_buffer_t {
    uint8_t* data;
    uint16_t len;
} hex_buffer_t;

/*******************************************/

/* A message will only have one of the following fields */
typedef union message_content_t {
    text_t text_plain;
    measure_t measure;
    hex_buffer_t hex_buffer;
} message_content_t;

/* General structure for every message */
typedef struct message_t {
    message_type_t type;
    message_content_t m_content;
} message_t;
```

Listing 24: Estructuras para el módulo messages_parser.

```
message_t* create_message(message_type_t type)
{
    message_t* message = (message_t *) malloc(sizeof(message_t));

    if(type == PLAIN_TEXT || type == TASKS)
    {
        ESP_LOGI(TAG, "Creating PLAIN_TEXT, TASKS");
        message->m_content.text_plain.num_messages = 0;
        message->m_content.text_plain.error_message = false;
    }
    else if(type == GET_STATUS)
    {
        ESP_LOGI(TAG, "Creating GET_STATUS");
        message->m_content.measure.value = 0;
        message->m_content.measure.addr = 0x0000;
    }
    else if(type == HEX_BUFFER || type == GET_DESCRIPTOR)
    {
        ESP_LOGI(TAG, "Creating HEX_BUFFER, GET_DESCRIPTOR");
        message->m_content.hex_buffer.data = NULL;
        message->m_content.hex_buffer.len = 0;
    }

    message->type = type;
    return message;
}
```

Listing 25: Función para crear un mensaje.

```c
/**
 * @brief Helper function to add a new message
 * @param m: messate_t * struct
 * @param error_message: if we add an error message
 * @param message: string with format
 * @param args: arguments to include in message
 */
void add_message_text_plain(message_t* m, bool error_message, const char* message, ...);

/**
 * @brief Helper function to fill a measure_t struct
 * @param m: message_t struct
 * @param addr: addr to add into measure_t
 * @param sensor_prop_id: sensor prop id which measure blongs to
 * @param measure: measure to add into measure_t
 */
void add_measure_to_message(message_t* m, uint16_t addr, uint16_t sensor_prop_id, int measure);

/**
 * @brief Helper function to fill hex_buffer_t
 * @param m: messate_t * struct
 * @param data: uint8_t* to copy into hex_buffer_t
 * @param len: number of elements in data
 */
void add_hex_buffer(message_t* m, uint8_t* data, uint16_t len);
```

Listing 26: Funciónes para rellenar el contenido de message_t.

```c
char* message_to_json(message_t *message)
{
    if(message->type == PLAIN_TEXT)
        return text_plain_to_json(&message->m_content.text_plain, "messages");

    if(message->type == TASKS)
        return text_plain_to_json(&message->m_content.text_plain, "tasks");

    if(message->type == GET_STATUS)
        return get_status_to_json(&message->m_content.measure);

    if(message->type == GET_DESCRIPTOR)
        return get_descriptor_to_json(&message->m_content.hex_buffer);

    if(message->type == HEX_BUFFER)
        return get_hex_buffer_to_json(&message->m_content.hex_buffer, "hex buffer");

    return NULL;
}
```

Listing 27: Función que forma un JSON en base al tipo de mensaje.

```c
/**
 * @brief Return uint16_t representation of a char*.
 * char* has to be 4-sized
 */
uint16_t string_to_hex_uint16_t(const char *string);

/**
 * @brief Return char[] representation of uint8_t[].
 * len is val[] size.
 */
char* uint8_array_to_string(uint8_t *val, uint16_t len);

/**
 * @brief Return char[] representation of uint16_t
 */
char* uint16_to_string(uint16_t value);
```

Listing 28: Funciones para transformar datos.

Capítulo 4

BLE Mesh en el despliegue

El código usado para el cliente BLE Mesh ha sido desarrollado a partir del ejemplo *sensor client* que podemos encontrar en el repositorio de github de ESP-IDF [13]. Este ejemplo, a parte de funcionar como cliente BLE Mesh, también funciona como provisionador dando las mismas claves a todos los dispositivos. El objetivo es tener la Raspberry pi como provisionador, ya que es un dispositivo portable de bajo consumo que podemos llevar a cualquier parte.

4.1. Configuración de Raspberry Pi

Para hacer funcionar la Raspberry pi como provisionador BLE Mesh, tenemos que realizar una serie de pasos y configuración ya que, por defecto, el kernel no viene preparado para ello. Por consiguiente, vamos a ejecutar los siguientes pasos, que están basados en la guía [14].

Modificación y compilación del Kernel

Como primer paso en la configuración, tenemos que instalar las siguientes dependencias:

```
sudo apt-get install -y git bc libusb-dev libdbus-1-dev libglib2.0-dev libudev-dev libical-dev
libreadline-dev autoconf bison flex libssl-dev
```

Ahora, nos descargamos la versión del kernel necesaria. Para este proyecto, la versión ha sido la 20200212-1. Después, nos situamos dentro de la carpeta descargada:

```
cd ~
wget https://github.com/raspberrypi/linux/archive/raspberrypi-kernel_1.20200212-1.tar.gz
tar -xvf raspberrypi-kernel_1.20200212-1.tar.gz
cd ./linux-raspberrypi-kernel_1.20200212-1/
```

Una vez dentro, tenemos que definir una serie de variables y ejecutar diferentes comandos. En este proyecto, el provisionador ha sido montado en una raspberry pi 3b+, por lo que según la guía hay que ejecutar los siguiente comandos para construir la configuración:

```
KERNEL=kernel7
make bcm2709_defconfig
make menuconfig
```

El siguiente paso es habilitar las opciones del kernel que nos permitan hacer funcionar la raspberry pi como un provisionador. Para ello, una vez hayamos ejecutado *make menuconfig*, debemos navegar hasta la entrada *Cryptographic API* y activar las siguientes opciones:

- CCM support: Método de cifrado (AES).
- CMAC support: Método de cifrado (AES).
- User-space interface for hash algorithms: Habilitar algoritmos de cifrado en el espacio de usuario.
- User-space interface for symmetric key cipher algorithms: Habilitar algoritmos de cifrado basados en clave simétrica en el espacio de usuario.
- User-space interface for AEAD cipher algorithms: Habilitar métodos de cifrado AEAD (Encriptacion autenticada con datos asociados) en el espacio de usuario.

Aunque la Raspberry pi tiene la posibilidad de usar Bluetooth, no se puede usar como provisionador. Primero, porque el kernel no tiene el código necesario para ello. Segundo, debido a la arquitectura de linux, necesitamos habilitar en el espacio de usuario las funcionalidades anteriores ya que usaremos una herramienta por linea de comandos para el

provisionamiento. Por último, y no menos importante, este proceso de provisionamiento se tiene que hacer de manera segura, por ello, necesitamos habilitar las capacidades criptográficas.

Una vez guardados los cambios, tenemos que compilar el kernel. Para ello hay que ejecutar el siguiente comando:

```
make -j4 zImage modules dtbs
```

Este proceso dura en torno a dos horas y media o tres. Una vez que ha sido compilado, procedemos a instalar el kernel y sus módulos. Para ello ejecutamos los siguientes comandos:

```
sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
sudo reboot
```

Instalación de BlueZ

BlueZ[8] es un paquete que provee soporte para las principales capas y protocolos de Bluetooth. Es flexible, eficiente y utiliza una implementación modular. Para poder usar esta implementación, tenemos que instalar previamente json-c con los siguientes comandos:

```
cd ~
wget https://s3.amazonaws.com/json-c_releases/releases/json-c-0.13.tar.gz
tar -xvf json-c-0.13.tar.gz
cd json-c-0.13/
./configure --prefix=/usr --disable-static && make
sudo make install
```

Una vez instalada, vamos a descargarnos el archivo comprimido de BlueZ:

```
cd ~
wget http://www.kernel.org/pub/linux/bluetooth/bluez-5.54.tar.xz
tar -xvf bluez-5.54.tar.xz
cd bluez-5.54/
```

Al descargarnos los binarios, debemos configurarlo para BLE Mesh. Para ello, vamos a ejecutar los siguientes comandos:

```
./configure --enable-mesh --enable-testing --enable-tools --prefix=/usr --mandir=/usr/share/man --
sysconfdir=/etc --localstatedir=/var
sudo make
sudo make install
```

Configuración de Systemd

Para que nuestra Raspberry pi use Bluetooth bajo BlueZ, tenemos que configurar el demonio que systemd ejecuta cuando requerimos el uso de Bluetooth. Primero, vamos a guardarnos una copia de seguridad del servicio de bluetooth ya existente:

```
sudo cp /usr/lib/bluetooth/bluetoothd /usr/lib/bluetooth/bluetoothd-550.orig
```

Ahora, vamos a crear los enlaces simbólicos necesarios para que systemd ejecute BlueZ:

```
sudo ln -sf /usr/libexec/bluetooth/bluetoothd /usr/lib/bluetooth/bluetoothd
sudo systemctl daemon-reload
cd ~/.config/
mkdir meshctl
cp ~/bluez-5.54/tools/mesh-gatt/local_node.json ~/.config/meshctl/
cp ~/bluez-5.54/tools/mesh-gatt/prov_db.json ~/.config/meshctl/
bluetoothd -v
```

Siguiendo este proceso, tendremos dos funcionalidades distintas para provisionar donde podemos elegir entre:

- **meshctl**: provisiona mediante GATT.
- **mesh-cfgclient**: provisiona mediante PB-ADV.

Con esto ya tendríamos la configuración completa de nuestra Raspberry pi para usarla como provisionador BLE Mesh.

4.2. Provisionamiento de dispositivos

Ahora que tenemos la Raspberry pi para provisionar, debemos elegir una de las dos herramientas por linea de comandos que se nos instala en el proceso descrito en el apartado anterior 4.1. En este proyecto, se ha usado *mesh-cfgclient* ya que ha sido la que menos problemas ha ocasionado, debido a que *meshctl* ni siquiera era capaz de encontrar nodos que no estaban provisionados.

Para hacer uso de la herramienta *mesh-cfgclient* en la Raspberry pi, debemos ser el propietario del controlador bluetooth. Para ello, tenemos que hacer un pequeño script con las siguientes lineas y ejecutarlo en otra consola antes de usar la propia herramienta:

```
sudo systemctl stop bluetooth-mesh.service bluetooth.service
sleep 2
sudo ~/bluez-5.54/mesh/bluetooth-meshd
```

Una vez ejecutado el script, abrimos otra consola y ejecutamos la herramienta *mesh-cfgclient*. Lo primero que debemos hacer es crear una red nueva, lo que se traduce en crear un fichero en la ruta *.config/meshcfg/config_db.json*. Para ello, ejecutamos los siguientes comandos:

```
create # creamos el archivo donde tendremos la configuración de nuestra red
appkey-create 0 0 # creamos una clave de red y de aplicación con index cero ambas claves
```

Una vez creada nuestra red, lo que debemos hacer es buscar nodos que no estén provisionados, para ello ejecutamos el comando:

```
discover-unprovisioned on 5
```

para que busque durante 5 segundos solamente. Si todo sale bien, nos aparecerán los nodos como vemos en la imagen 4.1. Vemos que la herramienta nos da la potencia de señal

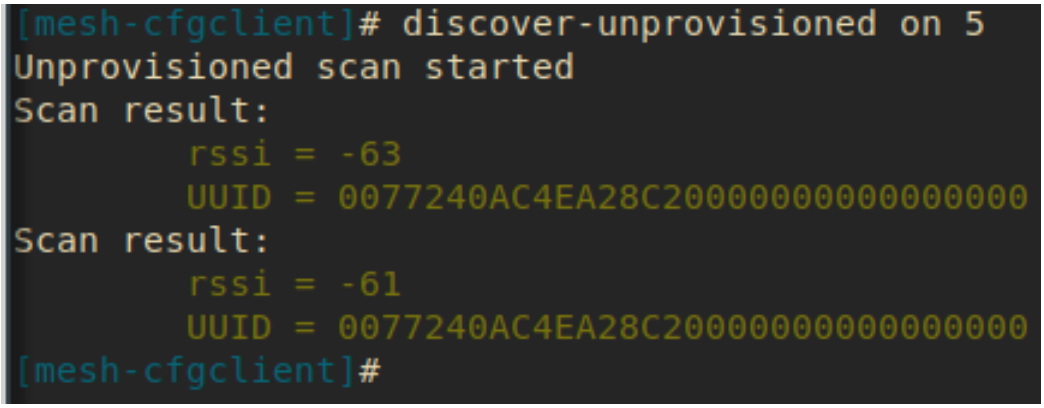

Figura 4.1: *Descubrimiento de nodos no provisionados*

y el UUID del dispositivo. Si queremos saber que dispositivo es el que estamos provisionado, nos viene bien saber que en el UUID se encuentra la MAC del dispositivo.

Para provisionar dicho nodo hay que ejecutar el comando *provision* junto con el UUID:

```
provision 0077240AC4EA28C20000000000000000
```

Este paso dará una dirección al nodo dentro de la red MESH y se realizaran todos los pasos descritos en el proceso de provisionamiento 2.2.4, salvo el último paso 2.2.4, donde la primera dirección que se entrega es la 00aa. Una vez ejecutado el comando anterior, vemos que efectivamente le ha entregado una dirección a nuestro nodo 4.2. También en la herramienta 4.3 podemos ver que se ha provisionado correctamente.

En este punto, nuestro nodo tiene una dirección pero no tiene las claves de red y aplicación necesarias para interactuar con otros nodos de manera segura. Por tanto, el siguiente paso es acceder a un submenu de la herramienta llamado config y poner como objetivo el nodo que queremos configurar, en este caso el nodo con dirección 00aa. Para ello debemos ejecutar el siguiente comando:

```
menu config # accedemos al submenu config
target 00aa # decimos a la herramienta que los comandos siguientes afectaran al nodo 00aa
```

Ahora, vamos a darle las claves de red y aplicación al nodo con los siguientes comandos:

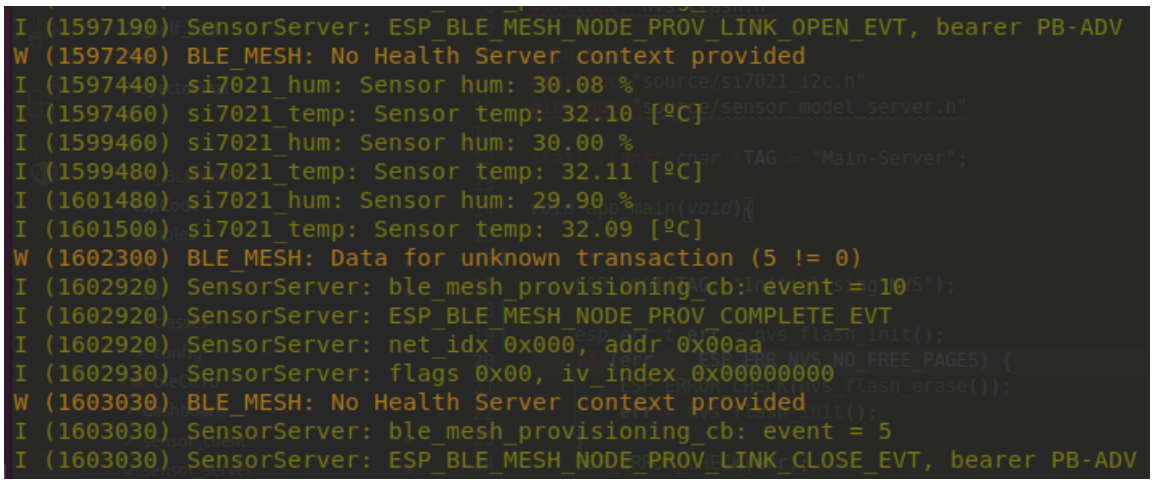

Figura 4.2: *Resultado provisionamiento en el esp32*

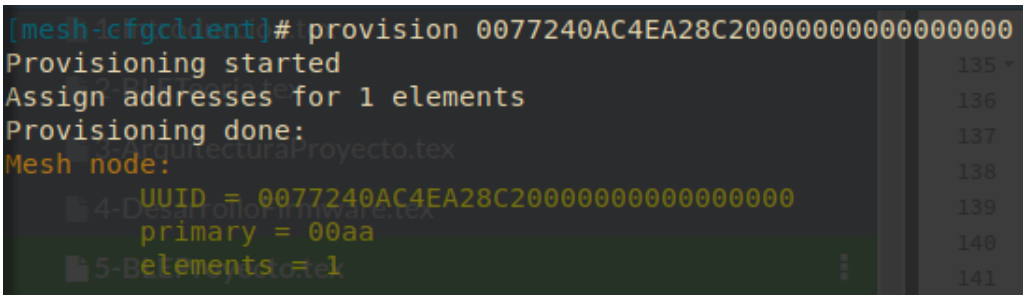

Figura 4.3: *Resultado provisionamiento en la herramienta*

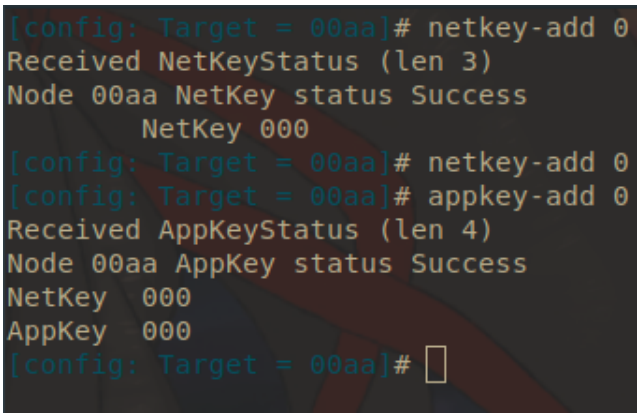

Figura 4.4: *Provisionamiento de claves*

```
netkey-add 0 # le damos la clave de red con index cero
appkey-add 0 # le damos la clave de aplicación con index cero
```

Si todo sale bien, la herramienta nos informara que ha salido con éxito como vemos en la imagen 4.4. En este punto hemos completado el último paso del proceso de provisionamiento 2.2.4.

Por último, debemos decirle al nodo que clave de aplicación debe utilizar con un modelo concreto. Esto significa que si nuestro nodo tiene 2 modelos, le podemos decir que utilice dos claves de aplicación diferentes y así encapsular por tipo de modelo por ejemplo. Para poder configurarlo, debemos ejecutar el siguiente comando:

```
bind 00aa 0 1100 # comando <dirección del nodo> <index de la clave de aplicación> <numero del modelo>
```

El numero del modelo es un valor que identifica a un modelo de manera única. Estos valores vienen dados por Bluetooth SIG que podemos encontrar en su documentación oficial [16]. En nuestro caso que usamos el modelo sensor, tenemos como valor para el modelo sensor servidor el numero 1100 y para el cliente 1102. Como vemos en la imagen 4.5, ha salido correctamente. Ahora mismo, nuestro nodo es capaz de interactuar con el resto de nodos de la red de manera segura.

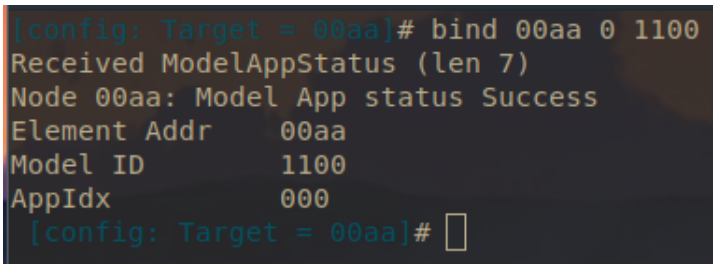

Figura 4.5: *Asociación de clave de aplicación con el modelo*

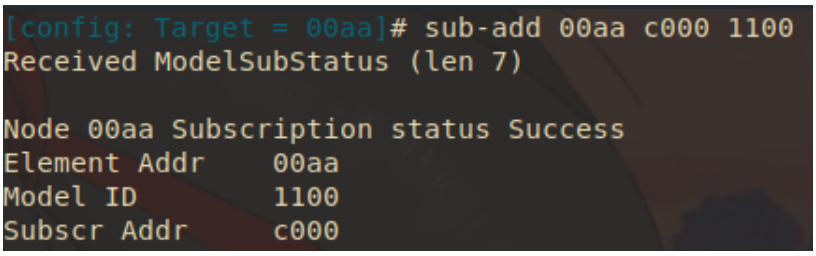

Figura 4.6:

Al tener una red de sensores, nos puede interesar que en vez de preguntar a dos nodos por separado, lo que conllevaría lanzar dos tareas, nos interese preguntar a una única dirección y que nos conteste más de un nodo. A esto se le llama dirección de grupo y en BLE se trata como una suscripción, es decir, el nodo se suscribe a una dirección de grupo por lo que si algun mensaje se pide a esa dirección, el nodo contestara. Para hacer que un nodo se subscriba a una dirección de grupo hay que ejecutar el siguiente comando:

```
sub-add 00ab c000 1100 # dirección del nodo, dirección de grupo, modelo que vamos a suscribir a la dire
```

Un ejemplo de este comando es la imagen 4.6 donde estamos suscribiendo el nodo 00aa a la dirección c000 para el modelo 1100 que corresponde con el modelo sensor servidor.

Este es el proceso de provisionamiento que se ha seguido en este trabajo. Los pasos pueden variar en función de la herramienta pero, por normal general, son los que se han ido esbozando es este apartado.

Capítulo 5

Gestión del despliegue y visualización

5.1. Interfaz de linea de comandos

Con el objetivo de poder interactuar con la red, se ha creado una herramienta por linea de comandos. Con esta CLI, seremos capaces de realizar las tareas que hemos estado viendo: crear o eliminar tareas u obtener que tareas están en ejecución. Para ello, se ha desarrollado dicha herramienta en Ruby ya que es un lenguaje interpretado y nos ofrece la comodidad propia de este tipo de lenguajes. Esta herramienta de comandos tiene los parámetros que vemos en la figura 5.1 donde:

- **-a:** Realizar una acción donde esto se traduce en crear tareas o eliminar tareas.
- **-t:** Consultar las tareas en ejecución en este momento.
- **-e, –editor:** Seleccionar que editor queremos usar para realizar una acción o acciones. Le podemos pasar un editor, por ejemplo *vim*, o no pasar dicho parámetro para que lo obtenga de la variable de entorno EDITOR. Guardara el resultado en el archivo

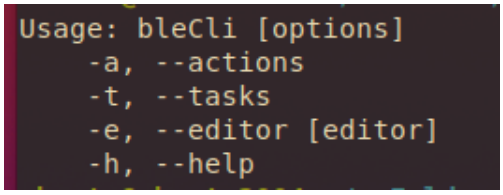

Figura 5.1: *Argumentos de la CLI*

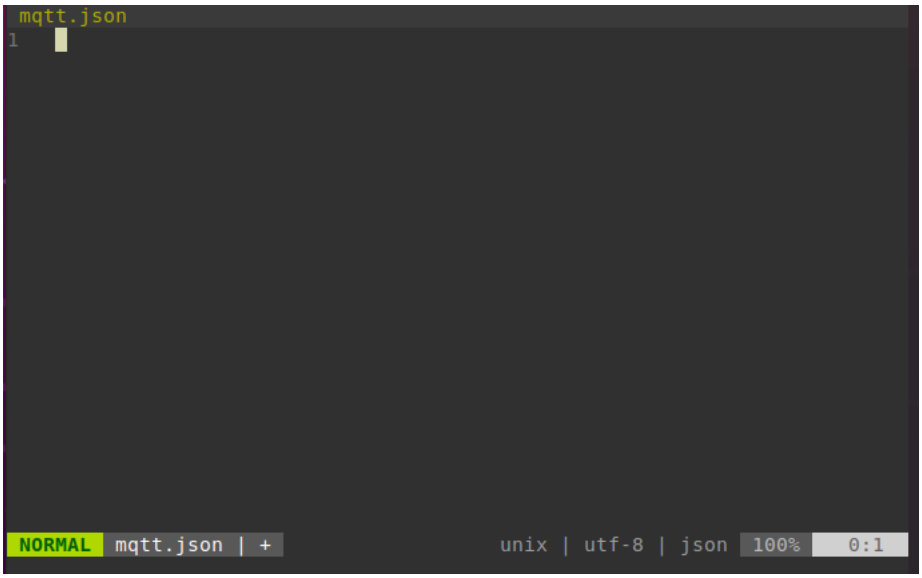

Figura 5.2: *Pantalla de creación de acciones*

/tmp/mqtt.json

- **-h, –help:** Nos mostrara la ayuda.

A la hora de realizar una acción, debemos pasar el editor o tener declarada la variable de entorno EDITOR que comentábamos anteriormente. En este trabajo se ha ejecutado la CLI con la variable de entorno EDITOR=vim para usar dicho editor. Cuando ejecutamos la herramienta con el parámetro -a, nos aparece la pantalla que vemos en la imagen 5.2 donde debemos escribir un json con un formato determinado. También, dependiendo del mensaje que queramos enviar, debemos rellenar ciertos campos.

Los mensajes BLE Mesh que se han usado en este trabajo son GET_DESCRIPTOR y GET_STATUS. El primer mensaje devuelve información que no cambia durante la vida útil del dispositivo por lo que no se permite que sea lanzado de manera periódica, lo que se traduce en que no se va a crear una tarea con un bucle en el cliente. Por otro lado, el mensaje GET_STATUS si se permite que se cree de manera periódica para obtener mediciones y asi poder visualizarlas en el dashboard.

El json que tenemos que crear debe ser el que vemos en la figura 5.3, aunque no todos los

campos son obligatorios, dependiendo de lo que queramos realizar. Los campos permitidos son:

- **auto:** Define si es una tarea automática o no. En el caso de que el opcode no se pueda lanzar automáticamente, el campo sera borrado cuando se mande al cliente por MQTT. Solo se puede poner con opcodes que permitan lanzar tareas automáticas, si no se pone se toma como falso.
- **addr:** Este campo es siempre obligatorio y define la dirección a la que queremos enviar el mensaje.
- **opcode:** Este campo es siempre obligatorio y define el tipo de mensaje BLE Mesh del modelo sensor que queremos mandar. Actualmente se soporta GET_STATUS y GET_DESCRIPTOR.
- **delay:** Numero de segundos que queremos que duerma la tarea. Es obligatorio si el opcode permite lanzar tareas automáticas y queramos que se cree una tarea.
- **name:** Nombre de la tarea. Obligatorio si queremos lanzar una tarea periódica. Si solo mandamos exclusivamente el campo *name*, entonces se eliminara la tarea que tenga el nombre que mandemos.
- **sensor_prop_id:** sensor_prop_id que identifica el estado del servidor al que mandamos el mensaje. Con este campo podemos pedir solo la temperatura o la humedad por ejemplo. Si no se pone, el mensaje va dirigido a todos los estados del servidor.

Estos campos son los que podemos mandar al cliente para que realice las acciones oportunas. En funcion del opcode, se requeriran unos campos u otros y la CLI mostrara error en caso de que falte algun campo o no se adecue a los requisitos que necesita. Por ejemplo, imaginemos que queremos crear una tarea que se ejecute cada 5 segundos y que mande un mensaje GET_STATUS como vemos en la imagen 5.4. En esta figura vemos como se nos

Figura 5.3: *Campos de acciones*

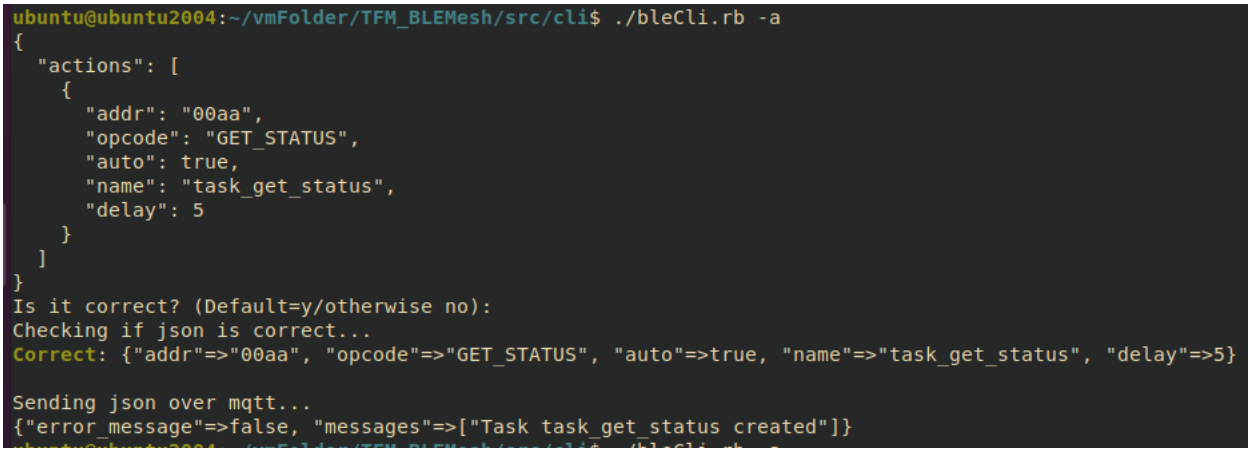

Figura 5.4: *Creación de una tarea automática cada 5 segundos*

muestra el json escrito y nos pide confirmación. Una vez aceptado, se chequea si el json es correcto para los datos proporcionados. Si lo es, nos informa de que es correcto y se espera a que el cliente BLE Mesh nos de feedback. Como vemos, la tarea ha sido creada correctamente. También nos devuelve un flag para saber si los mensajes son de error o no. Si por ejemplo ahora volviéramos a crear la misma tarea, es decir con el mismo nombre, nos informaría de que ya existe como vemos en la figura 5.5.

5.2. Dashboard

El Dashboard se encarga de dibujar los datos de medición o cualquier otra información que consideremos relevante en un panel para su posterior visualización y monitorización. Dentro del campo del IoT, existen muchas opciones de Dashboard, tanto on-premise como

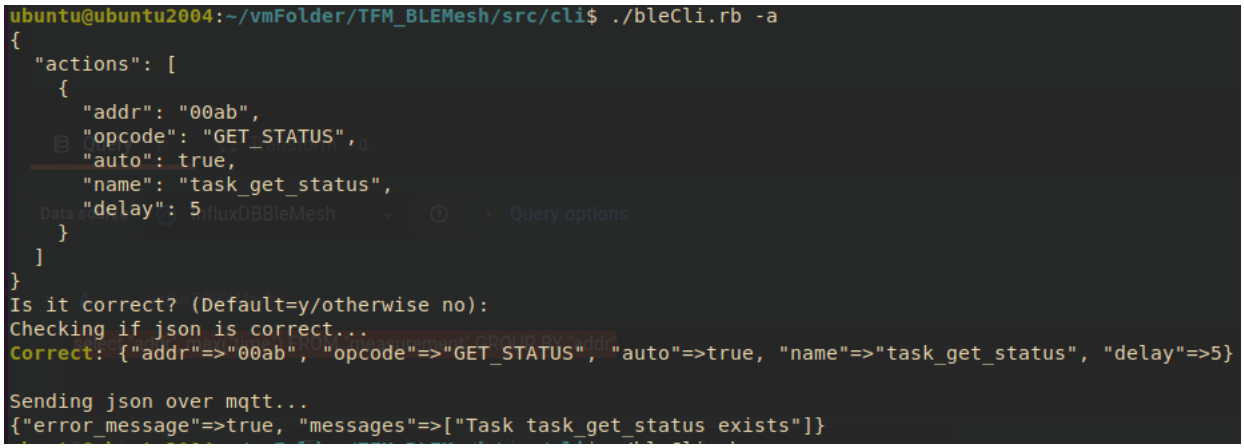

Figura 5.5: *Error al crear una tarea*

en la nube, cada uno de ellos con sus características propias. En este proyecto se ha elegido Grafana [3] ya que tiene múltiples plugins para dibujar gráficas o conectarse a diversas fuentes de datos, entre otros. Por otro lado, necesitamos almacenar las mediciones en alguna base de datos con el objetivo de poder obtenerlas cada vez que queramos. Para ello, se usa Influxdb [6]. Esta base de datos está pensada para almacenar series temporales, lo que se adecua perfectamente a nuestro despliegue donde tendremos multitud de sensores arrojando mediciones en un tiempo determinado. Por último, necesitamos una pieza que se encargue de escuchar por MQTT a la espera de cualquier información para almacenarla en Influxdb. Podríamos resolver esta parte con algun script en Ruby o Python donde aplicar nuestras transformaciones a los datos o introducir una marca temporal. Aunque, para este cometido, existe una herramienta llamada Telegraf [15], que no es más que una solución estándar, bastante potente y con múltiples conectores tanto para emitir datos como para ingerirlos. Si reunimos todos estos componentes, tenemos un esquema para el Dashboard como el mostrado en la figura 5.6. En este esquema, el cliente publica por MQTT mediciones en un topic determinado. Esta información la recoge el broker MQTT y lo reenvía a Telegraf, ya que está suscrito al mismo topic al cual el cliente emite las mediciones. Después Telegraf, recoge la información y la traduce para insertarlo en Influxdb.

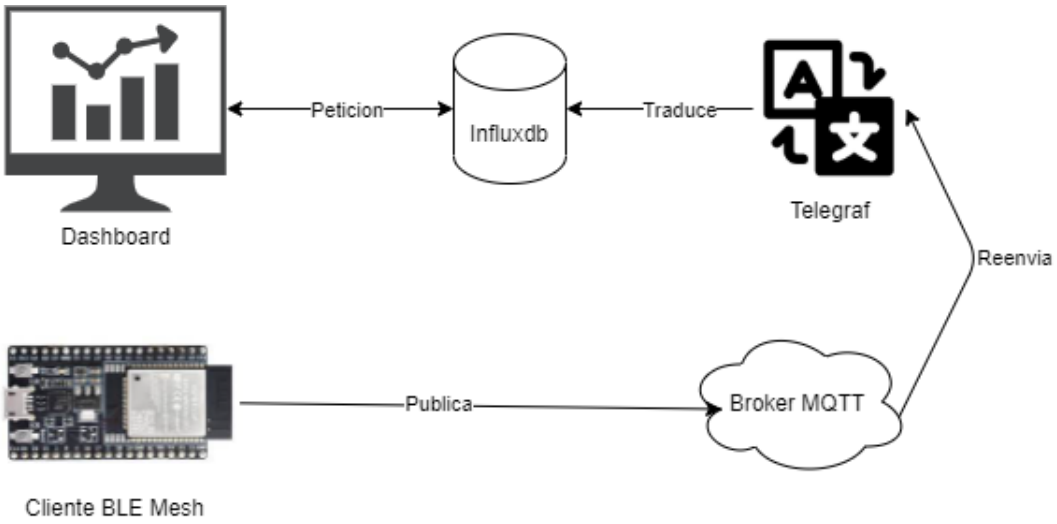

Figura 5.6: *Arquitectura del Dashboard*

```
13 [http]
14   # Determines whether HTTP endpoint is enabled.
15   enabled = true
16
17   # The bind address used by the HTTP service.
18   bind-address = "127.0.0.1:8086"
```

Figura 5.7: *Configuración Influxdb*

Influxdb

Influxdb tiene un cliente desde el que podemos hacer queries, crear bases de datos, borrar registros etc. Esto es muy útil para un administrador, pero para poder interactuar con nuestra base de datos desde otro software, debemos usar algun tipo de protocolo estándar desde el que se conecten programas externos. Por suerte Influxdb permite habilitar un endpoint HTTP. Para ello, hay que modificar el archivo de configuración */etc/influxdb/influxdb.conf* como vemos en la imagen 5.7. Con esta configuración habilitaremos un endpoint HTTP y permitiremos a Telegraf tener una comunicación completa con nuestra base de datos.

Una vez configurado Influxdb, vamos a crear las tablas y bases de datos necesarias para el despliegue. Una cosa a tener en cuenta cuando se usa Influxdb es que no tenemos que crear tablas como hariamos con SQL. Esta base de datos está enfocada a series temporales por lo que para usarla lo único que tenemos que hacer es, en cada inserción, dar un nombre

de serie temporal, sin crearla previamente, donde se insertara el valor. Además, cada registro insertado puede tener campos adicionales llamados tags para identificar cada valor de manera más clara frente a otros y así filtrarlos. También, al ser una base de datos enfocada en series temporales, se encargara de que en cada inserción se ponga una marca de tiempo sin tener nosotros que configurar nada. En el caso de nuestro despliegue crearemos una base de datos llamada *blemesh* donde insertaremos los datos de temperatura y humedad. Para crearla, lo único que tenemos que hacer es usar el siguiente comando:

```
create database blemesh
```

Telegraf

En la arquitectura 5.6 de nuestro *dashboard*, Telegraf se encarga de escuchar vía MQTT para después insertar la información recibida en Influxdb. Para configurarlo de esta manera, tenemos que modificar el archivo de configuración `/etc/telefrag/telegraf.conf` como vemos en la figura 5.8 donde todo lo definido como outputs sera hacia donde enviamos los datos, y lo definido como inputs serán fuentes de datos desde las que procesaremos la información.

Por la parte de Influxdb, debemos aportar el *endpoint*, el nombre de la base de datos y si deseamos comprobar si la base de datos ha sido creada previamente o no cuando insertemos. En nuestro caso, como la hemos creado anteriormente, no haría falta hacer ninguna comprobación. Por otra parte, para MQTT, es necesario definir:

- El servidor o servidores del o los *broker* MQTT.
- Una lista de *topics*. En nuestro caso sera uno definido como */sensors/results/dashboard*
- Al poder dar una lista de *topics*, podría ser interesante que, en cada registro que creemos en Influxdb, también insertemos desde que *topic* hemos obtenido el valor. En nuestro caso es indiferente, por lo que para que no nos inserte el *topic*, debemos dejar el parámetro *topic_tag* vacío.

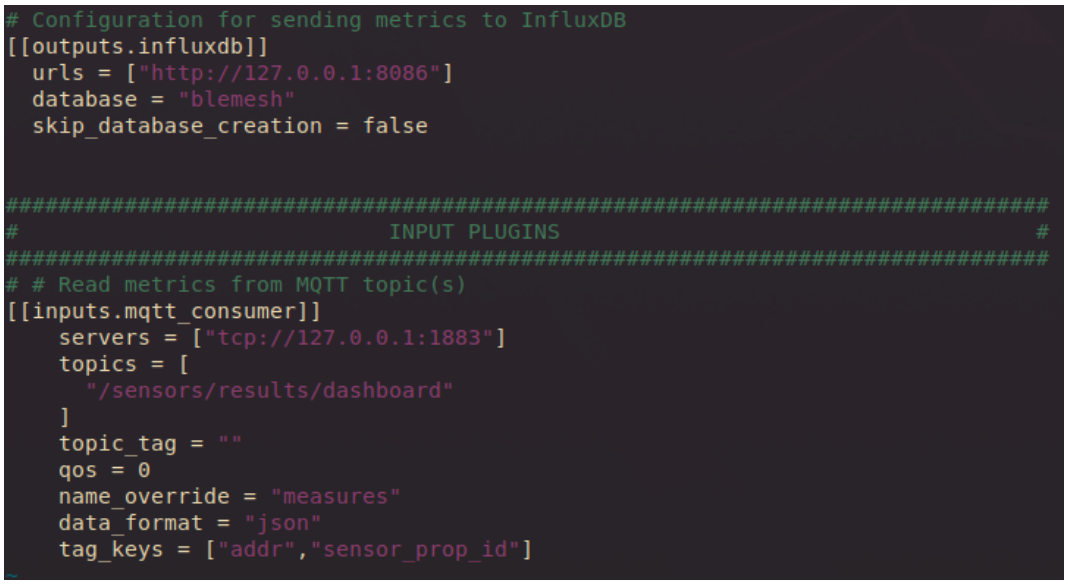

Figura 5.8: *Configuración de Telegraf.*

- En este caso, para el proyecto hemos dejado como QoS el valor cero. Esto debe ser ajustado en función de nuestros requisitos.

- El parámetro *name_override* es el nombre de la serie temporal donde almacenaremos los valores. En nuestro caso recuperamos dos tipos de mediciones por lo que la llamaremos *measures*.

- El parámetro *data_format* nos dice que tipo de formato vamos a recibir. En nuestro caso sera un json.

- El parámetro *tag_keys* contiene que datos del json que recibamos serán insertados como tags. En nuestro caso tendremos la dirección del nodo servidor que envía la medición y el sensor_prop_id al que se le ha pedido la medición. Esto luego nos permitirá crear dashboards para una medición, ya sea temperatura o humedad, y para poder filtrar por direcciones.

Como vemos, Grafana va a estar obteniendo datos de Influxdb cada cierto tiempo para luego pintarlo en los paneles que vayamos creando. Según esto, el dashboard propuesto se

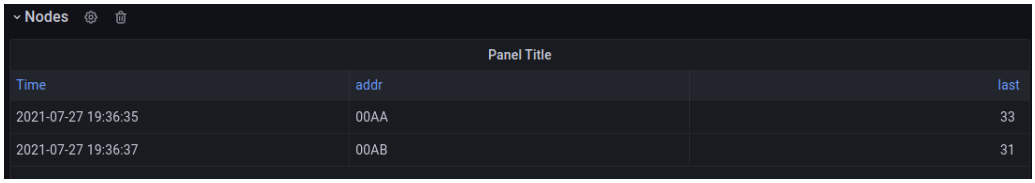

Figura 5.9: *Panel con información de los nodos*

Figura 5.10: *Panel con gráficas de las mediciones*

compone de dos paneles colapsables. Primero, tendríamos un panel llamado *nodes* donde visualizaríamos información de los nodos que están emitiendo mediciones 5.9. Por otro lado tendríamos otro panel llamado *graphics* donde tendríamos las gráficas de las mediciones. Lo que se propone es un panel general para ambas mediciones y luego gráficas de las mediciones por cada nodo como vemos en la figura 5.10

Capítulo 6

Conclusiones y trabajo futuro

A lo largo de esta memoria se ha podido ver como se han ido cumpliendo los objetivos marcados en un inicio, concretamente:

- **Firmware:** Este era el objetivo más importante. Se ha desarrollado un firmware capaz de funcionar bajo el *framework* de Espressif, ofreciendo una funcionalidad básica, con posibilidades de extensión, y con soporte completo para el modelo *sensor* de la especificación BLE Mesh.
- **CLI:** No solo se ha desarrollado el firmware sino que también se ha creado una pequeña herramienta por linea de comandos para poder interactuar con el cliente BLE Mesh, y así sacarle mayor partido a la red *mesh*.
- **Dashboard:** Por último, para poder visualizar los datos de sensorización, se ha desarrollado un dashboard en Grafana para poder monitorizar la red.

Al tratarse de una tecnología prácticamente nueva, se han encontrado dificultades a la hora de encontrar documentación o código de ejemplo para poder afianzar aun mas los conocimientos. Aun así, gracias al Máster de Internet de las Cosas, me he visto preparado para afrontar tales retos a través de los conocimientos adquiridos en diversas asignaturas. De hecho, el desarrollo del firmware tiene relación con la asignatura de Arquitectura de nodo IoT. También, la parte de MQTT y BLE tienen su desarrollo en las asignaturas Redes, Protocolos e Interfaces I y II.

Este proyecto tiene como objetivo ofrecer todas las piezas necesarias en un despliegue BLE Mesh. Esto incluye varios componentes como se han visto, tales como: un dashboard, un provisionador y un firmware para los clientes y servidores. Este firmware es una parte importante del trabajo donde, mas que intentar utilizar cada uno de los mensajes BLE Mesh para el modelo sensor, se ha querido dar una funcionalidad capaz de sacar partido a los dispositivos y que el firmware tenga un flujo de trabajo propicio para ir mejorándolo. Una linea de trabajo futuro seria implementar todos los mensajes que soporta el modelo sensor para poder utilizar los dispositivos de manera completa.

Desde la CLI podemos lanzar tareas sobre los nodos pero, ¿que ocurre si tenemos 5 nodos y solo estamos pidiendo mediciones a 2? En el dashboard podemos saber quien está enviando mediciones pero no podemos saber que nodos están vivos sin ser utilizados. Para esto existe un modelo llamado health que permite enviar mensajes para saber que un nodo está vivo. Esto seria muy interesante para poder pintarlo en el dashboard y ver que nodos tenemos en nuestra red, estén emitiendo mediciones o no, incluso dibujar un esquema de los dispositivos con sus datos que los identifiquen.

Capítulo 7

Conclusions and future work

Throughout this document we have been able to see how the objectives set at the beginning have been achieved, specifically:

- **Firmware:** This was the most important objective. A firmware capable of running under the Espressif framework has been developed, offering basic functionality, with extension possibilities, and with full support for the sensor model of the BLE Mesh specification.
- **CLI:** Not only the firmware has been developed but also a small command line tool has been created to interact with the BLE Mesh client, in order to get the most out of the mesh network.
- **Dashboard:** Finally, in order to visualize the sensorization data, a dashboard has been developed in Grafana to monitor the network.

Being a practically new technology, there have been difficulties in finding documentation or example code to further strengthen the knowledge. Even so, thanks to the Internet of Things Master, I have been prepared to face such challenges through the knowledge acquired in various subjects. In fact, firmware development is related to the IoT Node Architecture course. Also, the part of MQTT and BLE have their development in the subjects Networks, Protocols and Interfaces I and II.

This project aims to provide all the necessary pieces in a BLE Mesh deployment. This includes several components as we have seen, such as: a dashboard, a provisioner and firmware for the clients and servers. This firmware is an important part of the work where, rather than trying to use each of the BLE Mesh messages for the sensor model, we wanted to provide a functionality capable of taking advantage of the devices and that the firmware has a workflow conducive to improve it. A future line of work would be to implement all the messages supported by the sensor model to be able to use the devices in a complete way.

From the CLI we can launch tasks on the nodes but, what happens if we have 5 nodes and we are only requesting measurements to 2? In the dashboard we can know who is sending measurements but we cannot know which nodes are alive without being used. For this there is a model called health that allows to send messages to know that a node is alive. This would be very interesting to be able to paint it on the dashboard and see which nodes we have in our network, whether they are emitting measurements or not, even draw a diagram of the devices with their data that identify them.

Capítulo 8

Introduction

The data represented in Figure 8.1 reveals the importance that IoT (Internet of Things) has taken on in recent years. The reason, in part, lies in the adoption of mobile networks such as 2G/3G/4G and now 5G, which allow increasingly high-speed transmission from any corner of the globe. There has also been an increase in the number of devices that are connected, such as smartwatches, smart bracelets, home appliances in general, assistants, televisions, vehicles, buildings and so on. In general, we have a wide variety of different devices that can connect to each other and exchange information, a phenomenon that brings a new challenge for the adoption of new communication technologies and different network topologies that can adapt to each scenario.

Definitions for IoT are very diverse and disparate, although they all have one thing in common: the interconnection of different types of devices to exchange information. One type of network architecture widely used in IoT is based on mesh topologies. This type of architecture allows a plethora of nodes to be connected to each other, with "many-to-many" connections; they are also characterized by the fact that they allow nodes to be configured dynamically, so that if a node falls or is added to the network, it does not interfere with any communication (or if it does, it will do so transiently). In fact, this is the most important feature, since in terms of reliability it is more advantageous than other types of topologies. For example, in a home network all the information usually goes through a central point or router and, if it goes down, all communication is cut off; on the other hand, in a mesh

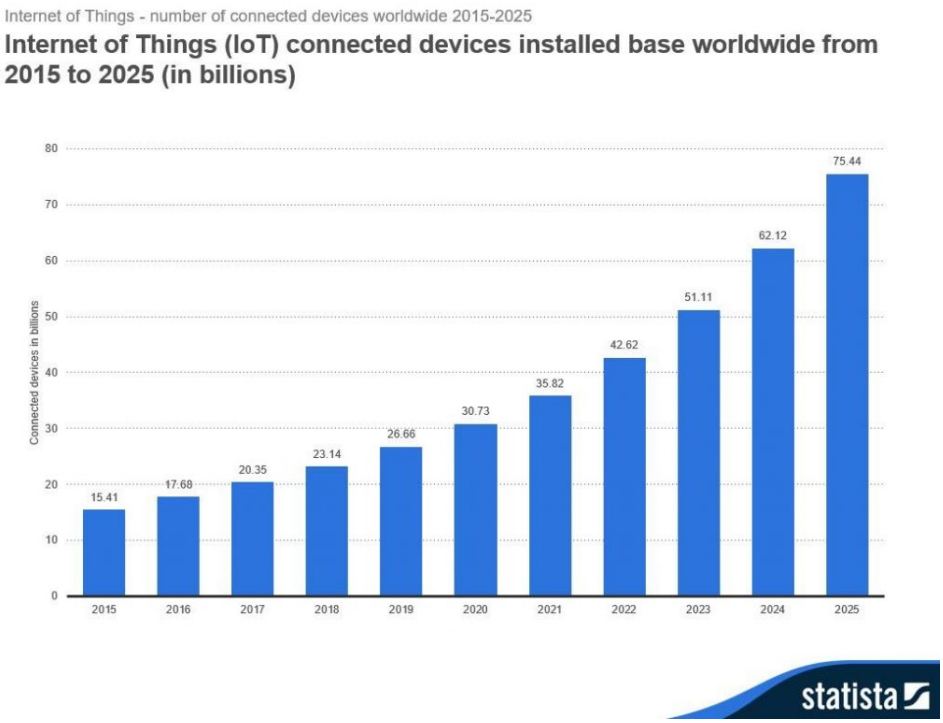

Figura 8.1: *Number of connected devices per year [7].*

network, the information can take different paths until it reaches its destination, since it is not a centralized network.

Mesh network architecture can be built on top of different network technologies such as WiFi, 6LowPan, LoRa and BLE. WiFi Mesh basically allows us to have a mesh network where the network layer is given by WiFi; 6LowPan is a standard that allows us to use IPv6 over networks operating under the IEEE 802.15.4 standard; LoRa is a wireless technology that uses a type of modulation patented by Semtech which allows communication over long distances.

In general terms, an IoT project is characterized by data collection, data ingestion and processing, monitoring and configuration. With this in mind, the aim of this work is to provide an application for the deployment of devices that work with Bluetooth Low Energy (BLE) Mesh technology, together with a control panel and a tool that allows us to configure them.

8.1. Motivation

BLE Mesh is a recent technology, of great application in different environments where not much information can be found. The first reason for doing this work is to try to go deeper into how this technology works, what possibilities it offers, what requirements we need or what scenarios are indicated.

When developing a BLE Mesh deployment, it is not only necessary to know the protocol itself, but it is also essential to arm it with functionality: ultimately, the protocol is the means by which the data is sent or the way in which it is sent. Therefore, much of this work focuses on the development of a firmware capable of running BLE Mesh on a low-power device.

Within BLE Mesh, we are going to make use of the sensor model. This model, without going into details of what it implies, allows us to have a configuration and make use of a sensor in a generic way, i.e., regardless of the sensor we use, the model is suitable for any sensor regardless of the type of sensing we do or the configuration we require. The fact of creating a general purpose firmware that works with the sensor model is motivating since no complete open source example has been found, except for code fragments that we can find on the net.This implies an effort for the development, since it is necessary to separate and organize the code well, try to meet minimum requirements for a type of mesh architecture etc. This has more to do with the engineering or architecture of a project, and this is where the third pillar of the work lies: organizing the development of a large hardware/software project.

In any application where you can configure or control certain devices, you will always find a control panel. If we also have the necessary technical knowledge, we can do it ourselves with HTML, CSS and Javascript/jQuery or using a CSS framework, although there are many tools on the market that allow us to develop dashboards using flow-based tools, such as Node-Red. This is another reason to focus this work and it is the creation of a dashboard together with a command line tool from which to control our mesh network, monitor it, and

get the most out of it within the framework of a BLE Mesh network deployment. This will allow us to have a complete project with all the important parts developed.

8.2. Objectives

The objective of this work is two-fold. On the one hand, to deepen and document in memory the BLE Mesh protocol, specifically all those relevant aspects to better understand the developed firmware and all the elements that are part of a typical BLE Mesh deployment. On the other hand, to develop a firmware that works under the ESP-IDF [5] framework capable of operating in a BLE Mesh network together with tools to monitor and interact with such network. These general objectives can be divided into the following specific objectives:

- Document the Bluetooth Low Energy protocol on which Bluetooth Mesh is built. This will allow us to know better the protocol to give way to the models, key piece in the BLE part of our firmware.
- Compare BLE Mesh with other similar technologies to give arguments why this protocol has been chosen.
- Explain what material has been used for this project. It will be important in order to have a global vision of the proposed deployment and what kind of devices we need.
- Document and explain the proposed development for a deployment of a network of devices working under BLE Mesh.
- Document and explain the proposed development for a control panel where we monitor the information received by the BLE Mesh network.
- Document and explain the proposed development of a command line tool (CLI) that will allow us to interact with our network.
- Document and explain the proposed project configuration of all deployment components.

Therefore, the report will be divided into a series of chapters dealing with the above-mentioned topics. Chapter 1 presents, at a high level, an overview of the work developed. Then, we will continue with chapter 2 where we will see in a theoretical way aspects related to Bluetooth Mesh in relation to the work developed, to move on to chapter 3 where we will see the development itself, both the firmware and the other components involved. Then, in chapter 4 we will see what is related to the provisioner and the provisioning process in relation to the BLE Mesh protocol. Finally, we will see chapter 5 where we will talk about the CLI and the dashboard.

8.3. Deployment Architecture

The idea of this project is to offer a firmware, under the Espressif framework, with the capability to work within a BLE Mesh network. On the other hand, when working with the sensor model, we must be able to obtain the measurements of the nodes or other interesting information and be able to change parameters of the model.

A mesh network architecture is a mesh architecture where, specifically in the case of BLE, we have two types of nodes: client and server. The server is in charge of taking measurements, providing information about its characteristics or modifying some of them. On the other hand, the client will be in charge of executing actions in the form of BLE Mesh messages. This means that it will ask for a measurement value, information about a feature or modify a feature for a specific server or servers. On the other hand, we need to provision all nodes with network and application keys so that they can communicate with each other and the information travels securely. Putting these ideas together, what we propose is a deployment architecture as shown in the figure 8.2. If we look at the scheme, we have on one side the BLE Mesh network, which is composed of the ESP32 development boards that will function as servers except one of them that will be the client. We have a Raspbery pi that will be in charge of provisioning the network nodes. The dashboard will have the function of painting any kind of interesting information in a more visual way, such as the measurements

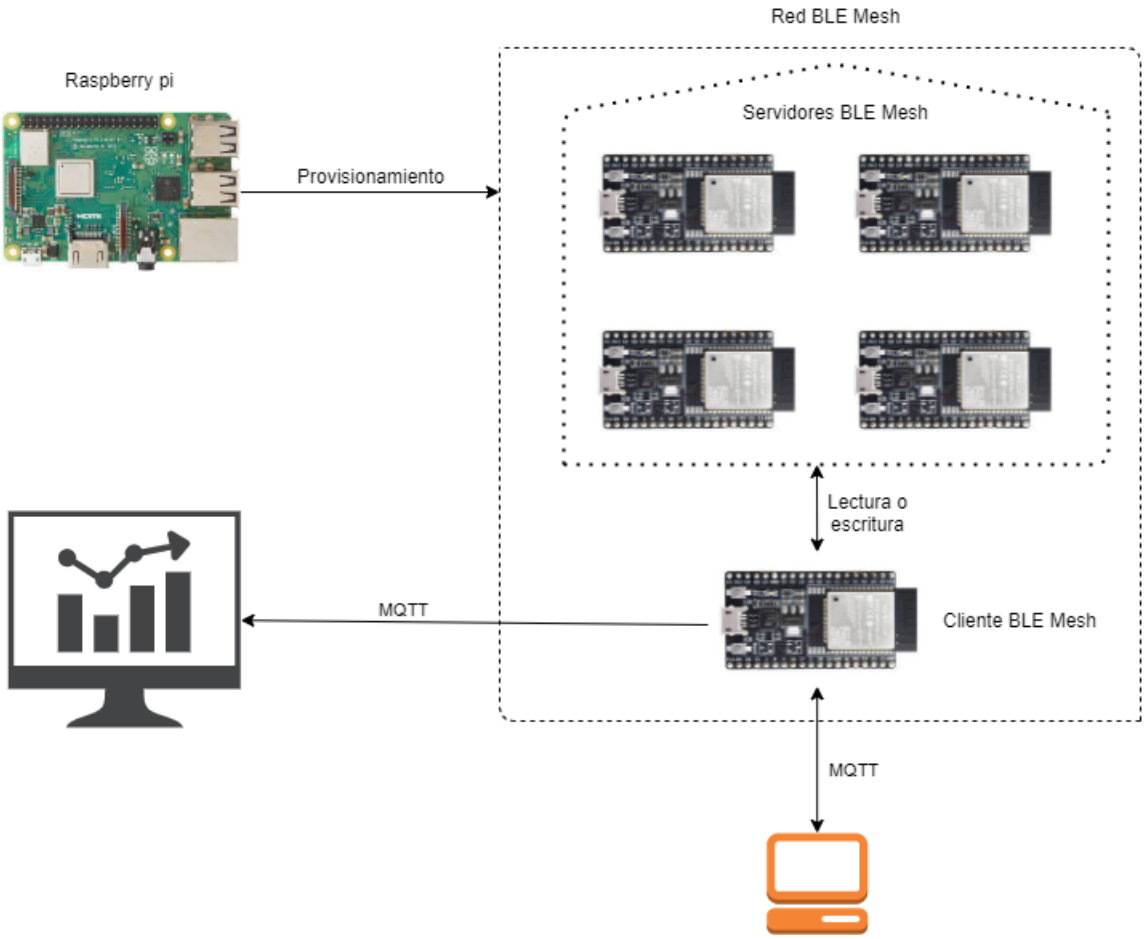

Figura 8.2: *Deployment Architecture.*

that are requested to the servers. Finally, we have a small command line tool (CLI) that we will use to interact with the network.

This architecture is the one proposed in this work since it is suitable for the type of network architecture we use and it is a generic model for any type of BLE Mesh application. It is possible that it may vary depending on the casuistry but, in general, we will have the elements mentioned with this layout. Also, potentially, both the dashboard and the command tool can be deployed on the Raspberry Pi itself.

8.4. Project Structure

As any programming project, we are looking for it to execute a specific functionality or to solve a given problem but, and not less important, we must also look for it to be well organized from the design point of view. This has been the main idea when creating the necessary files for the project, not only that we can get some measurement of the sensors, but if someone wants to make any modification or improvement, it will be as easy as possible without breaking any other point of the firmware. Also, we have made a cascade implementation, that is, when a component is initialized, it also initializes those that must be ready or need the module that is being initialized at this time. On the other hand, as the framework used has been ESP-IDF, all the code has been developed in C language with modular programming to separate the code into .c and .h files and thus have better encapsulation and organization.

8.4.1. Project Components

As seen in figure 8.2 where we talked about the architecture, we have the following components:

- **Raspberry Pi:** It will be our provisioner for the devices that are part of the mesh network. We will need to enable specific kernel options for it to have the functionality required by BLE Mesh for secure key exchange in the provisioning process. Also, potentially, not only can it function as a provisioner but we could also have both the dashboard and CLI hosted.
- **ESP32 as BLE Mesh server:** Operating as a server, it has to listen for any BLE messages from the client.
- **ESP32 as BLE Mesh client:** As a client, it must communicate with the outside and with the BLE Mesh network. On the external side, it must interpret the information

received to execute it or transform it into a BLE Mesh message to be sent to the network, either punctually or periodically.

- **CLI:** It must be able to send messages to the BLE Mesh client to execute a particular action. It must also be able to receive feedback from the BLE Mesh client to know if the action has been executed successfully or not.

- **Dashboard:** Since we have a number of servers taking measurements from a sensor, we need to be able to visualize them on a dashboard that is easy to read and can be interpreted correctly for monitoring. In this sense, the client will ask the servers for these measurements so that the client can export them to the outside and the dashboard can illustrate them.

8.4.2. Hardware/Software choosen

The development boards we will use are the ESP32-DevKitC V4 [4] boards. These devices are quite small, offer WiFi and BLE, have a GPIO interface wide enough to use ADCs, communication via I2C etc.. To these boards we will attach via GPIO interface Adafruit si7072 [1] sensors which offer temperature and humidity measurements. These sensors offer a relative humidity measurement of 80 % RH with a range of 0-80 % RH. On the other hand, it offers a temperature measurement accuracy of $\pm$ 0.4°C. For the device provisioning part, there is a fairly well-known Raspberry Pi 3b+ [9] where we have a Cortex-A53 (ARMv8) 64-bit processor at a frequency of 1.4 GHz. It has 1GB LPDDR2 of SDRAM memory and dual-band Wifi connectivity in addition to Bluetooth 4.2 and BLE.

8.4.3. Methodology and planning

The development is the most important part of this work so it is necessary to take special care when designing and implementing the code to be developed; therefore, the GitHub platform was used to upload all the code of this project [**?**]. On ESP32 devices, the development has been carried out under the Espressif framework, which is written in

C. For this reason, and in order to have a better organization, the code has been written with modular programming so that it does not affect several places in the code but that each module has a clear role and, if future modifications are desired, does not affect other parts of the firmware. The command line tool may also be subject to modifications or new features so it must also be developed with these circumstances in mind. Therefore, it has been developed in Ruby, which offers the flexibility of interpreted languages as well as a powerful library for reading command line parameters.

Bibliografía

[1] Lady Ada. *Adafruit Si7021 Temperature + Humidity Sensor.* Disponible en https://learn.adafruit.com/adafruit-si7021-temperature-plus-humidity-sensor.

[2] Mohammad Afaneh. The basics of bluetooth low energy (ble). 2016. Disponible en https://www.novelbits.io/basics-bluetooth-low-energy/.

[3] Grafana Dashboard. *Pagina oficial de grafana.* Disponible en https://grafana.com/.

[4] ESP-IDF. *ESP32-DevKitC V4 Getting Started Guide.* Disponible en https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html.

[5] ESP-IDF. *Documentación*, 2016. Disponible en https://docs.espressif.com/projects/esp-idf/en/latest/esp32/.

[6] Influxdb. *Pagina oficial de Influxdb.* Disponible en https://www.influxdata.com/.

[7] IoT World Online. Las grandes estadísticas del internet de las cosas (iot). 2020. Disponible en https://www.iotworldonline.es/las-grandes-estadisticas-del-internet-de-las-cosas-iot/.

[8] BlueZ Project. *Pagina oficial de BlueZ.* Disponible en http://www.bluez.org/.

[9] Raspberry. *Raspberry Pi 3 Model B+.* Disponible en https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[10] Kai Ren. Provisioning a bluetooth mesh network part 1. 2017. Disponible en https://www.bluetooth.com/blog/provisioning-a-bluetooth-mesh-network-part-1/.

[11] Kai Ren and Martin Woolley. The fundamental concepts of bluetooth mesh networking part 1. 2017. Disponible en https://www.bluetooth.com/blog/the-fundamental-concepts-of-bluetooth-mesh-networking-part-1/.

[12] Bluetooth SIG. *Pagina oficial de Bluetooth SIG.* Disponible en https://www.bluetooth.com/.

[13] Kai Ren Bluetooth SIG. *Repositorio oficial de ESP-IDF en GitHub.* Disponible en https://github.com/espressif/esp-idf.

[14] Kai Ren Bluetooth SIG. Developer study guide using bluez as a bluetooth® mesh provisioner. 2018. Disponible en https://www.bluetooth.com/wp-content/uploads/2020/04/Developer-Study-Guide-How-to-Deploy-BlueZ-on-a-Raspberry-Pi-Board-as-a-Bluetooth-Mesh-Provisioner.pdf.

[15] Telegraf. *Pagina oficial de telegraf.* Disponible en https://www.influxdata.com/time-series-platform/telegraf/.

[16] Martin Woolley. *Bluetooth Mesh Models. Technical Overview.* Disponible en https://www.bluetooth.com/wp-content/uploads/2019/04/1903_Mesh-Models-Overview_FINAL.pdf.

[17] Zephyr. *Zephyr.* Disponible en https://zephyrproject.org/.

[18] Jose Ángel Garrido Montoya. *TFM_BLEMesh*, 2021. Disponible en https://github.com/joseangelgm/TFM_BLEMesh.