industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

Universidad Politécnica de Cartagena

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

Diseño de sistema IoT para sensorización de atmósferas controladas

TRABAJO FIN DE GRADO

GRADO EN INGENIERIA EN TECNOLOGÍAS INDUSTRIALES

Autor: **Javier Gómez Consuegra**
Director: Roque Torres Sánchez
Codirector: Manuel Jiménez Buendía

Cartagena, diciembre de 2020

Agradecimientos

Me gustaría agradecer a mi director, Roque Torres Sánchez por confiar en mí y darme la oportunidad de realizar este proyecto en el que he aprendido mucho, así como por su ayuda y consejos.

A mis amigos, por haber hecho de estos cuatro años de universidad una etapa inolvidable.

A mi familia, especialmente a mis padres, por confiar siempre en mí y apoyarme en cada momento.

A todos vosotros, gracias por acompañarme a lo largo de este proyecto.

Índice de contenidos

Índice de figuras

Resumen

Con este proyecto se plantea el diseño de una plataforma IoT apta para la aplicación en un proceso de transporte de mercancías bajo atmósferas controladas. Para ello, se medirán variables tales como la temperatura, la humedad relativa y la luminosidad mediante un microcontrolador equipado con diferentes sensores.

Para poder enviar los datos a algún servidor, será necesario que el dispositivo esté conectado a alguna red *Wi-Fi*. Por ello, se ha desarrollado una aplicación móvil complementaria encargada de enviar dichas redes y sus respectivas contraseñas, las claves para almacenar los datos en la nube y el tiempo de muestreo entre cada medida a través de la tecnología *Bluetooth Low Energy*.

El microcontrolador será el encargado de, una vez conectado a alguna red, enviar dichas medidas a la nube a través del protocolo de comunicación MQTT. Una vez se hayan enviado los datos, o en su defecto, se hayan almacenado debido a un fallo de conexión, el dispositivo hibernará según el tiempo de muestreo seleccionado hasta el próximo ciclo con el objetivo de disminuir el consumo energético.

Abstract

The goal of this project is to design an IoT platform suitable for controlled atmosphere transportation. To that end, variables such as temperature, relative humidity and luminosity will be measured by a microcontroller equipped with different sensors.

In order to send data to a server, the device needs to be connected to a Wi-Fi network. With that goal, a complementary mobile app has been developed for the task of sending the details of these networks, the keys for the cloud service and the sampling time through Bluetooth Low Energy technology.

Said microcontroller is in charge of sending the data to the cloud using an MQTT communication protocol. Once the information has been sent or stored in case there is any issue with the network connection, the device will sleep until the next cycle according to the sampling time, with the objective of reducing power consumption.

Capítulo 1. El “Internet de las cosas”

Antes de comenzar con el proyecto, conviene explicar qué es el Internet de las cosas, sus orígenes y su situación en la actualidad.

1.1. Introducción [1] [2]

Según la IUT (Unión Internacional de Telecomunicaciones), el Internet de las cosas, o IoT (*Internet of Things)* por sus siglas en inglés, se entiende como “una sociedad definida por "cosas" inteligentes que pueden comunicarse entre sí directamente o a través de una red”. Se trata de la realización de la idea de que todo puede ser conectado en cualquier parte y en todo momento, y es un concepto que puede aplicarse a una grandísima variedad de sectores.

Una definición similar pero simplificada viene descrita por los investigadores Peter Friess y Patrick Guillemin en su artículo *Internet of Things Strategic Research Roadmap*: “El Internet de las cosas permite a "cosas" y personas estar conectadas en cualquier lugar, momento, a cualquier hora e idealmente utilizando cualquier red camino o servicio”.

Pero,¿cómo funciona un sistema IoT? A modo de ejemplo, se utilizará como ejemplo un sistema básico:

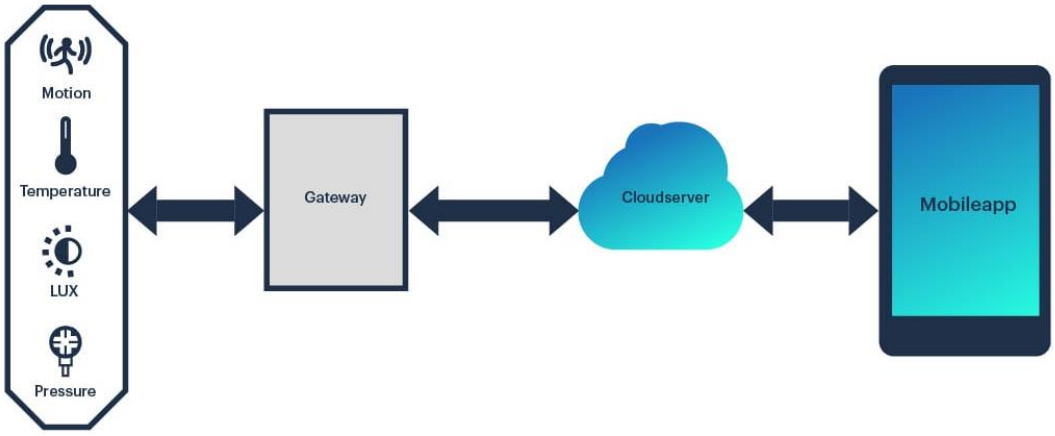


Figura 1.1. Ejemplo de un sistema IoT.

Fuente: https://www.baianat.com/articles/how-iot-is-transforming-industries

La ilustración muestra los diferentes pasos realizados por el sistema en cada ciclo de trabajo.

En una primera instancia, tenemos un dispositivo electrónico, generalmente un microcontrolador, equipado con sensores cuya tarea es recolectar información de los parámetros a analizar en un intervalo de tiempo indicado. En este caso, tenemos un dispositivo encargado de captar el movimiento y los valores de temperatura, luminosidad y presión del ambiente, pero puede ser casi cualquier otro ejemplo aplicado en una gran multitud de sectores.

Una vez captada dicha información, se necesita un “medio de transporte” para poder almacenar la información en algún tipo de servidor. En este ejemplo, el dispositivo se conecta a una red *Wi-Fi* a través de una puerta de enlace o *gateway*, que actúa de pasarela para que el dispositivo se

pueda conectar a dicha red, y envía los datos a un servidor en la nube. También es posible realizar esta conexión mediante otros tipos de tecnologías inalámbricas como puede ser *Bluetooth*.

Una vez que dicha información llega al servidor, se realiza un procesado de datos si fuese necesario, y finalmente se muestran los datos a través de una interfaz que facilite la comprensión y la monitorización de éstos. Esta interfaz puede estar disponible en forma de una aplicación para el móvil o el ordenador, o incluso se puede acceder a ella a través de una página web.

Este proceso se repite de forma periódica, lo que permite la comunicación entre el usuario y el dispositivo para cualquier instante y en cualquier lugar.

1.2. Historia [3] [4]

El término de Internet de las cosas es relativamente moderno, pero el origen de los objetos conectados no es algo de hace unas pocas décadas. Realmente, esta idea se remonta al siglo XIX, en el que tuvieron lugar los primeros experimentos telemétricos de la historia.

De entre todos, el primero que se conoce tuvo lugar 1874 de la mano de científicos franceses. En este primer experimento, una serie de dispositivos fueron instalados en la cima del *Mont Blanc* con el objetivo de obtener información meteorológica y de profundidad de nieve. Los datos captados por dichos dispositivos eran transmitidos a París a través de emisiones de onda corta.

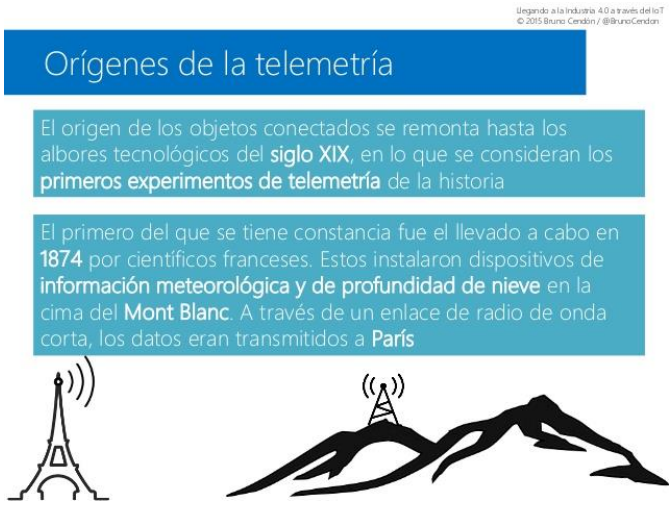


Figura 1.2. Orígenes de la telemetría

Fuente: http://www.bcendon.com/

Otros experimentos, ya en el siglo XX, tuvieron lugar en países como Estados Unidos o Rusia, impulsando el crecimiento de la telemetría gracias a los avances tecnológicos de la época en cuanto a las tecnologías de comunicación.

La posibilidad de conectar y crear objetos inteligentes era una idea ya existente en esta época de la mano de notables científicos como Nikola Tesla o Alan Turing.

En 1926, Nikola Tesla tuvo una entrevista con la revista *Colliers*, en esta, el científico ya advierte del futuro crecimiento de las tecnologías inalámbricas:

"Cuando lo inalámbrico esté perfectamente desarrollado, el planeta entero se convertirá en un gran cerebro, que de hecho ya lo es, con todas las cosas siendo partículas de un todo real y rítmico… y los instrumentos que usaremos para ellos serán increíblemente sencillos comparados con nuestros teléfonos actuales. Un hombre podrá llevar uno en su bolsillo"

También cabe destacar las palabras de Alan Turing en 1950 en un artículo para el *Computing Machinery and Intelligence in the Oxford Mind Journal*, en el que ya avanzó la necesidad de otorgar inteligencia a los dispositivos sensores:

"…también se puede sostener que es mejor proporcionar la máquina con los mejores órganos sensores que el dinero pueda comprar, y después enseñarla a entender y hablar inglés. Este proceso seguirá el proceso normal de aprendizaje de un niño"

A pesar de que esta visión tan temprana fue compartida por numerosos científicos de la época, la inmadurez tecnológica hizo que todo esto quedase en segundo plano. No fue hasta la década de 1970 cuando aparecieron los primeros protocolos de comunicación que sentarían las bases que hoy conocemos como Internet. Estos avances tecnológicos comenzaron a desarrollarse en el seno del Departamento de Defensa de los Estados Unidos, a través de la red ARPANET, en 1977. Sin embargo, durante los primeros años estos protocolos fueron exclusivamente de uso militar.

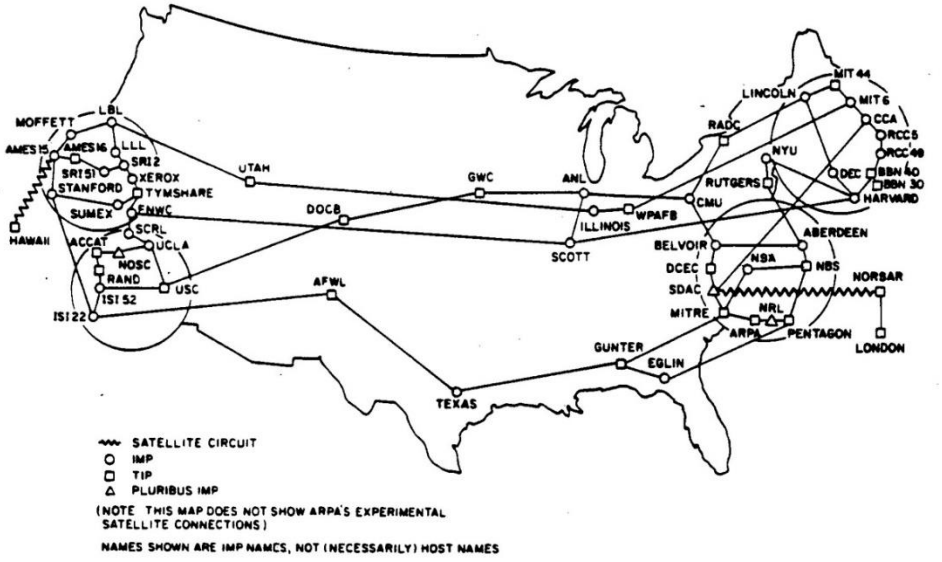


Figura 1.3. ARPANET en 1977
Fuente: http://som.csudh.edu/fac/lpress/history/arpamaps/

No fue hasta mediados de los años 90 que el Internet comercial comenzó su expansión definitiva. Con la aparición del famoso protocolo de comunicación TCP/IP, se sentaron las bases de Internet y fue posible la comunicación a media y larga distancia.

Con la popularización de Internet la idea de conectar objetos a través de esta red comenzó a popularizarse. Sin embargo, debido a que inicialmente era un protocolo principalmente cableado y dado que el coste del hardware era muy elevado, esta idea pasó inicialmente inadvertida.

El término de *Internet of Things* fue acuñado en 1999 por Kevin Ashton, fundador del *MIT Auto-ID Center* (Actual *Auto-ID Labs)*, que es un grupo de investigación en el campo de la identificación por radiofrecuencia y las tecnologías de detección emergentes. Ashton decía que el IoT tenía "potencial para cambiar el mundo, al igual que lo hizo Internet, quizás incluso más". Desde entonces, el crecimiento y la expectación alrededor del término ha ido creciendo de forma exponencial.

A principios del siglo XXI, con la popularización de la conexión inalámbrica, comenzó el crecimiento de los dispositivos conectados, especialmente a lo largo de la última década que dio paso al Internet de las cosas tal y como lo conocemos, siendo actualmente uno de los campos de investigación con mayor proyección.

1.3. Estado del arte [4]

Como ya se ha comentado, el Internet de las cosas fue un tema que inicialmente pasó inadvertido, pero que, gracias a los avances tecnológicos de este último siglo se ha convertido en uno de los campos con mayor expansión.

Según Google, el número de artículos periodísticos ha incrementado, hasta el punto de casi duplicar las cifras, entre 2004 y 2010. Desde dicho año, las cifras han ido aumentando de forma muy significativa. A continuación, se muestra el número de artículos desde 2004 hasta 2017.

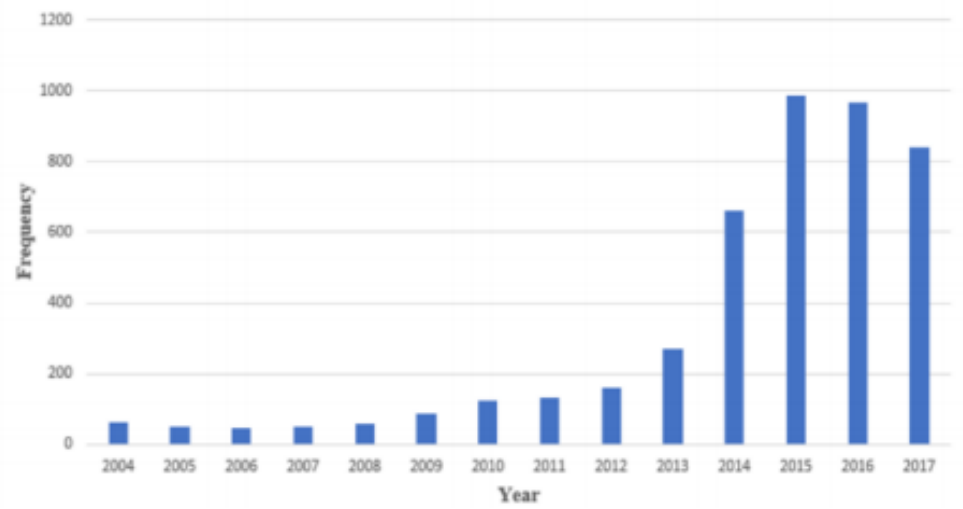


Figura 1.4. Número de artículos sobre IoT por año.

Fuente: https://infonomics-society.org/

Esta información también justifica el aumento que se ha producido en el número de personas y dispositivos conectados:

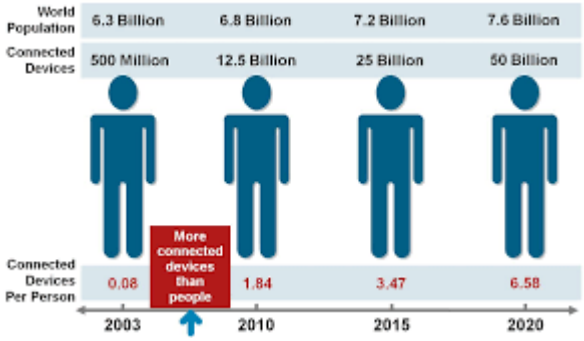


Figura 1.5. Número de "cosas" conectadas a Internet.

Fuente: https://www.cisco.com/

Desde aproximadamente 2008 ya existían más dispositivos conectados que personas y las previsiones sugieren que durante este año se habrán alcanzado o incluso superado los 50 billones de dispositivos conectados. Además, el desarrollo de las plataformas IoT se ha convertido en uno de los mercados tecnológicos más masivos, permitiendo el ahorro de billones de dólares a las compañías.

Dada la capacidad de conectar cualquier elemento de uso cotidiano a Internet, el IoT tiene una gran relevancia en numerosos campos, otorgando "inteligencia" a diferentes aplicaciones y servicios. Por poner un ejemplo, es posible informar inmediatamente al dueño de un invernadero si las condiciones de humedad y temperatura no son las ideales.

Algunos otros ejemplos se muestran a continuación:

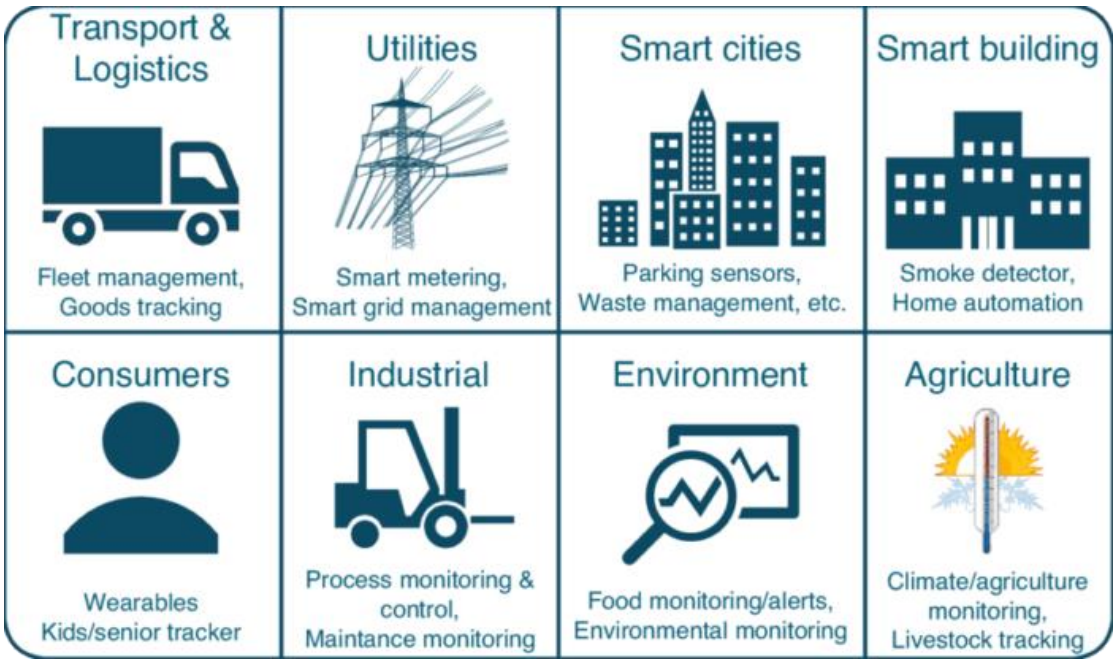


Figura 1. 6. Algunos campos de aplicación para el IoT.

Fuente: https://vizah.ch/en/developing-iot-applications-best-technologies-and-tools-for-iot-developers/

Todo esto justifica la definición de Internet de las cosas como una infraestructura global capaz de conectar tanto a humanos como a "cosas" en cualquier lugar y a cualquier momento.

Capítulo 2. Justificación del proyecto

2.1. Problemática [5]

Una de las mayores cualidades del Internet de las cosas es su versatilidad. Sin embargo, a medida que sus aplicaciones se han ido diversificando, han aparecido una serie de retos a afrontar por los ingenieros y desarrolladores durante la fase de diseño.

Concretamente para el transporte de mercancías bajo atmósferas controladas, podemos encontrar los siguientes obstáculos:

- **Conectividad:** Para poder mantener un registro de datos en el servidor, es necesario que exista un flujo continuo de información desde un dispositivo hacia él. Esto supone una tarea compleja debido a las limitaciones de la conectividad inalámbrica en ciertas situaciones.

- **Consumo energético:** El continuo proceso de monitorización del ambiente implica que es necesario tener un flujo de energía ininterrumpido, tanto para alimentar a los sensores como para que el dispositivo pueda enviar posteriormente la información. El hecho de que el espacio sea limitado y esto afecte a las baterías dificulta aún más el desarrollo del proyecto.

- **Configuración del dispositivo:** El funcionamiento del sistema debe ser comprendido no solo por el diseñador, sino también por el personal que vaya a utilizarlo. Por ello, es necesario que los distintos dispositivos, así como las interfaces para el usuario, sean lo más sencillos posible, de manera que sean capaces de visualizar el estado en el que se encuentre el proceso.

2.2. Finalidad y objetivos

Con el fin de solventar estos problemas, en este proyecto se programará un microcontrolador con la tarea de enviar o almacenar, en caso de que no haya sido posible conectarse a una red, los datos de las mediciones realizadas por los sensores conectados al mismo.

Esta información se enviará a la *Adafruit IO* (*AIO*), un servicio de nube que permite monitorizar datos de manera online, a través de un protocolo (MQTT), que se explicará más adelante. Una vez enviados o guardados los datos a la hora correspondiente, el dispositivo hibernará para reducir el consumo energético.

Para que el proceso sea lo más personalizable posible, se desarrollará también una aplicación para móviles que permitirá conectarse con el microcontrolador para proporcionarle los detalles de las redes a las que se ha de conectar, la información del usuario de la *AIO* y el tiempo de muestreo. Esta conexión se realizará a través de *Bluetooth Low Energy*, que también se explicará más adelante.

A modo de resumen, los objetivos del proyecto son los siguientes:

- Garantizar que todas las mediciones llegan al servidor, con información relativa a la hora de la toma de datos. En caso de que exista un error en la conexión, el microcontrolador deberá guardar el mensaje y enviarlo en sus posteriores ciclos de trabajo.

- Optimizar el consumo energético del proceso. El dispositivo permanecerá despierto solo durante el tiempo necesario para abrir el punto de acceso *Bluetooth* (si procede), conectarse a una red, tomar la medida correspondiente y enviar los datos. En cualquier otro instante, el dispositivo deberá permanecer en modo *deepsleep*, en el que estará hibernando para reducir de manera significativa el consumo de la batería.

- En cuanto a la aplicación móvil, deberá de ser capaz de proporcionar no solo las diferentes redes a las que se debe conectar el microcontrolador, sino que también debe ser posible cambiar el usuario de la *AIO* y el tiempo transcurrido entre cada toma de datos. Para facilitar la introducción de las redes y de la clave del usuario, se implementará un escáner de código QR capaz de decodificar dicha información.

Capítulo 3. *Hardware* empleado

3.1. *Wipy* 3.0 [6]

El *Wipy 3.0* es una placa de desarrollo o microcontrolador de *Pycom*, empresa dedicada a la tecnología del Internet de las cosas y que ofrece diferentes soluciones de *software* y *hardware* para ésta.

Todas las placas de desarrollo de esta empresa se programan a través de *MicroPython*, una implementación del lenguaje de programación *Python 3* que permite que éste sea ejecutado de manera eficiente en microcontroladores y entornos restringidos.

Concretamente, este modelo es una placa de desarrollo IoT para *Bluetooth* y *Wi-Fi*.

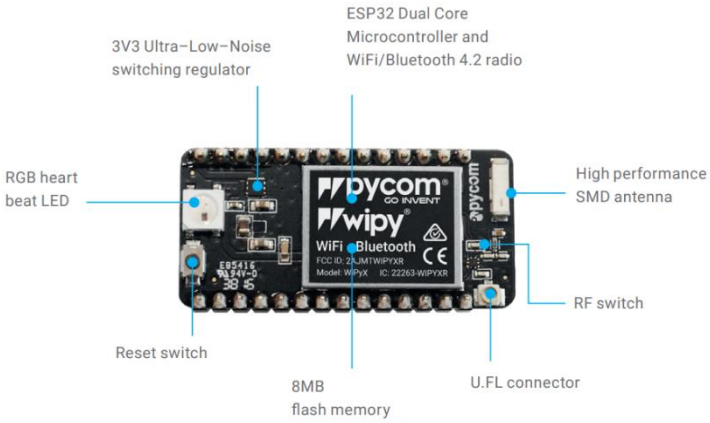


Figura 3.1. ***Wipy 3.0***

Fuente: https://pycom.io/product/wipy-3-0/

Funcionalidades

- Potente microprocesador, permite conexión por *BLE* y *Wi-Fi*.
- Alcance *Wi-Fi* de hasta 1 km.
- *MicroPython* habilitado.
- Consumo de energía muy reducido.

Características eléctricas

- **Tensión de alimentación a la entrada (V_{in}):** De 3.5 a 5.5V.
- **Tensión de alimentación a la salida (V_{out}):** 3.3V.
- **Corriente a la salida (I_{out}):** Hasta 1.2 A.

En la página web del fabricante podemos encontrar la siguiente figura:

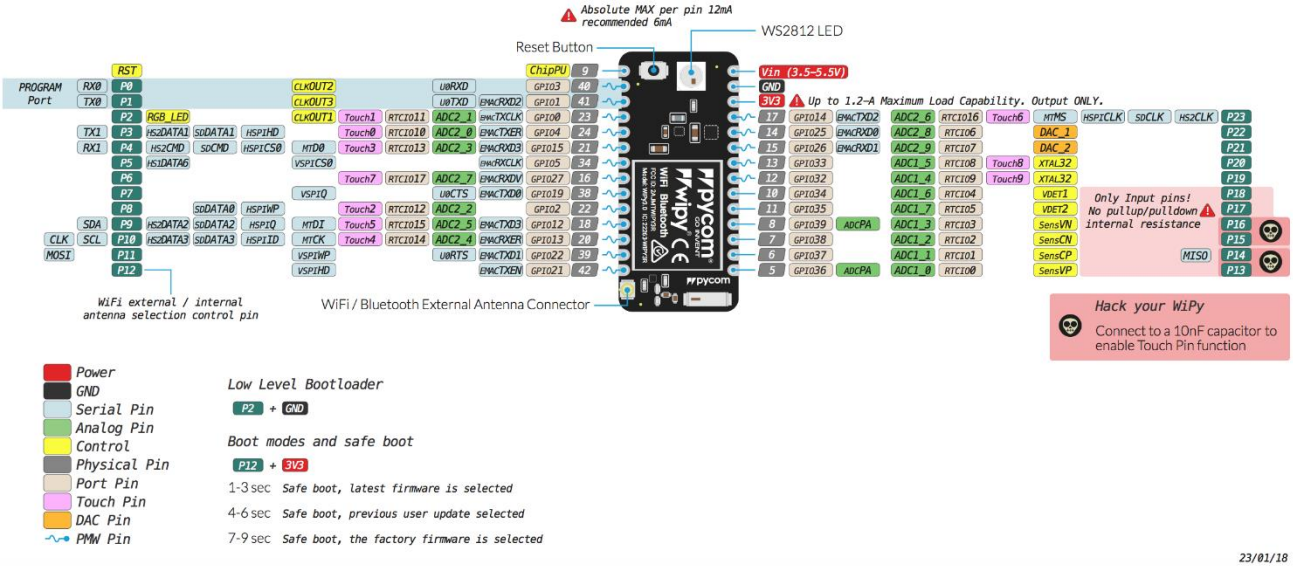


Figura 3.2. ***Disposición de los pines del Wipy*** **y características eléctricas.**

Fuente: https://docs.pycom.io/datasheets/development/wipy3/

3.2. Sensor de temperatura y humedad DHT22 [7]

Se trata de un sensor digital de bajo coste que permite medir la humedad y la temperatura del ambiente. Para ello, utiliza los siguientes elementos:

Para medir la humedad, utiliza un sensor capacitivo cuyo material dieléctrico absorbe o elimina vapor de agua del ambiente con los cambios de humedad. Esto afecta a la impedancia del sensor y produce un cambio en la medida registrada.

En cuanto a la temperatura, se utiliza un termistor, que no es más que una resistencia cuyo valor se ve afectado por la temperatura.

Se trata de un elemento muy básico, cuyo único inconveniente es que solo permite lecturas cada 2 segundos, condición que no nos afecta en el proyecto.

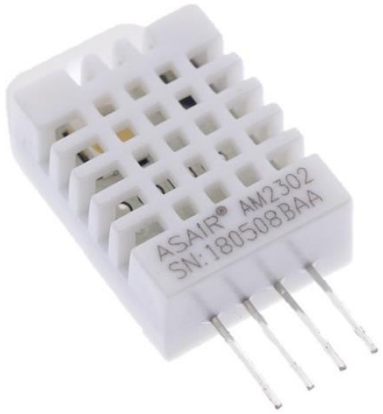


Figura 3.3. Sensor de temperatura y humedad.

Especificaciones técnicas

- Tensión de alimentación de 3.5 a 5V.
- Corriente máxima de 2.5mA
- Medidas de 0 a 100% de humedad relativa con un error de precisión en valor absoluto del 2 al 5%.
- Medidas de temperatura desde -40°C a 80°C, con un error de precisión de ±0.5°C.
- Frecuencia máxima de muestreo de 0.5 Hz (2 segundos).
- Posee 4 pines (alimentación, tierra, transmisión de datos y uno no conectado)

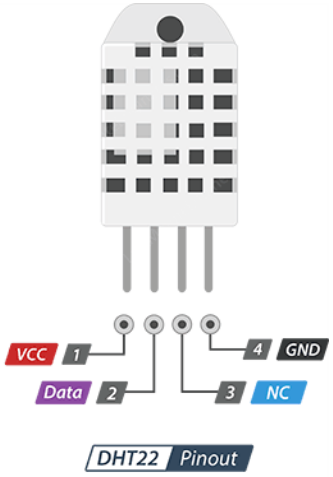


Figura 3.4. Pines del sensor de humedad y temperatura.

Fuente: https://lastminuteengineers.com/dht11-dht22-arduino-tutorial/

3.3. Fotorresistencia [8]

Figura 3.5. Fotorresistencia

Fuente: https://www.sparkfun.com/products/9088

Una fotorresistencia o LDR (*Light Dependent Resistor*) es un sensor utilizado para medir la luminosidad. Consta de una resistencia eléctrica variable cuyo principio de funcionamiento es la fotoconductividad. Este fenómeno óptico hace que cuando la luz incide en la superficie la conductividad del material aumenta debido a la absorción de radiación electromagnética.

Por ello, en zonas con mucha iluminación la resistencia será pequeña, mientras que en zonas muy poco iluminadas el sensor actuará como una gran resistencia de hasta 10 MΩ.

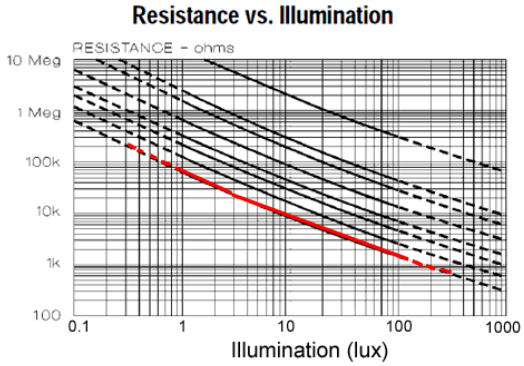


Figura 3.6. Resistencia del LDR en función de la iluminación.
Fuente: https://learn.adafruit.com/photocells/measuring-light

Especificaciones técnicas

- **Resistencia mínima:** 1 KΩ.
- **Resistencia máxima:** 10 KΩ.
- **Tensión máxima:** 150 V.

Estos tres elementos, que se conectan mediante una placa de expansión y se alimentan mediante una batería de 4 V, constituyen el dispositivo encargado de enviar o almacenar los valores de las medidas de temperatura, humedad y luminosidad registrados en el tiempo.

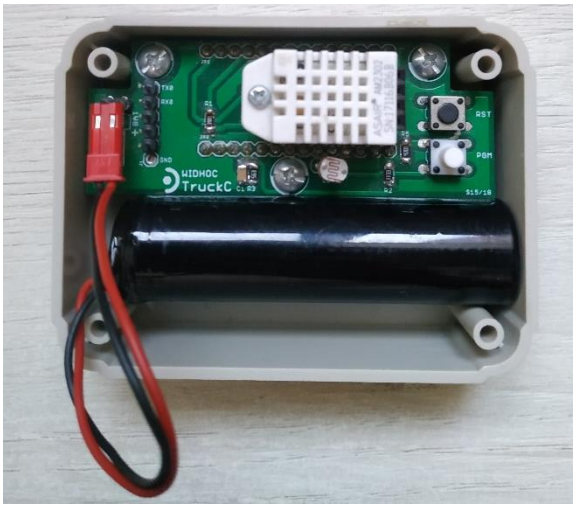


Figura 3.7. Montaje del dispositivo.

Al final del trabajo, en el **Anexo 1**, se puede ver el esquema de conexiones de este dispositivo.

Por otro lado, también es necesario el uso de un *smartphone* para la aplicación que le da las claves al microcontrolador.

Capítulo 4. *Software* y lenguajes de programación empleados.

4.1. *MicroPython* [9] [10]

Como ya se ha comentado en el subapartado del *Wipy*, el lenguaje de programación de este circuito integrado es *MicroPython*, que se trata de una implementación de *Python 3* para sistemas embebidos.

Dispone de ciertos módulos con una grandísima utilidad en proyectos relacionados con el Internet de las cosas, como pueden ser:

- *Network*: Permite la conexión por *Wi-Fi* y *Bluetooth*.
- *Machine*: Contiene diferentes funciones relativas al *hardware* del microcontrolador, como el reloj en tiempo real, *resets*, eventos en los pines o el modo *deepsleep* que permite reducir el consumo de energía hibernando el dispositivo, entre otras.
- *Time* y *utime*: Con funciones que permiten obtener una fecha y hora concreta, así como sincronizar el reloj en tiempo real con la fecha y hora actuales.
- *Json:* Especialmente útil para la transferencia de datos, ya que permite convertir los diferentes tipos de datos y objetos de *Python* a este formato (*JavaScript Object Notation*)

Estos son ejemplos de los módulos más importantes que se han utilizado en el proyecto. La documentación de éstos y de muchos más está disponible tanto en la página web de *MicroPython* como en la de *Pycom*.

4.2. *Atom* y *Pymakr*

Para programar el circuito integrado, es un editor de código sea compatible con *Python* y que además soporte el *plug-in Pymakr*.

Pymakr es un complemento, desarrollado por *Pycom*, que permite que detectar el microcontrolador y realizar un seguimiento del programa una vez subidos los archivos o descargar los archivos que tenía programados.

Atom es un editor de código multiplataforma que tiene una gran capacidad de customización y que tiene multitud de paquetes descargables adicionales, entre ellos, el complemento *Pymakr*, principal motivo por el que se ha seleccionado este editor.

A continuación, se muestra un ejemplo de la interfaz.

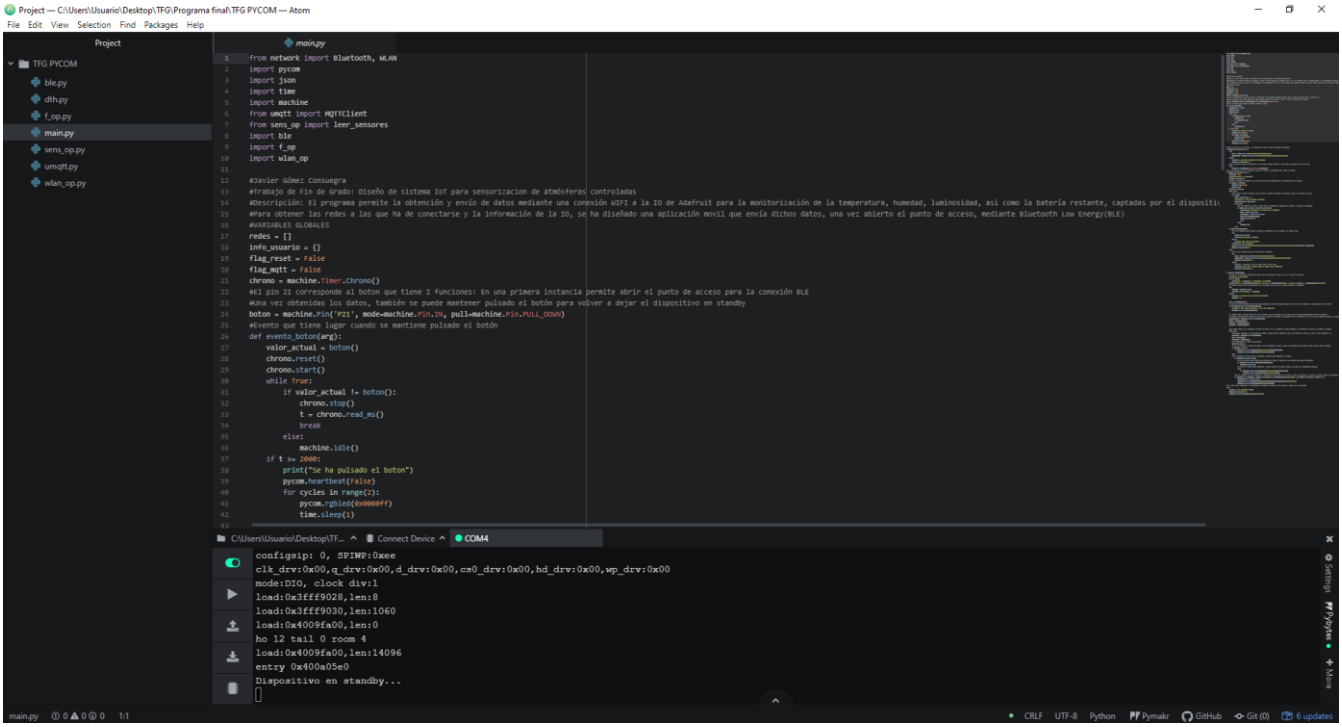


Figura 4.1. Interfaz de *Atom* con el complemento *Pymakr*.

4.3. *Adafruit IO* [11]

Adafruit IO (*AIO*) es un servicio en la nube desarrollado por la empresa electrónica *Adafruit*. Un servicio en nube es una plataforma que se pone a disposición del usuario a través de Internet con la finalidad de facilitar el flujo de datos a través de este. Principalmente se utiliza para el almacenamiento y la recuperación de información, pero también tiene otras funcionalidades, como pueden ser:

- Monitorización de datos en tiempo real
- Control de elementos en tiempo real a través de Internet (controlar luces, motores, leer sensores, etc.)
- Conectar proyectos a otros servicios web (meteorología, redes sociales como Twitter, etc.)
- Enlazar proyectos con otros dispositivos con acceso a Internet.

En esta plataforma existen dos elementos fundamentales: *feeds* y *dashboards*.

Feeds

Estos elementos son la pieza principal de la plataforma. Contienen tanto los datos recibidos como los metadatos acerca de éstos, como la fecha y hora a la que llegaron los datos o las coordenadas GPS de donde vinieron.

Los *feeds* pueden ser públicos o privados, de manera que es posible compartir la información con diferentes usuarios.

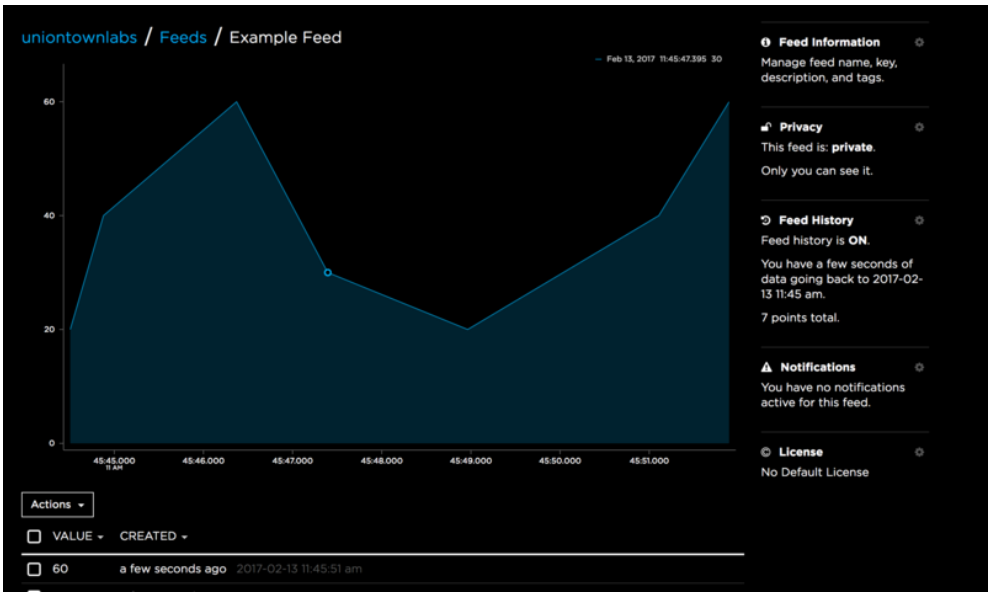


Figura 4.2. Ejemplo de *feed*.
Fuente: https://learn.adafruit.com/welcome-to-adafruit-io/getting-started-with-adafruit-io

En este proyecto se han utilizado 5 *feeds*: uno para mostrar el resumen de datos con la fecha y hora a la que fueron tomados (*datos_sensores*), y cuatro adicionales para visualizar el nivel de batería, la temperatura, la humedad relativa y la luminosidad de forma individual (*bat*, *temp*, *hum* y *lum*, respectivamente).

Dashboards

Actúan de interfaz, permitiendo visualizar los datos recibidos en los *feeds* y controlar el dispositivo conectado a la plataforma. Disponen de funcionalidades como representaciones gráficas, interruptores, botones, barras de deslizamiento, etc.

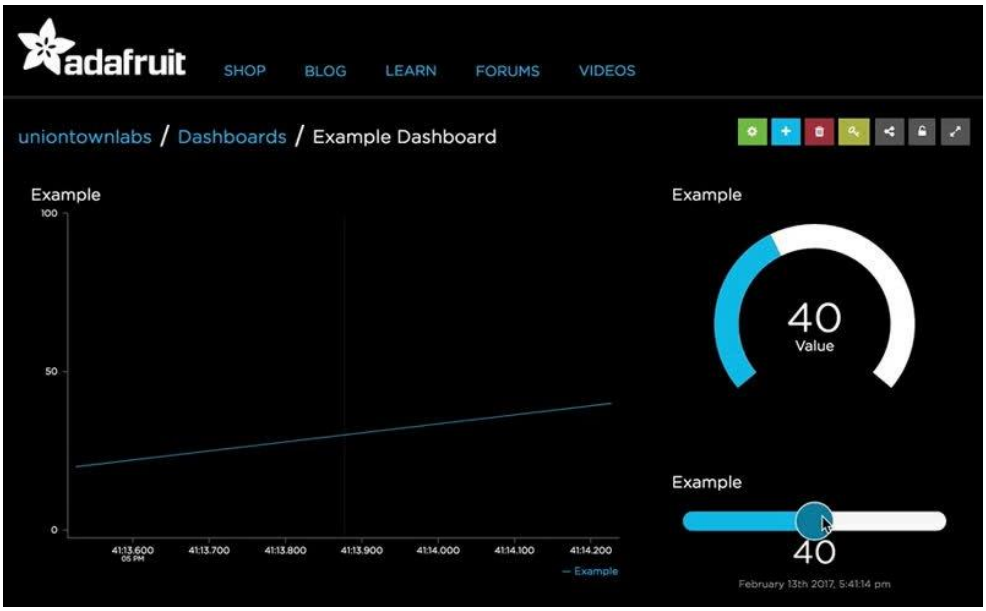


Figura 4.3. Ejemplo de *dashboard*.
Fuente: https://learn.adafruit.com/welcome-to-adafruit-io/getting-started-with-adafruit-io

En esta plataforma, el protocolo de transferencia de datos utilizado es *MQTT*, el cual se explicará en el próximo capítulo.

4.4. *MIT App Inventor* [12]

Para la aplicación móvil se ha utilizado *App Inventor*, un entorno de desarrollo *software* creado por *Google Labs* para la creación de aplicaciones destinadas al sistema operativo *Android*, aunque recientemente también existe una versión para *iOS*.

Este sistema es gratuito, y sus aplicaciones se caracterizan por su simplicidad, sin embargo, son capaces de cubrir un gran número de necesidades básicas en un dispositivo móvil. En este caso, ha permitido realizar la conexión entre el *Wipy* y un móvil mediante *Bluetooth Low Energy*, protocolo que se estudiará más adelante y que ha hecho posible la transmisión de la información acerca de las redes *Wi-Fi* a la que ha de conectarse, los datos del usuario de la *Adafruit IO* y el tiempo de muestreo entre cada medición por parte de los sensores.

Cabe destacar que utiliza un lenguaje de programación visual, de manera que el código se manipula de forma gráfica mediante la unión de los diferentes bloques disponibles en el sistema, y que el compilador traduce este lenguaje visual para las aplicaciones utilizando *Kawa*, lenguaje de programación distribuido por *Linux*.

Posee dos tipos de interfaz: Una está dedicada al diseño de la aplicación (*Designer*), mientras que la otra recoge los distintos bloques que contienen la lógica del programa (*Blocks*). A continuación se muestran ejemplos de ambas interfaces:

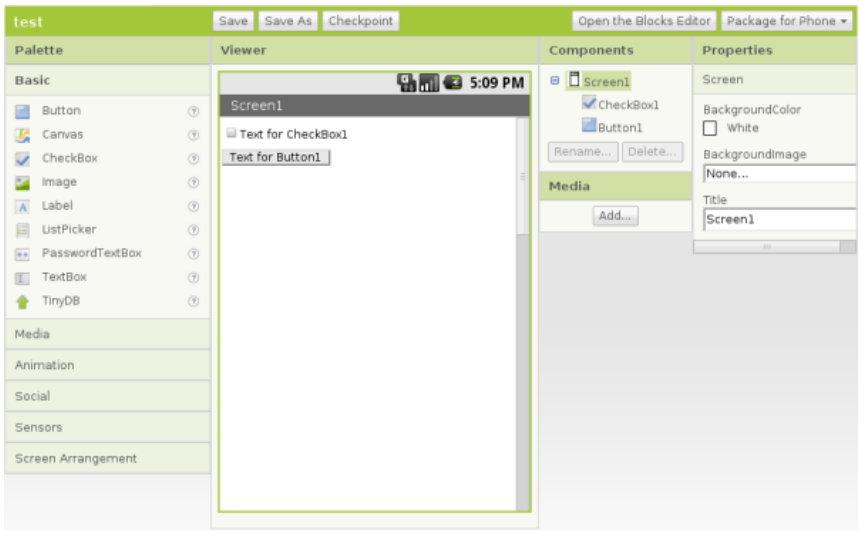


Figura 4.4. Interfaz de diseño de *App Inventor*.

Fuente: https://es.wikipedia.org/wiki/App_Inventor

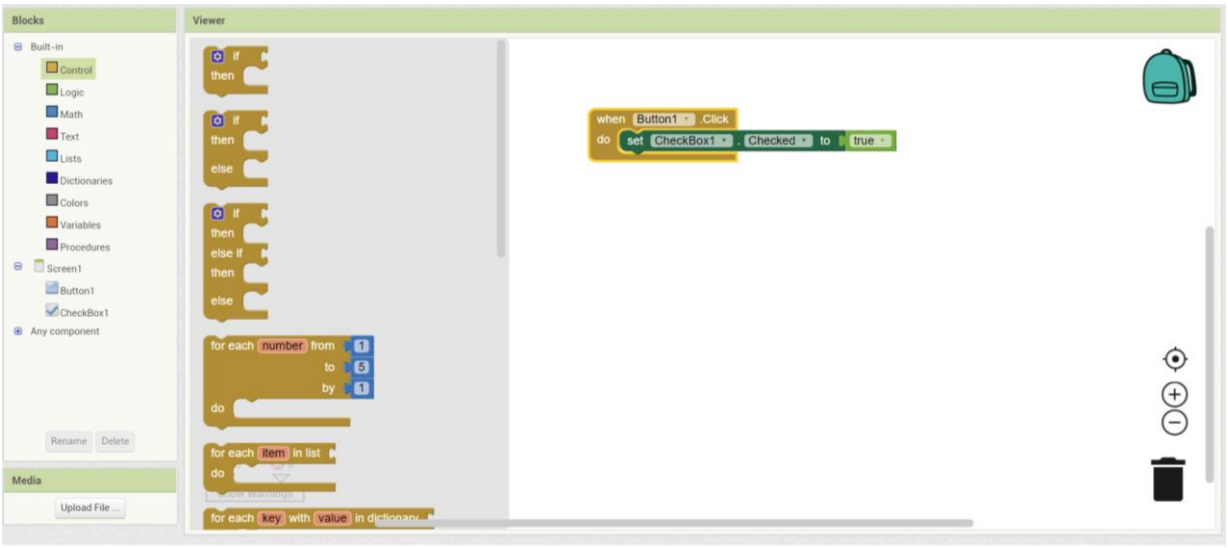


Figura 4.5. Ejemplo de bloques para la programación de *App Inventor*.

Capítulo 5. Protocolo *MQTT*

5.1. Introducción [13] [14]

MQTT (*Message Queuing Telemetry Transport*) es un protocolo de transporte de datos Cliente/Servidor.

Fue inventado en 1999 por Andy Standford-Clark (*IBM*) y Arlen Nippler (actual *Cirrus Link*) debido a la necesidad de un consumo mínimo de batería y de ancho de banda para conectarse a oleoductos vía satélite. Actualmente, este protocolo está estandarizado por *OASIS*, una organización promotora de estándares.

Es considerado como un protocolo ligero y eficiente en cuanto al bando de ancha debido a la estructura de sus paquetes. El mensaje, una vez codificado, tiene la siguiente estructura:

- **Encabezado de paquetes (*header*):** Se trata de un conjunto de bytes que contienen información acerca del mensaje (origen, destino, tipo de instrucción, etc.). Cada mensaje tiene 2 bytes fijos, pero puede tener un segundo encabezado variable con n bytes.

- **Carga útil (*payload*):** Se trata del conjunto de bytes en los que se encuentra el mensaje almacenado, excluyendo cualquier tipo de metadato presente en el encabezado. En este protocolo, la carga útil está limitada a 256 *bytes* de información.

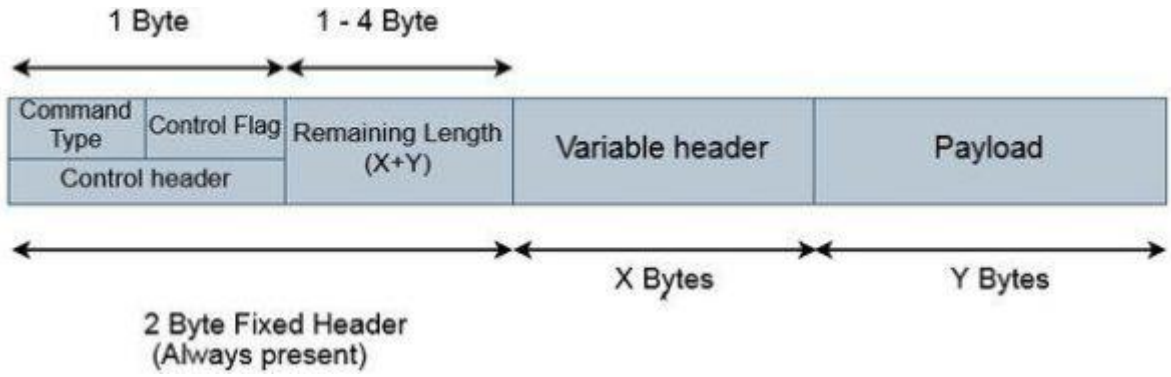


Figura 5.1. Codificación de datos en *MQTT*.

Fuente: https://openlabpro.com/guide/mqtt-packet-format/

Otras de los requisitos que exigían los inventores era que fuese un protocolo sencillo, de fácil implementación y que tuviese un algún tipo de control en la garantía a la hora de entregar el mensaje. Este último requisito es especialmente importante y se hablará más de él en los próximos apartados.

Todas estas prestaciones hacen que el protocolo *MQTT* sea ideal en muchas situaciones, como pueden ser entornos restringidos, comunicaciones Máquina a Máquina (*M2M*) e *Internet de las Cosas*, aplicaciones en las que se requiere que los mensajes no tengan un tamaño muy elevado y en los que el ancho de banda es un bien escaso o existen ciertas dificultades a la hora de conectarse a Internet.

5.2. Cliente, servidor y conexión. [15][16] [17]

Cliente

Un cliente *MQTT* es cualquier dispositivo (desde un microcontrolador hasta un servidor) que ejecuta una librería *MQTT* y se conecta a un servidor o bróker de este tipo a través de una red. Básicamente, cualquier dispositivo que pueda utilizar una conexión TCP/IP es considerado un cliente.

De entre todos los clientes, hemos de diferenciarlos en dos tipos: los *publishers* y los *subscribers*. Los primeros son los clientes encargados de enviar el mensaje, mientras que los segundos son los que reciben el mensaje. En nuestro caso, el *Wipy* sería el *publisher*, mientras que el ordenador o el móvil en el que vemos los datos recibidos sería el *subscriber*.

La conexión entre ellos es manejada por el bróker, que es el encargado de filtrar los mensajes recibidos y distribuirlos correctamente al segundo tipo de clientes. Tanto como el servidor como el tipo de conexión se verán a continuación con más detalle.

Servidor

Por otro lado, el servidor o bróker es la contraparte del cliente. Si el cliente es el encargado de o enviar los mensajes o recibirlos, el servidor es el responsable de recibir y filtrar todos estos mensajes, determinar que cliente está suscrito a cada mensaje y mandar los mensajes a dichos clientes. Dependiendo de la implementación, un bróker puede llegar a encargarse de millones de clientes.

El bróker es por lo tanto un eje central del protocolo que además ha de encargarse de la identificación y la autorización de los clientes.

Conexión

Como se ha mencionado, *MQTT* basa su conexión en el modelo TCP/IP, tanto el cliente como el servidor necesitan soportar este protocolo de red.

El modelo TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet) es un protocolo encargado del enlace de datos y que se utiliza en Internet. Posee cuatro capas, en las que están divididas las tareas de comunicación. De forma resumida, estas son las funciones de cada capa:

- **Capa de aplicación:** Es la capa que está más cerca del usuario y es la encargada de interactuar con el *software* para implementar una comunicación.
- **Capa de transporte:** Determina cuántos datos se deben enviar, dónde y a qué velocidad. Ayuda a garantizar la entrega de datos sin errores.
- **Capa de Internet:** También conocida como capa de red, es la responsable de la transmisión de datos a través de Internet, como su nombre indica, utilizando la ruta óptima.
- **Capa de acceso a la red:** Responsable de definir los detalles de cómo enviar los paquetes a través de la red. Se encarga de la asignación de las direcciones IP, que son una combinación de números que identifican la interfaz de red de un dispositivo que utilice Internet.

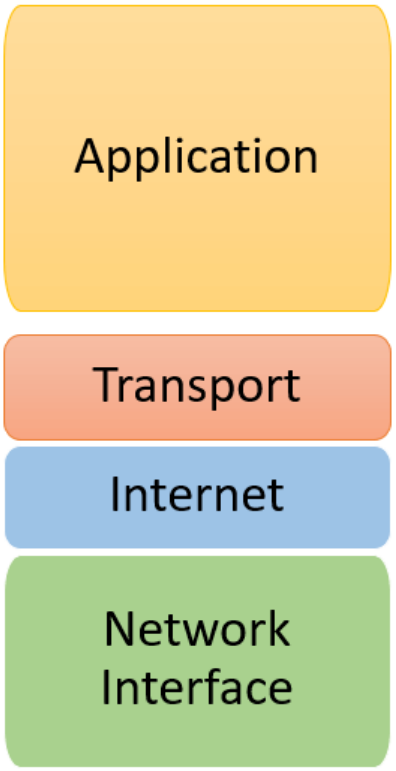


Figura 5.2. Capas conceptuales del protocolo TCP/IP
Fuente: guru99.com

La conexión *MQTT* siempre se realiza entre el bróker y un cliente, de manera que no hay interacción directa entre clientes. Para iniciar la conexión, el cliente le envía un mensaje *CONNECT* (solicitud de conexión al servidor) al bróker, y éste responde con un mensaje *CONNACK* (*Connection Acknowledged*, mensaje recibido).

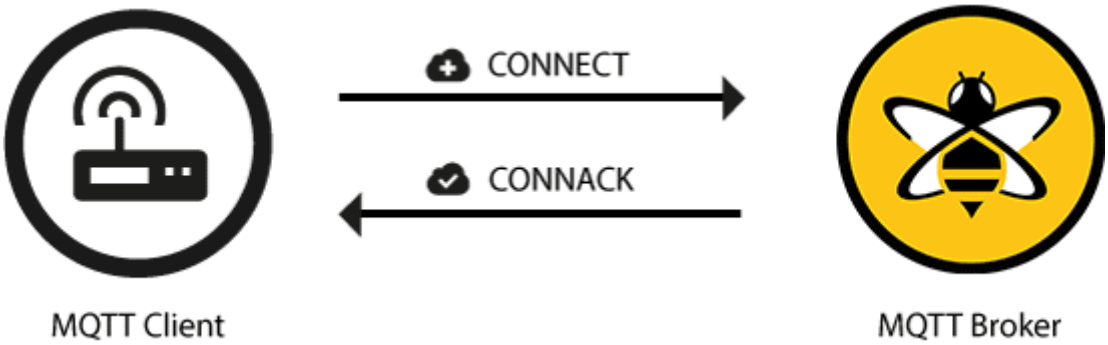


Figura 5.3. Inicio de la conexión MQTT entre cliente y servidor.
Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/

El contenido del mensaje *CONNECT* puede variar según el servidor, pero debe de aportar al servidor la información para la autorización del cliente (usuario y contraseña) y del tipo y calidad de conexión a establecer, que se verá más adelante en la "Calidad del servicio". Si el mensaje no está correctamente estructurado o pasa mucho tiempo desde que se abre la conexión y se envía este mensaje, el bróker cierra la conexión para evitar clientes "malintencionados".

El mensaje *CONNACK*, por otro lado, contiene dos parámetros: un primer elemento booleano, que indica si ya existía o no previamente una conexión entre dicho cliente y el bróker, y un segundo parámetro con un valor que indica el estado de la conexión. En esta última variable, el bróker devuelve un 0 si la conexión es aceptada, mientras que devuelve un valor entre 1 y 5 si la conexión se rechaza.

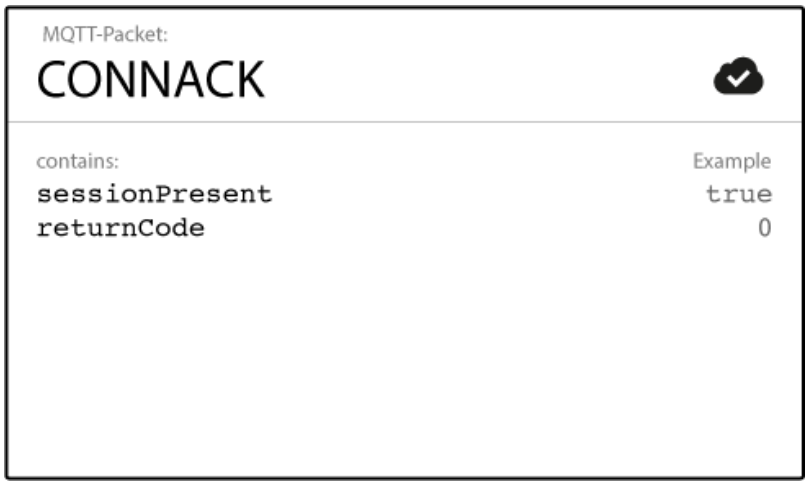


Figura 5.4. Contenido del mensaje *CONNACK*.
Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/

Return Code	Return Code Response
0	Connection accepted
1	Connection refused, unacceptable protocol version
2	Connection refused, identifier rejected
3	Connection refused, server unavailable
4	Connection refused, bad user name or password
5	Connection refused, not authorized

Figura 5.5. Código del estado de conexión en el mensaje *CONNACK*.
Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/

Los detalles de conexión de *Adafruit IO* son los siguientes:

- **Host:** io.adafruit.com.
- **Puerto:** 8883 o 1883.
- **Usuario:** Nombre del usuario en AIO.
- **Contraseña:** Clave AIO asociada para cada usuario.

5.3. Publicación y suscripción [18]

El mecanismo de publicación y suscripción (también conocido como pub/sub) proporciona una alternativa a la arquitectura tradicional de cliente-servidor. En esta última, el cliente se comunica directamente con el destinatario final, mientras que en la publicación y suscripción del protocolo *MQTT*, el cliente que envía el mensaje está desacoplado con el que lo recibe, estos dos clientes nunca están en contacto directo, sino que, como se ha dicho previamente, existe un tercer componente que se encarga de la conexión entre ellos, esto es, el servidor o bróker.

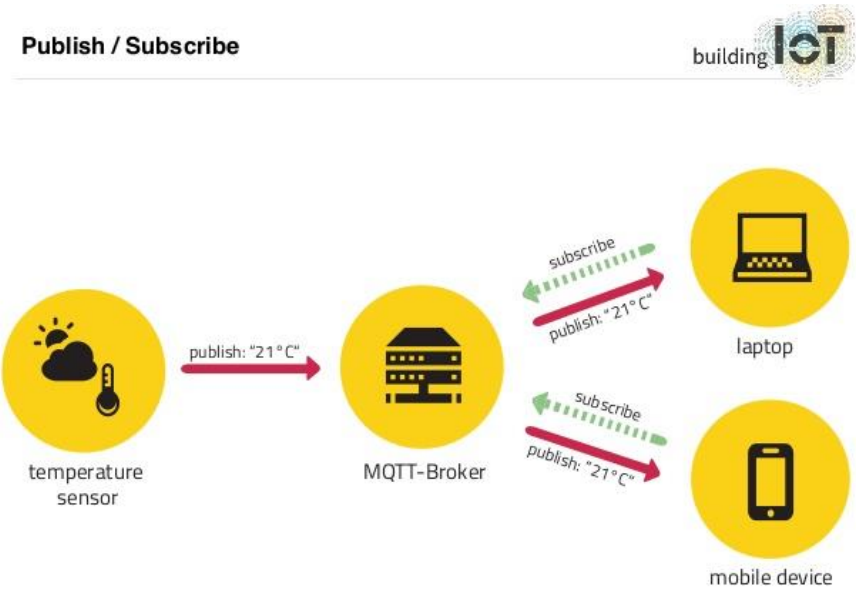


Figura 5.6. Esquema del proceso de publicación y suscripción.
Fuente: https://www.slideshare.net/dobermai/securing-mqtt-buildingiot-2016-slides

En el ejemplo de la Figura 5.6 podemos ver este mecanismo: en un extremo tenemos un dispositivo que publica los datos de un sensor de temperatura, estos datos, que son recibidos por el servidor, son enviados a los clientes que estaban suscritos. Ni el dispositivo ni el ordenador o móvil estaban conectados directamente, pero gracias a la acción del bróker se ha producido el transporte de información de un cliente al otro.

Claramente, el bróker juega un papel fundamental en el mecanismo de pub/sub. ¿Pero cómo puede ser capaz de enviar a cada cliente solamente los mensajes de interés? La respuesta a esto está en el mecanismo de filtrado de los mensajes.

Existen varias formas de filtrar los mensajes, pero la que más nos interesa, y con la que trabaja el servidor de *Adafruit IO*, es mediante el filtrado basado en temas.

Filtrado basado en temas

Este mecanismo se basa en el tema (*topic*) de cada mensaje. En *MQTT*, un tema es identificador compuesto por una cadena de texto compuesta por varios niveles y que asignan a que cliente han de ser mandados.

Tomando como referencia la plataforma utilizada, como hemos nombrado la *feed* en la que queremos almacenar los datos de la temperatura medidos como *temp,* el tema de dicha variable será `usuario/feeds/temp`. De esta forma, de los datos enviados por el microcontrolador, solo aquellos que tengan este identificador serán recibidos por *temp*.

A continuación, se muestra una explicación mas detallada de cada mecanismo de forma individual.

Publicación

Un cliente *MQTT* puede publicar mensajes tan pronto como se conecte con el servidor. Como hemos explicado previamente, cada mensaje publicado debe de contener un tema para que pueda ser reenviado a los clientes interesados.

Por otro lado, el mensaje tiene una carga útil que contiene la información a transmitir. En cuanto a la codificación del mensaje, *MQTT* independiente de los datos, el cliente determina como se estructura la carga útil: el mensaje pueden ser datos binarios, datos de texto o incluso *JSON*.

Un mensaje de este tipo tiene una serie de atributos, de los que cabe destacar los siguientes:

- **Nombre del tema:** Permite enviar el mensaje solamente a los clientes interesados.
- **QoS:** Indica la “Calidad del Servicio” del mensaje, esto indica el tipo de garantía que tiene el mensaje de llegar a su destinatario. Se trata de un aspecto bastante importante que se explicará en profundidad más adelante.
- **Carga útil:** Contenido real del mensaje, independiente del tipo de dato.

En la publicación, el cliente que envía ese mensaje solo se preocupa por entregarlo al servidor, siendo la responsabilidad de este último que el mensaje llegue a todos los suscriptores interesados.

Suscripción

La publicación de un mensaje no tiene ningún sentido si no hay ningún cliente que lo reciba, por eso es necesaria la suscripción. Para recibir mensajes de interés, el cliente manda un mensaje de suscripción (*SUBSCRIBE*) al bróker. Este mensaje es muy sencillo, y solamente está compuesto por un identificador, los temas a los que suscribirse y la Calidad del Servicio que quiere para cada tema.

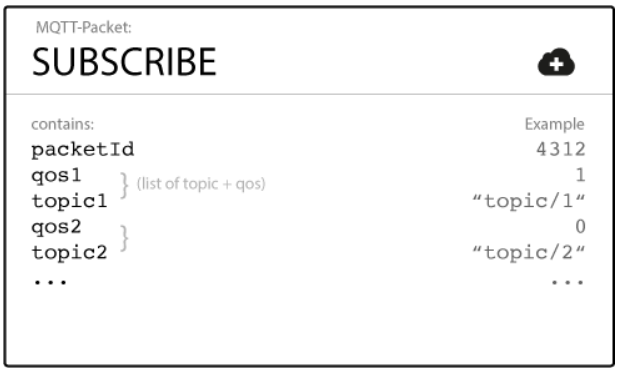


Figura 5.7. Contenido del mensaje *SUBSCRIBE*.

Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/

Para confirmar cada suscripción, el servidor envía un mensaje *SUBACK* (*Subscription Acknowledged*, suscripción recibida) para informar de la recepción del mensaje enviado por el cliente. Este mensaje contiene un código por cada suscripción pedida por el cliente, en el que se indica si se ha podido proporcionar el respectivo *QoS* de cada tema.

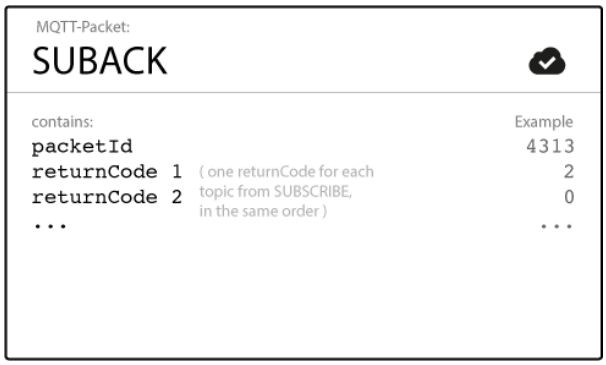


Figura 5.8. Contenido del mensaje *SUBACK*.

Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/

Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure

Figura 5.9. Código devuelto en el mensaje *SUBACK*.

Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/

Una vez el cliente recibe este mensaje, también recibe todos los mensajes publicados a los temas a los que se ha suscrito.

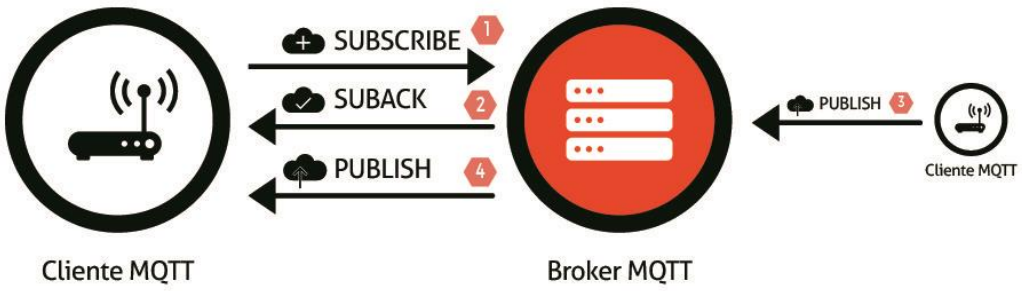


Figura 5.10. Recepción de datos por parte del suscriptor.
Fuente: https://www.factor.mx/portal/base-de-conocimiento/mqtt/

De igual forma, un cliente puede cancelar su suscripción a un tema determinado. El proceso es idéntico al de suscripción, cambiando los mensajes por *UNSUBSCRIBE y UNSUBACK*.

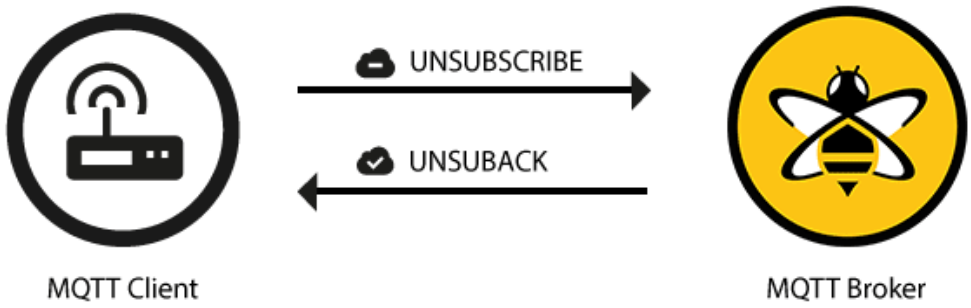


Figura 5.11. Cancelación de una suscripción.
Fuente: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/

5.4. Calidad del Servicio [19]

La Calidad del Servicio o *Quality of Service* (*QoS*) es un grado de acuerdo entre el cliente encargado de enviar el mensaje y el que lo recibe. Se trata de la garantía respecto a la entrega de cada mensaje específico. En *MQTT*, hay tres niveles de *QoS*:

- Como mucho una vez (0)
- Al menos una vez (1)
- Exactamente una vez (2)

Cuando se habla de *QoS*, es importante tener en cuenta que afecta a los dos tipos de cliente de forma individual. En una primera instancia, el cliente que publica el mensaje define un nivel de *QoS* cuando le envía dicho mensaje al bróker. Por otro lado, el servidor le transmite el mensaje al suscriptor con el nivel de *QoS* seleccionado por este segundo cliente, independientemente del seleccionado por el primero. Si el suscriptor selecciona un nivel menor, el mensaje será enviado con un nivel menor.

La Calidad del Servicio es un aspecto clave del protocolo *MQTT*, ya que otorga a los clientes la posibilidad de seleccionar un nivel acorde con las restricciones y la fiabilidad de sus redes.

QoS 0 - Como mucho una vez.

Este nivel de servicio es el mínimo, y garantiza una entrega como mucho, por lo que realmente no existe ninguna garantía de entrega. Ni el destinatario confirma la recepción del mensaje ni el remitente retransmite dicho mensaje.

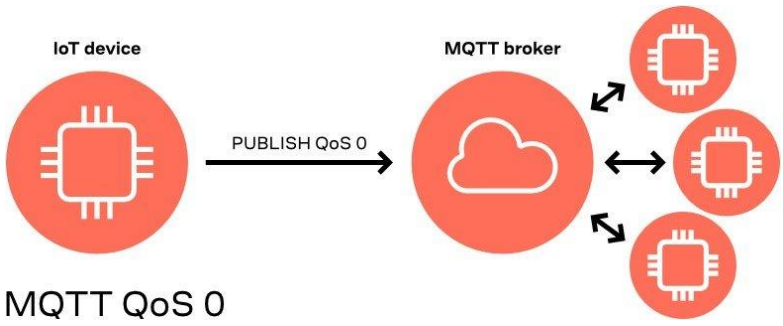


Figura 5.12. ***Quality of Service*** **0.**
Fuente: https://www.u-blox.com/en/blogs/insights/mqtt-beginners-guide

QoS 1 – Al menos una vez.

Una Calidad del Servicio de nivel 1 garantiza que el mensaje es recibido al menos una vez. El cliente que envía el mensaje lo guarda hasta que recibe un mensaje *PUBACK (Publishing Acknowledged*, publicación recibida) que confirma que el servidor ha recibido el mensaje. Sin embargo, es posible que el mensaje sea enviado más de una vez si el *PUBACK* no se ha recibido tras un tiempo determinado.

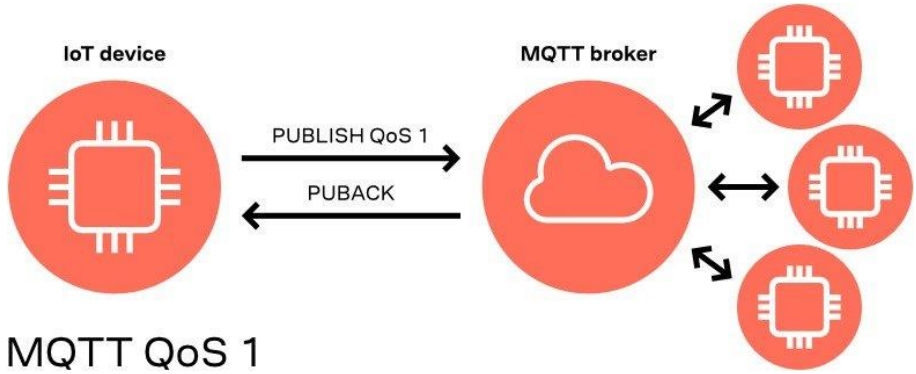


Figura 5.13. ***Quality of Service*** **1.**
Fuente: https://www.u-blox.com/en/blogs/insights/mqtt-beginners-guide

QoS 2 – Exactamente una vez.

Este es el nivel más alto de *QoS*, en el que se garantiza que el mensaje solo es recibido una vez. Es, por lo tanto, el nivel más seguro, pero también el más lento, ya que esta garantía se proporciona mediante el uso de al menos dos ciclos de solicitud/respuesta (*four way handshake*) entre el emisor y el receptor.

Cuando el servidor recibe una solicitud de un paquete con un *QoS* de nivel 2, éste procesa la publicación como siempre y envía al emisor un mensaje *PUBREC* que confirma dicha publicación. Si el cliente que publica el mensaje no recibe el *PUBREC*, vuelve a publicar el mensaje con una bandera que indica que es un mensaje duplicado.

Una vez que el emisor recibe el mensaje *PUBREC*, descarta el mensaje enviado, almacena el *PUBREC* y responde con un mensaje *PUBREL*. Cuando el servidor recibe este segundo mensaje, descarta cualquier estado intermedio guardado y responde con un mensaje *PUBCOMP*, que indica que la publicación se ha completado. Antes de realizar este último paso, el servidor almacena un estado intermedio con la información del mensaje a publicar para evitar que el mensaje sea procesado una segunda vez.

Finalmente, una vez que el emisor recibe el mensaje *PUBCOMP*, ambas partes están seguras de que el mensaje solo se ha enviado una vez. Si en algún caso se perdiese el paquete a lo largo de la transmisión, el emisor es el responsable de volver a enviarlo en un intervalo de tiempo razonable.

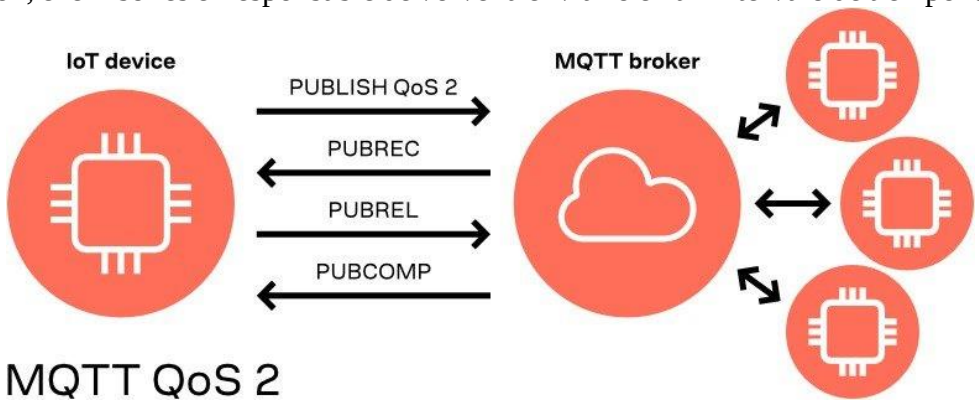


Figura 5.14. *Quality of Service* 2.

Fuente: https://www.u-blox.com/en/blogs/insights/mqtt-beginners-guide

Cabe destacar que *Adafruit IO* soporta una Calidad del Servicio de nivel 0 o 1. En este proyecto se utilizará de nivel 1, pues nos interesa que todos los mensajes lleguen al menos una vez y dado que en algunas situaciones la conexión puede ser algo desfavorable.

Capítulo 6. *Bluetooth Low Energy*

6.1. Introducción [20]

Bluetooth Low Energy (*BLE*) o *Bluetooth Smart*, es una tecnología de red desarrollada por *Bluetooth Special Interest Group* diseñada para proporcionar un bajo consumo de energía y unos costes considerablemente reducidos, mientras que mantiene un alcance similar al *Bluetooth* clásico.

Es un protocolo inalámbrico especialmente interesante debido a la gran cantidad de plataformas compatibles con esta tecnología. Los sistemas operativos móviles *iOS*, *Android*, *Windows Phone* y *BlackBerry*, así como *macOS*, *Linux*, *Windows* 8 y *Windows 10*, son compatibles con *Bluetooth Low Energy*.

Al igual que el clásico, *Bluetooth Low Energy* opera en la banda de frecuencia de los 2.4 GHz. Sin embargo, un hecho muy importante que los diferencia es que *BLE* permanece en modo *sleep* constantemente excepto cuando una conexión es iniciada. Además, los tiempos de conexión son de escasos milisegundos, mientras que en *Bluetooth* están alrededor de 100 milisegundos, aunque esto también es debido a que la tecnología clásica puede procesar más peticiones por segundo.

Specifications	Classic Bluetooth	Bluetooth Low Energy (BLE)
Range	100 m	Greater than 100 m
Data Rate	1-3 Mbps	1 Mbps
Application Throughput	0.7 -2.1 Mbps	0.27 Mbps
Frequency	2.4 GHz	2.4 GHz
Security	56/128-bit	128-bit AES with Counter Mode CBC-MAC
Robustness	Adaptive fast frequency hopping, FEC, fast ASK	24-bit CRC, 32-bit Message Integrity Check
Latency	100 ms	6 ms
Time Lag	100 ms	3 ms
Voice Capable	Yes	No
Network Topology	Star	Star
Power Consumption	1 W	0.01 to 0.5 W
Peak Current Consumption	less than 30 mA	less than 15 mA

Figura 6.1. Comparativa entre *Bluetooth* clásico y *Bluetooth Low Energy*.
Fuente: https://iotlab.tertiumcloud.com/2020/08/19/classic-bluetooth-vs-bluetooth-low-energy-ble/

Se trata, por lo tanto, de una tecnología excelente en aplicaciones en las que no sea necesario el manejo de una cantidad excesiva de datos y que requieran de un ahorro considerable de energía. Estas características convierten al *Bluetooth Low Energy* en un protocolo ideal para el uso en proyectos relacionados con el Internet de las Cosas.

6.2. *GAP* [21]

GAP son las siglas de *Generic Access Profile* (Perfil de Acceso Genérico), y es un estándar de comunicación que controla la conexión entre los dispositivos. Permite que un dispositivo sea visible y controla como pueden o no interactuar entre ellos.

Este estándar define varios roles para los dispositivos, existiendo dos conceptos clave:

- **Dispositivo periférico**: Son dispositivos secundarios, ligeros, con bajo consumo de energía y recursos limitados. Éstos se conectan a otros dispositivos mucho más potentes, denominados centrales.
- **Dispositivo central:** Generalmente poseen una mayor capacidad de procesamiento y memoria, suelen ser dispositivos móviles o *tablets*.

Para establecer la conexión entre dispositivos, se realiza generalmente mediante un conjunto de datos denominado *Advertising Data Payload*, un anuncio mediante el cual las diferentes centrales saben que el dispositivo que transmite dicho mensaje existe. Existen otros datos opcionales, denominados *Scan Response Payload*, que pueden pedir los dispositivos centrales y que se utilizan para que los diseñadores puedan enviar un mayor número de información, como el nombre del dispositivo, etc.

Ambos paquetes pueden contener hasta 31 bytes de información, pero solo el primero es obligatorio.

La siguiente figura muestra un ejemplo del proceso:

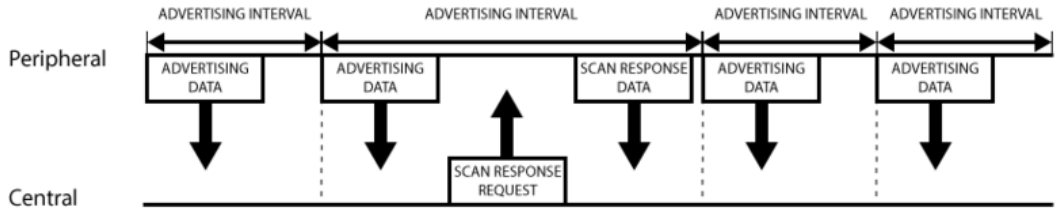


Figura 6.2. Proceso de *advertising* en *Bluetooth Low Energy*.
Fuente: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap

Un dispositivo periférico fija un intervalo de tiempo (*Advertising Interval*) en el que envía los datos de anuncio y se muestra como disponible para realizar una conexión. Cada vez que transcurre este intervalo, vuelve a transmitir el mensaje. Los dispositivos que se encuentren dentro del alcance máximo pueden realizar una conexión con dicho periférico y el paquete principal de la conexión. Si se requiere y el periférico tiene el segundo tipo de paquete, también pueden pedir la información adicional (*Scan Response Request*).

Mediante este proceso de *advertising* los periféricos transmiten este mensaje de anuncio para que se establezca una conexión y sea posible utilizar las Características y Servicios del protocolo *GATT*, concepto importantísimo que se verá en el próximo apartado, que permiten un intercambio de datos mucho mayor y bidireccional.

Es muy importante tener en cuenta que las conexiones son exclusivas, es decir, cada dispositivo periférico solamente puede estar conectado a un dispositivo central. Tan pronto como un periférico se haya conectado con otro dispositivo dejará de anunciarse y no será visible hasta que esta conexión se rompa, sin embargo, un dispositivo central si que puede estar conectado con varios periféricos.

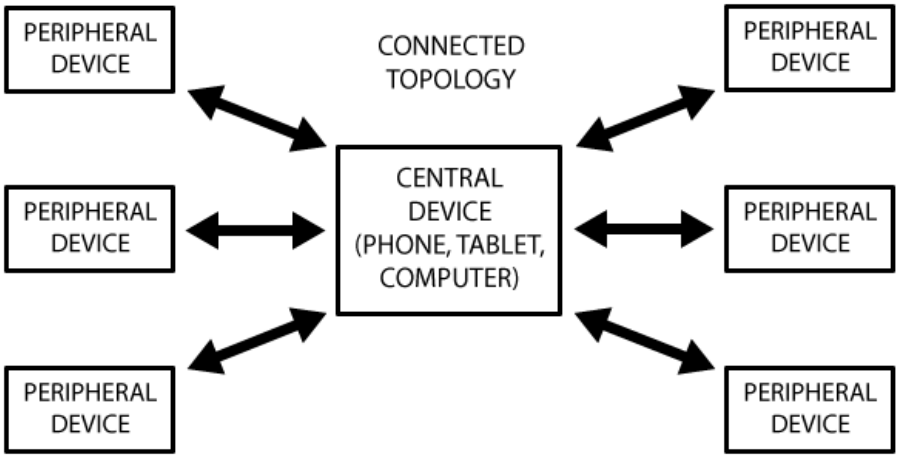


Figura 6.3. Esquema de conexión entre dispositivo central y periféricos.
Fuente: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap

6.3. *GATT* [22][23]

GATT son las siglas de *Generic Attribute Profile* (Perfil de Atributo Genérico), se trata posiblemente del concepto más importante de *Bluetooth Low Energy*, ya que establece la metodología de transferencia de datos entre dos dispositivos a través de los Servicios y las Características, que se explicarán a continuación.

Este protocolo entra en juego una vez que la conexión entre dispositivos ha sido establecida, es decir, cuando se ha pasado por el proceso de *advertising* gobernado por el *GAP*.

GATT dispone de un protocolo, denominado *Attribute Protocol* (*ATT*), encargado del intercambio de datos entre dispositivos. Para ello, se define el concepto de Atributos, que son las unidades de información más pequeñas y contienen datos o metadatos relacionados con la estructura o la agrupación de los diferentes servicios y características existentes, todos estos elementos se explicarán a continuación

Otro concepto importante del *GATT* es la relación entre cliente y servidor. El dispositivo periférico es el servidor *GATT*, en el que están contenidos los datos de los servicios y las características y los datos del protocolo *ATT*, mientras que el cliente *GATT*, que es el dispositivo central, manda peticiones al servidor.

Una vez establecida la conexión, el servidor sugiere un intervalo de conexión y el cliente intentará conectarse cada vez que transcurra dicho intervalo para ver si hay nueva información disponible. A continuación, se muestra un ejemplo de este proceso:

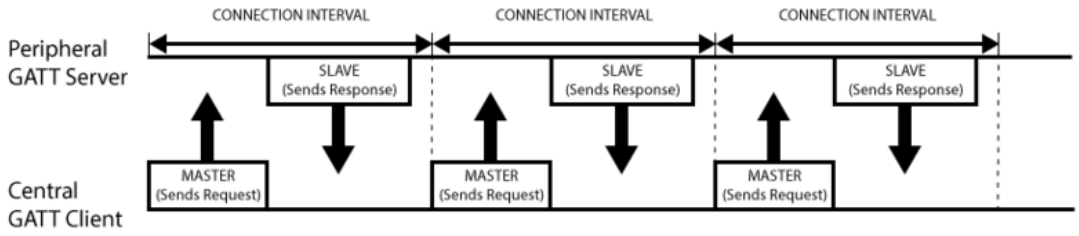


Figura 6.4. Intercambio de datos entre cliente y servidor *GATT*.
Fuente: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt

Sin embargo, es posible que esta interacción no se cumpla en cada intervalo de conexión, ya que el dispositivo central puede estar ocupado intercambiando información con otro periférico o simplemente el servicio no está disponible en ese momento.

Como ya hemos mencionado, la transmisión de datos mediante *GATT* se realiza a través de elementos denominados servicios y características, la estructura jerárquica de estos elementos se puede observar en la siguiente figura:

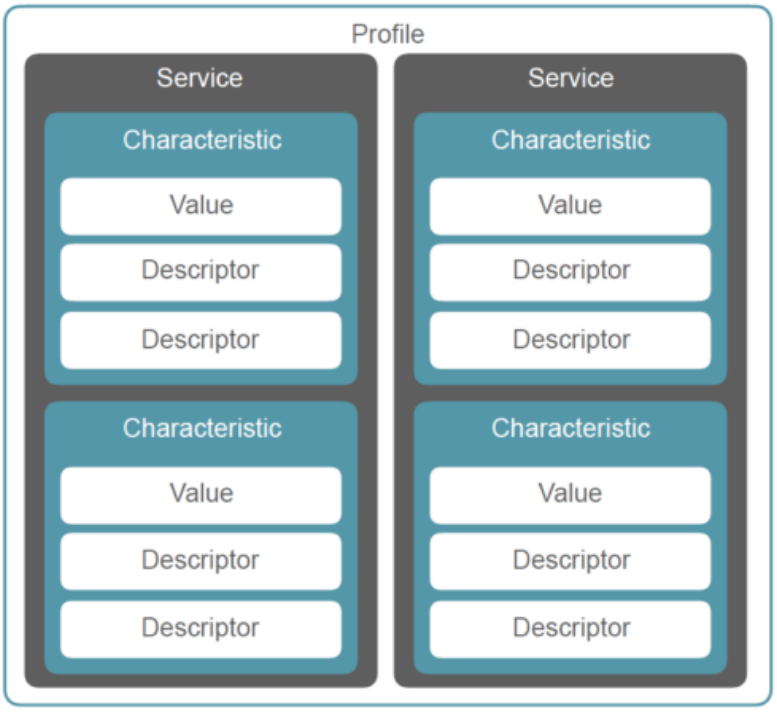


Figura 6.5. Perfiles, Servicios y Características de *GATT*.
Fuente: https://developer.bosch.com/web/xdk/bluetooth-low-energy-ble-1

Antes de definir los servicios y características, es necesario explicar qué son los atributos:

Atributos

Como se ha explicado anteriormente, los atributos son la unidad de información básica del protocolo conforman las Características y los Servicios.

Se puede pensar que el conjunto de atributos de un servidor conforma una tabla, en la que cada fila es un atributo y cada columna es una parte de éste:

Handle	Type	Permissions	Value	Value length
0x0201	$UUID_1$ (16-bit)	Read only, no security	`0x180A`	2
0x0202	$UUID_2$ (16-bit)	Read only, no security	`0x2A29`	2
0x0215	$UUID_3$ (16-bit)	Read/write, authorization required	"a readable UTF-8 string"	23
0x030C	$UUID_4$ (128-bit)	Write only, no security	`{0xFF, 0xFF, 0x00, 0x00}`	4
0x030D	$UUID_5$ (128-bit)	Read/write, authenticated encryption required	`36.43`	8
0x031A	$UUID_1$ (16-bit)	Read only, no security	`0x1801`	2

Figura 6.6. Tabla de Atributos.
Fuente: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html

Podemos observar que un atributo está compuesto por una serie de identificadores (*Handle* y *Type*), los permisos de cada uno de ellos y su valor.

- ***Handle*:** Se trata de un identificador de 16 bits que se utiliza como referencia y permite dirigirse a un Atributo en concreto.
- ***Type:*** Se trata de un UUID. Un UUID o Identificador Único Universal está compuesto por 16 bits o 128 bits y se expresa mediante 32 dígitos hexadecimales, divididos en cinco grupos y separados mediante guiones. En este caso se utiliza para indicar el tipo de datos presente.
- **Permissions:** Son metadatos qué operaciones (lectura, escritura, o ambas) se pueden ejecutar en cada Atributo en particular y con qué requisitos de seguridad (autorización o cifrado requeridos).
- ***Value*:** En esta parte se encuentra el valor de los datos contenidos en el Atributo.

Servicios

Los servicios se utilizan para dividir los datos en entidades lógicas y contienen fragmentos específicos de datos denominados características. Conceptualmente, se puede pensar en un servicio como una clase de un lenguaje que soporte programación orientada a objetos. Un perfil, por otro lado, sería una aplicación que usa un conjunto de clases.

Para definir un servicio, se utiliza un atributo del siguiente tipo:

Handle	Type	Permissions	Value	Value length
`0xNNNN`	$UUID_{primary\ service}$ or $UUID_{secondary\ service}$	Read Only	Service UUID	2, 4, or 16 bytes

Figura 6.7. Atributo para la definición de un Servicio
Fuente: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html#gatt_uuids

Podemos distinguir por lo tanto de servicios primarios y secundarios. Un servicio primario contiene todas las funciones primarias del dispositivo, mientras que un servicio secundario solamente dispone de funciones adicionales, generalmente de poca relevancia. En la práctica, el servicio secundario apenas se usa.

Este tipo de atributos es el primero en declararse a la hora de crear un servicio, seguido de los atributos de las diferentes características.

Características

Se trata del concepto más bajo en la transmisión de datos por *GATT*, y es el elemento que actúa como el contenedor de la información que se quiere enviar. Estos elementos siempre contienen al menos dos atributos: el atributo en el que se declara la característica (*Characteristic declaration*), y otro en el que se guarda la información a enviar (*Characteristic value*).

Handle	Type	Permissions	Value	Value length
`0xNNNN`	$UUID_{characteristic}$	Read only	Properties, value handle (`0xMMMM`), characteristic UUID	5, 7, or 19 bytes
`0xMMMM`	Characteristic UUID	Any	Actual value	Variable

Figura 6.8. Atributo para la definición de una Característica.

Fuente: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html#gatt_uuids

Por otro lado, estas características pueden ser complementadas mediante elementos denominados descriptores, que contienen información adicional de la declaración de la característica (como puede ser una simple descripción, indicando que, por ejemplo, esa característica contiene el valor de la temperatura de una casa).

Si volvemos a utilizar la analogía a la programación orientada a objetos, las características serían las variables que contienen distintas propiedades de dicha clase.

Capítulo 7. Aplicación móvil

A continuación se expone una descripción funcional del programa de la aplicación. El código de ésta está disponible en el Anexo 3.

El diseño de la aplicación es el siguiente:

Figura 7.1. Diseño de la aplicación móvil.

7.1. Escáner y almacén de redes *Wi-Fi*

Lo primero que podemos observar en la aplicación, es un bloque bajo el nombre de INFO REDES. En éste, podemos introducir las redes que queramos enviar al *Wipy*.

Inicialmente, podemos introducir las redes de dos formas: de manera manual, introduciendo un SSID y una contraseña, o escaneando un código QR. Mediante estas dos formas, obtendremos un código codificado de la siguiente forma: `WIFI:S:<SSID>;T:<WPA|WEP|>;P:<contraseña>;;` o `WIFI:T:<WPA|WEP|>;S:<SSID>;P:<password>;;`, siendo el primer formato el más común. Concretamente en el escáner del código QR, si el texto escaneado no tiene ninguno de estos formatos, no será reconocido como una red *Wi-Fi* y aparecerá un mensaje de advertencia.

Este proceso se puede repetir para las redes que se necesiten, que se irán almacenando en una lista. Una vez almacenadas todas las redes deseadas, es posible guardarlas en un archivo de texto en la tarjeta SD mediante el botón "Guardar archivo". Para guardar estos archivos, es necesarios añadirles la terminación .txt una vez que te pida el nombre deseado. Si no se añade dicha terminación, saltará un mensaje para recordarlo.

Por otro lado, también es posible seleccionar uno de estos archivos previamente guardados mediante el botón "Abrir archivo". Si se abre un archivo por error o se ha guardado una red incorrecta, siempre es posible borrar la lista de redes almacenadas mediante "Borrar redes".

7.2. Información de *Adafruit IO*.

El segundo bloque de la aplicación es el relacionado con el servidor *Adafruit IO*. Podemos observar que se nos pide un nombre de usuario, una clave IO y el tiempo de muestreo para el dispositivo.

El nombre de usuario y la clave IO se obtienen al registrarse en dicho servidor. Esto se verá en más profundidad en el siguiente capítulo. El usuario se puede meter manualmente, mientras que la clave IO se debe de introducir mediante un código QR que se consigue en la propia plataforma.

Esta clave tiene un formato muy específico: los tres primeros caracteres son "aio" y la longitud total es de 32 caracteres. Por ello, cualquier código escaneado que no cumpla con estos requisitos no será reconocido como clave IO.

Por último, podemos introducir el tiempo de muestreo en minutos.

7.3. Conexión y envío de datos por *Bluetooth*

El último bloque, y probablemente el de mayor importancia, es el del envío de datos por *Bluetooth*.

En primer lugar, podemos observar dos mensajes que nos indican tanto el estado de conexión como el estado del envío de datos. Estos mensajes, una vez que se haya conectado al dispositivo y se hayan enviado los datos, respectivamente, cambiarán su contenido y se pondrán en color verde.

Para realizar la conexión, es necesario que el microcontrolador abra un punto de acceso y anuncie los servicios y las características correspondientes para las redes y para los datos del servidor *Adafruit IO*, el mecanismo de este punto de acceso se verá en el próximo capítulo, en el que se explicará el código del *Wipy*.

Figura 7.2. Escaneo de dispositivos

Una vez abierto este punto de acceso, si pulsamos el botón izquierdo de escáner de dispositivos disponibles, debería aparecer un dispositivo disponible bajo el nombre de *WIPY*. Si este punto de acceso no se ha abierto, o si hay algún error en el escáner, aparecería un mensaje de aviso indicando que no se ha encontrado ningún dispositivo disponible.

Si se selecciona la opción con el nombre del dispositivo, se realizaría la conexión, que se indicaría mediante un mensaje de “Conexión establecida” en verde. Si tras seleccionar la opción pasan 20 segundos y no ha sido posible conectarse, se cancelará el intento y será necesario volver a escanear dispositivos. Esto se ha realizado con el objetivo de no dejar el programa en un estado muerto.

Figura 7.3. Conexión establecida

Una vez realizada la conexión, se puede enviar toda la información recogida en los bloques anteriores mediante el botón central, que corresponde con el envío de datos. Es importante mencionar que este botón solo funciona si se ha rellenado todos los campos anteriores, si hay algún campo en blanco, aparecerá un mensaje de advertencia indicándolo. Tampoco será posible enviar datos antes de haberse conectado a un dispositivo, ni volver a escanear dispositivos hasta que no se haya desconectado del primero.

La principal limitación de la transmisión de datos mediante *Bluetooth Low Energy* es que generalmente los dispositivos solo aceptan 20 *bytes* como máximo en cada característica, por lo que ha sido necesario tomar una serie de medidas.

Por un lado, las características de usuario y del tiempo de muestreo no tienen ningún problema y se han podido enviar sin realizar ninguna partición.

Por otro lado, tanto cada una de las redes codificadas mediante el formato comentado anteriormente como la clave IO ocupan más de 20 *bytes*, por lo que se ha optado por realizar estas dos acciones:

En primer lugar, para las redes, se ha decodificado de cada una de estas cadenas únicamente su SSID y su contraseña, que se envían a sus correspondientes características. Esto se ha realizado recortando cada una de estas cadenas a partir de los caracteres `;` y `:`, que proporcionaban una lista con posiciones siempre fijas para los respectivos SSID y contraseñas de cada red. Una vez separados estos elementos, el envío no supone ningún problema ya que ni las SSID ni las contraseñas de fábrica, que suelen ser las más largas, tienen una longitud de más de 20 caracteres.

Por otro lado, para la clave IO, se ha decidido partir la cadena por la mitad y enviar cada parte a una característica.

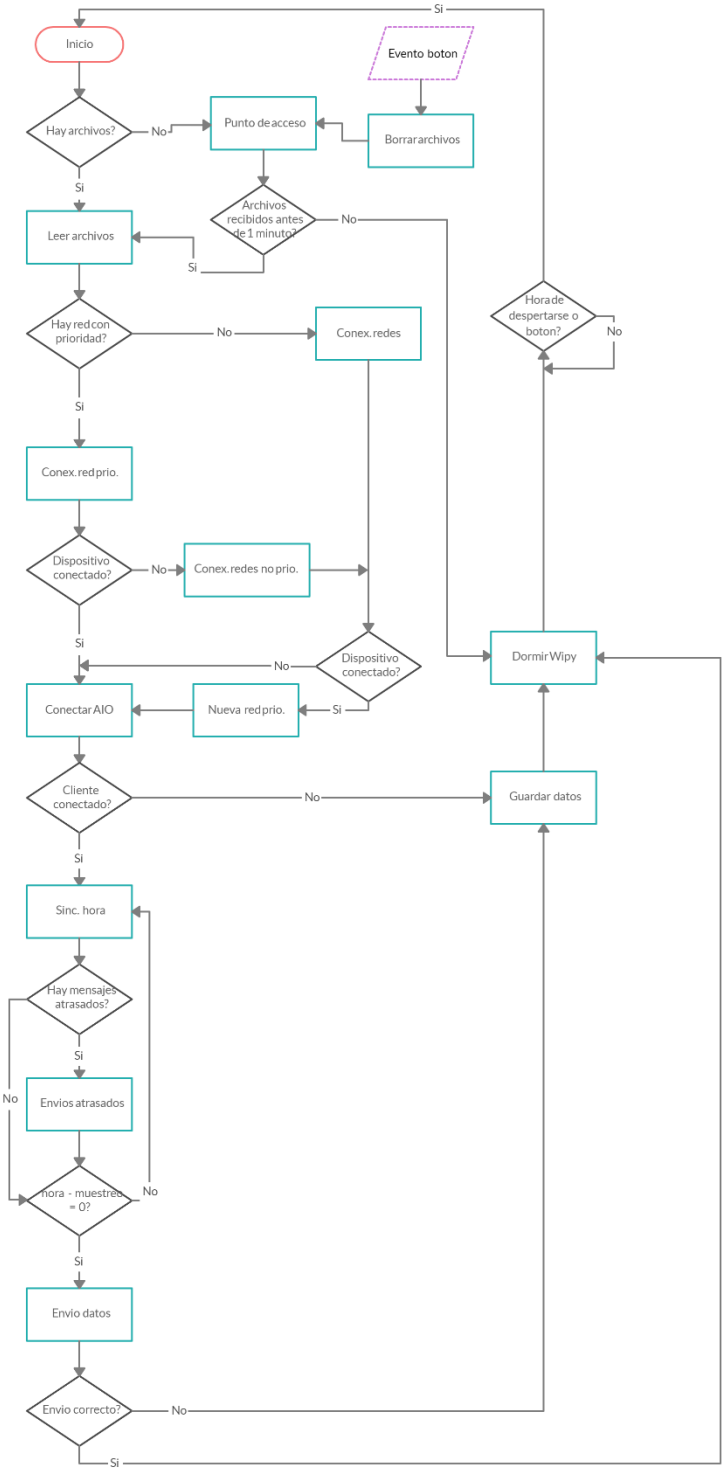
Una vez se hayan recibido estos datos, se notificará mediante un mensaje de "Datos enviados" en verde. Finalmente, el microcontrolador desconecta al dispositivo móvil una vez que guarde los datos. Sin embargo, también es posible desconectarse manualmente en cualquier instante mediante el botón de la derecha.

Figura 7.4. Datos enviados.

Capítulo 8. Programa del microcontrolador

En este capítulo se realiza una descripción resumida del programa. Si se desea, es posible ver el código completo en el Anexo 4.

El flujograma del programa es el siguiente:



Como se puede observar, el programa consiste fundamentalmente en dos partes:

- Un bucle principal
- Un *callback*, que es un evento que se ejecutaría en paralelo con el bucle principal. En este caso, si se mantiene pulsado el botón, se flashearía el dispositivo, borrando todos los archivos y mensajes guardados.

Tanto el bucle principal como el evento del botón se explicarán a continuación.

8.1. Lectura de archivos y *Bluetooth*

Los primeros pasos que realiza el microcontrolador es el de intentar leer los archivos que contienen las redes *Wi-Fi*, la información del servidor *Adafruit IO* y el tiempo de muestreo.

Si no es la primera vez que el dispositivo se pone en marcha y tampoco se ha flasheado, el microcontrolador contendrá estos archivos, por lo que no será necesario abrir el punto de acceso *Bluetooth* para obtenerlos. En este primer caso, el dispositivo pasará de este primer bloque a la conexión *Wi-Fi*, que se explicará más adelante.

Sin embargo, si no existen dichos archivos debido a que es la primera vez que se pone en marcha el dispositivo o porque se ha flasheado, es necesario obtener estas claves. Para ello, como ya se ha comentado en el capítulo de la aplicación móvil, se utiliza una conexión *Blueooth Low Energy*.

Durante un minuto, el *Wipy* abrirá un punto de acceso *Bluetooth*, caracterizado por una constante luz azul, anunciándose bajo el nombre de WIPY y creando una serie de servicios y características, tanto para los nombres y las contraseñas de las redes, como para la información del usuario del servidor y el tiempo de muestreo. Se crea una característica para cada uno de estos elementos excepto para la clave IO, ya que son 32 caracteres y ha de partirse por la mitad, que se crean dos.

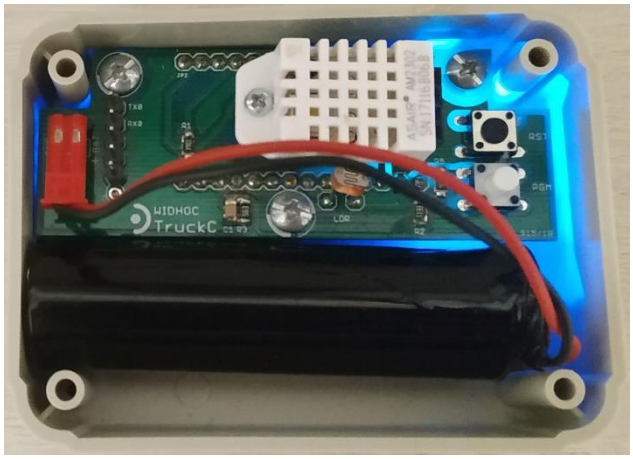


Figura 8.1. *Wipy* en modo *Access Point*.

Cada una de estas características tiene a su vez un *callback*, que, una vez recibida la información, la guarda en una variable (o una lista en el caso de las redes).

Una vez se hayan recibido los datos en cada una de las características, o si no se han recibido estos datos pero ha pasado el minuto, el dispositivo cerrará el punto de acceso, y no será posible conectarse a él o enviar más información hasta que se vuelva a abrir.

Una vez se cierra el punto de acceso, hay dos opciones: el dispositivo ha obtenido todas las claves correctamente, y puede continuar con la conexión *Wi-Fi* y con el servidor, o no se han enviado dichas claves o ha habido un error en la comunicación. En este segundo caso, el dispositivo hibernará el tiempo establecido y una vez se despierte volverá a abrirse el punto de acceso.

8.2. Conexión *Wi-Fi* y con *Adafruit IO*

Una vez obtenidas las claves, ya sea mediante los archivos previamente guardados o a través de *Bluetooth*, el siguiente bloque es el encargado de conectarse a una red *Wi-Fi* y posteriormente al servidor.

En una primera instancia, el dispositivo comprueba si existe alguna red prioritaria guardada, esto es, la última red en la que se pudo conectar. En caso de existir, se intentará conectar a dicha red. Si no fuese posible conectarse a la red prioritaria, se intentaría conectar a las demás redes.

Para la conexión con cada una de las redes tiene un tiempo de diez segundos. Si pasado este intervalo de tiempo no ha sido posible conectarse ha dicha red, lo intentaría con la siguiente en la lista, y así sucesivamente.

En el caso de que se haya podido conectar a una de las redes, se guardaría ésta como la nueva red prioritaria, y se pasaría a la conexión con la plataforma. Para la conexión con la plataforma, se crea una clase del tipo cliente MQTT, que además contiene información acerca del dominio web, el puerto del servidor y los *feeds* en los que se va a publicar el mensaje, para realizar la conexión y acordar una Calidad del servicio de nivel 1.

Si por el contrario no ha sido posible conectarse a ninguna red, tampoco será posible crear este cliente MQTT. La función encargada de crear el cliente también avisará de este error de conexión mediante una bandera con valor *False* y se encenderá una luz roja durante un segundo. Esta situación también puede darse si ha sido posible conectarse a una red, pero debido a que la señal es muy débil, no es posible realizar la conexión con el servidor.

8.3. Sincronización de la hora y envío/almacén de datos

En este último bloque tiene lugar la sincronización horaria del dispositivo y tanto el envío de datos como el guardado de estos mismos según el estado de la conexión.

En primer lugar, se intenta sincronizar la hora a través de 3 servidores: pool.ntp.org, hora.rediris.es y hora.roa.es. El primero es de los servidores más utilizados para los relojes en tiempo real en sistemas operativos, mientras que los otros dos corresponden al servidor RedIRIS (red académica española para servicios de telecomunicaciones) de Madrid y al servidor del Instituto Observatorio de la Armada, alojado en San Fernando, respectivamente. Estos servidores ofrecen la hora en UTC (Tiempo Universal Coordinado), por lo que para pasarlo a nuestra franja horaria es necesario sumarle una hora (CET).

La función encargada de la sincronización horaria comprueba la hora inicial del dispositivo antes de la sincronización: si la hora es menor al año actual (generalmente cuando no está sincronizado el dispositivo vuelve a 1970), esto indica que el dispositivo está flasheado, por lo que se activará una bandera para indicar que esta medición sería la primera a realizar en éste nuevo ciclo.

Durante la sincronización hay dos opciones: la primera, en la que si existe conexión a Internet y es posible sincronizarse con los servidores, y la segunda, en la que no es posible conectarse a Internet.

En el primer caso no habría problema y se podría pasar al envío/almacén de datos, pero en el segundo, nos podemos encontrar a su vez con dos situaciones:

- El dispositivo no ha podido conectarse a Internet, pero ya existía una sincronización anterior, por lo que la hora está correctamente actualizada y es posible almacenar los datos a la hora de la medida.

- El dispositivo no ha podido conectarse a Internet y no hay ninguna sincronización anterior. No es posible almacenar los datos con la hora de la medida, por lo que será necesario estimarla en próximos ciclos una vez el dispositivo se conecte a Internet.

Tenemos, por lo tanto, tres posibles situaciones finales:

Dispositivo conectado y hora sincronizada

Esta es la situación ideal, en la ha sido posible conectarse a Internet y la hora está sincronizada, por lo que se pueden mandar los datos a la plataforma,

Si el dispositivo se encuentra en este estado, antes de enviar la última medición comprobará si existen datos atrasados que no se pudieron mandar anteriormente y los enviará primero. Entre estos datos atrasados es posible que haya datos que no contienen la hora a la que fueron medidos, marcados con un asterisco. En este caso, se estima la hora de estas medidas en función de la hora actual y la cantidad de mensajes atrasados existentes.

Una vez enviados los datos atrasados, se comprueba si la bandera de la función de sincronización está activada, en cuyo caso se envía inmediatamente una medición actual. Si por el contrario esta bandera no está activada, indicando que no es la primera medición para este tiempo de muestreo, se comprobaría en bucle la diferencia entre la hora actual y la hora de la última medición enviada. En el momento en el que esta diferencia sea igual a la del tiempo de muestreo, se publicaría esta última medición.

El formato elegido para publicar los datos en cada *feed* es el siguiente:

- Para *datos_sensores*, se publica un diccionario que contiene información de todas las demás *feeds*: temperatura, humedad, luminosidad y batería, así como el EPOCH, que es un número que representa la cantidad de segundos transcurridos desde el 1 de enero de 1970 y que es especialmente útil para la conversión de unidades horarias a fecha.

- Para los demás *feeds*, se publica el valor numérico correspondiente con el objetivo de poder representarlo gráficamente.

A continuación, se muestra un ejemplo de un *dashboard* con diferentes mediciones recibidas:

Figura 8.2. Ejemplo de mediciones almacenadas en la plataforma

Dispositivo no conectado y hora sincronizada

En este caso, como si se había podido sincronizar el dispositivo anteriormente, es posible almacenar la medición con la hora correspondiente.

Al igual que cuando el dispositivo está conectado a Internet, se comprueba la diferencia entre la hora actual y la de la última medición y una vez la diferencia sea el tiempo de muestreo se guarda esta última medida.

Dispositivo no conectado y hora no sincronizada

En este último caso, se guarda la medición con un asterisco para que, como ya se ha mencionado en el caso del envío de los datos, el microcontrolador pueda diferenciarlo y estimar su hora en función de la hora de sincronización y la cantidad de mensajes guardados.

8.4. *Deepsleep* y fin del ciclo

Una vez realizadas todas las acciones anteriormente expuestas, el dispositivo tendría que esperar hasta que transcurriese el tiempo de muestreo para volver a tomar una medición. Con el objetivo de ahorrar energía, se obliga al microcontrolador a hibernar a través de la función *deepsleep*, que detiene el procesador durante un tiempo determinado.

Dado que para la conexión con cada red almacenada hacen falta al menos diez segundos, y que hay un tiempo de separación de un segundo entre cada mensaje atrasado a enviar, y que además el dispositivo tiene que estar despierto con tiempo de sobra, se ha estimado el periodo de hibernación del microcontrolador de la siguiente forma:

$$tiempo\ (ms) = tiempo_{muestreo} * 60000 - (num_{redes} * 12000 + num_{envios\ atrasados} * 3000 + 15000)$$

Por otro lado, en el caso en el que, ya sea porque pasó el minuto del punto de acceso o debido a que hubo un error de envío de las claves por *Bluetooth*, no se tenga el tiempo de muestreo, el dispositivo hibernará indefinidamente. Siempre es posible volver a abrir el punto de acceso *Bluetooth* a través del botón.

8.5. Flashear el dispositivo y volver a abrir el punto de acceso

Si se desea volver a abrir el punto de acceso, o flashear el dispositivo y eliminar todos los archivos y mediciones almacenados, es necesario activar un evento mediante el botón (no confundir con el de reset).

Para activar este evento, es necesario pulsar el botón tres veces. A modo de advertencia, las dos primeras veces que se pulse el botón, aparecerá una luz roja parpadeante. La tercera vez que se pulse, se activará el evento y se encenderá una luz roja de forma continua durante un segundo y medio que indicará que el dispositivo está borrando dicha información. Una vez borrada, el dispositivo volverá a abrir el punto de acceso durante un minuto.

Bibliografía

[1] "Introducción al Internet de las Cosas. FQIngenieria." [Online]. Available: https://www.fqingenieria.com/es/conocimiento/introduccion-al-internet-de-las-cosas-iot-y-rfid-72.

[2] P. G. Peter Friess, "Internet of Things Strategy Research Roadmap."

[3] B. Cendón, "El origen del IoT." [Online]. Available: http://www.bcendon.com/el-origen-del-iot/.

[4] H. F. Atlam, R. J. Walters, and G. B. Wills, "Internet of Things: State-of-the-art, Challenges, Applications, and Open Issues," *Int. J. Intell. Comput. Res.*, vol. 9, no. 3, pp. 928–938, 2018, doi: 10.20533/ijicr.2042.4655.2018.0112.

[5] K. T. Sook Hua Wong, Industry segment manager, "The Top Five Challenges of IoT." [Online]. Available: https://www.iotevolutionworld.com/iot/articles/445866-top-five-challenges-iot.htm.

[6] "Wipy 3.0 specifications @ docs.pycom.io." [Online]. Available: https://docs.pycom.io/datasheets/development/wipy3/.

[7] "Temperature and humidity sensor DHT22 @ www.adafruit.com." [Online]. Available: https://www.adafruit.com/product/385.

[8] "What is a Light Dependent Resistor and Its Applications @ www.watelectronics.com." [Online]. Available: https://www.watelectronics.com/light-dependent-resistor-ldr-with-applications/.

[9] "MicroPython Documentation @ docs.micropython.org." [Online]. Available: https://docs.micropython.org/en/latest/.

[10] "Pycom Documentation @ docs.pycom.io." [Online]. Available: https://docs.pycom.io/.

[11] "Getting Started with Adafruit IO @ learn.adafruit.com." [Online]. Available: https://learn.adafruit.com/welcome-to-adafruit-io/getting-started-with-adafruit-io.

[12] "App Inventor @ es.wikipedia.org." [Online]. Available: https://es.wikipedia.org/wiki/App_Inventor.

[13] "MQTT V3 @ Docs.Oasis-Open.Org," *MQTT Version 3.1.1*, 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[14] "Introducing the MQTT Protocol @ www.hivemq.com." [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/.

[15] "Client, Broker/Server and Connection Establishment @ www.hivemq.com." [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-

establishment/.

[16] “Protocolo TCP/IP @ docs.oracle.com.” [Online]. Available: https://docs.oracle.com/cd/E19957-01/820-2981/ipov-10/.

[17] “Adafruit IO MQTT API @ io.adafruit.com.” [Online]. Available: https://io.adafruit.com/api/docs/mqtt.html#adafruit-io-mqtt-api.

[18] “MQTT Publish, Subscribe & Unsubscribe @ www.hivemq.com.” [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/.

[19] “Quality of Service @ www.hivemq.com.” [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

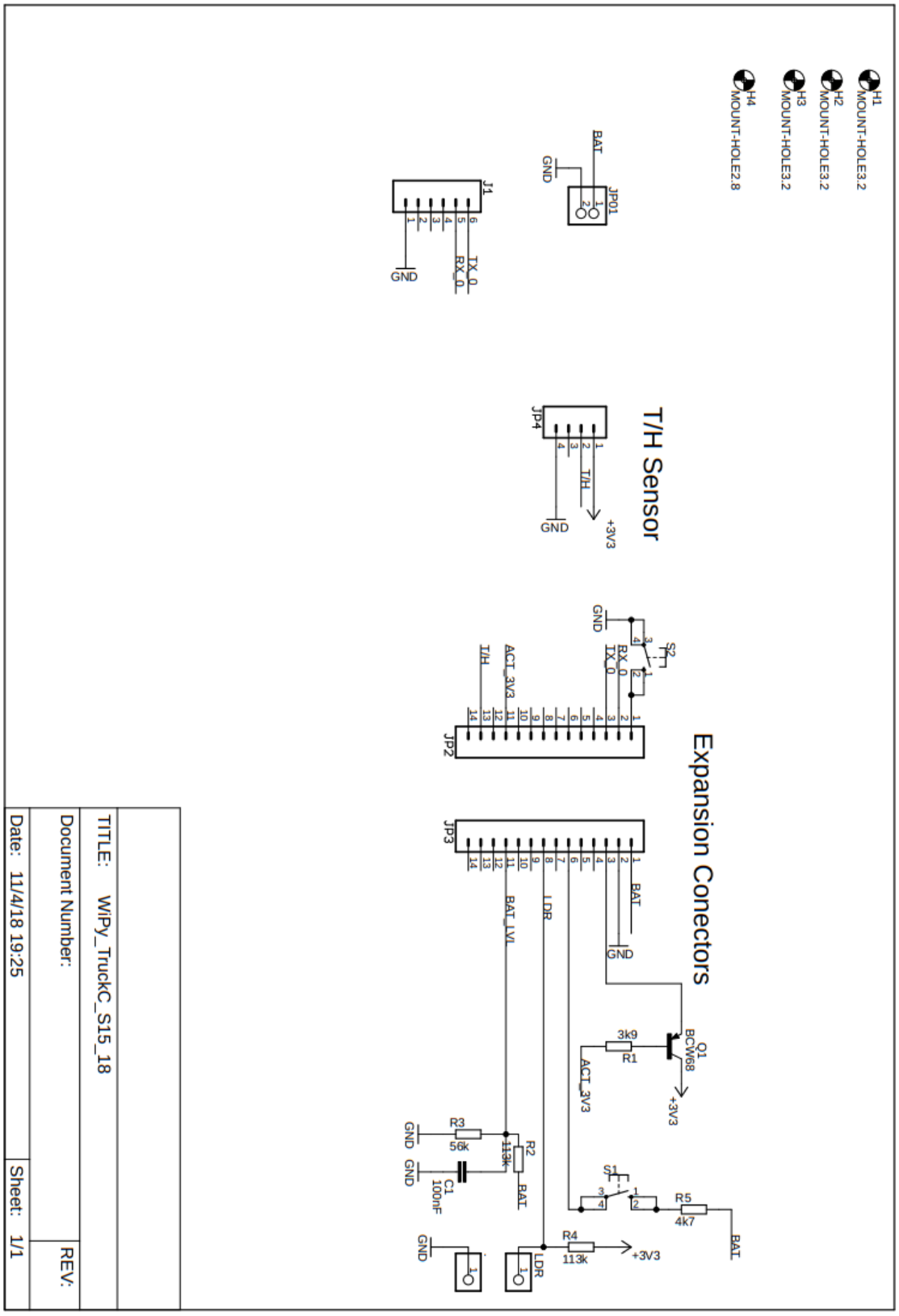
[20] “Bluetooth Low Energy Introduction @ Learn.Adafruit.Com.” [Online]. Available: https://learn.adafruit.com/adafruit-ultimate-gps-on-the-raspberry-pi/introduction.

[21] “GAP @ learn.adafruit.com.” [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap.

[22] “GATT @ learn.adafruit.com.” [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt.

[23] “Chapter 4. GATT @ Www.Oreilly.Com.” [Online]. Available: https://www.oreilly.com/library/view/high-performance-spark/9781491943199/ch04.html.

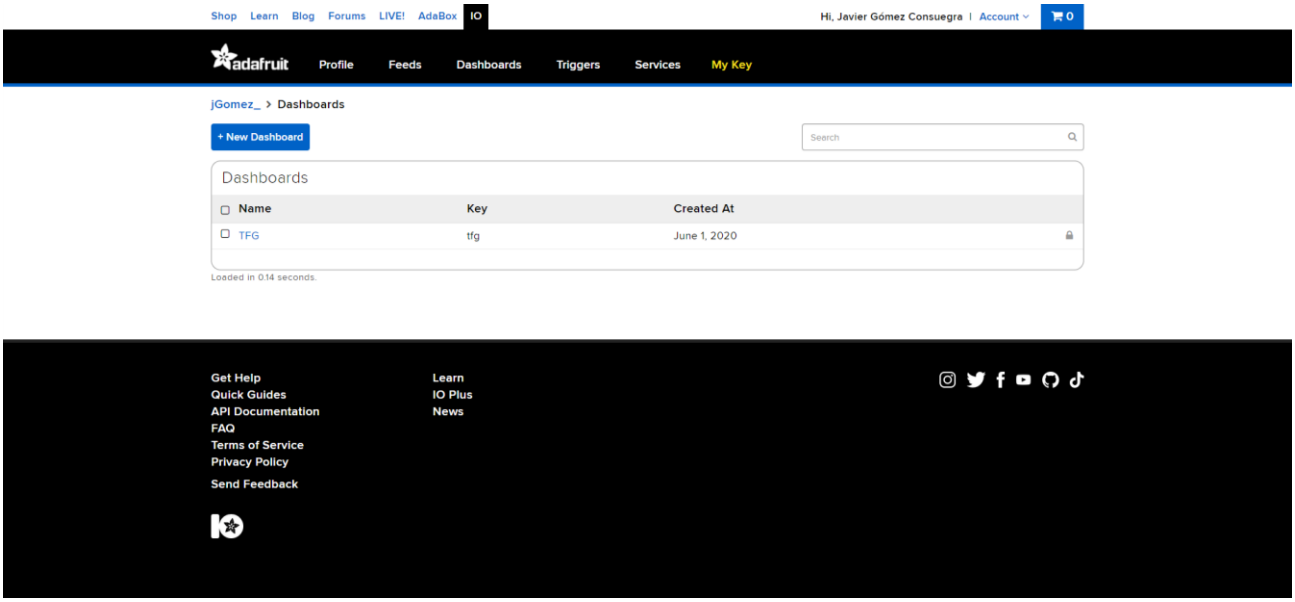
Anexo 1. Esquema del montaje del dispositivo *IoT*



Anexo 2. Puesta a punto de la interfaz de *Adafruit IO*

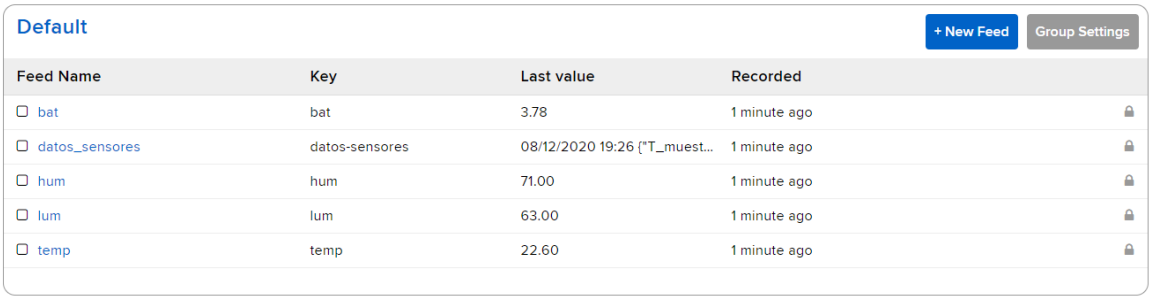
En primer lugar, es necesario registrarse en la página de *Adafruit IO* para obtener un usuario y una clave IO. El enlace para registrarse es el siguiente: https://accounts.adafruit.com/users/sign_up

Una vez creado el perfil, podemos acceder a diferentes pestañas como son nuestras *feeds*, *dashboards*, clave IO, etc.



Ejemplo de un perfil en *Adafruit IO*.

Como se ha explicado anteriormente, en este servidor los datos se almacenan en *feeds*, y que se pueden visualizar de manera personalizada a través de *dashboards*. Por ello, para que este proyecto funcione es muy importante tener *feeds* con estos mismos nombres:



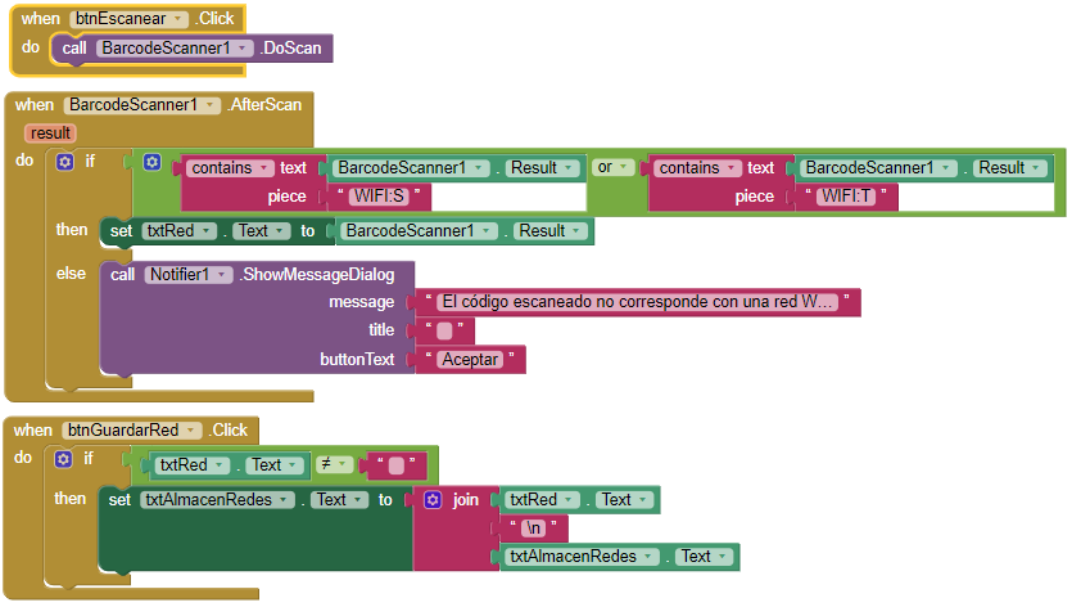
***Feeds* utilizadas en el proyecto.**

Una vez hecho esto, podemos crear una *dashboard* a nuestro gusto. En la configuración, podemos crear diferentes bloques, que son elementos personalizables que nos permiten incorporar los datos de diferentes feeds en múltiples formas. En este proyecto, por ejemplo, se ha utilizado un bloque con el resumen de todos los datos, un bloque con la representación gráfica de la humedad, temperatura y luminosidad, y cuatro bloques indicadores del valor de batería, humedad, temperatura y luminosidad.

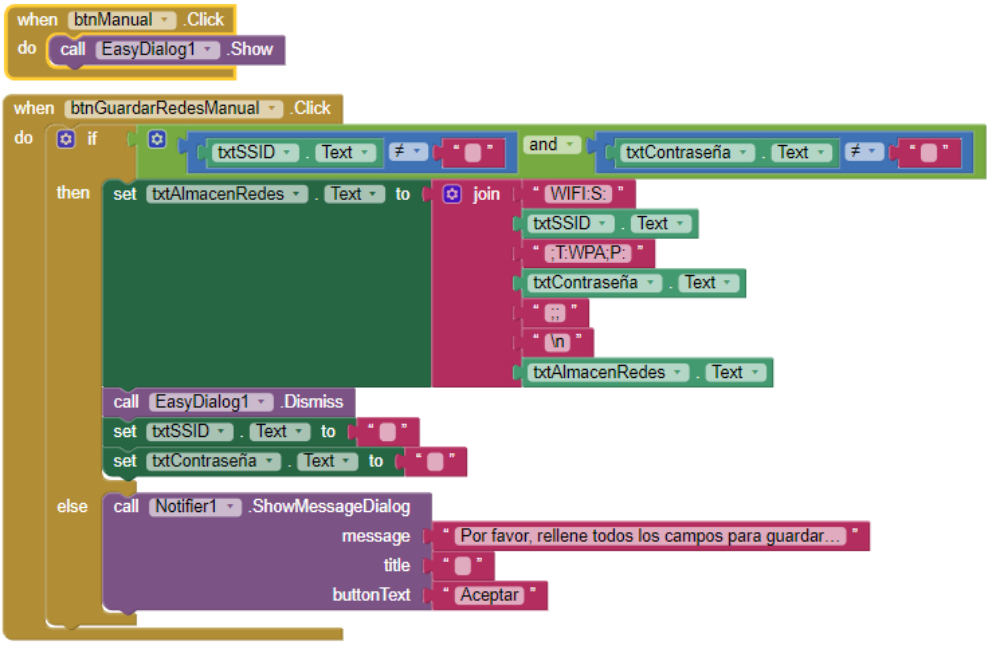
Ejemplo de *dashboard*.

Anexo 3. Bloques de la aplicación móvil

Con el botón de escanear, se inicia un escáner QR. En función de si el código coincide con el formato de las redes *Wi-Fi*, aparece o no un aviso. Si el formato coincide, se puede guardar dicha red con un botón.

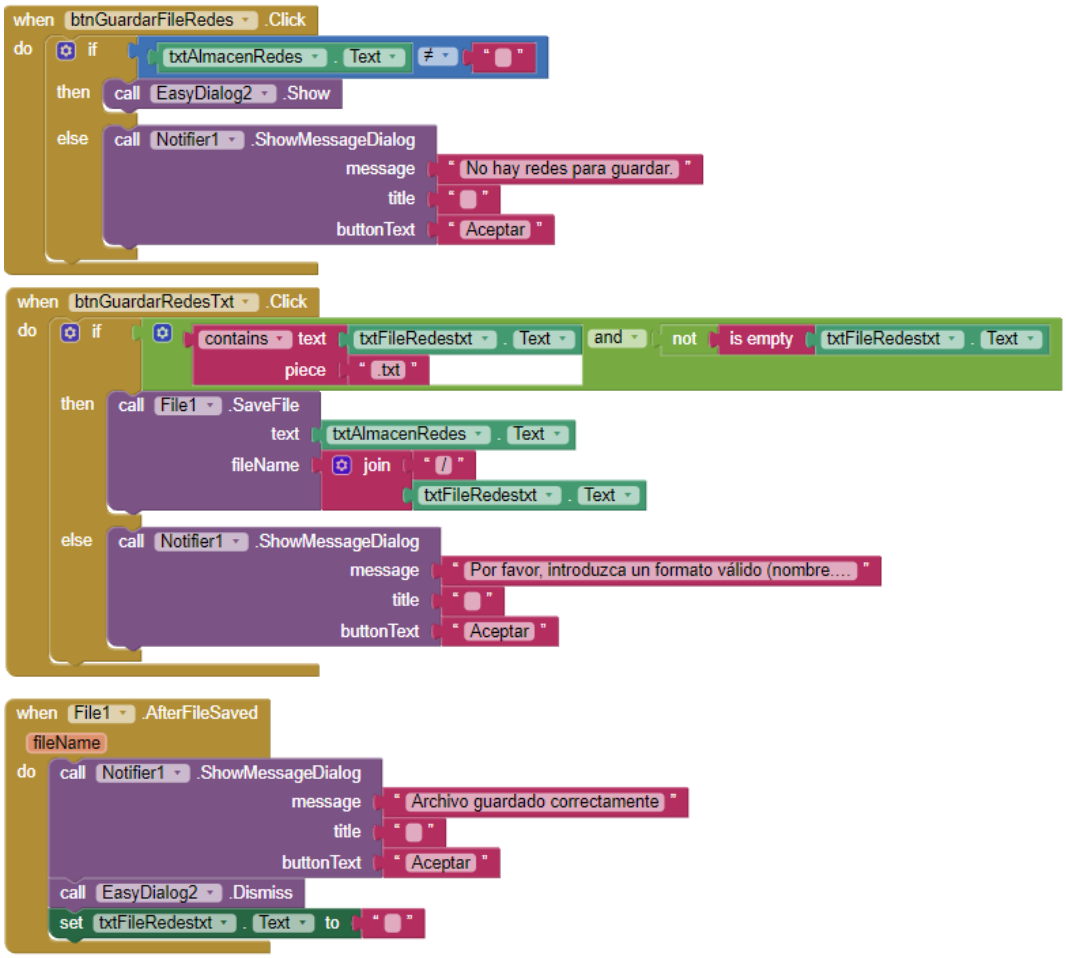


Si se pulsa el botón para introducirlos de forma manual, aparece una ventana en la que se puede introducir un SSID y una contraseña. Cuando se guarde esa red, se codifica para que coincida con las otras.

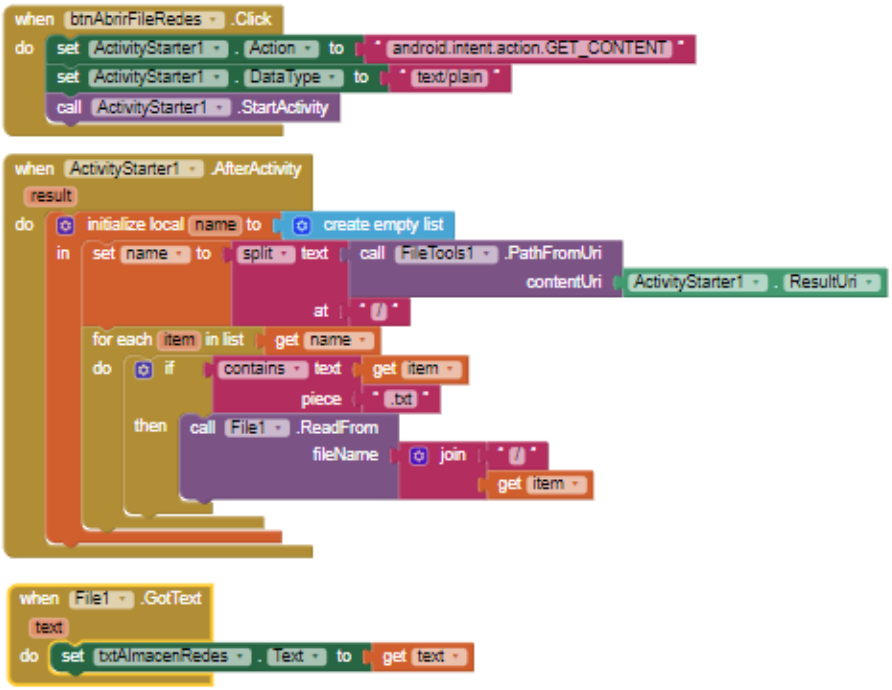


Para borrar las redes almacenadas

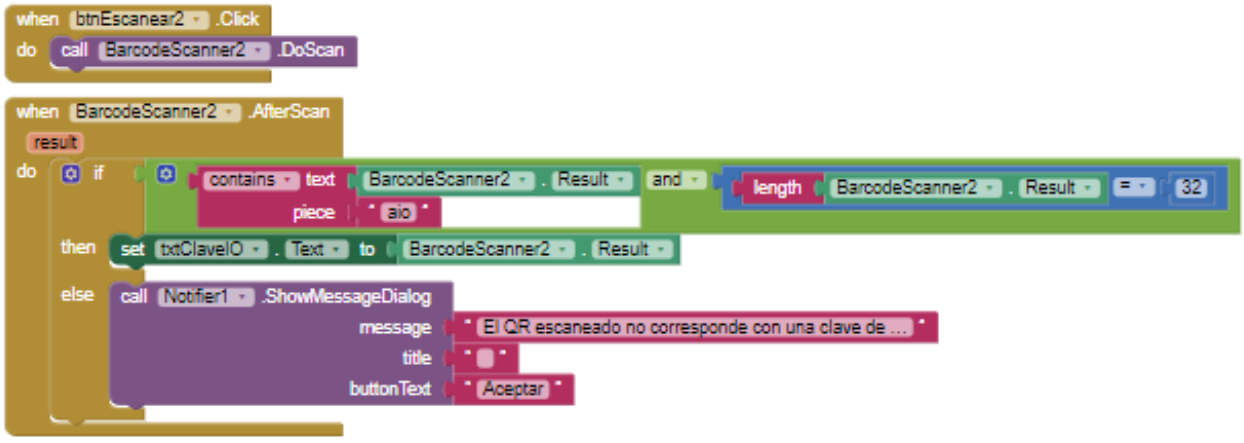
Para guardar un archivo de texto con las redes. El / que se le añade al nombre del fichero hace referencia a la ruta donde se va a guardar. Para Android inferior a 10 se guarda en la tarjeta SD, para versiones posteriores, el archivo se guarda en un archivo interno de la aplicación.



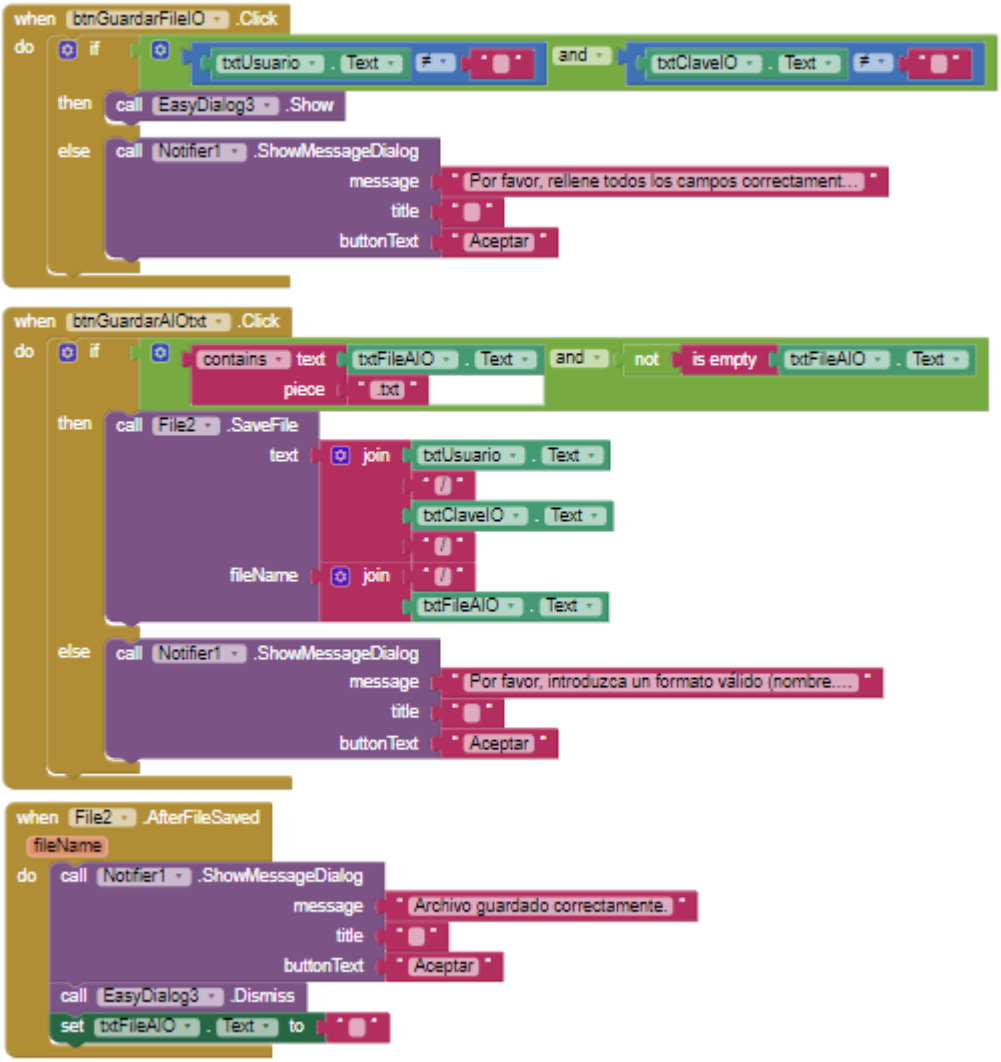
Para abrir un archivo previamente guardado. Se inicia una actividad que permite seleccionar un archivo de texto. Una vez seleccionado, se extrae de su directorio el nombre del archivo segmentándolo en una lista en función de los /.



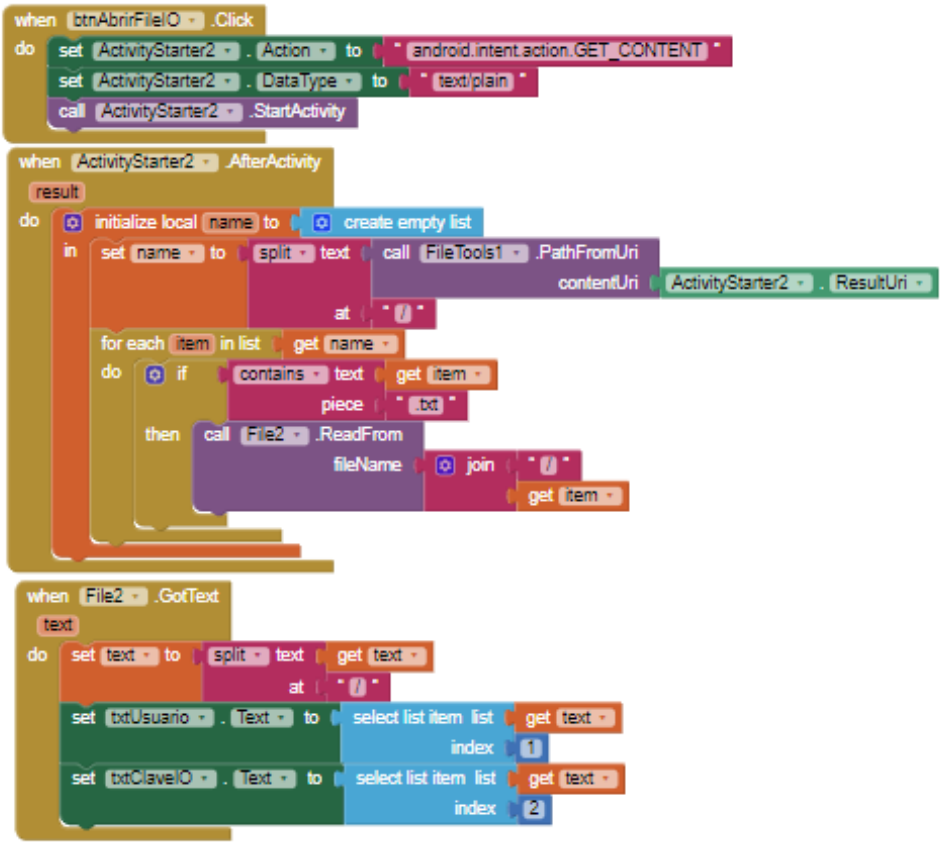
Escáner de la clave IO. Similar al de las redes, pero esta vez comprueba que sean 32 caracteres y que contenga “aio”



Guardado de archivo de texto para el usuario de *Adafruit IO*. En este caso separamos nombre y clave a través de un / para poder luego segmentar el texto en una lista y rellenar los campos de los cuadros de texto más facilmente.

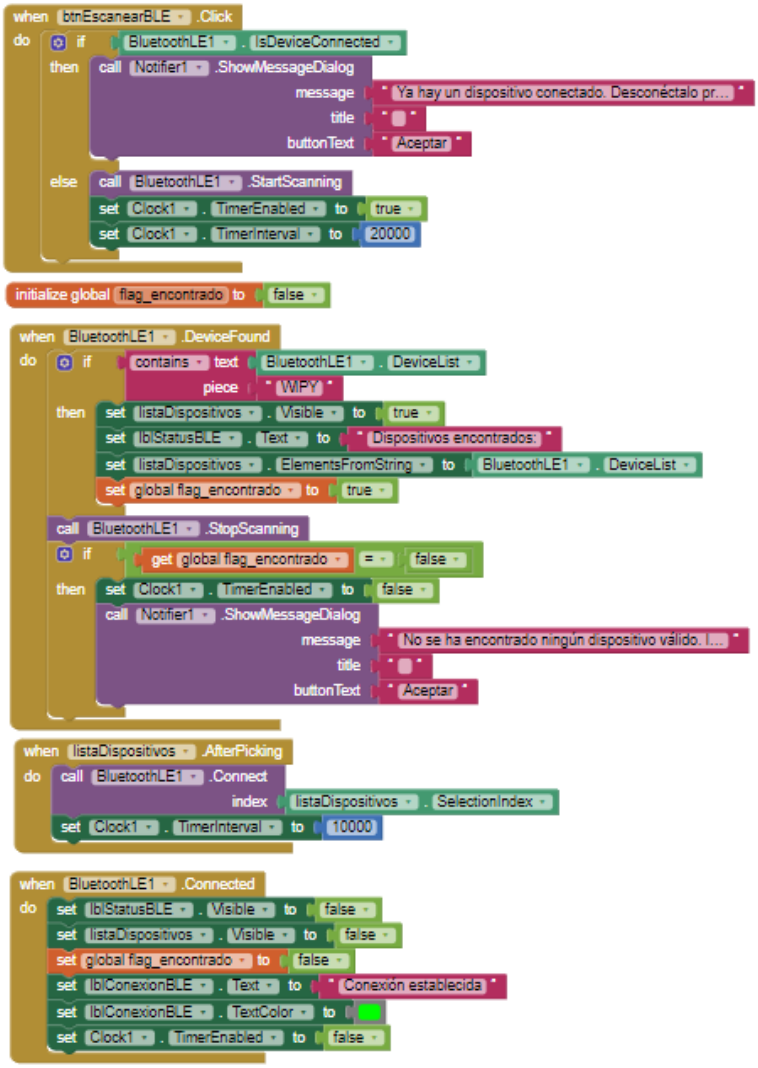


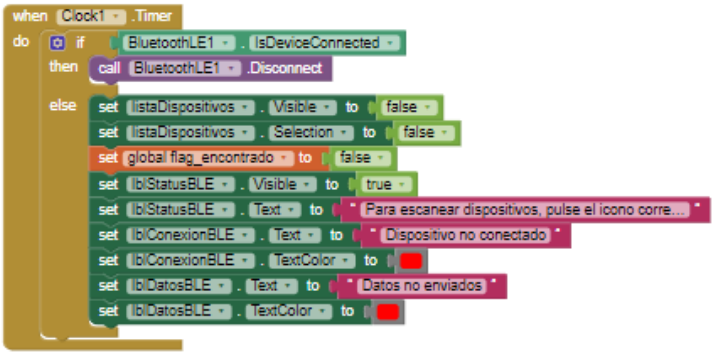
Para abrir archivos de los usuarios previamente guardados, idéntico al anterior salvo que partimos el texto cotenido en el archivo en una lista en función de / como hemos dicho, para rellenar los cuadros de texto.



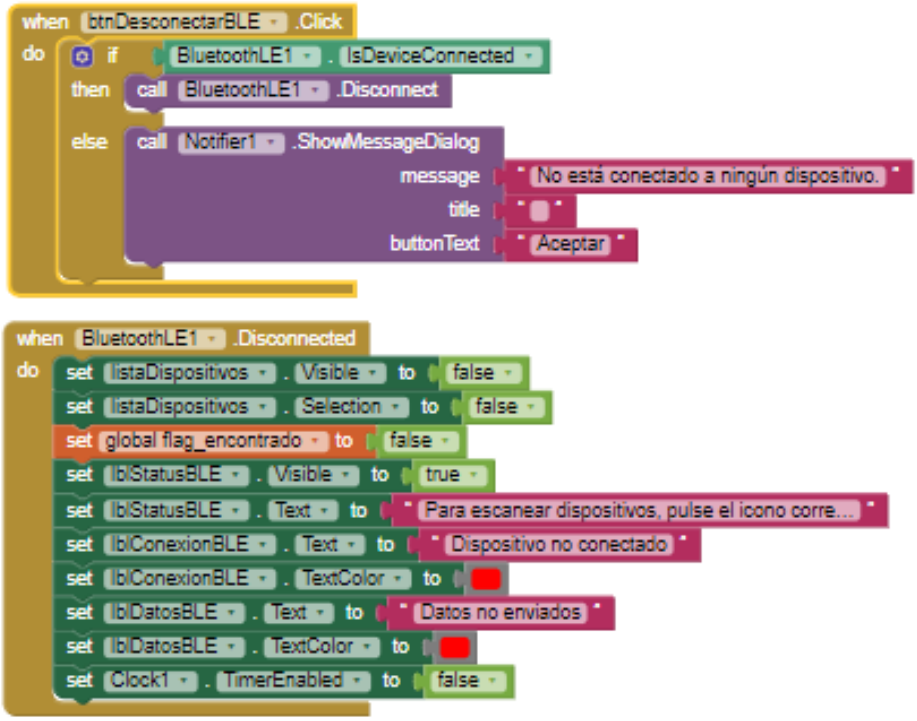
Para escanear dispositivos *BLE*, filtramos los posibles dispositivos en función de si se anuncian como WIPY o no. Si se pulsa el nombre de dicho dispositivo se intenta realizar la conexión. Hay dos temporizadores, de 20 y de 10 segundos, para evitar que la app se quede clavada en la lista de dispositivos disponibles y una vez se selecciona el dispositivo, respectivamente. Estos temporizadores se desactivan una vez se logra la conexión.

Una vez se consigue la conexión, aparece un mensaje verde confirmando dicho estado.

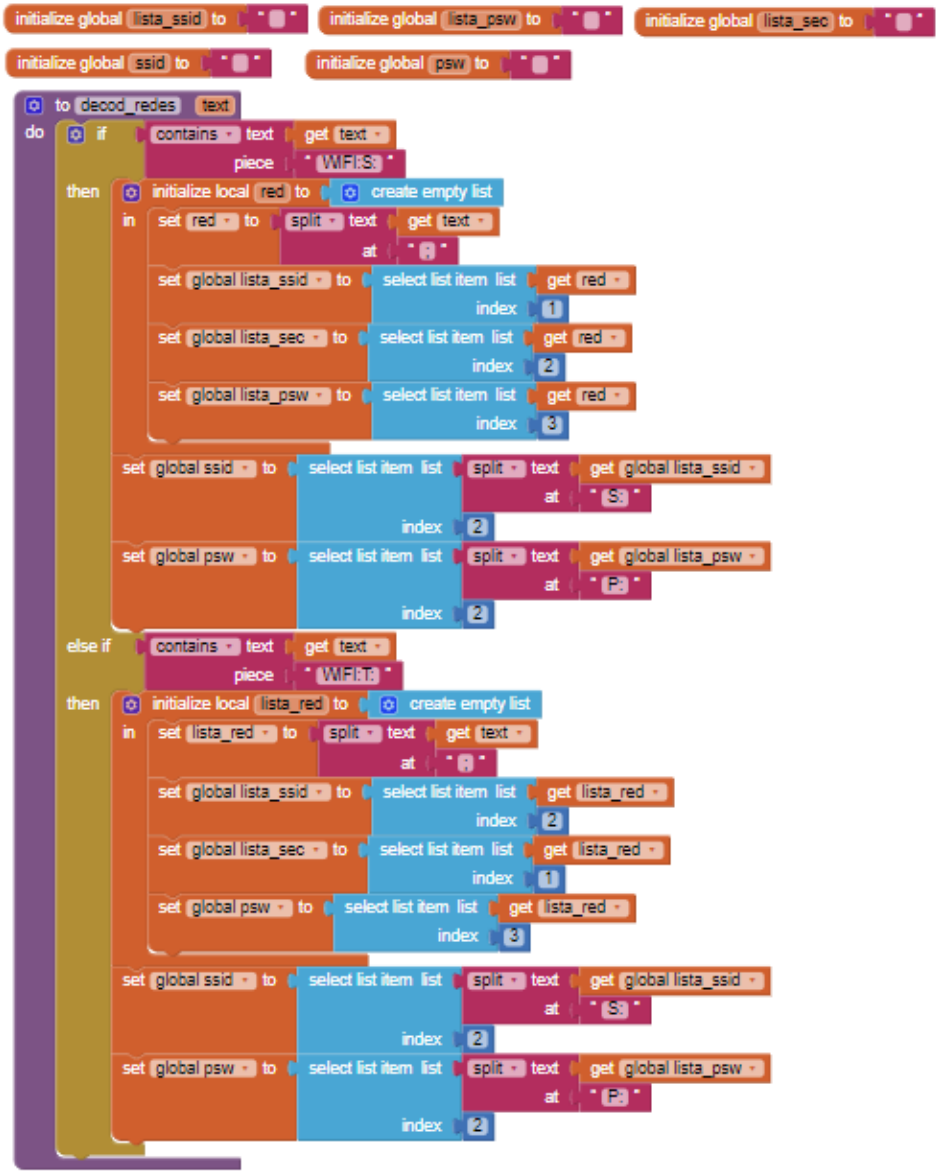




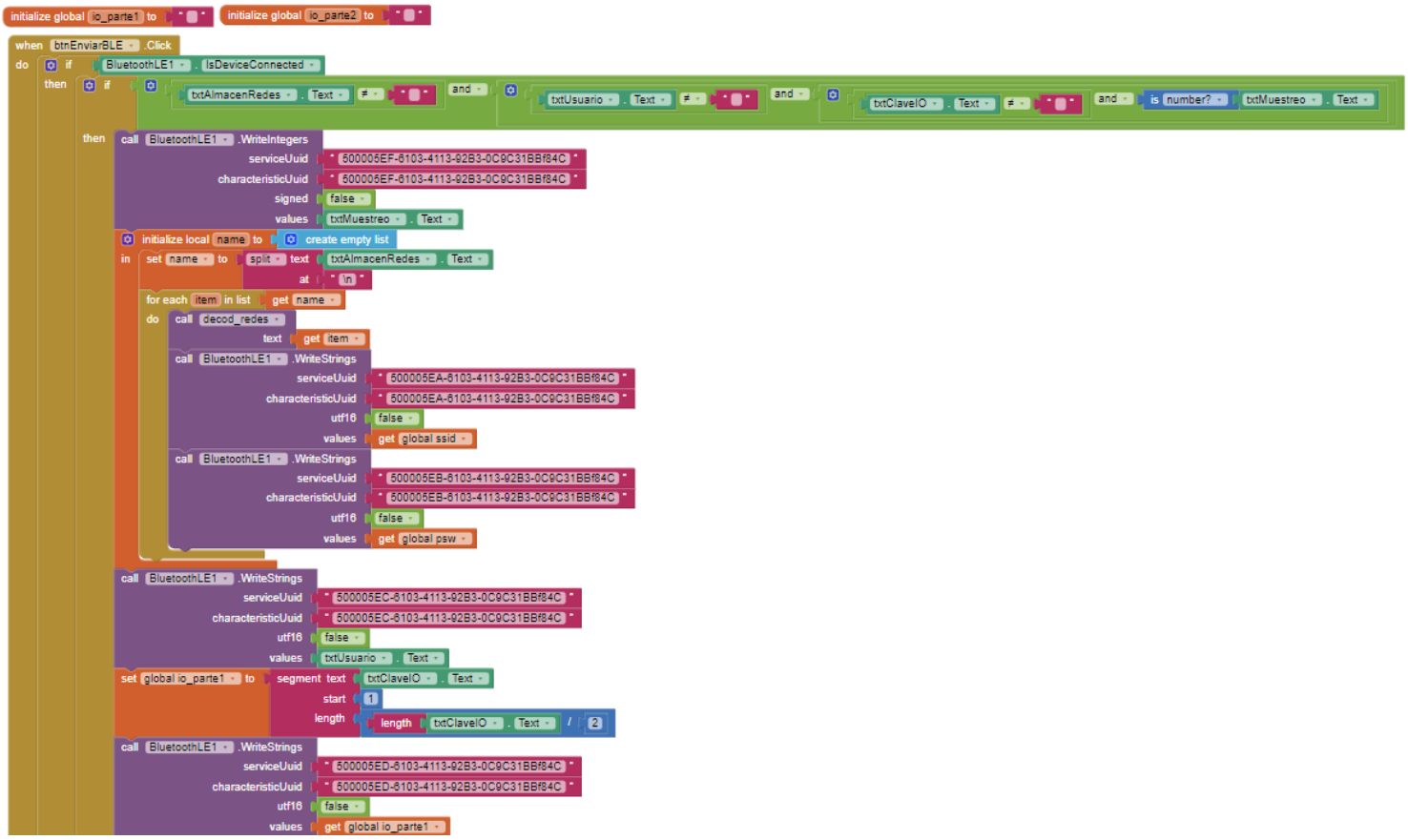
Desconexión del enlace *BLE*.



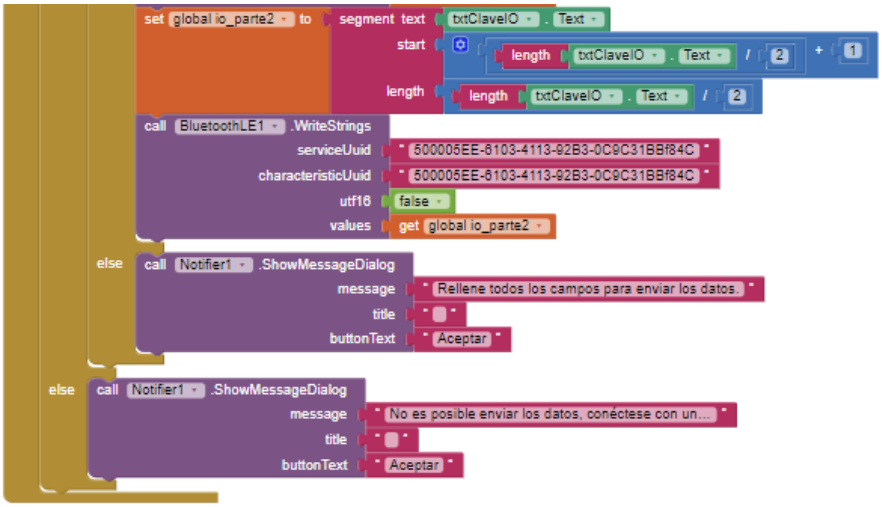
Función que se utilizara a la hora de envíar las redes por *Bluetooth*. Se decodifica cada una de las cadenas de texto de las redes partiendo dichas cadenas en una lista a través de los caracteres ;. Una vez realizada esta segmentación, se hace una segunda pasada para eliminar trozos como S: o P: para conseguir el SSID y la contraseña.

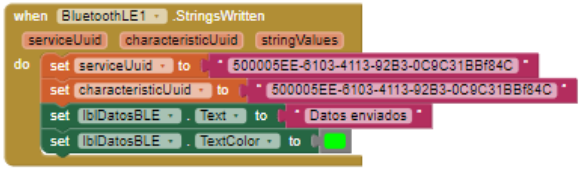


Por último, el envío de datos. Se escriben los datos en sus correspondientes características. Una vez se haya escrito en la última característica, la de la segunda parte de la clave IO, se confirma que los datos han sido enviados con una advertencia en color verde. Parte 1:



Parte 2:





Anexo 4. Código del *Wipy*

main.py – Código principal

```
from network import Bluetooth, WLAN
import pycom
import json
import time
import machine
from umqtt import MQTTClient
from sens_op import leer_sensores
import ble
import f_op
import wlan_op
import btn

#Javier Gómez Consuegra
#Trabajo de Fin de Grado: Diseño de sistema IoT para sensorizacion de atmósferas
controladas
#Descripción: El programa permite la obtención y envío de datos mediante una conexión
WIFI a la IO de Adafruit para la monitorización de la temperatura, humedad,
luminosidad, asi como la batería restante, captadas por el dispositivo
#Para obtener las redes a las que ha de conectarse y la información de la IO, se ha
diseñado una aplicación movil que envía dichos datos, una vez abierto el punto de
acceso, mediante Bluetooth Low Energy(BLE)
(wake_reason, gpio_list) = machine.wake_reason()
machine.pin_deepsleep_wakeup(('P3', 'P21'), mode=machine.WAKEUP_ANY_HIGH,
enable_pull=True)
pycom.heartbeat(False)
#VARIABLES GLOBALES
redes = []
info_usuario = {}
flag_reset = False
flag_mqtt = False

#Cuando no hay archivos
if pycom.nvs_get('estado') == 0:
    #Creamos el boton y su evento
    but = btn.BUTTON()
    try:
        redes = f_op.leer_redes("redes_ssid.txt","redes_psw.txt")
        info_usuario =
f_op.leer_info_usuario("usuario.txt","clave.txt","t_muestreo.txt")
    except:
        print("No se han podido encontrar los archivos")
        pycom.nvs_set('estado', 1)
    #Solo se activa el evento del boton una vez por ciclo, siempre y cuando no se haya
flasheado en dicho ciclo
    else:
        but.push = btn.push
```

```
#Si no hay archivos  el dispositivo abre el punto de acceso durante un minuto
pycom.heartbeat(False)
if pycom.nvs_get('estado') != 0:
    flag_reset = True
    pycom.rgbled(0x0000ff)
    while True:
            t_end = time.time() + 60
            print("Punto de acceso Bluetooth activado")
            dispositivo_ble = ble.ble_on()
            try:
                while True:
                    #Una vez recibidos los datos o cuando pasan 60 segundos, se
desconecta el cliente y se guarda la información
                    if time.time() == t_end or ble.comprobar_envio():
                        print("Mensajes recibidos o han pasado los 60 segundos")
                        redes = ble.enviar_redes()
                        info_usuario = ble.enviar_usuario()
                        ble.ble_off(dispositivo_ble)
                        break
                    else:
                        machine.idle()
                break
                #Si hay un error en las funciones BLE, se eliminan los datos
            except:
                redes = []
                info_usuario = {}
    pycom.rgbled(0x000000)
    if redes and info_usuario:
        #Si se han recibido nuevos datos, se borran los anteriores si es que existen y
se guardan estos
        try:
            f_op.borrar_archivos()
            print("Datos anteriores borrados")
        except:
            print("No habia datos para borrar")
        print("Nuevos datos guardados")

f_op.guardar_archivos("redes_ssid.txt","redes_psw.txt","usuario.txt","clave.txt","t_mu
estreo.txt",redes, info_usuario)
        pycom.nvs_set('estado', 0)

if redes and info_usuario:
    #Resumen de la información almacenada: redes, nombre del usuario, la clave de la
IO y el tiempo de muestreo
    print("Redes almacenadas:")
    for i in redes:
        print("SSID: " + i["ssid"] + " CONTRASEÑA: " + i["psw"])
    print("Usuario: " + info_usuario["usuario"]+ " Clave IO: " + info_usuario["clave"]
+ " Tiempo de muestreo: " + info_usuario["t_muestreo"])
    #Se intenta buscar una red con prioridad, que es la última red a la que se ha
podido conectar el dispositivo
    try:
        prio_wifi = f_op.leer_prio()
        print("Red con prioridad: " + prio_wifi)
    except:
        print("No hay ninguna red con prioridad almacenada")
        prio_wifi = ""

    wlan = WLAN(mode=WLAN.STA)
    #Si entre las redes guardadas está la que tiene prioridad, se intenta conectar
primero a ésta, si falla la conexión se intenta conectar a las demás
    if not wlan_op.conectar_prio(redes,prio_wifi):
        print("No ha sido posible conectarse a la red con prioridad")
        wlan_op.conectar_red(redes,prio_wifi)
```

```
    #Se intenta crear un cliente MQTT, si hay un error a la hora de crearlo, se
enciende una luz roja durante un segundo
    #La función crear_cliente() devuelve una lista con el cliente MQTT creado, un
diccionario con las variables de la IO a las que se envían los datos, y una bandera
que indica si hay o no conexión
    estado_cliente = wlan_op.crear_cliente(info_usuario)
    client = estado_cliente[0]
    dict_mqtt = estado_cliente[1]
    flag_mqtt = estado_cliente[2]


    #Este último bloque es el responsable del envío de datos a la IO de Adafruit si
existe conexión, o de almacenar los datos si la conexión ha fallado
    while True:
        #sinc_hora() sincroniza la hora mediante un reloj en tiempo real, y devuelve
la hora y una bandera que avisa si la hora se había reseteado o no
        lista_sinc = wlan_op.sinc_hora(flag_reset)
        hora = lista_sinc[0]
        flag_reset = lista_sinc[1]
        #La sincronización se produce cada segundo
        time.sleep_ms(1000)
        #Si no hay conexión, se guardan los datos, y una vez guardados se manda a
dormir el dispositivo para ahorrar batería hasta la próxima medición
        if flag_mqtt == False:
            if wlan_op.guardar_datos(info_usuario["t_muestreo"],flag_reset,hora):
                wlan_op.dormir_wipy(info_usuario["t_muestreo"],redes)
        else:
        #Si hay conexión, en primer lugar se comprueba si existen datos atrasados y se
envían
            if wlan_op.comprobar_almacen():
                #Si se han podido enviar estos datos atrasados, se borra el archivo
que los contenía para evitar repeticiones
                if wlan_op.envios_atrasados(client,hora,dict_mqtt):
                    wlan_op.borrar_datos()
                #En caso de no haber podido enviarlos, se guarda también esta última
medición y se duerme el dispositivo nuevamente
                else:
                    if
wlan_op.guardar_datos(info_usuario["t_muestreo"],flag_reset,hora):
                        wlan_op.dormir_wipy(info_usuario["t_muestreo"],redes)
            #Una vez enviados los datos atrasados, el dispositivo espera hasta que se
cumpla el tiempo de muestreo, y se envían los últimos datos. El dispositivo se duerme
después.
            if flag_reset or abs(hora[4] - pycom.nvs_get('min')) >=
int(info_usuario["t_muestreo"])  and pycom.nvs_get('seg') - hora[5] == 0:
                datos_sensores = leer_sensores()

wlan_op.enviar_datos(client,dict_mqtt,hora,datos_sensores,info_usuario["t_muestreo"])
                wlan_op.dormir_wipy(info_usuario["t_muestreo"],redes)
#Si no tiene redes almacenadas o la información del usuario, el dispositivo quedará
suspendido indefinidamente hasta que se pulse el boton para despertarlo
else:
    print("No se han encontrado datos")
    pycom.nvs_set('estado', 1)
    machine.deepsleep()
```

btn.py – Clase del botón

```
from time import sleep_ms, ticks_ms
import gc
import machine
import pycom
import f_op
import wlan_op

#Basado en la clase BUTTON del usuario crankshaft:
https://forum.pycom.io/topic/588/interrupt-button-debounce-with-long-short-press/8
class BUTTON:

  def __init__(self,pin='P21'):
    self.butms = 0
    self.pin = machine.Pin(pin, mode=machine.Pin.IN, pull=machine.Pin.PULL_DOWN)
    self.pin.callback(machine.Pin.IRQ_FALLING | machine.Pin.IRQ_RISING,  self.press)

  def __del__(self):
        print('Destructor called, Button deleted.')

  def push(self):
    pass

  def press(self,pin):
    now = ticks_ms()
    if self.butms == 0: self.butms = now
    else:
      if self.butms == now: return
    i = 0
    while i < 10:
      sleep_ms(1)
      if self.pin() == 1: i = 0
      else: i+=1

    if(i!=0): self.push()

    while self.pin() == 0: pass
    gc.collect()

contador = 0
#Cada vez que se pulsa el boton el led parpadea dos veces en color rojo. A la tercera
vez también se borran todos los archivos para flashear el dispositivo
def push():
    global contador
    contador  +=1
    print(contador)
    pycom.rgbled(0x7f0000)
    sleep_ms(100)
    pycom.rgbled(0x000000)
    sleep_ms(100)
    pycom.rgbled(0x7f0000)
    sleep_ms(100)
    pycom.rgbled(0x000000)
    if contador == 3:
      pycom.rgbled(0x7f0000)
      sleep_ms(1500)
      try:
          wlan_op.borrar_datos()
      except:
          print("No habia datos")
      try:
          f_op.borrar_archivos()
      except:
          print("No habia archivos")
      pycom.nvs_set('estado', 1)
      machine.deepsleep(1000)
```

ble.py – Funciones relacionadas con *Bluetooth Low Energy*

```
from network import Bluetooth
import time
import ubinascii
import ustruct

lista_ssid = []
lista_psw = []
redes = []
info_usuario = {}
usuario = ""
clave1 = ""
clave2 = ""
cont_clave = 0
tiempo = 0

def uuid2bytes(uuid):
    uuid = uuid.encode().replace(b'-',b'')
    tmp = ubinascii.unhexlify(uuid)
    return bytes(reversed(tmp))

def ble_on():
    #Creamos un objeto Bluetooth y anunciamos el dispositivo
    bluetooth = Bluetooth()
    bluetooth.set_advertisement(name='WIPY', service_uuid=b'1234567890123456')
    #Evento de conexión o desconexión de un cliente
    bluetooth.callback(trigger=Bluetooth.CLIENT_CONNECTED |
Bluetooth.CLIENT_DISCONNECTED, handler=conn_cb)
    #Se abre el punto de acceso
    bluetooth.advertise(True)
    #Servicio, característica y evento para los datos de las SSID de las redes
    srv1 = bluetooth.service(uuid=uuid2bytes('500005EA-6103-4113-92B3-0C9C31BBf84C'),
isprimary=True)
    chr1 = srv1.characteristic(uuid=uuid2bytes('500005EA-6103-4113-92B3-
0C9C31BBf84C'), value=5)
    char1_cb = chr1.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
Bluetooth.CHAR_READ_EVENT, handler=char1_cb_handler)
    #Datos de las contraseñas
    srv2 = bluetooth.service(uuid=uuid2bytes('500005EB-6103-4113-92B3-0C9C31BBf84C'),
isprimary=True)
    chr2 = srv2.characteristic(uuid=uuid2bytes('500005EB-6103-4113-92B3-
0C9C31BBf84C'), value=5)
    char2_cb = chr2.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
Bluetooth.CHAR_READ_EVENT, handler=char2_cb_handler)

    #Nombre de usuario
    srv3 = bluetooth.service(uuid=uuid2bytes('500005EC-6103-4113-92B3-0C9C31BBf84C'),
isprimary=True)
    chr3 = srv3.characteristic(uuid=uuid2bytes('500005EC-6103-4113-92B3-
0C9C31BBf84C'), value=5)
    char3_cb = chr3.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
Bluetooth.CHAR_READ_EVENT, handler=char3_cb_handler)
    #Clave del usuario, como solo se pueden mandar hasta 20 caracteres y la clave son
32, se parte en dos
    srv4 = bluetooth.service(uuid=uuid2bytes('500005ED-6103-4113-92B3-0C9C31BBf84C'),
isprimary=True)
    chr4 = srv4.characteristic(uuid=uuid2bytes('500005ED-6103-4113-92B3-
0C9C31BBf84C'), value=5)
    char4_cb = chr4.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
Bluetooth.CHAR_READ_EVENT, handler=char4_cb_handler)
    srv5 = bluetooth.service(uuid=uuid2bytes('500005EE-6103-4113-92B3-0C9C31BBf84C'),
isprimary=True)
    chr5 = srv5.characteristic(uuid=uuid2bytes('500005EE-6103-4113-92B3-
0C9C31BBf84C'), value=5)
    char5_cb = chr5.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
Bluetooth.CHAR_READ_EVENT, handler=char5_cb_handler)
```

```
    #Tiempo de muestreo
    srv6 = bluetooth.service(uuid=uuid2bytes('500005EF-6103-4113-92B3-0C9C31BBf84C'),
isprimary=True)
    chr6 = srv6.characteristic(uuid=uuid2bytes('500005EF-6103-4113-92B3-
0C9C31BBf84C'), value=5)
    char6_cb = chr6.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
Bluetooth.CHAR_READ_EVENT, handler=char6_cb_handler)

    return bluetooth

#Se cierra el punto de acceso
def ble_off(bluetooth):
    bluetooth.advertise(False)
    time.sleep_ms(3000)
    bluetooth.disconnect_client()

#Evento de conexión y desconexión del cliente BLE
def conn_cb (bt_o):
    events = bt_o.events()
    if  events & Bluetooth.CLIENT_CONNECTED:
        print("Cliente conectado")
    elif events & Bluetooth.CLIENT_DISCONNECTED:
        print("Cliente desconectado")

#Cada charN_cb_handler() corresponde con el evento que tiene lugar cuando se reciben
los valores en cada característica, todos siguen el mismo proceso
def char1_cb_handler(chr):
    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        #Se decodifica el mensaje recibido y se elimina cualquier carácter adicional
que pueda tener (\x00)
        ssid_decoded = chr.value().decode('utf-8').strip('\x00')
        global lista_ssid
        lista_ssid.append(ssid_decoded)

def char2_cb_handler(chr):
    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        psw_decoded = chr.value().decode('utf-8').strip('\x00')
        lista_psw.append(psw_decoded)

def char3_cb_handler(chr):
    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        global usuario
        usuario = chr.value().decode('utf-8').strip('\x00')

def char4_cb_handler(chr):
    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        clave1_decoded = chr.value().decode('utf-8')
        global clave1
        clave1 = clave1_decoded.strip('\x00')
        global cont_clave
        cont_clave = cont_clave + 1

def char5_cb_handler(chr):
    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        clave2_decoded = chr.value().decode('utf-8')
        global clave2
        clave2 = clave2_decoded.strip('\x00')
        global cont_clave
        cont_clave = cont_clave + 1
```

```
def char6_cb_handler(chr):
    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        global tiempo
        tiempo_decoded1 = chr.value().decode('utf-8')
        tiempo_decoded2 = ustruct.unpack('i', tiempo_decoded1)
        tiempo = str(tiempo_decoded2[0])

#Se guardan las SSID y las contraseñan de las redes recibidas
def enviar_redes():
    global redes
    for i in range(len(lista_ssid)):
        red = {"ssid": lista_ssid[i], "psw": lista_psw[i]}
        redes.append(red)
    return redes

#Se guarda la información del usuario
def enviar_usuario():
    global info_usuario
    info_usuario = {"usuario": usuario, "clave": clave1+clave2, "t_muestreo": tiempo}
    global cont_clave
    cont_clave = 0
    return info_usuario

#Si se han obtenido los dos mensajes de la clave IO, que es el último que se manda, se
indica que se ha recibido el envío
def comprobar_envio():
    if cont_clave == 2:
        return True
```

f_op.py – Funciones relacionadas con archivos de texto

```
import os
redes = []
info_usuario = {}

def leer_redes(f_ssid, f_psw):
    global redes
    #Buscamos las SSID y las contraseñas, y las almacenamos en una lista de
diccionarios
    f = open(f_ssid, 'r')
    line_read_ssid=f.readlines()
    for i in range(len(line_read_ssid)):
            line_read_ssid[i]=line_read_ssid[i].strip('\n')
    f.close()
    f = open(f_psw, 'r')
    line_read_psw=f.readlines()
    for i in range(len(line_read_psw)):
            line_read_psw[i]=line_read_psw[i].strip('\n')
    f.close()
    for i in range(len(line_read_ssid)):
        red = {"ssid": line_read_ssid[i], "psw": line_read_psw[i]}
        redes.append(red)
    return redes

def leer_info_usuario(f_usuario, f_clave, f_t_muestreo):
    global info_usuario
    #Hacemos lo mismo con la informacion de la IO
    f = open(f_usuario, 'r')
    line_read_usuario=f.readline().strip('\n')
    f.close()
    f = open(f_clave, 'r')
    line_read_clave=f.readline().strip('\n')
    f.close()
    f = open(f_t_muestreo, 'r')
    line_read_t_muestreo=f.readline().strip('\n')
    f.close()
    info_usuario ={"usuario": line_read_usuario,"clave":line_read_clave,"t_muestreo":
line_read_t_muestreo}
    return info_usuario

#Guardamos los redes, informacion del servidor,etc en sus respectivos archivos
def guardar_archivos(f_ssid, f_psw,f_usuario,
f_clave,f_t_muestreo,redes,info_usuario):
    f = open(f_ssid, 'a')
    for i in redes:
        f.write(i["ssid"])
        f.write('\n')
    f.close()
    f = open(f_psw, 'a')
    for i in redes:
        f.write(i["psw"])
        f.write('\n')
    f.close()
    f = open(f_usuario, 'w')
    f.write(info_usuario["usuario"])
    f.close()
    f = open(f_clave, 'w')
    f.write(info_usuario["clave"])
    f.close()
    f = open(f_t_muestreo, 'w')
    f.write(str(info_usuario["t_muestreo"]))
    f.close()
```

```
#Borramos los archivos
def borrar_archivos():
    os.remove("redes_ssid.txt")
    os.remove("redes_psw.txt")
    os.remove("usuario.txt")
    os.remove("clave.txt")
    os.remove("t_muestreo.txt")

#Guardamos la red prioritaria
def guardar_prio(red):
    f = open("prioridad_wifi.txt", 'w')
    f.write(red)
    f.close()

#Leemos la red prioritaria
def leer_prio():
    f = open("prioridad_wifi.txt", 'r')
    red = f.readline()
    f.close()
    return red

#Borramos la red prioritaria
def borrar_prio():
    os.remove("prioridad_wifi.txt")
```

wlan_op.py – Funciones relacionadas con las redes y la conexión *Wi-Fi*

```
from network import WLAN
import time
import utime
import machine
import pycom
import f_op
import json
import os
from sens_op import leer_sensores
from umqtt import MQTTClient
import ubinascii

#Intentamos conectar con la red prioritaria
def conectar_prio(redes,prio_wifi):
    wlan = WLAN(mode=WLAN.STA)
    if prio_wifi != "":
        for i in redes:
            if prio_wifi == i["ssid"]:
                    print("Intentando conectar a " + i["ssid"] + "...")
                    wlan.connect(i["ssid"], auth=(WLAN.WPA2, i["psw"]), timeout=10000)
                    for Delay in range(0, 10000/50):
                        if not wlan.isconnected():
                            machine.idle()
                            time.sleep_ms(50)
                        else:
                            break
                    #Si funciona devolvemos True, si no, False
                    if wlan.isconnected():
                        print("Conectado a " + i["ssid"])
                        f_op.guardar_prio(i["ssid"])
                        return True
                    else:
                       print("Conexion con " + i["ssid"] + " fallida")
                       return False

#Intentamos conectar con el resto de redes
def conectar_red(redes,prio_wifi):
    wlan = WLAN(mode=WLAN.STA)
    for i in redes:
        if prio_wifi != i["ssid"]:
            print("Intentando conectar a " + i["ssid"] + "..." )
            wlan.connect(i["ssid"], auth=(WLAN.WPA2, i["psw"]), timeout=10000)
            for Delay in range(0, 10000/50):
                if not wlan.isconnected():
                    machine.idle()
                    time.sleep_ms(50)
                else:
                    break
            if wlan.isconnected():
                print("Conectado a " + i["ssid"])
                f_op.guardar_prio(i["ssid"])
                print("Nueva prioridad: " + i["ssid"])
                break
            else:
                print("Conexion con " + i["ssid"] + " fallida")
```

```
def crear_cliente(info_usuario):
    # Informacion de la IO de Adafruit
    dict_mqtt = {}
    dict_mqtt['server'] = "io.adafruit.com"
    dict_mqtt['puerto'] = 1883
    dict_mqtt['usuario'] = info_usuario["usuario"]
    dict_mqtt['clave'] = info_usuario["clave"]
    dict_mqtt['id_cliente'] = ubinascii.hexlify(machine.unique_id())
    dict_mqtt['f_datos_sensores'] = dict_mqtt['usuario'] + "/feeds/datos_sensores"
    dict_mqtt['f_temp'] = dict_mqtt['usuario'] + "/feeds/temp"
    dict_mqtt['f_lum'] = dict_mqtt['usuario'] + "/feeds/lum"
    dict_mqtt['f_bat'] = dict_mqtt['usuario'] + "/feeds/bat"
    dict_mqtt['f_hum'] = dict_mqtt['usuario'] + "/feeds/hum"
    # Creamos el cliente MQTT e intentamos conectarnos
    client = MQTTClient(dict_mqtt['id_cliente'], dict_mqtt['server'],
dict_mqtt['puerto'], dict_mqtt['usuario'], dict_mqtt['clave'])
    try:
        client.connect()
    except Exception as e:
        print("ERROR MQTT")
        pycom.heartbeat(False)
        for cycles in range(2):
            pycom.rgbled(0x7f0000)
            time.sleep(1)
        estado_cliente = [client,dict_mqtt,False]
    else:
        print("Conectado a la IO de Adafruit")
        estado_cliente = [client,dict_mqtt,True]
    #Devolvemos una lista con el cliente, un diccionario con la informacion de la IO y
una bandera que indica si se ha podido conectar a la IO
    pycom.rgbled(0x000000)
    return estado_cliente

def sinc_hora(flag_reset):
    #pool.ntp.org. Servidor más usado por sistemas operativos
    #hora.rediris.es (130.206.3.166). Servidor de RedIRIS, Madrid
    #hora.roa.es (150.214.94.5). Servidor de Real Instituto y Observatorio de la
Armada, San Fernando (Cádiz)
    NTP_server = ["pool.ntp.org", "hora.rediris.es", "hora.roa.es"]
    rtc = machine.RTC()
    hora = rtc.now()
    print("Hora antes de la sincronización:")
    print(hora)
    #Si obtenemos un año menor a 2019 (suele ser 1970 cuando se resetea), se activa la
bandera para indicarlo
    if hora[0] <= 2019:
        flag_reset = True
        for Server in range(0, len(NTP_server)):
            rtc.ntp_sync(NTP_server[Server])
            for Attempts in range(2):
                if not rtc.synced():
                    utime.sleep(2)
                else:
                    print("RTC Sync con server NTP {}".format(NTP_server[Server]))
    else:
        print("El dispositivo ya esta sincronizado")
    #rtc.now() nos da la hora en la zona horaria UTC, la pasamos a la nuestra, CET
    utime.timezone(+3600)
    hora = utime.localtime()
    lista_sinc = [hora,flag_reset]
    print("Hora tras sincronizarse:")
    print(hora)
    return lista_sinc
```

```
def guardar_datos(t_muestreo,flag_reset,hora):
    f = open('datos_no_enviados.txt', 'a')
    #Si la hora esta sincronizada, guardamos los datos con su respectiva hora
    if flag_reset == True and hora[0] >= 2020 or flag_reset == False and abs(hora[4] -
pycom.nvs_get('min')) >= int(t_muestreo) and pycom.nvs_get('seg') - hora[5] == 0:
        datos_sensores =leer_sensores()
        hora_segundos = utime.mktime(hora)
        hora_para_epoch = utime.localtime(hora_segundos - 14400)
        hora_epoch =  utime.mktime(hora_para_epoch)
        datos_sensores['EPOCH'] = hora_epoch
        datos_sensores['T_muestreo'] = t_muestreo
        mensaje_guardado = "{:02d}".format(hora[2]) + "/" + "{:02d}".format(hora[1]) +
"/" + "{:02d}".format(hora[0]) + " " + "{:02d}".format(hora[3]) + ":" +
"{:02d}".format(hora[4])  +  " " + json.dumps(datos_sensores)
        f.write(mensaje_guardado)
        f.write('\n')
        f.close()
        print("Se ha guardado el siguiente mensaje: " + mensaje_guardado)
        pycom.nvs_set('hora',hora[3])
        pycom.nvs_set('min',hora[4])
        pycom.nvs_set('seg',hora[5])
        return True
    #Si no esta sincronizada, los marcamos con un asterisco
    elif flag_reset == True and hora[0] <= 2020:
        datos_sensores =leer_sensores()
        datos_sensores['EPOCH'] = "*"
        datos_sensores['T_muestreo'] = t_muestreo
        f.write(json.dumps(datos_sensores))
        f.write('\n')
        f.close()
        print("Se ha guardado el siguiente mensaje: " + str(datos_sensores))
        return True

#Comprobamos si hay datos no enviados
def comprobar_almacen():
    try:
        f = open('datos_no_enviados.txt', 'r')
        line_read=f.readlines()
        print("Hay datos no enviados")
        f.close()
        return True
    except:
        print("No hay datos no enviados")
        return False
```

```
def envios_atrasados(client,hora,dict_mqtt):
    f = open('datos_no_enviados.txt', 'r')
    line_read=f.readlines()
    f.close()
    try:
        #Buscamos todos los datos no enviados
        for i in range(len(line_read)):
            try:
                #Si es un diccionario, se trata de los datos a los que no se les ha
adjuntado la fecha
                if isinstance(json.loads(line_read[i]),dict):
                    datos_sensores = json.loads(line_read[i])
                    #Conversion horaria para obtener EPOCH (UTC) en funcion de la hora
de sincronizacion
                    hora_segundos = utime.mktime(hora)
                    hora_para_epoch = utime.localtime(hora_segundos - 14400)
                    hora_epoch =  utime.mktime(hora_para_epoch)
                    hora_epoch_atrasada = hora_epoch -
(int(datos_sensores['T_muestreo']) * 60 * (len(line_read) - i))
                    #Deshacemos la conversion horaria
                    hora_atrasada_para_epoch = utime.localtime(hora_epoch_atrasada-
7200)
                    hora_atrasada_epoch = utime.mktime(hora_atrasada_para_epoch)
                    datos_sensores['EPOCH'] = hora_atrasada_epoch
                    hora_atrasada = utime.localtime(hora_atrasada_epoch)
                    #Le adjuntamos finalmente la hora estimada
                    mensaje = "{:02d}".format(hora_atrasada[2]) + "/" +
"{:02d}".format(hora_atrasada[1]) + "/" + "{:02d}".format(hora_atrasada[0]) + " " +
"{:02d}".format(hora_atrasada[3]) + ":" + "{:02d}".format(hora_atrasada[4])  +  " " +
json.dumps(datos_sensores)
                    client.publish(topic=dict_mqtt['f_datos_sensores'],msg=mensaje,qos
= 1)
                    print("Publicando mensaje atrasado* : " + str(mensaje))
                    time.sleep_ms(1000)
            except ValueError:
                #Si no es un diccionario, ya tiene la fecha, podemos enviarlo
directamente

client.publish(topic=dict_mqtt['f_datos_sensores'],msg=line_read[i],qos = 1)
                print("Publicando mensaje atrasado : " + str(line_read[i]))
                time.sleep_ms(1000)
        return True
    except:
        print("Error al enviar mensajes atrasados")
        return False

#Borramos los datos no enviados
def borrar_datos():
    os.remove('datos_no_enviados.txt')
```

```
#Envio de los datos actuales
def enviar_datos(client,dict_mqtt,hora,datos_sensores,t_muestreo):
    hora_segundos = utime.mktime(hora)
    hora_para_epoch = utime.localtime(hora_segundos - 14400)
    hora_epoch =  utime.mktime(hora_para_epoch)
    datos_sensores['EPOCH'] = hora_epoch
    datos_sensores['T_muestreo'] = t_muestreo
    mensaje = "{:02d}".format(hora[2]) + "/" + "{:02d}".format(hora[1]) + "/" +
"{:02d}".format(hora[0]) + " " + "{:02d}".format(hora[3]) + ":" +
"{:02d}".format(hora[4]) + " " + json.dumps(datos_sensores)
    try:
        print("Enviando datos a " + dict_mqtt['f_datos_sensores'] + "..")
        print("Enviando datos a " + dict_mqtt['f_temp'] + "..")
        temp = str(datos_sensores['Temp'])
        print("Enviando datos a " + dict_mqtt['f_lum'] + "..")
        lum = str(datos_sensores['Lum'])
        print("Enviando datos a " + dict_mqtt['f_bat'] + "..")
        bat = str(datos_sensores['Bat'])
        print("Enviando datos a " + dict_mqtt['f_hum'] + "..")
        hum = str(datos_sensores['Hum'])
        #Datos individuales y actuales
        client.publish(topic=dict_mqtt['f_temp'],msg=temp,qos = 1)
        client.publish(topic=dict_mqtt['f_lum'],msg=lum,qos = 1)
        client.publish(topic=dict_mqtt['f_bat'],msg=bat,qos = 1)
        client.publish(topic=dict_mqtt['f_hum'],msg=hum,qos = 1)
        #Resumen de datos con hora de medida
        print("Publicando mensaje : " + mensaje)
        client.publish(topic=dict_mqtt['f_datos_sensores'],msg=mensaje,qos = 1)
    #Si hay un error de envio, se guardan los datos
    except Exception as e:
        print("ERROR de envio, guardando los datos")
        f = open('datos_no_enviados.txt', 'a')
        f.write(mensaje)
        f.write('\n')
        f.close()
        pycom.heartbeat(False)
        for cycles in range(2):
            pycom.rgbled(0x7f0000)
            time.sleep(1)
    else:
        pycom.nvs_set('hora',hora[3])
        pycom.nvs_set('min',hora[4])
        pycom.nvs_set('seg',hora[5])

#Se estima el tiempo del deepsleep en funcion de las redes a las que se tenga que
conectar y a los envios atrasados
def dormir_wipy(t_muestreo,redes):
    envios_atrasados = 0
    if comprobar_almacen():
        f = open('datos_no_enviados.txt', 'r')
        line_read=f.readlines()
        f.close()
        envios_atrasados = len(line_read)
    machine.deepsleep(int(t_muestreo) * 60000 - (len(redes) * 12000 + 15000 +
envios_atrasados * 3000))
```

umqtt.py – Clase de cliente *MQTT*

```
#Fuente: https://github.com/micropython/micropython-lib/tree/master/umqtt.simple
import usocket as socket
import ustruct as struct
from ubinascii import hexlify

class MQTTException(Exception):
    pass

class MQTTClient:

    def __init__(self, client_id, server, port=0, user=None, password=None,
keepalive=0,
                 ssl=False, ssl_params={}):
        if port == 0:
            port = 8883 if ssl else 1883
        self.client_id = client_id
        self.sock = None
        self.server = server
        self.port = port
        self.ssl = ssl
        self.ssl_params = ssl_params
        self.pid = 0
        self.cb = None
        self.user = user
        self.pswd = password
        self.keepalive = keepalive
        self.lw_topic = None
        self.lw_msg = None
        self.lw_qos = 0
        self.lw_retain = False

    def _send_str(self, s):
        self.sock.write(struct.pack("!H", len(s)))
        self.sock.write(s)

    def _recv_len(self):
        n = 0
        sh = 0
        while 1:
            b = self.sock.read(1)[0]
            n |= (b & 0x7f) << sh
            if not b & 0x80:
                return n
            sh += 7

    def set_callback(self, f):
        self.cb = f

    def set_last_will(self, topic, msg, retain=False, qos=0):
        assert 0 <= qos <= 2
        assert topic
        self.lw_topic = topic
        self.lw_msg = msg
        self.lw_qos = qos
        self.lw_retain = retain
```

```
def connect(self, clean_session=True):
    self.sock = socket.socket()
    addr = socket.getaddrinfo(self.server, self.port)[0][-1]
    self.sock.connect(addr)
    if self.ssl:
        import ussl
        self.sock = ussl.wrap_socket(self.sock, **self.ssl_params)
    premsg = bytearray(b"\x10\0\0\0\0\0")
    msg = bytearray(b"\x04MQTT\x04\x02\0\0")

    sz = 10 + 2 + len(self.client_id)
    msg[6] = clean_session << 1
    if self.user is not None:
        sz += 2 + len(self.user) + 2 + len(self.pswd)
        msg[6] |= 0xC0
    if self.keepalive:
        assert self.keepalive < 65536
        msg[7] |= self.keepalive >> 8
        msg[8] |= self.keepalive & 0x00FF
    if self.lw_topic:
        sz += 2 + len(self.lw_topic) + 2 + len(self.lw_msg)
        msg[6] |= 0x4 | (self.lw_qos & 0x1) << 3 | (self.lw_qos & 0x2) << 3
        msg[6] |= self.lw_retain << 5

    i = 1
    while sz > 0x7f:
        premsg[i] = (sz & 0x7f) | 0x80
        sz >>= 7
        i += 1
    premsg[i] = sz

    self.sock.write(premsg, i + 2)
    self.sock.write(msg)
    #print(hex(len(msg)), hexlify(msg, ":"))
    self._send_str(self.client_id)
    if self.lw_topic:
        self._send_str(self.lw_topic)
        self._send_str(self.lw_msg)
    if self.user is not None:
        self._send_str(self.user)
        self._send_str(self.pswd)
    resp = self.sock.read(4)
    assert resp[0] == 0x20 and resp[1] == 0x02
    if resp[3] != 0:
        raise MQTTException(resp[3])
    return resp[2] & 1

def disconnect(self):
    self.sock.write(b"\xe0\0")
    self.sock.close()

def ping(self):
    self.sock.write(b"\xc0\0")
```

```
def publish(self, topic, msg, retain=False, qos=0):
        pkt = bytearray(b"\x30\0\0\0")
        pkt[0] |= qos << 1 | retain
        sz = 2 + len(topic) + len(msg)
        if qos > 0:
            sz += 2
        assert sz < 2097152
        i = 1
        while sz > 0x7f:
            pkt[i] = (sz & 0x7f) | 0x80
            sz >>= 7
            i += 1
        pkt[i] = sz
        #print(hex(len(pkt)), hexlify(pkt, ":"))
        self.sock.write(pkt, i + 1)
        self._send_str(topic)
        if qos > 0:
            self.pid += 1
            pid = self.pid
            struct.pack_into("!H", pkt, 0, pid)
            self.sock.write(pkt, 2)
        self.sock.write(msg)
        if qos == 1:
            while 1:
                op = self.wait_msg()
                if op == 0x40:
                    sz = self.sock.read(1)
                    assert sz == b"\x02"
                    rcv_pid = self.sock.read(2)
                    rcv_pid = rcv_pid[0] << 8 | rcv_pid[1]
                    if pid == rcv_pid:
                        return
        elif qos == 2:
            assert 0

    def subscribe(self, topic, qos=0):
        assert self.cb is not None, "Subscribe callback is not set"
        pkt = bytearray(b"\x82\0\0\0")
        self.pid += 1
        struct.pack_into("!BH", pkt, 1, 2 + 2 + len(topic) + 1, self.pid)
        #print(hex(len(pkt)), hexlify(pkt, ":"))
        self.sock.write(pkt)
        self._send_str(topic)
        self.sock.write(qos.to_bytes(1, "little"))
        while 1:
            op = self.wait_msg()
            if op == 0x90:
                resp = self.sock.read(4)
                #print(resp)
                assert resp[1] == pkt[2] and resp[2] == pkt[3]
                if resp[3] == 0x80:
                    raise MQTTException(resp[3])
                return
```

```
# Wait for a single incoming MQTT message and process it.
    # Subscribed messages are delivered to a callback previously
    # set by .set_callback() method. Other (internal) MQTT
    # messages processed internally.
    def wait_msg(self):
        res = self.sock.read(1)
        self.sock.setblocking(True)
        if res is None:
            return None
        if res == b"":
            raise OSError(-1)
        if res == b"\xd0":  # PINGRESP
            sz = self.sock.read(1)[0]
            assert sz == 0
            return None
        op = res[0]
        if op & 0xf0 != 0x30:
            return op
        sz = self._recv_len()
        topic_len = self.sock.read(2)
        topic_len = (topic_len[0] << 8) | topic_len[1]
        topic = self.sock.read(topic_len)
        sz -= topic_len + 2
        if op & 6:
            pid = self.sock.read(2)
            pid = pid[0] << 8 | pid[1]
            sz -= 2
        msg = self.sock.read(sz)
        self.cb(topic, msg)
        if op & 6 == 2:
            pkt = bytearray(b"\x40\x02\0\0")
            struct.pack_into("!H", pkt, 2, pid)
            self.sock.write(pkt)
        elif op & 6 == 4:
            assert 0

    # Checks whether a pending message from server is available.
    # If not, returns immediately with None. Otherwise, does
    # the same processing as wait_msg.
    def check_msg(self):
        self.sock.setblocking(False)
        return self.wait_msg()
```

dth.py – Funciones de los sensores DHT en *Pycom*

```
#https://github.com/rbraggaar/sensor-city-
delft/blob/master/temp_humidity_am2302/lib/dth.py
import time
import pycom
from machine import enable_irq, disable_irq,  Pin

class DTHResult:
    'DHT sensor result returned by DHT.read() method'

    ERR_NO_ERROR = 0
    ERR_MISSING_DATA = 1
    ERR_CRC = 2

    error_code = ERR_NO_ERROR
    temperature = -1
    humidity = -1

    def __init__(self, error_code, temperature, humidity):
        self.error_code = error_code
        self.temperature = temperature
        self.humidity = humidity

    def is_valid(self):
        return self.error_code == DTHResult.ERR_NO_ERROR

class DTH:
    'DHT sensor (dht11, dht21,dht22) reader class for Pycom'

    #__pin = Pin('P3', mode=Pin.OPEN_DRAIN)
    __dhttype = 0

    def __init__(self, pin, sensor=0):
        self.__pin = Pin(pin, mode=Pin.OPEN_DRAIN)
        self.__dhttype = sensor
        self.__pin(1)
        time.sleep(1.0)

    def read(self):
        # pull down to low
        self.__send_and_sleep(0, 0.019)
        data = pycom.pulses_get(self.__pin,100)
        self.__pin.init(Pin.OPEN_DRAIN)
        self.__pin(1)
        #print(data)
        bits = []
        for a,b in data:
            if a ==1 and 18 <= b <= 28:
                bits.append(0)
            if a ==1 and 65 <= b <= 75:
                bits.append(1)
        #print("longueur bits : %d " % len(bits))
        if len(bits) != 40:
            return DTHResult(DTHResult.ERR_MISSING_DATA, 0, 0)
        #print(bits)
        # we have the bits, calculate bytes
        the_bytes = self.__bits_to_bytes(bits)
        # calculate checksum and check
        checksum = self.__calculate_checksum(the_bytes)
        if the_bytes[4] != checksum:
            return DTHResult(DTHResult.ERR_CRC, 0, 0)
        # ok, we have valid data, return it
        [int_rh, dec_rh, int_t, dec_t, csum] = the_bytes
        if self.__dhttype==0:       #dht11
            rh = int_rh          #dht11 20% ~ 90%
            t = int_t   #dht11 0..50°C
```

```
        else:            #dht21,dht22
            rh = ((int_rh * 256) + dec_rh)/10
            t = (((int_t & 0x7F) * 256) + dec_t)/10
            if (int_t & 0x80) > 0:
                t *= -1
        return DTHResult(DTHResult.ERR_NO_ERROR, t, rh)


    def __send_and_sleep(self, output, mysleep):
        self.__pin(output)
        time.sleep(mysleep)

    def __bits_to_bytes(self, bits):
        the_bytes = []
        byte = 0

        for i in range(0, len(bits)):
            byte = byte << 1
            if (bits[i]):
                byte = byte | 1
            else:
                byte = byte | 0
            if ((i + 1) % 8 == 0):
                the_bytes.append(byte)
                byte = 0
        #print(the_bytes)
        return the_bytes

    def __calculate_checksum(self, the_bytes):
        return the_bytes[0] + the_bytes[1] + the_bytes[2] + the_bytes[3] & 255
```

sens_op.py – Funciones relacionadas con las mediciones de los sensores

```
import pycom, json, time
from machine import Pin,ADC
from dth import DTH

def leer_sensores():

    datos={}

    Activacion_3v=Pin('P9',mode=Pin.OUT)
    Activacion_3v.hold(False)
    Activacion_3v(False)                                #Activo el transistor de
alimentacion en Pin9

    th = DTH('P11',1)                                   # Pin del sensor de humedad y
temperatura 11
    adc = ADC(0)                                        # ADC - Analog to Digital
Conversion, para el LDR y la bateria

    LDR = adc.channel(pin='P19',attn=ADC.ATTN_0DB) #Pin LDR
    porcentaje_LDR=abs(100-int(LDR.voltage()/10))
    datos['Lum']='{:3.2f}'.format(porcentaje_LDR)
    Bateria= adc.channel(pin='P16',attn=ADC.ATTN_11DB) #Pin bateria
    iteracciones=0
    Valor_Bateria=[]
    for x in range(20):     #Valores de tension en la bateria en una lista
        time.sleep_ms(20)
        Valor_Bateria.append(Bateria.voltage())
    mediana=calcular_mediana(Valor_Bateria)
    Volts_Bateria='{:3.2f}'.format(mediana/325)
    result = th.read()
    if result.is_valid():
        datos['Temp']='{:3.2f}'.format(result.temperature/1.0)
        datos['Hum']='{:3.2f}'.format(result.humidity/1.0)

    else:
        datos['Temp']=0
        datos['Hum']=0
    #Devolvemos los datos medidos
    datos['Bat']=Volts_Bateria
    datos_json=json.dumps(datos)
    Activacion_3v(True)
    Activacion_3v.hold(True)
    return(datos)

#Se estima lo que queda de bateria a traves de la mediana de la lista de los valores
de bateria
def calcular_mediana(lista):
    lista.sort()
    if len(lista) % 2 == 0:
        n = len(lista)
        mediana = (lista[int(n/2)-1]+lista[int(n/2)])/2
    else:
        mediana =lista[len(lista)/2]
    return (mediana)
```