Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Recolección de datos de sensores en la IoT con Arduino y su gestión mediante un protocolo de mensajes basado en el patrón publicación y subscripción

Autor: Dolores Raigada Romero
Tutor: Antonio Jesús Sierra Collado

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Recolección de datos de sensores en la IoT con Arduino y su gestión mediante un protocolo de mensajes basado en el patrón publicación y subscripción

Autor:
Dolores Raigada Romero

Tutor:
Antonio Jesús Sierra Collado

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Grado: Recolección de datos de sensores en la IoT con Arduino y su gestión mediante un protocolo de mensajes basado en el patrón publicación y subscripción

Autor: Dolores Raigada Romero

Tutor: Antonio Jesús Sierra Collado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A mi familia

A mis amigos

Agradecimientos

A mis padres, por hacer que esto sea posible. Gracias por darme todo lo que he necesitado y empujarme siempre a seguir hacia delante. Vuestro empeño a lo largo de todos mis años como estudiante han hecho que haya llegado a donde estoy ahora y, sobre todo, que sea quien soy hoy en día.

A mi hermana, por ser mi ejemplo a seguir. Por meterme en el mundo de la ingeniería y ayudarme cada vez que ha sido posible. Me has acompañado más que nunca a lo largo de mi etapa universitaria y, por supuesto, esto también es gracias a ti.

A toda la gente con la que me he cruzado en estos años y han hecho que fuese todo un poco más fácil.

A mis amigos de siempre y los que he hecho por el camino, gracias por apoyarme y confiar siempre en mí.

Dolores Raigada Romero

Sevilla, 2019

Resumen

Hoy día, vivimos en un mundo "conectado" en el que necesitamos estar al tanto de lo que ocurre a nuestro alrededor, o incluso en lugares no tan cercanos.

El Internet de las Cosas (IoT) es un concepto que define a objetos interconectados a través de internet permitiendo la gestión de los mismos. Hay tipos de información fácilmente accesibles para cualquiera que desee obtener alguna actualización, pero existe la necesidad de realizar una búsqueda para estar al corriente y, una vez llegados al dato, puede que ya haya cambiado su valor. El modelo de publicación/subscripción intenta hacer desaparecer este tipo de problemas, gracias a que el usuario tiene la posibilidad de subscribirse a los temas que considere de interés y recibir los datos actualizados una vez que cualquier otro usuario ha modificado su valor. De esta forma, el usuario subscrito obtiene la información actualizada en el mismo instante, sin necesidad de realizar ninguna petición.

Este modelo asíncrono de envío de mensajes se basa en dos elementos: cliente y *broker*. El cliente se subscribe al *broker* con un *topic* específico, en el cual puede publicar mensajes o recibirlos si otro cliente publica en el mismo *topic*. El *broker* es un servidor en la nube encargado de manejar los mensajes recibidos de los clientes. Es por esto que se considera un intermediario entre clientes, ya que nunca habrá conexión directa entre ambos.

A partir de este modelo, se ha desarrollado el proyecto que se presenta a continuación. En él, se expone la elección del protocolo de publicación/subscripción adecuado para la comunicación entre dos clientes finales: un Arduino y una aplicación. Los protocolos más comunes para la gestión de dispositivos de IoT a través del modelo de publicación/subscripción son XMPP, AMQP, MQTT y ZeroMQ; aunque, como se discute en la memoria, MQTT es el más adecuado para este proyecto. Este protocolo facilita la comunicación con el servidor para que, de esta forma, la aplicación pueda obtener los datos de los sensores conectados al Arduino, y realizar una fácil gestión de los mismos.

Abstract

Nowadays, we live in a “connected” world in which we need to be aware of what is happening around us, or even in places not so close.

The Internet of Things (IoT) is a concept which defines interconnected devices through the Internet allowing its management. There are some easily accessible types of information for whoever that wants to get some updates, but there is a need to perform a search to keep up to date and, once we get to the data, it may have already changed its value. The publish/subscribe model tries to resolve these kinds of problems, thanks to the fact the user has the possibility to be subscribed to the topics of interest and to receive the updated data once any other user has modified its value. This way, the subscribed user obtains the updated information at the precise instant it has been changed, without the need of making any request.

This asynchronous message sending model is based on two elements: client and broker. The client subscribes to the broker in a specific topic, in which the client is able to publish messages o receive them if another client publishes in the same topic. The broker is a cloud server in charge of managing messages from clients. This is why it is considered as an intermediary between clients, since there will never be a direct connection between them.

From this model, the project presented below has been developed. It describes the selection of the appropriate publish/subscribe protocol for communication between two clients: an Arduino and an application. The most common protocols for managing IoT devices through publish/subscribe model are XMPP, AMQP, MQTT y ZeroMQ, although, as it is discussed in the thesis, MQTT is the most suitable for this project. This protocol facilitates communication with the server so that, in this way, the application can obtain data from the sensors connected to the Arduino, and perform an easy management of them.

ÍNDICE

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

ÍNDICE **xv**

Índice de Ilustraciones **xvii**

Índice de Tablas **xix**

Acrónimos **xxi**

1 Objetivos **23**

2 Introducción **25**

2.1 Protocolos de publicación/subscripción *25*

2.2 Descripción de la solución *27*

2.2.1. Uso de CloudMQTT 27

2.2.2. Disposición de los datos en *dashboard* 27

3 Tecnología Utilizada **29**

3.1 Recursos hardware *29*

3.1.1 Ordenador Portátil ASUS K55A 29

3.1.2 Arduino UNO 29

3.1.3 Arduino Ethernet Shield 31

3.1.4 Sensor MPU-6050 31

3.1.5 Sensor NEO-6M 32

3.2 Recursos Software *32*

3.2.1 MQTT Cloud 32

3.2.2 IoT MQTT Panel 32

3.2.3 Arduino IDE 32

4 Aplicación Desarrollada **33**

4.1 Sensores *34*

4.1.1 MPU-6050 34

4.1.2 Módulo GPS NEO-6M 35

4.2 CloudMQTT *36*

4.2.1 Protocolo MQTT 37

4.2.2 Conexión 37

4.2.3 Modelo Publish/Subscribe 39

4.2.4 Plataforma CloudMQTT 43

4.3 IoT MQTT Panel *43*

4.3.1 *Dashboard* 43
4.3.2 Aplicación 44
4.4 Desarrollo 44
4.4.1 Lectura y envío de datos de los sensores 44
4.4.2 Broker 49
4.4.3 Presentación de datos 53

5 Validación 57
5.1 Ejecución 57
5.2 Resultados 58

6 Conclusiones y líneas futuras 61
6.1 Conclusiones 61
6.2 Líneas futuras 62

Referencias 65

Anexos 69

ÍNDICE DE ILUSTRACIONES

Ilustración 1: Presentación del escenario propuesto para la solución 24
Ilustración 2: Arduino UNO 30
Ilustración 3: Ethernet Shield 31
Ilustración 4: Módulo MPU-6050 31
Ilustración 5: Módulo GPS NEO-6M 32
Ilustración 6: Esquema de comunicación de los elementos del sistema 33
Ilustración 7: Ejes en el MPU-6050 34
Ilustración 8: Movimiento angular 35
Ilustración 9: Antena con conector U.FL para comunicación del módulo NEO-6M 36
Ilustración 10: Pila de protocolos de los dispositivos MQTT 37
Ilustración 11: Paso de mensajes SUBSCRIBE/PUBLISH 41
Ilustración 12: Paso de mensajes cancelación de subscripción 43
Ilustración 13: Esquema de conexión completo 44
Ilustración 14: Configuración acceso a internet 45
Ilustración 15: Conexión completa entre Arduino y sensores 45
Ilustración 16: Esquema de conexión MP6050 con Arduino 46
Ilustración 17: Configuración MPU 46
Ilustración 18: Lectura de datos MPU-6050 46
Ilustración 19: Conexión MPU con Arduino 47
Ilustración 20: Esquema de conexión NEO-6M con Arduino 47
Ilustración 21: Configuración NEO-6M 48
Ilustración 22: Lectura de datos GPS 48
Ilustración 23: Tratamiento de datos GPS para la localización 48
Ilustración 24: Conexión Arduino con GPS 49
Ilustración 25: Página de acceso a CloudMQTT. 49
Ilustración 26: Página principal con listado de instancias creadas. 50
Ilustración 27: Paso 1 para crear nueva instancia 50
Ilustración 28: Selección de región para la instancia 51
Ilustración 29: Información de la instancia creada en CloudMQTT 51
Ilustración 30: Creación cliente MQTT 51

Ilustración 31: Conexión al *broker* MQTT desde el Arduino 52
Ilustración 32: Subscripción a los *topics* desde el Arduino 52
Ilustración 33: Publicación de la temperatura obtenida en el *topic* 'temperature' 53
Ilustración 34: Página principal IoT MQTT Panel y nueva conexión 54
Ilustración 35: Configuración avanzada de la conexión 54
Ilustración 36: Creación Panel Gauge 55
Ilustración 37: Configuración panel gráficos 55
Ilustración 38: Configuración panel URI y *dashboard* final. 56
Ilustración 39: Cargar programa en el Arduino 57
Ilustración 40: Datos recogidos en la aplicación móvil 58
Ilustración 41: Localización obtenida gracias al módulo NEO-6M 59

ÍNDICE DE TABLAS

Tabla 1: Comparación entre los protocolos de publicación/subscripción 26
Tabla 2: Contenido mensaje CONNECT 38
Tabla 3: Contenido mensaje CONNACK 39
Tabla 4: Códigos de retorno del mensaje CONNACK 39
Tabla 5: Contenido mensaje PUBLISH 41
Tabla 6: Contenido mensaje SUBSCRIBE 41
Tabla 7: Contenido mensaje SUBACK 42
Tabla 8: Códigos de retorno del mensaje SUBACK 42
Tabla 9: Contenido mensaje UNSUBSCRIBE 42

Acrónimos

TTFF	Time To First Fix, Tiempo para la primera posición
IoT	Internet of Things, Internet de las Cosas
GPS	Global Positioning System, Sistema de Posicionamiento Global
U.FL	Conector coaxial en miniatura para señales de alta frecuencia
GB	Gigabyte
KB	Kilobyte
MHz	Megahercio
GHz	Gigahercio
CPU	Central Processing Unit, Unidad de Procesamiento Central
RAM	Random Access Memory, Memoria de Acceso Aleatorio
PWM	Pulse Width Modulation, Modulación de Ancho de Pulso
SPI	Serial Peripheral Interface, Interfaz Periférica en Serie
SDA	Serial Data
SCL	Serial Clock
ISO	International Organization for Standardization, Organización Internacional de Normalización
OASIS	Organization for the Advancement of Structured Information Standards
Pub	Publish
Sub	Subscribe
dV	Derivada de la velocidad
dt	Derivada del tiempo
dϴ	Derivada del desplazamiento angular
bps	Baudios por Segundo
QoS	Quality of Service, Calidad del Servicio.
UART	Asynchronous Receiver-Transmitter, Transmisor-Receptor Asíncrono Universal
PSM	Power Saving Mode, Modo de Ahorro de Energía
XMPP	Extensible Messaging and Presence Protocol
AMQP	Advanced Message Queuing Protocol
MQTT	Message Queue Telemetry Transport
AWS	Amazon Web Services

1 OBJETIVOS

Muere lentamente quien no se atreve a abandonar lo seguro para luchar por sus sueños.

Jesús Quintero

Hoy en día la tecnología se encuentra avanzando a un alto nivel, lo que supone importantes mejoras en nuestra calidad de vida y, por tanto, en nuestra sociedad, gracias a objetos conectados que continuamente se comunican entre ellos y estos, a su vez, se conectan con nosotros. En este primer capítulo se exponen los objetivos marcados en este proyecto relacionado con un tema actual como lo es el IoT.

El término de Internet de las Cosas (IoT por sus siglas en inglés) proviene del concepto dado por Kevin Ashton, el cual lo define como objetos conectados a internet e interconectados entre ellos, principalmente a través de sensores que remiten y reciben datos de una manera continua para, a partir de estos y de su interpretación, proceder a la ejecución de acciones [9].

Resulta de especial interés que simplemente con una aplicación podamos controlar elementos de nuestra vida diaria, a los que quizás en ese momento no tengamos acceso. Es por eso por lo que se ha llevado a cabo este proyecto. Me fascina hasta dónde puede llegar la capacidad de un dispositivo conectado a internet, gracias al cual se consigue tener información de distintos sensores y poder saber su valor desde cualquier parte del mundo, simplemente teniendo conexión a internet y la aplicación móvil con la configuración correcta.

El modelo Publicación/Subscripción es un estilo de mensajería en la que el emisor del mensaje no tiene visión directa con el receptor del mismo. En este modelo hay un intermediario encargado de recibir todos los mensajes y enviarlos al destinatario en cuestión [24].

Como objetivo principal, se pretende crear un sistema compuesto por un Arduino UNO al que conectar varios sensores, y configurarlos adecuadamente para poder obtener los datos tras la lectura de ambos. Una vez recibidos dichos datos, se pretende realizar publicaciones constantes hacia la nube, con el objetivo de que un cliente subscrito a esta pueda obtener el valor de los sensores en tiempo real. Estos datos se pretenden disponer en un panel de control a modo visual, desde una aplicación móvil que sea accesible en cualquier momento, mientras que el dispositivo presente conexión a la red de internet.

De este modo, los objetivos a realizar en este proyecto se disponen en el siguiente orden:

1. Obtener los datos de los sensores MPU-6050 y NEO-6M, para ello se necesita que ambos se encuentren correctamente conectados al Arduino y comenzar la comunicación adaptada a cada uno de ellos. Tras esto, se obtendrán los datos de un acelerómetro, un giroscopio, un sensor de temperatura y un GPS.

2. Conexión a internet por parte del Arduino configurándolo de manera adecuada para que presente conexión a la red.
3. Creación de instancia del *broker*, para poder administrar los mensajes de publicación/subscripción.
4. Conexión al *broker* desde el Arduino, funcionando como cliente.
5. Subscripción a los *topics* necesarios, uno para cada valor de sensor que vayamos a necesitar.
6. Publicación de los datos leídos en el *topic* creado para ese campo.
7. Creación de un *dashboard* en el que disponer los datos, con el *widget* más adecuado para cada valor.
8. Subscripción de la aplicación de paneles al *broker*, para poder recibir la información enviada por el otro cliente.
9. Visualización de la lectura de los sensores a través del *dashboard*.

En la ilustración 1 se muestra un esquema de la solución escogida en el que podemos apreciar tres componentes claramente diferenciados. A continuación, se realiza una breve descripción de cada uno de ellos, entrando en más detalle en el capítulo 4:

- El Arduino es conectado a su componente esencial en este proyecto, el *Ethernet Shield*, y a dos sensores (no mostrados en el esquema), el MPU-6050 y NEO-6M.
- El segundo elemento es un servidor en la nube al cual se conectará el Arduino para compartir sus datos. Este servidor admite mensajes de publicación y subscripción.
- El último elemento es una aplicación usada para disponer los datos de los sensores leídos visualmente. Esta aplicación está subscrita al servidor al partir del cual le llega la información a mostrar.

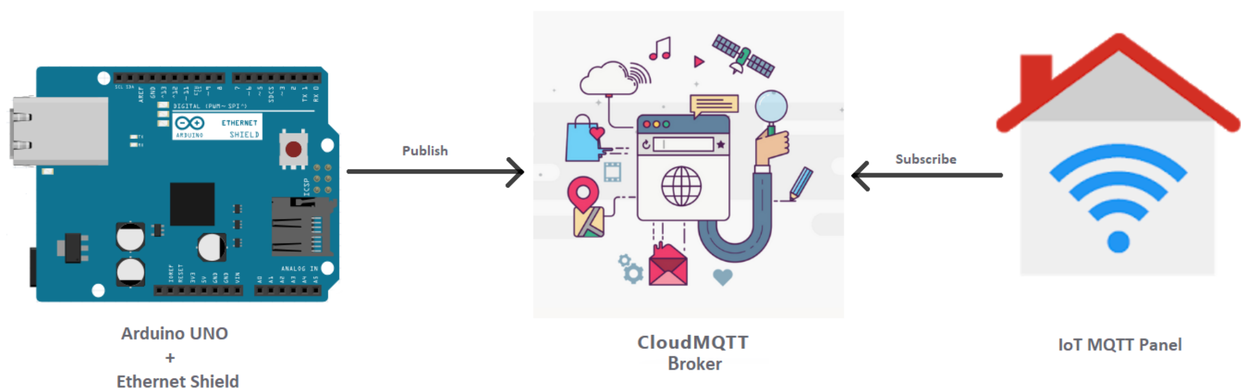

Ilustración 1: Presentación del escenario propuesto para la solución

En este capítulo hemos podido conocer los objetivos propuestos para la correcta realización de este trabajo. En el siguiente capítulo se realiza una introducción al protocolo elegido para su implementación, así como una presentación de la solución planteada para una aplicación dedicada al sistema propuesto.

2 INTRODUCCIÓN

La mejor organización no asegura los resultados. Pero una estructura equivocada sería garantía de fracaso.

Peter Drucker

En este capítulo se presentarán los protocolos de publicación/subscripción y sus características, para una correcta elección del protocolo más adecuado. Además de esto, se presenta la descripción de la solución propuesta para los objetivos marcados en el capítulo 1.

2.1 Protocolos de publicación/subscripción

Este proyecto se desea llevar a cabo mediante un protocolo de mensajes basado en el patrón publicación/subscripción. Como se explicará en capítulos posteriores, este tipo de envío de mensajes consiste en la comunicación entre un servidor y un cliente, el cual publica mensajes a un tema específico o recibe los mensajes que son publicados por otros clientes en dicho tema.

De este modo, presentaremos varios protocolos que siguen el modelo en cuestión pensando en su futura aplicación a IoT: XMPP, AMQP, MQTT y ZeroMQ [25].

- XMPP es un protocolo de mensajería extensible (Extensible Messaging and Presence Protocol) basado en el intercambio de datos XML. Fue diseñado para mensajería instantánea, pero al ser extensible, presenta soporte para VoIP. Su arquitectura es descentralizada, por lo que no depende de un servidor central. Es el protocolo detrás del servidor de mensajería instantánea Ejabberd.
- AMQP es un protocolo de cola de mensajes avanzado (Advanced Message Queuing Protocol) usado en implementaciones populares como RabbitMQ. Está orientado a mensajes, realiza encolamiento tanto punto a punto como publicación/subscripción presentando, a su vez, exactitud y seguridad en el paso de mensajes.
- MQTT (Message Queue Telemetry Transport) es un protocolo puro de publicación/subscripción para dispositivos restringidos y para redes de bajo ancho de banda, alta latencia y poco fiables. Está desarrollado por IBM y estandarizado por OASIS. Tiene varias implementaciones en código abierto como el Mosquitto *broker* y su librería de cliente.
- ZeroMQ es una biblioteca de mensajería que ofrece una API de socket con patrones de mensajería bastante avanzados. Soporta tanto los modelos petición/respuesta como de publicación/subscripción. Esta biblioteca de mensajería permite agregar nuevas funciones al protocolo central.

En la tabla 1 podemos ver una comparación entre los protocolos nombrados anteriormente. En la misma es posible observar si presentan alguna de las características más comunes de los modelos de publicación/subscripción.

CARACTERÍSTICAS		AMQP	MQTT	XMPP	ZeroMQ
Patrón de mensajes	Pub/Sub	✓	✓	✓	✓
	Punto a punto	✓		✓	✓
Filtro	Basado en *topic*	✓	✓	✓	✓
	Basado en contenido				
Topología	Descentralizado				✓
	Centralizado	✓	✓	✓	✓
	Híbrido			✓	
Formato del mensaje	*Payload*	✓	✓	✓	✓
	Codificación binaria	✓	✓		✓

Tabla 1: Comparación entre los protocolos de publicación/subscripción

Tras una exhaustiva búsqueda de sus características, llegamos a la conclusión de que ninguno de estos protocolos ofrece todas las características deseadas en una configuración de IoT. Sin embargo, existen diferencias significativas entre ellos. Aunque XMPP es un protocolo abierto extensible, no consigue alcanzar el comportamiento observado en los protocolos alternativos. Además, el servidor necesita que cada sentencia sea analizada para tomar una decisión en cada ruta a tomar. Debido a estos hechos, el protocolo XMPP se comporta de peor forma cuando hablamos del *throughput* y el retraso acarreado. AMQP es un middleware orientado a mensajes que presenta todas las funciones necesarias para la creación de un *broker* con mensajería IoT disponible. Su rendimiento con respecto al *throughput* y el retraso medio, es inferior al de otros protocolos como ZeroMQ y MQTT cuando se trata de un bajo nivel de mensajes. MQTT está específicamente diseñado para transportar datos de sensores. Su implementación en el *broker* Mosquitto, hace que presente un *throughput* altamente razonable y un bajo retraso. Por otro lado, ZeroMQ puede alcanzar niveles muy altos de *throughput* a la vez que mantiene un retraso medio bajo, sin importar el tamaño del paquete. Sin embargo, ZeroMQ no presenta una implementación directa de *broker* con todas las funciones necesarias, y es menos extensible que los otros protocolos, ya que solo admite la coincidencia de prefijos. Dadas estas características y debido a que los paquetes a enviar a través del protocolo no son de gran tamaño, podemos concluir con que el protocolo de publicación/subscripción más adecuado para este proyecto es MQTT.

2.2 Descripción de la solución

Como se ha comentado en el capítulo anterior, se desea realizar una comunicación entre el Arduino y la nube. Para llevar esto a cabo, Arduino dispone de una placa con puerto ethernet acoplable a la placa principal, y de otra placa, también acoplable, con tecnología Wi-Fi. Ambas placas soportan la lectura o escritura de una tarjeta SD, gracias a su puerto correspondiente. Si comparamos ambos dispositivos, llegamos a la conclusión de que la opción que más se adapta a nuestras necesidades es la placa con puerto Ethernet. Este dispositivo ofrece comunicación fiable por TCP presentando, a su vez, un rendimiento decente sin la aparición del llamado "cuello de botella".

La obtención de datos de los sensores se realiza por comunicación I^2C y UART. Su lectura se desarrolla mediante un código escrito en lenguaje C e implementado en el Arduino gracias a su software de código abierto, Arduino IDE.

El sensor MPU-6050 presenta un acelerómetro y un giroscopio de tres ejes cada uno, el cual permite gracias a su avanzada tecnología, añadir como entrada un compás de otros tres ejes para proporcionar una salida MotionFusion™ completa de nueve ejes.

El otro sensor es un módulo GPS capaz de proporcionar la longitud y latitud del dispositivo. Esto es posible gracias a una antena conexionada al módulo mediante un conector U.FL.

2.2.1. Uso de CloudMQTT

Una vez obtenidos los datos de ambos sensores, el Arduino, anteriormente conectado a internet gracias a su puerto ethernet, comenzará una conexión con el servidor MQTT. Este elemento, llamado CloudMQTT, será el encargado de recibir la lectura de los sensores. Tras ello, enviará la información recibida a aquellos clientes del servidor que estén subscritos a ella. En este caso, el cliente que reclama información es la aplicación IoT MQTT Panel.

2.2.2. Disposición de los datos en *dashboard*

Los datos leídos a través del Arduino se representarán en un panel de control para una mejor gestión de los mismos.

Para la visualización de la temperatura obtenida con el MPU-6050, se utilizará un calibrador. De este modo, cuando la temperatura sobrepase los 33 grados Celsius, su color pasará a ser rojo, mientras que si la temperatura es aceptable estará en verde y finalmente en azul cuando baje de los 10 grados.

Para los ejes x, y, z del acelerómetro y del giroscopio, se utilizarán dos gráficos de puntos, cuyas medidas aparecerán en el eje y de la gráfica, y al eje x le corresponderá la hora de llegada de cada dato.

Finalmente, la ubicación ofrecida por el módulo NEO-6M vendrá dada a modo de dirección URI. Una vez se lance esta dirección, la aplicación móvil redirigirá al usuario a la aplicación de mapas instalada en el dispositivo, la cual mostrará la localización del módulo mediante una chincheta.

El siguiente capítulo muestra una lista de los recursos utilizados para llevar a cabo el proyecto. De este modo, se presentan los recursos hardware y software envueltos en el desarrollo de la aplicación, aunque sus funcionalidades completas se explicarán en más detalle a medida que se vaya a explicar el comportamiento de la misma y cómo se ha llevado a cabo.

3 TECNOLOGÍA UTILIZADA

Todo nuestro conocimiento comienza merced a los sentidos

Immanuel Kant

En este capítulo se reúnen los elementos utilizados para la realización del proyecto. Tanto los componentes hardware como software son dispuestos para una mejor comprensión del trabajo realizado. Estos elementos han sido seleccionados para el correcto funcionamiento e implementación del mismo, así como para la obtención de los resultados adecuados.

3.1 Recursos hardware

En este apartado se engloban los componentes hardware relevantes en este proyecto.

3.1.1 Ordenador Portátil ASUS K55A

- Procesador Intel® Core™ i7-3630QM CPU @ 2.4GHz, 2401MHz, 4 Cores, 8 Procesadores Lógicos
- Memoria RAM 8 GB
- Sistema Operativo basado en 64 bits
- Disco Duro 443 GB
- Tarjeta gráfica integrada Intel® HD Graphics 4000

3.1.2 Arduino UNO

El Arduino Uno es una placa que incluye un microcontrolador de código abierto y presenta las siguientes características:

- Microcontrolador: ATmega328
- Voltaje Operativo: 5v
- Voltaje de Entrada (Recomendado): 7 – 12 v
- Pines de Entradas/Salidas Digital: 14 (De las cuales 6 son salidas PWM)
- Pines de Entradas Análogas: 6
- Memoria Flash: 32 KB (ATmega328) de los cuales 0,5 KB es usado por Bootloader.
- SRAM: 2 KB (ATmega328)

- EEPROM: 1 KB (ATmega328
- Velocidad del Reloj: 16 MHz

3.1.2.1 Comunicación

El Arduino presenta varios tipos de interfaces de comunicación con los periféricos: UART, I^2C y SPI [26].

La comunicación UART (Transmisor-Receptor Asíncrono Universal) es una de las comunicaciones más utilizadas ya que la mayoría de los microcontroladores presentan el hardware necesario para su implementación. Usa una línea de datos simple para transmitir y otra para recibir de forma asíncrona. Al realizar la transmisión, se envía un bit de inicio a nivel bajo, 8 bits de datos y un bit de parada a nivel alto.

I^2C es un bus de comunicación síncrona que usa simplemente dos líneas, una para el reloj (SCL) y otra para los datos (SDA), sin importar quién sea el emisor de los mismos, aunque la comunicación siempre es iniciada por el maestro. De esta forma, es el maestro quien comienza la señal de reloj y empieza la transmisión mediante direccionamiento.

SPI (Serial Peripheral Interface) es un bus de comunicaciones basado en un mínimo de tres líneas, dependiendo del número de esclavos conectados al maestro. Estas líneas de comunicación son SCK para el reloj, iniciado siempre por el maestro, MOSI (Master Output Slave Input) para los datos que salen del maestro hacia el esclavo y MISO (Master Input Slave Output) para el caso contrario. Si existen más de un esclavo para la comunicación SPI, es necesario utilizar la línea SS (Slave Select) encargada de seleccionar el esclavo correspondiente una vez se desee enviar información.

UART, al contrario que SPI e I^2C es asíncrono por lo que no está sincronizado con la señal de reloj y su velocidad está limitada a 2Mbps.

En este proyecto, se van a utilizar estos tres tipos de interfaces para la comunicación del Arduino con los módulos. La comunicación con el *Ethernet Shield* se realiza a través de SPI, con el módulo MPU-6050 a través de I^2C y con el módulo NEO-6M por UART.

Como se puede apreciar en la ilustración 2, se indican los puertos dedicados para cada conexión. Los puertos 10, 11, 12 y 13 están reservados para la comunicación SPI. A4, A5, 16 y 17 para I^2C y el resto de los pines, si no se trata de VCC o GND, pueden ser configurados para comunicación UART.

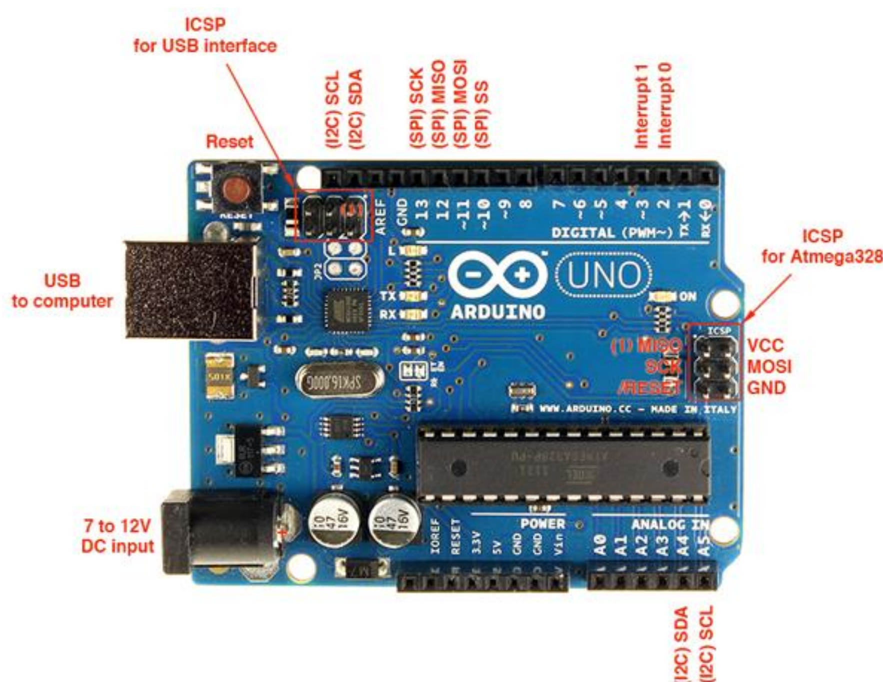

Ilustración 2: Arduino UNO

3.1.3 Arduino Ethernet Shield

Este dispositivo se conecta como un suplemento de Arduino UNO, añadiéndole así la capacidad de conexión a internet vía Ethernet y la posibilidad de manejo de una tarjeta de memoria SD, permitiendo su lectura y escritura. Para su comunicación con la placa principal a través de SPI (Interfaz Periférica en Serie), es necesario que los pines digitales del Arduino 10, 11, 12 y 13 no estén ocupados por otros dispositivos [10, 11].

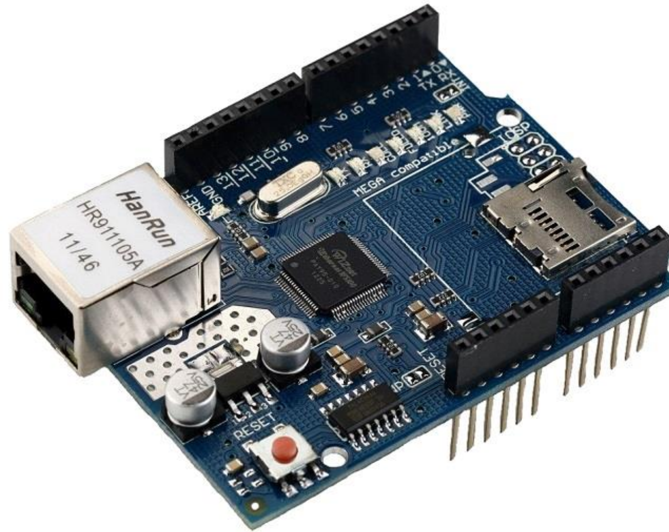

Ilustración 3: Ethernet Shield

3.1.4 Sensor MPU-6050

El sensor MPU-6050 permite combinar la aceleración y el movimiento de rotación más la información de rumbo en un único flujo de datos para la aplicación. El MPU-6050 es un dispositivo de seguimiento de movimiento de 6 ejes que combina un giroscopio de 3 ejes, un acelerómetro de 3 ejes y un Procesador Digital de Movimiento (DMP). Gracias a ello se obtienen las medidas de los 3 ejes cartesianos, tanto para la aceleración como para el movimiento de rotación. Este dispositivo permite también obtener la temperatura en el instante de lectura.

La comunicación maestro/esclavo se realiza a partir de la interfaz I^2C. Para ello, el sensor debe conectarse a los pines analógicos A4 (SDA) y A5 (SCL) del Arduino.

Ilustración 4: Módulo MPU-6050

3.1.5 Sensor NEO-6M

Gracias a la antena GPS presente en este sensor, se puede obtener la latitud y longitud para así, posteriormente, hallar la ubicación del dispositivo.

La comunicación de este periférico con el Arduino se realiza mediante UART, siendo simplemente necesario para la obtención de datos la configuración de dos pines cualesquiera, como puerto de recepción y puerto de transmisión.

Ilustración 5: Módulo GPS NEO-6M

3.2 Recursos Software

A continuación, se comentan los recursos software envueltos en este proyecto.

3.2.1 MQTT Cloud

MQTT Cloud es un *broker* en la nube encargado de manejar las peticiones de clientes MQTT y publicar datos a través de servidores basados en Mosquitto. Los clientes se subscriben a un tema, o publican en el mismo, a través del *broker* y este les hace llegar los mensajes de los temas a los que están subscritos, una vez ha llegado nueva información.

3.2.2 IoT MQTT Panel

IoT MQTT Panel es una aplicación móvil que permite observar los datos leídos de los sensores gráficamente. Mediante la creación de paneles podemos tanto enviar datos como recibirlos fácilmente, a través del servidor en la nube conectado a nuestros dispositivos a través de internet.

3.2.3 Arduino IDE

El Entorno de Desarrollo Integrado (IDE) de Arduino es una aplicación para ordenador escrita en Java. Este software es usado para el desarrollo de código y su posterior carga como programa a los dispositivos Arduino compatibles [15].

Una vez listados los elementos presentes en este proyecto, pasaremos al siguiente capítulo. El cuarto capítulo tiene como objetivo profundizar en la arquitectura del sistema, así como en la relación interna entre los distintos elementos. Se dispondrán las características de los sensores y su uso en este proyecto, así como la presentación de la plataforma utilizada y su protocolo de comunicación. De esta manera, se pretende dar a conocer de una forma completa los elementos que establecen el sistema para que, una vez llegados al desarrollo de la aplicación, tener un conocimiento íntegro de sus componentes.

4 APLICACIÓN DESARROLLADA

Con la tecnología a nuestra disposición, las posibilidades son infinitas.

Stephen Hawking

La arquitectura y análisis de un sistema es crucial para su correcto entendimiento. En este capítulo se presenta la descripción de los programas realizados y su arquitectura, así como la disposición de los datos en el panel de control. Como precedente, se desarrollan los elementos del sistema para una mayor profundización y entendimiento del proyecto.

En la ilustración 6 podemos observar los elementos que componen nuestro sistema.

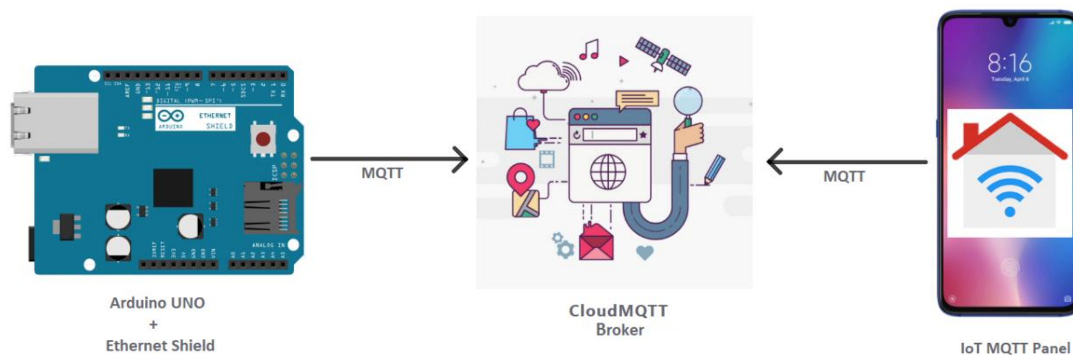

Ilustración 6: Esquema de comunicación de los elementos del sistema

- El primer elemento, al que van conectados los sensores MPU-6050 y NEO-6M, es una placa Arduino Uno, a la que se encuentra acoplada su componente *Ethernet Shield*. Este componente, dota al Arduino de la capacidad de conectarse a internet vía cable, una vez realizada su correcta configuración.
- El segundo elemento es la plataforma CloudMQTT con servidores Mosquitto en la nube. Esta plataforma es el elemento principal de nuestro sistema ya que es el encargado de recibir las peticiones de los clientes, enviando o recibiendo información de los mismos.
- El tercer elemento es una aplicación móvil gracias a la cual se dispone la información leída por el Arduino y recibida en la aplicación gracias al protocolo MQTT.

Antes de comenzar con el desarrollo de la aplicación, es necesario tener los conocimientos previos para el correcto entendimiento del sistema. Es por esto, por lo que comenzaremos explicando las distintas partes que lo componen en profundidad.

4.1 Sensores

La mayor parte de aplicaciones creadas para IoT se encargan de controlar sensores y recibir información crucial de los mismos. En este apartado se presentan los sensores usados en nuestro proyecto y se indaga en sus características más importantes.

4.1.1 MPU-6050

El MPU-6050 es un módulo dirigido a dispositivos con procesamiento de movimiento. Este dispositivo incluye una unidad de medición inercial o IMU (Inertial Measurment Units) de 6 grados de libertad (DoF) debido a la combinación de un acelerómetro y giroscopio de 3 ejes cada uno. Es capaz de procesar algoritmos de 9 ejes, y capturar movimiento en los ejes X, Y y Z a la misma vez.

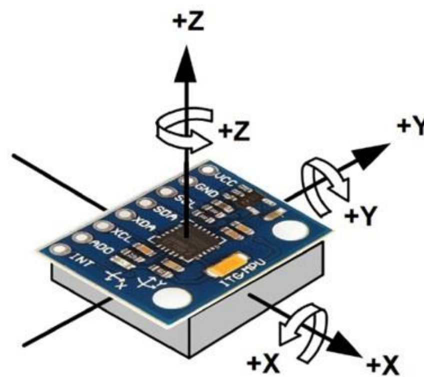

Ilustración 7: Ejes en el MPU-6050

Para comprender mejor el comportamiento de este módulo, es necesaria la correcta compresión del término aceleración, así como de la velocidad angular [13].

4.1.1.1 Aceleración

La aceleración es la variación de la velocidad por unidad de tiempo, definida matemáticamente como la derivada de la velocidad con respecto al tiempo:

$$a=dV/dt$$

A su vez, es también la segunda ley de Newton quien indica que, en un cuerpo con masa constante, la aceleración del cuerpo es proporcional a la fuerza que actúa sobre él mismo, de forma que:

$$a=F/m$$

Este segundo concepto es utilizado por los acelerómetros para medir la aceleración. Los acelerómetros internamente presentan los llamados MEMS (Micro Electro Mechanical Systems), gracias a los cuales es posible medir la aceleración. A pesar de que no exista movimiento, el acelerómetro siempre estará sintiendo la aceleración de la gravedad [18].

4.1.1.2 Velocidad Angular

La velocidad angular es la tasa de cambio del desplazamiento angular por unidad de tiempo, es decir, cómo de rápido gira un cuerpo alrededor de su eje:

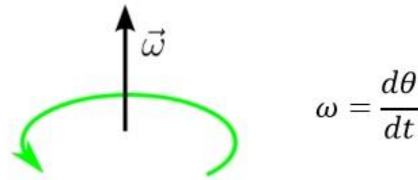

Ilustración 8: Movimiento angular

Los giroscopios, al igual que los acelerómetros, también utilizan MEMS, aunque en este caso para medir la velocidad angular usando el efecto Coriolis, el cual hace que un objeto que se mueve sobre el radio de un disco en rotación tienda a acelerarse con respecto a ese disco según si el movimiento es hacia el eje de giro o alejándose de éste [12].

Las salidas del giroscopio están en grados por segundo, por lo tanto, para poder obtener la posición angular, es necesario integrar la velocidad angular [18].

4.1.1.3 Características del MPU-6050

MPU-6050 consiste en un Procesador Digital de Movimiento (DMP), el cual tiene la propiedad de resolver cálculos complejos. Este dispositivo también cuenta con un convertidor analógico-digital de 16 bits, gracias al cual es posible capturar movimiento de tres dimensiones al mismo tiempo.

A parte de tener integrado un acelerómetro y un giroscopio, el MPU-6050 también presenta un sensor de temperatura como característica distintiva.

La comunicación del módulo se realiza a través de I^2C, esto le permite trabajar con la mayoría de los microcontroladores. I^2C es un bus direccional de dos cables usado para enviar datos entre circuitos integrados. Los pines SCL y SDA tienen una resistencia pull-up en la placa para obtener una conexión directa al microcontrolador o Arduino. El pin ADDR presenta internamente una resistencia a tierra por lo que, si no se conecta, la dirección por defecto será 0x68 (si se conectase, la dirección correspondiente al módulo sería 0x69) [19].

4.1.2 Módulo GPS NEO-6M

El módulo GPS permite obtener la localización del dispositivo mediante dos valores, la latitud y la longitud. A parte de estos dos parámetros, se pueden conocer el número de satélites con los que tiene visión, así como la hora a la que recibió la información, entre otros.

Para entender cuál es el funcionamiento de este dispositivo, es necesario conocer cómo funciona un receptor GPS, ya que este módulo funciona principalmente como uno, gracias a su antena extraíble.

4.1.2.1 Receptores GPS

El funcionamiento de los receptores GPS se basa en averiguar cómo de lejos se encuentran de un número de satélites, ya que están preprogramados para saber dónde se encuentran los satélites de GPS en cualquier momento.

Los satélites transmiten información sobre su posición en el momento dado a través de radio señales hacia la Tierra. Esas señales identifican a los satélites y le dicen al receptor dónde se encuentran situados.

El receptor entonces calcula cómo de lejos se encuentra cada satélite dependiendo de cuánto tardaron en llegar las señales. Una vez que obtiene información de al menos tres satélites, puede encontrar la ubicación en la Tierra. Este proceso se conoce como Trilateración Satelital.

4.1.2.2 Características NEO-6M

El módulo NEO-6M GPS puede seguir hasta 22 satélites en 50 canales y conseguir el mayor nivel de sensibilidad del mercado. Es capaz de identificar ubicaciones en cualquier parte del mundo a la vez que presenta un bajo consumo de potencia.

A diferencia de otros módulos GPS, este es capaz de obtener 5 actualizaciones de ubicación por segundo, con una precisión de 2.5 metros con respecto a la posición horizontal. Además, sus 6 funciones de posicionamiento cuentan con un TTFF definido como el tiempo requerido para que un dispositivo de navegación GPS adquiera señales de satélite y datos de navegación, y calcule una solución de posición, de menos de 1 segundo.

Una de las mejores características que el chip posee es el modo de ahorro de energía (PSM). Este modo permite una reducción en el consumo de energía del sistema al encender y apagar selectivamente partes del receptor.

El módulo se comunica con el microcontrolador a través de la UART, admitiendo una velocidad de transmisión de 4800bps a 230400bps, con una velocidad de transmisión predeterminada de 9600.

Hay un LED presente en el módulo que se encarga de indicar si se ha obtenido comunicación con algún satélite. Parpadeará a diferente velocidad dependiendo del estado en el que se encuentre:

- Si el LED no parpadea entonces indica que está buscando satélites.
- Al parpadear cada segundo nos indica que el módulo puede ver suficientes satélites.

A pesar de todas sus funcionalidades integradas, NEO-6M necesita una antena para realizar cualquier tipo de comunicación, por lo que presenta un conector U.FL hembra al que conectarla.

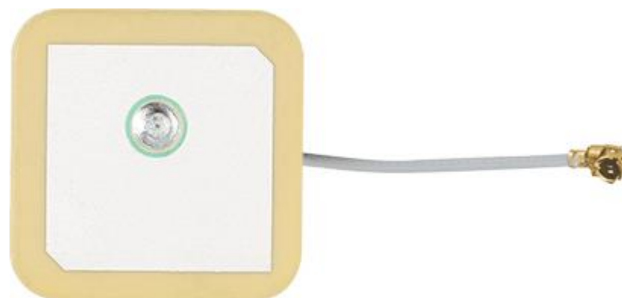

Ilustración 9: Antena con conector U.FL para comunicación del módulo NEO-6M

Una vez conexionada la antena al módulo, será necesario que la antena tenga visión directa con los satélites, ya que no es capaz de obtener información de los satélites necesarios en espacios cerrados [17].

4.2 CloudMQTT

En este apartado se explica el elemento principal del proyecto. Para poder entender el comportamiento de esta aplicación, es necesario conocer cómo funciona internamente. Para ello comenzaremos hablando del protocolo MQTT.

4.2.1 Protocolo MQTT

La documentación oficial de este protocolo lo define como:

> MQTT es un protocolo de transporte de mensajes *publish/subscribe* entre cliente y servidor. Es ligero, abierto, simple y está diseñado para ser fácil de implementar. Estas características lo convierten en una opción ideal en muchas situaciones, incluyendo entornos restringidos como la comunicación en contextos Máquina a Máquina (M2M) e Internet de las cosas (IoT), donde se requiere una pequeña huella de código y/o el ancho de banda de la red es muy importante [1].

Como hemos podido saber gracias a la definición anterior, MQTT es un protocolo estándar de mensajería e intercambio de datos para Internet de las cosas. Es un protocolo abierto, estandarizado por OASIS e ISO, el cual provee una forma escalable y rentable de conectar los dispositivos a través de internet. Este protocolo, binario y muy ligero gracias a su mínima sobrecarga de paquetes, sobresale frente a otros protocolos como HTTP cuando se trata del envío de datos a través de cables [2, 3].

Un cliente MQTT es cualquier dispositivo que ejecute una librería MQTT y sea capaz de conectarse a un *broker* MQTT a través de internet. Por lo tanto, cualquier dispositivo que se comunique por MQTT a través de la pila de protocolos TCP/IP, puede ser llamado cliente MQTT. Tanto los editores como los suscriptores son clientes MQTT. A su vez, un mismo cliente es capaz de publicar y recibir mensajes.

El anterior llamado *broker*, es el elemento complementario al cliente MQTT, constatado como el elemento principal de cualquier protocolo de publicación/subscripción. Es el responsable de recibir todos los mensajes, filtrarlos, determinar quién está subscrito a cada mensaje y enviar los mensajes a aquellos clientes subscritos [6].

Para poder manejar los mensajes correctamente, el *broker* dispone de varias opciones de filtrado: basado en el tema, basado en el contenido o basado en el tipo. El filtro de mensajes usado por MQTT es el basado en un tema. Este tipo de filtrado se basa en un *topic* (tema) que es parte de cada mensaje. El cliente se subscribe al *broker* para estar al tanto de los *topics* de interés. A partir de ese momento, el *broker* se asegura de que el cliente obtiene todos los mensajes que han sido publicados en esos temas [4].

4.2.2 Conexión

El protocolo MQTT está basado en TCP/IP por lo que, tanto el cliente como el *broker* necesitan tener una pila con estas características.

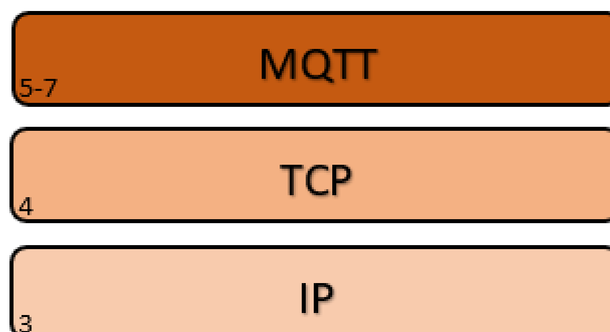

Ilustración 10: Pila de protocolos de los dispositivos MQTT

Las conexiones de los clientes siempre son manejadas por el *broker*. Para iniciar una conexión, el cliente le envía un mensaje CONNECT, el cual responde con CONNACK y un código de estado, devolviendo 0 en caso de que la conexión haya sido aceptada.

CONTENIDO	**CAMPO OPCIONAL**	**DESCRIPCIÓN**
`clientId`	Obligatorio	Identificador para cada cliente MQTT que se conecta a un *broker*, el cual lo utiliza para identificar al cliente y su estado actual. Debe ser único por cliente y *broker*.
`cleanSession`	Obligatorio	Bandera para indicar al *broker* si el cliente quiere establecer o no una sesión persistente. Si la sesión es persistente, el *broker* almacenar todas las subscripciones del cliente y los mensajes perdidos para los que está subscrito con un QoS de nivel 1 o 2.
`username`	Sí	Para la autenticación del cliente se puede enviar una combinación de usuario y contraseña. Si esta información no se encuentra encriptada, la contraseña será enviada como texto plano.
`username`	Sí	
`lastWillTopic`	Sí	Este mensaje notifica a otros clientes cuando un cliente se desconecta sin delicadeza (con un mensaje DISCONNECT incorrecto). En él se indica un *topic*, una bandera de mensaje retenido, QoS y un *payload*. El *broker* almacena este mensaje hasta que detecta que el cliente se ha desconectado. Cuando esto ocurre, el *broker* envía el mensaje indicado a los clientes subscritos al lastWillTopic.
`lastWillQoS`	Sí	
`lastWillMessage`	Sí	
`lastWillRetain`	Sí	
`keepAlive`	Obligatorio	Intervalo en segundos indicado al *broker* para definir el máximo periodo de tiempo que el *broker* y el cliente puede mantener una conexión sin enviar un mensaje.

Tabla 2: Contenido mensaje CONNECT

CONTENIDO	CAMPO OPCIONAL	DESCRIPCIÓN
`sessionPresent`	Obligatorio	Bandera que indica al cliente si el *broker* ya tiene una sesión persistente disponible de anteriores interacciones entre ambos.
`returnCode`	Obligatorio	Código de retorno que indica si la sesión ha sido establecida o, en caso contrario, por qué no se ha podido establecer.

Tabla 3: Contenido mensaje CONNACK

CÓDIGO	SIGNIFICADO
0	Conexión aceptada.
1	Conexión rechazada, la versión de protocolo no es aceptable.
2	Conexión rechazada, identificador rechazado.
3	Conexión rechazada, el servidor no se encuentra disponible.
4	Conexión rechazada, nombre de usuario o contraseña incorrectos.
5	Conexión rechazada, sin autorización.

Tabla 4: Códigos de retorno del mensaje CONNACK

Una vez que la conexión ha sido establecida, el *broker* la mantiene abierta hasta que el cliente se desconecta o la conexión se pierde.

Si el mensaje CONNECT está mal formado o ha pasado demasiado tiempo entre la apertura del socket y el envío del mensaje, el *broker* cierra la conexión para evitar que haya clientes que ralenticen al servidor [6].

4.2.3 Modelo Publish/Subscribe

El modelo publicación/subscripción proporciona una alternativa a la arquitectura tradicional de cliente/servidor. En el modelo cliente/servidor, un cliente se comunica directamente con el extremo. El modelo pub/sub separa al cliente que envía el mensaje (publicador) del cliente o clientes que reciben el

mensaje (subscriptores). Los publicadores y los subscriptores nunca se comunican entre ellos directamente. De hecho, puede que no estén al tanto de la existencia de los otros. La conexión entre ellos es manejada por un tercer componente, el *broker*. El trabajo del *broker* es filtrar todos los mensajes recibidos y distribuirlos correctamente a los subscriptores.

El aspecto más importante de esta arquitectura es la capacidad de dejar al margen al publicador en un mensaje enviado por el subscriptor.

En resumen, el modelo pub/sub elimina la comunicación directa entre el publicador del mensaje y el receptor. La capacidad de filtrado del *broker* hace posible controlar qué cliente recibirá qué mensaje. Para publicar o recibir mensajes, los publicadores y subscriptores solo necesitan saber la dirección IP, o el nombre del servidor, y el puerto del *broker*.

Pub/Sub escala mejor que el enfoque tradicional de cliente/servidor. Esto se debe a que las operaciones en el *broker* pueden ser paralelizadas y los mensajes pueden procesarse de una manera controlada por eventos. El almacenamiento en caché de mensajes y el enrutamiento inteligente de mensajes son, a menudo, factores decisivos para mejorar la escalabilidad [4].

Existen cinco tipos de mensajes para la comunicación entre cliente y servidor: publish, subscribe, suback, unsubscribe y unsuback [5].

4.2.3.1 PUBLISH

Cada mensaje presenta un *payload* (relleno) que contiene los datos a transmitir en el formato que el cliente indique. El contenido de este mensaje se puede observar en la tabla 5.

CONTENIDO	**DESCRIPCIÓN**
`packetId`	Identifica de forma exclusiva un mensaje a medida que fluye entre el cliente y el *broker*.
`topicName`	Cadena simple que está estructurada jerárquicamente con barras diagonales como delimitadores.
`qos`	Indica el nivel de la calidad de servicio (QoS) del mensaje. Hay tres niveles: 0, 1 y 2. El nivel de servicio determina qué tipo de garantía tiene un mensaje de alcanzar el destino previsto.
`retainFlag`	Esta bandera define si el mensaje ha sido guardado por el *broker* como el último valor conocido para un *topic* especifico. Cuando un cliente nuevo se subscribe a un *topic*, recibe el último mensaje almacenado para ese tema.
`payload`	Este campo es el contenido real del mensaje. Es posible enviar imágenes, texto en cualquier codificación, datos encriptados y cualquier dato binario.

`dupFlag`	Esta bandera indica que el mensaje es un duplicado y fue reenviado porque el destino previsto no asintió el mensaje original.

Tabla 5: Contenido mensaje PUBLISH

Cuando un cliente envía un mensaje al *broker* para ser publicado, este lee el mensaje, lo asiente, y lo procesa, de manera que determina qué clientes están subscritos al *topic* y les envía el mensaje.

4.2.3.2 SUBSCRIBE

Para recibir mensajes de un tema, el cliente envía un mensaje de subscripción al *broker*. Este mensaje es muy simple, conteniendo un identificador de paquete y una lista de subscripciones.

CONTENIDO	**DESCRIPCIÓN**
`packetId`	Identifica de forma exclusiva un mensaje a medida que fluye entre el cliente y el *broker*. Este identificador es fijado por la librería del cliente o por el *broker*.

`qos1`
`topic1`
`qos2`
`topic2`
`…`
	Un mensaje SUBSCRIBE puede contener múltiples subscripciones para un cliente. Cada subscripción se realiza a partir de un *topic* y un nivel de QoS.

Tabla 6: Contenido mensaje SUBSCRIBE

4.2.3.3 SUBACK

Para confirmar cada subscripción, el *broker* envía un mensaje de asentimiento al cliente. Este mensaje contiene el identificador de paquete del mensaje original de subscripción y una lista de códigos de retorno, devolviendo 128 en caso de fallo.

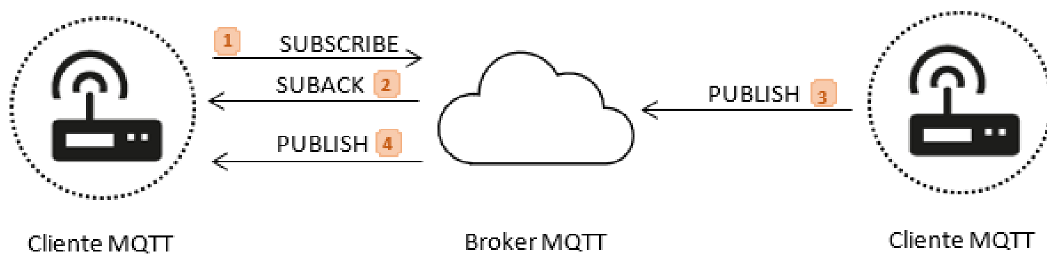

Ilustración 11: Paso de mensajes SUBSCRIBE/PUBLISH

Después de que un cliente envíe correctamente el mensaje de suscripción y reciba el asentimiento, obtendrá todos los mensajes publicados que coinciden con un tema en las suscripciones que contenía el mensaje SUBSCRIBE.

CONTENIDO	DESCRIPCIÓN
`packetId`	Identificador único usado para identificar a un paquete. Es el mismo que en el mensaje SUBSCRIBE.

`returnCode1`
`returnCode2`
...
	El *broker* envía un código de retorno por cada par de *topic*/QoS que recibe en el mensaje SUBSCRIBE.

Tabla 7: Contenido mensaje SUBACK

CÓDIGO	SIGNIFICADO
0	Éxito, con un máximo QoS de 0.
1	Éxito, con un máximo QoS de 1.
2	Éxito, con un máximo QoS de 2.
128	Ha habido fallo para un *topic* específico.

Tabla 8: Códigos de retorno del mensaje SUBACK

4.2.3.4 UNSUBSCRIBE

El mensaje complementario a SUBSCRIBE es el mensaje de UNSUBSCRIBE. Este mensaje elimina las subscripciones de un cliente existentes en el *broker*. Al igual que el mensaje de subscripción, este mensaje contiene un identificador de paquete y una lista de *topics*.

CONTENIDO	DESCRIPCIÓN
`packetId`	Identifica de forma exclusiva un mensaje a medida que fluye entre el cliente y el *broker*. Este identificador es fijado por la librería del cliente o por el *broker*.

`topic1`
`topic2`
…
	La lista de *topics* puede contener múltiples temas de los cuales el cliente quiere cancelar la subscripción. No es necesario enviar el QoS como en el mensaje SUBSCRIBE. El *broker* cancela la suscripción sin importar el nivel de servicio al que se encontraba originalmente subscrito el cliente.

Tabla 9: Contenido mensaje UNSUBSCRIBE

4.2.3.5 UNSUBACK

Para confirmar la cancelación de la subscripción, el *broker* envía un mensaje de asentimiento al cliente. Este mensaje contiene el identificador original del mensaje UNSUBSCRIBE.

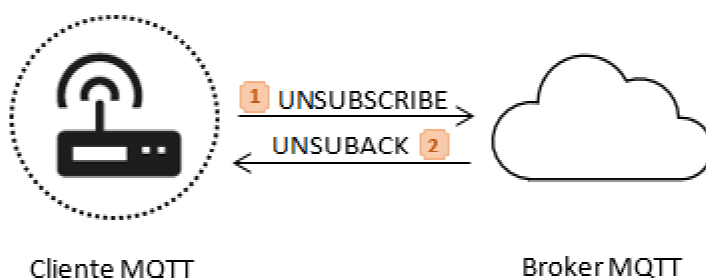

Ilustración 12: Paso de mensajes cancelación de subscripción

4.2.4 Plataforma CloudMQTT

CloudMQTT es un *broker* globalmente distribuido, constituido por servidores Mosquitto en la nube. Mosquitto implementa el protocolo MQTT que, como hemos visto anteriormente, provee métodos ligeros para el envío de mensajes usando el modelo de colas de publicación/subscripción.

Las colas de mensajes proveen un protocolo de comunicaciones asíncronas, gracias al cual el emisor y el receptor del mensaje no necesitan interactuar con la cola de mensajes al mismo tiempo. Los mensajes que son introducidos en la cola son guardados hasta que el receptor lo reclama o hasta que el mensaje expira [7].

El *broker* CloudMQTT usa números de puertos aleatorios que son generados por el *broker* en sí. Para conexiones sin seguridad, el número de puerto comienza por 1, para TLS/SSL en 2, y para soporte Websocket en 3 [8].

El protocolo MQTT posee un campo de usuario y contraseña en el mensaje CONNECT para la autenticación. De esta forma, poder obtener comunicación con el *broker*, simplemente es necesario el nombre del servidor o dirección IP, un usuario y su contraseña [16].

Para comenzar a usar CloudMQTT es necesario registrarse y, tras ello, crear una instancia. Esta aplicación ofrece un plan de instancias gratuito, llamado Cute Cat, que es el que usaremos para este proyecto. El proveedor de los servidores es Amazon Web Services, con región en Virginia del Norte.

Una vez creada la instancia, el *broker* nos proporcionará todos los datos necesarios para que podamos conectarnos a él.

4.3 IoT MQTT Panel

Para una mejor comprensión de los datos recogidos, existen aplicaciones en las que exponer las lecturas de forma gráfica. De este modo, este apartado se centra en la presentación de la aplicación elegida para mostrar nuestros datos.

4.3.1 *Dashboard*

Cuando realizamos la lectura de los sensores, los datos son recogidos en la API de CloudMQTT, gracias a su función de Websocket. Aun así, su visualización no es fácil ya que obtenemos una lista de datos constantemente actualizándose.

Los paneles de control o *dashboards* permiten monitorizar el nivel de los sensores en tiempo real, ofreciendo la administración rápida y fácil de los mismos.

En un panel de control se pueden añadir *widgets* para la gestión de los sensores, tales como gráficas o botones, y crear alarmas para ser notificados una vez ha habido un cambio en alguno de ellos [20].

4.3.2 Aplicación

IoT MQTT Panel es una aplicación móvil a modo de panel de control que permite gestionar y visualizar proyectos de IoT basados en el protocolo MQTT. Esta aplicación permite la conexión al servidor tanto a través del protocolo TCP como de Websocket, usado especialmente en redes restringidas por firewall. Además, permite una comunicación segura gracias al soporte de certificados SSL.

Proporciona soporte JSON para los mensajes de publicación y suscripción. Aun así, para mantener la aplicación simple, todos los *payloads* están implementados como cadenas.

Los paneles están configurados para publicar o recibir datos automáticamente, por lo tanto, son actualizados en tiempo real. IoT MQTT Panel está diseñada no solo para visualizar los estados de los distintos sensores, sino para organizar las conexiones, mensajes y dispositivos, entre otros. Esta aplicación puede gestionar múltiples conexiones, cada una conectada a un *broker* distinto. De este modo, los múltiples dispositivos de cada conexión son fácilmente separables, permitiendo una mejor gestión del sistema [21, 22].

Para comenzar la comunicación con el *broker*, es necesario introducir los datos de la instancia creada en CloudMQTT. Tras ello, podremos crear los paneles necesarios para nuestro proyecto.

4.4 Desarrollo

Una vez indagados todos los componentes del sistema, se va a llevar a cabo el desarrollo de la aplicación realizada. Primero comenzaremos por la lectura de los sensores, para su posterior envío de datos al servidor y, finalmente, la recepción de los datos en la aplicación móvil.

4.4.1 Lectura y envío de datos de los sensores

Para proceder a la lectura de los datos proporcionados por los sensores, se han realizado dos programas codificados en c, a través del software Arduino IDE y las librerías necesarias. Además de estos programas, se dispone de un código conjunto en el que mediante una única conexión al *broker*, ambos sensores publican sus datos.

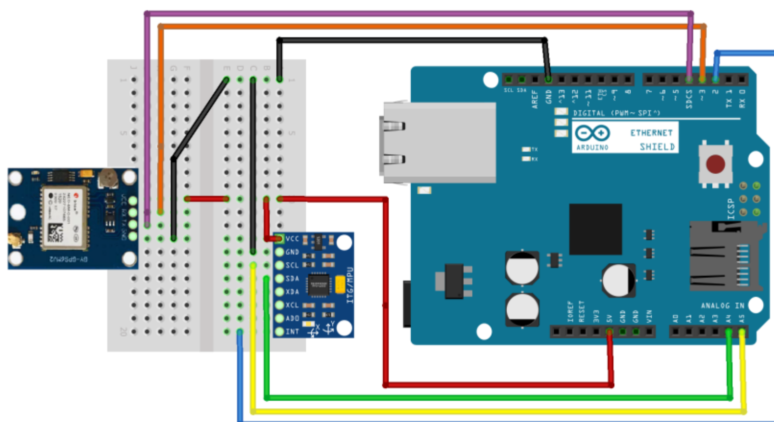

Ilustración 13: Esquema de conexión completo

El *Ethernet Shield* provee a Arduino la capacidad de internet, pero esto no es suficiente para que el dispositivo disponga de conexión a internet. Para ello, se necesitan las librerías 'SPI' y 'Ethernet'. La primera de ellas es necesaria debido a que el módulo Ethernet se comunica con el Arduino por medio del bus SPI. La MAC del dispositivo es establecida aleatoriamente. En este caso, como solo presentamos un dispositivo Arduino, no es posible que haya conflicto entre direcciones MAC iguales. Para configurar la conexión, es necesario añadir una dirección IP estática por si la capacidad DHCP del dispositivo fallara. Además, es necesario proporcionar la IP del servidor DNS, así como el *gateway* y la dirección de la subred.

```
// Internet Configuration
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 100);
IPAddress dnServer(192, 168, 1, 1);
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);
```

Ilustración 14: Configuración acceso a internet

Una vez definidos los parámetros necesarios, podemos comenzar la conexión a internet mediante la siguiente instrucción:

```
Ethernet.begin(mac, ip, dnServer, gateway, subnet);
```

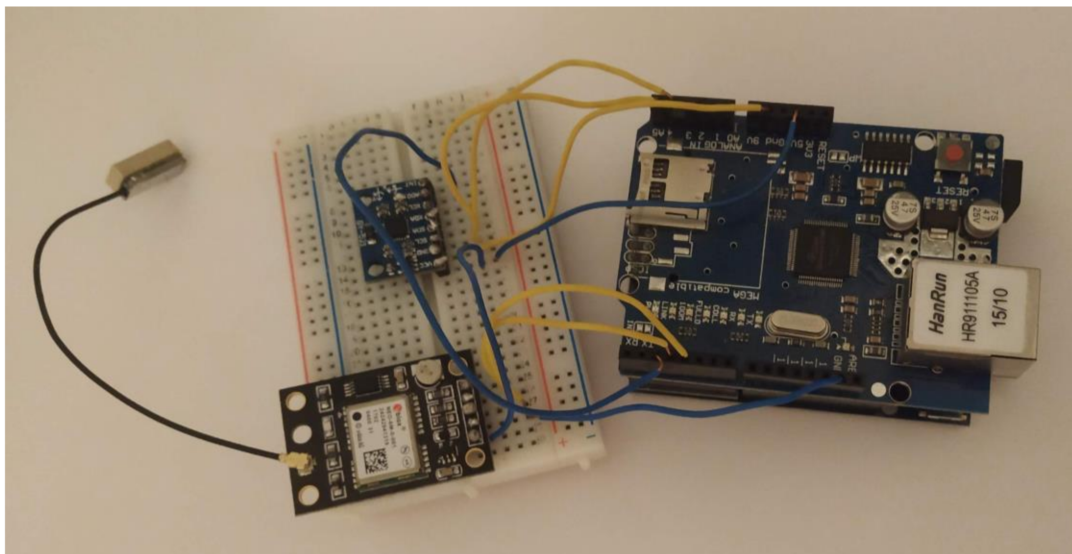

Ilustración 15: Conexión completa entre Arduino y sensores

En los siguientes subapartados se procede a la explicación de ambos programas, para una mayor profundización en cada sensor.

4.4.1.1 Sensor MPU-6050

El sensor MPU-6050 presenta funcionalidad de acelerómetro y giroscopio, a la vez que ofrece medidas de temperatura. Dispone de comunicación vía I^2C, por lo que sus pines SDA y SCL van conectados a los

puertos analógicos A4 y A5 respectivamente. Para que el Arduino realice la lectura de este sensor una vez haya habido cambios, se conectará el pin INT de interrupción al puerto 2 para que, de esta forma, se reduzca el consumo de potencia, evitando lecturas innecesarias del dispositivo [13].

Para que el Arduino pueda mantener comunicación por I²C, simplemente es necesario añadir la librería 'Wire'.

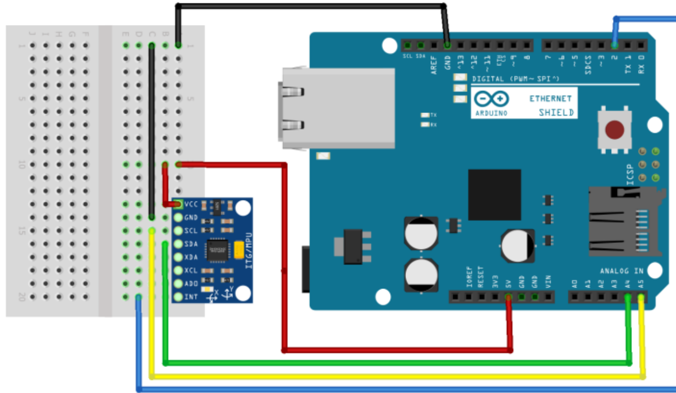

Ilustración 16: Esquema de conexión MP6050 con Arduino

La configuración del MPU se realiza comenzando la transmisión con la dirección del esclavo, 0x68. Tras esto, se envía un 0 al registro de gestión de la energía (0x6B) para despertar al dispositivo del modo *sleep*.

```
// MPU configuration
Wire.begin();
Wire.beginTransmission(MPU_ADDR); // Begins a transmission to the I2C slave (GY-521 board)
Wire.write(0x6B); // PWR_MGMT_1 register
Wire.write(0); // set to zero (wakes up the MPU-6050)
Wire.endTransmission(true);
```

Ilustración 17: Configuración MPU

Para empezar a leer los datos, se comienza con el registro 0x3B y a partir de este se obtienen 14 registros en total. A esta función se le denomina 'Burst Read'. De este modo, con una única petición obtenemos en orden la lectura de los tres ejes del acelerómetro, la temperatura, y los ejes del giroscopio.

```
Wire.beginTransmission(MPU_ADDR);
Wire.write(0x3B); // starting with register 0x3B (ACCEL_XOUT_H)
Wire.endTransmission(false); // the parameter indicates that the Arduino will send a restart.
Wire.requestFrom(MPU_ADDR, 7*2, true); // request a total of 7*2=14 registers

// "Wire.read()<<8 | Wire.read();" means two registers are read and stored in the same variable
accelerometer_x = Wire.read()<<8 | Wire.read(); // reading registers: 0x3B (ACCEL_XOUT_H) and 0x3C (ACCEL_XOUT_L)
accelerometer_y = Wire.read()<<8 | Wire.read(); // reading registers: 0x3D (ACCEL_YOUT_H) and 0x3E (ACCEL_YOUT_L)
accelerometer_z = Wire.read()<<8 | Wire.read(); // reading registers: 0x3F (ACCEL_ZOUT_H) and 0x40 (ACCEL_ZOUT_L)
temperature = Wire.read()<<8 | Wire.read(); // reading registers: 0x41 (TEMP_OUT_H) and 0x42 (TEMP_OUT_L)
gyro_x = Wire.read()<<8 | Wire.read(); // reading registers: 0x43 (GYRO_XOUT_H) and 0x44 (GYRO_XOUT_L)
gyro_y = Wire.read()<<8 | Wire.read(); // reading registers: 0x45 (GYRO_YOUT_H) and 0x46 (GYRO_YOUT_L)
gyro_z = Wire.read()<<8 | Wire.read(); // reading registers: 0x47 (GYRO_ZOUT_H) and 0x48 (GYRO_ZOUT_L)

temperature = temperature/340.00+36.53;
```

Ilustración 18: Lectura de datos MPU-6050

Los valores de los ejes de ambos instrumentos, así como de la temperatura, vienen dados por 16 bits en modo complemento a dos. Es por esto, por lo que se necesitan leer dos registros para obtener la lectura correcta de un dato. Estos registros son llamados alto o bajo, dependiendo de la posición final de sus bits [23].

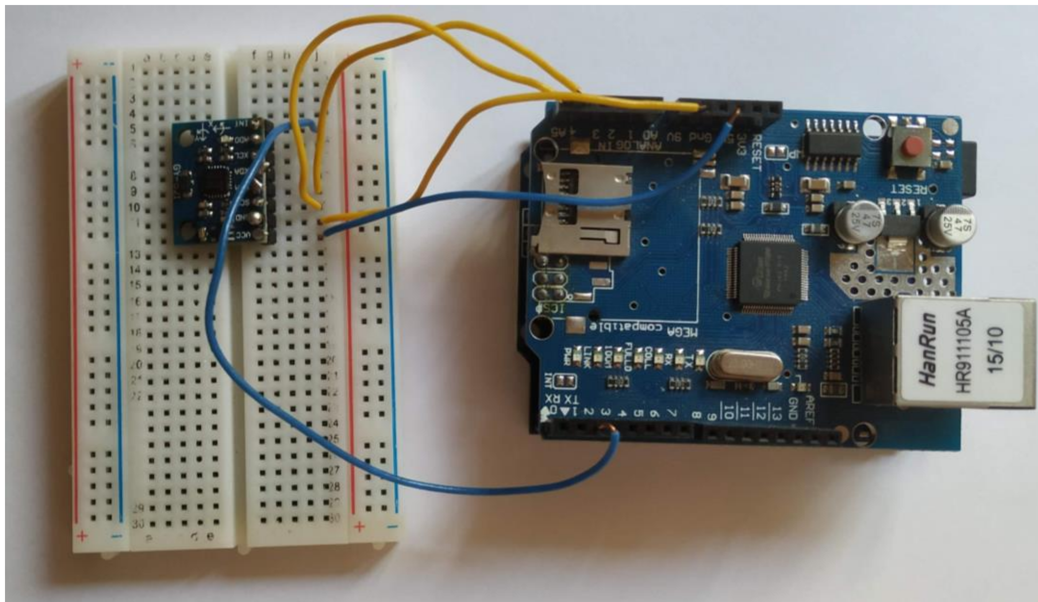

Ilustración 19: Conexión MPU con Arduino

4.4.1.2 Módulo NEO-6M

El sensor NEO-6M es un módulo GPS que ofrece una localización a modo de longitud y latitud. Estos datos los recibe gracias a una pequeña antena conectada al dispositivo a través de un diminuto conector coaxial. La comunicación del módulo con el Arduino la realiza a través de UART.

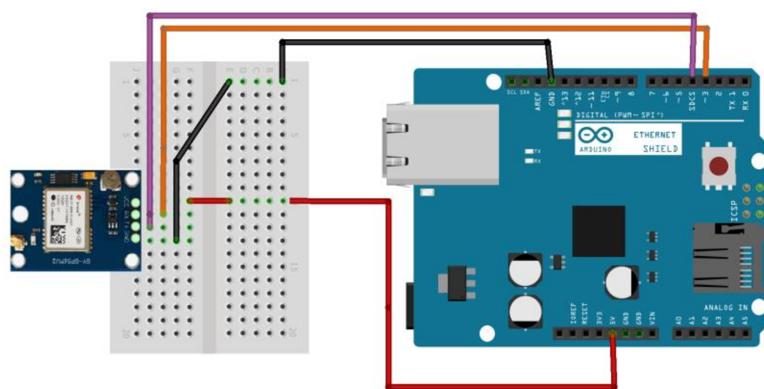

Ilustración 20: Esquema de conexión NEO-6M con Arduino

Los puertos 3 y 4 del Arduino son configurados como transmisor y receptor, en ese orden [14]. De este modo, para poder obtener comunicación con el esclavo es necesario incluir la librería 'SoftwareSerial' e iniciar los puertos mencionados anteriormente, añadiendo la velocidad de transmisión del GPS, 9600 baudios.

```
static const int RXPin = 4, TXPin = 3;
static const uint32_t GPSBaud = 9600;
SoftwareSerial ss(RXPin, TXPin);
```

Ilustración 21: Configuración NEO-6M

Una vez iniciados los puertos para la comunicación UART, comenzamos a recibir información a través del monitor serie presente en el IDE. La información enviada por el dispositivo sigue el protocolo NMEA (National Marine Electronics Asociation), de modo que los datos recibidos son sentencias estándares para la recepción de datos GPS.

```
while (ss.available() > 0)
  if (gps.encode(ss.read()))
    displayInfo();
```

Ilustración 22: Lectura de datos GPS

Estas sentencias tienen una longitud media de 70 caracteres, en la que la información se presenta separadas simplemente por comas. Debido a esto, obtener los datos se convierte en una tarea tediosa. Para ello, incluiremos la librería ‘TinyGPS++’, la cual se encarga de facilitarnos la substracción de los distintos campos a tener en cuenta.

```
LAT_deg = gps.location.rawLat().deg;
LAT_min = gps.location.rawLat().billionths;
LAT_coord = gps.location.rawLat().negative ? "-" : "+";

LONG_deg = gps.location.rawLng().deg;
LONG_min = gps.location.rawLng().billionths;
LONG_coord = gps.location.rawLng().negative ? "-" : "+";
```

Ilustración 23: Tratamiento de datos GPS para la localización

Los campos que hemos necesitado tanto para la latitud como para la longitud han sido los grados de ambos, con 6 decimales de precisión, y el signo de ellos, para poder hallar si se trata de norte o sur. Gracias a estos datos se consigue obtener la localización completa del dispositivo.

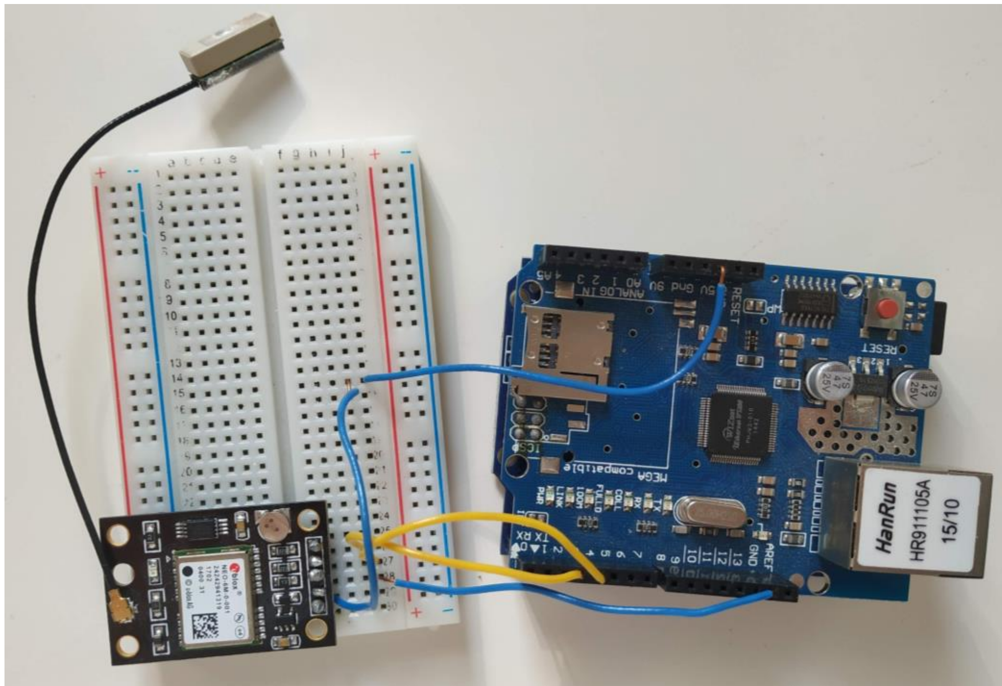

Ilustración 24: Conexión Arduino con GPS

4.4.2 Broker

Para que los clientes puedan comunicarse con el servidor, es necesaria la creación de una instancia, la cual ofrece el nombre del servidor, los puertos de conexión y distintos datos para la seguridad de la comunicación.

Para obtener una instancia del *broker* MQTT debemos acceder a la página oficial de CloudMQTT y registrarnos, si no lo hemos hecho anteriormente. También es posible acceder con una cuenta de GitHub o Google, por lo que en nuestro caso utilizaremos el acceso a través de la plataforma GitHub.

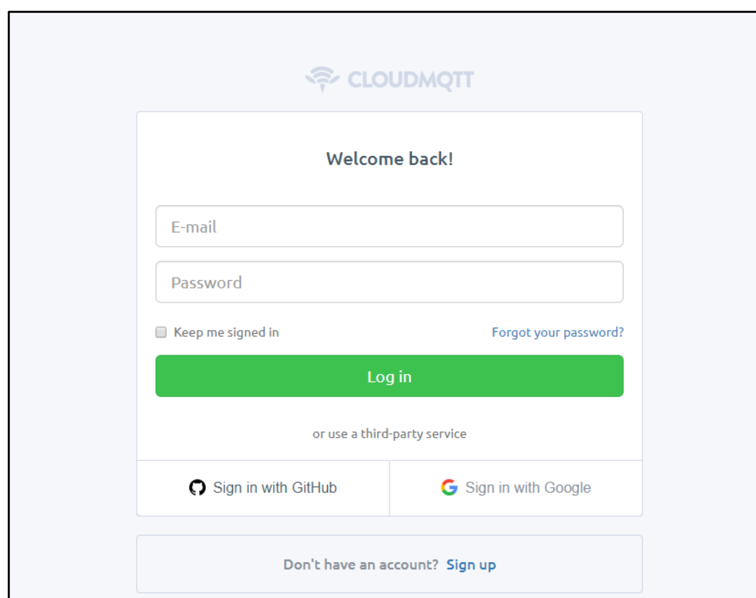

Ilustración 25: Página de acceso a CloudMQTT.

Tras acceder, nos encontraremos en la página principal de nuestra cuenta, en la que se muestra una lista de nuestras instancias creadas.

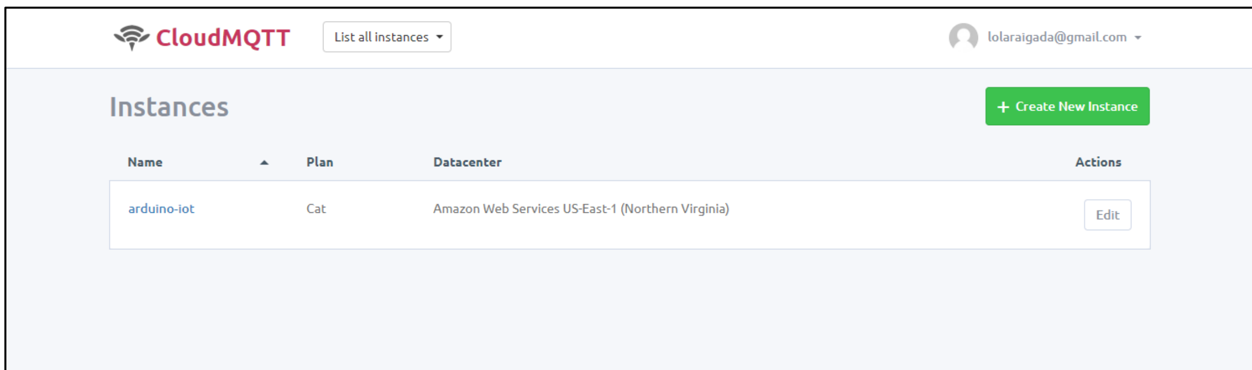

Ilustración 26: Página principal con listado de instancias creadas.

Para proceder a crear una nueva instancia, pulsaremos sobre el botón verde de la esquina superior derecha, que indica ‘Create New Instance’.

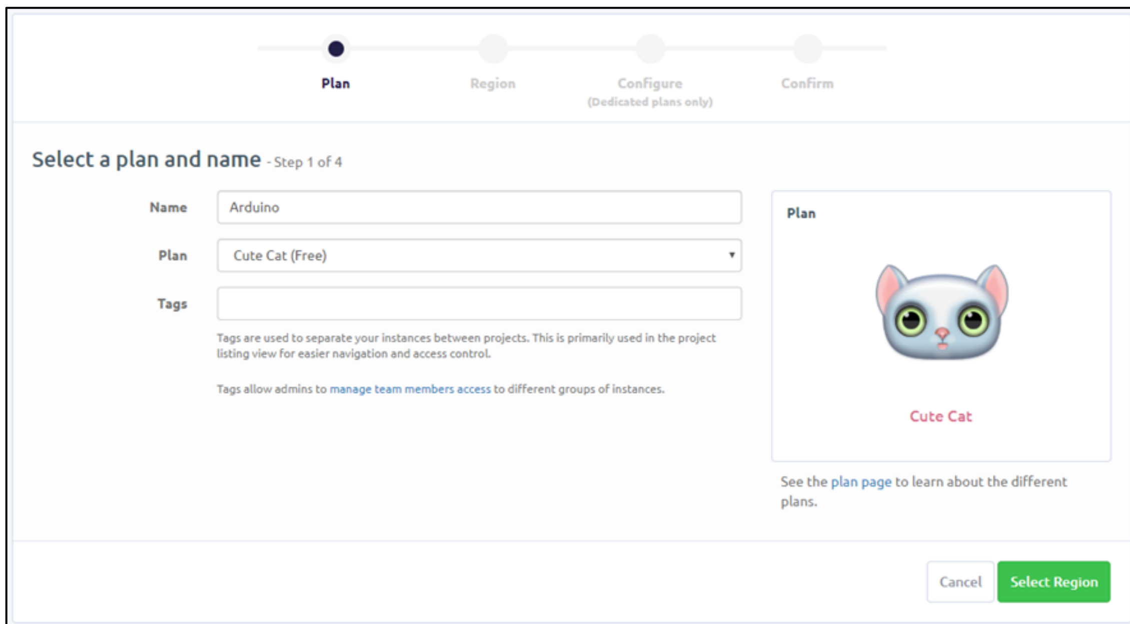

Ilustración 27: Paso 1 para crear nueva instancia

Esta plataforma dispone de distintos planes de subscripción dependiendo de las características del servidor que se vayan buscando. En nuestro caso, el plan Cute Cat es adecuado para nuestras necesidades ya que permite la conexión de hasta 5 clientes y es gratuito.

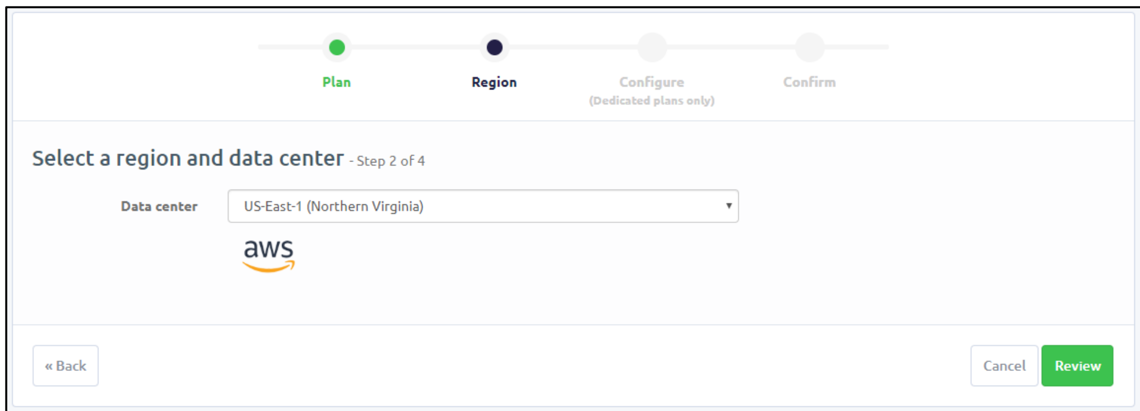

Ilustración 28: Selección de región para la instancia

Una vez seleccionada una región para los servidores de AWS, podremos confirmar la instancia y obtener toda la información de esta.

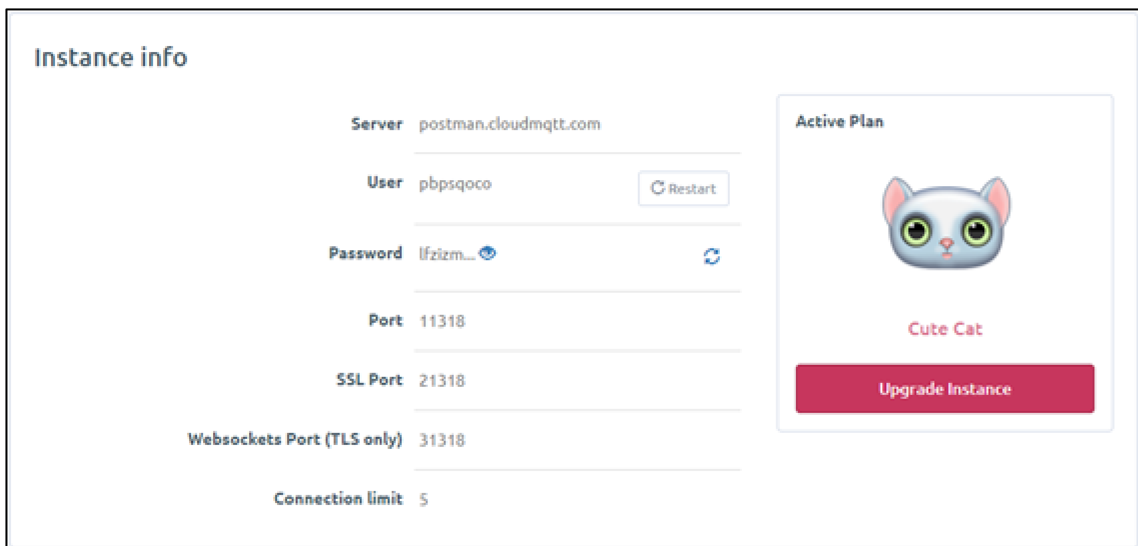

Ilustración 29: Información de la instancia creada en CloudMQTT

La comunicación de ambos sensores con el *broker* se lleva a cabo gracias a la librería 'PubSubClient'. Esta librería permite la conexión de un cliente MQTT con el servidor para el posterior envío/recepción de mensajes.

```
// Ethernet and MQTT related objects
EthernetClient ethClient;
PubSubClient mqttClient(ethClient);
```

Ilustración 30: Creación cliente MQTT

Para la configuración, se crea un cliente MQTT a partir de un cliente Ethernet. Con este cliente, se establece el servidor mediante el *hostname* y el puerto indicado en la instancia. Tras esto, ya es posible conectarse al *broker*.

```
const char* server = "postman.cloudmqtt.com";
mqttClient.setServer(server, 11318);

 // Attempt to connect to the server with the ID "dolrairom", and intance's user and password.
while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
  Serial.println("Attempting to connect");
  delay(1000);
}
Serial.println("Connection has been established, well done");
mqttClient.setCallback(subscribeReceive);
```

Ilustración 31: Conexión al *broker* MQTT desde el Arduino

La conexión al servidor se llevará a cabo mediante la autenticación usuario-contraseña indicados, de nuevo, en la instancia. El campo principal de la conexión es un identificador de cliente con el que se identifican los paquetes de publicación/subscripción. El resto de los campos obligatorios indicados en la tabla 2, se encuentran definidos en la función ‘connect’, tomando los valores por defecto de 15 segundos para ‘keepAlive’ y true para ‘cleanSession’.

Si la conexión ha sido exitosa, es necesario establecer un *callback* al que acudirá el programa tras recibir algún evento. En nuestro caso, esta función simplemente se encarga de imprimir al monitor.

Para proceder a la publicación de los datos, es necesario estar subscrito con anterioridad al tema al que se quiere enviar la información.

```
mqttClient.subscribe("map");
mqttClient.subscribe("temperature");
mqttClient.subscribe("acc_x");
mqttClient.subscribe("acc_y");
mqttClient.subscribe("acc_z");
mqttClient.subscribe("gyro_x");
mqttClient.subscribe("gyro_y");
mqttClient.subscribe("gyro_z");
```

Ilustración 32: Subscripción a los *topics* desde el Arduino

De esta forma, se van a necesitar 8 *topics* a los que subscribirse:

- MPU-6050
  - acc_x : eje x acelerómetro.
  - acc_y : eje y acelerómetro.
  - acc_z : eje z acelerómetro.
  - temperature : temperatura dada por el acelerómetro.
  - gyro_x : eje x giroscopio.

  - gyro_y : eje y giroscopio.
  - gyro_z : eje z giroscopio.
- NEO-6M
  - map : localización obtenida por el GPS.

Los datos enviados al *broker* deben ser cadenas, por lo que las lecturas del MPU son convertidas a tipo *String* antes de ser publicadas al *topic*.

```
// Attempt to publish a value to the topic "temperature"
if (mqttClient.publish("temperature", convert_int16_to_str(temperature))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}
```

Ilustración 33: Publicación de la temperatura obtenida en el *topic* ‘temperature’

4.4.3 Presentación de datos

Una vez que los datos son recogidos por el *broker*, todos los clientes subscritos a dichos *topics* obtendrán la información conforme vaya llegando al servidor.

La aplicación IoT MQTT Panel es un cliente MQTT que se encuentra subscrito a los *topics* definidos en el apartado anterior, y los presenta de forma visual. Para ello, la aplicación dispone de una larga lista de *widgets* de la cual se han elegido los siguientes:

- Gauge: la temperatura variará desde un mínimo de 0 a 50 grados con diferencia de color según el tramo en el que se encuentre.
- Gráfico: los ejes del acelerómetro, como los del giroscopio se disponen en dos gráficos, dentro de los cuales se diferencian tres gráficas de puntos, una por cada eje.
- URI: la URL enviada al *topic* ‘map’ podrá ser lanzada a través de este *widget* una vez recibida.

Esta aplicación se puede obtener gratuitamente en Google Play. Una vez instalada, hay que crear una nueva conexión al *broker* en cuestión. Para ello se pulsa sobre el círculo naranja en la esquina inferior derecha y nos lleva a la configuración del servidor al que conectarnos.

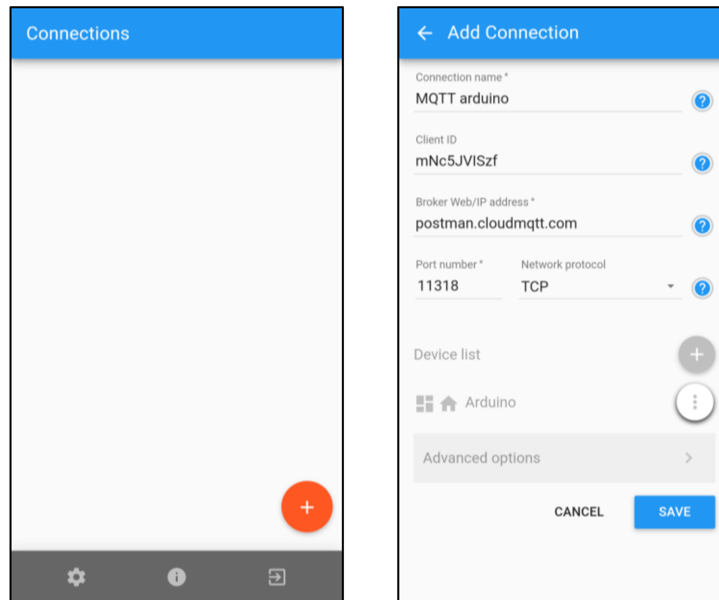

Ilustración 34: Página principal IoT MQTT Panel y nueva conexión

Los campos se rellenan con los datos recogidos en la instancia de CloudMQTT. En opciones avanzadas es necesario proporcionar el nombre del usuario y contraseña con el que acceder a la instancia.

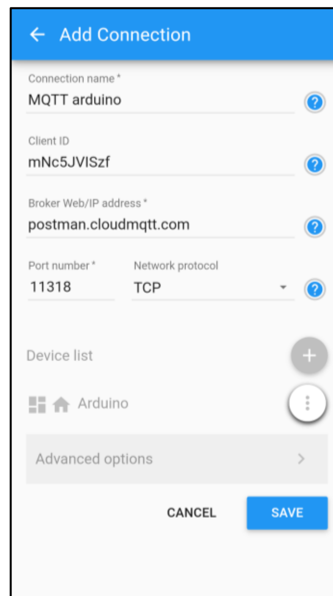

Ilustración 35: Configuración avanzada de la conexión

Una vez conectados al *broker* podemos comenzar a añadir los *widgets* necesarios para la visualización de los datos. Para ello, pulsaremos sobre el círculo naranja que aparece tal y como el de la ilustración 31.

Comenzaremos por el *widget* ‘gauge’, para el *topic* ‘temperature’.

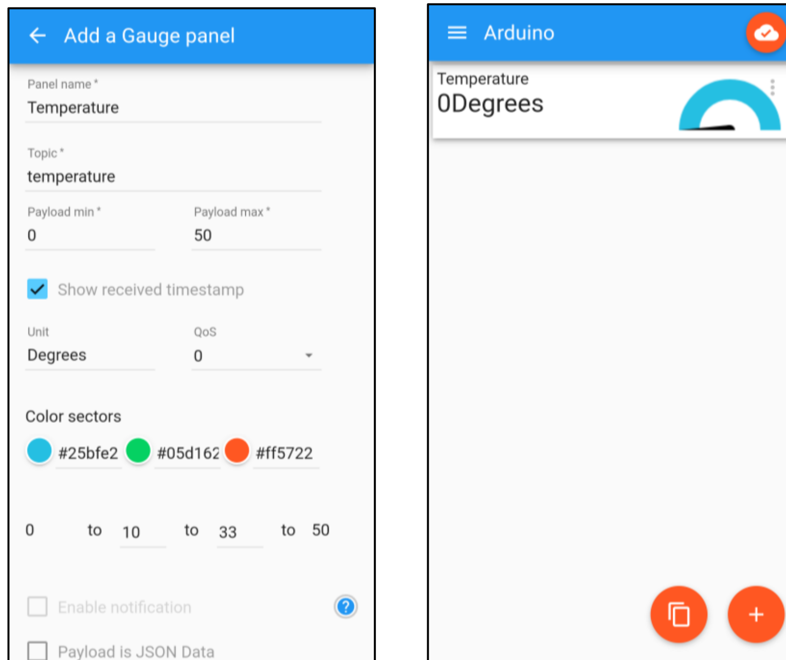

Ilustración 36: Creación Panel Gauge

El acelerómetro y giroscopio se disponen en dos gráficos de líneas, con la misma configuración, exceptuando el nombre de los *topics*.

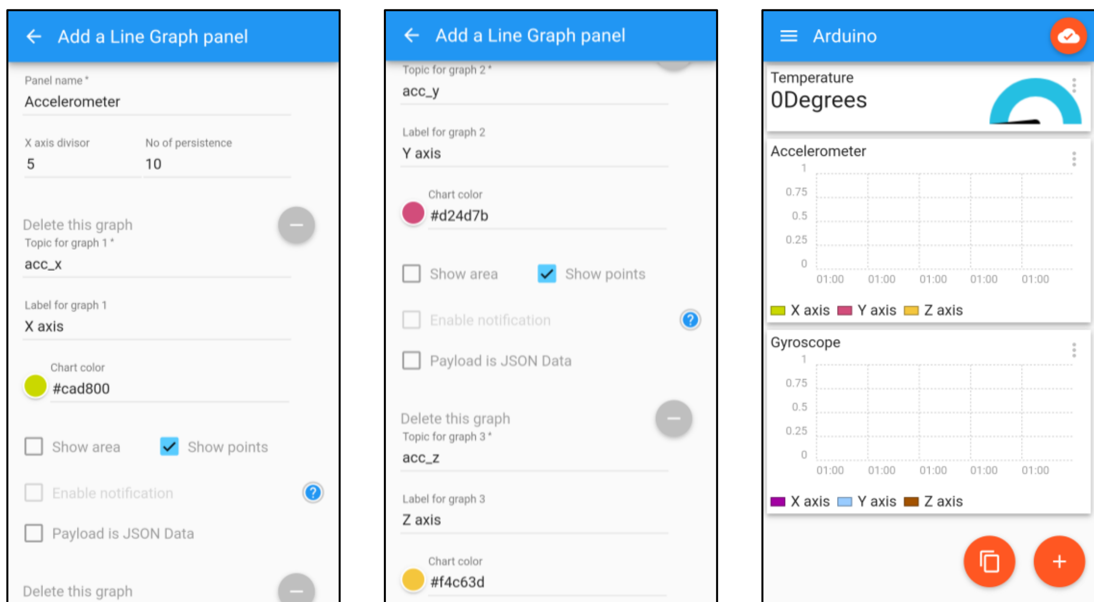

Ilustración 37: Configuración panel gráficos

El último panel está dedicado al GPS, con una configuración muy sencilla. Este *widget* se encarga de lanzar la URL recibida al *topic* 'map'.

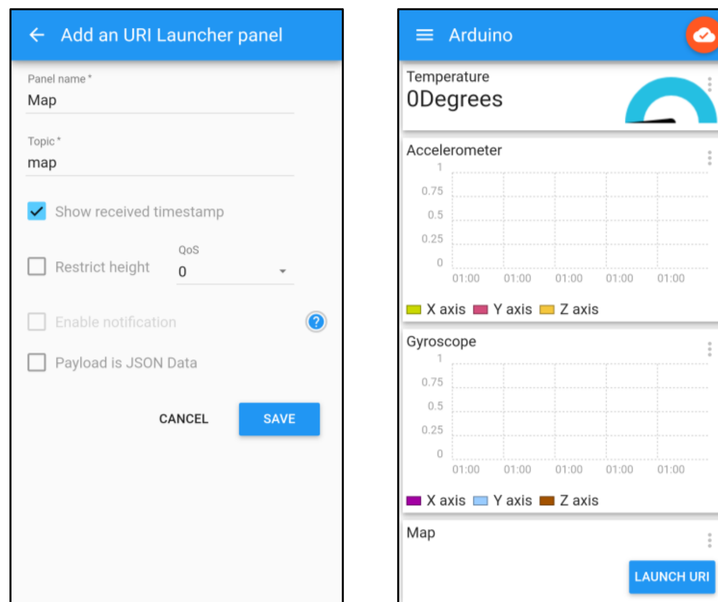

Ilustración 38: Configuración panel URI y *dashboard* final.

El siguiente y penúltimo capítulo, muestra un ejemplo de ejecución del proyecto realizado y los resultados obtenidos tras él.

5 VALIDACIÓN

Una obra sin finalidad se parecería más a los delirios de un loco que a los sobrios esfuerzos del genio o del sabio.

Hume

En este capítulo se exponen las pruebas realizadas con los distintos elementos del sistema para la validación de la correcta realización del proyecto. En él sepresentan los pasos realizados para la ejecución del programa, así como los resultados obtenidos.

5.1 Ejecución

El Arduino presenta una entrada para la carga de tensión, a la cual conectaremos un cable USB con puerto final el ordenador indicado en el subapartado 3.1.1. Ambos módulos se deben conectar a Arduino según la conexión especificada en el capítulo 4, como podemos comprobar en las ilustraciones 10 y 12. Además de esta conexión, es necesario que exista acceso a internet, por lo que debemos conectarle al módulo *Ethernet Shield* un cable Ethernet desde un *router* o desde algún puerto con conexión directa a la red.

Tras ello, abriremos el software Arduino IDE, gracias al cual podremos cargar el programa en el dispositivo.

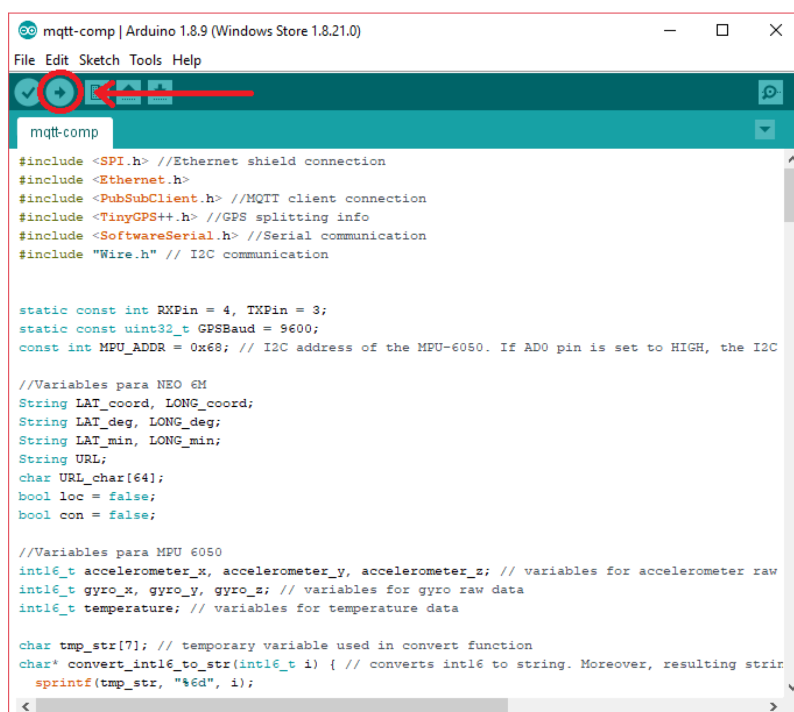

Ilustración 39: Cargar programa en el Arduino

Para que el módulo GPS sea capaz de comunicarse con los satélites, debe tener visión directa con ellos, por lo que es necesario que la antena se encuentre en el exterior.

La instancia se encuentra activa en todo momento, por lo que no es necesario iniciarla. Lo único necesario para una correcta comunicación, es que el cliente se haya conectado al servidor antes de enviar/recibir los datos.

5.2 Resultados

Una vez cargado el programa en el Arduino, simplemente es necesario esperar a que haya una conexión estable a internet, y que el cliente se haya conectado al *broker* con éxito. Tras ello, comenzaremos a recibir datos por parte del MPU-6050, y del NEO-6M una vez que haya encontrado los satélites suficientes para una localización correcta. Tras ello, podremos visualizar en nuestra aplicación móvil los datos de los *topics* a los que estamos subscritos.

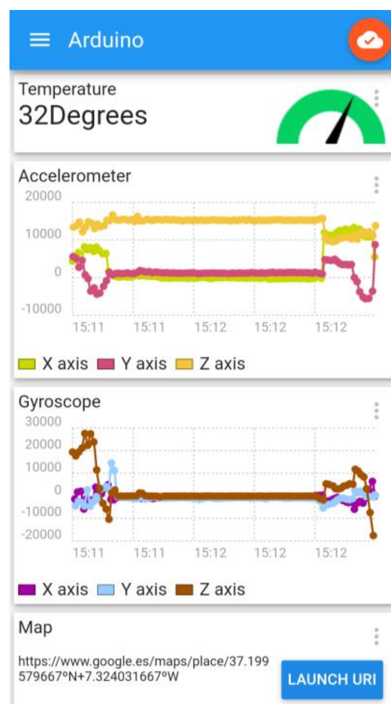

Ilustración 40: Datos recogidos en la aplicación móvil

Como podemos comprobar en la ilustración 40, en un instante de las 15:11 y de las 15:12 horas, tanto el acelerómetro como el giroscopio han recogido movimiento inusual. Esto se debe a que en ese instante el dispositivo fue desplazado con velocidad y girado en el mismo momento. La temperatura como vemos se mantiene en 32 grados. Finalmente, el módulo GPS ha enviado una localización que se puede lanzar mediante el botón 'LAUNCH URI'.

Ilustración 41: Localización obtenida gracias al módulo NEO-6M

En la ilustración 41 es posible visualizar la ubicación del dispositivo gracias a la aplicación de mapas lanzada. En el borde superior de la misma observamos la latitud y la longitud recogidas por el módulo, y en el borde inferior esos mismos valores convertidos a grados y minutos. Gracias a esta información, obtenemos en el mapa una marca color roja llamada ‘chincheta’ que nos indica el punto exacto al que se refieren los datos recogidos.

Tras haber comprobado en este capítulo la ejecución de la aplicación y los resultados obtenidos, el capítulo 6 realiza una visión por la realización del proyecto, llegando a conclusiones y líneas futuras.

6 CONCLUSIONES Y LÍNEAS FUTURAS

En tres palabras se puede resumir todo lo que he aprendido en la vida: sigue hacia delante

Robert Frost

Como cierre de la memoria, se presenta este capítulo en el que se comenta la realización del proyecto y los aspectos más destacables del mismo. A su vez, se realiza un estudio de posibles líneas futuras para una mejora del proyecto.

6.1 Conclusiones

IoT es un campo de trabajo muy actual y con mucho potencial. Antes de este proyecto no había tenido la suerte de realizar ningún tipo de trabajo relacionado con el Internet de las Cosas, lo que me ha llevado a un proceso de indagación y conocimiento de la arquitectura. Sus capacidades son prácticamente infinitas, así como la cantidad de tecnologías en las que se puede implementar. En este caso, se ha decidido dejar a un lado la tecnología tradicional del protocolo HTTP y proceder con una tecnología desconocida.

El protocolo MQTT era totalmente extraño para mí, y esto ha hecho que me lleve una grata sorpresa gracias a su sencillez y manejo tanto desde la parte del cliente como del servidor. La plataforma CloudMQTT ha sido un descubrimiento extraordinario debido a la gran facilidad que presenta a la hora de crear una instancia de servidores en la nube. En menos de 5 minutos es posible tener el *broker* activo a espera de conexiones.

A continuación, haremos un repaso de los objetivos marcados al principio de la memoria para comprobar si han sido alcanzados.

Gracias a las librerías 'Wire' y 'SoftwareSerial' ha sido posible la comunicación de los sensores con el Arduino a través de I^2C y UART. Ambos sensores envían sus datos mediante estos tipos de transmisión, por lo que hemos podido recibir la información e interpretarla correctamente. Ha sido necesario que en todo momento la antena del módulo NEO-6M se haya encontrado en el exterior, ya que no funciona en espacios cerrados y, aún así, el descubrimiento de los satélites necesarios ha llevado más tiempo del esperado.

Mediante la librería 'SPI', se ha conseguido la comunicación entre el módulo *Ethernet Shield* y Arduino Uno. Además de esto, hemos podido configurar correctamente y comenzar la conexión a internet gracias a la librería 'Ethernet'. Esta conexión se ha llevado a cabo gracias a la dirección IP asociada, ya que, comprobando la dirección del Arduino, esta correspondía a la dirección estática definida. Debido a esto podemos saber que no ha habido asociación de IP mediante DHCP.

Gracias a la plataforma CloudMQTT ha sido posible la obtención de una instancia a la que conectarnos en cualquier momento. Este elemento ha sido el punto de unión del resto del sistema, y ha cumplido el requisito principal del modelo de publicación/subscripción en el que se ha basado este proyecto. A partir de esta instancia hemos conectado el Arduino al *broker*. Para ello hemos creado un cliente MQTT, a partir del cual podernos conectar a la instancia mediante la librería 'PubSubClient'. Gracias a esta librería podemos

realizar las funciones de publicación, subscripción y conexión, esenciales para la comunicación entre el cliente y el servidor.

Una vez conectados con el servidor, ha sido posible realizar la subscripción a los temas en cuestión creándolos directamente, ya que no existían con anterioridad.

Una vez subscritos a los *topics* convenientes, y realizada la lectura de los sensores, comenzamos la publicación de dicha información de forma continua consiguiendo, así, un sistema totalmente actualizado.

La aplicación IoT MQTT Panel ha sido el otro punto clave del proyecto. Esta aplicación ha sido configurada para que funcione como otro cliente MQTT, aunque en este caso, al contrario que Arduino, un cliente subscriptor. Gracias a ella hemos podido comprobar visualmente que los datos recogidos por los sensores eran interpretados correctamente y que la conexión entre cliente y servidor funciona sin ningún problema. El *widget* de representación de la localización, aunque no consigue disponer un mapa directo del punto en el que se encuentra, nos ha permitido que otra aplicación realice dicha función y, poder así, visualizar la ubicación exacta.

Tras este análisis de los objetivos, podemos comprobar que todos ellos han sido realizados gracias a la tecnología escogida. Se perseguía una introducción al mundo de Internet de las Cosas mediante el modelo de publicación/subscripción, ahondar en tecnologías ya conocidas y descubrir las necesarias para poner en marcha este proyecto.

6.2 Líneas futuras

CloudMQTT establece conexiones ligeras con los clientes en las que, una vez cerradas dichas conexiones, elimina los datos recibidos en esa sesión. Lo mismo ocurre con el *dashboard*. Una vez que la aplicación se desconecta de la nube, los datos mostrados se pierden ya que ha habido un cierre de sesión.

Como futura mejora, aprovechando la capacidad de almacenaje en la tarjeta SD por parte del *Ethernet Shield*, sería posible mantener un registro de los datos leídos por los sensores o, al menos, recoger el último dato leído para que, si se pierde la conexión, quede registrado su último valor. De esta forma, la aplicación del panel de control podría mostrar los últimos valores recibidos por el *broker*, aun no estando conectados a este.

Los datos recogidos de los sensores pueden servir para una infinidad de cosas hoy en día. A continuación, se indican posibles aplicaciones futuras con la tecnología implementada:

- Prevención de accidentes al volante gracias a las medidas del acelerómetro y giroscopio. Si se detecta que el volante ha sufrido un giro brusco, por ejemplo, al intentar evitar un objeto en la carretera, se puede controlar para que estabilice y vuelva al estado de reposo tras el giro. Además, se puede enviar automáticamente la localización del vehículo obtenida por el GPS una vez detectado el giro por si acaso se hubiera producido un accidente.
- Deportistas entrenándose para carreras de velocidad tipo *Sprints*. Los sensores se podrían portar en el cuerpo tipo pulsera y podrían iniciarse automáticamente al recibir un nivel alto de acelaración para saber la acelaración que se ha conseguido en un tramo determinado. También se podría iniciar un cronómetro al girar la pulsera, o enviar los datos y la ubicación a algún usuario al realizar otro movimiento específico.
- Estos sensores también pueden incluirse en el mundo de los videojuegos. Como hemos dicho en el primer ejemplo, se puede saber la aceleración de un volante, por lo que mediante un mando inalámbrico que se conecte directamente a un juego en línea se podría manejar un coche ficticio, o cualquier juego con necesidad de movimiento.

REFERENCIAS

[1] MQTT 3.1.1 Specification, [En línea]. Disponible en: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

[2] MQTT Getting started, [En línea]. Disponible en: https://www.hivemq.com/mqtt/

[3] Introducing the MQTT Protocol - MQTT Essentials: Part 1, [En línea]. Disponible en: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/

[4] Publish & Subscribe - MQTT Essentials: Part 2, [En línea]. Disponible en: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/

[5] MQTT Publish, Subscribe & Unsubscribe - MQTT Essentials: Part 4, [En línea]. Disponible en: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/

[6] Client, Broker / Server and Connection Establishment - MQTT Essentials: Part 3, [En línea]. Disponible en: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/

[7] CloudMQTT Documentation, [En línea]. Disponible en: https://www.cloudmqtt.com/docs/index.html

[8] Security Enabled M2M Communication using CloudMQTT Broker : IOT Part 19, [En línea]. Disponible en: https://www.engineersgarage.com/contribution/D2D-Communication-MQTT-Protocol-ClodMQTT-Broker

[9] ¿Qué es el Internet de las Cosas y cómo va a cambiar nuestra sociedad?, [En línea]. Disponible en: https://www.blog.andaluciaesdigital.es/que-es-internet-de-las-cosas/

[10] Arduino Ethernet Shield, [En línea]. Disponible en: https://www.arduino.cc/en/Guide/ArduinoEthernetShield

[11] Video tutorial: Using the Arduino Ethernet shield, Part 1 of 2, [En línea]. Disponible en: https://www.youtube.com/watch?v=vcOE2XAQHzY

[12] Efecto Coriolis, [En línea]. Disponible en: https://es.wikipedia.org/wiki/Efecto_Coriolis

[13] Sensor MPU 6050, [En línea]. Disponible en: https://www.diarioelectronicohoy.com/blog/sensor-mpu6050

[14] Guide to NEO-6M GPS Module with Arduino, [En línea]. Disponible en: https://randomnerdtutorials.com/guide-to-neo-6m-gps-module-with-arduino/

[15] Arduino IDE, [En línea]. Disponible en: https://en.wikipedia.org/wiki/Arduino_IDE

[16] Authentication with Username and Password - MQTT Security Fundamentals, [En línea]. Disponible en: https://www.hivemq.com/blog/mqtt-security-fundamentals-authentication-username-password/

[17] Interface ublox NEO-6M GPS Module with Arduino, [En línea]. Disponible en: https://lastminuteengineers.com/neo6m-gps-arduino-tutorial/

[18] Tutorial MPU6050, Acelerómetro y Giroscopio, [En línea]. Disponible en: https://naylampmechatronics.com/blog/45_Tutorial-MPU6050-Aceler%C3%B3metro-y-Giroscopio.html

[19] Introduction to MPU6050, [En línea]. Disponible en: https://www.theengineeringprojects.com/2019/02/introduction-to-mpu6050.html

[20] La mejora de las soluciones en IoT pasa por la colaboración, [En línea]. Disponible en: https://blogthinkbig.com/iot-telefonica-softwareag

[21] IoT MQTT Panel FAQ and feedback, [En línea]. Disponible en: http://www.snrelectronicsblog.com/iot/iot-mqtt-panel-user-guide/

[22] IoT MQTT Panel, [En línea]. Disponible en: https://play.google.com/store/apps/details?id=snr.lab.iotmqttpanel.prod

[23] MPU-6050 Datasheet, [En línea]. Disponible en: https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

[24] Modelo Publicación/Subscripción, [En línea]. Disponible en: https://www.ibm.com/support/knowledgecenter/es/SSMKHH_10.0.0/com.ibm.etools.mft.doc/aq01120_.htm

[25] Happ, Daniel & Karowski, Niels & Menzel, Thomas & Handziski, Vlado & Wolisz, Adam. (2016). Meeting IoT platform requirements with open pub/sub solutions. Annals of Telecommunications. 72. 10.1007/s12243-016-0537-4.

[26] Comunicaciones con Arduino, [En línea]. Disponible en: https://aprendiendoarduino.wordpress.com/2014/11/18/tema-6-comunicaciones-con-arduino-4/

ANEXOS

Los programas realizados para ambos sensores presentan una arquitectura similar en la que se distinguen las siguientes partes:

1. Incluir librerías necesarias
2. Definir variables
3. Definir objetos
4. Función *setup*
5. Función *loop*

En la primera parte se incluyen las librerías nombradas en los subapartados anteriores. Sin ellas no sería posible que el programa funcionara correctamente.

El segundo apartado reúne la definición de constantes como la dirección de esclavo del MPU o los números de pines para la comunicación serie, así como variables para almacenar la información leída de ambos módulos.

Los objetos definidos son los necesarios para las comunicaciones de internet, con los sensores y con el *broker*.

La función *setup* se ejecuta una sola vez y en ella se incluye la configuración necesaria para el programa. Se inicia la comunicación serie, la conexión a internet y el servidor MQTT. En esta función también se configura el módulo MPU para que se encuentre iniciado.

Por último, la función *loop* se ejecuta una vez tras otra, por lo que en ella se incluye la lectura de los sensores y la subscripción/publicación de la información.

1. Aplicación IoT para Arduino con MPU-6050

```
#include <SPI.h>
#include <Ethernet.h>
#include <PubSubClient.h>
#include "Wire.h" // This library allows you to communicate with I2C devices.


const int MPU_ADDR = 0x68; // I2C address of the MPU-6050

int16_t accelerometer_x, accelerometer_y, accelerometer_z; // variables for accelerometer raw data

int16_t gyro_x, gyro_y, gyro_z; // variables for gyro raw data

int16_t temperature; // variables for temperature data

char tmp_str[7]; // temporary variable used in convert function

char* convert_int16_to_str(int16_t i) { // converts int16 to string
  sprintf(tmp_str, "%6d", i);
  return tmp_str;
}

// Function prototypes
void subscribeReceive(char* topic, byte* payload, unsigned int length);
```

```
// Set your MAC address and IP address here
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 100);
IPAddress dnServer(192, 168, 1, 1);
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);

// MQTT server hostname
const char* server = "postman.cloudmqtt.com";

// Ethernet and MQTT related objects
EthernetClient ethClient;
PubSubClient mqttClient(ethClient);

void setup() {
  // Useful for debugging purposes
  Serial.begin(115200);

  // Start the ethernet connection
  Ethernet.begin(mac, ip, dnServer, gateway, subnet);

  delay(3000); // Ethernet takes some time to boot

  // Set the MQTT server to the server stated above ^
  mqttClient.setServer(server, 11318);

   // Attempt to connect to the server with the ID "dolrairom", and intance's user and password.
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
    Serial.println("Attempting to connect");
  }
  Serial.println("Connection has been established, well done");
  mqttClient.setCallback(subscribeReceive);

  // MPU configuration
  Wire.begin();
  Wire.beginTransmission(MPU_ADDR); // Begins a transmission to the I2C slave
  Wire.write(0x6B); // PWR_MGMT_1 register
  Wire.write(0); // set to zero (wakes up the MPU-6050)
  Wire.endTransmission(true);

}

void loop() {
  mqttClient.loop();

 // Ensure that we are subscribed to the correct topics
 mqttClient.subscribe("temperature");
 mqttClient.subscribe("acc_x");
 mqttClient.subscribe("acc_y");
 mqttClient.subscribe("acc_z");
 mqttClient.subscribe("gyro_x");
 mqttClient.subscribe("gyro_y");
 mqttClient.subscribe("gyro_z");

 Wire.beginTransmission(MPU_ADDR);
 Wire.write(0x3B); // starting with register 0x3B (ACCEL_XOUT_H)
 Wire.endTransmission(false); // the parameter indicates that the Arduino will send a restart
 Wire.requestFrom(MPU_ADDR, 7*2, true); // request a total of 7*2=14 registers
```

```
// "Wire.read()<<8 | Wire.read();" means two registers are read and stored in the same variable
accelerometer_x = Wire.read()<<8 | Wire.read(); //reading registers: 0x3B(ACCEL_XOUT_H) and 0x3C(ACCEL_XOUT_L)
accelerometer_y = Wire.read()<<8 | Wire.read(); //reading registers: 0x3D(ACCEL_YOUT_H) and 0x3E(ACCEL_YOUT_L)
accelerometer_z = Wire.read()<<8 | Wire.read(); //reading registers: 0x3F(ACCEL_ZOUT_H) and 0x40(ACCEL_ZOUT_L)
temperature = Wire.read()<<8 | Wire.read(); // reading registers: 0x41 (TEMP_OUT_H) and 0x42 (TEMP_OUT_L)
gyro_x = Wire.read()<<8 | Wire.read(); // reading registers: 0x43 (GYRO_XOUT_H) and 0x44 (GYRO_XOUT_L)
gyro_y = Wire.read()<<8 | Wire.read(); // reading registers: 0x45 (GYRO_YOUT_H) and 0x46 (GYRO_YOUT_L)
gyro_z = Wire.read()<<8 | Wire.read(); // reading registers: 0x47 (GYRO_ZOUT_H) and 0x48 (GYRO_ZOUT_L)

temperature = temperature/340.00+36.53;

   // Attempt to publish a value to the topic "temperature"
if (mqttClient.publish("temperature", convert_int16_to_str(temperature))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}

// Attempt to publish a value to the topic "acc_x"
if (mqttClient.publish("acc_x", convert_int16_to_str(accelerometer_x))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}


// Attempt to publish a value to the topic "acc_y"
if (mqttClient.publish("acc_y", convert_int16_to_str(accelerometer_y))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}
 // Attempt to publish a value to the topic "acc_z"
 if (mqttClient.publish("acc_z", convert_int16_to_str(accelerometer_z))){
   Serial.println("Publish message success");
 }
 else{
   Serial.println("Could not send message :(");
   while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
     Serial.println("Attempting to reconnect");
   }
 }

 // Attempt to publish a value to the topic "gyro_x"
 if (mqttClient.publish("gyro_x", convert_int16_to_str(gyro_x))){
   Serial.println("Publish message success");
 }
 else{
   Serial.println("Could not send message :(");
   while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
     Serial.println("Attempting to reconnect");
   }
 }
```

```
  // Attempt to publish a value to the topic "gyro_z"
  if (mqttClient.publish("gyro_z", convert_int16_to_str(gyro_z))){
    Serial.println("Publish message success");
  }
  else{
    Serial.println("Could not send message :(");
    while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
      Serial.println("Attempting to reconnect");
    }
  }

}

void subscribeReceive(char* topic, byte* payload, unsigned int length)
{
  // Print the topic
  Serial.print("Topic: ");
  Serial.println(topic);

  // Print the message
  Serial.print("Message: ");
  for (int i = 0; i < length; i ++)
  {
    Serial.print(char(payload[i]));
  }

  Serial.println("");
}
```

2. Aplicación IoT para Arduino con NEO-6M

```
#include <SPI.h>
#include <Ethernet.h>
#include <PubSubClient.h>
#include <TinyGPS++.h>
#include <SoftwareSerial.h>

static const int RXPin = 4, TXPin = 3;
static const uint32_t GPSBaud = 9600;

String LAT_coord, LONG_coord;
String LAT_deg, LONG_deg;
String LAT_min, LONG_min;
String URL;
char URL_char[64];
bool loc = false;
bool con = false;

// Function prototypes
void subscribeReceive(char* topic, byte* payload, unsigned int length);

// The TinyGPS++ object
TinyGPSPlus gps;

// The serial connection to the GPS device
SoftwareSerial ss(RXPin, TXPin);
```

```
// Internet configuration
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 100);
IPAddress dnServer(192, 168, 1, 1);
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);

// Make sure to leave out the http and slashes!
const char* server = "postman.cloudmqtt.com";

// Ethernet and MQTT related objects
EthernetClient ethClient;
PubSubClient mqttClient(ethClient);

void setup()
{
  Serial.begin(115200);
  ss.begin(GPSBaud);

  // Start the ethernet connection
  Ethernet.begin(mac, ip, dnServer, gateway, subnet);

  delay(3000); // Ethernet takes some time to boot

  // Set the MQTT server to the server stated above ^
  mqttClient.setServer(server, 11318);

// Attempt to connect to the server with the ID "dolrairom", and intance's user and password.
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
    Serial.println("Attempting to connect");
    delay(1000);
  }
  Serial.println("Connection has been established, well done");
  mqttClient.setCallback(subscribeReceive);
}

void loop()
{
  mqttClient.loop();

 // Ensure that we are subscribed to the topic "map"
  mqttClient.subscribe("map");

  // This sketch displays information every time a new sentence is correctly encoded.
  while (ss.available() > 0)
    if (gps.encode(ss.read()))
      displayInfo();


  if (millis() > 5000 && gps.charsProcessed() < 10)
  {
    Serial.println(F("No GPS detected: check wiring."));
    while(true);
  }
```

```
  if(loc){
    // Attempt to publish a value to the topic "map". URLsize = 64 caracteres
    LAT_deg = gps.location.rawLat().deg;
    LONG_deg = gps.location.rawLng().deg;
    LAT_min = gps.location.rawLat().billionths;
    LONG_min = gps.location.rawLng().billionths;
    if((gps.location.rawLat().negative ? "-" : "+") == "-"){
     LAT_coord = "W";
    }
    else{
     LAT_coord = "N";
    }
    if((gps.location.rawLng().negative ? "-" : "+") == "-"){
     LONG_coord = "W";
    }
    else{
     LONG_coord = "N";
    }
    URL = "https://www.google.es/maps/place/"+LAT_deg+"."+LAT_min+"°"+LAT_coord+"+";
    URL += LONG_deg+"."+LONG_min+"°"+LONG_coord+"/";
    Serial.println(URL);
    URL.toCharArray(URL_char, 64);
    if (mqttClient.publish("map", URL_char)){
      Serial.println("Publish message success");
    }
    else{
      Serial.println("Could not send message :(");
      while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJlI3Q")){
        Serial.println("Attempting to reconnect");
        delay(1000);
      }
    }
  }
}

void subscribeReceive(char* topic, byte* payload, unsigned int length)
{
  // Print the topic
  Serial.print("Topic: ");
  Serial.println(topic);

  // Print the message
  Serial.print("Message: ");
  for (int i = 0; i < length; i ++)
  {
    Serial.print(char(payload[i]));
  }
  Serial.println("");
}

void displayInfo()
{
  Serial.print(F("Location: "));
  if (gps.location.isValid()){
    Serial.print(F("VALID"));
    loc = true;
  }
  else{
    Serial.print(F("INVALID"));
    loc = false;
  }

  Serial.println();
}
```

3. Aplicación IoT para Arduino con MPU-6050 y NEO-6M

```
#include <SPI.h> //Ethernet shield connection
#include <Ethernet.h>
#include <PubSubClient.h> //MQTT client connection
#include <TinyGPS++.h> //GPS splitting info
#include <SoftwareSerial.h> //Serial communication
#include "Wire.h" // I2C communication


static const int RXPin = 4, TXPin = 3;
static const uint32_t GPSBaud = 9600;
const int MPU_ADDR = 0x68; // I2C address of the MPU-6050.

//Variables para NEO 6M
String LAT_coord, LONG_coord;
String LAT_deg, LONG_deg;
String LAT_min, LONG_min;
String URL;

bool loc = false;
bool con = false;

//Variables para MPU 6050
int16_t accelerometer_x, accelerometer_y, accelerometer_z; // variables for accelerometer raw data
int16_t gyro_x, gyro_y, gyro_z; // variables for gyro raw data
int16_t temperature; // variables for temperature data

char tmp_str[7]; // temporary variable used in convert function
char* convert_int16_to_str(int16_t i) { // converts int16 to string.
  sprintf(tmp_str, "%6d", i);
  return tmp_str;
}

// Function prototypes
void subscribeReceive(char* topic, byte* payload, unsigned int length);

// The TinyGPS++ object
TinyGPSPlus gps;

// The serial connection to the GPS device
SoftwareSerial ss(RXPin, TXPin);

// Internet Configuration
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 100);
IPAddress dnServer(192, 168, 1, 1);
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);

// MQTT server hostname
const char* server = "postman.cloudmqtt.com";

// Ethernet and MQTT related objects
EthernetClient ethClient;
PubSubClient mqttClient(ethClient);
```

```
void setup()
{
  Serial.begin(115200);
  ss.begin(GPSBaud);

  // Start the ethernet connection
  Ethernet.begin(mac, ip, dnServer, gateway, subnet);

  delay(3000); // Ethernet takes some time to boot!

  // MPU configuration
  Wire.begin();
  Wire.beginTransmission(MPU_ADDR); // Begins a transmission to the I2C slave
  Wire.write(0x6B); // PWR_MGMT_1 register
  Wire.write(0); // set to zero (wakes up the MPU-6050)
  Wire.endTransmission(true);

  const char* server = "postman.cloudmqtt.com";
  mqttClient.setServer(server, 11318);

 // Attempt to connect to the server with the ID "dolrairom", and intance's user and password.
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
    Serial.println("Attempting to connect");
    delay(1000);
  }
  Serial.println("Connection has been established, well done");
  mqttClient.setCallback(subscribeReceive);


}

void loop()
{
  mqttClient.loop();

  // Ensure that we are subscribed to the correct topics
  mqttClient.subscribe("map");
  mqttClient.subscribe("temperature");
  mqttClient.subscribe("acc_x");
  mqttClient.subscribe("acc_y");
  mqttClient.subscribe("acc_z");
  mqttClient.subscribe("gyro_x");
  mqttClient.subscribe("gyro_y");
  mqttClient.subscribe("gyro_z");


   while (ss.available() > 0)
     if (gps.encode(ss.read()))
       displayInfo();

   if (millis() > 5000 && gps.charsProcessed() < 10)
   {
     Serial.println(F("No GPS detected: check wiring."));
     while(true);
   }
```

```
if(loc){
  URL = "https://www.google.es/maps/place/";

  LAT_deg = gps.location.rawLat().deg;
  URL += LAT_deg;
  URL += ".";
  Serial.println(LAT_deg);
  LAT_min = gps.location.rawLat().billionths;
  URL += LAT_min;
  Serial.println(LAT_min);
  LAT_coord = gps.location.rawLat().negative ? "-" : "+";
  if(LAT_coord == "-"){
   Serial.println("Negativo lat: W");
   LAT_coord = "W";
  }
  else{
   Serial.println("Positivo lat: N");
   LAT_coord = "N";
  }
  URL += "°";
  URL += LAT_coord;
  URL += "+";

  LONG_deg = gps.location.rawLng().deg;
  URL += LONG_deg;
  Serial.println(LONG_deg);
  URL += ".";
  LONG_min = gps.location.rawLng().billionths;
  URL += LONG_min;
  Serial.println(LONG_min);

  LONG_coord = gps.location.rawLng().negative ? "-" : "+";
  if(LONG_coord == "-"){
    Serial.println("Negativo lon: W");
   LONG_coord = "W";
  }
  else{
    Serial.println("Positivo lon: N");
   LONG_coord = "N";
  }
  URL += "°";
  URL += LONG_coord;
  URL += "/";


  URL.toCharArray(URL_char, 64);
  if (mqttClient.publish("map", URL_char)){
    Serial.println("Publish message success");
  }
  else{
    Serial.println("Could not send message :(");
    while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
      Serial.println("Attempting to reconnect");
      delay(1000);
    }
  }
}
```

```
Wire.beginTransmission(MPU_ADDR);
Wire.write(0x3B); // starting with register 0x3B (ACCEL_XOUT_H)
Wire.endTransmission(false); // the parameter indicates that the Arduino will send a restart.
Wire.requestFrom(MPU_ADDR, 7*2, true); // request a total of 7*2=14 registers

// "Wire.read()<<8 | Wire.read();" means two registers are read and stored in the same variable
accelerometer_x = Wire.read()<<8 | Wire.read(); //reading registers: 0x3B(ACCEL_XOUT_H) and 0x3C(ACCEL_XOUT_L)
accelerometer_y = Wire.read()<<8 | Wire.read(); //reading registers: 0x3D(ACCEL_YOUT_H) and 0x3E(ACCEL_YOUT_L)
accelerometer_z = Wire.read()<<8 | Wire.read(); //reading registers: 0x3F(ACCEL_ZOUT_H) and 0x40(ACCEL_ZOUT_L)
temperature = Wire.read()<<8 | Wire.read(); // reading registers: 0x41 (TEMP_OUT_H) and 0x42 (TEMP_OUT_L)
gyro_x = Wire.read()<<8 | Wire.read(); // reading registers: 0x43 (GYRO_XOUT_H) and 0x44 (GYRO_XOUT_L)
gyro_y = Wire.read()<<8 | Wire.read(); // reading registers: 0x45 (GYRO_YOUT_H) and 0x46 (GYRO_YOUT_L)
gyro_z = Wire.read()<<8 | Wire.read(); // reading registers: 0x47 (GYRO_ZOUT_H) and 0x48 (GYRO_ZOUT_L)

temperature = temperature/340.00+36.53;

   // Attempt to publish a value to the topic "temperature"
if (mqttClient.publish("temperature", convert_int16_to_str(temperature))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}

// Attempt to publish a value to the topic "acc_x"
if (mqttClient.publish("acc_x", convert_int16_to_str(accelerometer_x))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}


// Attempt to publish a value to the topic "acc_y"
if (mqttClient.publish("acc_y", convert_int16_to_str(accelerometer_y))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");

  }
}

// Attempt to publish a value to the topic "acc_z"
if (mqttClient.publish("acc_z", convert_int16_to_str(accelerometer_z))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}

// Attempt to publish a value to the topic "gyro_x"
if (mqttClient.publish("gyro_x", convert_int16_to_str(gyro_x))){
  Serial.println("Publish message success");
}
else{
  Serial.println("Could not send message :(");
  while(!mqttClient.connect("dolrairom", "pbpsqoco", "lfzizmmJ2K4R")){
    Serial.println("Attempting to reconnect");
  }
}
```

```
  // Attempt to publish a value to the topic "gyro_y"
  if (mqttClient.publish("gyro_y", convert_int16_to_str(gyro_y))){
    Serial.println("Publish message success");
  }
  else{
    Serial.println("Could not send message :(");
    while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
      Serial.println("Attempting to reconnect");
    }
  }

  // Attempt to publish a value to the topic "gyro_z"
  if (mqttClient.publish("gyro_z", convert_int16_to_str(gyro_z))){
    Serial.println("Publish message success");
  }
  else{
    Serial.println("Could not send message :(");
    while(!mqttClient.connect("dolrairom", "pbpsqoco", "1fzizmmJ2K4R")){
      Serial.println("Attempting to reconnect");
    }
  }

}

void subscribeReceive(char* topic, byte* payload, unsigned int length)
{
  // Print the topic
  Serial.print("Topic: ");
  Serial.println(topic);

  // Print the message
  Serial.print("Message: ");
  for (int i = 0; i < length; i ++)
  {
    Serial.print(char(payload[i]));
  }

  Serial.println("");
}

void displayInfo()
{
  Serial.print(F("Location: "));
  if (gps.location.isValid()){
    Serial.print(F("VALID"));
    loc = true;
  }
  else{
    Serial.print(F("INVALID"));
    loc = false;
  }

  Serial.println();
}
```