



UNIVERSITAT
POLITÀCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de un proxy MQTT para la plataforma OpenWeather

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Cambareri, Saverio

Tutores: Tavares de Araujo Cesariny Calafate, Carlos Miguel
Manzoni, Pietro

2020/21

Implementación de un proxy MQTT para la plataforma OpenWeather

Resumen

En los últimos años el protocolo MQTT se ha convertido en uno de los principales pilares del Internet de las cosas por su ligereza y flexibilidad. De hecho, que los dispositivos IoT estén conectados entre sí y puedan intercambiar informaciones es un requisito imprescindible para este tipo de sistemas. Respecto al tipo de información intercambiada, ésta puede variar en base al entorno de actuación, que puede ser tanto industrial como doméstico. En el caso concreto de este proyecto la información intercambiada será de tipo meteorológico, y nos centraremos en un entorno privado/doméstico, pero con las debidas medidas se podría también adaptar a un entorno rural o industrial.

Este trabajo fin de grado trata de diseñar un nodo proxy basado en un bróker MQTT que se encarga de contestar a peticiones para la plataforma meteorológica OpenWeather. Los clientes podrán conectarse al bróker y suscribirse a los topics relativos a la información que sea de su interés. El proxy tiene que capturar estas peticiones y realizar las consultas oportunas con el API de la plataforma OpenWeather, publicando los datos solicitados. El nodo proxy y el bróker (Mosquitto) se implementarán en Python usando una Raspberry Pi.

Palabras clave: IoT, proxy, HTTP, REST, MQTT, Python, Mosquitto, Raspberry PI.

Abstract

In recent years, the MQTT protocol has become one of the main pillars of the Internet of Things due to its lightness and flexibility. For IoT devices, being connected to each other and exchanging information is an essential requirement, the type of information exchanged may vary based on the operating environment, which can be both industrial and domestic. In this specific case, the information exchanged will be meteorological and the environment domestic, but with the proper measures it could also be adapted to an industrial environment.

This work tries to design a proxy node, using the Python language, based on an MQTT broker that is in charge of answering requests for the OpenWeather meteorological platform. Clients will be able to connect to the broker and subscribe to topics related to information of interest. The proxy has to capture these requests and make the appropriate queries with the OpenWeather platform API, publishing the requested data. The proxy node and the broker (Mosquitto) will be located on a Raspberry PI.

Keywords: IoT, proxy, HTTP, REST, MQTT, Python, Mosquitto, Raspberry PI.

Tabla de contenidos

1.	Introducción	10
1.1	Motivación.....	11
1.2	Objetivos	12
1.3	Metodología.....	12
1.4	Estructura del documento.....	13
2.	Estado del arte y características de las tecnologías implementadas.....	15
2.1	Internet of Things	15
2.1.1	Peculiaridades de la IoT	16
2.1.2	Protocolos	17
2.1.2.1	HTTP.....	18
2.1.2.2	MQTT	19
2.1.3	Raspberry Pi.....	21
2.1.4	Python	22
2.1.4.1	Librerías.....	23
3.	Análisis del problema.....	26
3.1	Identificación y análisis de soluciones posibles.....	26
3.1.1	API OpenWeather.....	26
3.1.1.1	Current weather data	26
3.1.1.2	Hourly forecast 4 days.....	31
3.1.1.3	Daily forecast 16 days	32
3.1.1.4	Solar radiation API	33
3.1.1.5	Air Pollution API.....	34
3.1.2	Adaptación del API al protocolo MQTT	35
3.1.2.1	Topic	35
3.1.2.1.1	Estructura	35
3.1.2.1.2	Comodines	36
3.1.2.2	Publicadores y suscriptores.....	37
3.1.2.2.1	Last Will and Testament	39
3.2	Solución propuesta	39
3.2.1	Arquitectura del sistema	39



3.2.2	Esquema de composición de un topic	41
3.2.3	Proxy	42
3.2.3.1	Recepción y respuestas a las peticiones	42
3.2.3.2	Gestión de topic y clientes	43
3.2.4	Cliente	44
3.3	Plan de trabajo	44
4.	Diseño de la solución	46
4.1	Instalación y configuración de componentes.....	46
4.1.1	Mosquitto.....	46
4.1.2	Python y librerías.....	48
4.2	Programación del proxy	50
4.2.1	Main	50
4.2.2	Función On_Connect	52
4.2.3	Función On_Message.....	53
4.2.4	Función Api_request	57
4.2.5	Peticiones a la API de OpenWeather	60
4.2.5.1	Función current_weather	61
4.2.5.2	Función forecast	66
4.2.5.3	Función solar_radiation.....	67
4.2.5.4	Función air_pollution.....	69
4.3	Programación del cliente.....	69
4.4	Función on_connect.....	72
4.5	Función on_message.....	73
5.	Validación y pruebas.....	75
5.1	Testeo de peticiones simples	75
5.2	Testeo de peticiones con frecuencia	78
5.3	Testeo de carga y esfuerzo	80
6.	Conclusiones	83
6.1	Dificultades encontradas	83
6.2	Posibles trabajos futuros	83
6.3	Ejemplo de uso posible	84
	Referencias.....	86

Tabla de ilustraciones

Ilustración 1 - Logo de la plataforma OpenWeather (Wikipedia).....	10
Ilustración 2 – Evolución del número de dispositivos conectados al IoT en el mundo. (Statista)	15
Ilustración 3 – Arquitectura de la IoT (A: tres capas) (B: cinco capas). (Hindawi)	17
Ilustración 4 – Arquitectura de publicación / suscripción MQTT (mqtt.org).	20
Ilustración 5 - Logo broker MQTT Eclipse Mosquitto. (Mosquitto.org).....	20
Ilustración 6 – Placa Raspberry Pi 3	21
Ilustración 7 – Logo de Python (Wikipedia)	23
Ilustración 8 – Ejemplo salida petición Current Weather API (Openweathermap.org).	30
Ilustración 9 – Arquitectura del sistema.....	40
Ilustración 10 – Esquema de composición de un topic.	41
Ilustración 11 – Captura salida comando mosquitto-v.	47
Ilustración 12 – Importación de librerías.....	50
Ilustración 13 – Modulo main del proxy.	50
Ilustración 14 – Función On_Connect.	52
Ilustración 15 – Función On_Message, topic “offline”	54
Ilustración 16 – Función <i>remove_client</i>	54
Ilustración 17 – Función <i>On_Message</i> , topic “sub”	55
Ilustración 18 – Función <i>add_client</i>	56
Ilustración 19 – Función <i>On_Message</i> topic “request”, decodificación mensaje e inicialización variables	56
Ilustración 20 – Función <i>On_Message</i> , llamada a la función <i>api_request</i> según el valor de frecuencia.....	57
Ilustración 21 – Función <i>Api_request</i> , descomposición topic.....	58
Ilustración 22 – Función <i>api_request</i> , comprobación colección de datos.....	59
Ilustración 23 – Función <i>api_request</i> , zona Thread. Inicialización de un nuevo <i>client</i> y verificación de existencia y frecuencia mínima del topic en el diccionario.	59
Ilustración 24 – Función <i>get_min_freq</i>	60
Ilustración 25 – Función <i>api_request</i> , zona Thread. Llamada a la función para realizar la petición, publicación mensaje y detenimiento ejecución según la frecuencia especificada.	60
Ilustración 26 – Función <i>current_weather</i> , inicializaciones variables.....	62
Ilustración 27 – Función <i>current_weather</i> , composición URL según la tipología de búsqueda.....	62
Ilustración 28 – Función <i>current_weather</i> , comprobación variable data, petición a la API de OpenWeather, comprobación código de estado mensaje, y llamada a la función para generar la respuesta para el cliente.....	63
Ilustración 29 – Función <i>get_current_weather</i> , inicializaciones variables	64
Ilustración 30 – Función <i>get_current_weather</i> , generación diccionario auxiliar.	65
Ilustración 31 – Función <i>get_current_weather</i> , generación de respuesta para el cliente.	65
Ilustración 32 – Función <i>forecast</i> , inicializaciones variables.....	66

Ilustración 33 – Función forecast, petición a la API de OpenWeather y llamada a la función para la generación de la respuesta según el “period” elegido por el cliente.	67
Ilustración 34 – Función solar_radiation, gestión datos, coordenadas geográficas y composición URL.	68
Ilustración 35 – Función solar_radiation, petición a la API de OpenWeather, y llamada a la función para generar la respuesta para el cliente.	68
Ilustración 36 – Función air_pollution, composición URL según parámetro ‘period’. 69	
Ilustración 37 – Client, importación librerías.	70
Ilustración 38 – Client, inicializaciones de variables.	71
Ilustración 39 – Client, gestión argumentos de entrada y verificación de wildcard.	71
Ilustración 40 – Client, conexión al broker.	72
Ilustración 41 – Client, función <i>on_connect</i>	73
Ilustración 42 – Client, función <i>on_message</i>	73
Ilustración 43 – Salida comando <i>mosquitto -v</i>	75
Ilustración 44 – Ejecución proxy y mensaje de conexión exitosa.	75
Ilustración 45 – Salida broker Mosquitto después de la conexión exitosa con el proxy 76	
Ilustración 46 – Ejemplo de petición sin frecuencia de respuesta donde se piden todos los parámetros (“all”).	76
Ilustración 47 – Ejemplo petición sin frecuencia de respuesta donde se piden algunos datos específicos	77
Ilustración 48 – Ejemplo petición donde se pide un dato inexistente.	77
Ilustración 49 – Ejemplo petición de datos meteorológicos para diferentes localidades.	78
Ilustración 50 – Ejemplo petición para probabilidad de precipitaciones. Valor mínimo = 0 (0%), valor máximo = 1 (100%).	78
Ilustración 51 – Ejemplo petición con una frecuencia de respuesta de 10 segundos.	79
Ilustración 52 – Sistema de riego en un entorno rural.	84

Índice de Tablas

Tabla 1 – Colecciones de datos plataforma OpenWeather.....	11
Tabla 2 – Especificaciones placa Raspberry Pi 3 Model B v1.2.....	22
Tabla 3 – Parámetros salida Current weather API que se incluyen en el proxy.	31
Tabla 4 – Parámetros salida Hourly Forecast 4 days API que se incluyen en el proxy.	32
Tabla 5 – Parámetros salida Daily Forecast 16 days API que se incluyen en el proxy... ..	32
Tabla 6 – Parámetros salida Solar Radiation API que se incluyen en el proxy	33
Tabla 7 – Parámetros salida Air Pollution API que se incluyen en el proxy	34
Tabla 8 – Códigos de estado de los mensajes.....	63
Tabla 9 – Test de carga. Listado de clientes con su topic y frecuencia.	80
Tabla 10 – Retrasos de los tiempos de publicación de cada topic analizados en 10 mensajes, y retraso acumulado (en segundos).....	81



1. Introducción

Conocer lo que ocurre a nuestro alrededor es fundamental para ser conscientes de cuáles son los problemas del mundo actual, y no es ningún misterio que uno de los temas que más está afectando nuestro presente, y que probablemente afectará drásticamente nuestro futuro, es el ambiental.

Los problemas medioambientales son varios, y van desde el calentamiento global hasta la contaminación de los mares y océanos.

Una vez que conocer lo que ocurre a nuestro alrededor es fundamental, la plataforma OpenWeather surge como una vía muy oportuna para este propósito. La empresa OpenWeather es formada por un equipo de expertos en TI y Data Scientists que han estado trabajando en ciencia de datos meteorológicos desde 2014. Para cada punto del mundo, OpenWeather proporciona datos meteorológicos históricos, actuales, y pronosticados en su sitio web OpenWeatherMap. Además, es posible integrar los datos ofrecidos por Openweather a través de una API basada en una interfaz REST.



Ilustración 1 - Logo de la plataforma OpenWeather (Wikipedia)

La API de OpenWeather permite el acceso a información sobre varias colecciones de datos, entre las cuales las más interesantes se pueden apreciar en la siguiente tabla.

Tabla 1 – Colecciones de datos plataforma OpenWeather

Colección de datos	Descripción
Current Weather Data	Datos meteorológicos actuales de cualquier lugar de la Tierra, incluso más de 200.000 ciudades.
Hourly Forecast 4 days	Pronóstico por hora durante 4 días, con 96 marcas de tiempo.
Daily Forecast 16 days	Pronóstico diario de 16 días disponible en cualquier lugar o ciudad.
Solar Radiation API	Datos de radiación solar actuales y datos de radiación solar pronosticados con 16 días de anticipación para cualesquiera coordenadas del mundo.
Air Pollution API	La API de contaminación del aire proporciona datos de contaminación del aire actuales, pronosticados e históricos para cualquier coordenada del mundo.

De todas estas colecciones de datos, proporcionaremos más detalles en los capítulos “Análisis del problema” y “Diseño de la solución”.

1.1 Motivación

Lo motivación principal de este proyecto ha sido trabajar en algo que tiene que ver con el medioambiente. En este contexto, la Agenda 2030 y los Objetivos de Desarrollo Sostenible (ODS) representan un enfoque global e interconectado para comprender y abordar los problemas existentes en la actualidad, persiguiendo la sostenibilidad económica, ambiental y social del planeta. Dentro de los objetivos que se enmarcan dentro de la protección del planeta se encuentra el ODS número 13 – Acción por el clima – en el cual que pretende que se adopten medidas urgentes para combatir el cambio climático y sus efectos. La UPV está directamente implicada en este desafío, y en este TFG se buscará contribuir en esta dirección ofreciendo una herramienta que permita monitorizar la evolución climática. [0] Además, la tecnología empleada será de bajo coste y de bajo impacto ambiental.

Otro aspecto que me ha llamado mucho la atención a nivel técnico ha sido la posibilidad de trabajar con protocolos diferentes, como REST y MQTT, para crear un sistema que permita a cualquier dispositivo que soporte el protocolo MQTT hacer peticiones de tipo REST (a través de un proxy) de forma transparente¹.

Por último, destacar la posibilidad de profundizar mis conocimientos sobre el lenguaje de programación Python, uno de los lenguajes más utilizado hoy en día.

¹ Se refiere a la habilidad de un protocolo de transmitir datos a través de la red de manera que sea transparente para aquellos que están usando el protocolo. [1]

1.2 Objetivos

El objetivo del trabajo es el de implementar un proxy MQTT, escrito en lenguaje Python, para la plataforma OpenWeather, utilizando la API ofrecida por dicha plataforma meteorológica para publicar información de interés a la cual se suscriben los clientes

Dicho proxy deberá ser ubicado y configurado en una Raspberry PI 3, un dispositivo de bajo consumo y bajo coste.

El proxy debe permitir una comunicación con los diferentes clientes a través del bróker MQTT Mosquitto.

Un cliente tiene que ser capaz de:

- I. Consultar nuevas informaciones meteorológicas/climáticas
- II. Suscribirse a un determinado tipo de información meteorológica que ya han consultado otros clientes
- III. Generar una petición de información meteorológica a la cual el proxy debe contestar con una determinada frecuencia, permitiendo así a otros clientes suscribirse a la misma tipología de información.

1.3 Metodología

Los pasos que se han seguido para la realización del proyecto han sido:

- I. Repaso del protocolo MQTT. En una asignatura de los cursos académicos anteriores ya había estudiado este protocolo, pero para llevar a cabo el trabajo tuve que profundizar en varios conceptos.
- II. Estudio y análisis de la API de OpenWeather. La API proporciona el acceso a muchas colecciones de datos; por lo tanto, analizando toda la documentación; elegí las que he considerado como más adecuadas para cumplir los objetivos del proyecto.
- III. Estudio del lenguaje de programación Python. Desarrollo de pequeños programas para el testeo de algunas funciones del proxy. Por ejemplo, un pequeño programa para hacer peticiones a través del API, e imprimir la respuesta por pantalla.
- IV. Una vez instalado el bróker MQTT en la Raspberry PI y configurado el acceso al router, varias pruebas de conexión con diferentes dispositivos.
- V. Ideación de posibles soluciones para hacer compatibles peticiones de tipo REST con un sistema basado en publicadores/suscriptores como el MQTT.
- VI. Desarrollo de varias versiones del proxy, cada vez más completas y complejas, hasta llegar a la versión final funcionante.

1.4 Estructura del documento

Después de esta breve introducción, se detallan los varios capítulos de que se compone este trabajo con una breve descripción de lo que se encuentra en cada uno de ellos.

- **Capítulo 2: Estado del arte y características de las tecnologías implementadas**
Se habla del contexto general y se analizan las características de las tecnologías, los protocolos y los lenguajes de programación implementados para llevar a cabo el proyecto.
- **Capítulo 3: Análisis del problema**
Se analizan en detalle la API de OpenWeather y el protocolo MQTT para encontrar la forma correcta de diseñar el proxy.
- **Capítulo 4: Diseño de la solución**
Se explica de manera detallada los varios pasos para llegar a la versión completa y funcional del proxy.
- **Capítulo 5: Validación y Pruebas**
Se enseñan mediante capturas varios ejemplos de funcionamiento del proxy, y se incluyen resultados del testeado de carga y esfuerzo para comprobar la eficiencia de la solución.
- **Capítulo 6: Conclusiones**
Se presentan las conclusiones, hablando también de las dificultades encontradas durante el desarrollo del proxy, así como posibles trabajos futuros.



2. Estado del arte y características de las tecnologías implementadas

2.1 Internet of Things

No existe una definición única del Internet de las Cosas (Internet of Things en inglés); una definición simple podría ser: un sistema de dispositivos computacionales interconectados. Estos dispositivos pueden ser de varios tipos (móviles, calentadores, sensores de temperatura, refrigeradores, sistema de iluminación...) y pueden comunicar entre si a través de protocolos de comunicación en red.

Otra definición interesante es la del Grupo de Soluciones Empresariales para Internet (IBSG) de Cisco:

“IoT es simplemente el momento en el que hay más “cosas u objetos” que personas conectadas a Internet.”

Según un estudio realizado por dicho grupo [2], en 2003 la población mundial era de 6300 millones, y había aproximadamente 500 millones de dispositivos conectados a la red. Así pues, la relación *dispositivo/persona* equivalía a 0.08.

En 2010, con la llegada de los smartphones y tabletas, el número de dispositivos conectados a Internet llegó a 12.500 millones mientras la población mundial a 6.800 millones. Eso quiere decir que la relación pasó de los 0.08 del 2003, a 1.84 en solo 7 años. En base a estos datos, y siguiendo la definición del IBSG, el IoT nació aproximadamente entre 2008 y 2009. A partir de ese momento, el crecimiento del número de dispositivos conectados en la IoT ha sido constante y, según las estimaciones de Statista [3], llegará a ser exponencial (ver Ilustración 2).

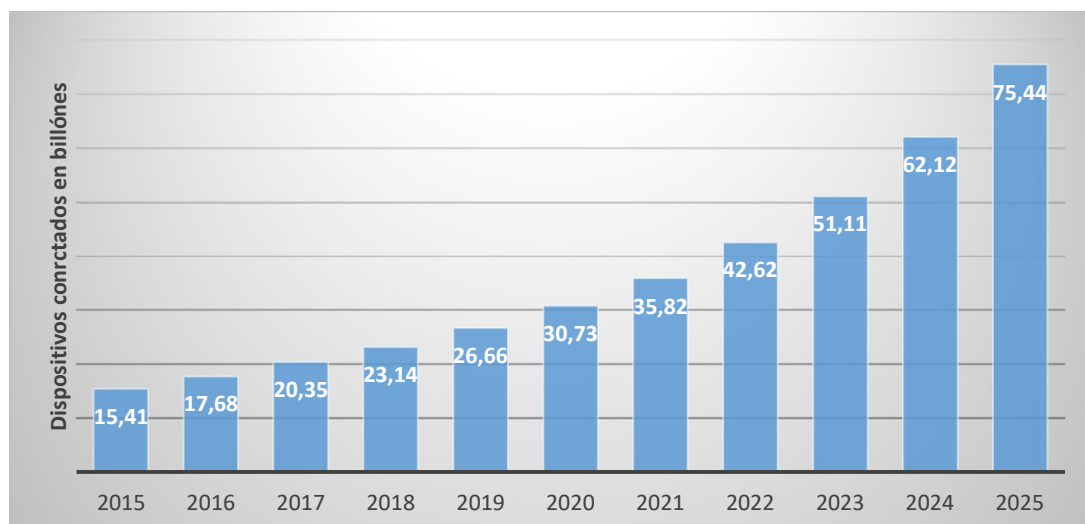


Ilustración 2 – Evolución del número de dispositivos conectados al IoT en el mundo. (Statista)

2.1.1 Peculiaridades de la IoT

El objetivo principal del Internet de las Cosas es crear un vínculo entre el mundo real y el mundo virtual. Siendo el Internet de las Cosas una red de objetos interconectados y capaces de comunicar entre sí y con el mundo externo, ha surgido la necesidad de organizar esta enorme estructura definiendo una arquitectura.

No existe un consenso único sobre qué arquitectura se aplica universalmente, pero cuando se trata el tema de IoT se suele hablar de dos tipologías de arquitecturas: de tres capas, y de cinco capas. [4]

La arquitectura de tres capas es la más básica (ver Ilustración 3) y, como se puede intuir fácilmente, es una arquitectura formada por tres capas: las capas de Percepción, Red y Aplicación.

- En la **capa de Percepción** se encuentran los sensores que tienen la función de detectar y recopilar información sobre el medio ambiente, o que interactúan directamente con los seres humanos. Se puede definir como la capa física.
- La **capa de Red** es la encargada de conectar todos los dispositivos con los demás dispositivos pertenecientes al IoT, o bien a dispositivos de Red (Router, Hub, Switch) o servidores.
- La **capa de Aplicación** define las varias aplicaciones en las que se puede implementar el IoT.

La arquitectura de tres capas es perfecta para describir conceptualmente la lógica de la tecnología IoT, pero, a la hora de realizar un servicio para un uso concreto, es conveniente considerar una arquitectura de cinco capas.

En la **arquitectura de cinco capas** siguen existiendo las capas de percepción y aplicación, las cuales mantienen el mismo papel en la arquitectura a tres capas. Las tres capas restantes son las capas de Transporte, Procesamiento y de Negocio (Ilustración 3).

- La **capa de Transporte** se encarga de transmitir información de la capa de Percepción (datos recopilados por sensores) a la capa de Procesamiento a través de redes inalámbricas (Bluetooth, NFC, 4G, 5G...)
- La **capa de Procesamiento** recoge todos los datos provenientes de la capa de Transporte y los almacena, analiza y procesa.
- La **capa de Negocio** es la última capa, y la de más alto nivel de abstracción.

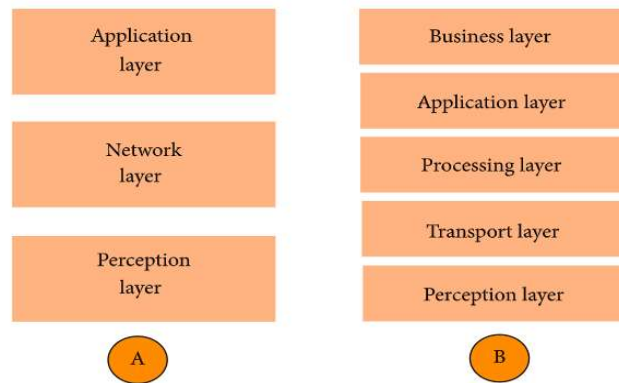


Ilustración 3 – Arquitectura de la IoT (A: tres capas) (B: cinco capas). (Hindawi)

2.1.2 Protocolos

Es fácil intuir como en un mundo de dispositivos interconectados lo ideal sería que todos pudiesen comunicar usando un mismo protocolo. Sin embargo, precisamente en virtud de la multitud de dispositivos diferentes y de tipologías de datos intercambiados, es algo imposible de implementar. Por lo tanto, coexisten varios protocolos, cada uno con su fortaleza y debilidades.

Hablando de protocolos para IoT, se puede hacer una primera categorización: protocolos cliente/servidor y protocolos basados en publicador/suscriptor [5].

En los **protocolos cliente/servidor** es necesario que un cliente se conecte a un servidor y realice solicitudes.

En los **protocolos publicador/suscriptor** los dispositivos se conectan a un “topic” de un gestor intermedio (bróker). Tanto los publicadores como los consumidores de esa información (suscriptores) tienen que conectarse al gestor intermedio y luego publicar o consumir informaciones de los “topics” de interés.

Entre los protocolos más utilizados en aplicaciones IoT tenemos:

- OPC UA
- HTTP (REST/JSON)
- MQTT
- CoAP
- DDS
- AMQP

En este trabajo vamos a utilizar dos de ellos: HTTP y MQTT.

2.1.2.1 HTTP

HTTP (*Hypertext Transfer Protocol*) es un protocolo sin estado² que permite la transferencia de información en la web. Es un protocolo que pertenece a la categoría cliente/servidor y es uno de los protocolos más utilizados hoy en día. Cuando hablamos de HTTP en IoT tenemos que centrarnos en la interfaz REST.

El término **REST** (*Representational State Transfer*) se refiere a una interfaz entre sistemas que utiliza HTTP para obtener datos o ejecutar operaciones sobre datos devolviéndolos en formatos específicos, como XML o JSON [6].

En la actualidad el formato más utilizado es el JSON ya que es más flexible, ligero y fácilmente legible en comparación con XML.

Como se ha dicho antes, una interfaz REST se apoya en HTTP, y por lo tanto usa los métodos básicos del protocolo de transferencia de hipertexto (GET, POST, PUT, DELETE...).

La clave de la interfaz REST es permitir crear una petición HTTP que, mediante las URI (*Uniform Resource Identifier*), manipula los objetos. Básicamente, el funcionamiento de una interfaz REST se puede definir de la siguiente manera:

- I. Un cliente que genera una petición (REQUEST) a un servidor mediante una URL, pasando en la misma todos los parámetros necesarios.
- II. El servidor devuelve los datos solicitados (RESPONSE) en el formato adecuado (suele ser JSON).

Un ejemplo de petición REST es la siguiente:

Imaginemos que queremos obtener los datos meteorológicos actuales para la ciudad de Valencia mediante la API de OpenWeather. Para hacerlo necesitamos componer la URL con todos los parámetros necesarios.

Analizando la documentación de la API [7], para esta tipología de “REQUEST” hay que incluir en la URL los parámetros “q”, donde hay que escribir el nombre de la ciudad, y “appid”, que sería la clave privada necesaria para usar la API (este parámetro es necesario en cualquier tipo de petición con la API de OpenWeather).

La URL básica para acceder a la colección de datos meteorológicos actuales es:

```
http://api.openweathermap.org/data/2.5/weather?
```

A esta URL base tenemos que añadir los parámetros necesarios para la petición:

² Un protocolo sin estado es un protocolo de comunicaciones que trata cada petición como una transacción independiente que no tiene relación con cualquier solicitud anterior. [8]

```
http://api.openweathermap.org/data/2.5/weather?q={cityname}&appid={API key}
```

Por lo tanto, la URL para el ejemplo propuesto sería:

```
http://api.openweathermap.org/data/2.5/weather?q=Valencia&appid=123456789
```

2.1.2.2 MQTT

El **MQTT** (*Message Queuing Telemetry Transport*) es un protocolo de comunicación M2M (machine-to-machine)³ del tipo *message queue*⁴ que pertenece a la familia de los protocolos publicador/suscriptor. Es considerado un estándar desde el 2014 según la OASIS⁵.

MQTT se puede considerar como uno de los pilares del Internet de las Cosas ya que, por sus características [11], es uno de los protocolos que más se adaptan a un uso en IoT.

Características:

- **Ligero y eficiente:** la comunicación que requiere recursos mínimos.
- **Comunicaciones bidireccionales:** permite intercambiar el papel de receptor y emisor de forma sencilla.
- **Escalable:** se adapta fácilmente al crecimiento del flujo de trabajo.
- **Seguro:** Facilita el cifrado mediante TLS y autenticación de clientes mediante protocolos como OAuth.

Este protocolo está basado en TCP/IP, y su funcionamiento es un servicio de mensajería con patrón publicador/suscriptor.

Cada cliente puede publicar o suscribirse a un determinado tipo de información que es organizado jerárquicamente por *topics*.

Para enlazar los diferentes clientes hay un “Broker”, que podemos definir como un gestor de los clientes. Todas las publicaciones y las suscripciones son gestionadas por dicho broker que recibe los mensajes envidados por los clientes y se encarga redistribuirlos entre sí.

Por lo tanto, un cliente puede publicar un mensaje en un determinado *topic*, y otros suscribirse al mismo *topic*. El Broker se encargará de que todos los clientes que se hayan suscrito a un *topic* reciban los mensajes correspondientes (ver Ilustración 4).

³ Se refiere al intercambio de información en formato de datos entre dos máquinas remotas. [9]

⁴ La cola de mensajes es una forma de comunicación asíncrona, los mensajes se guardan en la cola hasta que se procesan y se eliminan.

⁵ Consorcio internacional que se orienta al desarrollo, la convergencia y la adopción de los estándares de servicios web. [10]

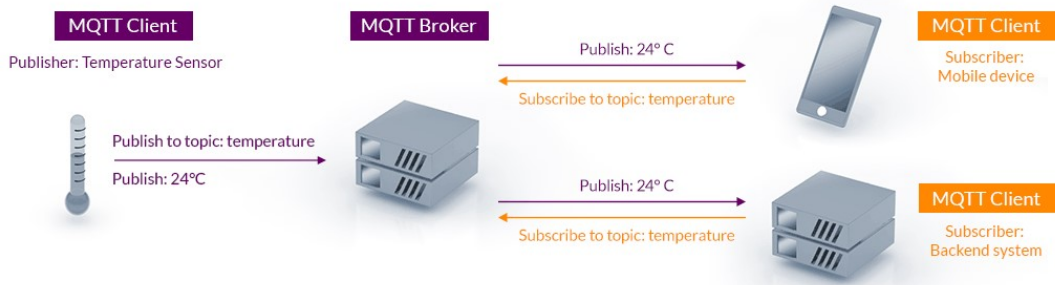


Ilustración 4 – Arquitectura de publicación / suscripción MQTT (mqtt.org).

MQTT dispone también de un **mecanismo de calidad del servicio** denominado **QoS** (*Quality of Service*). Existen tres niveles posibles:

- I. **QoS 0:** El mensaje se envía solo una vez. Si ocurre algún fallo es posible que algún mensaje no se entregue.
- II. **QoS 1:** El mensaje se envía hasta que la entrega está garantizada. Es posible que, en caso de fallo, el suscriptor pueda recibir uno o más mensajes duplicados.
- III. **QoS 2:** Se garantiza que cada mensaje llegue al suscriptor, y solo una vez.

Como se ha podido apreciar anteriormente el elemento clave del protocolo MQTT es el broker. Debido a la importancia y a que este protocolo se usa ampliamente, existe una grande cantidad de brokers MQTT, cada uno con sus peculiaridades y características.

El bróker más conocido y utilizado es el Eclipse Mosquitto, y es lo que se ha elegido para este proyecto.



Ilustración 5 - Logo broker MQTT Eclipse Mosquitto. (Mosquitto.org)

Mosquitto es un broker OpenSource desarrollado por la Eclipse Foundation. Es muy utilizado por su ligereza, lo que permite su uso en una gran variedad dispositivos, sobre todo si estos son de pocos recursos. En el apartado “motivación” del capítulo introductorio del trabajo se ha expreso la voluntad de trabajar con una tecnología de bajo recurso y costes, eligiendo una Raspberry Pi 3 como servidor donde ubicar el proxy y el broker. Por lo tanto, el broker Mosquitto es perfecto para nuestro propósito.

2.1.3 Raspberry Pi

Raspberry Pi es una serie de ordenadores de placa reducida⁶, de bajo coste, muy utilizados para crear soluciones IoT. Con más de 22 millones de unidades vendidas se sitúa como el tercer ordenador más vendido en el mundo. [12]

El primer modelo llegó en 2012 basado en dos pilares: código abierto y fines educativos. Debido a su bajo coste y versatilidad se ha convertido en poco tiempo en uno de los dispositivos más utilizados en los servicios IoT.

Desde el 2012 han visto la luz 4 modelos de Raspberry y varias versiones; en este trabajo se utiliza una **Raspberry PI 3 Model B v1.2** (ver Ilustración 6) con el sistema operativo Raspbian⁷, que es el SO recomendado para las Raspberry Pi. En la tabla 2 se ofrece un resumen de las especificaciones de este modelo.



Ilustración 6 – Placa Raspberry Pi 3

⁶ Computadora completa en un solo circuito.

⁷ Sistema operativo basado en una distribución de GNU/Linux llamada Debian.

Tabla 2 – Especificaciones placa Raspberry Pi 3 Model B v1.2

ESPECIFICACIONES	
PROCESADOR	Broadcom BCM2837, Cortex-A53 (ARMv8) 64-bit SoC
FRECUENCIA DE RELOJ	1,2 GHz
GPU	VideoCore IV 400 MHz
MEMORIA	1GB LPDDR2 SDRAM
CONECTIVIDAD INALAMBRICA	2.4GHz IEEE 802.11.b/g/n Bluetooth 4.1
CONECTIVIDAD DE RED	Fast Ethernet 10/100 Gbps
PUERTOS	GPIO 40 pines HDMI 4 x USB 2.0 CSI (cámara Raspberry Pi) DSI (pantalla tácil) Toma auriculares / vídeo compuesto Micro SD Micro USB (alimentación)

2.1.4 Python

Según una encuesta de desarrolladores realizada por la Eclipse Foundation en el abril de 2016, Java, C, JavaScript y Python eran las cuatro opciones principales para aquellos que estaban desarrollando soluciones para el Internet de las Cosas [13].

Para este proyecto se ha optado por utilizar Python, primero por sus características que, como veremos en seguida, se encajaban muy bien con las finalidades del trabajo y, segundo, por mi interés personal, ya que, como se ha explicado en el capítulo introductorio, me interesaba mucho profundizar en mis conocimientos sobre este lenguaje.

Python es un lenguaje de programación interpretado, multiparadigma, dinámico y multiplataforma. En los últimos años se ha convertido en uno de los leguajes de programación más usados en el mundo. La filosofía de Python está centrada en la legibilidad de su código.

Las principales ventajas de usar este lenguaje de programación son:

- **Facilidad de aprendizaje:** es un lenguaje con una sintaxis simple y muy fácil de entender.
- **Facilidad de uso:** tiene muchas bibliotecas preinstaladas, y la posibilidad de instalar nuevas para cubrir muchísimos usos y funcionalidades para cualquier tipología de proyecto.
- **Encaja bien con IoT:** debido a las características anteriores, y al ser multiplataforma, este lenguaje se puede integrar perfectamente en el Internet de las Cosas.



Ilustración 7 – Logo de Python (Wikipedia)

2.1.4.1 Librerías

Una de las fortalezas de Python, como se ha apreciado en el apartado anterior, son las numerosas bibliotecas instalables que le permiten enriquecer sus funcionalidades.

Para llevar a cabo el proyecto he disfrutado de varias librerías disponibles en la web, y de estas vamos a hablar en seguida. En este capítulo solo se proporcionarán informaciones genéricas sobre las librerías utilizadas para entender el contexto en el cual se encuentran. Los detalles sobre el uso concreto se encontrarán en el capítulo “Diseño de la solución”.

I. Paho-MQTT

Es la biblioteca que implementa MQTT en Python. La librería paho-mqtt está estructurada en tres módulos [14]:

- **Modulo Client:** gracias a este módulo se puede crear una instancia de cliente con la cual es posible conectarse al broker mediante la función `connect()`. Permite llamar una de las funciones `loop()` para mantener el flujo de tráfico con el broker. Además, gracias a las funciones `subscribe()` y `publish()`, se puede respectivamente suscribir a un topic y recibir mensajes, y comunicar mensajes para un determinado topic al broker. Al finalizar, usar la función `disconnect()` para desconectarse del broker.
- **Modulo Publish:** este módulo permite la publicación de mensajes sin la necesidad de crear una instancia de cliente. Se puede publicar un solo mensaje con la función `single()`, o varios con la función `multiple()`.

- **Modulo Suscribe:** permite la suscripción a los topics y la recepción de mensajes por parte del broker.

II. Requests

Es una librería HTTP con licencia de Apache2. Gracias a esta librería es posible efectuar peticiones HTTP de manera muy sencilla y directa.

III. JSON

JSON (*JavaScript Object Notation*) es un formato de texto para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos del lenguaje JavaScript y, debido su amplio uso como alternativa a XML, se considera un formato independiente.

La API OpenWeatherMap procesa los datos meteorológicos en formato JSON; por lo tanto, ha sido necesario importar la librería JSON en el diseño del proxy.

La librería JSON proporciona una compatibilidad completa con el formato de texto. Permite leer objetos JSON, decodificar, crear nuevos, extraer valores, y también convertir un diccionario⁸ Python en un objeto JSON.

IV. Threading

Como veremos más adelante, para el desarrollo del proxy será necesario considerar la posibilidad de generar varios procesos o subprocesos concurrentemente.

Para este propósito Python proporciona 2 opciones: *multiprocessing* o *multithreading*. Con el *multiprocessing* se logra la computación paralela, generando nuevos procesos, por lo que son necesarios varios núcleos. Con el *multithreading* se logra la computación paralela generando diferentes hilos de ejecución (subprocesos).

Considerando que ejecutar un proceso con varios hilos suele requerir menos recursos de memoria al equivalente en procesos separados y, además, analizando las características de la Raspberry Pi 3, he optado en utilizar el *multithreading*.

La librería que proporciona soporte a la programación multihilo en Python se llama **Threading**. Gracias a esta librería es posible crear objetos de tipo Thread y ejecutar subprocesos independientes.

⁸ Tipo de estructura de datos que permite almacenar un conjunto de pares clave-valor, siendo cada clave única en el mismo diccionario.



3. Análisis del problema

En este capítulo trataremos de analizar la API de OpenWeather y el protocolo MQTT con el fin de encontrar una solución para alcanzar los objetivos expresados en el capítulo introductorio del trabajo.

3.1 Identificación y análisis de soluciones posibles

3.1.1 API OpenWeather

En el capítulo introductorio de este trabajo ya se han proporcionado algunas informaciones sobre la API de OpenWeather. Ahora entraremos más en detalle, analizando todas las características de las diferentes colecciones de datos ofrecidas por la API, las informaciones que proporciona, y las respuestas a las diferentes peticiones.

El primer paso para poder utilizar la API es obtener un clave (key) y esto se puede hacer de manera muy sencilla en la página web de la empresa. Hay diferentes tipologías de claves: gratuitas, para estudiantes, y de pago. En mi caso, he optado por la clave para estudiantes que es una vía intermedia entre la gratuita y la de pago.

Lo que diferencia cada clave son las colecciones de datos a las cuales proporcionan el acceso. Por lo tanto, el segundo paso es analizar a cuáles colecciones de datos se puede acceder con la clave que se ha elegido. Esto es un tema que ya se ha tratado en el capítulo introductorio, donde se proporcionó un resumen (tabla 1) de las diferentes colecciones de datos que se integrarán en el proxy.

El último paso, que trataremos en las siguientes secciones, es analizar cada una de las colecciones de datos.

3.1.1.1 Current weather data

Esta colección de datos proporciona los datos meteorológicos actuales para cualquier lugar de la tierra. A la hora de hacer una petición se puede seleccionar una o más localidades en base a diferentes parámetros. [15]

I. Llamada a los datos meteorológicos actuales para una localidad.

En el capítulo 2, cuando se ha hablado de la interfaz REST se ha proporcionado un ejemplo de cómo hacer una consulta a través de una URL usando la API de OpenWeather.

Bien, el funcionamiento de la API es el mismo para todas las colecciones de datos, modificando la URL base y los parámetros según los datos que se quieren pedir.

En el caso de la colección de datos *Current weather data*, la **URL base** es la siguiente:

```
http://api.openweathermap.org/data/2.5/weather?
```

Como se ha podido apreciar en el ejemplo mencionado en el capítulo 2, la clave de la interfaz REST es la de proporcionar los diferentes parámetros en la URL con el fin de componer la que se llama “REQUEST”.

El parámetro fundamental (parámetro “q” en el ejemplo) de esta llamada es obviamente lo que sirve para seleccionar la localidad de la cual se requieren las informaciones meteorológicas. Esta búsqueda de localidad se puede hacer de varias maneras:

a) **Por nombre de ciudad**

El parámetro “q” puede ser el nombre de una ciudad. La API es ofrecida por diferentes idiomas [16], por lo tanto, el nombre de la ciudad se puede poner en cualquier idioma (si está soportado).

Por ejemplo, pasar como parámetro “London” es lo mismo que poner “Londres”.

Para evitar casos de homonimia es posible proporcionar, además del nombre de la ciudad, también el código del país al que pertenece, separados por coma.

Los códigos de los países son definidos según la ISO 3166 ⁹

Por lo tanto, así es como se compone la URL para hacer una petición por nombre de ciudad:

```
http://api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}
```

Y para evitar casos de homonimia:

```
http://api.openweathermap.org/data/2.5/weather?q={city name},{state code}&appid={API key}
```

b) **Por el ID de la ciudad**

OpenWeather dispone de una base de datos en la cual se pueden encontrar todas las ciudades cubiertas por el servicio meteorológico.

⁹ Estándar internacional para códigos de países y códigos para sus subdivisiones.



Implementación de un proxy MQTT para la plataforma OpenWeather

A cada ciudad es asociado un numero identificativo único (ID) y es posible buscar una ciudad pasando como parámetro su ID.

```
http://api.openweathermap.org/data/2.5/weather?id={city_id}&appid={API key}
```

El fichero con todas las ciudades es descargable (en formato JSON) en la web de OpenWeather.

c) **Por coordenadas geográficas**

Otra posibilidad es buscar una localidad por coordenadas geográficas. Esta es una opción muy útil para todos aquellos dispositivos que disponen de GPS.

Para esta tipología de búsqueda hay que pasar dos parámetros: latitud (lat) y longitud (lon).

```
http://api.openweathermap.org/data/2.5/weather?lat={latitud}&lon={longitud}&appid={API key}
```

d) **Por código postal**

La última opción de búsqueda es por código postal. En este caso el parámetro se llama “zip” y hay que proporcionar código postal y código del país separados por coma.

```
http://api.openweathermap.org/data/2.5/weather?zip={zip code},{country code}&appid={API key}
```

II. **Llamada a los datos meteorológicos actuales para varias localidades.**

La API de Openweather ofrece la posibilidad de obtener datos meteorológicos actuales para más ciudades en una única llamada.

a) **Ciudades dentro de una zona rectangular.**

Permite buscar varias ciudades dentro de un área rectangular. Hay que proporcionar 4 coordenadas geográficas para componer el rectángulo (lon-left, lat-bottom, lon-right, lat-top) y el número de ciudades que hay que incluir en la respuesta. En este caso el parámetro se llama “bbox”.

```
http://api.openweathermap.org/data/2.5/box/city?bbox={bbox}&appid={API key}
```

Un ejemplo de llamada es el siguiente:

```
http://api.openweathermap.org/data/2.5/box/city?bbox=12,32,15,37,10&appid=123456789
```

Este REQUEST nos devolverá los datos meteorológicos actuales para 10 ciudades incluidas dentro del rectángulo cuyas coordenadas son: 12, 32, 15 y 37.

b) **Ciudades dentro de una zona circular.**

Búsqueda de varias ciudades dentro de un área circular cuyo centro es el punto definido con las coordenadas geográficas.

Los parámetros que se tienen que especificar son latitud (lat) y longitud (lon) para definir el punto de búsqueda y el parámetro “cnt”, que es el número de ciudades que hay que buscar alrededor del punto geográfico definido.

```
http://api.openweathermap.org/data/2.5/find?lat={lat}&lon={lon}&cnt={cnt}&appid={API key}
```

Una vez definidas todas las tipologías de búsqueda, analicemos los parámetros opcionales que se pueden agregar para completar la solicitud. Todos estos parámetros se pueden añadir a cualquier tipología de solicitud de la API de OpenWeather.

- **mode:** este parámetro permite elegir el formato de respuesta. Los valores aceptados son: *xml* y *html*. El formato por defecto es JSON.
- **units:** permite elegir las unidades de medida. Los valores aceptados son: *standard* (por defecto), *metric* e *imperial*.
- **lang:** con este parámetro se puede elegir el idioma de la salida.

Ahora enfoquemos el análisis en la respuesta a una petición.

Como se ha podido apreciar anteriormente, la API proporciona tres formatos de salida; nos centraremos en la tipología de formato que se usa por defecto (JSON) ya que, como se ha podido apreciar en los capítulos anteriores, es el formato que se ha elegido utilizar e integrar en el proxy.



Implementación de un proxy MQTT para la plataforma OpenWeather

```
{
  "coord": {
    "lon": -122.08,
    "lat": 37.39
  },
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 282.55,
    "feels_like": 281.86,
    "temp_min": 280.37,
    "temp_max": 284.26,
    "pressure": 1023,
    "humidity": 100
  },
  "visibility": 16093,
  "wind": {
    "speed": 1.5,
    "deg": 350
  },
  "clouds": {
    "all": 1
  },
  "dt": 1560350645,
  "sys": {
    "type": 1,
    "id": 5122,
    "message": 0.0139,
    "country": "US",
    "sunrise": 1560343627,
    "sunset": 1560396563
  },
  "timezone": -25200,
  "id": 420006353,
  "name": "Mountain View",
  "cod": 200
}
```

Ilustración 8 – Ejemplo salida petición Current Weather API (Openweathermap.org).

Como se puede observar (ver ilustración 8) la salida está organizada en secciones, cada una con varios parámetros.

No toda la información presente en la respuesta es de interés en cuanto a la solicitud de informaciones meteorológicas, por lo que sería conveniente, a la hora de integrar esta colección de datos en el proxy, filtrar toda esta información y acotar el campo de parámetros a aquellos realmente interesantes.

Tabla 3 – Parámetros salida Current weather API que se incluyen en el proxy.

Parametro	Significado
temp	Temperatura
feels_like	Sensación térmica
temp_min	Temperatura mínima
temp_max	Temperatura máxima
pressure	Presión atmosférica (hPa)
humidity	Humedad (%)
wind_speed	Velocidad viento en m/s
wind_deg	Dirección viento en grados

3.1.1.2 Hourly forecast 4 days

Esta colección de datos proporciona pronósticos por hora durante 4 días, o sea, la respuesta será compuesta por 96 intervalos de tiempo (96 horas). [17]

La **URL base** es la siguiente:

```
http://pro.openweathermap.org/data/2.5/forecast/hourly?
```

Es obviamente necesario seleccionar la localidad para la cual queremos los pronósticos meteorológicos. Igual que para la colección de datos anterior, hay 4 tipologías de búsqueda (nombre de ciudad, ID de la ciudad, coordenadas geográficas y código postal) que se aplican de la misma manera. El único elemento que varía es la URL base.

Por ejemplo, en el caso de la búsqueda por nombre de ciudad, la URL sería:

```
http://pro.openweathermap.org/data/2.5/forecast/hourly?q={ciudad y name}&appid={API key}
```

En esta colección de datos no está disponible la búsqueda de más ciudades en una única llamada.

Los parámetros opcionales son los mismos que los de *Current weather data* y, también en este caso, la salida presentaba varios parámetros de poco interés; por lo tanto, se han seleccionado los considerados más importantes (tabla 4).



Tabla 4 – Parámetros salida Hourly Forecast 4 days API que se incluyen en el proxy.

Parametro	Significado
temp	Temperatura
feels_like	Sensación térmica
temp_min	Temperatura mínima
temp_max	Temperatura máxima
sea_level	Presión atmosférica sobre nivel del mar
grnd_level	Presión atmosférica sobre nivel del suelo
pressure	Presión atmosférica
humidity	Humedad
wind_speed	Velocidad viento
wind_deg	Dirección viento en grados
pop	Probabilidad de precipitaciones
clouds	Percentil cielo nublado

3.1.1.3 Daily forecast 16 days

Esta colección de datos ofrece el pronóstico diario hasta 16 días para cualquier lugar o ciudad. Tanto las tipologías de búsquedas que los parámetros opcionales son los mismos que los de las colecciones de datos anteriores. [18]

La **URL base** es:

<http://api.openweathermap.org/data/2.5/forecast/daily?>

El único aspecto diferente son los parámetros presentes en la salida (tabla 5).

Tabla 5 – Parámetros salida Daily Forecast 16 days API que se incluyen en el proxy

Parametro	Significado
temp_day	Temperatura durante el día
temp_night	Temperatura durante la noche
temp_min	Temperatura mínima
temp_max	Temperatura máxima
feels_like_day	Sensación térmica durante el día
feels_like_night	Sensación térmica durante la noche
pressure	Presión atmosférica
humidity	Humedad
wind_speed	Velocidad viento
wind_deg	Dirección viento en grados
clouds	Percentil cielo nublado
pop	Probabilidad de precipitaciones

3.1.1.4 Solar radiation API

Con este producto es posible consultar datos de radiación solar actuales y pronosticados con hasta 16 días de anticipación para cualquier coordenada del mundo [19].

A diferencia de las colecciones de datos anteriores, *Solar radiation API* permite seleccionar una localidad únicamente a través de las coordenadas geográficas (latitud y longitud).

La **URL base** para acceder a los datos de radiaciones solares actuales es:

```
http://api.openweathermap.org/data/2.5/solar\_radiation?
```

Y para acceder a los pronósticos:

```
http://api.openweathermap.org/data/2.5/solar\_radiation/forecast?
```

El pronóstico de radiaciones solares está disponible durante 16 días, los primeros 5 días en intervalos de una hora, y los últimos 11 en intervalos de 3 horas.

Los datos proporcionados son bajo dos modelos: *Clear Sky* (cielo limpio) y *Cloud Sky* (cielo nublado). La monitorización de las radiaciones solares está representada por tres índices de **Irradiancia**¹⁰ que proporcionan modelos tanto de cielo despejado como de cielo nublado (tabla 6).

Tabla 6 – Parámetros salida Solar Radiation API que se incluyen en el proxy

Parametro	Significado
ghi	Irradiancia horizontal global ¹¹ W/m ² – Cielo nublado
dni	Irradiancia normal directa ¹² W/m ² – Cielo nublado
dhi	Irradiancia horizontal difusa ¹³ W/m ² – Cielo nublado
ghi_cs	Irradiancia horizontal global W/m ² – Cielo limpio
dni_cs	Irradiancia normal directa W/m ² – Cielo limpio
dhi_cs	Irradiancia horizontal difusa W/m ² – Cielo limpio

¹⁰ La radiación que incide en un instante sobre una superficie determinada. [20]

¹¹ La suma de la radiación directa y difusa (sobre una misma superficie horizontal. Es el total de la radiación que llega a un determinado lugar. [20]

¹² Radiación que llega a un determinado lugar procedente del disco solar medida en la dirección del rayo incidente. [20]

¹³ Es la radiación procedente de toda la bóveda celeste excepto la procedente del disco solar. [20]



3.1.1.5 Air Pollution API

Air Pollution API proporciona datos de contaminación del aire actuales, pronosticado e históricos para cualquier coordenada del mundo. [21]

El pronóstico de contaminación del aire cubre hasta cinco días. Los datos históricos están disponibles a partir de 27 de noviembre 2020.

La búsqueda de localidades, como en el caso de la *Solar Radiation API*, está disponible solo mediante coordenadas geográficas.

La **URL base** para consultar datos actuales de contaminación del aire es:

```
http://api.openweathermap.org/data/2.5/air_pollution?
```

Para consultar los pronósticos:

```
http://api.openweathermap.org/data/2.5/air_pollution/forecast?
```

Y para consultar los datos históricos:

```
http://api.openweathermap.org/data/2.5/air_pollution/history?
```

Para las consultas sobre datos histórico es necesario agregar a la consulta, además de latitud y longitud, otros dos parámetros: una fecha de inicio (*start*) y de finalización (*end*) para delimitar el intervalo de tiempo.

```
http://api.openweathermap.org/data/2.5/air_pollution/history?lat={lat}&lon={lon}&start={start}&end={end}&appid={API key}
```

La información que devuelve la API incluye índices de calidad del aire, datos sobre gases contaminantes y partículas (tabla 7).

Tabla 7 – Parámetros salida Air Pollution API que se incluyen en el proxy

Parámetro	Significado
aqi	Índice de calidad aire (escala de 1 a 5)
co	Concentración de CO (Monóxido de Carbono) $\mu\text{g}/\text{m}^3$
no	Concentración de NO (Monóxido de Nitrógeno) $\mu\text{g}/\text{m}^3$
no2	Concentración de NO ₂ (Dióxido de Nitrógeno) $\mu\text{g}/\text{m}^3$
o3	Concentración de O ₃ (Ozono) $\mu\text{g}/\text{m}^3$
so2	Concentración de SO ₂ (Dióxido de azufre) $\mu\text{g}/\text{m}^3$
pm2_5	Concentración de PM _{2.5} $\mu\text{g}/\text{m}^3$
pm10	Concentración de PM ₁₀ $\mu\text{g}/\text{m}^3$
nh3	Concentración de NH ₃ (Amoníaco) $\mu\text{g}/\text{m}^3$

3.1.2 Adaptación del API al protocolo MQTT

En el capítulo 2, hablando del protocolo MQTT, se ha visto como el broker (el gestor de clientes) organiza toda la información que circula por el sistema jerárquicamente por **topics**.

En la *ilustración 4* se ha podido apreciar un pequeño esquema de funcionamiento de un sistema MQTT, con un sensor que detecta la temperatura y publica el dato, comunicándolo al broker, usando como topic *Temperatura*.

Ahora el objetivo es adaptar las tipologías de informaciones que proporciona la API de OpenWeather a un sistema MQTT; por lo tanto, el primer aspecto en el cual tenemos que centrarnos es el concepto de **topic**.

3.1.2.1 Topic

En esta sección vamos a profundizar en el concepto de topic en MQTT, hablando de su estructura y de sus características con el fin de encontrar una solución para adaptar el uso del protocolo a las comunicaciones con la interfaz REST de la API de OpenWeather.

3.1.2.1.1 Estructura

En el ejemplo citado anteriormente (sensor de Temperatura) el **topic** era simplemente una palabra.

En un sistema tan complejo como la IoT, organizar jerárquicamente la información en un topic de una sola palabra es bastante complejo. De hecho, el funcionamiento de los topics es más amplio.

Un **topic** se define mediante una cadena de texto en formato UTF-8¹⁴ y consta de uno o más niveles separados por barras (“/”).

Un ejemplo de topic en un nivel es lo del sensor de temperatura; los siguientes son ejemplos de topics de más niveles:

jardin/huerto/humedad

hogar/cocina/temperatura

¹⁴ Es un formato de codificación de caracteres Unicode que utiliza símbolos de longitud variable.



Por lo tanto, el primer problema al cual me he enfrentado ha sido adaptar la información que se puede pedir a través del API a la estructura jerárquica de los topics MQTT.

Como se ha visto en el apartado donde se ha analizado en detalle la API de OpenWeather, hay cinco colecciones de datos, cada una con una o más tipologías de búsquedas y parámetros de entrada y de salida.

Siguiendo un orden lógico, una estructura para los topics puede ser la siguiente:

- **Primer nivel:** Colección de datos (*Current weather, Air Pollution...*).
- **Segundo nivel:** Sub colección (si existe). Por ejemplo: *Current, Forecast, Historical, Hourly o Daily*.
- **Tercer nivel:** Tipología de búsqueda (*nombre de ciudad, coordenadas geográficas, código postal...*)
- **Cuarto nivel:** Datos de entrada según la tipología de búsqueda.
- **Quinto nivel:** Parámetros de salida. Se puede poner uno o más parámetros separados por coma. Si se quieren todos los parámetros hay que usar la palabra “all”

Siguiendo la estructura que se acaba de detallar un ejemplo de topic puede ser el siguiente:

weather/city_name/Valencia/temp,humidity

Un cliente que se suscribe a este topic recibe la temperatura y la humedad actual para la ciudad de Valencia.

Como se puede observar con esta propuesta de diseño de topic, estamos también ampliando las funcionalidades de la API. De hecho, hemos visto en el apartado de análisis de la API que en las respuestas a las peticiones están incluidos todos los parámetros existentes para una determinada consulta. Pues con esta propuesta se permite al cliente elegir los parámetros que le interesan, recibiendo así solo la información que considera relevante.

Obviamente el proxy se encargará de generar la respuesta, extrapolando del RESPONSE del API la información pedida por el cliente. Los detalles de esta operación se expondrán en el capítulo “*Diseño de la solución*”.

3.1.2.1.2 Comodines

Otro aspecto muy interesante de los topic MQTT son los que se llaman **comodines**.

Los **comodines** (*wildcards* en inglés) permiten a un cliente suscribirse a uno o más temas. La suscripción se puede realizar de dos maneras: la primera especificando el nombre exacto del *topic* al que desea suscribirse, y la segunda usando comodines, o sea, caracteres especiales que permiten suscribirse a un conjunto de *topics*.

El primer comodín es el carácter “+”. Este carácter especial permite suscribirse a un único nivel de *topics*.

Por ejemplo:

Imaginamos que el proxy está publicando de forma periódica en los siguientes *topics*:

- *weather/City_name/Valencia/temp*
- *weather/City_name/Valencia/pressure*
- *weather/City_name/Valencia/humidity*

Si un cliente quiere suscribirse a todo el tráfico de datos meteorológicos actuales para la ciudad de Valencia, simplemente tiene que suscribirse al siguiente *topic*:

- *weather/City_name/Valencia/+*

De esta manera el nuevo cliente se ha suscrito a los tres *topics* anteriores.

El segundo y último comodín es el carácter “#”. Es un comodín *multinivel* y tiene que ocupar siempre la posición final del *topic*.

Para explicar su funcionamiento proporcionamos otro ejemplo:

En este caso el proxy está publicando de forma periódica en los siguientes *topics*:

- *forecast/hourly/city_name/Valencia/all*
- *forecast/daily/geo_coord/31,52/humidity,pressure*
- *forecast/hourly/ZIP_code/89015,110/temp,pop,clouds*

Si un nuevo cliente se suscribe al *topic* “*forecast/#*” recibe todo el tráfico generado por los tres *topics* anteriores.

3.1.2.2 Publicadores y suscriptores

Una de las características principales del protocolo MQTT, como se ha podido apreciar en el capítulo 2, es la **Comunicación bidireccional**, la cual permite intercambiar el papel de receptor y emisor de forma sencilla. Esta característica es fundamental para el



desarrollo de la arquitectura del sistema; de hecho, proxy y clientes deben intercambiar el rol de publicador y suscriptor para poder realizar una comunicación efectiva.

El proxy tiene que comunicar con:

- El broker Mosquitto.
- Los servidores de OpenWeather a través del API.
- Los diferentes clientes que se conectan al sistema.

En realidad, las comunicaciones con los clientes se realizan a través del bróker. Por lo tanto, desde el punto de vista técnico, el proxy solo hace una comunicación efectiva con broker y servidores de OpenWeather. Un cliente solo comunica con el proxy a través del broker.

En el capítulo introductorio, en la sección de Objetivos, se han definido cuales son las acciones que tiene que poder cumplir un cliente:

- I. Consultar nuevas informaciones meteorológicas/climáticas.
- II. Suscribirse a un determinado tipo de información meteorológica que ya han consultado otros clientes.
- III. Generar una petición de información meteorológica a la cual el proxy debe contestar con una determinada frecuencia, permitiendo así a otros clientes suscribirse a la misma tipología de información.

Por lo tanto, el problema que hay que resolver es como hacer que los clientes puedan consultar nuevas informaciones meteorológicas comunicando al proxy que datos les interesan.

El proxy tiene que estar pendiente de recibir nuevas peticiones; entonces la solución puede ser crear un topic, que llamaremos *request*, en el cual el proxy tiene el papel de consumidor, o sea recibe las peticiones, y los clientes el papel de publicadores (publican peticiones).

Una vez recibida la petición hay que intercambiar el papel entre las dos partes: por un lado el cliente se convertirá en un consumidor, esperando la respuesta a la petición, enviada al topic correspondiente; por otro lado, el proxy se convertirá en publicador, contestando a la petición del cliente, obviamente después de haber consultado la API de OpenWeather.

En un contexto en el cual se pueden conectar muchos clientes para pedir informaciones meteorológicas, sería conveniente que el proxy pueda reconocer y distinguir cada uno de ellos. Por lo tanto, cada cliente debe tener asignado un código identificativo.

Para limitar el número de comunicaciones entre proxy y clientes la solución ideal sería que el mismo cliente genere un código identificativo aleatorio, y se lo comunique al proxy junto a la petición de informaciones meteorológicas.

Los mensajes para intercambiar estas informaciones serán todos en formato JSON.

Otro aspecto importante a considerar es el de contemplar la posibilidad que un cliente suscrito a un topic con una respuesta periódica pueda, en algún momento, perder la conexión. En esta situación el proxy, sin tener conocimiento de la pérdida de conexión, continuaría contestando a la petición periódica. Para solucionar el problema el protocolo MQTT proporciona una función perfecta: **Last Will and Testament**.

3.1.2.2.1 Last Will and Testament

No hay que olvidar que este proyecto está orientado al uso de dispositivos de bajo consumo y bajo coste. Por lo tanto, es razonable suponer que en algún momento algunos clientes puedan perder la conexión.

El **Last Will and Testament** (Ultima voluntad y testamento) es una función perfecta para notificar, en una comunicación MQTT, la pérdida de conexión a los demás clientes.

El funcionamiento es bastante sencillo. Cuando un cliente se conecta al broker, junto al mensaje de conexión envía otra comunicación denominada *last will message*. El *last will message* es un mensaje MQTT clásico; por lo tanto, hay que elegir un topic y un contenido a enviar. El broker envía pings periódicos a los clientes que se han conectado. Si en algún momento un cliente acaba de contestar a los pings del broker, y si dicho cliente en el momento de la conexión comunicó su *last will message*, el broker se encarga de publicarlo. En nuestro caso el topic para los *last will message* puede ser *offline*, y el contenido del mensaje el ID del cliente. De esta manera el proxy, suscribiéndose al topic *offline*, irá recibiendo los códigos identificativos de los clientes a los cuales se le ha caído la conexión.

3.2 Solución propuesta

Ahora que se han considerados todos los aspectos relevantes, y que se han propuesto varias soluciones a los diferentes problemas encontrados, propondremos, en este apartado, una solución completa y detallada, trazando también las pautas para la fase de diseño.

3.2.1 Arquitectura del sistema

Como se ha detallado en los capítulos anteriores el sistema está compuesto por los siguientes elementos:



Implementación de un proxy MQTT para la plataforma OpenWeather

- I. **Proxy:** ubicado en la Raspberry Pi y programado en Python. Es el elemento central del trabajo, y es el elemento encargado de gestionar las peticiones de los clientes, comunicando a través del broker.
- II. **Broker:** el broker Mosquitto que, al igual que el proxy, está ubicado en la Raspberry Pi. Es el elemento encargado de filtrar y redistribuir todas las comunicaciones entre proxy y clientes.
- III. **Clientes:** cualquier dispositivo que soporta el protocolo MQTT y que se conecta al broker para realizar peticiones meteorológicas, o suscribirse a un determinado topic de información.
- IV. **Servidor de OpenWeather:** el elemento que contesta a las peticiones del proxy. Ofrece el servicio mediante la API.
- V. **Estaciones meteorológicas:** elementos ubicados en todo el planeta cuya tarea es la recopilar informaciones meteorológicas y comunicarlas al servidor central de OpenWeather.

En la *Ilustración 9* se puede observar la arquitectura del sistema que acabamos de explicar.

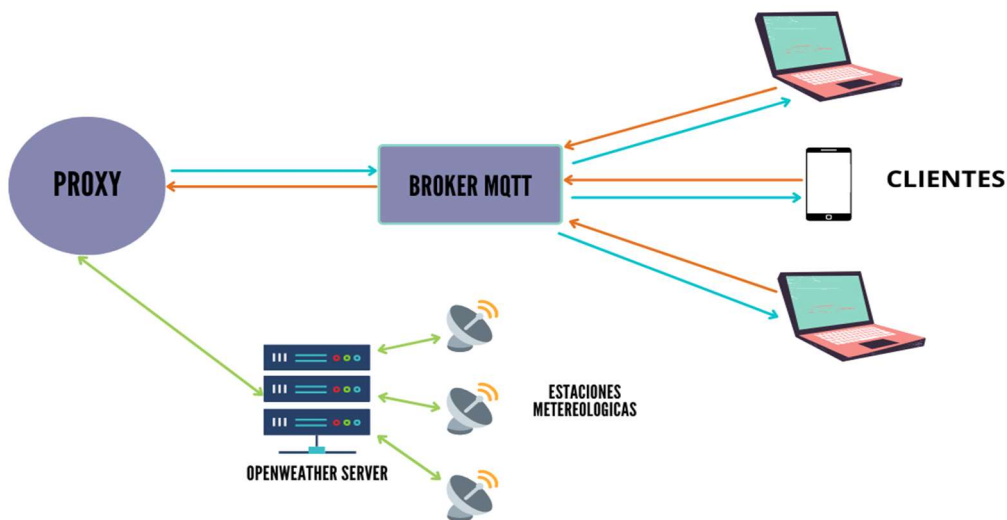


Ilustración 9 – Arquitectura del sistema.

3.2.2 Esquema de composición de un topic

En este apartado se muestra cual es el resultado final de la organización y de la estructura de los topics en el sistema del proyecto.

Toda la información disponible en las cinco colecciones de datos antes analizadas ha sido esquematizada con el fin de poder realizar consultas a través de un topic de manera sencilla y, además, incluyendo en el mismo topic toda la información necesaria para que el proxy pueda hacer las consultas a través la API de OpenWeather.

Un resumen de todas las posibles “combinaciones” para la composición del topic se puede consultar en la *Ilustración 10*.

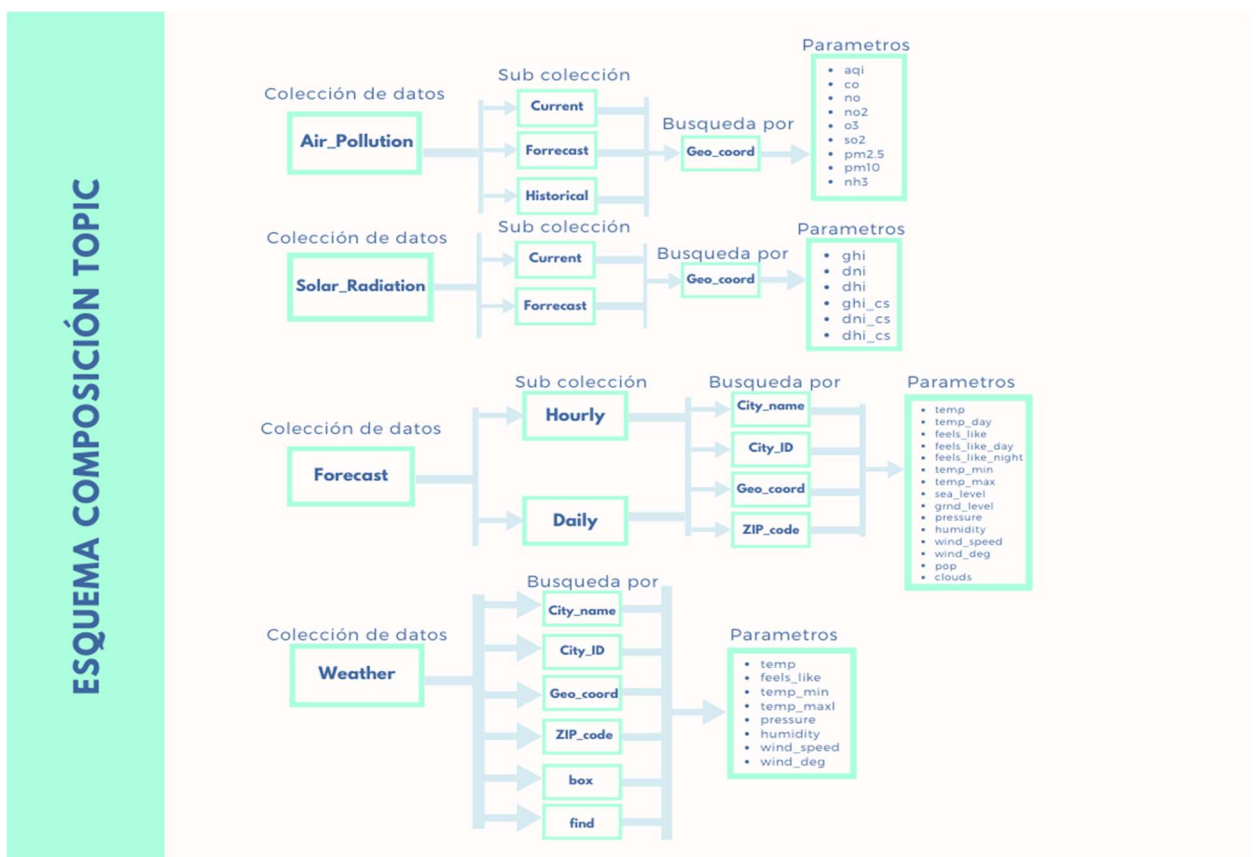


Ilustración 10 – Esquema de composición de un topic.

3.2.3 Proxy

El proxy se puede considerar un cliente MQTT que realiza diferentes tareas. No solo tiene que atender a las peticiones de los varios clientes a través del API, sino también gestionar dichos clientes, y organizarlos en base a los topics a los cuales se han suscrito.

3.2.3.1 Recepción y respuestas a las peticiones

Respecto a la recepción de las peticiones, el proxy, una vez se haya conectado al bróker, tiene que suscribirse a un topic denominado *request*, en el cual va recibiendo las diferentes peticiones.

La petición será un objeto JSON con tres parámetros:

- I. **Client_id:** código identificativo aleatorio generado por el cliente.
- II. **Topic:** las informaciones meteorológicas se piden siguiendo la estructura expuesta en el apartado anterior, y consultable en la *ilustración 10*. El cliente, una vez que haya enviado la petición, se suscribe al topic seleccionado; por lo tanto, el proxy tiene que publicar la respuesta a la petición en el topic correspondiente.
- III. **Frecuencia:** un cliente puede hacer dos tipos de peticiones: con o sin frecuencia. En las peticiones con frecuencia el cliente tiene que asignar al parámetro frecuencia un valor numérico. Este parámetro representará la frecuencia de respuesta (en segundos) al topic correspondiente. Si el cliente quiere hacer una petición “simple”, o sea una consulta sin respuesta en intervalos de tiempo, tiene que asignar al parámetro el valor cero.

Si la petición es con frecuencia, el proxy inicializa un *thread* que se encargará de realizar las publicaciones periódicas. Si la petición es sin frecuencia se encargará de contestar el hilo principal.

Para generar la respuesta es obviamente necesario consultar la API de OpenWeather. Por lo tanto, el paso a seguir será el de extrapolar la información necesaria del topic. Los detalles de esta operación se podrán apreciar en el siguiente capítulo.

Una vez realizada la petición a la API de OpenWeather, y generada la respuesta extrapolar los datos correspondientes, el broker contesta publicando un objeto JSON.

3.2.3.2 Gestión de topic y clientes

Trabajando con dispositivos de bajos recursos es imprescindible monitorizar el tráfico y evitar, cuando sea posible, cualquier tarea inútil o redundante.

Como se ha visto en el apartado anterior existen peticiones con o sin frecuencia. En las peticiones con frecuencia se genera un tráfico constante en los intervalos de tiempo elegidos por los clientes. Es muy importante considerar la posibilidad de que puedan llegar nuevos clientes que se suscriben a uno o más topics (usando comodines) de información ya existente, y, por lo tanto, no es necesario generar nuevo tráfico.

Para cumplir este propósito se utiliza una estructura de datos de Python que ya se había mencionado anteriormente: el *diccionario*.

Un diccionario es un tipo de dato que permite almacenar cualquier tipo de valor (cadenas, listas, enteros...).

Cada elemento se identifica con una *key* (clave) y un *value* (valor).

```
diccionario = {'Ciudad' : 'Valencia', 'temperatura' : 32 }
```

Para la gestión de clientes el elemento central es el topic (o topics) al cual se suscriben; por lo tanto, las claves del diccionario serán todos los topics activos. Los valores serán listas de pares ID cliente – frecuencia.

Cuando un cliente publica una petición con topic *request*, como sabemos, el proxy recibe dicha petición, que contiene ID cliente, topic y frecuencia. El proxy consulta el diccionario para verificar si el topic ya existe.

Si el topic no existe, añade la nueva clave (el topic) y establece como valor un listado que contiene el par ID cliente – frecuencia.

Si el topic ya existe simplemente añade el par ID cliente – frecuencia que acaba de recibir a la lista ya presente en el diccionario en el topic correspondiente.

Obviamente no todos los clientes pueden pedir la misma información con la misma frecuencia. Y es precisamente por eso que en el diccionario guardamos también la frecuencia. De esta manera elegimos la frecuencia más baja, que será la que se utilizará para publicar la respuesta.

Cuando un cliente se desconecta o cae la conexión, sabemos que el proxy recibe el *last will message* que contiene el ID del cliente. Una vez recibido un ID cliente en topic *offline*, se eliminan todos los pares que incluyen su ID de todos los listados de cada topic.



Si el cliente que acaba de desconectarse se había suscrito a un determinado topic con la frecuencia más baja de entre todas las peticiones, la frecuencia de publicación se actualiza con la nueva frecuencia más baja (si hay otros clientes activos).

Si el cliente que acaba de desconectarse era el único suscrito al topic de información meteorológica, dicho topic se elimina del diccionario, y el thread encargado de hacer las publicaciones periódicas acaba su ejecución.

3.2.4 Cliente

Para comprobar el correcto funcionamiento del proxy será necesario desarrollar también un cliente, programado en Python y ubicado en un ordenador portátil.

El cliente deberá ser capaz de conectarse al broker Mosquitto, publicar las peticiones, y suscribirse a los topics de información meteorológica.

Además, si el cliente se suscribe a más de un topic usando los *comodines*, a medida que recibe mensajes, debe comunicar al broker el topic de los mensajes que está recibiendo y su código identificativo. Esta comunicación se efectúa usando el topic “*sub*”, en el cual el proxy tendrá obviamente el papel de consumidor.

3.3 Plan de trabajo

Ahora que se ha definido una solución completa para el proyecto, definimos un plan de trabajo para el diseño práctico de dicha solución, cuyos detalles se podrán apreciar en el siguiente capítulo.

- I. Instalación y configuración de todos los componentes.
- II. Programación del proxy.
- III. Programación de un cliente para el testeo del proxy.
- IV. Testeo del proxy bajo varias condiciones.



4. Diseño de la solución

En este capítulo se proporcionan todos los detalles de la fase de diseño de la solución propuesta en el capítulo anterior. Para ello, primero nos centraremos en la fase de instalación y configuración de los varios componentes necesarios para el funcionamiento del sistema. A continuación se expondrán todos los pasos para la creación del proxy, analizando el código de todos los módulos y de las funciones desarrolladas. Por último, se detallará la programación del cliente MQTT en Python, un elemento necesario para comprobar el correcto funcionamiento del proxy.

4.1 Instalación y configuración de componentes

El primer paso ha sido conectar la Raspberry PI al enrutador de mi red doméstica. Este paso es necesario no solo para descargar e instalar todos los softwares y para que el proxy pueda comunicar con la API de Openweather, sino también para proporcionar el servicio a los clientes conectados en mi red doméstica. De hecho, como veremos más adelante, el broker tendrá asignado una dirección IP, que sería la dirección asignada por el enrutador a la Raspberry Pi cuando se realizó la conexión.

En las siguientes secciones se proporcionarán todos los detalles de la instalación de los softwares necesarios para la realización del proyecto.

4.1.1 Mosquitto

La descarga e instalación del broker Mosquitto se puede llevar a cabo de manera muy sencilla con la *Advanced Packing Tool*¹⁵ (Herramienta avanzada de Empaquetado, conocida por el acrónimo APT).

- I. Abierto el terminal, tenemos que actualizar la lista de los paquetes para asegurarnos de descargar la última versión disponible del software:

```
pi@raspberrypi:~ $ sudo apt update
```

- II. Instalar el broker mosquitto:

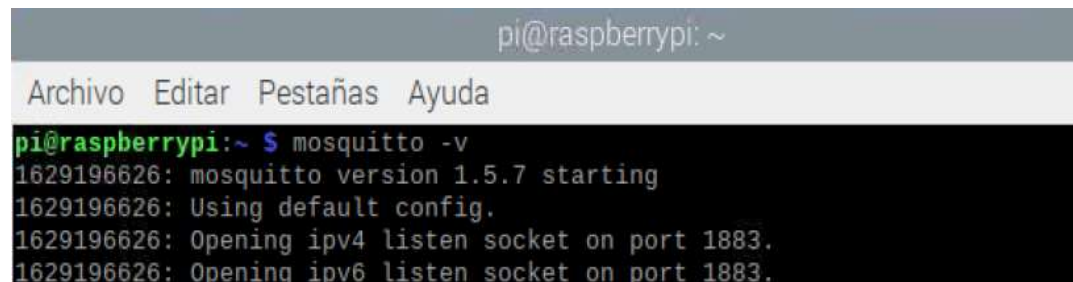
```
pi@raspberrypi:~ $ sudo apt install mosquitto
```

¹⁵ Es el programa de gestión de paquetes en los sistemas GNU/Linux.

- III. Para hacer que el broker se inicie automáticamente al arrancar la Raspberry Pi:

```
pi@raspberrypi:~ $ sudo systemctl enable mosquitto.service
```

- IV. Para comprobar la versión instalada, si el servicio se ha inicializado, el puerto en los cuales el broker escucha nuevas comunicaciones, y todos los mensajes de control en la consola:



```
pi@raspberrypi:~ $ mosquitto -v
1629196626: mosquitto version 1.5.7 starting
1629196626: Using default config.
1629196626: Opening ipv4 listen socket on port 1883.
1629196626: Opening ipv6 listen socket on port 1883.
```

Ilustración 11 – Captura salida comando mosquitto-v.

- V. Comprobar la dirección IP de la Raspberry con el siguiente comando:

```
pi@raspberrypi:~ $ hostname -I
```

La dirección IP en nuestro caso es: **192.168.1.264**.

La dirección IP y el puerto (1883) son dos parámetros necesarios para que el proxy y todos los clientes puedan conectarse al broker.

- VI. El último paso será completar el archivo de configuración de Mosquitto: *mosquitto.conf*.

El broker tiene diferentes parámetros modificables y una configuración por defecto; analizado el manual [22] se ha elegido la configuración adecuada para el proyecto añadiendo las siguientes líneas al fichero de configuración:

```
- pid_file /var/run/mosquitto.pid
```

De esta manera se indica la ruta del archivo que contendrá el número de identificación de proceso (PID) del broker Mosquitto.

```
- persistence true
persistence_location /var/lib/mosquitto/
```

Con *persistence* activada (*true*) los datos de conexión, suscripción y publicación se escribirán en el disco en la dirección indicada en *persistence location*.

```
- log_dest file /var/log/mosquitto/mosquitto.log
```

Indicamos la ubicación del fichero de log.

```
- listener 1883 0.0.0.0
```

Se vincula el puerto 1883 a cualquier dirección.

```
- allow_anonymous true
```

Se permiten conexiones anónimas, o sea, los clientes no tienen que proporcionar un nombre de usuario. Siendo el sistema ubicado en un entorno doméstico, se ha considerado inútil permitir el acceso solo a clientes autenticados.

4.1.2 Python y librerías

El sistema operativo de la Raspberry Pi tiene instalada por defecto una versión de Python.

En el caso de la Raspberry Pi 3 la versión de Python preinstalada es demasiado obsoleta; por lo tanto, se ha optado por instalar la versión 3.7, que ofrece compatibilidad con todas las librerías que necesitamos para el desarrollo del proxy y del client.

Antes de instalar y configurar Python 3.7 tenemos que instalar las dependencias necesarias. Por ello se utilizará nuevamente la herramienta APT:

```
pi@raspberrypi:~ $ sudo apt install -y build-essential tk-dev  
libncurses5-dev libncursesw5-dev libreadline6-dev libdb5.3-dev  
libgdbm-dev libsqlite3-dev libssl-dev libbz2-dev libexpat1-dev  
liblzma-dev zlib1g-dev libffi-dev
```

Seguidamente descargamos el paquete que contiene el software desde la página web de Python:

```
pi@raspberrypi:~ $ wget https://www.python.org/ftp/python/  
3.7.0/Python-3.7.0.tar.xz
```

Luego descomprimos el archivo que se acaba de descargar:

```
pi@raspberrypi:~ $ tar xf Python-3.7.0.tar.xz
```

Y nos ubicamos en la carpeta que contiene los archivos extraídos:


```
pi@raspberrypi:~ $ cd Python-3.7.0
```

Ejecutar el comando de configuración:

```
pi@raspberrypi:~ $ ./configure --prefix=/usr/local/opt/Python-3.7.0
```

Por último, instalar el programa:

```
pi@raspberrypi:~ $ sudo make altinstall
```

En cuanto a la instalación de los diferentes módulos y librerías necesarios para el correcto desarrollo del proxy y del cliente, utilizaremos el gestor de paquetes del *Python Package Index*¹⁶ (*PyPI*), llamado PIP.

Por lo tanto, el primer paso es instalar la herramienta mediante el *Advanced Packing Tool* (APT).

```
pi@raspberrypi:~ $ sudo apt-get python3-pip
```

El segundo y último paso, instalar todos los módulos y librerías mencionados el *apartado 2.1.4.1* del trabajo:

- Paho-MQTT

```
pi@raspberrypi:~ $ pip install paho-mqtt
```

- Requests

```
pi@raspberrypi:~ $ pip install requests
```

- Json

La librería Json está ya incluida en los paquetes “básicos” de Python, por lo tanto, no es necesaria ninguna instalación.

- Threading

Al igual que la librería Json, no es necesaria ninguna instalación.

¹⁶ Es el repositorio de software oficial para aplicaciones de terceros en el lenguaje de programación Python. [23]



4.2 Programación del proxy

En este apartado se expondrá de manera detallada el código del proxy, en todos sus módulos y componentes.

Antes de iniciar la programación de dichos módulos y funciones, es necesario importar todas las librerías mencionadas en los capítulos anteriores (ver ilustración 12).

```

1 import datetime as dt
2 import threading
3 import time
4 import json
5 import paho.mqtt.client as mqtt
6 import requests

```

Ilustración 12 – Importación de librerías.

Definida esta premisa, empezamos con el análisis de todos los módulos del proxy.

4.2.1 Main

Esta es la parte donde se inicia el programa; por lo tanto, se inicializan algunas variables, y sobre todo el *client MQTT* (ver ilustración 13)

```

550
551 api_key = "[REDACTED]"
552 lang = 'es'
553 topics = {}
554
555 # Main
556 ▶ if __name__ == '__main__':
557     client = mqtt.Client() # Inicializamos el cliente MQTT
558     client.on_connect = on_connect # Función de callback cuando ocurre la conexión.
559     client.on_message = on_message # Función de callback cuando llega un mensaje.
560
561     client.connect("192.168.1.246", 1883, 60) # Conexión al broker Mosquitto
562
563     client.loop_forever()
564

```

Ilustración 13 – Modulo main del proxy.

El parámetro *api_key* es la clave de acceso a todas las colecciones de datos de OpenWeather que, como se ha dicho en el capítulo 3, debe estar incluida en cualquier petición. Es una cadena alfanumérica de 32 caracteres.

El parámetro *lang* como el parámetro *api_key* estará incluido en todas las peticiones a la API de OpenWeather. Se inicializa con la cadena ‘**es**’, que es la sigla del idioma ‘español’.

La variable *topics* representa el diccionario que incluirá todos los *topics* activos (*keys*) y las listas de clientes que se han suscrito a cada *topic* con una determinada frecuencia (*values*).

Una vez inicializados parámetros y variables empieza el módulo *main*, en el cual la primera tarea que se cumple es inicializar el cliente MQTT, mediante la función **Client()** de la librería *Paho MQTT*. Seguidamente se llaman a dos funciones de *callback*.

Los *callback* son funciones que proporcionan una determinada respuesta según ciertas circunstancias (eventos). En un protocolo de comunicación como el MQTT es necesario que un cliente pueda reaccionar a un determinado evento como la conexión exitosa con el broker, la desconexión, la publicación de un mensaje, o la recepción de unos mensajes después de la suscripción a un determinado topic. De hecho, *Paho MQTT* proporciona soporte a varios eventos que pueden ocurrir en una comunicación MQTT. Los eventos a los que es necesario proporcionar una respuesta en este proyecto son la conexión exitosa con el bróker, y la recepción de mensajes en los diferentes *topics* a los cuales el proxy se ha suscrito.

Con **on_connect** identificamos el evento de conexión exitosa del cliente con el broker. En la *ilustración 13* podemos observar la siguiente línea de código:

```
client.on_connect = on_connect
```

Con *client.on_connect* se identifica el *trigger* de conexión exitosa para el cliente (*client*) que acabamos de inicializar. El segundo *on_connect* es la función que se dispara cuando se cumple el evento de conexión exitosa. Por lo tanto, se ha desarrollado una función que se llama *on_connect* (se hubiese podido llamar de cualquiera manera), y cuyo funcionamiento veremos más adelante.

Lo mismo ocurre con **on_message**, que identifica el evento de recepción de un mensaje en uno de los topics al cual el proxy se ha suscrito. Como en el caso anterior, se ha desarrollado una función que se llama *on_message* que se dispara cada vez que el proxy recibe un mensaje.

Una vez que se haya inicializado el cliente y las dos funciones de *callback* el proxy realiza la conexión al broker usando la función *connect()* y pasando como parámetros la dirección IP del broker (asignada por el enrutador), el puerto (establecido en el fichero de configuración del broker Mosquitto), y el intervalo **Keep Alive**.

La función **Keep Alive** (“mantener vivo” en español) permite garantizar que ambas partes (client y broker) de una conexión MQTT conozcan el estado de conexión del otro.

Cuando un cliente no está enviando ningún mensaje debe enviar un mensaje denominado *PINGREQ* antes que expire el intervalo **Keep Alive** definido en el



momento de la conexión. El broker, una vez recibido este mensaje, contestará con otro mensaje denominado *PINGRESP*.

Esta función es esencial en cuanto, trabajando con dispositivos de bajo coste, es conveniente evitar cualquiera comunicación inútil.

Por lo tanto, es así como el broker detecta cuando a un cliente se le ha caído la conexión, y entonces envía el *last will message*, argumento que ya se ha tratado en el capítulo 3 del trabajo.

En la última línea del *main* se utiliza la función *loop_forever()* de manera que el *client* se mantenga en un bucle de red, y no se desconecte hasta que el mismo proxy llame a la función *disconnect()*. Así el proxy estará constantemente conectado al broker esperando de recibir mensajes.

En el siguiente apartado se analizará la función *on_connect*, o sea, la función que se dispara cuando ocurre el evento de conexión exitosa con el broker.

4.2.2 Función On_Connect

Como se ha afirmado en el apartado anterior, la función *On_Connect* es la que se dispara cuando se establece la conexión entre cliente y broker.

Técnicamente, cuando un cliente quiere iniciar una conexión, envía un mensaje *CONNECT*. El broker contesta con un mensaje *CONNACK* y un código de estado. A partir de este momento se inicia una conexión que se mantiene hasta que ambas partes respeten el intervalo *Keep Alive*, o el cliente (el proxy en este caso) envíe el mensaje de desconexión.

Bien, una vez que se haya establecido la conexión el proxy, debe suscribirse a todos los *topics* necesarios (ver ilustración 14).

```
494  # La función callback que se dispara cuando el cliente
495  # recibe la respuesta CONNACK del broker.
496  def on_connect(client_proxy, user, password, rc):
497      print("Connected with result code " + str(rc))
498      client_proxy.subscribe("sub")
499      client_proxy.subscribe("offline")
500      client_proxy.subscribe("request")
501
```

Ilustración 14 – Función On_Connect.

Los parámetros de entrada de la función son:

- *client_proxy* que sería el cliente que se ha inicializado en el *main* (el proxy).

- *user* y *password* que no vamos a utilizar ya que, como afirmado antes, siendo en un entorno domestico se ha evaluado no necesario.
- *rc* es el código resultado enviado por el broker al proxy en el momento de la conexión.

Una vez imprimido por pantalla el código resultado, el proxy se suscribe a todos los *topics* necesarios para el correcto funcionamiento del sistema que ya se han definido en el capítulo “*Análisis del problema*”.

Con el *topic* “**sub**” el proxy se suscribe a los mensajes que contienen el código identificativo y el *topic* (o los *topics*) a los cuales se hayan suscritos los clientes mediante el uso de *wildcards*.

En el *topic* “**offline**” se suscribe a los *last will message* que contienen los códigos identificativos de los clientes a los cuales se le ha caído la conexión.

En el *topic* “**request**” se suscribe a los mensajes que contiene las peticiones de informaciones meteorológicas de los clientes, que contienen el código identificativo del cliente, el *topic* de la petición, y la frecuencia de recepción deseada.

4.2.3 Función On_Message

La función On_Message es la que se ejecuta cada vez que el proxy recibe un mensaje publicado en alguno de los *topics* al cual se ha suscrito.

Como sabemos, el proxy puede recibir mensajes en tres diferentes *topics*; por lo tanto, tiene que proporcionar una respuesta diferente en cada caso.

I. Topic “**offline**”

En el *topic* “**offline**” se publican los *last will messages* con el código identificativo de los clientes que no han respetado el intervalo *keep alive*.



```

508 def on_message(client_proxy, userdata, msg):
509     if msg.topic == "offline":
510         msg.payload = msg.payload.decode("utf-8")
511         id_client = str(msg.payload)
512         print(id_client + " disconetted")
513         remove_client(id_client)
514         print("Topics que quedan activos: ")
515         print(topics)

```

Ilustración 15 – Función On_Message, topic “offline”

Como se puede observar en la *ilustración 15* uno de los parámetros de entrada de la función es *msg*, que es el mensaje recibido.

Cada mensaje tiene diferentes parámetros consultables, el primero que tenemos que averiguar es el *topic* en el cual ha sido publicado el mensaje (*msg.topic*).

Si el *topic* es “*offline*”, el contenido del mensaje es una simple cadena de texto que contiene el código identificativo del cliente. Por lo tanto, primero hay que extrapolar esta información (*msg.payload*), y seguidamente eliminar el cliente del diccionario de *topics* activos. Para cumplir esta acción se ha desarrollado una función que se llama *remove_client*.

En esta función se pasa como parámetro el código identificativo del cliente, seguidamente se recorre cada listado de clientes (*value*) de cada topic (*key*) con el fin de encontrar todos los *topics* al cual se haya suscrito el cliente, eliminar el *client_id* en cada listado en el cual está presente, y por último comprobar si quedan clientes suscritos al *topic*. Si no quedan, se elimina el *topic* del diccionario de *topics* activos (ver ilustración 16).

```

464 # Función para eliminar el cliente de todas las listas de clientes
465 # en cada topic en el cual se haya suscrito.
466 def remove_client(id_client):
467     for key, value in topics.items():
468         for val in value:
469             if id_client in val:
470                 topic = key
471                 topics[topic].remove(val)
472                 if len(topics[topic]) == 0:
473                     topics.pop(topic)

```

Ilustración 16 – Función *remove_client*.

II. Topic “sub”

Si el topic donde se ha publicado el mensaje es “**sub**”, el proxy recibe un mensaje JSON con dos parámetros: *id_client* y *topic*. Cuando un cliente se suscribe a un *topic* usando un *wildcard* debe comunicar al proxy todos los topics de los mensajes que recibe (este argumento se tratará más en detalle en el apartado “*Programación del client*”), para que el proxy pueda saber que hay un nuevo cliente suscrito a uno o más topics y por lo tanto actualizar los listados de clientes suscritos en el diccionario de topic (*topics*).

```
517 elif msg.topic == "sub":
518     m_decode = str(msg.payload.decode("utf-8", "ignore"))
519     m_in = json.loads(m_decode)
520     message = m_in["message"]
521     id_client = message["client_id"]
522     topic = message["topic"]
523     add_client(topic, id_client)
```

Ilustración 17 – Función *On_Message*, topic “sub”.

La estructura del mensaje es la siguiente:

```
{ "message":
  { "client_id": client_id
    "topic": topic }
}
```

Por lo tanto, una vez decodificado el mensaje, cargamos su contenido usando la función *loads* de la librería *json*, se consultan los datos *id_client* y *topic*, y se llama la función *add_client* pasando como parámetros los dos datos que se acaban de recibir.

La función *add_client* permite añadir un cliente al listado de clientes del topic al cual se haya suscrito.

Como sabemos, cuando un cliente realiza una petición, puede comunicar la frecuencia de respuesta deseada. Si varios clientes se suscriben al mismo *topic* la prioridad será asignada a la frecuencia más baja. Bien, cuando un cliente se suscribe a un *topic* mediante *wildcard* no comunica ninguna frecuencia; por lo tanto, se ha optado por asignar automáticamente la frecuencia de menor prioridad (la más alta) del listado de clientes suscritos al *topic*.

Entonces, en la función *add_client* se busca la frecuencia más alta del listado de clientes del topic correspondiente, una vez encontrada se añade el nuevo cliente (ver ilustración 18).

```

476 # Función para añadir nuevo cliente suscrito con wildcard
477 def add_client(topic, id_client):
478     aux = topics[topic]
479     freq_max = aux[0][1]
480     for par in aux:
481         if par[1] >= freq_max:
482             freq_max = par[1]
483     topics[topic].append([id_client, freq_max])

```

Ilustración 18 – Función *add_client*.

III. Topic “request”

Los mensajes publicados en el topic “*request*” son las peticiones de informaciones meteorológicas.

La estructura del mensaje es la siguiente:

```

{ "message":
  { "client_id": client_id
    "topic": topic
    "freq": frecuencia }
}

```

Primero hay que decodificar el mensaje y cargar el contenido con la función *loads* de la librería *json*.

Segundo inicializar todas las variables asignando a cada una el contenido correspondiente presente en el mensaje.

```

513 else:
514     m_decode = str(msg.payload.decode("utf-8", "ignore"))
515     m_in = json.loads(m_decode)
516     message = m_in["message"]
517     print(msg.topic + " " + m_decode)
518     id_client = message["client_id"]
519     topic = message["topic"]
520     freq = message["freq"]
521     print('ID client = ' + id_client)
522     print(topic)
523     freq = float(freq)

```

Ilustración 19 – Función *On_Message* topic “*request*”, decodificación mensaje e inicialización variables

Por último, hay que averiguar el valor de frecuencia:

Si es mayor que cero, quiere decir que el cliente desea recibir las informaciones meteorológicas a la frecuencia solicitada. Por lo tanto, hay que comprobar si el *topic* ya existe, en cuyo caso simplemente hay que añadir el cliente (el par *id_client* – *frecuencia*) a la lista de clientes en el *topic* correspondiente del diccionario de *topics activos*. Si el *topic* no existe después de crear una nueva entrada en el diccionario de *topics activos* se inicializa un *thread* que ejecuta la función **api_request** pasando como parámetros el *topic* y *frecuencia*.

Si es igual a cero, el cliente está realizando una petición sin frecuencia de respuesta; por lo tanto, se ejecuta la función **api request**, se genera el mensaje en formato JSON usando la función *dumps* de la librería *json*, y se publica el mensaje en el *topic* correspondiente.

```
524     if freq > 0:
525         if topic not in topics.keys():
526             topics[topic] = [[id_client, freq]]
527             locals()[id_client] = threading.Thread(target=api_request, args=(topic, freq,))
528             locals()[id_client].start()
529         else:
530             topics[topic].append([id_client, freq])
531     else:
532         res = api_request(topic, freq)
533         res = json.dumps(res,
534                         indent=6,
535                         separators=(",", " : "),
536                         sort_keys=True)
537         client_proxy.publish(topic, res)
```

Ilustración 20 – Función *On_Message*, llamada a la función *api_request* según el valor de frecuencia.

4.2.4 Función *Api_request*

La función *api_request* es la encargada de seleccionar la base de datos correcta según la petición del cliente, llamar a la función que realiza la petición REST a la API de Openweather, y devolver la respuesta. Los parámetros de entrada son el *topic* y la *frecuencia*.

Como sabemos, el *topic* tiene una estructura multinivel donde el primer nivel representa la colección de datos. Por lo tanto, el primer paso es descomponer el *topic*, para poder consultar cada nivel que lo compone.

Siendo el *topic* una cadena de caracteres donde los diferentes niveles están divididos por el carácter “/”, se utiliza la función *split* de Python pasando como parámetro el carácter “/”. La función devuelve una lista con todos los niveles del *topic*.

Por ejemplo, siendo el topic:

`“weather/city_name/Valencia/temp”`

el resultado al aplicar la función `split` es:

`[“weather”, “city_name”, “Valencia”, “temp”]`

Como se ha podido apreciar en la *ilustración 10* del capítulo tres, el primer nivel del *topic* representa la colección de datos; por lo tanto, es el primer elemento de la lista denominada *call_items* (ver *ilustración 21*).

```
402     # Función encargada de seleccionar la base de datos correcta según
403     # la petición del cliente.
404     def api_request(topic, freq):
405         call_items = topic.split("/")
406         database = call_items[0]
```

Ilustración 21 – Función `Api_request`, descomposición *topic*.

Como sabemos hay cinco colecciones de datos, y por cada una de ellas se ha desarrollado una función que se encarga de realizar la petición REST componiendo la URL.

Antes de verificar cual es la colección de datos, hay que comprobar si la frecuencia es cero (petición simple) o mayor que cero (petición con frecuencia). Si la frecuencia es cero, quiere decir que la función está siendo ejecutada por el hilo principal, y tiene que contestar a una petición simple (sin frecuencia). Si la frecuencia es mayor que cero, la función se está ejecutando por algún *thread* y con una frecuencia de publicación.

En ambos casos se comprueba cual es la colección de datos, y se ejecuta la función correspondiente, pasando como parámetros todos los varios niveles del *topic* necesarios para realizar la consulta. Si la colección de datos es incorrecta se publica un mensaje de error en el *topic* de la petición (ver *ilustración 22*).

```

408     if database == 'weather':
409         res = current_weather(call_items[1], call_items[2], call_items[3])
410     elif database == 'forecast':
411         res = forecast(call_items[1], call_items[2], call_items[3], call_items[4], call_items[5])
412     elif database == 'solar_radiation':
413         res = solar_radiation(call_items[1], call_items[2], call_items[3])
414     elif database == 'air_pollution':
415         res = air_pollution(call_items[1], call_items[2], call_items[3])
416     else:
417         error = {}
418         message = "Database incorreto"
419         error.update({"Error": message})
420         return error
421     return res

```

Ilustración 22 – Función *api_request*, comprobación colección de datos

En el caso de una petición con frecuencia el *thread* actúa como un nuevo cliente MQTT; por lo tanto, tiene que inicializar un nuevo cliente MQTT (*client_aux*) y realizar una conexión con el *broker*. Seguidamente se inicia un bucle infinito (*while True*).

```

423     else:
424         client_aux = mqtt.Client()
425         client_aux.connect("192.168.1.246", 1883, 60)
426         while True:
427             if topic not in topics.keys():
428                 break
429             if get_min_freq(topic) != freq:
430                 freq = get_min_freq(topic)
431

```

Ilustración 23 – Función *api_request*, zona Thread. Inicialización de un nuevo *client* y verificación de existencia y frecuencia mínima del topic en el diccionario.

Primero se comprueba si el *topic* sigue existiendo en el diccionario de *topic*. Si el *topic* ha sido eliminado (por qué no quedaban clientes suscritos), el *thread* sale del bucle infinito y termina su ejecución.

Segundo, se comprueba si la frecuencia actual corresponde a la frecuencia mínima de la lista de clientes suscritos al *topic*, y en caso contrario se actualiza. Para comprobar la frecuencia mínima de un *topic* se ha desarrollado la función *get_min_freq*.

```

472 # Función para obtener la frecuencia mínima de respuesta de los
473 # clientes suscritos a un determinado topic.
474 def get_min_freq(topic):
475     aux = topics[topic]
476     freq_min = aux[0][1]
477     for par in aux:
478         if par[1] <= freq_min:
479             freq_min = par[1]
480     return freq_min

```

Ilustración 24 – Función *get_min_freq*

Tercero, se ejecuta la función para realizar la petición REST según la colección de datos.

Por último, una vez obtenida la respuesta, se publica en el *topic* correspondiente, el *thread* detiene su ejecución por el número de segundos definidos por la variable frecuencia (*time.sleep(freq)*)m y se vuelve al principio del bucle.

```

432     if database == 'weather':
433         res = current_weather(call_items[1], call_items[2], call_items[3])
434     elif database == 'forecast':
435         res = forecast(call_items[1], call_items[2], call_items[3], call_items[4], call_items[5])
436     elif database == 'solar_radiation':
437         res = solar_radiation(call_items[1], call_items[2], call_items[3])
438     elif database == 'air_pollution':
439         res = air_pollution(call_items[1], call_items[2], call_items[3])
440     else:
441         error = {}
442         message = "Database incorrecto"
443         error.update({"Error": message})
444         client_aux.publish(topic, error)
445         break
446     res = json.dumps(res,
447                     indent=6,
448                     separators=(",", " : "),
449                     sort_keys=True)
450     client_aux.publish(topic, res)
451     time.sleep(freq)

```

Ilustración 25 – Función *api_request*, zona *Thread*. Llamada a la función para realizar la petición, publicación mensaje y detenimiento ejecución según la frecuencia especificada.

4.2.5 Peticiones a la API de OpenWeather

Como se ha podido observar en el apartado anterior, se han desarrollado varias funciones para realizar las peticiones en la diferentes colecciones de datos ofrecidas por el API de OpenWeather.

Las funciones son:

- I. *current_weather()* para la colección de datos **Current Weather Data**.

- II. *forecast()* para las colecciones de datos **Hourly Forecast 4 days** y **Daily Forecast 16 days**.
- III. *solar_radiation()* para la colección de datos **Solar Radiation API**.
- IV. *air_pollution()* para la colección de datos **Air Pollution API**.

En los siguientes apartados vamos a analizar cada una de ellas.

4.2.5.1 Función *current_weather*

La función *current_weather()* es la encargada de generar la URL para realizar la petición a la API de Openweather, según los datos proporcionados por el cliente.

Hay 3 parametros de entrada:

- *Search_by* es el segundo nivel del topic. Representa la tipología de búsqueda; como se ha podido apreciar en el apartado del capítulo 3 donde se analizaba la API de OpenWeather, existen diferentes tipologías de búsqueda.
- *location* es el tercer nivel del topic. Son los datos según la tipología de búsqueda.
- *data* es el cuarto y último nivel del topic. Representa los datos meteorológicos pedidos por el cliente. Puede ser un parámetro, más de uno (separados por “,”), o todos los parámetros disponibles para la colección de datos usando la palabra “all”.

El primer paso es inicializar algunas variables:

- *get_all_data* es una variable booleana. Si está a **False** quiere decir que el cliente no ha pedido todos los parámetros disponibles usando la palabra “all”; en caso contrario está a **True**. En principio se inicializa a *False*.
- *num_city* es el número de ciudades incluidos en la búsqueda. Como sabemos la colección de dato Current Weather permite realizar peticiones de datos meteorológicos para diferentes ciudades en la misma consulta. Se inicializa a 1.
- *base_url* es una cadena de texto que representa la primera parte de la URL que hay que componer para realizar la petición REST. Para esta colección de datos existen tres “*url_base*” diferentes: una para la búsqueda de varias ciudades en una zona rectangular (*box*), otra para la búsqueda de diferentes ciudades en un área circular (*find*), y una última para todas las demás tipologías de búsqueda.



```

347 def current_weather(search_by, location, data):
348     get_all_data = 0
349     num_city = 1
350     if search_by == 'box':
351         base_url = "http://api.openweathermap.org/data/2.5/box/city?"
352     elif search_by == 'find':
353         base_url = "http://api.openweathermap.org/data/2.5/find?"
354     else:
355         base_url = "http://api.openweathermap.org/data/2.5/weather?"
356

```

Ilustración 26 – Función *current_weather*, inicializaciones variables.

El segundo paso es componer la URL según la tipología de búsqueda elegida por el cliente. Si el criterio de búsqueda elegido por el cliente no está soportado por el proxy se devuelve un mensaje de error.

```

357     if search_by == 'city_name':
358         city = location
359         url = base_url + "q=" + city + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
360     elif search_by == 'city_ID':
361         city_id = location
362         url = base_url + "id=" + city_id + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
363     elif search_by == 'geo_coord':
364         coord = location.split(',')
365         lat = coord[0]
366         lon = coord[1]
367         url = base_url + "lat=" + lat + "&lon=" + lon + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
368     elif search_by == 'ZIP_code':
369         zip_code = location
370         url = base_url + "zip=" + zip_code + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
371     elif search_by == 'box':
372         coord = location.split(',')
373         num_city = int(coord[4])
374         url = base_url + "bbox=" + location + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
375     elif search_by == 'find':
376         coord = location.split(',')
377         lat = coord[0]
378         lon = coord[1]
379         num_city = coord[2]
380         url = base_url + "lat=" + lat + "&lon=" + lon + "&cnt=" + num_city + "&lang=" + lang + "&appid=" + \
381             + api_key + "&units=metric"
382         num_city = int(coord[2])
383     else:
384         error = {}
385         message = search_by + " no es un criterio de búsqueda aceptado"
386         error.update({"Error": message})
387         return error

```

Ilustración 27 – Función *current_weather*, composición URL según la tipología de búsqueda.

El tercer paso, después de comprobar si el cliente ha pedido todos los parámetros (“all”), es realizar la petición a la API mediante la función *get()* de la librería HTTP *request*, pasando como parámetro la URL que se acaba de componer. El resultado será un mensaje en formato JSON.

El último paso es generar la respuesta para el *client* según los parámetros pedidos, de manera que el *thread* la pueda publicar en el topic correspondiente. Para ello se ha desarrollado una función específica.

Antes de llamar esta función hay que asegurarse que no haya ocurrido algún problema con la petición a la API de OpenWeather. Cada mensaje de respuesta de la API tiene, entre los varios parámetros, un código de estado (ver tabla 8), que son los típicos de HTTP.

Tabla 8 – Códigos de estado de los mensajes.

Código estado	Descripción
1xx	Mensaje informativo.
2xx	Éxito.
3xx	Redirección.
4xx	Error por parte del cliente.
5xx	Error por parte del servidor.

Si el código de estado corresponde a “200” esto quiere decir que la petición ha tenido el éxito esperado, y por lo tanto se puede generar la respuesta llamando la función `get_current_weather_data()`. En caso contrario se genera un mensaje de error extrapolando los datos del error recibidos en la respuesta del API. De esta manera el cliente será consciente de la tipología de error, y podrá tomar las medidas necesarias para solucionarlo (si depende de él).

```
389     if data == 'all':
390         get_all_data = True
391     else:
392         data = data.split(',')
393
394     req = requests.get(url)
395     req = req.json()
396     if str(req["cod"]) == "200":
397         res = get_current_weather_data(req, data, get_all_data, num_city)
398         return res
399     else:
400         error = {}
401         error.update({req["cod"]: req["message"]})
402         return error
```

Ilustración 28 – Función `current_weather`, comprobación variable `data`, petición a la API de OpenWeather, comprobación código de estado mensaje, y llamada a la función para generar la respuesta para el cliente.

Ahora pasamos a analizar la función `get_current_weather_data()`, con la cual generamos la respuesta a la petición del cliente.

Los parámetros de entrada de la función son:

- `req` es la respuesta en formato JSON de la API de OpenWeather. Es el mensaje donde hay que extrapolar todos los datos según los parámetros pedidos por el cliente.
- `data` es el parámetro o la lista de parámetros pedidos por el cliente.
- `get_all_data` es la variable booleana que indica si el cliente ha pedido todos los parámetros disponibles (`True`).
- `num_city` es una variable numérica que representa el número de ciudades de las cuales el cliente ha pedido informaciones meteorológicas realizando una búsqueda por área geográfica.

Para generar la respuesta se inicializa un diccionario vacío (`res`), que se irá rellenando con todos los parámetros pedidos por el cliente extrapolar los datos de la respuesta de la API (`req`).

Antes de generar la respuesta se declara un diccionario que será útil para modificar los nombres de los datos presentes en la respuesta (ver ilustración 23).

```
302 def get_current_weather_data(req, data, get_all_data, num_city):
303     res = {}
304     weather_dict = {"temp": "Temperatura", "feels_like": "Sensacion_termica",
305                   "temp_min": "Temperatura_min", "temp_max": "Temperatura_max",
306                   "pressure": "Presion_atmosferica", "humidity": "Humedad",
307                   "speed": "Velocidad_viento", "deg": "Direccion_viento"}
308
```

Ilustración 29 – Función `get_current_weather`, inicializaciones variables

Cuando se genera la respuesta para el cliente, por ejemplo, el parámetro `temp` aparecerá en la respuesta como `Temperatura` o el parámetro `feels_like` como `Sensación_termica`.

Bien, ahora se puede pasar a la generación de la respuesta.

El primer factor que hay que considerar es el número de ciudades que hay que incluir en la respuesta, ya que la estructura de los mensajes, tanto de la API como los que hay que generar para los clientes, es diferente si se piden datos de más ciudades respecto a una sola localidad.

En esta sección de la función lo que se hace es crear un diccionario auxiliar (*aux*) que será útil para luego generar la respuesta para el cliente. Como se puede observar en la *ilustración 30* si la petición realizada es para un numero de ciudades distinto de uno, el mensaje recibido por la API de OpenWeather tiene una estructura diferente respecto a lo que se recibe realizando una petición sobre una sola ciudad (o localidad).

```
309     for city in range(0, num_city):
310         aux = {}
311         keys = []
312         vals = []
313
314         if num_city != 1:
315             items = req['list']
316             if city == len(items):
317                 break
318             item = items[city]
319             aux.update(item["main"])
320             aux.update(item["wind"])
321             city_name = item["name"]
322         else:
323             aux.update(req["main"])
324             aux.update(req["wind"])
```

Ilustración 30 – Función *get_current_weather*, generación diccionario auxiliar.

El segundo factor por considerar son los parámetros pedidos por el cliente que, como se ha dicho, puede ser uno, más de uno, o todos aquellos disponibles. Si entre los parámetros pedidos hay uno que no está disponible, se incluirá en la respuesta poniendo como valor la cadena “*No disponible*”. Una vez generada la respuesta (que es un diccionario) la función la devuelve de manera que pueda ser convertida en un objeto JSON, y publicada en el topic correspondiente.

```
326     if not get_all_data:
327         for parameter in data:
328             if weather_dict.get(parameter) is not None:
329                 keys.append(weather_dict[parameter])
330                 vals.append(aux[parameter])
331             else:
332                 keys.append(parameter)
333                 vals.append("No disponible")
334         if num_city != 1:
335             res.update({city_name: dict(zip(keys, vals))})
336         else:
337             res.update({"main": dict(zip(keys, vals))})
338     else:
339         keys = weather_dict.values()
340         for key in weather_dict.keys():
341             vals.append(aux[key])
342         if num_city != 1:
343             res.update({city_name: dict(zip(keys, vals))})
344         else:
345             res.update({"main": dict(zip(keys, vals))})
346     return res
```

Ilustración 31 – Función *get_current_weather*, generación de respuesta para el cliente.

Todas las funciones desarrolladas para generar las peticiones son muy parecidas; de hecho, al analizar las restantes (*forecast*, *solar_radiation*, *air_pollution*) nos centraremos solo en los que difieren entre sí para que la explicación no resulte demasiado pesada y redundante.

4.2.5.2 Función *forecast*

La función *forecast* además de los parámetros de entrada *search_by*, *location* y *data* que ya conocemos, necesita de otros dos:

- *period* que se refiere a la periodicidad de las informaciones pedidas. De hecho, como sabemos, hay dos colecciones de datos que proporcionan predicciones meteorológicas: una a nivel horario (*hourly*), y otra a nivel diario (*daily*).
- *count* es una variable numérica que representa el número de horas o días de las cuales el cliente quiere las predicciones meteorológicas.

Por lo tanto, hay que comprobar si el *period* es diario o horario, ya que son dos colecciones de datos diferentes con una URL diferente. Si el periodo no corresponde a una de las dos opciones admitidas se devuelve un mensaje de error.

```
251 forecast(period, search_by, location, count, data):
252     get_all_data = 0
253     if period == 'hourly':
254         base_url = "http://pro.openweathermap.org/data/2.5/forecast/hourly?"
255     elif period == 'daily':
256         base_url = "http://api.openweathermap.org/data/2.5/forecast/daily?"
257     else:
258         error = {}
259         message = "Elegir 'hourly' o 'forecast'"
260         error.update({"Error": message})
261     return error
262
```

Ilustración 32 – Función *forecast*, inicializaciones variables.

Seguidamente se compone la URL según la tipología de búsqueda; se realiza la petición a la API de OpenWeather y, si la respuesta lleva el código de estado “200”, se llama a la función para generar la respuesta, que en el caso de *period* ‘Hourly’ es la función *get_forecast_hourly_data()*, mientras en el caso de *period* ‘Daily’ es la función *get_forecast_daily_data()*.

```

288     req = requests.get(url)
289     req = req.json()
290     if str(req["cod"]) == "200":
291         if period == 'hourly':
292             res = get_forecast_hourly_data(req, data, get_all_data, count)
293         if period == 'daily':
294             res = get_forecast_daily_data(req, data, get_all_data, count)
295         return res
296     else:
297         error = {}
298         error.update({req["cod"]: req["message"]})
299         return error

```

Ilustración 33 – Función forecast, petición a la API de OpenWeather y llamada a la función para la generación de la respuesta según el “period” elegido por el cliente.

Las funciones *get_forecast_hourly_data()* y *get_forecast_daily_data()* tienen una estructura muy parecida a la función *get_current_weather*. Se inicializa un diccionario para la traducción de los parámetros, y se genera un diccionario auxiliar para luego poder componer la respuesta del cliente según los datos pedidos.

La única diferencia es que, si en el caso de *current_weather*, la estructura del mensaje dependía del número de ciudades, ahora depende del número de horas (*get_forecast_hourly_data*) o días (*get_forecast_daily_data*).

4.2.5.3 Función solar_radiation

Los parámetros de entrada de la función *solar_radiation* son *period*, *location* y *data*.

Esta colección de datos admite solo búsquedas por coordenadas geográficas y, por lo tanto, el parámetro *location* solo admite los valores de *latitud* y *longitud* separados por “,”.

El parámetro *period* admite solo dos valores:

- *current* para obtener datos sobre radiaciones solares actuales.
- *forecast* para obtener predicciones sobre datos de radiaciones solares.

```

128 def solar_radiation(period, location, data):
129     get_all_data = False
130
131     if data == 'all':
132         get_all_data = True
133     else:
134         data = data.split(',')
135
136     coord = location.split(',')
137     lat = coord[0]
138     lon = coord[1]
139
140     if period == 'current':
141         count = 1
142         base_url = "http://api.openweathermap.org/data/2.5/solar_radiation?"
143         url = base_url + "lat=" + lat + "&lon=" + lon + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
144     elif period == 'forecast':
145         count = int(coord[2])
146         base_url = "http://api.openweathermap.org/data/2.5/solar_radiation/forecast?"
147         url = base_url + "lat=" + lat + "&lon=" + lon + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
148     else:
149         error = {}
150         message = "Elegir 'current' o 'forecast'"
151         error.update({"Error": message})
152         return error

```

Ilustración 34 – Función solar_radiation, gestión datos, coordenadas geográficas y composición URL.

Como siempre se compone la URL para realizar la petición REST y, una vez obtenida la respuesta, se llama a la función específica para extrapolar los datos y generar la respuesta para el cliente. En este caso la función es *get_solar_radiation_data()*, la cual se comporta de forma casi idéntica a las funciones de generación de mensajes descritas anteriormente (obviamente adaptándose a la colección de datos *solar_radiation*).

```

154     req = requests.get(url)
155     req = req.json()
156     if "list" in req:
157         res = get_solar_radiation_data(req, data, get_all_data, count)
158         return res
159     else:
160         error = {}
161         message = "No hay resultados"
162         error.update({req["cod"]: message})
163         return error

```

Ilustración 35 – Función solar_radiation, petición a la API de OpenWeather, y llamada a la función para generar la respuesta para el cliente.

4.2.5.4 Función `air_pollution`

La función `air_pollution` tiene los mismos parámetros de entrada de la función `solar_radation` y, al igual que la colección de datos `solar_radiation`, solo admite búsquedas por coordenadas geográficas.

La única diferencia se encuentra en los valores admitidos para el parámetro `period`:

- `current` para pedir datos actuales sobre contaminación aire.
- `forecast` para pedir predicciones de datos sobre contaminación aire.
- `historical` para pedir datos históricos sobre contaminación aire.

```
61 if period == "current":
62     base_url = "http://api.openweathermap.org/data/2.5/air_pollution/forecast?"
63     url = base_url + "lat=" + lat + "&lon=" + lon + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
64 elif period == "forecast":
65     base_url = "http://api.openweathermap.org/data/2.5/air_pollution/forecast?"
66     count = int(coord[2])
67     url = base_url + "lat=" + lat + "&lon=" + lon + "&lang=" + lang + "&appid=" + api_key + "&units=metric"
68 elif period == "historical":
69     base_url = "http://api.openweathermap.org/data/2.5/air_pollution/history?"
70     start = str(int(dt.datetime.strptime(coord[2], "%Y-%m-%d").timestamp()))
71     end = str(int(dt.datetime.strptime(coord[3], "%Y-%m-%d").timestamp()))
72     historical = True
73     url = base_url + "lat=" + lat + "&lon=" + lon + "&start=" + start + "&end=" + end + "&lang=" + lang + "&appid=" +
74         api_key + "&units=metric"
75 else:
76     error = {}
77     message = "Elegir una opción entre 'current'-'forecast'-'historical'"
78     error.update({"Error": message})
79     return error
```

Ilustración 36 – Función `air_pollution`, composición URL según parámetro 'period'.

Una vez generada la URL se llama a la función `get_air_pollution_data()` para generar la respuesta al cliente.

4.3 Programación del cliente

Para el testeo del proxy ha sido necesario desarrollar un cliente para poder realizar las peticiones meteorológicas. Al igual que el proxy, el `client` se ha diseñado usando Python.

El cliente se ha diseñado para ser ejecutado por la línea de comandos, y admite dos argumentos:

- I. El `topic` para pedir informaciones meteorológicas. Se identifica con la etiqueta '`t`' o '`topic`'.

- II. La frecuencia con la cual se quiere recibir los datos periódicamente. Está por defecto a cero, y si no se le asigna un valor se realiza una petición “simple”. Se identifica con la etiqueta *f* o ‘frecuencia’.

Un ejemplo de ejecución del client por la línea de comandos es el siguiente:

```
python client.py -t "weather/city_name/Valencia/all" -f 600
```

Hecha esta premisa, empezamos con el análisis del código desarrollado.

Para la programación del proxy se han utilizado algunas de las librerías usadas en el proxy:

- *JSON* para poder generar los mensajes que hay que enviar al proxy (el broker en realidad) y para poder leer las respuestas.
- *Paho MQTT* para las comunicaciones MQTT.

Y otras librerías:

- *random* y *string* se utilizan para la generación del código identificativo aleatorio.
- *sys* y *getopt* para la gestión de los argumentos de entrada (topic y frecuencia).

```
1 import json
2 import random
3 import string
4 import sys
5 import getopt
6 import paho.mqtt.client as mqtt
```

Ilustración 37 – Client, importación librerías.

Una vez importadas las librerías necesarias se inicializan algunas variables:

- *client_id*: es el código identificativo aleatorio alfanumérico. Se genera usando la funciones *random.choices* que coge *k* letras o números de forma aleatoria. Se ha elegido un valor de *k* igual a 5.
- Se inicializa la variable *topic* a una cadena vacía.
- Se inicializa la variable booleana *wildcard* a **False**. Esta variable pasará a **True** si el topic contiene uno de los caracteres especiales ‘+’ o ‘#’.
- La variable *freq* representa la frecuencia. Se inicializa a 0.

- La variable *topics* se inicializa como una lista vacía. En esta lista se guardarán los topics de los mensajes que el cliente recibe si se suscribe a un topic que contiene *wildcard*. (topic 'sub').
- *message* es un diccionario vacío. Esta variable se rellena con todos los datos necesarios para realizar la petición y publicarla en el topic 'request'.

```

9      client_id = ''.join(random.choices(string.ascii_letters + string.digits, k=5))
10     topic = ""
11     wildcard = False
12     freq = 0
13     topics = []
14     message = {}

```

Ilustración 38 – Client, inicializaciones de variables.

El siguiente paso es leer los argumentos de entrada. Como he dicho antes son dos: topic y frecuencia. Una vez asignados a las variables *topic* y *freq* los valores pasados como argumentos, se comprueba si el *client* se ha suscrito a un topic usando un *wildcard*. En caso afirmativo la variable *wildcard* pasa a **True**.

```

17     opts, args = getopt.getopt(
18         sys.argv[1:],
19         't:f:',
20         ['topic', 'frecuencia'],
21     )
22     for opt, arg in opts:
23         if opt in ('-t', '--topic'):
24             topic = arg
25         elif opt in ('-f', '--frecuencia'):
26             freq = arg
27
28     if '+' in topic or '#' in topic:
29         wildcard = True

```

Ilustración 39 – Client, gestión argumentos de entrada y verificación de *wildcard*.

Ahora que conocemos el *id* del cliente, el topic, y la frecuencia, se puede inicializar el client MQTT. Como en el caso del proxy se utilizan las funciones de *callback on_connect* y *on_message*, y la función *loop_forever* para mantener el bucle de red.

Una diferencia respecto al proxy es que, antes de conectarse al broker con la función *connect*, el *client* debe comunicar su *last will message*.

Para este propósito la librería *Paho MQTT*, dispone de la función *will_set()*.

Cuando se llama la función *will_set* hay que pasar los siguientes parámetros:

- El topic, que en este caso es “*offline*”.
- El mensaje (*payload*), que es el código identificativo del cliente (*client_id*).
- *Quality of Service* (qos) representa el mecanismo de **Calidad de servicio** proporcionado por MQTT. Como se ha podido apreciar en el capítulo 2, hay tres diferentes niveles de calidad de servicio. Como es imprescindible que el *last will message* llegue el proxy (para evitar comunicaciones innecesarias) se ha fijado a 1.
- Con *retain* a **True** el broker conserva el *last will message*. Se ha considerado innecesario.

Otro aspecto en el cual difiere del proxy es el intervalo *keep alive* que, mientras para el proxy era de 60 segundos, para los clientes será menos largo (10 segundos).

```
70 client = mqtt.Client()
71 client.on_connect = on_connect
72 client.on_message = on_message
73 client.will_set("offline", payload=client_id, qos=1, retain=True)
74 client.connect("192.168.1.246", 1883, 10)
75 client.loop_forever()
```

Ilustración 40 – Client, conexión al broker.

4.4 Función `on_connect`

Una vez realizada la conexión con el broker se ejecuta la función `on_connect()` como se ha visto con el proxy.

En el caso del *client* el modo de actuar depende si en el topic se han usado *wildcard*.

En caso afirmativo la tarea que se cumple es simplemente suscribirse al topic y quedar a la espera de recibir mensajes.

Si no se han usado *wildcards*, el cliente tiene que publicar la petición en el topic ‘*request*’. Por lo tanto, primero hay que añadir todos los datos necesarios en el diccionario vacío (*message*) declarado previamente. Seguidamente se convierte el mensaje en un objeto JSON, se suscribe al topic de la información meteorológica de interés, y se publica el mensaje en el topic ‘*request*’, quedando en fin a la espera de recibir la respuesta en el topic correspondiente.


```

33 def on_connect(client, user, password, rc):
34     global message
35     global topic
36     print("Connected with result code " + str(rc))
37     if wildcard:
38         client.subscribe(topic)
39     else:
40         message.update({"message": {"client_id": client_id, "topic": topic, "freq": freq}})
41         message = json.dumps(message,
42                               indent=6,
43                               separators=(", ", " : "),
44                               sort_keys=True)
45         client.subscribe(topic)
46         client.publish("request", message)
47

```

Ilustración 41 – Client, función *on_connect*.

4.5 Función *on_message*

Esta es la función que se dispara cada vez que el cliente recibe un mensaje.

Cuando llega un nuevo mensaje primero se imprime su contenido por pantalla. Después se comprueba si es un mensaje publicado en un topic en el cual el cliente se haya suscrito usando *wildcard*. En caso afirmativo se comprueba si el topic del mensaje que se acaba de recibir está ya presente en la lista de topics en cual el cliente se ha suscrito mediante *wildcard* (*topics*). Si el topic no está presente se debe comunicar al proxy que un nuevo cliente se ha suscrito al topic. Por lo tanto, se compone un nuevo mensaje con el *client_id* y el *topic* del mensaje que se acaba de recibir, y se publica en el topic 'sub'.

Si no se han usado *wildcard*, y la *frecuencia* es igual a cero, quiere decir que el cliente ha realizado una petición "simple", y por lo tanto, una vez recibido el mensaje, se desconecta del proxy usando la función *disconnect()*.

```

51 def on_message(client, userdata, msg):
52     global message
53     m_decode = str(msg.payload.decode("utf-8", "ignore"))
54     print("data Received", m_decode)
55     print("\n")
56     if wildcard:
57         topic_in = msg.topic
58         if topic_in not in topics:
59             topics.append(topic_in)
60             message.update({"message": {"client_id": client_id, "topic": topic_in}})
61             message = json.dumps(message,
62                                 indent=6,
63                                 separators=(", ", " : "),
64                                 sort_keys=True)
65             client.publish("sub", message)
66     if freq == 0 and not wildcard:
67         client.disconnect()

```

Ilustración 42 – Client, función *on_message*.

5. Validación y pruebas

En este capítulo se enseñará mediante capturas de pantalla el funcionamiento del proxy. Se llevarán a cabo diferentes pruebas bajo varias condiciones para comprobar que el proxy proporcione una respuesta adecuada según lo definido en los capítulos anteriores.

Para el testeo del proxy se ha ubicado el *client* en un ordenador portátil.

5.1 Testeo de peticiones simples

Como se ha podido apreciar en el capítulo anterior, el broker Mosquitto se ha configurado para que se inicie automáticamente cada vez que se encienda la Raspberry Pi.

Por lo tanto, primero hay que comprobar si el broker está activo ejecutando el comando:

```
pi@raspberrypi:~ $ mosquitto -v
```

obteniendo la siguiente respuesta:

```
pi@raspberrypi:~ $ mosquitto -v
1630138813: mosquitto version 1.5.7 starting
1630138813: Using default config.
1630138813: Opening ipv4 listen socket on port 1883.
1630138813: Opening ipv6 listen socket on port 1883.
```

Ilustración 43 – Salida comando *mosquitto -v*.

Como se puede observar en la *ilustración 43*, el broker Mosquitto ha arrancado y ha abierto los sockets¹⁷ para escuchar en el puerto 1883.

Bien, ahora podemos arrancar el proxy. Para ello, nos movemos a la carpeta donde está ubicado el fichero *proxy.py* y lo ejecutamos.

```
pi@raspberrypi:~ $ cd ftp/files/TFG/Testing/
pi@raspberrypi:~/ftp/files/TFG/Testing $ python3 proxy.py
Connected with result code 0
```

Ilustración 44 – Ejecución proxy y mensaje de conexión exitosa.

Además, habiendo ejecutado el broker mosquitto con la opción *-verbose* (*-v*, podemos verificar todas las comunicaciones que están pasando por el broker. Una vez conectado

¹⁷ constituyen el mecanismo para la entrega de paquetes de datos provenientes de la tarjeta de red a los procesos o hilos apropiados. Un socket queda definido por un par de direcciones IP local y remota, un protocolo de transporte y un par de números de puerto local y remoto. [24]

el proxy podemos verificar que se ha disparado la función *on_connect* ya que el broker ha recibido las suscripciones a los topics *sub*, *offline* y *request* (ver ilustración 45).

```

pi@raspberrypi:~ $ mosquitto -v
1630141802: mosquitto version 1.5.7 starting
1630141802: Using default config.
1630141802: Opening ipv4 listen socket on port 1883.
1630141802: Opening ipv6 listen socket on port 1883.
1630142359: New connection from 192.168.1.246 on port 1883.
1630142359: New client connected from 192.168.1.246 as 5c646b24-3aca-433e-9267-a0e579acf79b (c1, k60).
1630142359: No will message specified.
1630142359: Sending CONNACK to 5c646b24-3aca-433e-9267-a0e579acf79b (0, 0)
1630142359: Received SUBSCRIBE from 5c646b24-3aca-433e-9267-a0e579acf79b
1630142359:   sub (QoS 0)
1630142359: 5c646b24-3aca-433e-9267-a0e579acf79b 0 sub
1630142359: Sending SUBACK to 5c646b24-3aca-433e-9267-a0e579acf79b
1630142359: Received SUBSCRIBE from 5c646b24-3aca-433e-9267-a0e579acf79b
1630142359:   offline (QoS 0)
1630142359: 5c646b24-3aca-433e-9267-a0e579acf79b 0 offline
1630142359: Sending SUBACK to 5c646b24-3aca-433e-9267-a0e579acf79b
1630142359: Received SUBSCRIBE from 5c646b24-3aca-433e-9267-a0e579acf79b
1630142359:   request (QoS 0)
1630142359: 5c646b24-3aca-433e-9267-a0e579acf79b 0 request
1630142359: Sending SUBACK to 5c646b24-3aca-433e-9267-a0e579acf79b
    
```

Ilustración 45 – Salida broker Mosquitto después de la conexión exitosa con el proxy

A partir de este momento el proxy está listo para recibir las peticiones de los varios clientes. Por lo tanto usamos el ordenador portátil donde se ha ubicado el *client* para realizar las pruebas.

Nos movemos a la carpeta donde se ha ubicado el fichero *client.py* y realizamos la primera prueba: una petición sin frecuencia (simple) sobre todos los datos meteorológicos actuales para la ciudad de Valencia.

```

PS C:\Users\santo> cd .\Desktop\TFG\Testing\
PS C:\Users\santo\Desktop\TFG\Testing> python client.py -t "weather/city_name/Valencia/all"
Connected with result code 0
data Received {
  "main" : {
    "Direccion_viento" : 265,
    "Humedad" : 29,
    "Presion_atmosferica" : 1013,
    "Sensacion_termica" : 26.56,
    "Temperatura" : 27.28,
    "Temperatura_max" : 29.12,
    "Temperatura_min" : 25.57,
    "Velocidad_viento" : 1.45
  }
}
    
```

Ilustración 46 – Ejemplo de petición sin frecuencia de respuesta donde se piden todos los parámetros (“all”).

Ahora pedimos los datos de temperatura, humedad y presión atmosférica para la ciudad de Barcelona:

```
PS C:\Users\santo\Desktop\TFG\Testing> python client.py -t "weather/city_name/Barcelona/temp,humidity,pressure"
Connected with result code 0
data Received {
  "main" : {
    "Humedad" : 63,
    "Presion_atmosferica" : 1013,
    "Temperatura" : 25.77
  }
}
```

Ilustración 47 – Ejemplo petición sin frecuencia de respuesta donde se piden algunos datos específicos

Como se ha podido apreciar en el capítulo anterior, se ha contemplado también la posibilidad que el cliente pueda pedir algún parámetro inexistente:

```
PS C:\Users\santo\Desktop\TFG\Testing> python client.py -t "weather/city_name/Madrid/temp_min,feels_like,hola"
Connected with result code 0
data Received {
  "main" : {
    "Sensacion_termica" : 26.95,
    "Temperatura_min" : 25.59,
    "hola" : "No disponible"
  }
}
```

Ilustración 48 – Ejemplo petición donde se pide un dato inexistente.

Como sabemos, se pueden realizar también peticiones más complejas, como pedir datos meteorológicos actuales para diferentes ciudades, o predicciones meteorológicas de varias horas o días.

En este ejemplo se piden los datos de temperatura mínima y máxima para 6 ciudades haciendo una búsqueda por área circular, y eligiendo como punto central de la búsqueda las coordenadas de Valencia:

```
PS C:\Users\santo\Desktop\TFG\Testing> python client.py -t "weather/find/39.4702,-0.3768,6/temp_min,temp_max"
Connected with result code 0
Hora actual: 2021-08-28 15:07:31
data Received {
  "Campanar" : {
    "Temperatura_max" : 31.88,
    "Temperatura_min" : 28.69
  },
  "Eixample" : {
    "Temperatura_max" : 31.92,
    "Temperatura_min" : 28.73
  },
  "Mislata" : {
    "Temperatura_max" : 31.82,
    "Temperatura_min" : 28.63
  },
  "Tavernes Blanques" : {
    "Temperatura_max" : 31.97,
    "Temperatura_min" : 28.78
  },
  "Valencia" : {
    "Temperatura_max" : 31.99,
    "Temperatura_min" : 28.8
  }
}
```

Ilustración 49 – Ejemplo petición de datos meteorológicos para diferentes localidades.

Ahora realizamos una consulta de predicciones meteorológicas.

Por ejemplo, queremos conocer la probabilidad de precipitaciones (*pop*) para el pueblo de Teruel para los próximos 4 días:

```
PS C:\Users\santo\Desktop\TFG\Testing> python client.py -t "forecast/daily/city_name/Teruel/4/pop"
Connected with result code 0
Hora actual: 2021-08-28 15:22:03
data Received {
  "2021/08/28" : {
    "prob_precipitaciones" : 0
  },
  "2021/08/29" : {
    "prob_precipitaciones" : 0.53
  },
  "2021/08/30" : {
    "prob_precipitaciones" : 0.88
  },
  "2021/08/31" : {
    "prob_precipitaciones" : 0.92
  }
}
```

Ilustración 50 – Ejemplo petición para probabilidad de precipitaciones. Valor mínimo = 0 (0%), valor máximo = 1 (100%).

5.2 Testeo de peticiones con frecuencia

En el próximo ejemplo se realiza una petición periódica. Siendo una fase de testeo, se ha realizado una pequeña modificación al código del *cliente* añadiendo la siguiente línea de código antes de aquella donde se imprime el mensaje recibido:

```
print("Hora actual: " + str(datetime.datetime.now()))
```

De esta manera se imprime por pantalla la fecha y hora actual antes de cada mensaje. Esto será útil para comprobar si la frecuencia de recepción ha sido respetada.

Bien, ahora pedimos las concentraciones actuales de Monóxido de Carbono (CO) y de Amoniaco (NH₃) para la ciudad de Valencia (pasando sus coordenadas geográficas) con una frecuencia de respuesta de 10 segundos:

```
PS C:\Users\santo\Desktop\TFG\Testing> python client.py -t "air_pollution/current/39.4702,-0.3768/co,nh3" -f 10
Connected with result code 0
Hora actual: 2021-08-28 12:42:50
data Received {
  "main" : {
    "Concentracion_CO" : 257.02,
    "Concentracion_NH3" : 1.71
  }
}

Hora actual: 2021-08-28 12:43:01
data Received {
  "main" : {
    "Concentracion_CO" : 257.02,
    "Concentracion_NH3" : 1.71
  }
}

Hora actual: 2021-08-28 12:43:11
data Received {
  "main" : {
    "Concentracion_CO" : 257.02,
    "Concentracion_NH3" : 1.71
  }
}

Hora actual: 2021-08-28 12:43:21
data Received {
  "main" : {
    "Concentracion_CO" : 257.02,
    "Concentracion_NH3" : 1.71
  }
}

Hora actual: 2021-08-28 12:43:31
data Received {
  "main" : {
    "Concentracion_CO" : 257.02,
    "Concentracion_NH3" : 1.71
  }
}
```

Ilustración 51 – Ejemplo petición con una frecuencia de respuesta de 10 segundos.

Como se puede observar en la *ilustración 51*, se obtiene una respuesta cada 10 segundos.

Hasta ahora hemos comprobado simplemente como el proxy contesta a las varias peticiones meteorológicas, y se ha podido apreciar su funcionamiento cuando es solo un cliente que realiza peticiones o se suscribe a un topic de información. En el contexto del internet de las cosas serán muchos los dispositivos conectados con el broker que realizarán sus peticiones; por lo tanto, es necesario testear como el proxy reacciona a una determinada carga de trabajo.

5.3 Testeo de carga y esfuerzo

Cabe destacar que la API de OpenWeather asigna un límite de peticiones por minuto en función de la *key* que tengas. En el caso de este proyecto, como se ha dicho previamente, se ha elegido una clave para estudiantes, cuyo límite es de 3000 peticiones/minuto.

En un entorno doméstico, considerando un número de dispositivos limitado, es un número de llamadas por minuto aceptable. En un entorno rural o industrial, dependiendo de la estructura y de la tipología de uso, podría ser necesario optar por una de las claves con un número de llamadas por minuto más elevado.

Hecha esta premisa, podemos empezar con el testeo de carga del proxy. Se ha elegido un número de 20 clientes conectados simultáneamente, considerando esto un número plausible en un contexto doméstico.

De estos 20 clientes:

- 16 clientes realizan peticiones con información de frecuencia, generando por lo tanto un *topic* de información meteorológica.
- 4 se suscriben a varios *topics* activos usando comodines (*wildcards*).

Tabla 9 – Test de carga. Listado de clientes con su *topic* y frecuencia.

Cliente	Topic	Frecuencia (en seg.)
Cliente 1	weather/city_name/Valencia/temp, feels_like	2
Cliente 2	weather/city_name/Valencia/temp, pressure	4
Cliente 3	weather/city_name/Valencia/humidity,	6
Cliente 4	weather/city_name/Valencia/temp_max	8
Cliente 5	forecast/hourly/city_name/Barcelona/5/pop	2
Cliente 6	forecast/hourly/city_name/Madrid/4/temp	4
Cliente 7	forecast/daily/zip_code/46020,ES/7/clouds	6
Cliente 8	forecast/daily/geo_coord/37.38,-5.99/2/all	8
Cliente 9	air_pollution/current/-3.70,40.41/pm2_5,co	2
Cliente 10	air_pollution/current/39.47,-0.37/no	4
Cliente 11	air_pollution/forecast/39.47,-0.37/12/aqi	6
Cliente 12	air_pollution/forecast/-3.70,40.41/pm10	8
Cliente 13	solar_radiation/current/-3.70,40,41/ghi	2
Cliente 14	solar_radiation/current/39.47,-0.37/dni	4
Cliente 15	solar_radiation/forecast/39.47,-0.37/all	6
Cliente 16	solar_radiation/forecast/-3.70,40,41,5/all	8
Cliente 17	weather/city_name/Valencia/+	-
Cliente 18	forecast/houly/#	-
Cliente 19	forecast/daily/#	-
Cliente 20	air_pollution/#	-

Existen entonces 16 topics de informaciones diferentes y, en cada uno de ellos, hay suscrito el cliente que generó la petición.

Además, hay otros 4 clientes suscritos mediante *wildcard*:

- El **Cliente 17** suscrito a los topics generados por los clientes 1, 2, 3 y 4.
- El **Cliente 18** suscrito a los topics generados por los clientes 5 y 6.
- El **Cliente 19** suscrito a los topics generados por los clientes 7 y 8.
- El **Cliente 20** suscrito a los topics generados por los clientes 9, 10, 11 y 12.

El análisis se centrará en los tiempos de recepción de los mensajes, es decir, se analizará si se cumple la frecuencia de publicación establecida en cada topic y, en el caso en que no se cumpla, analizar cuanto es el retraso.

Una vez se hayan arrancado todos los clientes, y se haya generado el flujo de tráfico, se anotan los tiempos de recepción de 10 mensajes por cada uno de los clientes. Cuando se anota el valor 0 quiere decir que se ha respetado la frecuencia establecida; en caso contrario se anota el retraso en segundos. Obviamente no se consideran los mensajes recibidos por los clientes que han usado wildcards ya que van recibiendo mensajes de diferentes topics.

Tabla 10 – Retrasos de los tiempos de publicación de cada topic analizados en 10 mensajes, y retraso acumulado (en segundos).

Cliente	Mensaje 1	Mensaje 2	Mensaje 3	Mensaje 4	Mensaje 5	Mensaje 6	Mensaje 7	Mensaje 8	Mensaje 9	Mensaje 10	Retraso acumulado
Cliente 1	0	0	0	0	1	0	0	0	0	0	1
Cliente 2	0	0	0	0	0	0	0	0	0	0	0
Cliente 3	0	1	0	0	0	0	0	0	1	0	2
Cliente 4	0	0	0	0	0	0	0	0	0	0	0
Cliente 5	1	0	0	0	0	0	0	0	0	0	1
Cliente 6	0	0	0	0	0	0	0	0	0	0	0
Cliente 7	0	0	0	0	0	0	0	0	0	0	0
Cliente 8	0	0	0	0	0	0	0	0	0	0	0
Cliente 9	0	1	0	0	0	0	0	0	0	0	1
Cliente 10	0	0	0	0	0	1	0	0	0	0	1
Cliente 11	0	0	0	0	0	0	0	0	0	0	0
Cliente 12	0	0	0	0	0	0	0	0	0	0	0
Cliente 13	0	0	0	0	0	0	0	0	0	0	0
Cliente 14	0	0	0	0	0	0	0	0	0	0	0
Cliente 15	0	0	0	0	0	0	0	0	0	0	0
Cliente 16	0	0	0	0	0	0	0	0	0	0	0

Se puede observar que, de 16 topics activos, cinco en algún momento han sufrido un retraso en el tiempo de publicación. En cuatro topics se ha acumulado en un total de 10 mensajes un retraso de un segundo. En el restante topic se ha acumulado un retraso de 2 segundos.

Para un entorno doméstico, en el cual es muy difícil que se deban cumplir tareas donde los tiempos son críticos, estos tiempos se consideran bastante aceptables, considerando también la tecnología de bajos recursos implementada.



6. Conclusiones

En este capítulo se expondrán las conclusiones del proyecto, hablando primero de las dificultades encontradas, segundo de posibles modificaciones y ampliaciones futuras, y por último proponiendo algunos ejemplos de uso concretos.

6.1 Dificultades encontradas

En el capítulo introductorio se habían detallado todos los objetivos que se pretendía conseguir con este trabajo. Como se habrá podido apreciar a lo largo de este documento, todos los objetivos propuestos se han cumplido, obviamente no sin encontrar algunas dificultades.

Lo que más costó en principio fue la adaptación de las informaciones meteorológicas que se podían pedir con la API de OpenWeather a la estructura multinivel de los *topic*. Se pretendía no distorsionar el concepto de *topic*, siguiendo las pautas del protocolo MQTT, y adaptándolo a un tipo de información más complejo que aquel para el que se utiliza habitualmente. Creo que en este aspecto del trabajo la solución encontrada es adecuada.

Otro aspecto que tuve que revisar varias veces fue el de la comunicación entre clientes y proxy: el intercambio de información, en concreto como los distintos clientes debían comunicar al proxy (a través del broker) qué información meteorológica les interesaba. Creo que la solución desarrollada es capaz de aprovechar al máximo el potencial del protocolo MQTT, logrando no solo proponer una comunicación eficaz, sino también eficiente, limitando todas las comunicaciones innecesarias.

Durante la fase de diseño esperaba encontrar más dificultades, pero debo admitir que el lenguaje Python fue muy fácil de aprender y usar, a pesar de no conocerlo en profundidad previamente.

6.2 Posibles trabajos futuros

Actualmente el proxy no proporciona soporte completo a todas las colecciones de datos disponibles en la API de OpenWeather. Por lo tanto, un posible trabajo futuro podría ser la ampliación del número de colecciones de datos compatibles.

En general, disponiendo de una clave de pago, se podrá tener el acceso a todas las colecciones de datos y, por lo tanto, llegar a disponer de un soporte completo para la API de OpenWeather.

En particular, una API muy interesante es la que se llama “*Weather Station*”.



Como sabemos un elemento imprescindible para el funcionamiento de la API son las estaciones meteorológicas repartidas por todo el planeta con la tarea de recopilar datos e informaciones sobre el clima. Pues con esta API es posible comunicar datos recopilados por una estación meteorológica propia. Por lo tanto, poseyendo la tecnología adecuada (sensores de temperatura, presión, humedad, etc.) el proxy podría actuar como “puente” entre una estación meteorológica y los servidores centrales de OpenWeather.

Otro aspecto que se puede implementar en futuro es la posibilidad de hacer peticiones por nombre de ciudad, y también con aquellas colecciones de datos que no la soportan (*solar_radiation* y *air_pollution*). Para este propósito se podría utilizar un API llamada **Geocoding API**. Gracias a este API es posible realizar una petición REST en la cual, pasando como parámetro un nombre de ciudad, se recibe como respuesta un objeto JSON que contiene el nombre de esa ciudad en todos los idiomas soportados por el API, así como la posición geográfica de esta ciudad (latitud y longitud). Por lo tanto, un cliente podría realizar una petición sobre calidad del aire pasando como parámetro el nombre de ciudad, y luego el proxy se encarga de convertir el nombre de ciudad en datos de latitud y longitud (usando *geocoding API*) para luego realizar la petición a la API *air_pollution*. De esta manera, el proxy ampliaría las funcionalidades de la API.

6.3 Ejemplo de uso posible

En el contexto del internet de las cosas existen muchos dispositivos que pueden necesitar datos meteorológicos para realizar o planificar determinadas tareas.

Imaginamos el caso de encontrarnos en un contexto rural, donde una empresa se ocupa de la producción de un determinado tipo de cultivo, y por lo tanto dispone de un sistema de riego.



Ilustración 52 – Sistema de riego en un entorno rural.

Hay un elemento que se encarga de gestionar todo el sistema de riego, programando la activación y la parada de los aspersores en determinados intervalos de tiempo definidos por el programador. En caso de lluvia, obviamente no tiene sentido que se active el sistema de riego, especialmente considerando el tema climático (desperdicio de agua). Sería muy interesante que este elemento, mediante comunicación con el proxy, pudiera solicitar los datos de previsión meteorológica incluyendo la *probabilidad de lluvia*, para que pueda planificar el riego de manera eficiente, por ejemplo:

- Dejando pasar un cierto número de horas desde la última lluvia antes de activar los aspersores.
- No activando los aspersores si está prevista lluvia en las próximas horas. Imaginamos que los aspersores se han programado por activarse a las 22:00, pero está prevista lluvia a las 23:00. No tiene sentido que se activen una hora antes de la lluvia y, por lo tanto, se desactivan, evitando así el desperdicio de agua.



Referencias

- [0] Centro de Cooperación al Desarrollo Universitat Politècnica de València «riunet.upv.es» (2018) [En línea]. *Los objetivos de Desarrollo Sostenible. Claves para una universidad en constante cambio*. [Ultimo acceso: 28/07/2021] https://riunet.upv.es/bitstream/handle/10251/159792/CCD-LibroODS_16Dic2020.pdf?isAllowed=y&sequence=3
- [1] Colaboradores de Wikipedia, «Wikipedia, La enciclopedia libre» (agosto 2019) [En línea]. *Transparencia de red*. [Ultimo acceso: 28/07/2021] https://es.wikipedia.org/wiki/Transparencia_de_red
- [2] Dave Evans, «Cisco» (abril 2011) [En línea]. *Internet of Things. La próxima evolución lo está cambiando todo*. [Ultimo acceso: 22/07/2021] https://www.cisco.com/c/dam/global/es_es/assets/executives/pdf/Internet_of_Things_IoT_IBSG_0411FINAL.pdf
- [3] Miriam Barchilón, «La Vanguardia» (marzo 2019) [En línea]. *Internet de las cosas: cuando todo está conectado* [Ultimo acceso: 23/07/2021] <https://www.lavanguardia.com/vida/junior-report/20190301/46752655177/internet-cosas-dispositivos-conectados-iot.html>
- [4] Pallavi Sethi y Smruti R. Sarangi, «Hindawi» (enero 2017) [En línea]. *Internet of Things: Architectures, Protocols, and Applications* [Ultimo acceso: 23/07/2021] <https://www.hindawi.com/journals/jece/2017/9324035/>
- [5] Aron Semple, Kepware -eFalcon «Aadeca» (septiembre 2016) [En línea]. *Protocolos IoT para considerar* [Ultimo acceso: 24/07/2021] https://editores-srl.com.ar/sites/default/files/aa2_semle_protocolos_ilot.pdf
- [6] Colaboradores de Wikipedia «Wikipedia, La enciclopedia libre» (julio 2021) [En línea]. *Transferencia de Estado Representacional* [Ultimo acceso: 24/07/2021] https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional
- [7] Colaboradores de OpenWeather «OpenWeatherMap» (julio 2021) [En línea]. *Current weather data* [Ultimo acceso: 24/07/2021] <https://openweathermap.org/current>
- [8] Colaboradores de Wikipedia «Wikipedia, La enciclopedia libre» (enero 2020) [En línea]. *Protocolo sin estado* [Ultimo acceso: 28/07/2021] https://es.wikipedia.org/wiki/Protocolo_sin_estado
- [9] Colaboradores de Wikipedia «Wikipedia, La enciclopedia libre» (julio 2021) [En línea]. *M2M* [Ultimo acceso: 28/07/2021] <https://es.wikipedia.org/wiki/M2M>
- [10] Colaboradores de Wikipedia «Wikipedia, La enciclopedia libre» (julio 2019) [En línea]. *OASIS (organización)* [Ultimo acceso: 28/07/2021] <https://es.wikipedia.org/wiki/M2M>

- [11] Colaboradores de MQTT.org «MQTT.org» (julio 2021) [En línea]. *MQTT: el estándar para mensajería de IoT* [Ultimo acceso: 25/07/2021] <https://mqtt.org/>
- [12] Rubén Andrés «Computer Hoy» (octubre 2019) [En línea]. *Así ha conseguido la Raspberry Pi convertirse en un éxito de masas* [Ultimo acceso: 28/07/2021] <https://computerhoy.com/reportajes/tecnologia/raspberry-pi-exito-515523>
- [13] Ben Davis «mvorganizing.com» (marzo 2021) [En línea]. *What is Internet of things in simple words?* [Ultimo acceso: 28/07/2021] <https://www.mvorganizing.org/what-is-internet-of-things-in-simple-words/>
- [14] Colaboradores de Eclipse «Eclipse Foundation» (julio 2021) [En línea]. *Paho - Python Client – documentation Contents* [Ultimo acceso: 29/07/2021] <https://www.eclipse.org/paho/index.php?page=clients/python/docs/index.php>
- [15] Colaboradores de Openweather «Openweathermap.org» (julio 2021) [En línea]. *Current weather data* [Ultimo acceso: 02/08/2021] <https://openweathermap.org/current>
- [16] Colaboradores de Openweather «Openweathermap.org» (julio 2021) [En línea]. *Multilingual support* [Ultimo acceso: 02/08/2021] <https://openweathermap.org/current#multi>
- [17] Colaboradores de Openweather «Openweathermap.org» (julio 2021) [En línea]. *Hourly forecast* [Ultimo acceso: 03/08/2021] <https://openweathermap.org/api/hourly-forecast>
- [18] Colaboradores de Openweather «Openweathermap.org» (julio 2021) [En línea]. *Daily forecast 16 Days* [Ultimo acceso: 03/08/2021] <https://openweathermap.org/forecast16>
- [19] Colaboradores de Openweather «Openweathermap.org» (julio 2021) [En línea]. *Solar Radiation API* [Ultimo acceso: 03/08/2021] <https://openweathermap.org/api/solar-radiation>
- [20] Colaboradores de Agencia Anadaluz de la eneregia «agenciaandaluzadelaenergia.es» (julio 2021) [En línea]. *Glosario Radiación* [Ultimo acceso: 03/08/2021] <https://www.agenciaandaluzadelaenergia.es/Radiacion/glosario.php>
- [21] Colaboradores de Openweather «Openweathermap.org» (julio 2021) [En línea]. *Air Pollution API* [Ultimo acceso: 03/08/2021] <https://openweathermap.org/api/air-pollution>
- [22] Roger Light «Mosquitto.org» (julio 2021) [En línea]. *Mosquitto.conf man page* [Ultimo acceso: 08/08/2021] <https://mosquitto.org/man/mosquitto-conf-5.html>
- [23] Colaboradores de Wikipedia «Wikipedia, La enciclopedia libre» (octubre 2020) [En línea]. *Índice de paquetes de Python* [Ultimo acceso: 10/08/2021] https://es.wikipedia.org/wiki/%C3%8Dndice_de_paquetes_de_Python

[24] Colaboradores de Wikipedia «Wikipedia, La enciclopedia libre» (febrero 2021) [En línea]. *Socket de Internet* [Ultimo acceso: 28/08/2021] [https://es.wikipedia.org/wiki/Socket de Internet](https://es.wikipedia.org/wiki/Socket_de_Internet)