UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

# OPTIMIZACIÓN DE MQTT EN REDES LORAWAN

**Autor**

Ricardo Cortés Núñez

**Director**

Jorge Navarro Ortiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones

—

Granada, septiembre de 2021

# OPTIMIZACIÓN DE MQTT EN REDES LORAWAN

**Autor**

Ricardo Cortés Núñez

**Director**

Jorge Navarro Ortiz

# Optimización de MQTT en redes LoRaWAN

Ricardo Cortés Núñez

**Palabras clave:** Internet de las Cosas, MQTT, SDN, OpenFlow, Controlador, Bróker, Ryu, Mininet.

**Resumen:**

En la sociedad actual están creciendo de manera exponencial los dispositivos conectados a Internet que intercambian información con otros dispositivos. Este concepto básico es en el que se sustenta el Internet de las Cosas (IoT), un concepto que cada vez es más conocido por la población.

A nivel de aplicación, nivel que actúa como interfaz entre el usuario y el dispositivo con un protocolo de IoT determinado, existen distintos protocolos como Advanced Message Queuing Protocol (AMQP), Protocolo de aplicación restringida (CoAP), Servicio de distribución de datos (DDS)…, pero en el que se va a focalizar este trabajo es MQTT, un protocolo basado en el paradigma publicador / suscriptor y que utiliza un intermediario o bróker para intercambiar la información entre los dispositivos.

Por otro lado, se está avanzando cada vez más hacia la virtualización, haciendo desaparecer los límites físicos y reduciendo el uso de *hardware* de manera notoria. Específicamente, en el ámbito de las redes, aparecen las Redes Diseñadas por Software (SDN). Este tipo de redes separa el plano de control (*software*) del plano de datos (*hardware*), mejorando la configuración de la red, maximizando su uso y haciéndolas más escalables.

En concreto, en este trabajo confluyen los dos conceptos anteriormente explicados creando un escenario MQTT utilizando las ventajas de las SDN. En este escenario se busca eliminar la imagen del bróker, pero siendo totalmente transparente a los usuarios finales. Esto ayuda a tener un mejor control sobre el tráfico de la red y elimina un punto de fallo crítico de los escenarios MQTT.

Esta solución se consigue a través del controlador de la red SDN, el "cerebro" de este tipo de redes. El controlador por el que se opta es Ryu y con la programación de una de sus aplicaciones, según el análisis previo de los mensajes MQTT, se consigue emular al bróker.

# MQTT optimization in LoRaWAN networks

Ricardo Cortés Núñez

**Keywords:** Internet de las Cosas, MQTT, SDN, OpenFlow, Controlador, Bróker, Ryu, Mininet.

**Abstract**

In today's society, devices connected to the Internet and exchange information with other devices are growing exponentially. This basic concept is what the Internet of Things (IoT) is based on, a concept that is increasingly known by the population.

At the application level, a level that acts as an interface between the user and the device with a certain IoT protocol, there are different protocols such as Advanced Message Queuing Protocol (AMQP), Constrained Application Protocol (CoAP), Data Distribution Service (DDS) ..., but the one that this work will focus on is MQTT, a protocol based on the publisher / subscriber paradigm and that uses an intermediary or broker to exchange information between the devices.

On the other hand, more and more progress is being made towards virtualization, making physical limits disappear and reducing the use of hardware in a noticeable way. Specifically, in the field of networks, Software Designed Networks (SDN) appear. This type of network separates the control plane (software) from the data plane (hardware), improving the configuration of the network, maximizing its use and making it more scalable.

Specifically, in this work the two concepts explained above converge creating an MQTT scenario using the advantages of SDN. In this scenario it seeks to eliminate the image of the broker, but being totally transparent to the end users. This helps to have better control over network traffic and eliminates a critical point of failure from MQTT scenarios.

This solution is achieved through the controller of the SDN network, the "brain" of this type of network. The controller chosen is Ryu and with the programming of one of its applications, according to the previous analysis of the MQTT messages, it is possible to emulate the broker.

Yo, **Ricardo Cortés Núñez**, alumno de la titulación INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI XXXXXXXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Ricardo Cortés Núñez

Granada, septiembre de 2021.

**Jorge Navarro Ortiz**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

**Informa:**

Que el presente trabajo, titulado ***Optimización de MQTT en redes LoRaWAN***, ha sido realizado bajo su supervisión por **Ricardo Cortés Núñez**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada, septiembre de 2021.

**El director:**

**Jorge Navarro Ortiz**

# Agradecimientos

Estas líneas son las últimas que escribo como alumno de este grado del que me llevo momentos buenos y malos, pero sobre todo siendo una mejor persona y habiéndome formado en lo que me gusta. Echando la vista atrás no ha sido fácil el camino, pero una vez acabado resulta gratificante tanto a nivel personal como académico. En el trayecto han sido numerosas las personas que me han ayudado de alguna u otra manera y por ello me gustaría dedicarles unas palabras.

En primer lugar, a los docentes y a todos los compañeros que me llevo, sin ellos no sería posible estar aquí. En especial a Jorge, mi tutor durante este trabajo, que me ha echado una mano siempre que lo he necesitado y ha sabido orientarme en el transcurso del mismo. Por eso, y por ser tan amable y buen profesor, gracias.

Me gustaría continuar con mi familia, que siempre han estado pendientes de cómo iba durante el transcurso del trabajo y el grado en general, además de mostrarme constantemente un apoyo incondicional. A mi madre, mi padre y mi hermana que han vivido mejor que nadie todos mis altibajos desde que comencé el grado y han sentido como suyos todos los logros que he ido consiguiendo. Ellos son los tres pilares fundamentales de todo lo que yo logre a lo largo de mi vida.

También a mis amigos, de los que algunos de ellos conocen el mundo universitario, incluso en específico el de este grado de telecomunicaciones. Han sido un apoyo muy importante y con los que me he podido evadir de la rutina cuando lo he necesitado.

Ya para acabar, pero no menos importante, agradecer a la persona que mejor me ha sabido llevar durante este último año. En las duras y en las menos duras siempre ha estado a mi lado sin importarle nada más. Sin ti este trabajo no habría sido lo mismo. Te quiero Alba.

# Índice general

Índice de figuras ........ 21

Índice de tablas ........ 25

Glosario de siglas ........ 27

1. Introducción ........ 29

1.1. Motivación ........ 29

1.2. Contexto ........ 29

1.2.1. Internet of Things ........ 29

1.2.2. Software Defined Networking ........ 35

1.3. Estructura de la memoria ........ 38

2. Objetivos ........ 41

2.1. Estudio del protocolo MQTT ........ 41

2.2. Desarrollo de la red SDN ........ 41

2.3. Eliminación del broker MQTT ........ 41

3. Planificación y costes ........ 43

3.1. Definición de fases ........ 43

3.2. Planificación temporal ........ 44

3.3. Recursos y coste asociado ........ 44

4. Estado del arte ........ 47

4.1. MQTT-SN ........ 47

4.2. DM-MQTT ........ 50

4.3. MQTT-ST ........ 51

4.4. EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications ........ 53

5. Herramientas ........ 55

5.1. Mininet [16] ........ 55

5.1.1. Ventajas y desventajas ... 55

5.1.2. Topologías ... 56

5.1.3. Comandos ... 63

5.2. Ryu [22] ... 64

5.2.1. Instalación ... 65

5.2.2. Componentes de Ryu ... 65

5.3. Wireshark [27] ... 67

6. Fundamentos teóricos ... 69

6.1. MQTT ... 69

6.1.1. Modelo ... 69

6.1.2. Formato paquetes ... 70

6.1.3. Calidad de servicio ... 86

6.2. OpenFlow ... 88

6.2.1. Funcionamiento ... 88

6.2.2. Tablas ... 90

6.2.3. Matching ... 91

6.2.4. Mensajes ... 91

7. Desarrollo práctico ... 93

7.1. Estudio de MQTT ... 93

7.1.1. Diseño de la red ... 93

7.1.2. Implementación ... 94

7.1.3. Estudio con Wireshark ... 96

7.2. Eliminación del bróker ... 98

7.2.1. Análisis de los mensajes ... 99

7.2.2. Creación de mensajes de red. ... 103

7.2.3. Creación de mensajes MQTT ... 108

8. Resultados ... 117

8.1. Funcionamiento con varios publicadores ........................................................ 117

8.2. Funcionamiento con varios suscriptores.......................................................... 118

8.3. Funcionamiento con varios *topics*................................................................. 120

8.4. Comparativa de tiempos de envío .................................................................. 121

9. Conclusiones........................................................................................................ 125

9.1. Logros conseguidos ......................................................................................... 125

9.2. Trabajos futuros............................................................................................... 126

9.3. Valoración personal.......................................................................................... 126

Bibliografía ................................................................................................................ 129

A. Código ................................................................................................................. 133

a. Definición de la topología de red .................................................................... 133

b. Función _handle_arp ........................................................................................ 135

c. Función generate_tcp_ack y _handle_tcp........................................................ 136

d. Función _handle_mqtt ...................................................................................... 138

e. Función send_packet ........................................................................................ 142

# Índice de figuras

**Figura 1.1.** Usuarios de Internet en el mundo por regiones, 25 de marzo de 2017. Fuente: Internet World Stats [10]. 30
**Figura 1.2.** Tendencias emergentes de IoT y oportunidades potenciales [10]. 31
**Figura 1.3.** Requisitos generales para una arquitectura de referencia IoT [10]. 33
**Figura 1.4.** Modelo de referencia SDN [28]. 36

**Figura 3.1.** Diagrama de Gantt de la planificación temporal a priori. 44

**Figura 4.1.** Arquitectura de MQTT-SN [25]. 48
**Figura 4.2.** Gateways transparente y agregado [25]. 49
**Figura 4.3.** Arquitectura de DM-MQTT [19]. 50
**Figura 4.4.** Cambios en el *delay* de transferencia de datos dependiendo del número de muestras de datos [19]. 51
**Figura 4.5.** Estructura de EMMA [21]. 54

**Figura 5.1.** Esquemático de cómo funciona Mininet [17]. 55
**Figura 5.2.** Esquema de la topología *minimal*. 57
**Figura 5.3.** Topología *minimal* creada con Mininet. 58
**Figura 5.4.** Esquema de la topología *single*. 58
**Figura 5.5.** Topología *single* con 4 *hosts* creada con Mininet. 59
**Figura 5.6.** Esquema de la topología *linear*. 59
**Figura 5.7.** Topología *linear* con 4 *switches* creada con Mininet. 60
**Figura 5.8**. Esquema de la topología *tree*. 60
**Figura 5.9.** Topología *tree* con 2 niveles y 3 *hosts* por *switch* finales creada con Mininet. 61
**Figura 5.10.** Creación de la topología personalizada de ejemplo con Mininet. 62
**Figura 5.11.** Esquema de la topología personalizada de ejemplo. 63
**Figura 5.12.** Introducción de comando `help`. 63
**Figura 5.13.** Logo de Ryu [22]. 64
**Figura 5.14.** Controlador Ryu en un ambiente SDN [5]. 64

**Figura 5.15.** Interfaz gráfica de Wireshark. ........ 68

**Figura 6.1.** *Fixed header* o cabecera fija de paquete MQTT. ........ 70
**Figura 6.2.** Identificador de mensaje MSB y LSB. ........ 74
**Figura 6.3.** Cabecera fija de un mensaje CONNECT. ........ 75
**Figura 6.4.** Cabecera variable de un mensaje CONNECT. ........ 75
**Figura 6.5.** Cabecera fija de un mensaje CONNACK. ........ 76
**Figura 6.6.** Cabecera variable de un mensaje CONNACK. ........ 77
**Figura 6.7.** Cabecera fija de un mensaje PUBLISH. ........ 77
**Figura 6.8.** Cabecera variable de un mensaje PUBLISH. ........ 78
**Figura 6.9.** Cabecera fija de un mensaje PUBACK. ........ 78
**Figura 6.10.** Cabecera variable de un mensaje PUBACK. ........ 78
**Figura 6.11.** Cabecera fija de un mensaje PUBREC. ........ 79
**Figura 6.12.** Cabecera variable de un mensaje PUBREC. ........ 79
**Figura 6.13.** Cabecera fija de un mensaje PUBREL. ........ 80
**Figura 6.14.** Cabecera fija de un mensaje PUBCOMP. ........ 80
**Figura 6.15.** Cabecera fija de un mensaje SUBSCRIBE. ........ 81
**Figura 6.16.** Cabecera variable de un mensaje SUBSCRIBE. ........ 81
**Figura 6.17.** *Payload* de un mensaje SUBSCRIBE. ........ 82
**Figura 6.18.** Cabecera fija de un mensaje SUBACK. ........ 82
**Figura 6.19.** Cabecera variable de un mensaje SUBACK. ........ 82
**Figura 6.20.** *Payload* de un mensaje SUBACK. ........ 83
**Figura 6.21.** Cabecera fija de un mensaje UNSUBSCRIBE. ........ 83
**Figura 6.22.** Cabecera variable de un mensaje UNSUBSCRIBE. ........ 83
**Figura 6.23**. *Payload* de un mensaje UNSUBSCRIBE. ........ 84
**Figura 6.24.** Cabecera fija de un mensaje UNSUBACK. ........ 84
**Figura 6.25.** Cabecera variable de un mensaje UNSUBACK. ........ 85
**Figura 6.26.** Cabecera fija de un mensaje PINGREQ. ........ 85
**Figura 6.27.** Cabecera fija de un mensaje PINGRESP. ........ 86
**Figura 6.28.** Cabecera fija de un mensaje DISCONNECT. ........ 86
**Figura 6.29.** Arquitectura de OpenFlow [29]. ........ 89
**Figura 6.30.** OpenFlow *pipeline* [30]. ........ 90
**Figura 6.31.** Componentes de la tabla de flujo. ........ 90

**Figura 6.32.** Diagrama de flujo de un paquete dentro del OFS [30]........................................ 91

**Figura 7.1.** Topología de red. ................................................................................ 93
**Figura 7.2.** Puesta en marcha de la red.................................................................. 94
**Figura 7.3.** Topología de red con reparto de roles (*broker*, suscriptor y publicador)........... 95
**Figura 7.4.** Escenario MQTT en funcionamiento. .................................................... 96
**Figura 7.5.** Mensajes recibidos y enviados por el suscriptor con QoS=0. ........................... 97
**Figura 7.6.** Mensajes recibidos y enviados por el publicador con QoS=0........................... 97
**Figura 7.7.** Mensajes recibidos y enviados por el suscriptor con QoS=1. ........................... 97
**Figura 7.8.** Mensajes recibidos y enviados por el publicador con QoS=1........................... 98
**Figura 7.9.** Mensajes recibidos y enviados por el suscriptor con QoS=2 ............................ 98
**Figura 7.10.** Mensajes recibidos y enviados por el publicador con QoS=2.......................... 98
**Figura 7.11.** Conjunto de mensajes ARP capturados en Wireshark. ................................ 100
**Figura 7.12.** Formato de paquete ARP.................................................................. 100
**Figura 7.13.** Esquema del “3-way handshake”........................................................ 101
**Figura 7.14.** Conjunto de mensajes TCP capturados en Wireshark................................. 102
**Figura 7.15.** Formato de paquete TCP. ................................................................. 102
**Figura 7.16.** Conjunto de mensajes MQTT capturados en Wireshark.............................. 103
**Figura 7.17.** Mensajes mostrados en la terminal cuando se recibe un ARP Request y se responde con un ARP Reply.................................................................................. 104
**Figura 7.18.** Mensaje ARP Reply creado por el controlador y capturado con Wireshark... 104
**Figura 7.19.** Mensajes mostrados en la terminal cuando se recibe un TCP SYN y se responde con un TCP SYN+ACK. ........................................................................................ 105
**Figura 7.20.** Mensaje TCP SYN ACK creado por el controlador y capturado con Wireshark. ........................................................................................................... 106
**Figura 7.21.** Mensajes mostrados en la terminal cuando se recibe un TCP FIN por parte del publicador y se responde con un TCP FIN+ACK. ........................................................ 107
**Figura 7.22.** Mensajes mostrados en la terminal cuando se recibe un TCP FIN por parte del suscriptor y se responde con un TCP FIN+ACK............................................................ 107
**Figura 7.23.** Mensaje TCP FIN ACK creado por el controlador y capturado con Wireshark. ........................................................................................................... 107
**Figura 7.24.** Mensajes mostrados en la terminal cuando se recibe un CONNECT y se responde con un CONNACK. ........................................................................................ 109

**Figura 7.25.** Mensaje CONNACK creado por el controlador y capturado con Wireshark.. 109
**Figura 7.26.** Mensajes mostrados en la terminal cuando se recibe un SUBSCRIBE y se responde con un SUBACK. 110
**Figura 7.27.** Mensaje SUBACK creado por el controlador y capturado con Wireshark. .... 111
**Figura 7.28.** Mensajes mostrados en la terminal cuando se recibe un PINGREQ y se responde con un PINGRESP. 112
**Figura 7.29.** Mensaje PINGRESP creado por el controlador y capturado con Wireshark. . 112
**Figura 7.30.** Mensajes mostrados en la terminal cuando se recibe un PUBLISH por parte del publicador y es reenviado al suscriptor. 113
**Figura 7.31.** Mensaje PUBLISH recibido por el controlador de parte del publicador y capturado con Wireshark. 114
**Figura 7.32.** Mensaje PUBLISH reenviado por el controlador al suscriptor y capturado con Wireshark. 114
**Figura 7.33**. Mensajes mostrados en la terminal cuando se recibe un DISCONNECT por parte del suscriptor y se responde con FIN+ACK. 115
**Figura 7.34.** Mensaje FIN+ACK enviado por el controlador al suscriptor en respuesta a un DISCONNECT y capturado con Wireshark. 115

**Figura 8.1.** Topología con varios publicadores. 117
**Figura 8.2.** Escenario con varios publicadores y *broker*. 118
**Figura 8.3.** Escenario con varios publicadores y sin *broker*. 118
**Figura 8.4.** Topología con varios suscriptores. 119
**Figura 8.5.** Escenario con varios suscriptores y *broker*. 119
**Figura 8.6.** Escenario con varios suscriptores y sin *broker*. 120
**Figura 8.7.** Topología con suscriptores a *topics* diferentes. 120
**Figura 8.8.** Escenario con varios *topics* y *broker*. 121
**Figura 8.9.** Escenario con varios *topics* y sin *broker*. 121
**Figura 8.10.** Publicador a la izquierda y suscriptor a la derecha con *broker*. 122
**Figura 8.11.** Publicador a la izquierda y suscriptor a la derecha sin *broker*. 122

# Índice de tablas

**Tabla 3.1.** Coste total del trabajo. 45

**Tabla 5.1.** Algunas opciones que ofrece Mininet. 57
**Tabla 5.2.** Funciones/Clases/Métodos/Variables para topologías personalizadas. 61
**Tabla 5.3.** Algunos comandos de Mininet y su función. 63

**Tabla 6.1.** Tipos de mensajes MQTT. 71
**Tabla 6.2.** Valor de códigos de retorno. 73
**Tabla 6.3.** Flujo del protocolo de nivel QoS 0. 87
**Tabla 6.4.** Flujo del protocolo de nivel QoS 1. 87
**Tabla 6.5.** Flujo del protocolo de nivel QoS 2. 88

**Tabla 8.1.** Tiempos de envío mensaje PUBLISH con bróker. 122
**Tabla 8.2.** Tiempos de envío mensaje PUBLISH sin bróker. 122

# Glosario de siglas

| | |
|---|---|
| API | Application Programming Interface |
| ARP | Address Resolution Protocol |
| BPDU | Bridge Protocol Data Unit |
| CPU | Central Processing Unit |
| DM-MQTT | Direct Multicast Message Queuing Telemetry Transport |
| DUP | Duplicate Message Flag |
| IoT | Internet of Things |
| IP | Internet Protocol |
| ISO | International Organization for Standardization |
| LAN | Local Area Network |
| LSB | Least Significant Bit |
| M2M | Machine to Machine |
| MAC | Media Access Control |
| MQTT | Message Queuing Telemetry Transport |
| MQTT-SN | Message Queuing Telemetry Transport for Sensor Network |
| MASB | Most Significant Bit |
| NAT | Network Address Translation |
| OFP | OpenFlow Protocol |
| OFS | OpenFlow Switch |
| ONF | Open Networking Foundation |
| OSI | Open System Interconnection |
| QoS | Quality of Service |
| RAM | Random Access Memory |
| RTT | Round Trip Time |
| SSL | Secure Sockets Layer |

| | |
|---|---|
| STP | Spanning Tree Protocol |
| TCP | Transport Control Protocol |
| UDP | User Datagram Protocol |
| UTF | Unicode Transformation Format |
| WSNs | Wireless Sensor Networks |

# 1. Introducción

El primer apartado de este trabajo tiene como objetivo dar un contexto al mismo y presentarlo de manera breve, además de resumir un poco de lo que se trata en cada apartado de la memoria.

## 1.1. Motivación

Con este trabajo se busca mejorar el protocolo MQTT, un protocolo que utiliza el paradigma publicador / suscriptor y líder para conectarse a dispositivos IoT. Este protocolo utiliza un *broker* para almacenar los mensajes, por si los clientes no están conectados en el momento en el que el publicador manda datos.

La idea principal es conseguir diseñar e implementar un escenario MQTT básico, pero emulando el *broker* dentro del controlador de una red SDN.

## 1.2. Contexto

El *Internet of Things* y las *Software Defined Network* son los dos conceptos más importantes que engloban el proyecto.

Debido a su sencillez y ligereza el protocolo MQTT se ha convertido en uno de los más importantes si se habla de IoT. Esto es así debido a que los dispositivos IoT presentan limitaciones de potencia, consumo y ancho de banda.

Por otro lado, las redes SDN difieren de las redes tradicionales, que utilizan dispositivos de hardware dedicados para controlar el tráfico de la red, en que se puede crear y controlar una red virtual mediante software. Esta principal característica ayuda a que desaparezcan muchas limitaciones *hardware* de lo que se puede aprovechar el protocolo MQTT, obteniendo comunicaciones mejor controladas de manera rápida y fácil.

### 1.2.1. Internet of Things

*Internet of Things* es el concepto de conectar cualquier dispositivo a Internet y a otros dispositivos. IoT es una red gigante de cosas y personas conectadas, todas las cuales recopilan y comparten datos sobre la forma en que se utilizan y sobre el entrono que las rodea.

Eso incluye una gran cantidad de objetos de todas las formas y tamaños, desde microondas inteligentes, que cocinan automáticamente su comida durante el tiempo correcto, hasta automóviles autónomos, cuyos complejos sensores detectan objetos en su camino.

*1.2. Contexto*

### 1.2.1.1. Importancia

Al interpretar las estadísticas de los usuarios de Internet no es difícil decir que la mayoría de los objetos que nos rodean estarán conectados a Internet en diferentes formas. Internet, redes de sensores inalámbricos, Wi-Fi, 4G-LTE, IPv6, EPC, NFC, sensores, servicios de datos telefónicos y tecnologías RFID son tecnologías precursoras para hacer realidad esta idea. Para dar un ejemplo general, la cantidad de equipos que están conectados entre sí comenzó a superar la cantidad real de personas en la tierra solo en 2011 [11]. Según International Data Corporation (IDC), la cantidad de dispositivos interconectados es de 9 mil millones excluyendo tabletas, teléfonos inteligentes y computadoras por ahora. Según Information Handling Services (IHS), este número es de 17,6 mil millones, incluidos teléfonos inteligentes, tabletas y computadoras.

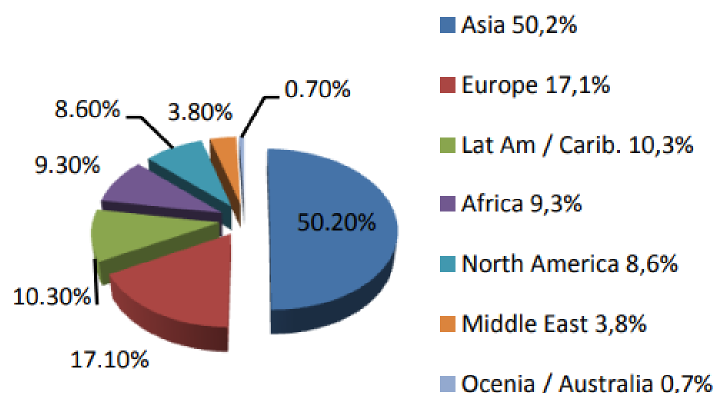


**Figura 1.1.** Usuarios de Internet en el mundo por regiones, 25 de marzo de 2017. Fuente: Internet World Stats [10].

Los dispositivos interconectados dependen principalmente de los sensores para detectar cambios en el entorno y de los actuadores para indicar, rastrear, identificar, controlar, automatizar y monitorear algunas acciones como abrir una puerta, activar un motor, apagar una lámpara, subir un ascensor, advertir a un usuario, bajar la velocidad de un coche, etc. La Figura 3 muestra las tendencias emergentes y las oportunidades potenciales de IoT. La computación ubicua y la computación omnipresente tienen una relación muy poderosa con la idea de IoT. Sin embargo, una diferencia bastante notable es que la computación ubicua y omnipresente se basa en conectar un solo dispositivo a un centro administrativo de telecomunicaciones, mientras que IoT se basa en conectar muchos dispositivos a un centro administrativo de telecomunicaciones y entre sí, una comunicación entre humano a humano, humano- a-cosas, cosas-a-cosas. Una cosa importante sobre IoT es que la palabra 'cosa' no significa necesariamente solo dispositivos electrónicos como servidores, computadoras, tabletas, teléfonos y teléfonos inteligentes.

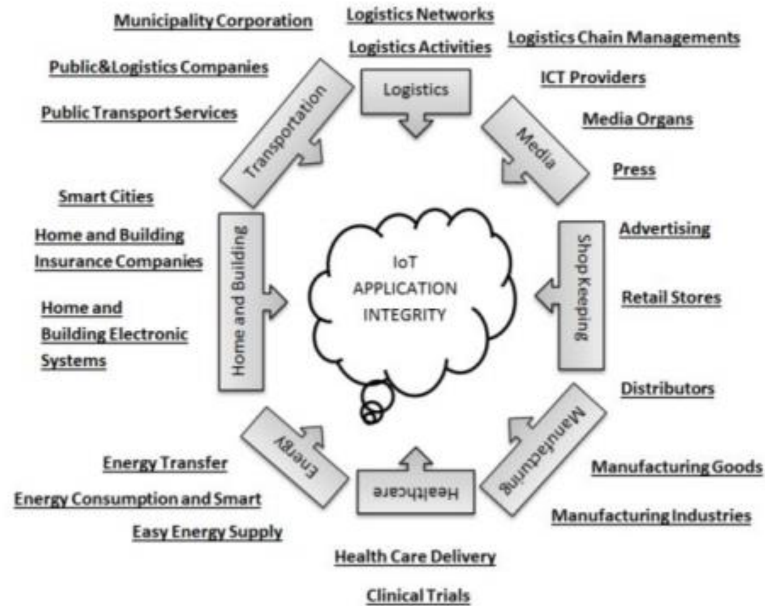


**Figura 1.2.** Tendencias emergentes de IoT y oportunidades potenciales [10].

#### 1.2.1.2. Arquitectura

IoT no tiene un único consenso universal sobre la arquitectura de IoT. Durante las últimas décadas se han presentado diversas propuestas de arquitectura IoT. La arquitectura principal de IoT propone una arquitectura de tres y cinco capas. La arquitectura de tres capas consta de *perception layer*, *network layer* y *applicatoin layer*. La arquitectura de cinco capas agrega *processing layer* y *business layer* a la arquitectura de tres capas. Según Ning y Wang [7] el fenómeno del procesamiento del cerebro humano es muy similar a la tecnología IoT ya que ambos son una especie de sistema inteligente complejo que puede ver, saborear, sentir y controlar cosas, o incluso tomar decisiones.

- ***Coding layer.*** La capa de codificación es la primera capa de IoT que identifica cada objeto de IoT. En esta capa, a cada objeto se le asigna una identificación única que facilita la distinción de los objetos.

- ***Perception layer (Device layer).*** La capa de percepciónes la capa más baja cuya característica principal es detectar los cambios ambientales mediante sensores. La capa de percepción se conoce como los órganos sensoriales de IoT, que perciben y detectan cambios ambientales como temperatura, presión, humedad, radiación, luz, sonido, olor, movilidad, ubicación, velocidad, recolectan datos útiles y convierten estas señales analógicas en señales digitales. Luego, esta información digital se pasa a la siguiente capa, Capa de Red, para su procesamiento posterior. Los dispositivos

IoT se conectan a Internet o entre sí a través de una conexión Ethernet, conexión WiFi, GSM, WiMaX, 3G, pasarela ZigBee, Bluetooth, etc.

- ***Network layer (Communication layer).*** Los dispositivos de IoT están conectados entre sí o con nubes a través de la capa de red. La capa de red necesita algunos protocolos de comunicación como HTTP/HTTPS, MQTT 3.1/3.1.1, Constrained Application Protocol (CoAP), IPv4, IPv6, DDS, etc. para conectar dispositivos. La capa de red transmite información de la capa de percepción a la capa de *middleware* de forma segura.
- ***Middleware layer (The Event Processing and Analytics Layer).*** La capa de *middleware* es una capa de interfaz de software entre la capa física y la capa de aplicación. Esta importante capa desconecta a los usuarios y desarrolladores del conocimiento exacto del heterogéneo conjunto de tecnologías adoptadas por las capas inferiores. Cada dispositivo en las aplicaciones de IoT se conecta y se comunica con solo algunos dispositivos que están involucrados en la misma aplicación. La capa responsable de la gestión de servicios de estos dispositivos es la capa de *middleware* que tiene un enlace a la base de datos. La capa de *middleware* recibe datos de la capa de red y mantiene estos datos en la base de datos. Procesa la información y realiza cálculos ubicuos. Según los resultados del procesamiento, la capa se vuelve automática.
- ***Application layer***. Esta capa superior ofrece diferentes aplicaciones a los usuarios de IoT. Estas aplicaciones pueden ser tan diversas como logística, transporte, hogar y construcción, energía, salud, manufactura, venta minorista, medios, etc.
- ***Business layer.*** La capa empresarial gestiona todo el proceso desde un punto de vista industrial al observar el comportamiento general de las aplicaciones y servicios de IoT. La información recibida de la capa de aplicación se pasa a la inteligencia empresarial para la identificación de beneficios. La evaluación del rendimiento se realiza sobre la salida de cada capa para mejorar los servicios. La capa empresarial es donde las aplicaciones IoT se integran en los procesos empresariales y los sistemas empresariales.

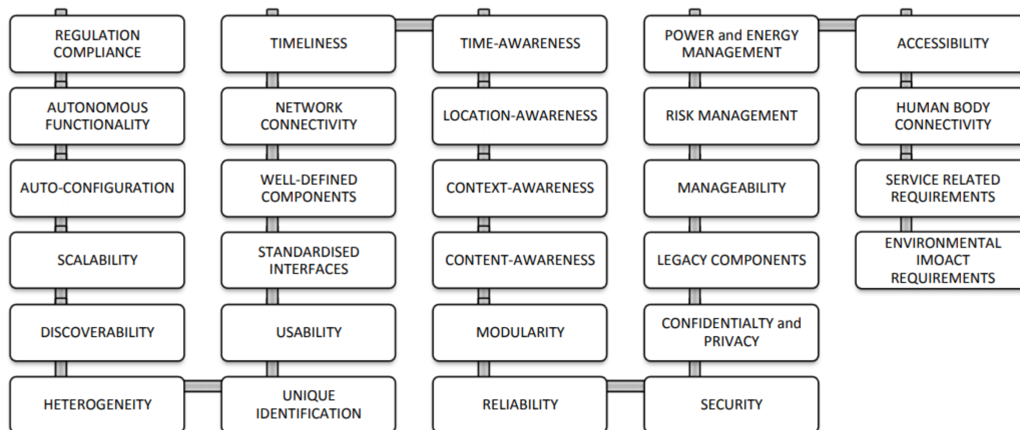


**Figura 1.3.** Requisitos generales para una arquitectura de referencia IoT [10].

#### 1.2.1.3. Desafíos

IoT es un tema en alza tanto por sus ventajas como desventajas. Como ya ocurrió con Internet cuando comenzó a desarrollarse y desplegarse, surgen y surgirán obstáculos para IoT. Por ello los desafíos y dudas más relevantes sobre esta tecnología son los siguientes:

- **Problemas de seguridad y privacidad.** Las empresas y los desarrolladores de tecnología de la información deben ser conscientes del hecho de que cada usuario de IoT en el mundo quiere confiar en la tecnología desde el punto de vista de la seguridad y la privacidad. Cualquier tecnología con una seguridad ineficiente está en el blanco de ataques cibernéticos y robos.

- **Interoperabilidad y estándares.** La interoperabilidad se define como la capacidad de un producto, un servicio o un sistema para interactuar, comunicarse, intercambiar información y funcionar entre sí. La interoperabilidad, la configuración, la designación y la estandarización de IoT aún no están elaboradas. Por lo tanto, la interoperabilidad y los estándares de IoT es un gran campo para los investigadores que están involucrados en los desafíos de IoT.

- **Derechos legales y asuntos regulatorios.** Los aspectos legales, de derechos y regulatorios de los dispositivos de IoT son quizás los desafíos más complicados desde el punto de vista de los gobiernos. A medida que cada país adopte la tecnología IoT, cada país constituirá sus propias reglas y regulaciones. Es necesario crear un sentido común y una sabiduría compartida general para todos los gobiernos.

- **Asuntos relacionados con la economía y el avance.** Según McKinsey Global Institute, IoT tiene un impacto económico potencial total de $ 3.9 trillones a $ 11.1

trillones por año en 2025. ¿Cómo se logrará una distribución equitativa para todas las partes del mundo? porque se cree que IoT debería ser una herramienta para el empoderamiento global sin tener en cuenta la ubicación, la región, el país o el nivel de desarrollo económico del consumidor.

- **Escalabilidad.** La capacidad de IoT de aplicación, estándares y servicios para desarrollar y extender es impredecible por el momento. Se deben realizar más investigaciones de las estimadas en relación con el rendimiento y el costo en respuesta a los cambios en el rendimiento o la demanda.
- **Problemas de control operativo y tolerancia a fallos.** El mundo de las cosas es mucho más vivo y cambiante que el mundo de las computadoras. Además, la operación y el control de los objetos inteligentes basados en IoT no deben considerarse tan simples como los de Internet de las computadoras debido a la complejidad de los objetos inteligentes.
- **Complejidad de software.** Una posible infraestructura necesaria para un entorno inteligente hace que el software de IoT sea bastante complejo hasta un grado inconmensurable.
- **Volúmenes de datos e interpretación de datos.** Es fácil predecir problemas relacionados con estos considerando todos los dispositivos IoT como sensores, actuadores, redes, datos que se están produciendo, etc. Los desarrollos comunes y el estudio mutuo de IoT y Big Data serán el corazón de los volúmenes de datos y los desafíos de interpretación.
- **Necesidades de energía.** Con el fenómeno de IoT aparecen una cantidad ilimitada de dispositivos los cuales también necesitan una cantidad ilimitada de energía. Los dispositivos IoT serán utilizados en entornos donde la carga no es posible. Existen algunas soluciones para superar este problema energético. La primera solución es aumentar la capacidad de la batería, pero la mayoría de los dispositivos IoT están diseñados en tamaño pequeño y deben ser livianos, por lo que no hay espacio adicional para baterías más grandes. La pregunta crítica es: ¿podrán los dispositivos producir la energía que necesitan?

### 1.2.2. Software Defined Networking

Últimamente, SDN se ha convertido en uno de los temas más populares en el ámbito de las TIC. La Open Networking Foundation (ONF) es un consorcio sin fines de lucro dedicado al desarrollo, estandarización y comercialización de SDN. ONF ha proporcionado la definición más explícita y bien recibida de SDN de la siguiente manera:

"Software-Defined Networking (SDN) es una arquitectura de red emergente donde el control de red está desacoplado del reenvío y es directamente programable."

Su singularidad reside en el hecho de que proporciona programabilidad a través del desacoplamiento de los planos de control y de datos. Específicamente, SDN ofrece dispositivos de red programables simples en lugar de hacer que los dispositivos de red sean más complejos como en el caso de las redes activas. Además, SDN propone la separación de los planos de control y de datos en el diseño arquitectónico de la red. Con este diseño, el control de la red se puede realizar por separado en el plano de control sin afectar los flujos de datos. Como tal, la inteligencia de la red puede extraerse de los dispositivos de conmutación y colocarse en los controladores. Al mismo tiempo, los dispositivos de conmutación ahora se pueden controlar externamente mediante software sin inteligencia integrada. El desacoplamiento del plano de control del plano de datos ofrece no solo un entorno programable más simple, sino también una mayor libertad para que el software externo defina el comportamiento de una red.

#### 1.2.2.1. Modelo de referencia

ONF también ha sugerido un modelo de referencia para SDN. Este modelo consta de tres capas, una capa de infraestructura, una capa de control y una capa de aplicación, apiladas unas sobre otras.

La **capa de infraestructura** consta de dispositivos de conmutación (conmutadores, enrutadores, etc.) en el plano de datos. Las funciones de estos dispositivos de conmutación son en su mayoría dobles. Primero, son responsables de recopilar el estado de la red, almacenarlos temporalmente en dispositivos locales y enviarlos a los controladores. El estado de la red puede incluir información como la topología de la red, las estadísticas de tráfico y los usos de la red. En segundo lugar, son responsables de procesar los paquetes según las reglas proporcionadas por un controlador.

La **capa de control** une la capa de aplicación y la capa de infraestructura a través de sus dos interfaces. Para interactuar hacia abajo con la capa de infraestructura, especifica funciones para

que los controladores accedan a las funciones proporcionadas por los dispositivos de conmutación. Las funciones pueden incluir informar el estado de la red e importar reglas de reenvío de paquetes. Para interactuar hacia arriba con la capa de aplicación, proporciona puntos de acceso al servicio en varias formas, por ejemplo, una interfaz de programación de aplicaciones (API). Las aplicaciones SDN pueden acceder a la información de estado de la red informada desde los dispositivos de conmutación a través de esta API, tomar decisiones de ajuste del sistema basadas en esta información y llevar a cabo estas decisiones estableciendo reglas de reenvío de paquetes a los dispositivos de conmutación que utilizan esta API.

La **capa de aplicación** contiene aplicaciones SDN diseñadas para cumplir con los requisitos del usuario. A través de la plataforma programable proporcionada por la capa de control, las aplicaciones SDN pueden acceder y controlar los dispositivos de conmutación en la capa de infraestructura. Los ejemplos de aplicaciones SDN podrían incluir control de acceso dinámico, movilidad y migración fluidas, equilibrio de carga del servidor y virtualización de la red.

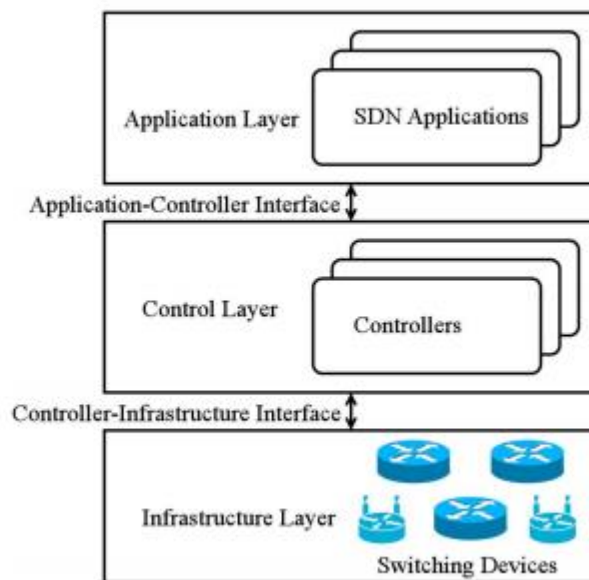


**Figura 1.4.** Modelo de referencia SDN [28].

#### 1.2.2.2. Beneficios

La característica de desacoplamiento del plano de control del plano de datos ofrece beneficios potenciales de configuración mejorada, rendimiento mejorado y fomento de la innovación en la arquitectura y las operaciones de red. Además, con la capacidad de adquirir el estado instantáneo de la red, SDN permite un control centralizado en tiempo real de una red basado tanto en el estado instantáneo de la red como en las políticas definidas por el usuario. El beneficio potencial de SDN se evidencia aún más por el hecho de que SDN ofrece una

plataforma conveniente para la experimentación de nuevas técnicas y fomenta nuevos diseños de red, atribuidos a su programabilidad de red y la capacidad de definir redes virtuales aisladas a través del plano de control.

- **Mejora de la configuración.** En la gestión de redes, la configuración es una de las funciones más importantes. Específicamente, cuando se agregan nuevos equipos a una red existente, se requieren configuraciones adecuadas para lograr un funcionamiento coherente de la red en su conjunto. Sin embargo, debido a la heterogeneidad entre los fabricantes de dispositivos de red y las interfaces de configuración, la configuración de red actual generalmente implica un cierto nivel de procesamiento manual. Este procedimiento de configuración manual es tedioso y propenso a errores. Al mismo tiempo, también se requiere un esfuerzo significativo para solucionar problemas de una red con errores de configuración. SDN ayudará a remediar esta situación en la gestión de la red. En SDN, la unificación del plano de control sobre todo tipo de dispositivos de red, incluidos los conmutadores, enrutadores, traductores de direcciones de red (NAT), cortafuegos y balanceadores de carga, permite configurar dispositivos de red desde un solo punto, automáticamente a través de software de control. Como tal, una red completa puede configurarse mediante programación y optimizarse dinámicamente en función del estado de la red.
- **Mejorando el desempeño.** En las operaciones de red, uno de los objetivos clave es maximizar la utilización de la infraestructura de red invertida. Sin embargo, debido a la coexistencia de diversas tecnologías y partes interesadas en una sola red, se ha considerado difícil optimizar el rendimiento de la red en su conjunto. SDN permite un control centralizado con una vista de red global y un control de retroalimentación con información intercambiada entre diferentes capas en la arquitectura de la red. Como tal, muchos problemas desafiantes de optimización del rendimiento se convertirían en manejable con algoritmos centralizados correctamente diseñados.
- **Fomento de la innovación.** En presencia de una evolución continua de las aplicaciones de red, la red futura debería fomentar la innovación en lugar de intentar predecir con precisión y cumplir perfectamente los requisitos de las aplicaciones futuras. Desafortunadamente, cualquier idea o diseño nuevo enfrenta de inmediato desafíos en la implementación, experimentación y despliegue en las redes existentes. El principal obstáculo surge del hardware propietario ampliamente utilizado en los

componentes de red convencionales, lo que impide la modificación para la experimentación. Además, incluso cuando los experimentos son posibles, a menudo se llevan a cabo en un banco de pruebas simplificado separado. Estos experimentos no aportan suficiente confianza para la adaptación industrial de estas nuevas ideas o diseños de redes. En comparación, SDN fomenta la innovación al proporcionar una plataforma de red programable para implementar, experimentar e implementar nuevas ideas, nuevas aplicaciones y nuevos servicios de generación de ingresos de manera conveniente y flexible. La alta capacidad de configuración de SDN ofrece una clara separación entre las redes virtuales, lo que permite la experimentación en un entorno real. El despliegue progresivo de nuevas ideas se puede realizar a través de una transición fluida de una fase experimental a una fase operativa.

## 1.3. Estructura de la memoria

Aquí se resume de lo que trata cada apartado de la memoria:

- **Apartado 1. Introducción:** Se explica el motivo del trabajo, así como el entorno en el que se engloba el trabajo.
- **Apartado 2. Objetivos:** Se exponen y se desarrollan un poco los objetivos que se busca alcanzar.
- **Apartado 3. Planificación y costes:** Se muestra la planificación temporal a priori y los costes (*hardware*, *software* y humano).
- **Apartado 4. Estado del arte:** Se habla sobre soluciones del protocolo MQTT similares a las de este trabajo.
- **Apartado 5. Herramientas:** Se explican cada una de las herramientas que ayudan a realizar el trabajo.
- **Apartado 6. Desarrollo teórico:** Se explican de manera teórica los dos protocolos más importantes del trabajo, MQTT y OpenFlow.
- **Apartado 7. Desarrollo práctico:** Se explica todo el proceso práctico para llegar al objetivo principal del trabajo.
- **Apartado 8. Resultados:** Se contrastan los resultados obtenidos a lo largo del trabajo.

- **Apartado 9. Conclusiones:** Se exponen las conclusiones obtenidas comparando los resultados con los objetivos iniciales.
- **Bibliografía:** Se especifican las referencias consultadas a lo largo del desarrollo del trabajo.
- **Anexo A. Código:** Contiene fragmentos del código utilizado a los cuales se hace referencia a lo largo del trabajo.

# 2. Objetivos

El objetivo principal que se va a tratar en este proyecto es eliminar el *broker* como una máquina independiente en una red MQTT. Para ello se usará un controlador SDN que lo sustituya y modifique su comportamiento como sea necesario.

Para alcanzar dicho objetivo, primero se realizará un estudio del protocolo MQTT donde se comprenderá el funcionamiento de estas redes y de los paquetes, para a continuación, establecer un escenario MQTT en una red SDN conservando el *broker*. Por último, para que el controlador actúe como *broker* se debe modificar el código.

## 2.1. Estudio del protocolo MQTT

Se pretende estudiar y analizar de forma teórica el protocolo MQTT para montar una red sencilla MQTT donde poder valorar el funcionamiento estándar de dicho protocolo, así como habituarse a la transmisión de paquetes.

Puesto que el escenario final de este proyecto pretende funcionar de igual manera que el primero, es indispensable realizar la fase previa y conocer la transmisión de paquetes y su estructura de antemano.

La forma de evaluar esta parte se realiza probando el envío y el recibo de mensajes en distintos clientes y diferentes QoS.

## 2.2. Desarrollo de la red SDN

En esta parte, el objetivo se evaluará probando el envío y la recepción de mensajes en distintos clientes MQTT dirigidos a un mismo *broker* con la diferencia del uso de un controlador externo gestionable en la red en comparación con el escenario anterior.

Es por ello que los resultados alcanzados deben ser iguales en ambos escenarios puesto que trata escenarios MQTT con *broker*.

## 2.3. Eliminación del broker MQTT

En esta última parte se pretende crear un escenario donde tras basarse en los resultados obtenidos con anterioridad se modifique el código del controlador de forma que actúe como *broker*.

De esta forma, la figura del *broker* es suprimida obteniendo un entorno MQTT con mayor velocidad sin intermediarios.

### *2.3. Eliminación del broker MQTT*

Al darse tal complejidad en la modificación del código del controlador, se divide en las siguientes tareas:

- Creación de tablas de flujos de los *switchs*.
- Parseo de los mensajes MQTT.
- Creación de los mensajes de comunicación y handshake de la red.
- Creación de los mensajes MQTT que enviaría el bróker.
- Funcionamiento correcto con varios publicadores.
- Funcionamiento correcto con varios suscriptores.
- Funcionamiento correcto de los *topic*.

Comprobando el envío y recepción de mensaje MQTT se evalúan los resultados y el cumplimiento de forma correcta del objetivo.

Los resultados deben coincidir con los escenarios anteriores a excepción de las modificaciones donde el controlador reemplaza las funciones del *broker*.

# 3. Planificación y costes

En este apartado se incluye la diferenciación de las distintas fases que han conformado la realización del trabajo, así como el orden cronológico y la duración de cada una de ellas a través de un diagrama de Gantt.

También se incluye una sección final con los recursos *hardware* y *software* utilizados en el trabajo, además de su respectivo coste y el humano (alumno y tutor)

## 3.1. Definición de fases

A continuación, se desarrollan cada una de las fases diferenciadas a lo largo del trabajo en orden cronológico a priori.

### Fase 1 – Preparación y familiarización

En esta primera fase se recoge la información inicial relacionada con el trabajo y se dan los primeros pasos en la creación de redes con Mininet así como el uso del controlador Ryu para familiarizarse con las herramientas. Además, se busca la información teórica relacionada con las herramientas que se van a utilizar y de los protocolos más relevantes del trabajo. Los objetivos principales también se plantean en esta fase.

Los primeros seis apartados de la memoria son los relacionados con esta fase.

### Fase 2 – Desarrollo del proyecto

Esta fase es el núcleo del trabajo. Se puede decir que corresponde con el apartado 7 en el que se desarrolla lo práctico del trabajo. En esta fase se diferencian:

i. Estudio de los paquetes del protocolo MQTT.
ii. Creación de la red SDN.
iii. Creación de la *app* de Ryu que sustituye al *broker*.

### Fase 3 – Análisis de los resultados

Tras acabar con el desarrollo práctico se comparan los resultados antes y después de sustituir al *broker*. Los resultados en ambos casos deben ser próximos ya que lo que se busca es realizar las funciones del *broker* sin el *broker*. En el apartado 8 se desarrolla todo esto.

### Fase 4 – Redacción de la memoria

Por último, se redacta una memoria en la que se refleje todo lo realizado en el trabajo, tanto de forma teórica como práctica. En esta se sigue el orden de las fases, yendo desde lo teórico y más general hasta el caso en particular que recoge el trabajo.

## 3.2. Planificación temporal

Se realiza la planificación temporal a priori, que es la distribución de las distintas fases anteriores a lo largo del tiempo en el que se va a llevar a cabo el trabajo (principios de marzo hasta finales de agosto). Se ha utilizado un diagrama de Gantt para representar esta planificación (ver Figura 3.1).

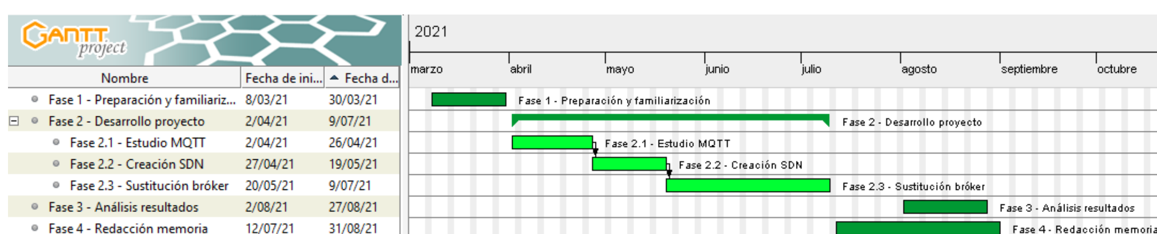


**Figura 3.1.** Diagrama de Gantt de la planificación temporal a priori.

Al finalizar el trabajo, y comparando el tiempo real de realización de cada fase, se puede afirmar que la fase que más tiempo llevo fue la relacionada con el desarrollo práctico (Fase 2). Lo que respecta a la planificación temporal, todas las fases se fueron sucediendo según lo planificado.

## 3.3. Recursos y coste asociado

Los tres tipos de recursos utilizados son de origen *software*, *hardware* y humano.

De recursos *software* se utiliza VirtualBox, Ryu, Mininet, Wireshark y Mosquitto. Todos ellos son gratuitos por lo que no han supuesto ningún coste.

Como *hardware* se utiliza únicamente un PC. Se ha utilizado un portátil ASUS modelo X556UJ valorado en unos 700 €. Este tiene una vida útil de 60 meses y se ha utilizado durante 6. Por lo tanto, el coste proporcional del PC es de 70 €.

En cuanto a recursos humanos, que se refiere a las horas de trabajo llevadas a cabo por una persona, han participado el alumno y el tutor. Se toma como referencia de sueldo para un alumno recién graduado 25 €/h y para un tutor 50 €/h.

Por parte del tutor, se tuvieron unas seis tutorías online de media hora cada una (3 horas), más la comunicación a través de correo (1 hora) y la solución de problemas y dudas concretas fuera de tutorías (4 horas), además de la corrección de la memoria (8 horas) hacen un total de 16 horas. Por lo tanto, da un total de costes en caso del tutor de 800 €.

Por parte del alumno, se tienen en cuenta las horas de tutorías y comunicación por correo, además del trabajo autónomo de investigación, desarrollo práctico y redacción de la memoria que supuso en su conjunto unas 21 semanas aproximadamente con una media de 20 horas semanales. Por lo tanto, da un total de costes en caso del alumno de 10.600€.

| **Recursos** | | **Coste** |
|---|---|---|
| Humanos | Alumno | 10.600 € |
| | Tutor | 800 € |
| Hardware | PC | 70 € |
| Software | Programas y máquinas virtuales | 0 € |
| | **TOTAL** | **11.470 €** |

**Tabla 3.1.** Coste total del trabajo.

# 4. Estado del arte

En este apartado se comentan las soluciones similares a la que se desarrolla a lo largo de este trabajo. Se habla sobre las variantes del protocolo MQTT para *sensor networks*, para utilizar *multicast* en SDNs, utilizando el protocolo *spanning tree* o usando *middlewares* que controlen el QoS.

Si se comparan con la versión que se ha creado a lo largo de este trabajo y que se desarrollará en los apartados posteriores se puede decir que una de las ventajas que presenta es que es transparente para los nodos finales, que siguen utilizando el mismo MQTT sin ninguna diferencia. Esto hace que sea más sencillo de utilizar para los equipos ya existentes, se cambia una parte de la red SDN para optimizar de manera focalizada dicha "zona", pero los nodos finales siguen viendo lo mismo.

## 4.1. MQTT-SN

MQTT-SN está adaptado a las peculiaridades de un entorno de comunicación inalámbrica como ancho de banda bajo, fallos de enlace, longitud de mensaje corta, etc. También está optimizado para la implementación en costes bajos, dispositivos que funcionan con baterías con recursos limitados de procesamiento y almacenamiento.

En comparación con MQTT, MQTT-SN se caracteriza por las siguientes diferencias:

- El mensaje CONNECT se divide en tres mensajes. Los dos adicionales son opcionales y se utilizan para transfiera el tema Will y el mensaje Will al servidor.
- Para hacer frente a la longitud del mensaje corto y el ancho de banda de transmisión limitado en las redes inalámbricas, el *topic name* en los mensajes PUBLISH se reemplaza por uno más corto, un *topic ID* de dos bytes de longitud. Se define un procedimiento de registro para permitir que los clientes registren los *topic names* con el servidor/*gateway* y obtengan los *topic ID* correspondientes. También se usa en la dirección opuesta para informar al cliente sobre el *topic name* y el *topic ID* correspondiente que se incluirá en el siguiente mensaje PUBLISH.
- Se introducen *topic IDs* "predefinidos" y *topic names* "breves", para los que no es necesario registrarse. Los *topic IDs* predefinidos también son un reemplazo de dos bytes del *topic name*; sin embargo, su asignación a los *topic names* es conocida de antemano tanto por la aplicación del cliente como por el servidor/*gateway*.

- Un procedimiento de descubrimiento ayuda a los clientes sin una dirección de *gateway* / servidor preconfigurada a descubrir la dirección de red actual de un servidor/*gateway* operativo. Pueden estar presentes múltiples *gateways* al mismo tiempo dentro de una sola red inalámbrica y pueden cooperar en un modo de carga compartida o en espera.
- Se define un nuevo procedimiento de mantenimiento fuera de línea para el soporte de los *sleeping clients*. Con este procedimiento, los dispositivos que funcionan con baterías pueden pasar a un estado de suspensión durante el cual todos los mensajes destinados a ellos se almacenan en el búfer en el servidor/*gateway* y se les entregan más tarde cuando se despiertan.

Hay tres tipos de componentes MQTT-SN: clientes, *gateways* y reenviadores. Los clientes se conectan a un servidor MQTT a través de un *gateway* utilizando el protocolo MQTT-SN. Un *gateway* puede o no estar integrado con un servidor MQTT. En el caso de un *gateway* autónomo, el protocolo MQTT se utiliza entre el servidor MQTT y el *gateway* MQTT-SN. Su función principal es la traducción entre MQTT y MQTT-SN. En la Figura 4.1 se muestra la arquitectura de MQTT-SN.

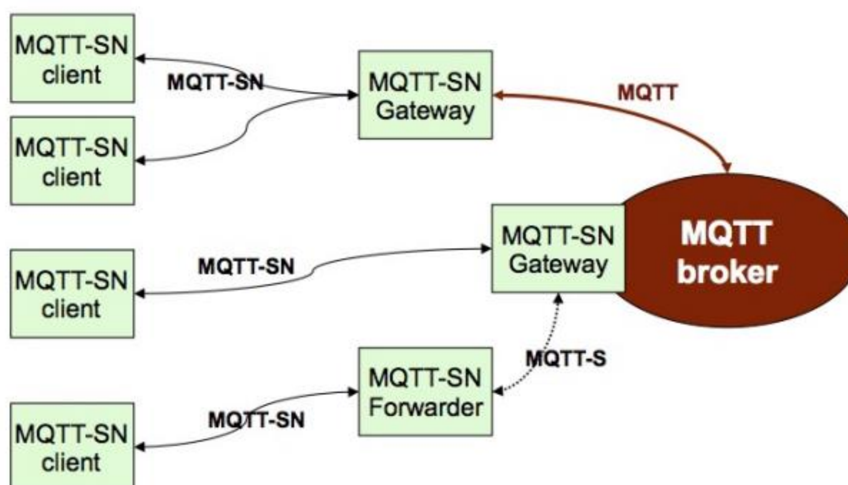


**Figura 4.1.** Arquitectura de MQTT-SN [25].

Los clientes también pueden acceder a un *gateway* a través de un reenviador en caso de que el *gateway* no esté conectado directamente a su red. El reenviador simplemente encapsula las tramas MQTT-SN que recibe en el lado inalámbrico y las reenvía sin cambios al *gateway*; en la dirección opuesta, desencapsula las tramas que recibe del *gateway* y las envía a los clientes, también sin cambios.

Dependiendo de cómo un *gateway* realice la traducción de protocolo entre MQTT y MQTT-SN, se puede diferenciar entre dos tipos de *gateways*: transparentes (*transparent*) y agregados (*aggregating*).

- **Gateway transparente.** Para cada cliente MQTT-SN conectado, un *gateway* transparente configura y mantiene una conexión MQTT al servidor MQTT. Esta conexión MQTT está reservada exclusivamente para el intercambio de mensajes extremo a extremo y casi transparente entre el cliente y el servidor. Habrá tantas conexiones MQTT entre el *gateway* y el servidor como clientes MQTT-SN conectados al *gateway*. El *gateway* transparente realizará una traducción de "sintaxis" entre los dos protocolos.

  Aunque la implementación del *gateway* transparente es más simple en comparación con la de un *gateway* agregado, requiere que el servidor MQTT admita una conexión separada para cada cliente activo. Algunas implementaciones de servidor MQTT pueden imponer una limitación en el número de conexiones simultáneas que admiten.

- **Gateway agregado.** En lugar de tener una conexión MQTT para cada cliente conectado, un *gateway* agregado tendrá solo una conexión MQTT al servidor. Todos los intercambios de mensajes entre un cliente MQTT-SN y un *gateway* agregado terminan en este último. Luego, el *gateway* decide qué información se proporcionará al servidor. Aunque su implementación es más compleja que la de un *gateway* transparente, un *gateway* agregado puede ser útil en el caso de WSNs con un número muy grande de sensores y actuadores porque reduce el número de conexiones MQTT que el servidor tiene que soportar simultáneamente.

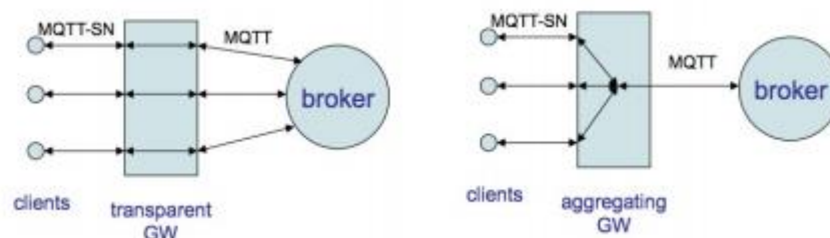


**Figura 4.2.** Gateways transparente y agregado [25].

## 4.2. DM-MQTT

El protocolo MQTT utiliza *brokers* para recopilar y transmitir los datos, lo cual es útil en redes con borde (o *edge*) de pequeño tamaño, pero puede presentar problemas como una mayor congestión en la red y retrasos en la transmisión de datos cuando se está produciendo un intercambio de datos entre diferentes redes de borde.

DM-MQTT propone una estructura jerárquica de *brokers* para resolver dichos problemas. DM-MQTT crea un *broker* esclavo (o *slave*) en el nodo borde de todas las redes de borde y el *broker* principal recopila información de borde recibida del *broker* esclavo. El controlador SDN usa la información que recibe del *broker* principal para establecer la ruta de transmisión de datos entre las diferentes redes de borde. Con esta mecánica se reduce el uso de la red al permitir que los datos se transfieran sin conexiones entre *brokers* y dispositivos en diferentes redes de borde. El mecanismo SDN-*based multicastk*, que utiliza la información de borde, permite un funcionamiento fluido de la multidifusión DM-MQTT al gestionar fácilmente grupos y árboles de multidifusión en comparación con *IP multicast*.

La estructura de dicho sistema se muestra en la Figura 4.3. La arquitectura consta de un controlador SDN con un *broker* maestro MQTT, un nodo borde con un *broker* esclavo MQTT y dispositivos IoT.

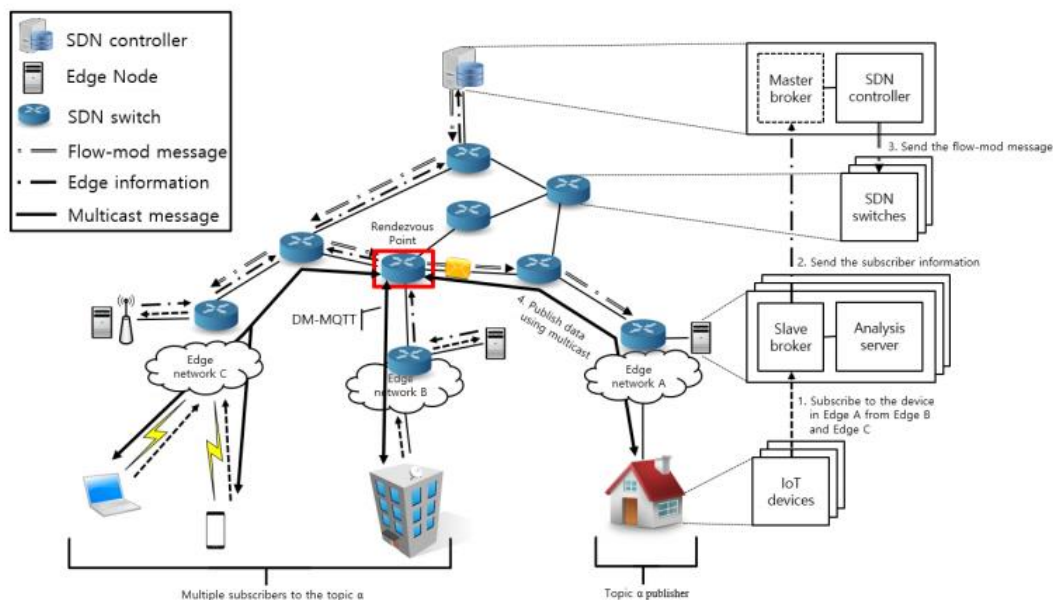


**Figura 4.3.** Arquitectura de DM-MQTT [19].

Como resultado, DM-MQTT reduce los retrasos en la transferencia de datos en un 65% y redujo el uso de la red en un 58%, en comparación con MQTT estándar. DM-MQTT no tiene

en cuenta la movilidad de los dispositivos IoT. A medida que aumenta la movilidad de los dispositivos de IoT, se requieren actualizaciones de información de borde causadas por unirse y salir de las redes de borde. Si se agrega un dispositivo de IoT a la red de borde y conectado al *broker* esclavo en el nodo de borde o eliminado en la red de borde, el *broker* esclavo debe actualizar la información de borde y enviarla al *broker* maestro. Si los dispositivos de IoT presentan movilidad dinámica, entonces se debe considerar cómo administrar la información de borde cada vez mayor y reconfigurar rápidamente los árboles de multidifusión.

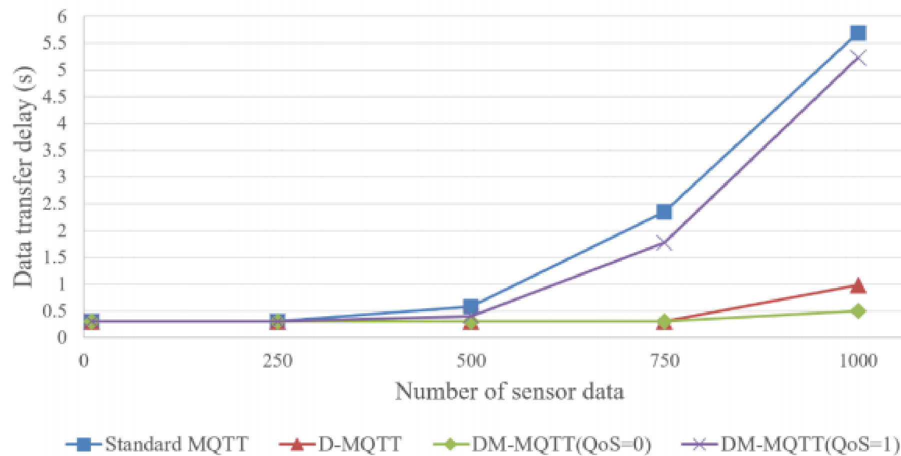


**Figura 4.4.** Cambios en el *delay* de transferencia de datos dependiendo del número de muestras de datos [19].

## 4.3. MQTT-ST

Algunas implementaciones de *broker* MQTT existentes (por ejemplo, Mosquitto, CloudMQTT, HiveMQ) permiten el uso de puentes, es decir, una conexión directa entre *brokers*. La función permite que un *broker* B se conecte a otro *broker* A como cliente estándar, suscribiéndose a todos o un subconjunto de los *topics* publicados por los clientes en A. Desafortunadamente, tal procedimiento es propenso a bucles de mensajes entre los *brokers*: de hecho, la existencia de Un ciclo en el que los corredores participantes vuelven a publicar un mensaje de forma continua, puede agotar rápidamente los recursos de un corredor y, en última instancia, hacer que no pueda entregar un tráfico significativo. Debido a la enorme complejidad de implementar la detección de duplicados en escenarios distribuidos, lo que requeriría realizar un seguimiento del productor original de cada mensaje recibido y reenviado por cualquier *broker*, las soluciones existentes requieren configurar manualmente las conexiones entre los *brokers* en una topología sin bucles, es decir, un árbol. Sin embargo, esta configuración manual de puentes MQTT tiene dos inconvenientes principales: primero, de manera similar al cableado de *switches* en empresas pequeñas o medianas, puede convertirse en una tarea muy confusa

con una alta probabilidad de crear conexiones duplicadas accidentales, especialmente en topologías grandes. En segundo lugar, al imponer una topología estática sin bucles entre los *brokers*, se pierde por completo la adaptabilidad y la solidez a los fallos. Otra opción es confiar en una forma automática de configurar un árbol entre los corredores. En redes conmutadas, esto se obtiene mediante el STP.

Los principales cambios y modificaciones de MQTT-ST con respecto a MQTT son:

- **Fase de conexión:** Al inicio, un *broker* que desee crear un puente con otro *broker* transmite un mensaje MQTT CONNECT. Para informar a un *broker* de que la solicitud de conexión proviene de otro *broker* y no de un cliente, establecemos el bit más significativo del byte de la versión del protocolo en el encabezado CONNECT.
- **Fase de señalización:** El estándar MQTT especifica un parámetro Keep Alive, que define el intervalo de tiempo máximo que puede transcurrir después de la última transmisión del cliente. En caso de que expire el temporizador, el corredor cierra la conexión con el cliente. Por lo tanto, para mantener viva la conexión, un cliente transmite mensajes PINGREQ periódicos. MQTT-ST reutiliza dichos mensajes, que desempeñan el papel de STP BPDU.
- **Selección de raíz:** El *broker* raíz juega un papel crucial en el árbol de *brokers*, ya que es el nodo de retransmisión de todo el tráfico y, por lo tanto, está sujeto a una mayor carga computacional. De hecho, la selección de un *broker* con recursos deficientes o sobrecargados puede resultar en un rendimiento general deficiente. En STP, la raíz se selecciona basándose únicamente en su identificador, lo que no se adapta bien al escenario considerado. En MQTT-ST, en cambio, el

  El *broker* raíz se selecciona de acuerdo con el valor de capacidad C, definido como:

  $$C = \alpha L + \beta M$$

  donde L es la velocidad de la CPU del *broker*, R es la cantidad de memoria RAM y $\alpha$, $\beta$ son parámetros de conversión ajustables.
- **Cálculo de ruta:** En STP, cada nodo selecciona la mejor ruta a la raíz de acuerdo con un criterio relacionado con el ancho de banda, para evitar el uso de enlaces de capacidad reducida en el árbol que pueden ralentizar toda la red. Para MQTT-ST se observa que la latencia, más que el ancho de banda, juega un papel crítico. Por lo tanto, cada *broker* monitorea continuamente el RTT a otros *brokers* y usa ese valor

para actualizar el costo de la ruta raíz P. Para hacer esto, aprovechamos el mecanismo de solicitud/respuesta ya presente en MQTT a través de PINGREQ/PINGRESP. Todas las demás conexiones están etiquetadas siguiendo la misma lógica del protocolo STP.

- **Comportamiento en tiempo de ejecución:** En cuanto a tiempo de ejecución, un *broker* MQTT-SN funciona exactamente como un *broker* MQTT desde la perspectiva de los clientes conectados. Además, por cada mensaje publicado sobre cualquier *topic*, el *broker* lo reenvía a sus conexiones no bloqueadas, mientras que cualquier mensaje entrante en una conexión bloqueada se descarta. El reenvío se realiza como un mensaje MQTT PUBLISH estándar.
- **Reacción a fallos:** Ante un fallo del *broker*, MQTT-SN maneja el error de *socket* correspondiente para restablecer el árbol de reenvío. En concreto, el *broker* que detecta el error de *socket* transmite un mensaje PINGREQ especial que se usa para reiniciar la construcción del árbol desde cero. El *broker* se establece a sí mismo como *root* y agrega al mensaje un campo de cambio de topología adicional, de manera similar a lo que sucede en STP. Cualquier *broker* que reciba un mensaje de este tipo reinicia el procedimiento de selección de raíz, que eventualmente convergerá en un nuevo árbol.

## 4.4. EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications

Muchos escenarios modernos de IoT tienen requisitos estrictos de QoS que no se pueden satisfacer usando mecanismos de *cloud computing*. Por ello surge EMMA, un *middleware* MQTT basado en el modelo suscriptor-publicador que usa *edge computing*.

EMMA consta de cuatro componentes principales: *gateway*, *brokers*, el controlador y el protocolo de monitoreo de red. En la Figura 4.5 se muestra la arquitectura de una implementación EMMA.

El papel de cada componente es el siguiente:

- **Gateway:** Son un componente clave para permitir la movilidad de clientes y *brokers* al facilitar la reconfiguración de conexiones. Su propósito es ocultar EMMA de los clientes reales mediante *tunneling* MQTT.

- **Bróker:** Administra las suscripciones a *topics* y difunde los mensajes publicados a los suscriptores. También actúan como puentes temáticos para reenviar mensajes a otros corredores que tienen suscriptores a esos *topics*.
- **Controlador:** Es el componente de orquestación del sistema donde los *gateways* y los brókers se registran cuando entran o salen de la red.
- **Monitoring:** Es un protocolo binario liviano que permite el monitoreo distribuido del QoS de la red. Cada componente de EMMA implementa dicho protocolo.

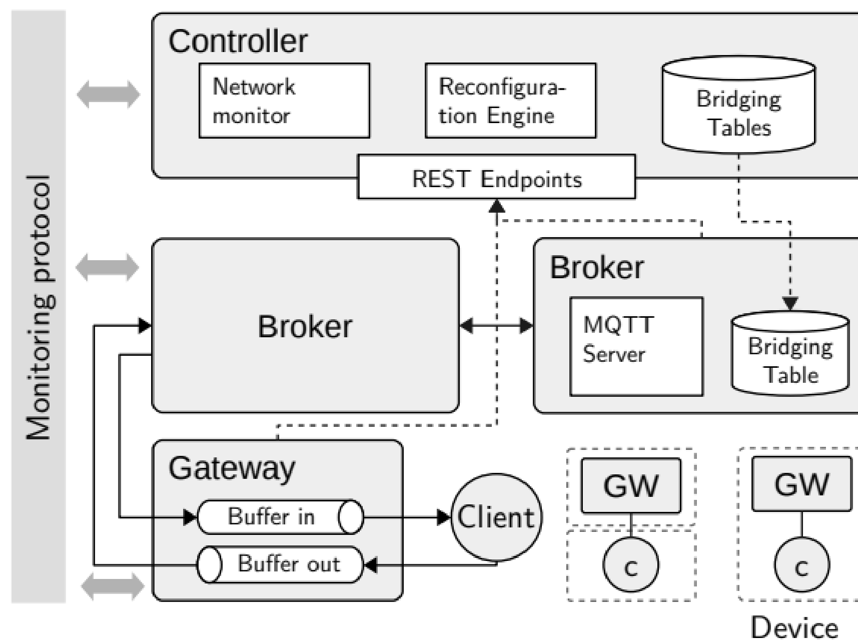


**Figura 4.5.** Estructura de EMMA [21].

EMMA se implementa como un conjunto de componentes modulares escritos en el lenguaje de programación Java. Es de código abierto y está publicado en los repositorios de código.

EMMA puede proporcionar una comunicación de baja latencia para dispositivos cercanos, a la vez que permite la difusión de mensajes a ubicaciones geográficamente dispersas con costos generalmente mínimos. Los *gateways* permiten que la infraestructura del cliente MQTT existente se conecte de forma transparente al sistema.

# 5. Herramientas

En este apartado se van a ver las herramientas más importantes a la hora del desarrollo práctico. En primer lugar, se habla de la herramienta Mininet y le siguen el controlador Ryu y el analizador de red Wireshark.

## 5.1. Mininet [16]

Mininet es un emulador de red que puede desplegar redes sobre un ordenador sencillo y con recursos limitados o máquina virtual. Éste es de código abierto y rápidamente configurable además de utilizar el kernel de Linux y otros recursos para emular los elementos de una SDN como el controlador, los *switches* OpenFlow y los *hosts*.

Existen varias opciones a la hora de instalar Mininet, pero la manera más sencilla se basa en la descarga de una máquina virtual Mininet / Ubuntu preempaquetada. La máquina virtual incluye el propio Mininet, todos los binarios y herramientas de OpenFlow preinstaladas, y modifica la configuración del *kernel* para admitir redes Mininet más grandes. En el caso de este trabajo se utiliza una máquina virtual basada en Ubuntu 18.04.03 LTS (64 bits) que incluye tanto el emulador de redes Mininet como el controlador SDN Ryu.

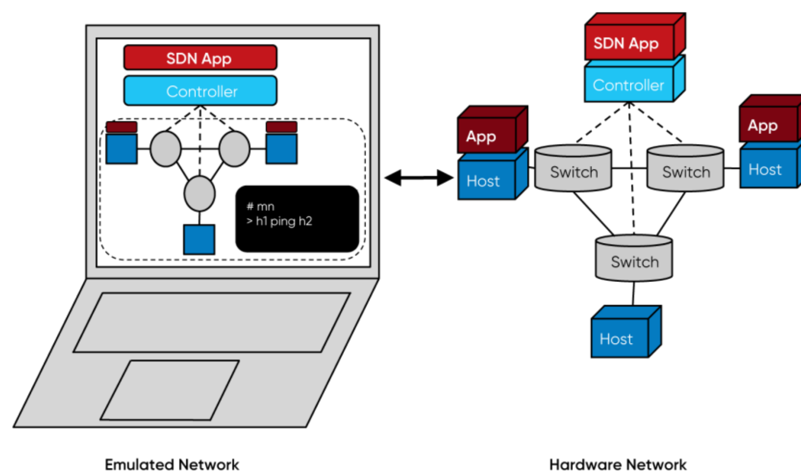


**Figura 5.1.** Esquemático de cómo funciona Mininet [17].

### 5.1.1. Ventajas y desventajas

A continuación, se muestran algunos de los puntos fuertes y limitaciones de Mininet:

- **Ventajas**
    - La creación y puesta en marcha de una red es rápida y no lleva más que unos pocos segundos.

  - Es realista ya que ofrece un comportamiento muy similar a una red hardware y escalable porque con una sola máquina física se pueden desplegar multitud de *switches* y *hosts*.
  - Se pueden crear topologías personalizadas a parte de las que ya trae por defecto Mininet. Estas topologías pueden tener el grado de complejidad que se desee.
  - Es un proyecto de código abierto, por lo que se puede examinar su código fuente, modificarlo y corregir errores.
  - Se encuentra en desarrollo activo, por lo que la comunidad de desarrolladores y usuarios de Mininet pueden intentar explicar, solucionar o ayudar a solucionar en caso de que algo no funcione o no tenga sentido.

- **Desventajas o limitaciones**
  - Mininet usa un solo kernel de Linux para todos los hosts virtuales; lo que significa que no puede ejecutar software que dependa de BSD, Windows u otros núcleos del sistema operativo.
  - Si se necesita un comportamiento de conmutación o enrutamiento personalizado, se debe encontrar o desarrollar un controlador con las funciones necesarias.
  - De manera predeterminada, la red Mininet se encuentra aislada de la LAN y de Internet.
  - Las mediciones de tiempo se basan en tiempo real ya que Mininet no tiene una noción sólida de tiempo virtual. Por ello los resultados más rápidos que en tiempo real no se pueden emular fácilmente.

### 5.1.2. Topologías

Las topologías que ofrece Mininet se pueden diferenciar en dos: predeterminadas y personalizadas. Las topologías predeterminadas se pueden usar utilizando el comando principal del programa `mn` seguido de la opción `--topo=TIPO` donde `TIPO` es la variable para indicar la topología deseada. Por otro lado, para utilizar una topología personalizada realizada con Python se puede usar la opción `--custom=CUSTOM` en caso de que se incluya una clase "topología" (p.ej. MyTopo o similar) pero si es una clase "Network" se ejecuta directamente con Python. Otras de las opciones que ofrece Mininet son las que se muestran en la Tabla 5.1.

| Opciones | Descripción |
|---|---|
| `--switch=default|ivs|lxbr|ovs|ovsbr|ovsk|user` | Crea un *switch* con las características indicadas. |
| `--host=cfs|proc|rt` | Crea un *host* con las características indicadas. |
| `--controller= default|none|nox|ovsc|ref|remote|ryu` | Indica el controlador a usar en la red. |
| `--custom=CUSTOM` | Leer clases personalizadas o parámetros de archivo(s) .py. |
| `--test= none|build|all|iperf|pingpair|iperfudp|pingall` | Realiza una prueba. |
| `-x, --xterms` | Hace aparecer xterms para cada nodo. |

**Tabla 5.1.** Algunas opciones que ofrece Mininet.

#### 5.1.2.1. Predeterminadas

Este tipo de topologías son las que ya tiene Mininet por defecto y están al alcance de los usuarios simplemente usando la opción *`--topo=TIPO`*. A continuación, se detalla que componen cada una de las topologías predeterminadas disponibles:

**Minimal**

Esta topología consta de: 1 controlador, 1 *switch* y 2 *hosts*, ambos conectados al *switch*. Se puede crear con Mininet simplemente usando el comando *mn* sin indicar nada más.

En la Figura 5.2 y Figura 5.3 se muestran tanto el esquema de esta topología como la introducción del comando en la terminal.

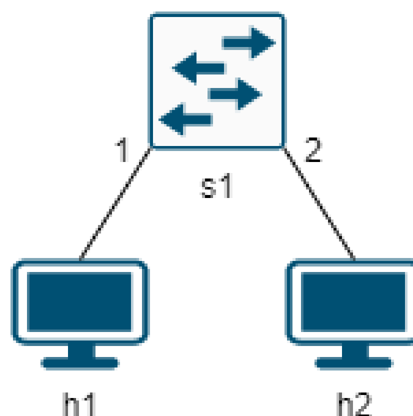


**Figura 5.2.** Esquema de la topología *minimal*.

```
jorge@jorge-sdn-vm:~/mininet/custom$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet>
```

**Figura 5.3.** Topología *minimal* creada con Mininet.

**Single**

Esta topología consta de: 1 controlador, 1 *switch* y de tantos *hosts* como se indique. Se puede crear con Mininet usando el comando *mn --topo=single,X* donde *X* representa al número de *hosts* que se desea.

En este caso, tal y como se puede ver en la Figura 5.4 y Figura 5.5, se ha fijado *X=4* para tener un total de 4 *hosts* conectados al único *switch* de esta topología.

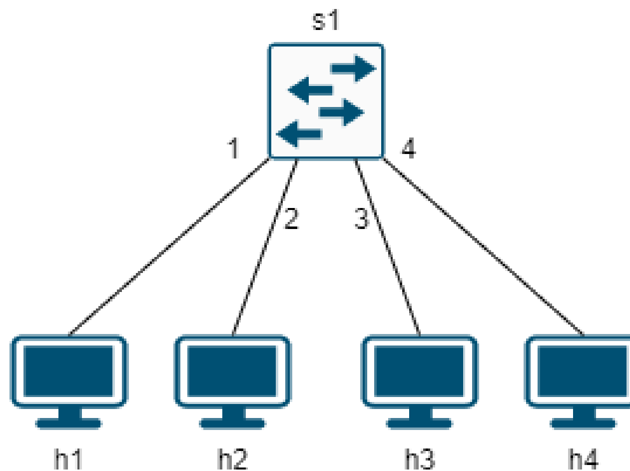


**Figura 5.4.** Esquema de la topología *single*.

```
jorge@jorge-sdn-vm:~/mininet/custom$ sudo mn --topo=single,4
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
h4 h4-eth0:s1-eth4
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0 s1-eth4:h4-eth0
c0
mininet>
```

**Figura 5.5.** Topología *single* con 4 *hosts* creada con Mininet

**Linear**

Esta topología consta de: 1 controlador, tantos *switchs* como se indiquen y de 1 *host* conectado a cada *switch*. Se puede crear con Mininet usando el comando *mn --topo=linear,X* donde *X* representa al número de *switchs* que se desean.

En este caso, tal y como se puede ver en la Figura 5.6 y Figura 5.7, se ha fijado *X=4* para tener un total de 4 *switchs* conectados entre sí y un *host* conectado a cada uno de ellos.

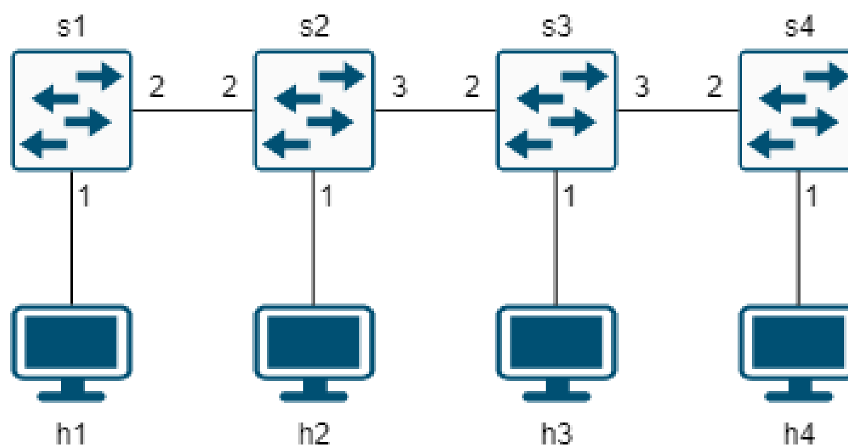


**Figura 5.6.** Esquema de la topología *linear*.

```
jorge@jorge-sdn-vm:~/mininet/custom$ sudo mn --topo=linear,4
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (s2, s1) (s3, s2) (s4, s3)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
h4 h4-eth0:s4-eth1
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth2
s2 lo:  s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo:  s3-eth1:h3-eth0 s3-eth2:s2-eth3 s3-eth3:s4-eth2
s4 lo:  s4-eth1:h4-eth0 s4-eth2:s3-eth3
c0
mininet>
```

**Figura 5.7.** Topología *linear* con 4 *switches* creada con Mininet.

**Tree**

Esta topología tiene este nombre por su similitud con un árbol. Se puede crear con Mininet utilizando el comando *mn --topo=tree,X,Y* donde *X* representa el número de niveles del árbol e *Y* representa el número de *hosts* conectados a los *switchs* del último nivel.

En este caso, tal y como se puede ver en la Figura 5.8 y Figura 5.9, se ha fijado *X=2* para tener un par de niveles e *Y=3* para tener tres *hosts* conectados a cada uno de los *switchs* del último nievel.

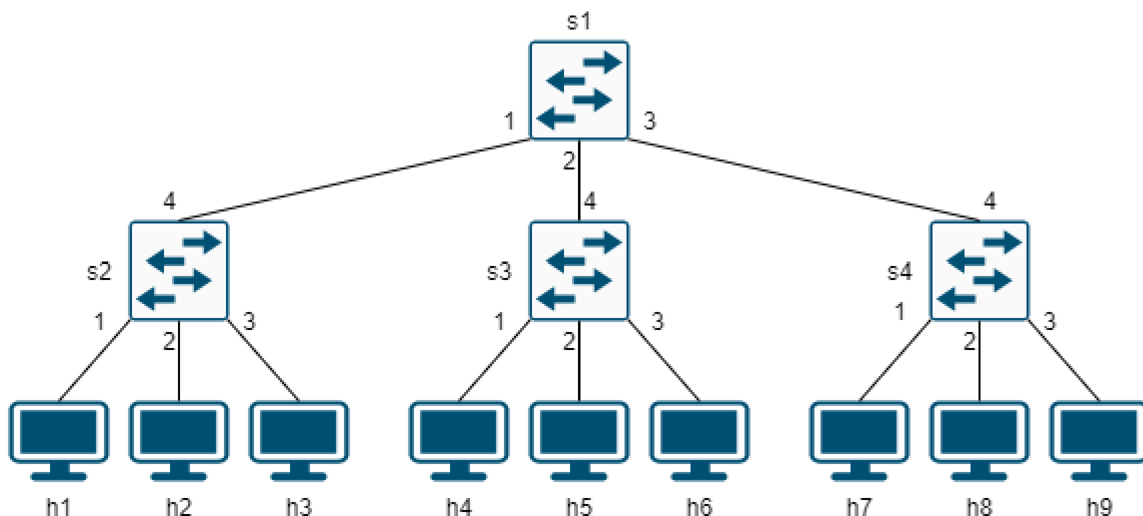


**Figura 5.8**. Esquema de la topología *tree*.

```
jorge@jorge-sdn-vm:~/mininet/custom$ sudo mn --topo=tree,2,3
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s2, h1) (s2, h2) (s2, h3) (s3, h4) (s3, h5) (s3, h6) (s4, h7) (s4, h8) (s4, h9)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s2-eth1
h2 h2-eth0:s2-eth2
h3 h3-eth0:s2-eth3
h4 h4-eth0:s3-eth1
h5 h5-eth0:s3-eth2
h6 h6-eth0:s3-eth3
h7 h7-eth0:s4-eth1
h8 h8-eth0:s4-eth2
h9 h9-eth0:s4-eth3
s1 lo:  s1-eth1:s2-eth4 s1-eth2:s3-eth4 s1-eth3:s4-eth4
s2 lo:  s2-eth1:h1-eth0 s2-eth2:h2-eth0 s2-eth3:h3-eth0 s2-eth4:s1-eth1
s3 lo:  s3-eth1:h4-eth0 s3-eth2:h5-eth0 s3-eth3:h6-eth0 s3-eth4:s1-eth2
s4 lo:  s4-eth1:h7-eth0 s4-eth2:h8-eth0 s4-eth3:h9-eth0 s4-eth4:s1-eth3
c0
mininet>
```

**Figura 5.9.** Topología *tree* con 2 niveles y 3 *hosts* por *switch* finales creada con Mininet.

#### 5.1.2.2. Personalizadas

Este tipo de topologías se pueden crear con la opción `--custom=CUSTOM` o bien ejecutando directamente con Python el *script* que contiene las características de la red diseñada. En la Tabla 5.2 se muestran las principales funciones y métodos que se pueden utilizar para diseñar la red con dicho *script*.

| Funciones/Clases/Metodos/Variables | Descripción |
|---|---|
| `Topo` | La clase base para topologías Mininet. |
| `build()` | Método para inicializar la topología. |
| `addSwitch()` | Añade un *switch* a la topología y devuelve su nombre. |
| `addHost()` | Añade un *host* a la topología y devuelve su nombre. |
| `addLink()` | Añade un enlace bidireccional a menos que se indique lo contrario. |
| `Mininet` | Clase principal para crear y administrar una red. |
| `start()` | Inicia una red. |
| `stop()` | Detiene la red. |
| `pingAll()` | Prueba la conectividad intentando que todos los nodos se hagan ping entre sí. |
| `net.hosts` | Todos los *hosts* de una red. |

**Tabla 5.2.** Funciones/Clases/Métodos/Variables para topologías personalizadas.

El siguiente código se corresponde con el *script* creado como ejemplo para familiarizarse e iniciarse en la creación de topologías personalizadas. Consta de un total de 4 *switchs* y 6 *hosts*, en la Figura 5.10 y Figura 5.11 se puede apreciar la forma de la topología.

```
1 """Custom topology"""
2
3 from mininet.topo import Topo
4
5 class MyTopo(Topo):
6         def __init__(self):
7                 "Create custom topo."
8                 #Initialize topology
9                 Topo.__init__(self)
10
11                #Add hosts and switches
12                highSwitch = self.addSwitch('s1')
13                leftSwitch = self.addSwitch('s2')
14                middleSwitch = self.addSwitch('s3')
15                rightSwitch = self.addSwitch('s4')
16                baHost = self.addHost('h1')
17                bbHost = self.addHost('h2')
18                caHost = self.addHost('h3')
19                cbHost = self.addHost('h4')
20                daHost = self.addHost('h5')
21                dbHost = self.addHost('h6')
22
23                #Add links
24                self.addLink(highSwitch, leftSwitch)
25                self.addLink(highSwitch, middleSwitch)
26                self.addLink(highSwitch, rightSwitch)
27                self.addLink(leftSwitch, baHost)
28                self.addLink(leftSwitch, bbHost)
29                self.addLink(middleSwitch, caHost)
30                self.addLink(middleSwitch, cbHost)
31                self.addLink(rightSwitch, daHost)
32                self.addLink(rightSwitch, dbHost)
33 topos = {'mytopo': (lambda: MyTopo())}
```

**Código 5.1.** Ejemplo de topología personalizada.

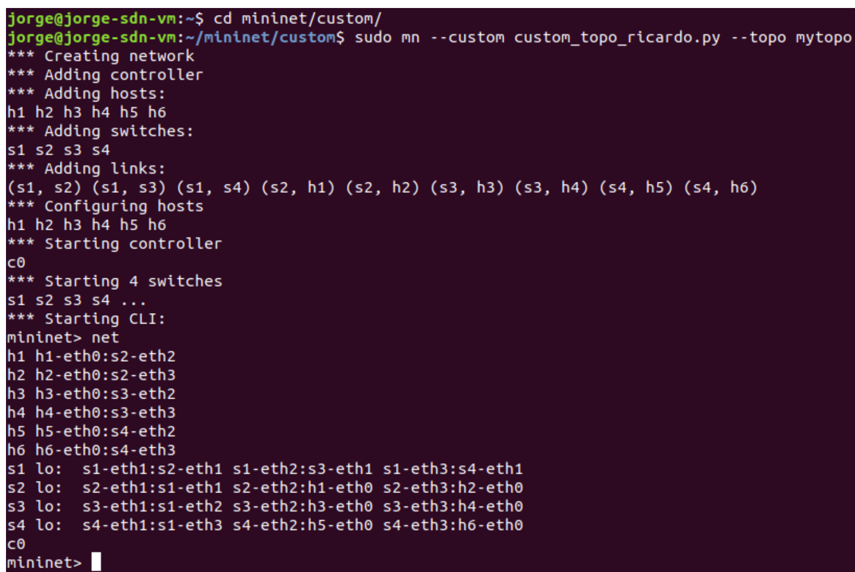


**Figura 5.10.** Creación de la topología personalizada de ejemplo con Mininet.

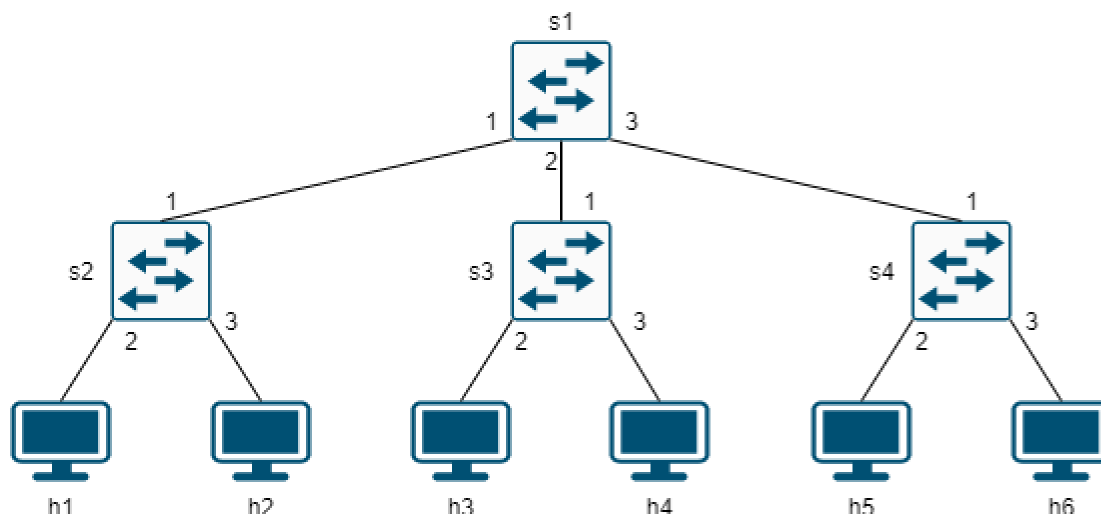


**Figura 5.11.** Esquema de la topología personalizada de ejemplo.

### 5.1.3. Comandos

Una vez se crea una topología el *prompt* cambia a `mininet>` y se pueden introducir una serie de comandos que se muestran a continuación (ver Figura 5.12 y Tabla 5.3):

```
mininet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperfudp  nodes        pingpair      py      switch
dpctl  help   link      noecho       pingpairfull  quit    time
dump   intfs  links     pingall      ports         sh      x
exit   iperf  net       pingallfull  px            source  xterm

You may also send a command to a node using:
  <node> command {args}
For example:
  mininet> h1 ifconfig

The interpreter automatically substitutes IP addresses
for node names when a node is the first arg, so commands
like
  mininet> h2 ping h3
should work.

Some character-oriented interactive commands require
noecho:
  mininet> noecho h2 vi foo.py
However, starting up an xterm/gterm is generally better:
  mininet> xterm h2

mininet> 
```

**Figura 5.12.** Introducción de comando `help`.

| **Comando** | **Descripción** |
|---|---|
| `exit/quit` | Cierra el sistema. |
| `xterm` | Abre un terminal del nodo(s) indicados. |
| `link [node1] [node2] [up/down]` | Crea/elimina un enlace entre nodos. |
| `[node] ping [node]` | Realiza un *ping* entre los nodos indicados. |
| `pingall` | Realiza un *ping* a todos los nodos de la red. |
| `net` | Muestra las conexiones entre los nodos de la red. |
| `dump` | Muestra información de los nodos. |
| `switch` | Permite dar órdenes a un *switch*. |
| `nodes` | Permite dar órdenes a un nodo. |
| `[node] iperf [options]` | Prueba el ancho de banda entre dos *hosts*. |

**Tabla 5.3.** Algunos comandos de Mininet y su función.

## 5.2. Ryu [22]

Ryu es un controlador de código abierto, bajo la licencia de Apache 2.0, utilizado para crear y diseñar SDNs. Este se encarga de aumentar la agilidad de la red facilitando la gestión y adaptación de cómo se maneja el tráfico. Ryu admite varios protocolos para administrar dispositivos de red, como OpenFlow, Netconf, OF-config, etc y se programa enteramente en Python. En cuanto a OpenFlow, que es el utilizado en este trabajo, Ryu admite completamente las extensiones 1.0, 1.2, 1.3, 1.4, 1.5 y Nicira.

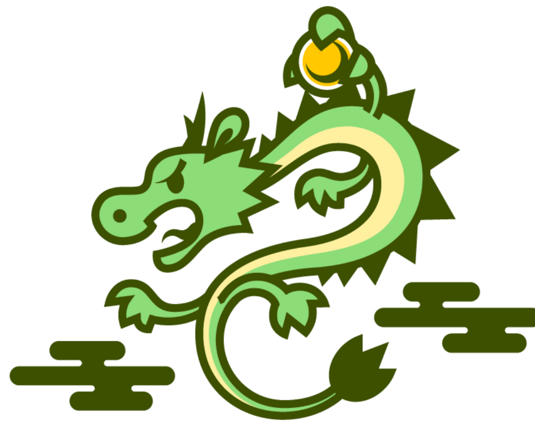

**Figura 5.13.** Logo de Ryu [22].

Provee componentes de software e interfaces API que ayudan al desarrollo de la administración y control de la SDN. La comunicación es hacia arriba (*northbound*) con las API's y (*southbound*) con OpenFlow, como se muestra en la Figura 5.14.

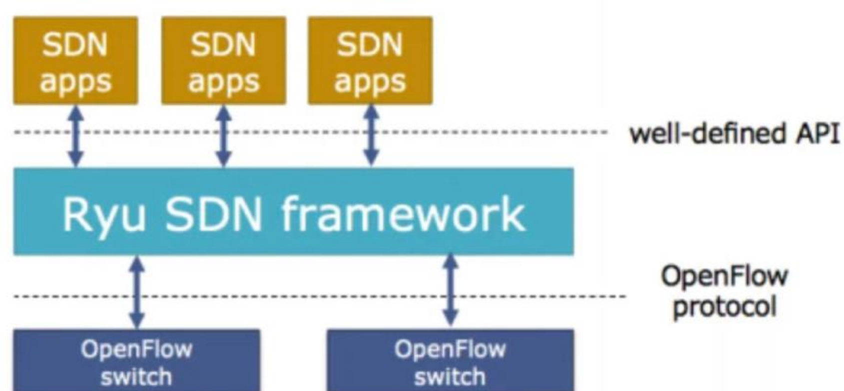


**Figura 5.14.** Controlador Ryu en un ambiente SDN [5].

### 5.2.1. Instalación

El código fuente de Ryu se encuentra en GitHub y es administrado y mantenido por la comunidad abierta de Ryu.

Ryu se puede instalar directamente desde la línea de comandos o desde el código fuente:

- Desde la línea de comandos usando el comando *pip* (opción más fácil):

  ```
  % pip install ryu
  ```

- Desde el código fuente:

  ```
  % git clone git://github.com/osrg/ryu.git
  % cd ryu; python ./setup.py install
  ```

### 5.2.2. Componentes de Ryu

- **Ejecutables**
  - *bin/ryu-manager*

    Este es el ejecutable principal.
- **Componentes base**
  - *ryu.base.app_manager*

    Gestión centralizada de las aplicaciones Ryu. Se encarga de cargar aplicaciones Ryu, proporcionar contextos a las aplicaciones Ryu y enrutar mensajes entre aplicaciones de Ryu.
- **Controlador OpenFlow**
  - *ryu.controller.controller*

    Componente principal del controlador OpenFlow. Se encarga de manejar conexiones desde *switches* y genera y enruta eventos a entidades apropiadas como aplicaciones Ryu.
  - *ryu.controller.dpset*

    Gestionar *switches*.
  - *ryu.controller.ofp_event*

    Definiciones de eventos de OpenFlow.

   - *ryu.controller.ofp_handler*

     Manejo básico de OpenFlow, incluida la negociación.

- **Codificador y decodificadores de protocolo OpenFlow**
   - *ryu.ofproto.ofproto_v1_(0,1,2,3,4 o 5)*

     Definiciones de Openflow.

   - *ryu.ofproto.ofproto_v1_(0,1,2,3,4 o 5)_parser*

     Implementaciones de decodificador/codificador de OpenFlow.

- **Aplicaciones de Ryu**
   - *ryu.app.cbench*

     Un respondedor de OpenFlow 1.0 para comparar el marco del controlador. Diseñado para usarse con oflops cbench.

   - *ryu.app.simple_switch*

     Una implementación de *learning switch* OpenFlow 1.0 L2.

   - *ryu.topology*

     Módulo de descubrimiento de *switches* y enlaces.

- **Bibliotecas**
   - *ryu.lib.packet*

     Biblioteca de paquetes Ryu. Implementaciones de decodificadores/codificadores de protocolos como TCP/IP.

   - *ryu.lib.ovs*

     Biblioteca de interacción ovsdb.

   - *ryu.lib.of_config*

     Implementación de OF-Config.

   - *ryu.lib.netconf*

     Definiciones NETCONF utilizadas por ryu/lib/of_config.

   - ryu.lib.xflow

     Una implementación de sFlow y NetFlow.

- **Bibliotecas de terceros**
  - *ryu.contrib.ovs*

    Abre el enlace de Python de vSwitch. Utilizado por *ryu.lib.ovs*.
  - *ryu.contrib.oslo.config*

    Biblioteca de configuración Oslo. Se utiliza para las opciones de línea de comandos y los archivos de configuración de ryu-manager.
  - *ryu.contrib.ncclient*

    Biblioteca de Python para el cliente NETCONF. Utilizado por *ryu.lib.of_config*.

## 5.3. Wireshark [27]

Wireshark es el analizador de protocolos de red más relevante y más usado del mundo. Éste permite observar lo que está sucediendo en la red a un nivel imperceptible. Wireshark es la continuación de un proyecto iniciado por Gerald Combs en 1998 y sigue en desarrollo gracias a las contribuciones voluntarias de expertos en redes de todo el mundo.

Algunas de las funciones que incluye Wireshark son las siguientes:

- Inspección profunda de cientos de protocolos, y se van agregando más.
- Captura en vivo y análisis fuera de línea.
- Multiplataforma: se ejecuta en Windows, Linux, macOS, Solaris, FreeBSD, NetBSD y muchos otros.
- Leer / escribir muchos formatos de archivo de captura diferentes: tcpdump (libpcap), Pcap NG, Catapult DCT2000, Cisco Secure IDS iplog, Microsoft Network Monitor, Network General Sniffer (comprimido y sin comprimir) y muchos otros.
- Los archivos de captura comprimidos con gzip se pueden descomprimir sobre la marcha.
- Los datos en vivo se pueden leer desde Ethernet, IEEE 802.11, PPP/HDL, ATM, Bluetooth, USB, Token Ring, Frame Relay, FDDI y otros.
- Soporte de descifrado para muchos protocolos, incluidos Ipsec, ISAKMP, Kerberos, SNMPv3, SSL/TLS, WEP y WAP/WAP2.
- La salida se puede exportar en XML, PostScript, CSV o texto sin formato.

## 5.3. Wireshark [27]

A continuación, se mencionan las partes de la interfaz gráfica de Wireshark con la cual se trabaja a lo largo del trabajo para capturar y analizar los paquetes. Apoyándose en la Figura 5.15 se distinguen las siguientes secciones de arriba abajo:

- **Barra de herramientas:** Muestra todas las opciones a realizar sobre la pre y pos captura.
- **Barra de herramientas principal:** Aquí se encuentran las opciones más utilizadas en Wireshark.
- **Listado de paquetes:** Se muestran los paquetes que se han capturado de manera resumida.
- **Panel de detalles de paquetes:** Seleccionando uno de los paquetes del listado se muestra información más detallada del mismo.
- **Panel de bytes de paquetes:** Seleccionando uno de los paquetes del listado se muestran los bytes del mismo y se resaltan los bytes pertenecientes al campo seleccionado en el panel de detalles de paquetes.

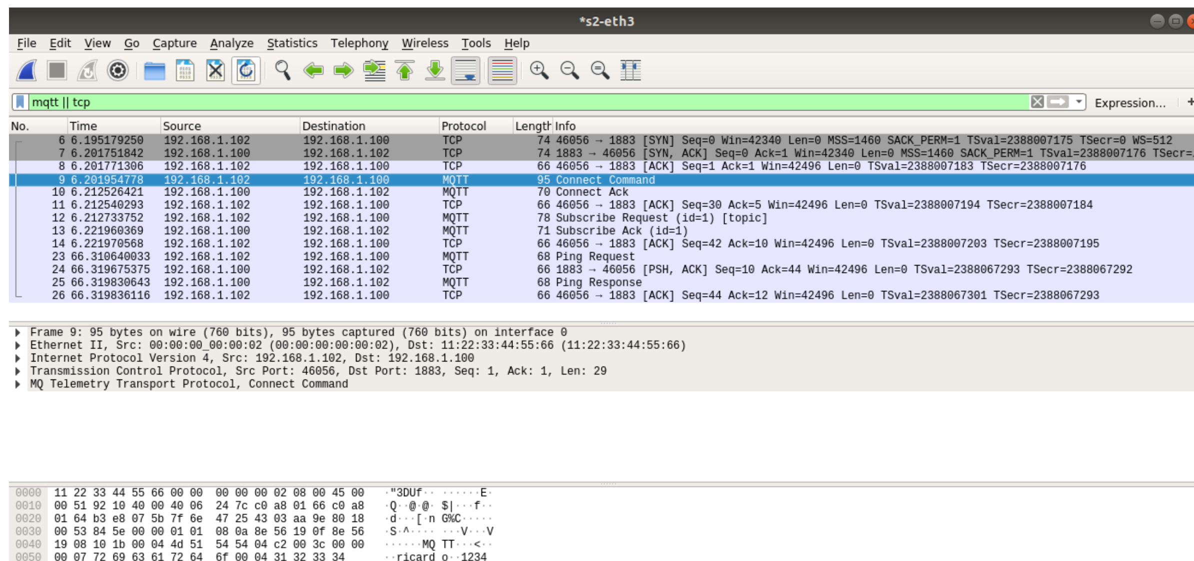


**Figura 5.15.** Interfaz gráfica de Wireshark.

# 6. Fundamentos teóricos

En este apartado se recoge la información teórica referente a los dos protocolos más importantes del trabajo, MQTT y OpenFlow.

## 6.1. MQTT

El protocolo de mensajería simple conocido como MQTT está diseñado para dispositivos con ancho de banda limitada, redes poco fiables o alta latencia.

Dicho protocolo fue inventado por el Dr. Andy Stanford-Clark de IBM y Arlen Nipper de Arcom en el año 1999 así como las versiones 5.0 y 3.1.1 (ratificado por el ISO) las cuales son estándares de OASIS. IANA, reserva los puertos TPC/IP 1883 para MQTT y 8883 para MQTT sobre SSL

MQTT puede trabajar tanto sobre TCP/IP como con otros protocolos que puedan permitir la conexión bidireccional, sin pérdidas y ordenada. El objetivo principal de este protocolo es rebajar los requerimientos de ancho de banda de red además de los recursos de los dispositivos que lo usan. Esto, manteniendo un grado de fiabilidad, es lo que conlleva que su uso en IoT o conexiones *machine-to-machine* (M2M) sea adecuado.

### 6.1.1. Modelo

En el modelo MQTT existe una característica principal y es que sigue la arquitectura de comunicación cliente-servidor con una topología en estrella normalmente junto con un nodo central que da uso como servidor y la capacidad de hasta 10.000 clientes.

Los componentes principales en un escenario MQTT son los siguientes:

- **Bróker:** este actúa como servidor encargándose de la transmisión de mensajes tanto con los clientes como con la gestión de la red manteniendo activo el canal con dichos clientes y respondiendo a los mensajes regulares que estos envían.
- **Cliente:** puede aparecer como publicador y suscriptor teniendo ambas funciones al mismo tiempo.
    - **Publicador:** se encarga de que la información sea transmitida al *broker* sobre un *topic* determinado.
    - **Suscriptor:** se encarga de recibir información del *broker* sobre un *topic* determinado.

- **Mensaje:** se trata de la información o unidades de datos que se transmite o recibe sobre un *topic*.
- **Topic:** se trata del tema en el que los clientes se suscriben para así recibir información sobre este y publicarla.

Es necesario el establecimiento de una sesión para que dé comienzo la comunicación entre el *broker* y el cliente la cual se mantiene activa el tiempo que sea necesario mediante mensajes periódicos. Durante dicha sesión hay un intercambio de paquetes y al concluir se cerrará.

En este proceso el *broker* es el encargado de controlar tanto la red como los mensajes mencionados anteriormente. Esto conlleva a que la comunicación entre el transmisor y el receptor acabe desacoplándose y suponiendo tres ventajas en este tipo de comunicaciones:

- El publicador sólo necesita conocer la dirección IP y puerto del *broker*, siendo irrelevantes en la publicación los destinatarios del mensaje.
- El publicador y suscriptor no tienen por qué estar conectados a la vez puesto que el *broker* se encargará de almacenar la información.
- El publicador y suscriptor no necesitan sincronizarse.

### 6.1.2. Formato paquetes

En el protocolo MQTT hay una serie de paquetes de control que se intercambian en un orden determinado para poder efectuar la comunicación. Estos se dividen en tres partes diferentes dependiendo del tipo de paquete.

#### 6.1.2.1. Cabecera Fija

Se conoce como *fixed header* o cabecera fija a la parte inicial de los paquetes MQTT. Esta es la única que está presente en todos ellos y se divide en distintos campos.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type | | | | DUP flag | QoS level | | RETAIN |
| byte 2 | Remaining Length | | | | | | | |

**Figura 6.1.** *Fixed header* o cabecera fija de paquete MQTT.

- ***Message Type:*** corresponde a los cuatro primeros bits del primer byte e indica los posibles tipos de mensajes del protocolo, como se detalla en la Tabla 6.1.

- ***Flags:*** aparecen en la segunda mitad del primer byte. Aunque la mayoría de los mensajes tienen unos valores determinados y fijos, algunos tienen variables para indicar ciertas características.
  - **DUP:** bit de duplicidad que indica si el receptor puede haber recibido ya ese mensaje.
  - **QoS:** indica la calidad de servicio que se está usando (0, 1 o 2).
  - **Retain:** si está activo (valor a 1) indica al servidor que retenga el mensaje PUBLISH para enviarlo a futuras suscripciones.
- ***Remaining Length:*** indica el número de bytes restantes del paquete incluyendo la cabecera variable y el *payload*. La longitud del propio campo también es variable, pudiendo ser desde uno hasta cuatro bytes.

| Mnemotecnia | Numeración | Descripción |
|---|---|---|
| Reserved | 0 | Reservado |
| CONNECT | 1 | Petición de conexión |
| CONNACK | 2 | Confirmación de conexión |
| PUBLISH | 3 | Mensaje de publicación |
| PUBACK | 4 | Confirmación de publicación |
| PUBREC | 5 | Recepción de publicación (entrega asegurada parte 1) |
| PUBREL | 6 | Lanzamiento de publicación (entrega asegurada parte 2) |
| PUBCOMP | 7 | Publicación completada (entrega asegurada parte 3) |
| SUBSCRIBE | 8 | Petición de suscripción |
| SUBACK | 9 | Confirmación suscripción |
| UNSUBSCRIBE | 10 | Petición de cancelación de suscripción |
| UNSUBACK | 11 | Confirmación de cancelación de suscripción |
| PINGREQ | 12 | Solicitud de PING |
| PINGRESP | 13 | Respuesta de PING |
| DISCONNECT | 14 | Desconexión de cliente |
| Reserved | 15 | Reservado |

**Tabla 6.1.** Tipos de mensajes MQTT.

#### 6.1.2.2. Cabecera variable

Se conoce como *variable header* o cabecera variable a la parte que se encuentra entre la cabecera fija y el *payload* de los paquetes MQTT. Solo algunos paquetes contienen esta cabecera.

Los campos que pueden aparecer en esta cabecera dependiendo del tipo de mensaje MQTT o del nivel de calidad de servicio (QoS) son:

- ***Protocol name:*** el nombre de protocolo aparece en la cabecera variable de un mensaje MQTT CONNECT.
- ***Protocol versión:*** la versión del protocolo está presente en la cabecera variable de un mensaje MQTT CONNECT. Es un campo de 8-bit de valor sin signo que representan el nivel de versión del protocolo usado por el cliente.
- ***Connect flags:*** el *Clean session*, *Will*, *Will QoS* y *Retain flags* están presentes en la cabecera variable de un mensaje MQTT CONNECT.
    - ***Clean sesión flag:*** ocupa el bit 1 del byte de *Connect flags.* Si no se establece (0), el servidor debe almacenar las suscripciones del cliente después de que se deconecte. Si se establece (1), el servidor debe descartar cualquier información previamente mantenida sobre el cliente y tratar la conexión como “limpia”.
    - ***Will flag:*** ocupa el bit 2 del byte de *Connect flags*. El mensaje *Will* define que el servidor publica un mensaje en nombre del cliente cuando el servidor encuentre un error E/S durante la comunicación con el cliente o el cliente no se comunica dentro del tiempo programado por el *Keep Alive*. Si se establece *Will flag*, los campos *Will QoS* y *Will Retain* deben estar presentes en el byte de *Connect flags*, y los campos *Will Topic* y *Will Message* deben estar presentes en el *payload*.
    - ***Will QoS:*** ocupa los bits 4 y 3 del byte de *Connect flags*. Un cliente que se conecta especifica el nivel de QoS en el campo Will QoS para un mensaje *Will* que se envía en caso de que el cliente se desconecte involuntariamente. El mensaje *Will* se define en el *payload* de un mensaje CONNECT.
    - ***Will Retain flags:*** ocupa el bit 5 del byte de *Connect flags*. El *Will Retain flag* indica si el servidor debe retener el mensaje *Will* que publica el servidor en nombre del cliente en caso de que el cliente se desconecte inesperadamente.
    - ***User name* y *password flags:*** ocupa los bits 6 y 7 del byte de *Connect flags*. Un cliente que se conecta puede especificar un nombre de usuario y una contraseña. Especificar estos bits significa que un nombre de usuario y opcionalmente una contraseña están incluidas in el *payload* de un mensaje CONNECT.

- ***Keep Alive timer:*** este temporizador está presente en la cabecera variable de un mensaje CONNECT. El temporizador es un valor de 16 bits, medido en segundos, que define el intervalo de tiempo máximo entre los mensajes recibidos de un cliente. Permite al servidor detectar que la conexión de red a un cliente se ha caído, sin tener que esperar el tiempo de espera prolongado de TCP / IP. El cliente tiene la responsabilidad de enviar un mensaje dentro de cada período de tiempo de *Keep Alive*.
- ***Connect return code:*** este código se envía en la cabecera variable de un mensaje CONNACK. Este campo define con un byte sin signo un código de retorno. En la Tabla 6.2 se muestran el significado de los valores:

| Numeración | HEX | Significado |
|---|---|---|
| 0 | 0x00 | Conexión aceptada |
| 1 | 0x01 | Conexión rechazada: versión de protocolo inaceptable |
| 2 | 0x02 | Conexión rechazada: identificador rechazado |
| 3 | 0x03 | Conexión rechazada: servidor no disponible |
| 4 | 0x04 | Conexión rechazada: nombre de usuario o contraseña incorrectos |
| 5 | 0x05 | Conexión rechazada: no autorizado |
| 6 - 255 | | Reservado para uso futuro |

**Tabla 6.2.** Valor de códigos de retorno.

- ***Topic name:*** está presente en la cabecera variable de un mensaje PUBLISH. El nombre del *topic* es la clave que identifica el canal de información en el que se publican los datos del *payload*. Los suscriptores utilizan el *topic* como clave para identificar los canales de información en los que quieren recibir información publicada. El nombre del *topic* es una cadena codificada en UTF.

#### 6.1.2.3. Carga útil

Se conoce como *payload* o carga útil a la parte que constituye el final del paquete. Ésta contiene la información que busca transmitirse, es de longitud variable y aparece sólo en algunos tipos de mensajes MQTT. Estos son:

- **CONNECT.** El *payload* contiene una o más cadenas codificadas en UTF-8. Especifican un identificador único para el cliente, un *topic* y mensaje, y el nombre

de usuario y la contraseña. Todos menos el primero son opcionales y su presencia se determina en función de los *flags* en la cabecera variable.

- **SUSCRIBIR.** El *payload* contiene una lista de nombres de *topics* a los que el cliente puede suscribirse y el nivel de QoS. Estas cadenas están codificadas en UTF.
- **SUBACK.** El *payload* contiene una lista de niveles de QoS concedidos. Estos son los niveles de QoS en los que los adminitradores del servidor han permitido que el cliente se suscriba a un nombre de *topic* en particular. Los niveles de QoS concedidos se enumeran en el mismo orden que los nombres de los *topics* en el mensaje SUBSCRIBE correspondiente.

La parte del *payload* de un mensaje PUBLISH contiene solo datos específicos de la aplicación.

#### 6.1.2.4. Identificadores de mensajes

El *Message identifier* o identificador de mensaje está presente en la cabecera variable de los siguientes mensajes MQTT: PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE y UNSUBACK.

El campo identificador de mensaje (*message ID*) solo está presente en mensajes donde los bits de QoS del encabezado fijo indican niveles de QoS 1 o 2.

El *message ID* es un entero sin signo de 16 bits que debe ser único entre el conjunto de mensajes “en vuelo” en una dirección de comunicación particular. Por lo general, aumenta exactamente uno de un mensaje a otro, pero no es necesario que lo haga.

El orden de los dos bytes del identificador de mensaje es MSB y luego LSB (*big-endian*).

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | Message Identifier MSB | | | | | | | |
| | Message Identifier LSB | | | | | | | |

**Figura 6.2.** Identificador de mensaje MSB y LSB.

#### 6.1.2.5. Tipos de mensajes

**CONNECT**

Cuando se establece una conexión de socket TCP/IP de un cliente a un servidor es el primer mensaje enviado por parte del cliente hacia el servidor.

A continuación, se detallan la cabecera fija, la cabecera variable y el *payload* en específico para un mensaje CONNECT:

### I. Cabecera fija

Los campos DUP, QoS y RETAIN no se usan en el mensaje CONNECT. La longitud restante es la longitud de la cabecera variable (12 bytes) y la longitud del *payload*.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (1) | | | | DUP flag | QoS level | | RETAIN |
| | 0 | 0 | 0 | 1 | x | x | x | x |
| byte 2 | Remaining Length | | | | | | | |

**Figura 6.3.** Cabecera fija de un mensaje CONNECT.

### II. Cabecera variable

*User name flag* puesto a (1), *Password flag* puesto a (1), *Clean Session flag* puesto a (1), *Keep Alive timer* puesto a 10 segundos (0x000A), *Will flag* puesto a (1), *Will QoS* es 1 y *Will RETAIN* está vacio (0).

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Protocol Name | | | | | | | | | |
| byte 1 | Length MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 2 | Length LSB (6) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| byte 3 | 'M' | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| byte 4 | 'Q' | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| byte 5 | 'I' | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| byte 6 | 's' | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| byte 7 | 'd' | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| byte 8 | 'p' | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Protocol Version Number | | | | | | | | | |
| byte 9 | Version (3) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| Connect Flags | | | | | | | | | |
| byte 10 | User name flag (1)<br>Password flag (1)<br>Will RETAIN (0)<br>Will QoS (01)<br>Will flag (1)<br>Clean Session (1) | 1 | 1 | 0 | 0 | 1 | 1 | 1 | x |
| Keep Alive timer | | | | | | | | | |
| byte 11 | Keep Alive MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 12 | Keep Alive LSB (10) | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

**Figura 6.4.** Cabecera variable de un mensaje CONNECT.

### III. Payload

La carga útil del mensaje CONNECT contiene una o más cadenas codificadas en UTF-8, según los campos del encabezado de la variable. Las cadenas, si están presentes, deben aparecer en el siguiente orden:

- **a.** ***Client Identifier.*** El identificador de cliente (ID de cliente) tiene entre 1 y 23 caracteres e identifica de forma exclusiva al cliente en el servidor.
- **b.** ***Will Topic.*** El *Will Message* se publica en el *Will Topic*. El nivel de QoS es definido por el campo *Will QoS* y el estado de RETAIN es definido por el campo *Will* RETAIN en la cabecera variable.
- **c.** ***Will Message.*** Define el contenido del mensaje que se publica en *Will Topic* si el cliente se desconecta inesperadamente
- **d.** ***User Name.*** El nombre de usuario identifica al usuario que se está conectando, que se puede utilizar para la autenticación.
- **e.** ***Password.*** Contraseña correspondiente al usuario que se está conectando, que se puede utilizar para la autenticación.

**CONNACK**

Es el mensaje enviado por el servidor en respuesta a una solicitud CONNECT de un cliente.

### I. Cabecera fija

Los campos DUP, QoS y RETAIN no se utilizan en el mensaje CONNACK.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message type (2) | | | | DUP flag | QoS flags | | RETAIN |
| | 0 | 0 | 1 | 0 | x | x | x | x |
| byte 2 | Remaining Length (2) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figura 6.5.** Cabecera fija de un mensaje CONNACK.

### II. Cabecera variable

Los posibles valores para el campo de un byte sin signo *Connect return* se muestran en la Tabla 6.2.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Topic Name Compression Response | | | | | | | | | |
| byte 1 | Reserved values. Not used. | x | x | x | x | x | x | x | x |
| Connect Return Code | | | | | | | | | |
| byte 2 | Return Code | | | | | | | | |

**Figura 6.6.** Cabecera variable de un mensaje CONNACK.

### III. Payload

No hay *payload*.

## PUBLISH

Un cliente envía este mensaje a un servidor para distribuirlo a los suscriptores interesados. Cada mensaje PUBLISH está asociado con un nombre de *topic* (también conocido como Asunto o Canal).

### I. Cabecera fija

El nivel QoS puesto a uno (1) y DUP puesto a cero (0), esto significa que el mensaje se ha enviado por primera vez. RETAIN puesto a cero (0), esto significa que no retener, y *Reamining Length* indica la longitud de la cabecera variable más la longitud del *payload*, puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message type (3) | | | | DUP flag | QoS level | | RETAIN |
| | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| byte 2 | Remaining Length | | | | | | | |

**Figura 6.7.** Cabecera fija de un mensaje PUBLISH.

### II. Cabecera variable

Contiene el *Topic Name*, una cadena codificada en UTF que indica el *topic* para el que se publica el mensaje (los dos primeros bytes indican la longitud del nombre) y el *Message ID* presente únicamente para mensajes con QoS nivel 1 y nivel 2.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Topic Name | | | | | | | | | |
| byte 1 | Length MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 2 | Length LSB (3) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| byte 3 | 'a' (0x61) | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| byte 4 | '/' (0x2F) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| byte 5 | 'b' (0x62) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| Message Identifier | | | | | | | | | |
| byte 6 | Message ID MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 7 | Message ID LSB (10) | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

**Figura 6.8.** Cabecera variable de un mensaje PUBLISH.

### III. Payload

Contiene los datos para publicar. El campo *Remaining Length* en la cabecera fija incluye tanto la longitud de la cabecera variable como la longitud del *payload*.

## PUBACK

Es la respuesta a un mensaje PUBLISH con nivel de QoS 1. Un mensaje PUBACK es enviado por un servidor en respuesta a un mensaje PUBLISH de un cliente de publicación, y por un suscriptor en respuesta a un mensaje PUBLISH del servidor.

### I. Cabecera fija

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*. La longitud de la cabecera variable, *Remaining length*, es de 2 bytes. Puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (4) | | | | DUP flag | QoS level | | RETAIN |
| | 0 | 1 | 0 | 0 | x | x | x | x |
| byte 2 | Remaining Length (2) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figura 6.9.** Cabecera fija de un mensaje PUBACK.

### II. Cabecera variable

Contiene el identificador de mensaje (*Message ID*) para el mensaje PUBLISH que se está reconociendo.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message ID MSB | | | | | | | |
| byte 2 | Message ID LSB | | | | | | | |

**Figura 6.10.** Cabecera variable de un mensaje PUBACK.

**III. Payload**

No hay *payload*.

**PUBREC (parte 1)**

Es la respuesta a un mensaje PUBLISH con QoS nivel 2. Es el segundo mensaje del flujo de protocolo QoS 2. Un mensaje PUBREC es enviado por el servidor en respuesta a un mensaje PUBLISH de un cliente de publicación, o por un suscriptor en respuesta a un mensaje PUBLISH del servidor.

**I. Cabecera fija**

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*. La longitud de la cabecera variable, *Remaining length*, es de 2 bytes. Puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (5) | | | | DUP flag | QoS level | | RETAIN |
| | 0 | 1 | 0 | 1 | x | x | x | x |
| byte 2 | Remaining Length (2) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figura 6.11.** Cabecera fija de un mensaje PUBREC.

**II. Cabecera variable**

Contiene el identificador de mensaje (Message ID) para el mensaje PUBLISH que se está reconociendo.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message ID MSB | | | | | | | |
| byte 2 | Message ID LSB | | | | | | | |

**Figura 6.12.** Cabecera variable de un mensaje PUBREC.

**III. Payload**

No hay *payload*.

**PUBREL (parte 2)**

Es la respuesta de un publicador a un mensaje PUBREC del servidor o del servidor a un mensaje PUBREC de un suscriptor. Es el tercer mensaje en el flujo del protocolo QoS2.

**I. Cabecera fija**

Los mensajes PUBREL utilizan un *QoS level* de 1 ya que se espera un reconocimiento en forma de PUBCOMP. *DUP flag* se pone a cero (0), esto significa que el mensaje se envía por

primera vez y *RETAIN flag* no se utiliza. La longitud de la cabecera variable, *Remaining length*, es de 2 bytes. Puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (6) | | | | DUP flag | QoS level | | RETAIN |
| | 0 | 1 | 1 | 0 | 0 | 0 | 1 | x |
| byte 2 | Remaining Length (2) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figura 6.13.** Cabecera fija de un mensaje PUBREL.

**II. Cabecera variable**

La cabecera variable contiene el mismo *Message ID* que el mensaje PUBREC que está reconociendo (ver Figura 6.12).

**III. Payload**

No tiene *payload*.

**PUBCOMP (parte 3)**

Este mensaje es la respuesta del servidor a un mensaje PUBREL de un publicador o la respuesta de un suscriptor a un mensaje PUBREL del servidor. Es el cuarto y último mensaje del flujo del protocolo QoS 2.

**I. Cabecera fija**

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*. La longitud de la cabecera variable, *Remaining length*, es de 2 bytes. Puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (7) | | | | DUP flag | QoS level | | RETAIN |
| | 0 | 1 | 1 | 1 | x | x | x | x |
| byte 2 | Remaining Length (2) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figura 6.14.** Cabecera fija de un mensaje PUBCOMP.

**II. Cabecera variable**

La cabecera variable contiene el mismo *Message ID* que el mensaje PUBREL reconocido (ver Figura 6.12).

**III. Payload**

No hay *payload*.

**SUBSCRIBE**

Permite a un cliente registrar su interés en uno o más *topics* con el servidor. Los mensajes publicados sobre estos temas se entregan desde el servidor al cliente como mensajes PUBLISH.

**I. Cabecera fija**

Los mensajes SUBSCRIBE utiliza *QoS level* 1 para reconocer múltiples solicitudes de suscripción. El mensaje SUBACK correspondiente se identifica haciendo coincidir el *Message ID* del mensaje. *DUP flag* se pone a cero (0), esto significa que el mensaje se envía por primera vez y *RETAIN flag* no se utiliza. La longitud del *payload*, *Remaining length*, puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (8) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 0 | 0 | 0 | 0 | 0 | 1 | x |
| byte 2 | Remaining Length | | | | | | | |

**Figura 6.15.** Cabecera fija de un mensaje SUBSCRIBE.

**II. Cabecera variable**

La cabecera variable contiene un *Message ID* porque en este caso el mensaje SUBSCRIBE tiene un *QoS level* de 1.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Message Identifier | | | | | | | | | |
| byte 1 | Message ID MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 2 | Message ID LSB (10) | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

**Figura 6.16.** Cabecera variable de un mensaje SUBSCRIBE.

**III. Payload**

El *payload* de un mensaje SUBSCRIBE contiene una lista de *topics* a los que el cliente desea suscribirse y el nivel de QoS en el que el cliente desea recibir los mensajes. Las cadenas están codificadas en UTF y el nivel de QoS ocupa 2 bits de un solo byte.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Topic name | | | | | | | | | |
| byte 1 | Length MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 2 | Length LSB (3) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| byte 3 | 'a' (0x61) | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| byte 4 | '/' (0x2F) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| byte 5 | 'b' (0x62) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| Requested QoS | | | | | | | | | |
| byte 6 | Requested QoS (1) | x | x | x | x | x | x | 0 | 1 |
| Topic Name | | | | | | | | | |
| byte 7 | Length MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 8 | Length LSB (3) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| byte 9 | 'c' (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| byte 10 | '/' (0x2F) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| byte 11 | 'd' (0x64) | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| Requested QoS | | | | | | | | | |
| byte 12 | Requested QoS (2) | x | x | x | x | x | x | 1 | 0 |

**Figura 6.17.** *Payload* de un mensaje SUBSCRIBE.

**SUBACK**

El servidor envía un mensaje SUBACK al cliente para confirmar la recepción de un mensaje SUBSCRIBE.

**I. Cabecera fija**

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*. La longitud del *payload*, *Remaining length*, puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (9) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 0 | 0 | 1 | x | x | x | x |
| byte 2 | Remaining Length | | | | | | | |

**Figura 6.18.** Cabecera fija de un mensaje SUBACK.

**II. Cabecera variable**

La cabecera variable contiene el *Message ID* para el mensaje SUBSCRIBE que se está reconociendo.

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message ID MSB | | | | | | | |
| byte 2 | Message ID LSB | | | | | | | |

**Figura 6.19.** Cabecera variable de un mensaje SUBACK.

### III. Payload

El *payload* contiene un vector de niveles de QoS concedidos. Cada nivel corresponde a un *topic* en el mensaje SUBSCRIBE correspondiente. El orden de los niveles de QoS en el mensaje SUBACK coincide con el orden del *topic* y los pares de QoS solicitados en el mensaje SUBSCRIBE. El *Message ID* en la cabecera variable le permite hacer coincidir los mensajes SUBACK con los mensajes SUBSCRIBE correspondientes.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| byte 1 | Granted QoS (0) | x | x | x | x | x | x | 0 | 0 |
| byte 1 | Granted QoS (2) | x | x | x | x | x | x | 1 | 0 |

**Figura 6.20.** *Payload* de un mensaje SUBACK.

## UNSUBSCRIBE

El cliente envía un mensaje UNSUBSCRIBE al servidor para cancelar la suscripción a los *topics* nombrados.

### I. Cabecera fija

Los mensajes UNSUBSCRIBE usa el nivel 1 de QoS para reconocer múltiples solicitudes de cancelación de suscripción. El mensaje UNSUBACK correspondiente se identifica mediante el *Message ID*. *DUP flag* se pone a cero (0), esto significa que el mensaje se envía por primera vez y *RETAIN flag* no se utiliza. La longitud del *payload*, *Remaining length*, puede ser un campo de varios bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (10) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 0 | 1 | 0 | 0 | 0 | 1 | x |
| byte 2 | Remaining Length | | | | | | | |

**Figura 6.21.** Cabecera fija de un mensaje UNSUBSCRIBE.

### II. Cabecera variable

La cabecera variable contiene un *Message ID* porque en este caso el mensaje UNSUBSCRIBE tiene un *QoS level* de 1.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Message Identifier | | | | | | | | | |
| byte 1 | Message ID MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 2 | Message ID LSB (10) | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

**Figura 6.22.** Cabecera variable de un mensaje UNSUBSCRIBE.

### III. Payload

El cliente se da de baja de una lista de *topics* nombrados en el *payload*. Las cadenas están codificadas en UTF y se empaquetan de forma contigua.

| | Description | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Topic Name | | | | | | | | | |
| byte 1 | Length MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 2 | Length LSB (3) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| byte 3 | 'a' (0x61) | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| byte 4 | '/' (0x2F) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| byte 5 | 'b' (0x62) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| Topic Name | | | | | | | | | |
| byte 6 | Length MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| byte 7 | Length LSB (3) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| byte 8 | 'c' (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| byte 9 | '/' (0x2F) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| byte 10 | 'd' (0x64) | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

**Figura 6.23**. *Payload* de un mensaje UNSUBSCRIBE.

## UNSUBACK

El servidor envía el mensaje UNSUBACK al cliente para confirmar la recepción de un mensaje UNSUBSCRIBE.

### I. Cabecera fija

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*. La longitud de la cabecera variable, *Remaining length*, es de 2 bytes.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (11) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 0 | 1 | 1 | x | x | x | x |
| byte 2 | Remaining length (2) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figura 6.24.** Cabecera fija de un mensaje UNSUBACK.

### II. Cabecera variable

La cabecera variable contiene el *Message ID* para el mensaje UNSUBSCRIBE que se está reconociendo.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message ID MSB | | | | | | | |
| byte 2 | Message ID LSB | | | | | | | |

**Figura 6.25.** Cabecera variable de un mensaje UNSUBACK.

### III. Payload

No hay *payload*.

## PINGREQ

El mensaje PINGREQ es un "¿estás vivo?" que se envía desde un cliente conectado al servidor.

### I. Cabecera fija

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (12) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 1 | 0 | 0 | x | x | x | x |
| byte 2 | Remaining Length (0) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figura 6.26.** Cabecera fija de un mensaje PINGREQ.

### II. Cabecera variable

No hay cabecera variable.

### III. Payload

No hay *payload*.

## PINGRESP

Un mensaje PINGRESP es la respuesta enviada por un servidor a un mensaje PINGREQ y significa "sí, estoy vivo".

#### I. Cabecera fija

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (13) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 1 | 0 | 1 | x | x | x | x |
| byte 2 | Remaining Length (0) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figura 6.27.** Cabecera fija de un mensaje PINGRESP.

#### II. Cabecera variable

No hay cabecera variable.

#### III. Payload

No hay *payload*.

**DISCONNECT**

El mensaje DISCONNECT se envía desde el cliente al servidor para indicar que está a punto de cerrar su conexión TCP/IP. Esto permite una desconexión limpia, en lugar de simplemente dejar caer la línea.

#### I. Cabecera fija

No se utilizan *QoS level*, *DUP flag* y *RETAIN flag*.

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type (14) | | | | DUP flag | QoS level | | RETAIN |
| | 1 | 1 | 1 | 0 | x | x | x | x |
| byte 2 | Remaining Length (0) | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figura 6.28.** Cabecera fija de un mensaje DISCONNECT.

#### II. Cabecera variable

No hay cabecera variable.

#### III. Payload

No hay *payload*.

### 6.1.3. Calidad de servicio

Aunque TCP/IP garantiza la entrega de datos, el protocolo MQTT utiliza tres niveles de calidad de servicio (QoS) que se debe acordar entre servidor y cliente. Esto se debe a que si la

conexión TCP se rompe se pueden producir pérdidas, por ello aparecen estos distintos niveles para asegurar la entrega correcta de un mensaje:

- ***QoS level 0: At most once delivery.*** El mensaje se entrega según *best-effort* de la red TCP / IP subyacente. No se espera una respuesta y el mensaje llega al servidor una vez o nunca.

| Cliente | Mensaje y dirección | Servidor |
|---|---|---|
| QoS = 0 | PUBLISH<br>→ | **Acción:** Publicar mensaje a suscriptores |

**Tabla 6.3.** Flujo del protocolo de nivel QoS 0.

- ***QoS level 1: At least once delivery.*** La recepción de un mensaje por parte del servidor se confirma mediante un mensaje PUBACK. Si se produce algún error como que el acuse de recibo no se recibe dentro de un período de tiempo específico, el remitente reenvía el mensaje con el bit DUP establecido a uno (1).

  Un mensaje con nivel QoS 1 tiene un ID de mensaje en la cabecera del mensaje.

| Cliente | Mensaje y dirección | Servidor |
|---|---|---|
| QoS = 1<br>DUP = 0<br>ID de mensaje = x<br><br>**Acción:** Almacenar mensaje | PUBLISH<br>→ | **Acciones:**<br>• Almacenar mensaje<br>• Publicar mensaje a suscriptores<br>• Borrar mensaje |
| **Acción:** descartar mensaje | PUBACK<br>← | |

**Tabla 6.4.** Flujo del protocolo de nivel QoS 1.

- ***QoS level 2: Exactly once delivery.*** Con este nivel de QoS se garantiza que los mensajes duplicados no se entreguen a la aplicación receptora. Hay un aumento en el tráfico de la red, pero generalmente es aceptable debido a la importancia del contenido del mensaje.

| Cliente | Mensaje y dirección | Servidor |
|---|---|---|
| QoS = 2<br>DUP = 0<br>ID de mensaje = x<br><br>**Acciones:** Almacenar mensaje | PUBLISH<br>→ | **Acción:** Almacenar mensaje<br><br>o<br><br>**Acciones:**<br>• Almacenar ID de mensaje<br>• Publicar mensaje a suscriptores |
| | PUBREC<br>← | ID de mensaje = x |
| ID de mensaje = x | PUBREL<br>→ | **Acciones:**<br>• Publicar mensaje a suscriptores.<br>• Borrar mensaje<br><br>o<br><br>**Acción:** Borrar ID del mensaje |
| **Acción:** descartar mensaje | PUBCOMP<br>← | ID del mensaje = x |

**Tabla 6.5.** Flujo del protocolo de nivel QoS 2.

## 6.2. OpenFlow

El protocolo de comunicación *Openflow* (OFP) está destinado a redes definidas por *sofware* el cual usa los protocolos TCP/SSL de capas 4 y 5 modelo OSI. Los planos de control y de datos en las redes SND normalmente, se encuentran separados para producir un uso más eficaz de los recursos de la red. Esto en las redes convencionales es algo imposible puesto que las trayectorias de control y datos se producen en el propio dispositivo.

### 6.2.1. Funcionamiento

Este protocolo va a permitir a un servidor poder controlar las rutas que usan los conmutadores para enviar paquetes de forma que el plano de datos continúa siendo la responsabilidad del dispositivo, entre tanto el enrutamiento de alto nivel es dirigido por el servidor.

Componentes principales de OpenFlow:

- **Controlador:** este es el encargado de plantear la red y programar el procesamiento de los flujos de paquetes. Para ello dialoga con los *switches* y así transmitirles la información.
- ***OpenFlow Switch* (OFS):** dispositivo de red que conecta con los equipos finales. Este cuenta con un canal seguro donde comunicarse con el controlador. Contiene las tablas de flujo.
- **Tablas de flujos:** indican la acción a realizar en las entradas de la tabla con cada flujo, de forma que tengan la información necesaria para procesar cualquier tipo de dato.

Para que quede más claro se representa en la Figura 6.29 la arquitectura de OFP. El controlador, en la parte derecha de la imagen, se encarga de generar la información básica que luego se transporta al OFS mediante OFP a través de un canal seguro SSL a nivel *software*. Cuando ya se tienen los datos se pueden crear las tablas de flujo a nivel *hardware* y se añaden las entradas con sus correspondientes acciones para cada flujo.

En caso de que aparezca un flujo que no tenga ninguna coincidencia con la tabla, se envía el paquete al controlador. Éste define un nievo flujo para el paquete y crea una nueva entrada en la tabla. Las nuevas entradas son enviadas al OFS para añadirlas a la tabla y se envía el paquete de vuelta para que pueda ser procesado tras la actualización de la tabla.

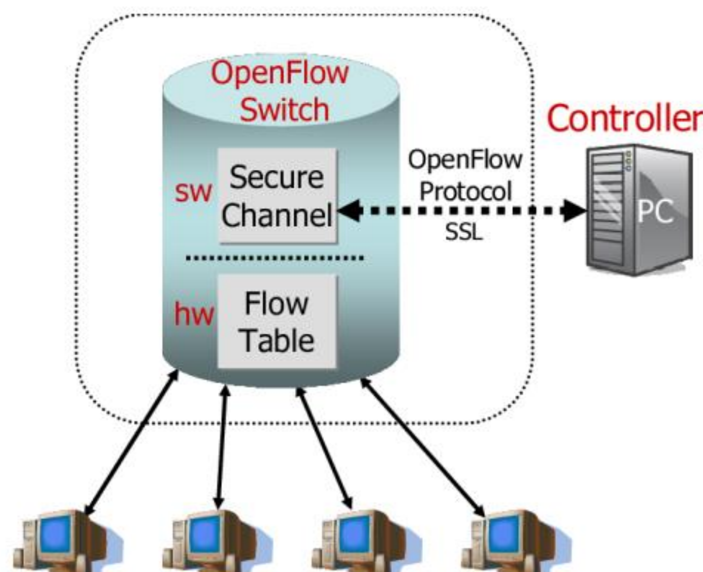


**Figura 6.29.** Arquitectura de OpenFlow [29].

### 6.2.2. Tablas

El procesamiento *pipeline* es aquel soportado por las tablas de flujo OpenFlow que permite al *switch* abarcar numerosas tablas de flujo con varias entradas cada una. Cada tabla actúa de diferente manera ya que depende de las reglas que se le haya indicado.

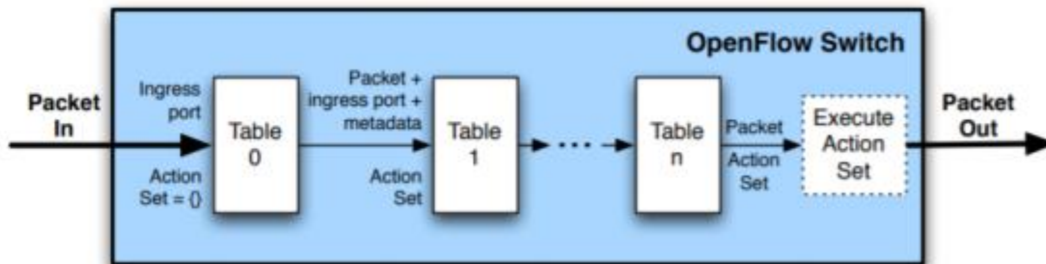


**Figura 6.30.** OpenFlow *pipeline* [30].

Cada entrada de las se componen las entradas de flujo se identifica de forma exclusiva por sus campos *match fields y priority*, los cuales contienen los siguientes elementos:

- ***Match Fields:*** consiste en el puerto de entrada y la cabecera del paquete. Sirven para comprobar coincidencias en la tabla.
- ***Priority:*** número que tiene una entrada para dar preferencia sobre otra del mismo flujo.
- **Counters:** este campo se incrementa cuando hay una coincidencia.
- ***Instructions**:* modifica la acción indicada.
- ***Timeouts*:** tiempo máximo antes de que el flujo expire.
- ***Cookie*:** es un paquete opaco del controlador que se usa para filtrar estadísticas de flujos y procesar paquetes.

| Match Fields | Priority | Counters | Instructions | Timeouts | Cookie |
|---|---|---|---|---|---|

**Figura 6.31.** Componentes de la tabla de flujo.

Se pueden realizar las siguientes acciones con las entradas de las tablas de flujo:

- ***Forwarding*:** los flujos de paquetes se reenvían por uno o varios puertos específicos.
- ***Encrypting*:** los flujos de paquetes se cifran y encapsulan para el controlador.
- ***Drop*:** los flujos de paquetes se borran y descartan.

### 6.2.3. Matching

El proceso que sigue un paquete en el interior OFS empieza con el *switch* buscando coincidencias en la primera tabla de flujo, este continuará buscando imágenes dependiendo si tiene un procesamiento secuencial.

Los *Match Fields* son sacados del paquete para buscar las coincidencias en las direcciones IP, MAC o puertos, normalmente. Si sus valores coinciden con los de una entrada en la tabla de flujo se produce una coincidencia donde se selecciona una entrada con la mayor prioridad, se actualizan los contadores y se lleva a cabo la instrucción indicada. Si la acción indica que hay que redireccionar hacia otra tabla se repite el proceso. Para el caso en el que no exista ninguna coincidencia en la tabla se descarta el paquete.

La siguiente Figura muestra dicho proceso:

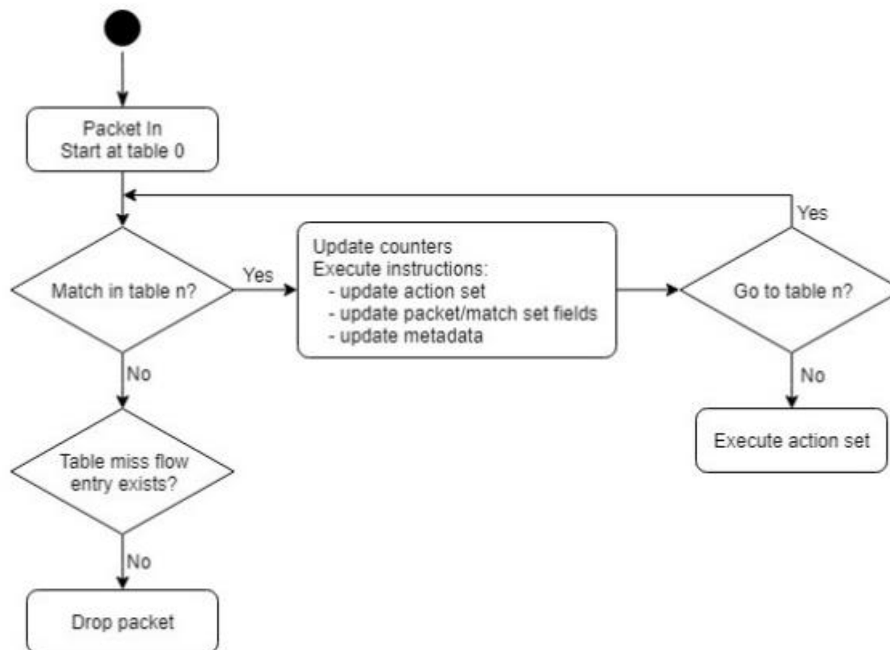


**Figura 6.32.** Diagrama de flujo de un paquete dentro del OFS [30].

### 6.2.4. Mensajes

Los mensajes que se intercambian el controlador y el *switch* a través de la interfaz de OpenFlow pueden ser de tres tipos distintos:

- ***Controller-to-Switch*:** mensajes inicializados por el controlador que no requieren respuesta del switch.
    - ***Features*:** el controlador pide las capacidades del *switch* que debe responder con ellas.

   - ***Configuration***: el controlador envía una petición para cambiar la configuración de un parámetro.
   - ***Modify-State***: el controlador envía este mensaje para modificar el estado de las tablas de flujo o su puerto.
   - ***Read-State***: lo usa el controlador para recoger información de configuración del *switch*.
   - ***Packet-Out***: usados por el controlador para mandar un mensaje completo hacia el exterior.
   - ***Barrier***: se usa para asegurarse de que las dependencias del mensaje has sido completadas.
   - ***Role-Request***: usado por el controlador para marcar el rol de su canal OpenFlow, especialmente cuando hay múltiples controladores.
   - ***Asynchronous-Configuration***: se usa para añadir filtros adicionales a los mensajes asíncronos.
- ***Asyncchronous***: mensajes enviados por los switches sin petición del controlador.
   - ***Packet-in***: para pasar el control de un paquete hacia el controlador, ya sea porque la tabla de flujo lo indique o porque no aparece una entrada con la que asociarse en la tabla.
   - ***Flow-Removed***: informa al controlador que una entrada de la tabla se ha eliminado.
   - ***Port-status***: informa al controlador del cambio de un puerto.
   - ***Error***: cuando se produce un error.
- ***Symmetric***: mensajes que se envían sin solicitud.
   - ***Hello***: estos mensajes se intercambian al iniciar una conexión.
   - ***Echo***: se deben responder por parte del receptor y se envían para asegurar que la conexión sigue activa.
   - ***Experimenter***: se usan para ofrecer funcionalidades adicionales a los mensajes OpenFlow.

# 7. Desarrollo práctico

En este apartado se explica todo el proceso llevado a cabo para llegar a lo que es el objetivo principal de este trabajo, hacer que el controlador sustituya al *broker*.

## 7.1. Estudio de MQTT

Lo primero que se busca hacer en esta parte de desarrollo práctico es implementar un escenario MQTT completo para poder estudiar y comprobar el funcionamiento real de dicho protocolo según lo visto en el apartado de funcionamiento teórico en la sección 6.1.

Se realiza el diseño de una topología de red con la que poder trabajar a lo largo del desarrollo práctico y a continuación se implementa haciendo uso de la herramienta Mininet explicada en la sección 5.1. Acto seguido, se produce el intercambio de mensajes entre los clientes MQTT (suscriptor y publicador) pasando por el *broker* y se hace un estudio de estos con ayuda de la herramienta Wireshark, explicada en la sección 5.3.

### 7.1.1. Diseño de la red

Para el diseño de la red se hace uso de un script escrito en Python con el que Mininet pueda crear la red. La red creada es simple y sencilla, con lo justo para poder comprobar el funcionamiento de MQTT y capturar los mensajes generados por dicho protocolo.

La topología de dicha red incluye tres *switchs* y cuatro *hosts*. Uno de los *switchs* está conectado a los otros dos mientras que estos estarán conectados a su vez a dos *hosts* cada uno. La representación de la topología se muestra en la Figura 7.1.

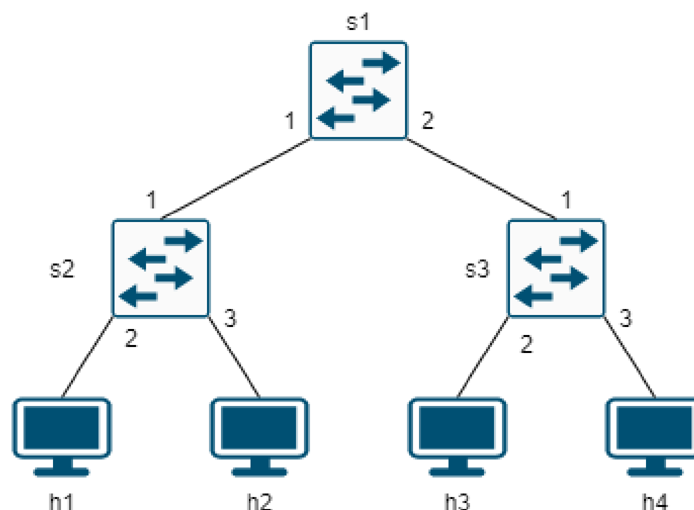


**Figura 7.1.** Topología de red.

El script mencionado anteriormente se muestra en la pág.133. Lo más destacable de este script es la parte en la que se crea la red, «*def myNetwork():*». En esta parte se añade el controlador con la función «*addController()*», los *hosts* con la función «*addHost()*» y los *switchs* con la función «*addSwitch()*». Por último, se crean los enlaces que unen los dispositivos anteriores con la función «*addLink()*» y se pone en marcha la red.

### 7.1.2. Implementación

Este nuevo apartado se enfoca en cómo poner en marcha la red diseñada y de su funcionamiento. Para implementar la red se necesita tener instalado Mininet en nuestro equipo o disponer de una máquina virtual con este software. En mi caso, se utiliza una máquina virtual, proporcionada por el tutor, con sistema operativo Ubuntu además de incluir Ryu y Mininet ya incorporados.

Trabajando ya con dicha máquina virtual, se ejecuta el script con el diseño de la red visto anteriormente. Dicho archivo se guarda en el directorio "*mininet/custom*" y se pone en marcha con el siguiente comando tal y como aparece en la Figura 7.2.

```
jorge@jorge-sdn-vm:~/mininet/custom$ sudo python topo_mqtt_lora_VM_bridged.py
*** Add controller
*** Add hosts
*** Add switches
*** Add links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting controllers
*** Starting CLI:
mininet>
```

**Figura 7.2.** Puesta en marcha de la red.

Ya puesta en marcha la topología de red se pasa a repartir los tres roles principales en un escenario MQTT entre los hosts de esta. Estos roles son los de *broker*, suscriptor y publicador. Los *hosts* que se eligieron fueron h1, h2 y h3, respectivamente. En la Figura 7.3 se muestra dicho reparto.

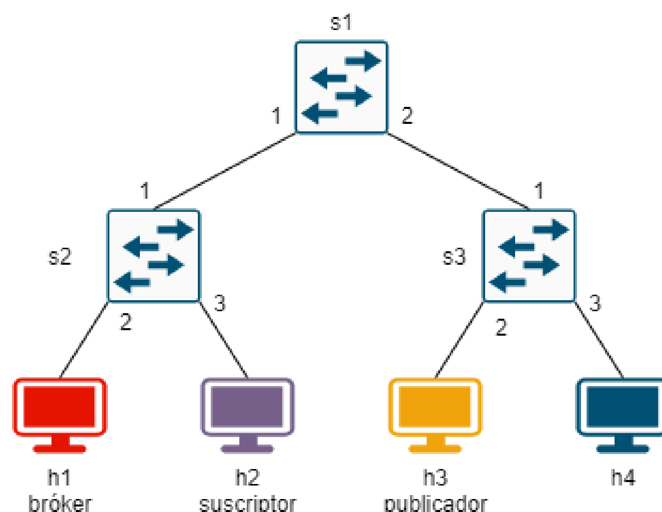


**Figura 7.3.** Topología de red con reparto de roles (*broker*, suscriptor y publicador).

Tal y como se queda la terminal en la Figura 7.2 se procede a introducir el comando "*xterm X*", donde "*X*" se corresponde con el nombre del dispositivo. Con este comando se abre un nuevo terminal específico para cada uno de los *hosts* que se mencionan en el párrafo anterior. A continuación, para cada *host* se introducen los siguientes comandos:

- **En h1:** Se inicia el *broker*.

  *mosquitto -c /etc/mosquitto/conf.d/default.conf*

- **En h2:** Se inicia el cliente que se suscribe al *broker* de h1 (con dirección IP 192.168.1.101) en el tópico "topic".

  *mosquitto_sub -h 192.168.1.101 -t "topic" -u "ricardo" -P "1234"*

- **En h3:** Se inicia el cliente que publica al *broker* de h1 (con dirección IP 192.168.1.101) en el tópico "topic" el mensaje "prueba…".

  *mosquitto_pub -h 192.168.1.101 -t "topic" -u "ricardo" -P "1234" -m "prueba…*

En los comandos anteriores, "-c" especifica el archivo de configuración del *broker*, "-h" especifica el *host* MQTT al que conectarse (por defecto es *localhost*), "-t" especifica el tópico al que suscribirse o al que publicar, "-u" especifica el nombre de usuario, "-P" especifica la contraseña y "-m" especifica el mensaje que envía el publicador. Aunque en este caso no se haya utilizado, también se puede utilizar la opción "-q" para especificar el nivel de calidad de servicio que por defecto es 0. En la Figura 7.4 se muestra el escenario MQTT descrito anteriormente.

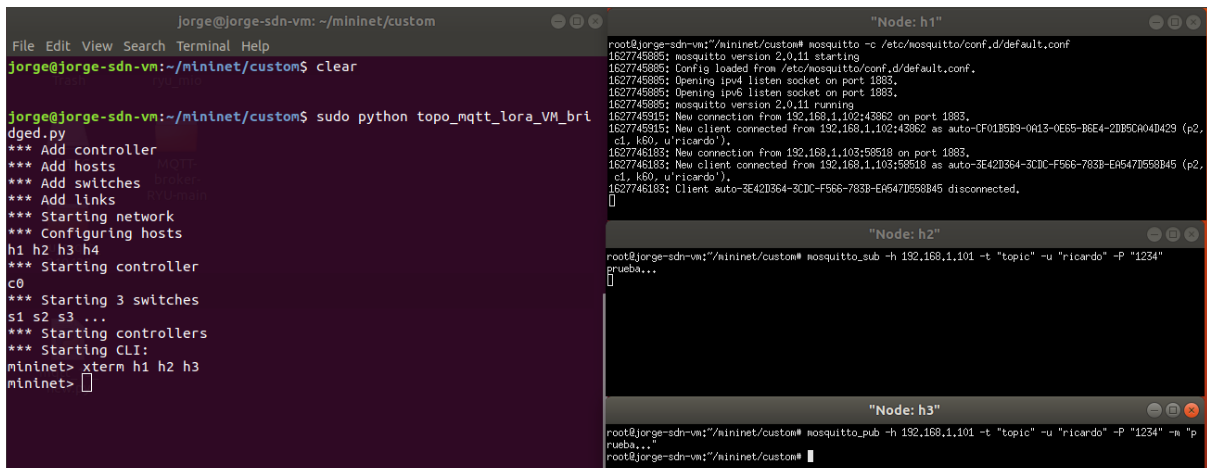


**Figura 7.4.** Escenario MQTT en funcionamiento.

### 7.1.3. Estudio con Wireshark

En este apartado se capturan y analizan los mensajes intercambiados del protocolo MQTT entre *broker*, suscriptor y publicador utilizando la herramienta Wireshark. Las interfaces en las cuales se realizan las capturas son la “s2-eth3” (suscriptor) y la “s3-eth2” (publicador).

Se hace una clasificación según la QoS (0, 1 o 2) que se utilice y para cada una de las QoS se diferencia lo que se recibe y lo que se envía en el suscriptor (dirección IP 192.168.1.102) y el publicador (dirección IP 192.168.1.103).

- **QoS=0**
    - **Suscriptor:** Se manda un mensaje CONNECT al *broker* (dirección IP 192.168.1.101) y este responde con CONNACK si todo ha ido correctamente. Tras esto se manda un mensaje SUBSCRIBE a lo que el *broker* responde con SUBACK.

      Más tarde, el suscriptor recibirá mensajes PUBLISH si el publicador envía algún mensaje asociado al tópico que se suscribió.

      Por último, se observan los mensajes PINGREQ y PINGRES que se utilizan para confirmar que el cliente aún sigue “vivo” tras un periodo de inactividad por su parte. El mensaje DISCONNECT marca el fin de la conexión TCP/IP entre el cliente y el servidor.

mqtt

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 26 | 47.415208038 | 192.168.1.102 | 192.168.1.101 | MQTT | 95 | Connect Command |
| 28 | 47.415567166 | 192.168.1.101 | 192.168.1.102 | MQTT | 70 | Connect Ack |
| 30 | 47.417371020 | 192.168.1.102 | 192.168.1.101 | MQTT | 78 | Subscribe Request (id=1) [topic] |
| 32 | 47.417444693 | 192.168.1.101 | 192.168.1.102 | MQTT | 71 | Subscribe Ack (id=1) |
| 37 | 77.821859275 | 192.168.1.101 | 192.168.1.102 | MQTT | 84 | Publish Message [topic] |
| 49 | 107.857842519 | 192.168.1.102 | 192.168.1.101 | MQTT | 68 | Ping Request |
| 51 | 107.859447667 | 192.168.1.101 | 192.168.1.102 | MQTT | 68 | Ping Response |
| 57 | 119.285384999 | 192.168.1.102 | 192.168.1.101 | MQTT | 68 | Disconnect Req |

**Figura 7.5.** Mensajes recibidos y enviados por el suscriptor con QoS=0.

- **Publicador:** Al igual que con el suscriptor, se manda un mensaje CONNECT al *broker* (dirección IP 192.168.1.101) y este responde con CONNACK si todo ha ido correctamente. Tras esto envía el mensaje PUBLISH y una vez hecho esto se cierra la conexión TCP/IP entre cliente y servidor con el mensaje DISCONECT.

mqtt

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 27 | 77.797938385 | 192.168.1.103 | 192.168.1.101 | MQTT | 95 | Connect Command |
| 29 | 77.815746610 | 192.168.1.101 | 192.168.1.103 | MQTT | 70 | Connect Ack |
| 31 | 77.815830553 | 192.168.1.103 | 192.168.1.101 | MQTT | 84 | Publish Message [topic] |
| 32 | 77.815849440 | 192.168.1.103 | 192.168.1.101 | MQTT | 68 | Disconnect Req |

**Figura 7.6.** Mensajes recibidos y enviados por el publicador con QoS=0.

- **QoS=1**
  - **Suscriptor:** En este caso, la única diferencia que existe con respecto a QoS=0 es que el suscriptor responde con PUBACK a los mensajes PUBLISH reenviados por el *broker*.

mqtt

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 4 | 0.000513861 | 192.168.1.102 | 192.168.1.101 | MQTT | 95 | Connect Command |
| 6 | 0.000723388 | 192.168.1.101 | 192.168.1.102 | MQTT | 70 | Connect Ack |
| 8 | 0.000825366 | 192.168.1.102 | 192.168.1.101 | MQTT | 78 | Subscribe Request (id=1) [topic] |
| 10 | 0.000856551 | 192.168.1.101 | 192.168.1.102 | MQTT | 71 | Subscribe Ack (id=1) |
| 16 | 9.167397935 | 192.168.1.101 | 192.168.1.102 | MQTT | 86 | Publish Message (id=1) [topic] |
| 18 | 9.167449807 | 192.168.1.102 | 192.168.1.101 | MQTT | 70 | Publish Ack (id=1) |
| 20 | 69.242252138 | 192.168.1.102 | 192.168.1.101 | MQTT | 68 | Ping Request |
| 22 | 69.242567489 | 192.168.1.101 | 192.168.1.102 | MQTT | 68 | Ping Response |
| 30 | 98.648413390 | 192.168.1.102 | 192.168.1.101 | MQTT | 68 | Disconnect Req |

**Figura 7.7.** Mensajes recibidos y enviados por el suscriptor con QoS=1.

  - **Publicador:** El único cambio con respecto a QoS=0 es la inclusión del mensaje PUBACK. El publicador manda el mensaje PUBLISH y el *broker* responde con PUBACK si todo ha ido correctamente.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 4 | 0.000586003 | 192.168.1.103 | 192.168.1.101 | MQTT | 95 | Connect Command |
| 6 | 0.000760073 | 192.168.1.101 | 192.168.1.103 | MQTT | 70 | Connect Ack |
| 8 | 0.000858245 | 192.168.1.103 | 192.168.1.101 | MQTT | 86 | Publish Message (id=1) [topic] |
| 10 | 0.000917130 | 192.168.1.101 | 192.168.1.103 | MQTT | 70 | Publish Ack (id=1) |
| 12 | 0.001068178 | 192.168.1.103 | 192.168.1.101 | MQTT | 68 | Disconnect Req |

**Figura 7.8.** Mensajes recibidos y enviados por el publicador con QoS=1.

- **QoS=2**
  - **Suscriptor:** La diferencia que existe con respecto a la QoS=0 es que aparecen el conjunto de mensajes PUBREC, PUBREL y PUBCOMP. Estos son intercambiados entre el suscriptor y el *broker* tras recibirse el mensaje PUBLISH en el suscriptor.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 4 | 0.000444906 | 192.168.1.102 | 192.168.1.101 | MQTT | 95 | Connect Command |
| 6 | 0.000651902 | 192.168.1.101 | 192.168.1.102 | MQTT | 70 | Connect Ack |
| 8 | 0.000849463 | 192.168.1.102 | 192.168.1.101 | MQTT | 78 | Subscribe Request (id=1) [topic] |
| 10 | 0.000899708 | 192.168.1.101 | 192.168.1.102 | MQTT | 71 | Subscribe Ack (id=1) |
| 16 | 7.057546452 | 192.168.1.101 | 192.168.1.102 | MQTT | 86 | Publish Message (id=1) [topic] |
| 18 | 7.057594948 | 192.168.1.102 | 192.168.1.101 | MQTT | 70 | Publish Received (id=1) |
| 20 | 7.057626532 | 192.168.1.101 | 192.168.1.102 | MQTT | 70 | Publish Release (id=1) |
| 22 | 7.057647768 | 192.168.1.102 | 192.168.1.101 | MQTT | 70 | Publish Complete (id=1) |
| 24 | 67.126601300 | 192.168.1.102 | 192.168.1.101 | MQTT | 68 | Ping Request |
| 26 | 67.126943688 | 192.168.1.101 | 192.168.1.102 | MQTT | 68 | Ping Response |
| 32 | 75.694110852 | 192.168.1.102 | 192.168.1.101 | MQTT | 68 | Disconnect Req |

**Figura 7.9.** Mensajes recibidos y enviados por el suscriptor con QoS=2

  - **Publicador:** Ocurre lo mismo que con el suscriptor, aparecen los mensajes PUBREC, PUBREL y PUBCOMP con respecto a la QoS=0.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 4 | 0.000609239 | 192.168.1.103 | 192.168.1.101 | MQTT | 95 | Connect Command |
| 6 | 0.000808918 | 192.168.1.101 | 192.168.1.103 | MQTT | 70 | Connect Ack |
| 8 | 0.000909133 | 192.168.1.103 | 192.168.1.101 | MQTT | 86 | Publish Message (id=1) [topic] |
| 10 | 0.000941320 | 192.168.1.101 | 192.168.1.103 | MQTT | 70 | Publish Received (id=1) |
| 12 | 0.000967952 | 192.168.1.103 | 192.168.1.101 | MQTT | 70 | Publish Release (id=1) |
| 14 | 0.001015953 | 192.168.1.101 | 192.168.1.103 | MQTT | 70 | Publish Complete (id=1) |
| 16 | 0.001216358 | 192.168.1.103 | 192.168.1.101 | MQTT | 68 | Disconnect Req |

**Figura 7.10.** Mensajes recibidos y enviados por el publicador con QoS=2.

## 7.2. Eliminación del bróker

En esta parte se desarrolla el objetivo principal del trabajo, eliminar el *broker* MQTT convencional gracias al uso de la red SDN. Se hace uso del controlador Ryu, cuyo objetivo es adquirir las funcionalidades del *broker*, sustituirlo y hacer que el protocolo MQTT siga funcionando del mismo modo que con *broker*. Se utiliza la topología de red creada en la subsección 7.1.1.

Se crea un script ejecutable (en el directorio "*ryu/ryu/app*") para que el controlador sustituya al *broker* y lo primero que habrá que definir es la dirección IP, la cual no debe pertenecer a ningún equipo. Esta IP se utiliza para identificar al nuevo supuesto *broker* de nuestra red. La dirección IP asignada es la 192.168.1.100 y la dirección MAC 11:22:33:44:55:66.

El controlador se encarga de responder a los mensajes ARP y TCP para que la red funcione con total normalidad. Además, debe recibir y responder a los mensajes del protocolo MQTT procedentes de los suscriptores y publicadores de la red.

A continuación, se analizan los mensajes de los protocolos que intervienen en este escenario, se crean los paquetes para el funcionamiento de la red y de MQTT y, por último, se ve como el controlador analiza y clasifica los mensajes que recibe.

### 7.2.1. Análisis de los mensajes

El análisis de los mensajes de los distintos protocolos que participan en un escenario MQTT consiste en estudiar detenidamente sus campos y cómo se componen. Esto se llevará a cabo utilizando Wireshark.

Se toman como referencia, y para que sea más dinámico, el mensaje inicial de cada protocolo con el objetivo de analizar los campos más significativos de cada uno de ellos. En el caso de MQTT, aunque se tiene más variedad de mensajes que en el caso de ARP o TCP, se analiza solo también el primer mensaje.

**ARP**

El protocolo ARP se usa para obtener la dirección MAC respectiva a una dirección IP. En el caso de la Figura 7.11, en el momento que el cliente se quiera conectar al *broker* pero no conozca su dirección MAC, debe enviar un ARP Request utilizando la dirección de broadcast de ARP ff:ff:ff:ff:ff:ff como dirección de destinatario. Tras esto, cada una de las estaciones de la red compara la dirección IP indicada en la solicitud con la suya propia y si coincide emite un mensaje ARP Reply que incorpora entre otros datos su dirección MAC.

El formato del paquete ARP para direcciones IPv4 en redes Ethernet sigue la siguiente estructura: los primeros 14 bytes marcados en verde corresponden a la capa de enlace del modelo OSI y a partir de aquí al protocolo ARP marcado en azul. Esto se puede apreciar tanto en la captura de Wireshark, Figura 7.11, como en el esquema de formato del paquete ARP, Figura 7.12.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 21 | 47.403107285 | 00:00:00_00:00:02 | Broadcast | ARP | 42 | Who has 192.168.1.101? Tell 192.168.1.102 |
| 22 | 47.410911555 | 00:00:00_00:00:01 | 00:00:00_00:00:02 | ARP | 42 | 192.168.1.101 is at 00:00:00:00:00:01 |
| 34 | 52.480410175 | 00:00:00_00:00:01 | 00:00:00_00:00:02 | ARP | 42 | Who has 192.168.1.102? Tell 192.168.1.101 |
| 35 | 52.480423969 | 00:00:00_00:00:02 | 00:00:00_00:00:01 | ARP | 42 | 192.168.1.102 is at 00:00:00:00:00:02 |
| 36 | 77.781267820 | 00:00:00_00:00:03 | Broadcast | ARP | 42 | Who has 192.168.1.101? Tell 192.168.1.103 |
| 39 | 82.944496617 | 00:00:00_00:00:01 | 00:00:00_00:00:02 | ARP | 42 | Who has 192.168.1.102? Tell 192.168.1.101 |
| 40 | 82.944531056 | 00:00:00_00:00:02 | 00:00:00_00:00:01 | ARP | 42 | 192.168.1.102 is at 00:00:00:00:00:02 |
| 53 | 112.896705538 | 00:00:00_00:00:02 | 00:00:00_00:00:01 | ARP | 42 | Who has 192.168.1.101? Tell 192.168.1.102 |
| 54 | 112.896880277 | 00:00:00_00:00:01 | 00:00:00_00:00:02 | ARP | 42 | Who has 192.168.1.102? Tell 192.168.1.101 |
| 55 | 112.896893963 | 00:00:00_00:00:02 | 00:00:00_00:00:01 | ARP | 42 | 192.168.1.102 is at 00:00:00:00:00:02 |
| 56 | 112.896914002 | 00:00:00_00:00:01 | 00:00:00_00:00:02 | ARP | 42 | 192.168.1.101 is at 00:00:00:00:00:01 |

```
▸ Frame 21: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▾ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: 00:00:00_00:00:02 (00:00:00:00:00:02)
    Sender IP address: 192.168.1.102
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.1.101

0000  ff ff ff ff ff ff 00 00  00 00 00 02 08 06 00 01   ········ ········
0010  08 00 06 04 00 01 00 00  00 00 00 02 c0 a8 01 66   ········ ·······f
0020  00 00 00 00 00 00 c0 a8  01 65                     ········ ·e
```

**Figura 7.11.** Conjunto de mensajes ARP capturados en Wireshark.

Los bytes indicados en rojo en la Figura 7.12 se han indicado tomando como ejemplo el mensaje ARP marcado en Wireshark en la Figura 7.11. Para el mensaje ARP Request el campo Operation code es 1 y para ARP Reply 2.

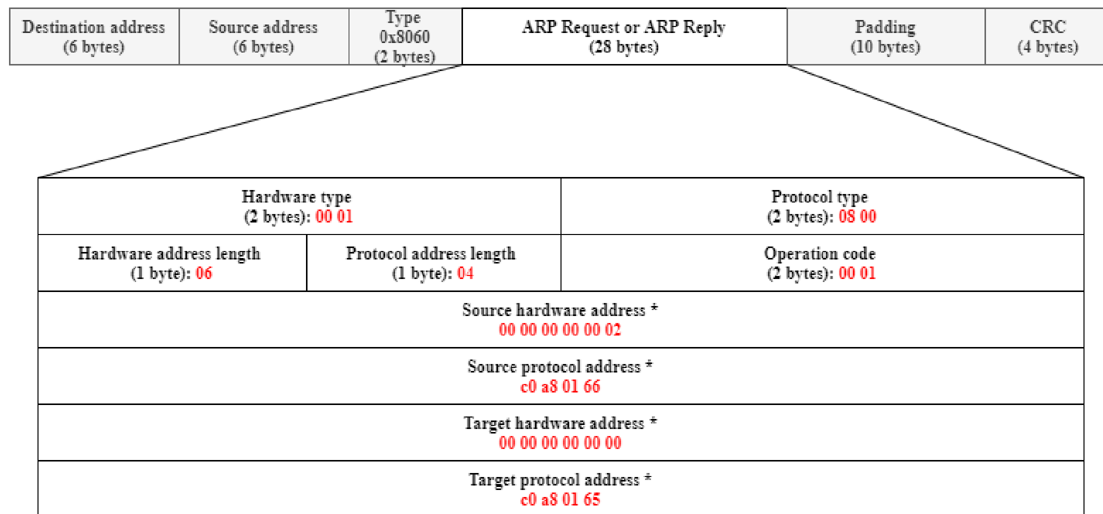


**Figura 7.12.** Formato de paquete ARP.

## TCP

El protocolo TCP (Transmission Control Protocol) se encarga de establecer y permitir la conexión entre dos hosts y asegurar el intercambio de datos entre ellos. Esto se lleva a cabo con un "*3-way handshake*" en el que participan los mensajes SYN, SYN/ACK y ACK, tal y como se aprecia en la Figura 7.13.

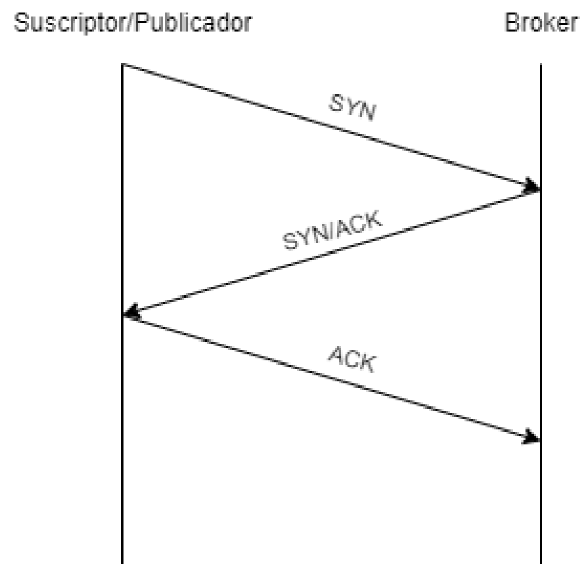


**Figura 7.13.** Esquema del “3-way handshake”.

A continuación, en la Figura 7.14, se observa un ejemplo de esta sucesión de mensajes del protocolo TCP. En concreto, se ha marcado el mensaje SYN que comienza con los 14 bytes marcados en verde correspondientes al protocolo Ethernet (capa de enlace). Le siguen 20 bytes marcados en rojo correspondientes al protocolo IP (capa de red), de los cuales los primeros indican la versión, longitudes del mensaje, ID (identificación) del mensaje, protocolo siguiente (el 6 indica TCP) o el *checksum* (suma de verificación), entre otros. Los últimos 8 bytes indican las direcciones IP origen y destino.

La parte final del mensaje marcada en azul corresponde al protocolo TCP. Apoyándose junto con la Figura 7.15 se pueden distinguir muy bien que bytes corresponden con que campos del paquete TCP. Los 4 primeros bytes indican el puerto de origen y de destino, le sigue el número de secuencia y el número ack, y un conjunto de *flags* que se encargan de indicar la función del mensaje. El paquete finaliza con el tamaño de ventana, *cheksum* y puntero urgente (si URG es establecido).

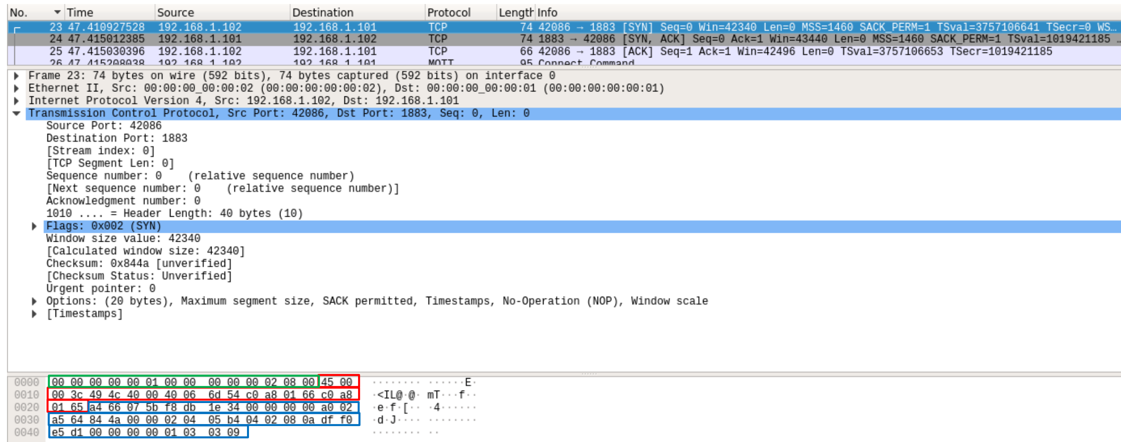


**Figura 7.14.** Conjunto de mensajes TCP capturados en Wireshark.

<table>
<tr><td colspan="11">Source Port Number<br>(16 bits): a4 66</td><td>Destination Port Number<br>(16 bits): 07 5b</td></tr>
<tr><td colspan="12">Sequence Number<br>(32 bits): f8 db 1e 34</td></tr>
<tr><td colspan="12">Acknowledgement Number<br>(32 bits): 00 00 00 00</td></tr>
<tr><td>Header Length<br>(4 bits):<br>1010</td><td>Reserved<br>(3 bits):<br>000</td><td>NS<br>0</td><td>CWR<br>0</td><td>ECE<br>0</td><td>UGR<br>0</td><td>ACK<br>0</td><td>PSH<br>0</td><td>RST<br>0</td><td>SYN<br>1</td><td>FIN<br>0</td><td>Windows Size<br>(16 bits): a5 64</td></tr>
<tr><td colspan="11">a0 02<br>TCP Checksum<br>(16 bits): 84 4a</td><td>Urgent Pointer<br>(16 bits): 00 00</td></tr>
<tr><td colspan="12">Options (if any)</td></tr>
<tr><td colspan="12">Data (if any)</td></tr>
</table>

**Figura 7.15.** Formato de paquete TCP.

### MQTT

Los mensajes MQTT ya se han analizado en el apartado de teoría 6.1, por lo tanto solo se van a señalar los campos más importantes para la hora de crear las respuestas respectivas a este tipo de mensajes.

El paquete señalado en la Figura 7.16, correspondiente al mensaje *CONNECT*, comienza con los campos pertenecientes al protocolo Ethernet, IP y TCP (recuadrados en color verde, rojo y azul respectivamente). La última parte del paquete depende del tipo de mensaje MQTT, en este caso se incluye el nombre del protocolo, la versión, el identificador, nombre de usuario o contraseña.

```
mqtt
No.   Time          Source         Destination    Protocol  Length Info
   26 47.415208038  192.168.1.102  192.168.1.101  MQTT      95 Connect Command
   28 47.415567166  192.168.1.101  192.168.1.102  MQTT      70 Connect Ack
   30 47.417371020  192.168.1.102  192.168.1.101  MQTT      78 Subscribe Request (id=1) [topic]
   32 47.417444693  192.168.1.101  192.168.1.102  MQTT      71 Subscribe Ack (id=1)
   37 77.821859275  192.168.1.101  192.168.1.102  MQTT      84 Publish Message [topic]
   49 107.857842519 192.168.1.102  192.168.1.101  MQTT      68 Ping Request
   51 107.859447667 192.168.1.101  192.168.1.102  MQTT      68 Ping Response
   57 119.285384999 192.168.1.102  192.168.1.101  MQTT      68 Disconnect Req
▸ Frame 26: 95 bytes on wire (760 bits), 95 bytes captured (760 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
▸ Internet Protocol Version 4, Src: 192.168.1.102, Dst: 192.168.1.101
▸ Transmission Control Protocol, Src Port: 42086, Dst Port: 1883, Seq: 1, Ack: 1, Len: 29
▾ MQ Telemetry Transport Protocol, Connect Command
  ▸ Header Flags: 0x10, Message Type: Connect Command
    Msg Len: 27
    Protocol Name Length: 4
    Protocol Name: MQTT
    Version: MQTT v3.1.1 (4)
  ▸ Connect Flags: 0xc2, User Name Flag, Password Flag, QoS Level: At most once delivery (Fire and Forget), Clean Session Flag
    Keep Alive: 60
    Client ID Length: 0
    Client ID:
    User Name Length: 7
    User Name: ricardo
    Password Length: 4
    Password: 1234
0000  00 00 00 00 00 01 00 00  00 00 00 02 08 00 45 00   ········ ······E·
0010  00 51 49 4e 40 00 40 06  6d 3d c0 a8 01 66 c0 a8   ·QIN@·@· m=···f··
0020  01 65 a4 66 07 5b f8 db  1e 35 ce 29 38 16 80 18   ·e·f·[·· ·5·)8···
0030  00 53 84 5f 00 00 01 01  08 0a df f0 e5 dd 3c c3   ·S·_···· ······<·
0040  22 01 10 1b 00 04 4d 51  54 54 04 c2 00 3c 00 00   "·····MQ TT···<··
0050  00 07 72 69 63 61 72 64  6f 00 04 31 32 33 34      ··ricard o··1234
```

**Figura 7.16.** Conjunto de mensajes MQTT capturados en Wireshark.

### 7.2.2. Creación de mensajes de red.

Tras haber analizado los paquetes con los que debe de tratar el *broker*, se pasa a crear los mensajes que debe enviar el controlador de la red SDN para dar respuesta a los mensajes que le llegan de parte de los clientes. Al desaparecer la imagen del *broker*, se debe especificar una dirección IP y MAC en el controlador para que los clientes sepan a qué direcciones dirigirse. Las direcciones elegidas son 192.168.1.100 y 11:22:33:44:55:66, respectivamente. Aun así, se irán recordando a lo largo de la memoria.

**ARP REPLY**

El mensaje ARP Reply se envía cuando en el controlador se recibe un ARP Request. La función encargada, *«_handle_arp»*, es llamada previa comprobación en el programa principal de que el paquete recibido contiene el protocolo ARP y que la IP de destino coincide con la del controlador.

Lo primero que se hace en la función *«_handle_arp»* es verificar que el paquete recibido es del tipo ARP Request, si no es así se saldrá de la función y se continuará con el programa principal. Lo siguiente es informar, a través de la terminal que se ejecuta el controlador, de que se ha recibido un ARP Request y se crea un nuevo paquete al que se le añaden las cabeceras de los protocolos Ethernet y ARP.

En la cabecera Ethernet se especifican la MAC de destino que coincide con la que originó el mensaje ARP Request y la MAC de origen que se corresponde con la del controlador, la 11:22:33:44:55:66.

Por otro lado en la cabecera ARP, se indican tanto las direcciones MAC como las direcciones IP de origen y destinatario además de especificar que se trata de un mensaje del tipo ARP Reply. Por último, se envía el paquete ARP Reply creado y se notifica por la terminal.

```
### 2021/08/15 19:12:57.472250 > ARP packet request received from 192.168.1.102
### 2021/08/15 19:12:57.472413 > ARP packet reply sent to 192.168.1.102
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2054,src='11:22:33:44:55:66'), arp(dst_ip='192.168.1.102',dst_mac='00:00:0
0:00:00:02',hlen=6,hwtype=1,opcode=2,plen=4,proto=2048,src_ip='192.168.1.100',src_mac='11:22:33:44:55:66')
```

**Figura 7.17.** Mensajes mostrados en la terminal cuando se recibe un ARP Request y se responde con un ARP Reply.

En la Figura 7.18, que se corresponde con una captura de Wireshark en la interfaz "s2-eth3" (suscriptor), se puede apreciar cómo se obtiene una respuesta prácticamente idéntica al caso reflejado en la Figura 7.11 en la que si se utiliza *broker*.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 11 | 6.092385673 | 00:00:00_00:00:02 | Broadcast | ARP | 42 | Who has 192.168.1.100? Tell 192.168.1.102 |
| 12 | 6.094743976 | 11:22:33:44:55:66 | 00:00:00_00:00:02 | ARP | 60 | 192.168.1.100 is at 11:22:33:44:55:66 |

```
▸ Frame 12: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
▾ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
  ▸ Destination: 00:00:00_00:00:02 (00:00:00:00:00:02)
  ▸ Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
    Type: ARP (0x0806)
    Padding: 000000000000000000000000000000000000
▾ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: 11:22:33:44:55:66 (11:22:33:44:55:66)
    Sender IP address: 192.168.1.100
    Target MAC address: 00:00:00_00:00:02 (00:00:00:00:00:02)
    Target IP address: 192.168.1.102

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 06 00 01   ·······" 3DUf····
0010  08 00 06 04 00 02 11 22  33 44 55 66 c0 a8 01 64   ·······" 3DUf···d
0020  00 00 00 00 00 02 c0 a8  01 66 00 00 00 00 00 00   ········ ·f······
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

**Figura 7.18.** Mensaje ARP Reply creado por el controlador y capturado con Wireshark.

**TCP SYN ACK**

El mensaje TCP SYN ACK se envía cuando el controlador recibe un TCP SYN. La función encargada, *«_handle_tcp»*, es llamada previa comprobación en el programa principal de que el paquete recibido contiene el protocolo TCP, que la IP de destino coincide con la del controlador y que el puerto de origen o destino sea el 1883 (MQTT).

Lo primero que se hace en la función *«_handle_tcp»* es verificar que el paquete recibido es del tipo TCP SYN, si no es así se saldrá de la función y se continuará con el programa principal. Lo siguiente es informar, a través de la terminal que se ejecuta el controlador, de que se ha recibido un TCP SYN y se llama a la función *«generate_tcp_ack»* para crear un nuevo paquete al que se le añaden las cabeceras de los protocolos Ethernet, IP y TCP.

En la cabecera Ethernet se especifican la MAC de destino que coincide con la que originó el mensaje TCP SYN y la MAC de origen que se corresponde con la del controlador, la 11:22:33:44:55:66.

En la cabecera IP se especifican la dirección IP de destino que coincide de nuevo con la que originó el mensaje TCP SYN y origen que se corresponde con la del controlador, la 192.168.1.100. En esta cabecera también se indica el protocolo que le sigue, TCP (6) en este caso.

Para la cabecera TCP se genera previamente un número de secuencia (*sequence number*) aleatorio de 32 bits y otro número de confirmación (*acknowledgment number*) sumándole uno al número de secuencia del mensaje TCP anterior. Se comprueban y añaden las opciones TCP y los *flags* continúan siendo los mismos, pero se le añade el correspondiente al ACK. Esta cabecera incluye todos estos campos más los correspondientes al puerto origen, puerto destino, tamaño de ventana y puntero urgente entre otros. Por último, se envía el paquete TCP SYN ACK creado y se notifica por la terminal.

```
######### TCP PARSER
######### TCP INIT
##### 2021/08/15 19:12:57.476672 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv
4(csum=23719,dst='192.168.1.100',flags=2,header_length=5,identification=23034,offset=0,option=None,proto=6,src='192.168.1.102',t
os=0,total_length=60,ttl=64,version=4), tcp(ack=0,bits=2,csum=40260,dst_port=1883,offset=10,option=[TCPOptionMaximumSegmentSize(
kind=2,length=4,max_seg_size=1460), TCPOptionSACKPermitted(kind=4,length=2), TCPOptionTimestamps(kind=8,length=10,ts_ecr=0,ts_va
l=2412240022), TCPOptionNoOperation(kind=1,length=1), TCPOptionWindowScale(kind=3,length=3,shift_cnt=9)],seq=567489889,src_port=
46668,urgent=0,window_size=42340)]
### 2021/08/15 19:12:57.478730 > TCP SYN received by the MQTT broker APP from 192.168.1.102
######### TCP INIT
### 2021/08/15 19:12:57.479231 > TCP SYN+ACK sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={}
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14241,dst='192.168.1.102',flags=0
,header_length=5,identification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=60,ttl=255,version=4), tcp
(ack=567489890,bits=18,csum=39614,dst_port=46668,offset=10,option=[TCPOptionMaximumSegmentSize(kind=2,length=4,max_seg_size=1460
), TCPOptionSACKPermitted(kind=4,length=2), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412240022,ts_val=2412240023), TCPOption
NoOperation(kind=1,length=1), TCPOptionWindowScale(kind=3,length=3,shift_cnt=9)],seq=2243039331,src_port=1883,urgent=0,window_si
ze=42340)
```

**Figura 7.19.** Mensajes mostrados en la terminal cuando se recibe un TCP SYN y se responde con un TCP SYN+ACK.

En la Figura 7.20, que se corresponde con una captura de Wireshark en la interfaz “s2-eth3” (suscriptor), se puede apreciar cómo se obtiene una respuesta prácticamente idéntica al caso reflejado en la Figura 7.14 en la que si se utiliza *broker*.

```
No.  Time         Source         Destination    Protocol  Length Info
  13 6.094752226  192.168.1.102  192.168.1.100  TCP       74 52944 → 1883 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=3446066914 TSecr=0 WS=…
  14 6.102720827  192.168.1.100  192.168.1.102  TCP       74 1883 → 52944 [SYN, ACK] Seq=0 Ack=1 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=3446066915…
  15 6.102741000  192.168.1.102  192.168.1.100  TCP       66 52944 → 1883 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=3446066924 TSecr=3446066915

▸ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▾ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 60
    Identification: 0x0000 (0)
  ▸ Flags: 0x0000
    Time to live: 255
    Protocol: TCP (6)
    Header checksum: 0x37a1 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.1.100
    Destination: 192.168.1.102
▾ Transmission Control Protocol, Src Port: 1883, Dst Port: 52944, Seq: 0, Ack: 1, Len: 0
    Source Port: 1883
    Destination Port: 52944
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 0    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    1010 .... = Header Length: 40 bytes (10)
  ▸ Flags: 0x012 (SYN, ACK)

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00
0010  00 3c 00 00 00 00 ff 06  37 a1 c0 a8 01 64 c0 a8
0020  01 66 07 5b ce d0 aa f0  3e 38 11 ef 96 5f a0 12
0030  a5 64 7e 37 00 00 02 04  05 b4 04 02 08 0a cd 66
0040  ce e3 cd 66 ce e2 01 03  03 09
```

**Figura 7.20.** Mensaje TCP SYN ACK creado por el controlador y capturado con Wireshark.

### TCP FIN ACK

El mensaje TCP FIN ACK se envía cuando el controlador recibe un TCP FIN. Al igual que en el caso anterior, esta función *«_handle_tcp»* es llamada previa comprobación en el programa principal de que el paquete recibido contiene el protocolo TCP, que la IP de destino coincide con la del controlador y que el puerto de origen o destino sea el 1883 (MQTT).

Lo primero que se hace en la función *«_handle_tcp»* es verificar que el paquete recibido es del tipo TCP FIN, si no es así se saldrá de la función y se continuará con el programa principal. Lo siguiente es informar, a través de la terminal que se ejecuta el controlador, de que se ha recibido un TCP FIN y se borra el tópico y el suscriptor (dirección IP + puerto TCP) tanto del diccionario «topicsSubscribers», «subscribersTCPInfo» y «subscribersTopics». Tras ello, se muestran los diccionarios actualizados por la terminal y se llama a la función *«generate_tcp_ack»* para crear un nuevo paquete al que se le añaden las cabeceras de los protocolos Ethernet, IP y TCP.

En cuanto a las cabeceras se generan igual que para el mensaje TCP SYN ACK excepto que el número de secuencia se iguala al número de confirmación del mensaje anterior y el número de confirmación se calcula como el número de secuencia del mensaje anterior más uno y el tamaño del *payload* TCP. Por último, se envía el paquete TCP FIN ACK creado y se notifica por la terminal.

```
##### 2021/08/15 19:13:11.423619 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:03'), ipv
4(csum=23009,dst='192.168.1.100',flags=2,header_length=5,identification=23749,offset=0,option=None,proto=6,src='192.168.1.103',t
os=0,total_length=54,ttl=64,version=4), tcp(ack=4264153383,bits=25,csum=46181,dst_port=1883,offset=8,option=[TCPOptionNoOperatio
n(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=1895634150,ts_val=1895634
172)],seq=4108383791,src_port=36930,urgent=0,window_size=83), mqtt(mqttControlPacketType=14,mqttRemainingLength=0)]
### 2021/08/15 19:13:11.429118 > TCP FIN received by the MQTT broker APP from 192.168.1.103
######### topicsSubscribers:  {b'topic': [['192.168.1.102-46668', 0]]}
######### subscribersTopics:  {'192.168.1.102-46668': [b'topic']}
######### subscribersTCPInfo: {'192.168.1.102-46668': [2243039343, 567489931, 2412240117]}
######### TCP INIT
### 2021/08/15 19:13:11.430317 > TCP FIN+ACK sent to 192.168.1.103
######### subscriber: 192.168.1.103-36930, subscribersTCPInfo={'192.168.1.102-46668': [2243039343, 567489931, 2412240117]}
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:03',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14248,dst='192.168.1.103',flags=0
,header_length=5,identification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=52,ttl=255,version=4), tcp
(ack=4108383794,bits=25,csum=37966,dst_port=36930,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(k
ind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=1895634172,ts_val=1895634173)],seq=4264153383,src_port=1883,urgent=
0,window_size=83)
```

**Figura 7.21.** Mensajes mostrados en la terminal cuando se recibe un TCP FIN por parte del publicador y se responde con un TCP FIN+ACK.

```
######### TCP PARSER
######### TCP INIT
##### 2021/08/15 19:14:39.052486 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv
4(csum=23717,dst='192.168.1.100',flags=2,header_length=5,identification=23044,offset=0,option=None,proto=6,src='192.168.1.102',t
os=0,total_length=52,ttl=64,version=4), tcp(ack=2243039361,bits=17,csum=63195,dst_port=1883,offset=8,option=[TCPOptionNoOperatio
n(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412300102,ts_val=2412341
596)],seq=567489935,src_port=46668,urgent=0,window_size=83)]
### 2021/08/15 19:14:39.054410 > TCP FIN received by the MQTT broker APP from 192.168.1.102
######### topicsSubscribers:  {}
######### subscribersTopics:  {}
######### subscribersTCPInfo: {}
######### TCP INIT
### 2021/08/15 19:14:39.055133 > TCP FIN+ACK sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={}
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14249,dst='192.168.1.102',flags=0
,header_length=5,identification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=52,ttl=255,version=4), tcp
(ack=567489936,bits=17,csum=21699,dst_port=46668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(ki
nd=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412341596,ts_val=2412341597)],seq=2243039361,src_port=1883,urgent=0
,window_size=83)
```

**Figura 7.22.** Mensajes mostrados en la terminal cuando se recibe un TCP FIN por parte del suscriptor y se responde con un TCP FIN+ACK.

En la Figura 7.23, que se corresponde con una captura de Wireshark en la interfaz "s2-eth3" (suscriptor), se puede apreciar cómo se obtiene una respuesta prácticamente idéntica al caso reflejado en la Figura 7.14 en la que si se utiliza *broker*.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 25 | 20.271270287 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 52944 → 1883 [FIN, ACK] Seq=44 Ack=28 Win=42496 Len=0 TSval=3446081093 TSecr=3446066960 |
| 26 | 20.283389087 | 192.168.1.100 | 192.168.1.102 | TCP | 66 | 1883 → 52944 [FIN, ACK] Seq=28 Ack=45 Win=42496 Len=0 TSval=3446081094 TSecr=3446081093 |
| 27 | 20.283407555 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 52944 → 1883 [ACK] Seq=45 Ack=29 Win=42496 Len=0 TSval=3446081105 TSecr=3446081094 |

```
▸ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▾ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0x0000 (0)
  ▸ Flags: 0x0000
    Time to live: 255
    Protocol: TCP (6)
    Header checksum: 0x37a9 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.1.100
    Destination: 192.168.1.102
▾ Transmission Control Protocol, Src Port: 1883, Dst Port: 52944, Seq: 28, Ack: 45, Len: 0
    Source Port: 1883
    Destination Port: 52944
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 28    (relative sequence number)
    [Next sequence number: 28    (relative sequence number)]
    Acknowledgment number: 45    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x011 (FIN, ACK)

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00   ·······" 3DUf··E·
0010  00 34 00 00 00 00 ff 06  37 a9 c0 a8 01 64 c0 a8   ·4······ 7····d··
0020  01 66 07 5b ce d0 aa f0  3e 54 11 ef 96 8b 80 11   ·f·[···· >T······
0030  00 53 e3 08 00 00 01 01  08 0a cd 67 06 46 cd 67   ·S······ ···g·F·g
0040  06 45                                              ·E
```

**Figura 7.23.** Mensaje TCP FIN ACK creado por el controlador y capturado con Wireshark.

### 7.2.3. Creación de mensajes MQTT

Este apartado se centra en la creación de los mensajes del protocolo MQTT más básicos e indispensables a la hora de poner en marcha una red en la que se utilice dicho protocolo.

La función encargada de esto, *«_handle_mqtt»*, es llamada previa comprobación en el programa principal de que el paquete recibido contiene el protocolo MQTT. Aunque dicha comprobación es posterior a las relacionadas con el protocolo IP, TCP, dirección IP de destino que coincida con la del controlador y que el puerto de origen o de destino sea el 1883.

Ya dentro de la función *«_handle_mqtt»* se notifica por la terminal que se ha recibido un paquete TCP, el tipo de *payload*, el paquete MQTT y el tipo, y el contenido del paquete MQTT en bytes. Tras ello, se pasa a comprobar el tipo de mensaje MQTT.

**CONNACK**

Como se está tratando con la respuesta CONNACK el tipo de mensaje recibido debe ser CONNECT, cuyo código coincide con 1.

En este caso, se pasa a preparar un paquete y añadirle las cabeceras Ethernet, IP y TCP de forma similar a lo ya explicado justo en el apartado anterior. Luego se notifica por la terminal que se ha recibido un mensaje CONNECT, se comprueba si el usuario y la contraseña son correctos, y se añade la cabecera MQTT al paquete creado con anterioridad. En la cabecera se añade el tipo de paquete MQTT (tipo 2 para este caso), la longitud restante (2 para este caso) y el código de retorno (5 si el usuario o la contraseña no coinciden y 0 en caso de que coincidan ambos). Por último, se envía el paquete CONNACK creado y se notifica de ello a través de la terminal.

```
##### 2021/08/15 19:12:57.501177 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv
4(csum=23696,dst='192.168.1.100',flags=2,header_length=5,identification=23036,offset=0,option=None,proto=6,src='192.168.1.102',t
os=0,total_length=81,ttl=64,version=4), tcp(ack=2243039332,bits=24,csum=38415,dst_port=1883,offset=8,option=[TCPOptionNoOperatio
n(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412240023,ts_val=2412240
034)],seq=567489890,src_port=46668,urgent=0,window_size=83), mqtt(mqttControlPacketType=1,mqttRemainingLength=27), b'\x00\x04MQT
T\x04\xc2\x00<\x00\x00\x00\x07ricardo\x00\x041234']
##### 2021/08/15 19:12:57.508207 > Received TCP packet (seq=567489890, ack=2243039332), payload type: <class 'ryu.lib.packet.mqt
t.mqtt'>, MQTT packet: mqtt(mqttControlPacketType=1,mqttRemainingLength=27), MQTT packet type: 1 (CONNECT), MQTT packet content
(29 bytes): b'\x10\x1b\x00\x04MQTT\x04\xc2\x00<\x00\x00\x00\x07ricardo\x00\x041234'
######### TCP INIT
### 2021/08/15 19:12:57.508659 > MQTT CONNECT (protocolName=b'MQTT', version=4, connectionFlags=11000010, keepAlive=60, clientID
=b'', willTopic=None, willMessage=None, userName=b'ricardo', password=b'1234') received from 192.168.1.102
######### MQTT INIT (mqttControlPacketType=2, mqttRemainingLength=2, returnCode=None, messageID=None, qos=None, topic=None, mess
age=None) #########
### 2021/08/15 19:12:57.508659 > MQTT CONNACK (returnCode=0) sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={}
######### MQTT SERIALIZE #########
######### MQTT SERIALIZE HEADER (mqttByte1=32, mqttRLByte1=2) #########
######### MQTT SERIALIZE CONNACK (mqttControlPacketType=2, returnCode=0) #########
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14245,dst='192.168.1.102',flags=0
,header_length=5,identification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=56,ttl=255,version=4), tcp
(ack=567489919,bits=24,csum=20059,dst_port=46668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(ki
nd=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412240034,ts_val=2412240035)],seq=2243039332,src_port=1883,urgent=0
,window_size=83), mqtt(mqttControlPacketType=2,mqttRemainingLength=2)
```

**Figura 7.24.** Mensajes mostrados en la terminal cuando se recibe un CONNECT y se responde con un CONNACK.

En la Figura 7.25, que se corresponde con una captura de Wireshark en la interfaz “s2-eth3” (suscriptor), se puede apreciar cómo se obtiene una respuesta por parte del controlador correcta y según lo esperado para el mensaje CONNECT.

```
No.   Time          Source          Destination     Protocol  Length Info
   16 6.102931075   192.168.1.102   192.168.1.100   MQTT          95 Connect Command
   17 6.116107852   192.168.1.100   192.168.1.102   MQTT          70 Connect Ack
   19 6.116348257   192.168.1.102   192.168.1.100   MQTT          78 Subscribe Request (id=1) [topic]
   20 6.137551226   192.168.1.100   192.168.1.102   MQTT          71 Subscribe Ack (id=1)

▸ Frame 17: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
▸ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▸ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
▸ Transmission Control Protocol, Src Port: 1883, Dst Port: 52944, Seq: 1, Ack: 30, Len: 4
▾ MQ Telemetry Transport Protocol, Connect Ack
  ▸ Header Flags: 0x20, Message Type: Connect Ack
    Msg Len: 2
  ▸ Acknowledge Flags: 0x00
    Return Code: Connection Accepted (0)

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00   ·······" 3DUf··E·
0010  00 38 00 00 00 00 ff 06  37 a5 c0 a8 01 64 c0 a8   ·8······ 7····d··
0020  01 66 07 5b ce d0 aa f0  3e 39 11 ef 96 7c 80 18   ·f·[···· >9···|··
0030  00 53 31 d8 00 00 01 01  08 0a cd 66 ce ed cd 66   ·S1····· ···f···f
0040  ce ec 20 02 00 00                                  ·· ···
```

**Figura 7.25.** Mensaje CONNACK creado por el controlador y capturado con Wireshark.

### SUBACK

Como se está tratando con la respuesta SUBACK el tipo de mensaje recibido debe ser SUBSCRIBE, cuyo código coincide con 8.

De nuevo, se pasa a preparar un paquete y añadirle las cabeceras Ethernet, IP y TCP de forma similar a lo ya explicado justo en el apartado anterior.

Lo siguiente es comprobar si el tópico al que se suscribe el cliente ya existe en el diccionario «topicsSubscribers» para añadir el nuevo suscriptor o, en caso de que no exista, crear una nueva entrada para dicho tópico y añadir su primer suscriptor. También se hace el proceso similar con el diccionario «subscribersTopics». Una vez el host está suscrito se incluye la información requerida de los campos TCP ([TCP seq, TCP ack, ts_val]) al suscriptor en el diccionario «subscribersTCPInfo».

Se notifica de que se ha recibido un mensaje SUBSCRIBE y se muestra el estado actual de los diccionarios «topicsSubscribers», «subscribersTopics» y «subscribersTCPInfo» a través de la terminal.

Por último, se añade la cabecera MQTT al paquete creado con anterioridad y se muestra por la terminal de que se ha enviado el mensaje SUBACK. En la cabecera se añade el tipo de paquete MQTT (tipo 9 para este caso), la longitud restante (3 para este caso), la ID del mensaje y la QoS.

```
##### 2021/08/15 19:12:57.544935 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv4(csum=23711,d
st='192.168.1.100',flags=2,header_length=5,identification=23038,offset=0,option=None,proto=6,src='192.168.1.102',tos=0,total_length=64,ttl=64,
version=4), tcp(ack=2243039336,bits=24,csum=42045,dst_port=1883,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(k
ind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412240035,ts_val=2412240076)],seq=567489919,src_port=46668,urgent=0,window_size=
83), mqtt(mqttControlPacketType=8,mqttRemainingLength=10), b'\x00\x01\x00\x05topic\x00']
##### 2021/08/15 19:12:57.549499 > Received TCP packet (seq=567489919, ack=2243039336), payload type: <class 'ryu.lib.packet.mqtt.mqtt'>, MQTT
 packet: mqtt(mqttControlPacketType=8,mqttRemainingLength=10), MQTT packet type: 8 (SUBSCRIBE), MQTT packet content (12 bytes): b'\x82\n\x00\x
01\x00\x05topic\x00'
######### TCP INIT
### 2021/08/15 19:12:57.551664 > MQTT SUBSCRIBE (grantedQoS=0, messageID=1, topic=b'topic') received from 192.168.1.102
######### topicsSubscribers:  {b'topic': [['192.168.1.102-46668', 0]]}
######### subscribersTopics:  {'192.168.1.102-46668': [b'topic']}
######### subscribersTCPInfo: {'192.168.1.102-46668': [2243039336, 567489931, 2412240076]}
######### MQTT INIT (mqttControlPacketType=9, mqttRemainingLength=3, returnCode=None, messageID=1, qos=None, topic=None, message=None) #######
##
### 2021/08/15 19:12:57.551664 > MQTT SUBACK (grantedQoS=0, messageID=1) sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={'192.168.1.102-46668': [2243039336, 567489931, 2412240076]}
######### subscribersTCPInfo updated for 192.168.1.102-46668 (TCP payload size=2): [2243039338, 567489931, 2412240076]
######### MQTT SERIALIZE #########
######### MQTT SERIALIZE HEADER (mqttByte1=144, mqttRLByte1=3) #########
######### MQTT SERIALIZE SUBACK (mqttControlPacketType=9, messageID=1, grantedQoS=0) #########
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14244,dst='192.168.1.102',flags=0,header_length
=5,identification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=57,ttl=255,version=4), tcp(ack=567489931,bits=24,csum=
56819,dst_port=46668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8
,length=10,ts_ecr=2412240076,ts_val=2412240077)],seq=2243039336,src_port=1883,urgent=0,window_size=83), mqtt(mqttControlPacketType=9,mqttRemai
ningLength=3)
```

**Figura 7.26.** Mensajes mostrados en la terminal cuando se recibe un SUBSCRIBE y se responde con un SUBACK.

En la Figura 7.27, que se corresponde con una captura de Wireshark en la interfaz “s2-eth3” (suscriptor), se puede apreciar cómo se obtiene una respuesta por parte del controlador correcta y según lo esperado para el mensaje SUBSCRIBE.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 16 | 6.102931075 | 192.168.1.102 | 192.168.1.100 | MQTT | 95 | Connect Command |
| 17 | 6.116107852 | 192.168.1.100 | 192.168.1.102 | MQTT | 70 | Connect Ack |
| 19 | 6.116348257 | 192.168.1.102 | 192.168.1.100 | MQTT | 78 | Subscribe Request (id=1) [topic] |
| 20 | 6.137551226 | 192.168.1.100 | 192.168.1.102 | MQTT | 71 | Subscribe Ack (id=1) |

```
▶ Frame 20: 71 bytes on wire (568 bits), 71 bytes captured (568 bits) on interface 0
▶ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▶ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
▶ Transmission Control Protocol, Src Port: 1883, Dst Port: 52944, Seq: 5, Ack: 42, Len: 5
▼ MQ Telemetry Transport Protocol, Subscribe Ack
  ▶ Header Flags: 0x90, Message Type: Subscribe Ack
    Msg Len: 3
    Message Identifier: 1
    Granted QoS: At most once delivery (Fire and Forget) (0)

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00   ·······" 3DUf··E·
0010  00 39 00 00 00 00 ff 06  37 a4 c0 a8 01 64 c0 a8   ·9······ 7····d··
0020  01 66 07 5b ce d0 aa f0  3e 3d 11 ef 96 88 80 18   ·f·[···· >=······
0030  00 53 c1 a8 00 00 01 01  08 0a cd 66 ce fb cd 66   ·S······ ···f···f
0040  ce fa 90 03 00 01 00                               ·······
```

**Figura 7.27.** Mensaje SUBACK creado por el controlador y capturado con Wireshark.

**PINGRESP**

Como se está tratando con la respuesta PINGRESP el tipo de mensaje recibido debe ser PINGREQ, cuyo código coincide con 12.

De nuevo, se pasa a preparar un paquete y añadirle las cabeceras Ethernet, IP y TCP de forma similar a lo ya explicado justo en el apartado anterior.

Lo siguiente es notificar por la terminal de que se ha recibido un mensaje PINGREQ y que se está enviando un mensaje TCP ACK. Se crea el paquete que va a ser enviado con la función *«generate_tcp_ack»*, se notifica de que ha sido enviado y se envía con la función *«_send_packet»*.

Tras ello se añade la cabecera MQTT al paquete creado con anterioridad, el cual ya tenía las cabeceras Ethernet, IP y TCP, y se notifica por terminal de que se ha enviado el mensaje PINGRESP. En la cabecera se añade el tipo de paquete MQTT (tipo 13 para este caso) y la longitud restante (0 para este caso).

```
##### 2021/08/15 19:13:57.551662 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv4(csum=23718,dst='192.
168.1.100',flags=2,header_length=5,identification=23041,offset=0,option=None,proto=6,src='192.168.1.102',tos=0,total_length=54,ttl=64,version=4), tcp(
ack=2243039359,bits=24,csum=49984,dst_port=1883,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOpti
onTimestamps(kind=8,length=10,ts_ecr=2412240118,ts_val=2412300101)],seq=567489931,src_port=46668,urgent=0,window_size=83), mqtt(mqttControlPacketType=
12,mqttRemainingLength=0)]
######### subscribersTCPInfo updated for 192.168.1.102-46668: [2243039359, 567489933, 2412300101]
##### 2021/08/15 19:13:57.553167 > Received TCP packet (seq=567489931, ack=2243039359), payload type: <class 'ryu.lib.packet.mqtt.mqtt'>, MQTT packet:
 mqtt(mqttControlPacketType=12,mqttRemainingLength=0), MQTT packet type: 12 (PINGREQ), MQTT packet content (2 bytes): b'\xc0\x00'
######### TCP INIT
### 2021/08/15 19:13:57.553557 > MQTT PING REQUEST received from 192.168.1.102, sending TCP ACK...
######### TCP INIT
### 2021/08/15 19:13:57.553557 > TCP ACK sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={'192.168.1.102-46668': [2243039359, 567489933, 2412300101]}
######### subscribersTCPInfo updated for 192.168.1.102-46668 (TCP payload size=0): [2243039359, 567489933, 2412300101]
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14249,dst='192.168.1.102',flags=0,header_length=5,ident
ification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=52,ttl=255,version=4), tcp(ack=567489933,bits=24,csum=39152,dst_port=4
6668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=241230
0101,ts_val=2412300102)],seq=2243039359,src_port=1883,urgent=0,window_size=83)
######### MQTT INIT (mqttControlPacketType=13, mqttRemainingLength=0, returnCode=None, messageID=None, qos=None, topic=None, message=None) #########
### 2021/08/15 19:13:57.553557 > MQTT PING RESPONSE sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={'192.168.1.102-46668': [2243039359, 567489933, 2412300101]}
######### subscribersTCPInfo updated for 192.168.1.102-46668 (TCP payload size=2): [2243039361, 567489933, 2412300101]
######### MQTT SERIALIZE #########
######### MQTT SERIALIZE HEADER (mqttByte1=208, mqttRLByte1=0) #########
######### MQTT SERIALIZE PING RESPONSE (mqttControlPacketType=13) #########
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14247,dst='192.168.1.102',flags=0,header_length=5,ident
ification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=54,ttl=255,version=4), tcp(ack=567489933,bits=24,csum=51437,dst_port=4
6668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=241230
0101,ts_val=2412300102)],seq=2243039359,src_port=1883,urgent=0,window_size=83), mqtt(mqttControlPacketType=13,mqttRemainingLength=0)
```

**Figura 7.28.** Mensajes mostrados en la terminal cuando se recibe un PINGREQ y se responde con un PINGRESP.

En la Figura 7.29, que se corresponde con una captura de Wireshark en la interfaz "s2-eth3" (suscriptor), se puede apreciar cómo se obtiene una respuesta por parte del controlador correcta y según lo esperado para el mensaje PINGREQ.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 13 | 6.221960369 | 192.168.1.100 | 192.168.1.102 | MQTT | 71 | Subscribe Ack (id=1) |
| 14 | 6.221970568 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 46056 → 1883 [ACK] Seq=42 Ack=10 Win=42496 Len=0 TSval=2388007203 TSecr=2388007195 |
| 23 | 66.310640033 | 192.168.1.102 | 192.168.1.100 | MQTT | 68 | Ping Request |
| 24 | 66.319675375 | 192.168.1.100 | 192.168.1.102 | TCP | 66 | 1883 → 46056 [PSH, ACK] Seq=10 Ack=44 Win=42496 Len=0 TSval=2388067293 TSecr=2388067292 |
| 25 | 66.319830643 | 192.168.1.100 | 192.168.1.102 | MQTT | 68 | Ping Response |
| 26 | 66.319836116 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 46056 → 1883 [ACK] Seq=44 Ack=12 Win=42496 Len=0 TSval=2388067301 TSecr=2388067293 |

```
▸ Frame 25: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 0
▸ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▸ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
▸ Transmission Control Protocol, Src Port: 1883, Dst Port: 46056, Seq: 10, Ack: 44, Len: 2
▾ MQ Telemetry Transport Protocol, Ping Response
  ▸ Header Flags: 0xd0, Message Type: Ping Response
    Msg Len: 0

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00   ·······" 3DUf··E·
0010  00 36 00 00 00 00 ff 06  37 a7 c0 a8 01 64 c0 a8   ·6······ 7····d··
0020  01 66 07 5b b3 e8 43 03  aa a7 7f 6e 47 50 80 18   ·f·[··C· ···nGP··
0030  00 53 8e 2f 00 00 01 01  08 0a 8e 57 03 dd 8e 57   ·S·/···· ···W···W
0040  03 dc d0 00                                         ····
```

**Figura 7.29.** Mensaje PINGRESP creado por el controlador y capturado con Wireshark.

### PUBLISH

Como se está tratando con la respuesta PUBLISH el tipo de mensaje recibido debe ser PUBLISH, cuyo código coincide con 3.

Lo primero es notificar por la terminal de que se ha recibido un mensaje PUBLISH y se está enviando un mensaje TCP ACK. Se crea el paquete con la función «*generate_tcp_ack*», se notifica de que se ha enviado el paquete y se envía con la función «*_send_packet*».

Lo siguiente es comprobar que suscriptores están suscritos al tópico que va incluido en el mensaje PUBLSIH consultándolo en el diccionario «topicsSubscribers» y se obtienen la

dirección IP, puerto TCP, dirección MAC de destino, número de secuencia TCP, número de confirmación TCP y la marca de tiempo TCP (ts_val).

Seguido se comprueba que la dirección MAC del *switch* esté incluida en la tabla MAC o si la dirección MAC de destino está incluida en la tabla MAC del *switch*. Si se da el caso de que no se cumpla alguna de las dos, se recomienda a través de la terminal que se reinicie Mininet y se haga un *pingall*.

Tras comprobar las tablas MAC con éxito se notifica a través de la terminal del reenvío del mensaje PUBLISH recibido. Se crea un nuevo paquete y se le añaden las cabeceras Ethernet, IP y TCP igual que se ha ido haciendo en otras respuestas creadas anteriormente. Ha este paquete se le añade la cabecera MQTT en la cual se indican el tipo de paquete MQTT (tipo 3 para este caso), la longitud restante, el tópico y el mensaje.

Por último, se muestra en la terminal la tabla MAC completa, el *datapath.id*, la dirección MAC y la tabla MAC para ese *datapath.id* en específico. Además, se notifica de que se ha reenviado el PUBLISH y se envía con la función *«_send_packet»*.

```
##### 2021/08/15 19:13:11.401310 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:03'), ipv4(csum=22994,dst='192.168.1.100',flags=2,header_length=5,identifi
cation=23748,offset=0,option=None,proto=6,src='192.168.1.103',tos=0,total_length=70,ttl=64,version=4), tcp(ack=4264153383,bits=24,csum=46294,dst_port=1883,offset=8,option=[TCPOptionNoOperation(
kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=1895634150,ts_val=1895634172)],seq=4108383773,src_port=36930,urgent=0,window_size=83), mqtt(
mqttControlPacketType=3,mqttRemainingLength=16), b'\x00\x05topicprueba...']
##### 2021/08/15 19:13:11.404278 > Received TCP packet (seq=4108383773, ack=4264153383), payload type: <class 'ryu.lib.packet.mqtt.mqtt'>, MQTT packet: mqtt(mqttControlPacketType=3,mqttRemainin
gLength=16), MQTT packet type: 3 (PUBLISH), MQTT packet content (18 bytes): b'0\x10\x00\x05topicprueba...'
### 2021/08/15 19:13:11.404278 > MQTT PUBLISH (topic=b'topic', message=b'prueba...') received from 192.168.1.103, sending TCP ACK...
######### TCP INIT
### 2021/08/15 19:13:11.404278 > TCP ACK sent to 192.168.1.103
######### subscriber: 192.168.1.103-36930, subscribersTCPInfo={'192.168.1.102-46668': [2243039341, 567489931, 2412240117]}
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:03',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14248,dst='192.168.1.103',flags=0,header_length=5,identification=0,offset=0,option=None,proto=6,sr
c='192.168.1.100',tos=0,total_length=52,ttl=255,version=4), tcp(ack=4108383791,bits=24,csum=37970,dst_port=36930,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kin
d=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=1895634172,ts_val=1895634173)],seq=4264153383,src_port=1883,urgent=0,window_size=83)
### 2021/08/15 19:13:11.404278 > forward MQTT PUBLISH (topic=b'topic', message=b'prueba...') to 192.168.1.102 (MAC=00:00:00:00:00:02, TCP port=46668, seq=2243039341, ack=567489931, ts_val=24122
40117)
######### TCP INIT
######### MQTT INIT (mqttControlPacketType=3, mqttRemainingLength=16, returnCode=None, messageID=None, qos=None, topic=topic, message=prueba...) #########
MAC table={1: {'52:11:8c:4b:fa:d9': 1, '00:00:00:00:00:02': 1, '00:00:00:00:00:04': 2, '00:00:00:00:00:01': 1, '72:56:69:d2:2a:f9': 2, '00:00:00:00:00:03': 2}, 3: {'02:83:35:54:fb:58': 1, '00:0
0:00:00:00:04': 3, '00:00:00:00:00:02': 1, '00:00:00:00:00:01': 1, '00:00:00:00:00:03': 2, '52:11:8c:4b:fa:d9': 1}, 2: {'00:00:00:00:00:02': 3, '00:00:00:00:00:04': 1, '00:00:00:00:00:01': 2, '
72:56:69:d2:2a:f9': 1, '00:00:00:00:00:03': 1, 'ae:aa:4a:37:fb:06': 1}}
datapath.id=3
MAC addr=00:00:00:00:00:02
MAC table={'02:83:35:54:fb:58': 1, '00:00:00:00:00:04': 3, '00:00:00:00:00:02': 1, '00:00:00:00:00:01': 1, '00:00:00:00:00:03': 2, '52:11:8c:4b:fa:d9': 1}, MAC addr=00:00:00:00:00:02
### 2021/08/15 19:13:11.404278 >>>>> MQTT PUBLISH message: ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=0,dst='192.168.1.102',flags=0,header_length=5,iden
tification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=0,ttl=255,version=4), tcp(ack=567489931,bits=24,csum=0,dst_port=46668,offset=8,option=[TCPOptionNoOperation(kind
=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412240117,ts_val=2412240118)],seq=2243039341,src_port=1883,urgent=0,window_size=83), mqtt(mqttC
ontrolPacketType=3,mqttRemainingLength=16) (datapath=3, port=1)
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={'192.168.1.102-46668': [2243039341, 567489931, 2412240117]}
######### subscribersTCPInfo updated for 192.168.1.102-46668 (TCP payload size=2): [2243039343, 567489931, 2412240117]
######### MQTT SERIALIZE #########
######### MQTT SERIALIZE HEADER (mqttByte1=48, mqttRLByte1=16) #########
######### MQTT SERIALIZE PUBLISH (mqttControlPacketType=3, topic=topic, message=prueba...) #########
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14231,dst='192.168.1.102',flags=0,header_length=5,identification=0,offset=0,option=None,proto=6,sr
c='192.168.1.100',tos=0,total_length=70,ttl=255,version=4), tcp(ack=567489931,bits=24,csum=36353,dst_port=46668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind
=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=2412240117,ts_val=2412240118)],seq=2243039341,src_port=1883,urgent=0,window_size=83), mqtt(mqttControlPacketType=3,mqttRemainingLength=
16)
```

**Figura 7.30.** Mensajes mostrados en la terminal cuando se recibe un PUBLISH por parte del publicador y es reenviado al suscriptor.

En la Figura 7.31 y Figura 7.32, que se corresponden con capturas de Wireshark en las interfaces “s3-eth2” (publicador) y “s2-eth3” (suscriptor) respectivamente, se aprecia como se recibe el mensaje PUBLISH por parte del publicador y se reenvía al suscriptor de manera correcta.

```
No.   Time          Source          Destination     Protocol  Length Info
   16 11.700436147  192.168.1.103   192.168.1.100   MQTT          95 Connect Command
   17 11.713837524  192.168.1.100   192.168.1.103   MQTT          70 Connect Ack
   19 11.714073353  192.168.1.103   192.168.1.100   MQTT          84 Publish Message [topic]
   20 11.714094681  192.168.1.103   192.168.1.100   MQTT          68 Disconnect Req

▶ Frame 19: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface 0
▶ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 11:22:33:44:55:66 (11:22:33:44:55:66)
▶ Internet Protocol Version 4, Src: 192.168.1.103, Dst: 192.168.1.100
▶ Transmission Control Protocol, Src Port: 54758, Dst Port: 1883, Seq: 30, Ack: 5, Len: 18
▼ MQ Telemetry Transport Protocol, Publish Message
  ▶ Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
    Msg Len: 16
    Topic Length: 5
    Topic: topic
    Message: prueba...

0000  11 22 33 44 55 66 00 00  00 00 00 03 08 00 45 00   ·"3DUf·· ······E·
0010  00 46 57 16 40 00 40 06  5f 80 c0 a8 01 67 c0 a8   ·FW·@·@· _····g··
0020  01 64 d5 e6 07 5b c1 e9  fd 23 5d e8 ed ae 80 18   ·d···[·· ·#]·····
0030  00 53 84 54 00 00 01 01  08 0a 6d ee ca 58 6d ee   ·S·T···· ··m··Xm·
0040  ca 4b 30 10 00 05 74 6f  70 69 63 70 72 75 65 62   ·K0···to picprueb
0050  61 2e 2e 2e                                        a...
```

**Figura 7.31.** Mensaje PUBLISH recibido por el controlador de parte del publicador y capturado con Wireshark.

```
No.   Time          Source          Destination     Protocol  Length Info
   16 6.102931075   192.168.1.102   192.168.1.100   MQTT          95 Connect Command
   17 6.116107852   192.168.1.100   192.168.1.102   MQTT          70 Connect Ack
   19 6.116348257   192.168.1.102   192.168.1.100   MQTT          78 Subscribe Request (id=1) [topic]
   20 6.137551226   192.168.1.100   192.168.1.102   MQTT          71 Subscribe Ack (id=1)
   22 11.768188333  192.168.1.100   192.168.1.102   MQTT          84 Publish Message [topic]
   24 20.271242715  192.168.1.102   192.168.1.100   MQTT          68 Disconnect Req

▶ Frame 22: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface 0
▶ Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▶ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
▶ Transmission Control Protocol, Src Port: 1883, Dst Port: 52944, Seq: 10, Ack: 42, Len: 18
▼ MQ Telemetry Transport Protocol, Publish Message
  ▶ Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
    Msg Len: 16
    Topic Length: 5
    Topic: topic
    Message: prueba...

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00   ·······" 3DUf··E·
0010  00 46 00 00 00 00 ff 06  37 97 c0 a8 01 64 c0 a8   ·F······ 7····d··
0020  01 66 07 5b ce d0 aa f0  3e 42 11 ef 96 88 80 18   ·f·[···· >B······
0030  00 53 71 de 00 00 01 01  08 0a cd 66 cf 10 cd 66   ·Sq····· ···f···f
0040  cf 0f 30 10 00 05 74 6f  70 69 63 70 72 75 65 62   ··0···to picprueb
0050  61 2e 2e 2e                                        a...
```

**Figura 7.32.** Mensaje PUBLISH reenviado por el controlador al suscriptor y capturado con Wireshark.

**DISCONNECT**

En este caso, en realidad no se hace nada, únicamente se notifica que se ha recibido un DISCONNECT por la terminal. Si se recibe un mensaje DISCONNECT, con código 14 y que incluye TCP FIN, se maneja con la función *«_handle_tcp»*. En la pág.106 se explica este proceso ya que coincide con la respuesta TCP FIN ACK.

```
##### 2021/08/15 19:14:39.046639 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv4(csum=23716,dst='192.
168.1.100',flags=2,header_length=5,identification=23043,offset=0,option=None,proto=6,src='192.168.1.102',tos=0,total_length=54,ttl=64,version=4), tcp(
ack=2243039361,bits=24,csum=5844,dst_port=1883,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptio
nTimestamps(kind=8,length=10,ts_ecr=2412300102,ts_val=2412341596)],seq=567489933,src_port=46668,urgent=0,window_size=83), mqtt(mqttControlPacketType=1
4,mqttRemainingLength=0)]
######### subscribersTCPInfo updated for 192.168.1.102-46668: [2243039361, 567489935, 2412341596]
##### 2021/08/15 19:14:39.050288 > Received TCP packet (seq=567489933, ack=2243039361), payload type: <class 'ryu.lib.packet.mqtt.mqtt'>, MQTT packet:
 mqtt(mqttControlPacketType=14,mqttRemainingLength=0), MQTT packet type: 14 (DISCONNECT), MQTT packet content (2 bytes): b'\xe0\x00'
### 2021/08/15 19:14:39.050288 > MQTT DISCONNECT received from 192.168.1.102
######### TCP PARSER
######### TCP INIT
##### 2021/08/15 19:14:39.052486 > pkt.protocols: [ethernet(dst='11:22:33:44:55:66',ethertype=2048,src='00:00:00:00:00:02'), ipv4(csum=23717,dst='192.
168.1.100',flags=2,header_length=5,identification=23044,offset=0,option=None,proto=6,src='192.168.1.102',tos=0,total_length=52,ttl=64,version=4), tcp(
ack=2243039361,bits=17,csum=63195,dst_port=1883,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOpti
onTimestamps(kind=8,length=10,ts_ecr=2412300102,ts_val=2412341596)],seq=567489935,src_port=46668,urgent=0,window_size=83)]
### 2021/08/15 19:14:39.054410 > TCP FIN received by the MQTT broker APP from 192.168.1.102
######### topicsSubscribers:  {}
######### subscribersTopics:  {}
######### subscribersTCPInfo: {}
######### TCP INIT
### 2021/08/15 19:14:39.055133 > TCP FIN+ACK sent to 192.168.1.102
######### subscriber: 192.168.1.102-46668, subscribersTCPInfo={}
######### TCP SERIALIZE
packet-out ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='11:22:33:44:55:66'), ipv4(csum=14249,dst='192.168.1.102',flags=0,header_length=5,ident
ification=0,offset=0,option=None,proto=6,src='192.168.1.100',tos=0,total_length=52,ttl=255,version=4), tcp(ack=567489936,bits=17,csum=21699,dst_port=4
6668,offset=8,option=[TCPOptionNoOperation(kind=1,length=1), TCPOptionNoOperation(kind=1,length=1), TCPOptionTimestamps(kind=8,length=10,ts_ecr=241234
1596,ts_val=2412341597)],seq=2243039361,src_port=1883,urgent=0,window_size=83)
```

**Figura 7.33**. Mensajes mostrados en la terminal cuando se recibe un DISCONNECT por parte del suscriptor y se responde con FIN+ACK.

En la Figura 7.34, que se corresponde con una captura de Wireshark en la interfaz “s2-eth3” (suscriptor), se puede apreciar cómo se obtiene una respuesta por parte del controlador correcta y según lo esperado para el mensaje DISCONNECT.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 22 | 11.768188333 | 192.168.1.100 | 192.168.1.102 | MQTT | 84 | Publish Message [topic] |
| 23 | 11.768201824 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 52944 → 1883 [ACK] Seq=42 Ack=28 Win=42496 Len=0 TSval=3446072589 TSecr=3446066960 |
| 24 | 20.271242715 | 192.168.1.102 | 192.168.1.100 | MQTT | 68 | Disconnect Req |
| 25 | 20.271270287 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 52944 → 1883 [FIN, ACK] Seq=44 Ack=28 Win=42496 Len=0 TSval=3446081093 TSecr=3446066960 |
| 26 | 20.283389087 | 192.168.1.100 | 192.168.1.102 | TCP | 66 | 1883 → 52944 [FIN, ACK] Seq=28 Ack=45 Win=42496 Len=0 TSval=3446081094 TSecr=3446081093 |
| 27 | 20.283407555 | 192.168.1.102 | 192.168.1.100 | TCP | 66 | 52944 → 1883 [ACK] Seq=45 Ack=29 Win=42496 Len=0 TSval=3446081105 TSecr=3446081094 |

```
Frame 26: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.102
Transmission Control Protocol, Src Port: 1883, Dst Port: 52944, Seq: 28, Ack: 45, Len: 0
    Source Port: 1883
    Destination Port: 52944
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 28    (relative sequence number)
    [Next sequence number: 28    (relative sequence number)]
    Acknowledgment number: 45    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
  Flags: 0x011 (FIN, ACK)
    Window size value: 83
    [Calculated window size: 42496]
    [Window size scaling factor: 512]
    Checksum: 0xe308 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  [SEQ/ACK analysis]
  [Timestamps]

0000  00 00 00 00 00 02 11 22  33 44 55 66 08 00 45 00   ·······" 3DUf··E·
0010  00 34 00 00 00 00 ff 06  37 a9 c0 a8 01 64 c0 a8   ·4······ 7····d··
0020  01 66 07 5b ce d0 aa f0  3e 54 11 ef 96 8b 80 11   ·f·[···· >T······
0030  00 53 e3 08 00 00 01 01  08 0a cd 67 06 46 cd 67   ·S······ ···g·F·g
0040  06 45                                              ·E
```

**Figura 7.34.** Mensaje FIN+ACK enviado por el controlador al suscriptor en respuesta a un DISCONNECT y capturado con Wireshark.

# 8. Resultados

En este apartado se hace un estudio de los resultados poniendo a prueba la red creada con la sustitución del *broker* y comparándola en distintas situaciones con la red inicial con *broker*. Se va a utilizar la topología de la Figura 7.1 vista en la subsección 7.1.1 Diseño de la red En ambos casos, con *broker* y sin él, se utiliza un nivel de QoS 0.

## 8.1. Funcionamiento con varios publicadores

El primer escenario que se va a contemplar es teniendo en cuenta más de un publicador para un mismo *topic*.

Para el caso con *broker*: el *host* h1 hace de éste, el *host* h2 de suscriptor, los *hosts* h3 y h4 de publicadores y el *topic* es "tfg". La topología y los roles de cada *host* se muestran en la Figura 8.1.

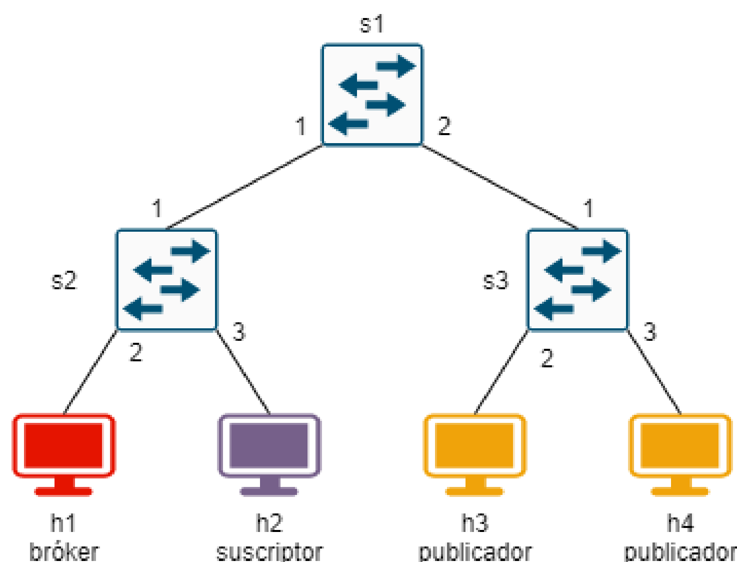


**Figura 8.1.** Topología con varios publicadores.

Se sigue un proceso muy similar al visto en la subsección 7.1.2 del apartado de Desarrollo práctico. La única diferencia es que se añade un publicador más, lo cual se hace de manera análoga a como se ha hecho para el *host* h3. Este escenario con *broker* se muestra en la Figura 8.2.

Se hace un envío total de cuatro mensajes por parte de los publicadores, dos desde el *host* h3 ("prueba…1" y "prueba…3") y dos desde el *host* h4 ("prueba…2" y "prueba…4"). Todos llegan de manera satisfactoria y ordenada al suscriptor.

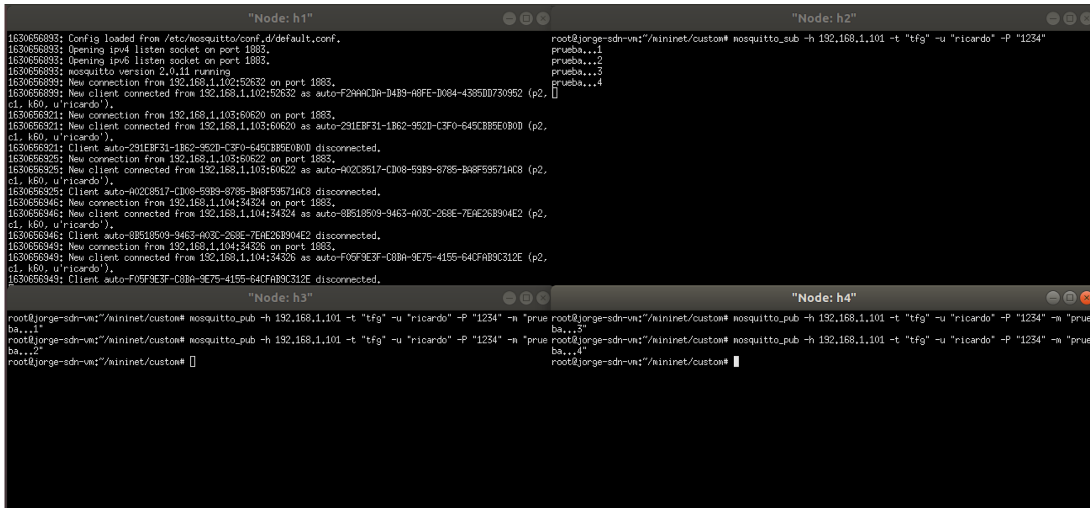


**Figura 8.2.** Escenario con varios publicadores y *broker*.

Para el caso sin *broker* ya no se le da ese rol a ningún *host*. Ahora, se utiliza como referencia la IP ficticia 192.168.1.100, especificada en la aplicación creada para el controlador Ryu tal y como se ve en la sección 7.2. Este nuevo escenario, sin la figura del *broker*, se muestra en la Figura 8.3. Los mensajes publicados son exactamente los mismos que para el escenario anterior.

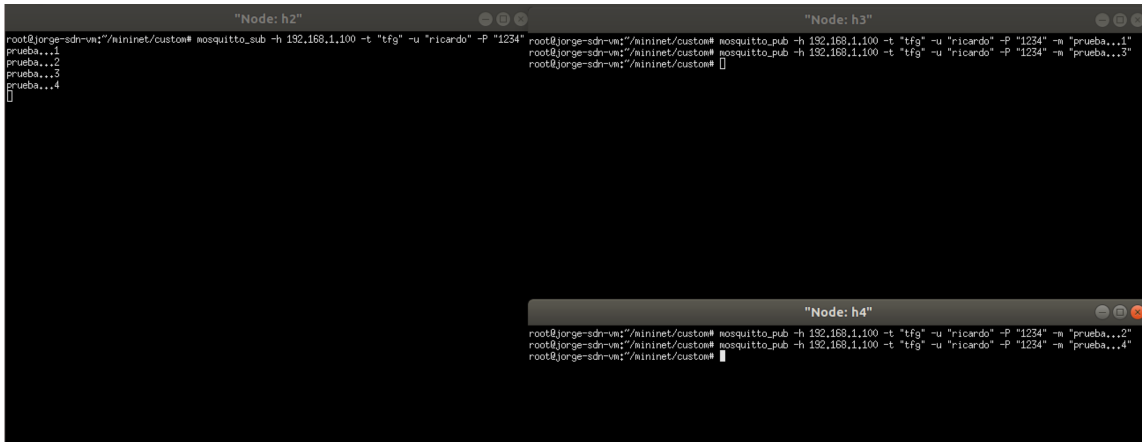


**Figura 8.3.** Escenario con varios publicadores y sin *broker*.

## 8.2. Funcionamiento con varios suscriptores

El segundo escenario que se va a contemplar es teniendo en cuenta más de un suscriptor para un mismo *topic*.

Para el caso con *broker*: el *host* h1 hace de *broker*, los *hosts* h2 y h4 de suscriptores, el *host* h3 de publicador y el *topic* es “tfg”. La topología y los roles de cada *host* se muestran en la Figura 8.4.

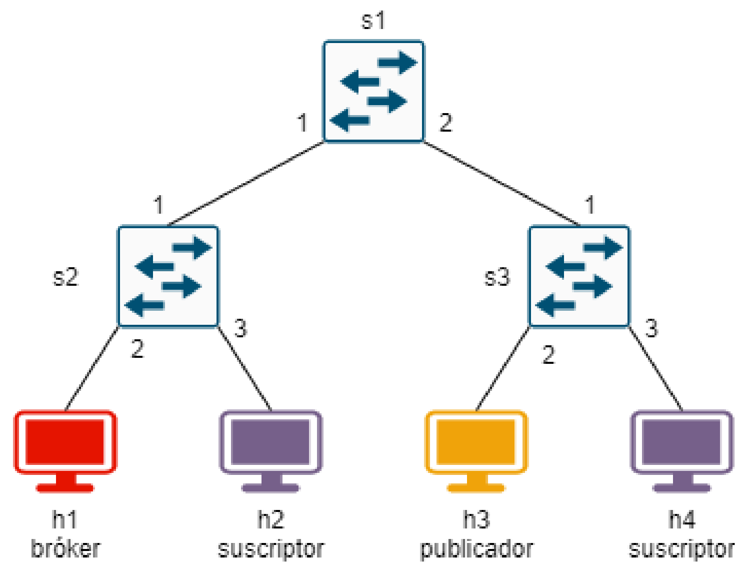


**Figura 8.4.** Topología con varios suscriptores.

Se vuelve a seguir un proceso muy similar al visto en la subsección 7.1.2 del apartado de Desarrollo práctico, pero en este caso se añade un suscriptor más en vez de un publicador. Ambos suscriptores reciben los mismos mensajes ya que están suscritos al mismo *topic*. Este escenario con *broker* se muestra en la Figura 8.5.

**Figura 8.5.** Escenario con varios suscriptores y *broker*.

Lo mismo sucede para el caso sin *broker*. Este nuevo escenario, sin la figura del *broker*, se muestra en la Figura 8.6. Los mensajes publicados son exactamente los mismos que para el escenario anterior.

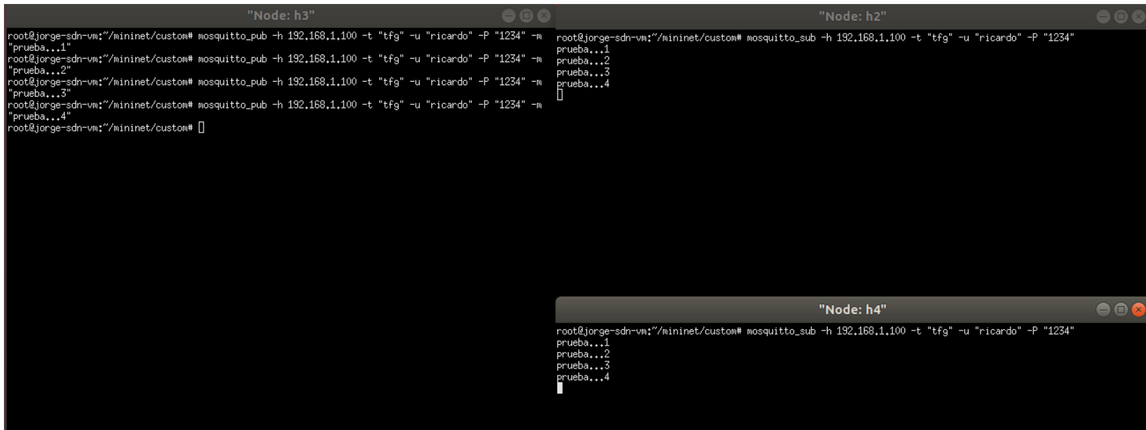


**Figura 8.6.** Escenario con varios suscriptores y sin *broker*.

## 8.3. Funcionamiento con varios *topics*

El tercer escenario que se va a contemplar es teniendo en cuenta más de un suscriptor para distintos *topics*.

Para el caso con *broker*: el *host* h1 hace de *broker*, los *hosts* h2 y h4 de suscriptores, el *host* h3 de publicador y los *topics* son "tfg" y "tfg2", respectivamente para cada suscriptor. La topología y los roles de cada *host* se muestran en la Figura 8.7.

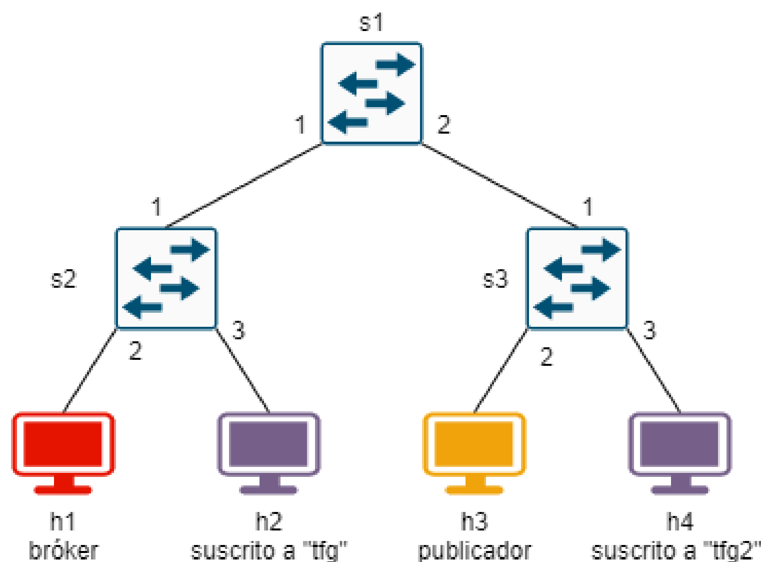


**Figura 8.7.** Topología con suscriptores a *topics* diferentes.

Se repite lo mismo que en la sección 8.2 pero esta vez cada suscriptor se suscribe a un *topic* distinto. El publicador se encarga de enviar mensajes para cada uno de los *topics* y se observa en la Figura 8.8 como estos llegan adecuadamente al suscriptor correcto.

Figura 8.8. Escenario con varios *topics* y *broker*.

Se repite lo mismo, pero sin *broker*, y se obtienen los mismos resultados tal y como se observa en la Figura 8.9.

Figura 8.9. Escenario con varios *topics* y sin *broker*.

## 8.4. Comparativa de tiempos de envío

Por último, se va a comparar el tiempo que tarda un mensaje PUBLISH en llegar al suscriptor correspondiente contando desde que sale del publicador. El *host* h2 hace de suscriptor y el *host* h3 de publicador en ambos casos, tanto con *broker* como sin él. Se utiliza la herramienta Wireshark para capturar en la interfaz "s2-eth3" y "s3-eth2", correspondientes al suscriptor y el publicador respectivamente.

Primero se muestran las capturas respectivas al caso con *broker* (ver Figura 8.10) y se calcula la diferencia de tiempo entre la salida del mensaje y su llegada (ver Tabla 8.1).

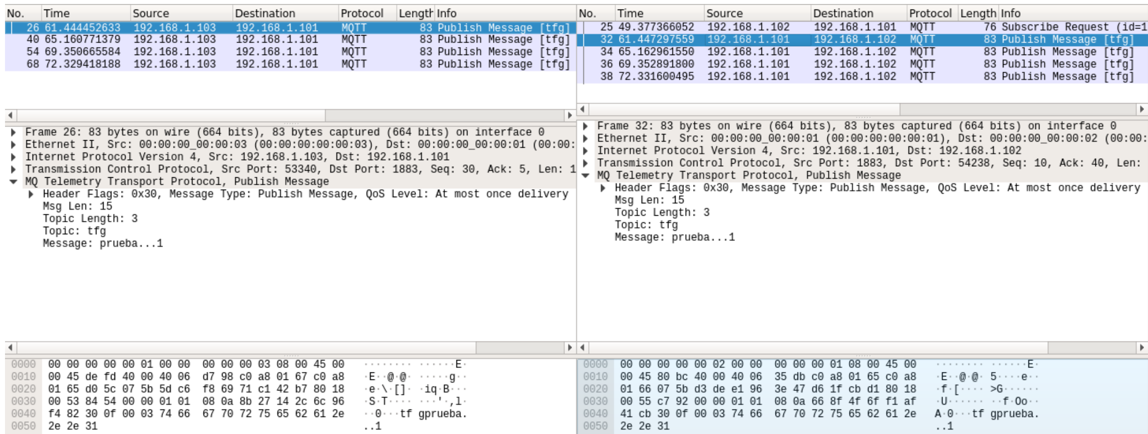

**Figura 8.10.** Publicador a la izquierda y suscriptor a la derecha con *broker*.

| Envío desde publicador (s) | Recepción en el suscriptor (s) | Diferencia (s) |
|---|---|---|
| 61,444 | 61,447 | 0,003 |
| 65,161 | 65,163 | 0,002 |
| 69,351 | 69,353 | 0,002 |
| 72,329 | 72,332 | 0,003 |
| | **Media** | **0,0025** |

**Tabla 8.1.** Tiempos de envío mensaje PUBLISH con *broker*.

En segundo lugar, se muestran las capturas respectivas al caso sin *broker* (ver Figura 8.11) y se calcula la diferencia de tiempo entre la salida del mensaje y su llegada (ver Tabla 8.2).

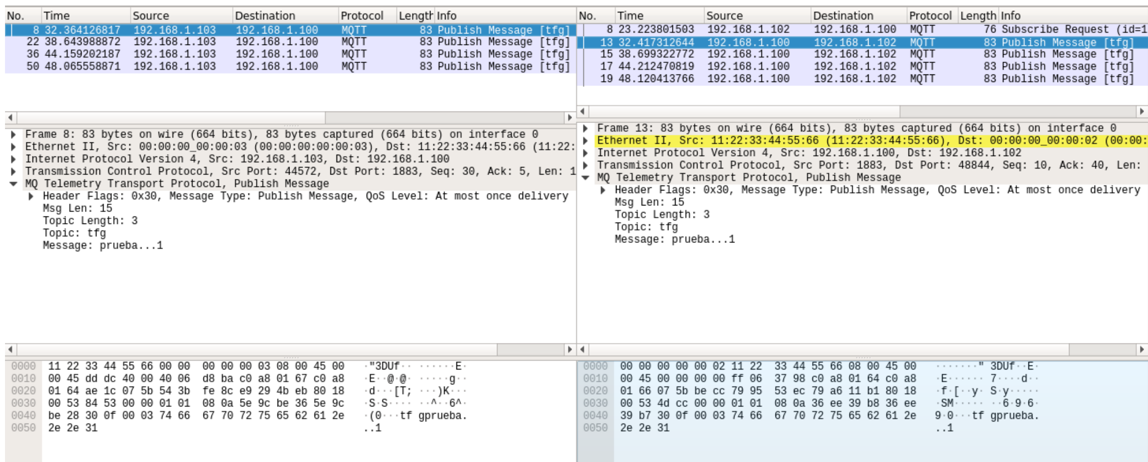

**Figura 8.11.** Publicador a la izquierda y suscriptor a la derecha sin *broker*.

| Envío desde publicador (s) | Recepción en el suscriptor (s) | Diferencia (s) |
|---|---|---|
| 32,364 | 32,417 | 0,053 |
| 38,644 | 38,699 | 0,055 |
| 44,159 | 44,212 | 0,053 |
| 48,066 | 48,120 | 0,054 |
| | **Media** | **0,0537** |

**Tabla 8.2.** Tiempos de envío mensaje PUBLISH sin *broker*.

Apreciando los resultados de la Tabla 8.1 y Tabla 8.2 se puede decir que los tiempos de envío son constantes en ambos casos y el tiempo de envío es mayor en el caso sin *broker*. Esto no es lo esperado y se debe entre otras cosas a que para que la comparación sea justa, el *broker* tendría que estar programado de la misma forma que el controlador SDN. Aunque lo que hay en el controlador SDN es más sencillo y deber ser más rápido, puede que mosquitto [13] *broker* esté hecho con otro lenguaje de programación más rápido, que esté más optimizado porque ha tenido muchas versiones y mucho trabajo de desarrollo... no es trivial que tenga que salir peor tiempo de envío.

# 9. Conclusiones

## 9.1. Logros conseguidos

Se han finalizado con éxito los objetivos vistos al comienzo del trabajo. Los más relevantes son:

- **Estudiar el protocolo MQTT.** Se ha estudiado el protocolo MQTT detenidamente, en especial de qué compone cada tipo de mensaje para posteriormente decodificarlos y codificarlos.
- **Utilizar Mininet para crear redes SDN.** Se han diseñado topologías básicas y algo más complejas utilizando tanto las que vienen por defecto en esta herramienta como personalizadas.
- **Utilizar controlador Ryu.** Se ha conseguido utilizar este controlador usando tanto aplicaciones por defecto que vienen en Ryu, como creando una nueva aplicación que se encarga de gestionar los mensajes MQTT que le van llegando.
- **Decodificar (*parse*) y codificar (*serialize*) los mensajes MQTT.** Se ha creado un nuevo decodificador/codificador específico para paquetes MQTT en la biblioteca de paquetes de Ryu
- **Hacer que el controlador emule al *broker*.** Se ha logrado sustituir la figura del bróker haciendo que el controlador pueda gestionar los mensajes MQTT directamente gracias a la nueva aplicación creada. Se asigna una IP ficticia para que los clientes sepan a quien dirigirse como “bróker”.
- **Crear un escenario MQTT sin la figura del bróker.** Se ha alcanzado lo que es el objetivo final del trabajo, crear un escenario MQTT sin *broker* para el nivel 0 de QoS que funcione con la misma normalidad que con *broker*. El controlador SDN, Ryu, ha sido el encargado de emular al *bróker* gracias a su programación según lo necesario.

## 9.2. Trabajos futuros

Existen posibles mejoras sobre el trabajo actual. A continuación, se mencionan algunas de estas:

- **Niveles 1 y 2 de QoS.** En este trabajo sólo se ha tenido en cuenta el nivel 0 de QoS que ofrece el protocolo MQTT. Por ello, si se busca implementar los niveles 1 y 2 de QoS que ofrece el protocolo MQTT, se debe trabajar en los nuevos mensajes que aparecen para estas calidades de servicio. Con el aumento del nivel de QoS se toleran menos fallos y la complejidad de implementación aumenta.
- **Reducir los tiempos de envío.** Tal y como se ha visto en 8.4 Comparativa de tiempos de envío, el tiempo en el caso sin *broker* ha aumentado. Por ello, se puede optimizar mejor el código del controlador para reducir el tiempo de envío al menos hasta igualar al caso con *broker*.
- **Enviar los datos entre *brokers* por UDP y usar *multicast*.** Esto ayudaría a reducir el tráfico y el retardo dentro de la red SDN ya que el protocolo UDP genera menos mensajes que el protocolo TCP y *multicast* permite enviar un único flujo de datos a múltiples destinos al mismo tiempo.

## 9.3. Valoración personal

Durante los últimos años del grado había oído hablar sobre el tema de las IoT causándome interés por conocer mejor este campo. La realización de este trabajo ha supuesto un mayor aumento de los conocimientos relacionados con el mundo de las IoT, un concepto que irá adoptando mayor importancia con el paso de los años hasta estar totalmente implantado en la sociedad.

El desarrollo del trabajo realizado ha supuesto un desafío puesto que ha cubierto mucho más tiempo del que esperaba, ya que pretendía presentarlo en el mes de julio y a causa de la cantidad de asignaturas en la que estaba matriculado ha tenido que ser retrasado hasta hoy. A pesar de esto, el continuar realizando el trabajo durante el verano me ha servido para poder centrarme en él en una mayor medida y nutrirme mucho más en el tema.

Como es normal han ido apareciendo algunas complicaciones durante el desarrollo, como por ejemplo fallos en la puesta en funcionamiento del *broker* o el programar en Python ya que era mi primer acercamiento a este lenguaje de programación. A causa de estancarme en los

fallos o problemas mencionados, entre otros, ha servido para incrementar el entendimiento del tema.

Muchas cosas de las aprendidas a lo largo del grado, en concreto de la especialidad de telemática, me han sido útiles a la hora de ejecutar el trabajo. Por ello, junto a los conocimientos que he adquirido en la elaboración de este, finalizo de forma satisfecha esta etapa.

# Bibliografía

[1] Anon, Welcome to RYU the network Operating system(nos)., (2014) *Welcome to RYU the Network Operating System(NOS) - Ryu 4.34 documentation*. Disponible en: https://ryu.readthedocs.io/en/latest/index.html [*última visita:* 22/08/2021]

[2] Baehost Blog., (2020) *Protección TCP con BAEHOST.* Baehost web hosting & cloud. Disponible en: https://blog.baehost.com/proteccion-tcp-en-baehost/ [última visita: 08/08/2021]

[3] Castillo, J.A., (2021) *Protocolo TCP/IP – Qué es y cómo funciona.* Profesional Review. Disponible en: https://www.profesionalreview.com/2020/03/21/protocolo-tcp-ip/#Protocolo_TCP [última visita: 08/08/2021]

[4] Clark, J., (2016) *What is the Internet of Things (IoT)?.* IBM. Disponible en: https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/ [última visita: 29/08/2021]

[5] ESPE., *RYU Controller.*, (2015) RYU. Disponible en: https://tinyurl.com/c4hztrrz [última visita: 22/08/2021]

[6] GitHub., *Introducción a Mininet.* GitHub. Disponible en: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet [última visita: 18/08/2021]

[7] H. Ning and Z. Wang, "Future Internet of Things Architecture :," IEEE Commun. Lett., vol. 15, no. 4, pp. 461–463, 2011.

[8] IBM., *CONNECT - Client requests a connection to a serve.* MQTT V3.1 Protocol Specification. Disponible en: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html#connect [última visita: 14/06/2021]

[9] IONOS., (2019) *¿Qué es el ARP (Address Resolution Protocol)?.* Digital Guide. Disponible en: https://www.ionos.es/digitalguide/servidores/know-how/arp-resolucion-de-direcciones-en-la-red/ [última visita: 05/08/2021]

[10] Irmak, E., & Bozdal, M., (2018). *Internet of Things (IoT): The Most Up-To-Date Challenges, Architectures, Emerging Trends and Potential Opportunities.* International Journal of Computer Applications, 179*, 20-27*. Disponible en:

https://www.semanticscholar.org/paper/Internet-of-Things-(IoT)%3A-The-Most-Up-To-Date-and-Irmak-Bozdal/d3fcc5b4bf251e15229eed34d95e89442fda73ec [última visita: 30/08/2021]

[11] J. Gubbi, R. Buyya, and S. Marusic, "Internet of Things ( IoT ): A Vision , Architectural Elements , and Future Directions," Futur. Gener. Comput. Syst., vol. 29, no. 1, pp. 1–19, 2013.

[12] Juncosa, M., (2019) *ARP: Address Resolution Protocol.* AprendeDeRedes. Disponible en: https://aprendederedes.com/redes/ip/arp-adress-resolution-protocol/ [última visita: 05/08/2021]

[13] Llamas, L., (2020) *Cómo instalar Mosquitto, el popular bróker MQTT.* Ingeniería informática y diseño. Disponible en: https://www.luisllamas.es/como-instalar-mosquitto-el-broker-mqtt/ [última visita: 21/03/2021]

[14] Llamas, L., (2021) *¿QUÉ ES MQTT? SU IMPORTANCIA COMO PROTOCOLO IOT.* Ingeniería informática y diseño. Disponible en: https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/ [última visita: 21/03/2021]

[15] Longo, E., Redondi, A. E. C., Cesana, M., Arcia-Moret, A., Manzoni. P., (2019) *MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers.* [Artículo científico] Disponible en: https://arxiv.org/pdf/1911.07622.pdf [última visita: 27/08/2021]

[16] Mininet., *Descargar/Empezar con Mininet.* Mininet.org. Disponible en: http://mininet.org/download/ [última visita: 18/07/2021]

[17] ONF., *Mininet.* Open Networking Fundation. Disponible en: https://opennetworking.org/mininet/ [última visita: 18/07/2021]

[18] Pachés, A. J., (2020) *Estudio del controlador SDN Ryu sobre una Raspberry-Pi Model 4.* [Trabajo Fin de Grado, Universitat Politècnica de València] Disponible en: https://tinyurl.com/45xfez8r [última visita: 22/08/2021]

[19] Park, J., Kim, H. y Kim, W. (2018). DM-MQTT: un MQTT eficiente basado en SDN Multicast para comunicaciones masivas de IoT. *Sensores (Basilea, Suiza), 18. Disponible en:* https://tinyurl.com/43v9eyu7 [última visita: 27/08/2021]

[20] PDAControl., (2016) *Instalacion de Mosquitto Broker MQTT en lubuntu (Ubuntu) Linux.* PDAControl. Disponible en: http://pdacontroles.com/instalacion-de-mosquitto-broker-mqtt-en/ [última visita: 01/04/2021]

[21] Rausch,T., Nastic, S. Dustdar, S., *EMMA: Distributes QoS-Aware MQTT Middleware for Edge Computing Applications.* Disponible en: https://dsg.tuwien.ac.at/team/trausch/pub/PID5190461.pdf [última visita: 27/08/2021]

[22] Ryu SDN Framework., (2017) *What's Ryu?.* Build SDN Agilely. Disponible en: https://ryu-sdn.org/ [última visita: 22/08/2021]

[23] Qa|cafe., *La opción de la marca de tiempo TCP.* Qa|cafe. Disponible en: https://www.qacafe.com/resources/tcp-timestamp-option/ [*última visita:* 13/08/2021]

[24] Scott, L., (2016) *Understanding the Ryu API: Dissecting Simple Switch.* Inside OpenFlow. Disponible en: https://inside-openflow.com/2016/07/21/ryu-api-dissecting-simple-switch/ [última visita: 17/04/2021]

[25] Standford-Clark, A., Troung, H. L., (2013) *MQTT For Sensor Networks (MQTT-SN)* Disponible en: https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf [última visita: 27/08/21]

[26] Valencia, B., Santacruz, S. Becerra, Y., Padilla, J.J., (2025) *Mininet: una herramienta versátil para emulación y prototipado de Redes Definidas por Software.* Entre ciencia e ingeniería. Disponible en: http://www.scielo.org.co/pdf/ecei/v9n17/v9n17a09.pdf [última visita: 18/08/2021]

[27] Wireshark., Disponible en: https://www.wireshark.org/ [*última visita:* 27/08/2021]

[28] Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H., (2014) *A Survey on Software-Defined Networking.* IEEE Xplore. Disponible en: https://ieeexplore.ieee.org/abstract/document/6834762/authors [última visita: 29/08/2021]

[29] Daniel F. Blandón. (2013) *OpenFlow: El protocolo del futuro.* Disponible en: https://www.academia.edu/14589122/OPENFLOW_EL_PROTOCOLO_DEL_FUTURO_Openflow_The_future_protocol [última visita: 25/06/2021]

[30] Open Networking. (2012) *OpenFlow Switch Specification.* Disponible en: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf [última visita: 26/06/2021]

# A. Código

Aquí se recogen algunos de los códigos Python utilizados durante el trabajo o fragmentos de ellos a los que se hace referencia durante la memoria.

## a. Definición de la topología de red

```
1  #!/usr/bin/python
2
3  from mininet.net import Mininet
4  from mininet.cli import CLI
5  from mininet.node import Controller, OVSKernelSwitch, RemoteController, OVSSwitch
6  from mininet.log import setLogLevel, info, error
7  from mininet.link import Intf, TCLink
8  from mininet.util import quietRun
9  from mininet.nodelib import NAT
10 from mininet.topolib import TreeNet
11
12 import re
13 import sys
14
15 def checkIntf( intf ):
16     "Make sure intf exists and is not configured."
17     if ( ' %s:' % intf ) not in quietRun( 'ip link show' ):
18         error( 'Error:', intf, 'does not exist!\n' )
19         exit( 1 )
20     ips = re.findall( r'\d+\.\d+\.\d+\.\d+', quietRun( 'ifconfig ' + intf ) )
21     if ips:
22         error( 'Error:', intf, 'has an IP address,' 'and is probably in use!\n' )
23         exit( 1 )
24
25 def myNetwork():
26         net = Mininet(controller=RemoteController,switch=OVSKernelSwitch, listenPort=6634)
27
28         info( '*** Add controller\n' )
29         c0=net.addController(name='c0',controller=RemoteController,protocols='OpenFlow13',
                                   ip='127.0.0.1',port=6633)
30
31         info( '*** Add hosts\n' )
32         num_hosts=4
33         hosts = []
34         for i in xrange(num_hosts):
```

```
35                  hostname = 'h%d' % (i+1)
36                  host = net.addHost(hostname, mac='00:00:00:00:00:%d' % (i+1),
                                   ip='192.168.1.%d' % (100+i+1))
37                  hosts.append(host)
38
39          info( '*** Add switches\n' )
40          num_switches = 3
41          switches = []
42          for i in xrange(num_switches):
43                  switch = net.addSwitch('s%d' % (i+1), cls=OVSSwitch,
                                        protocols='OpenFlow13')
44                  switches.append(switch)
45
46          info( '*** Add links\n')
47          net.addLink( net.get('s1'), net.get('s2') )
48          net.addLink( net.get('s1'), net.get('s3') )
49          net.addLink( net.get('s2'), net.get('h1') )
50          net.addLink( net.get('s2'), net.get('h2') )
51          net.addLink( net.get('s3'), net.get('h3') )
52          net.addLink( net.get('s3'), net.get('h4') )
53
54          info( '*** Starting network\n')
55          net.build()
56          net.start()
57
58          info( '*** Starting controllers\n')
59          for controller in net.controllers:
60                  controller.start()
61          net.get('s1').start([c0])
62
63          CLI(net)
64
65          net.stop()
66  if __name__ == '__main__':
67          setLogLevel( 'info' )
68          myNetwork()
```

## b. Función _handle_arp

```
1    def _handle_arp(self, datapath, in_port, pkt_ethernet, pkt_arp):
2        if pkt_arp.opcode != arp.ARP_REQUEST:
3           return
4
5        now = datetime.now().strftime('%Y/%m/%d %H:%M:%S.%f')
6        self.logger.info("### %s > ARP packet request received from %s", now,
                          pkt_arp.src_ip)
7
8        if pkt_arp.dst_ip == self.ip_addr:
9            pkt = packet.Packet()
10
11          pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                             dst=pkt_ethernet.src, src=self.mac_addr))
12
13           pkt.add_protocol(arp.arp(opcode=arp.ARP_REPLY, src_mac=self.mac_addr,
                             src_ip=self.ip_addr, dst_mac=pkt_arp.src_mac,
                             dst_ip=pkt_arp.src_ip))
14
15           now = datetime.now().strftime('%Y/%m/%d %H:%M:%S.%f')
16           self.logger.info("### %s > ARP packet reply sent to %s", now, pkt_arp.src_ip)
17           self._send_packet(datapath, in_port, pkt)
```

## c. Función generate_tcp_ack y _handle_tcp

```
1    def generate_tcp_ack(self, datapath, in_port, data, pkt_ethernet, pkt_ipv4, pkt_tcp):
2            flagsTCP = pkt_tcp.bits
3            if flagsTCP & tcp.TCP_SYN:
4                bits=(tcp.TCP_SYN | tcp.TCP_ACK)
5
6                seq=random.getrandbits(32)
7                ack=pkt_tcp.seq+1
8
9                self.ipToMAC[pkt_ipv4.src] = pkt_ethernet.src
10
11           elif flagsTCP & tcp.TCP_FIN:
12               bits=(tcp.TCP_FIN | tcp.TCP_ACK)
13
14               pkt_len = len(data)
15               header_size = len(pkt_ethernet) + len(pkt_ipv4) + len(pkt_tcp)
16               tcpPayloadSize = pkt_len - header_size
17
18               seq=pkt_tcp.ack
19               ack=pkt_tcp.seq+1 + tcpPayloadSize
20
21           else:
22               pkt_len = len(data)
23               header_size = len(pkt_ethernet) + len(pkt_ipv4) + len(pkt_tcp)
24               tcpPayloadSize = pkt_len - header_size
25               tcpPayload = data[header_size:]
26
27               seq=pkt_tcp.ack
28               ack=pkt_tcp.seq + tcpPayloadSize
29
30           pkt = packet.Packet()
31           pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                                  dst=pkt_ethernet.src, src=self.mac_addr))
32           pkt.add_protocol(ipv4.ipv4(dst=pkt_ipv4.src, src=self.ip_addr, proto=pkt_ipv4.proto))
33
34           tcpOptions = []
35           for op in pkt_tcp.option:
36               if op.kind == tcp.TCP_OPTION_KIND_NO_OPERATION:
37                   tcpOptions.append(tcp.TCPOptionNoOperation())
38               elif op.kind == tcp.TCP_OPTION_KIND_MAXIMUM_SEGMENT_SIZE:
39                   max_seg_size = op.max_seg_size
40                   tcpOptions.append(tcp.TCPOptionMaximumSegmentSize(max_seg_size=max_seg_size))
41               elif op.kind == tcp.TCP_OPTION_KIND_SACK_PERMITTED:
42                   tcpOptions.append(tcp.TCPOptionSACKPermitted())
43               elif op.kind == tcp.TCP_OPTION_KIND_WINDOW_SCALE:
44                   shift_cnt = op.shift_cnt
45                   tcpOptions.append(tcp.TCPOptionWindowScale(shift_cnt=shift_cnt))
46               elif op.kind == tcp.TCP_OPTION_KIND_TIMESTAMPS:
47                   ts_val = op.ts_val
48                   ts_ecr = op.ts_ecr
49                   tcpOptions.append(tcp.TCPOptionTimestamps(ts_val=ts_val+1,ts_ecr=ts_val))
50
51           newFlagsTCP = flagsTCP | tcp.TCP_ACK
52           pkt.add_protocol(tcp.tcp(src_port=pkt_tcp.dst_port, dst_port=pkt_tcp.src_port,
                                  seq=seq, ack=ack, offset=pkt_tcp.offset, bits=newFlagsTCP,
                                  window_size=pkt_tcp.window_size, csum=0, urgent=pkt_tcp.urgent,
                                  option=tcpOptions))
53
54           return pkt
55
56       def _handle_tcp(self, datapath, in_port, data, pkt_ethernet, pkt_ipv4, pkt_tcp):
57           flagsTCP = pkt_tcp.bits
```

```
58            now = datetime.now().strftime('%Y/%m/%d %H:%M:%S.%f')
59
60            if (flagsTCP & tcp.TCP_SYN) or (flagsTCP & tcp.TCP_FIN):
61                if flagsTCP & tcp.TCP_SYN:
62                    self.logger.info("### %s > TCP SYN received by the MQTT broker APP from
                                       %s", now, pkt_ipv4.src)
63
64                if flagsTCP & tcp.TCP_FIN:
65                    self.logger.info("### %s > TCP FIN received by the MQTT broker APP from
                                        %s", now, pkt_ipv4.src)
66
67                    subscriberStr = pkt_ipv4.src + "-" + str(pkt_tcp.src_port)
68                    if subscriberStr in self.subscribersTopics:
69                        topics = self.subscribersTopics[subscriberStr]
70                        for topic in topics:
71                            subscribersList = self.topicsSubscribers[topic]
72                            subscribersList = [x for x in subscribersList if x[0] != subscriberStr]
73                            self.topicsSubscribers[topic] = subscribersList
74                            if not self.topicsSubscribers[topic]:
75                                del self.topicsSubscribers[topic]
76
77                    if subscriberStr in self.subscribersTopics:
78                        self.subscribersTopics.pop(subscriberStr)
79                    if subscriberStr in self.subscribersTCPInfo:
80                        self.subscribersTCPInfo.pop(subscriberStr)
81
82                    self.logger.debug("######### topicsSubscribers:  %s", self.topicsSubscribers)
83                    self.logger.debug("######### subscribersTopics:  %s", self.subscribersTopics)
84                    self.logger.debug("######### subscribersTCPInfo: %s", self.subscribersTCPInfo)
85
86                ackPkt = self.generate_tcp_ack(datapath, in_port, data, pkt_ethernet,
                                                 pkt_ipv4, pkt_tcp)
87
88                now = datetime.now().strftime('%Y/%m/%d %H:%M:%S.%f')
89                if flagsTCP & tcp.TCP_SYN:
90                    self.logger.info("### %s > TCP SYN+ACK sent to %s", now, pkt_ipv4.src)
91                if flagsTCP & tcp.TCP_FIN:
92                    self.logger.info("### %s > TCP FIN+ACK sent to %s", now, pkt_ipv4.src)
93
94                self._send_packet(datapath, in_port, ackPkt)
95                return
96
97            else:
98                subscriberStr = pkt_ipv4.src + "-" + str(pkt_tcp.src_port)
99                if subscriberStr in self.subscribersTCPInfo:
100                   pkt_len = len(data)
101                   header_size = len(pkt_ethernet) + len(pkt_ipv4) + len(pkt_tcp)
102                   tcpPayloadSize = pkt_len - header_size
103
104                   if pkt_tcp.option:
105                       timestamps = [op for op in pkt_tcp.option if op.kind ==
                                     tcp.TCP_OPTION_KIND_TIMESTAMPS][0]
106
107                   self.subscribersTCPInfo[subscriberStr] = [pkt_tcp.ack, pkt_tcp.seq +
                                                               tcpPayloadSize,
                                                               timestamps.ts_val]
108                   self.logger.debug("######### subscribersTCPInfo updated for %s: %s",
                                        subscriberStr, self.subscribersTCPInfo[subscriberStr])
```

## d. Función _handle_mqtt

```
1   def _handle_mqtt(self, datapath, in_port, data, pkt_ethernet, pkt_ipv4, pkt_tcp, pkt_mqtt):
2           tcpPayloadType = pkt_tcp.get_payload_type(pkt_tcp.src_port, pkt_tcp.dst_port)
3           now = datetime.now().strftime('%Y/%m/%d %H:%M:%S.%f')
4           mqttPacketType = pkt_mqtt.mqttControlPacketType
5
6           pkt_len = len(data)
7           header_size = len(pkt_ethernet) + len(pkt_ipv4) + len(pkt_tcp)
8           tcpPayloadSize = pkt_len - header_size
9           tcpPayload = data[header_size:]
10
11          self.logger.debug("##### %s > Received TCP packet (seq=%d, ack=%d), payload type:
                              %s, MQTT packet: %s, MQTT packet type: %d (%s), MQTT packet
                              content (%d bytes): %s", now, pkt_tcp.seq, pkt_tcp.ack,
                              tcpPayloadType, pkt_mqtt, mqttPacketType,
                              pkt_mqtt.mqttPacketControlTypeStr[mqttPacketType],
                              tcpPayloadSize, tcpPayload)
12
13          if mqttPacketType == 1 or mqttPacketType == 8 or mqttPacketType == 12:
14              pkt = packet.Packet()
15              pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                                 dst=pkt_ethernet.src, src=self.mac_addr))
16              pkt.add_protocol(ipv4.ipv4(dst=pkt_ipv4.src, src=self.ip_addr,
                                 proto=pkt_ipv4.proto))
17              tcpOptions = []
18              for op in pkt_tcp.option:
19                  if op.kind == tcp.TCP_OPTION_KIND_NO_OPERATION:
20                      tcpOptions.append(tcp.TCPOptionNoOperation())
21                  elif op.kind == tcp.TCP_OPTION_KIND_MAXIMUM_SEGMENT_SIZE:
22                      max_seg_size = op.max_seg_size
23                      tcpOptions.append(tcp.TCPOptionMaximumSegmentSize
                                          (max_seg_size=max_seg_size))
24                  elif op.kind == tcp.TCP_OPTION_KIND_SACK_PERMITTED:
25                      tcpOptions.append(tcp.TCPOptionSACKPermitted())
26                  elif op.kind == tcp.TCP_OPTION_KIND_WINDOW_SCALE:
27                      shift_cnt = op.shift_cnt
28                      tcpOptions.append(tcp.TCPOptionWindowScale(shift_cnt=shift_cnt))
29                  elif op.kind == tcp.TCP_OPTION_KIND_TIMESTAMPS:
30                      ts_val = op.ts_val
31                      ts_ecr = op.ts_ecr
32                      tcpOptions.append(tcp.TCPOptionTimestamps(ts_val=ts_val+1,
                                          ts_ecr=ts_val))
33
34              pkt.add_protocol(tcp.tcp(src_port=pkt_tcp.dst_port,dst_port=pkt_tcp.src_port,
                                         seq=pkt_tcp.ack, ack=pkt_tcp.seq + tcpPayloadSize,
                                         offset=pkt_tcp.offset, bits=pkt_tcp.bits,
                                         window_size=pkt_tcp.window_size, csum=0,
                                         urgent=pkt_tcp.urgent, option=tcpOptions))
35              now = datetime.now().strftime('%Y/%m/%d %H:%M:%S.%f')
36              if mqttPacketType == 1:
37                  self.logger.info("### %s > MQTT CONNECT (protocolName=%s, version=%s,
                                     connectionFlags=%s, keepAlive=%d, clientID=%s,
                                     willTopic=%s, willMessage=%s, userName=%s, password=%s)
                                     received from %s", now, pkt_mqtt.protocolName,
                                     pkt_mqtt.version, format(pkt_mqtt.connectionFlags,
                                     '08b'), pkt_mqtt.keepAlive, pkt_mqtt.clientID,
                                     pkt_mqtt.willTopic, pkt_mqtt.willMessage,
                                     pkt_mqtt.userName, pkt_mqtt.password, pkt_ipv4.src)
38
39                  returnCode=5
40                  if pkt_mqtt.userName.decode("utf-8") == self.mqtt_user and
                                         pkt_mqtt.password.decode("utf-8") == self.mqtt_pass:
```

```
41                  returnCode=0
42
43              pkt.add_protocol(mqtt.mqtt(2, 2, returnCode=returnCode, messageID=None,
                                 qos=None, topic=None, message=None))
44              self.logger.info("### %s > MQTT CONNACK (returnCode=%d) sent to %s", now,
                                  returnCode, pkt_ipv4.src)
45
46          elif mqttPacketType == 8:
47
48              if pkt_mqtt.topic in self.topicsSubscribers:
49                  subscribersList = self.topicsSubscribers[pkt_mqtt.topic]
50                  subscribersList.append([pkt_ipv4.src + "-" + str(pkt_tcp.src_port),
                                          pkt_mqtt.qos])
51              else:
52                  self.topicsSubscribers[pkt_mqtt.topic] = [[pkt_ipv4.src + "-" +
                                              str(pkt_tcp.src_port), pkt_mqtt.qos]]
53
54              subscriberStr = pkt_ipv4.src + "-" + str(pkt_tcp.src_port)
55              if subscriberStr in self.subscribersTopics:
56                  subscriberInfo = self.subscribersTopics[subscriberStr]
57                  subscriberInfo.append(pkt_mqtt.topic)
58              else:
59                  self.subscribersTopics[subscriberStr] = [pkt_mqtt.topic]
60
61              if pkt_tcp.option:
62                  timestamps = [op for op in pkt_tcp.option if op.kind ==
                                  tcp.TCP_OPTION_KIND_TIMESTAMPS][0]
63
64              subscriberStr = pkt_ipv4.src + "-" + str(pkt_tcp.src_port)
65
66              self.subscribersTCPInfo[subscriberStr] = [pkt_tcp.ack, pkt_tcp.seq +
                                                          tcpPayloadSize,
                                                          timestamps.ts_val]
67
68              self.logger.info("### %s > MQTT SUBSCRIBE (grantedQoS=%s, messageID=%d,
                                  topic=%s) received from %s", now, pkt_mqtt.qos,
                                  pkt_mqtt.messageID, pkt_mqtt.topic, pkt_ipv4.src)
69              self.logger.debug("######### topicsSubscribers:  %s", self.topicsSubscribers)
70              self.logger.debug("######### subscribersTopics:  %s", self.subscribersTopics)
71              self.logger.debug("######### subscribersTCPInfo: %s", self.subscribersTCPInfo)
72              grantedQoS = pkt_mqtt.qos
73              messageID = pkt_mqtt.messageID
74              pkt.add_protocol(mqtt.mqtt(9, 3, returnCode=None, messageID=messageID,
                                          qos=grantedQoS, topic=None, message=None))
75              self.logger.info("### %s > MQTT SUBACK (grantedQoS=%s, messageID=%d) sent
                                  to %s", now, grantedQoS, messageID, pkt_ipv4.src)
76
77          elif mqttPacketType == 12:
78              self.logger.info("### %s > MQTT PING REQUEST received from %s,
                                  sending TCP ACK...", now, pkt_ipv4.src)
79
80              ackPkt = self.generate_tcp_ack(datapath, in_port, data, pkt_ethernet,
                                              pkt_ipv4, pkt_tcp)
81
82              self.logger.info("### %s > TCP ACK sent to %s", now, pkt_ipv4.src)
83
84              self._send_packet(datapath, in_port, ackPkt)
85
86              pkt.add_protocol(mqtt.mqtt(13, 0, returnCode=None, messageID=None,
                                          qos=None, topic=None, message=None))
87              self.logger.info("### %s > MQTT PING RESPONSE sent to %s", now, pkt_ipv4.src)
88
89          self._send_packet(datapath, in_port, pkt)
90          return
```

```
 91
 92             elif mqttPacketType == 3:
 93                 self.logger.info("### %s > MQTT PUBLISH (topic=%s, message=%s)
                                     received from %s, sending TCP ACK...", now, pkt_mqtt.topic,
                                     pkt_mqtt.message, pkt_ipv4.src)
 94
 95                 ackPkt = self.generate_tcp_ack(datapath, in_port, data, pkt_ethernet,
                                                    pkt_ipv4, pkt_tcp)
 96
 97                 self.logger.info("### %s > TCP ACK sent to %s", now, pkt_ipv4.src)
 98
 99                 self._send_packet(datapath, in_port, ackPkt)
100
101                 if pkt_mqtt.topic in self.topicsSubscribers:
102                     subscribersList = self.topicsSubscribers[pkt_mqtt.topic]
103                     for x in subscribersList:
104                         qos = x[1]
105                         subscriberStr = x[0]
106                         [subscriberIpStr, subscriberTCPPortStr] = subscriberStr.split("-")
107                         macDstAddr = self.ipToMAC[subscriberIpStr]
108                         if subscriberStr in self.subscribersTCPInfo:
109                             subscriberTcpInfo =self.subscribersTCPInfo[subscriberStr]
110                             subscriberSeq = subscriberTcpInfo[0]
111                             subscriberAck = subscriberTcpInfo[1]
112                             subscriberTS = subscriberTcpInfo[2]
113
114                             if datapath.id not in self.mac_to_port:
115                                 self.logger.info("### %s > CANNOT FORWARD MQTT PUBLISH since
                                                  switch is not included in the MAC table.
                                                  Restart mininet and perform a pingall!!!",
                                                  now)
116                                 return
117
118                             else:
119                                 if macDstAddr not in self.mac_to_port[datapath.id]:
120                                     self.logger.info("### %s > CANNOT FORWARD MQTT PUBLISH
                                                      since the MAC address is not included
                                                      in the MAC table of switch %s. Try to
                                                      perform a pingall. If it does not work,
                                                      restart mininet!!!", now, datapath.id)
121                                     return
122
123                             self.logger.info("### %s > forward MQTT PUBLISH (topic=%s,
                                              message=%s) to %s (MAC=%s, TCP port=%s, seq=%d,
                                              ack=%d, ts_val=%d)", now, pkt_mqtt.topic,
                                              pkt_mqtt.message, subscriberIpStr, macDstAddr,
                                              subscriberTCPPortStr, subscriberSeq,
                                              subscriberAck, subscriberTS)
124
125                             pkt = packet.Packet()
126                             pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                                                 dst=macDstAddr, src=self.mac_addr))
127                             pkt.add_protocol(ipv4.ipv4(dst=subscriberIpStr, src=self.ip_addr,
                                                           proto=pkt_ipv4.proto))
128                             tcpOptions = []
129                             for op in pkt_tcp.option:
130                                 if op.kind == tcp.TCP_OPTION_KIND_NO_OPERATION:
131                                     tcpOptions.append(tcp.TCPOptionNoOperation())
132                                 elif op.kind == tcp.TCP_OPTION_KIND_MAXIMUM_SEGMENT_SIZE:
133                                     max_seg_size = op.max_seg_size
134                                     tcpOptions.append(tcp.TCPOptionMaximumSegmentSize
                                                      (max_seg_size=max_seg_size))
135                                 elif op.kind == tcp.TCP_OPTION_KIND_SACK_PERMITTED:
136                                     tcpOptions.append(tcp.TCPOptionSACKPermitted())
```

```
137                             elif op.kind == tcp.TCP_OPTION_KIND_WINDOW_SCALE:
138                                 shift_cnt = op.shift_cnt
139                                 tcpOptions.append(tcp.TCPOptionWindowScale
                                                  (shift_cnt=shift_cnt))
140                             elif op.kind == tcp.TCP_OPTION_KIND_TIMESTAMPS:
141                                 ts_val = op.ts_val
142                                 ts_ecr = op.ts_ecr
143                                 tcpOptions.append(tcp.TCPOptionTimestamps
                                                  (ts_val=subscriberTS+1,
                                                   ts_ecr=subscriberTS))
144
145                         pkt.add_protocol(tcp.tcp(src_port=mqtt.TCP_SERVER_PORT,
                                                  dst_port=int(subscriberTCPPortStr),
                                                  seq=subscriberSeq, ack=subscriberAck,
                                                  offset=pkt_tcp.offset,bits=pkt_tcp.bits,
                                                  window_size=pkt_tcp.window_size,
                                                  csum=0, urgent=pkt_tcp.urgent,
                                                  option=tcpOptions))
146
147                         pkt.add_protocol(mqtt.mqtt(3, pkt_mqtt.mqttRemainingLength,
                                                   returnCode=None, messageID=None,
                                                   qos=None, topic=str(pkt_mqtt.topic.decode()),
                                                   message=str(pkt_mqtt.message.decode())))
148
149                         self.logger.debug("MAC table=%s", self.mac_to_port)
150                         self.logger.debug("datapath.id=%s", datapath.id)
151                         self.logger.debug("MAC addr=%s", macDstAddr)
152                         self.logger.debug("MAC table=%s, MAC addr=%s",
                                           self.mac_to_port[datapath.id], macDstAddr)
153                         self.logger.debug("### %s >>>>> MQTT PUBLISH message: %s
                                          (datapath=%s, port=%s)", now, pkt, datapath.id,
                                           self.mac_to_port[datapath.id][macDstAddr])
154
155                         self._send_packet(datapath, self.mac_to_port[datapath.id]
                                           [self.ipToMAC[subscriberIpStr]], pkt)
156                 return
157
158             elif mqttPacketType == 14:
159                 self.logger.info("### %s > MQTT DISCONNECT received from %s", now, pkt_ipv4.src)
160             else:
161                 self.logger.info("### %s > MQTT message with Packet Type=%d received from %s,
                                  NOT SUPPORTED YET!", now, mqttPacketType, pkt_ipv4.src)
```

## e. Función send_packet

```
1 def _send_packet(self, datapath, port, pkt):
2         pkt_ethernet = pkt.get_protocols(ethernet.ethernet)[0]
3         pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
4         if pkt_ipv4:
5             pkt_tcp = pkt.get_protocol(tcp.tcp)
6             if pkt_tcp:
7                 subscriberStr = pkt_ipv4.dst + "-" + str(pkt_tcp.dst_port)
8                 self.logger.debug("######### subscriber: %s, subscribersTCPInfo=%s",
                                   subscriberStr, self.subscribersTCPInfo)
9                 if subscriberStr in self.subscribersTCPInfo:
10                    tcpPayloadSize=0
11                    if len(pkt.protocols) == 4:
12                        tcpPayload = pkt.protocols[-1]
13                        tcpPayloadSize = len(tcpPayload)
14
15                    if pkt_tcp.option:
16                        timestamps = [op for op in pkt_tcp.option if op.kind ==
                                      tcp.TCP_OPTION_KIND_TIMESTAMPS][0]
17
18                    self.subscribersTCPInfo[subscriberStr] = [pkt_tcp.seq +
                                                             tcpPayloadSize, pkt_tcp.ack,
                                                             timestamps.ts_ecr]
19
20                    self.logger.debug("######### subscribersTCPInfo updated for %s (TCP
                                   payload size=%d): %s", subscriberStr, tcpPayloadSize,
                                   self.subscribersTCPInfo[subscriberStr])
21
22        ofproto = datapath.ofproto
23        parser = datapath.ofproto_parser
24        pkt.serialize()
25        self.logger.info("packet-out %s" % (pkt,))
26        data = pkt.data
27        actions = [parser.OFPActionOutput(port=port)]
28
29        out = parser.OFPPacketOut(datapath=datapath, buffer_id=ofproto.OFP_NO_BUFFER,
                                  in_port=ofproto.OFPP_CONTROLLER, actions=actions,
                                  data=data)
30        datapath.send_msg(out)
```