UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA MQTT SIN BRÓKER BASADO EN SDN

Autor

Álvaro Arco Castillo

Director

Jorge Navarro Ortiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, septiembre de 2019

UNIVERSIDAD DE GRANADA

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA MQTT SIN BRÓKER BASADO EN SDN

Autor

Álvaro Arco Castillo

Director

Jorge Navarro Ortiz

Diseño e implementación de un sistema MQTT sin bróker basado en SDN.

Álvaro Arco Castillo

Palabras clave: Internet de las Cosas, MQTT, SDN, OpenFlow, Controlador, OpenDaylight.

Resumen

La tecnología ha evolucionado de manera exponencial en los últimos años y el concepto de Internet de las Cosas (IoT) está cada vez más presente en nuestra sociedad. En la actualidad, prácticamente cualquier dispositivo puede conectarse a Internet e interactuar con otros objetos, tendiendo a un mundo interconectado en todos los ámbitos de la vida cotidiana y profesional.

Se estima que actualmente hay varios miles de millones de dispositivos conectados. Muchos de estos dispositivos son sensores y actuadores que generan poco tráfico y requieren de protocolos sencillos que permitan el intercambio de información. Uno de estos protocolos es MQTT (*Message Queuing Telemetry Transport*), protocolo basado en el paradigma publicador/suscriptor. Para ello, los dispositivos intercambian información con un intermediario o bróker.

Por otro lado, la tecnología se está inclinando hacia un soporte virtual, en el que los tradicionales límites físicos se reduzcan considerablemente. En este contexto, las redes diseñadas por software (SDN), al separar el plano de control (*software*) del plano de datos (*hardware*), facilitan la implementación de servicios de red de una manera determinista, dinámica y escalable, evitando al administrador de red gestionar dichos servicios a bajo nivel.

El presente trabajo se centra precisamente en la convergencia entre los dos conceptos anteriores aplicando las ventajas de las SDN a uno de los principales protocolos del IoT, MQTT. Para ello, se propone una implementación de MQTT, compatible con los equipos finales, pero que elimine la necesidad de un bróker como elemento independiente para sus comunicaciones. Esto permitirá controlar mejor el tráfico en la red y eliminar un punto único de fallo, ya que las implementaciones actuales no pueden funcionar si el bróker no está disponible.

Esta solución se realizará para redes SDN, haciendo uso de los mensajes OpenFlow y se programará como un módulo de un controlador SDN. El controlador elegido será OpenDaylight, que se encargará de llevar a cabo las funciones del bróker MQTT además de poner la red en funcionamiento. Para ello se deberán programar distintos mensajes de respuesta que permitan que la red se comporte como lo haría en un escenario MQTT típico.

MQTT system without a broker. SDN based design and implementation.

Álvaro Arco Castillo

Keywords: Internet de las Cosas, MQTT, SDN, OpenFlow, Controller, OpenDaylight.

Abstract

Technology has evolved exponentially in recent years and the concept of Internet of Things (IoT) is frequently used in our society. Nowadays, almost every device can connect to Internet and interact with other devices, approaching us to an interconnected world in a lot of areas of our everyday and professional life.

Estimations say that there are currently several billion connected devices. Many of those devices are sensors and actuators that generate little traffic and require simple protocols that allow information exchange. One of these protocols is MQTT (Message Queuing Telemetry Transport), a protocol based on the publisher/subscriber paradigm. In order to do this, the devices exchange information with a broker.

Technology is also approaching virtual support, where the traditional physical limits are considerably reduced. In this context, software designed networks (SDN), by separating the control plane (software) from the data plane (hardware), facilitate the implementation of network services in a deterministic, dynamic and scalable way, making it easier for the administrator to manage these low level services.

This paper focuses on the convergence between the two previous concepts by applying the advantages of SDN networks to one of the main protocols of IoT, MQTT. For this reason, an implementation of MQTT is proposed, suitable for user devices, but removing the need of a broker as an independent element for the communications. This will allow a better traffic control and will eliminate a possible failure point, as current implementations cannot work if the broker is not available.

This solution will be implemented for SDN networks, making use of OpenFlow messages, and will be programmed as a SDN controller module. The chosen controller will be OpenDaylight, which will be responsible for implementing the functions of the MQTT broker, in addition to operating the network. In order to do this, different response messages must be programmed to make the network behave as it would in a typical MQTT scenario.

Yo, **Álvaro Arco Castillo**, 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 XXX, 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: Álvaro Arco Castillo

Granada a 20 de agosto de 2019 .

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

Informa:

Que el presente trabajo, titulado ***Diseño e implementación de un sistema MQTT sin bróker basado en SDN***, ha sido realizado bajo su supervisión por **Álvaro Arco Castillo**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 20 de agosto de 2019.

El director:

Jorge Navarro Ortiz

Agradecimientos

La entrega de este proyecto supone el fin de una de las etapas más importantes de mi vida, tanto a nivel académico como a nivel personal. Por tanto me gustaría agradecer a muchas personas que han hecho esto posible.

En primer lugar a mi tutor Jorge, por proponerme un tema tan interesante con el que he disfrutado trabajando, y por ayudarme y orientarme en mis momentos más desesperados sacando tiempo de donde no lo tenía.

En segundo lugar tengo que darle las gracias a toda mi familia, especialmente a mis padres, por haber creído siempre en mí y haber celebrado cada uno de mis pequeños logros a los largo de mi vida con mayor orgullo y entusiasmo del que yo mismo siento.

Por supuesto también tengo que agradecer a todos mis amigos su constante interés y preocupación por un tema tan lejano a algunos de ellos, pero que siempre han estado ahí cuando necesitaba despejarme o consuelo.

Por último y especialmente gracias a la persona más importante de mi vida en estos últimos años, mi compañera de penas y alegrías a la que con tanta frecuencia he acudido en busca de ayuda. Sabes que sin tí esto no habría sido posible. Gracias Verónica.

Índice general

Índice de figuras

Índice de tablas

Glosario de siglas

AD-SAL	Aplication Driven Service Abstraction Layer
AES	Advanced Encryption Standard
API	Application Programming Interface
CPU	Central Processing Unit
DES	Data Encryption Standard
DUP	Duplicate Message Flag
IDC	International Data Corporation
IoT	Internet of Things
IP	Internet Protocol
ITU	International Telecommunication Union
LLDP	Link Layer Discovery Protocol
LSB	Least Significant Bit
M2M	Machine To Machine
MAC	Media Access Control
MD-SAL	Model Driven Service Abstraction Layer
MQTT	Message Queuing Telemetry Transport
MSB	Most Significant Bit
NVF	Network Functions Virtualization
OFP	OpenFlow Protocol
OFS	OpenFlow Switch
ONF	Open Networking Foundation
OSI	Open System Interconnection
OVSDB	Open vSwitch Database Management Protocol
PCEP	Path Computation Element Protocol

QoS	Quality of Service
RPC	Remote Procedure Call
SDN	Software Defined Network
SNMP	Simple Network Management Protocol
SSL	Secure Sockets Layer
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
YANG	Yet Another Next Generation

1

Introducción

En este primer capítulo se presenta el proyecto y se justifica la necesidad de llevarlo a cabo. Pretende despertar el interés del lector por el trabajo contextualizándolo y explicando la estructura de la memoria.

1.1. Motivación.

Este trabajo, «Diseño e implementación de un sistema MQTT sin bróker basado en SDN», pretende mejorar del protocolo MQTT, uno de los más utilizados en el mundo IoT. Consiste, tal y como indica el título, en diseñar e implementar un escenario MQTT sin bróker como elemento de red diferenciado, pero que mantenga el mismo comportamiento que uno con bróker. Para ello se hace uso de un controlador, elemento característico de las redes SDN.

Esta nueva implementación del protocolo MQTT facilita enormemente el control del entorno, ya que en lugar de que los mensajes sean dirigidos por un bróker de código cerrado, pasarán por el controlador SDN cuyo comportamiento podemos cambiar a voluntad.

1.2. Contexto.

Para comprender el entorno de este proyecto es necesario profundizar en dos conceptos actuales: *Internet of Things* (IoT) y *Software Defined Network* (SDN).

En primer lugar, el protocolo MQTT surge en el ámbito del IoT como un protocolo sencillo y ligero, ideal para los dispositivos IoT porque, a menudo, tienen limitaciones de potencia, consumo, y ancho de banda. Si bien hay muchos otros protocolos IoT con estas características, MQTT es de los más utilizados porque además, se trata de un protocolo de sistema de mensajes asíncrono, es decir, que separa al emisor y al receptor del mensaje tanto en el tiempo como en el espacio y, por lo tanto, es escalable en ambientes de red que no sean de confianza.

En cuanto a SDN, su principal ventaja es que las redes se diseñan por *software*, tal y como su nombre indica, lo que elimina las limitaciones de muchas redes como consecuencia del *hardware*. Aplicando esta tecnología al protocolo MQTT, se pueden obtener comunicaciones más controlables de forma rápida y sencilla.

1.2.1. *Internet Of Things.*

Desde hace unos años se está produciendo un crecimiento exponencial del número de dispositivos que tienen conexión a Internet. Actualmente, no solo acceden a la red

los móviles, los ordenadores y las tabletas, también lo hacen aparatos como sistemas de climatización, sensores, coches, frigoríficos, cámaras, bombillas, etc. Este es el conocido *Internet of Things* (IoT), que persigue que todos los dispositivos se comuniquen entre sí dotándolos de mayor inteligencia e independencia.

1.2.1.1. Características.

Según la Unión Internacional de Telecomunicaciones (ITU) [1] se define el Internet de las Cosas (IoT) como «una infraestructura global para la sociedad de la información, que permite servicios avanzados mediante la interconexión de elementos (físicos y virtuales), basados en tecnologías de comunicación e información interoperables existentes y en evolución.»

Una *cosa* se describe en este ámbito como una instancia del mundo físico o del mundo de la información que se puede identificar de manera única y que se puede integrar en las redes de comunicación. Para que los dispositivos IoT se puedan conectar, estas *cosas* deben describirse juntos a sus capacidades para garantizar su interoperabilidad [2].

Al estar Internet presente en todas partes, las personas pueden monitorizar y controlar los objetos conectados a Internet desde cualquier lugar a través de la infraestructura IoT. Aunque todavía no se ha desarrollado una infraestructura fija que siempre deba utilizarse con IoT, han aparecido numerosas aplicaciones que hacen su propuesta [2]. La Figura 1.1 representa este concepto asignando dispositivos del mundo físico al mundo digital a través de las redes de comunicación.

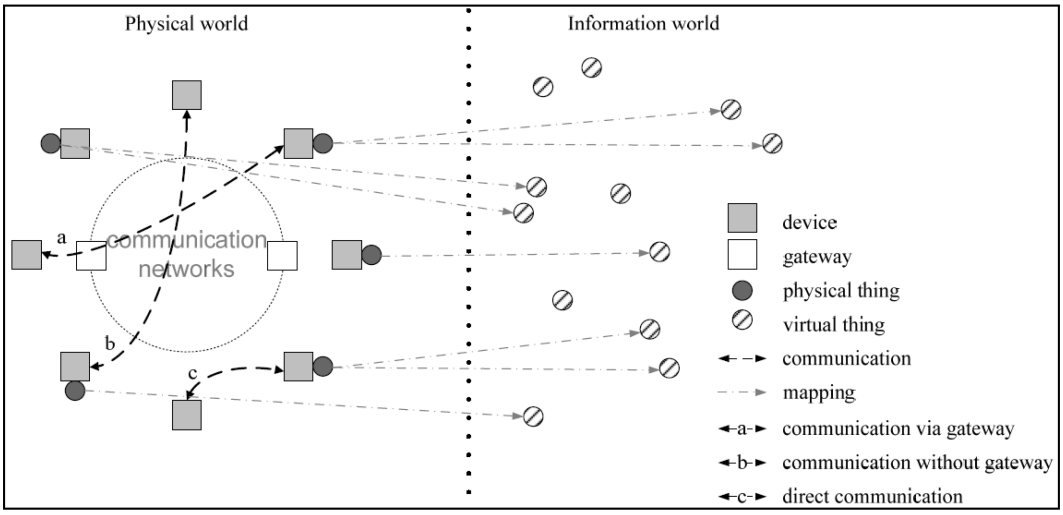


Figura 1.1: Descripción técnica del IoT [2].

En el futuro los dispositivos serán cada vez más inteligentes, ofreciendo mejores servicios digitales y creando la llamada «malla digital inteligente». Con malla se habla de las conexiones de un conjunto de personas, dispositivos, contenido o servicios, todos ellos en su ámbito digital. Además debe ser inteligente, lo que se refiere al uso de inteligencia artificial con aprendizaje automático que se introduzca en todos los campos y tecnologías posibles facilitando su uso y mejorando su eficiencia.

1.2.1.2. Impacto social y económico.

El IoT que se está desarrollando es el paso evolutivo natural que está provocando una nueva revolución de Internet. A medida que Internet se fue haciendo pública a principios

de 1990, surgió una ola de su explotación y despliegue centrada en los servicios y aplicaciones cotidianos. Fue una revolución que provocó la digitalización de una gran cantidad de servicios y empresas que se centraron en mejorar la vida del ser humano [3].

El impacto económico ha sido tremendo debido principalmente a los nuevos modelos de negocio y la nube. Actualmente, se experimenta una nueva forma de vida debido a la llegada de Internet a la gran mayoría de los hogares y entornos de trabajo [3]. Los datos estimados del aumento de dispositivos con conexión en los próximos años se muestran en la Figura 1.2.

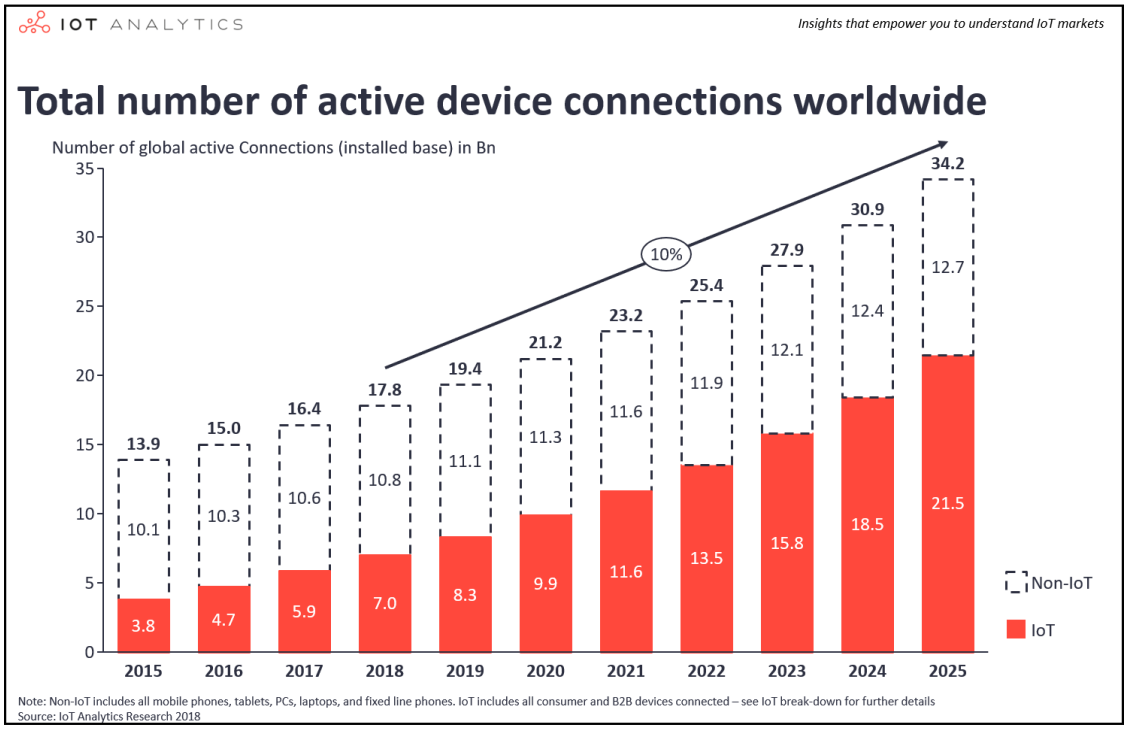


Figura 1.2: Pronóstico de crecimiento de dispositivos IoT [4].

1.2.1.3. Aplicaciones IoT.

Los sistemas IoT cuentan con un gran número de aplicaciones en los que pueden resultar útiles. A continuación se destacan las más populares según un ranking basado en el número de búsquedas en Google, las conversaciones en Twitter y las publicaciones en LinkedIn. A continucación se muestras las diez aplicaciones más comunes ordenadas siguiento los criterios mencionados, siendo la primera el hogar inteligente (*smart home*) [3, 5].

1. ***Smart home*:** mejoran el estilo de vida personal facilitando la monitorización de los aparatos y sistemas domésticos de forma remota (domótica).

2. ***Wearables*:** conjunto de aparatos como relojes o zapatillas que se incorporan en alguna parte del cuerpo, interactuando de forma continua con el usuario.

3. ***Smart city*:** mejoran la calidad de vida facilitando a los residentes encontrar información de interés y proporcionando servicios.

4. ***Smart grids*:** ayudan a los proveedores a controlar y administrar los recursos para proporcionar energía proporcionalmente al aumento de la población.

5. ***Industrial internet*:** se usan dispositivos robóticos para completar las tareas de fabricación con una participación humana mínima y sensores para controlar los procesos industriales en sí mismos y el estado del equipo.

6. ***Connected car*:** el coche conectado se acerca lentamente porque los ciclos de desarrollo en la industria automotriz generalmente duran de 2 a 4 años.

7. ***Connected health*:** incorporación de sensores y actuadores en los pacientes y sus medicamentos para fines de seguimiento.

8. ***Smart retail*:** la publicidad basada en la proximidad como un subconjunto del comercio minorista inteligente está empezando a despegar.

9. ***Smart suplay chain*:** las soluciones para rastrear bienes mientras están en el camino o hacer que los proveedores intercambien información de inventario.

10. ***Smart farming*:** aplicable a la utilización de recursos agrícolas, manejo cuantitativo en la producción, monitoreo ambiental, manejo de calidad, seguridad o trazabilidad de los cultivos.

La relevancia de las tecnologías IoT, así como las ventajas que ofrecen en nuestro día a día o su crecimiento continuo (Figura 1.2), se han hecho realidad en los últimos años. Sin embargo, esto también supone algunos inconvenientes a tener en cuenta como que la información es cada vez más sensible, el crecimiento de vulnerabilidades a la vez que aumentan los dispositivos, o el necesitar una mejor política de seguridad, que son tareas cada vez más difíciles de controlar. Esto afecta tanto a los consumidores como a los desarrolladores y fabricantes. Además de los poblemas de seguridad y privacidad mencionados hay otros muchos aspectos de los que preocuparse para conseguir una aplicación de IoT exitosa [3, 5, 7].

1.2.1.4. Modelo de referencia.

Los sistemas y servicios IoT requieren el desarrollo de arquitecturas capaces de operar eficientemente. Hoy en día hay numerosos estándares de IoT para facilitar y simplificar el trabajo a los programadores de aplicaciones y proveedores de servicios. Este apartado se centra en el modelo de referencia para IoT según la ITU representado en la Figura 1.3. Es un modelo con cuatro capas horizontales y dos verticales trasversales en las que se pueden incluir los distintos protocolos IoT [3, 5, 6, 7].

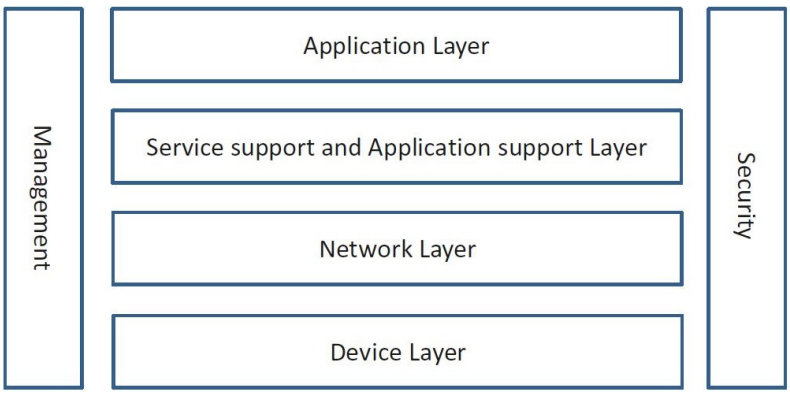


Figura 1.3: Modelo de referencia IoT según ITU [3].

- **Capa de dispositivo:** incluye las funcionalidades de los nodos y los *gateway*s. Es la más baja de la jerarquía.
    - Protocolos: WiFi, Bluetooth Low Energy, LoRaWAN, Z-Wave, ZigBee Smart, Weighless, HomePlug GP, DECT/ULE, 3G/LTE, NFC,802.11ah, 802.15.4e, WirelessHART, ANT+, LTE-A, etc.
- **Capa de red:** encapsula los datos del dispositivo y los convierte al protocolo correspondiente en la capa de red. Incluye todas las funcionalidades de los protocolos de red y transporte del modelo OSI.
    - Protocolos: 6LowPAN, 6TiSCH, 6Lo, CORPL, CARP, Thread, RPL, etc.
- **Capa de soporte de servicio y la aplicación:** funcionalidades genéricas y específicas para habilitar las aplicaciones y los servicios de IoT.
- **Capa de aplicación:** incluye las aplicaciones y servicios de (I)IoT. Es la capa jerárquica más alta de la jerarquía.
    - Protocolos: MQTT, SMQTT, CoRE, DDS, AMQP, XMPP, CoAP, Web-sockets, etc.
- **Capa de gestión:** capa vertical que incluye las funcionalidades de aplicación genéricas (configuración, topología, rendimiento, fallos, seguridad, administración de cuentas,...) y específicas (cumplen con los requisitos de la aplicación).
    - Protocolos: IEEE 1905, IEEE 1451, etc.
- **Capa de seguridad:** capa vertical que incluye las funcionalidades de aplicación genéricas (autorización, autenticación, integridad, confidencialidad y control de acceso) y específicas (cumplen con los requisitos de la aplicación).
    - Protocolos: TCG, Oath, SMACK, SASL, ISASecure, ace, STLS, Dice, IPSec, etc.

Cabe destacar que aunque hay multitud de protocolos IoT, el único al que se hace referencia en este proyecto es MQTT. Del mismo modo, es importante tener en cuenta que los protocolos enumerados anteriormente son específicos de IoT, por lo que hay protocolos como ARP y TCP que no aparecen en este apartado pero que serán relevantes en el trabajo.

1.2.2. *Software Defined Networking.*

Las arquitecturas de red tradicionales han comenzado a no satisfacer los requerimientos actuales de operadores, empresas y usuarios. Algunos de los métodos que han aparecido para cumplir estos nuevos requerimientos son NFV (*Network Functions Virtualization*) o SDN (*Software Defined Network*), y en este último se va a centrar el apartado actual. NFV modifica el diseño y dimensionado de los operadores de red, ya que permite centralizar servicios de red en uno o varios puntos, facilitando el control y gestión de los mismos. Por otro lado, con SDN puede automatizarse la gestión y provisión de la red de forma virtualizada, creando una red inteligente más flexible, escalable y programable [8].

1.2.2.1. El futuro de las redes.

Las redes jerárquicas diseñadas para escenarios cliente-servidor convencionales no se adapta bien al gran aumento de dispositivos móviles, contenido virtual, o la virtualización de servidores y servicios en la nube de la actualidad. La aparición de nuevos diseños es necesario por las siguientes tendencias [9]:

- **Patrón de tráfico:** el tráfico ha pasado a generarse mayormente entre bases de datos y servidores antes de enviar la información al cliente, en lugar de comunicarse constantemente con el cliente. Además los usuarios se conectan a cualquier hora del día desde cualquier lugar.
- **Aumento de consumo:** los usuarios hacen cada vez un uso más intenso de sus numerosos dispositivos. A todos estos tráficos hay que proveerlos de seguridad, disponibilidad y confiabilidad.
- ***Cloud Services*:** el uso de servicios de nubes privadas y públicas es cada vez más común, queriendo acceder a aplicaciones, infraestructuras y otros recursos bajo demanda. Además estos servicios deben estar siempre disponibles, accesibles de forma segura y soportar diversas demandas.
- ***Big Data*:** el manejo de este tipo de información requiere una cantidad de recursos y procesamiento muy grande, y los responsables de red deben encargarse de hacer que sus redes soporten ese tráfico.

Con el SDN, desarrollado por ONF [10] en 2011, se pretende cubrir todas estas necesidades y cambiar los modelos de arquitecturas de red. El protocolo OpenFlow se crearía en 2012 para utilizarlo en SDN, y en 2017 ONF también crea CORD, una solución para el nuevo movimiento de redes *edge computing*.

Según IDC (*International Data Corporation*), los ingresos de productos SDN y servicios asociados aumentaron de 360 a 3700 millones de dólares entre los años 2013 y 2016, y su uso ha crecido todavía más en estos últimos años [8].

1.2.2.2. Definición y características.

SDN es una arquitectura de red dinámica, eficiente, adaptable y controlable, muy adecuada para aplicaciones dinámicas con alto consumo de ancho de banda. Separa las funciones de reenvío y de control de red, permitiendo programar de manera concreta la gestión y hacer el resto de la estructura abstracta para las aplicaciones [10]. Así se pueden aplicar las políticas de red de forma automática y modificar de forma sencilla sin alterar las aplicaciones y funcionalidades de alto nivel. Utiliza el protocolo OpenFlow, que se explica en la sección 6.2.

Las principales características de SDN son [10, 11]:

- **Programable:** el control de red es programable al estar separado del resto de funciones. Los encargados de ello se deben encargar de configurar, controlar, optimizar y hacer seguro el funcionamiento.
- **Ágil:** permite administrar dinámicamente el tráfico a lo largo de la red y ajustar sus necesidades.
- **Centralizada:** la inteligencia de la red se encuentra centralizada en controladores SDN que tienen una visión global de la red que para los otros dispositivos parece otro *switch*.

- **Uso de nubes:** para flexibilizar el despliegue de recursos y servicios y reducir tiempos o posibles costes.
- **Distribución libre:** permite su uso libre y el de protocolos estandarizados para SDN, y con los controladores se puede manejar toda la red sin tener en cuenta incompatibilidades de marcas y protocolos.

1.2.2.3. Arquitectura.

La inteligencia de las redes SDN está centralizada en los controladores basados en *software*. Esto permite a los encargados de la red tener independencia y control sobre la infraestructura de red desde un punto lógico, simplificando el diseño. Gracias a la separación de la estructura en el controlador también se pueden implementar servicios de enrutamiento, *multicast*, control de acceso, seguridad o calidad de servicio. Además los dispositivos no tienen que responder a multitud de protocolos, solo seguir las órdenes del controlador [9].

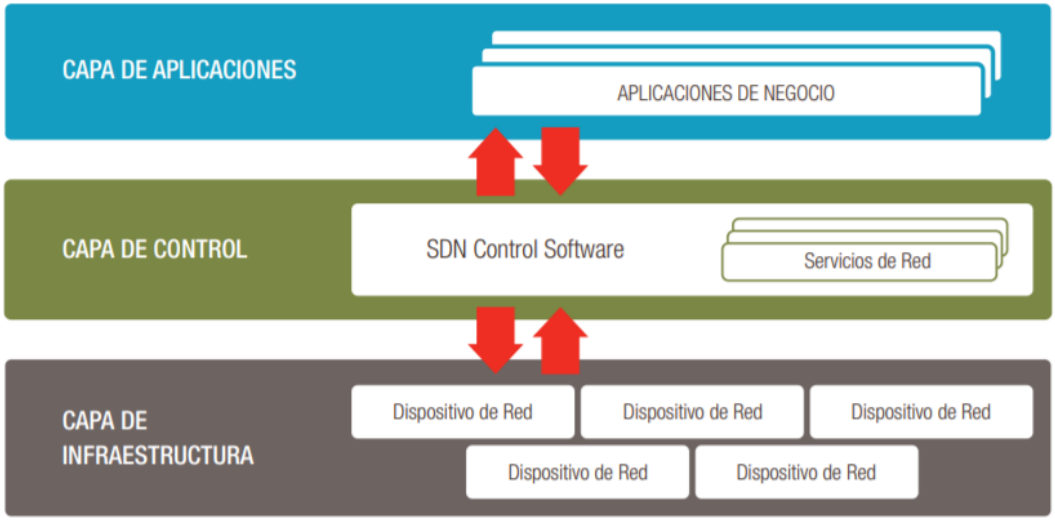


Figura 1.4: Arquitectura de SDN [9].

En la Figura 1.4, siguiendo las especificaciones de ONF, se muestra la arquitectura a alto nivel de SDN dividida en las siguientes capas [8]:

Capa de infraestructura.

La componen todos los nodos de red que realizan conmutación o encaminamiento de paquetes. Es la capa más baja de las tres y proporciona acceso programable a otros niveles a través de APIs (*Application Programming Interfaces*).

Capa de control.

Es *software* centralizado que permite a los desarrolladores modificar capacidades de la red abstrayéndose de la topología. El controlador SDN controla y configura todos los nodos de red para dirigir los flujos de tráfico por el mejor camino, encargándose de las decisiones de encaminamiento y enrutamiento. La arquitectura es independiente de los protocolos utilizados en las interfaces y puede controlar todos los recursos del plano de datos y simplificar su configuración.

Capa de aplicación.

Se compone de las aplicaciones de negocio de los usuarios, que utilizan servicios de la capa de control de SDN a través de APIs. Así los servicios y aplicaciones pueden automatizar y simplificar configuraciones y gestiones de los servicios de red.

Normalmente con SDN se utiliza el ya mencionado OpenFlow, un protocolo de bajo nivel usado para implementar control en los nodos. Al ser de código abierto aparecen numerosas opciones que implementen un controlador SDN pero en este proyecto se usará OpenDaylight, explicado en la sección 5.2.

1.3. Estructura de la memoria.

En esta sección se explica brevemente qué aborda cada uno de los capítulos que constituyen la memoria:

- **Capítulo 1. Introducción:** Presenta el proyecto y su contexto.
- **Capítulo 2. Objetivos:** Reúne las distintas metas propuestas para el proyecto.
- **Capítulo 3. Planificación y costes:** Organiza la distribución temporal en función de los plazos y las tareas a realizar, y analiza los recursos materiales.
- **Capítulo 4. Estado del arte:** Recoge trabajos publicados similares con el fin de situar el proyecto y ofrecer una base inicial.
- **Capítulo 5. Herramientas:** Explica las herramientas que se usan durante el desarrollo del proyecto.
- **Capítulo 6. Desarrollo teórico:** Explica los protocolos involucrados en el desarrollo del proyecto.
- **Capítulo 7. Desarrollo práctico:** Explica paso a paso el procedimiento realizado durante el proyecto.
- **Capítulo 8. Resultados:** Contrasta los resultados obtenidos en el desarrollo con los previstos.
- **Capítulo 9. Conclusiones:** Valora el trabajo realizado comparando los resultados con los objetivos iniciales.
- **Bibliografía:** Incluye únicamente las referencias usadas en la elaboración de la memoria ya que para el desarrollo y preparación del proyecto se consultan muchas otras fuentes.
- **Anexo A. Código:** Contiene las distintas versiones del código del controlador.

2

Objetivos

El principal objetivo del proyecto es lograr que una red MQTT prescinda del bróker como máquina independiente, usando un controlador SDN que lo sustituya y que permita modificar su comportamiento como se necesite.

Para lograr dicho objetivo, se divide en objetivos menores, con el fin de obtener tareas más concretas y evaluables. En primer lugar, se va a realizar un estudio del protocolo MQTT para comprender el funcionamiento de este tipo de redes y familiarizarse con los paquetes. A continuación, como paso intermedio, se crea un escenario MQTT en una red SDN manteniendo el bróker y, finalmente, se modifica el código para que sea el controlador el que actúa como bróker. En las siguientes secciones se explican estos tres subobjetivos con mayor detalle.

Esta misma clasificación será la empleada en el desarrollo práctico del capítulo 7.

2.1. Estudio del protocolo MQTT.

El primer objetivo es el más teórico de los tres. Pretende estudiar de forma teórica el protocolo MQTT, comprendiéndolo y analizándolo para montar una red sencilla MQTT que permita evaluar el funcionamiento estándar del protocolo y familiarizarse con la transmisión de paquetes en la red.

Es fundamental realizar esta fase previa porque se pretende el escenario final del proyecto funcione de manera idéntica a este primer escenario y para ello, hay que conocer de antemano cómo se transmiten los mensajes y qué estructura tienen.

Esta parte se evaluará probando el envío y recepción de mensajes en varios clientes y para varias QoS.

2.2. Desarrollo de la red SDN.

El segundo objetivo pretende crear un escenario que mantenga la red como en el objetivo 2.1 y que sirva como paso previo para 2.3. Para ello, se creará una red con OpenDaylight como controlador.

Al igual que para el primer objetivo, esta parte se evaluará probando el envío y recepción de mensajes en clientes MQTT hacia el mismo bróker. La única diferencia respecto al escenario anterior es el uso de un controlador externo gestionable en la red. Por lo tanto, los resultados obtenidos deberían ser idénticos en ambos escenarios, ya que se trata de escenarios MQTT con bróker.

2.3. Eliminación del bróker MQTT.

Este tercer y último objetivo pretende crear un escenario que cumpla con el objetivo final del proyecto. Se basa en los resultados obtenidos previamente y modifica el código del controlador para que actúe como bróker. De esta manera, se suprime la máquina independiente del bróker y se obtiene un entorno MQTT más rápido y sin intermediarios.

La modificación del código del controlador es compleja, por lo que se divide en las siguientes tareas:

- Creación de tablas de flujos de los *switches*.
- 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*.

De nuevo, la manera de evaluar los resultados y el cumplimiento del objetivo deseado será probar el envío y recepción de mensajes MQTT de forma correcta. Los resultados deben coincidir con los escenarios previos, salvo modificaciones relacionadas con que el controlador reemplazará las funciones del bróker.

3

Planificación y costes

En este capítulo se organiza el proyecto para garantizar su coherencia respecto a los objetivos.

En la sección 3.1 se estructuran las tareas a realizar. Se incluyen tanto las relacionadas con la formación, documentación, análisis, etc. como las específicas del proyecto.

En la sección 3.2 se presenta la duración y orden de ejecución de las tareas de la sección anterior. Se incluye tanto la planificación prevista como la seguida, ya que la primera es la utilizada durante la realización del proyecto a modo de guía mientras que la segunda permite estimar el tiempo real empleado para cada actividad.

La última sección recoge la estimación del coste asociado, considerando tanto al personal como los recursos de *hardware* y *software* implicados.

3.1. Definición de tareas.

Las fases mostradas a continuación pretenden mostrar cronológicamente el trabajo seguido y se ha buscado que la memoria siga un orden similar para facilitar su comprensión y recreación, si fuese necesario.

Fase 1 - Preparación.

La primera fase recoge todas las tareas previas al desarrollo del proyecto. Es necesario un tiempo de preparación en el que documentarse, investigar y aprender a utilizar las herramientas. Además, hay que analizar los objetivos previstos, decidir cómo realizarlos y estimar la duración y los recursos de cada tarea.

Podría decirse que esta fase trabaja los primeros seis capítulos de la memoria: introducción, objetivos, planificación, estado del arte, herramientas y fundamentos teóricos.

Fase 2 - Desarrollo del proyecto.

Esta fase corresponde al desarrollo del proyecto en sí mismo y debe solucionar todos los objetivos del capítulo 2. De hecho, dada la envergadura del proyecto, se divide en tareas más pequeñas, correspondientes a capa objetivo y a las secciones del capítulo 7.

- Fase 2.1 - Estudio MQTT.
- Fase 2.2 - Desarrollo de la red SDN.
- Fase 2.3 - Eliminación del bróker.

Fase 3 - Análisis de resultados.

Una vez finalizado el desarrollo práctico se procede a comparar los resultados obtenidos con los previstos, es decir, con los de ese mismo escenario usando un bróker.

Además se verificará el comportamiento en distintos casos para asegurarse de si se cumplen los objetivos marcados.

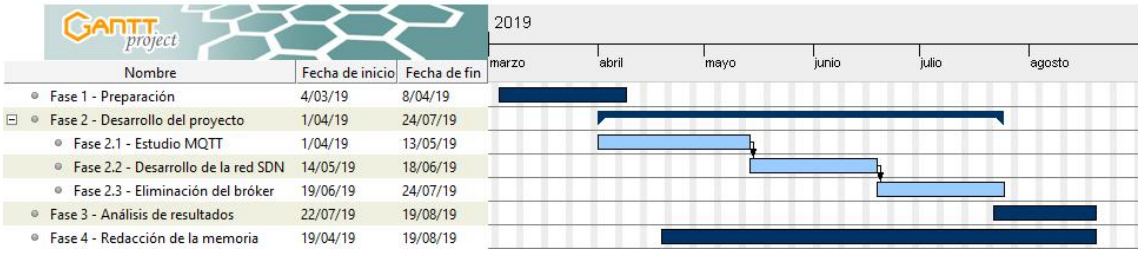
Estos resultados se encuentran en el capítulo 8.

Fase 4 - Redacción de la memoria.

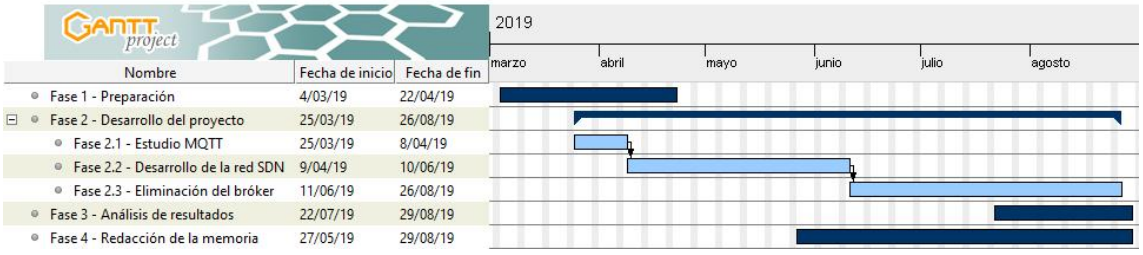
Para cerrar el proyecto se debe redactar una memoria que recoja todo lo investigado, realizado y testado en el proyecto, de manera comprensible para cualquier lector pero con los suficientes detalles técnicos para reproducir el proyecto.

3.2. Planificación temporal.

Partiendo de las fases presentadas en la sección anterior, se establece la organización temporal de las tareas. La herramienta usada para este tipo de planificación es el diagrama de Gantt, que permite enumerar las tareas otorgándoles un plazo y una prioridad.



(a) Tareas y plazos previstos.



(b) Tareas y plazos reales.

Figura 3.1: Diagramas de Gantt del proyecto.

Al comenzar el proyecto, se diseña un diagrama de Gantt con la planificación prevista según el tiempo total del que se dispone. Este diagrama permite fijar *milestones*, como al final de cada fase en este caso, que ayudan a cumplir unos plazos y evaluar progresivamente el trabajo. Además, se debe disponer de un margen de tiempo al final para compensar posibles retrasos e imprevistos. La Figura 3.1a muestra la distribución inicial de las tareas, asignando aproximadamente unas 5 semanas para cada una de las fases, a excepción de la memoria que puede irse desarrollando en paralelo al proyecto.

Al finalizar el proyecto se diseña el diagrama de Gantt real de la Figura 3.1b. Las tareas más complicadas de llevar a cabo corresponden con las secciones 7.2 y 7.3, lo cual se refleja en el tiempo destinado a esas fases. La Fase 1 también se alarga considerablemente porque para comprender y realizar las Fases 2.1 y 2.2, seguía siendo necesario investigar y estudiar el funcionamiento del protocolo MQTT.

El contraste entre los dos diagramas ayuda a identificar las partes más complicadas del trabajo y sirve de guía a futuros proyectos que necesiten recrearlo.

3.3. Recursos y coste asociado.

Los recursos necesarios en la elaboración de un proyecto son un límite que hay que considerar desde el principio. En concreto, hay que considerar los gastos en recursos humanos, en *software* y en *hardware*.

Los recursos humanos se refieren al trabajo realizado por cualquier persona involucrada en el proyecto, que en este caso es el alumno y el tutor. Se estima que el sueldo de un alumno recién graduado es de 20 €/h aproximadamente y que se han dedicado 25 semanas de trabajo con una media de 20 horas semanales, lo que resulta en un coste de 10.000 €. En cuanto al tutor, se estima un sueldo de 40 €/h y se considera un total de 8 tutorías de una hora de duración, además de unas 10 horas invertidas en la corrección de la memoria. Como resultado, se obtiene un total de 720 €.

Como herramienta *hardware* es suficiente con un ordenador. Para este proyecto se usa un portátil Lenovo Ideapad 520 valorado en 900 €, capaz de trabajar con las máquinas virtuales necesarias. Como el ordenador tiene una vida útil de unos 50 meses y se ha usado durante 6, el coste aproximado obtenido es de 100 €. Precisamente uno de las principales ventajas de las redes SDN es que no requieren de elementos extra que hubiesen encarecido considerablemente el proyecto.

En cuanto al *software*, los principales han sido Virtualbox, OpenDaylight, Mininet y Wireshark. Todos estos programas y sus máquinas virtuales son de uso libre y gratuito, por lo que el coste es nulo.

Campos		**Coste**	**Total**
Recursos humanos	Alumno	10.000 €	10.720 €
	Tutor	720 €	
Hardware	Portátil	100 €	100 €
Software	Programas	0 €	0 €
	Máquinas virtuales	0 €	
			10.820 €

Tabla 3.1: Coste total del proyecto.

4

Estado del arte

En este capítulo se habla sobre algunos de los trabajos académicos o investigaciones con temáticas relacionadas a las de este proyecto. Se habla por ejemplo de variaciones del protocolo MQTT para utilizar *multicast* en SDNs, opciones para administrar redes de IoT, *middlewares* para envío de notificaciones en IoT con MQTT o que controlen la QoS. El conocer estos contenidos puede resultar muy útil a la hora de ver cómo abordar un proyecto de esta temática y coger algunas ideas.

4.1. DM-MQTT.

El protocolo MQTT utiliza un bróker para distribuir los mensajes de los dispositivos IoT en una red, y este tipo de metodología donde un solo dispositivo se encarga de controlar la red se conoce como *cloud computing*. El estándar de MQTT puede ser muy eficiente para sistemas de pequeño tamaño pero sufre con grandes cantidades de flujos de datos. Para solucionar este problema típico en *cloud computing,* se sustituye el paradigma por *edge computing*, donde los datos producidos por los dispositivos IoT se procesan cerca de su punto de creación en lugar de llevarlo a un punto de procesamiento centralizado.

En el artículo [13] llamado *«DM-MQTT: An Efficient MQTT Based on SDN Multicast for Massive IoT Communications»* se propone el uso de *edge computing* para el protocolo MQTT en redes de gran tamaño, haciendo uso de SDN y de mecanismos *multicast* que minimicen la trasferencia de datos y reduzcan la congestión. Esta variación se llama DM-MQTT (*Direct Multicast MQTT*). En la Figura 4.1 se muestra una arquitectura de red haciendo uso de DM-MQTT, que se compone de un controlador SDN y un bróker principal (o *master*) de MQTT, un nodo de borde con un bróker secundario (o *slave*) y dispositivos IoT.

Con esta modificación habría distintos brókers *slaves* en los bordes de la red que se comunicarían con los dispositivos IoT, y para publicar en los suscriptores el bróker establece un *multicast* SDN bidireccional entre los publicadores y sus suscriptores. La información de red como direcciones IP, QoS o *topics* para crear los grupos de *multicast* llega al controlador SDN mediante el *master* bróker a través de sus *slaves*. Con este complejo método se puede reducir el retardo de transmisión en un 65 % y el uso de red en un 58 % en comparación al MQTT estándar. En la Figura 4.2 se puede apreciar una comparativa de retardo al transmitir datos en función del número de sensores en la red.

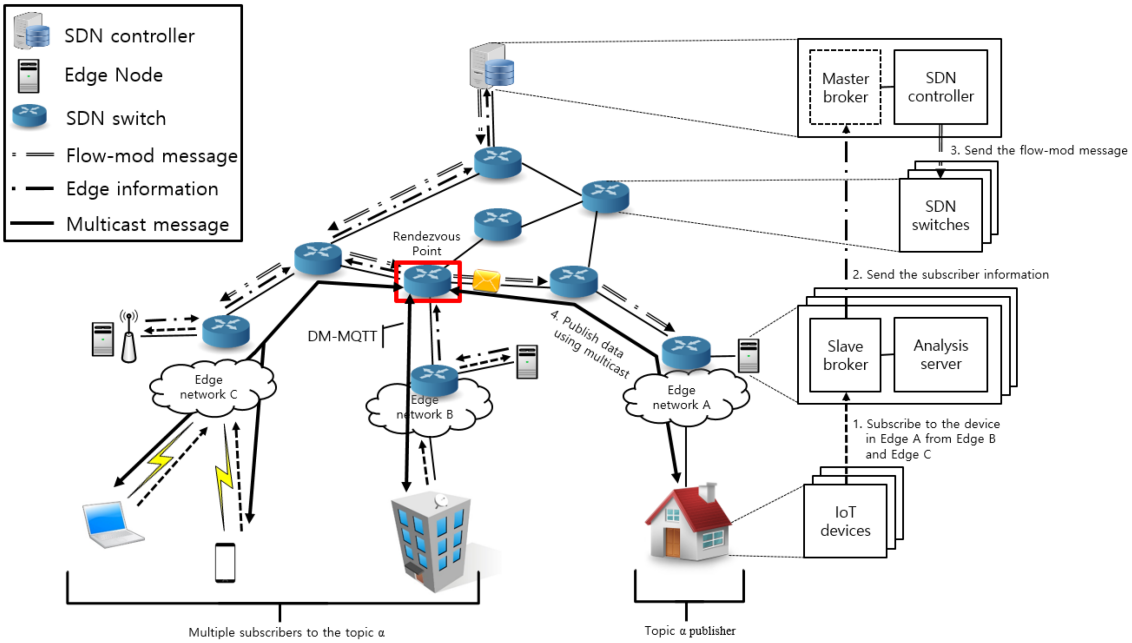


Figura 4.1: Arquitectura DM-MQTT [13].

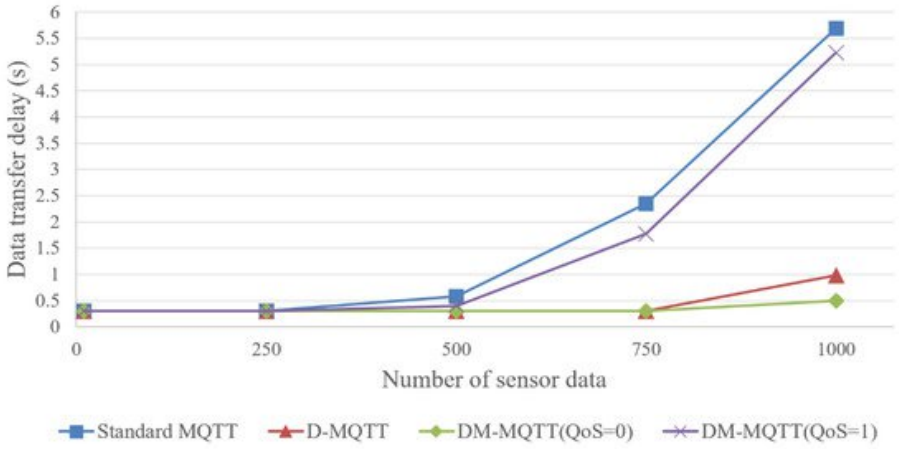


Figura 4.2: Comparativa retardo/número de sensores [13].

4.2. SDN-Based Management Framework for IoT.

La aparición de Internet ha sido una gran revolución en nuestra vida diaria y el desarrollo de IoT puede ser el siguiente paso. El control y administración de redes cuando se usan numerosos de estos dispositivos IoT puede convertirse en un problema importante. Por ello en el artículo [14] se propone una forma de hacerlo con SDN y el protocolo MQTT.

En la Figura 4.3 se muestra un escenario típico con una red SDN, bases de datos y componentes de MQTT. Los dispositivos IoT envían información de dos tipos a las bases de datos. La primera es información del dispositivo (como la ID del dispositivo) que se almacena en una base de datos y la otra son los datos recogidos que se envían a otra base de datos distinta. La información del equipo se publica en el bróker utilizando MQTT y los dispositivos IoT se suscriben a sus propias IDs para recibir órdenes de usuarios. A través

de una interfaz de navegador utilizando la base de datos, los usuarios pueden suscribirse a *topics* para obtener la información obtenida por los dispositivos y enviarles peticiones.

Las funciones principales de SDN en este proyecto son la configuración de rutas, monitorización del tráfico y el control de errores de conmutación. Por tanto cuando se quiere añadir un equipo IoT a la red es SDN el encargado de controlar los envíos hasta el bróker MQTT, o cuando se produce un error de rutas SDN activa el protocolo STP (*Spanning Tree Protocol*) y busca una nueva ruta de comunicación.

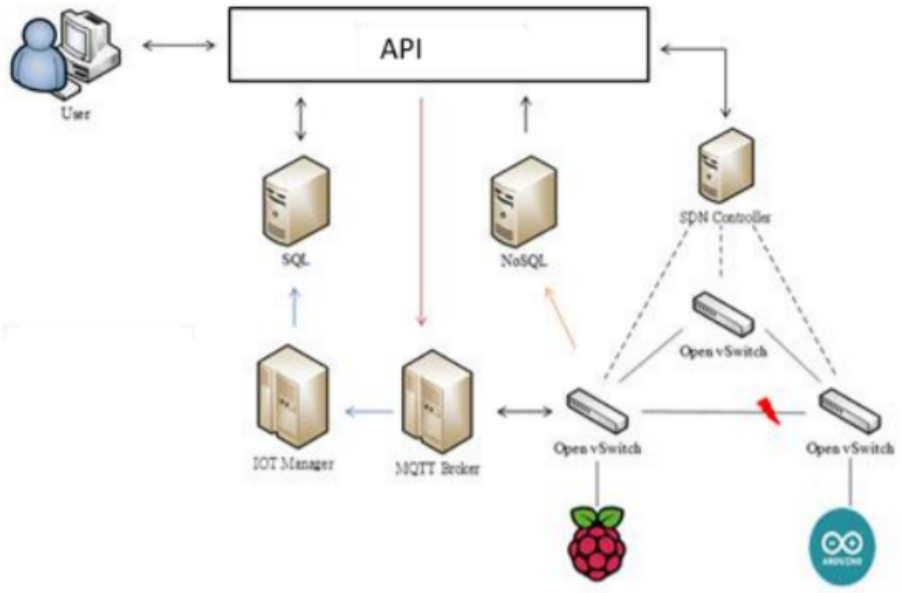


Figura 4.3: Escenario típico de implementación [14].

4.3. SDN-enabled IoT Data Exchange Middleware.

En el artículo [15] llamado *«An Implementation Experience with SDN-enabled IoT Data Exchange Middleware»* se explica la implementeción de FireDeX [16], un *middleware* multicapa para envío de mensajes y notificaciones en un servicio de intercambio de datos IoT.

FireDeX utiliza el paradigma publicador-suscriptor con brókers en los extremos de la red para controlar la entrega de eventos críticos sobre datos IoT a los suscriptores más importantes. Toma parámetros de las capas de aplicación, intercambio y red para estimar valores de rendimiento y diseña un modelo de colas para las interacciones multicapa. También utiliza un algoritmo que, dependiendo de los datos anteriores, modifica las configuraciones de envío para priorizar los datos más importantes en caso de congestión, lo que puede mejorar el rendimiento de intercambio de información en un 36 %, o políticas de descarte de paquetes para mejorar un 42 %. A su vez FireDeX se aprovecha de metodologías SDN para mejorar la arquitectura de red de IoT.

La Figura 4.4 muestra la arquitectura de las distintas capas de FireDeX. En la capa de aplicación los publicadores se conectan a el intercambiador de datos del bróker con MQTT. Por otro lado, los suscriptores se conectan al mismo elemento (con funciones que asignan un valor de importancia) para recibir eventos, y con el coordinador de servicios FireDeX para que le asigne conexiones.

En la capa de intercambio de datos el bróker se encarga de las conexiones publicador-suscriptor, y MQTT-SN *gateway* traduce el protocolo MQTT para que trabaje sobre UDP. El coordinador de servicios FireDeX es el centro de todo y se encarga de asignar prioridades, conexiones o usar algoritmos de descarte de mensajes.

Por último la capa de red obtiene la información del coordinador FireDeX a través del controlador SDN y aquí se aplican todas las decisiones del controlador a las notificaciones de red que van llegando. En esta capa también se obtienen los parámetros de tráfico usados en los algoritmos.

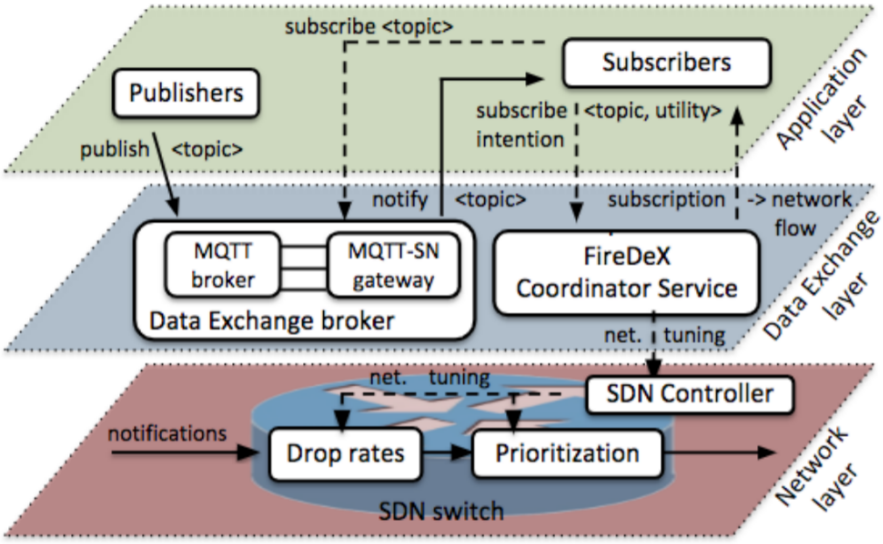


Figura 4.4: *Middleware* multicapa de FireDeX [15].

4.4. EMMA.

Muchos escenarios de IoT tienen requerimientos de QoS que no se pueden cumplir con mecanismos de *cloud computing*. Para solucionar este problema aparece EMMA, explicado en el artículo [17] llamado *«EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications»*, un *middleware* basado en el paradigma suscriptor-publicador que utiliza *edge computing*.

La red cuenta con distintos brókers MQTT que monitorizan la QoS en todo momento para optimizarla según el estado de la red. Además el sistema ofrece detección de proximidad basado en latencia y distribución orquestada de los *gateways* y brókers de la red entre los clientes.

La arquitectura de EMMA se muestra en la Figura 4.5 y se divide en cuatro componentes principales:

- *Gateway*: permite la movilidad de los clientes reconfigurando las conexiones. Usa tunelado por el que enrutar el tráfico MQTT.
- Brókers: actúan como servidores MQTT y controlan los *topics* de los clientes. Incluyen *bridging tables* sincronizadas por el controlador para tener un listado de los clientes de cada *topic*.
- Controlador: es el encargado de sincronizar los componentes del sistema. Se comunica con los *gateways* y los brókers, lo configura y realiza balanceo de carga.

- *Monitoring*: los componentes de EMMA incluyen un protocolo ligero de monitorización para tener información de ellos.

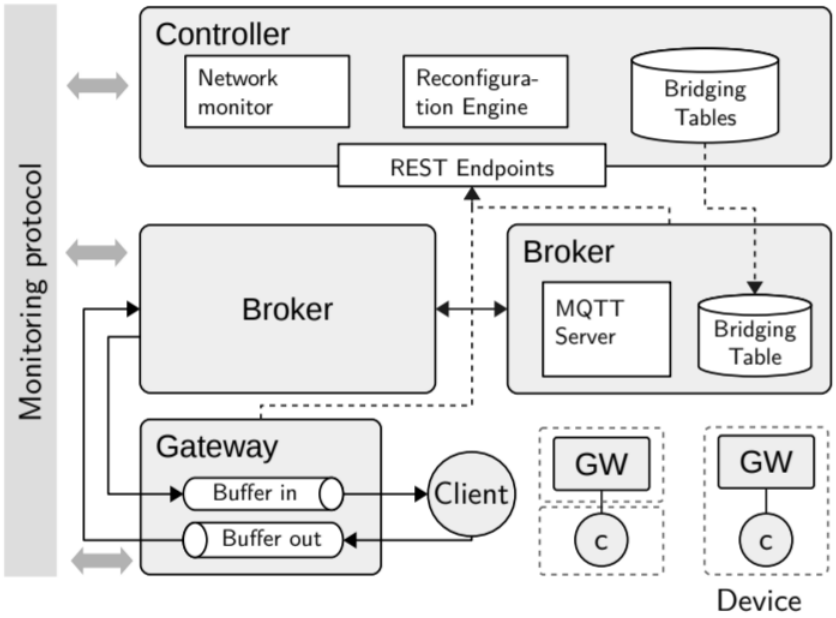


Figura 4.5: Arquitectura de EMMA [17].

Para implementar la monitorización de la QoS se añade un protocolo de este tipo sobre UDP. El controlador envía un QOSREQ al *gateway*, que reenvía 10 PINGREQ al bróker. El bróker responde con 10 PINGRESP y el *gateway* guarda las diferencias temporales entre paquetes y envía al controlador un QOSRESP con la latencia media. El controlador usará este dato para calcular posibles pérdidas de paquetes o *jitter* y asignar una QoS.

Para que los clientes estén repartidos y organizados mandan sus peticiones MQTT *connect* hacia el *gateway* en lugar de hacia el bróker. Así el controlador puede decidir si conectar un cliente a otro bróker distinto que esté menos congestionado. Los clientes que ya estén conectados a un bróker se comprueban periódicamente para obtener sus valores de latencia, y en caso de que el algoritmo encuentre una QoS mejor el cliente sería cambiado de bróker.

5

Herramientas

En este capítulo se recogen las herramientas que permiten desarrollar el trabajo. Se comenzará hablando de Mininet y su uso en creación de redes, luego se analizará OpenDaylight y sus funciones para crear controladores de red y por último se hablará brevemente de Wireshark.

5.1. Mininet.

Mininet es un *software* emulador para hacer prototipos de grandes redes en un solo equipo. Permite crear de forma sencilla y realista redes virtuales personalizadas con las que interactuar o compartir, haciendo prototipos de SDNs que simulen una topología de red con OpenFlow *switches*.

Fue creado en 2009 por Bob Lantz y Brandon Heller basándose en un prototipo demostrado por Bob Lantz. Actualmente el proyecto de Mininet sigue en desarrollo por lo que periódicamente se publican parches y nuevas versiones con mejoras y actualizaciones [18].

Para realizar la instalación de Mininet de forma sencilla se puede seguir un tutorial como el de Brian Linkletter [19] o el *Walkthrough* de la página oficial [18].

5.1.1. Características.

En este apartado se listan las principales características de Mininet [20]:

- Es un proyecto ***Open Source*** por lo que se puede examinar el código, usar y modificar de forma gratuita.
- **Sencillo**, ya que se puede instalar y empezar a trabajar con una topología de red en poco tiempo y de forma intuitiva.
- ***Software* ligero** que puede correr en un servidor, máquina virtual o en la nube, sin requerir muchos recursos.
- Permite crear escenarios con **grandes redes virtuales** en los que se pueden controlar todos los dispositivos que la forman.
- Los *switches* de Mininet hacen posible **personalizar el flujo** de paquetes usando **OpenFlow**.
- Las **topologías** se pueden **personalizar** para crear escenarios complejos, o usar las básicas que se incluyen por defecto.

- Los **elementos** como *routers, switches* o *hosts* se comportan igual que los **reales** y se pueden controlar de forma independiente.
- En los elementos se pueden ejecutar **programas externos** instalados en el sistema, lo que facilita la creación de redes complejas y su monitorización. Todos comparten los datos del sistema principal, lo que en algunos casos puede resultar perjudicial.
- En los *hosts* solo puede usarse **Linux** kernel.
- **No incluye** un **controlador** OpenFlow. Debe ser externo.
- La **red** de Mininet estará **aislada** de la LAN personal y de internet.

Tras saber las características principales de Mininet, se puede llegar a la conclusión de que cumple gran parte de los requerimientos para este proyecto y, por tanto, será una de las herramientas elegidas.

5.1.2. Topologías.

Mininet ofrece varias opciones para crear topologías de red que se puede dividir en básicas y personalizadas.

El comando principal para llamar al programa es `«mn»` y para indicar la topología se utiliza `«--topo=NOMBRE, X»`, donde `NOMBRE` es el tipo que se quiere crear y `X` la variable, en caso de necesitarla [18]. También se ofrecen otras opciones en la creación de topologías como se muestra en la Tabla 5.1.

Comando	Funcionalidad
`--controller = [default, remote, nox...]`	Indica el controlador a usar en la red.
`--switch = [default, ovs, user...]`	Crea un *switch* con las características indicadas.
`--mac`	Asigna dirección MAC a los *hosts*.
`--test = [pingall, cli,iperf...]`	Realiza una prueba.
`--custom`	Carga una topología desde un *script* python.
`--h help`	Muestra la ayuda.

Tabla 5.1: Opciones al crear topologías.

5.1.2.1. Básicas.

Las topologías básicas son las que ofrece Mininet por defecto a partir de un simple comando. Estas facilitan el trabajo a los usuarios ya que no tienen que hacer ningún tipo de programación para tener la red en funcionamiento [21].

En este proyecto, se usarán algunas de ellas para la primera fase, ya que se busca la familiarización con las herramientas y la comprensión del funcionamiento de la red y para ello, lo ideal es comenzar por un escenario sencillo. Sin embargo, para el desarrollo práctico se usará una red personalizada, tal y como se explica en la sección 5.1.2.2.

Mínimal.

Topología por defecto que aparece cuando no se indica ninguna. Es muy simple porque consta tan solo de 4 entidades: 1 controlador, 1 *switch* y 2 *hosts* conectados al *switch*.

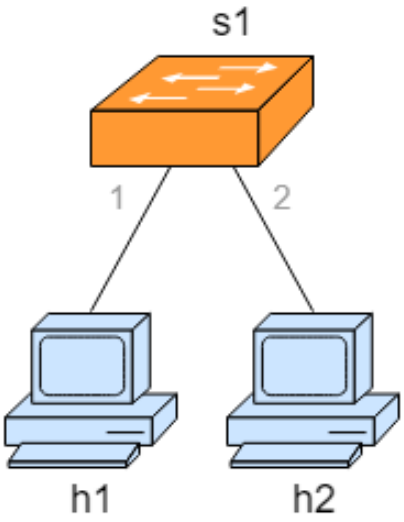


Figura 5.1: Esquema de la topología *minimal*.

La Figura 5.2 muestra la creación de la topología *minimal* en Mininet.

```
mininet@mininet-vm:~$ 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
```

Figura 5.2: Creación en Mininet de la topología *minimal*.

Single.

Incluye 1 controlador, 1 *switch* y «X» *hosts* conectados al *switch*, según el valor de la variable.

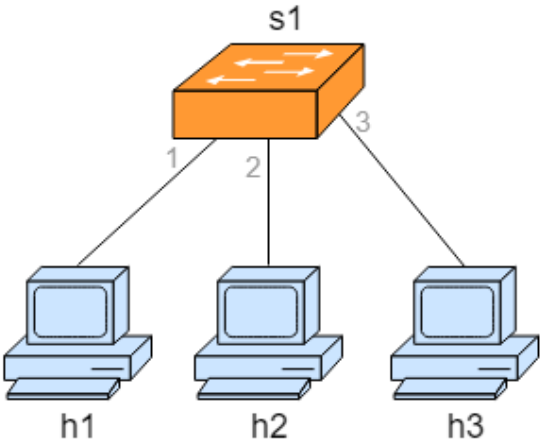


Figura 5.3: Esquema de una topología *single*.

Para el ejemplo de las Figuras 5.3 y 5.4 se considera `X=3`.

```
mininet@mininet-vm:~$ sudo mn --topo=single,3
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** 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
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0
c0
```

Figura 5.4: Creación en Mininet de una topología *single*.

Linear.

Incluye 1 controlador, «X» *switches* según el valor de la variable y 1 *host* conectado a cada *switch*.

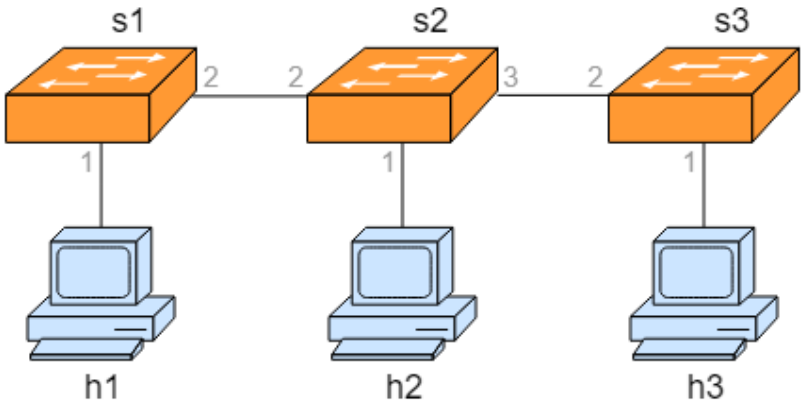


Figura 5.5: Esquema de una topología *linear*.

Para el ejemplo de las Figuras 5.5 y 5.6 se considera `X=3`.

```
mininet@mininet-vm:~$ sudo mn --topo=linear,3
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-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
c0
```

Figura 5.6: Creación en Mininet de una topología *linear*.

Tree.

Crea una topología tipo árbol. Utiliza 2 variables, «N» para indicar el número de niveles del árbol y «M» para indicar el número de *hosts* por cada *switch*.

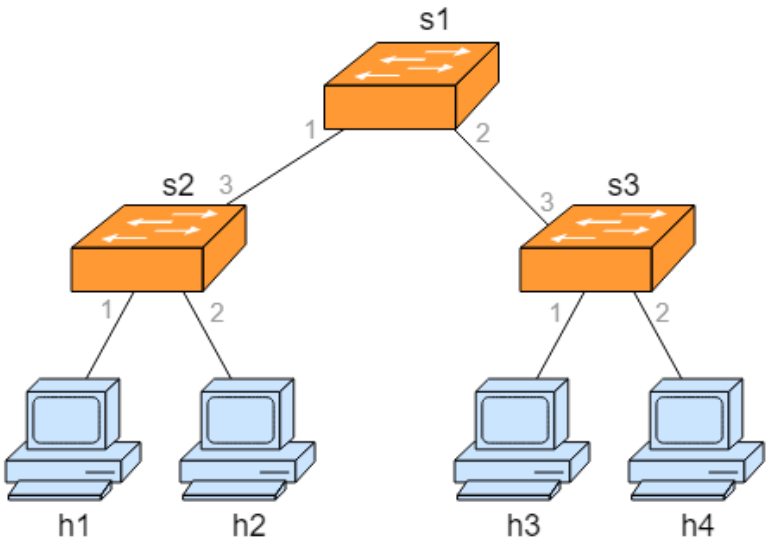


Figura 5.7: Esquema de una topología *tree*.

Para el ejemplo de las Figuras 5.7 y 5.8 se considera `N=2` y `M=2`.

```
mininet@mininet-vm:~$ sudo mn --topo=tree,2,2
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s2-eth1
h2 h2-eth0:s2-eth2
h3 h3-eth0:s3-eth1
h4 h4-eth0:s3-eth2
s1 lo:  s1-eth1:s2-eth3 s1-eth2:s3-eth3
s2 lo:  s2-eth1:h1-eth0 s2-eth2:h2-eth0 s2-eth3:s1-eth1
s3 lo:  s3-eth1:h3-eth0 s3-eth2:h4-eth0 s3-eth3:s1-eth2
c0
```

Figura 5.8: Creación en Mininet de una topología *tree*.

5.1.2.2. Personalizadas.

Mininet permite la creación de topologías personalizadas (o *custom*) con unas líneas de código. Para ello se crea un *script* de Python en el que programar la red con todos los detalles necesarios, al que se pueden importar paquetes con funciones creadas para facilitar el trabajo. Las principales funciones y métodos se muestran en la Tabla 5.2.

```
mininet@mininet-vm:~$ sudo mn --custom mytopo.py --topo mytopo
```

Figura 5.9: Comando topología *custom*.

Comando	Funcionalidad
`build() / __init__()`	Método para inicializar la topología.
`addSwitch()`	Añade un *switch*. Se pueden definir ciertas opciones de configuración.
`addHost()`	Añade un *host*. Permite configurar opciones como IP o MAC.
`addController()`	Añade un controlador. Puede configurarse el tipo, protocolos, etc.
`addLink()`	Añade un enlace entre dos nodos.
`start()`	Inicia la red.
`stop()`	Finaliza la red.
`pingAll()`	Hace *ping* a todos los dispositivos.
`net.hosts`	Se refiere a todos los *hosts* de la red.
`dumpNodeConnections()`	Deshabilita la conexión de/a un nodo de la red.

Tabla 5.2: Funciones y métodos en topologías *custom* [20].

En el siguiente cuadro de código se ve un ejemplo de *script* de topología personalizada donde se crean 2 *switches* con 2 *hosts* conectados a cada uno, y en la 5.10 el resultado tras ejecutarlo.

```
1  """Custom topology example"""
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         leftHost = self.addSwitch( 'h1' )
13         rightHost = self.addSwitch( 'h2' )
14         leftSwitch = self.addSwitch( 's3' )
15         rightSwitch = self.addSwitch( 's4' )
16
17         # Add links
18         self.addLink( leftHost, leftSwitch )
19         self.addLink( leftSwitch, rightSwitch )
20         self.addLink( rightSwitch, rightHost )
21
22 topos = { 'mytopo': ( lambda: MyTopo() ) }
```

```
ubuntu@sdnhubvm:~/mininet/custom[13:10] (master)$ sudo mn --custom topo-2sw-2host.py --topo mytopo
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s3 s4
*** Adding links:
(h1, s3) (s3, s4) (s4, h2)
*** Configuring hosts
h1 h2
*** Starting controller

*** Starting 2 switches
s3 s4 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s3-eth1
h2 h2-eth0:s4-eth2
s3 lo:  s3-eth1:h1-eth0 s3-eth2:s4-eth1
s4 lo:  s4-eth1:s3-eth2 s4-eth2:h2-eth0
```

Figura 5.10: Creación en Mininet de una topología *custom*.

El resultado obtenido es, por lo tanto, el representado en el esquema de la Figura 5.11.

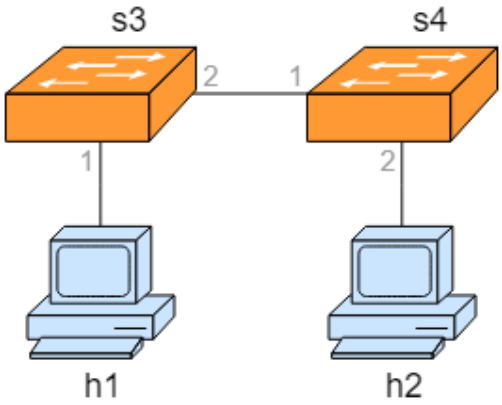


Figura 5.11: Esquema de una topología *custom*.

5.1.3. Comandos.

Una vez se crea una topología se pueden realizar numerosas acciones a través de comandos [21]. Los principales se muestran en la Tabla 5.3.

Comando	Funcionalidad
`exit / quit`	Cierra el sistema.
`xterm`	Abre un terminal del nodo señalado.
`link [node1][node2][up / down]`	Crea / elimina una interfaz entre nodos.
`[node] ping [node]`	Realiza un *ping* entre los nodos especificados.
`pingall`	Hace 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.
`help`	Muestra la ayuda de Mininet.

Tabla 5.3: Comandos de Mininet.

```
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
```

Figura 5.12: Comando `help`.

5.1.4. Uso en el proyecto.

En este proyecto Mininet se utilizará como herramienta principal para crear redes virtuales y topologías específicas. Sobre los elementos de Mininet se construirán escenarios MQTT para ver el funcionamiento del protocolo y posteriormente se utilizará la red completa para añadir un controlador SDN externo y modificar el comportamiento de la red. Por tanto, Mininet se puede considerar la base del proyecto.

5.2. OpenDayLight.

OpenDaylight es un controlador SDN que permite a los ingenieros de una red dirigir de forma programada los servicios de red mediante una API (*Application Programming Interface*) [22]. El proyecto de OpenDaylight proporciona una plataforma de código abierto que utiliza protocolos libres para dar control de red de forma sencilla y centralizada [23].

Algunas de las diferencias principales de este *software* con otros SDN son las siguientes [23]:

- Una arquitectura de microservicios, es decir, una serie de protocolos o servicios que se pueden activar en el controlador.
- Soporte para una gran número de protocolos como OpenFlow, PCEP, OVSDB o SNMP.
- Uso de MD-SAL (*Model Driven Service Abstraction Layer*) para crear esquemas de bases de datos y generar aplicaciones o código automáticamente.

OpenDaylight fue creado en 2013 por Linux Foundation como su primer proyecto de red para facilitar el acceso a las redes SDN y sigue trabajando en él con la colaboración de LF Networking. La primera distribución lanzada se llamó *Hydrogen*, y en 2019 se ha lanzado la décima versión llamada *Neon* [24]. En este proyecto se usará el software *Helium* ya que es muy estable y cuenta con mucha información y facilidades con la interfaz.

Para realizar fácilmente la instalación del programa se puede seguir el tutorial ofrecido por John Sobanski [22] con el apoyo de SDN Hub [25].

5.2.1. Componentes.

OpenDaylight utiliza una serie de programas y herramientas para mejorar y complementar su funcionamiento. Algunas de las principales se describen a continuación [25].

Maven.

Usado para compilar el proyecto de forma fácil y automática. Se basa en los archivos «pom.xml» y «feature.xml» para organizar la compilación, y cuando un proyecto es modificado para incluir nuevos módulos estos archivos deben actualizarse.

Para poner en funcionamiento Maven se usa el comando «`mvn`», seguido de «`install`» para compilar el código o de «`clean`» para limpiar los archivos temporales. En la Figura 5.13 se puede observar la instalación de uno paquetes tras su correcta compilación.

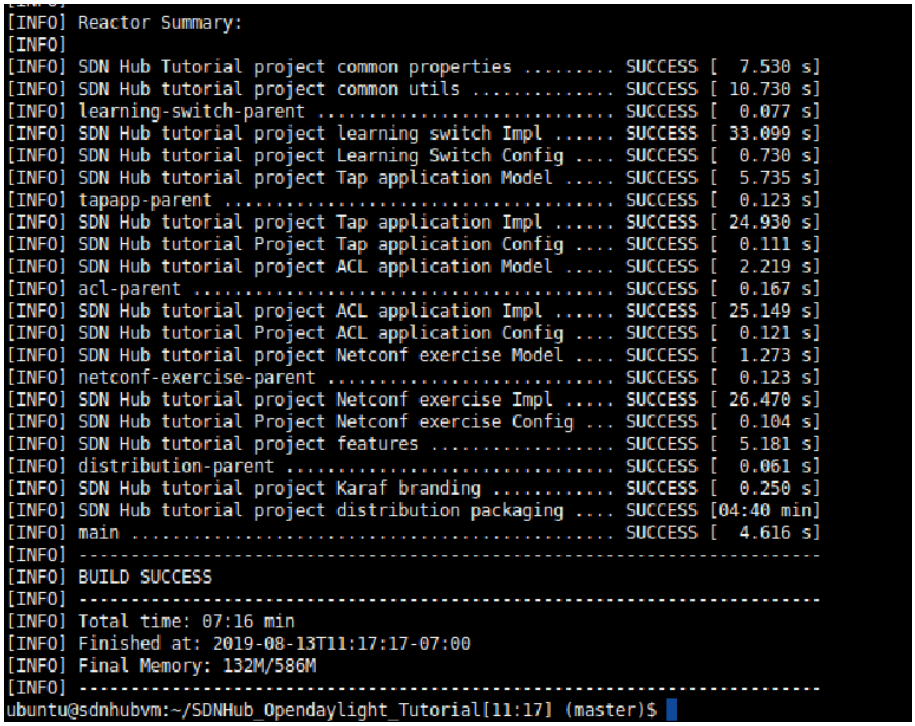


Figura 5.13: Paquetes compilados con Maven.

Maven funciona resolviendo dependencias entre paquetes, recorriendo directorios y haciendo descarga de archivos en caso de ser necesario. Por tanto, añadir las dependencias en los archivos «pom.xml» acorta notablemente el proceso de compilación. Uno de estos archivos es el principal, situado en «~/NOMBRE_DEL_PROYECTO/pom.xml», que sirve de base para el resto de estos módulos. El resto se encuentra en «common/parent/pom.xml».

Karaf.

Se trata del contenedor del controlador. Al arrancarlo se inician todos los paquetes instalados en el entorno, preparándose para cambiar de estado y entrar en funcionamiento. Se puede navegar dentro de la aplicación usando la tecla del tabulador para mostrar las opciones posibles en cada momento.

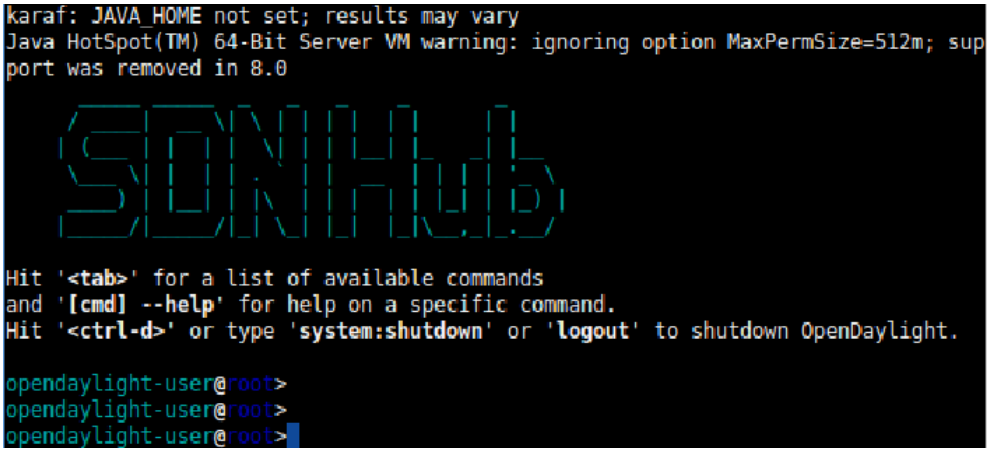


Figura 5.14: Karaf al ser inicializado.

Los comandos «`feature:list`» y «`bundle:list`» ayudan a encontrar o usar las características y paquetes de los que se dispone dentro del entorno. Cuando se muestra la lista, las funcionalidades activas aparecerán marcadas con una «X».

```
opendaylight-user@root>bundle:list
START LEVEL 100 , List Threshold: -1
 ID | State    | Lvl | Version                          | Name
----------------------------------------------------------------------------------------------------------------------------------------
  0 | Active   |   0 | 3.8.2.v20130124-134944           | OSGi System Bundle
  1 | Active   |   5 | 2.3.0                            | OPS4J Pax Url - aether:
  2 | Active   |   5 | 2.3.0                            | OPS4J Pax Url - wrap:
  3 | Active   |   8 | 1.8.1                            | OPS4J Pax Logging - API
  4 | Active   |   8 | 1.8.1                            | OPS4J Pax Logging - Service
  5 | Active   |  10 | 3.0.3                            | Apache Karaf :: Service :: Guard
  6 | Active   |  10 | 1.8.0                            | Apache Felix Configuration Admin Service
  7 | Active   |  11 | 3.4.2                            | Apache Felix File Install
  8 | Active   |  12 | 5.0.3                            | ASM all classes with debug info
  9 | Active   |  20 | 1.1.0                            | Apache Aries Util
 10 | Active   |  20 | 1.0.1                            | Apache Aries Proxy API
 11 | Active   |  20 | 1.0.4                            | Apache Aries Proxy Service
 12 | Active   |  20 | 1.0.1                            | Apache Aries Blueprint API
 13 | Active   |  20 | 1.0.5                            | Apache Aries Blueprint CM
 14 | Resolved |  20 | 1.0.0                            | Apache Aries Blueprint Core Compatiblity Fragment Bundle, Hosts: 15
 15 | Active   |  20 | 1.4.2                            | Apache Aries Blueprint Core, Fragments: 14
 16 | Active   |  24 | 3.0.3                            | Apache Karaf :: Deployer :: Spring
 17 | Active   |  24 | 3.0.3                            | Apache Karaf :: Deployer :: Blueprint
 18 | Active   |  24 | 3.0.3                            | Apache Karaf :: Deployer :: Wrap Non OSGi Jar
 19 | Active   |  25 | 3.0.3                            | Apache Karaf :: Region :: Core
 20 | Active   |  25 | 3.0.3                            | Apache Karaf :: Features :: Core
 21 | Active   |  26 | 3.0.3                            | Apache Karaf :: Deployer :: Features
 22 | Active   |  30 | 2.12.0                           | JLine
 23 | Active   |  30 | 0.2.1                            | JLEdit :: Core
 24 | Active   |  30 | 3.0.3                            | Apache Karaf :: Features :: Command
 25 | Active   |  30 | 3.0.3                            | Apache Karaf :: Shell :: Console
 26 | Active   |  30 | 3.0.3                            | Apache Karaf :: JAAS :: Modules
```

Figura 5.15: Paquetes activos en karaf.

```
opendaylight-user@root>feature:list
Name                                | Version          | Installed | Repository                          | Description
------------------------------------------------------------------------------------------------------------------------------------------------
odl-netconf-all                     | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: All
odl-netconf-api                     | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: API
odl-netconf-mapping-api             | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Mapping API
odl-netconf-util                    | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     |
odl-netconf-impl                    | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Impl
odl-config-netconf-connector        | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Connector
odl-netconf-netty-util              | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Netty Util
odl-netconf-client                  | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Client
odl-netconf-monitoring              | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Monitoring
odl-netconf-notifications-api       | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Notification :: Api
odl-netconf-notifications-impl      | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Monitoring :: Impl
odl-netconf-ssh                     | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf Connector :: SSH
odl-netconf-tcp                     | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf Connector :: TCP
odl-netconf-mdsal                   | 1.3.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: Netconf :: Mdsal
odl-aaa-netconf-plugin              | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: AAA :: ODL NETCONF Plugin
odl-aaa-netconf-plugin-no-cluster   | 1.0.1-Beryllium-SR1 |           | odl-netconf-1.0.1-Beryllium-SR1     | OpenDaylight :: AAA :: ODL NETCONF Plugin - NO CLU
sdnhub-tutorial-netconf-exercise    | 1.0.0-SNAPSHOT      |           | tutorial-features-1.0.0-SNAPSHOT    | SDN Hub Tutorial :: OpenDaylight :: Netconf exerci
sdnhub-tutorial-tapapp              | 1.0.0-SNAPSHOT      | x         | tutorial-features-1.0.0-SNAPSHOT    | SDN Hub Tutorial :: OpenDaylight :: Tap applicatio
sdnhub-tutorial-learning-switch     | 1.0.0-SNAPSHOT      |           | tutorial-features-1.0.0-SNAPSHOT    | SDN Hub Tutorial :: OpenDaylight :: Learning switc
sdnhub-tutorial-acl                 | 1.0.0-SNAPSHOT      |           | tutorial-features-1.0.0-SNAPSHOT    | SDN Hub Tutorial :: OpenDaylight :: Access Control
odl-aaa-shiro                       | 0.3.1-Beryllium-SR1 |           | odl-aaa-0.3.1-Beryllium-SR1         | OpenDaylight :: AAA :: Shiro
odl-restconf-all                    | 1.3.1-Beryllium-SR1 |           | odl-controller-1.3.1-Beryllium-SR1  | OpenDaylight :: Restconf :: All
odl-restconf                        | 1.3.1-Beryllium-SR1 |           | odl-controller-1.3.1-Beryllium-SR1  | OpenDaylight :: Restconf
odl-restconf-noauth                 | 1.3.1-Beryllium-SR1 |           | odl-controller-1.3.1-Beryllium-SR1  | OpenDaylight :: Restconf
odl-mdsal-apidocs                   | 1.3.1-Beryllium-SR1 |           | odl-controller-1.3.1-Beryllium-SR1  | OpenDaylight :: MDSAL :: APIDOCS
framework-security                  | 3.0.3               |           | standard-3.0.3                      | OSGi Security for Karaf
standard                            | 3.0.3               | x         | standard-3.0.3                      | Karaf standard feature
aries-annotation                    | 3.0.3               |           | standard-3.0.3                      | Aries Annotations
wrapper                             | 3.0.3               |           | standard-3.0.3                      | Provide OS integration
service-wrapper                     | 3.0.3               |           | standard-3.0.3                      | Provide OS integration (alias to wrapper feature)
obr                                 | 3.0.3               |           | standard-3.0.3                      | Provide OSGi Bundle Repository (OBR) support
config                              | 3.0.3               | x         | standard-3.0.3                      | Provide OSGi ConfigAdmin support
region                              | 3.0.3               | x         | standard-3.0.3                      | Provide Region Support
package                             | 3.0.3               | x         | standard-3.0.3                      | Package commands and mbeans
```

Figura 5.16: Funcionalidades activas en karaf.

Config-subsystem.

OpenDaylight cuenta con una opción que permite al sistema tener los paquetes cargados y ordenados con el MD-SAL y sus correspondientes dependencias. Esto se puede realizar en el archivo de configuración «config.xml» y su correspondiente implementación YANG en un archivo del tipo «impl.yang».

Mininet.

Esta herramienta se ha explicado en profundidad en la sección 5.1. Se pueden especificar los flujos creados con el protocolo OpenFlow como se explicará más adelante.

5.2.2. Arquitectura.

OpenDaylight se compone de un gran número de módulos y plataformas, como se observa en la figura 5.17, desarrollados en otros proyectos. La idea es unir todas las funcionalidades de las plataformas exportando los servicios a través de interfaces de Java, llegando la mayoría a una capa de adaptación llamada MD-SAL, donde se interconectan [25]. A continuación se explican algunas de las principales partes.

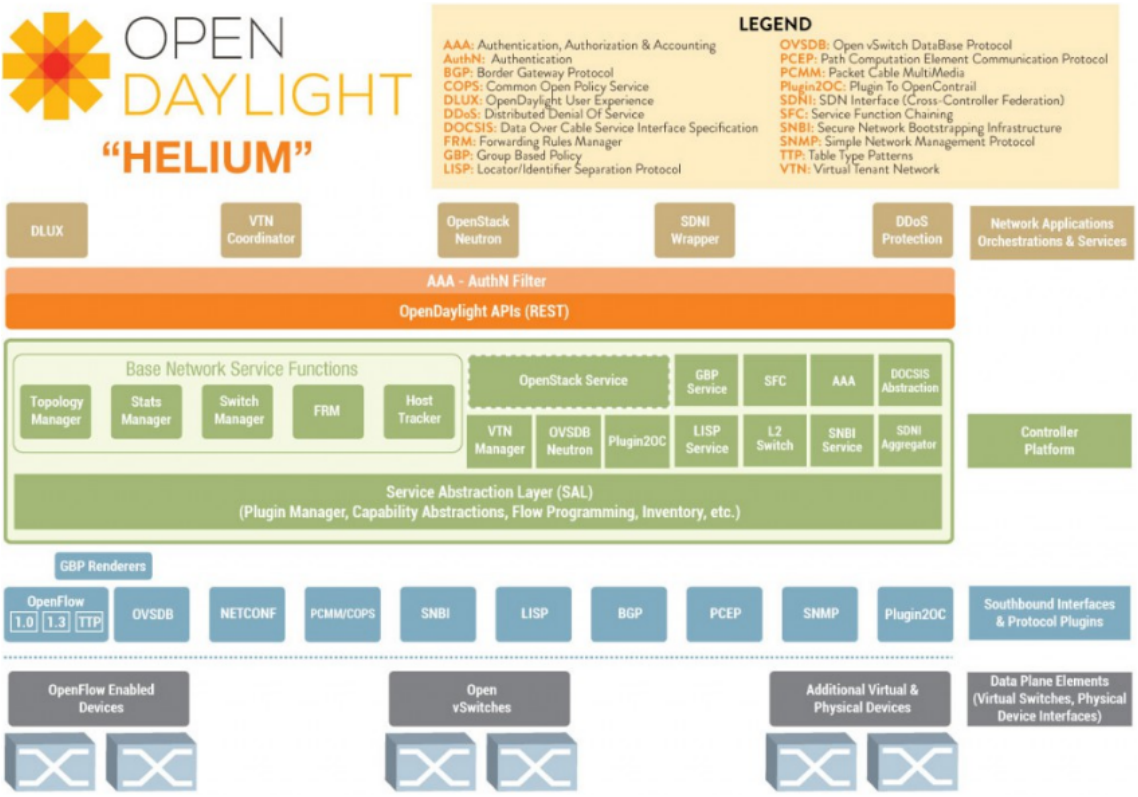


Figura 5.17: Arquitectura de OpenDaylight [25].

MD-SAL.

Es el núcleo de la plataforma donde las capas y módulos se interconectan mediante APIs. Cada API se genera con modelos definidos en lenguaje YANG. Los datos de cada aplicación se separan en datos de configuración y datos de operación.

Existe otro tipo de SAL que se puede usar en OpenDaylight: los AD-SAL basados directamente en las APIs, en lugar de en los modelos. Se ha demostrado que trabajar con arquitecturas AD-SAL es menos eficiente ya que cada API se dedica a una función diferente en lugar de estar todo centrado en un modelo principal. Por este motivo se usa MD-SAL para el proyecto.

YANG.

YANG es un lenguaje de modelado de datos para protocolos de control de red. OpenDaylight usa modelos YANG para describir la estructura básica de estas aplicaciones y organizar sus datos, y una vez creada la estructura se pasa al MD-SAL.

Existen tres tipos de operadores YANG:

- *Augmentations*: permite expandir el modelo YANG.,
- *Notifications*: envía notificaciones cuando ocurre algún suceso.
- *Remote Procedure Call* (RPC): permite a un módulo comunicarse mediante entradas/salidas con otro módulo.

Identificadores/Datos.

Los objetos se almacenan en jerarquías y se puede acceder a ellos mediante identificadores YANG. Una vez se tiene el identificador de un objeto se pueden realizar acciones de lectura o escritura sobre ellos haciendo uso de los recursos de CPU del sistema.

Plugins.

Los *plugins* son pequeñas abstracciones de código para integrar en sistemas externos. Existen dos tipos principales para OpenDaylight:

- Northbound: los dos más típicos de este tipo son «RESTCONF» y «NETCONF».
- Southbound: los dos más típicos de este tipo son «OpenFlow» y «NETCONF connector».

5.2.3. Flujos.

Para el controlador existen dos formas de crear flujos en las tablas: reactiva y proactiva. La primera crea los flujos de forma automática cuando se produce una coincidencia en su tabla, mientras que la segunda necesita que los flujos se introduzcan de forma manual por el administrador.

5.2.3.1. Flujos Reactivos.

Para ello se puede usar el código del ejemplo «*Learning Switch*» proporcionado de forma oficial [25]. El controlador hace uso de la función «*onPacketReceived*», de donde extrae los parámetros necesarios para crear los flujos y se usan distintas funciones para saber los nodos que tiene la red. Se busca una coincidencia con la tabla del controlador y en caso de encontrarse se encamina el paquete hacia ese nodo, y en caso de no hacerlo se realiza un *flood*.

Para que la red comience a funcionar es necesario iniciar tanto la red de Mininet, con el primer comando, como el controlador, con el segundo.

```
$ sudo mn --topo single,3 --mac --switch ovsk,protocols=
    OpenFlow13 --controller remote
$ feature:install sdnhub-tutorial-learning-switch
```

La Figura 5.18 muestra un ejemplo de esta inicialización.

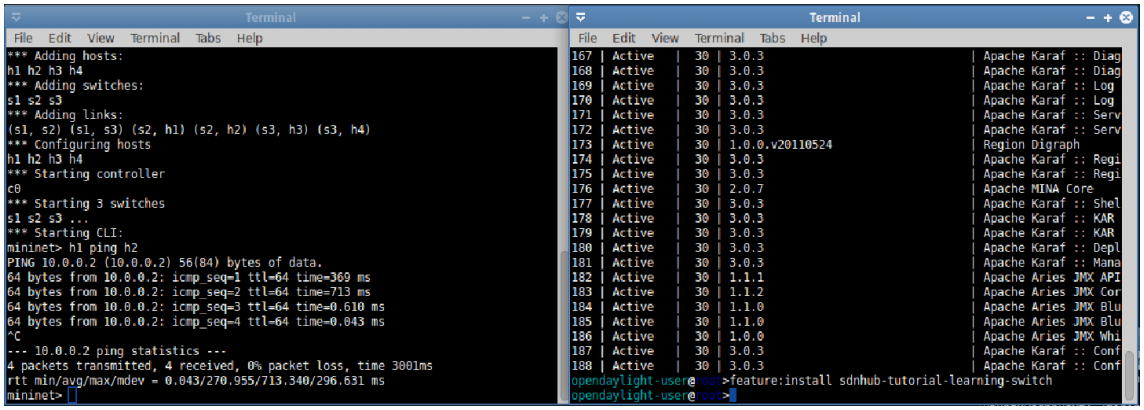


Figura 5.18: Ejemplo de funcionamiento de flujos reactivos.

5.2.3.2. Flujos Proactivos.

En este caso se inicia la red de forma similar a la anterior:

```
$ sudo mn --topo single,3 --mac --switch ovsk,protocols=
    OpenFlow13 --controller remote
```

Sin embargo, no se añaden los flujos en el comando, ya que se generan posteriormente de forma manual como se muestra en código siguiente. Tras esto la red comenzaría a funcionar correctamente.

```
$ sudo ovs-ofctl add-flow -OOpenFlow13 s$i priority=1,actions=
    output:controller
$ sudo ovs-ofctl add-flow -OOpenFlow13 s$i dl_type=0x0806,
    priority=65534,actions=output:controller
```

5.2.4. Uso en el proyecto.

En este proyecto OpenDaylight se usará como nuestro controlador de red SDN para la red de Mininet. En la segunda parte del desarrollo práctico será la pieza principal ya que todas las modificaciones que se quieren realizan para eliminar el bróker de la red MQTT tendrán que introducirse sobre este controlador, partiendo del proyecto ya creado de «*Learning Switch*».

5.3. Wireshark.

Es un analizador de protocolos de *software* libre muy extendido utilizado para analizar redes y solucionar sus problemas. Permite ver los paquetes transmitidos sobre uno o varios interfaces de red en una comunicación, y el uso de filtros de protocolo o direcciones entre otros para visualizar solamente la parte que se desea analizar.

Las variables a destacar en el uso de Wireshark se encuentran señaladas en la Figura 5.19. El recuadro amarillo indica la interfaz donde se están capturando los paquetes, en este caso la «ether0» del *host* «h1», ya que se ha abierto el programa desde el interior de un *host*. El recuadro negro señala los distintos campos que muestra Wireshark al obtener un paquete. Por último el rojo muestra el filtro utilizado donde se puede escribir cualquiera de los campos del recuadro negro, normalmente la dirección IP o el protocolo de interés.

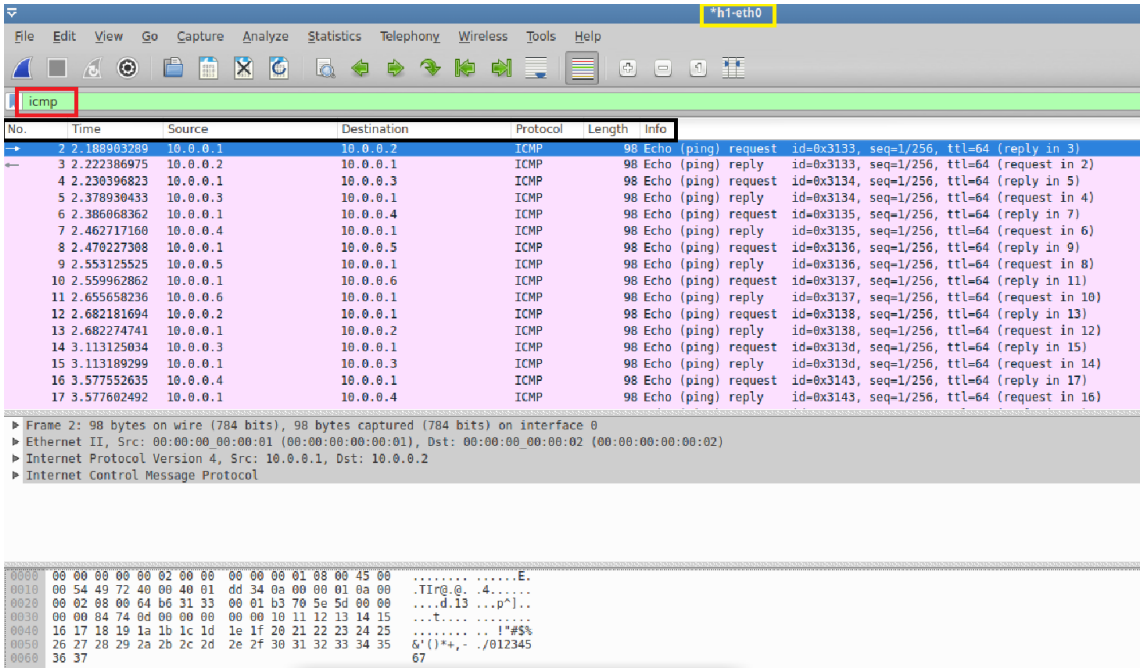


Figura 5.19: Principales variables de Wireshark.

En este proyecto se va a utilizar para analizar la transmisión de mensajes MQTT en las redes creadas, primero en una red común de MQTT y posteriormente en la red modificada, con la intención de comparar el comportamiento en ambos casos.

6

Fundamentos teóricos

En este capítulo se incluye la información recopilada correspondiente al marco teórico, concretamente sobre los protocolos más importantes para el proyecto.

MQTT es la base del proyecto, por lo que se debe conocer en profundidad todas las funcionalidades y características del protocolo, mientras que OpenFlow es muy importante para el desarrollo de los escenarios en los que se trabaje con un controlador SDN.

6.1. MQTT.

MQTT (*Message Queuing Telemetry Transport*) es un protocolo de mensajería simple diseñado para dispositivos con ancho de banda limitado, alta latencia o redes poco fiables. Se basa en el paradigma publicador-suscriptor en el que un bróker actúa como intermediario encargándose de direccionar los mensajes con el uso de un *topic* común.

El protocolo trabaja sobre TCP/IP u otros protocolos que permitan conexiones bidireccionales, ordenadas y sin pérdidas. Su principal objetivo es minimizar los requerimientos de ancho de banda de red y los recursos de los dispositivos que lo usan, manteniendo cierto grado de fiabilidad, lo que hace que su uso en IoT o conexiones *machine-to-machine* (M2M) sea muy conveniente [26].

MQTT se inventó en 1999 por el Dr. Andy Stanford-Clark de IBM y Arlen Nipper de Arcom y las versiones 5.0 y 3.1.1 (ratificada por el ISO) son estándares de OASIS [27]. Además IANA tiene reservados los puertos TCP/IP 1883 para MQTT y 8883 para MQTT sobre SSL [26].

6.1.1. Modelo.

La característica principal en el modelo MQTT es que sigue la arquitectura de comunicación cliente - servidor normalmente con una topología en estrella, con un nodo central que funciona de servidor y una capacidad de hasta 10.000 clientes [28].

A continuación se describen los componentes principales que aparecen en un escenario MQTT [29]:

1. **Bróker:** actúa como servidor encargándose de la transmisión de mensajes con los clientes y la gestión de la red. También mantiene activo el canal con los clientes respondiendo a los mensajes periódicos que estos envían.

2. **Cliente:** puede tener funciones de publicador y de suscriptor al mismo tiempo.

    a) **Publicador:** actúa como cliente y se encarga de transmitir información al bróker sobre un determinado *topic*.

   b) **Suscriptor:** actúa como cliente y se encarga de recibir información del bróker sobre un determinado *topic*.

3. **Mensaje:** unidad de datos o información que se transmite o recibe sobre un *topic*.

4. **Topic:** tema en el que los clientes pueden suscribirse para recibir información sobre ese asunto o publicarla. Es similar a una cola de mensajes, pero los mensajes pueden almacenarse hasta que sean consumidos y se pueden distribuir a varios clientes.

Para comenzar una comunicación entre el bróker y un cliente se establece una sesión, que se cerrará al concluir, con un intercambio de paquetes como se verá en profundidad en el siguiente apartado, y se puede mantener activa el tiempo que se necesite con mensajes periódicos que el cliente envía y el bróker debe responder [30].

Al ser el bróker el encargado de controlar la red y todos los mensajes, hace que la comunicación entre transmisor y receptor se desacople, lo que supone tres ventajas en comunicaciones de este tipo [31]:

- El publicador únicamente necesita conocer la dirección IP y puerto del bróker, siendo irrelevantes en la publicación los destinatarios del mensaje.

- Publicador y suscriptor no tienen que estar conectados a la vez ya que el bróker se encargará de almacenar la información.

- Publicador y suscriptor no necesitan sincronizarse.

En la figura 6.1 podemos ver un escenario básico de MQTT con tres clientes, donde el cliente de la izquierda hace la función de publicador y los de la derecha de suscriptores. El mensaje enviado desde el publicador en cierto *topic* llega hasta el bróker, que se encarga de distribuirlo a los dos clientes suscritos en el mismo *topic*.

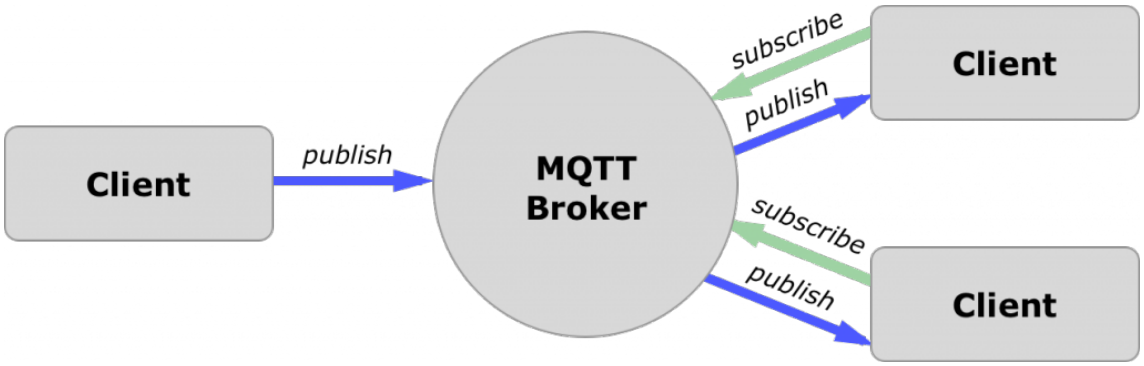


Figura 6.1: Modelo de comunicación MQTT [29].

6.1.2. Formato.

En el protocolo MQTT se intercambian una serie de paquetes de control en un orden determinado para llevar a cabo la comunicación, los cuales se componen de tres partes diferentes dependiendo del tipo de paquete.

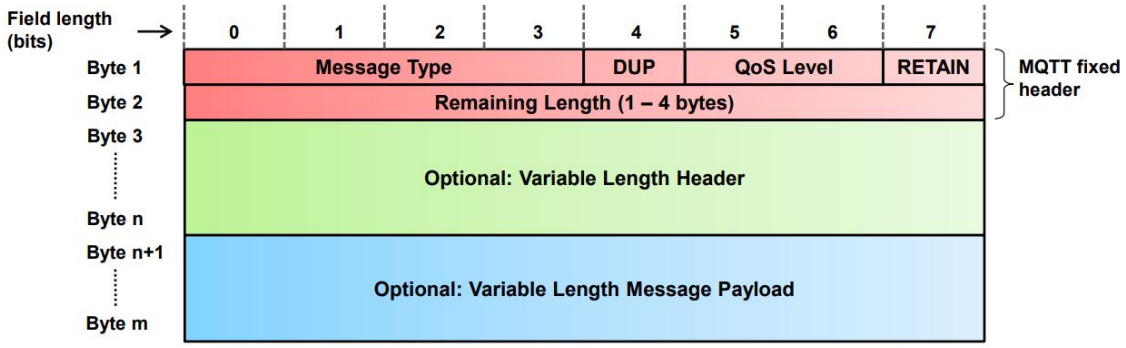


Figura 6.2: Estructura paquete MQTT [30].

6.1.2.1. Cabecera fija.

La cabecera fija o *Fixed Header* es la parte inicial de los paquetes MQTT y es la única que se encuentra presente en todos ellos. Esta cabecera se divide a su vez en distintos campos.

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2...	Remaining Length							

Figura 6.3: *Fixed Header* paquete MQTT [27].

- *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*: se encuentran en la segunda mitad del primer byte y 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 (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.

Nombre	Valor	Dirección del flujo	Descripción	Payload
Reserved	0	-	Reservado	No hay
`connect`	1	Cliente → Servidor	Petición de conexión	Obligatorio
`connack`	2	Cliente ← Servidor	Confirmación conexión	No hay
`publish`	3	Cliente ↔ Servidor	Mensaje de publicación	Opcional
`puback`	4	Cliente ↔ Servidor	Confirmación publicación	No hay
`pubrec`	5	Cliente ↔ Servidor	Recepción de publicación (entrega asegurada I)	No hay
`pubrel`	6	Cliente ↔ Servidor	Lanzamiento de publicación (entrega asegurada II)	No hay
`pubcomp`	7	Cliente ↔ Servidor	Publicación completada (entrega asegurada III)	No hay
`subscribe`	8	Cliente → Servidor	Petición de suscripción	Obligatorio
`suback`	9	Cliente ← Servidor	Confirmación suscripción	Obligatorio
`unsubscribe`	10	Cliente → Servidor	Petición de cancelación de suscripción	Obligatorio
`unsuback`	11	Cliente ← Servidor	Confirmación cancelación de suscripción	No hay
`pingreq`	12	Cliente → Servidor	Solicitud de PING	No hay
`pingresp`	13	Cliente ← Servidor	Respuesta de PING	No hay
`disconnect`	14	Cliente → Servidor	Client desconectado	No hay
Reserved	15	-	Reservado	No hay

Tabla 6.1: Descripción de los mensajes de control MQTT [27].

6.1.2.2. Cabecera variable.

La cabecera variable o *Variable Header* se encuentra presente en algunos de los paquetes MQTT. Dependiendo del tipo de mensaje o de su nivel de calidad de servicio (QoS) puede contener distintos campos que aporten información específica, como el nombre del *topic,* pero casi siempre contendrá los bytes de *Packet Identifier/Message ID,* que identifica el paquete con la fórmula del bit más significativo (MSB) y del menos significativo (LSB).

Bit	7	6	5	4	3	2	1	0
byte 1	Packet Identifier MSB							
byte 2	Packet Identifier LSB							

Figura 6.4: *Packet Identifiers* [27].

6.1.2.3. *Payload.*

La carga útil o *Payload* es la parte final del paquete, de longitud variable, que contiene la información que se quiere transmitir y que se encuentra solo en algunos tipos de paquetes como se muestra en el campo *Payload* de la Tabla 6.1.

6.1.2.4. Tipos de mansajes.

`CONNECT`

Es el primer mensaje que manda el cliente al servidor una vez que la comunicación se ha establecido y solo debe enviarse uno. A continuación, se muestra un mensaje `connect` y se explican los campos específicos de *Variable Header* y *Payload*.

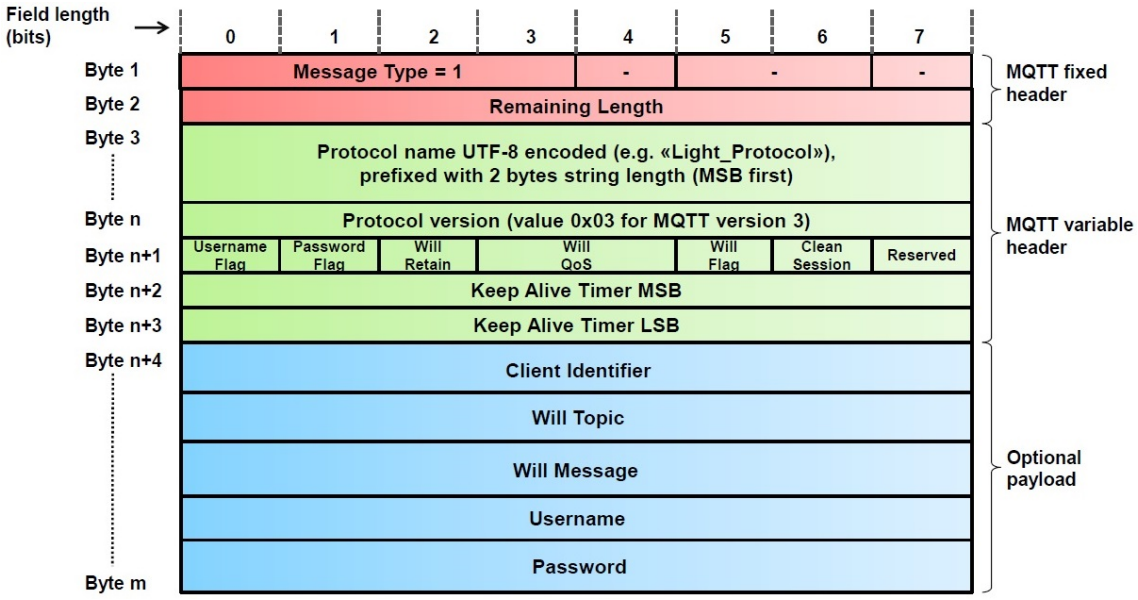


Figura 6.5: Mensaje `connect` [30].

- *Protocol name:* caracteres con el nombre del protocolo codificado en UTF-8.
- *Protocol version:* entero con el número de versión.
- *User/Password Flag:* activo (1) si los campos de usuario/contraseña aparecen en el *payload*.
- *Will Retain:* activo si el servidor debe retener el mensaje.
- *Will QoS:* especifica el nivel de QoS.
- *Will Flag:* indica si aparece un mensaje *Will* en el *payload*.
- *Clean Session:* si está activo (1), el servidor limpiará la información sobre el cliente.
- *Keep Alive Timer:* intervalo de tiempo máximo entre mensajes del cliente. El servidor comprobará el estado.
- *Client Identifier:* identificación del cliente.
- *Will Topic: topic* en el que se publicará el mensaje.
- *Will Message:* mensaje que se publicará en el *topic*.
- *Username/Password:* usuario y contraseña.

`CONNACK`

Es el mensaje de respuesta a un `connect` en el que se acepta (0) o se rechaza la conexión (1-5). La conexión puede rechazarse por incompatibilidad de versión (1), error de identificador (2), de servidor (3), de usuario o contraseña (4) o por no estar autorizada la conexión (5).

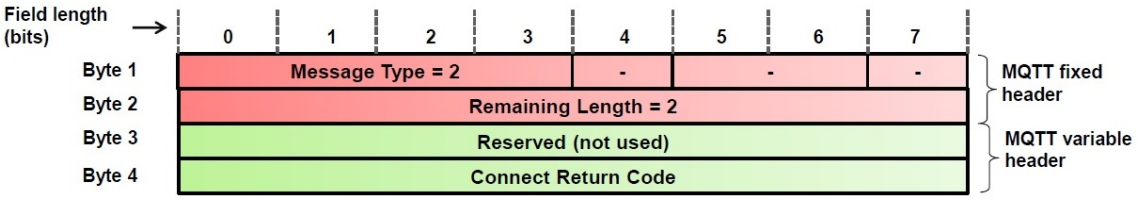


Figura 6.6: Mensaje `connack` [30].

`PUBLISH`

Este mensaje se utiliza para publicar mensajes en un *topic* determinado y que los suscriptores reciban la información.

- *Topic Name String Length/Topic Name:* tema donde se publica el mensaje. Los dos primeros bytes indican la longitud del nombre.
- *Message ID:* solo presente si la QoS es distinta a 0.
- *Publish Message:* mensaje a publicar.

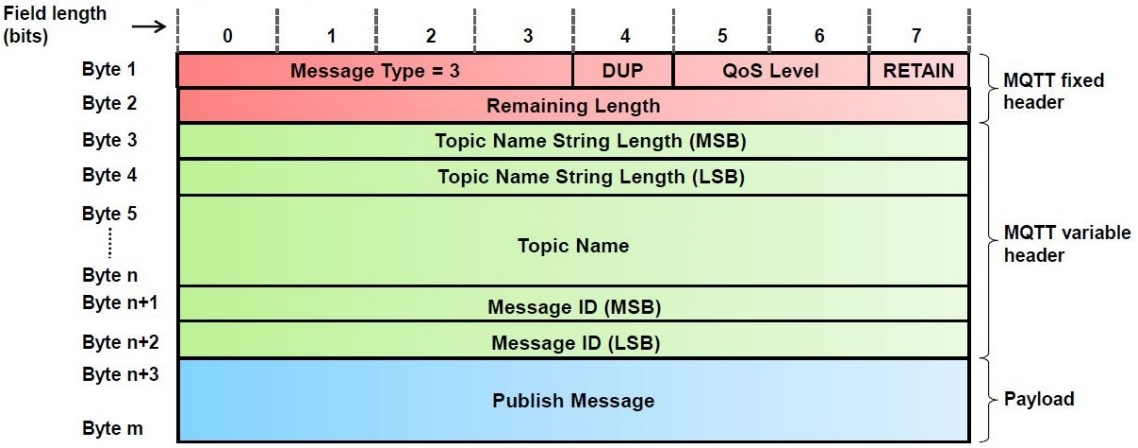


Figura 6.7: Mensaje `publish` [30].

`PUBACK`

Este paquete se utiliza como confirmación en caso de que el mensaje `publish` use QoS de nivel 1.

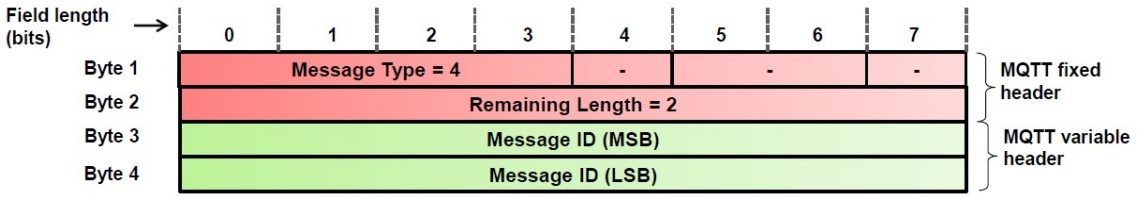


Figura 6.8: Mensaje `puback` [30].

`PUBREC/PUBREL/PUBCOMP`

Estos paquetes se utilizan como confirmación en caso de que el mensaje `publish` use QoS de nivel 2. La respuesta a `publish` será `pubrec`, la respuesta a este será `pubrel` y, por último, se confirmará con `pubcomp`.

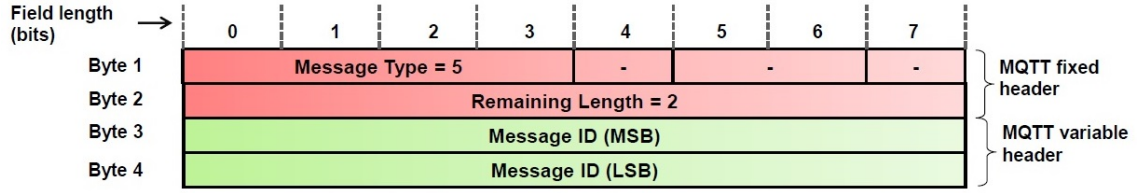


Figura 6.9: Mensaje `pubrec` [30].

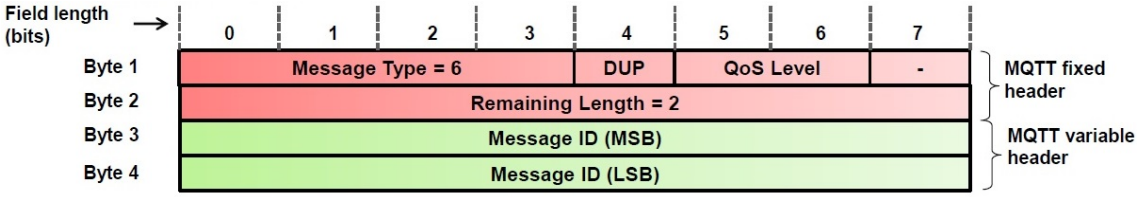


Figura 6.10: Mensaje `pubrel` [30].

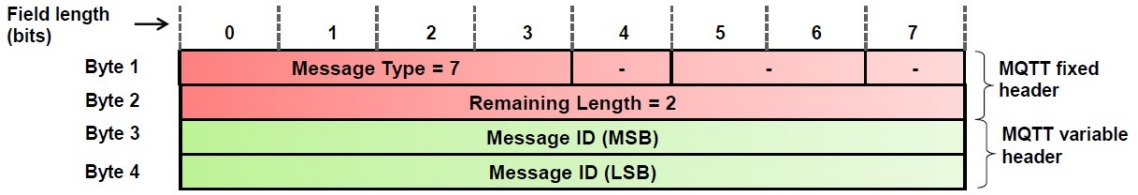


Figura 6.11: Mensaje `pubcomp` [30].

`SUBSCRIBE`

Este paquete se enviará desde el cliente hacia el servidor para suscribirse en uno o más *topics* y que el cliente reciba todos los mensajes que se publiquen en dicho *topic*. Deberá incluir siempre el *Topic Name* y los campos de QoS.

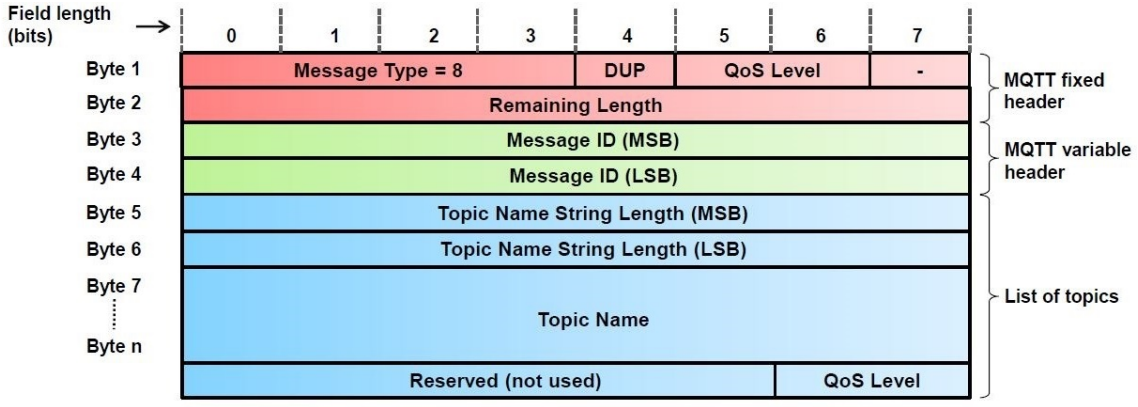


Figura 6.12: Mensaje `subscribe` [30].

`SUBACK`

Este paquete se utiliza como confirmación a los paquetes `subscribe`.

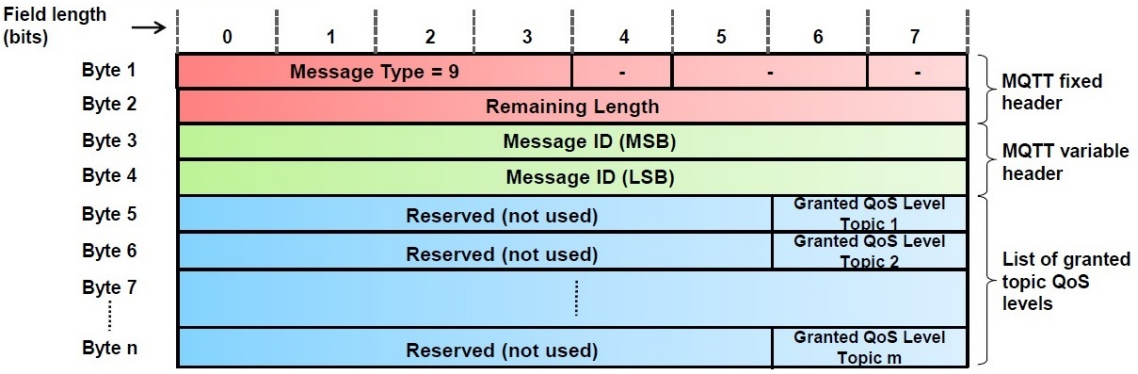


Figura 6.13: Mensaje `suback` [30].

`UNSUBSCRIBE`

En estos paquetes el cliente notifica al servidor para que no continúe enviando información sobre uno o más *topics*.

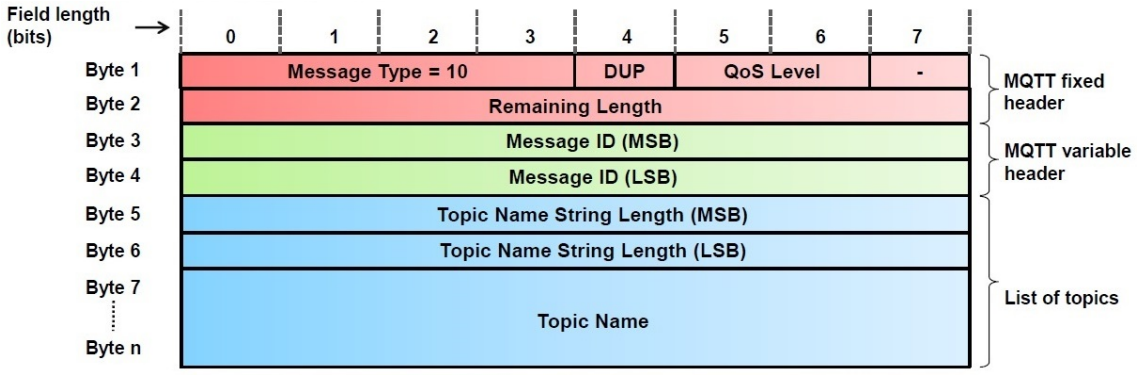


Figura 6.14: Mensaje `unsubscribe` [30].

`UNSUBACK`

Este paquete se utiliza como confirmación a los paquetes `unsubscribe`.

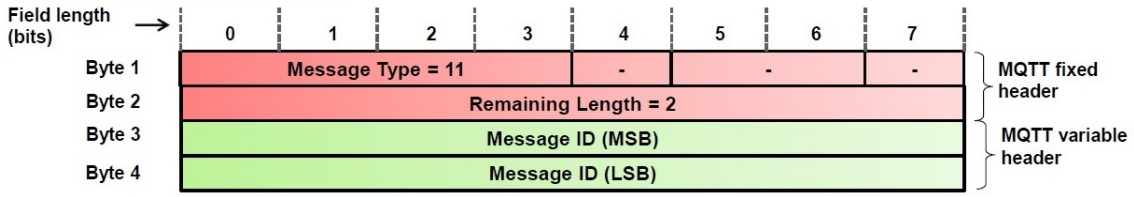


Figura 6.15: Mensaje `unsuback` [30].

`DISCONNECT/PINGREQ/PINGRESP`

Estos mensajes, junto a `connect/connack`, constituyen los mensajes de sesión del protocolo MQTT. Con el mensaje `disconnect` se informa que la sesión ha terminado y debe cerrarse, mientras que con `pingreq/pingresp` la comunicación se mantiene activa para que no se desactive transcurrido un tiempo.

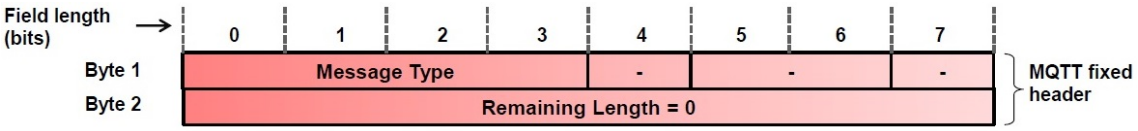


Figura 6.16: Mensajes `disconnect/pingreq/pingresp` [30].

6.1.3. QoS.

La calidad de servicio (QoS) es el nivel de acuerdo entre emisor y el receptor para garantizar la correcta entrega de un mensaje. Aunque TCP/IP garantiza la entrega de datos, las pérdidas pueden seguir ocurriendo si las conexiones TCP se rompen, por lo que el protocolo MQTT incluye tres niveles distintos de QoS [31]:

- *At most once* (0): es el nivel mínimo de calidad en el que se utiliza *best-effort*. La recepción del mensaje no está garantizada ya que los mensajes no se almacenan y, una vez que se envían, el protocolo se olvida de ellos.

- *At least once* (1): este nivel de calidad garantiza que el mensaje ha llegado al destino al menos una vez. Los mensajes se almacenan hasta que se recibe un `puback` con el que se confirma la correcta entrega.

- *Exactly once* (2): es el nivel más alto de calidad de servicio (y el más lento) en el que se asegura que el mensaje ha llegado solo una vez. Esto se consigue con un *four-part handshake*, es decir, un intercambio de cuatro mensajes para confirmar la llegada del mensaje, como se explicó en el punto anterior y se representa en la Figura 6.17.

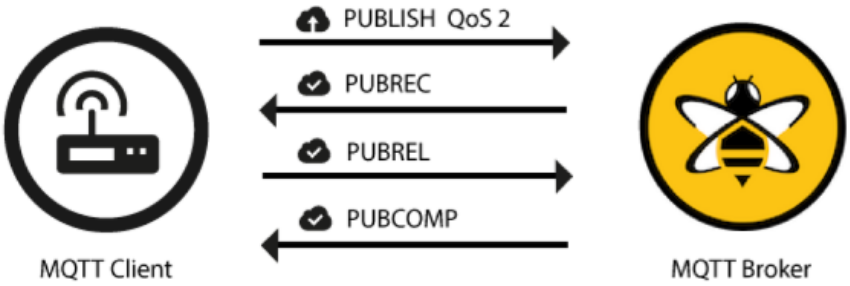


Figura 6.17: Mensajes para QoS nivel 2 [31].

6.1.4. Seguridad.

El protocolo MQTT ofrece distintas opciones para implementar mecanismos de seguridad y que las comunicaciones no se vean comprometidas.

El más recomendado es el uso de MQTT con TLS (*secure-mqtt*). Se implementa en el servidor y debe usar el puerto TCP 8883, que permite la encriptación en el intercambio de mensajes usando concretamente los algoritmos de cifrado simétrico AES o DES. Además, el uso de TLS provee integridad y privacidad en la comunicación [27].

También se proporciona autenticación con los campos usuario y contraseña de los paquetes y comprobando que los identificadores del paquete o las direcciones corresponden a lo esperado.

6.1.5. Variantes.

En este apartado se muestran algunas de las posibles variantes que se utilizan sobre el protocolo MQTT para mejorar algunas de sus características.

- SMQTT (*Secure MQTT*): esta variante usa TLS con el protocolo por lo que lo hace más seguro, como se ha explicado en el apartado anterior.
- MQTT-SN (*MQTT Sensor Networks*): es una variante diseñada específicamente para redes inalámbricas con sensores de poca batería y la mayor parte del tiempo en *sleep mode*. La diferencia principal es que usa unos paquetes con *payload* reducido y que utilizan UDP para no necesitar una conexión permanente [32].
- DM-MQTT (*Direct Multicast MQTT*): es una variación del protocolo para funcionar de forma más eficiente en redes *multicast* de SDN con *edge computing*. Esta variante ya se ha explicado con mayor profundidad en la sección 4.1.

6.2. OpenFlow.

OpenFlow protocol (OFP) es un protocolo de comunicación para redes definidas por *software* (SDN) que surgió en la Universidad de Stanford en 2008 y utiliza los protocolos TCP/SSL de capas 4 y 5 del modelo OSI. En las redes SDN los planos de control y de datos suelen estar separados para hacer un uso más eficiente de los recursos de la red, algo imposible en las redes convencionales, donde trayectorias de control y datos ocurren en el propio dispositivo [33].

6.2.1. Funcionamiento.

Openflow es el lenguaje de comunicación entre el controlador y los *switches* OpenFlow. El protocolo permite a un servidor controlar las rutas que usarán los conmutadores para enviar paquetes, centralizando estas decisiones y pudiendo programar la red al completo, independientemente de los conmutadores. De esta forma, el plano de datos sigue siendo responsabilidad del dispositivo mientras que el enrutamiento de alto nivel lo dirige el servidor/controlador [34].

Los componentes principales de OpenFlow son:

- **Controlador:** se encarga de plantear la red y programar el procesamiento de los flujos de paquetes. Dialoga con los *switches* para transmitirles la información.

- ***OpenFlow Switch* (OFS):** dispositivo de red que conecta con los equipos finales, que cuenta con un canal seguro para comunicarse con el controlador y contiene las tablas de flujo.

- **Tablas de flujos:** indican en las entradas de la tabla la acción a realizar con cada flujo, de forma que tengan la información necesaria para procesar cualquier tipo de dato.

En la Figura 6.18 se muestra un escenario trabajando con OFP. Primero, el controlador genera la información básica, que se lleva al *switch* (OFS) mediante OFP a través de un canal seguro establecido con SSL a nivel de *software*. Una vez que lleguen los datos se crearán tablas de flujo a nivel de *hardware* y se añadirán las entradas con sus correspondientes acciones para cada flujo [33].

Cuando aparece un flujo que no tiene ninguna coincidencia con la tabla, el paquete se enviará al controlador, que definirá un nuevo flujo para el paquete y creará una nueva entrada para la tabla. Estas entradas se envían al OFS para añadirlas a la tabla y se devuelve el paquete para volver a procesarlo tras la actualización de la tabla.

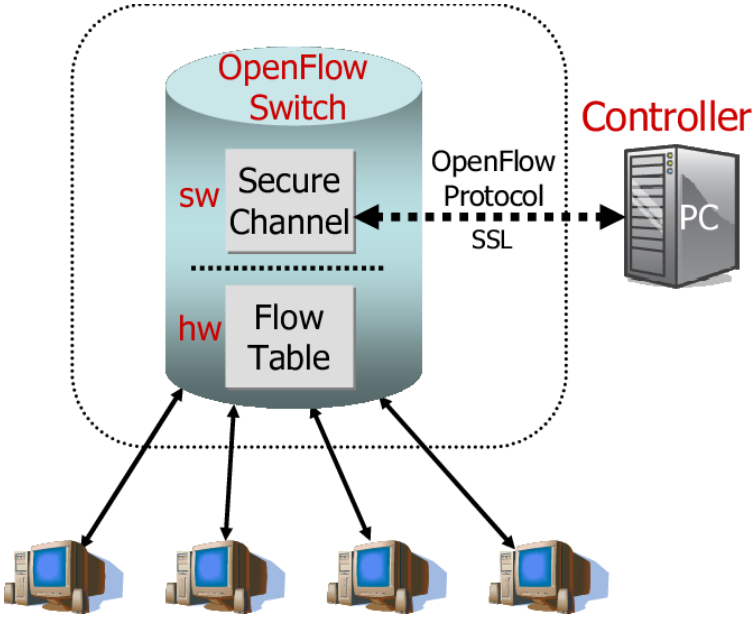


Figura 6.18: Arquitectura OpenFlow [33].

6.2.2. Tablas.

Las tablas de flujo OpenFlow soportan el procesamiento segmentado o *pipeline*. Este tipo de procesamiento permite al *switch* contener numerosas tablas de flujo con múltiples entradas cada una, y en cada tabla se actuará de una forma distinta dependiendo de las reglas que se hayan indicado [35].

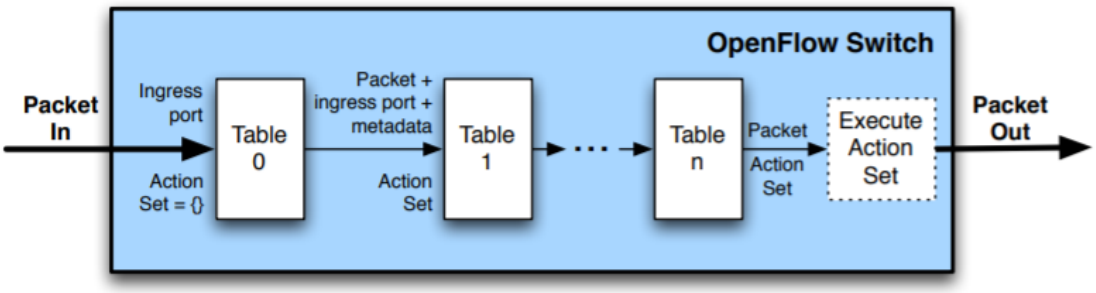


Figura 6.19: OpenFlow *pipeline* [35].

Las tablas de flujo se componen de entradas, identificadas de forma única por sus campos *match fields* y *priority*, y contienen los siguiente elementos[35]:

- *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

Tabla 6.2: Componentes de la tabla de flujo [35].

Con las entrada de las tablas de flujo se pueden realizar las siguientes acciones [33]:

- *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.*

En este apartado se explica el proceso que sigue un paquete en el interior del OFS, como se muestra en la Figura 6.20.

El *switch* comienza buscando coincidencias en la primera tabla de flujo y, dependiendo de si cuenta con procesamiento secuencial, seguirá buscando en las siguientes. Se extraen los *Match Fields* del paquete para buscar las coincidencias, normalmente las direcciones IP, MAC o puertos, o si se ha aplicado alguna acción específica en el flujo.

Si los valores de los *Match Fields* coinciden con los de una entrada en la tabla de flujo se produce una coincidencia, se selecciona la 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.

En caso de que no exista ninguna coincidencia en la tabla se procederá a descartar el paquete (*Drop*).

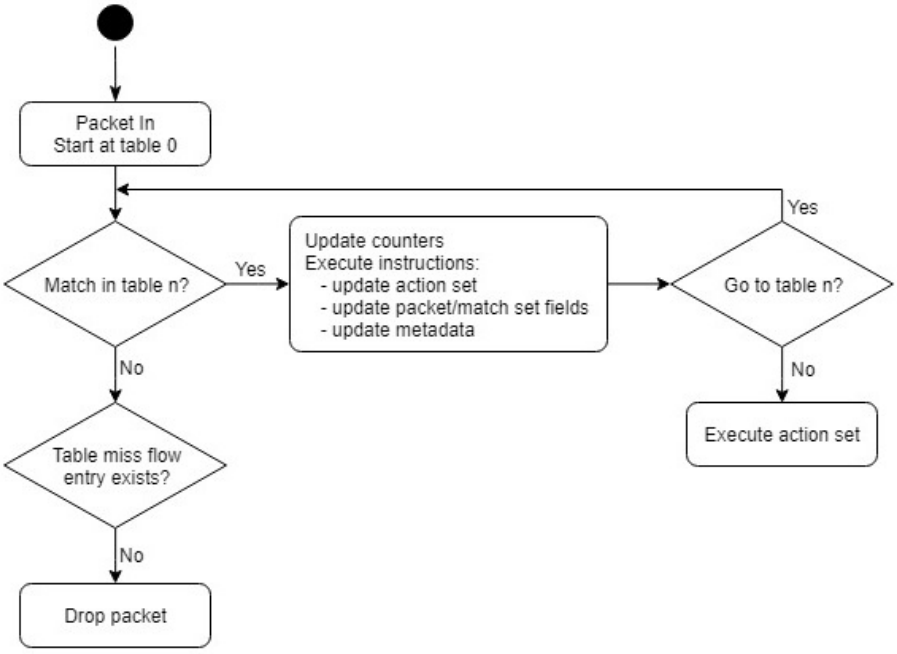


Figura 6.20: Diagrama de flujo de un paquete dentro del OpenFlow *Switch* [35].

6.2.4. Mensajes.

A través de la interfaz OpenFlow el controlador y el *switch* se conectan para intercambiar mensajes que deben seguir el formato del protocolo. Los mensajes pueden ser de tres tipos distintos [35]:

- *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 capítulo, partiendo de la teoría adquirida en los capítulos anteriores, se va a explicar el procedimiento para lograr los objetivos del proyecto. Al igual que en el capítulo 2, se van a distinguir tres apartados progresivos, que permitirán alcanzar el objetivo final, es decir, obtener un entorno MQTT sin bróker convencional.

7.1. Estudio MQTT.

En esta primera parte del desarrollo práctico se va a implementar un escenario MQTT completo con el que estudiar el comportamiento real del protocolo en cada situación tras ver su supuesto funcionamiento en el la sección 6.1.

Primero, se hará un diseño de red con el que trabajar durante el resto del desarrollo y a continuación se implementará haciendo uso de la herramienta Mininet explicada en la sección 5.1. Por último, se estudiará un estudio con la herramienta Wireshark, explicada en la sección 5.3, para analizar el intercambio de mensajes entre los clientes MQTT pasando por el bróker, en cada una de las posibles situaciones.

Tras realizar este estudio de MQTT los resultados obtenidos podrán compararse con los de la siguiente sección, donde se llevarán a cabo modificaciones para detectar posibles defectos en la modificación del protocolo.

7.1.1. Diseño de red.

Este proyecto se basa en redes de comunicación sobre las que se aplican los distintos protocolos estudiados, por lo que se decide diseñar una topología de red adecuada a los distintos escenarios a crear. Para ello se va a utilizar un *script* de Python con el que Mininet pueda crear la red, que será de un tamaño reducido pero con suficientes niveles y dispositivos para ver el funcionamiento completo de MQTT.

La topología constará de cuatro *switches* y seis *hosts*, un *switch* en un nivel superior conectado a los otros tres, y cada uno de estos tres *switches* tan solo se conectará a dos *host* y al *switch* del nivel superior. La topología está representada en la imagen 7.1.

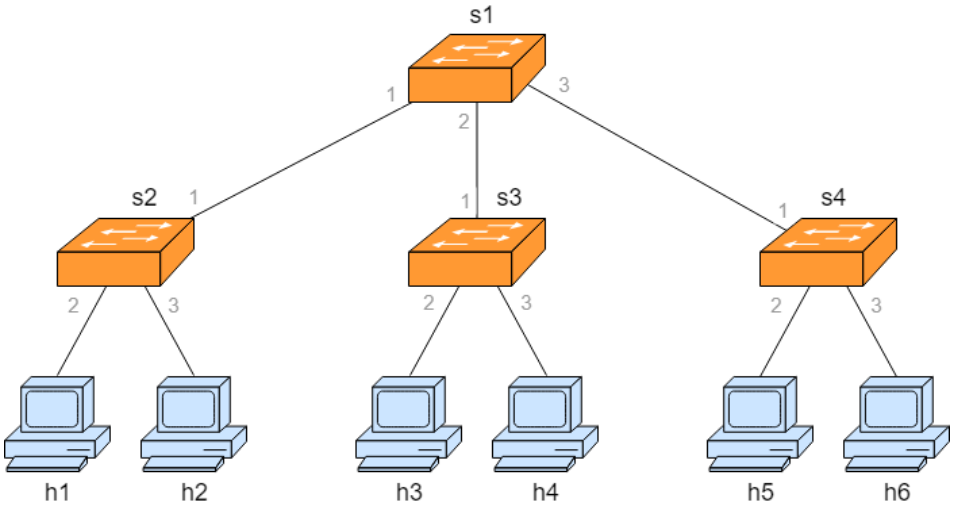


Figura 7.1: Topología de red.

El *script* con el que se genera la topología en Mininet se muestra en el código inferior. Como se puede ver se inicializa la clase con «`__init__()`» y a continuación se añaden los dispositivos, utilizando las funciones «`addSwitch()`» para los *switches* y «`addHost()`» para los *hosts*. Finalmente, se crean los *links* que unen los dispositivos entre sí con «`addLink()`».

```
"""Custom topology"""

from mininet.topo import Topo

class MyTopo( Topo ):
    def __init__( self ):
        "Create custom topo."
        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        highSwitch = self.addSwitch( 's1' )
        leftSwitch = self.addSwitch( 's2' )
        middleSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
        baHost = self.addHost( 'h1' )
        bbHost = self.addHost( 'h2' )
        caHost = self.addHost( 'h3' )
        cbHost = self.addHost( 'h4' )
        daHost = self.addHost( 'h5' )
        dbHost = self.addHost( 'h6' )

        # Add links
        self.addLink( highSwitch, leftSwitch )
        self.addLink( highSwitch, middleSwitch )
        self.addLink( highSwitch, rightSwitch )
        self.addLink( leftSwitch, baHost )
        self.addLink( leftSwitch, bbHost )
        self.addLink( middleSwitch, caHost )
        self.addLink( middleSwitch, cbHost )
        self.addLink( rightSwitch, daHost )
        self.addLink( rightSwitch, dbHost )

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

7.1.2. Implementación.

En este punto se explica cómo poner en funcionamiento la red diseñada en el apartado anterior. Para llevar a cabo esta implementación se necesita una máquina que disponga de Mininet y, para asegurarse de que la máquina no modifica el funcionamiento de los protocolos por alguna modificación o configuración indebida, se va a usar la máquina virtual que proporciona la página oficial de Mininet [18].

Tras descargar la imagen de la máquina virtual y encenderla se accederá a ella usando SSH a través de un servidor X11 [19]. Esto tiene como ventaja abrir los *hosts* en terminales externos para usar aplicaciones y controlarlos y facilitar el trabajo en la consola virtual. Para acceder con SSH se puede usar por ejemplo el programa Putty introduciendo la dirección IP de la máquina virtual, que en este caso se ha configurado como 192.168.56.2 en uno de los interfaces.

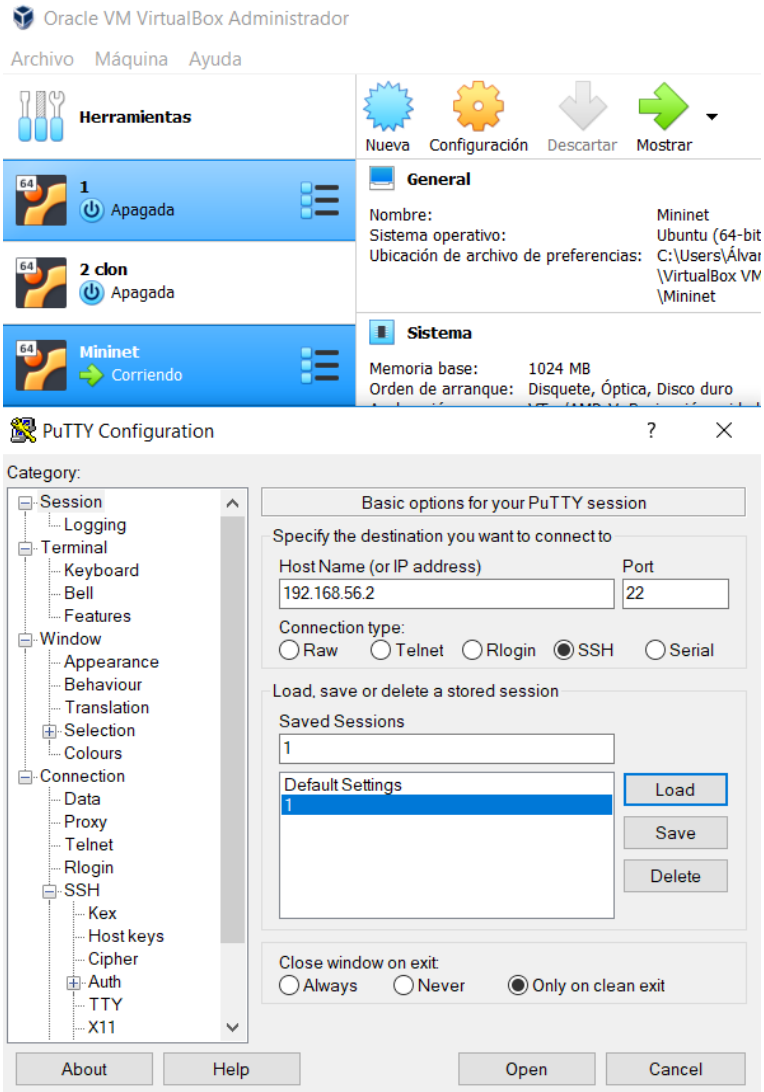


Figura 7.2: Putty.

Una vez dentro hay que introducir el usuario y contraseña de la máquina que en este caso ambos son «mininet». Para poner la red en funcionamiento se ejecuta el *script* con el diseño de la red (guardado en el directorio «mininet/custom») usando el siguiente comando

tal y como aparece en Figura 7.3.

```
$ sudo mn custom mininet/custom/custom-topo-alv.py topo mytopo
```

```
mininet@mininet-vm:~$ sudo mn --custom mininet/custom/custom-topo-alv.py --topo mytopo
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s2, h1) (s2, h2)
(s3, h3) (s3, h4) (s4, h5) (s4, h6)
*** Configuring hosts
h1 h2 h3 h4 h5 h6
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet>
```

Figura 7.3: Comando para ejecutar la red.

El siguiente paso es elegir los tres *hosts* con los que formar el escenario MQTT: uno para actuar como bróker, otro como suscriptor y otro como publicador. En este caso se han elegido los *hosts* h1, h3 y h5, respectivamente, porque cada uno pertenece a un *switch* diferente, lo que permite comprobar que los mensajes recorren la red y tienen acceso más allá de su *switch*.

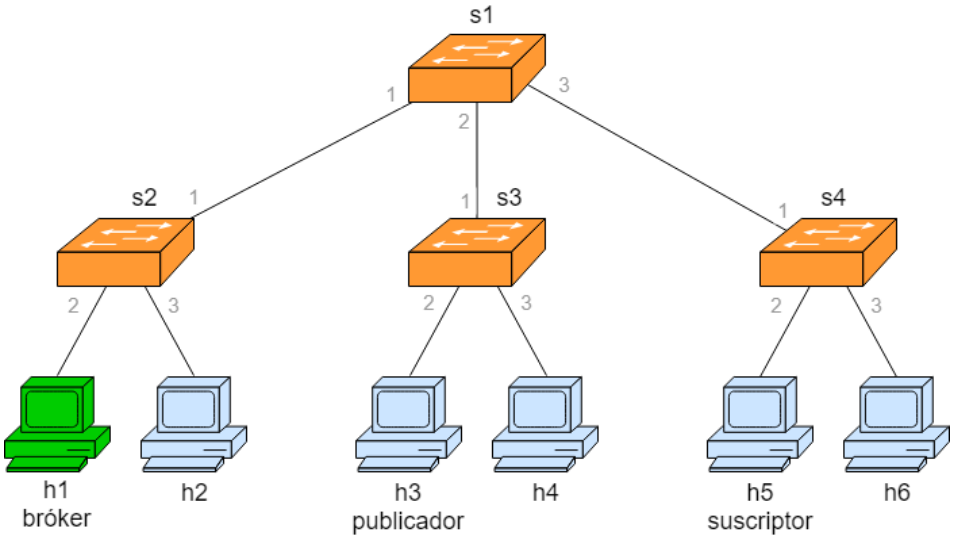


Figura 7.4: Comando para ejecutar la red.

Usando el comando «`xterm X`», donde «`X`» se sustituye por el nombre de un elemento, se puede abrir en un nuevo terminal el *host* indicado y los comandos introducidos para cada caso son:

- En h1: Inicia el bróker.

  ```
  $ sudo service mosquitto start
  ```

- En h3: Inicia un cliente que se suscribe al bróker de h1 (con dirección IP 10.0.0.1) en el *topic* «prueba».

```
$ mosquitto_sub -h 10.0.0.1 -t prueba -q 0
```

- En h5: Inicia un cliente que publica al bróker de h1 (con IP dirección 10.0.0.1) en el *topic* «prueba» el mensaje «hola».

```
$ mosquitto_pub -h 10.0.0.1 -t prueba -q 0 -m hola
```

El comando `-h` indica la dirección IP del bróker, `-t` el *topic* donde se publica, `-q` la QoS y `-m` el mensaje a transmitir. En la Figura 7.5 se puede ver el escenario en funcionamiento.

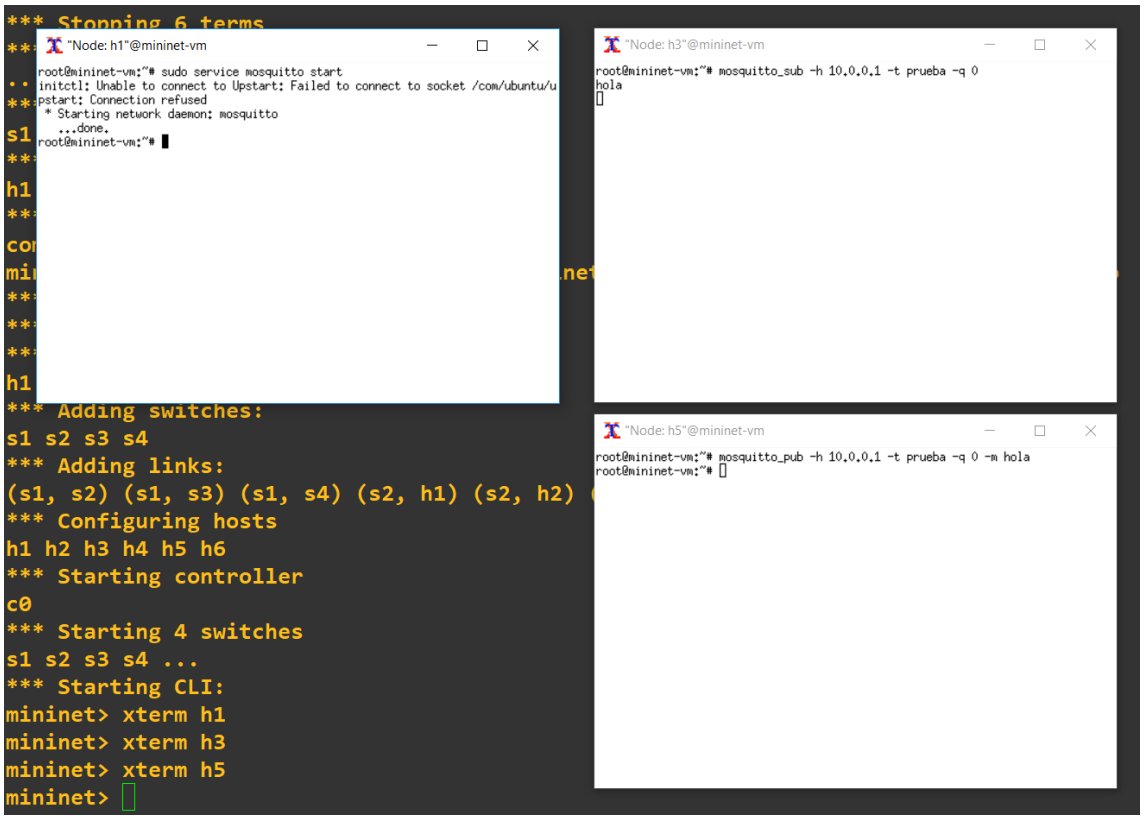


Figura 7.5: Escenario en funcionamiento.

7.1.3. Estudio Wireshark.

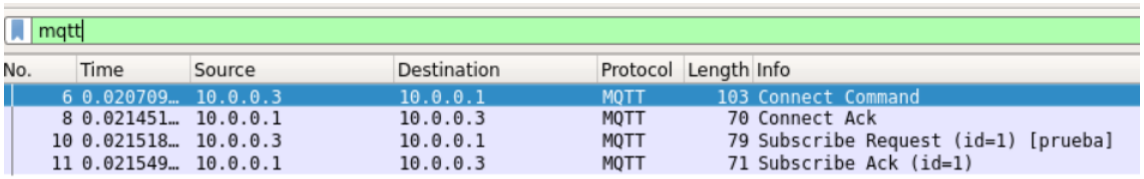
En este apartado se van a analizar los mensajes transmitidos con el protocolo MQTT utilizando la herramienta Wireshark. Las capturas se realizarán desde la interfaz que conecta con el bróker (s2-eth2:h1-eth0), por donde pasan todos los mensajes.

Para ordenar los posibles casos se crearán tres apartados según el tipo de mensaje que sean: mensajes de suscripción que envía el suscriptor (dirección IP 10.0.0.3) al bróker (dirección IP 10.0.0.1), mensajes de publicación donde un publicador (dirección IP 10.0.0.5) publica en el bróker, o mensajes de publicación en una suscripción donde se suscribe un dispositivo y se publica en su *topic*. Dentro de estos mensajes podemos tener tres casos según la QoS (0, 1 o 2) que se utilice.

Mensajes de suscripción.

- **QoS=0:** El suscriptor manda un mensaje *connect* al bróker y cuando recibe el *ack* (*acknowledgement*) manda un *subscribe*, que el bróker vuelve a confirmar con un *ack*.

```
$ mosquitto_sub -h 10.0.0.1 -t prueba -q 0
```



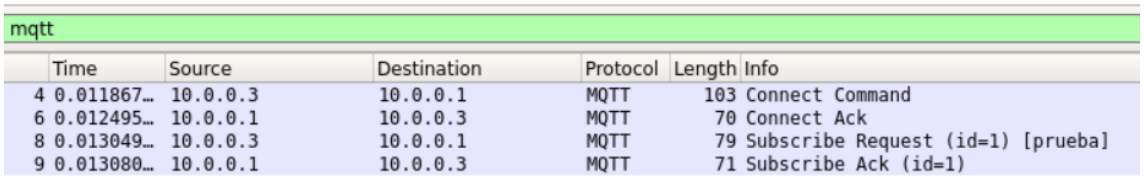
mqtt

No.	Time	Source	Destination	Protocol	Length	Info
6	0.020709…	10.0.0.3	10.0.0.1	MQTT	103	Connect Command
8	0.021451…	10.0.0.1	10.0.0.3	MQTT	70	Connect Ack
10	0.021518…	10.0.0.3	10.0.0.1	MQTT	79	Subscribe Request (id=1) [prueba]
11	0.021549…	10.0.0.1	10.0.0.3	MQTT	71	Subscribe Ack (id=1)

Figura 7.6: Mensajes suscriptor con QoS=0.

- **QoS=1:** No se produce ningún cambio respecto a los mensajes de QoS=0.

  ```
  $ mosquitto_sub -h 10.0.0.1 -t prueba -q 1
  ```



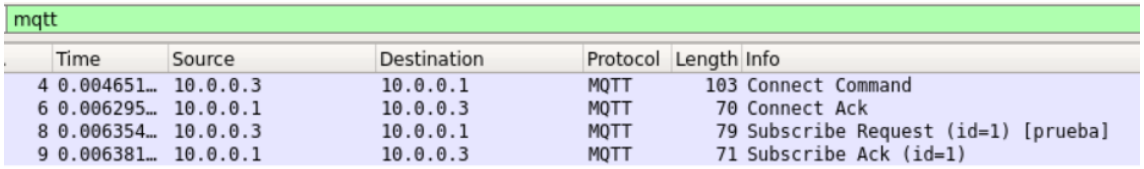
mqtt

	Time	Source	Destination	Protocol	Length	Info
4	0.011867…	10.0.0.3	10.0.0.1	MQTT	103	Connect Command
6	0.012495…	10.0.0.1	10.0.0.3	MQTT	70	Connect Ack
8	0.013049…	10.0.0.3	10.0.0.1	MQTT	79	Subscribe Request (id=1) [prueba]
9	0.013080…	10.0.0.1	10.0.0.3	MQTT	71	Subscribe Ack (id=1)

Figura 7.7: Mensajes suscriptor con QoS=1.

- **QoS=2:** No se produce ningún cambio con los mensajes de QoS=0.

  ```
  $ mosquitto_sub -h 10.0.0.1 -t prueba -q 2
  ```



mqtt

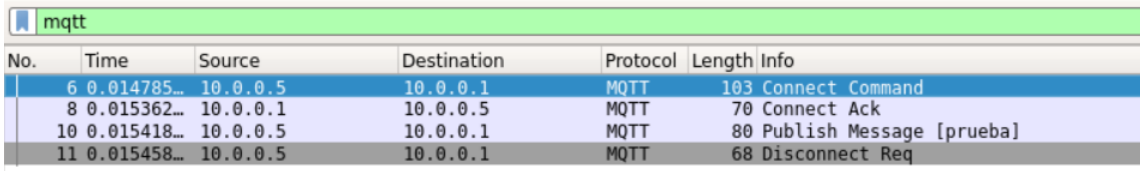
	Time	Source	Destination	Protocol	Length	Info
4	0.004651…	10.0.0.3	10.0.0.1	MQTT	103	Connect Command
6	0.006295…	10.0.0.1	10.0.0.3	MQTT	70	Connect Ack
8	0.006354…	10.0.0.3	10.0.0.1	MQTT	79	Subscribe Request (id=1) [prueba]
9	0.006381…	10.0.0.1	10.0.0.3	MQTT	71	Subscribe Ack (id=1)

Figura 7.8: Mensajes suscriptor con QoS=2.

Mensajes de publicación.

- **QoS=0:** El publicador manda un mensaje *connect* al bróker y cuando recibe el *ack* manda un *publish* con el mensaje que quiere transmitir. Cierra la comunicación con un *disconnect*.

  ```
  $ mosquitto_pub -h 10.0.0.1 -t prueba -q 0 -m hola
  ```



mqtt

No.	Time	Source	Destination	Protocol	Length	Info
6	0.014785…	10.0.0.5	10.0.0.1	MQTT	103	Connect Command
8	0.015362…	10.0.0.1	10.0.0.5	MQTT	70	Connect Ack
10	0.015418…	10.0.0.5	10.0.0.1	MQTT	80	Publish Message [prueba]
11	0.015458…	10.0.0.5	10.0.0.1	MQTT	68	Disconnect Req

Figura 7.9: Mensajes publicador con QoS=0.

- **QoS=1:** La diferencia con los mensajes QoS=0 es que el mensaje *publish* se confirma con un *ack*.

  ```
  $ mosquitto_pub -h 10.0.0.1 -t prueba -q 1 -m hola
  ```

mqtt

	Time	Source	Destination	Protocol	Length	Info
4	0.005467…	10.0.0.5	10.0.0.1	MQTT	103	Connect Command
6	0.006050…	10.0.0.1	10.0.0.5	MQTT	70	Connect Ack
8	0.006109…	10.0.0.5	10.0.0.1	MQTT	82	Publish Message (id=1) [prueba]
9	0.006134…	10.0.0.1	10.0.0.5	MQTT	70	Publish Ack (id=1)
10	0.006171…	10.0.0.5	10.0.0.1	MQTT	68	Disconnect Req

Figura 7.10: Mensajes publicador con QoS=1.

- **QoS=2:** La diferencia con los mensajes QoS=0 es que tras el mensaje *publish* se envían una serie de mensajes *received*, *release* y *complete* para asegurar que el mensaje ha llegado tan solo una vez.

  ```
  $ mosquitto_pub -h 10.0.0.1 -t prueba -q 2 -m hola
  ```

mqtt

	Time	Source	Destination	Protocol	Length	Info
4	0.006238…	10.0.0.5	10.0.0.1	MQTT	103	Connect Command
6	0.006821…	10.0.0.1	10.0.0.5	MQTT	70	Connect Ack
8	0.006880…	10.0.0.5	10.0.0.1	MQTT	82	Publish Message (id=1) [prueba]
9	0.006904…	10.0.0.1	10.0.0.5	MQTT	70	Publish Received (id=1)
10	0.006936…	10.0.0.5	10.0.0.1	MQTT	70	Publish Release (id=1)
11	0.006955…	10.0.0.1	10.0.0.5	MQTT	70	Publish Complete (id=1)
12	0.006987…	10.0.0.5	10.0.0.1	MQTT	68	Disconnect Req

Figura 7.11: Mensajes publicador con QoS=2.

Mensajes de publicación en una suscripción.

- **QoS=0:** El suscriptor manda un mensaje *connect* al bróker y cuando recibe el *ack* manda un *subscribe*, que el bróker vuelve a confirmar con un *ack*. El publicador manda un mensaje *connect* al bróker y cuando recibe el *ack* envía un *publish* con el mensaje que quiere transmitir, que el bróker vuelve a enviar hacia el suscriptor. Cierra la comunicación con un *disconnect*.

  ```
  $ mosquitto_sub -h 10.0.0.1 -t prueba -q 0
  $ mosquitto_pub -h 10.0.0.1 -t prueba -q 0 -m hola
  ```

mqtt

	Time	Source	Destination	Protocol	Length	Info
4	0.004523…	10.0.0.3	10.0.0.1	MQTT	103	Connect Command
6	0.005207…	10.0.0.1	10.0.0.3	MQTT	70	Connect Ack
8	0.005276…	10.0.0.3	10.0.0.1	MQTT	79	Subscribe Request (id=1) [prueba]
9	0.005308…	10.0.0.1	10.0.0.3	MQTT	71	Subscribe Ack (id=1)
14	4.091436…	10.0.0.5	10.0.0.1	MQTT	103	Connect Command
16	4.091969…	10.0.0.1	10.0.0.5	MQTT	70	Connect Ack
18	4.092373…	10.0.0.5	10.0.0.1	MQTT	80	Publish Message [prueba]
19	4.092406…	10.0.0.1	10.0.0.3	MQTT	80	Publish Message [prueba]
21	4.093046…	10.0.0.5	10.0.0.1	MQTT	68	Disconnect Req

Figura 7.12: Mensajes suscriptor y publicador con QoS=0.

- **QoS=1:** La diferencia con los mensajes QoS=0 es que los mensajes *publish* se confirma con un *ack*.

```
$ mosquitto_sub -h 10.0.0.1 -t prueba -q 1
$ mosquitto_pub -h 10.0.0.1 -t prueba -q 1 -m hola
```

mqtt

	Time	Source	Destination	Protocol	Length	Info
4	0.006183…	10.0.0.3	10.0.0.1	MQTT	103	Connect Command
6	0.006778…	10.0.0.1	10.0.0.3	MQTT	70	Connect Ack
8	0.006836…	10.0.0.3	10.0.0.1	MQTT	79	Subscribe Request (id=1) [prueba]
9	0.006863…	10.0.0.1	10.0.0.3	MQTT	71	Subscribe Ack (id=1)
14	4.424974…	10.0.0.5	10.0.0.1	MQTT	103	Connect Command
16	4.425552…	10.0.0.1	10.0.0.5	MQTT	70	Connect Ack
18	4.425611…	10.0.0.5	10.0.0.1	MQTT	82	Publish Message (id=1) [prueba]
19	4.425638…	10.0.0.1	10.0.0.5	MQTT	70	Publish Ack (id=1)
20	4.425652…	10.0.0.1	10.0.0.3	MQTT	82	Publish Message (id=1) [prueba]
22	4.425696…	10.0.0.3	10.0.0.1	MQTT	70	Publish Ack (id=1)
23	4.428031…	10.0.0.5	10.0.0.1	MQTT	68	Disconnect Req

Figura 7.13: Mensajes suscriptor y publicador con QoS=1.

- **QoS=2:** La diferencia con los mensajes QoS=0 es que tras los mensajes *publish* se envían una serie de mensajes *received*, *release* y *complete* para asegurarse de que el mensaje ha llegado tan solo una vez.

```
$ mosquitto_sub -h 10.0.0.1 -t prueba -q 2
$ mosquitto_pub -h 10.0.0.1 -t prueba -q 2 -m hola
```

mqtt

	Time	Source	Destination	Protocol	Length	Info
4	0.004572…	10.0.0.3	10.0.0.1	MQTT	103	Connect Command
6	0.007155…	10.0.0.1	10.0.0.3	MQTT	70	Connect Ack
8	0.007215…	10.0.0.3	10.0.0.1	MQTT	79	Subscribe Request (id=1) [prueba]
9	0.007242…	10.0.0.1	10.0.0.3	MQTT	71	Subscribe Ack (id=1)
14	4.477634…	10.0.0.5	10.0.0.1	MQTT	103	Connect Command
16	4.479404…	10.0.0.1	10.0.0.5	MQTT	70	Connect Ack
18	4.479466…	10.0.0.5	10.0.0.1	MQTT	82	Publish Message (id=1) [prueba]
19	4.479489…	10.0.0.1	10.0.0.5	MQTT	70	Publish Received (id=1)
20	4.479521…	10.0.0.5	10.0.0.1	MQTT	70	Publish Release (id=1)
21	4.479542…	10.0.0.1	10.0.0.5	MQTT	70	Publish Complete (id=1)
22	4.479554…	10.0.0.1	10.0.0.3	MQTT	82	Publish Message (id=1) [prueba]
24	4.479601…	10.0.0.3	10.0.0.1	MQTT	70	Publish Received (id=1)
25	4.479659…	10.0.0.1	10.0.0.3	MQTT	70	Publish Release (id=1)
26	4.480128…	10.0.0.3	10.0.0.1	MQTT	70	Publish Complete (id=1)
27	4.480334…	10.0.0.5	10.0.0.1	MQTT	68	Disconnect Req

Figura 7.14: Mensajes suscriptor y publicador con QoS=2.

7.2. Desarrollo de la red SDN

En esta segunda parte del desarrollo teórico va a crear la red con el controlador SDN OpenDaylight, que más adelantes se encargará de sustituir al bróker de MQTT.

7.2.1. Montaje de red.

El primer paso para poder realizar este apartado es crear de nuevo la red con una topología similar a la del primer apartado pero sobre una red SDN con controlador. Para ello se genera una máquina virtual con Ubuntu 14.04, escogiendo esa versión por su estabilidad

y compatibilidad con el resto de herramientas vistas en el capítulo 5 y se instalan todas ellas. Una vez hecho esto se instala el *software* Eclipse para poder programar el controlador OpenDaylight modificando el código de sus distintos archivos de forma más sencilla.

El proyecto base utilizado en el controlador será «Learning Switch» de «SDNHub_ OpenDaylight_tutorial» [25] en su versión completa con el encaminamiento de capa dos implementado y sobre este proyecto se realizarán modificaciones con el objetivo de analizar el escenario de MQTT en mayor profundidad. Para poner el controlador en funcionamiento debe tener una IP asignada que será 127.0.0.1 en este caso, la IP de *localhost* que viene asignada por defecto. En cuanto a la topología creada por Mininet hay que añadir el controlador al *script* que genera la red como un elemento más indicando su IP para que Mininet lo encuentre.

```
"""Custom topology"""

from mininet.topo import Topo

class MyTopo( Topo ):
    def __init__( self ):
        "Create custom topo."
        # Initialize topology
        Topo.__init__( self )

        # Add controller
        c0=net.addController(name='c0', controller=RemoteController,
            protocols='OpenFlow13', ip='127.0.0.1')

        # Add hosts and switches
        highSwitch = self.addSwitch( 's1' )
        leftSwitch = self.addSwitch( 's2' )
        middleSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
        baHost = self.addHost( 'h1' )
        bbHost = self.addHost( 'h2' )
        caHost = self.addHost( 'h3' )
        cbHost = self.addHost( 'h4' )
        daHost = self.addHost( 'h5' )
        dbHost = self.addHost( 'h6' )

        # Add links
        self.addLink( highSwitch, leftSwitch )
        self.addLink( highSwitch, middleSwitch )
        self.addLink( highSwitch, rightSwitch )
        self.addLink( leftSwitch, baHost )
        self.addLink( leftSwitch, bbHost )
        self.addLink( middleSwitch, caHost )
        self.addLink( middleSwitch, cbHost )
        self.addLink( rightSwitch, daHost )
        self.addLink( rightSwitch, dbHost )

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Una vez completado lo anterior se podría poner la red en funcionamiento con los siguientes pasos:

1. **Iniciar Mininet:** Se inicia la topología de red sobre la que se trabajará.

   ```
   $ sudo -u root mn --topo mytopo --custom custom-topo-alv.py~
     --mac -- arp --switch ovsk,protocols=OpenFlow13 --
     controller remote,ip=127.0.0.1
   ```

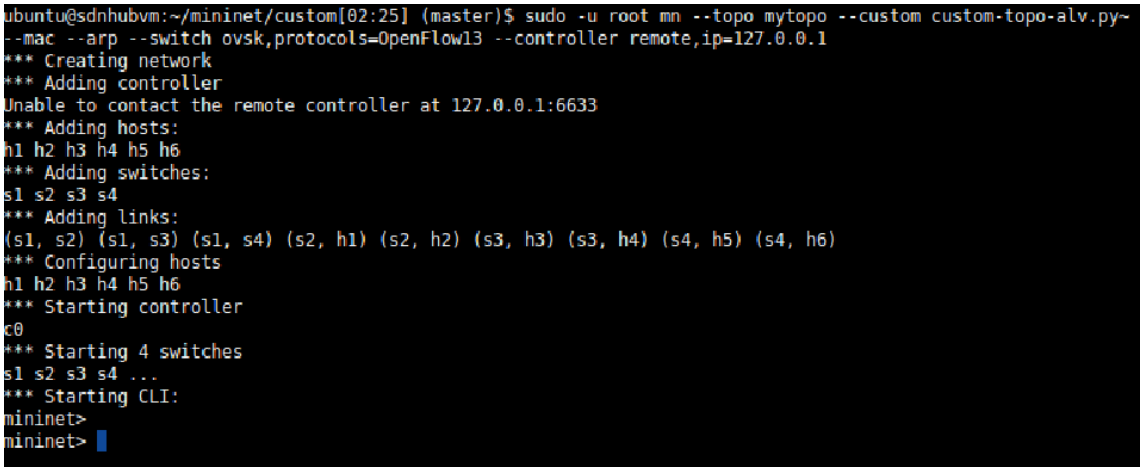


Figura 7.15: Topología Mininet inicializada.

2. **Instalar el proyecto:** Tras entrar en el controlador OpenDaylight se busca el proyecto que se quiere utilizar y se instala.

   ```
   $ feature:install sdnhub-tutorial-learning-switch
   ```

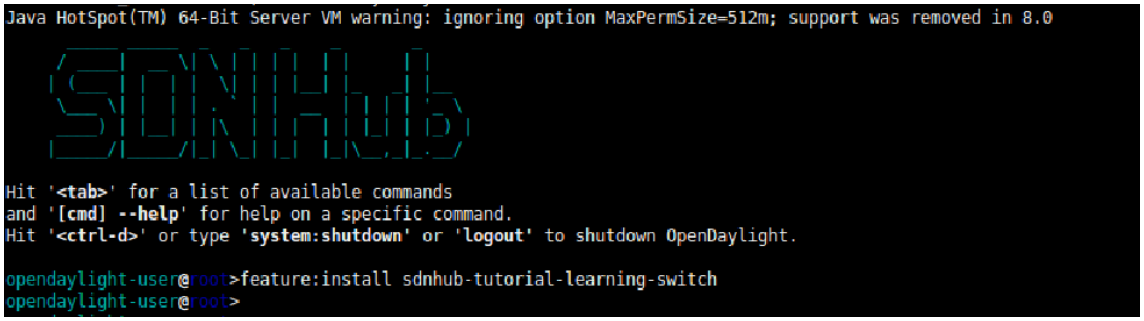


Figura 7.16: Instalación del proyecto en OpenDaylight.

3. **Añadir los flujos:** Con este script se borran los flujos creados en los switch y posteriormente se indica que todos los paquetes se envíen hacia el controlador.

```
1 NO_SWITCHES=$1
2 for i in `seq 1 ${NO_SWITCHES}`
3 do
4     sudo ovs-ofctl del-flows -OOpenFlow13 s$i
5 done
6 for i in `seq 1 ${NO_SWITCHES}`
7 do
8     sudo ovs-ofctl add-flow -OOpenFlow13 s$i priority=65534,
          actions=output:controller
9 done
```

7.2.2. Código del controlador.

En este apartado se van a explicar las principales funcionalidades que incluye el código del proyecto que se está utilizando para poner en funcionamiento el controlador. Para

adaptarlo a una red con MQTT no haría falta realizar ningún cambio pero se incluirán algunos mensajes que impriman información y sirvan para ver el contenido de los paquetes con los que se va a trabajar.

- `onPacketReceived`: Esta es la función por la que pasa un paquete al llegar al controlador. Lo primero que hace es ignorar los paquetes LLDP (*Link Layer Discovery Protocol*), que son paquetes de descubrimiento que no interesa analizar, y comprueba si se tiene que comportar como un *hub*, en cuyo caso mandaría los paquetes como *flood* o inundación por sus interfaces. Si no debe comportarse como un *hub*, significa que se debe implementar *learning switch* creando tablas de flujo que le indiquen al *switch* por qué interfaz enviar el paquete correspondiente. Para ello extrae la dirección MAC, actualiza su tabla y comprueba si se encuentra en ella. Si aparece, crea el flujo y envía el paquete por la interfaz que corresponda y si no realiza un *flood* buscando el camino por el que llegar a su destino e incluirlo en la tabla cuando llegue la respuesta. En la Figura 7.17 se puede ver el flujo de decisiones que toma la función y en el Anexo A.1 se puede ver el código en su primera versión, ya que se modificará en las siguientes secciones.

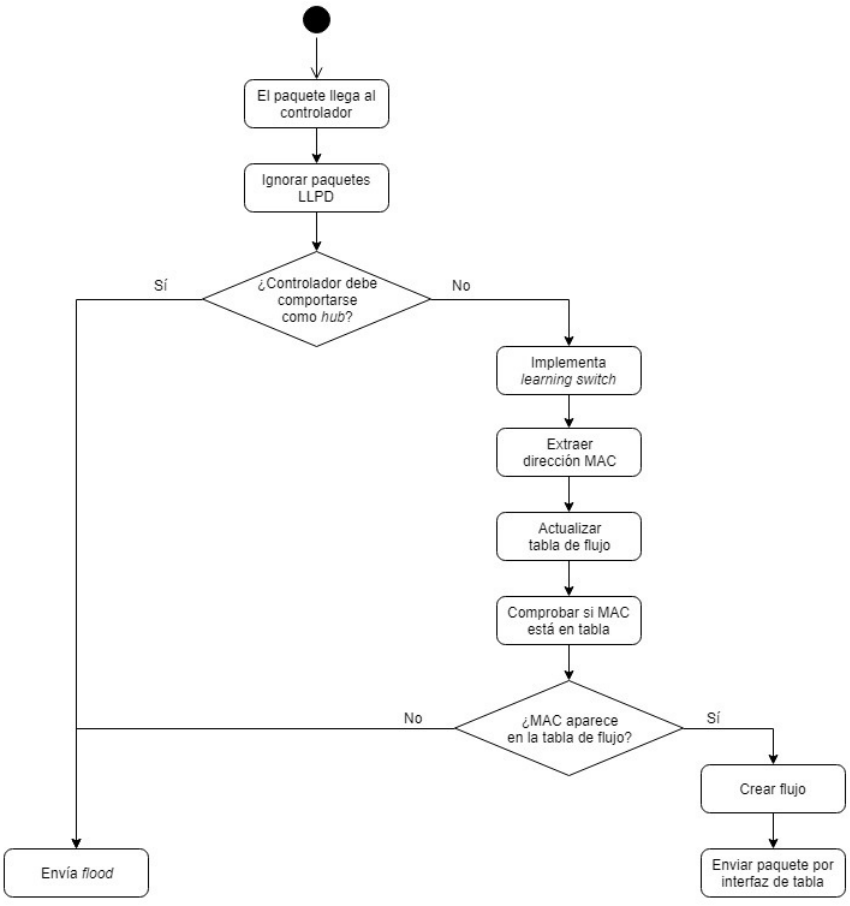


Figura 7.17: Diagrama de flujo de la función `onPacketReceived`.

- `programL2Flow`: Esta función crea un objeto de flujo que tiene identificadores y una lista de instrucciones, lo añade al nodo correspondiente para enviarlo y estos datos se llevan hacia el *switch* donde se programará con los datos del flujo.

- Dijkstra: Es un algoritmo para determinar el camino más corto desde un vértice hasta el resto de vértices, en este caso de una topología.

- Funciones de parseo: En este punto se incluyen todas las funciones útiles para el parseo de los mensajes y que se encuentran en el archivo «*PacketParsingUtils.java*». La mayoría de ellas sirven para extraer un dato concreto de un tipo de paquete o para traducir datos de un tipo a otro, como por ejemplo unos bytes que contengan una MAC a una dirección MAC en decimal separada por «*:*».

Aunque se hayan explicado estas funciones, existen otras muchas que se encargan de complementar a las principales o que no interesan para este proyecto, como la creación de tablas de enrutamiento capa tres o la configuración de grupos de difusión.

7.2.3. Flujo.

Para que en este tipo de redes se establezca una comunicación entre dos elementos debe existir un flujo en las tablas de encaminamiento de sus *switches*. Con el siguiente comando se añade un flujo, con el protocolo OpenFlow en la versión 1.3 para la tabla del switch uno, e indica que todos los mensajes de tipo Ethernet deben enviarse con la máxima prioridad (65534) hacia el controlador. Como ya se explicó en el apartado 5.2.3 los flujos pueden ser proactivos o reactivos.

```
$ sudo ovs-ofctl add-flow -OOpenFlow13 s1 dl_type=0x0806,priority
    =65534,actions=output:controller
```

- **Flujos proactivos:** Se generan manualmente con comandos y se pueden agrupar en *scripts* para facilitar su uso. En este caso, el *script* borra todos los flujos del *switch* con el primer bucle *«for»* y manda todos los mensajes hacia el controlador con el segundo como ya se indicó en la Sección 7.2.1. Tras esto, se añaden como comentario unos flujos con menor prioridad para el *switch* uno («s1») dependiendo de la dirección MAC que tengan, y se indica la interfaz por la que tendrían que salir. Con este tipo de comandos se podrían crear de forma exacta las rutas para llegar desde un *switch* a otro antes de poner en funcionamiento la red.

```
1  NO_SWITCHES=$1
2  for i in `seq 1 ${NO_SWITCHES}`
3  do
4      sudo ovs-ofctl del-flows -OOpenFlow13 s$i
5  done
6  for i in `seq 1 ${NO_SWITCHES}`
7  do
8      sudo ovs-ofctl add-flow -OOpenFlow13 s$i priority=65534,actions=
           output:controller
9      sudo ovs-ofctl add-flow -OOpenFlow13 s1 dl_type=0x0806,priority
           =65534,actions=output:controller
10 #if test "$i" = "1"
11 #then
12 #    sudo ovs-ofctl add-flow -OOpenFlow13 s$i priority=2,dl_dst
       =00:00:00:00:00:64,actions=output:2
13 #    sudo ovs-ofctl add-flow -OOpenFlow13 s$i priority=2,dl_dst
       =00:00:00:00:00:02,actions=output:1
14 #fi
15 done
```

- **Flujos reactivos:** Las rutas entre *switches* se generan de forma automática por el controlador. En la Figura 7.18 se pueden ver las tablas de flujo de los distintos switch según nos muestra el log del controlador, y en la Tabla 7.1 se pueden observar los resultados para el *switch* tres («openflow:3») de forma más clara.

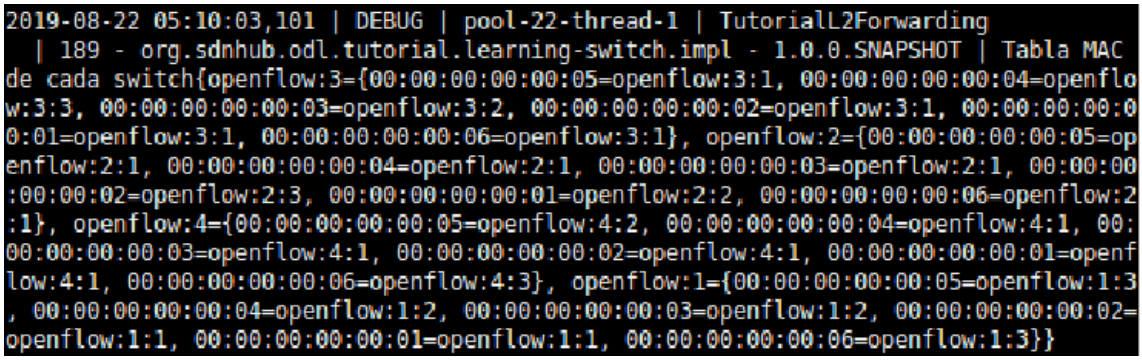


Figura 7.18: Tabla de flujos generada por el controlador.

MAC Destino	Interfaz de envío
00:00:00:00:00:05	1
00:00:00:00:00:04	3
00:00:00:00:00:03	2
00:00:00:00:00:02	1
00:00:00:00:00:01	1
00:00:00:00:00:06	1

Tabla 7.1: Tabla de flujos para «s3».

7.2.4. Revisión de funcionamiento.

En este punto se va a repetir el proceso de la Sección 7.1.3 para estudiar brevemente el intercambio final de los mensajes MQTT en el nuevo escenario. En este caso se va a repetir el ejemplo de una publicación en una suscripción con QoS=0, que es el caso más completo y representativo con la calidad básica que se va a usar para realizar las modificaciones.

Como se puede observar en la Figura 7.19 el intercambio de mensajes es exactamente el mismo, por lo que el comportamiento es el correcto. Cabe destacar que las direcciones IP del suscriptor y publicador han cambiado a 10.0.0.2 y 10.0.0.3, respectivamente.

```
$ mosquitto_sub -h 10.0.0.1 -t prueba -q 0
$ mosquitto_pub -h 10.0.0.1 -t prueba -q 0 -m hola
```

mqtt

	Time	Source	Destination	Protocol	Length	Info
5	1.088951931	10.0.0.2	10.0.0.1	MQTT	105	Connect Command
7	1.089188170	10.0.0.1	10.0.0.2	MQTT	72	Connect Ack
9	1.123684839	10.0.0.2	10.0.0.1	MQTT	81	Subscribe Request
10	1.123890286	10.0.0.1	10.0.0.2	MQTT	73	Subscribe Ack
17	5.539838584	10.0.0.3	10.0.0.1	MQTT	105	Connect Command
19	5.540019460	10.0.0.1	10.0.0.3	MQTT	72	Connect Ack
21	5.712113210	10.0.0.3	10.0.0.1	MQTT	82	Publish Message
22	5.712374452	10.0.0.1	10.0.0.2	MQTT	82	Publish Message
23	5.724791322	10.0.0.3	10.0.0.1	MQTT	70	Disconnect Req

Figura 7.19: Mensajes suscriptor y publicador con QoS=0.

7.3. Eliminación del bróker.

En esta tercera parte del desarrollo práctico se busca el objetivo principal del proyecto: la eliminación del bróker MQTT convencional de la red SDN. Para ello, se sigue haciendo uso del controlador externo OpenDaylight, cuyo trabajo será adquirir las funcionalidades del bróker y hacer que el protocolo MQTT funcione de la misma forma que lo haría sin modificaciones. Se va a utilizar la red ya creada en la sección 7.2.

Lo primero que hay que definir para que el controlador sustituya al bróker es una dirección IP que no pertenezca a ningún equipo. Esta IP «imaginaria» se usará como la del supuesto nuevo bróker, que utilizará el controlador para responder a los clientes de MQTT con ella, y que en este apartado será la 10.0.0.100. Esta dirección debe actuar como la de cualquier dispositivo que forme parte de la red para que el resto sepa que el elemento existe y dónde se encuentra.

Por lo tanto, el controlador debe responder los mensajes de ARP y TCP para que la red funcione con normalidad tras incluir la nueva IP. También se debe encargar de responder los mensajes del propio protocolo MQTT procedentes de los clientes.

En los siguientes puntos se analizarán los mensajes de los protocolos que influyen en el escenario, se crearán los paquetes para el funcionamiento de red y de MQTT y, por último, se verá como el controlador analiza y clasifica los mensajes que recibe.

7.3.1. Análisis de los mensajes

El primer paso para recrear los mensajes de los distintos protocolos es estudiar de forma exhaustiva sus campos y cómo se componen, por lo que se usará de nuevo la herramienta Wireshark.

Se va a analizar el mensaje inicial de cada protocolo con la idea de conocer los campos más significativos o que puedan necesitar ser modificados, aunque puede que haya más de estos, como en el caso de MQTT.

ARP

El protocolo ARP se utiliza para obtener la dirección MAC correspondiente a una IP. Cuando el cliente intente conectarse al bróker pero no conozca su dirección MAC, enviará mensajes ARP *request* en difusión, que serán respondidos por un ARP *reply* con la dirección deseada. En la Figura 7.20 se puede ver un mensaje ARP *request*.

Los campos que pertenecen a la primera línea forman la capa de enlace (capa dos de OSI) del mensaje. Sus primeros bytes dan información sobre el tipo de dirección, envío y longitud, el primer campo subrayado es la dirección MAC del transmisor y el segundo corresponde al tipo de protocolo del mensaje (en este caso ARP).

El resto del mensaje pertenece a ARP donde, tras unos campos de tipos y tamaño de protocolo y *hardware*, se indica en el primer campo señalado el tipo de mensaje (*request* corresponde con el 1) y en los siguientes las direcciones MAC e IP del trasmisor y objetivo, respectivamente. Cabe destacar que la segunda MAC está a cero porque no la conoce.

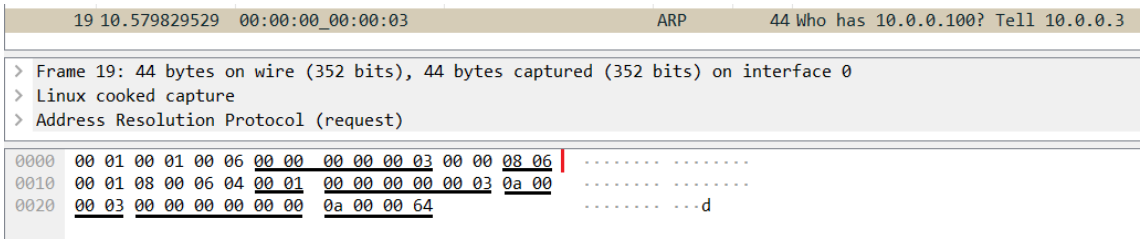


Figura 7.20: Mensaje ARP *request*.

En la Figura 7.21 se detalla el valor de los campos de la trama ARP *request* explicados anteriormente.

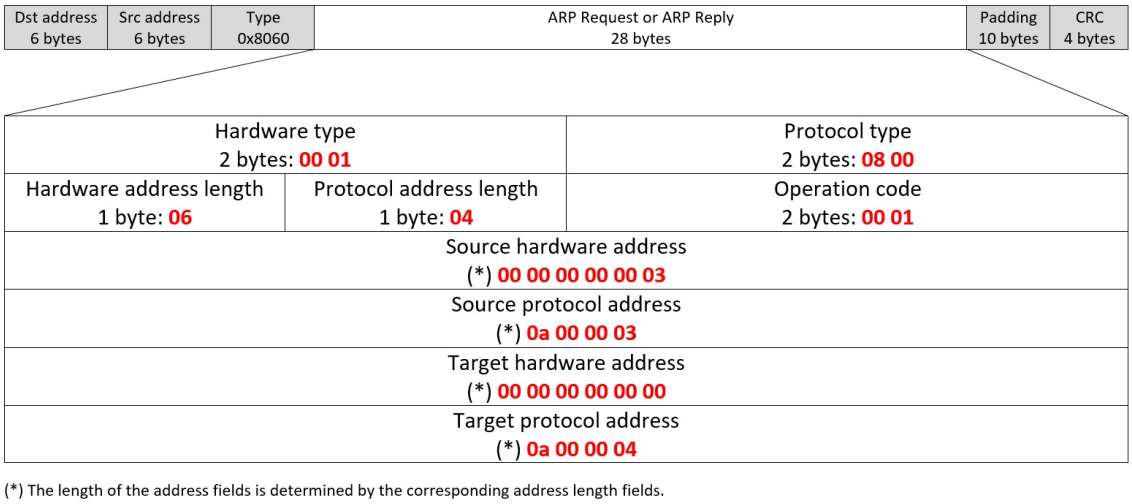


Figura 7.21: Trama con campos de ARP *request*.

TCP

El protocolo TCP es el encargado de crear las conexiones para la transmisión de flujos de datos. Lo hace a través de un TCP *handshake* donde un *host* sincroniza y confirma una conexión con otro elemento de red usando los mensajes SYN, SYN/ACK y ACK, como se representa en la Figura 7.22.

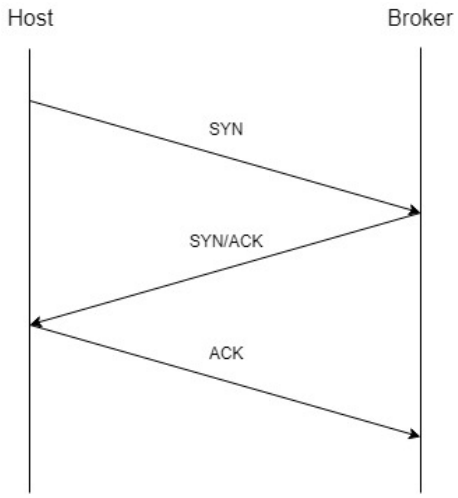


Figura 7.22: Mensaje SYN del TCP *handshake*.

En la Figura 7.23 se observa un ejemplo del TCP *hasndshake*. A continuación se va a analizar el mensaje SYN, teniendo en cuenta que en las tramas también aparecen unos campos de la cabecera del protocolo IP con el que TCP suele trabajar.

La primera línea de bytes pertenece a la capa de enlace ya explicada. En la segunda comienzan los campos IP con unos bytes para indicar longitudes del mensaje, ID (identificación) del mensaje, protocolo siguiente (el 6 indica TCP) o el *checksum* (suma de verificación), entre otros. Los campos destacados son las direcciones IP origen y destino del mensaje.

La última parte del mensaje pertenece al protocolo TCP donde se señalan los puertos usados, los números de secuencia y certificación usados para asegurar que se trata de la misma comunicación, y la bandera TCP que indicará la función del mensaje. Tras esto, finaliza con algunos campos más como el *checksum* del protocolo, la longitud del mensaje o marcas temporales.

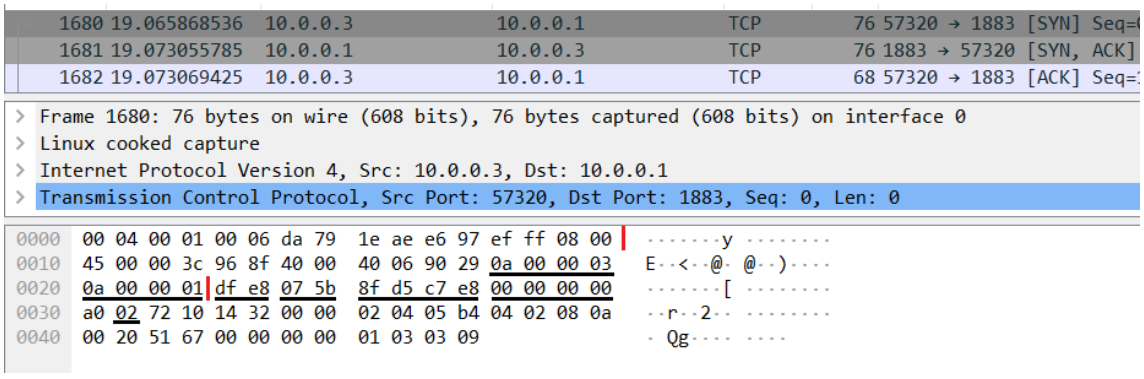


Figura 7.23: Mensaje SYN del TCP *handshake*.

En la Figura 7.24 se detalla el valor de los campos de la trama TCP SYN explicados anteriormente.

Source Port Number 16 bits: **df e8**								Destination Port Number 16 bits: **07 5b**
Sequence Number 32 bits: **8f d5 c7 e8**								
Acknowledgement Number 32 bits: **00 00 00 00**								
Header Length 4 bits: 0111	Reserved 6 bits: 000000 **a0 02**	URG 0	ACK 0	PSH 0	RST 0	SYN 1	PIN 0	Windows Size 16 bits: **00 3a**
TCP Checksum 16 bits: **14 4f**								Urgent Pointer 16 bits: **00 00**
Options (if any)								
Data (if any)								

Figura 7.24: Trama con campos de TCP SYN.

MQTT

Los mensajes de MQTT ya se han analizado en la sección 6.1 así que ahora se van a señalar simplemente los principales campos en los que centrarse al crear una respuesta.

El paquete comienza con los campos de capa de enlace, protocolo IP y TCP. A continuación, en la parte MQTT, el primer byte indica el tipo de mensaje que se trata y el segundo la longitud restante del paquete. La última parte dependerá del tipo de mensaje y, en el caso de *connect,* se incluye el nombre del protocolo, la versión y el identificador del cliente (Figura 6.5).

```
  1683 19.073746358  10.0.0.3          10.0.0.1          MQTT      105 Connect Command
  1685 19.074040487  10.0.0.1          10.0.0.3          MQTT       72 Connect Ack

> Frame 1683: 105 bytes on wire (840 bits), 105 bytes captured (840 bits) on interface 0
> Linux cooked capture
> Internet Protocol Version 4, Src: 10.0.0.3, Dst: 10.0.0.1
> Transmission Control Protocol, Src Port: 57320, Dst Port: 1883, Seq: 1, Ack: 1, Len: 37
> MQ Telemetry Transport Protocol, Connect Command

0000  00 04 00 01 00 06 da 79  1e ae e6 97 ef ff 08 00   .......y ........
0010  45 00 00 59 96 91 40 00  40 06 90 0a 0a 00 00 03   E..Y..@. @.......
0020  0a 00 00 01 df e8 07 5b  8f d5 c7 e9 4b 4d e7 3e   .......[ ....KM.>
0030  80 18 00 3a 14 4f 00 00  01 01 08 0a 00 20 51 69   ...:.O.. ..... Qi
0040  00 20 51 68 10 23 00 04  4d 51 54 54 04 02 00 3c   . Qh.#.. MQTT...<
0050  00 17 6d 6f 73 71 70 75  62 7c 33 39 31 33 2d 6d   ..mosqpu b|3913-m
0060  69 6e 69 6e 65 74 2d 76  6d                       ininet-v m
```

Figura 7.25: Mensajes MQTT *connect.*

7.3.2. Creación de mensajes de red.

Tras analizar la metodología que siguen los paquetes con los que trabaja un bróker, hay que comenzar a crear los mensajes que permitirán la comunicación de un cliente con el controlador de la red. De aquí en adelante todos los mensajes necesitarán dirigirse hacia unas direcciones IP y MAC imaginarias, ya que el supuesto bróker al que se dirigirán los clientes no existe. Las direcciones elegidas son las básicas terminadas en 100, es decir 10.0.0.100 y 00:00:00:00:00:64.

Además de las tramas de Wireshark, se hace uso del *log* que incluye el controlador. Este, tras ser configurado adecuadamente, mostrará toda la información o variables que se incluyan en el código tras el comando «`LOG.debug()`». De esta manera se puede comprobar

en cada momento si se están usando las variables correctas o si se accede a las zonas y funciones esperadas, lo que es muy útil para depurar el código. En la Figura 7.26 se muestra una parte del *log* del controlador que incluye el contenido de los bytes de uno de los mensajes capturados y la IP de origen y destino.

```
2019-08-26 00:53:10,505 | DEBUG | pool-22-thread-1 | TutorialL2Forwarding              | 189 - org.sdnhub
.odl.tutorial.learning-switch.impl - 1.0.0.SNAPSHOT | 111111111111 : 0 0 0 0 0 1 0 0 0 0 0 2 8 0 69 0 0
84 12 182 0 0 64 1 89 241 10 0 0 2 10 0 0 1 0 0 24 140 17 28 0 4 230 143 99 93 0 0 0 0 198 147 7 0 0 0 0
 0 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
2019-08-26 00:53:10,505 | DEBUG | pool-22-thread-1 | TutorialL2Forwarding              | 189 - org.sdnhub
.odl.tutorial.learning-switch.impl - 1.0.0.SNAPSHOT | packet in source ip : 10.0.0.2
2019-08-26 00:53:10,505 | DEBUG | pool-22-thread-1 | TutorialL2Forwarding              | 189 - org.sdnhub
.odl.tutorial.learning-switch.impl - 1.0.0.SNAPSHOT | packet in destination ip : 10.0.0.1
```

Figura 7.26: Fragmento del *log* del controlador.

En los siguientes apartados se explica cómo comprobar si ha llegado el mensaje esperado, generar la respuesta y enviarla por la red. Es necesario generar estos mensajes de red en el proyecto ya que use una IP que no pertenece a un elemento real, independientemente de que el protocolo principal sea MQTT, como en este caso, o no.

ARP REPLY

Este mensaje se enviará cuando el controlador reciba un ARP *request*, que se puede comprobar con una pequeña función que tome el tipo de protocolo y compruebe si es ARP a la dirección de *broadcast*. El siguiente código pertenece a la función `onpacketreceived`, por donde pasan todos los paquetes y se encarga de decidir qué hacer con los ARP *reply*.

```
1 if ((checkARP==1) && (ingressNodeConnectorIdStr != null)){
2     LOG.debug("ARP message");
3     byte[] packet=new byte[42];
4     int intnoPacket = (int) noPacket;
5     packet=PacketParsingUtils.createARPReply(payload, intnoPacket);

6     packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);
7 }
```

Si efectivamente se ha recibido un ARP *request*, se llamará a la función `createARPReply`, que crea el ARP *reply* a partir del ARP *request* recibido. De la primera parte de cabecera de capa dos habría que cambiar la dirección MAC de origen a la 00:00:00:00:00:64, y de la segunda parte habría que incluir sus direcciones IP y MAC e intercambiar el orden entre las del emisor y el objetivo. Por último, el byte que indica el tipo de mensaje ARP habría que cambiarlo de 1 (ARP *request*) a 2 (ARP *reply*). En el anexo A.3 aparece un ejemplo de cómo se generan este tipo de funciones.

En la Figura 7.27 se puede observar uno de los mensajes ARP creados por la función anterior funcionando correctamente.

```
      4 9.664804223   00:00:00_00:00:64                    ARP      44 10.0.0.100 is at 00:00:00:00:00:03
> Frame 4: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface 0
> Linux cooked capture
> [Duplicate IP address detected for 10.0.0.3 (00:00:00:00:00:64) - also in use by 00:00:00:00:00:03 (frame 3)]
> Address Resolution Protocol (reply)
0000  00 00 00 01 00 06 00 00  00 00 00 64 00 00 08 06   ........ ...d....
0010  00 01 08 00 06 04 00 02  00 00 00 00 00 03 0a 00   ........ ........
0020  00 64 00 00 00 00 00 64  0a 00 00 03              .d.....d ....
```

Figura 7.27: Mensaje ARP *reply* creado.

Una vez creado el mensaje ARP *reply*, se enviará por las interfaces correspondientes, es decir, la misma interfaz por la que se envió el mensaje pero en sentido contrario, para que lo reciba el emisor inicial. Un paquete se transmite con la función `packetOut`, pasándole el nodo (*switch*) por donde hay que enviarlo, la interfaz y el paquete en sí.

TCP SYN ACK

Este mensaje se enviará cuando se reciba un TCP SYN. Por lo tanto, en caso de que el campo de tipo de mensaje TCP sea 2, se genera el TCP SYN ACK con la función correspondiente y se envía hacia su destino. El siguiente código es el utilizado para llevar a cabo esas comprobaciones.

```
1  byte typeTCPRaw = payload[47];
2  int typeTCP=PacketParsingUtils.rawToInt(typeTCPRaw);
3  if(typeTCP==2 ){
4      LOG.debug("TCP SYN "+typeTCP);
5      if(ingressNodeConnectorIdStr != null) {
6          byte[] packet=new byte[74];
7          intnoPacket = (int) noPacket;
8          packet=PacketParsingUtils.createTCPAck(payload, intnoPacket);

9          packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);

10     }
11 }
```

La función `createTCPAck` intercambia o añade las direcciones IP y MAC a los campos correspondientes, modifica el identificador del paquete con la variable `noPacket` (contador de los paquetes que han pasado por el controlador) y genera el *checksum* IP. En la parte TCP intercambia los puertos y los números de secuencia y confirmación (sumando uno a este último), disminuye el tamaño de la ventana, genera el *checksum*, incrementa la marca temporal y cambia el valor de TCP *Flag* de 2 a 18 para indicar que se trata de un ACK. En la Figura 7.28 se puede observar uno de estos mensajes creado ya en funcionamiento.

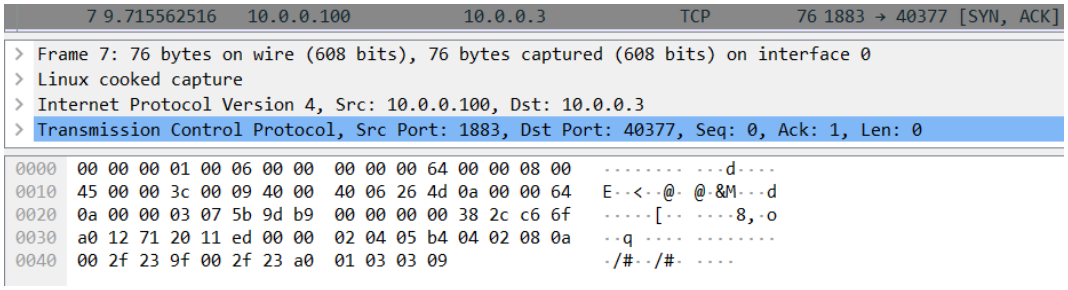


Figura 7.28: Mensaje TCP SYN ACK creado.

Una vez creado el mensaje, se envía por la misma interfaz por la que se recibió el TCP SYN.

TCP FIN ACK

Este mensaje se enviará cuando se reciba otro TCP FIN ACK, correspondiente al valor 17 del tipo TCP que indica que se quiere cerrar la conexión TCP. El paquete enviado será uno del mismo tipo para confirmar el cierre, en caso de que la IP origen no sea la del controlador (terminada en 100), para evitar reenvíos infinitos de estos mensajes. De nuevo, se añade el fragmento de código donde se realizan tales acciones.

```
1  else if(typeTCP==17 && ingressNodeConnectorIdStr != null){
2      LOG.debug("TCP FIN "+typeTCP);
3      byte[] packet=new byte[66];
4      int intnoPacket = (int) noPacket;
5      packet=PacketParsingUtils.createTCPFIN(payload, intnoPacket);

6      if(srcIpRaw[3]!=100){
7          packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);

8      }
9  }
```

La función `createTCPFIN` realiza las mismas modificaciones que las del punto anterior con el TCP SYN ACK, con la diferencia de que el TCP *Flag* debe mantenerse en el valor 17 para indicar el TCP FIN ACK. En la Figura 7.29 aparece un mensaje de este tipo actuando.

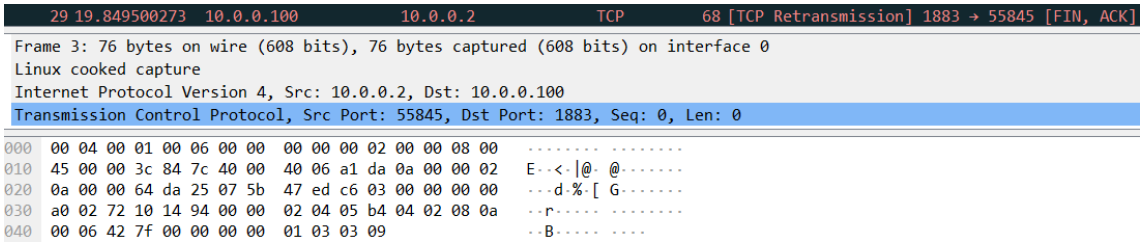


Figura 7.29: Mensaje TCP FIN ACK creado.

Una vez creado el mensaje, se envía por la misma interfaz por la que se recibió el TCP FIN ACK.

7.3.3. Creación de mensajes MQTT.

De la misma forma que en el apartado anterior, ahora se analizará la creación de los mensajes específicos del protocolo MQTT más básicos y necesarios para poner en funcionamiento un escenario con el protocolo. De nuevo, se hace uso del *log* del controlador y de la herramienta Wireshark para asegurar que los pasos hasta llegar al resultado final son los correctos.

CONNACK

Este mensaje se envía cuando llega al controlador un mensaje *connect*, que se comprueba verificando primero que se trata del protocolo MQTT y posteriormente, como se muestra en el siguiente código, si el tipo del protocolo es 16. Si es así, se crea el paquete con la respuesta y se envía por la interfaz correspondiente. También se activan unas variables correspondientes a cada *host*, que se usarán para comprobar si se ha enviado un mensaje *publish* hacia ellos y evitar bucles de reenvío.

```
1  byte typeMQTTRaw = payload[66];
2  int typeMQTT=PacketParsingUtils.rawToInt(typeMQTTRaw);

3  LOG.debug("TYPE MQTT : "+ typeMQTT);
4  if (typeMQTT==16){
5      LOG.debug("MENSAJE CONNECT     "+ typeMQTT);
6      byte[] packet=new byte[70];
7      int intnoPacket = (int) noPacket;
```

```
8       packet=PacketParsingUtils.createConnectAck(payload, intnoPacket);

9       packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);
10
11      active1=1;
12      active2=1;
13      ...
14  }
```

En la función `createConnectAck` se comienza intercambiando las direcciones MAC origen y destino en la primera cabecera. Luego se actualiza el identificador, se calcula el *checksum* y se intercambian las direcciones IP origen y destino en la cabecera IP. En la cabecera TCP se intercambian los puertos y los números de secuencia, y se actualiza la secuencia de comprobación sumando la longitud del paquete recibido (que será de 37 en estos casos). También se actualiza la longitud de ventana, se genera el *checksum* TCP y en la última parte se cambia el tipo de MQTT a 32 (que indica un mensaje *connack*) y la longitud restante a 2.

En el anexo A.3 aparece el código de esta función y en la Figura 7.30 se encuentra una de estas respuestas creadas.

```
    12 9.796389920   10.0.0.100          10.0.0.3            MQTT       72 Connect Ack
▸ Frame 7: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 10.0.0.100, Dst: 10.0.0.3
▸ Transmission Control Protocol, Src Port: 1883, Dst Port: 40377, Seq: 0, Ack: 1, Len: 0

0000  00 00 00 01 00 06 00 00  00 00 00 64 00 00 08 00   ........ ...d....
0010  45 00 00 3c 00 09 40 00  40 06 26 4d 0a 00 00 64   E..<..@. @.&M...d
0020  0a 00 00 03 07 5b 9d b9  00 00 00 00 38 2c c6 6f   .....[.. ....8,.o
0030  a0 12 71 20 11 ed 00 00  02 04 05 b4 04 02 08 0a   ..q ..... ........
0040  00 2f 23 9f 00 2f 23 a0  01 03 03 09               ./#../#. ....
```

Figura 7.30: Mensaje *connack* generado.

SUBACK

Este mensaje se envía cuando llega al controlador un mensaje *subscribe*, cuyo tipo del protocolo MQTT es 130. Si es así, se guardan mediante tres funciones simples el puerto origen y los números de secuencia y de confirmación del mensaje recibido para utilizarlos posteriormente al publicar en este suscriptor. También se almacenan los valores de las direcciones IP y MAC en unas variables y el *topic* al que se suscribe el cliente en un *array*. Después crea el paquete con la respuesta y se envía por la interfaz correspondiente. El siguiente código muestra estas operaciones para el primer *host* (con IP terminada en 1), pero se repetirá para tantos como haya en la red repitiendo el fragmento y modificando el nombre de las distintas variables.

```
1   else if (typeMQTT==130){
2       LOG.debug("MENSAJE SUBSCRIBE      "+ typeMQTT);
3       byte[] packet=new byte[71];
4       int intnoPacket = (int) noPacket;
5
6       if(srcIpRaw[3]==1){
7           packet11=PacketParsingUtils.createSUBSCRIBEAck1(payload);
8           packet21=PacketParsingUtils.createSUBSCRIBEAck2(payload);
9           packet31=PacketParsingUtils.savePort(payload);
10          sourceIPRawPriv1=srcIpRaw;
```

```
11              sourceMacRawPriv1=srcMacRaw;
12              ingressNodeRefFix1=ingressNodeRef;
13              ingressNodeConnectorRefFix1=ingressNodeConnectorRef;
14              int toplen=(int) (payload[71] & 0xFF);
15              topic1=new byte[toplen];
16
17              for(int i=0; i<toplen; i++){
18                  topic1[i]=payload[i+72];
19              }
20
21              packet=PacketParsingUtils.createSUBSCRIBEAck(payload, intnoPacket);
22              packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);
23          }
24          ...
25      }
```

En la función `createSUBSCRIBEAck` se repite el mismo proceso que con el mensaje del apartado anterior. La única diferencia es que se modifica el byte de tipo de mensaje de *susbscribe* (valor 130) por *suback* (valor 144), y la longitud a 3, indicándose en los dos siguientes bytes el mismo identificador que en el mensaje recibido. En la Figura 7.31 se muestra el resultado.

```
    15 6.319447729   10.0.0.100          10.0.0.2            MQTT       73 Subscribe Ack (id=1)

Frame 15: 73 bytes on wire (584 bits), 73 bytes captured (584 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.0.100, Dst: 10.0.0.2
Transmission Control Protocol, Src Port: 1883, Dst Port: 55845, Seq: 5, Ack: 51, Len: 5
MQ Telemetry Transport Protocol, Subscribe Ack

000  00 00 00 01 00 06 00 00  00 00 00 64 00 00 08 00   ........ ...d....
010  45 00 00 39 00 05 40 00  40 06 26 55 0a 00 00 64   E..9..@. @.&U...d
020  0a 00 00 02 07 5b da 25  00 00 00 05 47 ed c6 36   .....[.% ....G..6
030  80 18 00 39 00 c0 00 00  01 01 08 0a 00 06 42 9a   ...9.... ......B.
040  00 06 42 96 90 03 00 01  00                        ..B..... .
```

Figura 7.31: Mensaje *suback* creado.

PUBLISH

Este mensaje se envía cuando llega al controlador otro mensaje *publish*, que se comprueba mirando si el tipo de MQTT es 48. Si es así, se comprueba la longitud del paquete MQTT, se crea el *array* del tamaño adecuado y se guarda el *topic* del mensaje que ha llegado en otro *array*.

Después se comprueba que no sea un *publish* proveniente del controlador y que la variable sigue activa, para controlar los bucles y reenvíos de paquetes, y se compara el *topic* del mensaje que ha llegado con el del suscriptor. Si todo es correcto se crea el nuevo paquete publish, pasándole a la función los valores que se habían guardado a la llegada de un suscriptor. A diferencia del resto de mensajes éste se envía por la interfaz en la que se obtuvo el mensaje de suscripción, que es el cliente interesado en recibir dicho paquete. Por último se pone la variable a 0 y se actualiza el número de secuencia que deberá usar el próximo envío hacia ese suscriptor. Esta operación se repetirá para los distintos *hosts* de la red.

```
1  else if (typeMQTT==48){
2      LOG.debug("MENSAJE PUBLISH     "+ typeMQTT);
3      int res=(int) (payload[67] & 0xFF);
4      byte[] packet=new byte[68+res];
5      int intnoPacket = (int) noPacket;
```

```
6       int toplen=(int) (payload[69] & 0xFF);
7       topic0=new byte[toplen];
8       for(int i=0; i<toplen; i++){
9           topic0[i]=payload[i+70];
10      }
11
12      if(srcIpRaw[3]!=100 && active1==1 && Arrays.equals(topic0, topic1)){
13          packet=PacketParsingUtils.createPUBLISH2(payload, intnoPacket,
                packet11, packet21, packet31, sourceIPRawPriv1, sourceMacRawPriv1
                );
14          packetOut(ingressNodeRefFix1, ingressNodeConnectorRefFix1, packet);

15          active1=0;
16          packet21=PacketParsingUtils.actualizarP2(packet21,res);

17      }
18      ...
19  }
```

Al llamar a la función `createPUBLISH2` se añaden distintos datos del cliente suscriptor al que hay que enviar el mensaje, por lo que además de los cambios que se han ido repitiendo, hay que modificar las direcciones de destino, los números de secuencia y el puerto destino. Ya en la parte de MQTT se mantiene el tipo de mensaje en 48, la longitud y los campos *topic* y el *message* del paquete recibido. También es posible que haya que ajustar la longitud del mensaje ya que en algunas ocasiones los mensajes de *publish* y *disconnect* se envían en un solo paquete.

```
    20 10.662830751  10.0.0.100          10.0.0.2            MQTT       84 Publish Message [aaaaaa]
Frame 20: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.0.100, Dst: 10.0.0.2
Transmission Control Protocol, Src Port: 1883, Dst Port: 55845, Seq: 10, Ack: 51, Len: 16
MQ Telemetry Transport Protocol, Publish Message
000  00 03 00 01 00 06 00 00  00 00 00 64 00 00 08 00   ........ ...d....
010  45 00 00 44 00 12 40 00  40 06 26 3d 0a 00 00 64   E..D..@. @.&=...d
020  0a 00 00 02 07 5b da 25  00 00 00 0a 47 ed c6 36   .....[.% ....G..6
030  80 18 00 39 f9 48 00 00  01 01 08 0a 00 06 46 d6   ...9.H.. ......F.
040  00 06 46 cc 30 0e 00 06  61 61 61 61 61 61 65 65   ..F.0... aaaaaaee
050  65 65 65 65                                        eeee
```

Figura 7.32: Mensaje *publish* generado.

PINGRESP

Este mensaje se envía cuando llega al controlador un mensaje *pingreq*, que se comprueba mirando si el tipo de MQTT es 192. Si es así, se crea el mensaje de respuesta y se envía por la interfaz por la que ha llegado.

```
1   if (typeMQTT==192){
2       LOG.debug("MENSAJE PINGREQ      "+ typeMQTT);
3       byte[] packet=new byte[68];
4       intnoPacket = (int) noPacket;
5       packet=PacketParsingUtils.createPINGResp(payload, intnoPacket);
6       packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);

7   }
```

En la función `createPINGResp` se repite el mismo proceso que en los mensajes anteriores. La única diferencia es que se modifica el byte de tipo de mensaje MQTT de *pingreq*

(valor 192) por *pingresp* (valor 208), y el siguiente byte se deja a 0. En la Figura 7.33 se muestra el resultado.

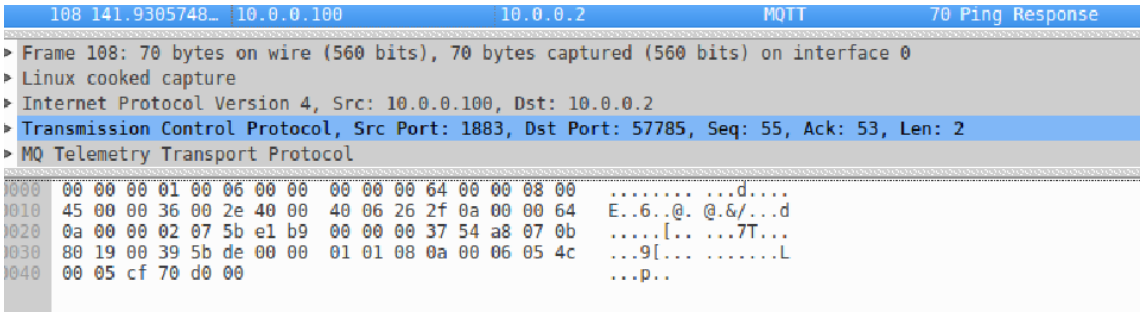


Figura 7.33: Mensaje *pingresp* generado.

7.3.4. Obtención y clasificación de mensajes en el controlador.

En la sección 7.2 la función encargada de obtener los mensajes y hacer que siguieran unas órdenes determinadas era `onPacketReceived`. En este punto esa función seguirá siendo la encargada de ello pero añadiendo numerosas modificaciones para clasificar el mensaje recibido. En el anexo A.2 aparece el código de la función, que se explicará a continuación.

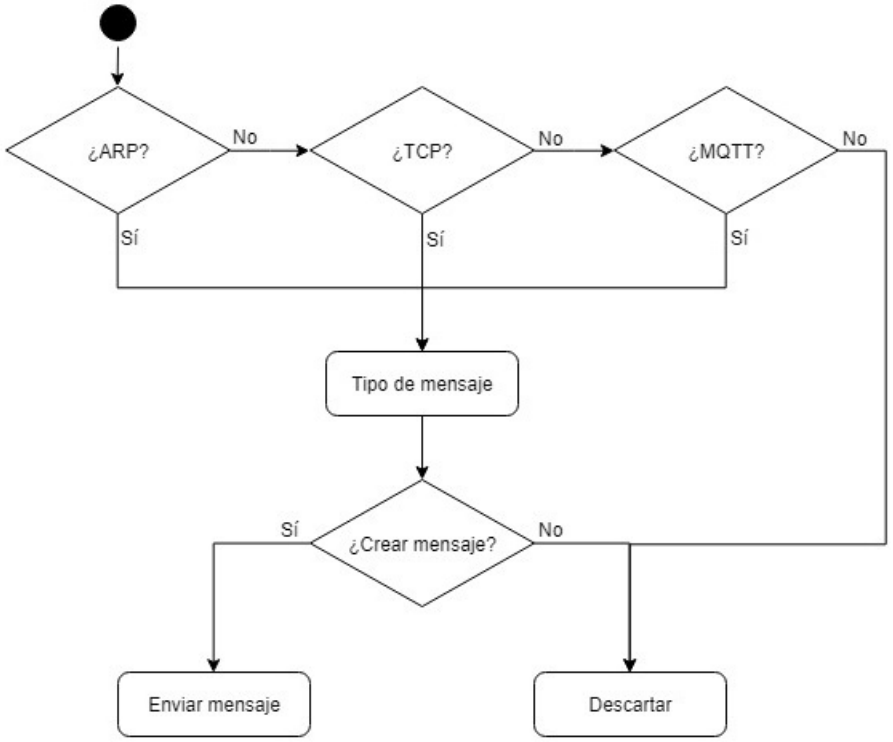


Figura 7.34: Diagrama de flujo de la parte añadida a la función `onPacketReceived`.

El primer cambio respecto a la primera versión se produce en la implementación de *learning switch*, donde se añaden algunas variables y *logs* para obtener más información, como direcciones IP. Al terminar esta parte, se programa la clasificación de los mensajes. Tras mostrar algunas variables, se comprueba si el mensaje recibido es un ARP con el código del apartado 7.3.2 y, si no es el caso, se pasa a comprobar el tipo de mensaje TCP. Si se trata de un mensaje TCP FIN o TCP SYN se pasa a los otros dos apartados

ya explicados en el apartado 7.2.1. Por último, se comprueba si se trata de un mensaje MQTT y, en función del valor de su campo de tipo, se lleva a cabo la acción necesaria, como puede ser responder al mensaje o reenviarlo a otro cliente, tal como se ha visto en el apartado 7.3.3.

Al enviar los mensajes por las interfaces con los datos del controlador usando la función `packetOut` se crean nuevas entradas en las tablas de flujo de los *switches*. Estas entradas junto con las creadas de los flujos del apartado 7.2.3 serán suficientes para hacer que la red siga funcionando con normalidad.

8

Resultados

En este capítulo se estudian los resultados obtenidos tras completar el desarrollo práctico visto en el capítulo 7. Para ello se van a comparar la red final y una red MQTT con bróker habitual, y se verán las posibles diferencias. En estos casos se usará siempre QoS=0, que es la calidad que se ha implementado en el proyecto, y se usará como escenario la red diseñada en la Figura 7.1.

8.1. Funcionamiento con varios publicadores.

El primer caso a comparar será uno donde varios publicadores envíen mensajes a un *topic* en el que habrá un *host* suscrito. El *topic* será «topic1», los publicadores los *host* «h3», «h4» y «h5», y el suscriptor «h2». Esta topología se muestra en la Figura 8.1.

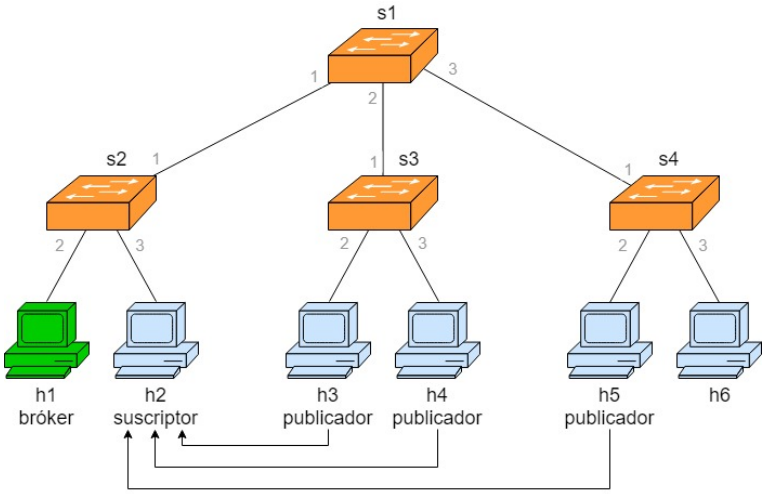


Figura 8.1: Topología con varios publicadores.

Al ser el primero de los casos analizados se añade la Figura 8.2, en la que se muestra el recorrido de los flujos desde que se generan en el publicador hasta llegar al suscriptor. Sin embargo esto no ocurrirá cuando el bróker ya no se encuentre como elemento en la red. Los mensajes en este caso se enviarán por su interfaz hacia el *switch* al que se conectan, y el propio controlador se encargará de modificar lo necesario e introducirlo en la interfaz a la que se conecte el suscriptor interesado en ese *topic*.

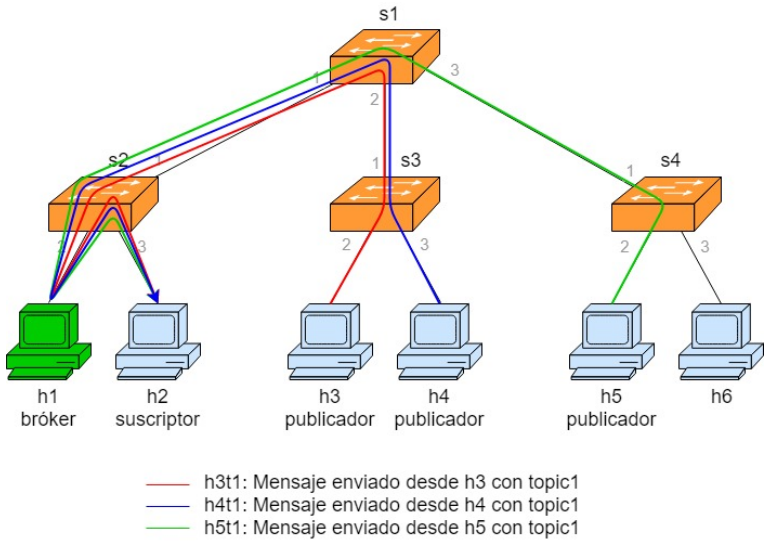


Figura 8.2: Camino seguido por los mensajes.

En el escenario habitual de MQTT se utilizará como bróker al *host* «h1», cuya dirección IP es 10.0.0.1, y que el resto de *hosts* deberán utilizar como IP con la que comunicarse. En la Figura 8.3 se muestra la activación de este bróker.

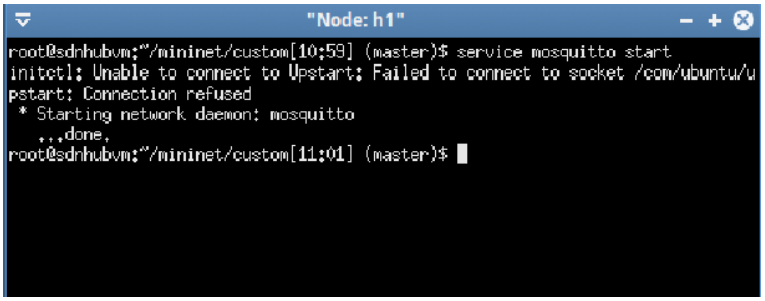


Figura 8.3: Bróker MQTT activado en «h1».

En la Figura 8.4 se pueden ver los distintos mensajes enviados por los publicadores y cómo se visualizan en el suscriptor, tras pasar por el bróker. Todos estos mensajes incluyen un «hola», al que se añade el número correspondiente al *host* que lo envía, y puede incluir más de una vez este número para comprobar que los mensajes siguen llegando sin ser mensajes duplicados de un envío de uno anterior.

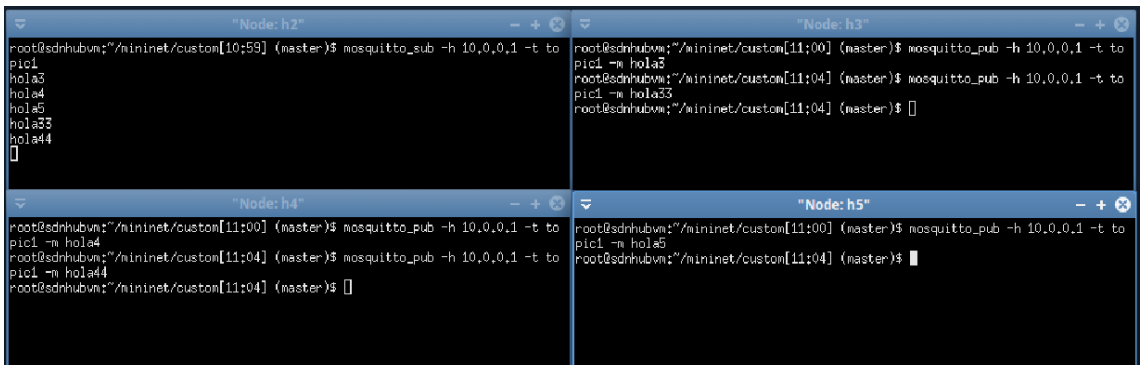


Figura 8.4: Varios publicadores con bróker.

El siguiente escenario es el obtenido con el código del proyecto implementado, manteniendo la topología de la Figura 8.1. En este caso no se activa uno de los *hosts* como bróker, sino que se utiliza una dirección IP ficticia con la que el controlador SDN sabe que tiene que responder esos mensajes. En la Figura 8.5 se puede ver el resultado de repetir exactamente el mismo intercambio de mensajes que en la Figura 8.4, y se comprueba que no hay ninguna variación.

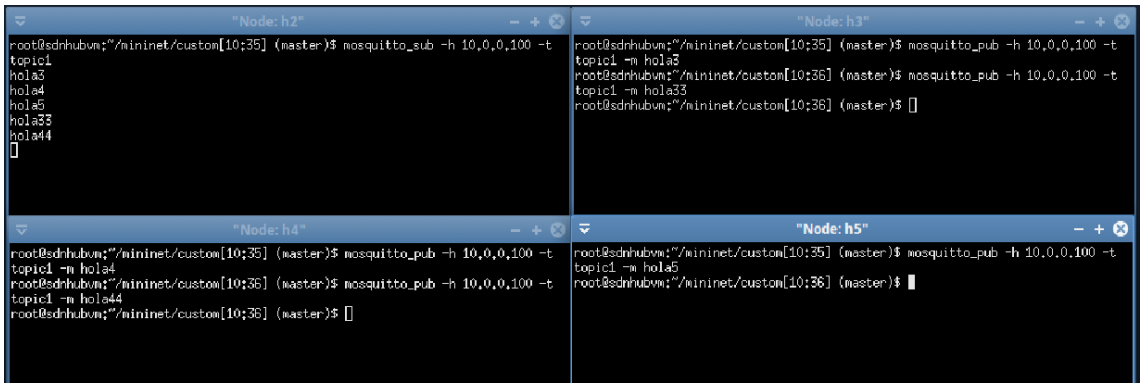


Figura 8.5: Varios publicadores con código generado.

8.2. Funcionamiento con varios suscriptores.

En este apartado se quiere comprobar el funcionamiento de los escenarios con más de un suscriptor conectado a un *topic*. Se repiten las condiciones del anterior, con la diferencia de que el *host* «h3» también estará suscrito al *topic* «topic1». El resultado se puede apreciar en la Figura 8.6.

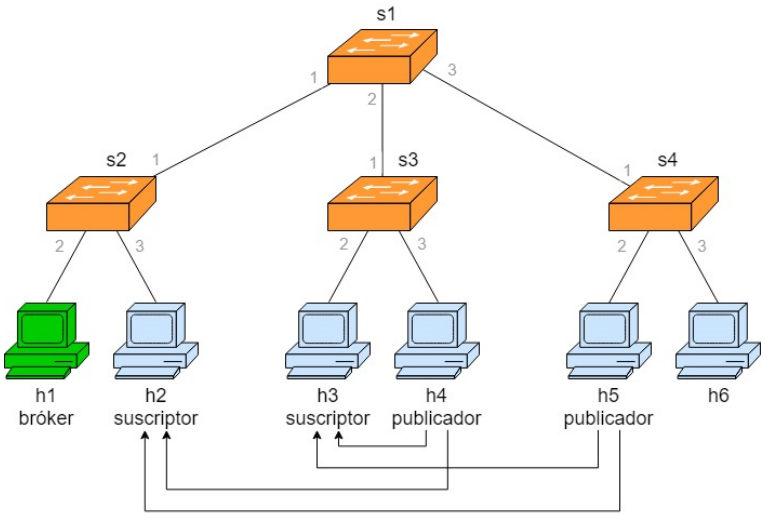


Figura 8.6: Topología con varios suscriptores.

En la Figura 8.7 se ve el resultado del escenario MQTT habitual. La lógica del contenido enviado sigue siendo la misma que la del apartado anterior.

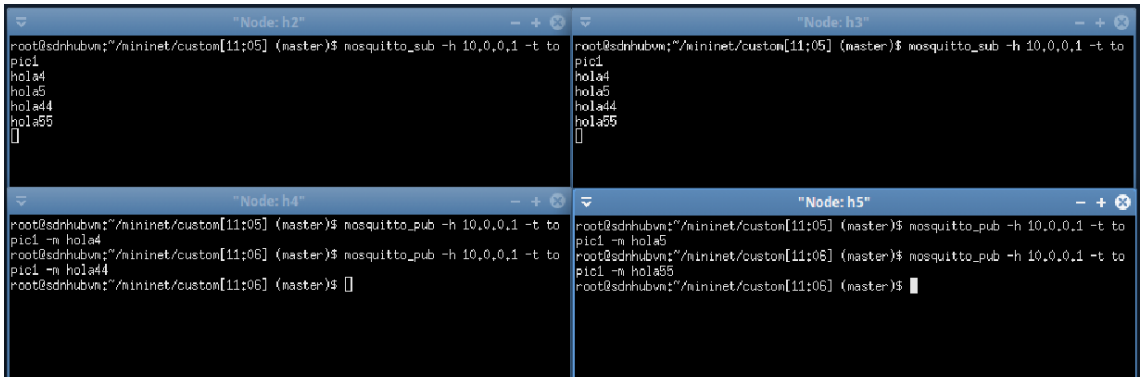


Figura 8.7: Varios suscriptores con bróker.

En la Figura 8.8 se puede observar el resultado en el escenario usando la red SDN con el código generado. Se observa de nuevo que no hay ninguna diferencia entre los casos.

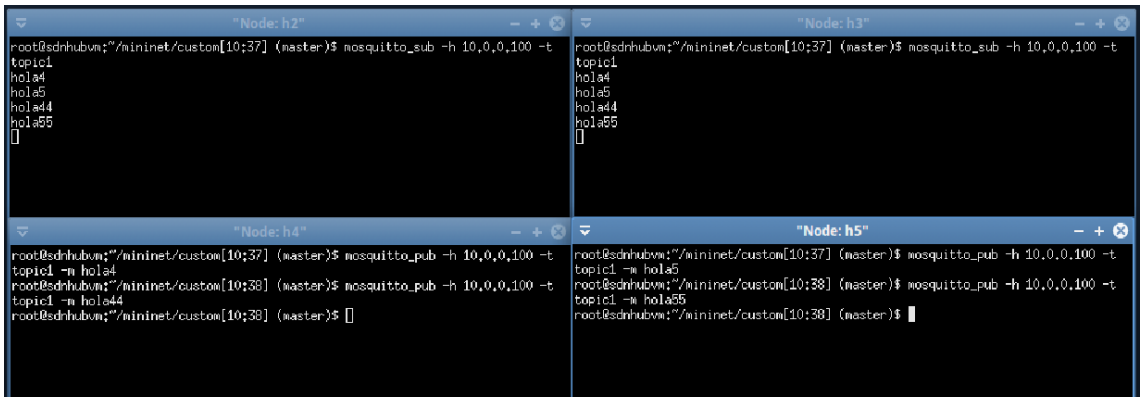


Figura 8.8: Varios suscriptores con código generado.

8.3. Funcionamiento con varios *topics*.

También se va a comprobar lo que ocurriría si se utilizan *topics* distintos. Ahora «h2» está suscrito al «topic1» y «h3» usará «topic2» para publicar. Los dos publicadores enviarán mensajes a ambos *topics*, como se muestra en la topología de la Figura 8.9.

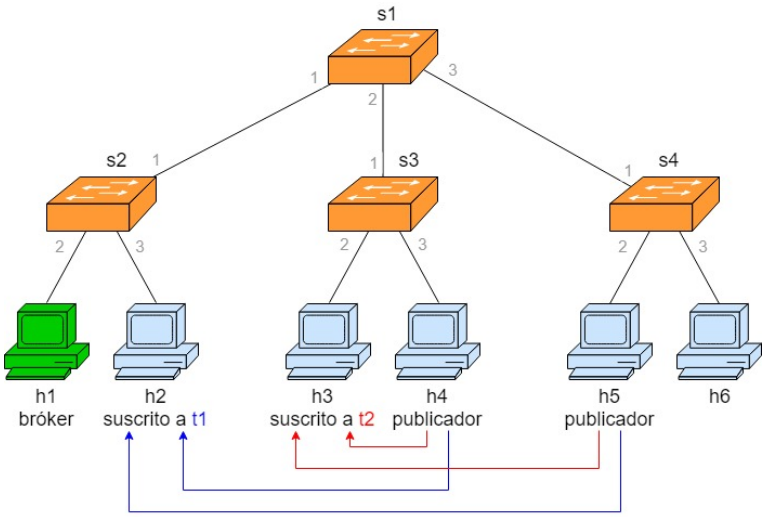


Figura 8.9: Topología con varios *topics*.

En la Figura 8.10 aparece lo obtenido usando un bróker MQTT. El contenido de los mensajes enviados sigue siendo la palabra «hola» seguida del número de *host*, pero ahora se añade con «t1» o «t2» el *topic* hacia el que va dirigido cada mensaje. Los últimos mensajes se envían para comprobar que se puede mandar un mensaje sin que llegue al otro de forma análoga, y no incluyen el «t2» para probar otra longitud de mensaje.

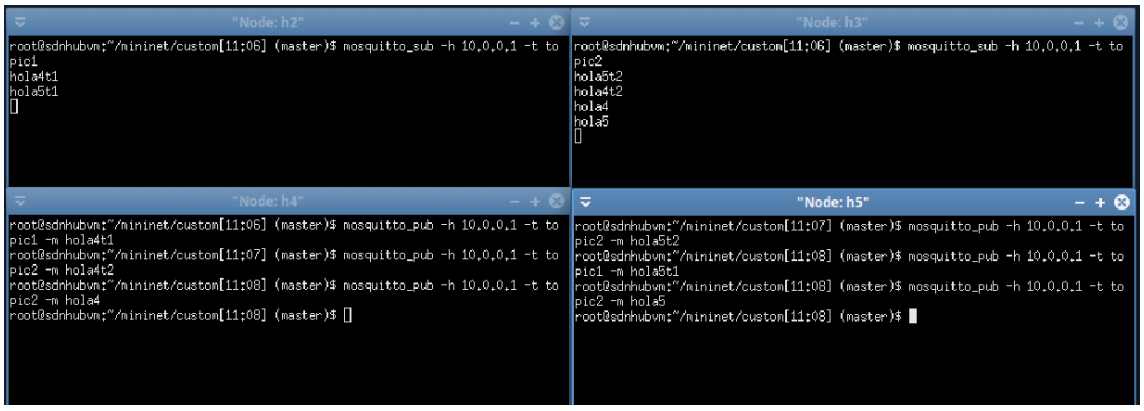


Figura 8.10: Varios *topics* con bróker.

En la Figura 8.11 se utiliza el código del proyecto y de nuevo se obtienen los mismos resultados.

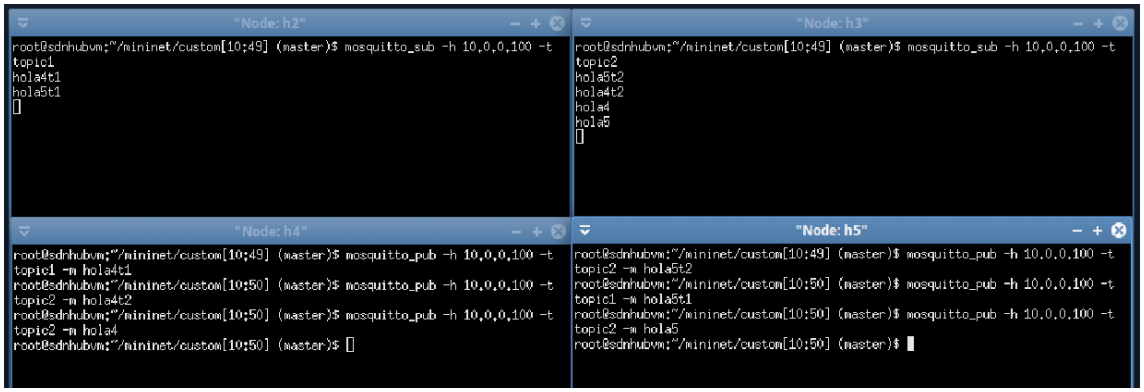


Figura 8.11: Varios *topics* con código generado.

Con estas imágenes también se comprueba que si se introdujera un *topic* erróneo o sin suscriptores el mensaje no llegaría a ningún suscriptor, ya que a cada uno solamente llegan los mensajes publicados en su *topic*.

8.4. Comparativa de tiempos de envío.

Por último se va a realizar una comparativa entre los tiempos de publicación de ambos escenarios. Para ello, tras poner en marcha uno de los escenarios se abrirá la herramienta Wireshark desde la máquina principal, lo que nos permitirá tener acceso a todos los mensajes de la red utilizando la interfaz «any». Una vez se comience a capturar, el publicador enviará mensajes al *topic* del suscriptor, todos con la misma longitud para evitar diferencias de procesamiento. En ambos casos se usará la misma topología de red, con «h2» como suscriptor y «h3» como publicador. En la Figura 8.12 se pueden observar las tramas capturadas de los *publish* al llegar a su destino en el escenario con bróker MQTT.

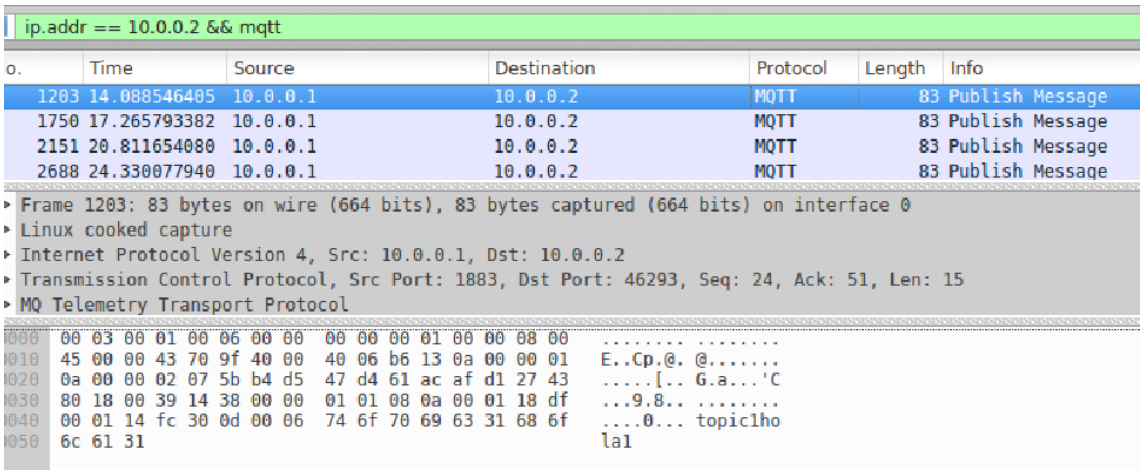


Figura 8.12: Varios *topics* con código generado.

Una vez capturadas las tramas se tiene el tiempo relativo en el que los mensajes fueron enviado desde el publicador y el del momento en que llegaron al suscriptor, como aparece en la Figura 8.12. Con estos tiempos se puede calcular lo que ha tardado el mensaje en llegar a su destino, que será la diferencia entre ambos.

En la tabla 8.1 se muestran los resultados obtenidos de cinco mensajes y la media de sus diferencias en el escenario donde se mantiene el bróker MQTT.

Envío desde el publicador (s)	**Recepción del suscriptor (s)**	**Diferencia (s)**
13.994554	14.088546	0.093992
17.188842	17.265793	0.076951
20.750384	20.811654	0.061269
24.254059	24.330077	0.076018
27.906731	27.969869	0.063137
Media		**0.073375**

Tabla 8.1: Tiempos de retardo con bróker MQTT.

Para el escenario creado con SDN y el código implementado se repite el proceso, obteniendo los resultados de la tabla 8.2.

Envío desde el publicador (s)	**Recepción del suscriptor (s)**	**Diferencia (s)**
14.787896	14.822507	0.034611
18.543312	18.586752	0.043439
21.417600	21.444997	0.027396
24.225604	24.264230	0.038626
28.010285	28.067290	0.057004
Media		**0.039040**

Tabla 8.2: Tiempos de retardo con código implementado.

En la Figura 8.13 se muestra una comparativa entre los resultados de ambos casos. Se puede observar que el tiempo que tarda un mensaje desde que se genera en el publicador hasta llegar al suscriptor con el bróker es siempre mayor al del código creado con este proyecto, y en muchas ocasiones es próximo al doble de retardo. Esta diferencia puede no parecer demasiada al hablar de milisegundo, pero en el ámbito de las comunicaciones son diferencias bastante significativas.

La disminución en los retardos respecto al uso del bróker puede deberse a que, como se ha explicado anteriormente junto a la Figura 8.2, en el caso tradicional de MQTT los mensajes tienen que recorrer toda la red hasta el bróker, y desde ahí volver a enviarse hacia el suscriptor. Sin embargo, con el controlador SDN haciendo esa función, los mensajes al ser enviados por el publicador se modifican y se envían directamente por la interfaz del suscriptor, sin necesidad de moverse por gran parte de la red.

El hecho de que recorrer más nodos en la red aumenta el retardo se puede comprobar con un *ping*. Al realizarlo se observa que el tiempo de respuesta entre «h1» y «h2» (que se encuentran conectados al mismo switch) es de unos 25 ms, mientras que el de h1 con h6 es del orden de los 80 ms.

Esta disminución del retardo también podría deberse al tiempo de procesamiento. El bróker MQTT tradicional debería ser más complejo que la versión sencilla que se ha implementado aquí, y por tanto también más lento.

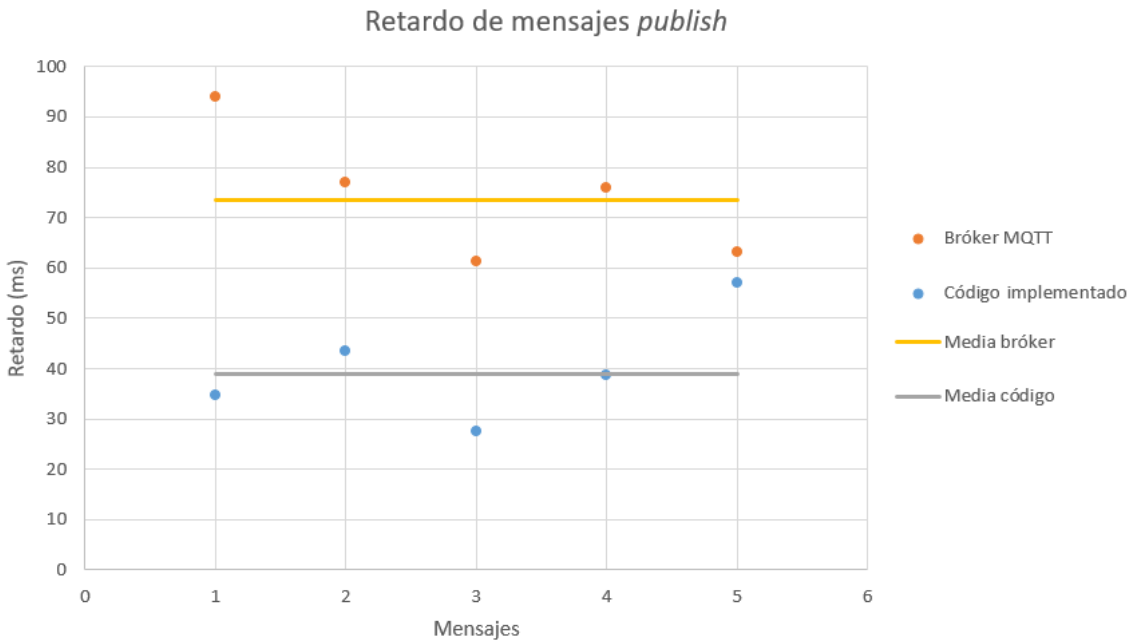


Figura 8.13: Comparativa de retardo en los mensajes *publish*.

9

Conclusiones

9.1. Logros conseguidos.

Se han conseguido los objetivos planteados al comienzo del proyecto, entre los que destacan los siguientes:

- **Estudiar el protocolo MQTT:** se ha realizado un estudio exhaustivo del protocolo MQTT, base fundamental del proyecto y paso necesario para la creación de los mensajes.
- **Crear escenarios de red con Mininet:** inicialmente, con el fin de familiarizarse con la herramienta, se diseñaron escenarios básicos de prueba.
- **Añadir SDN a las redes Mininet:** una vez logrado el funcionamiento y comprensión de los escenarios básicos, se crearon escenarios más complejos con SDN, añadiendo el controlador OpenDaylight. Además de analizar estas dos herramientas se ha aprendido a utilizarlas, a comunicarlas con flujos del protocolo OpenFlow, a crear distintas topologías o a cargar y configurar proyectos en el controlador.
- **Parsear y crear mensajes:** se ha conseguido recibir mensajes de la red diseñada y, una vez comprobado el tipo de protocolo, analizar sus distintos campos para crear artificialmente la respuesta correspondiente.
- **Otorgar al controlador las funcionalidades del bróker:** se ha logrado que la red interactúe con el controlador como si fuera el bróker asignándole una dirección IP virtual que no pertenecía a ningún dispositivo físico. De esta manera los dispositivos interactúan con lo que creen que es otro dispositivo independiente.
- **Creación de un escenario MQTT sin bróker habilitado:** se ha conseguido el que era el objetivo final del proyecto, es decir, un escenario MQTT que funciona con normalidad, para la QoS básica, sin necesidad de habilitar un bróker. Para esto se ha tenido que implementar que trabajo con varios publicadores simultáneamente, varios suscriptores y que cada *topic* reciba su información correspondiente. Además al ser el controlador SDN (que es totalmente programable) el que asume nuevas funciones, aumenta la flexibilidad de la red para modificarla como se necesite.
- **Disminución de tiempos de retardo:** al implementar el código del proyecto se han conseguido disminuir los tiempos de retardo entre los mensajes publicados, como se demuestra en el apartado 8.4.

9.2. Trabajos futuros.

Una vez cumplidos los objetivos de este proyecto surgen opciones con las que podría mejorarse el trabajo actual. A continuación se habla sobre algunas de ellas:

- **Calidades de servicio:** en este proyecto se ha implementado la QoS básica que ofrece el protocolo MQTT. Si quisieran implementarse las nuevas calidades habría que crear y trabajar el resto de mensajes que ofrece el protocolo, además de crear nuevas variables y *arrays* con los que almacenar lo datos y mensajes que se requerirían. Además habría que recrear la lógica de algunas funciones y se tolerarían mucho menos los posibles fallos.

- **Generalización del código:** algunas partes del código se han pensado especialmente para este tipo de redes. Aunque la gran mayoría de las funciones se han generalizado para que actúen correctamente en cualquier situación, algunas de ellas son más complejas y se podrían recrear para que trabajasen con cualquier tamaño de red.

- **Modificaciones específicas:** con este proyecto se ha recreado el funcionamiento típico de MQTT, pero una de las grandes ventajas que se obtienen respecto al bróker habitual es que se puede modificar cualquier cosa como se necesite. Las funciones del bróker se integran en el controlador SDN, y del controlador se tiene dominio absoluto, por lo que se puede modificar la forma de actuar de cada dispositivo de la red.

9.3. Valoración personal.

Desde hace tiempo me interesa bastante el IoT, su desarrollo y sus numerosas aplicaciones porque considero que es un tema muy actual y que, bajo mi punto de vista, tendrá una gran repercusión. El hecho de haber tenido la oportunidad de realizar un proyecto directamente enfocado a este campo me hace sentir bastante afortunado.

La realización de este proyecto ha supuesto un reto muy grande para mí, tanto a nivel académico como personal. La cantidad de horas y esfuerzo dedicado ha superado bastante al que yo esperaba en un principio. Sin embargo, todo ese esfuerzo adicional ha sido proporcional a la satisfacción que sentía al dar otro paso. Al principio del año mi idea era dejar todo el proyecto terminado o casi terminado antes de verano, pero tras numerosos inconvenientes y nuevas obligaciones mi tiempo se vio bastante afectado. Como consecuencia, gran parte del resultado final la he obtenido en estos últimos meses de verano.

Me gustaría destacar en especial la sección 7.3 por su complejidad y tiempo dedicado. Algunas de las dificultades que han surgido se debían a un planteamiento inicial erróneo que, tras estudiarlo mejor y mejorar la implementación, se resolvían. Pero sin duda, han sido los pequeños detalles o fallos humanos lo más laborioso de corregir ya que solo se reconocían tras mucho *troubleshooting* y rigurosas revisiones del código.

Respecto a los conocimientos adquiridos estoy bastante satisfecho porque creo que he aprendido bastante y que me será útil en el futuro, ya que son temas bastante comunes en el ámbito de las telecomunicaciones. Además, como he tenido que utilizar muchas de las competencias asimiladas durante todo el grado, siento que finalizo esta etapa con la teoría mejor asimilada. Me han sido especialmente útiles los conceptos de la rama de telemática, ya que es la especialidad más cercana al proyecto, aunque en general creo que todas han sido de ayuda.

Por último puedo decir que estoy bastante contento de haber elegido este proyecto y de haber conseguido cumplir sus objetivos, y creo que los conocimientos y métodos que he aprendido me serán útiles en mi desarrollo laboral y personal.

Bibliografía

[1] ITU, T. (1993). Telecommunication Standardization Sector of ITU. Annex C: RTP payload format for H, 261, 108-113.

[2] Jazayeri, M., Liang, S., & Huang, C. Y. (2015). Implementation and evaluation of four interoperable open standards for the internet of things. Sensors, 15(9), 24343-24373. https://bit.ly/2WDMd3C

[3] Serpanos, D., & Wolf, M. (2017). «Internet-of-Things (IoT) Systems: Architectures, Algorithms, Methodologies». Springer.. Disponible en https://bit.ly/2FbmJ7B

[4] IOT Analitics. «State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating». Disponible en https://bit.ly/2RsJqJ2

[5] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). «Internet of things: A survey on enabling technologies, protocols, and applications». IEEE communications surveys & tutorials, 17(4), 2347-2376. Disponible en https://bit.ly/2KYNss3

[6] Makkad Asim. «A Survey on Application Layer Protocols for Internet of Things (IoT)». ISSN No. 0976-5697. Disponible en https://bit.ly/2Zzsowj

[7] Tara Salman. «Internet of Things Protocols and Standards». Disponible en https://bit.ly/2WI2PMp

[8] Ramón Millán. «SDN: el futuro de las redes inteligentes». Disponible en https://bit.ly/2lqiT3J

[9] Carlos Spera. «SDN: el futuro de las arquitecturas de red». Disponible en https://bit.ly/2lUB9Cv

[10] Open Networking. «SDN Definition». Disponible en https://bit.ly/2I14YYN

[11] Juniper. «Decoding SDN». Disponible en https://juni.pr/2lsW0N3

[12] Open Networking. «ONF SDN Evolution». Disponible en https://bit.ly/2lokJ50

[13] Park, J. H., Kim, H. S., & Kim, W. T. (2018). «DM-MQTT: An Efficient MQTT Based on SDN Multicast for Massive IoT Communications». Sensors, 18(9), 3071. Disponible en https://bit.ly/2knEJ7w

[14] Shieh, C. S., Yan, J. Y., & Gu, H. X. (2019). «SDN-Based Management Framework for IoT». International Journal of Computer Theory and Engineerin, 11(1). Disponible en https://bit.ly/2lUAwsD

[15] L. Scalzotto, K. Benson, G. Bouloukakis, P. Bellavista, V. Issarny, et al.. «An Implementation Experience with SDN-enabled IoT Data Exchange Middleware». Middleware 2018 - ACM/IFIP/USENIX Middleware conference, Dec 2018, Rennes, France. Disponible en https://bit.ly/2kmzTHO

[16] Bibliografía L. Scalzotto, K. Benson, G. Bouloukakis, P. Bellavista, V. Issarny, et al.. «FireDeX: a Prioritized IoT Data Exchange Middleware for Emergency Response». 19th International Middleware Conference (Middleware 18). ACM, New York, NY, USA, Disponible en https://bit.ly/2kly65N

[17] T. Rausch, S. Nastic, S. Dustdar. «EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications». Disponible en https://bit.ly/2lqSpz2

[18] Página oficial de Mininet. Disponible en https://bit.ly/2lo6JrO

[19] Brian Link. «Set up the Mininet network simulator». Disponible en https://bit.ly/2llMsDn

[20] Página oficial de Mininet. «Introduction to Mininet». Disponible en https://bit.ly/1ku23vc

[21] David Morales. «Tutorial Mininet». Disponible en https://bit.ly/2k9TYkF

[22] John Soban. «How to install OpenDaylight». Disponible en https://bit.ly/2jRzNaQ

[23] Página documentación OpenDaylight. Disponible en https://bit.ly/2knE7yK

[24] Página OpenDaylight. Disponible en https://bit.ly/2eeP9kA

[25] SDN Hub. «OpenDaylight Application Developer's tutorial». Disponible en https://bit.ly/2jWVAOY

[26] Página oficial MQTT. Disponible en https://bit.ly/2Oejbn9

[27] Oasis. Documentación oficial sobre MQTT. Disponible en https://bit.ly/1UVi0ur

[28] Geeky Theory. «¿Qué es MQTT?». Disponible en https://bit.ly/2Xjzibc

[29] 1 Sheeld. «MQTT Protocol». Disponible en https://bit.ly/2lJglgZ

[30] Peter R. Egli. «MQTT». Disponible en https://bit.ly/2XNPKOn

[31] HiveMQ. «MQTT Essentials». Disponible en https://bit.ly/2ZztWq7

[32] Steve I.G. «Understanding the MQTT Protocol Packet Structure». Disponible en https://bit.ly/2kduLpx

[33] Daniel F. Blandón. «OpenFlow: El protocolo del futuro». Disponible en https://bit.ly/2lUxynZ

[34] Search Datacenter. «OpenFlow». Disponible en https://bit.ly/2lqoUO7

[35] Open Networking. «OpenFlow Switch Specification». Disponible en https://bit.ly/2lnWyn4

[36] Carlos Santamara Espinosa. «Protocolos Multicast en redes SDN». Disponible en https://bit.ly/2lTV4kY

[37] Veronica Gonzalez Contreras. «Control de regadío en IoT basado en LoRaWAN».

Anexo A

Código

En este anexo se recogen las principales funciones usadas en el desarrollo del proyecto. La función `onPacketReceived` va evolucionando a lo largo del proyecto, por lo que se adjunta su primera versión en la sección A.1 y la segunda en la A.2. En la sección A.3 se incluye el código de la función `createConnectAck`, a modo de ejemplo de la creación de los mensajes explicada en la sección 7.3.

A.1. Función de paquete recibido (versión 1).

La función `onPacketReceived` es referida a lo largo de todo el capítulo 7 y es una de las principales funcionalidades del controlador. A continuación se añade el código de su primera versión, correspondiente a la fase del desarrollo de la red SDN (sección 7.2), que está explicado en la Figura 7.17.

```
1  public void onPacketReceived(PacketReceived notification) {
2      LOG.trace("Received packet notification {}", notification.getMatch());
3      NodeConnectorRef ingressNodeConnectorRef = notification.getIngress();
4      NodeRef ingressNodeRef = InventoryUtils.getNodeRef(ingressNodeConnectorRef);
5      NodeConnectorId ingressNodeConnectorId = InventoryUtils.getNodeConnectorId(
           ingressNodeConnectorRef);
6      NodeId ingressNodeId = InventoryUtils.getNodeId(ingressNodeConnectorRef);
7
8      // Useful to create it beforehand
9      NodeConnectorId floodNodeConnectorId = InventoryUtils.getNodeConnectorId(
           ingressNodeId, FLOOD_PORT_NUMBER);
10     NodeConnectorRef floodNodeConnectorRef = InventoryUtils.getNodeConnectorRef(
           floodNodeConnectorId);
11
12     /*
13      * Logic:
14      * 0. Ignore LLDP packets and other packets
15      * 1. If behaving as "hub", perform a PACKET_OUT with FLOOD action
16      * 2. Else if behaving as "learning switch",
17      *     2.1. Extract MAC addresses
18      *     2.2. Update MAC table with source MAC address
19      *     2.3. Lookup in MAC table for the target node connector of dst_mac
20      *          2.3.1 If found,
21      *                2.3.1.1 perform FLOW_MOD for that dst_mac through the target
            node connector
22      *                2.3.1.2 perform PACKET_OUT of this packet to target node
            connector
23      *          2.3.2 If not found, perform a PACKET_OUT with FLOOD action
24      */
25
26     //ignore LLDP packets, or you will be in big trouble (0)
27     byte[] etherTypeRaw = PacketParsingUtils.extractEtherType(notification.
           getPayload());
28     int etherType = (0x0000ffff & ByteBuffer.wrap(etherTypeRaw).getShort());
```

```
29      if (etherType == 0x88cc) {
30          return;
31      }
32
33      //ignore IPv6 packets (0)
34      if (etherType == 0x86dd) {
35          return;
36      }
37
38      // HUB IMPLEMENTATION
39      if (function.equals("hub")) {
40          //flood packet (1)
41          packetOut(ingressNodeRef, floodNodeConnectorRef, notification.getPayload())
                ;
42
43      // LEARNING SWITCH IMPLEMENTATION
44      } else {
45          noPacket = noPacket + 1;
46          byte[] payload = notification.getPayload();
47          byte[] dstMacRaw = PacketParsingUtils.extractDstMac(payload);
48          byte[] srcMacRaw = PacketParsingUtils.extractSrcMac(payload);
49
50          //Extract MAC addresses (2.1)
51          String srcMac = PacketParsingUtils.rawMacToString(srcMacRaw);
52          String dstMac = PacketParsingUtils.rawMacToString(dstMacRaw);
53
54          //Strings for ingressNodeId and ingressNodeConnectorId
55          String ingressNodeIdStr = ingressNodeId.getValue();
56          String ingressNodeConnectorIdStr = ingressNodeConnectorId.getValue();
57
58          LOG.debug("JNa>> noPacket {}, switch:port {} -> RECEIVING PACKET of size {}
                 and etherType {} with srcMac {} and dstMac {}", noPacket,
                ingressNodeConnectorIdStr, payload.length, etherType, srcMac, dstMac);
59
60          //Create a table for this switch if it does not exist
61          if(!this.macTablePerSwitch.containsKey(ingressNodeIdStr)) {
62              //LOG.debug("JNa>> noPacket {}, switch:port {} -> Creating MAC table
                    for this switch in the controller", noPacket,
                    ingressNodeConnectorIdStr, ingressNodeIdStr);
63              this.macTablePerSwitch.put(ingressNodeIdStr, new HashMap<String, String
                    >());
64          }
65          //Learn source MAC address (2.2)
66          String previousSrcMacPortStr = this.macTablePerSwitch.get(ingressNodeIdStr)
                .get(srcMac);
67          if ((previousSrcMacPortStr == null) || (!ingressNodeConnectorIdStr.equals(
                previousSrcMacPortStr))) {
68              LOG.debug("JNa>> noPacket {}, switch:port {} ->  ADDING ENTRY for
                    source (MAC {}, port {}) to the MAC table of this switch in the
                    controller (previous port {})", noPacket, ingressNodeConnectorIdStr
                    , srcMac, ingressNodeConnectorIdStr, previousSrcMacPortStr);
69              this.macTablePerSwitch.get(ingressNodeIdStr).put(srcMac,
                    ingressNodeConnectorIdStr);
70              //programL2Flow(ingressNodeId, srcMac, null, ingressNodeConnectorId);
71          } else {
72              LOG.debug("JNa>> noPacket {}, switch:port {} -> ENTRY NOT MODIFIED (Mac
                     {}, port {}) in the MAC table of this switch in the controller",
                    noPacket, ingressNodeConnectorIdStr, srcMac,
                    ingressNodeConnectorIdStr);
73          }
74          //Lookup destination MAC address in table (2.3)
75          String egressNodeConnectorIdStr = this.macTablePerSwitch.get(
                ingressNodeIdStr).get(dstMac);
76          NodeConnectorId egressNodeConnectorId = null;
77          NodeConnectorRef egressNodeConnectorRef = null;
78          if (egressNodeConnectorIdStr != null) {
79              //Entry found (2.3.1)
80              LOG.debug("JNa>> noPacket {}, switch:port {} -> FOUND ENTRY for
                    destination (MAC {}, port {}) in switch this -> PROGRAMMING L2 FLOW
                    ", noPacket, ingressNodeConnectorIdStr, dstMac,
```

```
                egressNodeConnectorIdStr);
81
82              egressNodeConnectorId = new NodeConnectorId(egressNodeConnectorIdStr);
83              egressNodeConnectorRef = InventoryUtils.getNodeConnectorRef(
                    egressNodeConnectorId);
84
85              //Perform FLOW_MOD (2.3.1.1)
86              programL2Flow(ingressNodeId, dstMac, ingressNodeConnectorId,
                    egressNodeConnectorId);
87
88              //Perform PACKET_OUT (2.3.1.2)
89              packetOut(ingressNodeRef, egressNodeConnectorRef, payload);
90          } else {
91              //Flood packet (2.3.2)
92              LOG.debug("JNa>> noPacket {}, switch:port {} -> NOT FOUND ENTRY for
                    destination (Mac {}) in switch {} -> PACKETOUT to FLOODING port",
                    noPacket, ingressNodeConnectorIdStr, dstMac, ingressNodeIdStr);
93
94              //packetOut(ingressNodeRef, floodNodeConnectorRef, payload); //Using
                    packetOut to the flood port -> not working properly!!!
95              floodingPacket(ingressNodeConnectorRef, payload);
96          // FLOODING packets to each port for any topology -> working properly
97          }
98  }
99  }
```

A.2. Función de paquete recibido (versión 2).

En la fase de eliminación del bróker (sección 7.3) se modifica la función `onPacketReceived` A. El siguiente código corresponde por lo tanto a la segunda versión de dicha función, esquematizada en la Figura 7.34.

```
1   public void onPacketReceived(PacketReceived notification) {
2       LOG.trace("Received packet notification {}", notification.getMatch());
3
4       NodeConnectorRef ingressNodeConnectorRef = notification.getIngress();
5       NodeRef ingressNodeRef = InventoryUtils.getNodeRef(ingressNodeConnectorRef);
6       NodeConnectorId ingressNodeConnectorId = InventoryUtils.getNodeConnectorId(
            ingressNodeConnectorRef);
7       NodeId ingressNodeId = InventoryUtils.getNodeId(ingressNodeConnectorRef);
8
9       // Useful to create it beforehand
10      NodeConnectorId floodNodeConnectorId = InventoryUtils.getNodeConnectorId(
            ingressNodeId, FLOOD_PORT_NUMBER);
11      NodeConnectorRef floodNodeConnectorRef = InventoryUtils.getNodeConnectorRef(
            floodNodeConnectorId);
12
13      //ignore LLDP packets, or you will be in big trouble (0)
14      byte[] etherTypeRaw = PacketParsingUtils.extractEtherType(notification.
            getPayload());
15      int etherType = (0x0000ffff & ByteBuffer.wrap(etherTypeRaw).getShort());
16      byte[] payload = notification.getPayload();
17      int checkARP=PacketParsingUtils.checkARP(payload);
18      LOG.debug("ARP?" + checkARP);
19      if (etherType == 0x88cc) {
20          return;
21      }
22
23      //ignore IPv6 packets (0)
24      if (etherType == 0x86dd) {
25          return;
26      }
27
28      // HUB IMPLEMENTATION
29      if (function.equals("hub")) {
30          packetOut(ingressNodeRef, floodNodeConnectorRef, notification.getPayload())
                ;
```

```
31      }
32
33      // LEARNING SWITCH IMPLEMENTATION
34      else {
35          noPacket = noPacket + 1;
36
37          //byte[] payload = notification.getPayload();
38          byte[] dstMacRaw = PacketParsingUtils.extractDstMac(payload);
39          byte[] srcMacRaw = PacketParsingUtils.extractSrcMac(payload);
40          byte[] dstIpRaw = PacketParsingUtils.extractDstIP(payload);
41          byte[] srcIpRaw = PacketParsingUtils.extractSrcIP(payload);
42          byte protoTypeRaw = PacketParsingUtils.extractProtoType(payload);
43          byte igmpTypeRaw = PacketParsingUtils.extractIGMPType(payload);
44          byte[] allRaw = PacketParsingUtils.extractAll(payload);
45          byte[] srcPortRaw = PacketParsingUtils.extractSrcPort(payload);
46          byte[] dstPortRaw = PacketParsingUtils.extractDstPort(payload);
47
48          //Extract MAC addresses (2.1)
49          String srcMac = PacketParsingUtils.rawMacToString(srcMacRaw);
50          String dstMac = PacketParsingUtils.rawMacToString(dstMacRaw);
51          String srcIp = PacketParsingUtils.rawIpToString(srcIpRaw);
52          String dstIp = PacketParsingUtils.rawIpToString(dstIpRaw);
53          String data = PacketParsingUtils.rawDataToString(allRaw);
54          int srcPort=PacketParsingUtils.rawPortToInt(srcPortRaw);
55          int dstPort=PacketParsingUtils.rawPortToInt(dstPortRaw);
56
57          //comments
58          LOG.debug("111111111111 : "+ data);
59          LOG.debug("packet in source ip : " +srcIp);
60          LOG.debug("packet in destination ip : " +dstIp);
61          LOG.debug("packet in source port: " +srcPort);
62          LOG.debug("packet in destination port: " +dstPort);
63          String ingressNodeIdStr = ingressNodeId.getValue();
64          String ingressNodeConnectorIdStr = ingressNodeConnectorId.getValue();
65
66          if(protoTypeRaw != 2) {
67              //Create a table for this switch if it does not exist
68             if(!this.macTablePerSwitch.containsKey(ingressNodeIdStr)) {
69                  LOG.debug("IF TABLE 1111111111 JNa>> noPacket {}, switch:port {} ->
                         Creating MAC table for this switch in the controller",
                        noPacket, ingressNodeConnectorIdStr, ingressNodeIdStr);
70                  macTablePerSwitch.put(ingressNodeIdStr, new HashMap<String, String
                        >());
71              }
72
73              //Create a table for this switch if it does not exist
74              if(!this.macTablePerSwitch.containsKey(ingressNodeIdStr)) {
75                  LOG.debug("IF TABLE 2222222222 JNa>> noPacket {}, switch:port {} ->
                         Creating MAC table for this switch in the controller",
                        noPacket, ingressNodeConnectorIdStr, ingressNodeIdStr);
76                  this.macTablePerSwitch.put(ingressNodeIdStr, new HashMap<String,
                        String>());
77              }
78
79              //Learn source MAC address (2.2)
80              String previousSrcMacPortStr = this.macTablePerSwitch.get(
                    ingressNodeIdStr).get(srcMac);
81              if ((previousSrcMacPortStr == null) || (!ingressNodeConnectorIdStr.
                    equals(previousSrcMacPortStr))) {
82                  LOG.debug("JNa>> noPacket {}, switch:port {} -> ADDING ENTRY for
                        source (MAC {}, port {}) to the MAC table of this switch in the
                         controller (previous port {})", noPacket,
                        ingressNodeConnectorIdStr, srcMac, ingressNodeConnectorIdStr,
                        previousSrcMacPortStr);
83                  this.macTablePerSwitch.get(ingressNodeIdStr).put(srcMac,
                        ingressNodeConnectorIdStr);
84                  LOG.debug("!!!!!!!!!!!!!!!!!!Learn source MAC address (2.2)" +
                        srcMac);
85              } else {
86                  LOG.debug("!!!!!!!!!!!!!!!!else");
```

```
87                  LOG.debug("JNa>> noPacket {}, switch:port {} -> ENTRY NOT MODIFIED
                        (Mac {}, port {}) in the MAC table of this switch in the
                        controller", noPacket, ingressNodeConnectorIdStr, srcMac,
                        ingressNodeConnectorIdStr);
88              }
89
90              //Lookup destination MAC address in table (2.3)
91              String egressNodeConnectorIdStr = this.macTablePerSwitch.get(
                    ingressNodeIdStr).get(dstMac);
92              NodeConnectorId egressNodeConnectorId = null;
93              NodeConnectorRef egressNodeConnectorRef = null;
94              if (egressNodeConnectorIdStr != null) {
95                  //Entry found (2.3.1)
96                  LOG.debug("Entry found (2.3.1) JNa>> noPacket {}, switch:port {} ->
                        FOUND ENTRY for destination (MAC {}, port {}) in switch this
                        -> PROGRAMMING L2 FLOW", noPacket, ingressNodeConnectorIdStr,
                        dstMac, egressNodeConnectorIdStr);
97                  egressNodeConnectorId = new NodeConnectorId(
                        egressNodeConnectorIdStr);
98                  egressNodeConnectorRef = InventoryUtils.getNodeConnectorRef(
                        egressNodeConnectorId);
99
100                 //Perform FLOW_MOD (2.3.1.1)
101                 programL2Flow(ingressNodeId, dstMac, ingressNodeConnectorId,
                        egressNodeConnectorId);
102
103                 //Perform PACKET_OUT (2.3.1.2)
104                 LOG.debug("PACKET OUT");
105                 packetOut(ingressNodeRef, egressNodeConnectorRef, payload);
106             } else {
107                 //Flood packet (2.3.2)
108                 LOG.debug("JNa>> noPacket {}, switch:port {} -> NOT FOUND ENTRY for
                        destination (Mac {}) in switch {} -> PACKETOUT to FLOODING
                        port", noPacket, ingressNodeConnectorIdStr, dstMac,
                        ingressNodeIdStr);
109                 floodingPacket(ingressNodeConnectorRef, payload);
110              // FLOODING packets to each port for any topology -> working properly
111             }
112
113             //Implementación MQTT
114
115             LOG.debug("Tabla MAC de cada switch" + macTablePerSwitch);
116             ingressNodeConnectorId = new NodeConnectorId(ingressNodeConnectorIdStr)
                    ;
117             ingressNodeConnectorRef = InventoryUtils.getNodeConnectorRef(
                    ingressNodeConnectorId);
118             byte checkARPsRaw = payload[15];
119             int checkARPs=PacketParsingUtils.rawToInt(checkARPsRaw);
120             LOG.debug("ingressNodeRef  " + ingressNodeRef);
121             LOG.debug("ingressNodeConnectorRef  " + ingressNodeConnectorRef);
122             LOG.debug("ingressNodeConnectorIdStr  " + ingressNodeConnectorIdStr);
123
124             LOG.debug("ingressNodeIdStr  " + ingressNodeIdStr);
125            int intnoPacket = (int) noPacket;
126            LOG.debug("intnoPacket " + intnoPacket);
127            if ((checkARP==1) && (ingressNodeConnectorIdStr != null)){
128                LOG.debug("ARP message");
129                byte[] packet=new byte[42];
130             intnoPacket = (int) noPacket;
131             packet=PacketParsingUtils.createARPAck(payload, intnoPacket);
132             packetOut(ingressNodeRef, ingressNodeConnectorRef, packet);
133            }else if(checkARPs==1){
134                 LOG.debug("checkARPs ACK" + checkARPs);
135                 }else {
136                     byte typeTCPRaw = payload[47];
137                     int typeTCP=PacketParsingUtils.rawToInt(typeTCPRaw);
138                     LOG.debug("Type TCP    "+ typeTCP);
139                     if(typeTCP==16 ){
140                     LOG.debug("TCP ACK    "+ typeTCP);
141                     }else if(typeTCP==4 || typeTCP==20 ){
```

```
142                     LOG.debug("TCP FUERA     "+ typeTCP);
143                     }else if(typeTCP==17 ){
144                         LOG.debug("TCP FIN     "+ typeTCP);
145
146                     if(ingressNodeConnectorIdStr != null) {
147                             byte[] packet=new byte[66];
148                             intnoPacket = (int) noPacket;
149                             packet=PacketParsingUtils.createTCPFIN(payload,
                                    intnoPacket);
150
151                             if(srcIpRaw[3]!=100){
152                                 packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                        packet);
153                             }
154                         }
155                     }else if (((dstPort==1883) || (srcPort==1883))){
156                         byte typeMQTTRaw = payload[66];
157                         int typeMQTT=PacketParsingUtils.rawToInt(typeMQTTRaw);
158                         LOG.debug("TYPE MQTT : "+ typeMQTT);
159                         if (typeMQTT==0){
160                             LOG.debug("TCP     "+ typeMQTT);
161                         if(typeTCP==2 ){
162                         LOG.debug("TCP SYN     "+ typeTCP);
163
164                             if(ingressNodeConnectorIdStr != null) {
165                              byte[] packet=new byte[74];
166                              intnoPacket = (int) noPacket;
167                              packet=PacketParsingUtils.createTCPAck(payload,
                                     intnoPacket);
168                              packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                     packet);
169                             }
170                         }else if(typeTCP==18 ){
171                             LOG.debug("TCP SYN ACK     "+ typeTCP);
172                         }else if(typeTCP==16 ){
173                             LOG.debug("TCP ACK     "+ typeTCP);
174                         }else{
175                             LOG.debug("Type TCP?     "+ typeTCP);
176                          }
177                     }
178                     else if (typeMQTT==16){
179                          LOG.debug("MENSAJE CONNECT     "+ typeMQTT);
180                          LOG.debug("noPacket : "+ noPacket);
181                          byte[] packet=new byte[70];
182                          intnoPacket = (int) noPacket;
183                          packet=PacketParsingUtils.createConnectAck(payload,
                                 intnoPacket);
184                          packetOut(ingressNodeRef, ingressNodeConnectorRef, packet)
                                 ;
185                          active1=1;
186                          active2=1;
187                          active3=1;
188                          active4=1;
189                          active5=1;
190                          active6=1;
191                     }else if (typeMQTT==32){
192                          LOG.debug("MENSAJE CONNACK     "+ typeMQTT);
193                     }else if (typeMQTT==48){
194                          LOG.debug("MENSAJE PUBLISH     "+ typeMQTT);
195                          LOG.debug("noPacket : "+ noPacket);
196                          int res=(int) (payload[67] & 0xFF);
197                          byte[] packet=new byte[68+res];
198                          //byte[] packet=new byte[82];
199                          intnoPacket = (int) noPacket;
200                          int toplen=(int) (payload[69] & 0xFF);
201                          topic0=new byte[toplen];
202                          for(int i=0; i<toplen; i++){
203                              topic0[i]=payload[i+70];
204                          }
205                          byte[] packetack=new byte[66];
```

```
206                     packetack=PacketParsingUtils.createTCPACKSimple(payload,
                            intnoPacket);
207                     packetOut(ingressNodeRef, ingressNodeConnectorRef,
                            packetack);
208                     if(srcIpRaw[3]!=100 && active1==1 && Arrays.equals(topic0,
                            topic1)){
209                         packet=PacketParsingUtils.createPUBLISH2(payload,
                                intnoPacket, packet11, packet21, packet31,
                                sourceIPRawPriv1, sourceMacRawPriv1);
210                     packetOut(ingressNodeRefFix1, ingressNodeConnectorRefFix1,
                            packet);
211                         contadorpub++;
212                         active1=0;
213                         packet21=PacketParsingUtils.actualizarP2(packet21,res)
                                ;
214                     }
215                     if(srcIpRaw[3]!=100 && active2==1 && Arrays.equals(topic0,
                            topic2)){
216                         packet=PacketParsingUtils.createPUBLISH2(payload,
                                intnoPacket, packet12, packet22, packet32,
                                sourceIPRawPriv2, sourceMacRawPriv2);
217                     packetOut(ingressNodeRefFix2, ingressNodeConnectorRefFix2,
                            packet);
218                         contadorpub++;
219                         active2=0;
220                         packet22=PacketParsingUtils.actualizarP2(packet22,res)
                                ;
221                     }
222                     if(srcIpRaw[3]!=100 && active3==1 && Arrays.equals(topic0,
                            topic3)){
223                         packet=PacketParsingUtils.createPUBLISH2(payload,
                                intnoPacket, packet13, packet23, packet33,
                                sourceIPRawPriv3, sourceMacRawPriv3);
224                     packetOut(ingressNodeRefFix3, ingressNodeConnectorRefFix3,
                            packet);
225                         contadorpub++;
226                         active3=0;
227                         packet23=PacketParsingUtils.actualizarP2(packet23,res)
                                ;
228                     }
229                     if(srcIpRaw[3]!=100 && active4==1 && Arrays.equals(topic0,
                            topic4)){
230                         packet=PacketParsingUtils.createPUBLISH2(payload,
                                intnoPacket, packet14, packet24, packet34,
                                sourceIPRawPriv4, sourceMacRawPriv4);
231                     packetOut(ingressNodeRefFix4, ingressNodeConnectorRefFix4,
                            packet);
232                         contadorpub++;
233                         active4=0;
234                         packet24=PacketParsingUtils.actualizarP2(packet24,res)
                                ;
235                     }
236                     if(srcIpRaw[3]!=100 && active5==1 && Arrays.equals(topic0,
                            topic5)){
237                         packet=PacketParsingUtils.createPUBLISH2(payload,
                                intnoPacket, packet15, packet25, packet35,
                                sourceIPRawPriv5, sourceMacRawPriv5);
238                     packetOut(ingressNodeRefFix5, ingressNodeConnectorRefFix5,
                            packet);
239                         contadorpub++;
240                         active5=0;
241                         packet25=PacketParsingUtils.actualizarP2(packet25,res)
                                ;
242                     }
243                     if(srcIpRaw[3]!=100 && active6==1 && Arrays.equals(topic0,
                            topic6)){
244                         packet=PacketParsingUtils.createPUBLISH2(payload,
                                intnoPacket, packet16, packet26, packet36,
                                sourceIPRawPriv6, sourceMacRawPriv6);
```

```
245                         packetOut(ingressNodeRefFix6, ingressNodeConnectorRefFix6,
                                packet);
246                             contadorpub++;
247                             active6=0;
248                             packet26=PacketParsingUtils.actualizarP2(packet26,res)
                                    ;
249                         }
250                         LOG.debug("contadorpub: "+ contadorpub);
251                     }else if (typeMQTT==56){
252                         LOG.debug("MENSAJE PUBLISH DUP FLAG   "+ typeMQTT);
253                     }
254                     //Falta QoS
255                     else if (typeMQTT==64){
256                         LOG.debug("MENSAJE PUBACK     "+ typeMQTT);
257                     }else if (typeMQTT==80){
258                         LOG.debug("MENSAJE PUBREC     "+ typeMQTT);
259                     }else if (typeMQTT==98){
260                         LOG.debug("MENSAJE PUBREL     "+ typeMQTT);
261                     }else if (typeMQTT==112){
262                         LOG.debug("MENSAJE PUBCOMP     "+ typeMQTT);
263                     }else if (typeMQTT==130){
264                         LOG.debug("MENSAJE SUBSCRIBE     "+ typeMQTT);
265                         LOG.debug("noPacket : "+ noPacket);
266                         byte[] packet=new byte[71];
267                         intnoPacket = (int) noPacket;
268                         if(srcIpRaw[3]==1){
269                             packet11=PacketParsingUtils.createSUBSCRIBEAck1(
                                    payload);
270                             packet21=PacketParsingUtils.createSUBSCRIBEAck2(
                                    payload);
271                             packet31=PacketParsingUtils.savePort(payload);
272                             sourceIPRawPriv1=srcIpRaw;
273                             sourceMacRawPriv1=srcMacRaw;
274                             ingressNodeRefFix1=ingressNodeRef;
275                             ingressNodeConnectorRefFix1=ingressNodeConnectorRef;
276                             int toplen=(int) (payload[71] & 0xFF);
277                             topic1=new byte[toplen];
278                             for(int i=0; i<toplen; i++){
279                                 topic1[i]=payload[i+72];
280                             }
281                             packet=PacketParsingUtils.createSUBSCRIBEAck(payload,
                                    intnoPacket);
282                             packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                    packet);
283                         }else if(srcIpRaw[3]==2){
284                             packet12=PacketParsingUtils.createSUBSCRIBEAck1(
                                    payload);
285                             packet22=PacketParsingUtils.createSUBSCRIBEAck2(
                                    payload);
286                             packet32=PacketParsingUtils.savePort(payload);
287                             sourceIPRawPriv2=srcIpRaw;
288                             sourceMacRawPriv2=srcMacRaw;
289                             ingressNodeRefFix2=ingressNodeRef;
290                             ingressNodeConnectorRefFix2=ingressNodeConnectorRef;
291                             int toplen=(int) (payload[71] & 0xFF);
292                             topic2=new byte[toplen];
293                             for(int i=0; i<toplen; i++){
294                                 topic2[i]=payload[i+72];
295                             }
296                             packet=PacketParsingUtils.createSUBSCRIBEAck(payload,
                                    intnoPacket);
297                             packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                    packet);
298                         }else if(srcIpRaw[3]==3){
299                             packet13=PacketParsingUtils.createSUBSCRIBEAck1(
                                    payload);
300                             packet23=PacketParsingUtils.createSUBSCRIBEAck2(
                                    payload);
301                             packet33=PacketParsingUtils.savePort(payload);
302                             sourceIPRawPriv3=srcIpRaw;
```

```
303                         sourceMacRawPriv3=srcMacRaw;
304                         ingressNodeRefFix3=ingressNodeRef;
305                         ingressNodeConnectorRefFix3=ingressNodeConnectorRef;
306                         int toplen=(int) (payload[71] & 0xFF);
307                         topic3=new byte[toplen];
308                         for(int i=0; i<toplen; i++){
309                             topic3[i]=payload[i+72];
310                         }
311                         packet=PacketParsingUtils.createSUBSCRIBEAck(payload,
                                intnoPacket);
312                         packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                packet);
313                     }else if(srcIpRaw[3]==4){
314                         packet14=PacketParsingUtils.createSUBSCRIBEAck1(
                                payload);
315                         packet24=PacketParsingUtils.createSUBSCRIBEAck2(
                                payload);
316                         packet34=PacketParsingUtils.savePort(payload);
317                         sourceIPRawPriv4=srcIpRaw;
318                         sourceMacRawPriv4=srcMacRaw;
319                         ingressNodeRefFix4=ingressNodeRef;
320                         ingressNodeConnectorRefFix4=ingressNodeConnectorRef;
321                         int toplen=(int) (payload[71] & 0xFF);
322                         topic4=new byte[toplen];
323                         for(int i=0; i<toplen; i++){
324                             topic4[i]=payload[i+72];
325                         }
326                         packet=PacketParsingUtils.createSUBSCRIBEAck(payload,
                                intnoPacket);
327                         packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                packet);
328                     }else if(srcIpRaw[3]==5){
329                         packet15=PacketParsingUtils.createSUBSCRIBEAck1(
                                payload);
330                         packet25=PacketParsingUtils.createSUBSCRIBEAck2(
                                payload);
331                         packet35=PacketParsingUtils.savePort(payload);
332                         sourceIPRawPriv5=srcIpRaw;
333                         sourceMacRawPriv5=srcMacRaw;
334                         ingressNodeRefFix5=ingressNodeRef;
335                         ingressNodeConnectorRefFix5=ingressNodeConnectorRef;
336                         int toplen=(int) (payload[71] & 0xFF);
337                         topic5=new byte[toplen];
338                         for(int i=0; i<toplen; i++){
339                             topic5[i]=payload[i+72];
340                         }
341                         packet=PacketParsingUtils.createSUBSCRIBEAck(payload,
                                intnoPacket);
342                         packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                packet);
343                     }else if(srcIpRaw[3]==6){
344                         packet16=PacketParsingUtils.createSUBSCRIBEAck1(
                                payload);
345                         packet26=PacketParsingUtils.createSUBSCRIBEAck2(
                                payload);
346                         packet36=PacketParsingUtils.savePort(payload);
347                         sourceIPRawPriv6=srcIpRaw;
348                         sourceMacRawPriv6=srcMacRaw;
349                         ingressNodeRefFix6=ingressNodeRef;
350                         ingressNodeConnectorRefFix6=ingressNodeConnectorRef;
351                         int toplen=(int) (payload[71] & 0xFF);
352                         topic6=new byte[toplen];
353                         for(int i=0; i<toplen; i++){
354                             topic6[i]=payload[i+72];
355                         }
356                         packet=PacketParsingUtils.createSUBSCRIBEAck(payload,
                                intnoPacket);
357                         packetOut(ingressNodeRef, ingressNodeConnectorRef,
                                packet);
358                     }
```

```
359                         }else if (typeMQTT==140){
360                             LOG.debug("MENSAJE SUBACK     "+ typeMQTT);
361                         }else if (typeMQTT==162){
362                             LOG.debug("MENSAJE UNSUBSCRIBE     "+ typeMQTT);
363                         }else if (typeMQTT==176){
364                             LOG.debug("MENSAJE UNSUBACK     "+ typeMQTT);
365                         }else if (typeMQTT==192){
366                             LOG.debug("MENSAJE PINGREQ     "+ typeMQTT);
367                             LOG.debug("noPacket : "+ noPacket);
368                             byte[] packet=new byte[68];
369                             intnoPacket = (int) noPacket;
370                             packet=PacketParsingUtils.createPINGResp(payload,
                                    intnoPacket);
371                             packetOut(ingressNodeRef, ingressNodeConnectorRef, packet)
                                    ;
372                         }else if (typeMQTT==208){
373                             LOG.debug("MENSAJE PINGRESP     "+ typeMQTT);
374                         }else if (typeMQTT==224){
375                             LOG.debug("MENSAJE DISCONNECT     "+ typeMQTT);
376                         }else if (typeMQTT==240){
377                             LOG.debug("MENSAJE Reserved     "+ typeMQTT);
378                         }
379                     }
380                 }
381             }
382         }
383 }
```

A.3. Función para crear paquete *connack*.

En la fase de eliminación del bróker (sección 7.3) se programa la creación de distintos mensajes, necesarios para el correcto funcionamiento de la red sin el uso del bróker. El siguiente código muestra la función `createConnectAck`, a modo de ejemplo. La explicación de cómo y cuándo llamar esta función pertenece a la sección 7.3.2.

```
1  public static byte[] createConnectAck(byte[] payload, int noPacket) {
2      int i=0;
3      int a=0;
4      int b=0;
5      byte[] packet=new byte[70];
6      byte aa;
7      byte bb;
8      byte cc;
9
10     for(i=0; i<11; i++){
11         packet[i]=payload[i];
12     }
13
14     a=100;
15     aa=(byte)a;
16     packet[11]=aa;
17
18     for(i=0; i<5; i++){
19         packet[i+12]=payload[i+12];
20     }
21
22     a=56;
23     aa=(byte)a;
24     packet[17]=aa;
25
26     if (noPacket<256){
27         a=0;
28         bb=(byte)a;
29         packet[18]=bb;
30         cc=(byte)noPacket;
31         packet[19]=cc;
32     }else if (noPacket>=256){
```

```
33            a=noPacket/256;
34            bb=(byte)a;
35            packet[18]=bb;
36            b=noPacket-(a*256);
37            cc=(byte)b;
38            packet[19]=cc;
39        }
40
41        for(i=0; i<4; i++){
42            packet[i+20]=payload[i+20];
43        }
44
45        a=0;
46        aa=(byte)a;
47        packet[24]=aa;
48        packet[25]=aa;
49
50        for(i=0; i<4; i++){
51            packet[i+26]=payload[i+30];
52        }
53        for(i=0; i<4; i++){
54            packet[i+30]=payload[i+26];
55        }
56
57        short checksum = computeChecksumIP(packet, 14);
58        packet[25] = (byte)(checksum & 0xff);
59        packet[24] = (byte)((checksum >> 8) & 0xff);
60
61        for(i=0; i<2; i++){
62            packet[i+34]=payload[i+36];
63        }
64        for(i=0; i<2; i++){
65            packet[i+36]=payload[i+34];
66        }
67
68        for(i=0; i<4; i++){
69            packet[i+38]=payload[i+42];
70        }
71
72        int ack=(int) (payload[38]<<24 | payload[39]<<16 & 0xFF | payload[40]<<8 & 0xFF
              | payload[41] & 0xFF);
73        int ackfin=ack+37;
74        packet[45] = (byte)(ackfin & 0xFF);
75        packet[44] = (byte)((ackfin >> 8) & 0xFF);
76        packet[43] = (byte)((ackfin >> 16) & 0xFF);
77        packet[42] = (byte)((ackfin >> 24) & 0xFF);
78
79        for(i=0; i<3; i++){
80            packet[i+46]=payload[i+46];
81        }
82
83        a=57;
84        aa=(byte)a;
85        packet[49]=aa;
86
87        a=0;
88        aa=(byte)a;
89        packet[50]=aa;
90        packet[51]=aa;
91
92        for(i=0; i<14; i++){
93            packet[i+52]=payload[i+52];
94        }
95
96        checksum = computeChecksumTCPConn(packet, 34);
97        packet[51] = (byte)(checksum & 0xff);
98        packet[50] = (byte)((checksum >> 8) & 0xff);
99
100       a=32;
101       aa=(byte)a;
```

```
102         packet[66]=aa;
103
104         a=2;
105         aa=(byte)a;
106         packet[67]=aa;
107
108         a=0;
109         aa=(byte)a;
110         packet[68]=aa;
111         packet[69]=aa;
112
113         return packet;
114     }
```