UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Diseño de una solución basada en SDN para la microsegmentación de servicios IoT

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Martínez Galarza, Angely Georgette

Tutor/a: Tavares de Araujo Cesariny Calafate, Carlos Miguel

Director/a Experimental: PADRON PEREZ, JOSE DANIEL

CURSO ACADÉMICO: 2022/2023

Dedicatoria

A mi familia, por ser mi soporte incondicional no solo durante el tramo de la universidad, sino a lo largo de toda mi vida. Gracias mamá, por haber hecho de padre y madre, por el apoyo a perseguir todos mis sueños y el esfuerzo que has hecho y sigues haciendo para ofrecernos siempre lo mejor posible, incluso en los momentos más complicados. A Rose y Shaila, por ayudarnos entre nosotras y ser también una fuente de inspiración para mí, estoy muy orgullosa de vosotras y sin duda vais a conseguir cualquier cosa que os propongáis.

A mi tío y mi abuela, sois las figuras referentes de que todo se consigue con empeño y esfuerzo en la vida. Gracias por el apoyo que me brindáis día a día desde que me tuvisteis en vuestros brazos, sin vosotros tampoco podría haber llegado hasta aquí.

A ti, bisabuela, por haber sido mi segunda madre. No hay palabras para describirte, solo puedo decir que fuiste y sigues siendo la mujer ejemplar de nuestras vidas. Esto va por ti y espero que desde allí arriba estés orgullosa, porque sin tu camino sembrado esto sería imposible.

A Jaime, mi otro pilar fundamental. Gracias por aparecer en mi vida y convertirte en mi alma gemela. Es más que complicado expresar con palabras lo que hemos construido estos años, espero que podamos seguir creciendo juntos y ver cómo logras todas tus metas.

A mi pequeño felino, Jack, el cual me ha hecho descubrir el amor por los animales, que desde que llegó a casa hace tres años no se ha separado de mí durante un segundo y ha sufrido conmigo en la habitación interminables horas de estudio y trabajo.

Por último, a los docentes que han marcado un antes y después en mi etapa de la E.S.O y bachillerato: Marcos, Pepe, Carmen y Abelardo. No dudéis nunca que sois un ejemplo a seguir en todos los sentidos y marcáis la diferencia tanto a nivel personal como profesional. Gracias a vuestra vocación y personalidad lográis que el paso de un alumno por dicho tramo sea mucho más llevadero e inolvidable.

Resumen

Este proyecto consiste en el diseño y desarrollo de una solución basada en la arquitectura de redes definidas por software (SDN) para la administración de servicios IoT. Se simulará una topología de red mediante la herramienta *Mininet WiFi*, la cual podrá ser gestionada mediante una aplicación web.

El escenario de red simulado comprenderá dos roles diferenciados en los nodos: uno de ellos desempeñará el papel de cámaras de seguridad, mientras que los otros actuarán como clientes encargados de recibir las imágenes provenientes de dichas cámaras.

La aplicación web mostrará la agilidad y flexibilidad para gestionar servicios que ofrecen las SDN, permitiendo en este caso al administrador de red realizar cambios en la topología de forma dinámica, como eliminar clientes o establecer horarios de envío de tráfico en las cámaras.

Para llevar a cabo esta solución ha sido necesario desarrollar las tres capas que forman la arquitectura SDN: infraestructura, control y aplicación.

Palabras clave: SDN, Redes Definidas por Software, IoT, Mininet WiFi

Resum

Aquest projecte consisteix en el disseny i desenvolupament d'una solució basada en l'arquitectura de xarxes definides per programari (SDN) per a l'administració de serveis IoT. Se simularà una topologia de xarxa mitjançant l'eina *Mininet Wifi*, la qual podrà ser gestionada mitjançant una aplicació web.

L'escenari de xarxa simulat comprendrà dos rols diferenciats en els nodes: un d'ells exercirà el paper de càmeres de seguretat, mentre que els altres actuaran com a clients encarregats de rebre les imatges provinents d'aquestes cambres.

L'aplicació web mostrarà l'agilitat i flexibilitat per a gestionar serveis que ofereixen les SDN, permetent en aquest cas a l'administrador de xarxa fer canvis en la topologia de manera dinàmica, com eliminar clients o establir horaris d'enviament de trànsit en les cambres.

Per a dur a terme aquesta solució ha sigut necessari desenvolupar les tres capes que formen l'arquitectura SDN: infraestructura, control i aplicació.

Paraules clau: SDN, xarxes definides per programari, IoT, Mininet Wifi

Abstract

This project consists of the design and development of a solution based on software-defined networking (SDN) architecture for IoT service management. A network topology will be simulated using the *Mininet WiFi* tool, which can be managed by a web application.

The simulated network scenario will contain two distinct roles in the nodes: one of them will play the role of security cameras, while the others will act as clients in charge of receiving the images from those cameras.

The web application will show the agility and flexibility to manage services offered by the SDN, in this case allowing the network administrator to make changes in the topology in a dynamic way, such as eliminating customers or setting times for sending traffic in the cameras.

To carry out this solution it has been necessary to develop the three layers that form the SDN architecture: infrastructure, control and application.

Key words: SDN, Software-Defined Networking, IoT, Mininet WiFi

Índice general

Índice general V
Índice de figuras VII
Índice de tablas VIII

1 Introducción 1
1.1 Motivación . . . 2
1.2 Objetivos principales . . . 2
1.3 Estructura del documento . . . 3

2 Estado del arte 5
2.1 Internet of Things (IoT) . . . 5
2.1.1 Arquitectura IoT . . . 6
2.1.2 Aplicaciones principales y desafíos . . . 7
2.1.3 Cámaras como sensores IoT . . . 9
2.2 Redes definidas por software (SDN) . . . 10
2.2.1 Arquitectura SDN . . . 11
2.2.2 Protocolo OpenFlow . . . 12
2.2.3 Controladores de red SDN . . . 13
2.3 Uso de SDN en entornos IoT . . . 16
2.4 Soluciones previas en el ámbito académico . . . 18

3 Diseño de la solución 19
3.1 Análisis del problema . . . 19
3.2 Solución y arquitectura propuesta . . . 20
3.3 Tecnologías utilizadas . . . 21
3.3.1 Mininet-WiFi . . . 22
3.3.2 Python y el framework Flask . . . 22
3.3.3 XAMPP y MariaDB/MySQL . . . 23
3.4 Estrategias de comunicación . . . 23
3.4.1 Multicast . . . 23
3.4.2 APIs . . . 25
3.5 Diseño de las capas . . . 26
3.5.1 Diseño capa de infraestructura . . . 26
3.5.2 Diseño capa de control . . . 29
3.5.3 Diseño capa de aplicación . . . 30

4 Desarrollo de la solución propuesta 35
4.1 Solución inicial . . . 35
4.1.1 Simulación de la topología y envío de tráfico . . . 35
4.1.2 Añadiendo el controlador de red al escenario . . . 36
4.1.3 Aplicación web . . . 37
4.2 Funcionalidades avanzadas . . . 38

4.2.1 Gestión dinámica de clientes 38
4.2.2 Restricciones de tráfico con granularidad temporal fina . . . 39
5 Pruebas de validación **41**
5.1 Registro de nuevas reglas 41
5.2 Pruebas de carga 42
5.2.1 Carga individual sobre un cliente 42
5.2.2 Actualización de toda la red 44
5.2.3 Tiempo necesario para borrar un cliente 45
5.2.4 Porcentaje de uso de CPU en un vSwitch 47
6 Conclusiones **51**
6.1 Relación del trabajo con los estudios cursados 52
6.2 Trabajos futuros 53
Bibliografía **55**

Apéndices

A Objetivos de desarrollo sostenible **57**
B Guía para la instalación del entorno de trabajo **59**

Índice de figuras

2.1 Número de dispositivos conectados a la red 6
2.2 Arquitecturas IoT propuestas. 6
2.3 Arquitectura SDN. 12
2.4 Protocolo OpenFlow. 13
2.5 Ryu Controller 15
2.6 Arquitectura del controlador de red Ryu. 15
2.7 Redes definidas por software e IoT. 16

3.1 Arquitectura SDN de la solución propuesta. 21
3.2 Mininet WiFi. 22
3.3 Comunicación *multicast*. 23
3.4 Comunicaciones en la arquitectura de la solución. 25
3.5 Arquitectura de red de las cámaras. 26
3.6 Arquitectura de red de los clientes. 27
3.7 Flujo *multicast* de los clientes 1 y 5. 27
3.8 Ejemplo de asignación de puertos para el envío de tráfico *multicast*. 29
3.9 Estructura y diseño de la capa de control 29
3.10 Diseño arquitectura aplicación web. 31
3.11 Tablas de la base de datos. 31
3.12 Diseño de interfaces 32
3.13 Ejemplo de proceso al añadir una regla desde la app web. 33

5.1 Comprobación de reglas de tráfico en el dispositivo. 41
5.2 Log del controlador de red. 42
5.3 Dispositivo de red tras recibir reglas. 42
5.4 Pruebas cliente-servidor de tráfico *multicast* 44
5.5 Wireshark: envío y recibo de reglas 44
5.6 Gráfica de tráfico/tiempo de los clientes 45
5.7 Info de la app sobre el borrado de reglas. 46
5.8 Borrado de cliente 46
5.9 Procesos CPU del vSwitch. 47
5.10 Gráfica de paquetes/segundo en el switch 49

B.1 Tarjetas de red del sistema 59
B.2 Configuración de red de la máquina 60
B.3 Versión de Ryu 61

Índice de tablas

2.1 Arquitectura IoT de 5 capas 7
2.2 Controladores SDN de código abierto 14

3.1 Ejemplo de relación cámara-cliente 24
3.2 Ejemplo de relación cámara-cliente inicial 28
3.3 Lista de operaciones API del controlador 30

5.1 Resultados de la prueba cliente/servidor de datagramas UDP . . . 43
5.2 Instantes de tiempo de envío y actualización 44
5.3 Comparación del uso de CPU y carga en un switch tras recibir reglas 48
5.4 Comparación del uso de CPU y carga en un switch cuando circula tráfico por la red 48

CAPÍTULO 1
Introducción

Es por todos conocido cómo han cambiado nuestras vidas con la aparición de Internet, y la presencia que esta tiene en nuestro día a día, siendo indispensable en la mayoría de los casos. El rol que juegan las redes de computadoras en este escenario es clave, ya que hace posible la interconexión de los ordenadores y demás dispositivos para que puedan compartir e intercambiar información. Todo ello implica el uso de distintas tecnologías hardware y software, y establecer comunicaciones de manera transparente, independientemente de sí están situadas en el mismo lugar físico o geográficamente dispersas.

Siguiendo esta línea, la necesidad de alojar cada vez más datos y servicios web origina un crecimiento imparable de los centros de datos, lo cual desemboca, entre otras cosas, en topologías de red masivas y difíciles de gestionar [1]. Por esta razón principal, y muchas otras que se desarrollarán más adelante, surgen las redes definidas por software, un nuevo paradigma de arquitectura y gestión de redes.

A muy grandes rasgos, las redes definidas por software (SDN) son una tecnología que permite gestionar toda una red mediante una aplicación de software. Entre las muchas ventajas que nos ofrece, destaca la de poder crear reglas que se aplican en toda la red, diciéndole así a cada dispositivo cómo manejar el tráfico. Todo esto nos evita tener que configurar cada equipo individualmente, como se viene haciendo hasta ahora con la arquitectura de las redes tradicionales. Por ello es que cada vez más las empresas no dudan en implementar esta tecnología en sus infraestructuras [2].

Por otro lado, en el mismo ecosistema digital ha ido creciendo en paralelo otra red de redes: el Internet de las Cosas, más conocido como IoT. Para 2022 se calculó que habían más de 14.000 millones de dispositivos IoT conectados, mientras que para 2030 se prevé que se pueda llegar hasta los 30.000 millones [3].

El paradigma de IoT se ha extendido tanto a nivel individual como a nivel de sociedad. A nivel personal ha abarcado áreas como la domótica, vehículos autónomos, *wearables*, *e-learning* y otros avances antes inimaginables en nuestra vida diaria.

A nivel de sociedad, el Internet de las Cosas está teniendo un impacto transformador en la eficiencia de las ciudades, llevándolas a convertirse en *Smart Cities*. En el entorno urbano actual podemos observar una amplia variedad de tecnologías emergentes, como sistemas de videovigilancia con reconocimiento fa-

cial, soluciones de optimización energética para edificios, alumbrado público o sistemas de transporte inteligente. Estas innovaciones están cambiando la forma en que las ciudades funcionan, mejorando la seguridad, el consumo de energía, la movilidad y promoviendo una mayor calidad de vida para los ciudadanos [4].

En vista de todo esto, es de vital importancia administrar de manera óptima estas tecnologías emergentes. Con miles de millones de objetos conectados, la gestión y control de una extensa red distribuida se vuelve una tarea compleja. En este contexto, SDN brinda flexibilidad y capacidad de programación a la red del Internet de las Cosas (IoT), sin alterar la arquitectura subyacente de las implementaciones existentes. Esto permite una gestión más eficiente y adaptable de la red IoT, facilitando el despliegue y la administración de los dispositivos conectados de manera más fluida y efectiva.

1.1 Motivación

Este proyecto surge de las necesidades e inquietudes que generan las nuevas tecnologías que se desarrollan y aparecen cada día, y cómo gracias a ello podemos aprovecharlas para aplicarlas en todos los ámbitos posibles. En este caso, la aparición del nuevo enfoque de las redes tradicionales, las redes definidas por software, y la aplicación que se puede hacer de ella sobre el paradigma de IoT, otra red en constante evolución.

Investigar, diseñar e implementar todas las prestaciones que nos ofrece este nuevo tipo de redes es una gran oportunidad para mostrar por qué han surgido, y cuáles son todas las posibilidades que abarca aplicar esta nueva arquitectura. De hecho, las tecnologías de la información, especialmente el ámbito de las comunicaciones, representan un campo que no para de crecer, y la necesidad de interconectar todos los dispositivos es un gran reto.

A nivel personal, este trabajo me ha generado una gran motivación durante todo su desarrollo. Es innegable la dependencia que nuestra sociedad tiene de estar constantemente 'conectada' a la red, así como las múltiples ventajas que esto conlleva. Como ingenieros informáticos, tenemos el trabajo de ofrecer y diseñar las soluciones más eficientes para satisfacer esta creciente necesidad. Actualmente trabajo como ingeniera de redes y seguridad, brindando un gran abanico de soluciones a empresas de diversos sectores (turismo, ocio, administración, educación, sanidad...). Esta experiencia ha ampliado mi visión sobre la importancia vital de las tecnologías de la información en nuestro día a día, ya que son indispensables para el funcionamiento óptimo de cualquier organización, y para garantizar la prestación de sus servicios. Continuar contribuyendo a este campo en constante evolución es un desafío que me impulsa a seguir aprendiendo y mejorando constantemente.

1.2 Objetivos principales

En este proyecto se plantea el objetivo de crear una solución basada en SDN que permite gestionar de manera flexible y eficiente el acceso a diferentes fuentes

de datos provenientes de distintos dispositivos IoT, los cuales se quiere compartir de manera condicional con diferentes usuarios del servicio. Concretamente, el escenario de red que se implementará consiste, por un lado, en nodos IoT con flujos de datos elevados, como pueden ser cámaras de videovigilancia. Por otro lado, habrá nodos (servidores) conectados de forma física, los cuales tendrán el rol de clientes, y recibirán el tráfico de dichas cámaras según las condiciones del servicio que se hayan acordado. Siendo así, la idea principal es permitir al operador añadir y/o eliminar los nodos de forma dinámica y segura, controlando a su vez el flujo de toda la red hacia los diferentes clientes.

En resumen, se desplegará una red definida por software en la que se podrá observar la administración y flexibilidad que brinda esta tecnología, entre otras muchas características. Para ello habrá que implementar 3 capas: infraestructura, control y aplicación. Para lograrlo se abordarán los siguientes subobjetivos:

1. Estudiar y analizar la tecnología SDN.
2. Investigar el paradigma IoT y su integración con las redes definidas por software.
3. Analizar el entorno de pruebas y sus correspondientes herramientas principales: Mininet WiFi y el controlador Ryu.
4. Simular un entorno de red específico con la herramienta Mininet WiFi.
5. Implementar el controlador de red, en este caso *Ryu Controller*, en base a los requisitos del sistema.
6. Desarrollar una aplicación web que se comunique con la API del controlador y permita gestionar todo el escenario de la red.

1.3 Estructura del documento

El siguiente capítulo, dedicado al estado del arte, tratará de acercar al lector a la situación actual tecnológica de los dos paradigmas principales que engloba el trabajo: la red IoT y las redes definidas por software. Se definirán ambos conceptos teóricamente, se profundizará en su arquitectura y se explicará cuáles son sus aplicaciones más recurrentes a día de hoy. Como último punto de esta primera vertiente, se expondrá el nexo que se da entre ellas, y finalmente se argumentará qué puede aportar el TFG, justificando su desarrollo.

En el capítulo 3 se pasará a la fase de diseño, donde el primer paso será plantear y analizar los desafíos que propone el crecimiento de la red IoT, y justificar por qué es una tarea complicada seguir gestionándolas a través de las redes tradicionales. A continuación, se describirá la solución propuesta a este conflicto, y en consecuencia las primeras decisiones para llevarla a cabo (herramientas usadas, estrategias de comunicación y el diseño de la arquitectura).

En el capítulo 4 se presentará el desarrollo de la solución propuesta, ilustrando cuáles han sido los pasos a seguir para desarrollar todas las capas de nuestra arquitectura (por ejemplo, se muestra a grandes rasgos alguna parte de código),

y determinados conflictos que se han ido encontrando a lo largo del camino. Acto seguido, también se expondrán las funcionalidades avanzadas que se añaden sobre la solución base posteriormente.

A continuación, en el capítulo 5, se aportarán diversas pruebas de validación y de carga del conjunto de la arquitectura y de elementos específicos que la conforman. Se mostrará que la solución funciona correctamente y, por otro lado, a través de estadísticas de tráfico de la red y de tiempo, se analizará cuál es su consumo de recursos y/o eficiencia.

Finalmente, en el capítulo 6, se presentarán las conclusiones y posibles trabajos futuros.

CAPÍTULO 2
Estado del arte

En este capítulo se va a dar una visión general sobre diferentes conceptos relacionados con este TFG, incluyendo IoT, SDN, y uso de SDN en entornos IoT. Por último, se presenta una crítica al estado del arte actual.

2.1 Internet of Things (IoT)

Estamos entrando en una nueva era de la tecnología y, sin lugar a dudas, la aparición del paradigma IoT y la comunicación máquina a máquina (M2M) [1], presentan un gran potencial y un gran reto para el sector de las comunicaciones.

La realidad es que, a la hora de definir el concepto de IoT, nos encontramos con varias visiones diferentes y sus correspondientes definiciones. Sin embargo, la más completa, y la que unifica todos los conceptos, sería la siguiente: 'Una red abierta y completa de objetos inteligentes que tienen la capacidad de auto-organizarse, compartir información, datos y recursos, reaccionar y actuar ante situaciones y cambios en el entorno' [5].

Su crecimiento, y en consecuencia los billones de dispositivos conectados a la red, es irrefrenable, y así lo demuestra un estudio del fabricante Cisco, el cual advierte que superará 3 veces más a la población global existente en el año 2023 [6]: *"Habrá 3,6 dispositivos en red per cápita para 2023, frente a los 2,4 dispositivos en red per cápita en 2018. Lo que quiere decir que existirán 29.300 millones de dispositivos en red para 2023, frente a los 18.400 millones en 2018."* En la figura 2.1 vemos representada esta predicción de forma gráfica.

[1] El M2M, o comunicación machine to machine, describe el intercambio de información automatizado entre 2 o más dispositivos sin intervención humana.

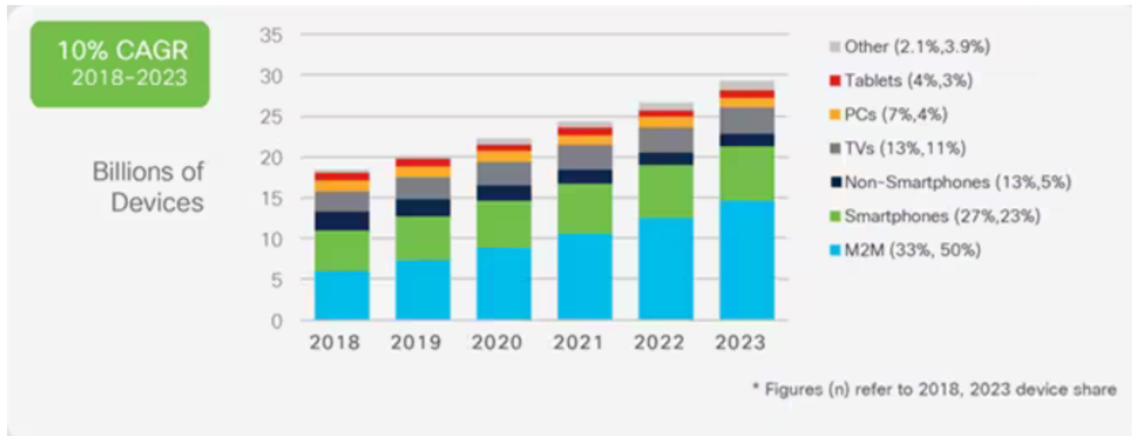


Figura 2.1: Número de dispositivos conectados a la red

¿Dónde están conectados, quién gestiona y/o para qué se usan todos estos dispositivos? Es lo que analizaremos y explicaremos más adelante.

2.1.1. Arquitectura IoT

Antes de describir qué aplicaciones tiene esta tecnología, es esencial saber cuál es la infraestructura de este sistema; de este modo sabremos cuál es la función de cada componente que lo integra.

Actualmente, no existe un solo estándar establecido para la arquitectura IoT. Debido a ello, diferentes investigadores y/o instituciones proponen modelos equivalentes. De entre todos los propuestos se puede agrupar en los 3 tipos que vemos en la figura siguiente 2.2 :

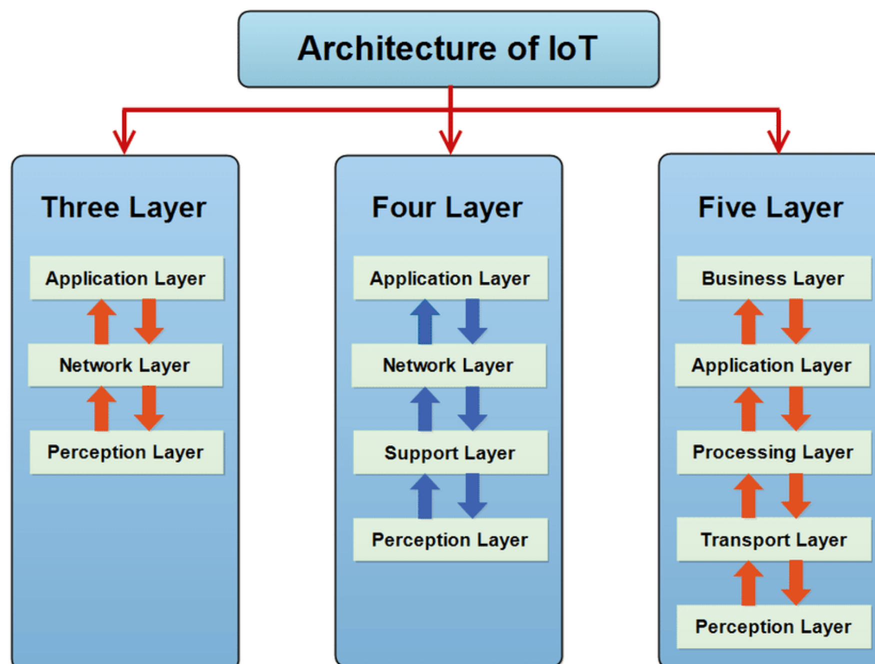


Figura 2.2: Arquitecturas IoT propuestas.

Como se puede ver en la imagen 2.2, son muy parecidas entre ellas, y la más completa es la de 5 capas. Por ello vamos a detallar esta última.

Capa	**Descripción**
Capa de percepción	Formada por los sensores, los cuales son los encargados de recopilar la información del entorno.
Capa de transporte	Es la encargada de transferir los datos recopilados por los sensores en la capa de percepción hasta la capa superior (procesamiento). Resuelve las comunicaciones a nivel de red (Bluetooth, WiFi, NB-IoT, NFC...).
Capa de procesamiento	Tiene el rol de procesar, analizar y almacenar los datos que le ha proporcionado la capa de transporte. Además de esto, gestiona y proporciona diferentes servicios a las 2 capas inferiores: bases de datos, *Big Data*, módulos de procesamiento, *Cloud Computing*, etc.
Capa de aplicación	Responsable de entregar los servicios de aplicación a los usuarios a través de interfaces de usuario. Va desde aplicaciones domésticas hasta aplicaciones de logística para empresas.
Capa de negocio	Gestiona y resuelve los «problemas» del más alto nivel de abstracción, como por ejemplo la privacidad de los datos de usuario o los modelos de negocio. Todo esto abarca la posibilidad de mejorar y optimizar el sistema IoT, y por ello es una de las capas más usadas por los fabricantes.

Tabla 2.1: Arquitectura IoT de 5 capas

2.1.2. Aplicaciones principales y desafíos

El número de aplicaciones y servicios que puede proporcionar el paradigma IoT es prácticamente ilimitado, pudiendo adaptarse a muchos campos de la actividad humana al facilitar y mejorar su calidad de vida de múltiples maneras. Vamos a presentar las aplicaciones y servicios más destacables.

- **Monitorización ambiental**: estos sensores y aplicaciones de monitoreo ambiental ayudan, por ejemplo, a hacer predicciones de desastres naturales como terremotos, incendios, inundaciones, etc. También ayudan a hacer controles de calidad del aire, agua o atmósfera, contribuyendo esto a la protección del medio ambiente.

- **IoT doméstico**: también conocido como *"Smart Homes"* (Casas Inteligentes). Capaz de agilizar y automatizar el uso la mayoría de dispositivos de una casa, convirtiéndolos así en sensores inteligentes: controlar la calefacción y aire acondicionado, sensores de ruido, humedad y concentraciones de dióxido de carbono [7], cerraduras de puertas inteligentes, o control de la energía, agua y gas en tiempo real.

- **Salud y Hospitales Inteligentes**: la digitalización en el tratamiento de la salud se ha disparado con las aplicaciones que se pueden hacer del IoT en

este ámbito. Mejorando la experiencia del paciente, las capacidades médicas y la optimización de costes, se pueden destacar casos de uso como la monitorización de pacientes y enfermedades crónicas (a través de *wearables* y dispositivos médicos), la gestión de tratamientos en remoto (sensores de movimiento, dispensadores de medicina), o la monitorización y operaciones remotas de infraestructuras médicas (monitorizar equipamiento médico y realizar intervenciones remotas a pacientes).

- **Transporte y sistemas de tráfico**: por un lado, tenemos los vehículos autónomos usando servicios IoT que ofrecen información avanzada (control mejorado de la energía, GPS avanzado, autodiagnósticos, acelerómetros, etc.). Por el otro, existen sistemas de monitoreo de tráfico inteligente que proporcionan mejoras como identificación de vehículos, notificación de accidentes de tráfico, o enriquecer la experiencia de la conducción reduciendo las congestiones.

- **Ciudades inteligentes**: aunque la definición de *Smart Cities* actualmente no está del todo clara, se podría resumir en *'el uso de las tecnologías de la información y sus derivadas para ayudar a solucionar problemas urbanos, gestionar ciudades y mejorar la calidad de vida'* [7]. Todo ello implica cientos de aplicaciones como: optimización de los sistemas de transporte públicos, integración de los servicios de seguridad (cámaras, sensores inteligentes), crecimiento económico, riego de parques y jardines, contenedores de basura inteligentes [8], etc.

Sin duda, es evidente como la aplicación de las tecnologías IoT está evolucionando y transformando todos los sectores de la sociedad. Sin embargo, siguen actualmente consolidándose en el mercado, y es por ello que a día de hoy presenta una serie de **desafíos** que aún no están resueltos:

La **Ciberseguridad** en todos sus ámbitos (personas, procesos y tecnología). IoT hace que cada cosa y persona sea localizable y, lo que es una gran ventaja, también se convierte en una falta de confianza sobre la seguridad y privacidad de los datos del usuario. Existe, por ejemplo, una gran brecha en toda la red de sensores IoT [9], ya que las WSN [2] son vulnerables a varios tipos de ataques como *jamming, tampering, sybil o flooding.* [3]. Es por ello que es imprescindible un modelo de ciberseguridad para IoT, y proteger toda la cadena de valor que la compone (software, hardware, seguridad física y protocolos de comunicaciones).

La **Estandarización**, ya que, como hemos podido ver hasta ahora, no hay definiciones concisas o totalmente establecidas de toda la tecnología IoT. Esto se debe, a que día de hoy, es un sector bastante nuevo comparado con los demás,

[2]Las redes de sensores inalámbricos (en inglés, wireless sensor networks, abreviadamente WSN), también llamadas redes de sensores y actuadores (wireless sensor and actuator networks, WSAN) son sensores autónomos espacialmente distribuidos para monitorizar condiciones físicas o ambientales, como temperatura, sonidos, presión, etc [10].

[3]Jamming: obstruye la red entera interfiriendo las frecuencias de los nodos sensores.
Tampering: Es la forma de ataque en la que los datos del nodo pueden ser extraídos o alterados por el atacante para hacerlo totalmente controlable.
Sybil: utiliza un solo nodo para operar muchas identidades falsas activas (o identidades Sybil) simultáneamente, dentro de una red de igual a igual.
Flooding: ataque de tipo DoS.

pero ello no quita que sea urgente una estandarización a diferentes niveles (técnico, normativo, alcance geográfico) para que la industria y el mercado puedan producir soluciones heterogéneas.

La **Energía** y su gestión, ya que los dispositivos que forman parte de la red IoT obtienen energía de la red eléctrica o utilizan baterías. Dado el aumento previsto en la cantidad de dispositivos en los próximos años, es crucial desarrollar sistemas de almacenamiento de energía más confiables y sostenibles. Estos sistemas permitirían que los dispositivos actuales tengan una mayor autonomía de funcionamiento sin necesidad de cargarlos con frecuencia. Todo ello requiere investigar y crear hardware y protocolos de comunicación más eficientes en términos de consumo energético, lo que ayudaría a reducir la demanda de energía de los dispositivos.

2.1.3. Cámaras como sensores IoT

Tradicionalmente, el concepto de 'cámaras de seguridad' se asocia a sensores de presencia en modo pasivo que solo envían imágenes, lo cual no es erróneo, sin embargo, el rol actual de estos sensores va mucho más allá. A día de hoy las cámaras están conectadas a la red, y se han convertido en cámaras IoT, consistiendo de una unidad de procesamiento que filtra los datos, y solo envía la información relevante a la nube para su posterior procesamiento y almacenamiento.

Aparte de mejorar nuestra seguridad física gracias al vídeo en tiempo real, estos sensores también pueden realizar análisis del entorno: reconocimiento facial, alerta ante peligros o anomalías, etc. Todo esto se transforma en soluciones como cerraduras inteligentes o sensores de fugas de agua, gas y contaminación ambiental. Un estudio de *McKinsey Global* [11] señala que *'el análisis de grandes volúmenes de datos recopilados por las cámaras y sensores tienen el potencial de disminuir las tasas delictivas en un rango del 30 al 40 %, acelerando también así la respuesta ante situaciones de emergencia hasta un 35 %.'*

En el ámbito de las *smart cities* estas cámaras y sensores tienen un papel sustancial: "*Para 2050, la ONU prevé que la videovigilancia inteligente será parte fundamental y cotidiana de la vida de sus ciudadanos.*" [12] Recordemos que uno de los principales objetivos de las *smart cities* es mejorar las condiciones y calidad de vida de sus ciudadanos, y las cámaras IoT contribuyen a ello teniendo diversas aplicaciones como el tráfico y la movilidad, la seguridad ciudadana, el medio ambiente, o la seguridad sanitaria.

Por ejemplo, respecto a la seguridad ambiental, es factible utilizar sensores que puedan identificar la existencia de partículas dañinas o explosivas en el aire. Estos sensores tienen establecidos ciertos límites máximos para determinar niveles aceptables de contaminación. Si se alcanzan o se superan esos límites, se pondrían en marcha procedimientos específicos para garantizar la protección física de las personas, como la emisión de alarmas relacionadas con la contaminación.

En cuanto a la seguridad sanitaria, las cámaras pueden identificar patrones y reconocer desplazamientos habituales; para ilustrar esta aplicación, está el caso del control de alerta sanitaria en la reciente crisis del Covid-19. En la pandemia, fueron clave las cámaras térmicas para detectar posibles infecciones de la enfer-

medad. Además, esta tecnología también se puede aplicar en control del tráfico para agilizar el paso de vehículos de emergencia, o para gestionar áreas de acceso y aparcamiento restringido en el entorno de centros de salud.

Asimismo, todas estas soluciones tecnológicas implican unos riesgos de ciberseguridad inherentes a la misma. Sin ir más allá, uno de los grandes desafíos es la posible intromisión en la privacidad de los ciudadanos a través de estas cámaras. La estrategia para afrontar todos estos peligros ha de basarse en una garantía de seguridad, estableciendo protocolos y estándares seguros para avalar la protección de datos utilizados por los sensores IoT.

2.2 Redes definidas por software (SDN)

Las redes definidas por software (SDN por sus siglas en inglés) son la solución a las limitaciones de las redes tradicionales actuales. Hoy por hoy, hay una infinidad de campos que ofrecen distintas descripciones del concepto de las SDN, y una de las definiciones pragmáticas que podemos encontrarnos es la siguiente: "*Las redes definidas por software ofrecen a los administradores acceder a la red mediante programación, lo cual proporciona una gestión automatizada y técnicas de orquestación avanzadas; aplicaciones para configurar políticas en múltiples routers, switches y servidores; y el desacoplamiento de la aplicación que realiza todas estas operaciones desde el sistema operativo de los dispositivos de red*". [13]

Esta gestión dinámica y flexible que ofrece es debido a su arquitectura, ya que separa el plano de control del plano de datos de la red. El plano de control hace referencia a los procesos de red que dirigen el tráfico de red (establecen las rutas y comunican qué protocolos deben utilizarse), mientras que el plano de datos son los propios flujos de paquetes de datos que atraviesan la red. Esta separación del plano del control respecto al hardware subyacente, hace posible administrar y gestionar la topología de red mediante software en lugar de hardware, lo que habilita que las configuraciones de dispositivos de red y sus políticas ya no tengan que ser implementadas sobre el propio hardware.

Teniendo en cuenta las funcionalidades que ofrece, se pueden resumir las características de SDN en las siguientes:

1. **Separación de planos**. La característica clave de SDN es el desacople entre los planos de control y de datos. El plano de datos contiene las tablas de reenvío y la lógica para tratar los paquetes entrantes según dirección MAC y/o IP. El plano de control maneja dicha lógica, además de determinar cómo se configuran las tablas de reenvío. En SDN, todos los distintos planos de control se trasladan a un controlador centralizado.

2. **Simplificación del dispositivo y controlador centralizado**. Los dispositivos de red, en lugar de tener en su plano de control un software ejecutando miles de línea de código, éste se elimina, y el software se coloca en el controlador centralizado con mucha más eficiencia. El controlador proporciona las órdenes a estos dispositivos cuando sea necesario, permitiéndoles tomar decisiones más rápidas sobre los paquetes que les llegan.

3. **Transparencia**. Otra característica básica que ofrecen las redes SDN es lo transparente que son, puesto que se mantienen constantemente documentadas y estandarizadas abiertamente; esto permite a cualquier interesado/a del área probar e implementar nuevas ideas, resultando en una contribución de una gran comunidad que da lugar a corregir errores más rapidamente, y a más avances tecnológicos.

4. **Virtualización de la red**. La idea de la virtualización es crear una abstracción de nivel superior que se ejecute sobre la instancia física, de tal modo que se permita la coexistencia de múltiples instancias de red en una infraestructura física compartida. Con la virtualización de la red, el administrador puede crear, ampliar y contratar una red en cualquier momento y en cualquier lugar según requisitos [14].

Mercado e Industria SDN

Actualmente, desde el punto de vista práctico, SDN es todavía una arquitectura en evolución y en fase de desarrollo. No obstante, su mercado ha mostrado un significativo crecimiento durante los últimos años, siendo mayormente demandado en la industria del *Cloud Computing*, *Data Centers*, entornos híbridos, el *Edge*, o para la red IoT y sus dispositivos conectados. Y es que, según informe de *Expert Market Research* [15], las redes SDN están creciendo a un ritmo acelerado, puesto que alcanzaron un volumen de US$9.2 mil millones en 2020, y se prevé que se llegue en 2026 a los US$35.6 mil millones, lo que equivale a un crecimiento del 500 %.

El crecimiento en los últimos años se debe a las ventajas que ofrecen las redes definidas por software frente a las redes tradicionales, lo que ha hecho que la industria se focalice en ellas para mejorar los servicios en sus propias redes privadas, o para desarrollar y proporcionar sus propias soluciones SDN comerciales.

Uno de los ejemplos más característicos de la adopción de SDN es Google, la cual entró en el mundo de las SDN con su red B4, desarrollada para conectar todos sus centros de datos distribuidos por el mundo. Como explicaron los ingenieros de Google [16], fue una decisión clave para satisfacer las demandas de ancho de banda de sus aplicaciones, además de que actualmente la red SDN que tienen implementada transporta más tráfico WAN que la pública.

Empresas como Cisco, HP, Alcatel, Oracle y NTT, entre muchas otras, también están en el mercado SDN, reconociendo sus ventajas, y ofreciendo sus propias soluciones destinadas a otras empresas y proveedores de servicios en la nube.

2.2.1. Arquitectura SDN

La arquitectura de redes definidas por software ofrece un enfoque revolucionario en la gestión de redes, brindando flexibilidad, agilidad y control centralizado. A diferencia de las redes tradicionales, en las que los dispositivos de red individualmente gestionan y controlan su comportamiento, SDN separa el plano de control del plano de datos, como bien hemos explicado, permitiendo una gestión y control más centralizados.

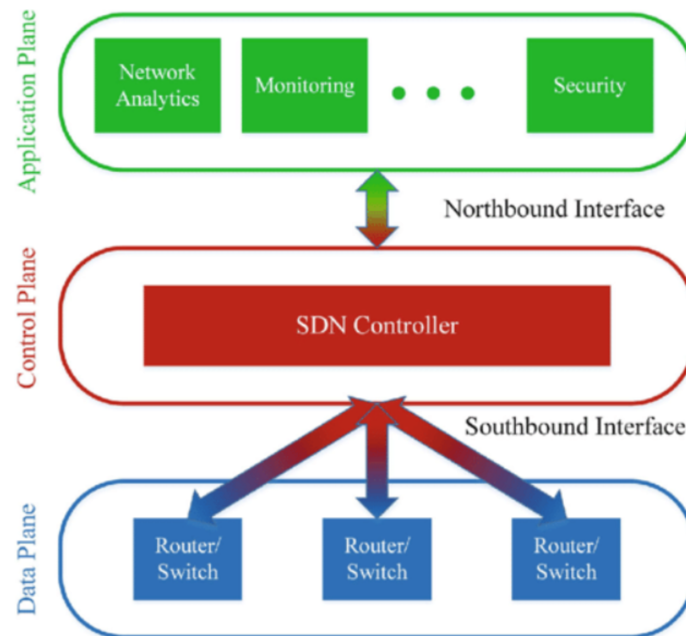


Figura 2.3: Arquitectura SDN.

En la figura 2.3 podemos observar la arquitectura SDN, la cual está compuesta por 3 partes bien diferenciadas: capas de aplicación, control y datos.

- **Capa de datos**. Abarca los dispositivos de red, switches y routers, que son los encargados del envío y procesamiento de los datos. Reciben información del controlador SDN a través de la «API hacia abajo» *(southbound interface)*, como por ejemplo *OpenFlow*, para poder saber dónde y cómo mover los datos. Los dispositivos también envían datos a la capa de control, como por ejemplo recopilaciones de información crítica o de uso, o la topología de red.

- **Capa de control**. Formada por el controlador SDN, el cual es el *cerebro* de toda la arquitectura, además de ser el punto de unión entre la capa de aplicación y la de datos. Por un lado, es la entidad que controla, administra políticas, y configura los nodos de red para dirigir correctamente los flujos de tráfico. Por el otro, utiliza la información procedente de las aplicaciones para transmitirla a los componentes de la red.

- **Capa de aplicación**. Conjunto de aplicaciones y funciones de red que comunican instrucciones específicas al controlador SDN. Es abarcada en su mayoría por aplicaciones de negocio de usuarios finales. La comunicación se realiza a través de la «API hacia arriba» *(northbound interface)*, tales como REST, JSON, XML, etc. Esta capa permite simplificar y automatizar las tareas de configuración y provisión, así como gestionar nuevos servicios en la red.

2.2.2. Protocolo OpenFlow

OpenFlow es un protocolo de estándar abierto utilizado para la gestión de redes SDN, el cual establece las comunicaciones entre el plano de control y el plano

de datos. Fue estandarizado en 2011 por la organización sin ánimo de lucro ONF (*Open Networking Foundation*), [4], la cual trabaja continuamente con el objetivo de comercializar e impulsar el uso del protocolo en todas las redes en producción.

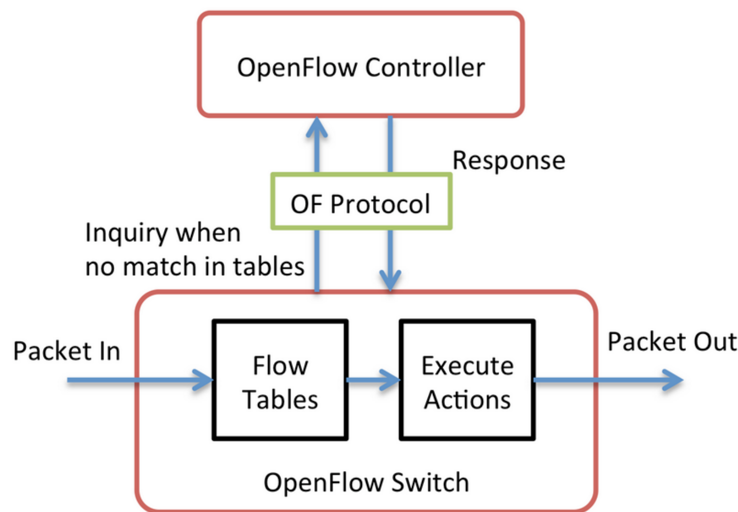


Figura 2.4: Protocolo OpenFlow.

La figura 2.4 nos muestra cómo trabaja el protocolo: un software central (*OpenFlow Controller*) define el comportamiento del dispositivo *OpenFlow Switch* a través del protocolo de comunicación *OpenFlow*. El switch dispone internamente de tablas de flujo; por tanto, cuando llegue un paquete, intentará hacer *match* con alguna de las entradas de la tabla y, en caso de coincidir, ejecutará la acción correspondiente con el paquete. Si no coincide con ninguna entrada, el resultado dependerá de la configuración que se haya establecido previamente: se reenviará al controlador de *OpenFlow*, se continuará intentando hacer *matching* con la siguiente tabla del switch, o se descartará directamente [17].

2.2.3. Controladores de red SDN

Llegado a este punto del trabajo, ya se ha definido en profundidad qué es un controlador de red SDN, y cuál es su función dentro de la arquitectura SDN. Por ello, expondremos a continuación algunas de las ventajas y desventajas de su uso, y a continuación cuáles son las opciones que nos podemos encontrar actualmente en el mercado para implementarlos en nuestra red.

Por un lado, una de las grandes ventajas de este control centralizado sobre la red es la rápida configuración que permite sobre la topología, ahorrando tiempo a los administradores en lugar de ir configurando los dispositivos uno por uno. También es destacable la escalabilidad que ofrece, puesto que, a medida que crece la red, es necesario agregar nuevos dispositivos, y el controlador es capaz de detectarlos automáticamente y asignarles la configuración y políticas correspondientes.

[4]Open Networking Foundation (ONF) es una organización impulsada por el usuario dedicada al desarrollo e implementación de redes definidas por software (SDN). Un logro notable de la organización ha sido su adopción del Estándar OpenFlow. ONF trabaja con operadores para aprovechar SDN para sus clientes e implementar OpenFlow en sus redes.

Respecto a desventajas, nos encontramos con que su punto más fuerte, la centralización, lo convierte en una carencia a niveles de independencia o de seguridad. Si el controlador fallase o experimentase cualquier tipo de problema, lo más probable es que acabe afectando a toda la red. No obstante, en mayor medida, esto podría gestionarse teniendo *backups* de controladores para conseguir redundancia. De igual modo ocurre en cuanto a niveles de seguridad: el control está centralizado y es crucial garantizar medidas de seguridad robustas para protegerlo.

En último lugar, cabe ilustrar los controladores de código abierto más populares que existen actualmente en el ámbito, y cuáles son sus características (ver tabla 2.2).

Nombre	**Lenguaje de programación**	**Licencia**	**Comentario**
NOX	C++	GPL	Inicialmente desarrollado en la Universidad de Stanford y el primer controlador original del protocolo OpenFlow. Soportado en sistemas Linux.
POX	Python	Apache	Escrito en Python y multiplataforma (Linux, Windows o Macintosh).
Ryu	Python	BSD	Ryu soporta varios protocolos para la gestión de dispositivos de red, como OpenFlow, Netconf, etc. Solo soportado en sistemas Linux.
Floodlight	Java	Apache	Es uno de los controladores más utilizados. Destaca por integrar una API Rest y dos entornos gráficos, uno web y otro Java.
OpenDaylight	Java	EPL	Es un controlador creado para Linux en 2013, y no obstante es multiplataforma. Es, según su página oficial, el controlador SDN de código abierto más implementado en la actualidad.

Tabla 2.2: Controladores SDN de código abierto

También podemos encontrar en el mercado controladores SDN comerciales de grandes fabricantes como:

- **Cisco**: *Cisco Application Policy Infrastructure Controller (APIC)*, controlador SDN que ofrece una gestión centralizada y automatización de políticas para redes empresariales.
- **Juniper Networks**: denominado *Juniper Contrail*, el cual permite automatización y orquestación de redes definidas por software, simplificando la gestión de redes complejas.
- **VMware**: Controlador SDN de VMware que proporciona virtualización de redes y seguridad, permitiendo la creación y gestión de redes virtuales en entornos de nube y centros de datos.

Controlador de red SDN: Ryu

Como último apartado de esta sección, es necesario ahondar y describir todas las ventajas que nos ofrece el controlador Ryu, puesto que es el que hemos usado para el desarrollo de la solución.

Ryu[5] es un controlador de red definido por software (SDN), diseñado para aumentar la agilidad de la red al facilitar la administración y la adaptación del tráfico [18].

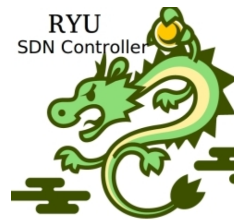


Figura 2.5: Ryu Controller

Ofrece componentes de software con interfaz de programación de aplicaciones (API) bien definidas, que simplifican la labor de los desarrolladores al crear nuevas aplicaciones para la gestión y control de redes. Este controlador se ha desarrollado utilizando Python, un lenguaje que presenta ventajas en términos de su sintaxis y semántica, ya que es relativamente sencillo en comparación con otros lenguajes similares; además, es completamente compatible con la programación orientada a objetos.

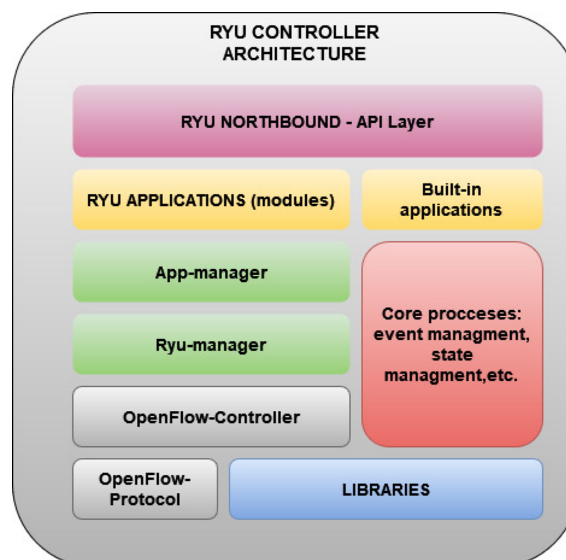


Figura 2.6: Arquitectura del controlador de red Ryu.

En la figura 2.6 podemos contemplar la estructura interna de este controlador. Destaquemos los principales componentes:

Ryu dispone de un gran abanico de opciones en cuanto a **librerías**, desde protocolos *southbound* (los protocolos que se comunican con la capa de infraes-

[5] https://ryu-sdn.org/

tructura), hasta operaciones para procesar de forma más eficiente los paquetes de red.

En cuanto al **protocolo OpenFlow**, cabe destacar que se pueden usar muchos otros; no obstante, es el más usado en este controlador para gestionar la red y los flujos de tráfico.

Ryu-manager es el núcleo de Ryu, y permite la conexión de switches OpenFlow. Mientras que **App-manager**, el gestor de aplicaciones, es esencial para todas las aplicaciones de Ryu. El componente central (***core procceses***) se encarga de la gestión de eventos y la mensajería, además de admitir componentes en otros lenguajes.

Ryu incluye diversas **aplicaciones** como *simple_switch, router, isolation, firewall, GRE tunnel, topology, VLAN*, etc. Estas aplicaciones son entidades de un solo hilo que se comunican entre sí mediante eventos asíncronos.

En último lugar, nos encontramos con las **API *northbound*** (hacia arriba), las cuales ofrecen, a través de interfaces REST, operaciones OpenFlow que irán dirigidas después a los dispositivos. Además, utilizando WSGI (un marco para conectar aplicaciones web y servidores web en Python), se pueden introducir fácilmente nuevas API REST en una aplicación.

2.3 Uso de SDN en entornos IoT

Como ya se ha comentado en capítulos previos, las redes IoT están en constante crecimiento, y el número de objetos conectados asciende a billones; por ello, su gestión y control se convierte en una tarea compleja dentro de una gran red distribuida. Estos dispositivos también generan una gran cantidad de datos que son gestionados por las infraestructuras tradicionales (*access points*, switches, routers, redes 3G/4G/5G), las cuales deben adaptarse a los nuevos servicios como *QoS*, *cloud computing*, VoIP o aplicaciones IoT, entre otros. Sin embargo, las redes y protocolos tradicionales actuales no están diseñados para hacer frente a los altos niveles de escalabilidad a los que se ha llegado, las grandes cantidades de tráfico, y la movilidad [19].

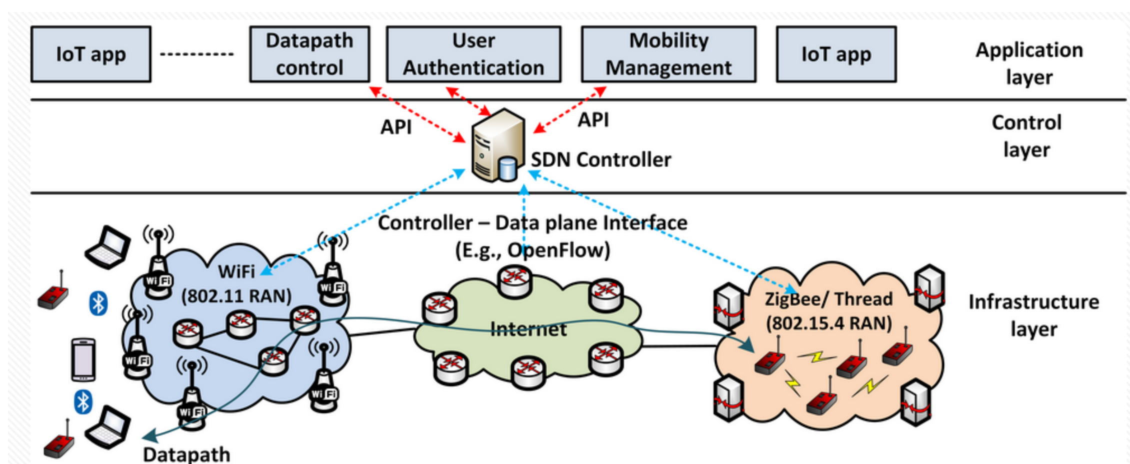


Figura 2.7: Redes definidas por software e IoT.

En base a la información brindada hasta ahora, es evidente que las redes SDN eliminan la rigidez que siempre han tenido las redes tradicionales. Debido a su estructura, estas nuevas redes son más flexibles y adaptables al dinamismo que pueda presentar cualquier paradigma emergente. Es por esto que la automatización, aprovisionamiento, programabilidad y orquestación que aporta SDN se adapta completamente a las necesidades y requerimientos que se dan en la tecnología IoT.

Ahora bien, cabe destacar que IoT y SDN son dos tecnologías totalmente distintas. La arquitectura IoT consiste en múltiples capas en la cual cada una consta de múltiples tecnologías, mientras que SDN es una separación entre el plano de control y de datos de una red. No obstante, es por ello que IoT puede aprovechar los beneficios que ofrece la capa de independencia de control SDN. El ejemplo podemos verlo en la figura 2.7.

Alguna de las ventajas que aporta la aplicación de las redes definidas por software sobre IoT son las siguientes:

- **Incremento en el control con más velocidad y flexibilidad.** En lugar de tener que programar manualmente diversos dispositivos de hardware específicos de cada proveedor, los desarrolladores cuentan con la posibilidad de gestionar el flujo de tráfico en la red mediante la programación de un controlador basado en software de estándar abierto.

- **Infraestructura de red personalizable.** Mediante una red definida por software, los administradores tienen la capacidad de diseñar servicios de red de manera centralizada, y asignar recursos virtuales de forma instantánea para adaptar la topología de red. Esta capacidad permite a los administradores priorizar las aplicaciones que requieren una mayor disponibilidad, y optimizar el flujo de datos en toda la red.

- **Optimización del ancho de banda.** La capacidad de las SDN de gestionar dinámicamente las cargas en la red permite a los administradores supervisar y orquestar automáticamente los cambios en el ancho de banda en toda la infraestructura. Además, esta planificación garantiza anticiparse a las necesidades de tráfico de una aplicación o cliente en momentos específicos. Esta funcionalidad es una gran ventaja para IoT, puesto que una de sus características principales es que existen dispositivos que solo envían datos periódicamente, y en momentos predeterminados.

- **Seguridad sólida.** En el contexto de IoT, una red definida por software proporciona una visibilidad completa de toda la red, lo que permite tener una imagen clara de las amenazas de seguridad. Los administradores tienen la capacidad de crear zonas separadas para dispositivos que requieran diferentes niveles de seguridad (microsegmentación), o de aislar de inmediato los dispositivos comprometidos para evitar que infecten al resto de la red. Esto asegura una protección efectiva contra posibles ataques y garantiza la integridad de los sistemas.

2.4 Soluciones previas en el ámbito académico

Habiendo analizado las soluciones académicas presentadas en la ETSINF sobre temática relacionada, si bien cabe decir que alguna vez se menciona el concepto de SDN, la conclusión es que realmente el tema apenas se ha estudiado y/o abordado en profundidad. No obstante, toda la información que se ha podido hallar sobre el ámbito ha sido un gran respaldo para lograr apreciar fundamentos teóricos de esta tecnología en auge. Es por ello que este trabajo puede aportar una visión más minuciosa y desglosada sobre este paradigma, tanto a niveles teóricos como prácticos.

En la misma línea, una de las principales herramientas utilizadas en el desarrollo de este trabajo, *Mininet WiFi*, también ha sido apenas expuesta en las soluciones que se han investigado de la escuela. De este modo, este TFG da a conocer esta novedosa herramienta, y la muestra en práctica desde el enfoque de nuestro proyecto.

Asimismo, y a diferencia del anterior, el modelo IoT es un tema mucho más renombrado, y que se ha evaluado desde distintos puntos de vista. Aun así, no se ha implementado a niveles funcionales desde la concepción de las redes definidas por software.

En consecuencia, el desarrollo de este proyecto es innovador en el área, puesto que pretende dar a conocer y exponer al público interesado todas las ventajas que nos ofrecen las SDN, además de detallar cómo las SDN pueden paliar crecimiento desbordado que supone la red IoT. A niveles pragmáticos, el TFG exhibe una solución en la que se ha diseñado y desarrollado una arquitectura basada en SDN, con el objetivo de mostrar la agilidad que ofrecen para gestionar cualquier tipo de servicios.

CAPÍTULO 3
Diseño de la solución

3.1 Análisis del problema

Las redes informáticas son el eje principal de nuestra actividad diaria y de Internet. Es incontable la cantidad de datos que circula por ellas, generando así el concepto y paradigma del *big data*. Por la naturaleza estática de la arquitectura que compone todos los dispositivos de red (routers, switches, firewalls, etc.) se acaba generando un cuello de botella, estableciendo a su vez limitaciones en cuanto a escalabilidad o automatización. En cuanto a escalabilidad, las redes tradicionales se ven acotadas debido a la dependencia que tienen sobre el hardware físico de los propios dispositivos. Por otro lado, a nivel de automatización, tienen pocas capacidades, y requieren de una cantidad significativa de intervenciones manuales. Por todo esto, el mantenimiento de una red tan amplia que está creciendo y cambiando dinámicamente se ajusta cada vez menos a las necesidades comerciales y demandas de los usuarios, siendo al mismo tiempo una tarea muy compleja el hecho de administrarla.

Respecto al paradigma IoT, la multitud de dispositivos conectados a Internet que esto implica contribuye al crecimiento de tráfico en la red y a la transmisión de datos, poniendo también a prueba la capacidad de los centros de datos y de la nube. Día tras día esta tecnología está siendo incluida a nivel personal y en todo tipo de industrias, desde la fabricación a la salud o el transporte. Según la empresa internacional *IDC:* [1] *"para 2025 habrá unos 55,9 billones de dispositivos conectados en todo el mundo, y el 75 % de ellos estarán conectados a través de una plataforma Internet of Things [21]"*. La mayor parte de todos estos datos provendrán de sistemas de seguridad y de la videovigilancia, puesto que son los tipos de archivos de mayor peso.

Observando el estado de ambos campos, se puede ver cómo supone un gran reto seguir gestionando las redes tradicionales y los dispositivos que forman parte de la red del Internet de Las Cosas.

[1] International Data Corporation (IDC) es una empresa de análisis y de investigación de mercados especializada en tecnología de la información, telecomunicaciones y tecnología de consumo. Dispone actualmente de más de 1,300 analistas de datos para estudiar y ofrecer soluciones en la industria de la tecnología.[20]

3.2 Solución y arquitectura propuesta

Una vez analizado el problema, la propuesta es diseñar e implementar una solución basada en redes definidas por software, la cual usará la técnica de microsegmentación [2] para gestionar los dispositivos IoT que forman parte de dicha red.

Para ello, se implementará una prueba de concepto compuesta por diferentes capas, donde se simulará un escenario formado por dispositivos de red que podrá ser gestionado mediante una capa de aplicación gracias a interactuar con el controlador Ryu, el cual se sitúa entre los niveles de red y de aplicación, permitiéndole así interactuar con ambas.

El escenario de red estará formado por:

Nodos IoT: nodos que simularán ser cámaras de videovigilancia situadas en distintos puntos, y enviarán tráfico de las imágenes captadas a los clientes que se indique. Estos nodos estarán conectados de forma inalámbrica (WiFi), y usarán la técnica de transmisión *multicast* para el envío de datos.

Nodos cliente: en este caso estos nodos tienen el rol de recibir el tráfico de las cámaras a las que esté suscrito.

Esta solución busca mostrar la agilidad que ofrecen las redes definidas por software para administrar y modificar el comportamiento de una red de forma dinámica y segura, exponiendo a la vez cómo podría aplicarse sobre un escenario que esté compuesto por dispositivos IoT.

A continuación se presenta la arquitectura inicial de la solución basada en SDN.

[2] La microsegmentación es una técnica de seguridad de red que permite a los arquitectos de seguridad dividir de forma lógica el centro de datos en diferentes segmentos de seguridad hasta el nivel de carga de trabajo individual. Esto permite a los administradores programar políticas de seguridad en función de dónde se pueda utilizar una carga de trabajo, a qué tipo de datos accederá, y qué importancia o sensibilidad tiene la aplicación.

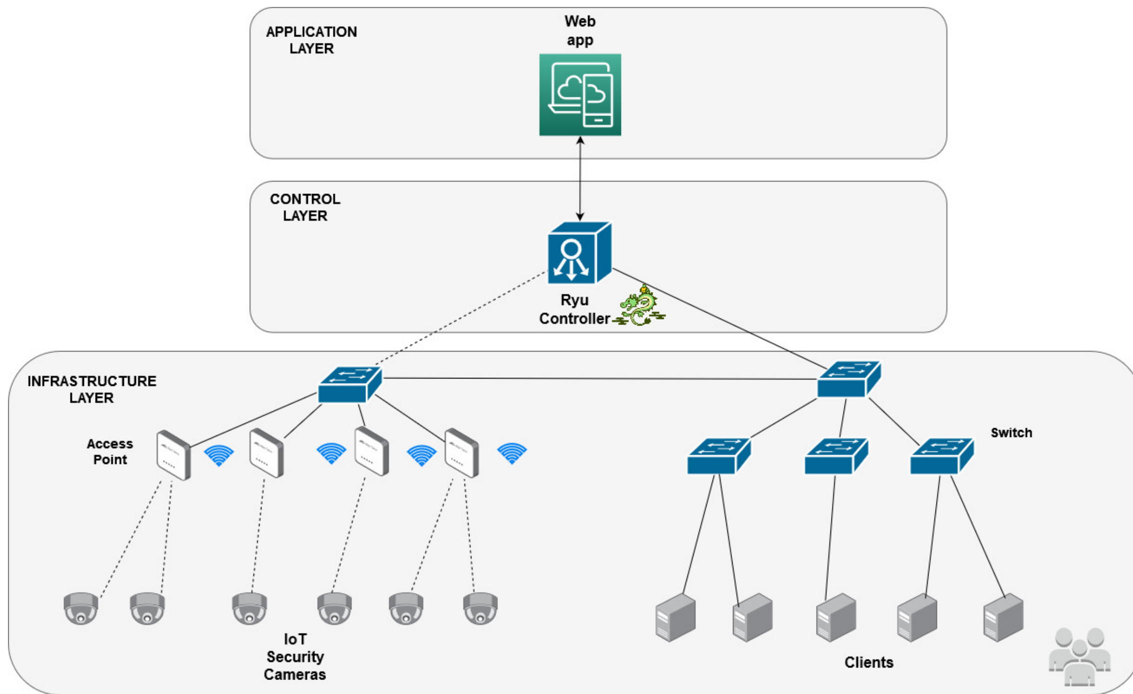


Figura 3.1: Arquitectura SDN de la solución propuesta.

Cada una de las capas de la figura 3.1 está formada por los siguientes elementos:

1. **Capa de infraestructura**: es la capa formada por cada uno de los dispositivos que participan en la red. Se simularán, por un lado, 6 cámaras de videovigilancia. Todas ellas conectadas a un AP (Access Point).

   Por el otro lado, a diferentes switches estarán conectados 5 clientes.

2. **Capa de control**: compuesta por el controlador de red *Ryu*.

3. **Capa de aplicación**: aplicación web la cual permitirá al administrador de red la flexibilidad de realizar cambios en la infraestructura. En esta solución se permitirá agregar o eliminar clientes de forma dinámica y establecer reglas basadas en horarios para recibir tráfico.

3.3 Tecnologías utilizadas

En las siguientes secciones mencionaremos las herramientas, lenguajes y sistemas de bases de datos que han sido necesarias para el diseño y desarrollo de la solución.

3.3.1. Mininet-WiFi

Mininet-WiFi[3] es un emulador para redes inalámbricas, siendo una rama de la herramienta Mininet [4], y está basado y dirigido solamente para sistemas Linux.

Mininet-WiFi amplía la funcionalidad de Mininet (capacidad de agregar hosts, switches y controladores OpenFlow), agregando puntos de acceso y estaciones WiFi virtualizadas basadas en los controladores inalámbricos estándar de Linux y el controlador de simulación inalámbrica 80211_hwsim. También permite el procesamiento de paquetes utilizando el protocolo OpenFlow.

Figura 3.2: Mininet WiFi.

Otra de sus particularidades la cual ofrece una gran flexibilidad es la capacidad de crear topologías de red personalizadas a través de su API de Python multinivel.

3.3.2. Python y el framework Flask

El lenguaje de programación Python[5] es de alto nivel, interpretado, y ampliamente utilizado para campos como aplicaciones web, *machine learning*, ciencia de datos, inteligencia artificial, *scripting* y automatización, entre muchos otros.

Actualmente es el lenguaje de programación más usado a nivel global según el ranking PYPL[6], y ello es debido a su sencillez si alguien desea iniciarse en el mundo de la programación, su alta similitud con el lenguaje humano, y el amplio abanico de ventajas que ofrece a la hora de desarrollar.

Para sacarle el máximo partido se ha usado para el desarrollo de la solución uno de sus *frameworks*: ***Flask***[7]. Flask es un módulo de Python compatible con *WSGI* que facilita el desarrollo de aplicaciones web, ofreciendo características interesantes como enrutamientos de URL, servidor web de desarrollo, depurador y soporte para pruebas unitarias, o un gran motor de plantillas, según necesidades.

[3] https://mininet-wifi.github.io/
[4] http://mininet.org/
[5] https://www.python.org/
[6] *Popularity of Programming Language Index* es el significado de sus siglas. Es una página de ránkings y estadísticas de lenguajes de programación basada en analizar cómo es de frecuente la búsqueda en Google de ellos.
[7] https://pypi.org/project/Flask/

3.3.3. XAMPP y MariaDB/MySQL

XAMPP [8] es uno de los servidores web multiplataforma más ampliamente utilizados, ya que ayuda a los desarrolladores a crear y testear sus programas en un servidor web local. Fue desarrollado por Apache Friends y su código fuente nativo puede ser revisado o modificado por la comunidad. Consta de Apache HTTP Server, MariaDB, e intérprete para los diferentes lenguajes de programación como PHP y Perl.

En nuestro proyecto ha sido una pieza clave para disponer de una base de datos en un servidor local, **MariaDB**, almacenando información de las cámaras y los clientes tanto de forma estática como dinámica (según se vaya poblando). Esta base de datos, como veremos más adelante, es constantemente consultada por la aplicación web que desarrollaremos y usaremos.

3.4 Estrategias de comunicación

Para poder establecer las comunicaciones que se dan en la arquitectura se han utilizado diferentes métodos basándose en las necesidades y limitaciones de cada capa.

3.4.1. Multicast

Las cámaras usarán la técnica *multicast* para enviar el tráfico a los clientes.

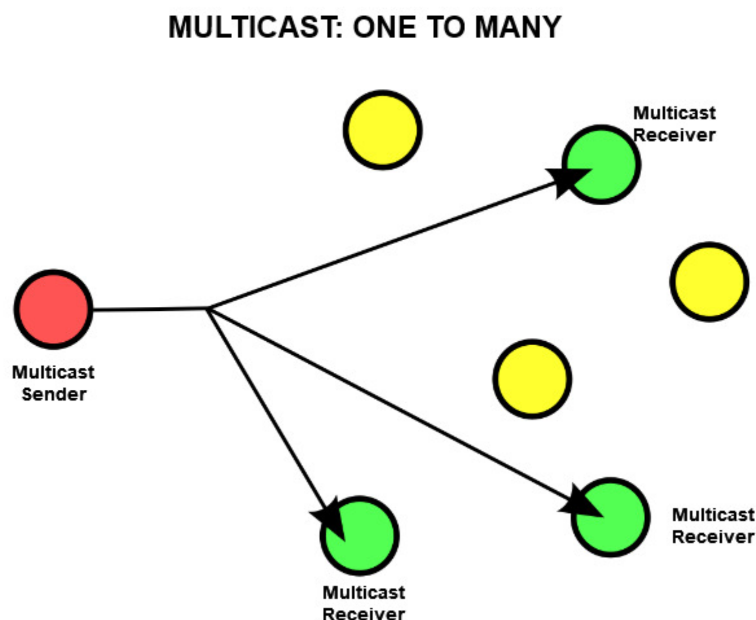


Figura 3.3: Comunicación *multicast*.

[8] https://www.apachefriends.org/es/download.html

Como se puede observar en la figura 3.3, el envío *multicast* consiste en un método de red para entregar datos de forma simultánea desde un nodo receptor a un grupo de nodos receptores. Para este tipo de comunicación se usan direcciones IP de clase D, clase que agrupa direcciones desde la 224.0.0.0 hasta la 239.255.255.255, y únicamente se le da uso a este conjunto para envíos *multicast* [22].

En la solución, la forma más eficiente de que cada cliente reciba el tráfico correspondiente solamente de las cámaras a las que tiene permiso, y no a ninguna otra, ha sido diseñar sockets *senders* y *receivers*. De este modo, podremos establecer una relación cámara-clientes siguiendo una estructura como la siguiente, en la que los clientes tendrán que unirse al grupo que les corresponda:

Camera	**Clients**
Camera 1 (224.0.0.1)	A,B,D
Camera 2 (224.0.0.2)	B,E
Camera 3 (224.0.0.3)	B,C
Camera 4 (224.0.0.4)	A

Tabla 3.1: Ejemplo de relación cámara-cliente

Vamos a ver a modo de ejemplo una parte del código de los sockets *senders*, el cual se ejecuta en las cámaras:

```
def main(multicast_group):
    multicast_ports = MULTICAST_GROUPS.get(multicast_group)
    if multicast_ports is None:
        print("Unknown multicast group.")
        return

    multicast_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.
        IPPROTO_UDP)
    multicast_socket.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 32)

    messages = {
        '224.0.0.1': b'Images from the camera 1 -> Street Ruzafa...',
        '224.0.0.2': b'Images from the camera 2 -> Food Market...',
        '224.0.0.3': b'Images from the camera 3 -> Lawyers office...',
        '224.0.0.4': b'Images from the camera 4 -> Parking - Shopping Center...',
        '224.0.0.5': b'Images from the camera 5 -> Hospital...',
        '224.0.0.6': b'Images from the camera 6 -> UPV University...'
    }

    while True:
        message = messages.get(multicast_group)
        if message is None:
            print("Unknown multicast group.")
            break

```

```
25         for port in multicast_ports:
26             multicast_socket.sendto(message, (multicast_group, port))
27             print(f"Sent message to {multicast_group}:{port}")
28 
29         time.sleep(5)
```

La función *main* definida envía los mensajes a un grupo particular de *multicast*, el cual se indicará en su lanzamiento. Al mismo tiempo, podemos ver en la línea 27 como cada mensaje, aparte de lanzarse hacia el grupo *multicast*, también se envía a un conjunto de puertos predeterminado donde estará escuchando el cliente correspondiente.

Se ha escogido la transmisión tipo *multicast*, puesto que, entre las muchas ventajas que ofrece, es la más adecuada para envíos de contenido tipo multimedia en tiempo real (debido a que usa el protocolo UDP), además de que permite un gran número de participantes receptores. Todo esto se adecua perfectamente a nuestra solución propuesta, y es fundamental para el envío de tráfico durante la ejecución.

3.4.2. APIs

El término API es una abreviatura de *Application Programming Interface*, que en español significa interfaz de programación de aplicaciones. Es un conjunto de reglas definidas que permiten que diferentes aplicaciones se comuniquen entre sí.

En nuestro caso, el uso de la API ha sido fundamental para poder implementar la comunicación entre las 3 capas, y que puedan interactuar entre ellas. En la siguiente figura 3.4 se observa la arquitectura de comunicación de nuestro escenario.

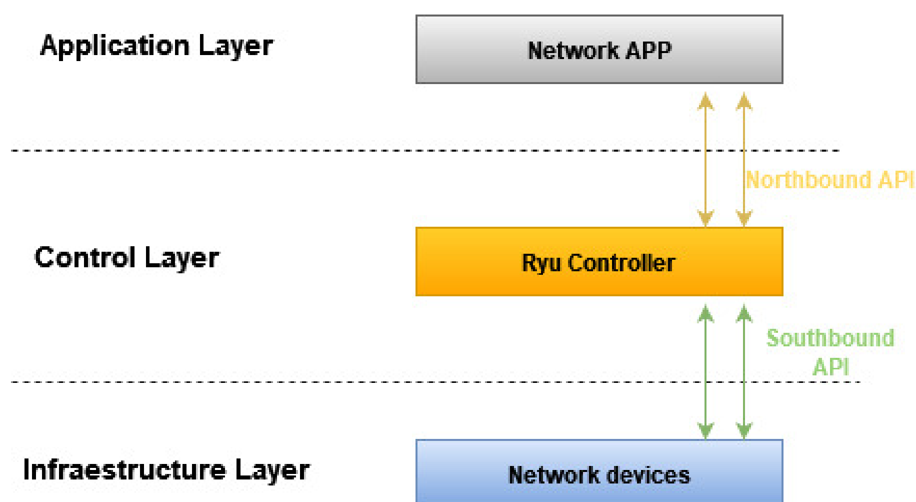


Figura 3.4: Comunicaciones en la arquitectura de la solución.

Se profundizará en cada API en el diseño correspondiente de la capa.

3.5 Diseño de las capas

En esta sección vamos a realizar el diseño de las diferentes capas de red relevantes para este trabajo: infraestructura, control y aplicación.

3.5.1. Diseño capa de infraestructura

La arquitectura de esta capa la compone la topología de red de la solución, por tanto, vamos a analizar el diseño de la red de las cámaras, clientes, y el envío de flujos que se da en toda la estructura.

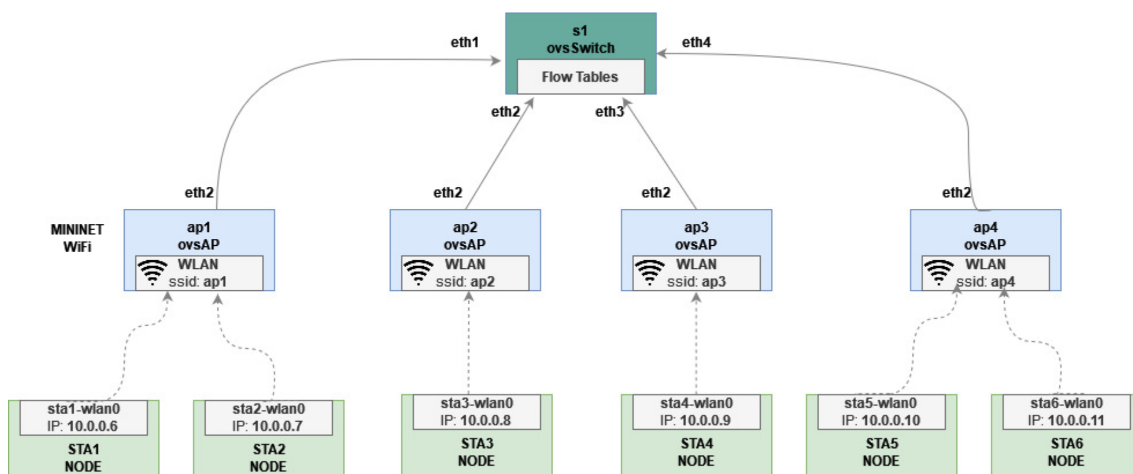


Figura 3.5: Arquitectura de red de las cámaras.

En la imagen 3.5 vemos representada la arquitectura de red relativa a los nodos que actúan como cámaras. En el nivel de los nodos tenemos todas las estaciones conectadas de forma inalámbrica al punto de acceso WiFi correspondiente. Vemos, por otro lado, que aparece el nombre de la interfaz de red y la IP asignada, por ejemplo en la estación 1: *sta1-wlan0* con IP *10.0.0.6*.

En cuanto a los AP, se observa que están conectados físicamente a un switch, y que este último gestiona el tráfico en este segmento de red, basándose en las *flow tables*, tablas donde se encuentran las reglas que añadimos, modificamos y/o eliminamos mediante el protocolo *OpenFlow*.

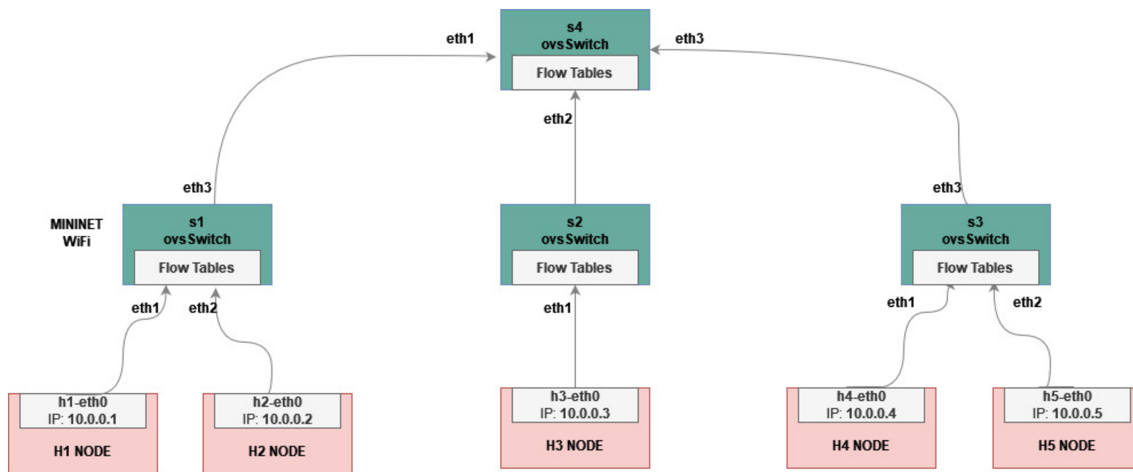


Figura 3.6: Arquitectura de red de los clientes.

Paralelamente, nos encontramos con el otro lado de la topología en el diagrama que se muestra en la figura 3.6. Ahora contemplamos que, en el nivel de los nodos cliente, en vez de estar conectados a los AP, lo están directamente a switches. No olvidemos que el switch del primer nivel estará conectado al switch de primer nivel de la topología de las cámaras, como hemos representado previamente en la figura 3.1; esto garantiza que ambas topologías se 'vean', y sea posible la comunicación aún estando el controlador de por medio en la capa de control.

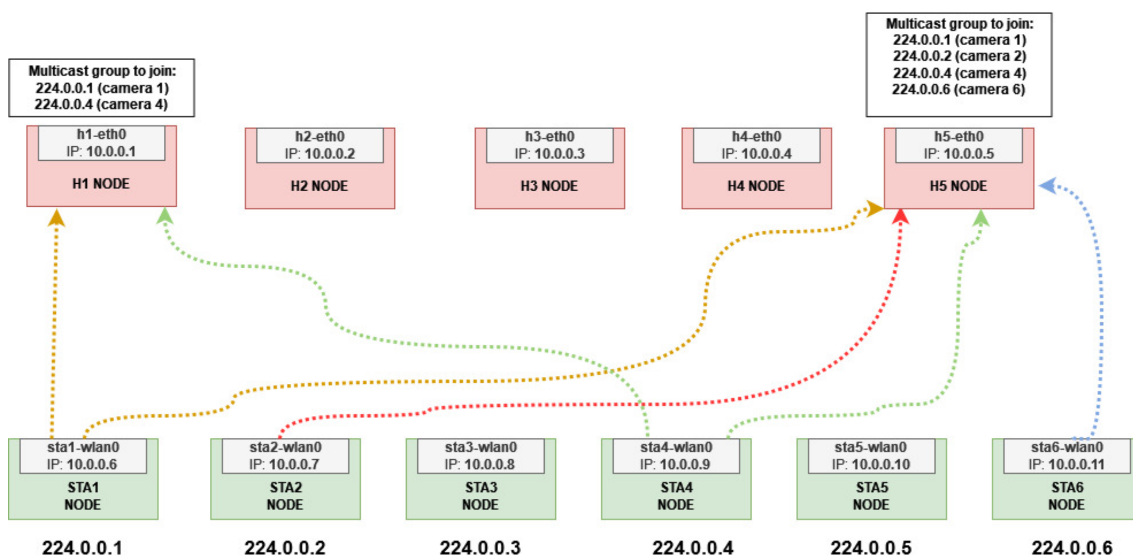


Figura 3.7: Flujo *multicast* de los clientes 1 y 5.

Por último, tenemos a modo de ejemplo una representación del flujo que se transmite en la red desde las diferentes cámaras hacia los clientes 1 y 5, ver figura 3.7, y todo esto por supuesto pasando previamente por los dispositivos de red correspondientes. Esta representación se basa en la relación cámara-cliente que estableceremos para la solución base que tendremos inicialmente. El flujo *multicast* que se envía está regido acorde a la siguiente relación:

Camera	Clients
Camera 1 - Street Ruzafa (224.0.0.1)	1 - Traffic Monitor SL 3 - IA Face Recognition Citizens 5 - Police
Camera 2 - Food Market (224.0.0.2)	3 - IA Face Recognition Citizens 5 - Police
Camera 3 - Lawyer's office (224.0.0.3)	2 - Security Branches SL 4 - IT Team Lawyers' Office
Camera 4 - Parking - Shopping Center (224.0.0.4)	1 - Traffic Monitor SL 5 - Police
Camera 5 - Hospital (224.0.0.5)	3 - IA Face Recognition Citizens
Camera 6 - UPV University (224.0.0.6)	5 - Police

Tabla 3.2: Ejemplo de relación cámara-cliente inicial

Para que el tráfico que envían las cámaras sea gestionable, y podamos decidir quién lo recibe y quién no, ha sido necesario asignar a cada cliente seis puertos (tantos puertos UDP como cámaras disponibles), donde en cada uno de ellos *escuchará* al grupo *multicast* (flujo de vídeo) correspondiente. Las principales razones por las que se ha escogido esta técnica han sido las siguientes, basándose en que se ha de poder aceptar/denegar tráfico dinámicamente mediante reglas:

1. No podemos denegar todo el tráfico a un host (cliente), ya que tiene que seguir recibiendo datos de otras cámaras. Por ello se descarta la opción de crear una regla basada en denegar el tráfico según la MAC destino.

2. No podemos denegar todo el tráfico proveniente de un grupo *multicast*, debido a que también tiene que enviarlo a otros clientes. Por tanto, no es factible implementar reglas basadas en la MAC origen del grupo *multicast*.

Teniendo en cuenta esto, la opción eficaz y a su vez eficiente es utilizar reglas basadas en puertos de la capa de transporte (UDP). A la par, este criterio mejora la seguridad del sistema, ya que cada cliente tiene un puerto distinto asignado, y al mismo tiempo los grupos *multicast* enviarán los datos a un máximo de siete clientes (siete puertos determinados), lo que evita que cualquiera que no deba pueda unirse al grupo. De cualquier modo, esto siempre podría ser escalable.

En la siguiente imagen 3.8 vemos los puertos asignados a cada grupo. La nomenclatura que se sigue para definir el número de puerto UDP es: 50+nº cliente+nº cámara. Por ejemplo, el cliente nº4 recibirá el tráfico *multicast* de la cámara nº6 en el puerto 5046. Obviamente, se podrían usar otros criterios más complejos en caso de ser necesario escalar a un gran número de clientes y de cámaras.

```
 9  MULTICAST_PORTS_CAMERA1 = [5011, 5021, 5031, 5041, 5051, 5061, 5071]
10  MULTICAST_PORTS_CAMERA2 = [5012, 5022, 5032, 5042, 5052, 5062, 5072]
11  MULTICAST_PORTS_CAMERA3 = [5013, 5023, 5033, 5043, 5053, 5063, 5073]
12  MULTICAST_PORTS_CAMERA4 = [5014, 5024, 5054, 5044, 5054, 5064, 5074]
13  MULTICAST_PORTS_CAMERA5 = [5015, 5025, 5035, 5045, 5055, 5065, 5075]
14  MULTICAST_PORTS_CAMERA6 = [5016, 5026, 5036, 5046, 5056, 5066, 5076]
```

Figura 3.8: Ejemplo de asignación de puertos para el envío de tráfico *multicast*.

3.5.2. Diseño capa de control

Como se ha comentado en capítulos previos, esta capa actúa como el cerebro de todo el conjunto de la arquitectura SDN y gestiona todas las políticas y flujos de tráfico de toda la red.

En nuestra solución se ha optado por usar una aplicación que nos ofrece Ryu denominada *REST Firewall* [23], la cual nos permitirá gestionar el tráfico mediante reglas de firewall: permitir o bloquear paquetes basándose en la cabecera de estos (origen/destino de la dirección MAC, dirección IPv4/IPv6, número de puerto, protocolo usado, etc.). Todas las reglas están basadas en el protocolo *OpenFlow*, haciendo *match* con todos sus campos.

La clave principal de este controlador elegido es que el administrador de red puede acceder, insertar, eliminar o modificar políticas del firewall a través de una API REST.

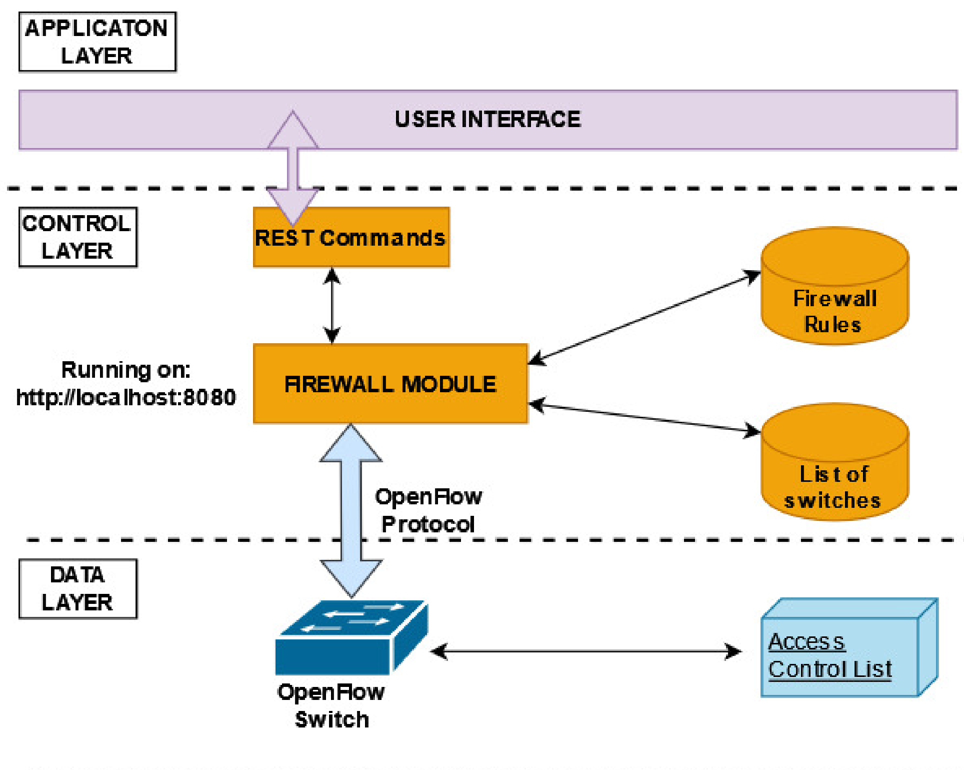


Figura 3.9: Estructura y diseño de la capa de control

Como se ilustra en la figura 3.9 la aplicación del firewall consiste en 4 componentes: el mismo módulo, comandos REST, una lista de switches de toda la red y otra con todas las reglas.

El **módulo del firewall** es el corazón de la aplicación, el cual se coordina con el controlador para implementar las reglas del firewall en los dispositivos de red. Está constantemente monitorizando toda la actividad y listas de accesos de los switches, observando que no sea modificada por ningún sistema externo y/o interno, y desviando el tráfico en caso de detectar algún cambio inesperado.

Las dos **listas** que mantiene el módulo, **reglas y switches**, ayudan a mantener las políticas de seguridad en toda la red. Por un lado, en la lista cada switch está definida por un identificador único, ofreciendo al administrador de la red la posibilidad de añadir reglas específicamente en uno de ellos o en todos. Del primer modo, se reducen las reglas redundantes y se establecen diferentes niveles de seguridad en distintos segmentos de la red.

Los **comandos REST** darán pie a realizar peticiones HTTP (GET, POST, PUT, DELETE) para leer, escribir, modificar y eliminar datos de toda la infraestructura de red. El administrador enviará peticiones desde la aplicación web al controlador y este último las decodificará para realizar los cambios correspondientes. Vemos a continuación en la tabla la lista de posibilidades que ofrece la API y que usaremos:

Acción	**Método**	**URL**
Ver estado de todos los switches	GET	/firewall/module/status
Conectar/desconectar switch a la red	PUT	/firewall/module/{op}/{switch}
Ver las reglas de toda la red	GET	/firewall/rules/{switch}[/{vlan}]
Añadir regla	POST	/firewall/rules/{switch}[/{vlan}]
Eliminar regla	DELETE	/firewall/rules/{switch}[/{vlan}]
Adquirir LOGs de los dispositivos	GET	/firewall/log/status

Tabla 3.3: Lista de operaciones API del controlador

3.5.3. Diseño capa de aplicación

Esta última capa ofrecerá al usuario una interfaz para interactuar y gestionar la infraestructura de red.

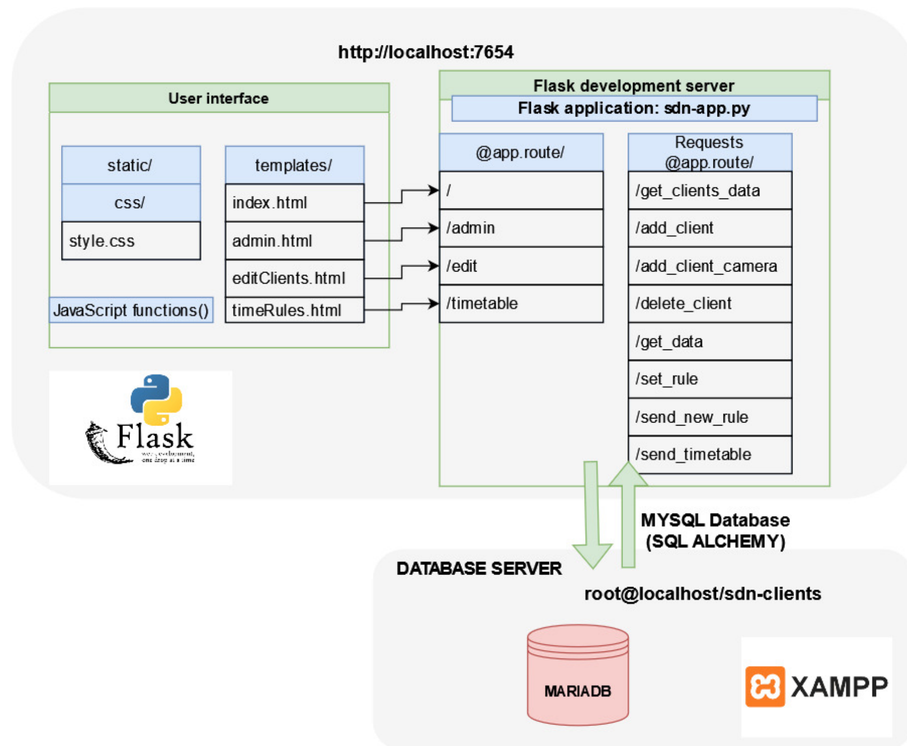


Figura 3.10: Diseño arquitectura aplicación web.

En la figura 3.10 se ilustra la arquitectura de la capa de aplicación, donde se pueden ver 3 partes bien diferenciadas:

1. El **servidor de bases de datos**, ejecutándose en otro puerto del servidor, y guardando la información en tiempo real de los cambios que se realizan sobre los nodos de la infraestructura. Es un pilar clave puesto que permite a la aplicación consultarla para poder realizar cambios en la topología y conocer su estado.

   Vamos a ver en la siguiente figura la pequeña estructura de la base de datos que recogerá la información de las relaciones cámara-cliente.

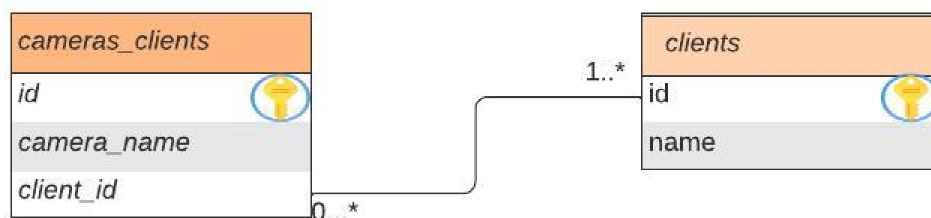


Figura 3.11: Tablas de la base de datos.

El diseño mostrado en la imagen 3.11 ilustra las dos tablas que componen la base de datos. Se ha optado por no tener una tabla solo con las cámaras, puesto que es siempre estática (el administrador nunca podrá añadir/eliminar cámaras) y las incluimos directamente en la tabla *cameras_clients*, la cual representa la asociación con sus respectivos clientes.

2. El **front-end** de la aplicación (*user interface*), interfaz gráfica compuesta por sus correspondientes ficheros y con la cual interactuará el administrador de red. En la siguiente figura 3.12 podemos observar el diseño de algunas interfaces con las que se encontrará el administrador.

ADD A CLIENT TO A CAMERA GROUP

Enter ID Client | Select a camera | Add

Camera	Client
1 - Street Ruzafa	Traffic Monitor SL IA Face Recognition Citizens Police
2 - Food Market	IA Face Recognition Citizens Police
3 - Lawyers' office	Security Branches SL IT Team Lawyers' Office
4 - Parking - Shopping Center	Traffic Monitor SL Police
5 - Hospital	IA Face Recognition Citizens
6 - UPV University	Police

Send Info

(a) Interfaz 'añadir cliente a cámara'

SDN APP

Camera	Timetable
1 - Street Ruzafa	-- : -- / -- : -- Send
2 - Food Market	dd / mm / aaaa , -- : -- / dd / mm / aaaa , -- : -- Send
3 - Lawyers' office	dd / mm / aaaa , -- : -- / dd / mm / aaaa , -- : -- Send
4 - Parking - Shopping Center	dd / mm / aaaa , -- : -- / dd / mm / aaaa , -- : -- Send
5 - Hospital	Not modifiable
6 - UPV University	-- : -- / -- : -- Send

(b) Interfaz 'establecer horarios en cámara'

Figura 3.12: Diseño de interfaces

3. El **back-end** (*Flask application*), quien agrega toda la lógica a la aplicación web. Gracias al *framework Flask* podemos ejecutar la aplicación sobre un servidor local (`http://localhost:7654` en nuestro caso) e ir depurándola sobre la marcha. El fichero *sdn-app.py* está compuesto por todas las rutas para

poder lanzar la aplicación y, al mismo tiempo, con todas las funciones que harán las peticiones a la API del controlador para actualizar la red.

Por otro lado, es el encargado de comunicarse con la **base de datos** que, dinámicamente, irá poblando sus tablas con la información de cámaras y clientes, en base a los cambios que realice la aplicación web.

Como punto final, observemos como ejemplo cuál sería el proceso de comunicación en la arquitectura si el administrador realizase la operación de **añadir un cliente a una cámara** desde la aplicación web.

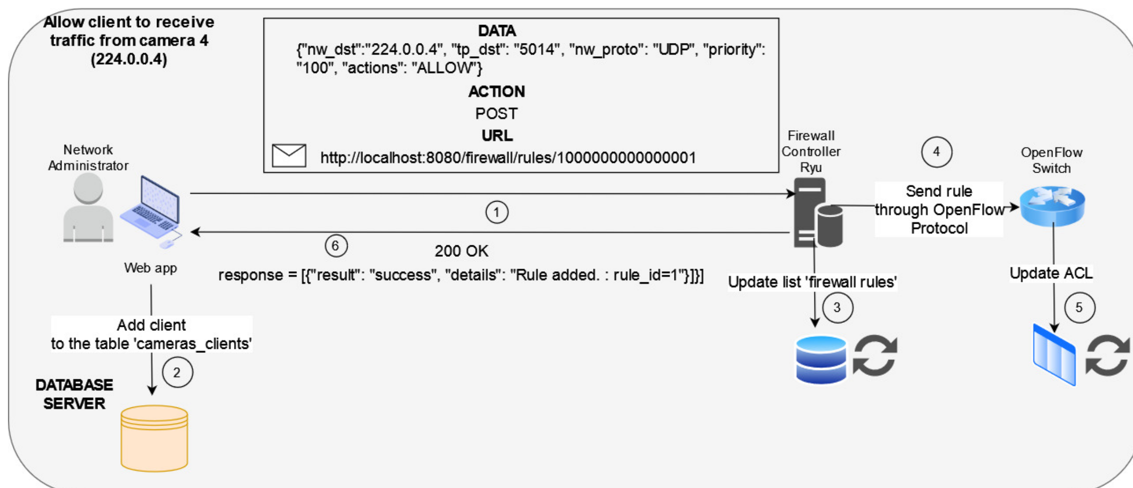


Figura 3.13: Ejemplo de proceso al añadir una regla desde la app web.

La figura 3.13 ilustra, por un lado, el envío de la petición POST para aceptar el tráfico *multicast* del grupo 224.0.0.4 hacia el cliente 1, con su correspondiente formato en JSON. Primero actualizará la base de datos actualizando la tabla de relación cámara-cliente. Luego, podemos ver como el controlador procesa la petición, lo ordena al dispositivo de red correspondiente, y finalmente envía una respuesta HTTP avisando del éxito de la petición (200 OK).

CAPÍTULO 4
Desarrollo de la solución propuesta

El siguiente paso de este proyecto es pasar a la fase de desarrollo e implementación de todas las adaptaciones que hemos diseñado hasta ahora. Este capítulo se centrará en explicar los pasos que se han seguido para poner en marcha la solución, al mismo tiempo que describimos dificultades encontradas, y justificamos las decisiones que se han tomado.

Este capítulo lo dividiremos en tres secciones. En la primera de ellas, ilustraremos cómo se ha llegado hasta la solución base del trabajo. A continuación, en las dos secciones siguientes, se mostrarán las dos mejoras en cuanto a opciones de administración de la red que se han añadido sobre la solución base.

4.1 Solución inicial

4.1.1. Simulación de la topología y envío de tráfico

Siguiendo el orden de la arquitectura de la aplicación, el primero paso es realizar la simulación del escenario. La topología se ha desarrollado en su totalidad mediante código Python. Mediante dicho código creamos y personalizamos todos los dispositivos que forman parte de la red y también su comportamiento.

```
1 ap1 = net.addAccessPoint('ap1', cls=OVSKernelAP, ssid='ap1-ssid', mode='g',
      channel='5', position='25,50,0', range='70')
2 sta1 = net.addStation('sta1', mac='00:00:00:00:00:06', ip='10.0.0.6/8', position=
      '50,50,0')
3 sta2 = net.addStation('sta2', mac='00:00:00:00:00:07', ip='10.0.0.7/8', position=
      '20,50,0')
```

A modo de ejemplo, podemos ver en el código superior cómo definimos un *Access Point* en la línea 1, y dos estaciones inalámbricas en las dos siguientes líneas. En ambos es modificable su posición; en las estaciones podemos también personalizar tanto la dirección MAC como la IP. Del mismo modo se implementa de forma muy parecida los hosts que se conectan de forma física y los switches.

Una vez puesto el escenario en marcha con todos los dispositivos creados, y establecidas las diferentes relaciones entre ellos, como hemos diseñado previa-

mente, pasamos a poder permitir el tráfico *multicast* que se enviará posteriormente. Para ello es necesario que los hosts conozcan las rutas y las tengan añadidas a nivel interno. Agregar las rutas a cada nodo del escenario es posible gracias a los *network namespaces*[1] que nos ofrece la herramienta de Mininet WiFi al estar basada en el kernel de Linux. En el siguiente código vemos como añadimos individualmente a cada host el prefijo de red *multicast* a su correspondiente interfaz.

```
net.get('h1').cmd('route add -net 224.0.0.0 netmask 240.0.0.0 dev h1-eth0')
net.get('h2').cmd('route add -net 224.0.0.0 netmask 240.0.0.0 dev h2-eth0')
net.get('h3').cmd('route add -net 224.0.0.0 netmask 240.0.0.0 dev h3-eth0')
net.get('h4').cmd('route add -net 224.0.0.0 netmask 240.0.0.0 dev h4-eth0')
net.get('h5').cmd('route add -net 224.0.0.0 netmask 240.0.0.0 dev h5-eth0')
```

Con el escenario en marcha y las configuraciones realizadas, podemos pasar a añadir el controlador de red.

4.1.2. Añadiendo el controlador de red al escenario

El siguiente paso es incluir el controlador de red en la arquitectura para que empiece a gestionar todo el tráfico de la topología, y posteriormente se comunique al mismo tiempo con la capa de aplicación.

Una de las primeras dificultades que nos encontramos después de comprobar que el controlador se comunicaba correctamente con la red fue integrarlo con la topología para que se iniciase desde el principio con ella, automatizando así al máximo el conjunto de la arquitectura. La solución la encontramos en el siguiente código.

```
with open('log_firewall.txt', 'w') as f:
    c0 = subprocess.Popen(['ryu-manager', 'ryu.app.rest_firewall'], stdout=f)
c0.wait()
time.sleep(5)

info("Ryu app running!\n")

print(net.get('c0').cmd('curl -X PUT http://localhost:8080/firewall/module/enable
    /1000000000000001'))
print(net.get('c0').cmd('curl -X PUT http://localhost:8080/firewall/module/enable
    /0000000000000004'))

net.get('c0').cmd('curl -X POST -d \'{"nw_dst": "224.0.0.1", "nw_proto": "UDP"}\'
     http://localhost:8080/firewall/rules/0000000000000004')
net.get('c0').cmd('curl -X POST -d \'{"nw_dst": "224.0.0.2", "nw_proto": "UDP"}\'
     http://localhost:8080/firewall/rules/0000000000000004')
net.get('c0').cmd('curl -X POST -d \'{"nw_dst": "224.0.0.3", "nw_proto": "UDP"}\'
     http://localhost:8080/firewall/rules/0000000000000004')
```

[1] Los espacios de nombres de red Linux son una característica del núcleo de Linux que nos permite aislar los entornos de red mediante la virtualización; por tanto, cada uno dispone su propio conjunto de interfaces, direcciones IP, tablas de enrutamiento, etc.

```
14 net.get('c0').cmd('curl -X POST -d \'{"nw_dst": "224.0.0.4", "nw_proto": "UDP"}\'
       http://localhost:8080/firewall/rules/0000000000000004')
15 net.get('c0').cmd('curl -X POST -d \'{"nw_dst": "224.0.0.5", "nw_proto": "UDP"}\'
       http://localhost:8080/firewall/rules/0000000000000004')
16 net.get('c0').cmd('curl -X POST -d \'{"nw_dst": "224.0.0.6", "nw_proto": "UDP"}\'
       http://localhost:8080/firewall/rules/0000000000000004')
```

Por un lado, en el primer bloque de código (de la línea 1 a la 4) nos encontramos con el lanzamiento del controlador en *background*, guardando toda la información que va generando en el fichero *log_firewall.txt*. En dicho fichero encontraremos todo el flujo que se mueve por la red, tanto paquetes permitidos como bloqueados.

Por el otro, en las líneas 8 y 9, una vez ya iniciado el controlador, tenemos que comunicarnos con él, y es por ello que lo hacemos con su API a través de peticiones *curl*. Iniciamos los módulos de firewall en los dispositivos de red principales, donde enviaremos todas las reglas para que se apliquen al segmento de red correspondiente.

Finalmente, en el resto de código, le comunicamos al controlador que tenga presente los seis grupos *multicast* que van a estar enviando flujo hacia los clientes.

Con todo esto, ya tendríamos preparado el escenario para que más adelante se empiece a enviar el tráfico de las cámaras hacia los clientes, y el controlador para recibir y enviar órdenes tanto de la capa de datos como de la capa de aplicación.

4.1.3. Aplicación web

En última instancia, nos encontramos con el desarrollo e implementación de la aplicación web con la que interactuará el administrador de red.

Para esta aplicación, como se ha comentado previamente, se ha desarrollado toda su lógica también en Python. Lo principal fue programar la interfaz inicial, la cual es la primera que aparece al iniciar la aplicación, y tiene como objetivo mostrar las relaciones cámara-cliente preexistentes, y, por tanto, enviar esa información y aplicarla a la topología de red. Sin esta interfaz no sería posible que empiece a fluir el tráfico de las cámaras hacia los clientes, ya que se envían todas las reglas permitiendo y/o denegando el tráfico correspondiente.

El primer paso en cuanto a la interfaz comentada fue crear las bases de datos con las relaciones cámara-cliente que queremos presentar en pantalla, y que inicialmente son estáticas. Posteriormente, se dará la opción de modificar clientes suscritos a la cámara que se desee. Este paso fue unos de los que más tiempo llevó, puesto que previamente nunca había creado bases de datos ni sus respectivas tablas desde código Python, para luego insertarlas en un servidor externo.

Con esto creado, el paso posterior fue que la aplicación recogiese dicha información de las cámaras y clientes, y la representase en la interfaz. Esto también causó conflicto a la hora de conectarse a la base de datos y a la tabla correspondiente para recoger la información de los nodos, y representarlos en el formato que se deseaba.

Finalmente, el botón clave de la ventana principal de la aplicación es el que envía la información que se ha de aplicar a la topología, y para ello debe comunicárselo al controlador de red.

```
@app.route('/send_rules')
def set_rule():
    api_url_sw = "http://localhost:8080/firewall/rules/0000000000000004"
    api_url_ap = "http://localhost:8080/firewall/rules/1000000000000001"
    #AP : CAMARA 1 (STA1) + CLIENTE 1,3,5
    camera1_client1 = {"nw_dst":"224.0.0.1", "tp_dst": "5011", "nw_proto": "UDP",
        "priority": "100", "actions": "ALLOW"}
    camera1_client3 = {"nw_dst":"224.0.0.1", "tp_dst": "5031", "nw_proto": "UDP",
        "priority": "100", "actions": "ALLOW"}
    camera1_client5 = {"nw_dst":"224.0.0.1", "tp_dst": "5051", "nw_proto": "UDP",
        "priority": "100", "actions": "ALLOW"}

    response1 = requests.post(api_url_ap, json=camera1_client1)
    print(response1.content)
```

Como podemos observar en el código insertado, tenemos un ejemplo de cómo se envían las reglas a la API del controlador en formato JSON de la cámara número uno, y de los respectivos clientes suscritos a ella. El módulo del firewall aplicará estas reglas recibidas, actualizará la topología en base a los cambios indicados, y el flujo que se transmite por la red se actualiza dinámicamente.

4.2 Funcionalidades avanzadas

En esta sección se detalla cómo se han implementado funcionalidades más complejas, incluyendo la gestión de clientes en caliente, y la introducción de permisos con una granularidad temporal muy fina.

4.2.1. Gestión dinámica de clientes

Con la solución base ya establecida, la cual permite que una aplicación gestione y regule el tráfico permitido en la red, esta opción avanzada consiste en que el administrador pueda eliminar clientes en su totalidad (no se le permita recibir tráfico de ninguna de las cámaras que tiene autorizadas), o añadir clientes nuevos a las cámaras ya existentes.

Para ello ha sido necesario, por un lado, definir funciones que añadan nuevos datos al servidor de bases de datos desde la aplicación web, y así puedan estar sincronizados ambos a la misma vez que se retroalimentan. En su defecto, lo mismo ocurre de cara a eliminar clientes.

En la misma línea, sería lo mismo para la segunda opción, la cual permite añadir clientes a una cámara en específico.

Paralelamente a la actualización de la base de datos, también se le indica al controlador qué puertos de red se deben permitir/denegar dependiendo de si

se añaden y/o eliminan clientes, lo que se reflejará en el volumen de tráfico que reciba cada cliente.

4.2.2. Restricciones de tráfico con granularidad temporal fina

Como última funcionalidad avanzada, se desarrolla y se le da al administrador de red la opción de establecer horarios para el envío de flujo por parte de las cámaras. Es decir, a cierta hora o día del mes/semana, los clientes recibirán (o no) el tráfico que envían las cámaras basándose en las restricciones que se apliquen.

Esta parte del desarrollo ha sido la más complicada, puesto que desde el principio supuso dificultad el decidir cómo enfocarlo e implementarlo. En primer lugar, se planteó la idea de modificar el código del módulo del controlador [24], y para ello se estuvo analizando dicho código a fondo. Se descubrió que existen dos parámetros en los paquetes del protocolo *OpenFlow* que podrían ayudarnos a eliminar un flujo basado en restricciones de tiempo: *hard timeout* e *idle timeout*. Modificando dichos indicadores podemos indicar, en segundos, cuándo deseamos que expire dicha regla. Sin embargo, intentar modificar estas variables directamente del código alteraba totalmente el comportamiento del controlador.

La solución que se le encontró finalmente fue definir un par de funciones sobre el código de la aplicación, las cuales programan el envío de reglas a la API del controlador.

```
1  scheduler = sched.scheduler(time.time, time.sleep)
2  current_datetime = datetime.now()
3  print(current_datetime)
4
5  if initial_datetime_str:
6      try:
7          initial_datetime = datetime.strptime(initial_datetime_str, "%Y-%m-%dT%H
               :%M")
8      except ValueError:
9          initial_datetime = datetime.combine(datetime.today().date(), datetime.
               strptime(initial_datetime_str, "%H:%M").time())
10
11     current_datetime = current_datetime.replace(second=0, microsecond=0)
12     if initial_datetime > current_datetime:
13         print(initial_datetime)
14         print(current_datetime)
15         send_end_message()
16         scheduler.enterabs(initial_datetime.timestamp(), 1, send_initial_message)
```

El código superior insertado muestra la definición de una de las funciones que se ha desarrollado para planificar el envío de reglas. En la línea 1 se observa el uso de la biblioteca *sched*, nativa de Python, la cual proporciona la posibilidad de programar eventos, siendo clave en nuestra solución para fijar el envío de reglas en dicho segundo, minuto, hora o fecha.

El resto de la función se apoya también en bibliotecas basadas en tiempo (*time* y *datetime*) las cuales ayudan a manejar los datos que enviará el usuario en cierto formato y transformarlos.

De la línea 5 a la 9 comprueba si se han insertado una hora o fecha inicial (ya que ha de introducirse una inicial y otra de finalización) y lo transforma al formato correspondiente. De no ser así, no dará por válida la inserción.

Finalmente, con esta combinación de bibliotecas y comprobaciones de lógica que implementamos conseguimos programar la función *send_initial_message*, la cual le enviará la regla correspondiente a la API del controlador a la hora o fecha deseada.

CAPÍTULO 5
Pruebas de validación

En este último capítulo se presentarán diferentes pruebas para verificar que la solución implementada en este trabajo funciona según esperado. Igualmente, se analizará el rendimiento de la aplicación en término de volumen de tráfico, tiempos de ejecución, carga de tráfico, etc.

5.1 Registro de nuevas reglas

Para empezar, vamos a mostrar que las reglas que enviamos desde el nivel de aplicación llegan correctamente al controlador y se aplican sobre la topología. Para ello, expondremos a continuación el estado de cada capa, y qué comunicaciones realizan en cada momento.

```
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$ date && curl http://localhost:8080/firewall/rules/1000000000000001/all
Tue 13 Jun 2023 04:04:31 PM CEST
[{"switch_id": "1000000000000001", "access_control_list": []}]sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$
```

Figura 5.1: Comprobación de reglas de tráfico en el dispositivo.

En primer lugar, en la figura 5.1, consultamos mediante el comando *'date && curl http://localhost:8080/firewall/rules/1000000000000001/all'* la fecha y hora actual y mediante una petición a la API del controlador de qué reglas dispone el dispositivo de red correspondiente en ese momento. La salida nos muestra lo siguiente:

```
Tue 13 Jun 2023 04:04:31 PM CEST
[{"switch_id": "1000000000000001", "access_control_list": []}]
```

Lo cual nos indica que el switch no dispone de reglas, puesto que la lista *access_control_list* está vacía.

```
sdn@ubuntuserver:~/mininet-wifi/pruebas$
sdn@ubuntuserver:~/mininet-wifi/pruebas$ cat log_firewall.txt | grep POST | grep 1000000000000001
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000723
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000628
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000607
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000485
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000466
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000415
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000439
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000537
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 249 0.000403
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 250 0.000398
127.0.0.1 - - [13/Jun/2023 16:13:40] "POST /firewall/rules/1000000000000001 HTTP/1.1" 200 250 0.000402
sdn@ubuntuserver:~/mininet-wifi/pruebas$
```

Figura 5.2: Log del controlador de red.

Acto seguido, procedemos a enviar desde la aplicación las reglas de tráfico, las cuales llegarán a la API del controlador y vemos en la figura 5.2 que este último las recibe correctamente y lo refleja a través de su log.

```
sdn@ubuntuserver:~/mininet-wifi/pruebas$ date & curl http://localhost:8080/firewall/rules/1000000000
00001/all
Tue 13 Jun 2023 04:15:00 PM CEST
[{"switch_id": "1000000000000001", "access_control_list": [{"rules": [{"rule_id": 1, "priority": 100,
"dl_type": "IPv4", "nw_dst": "224.0.0.1", "nw_proto": "UDP", "tp_dst": 5011, "actions": "ALLOW"}, {"r
le_id": 2, "priority": 100, "dl_type": "IPv4", "nw_dst": "224.0.0.1", "nw_proto": "UDP", "tp_dst": 50
1, "actions": "ALLOW"}, {"rule_id": 3, "priority": 100, "dl_type": "IPv4", "nw_dst": "224.0.0.1", "nw
proto": "UDP", "tp_dst": 5051, "actions": "ALLOW"}, {"rule_id": 4, "priority": 100, "dl_type": "IPv4"
 "nw_dst": "224.0.0.2", "nw_proto": "UDP", "tp_dst": 5032, "actions": "ALLOW"}, {"rule_id": 5, "prior
ty": 100, "dl_type": "IPv4", "nw_dst": "224.0.0.2", "nw_proto": "UDP", "tp_dst": 5052, "actions": "AL
OW"}, {"rule_id": 6, "priority": 100, "dl_type": "IPv4", "nw_dst": "224.0.0.3", "nw_proto": "UDP", "t
_dst": 5023, "actions": "ALLOW"}, {"rule_id": 7, "priority": 100, "dl_type": "IPv4", "nw_dst": "224.0
0.3", "nw_proto": "UDP", "tp_dst": 5043, "actions": "ALLOW"}, {"rule_id": 8, "priority": 100, "dl_typ
": "IPv4", "nw_dst": "224.0.0.4", "nw_proto": "UDP", "tp_dst": 5014, "actions": "ALLOW"}, {"rule_id":
9, "priority": 100, "dl_type": "IPv4", "nw_dst": "224.0.0.4", "nw_proto": "UDP", "tp_dst": 5054, "act
ons": "ALLOW"}, {"rule_id": 10, "priority": 100, "dl_type": "IPv4", "nw_dst": "224.0.0.5", "nw_proto"
 "UDP", "tp_dst": 5035, "actions": "ALLOW"}, {"rule_id": 11, "priority": 100, "dl_type": "IPv4", "nw_
st": "224.0.0.6", "nw_proto": "UDP", "tp_dst": 5056, "actions": "ALLOW"}]}]}]sdn@ubuntuserver:~/minin
```

Figura 5.3: Dispositivo de red tras recibir reglas.

En último lugar, volvemos a comprobar cuál es el estado del dispositivo de red para confirmar que ha recibido las reglas desde la capa de aplicación. La imagen anterior 5.3 nos confirma que ahora sí, su lista *access_control_list* ha sido actualizada.

5.2 Pruebas de carga

A continuación mostraremos distintas pruebas de tráfico y tiempo que se dan en la red para comprobar la eficiencia y el consumo de recursos de la topología.

5.2.1. Carga individual sobre un cliente

En este apartado se procede a observar cuál es el rendimiento de la conexión UDP *multicast* al recibir datagramas en un cliente enviados por un nodo cámara. Para ello usaremos el comando de Linux *iperf*, herramienta que permite realizar pruebas de ancho de banda cliente/servidor y nos ofrece varios parámetros de prestaciones del resultado de la conexión, tales como *jitter*[1] y pérdida de paquetes.

Ejecutaremos los siguientes comandos:

[1] El jitter mide las variaciones en el ping en una línea temporal. Si el valor es alto, quiere decir que se producen interrupciones en la conexión.

```
// En el nodo cliente:
iperf -s -u -B 224.0.0.1 -p 5031 -i 1
// En el nodo camara:
iperf -c 224.0.0.1 -u -B 224.0.0.1 -p 5031 -i 5 -t 10
```

El comando de la línea 2 inicia un servidor en modo UDP, escuchando en la dirección IP *multicast* 224.0.0.1 y el puerto 5031. Por el otro lado, el nodo cámara en la línea 4 se conecta en modo UDP a la dirección IP 224.0.0.1 y puerto 5031, donde enviará paquetes durante 10 segundos.

Los resultados obtenidos por el cliente son los siguientes:

Tabla 5.1: Resultados de la prueba cliente/servidor de datagramas UDP

Interval	Transfer (KBytes)	Bandwidth (KBits/sec)	Jitter (ms)	Lost/Total Datagrams
0.0-1.0	122	1000	14.023	0/85 (0 %)
1.0-2.0	111	906	8.178	0/77 (0 %)
2.0-3.0	111	906	10.398	0/77 (0 %)
3.0-4.0	109	894	9.263	0/76 (0 %)
4.0-5.0	109	894	9.764	0/76 (0 %)
5.0-6.0	109	894	12.770	0/76 (0 %)
6.0-7.0	115	941	12.097	0/80 (0 %)
7.0-8.0	105	858	10.659	0/73 (0 %)
8.0-9.0	115	941	10.706	0/80 (0 %)
9.0-10.0	77.5	635	6.299	0/54 (0 %)
10.0-11.0	67.5	553	8.012	0/47 (0 %)
0.0-11.1	1130	854	60.363	0/803 (0 %)

Analizando los resultados proporcionados de la prueba que podemos ver en la tabla 5.1 obtenemos las siguientes conclusiones:

- El **ancho de banda** promedio es de 854 kbits/sec, lo cual indica una capacidad de transferencia de datos moderada al tener en cuenta que se está implementando sobre un entorno Mininet.

- El ***jitter***, que mide la variabilidad del retardo de transmisión, oscila entre los 6.3 ms y 12.8 ms. Un *jitter* alto impacta negativamente sobre la calidad de la transmisión, siendo más sensibles las aplicaciones que se encargan de transmitir vídeo. Al realizarse sobre nuestra simulación no podemos decir que el resultado de la transmisión sea deficiente; sin embargo, si se implementase sobre un entorno real, sería conveniente implementar medidas como la gestión de la congestión de la red, una asignación adecuada de recursos, o usar técnicas de enrutamiento más eficientes.

- En cuanto a la **pérdida de datagramas**, no se registra ninguna durante el intervalo medido, lo que sugiere que en dicho momento la red estaba funcionando correctamente sin descartar ningún paquete.

Por último, podemos ver en la figura 5.4 la salida por terminal de los comandos, de donde hemos obtenido los resultados y posteriormente representado en la tabla.

```
"Node: h3"@ubuntuserver
root@ubuntuserver:~/mininet-wifi/pruebas# iperf -s -u -B 224.0.0.1 -p 5031 -i 1
------------------------------------------------------------
Server listening on UDP port 5031
Binding to local address 224.0.0.1
Joining multicast group  224.0.0.1
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 224.0.0.1 port 5031 connected with 10.0.0.6 port 36752
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  5]  0.0- 1.0 sec   122 KBytes  1000 Kbits/sec  14.023 ms    0/   85 (0%)
[  5]  1.0- 2.0 sec   111 KBytes   906 Kbits/sec   8.178 ms    0/   77 (0%)
[  5]  2.0- 3.0 sec   111 KBytes   906 Kbits/sec  10.398 ms    0/   77 (0%)
[  5]  3.0- 4.0 sec   109 KBytes   894 Kbits/sec   9.263 ms    0/   76 (0%)
[  5]  4.0- 5.0 sec   109 KBytes   894 Kbits/sec   9.764 ms    0/   76 (0%)
[  5]  5.0- 6.0 sec   109 KBytes   894 Kbits/sec  12.770 ms    0/   76 (0%)
[  5]  6.0- 7.0 sec   115 KBytes   941 Kbits/sec  12.097 ms    0/   80 (0%)
[  5]  7.0- 8.0 sec   105 KBytes   858 Kbits/sec  10.659 ms    0/   73 (0%)
[  5]  8.0- 9.0 sec   115 KBytes   941 Kbits/sec  10.706 ms    0/   80 (0%)
[  5]  9.0-10.0 sec  77.5 KBytes   635 Kbits/sec   6.299 ms    0/   54 (0%)
[  5] 10.0-11.0 sec  67.5 KBytes   553 Kbits/sec   8.012 ms    0/   47 (0%)
[  5]  0.0-11.1 sec  1.13 MBytes   854 Kbits/sec  60.363 ms    0/  803 (0%)
```

(a) Nodo cliente recibiendo datagramas UDP

```
"Node: sta1"@ubuntuserver
root@ubuntuserver:~/mininet-wifi/pruebas# iperf -c 224.0.0.1 -u -B 224.0.0.1 -p 5031 -i 5 -t 10
------------------------------------------------------------
Client connecting to 224.0.0.1, UDP port 5031
Binding to local address 224.0.0.1
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
Setting multicast TTL to 1
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 224.0.0.1 port 36752 connected with 224.0.0.1 port 5031
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec   642 KBytes  1.05 Mbits/sec
[  5]  0.0-10.4 sec  1.13 MBytes   911 Kbits/sec
[  5] Sent 803 datagrams
```

(b) Nodo cámara enviando datagramas UDP a grupo *multicast*

Figura 5.4: Pruebas cliente-servidor de tráfico *multicast*

5.2.2. Actualización de toda la red

Este caso de prueba consiste en medir el tiempo que los nodos cliente van a tardar en recibir el tráfico de la cámara, contando el tiempo desde que enviamos las instrucciones desde el nivel de aplicación.

```
40054 2023-06-20 09:49:31.075873843  192.168.56.1    192.168.56.101  HTTP  GET /send_rules HTTP/1.1
40075 2023-06-20 09:49:31.113749467  192.168.56.101  192.168.56.1    HTTP  HTTP/1.1 200 OK  (application/json)

Frame 40054: 441 bytes on wire (3528 bits), 441 bytes captured (3528 bits) on interface enp0s3, id 0
  Interface id: 0 (enp0s3)
  Encapsulation type: Ethernet (1)
  Arrival Time: Jun 20, 2023 09:49:31.075873843 CEST
```

Figura 5.5: Wireshark: envío y recibo de reglas

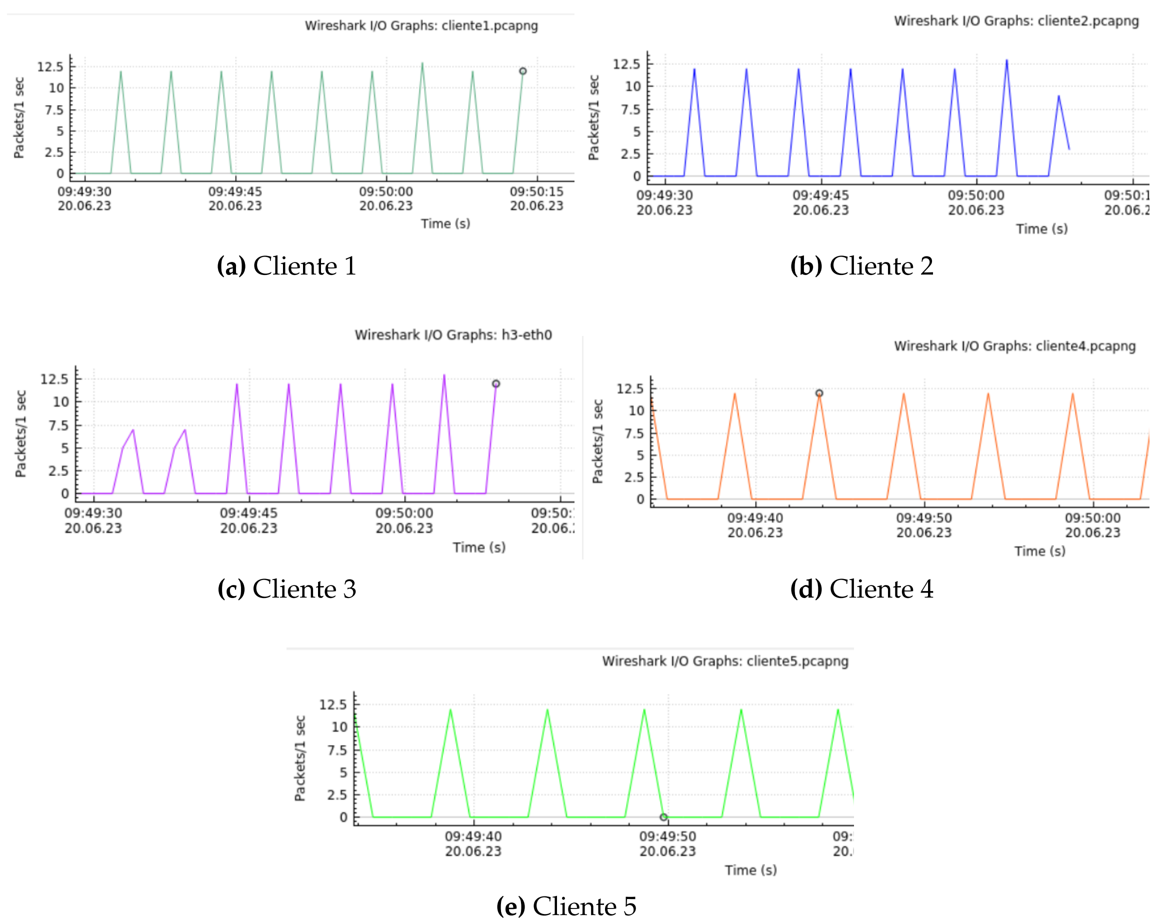
Capa	Instante de tiempo
Aplicación	09:49:31.075873843 (envío)
Control	09:49:31.113749467 (200 OK)

Tabla 5.2: Instantes de tiempo de envío y actualización

Para ello, como podemos ver en la imagen 5.5 la aplicación hace una petición *GET /send_rules* y el controlador le avisa de que ha actualizado la red, tardando la respuesta una cantidad de tiempo de **37.875624 milisegundos**.

El resultado obtenido representa el intervalo necesario para implementar los cambios y asegurar que la red refleje las nuevas configuraciones. Es importante tener en cuenta que esto variará en función de diversos factores como el tamaño o la complejidad de la topología de red. No obstante, el tiempo obtenido para nuestro caso es una eficiencia muy positiva en cuanto a la actualización de la red.

En la última figura 5.6 se ilustra un resumen en modo gráfico el instante aproximado en el que los clientes empiezan a recibir tráfico, coincidiendo obviamente con el instante de tiempo en el que se refina la red.



(a) Cliente 1 **(b)** Cliente 2

(c) Cliente 3 **(d)** Cliente 4

(e) Cliente 5

Figura 5.6: Gráfica de tráfico/tiempo de los clientes

5.2.3. Tiempo necesario para borrar un cliente

Esta evaluación consistirá en medir el tiempo que tardan los dispositivos de red en actualizar las reglas que se reciben desde el nivel de aplicación, en este caso denegar que un cliente reciba tráfico. Para esta prueba se usará como actor el cliente número cinco, puesto que en la solución inicial es el que más cámaras tiene permitidas.

El paso inicial es seleccionar desde la aplicación el cliente que deseamos eliminar, después de ello el servidor nos avisará del éxito de la petición como vemos en la figura 5.7 :

```
b'[{"switch_id": "1000000000000001", "command_result": [{"result": "success", "details": "Rule deleted. : ruleID=3"}]}]'
Borrado de la cámara 1 realizado a las : 19/Jun/2023 11:34:54
b'[{"switch_id": "1000000000000001", "command_result": [{"result": "success", "details": "Rule deleted. : ruleID=5"}]}]'
Borrado de la cámara 2 realizado a las : 19/Jun/2023 11:34:54
b'[{"switch_id": "1000000000000001", "command_result": [{"result": "success", "details": "Rule deleted. : ruleID=9"}]}]'
Borrado de la cámara 4 realizado a las : 19/Jun/2023 11:34:54
b'[{"switch_id": "1000000000000001", "command_result": [{"result": "success", "details": "Rule deleted. : ruleID=11"}]}]'
Borrado de la cámara 6 realizado a las : 19/Jun/2023 11:34:54
Todas las reglas de borrado se han finalizado a las: 19/Jun/2023 11:34:54
192.168.56.1 - - [19/Jun/2023 11:34:54] "GET /send_new_rules HTTP/1.1" 200 -
```

Figura 5.7: Info de la app sobre el borrado de reglas.

Sin embargo, en este caso de prueba necesitamos más precisión del instante de tiempo en el que se realizan las acciones correspondientes. Por ello, se analiza también con Wireshark en qué momento ocurre esto y el resultado podemos verlo en la siguiente imagen 5.8a.

(a) Captura Wireshark borrado de reglas

Wireshark I/O Graphs: h5-eth0

Packets/1 sec: 0, 50, 100, 150

11:34:10 19.06.23 · 11:34:20 19.06.23 · 11:34:30 19.06.23 · 11:34:40 19.06.23 · 11:34:50 19.06.23

Time (s)

(b) Gráfica de tráfico del cliente 5

Figura 5.8: Borrado de cliente

Wireshark nos muestra en dicha captura que el envío de las reglas de borrado (*GET /send_new_rules*) se ha realizado exactamente a las *11:34:54.110524491* y que el controlador nos avisa de que las reglas se han aplicado correctamente en el instante de tiempo *11:34:54.177576699*. En la figura 5.8b se ilustra gráficamente también el momento en el que el cliente deja de recibir tráfico. El proceso de borrado ocurre en un lapso de tiempo extremadamente breve, dándonos como resultado una diferencia de **67.052208 ms**.

Estos resultados obtenidos son altamente satisfactorios, puesto que el procedimiento se lleva a cabo en menos de 1 décima de segundo, y se ha realizado sobre el cliente que más tráfico estaba recibiendo. Esto muestra la agilidad de la arquitectura SDN en esta solución para adaptarse rápidamente a los cambios y

demandas del entorno de red, ofreciendo eficacia y una baja latencia en cuanto a este tipo de peticiones.

5.2.4. Porcentaje de uso de CPU en un vSwitch

Como última prueba, mostraremos si el envío de flujos (reglas) hacia los dispositivos de red aumenta la carga de la CPU sobre estos significativamente y cuál es el estado mientras redirige tráfico. Para ello, por un lado, la evaluación la realizaremos en el switch principal, por donde circulan todas las reglas y el tráfico correspondiente que se genera. Por el otro lado, usaremos la herramienta de Linux vía terminal *htop*, la cual permite monitorear y administrar los procesos en un sistema Linux, al mismo tiempo que muestra de forma gráfica el rendimiento y recursos utilizados.

```
1  [                            0.0%]   Tasks: 163, 290 thr; 1 running
2  [                            0.0%]   Load average: 0.07 0.06 0.07
Mem[||||||||||          1.10G/7.76G]   Uptime: 13:59:11
Swp[                          0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
29381 root       20   0 62676 54652 12004 S  0.7  0.7  0:00.50 /usr/bin/python
  810 root       10 -10  229M 39332 11632 S  0.7  0.5  0:34.27 ovs-vswitchd un
```

(a) CPU y procesos del switch antes de recibir flujos.

```
1  [|                           0.7%]   Tasks: 162, 286 thr; 1 running
2  [||                          1.3%]   Load average: 0.08 0.07 0.07
Mem[||||||||||          1.09G/7.76G]   Uptime: 14:00:01
Swp[                          0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
29412 root       20   0  8116  4084  3240 R  1.3  0.1  0:00.10 htop
29365 root       10 -10  229M 39332 11632 S  0.7  0.5  0:00.02 ovs-vswitchd un
 2005 root       20   0  280M   736   624 S  0.7  0.0  0:03.03 /usr/bin/VBoxDR
```

(b) Instante en el que recibe flujos.

```
1  [                            0.0%]   Tasks: 162, 286 thr; 1 running
2  [|                           0.7%]   Load average: 0.03 0.05 0.07
Mem[||||||||||          1.10G/7.76G]   Uptime: 14:04:33
Swp[                          0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
29438 root       20   0  8104  4112  3284 R  0.7  0.1  0:00.18 htop
29194 root       20   0 86544  1212  1080 S  0.7  0.0  0:00.44 wmediumd -l 0 -
  810 root       10 -10  229M 39332 11632 S  0.0  0.5  0:34.55 ovs-vswitchd un
```

(c) Después de recibir los flujos.

Figura 5.9: Procesos CPU del vSwitch.

En primer lugar, en la figura 5.9 tenemos el resultado de ejecutar *htop* sobre el switch correspondiente en tres fases: antes de enviarle un conjunto de reglas (ver fig. 5.9a), momento en el que recibe las reglas (ver fig. 5.9b), y después de haberlas recibido, (ver fig. 5.9c), lo que quiere decir que también pasará tráfico a través de él. Las prestaciones obtenidas se resumen en la tabla 5.3.

Antes de recibir los flujos		
Uso de CPU	Load Average	Tasks
0.0 %	*0.07*	*163*
Instante en el que los recibe		
Uso de CPU	Load Average	Tasks
2 %	*0.08*	*162*
Después de recibir los flujos		
Uso de CPU	Load Average	Tasks
0.7 %	*0.03*	*162*

Tabla 5.3: Comparación del uso de CPU y carga en un switch tras recibir reglas

Basándonos en los resultados obtenidos, podemos concluir que al recibir las reglas en el switch, la parte del sistema más afectada es el uso de la CPU, ya que experimenta un aumento del 2 %. Sin embargo, este incremento es insignificante en términos generales. Es importante destacar que este aumento en la carga CPU solo se observa en el instante de recibir las reglas y no se mantiene constante.

Además, podemos notar que el parámetro de *load average*, que representa la carga promedia del sistema, vuelve a disminuir después de recibir los flujos. Esto indica que la carga del sistema se estabiliza y vuelve a niveles normales. Por otro lado, el parámetro *tasks*, que representa la cantidad de tareas en ejecución, no se ve afectado en ningún sentido por la recepción de los flujos.

En segundo lugar, observemos en la tabla 5.4 cómo afecta a la CPU el hecho de que comience a circular el tráfico por la red, puesto que previamente se lo hemos permitido mediante las reglas.

Después de recibir los flujos			
Uso de CPU (avg)	Load Average	Tasks	Tráfico en la red
6.1 %	*0.08*	*172*	**180 paquetes/segundo**
8.3 %	*0.12*	*172*	**360 paquetes/segundo**
12.7 %	*0.18*	*173*	**720 paquetes/segundo**
16.1 %	*0.28*	*173*	**1000 paquetes/segundo**

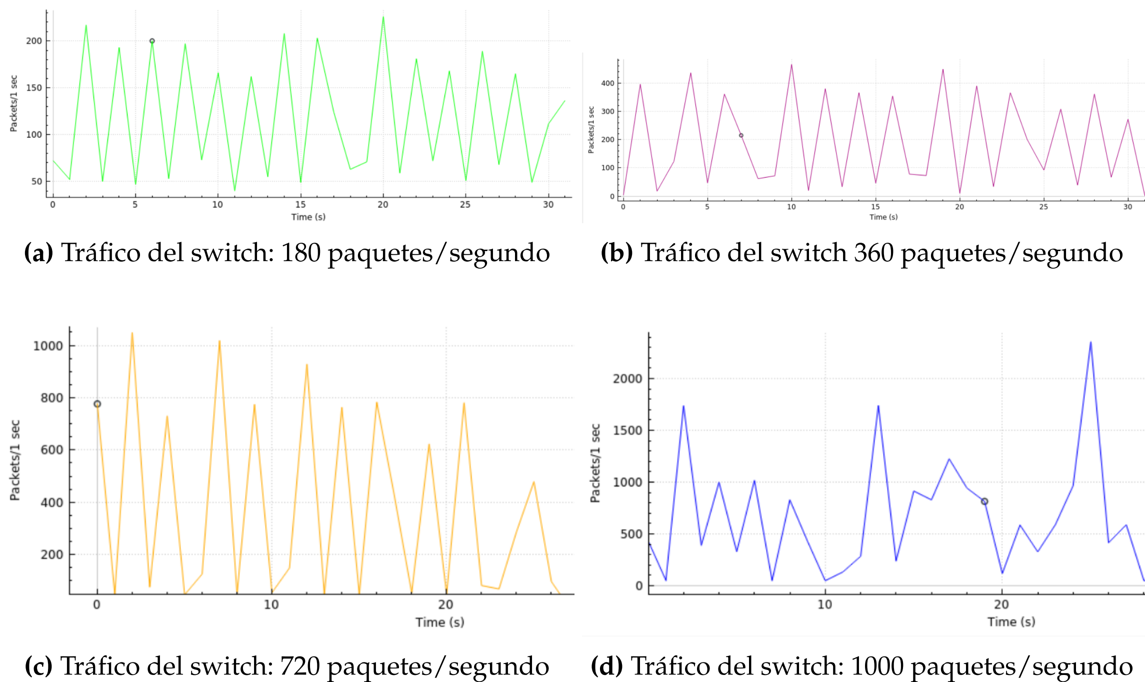
Tabla 5.4: Comparación del uso de CPU y carga en un switch cuando circula tráfico por la red

En general, los resultados muestran una eficiencia más que aceptable en el sistema evaluado. Por un lado, el uso de CPU promedio se mantiene relativamente bajo a medida que aumenta la carga, lo que indica un uso eficiente de los recursos. Además, la carga promedio también se mantiene en valores normales, lo que sugiere que el sistema no está experimentando ninguna carga excesiva.

En cuanto a la cantidad de tareas en el switch, se mantiene en un rango similar en todos los casos, lo cual es un indicador positivo.

En resumen, los resultados muestran que a medida que crece la cantidad de tráfico en la red, los parámetros evaluados tienden también a aumentar. Sin embargo, no lo hacen de manera significativa. Podemos concluir que se muestra

una eficiencia más que aceptable dado el contexto y los aspectos del sistema, demostrando que se puede manejar un mayor flujo de datos sin experimentar un deterioro en el rendimiento.



(a) Tráfico del switch: 180 paquetes/segundo

(b) Tráfico del switch 360 paquetes/segundo

(c) Tráfico del switch: 720 paquetes/segundo

(d) Tráfico del switch: 1000 paquetes/segundo

Figura 5.10: Gráfica de paquetes/segundo en el switch

Como muestra del tráfico generado desde las cámaras para la evaluación de la carga de la CPU tenemos la figura 5.10, donde se ilustra la cantidad de datos que fluyen por la red, siendo también observable algunos picos máximos que se han alcanzado a la par que caídas.

CAPÍTULO 6
Conclusiones

El objetivo principal de este proyecto de fin de grado era diseñar y desarrollar una arquitectura basada en redes definidas por software, reflejando al mismo tiempo cómo son capaces de ofrecer una agilidad y flexibilidad innovadora para gestionar servicios, en este caso dispositivos IoT, mediante la simulación de un escenario de red. Este propósito en su conjunto se ha logrado satisfactoriamente, aunque cabe destacar qué conflictos en términos generales nos hemos ido encontrando a lo largo del camino, y cuáles han sido nuestras impresiones ante todo esto.

En primer lugar, no fue nada simple desde el principio comprender el paradigma de las redes definidas por software y todo lo que abarca, puesto que es una tecnología novedosa y comprende el conocimiento de tres distinguidas capas. Para ello, lo principal fue abastecerse de toda la información posible, e investigar toda esta área, ya que sin ello no habría sido posible desarrollar la idea con total comprensión.

En segundo lugar, posteriores desafíos fueron enfrentarse a herramientas y/o lenguajes de programación en los cuales no se tenía experiencia y no se habían visto a lo largo de la carrera. Comprender y aprender a utilizar *Mininet WiFi* fue una tarea prolongada para poder llegar a programar y automatizar la topología de red que se buscaba.

Otro de los conflictos encontrados fue encontrar un método para que la capa de datos generase tráfico de red automatizado, y llegase correctamente a sus destinos correspondientes. Esto se solucionó mediante la implementación de scripts que generan tráfico *multicast* y scripts *cliente* que lo reciben.

A lo largo del desarrollo, también se tropezó con la capa de control, en la cual había que llegar a entender en su totalidad el funcionamiento de un controlador de red, integrarlo con la topología, y aprender a gestionarlo para implementar el comportamiento que se deseaba. Es el cerebro de toda la arquitectura, y mi mejor consejo o solución para esto es dedicarle todo el tiempo necesario, y analizar su funcionamiento en diferentes escenarios, puesto que va a ser el encargado de comunicar toda la arquitectura entre sí.

La capa de aplicación, formada por la aplicación web desarrollada, supuso del mismo modo un mar de decisiones e implementaciones: diseño, alojamiento, gestión de datos, almacenamiento, uso de varias API, etc. Uno de los mayores obstáculos para su desarrollo fue cómo mantener la persistencia de los datos para

que se encuentre en sincronía con la topología de red y sus cambios; de esta forma le mostrará al administrador el estado actualizado. La solución a ello fue usar el sistema de gestión de bases de datos MySQL, e integrarla con la aplicación web.

El lenguaje de programación que engloba casi la totalidad de la arquitectura es Python, del cual apenas se tenía conocimiento antes del proyecto, además de que no se ve durante la carrera. Es un lenguaje clave a nivel general para ámbitos como la automatización, el mundo de las IAs, el manejo de los datos y, en nuestro caso, lo ha sido a nivel de redes y software enfocado al desarrollo web.

Para finalizar, es necesario subrayar lo enriquecedor que ha sido tanto a nivel personal como profesional el desarrollo de este proyecto. Desde el principio fue motivador el afrontar la idea del TFG, debido al interés personal por el mundo de las redes y la seguridad. Además, sin duda era una oportunidad excelente para investigar y conocer un nuevo paradigma de nuestro sector. Acabo este trabajo con un conocimiento muy detallado sobre las redes definidas por software, y con un dominio destacable en todas las herramientas utilizadas que no conocía previamente.

A nivel profesional, no cabe la menor duda de que es un respaldo increíble el haber estudiado y desarrollado una arquitectura tan minuciosamente, más aún el haber abarcado dos campos que parece que no tienen nada que ver: las redes y el software.

6.1 Relación del trabajo con los estudios cursados

La solución que ofrece este proyecto está ampliamente ligada con los conocimientos que se han adquirido durante la carrera. La carrera de Ingeniería Informática te brinda durante los primeros años una base muy sólida en cuanto a fundamentos y técnicas de programación, es por ello y por haber programado en otros lenguajes que no ha sido una tarea demasiado complicada familiarizarse con Python.

Al mismo tiempo, debido a mi especialización en la rama de Tecnologías de la Información, donde las redes de computadores son el eje central, esta vertiente te ofrece un conjunto de conocimientos, recursos y herramientas para resolver conflictos y necesidades del ámbito de las comunicaciones, lo cual ha sido fundamental para permitirme entender y abordar el paradigma de las redes definidas por software y posteriormente desarrollar su arquitectura.

En resumidas cuentas, he reforzado todos los conocimientos comentados; sin embargo, enfrentarse a tecnologías y herramientas que no se habían estudiado en la carrera ha sido un desafío muy gratificante. Sin duda, ha sido un estímulo para mostrar la capacidad de adaptarse a lo rápido que avanza nuestro sector y al mundo laboral.

6.2 Trabajos futuros

Algunos de los aspectos mejorables del proyecto, que no se pudieran llevar a cabo debido a limitaciones de tiempo, son los siguientes:

- **Diseño de la aplicación web.** La aplicación cumple con su funcionalidad, sin embargo, es sencilla a nivel visual, y habría sido más satisfactorio poder dedicarle muchas más horas.
- **«*Exprimir*» las funcionalidades del controlador de red.** Como se ha comentado previamente, el controlador es un elemento clave de la arquitectura, y se le dedicó un tiempo muy prolongado a averiguar cómo funcionaba, y cómo integrarlo con nuestra solución. Aun así, tiene una cantidad elevada de funcionalidades en las que no se ha ahondado, y habrían sido interesantes trasladarlas a nuestro proyecto.

En cuanto a futuras mejoras, se despliega un abanico enorme de ampliaciones tanto de eficiencia como de funcionalidades de la solución:

- **Investigación de otros controladores de red.** La solución del proyecto usa un controlador en específico, pero como se comenta en su correspondiente sección, existe una amplia variedad de ellos, y quizás la elección de otro podría encajar mejor u ofrecer más funcionalidades.
- **Ampliar las funcionalidades que ofrece la aplicación web.** La arquitectura SDN ofrece una gestión y flexibilidad sorprendente a la hora de administrar un escenario de red. Por ello, en la aplicación podrían mostrarse más opciones como: ver logs de los dispositivos, generar gráficas de tráfico, ver un mapa de la topología de red en tiempo real, etc.
- **Aumentar la complejidad de la topología de red.** Añadir más dispositivos de red es una mejora interesante para mostrar quizás más «realismo» y comprobar cómo se comportaría el simulador.
- **Mejorar la eficiencia del tráfico generado.** Los resultados de las pruebas de carga y eficiencia del TFG son bastante aceptables, sin embargo, no cabe duda que examinando e implementando técnicas de red avanzadas (enrutamiento, gestión de la congestión de red, asignación de recursos, etc.), esto potenciaría las prestaciones de la solución.

Bibliografía

[1] Siquan Hu, Pengyuan Zhou, Junfeng Wang. The Inter-Datacenter Connection in SDN and Traditional Hybrid Network. *School of Computer and Communications Engineering, University of Science and Technology Beijing, Beijing, China*, September 2020.

[2] Živko Bojović, Petar D. Bojović, Jelena Šuh. The implementation of Software Defined Networking in enterprise networks. *The Journal (Institute of Telecommunications Professionals)*, March 2018.

[3] State of IoT 2022: Number of connected IoT devices growing 18 to 14.4 billion globally. Consultado el 29 de mayo de 2023. Disponible en: https://iot-analytics.com/number-connected-iot-devices/.

[4] Ciudades inteligentes: las ciudades del futuro. Consultado el 29 de mayo de 2023. Disponible en: https://www.microsoft.com/es-es/industry/government/resources/smart-cities.

[5] Somayya Madakam. Internet of Things: Smart Things. *International Journal of Future Computer and Communications*, Vol. 4, No. 4, august, 2015.

[6] Global Internet adoption and devices and connection. Consultado el 30 de mayo de 2023. Disponible en: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[7] Enrique Villa Crespo, Ismael Morales Alonso. *Ciberseguridad IoT y su aplicación en Ciudades Inteligentes*. RA-MA Editorial, Madrid, marzo, 2023.

[8] Jordi Salazar, Santiago Silvestre. *Internet of things*. Czech Technical University of Prague Faculty of electrical engineering, 1st Edition, 2017.

[9] M.U Farooq, Muhammad Waseem, Sadia Mazhar, Anjum Khairi, Talha Kamal. A Review on Internet of Things (IoT). *International Journal of Computer Applications*, Volume 113 - No. 1, March 2015.

[10] Red de sensores. Consultado el 4 de junio de 2023. Disponible en https://es.wikipedia.org/wiki/Red_de_sensores.

[11] Smart cities: How technology can deliver a better quality of life. Consultado el 4 de junio de 2023. Disponible en https://www.mckinsey.com/capabilities/operations/our-insights/smart-cities-digital-solutions-for-a-more-livable-future.

[12] Smart Cities: Videovigilancia inteligente. Consultado el 5 de junio de 2023. Disponible en https://secmotic.com/smart-cities-videovigilancia-inteligente/#gref.

[13] Thomas D. Nadeau & Ken Gray. *An Authoritative Review of Network Programmability Techonologies - Software Defined Networks*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st Edition, August 2013.

[14] N.Sabitha, H.Jayasree, A.V. Krishna Prasad. Virtualization of Traditional Networks using SDN. *International Journal of Interdisciplinary Research and Innovations*, september, 2019.

[15] Global Software-Defined Networking (SDN) Market Outlook. Consultado el 19 de junio de 2023. Disponible en https://www.expertmarketresearch.com/reports/software-defined-networking-market

[16] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China (2013).

[17] Open Networking Foundation. *OpenFlow Switch Specification*. Version 1.3.1 (Wire Protocol 0x04), 2008.

[18] Ryu development team. *Ryu Documentation*. Release 4.34, Jun 09, 2022.

[19] Ángel Leonardo Valdivieso Caraguay, Alberto Benito Peral, Lorena Isabel Barona López, and Luis Javier García Villalba. SDN: Evolution and Opportunities in the Development IoT Applications. *International Journal of Distributed Sensor Networks*, Volume 2014, 4 May, 2014.

[20] International Data Corporation (IDC). Consultado el 6 de junio de 2023. Disponible en https://en.wikipedia.org/wiki/International_Data_Group#International_Data_Corporation_(IDC).

[21] El crecimiento de los datos de IoT requiere una estrategia de almacenamiento a largo plazo. Consultado el 6 de junio de 2023. Disponible en https://almacenamientoit.ituser.es/go-to/39144.

[22] Multidifusión. Consultado el 11 de junio de 2023. Disponible en https://es.wikipedia.org/wiki/Multidifusi%C3%B3n.

[23] REST Firewall. Consultado el 14 de junio de 2023. Disponible en https://osrg.github.io/ryu-book/en/html/rest_firewall.html.

[24] Rest_Firewall.py. Consultado el 18 de junio 2023. Disponible en https://github.com/faucetsdn/ryu/blob/d6cda4f427ff8de82b94c58aa826824a106014c2/ryu/app/rest_firewall.pyl.

APÉNDICE A
Objetivos de desarrollo sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**		X		
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**	X			
ODS 7. **Energía asequible y no contaminante.**	X			
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**	X			
ODS 12. **Producción y consumo responsables.**	X			
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**		X		
ODS 15. **Vida de ecosistemas terrestres.**		X		
ODS 16. **Paz, justicia e instituciones sólidas.**		X		
ODS 17. **Alianzas para lograr objetivos.**		X		

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Las cámaras de videovigilancia implementadas en la solución están situadas en el contexto de las Smart Cities, es por ello que el TFG está relacionado con varios objetivos de Desarrollo Sostenible como veremos a continuación.

*Está ligado en un grado de nivel alto con los objetivos de «**Agua limpia y saneamiento**» y «**Energía asequible y no contaminante**» puesto que uno de los objetivos principales de las Smart Cities y los sensores IoT que se implementan en ellas es la monitorización ambiental para controlar y mejorar la calidad del agua, aire, atmósfera, etc. Por ello, cámaras y/o sensores como las de nuestra solución ayudan en gran medida y cada vez más a regular esta situación.*

*En cuanto al objetivo de «**Industria, innovación e infraestructuras**», estas tecnologías contribuyen a mejorar la gestión de las infraestructuras urbanas al proporcionar datos en tiempo real sobre el estado de diferentes aspectos, como el tráfico o la iluminación pública. Esto permite una planificación más eficiente de los recursos.*

*Las cámaras de la red IoT y las redes SDN también contribuyen a promover el objetivo de «**Producción y consumo responsables**», debido a que el análisis de datos obtenidos a través de estas cámaras y redes puede ayudar a identificar patrones de consumo y a fomentar prácticas más sostenibles.*

*Con el objetivo con el cual está mayormente relacionado es «**Ciudades y comunidades sostenibles**», teniendo en cuenta que es una de las características principales de las Smart Cities, la implementación de cámaras de videovigilancia promueve la creación de comunidades sostenibles al mejorar la seguridad y la calidad de vida en las ciudades. El integrar esta tecnología en las ciudad permite una gestión eficiente de los recursos urbanos, facilitando la planificación del crecimiento urbano, el transporte inteligente y la optimización de los servicios públicos. Al mismo tiempo, las redes SDN ayudan a reducir la compra innecesaria de dispositivos informáticos, ya que con ellas se puede implementar el sistema previamente y hacer las pruebas necesarias para saber cuáles van a ser los elementos fundamentales.*

*Como último objetivo relacionado en mayor grado, tenemos «**Acción por el clima**», donde la implementación de cámaras de videovigilancia en las Smart Cities desempeña un papel fundamental ya que que estas permiten el monitoreo y control ambiental en tiempo real, ayudando a identificar y abordar los impactos negativos en el medio ambiente.*

En resumen, las cámaras IoT en Smart Cities contribuyen al desarrollo sostenible al mejorar la eficiencia de los recursos, promover la seguridad y optimizar la gestión de las ciudades. Estas tecnologías permiten una planificación urbana inteligente, una respuesta más rápida ante situaciones de emergencia y la reducción de impactos ambientales negativos.

APÉNDICE B

Guía para la instalación del entorno de trabajo

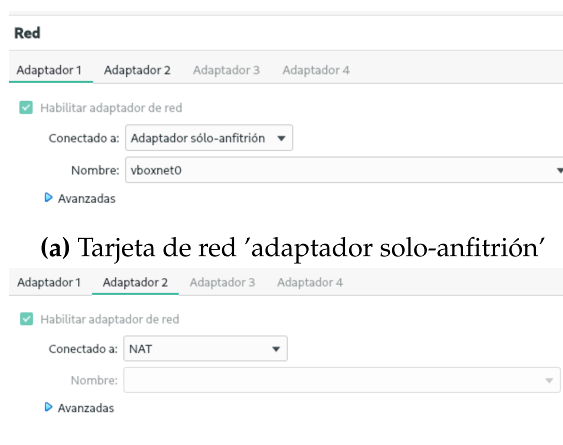
La solución implementada se ha realizado sobre el siguiente entorno, con las correspondientes herramientas y requisitos:

- Máquina virtual con S.O. Ubuntu 20.04 con dos tarjetas de red
- *Mininet WiFi* versión 2.6
- Controlador Ryu versión 4.30
- Versión de Python 3.8.10

Instalación de la máquina virtual

La máquina virtual se ha instalado, en nuestro caso, usando el software de virtualización *VirtualBox*[1]. Referencia usada para la instalación: https://getlabsdone.com/how-to-install-ubuntu-20-04-server-on-virtualbox/

Una vez tengamos el entorno instalado, debemos configurar las dos tarjetas de red como veremos en la imagen adjunta (ver figura B.1).



(a) Tarjeta de red 'adaptador solo-anfitrión'

(b) Tarjeta de red 'NAT'

Figura B.1: Tarjetas de red del sistema

[1]https://www.virtualbox.org/wiki/Downloads

Con esta configuración, las tarjetas de red deberían quedar configuradas de manera semblante a la siguiente B.2:

```
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.101  netmask 255.255.255.0  broadcast 192.168.56.255
        inet6 fe80::a00:27ff:fe6e:2a23  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:6e:2a:23  txqueuelen 1000  (Ethernet)
        RX packets 2902643  bytes 214413161 (214.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4430872  bytes 2905730695 (2.9 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.3.15  netmask 255.255.255.0  broadcast 10.0.3.255
        inet6 fe80::a00:27ff:fecc:a7ff  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:cc:a7:ff  txqueuelen 1000  (Ethernet)
        RX packets 92023  bytes 134221919 (134.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 29320  bytes 2115935 (2.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figura B.2: Configuración de red de la máquina

Instalación de Mininet WiFi

Una de las sencillas opciones para instalar la herramienta es la ejecución por terminal de los siguientes comandos:

```
git clone https://github.com/intrig-unicamp/mininet-wifi
cd mininet-wifi/
sudo util/install.sh -Wlnfv
```

Una vez instalada, el siguiente comando ejecutado debería darnos la salida de la versión:

```
sdn@ubuntuserver:~$ mn --wifi --version
2.6
```

Instalación del controlador de red Ryu

Para poder ejecutar Ryu sin problemas, es necesario que tengamos bien definida la ruta del equipo, ya que Ryu se ejecutará sobre esta para poder llamar a la aplicación desde donde queramos. Ejecutamos el comando « nano /.bashrc » y añadimos la siguiente línea:

```
export PATH="/usr/local/bin:$PATH"
```

Después de guardar el fichero, ejecutaremos los siguientes comandos para hacer efectiva la ruta e instalar Ryu:

```
source ~/.bashrc
pip install ryu
```

En nuestro caso, la versión instalada es la 4.30 y podemos comprobarlo como vemos en la imagen B.3

```
2.6
sdn@ubuntuserver:~$ ryu-manager --version
ryu-manager 4.30
sdn@ubuntuserver:~$
```

Figura B.3: Versión de Ryu

Versión de Python 3.8.10

Seguramente, versiones previas o en el futuro versiones posteriores sean compatibles con el entorno y las herramientas específicas instaladas, pero hasta dicho momento del desarrollo del trabajo y haber probado múltiples combinaciones posibles la que ha funcionado como se deseaba ha sido la 3.8.10. Se puede instalar de la siguiente forma:

```
1 sudo apt install python3.8
```