

Universidad de Alcalá Escuela Politécnica Superior

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Diseño e implementación de protocolo
de control escalable en redes IoT para
entornos 6G

ESCUELA POLITECNICA
SUPERIOR

Autor: David Carrascal Acebron

Tutor: Elisa Rojas Sánchez

2023



Máster Universitario en Ingeniería de Telecomunicación



Universidad
de Alcalá

Madrid, 7 de julio de 2023

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

**Diseño e implementación de protocolo
de control escalable en redes IoT para entornos 6G**

Autor: David Carrascal Acebron

Tutor: Elisa Rojas Sánchez

Tribunal:

Presidente: Juan Manuel Arco Rodríguez

Vocal 1º: M^a Elena López Guillén

Vocal 2º: Isaias Martinez Yelmo

*A mis hermanas, Natalia y Violeta,
quienes día a día, por oscura que sea la noche,
arrojan luz y esperanza a mi vida.*

Agradecimientos

Quiero empezar agradeciendo y reconociendo a mi tutora, Elisa Rojas, sin la cual este trabajo no habría sido posible. Quien desde segundo de carrera creyó en mi, y aun día de hoy, sigue apostando día a día en mis capacidades, incluso cuando ni yo mismo soy capaz de verlas. Su destreza y conocimiento, su apoyo incondicional y carisma, su maestría y pasión por lo que hace, y a lo que se dedica, han hecho que etapa, tras etapa académica siga aprendiendo y disfrutando como el primer día. Este trabajo ha sido financiado por subvenciones de la Comunidad de Madrid a través de los proyectos TAPIR-CM (S2018/TCS-4496) y MistLETOE-CM (CM/JIN/2021-006), y por el proyecto ONENESS (PID2020-116361RA-I00) del Ministerio de Ciencia e Innovación de España.

También me gustaría agradecer a mi familia, por su cariño, comprensión e inspiración en estos meses que han sido tan duros para mí. Y que decir de mis amigos, a los *Caye de Calle*, a los *C de Chill*, a mis estimados *Pueblerinos*, como no, a Pablo y Olga, a mis queridas Noci, a mi señor abuelo de confianza, Boby, a la señorita Laura de Diego, mis compis de la uni y toda la gente nueva que ha llegado a mi vida durante estos meses, a todos vosotros, gracias por las risas y los buenos momentos que hemos compartido juntos. Gracias de verdad.

No puedo terminar sin agradecer a toda la gente del Laboratorio LE34, quienes me alentando a seguir por este arduo camino de la investigación y quienes con sus consejos y experiencias han ido conformando al ingeniero que soy a día de hoy.

Sinceramente, mil gracias a todos.

Resumen

En este Trabajo Final de Máster (TFM) se presenta el diseño e implementación de un protocolo de control escalable de redes Internet of Things (IoT) para entornos Software-Defined Networking (SDN) en la nueva generación de redes móviles, the sixth generation of mobile technologies (6G). Dicho protocolo de control seguirá un paradigma de control de tipo *in-band*, con el cual se dotará de conectividad a los nodos de la red con el ente de control, empleando el plano de datos para la transmisión de información de control.

En aras de completar el proyecto, se ha partido por analizar las necesidades y características de las distintas tecnologías que se emplearán en ejecución del objetivos predefinidos y así discernir aquellas herramientas necesarias para la implementación del protocolo control. Una vez seleccionadas las herramientas, se estudiarán a fondo para realizar una implementación lo optimizada en la medida de lo posible. Este proyecto concluirá con la validación mediante emulación del protocolo desarrollado para comprobar el correcto funcionamiento del mismo en distintos casos de uso.

Palabras clave: [6G](#); [IoT](#); [SDN](#); [Control in-band](#); [Plano de control](#)

Abstract

In this Master's Thesis (TFM) we present the design and implementation of a scalable control protocol for Internet of Things (IoT) networks for Software-Defined Networking (SDN) environments in the new generation of mobile networks, the sixth generation of mobile technologies (6G). This control protocol will be based on an *in-band* control paradigm, which will provide connectivity between the network nodes and the control entity, using the data plane for the transmission of control information.

In order to fulfil the project, we have started by analysing the requirements and characteristics of the different technologies that will be used in the execution of the predefined objectives and thus be able to determine the tools necessary for the implementation of the control protocol. Once the tools have been selected, they will be studied in depth in order to carry out an optimised implementation as far as possible. This project will conclude with the validation the developed protocol by means of emulation to check its correct operation in different use cases.

Keywords: 6G; IoT; SDN; In-band control; Control plane

“No hay ningún viento favorable para el que no sabe a qué puerto se dirige”

Arthur Schopenhauer.

Índice general

Resumen	v
Abstract	vii
1. Introducción	1
1.1. El Internet de las Cosas y la red 6G	1
1.2. Redes SDN	4
1.3. Objetivos	6
1.4. Estructura del TFM	7
1.5. Contribuciones	8
2. Estado del arte	9
2.1. Red de comunicación 6G	9
2.2. Tecnología IoT	9
2.2.1. Arquitectura	10
2.2.2. Topologías	10
2.2.3. Redes LLN	11
2.2.3.1. IEEE 802.15.4	12
2.3. Redes SDN	12
2.3.1. Arquitectura	13
2.3.2. OpenFlow	13
2.4. Controladores SDN	14
2.4.1. Ryu	17
2.4.2. ONOS	19
2.5. Software Switches SDN	21
2.5.1. OvS	22
2.5.2. BOFUSS	24
2.5.2.1. Ports	25
2.5.2.2. Packet Parser	27
2.5.2.3. Flow Tables	28
2.5.2.4. Group Table	30
2.5.2.5. Meter Table	30

2.5.2.6. oflib	31
2.5.2.7. Communication Channel	32
2.6. Linux Networking	33
2.6.1. Interfaz virtual - tun/tap	33
2.6.2. Interfaz virtual - veth	35
2.6.3. Herramienta TC	36
2.6.3.1. Qdiscs	37
2.6.3.2. Classes	37
2.6.3.3. Filters	37
2.6.4. Namespaces	39
2.6.4.1. Persistencia de las Namespaces	39
2.6.4.2. Concepto de las Network Namespaces	40
2.6.4.3. Comunicación inter-Namespaces: Veth	41
2.7. Mininet y Mininet-WiFi	42
2.7.1. Mininet	42
2.7.2. Funcionamiento de Mininet	43
2.7.3. Mininet-WiFi	47
2.7.4. Mininet-IoT	47
2.8. Contiki-ng	48
2.8.1. Simulador Cooja	49
2.9. Contribuciones en GitHub	50
Bibliografía	53
Lista de Acrónimos y Abreviaturas	59
A. Anexo I - Pliego de condiciones	61
A.1. Condiciones materiales y equipos	61
A.1.1. Especificaciones Máquina A	61
A.1.2. Especificaciones Máquina B	61
A.1.3. Especificaciones Máquina C	62
B. Anexo II - Presupuesto	65
B.1. Duración del proyecto	65
B.2. Costes del proyecto	66
C. Anexo III - Manuales de usuario e Instalación	69
C.1. Instalación de dependencias de los casos de uso	69
C.1.1. Instalación de dependencias máquina XDP	69

C.1.2.	Instalación de dependencias máquina P4	70
C.2.	Herramienta <code>iproute2</code>	71
C.2.1.	¿Qué es <code>iproute2</code> ?	71
C.2.2.	¿Por qué necesitamos <code>iproute2</code> ?	72
C.2.3.	Estudio de compatibilidad de la herramienta <code>iproute2</code> en Ubuntu . .	72
C.2.4.	Compilación e instalación de <code>iproute2</code>	74
C.2.4.1.	Diferencias con Ubuntu 18.04	75
C.2.5.	Comandos útiles con <code>iproute2</code>	75
C.3.	Herramienta <code>tcpdump</code>	77
C.3.1.	¿Qué es <code>tcpdump</code> ?	77
C.3.2.	¿Por qué necesitamos <code>tcpdump</code> ?	77
C.3.3.	Instalación de <code>tcpdump</code>	78
C.3.4.	Comandos útiles con <code>tcpdump</code>	78
C.4.	Herramienta Mininet	79

Índice de figuras

1.1.	Estudio de las conexiones IoT máximas simultáneas a nivel global [3]	2
1.2.	<i>Roadmap</i> propuesto por el SNS-JU para el desarrollo del 6G [6]	3
1.3.	Paradigma en las redes SDN [17]	5
1.4.	Paradigma control en las redes SDN [18]	6
2.1.	Arquitectura básica IoT	11
2.2.	Tipos de topología con dispositivos IoT [21]	11
2.3.	Pila de protocolos 6LoWPAN [23]	12
2.4.	Arquitectura básica SDN	14
2.5.	Arquitectura genérica de controlador SDN [27]	15
2.6.	Arquitectura del controlador SDN RYU [30]	18
2.7.	Arquitectura del controlador SDN onos [33]	20
2.8.	Arquitectura genérica de un <i>softswitch</i> SDN [34]	21
2.9.	Arquitectura del OvS [36]	24
2.10.	Evolución del BOFUSS	24
2.11.	Arquitectura del BOFUSS [38]	26
2.12.	Parseador de paquetes del BOFUSS [38]	28
2.13.	Estructura de las <i>Group tables</i> del BOFUSS [38]	30
2.14.	Estructura de las <i>Meter tables</i> del BOFUSS [38]	31
2.15.	Proceso de Marshaling y Unmarshaling	32
2.16.	Diagrama de funcionamiento de las interfaces virtuales TUN/TAP [40]	34
2.17.	Comprobación con <code>ethtool</code> de tipo de interfaz virtual.	35
2.18.	Mecanismo de calidad de servicio implementado con clases y sub-clases [44]	38
2.19.	Enlace entre interfaces Veth separadas en dos Network Namespaces [17]	41
2.20.	Arquitectura de Mininet [48]	44
2.21.	Salida por pantalla de la ejecución de la topología por defecto	45
2.22.	Listado de Named Network Namespaces existentes en el sistema	45
2.23.	Listado de procesos con referencias a Mininet	46
2.24.	Información de contexto sobre el proceso del Host1	46
2.25.	Información de contexto sobre el proceso del Host2	47
2.26.	Gestión de la memoria en un sistema con Contiki OS [51]	49

2.27. Interfaz gráfica del simulador Cooja [53]	50
A.1. Especificaciones de la máquina A	62
A.2. Especificaciones de la máquina B	62
A.3. Especificaciones de la máquina C	63
B.1. Diagrama de Gantt del proyecto	68
C.1. Ramificación de dependencias de Iproute2.	73
C.2. Interfaz CLI de Tcpdump	79

Índice de tablas

2.1. Resumen de los tipos de Namespaces en el Kernel de Linux	39
2.2. Resumen de contribuciones realizadas	51
B.1. Promedio de horas de trabajo	65
B.2. Presupuesto desglosado del Hardware para el TFM	66
B.3. Presupuesto desglosado del Software para el TFM	66
B.4. Presupuesto total con IVA	67
C.1. Comparativa de herramientas Iproute2 con paquete net-tools	72
C.2. Estudio de compatibilidad de la herramienta Iproute2	73
C.3. Especificaciones máquina de instalación Mininet	79

Índice de Códigos

2.1. Manejo de interfaces TUN - TAP	34
2.2. Uso de las interfaces Veths	36
2.3. Casos de uso de las Netns	41
2.4. Ejecución de Mininet con la topología por defecto	44
2.5. Listar Named Network Namespaces	45
C.1. Descarga del repositorio del TFG	69
C.2. Instalación de dependencias XDP	70
C.3. Instalación de dependencias P4	71
C.4. Instalación de las dependencias de Iproute2	74
C.5. Obtención del source de Iproute2	74
C.6. Compilación e instalación de Iproute2	75
C.7. Instalación de las dependencias de Iproute2 - Ubuntu 18.04	75
C.8. Comandos útiles con iproute2	75
C.9. Instalación de Tcpdump	78
C.10. Comandos útiles con Tcpdump	78
C.11. Instalación de la herramienta git	79
C.12. Instalación de la herramienta Mininet	79

1. Introducción

En este primer capítulo, se desea presentar de manera concisa los aspectos más relevantes del TFM, como son, las redes de dispositivos IoT, la llegada de los entornos 6G, y la tecnología habilitadora en dichos entornos, el SDN. Se explorarán las necesidades actuales de las redes de sensores IoT, se indagará la postulada nueva generación de redes móviles, 6G, y se verá donde entrará las redes SDN, y qué mejoras deberán hacerse para hacer frente a las necesidades imperantes de las próximas redes de sensores.

Se establecerán objetivos claros para el TFM y se describirá detalladamente cómo se planea llevarlos a cabo cada uno de ellos. Estos objetivos ayudarán a al diseño y desarrollo de un nuevo protocolo de comunicación de control escalable para redes de sensores en un ámbito de red SDN. De forma adicional, se presentará la estructura general del TFM, describiendo de manera breve los temas que se abordarán en cada capítulo. Por último, se indicarán las contribuciones realizadas en revistas científicas de alto impacto de este proyecto.

1.1. El Internet de las Cosas y la red 6G

Los recientes avances en las comunicaciones móviles junto a la mejora de las capacidades tecnológicas de los elementos hardware han llevado al IoT a un punto álgido, donde, a día de hoy, interconecta billones de objetos entre sí con comunicaciones Machine to machine (M2M) tanto en entornos particulares, como en entornos industriales [1]. Se puede afirmar que sin lugar a dudas el IoT es parte del hoy y el mañana de Internet, ha revolucionado la forma en se interactúa con el mundo que nos rodea, permitiendo conectar dispositivos entre sí de forma completamente autónoma a través de la red, proveyendo al humano de entornos inteligentes y adaptativos a las necesidades de la sociedad.

Sin embargo, el aumento exponencial de los dispositivos IoT conectados a las redes de comunicaciones móviles ha generado nuevas necesidades en términos de capacidad, rendimiento, latencia y eficiencia de las redes que deben ser solventadas. Las tecnologías móviles the fifth generation of mobile technologies (5G) ya se han propuesto y desplegado comercialmente para dar soporte a las necesidades de las redes IoT y sus aplicaciones. Esta tecnología habilitadora daba solución a las necesidades preliminares del IoT haciendo uso de las fun-

cionalidades que traía consigo, como por ejemplo, enhanced Mobile BroadBand (eMBB), massive Machine-Type Communication (mMTC), Ultra-Reliable and Low-Latency Communication (URLLC) [2]. Dichas funcionalidades proveían a los ecosistemas IoT de servicios de alto ancho de banda, baja latencia y optimización del consumo, siendo esta última muy importante para los dispositivos IoT. No obstante, con la rápida proliferación de nuevos de sensores, y con ello, el aumento de las redes IoT según se puede apreciar en la figura 1.1, los requisitos técnicos necesarios se han visto aumentados para poder seguir manteniendo entornos M2M tal cual se conocían, completamente autónomos, dinámicos e inteligentes.



Figura 1.1: Estudio de las conexiones IoT máximas simultáneas a nivel global [3]

Por ello, se necesita una tecnología más avanzada que pueda satisfacer las futuras demandas de las redes IoT, y la tecnología 6G se postula como una solución a todos los nuevos retos planteados. Por esa razón, para facilitar el desarrollo de las futuras redes IoT la investigación en la siguiente generación de redes móviles ha recibido mucha atención tanto por parte de academia e instituciones, como por parte de la industria [4].

Si bien es cierto que la tecnología 6G está todavía siendo formulada, se espera que esta pueda proporcionar una Calidad de servicio (QoS) totalmente mejorada frente a la generación

anterior, dado sus prestaciones son claramente superiores, como por ejemplo, comunicaciones de ultra-baja latencia, tasas de velocidad de datos mejoradas y entornos inteligentes de comunicación con satélites.

Por consiguiente, muchos de los esfuerzos de las instituciones están pasando por impulsar las redes 6G-IoT. Desde Europa se está impulsando la carrera por el 6G poniendo sobre la mesa una hoja de ruta dividida en cuatro fases diferenciadas, esperando poder finalizar en 2030 [5]. Dichos esfuerzos están siendo coordinados desde la EU's Smart Networks and Services Joint Undertaking (SNS-JU) la cual tiene como misiones principales impulsar el despliegue del 5G, y situar a los países miembros de la EU a la vanguardía del desarrollo de la próxima generación de redes móviles [6]. Estas misiones planean llevarlas a cabo a través de un detallado *roadmap* el cual se puede apreciar en la figura 1.2.



Figura 1.2: *Roadmap* propuesto por el SNS-JU para el desarrollo del 6G [6]

Dicho *roadmap* se compone de varias fases, las cuales se han subdividido en streams. Los streams principales son cuatro, actualmente nos encontramos en el stream B del roadmap [7], el cual tiene como objetivo el impulso de la investigación en las áreas tecnológicas habilitadoras del 6G además de la cooperación internacional con estados estratégicos como Estados Unidos. De entre todos los proyectos iniciados en 2023, destacan por ejemplo, Hexa-X [8], iniciativa principal de Europa liderada por Nokia financiado por el programa europeo Horizon 2020 que busca desarrollar el prototipado de los sistemas 6G. Otro ejemplo, es el proyecto 6Genesis [9] financiado por Finlandia, que busca la generación de las primeras pruebas de concepto experimentales de redes 6G-IoT. Si nos vamos a proyectos más específicos, podemos encontrar ADROIT6G [10] y 6GTandem [11], ambos han comenzado en Enero de 2023 y buscan mejorar y optimizar los sistemas distribuidos de acceso al medio mediante sistemas duales en frecuencia empleando procesamiento de señal Multiple-Input and Multiple-Output (MIMO).

Pero el interés en la próxima generación de redes móviles, no es meramente local, si nos vamos a Estados Unidos, podemos ver como ya la FCC libera espectro en la banda de los THz en US para hacer pruebas de concepto para el 6G [12]. O a Corea del Sur, donde ya han planeado lanzar un proyecto piloto de 6G para el 2026 [13].

Como se puede ver, el interés por el 6G es real, y se están realizando esfuerzos exhaustivos para la proliferación de las tecnologías habilitadoras del 6G, por lo que distintas instituciones apuntan a que las primeras redes 6G que podrán ser desplegadas en 2028, pero que su comercialización y la llegada a las personas de a pie no llegará hasta el 2030 [4]. Uno de los aspectos relevantes en la nueva generación de redes móviles, es la arquitectura de interconexión física que se va a plantear. En el 5G, ya se reaprovecho el *backbone* existente de SDN, al cual haciendo uso de flexibilidad y de la programabilidad, se le indujeron modificaciones software para atender las nuevas especificaciones de la arquitectura planteada [2]. Teniendo esto en cuenta, al lector le pueden surgir dudas de si la próxima red de 6G hará uso de las bondades de las redes SDN para impulsar el procesamiento de datos en su arquitectura como parte del *backbone*. Según los informes preliminares [14] [15] [16] que se han presentado en cuanto al diseño de la red 6G, se indica que harán uso del SDN, junto a la tecnología Programming Protocol-independent Packet Processors (P4) y técnicas Artificial Intelligence (AI)/Machine Learning (ML), para la definición de plano de procesamiento de datos y mejorar el rendimiento y orquestación del *backbone* ya existente.

1.2. Redes SDN

Como se ha podido ver, las redes SDN serán una realidad en las próximas redes de comunicaciones móviles 6G. La figura 1.3 muestra cómo con estas redes, se pretende separar el plano de control de los dispositivos intermedios de procesamiento de la red y centralizarlo en entidades denominadas controladores, lo que permitirá una administración más flexible y centralizada de la red.

Antes de que apareciera el concepto de SDN, las redes convencionales solían tener un plano de control unificado en los propios dispositivos, llamado generalmente *Control plane*, en el que se definía la lógica que dictaba cómo se debía llevar a cabo el forwarding de los paquetes, y un plano de datos, conocido como *Data plane*, que se implementaba definiendo su datapath, compuesto por varios bloques de procesamiento para reenviar los paquetes. Sin embargo, con la aparición del paradigma de las redes SDN, como se muestra en la Figura 1.3, los nodos tradicionales de la red verían cómo su plano de control sería delegado a una enti-

dad externa llamada controlador. Este controlador tendría una perspectiva global de toda la red en su conjunto, lo que permitiría una gestión más flexible, dinámica y optimizada.

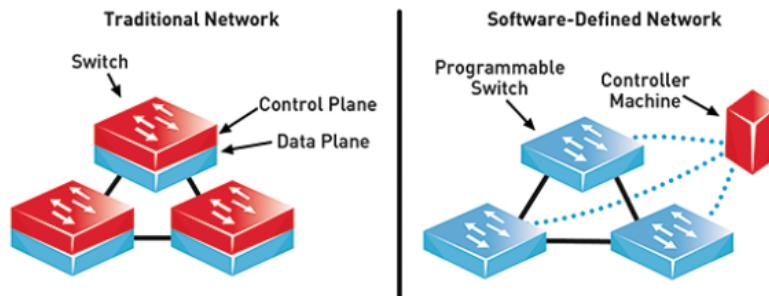


Figura 1.3: Paradigma en las redes SDN [17]

A la hora de la transmisión de la información de control desde el controlador hacia los nodos SDN, se plantean dos puntos importantes. El primero de ellos es qué esquema en la red control se va a plantear, y la segunda que protocolo de comunicaciones se va a emplear. Empezando por los protocolos de comunicación de información de control en redes SDN, el más utilizado es OpenFlow, aunque más adelante se verán en detalle alguno que otro más.

En cuanto a los paradigmas de control SDN, tenemos dos vertientes [18], ***out-of-band*** e ***in-band***. La diferencia entre cada paradigma, ver Figura 1.4, es que en el modelo *out-of-band*, cada nodo SDN tiene un enlace dedicado con el controlador, es decir la información de control tiene una red dedicada por y para ella. El modelo *in-band* por el contrario, se tiene que solo alguno/s de los nodos SDN gestionados tiene un enlace con el controlador, y el resto de equipos emplean ese enlace para hacerle llegar al controlador la información de control. Se quiere señalar que en este último modo la información de control al no tener una red dedicada para ella misma, viajará de forma conjunta por el plano de datos hasta llegar al controlador.

No hay un paradigma mejor que otro, cada paradigma de control tiene unos pros y unos contras, y será el caso de uso quien predisponga cuál de los dos usar. Por ejemplo, el modelo *out-of-band* es un modelo mucho más caro dado que se tiene un enlace dedicado para comunicación controlador - nodo SDN, pero por ello, también es más seguro dado que el tráfico de control está aislado. Por el contrario, el modelo *in-band*, es mucho más barato dado que se emplea de red de datos para la transmisión de la información de control. Sin embargo, es un modelo más inseguro, ya que la información de control está expuesta al plano de datos.



Figura 1.4: Paradigma control en las redes SDN [18]

Uno de los puntos diferenciadores entre ambos paradigmas de control, es la configuración requerida en cada caso. En el modelo de *out-of-band*, no hay apenas configuración requerida, dado que, si la información es de control, tendremos una interfaz de red exclusiva con la cual trabajar. Por el contrario, en el modelo *in-band*, si tenemos que tratar información de control no sabemos con qué interfaz operar. El nodo SDN deberá saber a priori por donde reenviar los paquetes de control. Dicha configuración se obtiene de algún tipo de protocolo de comunicación *in-band*, el cual proporcione a cada nodo de la red la capacidad de alcanzar el nodo que les da acceso al controlador de la red SDN.

Como se ha indicado anteriormente, no hay paradigma mejor que otro, cada cual ofrece unas características propias que tendrán un comportamiento mejor o peor en función del caso de uso. Por ejemplo, en un entorno IoT, donde los dispositivos a lo sumo tienen una única interfaz de comunicaciones, el modelo *in-band* sería ideal. El coste de añadir otra interfaz de comunicaciones no es solo monetario, sino que también impacta en la vida útil del sensor, al tener que alimentar una interfaz de comunicaciones adicional.

1.3. Objetivos

El objetivo principal de este proyecto es conseguir el desarrollo de un protocolo de control *in-band* eficiente para la integración de la tecnología IoT en las futuras de redes de 6G. Como ya hemos introducido, con la llegada del IoT la dimensión de las redes va a crecer exponencialmente. Por lo consiguiente, la complejidad de la administración de dichas redes

va a suponer un gran desafío. Los dispositivos IoT se podrán beneficiar de la integración con las redes SDN en 6G, ya que estas les reportará la flexibilidad y programabilidad requerida para una correcta gestión y administración de cada elemento de la red. Por ello, se pueden resumir los objetivos del proyecto en los siguientes puntos:

- **Analizar el estado del arte y necesidades actuales de IoT en 6G.** Se realizará una búsqueda y recolección de información, artículos e informes sobre las demandas del IoT y las bondades preliminares del 6G.
- **Diseño de un protocolo in-band eficiente para IoT integrado con SDN.** Se realizará un estudio previo de las soluciones *in-band* ya implementadas, se analizarán, y se propondrá una solución a medida que cubra los equipos IoT en las redes SDN.
- **Emulación mediante plataformas como Contiki-NG, Mininet y ONOS.** El desarrollo inicial se probará sobre las plataformas más utilizadas en el prototipado de protocolos de comunicaciones.
- **Implementación y despliegue en hardware real** (tarjetas Raspberry Pi, o remotas IoT-LAB), en función de la ejecución del proyecto se estudiará la implementación del protocolo en *hardware* en real.

1.4. Estructura del TFM

En esta sección se indica la estructura organizativa de la memoria del proyecto TFM, haciendo un resumen de los aspectos más significativos de cada capítulo.

Capítulo 1: Introducción. Se hará una breve introducción de la motivación principal que ha originado la realización de este TFM, así como una breve explicación de los aspectos generales y de los objetivos que se quieren alcanzar con el trabajo presentado. Por último, se indicarán las contribuciones que se han conseguido con el mismo, en revistas científicas.

Capítulo 2: Estado del arte. Se indicarán los conceptos fundamentales en relación al proyecto. La motivación de este capítulo es la de establecer un marco teórico lo suficientemente consistente para abordar el diseño del protocolo de comunicación *in-band*.

Capítulo 3: Diseño del protocolo de control In-Band. Se analizará soluciones anteriores *in-band*, debatiendo las funcionalidades básicas que debe tener el protocolo. Se explicará el funcionamiento del mismo y se decidirá la plataforma para implementarlo.

Capítulo 4: Desarrollo y evaluación del protocolo. Se describirá el desarrollo realizado en la plataforma elegida. Señalando aquellas partes que se consideran de mayor importancia. Por último, se incluirá una evaluación del mismo.

Capítulo 5: Conclusiones y trabajo futuro. Se terminará la memoria de este TFM con las conclusiones, y se presentarán las vías de trabajo a futuro que tiene este proyecto.

Bibliografía y referencias. Se añadirán todos los artículos, libros, materiales consultados y empleados en la elaboración de esta memoria. Se seguirá el estilo de citación del Institute of Electrical and Electronics Engineers (IEEE), siguiendo las recomendaciones oficiales de la normativa sobre TFMs de la Universidad de Alcalá (UAH).

Anexos. Se incluirán todos los manuales de usuario e instalación que se consideren oportunos. De forma adicional, se añadirán las características técnicas del *hardware* con el cual se ha desarrollado este TFM. Por último, se hará un presupuesto que incluya el coste de mano de obra, material y gastos generales.

1.5. Contribuciones

Este TFM ha proporcionado significativas contribuciones a la comunidad científica, incluyendo tres publicaciones en revistas de alto impacto indexadas en el JCR (3 Q2). A continuación, se presentan estas contribuciones en resumen.

Artículos de revistas científicas de alto impacto.

1. Rojas, E., Hosseini, H., Gomez, C., **Carrascal, D.** and Cotrim, J.R., 2021. Outperforming RPL with scalable routing based on meaningful MAC addressing. *Ad Hoc Networks*, 114, p.102433. (JCR Q2)
2. Alvarez-Horcajo, J., Martinez-Yelmo, I., Rojas, E., Carral, J.A. and **Carrascal, D.**, 2022. ieHDDP: An Integrated Solution for Topology Discovery and Automatic In-Band Control Channel Establishment for Hybrid SDN Environments. *Symmetry*, 14(4), p.756. (JCR Q2)
3. **Carrascal, D.**, Rojas, E., Lopez-Pajares, D., Alvarez-Horcajo, J. and Carral, J.A., 2023. A Comprehensive Survey of In-Band Control in SDN: Challenges and Opportunities. *Electronics*, 12(6):1265. (JCR Q2)

2. Estado del arte

En el presente capítulo se describirán los conceptos fundamentales relacionados con el proyecto, así como las diversas herramientas que se utilizarán mayoritariamente. El propósito de este capítulo es establecer un marco teórico sólido que permita abordar el análisis y el diseño del protocolo de comunicación *in-band* de manera óptima antes de su desarrollo. Finalmente, se hará referencia a las contribuciones y la documentación generada con el fin de difundir los contenidos del proyecto a través de la plataforma *GitHub*.

2.1. Red de comunicación 6G

2.2. Tecnología IoT

La premisa básica de la tecnología IoT [19] conectar cualquier dispositivo que tenga cierta capacidad de cómputo. Esto significa que los objetos que actualmente no están conectados a Internet, estarán conectados de manera que puedan comunicarse e interactuar con personas y otros objetos.

La tecnología IoT es una transición tecnológica en la que los dispositivos al ser dotados de inteligencia por estar conectados a Internet podrán proveer de entornos inteligentes a los humanos. Cuando los objetos puedan ser controlados a distancia a través de una red, se habilitará una integración más estrecha entre el mundo físico y las máquinas, permitiendo mejoras en las áreas de medicina, automatización y logística [20].

El ecosistema IoT es amplio, e incluso se puede parecer un poco caótico debido a la gran cantidad de componentes y protocolos que abarca. Es recomendable en vez de ver el IoT como un término único, verlo como un paraguas de varios conceptos, protocolos y tecnologías, enfocados a un mismo propósito de interconectar “Cosas” a Internet. Si bien la amplia mayoría de elementos IoT están diseñados para aportar numerosos beneficios en las áreas de productividad y automatización, al mismo tiempo introducen nuevos desafíos, como por ejemplo la gestión de la gran cantidad de dispositivos que van a aparecer en las redes, y la cantidad de datos y mensajes que todos estos generarán [19].

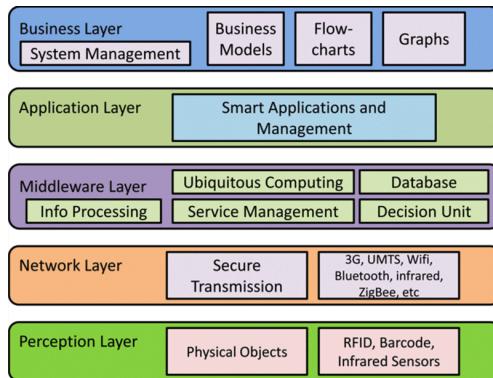
2.2.1. Arquitectura

Aunque en el ecosistema hay diferentes *stacks* de protocolos, todos ellos se pueden resumir en la siguiente arquitectura básica [19].

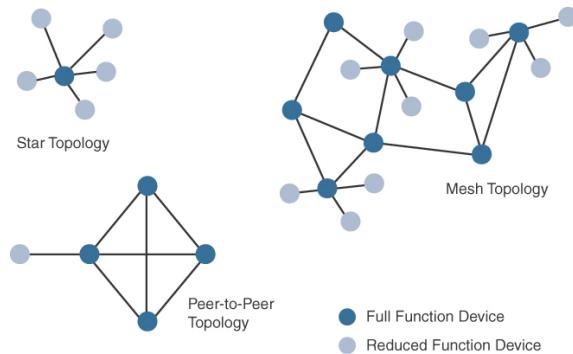
- *Perception Layer*, en esta capa se da un significado físico a cada objeto. Consiste en sensores de diferentes tipos como etiquetas RFID, sensores IR u otras redes de sensores que podrían detectar la temperatura, la humedad, la velocidad y la ubicación, etc. Esta capa recolecta información útil a partir de los sensores vinculados a los objetos, convierte dicha información en señales digitales que más tarde se delegarán a la Capa de Red para su posterior transmisión.
- *Network Layer*, el propósito de esta capa es la de recibir la información en forma de señales digitales desde la capa de Percepción y transmitirla a los sistemas de procesamiento en la capa de *Middleware*. Esto se llevará a cabo a través de las distintas tecnologías de acceso como WiFi, BLE, WiMax, ieee802154 y con protocolos como IPv4, IPv6, MQTT.
- *Middleware Layer*, en esta capa se procesa la información recibida de todos los sensores. En esta capa se puede incluir las tecnologías como Cloud computing, o sistemas gestores, que aseguran un acceso directo a bases de datos donde se puede almacenar toda la información recolectada. Teniendo una gran cantidad de información centralizada, generalmente se aplican sistemas de inteligencia artificial para procesar la información, y tomar decisiones predictivas totalmente automatizadas. Estos sistemas suelen ser utilizados para analizar el tiempo, contaminación o tráfico en las ciudades.
- *Application Layer*, la finalidad de esta capa es la de realizar las aplicaciones IoT para el usuario final. Estas aplicaciones se valdrán de los datos procesados para ofrecer funcionalidades al usuario final, por ejemplo, una aplicación del tiempo.
- *Business Layer*, esta capa aunque es un poco abstracta, se suele añadir para representar la gestión de múltiples las aplicaciones y servicios IoT.

2.2.2. Topologías

Entre las tecnologías de acceso disponibles para conectar los dispositivos de IoT, dominan tres esquemas topológicos principales, estrella, malla y p2p. Para las tecnologías de acceso de largo y corto alcance, predomina la topología de estrella, como se puede encontrar en las redes datos móviles, LoRa y BLE. Las topologías en estrella utilizan una única estación base central para permitir las comunicaciones con los nodos finales. En cuanto a las tecnologías de mediano alcance, se pueden encontrar topologías en estrella, de p2p o en malla, como se

**Figura 2.1:** Arquitectura básica IoT

ve en la figura 2.2. Generalmente se suele hacer uso de un tipo de topología sobre otra en función de las limitaciones de los nodos que la conforman [21].

**Figura 2.2:** Tipos de topología con dispositivos IoT [21]

2.2.3. Redes LLN

Las redes de baja capacidad, conocidas como, Low power and Lossy Networks (LLN)¹, se caracterizan por estar compuestas de dispositivos (sensores, motas) con limitaciones de memoria, batería y procesamiento. Dichos nodos, se interconectarán haciendo uso de distintos tipos de enlace, como por ejemplo ieee802154 ó LowPower-WiFi [22]. Este tipo de redes pueden estar presentes en distintos campos de aplicación, entre las que se incluyen asistencia sanitaria, monitorización industrial, redes de sensores, etc.

Otra condición de las redes LLN, son las pérdidas en capa física debidas a las interferencias y variabilidad de los “complicados” entornos radio donde estarán desplegadas dichas redes.

¹<https://tools.ietf.org/html/rfc7228>

Por ello, y teniendo en cuenta que los nodos que formarán parte de las redes LLN serán de baja capacidad, es necesario que los protocolos utilizados en esta red sean capaces de optimizar al máximo los recursos de los dispositivos [21].

2.2.3.1. IEEE 802.15.4

El estándar ieee802154² define la tecnología acceso a un entorno inalámbrico (*phy* y *mac*) para dispositivos de baja capacidad (limitados en batería y capacidad de transmisión). Este estándar se caracteriza por la optimización de los recursos del nodo en cuestión, consiguiendo una duración prolongada de la batería, además, esta tecnología de acceso permite un fácil uso utilizando un *stack* de protocolos compacto, al tiempo que sigue siendo simple y flexible. Por todas estas características, el estándar ieee802154 es usado en la mayoría de *stack* de protocolos enfocados al IoT. Uno de los más utilizados es el *stack* 6LoWPAN, definido por el Internet Engineering Task Force (IETF), consiste en una capa de adaptación de IPv6 sobre las capas del estándar ieee802154 (Ver figura 2.3) [23].

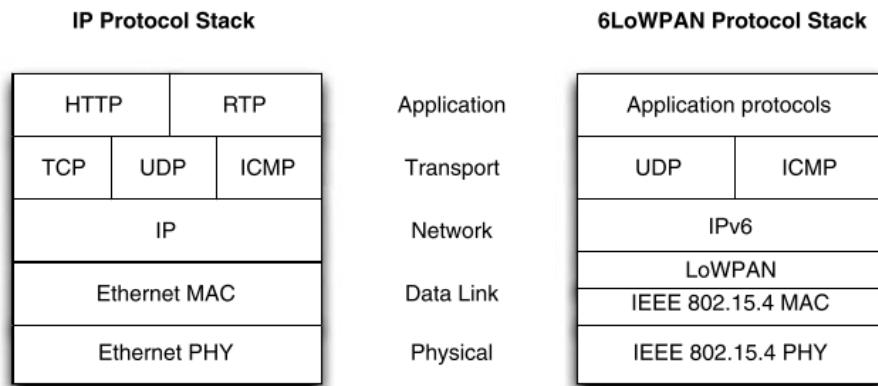


Figura 2.3: Pila de protocolos 6LoWPAN [23]

2.3. Redes SDN

El paradigma SDN [24] consiste en una arquitectura de red donde se lleva a cabo la separación del plano de control de la red para centralizarlo en un único ente llamado controlador. De esta forma se consigue que la administración de red sea una tarea más centralizada y flexible [24]. La idea del SDN empezó a germinar en la Universidad de Stanford por el año 2003, donde el profesor asociado en su momento, Nick McKeown, planteaba las limitaciones

²<https://tools.ietf.org/id/draft-ietf-lwig-terminology-05.html>

de las redes convencionales y veía la necesidad de replantear como los *backbones* debían operar [25]. Dicha idea se acuñó como SDN en el año 2011, cuando a la par se lanzó la organización Open Networking Foundation (ONF) [26] como un portador de los estándares relacionados con el SDN y su difusión.

2.3.1. Arquitectura

La arquitectura SDN destaca por ser dinámica, rentable y adaptable haciéndola ideal para las demandas presentadas hoy en día por las redes de comunicaciones. Como ya se ha comentado, la arquitectura se basa en separar el plano de control del plano de datos, y llevar ese plano de control a una entidad llamada controlador. Desde dicha entidad se ofrecerán las interfaces necesarias para que aplicaciones de servicios de red puedan hacer uso de ellas. De esta forma, el control de la red se vuelve directamente programable, consiguiendo que la gestión se vuelva más ágil y dinámica.

Como se puede ver en la figura 2.4, la arquitectura SDN se divide en tres capas, la primera, el plano de datos que contendrá todos los elementos de red que habiliten el forwarding. La segunda, el plano de control, compuesto de los distintos controladores de la red SDN, y por último, la capa de aplicación, en la cual se encontrarán todas las aplicaciones que se comuniquen con el controlador SDN.

Dichas capas se comunicarán entre ellas a través de interfaces abiertas. Por ejemplo, la interfaz *Southbound* permite programar el estado de reenvío de los elementos de red del plano de datos. En cambio, la interfaz *Northbound* comunica las aplicaciones con los controladores SDN, habilitando la obtención de datos o el ajuste de parámetros a través de una API-Rest. También se pueden encontrar otro tipo de interfaces, *Westbound* y *Eastbound*, que se han consolidado los últimos años para la interconexión de controladores con la finalidad de establecer una misma política entre distintos dominios SDN.

2.3.2. OpenFlow

Existen varios protocolos para el control de los elementos de red desde el controlador, pero el más utilizado es OpenFlow. OpenFlow es un protocolo de la interfaz *Southbound* que comunica los controladores SDN con los elementos de red para configurar el estado de reenvío de estos últimos. La especificación de este protocolo se encuentra recogida por la ONF³, contando con numerosas versiones siendo la última la versión 1.5.1 del 2015.

³<https://www.opennetworking.org/software-defined-standards/specifications/>

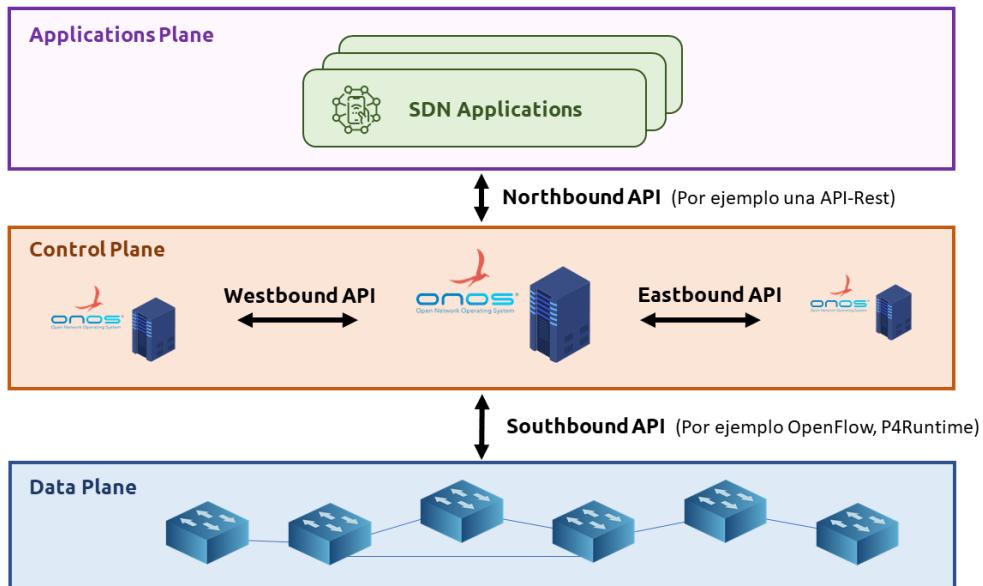


Figura 2.4: Arquitectura básica SDN

El elemento clave de OpenFlow es el flujo (*flow*), los cuales se conforman de paquetes que ha sido clasificados en función de reglas. Dichas reglas se encuentran en las tablas de flujo (*flow table*) y suelen estar relacionadas con los puertos de entrada o valores de cabecera típicos. Cuando estos criterios coinciden con los del paquete entrante se produce un ***match***.

En el momento en que se produce un *match*, el paquete en cuestión se verá sujeto a una serie de instrucciones asociadas a la regla con la que a hecho *matching*. Estas instrucciones pueden ir desde, hacer una medición del paquete, aplicar una acción ó ir a otra tabla de flujo. De esta forma, con unas tablas de flujo completadas con unas reglas suministradas por el controlador SDN, se conforma el estado de reenvío del switch en cuestión [24].

2.4. Controladores SDN

Los controladores SDN, también conocidos como sistemas operativos de red, son una pieza clave en los entornos SDN dado que tienen una vista global de toda la red que gestionan, que flujos atraviesan la red, estadísticas, usuarios finales, modelos y características de dichos equipos. Este controlador tiene la funcionalidad de interconectar los recursos disponibles en la red que gestiona, a aplicaciones o servicios que corran encima de él [24]. De esta forma, cada vez que se quiera añadir una nueva funcionalidad, solo se tendrá que programar un nuevo servicio que corra encima del controlador que gestiona la red, y este a su vez se encargará de traducir las demandas del servicio a políticas de red a cada dispositivo impactado.

En este matiz se puede llegar a apreciar el sentido de nombre del paradigma SDN, ya que estamos definiendo por software el comportamiento intrínseco de la red.

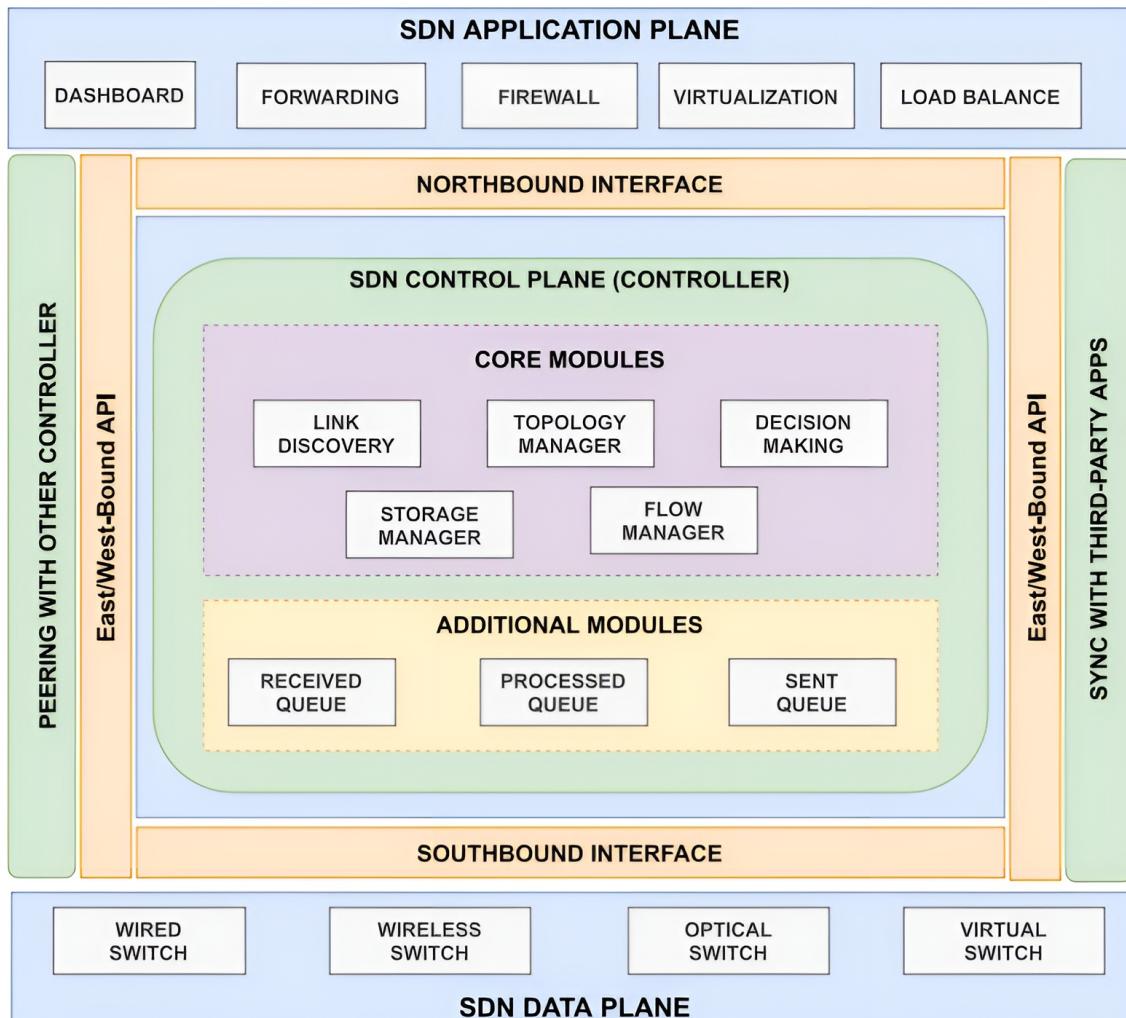


Figura 2.5: Arquitectura genérica de controlador SDN [27]

En la literatura hay muchas propuestas de arquitecturas para los controladores, pero en vez de ir una por una viendo las diferencias vamos a presentar la arquitectura genérica que se puede encontrar en la gran mayoría de controladores SDN. Si nos fijamos en la figura 2.5, podemos ver dos partes claramente diferenciadas, el núcleo del controlador y las interfaces del mismo [27]. Empezando por el *core*, se puede resumir que las funciones básicas del controlador están relacionadas principalmente con el descubrimiento de la topología y la gestión de los flujos de tráfico. El módulo de descubrimiento de la topología suele trabajar con el

protocolo Link Layer Discovery Protocol (LLDP). La implementación puede variar entre controladores y aplicaciones de descubrimiento topológico, pero en términos generales, se transmite regularmente consultas utilizando mensajes `packet_out`, los cuales viajarán por la topología física y volverán al controlador en forma de mensajes `packet_in`, que permiten al controlador construir la topología de la red.

Una vez que se conoce la topología de red, el controlador puede empezar a poner en marcha distintos módulos de toma de decisiones para encontrar los caminos óptimos entre los nodos de la red. Con todos los caminos ya construidos, entran en juego otros módulos del controlador como son QoS y de seguridad, los cuales pueden optar por instalar una ruta sub-óptima para satisfacer los criterios de QoS o de seguridad. De forma adicional, el controlador puede tener un recopilador de estadísticas y un gestor de colas para recopilar información sobre el rendimiento de las diferentes colas de paquetes entrantes y salientes de los dispositivos de red que gestiona, y con ellos realimentar a los módulos de QoS. Por último, tenemos uno de los módulos más importantes del controlador, el gestor de flujos. El gestor de flujos, puede variar su implementación en función de los protocolos que se utilicen en la Southbound Interface (SBI), pero su misión es la misma, instalar reglas en los dispositivos de red que gestionan las directrices necesarias para gestionar los paquetes de un determinado flujo de una determinada manera.

Siguiendo con otra parte fundamental del controlador SDN genérico, son las interfaces. El controlador está rodeado de interfaces para interactuar con otras capas, superior e inferior, y otros controladores, este y oeste (E-WBIs). Empezando por la interfaz SBI, la cual es la encargada de interconectar dispositivos SDN con el controlador, define un conjunto de reglas, que variarán en función del protocolo que se utilice, las cuales permiten definir el procesamiento y las políticas de reenvío de los dispositivos SDN. El protocolo OpenFlow es una de las SBI más utilizadas, y es un estándar de facto para la industria, con el cual podemos definir flujos y clasificar el tráfico de red basándose en un conjunto de reglas predefinidas. Pero también se pueden encontrar otras SBI, como por ejemplo, P4Runtime de facto el futuro para las SBIs, o podemos encontrar algunas más *legacy*, como por ejemplo, Netconf o incluso Simple Network Management Protocol (SNMP).

Si nos vamos de la API sur, al norte, encontraremos la conocida como Northbound Interface (NBI), la cual interconecta el controlador SDN con las aplicaciones de los desarrolladores o los servicios que definen el comportamiento intrínseco de la red. Los controladores admiten varias interfaces de programación de aplicaciones (API) northbound, pero la mayoría de ellas se basan en la API REST. Generalmente se quiere que la interfaz NBI sea una interfaz genérica para que limite a los desarrolladores. Para la comunicación entre controladores, se

utilizan las interfaces conocidas como de este y oeste (E/WBI), las cuales no tienen una interfaz de comunicación estándar, por lo que, en función del controlador se tendrá una implementación u otra.

A continuación, se van a ir presentando algunos de los controladores SDN más conocidos, indicando algunas de sus virtudes y funcionamiento en particular.

2.4.1. Ryu

Ryu⁴ es un controlador de red de código abierto diseñado específicamente para redes SDN. Se desarrolló en Python por el equipo de NTT (*Nippon Telegraph and Telephone*) y proporciona una plataforma flexible, sencilla y extensible para desarrollar aplicaciones de red basadas en SDN. Como se comentó anteriormente, la funcionalidad primordial que lleva a cabo es la de ser un intermediario entre los elementos de red, como por ejemplo switches SDN o nodos virtuales, y las aplicaciones o servicios que controlan la red a través de la NBI [28]. De esta forma, a través de la NBI, permite a los desarrolladores programar el comportamiento de la red de manera dinámica y centralizada, facilitando la implementación de políticas de red, la configuración de routing y la gestión de tráfico.

Ryu es altamente modular y proporciona una API bien definida que permite a los desarrolladores construir aplicaciones de red personalizadas hasta el más mínimo nivel. También es compatible con varios protocolos de comunicación utilizados en SDN, como OpenFlow (versiones 1.0, 1.2, 1.3, 1.4, 1.5), NETCONF y OF-config [28]. Las aplicaciones Ryu son entidades que implementan varias funcionalidades dentro de Ryu y se comunican entre sí a través de eventos. Los eventos sirven como mensajes intercambiados entre las aplicaciones Ryu. Estas aplicaciones se envían eventos asíncronos entre sí, creando un flujo de comunicación. Además de las aplicaciones Ryu, existen ciertas fuentes de eventos internas al propio Ryu. Un ejemplo de estas fuentes de eventos es el controlador OpenFlow. Cada aplicación Ryu posee una cola de recepción específicamente diseñada para eventos. La cola funciona según el principio First In, First Out (FIFO), lo que garantiza que se mantenga el orden de los eventos. Para procesar estos eventos, cada aplicación Ryu tiene un único hilo dedicado responsable de la gestión de eventos. Este subproceso vacía continuamente la cola de recepción retirando los eventos de la cola e invocando al manejador de eventos apropiado en función del tipo de evento [29].

⁴Del Japonés, significa "flujo" y también "dragón", ambos símbolos Kanji se leen igual como RYU, de ahí que el logo sea un dragón.

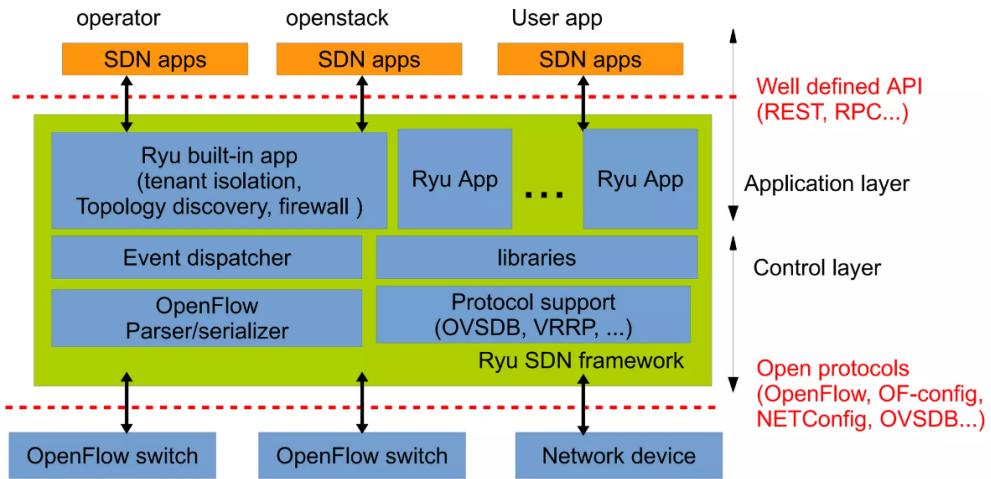


Figura 2.6: Arquitectura del controlador SDN RYU [30]

En la figura 2.6, se puede apreciar como la arquitectura del controlador está dividida en dos grandes bloques de acuerdo se han estudiado los controladores SDN, la parte de *core* y la parte de interfaces. En la parte del núcleo se puede apreciar como se ha programado todas las librerías y el manejador de eventos que se mencionaba anteriormente. Por otro lado también se puede apreciar las capas de adaptación a la SBI y a la NBI, la primera de ellas se adapta a través de una API REST a servicios de aplicaciones externas, mientras que la interfaz SBI implementa todos los protocolos de control SDN. Algunas de las características y funcionalidades de Ryu incluyen:

- Compatibilidad con múltiples protocolos SDN de la interfaz SBI.
- Capacidad para implementar políticas de red y reglas de enrutamiento de forma sencilla.
- Soporte para la recopilación y el análisis de datos de red en tiempo real.
- Funcionalidad de control de QoS.
- La más importante de todas, fácil desarrollo y portabilidad sencilla al estar escrita en Python. Esta última se puede ver también como una desventaja, dado el pobre rendimiento de un lenguaje interpretado.

Ryu es utilizado en una amplia gama de entornos, desde laboratorios de investigación hasta en las clases de forma educativa. Esta herramienta suele ser el punto de entrada para muchas personas que se inician en el SDN, sin embargo, al tener un pobre rendimiento, en entornos comerciales no se suele ver con tanta frecuencia [28].

2.4.2. ONOS

Open Network Operating System (ONOS) es un controlador de red de código abierto diseñado específicamente para redes SDN. Como su nombre indica, ONOS es un sistema operativo para redes que proporciona funcionalidades avanzadas de control y gestión de redes. ONOS está diseñado para ser escalable, confiable y de alto rendimiento, lo que lo hace adecuado para despliegues de red a gran escala. Es compatible con una amplia variedad de protocolos y tecnologías de red, como OpenFlow, NETCONF, BGP y P4 [31]. El controlador está respaldado por la ONF, la cual es una organización sin ánimo de lucro fundada en 2011 con el objetivo de promover y acelerar la adopción de la tecnología SDN y el enfoque de redes abiertas. Además de la ONF, numerosos proveedores de internet están impulsando el proyecto, así como, grandes empresas del sector TIC. Se pueden resumir las principales características y funcionalidades de ONOS en los siguientes puntos.

- Control centralizado de la red: ONOS proporciona un punto central de control para la gestión de toda la red. Permite la configuración dinámica de la red, el enrutamiento y la asignación de recursos.
- Escalabilidad: ONOS está diseñado para manejar redes de gran escala, distribuyendo la carga de trabajo entre múltiples nodos para lograr un rendimiento óptimo y una alta disponibilidad.
- Programabilidad: ONOS permite a los desarrolladores crear aplicaciones personalizadas utilizando una amplia gama de APIs y marcos de desarrollo. Esto facilita la implementación de políticas de red, la orquestación de servicios y la integración con otras aplicaciones y sistemas.
- Gestión de topología: ONOS proporciona una visión global de la topología de la red, permitiendo el descubrimiento de dispositivos, enlaces y rutas. Esto facilita la toma de decisiones basadas en el estado actual de la red.
- Gestión de flujos: ONOS admite la programación y gestión de flujos de red, lo que permite la implementación de políticas de enrutamiento y QoS de manera dinámica y centralizada.
- Segmentación de red: ONOS es compatible con la segmentación de red, lo que permite crear *network slices* para proporcionar aislamiento y asignación de recursos personalizada en una infraestructura compartida.

ONOS se concibe como un sistema operativo de red completo que va más allá de ser solo un controlador SDN. Ofrece una amplia gama de funcionalidades que incluyen herramientas

como APIs que proporcionan abstracción para el desarrollo de aplicaciones SDN, así como APIs para la administración, supervisión y programación de dispositivos de red. Además, ONOS ofrece capacidades de virtualización, aislamiento, acceso seguro y abstracción de los recursos de red administrados por el sistema operativo. ONOS tiene la capacidad de multiplexar recursos tanto hardware como software entre las aplicaciones SDN, permitiendo una utilización eficiente de los recursos disponibles. Además, facilita la configuración de políticas de red basadas en las intenciones de las aplicaciones, lo que implica la aplicación de políticas de red diseñadas para satisfacer los requisitos específicos de las aplicaciones, así como el procesamiento de eventos de red [32].

Si nos fijamos en la figura 2.7, podemos ver que este sistema operativo está diseñado para operar con dispositivos de red *whitebox*, con el objetivo de reducir los costes asociados con las soluciones propietarias. ONOS posee una arquitectura flexible que facilita la integración sencilla de nuevos dispositivos de hardware en el framework SDN. Solo se requiere que se añada un driver del equipo o target propietario. Además, puede funcionar como un sistema distribuido a través de múltiples servidores en modo cluster, lo que permite aprovechar los recursos de CPU y memoria de varios equipos simultáneamente. Esta capacidad también proporciona resiliencia ante posibles fallos de los servidores y permite realizar cambios en hardware y software sin interrumpir el tráfico de red [33].

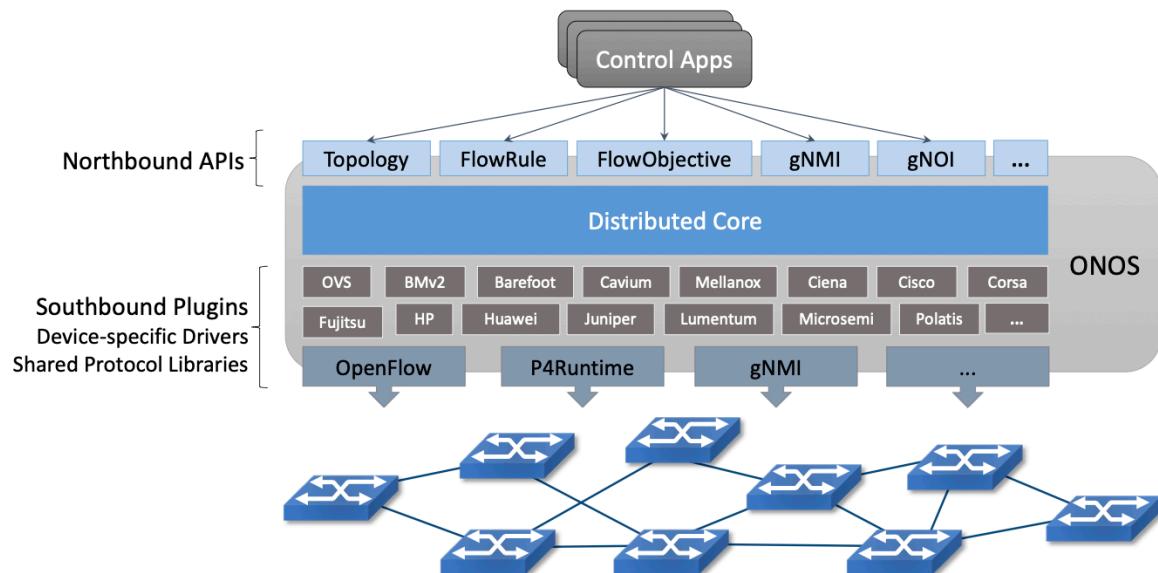


Figura 2.7: Arquitectura del controlador SDN onos [33]

ONOS cuenta con una comunidad activa de desarrolladores y usuarios que contribuyen al desarrollo y mejora del sistema. Además, ONOS es utilizado en diversos casos de uso, como

redes de transporte, redes de centros de datos y redes de telecomunicaciones, entre otros muchos casos de uso. Si bien es cierto que tiene un muy buen rendimiento, no es controlador sencillo de manejar, y la curva de aprendizaje puede ser bastante pronunciada.

2.5. Software Switches SDN

En las redes definidas por software, los software switches desempeñan un papel fundamental al permitir la virtualización y la gestión centralizada de las redes. Estos switches, a diferencia de los switches de hardware tradicionales, se implementan como software y se ejecutan en servidores convencionales. Un software switch en SDN, en adelante *softswitch*, es una entidad lógica que reside generalmente en una instancia virtual o en un servidor, y se comunica con el controlador SDN para recibir instrucciones sobre cómo procesar los paquetes de datos que fluyen a través de la red. Al estar basados en software, estos switches pueden ser escalados y desplegados de manera flexible según las necesidades y demandas de la red. La principal ventaja de los *softswitches* radica en su capacidad para adaptarse y responder de manera dinámica a las necesidades de la red. Pueden implementar diferentes funciones de red, como enrutamiento, commutación, balanceo de carga y seguridad, a través de la instalación de reglas desde el controlador SDN.

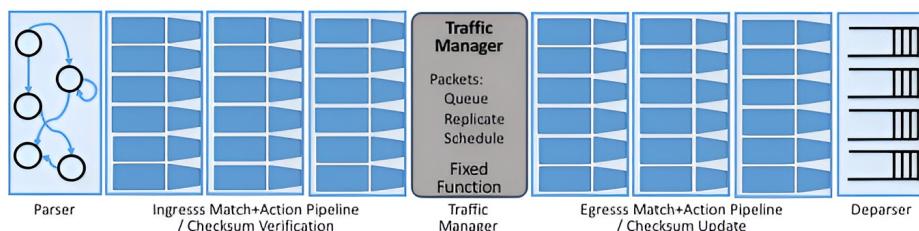


Figura 2.8: Arquitectura genérica de un *softswitch* SDN [34]

Según se puede apreciar en la figura 2.8, la arquitectura genérica de un *softswitch* se puede resumir en los siguientes bloques. El primero de todos tiene que ser un parser, que vaya inspeccionando los paquetes entrantes a la pipeline de procesamiento del switch para identificar qué tipo es. Una vez se ha identificado el paquete que se va a procesar, el siguiente bloque son las tablas de *match-action*, las cuales tienen una interfaz de comunicación con el controlador SDN para establecer criterios y campos de *match*, y en caso de haber un *match*, definir una serie de acciones para llevar a cabo. En función del *softswitch*, la verificación de *checksum*⁵ se puede llevar a cabo en el parser o en la etapa de las tablas de *match-action*.

⁵Suma de comprobación, campo para comprobar la integridad del paquete de datos

El siguiente bloque que nos podemos encontrar en un *softswitch* es el conocido como gestor de tráfico, que variará en función de la implementación, pero nos proveerá de gestión de colas, QoS, duplicado de paquetes, etc. La siguiente etapa ya es más opcional, que se suele denominar como *egress match-action*, la cual se puede utilizar para definir algún tipo de lógica a la salida de los switches, aunque en la realidad se suele utilizar para actualizar los campos de TTL y recalcular el *checksum* dado que el paquete se habrá visto modificado. El último bloque es el deparser, el cual se encarga de ensamblar de nuevo el paquete y prepararlo para sacarlo por el puerto de salida del switch.

2.5.1. OvS

Open vSwitch (OvS), es uno de los *softswitches* de referencia en el mundo de las redes, ampliamente utilizado tanto en la industria como en la academia. El switch está ofrecido como un proyecto opensource y respaldado por la Linux Foundation. Es uno de los software switches más utilizados y ampliamente adoptados en entornos SDN debido a su flexibilidad y funcionalidades avanzadas, aunque si tiene que destacar por algo, es por su gran rendimiento al trabajar a nivel de Kernel siendo idóneo para entornos de producción [35]. El OvS actúa como un switch virtual, proporcionando capacidades de conmutación y reenvío para máquinas virtuales y contenedores en entornos de virtualización. Puede ejecutarse en hipervisores populares, como KVM (Kernel-based Virtual Machine), Xen y VMware, y también puede ser implementado como un switch independiente en instancias o servidores en modo standalone. Las características clave de Open vSwitch incluyen:

- Agente SDN: el switch ofrece una interfaz que permite que sea controlado mediante un controlador SDN, como OpenDaylight o ONOS. Esto permite una gestión centralizada y un control más granular de la red, así como la implementación de políticas de red definidas por software.
- Funcionalidades avanzadas de alto rendimiento: OvS ofrece una amplia gama de funcionalidades, como fast-forwarding al trabajar con frameworks como eXpress Data Path (XDP), DPDK o extended Berkeley Packet Filter (eBPF). Estas características permiten una gestión eficiente de la red, y entorno de altas prestaciones ofreciendo un alto rendimiento perfecto para despliegues de producción.
- Integración con tecnologías de virtualización: OvS se integra estrechamente con tecnologías de virtualización como OpenStack y Docker. Puede proporcionar conectividad de red entre máquinas virtuales, contenedores y hosts físicos, facilitando la migración y la gestión de recursos en entornos virtualizados.
- Extensibilidad y soporte para estándares: El OvS es altamente extensible y se puede ampliar mediante la integración de módulos y complementos personalizados. Por

ejemplo, para la integración del lenguaje de P4 se está haciendo de forma paulatina mediante módulos. Además, cumple con los estándares de la industria, como el protocolo OpenFlow, para garantizar la interoperabilidad con otros componentes SDN.

Si nos fijamos en la figura 2.9, podemos apreciar los componentes principales de la arquitectura del OvS. Como se puede apreciar en la figura hay dos partes claramente diferenciadas en el *softswitch*, una de espacio de usuario y otra parte de espacio de Kernel. Esta última es la que proveerá al OvS de un alto rendimiento en comparación con otros software switches. Los componentes principales del OvS se pueden resumir en los siguientes puntos [35].

- **ovs-vswitchd**, es un *daemon* que corre en espacio de usuario que implementa el switch, este proceso corre de forma conjunta con un módulo que corre en espacio de kernel, los cuales se comunican por netlink⁶ para establecer la política de gestión de flujos.
- **ovsdb-server**, es un servidor de base de datos ligera, a la cual el proceso de espacio de usuario **ovs-vswitchd** le hace queries para obtener su configuración.
- **ovs-dpctl**, es una herramienta de gestión del módulo del kernel que implementa el datapath. El nombre de la herramienta hereda del nombre puesto a la herramienta original **dpctl**, la cual fue propuesta por Stanford en 2009⁷, y es un acrónimo de *datapath-control*.
- **ovs-vsctl**, es una utilidad para gestionar la configuración del proceso de espacio de usuario **ovs-vswitchd**.
- **ovs-appctl**, es una herramienta para mandar comandos a los *daemons* OvS.
- **ovs-ofctl**, es una herramienta con la cual podemos mandar comandos y controlar vía OpenFlow el funcionamiento de switch.
- **ovs-testcontroller**, es un controlador experimental que implementan para testear la recepción y manejo de mensajes OpenFlow.

En resumen, el OvS es un software switch SDN ampliamente utilizado que proporciona capacidades avanzadas de conmutación y enrutamiento para entornos de virtualización. Su flexibilidad, funcionalidades avanzadas, rendimiento y compatibilidad con estándares lo convierten en una opción popular para implementaciones SDN en diversos entornos, desde centros de datos hasta infraestructuras de proveedores de servicios.

⁶<https://man7.org/linux/man-pages/man7/netlink.7.html>

⁷<https://github.com/mininet/openflow/blob/master/utilities/dpctl.c>

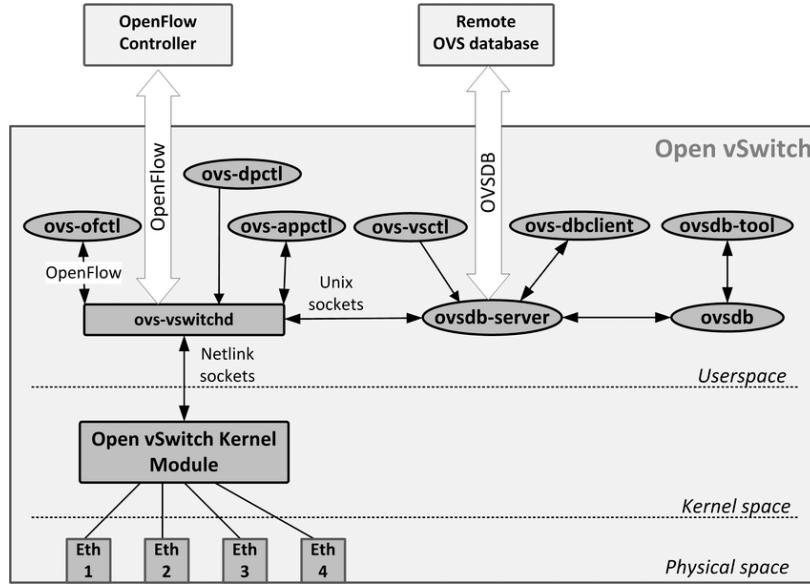


Figura 2.9: Arquitectura del OvS [36]

2.5.2. BOFUSS

Basic OpenFlow User-space Software Switch (BOFUSS), es la otra gran alternativa en el mundo de los software switches SDN. Este switch nació como una primera implementación por el 2008 en la universidad de Stanford, acuñado como *The Stanford Reference OpenFlow Switch*. Esta implementación era un mínimo producto viable para demostrar y ayudar en el proceso de estandarización del protocolo **Openflow 1.0**. Dicho mínimo producto viable, fue retomado por los laboratorios de Ericsson, *Ericsson Research TrafficLab*, para desarrollar la versión **Openflow 1.1** [37].

Este último desarrollo fue retomado por el investigador Eder Leão Fernandes, desde el CPqD de Brasil, fue parte de su trabajo fin de máster y tesis doctoral, donde completo lo que hoy conocemos como el BOFUSS, el cual da soporte para la versión **Openflow 1.3**. Por último, se verá más adelante que este switch fue tomado por Boby Nicusor Constantin y modificado para hacer una implementación de control In-band. En la siguiente figura, ver figura 2.10, se puede apreciar un pequeño resumen de la historia del software switch BOFUSS.

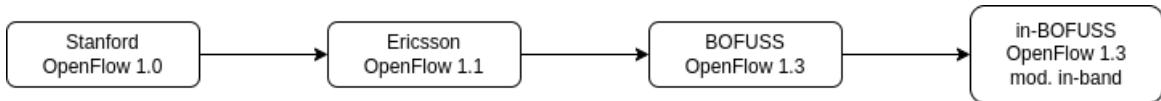


Figura 2.10: Evolución del BOFUSS

La arquitectura del BOFUSS se puede apreciar en la figura 2.11. Según se ha podido

averiguar, por el propio creador, esta arquitectura si bien es cierto que no trata de replicar la especificación de OpenFlow al 100%, es la implementación más cercana a la especificación oficial de **Openflow 1.3**. Como se puede ver en la figura siguiente (Figura 2.11), el software switch se compone de dos bloques fundamentales.

- Plano de datos, *Datapath*, en la herramienta al plano de datos lo podemos encontrar como **udatapath/ofdatapath**.
- Plano de control, *Control plane*, en la herramienta al plano de control lo podemos encontrar como **secchan/ofprotocol**.

El primero de ellos, el **udatapath/ofdatapath** se caracteriza por ser el bloque funcional de gestionar el procesamiento de los paquetes datos, y en ocasiones de control (en función del paradigma de control). Dentro de este bloque funcional se pueden encontrar elementos internos como por ejemplo, los puertos, conocidos como **Port**, los **Flow**, **Meter**, **Group**, **Table** y el **Packet Parser**. El bloque del agente de control, es el encargado de gestionar la información de control entre el controlador y el dispositivo. Los mensajes de Openflow viajarán desde el plano de secure channel, a la librería de **oflib**, y de ahí al datapath para instanciar en las tablas de flujos correspondientes. Más adelante se explican en detalle cada bloque de la arquitectura.

2.5.2.1. Ports

Los puertos OpenFlow desempeñan un papel fundamental como puntos de entrada y salida para los paquetes de datos en un entorno OpenFlow. Cuando se ejecuta un software switch en una máquina, puede utilizar interfaces físicas o virtuales como sus puertos (interfaces físicas o virtuales como Virtual Ethernet Device (Veth) o también radio taps emulados). Los puertos físicos permiten el control de interfaces Ethernet o WiFi, lo que facilita la gestión de tanto topologías de red realistas como emuladas. Aunque la velocidad del software switch puede ser limitada dado que trabaja en espacio de usuario, la posibilidad de crear un entorno de pruebas mejora la experiencia de los usuarios que desarrollan y evalúan aplicaciones OpenFlow. Se podría pensar que los puertos del switch se limitan simplemente a enviar y recibir paquetes de red, pero en verdad, también tienen una serie de responsabilidades relacionadas con la gestión del protocolo OpenFlow. Estas responsabilidades se pueden resumir en los siguientes puntos.

- OpenFlow permite cierto nivel de control sobre el comportamiento que tiene que tener un puerto en particular. Si se recibe un mensaje de modificación de puerto, este tiene que permitir configurar el estado del puerto. Los puertos pueden configurarse para que descarten todos los paquetes recibidos, prohíban la generación de mensajes de

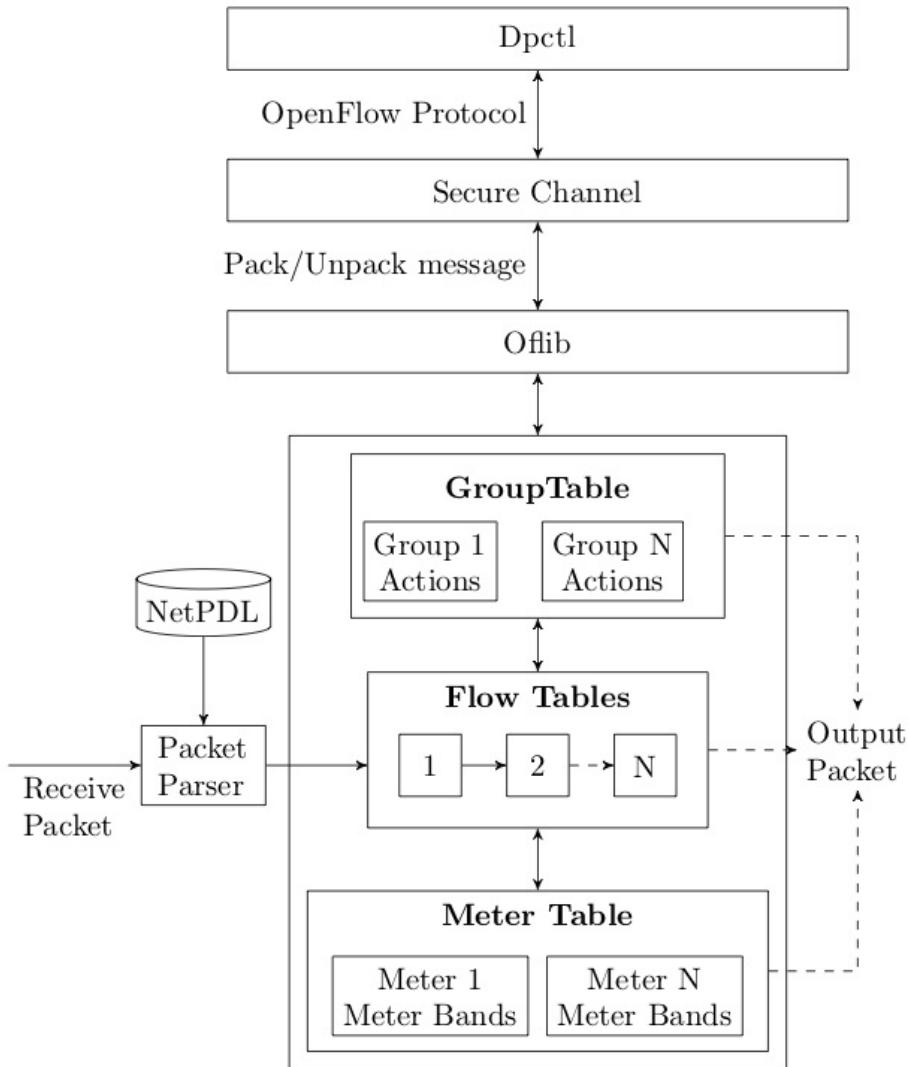


Figura 2.11: Arquitectura del BOFUSS [38]

tipo openflow **Packet-In** a partir de los paquetes que llegan, además de marcar el estado del puerto como fuera de servicio. El agente que gestione el puerto deben gestionar estos mensajes de configuración que le lleguen, y cambiar el comportamiento del puerto según la configuración recibida.

- Los puertos Openflow tienen que llevar un monitoreo del estado de la interfaz física o emulada que gestionan. Si bien es cierto que el controlador no puede actuar sobre el estado real de la interfaz, el switch tiene que informar sobre los cambios de estado del enlace.
- Generalmente cuando se lleva a cabo un **Packet-In** porque hay un *miss*, solo se man-

da al controlador las cabeceras del mensaje a consultar junto al propio mensaje del **Packet-In**. Los puertos, durante dicha consulta tendrán que gestionar los buffers que almacenarán los paquetes a consultar para ser procesados más tarde.

- El controlador a través del agente de control, también puede consultar sobre la descripción de un puerto. Por tanto, el software switch tendrá que recolectar la información que considere oportuna como por ejemplo la velocidad actual y máxima de las interfaces reales, almacenarla para enviarla posteriormente cuando el controlador se lo requiera.
- Las colas según se ha podido consultar no son parte de la definición estándar de Openflow. Sin embargo, Openflow puede configurar colas asociadas a unos puertos dados. Los puertos por tanto tendrán la responsabilidad de llevar a cabo la asociación de configuración de cola y asociación de cola con un puerto además de actualizar los contadores de paquetes de puerto y cola asociada.

2.5.2.2. Packet Parser

Antes que el paquete en cuestión llegue a la pipeline de procesamiento del software switch, este debe ser procesado para adaptarlo a las estructuras de datos que se manejan en el switch. Para ello, el cómo se tiene que parsear los paquetes se definieron en el estándar de **OpenFlow 1.1**. Esto es importante, ya que debe haber consistencia en como los paquetes deben ser pasados, pero esto a su vez supuso una limitación para nuevos diseños de switches, y supone modificaciones cada vez que se añade un nuevo protocolo. Por ello, más adelante con especificaciones posteriores del protocolo esta limitación se vio eliminada. El parser que se tiene en el BOFUSS hace uso de **NetBee** como disector y parseador de paquetes. Una vez que se han identificado los campos de protocolo que contiene el paquete, se crea una estructura de matching con la cual se pasará a la pipeline de procesamiento del switch. A continuación, en la figura 2.12 se puede ver la estructura básica del parser.

Este paso de procesado del paquete anteriormente descrito se puede dar en dos ocasiones. A continuación se indican.

- Que el paquete de red entre por uno de los puertos gestionados por el software switch.
- Que un paquete que ya ha sido modificado y redirigido por la pipeline de procesamiento, o enviado a una nueva tabla de adelante con la instrucción de **Go To Table**.

Esto se hace esta forma, dado que una revalidación del paquete es necesaria, y el parsear también se encarga de comprobar la validez del campo TTL. Además de añadir información de metadatos al mismo. Cada vez que se quiera dar soporte a nuevos protocolos, se tendrá que modificar el parsing del switch. Para ello, se tiene que llevar modificaciones a cabo en

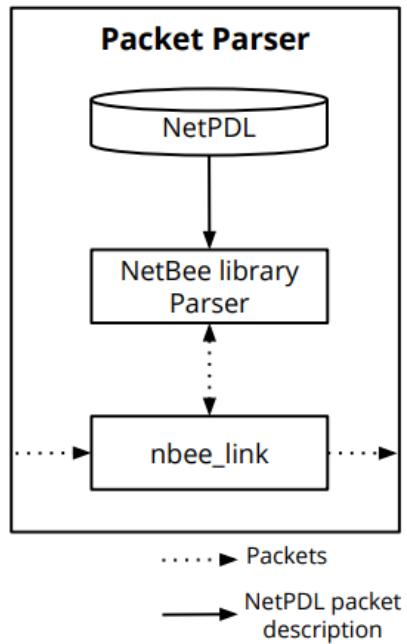


Figura 2.12: Parseador de paquetes del BOFUSS [38]

el fichero `*.xml` en lenguaje NetPDL. NetPDL es un lenguaje de descripción que describe cómo Netbee debe analizar los protocolos, la implementación actual del BOFUSS tiene su propio fichero ya creado, se puede consultar aquí⁸.

2.5.2.3. Flow Tables

Las *Flow Tables* son el core de procesamiento del software switch. Las flow tables siempre son el siguiente paso después del procesamiento y podrían considerarse el corazón del entorno openflow dado que son el primer componente en la pipeline de procesamiento del switch. Aunque el uso de múltiples tablas de flujos es opcional, la especificación indica que se utilice al menos una tabla, a la hora de la implementación se considera más que recomendado, dado que, es inviable el hecho de gestionar el escalado de una aplicación con únicamente solo una tabla de flujo.

En pocas palabras podríamos definir una *Flow Table* como una lista de flujos, donde cada flujo se compone de unos campos de matching y de unas instrucciones asociadas en caso de que haya match. Unas instrucciones que como se indican, tienen que estar definidas previamente en la especificación de Openflow. Una vez el paquete se ha validado y se ha

⁸<https://github.com/NETSERV-UAH/in-BOFUSS/blob/main/customnetpdl.xml>

parseado en una estructura de matching, se comprueba con una entrada de un flujo con el campo de matching, en caso de que coincida con dicho flujo, las instrucciones asociadas se ejecutan. A continuación, se indican algunos de los aspectos que tienen que cumplir las tablas de flujo.

- La implementación de una tabla de *miss* es obligatoria, ya que el switch tiene que hacer algo con los paquetes los cuales no coinciden con ninguna *flow entry*. Por el contrario, si no se implementa ninguna tabla de *miss*, se tiene que establecer una *action* por defecto. En el caso del switch, se ha establecido que se tiren los paquetes.
- Otro aspecto a considerar, el cual, tiene que ser gestionado por las *Flow table* es la gestión de los paquetes *Flow-Mod*. Estos mensajes son generados desde el controlador y gestionados por el agente de control del switch para creación o eliminación de entradas de flujo en alguna *Flow table*.
- El switch debe permitir al controlador de reconfigurar sus prestaciones. Es decir, las propiedades de las tablas deben ser conocidas por el controlador, y las tablas deben en todo momento responder a los mensajes de consulta de las características.
- Por cada paquete del plano de datos que les lleguen, deben llevar a cabo un *look up*, es decir, consultar si el paquete entrante de datos coincide con algún campo de match de algún flujo de la tabla. En caso de que exista algún match, se ejecutará la instrucción asociada.
- Otro aspecto a llevar a cabo por el switch, es llevar el recuento de las estadísticas de las entradas activas, los *look ups* realizados, y paquetes que han hecho match.

En cuanto aspectos de implementación, podemos destacar, las reglas se indexan en las tablas de flujos en orden de prioridad, si tienen la misma prioridad, se indexarán en orden de llegada. En cuanto a la complejidad del tiempo de *look up*, es lineal es decir $O(n)$, donde n es el número de flujos. Esto no es muy eficiente, debido a que crece de forma lineal según el número de flujos aumenta. Aunque según ha indicado el autor, es suficiente para llevar pruebas de concepto, sin embargo, no es adecuado para un entorno industrial de producción. Otro detalle de implementación, que tenemos que tener en cuenta es el número de entradas de flujos por tabla de flujos, actualmente está definido a 64 por una macro. Si se quisiera cambiar este parámetro solo habría que cambiar la macro y recomilar el proyecto. Por último, mencionar que las tablas de flujos tienen una lista de *idle* y *hard timeout*, que comprueban cada 100ms, para ver si alguna de sus entradas de flujos han expirado.

2.5.2.4. Group Table

Las *Group Tables*, se utilizan para agregar flow entries que tienen una política de acción similar. Cada grupo tiene un identificador único y contiene una lista de buckets que definen las acciones a tomar en caso de que el paquete coincida con ese grupo en particular. Cada bucket en una *Group Tables* define una serie de acciones a ejecutar para un paquete que coincide con ese grupo. Estas acciones pueden incluir enrutamiento, reenvío a puertos específicos, encapsulación, copia de paquetes, descarte, entre otras. Un bucket puede tener múltiples acciones y también puede contener una acción especial para indicar que el paquete debe ser procesado por otros grupos en cascada. La utilización de *Group Tables* permite un procesamiento más eficiente de los flujos de paquetes, ya que se pueden realizar acciones comunes de manera conjunta en lugar de procesar cada flujo de paquetes individualmente. Esto reduce la carga de procesamiento en los switches OpenFlow y facilita la implementación de políticas de red más complejas y flexibles. A continuación, en la figura ?? se puede apreciar la estructura de las *Group Tables*.

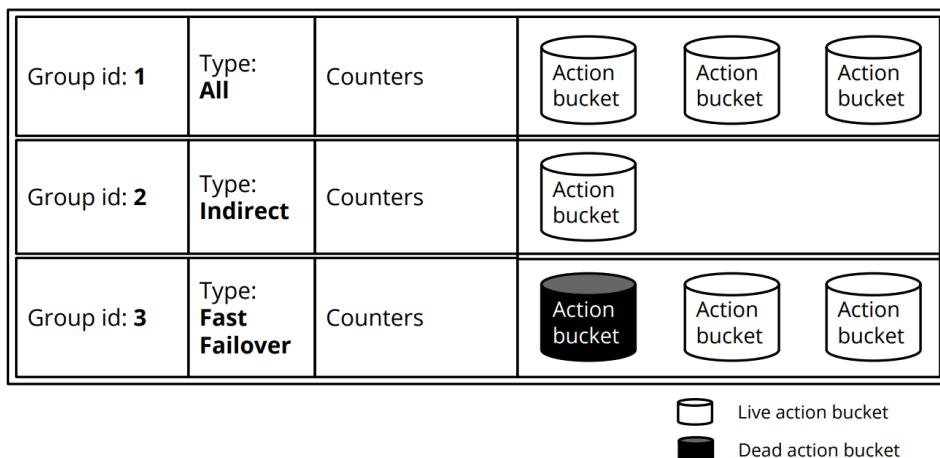


Figura 2.13: Estructura de las *Group tables* del BOFUSS [38]

2.5.2.5. Meter Table

La *Meter Table* es el core del QoS del software switch. Para cada flujo se tienen unas meter asociadas en la propia entrada en la tabla de flujo. Dichas meters, tienen una entrada en la meter table. Cada entrada se compone de una ID, un contador, y unas meter bands. Estas últimas, las meters-bands son las encargadas de llevar a cabo las operaciones de QoS. Cada *Meter Table* debe tener un tipo, un ratio, el cual será el límite que tiene que superarse para aplicar la acción definida por el tipo de la meter. A continuación, se ver la figura 2.14

que ilustra la arquitectura de una Meter table.

Meter id: 1	Counters	Meter Band			
		Type: DSCP Remark	Rate: 100 kbps	Precedence Level: 1	Counters
Meter id: 2	Counters	Meter Band			
		Type: Drop	Rate: 100 kbps	Counters	

Figura 2.14: Estructura de las *Meter tables* del BOFUSS [38]

Entre las responsabilidades de las meter tables podemos encontrarnos las siguientes:

- Creación, destrucción y modificación de las entradas de las meters
- Medir el ratio de aquellos paquetes que han matchado en una flow entry, y apuntan a una meter table.
- Mantener actualizado los contadores de las estadísticas de los paquetes procesados por cada entrada en la meter table.

2.5.2.6. oflib

Los mensajes de OpenFlow están definidos de una manera en particular para ser transmitidos por la red. Los mensajes tienen que estar en modo 8-byte alineados, por lo que habrá alguna ocasión donde se tenga que añadir padding para que se cumpla esta regla. Otro requisito es que el mensaje tiene que estar en Network byte order, es decir, Big Endian. Los mensajes OpenFlow que se mandan por la red tienen que estar en el formato indicado anteriormente, es decir, el byte de mayor peso de una palabra tiene que estar almacenado en la posición más pequeña (la dirección más baja).

Las arquitecturas de cada máquina pueden variar, y el formato de datos con en el que trabajan también. Por ejemplo para ARM e Intel el formato con el cual trabajan ambas arquitecturas es Little Endian byte order. Por ello, en aras de manejar y codificar mensajes openflow se requiere una conversión big-endian a little-endian. Debido a cuál, se necesita una capa de abstracción de la arquitectura donde se vaya a correr dicho software switch. Por ello, aunque el estándar de Openflow no lo indique, se ha añadido esta librería denominada como **oflib**. La función principal de esta librería es las operaciones de marshaling y unmarshaling de los mensajes OpenFlow. Para que la transmisión de información de mensajes Openflow a la red se lleve a cabo de forma completamente autónoma. Las responsabilidades de esta librería por tanto son las siguientes.

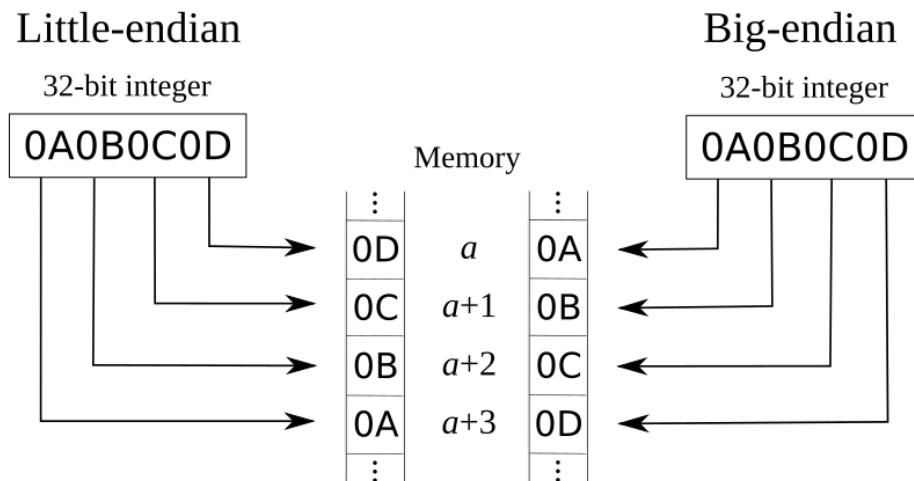


Figura 2.15: Proceso de Marshaling y Unmarshaling

- Cada mensaje openflow debe tener una función para empaquetarlo y desempaquetarlo. De aquí en adelante, y en el repositorio, nos referimos a hacer un *pack* es coger una estructura de datos openflow y prepararla para ser transmitida por la red. Cuando realizamos una operación de *unpack*, cogemos información que viene por la red, y la convertimos a estructuras que entienda la arquitectura sobre la cual está corriendo nuestro software switch.
- Otra responsabilidad que tiene la librería es señalar con errores o warnings en caso de los mensajes Openflow estén mal codificados.

2.5.2.7. Communication Channel

El software switch se comunica con el controlador SDN a través de este agente de control que actúa de proxy entre el datapath y el controlador SDN. Este agente se encarga de gestionar las conexiones con el controlador SDN, aunque, si bien es cierto que este agente no está pactado en el estándar de Openflow, esto da libertad a las diferentes implementaciones para configurar la conexión hacia al controlador como quieran. Por ejemplo, si se quiere que la conexión sea segura extremo a extremo, se tendría que utilizar TLS encima de TCP. Esta libertad de diseño en el canal de comunicación con el controlador, habrá un *gap* que se puede utilizar para desarrollar implementaciones dispares que ofrezcan distintas bondades y funcionalidades. Entre las responsabilidades de esta capa podemos mencionar las siguientes.

- El agente de control se tiene que encargar de abrir una conexión TCP entre el switch y el controlador.
- El establecimiento de la conexión es responsabilidad del agente de control. Después del

inicio de la conexión, el switch negocia la versión Openflow a utilizar entre el switch y el controlador. Este proceso se conoce como handshake.

- El agente de control debe soportar más de un controlador.
- Además, el agente de control debe soportar más de una conexión con el mismo controlador.

2.6. Linux Networking

Esta sección recopilará todos los conceptos y herramientas relacionadas con la parte de Networking en Linux, que son fundamentales para el desarrollo, análisis y validación de este proyecto.

2.6.1. Interfaz virtual - tun/tap

En el mundo de las redes siempre se habla de las interfaces tun/tap de forma indistinta cuando van a utilizarse, sin embargo, cada una tiene su cometido. Como se ha indicado, en networking, las interfaces TUN/TAP son interfaces virtuales que se crean y se gestionan en espacio de kernel. Mencionar que como estas interfaces son virtuales y se gestionan directamente vía software, no como las interfaces reales que se gestionan con unos drivers diferentes, cada interfaz con su driver específico de la interfaz. Los drivers de las interfaces TUN/TAP se crearon en los 2000 como una unión de los avances de los drivers desarrollados en las comunidades de Solaris, Linux, BSD. Actualmente los drivers solo tienen mantenimiento por los kernels de linux y FreeBSD. Ambos tipos de interfaces se utilizan para tunelado, pero no pueden ser utilizadas a la vez dado que trabajan en niveles distintos. Las TUN, de *network TUNnel*, emula la capa de red y puede llegar hacer reenvío de los paquetes. En cambio las interfaces TAP, trabajan en capa 2 solo en capa dos, y emulan un equipo que trabaja en dicha capa, como por ejemplo un switch [39]. Por tanto, hay que dejar claro lo que se puede llegar a realizar con cada interfaz (Ver figura 2.16).

- TUN se puede llegar a utilizar para routing.
- TAP se puede llegar a utilizar para crear un bridge.

Generalmente, cuando los paquetes son enviados por el sistema operativo a través de una interfaz TUN/TAP, serán recibidos por algún programa de espacio de usuario, el cual, está enganchado directamente en la interfaz. Cualquier programa de espacio de usuario podrá pasar paquetes por las interfaces, y las interfaces virtuales se lo pasarán al *stack* de red por defecto, emulando la recepción de los paquetes inyectados desde espacio de usuario.

TUN and TAP in the network stack

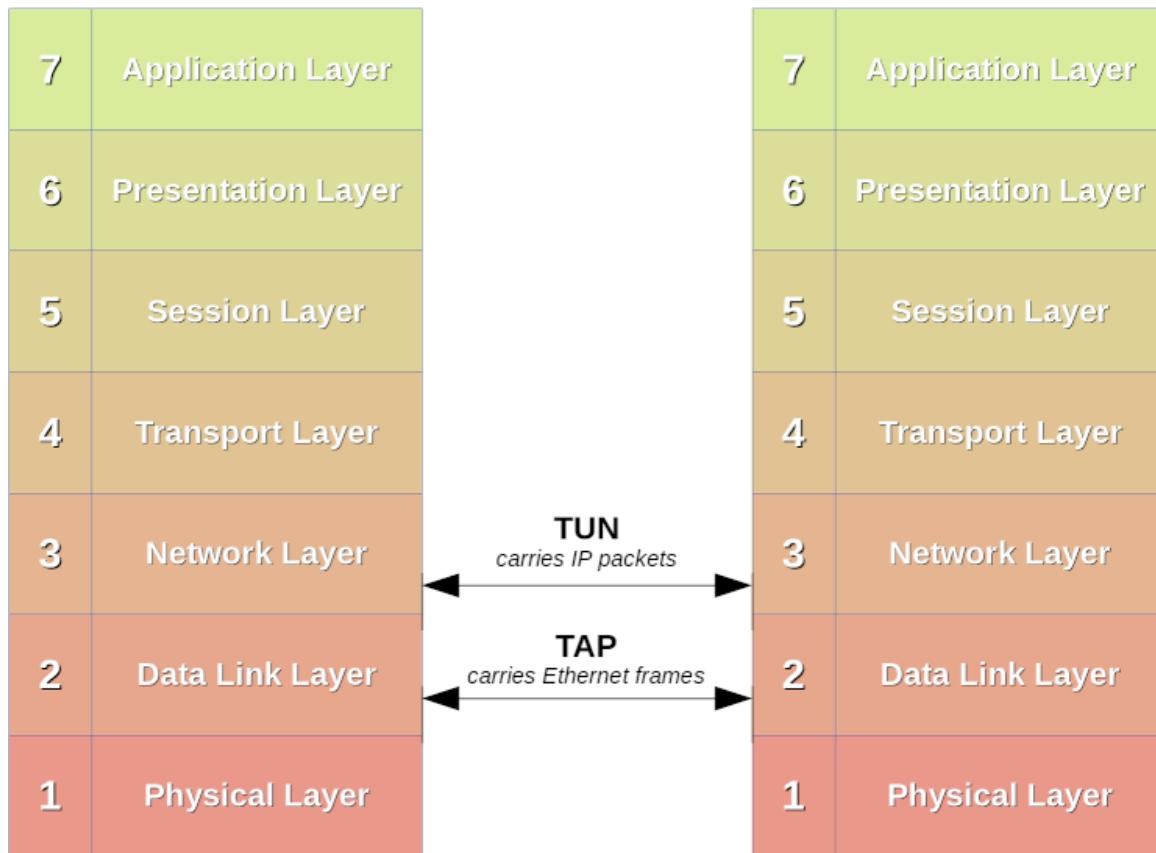


Figura 2.16: Diagrama de funcionamiento de las interfaces virtuales TUN/TAP [40]

Para la creación de estas interfaces lo podemos hacer por ioctl o podemos hacerlo más fácil a través del binario tunctl o en su defecto con el comando de tunctl del set de herramientas iproute2. A continuación, en el bloque 2.1 se indican todos los comandos necesarios para trabajar con las interfaces TUN/TAP.

Código 2.1: Manejo de interfaces TUN - TAP

```

1 # En caso de querer utilizar tunctl hay que instalar el binario
2 sudo apt install -y uml-utilities
3
4 # Para crear una interfaz podemos hacer lo siguiente
5 tunctl -t {nombre_tun}
6
7 # Para eliminarla
8 tunctl -d {nombre_tun}
9
10 # Para crear interfaces de tipo TAP hay que hacer lo siguiente

```

```

11  tunctl -p -t {nombre_tun}
12
13  # El comando análogo con iproute2 sería el siguiente
14  ip tuntap add dev {nombre_tun} mode {tun|tap}

```

En caso de que queramos comprobar que las interfaces se han creado correctamente siempre se puede hacer uso de la herramienta `ethtool`. A continuación, en la figura 2.17, se puede ver como en el campo `bus-info` nos indica que tipo de interfaz es.

```

arppath@arppath-david:~$ sudo ip tuntap add dev tap9 mode tap
arppath@arppath-david:~$ ethtool -i tap9
driver: tun
version: 1.0
firmware-version:
expansion-rom-version:
bus-info: tap
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
arppath@arppath-david:~$ sudo ip tuntap add dev tun9 mode tun
arppath@arppath-david:~$ ethtool -i tun9
driver: tun
version: 1.0
firmware-version:
expansion-rom-version:
bus-info: tun
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
arppath@arppath-david:~$ 

```

Figura 2.17: Comprobación con `ethtool` de tipo de interfaz virtual.

2.6.2. Interfaz virtual - veth

Las Veth son interfaces de Ethernet virtuales creadas como un par de interfaces interconectadas entre sí. El modelo funcional es sencillo: los paquetes enviados desde una interfaz son recibidos por la otra de forma directa, similar al funcionamiento de las *pipes* (mecanismo de comunicación interprocesos en linux). Una característica interesante de estas interfaces es que su gestión está asociada, lo que significa que si se habilita una extremo de la Veth, el otro extremo también se habilitará, y si se deshabilita o se elimina un extremo de un par de Veth, el otro extremo también se verá afectado [41].

Es muy común utilizar las Veth para interconectar *Network Namespaces*. Al saber que estas interfaces estarán conectadas de forma directa, se puede utilizar este enlace como una pasarela entre dos *Network Namespaces*. De esta manera, se pueden interconectar dos *stacks* independientes de red. Más adelante se verá que alguno de los emuladores más utilizados para comprobar despliegues de red SDN hace uso de estas interfaces. Y podemos ir más allá, no sé quiere tampoco entrar en el mundo de los contenedores dado que el proyecto no va a utilizarlos, pero simplemente indicar que estas interfaces también son utilizadas para la

interconexión y contenedores de Docker [42]. La creación y eliminación de este tipo de interfaces se puede observar en el bloque de código 2.2. Se recuerda que se necesitan permisos de root para realizar estas operaciones.

Código 2.2: Uso de las interfaces Veths

```

1  # COn este comando se crean un par interfaces veth
2  ip link add {veth1_name} type veth peer name {veth2_name}
3
4  # Solo hace falta indicar uno de los dos nombres del par de Veths
5  ip link delete {veth_name}
```

2.6.3. Herramienta TC

En Linux, Traffic control (TC) es un subsistema del kernel, que tiene una interfaz en espacio de usuario que se utiliza para controlar y gestionar el tráfico de red. Proporciona una serie de herramientas y mecanismos para aplicar políticas de QoS, limitar el ancho de banda, establecer prioridades de tráfico, realizar filtrado y dar forma al tráfico de red [43]. Se ha añadido esta sección a la memoria dado que la mayoría de emuladores de red en Linux suelen hacer uso de esta herramienta para modelar los enlaces de las topologías emuladas. El TC se utiliza principalmente para las siguientes tareas.

- Control de ancho de banda: Nos permite limitar la cantidad de ancho de banda que un flujo de datos específico puede utilizar. Esto es útil para evitar que un flujo de tráfico acapare todo el ancho de banda y afecte el rendimiento de otros flujos.
- Priorización de tráfico: Permite asignar prioridades diferentes a los diferentes flujos de datos. Esto asegura que ciertos flujos de tráfico, como VoIP o similares al ser más sensibles, tengan prioridad sobre otros, como la descarga de archivos donde la latencia y el jitter no son un problema, pero si la integridad.
- Modelado y conformación de tráfico: Permite dar forma al tráfico de red mediante la definición de límites en el ritmo de transmisión. Esto es útil para evitar congestiones en la red y garantizar un flujo de tráfico constante y uniforme.
- Marcado y filtrado de paquetes: Permite clasificar y filtrar paquetes de acuerdo a diferentes criterios, como direcciones IP, puertos, protocolos, etc. Esto permite aplicar políticas específicas a ciertos flujos de datos o bloquear ciertos tipos de tráfico no deseado. Esta funcionalidad es útil, pero está duplicada con otros submódulos del kernel de Linux, como por ejemplo las `ip-tables`.

El procesamiento del tráfico para conseguir llevar a cabo las tareas anteriormente mencionadas, se emplean tres tipos de objetos: **qdiscs**, **classes** y **filters** [43].

2.6.3.1. Qdiscs

El objeto qdiscs, que significa “Quality-Deficit Inverse Congestion Control Scheduler” (Pla-nificador de Control de Congestión Inverso de Calidad-Deficit), es un concepto fundamental en el networking de Linux que determina el orden en el que los miembros de la cola, en este caso los paquetes, son seleccionados para su servicio.

Por ejemplo, cuando una herramienta de espacio de usuario necesita transmitir un paquete en un momento dado, ese paquete se entrega al *stack* de red y, finalmente, llega a la interfaz de red por la cual será transmitido. En ese momento, el paquete se encuentra encolado en una cola, esperando ser transmitido. Estas colas son gestionadas por un qdiscs. El qdiscs por defecto en Linux es un pfifo, que significa “Priority-First-In, First-Out”, señalar el concepto de prioridad, dado que es fundamental. Por tanto lo podemos ver como una cola pura en la que se sigue el orden de llegada (*first-in*) y se atienden los paquetes en ese orden siempre y cuando no haya alguna directriz de prioridad, con una limitación en el tamaño de la cola en términos del número de paquetes que puede contener.

2.6.3.2. Classes

Las clases pueden entenderse como sub-qdiscs dentro de un qdiscs. Una clase puede contener a su vez otras clases, formando sistemas de QoS detallados, como se muestra en la figura 2.18. Cuando los paquetes son recibidos en una cola gestionada por un qdiscs, pueden ser encolados en base a las características del paquete en otras colas administradas por otras clases. Esto permite, por ejemplo, priorizar el envío de datos de una aplicación sobre otra. Para lograrlo, los paquetes de ambas aplicaciones se clasificarán en clases diferentes, asignándole a una clase más prioridad que a la otra. Esto implica asignar más recursos de transmisión y recepción a la clase prioritaria.

En resumen, las clases en el contexto de los qdiscs representan subdivisiones que permiten una gestión más detallada del tráfico de red. Al agrupar paquetes en diferentes clases y asignar prioridades, se puede establecer un control más minucioso sobre la transmisión y recepción de datos, lo que facilita la implementación de políticas de QoS y la asignación de recursos de manera más eficiente.

2.6.3.3. Filters

En el contexto de TC en Linux, los filtros son objetos utilizados para seleccionar y clasificar paquetes de red con el fin de aplicar acciones específicas sobre ellos. Los filtros permiten establecer reglas que determinan cómo se deben tratar los paquetes en función de diferentes criterios, como direcciones IP, puertos, protocolos, etiquetas VLAN, campos de encabezado,

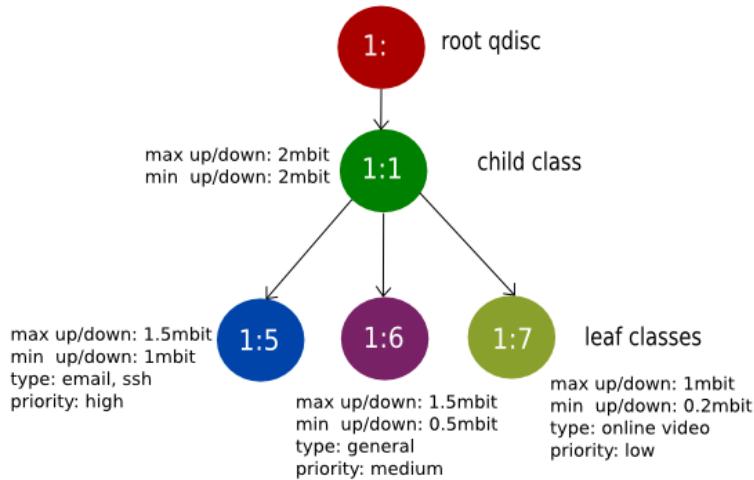


Figura 2.18: Mecanismo de calidad de servicio implementado con clases y sub-clases [44]

entre otros. Los filtros en TC se utilizan en conjunción con los qdiscs para controlar y gestionar el tráfico de red de manera más granular. Se pueden aplicar diferentes acciones a los paquetes que coinciden con los criterios definidos por los filtros, como encollarlos en clases específicas, modificar sus encabezados, descartarlos, redirigirlos a interfaces de red diferentes, entre otras opciones. Existen varios tipos de filtros disponibles en TC, entre ellos podemos ver los siguientes.

- Tipo U32: Permite filtrar paquetes basados en campos de encabezado, como direcciones IP, puertos y protocolos.
- Tipo Basic: Permite filtrar paquetes utilizando criterios básicos como direcciones IP de origen y destino, puertos y protocolos.
- Tipo Route: Permite filtrar paquetes en función de las rutas de red.
- Tipo Berkeley Packet Filter (BPF): Permite utilizar programas BPF para filtrar y procesar paquetes.

Estos son solo algunos ejemplos de los tipos de filtros disponibles en TC. Cada tipo de filtro tiene su propia sintaxis y opciones de configuración específicas. Por tanto, los filtros en TC se utilizan para seleccionar y clasificar paquetes de red en función de diferentes criterios. Estos filtros permiten aplicar acciones específicas a los paquetes que coinciden con los criterios definidos, lo que facilita la gestión y el control del tráfico de red en Linux.

2.6.4. Namespaces

Una *Namespace* se utiliza para encapsular un recurso del sistema operativo en una abstracción que engaña a los procesos dentro de esa *Namespace* haciéndoles creer que tienen su propia instancia aislada del recurso en cuestión, separada del sistema real. Los cambios realizados en los recursos aislados solo son visibles para los procesos que pertenecen a esa *Namespace*, mientras que son invisibles para otros procesos que pertenecen al sistema o a otra *Namespace*.

Existen varios tipos de *Namespaces*, los cuales se detallan en la tabla 2.1. Cada tipo de *Namespace* tiene diversos usos y aplicaciones, pero uno de los más relevantes es el de los contenedores. Los contenedores proporcionan un entorno aislado para ejecutar aplicaciones o herramientas. Actualmente, los contenedores más populares son los de Docker, una plataforma ampliamente utilizada en el mundo del desarrollo de software. Docker aprovecha las bondades del kernel de Linux para aislar recursos y crear instancias aisladas del sistema Host sobre el cual se ejecutan. Utiliza la API proporcionada por el kernel para crear y destruir *Namespaces* según sea necesario. En esencia, Docker es un conjunto de llamadas al sistema que se encarga de gestionar *Namespaces*. Además de la gestión de *Namespaces*, Docker proporciona otras características útiles, como el mecanismo de *copy-on-write* y una configuración de red en modo *bridge* hacia el exterior. Sin embargo, en su núcleo, Docker es simplemente un *wrapper* que facilita la gestión de *Namespaces* con el objetivo de implementar contenedores.

Tipo de Namespace	Descripción
Cgroup	Namespace utilizado generalmente para establecer límites de recursos, como CPU, memoria, lecturas y escrituras a disco, para todos los procesos dentro de la misma Namespace.
Time	Namespace utilizado para establecer una hora del sistema diferente a la del sistema global.
Network	Namespace utilizado para crear una réplica aislada del stack de red del sistema dentro del propio sistema.
User	Namespace utilizado para aislar un grupo de usuarios.
PID	Namespace utilizado para tener identificadores de proceso independientes de otras namespaces.
IPC	Namespace utilizado para aislar los mecanismos de comunicación entre procesos.
Uts	Namespace utilizado para establecer un nombre de host y nombre de dominio diferentes de los establecidos en el sistema.
Mount	Namespace utilizado para aislar los puntos de montaje en el sistema de archivos.

Tabla 2.1: Resumen de los tipos de Namespaces en el Kernel de Linux

2.6.4.1. Persistencia de las Namespaces

Según la página man-page sobre las *Namespaces* [45], es importante tener en cuenta que las *Namespaces* tienen una vida finita. La duración de una *Namespace* dependerá de si está siendo referenciada, lo que significa que vivirá mientras siga siendo referenciada y será destruida cuando deje de serlo. Comprender este concepto de vida finita es crucial para tener una mejor comprensión del funcionamiento interno de Mininet o Mininet-WiFi, ya que estas herramientas aprovechan estos conceptos para optimizar las operaciones y mejorar el

rendimiento de sus respectivos emuladores.

Actualmente, una *Namespace* puede ser referenciada de tres maneras diferentes:

- Siempre que haya un proceso en ejecución dentro de dicha *Namespace*.
- Siempre que haya abierto un descriptor de archivo para el archivo identificador de la *Namespace* (por ejemplo, `/proc/pid/ns/tipo_namespace`).
- Siempre que exista un *bind-mount* del archivo (`/proc/pid/ns/tipo_namespace`) de la *Namespace* en cuestión.

Si ninguna de estas condiciones se cumple, el kernel eliminará automáticamente la *Namespace*. En el caso de una *Network Namespace*, las interfaces que se encuentren dentro de la *Namespace* que está siendo eliminada volverán a la *Network Namespace* predeterminada [45]. Es fundamental tener en cuenta estos detalles, ya que ayudan a comprender cómo se manejan las *Namespaces* y cómo se gestionan los recursos en el contexto de Mininet o Mininet-WiFi.

2.6.4.2. Concepto de las Network Namespaces

Una vez comprendido el concepto de *Namespace* en Linux, es necesario introducir las *Network Namespace*, las cuales desempeñarán un papel fundamental en las plataformas utilizadas para emular. Estas *Network Namespace* consisten en réplicas lógicas del *stack* de red predeterminado de Linux, que incluye rutas, tablas ARP, Iptables e interfaces de red [46].

Linux se inicia con una *Network Namespace* predeterminada, conocida como el espacio *root*, que tiene su propia tabla de rutas, tabla ARP, Iptables e interfaces de red. Sin embargo, también es posible crear *Network Namespace* adicionales que no son predeterminadas, crear nuevos dispositivos dentro de esos espacios de nombres o trasladar dispositivos existentes de un espacio de nombres a otro.

La herramienta más sencilla para llevar a cabo todas estas tareas de gestión con las Netns es `iproute2` (consulte el Anexo C.2). Esta herramienta, utilizando el módulo `netns`, permite gestionar todo lo relacionado con las *Network Namespace* con nombre (*named network namespaces*). La característica “con nombre” significa que todas las *Network Namespace* administradas mediante `iproute2` serán persistentes, ya que se realizará un *bind-mount* del archivo identificador de la *Namespace* en cuestión con su nombre correspondiente en el directorio `/var/run/netns`. A continuación, en el bloque 2.3, se enumeran los comandos más comunes utilizados para gestionar las *Network Namespace*. Se asume que estos comandos se ejecutan con privilegios de root.

Código 2.3: Casos de uso de las Netns

```

1 # Con el siguiente comando creamos una network namespace
2 ip netns add {netns_name}
3
4 # Listamos las Named Network Namespaces :)
5 ip netns list
6
7 # Movemos una interfaz a una Network Namespace determinada
8 ip netns set {netns_name} Veth
9
10 # Ejecutamos un comando dentro de una Network Namespace
11 ip netns exec {netns_name} {cmd}
12
13 # De esta forma, eliminamos una Network Namespace
14 ip netns del {netns_name}

```

2.6.4.3. Comunicación inter-Namespace: Veth

Por lo tanto, se puede representar el siguiente diagrama básico para ilustrar el funcionamiento de un par de interfaces Veth asignadas a diferentes *Network Namespaces*. Como se muestra en la Figura 2.19, ambas interfaces están directamente conectadas internamente en el Kernel. Si se generan paquetes desde una *Network Namespace* hacia la otra, estos paquetes viajarán desde un extremo de la Veth directamente al otro extremo de la Veth a través del Kernel, sin ser percibidos por la *Network Namespace* predeterminada. Esta configuración resulta muy útil para establecer enlaces entre nodos de red independientes. Los nodos se replicarán utilizando *Network Namespaces* y los enlaces se establecerán utilizando pares de interfaces Veth. Estos mecanismos son ampliamente utilizados por herramientas de emulación de redes como Mininet o Mininet-WiFi, lo cual se explicará en detalle más adelante.

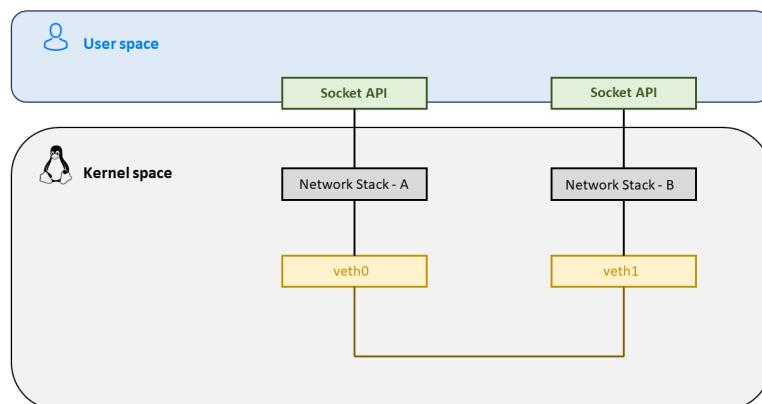


Figura 2.19: Enlace entre interfaces Veth separadas en dos Network Namespaces [17]

2.7. Mininet y Mininet-WiFi

En esta sección se abordará el marco teórico relacionado con las principales herramientas de emulación utilizadas en este proyecto. Se prestará especial atención a Mininet, que es la herramienta principal y la base sobre la cual han surgido otras herramientas de emulación, como Mininet-WiFi y Mininet-IoT.

2.7.1. Mininet

Mininet es una herramienta utilizada para emular redes, principalmente del tipo SDN (Software-Defined Networking). Permite emular hosts, routers, switches y enlaces en una sola máquina a un coste reducido, siempre y cuando se cuente con el Kernel de Linux en dicha máquina. Para lograr esto, Mininet utiliza una forma de virtualización "ligera" que aprovecha las capacidades del Kernel de Linux para virtualizar recursos, como las *Namespaces* (consultar 2.6.4) [47].

La cantidad de recursos virtualizados dependerá de las características de cada nodo, y esto también afectará el rendimiento de la emulación. Por ejemplo, los nodos tipo **Host** en Mininet requieren el uso de una *Network Namespace*, lo que les proporciona su propio *stack* de red y los hace completamente independientes⁹ del sistema y otros nodos en la red emulada. Sin embargo, por defecto, todos los nodos **Host** comparten el sistema de archivos, la numeración de procesos (PIDs), los usuarios, etc. En términos técnicos, no están completamente aislados como un host real. Esto se debe a que Mininet virtualiza solo los recursos necesarios para llevar a cabo la emulación, lo que mejora el rendimiento y permite que máquinas con recursos limitados puedan realizar la emulación [47].

En cuanto a la creación de topologías en Mininet, existen dos enfoques. El primero es utilizar la API escrita en Python para interactuar con las clases de Mininet. Con esta API, se puede construir la topología importando los módulos y clases necesarios para definirla en un script de Python. El segundo enfoque es utilizar la herramienta llamada **MiniEdit**, que proporciona una interfaz gráfica (GUI) donde los usuarios pueden crear la topología arrastrando y soltando nodos de red. Desde la misma GUI, se puede exportar la topología generada a un archivo (*.mn) para recuperarla más tarde o a un script en Python (*.py) para cargarla en el intérprete de Python cuando sea necesario. Esta herramienta es especialmente útil para aquellos que no tienen conocimientos de programación en Python pero desean utilizar el emulador, lo que representa una ventaja significativa [47].

⁹En lo que respecta a Networking

Por lo tanto, se puede concluir que los aspectos más fuertes de Mininet son los siguientes puntos:

- Mininet es rápido gracias a su diseño basado en *Namespaces*, lo que permite una gestión eficiente de los recursos. En la sección 2.6.4, se explica cómo se lleva a cabo esta gestión.
- Mininet no consume recursos en exceso, ya que virtualiza únicamente los componentes necesarios para la emulación. Además, se pueden establecer límites máximos de recursos para la emulación en caso de ser necesario.
- Mininet brinda libertad al usuario para crear topologías y escenarios personalizados utilizando la API en Python de Mininet. Estos escenarios pueden ser fácilmente transferidos a otra máquina, ya que solo se requiere compartir el script que describe la topología. Es importante tener en cuenta que los resultados de las pruebas pueden variar entre diferentes máquinas, ya que Mininet emula la red en lugar de simularla. Por lo tanto, los resultados dependerán de las condiciones de la máquina donde se ejecuten las pruebas.

Aunque Mininet ofrece muchas ventajas, también tiene una limitación importante que debe tenerse en cuenta. Como se mencionó anteriormente, Mininet utiliza una forma de virtualización “ligera” basada en las *Namespaces* del Kernel de Linux. Si bien esta decisión de diseño proporciona beneficios significativos en términos de rendimiento al aprovechar el propio sistema operativo para virtualizar recursos, surge un problema cuando se intenta exportar el emulador a otra plataforma con un sistema operativo diferente. Es posible que este sistema operativo no admita un equivalente funcional a las *Namespaces* de Linux o, incluso si lo hace, su API para utilizarlas puede ser completamente diferente. Esto puede dificultar o incluso impedir la ejecución de Mininet en plataformas que no porten el kernel de Linux.

2.7.2. Funcionamiento de Mininet

Anteriormente se mencionó que Mininet utiliza *Network Namespaces* como método para virtualizar *stacks* de red independientes y así emular redes con un costo mínimo. En la figura 2.20, se muestra la arquitectura interna de Mininet para una topología compuesta por dos nodos **Host** y un software switch conectado por TCP a un controlador remoto.

Como se puede observar, cada nodo **Host** está aislado en su propia *Namepace* de red, mientras que el switch se ejecuta en la *Namepace* por defecto (root). La comunicación

entre los nodos de esta topología se realiza mediante Veths (consultar 2.6.4.3), las cuales permiten emular los enlaces entre los diferentes nodos de la red.

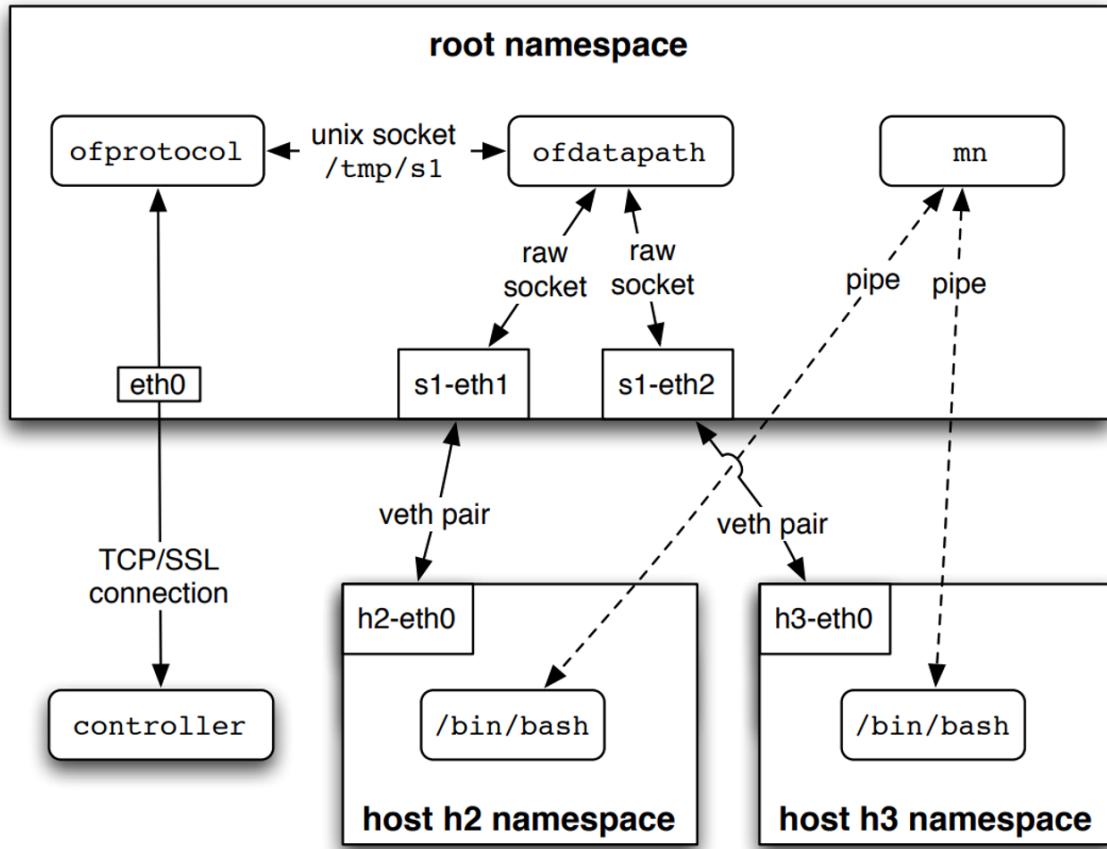


Figura 2.20: Arquitectura de Mininet [48]

Una vez presentada toda la teoría sobre Mininet, puede surgir la pregunta de cómo se puede comprobar si realmente utiliza *Network Namespaces*. Para hacerlo, lo primero que se debe hacer es crear el escenario para que Mininet pueda crear las *Network Namespaces* necesarias. En este caso, se utilizará la topología mostrada en la figura 2.20. Para crear esta topología, solo se necesita tener Mininet instalado y seguir los pasos que se indican en el bloque 2.4. Una vez levantado el escenario, se debería obtener el output indicado en la figura 2.21.

Código 2.4: Ejecución de Mininet con la topología por defecto

```

1 # Lanzamos Mininet con la topo por defecto :)
2 sudo mn

```

```
n0obie@n0obie-VirtualBox:~$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet-wifi> dump
<Host h1: h1-eth0:10.0.0.1 pid=9518>
<Host h2: h2-eth0:10.0.0.2 pid=9520>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=9525>
<OVSCController c0: 127.0.0.1:6653 pid=9511>
mininet-wifi>
```

Figura 2.21: Salida por pantalla de la ejecución de la topología por defecto

Ahora que hemos creado el escenario, podemos verificar si hay *Network Namespaces* en el sistema utilizando el conjunto de herramientas iproute2 (consultar C.2). El comando más utilizado para listar las *Network Namespaces* utilizando el módulo **netns** se muestra en el bloque 2.5.

Código 2.5: Listar Named Network Namespaces

```
1 # Listamos las network namespaces con Nombre! ojito :)
2 sudo ip netns list
```

```
n0obie@n0obie-VirtualBox:~$ sudo ip netns list
n0obie@n0obie-VirtualBox:~$
```

Figura 2.22: Listado de Named Network Namespaces existentes en el sistema

Al observar la figura 2.22, no parece haber ninguna *Network Namespace* en el sistema. Entonces, ¿dónde está el problema? La razón por la que el comando **ip netns list** no muestra información es que Mininet no está creando el enlace simbólico (*softlink*) necesario para que la herramienta pueda listar las *Network Namespaces*. Según la documentación del comando, este lee desde la ruta **/var/run/netns/**, donde se encuentran todas las *Network Namespaces* con nombre. Estas Netns son aquellas en las que se ha realizado un *bindmount* con su nombre en ese directorio para que persistan incluso si no hay ningún proceso en ejecución en ellas. Como se mencionó anteriormente, las *Namespaces* tienen una vida finita y solo existen mientras estén referenciadas (consulte la tabla 2.1). Por lo tanto, si no se cumple ninguna condición de referencia, la *Namespace* en cuestión se elimina.

Mininet se encarga de recrear la red emulada y, cuando el usuario finaliza la emulación, la red emulada debe desaparecer. Este proceso debe ser lo más rápido y eficiente posible para brindar una mejor experiencia al usuario. La naturaleza del diseño de Mininet sugiere que la creación y destrucción de las *Network Namespaces* están asociadas con la primera condición de referencia de una *Namespace*.

En otras palabras, no tendría sentido realizar enlaces o enlaces simbólicos que luego se deban eliminar, ya que esto implicaría una carga de trabajo significativa para emulaciones de redes grandes y aumentaría el tiempo necesario para limpiar el sistema una vez finalizada la emulación. Además, hay que tener en cuenta que existe una condición que se adapta bien a las necesidades de Mininet: solo se requiere un proceso en ejecución por cada *Network Namespace*, y al realizar la limpieza, solo se deben finalizar los procesos que mantienen las *Network Namespaces*. Cuando no haya más procesos en ejecución en una *Namespace*, el kernel se encargará de eliminarla.

De acuerdo con el razonamiento expuesto, se deberían ver varios procesos que se crean al iniciar el escenario en Mininet. Cada uno de estos procesos deberá tener un archivo de *Network Namespace* (`/proc/pid/ns/net`) con un número de nodo único (*inode*) para los procesos que se ejecutan en diferentes *Network Namespaces*.

```
n0obie@n0obie-VirtualBox:~$ sudo ps aux | grep mininet
n0obie    2868  0.0  0.0  21312   944 pts/13   S+  13:42   0:00 grep --color=auto mininet
root     9511  0.0  0.0  28140  3436 pts/9    Ss+ 12:29   0:00 bash --norc --noediting -is mininet:c0
root     9518  0.0  0.0  28136  3572 pts/10   Ss+ 12:29   0:00 bash --norc --noediting -is mininet:h1
root     9520  0.0  0.0  28136  3364 pts/11   Ss+ 12:29   0:00 bash --norc --noediting -is mininet:h2
root     9525  0.0  0.0  28136  3416 pts/12   Ss+ 12:29   0:00 bash --norc --noediting -is mininet:s1
n0obie@n0obie-VirtualBox:~$ █
```

Figura 2.23: Listado de procesos con referencias a Mininet

Si examinamos el archivo `/proc/pid/ns/net` para cada proceso mencionado en la figura 2.23, podremos determinar cuáles de ellos se encuentran en una *Network Namespace* distinta, según el valor del *inode*. Por ejemplo, verifiquemos los procesos asociados a los Host1 y Host2.

```
n0obie@n0obie-VirtualBox:~$ sudo ps aux | grep mininet | grep h1
root     9518  0.0  0.0  28136  3572 pts/10   Ss+ 12:29   0:00 bash --norc --noediting -is mininet:h1
n0obie@n0obie-VirtualBox:~$ sudo ls -la /proc/9518/ns/net | grep -e 'net:[\[\[0-9]\+\]]'
lrwxrwxrwx 1 root root 0 jun 13 13:47 /proc/9518/ns/net -> net:[4026532227]
n0obie@n0obie-VirtualBox:~$ █
```

Figura 2.24: Información de contexto sobre el proceso del Host1

```
n0obie@n0obie-VirtualBox:~$ sudo ps aux | grep mininet | grep h2
root      9520  0.0  0.0  28136  3364 pts/11    Sst+  12:29   0:00 bash --norc --noediting -is mininet:h2
n0obie@n0obie-VirtualBox:~$ sudo ls -la /proc/9520/ns/net | grep -e 'net:[\[(0-9)\+\]\]' 
lrxwxrwxrwx 1 root root 0 jun 13 13:49 /proc/9520/ns/net -> net:[4026532343]
n0obie@n0obie-VirtualBox:~$ █
```

Figura 2.25: Información de contexto sobre el proceso del Host2

Como se puede observar, hay diferentes *inodes*, archivos distintos y *Network namespaces* diferentes. Esta prueba demuestra cómo Mininet utiliza procesos de bash para mantener las *Network Namespaces* de los nodos que lo requieren.

2.7.3. Mininet-WiFi

Mininet-WiFi [49] es un emulador de redes inalámbricas diseñado principalmente para trabajar bajo el estándar ieee80211. Esta herramienta nació de Mininet, es decir, es un *fork* de la misma. Por ello, comparten todas las bases sobre virtualización “ligera” haciendo uso de *Namespaces* y Veths, por tanto, todos los scripts de Mininet son compatibles en Mininet-WiFi.

Esto es así ya que toda la funcionalidad wireless es un añadido sobre la base que desarrollaron para Mininet. Los desarrolladores de Mininet-WiFi se valieron del subsistema wireless del Kernel de Linux y del módulo mac80211_hwsim, para conseguir emular las interfaces y el supuesto medio inalámbrico. Para más información sobre esta herramienta se recomienda ir al **punto ??**, donde se hace un análisis profundo sobre las jerarquías de clases añadidas en Mininet-WiFi, como opera internamente y como se comunica con el módulo en el kernel para generar los escenarios inalámbricos.

2.7.4. Mininet-IoT

La herramienta Mininet-IoT [50] es un emulador de redes de baja capacidad diseñado para trabajar en conjunto bajo el estándar ieee802154 y la capa de adaptación 6LoWPAN. Esta herramienta nació de Mininet-WiFi, que a su vez nació de Mininet, por lo que en la práctica, Mininet-IoT comparte todas las técnicas de virtualización “ligera” de Mininet. Al heredar de Mininet-WiFi y Mininet, todos los scripts para desplegar topologías alámbricas y WiFi son compatibles en Mininet-IoT.

La gran diferencia entre Mininet-IoT y Mininet-WiFi, radica en el módulo que emplean para conseguir emular las interfaces y el supuesto medio inalámbrico. Mininet-WiFi hace uso del módulo mac80211_hwsim, mientras que Mininet-IoT hace uso del módulo del Kernel mac802154_hwsim (es necesario tener una versión del Kernel superior a la 4.18.x para obtener dicho modulo). Toda la gestión de nodos, interfaces y enlaces es exactamente la

misma a la de Mininet-WiFi. Por ello, Ramon Fontes (principal desarrollador de la herramienta), creó una clase agnóstica para gestionar módulos del Kernel en Mininet-WiFi, y migró todo el proyecto de Mininet-IoT a Mininet-WiFi. De esta forma, el mantenimiento del *core* que compartían ambas herramientas se hacía únicamente en un proyecto, y daba la posibilidad al usuario de Mininet-WiFi de establecer enlaces de baja capacidad en sus topologías inalámbricas.

2.8. Contiki-ng

Contiki es un sistema operativo diseñado específicamente para dispositivos de baja capacidad, como sensores. Fue desarrollado por Adam Dunkels en colaboración con Bjorn Gronvall y Thiemo Voigt en 2002. A lo largo de los años, el proyecto Contiki ha crecido enormemente y ha involucrado a numerosas empresas y cientos de colaboradores en su repositorio de GitHub¹⁰. El objetivo principal de Contiki era proporcionar a los nodos de las redes de sensores inalámbricos (WSN) un sistema operativo liviano capaz de cargar y descargar servicios de forma dinámica [51].

El kernel de Contiki se basa en un modelo orientado a eventos y admite multitarea con prelación. Está escrito en lenguaje C y ha sido portado a diversas arquitecturas de microcontroladores, como el MSP430 de Texas Instruments y sus variantes.

En un sistema que ejecuta Contiki, se divide en dos partes claramente diferenciadas del firmware, como se muestra en la Figura 2.26: el núcleo (*core*) y los programas o servicios cargados. Esta partición se realiza durante la etapa de compilación y es independiente del objetivo (target) donde se desplegará el sistema. El núcleo (*core*) incluye el propio kernel, un conjunto de servicios base (como temporizadores y controladores), bibliotecas, controladores y el stack de comunicación. Los programas o servicios cargados se mapean en memoria mediante el cargador del kernel durante el tiempo de ejecución.

En los últimos años ha surgido un nuevo proyecto llamado **Contiki-ng**¹¹, el cual es un fork del sistema operativo Contiki. Este proyecto ha ganado popularidad y ha superado a su predecesor con el eslogan “Contiki-NG: El sistema operativo para dispositivos IoT de próxima generación”. Actualmente, toda la comunidad de Contiki se está centrando en este nuevo proyecto, el cual proporciona un stack de comunicación más compatible con las especificaciones RFC, ofrece soporte para protocolos como IPv6/6LoWPAN y 6TiSCH, y se está volviendo cada vez más popular por su capacidad de brindar soporte a microcontroladores

¹⁰<https://github.com/contiki-os/contiki>

¹¹<https://github.com/contiki-ng/contiki-ng>

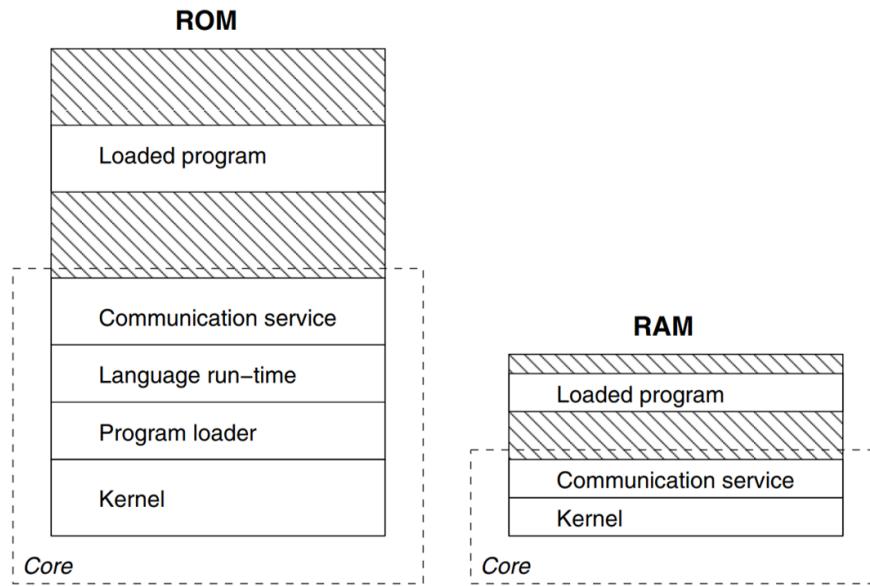


Figura 2.26: Gestión de la memoria en un sistema con Contiki OS [51]

con arquitectura ARM [52].

2.8.1. Simulador Cooja

El flujo de trabajo con Contiki o Contiki-ng varía dependiendo de si se trabaja con hardware real o si se realiza una simulación de los programas desarrollados. En el caso de trabajar con hardware real, el proceso consiste en compilar el sistema operativo y los programas específicos utilizando el objetivo (target) correspondiente, lo que generará un archivo binario que se puede cargar en la memoria del hardware.

Por otro lado, si se opta por la simulación, se utilizará el simulador llamado Cooja¹². Cooja es un simulador escrito en Java que permite **simular** una serie de nodos IoT. Al simular, es posible observar el comportamiento del programa desarrollado en diferentes plataformas. El proceso de compilación del núcleo (core) de Contiki y los programas desarrollados está integrado en el propio simulador, lo que facilita al usuario la compilación de sus programas para diferentes tipos de nodos. Cada simulación se puede guardar en un archivo con extensión ***.csc**, que almacena todos los datos de la simulación, como la semilla (*seed*), las posiciones y los tipos de nodos, utilizando una estructura XML.

De esta manera, tanto si se trabaja con hardware real como si se realiza una simulación, Contiki y Contiki-ng ofrecen herramientas y entornos integrados que permiten desarrollar y

¹²<https://github.com/contiki-ng/cooja/tree/master>

probar programas para sistemas embebidos y dispositivos IoT. A continuación, en la figura 2.27, se indica la interfaz gráfica del simulador.

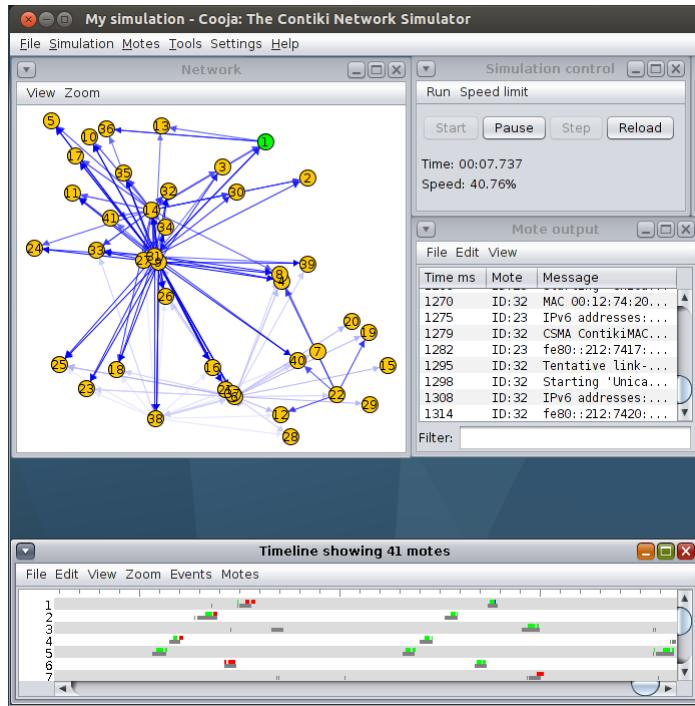


Figura 2.27: Interfaz gráfica del simulador Cooja [53]

2.9. Contribuciones en GitHub

La herramienta GitHub [54] es una plataforma para alojar repositorios de forma remota. En este Trabajo Final de Grado (TFG), se hará uso de la herramienta de control de versiones Git¹³, y de GitHub como plataforma para alojar el código. Pero no se hará un uso exclusivo de la plataforma para almacenar el código desarrollado en el TFG, sino que se aprovechará el carácter público del repositorio para ofrecer documentación y ejemplos a todos los usuarios interesados que lo visiten.

- Enlace al repositorio del TFG: <https://github.com/davidcawork/TFG>

Todo el proceso de documentación en el repositorio pasa por los ficheros README, los cuales se podrán encontrarán en todos los directorios del repositorio. Estos ficheros suministrarán

¹³<https://git-scm.com/>

la información necesaria a los visitantes para poder replicar las pruebas realizadas, y hacer uso del *software* desarrollado. Pero además, se han añadido las explicaciones y los análisis teóricos necesarios, para que el visitante realmente entienda la naturaleza de las pruebas, que se espera de ellas y que conclusiones se podrán sacar de dichos test.

La finalidad del repositorio por tanto es doble, ya que servirá para almacenar el código, pero también ayudará a divulgar los contenidos de este proyecto. De forma adicional, todos los desarrollos útiles y que pueden aportar en otros repositorios se han ofrecido en forma de contribución vía *pull-request*. A continuación, se en la tabla 2.2 se indican todas las contribuciones realizadas.

Contribución	Enlace al Pull-Request
Nuevo método de instalación de todas las dependencias del entorno P4	https://github.com/p4lang/tutorials/pull/261
Corregir documentación del repositorio XDP Tutorial	https://github.com/xdp-project/xdp-tutorial/pull/95
Integración del BMV2 en Mininet-Wifi	https://github.com/intrig-unicamp/mininet-wifi/pull/302
Arreglar interfaz gráfica cuando los APs tienen movilidad	https://github.com/intrig-unicamp/mininet-wifi/pull/229
Dar soporte de las estaciones Wifi en el comando pingallfull	https://github.com/intrig-unicamp/mininet-wifi/pull/230

Tabla 2.2: Resumen de contribuciones realizadas

Bibliografía

- [1] S. Balaji, K. Nathani, and R. Santhakumar, “IoT Technology, Applications and Challenges: A Contemporary Survey,” *Wireless Personal Communications*, vol. 108, pp. 363–388, 9 2019. [Online]. Available: <https://link.springer.com/article/10.1007/s11277-019-06407-w>
- [2] S. Li, L. D. Xu, and S. Zhao, “5G Internet of Things: A survey,” *Journal of Industrial Information Integration*, vol. 10, pp. 1–9, 6 2018.
- [3] IoT-Analytics, “Number of connected IoT devices growing 18% globally.” [Online]. Available: <https://iot-analytics.com/number-connected-iot-devices/>
- [4] D. C. Nguyen, M. Ding, P. N. Pathirana, A. Seneviratne, J. Li, D. Niyato, O. Dobre, and H. V. Poor, “6G Internet of Things: A Comprehensive Survey,” *IEEE Internet of Things Journal*, vol. 9, pp. 359–383, 1 2022.
- [5] “Europe scales up 6G research investments Shaping Europe’s digital future.” [Online]. Available: <https://digital-strategy.ec.europa.eu/en/news/europe-scales-6g-research-investments-and-selects-35-new-projects-worth-eu250-million>
- [6] “The Smart Networks and Services Joint Undertaking | Shaping Europe’s digital future.” [Online]. Available: <https://digital-strategy.ec.europa.eu/en/policies/smart-networks-and-services-joint-undertaking>
- [7] “Europe launches the second phase of its 6G Research and Innovation Programme - Shaping Europe’s digital future.” [Online]. Available: <https://digital-strategy.ec.europa.eu/en/news/europe-launches-second-phase-its-6g-research-and-innovation-programme>
- [8] M. A. Uusitalo, M. Ericson, B. Richerzhagen, E. U. Soykan, P. Rugeland, G. Fettweis, D. Sabella, G. Wikström, M. Boldi, M.-H. Hamon, H. D. Schotten, V. Ziegler, E. C. Strinati, M. Latva-aho, P. Serrano, Y. Zou, G. Carrozzo, J. Martrat, G. Stea, P. Demestichas, A. Pärssinen, and T. Svensson, “Hexa-X The European 6G flagship project,” in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, 2021, pp. 580–585.

- [9] M. Katz, M. Matinmikko-Blue, and M. Latva-Aho, “6Genesis Flagship Program: Building the Bridges Towards 6G-Enabled Wireless Smart Society and Ecosystem,” *Proceedings - 2018 10th IEEE Latin-American Conference on Communications, LATINCOM 2018*, 1 2019.
 - [10] “ADROIT6G: Distributed Artificial Intelligence-Driven Open & Programmable Architecture for 6G Networks.” [Online]. Available: <https://adroit6g.eu/>
 - [11] “6GTandem: Unlock new potential of wireless network.” [Online]. Available: <https://horizon-6gtandem.eu/>
 - [12] “FCC Opens Spectrum Horizons for New Services & Technologies - Federal Communications Commission.” [Online]. Available: <https://www.fcc.gov/document/fcc-opens-spectrum-horizons-new-services-technologies>
 - [13] “South Korea to launch 6G pilot project in 2026: Report.” [Online]. Available: <https://www.rcrwireless.com/20200810/asia-pacific/south-korea-launch-6g-pilot-project-2026-report>
 - [14] M. A. Uusitalo, P. Rugeland, M. R. Boldi, E. C. Strinati, P. Demestichas, M. Ericson, G. P. Fettweis, M. C. Filippou, A. Gati, M. H. Hamon, M. Hoffmann, M. Latva-Aho, A. Parssinen, B. Richerzhagen, H. Schotten, T. Svensson, G. Wikstrom, H. Wymeersch, V. Ziegler, and Y. Zou, “6G Vision, Value, Use Cases and Technologies from European 6G Flagship Project Hexa-X,” *IEEE Access*, vol. 9, pp. 160 004–160 020, 2021.
 - [15] 5G PPP Architecture Working Group, “The 6G Architecture Landscape European perspective,” 12 2022. [Online]. Available: <https://5g-ppp.eu/6g-architecture-landscape-new-white-paper-for-public-consultation/>
 - [16] Hexa-X, “Targets and requirements for 6G - initial E2E architecture,” 2022. [Online]. Available: https://hexa-x.eu/wp-content/uploads/2022/03/Hexa-X_D1.3.pdf
 - [17] D. Carrascal Acebron *et al.*, “Diseño y estudio de dispositivos IoT integrados en entornos SDN,” 2020.
 - [18] D. Carrascal, E. Rojas, J. M. Arco, D. Lopez-Pajares, J. Alvarez-Horcajo, and J. A. Carral, “A Comprehensive Survey of In-Band Control in SDN: Challenges and Opportunities,” *Electronics*, vol. 12, no. 6, p. 1265, 2023.
 - [19] M. U. Farooq, M. Waseem, S. Mazhar, A. Khairi, and T. Kamal, “A review on internet of things (iot),” *International Journal of Computer Applications*, vol. 113, no. 1, pp. 1–7, 2015.
-

- [20] V. Gazis, M. Goertz, M. Huber, A. Leonardi, K. Mathioudakis, A. Wiesmaier, and F. Zeiger, “Short paper: Iot: Challenges, projects, architectures,” in *2015 18th International Conference on Intelligence in Next Generation Networks*, 2015, pp. 145–147.
- [21] D. Hanes, G. Salgueiro, P. Grosssetete, R. Barton, and J. Henry, *IoT fundamentals: Networking technologies, protocols, and use cases for the internet of things*. Cisco Press, 2017.
- [22] “Terminology for constrained-node networks,” <https://tools.ietf.org/html/rfc7228>, accessed: 2020-06-30.
- [23] C. L. Devasena, “Ipv6 low power wireless personal area network (6lowpan) for networking internet of things (iot)—analyzing its suitability for iot,” *Indian Journal of Science and Technology*, vol. 9, no. 30, pp. 1–6, 2016.
- [24] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks: an authoritative review of network programmability technologies*. ” O'Reilly Media, Inc.”, 2013.
- [25] D. Levy and N. McKeown, “Overhaul may bring better, faster internet to 100 million homes,” *Stanford University News*, 2003.
- [26] “Onf overview,” <https://www.opennetworking.org/mission/>, accessed: 2020-06-10.
- [27] L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani, “Sdn controllers: A comprehensive analysis and performance evaluation study,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 6, pp. 1–40, 2020.
- [28] F. Tomonori, “Introduction to ryu sdn framework,” *Open Networking Summit*, pp. 1–14, 2013.
- [29] Nippon Telegraph and Telephone Corporation, “Ryu application API - Read the Docs,” 2014. [Online]. Available: https://ryu.readthedocs.io/en/latest/ryu_app_api.html
- [30] Isaku Yamahata, “Ryu SDN framework,” 2013. [Online]. Available: <https://www.slideshare.net/yamahata/ryu-sdnframeworkupload>
- [31] Elena Olkhovskaya, Ayaka Koshibe, “ONOS : An Overview,” 2016. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/ONOS+:+An+Overview>
- [32] Carmelo Cascone, “ONOS - Tutorials,” 2019. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Tutorials>
- [33] Peterson, Cascone, O'Connor, Vachuska, and Davie, “Software-Defined Networks: A Systems Approach,” 2022. [Online]. Available: <https://sdn.systemsapproach.org/onos.html>
-

- [34] O'Connor, Vachuska, and Davie, "Software-Defined Networks: Switch-Level Schematic," 2022. [Online]. Available: <https://sdn.systemsapproach.org/switch.html#switch-level-schematic>
- [35] Open vSwitch - Docs, "What Is Open vSwitch?" 2016. [Online]. Available: <https://docs.openvswitch.org/en/latest/intro/what-is-ovs/>
- [36] A. Mendiola, J. Astorga, E. Jacob, and M. Higuero, "A survey on the contributions of software-defined networking to traffic engineering," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 918–953, 2016.
- [37] E. L. Fernandes, E. Rojas, J. Alvarez-Horcajo, Z. L. Kis, D. Sanvito, N. Bonelli, C. Cascone, and C. E. Rothenberg, "The road to BOFUSS: The basic OpenFlow userspace software switch," *Journal of Network and Computer Applications*, vol. 165, p. 102685, 2020.
- [38] E. L. Fernandes, "Software Switch 1.3: An experimenter-friendly OpenFlow implementation," Ph.D. dissertation, PhD thesis. Universidade Estadual de Campinas, 2015.
- [39] Maxim Krasnyansky, "Universal TUN/TAP device driver," 2000. [Online]. Available: <https://www.kernel.org/doc/html/v5.8/networking/tuntap.html>
- [40] Ikluft, "diagram of the TUN and TAP drivers in the OSI network stack," 2019. [Online]. Available: <https://commons.wikimedia.org/wiki/File:Tun-tap-osilayers-diagram.png>
- [41] M. Kerrisk, *veth - Virtual Ethernet Device*, The Linux man-pages project, June 2020.
- [42] docker community, "Docker Docs - Networking overview," 2023. [Online]. Available: <https://docs.docker.com/network/>
- [43] bert hubert, "tc - show / manipulate traffic control settings," 2001. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc.8.html>
- [44] "Traffic control - debian," <https://wiki.debian.org/TrafficControl>, accessed: 2020-06-30.
- [45] M. Kerrisk, *Namespaces (7) - overview of Linux namespaces*, The Linux man-pages project, May 2020.
- [46] Michael Kerrisk, *Network namespaces - overview of Linux network namespaces*, The Linux man-pages project, June 2020.
- [47] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.
-

- [48] B. Heller, “Reproducible network research with high-fidelity emulation,” Ph.D. dissertation, Stanford University, 2013.
- [49] R. R. Fontes, S. Afzal, S. H. B. Brito, M. A. S. Santos, and C. E. Rothenberg, “Mininet-wifi: Emulating software-defined wireless networks,” in *2015 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 384–389.
- [50] “Mininet-iot,” <https://github.com/ramonfontes/mininet-iot>, accessed: 2020-07-01.
- [51] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *29th Annual IEEE International Conference on Local Computer Networks*, 2004, pp. 455–462.
- [52] A. Kurniawan, *Practical Contiki-NG: Programming for Wireless Sensor Networks*. Apress, 2018.
- [53] Adnk, “Contiki operating system running an IPv6 routing protocol on 41 nodes in the Cooja Contiki network simulator,” 2012. [Online]. Available: <https://commons.wikimedia.org/wiki/File:Contiki-ipv6-rpl-couja-simulation.png>
- [54] I. GitHub, “Github,” URL: <https://github.com/>, 2016.

Lista de Acrónimos y Abreviaturas

5G	the fifth generation of mobile technologies.
6G	the sixth generation of mobile technologies.
AI	Artificial Intelligence.
BOFUSS	Basic OpenFlow User-space Software Switch.
BPF	Berkeley Packet Filter.
eBPF	extended Berkeley Packet Filter.
ELF	Executable and Linkable Format.
eMBB	enhanced Mobile BroadBand.
FIFO	First In, First Out.
IEEE	Institute of Electrical and Electronics Engineers.
IETF	Internet Engineering Task Force.
IoT	Internet of Things.
LLDP	Link Layer Discovery Protocol.
LLN	Low power and Lossy Networks.
M2M	Machine to machine.
MIMO	Multiple-Input and Multiple-Output.
ML	Machine Learning.
mMTC	massive Machine-Type Communication.
NBI	Northbound Interface.
ONF	Open Networking Foundation.
ONOS	Open Network Operating System.
OvS	Open vSwitch.
P4	Programming Protocol-independent Packet Processors.
QoS	Calidad de servicio.
SBI	Southbound Interface.
SDN	Software-Defined Networking.
SNMP	Simple Network Management Protocol.
TC	Traffic control.

TFG	Trabajo Final de Grado.
TFM	Trabajo Final de Máster.
UAH	Universidad de Alcalá.
URLLC	Ultra-Reliable and Low-Latency Communication.
Veth	Virtual Ethernet Device.
XDP	eXpress Data Path.

A. Anexo I - Pliego de condiciones

En este anexo se podrán encontrar las condiciones materiales de las distintas máquinas donde se ha llevado a cabo el desarrollo y evaluación del proyecto. De forma adicional, se han indicado las limitaciones *hardware* así como las especificaciones *software* para poder replicar el proyecto de manera íntegra en otro sistema.

A.1. Condiciones materiales y equipos

A continuación, se presentan todas las máquinas empleadas en el desarrollo del proyecto, indicando únicamente aquellas características relevantes para la ejecución del TFM. De esta forma, se quiere garantizar que en caso de que se replique las pruebas y validaciones del proyecto se obtengan los mismos resultados, siendo el entorno completamente replicable.

A.1.1. Especificaciones Máquina A

- Procesador: i7-8700K (12) @ 4.700GHz
- Memoria: 15674MiB
- Gráfica: Intel UHD Graphics 630
- Sistema operativo: Ubuntu 20.04.6 LTS x86_64

A.1.2. Especificaciones Máquina B

- Procesador: Intel i5-7500 (4) @ 3.800GHz
- Memoria: 31984MiB
- Gráfica: Intel HD Graphics 630
- Sistema operativo: Ubuntu 22.04.2 LTS x86_64

```
arppath@arppath-desktop:~/TFM$ neofetch
  .-/+oossssoo+/-.
  `:+ssssssssssssssssssssss+`:
  .+ssssssssssssssssssssyyssss+-.
  .osssssssssssssssssdMMMyssso.
  /sssssssssshdmmNNmmyNMMMHssssss/
  +ssssssssshnydMMMMMMMddddyssssss+.
  /sssssssshdMMMNhyhyyyyhmNMMNhssssss/
  .ssssssssdMMMNhssssssssshNMMDssssss.
  +sssshhhyNMMNyssssssssssyNMMMyssssss+.
  ossyNMMNyMMhssssssssssshmmhssssss
  ossyNMMNyMMhssssssssssshmmhssssss
  +sssshhhyNMMNyssssssssssyNMMMyssssss+.
  .ssssssssdMMMNhssssssssshNMMDssssss.
  /sssssssshdmmNNmhyhyyyyhdNMMNhssssss/
  +ssssssssdnydMMMMMMMddddyssssss+.
  /sssssssssshdmmNNNmyNMMMHssssss/
  .osssssssssssssssdMMMyssso.
  +ssssssssssssssyyssss+-.
  `:+ssssssssssssssssss+`:
  .-/+oossssoo+/-.

arppath@arppath-desktop:~/TFM$ 
```

Figura A.1: Especificaciones de la máquina A

```
arppath@arppath-david:~$ neofetch
  .-/+oossssoo+/-.
  `:+ssssssssssssssssss+`:
  .+ssssssssssssssssyyssss+-.
  .osssssssssssssssdMMMyssso.
  /ssssssssshnydMMMMMMMddddyssssss+.
  /sssssssshdmmNNmmyNMMMHssssss/
  +ssssssssshnydMMMMMMMddddyssssss+.
  /sssssssshdmmNNmhyhyyyyhmNMMNhssssss/
  .ssssssssdMMMNhssssssssshNMMDssssss.
  +sssshhhyNMMNyssssssssyNMMMyssssss+.
  ossyNMMNyMMhssssssssssshmmhssssss
  ossyNMMNyMMhssssssssssshmmhssssss
  +sssshhhyNMMNyssssssssyNMMMyssssss+.
  .ssssssssdMMMNhssssssssshNMMDssssss.
  /sssssssshdmmNNmhyhyyyyhdNMMNhssssss/
  +ssssssssdnydMMMMMMMddddyssssss+.
  /sssssssshdmmNNNmyNMMMHssssss/
  .osssssssssssssdMMMyssso.
  +ssssssssssssssyyssss+-.
  `:+ssssssssssssssss+`:
  .-/+oossssoo+/-.

arppath@arppath-david:~$ 
```

Figura A.2: Especificaciones de la máquina B

A.1.3. Especificaciones Máquina C

- Procesador: Intel(R) Core(TM) 12th Gen i7-1260P (16) CPU @ 4.70Ghz
- Memoria: 15674MiB
- Gráfica: Intel Alder Lake-P
- Sistema operativo: Ubuntu 22.04.2 LTS x86_64

```
n0obie@n0obie-Zenbook:~$ neofetch
      .-/+o0sssoo+-.
      `:+ssssssssssssssssssss+-`:
      -+ssssssssssssssssssssss+-+
      .osssssssssssssssssdMMMNyssso.
      /sssssssssshdmmNnmnyNMNMNhsssss/
      +ssssssssshmydMMMMMMNmdddyssssssss+
      /ssssssshNMNMMyhhyyyhNmNMNMNhssssss/
      .ssssssssdMMMNhssssssssshNMNMdssssss.
      +sssshhhyNMNMysssssssssssyNMNMysssssss+
      ossyNMMNMymMhsssssssssssshmnhssssssso
      ossyNMMNMymMhsssssssssssshmnhssssssso
      +sssshhhyNMNMysssssssssssyNMNMysssssss+
      .ssssssssdMMMNhssssssssshNMNMdssssss.
      /ssssssshNMNMMyhhyyyhdNMNMNhssssss/
      +ssssssssdmymdMMMMMMMdddyssssssss+
      /sssssssssshdmNNNmyNMNMNhsssss/
      .osssssssssssssssdMMMNyssso.
      -+ssssssssssssssssyyssss+-.
      `:+ssssssssssssssss+-`:
      .-/+o0ssssoo+-.

n0obie@n0obie-Zenbook
-----
OS: Ubuntu 22.04.2 LTS x86_64
Host: Zenbook UX3402ZA_UX3402ZA 1.0
Kernel: 5.19.0-35-generic
Uptime: 3 days, 13 hours, 37 mins
Packages: 2118 (dpkg), 12 (snap)
Shell: bash 5.1.16
Resolution: 2880x1800
DE: GNOME 42.5
WM: Mutter
WM Theme: Adwaita
Theme: Yaru-dark [GTK2/3]
Icons: Yaru [GTK2/3]
Terminal: gnome-terminal
CPU: 12th Gen Intel i7-1260P (16) @ 4.700GHz
GPU: Intel Alder Lake-P
Memory: 3257MiB / 15624MiB
```

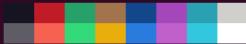


Figura A.3: Especificaciones de la máquina C

B. Anexo II - Presupuesto

En este anexo se quiere hacer una aproximación del presupuesto que tendría el proyecto llevado a cabo. Para ello, debemos hacer recuento de cuantas horas efectivas se han dedicado al proyecto, dado que, si bien es cierto que el TFM se ha extendido el doble del tiempo que se había estimado, no se ha dedicado el doble de semanas para la realización del mismo, dado que se ha ido compaginando con otros proyectos en paralelo. También hay que tener en cuenta los medios que se han utilizado para el desarrollo, clasificándolos en elementos *hardware* y en elementos *software*.

B.1. Duración del proyecto

Según se ha comentado anteriormente, con el propósito de obtener el número de horas **efectivas** dedicadas al proyecto, se va a realizar un diagrama de Gantt. De esta forma se pretende aclarar cuantas semanas se han dedicado a cada etapa del proyecto, y de este modo, poder llegar a estimar un número de horas efectivas de trabajo.

Se quiere aclarar, que la duración del proyecto se ha extendido en el tiempo casi un año más de lo previsto, sin embargo, esta extensión temporal no ha sido completamente efectiva sobre el proyecto, dado que se ha ido compaginando con otros proyectos en paralelo. Se quiere de esta forma obtener, aproximadamente, una estimación temporal en semanas de cuanto ha llevado cada etapa del TFM.

Como se puede ver en la figura B.1, la etapa crítica de este proyecto ha sido el aprendizaje del funcionamiento interno del BOFUSS y el desarrollo de las modificaciones sobre el mismo. Si bien es cierto que la curva de aprendizaje del BOFUSS es complicada, una de las grandes causas en la duración de dichas etapas ha sido la paralelización de este trabajo con otros proyectos, gastando bastante tiempo en los cambios de contexto entre ellos.

Número de horas totales	Horas efectivas	Horas efectivas por semana
24000h	≈ 1200 - 1600h	≈ 20h

Tabla B.1: Promedio de horas de trabajo

B.2. Costes del proyecto

Para realizar el cálculo de los costes del proyecto, se llevará a cabo una diferenciación previa en términos de *Hardware*, *Software* y mano de obra. Esta metodología permitirá un desglose detallado de los costes, lo que brindará una mayor claridad en cuanto a la cuantía total del proyecto. En primer lugar, se considerarán los costes relacionados con el *Hardware*. Esto implica evaluar los gastos asociados a la adquisición de equipos, dispositivos y componentes físicos necesarios para el desarrollo y funcionamiento del proyecto. En segundo lugar, se analizarán los costes de *Software*. Esto involucra evaluar los gastos relacionados con las licencias de software, el desarrollo de aplicaciones personalizadas, la adquisición de paquetes de software especializados y los costes de mantenimiento y actualizaciones de los programas utilizados en el proyecto. Finalmente, se tendrán en cuenta los costes de mano de obra. Esto incluirá los gastos relacionados con los recursos humanos involucrados en el proyecto, como los salarios de los empleados. Es importante destacar que al desglosar los costes del proyecto de esta manera, se proporcionará una visión más completa y detallada de los recursos financieros requeridos en cada área. Esto permitirá una mejor planificación, seguimiento y control de los costes de cara a futuro en el desarrollo de un proyecto de mismas características.

Producto (IVA incluido)	Valor (€)
Ordenador portátil Asus zenbook	1560,00
Servidor A	2100,00
Servidor B	1700,00
Pantalla Lenovo L27i	130,00
Pantalla Benq 21"	80,00
Periféricos	300,00
Infraestructura de Red (APs y Router Asus)	250,00

Tabla B.2: Presupuesto desglosado del Hardware para el TFM

Las licencias de software generalmente se venden por años, o por meses. Por ello, se ha calculado el precio equivalente asociado a la duración del TFM.

Producto (IVA incluido)	Valor (€)
Microsoft Office	300,00
Adobe Suite	500,00
Overleaf	40,00
G suite	120,00

Tabla B.3: Presupuesto desglosado del Software para el TFM

Para determinar los costes de mano de obra, se han utilizado como referencia los honorarios de un ingeniero senior, los cuales ascienden a 30€ por hora. Estos honorarios se aplicarán en función de la cantidad de horas efectivas dedicadas al proyecto. En cuanto a los costes relacionados con el *hardware* y el *software*, se han agrupado como un único elemento dentro del presupuesto. Se ha considerado el valor total resultante del desglose de los productos indicados en el análisis. Al agregar estos costes de *hardware* y *software* al presupuesto, se obtiene el valor total necesario para cubrir estos componentes esenciales del proyecto. Es importante destacar que estos cálculos se basan en las referencias proporcionadas y están sujetos a ajustes según las necesidades específicas del proyecto y las tarifas y precios aplicables en cada caso particular. El desglose y la inclusión de estos costes en el presupuesto garantizan una consideración adecuada de los recursos necesarios para el éxito y la ejecución del proyecto.

Descripción (IVA incluido)	Unidades	Coste unitario (€)	Coste total (€)
Material Hardware	1	6120,00	6120,00
Material Software	1	960,00	960,00
Mano de obra	1400	30,00	42000,00
Costes fijos (Luz, Internet)	1	650,00	650,00
TOTAL			49.730,00 €

Tabla B.4: Presupuesto total con IVA

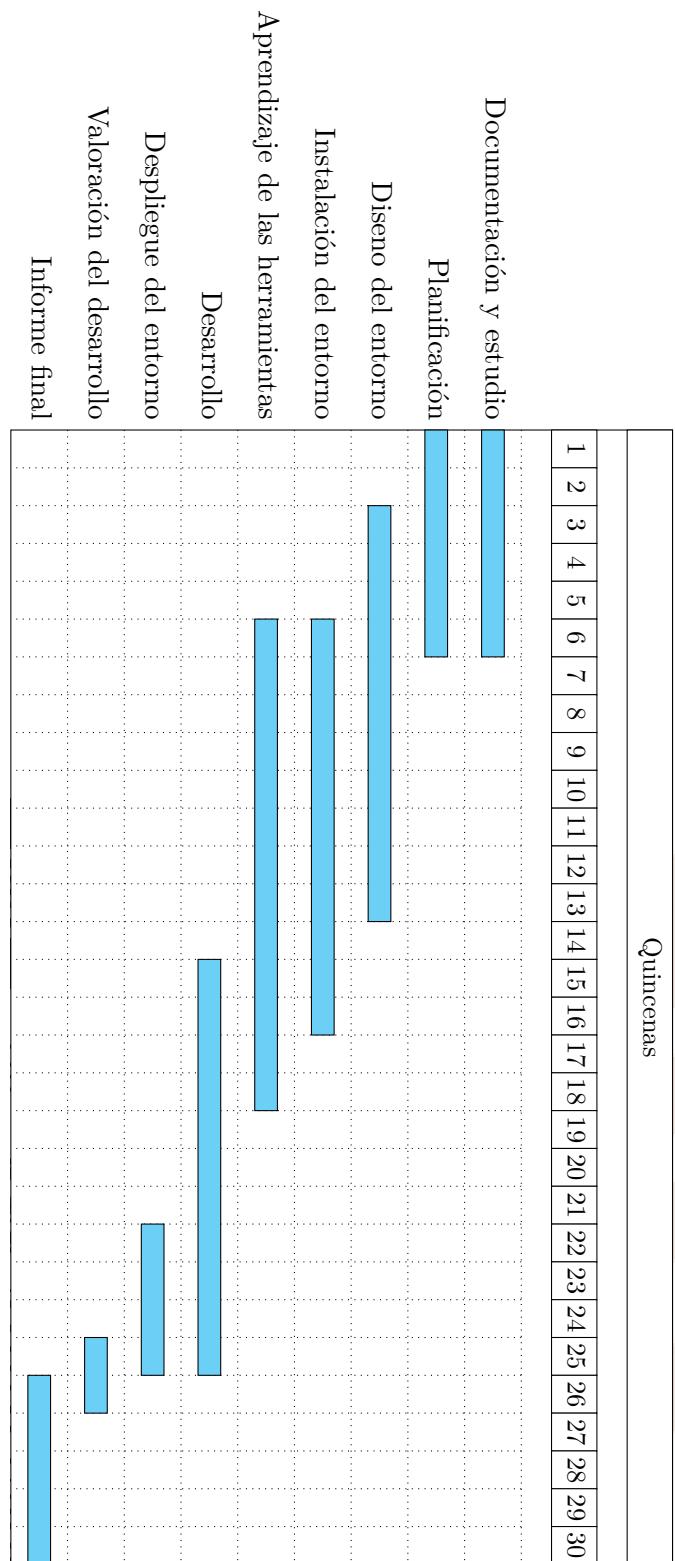


Figura B.1: Diagrama de Gantt del proyecto

C. Anexo III - Manuales de usuario e instalación

En este anexo se incluirán todos los manuales de usuario e instalación sobre aquellas herramientas que se crean necesarias para el desarrollo y comprobación de funcionamiento del TFG. De forma adicional, se comentará cómo funcionan los scripts de instalación generados para que cualquier persona interesada en replicar los distintos casos de uso, tenga un fácil acceso a ellos.

C.1. Instalación de dependencias de los casos de uso

La motivación de esta sección es plasmar en un punto como hacer uso de las herramientas que se han dejado desarrolladas para la instalación de las dependencias de los casos de uso. Como ya se indicó en el Pliego de condiciones, al tener dos entornos de trabajo muy diferenciados se iban a crear dos máquinas virtuales (??, ??) para conseguir aislar todo posible conflicto de dependencias. A continuación, se indicará como instalar las dependencias asociadas a cada entorno.

C.1.1. Instalación de dependencias máquina XDP

La tecnología XDP, al ser desarrollada propiamente en el Kernel de Linux, no necesitará de muchas dependencias para trabajar con ella. Todas las dependencias inducidas vienen por la necesidad de ciertos compiladores para establecer todo el proceso de compilación de un programa XDP, desde su C restringido hasta su forma de bytecode. Para instalar dichas dependencias se necesitará haber descargado el repositorio de este TFG en local. Esto se puede realizar según se indica en el bloque C.1.

Código C.1: Descarga del repositorio del TFG

```
1 # En caso de no tener "git" instalado lo podemos hacer de la siguiente forma
2 sudo apt install -y git
3
4
5 # Una vez que está instalado git, haremos un "clone" del repositorio
6 git clone https://github.com/davidcawork/TFG.git
```

Una vez descargado el repositorio se debería encontrar un directorio llamado TFG en el directorio donde se haya ejecutado dicho comando. El siguiente paso para instalar las dependencias, será movernos hasta el directorio de los casos de uso XDP y lanzar el script de instalación con permisos de super-usuario según se indica en el bloque C.2.

Código C.2: Instalación de dependencias XDP

```

1 # Nos movemos al directorio de los casos de uso XDP
2 cd TFG/src/use_cases/xdp/
3
4 # Lanzamos el script de instalación con permisos de super usuario
5 sudo ./install.sh

```

Este script de instalación añadirá los siguientes paquetes e inicializará el submódulo de la librería `libbpf`:

- Paquetes necesarios para el proceso de compilación de programas XDP: `clang llvm libelf-dev gcc-multilib`
- Paquete necesario para tener todos los archivos de cabecera del Kernel que se tenga instalado: `linux-tools-$(uname -r)`
- Paquete necesario en caso de querer hacer debug vía excepciones con la herramienta perf: `linux-tools-generic`

Por último, se quiere comentar el hecho de que es muy recomendable tener una versión superior a la v4.12.0 de iproute2 ya que en versiones anteriores no se da soporte para XDP. En Ubuntu 18.04 ya viene por defecto una versión compatible con XDP por lo que no será necesario actualizarla, más información en el punto C.2.

C.1.2. Instalación de dependencias máquina P4

El entorno de trabajo P4 es bastante áspero y complicado, ya que se requieren de numerosas dependencias para poder empezar a trabajar con la tecnología P4. Por ello, para la instalación del entorno de P4 se ha dejado un script de instalación en el directorio de los casos de uso P4, bajo la carpeta `vm` con el nombre de `install.sh`. En el repositorio oficial, hay un método de instalación similar pero enfocado a un aprovisionamiento de Vagrant¹.

El equipo de *p4lang* monta una máquina virtual personalizada que al parecer del autor de este TFG es demasiado *User Friendly* ya que deja poco margen de maniobra para hacer una instalación más perfilada a un entorno de desarrollo real. Por ello, se ha tenido que

¹<https://www.vagrantup.com/>

desarrollar un script propio para su instalación. Esta nueva vía de instalación fue ofrecida en forma de pull-request al equipo *p4lang* se puede consultar [aquí](#).

En primer lugar, se debe descargar el repositorio de este TFG. Si no lo ha hecho aún puede consultarla en el bloque C.1. Acto seguido, se deberá navegar hasta el directorio de los casos de uso P4 y lanzar el script como se indica en el bloque C.3.

Código C.3: Instalación de dependencias P4

```

1 # Nos movemos al directorio de los casos de uso XDP
2 cd TFG/src/use_cases/p4/
3
4 # Lanzamos el script de instalación con permisos de super usuario
5 sudo ./vm/install.sh -q

```

Este script de instalación añadirá los siguientes paquetes y herramientas necesarias para el desarrollo en P4:

- Paquetes necesarios que son dependencias de las herramientas principales de P4env.
- Herramientas del P4env: P4C PI P4Runtime
- Paquetes necesarios para la prueba de P4: Mininet BMV2 gRPC Protobuf

C.2. Herramienta *iproute2*

Se ha querido añadir esta sección, ya que la herramienta iproute2 va a ser fundamental a la hora de cargar los programas XDP en el Kernel, consultar interfaces, o verificar en qué *Network namespace* se encuentra el usuario. Por todo lo anterior, la herramienta iproute2 será una de las piezas claves para gestión de las *Network Namespaces*, y la verificación de los casos de uso.

C.2.1. ¿Qué es iproute2?

Iproute2 es un paquete utilitario de herramientas para la gestión del *Networking* en los sistemas Linux. Además, se encuentra ya en la mayoría de las distribuciones actuales. Sus desarrolladores principales son Alexey Kuznetsov y Stephen Hemminger, aunque hoy en día es un proyecto opensource donde cientos de personas contribuyen activamente en el repositorio².

Actualmente, la versión más reciente de la herramienta es v5.2.0. Dicha versión será la que se utilizará en Ubuntu 18.04. El conjunto de utilidades que ofrece iproute2 está pensado

²<https://github.com/shemminger/iproute2>

para la sustitución de herramientas que se recogen en el paquete de **net-tools**, como por ejemplo a **ifconfig**, **route**, **netstat**, **arp**, etc. En la tabla C.1 se pueden apreciar las herramientas de net-tools equivalentes en iproute2.

net-tools	iproute2
arp	ip neigh
ifconfig	ip link
ifconfig -a	ip addr
iptunnel	ip tunnel
route	ip route

Tabla C.1: Comparativa de herramientas Iproute2 con paquete net-tools

C.2.2. ¿Por qué necesitamos iproute2?

Cuando se está trabajando con los programas XDP y se quiere comprobar su funcionamiento, se debe compilarlos. Esto se hará con los compiladores LLVM³ más clang⁴, como ya se comentaba en el estado del arte. Este proceso de compilación convertirá el código de los programas XDP, en un *bytecode* BPF, y más tarde, se almacenará este *bytecode* en un fichero de tipo Executable and Linkable Format (ELF). Una vez compilados, se tendrá que anclarlos en el Kernel, y es en este punto es donde entrará iproute2, ya que tiene un cargador ELF (generalmente se trabajará con extensiones del tipo *.o).

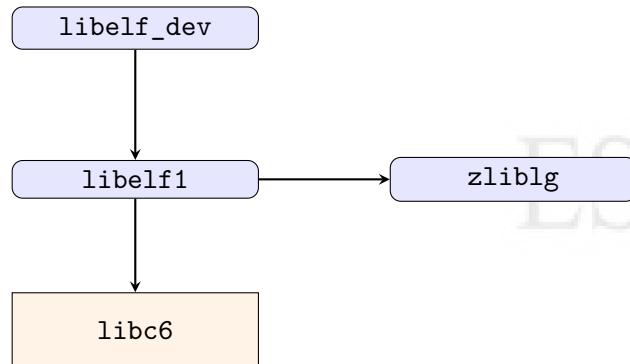
Además, la herramienta iproute2 permite al usuario comprobar si una interfaz tiene cargado un programa XDP. Arrojando en dicho caso, el identificador del programa XDP, que tiene anclado la interfaz y si este programa está cargado de una forma nativa o de una forma genérica. Al final de la esta sección, se indicará cómo hacer esta comprobación.

C.2.3. Estudio de compatibilidad de la herramienta iproute2 en Ubuntu

Al trabajar con esta herramienta para cargar programas XDP, se necesita la versión que soporte el cargador ficheros ELF. Si usted tiene la versión de iproute2 que viene instalada por defecto en Ubuntu 16.04, le indicamos que aún no da soporte a XDP. Inicialmente se buscó información relativa a partir de que versión se daba soporte a XDP tanto en Ubuntu 16.04, como en Ubuntu 18.04. Como no se encontró información precisa sobre ello, se ha realizado un estudio de la compatibilidad de iproute2 a través de Ubuntu 16.04 y Ubuntu 18.04.

³<https://llvm.org/>

⁴<https://clang.llvm.org/>

**Figura C.1:** Ramificación de dependencias de Iproute2.

Este estudio de compatibilidad se llevó a cabo descargando cada versión de iproute2, compilándola, e instalándola en nuestra máquina. Por último, para verificar si dicha versión daba soporte a XDP, se comprobaba si un programa XDP genérico que se sabía que funcionaba, cargaba o no, y si éste mostraba estadísticas sobre su carga. Más adelante, se indicará cómo compilar e instalar una versión en particular de iproute2.

Como se puede apreciar en la siguiente tabla C.2, en Ubuntu 16.04 a partir de la versión v4.14.0 no existe compatibilidad. Esto es debido a que requiere librerías de enlazado extensible de formato (No ELF Support). Para resolver este requerimiento se debería añadir una versión más reciente de la librería **libelf_dev**. Se puede agregar dicha librería, pero al hacerlo aparecerán dependencias que se van ramificando una a una llegando a librerías más sensibles para nuestro sistema como **libc6**, por lo que se decidió no comprobar el funcionamiento añadiendo las nuevas librerías requeridas para no comprometer el sistema.

Versiones IProute2	v4.9.0	v4.10.0	v4.11.0	v4.12.0	v4.13.0	v4.14.0	v4.15.0	v4.16.0	v4.17.0	v4.18.0	v4.20.0	v5.1.0	v5.2.0
Ubuntu 16.04	No XDP supp	No XDP supp	No XDP supp	Si	Si	No	No	No	No	No	No	No	No
Ubuntu 18.04	-	-	-	-	-	-	Si	Si	Si	Si	Si	Si	Si

Tabla C.2: Estudio de compatibilidad de la herramienta Iproute2

C.2.4. Compilación e instalación de iproute2

El proceso es prácticamente análogo tanto en Ubuntu 16.04 como en Ubuntu 18.04, salvo por una única diferencia que se indicará más adelante. Ahora se mostrarán los pasos necesarios para la compilación e instalación de una versión, en concreto de la herramienta iproute2.

- En primer lugar, se necesitará de instalar los paquetes necesarios para la configuración previa a la compilación.
 - **bison**, es un herramienta generadora de analizadores sintácticos de propósito general.
 - **flex**, es una herramienta para generar programas que reconocen patrones léxicos en el texto.
 - **libmnl-dev**, es una librería de espacio de usuario orientada a los desarrolladores de Netlink. Netlink⁵ es una interfaz entre espacio de usuario y espacio de Kernel vía sockets.
 - **libdb5.3-dev**, éste es un paquete de desarrollo que contiene los archivos de cabecera y librerías estáticas necesarias para la BBDD de Berkley (*Key/Value*).
 - Se entiende que se tiene el paquete **wget**. En caso de no tenerlo, solo se deberá añadir para poder descargar la herramienta.

Código C.4: Instalación de las dependencias de Iproute2

```
1 sudo apt-get install bison flex libmnl-dev libdb5.3-dev
```

- En segundo lugar, se debe descargar el comprimido de la herramienta iproute2. Al haber varios paquetes, se descargará aquel cuya versión sea con la que se quiere trabajar. Podemos descargarlas desde aquí: kernel.org.

Código C.5: Obtención del source de Iproute2

```
1 wget -c http://ftp.iij.ad.jp/pub/linux/kernel/linux/utils/net/iproute2/iproute2-4.15.0.tar.gz
```

- En tercer lugar, se debe descomprimir el comprimido de la herramienta. Acto seguido, se procederá a configurarla, compilarla e instalarla.

⁵<https://www.man7.org/linux/man-pages/man7/netlink.7.html>

Código C.6: Compilación e instalación de Iproute2

```

1  # Se descomprime y se entra al directorio
2  tar -xvfz $(tar).tar.gz && cd $tar
3
4  # Se configura
5  ./configura
6
7  # Se compila e instala, para añadir el nuevo binario en el path
8  sudo make
9  sudo make install

```

C.2.4.1. Diferencias con Ubuntu 18.04

La única diferencia en el proceso de instalación de la herramienta de iproute2 en Ubuntu 18.04, es añadir un paquete extra antes de proceder a configurar, compilar e instalar. El paquete extra es **pkg-config**; de no añadirlo fallará al lanzar el script de configuración y hacer el build.

Código C.7: Instalación de las dependencias de Iproute2 - Ubuntu 18.04

```
1  sudo apt-get install bison flex libmnl-dev libdb5.3-dev pkg-config
```

C.2.5. Comandos útiles con iproute2

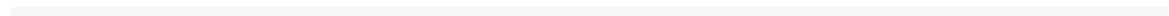
A continuación, se indican los comandos más frecuentes con la herramienta iproute2. Todos ellos han sido utilizados en el proceso de desarrollo del proyecto y en el proceso de verificación de los distintos casos de uso. Por ello, se considera que este apartado puede ser de gran utilidad para el lector que nunca ha trabajado con esta herramienta.

Código C.8: Comandos útiles con iproute2

```

1  # Listar interfaces y ver direcciones asignadas
2  ip addr show
3
4  # Poner/Quitar dirección a una interfaz
5  ip addr add {IP} dev {interfaz}
6  ip addr del {IP} dev {interfaz}
7
8  # Levantar/deshabilitar una interfaz
9  ip link set {interfaz} up/down
10
11 # Listar rutas
12 ip route list
13
14 # Obtener ruta para una determinada dirección IP
15 ip route get {IP}
16
17 # Listar Network namespace con nombre
18 ip netns list

```



C.3. Herramienta tcpdump

La motivación de añadir esta sección ha sido la de tener un punto de encuentro para las personas que nunca han utilizado tcpdump, ya que a lo largo de todas las secciones del proyecto, se hará uso de esta herramienta para verificar si los casos de uso realmente funcionan según lo esperado.

C.3.1. ¿Qué es tcpdump?

Tcpdump es un analizador de tráfico para inspeccionar los paquetes entrantes y salientes de una interfaz. La peculiaridad de esta herramienta es que funciona por línea de comandos, y tiene soporte en la mayoría de sistemas UNIX⁶, como por ejemplo Linux, macOS y OpenWrt. La herramienta está escrita en lenguaje C por lo que tiene un gran rendimiento y hace uso de libpcap⁷ como vía para interceptar los paquetes.

La herramienta fue escrita en el año 1988 por trabajadores de los laboratorios de Berkeley. Actualmente, cuenta con una gran comunidad de desarrolladores a su espalda en su repositorio oficial⁸ sacando nuevas actualizaciones de forma periódica (última versión v4.9.3).

C.3.2. ¿Por qué necesitamos tcpdump?

Hoy en día, es un hecho que en la mayoría de los casos no se suele desarrollar en una misma máquina. Se suele utilizar contenedores o máquina virtuales con el propósito de tener acotado el escenario de desarrollo. Por ello, se suele trabajar la mayoría de veces de forma remota, conectándose a la máquina/contenedor haciendo uso de ssh⁹.

Esto implica numerosas ventajas, pero también complicaciones. Si una persona no sabe configurar un *X Server* con el cual ejecutar aplicaciones gráficas de forma remota, no podría correr por ejemplo Wireshark. En este punto entra tcpdump, el cual no requiere de ningún tipo de configuración extra para poder ser ejecutado de forma remota. Esto añadido al hecho de su rápida puesta en marcha, con respecto a otros *sniffers* como Wireshark, han convertido a tcpdump en una herramienta fundamental en los procesos de verificación de los casos de uso.

⁶Unix es un sistema operativo desarrollado en 1969 por un grupo de empleados de los laboratorios Bell

⁷<https://github.com/the-tcpdump-group/libpcap>

⁸<https://github.com/the-tcpdump-group/tcpdump>

⁹<https://www.ssh.com/ssh/>

C.3.3. Instalación de tcpdump

Como ya se comentaba en la introducción, esta herramienta tiene un gran soporte entre los sistemas UNIX, por lo que generalmente suele encontrarse ya instalado en la mayoría de distribuciones Linux. De no tenerla instalada, siempre se podrá instalar de la siguiente forma C.9.

Código C.9: Instalación de Tcpdump

```
1 sudo apt install tcpdump
```

C.3.4. Comandos útiles con tcpdump

A continuación, se indican los comandos más frecuentes con la herramienta tcpdump. Todos ellos han sido utilizados en su mayoría en el proceso de verificación de los distintos casos de uso. Por lo tanto, se considera que este apartado puede ser de gran utilidad para el lector que nunca ha trabajado con esta herramienta. Además, se recomienda al lector consultar su *man-page*¹⁰ donde podrá encontrar información más detallada sobre el uso básico de tcpdump.

Código C.10: Comandos útiles con Tcpdump

```
1 # Indicar sobre que Interfaz se quiere escuchar
2 tcpdump -i {Interfaz}

3

4 # Almacenar la captura a un archivo para su posterior análisis
5 tcpdump -w fichero.pcap -i {Interfaz}

6

7 # Leer captura desde un archivo
8 tcpdump -r fichero.pcap

9

10 # Filtrar por puerto
11 tcpdump -i {Interfaz} port {Puerto}

12

13 # Filtrar por dirección IP destino/origen
14 tcpdump -i {Interfaz} dst/src {IP}

15

16 #Filtrar por protocolo
17 tcpdump -i {Interfaz} {protocolo}

18

19 # Listar interfaces disponibles
20 tcpdump -D

21

22 # Limitar el número de paquetes a escuchar
```

¹⁰<https://linux.die.net/man/8/tcpdump>

```
23    tcpdump -i {Interfaz} -c {Número de paquetes}
```

```
n0obie@n0obie-VirtualBox:~$ sudo tcpdump -i enp0s3 -c1
[sudo] contraseña para n0obie:
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
20:25:55.733209 ARP, Request who-has LAPTOP-7E128QAP.home tell liveboxfibra, length 46
1 packet captured
5 packets received by filter
0 packets dropped by kernel
n0obie@n0obie-VirtualBox:~$ █
```

Figura C.2: Interfaz CLI de Tcpdump

C.4. Herramienta Mininet

La motivación de añadir esta sección es la de poder indicar al lector como se ha instalado la herramienta de Mininet en las distintas máquinas descritas en el pliego de condiciones (Anexo A). Todo el proceso de instalación se ha llevado acabo en una máquina Linux, con las especificaciones indicadas en la siguiente tabla (Tabla C.3).

Distribución Linux	Cores	Memoria	Disco
Ubuntu 22.04 LTS	2	4096 MiB	40 GiB

Tabla C.3: Especificaciones máquina de instalación Mininet

Para la instalación necesitaremos de conectividad a Internet y de la herramienta `git` para clonar el repositorio de `Mininet`. En caso de no tener la herramienta de `git`, podremos instalarla ejecutando el comando del bloque de código C.11.

Código C.11: Instalación de la herramienta git

```
1  # El parametro -y se indica para confirmar la instalación de la herramienta
2  sudo apt install -y git
```

Una vez disponemos de la herramienta `git` para clonar el repositorio de `Mininet`, vamos a clonarlo, y lanzar el script de instalación que incorpora junto al source para instalar la herramienta. A continuación, en el bloque de código C.12 se indica los comandos ejecutados para instalar la herramienta de `Mininet`.

Código C.12: Instalación de la herramienta Mininet

```
1 # Clonamos el repositorio de Mininet
2 git clone https://github.com/davidcawork/mininet.git
3
4 # Accedemos al directorio
5 cd mininet
6
7 # Lanzamos el script de instalación (Openflow 1.3 - Ryu - Wireshark dissector)
8 sudo util/install.sh -3fmnyv
```

