

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

2022



Máster Universitario en Ingeniería de Telecomunicación



Madrid, 24 de octubre de 2022

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 Antonio Carral Pelayo

Vocal 1º: José Manuel Rodríguez Ascáriz

Vocal 2º: Elisa Rojas Sánchez

*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	19
2.5.1. OVS	19
2.5.2. BOFUSS	19
2.6. Linux Networking	19
2.6.1. Estructura <code>sk_buff</code>	19
2.6.2. Herramienta TC	22
2.6.2.1. Qdiscs	23
2.6.2.2. Classes	23

2.6.2.3. Filters	23
2.6.3. Namespaces	24
2.6.3.1. Persistencia de las Namespaces	25
2.6.3.2. Concepto de las Network Namespaces	25
2.6.3.3. Métodos de comunicación inter-Namespace: Veth	26
2.7. Mininet y Mininet-WiFi	27
2.7.1. Mininet	28
2.7.2. Funcionamiento de Mininet	29
2.7.3. Mininet CLI	33
2.7.4. Mininet-WiFi	33
2.7.5. Mininet-IoT	34
2.8. Contiki-ng	34
2.8.1. Simulador Cooja	35
2.9. Contribuciones en GitHub	36
Bibliografía	39
Lista de Acrónimos y Abreviaturas	43
A. Anexo I - Pliego de condiciones	45
A.1. Condiciones materiales y equipos	45
A.1.1. Especificaciones Máquina A	45
A.1.2. Especificaciones Máquina B	45
A.1.3. Especificaciones Máquina C	46
B. Anexo II - Presupuesto	49
B.1. Duración del proyecto	49
B.2. Costes del proyecto	50
C. Anexo III - Manuales de usuario e Instalación	53
C.1. Instalación de dependencias de los casos de uso	53
C.1.1. Instalación de dependencias máquina XDP	53
C.1.2. Instalación de dependencias máquina P4	54
C.2. Herramienta iproute2	55
C.2.1. ¿Qué es iproute2?	55
C.2.2. ¿Por qué necesitamos iproute2?	56
C.2.3. Estudio de compatibilidad de la herramienta iproute2 en Ubuntu	56
C.2.4. Compilación e instalación de iproute2	58
C.2.4.1. Diferencias con Ubuntu 18.04	59

C.2.5. Comandos útiles con <code>iproute2</code>	59
C.3. Herramienta <code>tcpdump</code>	61
C.3.1. ¿Qué es <code>tcpdump</code> ?	61
C.3.2. ¿Por qué necesitamos <code>tcpdump</code> ?	61
C.3.3. Instalación de <code>tcpdump</code>	62
C.3.4. Comandos útiles con <code>tcpdump</code>	62
C.4. Herramienta Mininet	63

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 generica de controlador SDN [27]	15
2.6.	Arquitectura del controlador SDN RYU [30]	18
2.7.	Sistema de colas doblemente enlazada	21
2.8.	Punteros de la estructura sk_buff	22
2.9.	Sistema de QoS implementado con distintas clases [31]	23
2.10.	Enlace entre interfaces Veth separadas en dos Network Namespaces	27
2.11.	Arquitectura de Mininet [36]	30
2.12.	Topología de ejemplo levantada	31
2.13.	Listado de Network Namespaces existentes en el sistema	31
2.14.	Listado de procesos asociados a Mininet	32
2.15.	Información relativa al proceso del Host1	32
2.16.	Información relativa al proceso del Host2	33
2.17.	Particionado en un sistema con Contiki OS [39]	35
A.1.	Especificaciones de la máquina A	46
A.2.	Especificaciones de la máquina B	46
A.3.	Especificaciones de la máquina C	47
C.1.	Ramificación de dependencias de Iproute2.	57
C.2.	Interfaz CLI de Tcpdump	63

Índice de tablas

2.1.	Resumen de los tipos de Namespaces en el Kernel de Linux	24
2.2.	Resumen comandos existentes en Mininet	33
2.3.	Resumen de contribuciones realizadas	37
B.1.	Promedio de horas de trabajo	50
B.2.	Presupuesto desglosado del Hardware	50
B.3.	Presupuesto desglosado del Software	50
B.4.	Presupuesto total con IVA	51
C.1.	Comparativa de herramientas Iproute2 con paquete net-tools	56
C.2.	Estudio de compatibilidad de la herramienta Iproute2	57
C.3.	Especificaciones máquina de instalación Mininet	63

Índice de Códigos

2.1. Estructura sk_buff_head	20
2.2. Comandos útiles con iproute2 - Netns	26
2.3. Manejo de Veths	26
2.4. Levantamiento de la topología de ejemplo	30
2.5. Listar Network Namespaces	30
C.1. Descarga del repositorio del TFG	53
C.2. Instalación de dependencias XDP	54
C.3. Instalación de dependencias P4	55
C.4. Instalación de las dependencias de Iproute2	58
C.5. Obtención del source de Iproute2	58
C.6. Compilación e instalación de Iproute2	59
C.7. Instalación de las dependencias de Iproute2 - Ubuntu 18.04	59
C.8. Comandos útiles con iproute2	59
C.9. Instalación de Tcpdump	62
C.10. Comandos útiles con Tcpdump	62
C.11. Instalación de la herramienta git	63
C.12. Instalación de la herramienta Mininet	63

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.

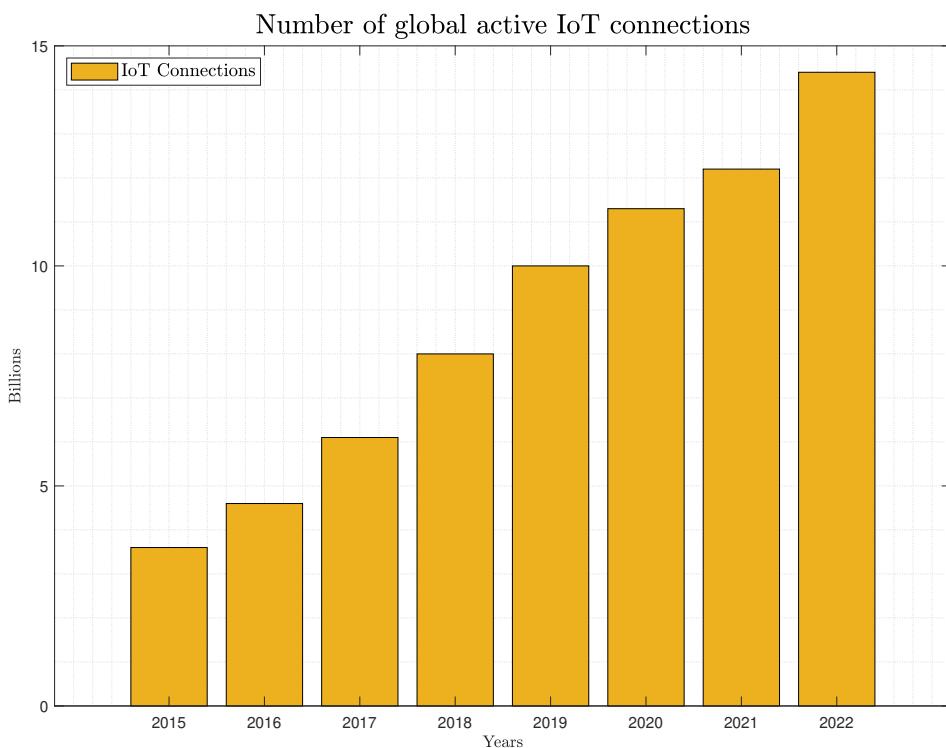


Figura 1.1: Estudio de las conexiones IoT máximas simultáneas a nivel global [3]

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.

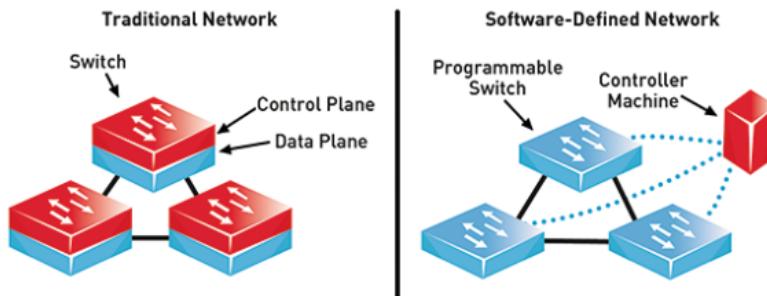


Figura 1.3: Paradigma en las redes SDN [17]

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].

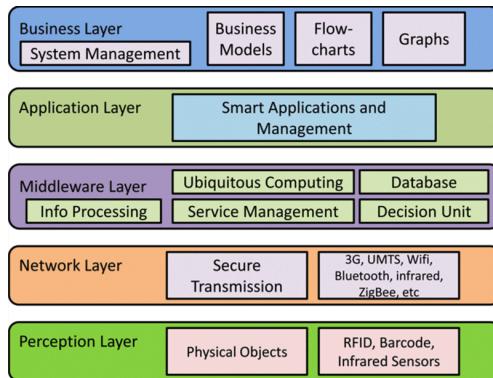
2.2.1. Arquitectura

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

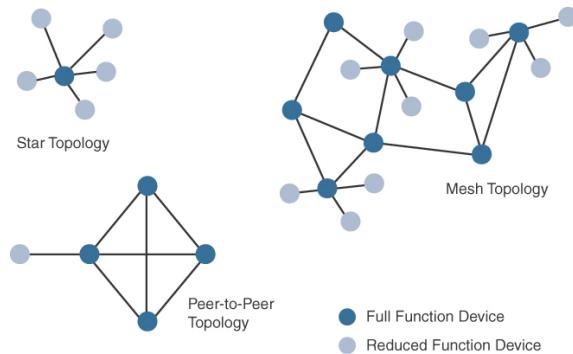
- *Perception Layer*, en esta capa se da un significado físico a cada objeto. Consiste en sensores de diferentes tipos como etiquetas RFID, sensores IR u otras redes de sensores que podrían detectar la temperatura, la humedad, la velocidad y la ubicación, etc. Esta capa recolecta información útil a partir 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].

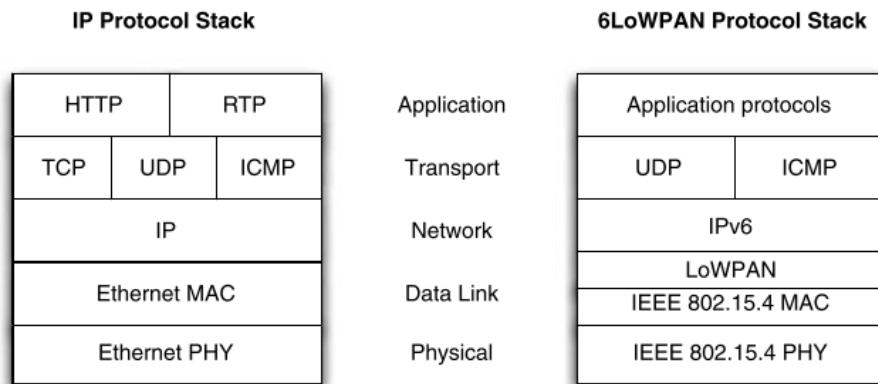


Figura 2.3: Pila de protocolos 6LoWPAN [23]

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/>

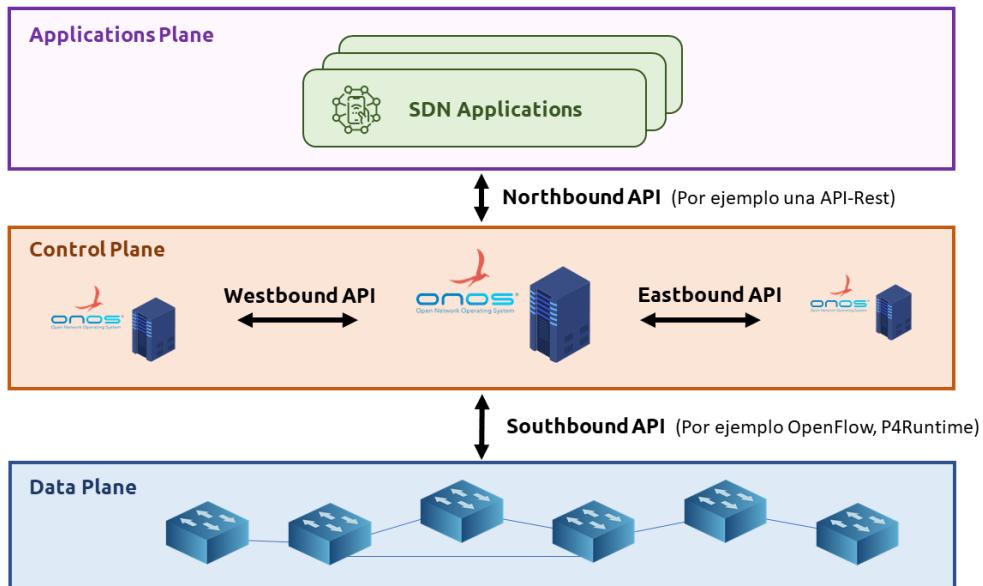


Figura 2.4: Arquitectura básica SDN

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.

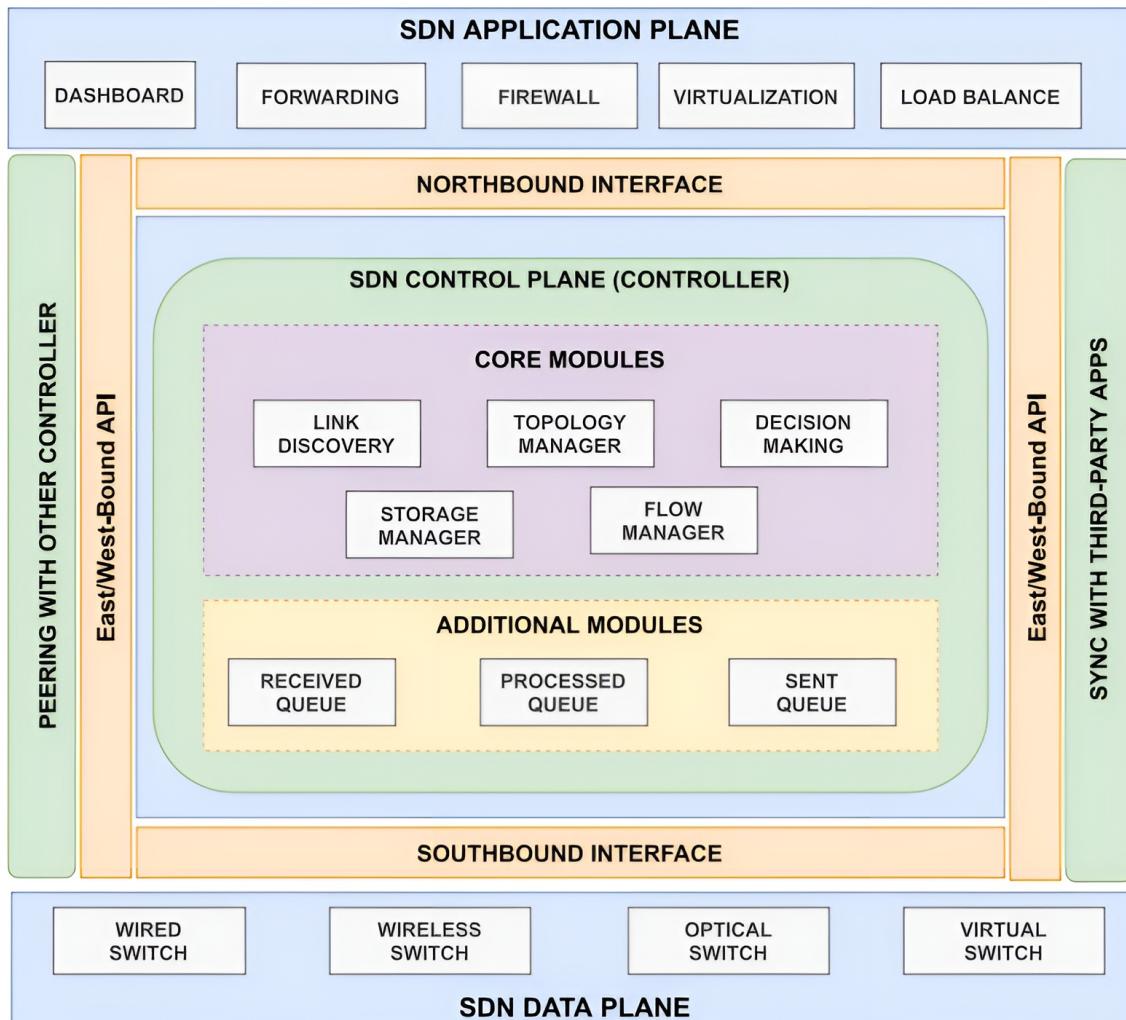


Figura 2.5: Arquitectura genérica de controlador SDN [27]

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.

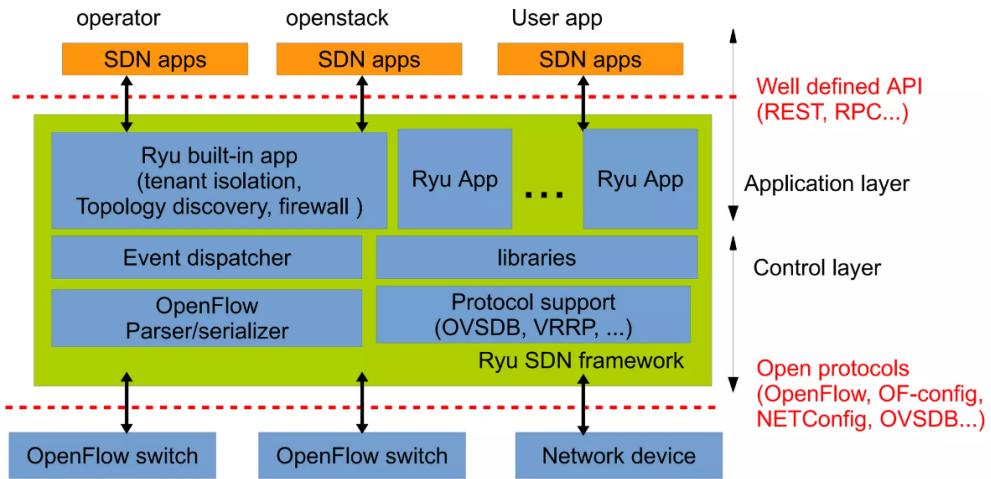


Figura 2.6: Arquitectura del controlador SDN RYU [30]

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

2.5. Software Switches SDN

2.5.1. OVS

2.5.2. BOFUSS

2.6. Linux Networking

En esta sección se van a recoger todos los conceptos y herramientas que se engloben en la parte de *Networking* de Linux y se crean fundamentales en el desarrollo de este proyecto.

2.6.1. Estructura `sk_buff`

Se puede afirmar que esta estructura es una de las más importantes de todo el Kernel de Linux en cuanto a la parte de *Networking* se refiere. Esta estructura representa las cabeceras o información de control para los datos que se van a enviar o para aquellos que se acaban de recibir. Dicha estructura se encuentra definida en el siguiente archivo de cabecera `<include/linux/skbuff.h>`, y contiene campos de todo tipo, para así tratar de ser una estructura “todo terreno” que tenga una gran versatilidad. Por ello, se puede llegar a entender que eXpress Data Path (XDP) no haga uso de esta estructura, ya que habría que hacer una reserva de memoria por cada paquete recibido y esto consumiría bastante tiempo viéndolo a gran escala, dadas las dimensiones de la estructura.

Esta estructura es usada por bastantes capas de la pila de protocolos del Kernel, y muchas veces la misma estructura es reutilizada de una capa hacia otra. Por ejemplo, cuando se genera una paquete TCP se añaden las cabeceras de la capa de transporte al payload, acto seguido se le pasa a la capa de red, IPv4 - IPv6. En esta capa se deben añadir de igual manera sus cabeceras, por ello la estructura `sk_buff` que gobierna dicho paquete tendrá que iniciar un proceso de adicción de cabeceras. Este proceso se lleva a cabo haciendo reservas de memoria en el *buffer* gestionado por la estructura, llamando a la función `skb_reserve`. Una vez que se tiene reservada memoria para las nuevas cabeceras éstas se copian, se actualizan los punteros de la estructura a las nuevas posiciones de inicio de paquete, y se delega la estructura *stack* de red abajo.

Podría surgir la siguiente pregunta, ¿Cuando llega un paquete qué ocurre? En primera instancia se creía que las cabeceras iban siendo eliminadas del paquete como si de una pila se tratase, cada capa que a travesaba del *stack* de red iba haciendo un *pop-out* de una cabecera. El funcionamiento sin embargo, se lleva a cabo con esta estructura, y no se opera como una

pila haciendo *reloc* eliminado capas, sino que en realidad se va moviendo un puntero que apunta al *payload* y a la información de control según en la capa del *stack* de red en el que se encuentre.

Por ejemplo, si el paquete se encuentra en la capa dos, el puntero de datos apuntaría a las cabeceras de capa dos. Una vez que estas se han parseado, y se pasa el paquete a la capa tres, el puntero se desplazaría a las cabeceras de capa tres. De esta manera el procesamiento de los paquetes supone menos ciclos de CPU, y se consigue un mejor rendimiento ya que no es necesario hacer un **realloc** con la nueva información “útil” del paquete.

Como ya se ha explicado, esta estructura juega un papel fundamental en el control de cabeceras, pero es igual de importante en el control de colas. Generalmente las colas se organizan en una lista doblemente enlazada, como toda lista doblemente enlazada elemento de la lista tiene un puntero que apunta al siguiente elemento y otro puntero que apunta al elemento anterior de la lista. Pero, una característica especial de esta estructura, es que cada estructura **sk_buff** debe ser capaz de encontrar la cabeza de toda la lista rápidamente, de esta forma hace la lista mucho más accesible. Por ello, cada elemento de lista tiene un puntero a una estructura del tipo **sk_buff_head**, la cual estará siempre al principio de la lista. La definición de esta estructura **sk_buff_head** es la siguiente.

Código 2.1: Estructura **sk_buff_head**

```

1  struct sk_buff_head {
2      /* These two members must be first. */
3      struct sk_buff *next;
4      struct sk_buff *prev;
5
6      __u32 qlen;
7      spinlock_t lock;
8  };

```

El elemento **lock** se utiliza como cerrojo para prevenir accesos simultáneos a la cola, será un atributo crucial para lograr la atomicidad en las operaciones relativas a la cola. En cuanto al elemento **next** y **prev** sirven como elementos para recorrer la cola apuntando estos al primer buffer y al último de ellos. Al contener estos elementos, **next** y **prev**, la estructura **sk_buff_head** es completamente compatible en la lista doblemente enlazada, ya que se puede recorrer la cola desde el primer *item* sin problemas. Por último, el elemento **qlen** es para llevar el número de elementos que hay en la cola en un momento dado.

Por claridad en la figura 2.7 no se ha dibujado el enlace de cada elemento de la lista hacia la cabeza de la misma. Pero recordemos que no es una simple lista enlazada, cada elemen-

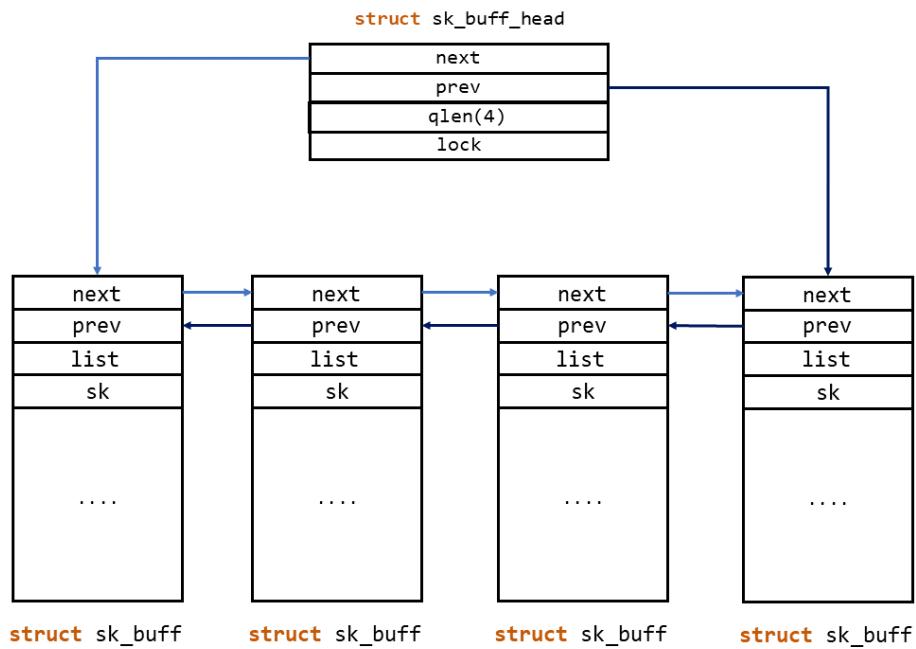


Figura 2.7: Sistema de colas doblemente enlazada

to de lista tiene un puntero que apunta al primer elemento de la lista, con su parámetro `list`.

Muchos campos de la estructura resultan muy familiares, ya que con XDP se tiene que replicar su funcionalidad al no poder disponer de ellos. Pero hay campos en las dos estructuras que son constantes; estos son los punteros que apuntan a posiciones clave del paquete en memoria.

Estos cuatro punteros a memoria son de utilidad cuando se tiene que llevar a cabo una reserva de memoria para añadir una nueva cabecera. El puntero `head` siempre apuntará al principio de la posición de memoria reservada en primera instancia, el puntero `data` siempre apuntará al principio del paquete, por lo que si queremos superponer una cabecera sobre otra simplemente tendremos que reservar la memoria suficiente entre ambos punteros. En cuanto a los punteros, `tail` y `end` son útiles para añadir información de control al final del paquete, como por ejemplo el CRC de Ethernet.

Esta estructura tiene muchísimas aplicaciones y usos en el Kernel de Linux. En nuestro caso, está estructura está relacionada con el proyecto, por el hecho de que numerosos *helpers* Berkeley Packet Filter (BPF) hacen uso de ella, pero con XDP no podremos al disponer de ella. Por ello, se tendrán que plantear soluciones a la ausencia de funcionalidades por los

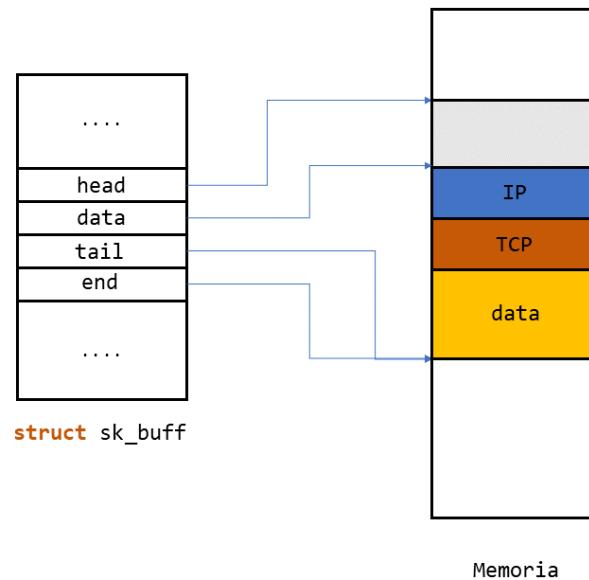


Figura 2.8: Punteros de la estructura sk_buff

helpers BPF, como por ejemplo clonar paquetes, muy útil para difundir paquetes por toda la red o *broadcast*.

2.6.2. Herramienta TC

Traffic control (TC), es un programa de espacio de usuario el cual es la pieza fundamental de la QoS en el Kernel de Linux. Muchas de sus funcionalidades se pueden resumir en cuatro puntos:

- SHAPING
- SCHEDULING
- POLICING
- DROPPING

Según su *man-page*⁵, el procesamiento del tráfico para conseguir dichas funcionalidades, se lleva a cabo con tres tipos de objetos: **qdiscs**, **classes** y **filters**.

⁵<https://man7.org/linux/man-pages/man8/tc.8.html>

2.6.2.1. Qdiscs

El objeto Queueing discipline (Qdiscs), disciplina de cola, es un concepto básico en el Networking de Linux que indica el orden en que los miembros de la cola, en este caso paquetes, se seleccionan para el servicio. Por ejemplo, en un momento dado puede que una herramienta de espacio de usuario requiera de transmitir un paquete, dicho paquete será entregado al *stack* de red, llegando en última instancia a la interfaz de red por la cual va a ser transmitido. En ese momento el paquete se encontrará encolado en una cola de espera de ser transmitido, estas colas estarán regidas por un Qdiscs. El Qdiscs por defecto es un pfifo, es un puro *first-in, first-out* con limitación en el tamaño de cola en número de paquetes.

2.6.2.2. Classes

Las clases se podrían ver como una sub-Qdiscs de una Qdiscs. Una clase puede contener a su vez otra clase, o más clases, pudiendo conformar sistemas de QoS en detalle, véase la figura 2.9. Cuando los paquetes son recibidos en una cola administrada por un Qdiscs, estos pueden ser encolados en base a las características del paquete en otras colas, gestionadas por otras clases. Esto permite por ejemplo, priorizar el envío de datos de una aplicación sobre otra. Para ello, los paquetes de ambas aplicaciones se clasificarán en distintas clases, dándole más prioridad a una clase sobre la otra, asignándole más recursos de transmisión y recepción.

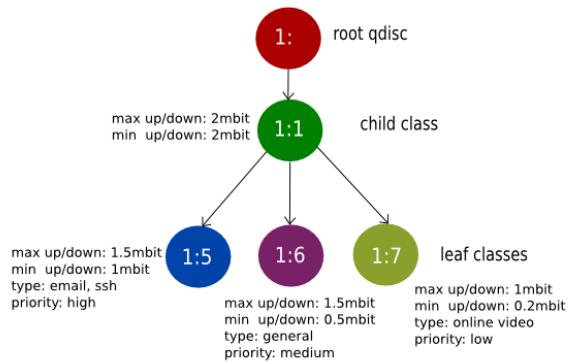


Figura 2.9: Sistema de QoS implementado con distintas clases [31]

2.6.2.3. Filters

Un filtro es usado para determinar con qué clase debe ser encolado el paquete. Para ello, el paquete siempre debe ser clasificado con una clase determinada. Varios tipos de filtros se

pueden utilizar para clasificar los paquetes, pero en este caso será de interés el tipo de filtro BPF⁶, los cuales permiten anclar un *bytecode* BPF. Estos filtros se utilizarán para cargar programas BPF que trabajarán en conjunto con XDP con la finalidad de lograr el Broadcast. Esto es así, ya que en el TC ya se hace uso de la estructura `sk_buff` (Ir a 2.6.1), por lo que ciertos *helpers* BPF para clonar paquetes podrán ser utilizados por *bytecode* anclado en el filtro.

2.6.3. Namespaces

Una *Namespace* se utiliza para aislar un recurso del sistema en una abstracción que hace creer a los procesos dentro de dicha *Namespace* que tienen su propia instancia del recurso en cuestión aislada del sistema real. Los cambios realizados sobre recursos aislados del sistema, son solo visibles para procesos que son pertenecientes a la *Namespace*, pero son invisibles para otros procesos pertenecientes al sistema o a otra *Namespace*.

Hay muchos tipos de *Namespaces*, en la tabla 2.1 se pueden apreciar todos los tipos que existen a día de hoy. El uso de todos estos tipos de *Namespaces* puede ser muy variado, pero el más importante de todos ellos son los contenedores. Los contenedores son elementos para correr aplicaciones, o herramientas en un entorno aislado del sistema. A día de hoy, los contenedores más utilizados son los de Docker⁷, pero, ¿Realmente qué hace Docker, qué es Docker?

Docker se vale de las bondades del Kernel de Linux para aislar recursos y con esto conformar contenedores. Éste hará uso de las APIs suministradas por el Kernel para crear y destruir a conveniencia las distintas *Namespaces*. Por lo que, Docker en verdad simplemente es un envoltorio de las llamadas al sistema para gestionar *Namespaces*. Docker además, provee de distintos aspectos al usuario como *copy-on-write*, o una configuración en modo *bridge* hacia el exterior, pero en el fondo es un mero *wrapper* para la gestión de las *Namespaces* con la finalidad de implementar contenedores.

Tipo de Namespace	Descripción
Cgroup	Namespace utilizada generalmente para establecer unos límites de recursos, por ejemplo, CPU, memoria, lecturas y escrituras a disco, de todos los procesos que corran dentro de dicha Namespace.
Time	Namespace para establecer una hora del sistema diferente a la del sistema.
Network	Namespace utilizada para tener una replica aislada del stack de red del sistema, dentro del propio sistema.
User	Namespace utilizada para tener aislados a un grupo de usuarios.
PID	Namespace utilizada para tener identificadores de proceso independientes de otras namespaces.
IPC	Namespace utilizada para aislar los mecanismos de comunicación entre procesos.
Uts	Namespace utilizada para establecer un nombre de Host y nombre de dominio diferentes de los establecidos en el sistema
Mount	Namespace utilizada 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

⁶<https://man7.org/linux/man-pages/man8/tc-bpf.8.html>

⁷<https://www.docker.com/>

2.6.3.1. Persistencia de las Namespaces

Atendiendo a la *man-page* [32] sobre *Namespaces*, es importante señalar que las *Namespaces* tienen una vida finita. La vida finita de la *Namepace* dependerá de si la *Namepace* en cuestión está referenciada, por lo que éstas vivirán siempre y cuando estén referenciadas, cuando dejen de estarlo serán destruidas.

Este concepto de vida finita, será útil entenderlo para tener una mejor comprensión sobre el funcionamiento interno de Mininet ó Mininet-WiFi, los cuales se valen de estos conceptos para ahorrarse operaciones y ganar en rendimiento. Una *Namepace* a día de hoy puede ser referenciada de tres maneras distintas:

- Siempre y cuando haya un proceso corriendo dentro de esta *Namepace*.
- Siempre que haya abierto un descriptor de archivo al fichero identificativo de la *Namepace* (`/proc/pid/ns/tipo_namespace`).
- Siempre que exista un *bind-mount* del fichero (`/proc/pid/ns/tipo_namespace`) de la *Namepace* en cuestión.

Si ninguna de estas condiciones se cumple, la *Namepace* en cuestión es eliminada automáticamente por el Kernel. Si se tratase de una *Network Namespace*, aquellas interfaces que se encuentren en la *Namepace* en desaparición volverán a la *Network Namespace* por defecto [32].

2.6.3.2. Concepto de las Network Namespaces

Una vez entendido el concepto de *Namepace* en Linux, se introducen las *Network Namespace*, las cuales serán fundamentales para las plataformas donde se probarán los distintos test. Éstas consisten en una replica lógica de *stack* de red que por defecto tiene Linux, replicando rutas, tablas ARP, Iptables e interfaces de red [33].

Linux se inicia con un *Network Namespace* por defecto, el espacio *root*, con su tabla de rutas, su tabla ARP, Iptables e interfaces de red. Pero también es posible crear más *Network Namespace* no predeterminadas, crear nuevos dispositivos en esos espacios de nombres, o mover un dispositivo existente de un espacio de nombres a otro.

Para llevar todas estas tareas a cabo, la herramienta más sencilla será `iproute2` (Ir a Anexo C.2). Esta herramienta, haciendo uso del módulo `netns`, se podrá gestionar todo en lo relativo a las *Network Namespace* con nombre. Esta coletilla, “con nombre”, atiende a que

todas las *Network Namespace* que se gestionen desde `iproute2` serán persistentes debido a que se realizará un *bind-mount* con el nombre de la *Namespace*, del fichero identificativo de la *Namespace* en cuestión, bajo el directorio `/var/run/netns`. A continuación, se listan los comandos más frecuentes a la hora de gestionar *Network Namespaces*, se entiende que se ejecutan con permisos de súper usuario.

Código 2.2: Comandos útiles con iproute2 - Netns

```

1 # Para crear una Network Namespace
2 ip netns add {nombre netns}
3
4 # Para listar las Network Namespaces "con nombre"
5 ip netns list
6
7 # Para añadir una interfaz a una Network Namespace
8 ip netns set {nombre netns} Veth
9
10 # Para ejecutar un comando dentro de una Network Namespace
11 ip netns exec {nombre netns} {cmd}
12
13 # Para eliminar una Network Namespace
14 ip netns del {nombre netns}
```

2.6.3.3. Métodos de comunicación inter-Namespace: Veth

Las Virtual Ethernet Device (Veth), son interfaces de Ethernet virtuales creadas como un par de interfaces interconectadas entre si. El modelo funcional es sencillo, los paquetes enviados desde una son recibidos por la otra interfaz de forma directa, bastante parecido al funcionamiento de las *pipes*. Una condición interesante de estas interfaces es que su gestión está asociada, es decir, si se levanta una extremo de la Veth, la otra también lo hará, si por el contrario se deshabilita o se destruye algún extremo de un par de Veth el otro extremo también se verá afectado [34].

Es muy común hacer uso de las Veth para interconectar *Network Namespaces*, ya que, sabiendo que estas van a estar conectadas de forma directa, se puede utilizar este enlace como pasarela entre dos *Network Namespaces*. De esta forma, se estaría interconectando dos *stacks* independientes de red. La creación y destrucción de este tipo de interfaces se puede apreciar en el bloque 2.3, se recuerda que es necesario permisos de súper usuario.

Código 2.3: Manejo de Veths

```

1 # Crear un par veth
2 ip link add {nombre_veth1} type veth peer name {nombre_veth2}
3
4 # Si se elimina un extremo, el otro también lo hará
```

```
5 ip link delete {nombre_veth}
```

Por lo tanto, se puede llegar al siguiente diagrama básico del funcionamiento de un par de Veth, las cuales estarán asignadas a una *Network Namespace* distinta. Como se puede apreciar en la figura 2.10, ambas interfaces están conectadas entre si directamente de forma interna en el propio Kernel. En el caso de que se generen paquetes desde una *Network Namespace* hacia la otra, estos paquetes llegarán desde un extremo de la Veth directamente al otro extremo de la Veth a través del Kernel, y en este caso, la *Network Namespace* por defecto no percibirá dicho tráfico.

Esta condición será de gran utilidad para recrear enlaces entre nodos independientes de red, los nodos se replicarán con *Network Namespaces* y los enlaces con Veths. Estos mecanismos serán utilizados por herramientas de emulación de redes como Mininet o Mininet-WiFi, más adelante se destallará.

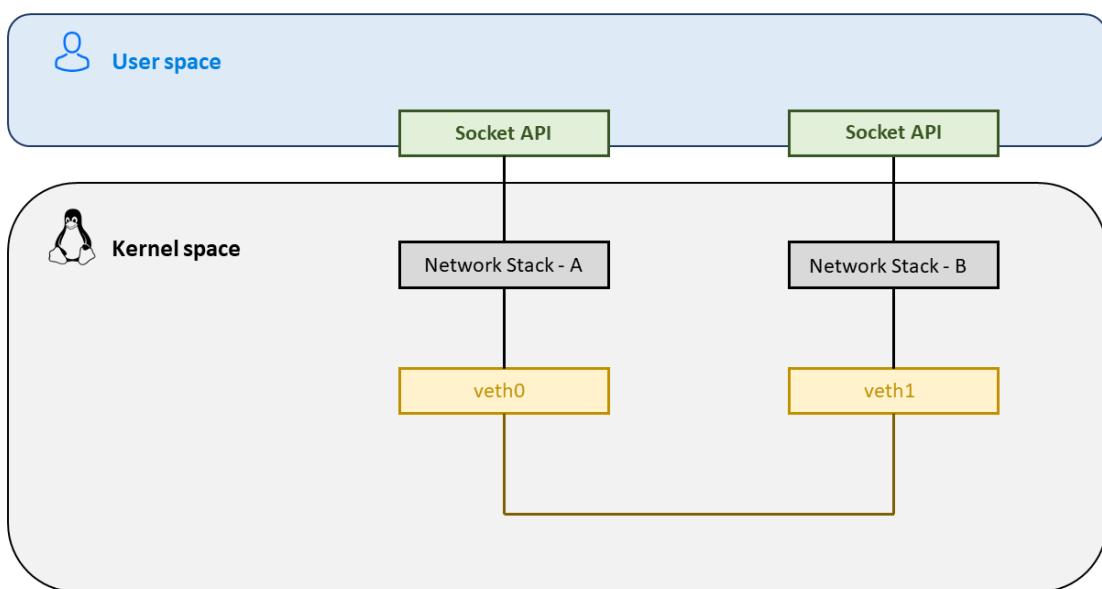


Figura 2.10: Enlace entre interfaces Veth separadas en dos Network Namespaces

2.7. Mininet y Mininet-WiFi

En esta sección se cubrirá el marco teórico sobre las principales herramientas de emulación utilizadas en este proyecto. Se indagará en mayor parte Mininet, al ser la herramienta

principal desde la cual han nacido otras herramientas de emulación como Mininet-WiFi.

2.7.1. Mininet

Mininet es una herramienta que se utiliza para **emular** redes, generalmente del tipo SDN. Con ella se pueden emular host, routers, switches y enlaces en una misma máquina a un bajo coste, pero con la condición de contar con el Kernel de Linux en dicha máquina. Para lograr este cometido se hace uso de una virtualización “ligera”, la cual consiste en hacer uso de las bondades del Kernel de Linux para virtualizar recursos, como son las *Namespaces* (Ir a 2.6.3) [35].

En función de las características de cada nodo, se virtualizarán más o menos recursos, y esto dependerá también en el rendimiento de la emulación. Por ejemplo, los nodos del tipo **Host** en Mininet requieren hacer uso de una *Network Namespace*, de esta forma tendrán su propio *stack* de red y serán completamente independientes⁸ del sistema y de otros nodos de la red a emular. Sin embargo, por defecto todos los nodos **Host** comparten sistema de archivos, numeración de PIDs, usuarios, etc. Por lo que técnicamente hablando no están aislados completamente como de un propio **Host** real se tratase. Esto se debe a que Mininet virtualizará solo aquellos recursos que sean los estrictamente necesarios para llevar a cabo la emulación, de esta forma, se obtiene un mejor rendimiento, y además, permite que máquinas con pocos recursos sean capaces de realizar la emulación [35].

En cuanto a la creación de las topologías a emular en Mininet, existen dos vías para hacerlo. La primera es hacer uso de la API escrita en Python para interactuar con las clases de Mininet. Con ella, se podrá conformar toda la topología importando los módulos y clases necesarias de la API para definir dicha topología en un script de Python. La segunda vía es hacer uso de la herramienta **MiniEdit**, la cual ofrece al usuario una Graphical User Interface (GUI) donde podrá crear la topología emular arrastrando al clickble los distintos nodos de la red. De la misma GUI además se podrá exportar la topología generada a un fichero (*.mn) para recuperarla más tarde, o a un script en Python (*.py) para levantarla cuando se quiera con el interprete de Python. Esta herramienta es de gran utilidad para las personas que no saben programar en Python y quieran hacer uso del emulador, por lo que es un gran punto a favor.

Por tanto, se podrían resumir los aspectos más fuertes de Mininet en los siguientes puntos:

- Es rápido, debido a su condición de diseño con *Namespaces*, más adelante se indicará

⁸En la parte de Networking

como se lleva a cabo su gestión.

- No consume recursos en exceso, virtualiza únicamente lo necesario, y en el caso que fuera necesario se pueden establecer los recursos máximos para la emulación.
- Ofrece libertad al usuario para crear topologías y escenarios personalizados a través de la API en Python de Mininet. Además, estos escenarios son fácilmente extrapolables⁹ a otra máquina, ya que únicamente se debe compartir el script que describe la topología.

Al igual que se han indicado los puntos fuertes, se va a indicar la mayor limitación de Mininet. Como ya se comentaba antes Mininet hace uso de una virtualización “ligera”, la cual está sustentada por las *Namespaces* del Kernel de Linux. Es cierto que esta decisión de diseño da muchos beneficios en rendimiento al hacer uso del propio sistema para virtualizar recursos, pero el problema llega cuando este mismo emulador quiere ser exportado a otra plataforma, con un sistema operativo distinto. El cual puede que no soporte el equivalente funcional en “*Namespaces*”, o en caso de hacerlo, su API para hacer uso de ellas sea completamente diferente.

2.7.2. Funcionamiento de Mininet

Se ha introducido anteriormente que Mininet hace uso de *Network Namespaces* como método para virtualizar *stacks* de red independientes entre sí, y así poder emular redes a un coste mínimo. En la figura 2.11, se puede ver la arquitectura interna de Mininet para una topología compuesta de dos **Host**, y de un soft-switch conectado por TCP a un controlador remoto.

Como se puede apreciar los host están aislados en sus propias *Namespaces*, y en este caso el switch está corriendo en la *Namepace* por defecto (root). El mecanismo para comunicar a los nodos de esta topología como se adelantaba anteriormente son las Veths (Ir a 2.6.3.3), las cuales permitirán emular los enlaces entre los distintos nodos de la red.

Una vez expuesta toda la teoría sobre Mininet, podría surgir la siguiente pregunta, ¿Cómo se puede comprobar que realmente hace uso de *Network Namespaces*? Lo primero que se debe hacer es levantar el escenario para que así, Mininet cree las *Network namespaces* que tenga que crear. En este caso, se utilizará la topología expuesta en la figura 2.11, para levantar dicha topología únicamente se tiene que tener Mininet instalado y seguir los pasos que se indican el bloque 2.4.

⁹Los resultados de las pruebas no tienen por que ser exactos en dos máquinas distintas, se emula, no se simula. Por tanto se depende de las condiciones de la máquina donde se vayan a correr las pruebas

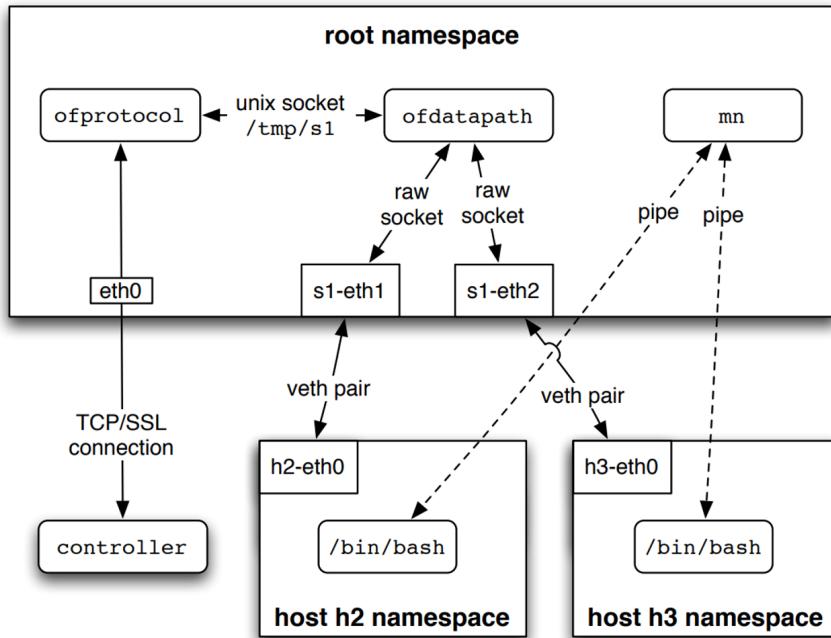


Figura 2.11: Arquitectura de Mininet [36]

Código 2.4: Levantamiento de la topología de ejemplo

```
1 # Por defecto siempre carga la topología descrita en la figura anterior
2 sudo mn
```

Ahora que ya se ha levantado el escenario se debería ser capaces de poder ver si hay *Network Namespaces* en el sistema, para ello se hará uso del *pack* de herramientas iproute2 (Ir a C.2). El comando por excelencia para listar las *Network Namespaces* haciendo uso del módulo **netns** se puede ver en el bloque 2.5.

Código 2.5: Listar Network Namespaces

```
1 sudo ip netns list
```

Según se puede apreciar en la figura 2.13, no parece que haya ninguna *Network Namespace* en el sistema, pero entonces, ¿Dónde está el problema? El problema de que el comando **ip netns list** no arroje información, es que Mininet no está creando el *softlink* requerido para que la herramienta sea capaz de listar las *Network Namespaces*. Atendiendo a la documentación del comando se puede averiguar que dicho comando lee del path **/var/run/netns/** donde se encuentran todas las *Network Namespaces* con nombre¹⁰.

¹⁰Aquellas Netns las cuales se ha hecho un bindmount con su nombre en ese directorio para que persistan aunque no haya ningún proceso corriendo en ellas.

```
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 2.12: Topología de ejemplo levantada

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

Figura 2.13: Listado de Network Namespaces existentes en el sistema

Como ya se explicó, las *Namespaces* tienen una vida finita, éstas viven siempre y cuando estén referenciadas (Ir a 2.1), por tanto, si ninguna condición de referenciación se cumple, la *Namespace* en cuestión es eliminada.

Mininet se encarga de recrear la red emulada, y cuando el usuario termine la emulación, la red emulada debe desaparecer, este proceso debe ser lo más ligero y rápido posible para así ofrecer una mejor experiencia al usuario. La naturaleza del diseño de Mininet incita a pensar que la creación y destrucción de las *Network Namespace* vienen asociadas a la primera condición de refereciación de una *Namespace*.

Es decir, no tendría sentido hacer *mounts* ni *softlinks* que a posteriori se deberán eliminar, ya que supondría una carga de trabajo bastante significativa para emulaciones de redes grandes y un aumento del tiempo destinado a la limpieza del sistema una vez que la emulación haya terminado. Además, se debe tener en cuenta que existe una condición que es bastante idónea con las necesidades de Mininet, ya que solo es necesario un proceso corriendo por cada *Network Namespace*, y a la hora de limpiar únicamente se debe terminar con los procesos que sostienen las *Network Namespace*. Cuando ya no haya ningún proceso corriendo en la *Namespace*, y el Kernel se encargará de eliminar las *Namespaces*.

Según el razonamiento expuesto, se debería ver varios procesos que son creados a la ho-

ra del levantamiento del escenario en Mininet. Estos procesos deberán tener cada uno un fichero de *Network Namespace*, `/proc/{pid}/ns/net`, con un *inode* distinto para aquellos procesos que corren en distintas *Network Namespaces*.

```
n0obie@n0obie-VirtualBox:~$ sudo ps aux | grep mininet
n0obie    2868  0.0  0.0  21312  944 pts/l3   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:sl
n0obie@n0obie-VirtualBox:~$ █
```

Figura 2.14: Listado de procesos asociados a Mininet

Si se inspecciona el fichero `/proc/{pid}/ns/net` para cada proceso indicado en la figura 2.14 se podrá ver cuales de ellos están en una *Network Namespace* distinta, en función del valor que tenga el *inode*. Por ejemplo, se va a comprobar 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 2.15: Información relativa al 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:[\([0-9]+\)]'
lrwxrwxrwx 1 root root 0 jun 13 13:49 /proc/9520/ns/net -> net:[4026532343]
n0obie@n0obie-VirtualBox:~$
```

Figura 2.16: Información relativa al proceso del Host2

Como se puede ver, *inode* distintos, ficheros distintos, distintas *Network namespaces*. Con esta prueba se puede ver como Mininet hace uso de procesos de bash para sostener las *Network Namespaces* de los nodos que lo requieran.

2.7.3. Mininet CLI

Mininet incluye una Command Line Interface (CLI), la cual puede ser invocada desde el script donde se describe la topología. Dicha CLI, contiene una gran variedad de comandos, por ejemplo listar la red, tirar enlaces, comprobar el estado de las interfaces, abrir una *xterm* a un nodo, etc. A continuación se indica la tabla 2.2, la cual resume todos los comandos existentes a día de hoy, para una explicación de cada uno de ellos en detalle se recomienda seguir esta [guía](#)¹¹.

EOF	gterm	links	pingallfull	py	stop
distance	help	net	pingpair	quit	switch
dpctl	intfs	nodes	pingpairfull	sh	time
dump	iperf	noecho	ports	source	x
exit	iperfudp	pingall	px	start	xterm

Tabla 2.2: Resumen comandos existentes en Mininet

2.7.4. Mininet-WiFi

Mininet-WiFi [37] 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

¹¹<https://github.com/mininet/mininet/wiki/Introduction-to-Mininet>

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.

2.7.5. Mininet-IoT

La herramienta Mininet-IoT [38] 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 enfocado a sensores de baja capacidad. Este sistema operativo fue desarrollado por Adam Dunkels con la ayuda de Bjorn Gronvall y Thiemo Voigt en el año 2002. Desde entonces hasta los últimos años, el proyecto Contiki ha involucrado tanto a empresas como a cientos de colaboradores en su repositorio de GitHub¹². Contiki estaba diseñado con el propósito de ofrecer a los nodos de las redes Wireless Sensor Networks (WSN) un sistema operativo ligero con capacidad de carga y descarga de servicios únicos de forma dinámica [39].

El Kernel de Contiki está orientado a eventos, y soporta tareas multi-hilo con requisa.

¹²<https://github.com/contiki-os/contiki>

Contiki está escrito en el lenguaje C y ha sido portado a numerosas de arquitecturas de microcontroladores, como el MSP430 de Texas Instruments y derivados.

En un sistema que ejecute el sistema operativo de Contiki, éste estará dividido en dos partes claramente diferenciadas según se puede ver en la figura 2.17, el *core* y los programas o servicios cargados. El particionado se lleva a cabo en el momento de la compilación y es independiente de cada target en el que se vaya a desplegar el sistema. El *core* consiste en el propio Kernel, un conjunto de servicios de base (*timers, handlers*), , librerías, drivers y el *stack* de comunicación. Los programas o servicios cargados se mapearán en memoria por el propio cargador que tiene el Kernel en tiempo de ejecución.

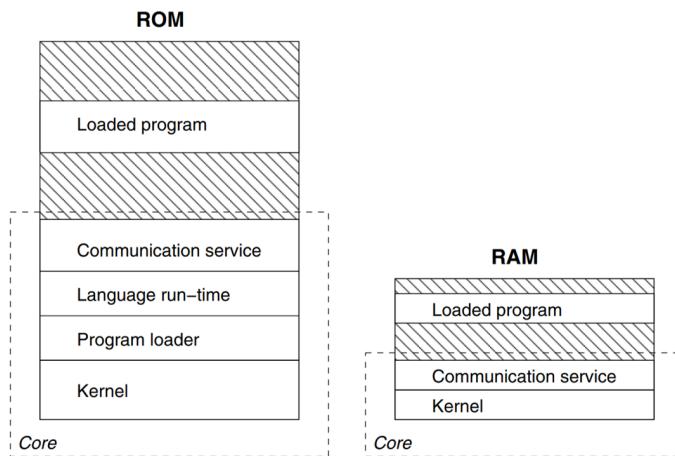


Figura 2.17: Particionado en un sistema con Contiki OS [39]

En los últimos años apareció un nuevo proyecto, **Contiki-ng**¹³, el cual es fork de Contiki OS. Este nuevo proyecto desbanaría a su predecesor bajo el eslogan de “*Contiki-NG: The OS for Next Generation IoT Devices*”. Actualmente toda la comunidad de Contiki está enfocada en este nuevo proyecto, el cual proporciona un *stack* de comunicación más cercano a las RFCs, soporte a protocolos como IPv6/6LoWPAN y 6TiSCH, y por lo que se está haciendo más popular, dar soporte a microcontroladores con la arquitectura ARM [40].

2.8.1. Simulador Cooja

El flujo de trabajo con Contiki o Contiki-ng, dependerá si se va a trabajar sobre hardware real o si se va a simular los programas desarrollados. Cuando se trabaja sobre hardware real,

¹³<https://github.com/contiki-ng/contiki-ng>

el flujo de trabajo consistirá en la compilación del sistema operativo y de los programas desarrollados con el target donde se vaya a trabajar, generando así un binario el cual se podrá instanciar en el disco del hardware.

Si por el contrario se va a simular, se hará uso del simulador llamado Cooja¹⁴. Cooja es una simulador escrito en Java que permite simular una serie de motas IoT. Por tanto, a la hora de simular se podrá ver el comportamiento del programa desarrollado en distintas plataformas.

Todo el proceso de compilación del *core* de Contiki y programas desarrollados está integrado en el propio simulador, permitiendo al usuario compilar sus programas hacia distintos tipos de motas. Cada simulación se podrá almacenar en un fichero con la extensión `*.csc`, los cuales almacenarán todos los datos de la simulación, *seed*, posiciones y tipos de motas en una estructura XML.

2.9. Contribuciones en GitHub

La herramienta GitHub [41] 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 Git¹⁵, 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 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.

¹⁴<https://github.com/contiki-ng/cooja/tree/master>

¹⁵<https://git-scm.com/>

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 2.3 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 2.3: 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. Grossete, 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] “Traffic control - debian,” <https://wiki.debian.org/TrafficControl>, accessed: 2020-06-30.
 - [32] M. Kerrisk, *Namespaces (7) - overview of Linux namespaces*, The Linux man-pages project, May 2020.
 - [33] Michael Kerrisk, *Network namespaces - overview of Linux network namespaces*, The Linux man-pages project, June 2020.
-

- [34] M. Kerrisk, *veth - Virtual Ethernet Device*, The Linux man-pages project, June 2020.
- [35] 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.
- [36] B. Heller, “Reproducible network research with high-fidelity emulation,” Ph.D. dissertation, Stanford University, 2013.
- [37] 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.
- [38] “Mininet-iot,” <https://github.com/ramonfontes/mininet-iot>, accessed: 2020-07-01.
- [39] A. Dunkels, B. Grönvall, 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.
- [40] A. Kurniawan, *Practical Contiki-NG: Programming for Wireless Sensor Networks*. Apress, 2018.
- [41] 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.
BPF	Berkeley Packet Filter.
CLI	Command Line Interface.
ELF	Executable and Linkable Format.
eMBB	enhanced Mobile BroadBand.
FIFO	First In, First Out.
GUI	Graphical User Interface.
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.
P4	Programming Protocol-independent Packet Processors.
Qdiscs	Queueing discipline.
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.
WSN	Wireless Sensor Networks.
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+-.
  .osssssssssssssssssdMMMyssssso.
  /sssssssssshdmmNNmmyNMMMHssssss/
  +ssssssssshnydMMMMMMMddddyssssss+.
  /sssssssssshdmmNNmhyhyyyyhmNMMNhssssss/
  .ssssssssdMMMNhsssssssssshNMMDssssss.
  +sssshhhyNMMNyssssssssssyNMMMyssssss+.
  ossyNMMNyMMhsssssssssssshmhhssssss
  ossyNMMNyMMhsssssssssssshmhhssssss
  +sssshhhyNMMNyssssssssssyNMMMyssssss+.
  .ssssssssdMMMNhsssssssssshNMMDssssss.
  /sssssssssshdmmNNmhyhyyyyhdNMMNhssssss/
  +sssssssssdnydMMMMMMMddddyssssss+.
  /sssssssssshdmmNNNmyNMMMHssssss/
  .osssssssssssssssssdMMMyssssso.
  +ssssssssssssssssyyssss+-.
  `:+ssssssssssssssssss+`:
  .-/+oossssoo+/-.

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

Figura A.1: Especificaciones de la máquina A

```
arppath@david:~$ neofetch
  .-/+oossssoo+/-.
  `:+ssssssssssssssssss+`:
  .+ssssssssssssssssyyssss+-.
  .osssssssssssssssdMMMyssssso.
  /sssssssssshdmmNNmmyNMMMHssssss/
  +ssssssssshnydMMMMMMMddddyssssss+.
  /sssssssssshdmmNNmhyhyyyyhmNMMNhssssss/
  .ssssssssdMMMNhsssssssssshmhhssssss.
  +sssshhhyNMMNyssssssssssyNMMMyssssss+.
  ossyNMMNyMMhsssssssssshmhhssssss
  ossyNMMNyMMhsssssssssssshmhhssssss
  +sssshhhyNMMNyssssssssssyNMMMyssssss+.
  .ssssssssdMMMNhsssssssssshmhhssssss.
  /sssssssssshdmmNNmhyhyyyyhdNMMNhssssss/
  +sssssssssdnydMMMMMMMddddyssssss+.
  /sssssssssshdmmNNNmyNMMMHssssss/
  .osssssssssssssssdMMMyssssso.
  +ssssssssssssssssyyssss+-.
  `:+ssssssssssssssss+`:
  .-/+oossssoo+/-.

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/
      +ssssssssshmydMMMMMMNmdddyssssss+
      /ssssssshNMNMMyhhyyyhNmNMNMNhssssss/
      .ssssssssdMMMNhssssssssshNMNMdssssss.
      +sssshhhyNMNMysssssssssyNMNMysssss+
      ossyNMMNMymMhsssssssssssshmnhssssso
      ossyNMMNMymMhsssssssssssshmnhssssso
      +sssshhhyNMNMysssssssssyNMNMysssss+
      .ssssssssdMMMNhssssssssshNMNMdssssss.
      /ssssssshNMNMMyhhyyyhdNMNMNhssssss/
      +ssssssssdmymdMMMMMMMdddyssssss+
      /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
```

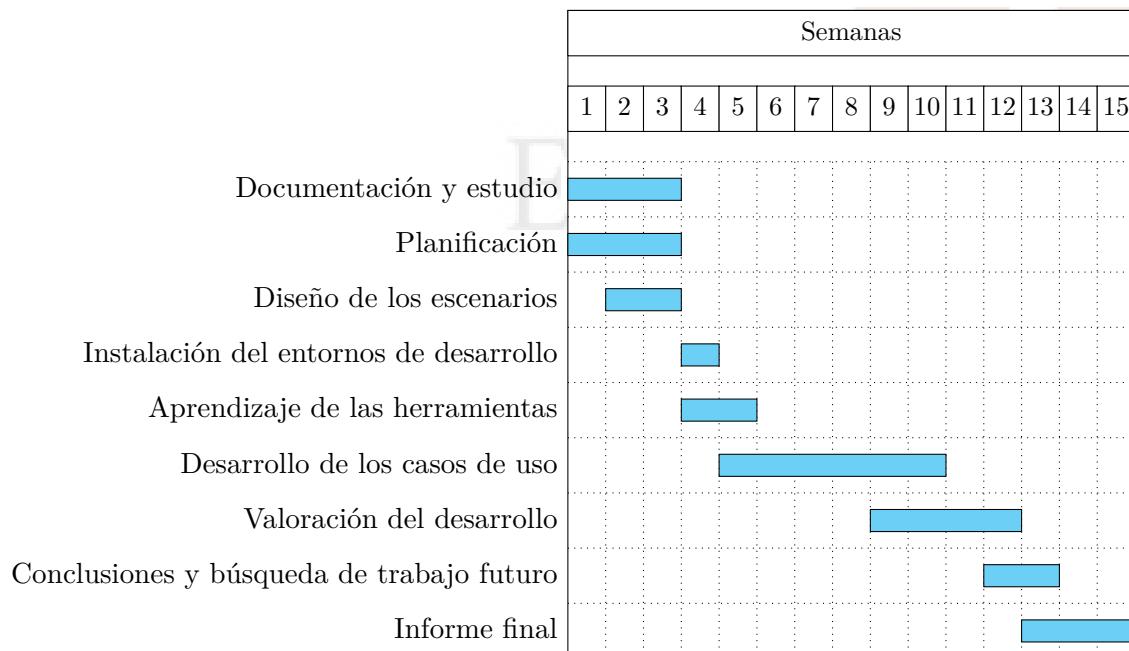
Figura A.3: Especificaciones de la máquina C

B. Anexo II - Presupuesto

En este anexo se expondrá de una forma detallada el presupuesto del proyecto. Para que este presupuesto sea lo más próximo a la realidad, se hará un breve análisis sobre la duración del proyecto. De esta forma, se podrán calcular la mano de obra con mayor exactitud.

B.1. Duración del proyecto

Con el propósito de obtener el número de horas de trabajo por semana del proyecto en **promedio**, se va a realizar un diagrama de Gantt. De este modo se podrá apreciar la distribución de tareas a lo largo del TFG y así ser capaces de obtener un número de horas de trabajo aproximado por semana.



Número de horas totales	Horas por semana	Horas diarias
425h	≈ 28h	≈ 4.2h

Tabla B.1: Promedio de horas de trabajo

B.2. Costes del proyecto

El cálculo de los costes del proyecto se va a realizar diferenciando previamente por *Hardware*, *Software* y mano de obra. De esta manera, se pretende que los costes se desglosen aportando claridad sobre la cuantía total.

Producto (IVA incluido)	Valor (€)
Ordenador portátil Lenovo Legion	1349,00
Ordenador de sobremesa	1450,00
Pantalla Lenovo L27i	129,99
Pantalla Benq 21"	79,89
Periféricos	150,00
Infraestructura de Red (PLCs y Router Livebox)	70,00

Tabla B.2: Presupuesto desglosado del Hardware

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 TFG.

Producto (IVA incluido)	Valor (€)
Microsoft Office	300,00
Adobe Photoshop y Adobe Premiere Pro	241,96

Tabla B.3: Presupuesto desglosado del Software

Se han tomado de referencia los honorarios de un ingeniero junior, los cuales corresponden a 20€ la hora. Los costes del *hardware* y *software* se han agregado como un único elemento, añadiéndolo al presupuesto con el valor total del desglose de los productos indicados.

Descripción (IVA incluido)	Unidades	Coste unitario (€)	Coste total (€)
Material Hardware	1	3228,89	3228,89
Material Software	1	541,96	541,96
Mano de obra	425	20,00	8500,00
Costes fijos (Luz, Internet)	4	76,00	304,00
TOTAL			12.574,855 €

Tabla B.4: Presupuesto total con IVA

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/>

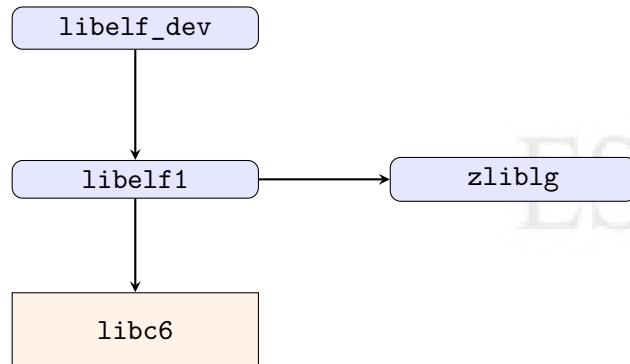


Figura C.1: Ramificación de dependencias de Iproute2.

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

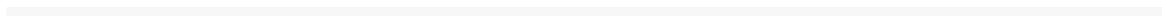
A continuación, se indican los comandos más frecuentes con la herramienta iproute2. Todos ellos han sido utilizados en el proceso de desarrollo del proyecto y en el proceso de verificación de los distintos casos de uso. Por ello, se considera que este apartado puede ser de gran utilidad para el lector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

