

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



Universidad
de Alcalá

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 mi pareja, Yuliia, a mi familia y a mis amigos,
quienes día a día han inculcado en mí
el ejemplo del esfuerzo, trabajo y superación.*

Agradecimientos

Este trabajo no habría sido posible sin el apoyo y el estímulo de mis tutores, Elisa y Joaquín, que compartieron su destreza y conocimiento desde el primer día que les conocí. Quienes, bajo su supervisión, me han enseñado lo entretenido y bonito que es el mundo de las Redes.

También me gustaría agradecer a mi profesor de Sistemas Operativos, Juan Ignacio García Tejedor, cuya maestría y pasión por lo que hace, y a lo que se dedica, me hicieron aprender y disfrutar cada día que iba sus clases. Es más, aún habiendo terminado la asignatura me ayudó en uno de los momentos más difíciles de este trabajo, arrojando luz y aplanando el duro camino al Kernel de Linux. No me podía olvidar de mis amigos, Rubén y Laura, con los que he compartido momentos de estrés y tardes de risas, y de Pablo, a quien lo he conocido casi al final de la carrera, pero con el que comparto un montón de ratos y pasiones.

No puedo terminar sin agradecer a mis compañeros del Laboratorio LE34, quienes me han enseñado a ver los problemas con perspectiva, en ocasiones, a priorizar según el caso. Me han alegrado cada semana de estos dos últimos años con su apoyo y compañía, y me han alentado a mejorar cada día, marcándome el camino en un futuro incierto.

Sinceramente, mil gracias a todos.

Resumen

Este Trabajo Final de Grado (TFG) se enfoca en el estudio de las distintas tecnologías disponibles para la definición del denominado *datapath*, con el objetivo de conseguir una integración de dispositivos Internet of Things (IoT) en entornos de Software-Defined Networking (SDN).

Las dos principales tecnologías que se han analizado para la definición de datapaths son eXpress Data Path (XDP) y el lenguaje P4. A su vez, se han planteado casos de uso donde se quiere demostrar qué ciertas funcionalidades, que debe tener un nodo IoT, pueden ser implementadas. Cuando los casos de uso se realicen, se justificará qué tecnología es más viable en la definición de los datapaths para alcanzar la integración de nodos IoT en entornos SDN.

Palabras clave: IoT; SDN; Lenguaje P4; XDP; Plano de datos

Abstract

This Bachelor's Degree Final Project is focused on the study of the different technologies available for the definition of the so-called datapaths, with the aim of achieving an integration of Internet of Things (IoT) devices in Software-Defined Networking (SDN) environments.

The two main technologies that have been analyzed for the definition of datapaths are eXpress Data Path (XDP) and the P4 language. At the same time, a set of use cases has been presented to demonstrate that certain functionalities that an IoT node must have can be implemented. When the use cases are carried out, it will be justified which technology is more viable in the definition of the datapaths to achieve the integration of IoT nodes in SDN environments.

Keywords: IoT; SDN; P4 Language; XDP; Datapaths

“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. Redes SDN y dispositivos IoT	1
1.2. Tecnologías para la definición del datapath	3
1.3. Objetivos	4
1.4. Estructura del TFG	5
2. Estado del arte	7
2.1. Tecnología SDN	7
2.1.1. Arquitectura	7
2.1.2. OpenFlow	8
2.2. Tecnología IoT	9
2.2.1. Arquitectura	9
2.2.2. Topologías	10
2.2.3. Redes LLN	11
2.2.3.1. IEEE 802.15.4	11
2.3. Tecnología P4	12
2.3.1. Objetivos	13
2.3.2. Arquitectura y targets	14
2.4. Tecnología XDP	15
2.4.1. Modos de operación	16
2.4.2. Procesamiento de paquetes	16
2.4.3. Limitaciones con XDP	17
2.4.3.1. Accesos a memoria y comprobación de límites	17
2.4.3.2. Bucles y funciones inline	18
2.5. Linux Networking	19
2.5.1. Estructura <code>sk_buff</code>	19
2.5.2. Herramienta TC	22
2.5.2.1. Qdiscs	22
2.5.2.2. Classes	22
2.5.2.3. Filters	22
2.5.3. Namespaces	23
2.5.3.1. Persistencia de las Namespaces	23
2.5.3.2. Concepto de las Network Namespaces	24
2.5.3.3. Métodos de comunicación inter-Namespace: Veth	25

2.6.	Mininet y Mininet-WiFi	26
2.6.1.	Mininet	26
2.6.2.	Funcionamiento de Mininet	28
2.6.3.	Mininet CLI	31
2.6.4.	Mininet-WiFi	31
2.6.5.	Mininet-IoT	32
2.7.	Contiki-ng	32
2.7.1.	Simulador Cooja	33
2.8.	Contribuciones en GitHub	34
3.	Diseño y análisis de casos de uso	35
3.1.	Funcionalidades básicas	35
3.2.	Plataformas de pruebas alámbricas	37
3.3.	Plataformas de pruebas inalámbricas	38
4.	Desarrollo y evaluación de casos de uso	41
4.1.	Casos de uso XDP en medios cableados	41
4.1.1.	Case01 - Drop	42
4.1.2.	Case02 - Pass	46
4.1.3.	Case03 - Echo server	50
4.1.4.	Case04 - Layer 3 forwarding	55
4.1.4.1.	Hardcoded forwarding	57
4.1.4.2.	Semi-Hardcoded forwarding (BPF maps)	59
4.1.4.3.	Forwarding auto (Kernel FIBs)	62
4.1.5.	Case05 - Broadcast	64
4.2.	Casos de uso P4 en medios cableados	71
4.2.1.	Case01 - Drop	72
4.2.2.	Case02 - Pass	76
4.2.3.	Case03 - Echo server	77
4.2.4.	Case04 - Layer 3 forwarding	82
4.2.5.	Case05 - Broadcast	88
4.3.	Casos de uso P4 en medios inalámbricos	93
4.3.1.	Integración del BMv2 en Mininet-WiFi	94
4.3.1.1.	Análisis de la interfaz BMv2 - Mininet	94
4.3.1.2.	Análisis del funcionamiento interno de Mininet-WiFi	96
4.3.1.3.	Desarrollo de la interfaz BMV2 - Mininet-WiFi	101
4.3.2.	Case01 - Drop	102
4.3.3.	Case02 - Pass	106
4.3.4.	Case03 - Echo server	106
4.3.5.	Case04 - Layer 3 forwarding	109
4.3.6.	Case05 - Broadcast	111
4.4.	Casos de uso XDP en medios inalámbricos	116
4.4.1.	Case01 - Drop	117
4.4.2.	Case02 - Pass	120
4.4.3.	Case03 - Echo server	123

4.4.4.	Case04 - Layer 3 forwarding	127
4.4.4.1.	Hardcoded forwarding	129
4.4.4.2.	Semi-Hardcoded forwarding (BPF maps)	132
4.4.4.3.	Forwarding auto (Kernel FIBs)	134
4.4.5.	Case05 - Broadcast	134
5.	Conclusiones y trabajo futuro	139
5.1.	Conclusiones del TFG	139
5.2.	Líneas de trabajo futuro	141
5.2.1.	Integración de interfaces en modo monitor con los casos de uso	141
5.2.1.1.	Inyección de paquetes en interfaz en modo monitor	143
5.2.2.	Emulación de redes de baja capacidad - mac802154_hwsim	146
5.2.2.1.	Herramientas para interactuar con el stack ieee802154	146
5.2.2.2.	Arquitectura del stack ieee802154	146
5.2.2.3.	Viabilidad de los casos de uso	148
5.2.2.4.	Carga de programas XDP en interfaces ieee802154	149
A.	Anexo I - Pliego de condiciones	153
A.1.	Condiciones materiales y equipos	153
A.1.1.	Especificaciones Máquina A	153
A.1.2.	Especificaciones Máquina B	153
A.1.3.	Especificaciones máquinas virtuales	153
A.1.3.1.	Máquina virtual entorno P4	154
A.1.3.2.	Máquina virtual entorno XDP	155
B.	Anexo II - Presupuesto	157
B.1.	Duración del proyecto	157
B.2.	Costes del proyecto	158
C.	Anexo III - Manuales de usuario e Instalación	159
C.1.	Instalación de dependencias de los casos de uso	159
C.1.1.	Instalación de dependencias máquina XDP	159
C.1.2.	Instalación de dependencias máquina P4	160
C.2.	Herramienta iproute2	161
C.2.1.	¿Qué es iproute2?	161
C.2.2.	¿Por qué necesitamos iproute2?	161
C.2.3.	Estudio de compatibilidad de la herramienta iproute2 en Ubuntu	162
C.2.4.	Compilación e instalación de iproute2	163
C.2.4.1.	Diferencias con Ubuntu 18.04	164
C.2.5.	Comandos útiles con iproute2	164
C.3.	Herramienta tcpdump	165
C.3.1.	¿Qué es tcpdump?	165
C.3.2.	¿Por qué necesitamos tcpdump?	165
C.3.3.	Instalación de tcpdump	165
C.3.4.	Comandos útiles con tcpdump	166

Índice de figuras

1.1. Paradigma en redes SDN	1
1.2. Integración parcial de dispositivos IoT en el core SDN	2
1.3. Integración total de dispositivos IoT en el core SDN	3
2.1. Arquitectura básica SDN	8
2.2. Arquitectura básica IoT	10
2.3. Tipos de topología con dispositivos IoT [?]	10
2.4. Pila de protocolos 6LoWPAN [?]	11
2.5. Ecosistema P4	12
2.6. Cambio de paradigma con la tecnología P4 [?]	13
2.7. Arquitectura de la tecnología P4 [?]	14
2.8. Procesamiento de paquetes con XDP	17
2.9. Sistema de colas doblemente enlazada	20
2.10. Punteros de la estructura sk_buff	21
2.11. Sistema de QoS implementado con distintas clases [?]	23
2.12. Enlace entre interfaces Veth separadas en dos Network Namespaces	26
2.13. Arquitectura de Mininet [?]	28
2.14. Topología de ejemplo levantada	29
2.15. Listado de Network Namespaces existentes en el sistema	29
2.16. Listado de procesos asociados a Mininet	30
2.17. Información relativa al proceso del Host1	30
2.18. Información relativa al proceso del Host2	31
2.19. Particionado en un sistema con Contiki OS [?]	33
3.1. Ejemplo de carga de un programa XDP sobre una interfaz Ethernet virtual	36
3.2. Ejemplo de carga de un programa P4 sobre el BMV2	37
3.3. Grupos de trabajo IEEE 802	38
4.1. Escenario cableado del Case01 - XDP	44
4.2. Comprobación de funcionamiento del Case01 - XDP	45
4.3. Escenario cableado del Case02 - XDP	48
4.4. Comprobación de funcionamiento del Case02 - XDP	50
4.5. Escenario cableado del Case03 - XDP	53
4.6. Comprobación de funcionamiento del Case03 - XDP	54
4.7. Ping a máquina inexistente - XDP	55
4.8. Escenario cableado Hardcoded forwarding del Case04 - XDP	57
4.9. Comprobación de funcionamiento Hardcoded forwarding del Case04 - XDP	59
4.10. Escenario cableado Semi-Hardcoded forwarding del Case04 - XDP	60
4.11. Comprobación de funcionamiento Semi-Hardcoded forwarding del Case04 - XDP	61

4.12. Escenario cableado Forwarding auto del Case04 - XDP	62
4.13. Comprobación de funcionamiento Forwarding auto del Case04 - XDP	64
4.14. Escenario a recrear del Case05 - XDP	65
4.15. Escenario propuesto del Case05 - XDP	66
4.16. Solución propuesta para el Case05 - XDP	67
4.17. Escenario del Case05 - XDP	68
4.18. Comprobación de funcionamiento del Case05 - XDP	70
4.19. Proceso de compilación programa P4	73
4.20. Proceso puesta en marcha de un switch BMv2 [?]	73
4.21. Escenario del Case01 - P4	74
4.22. Comprobación de funcionamiento (Ping) del Case01 - P4	75
4.23. Comprobación de funcionamiento (Sniffer) del Case01 - P4	76
4.24. Escenario del Case03 - P4	81
4.25. Comprobación de funcionamiento del Case03 - P4	82
4.26. Funcionamiento de las tablas en P4 [?]	84
4.27. Escenario del Case04 - P4	86
4.28. Comprobación de funcionamiento del Case04 - P4	87
4.29. Escenario del Case05 - P4	90
4.30. Comprobación de funcionamiento del Case05 - P4	92
4.31. UML del punto de entrada de la interfaz BMv2 - Mininet	95
4.32. UML interfaz BMv2 - Mininet	95
4.33. Arquitectura Mininet-WiFi [?]	96
4.34. UML sobre las relaciones de las clases de tipo Nodo.	97
4.35. UML sobre las relaciones de las clases de tipo Interfaz.	97
4.36. Arquitectura del subsistema Wireless de Linux [?]	98
4.37. Flujo para la transmisión con el módulo mac80211_hwsim [?]	99
4.38. Flujo para la recepción con el módulo mac80211_hwsim [?]	100
4.39. UML de la integración BMv2 - Mininet-WiFi	101
4.40. Escenario del Case01 - P4 Wireless	103
4.41. Comprobación de funcionamiento (Ping) del Case01 - P4 Wireless	105
4.42. Comprobación de funcionamiento (Sniffer) del Case01 - P4 Wireless	106
4.43. Escenario del Case03 - P4 Wireless	107
4.44. Comprobación de funcionamiento del Case03 - P4 Wireless	109
4.45. Escenario del Case04 - P4 Wireless	110
4.46. Comprobación de funcionamiento del Case04 - P4 Wireless	112
4.47. Escenario del Case05 - P4 Wireless	113
4.48. Comprobación de funcionamiento del Case05 - P4 Wireless	115
4.49. Escenario inalámbrico del Case01 - XDP	118
4.50. Comprobación de funcionamiento del Case01 - XDP Wireless	119
4.51. Escenario inalámbrico del Case02 - XDP	121
4.52. Comprobación de funcionamiento (Ping) del Case02 - XDP Wireless	122
4.53. Comprobación de funcionamiento (Stats) del Case02 - XDP Wireless	123
4.54. Escenario inalámbrico del Case03 - XDP	125
4.55. Comprobación de funcionamiento (Ping) del Case03 - XDP Wireless	126
4.56. Comprobación de funcionamiento (Stats) del Case03 - XDP Wireless	127

4.57. Escenario inalámbrico Hardcoded forwarding del Case04 - XDP	129
4.58. Ping Hardcoded forwarding del Case04 - XDP Wireless	131
4.59. Sniffer Hardcoded forwarding del Case04 - XDP Wireless	131
4.60. Escenario inalámbrico Semi-Hardcoded forwarding del Case04 - XDP Wireless	132
4.61. Ping Semi-Hardcoded forwarding del Case04 - XDP Wireless	133
4.62. Stats Semi-Hardcoded forwarding del Case04 - XDP Wireless	134
4.63. Escenario inalámbrico del Case05 - XDP	135
4.64. Comprobación de funcionamiento del Case05 - XDP Wireless	137
5.1. Verificación de la creación de la interfaz en modo monitor.	144
5.2. Generación de ping vía interfaz en modo monitor.	144
5.3. Comprobación de las cabeceras del ping generado vía interfaz en modo monitor.	145
5.4. Arquitectura del stack <code>ieee802154 [?]</code>	147
A.1. Especificaciones de la máquina virtual P4	154
A.2. Especificaciones de la máquina virtual XDP	155
C.1. Ramificación de dependencias de Iproute2.	163
C.2. Interfaz CLI de Tcpdump	166

Índice de tablas

2.1. Resumen modos de operación en XDP	16
2.2. Resumen de los tipos de Namespaces en el Kernel de Linux	24
2.3. Resumen comandos existentes en Mininet	31
2.4. Resumen de contribuciones realizadas	34
4.1. Resumen de la documentación sobre los casos de uso XDP en entornos cableados	42
4.2. Resumen sobre los códigos de retorno XDP	51
4.3. Estructuras de datos para procesar las cabeceras de los paquetes	51
4.4. Resumen de la documentación sobre los casos de uso P4 en entornos cableados	71
4.5. Resumen de la documentación sobre los casos de uso P4 en entornos inalámbricos	93
4.6. Resumen de las versiones requeridas de la interfaz BMv2 - Mininet-WiFi . . .	103
4.7. Resumen de la documentación sobre los casos de uso XDP en entornos inalámbricos	116
5.1. Resumen sobre los casos de uso desarrollados	139
5.2. Resumen de las herramientas del entorno ieee802154	146
B.1. Promedio de horas de trabajo	157
B.2. Presupuesto desglosado del Hardware	158
B.3. Presupuesto desglosado del Software	158
B.4. Presupuesto total con IVA	158
C.1. Comparativa de herramientas Iproute2 con paquete net-tools	161
C.2. Estudio de compatibilidad de la herramienta Iproute2	162

Índice de Códigos

2.1. Definición de códigos de retorno XDP	16
2.2. Comprobación de límites en XDP	17
2.3. Loops Unrolling	18
2.4. Estructura sk_buff_head	20
2.5. Comandos útiles con iproute2 - Netns	25
2.6. Manejo de Veths	25
2.7. Levantamiento de la topología de ejemplo	28
2.8. Listar Network Namespaces	29
4.1. Programa básico XDP - Case01	42
4.2. Compilación programa XDP - Case01	43
4.3. Puesta en marcha del escenario - Case01	43
4.4. Carga del programa XDP - Case01	44
4.5. Programa básico XDP - Case01	46
4.6. Compilación programa XDP - Case02	46
4.7. Puesta en marcha del escenario - Case02	47
4.8. Carga del programa XDP - Case02	48
4.9. Compilación programa XDP - Case03	51
4.10. Puesta en marcha del escenario - Case03	52
4.11. Carga del programa XDP - Case03	53
4.12. Helper BPF para realizar Forwarding - Case04	56
4.13. Compilación programa XDP - Case04	56
4.14. Puesta en marcha del escenario - Case04	56
4.15. Ejemplo MAC e Ifindex - Case04	58
4.16. Carga del programa XDP Hardcoded forwarding - Case04	58
4.17. Comprobación del funcionamiento Hardcoded forwarding - Case04	58
4.18. Carga del programa XDP Semi-Hardcoded forwarding - Case04	60
4.19. Carga del programa XDP Forwarding auto - Case04	63
4.20. Comprobación del funcionamiento Forwarding auto - Case04	63
4.21. Helper BPF para realizar un Broadcast - Case05	65
4.22. Compilación programa XDP - Case05	67
4.23. Puesta en marcha del escenario - Case05	68
4.24. Carga del programa XDP - Case05	69
4.25. Comprobación del funcionamiento - Case05	69
4.26. Compilación programa P4 y puesta en marcha del escenario - Case01	74
4.27. Limpieza del escenario P4 - Case01	74
4.28. Limpieza segura del escenario P4 - Case01	74
4.29. Comprobación de funcionamiento - Case01	75
4.30. Comprobación de funcionamiento - Case01	76

4.31. Estructura cabecera ICMP - Case03	77
4.32. Estructura cabeceras IPv6 e ICMPv6 - Case03	77
4.33. Macros para parsear L2 y L3 - Case03	78
4.34. Lógica para filtrar paquetes ICMP e ICMPv6 - Case03	79
4.35. Lógica para procesar paquetes ICMP - Case03	79
4.36. Compilación programa P4 y puesta en marcha del escenario - Case03	80
4.37. Limpieza del escenario P4 - Case03	80
4.38. Limpieza segura del escenario P4 - Case03	81
4.39. Comprobación de funcionamiento - Case03	82
4.40. Redirección del paquete - Case04	83
4.41. Ejecución de instancia del BMv2 - Case04	83
4.42. Acción propuesta para llevar a cabo el forwarding - Case04	84
4.43. Compilación programa P4 y puesta en marcha del escenario - Case04	85
4.44. Limpieza del escenario P4 - Case04	85
4.45. Limpieza segura del escenario P4 - Case04	85
4.46. Ejemplo Json Grupo Multicast - Case05	88
4.47. Acción propuesta para llevar a cabo el Broadcast - Case05	88
4.48. Acción propuesta para descartar paquetes sobrantes de Broadcast - Case05	89
4.49. Compilación programa P4 y puesta en marcha del escenario - Case05	89
4.50. Limpieza del escenario P4 - Case05	90
4.51. Limpieza segura del escenario P4 - Case04	90
4.52. Apertura de terminales - Case05	91
4.53. Puesta en escucha - Case05	91
4.54. Generación de ARP-Request - Case05	91
4.55. Instalación de Mininet-WiFi modificado	102
4.56. Compilación programa P4 - Case01	104
4.57. Puesta en marcha del escenario - Case01	104
4.58. Pasos a seguir para comprobar el funcionamiento - Case01	104
4.59. Compilación programa P4 - Case03	108
4.60. Puesta en marcha del escenario - Case03	108
4.61. Pasos a seguir para comprobar el funcionamiento - Case03	108
4.62. Compilación programa P4 - Case04	110
4.63. Puesta en marcha del escenario - Case04	111
4.64. Pasos a seguir para comprobar el funcionamiento - Case04	111
4.65. Compilación programa P4 - Case05	113
4.66. Puesta en marcha del escenario - Case05	114
4.67. Pasos a seguir para comprobar el funcionamiento - Case05	114
4.68. Compilación programa XDP - Case01	117
4.69. Compilación programa XDP - Case01	117
4.70. Carga del programa XDP - Case01	118
4.71. Compilación programa XDP - Case02	120
4.72. Compilación programa XDP - Case02	121
4.73. Carga del programa XDP - Case02	121
4.74. Comprobación del funcionamiento - Case02	122
4.75. Compilación programa XDP - Case03	124

4.76. Compilación programa XDP - Case03	125
4.77. Carga del programa XDP - Case03	126
4.78. Comprobación del funcionamiento - Case03	126
4.79. Compilación programa XDP - Case04	128
4.80. Compilación programa XDP - Case04	129
4.81. Carga del programa XDP Hardcoded forwarding - Case04	130
4.82. Comprobación del funcionamiento Hardcoded forwarding - Case04	130
4.83. Carga del programa XDP Semi-Hardcoded forwarding - Case04	133
4.84. Comprobación del funcionamiento Semi-Hardcoded forwarding - Case04	133
4.85. Compilación programa XDP - Case05	135
4.86. Compilación programa XDP - Case05	136
4.87. Carga del programa XDP - Case05	136
4.88. Comprobación del funcionamiento - Case05	136
 5.1. Herramienta wping	142
5.2. Ejecución del escenario	143
5.3. Creación de interfaz en modo monitor	143
5.4. Estructura para manejar paquetes de l estándar ieee802154	148
5.5. Instalación de las dependencias de Mininet-WiFi - mac802154_hwsim	149
5.6. Ejecución ejemplo 6LoWPan.py	149
5.7. Compilación del programa XDP - mac802154_hwsim	150
5.8. Obtención de interfaz 802.15.4	150
5.9. Obtención de interfaz 802.15.4 haciendo uso de la herramienta iwpan	150
5.10. Carga de programa XDP en interfaz ieee802154	151
5.11. Comprobación funcionamiento programa XDP en interfaz ieee802154	151
5.12. Comprobación de funcionamiento del programa XDP - 2	152
 C.1. Descarga del repositorio del TFG	159
C.2. Instalación de dependencias XDP	159
C.3. Instalación de dependencias P4	160
C.4. Instalación de las dependencias de Iproute2	163
C.5. Obtención del source de Iproute2	163
C.6. Compilación e instalación de Iproute2	164
C.7. Instalación de las dependencias de Iproute2 - Ubuntu 18.04	164
C.8. Comandos útiles con iproute2	164
C.9. Instalación de Tcpdump	165
C.10. Comandos útiles con Tcpdump	166

1. Introducción

En este capítulo de introducción se quiere exponer de manera breve los conceptos más importantes del TFG, como son el SDN y la tecnología IoT. Se discutirán las ventajas que tiene cada una de ellas, y qué valor añadido genera la integración de ambas. Más adelante, se indicarán las tecnologías con las cuales se podrá definir el denominado *datapath* de los dispositivos IoT, para alcanzar la integración de estos en un entorno SDN.

Con todo ello, se marcarán unos objetivos de este trabajo y cómo se quieren llevar a cabo. Dichos objetivos servirán para prefijar con qué tecnologías es más viable la definición del plano de datos de los dispositivos IoT. Por último, se expondrá como es la estructura general de este trabajo, explicando brevemente de que tratará cada capítulo.

1.1. Redes SDN y dispositivos IoT

El Internet de las Cosas (del inglés *Internet of Things*, IoT) [?] y las Redes Definidas por Software (del inglés *Software-Defined Networking*, SDN) [?], son tecnologías emergentes que desde hace unos años están revolucionando los paradigmas sobre los modelos de redes comunicaciones establecidos en el pasado.

Con las redes SDN, como se puede apreciar en la figura 1.1, se desea extraer el plano de control de los dispositivos intermedios de procesamiento de la red, unificándolo en entes llamados controladores, haciendo la administración de red una tarea más centralizada y flexible. En cuanto al IoT, se pretende interconectar billones de objetos entre sí a través de Internet con la finalidad de que exista una comunicación máquina – máquina para proveer al usuario final de entornos inteligentes.

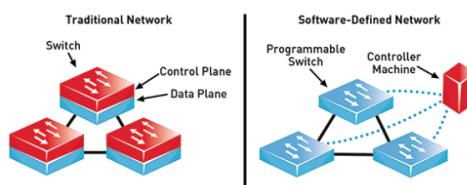


Figura 1.1: Paradigma en redes SDN

Se considera como un punto de convergencia entre ambas tecnologías las redes 5G basadas fundamentalmente en SDN, ya que prevén la incorporación de múltiples dispositivos IoT. Sin embargo, no todos los dispositivos IoT soportan una gestión basada en SDN debido a las limitaciones que estos tienen en memoria, capacidad de procesamiento y batería. Conforme el mundo de los dispositivos IoT avanza, las placas Single Board Computer (SBC) cada vez

se hacen más potentes y de un tamaño más reducido, haciendo posible que éstas puedan ser incluidas como una pieza fundamental en proyectos IoT de alto rendimiento [?].

Además, los sensores IoT, comúnmente conocidas como “motas”, continuamente se fabrican con más capacidad de procesamiento, memoria y batería con la finalidad de hacer frente a las nuevas redes de comunicaciones 5G [?]. Por todo ello, se estima que la integración con las redes SDN está cada vez más próxima. Generalmente, se están siguiendo estas vías de actuación:

- Realizar una integración parcial, haciendo uso de dispositivos mediadores entre las motas y el core SDN. Este dispositivo mediador, sería un dispositivo híbrido con interfaces inalámbricas y alámbricas para tener de acceso a la red SDN [?].

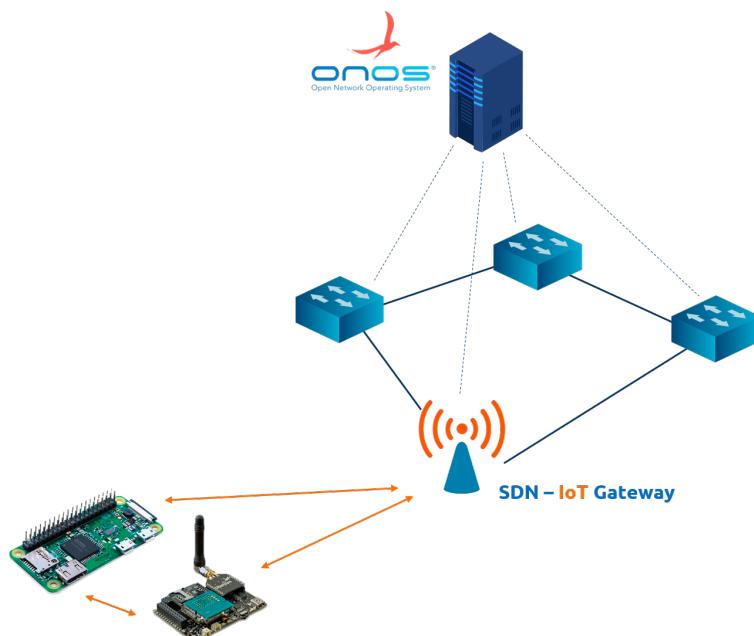


Figura 1.2: Integración parcial de dispositivos IoT en el core SDN

Como se puede ver en la figura 1.2, haciendo uso de un dispositivo mediador, este actuaría de pasarela o *gateway* para las redes de sensores. De esta manera, se delega toda la carga de procesamiento y consumo de batería que supone integrar la interfaz de control SDN. Además, se implementarían los distintos protocolos y estándares para comunicarse con los dispositivos IoT. Esta vía también ofrece otros aspectos positivos, entre los cuales se puede destacar, emplear el gateway como un traductor de protocolos IoT [?]. Esto permitiría que distintos sensores que tuvieran implementados diferentes protocolos como, por ejemplo, Bluetooth Low Energy (BLE)¹ y Zigbee², pudieran interactuar entre sí a través del gateway IoT.

¹Especificación del protocolo: <https://www.bluetooth.com/specifications/>

²Protocolo de alto nivel basado en el estandar Institute of Electrical and Electronics Engineers (IEEE) 802.15.4, especificación: <https://zigbeealliance.org/solution/zigbee/>

- Realizar una integración total, como se puede apreciar en la figura 1.3, donde el propio dispositivo IoT es capaz de co-existir en la red SDN. Esta última vía es la más ambiciosa debido a que se requiere de hardware lo suficientemente potente para soportar la lógica SDN, y de una tecnología que establezca su plano de datos de forma más eficiente posible para optimizar al máximo los recursos limitados del dispositivo.

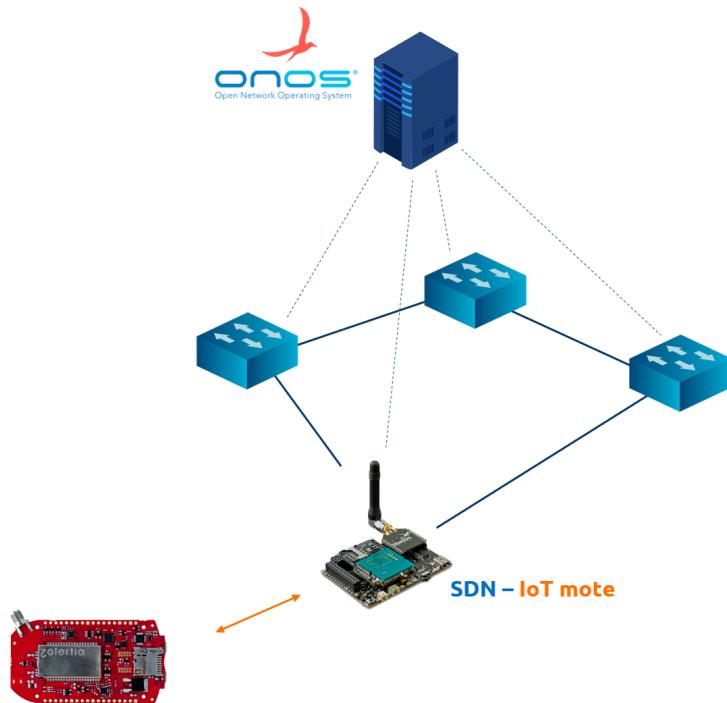


Figura 1.3: Integración total de dispositivos IoT en el core SDN

1.2. Tecnologías para la definición del datapath

En redes convencionales previas al concepto de SDN, los nodos de la red tenían unificado un plano de control, *Control plane*, donde se definía la lógica que dictaba como debía llevarse a cabo el *forwarding* de los paquetes, y un plano de datos, *Data plane*, el cual se puede implementar definiendo su datapath. Dicho datapath se compone de varios bloques de procesado para reenviar los paquetes. Con la llegada del paradigma de las redes SDN, como se puede apreciar en la figura 1.1, los nodos clásicos de red verían como su plano de control sería delegado a una entidad llamada controlador. El controlador tendría una perspectiva global de toda la red en su conjunto pudiendo gestionarla de una manera más flexible y centralizada.

En consecuencia, los nodos de la red estarían destinados a implementar un plano de datos, y una interfaz de control para ser configurados por el controlador mediante protocolos como, por ejemplo, OpenFlow³.

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

Por lo tanto, para ejecutar la integración de los dispositivos IoT en las redes SDN se debe poder definir un datapath en dichos dispositivos y que estos posean una interfaz de control para ser configurados desde un hipotético controlador. Debido a esto, se requiere hacer uso de tecnologías que nos permitan definir el plano de datos de estos dispositivos. En este TFG, se abordarán las siguientes tecnologías:

- El lenguaje **P4**. Este lenguaje de alto nivel nos permite definir el procesado de los paquetes para un conjunto de arquitecturas. Dichas arquitecturas tienen su propia especificación. Con ello, se quiere conseguir que los programas P4 sean independientes del *hardware* donde se ejecute [?].
- **XDP**. Es un *framework* programable y de alto rendimiento para el procesado de paquetes en el Kernel de Linux. El procesado de los paquetes se lleva a cabo en el punto más bajo de la pila de red de Linux, consiguiendo así añadir nuevas funcionalidades sin necesidad de modificar el propio Kernel. Además, el hecho de definir el procesado de paquetes casi en la propia interfaz, reporta un gran rendimiento ya que no es necesario atravesar toda la pila de protocolos, con todo lo que eso conlleva (reservas de memoria, reservas de memoria para metadatos, encolado de los paquetes, desencapsulado de cabeceras, análisis y tratamiento de cabeceras, etc) [?].

1.3. Objetivos

El objetivo de este proyecto es realizar un estudio y análisis de las tecnologías P4 y XDP para la integración de dispositivos IoT en entornos SDN. Como ya hemos introducido, con la llegada del IoT, la dimensión de las redes va a crecer exponencialmente. Por 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, ya que éstas les reportarán la **flexibilidad y programabilidad** requerida para una correcta gestión y administración de cada elemento de la red. Para alcanzar dicho objetivo, se han planteado los siguientes puntos:

- **Documentación y estudio:** Búsqueda y recolección de información, artículos y guías para tener los conocimientos básicos necesarios sobre el estado del arte actual y de las tecnologías para definir el plano de datos. De esta forma, se podrá plantear los diseños y desarrollos más optimizados, de cara a favorecer las limitaciones impuestas por los dispositivos IoT.
- **Planteamiento de casos de uso típicos y elección de escenarios:** Se hará un planteamiento de funcionalidades básicas, casos de uso, que una tecnología que define un datapath debe ser capaz de proveer. Además, se elegirán los escenarios y plataformas para desplegar dichos casos de uso.
- **Desarrollo de los casos de uso:** Se abordará el desarrollo de dichos casos de uso con ambas tecnologías (P4, XDP) en los escenarios planteados. Se tratará que los escenarios donde se desplieguen los desarrollos sean lo más homogéneos posibles, para que los puntos fuertes y débiles de cada tecnología puedan ser esclarecidos.

- **Evaluación y comprobación de funcionamiento:** Se comprobará el correcto funcionamiento de todos los casos de uso desarrollados y por último, se evaluará que tecnología es más idónea para definir el datapath de los dispositivos IoT en su integración en entornos SDN.

1.4. Estructura del TFG

En esta sección se indica la estructura básica de este TFG, haciendo un breve resumen de los aspectos más importantes y significativos de cada capítulo.

Capítulo 1: Introducción. Se hará una breve introducción de la motivación que ha originado la realización de este TFG, así como una breve explicación de los aspectos generales y de los objetivos que se quieren alcanzar con el trabajo presentado.

Capítulo 2: Estado del arte. Se documentarán conceptos fundamentales en relación al proyecto, además de todas las diferentes herramientas que sean utilizadas. La motivación de este capítulo es la de establecer un marco teórico lo suficientemente consistente para abordar el análisis y el diseño de forma óptima, previo al desarrollo.

Capítulo 3: Diseño y análisis de casos de uso. Se debatirá y analizará que funcionalidades básicas deben tener las distintas tecnologías para definir el datapath. Para ello, se diseñarán distintos casos de uso para así demostrar la eficiencia de una tecnología sobre la otra en la integración de los dispositivos IoT en entornos SDN.

Capítulo 4: Desarrollo y evaluación de casos de uso. Se describirá el desarrollo realizado, indicando las partes más importantes de cada caso de uso, ofreciendo al final de cada caso una evaluación de su funcionamiento.

Capítulo 5: Conclusiones y trabajo futuro. Se terminará la memoria con las conclusiones del trabajo realizado, y se presentarán las vías de trabajo 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 IEEE, siguiendo las recomendaciones oficiales de la normativa sobre TFGs 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 TFG. Por último, se hará un presupuesto que incluya el coste de mano de obra, material y gastos generales.

2. Estado del arte

En este capítulo se documentarán los conceptos fundamentales en relación al proyecto, además de todas las diferentes herramientas que sean utilizadas mayoritariamente. La motivación de este capítulo es la de establecer un marco teórico lo suficientemente consistente para abordar el análisis y el diseño de forma óptima, previo al desarrollo. Por último, se mencionará las contribuciones y la documentación generada con la intención de divulgar los contenidos de este proyecto a través de la herramienta *GithHub*.

2.1. Tecnología SDN

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

2.1.1. 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 2.1, 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 comunican 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.

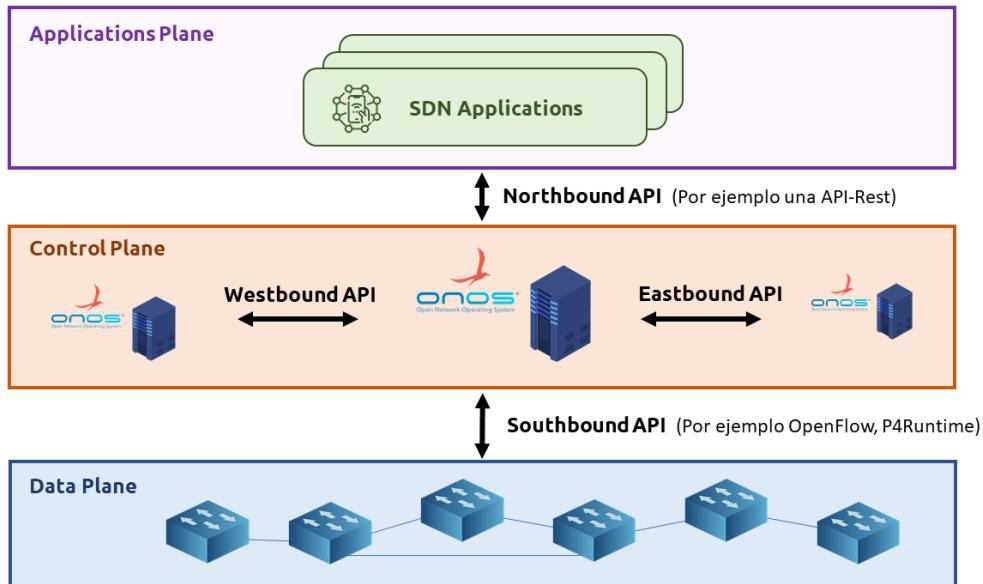


Figura 2.1: Arquitectura básica SDN

2.1.2. 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 ONF¹, contando con numerosas versiones siendo la última la versión 1.5.1 del 2015.

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 cabeecera 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 [?].

¹<https://www.opennetworking.org/software-defined-standards/specifications/>

2.2. Tecnología IoT

La premisa básica de la tecnología IoT [?] 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 [?].

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 generaran [?].

2.2.1. Arquitectura

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

- *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.

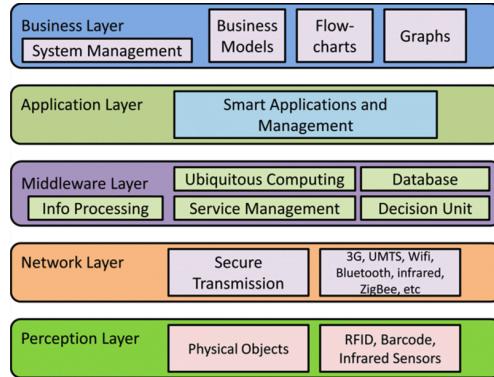


Figura 2.2: Arquitectura básica IoT

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 ve en la figura 2.3. Generalmente se suele hacer uso de un tipo de topología sobre otra en función de las limitaciones de los nodos que la conforman [?].

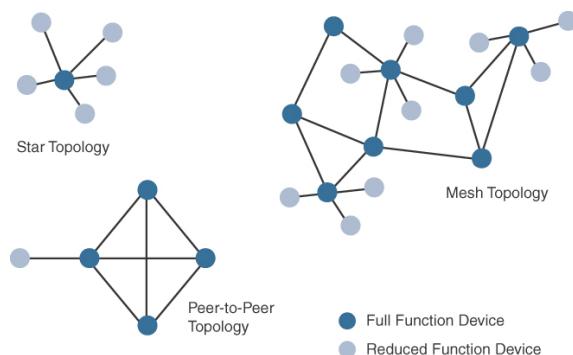


Figura 2.3: Tipos de topología con dispositivos IoT [?]

2.2.3. Redes LLN

Las redes de baja capacidad, conocidas como, Low power and Lossy Networks (LLN)², 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 [?]. 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. 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 [?].

2.2.3.1. IEEE 802.15.4

El estándar ieee802154³ 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 2.4) [?].

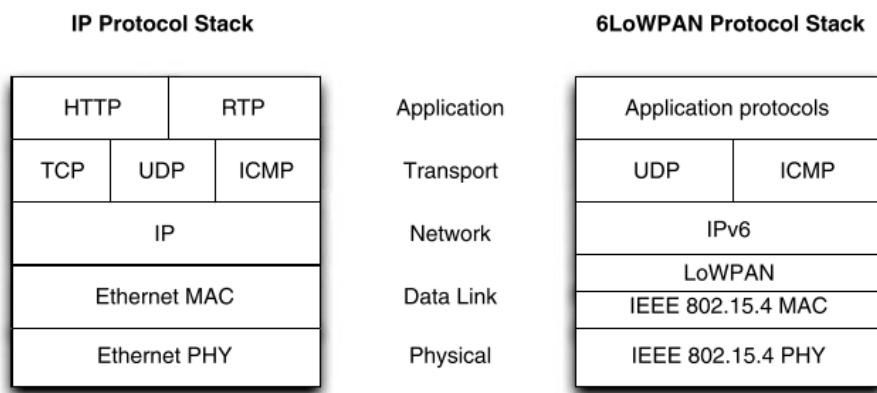


Figura 2.4: Pila de protocolos 6LoWPAN [?]

²<https://tools.ietf.org/html/rfc7228>

³<https://tools.ietf.org/id/draft-ietf-lwig-terminology-05.html>

2.3. Tecnología P4

La tecnología P4⁴ se diseñó principalmente como un lenguaje de alto nivel para programar procesadores de paquetes. Su impulso vino de la mano de un consorcio de empresas privadas relacionadas con el mundo de las telecomunicaciones y fabricantes de hardware, conocido como P4 Language Consortium. Más tarde, conforme la tecnología fue ganando fuerza, el proyecto P4 pasó a ser parte de la ONF ganando así más difusión en la comunidad.

Dicha tecnología nació con el propósito de cubrir las limitaciones del protocolo OpenFlow. El protocolo OpenFlow a través de distintas versiones había aumentado la cantidad de campos de cabeceras, admitiendo así más protocolos, pasando de 12 campos a 41 en cuatro años. Este hecho supuso un aumento en la complejidad de la especificación del protocolo, y aún seguían limitados en dar flexibilidad para añadir nuevas cabeceras [?].

Debido a lo cual, se presentaba P4 como una solución hacia donde debía evolucionar OpenFlow. Para ello, se basaron en tres pilares fundamentales, el primero de ellos era conseguir que los dispositivos de red fuera reconfigurables en vuelo, el segundo consistía en que los switches fueran agnósticos a cualquier protocolo de red, y por último, se quería que el lenguaje P4 fuera completamente independiente del hardware sobre el cual se fuera a implantar.

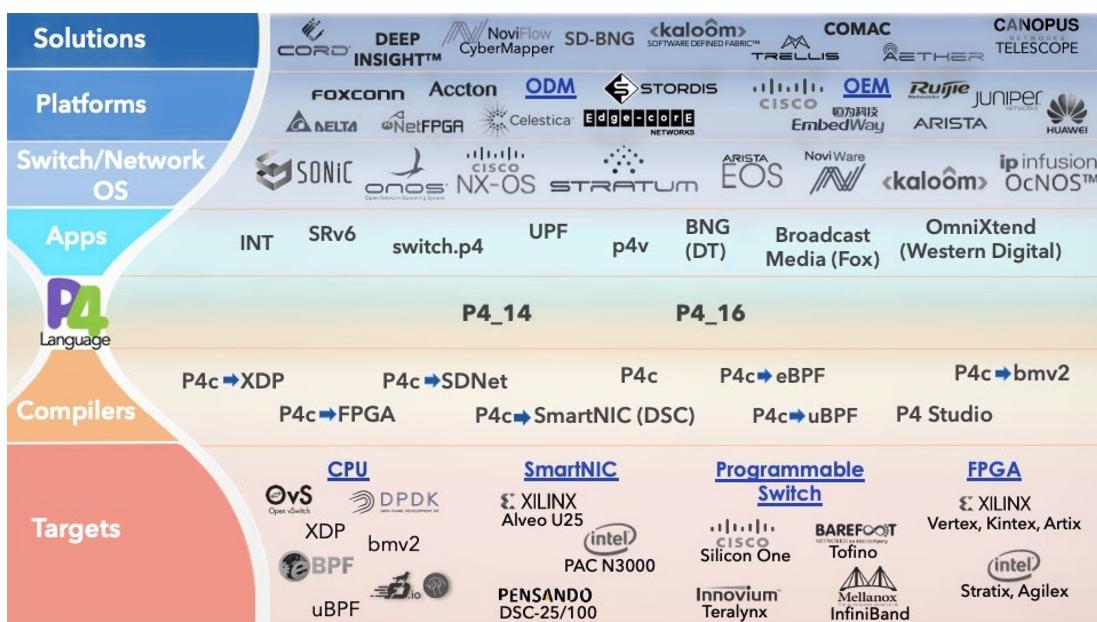


Figura 2.5: Ecosistema P4

A día de hoy la tecnología P4 ya está ampliamente implantada. Como se puede apreciar en la figura 2.5, se ha generado todo un ecosistema entorno a esta tecnología, contando con numerosos compiladores de *backend* enfocados para los distintos *targets*, con aplicaciones y soluciones ya puestas en funcionamiento. Por lo que todo apunta a que este ecosistema tiende

⁴Programming Protocol-independent Packet Processors

a ir a más, si bien es verdad que aún tiene ciertas dificultades con ciertos tipos de hardware, pero su futuro es bastante prometedor.

2.3.1. Objetivos

Por tanto, la llegada de la tecnología P4 supuso una cambio de paradigma a la hora de diseñar nuevos protocolos. Esto es así ya que permitía decir a los *switches* como debían operar en vez de estar limitados por las condiciones de diseño del propio *switch* (Ver figura 2.6). Este paradigma se alcanzaría encontrando un término medio entre la expresividad del lenguaje P4 y la compatibilidad de éste entre la gran mayoría de hardware y soft-switches. Por ello, se plantearon tres hitos a conseguir [?].

- **Reconfigurabilidad.** En un entorno SDN, el controlador debería ser capaz de redefinir el *datapath* de los *switches*.
- **Agnóstico a protocolos.** Los *switches* no deben tener ningún conocimiento sobre los protocolos de red, es deber del controlador especificar como debe analizar y procesar los paquetes que le lleguen para extraer las cabeceras, y una serie de tablas del tipo *match-action* para procesar esas cabeceras.
- **Independencia de la arquitectura.** Al igual que los programadores de aplicaciones no tienen que especificar sobre que arquitectura están compilando su aplicación, se quería que los programadores de P4 no se preocuparan sobre en que arquitectura se iba a correr dicho programa P4. Será deber de los compiladores tomar el programa P4 totalmente independiente de arquitectura y convertirlo en un programa que dependa de la arquitectura en cuestión.

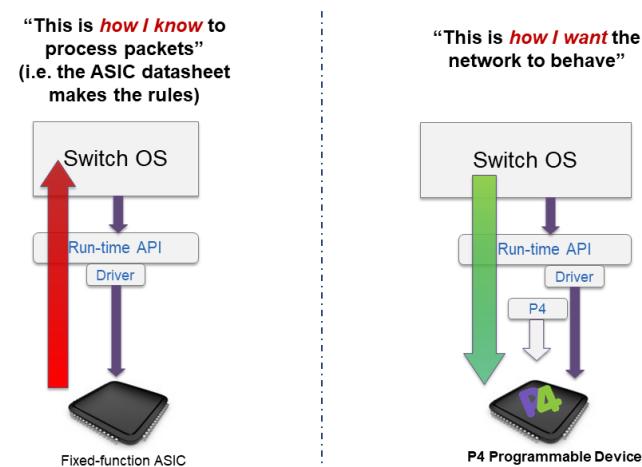


Figura 2.6: Cambio de paradigma con la tecnología P4 [?]

2.3.2. Arquitectura y targets

A partir de la especificación P4₁₆ se definen dos nuevos conceptos para conseguir alcanzar la independencia de los programas P4 de las distintas arquitecturas de cada fabricante.

- El concepto de *Target*, se puede ver como una implementación del hardware/software específica donde se va a correr un programa P4.
- El concepto de *Architecture*, se puede definir como la interfaz presentada por el fabricante para programar un *target* a través de componentes P4.

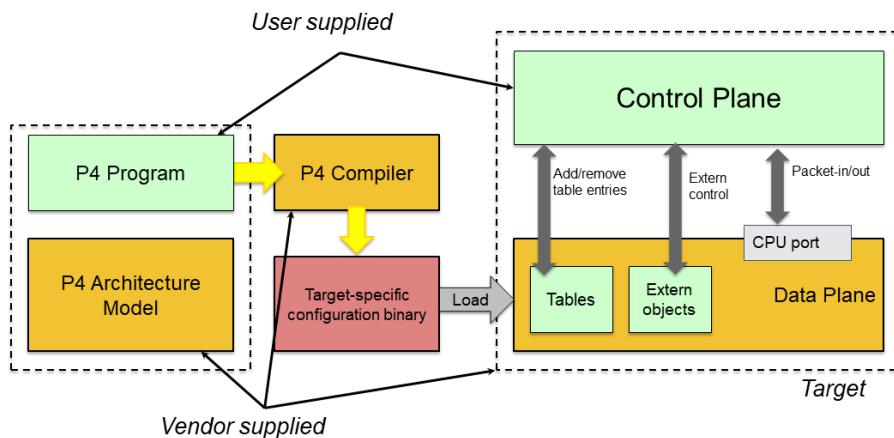


Figura 2.7: Arquitectura de la tecnología P4 [?]

Como se puede ver en la figura 2.7, el fabricante es el encargado de proveer la arquitectura, compilador y el *target*. Y el usuario solo se tendrá que preocupar de escribir un programa P4 y definir la interfaz de control. De esta forma, se quiere conseguir que el usuario solo se preocupe en seguir la interfaz/arquitectura dada por el fabricante para programar un *target* en específico.

Un programa P4 desarrollado para una arquitectura será exportable entre distintos *targets* que implementen la misma arquitectura. Por lo que, si las arquitecturas como por ejemplo v1model, PSA se asientan entre los distintos fabricantes de hardware se conseguirá la independencia del hardware, para depender solo del software.

2.4. Tecnología XDP

La tecnología XDP, es una tecnología para procesar paquetes, programable, de alto rendimiento e integrada en el *datapath* del Kernel de Linux. Esta tecnología se basa en ejecutar *bytecodes* Berkeley Packet Filter (BPF) cuando la interfaz sobre la cual está anclado un programa XDP recibe un paquete. Esto habilita que ciertas decisiones para según qué paquetes, se realicen lo antes posible casi en la propia interfaz, quitándose de encima todas las capas del *stack* que harían operaciones redundantes.

Pero el hecho de que se ejecute casi en la propia interfaz no es la única razón del alto rendimiento de los programas XDP. Otras decisiones que han sido vitales para hacer de esta tecnología una pieza clave para las nueva generación de herramientas⁵ en Linux.

- No se reserva memoria mientras se hace el procesamiento de paquetes con XDP.
- No se emplea las estructuras `sk_buff` para gestionar la información de metadatos de los paquetes, en cambio se hace uso de una estructura `xdp_buff` la cual no contiene tanta información en exceso.
- No se permiten bucles, ni un número excesivo de iteraciones en los programas anclados a la Network Interface Card (NIC).

El funcionamiento de un programa XDP se basa generalmente en analizar un paquete, editarlo en caso de que sea necesario y por último, devolver un código de retorno que indique que se debe hacer con dicho paquete. Los códigos de retorno XDP, se utilizan para determinar qué acción se va a tomar con el paquete, estas acciones pueden ser muy variadas, desde hacer que el paquete sea descartado, hasta delegarlo al *stack* de red para que se encargue este del procesado, o reenviar el paquete para que sea transmitido de nuevo.

Antes se mencionaba que los programas XDP, consistían en ejecutar *bytecodes* BPF cuando llega un paquete a la interfaz donde está cargado el programa, pero, ¿Qué relación existe entre XDP y BPF? Resulta que todos los programas XDP que se escriben en un C restringido, se compilarán haciendo uso del compilador de `clang`⁶ como *frontend* y del compilador `LLVM`⁷ como *backend*, para conseguir un *bytecode* BPF.

La motivación de esta “traducción” radica en que los programas XDP son cargados en el Kernel a través de la llamada al sistema BPF, indicando que se trata de un tipo de programa XDP con la macro, `BPF_PROG_TYPE_XDP`. De esta forma, se permite reutilizar los mecanismos de carga de *bytecodes* en el Kernel utilizados con BPF, pero en este caso con XDP estarán únicamente enfocados en la carga de dichos *bytecodes* en la NIC. Por lo que, XDP se podría ver como un *framework* BPF diseñado para trabajar en la interfaz, con limitaciones y pautas de desarrollo muy marcadas para conseguir que la tecnología sea de alto rendimiento [?].

⁵ <https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git>

⁶ Compilador de tipo frontend de referencia de la familia del lenguaje C.

⁷ Infraestructura para implementar compiladores de forma agnóstica al lenguaje a compilar, generalmente usado como compilador de Backend.

2.4.1. Modos de operación

Modo de Operación	Descripción
OFFLOADED XDP	En este modo de operación, el programa XDP se carga directamente en la propia NIC en lugar de ser ejecutado en la CPU del host. Al sacar la ejecución fuera del sistema y delegarla a la propia NIC, este modo tiene las mejores ganancias de rendimiento.
NATIVE XDP	En este modo de operación los programas XDP se ejecutan lo antes posible una vez recibidos por el driver de la NIC. Este modo no está disponible en todos los drivers (Todos aquellos que permiten este modo gestionan la macro XDP_SETUP_PROG).
GENERIC XDP	<p>Este modo de operación se proporciona como un modo de prueba para desarrollar programas XDP. Está soportado desde la versión 4.12 del Kernel y se puede utilizar este modo por ejemplo en pares de Veths.</p> <p>Nótese que no se obtendrá el mismo rendimiento que con los dos modos anteriores de funcionamiento.</p>

Tabla 2.1: Resumen modos de operación en XDP

2.4.2. Procesamiento de paquetes

Una vez que se ha analizado el paquete, se ha filtrado por los valores de sus cabeceras o se ha modificado, se tendrá ya pensada una acción a realizar con este paquete. Para expresar dicha acción, se hará uso de los códigos de retorno XDP [?].

- XDP_DROP, se descarta el paquete, esto se hará lo antes posible en la etapa de recepción de los paquetes. Siendo un mecanismo de muy útil para proteger al sistema de ataques Denial of Service (DoS), ya que cada paquete implicará una cantidad ínfima de procesamiento.
- XDP_ABORTED, este código de retorno se utilizará para denotar un error, ya que descartará el paquete y además generará una excepción del tipo `xdp_exception`.
- XDP_PASS, con este código de retorno se delegarán los paquetes al *stack* de red.
- XDP_TX, este código de retorno XDP, se utiliza para reenviar el paquete a la misma interfaz por la cual se recibió para su transmisión.
- XDP_REDIRECT, este código de retorno generalmente su emplea cuando se hace un reenvío del paquete desde una NIC a otra NIC.

Todos los códigos de retorno se pueden encontrar definidos como una enumeración en el archivo de cabecera llamado `<linux/bpf.h>`. En el bloque 2.1 se puede ver dicha definición.

Código 2.1: Definición de códigos de retorno XDP

```

1 enum xdp_action {
2     XDP_ABORTED = 0,
3     XDP_DROP,
4     XDP_PASS,
5     XDP_TX,
6     XDP_REDIRECT,
7 };

```

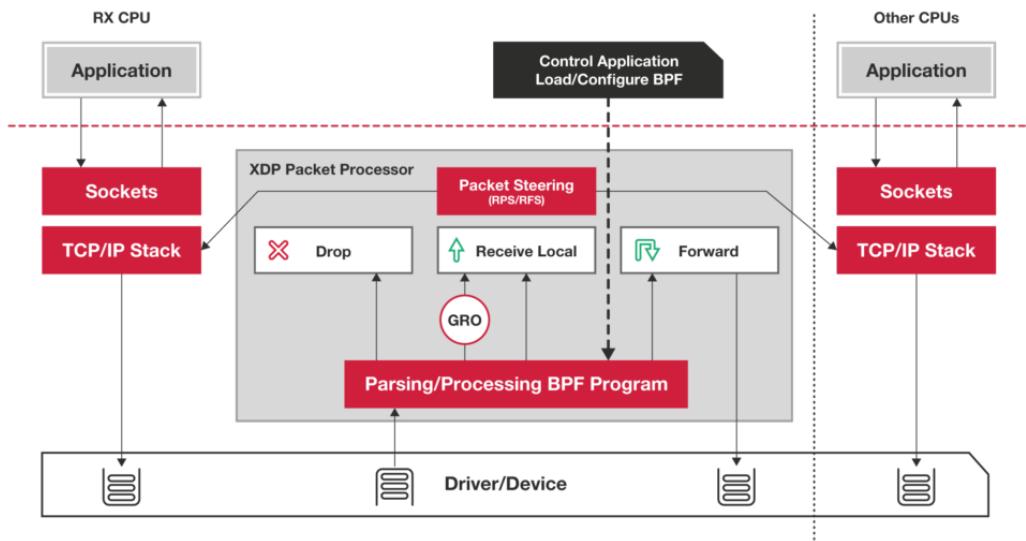


Figura 2.8: Procesamiento de paquetes con XDP

2.4.3. Limitaciones con XDP

En esta subsección se indicarán todas las limitaciones que se pueden encontrar a la hora de trabajar con la tecnología XDP, además de ciertos detalles que hayan parecido limitaciones aunque sean características de la propia tecnología en favor de ofrecer más seguridad.

2.4.3.1. Accesos a memoria y comprobación de límites

El acceso a los paquetes se hará de modo directo a la memoria, sabiendo que el verificador se asegurará de que dichos accesos sean seguros. Sin embargo, hacer esto en tiempo de ejecución para cada acceso a memoria resultaría en una sobrecarga de rendimiento significativa. Por lo tanto, lo que el verificador hace es comprobar que el programa XDP hace su propia comprobación de límites, en su propia lógica [?]. Por ejemplo, si con un programa se están parseando cabeceras Ethernet se haría la comprobación de límites indicada en el bloque 2.2.

Código 2.2: Comprobación de límites en XDP

```

1 SEC("xdp_prog")
2 int xdp_ether_parser(struct xdp_md *ctx){
3
4     void *data_end = (void *) (long) ctx->data_end;
5     void *data = (void *) (long) ctx->data;
6     int hdrsize = sizeof(struct ethhdr);
7
8     /* Comprobación de límites */
9     if (data + hdrsize > data_end)
10         return -1;
11
12     return XDP_PASS;
13 }
```

En el caso de que se manejen más tipos de cabeceras posteriormente, habría que hacer también comprobaciones de límites con los distintos tipos de cabeceras. De forma general, las definiciones de los parsers se suelen definir aparte como funciones *inline* y se llaman desde el propio programa XDP para hacer el código más legible.

Cuando el verificador realiza su análisis estático en tiempo de carga, rastreará todas las direcciones de memoria utilizadas por el programa, y buscará comparaciones con el puntero data-end, el cual será puesto al final del paquete en tiempo de ejecución. En el momento que el verificador crea que puede darse un acceso indebido, descartará el programa XDP.

2.4.3.2. Bucles y funciones inline

Puesto que XDP se puede ver como un *framework* BPF, también comparte sus limitaciones como son las limitaciones de llamadas a funciones y los bucles. Por ello, muchas de las funciones que se van a desarrollar en los programas XDP necesitarán ser de un carácter *inline* (`_always_inline`) para que así el compilador las sustituya tal cual en código.

De esta forma se consigue una mejora de rendimiento, ya que no hay que hacer una llamada a una función con todo lo que ello conlleva (Cambio de contexto, salvado de registros, incremento del PC), y se consigue evitar las limitaciones de llamadas a funciones del verificador [?].

En cuanto a la limitación de los bucles, BPF no soporta los bucles, por lo que se deberá desenrollar los bucles. Pero, ¿Qué significa desenrollar un bucle? Atendiendo al bloque 2.3 podemos hacernos una idea.

```
Código 2.3: Loops Unrolling
1
2 #pragma unroll
3 for ( int j = 0; j < 4 ; j++)
4     printf("Contador %d \n", j);
5
6
7 // Si lo desenrollamos, quedaría así:
8 printf("Contador %d \n", 0);
9 printf("Contador %d \n", 1);
10 printf("Contador %d \n", 2);
11 printf("Contador %d \n", 3);
```

Para conseguir esto debemos añadir la siguiente declaración antes del bucle *pragma unroll*. Esta declaración solo es válida cuando el numero de iteraciones del bucle es conocida en tiempo de compilación y no supera el numero máximo de instrucciones que admite el verificador del Kernel (4096) [?].

2.5. 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.

2.5.1. 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 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 *realloc* 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 2.4: 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.

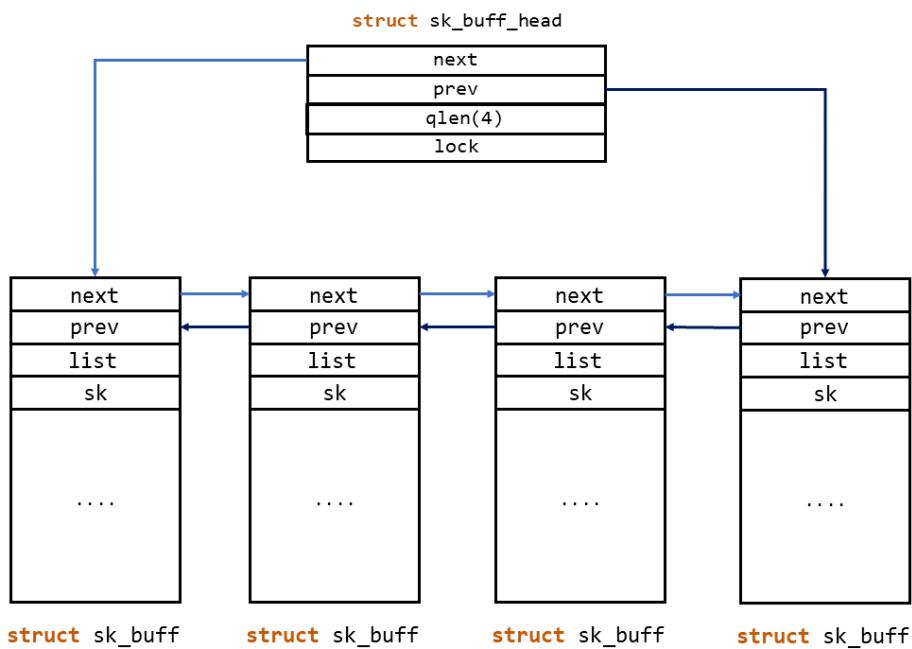


Figura 2.9: Sistema de colas doblemente enlazadas

Por claridad en la figura 2.9 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 elemento 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.

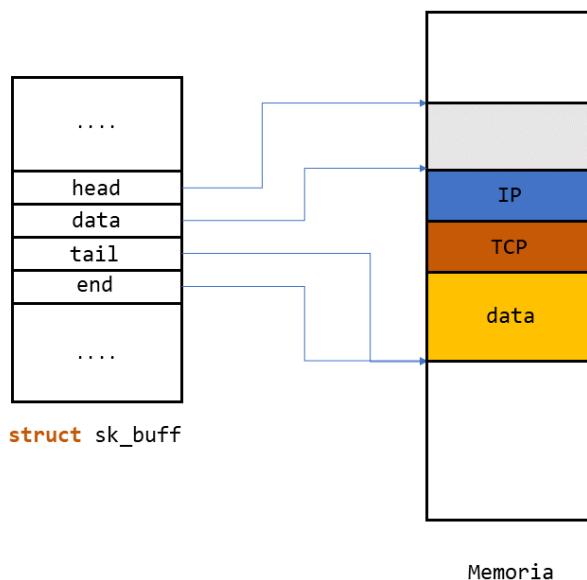


Figura 2.10: Punteros de la estructura `sk_buff`

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

2.5.2. Herramienta TC

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

- SHAPING
- SCHEDULING
- POLICING
- DROPPING

Según su *man-page*⁸, el procesamiento del tráfico para conseguir dichas funcionalidades, se lleva a cabo con tres tipos de objetos: **qdiscs**, **classes** y **filters**.

2.5.2.1. 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 a la 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.

2.5.2.2. Classes

La 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 2.11. 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.

2.5.2.3. 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 BPF⁹, 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 2.5.1), por lo que

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

⁹<https://man7.org/linux/man-pages/man8/tc-bpf.8.html>

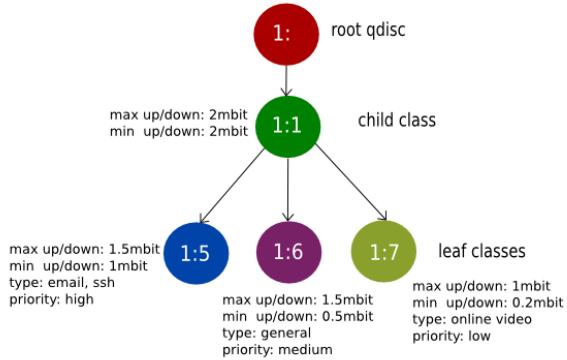


Figura 2.11: Sistema de QoS implementado con distintas clases [?]

ciertos *helpers* BPF para clonar paquetes podrán ser utilizados por *bytecode* anclado en el filtro.

2.5.3. 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 2.2 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 Docker¹⁰, pero, ¿Realmente qué hace 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.

2.5.3.1. Persistencia de las Namespaces

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

¹⁰<https://www.docker.com/>

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 2.2: Resumen de los tipos de Namespaces en el Kernel de Linux

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 Namespace a día de hoy puede ser referenciada de tres maneras distintas:

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

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

2.5.3.2. Concepto de las Network Namespaces

Una vez entendido el concepto de *Namespace* 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 [?].

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 2.5: 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}
```

2.5.3.3. 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 [?].

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 2.6, se recuerda que es necesario permisos de súper usuario.

Código 2.6: 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 2.12, 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.

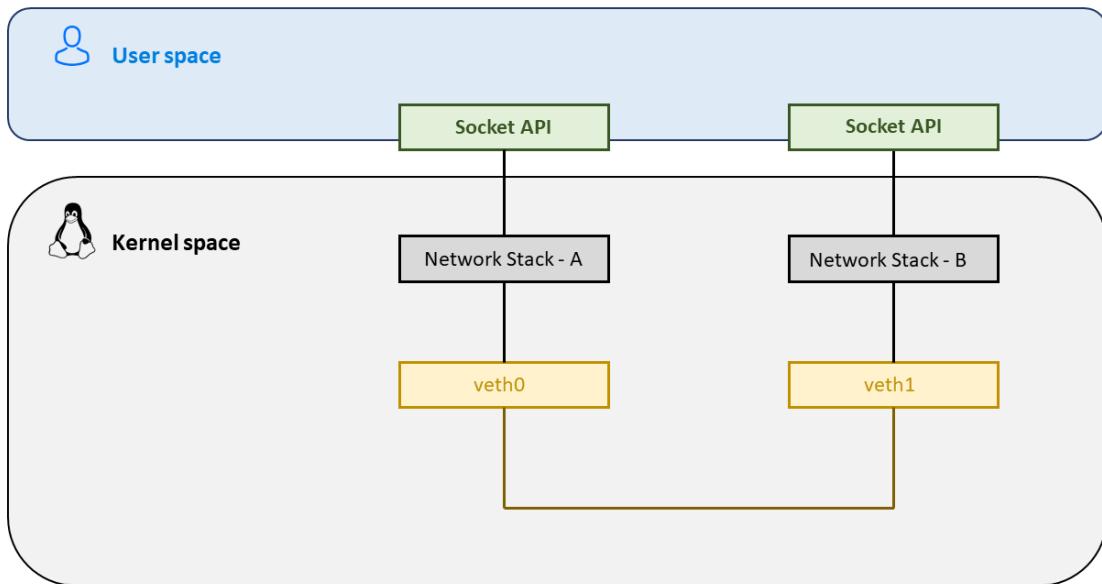


Figura 2.12: Enlace entre interfaces Veth separadas en dos Network Namespaces

2.6. 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.

2.6.1. 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 2.5.3) [?].

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 independientes¹¹ 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 [?].

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 *Namepaces*, más adelante se indicará 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 extrapolables¹² 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 *Namepaces* 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 “*Namepaces*”, o en caso de hacerlo, su API para hacer uso de ellas sea completamente diferente.

¹¹En la parte de Networking

¹²Los 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

2.6.2. 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 2.13, 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 *Namespace* por defecto (root). El mecanismo para comunicar a los nodos de esta topología como se adelantaba anteriormente son las Veths (Ir a 2.5.3.3), las cuales permitirán emular los enlaces entre los distintos nodos de la red.

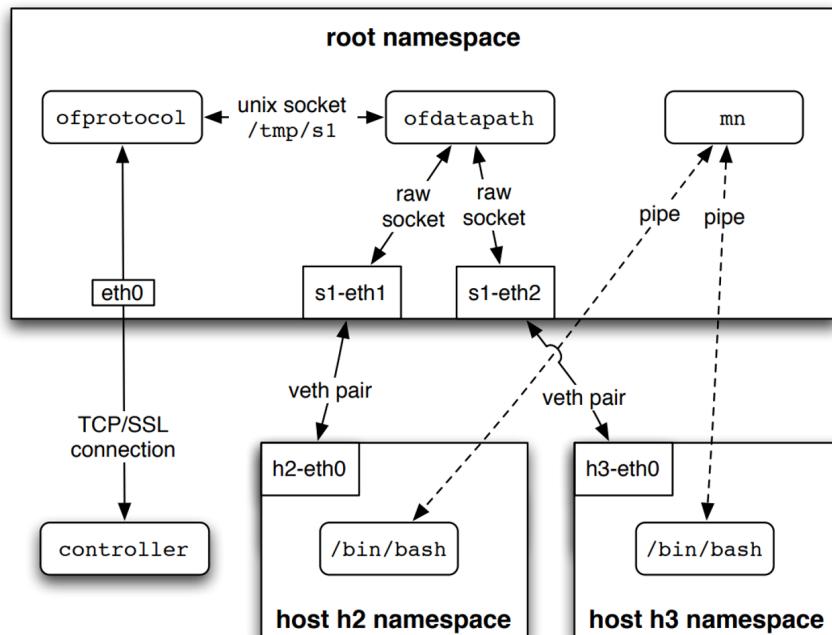


Figura 2.13: Arquitectura de Mininet [?]

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 2.13, para levantar dicha topología únicamente se tiene que tener Mininet instalado y seguir los pasos que se indican el bloque 2.7.

Código 2.7: Levantamiento de la topología de ejemplo

```

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

```

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

Figura 2.14: Topología de ejemplo levantada

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 2.8.

Código 2.8: Listar Network Namespaces

```
1 sudo ip netns list
```

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

Figura 2.15: Listado de Network Namespaces existentes en el sistema

Según se puede apreciar en la figura 2.15, 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 nombre¹³.

Como ya se explicó, las *Namespaces* tienen una vida finita, éstas viven siempre y cuando estén referenciadas (Ir a 2.2), por tanto, si ninguna condición de referencia 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

¹³ Aquellas 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.

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 hora 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:s1
n0obie@n0obie-VirtualBox:~$ █
```

Figura 2.16: Listado de procesos asociados a Mininet

Si se inspecciona el fichero `/proc/{pid}/ns/net` para cada proceso indicado en la figura 2.16 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 2.17: 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 2.18: 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.

2.6.3. 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 2.3, 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](#)¹⁴.

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 2.3: Resumen comandos existentes en Mininet

2.6.4. Mininet-WiFi

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

Esto es así ya que toda la funcionalidad wireless es un añadido sobre la base que desarrollaron para Mininet. Los desarrolladores de Mininet-WiFi se valieron del subsistema wireless del Kernel de Linux y del módulo `mac80211_hwsim`, para conseguir emular las interfaces y el supuesto medio inalámbrico. Para más información sobre esta herramienta se recomienda ir al [punto 4.3.1](#), 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.

¹⁴<https://github.com/mininet/mininet/wiki/Introduction-to-Mininet>

2.6.5. Mininet-IoT

La herramienta Mininet-IoT [?] 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.

2.7. 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 GitHub¹⁵. 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 [?].

El Kernel de Contiki está orientado a eventos, y soporta tareas multi-hilo con requisa. Contiki está escrito en el lenguaje C y ha sido portado a numerosas 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 2.19, 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.

¹⁵<https://github.com/contiki-os/contiki>

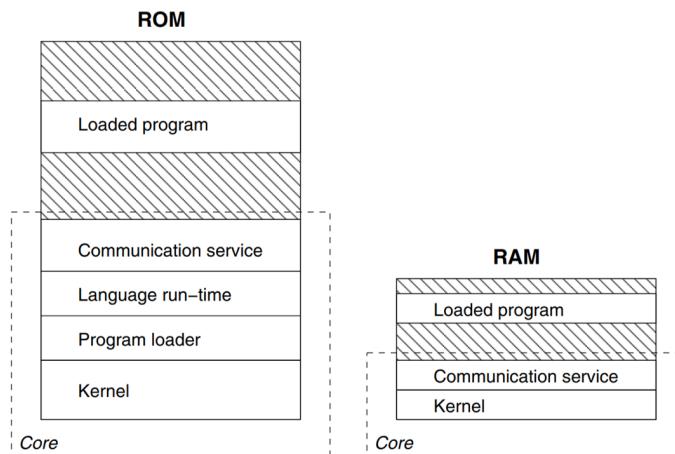


Figura 2.19: Particionado en un sistema con Contiki OS [?]

En los últimos años apareció un nuevo proyecto, **Contiki-*ng***¹⁶, el cual es fork de Contiki OS. Este nuevo proyecto desbancaría 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 [?].

2.7.1. 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, 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 Cooja¹⁷. 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.

¹⁶<https://github.com/contiki-ng/contiki-ng>

¹⁷<https://github.com/contiki-ng/cooja/tree/master>

2.8. Contribuciones en GitHub

La herramienta GitHub [?] es una plataforma para alojar repositorios de forma remota. En este TFG, se hará uso de la herramienta de control de versiones Git¹⁸, 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.

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 2.4 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 2.4: Resumen de contribuciones realizadas

¹⁸<https://git-scm.com/>

3. Diseño y análisis de casos de uso

En este capítulo, teniendo en cuenta el marco teórico explicado en el Estado del Arte (Cap. 2), se analizará qué funcionalidades básicas deben poder ser implementadas por las distintas tecnologías para definir el *datapath* de los dispositivos IoT, con el fin de alcanzar los objetivos indicados en la Introducción (Cap. 1).

Para ello, se diseñarán distintos casos de uso para comparar así las características que ofrecen cada una distintas tecnologías en relación a su integración de los dispositivos IoT en entornos SDN. De forma adicional, se estudiará sobre que plataformas se desplegarán los casos de uso, asegurándose de que tienen todos los mecanismos necesarios para llevar a cabo las pruebas.

3.1. Funcionalidades básicas

Como se ha podido ver en el capítulo del Estado del arte (Cap. 2), las tecnologías P4 y XDP tienen muchos puntos fuertes, y ambas permiten definir el *datapath* del dispositivo en cuestión. Con XDP, obtenemos un gran rendimiento ya que el procesamiento de los paquetes se realiza casi en la propia interfaz, no consume recursos de forma activa, ya que opera de forma reactiva a los paquetes que llegan a la interfaz. Por otro lado, se encuentra la tecnología P4, la cual propone hacer uso de un lenguaje agnóstico del hardware para definir el *datapath*. Esto aporta mucha versatilidad a la hora de implementar nuevos protocolos, o nuevas especificaciones de estándares.

Teniendo en cuenta estas premisas, se ha decidido plantear una serie de funcionalidades básicas que una hipotética moto IoT tendría que implementar. Estas funcionalidades se van a recoger en casos de uso, los cuales serán desarrollados con ambas tecnologías (P4 y XDP) para poder concluir con cual de ellas es más sencillo/eficiente realizar la integración de los dispositivos IoT en entornos SDN, tal y como se ha definido en el objetivo principal del presente TFG.

Los casos de uso que se plantean son los siguientes:

- **Case01 - Drop:** En este caso de uso se quiere ver si es posible descartar paquetes.
- **Case02 - Pass:** En este caso de uso se quiere comprobar si es posible dejar pasar los paquetes, sin afectarles el plano de datos programado con la tecnología.
- **Case03 - Echo server:** Este caso de uso, aunque se desarrollará un servidor el cual contestará todos los ECHO-Resquest que le lleguen, lo que se quiere ver es la facilidad que tenemos con ambas tecnologías de filtrar/parsear paquetes en base a sus cabeceras.

- **Case04 - Layer 3 forwarding:** Con este caso de uso se comprobará qué tan sencillo es reenviar paquetes desde una interfaz a otra del dispositivo.
- **Case05 - Broadcast:** Por último, con este caso de uso se quería abordar cómo se puede conseguir la difusión (o broadcast) de un hipotético paquete, es decir, si es posible clonar paquetes con las tecnologías o, en su lugar, crearlos de cero.

Una vez definidas las funcionalidades a desarrollar con ambas tecnologías, se debe decidir sobre qué elementos se van a probar. En el caso de XDP es sencillo, ya que como se explicó en el Estado del arte, los programas XDP pueden ser cargados en la mayoría de interfaces gestionadas por el Kernel de Linux.

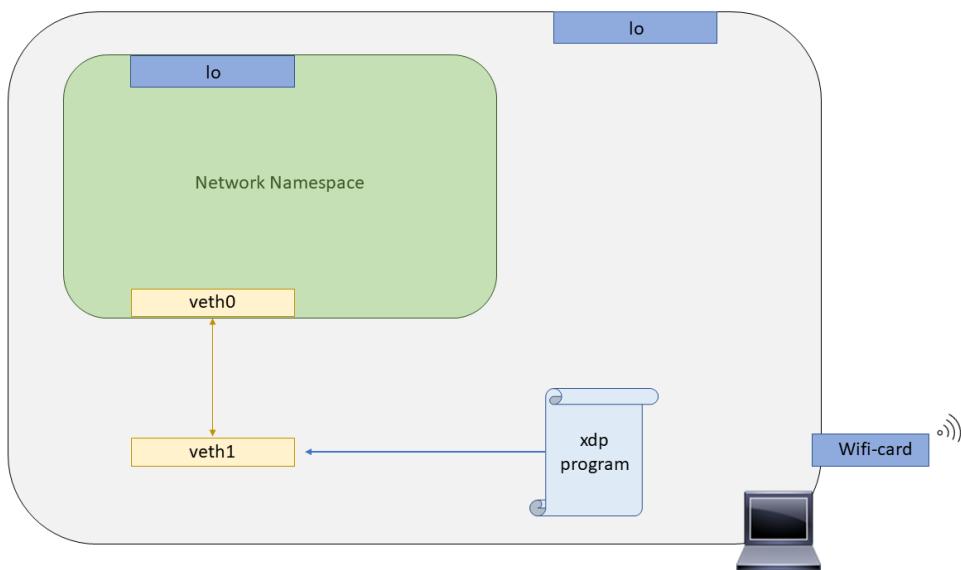


Figura 3.1: Ejemplo de carga de un programa XDP sobre una interfaz Ethernet virtual

En el caso de los programas P4 sí será necesario hacer uso de un elemento adicional para probar la funcionalidad desarrollada. Dicho elemento es el Behavioral Model (BMv2) [?], un soft-switch¹ desarrollado con la finalidad de probar código P4. Como se puede apreciar en la figura 3.2, el programa P4 una vez compilado se obtendrá un fichero JSON [?] resultante que será el que le indique al BMv2 qué *datapath* debe implementar.

Atendiendo a los modelos de integración de dispositivos IoT comentados en la Introducción 1.2 y 1.3, se van a desarrollar casos de uso tanto en entornos **cableados** como para entornos **inalámbricos**. La motivación de esta decisión se basa en que, como se puede ver en la figura 1.2, habrá dispositivos híbridos IoT mediadores, los cuales estarán parcialmente conectados al core SDN de manera cableada, y de manera inalámbrica con el resto de dispositivos IoT. Por ello, se estima necesario comprobar el rendimiento de ambas tecnologías en ambos entornos ,

¹Switch virtual que se implementa a nivel de software.

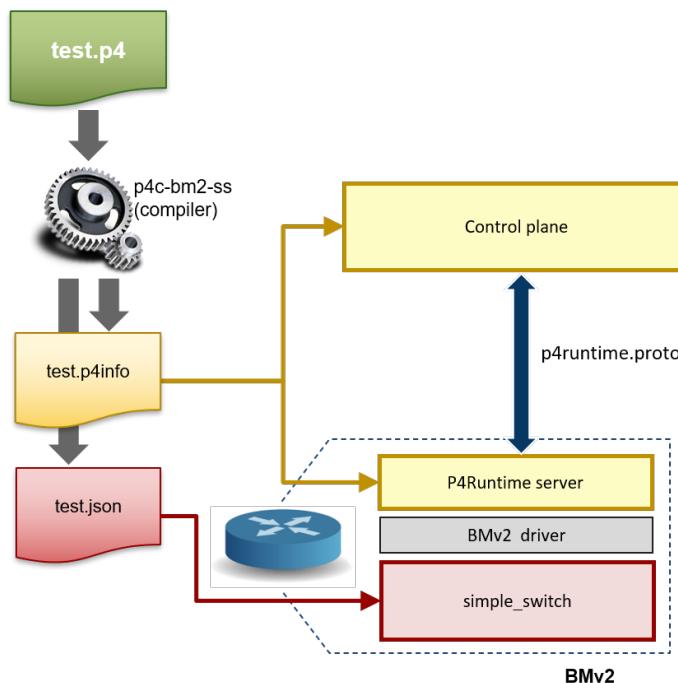


Figura 3.2: Ejemplo de carga de un programa P4 sobre el BMV2

ya que ciertos dispositivos IoT en las integraciones parciales pueden ser favorecidos de igual manera.

A continuación, se analizará sobre qué plataformas se llevará a cabo la evaluación de las funcionalidades básicas desarrolladas. Se elegirán plataformas para entornos cableados y para entornos inalámbricos, intentando siempre hacer uso de un número mínimo de ellas para que así las evaluaciones sean lo más homogéneas posibles.

3.2. Plataformas de pruebas alámbricas

Para evaluar los desarrollos de XDP y P4 en entornos cableados se hará uso del estándar **ieee8023** (Ethernet), aunque este estándar se diseñó en un principio para operar en redes de carácter local, con el paso de los años la tecnología Ethernet fue avanzando a la par que su popularidad, alcanzando tasas de transmisión de *40 Gbit/s* con el estándar **ieee8023ba**².

Las evaluaciones de los programas XDP serán realizadas sobre una plataforma Linux donde se generarán topologías de red conformadas por *Network Namespaces*, para definir los nodos independientes de la red, y de Veth, para emular los enlaces entre distintos nodos. Todos los escenarios empleados serán suministrados en el repositorio del TFG en forma de *shellscrip*, indicando al usuario cómo lanzar dicho escenario, y cómo hacer uso del mismo script para limpieza del escenario recreado. Se ha querido hacer uso de *Network Namespaces* como pla-

²<http://www.ieee802.org/3/ba/>

taforma para validar los desarrollos, ya que es la expresión más básica de emulación de redes en el Kernel de Linux. De esta forma, y debido a la complejidad que supone trabajar con programas XDP, se buscaba que la plataforma donde se fueran a evaluar los casos de uso aportase la mínima incertidumbre posible sobre los resultados obtenidos.

En cuanto a la plataforma elegida para evaluar los programas P4, se hará uso de Mininet. Como ya se comentó en el Estado del Arte (Cap. 2), Mininet es una herramienta para la emulación de topologías de red SDN, de la cual nació Mininet-WiFi añadiendo soporte para redes inalámbricas. Se ha elegido Mininet, debido a que el equipo de *p4lang* desarrolló una interfaz³ del BMv2 con dicha herramienta, de esta forma es posible incorporar el soft-switch de referencia para probar código P4, como un tipo de nodo más en Mininet.

3.3. Plataformas de pruebas inalámbricas

Para evaluar los desarrollos de XDP y P4 en entornos inalámbricos se hará uso del estándar **ieee80211** (WiFi). Se ha elegido dicho estándar como un primer paso, debido a que a día de hoy, existen más herramientas y desarrollos que para el estándar **ieee802154**. El estándar **ieee80211** es uno de los estándares de redes inalámbricas más populares del mundo recibiendo constantes actualizaciones por parte del IEEE para mejorar su rendimiento, o dar soporte a nuevas funcionalidades [?]. Siendo este grupo de trabajo uno de los más activos dentro del proyecto de estandarización del IEEE de redes de área local (Fig. 3.3).

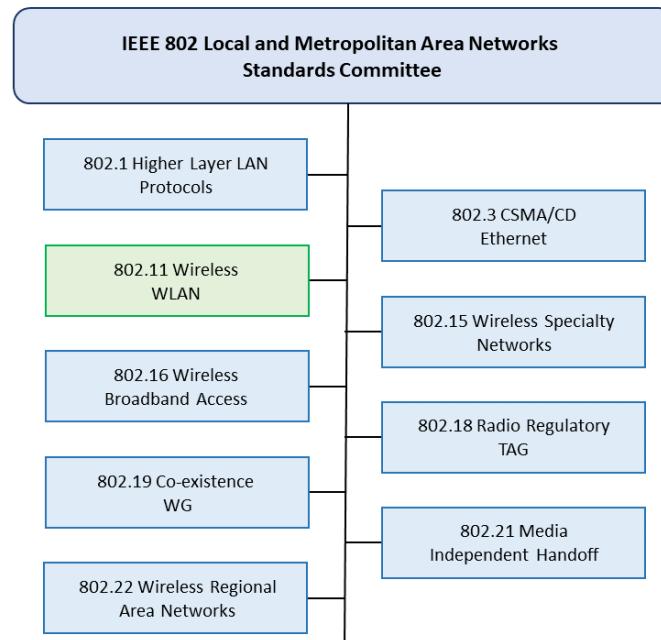


Figura 3.3: Grupos de trabajo IEEE 802

³<https://github.com/p4lang/tutorials/tree/master/utils/mininet>

La plataforma que se utilizará para la evaluación de los distintos casos de uso, tanto en XDP como en P4, será la misma. Se hizo uso del emulador Mininet-WiFi en lugar del simulador Cooja para evaluar el funcionamiento de ambas tecnologías (XDP y P4) en entornos inalámbricos. Esto se debe a que para probar ambas tecnologías se requiere de interfaces reales o emuladas donde poder evaluar el funcionamiento de los desarrollos. También se podría haber valorado hacer uso del emulador Mininet-IoT, pero como se mencionó en la sección 2.6.5, dicha herramienta se integró en Mininet-Wifi como un complemento más. Por lo que, eligiendo a Mininet-WiFi se tiene las nuevas funcionalidades de Mininet-IoT y además, cuenta con mayor madurez al llevar más años en la escena.

En los desarrollos XDP, el flujo de trabajo consistirá en a levantar la topología en Mininet-WiFi, a través de un script en Python que hará uso de la API de Mininet-WiFi para recrear el escenario, acto seguido se anclará el programa XDP en las interfaces del nodo de la red que corresponda. En cuanto a los programas P4, se ha visto que la plataforma no contempla ningún tipo de nodo que de soporte al BMv2 por lo que será necesario desarrollar previamente una integración del BMv2 en Mininet-WiFi. Es importante comentar que durante el desarrollo de este TFG, el desarrollador principal de Mininet-WiFi, profesor de la IFBA, Ramon Fontes⁴ empezó de forma paralela una integración de P4 en Mininet-WiFi abriendo un *Issue*⁵ donde mencionaba cómo debía desarrollarse la integración. En dicho *Issue*, se pudieron debatir los detalles de la implementación de los soft-switch P4 en Mininet-WiFi. Más adelante, en el capítulo de desarrollo se expondrá como se abordó la integración, explicando su pre-implementación y desarrollo.

El desarrollo de esta integración será vital para el proyecto. Teniendo en cuenta que la evaluación de los programas P4 en entornos inalámbricos dependerá de si se es capaz de embeber el BMv2 en la arquitectura de Mininet-WiFi. Los pasos propuestos para llevar a cabo la integración son los siguientes:

- Estudio y análisis de la interfaz creada desde el equipo *p4Lang*, del BMv2 con Mininet.
- Estudio del subsistema *Wireless* de Linux, y análisis del funcionamiento interno de Mininet-WiFi. Este punto será de vital importancia, puesto que se debe tener un buena idea de cómo trabaja Mininet-WiFi a bajo nivel.
- Desarrollo de la integración, se debe conseguir que el BMv2 gestione las interfaces que Mininet-WiFi cree en los nodos del tipo Access Point (AP). De forma adicional, se tendrá que conseguir que el BMv2 se capaz de correr dentro de una *Network Namespace*. Esta condición fue impuesta por Ramon Fontes en el *Issue* mencionado anteriormente.

Esta condición debía conseguirse para evitar problemáticas de *by-pass* en redes Ad-Hoc. No se podía tener corriendo en una misma *Network Namespace* todos los BMv2, ya que ciertos paquetes irían directamente al BMv2 final, sin seguir el camino dispuesto en la red emulada.

⁴<https://github.com/ramonfontes>

⁵<https://github.com/intrig-unicamp/mininet-wifi/issues/295>

4. Desarrollo y evaluación de casos de uso

En este capítulo, se describirá el desarrollo realizado tanto para los medios cableados como para los entornos inalámbricos, con la finalidad de cumplir el objetivo del presente TFG. Las secciones que se pueden encontrar en este capítulo son las siguientes: XDP en entornos cableados; P4 en entornos cableados; P4 en entornos inalámbricos; y por último, XDP en entornos inalámbricos. Según se puede apreciar el orden elegido para las secciones no es simétrico, esto se debe a que a la hora del desarrollo en medios inalámbricos hubo que analizar en profundidad el subsistema wireless de Linux con la finalidad de adaptar el BMv2 a Mininet-WiFi. Como resultado de dicho análisis se obtuvieron valiosas conclusiones que nos ayudarían a implementar los casos de uso de la tecnología XDP en entornos inalámbricos. Por ello, se cree que el orden elegido ayudará al lector a tener una mejor compresión de las implementaciones. Cada sección estará compuesta de cinco subsecciones, donde cada subsección representará un caso de uso descrito en el capítulo de Análisis (Cap. 3).

Se indicarán las partes más importantes de cada caso de uso, añadiendo explicaciones teóricas cuando proceda, y ofreciendo al final de cada caso una evaluación de su funcionamiento, proveyendo indicaciones para que se pueda replicar la funcionalidad esperada. En todos los casos de uso se han suministrado scripts para el despliegue de los desarrollos en entornos de red mínimos, intentando representar siempre una mota IoT.

4.1. Casos de uso XDP en medios cableados

En esta sección se introducirán todos los casos de uso realizados con la tecnología XDP en entornos cableados. Todos los casos de uso se han nombrado siguiendo la misma sintaxis que en el repositorio del TFG, alojado en GitHub. Para la **instalación de las dependencias** de la tecnología XDP, se ha generado el Anexo C.1 donde se detallan todos los pasos a seguir.

Los casos de uso se han dividido en cinco partes, con la finalidad de que la lectura de estos sea más clara y ordenada.

- **Introducción:** En esta parte se abordarán las explicaciones teóricas complementarias en caso de que sean necesarias, explicaciones propias sobre el caso de uso y comentarios sobre el código desarrollado.
- **Compilación:** En esta parte se explicará al lector cómo proceder para compilar el programa XDP.
- **Puesta en marcha del escenario:** En esta parte se explicará al lector cómo levantar el escenario y cómo poder limpiarlo tras finalizar la comprobación de funcionamiento, y por último se comentará el escenario recreado.

- **Carga de los programas XDP:** En esta parte se indicará sobre qué interfaz se producirá la carga de los programas XDP, y cómo realizar dicha carga.
- **Evaluación del funcionamiento:** Por último se hará una evaluación sobre el funcionamiento del caso de uso.

Para que el lector pueda seguir el desarrollo con los casos de uso, a continuación se indica la tabla 4.1, la cual expone en qué ruta del repositorio del TFG se puede encontrar dicho caso de uso, y un vídeo demostración donde el autor va comentando paso a paso el caso de uso y su evaluación.

Caso de uso	Enlace al repositorio	Enlace al vídeo demostración
case01 - Drop	Enlace al código	Enlace al vídeo
case02 - Pass	Enlace al código	Enlace al vídeo
case03 - Echo server	Enlace al código	Enlace al vídeo
case04 - Layer 3 forwarding	Enlace al código	Enlace al vídeo
case05 - Broadcast	Enlace al código	Enlace al vídeo

Tabla 4.1: Resumen de la documentación sobre los casos de uso XDP en entornos cableados

4.1.1. Case01 - Drop

En este caso de uso se probará que es posible descartar todos los paquetes recibidos haciendo uso de la tecnología XDP. Para la realizar la prueba, primero se debe compilar el programa XDP desarrollado y levantar el escenario donde se va a realizar la prueba. Acto seguido se anclará el binario a una interfaz del escenario, y por último, se observarán los resultados cuando se genere tráfico que atraviese dicha interfaz. En este caso, al descartar todos los paquetes con el código de retorno XDP_DROP, no habría conectividad.

Código 4.1: Programa básico XDP - Case01

```

1   SEC("xdp_case01")
2   int xdp_pass_func(struct xdp_md *ctx){
3
4       int action = XDP_DROP;
5       goto out;
6
7   out:
8       return xdp_stats_record_action(ctx, action);
9   }
10
11  char _license[ ] SEC("license") = "GPL";

```

Como se puede apreciar en el bloque 4.1, el programa es bastante básico, únicamente se está retornando un valor que indica que el paquete en cuestión debe ser descartado. La función `x dp _stats _record _action()` se explicará más adelante cuando sea necesario tomar estadísticas sobre los códigos de retorno XDP. Actualmente es totalmente inocua para el funcionamiento del programa.

Compilación

Para compilar el programa XDP se ha dejado un Makefile preparado en el directorio del caso de uso, por lo que de momento no es necesario entender todo el proceso de compilación. Más adelante se detallará este proceso, pero por ahora y dado que es el primer caso de uso, y puede que el primer contacto con esta tecnología, se ha preferido dar una visión directa y evitar los detalles minuciosos. Por lo que exclusivamente se deben seguir los pasos del bloque 4.2.

Código 4.2: Compilación programa XDP - Case01

```

1 # En caso de no haber entrado en el directorio asignado del caso de uso
2 cd TFG/src/use_cases/xdp/case01
3
4 # Hacemos uso del Makefile suministrado
5 sudo make

```

Con la ejecución de los comandos presentados, ya se habría compilado el programa XDP. Ahora se podrá observar que en el directorio actual se han generado varios ficheros con extensiones `*.11`, `*.o`, varios ejecutables que se utilizarán más adelante para anclar programas XDP en las interfaces (xdp_loader), y para comprobar los códigos de retorno de nuestros programas XDP una vez ya anclados (xdp_stats).

Puesta en marcha del escenario

Para testear los programas XDP se hará uso de las *Network Namespaces*. Debido a que el concepto de las *Network namespaces* podría ser un barrera de entrada para aquellas personas que nunca han trabajado con ellas y quisieran replicar los test, se decidió escribir un script para levantar el escenario, y para su posterior limpieza. De esta manera, aunque se haga uso de un concepto un poco “abstracto” del Kernel de Linux, este no será un impedimento para corroborar el funcionamiento de los casos de uso. Para levantar el escenario tenemos que ejecutar el *shellscrip*t indicándole el parámetro `-i` (*Install*).

Para limpiar la máquina del escenario recreado anteriormente, se puede correr el mismo script indicándole ahora el parámetro `-c` (*Clean*). En el peor de los casos, y si se cree que la limpieza se no se ha realizado de manera satisfactoria, se puede llevar a cabo un reinicio de la máquina consiguiendo así que todos los entes no persistentes (Veth, netns..) desaparezcan del equipo.

Código 4.3: Puesta en marcha del escenario - Case01

```

1 # Para levantar el escenario (Importante hacerlo con permisos de super usuario)
2 sudo ./runenv.sh -i
3
4
5 # Una vez finalizado la comprobación del caso de uso, limpiaremos nuestra maquina:
6 sudo ./runenv.sh -c

```

Por último, únicamente describir que el escenario recreado (Ver figura 4.1) es el siguiente,

compuesto exclusivamente de una *Network Namespace* (`uno`) y un par de Veths (`veth0` – `uno`) para comunicar la *Network Namespace* creada con la *Network Namespace* por defecto.

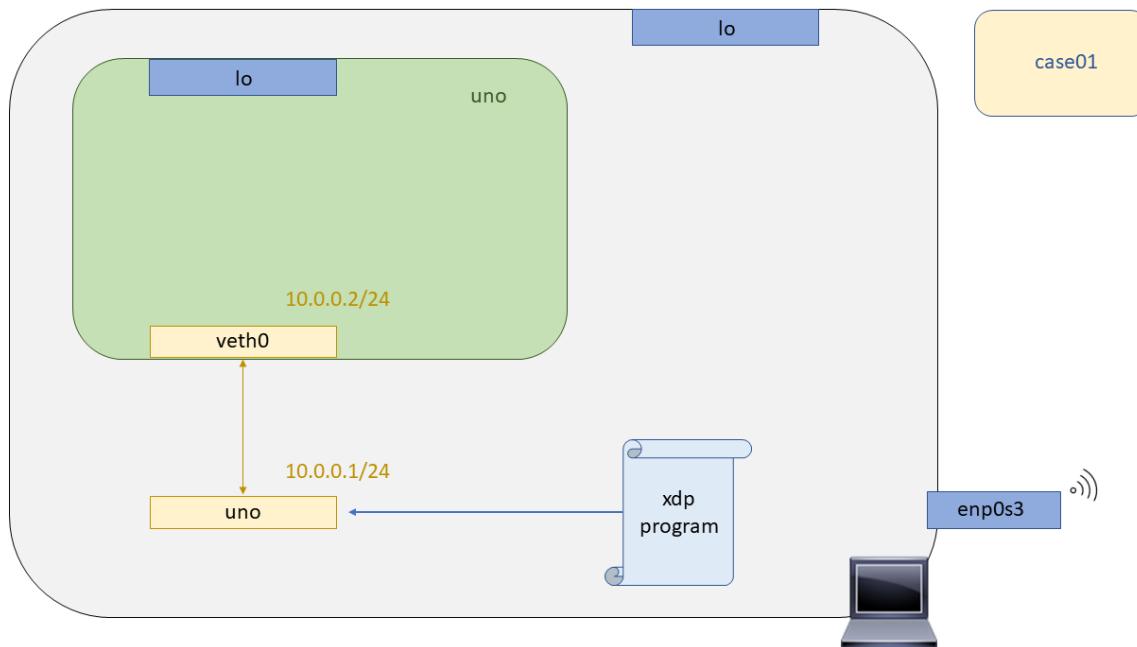


Figura 4.1: Escenario cableado del Case01 - XDP

Carga del programa XDP

Es hora de cargar el programa XDP en el Kernel. Para realizarlo, habría dos maneras de cargar el bytecode en el Kernel. La primera sería hacer uso de la herramienta iproute2 (C.2) a partir de la versión v4.12. La segunda, y la más utilizada debido a las limitaciones de iproute2 para trabajar con los mapas BPF, es hacer uso de la librería **libbpf**¹. En este caso se hará uso de un programa hecho en C haciendo uso de dicha librería para cargar los programas XDP en el kernel, mapas BPF y demás. El código de dicho programa se puede encontrar [aquí](#). Este *loader* fue desarrollado siguiendo el tutorial de los desarrolladores del kernel de Linux llamado *xdp-tutorial*².

Al *loader* se le está indicando `-d (device)`, `-F (Force)` para que haga un *override* en caso de que ya haya un programa XDP anclado a dicha interfaz, y por último, se le indica el `--progsec (program section)` utilizados en XDP para englobar distintas funcionalidades en un mismo *bytecode* compilado.

Código 4.4: Carga del programa XDP - Case01

¹<https://github.com/libbpf/libbpf>

²<https://github.com/xdp-project/xdp-tutorial>

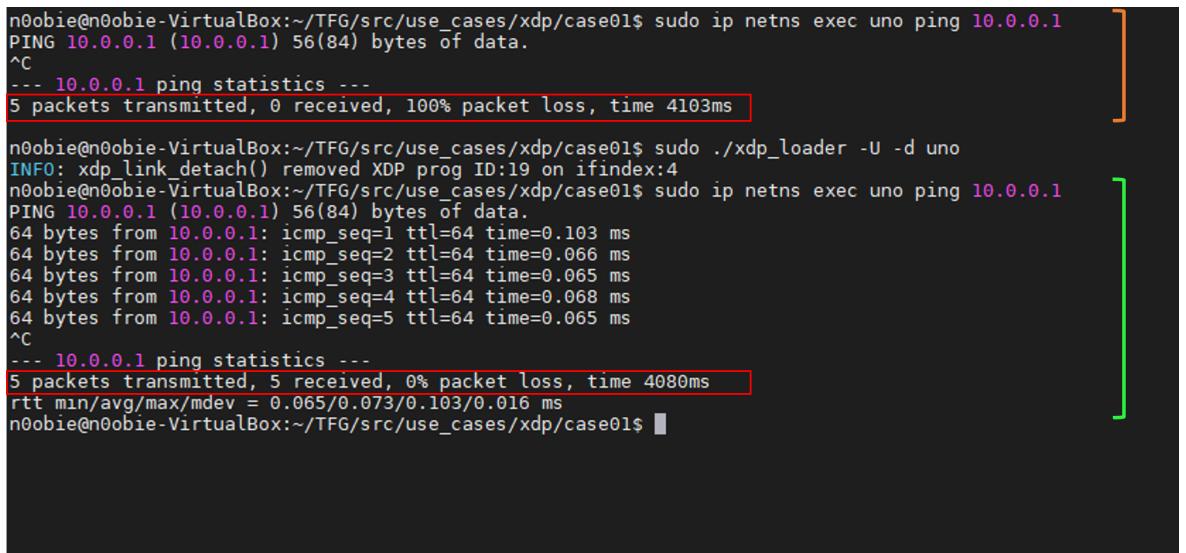
```

1 # Cargaremos el programa sobre la interfaz "uno"
2 sudo ./xdp_loader -d uno -F --progsec xdp_case01

```

Comprobación del funcionamiento

Una vez que el programa XDP fue anclado a la interfaz se comprobará de que funciona según lo esperado. Esto se hará generando tráfico desde un extremo de una Veth para que atraviese por la interfaz que tiene anclado el programa XDP. En este caso el comportamiento esperado del programa XDP es que haga un drop de los paquetes nada más llegar a la interfaz uno.



```

n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case01$ sudo ip netns exec uno ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4103ms

n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case01$ sudo ./xdp_loader -U -d uno
INFO: xdp_link_detach() removed XDP prog ID:19 on ifindex:4
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case01$ sudo ip netns exec uno ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.103 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.066 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.065 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.068 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.065 ms
^C
--- 10.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4080ms
rtt min/avg/max/mdev = 0.065/0.073/0.103/0.016 ms
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case01$ █

```

Figura 4.2: Comprobación de funcionamiento del Case01 - XDP

Como se puede ver en la figura 4.2, se accede al interior de la *Network namespace* uno y se realiza un ping █ hacia la interfaz con el programa XDP. Se puede apreciar que no hay conectividad, por lo que el programa XDP desarrollado está funcionando según lo esperado. Para corroborar que la falta de conectividad se debe al programa XDP, se va a desanclar de la interfaz y se probará de nuevo la conectividad. Según se puede ver en el último ping █, ya hay conectividad, por lo que se afirma que realmente los paquetes que atravesaban la interfaz eran descartados por el programa XDP.

4.1.2. Case02 - Pass

En este caso de uso se probará que es posible admitir todos los paquetes recibidos en la interfaz donde está anclado el programa XDP. Pero podría surgir la siguiente pregunta: ¿qué significa “Admitir”? Se quiere comprobar si es posible dejar pasar los paquetes, sin que les afecte el datapath programado con la tecnología. El motivo es que, aunque XDP se concibe normalmente como un mecanismo para hacer un *by-pass* al *stack* de red del Kernel de Linux (es decir, una completa redefinición del mismo), en muchas ocasiones será útil trabajar en conjunto para conseguir la funcionalidad deseada.

Para la realizar la prueba, al igual que en el case01 (4.1.1) primero se deberá compilar el programa XDP, acto seguido levantar el escenario donde se va a realizar la prueba, y por último, anclar el binario a una interfaz del escenario.

Código 4.5: Programa básico XDP - Case01

```

1  SEC("xdp_case02")
2  int xdp_pass_func(struct xdp_md *ctx)
3  {
4      int action = XDP_PASS;
5      goto out;
6  out:
7      return xdp_stats_record_action(ctx, action);
8  }
9
10 char __license[] SEC("license") = "GPL";

```

Como se puede ver en el bloque 4.5, el programa es exactamente igual al desarrollado en el case01 (4.1.1) cambiando únicamente el código de retorno XDP. De esta forma se consigue modificar la funcionalidad del programa, a la deseada.

Compilación

Para compilar el programa XDP se ha dejado un Makefile preparado en este directorio al igual que en el case01 (4.1.1), por lo que para compilarlo únicamente hay que seguir las indicaciones del bloque 4.6.

Código 4.6: Compilación programa XDP - Case02

```

1  # En caso de no haber entrado en el directorio asignado del caso de uso
2  cd TFG/src/use_cases/xdp/case02
3
4
5  # Hacemos uso del Makefile suministrado
6  sudo make

```

Ahora bien, en el anterior caso de uso no se quiso incidir demasiado en este aspecto, pero ¿Cómo se produce la compilación de los programas XDP? Como ya se ha podido ver, los programas XDP están escritos en lo que llaman un lenguaje C restringido, en los cuales la secuencia de programa siempre empieza por “xdp”. Esto es así ya que, si no, el verificador del Kernel no podrá saber de qué tipo de *bytecode* se trata y lo rechazará.

Este código C restringido, se compilará haciendo uso de la herramienta **clang** como compilador de *frontend*, y de la herramienta **LLVM** como compilador de *backend*. De esta manera se generará un *bytecode* BPF, y posteriormente se almacenará en un objeto de tipo Executable and Linkable Format (ELF). Estos últimos serán los que se carguen en el Kernel (*.o). Es curioso el hecho de entender cómo pasamos de los hipotéticos programas XDP (C restringido) a *bytecode* BPF, ya que, cuando se indaga un poco más en XDP, se llega a la conclusión de que XDP se podría ver como un *framework* de BPF para trabajar a nivel de NIC.

Hay más factores que lo diferencian de los programas BPF, como son las estructuras de datos que manejan, además de los metadatos; pero al fin y al cabo, para anclar un programa XDP este debe antes ser traducido a un *bytecode* BPF.

Puesta en marcha del escenario

Para testear los programas XDP se hará uso de las *Network Namespaces* (más información en la sección 2.5.3). Como ya se comentaba, para que no suponga una barrera de entrada el concepto de las *Network Namespaces*, se ha dejado escrito un script para levantar el escenario, y para su posterior limpieza. Es importante señalar que el script debe ser lanzado con permisos de root. Para levantar el escenario se debe ejecutar dicho script como se indica en el bloque 4.7.

Para limpiar la máquina del escenario recreado anteriormente, se puede correr el mismo script indicándole ahora el parámetro **-c** (*Clean*). En el peor de los casos, y si se cree que la limpieza se ha realizado de manera satisfactoria, se puede llevar a cabo un reinicio de la máquina consiguiendo así que todos los entes no persistentes (Veth, netns..) desaparezcan del equipo.

Código 4.7: Puesta en marcha del escenario - Case02

```

1  # Para levantar el escenario (Importante hacerlo con permisos de super usuario)
2  sudo ./runenv.sh -i
3
4
5  # Una vez finalizado la comprobación del caso de uso, limpiaremos nuestra maquina:
6  sudo ./runenv.sh -c

```

El escenario que se va a manejar en este caso de uso es el siguiente (Ver figura 4.3), compuesto únicamente de una *Network Namespace* (**uno**) y un par de Veths (**veth0** – **uno**) para comunicar la *Network Namespace* creada con la *Network Namespace* por defecto.

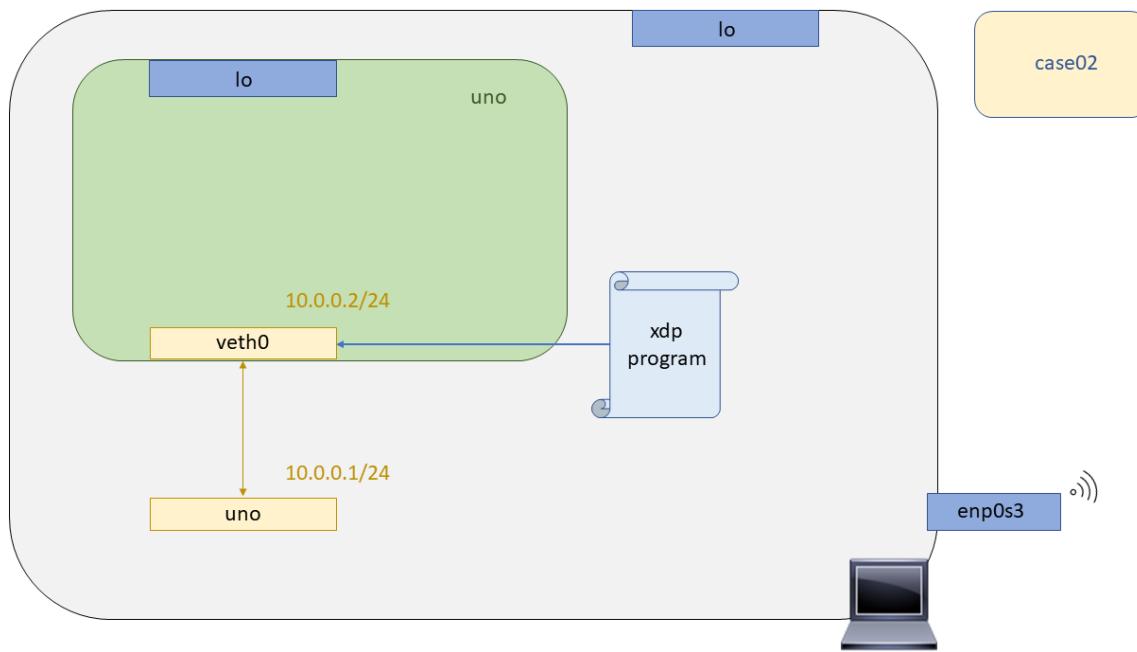


Figura 4.3: Escenario cableado del Case02 - XDP

Carga del programa XDP

Una vez que se tiene el escenario y el programa XDP compilado, se procederá a cargarlo en el Kernel. Para más información sobre el programa `xdp_loader`, qué aporta la librería `libbpf`, o por que no se hace uso de la herramienta `iproute2` para cargar los programas XDP en el Kernel, se recomienda regresar al caso01 (4.1.1) donde se intenta abordar todas estas cuestiones.

Como se comentaba, es hora de cargar el programa en el Kernel, esta vez por variar se va a cargar el programa XDP en el extremo de la Veth que se encuentra en el interior de la *Network Namespace*. Se podría haber hecho de la misma manera que en el caso de uso anterior y cargar el programa XDP en la Veth que se ve desde la *Network Namespace* por defecto, pero de esta manera se explorará como ejecutar comandos “dentro” de una *Network Namespace*. Para ejecutar comandos “dentro” de una *Network Namespace* se hará uso de la herramienta `iproute2` (C.2), más concretamente su módulo llamado **netns**. A la herramienta se le debe indicar el parámetro `exec` y el nombre de la *Network Namespace* dónde se quiera ejecutar el comando, y por último, se le indicará el comando que correrá “dentro” de la *Network Namespace*.

Código 4.8: Carga del programa XDP - Case02

```

1 # Cargaremos el programa sobre la interfaz "veth0"
2 sudo ip netns exec uno ./xdp_loader -d veth0 -F --progsec xdp_case02

```

Por lo que, entendiendo como funciona el módulo netns, y los parámetros suministrados al programa `xdp_loader`, explicados en el caso de uso `case01` (4.1.1), se podrá intuir que el resultado de la sentencia anterior es la de cargar el programa XDP en la interfaz llamada `veth0` contenida en la *Network Namespace uno*.

Comprobación del funcionamiento

La comprobación del funcionamiento del programa XDP anclado a la interfaz `veth0` se llevará a cabo testeando la conectividad entre el par de Veths. Puede que sea una prueba un poco simple, pero es suficiente, ya que únicamente se quiere verificar que el programa anclado en la interfaz está delegando los paquetes que le llegan, al *stack* de red.

En este punto se quiere comentar un aspecto de XDP, y es que, desde que se empezó a trabajar con XDP, se veía al *stack* de red de Linux como a un enemigo a batir, o “*by-passear*”; ya que con XDP se quiere conseguir definir de manera exclusiva el datapath que se desea implementar con un equipo que porte el Kernel de Linux. De esta manera, se consigue técnicamente un rendimiento superior ya que se quita de encima toda la redundancia y capas que no son necesarias para el procesamiento del hipotético datapath.

Ahora bien, en el transcurso del aprendizaje con esta tecnología se ha visto que en ocasiones trabajar de manera cooperativa con el *stack* de red puede reportar muchas facilidades y muchas otras funcionalidades ya implementadas en éste. No es cuestión de reinventar la rueda, más que nada porque el rendimiento que se puede llegar a ganar por hacer todo el procesamiento de manera exclusiva en la propia NIC, en opinión del autor, no compensa con la robustez y fiabilidad que tendrá dicha funcionalidad en el *stack* de red de Linux.

Como se puede apreciar en la figura 4.4, prestando atención al ping  (Fig. 4.4a), hay perfecta conectividad. Eso significa que el programa XDP está delegando correctamente los paquetes al *stack* de red, se puede verificar dicho funcionamiento atendiendo a las estadísticas sobre los códigos de retorno de la figura 4.4b, dónde el código de retorno `XDP_PASS` va en aumento. Dichas estadísticas han sido recolectadas haciendo uso del binario `xdp_stats`. En el siguiente caso de uso se explicará brevemente el funcionamiento de la recolección de estadísticas de los programas XDP.

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case02$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.114 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.084 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.058 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.079 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.077 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.083 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.082 ms
^C
--- 10.0.0.2 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8204ms
rtt min/avg/max/mdev = 0.058/0.082/0.114/0.015 ms
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case02$
```

(a) Ejecución de ping hacia la interfaz con el programa XDP

```
root@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case02# sudo ./xdp_stats -d veth0
Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:21 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251584
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251542
XDP_PASS         7 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251518
XDP_TX           0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251496
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251475

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001442
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001467
XDP_PASS         9 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.001488
XDP_TX           0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001506
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001526
^C
root@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case02#
```

(b) Estadísticas de los códigos de retorno XDP

Figura 4.4: Comprobación de funcionamiento del Case02 - XDP

4.1.3. Case03 - Echo server

En este caso de uso se hará un especial hincapié en el análisis de paquetes, su filtrado y manejo. En los anteriores caso de uso se definía un comportamiento de los paquetes haciendo un uso exclusivo de los códigos de retorno XDP, más concretamente `XDP_DROP` para tirar los paquetes y `XDP_PASS` para admitir los paquetes. Hay más códigos de retorno XDP pero con ellos no se puede lograr desarrollar todas las lógicas posibles. Como ya se comentaba en la sección 2.4, estos códigos se encuentran definidos en el archivo de cabecera `bpf.h`. En la tabla 4.2 se pueden recordar y ver qué funcionalidad aportaba cada uno de ellos.

Ahora bien, ¿Cómo se puede implementar una lógica más avanzada? Esto se podrá conseguir filtrando los paquetes, y en base al tipo de paquete aplicar unas acciones u otras haciendo uso de los códigos de retorno XDP. Para filtrar paquetes, se tendrá que hacer uso de las estructuras de datos de los protocolos de red definidas en el Kernel de Linux. Además de hacer numerosas comprobaciones de límites de acceso a memoria, para que el verificador del Kernel

Código de Retorno	Comportamiento
XDP_PASS	Admitir el paquete, pasarselo al <i>stack</i> de red
XDP_DROP	Tirar el paquete
XDP_ABORTED	Tirar el paquete y generar una <code>xdp:xdp_exception</code> , útiles para depurar
XDP_REDIRECT	Utilizado cuando se realiza un forwarding del paquete a otra interfaz
XDP_TX	Re-transmitir el paquete por la misma interfaz por la cual se ha recibido el paquete

Tabla 4.2: Resumen sobre los códigos de retorno XDP

no nos tire el paquete por un acceso indebido según se comentaba en las limitaciones XDP (Sección 2.4). Las estructuras de datos que han sido utilizadas para filtrar paquetes en este caso de uso se dejan indicadas en la tabla 4.3.

Estructura	Archivo de cabecera
<code>struct ethhdr</code>	<code><linux/if_ether.h></code>
<code>struct ipv6hdr</code>	<code><linux/ipv6.h></code>
<code>struct iphdr</code>	<code><linux/ip.h></code>
<code>struct icmp6hdr</code>	<code><linux/icmpv6.h></code>
<code>struct icmphdr</code>	<code><linux/icmp.h></code>

Tabla 4.3: Estructuras de datos para procesar las cabeceras de los paquetes

Es importante señalar que los paquetes vienen por la red en un *byte order* denominado como *Network byte order*. Por esta razón, se necesitará traducirlo al *byte order* usado por la máquina (*Host order*) en el caso de que se quiera comprobar o hacer uso del valor de algunos de sus campos. Para ello, se hará uso de las funciones `bpf_ntohs()` y `bpf_htons()` respectivamente.

Por lo tanto, sabiendo qué estructuras de datos utilizar para el filtrando los paquetes, se filtrarán todos los paquetes de tipo ICMP - ICMPv6 con un código ICMP-ECHO. El resto de paquetes se le delegarán al *stack* de red para que los maneje. En cuanto a los paquetes ICMP filtrados, se contestarán automáticamente desde el programa XDP desarrollado. Esto se conseguirá en la propia NIC, cambiando el ICMP-ECHO por un ICMP-REPLY, cambiando MACs, y por último, actualizando el *checksum* de la cabecera ICMP. Como en este caso el paquete debe ser reenviado por la misma interfaz por la cual se recibió, se hará uso de código de retorno XDP_TX.

Compilación

Para compilar el programa XDP se ha dejado un Makefile preparado en este directorio al igual que en el case02 (4.1.2), por lo que para compilarlo únicamente hay que seguir las indicaciones del bloque 4.9.

```
Código 4.9: Compilación programa XDP - Case03
```

```

1 # En caso de no haber entrado en el directorio asignado del caso de uso
2 cd TFG/src/use_cases/xdp/case03
3
4
5 # Hacemos uso del Makefile suministrado
6 sudo make

```

Para más información sobre el proceso de compilación del programa XDP, recomendamos que vuelva al case02 (4.1.2) donde se hace referencia al flujo dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para comprobar el funcionamiento de los programas XDP se hará uso de nuevo de las *Network Namespace* (más información en la sección 2.5.3). Como ya se comentaba, para que no suponga una barrera de entrada el concepto de las *Network Namespace*, se ha dejado escrito un script para levantar el escenario, y para su posterior limpieza. Es importante señalar que el script debe ser lanzado con permisos de root. Para levantar el escenario debemos ejecutar dicho script como se indica en el bloque 4.10. Para limpiar la máquina del escenario recreado anteriormente, se puede correr el mismo script indicándole ahora el parámetro -c (*Clean*). En el peor de los casos, y si se cree que la limpieza se no se ha realizado de manera satisfactoria, se puede llevar a cabo un reinicio de la máquina consiguiendo así que todos los entes no persistentes (Veth, netns..) desaparezcan del equipo.

Código 4.10: Puesta en marcha del escenario - Case03

```

1 # Para levantar el escenario (Importante hacerlo con permisos de super usuario)
2 sudo ./runenv.sh -i
3
4
5 # Una vez finalizado la comprobación del caso de uso, limpiaremos nuestra máquina:
6 sudo ./runenv.sh -c

```

El escenario que se va a manejar en este caso de uso es el siguiente (Ver figura 4.5), compuesto únicamente de una *Network Namespace* (uno) y un par de Veths (veth0 – uno) para comunicar la *Network Namespace* creada con la *Network Namespace* por defecto.

Carga del programa XDP

Una vez que se tiene el escenario levantado y el programa XDP compilado, se procederá a cargarlo en el Kernel. Para más información sobre el programa xdp_loader, qué aporta la librería libbpf, o por que no se hace uso de la herramienta iproute2 para cargar los programas XDP en el Kernel, se recomienda regresar al case01 (4.1.1) donde se intenta abordar todas estas cuestiones. De forma adicional, es interesante comentar que se va hacer uso del módulo **netns** de la herramienta iproute2, si tiene alguna duda sobre dicho módulo le recomendamos que consulte su *man-page* o vuelva al case02 (4.1.2) donde se hace una pequeña introducción sobre éste, y su funcionamiento básico para ejecutar comandos “dentro” de una *Network*

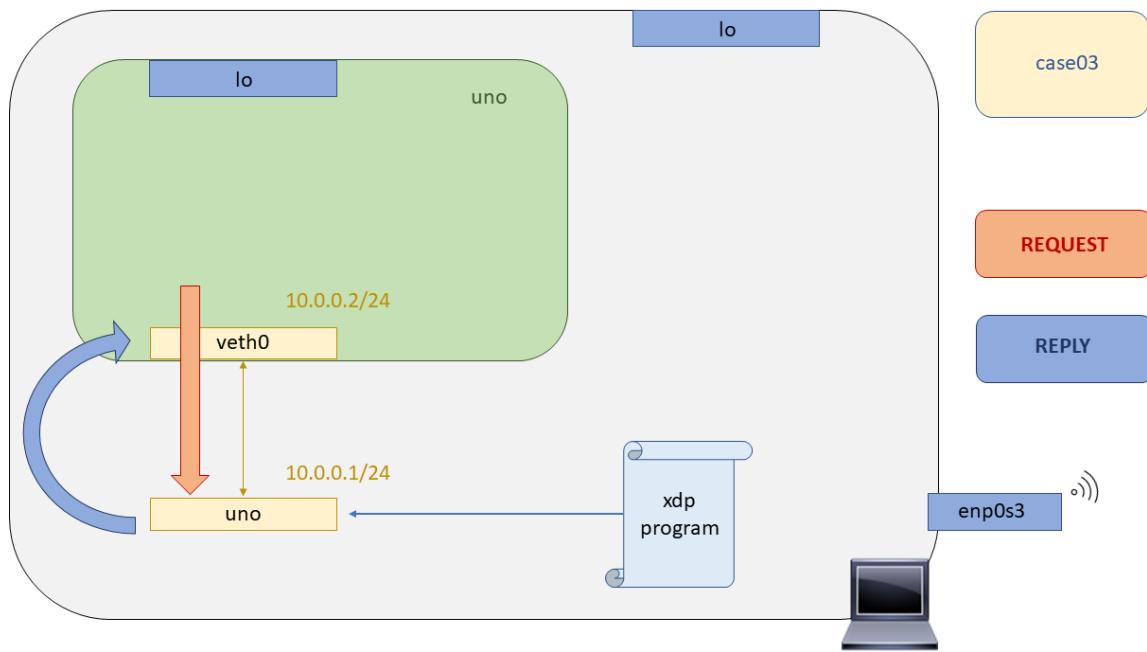


Figura 4.5: Escenario cableado del Case03 - XDP

Namespace.

Código 4.11: Carga del programa XDP - Case03

```

1  # Anclamos el programa XDP (xdp_pass) en la interfaz veth0, perteneciente a la Network ↵
   ↵ Namespace "uno"
2  sudo ip netns exec uno ./xdp_loader -d veth0 -F --progsec xdp_pass
3
4  # Anclamos el programa XDP en la interfaz uno, perteneciente a la Network Namespace por defecto
5  sudo ./xdp_loader -d uno -F --progsec xdp_case03

```

En este caso de uso se anclará el programa XDP a validar en la Veth exterior, por lo que las pruebas vendrán inducidas desde “dentro” de la *Network Namespace* uno. Para anclar el programa se ha hecho uso de nuevo del programa `xdp_loader`. Es importante señalar que se ha tenido que anclar un *dummy program* que permite pasar todos los paquetes a la Veth destino, esta es una limitación propia por trabajar con Veths y XDP, de momento se trata de una limitación de implementación, pero puede que a un corto plazo esta limitación se vea ya superada.

Para más información sobre esta limitación recomendamos ver la charla de la Netdev llamada “*Veth: XDP for containers*”³ donde explican con un mayor detalle la misma, cómo abordarla y por qué está inducida.

Comprobación del funcionamiento

³<https://netdevconf.info/0x13/session.html?talk-veth-xdp>

La comprobación del funcionamiento del programa XDP anclado a la interfaz `uno` se llevará a cabo generando pings desde “dentro” la *Network Namespace uno* hacia afuera. De esta forma la interfaz `uno` los filtrará, analizará y generará una respuesta.

El funcionamiento del programa `xdf_stats` es muy simple, ya que desde el programa anclado en el Kernel se genera un mapa BPF del tipo clave-valor, donde las claves son los distintos códigos de retorno y los valores son contadores. Después, desde el espacio de usuario el programa `xdf_stats`, sabiendo el nombre del mapa BPF, y dónde se almacena, va a buscarlo. Para ello, abre el descriptor de archivo asociado a ese mapa almacenado en el path `/sys/fs/bpf/`. Una vez que tiene abierto el descriptor de archivo, este irá leyendo por clave del mapa todas las estadísticas sobre los códigos de retorno de forma periódica.

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case03$ sudo ip netns exec uno ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.058 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.043 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.059 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.047 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.059 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.058 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=0.062 ms
64 bytes from 10.0.0.1: icmp_seq=9 ttl=64 time=0.057 ms
64 bytes from 10.0.0.1: icmp_seq=10 ttl=64 time=0.060 ms
64 bytes from 10.0.0.1: icmp_seq=11 ttl=64 time=0.057 ms
^C
--- 10.0.0.1 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10245ms
rtt min/avg/max/mdev = 0.038/0.054/0.062/0.009 ms
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case03$
```

(a) Ejecución de ping hacia la interfaz con el programa XDP

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case03$ sudo ./xdp_stats -d uno
Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:39 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251171
XDP_DROP          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251111
XDP_PASS          5 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251085
XDP_TX           4 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251061
XDP_REDIRECT      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.251038

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.009977
XDP_DROP          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.009978
XDP_PASS          5 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.010120
XDP_TX           6 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.010130
XDP_REDIRECT      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.010132

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000549
XDP_DROP          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000551
XDP_PASS          5 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000409
XDP_TX           8 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.000426
XDP_REDIRECT      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000426

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001245
XDP_DROP          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001266
XDP_PASS          5 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001269
XDP_TX           10 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.001244
XDP_REDIRECT      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001242

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000479
XDP_DROP          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000464
XDP_PASS          5 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000463
XDP_TX           11 pkts (      0 pps)      1 Kbytes (      0 Mbits/s) period:2.000478
XDP_REDIRECT      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.000479
```

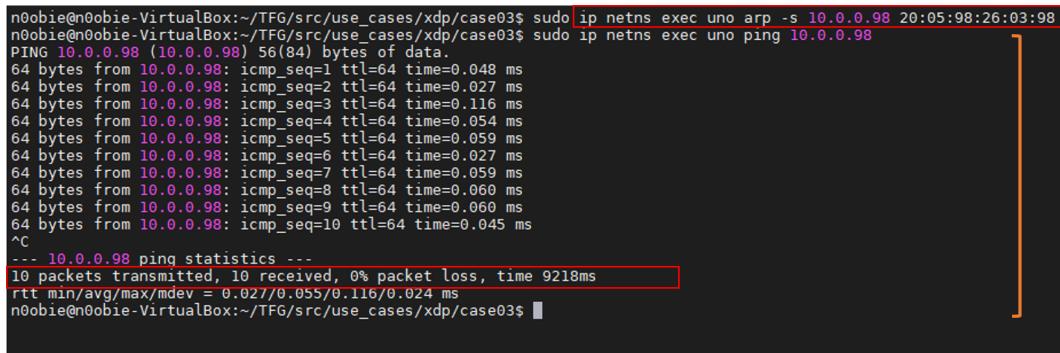
(b) Estadísticas de los códigos de retorno XDP

Figura 4.6: Comprobación de funcionamiento del Case03 - XDP

Si todo funciona correctamente se debería ver como los códigos de retorno mayormente

empleados son los de XDP_TX siempre y cuando no se haya detenido el ping desde dentro de la *Network Namespace*. Como se puede apreciar en la figura 4.6, el ping  está siendo contestado correctamente por el programa XDP dado que los códigos XDP_TX van aumentando.

De forma adicional, se va a comprobar cómo todos los pings que se manden desde dentro de la *Network Namespace* uno serán contestados. Esto es así, ya que el programa XDP, contestará todos los ECHO-REQUEST que le lleguen independientemente de si van dirigidos a él. En la figura 4.7 se puede ver cómo funciona un ping  a una máquina inexistente. Para ello, se ha tenido que añadir una MAC inventada en la tabla ARP asociada a la IP a la cual vamos hacer ping para que no se iniciara una resolución ARP. En el caso de que se iniciara una resolución ARP, el ping estaría bloqueado a la espera de completar una resolución ARP sobre una máquina inexistente.



```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case03$ sudo ip netns exec uno arp -s 10.0.0.98 20:05:98:26:03:98
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case03$ sudo ip netns exec uno ping 10.0.0.98
PING 10.0.0.98 (10.0.0.98) 56(84) bytes of data.
64 bytes from 10.0.0.98: icmp_seq=1 ttl=64 time=0.048 ms
64 bytes from 10.0.0.98: icmp_seq=2 ttl=64 time=0.027 ms
64 bytes from 10.0.0.98: icmp_seq=3 ttl=64 time=0.116 ms
64 bytes from 10.0.0.98: icmp_seq=4 ttl=64 time=0.054 ms
64 bytes from 10.0.0.98: icmp_seq=5 ttl=64 time=0.059 ms
64 bytes from 10.0.0.98: icmp_seq=6 ttl=64 time=0.027 ms
64 bytes from 10.0.0.98: icmp_seq=7 ttl=64 time=0.059 ms
64 bytes from 10.0.0.98: icmp_seq=8 ttl=64 time=0.060 ms
64 bytes from 10.0.0.98: icmp_seq=9 ttl=64 time=0.060 ms
64 bytes from 10.0.0.98: icmp_seq=10 ttl=64 time=0.045 ms
^C
--- 10.0.0.98 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9218ms
rtt min/avg/max/mdev = 0.027/0.055/0.116/0.024 ms
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case03$
```

Figura 4.7: Ping a máquina inexistente - XDP

4.1.4. Case04 - Layer 3 forwarding

En este caso de uso se explorará como realizar forwarding de paquetes. En este punto, ya se ha revisado cómo filtrar por las cabeceras de los paquetes, analizarlos y establecer una lógica asociada a ese filtrado con los códigos de retorno XDP. Una acción adicional a realizar con los paquetes será el forwarding. En XDP vendrá implementado por códigos de retorno y por *helpers* BPF porque, como ya comentábamos en el case02 (4.1.2), XDP se termina traduciendo en BPF (eBPF), por lo que ciertas funciones, no todas, para trabajar con BPF están disponibles a la hora de trabajar con XDP.

A lo largo de este caso de uso, se han explorado las distintas maneras para lograr el forwarding con XDP. Se ha ido desde la manera más simple a la manera más robusta y, a su vez, compleja. Para que no haya diferencias entre las distintas formas de realizar el forwarding, se ha creado un mismo escenario donde se explorarán estas vías sin que existan diferencias inducidas por este. En el caso de uso anterior ya se estaba haciendo un forwarding, ya que, con el código XDP_TX se realiza un forwarding hacia la interfaz por la cual se recibió dicho paquete. Pero, ¿cómo se hace un forwarding hacia otras interfaces? Leyendo la *man-page* de los *helpers* BPF se han encontrado las funciones `bpf_redirect()`, `bpf_redirect_map()`, las cuales, leyendo su descripción, serán la vía utilizada para abordar esta necesidad.

Código 4.12: Helper BPF para realizar Forwarding - Case04

```

1 int bpf_redirect(u32 ifindex, u64 flags);
2
3 int bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags);

```

Compilación

Para compilar el programa XDP se ha dejado un Makefile preparado en este directorio al igual que en el case03 (4.1.3), por lo que para compilarlo únicamente hay que seguir las indicaciones del bloque 4.13.

Código 4.13: Compilación programa XDP - Case04

```

1 # En caso de no haber entrado en el directorio asignado del caso de uso
2 cd TFG/src/use_cases/xdp/case04
3
4 # Hacemos uso del Makefile suministrado
5 sudo make

```

Podrá encontrar más información sobre el proceso de compilación de los programas XDP en el case02 (4.1.2).

Puesta en marcha del escenario

Para comprobar el funcionamiento de los programas XDP se hará uso de las *Network Namespace* (más información en la sección 2.5.3). Como ya se comentaba, para que no suponga una barrera de entrada el concepto de las *Network Namespace*, se ha dejado escrito un script para levantar el escenario, y para su posterior limpieza. Es importante señalar que el script debe ser lanzado con permisos de root. Para levantar el escenario debemos ejecutar dicho script como se indica en el bloque 4.10. Para limpiar la máquina del escenario recreado anteriormente, se puede correr el mismo script indicándole ahora el parámetro **-c** (*Clean*). En el peor de los casos, y si se cree que la limpieza se no se ha realizado de manera satisfactoria, se puede llevar a cabo un reinicio de la máquina consiguiendo así que todos los entes no persistentes (Veth, netns..) desaparezcan del equipo.

Código 4.14: Puesta en marcha del escenario - Case04

```

1 # Para levantar el escenario (Importante hacerlo con permisos de super usuario)
2 sudo ./runenv.sh -i
3
4
5 # Una vez finalizado la comprobación del caso de uso, limpiaremos nuestra maquina:
6 sudo ./runenv.sh -c

```

4.1.4.1. Hardcoded forwarding

La primera forma de implementación del forwarding es lo que denominaremos como Hard-coded forwarding, dado que es necesario hardcodear⁴ información de forwarding en el propio programa XDP. El escenario levantado se puede apreciar en la figura 4.8, está compuesto de dos *Network Namespace* (*uno* y *dos*), y de dos pares de Veths (*veth0* – *uno* y *veth0* – *dos*) las cuales se utilizarán para comunicar las dos *Network Namespaces* entre sí, a través del la *Network Namespace* por defecto. En este caso el forwarding lo haremos desde la interfaz *dos* hacia la interfaz *uno*.

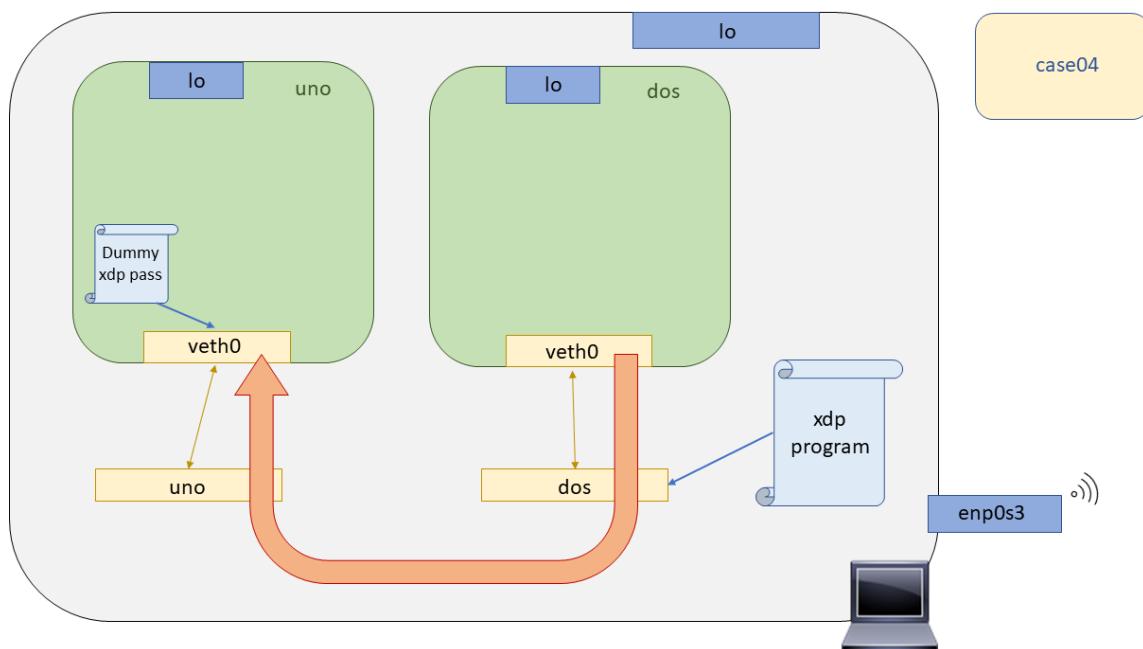


Figura 4.8: Escenario cableado Hardcoded forwarding del Case04 - XDP

Carga del programa XDP

Antes de realizar la carga del programa se deben obtener **dos datos**, la *ifindex* de la interfaz uno a la cual se van a mandar los paquetes generados desde el interior de la *Network Namespace* dos, y la MAC de la interfaz interna de la *Network Namespace* uno, ya que será necesario que los paquetes que se dirijan a la interfaz uno lleven como MAC destino la de la *veth0*, para que así los paquetes no sean descartados.

Una vez se tengan estos datos anotados se abrirá el programa XDP (*.c) con cualquier editor de texto, se irá a la declaración de variables y se hardcodeará como se puede ver en el bloque 4.15 tanto el *ifindex* como la MAC.

⁴Hardcodear: perder flexibilidad y/o prolijidad dejando valores y/o comportamientos fijos en el código del programa.

Código 4.15: Ejemplo MAC e Ifindex - Case04

```

1  /* Para un ifindex: 6 y una MAC: 9A:DE:AF:EC:18:6E */
2
3  ...
4
5  unsigned char dst[ETH_ALEN + 1] = {0x9a,0xde,0xaf,0xec,0x18,0x6e, '\0'} ;
6  unsigned ifindex = 6;
7
8  ...

```

Una vez que se tenga hardcodeado los datos para realizar el forwarding se debe recomilar el programa XDP para que el *bytecode* que se ancle a la interfaz dos haga correctamente el forwarding. Por ello, simplemente se tiene que hacer un *make* nuevamente.

Por lo tanto, teniendo todo preparado es hora de anclar de nuevo el programa XDP. Recordemos que por el estar trabajando con interfaz Veths se debe anclar un *dummy program*⁵ en el extremo donde se vayan a recibir los paquetes.

Código 4.16: Carga del programa XDP Hardcoded forwarding - Case04

```

1  # Entramos a la Network Namespace "uno" y anclamos el dummy program a la interfaz veth0
2  sudo ip netns exec uno ./xdp_loader -d veth0 -F --progsec xdp_pass
3
4  # Acto seguido, anclamos el programa a la interfaz "dos" como ya comentábamos antes
5  sudo ./xdp_loader -d dos -F --progsec xdp_case04

```

Comprobación del funcionamiento

La comprobación de funcionamiento de este programa XDP es bastante básica, se va a generar paquetes desde el interior de la *Network Namespace* dos hacia la Veth interna de la *Network Namespace* uno. Si todo funciona correctamente deberían llegar los paquetes en el sentido dispuesto del forwarding hardcodeado, sería una comunicación unidireccional. Para ello se abrirán tres terminales, en cada una de ellas se llevará a cabo una tarea. Ver bloque 4.17, donde se indican todos los comandos necesarios para comprobar el funcionamiento del Hardcoded forwarding.

Código 4.17: Comprobación del funcionamiento Hardcoded forwarding - Case04

```

1  # En esta terminal generaremos el ping hacia la interfaz de la Network Namespace "uno" desde la ↵
   ↵ Network Namespace "dos"
2  [Terminal:1] sudo ip netns exec dos ping {IP_veth0_uno}
3
4  # En esta terminal pondremos a un sniffer a escuchar los paquetes que nos lleguen dentro de la ↵
   ↵ Network Namespace "dos"
5  [Terminal:2] sudo ip netns exec uno tcpdump -l
6
7  # Por último, opcionalmente, podemos ejecutar el programa que actuaba como recolector de estadísticas sobre los códigos de retorno XDP
8  [Terminal:3] sudo ./xdp_stats -d dos

```

Como se puede apreciar en la figura 4.9, el ping █ no llega a completarse. Esto se debe a que al implementar una comunicación unidireccional, el ping se queda bloqueado en la resolución

⁵<https://netdevconf.info/0x13/session.html?talk-veth-xdp>

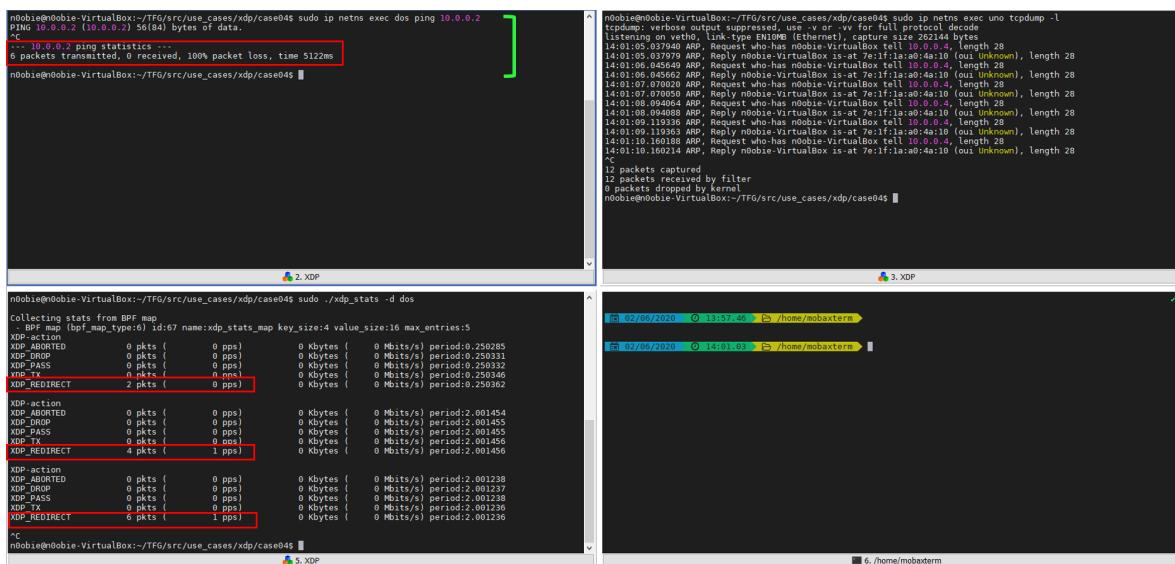


Figura 4.9: Comprobación de funcionamiento Hardcoded forwarding del Case04 - XDP

ARP. Se puede corroborar este funcionamiento atendiendo a los códigos de retorno XDP, `XDP_REDIRECT`, como van en aumento debido a los reenvíos de los paquetes ARP-Request de una interfaz a la otra y por la escucha de `tcpdump` en la terminal superior derecha.

4.1.4.2. Semi-Hardcoded forwarding (BPF maps)

La segunda forma de implementar el forwarding se denominará Semi-Hardcoded forwarding, ya que la información irá hardcodeada, pero no en el propio programa XDP, sino, en los mapas BPF. El escenario levantado se puede apreciar en la figura 4.10, está compuesto de dos *Network Namespace* (**uno** y **dos**), y de dos pares de Veths (**veth0 – uno** y **veth0 – dos**) las cuales se utilizarán para comunicar las dos *Network Namespaces* entre sí, a través del la *Network Namespace* por defecto. En este caso el forwarding se hará desde la interfaz **dos** hacia la interfaz **uno** y viceversa, por lo que la comunicación será bidireccional.

Carga del programa XDP

Esta manera de hacer el forwarding no requiere de hardcodear datos en el propio programa XDP que irá al Kernel, si no, que se usarán los mapas BPF como medio para guardar datos de forwarding como son las *ifindex* y las MAC destino desde espacio de usuario. De esta forma, posteriormente el programa anclado en el Kernel sea capaz de leer los mapas, obtener la información de forwarding y realizarlo en base a la información percibida de los mapas BPF.

De nuevo, y como en este caso la comunicación será bidireccional se debe anclar dos *dummy program* en los dos extremos donde van a llegar los paquetes, si no se está al tanto de esta limitación vuelve a la subsección (4.1.4.1) donde se menciona la limitación.

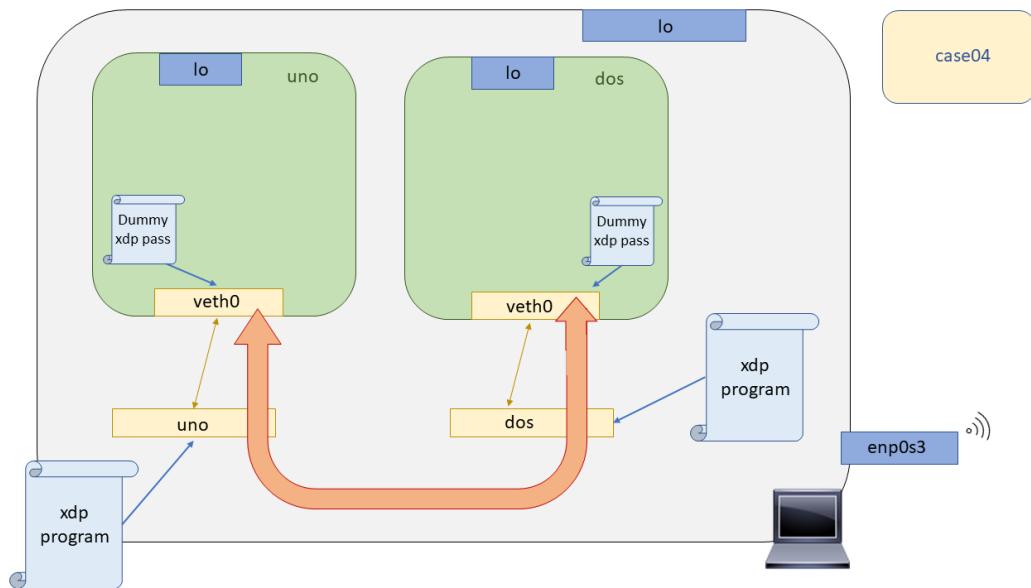


Figura 4.10: Escenario cableado Semi-Hardcoded forwarding del Case04 - XDP

Es importante señalar que los programas anclados previamente deben ser retirados, por lo que una opción sería hacer un *clean* del escenario haciendo uso del script dado (`sudo ./runenv.sh -c`) y empezar de nuevo.

Código 4.18: Carga del programa XDP Semi-Hardcoded forwarding - Case04

```

1  # Entramos en cada Network Namespace y anclamos los "dummy programs"
2  sudo ip netns exec uno ./xdp_loader -d veth0 -F --progsec xdp_pass
3  sudo ip netns exec dos ./xdp_loader -d veth0 -F --progsec xdp_pass
4
5  # Anclamos el programa XDP en cada interfaz para conseguir un comunicacion bidireccional
6  sudo ./xdp_loader -d uno -F --progsec xdp_case04_map
7  sudo ./xdp_loader -d dos -F --progsec xdp_case04_map
8
9  # Almacenamos la informacion necesaria para realizar el forwarding
10 src="uno"
11 dest="dos"
12 src_mac=$(sudo ip netns exec $src cat /sys/class/net/veth0/address)
13 dest_mac=$(sudo ip netns exec $dest cat /sys/class/net/veth0/address)
14
15 # Populamos los mapas BPF con la informacion util para llevar a cabo el forwarding en ambas ↵
16   ↵ direcciones
16 ./prog_user -d $src -r $dest --src-mac $src_mac --dest-mac $dest_mac
17 ./prog_user -d $dest -r $src --src-mac $dest_mac --dest-mac $src_mac

```

Comprobación del funcionamiento

La comprobación de funcionamiento de este programa puede ser llevada a cabo desde un extremo u otro debido a que, si todo funciona correctamente, existirá una comunicación bi-

direccional. Por lo que, en este caso se harán las pruebas desde la *Network namespace uno* hacia la *dos*.

Como se puede apreciar en la figura 4.11, existe una comunicación bidireccional ya que el ping  se ejecuta con normalidad. Además, en la subfigura 4.11b se puede ver como los códigos de retorno XDP de forwarding van aumentando, por lo que el programa XDP está funcionando según lo previsto.

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$ sudo ip netns exec dos ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.100 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.109 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.095 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.103 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.049 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.049 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.087 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.100 ms
^C
--- 10.0.0.2 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8182ms
rtt min/avg/max/mdev = 0.049/0.086/0.109/0.021 ms
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$
```

(a) Ejecución de ping hacia la interfaz con el programa XDP

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$ sudo ./xdp_stats -d uno
Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:95 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.260287
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.260312
XDP_PASS         2 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.260332
XDP_TX           0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.260349
XDP_REDIRECT     3 pkts (      4 pps)      0 Kbytes (      0 Mbits/s) period:0.260353

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006800
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006799
XDP_PASS         2 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006799
XDP_TX           0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006799
XDP_REDIRECT     4 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006799

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.003486
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.003487
XDP_PASS         2 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.003488
XDP_TX           0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.003488
XDP_REDIRECT     6 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.003488

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001165
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001165
XDP_PASS         2 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001164
XDP_TX           0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.001164
XDP_REDIRECT     9 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.001164

^C
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$
```

(b) Estadísticas de los códigos de retorno XDP

Figura 4.11: Comprobación de funcionamiento Semi-Hardcoded forwarding del Case04 - XDP

4.1.4.3. Forwarding auto (Kernel FIBs)

La tercera forma de hacer forwarding hace referencia al forwarding automático, esto es debido a que no habrá ningún tipo de información de forwarding hardcodeada en el programa XDP, la información se obtendrá del propio *stack* de red trabajando de forma cooperativa con éste. El escenario sobre el cual se trabajará será el mismo que el anterior por lo que únicamente es necesario preocuparse de limpiar el escenario de los programas XDP anteriormente anclados a cada interfaz para que no interfieran con los nuevos programas XDP que se van anclar.

En este caso, se va a ir un paso más allá y el forwarding será automático, es decir, no se hardcodeará ningún tipo de información para hacer el forwarding a los paquetes. Pero, entonces: ¿cómo se sabrá a dónde hay que mandar los paquetes? Esta información se conseguirá del *stack* de red del Kernel de Linux el cual tiene una FIB (*Forwarding Information Base*) con reglas muy útiles las cuales se pueden sacar partido.

Por lo que, se realizará una consulta a la FIB con el helper `bpf_fib_lookup()` para obtener información de forwarding desde el propio *stack* de red, este es un claro ejemplo donde la cooperación con el *stack* de red hace que el programa XDP sea más robusto e independiente del espacio de usuario.

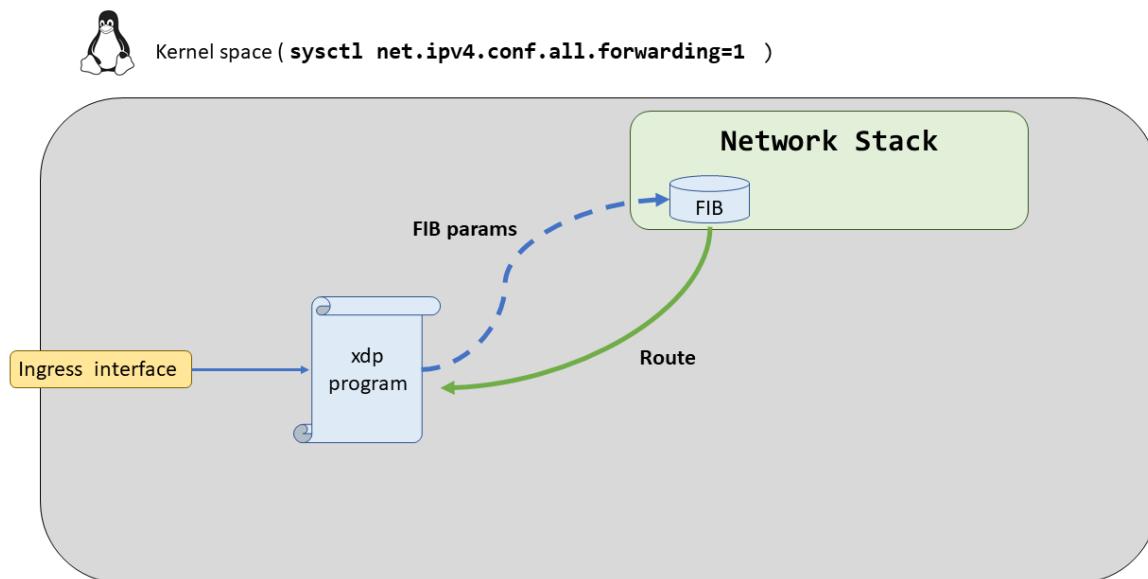


Figura 4.12: Escenario cableado Forwarding auto del Case04 - XDP

Carga del programa XDP

Para la carga del programa XDP se deberá primero habilitar el forwarding en nuestro Kernel, acto seguido se anclarán los *dummy program* y por último, se anclarán anclar los programas XDP en ambas interfaces tanto uno como dos para conseguir que la comunicación sea bidireccional.

Código 4.19: Carga del programa XDP Forwarding auto - Case04

```

1  # Habilitamos el forwarding
2  sudo sysctl net.ipv4.conf.all.forwarding=1
3
4  # Anclamos los programas XDP a cada interfaz
5  sudo ./xdp_loader -d uno -F --progsec xdp_case04_fib
6  sudo ./xdp_loader -d dos -F --progsec xdp_case04_fib
7
8  # Ahora añadimos a cada interfaz destino su "dummy program"
9  sudo ip netns exec uno ./xdp_loader -d veth0 -F --progsec xdp_pass
10 sudo ip netns exec dos ./xdp_loader -d veth0 -F --progsec xdp_pass
11
12 # Habilitamos los ifindex
13 sudo ./prog_user -d uno
14 sudo ./prog_user -d dos

```

Comprobación del funcionamiento

La comprobación de funcionamiento de este programa puede ser llevada a cabo desde un extremo u otro debido a que, si todo funciona correctamente, existirá una comunicación bidireccional y completamente automática ya que no ha almacenado ningún tipo de información en los programas anclados. Por lo que, se harán las pruebas desde la *Network namespace uno* hacia la *dos*. Como se puede apreciar en la figura 4.13, atendiendo al funcionamiento del ping [■], hay una perfecta comunicación entre ambas *Network namespaces*, por lo que el programa está tomando correctamente las rutas de la FIB.

Código 4.20: Comprobación del funcionamiento Forwarding auto - Case04

```

1  # Hacemos un ping desde el interior de la Network namespace "uno" hacia la veth0 de la Network ↵
   ↵ namespace "dos"
2  ping {IP_veth_dos} [ y viceversa...]
3
4  # Comprobamos con el recolector de estadísticas que se están produciendo XDP_REDIRECT
5  sudo ./xdp_stats -d uno

```

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$ sudo ip netns exec uno ping 10.11.2.2
PING 10.11.2.2 (10.11.2.2) 56(84) bytes of data.
64 bytes from 10.11.2.2: icmp_seq=1 ttl=63 time=0.064 ms
64 bytes from 10.11.2.2: icmp_seq=2 ttl=63 time=0.068 ms
64 bytes from 10.11.2.2: icmp_seq=3 ttl=63 time=0.088 ms
64 bytes from 10.11.2.2: icmp_seq=4 ttl=63 time=0.112 ms
64 bytes from 10.11.2.2: icmp_seq=5 ttl=63 time=0.101 ms
64 bytes from 10.11.2.2: icmp_seq=6 ttl=63 time=0.097 ms
64 bytes from 10.11.2.2: icmp_seq=7 ttl=63 time=0.101 ms
64 bytes from 10.11.2.2: icmp_seq=8 ttl=63 time=0.087 ms
64 bytes from 10.11.2.2: icmp_seq=9 ttl=63 time=0.102 ms
64 bytes from 10.11.2.2: icmp_seq=10 ttl=63 time=0.042 ms
^C
--- 10.11.2.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9205ms
rtt min/avg/max/mdev = 0.042/0.086/0.112/0.021 ms
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$
```

(a) Ejecución de ping hacia la interfaz con el programa XDP

```
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$ sudo ./xdp_stats -d uno
[sudo] contraseña para n0obie:

Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:103 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0.257409
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0.257415
XDP_PASS         2 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0.257414
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0.257398
XDP_REDIRECT     5 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0.257398

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000425
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000429
XDP_PASS         2 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000431
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000432
XDP_REDIRECT     7 pkts (    1 pps)      0 Kbytes (    0 Mbits/s) period:2.000434

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000914
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000913
XDP_PASS         2 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000907
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.000902
XDP_REDIRECT     9 pkts (    1 pps)      0 Kbytes (    0 Mbits/s) period:2.000896

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.001884
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.001879
XDP_PASS         2 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.001883
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.001886
XDP_REDIRECT    10 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2.001890

^C
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case04$ ~
```

(b) Estadísticas de los códigos de retorno XDP

Figura 4.13: Comprobación de funcionamiento Forwarding auto del Case04 - XDP

4.1.5. Case05 - Broadcast

Por último, en este caso de uso se explorará la capacidad de reenvío a múltiples interfaces de XDP. Por ello, se ha intentado replicar un escenario básico de broadcast con *Network Namespaces*. Se ha planteado hacer uso de la herramienta arping para emular una resolución ARP, generando paquetes ARP-Request. Dichos paquetes llevarán su MAC destino todo a FF:FF:FF:FF:FF y su dominio de difusión englobaría todos aquellos nodos de la red que operen hasta capa 2. Como por ejemplo un hub, o un switch. En la figura 4.14 se puede encontrar el escenario a recrear en este caso de uso.

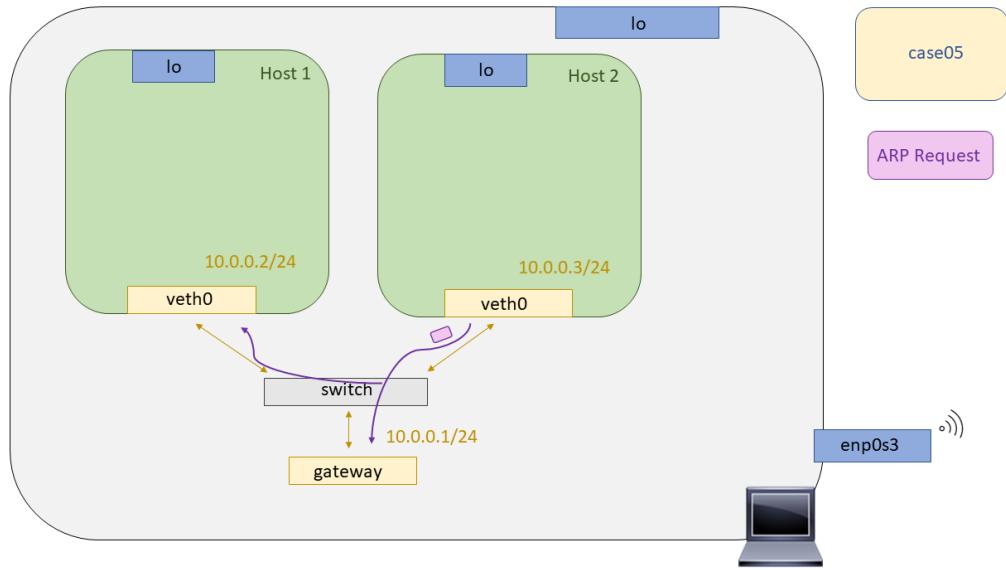


Figura 4.14: Escenario a recrear del Case05 - XDP

El escenario propuesto para emular dicho escenario ha sido el siguiente (Ver figura 4.15). Este estaría compuesto de tres *Network Namespaces* replicando así cada una de ellas un nodo independiente de la red, después, para intercomunicar cada “nodo” de la red, se ha hecho uso de Veths. El supuesto switch será la *Network Namespace* llamada **switch**, la cual requerirá de un programa XDP para poder actuar como tal, ya que de no ser así sus interfaces tendrán todo el *stack* de red de Linux por encima de ellas. Es decir, el nodo implementará todas las capas del modelo TCP/IP (DoD), replicando así la funcionalidad de un hipotético host y no actuando como un switch.

Para realizar el broadcast se indagó sobre los *helpers BPF* en busca de alguna función que ayudara a satisfacer la necesidad, y se encontró una función (4.21) que a primera vista se creía que podía ser de utilidad.

Código 4.21: Helper BPF para realizar un Broadcast - Case05

```
1 int bpf_clone_redirect(struct sk_buff *skb, u32 ifindex, u64 flags);
```

Pero hubo un pequeño detalle que se pasó por alto, y es que requiere que el paquete ya se encuentre en una estructura **sk_buff**. Es decir, podría surgir la siguiente cuestión: ¿qué implica que la función requiera de una estructura de datos **sk_buff**? Antes de continuar con este caso de uso, se recomienda regresar a la sección 2.5.1 dónde se explica la estructura **sk_buff**, para qué se utiliza y qué puede ofrecer.

Restricciones para hacer Broadcast con XDP

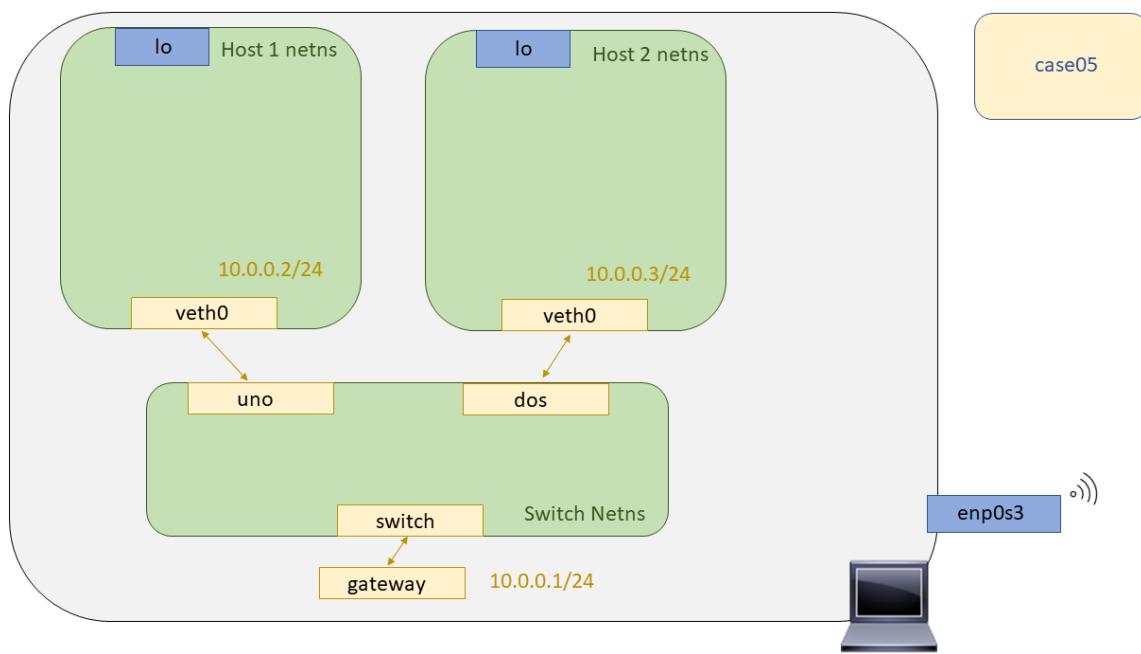


Figura 4.15: Escenario propuesto del Case05 - XDP

Atendiendo a las características de la estructura `sk_buff`, se pueden entender un poco mejor las restricciones que implica que el *helper BPF* haga uso de esta estructura. Cuando se trabaja con XDP, se maneja una estructura mucho más simple y menos pesada que el `sk_buff`, llamada `xdp_buff`, en la que se introduce la información exclusiva para operar con el paquete en la propia interfaz.

Por ello, no se puede hacer uso del *helper BPF* ya en caso de querer hacer uso de él se debería hacer una reserva para esta estructura y hacer una traducción a mano de `xdp_buff` a `sk_buff`. Haciendo esto se estaría operando de manera intrusiva con la lógica propia del *stack* de red del Kernel de Linux, y además se estaría perdiendo el rendimiento que reporta no trabajar con estas estructuras.

Por lo que investigando y aprendiendo un poco más sobre BPF para poder valorar las posibles opciones antes de desistir, se vio que el siguiente punto siguiendo el *datapath* de Linux donde se producen “hooks” (proceso de anclaje de un *bytecode* en el Kernel de Linux) es en el TC. Sabiendo cómo opera el TC (en caso no ser así se recomienda la lectura de la sección 2.11), se propone la siguiente solución para llevar a cabo el forwarding (Ver fig. 4.16).

De esta forma, en el TC ya se tendría el manejo de la estructura `sk_buff` y por ende, ya se podría hacer uso del *helper BPF* para clonar el paquete y hacer un reenvío a cada una de las interfaces que se quiera, para completar así el broadcast.

Compilación

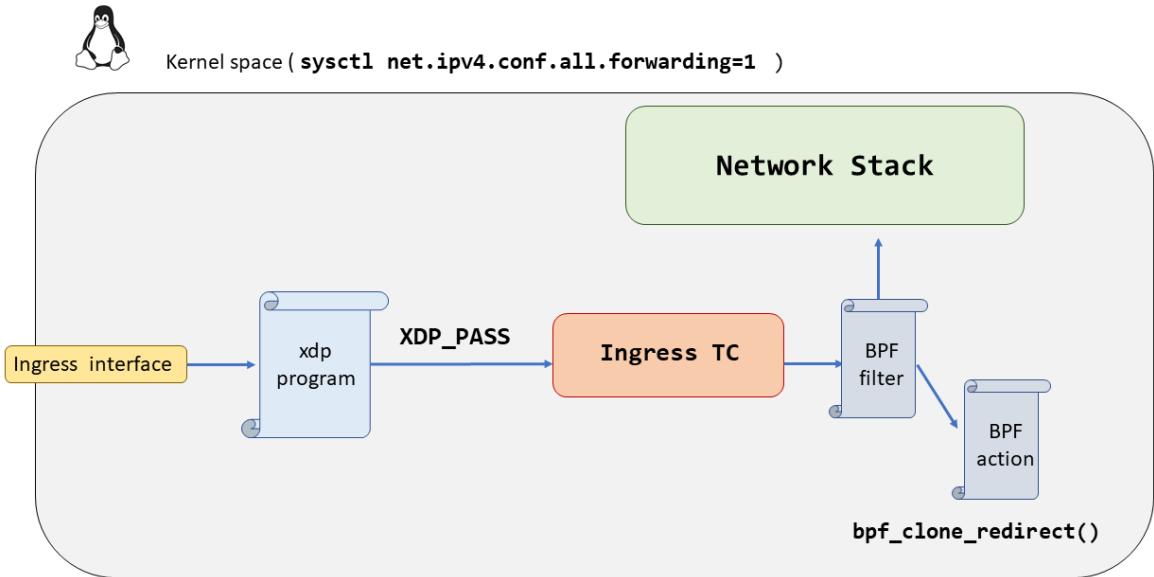


Figura 4.16: Solución propuesta para el Case05 - XDP

Para compilar el programa XDP se ha dejado un Makefile preparado en este directorio al igual que en el case04 (4.1.4), por lo que para compilarlo únicamente hay que seguir las indicaciones del bloque 4.13.

Código 4.22: Compilación programa XDP - Case05

```

1  # En caso de no haber entrado en el directorio asignado del caso de uso
2  cd TFG/src/use_cases/xdp/case05
3
4
5  # Hacemos uso del Makefile suministrado
6  sudo make

```

Si tiene dudas sobre el proceso de compilación del programa XDP le recomendamos que vuelva al case02 (4.1.2) donde se hace referencia al *flow* dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para comprobar el funcionamiento de los programas XDP se hará uso de las *Network Namespaces* (más información en la sección 2.5.3). Como ya se comentaba, para que no suponga una barrera de entrada el concepto de las *Network Namespaces*, se ha dejado escrito un script para levantar el escenario, y para su posterior limpieza. Es importante señalar que el script

debe ser lanzado con permisos de root. Para levantar el escenario debemos ejecutar dicho script como se indica en el bloque 4.10.

Para limpiar la máquina del escenario recreado anteriormente, se puede correr el mismo script indicándole ahora el parámetro `-c` (*Clean*). En el peor de los casos, y si se cree que la limpieza se ha realizado de manera satisfactoria, se puede llevar a cabo un reinicio de la máquina consiguiendo así que todos los entes no persistentes (Veth, netns..) desaparezcan del equipo.

Código 4.23: Puesta en marcha del escenario - Case05

```

1 # Para levantar el escenario (Importante hacerlo con permisos de super usuario)
2 sudo ./runenv.sh -i
3
4
5 # Una vez finalizado la comprobación del caso de uso, limpiaremos nuestra maquina:
6 sudo ./runenv.sh -c

```

Una vez levantado el escenario, tendríamos el escenario que se muestra en la figura 4.17 montado con tres *Network Namespaces* y pares de Veths para interconectarlas. Los programas irán anclados a las interfaces de la *Network Namespace switch*.

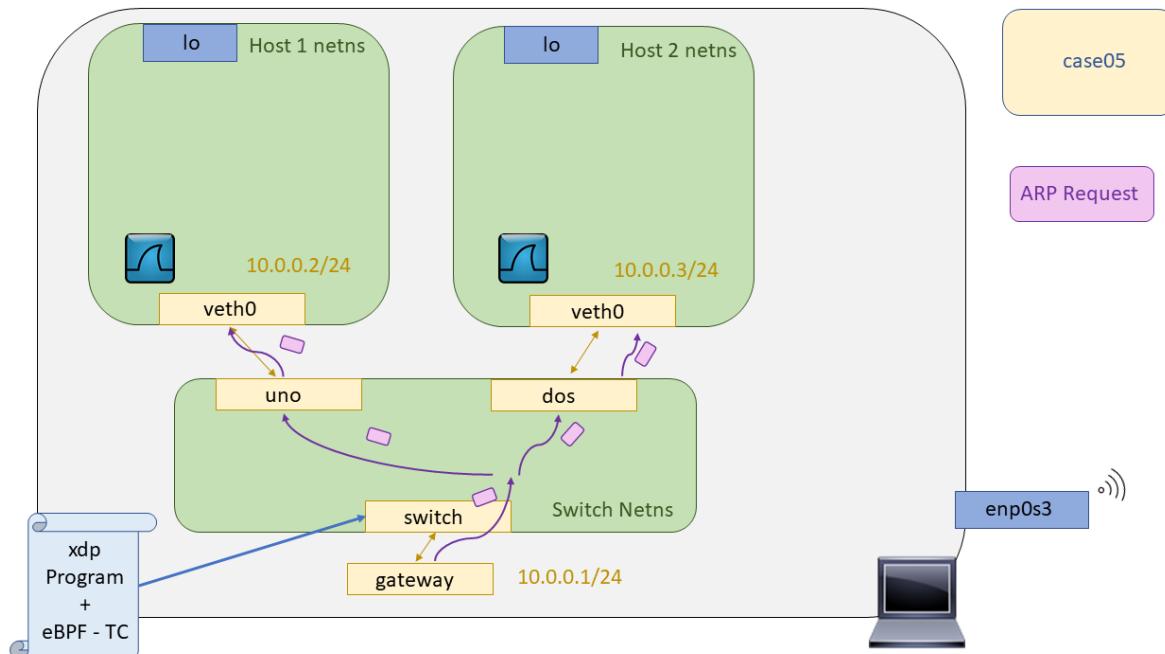


Figura 4.17: Escenario del Case05 - XDP

Carga del programa XDP

Una vez compilado tanto el programa XDP como los programas BPF que irán al TC, es hora de cargarlos. Por lo que, para hacerlo y por mayor comodidad se abrirá un proceso de bash en la *Network Namespace* switch. Acto seguido, se cargará el programa XDP, y se incluirán los programas BPF añadiendo una nueva qdisc con un filtro BPF asociado que derivará en una acción. Dicha acción, será otro programa BPF con la función de clonar los paquetes y mandarlos a otras interfaces.

Código 4.24: Carga del programa XDP - Case05

```

1 # Nos abrimos un proceso de bash sobre la Network Namespace "switch"
2 sudo ip netns exec switch bash
3
4 # Cargamos el programa XDP_PASS
5 sudo ./xdp_loader -d switch xdp_pass
6
7 # Creamos la nueva qdisc
8 sudo tc qdisc add dev switch ingress handle ffff:
9
10 # Aplicamos a la qdisc creada un filtro BPF que en caso de matchear aplicará una acción (Otro ↪
11   ↪ programa BPF que hará nuestro broadcast)
11 sudo tc filter add dev switch parent ffff: bpf obj bpf.o sec classifier flowid ffff:1 action ↪
    ↪ bpf obj bpf.o sec action

```

Comprobación del funcionamiento

Para comprobar el funcionamiento del sistema de broadcast se realizará la siguiente prueba, donde desde la *Network Namespace* por defecto se generarán ARP-Request hacia la IP de una de las Veths de las *Network Namespace* destino.

Si el sistema de broadcast funciona correctamente, escuchando en las *Network Namespace* destino uno y dos, se debería ver como los paquetes ARP-Request llegan sin problemas. Solo el “Host” al cual iban dirigidos los ARP-Request será el que intentará contestarlos sin éxito al haber implementado un sistema unidireccional. Para solucionar esta limitación se propone hacer uso de los programas XDP desarrollados en case04 (4.1.4) para conseguir una comunicación bidireccional.

Código 4.25: Comprobación del funcionamiento - Case05

```

1 # Generamos el ARP-REQUEST
2 arping 10.0.0.2/3
3
4 # Escuchamos en las Network Namespace destino a la espera de ver ARP-REQUEST.
5 sudo ip netns exec uno tcpdump -l
6 sudo ip netns exec dos tcpdump -l

```

En este caso no se podrá hacer uso del programa xdp_stats para ver si realmente los programas XDP están funcionando como se quiere que funcionen, ya que la lógica de broadcast se encuentra en el programa BPF anclado como acción en un filtro del TC.

Como se puede apreciar en la figura 4.18, los ARP-Request █ llegan a ambos Host, y solo aquel al cual iba dirigida la resolución ARP contesta. Pero como ya se comentaba antes, al no haber implementado un sistema de forwarding, la comunicación únicamente está planteada en un sentido.

Esta funcionalidad se podría aumentar haciendo uso de los programas desarrollados en el caso de uso anterior (4.1.4). Por lo que se concluye afirmando que se ha podido hacer broadcast, pero no de forma exclusiva con XDP ya que se ha tenido que añadir BPF nativo en el TC.

The figure consists of five screenshots labeled 2, 3, 5, and 8, each showing a terminal window with command-line output. Screenshot 2 shows the command 'arping 10.0.0.2 -I gateway' with a response from 'n0obie@n0obie-VirtualBox'. Screenshot 3 shows 'tcpdump -l' output for interface 'veth0' with many ARP requests and responses. Screenshot 5 shows the directory tree for 'use_cases/xdp/case05'. Screenshot 8 shows the same 'tcpdump -l' output as screenshot 3, but with fewer ARP responses, indicating partial broadcast functionality.

```

2. XDP
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ arping 10.0.0.2 -I gateway
ARPING 10.0.0.2 from 10.0.0.1 gateway
AC>Sent 11 probes (11 broadcast(s))
Received 0 responses(0)
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ ^C
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ ■

3. XDP
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ sudo ip netns exec uno tcptdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0, link-type EN10MB (Ethernet), capture size 26144 bytes
16:55:53.766887 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:54.768849 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:55.769788 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:56.770937 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:57.773697 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:58.775217 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:55:59.776516 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:55:59.776555 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:00.777573 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:01.778529 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:01.778570 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:02.779900 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:02.779922 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:03.780268 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:03.780298 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
■

5. XDP
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ tree
.
└── bpf
    ├── bpf.c
    ├── bpf_helpers.h
    ├── bpf.o
    ├── Makefile
    ├── prog_kern.c
    ├── prog_kern.ll
    ├── prog_kern.o
    ├── RNDPE.h
    └── runme.sh
.
└── xdp_loader
└── xdp_stats

0 directories, 11 files
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ ■

8. XDP
n0obie@n0obie-VirtualBox:~/TFG/src/use_cases/xdp/case05$ sudo ip netns exec dos tcptdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0, link-type EN10MB (Ethernet), capture size 26144 bytes
16:55:53.766887 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:54.768849 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:55.769788 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:56.770937 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:57.773697 ARP Request who-has 10.0.0.2 (Broadcast) tell 10.0.0.1, length 28
16:55:58.775217 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:55:59.776516 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:55:59.776555 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:00.777573 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:01.778529 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:01.778570 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:02.779900 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:02.779922 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:03.780268 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
16:56:03.780298 ARP Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.1, length 28
■

```

Figura 4.18: Comprobación de funcionamiento del Case05 - XDP

4.2. Casos de uso P4 en medios cableados

En esta sección se introducirán todos los casos de uso realizados con la tecnología P4 en entornos cableados. Todos los casos de uso se han nombrado siguiendo la misma sintaxis que en el repositorio del TFG, alojado en GitHub. Para la **instalación de las dependencias** de la tecnología P4, se ha generado el Anexo C.1 donde se detallan todos los pasos a seguir. Se advierte al lector que quiera replicar los casos de uso, que debe ser extremadamente cuidadoso con las dependencias del entorno P4 que instala, ya que, de querer hacer un *upgrade* alguna de ellas puede generar numerosas incompatibilidades.

Los casos de uso P4, al igual que con los casos de uso XDP, se han dividido en partes diferenciadas con la finalidad de que la lectura de estos sea más clara y ordenada.

- **Introducción:** En esta parte se abordarán las explicaciones teóricas complementarias, explicaciones propias sobre el caso de uso y comentarios sobre el código P4 desarrollado.
- **Compilación y puesta en marcha del escenario:** En esta parte se explicará al lector cómo proceder para compilar el programa P4 y cómo levantar el escenario. En este caso ambos procesos están integrados en un mismo Makefile, por lo que, la replicación los casos de uso será más amena.
- **Evaluación del funcionamiento:** Por último, se hará una evaluación sobre el funcionamiento del caso de uso empleando la CLI de Mininet.

Para que el lector pueda seguir el desarrollo de los casos de uso P4, a continuación se indica la tabla 4.4, la cual expone en qué ruta del repositorio del TFG se puede encontrar dicho caso de uso, y un vídeo demostración donde el autor va comentando paso a paso el caso de uso y su evaluación.

Caso de uso	Enlace al repositorio	Enlace al vídeo demostración
case01 - Drop	Enlace al código	Enlace al vídeo
case02 - Pass	Enlace al código	Enlace al vídeo
case03 - Echo server	Enlace al código	Enlace al vídeo
case04 - Layer 3 forwarding	Enlace al código	Enlace al vídeo
case05 - Broadcast	Enlace al código	Enlace al vídeo

Tabla 4.4: Resumen de la documentación sobre los casos de uso P4 en entornos cableados

En esta ocasión, como ya se comentaba en el capítulo de Análisis y Diseño (3), la plataforma que se utilizará para evaluar el funcionamiento de los programas P4 en entornos cableados será Mininet. Por lo que si usted no está familiarizado con la herramienta Mininet, se le recomienda que acuda a la sección 2.6 donde se hace una introducción a esta herramienta.

4.2.1. Case01 - Drop

En este caso de uso se probará que es posible descartar todos los paquetes recibidos con un programa P4. Como tal, el programa P4 no es suficiente para probar esta funcionalidad, ya que requiere de una plataforma que sea capaz de soportar el lenguaje P4. Según se comentó en el capítulo de Análisis y Diseño (Cap. 3), se hará uso de soft-switch llamado behavioral-model, BMv2 en adelante, para testear los programas P4 y de Mininet como escenario para recrear las topologías de Red.

Antes de continuar con el caso de uso, se quiere remarcar un detalle importante. Continuamente se estará refiriendo al BMv2 como un “switch”, pero se debe entender que con el lenguaje P4 estamos definiendo el *datapath* que tendrá la entidad que cargue con el programa P4, en este caso el BMv2. Por lo que, la denominación de switch puede no ser totalmente correcta, ya que depende del programa P4 que porte para comportarse como un switch. Nótese que el uso del término “switch” deriva de la denominación de dispositivos programables en entornos SDN, que tradicionalmente se nombraron como “switches”

Debido a esto, el BMv2 puede actuar como un hub, un switch, un router o un firewall, etc. Dependerá de la funcionalidad implementada en el programa P4. Se aprovechará la interfaz implementada del BMv2 con Mininet, desarrollada desde la equipo de p4Lang, para conseguir integrar estos nodos en el escenario de red pudiendo así comprobar el funcionamiento del programa P4 desarrollado.

Compilación y puesta en marcha del escenario

Para la compilación del programa P4 se hará uso del compilador p4c. Este es el compilador de referencia para el lenguaje P4, es modular y permite escoger distintos *targets* para llevar a cabo la compilación. El proceso de compilación de los programas P4 se lleva a cabo en dos etapas, una etapa de compilación de *frontend* donde se genera un archivo ***.p4info**, el cual recoge todos los atributos necesarios del programa P4 en tiempo de ejecución (identificadores de tablas, su estructura, *actions..*), y una etapa de *backend*, en la cual se hace uso del archivo generado ***.p4info** para generar los archivos necesarios para programar al *target* en cuestión.

Por ejemplo, el compilador de *backend* que ataca al BMV2 genera un fichero ***.json**. Este fichero será suficiente para establecer todo el *datapath* según lo programado en el programa P4. El target del compilador p4c que se utilizará es el **p4c-bm2-ss**, P4 simple_switch - bmv2, el cual soporta la arquitectura **v1model**.

Con la finalidad de poner en marcha del escenario, se ha dejado escrito un Makefile, el cual compilará el programa P4, generando los ficheros ***.p4info** y ***.json**. Acto seguido, se lanzará el script llamado **run_exercise.py**, el cual levantará toda la topología descrita en el fichero **scenario/topology.json** con Mininet. Cada “switch” de la topología tendrá implementada toda la lógica descrita en el programa P4 dentro de una instancia del BMv2. A continuación, en la figura 4.20 se puede ver una imagen resumen del levantamiento de un único “switch”.

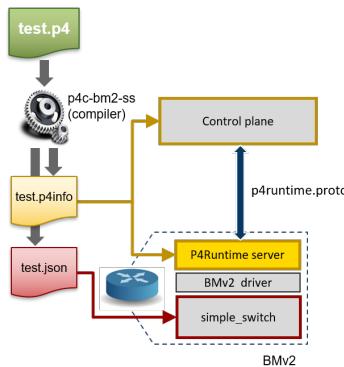


Figura 4.19: Proceso de compilación programa P4

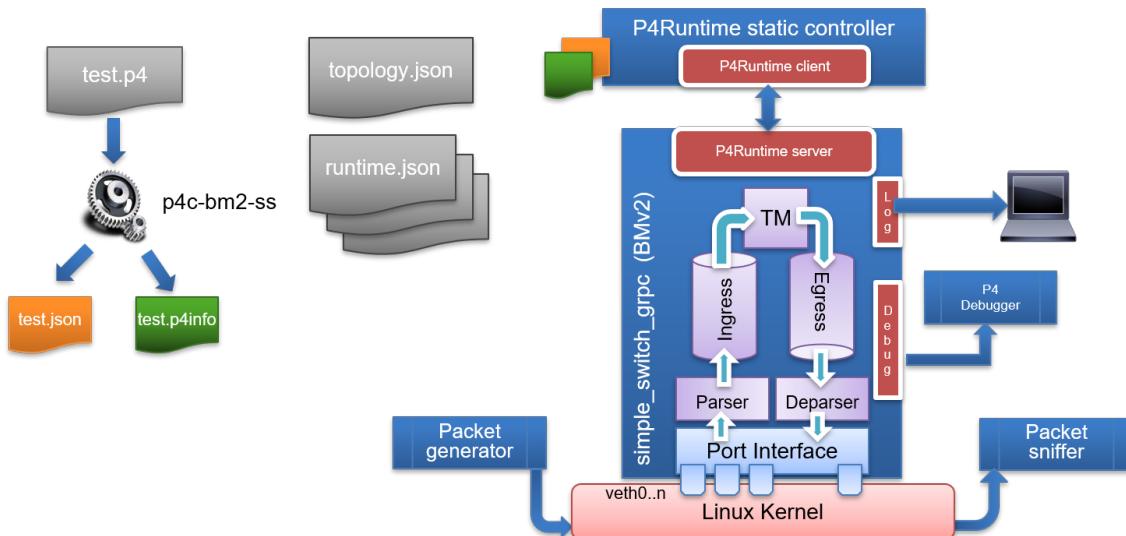


Figura 4.20: Proceso puesta en marcha de un switch BMv2 [?]

Debido a que las personas que quieran replicar los casos de uso puede que no estén muy familiarizadas con todo este proceso de compilación y carga en los procesos de BMv2, se ha dispuesto un de un Makefile para automatizar las tareas de compilación y carga de los programas, y también, para las tareas de limpieza del caso de uso una vez finalizada su demostración. Entonces, para proceder con la puesta en marcha del caso de uso, se deben seguir los pasos indicados en el bloque 4.26.

Código 4.26: Compilación programa P4 y puesta en marcha del escenario - Case01

```

1 # Entramos al directorio
2 cd TFG/src/use_cases/p4/case01
3
4 # Hacemos uso del Makefile
5 sudo make run

```

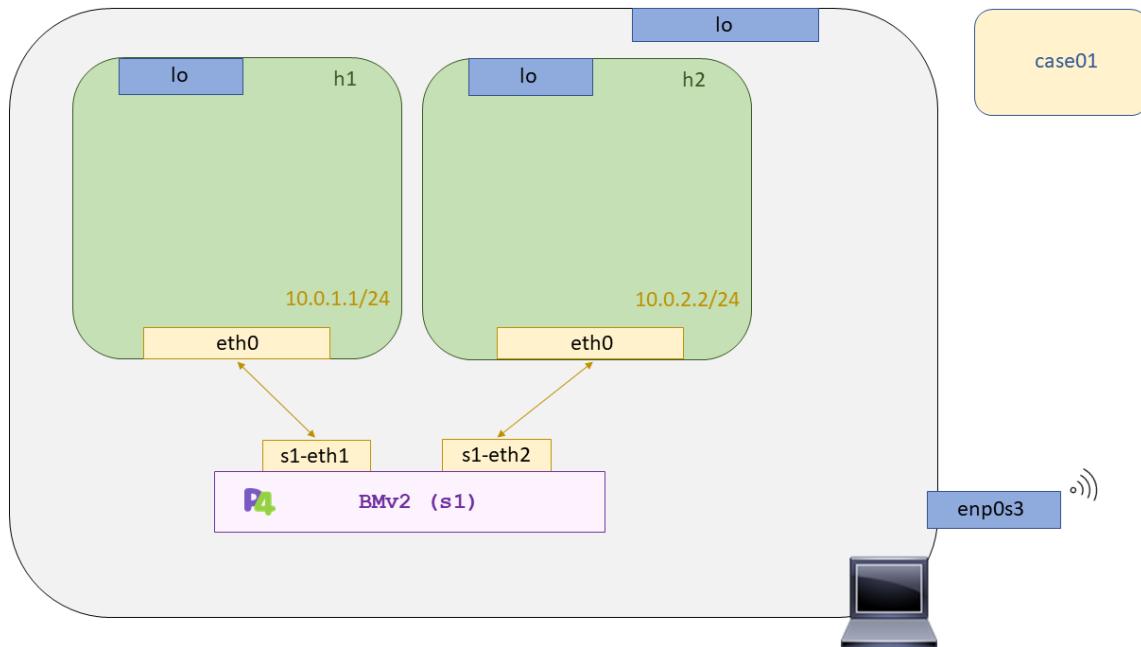


Figura 4.21: Escenario del Case01 - P4

Una vez se haya finalizado la comprobación del funcionamiento del caso de uso, se debe hacer uso de otro target (*clean*) del Makefile para limpieza total del directorio.

Código 4.27: Limpieza del escenario P4 - Case01

```

1 # Hacemos uso del Makefile
2 sudo make clean

```

Es importante señalar que este target limpiará tanto los ficheros auxiliares para la carga del programa P4 en el BMv2, como los directorios de **pcaps**, **log**, y **build** generados en la puesta en marcha del escenario. Por lo que, si se desea conservar las capturas de las distintas interfaces de los distintos BMv2, se deben copiar o limpiar del escenario a mano siguiendo las indicaciones del bloque 4.28.

Código 4.28: Limpieza segura del escenario P4 - Case01

```

1 # Limpiamos Mininet
2 sudo mn -c
3

```

```

4 # Limpiamos los directorios generados dinámicamente en la carga del escenario
5 sudo rm -rf build logs

```

Comprobación del funcionamiento

Una vez realizado el `make run` en este directorio, se tendrá levantada la topología descrita para este caso de uso, la cual se puede apreciar en la figura 4.21. La topología de este caso de uso se compondrá de una instancia del BMv2 (`s1`), y de dos host (`h1, h2`) que se conectarán al “switch”. Como ya se comentaba anteriormente, la topología puede encontrarse descrita bajo el directorio `scenario`, en un fichero JSON llamado `topology.json`. En este fichero también se describe la localización de los archivos que describen el plano de control de cada “switch” de la topología. En todos los casos de uso, se ha respetado los nombres tipo utilizados por la organización de p4Lang, `sX-runtime.json`, donde X es el número que ocupa dicho switch en la topología de Mininet. Volviendo de nuevo a la comprobación del funcionamiento del caso de uso, se tendrá la CLI de Mininet abierta, por lo que se abrirá una terminal para el `host1` y otra para el `host2`.

Código 4.29: Comprobación de funcionamiento - Case01

```
1 mininet> xterm h1 h2
```

Ahora con ambas terminales abiertas, desde el `h2` se pondrá a escuchar por su interfaz. Se puede utilizar wireshark, o el sniffer que el lector crea conveniente. En este caso por simplicidad y no disponer de una interfaz gráfica, se utilizará tcpdump (C.3).

```

X "Node:h1@n0obie-VirtualBox"
root@n0obie-VirtualBox:~/TFC/src/use_cases/p4/case01# ping 10.0.2.2
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Host Unreachable
From 10.0.1.1 icmp_seq=2 Destination Host Unreachable
From 10.0.1.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.2.2 ping statistics ---
6 packets transmitted, 0 received, +3 errors, 100% packet loss, time 5109ms
PIPE 4
root@n0obie-VirtualBox:~/TFC/src/use_cases/p4/case01#

```

Figura 4.22: Comprobación de funcionamiento (Ping) del Case01 - P4

Figura 4.23: Comprobación de funcionamiento (Sniffer) del Case01 - P4

Una vez que se esté escuchando en la interfaz del **host2**, se hará ping **host1** desde el **host1** al **host2** y no se debería tener conectividad. Como se puede ver en las figuras 4.22 y 4.23, el programa P4 desarrollado funciona correctamente. Para asegurarse de que realmente se está ejecutando el action para tirar paquetes se puede consultar los logs del BMv2, los cuales generan un fichero de log por cada instancia target del simple switch que se haya levantado.

Estos logs se encuentran en el directorio **logs**, y cada fichero de log pertenece a un único “switch”, llevando el mismo nombre que este en Mininet. En este caso, si se quisieran monitorizar los logs del **s1** bastaría con seguir las indicaciones del bloque 4.30.

Código 4.30: Comprobación de funcionamiento - Case01

```

1  # Entramos al directorio
2  cd TFG/src/use_cases/p4/case01
3
4  # Monitorizamos los logs del Switch (s1)
5  tail -f logs/s1.log

```

4.2.2. Case02 - Pass

Como se puede apreciar en esta sección, no se ha desarrollado ningún programa P4. Esto se debe a que no hay equivalente en P4 del código de retorno XDP_PASS, por ello, no se puede hacer nada en este caso de uso. A continuación, se indica el porqué no se encuentra un equivalente entre ambas tecnologías. El código de retorno en XDP es una forma para llevar a cabo una acción con el paquete que llega a la interfaz, en la cual hay anclado un programa XDP. Para más información sobre los códigos de retorno consultar la sección 2.4 donde se indica en detalle cuantos hay, para qué sirven y qué limitaciones tienen.

En este caso, el código de retorno `XDP_PASS` implica que el paquete se pasa al siguiente punto del procesamiento del *stack* de red en el Kernel de Linux. Es decir, si el programa está anclado a la NIC, se dejará pasar el paquete al TC, de ahí al propio *stack* de red para parsear sus cabeceras, y más adelante, pasárselo a la interfaz de sockets. En P4, el entorno donde se llevarán a cabo los casos de uso será Mininet con “switches” (BMv2) y host. Los “switches” (BMv2) es un soft-switch que permite inyectarle código P4, con el cual se puede definir el *datapath* del mismo.

Ahora bien, aquí viene la gran diferencia entre ambas tecnologías, con XDP siempre se puede pasar el paquete al Kernel para que se encargue el del procesamiento, pero en P4, se debe definir de forma exclusiva todo el *datapath*, por lo que no hay a quién delegar el paquete, dado que se tiene que encargar el propio programa P4. Entonces, como tal, no habría equivalente del `XDP_PASS` en P4.

4.2.3. Case03 - Echo server

En este caso de uso desarrollaremos un servidor de Echo que responda todos los pings que le lleguen. Como tal el programa P4 no es suficiente para probar esta funcionalidad, como ya se mencionó, se requiere de una plataforma que sea capaz de soportar el lenguaje P4, el BMv2.

El programa P4 deberá ser capaz de analizar los paquetes que le lleguen, analizarlos y filtrarlos. Solo aquellos paquetes filtrados serán los que se deberán responder. ¿Cómo se filtrarán? Se añadirán nuevos estados en el parser que comprueben si las cabeceras ICMP están presentes. Por ello, antes de nada se debe declarar las cabeceras ICMP necesarias para poder así analizar las cabeceras de los paquetes que lleguen.

Código 4.31: Estructura cabecera ICMP - Case03

```

1  header icmp_t {
2      bit<8> type;
3      bit<8> code;
4      bit<16> checksum;
5 }
```

Como se quería que este programa P4 fuera compatible con direccionamiento IPv6 se han añadido también los equivalentes en ICMPv6 y la cabecera de red de IPv6, ver bloque 4.32.

Código 4.32: Estructuras cabeceras IPv6 e ICMPv6 - Case03

```

1  header ipv6_t {
2      bit<4> version;
3      bit<8> trafficClass;
4      bit<20> flowLabel;
5      bit<16> payloadLen;
6      bit<8> nextHdr;
7      bit<8> hopLimit;
8      ip6Addr_t srcAddr;
9      ip6Addr_t dstAddr;
10 }
11
12
13  header icmp6_t {
14      bit<8> type;
```

```

15     bit<8> code;
16     bit<16> checksum;
17 }
```

Ahora que ya se tiene las cabeceras definidas, se deberá preocuparse en definir las macros asociadas a los posibles *ethertypes* que se quieran manejar, IPv4 y IPv6. De forma adicional, se deberán definir los códigos de protocolo de las cabeceras de red, para asegurare que sobre la cabecera de red únicamente se procesarán aquellos paquetes que porten información ICMP. Se tuvo que consultar la RFC asociada a IPv6 para saber que codificación hacían con el campo de *nextHeader* y por lo visto utilizan los mismo valores que en IPv4. A continuación, en la figura 4.33 se puede ver la definición de dichas macros y el extracto de la RFC asociada a la codificación de las cabeceras IPv6 .

Código 4.33: Macros para parsear L2 y L3 - Case03

```

1  /* --- Layer 2 MACROS --- */
2  const bit<16> ETHERTYPE_IPV4 = 0x0800;
3  const bit<16> ETHERTYPE_IPV6 = 0x86dd;
4
5  /* --- Layer 3 MACROS --- */
6  const bit<8> IP_PROTOCOL_ICMP = 0x01;
7  const bit<8> IP_PROTOCOL_ICMPv6 = 0x3a;
8  const bit<8> ICMP_ECHO_REQUEST_TYPE = 0x08;
9  const bit<8> ICMP_ECHO_REQUEST_CODE = 0x00;
10 const bit<8> ICMP_ECHO_REPLY_TYPE = 0x00;
11 const bit<8> ICMP_ECHO_REPLY_CODE = 0x00;
12
13 /*
14 * According to RFC 2460, the codes of the protocol immediately above,
15 * layer 4, are the same as those used in IPv4. And I quote:
16 *
17 * Next Header 8-bit selector. Identifies the type of header
18 * immediately following the IPv6 header. Uses the
19 * same values as the IPv4 Protocol field [RFC-1700
20 * et seq.]
21 *
22 */
```

Como se ha indicado en el bloque 4.33, de forma adicional se han definido macros sobre los valores con los que se va estar trabajando (ECHO-Request y ECHO-Reply). De esta forma el código resultante de las *actions* será más interpretable y sostenible. Teniendo ya todas las herramientas necesarias para declarar los nuevos estados del parser, se puede proceder con el siguiente bloque del datapath.

Una vez que se es capaz de filtrar los paquetes que interesan, se va a ver cómo se ha implementado la lógica de procesamiento de aquellos paquetes que ya han sido filtrados. Es de interés interceptar todos los paquetes ICMP que lleguen a nuestro “switch” para así contestarlos desde el mismo ”switch”. Por ello, se hará uso del bit de validez de las cabeceras que han sido analizadas correctamente, es decir, si el paquete contiene dichas cabeceras. De no contenerlas, su bit de validez estará a *false*. A continuación en la figura 4.34, mostramos la lógica de control del bloque *Ingress*.

Código 4.34: Lógica para filtrar paquetes ICMP e ICMPv6 - Case03

```

1  /*
2   * De esta forma nos aseguramos que únicamente procesamos paquetes ICMP,
3   * o en su defecto ICMPv6.
4   *
5   */
6
7
8  apply {
9    if(hdr.ipv4.isValid() && hdr.icmp.isValid()){
10      echo();
11    }else if (hdr.ipv6.isValid() && hdr.icmp6.isValid()){
12      echo6();
13    }
14 }
```

Ya se tienen todos los paquetes ICMP filtrados, solo quedaría programar la lógica para modificar las cabeceras de dicho paquete para conseguir contestar satisfactoriamente al emisor del ping. Para ello, se hará uso de un *action*, una función. El cual intercambiará tanto MAC destino-origen, como IP destino-origen, modificará la cabecera ICMP para indicarle que se trata de un *reply*, y por último mandará el paquete por el puerto por el cual llegó al BMv2, de esta forma se asegura que dicho paquete llegará al emisor del mismo.

Código 4.35: Lógica para procesar paquetes ICMP - Case03

```

1  action echo (){
2    /* --- Auxiliary variables --- */
3    macAddr_t temp_mac = hdr.ethernet.srcAddr;
4    ip4Addr_t temp_ip = hdr.ipv4.srcAddr;
5
6    /* --- Swap MACs --- */
7    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
8    hdr.ethernet.dstAddr = temp_mac;
9
10   /* --- Swap Ips --- */
11   hdr.ipv4.srcAddr = hdr.ipv4.dstAddr;
12   hdr.ipv4.dstAddr = temp_ip;
13
14   /* --- Re-Write ICMP's type and code --- */
15   hdr.icmp.type = ICMP_ECHO_REPLY_TYPE;
16   hdr.icmp.code = ICMP_ECHO_REPLY_CODE;
17
18   /* --- Forward the packet to the ingress intf --- */
19   standard_metadata.egress_spec = standard_metadata.ingress_port;
20
21 }
```

Préstese atención a la última sentencia de la función donde le indicamos que saque el paquete por la misma interfaz por la cual ha entrado. Este es el equivalente directo al código de retorno en XDP, `XDP_TX` con el cual re-circulábamos el paquete de la misma forma a la interfaz de entrada.

Como curiosidad, es interesante comentar que el campo `checksum` de la cabecera ICMP debe ser re-calculado de nuevo por lo que se deberá hacer antes del `Egress`. Esto supuso un problema, ya que continuamente estaban saliendo paquetes con un `checksum` incorrecto, lo cual se comprobó con el disector de Wireshark. Finalmente se vio que no había que incluir el campo `checksum` en la nueva suma del nuevo `checksum`⁶, de esta forma los paquetes ECHO-Reply que salían ya llevaban el `checksum` calculado correctamente.

Compilación y puesta en marcha del escenario

Para la compilación del programa P4 se hará uso de nuevo del compilador P4c (más información sobre el proceso de compilación en la subsección 4.2.1).

Dado que las personas que quieran replicar los casos de uso puede que no estén muy familiarizadas con todo este proceso de compilación y carga en los procesos de BMv2, se ha dispuesto un de un Makefile para automatizar las tareas de compilación y carga, y las tareas de limpieza del caso de uso. Para la puesta en marcha del caso de uso se deben seguir los pasos indicados en el bloque 4.36.

Código 4.36: Compilación programa P4 y puesta en marcha del escenario - Case03

```

1 # Entramos al directorio
2 cd TFG/src/use_cases/p4/case03
3
4 # Hacemos uso del Makefile
5 sudo make run

```

Una vez se haya finalizado la comprobación del funcionamiento del caso de uso, se debe hacer uso de otro target (`clean`) del Makefile para limpieza total del directorio según se indica en el bloque 4.37.

Código 4.37: Limpieza del escenario P4 - Case03

```

1 # Hacemos uso del Makefile
2 sudo make clean

```

Es importante señalar que este target limpiará tanto los ficheros auxiliares para la carga del programa P4 en el BMv2, como los directorios de `pcaps`, `log`, y `build` generados en la puesta en marcha del escenario. Por lo que si se desea conservar las capturas de las distintas interfaces de los distintos BMv2, se deben copiar o limpiar del escenario a mano siguiendo

⁶Habría sido genial que la RFC asociada hubiera estado más clara al respecto.

las indicaciones del bloque 4.38.

Código 4.38: Limpieza segura del escenario P4 - Case03

```

1  # Limpiamos Mininet
2  sudo mn -c
3
4  # Limpiamos los directorios generados dinámicamente en la carga del escenario
5  sudo rm -rf build logs

```

Comprobación del funcionamiento

Una vez realizado el make run en este directorio, se tendrá levantada la topología descrita para este caso de uso, la cual se puede apreciar en la figura 4.24. Como en el *datapath* no se contempla el manejo del protocolo ARP, se ha añadido el ARP entry a directamente en los hosts, desde el fichero `topology.json` consiguiendo así que no se genere la resolución ARP en el envío de los ECHO-Request. En este caso, la demostración se va hacer uso de IPv4, en el caso de querer hacerla con IPv6, se deberá añadir la entradaanalogapara el protocolo ND (*Neighbor Discovery*).

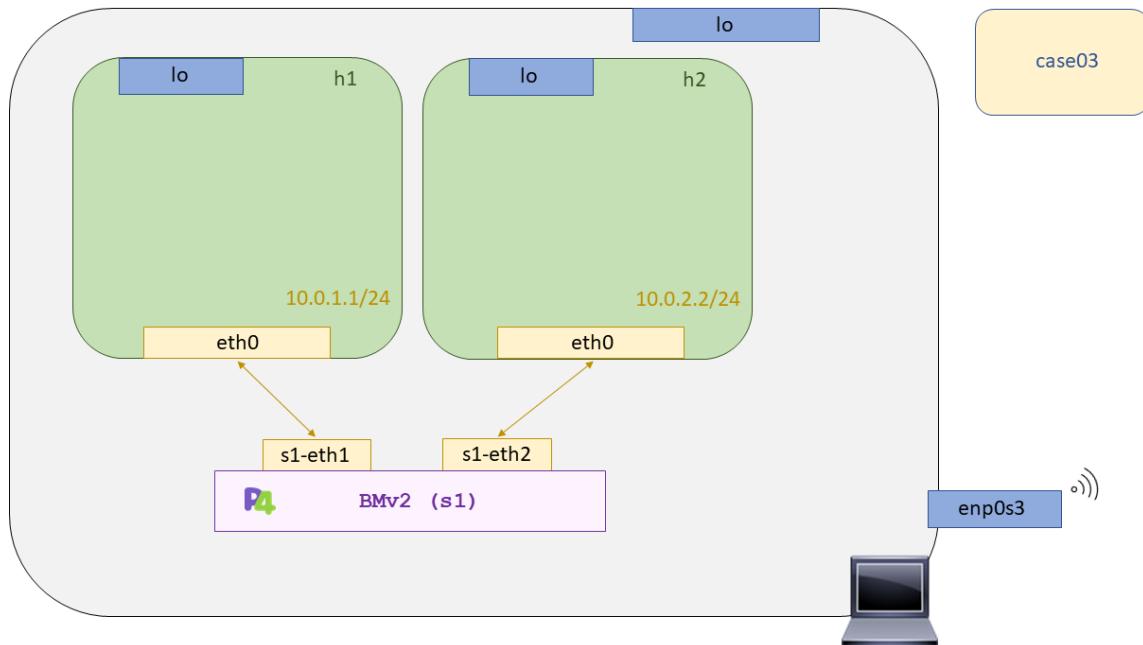


Figura 4.24: Escenario del Case03 - P4

Volviendo de nuevo a la comprobación del funcionamiento del caso de uso, se tendrá la CLI de Mininet abierta, por lo que se procederá a abrir una terminal para el `host1`, desde el cual se llevará a cabo los pings.

Código 4.39: Comprobación de funcionamiento - Case03

```
1 mininet> xterm h1
```

Una vez que se tenga la terminal abierta, se procederá a abrir Wireshark. En este caso se recomienda Wireshark ya que se podrá filtrar y comprobar de una forma más sencilla la validez de los *checksums*.

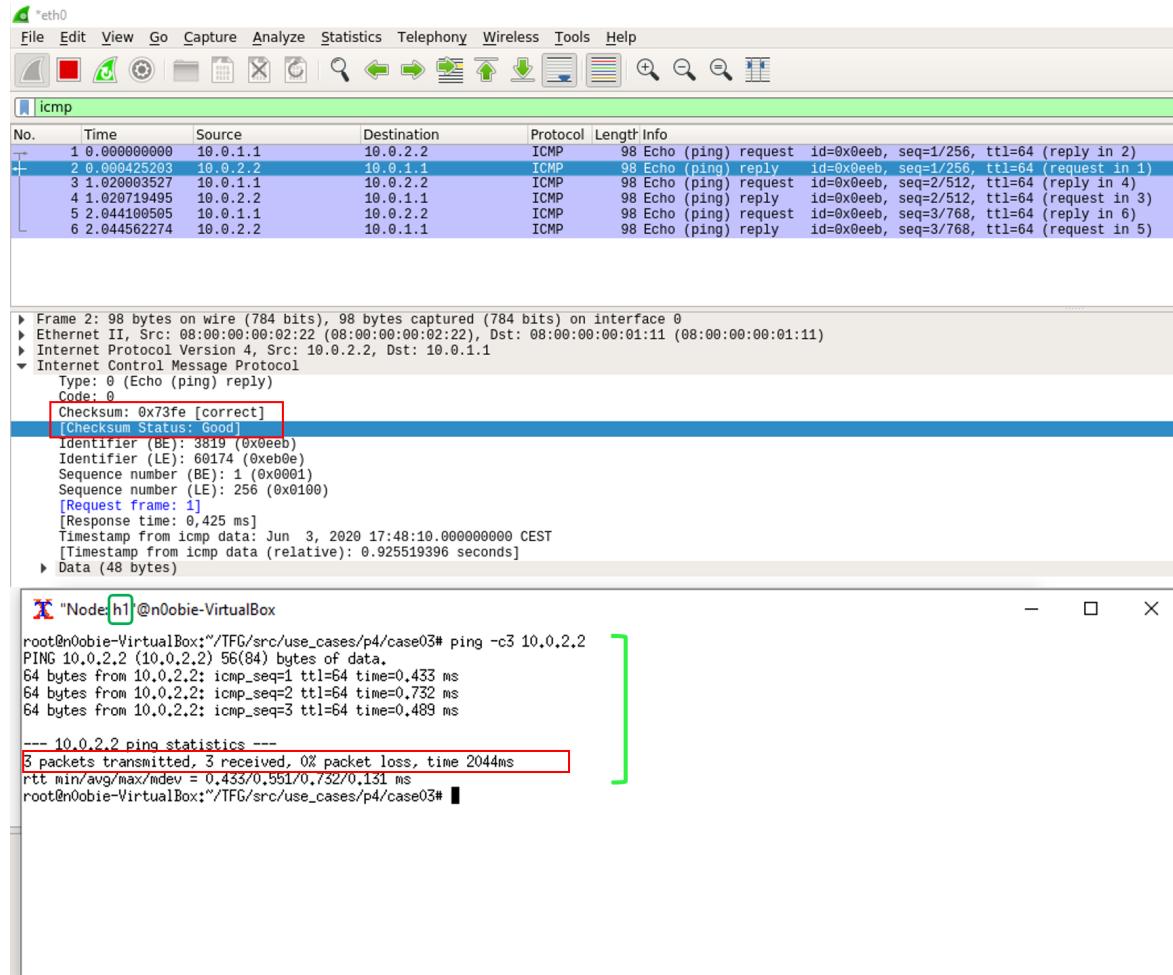


Figura 4.25: Comprobación de funcionamiento del Case03 - P4

Como se puede ver en la figura 4.25, todo funciona correctamente ya que se están recibiendo respuesta a los pings 3. De forma adicional, se puede comprobar con Wireshark como el *checksum* de la cabecera ICMP viene correctamente calculado.

4.2.4. Case04 - Layer 3 forwarding

En este caso de uso, se abordará la implementación un forwarding a nivel de red, capa 3, por lo que el hipotético “switch” será ahora un router muy básico y con muy pocas

funcionalidades. Por ello conviene recordar las anotaciones que se hicieron sobre el BMv2 y por qué no debemos definirlo únicamente como un soft-switch, ya que depende del programa P4 que perte para definir su *datapath* y su interfaz con el plano de control.

La motivación de este caso de uso, es ver el equivalente en P4 que tiene el código de retorno XDP llamado **XDP_REDIRECT**. En el anterior caso de uso ya se hacía uso de la información de metadatos para elegir el puerto de salida del paquete. Por lo que podríamos afirmar que este es el equivalente directo al forwarding en XDP, solo que aquí en P4 resulta más sencillo ya que únicamente indicamos por número de puerto mientras que XDP se debía obtener el *ifindex* de la interfaz a la cual íbamos a hacer el reenvío. Como se puede apreciar en la figura 4.40, únicamente le indicamos el puerto por el cual debe salir el paquete.

Código 4.40: Redirección del paquete - Case04

```
1 standard_metadata.egress_spec = port
```

Pero, ¿Qué son los números de puerto? ¿Cuando se han establecido estos identificadores? Estos números de puerto se establecen cuando se levanta la instancia del BMv2. Se asocia interfaz real con un número identificativo. Serán estos números identificativos los números de puerto de cada “switch”. De esta manera el “switch” tiene a su alcance todos sus posibles puertos. El levantamiento de las instancias de BMv2 se llevan a cabo con la carga del script **run_exercise.py**. Este a su vez hace uso de la clase **P4RuntimeSwitch** definida en la interfaz de Python hacia Mininet, para levantar dicho “switch” pasándole los parámetros adecuados. Se puede levantar una instancia del BMv2 siguiendo las indicaciones del bloque 4.41.

Código 4.41: Ejecución de instancia del BMv2 - Case04

```
1 sudo simple_switch_grpc --log-console --dump-packet-data 64 \
2   i 0@veth0 -i 1@veth2 ...[--pcap] --no-p4 \
3   --grpc-server-addr 0.0.0.0:50051 --cpu-port 255 \
4   test.json
```

Una vez entendido de dónde salen los números de puerto, se debe plantear la siguiente cuestión, ¿Cómo un programa P4 es capaz de conocer los posibles puertos de los que puede disponer para reenviar un paquete?

No puede, y tampoco es lógico que en un programa P4, donde se define el *datapath*, ya estén preestablecidos los números de puerto posibles a manejar. Ya que si eso fuera así, se tendría que tener distintos programas P4 por cada entidad con un número de puertos distintos (únicamente para implementar un único *datapath*). No es viable. Por ello, toda la información relativa a los puertos, asociada al forwarding, generalmente suele venir desde el plano de control quien tiene constancia de los puertos de dicho “switch”.

Por ejemplo, si se quiere hacer forwarding de un paquete a un puerto en específico dado su IP, necesitaremos que el plano de control indique el criterio a seguir, qué IPs están asociadas a según qué puertos. Esta interfaz entre el plano de datos y el plano de control está definidas

por tablas.

Desde el programa P4 se tiene que definir el esqueleto de la tabla, indicando qué *actions* están disponibles, qué parámetros reciben esas *actions*, qué criterio de match tendrá la tabla, sobre qué *keys* se realizará el *lookup* y cuál es el número máximo de entradas en dicha tabla. A continuación, se muestra la figura 4.26 que resume bastante bien la funcionalidad de las tablas.

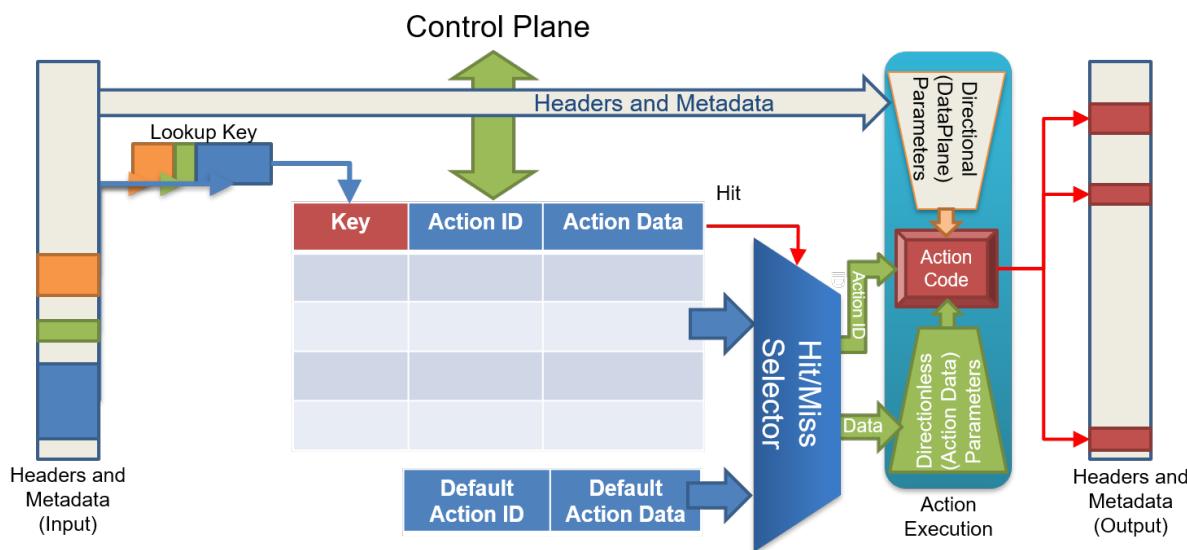


Figura 4.26: Funcionamiento de las tablas en P4 [?]

Por tanto, desde el plano de control, bien vía P4Runtime ó vía json con los ficheros `sX-runtime.json` (que se cargarán a través de la CLI-BMv2), se llenarán las entradas de dicha tabla y los parámetros de las acciones a llevar a cabo cuando haya un *hit* con dicha entrada. En este caso se asociarán IPs a una acción de forwarding donde se le suministrará por qué puerto tiene que salir el paquete y que MAC destino debe llevar.

Una vez entendidos los números de puerto y el concepto de las tablas, se debe abordar el cómo hacer una acción de forwarding para reenviar los paquetes que lleguen al BMv2. Esta acción de forwarding deberá ser capaz de actualizar el puerto de salida, actualizar la MAC destino y decrementar en uno el campo `ttl` de la cabecera IP. A continuación, se indica la acción propuesta para llevar a cabo dicho cometido (Ver bloque 4.40).

Código 4.42: Acción propuesta para llevar a cabo el forwarding - Case04

```

1  /*
2   * Se quiso hacer uso del operador '-=' pero la sintaxis de P4 no lo permite :(
3   */
4
5  action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
6      standard_metadata.egress_spec = port;
7      hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
8      hdr.ethernet.dstAddr = dstAddr;

```

```

9     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
10    }

```

Teniendo la acción ya disponible, solo quedaría aplicar la tabla en nuestra pipeline del programa P4 y ya se estaría haciendo un forwarding en capa 3.

Compilación y puesta en marcha del escenario

Para la compilación del programa P4 se hará uso del compilador P4c (más información sobre el proceso de compilación en la subsección 4.2.1).

Dado que las personas que quieran replicar los casos de uso puede que no estén muy familiarizadas con todo este proceso de compilación y carga en los procesos de BMv2, se ha dispuesto un de un Makefile para automatizar las tareas de compilación y carga, y las tareas de limpieza del caso de uso. Entonces para la puesta en marcha del caso de uso se deben seguir los pasos indicados en el bloque 4.43.

Código 4.43: Compilación programa P4 y puesta en marcha del escenario - Case04

```

1 # Entramos al directorio
2 cd TFG/src/use_cases/p4/case04
3
4 # Hacemos uso del Makefile
5 sudo make run

```

Una vez se haya finalizado la comprobación del funcionamiento del caso de uso, se debe hacer uso de otro target (*clean*) del Makefile para limpieza total del directorio según se indica en el bloque 4.44.

Código 4.44: Limpieza del escenario P4 - Case04

```

1 # Hacemos uso del Makefile
2 sudo make clean

```

Es importante señalar que este target limpiará tanto los ficheros auxiliares para la carga del programa P4 en el BMv2, como los directorios de **pcaps**, **log**, y **build** generados en la puesta en marcha del escenario. Por lo que si se desea conservar las capturas de las distintas interfaces de los distintos BMv2, cópielas o haga la limpieza del escenario a mano siguiendo las indicaciones del bloque 4.45.

Código 4.45: Limpieza segura del escenario P4 - Case04

```

1 # Limpiamos Mininet
2 sudo mn -c
3
4 # Limpiamos los directorios generados dinámicamente en la carga del escenario
5 sudo rm -rf build logs

```

Comprobación del funcionamiento

Así pues, después de realizar el `make run` en este directorio, se tendrá levantada la topología descrita para este caso de uso, la cual se puede apreciar en la figura 4.27. Como en el *datapath* no se contempla el manejo de ARP, se ha añadido el ARP entry a directamente desde el fichero `topology.json` consiguiendo así que no se genere la resolución ARP en el envío de los ECHO-Request. Este arreglo es poco elegante ya que se le está indicando un hipotético *gateway* que no existe, y se está añadiendo un ARP entry con la MAC de dicho *gateway*. De esta forma, todos los paquetes saldrán con la MAC destino indicada en la entry y no se producirá la resolución ARP. Al llegar al “switch” el paquete verá modificada su MAC destino en función de su IP destino, por lo que la “chapuza” no irá a más.

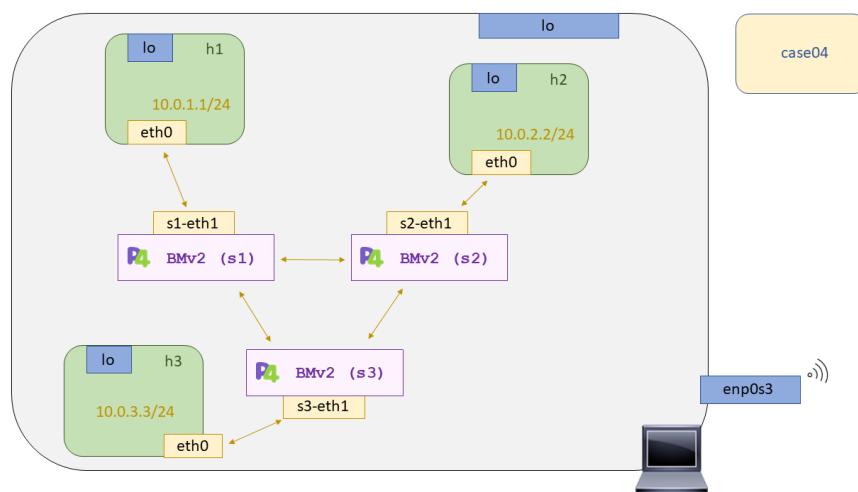


Figura 4.27: Escenario del Case04 - P4

Volviendo de nuevo a la comprobación del funcionamiento del caso de uso, se tendrá la CLI de Mininet abierta, por lo que simplemente se probará la conectividad entre todos los hosts. Esto se puede realizar ejecutando la orden `pingall`, la cual ejecutará pings entre todos los hosts, comprobando así la conectividad bidireccional de todos los *paths* preestablecidos en el escenario. De forma adicional, se podría hacer uso de un *sniffer* para comprobar que los paquetes llegan con el campo `ttl` modificado, en función de los saltos que ha dado el paquete. Según se puede apreciar en la figura 4.28, hay perfecta conectividad entre todos los hosts de la topología.

```

mininet> dump
<P4Host h1: eth0:10.0.1.1 pid=27369>
<P4Host h2: eth0:10.0.2.2 pid=27371>
<P4Host h3: eth0:10.0.3.3 pid=27373>
<ConfiguredP4RuntimeSwitch s1: s1-eth1:None,s1-eth2:None,s1-eth3:None pid=27375>
<ConfiguredP4RuntimeSwitch s2: s2-eth1:None,s2-eth2:None,s2-eth3:None pid=27384>
<ConfiguredP4RuntimeSwitch s3: s3-eth1:None,s3-eth2:None,s3-eth3:None pid=27387>
mininet> pingall
*** Ping: testing ping reachability
h1 -> *** h1 : (u'ping -c1 10.0.2.2',)
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=62 time=3.00 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.005/3.005/3.005/0.000 ms
h2 *** h1 : (u'ping -c1 10.0.3.3',)
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=2.71 ms

--- 10.0.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.710/2.710/2.710/0.000 ms
h3
h2 -> *** h2 : (u'ping -c1 10.0.1.1',)
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=62 time=13.2 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 13.265/13.265/13.265/0.000 ms
h1 *** h2 : (u'ping -c1 10.0.3.3',)
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=3.23 ms

--- 10.0.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.239/3.239/3.239/0.000 ms
h3
h3 -> *** h3 : (u'ping -c1 10.0.1.1',)
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=62 time=2.24 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.249/2.249/2.249/0.000 ms
h1 *** h3 : (u'ping -c1 10.0.2.2',)
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=62 time=2.99 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.991/2.991/2.991/0.000 ms
h2
*** Results: 0% dropped (6/6 received)
mininet> █

```

Figura 4.28: Comprobación de funcionamiento del Case04 - P4

4.2.5. Case05 - Broadcast

En este caso de uso, se desarrollará un programa p4 que haga Broadcast. En este caso únicamente se hará Broadcast a nivel de enlace, capa 2. La motivación de este caso de uso es ver la diferencia de dificultad respecto del entorno XDP donde se tuvo que anclar de manera adicional un *bytecode* eBPF en el TC, además del propio programa XDP anclado en la interfaz para lograr realizar un broadcast.

Para conseguir hacer una difusión de los paquetes en P4, se hará uso de los llamados grupos multicast. Los grupos multicast se utilizan para difundir por una serie de números de puertos del “switch”. Para más información sobre de dónde salen estos números de puerto vuelva al caso04 (4.2.4) donde se explica que representan estos identificadores y cuándo se asignan. Cada grupo multicast tiene un identificador único, y en él se definen una serie de réplicas, es decir, cuántas copias del paquete se llevarán a cabo y qué números de puerto se verán afectados.

Los grupos multicast se definen en la información del plano de control del “switch”, bien vía P4Runtime ó vía json con los ficheros **sX-runtime.json**. A continuación, en el bloque 4.46 se deja la definición del grupo multicast utilizado para este caso de uso.

Código 4.46: Ejemplo Json Grupo Multicast - Case05

```

1   "multicast_group_entries" : [
2     {
3       "multicast_group_id" : 1,
4       "replicas" : [
5         {
6           "egress_port" : 1,
7           "instance" : 1
8         },
9         {
10          "egress_port" : 2,
11          "instance" : 1
12        },
13        {
14          "egress_port" : 3,
15          "instance" : 1
16        }
17      ]
18    }
19  ]

```

Con esta definición se está indicando que todos los paquetes que pertenezcan al grupo multicast cuyo identificador sea 1, generarán una copia del paquete por los puertos del “switch” 1, 2 y 3. Es decir por todos los puertos de nuestro “switch” para este caso de uso. Por tanto la acción de broadcast/multicast será la siguiente:

Código 4.47: Acción propuesta para llevar a cabo el Broadcast - Case05

```

1   action multicast() {
2     standard_metadata.mcast_grp = 1;
3   }

```

Únicamente se asigna el paquete a dicho grupo multicast, y el BMv2 ya se encarga de clonar

el paquete y reenviarlo por los puertos indicados. Dado que no se quiere generar paquetes por el puerto por el cual se recibió el paquete de broadcast, podría plantearse tener tantos grupos multicast como puertos tuviera el BMv2, generando copias por puertos específicos y asociando cada grupo multicast a los paquetes que llegaran por cada puerto. Pero esta solución no sería del todo escalable. Por ello, como se puede ver en el grupo multicast (bloque de código 4.46), se generan copias en todos los puertos asociados a esta instancia del BMv2. Será por tanto en la fase de egress cuando se comprueben los metadatos del paquete, y si el puerto de entrada es igual al puerto de salida, se descartará el paquete (Ver bloque de código 4.48).

Código 4.48: Acción propuesta para descartar paquetes sobrantes de Broadcast - Case05

```

1  control MyEgress(inout headers hdr,
2                     inout metadata meta,
3                     inout standard_metadata_t standard_metadata) {
4
5      action drop() {
6          mark_to_drop(standard_metadata);
7      }
8
9      apply {
10         // Prune multicast packet to ingress port to preventing loop
11         if (standard_metadata.egress_port == standard_metadata.ingress_port)
12             drop();
13     }
14 }
```

En este caso, se ha encontrado que implementar esta funcionalidad en P4 es mucho más sencillo que con XDP. Por lo que, en caso de necesitar probar protocolos que implementen la difusión como parte de su lógica, es mucho más recomendable y viable hacer uso de P4.

Compilación y puesta en marcha del escenario

Para la compilación del programa P4 se hará uso del compilador P4c (más información sobre el proceso de compilación en la subsección 4.2.1).

Dado que las personas que quieran replicar los casos de uso puede que no estén muy familiarizadas con todo este proceso de compilación y carga en los procesos de BMv2, se ha dispuesto un de un Makefile para automatizar las tareas de compilación y carga, y las tareas de limpieza del caso de uso. Entonces para la puesta en marcha del caso de uso se deben seguir los pasos indicados en el bloque 4.49.

Código 4.49: Compilación programa P4 y puesta en marcha del escenario - Case05

```

1  # Entramos al directorio
2  cd TFG/src/use_cases/p4/case05
3
4  # Hacemos uso del Makefile
5  sudo make run
```

Una vez se haya finalizado la comprobación del funcionamiento del caso de uso, se debe hacer uso de otro target (*clean*) del Makefile para limpieza total del directorio según se indica

en el bloque 4.50.

Código 4.50: Limpieza del escenario P4 - Case05

```
1 # Hacemos uso del Makefile
2 sudo make clean
```

Es importante señalar que este target limpiará tanto los ficheros auxiliares para la carga del programa P4 en el BMv2, como los directorios de `pcaps`, `log`, y `build` generados en la puesta en marcha del escenario. Por lo que si se desea conservar las capturas de las distintas interfaces de los distintos BMv2, cópielas o haga la limpieza del escenario a mano siguiendo las indicaciones del bloque 4.51.

Código 4.51: Limpieza segura del escenario P4 - Case04

```
1 # Limpiamos Mininet
2 sudo mn -c
3
4 # Limpiamos los directorios generados dinámicamente en la carga del escenario
5 sudo rm -rf build logs
```

Comprobación del funcionamiento

Una vez realizado el make run en el directorio, se tendrá levantada la topología descrita para este caso de uso, la cual se puede apreciar en la figura 4.29.

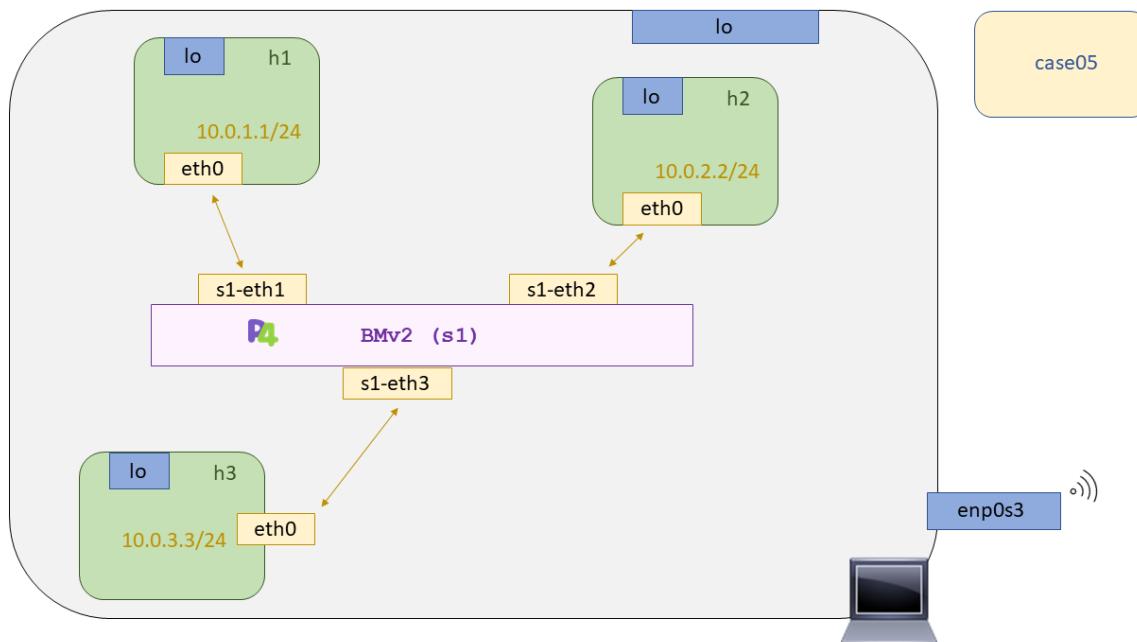


Figura 4.29: Escenario del Case05 - P4

Volviendo de nuevo a la comprobación del funcionamiento del caso de uso, se tendrá la CLI de Mininet abierta, por lo que se procederá a abrir tres terminales, una por cada host de la topología. Esto se puede hacer conseguir siguiendo las indicaciones del bloque 4.52.

Código 4.52: Apertura de terminales - Case05

```
1 mininet> xterm h1 h2 h3
```

Cuando ya se tengan las tres terminales abiertas, se procederá a escuchar las interfaces de los **host2** y **host3** con la finalidad de comprobar si realmente los paquetes están siendo clonados por los puertos indicados por el grupo multicast. Se hará uso de la herramienta tcpdump, podríamos utilizar también Wireshark.

Código 4.53: Puesta en escucha - Case05

```
1 # Hacemos lo mismo en el host2 y host3
2 tcpdump -l
```

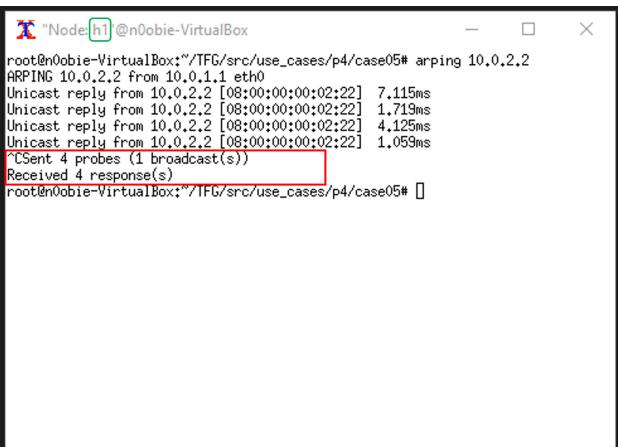
Una vez que se tenga tcmdump escuchando desde el **host2** y **host3**, se va a generar ARP-Request desde el **host1** para que así, al ir con MAC destino en difusión se propague por todos los posibles puertos del “switch” menos por aquel por el cual le llegó. Por tanto, para generar dicho ARP-Request se seguirán los pasos del bloque 4.54.

Código 4.54: Generación de ARP-Request - Case05

```
1 # Desde el host1
2 arping 10.0.2.2
```

Como se puede apreciar en la figura 4.30, el caso de uso funciona correctamente. Se puede observar cómo el paquete ARP-Request está llegando tanto al **host2** como al **host3**. Pero solo será el **host2**, en este caso, el que contesta ya que va dirigido a éste. Se puede ver que al **host3** solo le llega un ARP-Request, y después se detiene.

Esto es así ya que al completarse la resolución ARP el **host1** ya conoce la dirección MAC del **host2**, por tanto los ARP-Request que se generen a posteriori llevarán la MAC destino del **host2**, entonces el “switch” no lo difundirá por todos su puertos, se lo pasará directamente al **host2**.



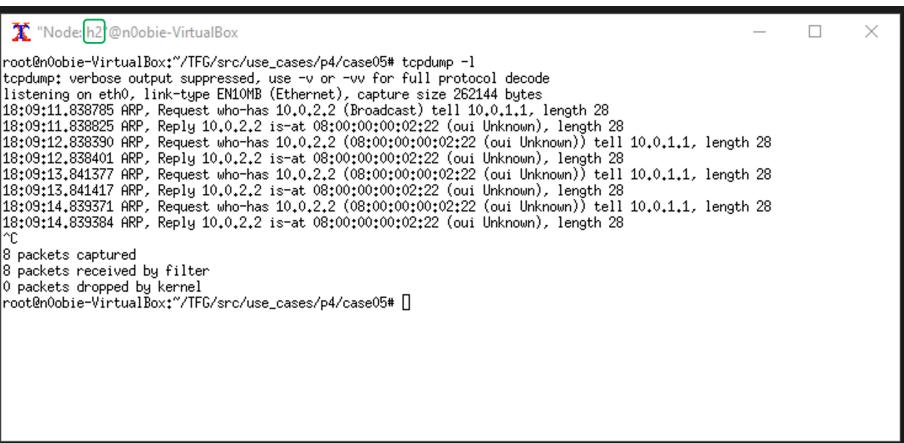
```
root@n0obie-VirtualBox:/TFG/src/use_cases/p4/case05# arping 10.0.2.2
ARPING 10.0.2.2 from 10.0.1.1 eth0
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 7.115ms
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 1.719ms
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 4.125ms
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 1.059ms
*Sent 4 probes (1 broadcast(s))
Received 4 response(s)
root@n0obie-VirtualBox:/TFG/src/use_cases/p4/case05#
```

(a) Ejecución de arping en el Host1 hacia el Host2



```
root@n0obie-VirtualBox:/TFG/src/use_cases/p4/case05# tcpdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
18:09:11.839278 ARP, Request who-has 10.0.2.2 (Broadcast) tell 10.0.1.1, length 28
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
root@n0obie-VirtualBox:/TFG/src/use_cases/p4/case05#
```

(b) Escucha con Tcpdump en el Host3



```
root@n0obie-VirtualBox:/TFG/src/use_cases/p4/case05# tcpdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
18:09:11.838785 ARP, Request who-has 10.0.2.2 (Broadcast) tell 10.0.1.1, length 28
18:09:11.838825 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
18:09:12.838390 ARP, Request who-has 10.0.2.2 (08:00:00:00:02:22 (oui Unknown)) tell 10.0.1.1, length 28
18:09:12.838401 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
18:09:13.841377 ARP, Request who-has 10.0.2.2 (08:00:00:00:02:22 (oui Unknown)) tell 10.0.1.1, length 28
18:09:13.841417 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
18:09:14.839374 ARP, Request who-has 10.0.2.2 (08:00:00:00:02:22 (oui Unknown)) tell 10.0.1.1, length 28
18:09:14.839384 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
^C
8 packets captured
8 packets received by filter
0 packets dropped by kernel
root@n0obie-VirtualBox:/TFG/src/use_cases/p4/case05#
```

(c) Escucha con Tcpdump en el Host2

Figura 4.30: Comprobación de funcionamiento del Case05 - P4

4.3. Casos de uso P4 en medios inalámbricos

En esta sección se introducirán todos los casos de uso realizados con la tecnología P4 en entornos inalámbricos. Todos ellos se han nombrado siguiendo la misma sintaxis que en el repositorio del TFG, alojado en GitHub. Para la **instalación de las dependencias** de la tecnología P4, se ha dejado escrito el anexo C.1 donde se detallan todos los pasos a seguir. De forma adicional, será necesario la instalación de Mininet-WiFi.⁷

Estos casos de uso, al igual que los anteriores, se han dividido en partes diferenciadas con la finalidad de que la lectura de estos sea más clara y ordenada. Puesto que muchos conceptos/desarrollos serán iguales que en los casos de uso P4 en entornos cableados, se referenciará directamente al lector a las secciones correspondientes.

- **Introducción:** En esta parte se abordará las explicaciones teóricas complementarias, explicaciones propias sobre el caso de uso y comentarios sobre el código P4 desarrollado.
- **Compilación:** En esta parte se explicará al lector cómo proceder para compilar el programa P4.
- **Puesta en marcha del escenario:** En esta parte se explicará cómo levantar el escenario.
- **Evaluación del funcionamiento:** Por último se hará una evaluación sobre el funcionamiento del caso de uso haciendo uso de la CLI de Mininet-WiFi.

Para que el lector pueda seguir el desarrollo de los casos de uso P4, a continuación se indica la tabla 4.5, la cual expone en qué ruta del repositorio del TFG se puede encontrar dicho caso de uso, y un vídeo demostración donde el autor va comentando paso a paso el caso de uso y su evaluación.

Caso de uso	Enlace al repositorio	Enlace al vídeo demostración
case01 - Drop	Enlace al código	Enlace al vídeo
case02 - Pass	Enlace al código	Enlace al vídeo
case03 - Echo server	Enlace al código	Enlace al vídeo
case04 - Layer 3 forwarding	Enlace al código	Enlace al vídeo
case05 - Broadcast	Enlace al código	Enlace al vídeo

Tabla 4.5: Resumen de la documentación sobre los casos de uso P4 en entornos inalámbricos

En esta ocasión, como ya se comentaba en el capítulo de Análisis y Diseño (3), la plataforma que se utilizará para evaluar el funcionamiento de los programas P4 en entornos cableados será Mininet-WiFi. Pero como se indicó, la plataforma no contempla ningún tipo de nodo que de soporte al BMv2, por lo que será necesario desarrollar previamente una integración del BMv2 en Mininet-WiFi.

⁷Más adelante se indicará que versión se debe instalar.

4.3.1. Integración del BMv2 en Mininet-WiFi

En esta subsección se abordará la integración del BMv2 en Mininet-WiFi. La integración se ha dividido en tres partes. En las dos primeras se estudiará y analizará, conceptos básicos y desarrollos previos que serán de gran utilidad a la hora de abordar el desarrollo de la interfaz BMv2 en Mininet-WiFi.

4.3.1.1. Análisis de la interfaz BMv2 - Mininet

Desde el equipo de *p4lang* se quiso suministrar un entorno de pruebas donde se pudiera probar los programas P4 desarrollados. El soft-switch donde iban a cargar el programa P4 ya lo tenían, que es el BMv2, pero les faltaba una plataforma donde poder desplegar dicho “switch” e interconectarlo con otras entidades de red. Esta plataforma sería Mininet, una herramienta de emulación de redes.

Mininet generalmente se utiliza para emular entornos SDN con switches y controladores. Por ello, para lograr la integración de su nuevo switch con los switches ya disponibles en Mininet, tuvieron que añadir una nueva clase llamada **P4Switch**. Esta nueva clase, heredaría de la clase **Switch** de Mininet, añadiendo así todos los métodos y atributos necesarios para la orquestación del BMv2.

Usualmente, cuando se trabaja con Mininet se desarrollan scripts donde se define la topología de la red a emular. En este caso, para brindar de una mayor facilidad a los usuarios de los tutoriales de P4⁸, las topologías se definen en archivos ***.json**, los cuales definen toda la topología de red y toda la información del plano de control de cada “switch” P4. Un ejemplo de dicho archivo pueden encontrarse [aquí](#). De este modo, se consigue abstraer el la definición de la topología del propio script de orquestación de la misma. El script de orquestación de la topología el cual hará uso de la API de Python de Mininet será el script **run_exercise.py**, que se puede encontrar [aquí](#). De esta manera se tendrá tantos ficheros ***.json** como topologías se quieran, pero un único script de orquestación.

El script **run_exercise.py** inicializa un objeto de la clase **ExerciseRunner** el cual leerá el fichero ***.json**, procesará la topología con la ayuda de la clase **ExerciseTopo** y levantará la topología haciendo uso de la API de Python de Mininet. A continuación, en la figura 4.31 se puede apreciar un diagrama UML donde se indica la relación de clases del entrypoint en el levantamiento de los distintos casos de uso.

Como ya se comentaba anteriormente, para lograr la integración del BMv2 en Mininet se tuvo que crear la clase **P4Switch**. Esta nueva clase, heredaría de la clase **Switch** de Mininet. A su vez, se crearán nuevas clases hijas de la clase **P4Switch**, la más importante **P4RuntimeSwitch** la cual se utilizaría para configurar dicho “switch” vía P4Runtime. Si se desea ver una vista más amplia de esta relación de clases, ir a la figura 4.32.

⁸<https://github.com/p4lang/tutorials>

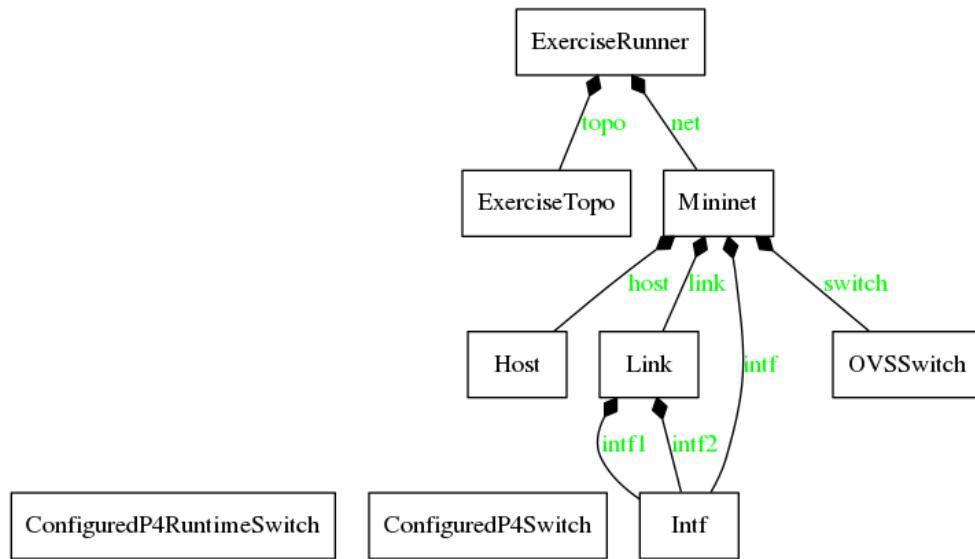


Figura 4.31: UML del punto de entrada de la interfaz BMv2 - Mininet

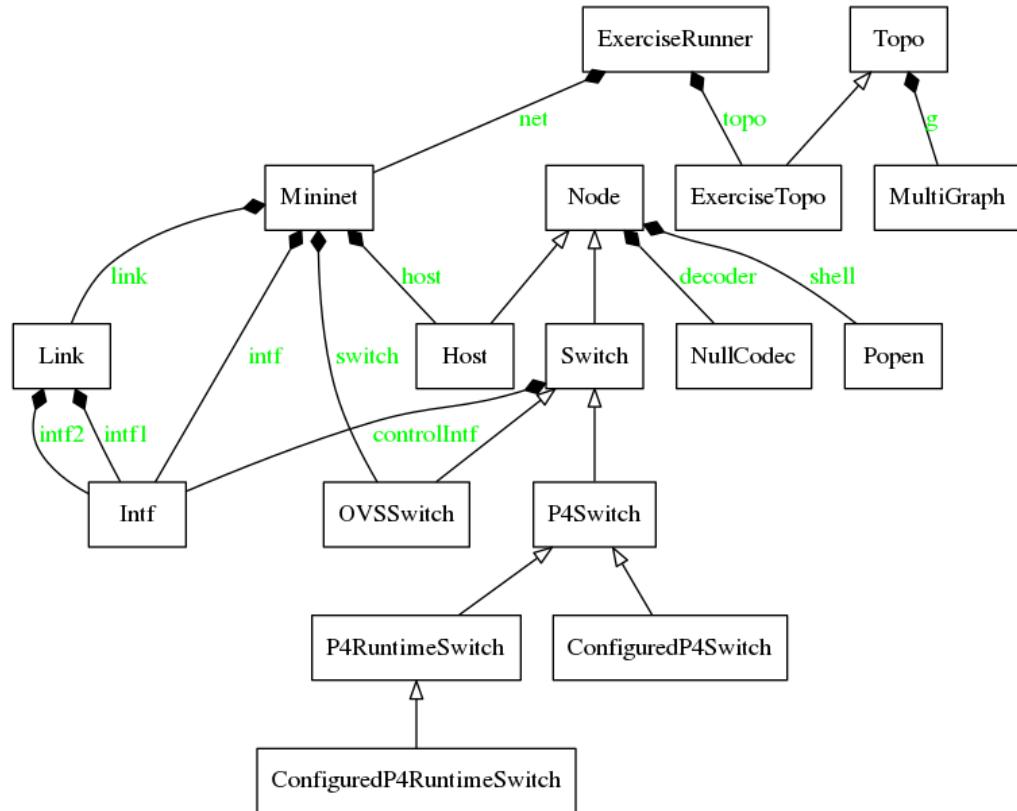


Figura 4.32: UML interfaz BMv2 - Mininet

4.3.1.2. Análisis del funcionamiento interno de Mininet-WiFi

Una vez que se revisó el funcionamiento interno de la API de Mininet-P4, se da paso a analizar Mininet-WiFi en profundidad. Como se comentaba en el capítulo 2, Mininet-WiFi es una herramienta de emulación de redes wireless que ha nacido de Mininet, es decir es un fork de este. Al ser un fork comparte gran parte de la jerarquía de clases así como su capacidad para simular ciertos elementos de la red. Aun así, si bien se ha mencionado en numerosas ocasiones que Mininet-WiFi esta desarrollado a partir de Mininet, la capacidad de emulación de interfaces *Wireless* la toma de subsistema wireless de Linux.

La arquitectura de virtualización empleada en Mininet-WiFi funciona de forma similar a la Mininet; se hace uso de la herramienta `mnexec`⁹ para lanzar distintos procesos de bash en nuevas *Network Namespaces*, uno por cada nodo independiente de la red. De estos procesos colgarán todos los procesos relativos a los distintos nodos de la red. Cuando la emulación haya terminado, se matarán dichos procesos de bash, consiguiendo que no haya ninguna condición de referenciación de las *Network Namespaces*, y estas sean eliminadas por el Kernel.

De esta manera, los nodos de la red ya estarían aislados entre sí, por lo que lo único que quedaría por virtualizar son las capacidades *wireless* de los nodos que las requieran. Para ellos se hará uso del subsistema *wireless* del Kernel de Linux, más concretamente el módulo `mac80211_hwsim` el cual creará las interfaces *wireless* en nuestro equipo. Este módulo se comunicará con framework `mac80211` el cual proveerá de las capacidades de gestión de acceso al medio de la interfaz *wireless*. Además, en el espacio de Kernel aun hay un bloque más, llamado `cfg80211`, el cual servirá de API para la configuración de las cabeceras 802.11. Esta configuración puede ser realizada por la interfaz netlink de espacio de usuario llamada `nl80211`. Para la configuración de los puntos de acceso, se hará uso del programa `HostApd` el cual indicándole la configuración del punto de acceso y la interfaz sobre la cual debe correr, emulará el funcionamiento de un punto de acceso estándar. En la figura 4.33 se puede ver de manera resumida la arquitectura básica de Mininet-WiFi.

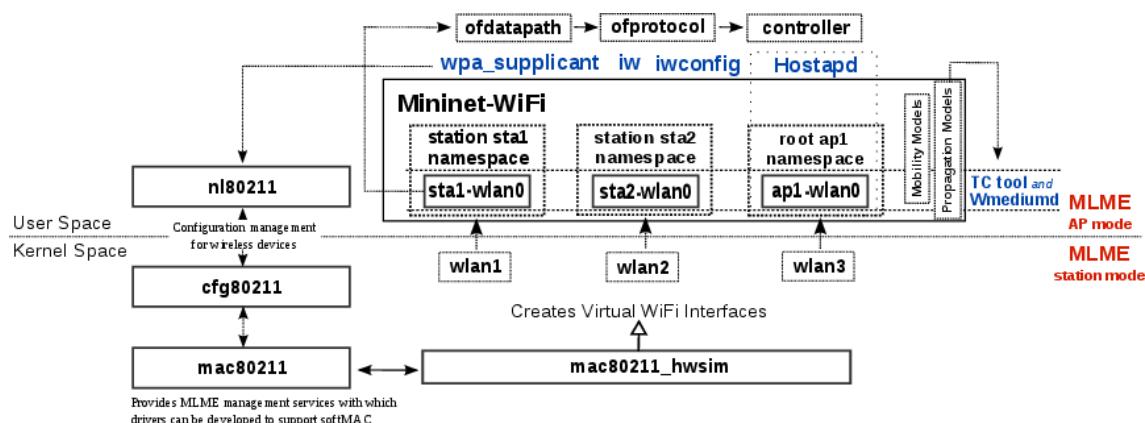


Figura 4.33: Arquitectura Mininet-WiFi [?]

⁹<https://github.com/intrig-unicamp/mininet-wifi/blob/master/mnexec.c>

En cuanto a la jerarquía de clases, únicamente cabe mencionar que es bastante similar a la de Mininet. Por destacar dos clases claves en la jerarquía de Mininet-WiFi, serían Node_Wifi, de la cual heredan todos los nodos con capacidades wireless que posee Mininet-WiFi y por último, la clase IntfWireless, de la cual heredan todos los tipos de enlaces disponibles de Mininet-WiFi (Bajo el estándar ieee80211). A continuación, en las figuras 4.34 y 4.35, se indican los UML referentes a dichas clases.

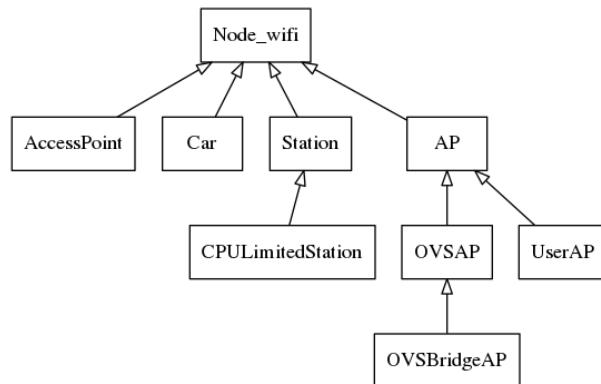


Figura 4.34: UML sobre las relaciones de las clases tipo Nodo.

Como se puede apreciar en los esquemas UML, se ha conseguido aislar la funcionalidad común en las clases padres, con la finalidad de optimizar la cantidad de código de las clases hijas. De esta forma, añadir nuevos tipos de enlaces y nodos en Mininet-WiFi resulta bastante asequible.

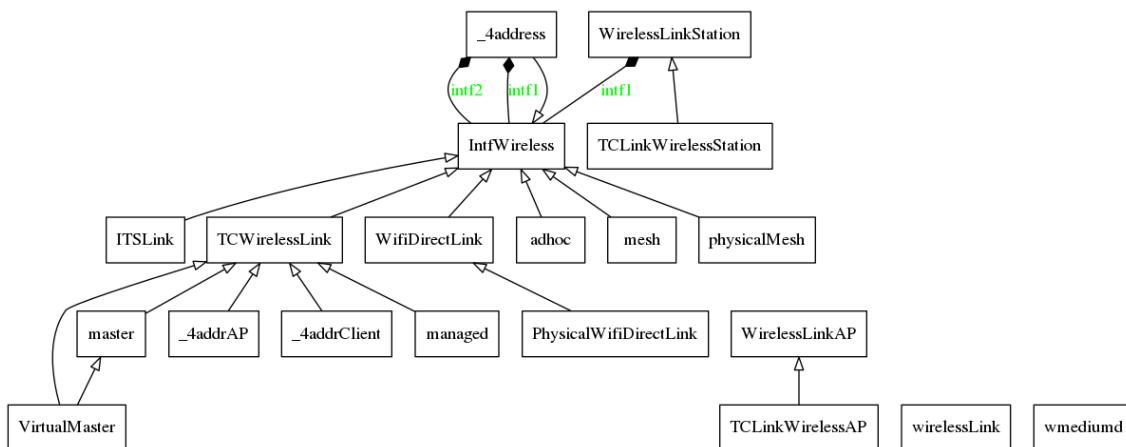


Figura 4.35: UML sobre las relaciones de las clases de tipo Interfaz.

Linux Wireless Subsystem

El subsistema *wireless* de Linux consiste en un set de varios módulos que se encuentran en el Kernel de Linux. Estos manejan la configuración del hardware bajo el estándar `ieee80211`, además de la gestión de la transmisión y la escucha de los paquetes de datos. Yendo de abajo hacia arriba en la arquitectura del subsistema, el primer bloque que se encuentra es el módulo `mac80211_hwsim`. Este módulo como ya se comentaba es el responsable de crear las interfaces *wireless* virtuales en Mininet-WiFi.

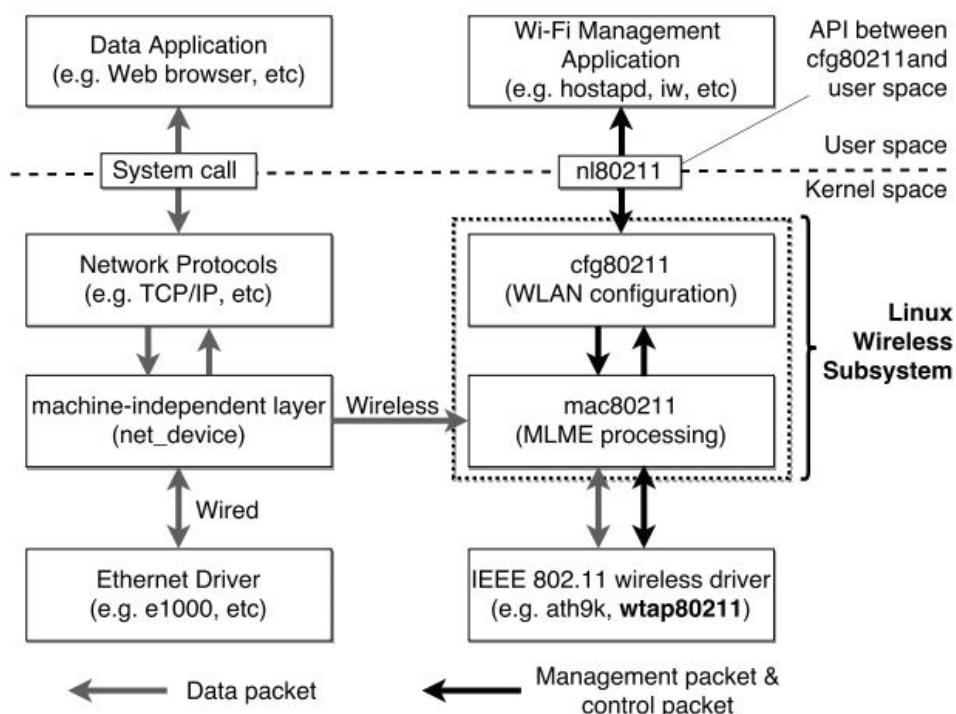


Figura 4.36: Arquitectura del subsistema Wireless de Linux [?]

El objetivo principal de este módulo `mac80211_hwsim` es facilitar a los desarrolladores de drivers de tarjetas *wireless* la prueba de su código e interacción con el siguiente bloque llamado `mac80211`. Las interfaces virtualizadas no tienen ciertas limitaciones, es decir, a diferencia del hardware real, resulta más sencillo la creación de distintas pruebas con distintas configuraciones sin estar cohibidos por falta de recursos materiales. Este módulo generalmente recibe un único parámetro, que es el número de “radios”, interfaces virtuales, a virtualizar. Dado que las posibilidades que ofrece este módulo eran un poco reducidas, muchos *wrappers* (“envoltorios al software original”) han sido creados para ofrecer más funcionalidad a parte de la dada por el propio módulo. La mayoría de herramientas creadas hacen uso de la librería Netlink para comunicarse directamente con el subsistema en el Kernel y así conse-

uir configuraciones extra, como pueden ser añadir un Received Signal Strength Indicator (RSSI) o darle nombre a la interfaz. Un ejemplo de dichas herramientas sería la herramienta `mac80211_hwsim_mgmt`, la cual es usada por Mininet-WiFi para gestionar la creación de las interfaces *wireless* en cada nodo que las requiera.

Es importante mencionar el cambio de paradigma que existe en el subsistema *wireless* de Linux con el concepto de interfaz. Generalmente se está acostumbrado a pensar en el concepto de interfaz como un elemento que gestiona el acceso al medio, capa dos, y el propio hardware, capa física, un ejemplo de ello sería una interfaz de Ethernet. Bien, pues en el subsistema wireless se desglosa la interfaz en dos capas [?]. Una de ellas es la capa física (*PHY*) donde se puede gestionar por ejemplo en que canal está escuchando la tarjeta wireless emulada. La otra capa es el acceso al medio, representado por las interfaces virtuales que “cuelgan” de una tarjeta wireless.

La idea detrás de este paradigma, es que puede tener N interfaces virtuales asociadas a la misma tarjeta WiFi emulada, lo que resultó sorprendente es que las interfaces virtuales funcionan principalmente con Ethernet (dejando de lado las que están en modo monitor).

Limitaciones encontradas

Como ya se ha comentado, la mayoría de interfaces virtuales asociadas a una tarjeta wireless emulada son del tipo de Ethernet, por lo que todos los paquetes que llegan vienen con cabeceras Ethernet. Esto supone una limitación ya que en los casos de uso se quería gestionar las cabeceras WiFi, pero si todas las interfaces virtuales son generalmente del tipo Ethernet, no será posible.

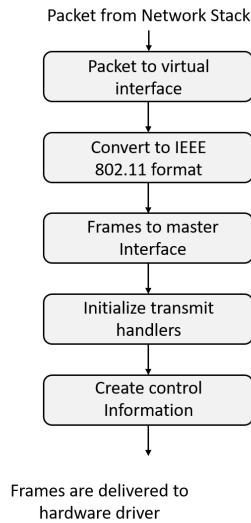


Figura 4.37: Flujo para la transmisión con el módulo `mac80211_hwsim` [?]

Pero, ¿Qué sentido tiene tener convertir las cabeceras WiFi a cabeceras Ethernet? De momento la única razón que se ha encontrado de esta decisión de diseño, es hacer un diseño

más sencillo de todos los drivers que operan bajo el módulo mac80211_hwsim, que convierten a Ethernet y se lo entregan al stack de red para que lo gestione como un paquete más de una red cableada. De esta forma, las aplicaciones que operen a nivel de interfaz serán más extrapolables.

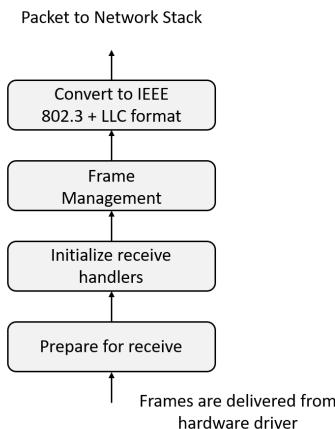


Figura 4.38: Flujo para la recepción con el módulo mac80211_hwsim [?]

Pero, hay que mencionar que esto supone un gasto de recursos considerable, ya que el paquete es encolado hasta tres veces (driver, ethernet queue, qdisc queue) y se tiene que invertir tiempo y recursos en el proceso de traducción de las cabeceras. [Aqui¹⁰](#) se puede ver la función en el kernel donde se lleva a cabo ese proceso.

Esto sucede generalmente y no siempre ya que en el único modo que una interfaz puede llegar a escuchar los paquetes WiFi es en el modo monitor. Pero el modo monitor está pensado únicamente para escuchar paquetes, no para transmitir. Este modo puede ser llevado al límite haciendo una inyección de paquetes (packet injection) por la interfaz.

De esta forma se conseguiría el objetivo de manejar las cabeceras WiFi, pero dadas las competencias del TFG, este desarrollo se escapa completamente. Ni si quiera Mininet-WiFi, siendo la herramienta de facto para emular redes *wireless*, contempla el manejo de cabeceras WiFi. Por ello, este objetivo se plantea como un desarrollo a posteriori y se mencionará en el trabajo futuro (Ver sección 5.2).

El desarrollo pues sobre Mininet-WiFi, consistirá en la integración del BMv2 en Mininet-WiFi, y probar los casos de uso desarrollados para P4, bajo Mininet-WiFi. Las cabeceras que llegarán a la pipeline serán las de Ethernet, pero como ya se ha explicado, es el propio Kernel quien se encarga de hacer un *casting* de las cabeceras WiFi hacia cabeceras Ethernet.

¹⁰https://elixir.bootlin.com/linux/latest/C/ident/_ieee80211_data_to_8023

4.3.1.3. Desarrollo de la interfaz BMV2 - Mininet-WiFi

Ahora que ya se tiene una idea sobre el funcionamiento interno de ambas herramientas, vamos a proceder a su integración. A la par que se estaba trabajando en el desarrollo de la integración del BMv2 en Mininet-WiFi, el investigador Ramon Fontes (desarrollador de la herramienta) abrió un Issue (es decir, una publicación de debate y ampliación de funcionalidad de GitHub) donde se exponía la intención de crear dicha integración. En este issue se pudo debatir con Ramon Fontes cómo debía hacerse dicho desarrollo.

En concreto, se decidió hacer un jerarquía de clases genérica para el BMv2 para que de estas se puedan crear clases personalizadas con añadidos nuevos. Por así decirlo, estas clases serán la base para clases que controlen el BMv2 de una forma más particular.

Una clase P4AP, la cual contiene todos los atributos y métodos comunes al BMv2 como son el path de ejecución del BMv2, el json compilado del programa P4, el thrift-port, configuración de log y identificador básico del BMv2. De esta clase se quiere que herede una clase llamada P4RuntimeAP, la cual proporcionará todos los elementos necesarios para dar soporte a la configuración vía P4Runtime vía gRPC-port del BMv2.

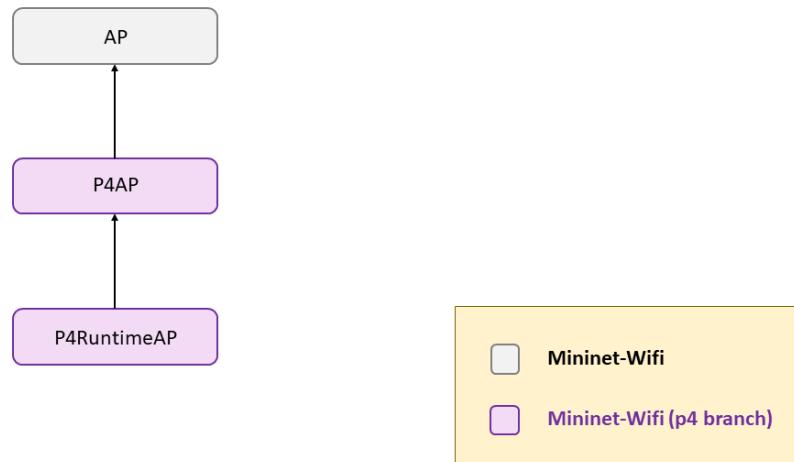


Figura 4.39: UML de la integración BMv2 - Mininet-WiFi

Además debatiendo con Ramon Fontes sobre la implementación de estas clases, indicó que sería de gran utilidad que dichas clases tuvieran soporte de ejecución en Network Namespaces, por lo que de forma paralela se tuvo que crear una clase llamada `Netns_mgmt`¹¹. Esta clase

¹¹https://github.com/davidcawork/TFG/tree/master/src/netns_mgmt

ayudará a gestionar la ejecución de código Python en *runtime* en una Network namespace indicada haciendo uso de la llamada al sistema `setns` la cual asocia el procesos sobre la cual se realiza esta llamada, a una *Namespace* indicada.

Con la ayuda de esta clase, `Netns_mgmt`, se pudo conseguir configurar cada BMv2 en su propia *Network Namespace*, por lo que la integración se dio por terminada. Adicionalmente se añadieron dos ejemplos a Mininet-WiFi con la finalidad de ayudar a las personas que vayan hacer uso de clase. Dichos ejemplos pueden ser encontrados [aquí](#)¹²

Todo este desarrollo se llevó a cabo en un fork de Mininet-WiFi, y dentro de este en una rama en particular, en la cual se llevan a cabo todos los desarrollos de P4. Una vez finalizada la integración, se ofreció el desarrollo al repositorio oficial de Mininet-WiFi vía pull-request. Actualmente se ha dejado a la espera de hacer un *upgrade* a las dependencias donde fue llevada a cabo la integración, ya que Mininet-WiFi está trabajando con las últimas versiones. En cambio, para el desarrollo de los casos de uso P4-wireless, se hará uso de las últimas versiones estables de las dependencias del entorno P4.

Puesta en marcha del Mininet-WiFi modificado

Para poder probar el desarrollo llevado a cabo con Mininet-WiFi se deberán seguir los pasos indicados en el bloque 4.55 previamente. Se deberá bajar el repositorio del fork de Mininet-WiFi. Se hará un checkout para moverse desde la rama master del repositorio a la rama de desarrollo de elementos P4.

Código 4.55: Instalación de Mininet-WiFi modificado

```

1 # Se descarga el repositorio
2 git clone https://github.com/davidcawork/mininet-wifi
3
4 # Se adquieren las etiquetas y se cambia de rama
5 cd mininet-wifi && git fetch
6 git checkout p4
7
8 # Se "recompilará" Mininet-Wifi
9 sudo make install

```

Como la rama de desarrollo añade módulos nuevos a Mininet-WiFi se deberá “recompilar” de nuevo haciendo un make install desde el directorio `/mininet-wifi`. Para que los módulos añadidos funcionen correctamente se deberán tener las dependencias del entorno P4, en la tabla 4.6 se pueden comprobar que versiones son requeridas.

4.3.2. Case01 - Drop

En este caso de uso se probará que es posible descartar todos los paquetes recibidos con un programa P4 en una interfaz virtual *Wireless*. Como tal el programa P4 no es suficiente

¹²<https://github.com/davidcawork/mininet-wifi/tree/p4/examples/p4>

Dependencia	Versión requerida
BMV2	b447ac4c0cf83e5e72a3cc6120251c1e91128ab
PI	41358da0ff32c94fa13179b9cee0ab597c9ccbcc
P4C	69e132d0d663e3408d740aaf8ed534ecef88810
PROTOBUF	v3.2.0
GRPC	v1.3.2

Tabla 4.6: Resumen de las versiones requeridas de la interfaz BMv2 - Mininet-WiFi

para probar esta funcionalidad ya que requiere de una plataforma que sea capaz de soportar el lenguaje P4. Se hará uso de software switch llamado behavioral-model, BMv2 en adelante, para testear los programas P4, y de la integración desarrollada anteriormente con Mininet-WiFi como escenario para recrear las topologías de Red.

Como este caso de uso ya se ha explicado anteriormente en el case01 de P4 cableado (Ver subsección 4.2.1), y no hay ninguna diferencia inducida en el cambio de entorno según se explico en la sección anterior, únicamente se harán indicaciones sobre como poder compilarlo y ejecutarlo. Importante, si usted está replicando este caso de uso, sin antes haber adecuado las dependencias necesarias de Mininet-WiFi con soporte del BMv2, vuelva a este punto 4.3.1.3 y siga los pasos indicados.

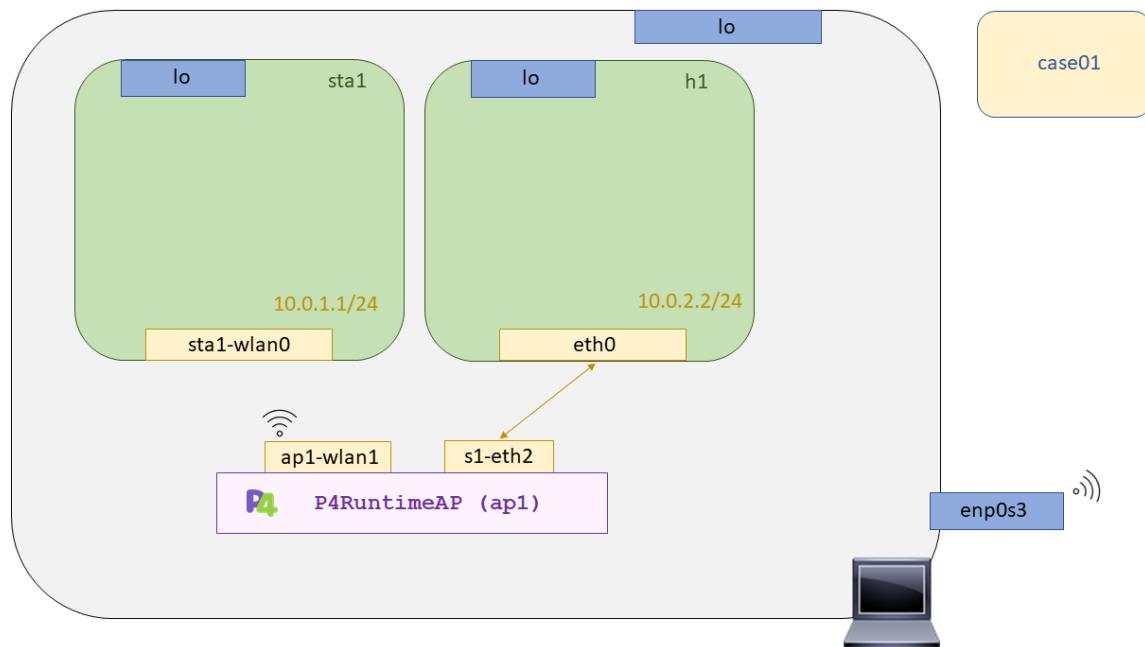


Figura 4.40: Escenario del Case01 - P4 Wireless

Compilación

Para la compilación de este caso de uso, se ha dejado preparado un Makefile, por tanto no es necesario que el usuario aprenda a utilizar el compilador p4c. Si se quiere saber más sobre como funciona el proceso de compilación, qué etapas hay, como se le ”inyecta” el json generado al BMv2, o qué distintos *targets* hay en función de la arquitectura, le recomendamos que vuelva al case01 (4.2.1). Para llevar a cabo la compilación solo se tendrá que seguir los pasos indicados en el bloque 4.56.

Código 4.56: Compilación programa P4 - Case01

```

1 # Entramos al directorio
2 cd TFG/src/use_cases/p4-wireless/case01
3
4 # Hacemos uso del Makefile
5 sudo make

```

Una vez ejecutado el make, se habrá generado una estructura de directorios que se utilizarán en el lanzamiento del caso de uso. Bajo el directorio build se podrá encontrar el json generado por el compilador, será este json quien tenga toda la información requerida para conformar el BMv2.

Puesta en marcha del escenario

Al igual que en la compilación, se ha dejado preparado un script en Python para automatizar la puesta en marcha del escenario. Este script describe la topología que se utilizará en este caso de uso. Recordemos que es necesario volver hacer un make install para instalar los módulos adicionales generados para la integración del BMv2 y Mininet-WiFi, además de tener instaladas las versiones indicadas en el análisis de la integración. Estas dependencias se pueden encontrar en el apartado 4.3.1.3

Una vez comprobado que posee todas la dependencias, simplemente se tendrá que ejecutar el script con el interprete de Python. Este script levantará la topología descrita en la figura 4.40, compuesto por dos hosts y por una instancia del nodo P4RuntimeAP. El nodo ap1, del tipo P4RuntimeAP, tendrá dos interfaces, una wireless y un par de Veth.

Código 4.57: Puesta en marcha del escenario - Case01

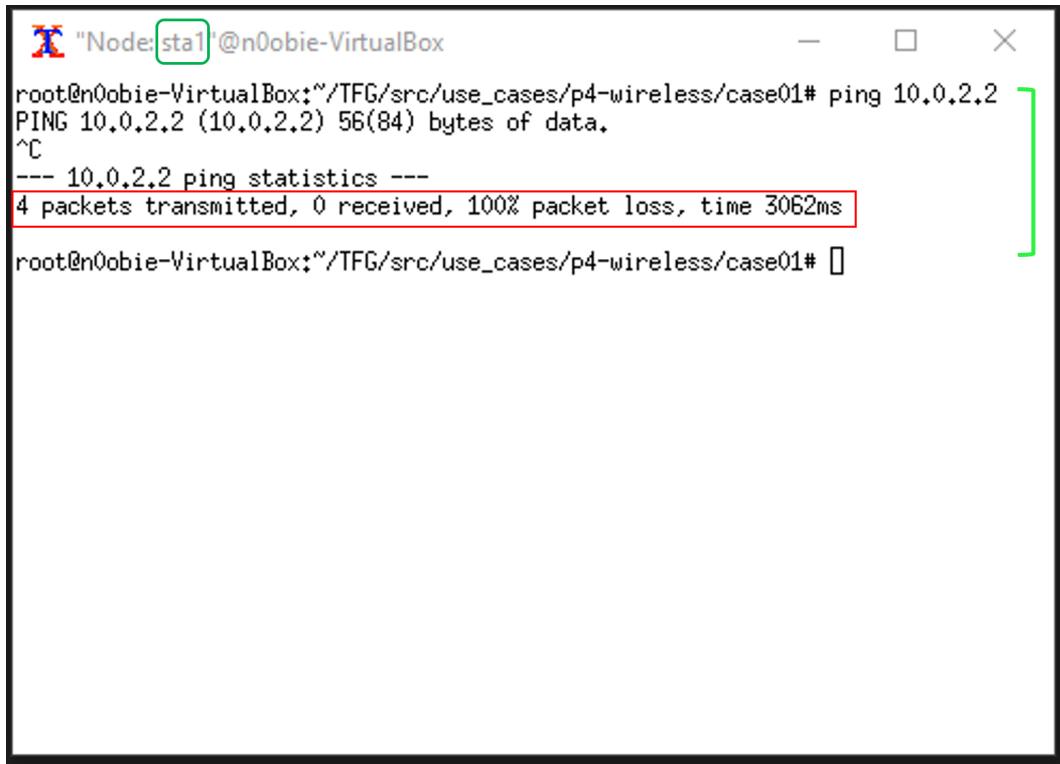
```
1 sudo python scenario.py
```

Comprobación del funcionamiento

Tras las ejecución del script `scenario.py`, se tendrá el escenario 4.40 levantado, y la CLI de Mininet-WiFi abierta. Para la comprobación de funcionamiento de este caso de uso, se van a seguir los mismos pasos que en el case01 (4.2.1) - P4 en un entorno alámbrico. Por tanto no se entrará en hacer explicaciones que se creen redundantes, se indicarán los pasos seguidos para llevar a cabo la comprobación de funcionamiento y los resultados de dichas pruebas.

Código 4.58: Pasos a seguir para comprobar el funcionamiento - Case01

```
1 mininet-wifi> xterm sta1 h1
2
3 # Se hará ping desde la estación Wifi al Host
4 [sta1] ping 10.0.2.2
5
6
7 # Se pondrá a escuchar en el Host
8 [h1] tcpdump -l
```



```
X "Node: sta1'@n0obie-VirtualBox" - X
root@n0obie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case01# ping 10.0.2.2
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
^C
--- 10.0.2.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3062ms
root@n0obie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case01#
```

Figura 4.41: Comprobación de funcionamiento (Ping) del Case01 - P4 Wireless

Como se puede ver en la figura 4.41, no hay conectividad entre ambos nodos dado que el ping no está llegando. Por lo que, el programa P4 está funcionando según lo esperado, tirando los paquetes que llegan a su *pipeline* de procesamiento. Para asegurarse de que realmente se está ejecutando el *action* para tirar paquetes podemos consultar los logs del BMv2, los cuales generan un fichero de log por cada instancia *target* del P4RuntimeAP que se haya levantado.

```

X "Node h1" @n0obie-VirtualBox
root@n0obie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case01# tcpdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C
0 packets captured
0 packets received by filter
0 packets dropped by kernel
root@n0obie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case01# 

```

Figura 4.42: Comprobación de funcionamiento (Sniffer) del Case01 - P4 Wireless

4.3.3. Case02 - Pass

Como se puede apreciar, en este directorio no hay ningún programa P4, al igual que en caso de uso P4, case02 (4.2.2), en entornos cableados. Esto se debe a que no hay equivalente en P4 del código de retorno `XDP_PASS`, por ello, no se puede hacer nada en este caso de uso. El código de retorno en XDP es una forma para llevar a cabo una acción con el paquete que llega a la interfaz, en la cual hay anclado un programa XDP. En este caso el código de retorno `XDP_PASS` implica que el paquete se pasa al siguiente punto del procesamiento del *stack* de red en el Kernel de Linux. Es decir, si el programa está anclado a la NIC, se dejará pasar el paquete al TC, de ahí al propio *stack* de red para parsear sus cabeceras, y más adelante, dárselo de comer a la interfaz de sockets.

En P4, el entorno donde se llevaran a cabo los casos de uso será Mininet-WiFi con APs (BMv2) y host. Los APs (BMv2) son un software switch en los que podemos inyectar código P4, con el cual podemos definir el *datapath* del mismo.

Ahora bien, aquí viene la gran diferencia entre ambas tecnologías, con XDP siempre es posible pasarlo al Kernel para que se encargue el del procesamiento, pero en P4, debemos definir nosotros de forma exclusiva todo el *datapath*, por lo que no hay a quien delegar el paquete, se debe encargar el propia programa P4. Entonces, como tal, no habrá equivalente del `XDP_PASS` en P4, como es obvio ni en escenarios cableados ni inalámbricos, es una característica de la propia tecnología.

4.3.4. Case03 - Echo server

En este caso de uso desarrollaremos un servidor de echo que responda todos los pings que le lleguen al BMv2. Como tal el programa P4 no es suficiente para probar esta funcionalidad ya que requiere de una plataforma que sea capaz de soportar el lenguaje P4. Se hará uso

de software switch BMv2, para testear los programas P4, y de la integración desarrollada anteriormente con Mininet-WiFi como escenario para recrear las topologías de Red. Como el BMv2 será configurado vía P4Runtime se hará uso de la clase de `P4RuntimeAP`.

Dado que este caso de uso ya se ha explicado anteriormente en el case03 de P4 cableado (Ver subsección 4.2.3), y no hay ninguna diferencia inducida en el cambio de entorno según se explico en las sección anterior, únicamente se harán indicaciones sobre como poder compilarlo y ejecutarlo. Importante, si usted está replicando este caso de uso, sin antes haber adecuado las dependencias necesarias de Mininet-WiFi con soporte del BMv2, vuelva a este punto 4.3.1.3 y siga los pasos indicados.

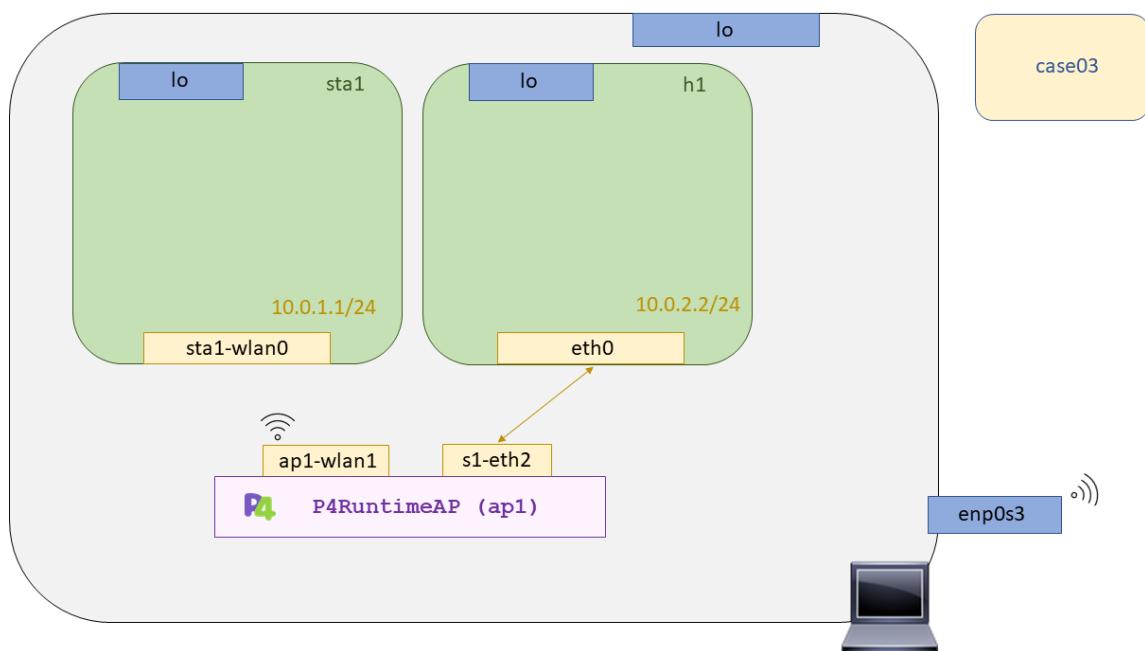


Figura 4.43: Escenario del Case03 - P4 Wireless

Compilación

Para la compilación de este caso de uso, al igual que en los casos de uso anteriores se ha dejado preparado un Makefile, por tanto no es necesario que el usuario haga un uso directo del compilador p4c. Si se quiere saber más sobre como funciona el proceso de compilación, qué etapas hay, como se le “inyecta” el json generado al BMv2, o qué distintos *targets* hay en función de la arquitectura, le recomendamos que vuelva al case01 (4.2.1). Para llevar a cabo la compilación solo se tendrá que seguir los pasos indicados en el bloque 4.59.

Código 4.59: Compilación programa P4 - Case03

```

1 # Entramos al directorio
2 cd TFG/src/use_cases/p4-wireless/case03
3
4 # Hacemos uso del Makefile
5 sudo make

```

Una vez ejecutado el make, se habrá generado una estructura de directorios que se utilizarán en el lanzamiento del caso de uso. Bajo el directorio `build` se podrá encontrar el `json` generado por el compilador, será este `json` quien tenga toda la información requerida para conformar el BMv2. Los directorios `log` y `pcap`, se utilizarán respectivamente para almacenar los logs del BMv2 y para guardar las capturas de paquetes de las interfaces asociadas a la instancia BMv2.

Puesta en marcha del escenario

Al igual que en la compilación, se ha dejado preparado un script en Python para automatizar la puesta en marcha del escenario. Este script describe la topología que se utilizará en este caso de uso. Recordar que es necesario volver hacer un `make install` para instalar los módulos adicionales generados para la integración del BMv2 y Mininet-WiFi, además de tener instaladas las versiones indicadas en el análisis de la integración. Estas dependencias se pueden encontrar en el apartado 4.3.1.3. Una vez comprobado que posee todas la dependencias, simplemente se tendrá que ejecutar el script con el interprete de Python. Este script levantará la topología descrita en la figura 4.43, compuesto por dos estaciones WiFi y por una instancia del nodo `P4RuntimeAP`. El nodo `ap1`, del tipo `P4RuntimeAP`, tendrá dos interfaces, una wireless y un par de Veth.

Código 4.60: Puesta en marcha del escenario - Case03

```
1 sudo python scenario.py
```

Comprobación del funcionamiento

Tras las ejecución del script `scenario.py`, se tendrá el escenario 4.43 levantado, y la CLI de Mininet-WiFi abierta. Para la comprobación de funcionamiento de este caso de uso, se van a seguir los mismos pasos que en el case03 (4.2.3) - P4 en un entorno alámbrico, adaptados en Mininet-WiFi. Por tanto solo se indicarán los pasos seguidos para llevar a cabo la comprobación de funcionamiento y los resultados de dichas pruebas.

Código 4.61: Pasos a seguir para comprobar el funcionamiento - Case03

```

1 mininet-wifi> xterm sta1
2
3 # Se hará ping desde la estación Wifi al Host.
4 [sta1] wireshark &
5 [sta1] ping 10.0.2.2

```

Como se puede ver en la figura 4.44, todo funciona correctamente ya que se están recibiendo respuesta a los pings  . De forma adicional, se puede comprobar con Wireshark como el

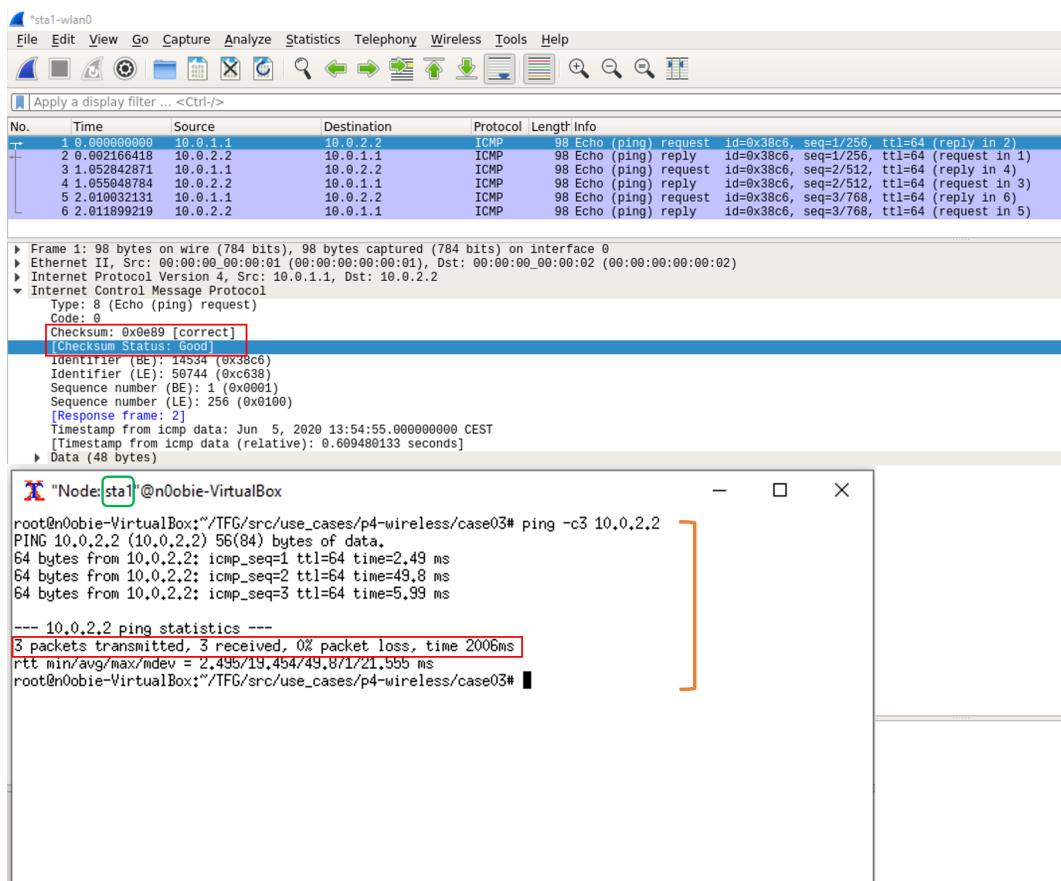


Figura 4.44: Comprobación de funcionamiento del Case03 - P4 Wireless

checksum de la cabecera ICMP viene correctamente calculado.

4.3.5. Case04 - Layer 3 forwarding

En este caso de uso trataremos de implementar un forwarding a nivel de red, capa 3, por ello nuestro hipotético "switch" será ahora un router muy básico y con muy pocas funcionalidades. Al igual que en los casos de uso anteriores, se hará uso de software switch llamado BMv2, para testear los programas P4, y de la integración desarrollada anteriormente con Mininet-WiFi como escenario para recrear las topologías de Red. En este caso también, el BMv2 será configurado vía P4Runtime por lo que se hará uso de la clase de P4RuntimeAP. En el caso de querer configurar el BMv2 vía Thrift-port se podría utilizar la clase desarrollada P4AP.

Dado que toda la información relativa a este caso de uso ya se ha explicado anteriormente en el case04 de P4 cableado (Ver subsección 4.2.4), y no hay ninguna diferencia inducida en el cambio de entorno según se explico en las sección anterior, únicamente se harán indicaciones sobre como poder compilarlo y ejecutarlo. Importante, si usted está replicando este caso de uso, sin antes haber adecuado las dependencias necesarias de Mininet-WiFi con soporte del

BMv2, vuelva a este punto 4.3.1.3 y siga los pasos indicados.

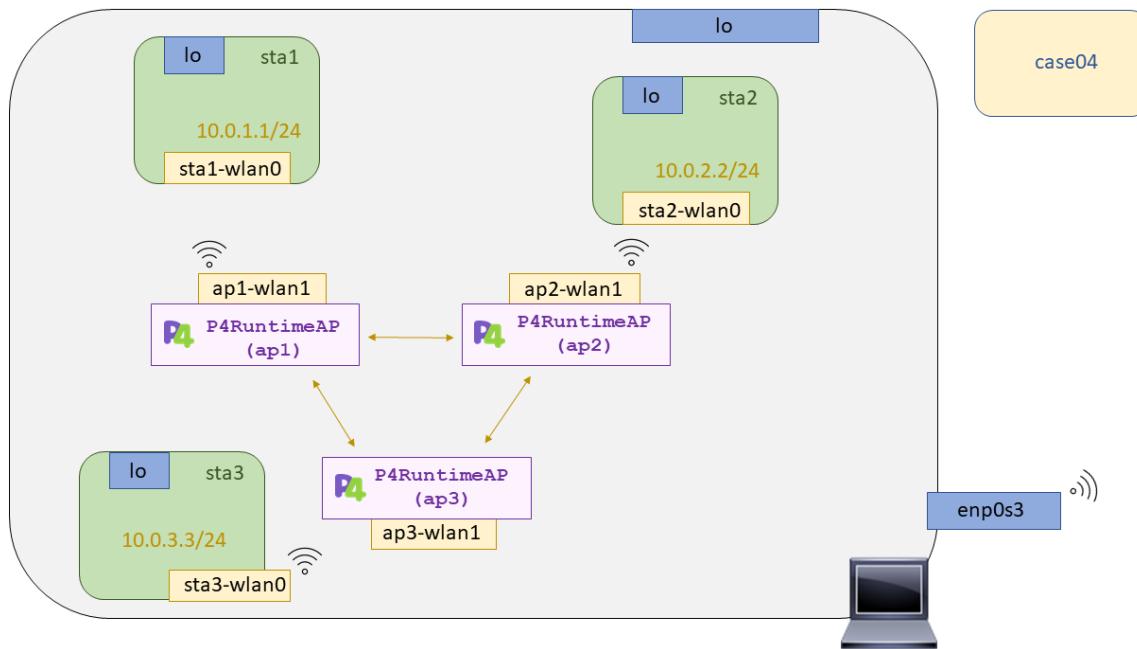


Figura 4.45: Escenario del Case04 - P4 Wireless

Compilación

Para la compilación de este caso de uso, al igual que en los casos de uso anteriores se ha dejado preparado un Makefile, por tanto no es necesario que el usuario haga un uso directo del compilador p4c. Si se quiere saber más sobre como funciona el proceso de compilación, qué etapas hay, como se le "inyecta" el json generado al BMv2, o qué distintos *targets* hay en función de la arquitectura, le recomendamos que vuelva al case01 (4.2.1). Para llevar a cabo la compilación solo se tendrá que seguir los pasos indicados en el bloque 4.62.

Código 4.62: Compilación programa P4 - Case04

```

1  # Entramos al directorio
2  cd TFG/src/use_cases/p4-wireless/case04
3
4  # Hacemos uso del Makefile
5  sudo make

```

Una vez ejecutado el make, se habrá generado una estructura de directorios que se utilizarán en el lanzamiento del caso de uso. Bajo el directorio `build` se podrá encontrar el json generado por el compilador, será este json quien tenga toda la información requerida para conformar el BMv2. Los directorios `log` y `pcap`, se utilizarán respectivamente para almacenar los logs del BMv2 y para guardar las capturas de paquetes de las interfaces asociadas a la instancia BMv2. Comentar que la interfaz desarrollada admite opciones para no generar

información de log, ni pcaps, se podrá configurar a medida desde el propio script que lanza el escenario.

Puesta en marcha del escenario

Al igual que en la compilación, se ha dejado preparado un script en Python para automatizar la puesta en marcha del escenario. Este script describe la topología que se utilizará en este caso de uso. Recordar que es necesario volver hacer un `make install` para instalar los módulos adicionales generados para la integración del BMv2 y Mininet-WiFi, además de tener instaladas las versiones indicadas en el análisis de la integración. Estas dependencias se pueden encontrar en el apartado 4.3.1.3. Una vez comprobado que posee todas la dependencias, simplemente se tendrá que ejecutar el script con el interprete de Python. Este script levantará la topología descrita en la figura 4.45, compuesto por tres estaciones WiFi y por tres instancias del nodo P4RuntimeAP. Los nodos `apX`, del tipo P4RuntimeAP, tendrán tres interfaces, una wireless y dos par de Veth para comunicarse entre los distintos puntos de acceso.

Código 4.63: Puesta en marcha del escenario - Case04

```
1 sudo python scenario.py
```

Comprobación del funcionamiento

Tras la ejecución del script `scenario.py`, se tendrá el escenario 4.45 levantado, y la CLI de Mininet-WiFi abierta. Para la comprobación de funcionamiento de este caso de uso, se van a seguir los mismos pasos que en el case05 (4.2.5) - P4 en un entorno alámbrico. Por tanto no se entrará a hacer explicaciones que se creen redundantes, se indicarán los pasos seguidos para llevar a cabo la comprobación de funcionamiento y los resultados de dichas pruebas.

Código 4.64: Pasos a seguir para comprobar el funcionamiento - Case04

```
1 mininet-wifi> pingall
```

Según se puede apreciar en la figura 4.46, hay perfecta conectividad entre todos los hosts de la topología. Además, se podría hacer uso de un *sniffer* para comprobar que los paquetes llegan con el campo `ttl` modificado, en función de los saltos que ha dado el paquete.

4.3.6. Case05 - Broadcast

En este test desarrollaremos un programa p4 que haga Broadcast. En este caso únicamente haremos Broadcast a nivel de enlace, capa 2. La motivación de este caso de uso es ver la diferencia de dificultad respecto del entorno XDP donde tuvimos que anclar de manera adicional un *bytecode* eBPF en el TC además de propio programa XDP anclado en la interfaz para lograr hacer un broadcast. Al igual que en los casos de uso anteriores, se hará uso de software switch llamado BMv2, para testear los programas P4, y de la integración desarrollada anteriormente con Mininet-WiFi como escenario para recrear las topologías de Red.

```

mininet-wifi> dump
<Station sta1: sta1-wlan0:10.0.1.1,sta1-wlan0:None,sta1-wlan0:None pid=15803>
<Station sta2: sta2-wlan0:10.0.2.2,sta2-wlan0:None,sta2-wlan0:None pid=15805>
<Station sta3: sta3-wlan0:10.0.3.3,sta3-wlan0:None,sta3-wlan0:None pid=15807>
<P4RuntimeAP ap1: lo:127.0.0.1,ap1-wlan1:None,ap1-eth2:None,ap1-eth3:None pid=15809>
<P4RuntimeAP ap2: lo:127.0.0.1,ap2-wlan1:None,ap2-eth2:None,ap2-eth3:None pid=15814>
<P4RuntimeAP ap3: lo:127.0.0.1,ap3-wlan1:None,ap3-eth2:None,ap3-eth3:None pid=15819>
mininet-wifi> pingall
*** Ping: testing ping reachability
sta1 -> *** sta1 : ('ping -cl 10.0.2.2',)
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=62 time=2.45 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.458/2.458/2.458/0.000 ms
sta2 *** sta1 : ('ping -cl 10.0.3.3',)
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=1.96 ms

--- 10.0.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.969/1.969/1.969/0.000 ms
sta3
sta2 -> *** sta2 : ('ping -cl 10.0.1.1')
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=62 time=1.89 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.896/1.896/1.896/0.000 ms
sta1 *** sta2 : ('ping -cl 10.0.3.3')
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=1.51 ms

--- 10.0.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.514/1.514/1.514/0.000 ms
sta3
sta3 -> *** sta3 : ('ping -cl 10.0.1.1',)
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=62 time=1.87 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.870/1.870/1.870/0.000 ms
sta1 *** sta3 : ('ping -cl 10.0.2.2')
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=62 time=1.72 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.723/1.723/1.723/0.000 ms
sta2
*** Results: 0% dropped (6/6 received)
mininet-wifi> 
```

Figura 4.46: Comprobación de funcionamiento del Case04 - P4 Wireless

Dado que toda la información relativa a este caso de uso ya se ha explicado anteriormente en el case05 de P4 cableado (Ver subsección 4.2.5), y no hay ninguna diferencia inducida en el cambio de entorno según se explico en las sección anterior, únicamente se harán indicaciones sobre como poder compilarlo y ejecutarlo. Importante, como ya se ha indicado, si usted está replicando este caso de uso, sin antes haber adecuado las dependencias necesarias de Mininet-WiFi con soporte del BMv2, vuelva a este punto 4.3.1.3 y siga los pasos indicados.

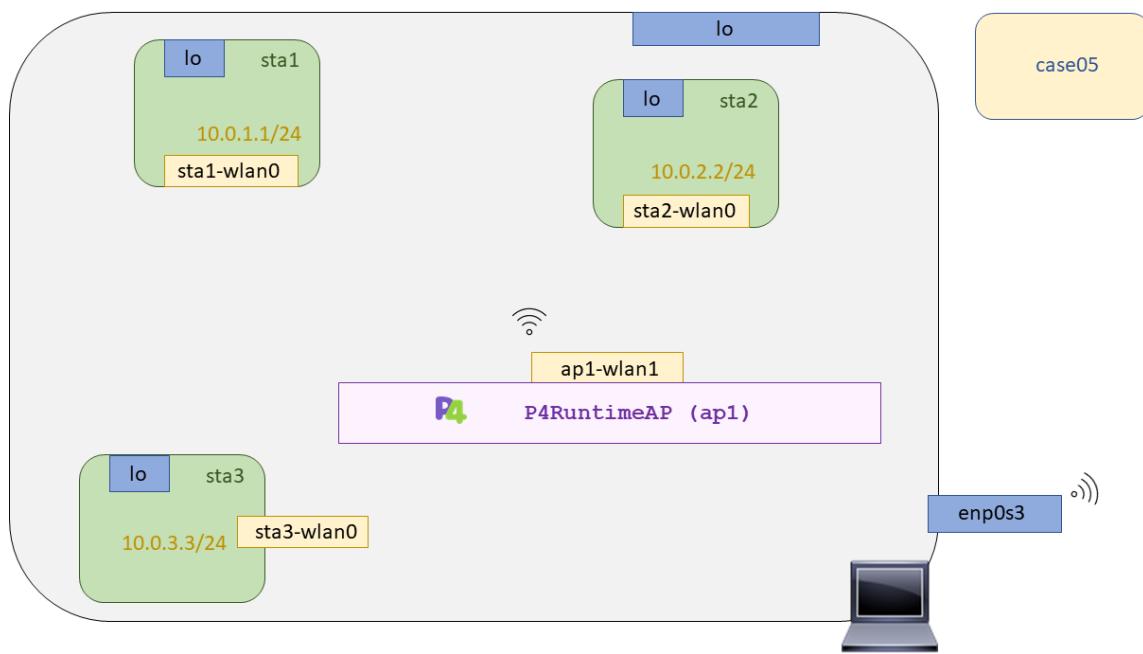


Figura 4.47: Escenario del Case05 - P4 Wireless

Compilación

Para la compilación de este caso de uso, al igual que en los casos de uso anteriores se ha dejado preparado un Makefile, por tanto no es necesario que el usuario haga un uso directo del compilador p4c. Si se quiere saber más sobre como funciona el proceso de compilación, qué etapas hay, como se le "inyecta" el json generado al BMv2, o qué distintos *targets* hay en función de la arquitectura, le recomendamos que vuelva al case01 (4.2.1). Para llevar a cabo la compilación solo se tendrá que seguir los pasos indicados en el bloque 4.62.

Código 4.65: Compilación programa P4 - Case05

```

1 # Entramos al directorio
2 cd TFG/src/use_cases/p4-wireless/case05
3
4 # Hacemos uso del Makefile
5 sudo make

```

Una vez ejecutado el make, se habrá generado una estructura de directorios que se utilizarán en el lanzamiento del caso de uso. Bajo el directorio build se podrá encontrar el json generado por el compilador, será este json quien tenga toda la información requerida para conformar el BMv2. Los directorios log y pcap, se utilizarán respectivamente para almacenar los logs del BMv2 y para guardar las capturas de paquetes de las interfaces asociadas a la instancia BMv2.

Puesta en marcha del escenario

Al igual que en la compilación, se ha suministrado un script en Python para automatizar la puesta en marcha del escenario. Este script conformará la topología que se utilizará en este caso de uso. Se quiere recordar que es necesario volver hacer un `make install` para instalar los módulos adicionales generados para la integración del BMv2 y Mininet-WiFi, además de tener instaladas las versiones indicadas en el análisis de la integración. Estas dependencias se pueden encontrar en el apartado 4.3.1.3

Una vez comprobadas la dependencias, simplemente se tendrá que ejecutar el script con el interprete de Python. Este script levantará la topología descrita en la figura 4.47, compuesto por tres estaciones WiFi y por una instancia del nodo P4RuntimeAP. El nodo `ap1`, del tipo P4RuntimeAP, tendrá una interfaz, del tipo wireless, con la cual se comunicará con todas las estaciones WiFi.

Código 4.66: Puesta en marcha del escenario - Case05

```
1 sudo python scenario.py
```

Comprobación del funcionamiento

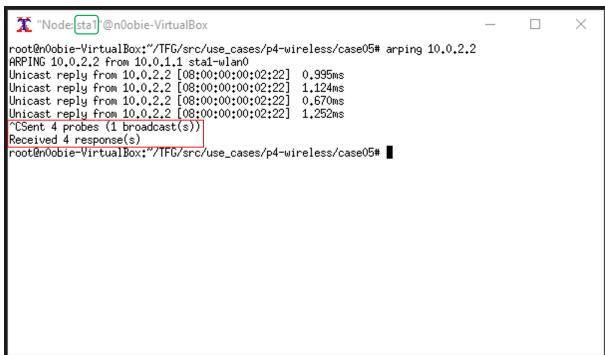
Tras las ejecución del script `scenario.py`, se tendrá el escenario 4.47 levantado, y la CLI de Mininet-WiFi abierta. Para la comprobación de funcionamiento de este caso de uso, se van a seguir los mismos pasos que en el case05 (4.2.5) - P4 en un entorno alámbrico. Por tanto no se entrará hacer explicaciones que se creen redundantes, se indicarán los pasos seguidos para llevar a cabo la comprobación de funcionamiento y los resultados de dichas pruebas.

Código 4.67: Pasos a seguir para comprobar el funcionamiento - Case05

```
1 mininet-wifi> xterm sta1 sta2 sta3
2
3 # Nos ponemos a escuchar en las estaciones wifi destino
4 [sta2] tcpdump -l
5 [sta3] tcpdump -l
6
7
8 # Generamos ARP-Request desde sta1
9 [sta1] arping 10.0.2.2
```

Como se puede apreciar en la figura 4.48 el ARP-Request llega correctamente. Se ve como los paquetes ARP-Request están llegando tanto al `sta2` como a la `sta3`. Pero solo será la `sta2`, en este caso, la que contesta ya que va dirigido a ésta.

Esto se debe a que al completarse la resolución ARP, la `sta1` ya conoce la dirección MAC de la `sta2`, por tanto los ARP-Request que se generen a posteriori llevarán la MAC destino la de la estación WiFi `sta2`, entonces el AP no lo difundirá por todos su puertos, se lo pasará directamente a la `sta2`.



```

root@n0bie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case05# arping 10.0.2.2
ARPING 10.0.2.2 from 10.0.1.1 sta1-wlan0
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 0.995ms
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 1.124ms
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 0.670ms
Unicast reply from 10.0.2.2 [08:00:00:00:02:22] 1.252ms
[Sent 4 probes (1 broadcast(s))
Received 4 response(s)
root@n0bie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case05#

```

(a) Ejecución de arping hacia la estación WiFi sta2



```

root@n0bie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case05# tcpdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on sta3-wlan0, link-type EN10MB (Ethernet), capture size 262144 bytes
14:15:19.132796 ARP, Request who-has 10.0.2.2 (Broadcast) tell 10.0.1.1, length 28
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
root@n0bie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case05#

```

(b) Escucha con Tcpdump en la estación WiFi sta3



```

root@n0bie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case05# tcpdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on sta2-wlan0, link-type EN10MB (Ethernet), capture size 262144 bytes
14:15:19.132786 ARP, Request who-has 10.0.2.2 (Broadcast) tell 10.0.1.1, length 28
14:15:19.132837 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
14:15:20.139388 ARP, Request who-has 10.0.2.2 (08:00:00:00:02:22 (oui Unknown)) tell 10.0.1.1, length 28
14:15:20.138543 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
14:15:21.138468 ARP, Request who-has 10.0.2.2 (08:00:00:00:02:22 (oui Unknown)) tell 10.0.1.1, length 28
14:15:21.138529 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
14:15:22.138603 ARP, Request who-has 10.0.2.2 (08:00:00:00:02:22 (oui Unknown)) tell 10.0.1.1, length 28
14:15:22.139070 ARP, Reply 10.0.2.2 is-at 08:00:00:00:02:22 (oui Unknown), length 28
^C
8 packets captured
8 packets received by filter
0 packets dropped by kernel
root@n0bie-VirtualBox:~/TFG/src/use_cases/p4-wireless/case05#

```

(c) Escucha con Tcpdump en la estación WiFi sta2

Figura 4.48: Comprobación de funcionamiento del Case05 - P4 Wireless

4.4. Casos de uso XDP en medios inalámbricos

En esta sección se introducirán todos los casos de uso realizados con la tecnología XDP en entornos inalámbricos. Todos los casos de uso se han nombrado siguiendo la misma sintaxis que en el repositorio del TFG, alojado en GitHub. Para la **instalación de las dependencias** de la tecnología XDP se ha dejado escrito el Anexo C.1, donde se detallan todos los pasos a seguir. De forma adicional, y como se comentó en el capítulo 3, se hará uso de Mininet-WiFi como plataforma para probar los casos de uso desarrollados en un entorno inalámbrico.

Los casos de uso se han dividido en cinco partes, con la finalidad de que la lectura de estos sea más clara y ordenada.

- **Introducción:** En esta parte se abordará las explicaciones teóricas complementarias en caso de que sean necesario, muchas de ellas se habrán dado ya en los casos de uso XDP en entornos cableados, por lo que se referenciará a dichas explicaciones.
- **Compilación:** En esta parte se explicará al lector cómo proceder para compilar el programa XDP.
- **Puesta en marcha del escenario:** En esta parte se explicará al lector cómo levantar el escenario y cómo poder limpiarlo tras finalizar la comprobación de funcionamiento.
- **Carga de los programas XDP:** en esta parte se indicará sobre qué interfaz se producirá la carga de los programas XDP, y se ha realizado dicha carga, ya que se hará de forma automática en el script que levanta el escenario.
- **Evaluación del funcionamiento:** Por último se hará una evaluación sobre el funcionamiento del caso de uso.

Para que el lector pueda seguir el desarrollo con los casos de uso, a continuación se indica la tabla 4.7, la cual expone en qué ruta del repositorio del TFG se puede encontrar dicho caso de uso, y un vídeo demostración donde el autor va comentando paso a paso el caso de uso y su evaluación.

Caso de uso	Enlace al repositorio	Enlace al vídeo demostración
case01 - Drop	Enlace al código	Enlace al vídeo
case02 - Pass	Enlace al código	Enlace al vídeo
case03 - Echo server	Enlace al código	Enlace al vídeo
case04 - Layer 3 forwarding	Enlace al código	Enlace al vídeo
case05 - Broadcast	Enlace al código	Enlace al vídeo

Tabla 4.7: Resumen de la documentación sobre los casos de uso XDP en entornos inalámbricos

4.4.1. Case01 - Drop

En este caso de uso se probará que es posible descartar todos los paquetes recibidos haciendo uso de la tecnología XDP en un entorno inalámbrico. Para la realizar la prueba, primero se debe compilar el programa XDP y segundo levantar el escenario donde se va a realizar la prueba de funcionamiento. En el proceso del levantamiento de la topología en Mininet-WiFi se anclará el binario a un interfaz del escenario, por lo que únicamente se tendrá que preocupar de observar los resultados cuando se genere tráfico que atraviese dicha interfaz.

Compilación

Para compilar el programa XDP se ha dejado un Makefile preparado en este directorio al igual que en los casos de uso XDP en entornos cableados. Por lo tanto, para compilarlo únicamente hay que seguir las indicaciones del bloque 4.68.

Código 4.68: Compilación programa XDP - Case01

```

1  # En caso de no haber entrado en el directorio asignado del caso de uso
2  cd TFG/src/use_cases/xdp-wireless/case01
3
4
5  # Hacemos uso del Makefile suministrado
6  sudo make

```

Si tiene dudas sobre el proceso de compilación del programa XDP le recomendamos que vuelva al case02 (XDP - Cableado 4.1.2) donde se hace referencia al *flow* dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para testear los programas XDP en un entorno inalámbrico, se hará uso de Mininet-WiFi para emular las topologías de red. Para levantar el escenario solo se tendrá que ejecutar el script en Python que hace uso de la API de Mininet-WiFi para generar toda la topología de red. Una vez ejecutado este abrirá la interfaz de linea de comandos de Mininet-WiFi, desde la cual se podrá comprobar el funcionamiento del caso de uso. En este caso, se realiza la carga del programa XDP desde el propio script de Python, haciendo uso de la herramienta xdp_loader desarrollada para ello. Por tanto, como se ha dicho este script está auto-contenido, por lo que solo se deberá ejecutarlo.

Para limpiar la máquina del escenario recreado anteriormente con Mininet-WiFi se podría realizar un `sudo mn -c`, pero se recomienda al usuario que haga uso del *target* del Makefile destinado para ello, ya que adicionalmente limpiará los ficheros intermedios generados en el proceso de compilación de nuestro programa XDP. Ejecutando el siguiente comando se limpiaría la máquina.

Código 4.69: Compilación programa XDP - Case01

```

1 # Levantamos el escenario
2 sudo python runenv.py
3
4
5 # Limpiamos el escenario
6 sudo make clean

```

Por último, únicamente indicar que el escenario recreado es el siguiente, compuesto exclusivamente de dos estaciones *wireless*, aisladas en sus propias *Network Namespaces*, y un punto de acceso corriendo el *daemon* de HostApd para intercomunicar dichas estaciones WiFi.

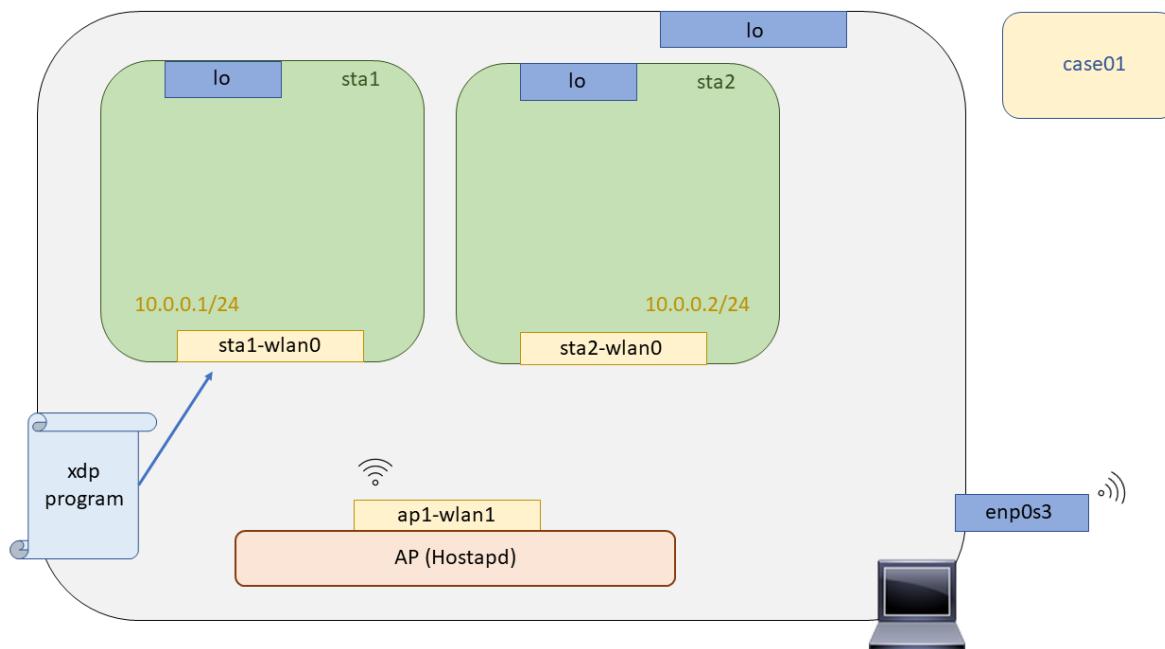


Figura 4.49: Escenario inalámbrico del Case01 - XDP

Carga del programa XDP

Como ya se ha comentado, la carga del programa XDP, se llevará a cabo en el proceso del levantamiento del escenario, descrito en el script de Python para crear la topología. La carga del programa XDP, se hará con el programa `xdp_loader`, utilizado anteriormente para cargar los programas XDP en interfaces alámbricas.

Código 4.70: Carga del programa XDP - Case01

```

1 # Línea 38 del script runenv.py
2 sta1.cmd("./xdp_loader -S -d sta1-wlan0 -F --progsec xdp_case01")

```

Es importante señalar, que el comando ejecutado dentro de la *Network Namespace* de la estación WiFi **sta1**, no se ha lanzado con permisos de root, ya que esta orden los heredará al lanzar el script que levanta la topología en Mininet-WiFi.

Comprobación del funcionamiento

Una vez que el programa XDP fue anclado a la interfaz de la estación WiFi **sta1**, es necesario comprobar si realmente funciona según lo esperado, y ver si las interfaces generadas por el módulo mac80211_hwsim son compatibles con XDP. Esto se hará generando tráfico desde una estación WiFi hacia la otra, para que atraviese por la interfaz que tiene anclado el programa XDP. En este caso el comportamiento esperado es que haga un *drop* de los paquetes nada más llegar a la interfaz, en este caso la interfaz **sta1-wlan0**.

Como en anteriores casos de uso, se ha habilitado la recolección de estadísticas sobre los códigos de retorno XDP, por lo que sería una buena práctica comprobar si se están empleando códigos de retorno **XDP_DROP**.

```
mininet-wifi> stal ping sta2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3049ms
pipe 4
mininet-wifi> stal ./xdp_loader -S -U -d stal-wlan0
INFO: xdp_link_detach() removed XDP prog ID:21 on ifindex:10
mininet-wifi> stal ping sta2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=3.98 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=2.98 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=6.46 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=4.43 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3007ms
rtt min/avg/max/mdev = 2.984/4.467/6.469/1.271 ms
mininet-wifi>
```

Figura 4.50: Comprobación de funcionamiento del Case01 - XDP Wireless

Como se puede apreciar en la figura 4.50, inicialmente se realiza un ping **■** entre las estaciones WiFi, pero no hay conectividad debido a que el programa XDP está tirando los paquetes. Acto seguido se quita el programa XDP de la interfaz y se vuelve a probar la conectividad entre las estaciones WiFi ejecutando un segundo ping **■**. Se ve que una vez desanclado el programa, la conectividad vuelve, por lo que se puede afirmar que el funcionamiento es el esperado.

4.4.2. Case02 - Pass

En este caso de uso se probará que es posible admitir todos los paquetes recibidos haciendo uso de la tecnología XDP en un entorno inalámbrico. ¿Qué significa “Admitir”? se quiere comprobar si es posible dejar pasar los paquetes, sin afectarles el plano de datos programado con la tecnología, ya que, aunque XDP mucha gente lo concibe para hacer un *by-pass* al *stack* de red del Kernel de Linux, una completa redefinición del mismo, en muchas ocasiones será útil trabajar en conjunto para conseguir la funcionalidad deseada. En este caso, además se verá si existe alguna diferencia inducida por el cambio de entorno, de alámbrico a *wireless*.

Compilación

Para compilar el programa XDP, al igual que en casos de uso anteriores, se ha dejado un Makefile preparado en este directorio. Por lo tanto, para compilarlo únicamente hay que seguir las indicaciones del bloque 4.71.

Código 4.71: Compilación programa XDP - Case02

```

1  # En caso de no haber entrado en el directorio asignado del caso de uso
2  cd TFG/src/use_cases/xdp-wireless/case02
3
4
5  # Hacemos uso del Makefile suministrado
6  sudo make

```

Si tiene dudas sobre el proceso de compilación del programa XDP le recomendamos que vuelva al case02 (XDP - Cableado 4.1.2) donde se hace referencia al *flow* dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para testear los programas XDP en un entorno inalámbrico, se hará uso de Mininet-WiFi para emular las topologías de red. Para levantar el escenario solo se tendrá que ejecutar el script en Python que hace uso de la API de Mininet-WiFi para generar toda la topología de red. Una vez ejecutado este abrirá la interfaz de linea de comandos de Mininet-WiFi, desde la cual se podrá comprobar el funcionamiento del caso de uso. En este caso, se realiza la carga del programa XDP desde el propio script de Python, haciendo uso de la herramienta *xdp_loader* desarrollada para ello. Por tanto, como se ha dicho este script está auto-contenido, por lo que solo se deberá ejecutarlo.

Para limpiar la máquina del escenario recreado anteriormente con Mininet-WiFi se podría realizar un *sudo mn -c*, pero se recomienda al usuario que haga uso del *target* del Makefile destinado para ello, ya que adicionalmente limpiará los ficheros intermedios generados en el proceso de compilación de nuestro programa XDP. Ejecutando el siguiente comando se

limpiaría la máquina.

Código 4.72: Compilación programa XDP - Case02

```

1  # Levantamos el escenario
2  sudo python runenv.py
3
4
5  # Limpiamos el escenario
6  sudo make clean

```

Por último, únicamente indicar que el escenario recreado es el siguiente, compuesto exclusivamente de dos estaciones *wireless*, aisladas en sus propias *Network Namespaces*, y un punto de acceso corriendo el *daemon* de HostApd para intercomunicar dichas estaciones WiFi.

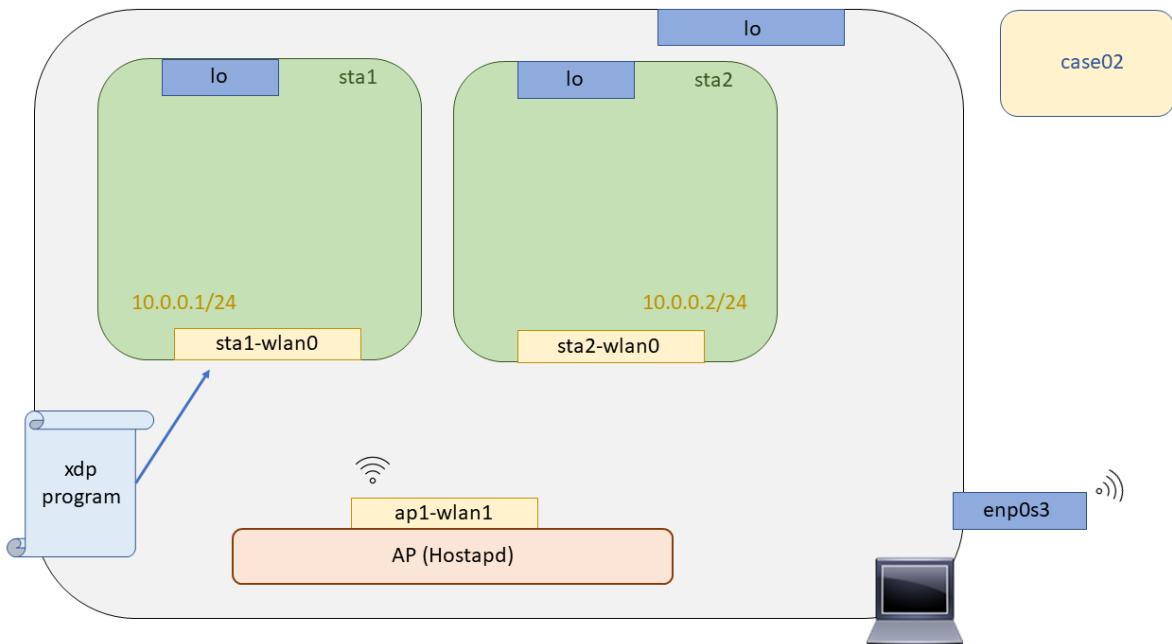


Figura 4.51: Escenario inalámbrico del Case02 - XDP

Carga del programa XDP

Como ya se ha comentado, la carga del programa XDP, se llevará a cabo en el proceso del levantamiento del escenario, descrito en el script de Python para crear la topología. La carga del programa XDP, se hará con el programa `xperf_loader`, utilizado anteriormente para cargar los programas XDP en interfaces alámbricas.

Código 4.73: Carga del programa XDP - Case02

```

1 # Línea 37 del script runenv.py
2 sta1.cmd("./xdp_loader -S -d sta1-wlan0 -F --progsec xdp_case02")

```

En estos primeros casos de uso XDP en un entorno inalámbrico, la carga de los programas se está realizando en el propio script que levanta la topología en Mininet-WiFi, pero más adelante en el case04 (Ir a subsección 4.4.4) se verá como se puede ejecutar comandos dentro de la *Network Namespace* indicada para lograr la carga de un programa XDP en una interfaz de un nodo independiente de la red desde la propia CLI de Mininet-WiFi.

Comprobación del funcionamiento

Una vez que el programa XDP fue anclado a la interfaz de la estación WiFi `sta1`, es necesario asegurarse de que funciona según lo esperado, y ver si el funcionamiento obtenido difiere al funcionamiento de este mismo caso de uso en un medio cableado. La prueba que se va a realizar puede que sea un poco simple, pero es válida, ya que únicamente queremos verificar que el programa anclado en la interfaz está delegando los paquetes que le llegan al *stack* de red.

Código 4.74: Comprobación del funcionamiento - Case02

```

1 # Hacemos ping desde una estación wifi hacia la otra. Deberíamos tener conectividad.
2 mininet-wifi> sta2 ping sta1
3
4 # Comprobamos los códigos de retorno XDP
5 mininet-wifi> sta1 ./xdp_stats -d sta1-wlan0

```

```

mininet-wifi> sta2 ping sta1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=2.93 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=6.86 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=3.72 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=2.99 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=8.45 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=3.02 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=1.99 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=3.61 ms
^C
--- 10.0.0.1 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7017ms
rtt min/avg/max/mdev = 1.999/4.201/8.457/2.095 ms
mininet-wifi>

```

Figura 4.52: Comprobación de funcionamiento (Ping) del Case02 - XDP Wireless

```

X "Node[sta]@n0obie-VirtualBox"
root@n0obie-VirtualBox:~/TFG/src/use_cases/xdp-wireless/case02# ./xdp_stats -d sta1-wlan0

Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:21 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0,261218
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0,261171
XDP_PASS         4 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0,258864
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0,258808
XDP_REDIRECT     0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:0,258784

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001847
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001893
XDP_PASS         7 pkts (    1 pps)      0 Kbytes (    0 Mbits/s) period:2,001900
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001921
XDP_REDIRECT     0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001942

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001062
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001016
XDP_PASS         9 pkts (    1 pps)      0 Kbytes (    0 Mbits/s) period:2,000993
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,000972
XDP_REDIRECT     0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,000950

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001481
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001494
XDP_PASS         11 pkts (    1 pps)      0 Kbytes (    0 Mbits/s) period:2,001518
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001527
XDP_REDIRECT     0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,001536

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,000641
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,000637
XDP_PASS         14 pkts (    1 pps)      1 Kbytes (    0 Mbits/s) period:2,000620
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,000616
XDP_REDIRECT     0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,000613

XDP-action
XDP_ABORTED      0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,009490
XDP_DROP         0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,009597
XDP_PASS         15 pkts (    0 pps)      1 Kbytes (    0 Mbits/s) period:2,009516
XDP_TX          0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,009628
XDP_REDIRECT     0 pkts (    0 pps)      0 Kbytes (    0 Mbits/s) period:2,009641

^C
root@n0obie-VirtualBox:~/TFG/src/use_cases/xdp-wireless/case02# 

```

Figura 4.53: Comprobación de funcionamiento (Stats) del Case02 - XDP Wireless

Según se puede ver en la figura 4.52, hay perfecta conectividad entre las estaciones WiFi dado que el ping está funcionando correctamente. Por lo que, se presupone que el programa XDP está dejando pasar los paquetes ICMP hacia el *stack* de red. Apreciando la figura 4.52, dicha presuposición se confirma, atendiendo al contador de los códigos de retorno XDP_PASS.

4.4.3. Case03 - Echo server

En este caso de uso se explorará el análisis de paquetes, su filtrado y manejo. En los anteriores caso de uso exclusivamente se definía un comportamiento de los paquetes haciendo un uso exclusivo de los códigos de retorno XDP, más concretamente XDP_DROP para tirar los paquetes y XDP_PASS para admitir los paquetes.

Por tanto, en este test se filtrarán los paquetes por sus cabeceras, haciendo uso de las estructuras indicadas en este mismo caso de uso en un entorno cableado (Ir a 4.1.3). Se filtrarán los paquetes ICMP con el código ECHO-Request, se modificarán sus campos para conformar su respuesta, y se reenviará dicho paquete a la misma interfaz por la cual llegó para que sea entregado a su emisor. De esta forma se conformará un servidor de Echo, que responderá a todos los pings que lleguen a la interfaz que tenga cargado este programa XDP.

Compilación

Para compilar el programa XDP, al igual que en casos de uso anteriores, se ha dejado un Makefile preparado en este directorio. Por lo tanto, para compilarlo únicamente hay que seguir las indicaciones del bloque 4.75.

Código 4.75: Compilación programa XDP - Case03

```
1 # En caso de no haber entrado en el directorio asignado del caso de uso
2 cd TFG/src/use_cases/xdp-wireless/case03
3
4
5 # Hacemos uso del Makefile suministrado
6 sudo make
```

Si tiene dudas sobre el proceso de compilación del programa XDP le recomendamos que vuelva al case02 (XDP - Cableado 4.1.2) donde se hace referencia al *flow* dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para testear los programas XDP en un entorno inalámbrico, se hará uso de Mininet-WiFi para emular las topologías de red. Para levantar el escenario solo se tendrá que ejecutar el script en Python que hace uso de la API de Mininet-WiFi para generar toda la topología de red. Una vez ejecutado este abrirá la interfaz de linea de comandos de Mininet-WiFi, desde la cual se podrá comprobar el funcionamiento del caso de uso. En este caso, se realiza la carga del programa XDP desde el propio script de Python, haciendo uso de la herramienta xdp_loader desarrollada para ello.

Para limpiar la máquina del escenario recreado anteriormente con Mininet-WiFi se podría realizar un `sudo mn -c`, pero se recomienda al usuario que haga uso del *target* del Makefile destinado para ello, ya que adicionalmente limpiará los ficheros intermedios generados en el proceso de compilación de nuestro programa XDP. Ejecutando el siguiente comando se limpiaría la máquina.

Código 4.76: Compilación programa XDP - Case03

```

1 # Levantamos el escenario
2 sudo python runenv.py
3
4
5 # Limpiamos el escenario
6 sudo make clean

```

Por último, únicamente indicar que el escenario recreado es el siguiente, compuesto exclusivamente de dos estaciones *wireless*, aisladas en sus propias *Network Namespaces*, y un punto de acceso corriendo el *daemon* de HostApd para intercomunicar dichas estaciones WiFi.

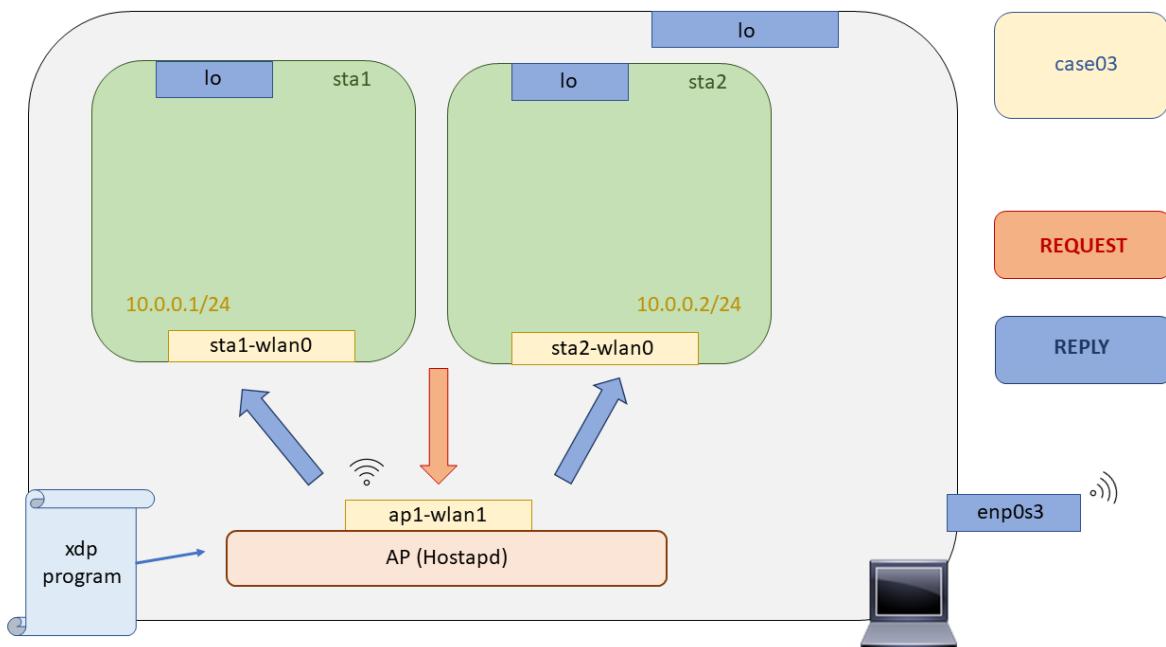


Figura 4.54: Escenario inalámbrico del Case03 - XDP

Carga del programa XDP

Como ya se ha comentado, la carga del programa XDP, se llevará a cabo en el proceso del levantamiento del escenario, descrito en el script de Python para crear la topología. La carga del programa XDP, se hará con el programa `xdp_loader`, utilizado anteriormente para cargar los programas XDP en interfaces alámbricas.

Código 4.77: Carga del programa XDP - Case03

```
1 # Línea 37 del script runenv.py
2 ap1.cmd("./xdp_loader -S -d ap1-wlan1 -F --progsec xdp_case03")
```

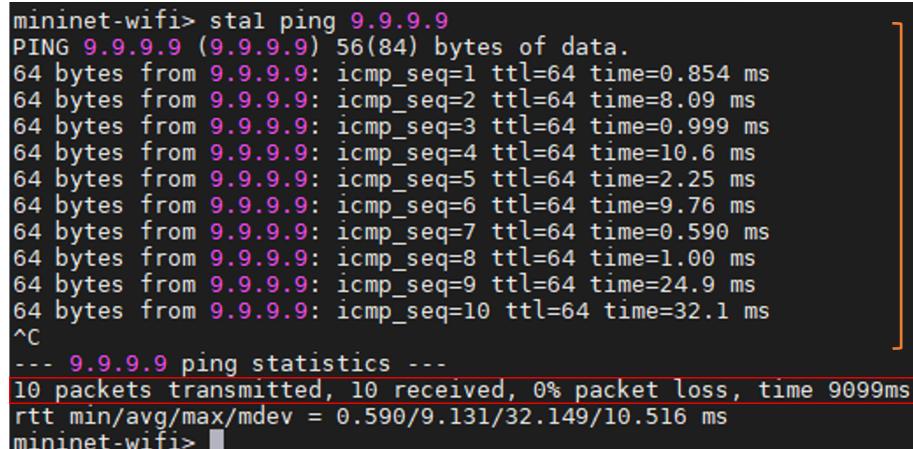
En este caso, se puede apreciar como la carga del programa XDP se está llevando a cabo sobre la interfaz del punto de acceso ap1. Dado que únicamente son los paquetes ICMP los que serán interceptados y contestados, los paquetes de control del punto de acceso no se verán afectados, irán directamente al *daemon* HostApd haciendo uso del código de retorno XDP_PASS.

Comprobación del funcionamiento

La comprobación del funcionamiento del programa XDP anclado a la interfaz ap1-wlan1 se llevará a cabo generando pings desde las estaciones wireless hacia el punto de acceso, para que la interfaz ap1-wlan1 los filtre, analice y nos genere una respuesta. Si todo funciona correctamente deberíamos ver como los códigos de retorno mayormente empleados son los de XDP_TX siempre y cuando no hayamos detenido el ping desde dentro de la *Network Namespace*.

Código 4.78: Comprobación del funcionamiento - Case03

```
1 # Lanzamos un ping desde una estación wireless hacia cualquier máquina
2 mininet-wifi> stal ping 9.9.9.9
3
4 # En una consola aparte lanzamos el programa xdp_stats para ir viendo a tiempo real
5 # los códigos de retorno XDP empleados
6 mininet-wifi> ap1 ./xdp_stats -d ap1-wlan1
```

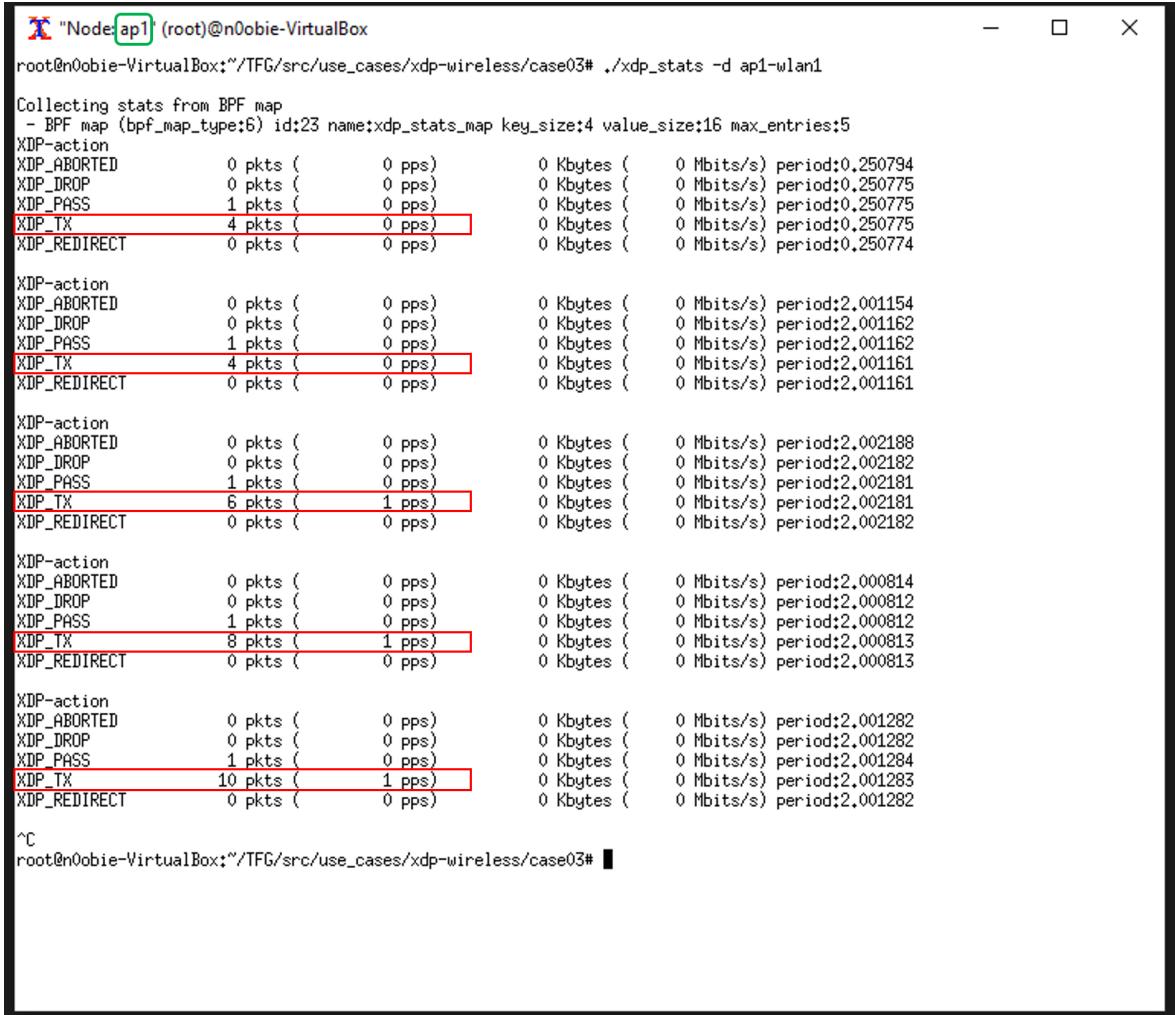


```
mininet-wifi> stal ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=64 time=0.854 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=64 time=0.09 ms
64 bytes from 9.9.9.9: icmp_seq=3 ttl=64 time=0.999 ms
64 bytes from 9.9.9.9: icmp_seq=4 ttl=64 time=10.6 ms
64 bytes from 9.9.9.9: icmp_seq=5 ttl=64 time=2.25 ms
64 bytes from 9.9.9.9: icmp_seq=6 ttl=64 time=9.76 ms
64 bytes from 9.9.9.9: icmp_seq=7 ttl=64 time=0.590 ms
64 bytes from 9.9.9.9: icmp_seq=8 ttl=64 time=1.00 ms
64 bytes from 9.9.9.9: icmp_seq=9 ttl=64 time=24.9 ms
64 bytes from 9.9.9.9: icmp_seq=10 ttl=64 time=32.1 ms
^C
--- 9.9.9.9 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9099ms
rtt min/avg/max/mdev = 0.590/9.131/32.149/10.516 ms
mininet-wifi>
```

Figura 4.55: Comprobación de funcionamiento (Ping) del Case03 - XDP Wireless

Como se puede ver en la figura 4.55, se está realizando ping hacia una máquina inexistente, pero como en el script de aprovisionamiento se le han indicado a todas las estaciones WiFi un hipotético *gateway*, y se ha añadido en la tabla ARP la MAC de ese hipotético *gateway*,

no habrá una resolución ARP previa al envío del ping. Por tanto se generará el ping  , que llegará a la interfaz la cual tiene cargado el programa XDP, y como se puede apreciar en la figura 4.56, dichos pings  son contestados correctamente por el programa devolviendo códigos de retorno XDP_TX.



```

X "Node[ap1] (root)@n0obie-VirtualBox"
root@n0obie-VirtualBox:~/TFG/src/use_cases/xdp-wireless/case03# ./xdp_stats -d ap1-wlan1

Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:23 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0,250794
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0,250775
XDP_PASS         1 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0,250775
XDP_TX           4 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0,250775
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0,250774

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001154
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001162
XDP_PASS         1 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001162
XDP_TX           4 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001161
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001161

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,002188
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,002182
XDP_PASS         1 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,002181
XDP_TX           6 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2,002181
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,002182

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,000814
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,000812
XDP_PASS         1 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,000812
XDP_TX           8 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2,000813
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,000813

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001282
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001282
XDP_PASS         1 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001284
XDP_TX           10 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2,001283
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2,001282

^C
root@n0obie-VirtualBox:~/TFG/src/use_cases/xdp-wireless/case03# 

```

Figura 4.56: Comprobación de funcionamiento (Stats) del Case03 - XDP Wireless

4.4.4. Case04 - Layer 3 forwarding

En este caso de uso se explorará el forwarding de paquetes, en este punto ya se debe conocer como filtrar por las cabeceras de los paquetes, analizarlos y establecer una lógica asociada a ese filtrado con los códigos de retorno XDP. La gran diferencia en este entorno inalámbrico es el número de interfaces que suele tener un dispositivo *wireless*. Generalmente en entornos cableados, un nodo suele tener tantas interfaces Ethernet como enlaces tenga con otros nodos, en cambio, en entornos wireless, con una misma interfaz puede tener enlaces con otros nodos siempre y cuando se encuentren en rango.

Por tanto, los programas XDP en entornos inalámbricos podrían realizar todo su forwarding haciendo uso de un único código de retorno XDP, `XDP_TX`. Con dicho código se reenviarían los paquetes recibidos a la interfaz a la cual llegaron los paquetes. Dado que en los entornos cableados, se tuvieron que hacer uso de los *helpers* BPF indicados en el bloque 4.12, se van a plantear escenarios similares donde sea necesario hacer un forwarding de una interfaz A a otra B. De esta forma, se verificará que dichas funciones siguen operativas en interfaces creadas con el módulo `mac80211_hwsim`.

No se ha querido plantear este caso de uso haciendo solo uso del código de retorno `XDP_TX`, ya que en el case03 XDP *Wireless* (Ir a 4.4.3), ya se hizo uso de este código y se vio que funcionaba correctamente, por lo que se entendía que era preferible explorar los *helpers* BPF en entornos inalámbricos.

Compilación

Para compilar el programa XDP, al igual que en casos de uso anteriores, se ha dejado un Makefile preparado en este directorio. Por lo tanto, para compilarlo únicamente hay que seguir las indicaciones del bloque 4.79.

Código 4.79: Compilación programa XDP - Case04

```

1  # En caso de no haber entrado en el directorio asignado del caso de uso
2  cd TFG/src/use_cases/xdp-wireless/case04
3
4
5  # Hacemos uso del Makefile suministrado
6  sudo make

```

Si tiene dudas sobre el proceso de compilación del programa XDP le recomendamos que vuelva al case02 (XDP - Cableado 4.1.2) donde se hace referencia al *flow* dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para testear los programas XDP en un entorno inalámbrico, se hará uso de Mininet-WiFi para emular las topologías de red. Para levantar el escenario solo se tendrá que ejecutar el script en Python que hace uso de la API de Mininet-WiFi para generar toda la topología de red. Una vez ejecutado este abrirá la interfaz de linea de comandos de Mininet-WiFi, desde la cual se podrá comprobar el funcionamiento del caso de uso.

Para limpiar la máquina del escenario recreado anteriormente con Mininet-WiFi se podría realizar un `sudo mn -c`, pero se recomienda al usuario que haga uso del *target* del Makefile destinado para ello, ya que adicionalmente limpiará los ficheros intermedios generados en el proceso de compilación de nuestro programa XDP. Ejecutando el siguiente comando se limpiaría la máquina.

Código 4.80: Compilación programa XDP - Case04

```

1 # Levantamos el escenario
2 sudo python runenv.py
3
4 # Limpiamos el escenario
5 sudo make clean

```

4.4.4.1. Hardcoded forwarding

La primera forma de implementación del forwarding es lo que denominaremos como Hard-coded forwarding. Como ya se comentó, es necesario hardcodear información de forwarding en el propio programa XDP. El escenario levantado (Ver figura 4.57), está compuesto de una estación wireless y un host, que corren dentro de su respectivas *Network Namespaces*, y un punto de acceso corriendo un proceso de HostApd para comunicar las dos estaciones wireless entre sí. De forma adicional, y por consistencia del caso de uso, se ha decidido correr dicho switch dentro de su propia *Network Namespace* para que no haya lugar a dudas de que el forwarding se está realizando correctamente y no está habiendo *by-passes* de ningún tipo. En este caso el forwarding lo haremos desde la interfaz `ap1-wlan1` hacia la interfaz `ap1-eth2`.

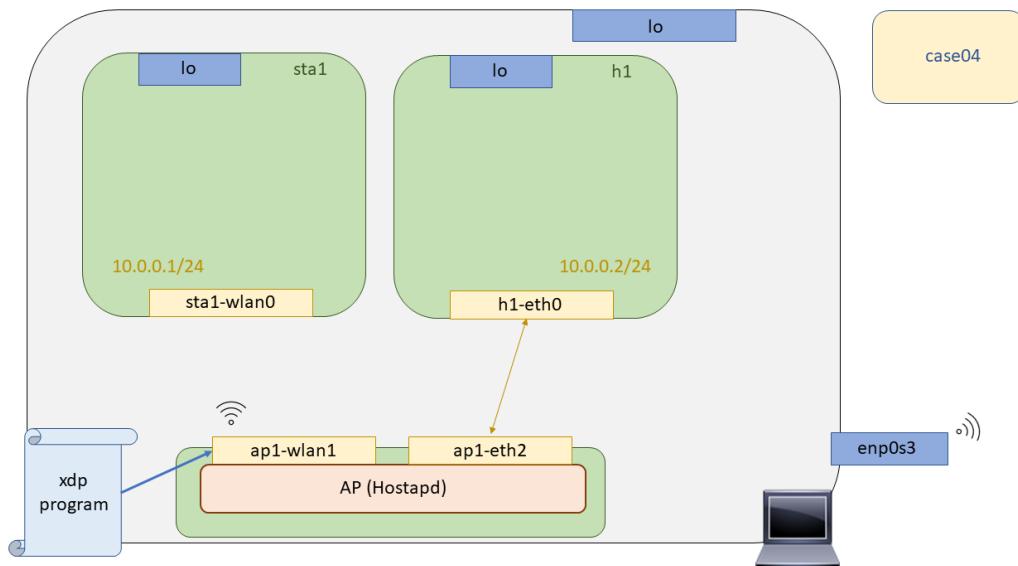


Figura 4.57: Escenario inalámbrico Hardcoded forwarding del Case04 - XDP

Carga del programa XDP

Antes de realizar la carga del programa se deben obtener dos datos, la `ifindex` de la interfaz `ap1-eth2` a la cual se va a mandar los paquetes generados desde la estación wireless `sta1`. Y la MAC de la interfaz del host1, ya que será necesario que los paquetes que se dirijan a la

interfaz `ap1-wlan1` lleven como MAC destino la de la `ap1-eth2` para que así los paquetes no sean descartados.

Una vez se tengan estos datos anotados se abrirá el programa XDP (`*.c`) cuan cualquier editor de texto, se irá a la declaración de variables y se hardcodeará tanto el `ifindex` como la MAC. Véase un ejemplo en el bloque 4.15.

Una vez que se tenga hardcodeado los datos para realizar el forwarding se deberá recompilar el programa XDP para que el `bytocode` que se ancle a la interfaz `ap1-wlan1` haga correctamente el forwarding. Por ello, simplemente se tiene que hacer un `make` nuevamente. Ahora sí, ya se tiene todo preparado para anclar de nuevo el programa XDP. En este caso la carga del programa XDP se llevará a cabo una vez levantado el escenario vía CLI de Mininet-WiFi como se puede apreciar en el bloque 4.81.

Código 4.81: Carga del programa XDP Hardcoded forwarding - Case04

```

1 # Antes que nada, debemos lanzar el escenario en caso de que no lo hayamos hecho todavía.
2 sudo python runenv.py
3
4
5 # Ejecutamos el siguiente comando dentro de la Network Namespace del AP1, cargando así el
6 # programa XDP en la interfaz ap1-wlan1.
7 mininet-wifi> ap1 ./xdp_loader -d ap1-wlan1 -F --progsec xdp_case04 -S

```

Comprobación del funcionamiento

La comprobación de funcionamiento de este programa XDP es bastante básica: se van a generar paquetes desde la estación `wireless sta1` hacia el host, `h1`. Para ello se abrirán dos terminales, en cada una de ellas se llevará a cabo una tarea.

Código 4.82: Comprobación del funcionamiento Hardcoded forwarding - Case04

```

1 # Abrimos las dos terminales
2 mininet-wifi> xterm h1 sta1
3
4 # En la terminal del host1 pondremos a un sniffer a escuchar los paquetes que nos lleguen.
5 [Terminal:h1] tcpdump -l
6
7 # En la terminal de la sta1 lanzaremos pings contra el h1
8 [Terminal:sta1] ping 10.0.0.2
9
10 # Por último, opcionalmente, podemos ejecutar el programa que actuaba como recolector de estadísticas sobre los códigos de retorno XDP
11 mininet-wifi> ap1 ./xdp_stats -d ap1-wlan1

```

Atendiendo a la figura 4.58, se puede ver como el ping  no funciona, y podría surgir la siguiente pregunta, ¿por qué no hay conectividad? No es que el programa no funcione, lo que sucede es que la comunicación actualmente solo está soportada en un sentido, ya que solo se está soportando el forwarding de la interfaz `ap1-wlan1` a la interfaz `ap1-eth2`. Actualmente el proceso de ping está detenido en la resolución ARP de la MAC asociada a la dirección IP `10.0.0.2`, llegando los ARP-Request pero los ARP-Reply nunca llegarán a su destino.

```

root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case04# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.2 ping statistics ---
6 packets transmitted, 0 received, +3 errors, 100% packet loss, time 5131ms
Pipe 4
root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case04#

```

Figura 4.58: Ping Hardcoded forwarding del Case04 - XDP Wireless

```

root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case04# tcpdump -l
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:22:52.620458 ARP, Request who-has n0obie-VirtualBox tell 10.0.0.1, length 28
23:22:52.620458 ARP, Reply n0obie-VirtualBox is-at 02:02:02:02:02:02 (oui Unknown), length 28
23:22:53.645932 ARP, Request who-has n0obie-VirtualBox tell 10.0.0.1, length 28
23:22:53.646019 ARP, Reply n0obie-VirtualBox is-at 02:02:02:02:02:02 (oui Unknown), length 28
23:22:54.649227 IP 0.0.0.66 > 255.255.255.255.bootps: BOOTP/DHCP, Request from 10:00:00:00:00:01 (oui Unknown), length 254
23:22:54.671377 ARP, Request who-has n0obie-VirtualBox tell 10.0.0.1, length 28
23:22:54.671459 ARP, Reply n0obie-VirtualBox is-at 02:02:02:02:02:02 (oui Unknown), length 28
23:22:55.633762 ARP, Request who-has n0obie-VirtualBox tell 10.0.0.1, length 28
23:22:55.633960 ARP, Reply n0obie-VirtualBox is-at 02:02:02:02:02:02 (oui Unknown), length 28
23:22:56.637181 IP 0.0.0.66 > 255.255.255.255.bootps: BOOTP/DHCP, Request from 10:00:00:00:00:01 (oui Unknown), length 254
23:22:56.716045 ARP, Request who-has n0obie-VirtualBox tell 10.0.0.1, length 28
23:22:56.716135 ARP, Reply n0obie-VirtualBox is-at 02:02:02:02:02:02 (oui Unknown), length 28
^C
12 packets captured
12 packets received by filter
0 packets dropped by kernel
root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case04#

```

Figura 4.59: Sniffer Hardcoded forwarding del Case04 - XDP Wireless

Se puede, de forma adicional, añadir la entrada ARP de forma estática a la *Network Namespace* de la estación *wireless sta1* pero el ping seguiría sin funcionar, ya que los paquetes ECHO-Reply asociados a los ECHO-Request recibidos nunca podrán llegar de vuelta a la estación *wireless sta1*. En el siguiente escenario de forwarding se abordará esta carencia en la comunicación con un nuevo planteamiento para conseguir bidireccionalidad en la comunicación, a la par que escalabilidad.

4.4.4.2. Semi-Hardcoded forwarding (BPF maps)

La segunda forma de implementar el forwarding se denominará Semi-Hardcoded forwarding, como ya se comentó, la información irá hardcodeada, pero no en el propio programa XDP, sino, en los mapas BPF. El escenario levantado (Ver figura 4.60), está compuesto de una estación wireless y un host, que corren dentro de su respectivas *Network Namespaces*, y un punto de acceso corriendo un proceso de HostApd para comunicar las dos estaciones wireless entre sí. De forma adicional, y por consistencia del caso de uso, se ha decidido ejecutar dicho switch dentro de su propia *Network Namespace* para que no haya lugar a dudas de que el forwarding se está realizando correctamente y no está habiendo *by-passes* de ningún tipo. En este caso el forwarding se hará desde la interfaz `ap1-wlan1` hacia la interfaz `ap1-eth2` y viceversa.

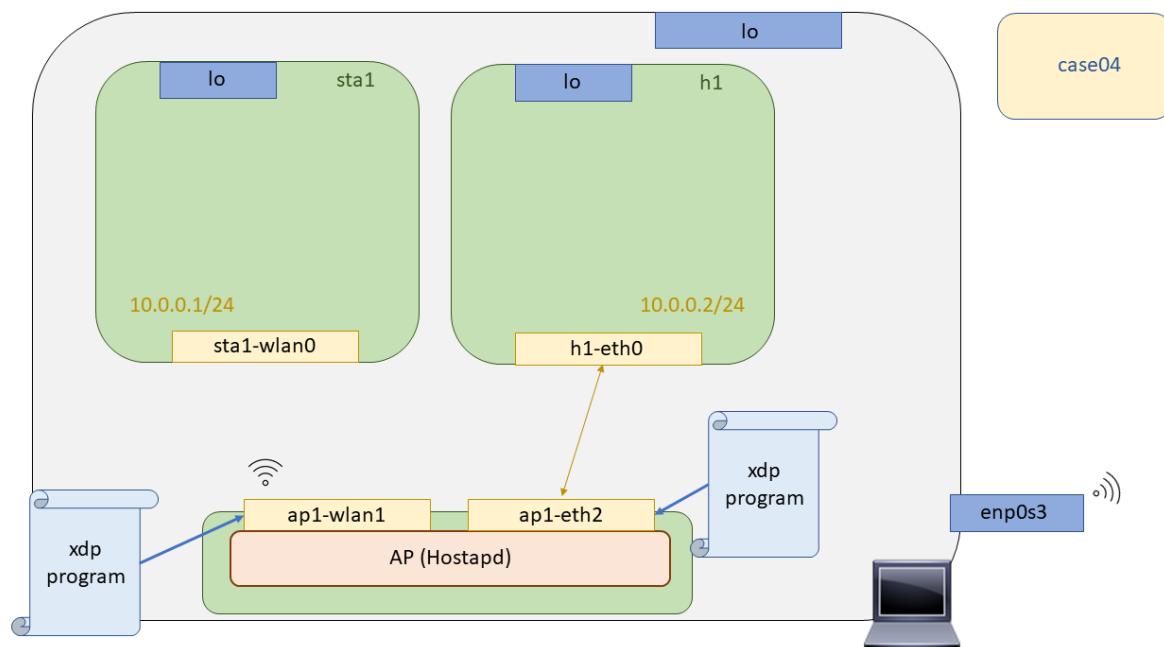


Figura 4.60: Escenario inalámbrico Semi-Hardcoded forwarding del Case04 - XDP Wireless

Carga del programa XDP

Esta manera de hacer el forwarding no requiere de hardcodear datos en el propio programa XDP. En cambio, se usarán los mapas BPF como medio para guardar información de forwarding (ifindex y MACs destino) desde espacio de usuario, para que posteriormente el programa anclado en el Kernel sea capaz de leer los mapas BPF, obtener la información de forwarding y realizarlo en base a la información percibida de los mapas BPF. Es importante señalar que los programas anclados previamente deben ser eliminados, por lo que una opción sería hacer un *clean* del escenario haciendo uso del Makefile dado (`sudo make clean`) y empezar de nuevo. Por tanto, para anclar los programas XDP se deberán seguir las indicaciones del

bloque 4.81.

Código 4.83: Carga del programa XDP Semi-Hardcoded forwarding - Case04

```

1 # Lanzamos el script que levanta el escenario
2 sudo python runenv.py
3
4 # Anclamos el programa XDP en cada interfaz para conseguir un comunicación bidireccional
5 mininet-wifi> ap1 ./xdp_loader -d ap1-wlan1 -F --progsec xdp_case04_map -S
6 mininet-wifi> ap1 ./xdp_loader -d ap1-eth2 -F --progsec xdp_case04_map -S
7
8 # Almacenamos la información necesaria para realizar el forwarding y populamos los mapas BPF
9 # con la información útil para llevar a cabo el forwarding en ambas direcciones.
10 mininet-wifi> ap1 ./redirect.sh

```

Comprobación del funcionamiento

La comprobación de funcionamiento de este programa puede ser llevada a cabo desde un extremo u otro debido a que, si todo funciona correctamente, existirá una comunicación bidireccional. Por lo que en este caso se harán las pruebas desde la estación wireless **sta1** hacia el host **h1**.

Código 4.84: Comprobación del funcionamiento Semi-Hardcoded forwarding - Case04

```

1 # Hacemos un ping desde el h1 a la sta1, o al revés
2 mininet-wifi> sta1 ping h1
3
4 # Comprobamos con el recolector de estadísticas que se están produciendo XDP_REDIRECT
5 mininet-wifi> ap1 sudo ./xdp_stats -d ap1-wlan1

```

The terminal window shows a ping session between sta1 and h1. The output is as follows:

```

X "Node[sta1]"@n0obie-VirtualBox
root@n0obie-VirtualBox:"/TFG/src/use_cases/xdp-wireless/case04# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.51 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=4.62 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.09 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=6.04 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=3.19 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=10.9 ms
^C
--- 10.0.0.2 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5010ms
rtt min/avg/max/mdev = 1.098/4.575/10.978/3.330 ms
root@n0obie-VirtualBox:"/TFG/src/use_cases/xdp-wireless/case04#

```

Figura 4.61: Ping Semi-Hardcoded forwarding del Case04 - XDP Wireless

Como se puede apreciar en la figura 4.61, hay perfecta conectividad, ya que el ping se ejecuta sin problemas entre la estación WiFi y el host, además los códigos de retorno XDP_REDIRECT van aumentando (Ver figura 4.62) siendo esto un indicador de que el programa efectivamente está llevando a cabo el forwarding.

```

mininet-wifi> a1 sudo ./xdp_stats -d a1-wlan1
Collecting stats from BPF map
- BPF map (bpf_map_type:6) id:35 name:xdp_stats_map key_size:4 value_size:16 max_entries:5
XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.250994
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.250995
XDP_PASS         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.250994
XDP_TX          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.250994
XDP_REDIRECT     0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:0.250994

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.004091
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.004092
XDP_PASS         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.004091
XDP_TX          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.004091
XDP_REDIRECT     2 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.004092

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006941
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006962
XDP_PASS         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006976
XDP_TX          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.006987
XDP_REDIRECT     4 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.006997

XDP-action
XDP_ABORTED      0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.002993
XDP_DROP         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.002974
XDP_PASS         0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.002961
XDP_TX          0 pkts (      0 pps)      0 Kbytes (      0 Mbits/s) period:2.002950
XDP_REDIRECT     6 pkts (      1 pps)      0 Kbytes (      0 Mbits/s) period:2.002938

```

Figura 4.62: Stats Semi-Hardcoded forwarding del Case04 - XDP Wireless

4.4.4.3. Forwarding auto (Kernel FIBs)

En esta última alternativa, no se pudo replicar debido a que existía una incompatibilidad del escenario propuesto con ese tipo de forwarding. Esto debe a que el forwarding automático se alimentaba de las rutas de la FIB del Kernel, pero al estar aislado el punto de acceso en su propia *Network Namespace*, no era capaz de obtener dicha información.

4.4.5. Case05 - Broadcast

Por último, en este caso de uso exploraremos la capacidad de broadcast de XDP en entornos inalámbricos. Por ello se ha intentado replicar un escenario básico de broadcast en un escenario inalámbrico. Se ha planteado hacer uso de la herramienta arping para emular una resolución ARP, generando ARP-Request, estos llevan su MAC destino todo a FF:FF:FF:FF:FF:FF y su dominio de difusión englobaría todos aquellos nodos de la red que operen hasta capa 2.

El escenario propuesto para este caso de uso se puede apreciar en la figura 4.63. Este estaría compuesto de dos estaciones wireless conectadas a un punto de acceso, sobre el cual irá toda la lógica XDP, y por último, un host desde el que podremos empezar las resoluciones ARP para ver así su difusión en un medio wireless. En este caso, sí es viable hacer Broadcast debido a la condición de un medio inalámbrico, ya que al tener una única interfaz *wireless* para interconectar todos los nodos inalámbricos, solo sería necesario enviar el paquete con MAC destino todo a FF:FF:FF:FF:FF:FF para lograr el Broadcast.

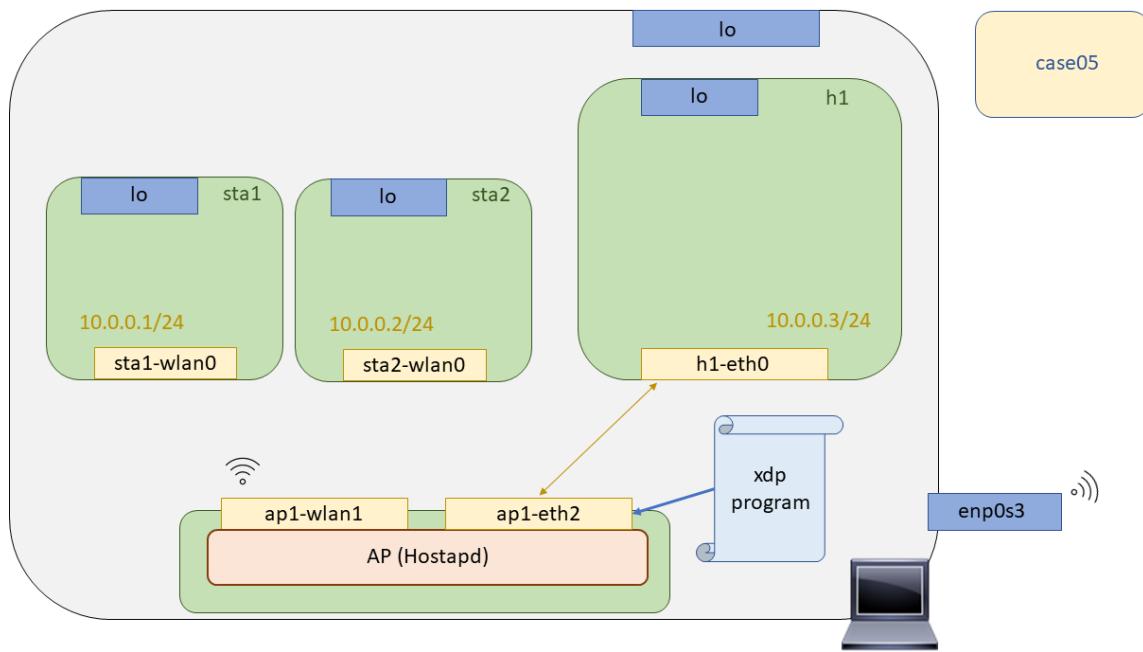


Figura 4.63: Escenario inalámbrico del Case05 - XDP

Compilación

Para compilar el programa XDP, al igual que en casos de uso anteriores, se ha dejado un Makefile preparado en este directorio. Por lo tanto, para compilarlo únicamente hay que seguir las indicaciones del bloque 4.85.

Código 4.85: Compilación programa XDP - Case05

```

1 # En caso de no haber entrado en el directorio asignado del caso de uso
2 cd TFG/src/use_cases/xdp-wireless/case05
3
4
5 # Hacemos uso del Makefile suministrado
6 sudo make

```

Si tiene dudas sobre el proceso de compilación del programa XDP le recomendamos que vuelva al case02 (XDP - Cableado 4.1.2) donde se hace referencia al *flow* dispuesto para la compilación de los programas XDP.

Puesta en marcha del escenario

Para testear los programas XDP en un entorno inalámbrico, se hará uso de Mininet-WiFi para emular las topologías de red. Para levantar el escenario solo se tendrá que ejecutar el script en Python que hace uso de la API de Mininet-WiFi para generar toda la topología de

red. Una vez ejecutado este abrirá la interfaz de linea de comandos de Mininet-WiFi, desde la cual se podrá comprobar el funcionamiento del caso de uso. Para limpiar la máquina del escenario recreado anteriormente con Mininet-WiFi se podría realizar un `sudo mn -c`, pero se recomienda al usuario que haga uso del *target* del Makefile destinado para ello, ya que adicionalmente limpiará los ficheros intermedios generados en el proceso de compilación de nuestro programa XDP. Ejecutando el siguiente comando se limpiaría la máquina.

Código 4.86: Compilación programa XDP - Case05

```

1 # Levantamos el escenario
2 sudo python runenv.py
3
4
5 # Limpiamos el escenario
6 sudo make clean

```

Carga del programa XDP

Una vez compilado tanto el programa XDP, y nuestro escenario lanzado con la ejecución del script `runenv.py`, vamos a proceder con la carga del programa XDP haciendo uso de la herramienta `xdp_loader`.

Código 4.87: Carga del programa XDP - Case05

```

1 # Antes que nada, debemos lanzar el escenario en caso de que no lo hayamos hecho todavía.
2 sudo python runenv.py
3
4
5 # Ejecutamos el siguiente comando dentro de la Network Namespace del AP1, cargando así el
6 # programa XDP en la interfaz ap1-eth2.
7 mininet-wifi> ap1 ./xdp_loader -d ap1-eth2 -F --progsec xdp_case05 -S

```

Comprobación del funcionamiento

Para comprobar el funcionamiento del sistema de Broadcast se realizará la siguiente prueba, desde el host `h1` se generarán paquetes ARP-Request preguntando por alguna de las MACs de las estaciones *wireless*. Si el sistema de Broadcast funciona, escuchando en las estaciones *wireless* destino `sta1` y `sta2` se debería ver como los paquetes ARP-Request llegan sin problemas.

Código 4.88: Comprobación del funcionamiento - Case05

```

1 # Generamos el ARP-REQUEST
2 mininet-wifi> h1 arping 10.0.0.1-2
3
4 # Escuchamos en las estaciones wireless a la espera de ver ARP-REQUEST.
5 mininet-wifi> staX tcpdump -l arp

```

```
mininet-wifi> h1 arping 10.0.0.1
ARPING 10.0.0.1 from 10.0.0.3 h1.eth0
^CSent 5 probes (5 broadcast(s))
Received 0 response(s)
mininet-wifi>
```

(a) Ejecución de arping hacia la estación WiFi sta1

```
root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case05# tcpdump -l arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
00:06:38.387194 ARP, Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.3, length 28
00:06:38.387611 ARP, Reply n0obie-VirtualBox is-at 08:01:01:01:11 (oui Unknown), length 28
00:06:39.387922 ARP, Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.3, length 28
00:06:39.388493 ARP, Reply n0obie-VirtualBox is-at 08:01:01:01:11 (oui Unknown), length 28
00:06:40.388228 ARP, Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.3, length 28
00:06:40.388888 ARP, Reply n0obie-VirtualBox is-at 08:01:01:01:11 (oui Unknown), length 28
00:06:41.388934 ARP, Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.3, length 28
00:06:41.389499 ARP, Reply n0obie-VirtualBox is-at 08:01:01:01:11 (oui Unknown), length 28
00:06:42.389855 ARP, Request who-has n0obie-VirtualBox (Broadcast) tell 10.0.0.3, length 28
00:06:42.390336 ARP, Reply n0obie-VirtualBox is-at 08:01:01:01:11 (oui Unknown), length 28
^C
10 packets captured
10 packets received by filter
0 packets dropped by kernel
root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case05#
```

(b) Escucha con Tcpdump en la estación WiFi sta1

```
root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case05# tcpdump -l arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
00:06:38.387317 ARP, Request who-has 10.0.0.1 (Broadcast) tell 10.0.0.3, length 28
00:06:39.387976 ARP, Request who-has 10.0.0.1 (Broadcast) tell 10.0.0.3, length 28
00:06:40.388283 ARP, Request who-has 10.0.0.1 (Broadcast) tell 10.0.0.3, length 28
00:06:41.388994 ARP, Request who-has 10.0.0.1 (Broadcast) tell 10.0.0.3, length 28
00:06:42.390017 ARP, Request who-has 10.0.0.1 (Broadcast) tell 10.0.0.3, length 28
^C
5 packets captured
5 packets received by filter
0 packets dropped by kernel
root@n0obie-VirtualBox:/TFG/src/use_cases/xdp-wireless/case05#
```

(c) Escucha con Tcpdump en la estación WiFi sta2

Figura 4.64: Comprobación de funcionamiento del Case05 - XDP Wireless

Como se puede apreciar en la figura 4.64, ambas estaciones WiFi reciben el ARP-Request, siendo solo **sta1** quien conteste. La resolución ARP no se puede completar ya que no se ha implementado un sistema de forwarding en la *Network Namespace* del **ap1**. Al igual que en el case05 (XDP Cableado 4.1.5), se propone hacer uso del los programas XDP desarrollados en el caso de uso anterior para implementar un forwarding, y que así, las resoluciones se completen.

5. Conclusiones y trabajo futuro

En este capítulo, se finalizará la memoria con las conclusiones del trabajo realizado, haciendo una valoración del mismo y de los resultados obtenidos del desarrollo. Además, se presentarán las vías de trabajo futuro que han ido apareciendo a medida que se realizaba el TFG. En concreto, se expondrán dichas vías, y se concluirá con un breve análisis sobre la viabilidad de las mismas.

5.1. Conclusiones del TFG

La finalidad de los casos de uso era la de ver con qué tecnología (XDP, P4) es más sencillo y eficiente definir el *datapath* de los dispositivos IoT en entornos SDN. Por tanto, antes de realizar las conclusiones, se deberá revisar los resultados de las evaluaciones de ambas tecnologías en los distintos entornos, cableado e inalámbrico.

Como se puede apreciar en la tabla 5.1, con la tecnología XDP en medios cableados no se pudo realizar un Broadcast de forma nativa, es decir, dicha tecnología no fue suficiente para lograr el broadcast. Es verdad que dicha limitación fue estudiada, y se planteó una solución, haciendo uso de un programa BPF adicional en el TC, pero aún teniendo en cuenta que se solucionó, no se puede afirmar que la tecnología lo soporte. En cambio, si se observa dicha funcionalidad en un entorno Wireless, ya no habría dicha limitación dado que no sería necesario clonar los paquetes. Generalmente cambiando la dirección destino del paquete de capa dos a difusión, y transmitiéndolo al medio, será suficiente para lograr el Broadcast.

En cambio, la tecnología P4, a diferencia de XDP, tiene una interfaz de alto nivel para definir funcionalidades de broadcast/multicast. Esta interfaz, comúnmente conocida como “grupos multicast”, permite definir en una estructura de datos de tipo JSON por qué puertos inundar y con qué cantidad de réplicas por puerto. De forma adicional, cabe mencionar que los grupos multicast permiten modificar el tipo de instancia de los paquetes clonados, siendo posible diferenciar los paquetes originales de los clonados.

Caso de uso	XDP	P4	P4-Wireless	XDP-Wireless
case01 - Drop	✓	✓	✓	✓
case02 - Pass	✓	✗	✗	✓
case03 - Echo server	✓	✓	✓	✓
case04 - Layer 3 forwarding	✓	✓	✓	✓
case05 - Broadcast	✗	✓	✓	✓

Tabla 5.1: Resumen sobre los casos de uso desarrollados

Volviendo a la tabla 5.1, se quiere hacer especial hincapié en que la tecnología P4 no soporta la funcionalidad de dejar pasar o delegar los paquetes, sin afectarles el plano de datos programado en P4. Por otro lado, con XDP siempre se puede pasar el paquete al Kernel para que se encargue él del procesamiento. Sin embargo, en P4, se debe definir de forma exclusiva todo el *datapath*, por lo que no hay a quien delegar el paquete, es decir, se tiene que encargar el propio programa P4. Este punto es muy positivo para XDP, ya que acciones sencillas y repetitivas pueden ser definidas en un programa XDP y delegar el resto de funcionalidades al Kernel. Por tanto, se actuaría de forma cooperativa y ganando en rendimiento como se vio en el case04 (4.1.4), donde se trabajaba de forma conjunta para obtener información de routing desde el propio Kernel.

Por último, se quiere mencionar la diferencia que existe entre ambas tecnologías en la interfaz de control. En P4 la interfaz de control serán las tablas, las cuales se definirán su estructura desde el propio programa P4 pero se irán completando vía P4Runtime por un controlador externo. Por ejemplo, ONOS¹, el cual soporta la configuración vía P4Runtime. En cambio, en XDP se podría decir que el equivalente a las tablas serían los mapas BPF.

Estos mapas, de tipo clave-valor, son definidos por el programa que se ancla en el Kernel, y generalmente son completados por programas de espacio de usuario, que acceden a éste a través de descriptores de archivo. Se podría decir que es un sistema al que aún le queda recorrido para ser igual de consistente que las tablas en P4, dado que sería necesario el desarrollo de un servidor XDP de espacio de usuario que estuviera a la escucha de información de control. Este servidor se encargaría de escribir en los mapas BPF la información equivalente de control que permitiese que el programa anclado en el Kernel actuara acorde a las directrices de un hipotético controlador.

Por tanto, se concluye indicando que la tecnología XDP actualmente está mejor enfocada para escenarios de integración totales (Ver figura 1.3), donde el dispositivo IoT únicamente dispone de interfaces wireless. Este dispositivo se beneficiará del rendimiento y la optimización de recursos que ofrece XDP al actuar de forma reactiva a los paquetes.

En cuanto a la tecnología P4 se cree que está más enfocada a escenarios de integración parcial (Ver figura 1.2), donde el dispositivo IoT mediador dispone tanto de interfaces cableadas para comunicarse con el core SDN, como de una interfaz inalámbrica para comunicarse con otros dispositivos IoT. Dichos dispositivos se beneficiarán de las facilidades que otorga P4 para realizar multicast, y de la interfaz de control tan bien definida que tiene para ser controlado por un controlador SDN convencional.

¹<https://wiki.onosproject.org/display/ONOS/P4+brigade>

5.2. Líneas de trabajo futuro

Como se ha podido apreciar a lo largo del TFG, la tecnología XDP aun se está asentando; se está abriendo camino dando nuevas posibilidades a los desarrolladores de la parte de *Networking* del Kernel, reescribiendo herramientas y distintos procesos del *stack* de red del Kernel de Linux². Por otro lado, se encuentra la tecnología P4, la cual ya lleva un par de años en escena, pero cuya comunidad va aumentando día tras día, con ello, la calidad y robustez de sus implementaciones.

Desgraciadamente, aunque estas dos tecnologías estén en constante crecimiento, no parece que a día de hoy haya un interés especial por llevarlas al IoT. Debido a lo cual, a lo largo de este TFG no se ha encontrado la información suficiente, ni las plataformas adecuadas para realizar el estudio y análisis de la integración de los dispositivos IoT en entornos SDN, llegando incluso a tener que realizar implementaciones personalizadas de plataformas para la emulación de redes wireless con finalidad de poder evaluar los casos de uso.

El hecho que de que aún no haya un interés general por llevar a dichas tecnologías al mundo IoT, deja mucho por hacer, explorar, desarrollar y probar. A continuación, se indican las vías de trabajo futuro que se cree que abrirán el camino a futuros trabajos e implementaciones, que son principalmente dos: la manipulación directa de las cabeceras `ieee80211` y la evaluación en redes de baja capacidad (`ieee802154`).

5.2.1. Integración de interfaces en modo monitor con los casos de uso

Según lo explicado en el análisis de casos de uso inalámbricos, donde se hablaba sobre el módulo `mac80211_hwsim`, al trabajar con dicho módulos terminamos haciendo uso de interfaces Ethernet que comunican con un radio `ieee80211`. Por tanto todos los paquetes que llegan a las interfaces llevan cabeceras Ethernet. Esto es debido a que en el Kernel se produce una traducción de cabeceras del estándar `ieee80211` al estándar `ieee8023`, y acto seguido se delegan los paquetes a la interfaz en cuestión. Para más información sobre este proceso de traducción se recomienda acudir al apartado 4.3.1.2 donde se explica con mayor detalle este proceso, a qué se debe, qué aporta y qué limitaciones induce al trabajar de esta manera.

Debido a esta condición de diseño del módulo `mac80211_hwsim`, no se ha podido gestionar de forma directa los paquetes con las cabeceras WiFi. Las interfaces creadas por este módulo, soportan varios modos de funcionamiento, y en el único modo de funcionamiento en el cual se puede llegar a ver las cabeceras WiFi es en el modo monitor, el cual está pensado para hacer una escucha pasiva del medio inalámbrico. Por lo tanto, aquí se abre una nueva vía de trabajo futuro, en la cual se quiere hacer uso de interfaces en modo monitor para la escucha de paquetes WiFi y para su transmisión.

Pero el modo monitor está pensado únicamente para escuchar paquetes, no para transmitir. Este modo puede ser llevado al límite haciendo una inyección de paquetes (*packet injection*) por la interfaz. Esto significa que la construcción de las cabeceras WiFi debe ser realizada

²<https://github.com/xdp-project/net-next>

por un módulo o herramienta externa al Kernel, abrir un socket *raw* con dicha interfaz y transmitir el paquete. Puesto que este proceso implica llevar al límite la funcionalidad de un modo de la interfaz, se ha querido realizar una prueba de concepto básica donde se pueda ver si es viable realizar inyecciones de paquetes. Por ello, se ha desarrollado, como se puede ver en el bloque 5.1, una herramienta en Python que genera un ping sobre el estándar IEEE80211. Para conseguir el ping sobre cabeceras WiFi se hizo uso de Scapy³ para el conformado de las cabeceras necesarias para transmitir el mensaje, y del módulo socket para generar un socket *raw* con la interfaz en modo monitor.

Código 5.1: Herramienta wping

```

1  from scapy.all import *
2  import socket, subprocess
3
4  class wPing():
5      """ Ping wireless """
6
7      def __init__(self, intf='mon0', mac_receiver='ff:ff:ff:ff:ff:ff', mac_dst='ff:ff:ff:ff:ff:ff', ip_dst=None, ↵
8          ↵ ping=None):
9          """ intf: interface name (default mon0)
10             mac_receiver: next hop MAC
11             mac_dst: destination MAC"""
12
13     self.sock = None
14     self.Intf = intf
15     self.src_mac = None
16     self.dst_mac = mac_dst
17     self.rcv_mac = mac_receiver
18     self.dst_ip = ip_dst
19     self.ping = ping
20     self.config()
21     if ping is None:
22         self.build()
23
24     def config(self):
25         """ Get socket to the given intf and configure params """
26         try:
27             self.sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
28             self.sock.bind((self.Intf, 0))
29         except:
30             print('Oops! Unable to get a socket raw with the interface - ' + self.Intf)
31
32     self.src_mac = subprocess.check_output('cat /sys/class/net/' + self.Intf + '/address', stderr=←
33                                         ← subprocess.STDOUT, shell=True).decode('utf-8').split("\n")[0]
34
35     def build(self):
36         """ Build ping with scapy classes"""
37         self.ping = RadioTap() / Dot11(FCfield=0x01, addr1=self.rcv_mac, addr2=self.src_mac, addr3=←
38             ← =self.dst_mac, addr4=self.src_mac) / LLC() / SNAP() / \
39             IP(dst=self.dst_ip) / ICMP(id=0x178b)
40
41     def run(self):
42         self.sock.send(bytes(self.ping))
43
44     if __name__ == "__main__":
45         wireless_ping = wPing('mon0', '02:00:00:00:02:00', '02:00:00:00:01:00', '10.0.0.2')
46         wireless_ping.run()

```

³<https://scapy.readthedocs.io/en/latest/api/scapy.layers.dot11.html>

5.2.1.1. Inyección de paquetes en interfaz en modo monitor

Para evaluar si realmente las interfaces creadas por el módulo mac80211_hwsim en modo monitor soportan la inyección de paquetes, se va hacer uso de Mininet-WiFi, el cual opera sobre el dicho modulo para la creación de las interfaces inalámbricas. Se utilizará la topología por defecto que tiene Mininet-WiFi, compuesta de dos estaciones WiFi interconectadas entre sí a través de un punto de acceso con capacidad SDN. Lo primero que se va a realizar es el levantamiento del escenario como se puede ver en el bloque 5.2

Código 5.2: Ejecución del escenario

```

1 sudo mn --wifi
2 *** Creating network
3 *** Adding controller
4 *** Adding stations:
5 sta1 sta2
6 *** Adding access points:
7 ap1
8 *** Configuring wifi nodes...
9 *** Adding link(s):
10 (sta1, ap1) (sta2, ap1)
11 *** Configuring nodes
12 *** Starting controller(s)
13 c0
14 *** Starting L2 nodes
15 ap1 ...
16 *** Starting CLI:
17 mininet-wifi> dump
18 <Controller c0: 127.0.0.1:6653 pid=502>
19 <Station sta1: sta1-wlan0:10.0.0.1,sta1-wlan0:None,sta1-wlan0:None pid=509>
20 <Station sta2: sta2-wlan0:10.0.0.2,sta2-wlan0:None,sta2-wlan0:None pid=511>
21 <OVSPAP ap1: lo:127.0.0.1,ap1-wlan1:None pid=516>
22 mininet-wifi>
```

Una vez levantado el escenario, se tendrá la CLI de Mininet-WiFi abierta, por lo que se seguirá con la creación de una interfaz en modo monitor en una de las estaciones WiFi de la topología. Como se puede apreciar en el bloque 5.3, primero se obtiene el nombre del radio sobre el cual se va a crear la interfaz en modo monitor, y acto seguido se añade la interfaz mon0 en modo monitor.

Código 5.3: Creación de interfaz en modo monitor

```

1 mininet-wifi> sta1 iw phy | head -n1 | cut -d' ' -f2
2 mn0s00
3 mininet-wifi> sta1 iw phy mn0s00 interface add mon0 type monitor
```

Cuando la interfaz en modo monitor esté levantada y correctamente enganchada a su radio, esto se puede verificar siguiendo los pasos que se indican en la figura 5.1. Una vez verificado, se procederá a hacer uso de la herramienta 5.1 para generar pings desde una estación WiFi, y a su vez, se pondrá a escuchar con un *sniffer* en la estación WiFi destino para ver si dichos paquetes están siendo recibidos.

Acto seguido, según se comentaba, se va a proceder abriendo Wireshark⁴ como *sniffer* en la estación WiFi (*sta2*), y se va a generar un único ping desde la estación WiFi (*sta1*). Como

⁴<https://www.wireshark.org/>

```
mininet-wifi> st1 iw phy mn0s00 interface add mon0 type monitor
mininet-wifi> st1 iw dev
phy#9
    Interface mon0
        ifindex 2
        wdev 0x90000002
        addr 02:00:00:00:00:00
        type monitor
        txpower 14.00 dBm
    Interface st1-wlan0
        ifindex 21
        wdev 0x90000001
        addr 02:00:00:00:00:00
        ssid my-ssid
        type managed
        channel 1 (2412 MHz), width: 20 MHz (no HT), center1: 2412 MHz
        txpower 14.00 dBm
mininet-wifi>
```

Figura 5.1: Verificación de la creación de la interfaz en modo monitor.

se puede apreciar en la figura 5.2, llega correctamente el ping al destino, pero como estamos escuchando en una interfaz en modo managed de tipo Ethernet, el Kernel ya ha realizado la traducción de las cabeceras.

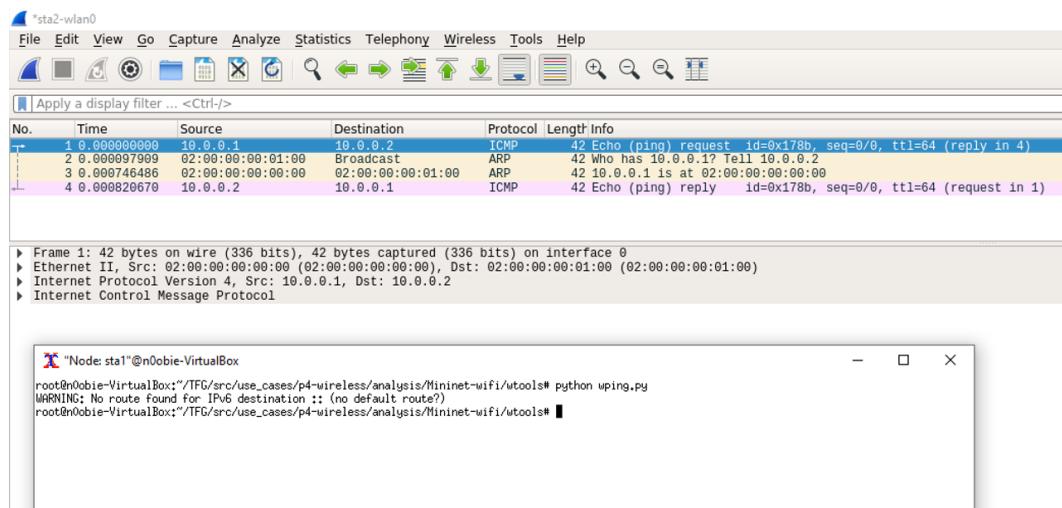


Figura 5.2: Generación de ping vía interfaz en modo monitor.

Si se quiere ver con qué cabeceras sale realmente el paquete, se puede realizar una escucha en la interfaz en modo monitor en la cual se está realizando la inyección de paquetes. De esta forma, los paquetes al ser injectados no se verán afectados por ningún tipo de traducción realizada en el Kernel. En la figura 5.3, se puede ver como realmente el paquete sale con las cabeceras WiFi indicadas. Además, si nos fijamos en sus direcciones MAC, podemos ver cómo están establecidas, indicando siguiente salto, transmisor, destino final y el origen del ping.

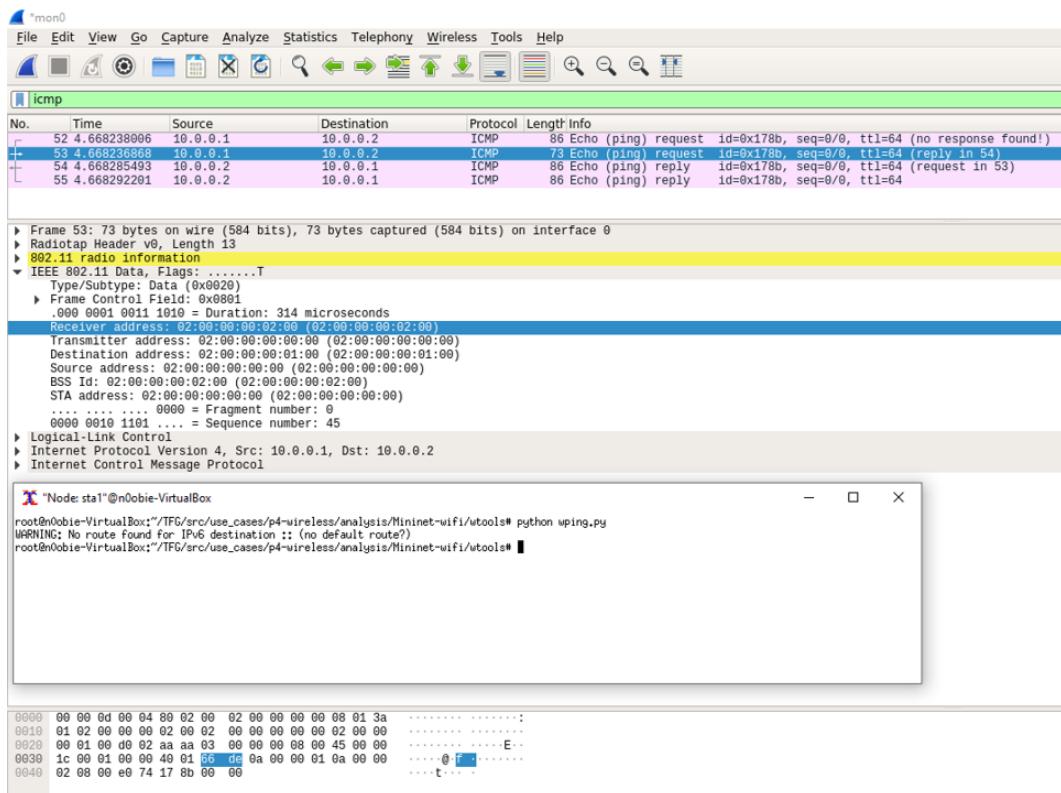


Figura 5.3: Comprobación de las cabeceras del ping generado vía interfaz en modo monitor.

Por lo tanto, se concluye diciendo que sí es viable la transmisión de paquetes a través de interfaces en modo monitor. Como se puede apreciar en la figura 5.3, hay paquetes presuntamente duplicados, pero si nos fijamos en sus cabeceras se puede ver como dichos paquetes son los paquetes del salto intermedio entre las estaciones WiFi (*sta1 - ap1 - sta2*).

Esto ocurre debido a que la interfaz monitor hace una escucha pasiva del medio, escuchando todos los paquetes que estén en el rango de la interfaz. Por esto, se propone aplicar un filtrado por MAC destino tanto en XDP como en P4 para recibir únicamente los paquetes que van dirigidos a dicha interfaz. De esta modo, se consigue abordar la problemática de transmisión y recepción de paquetes con interfaces creadas con el módulo `mac80211_hwsim` en modo monitor, habilitando así la gestión directa de las cabeceras `ieee80211`.

Por consiguiente, se cree que esta vía de trabajo futuro puede ayudar a trabajar más cerca de la tecnología wireless al poder gestionar directamente sus cabeceras y paquetes de control (*beacons*).

5.2.2. Emulación de redes de baja capacidad - mac802154_hwsim

En este punto, se expone una nueva vía de trabajo futuro, en la cual se propone exportar los casos de uso realizados a un entorno más enfocado al IoT. Por ello, se analizará si es viable exportar los casos de uso realizados anteriormente en medios cableados e inalámbricos bajo el estándar wireless de baja capacidad llamado `ieee802154`.

De esta forma se quiere explorar las fortalezas y debilidades, tanto de XDP como del lenguaje P4, en la definición del *datapath* de dispositivos de baja capacidad en un escenario IoT. Dichos dispositivos, al tener una condición de baja capacidad (estando bastante limitados en batería, alcance, memoria y procesamiento), necesitarán que la tecnología que les defina su *datapath* sea lo más eficiente posible, haciendo un uso optimizado de los recursos de la moto IoT. Este escenario tan limitado esclarecerá qué tecnología es más idónea a la hora de trabajar en IoT.

5.2.2.1. Herramientas para interactuar con el stack `ieee802154`

Se ha empezado analizando qué herramientas de espacio de usuario se tienen disponibles actualmente para trabajar con el subsistema `ieee802154`. Se encontraron numerosas herramientas y recursos después de ver qué dependencias instala Mininet-WiFi para interactuar con su módulo de IoT.

Todas las herramientas que han sido útiles en este análisis se muestran en la tabla 5.2. Además, se ha incluido un enlace a la página oficial de su código fuente, donde se indica cómo instalarlas y cómo hacer uso de ellas.

Herramienta	Explicación	Enlace
<code>iwpan</code>	Herramienta principal para la configuración del subsistema <code>ieee802154</code> , sus interfaces y sus radios. Esta herramienta es muy similar a la herramienta <code>iw</code> pensada para controlar el subsistema Wireless del Kernel de Linux.	Enlace al código
<code>wpan-hwsim</code>	Herramienta para gestionar el modulo <code>mac802154_hwsim</code> , permitiendo añadir o eliminar radios, así como añadir enlaces entre distintos radios a través de netlink.	Enlace al código
<code>wpan-ping</code>	Herramienta para hacer ping directamente con el stack <code>ieee802154</code> .	Enlace al código

Tabla 5.2: Resumen de las herramientas del entorno `ieee802154`

5.2.2.2. Arquitectura del stack `ieee802154`

A continuación, se expone un diagrama de la arquitectura del stack `ieee802154`⁵, obtenido del repositorio oficial del desarrollo de las herramientas y de los módulos asociados al estándar `ieee802154` del Kernel de Linux. Este esquema ha sido de gran utilidad para comparar sus distintos bloques con el módulo `mac80211_hwsim`⁶, el cual se utilizó para emular redes inalámbricas bajo el estándar `ieee80211`.

⁵<https://github.com/linux-wpan/wpan-misc>

⁶https://wireless.wiki.kernel.org/en/users/drivers/mac80211_hwsim

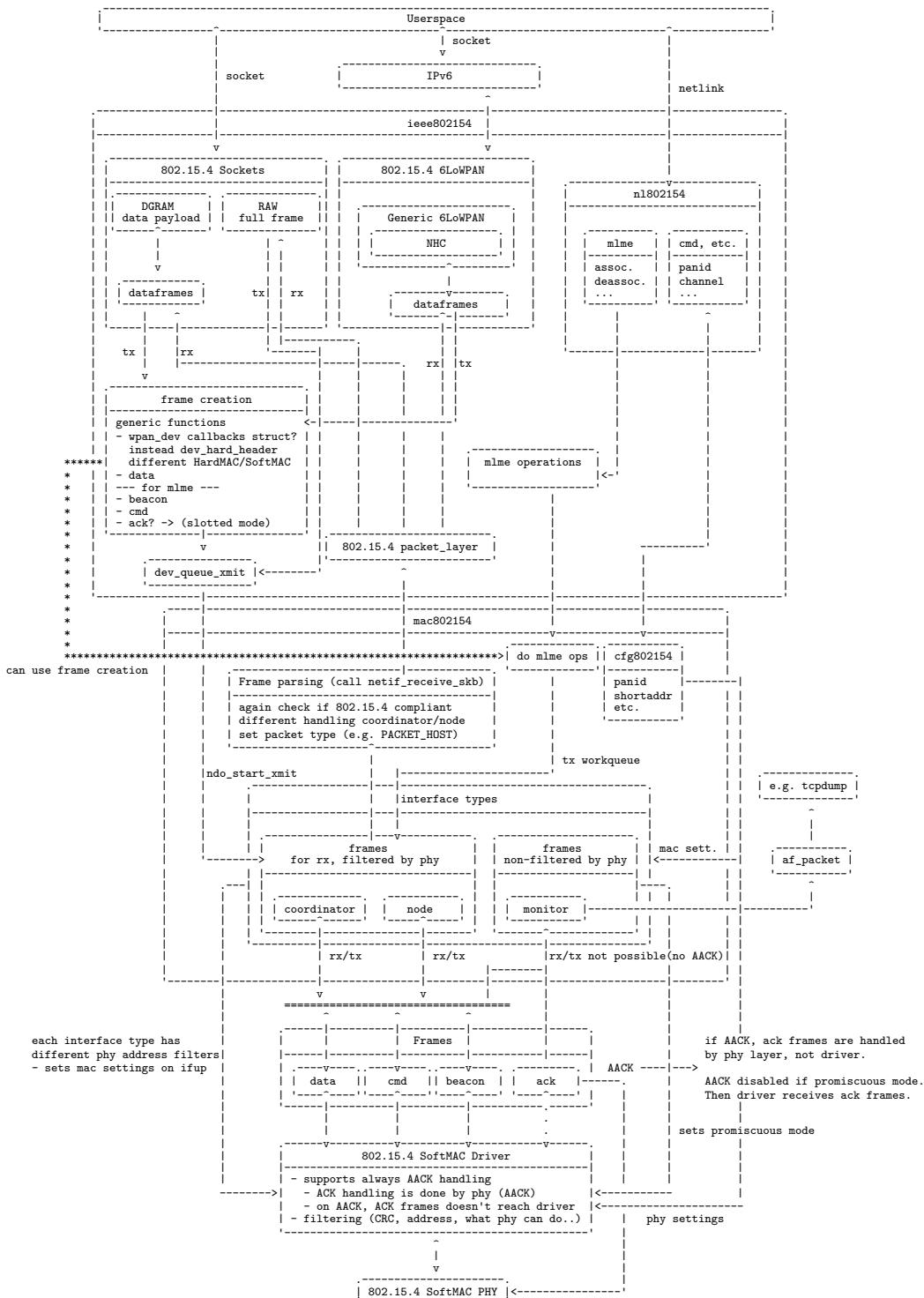


Figura 5.4: Arquitectura del stack ieee802154 [?]

Tras analizar la arquitectura es posible ver que este módulo, a diferencia del módulo mac80211_hwsim, no conecta a la capa física (*phy*) interfaces del tipo Ethernet. Es importante recordar que cuando se trabaja con WiFi se estudió por qué se hacía un parseo de paquetes con cabeceras 802.11 hacia cabeceras 802.3 y viceversa en el Kernel.

Al generalizar hacia interfaces Ethernet, las aplicaciones que trabajan a nivel de interfaz son muy extrapolables, ya que pueden ser portadas a distintos tipos de dispositivos con distintas características. Es verdad que se pierde en rendimiento, ya que el paquete se encola hasta tres veces en el Kernel y requiere procesamiento adicional, pero por otro lado se gana muchísima versatilidad.

5.2.2.3. Viabilidad de los casos de uso

Teniendo en cuenta que no se emplean interfaces de Ethernet para comunicar la capa física con las capas superiores, los casos de uso no serían extrapolables de forma inmediata, ya que en las etapas de parseo de todos los casos de uso, se parseaba a una cabecera del tipo Ethernet, tanto en XDP haciendo accesos a memoria, como en P4 con las distintas etapas de *parsing*.

Por tanto, en vez de parsear los paquetes hacia estructuras de Ethernet, habría que hacerlo hacia estructuras que definieran la estructura de la cabecera ieee802154. Se ha buscando en el [Kernel](#), donde se ha encontrado que dicha estructura ya está definida, por lo que solo habría que hacer uso de ella en los casos de uso de XDP, y en los casos de uso de P4 habría que declarar la estructura análoga y sustituir el parser de Ethernet por el parser de cabeceras 802.15.4.

Código 5.4: Estructura para manejar paquetes de l estándar ieee802154

```

1 struct ieee802154_hdr {
2     struct ieee802154_hdr_fc fc;
3     u8 seq;
4     struct ieee802154_addr source;
5     struct ieee802154_addr dest;
6     struct ieee802154_sechdr sec;
7 };

```

No obstante, aunque la vía de trabajo futuro ya se encuentra bien definida, se va a realizar una prueba de concepto probando a añadir un programa XDP a una interfaz generada por el módulo mac802154_hwsim. De este modo, se quiere consolidar esta vía de trabajo futuro probando que si admite soporte para XDP.

En cuanto a el BMv2, se entiende que no debe suponerle un problema hacer uso de las interfaces suministradas, ya que este hace uso de la librería [libpcap](#) para gestionar el envío y recepción de los paquetes. Esta librería se ha visto que hace uso de la interfaz de sockets en última instancia, se puede ver [aquí](#) en el código fuente, por lo que no supondría ninguna incompatibilidad con la arquitectura de subsistema ieee802154 que admite la operabilidad con distintas familias de sockets.

5.2.2.4. Carga de programas XDP en interfaces ieee802154

Como ya se ha comentado, la viabilidad de replicar los casos de uso sobre el módulo mac802154_hwsim pasa por si el stack ieee802154 es capaz de gestionar la carga de programas XDP en sus interfaces. Por otro lado, como ya se había mencionado con el BMv2 no habría ningún problema ya que iría sobre la interfaz de sockets del Kernel, por lo que la arquitectura del propio módulo lo contempla.

Se ha buscado información al respecto sobre si el modulo era compatible con la carga de programas BPF, que al final es en lo que se traduce un programa XDP, pero no se ha encontrado información precisa al respecto, por lo que se ha decidido hacer una prueba de concepto de carga de programas XDP. Como se comentó previamente, los programas XDP no son extrapolables directamente al subsistema ieee802154, por esto se hará uso del **case01** donde únicamente se implementaba una funcionalidad del *datapath* haciendo uso de los códigos de retorno XDP.

De esta forma no será necesario parsear las cabeceras ieee802154, y se podrá ver de una forma rápida si dichas interfaces asociadas a los radios con la capa física ieee802154 Soft-MAC soportan la carga de un programa XDP genérico. Esta prueba en primera instancia, se hizo directamente cargando el módulo y creando las interfaces pero, ya que Mininet-WiFi da soporte⁷ para gestionar dicho módulo a través de la implementación de Mininet-IoT, se pensó que daría cierta continuidad el desarrollo de los distintos casos de uso en dicha herramienta de emulación.

A partir de este punto solo puede continuar si se posee una versión del Kernel v4.18.x o superior, ya es a partir de esta donde se incluye el módulo mac802154_hwsim. Lo primero que se hará es volver a la instalación de Mininet-WiFi donde se llevarán a cabo los casos de uso wireless, e instalar las dependencias necesarias para la gestión del módulo mac802154_hwsim.

Código 5.5: Instalación de las dependencias de Mininet-WiFi - mac802154_hwsim

```

1  # Entramos en el directorio de mininet-wifi
2  cd mininet-wifi
3
4
5  # Añadimos las dependencias para la gestión del módulo mac802154_hwsim
6  sudo util/install.sh -6

```

Ahora con las dependencias ya añadidas, vamos a se va hacer uso del ejemplo que utiliza el módulo mac802154_hwsim disponible en Mininet-WiFi, para probar la carga del programa XDP desarrollado en el **case01**. Dicho ejemplo puede ser encontrado bajo el directorio **examples** con el nombre de **6LoWPan.py**. Cabe mencionar que en este ejemplo se añade una interfaz auxiliar que añade la capa de gestión del estándar 6LoWPan⁸. En este caso, esta interfaz auxiliar es irrelevante, ya que la carga del programa se hará sobre la interfaz nativa de subsistema ieee802154.

⁷ https://github.com/intrig-unicamp/mininet-wifi/tree/master/mn_wifi/sixLoWPAN

⁸ <https://tools.ietf.org/html/rfc8138>

Código 5.6: Ejecución ejemplo 6LoWPan.py

```

1 # Entramos al directorio de los ejemplos
2 cd examples
3
4 # Ejecutamos el ejemplo
5 sudo python 6LoWPan.py

```

Una vez cargado el ejemplo, se tendrá la CLI de Mininet-WiFi abierta, y lo siguiente que se hará será compilar el caso de uso XDP. Como ya comentamos anteriormente, será el [case01](#).

Código 5.7: Compilación del programa XDP - mac802154_hwsim

```

1 # Nos movemos de directorio y compilamos el case01
2 mininet-wifi> sh cd ~/TFG/src/use_cases/xdp/case01 && sudo make

```

Ya se tendría compilado el caso de uso XDP y listo para ser cargado sobre la interfaz. Pero, se debe plantear la siguiente cuestión ¿Sobre qué interfaz se debe cargar el programa XDP? Bien, para contestar esta pregunta se debe inspeccionar antes las interfaces de uno de los sensores que hay en la topología del ejemplo.

Como los sensores corren en su propia *Network namespace*, se tendrá tendremos que abrir una xterm o indicar el nombre de nodo previamente en la CLI de Mininet-WiFi para que dicho comando se ejecute dentro de la *Network Namespace* indicada.

Código 5.8: Obtención de interfaz 802.15.4

```

1 mininet-wifi> sensor1 ip a s
2 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
3     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4     inet 127.0.0.1/8 scope host lo
5         valid_lft forever preferred_lft forever
6     inet6 ::1/128 scope host
7         valid_lft forever preferred_lft forever
8 2: sensor1-pan0@sensor1-wpan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc noqueue ←
9     ↪ state UNKNOWN group default qlen 1000
10    link/6lowpan 9e:7b:c7:68:bb:9c:31:56 brd ff:ff:ff:ff:ff:ff:ff:ff
11    inet6 2001::1/64 scope global
12        valid_lft forever preferred_lft forever
13 248: sensor1-wpan0: <BROADCAST,NOARP,UP,LOWER_UP> mtu 123 qdisc fq_codel state UNKNOWN group←
14     ↪ default qlen 300
15     link/ieee802.15.4 9e:7b:c7:68:bb:9c:31:56 brd ff:ff:ff:ff:ff:ff:ff:ff

```

Como se puede apreciar en el bloque 5.8 la interfaz que añade la capa 6LoWPan es la interfaz **sensor1-pan0**, por lo tanto únicamente se hará uso de la interfaz que comunica de forma directa con el subsistema **ieee802154**. De forma adicional, se puede utilizar la herramienta **iwpan**, mencionada en la sub-sección de herramientas del stack **ieee802154**, para ver qué interfaces (*dev*) están disponibles y sobre qué radio están conectadas según se puede ver en el bloque 5.9.

Código 5.9: Obtención de interfaz 802.15.4 haciendo uso de la herramienta iwpan

```

1 mininet-wifi> sensor1 iwpan dev
2 phy#24
3 Interface sensor1-wpan0

```

```

4           ifindex 248
5           wpan_dev 0x1800000002
6           extended_addr 0x9e7bc768bb9c3156
7           short_addr 0xfffff
8           pan_id 0xbeef
9           type node
10          max_frame_retries 3
11          min_be 3
12          max_be 5
13          max_csmam_backoffs 4
14          lbt 0
15          ackreq_default 0

```

Por tanto, una vez que se sabe sobre qué interfaz operar (en este caso se trabajará sobre el sensor1 en la interfaz `sensor1-wpan0`), solo queda cargar el programa y ver si realmente soporta la funcionalidad del mismo.

Código 5.10: Carga de programa XDP en interfaz ieee802154

```

1  mininet-wifi> sensor1 cd /home/n0obie/TFG/src/use_cases/xdp/case01
2  mininet-wifi> sudo ./xdp_loader -S -d sensor1-wpan0 -F --progsec xdp_case01
3      Success: Loaded BPF-object(prog_kern.o) and used section(xdp_case01)
4      - \gls{xdp} prog attached on device:sensor1-wpan0(ifindex:248)
5      - Pinning maps in /sys/fs/bpf/sensor1-wpan0/
6  mininet-wifi>

```

Se puede ver como el programa ha sido correctamente cargado en la interfaz. Además, los mapas se anclaron de manera satisfactoria en sistema de archivos BPF por lo que se habilita la recolección de estadísticas relativas a los códigos de retorno XDP.

Para la comprobación de la funcionalidad del programa XDP, se llevará a cabo un ping entre los dos sensores, ya que al implementar un código de retorno `XDP_DROP` no habrá conectividad entre ellos.

Código 5.11: Comprobación funcionamiento programa XDP en interfaz ieee802154

```

1  # Ojo! Con el programa XDP cargado en la interfaz
2  mininet-wifi> sensor1 ping sensor2
3  PING 2001::2(2001::2) 56 data bytes
4  From 2001::1 icmp_seq=1 Destination unreachable: Address unreachable
5  From 2001::1 icmp_seq=2 Destination unreachable: Address unreachable
6  From 2001::1 icmp_seq=3 Destination unreachable: Address unreachable
7  ^CsendInt: writing chr(3)
8
9  --- 2001::2 ping statistics ---
10 4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3072ms
11
12 mininet-wifi>

```

Como se puede apreciar el comportamiento es el esperado, pero aun así, cabría pensar que la falla de conectividad entre ambos sensores es debida a otra razón, por ello se va eliminar el programa XDP de la interfaz y se va a probar de nuevo la conectividad entre ambos sensores.

Código 5.12: Comprobación de funcionamiento del programa XDP - 2

```

1 # Quitamos el programa XDP de la interfaz
2 mininet-wifi> sensor1 cd /home/n0obie/TFG/src/use_cases/xdp/case01
3 mininet-wifi> sudo ./xdp_loader -S -U -d sensor1-wpan0
4 INFO: xdp_link_detach() removed XDP prog ID:677 on ifindex:248
5 mininet-wifi>
6
7
8 # Probamos de nuevo la conectividad entre ambos sensores
9 mininet-wifi> sensor1 ping sensor2
10 PING 2001::2(2001::2) 56 data bytes
11 64 bytes from 2001::2: icmp_seq=1 ttl=64 time=0.053 ms
12 64 bytes from 2001::2: icmp_seq=2 ttl=64 time=0.044 ms
13 64 bytes from 2001::2: icmp_seq=3 ttl=64 time=0.044 ms
14 64 bytes from 2001::2: icmp_seq=4 ttl=64 time=0.106 ms
15 ^CsendInt: writing chr(3)
16
17 --- 2001::2 ping statistics ---
18 4 packets transmitted, 4 received, 0% packet loss, time 3049ms
19 rtt min/avg/max/mdev = 0.044/0.061/0.106/0.027 ms
20 mininet-wifi>
```

Teniendo en cuenta todos los aspectos que se han comentado en este análisis de viabilidad sobre esta vía de trabajo futuro, se concluye diciendo que los casos de uso no son extrapolables directamente, ya que las interfaces no son del tipo Ethernet. Esto implicaría rehacer el módulo de *parsing* de capa dos, algo admisible y factible.

Por tanto, se propone a futuro llevar a cabo dicho desarrollo, además de estudiar y analizar distintas variables que entran en juego en un entorno inalámbrico de baja capacidad. Como son la batería, la memoria, el numero de transmisión y recepciones, entre otras. De esta forma se podrá concluir qué tecnología, si P4 ó XDP, nos permitirá programar dispositivos IoT con una mayor eficiencia en redes 802.15.4.

Bibliografía

- [1] 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.
- [2] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks: an authoritative review of network programmability technologies.* ” O’Reilly Media, Inc.”, 2013.
- [3] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, “Exploring container virtualization in iot clouds,” in *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2016, pp. 1–6.
- [4] M. Capra, R. Peloso, G. Masera, M. Ruo Roch, and M. Martina, “Edge computing: A survey on the hardware requirements in the internet of things world,” *Future Internet*, vol. 11, no. 4, p. 100, 2019.
- [5] M. Ojo, D. Adami, and S. Giordano, “A sdn-iot architecture with nfv implementation,” in *2016 IEEE Globecom Workshops (GC Wkshps)*, 2016, pp. 1–6.
- [6] M. Uddin, S. Mukherjee, H. Chang, and T. V. Lakshman, “Sdn-based multi-protocol edge switching for iot service automation,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2775–2786, 2018.
- [7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
- [8] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “The express data path: Fast programmable packet processing in the operating system kernel,” in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. [Online]. Available: <https://doi.org/10.1145/3281411.3281443>
- [9] D. Levy and N. McKeown, “Overhaul may bring better, faster internet to 100 million homes,” *Stanford University News*, 2003.
- [10] “Onf overview,” <https://www.opennetworking.org/mission/>, accessed: 2020-06-10.
- [11] 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.

- [12] D. Hanes, G. Salgueiro, P. Grossetete, R. Barton, and J. Henry, *IoT fundamentals: Networking technologies, protocols, and use cases for the internet of things.* Cisco Press, 2017.
- [13] “Terminology for constrained-node networks,” <https://tools.ietf.org/html/rfc7228>, accessed: 2020-06-30.
- [14] 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.
- [15] B. O. Stephen Ibanez and M. Arashloo, “P4 tutorials,” in *P4 Tutorials ACM SIGCOMM August 2019*, 2019.
- [16] W. Tu, F. Ruffy, and M. Budiu, “Linux network programming with p4,” in *Linux Plumbers’ Conference 2018*, 2018.
- [17] “Traffic control - debian,” <https://wiki.debian.org/TrafficControl>, accessed: 2020-06-30.
- [18] M. Kerrisk, *Namespaces (7) - overview of Linux namespaces*, The Linux man-pages project, May 2020.
- [19] Michael Kerrisk, *Network namespaces - overview of Linux network namespaces*, The Linux man-pages project, June 2020.
- [20] M. Kerrisk, *veth - Virtual Ethernet Device*, The Linux man-pages project, June 2020.
- [21] 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.
- [22] B. Heller, “Reproducible network research with high-fidelity emulation,” Ph.D. dissertation, Stanford University, 2013.
- [23] 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.
- [24] “Mininet-iot,” <https://github.com/ramonfontes/mininet-iot>, accessed: 2020-07-01.
- [25] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *29th Annual IEEE International Conference on Local Computer Networks*, 2004, pp. 455–462.
- [26] A. Kurniawan, *Practical Contiki-NG: Programming for Wireless Sensor Networks*. Apress, 2018.
- [27] I. GitHub, “Github,” URL: <https://github.com/>, 2016.
- [28] P. L. Consortium *et al.*, “Behavioral model (bmv2),” URL: <https://github.com/p4lang/behavioral-model> [cited 2020-01-21], 2018.

- [29] “Json standard,” <https://www.json.org/>, accessed: 2020-06-26.
- [30] M. Gast, *“802.11 wireless networks: the definitive guide”*. O'Reilly Media, Inc., 2005.
- [31] A. Kato, M. Takai, and S. Ishihara, “Design and implementation of a wireless network tap device for ieee 802.11 wireless network emulation,” in *2017 Tenth International Conference on Mobile Computing and Ubiquitous Network (ICMU)*, 2017, pp. 1–6.
- [32] M. Vipin and S. Srikanth, “Analysis of open source drivers for ieee 802.11 wlans,” in *2010 International Conference on Wireless Communication and Sensor Computing (ICWCSC)*, 2010, pp. 1–5.
- [33] “Wpan arch.” <https://github.com/linux-wpan/wpan-misc/tree/master/architecture>, accessed: 2020-06-26.
- [34] “Cooja simulator,” https://anrg.usc.edu/contiki/index.php/Cooja_Simulator, accessed: 2020-06-10.

A. Anexo I - Pliego de condiciones

Este anexo se ha añadido siguiendo la recomendación oficial de la UAH sobre TFGs. En él se indicarán las condiciones materiales de las distintas máquinas donde se ha desarrollado el proyecto. Por último, se resumirán tanto las limitaciones de *hardware*, como las especificaciones del *software* instalado en las máquinas virtuales donde se llevó a cabo los distintos casos de uso.

A.1. Condiciones materiales y equipos

A continuación, se muestran todos los materiales que se han utilizado en el proyecto. Únicamente se indicarán las características de los materiales vitales para el desarrollo del TFG. De este modo, se pretende que queden recogidos todas las especificaciones técnicas con las se han realizado las pruebas y validaciones del desarrollo.

A.1.1. Especificaciones Máquina A

- Procesador: Intel(R) Core(TM) i7-4790 CPU @ 3.60Ghz
- Memoria: 16GB DRR4 (2 slots de 8GB)
- Gráfica: GeForce GTX 970 Windforce - 4GB
- Sistema operativo: Windows 10 - v1909 (Compilación 18363.836)

A.1.2. Especificaciones Máquina B

- Procesador: Intel Core i7-8750H, Hexa Core 2.2GHz-4.1GHz
- Memoria: 16GB DDR4, 2400MHz
- Gráfica: GeForce GTX0150Ti-4GB GDDR5
- Sistema operativo: Windows 10 - v1903 (Compilación 18362.836)

A.1.3. Especificaciones máquinas virtuales

Todas las máquinas virtuales se han ejecutado sobre la máquina B (A.1.2), y se ha conectado a ellas vía SSH con la máquina A (A.1.1). Las máquinas virtuales se desplegaron sobre el hipervisor **VirtualBox**¹, haciendo uso de su versión v6.0.14 r133895 (Qt5.6.2). Dado que había dos entornos de desarrollo muy definidos, como son el entorno P4 y el entorno XDP, se crearon dos máquinas virtuales pudiendo así aislar posibles conflictos de

¹<https://www.virtualbox.org/>

dependencias. Las dependencias de cada máquina virtual para poder replicar los casos de uso, se pueden instalar haciendo uso de los scripts de instalación suministrados en el repositorio del TFG², pudiendo cualquier interesado replicar los escenarios sin mayor complicación.

A.1.3.1. Máquina virtual entorno P4

- Procesador: Intel Core i7-8750H, 2 cores 2.2GHz-4.1GHz
- Memoria: 8GB DDR4, 2400MHz
- Sistema operativo: Ubuntu 16.04.6 LTS x86_64
- Kernel: 4.15.0-88-generic
- Configuración de red: Modo bridge, conectado a interfaz Intel(R) Wireless-AC 9560

```
n0obie@n0obie-VirtualBox:~$ neofetch
  .-/+o0ssssoo+/-. 
   `:+ssssssssssssssssssss+-` 
   +-ssssssssssssssssssssyssss+- 
   .osssssssssssssssssdMMMNyssso. 
   /sssssssssssshdmmNNmNyNMMMHssssss/ 
   +ssssssssssshydMMMMMMNdddyssssssss+ 
   /ssssssssshNMMMyhyyyyhmNMMMNhssssssss/ 
   .ssssssssdMMMNhssssssssshNMMMdssssssss. 
   +sssshhhyNMMNyssssssssssyNMMMyssssssss+ 
   ossyNMMMNyMMhssssssssssssshmmhssssssso 
   osyNMMMNyMMhssssssssssssshmmhssssssso 
   +sssshhhyNMMNyssssssssssyNMMMyssssssss+ 
   .ssssssssdMMMNhssssssssshNMMMdssssssss. 
   /sssssssshNMMMyhyyyyhdNMMMNhssssssss/ 
   +sssssssssdhydMMMMMMNdddyssssssss+ 
   /sssssssssshdmmNNmNyNMMMHssssss/ 
   .osssssssssssssssssdMMMNyssso. 
   -+ssssssssssssssssyyssss+- 
   `:+ssssssssssssssssss+-` 
   .-/+o0ssssoo/-.

n0obie@n0obie-VirtualBox
-----
OS: Ubuntu 16.04.6 LTS x86_64
Host: VirtualBox 1.2
Kernel: 4.15.0-88-generic
Uptime: 6 mins
Packages: 1953 (dpkg)
Shell: bash 4.3.48
Resolution: 1920x950
DE: Unity
WM: Compiz
WM Theme: Ambiance
Theme: Ambiance [GTK2/3]
Icons: ubuntu-mono-dark [GTK2/3]
Terminal: gnome-terminal
CPU: Intel i7-8750H (2) @ 2.207GHz
GPU: 00:02.0 VMware SVGA II Adapter
Memory: 827MiB / 7976MiB
```

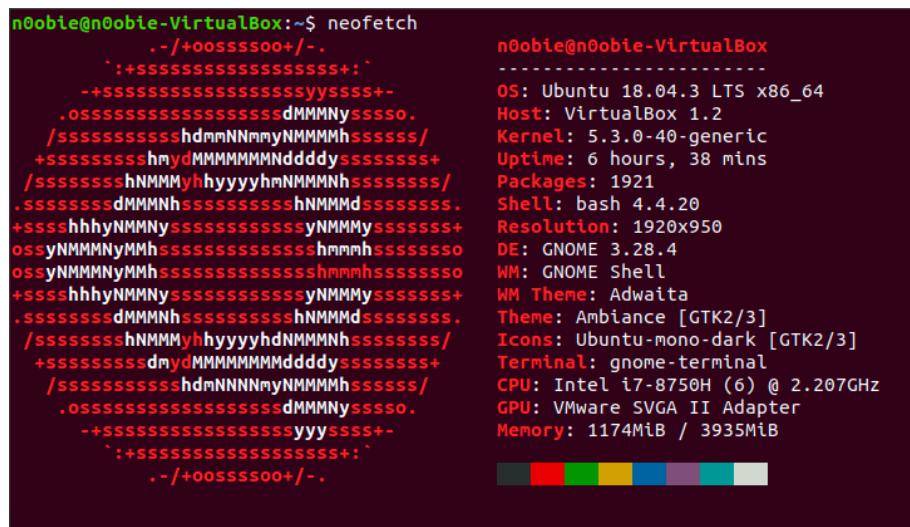


Figura A.1: Especificaciones de la máquina virtual P4

²<https://github.com/davidcawork/TFG>

A.1.3.2. Máquina virtual entorno XDP

- Procesador: Intel Core i7-8750H, 6 cores 2.2GHz-4.1GHz
- Memoria: 4GB DDR4, 2400MHz
- Sistema operativo: Ubuntu 18.04.3 LTS x86_64
- Kernel: 5.3.0-40-generic
- Configuración de red: Modo bridge, conectado a interfaz Intel(R) Wireless-AC 9560



```
n0obie@n0obie-VirtualBox:~$ neofetch
  .-/+o0ssssoo+/- .
   `:+ssssssssssssssssssss+`:
   -+ssssssssssssssssssyyssss+-.
   .osssssssssssssssssdMMMNyssssso.
   /sssssssssshdmmNNmmyNMMMMhssssss/
   +ssssssssshmydMMMMMMNmdddyssssss+.
   /sssssssshnMMMyhyyyyhnNMNMhssssss/
   .ssssssssdMMMNhsssssssssshNMMDssssss.
   +sssshhhyNMMNyssssssssssyNMMMyssssss+.
   ossyNMMMNyMMhssssssssssshmmhssssssso
   ossyNMMMNyMMhssssssssssshmmhssssssso
   +sssshhhyNMMNyssssssssssyNMMMyssssss+.
   .ssssssssdMMMNhsssssssssshNMMDssssss.
   /sssssssshNMMMyhyyyyhdNMNMhssssss/
   +ssssssssdmydMMMMMMMdddyssssss+.
   /sssssssssshdmNNNNmyNMMMMhssssss/
   .osssssssssssssssssdMMMNyssssso.
   -+ssssssssssssssssyyssss+-.
   `:+ssssssssssssssssss+`:
   .-/+o0ssssoo+/- .
n0obie@n0obie-VirtualBox
-----
OS: Ubuntu 18.04.3 LTS x86_64
Host: VirtualBox 1.2
Kernel: 5.3.0-40-generic
Uptime: 6 hours, 38 mins
Packages: 1921
Shell: bash 4.4.20
Resolution: 1920x950
DE: GNOME 3.28.4
WM: GNOME Shell
WM Theme: Adwaita
Theme: Ambiance [GTK2/3]
Icons: Ubuntu-mono-dark [GTK2/3]
Terminal: gnome-terminal
CPU: Intel i7-8750H (6) @ 2.207GHz
GPU: VMware SVGA II Adapter
Memory: 1174MiB / 3935MiB
```

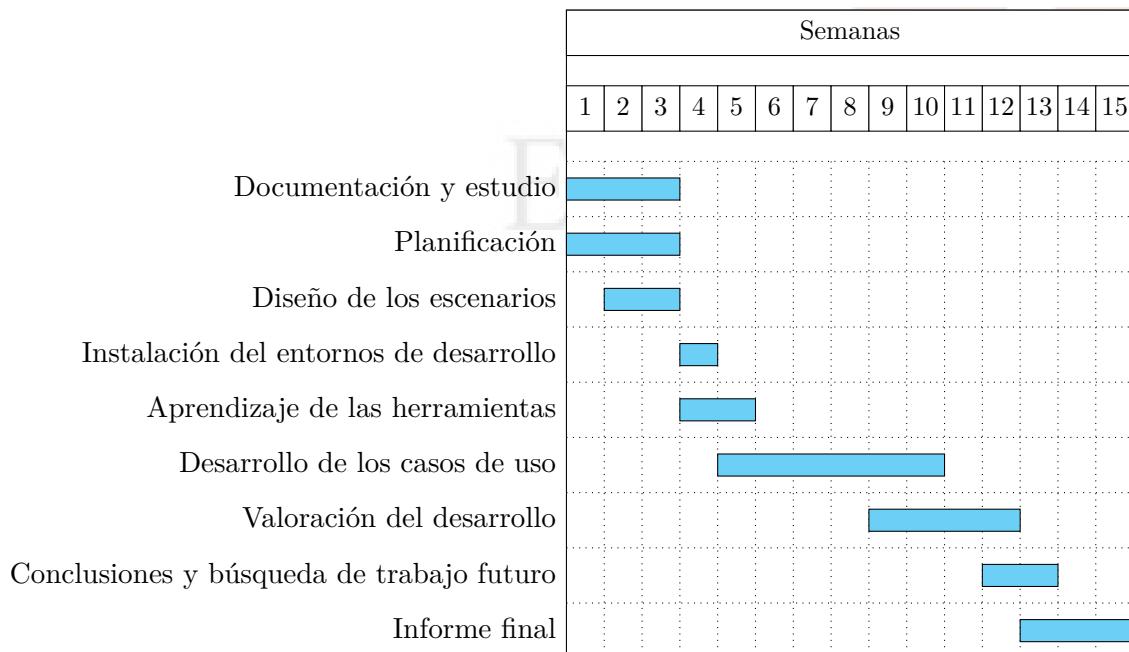
Figura A.2: Especificaciones de la máquina virtual XDP

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 (A.1.3.1, A.1.3.2) 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 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 consultarlo 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

```

¹<https://www.vagrantup.com/>

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 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
<code>arp</code>	<code>ip neigh</code>
<code>ifconfig</code>	<code>ip link</code>
<code>ifconfig -a</code>	<code>ip addr</code>
<code>iptunnel</code>	<code>ip tunnel</code>
<code>route</code>	<code>ip route</code>

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

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

³<https://llvm.org/>

⁴<https://clang.llvm.org/>

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 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.

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.

Versión 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

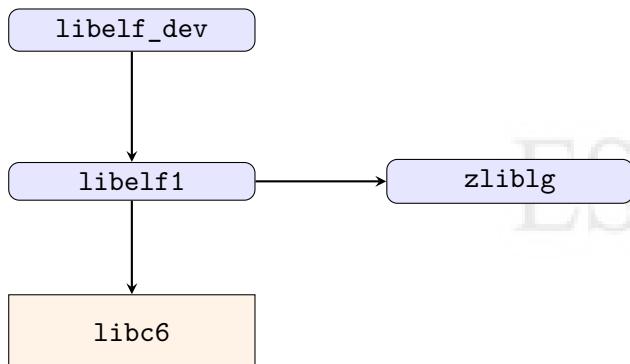


Figura C.1: Ramificación de dependencias de Iproute2.

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.⁵ 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
```

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

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

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.

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
```

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.3.4. 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
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

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

