UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Dispositivos de Red Programables

Proyecto de grado

Belén Brandino

Programa de Grado de Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Febrero de 2022

UNIVERSIDAD DE LA REPUBLICA
URUGUAY

Dispositivos de Red Programables

Proyecto de grado

Belén Brandino

Tesis de Grado presentada al Programa de Grado de Ingeniería en Computación, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Grado en Ingeniería en Computación.

Directores:
Prof. Dr. Ing. Eduardo Grampín
Prof. Dr. Ing. Alberto Castro

Montevideo – Uruguay
Febrero de 2022

Brandino, Belén

Dispositivos de Red Programables / Belén Brandino. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2022.

v, 76 p. 29, 7cm.

Directores:

Eduardo Grampín

Alberto Castro

Tesis de Grado – Universidad de la República, Programa de Ingeniería en Computación, 2022.

Referencias bibliográficas: p. 75 – 75.

1. Dispositivos de red programables, 2. Programación del plano de datos, 3. P4. I. Grampín, Eduardo, Castro, Alberto, . II. Universidad de la República, Programa de Grado de Ingeniería en Computación. III. Título.

RESUMEN

La programabilidad de la red le permite a los usuarios (típicamente un operador de telecomunicaciones) cambiar la funcionalidad de los dispositivos de red, diseñando sus propios algoritmos de procesamiento de paquetes, protocolos y entidades de red, sin depender de soluciones brindadas por fabricantes. Los dispositivos de red programables son clave para enfrentar el advenimiento de la próxima generación de servicios y aplicaciones de red, que soportan múltiples y heterogéneos casos de uso resultando en dispositivos que deben ser óptimos y específicos, pero también básicos y de propósito general. Esto hace que los dispositivos de red se encuentren implementando un amplio conjunto de funciones en constante evolución.

En este proyecto se presentan los principales conceptos de la programabilidad de red, y en particular de la programabilidad del plano de datos, la cual permite diseñar algoritmos de procesamiento de paquetes. Esto hace posible la construcción de dispositivos personalizados, sin comprometer su rendimiento. En conjunto, se presentan distintas herramientas existentes que permiten desplegar dispositivos de red programables, utilizando en particular el lenguaje de programación del plano de datos P4 (*Programming protocol-independent packet processors*). Finalmente, se presentan casos de uso de interés, desarrollando algoritmos del plano de datos y comprobando los beneficios que traen los dispositivos de red programables, en particular aquellos que soportan P4.

Palabras claves:
Dispositivos de red programables, Programación del plano de datos, P4.

Lista de acrónimos

Lista de acrónimos

ALU *Arithmetic Logic Unit* 9

API *Application Programming Interface* 2, 5, 6, 7, 19, 20, 34, 36, 44, 46, 47, 53

ASIC *Application-Specific Integrated Circuit* 3, 7, 21, 34

BGP *Border Gateway Protocol* 6

BMv2 *Behavioral Model version 2* 43, 46, 47, 48, 76

BPF *Berkeley Packet Filter* 12, 13, 14, 41

CFG *Control Flow Graph* 13

CLI Interfaz de línea de comandos 2, 39, 47

CPU *Central Processing Unit* 44

DPDK *Data Plane Development Kit* 12, 14, 15, 16, 44, 45

DPP *Data Plane Programming* 7

IDL *Interface Definition Language* 35

IP *Internet Protocol* 17, 18, 29, 30, 49, 50, 51, 52, 60, 62

ISP Proveedor de servicios de Internet 5

LPM *Longest Prefix Match* 17, 30, 60, 62

MIB Management Information Base 58

NIC *Network Interface Controller* 15

OF *OpenFlow* 6, 7, 42

OVS *Open vSwitch* 41, 42, 43, 46

P4 *Programming protocol-independent packet processors* 4, 11, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 30, 32, 33, 34, 36, 37, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 53, 57, 58, 59, 60, 62, 63, 64, 69, 70, 71, 72, 73, 74, 76

PISA *Protocol-Independent Switch Architecture* 8, 9, 10, 11, 25, 72

PNA *Portable NIC Architecture* 25

POC Prueba de Concepto 43

PSA *Portable Switch Architecture* 25, 44, 47

RPC *Remote Procedure Call* 35, 36, 44

RR *Round Robin* 58, 59, 60, 61, 62, 63, 64, 65, 68, 70

RX *Receive* 15

SDN Redes Definidas por Software 5, 6, 7, 37, 39

SNMP Protocolo Simple de Gestión de Red 58

SR *Segment Routing* 39, 40

SRAM *Static Random-Access Memory* 9

TC *Traffic Control* 14, 15

TCAM *Ternary Content-Addressable Memory* 9

TCP *Transmission Control Protocol* 15, 46, 58, 66, 67, 68

TTL *Time To Live* 30, 51, 62

UDP *User Datagram Protocol* 58, 66, 67

XDP *eXpress Data Path* 12, 14, 15, 42

cBPF *Classic Berkeley Packet Filter* 12, 13, 14

eBPF *Extended Berkeley Packet Filter* 12, 14, 15, 22, 42

protobuf *Protocol Buffers* 34, 35, 76

uBPF *Userspace Berkeley Packet Filter* 14, 22, 41, 42, 43

Tabla de contenidos

Capítulo 1

Introducción

Los dispositivos de red convencionales como routers o switches incluyen tanto el plano de control como el plano de datos en el mismo dispositivo. El plano de control se encarga de establecer políticas de procesamiento de paquetes, tales como a dónde reenviar un paquete o cómo reescribir su encabezado, y administrar las operaciones del dispositivo. Por otro lado, el plano de datos es responsable únicamente de ejecutar la política de procesamiento de paquetes establecida por el plano de control, es decir que se encarga del reenvío, por lo cual muchas veces se le llama "*forwarding plane*". Sin embargo, esta no es la única función que puede implementar el plano de datos. [6]

En el enfoque tradicional, los algoritmos de enrutamiento determinan las rutas que siguen los paquetes desde su origen a su destino y por ende determinan el contenido de las tablas de reenvío de los dispositivos de red. En el ejemplo presentado en la Figura 1.1 se puede ver como cada dispositivo de red ejecuta un algoritmo de enrutamiento, por lo que posee funciones de enrutamiento y a su vez de reenvío.

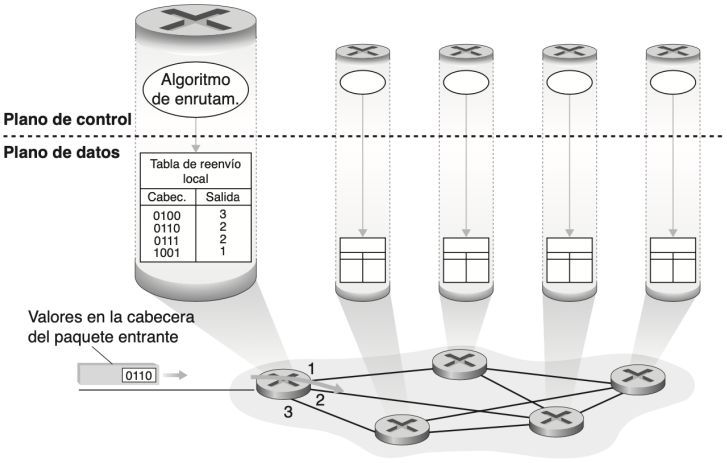


Figura 1.1: Enfoque tradicional: plano de control y plano de datos [38]

Entonces, los dispositivos de red procesan los paquetes utilizando algoritmos del plano de control y plano de datos. Los usuarios poseen cierta libertad para configurar ciertos aspectos del plano de control, como funcionalidades o protocolos, a través de Interfaz de línea de comandos (CLI)s, interfaces web o *Application Programming Interface* (API)s de administración, pero los algoritmos subyacentes solo pueden ser modificados por los fabricantes. Esto hace que agregar nuevas funciones resulte en un largo proceso de desarrollo, haciendo que algunas funcionalidades solo sean implementadas cuando haya una alta demanda, obstaculizando la innovación. Además, al no permitir la modificación del plano de control, es necesario que todos los protocolos de red estén implementados de fábrica, lo cual lleva a ineficiencias (protocolos implementados pero que no son utilizados). En este contexto, se define programabilidad como la habilidad del software o hardware de ejecutar un algoritmo de procesamiento externo, lo cual es diferente a entidades flexibles o configurables, ya que estas solo permiten cambiar ciertos parámetros del algoritmo definido internamente, que permanece igual. Entonces, el término programabilidad de la red refiere a la habilidad de definir el algoritmo de procesamiento ejecutado en una red y particularmente en los nodos individuales, como switches, routers, balanceadores de carga, entre otros. La programabilidad de la red implica entonces poder especificar y cambiar algoritmos del plano de control y del plano de datos, que en la práctica significa que los usuarios finales puedan definir estos algoritmos por sí mismos, sin la necesidad de involucrar fabricantes. Para los proveedores de equipos de red, la programabilidad de la red implica la capacidad de definir algoritmos del plano de datos sin la necesidad de involucrar a los diseñadores

originales del circuito integrado específico de la aplicación (*Application-Specific Integrated Circuit* (ASIC)) de procesamiento de paquetes elegido. Entonces, la programabilidad de la red permite a los fabricantes y los usuarios finales crear redes que se adapten a sus necesidades específicas, además pueden hacerlo significativamente más rápido y generalmente a menor costo sin comprometer rendimiento o calidad del equipo. Para lograr la programabilidad completa de los dispositivos de red, se debe lograr tanto la programación del plano de control, como la del plano de datos. [38][34]

En la actualidad, las redes de telecomunicaciones soportan numerosos y heterogéneos casos de uso, con el fin de brindar soporte a las infraestructuras tecnológicas modernas. Este uso generalizado y esta heterogeneidad complican el diseño de sistemas de comunicaciones y, en particular, sus principales componentes básicos, es decir, los dispositivos de red. Mientras que por un lado existe una tendencia hacia la especialización que permite optimizar los dispositivos de red para una determinada tarea, por otro lado se requiere que los dispositivos de red sean básicos y de propósito general para reducir los costos de ingeniería. Estos opuestos han impulsado la necesidad (y definición) de dispositivos de red programables, lo que permite al usuario (típicamente un operador de telecomunicaciones) cambiar la funcionalidad del dispositivo mediante una interfaz de programación. Un ejemplo de esto es el protocolo NDP [33], que busca solucionar la falta de protocolos de transporte para *data-centers* que provean alta capacidad y baja latencia. Crear un nuevo protocolo implica la necesidad de contar con dispositivos de red capaces de manejar los paquetes definidos bajo este nuevo protocolo. Para lograr esto, sin depender de soluciones brindadas por fabricantes, es que resulta de gran utilidad la programabilidad de los dispositivos de red. Esto fue exactamente lo que hicieron los desarrolladores de NDP, brindando una implementación de su protocolo para los dispositivos de red a través de la programación del plano de datos.

Este proyecto tiene como objetivo principal conocer el estado del arte en la programabilidad de dispositivos de red, debido al poder que brinda a los usuarios y la necesidad de la misma frente al crecimiento inminente tecnológico. En particular, la programación del plano de control fue la que surgió primero, haciendo que la programación del plano de datos fuera el cierre necesario para completar la programabilidad de la red [34]. Es por esto que este trabajo se centra en la programación del plano de datos, con el fin de conocer los modelos

y lenguajes existentes, y en base a esta revisión, el desarrollo de una prueba de concepto, implementando casos de uso de interés. En particular, se utiliza *Programming protocol-independent packet processors* (P4) [7], el lenguaje de programación del plano de datos más usado en la actualidad.

En lo que respecta a la distribución del tiempo, las tareas de relevamiento de estado del arte e introducción al lenguaje tomaron un 30 % del tiempo total del proyecto, por otra parte la evaluación y puesta en funcionamiento de herramientas llevó un 45 % y por último los casos de uso y la redacción de este documento un 25 %.

El resto de este documento se organiza de la siguiente manera: en la sección Fundamentos teóricos se presentan los conceptos principales sobre la programabilidad del plano de control y de datos, incluyendo la descripción del lenguaje P4 y su ecosistema. Luego, en la sección Evaluación de herramientas se presenta la evaluación de algunos ambientes y plataformas capaces de dar soporte a la programabilidad con P4. En la sección Prueba de concepto se presentan dos casos de uso desarrollados con el fin de poner a prueba el lenguaje utilizado y sus capacidades, además del entorno elegido. Por último, en la sección Consideraciones finales se presentan las principales conclusiones del proyecto y el trabajo a futuro.

Capítulo 2

Fundamentos teóricos

2.1. Data Plane Programming

Las Redes Definidas por Software (SDN), el primer y más popular intento de hacer el plano de control programable, procuran hacer a los dispositivos de red programables, a través de la introducción de una API que le permite al usuario pasar por alto los algoritmos del plano de control incorporados y reemplazarlos por algoritmos definidos por él mismo, a través de la separación física (y lógica) del plano control y el plano de datos. Estos algoritmos normalmente están desarrollados en software y corren en un controlador SDN, que posee una visión global de la red. Este controlador remoto calcula y distribuye las tablas de reenvío que debe usar cada dispositivo de red. Suele implementarse en un *datacenter* remoto de alta fiabilidad y redundancia, y puede ser gestionado por el Proveedor de servicios de Internet (ISP) o algún otro proveedor. En este caso, la funcionalidad del plano de datos sigue siendo igual que en el enfoque tradicional, pero el plano de control está separado del dispositivo físico, como se ve en la Figura 2.1. Entonces, los dispositivos solo se encargan del reenvío (tradicional), mientras que el controlador remoto se encarga del enrutamiento y otras funciones de control.

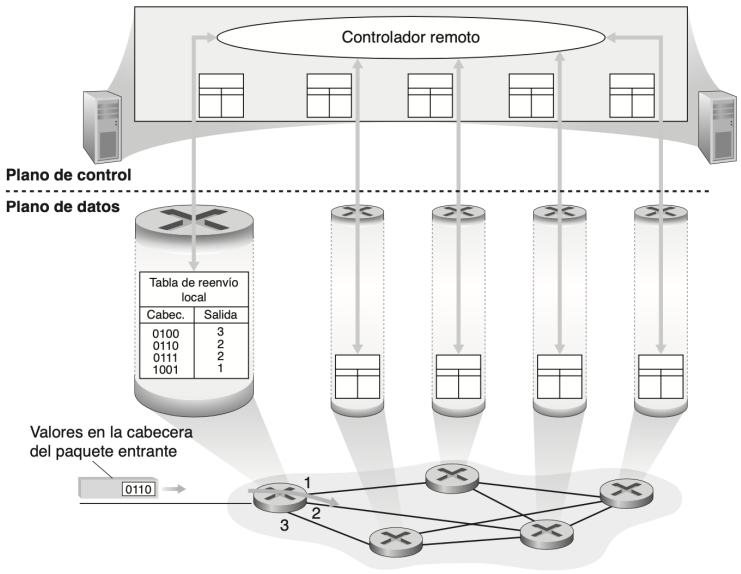


Figura 2.1: SDN: Plano de control y plano de datos [38]

Para permitir la comunicación entre los dispositivos y el controlador, se decidió proveer una API que pudiera ser llamada de manera remota, y de esta manera surgieron las SDN. En consecuencia, se volvió posible implementar algoritmos del plano de control en toda la red en un controlador centralizado. En varios casos, como *datacenters* masivos, estos algoritmos centralizados demostraron ser más simples y eficientes que los algoritmos tradicionales (por ejemplo, *Border Gateway Protocol* (BGP) [55]) diseñados para el control descentralizado de muchas redes autónomas. La estandarización de este enfoque resultó en el desarrollo de *OpenFlow* (OF) [43]. La esperanza era que una vez que OF estandarizara la API para controlar la funcionalidad del plano de datos, las aplicaciones SDN podrían aprovechar las funciones que ofrece esta API para implementar el control de la red. Sin embargo, OF asumía cierta funcionalidad específica del plano de datos que no estaba formalmente especificada, y que además no se podía cambiar. Es por esto, en parte, que surgió la programación del plano de datos. En la Figura 2.2 se puede ver la diferencia entre un dispositivo de red tradicional y con SDN.

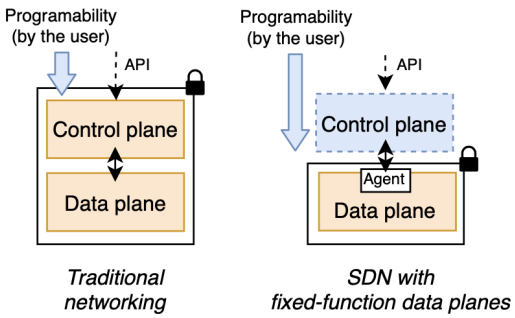


Figura 2.2: Dispositivos de red tradicionales vs SDN con funciones del plano de datos fijas [34]

Entonces, las SDN solo le permiten al usuario proveer su propio plano de control, que le brinda al plano de datos la información de *forwarding* necesaria, pero el plano de datos de los distintos dispositivos sigue bajo el control de los fabricantes. Esta restricción se soluciona a través de la programación del plano de datos, en inglés *Data Plane Programming* (DPP), lo cual habilita que el plano de datos junto con sus algoritmos puedan ser definidos por los usuarios finales, ya sean estos operadores de red o diseñadores de equipos que trabajen con una ASIC para procesamiento de paquetes. Esto cambia sustancialmente el poder que poseen los usuarios, ya que les permite construir equipos de red personalizados sin comprometer rendimiento, escalabilidad, velocidad o potencia. Para redes personalizadas, se pueden diseñar nuevos planos de control y aplicaciones SDN, y para éstas los usuarios finales pueden diseñar sus propios algoritmos de plano de datos que se ajusten perfectamente. La programación del plano de datos no implica necesariamente ninguna provisión de APIs ni requiere soporte para planos de control externos, como lo hace OF. Observar que los algoritmos del plano de datos son responsables de procesar todos los paquetes que pasan a través de un sistema de telecomunicaciones, por ende, son los que definen la funcionalidad, rendimiento y escalabilidad de estos sistemas. Intentar implementar funcionalidades del plano de datos en capas superiores, como el plano de control, generalmente lleva a degradación del rendimiento. [34]

Los algoritmos de plano de datos pueden expresarse, y generalmente lo hacen, a través de lenguajes de programación estándar, sin embargo, estos no se mapean muy bien en hardware especializado, como los ASICs de alta velocidad. Es por esto que muchos modelos de programación del plano de datos se han propuesto como abstracciones del hardware. Los lenguajes se adaptan

para esos modelos de plano de datos y proveen maneras de expresar algoritmos para ellos de forma abstracta. Luego, el código resultante es compilado para su ejecución en un dispositivo específico que admita el modelo de programación del plano de datos para el cual se programó.

Data flow graph abstractions y *Protocol-Independent Switch Architecture* (PISA) son dos ejemplos de modelos de programación del plano de datos. En el caso de *Data flow graph abstractions*, el procesamiento de paquetes se describe a través de un grafo dirigido. Los nodos del grafo representan primitivas simples y reusables que pueden ser aplicadas a los paquetes, por ejemplo, la modificación de un cabezal del paquete. Las aristas dirigidas del grafo representan recorridos de los paquetes, en los cuales las decisiones de recorrido se realizan en los nodos, para cada paquete. En la Figura 2.3 se puede ver un grafo de ejemplo para el reenvío de paquetes IPv4 e IPv6. Algunos lenguajes de programación que implementan este modelo son Click [37], Vector Packet Processors [24] y BESS [59].

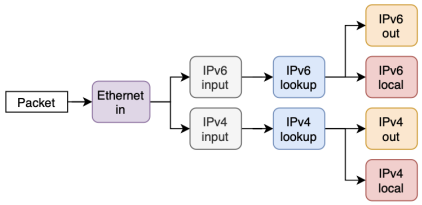


Figura 2.3: Un ejemplo de un grafo del modelo *Data flow graph abstractions* para el reenvío de paquetes IPv4 e IPv6 [34]

En el caso de PISA, se basa en el concepto de *pipeline* programable de correspondencia-acción (*programmable match-action pipeline*) que se adapta de buena manera al hardware de *switching* moderno. Consiste en un *parser* programable, un *deparser* programable y en entre estos dos el *pipeline* programable de correspondencia-acción (*programmable match-action pipeline*), que consiste de varias etapas. La Figura 2.4 representa el modelo PISA.

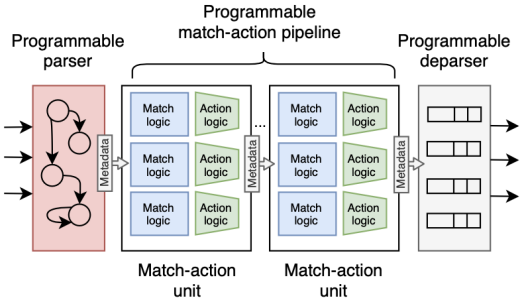


Figura 2.4: Protocol-Independent Switch Architecture [34]

Las etapas de PISA son las siguientes:

- El *parser* programable: le permite a los programadores declarar cabezales arbitrarios junto con una máquina de estados finita que define el orden de estos cabezales en el paquete a procesar. Convierte los paquetes serializados en una forma bien estructurada.
- *Pipeline* programable de correspondencia-acción: consiste de múltiples unidades de correspondencia-acción (*match-action units*). Cada una de estas unidades incluye una o más tablas de correspondencia-acción (*match-action-tables*), donde se busca la coincidencia de una entrada de la tabla con uno (o varios) campos de un paquete. Cada entrada de la tabla posee una acción específica, junto con los datos a proporcionar, por lo que la acción a ejecutar depende de la coincidencia del paquete. La mayor parte de un algoritmo de procesamiento de paquetes está definido en forma de estas tablas. Cada tabla incluye una lógica coincidente acoplada con la memoria (*Static Random-Access Memory* (SRAM) o *Ternary Content-Addressable Memory* (TCAM)) para guardar claves de búsqueda y los datos para las acciones correspondientes. La lógica de las acciones, como operaciones aritméticas o cambios en los cabezales son implementados por *Arithmetic Logic Unit* (ALU)s. Otras lógicas pueden ser implementadas usando objetos con estado, por ejemplo, contadores, *meters* o registros que son guardados en la SRAM. Un plano de control maneja la lógica de coincidencia al escribir entradas en las tablas para influenciar el comportamiento del dispositivo en tiempo de ejecución.
- El *deparser* programable: los programadores declaran como se serializan los paquetes.

PISA provee un modelo abstracto que es aplicado de varias maneras para crear arquitecturas concretas, de manera que permite especificar *pipelines* que

contienen diferentes combinaciones de bloques programables, por ejemplo, un *pipeline* sin *parser* o sin *deparser*, uno con dos *parsers* y dos *deparsers*, y múltiples unidades de correspondencia-acción entre ellos. Asimismo permite utilizar componentes especializados que son requeridos para procesamiento avanzado, como cálculos de suma de comprobación o *hash*.

Además de los componentes programables de PISA, las arquitecturas de switches típicamente incluyen también componentes configurables de funcionalidad fija. Por ejemplo, los bloques de puertos de entrada/salida que reciben o envían paquetes, los motores de replicación de paquetes que implementan *multicast* o la clonación/duplicación de paquetes y los administradores de tráfico que son responsables del almacenamiento en *buffer* de paquetes, en la cola y la planificación.

Un paquete procesado por un *pipeline* PISA consiste de la carga útil (*payload*) del paquete y los metadatos del paquete. PISA solo procesa los metadatos de los paquetes (que viajan desde el *parser* al *deparser*), pero no la carga útil, que viaja por separado. Los metadatos pueden ser divididos en:

- *Packet headers* (cabezales de paquetes): son metadatos que corresponden a los cabezales de los protocolos de red. Generalmente son extraídos en el *parser*, y emitidos en el *deparser*.
- *Intrinsic metadata* (metadatos intrínsecos): son metadatos relacionados con los componentes de funcionalidad fija. Los componentes programables pueden recibir información de los componentes de funcionalidad fija a través de la lectura de los metadatos intrínsecos que producen, o controlar su comportamiento a través de la configuración de los metadatos intrínsecos que consumen estos componentes. Por ejemplo, el bloque de puerto de ingreso genera metadatos que representan el número del puerto de ingreso, que puede ser utilizado en las unidades de correspondencia-acción. En cambio, para enviar un paquete, las unidades de correspondencia-acción generan metadatos intrínsecos que representan el número del puerto de salida, que es consumido por el *traffic manager* y/o el bloque de puerto de salida.
- *User-defined metadata* (metadatos definidos por el usuario): comúnmente se les llama simplemente *metadata*, y son almacenamiento temporal, similar a las variables locales en otros lenguajes de programación. Le permite a los desarrolladores agregar información a los paquetes que pueda

ser utilizada a lo largo del *pipeline* de procesamiento.

Observar que todos los *metadata* son descartados cuando el paquete correspondiente sale del *pipeline* de procesamiento, por ejemplo cuando se descarta el paquete o cuando sale del switch. La Figura 2.5 representa una arquitectura de switch típica basada en PISA. La misma comprende un *pipeline* de ingreso y de egreso programable y tres componentes de funcionalidad fija: un bloque de ingreso, uno de egreso y un motor de replicación junto con un *traffic manager* entre el *pipeline* de ingreso y el de egreso.

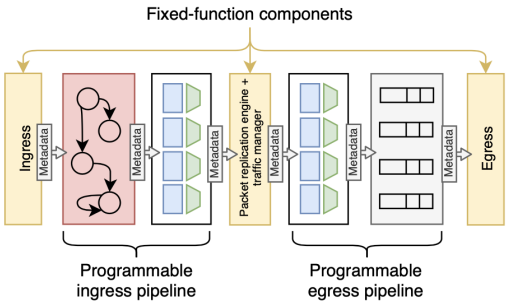


Figura 2.5: Arquitectura de switch típica basada en PISA [34]

Incluyendo los ya mencionados, existen varios modelos para programación del plano de datos, cada uno con sus implementaciones y lenguajes de programación. P4 es actualmente la abstracción, lenguaje de programación y concepto de programación del plano de datos más extendido. Forma parte de los lenguajes para describir algoritmos para PISA. Fue publicado por primera vez como un artículo de investigación en 2014 [7] y actualmente es desarrollado y estandarizado por P4 Language Consortium. Es soportado por varios dispositivos de software y hardware, y es ampliamente aplicado en la academia y en la industria. Su especificación es abierta y pública. [34][40][6] Otros lenguajes de programación del plano de datos para PISA son FAST [44], OpenState [5], Domino [57], FlowBlaze [53], Protocol-Oblivious Forwarding [58] y NetKAT [1].

2.2. Procesamiento de paquetes

En la búsqueda de soluciones para mejorar la programabilidad de red, surgieron soluciones destinadas a mejorar el rendimiento del procesamiento de red de host final. Entre estos se encuentran *Berkeley Packet Filter* (BPF), *eXpress Data Path* (XDP) y *Data Plane Development Kit* (DPDK). [62]

2.2.1. BPF

2.2.1.1. cBPF

Inspirado en distintos filtros de paquetes en el núcleo del sistema operativo, BPF surgió como una solución para realizar filtrado de paquetes en el núcleo de sistemas Unix BSD. Inicialmente, el código de una aplicación se pasaba desde del espacio de usuario al núcleo, donde luego se verificaba el mismo para garantizar seguridad y evitar fallas del núcleo. Después de pasar esta verificación, el sistema adjuntaba el programa a un socket y se ejecutaba por cada paquete que llegaba. BPF surgió como una manera de filtrar paquetes lo antes posible, evitando la necesidad de copiar paquetes del espacio del núcleo al espacio del usuario para filtrarlos a través de las herramientas de monitorización de la red del espacio de usuario. De esta manera, se mejoró significativamente el rendimiento del filtrado de paquetes, comparado con soluciones ya existentes. BPF permite agregar un filtro a cualquier *socket* desde un programa del espacio de usuario. Esta versión original de BPF es comúnmente llamada *Classic Berkeley Packet Filter* (cBPF), mientras que la versión extendida, a la cual se le agregaron distintas componentes y funcionalidades, se la llama *Extended Berkeley Packet Filter* (eBPF). Una de las herramientas más populares que usa BPF es la biblioteca `libpcap`, que es usada por la herramienta `tcpdump`, ya que al usar `tcpdump` un usuario puede elegir una expresión para filtro de paquetes, de manera que solo se capturen los que coincidan con la expresión. Esta expresión puede ser reducida por un compilador a código BPF. [60] [63]

BPF tiene dos componentes principales: la *network tap* y el filtro de paquetes. La *network tap* junta copias de paquetes de los *drivers* de dispositivos de red y los entrega a las aplicaciones de escucha. El filtro decide si se debe aceptar un paquete y si lo hace, cuánto de éste copiar en la aplicación de escucha. La Figura 2.6 muestra cómo funciona BPF. Normalmente, cuando llega un paquete a una interfaz de red, el *link-level driver* (controlador de dispositivo de nivel de enlace) lo envía al *stack* de protocolos del sistema. Sin embargo, cuando BPF se encuentra escuchando en dicha interfaz, el controlador llamará primero a BPF. Entonces, BPF envía el paquete al filtro de cada proceso participante. Este filtro, definido por el usuario, decide si un paquete será aceptado y cuántos bytes de cada paquete deberían ser guardados. Por cada filtro que acepta el paquete, BPF copia la cantidad de datos solicitados en el *buffer* asociado con este filtro, y entonces el controlador de dispositivo toma el control

de nuevo. Si el paquete no estaba dirigido a *localhost*, el controlador regresa de la interrupción, de lo contrario, sigue el procesamiento normal del protocolo. Dado que un proceso puede querer ver todos los paquetes en una red, y como el tiempo entre cada paquete puede ser solo de unos pocos microsegundos, no es posible hacer una llamada al sistema de lectura por cada paquete, por lo que BPF debe recopilar los datos de varios paquetes y retornarlos como una unidad cuando la aplicación de monitorización hace una lectura. Para mantener los límites de los paquetes, BPF encapsula los datos capturados de cada paquete con un encabezado que incluye un *timestamp*, largo y *offsets* para la alineación de datos. Observar que la eficiencia de los filtros es muy importante, ya que la mayoría de las aplicaciones que capturan paquetes descartan más de los que aceptan. cBPF utiliza el modelo de filtro *directed acyclic Control Flow Graph* (CFG). [42, 60]

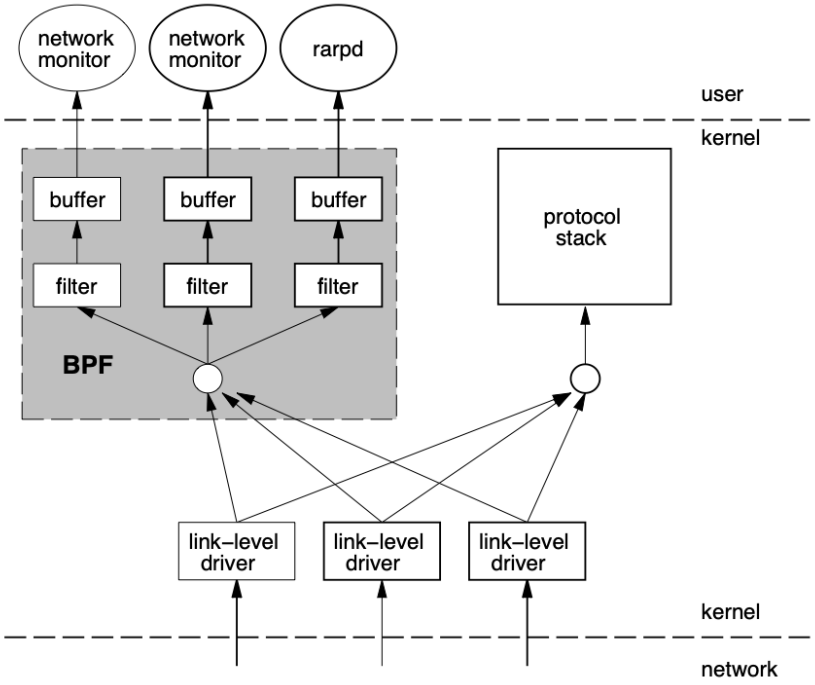


Figura 2.6: Funcionamiento de BPF [42]

2.2.1.2. eBPF

Aunque BPF es muy útil para el filtrado de paquetes, otras áreas pueden beneficiarse de su habilidad para programar el núcleo. Se introdujeron muchas mejoras en BPF para transformarlo en una *universal in-kernel virtual machine* (máquina virtual universal en el núcleo). Esta nueva versión es llamada eBPF.

Se cambiaron varios aspectos de la arquitectura, con la adición de algunas funcionalidades importantes como la capacidad de persistir datos entre ejecuciones, y compartir información entre estas y el espacio de usuario y se agregó la opción de poder llamar funciones que corren dentro del núcleo. También se pueden cambiar, modificar o recargar los programas en tiempo de ejecución. Además, se agregaron *function calls* en la arquitectura, en las cuales se pasan los parámetros a través de registros, al igual que en el hardware nativo. Esto hace que sea posible mapear una *function call* de eBPF a una instrucción de hardware, lo cual resulta en casi ningún *overhead*, por lo que actualmente, las instrucciones de cBPF son convertidas internamente a instrucciones de eBPF, pudiendo mejorar la *performance*. Entonces, eBPF proporciona un conjunto de instrucciones y un entorno de ejecución dentro del núcleo de Linux y es utilizado para modificar el procesamiento de paquetes, además permitiendo la programación de dispositivos de red. Para esto, un desarrollador escribe una aplicación en lenguaje C restringido y lo compila en instrucciones eBPF. [63]

Por último, se cuenta con *Userspace Berkeley Packet Filter* (uBPF), que vuelve a implementar la máquina virtual basada en el núcleo eBPF. Mientras que BPF se diseñó originalmente para permitir la ejecución segura de código en el núcleo, el proyecto uBPF permite ejecutar programas BPF en el espacio del usuario. Por lo tanto, la máquina virtual uBPF se puede integrar fácilmente con aplicaciones que pasan por alto el núcleo (por ejemplo, DPDK y XDP). [50]

2.2.2. XDP

En las redes de computadoras, los *hooks* son utilizados para interceptar paquetes antes o durante la llamada en el sistema operativo. El núcleo de Linux expone varios *hooks* a los cuales se pueden enlazar programas eBPF, habilitando colección de datos y manejo de eventos personalizado. Dos de estos *hooks* son el XDP y el *Traffic Control* (TC), que juntos permiten procesar paquetes cerca de la tarjeta de red (*Network Interface Controller* (NIC)), habilitando la creación y desarrollo de múltiples aplicaciones de red.

Los paquetes que entran al sistema operativo son procesados por varias capas en el núcleo; la capa de socket, stack *Transmission Control Protocol* (TCP), Netfilter, TC, el XDP y la tarjeta de red. Esto se puede ver en la Figura 2.7. Los paquetes destinados al espacio de usuario atraviesan todas estas capas, y pueden ser interceptados y modificados en el camino.

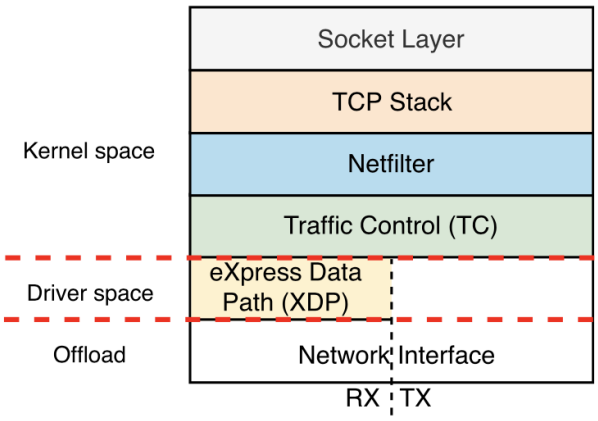


Figura 2.7: Stack de red del núcleo de Linux [63]

XDP es la capa más baja del stack de red del núcleo de Linux. Está presente solamente en el *Receive* (RX) path, dentro del *driver* de red, permitiendo el procesamiento de paquetes en el punto más temprano del *stack* de red, incluso antes de que el sistema operativo realice la asignación de memoria. Expone un *hook* al cual se pueden enlazar programas eBPF. En este *hook*, los programas son capaces de tomar decisiones rápidas sobre los paquetes entrantes además de poder realizar modificaciones arbitrarias en los mismos, evitando el *overhead* impuesto por el procesamiento en el núcleo. Esto hace que XDP sea el *hook* con mejor rendimiento en términos de velocidad para aplicaciones. Luego de procesar un paquete, un programa XDP devuelve una acción, que representa el veredicto final sobre qué se debe hacer con el paquete luego de terminado el programa.[63]

2.2.3. DPDK

DPDK es una solución de software de código abierto de plano de datos optimizada, desarrollada por Intel para sus procesadores de múltiples núcleos. El propósito es proveer a los programadores un framework simple y completo para el rápido procesamiento de paquetes. De esta manera, los usuarios pueden crear prototipos o agregar su propio protocolo al *stack*, según sus necesidades [4]. En resumen, DPDK es una biblioteca de espacio de usuario, que se encarga de proveer funciones al programador que le permiten interceptar paquetes antes de pasar por el núcleo y procesarlos según sus necesidades, a su vez acelerando el procesamiento.

DPDK implementa un modelo de *run to completion* para el procesamiento de paquetes, donde todos los recursos deben asignarse antes de llamar a las aplicaciones del plano de datos, ejecutándose como unidades de ejecución en núcleos de procesamiento lógico, llamados *lcores*. Se accede a todos los dispositivos mediante sondeo (*polling*), para no utilizar interrupciones por la sobrecarga de rendimiento que imponen [23]. En la Figura 2.8, se puede ver a la izquierda el procesamiento de paquetes tradicional y a la derecha el procesamiento de paquetes utilizando DPDK. En este último, se puede ver que todas las interacciones con la tarjeta de red se realizan a través de controladores y bibliotecas especiales. [65]

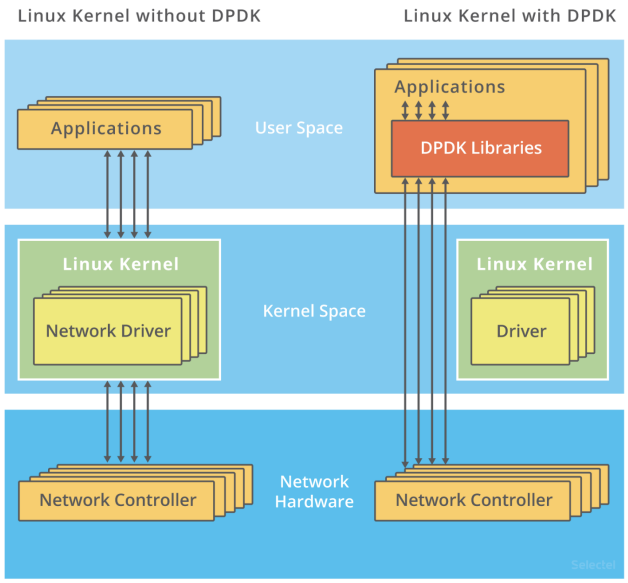


Figura 2.8: Procesamiento tradicional de paquetes vs Procesamiento con DPDK [65]

2.3. El lenguaje P4

P4 es un lenguaje para expresar como los paquetes son procesados por el plano de datos de un dispositivo de reenvío programable, ya sea un switch de software o hardware, una tarjeta de interfaz de red, un router o dispositivo de red. A estos dispositivos, se los conoce como *target*, es decir un sistema de procesamiento de paquetes capaz de ejecutar un programa P4.

P4 le permite al usuario especificar el formato (cabezales) de los paquetes a ser reconocidos por los dispositivos de red y las acciones a realizar en paque-

tes entrantes (por ejemplo, reenviar el paquete, modificar cabezales, etc.). P4 permite además definir comportamientos de reenvío con estado, basados en el uso de registros de memoria que pueden ser accedidos al procesar un paquete.

Muchos *targets* implementan tanto el plano de control como el plano de datos, pero el programa P4 está diseñado para especificar solamente la funcionalidad del plano de datos del *target*. Los programas P4 a su vez definen parcialmente la interfaz por la cual se comunicarán el plano de datos y el plano de control, pero no definen la funcionalidad del plano de control. En la Figura 2.9, se puede ver la diferencia entre un switch tradicional, con funcionalidad fija (arriba) y un switch programable con P4 (abajo), donde en el switch tradicional son los fabricantes quienes deciden la funcionalidad del plano de datos. El plano de control controla el plano de datos, mediante el manejo de entradas en tablas (por ejemplo, tablas de reenvío), la configuración de objetos especializados y el procesamiento de paquetes de control o eventos asincrónicos, tales como cambios de estado de enlaces. En cambio, en un switch programable con P4 la funcionalidad del plano de datos no es fijada de antemano, sino que es definida por el programa P4, por ende el mismo no tiene conocimiento sobre protocolos de red existentes. A su vez, el plano de datos se comunica con el plano de control utilizando los mismos canales que en el switch tradicional, solo que el conjunto de tablas y otros objetos del plano de datos ya no es fijo, sino que los mismos están definidos por el programa P4. Entonces, el programa P4 podrá definir el formato de las tablas, lo cual incluye:

- El campo que será la clave (*key*) de la tabla, por ejemplo, la dirección de destino del cabezal *Internet Protocol* (IP), el protocolo del cabezal Ethernet, entre otros.
- El algoritmo de coincidencia para la búsqueda en la tabla, por ejemplo *Longest Prefix Match* (LPM).
- Las posibles acciones a tomar sobre los paquetes, definiendo también la lógica de las mismas y qué parámetros utilizarán.

El encargado de poblar esta tabla es el plano de control, según el diseño especificado por el programa P4. Por ejemplo, si la clave de la tabla definida por el programa P4 fuera la dirección IP de destino, el plano de control será quien seleccione para cada IP qué acción se tomará (dentro de las definidas por el programa P4) y brinde los valores de los parámetros de tal acción (si el programa P4 definió, por ejemplo, que hay un parámetro que representa el

puerto por donde debe salir el paquete, el plano de control brinda el valor de ese parámetro).

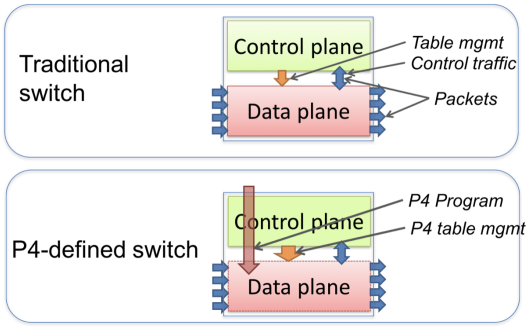


Figura 2.9: Switch tradicional vs switch programable con P4 [10]

Se llamará arquitectura a un conjunto de componentes programables con P4 y las interfaces del plano de datos entre ellos. La arquitectura P4 identifica los bloques programables P4 que pueden haber dentro de un programa P4 (definido para esa arquitectura). Se puede considerar como un contrato entre el programa y el *target*, por lo cual el fabricante del *target* debe proveer tanto un compilador como una definición de la arquitectura para el *target*. Los programas P4 son escritos para arquitecturas y no para los dispositivos particulares, ya que varios dispositivos pueden utilizar la misma arquitectura.

Las abstracciones centrales proporcionadas por el lenguaje P4 son:

- Tipos *header* (cabezales): estos describen el formato (el conjunto de campos y el tamaño de los mismos) de cada cabezal dentro de un paquete. Por ejemplo, el siguiente código define el header Ethernet con nombre `ethernet_t`, con los campos `dstAddr` de 48 bits, `srcAddr` de 48 bits y `etherType` de 16 bits, en ese orden.

```
header ethernet_t {
    bit<48> dstAddr
    bit<48> srcAddr
    bit<16> etherType
}
```

- *Parsers*: describen la secuencia de *headers* permitidos en los paquetes recibidos, como identificar esas secuencias, y que *headers* y campos extraer de los paquetes. De esta manera, luego es posible acceder a los campos individualmente.

- Tablas: asocian las claves definidas por el usuario con acciones y parámetros. Estas tablas se pueden utilizar para implementar tablas de reenvío, tablas de flujo, etc.
- Acciones: son fragmentos de código que describen como se manipulan los campos de los cabezales de los paquetes y los metadatos. A su vez, pueden tener datos (parámetros), proporcionados por el plano de control en tiempo de ejecución.
- Unidades de correspondencia-acción (*match-action units*): construye claves de búsqueda (*lookup keys*) a partir de campos de paquetes o metadatos calculados, luego realiza la búsqueda en la tabla a partir de la clave construida, eligiendo una acción (incluyendo los parámetros asociados) para ejecutar, y por último la ejecuta.
- Flujo de control (*Control Flow*): expresa un programa imperativo que describe el procesamiento de paquetes en un *target*, incluyendo la secuencia de invocaciones de unidades de correspondencia-acción. En definitiva, define la secuencia de ejecución de las tablas, habilitando ejecuciones condicionales.
- Objetos externos: son construcciones específicas de cada arquitectura P4 y pueden ser manipuladas por los programas P4 a través de APIs. Su comportamiento interno está cableado, es decir que no son programables usando P4. Un ejemplo son las unidades de suma de comprobación. Diferentes arquitecturas tienen diferentes objetos externos.
- Metadatos definidos por el usuario (*user-defined metadata*): son metadatos que define el usuario asociados con cada paquete.
- Metadatos intrínsecos (*intrinsic metadata*): son metadatos proporcionados por la arquitectura, asociados con cada paquete, por ejemplo el puerto de ingreso por el cual se recibió el paquete.

La Figura 2.10 muestra un flujo de trabajo de herramientas típico cuando se programa un *target* usando P4. Los fabricantes proveen el *framework* de implementación de hardware o software, una definición de la arquitectura y un compilador P4 para el *target*. Los programadores son quien proporcionan el programa P4, que debe estar diseñado para la arquitectura específica, ya que el código fuente no es consumido directamente por los dispositivos de red y por ende necesita ser compilado. La compilación del programa genera una configuración del plano de datos que implementa la lógica de reenvío descrita

en el programa de entrada, y una API para administrar el estado de los objetos del plano de datos desde el plano de control, es decir para la comunicación entre estos. [10]

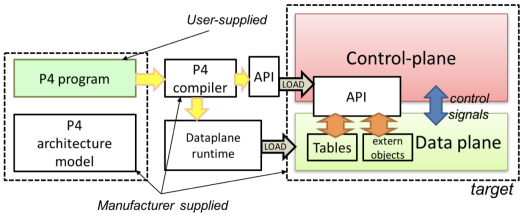


Figura 2.10: Programación de un *target* con P4 [10]

El lenguaje P4 cuenta con distintos elementos [13]:

- Los *parsers* son los principales bloques programables que se describen en P4. Son implementados como máquinas de estado y su propósito es extraer cabezales, indicando su orden.
- Los bloques *control* son responsables del procesamiento principal de correspondencia-acción. En los bloques *control* se definen y aplican las tablas, junto con las acciones de las mismas.
- Expresiones: La implementación de los *parsers* y *controls* es provista por el programador P4, para lo cual usa expresiones y tipos de datos definidos en el lenguaje. Estas expresiones pueden utilizar operaciones básicas y operadores, como operaciones aritméticas y lógicas estándar.
- Tipos de datos:
  - Tipos básicos, como `bit<n>`, que representa una cadena de bits (sin signo) de tamaño n, `int<n>` que representa un entero con signo de tamaño n, o `varbit<n>`, que representa una cadena de bits de largo variable de tamaño máximo n.
  - Tipo `header`, que es una colección ordenada de miembros, que pueden contener `bit<n>`, `int<n>`, o `varbit<n>`. Son alineados por bytes, pueden ser validos o inválidos y proveen varias operaciones para manipular el bit de validez, como `isValid()`, `setValid()` y `setInvalid()`.
  - Tipo `struct` que consiste de una colección sin ordenar de miembros, sin restricción de alineamiento. Por lo general, los programas P4 combinan una colección de *headers* en un *struct* para utilizar en los

bloques programables. Recordar que el orden de *headers* a reconocer en el paquete lo determina el *parser*.

- Tipo `header stack`, que representa un arreglo de *headers* del mismo tipo.
- Tipo `header union`, que es una alternativa al tipo anterior, que contiene a lo sumo uno de los diferentes *headers*. Por ejemplo, se podría declarar un *header union* que consista de los cabezales IPv4 e IPv6, si se espera que todos los paquetes contengan solo uno de estos y no ambos.

Observar que con `typedef`, se puede definir un nombre alternativo para un tipo, con `const` se pueden definir constantes y también se puede utilizar la primitiva `#define`.

P4 es un lenguaje específico de dominio que está diseñado para implementarse en una gran variedad de *targets*, incluyendo tarjetas de interfaz de red programables, FPGAs[1], switches de software y ASICs de hardware, y es por esto que el lenguaje está restringido a construcciones que puedan ser implementadas de manera eficiente en todas estas plataformas. Algunos ejemplos de *targets* basados en hardware son Barefoot Tofino [36], FPGA [35], o basados en software, como BMv2 (*Behavioral Model version 2*) [12] (el switch de software de referencia de P4), eBPF/XDP [64], P4OvS [49], P4rt-OVS [48] y PISCES [56]. P4c [15] es el compilador de referencia de P4, que soporta ambas versiones del lenguaje P4, $P4_{14}$ [18] (versión discontinuada pero aún soportada en varios *targets*) y $P4_{16}$ [10]. Este compilador incluye los siguientes *backends* [15]:

- p4c-bm2-ss: puede ser usado para el *target simple_switch* (presentado más adelante)
- p4c-dpdk: puede ser usado para el *target* switch de software de DPDK
- p4c-ebpf: puede ser usado para generar código C que puede ser compilado a eBPF
- p4c-graphs: puede ser usado para generar representaciones visuales de un programa P4
- p4c-ubpf: puede ser usado para generar código uBPF que se ejecuta en el espacio de usuario

[1] https://netfpga.org

Sin embargo, p4c es un compilador modular, de manera que agregar distintos *backends* debería ser fácil. Al compilar un archivo $P4_{14}$ o $P4_{16}$, se crearán dos archivos:

- Un archivo con extensión `.p4i` que es el resultado de ejecutar el preprocesador con el programa P4 dado.
- Un archivo con extensión `.json` que es el formato de archivo esperado por el *target*

Además, es posible agregar una flag de manera que el compilador también cree un archivo de texto con formato "P4Info", que contiene una descripción de las tablas y otros objetos del programa P4, que será de utilidad para el control del plano de datos.

En cuanto al rendimiento de dispositivos programables con P4, asumiendo un costo fijo por operaciones de búsqueda en tablas e interacciones con objetos externos, todos los programas P4 (*parsers* y *controls*) ejecutan un número constante de operaciones por cada byte de un paquete recibido y analizado. Aunque los *parsers* pueden contener bucles, siempre que se extraiga algún cabezal en cada ciclo, el paquete en sí proporciona un límite en la ejecución total del *parser*. Esto quiere decir que la complejidad computacional de un programa P4 es lineal en el tamaño total de todos los cabezales, y nunca depende del tamaño de estado acumulado en el procesamiento de datos (por ejemplo, la cantidad de flujos, el tamaño de paquetes procesados, etc.). Estas garantías son necesarias, pero no suficientes, para permitir el procesamiento rápido de paquetes en una variedad de *targets*. [10][40]

Comparado con los sistemas de procesamiento de paquete más novedosos (por ejemplo, basados en escribir microcódigo sobre hardware personalizado), P4 ofrece una cantidad de beneficios significativa, como lo son [10]:

- Flexibilidad: P4 hace que muchas políticas de reenvío puedan expresarse como programas, a diferencia de los switches tradicionales donde se exponen funciones fijas de reenvío a los usuarios.
- Expresividad: P4 puede expresar algoritmos de procesamiento de paquetes sofisticados e independientes del hardware, utilizando solamente operaciones de propósito general y búsquedas en tablas. Además, estos programas son portables a otros *targets* que implementen la misma arquitectura.

- Mapeo y gestión de recursos: Los programas P4 describen recursos de almacenamiento de manera abstracta (cualquier campo de los cabezales), los compiladores mapean estos campos definidos por el usuario con los recursos de hardware disponibles, y maneja detalles de bajo nivel tales como planificación y asignación.
- Ingeniería de Software: Los programas P4 brindan beneficios importantes, como la verificación de tipos, la ocultación de información y la reutilización de software.
- Bibliotecas de componentes: Bibliotecas de componentes proporcionadas por fabricantes pueden ser utilizadas para envolver funciones específicas de hardware en construcciones P4 portables de alto nivel.
- Desacoplamiento de la evolución del hardware y el software: Los fabricantes de los *target* pueden usar arquitecturas abstractas para desacoplar aún más la evolución de detalles arquitectónicos de bajo nivel del procesamiento de alto nivel.
- *Debugging*: los fabricantes pueden proveer modelos de software de una arquitectura para ayudar en el desarrollo y *debugging* de programas P4.

2.3.1. Modelo de Arquitectura

Como se mencionó anteriormente, la arquitectura P4 identifica los bloques programables y sus interfaces de plano de datos. No es necesario que la arquitectura exponga toda la superficie programable del plano de datos, por ejemplo un fabricante podría proveer múltiples definiciones para el mismo dispositivo, cada una con distintas capacidades (por ejemplo, con o sin soporte de *multicast*).

En la Figura 2.11 se puede ver un *target* que posee dos bloques programables P4 (P4 block #1 y P4 block #2), donde se pueden ver las interfaces que hay entre los bloques. Cada uno de estos bloques está programado por un fragmento de código P4 separado. El *target* interactúa con el programa P4 a través de un conjunto de registros o señales de control. Los controles de entrada proveen información al programa P4 (por ejemplo, el puerto de entrada por el cual se recibió el paquete), mientras que los controles de salida pueden ser escritos por el programa P4 para modificar el comportamiento del *target* (por ejemplo, el puerto de salida al cual debe dirigirse un paquete). Los registros o señales de control son representados en P4 como metadatos intrínsecos. A

su vez, los programas P4 pueden guardar y manipular datos pertenecientes a cada paquete utilizando los *user-defined metadata*.

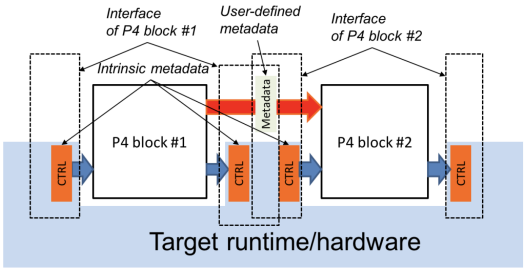


Figura 2.11: Interfaces de un programa P4 [10]

El comportamiento de un programa P4 puede ser completamente descrito en términos de transformaciones que mapean vectores de bits a vectores de bits. Para procesar un paquete, el modelo de arquitectura interpreta los bits que escribe el programa P4 en los metadatos intrínsecos. Por ejemplo, para que un paquete sea reenviado en un puerto de salida específico, un programa P4 puede necesitar escribir el identificador del puerto de salida en un registro de control dedicado. De manera similar, para que un paquete sea descartado, un programa P4 puede necesitar marcar un bit de "descarte" en otro registro de control dedicado. Observar que, los detalles de como se interpretan los metadatos intrínsecos son específicos a la arquitectura.

Los programas P4 pueden invocar servicios implementados por objetos externos y funciones previstas por la arquitectura. La Figura 2.12 representa como un programa P4 invoca los servicios de una unidad de cálculo de suma de comprobación incorporada en un *target*. La implementación de la unidad de suma de comprobación no está especificada en P4, pero su interfaz si. En general, la interfaz para un objeto externo describe cada operación que proporciona, como sus parámetros y tipos de retorno. Estos bloques externos se pueden ver como una caja negra.

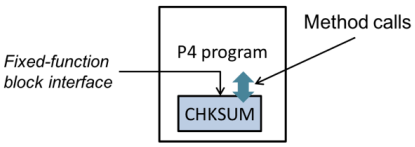


Figura 2.12: Programa P4 invocando los servicios de un objeto de función fija [10]

En general, no se espera que los programas P4 sean portables entre distintas

arquitecturas, por ejemplo, ejecutar un programa P4 que haga *broadcast* de los paquetes a través de la escritura de un registro de control personalizado no va a funcionar de manera correcta en un *target* que no tiene tal registro de control. Sin embargo, los programas P4 escritos para una arquitectura dada deberían ser portables para todos los *targets* que implementen fielmente el modelo, siempre que existan los recursos suficientes. [10]

2.3.2. Arquitectura V1Model

Existen varias arquitecturas P4, tales como *Portable Switch Architecture* (PSA) [32], SimpleSumeSwitch [45], *Portable NIC Architecture* (PNA) [20] o V1Model [19]. La arquitectura V1Model es un buen ejemplo para mostrar los principales conceptos de la arquitectura PISA en P4.

La Figura 2.13 presenta los distintos componentes de la arquitectura V1Model.

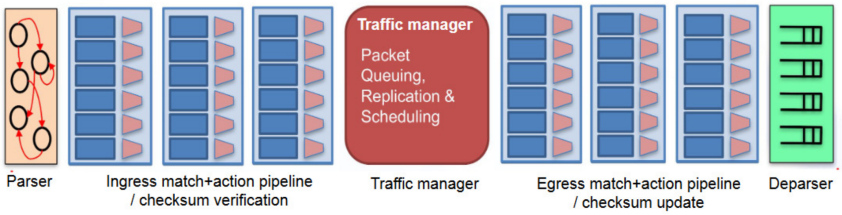


Figura 2.13: Arquitectura V1Model [40]

La arquitectura se compone del *pipeline* de ingreso, el *traffic manager* y el *pipeline* de egreso.

El ingreso está conformado por el *parser* programable, un bloque *control* especializado diseñado para verificar las sumas de comprobación y el *pipeline* de ingreso de correspondencia-acción. Entre el *pipeline* de ingreso y el de egreso la arquitectura V1Model incluye un bloque no programable, que representa el administrador de tráfico, incluyendo las funciones fijas proporcionadas por el fabricante del *target*. Luego, el *pipeline* de egreso está conformado por el *pipeline* de egreso de correspondencia-acción, un bloque *control* especializado diseñado para calcular las sumas de comprobación en los *headers* transmitidos y el *deparser*. Un paquete pasa por todos estos bloques y sale del switch. [40]

2.3.2.1. Un ejemplo: forwarding básico

Con el fin de explicar el lenguaje P4 con más detalle, se presentará y explicará un código[1] de ejemplo para realizar un *forwarding* IPv4 básico. Con este fin, se utilizará la arquitectura V1Model, ya que es la utilizada en la Prueba de concepto y además la misma es un buen ejemplo para ilustrar los conceptos clave. Las arquitecturas definen ciertos campos (dependientes de la arquitectura) como parte de los metadatos intrínsecos. En el caso de V1Model, los metadatos intrínsecos son llamados `standard_metadata` y son representados por el *struct* de tipo `standard_metadata_t`. Es posible consultar en [17] los campos que componen este *struct* y qué representa cada uno de estos.

Al comienzo del programa P4 se definen los *headers* a reconocer en el *parser* y los metadatos definidos por el usuario (llamados `metadata`). En este caso los *headers* serán Ethernet e IPv4. También se debe definir el *struct* que contenga los *headers* declarados. Además se deben incluir los archivos correspondientes al *core* de P4 y a la arquitectura V1Model, que definen la firma de cada bloque. En el caso de un programa $P4_{16}$ basta con incluir el archivo correspondiente a la arquitectura para acceder a los campos del `standard_metadata`, en cambio para el caso de $P4_{14}$ se deben definir algunos *headers* extras para poder acceder a todos estos valores. [17]

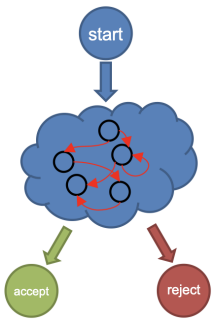
```
#include <core.p4>
#include <v1model.p4>

header ethernet_t {
    bit<9>    dstAddr;
    bit<48>   srcAddr;
    bit<16>   etherType;
}
header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
```

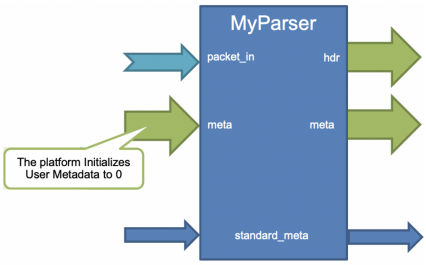
[1] el código fue obtenido de:
https://github.com/p4lang/tutorials/blob/master/exercises/basic/solution/basic.p4

```
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
struct metadata {
   // empty
}
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

Después se define el primer bloque del *pipeline* de ingreso, el *parser*, que debe marcar el orden de los *headers* a reconocer. Cada *parser* (en $P4_{16}$, esto no es dependiente de la arquitectura), es definido como una máquina de estados que tiene tres estados predefinidos: `start`, `accept` y `reject`, además el programador puede definir otros estados. La Figura 2.14a muestra el *parser* como una máquina de estados, que comienza en el estado `start`, con estados intermedios definidos por el usuario y debe terminar en `accept` o `reject`, que indica un error en el *parsing*. Como se puede ver en la Figura 2.14b, el *parser* tiene como parámetros de entrada el paquete entrante, los metadatos definidos por el usuario y el `standard_metadata`. Los parámetros de salida son los *headers* extraídos, los metadatos del usuario y el `standard_metadata`. La dirección de los parámetros se especifica con las directivas `in`, `out` e `inout`.



(a) El *parser* como una máquina de estados [13]



(b) Datos del *parser* [13]

Figura 2.14: Representaciones del *parser*

En cada estado, se ejecutan cero o varias sentencias, y se transiciona a otro estado, usando la primitiva `transition`. Para facilitar la escritura de código, es posible utilizar la primitiva `select`, similar al *case* de C o Java, a diferencia que la expresión para cada caso debe ser un estado del *parser*.

En este ejemplo, se tienen dos estados definidos por el usuario. Al principio se transiciona del estado inicial `start` al estado `parse_ethernet`, definido por el usuario. En este estado se extrae el *header* Ethernet y luego se utiliza un `select`, de manera que si el campo tipo del cabezal Ethernet es IPv4, se transicione al otro estado definido por el usuario, `parse_ipv4`, sino al estado `accept`. Por último, en el estado `parse_ipv4` se extrae el *header* IPv4 y se transiciona al estado `accept`. En la Figura 2.15 se pueden ver las transiciones entre estados.

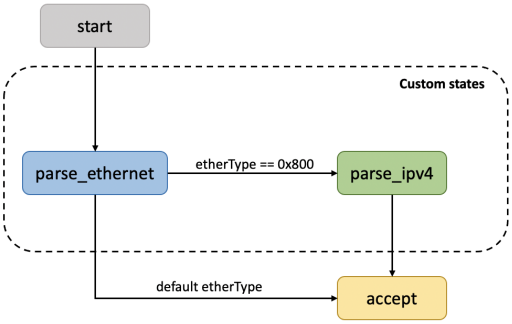


Figura 2.15: Transiciones entre estados del *parser* para el ejemplo de *forwarding* básico

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select (hdr.ethernet.etherType) {
            0x800: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

Los siguientes bloques en el *pipeline* de ingreso son dos bloques *control*, similares a las funciones en lenguajes de programación. Son llamados por un

método `apply()`, tienen parámetros, y pueden llamar a otros bloques. En estos se pueden definir tablas, variables, llamados a externos, acciones, etc. En este caso los bloques *control* son `MyIngress` y `MyVerifyChecksum`

El bloque *control* `MyVerifyChecksum` se deja en blanco, ya que en este caso no es estrictamente necesario realizarlo, sin embargo, se pueden utilizar objetos externos pertenecientes a la arquitectura V1Model para verificar la suma de comprobación, como la función `verify_checksum`.

```
control MyVerifyChecksum(inout headers hdr, inout metadata meta){
    apply{ }
}
```

En el bloque *control* `MyIngress`, la acción a realizar es el reenvío, por lo que primero se debe definir una tabla. En la misma se define con qué clave (puede ser más de una) buscar en la tabla, qué algoritmo de coincidencia usar, qué acciones tomar, y los datos proporcionados para la acción (en caso de existir). Esto le permite al programador diseñar el formato de la tabla, pero es el plano de control quien se encarga de poblarla.

En este caso, la clave será la dirección IP de destino, representada por `hdr.ipv4.dstAddr`. También se debe seleccionar el algoritmo de coincidencia. Dentro del *core* de P4 los algoritmos de interés son `lpm` que representa el algoritmo Longest Prefix Match (el algoritmo elegido en este caso) y `exact`, que representa la búsqueda exacta de la clave. Luego, se definen las posibles acciones de la tabla, que en este caso son `ipv4_forward` y `drop`. Además, es necesario definir la lógica de las mismas. La acción `drop`, que además se define en la tabla como acción por defecto, simplemente utiliza la función `mark_to_drop` que es una función externa definida por la arquitectura V1Model. Esta función setea un campo del `standard_metadata` con un valor específico indicando que el paquete debería ser descartado. La otra acción, que se encarga del reenvío, deberá realizar cuatro cosas:

- Asignar el puerto de salida, que es por el cual debe salir el paquete para llegar al siguiente salto (*next hop*). Este valor se obtiene a partir del plano de control, por eso es un parámetro en la acción. Es decir que cuando ingrese un paquete y su dirección IP de destino coincida con alguna entrada en la tabla, según el algoritmo LPM, uno de los parámetros encontrados en la tabla para la acción será el puerto de salida. Para actualizar el puerto de salida del paquete, basta con asignar el parámetro

obtenido al campo `standard_metadata.egress_spec`, que controla por cuál puerto debe salir el paquete.

- Actualizar la dirección MAC de origen del paquete asignándole la dirección MAC del puerto de salida. Este valor nuevamente se obtiene a partir del plano de control. Basta con asignar este valor al campo de dirección de origen del header Ethernet (`hdr.ethernet.srcAddr`).
- Actualizar la dirección MAC de destino del paquete asignándole la MAC del siguiente salto. La MAC del siguiente salto es un parámetro que también es obtenido a partir del plano de control. Para esto basta con asignarle ese valor al campo de dirección de destino del header Ethernet (`hdr.ethernet.dstAddr`).
- Decrementar el *Time To Live* (TTL) del paquete. Para esto es suficiente con decrementar este valor del header IP (`hdr.ipv4.ttl`).

Por último, en `apply` se chequea si el paquete efectivamente tenía un *header* IPv4 (con la función externa `isValid`), y se aplica la tabla al paquete que está siendo procesado.

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t srcAddr, macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = srcAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
```

```
            ipv4_lpm.apply();
        }
    }
}
```

Luego, viene el *pipeline* de egreso. En este caso se tiene un *control* `MyEgress` con su correspondiente *apply* vacío, ya que no se necesita hacer nada más en cuanto a procesamiento del paquete.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    apply { }
}
```

Luego, un *control* `MyComputeChecksum` que se encarga de computar nuevamente la suma de comprobación, con la función `update_checksum`, función externa de V1Model.

```
control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
    apply {
        update_checksum(
            hdr.ipv4.isValid(),
                { hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
                hdr.ipv4.fragOffset,
                hdr.ipv4.ttl,
                hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}
```

Como último bloque se encuentra el *deparser*, donde solo es necesario emitir los *headers* deseados para serializar los paquetes. Esto se logra con la función `emit`, perteneciente al *core* de P4, que serializa un *header* si es válido.

```
control MyDeparser(acket_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

Por último, es necesario realizar una "instanciación de paquete". Debe haber al menos uno llamado "*main*", en cualquier programa $P4_{16}$ completo. [25]

```
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

Para conocer más detalles sobre las primitivas, en [25] se puede consultar un ejemplo de programa P4 comentado en detalle, o la definición de la arquitectura V1Model [19] .

2.3.2.2. Contadores y registros

Como se mencionó anteriormente, los metadatos definidos por el usuario representan almacenamiento temporal, de manera que pueden mantener estado durante el procesamiento de un paquete. Para lograr mantener estado entre paquetes, es posible utilizar contadores y registros, objetos externos pertenecientes a la arquitectura V1Model. Estos objetos serán de utilidad para los casos de uso presentados en Prueba de concepto.

Los registros (tipo *register*) permiten guardar datos arbitrarios y pueden ser leídos y escritos desde el programa P4. Son asignados en arreglos, es decir que se tiene un arreglo de valores arbitrarios y declarados de la siguiente manera:

```
register<Type>(N) nombre_reg;
```

Donde *Type* es el tipo de los datos a guardar (por ejemplo bit<9>), N es el tamaño del arreglo, y *nombre_reg* el nombre del registro. En la Figura 2.16 se puede ver una representación de un registro, donde en cada posición del arreglo se tiene un valor del tipo *Type*. Estos valores pueden ser leídos con la función `read`, usando como parámetros el índice del arreglo que se quiere leer (en la figura el índice 0) y dónde se desea escribir el resultado (*result*). También pueden ser escritos con la función `write`, usando como parámetros el índice del arreglo que se quiere escribir (0) y qué se desea escribir en el (*value*).

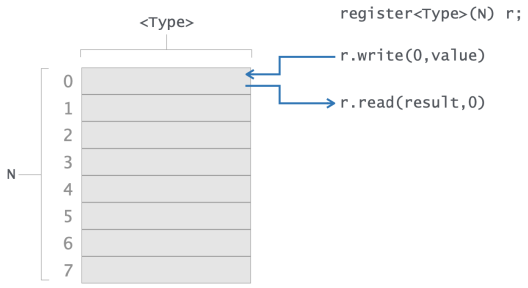
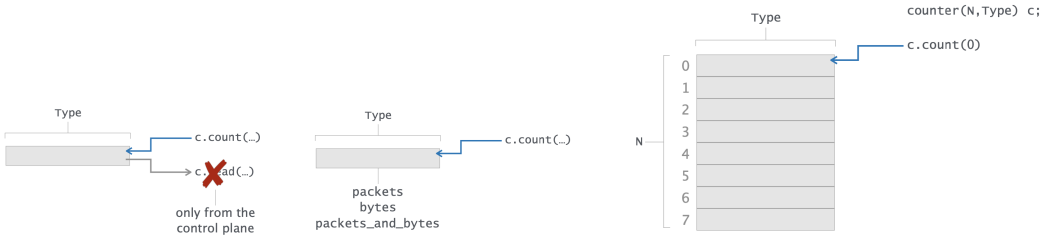


Figura 2.16: Representación de registros de la arquitectura V1Model [61]

Los contadores (tipo *counter*) sirven para contar, por ejemplo eventos. Los contadores, al igual que los registros, son asignados en arreglos y declarados de la siguiente manera:

```
counter(N, Type) nombre_cont;
```

Donde *Type* es el tipo del contador, *N* es el tamaño del arreglo, y *nombre_cont* el nombre del contador. La principal diferencia es que los contadores no pueden guardar datos arbitrarios, sino que son de tres tipos distintos, *packets* es decir que cuentan paquetes, *bytes* que cuentan bytes, o *packets_and_bytes* que cuentan ambos. Desde el programa P4, solo se puede utilizar la función `count`, que incrementa el valor del contador según el tipo. No es posible leer un contador desde el programa P4, pero si es posible leerlo desde el plano de control. [61]



(a) Funciones sobre el tipo contador [61]

(b) Tipos de contador [61]

(c) Representación de contadores de la arquitectura V1Model [61]

Figura 2.17: Representaciones de contadores de la arquitectura V1Model

2.3.3. P4Runtime

P4Runtime es otro componente del ecosistema P4, pero son proyectos separados. P4Runtime se define como una API estándar, abierta e independiente

del hardware, para permitir el control en tiempo de ejecución de los planos de datos P4. Es abierta, es decir que puede ser utilizada para controlar cualquier switch ASIC, extensible, y personalizable (diferentes redes pueden utilizar diferentes protocolos y funcionalidades usando la misma API) [40].

La API P4Runtime permite controlar los elementos del plano de datos de un dispositivo definidos por un programa P4. Para lograr esto, genera un controlador, el cual podrá manejar los dispositivos P4. Este controlador puede, entre otras cosas, manipular las tablas definidas por el programa P4 (creando/-borrando/modificando/consultando entradas), consultar elementos con estado, cambiar la configuración del *pipeline* de reenvío (es decir, cambiar el programa P4 que está ejecutando un switch), todo en tiempo de ejecución.

2.3.3.1. Protobuf y gRPC

La API P4Runtime se encarga de definir los mensajes y la semántica de la interfaz entre el controlador y el *target* que controlará. La definición de la API está hecha a través de archivos *Protocol Buffers* (protobuf), que son un mecanismo extensible, independiente del lenguaje y la plataforma, para serializar datos estructurados. Es desarrollado por Google y es de código abierto. Se define solo una vez cómo se quiere estructurar los datos, y luego se puede utilizar código fuente generado para escribir y leer fácilmente los datos estructurados desde y hacia una variedad de flujos de datos y varios lenguajes. Actualmente soporta código generado en Java, Python, Objective-C y C++. Protobuf fue desarrollado como una alternativa a XML. [16][31][22]

En primer lugar se debe definir la estructura de los datos que se quieren serializar en un archivo *proto* (archivo de texto con extensión `.proto`). Estos datos se estructuran como *messages* (mensajes) donde cada mensaje es un pequeño registro lógico de información que contiene una serie de pares nombre-valor llamados *fields* (campos). Por ejemplo:

```
message Persona {
    string nombre = 1;
    int32 identificador = 2;
    string correo_electronico = 3;
}
```

Una vez especificadas las estructuras de datos, se usa el compilador de protobuf, llamado `protoc`, para generar clases de acceso a los datos en el lenguaje de preferencia. Estos proporcionan accesos simples para cada campo,

como *getters/setters*, así como métodos para serializar/deserializar la estructura completa desde/hacia bytes sin procesar (crudos). Por ejemplo, si se elige el lenguaje C ++, ejecutar el compilador en el ejemplo genera una clase llamada `Persona`. Luego, se puede usar esta clase en una aplicación para poblar, serializar y recuperar mensajes protobuf `Persona`.

gRPC es un *framework* de código abierto de alto rendimiento para *Remote Procedure Call* (RPC) que puede correr en cualquier entorno, desarrollado por Google. gRPC puede usar protocol buffers como su *Interface Definition Language* (IDL) y como su formato de intercambio de mensajes subyacente. En gRPC, una aplicación cliente puede llamar directamente a un método en una aplicación de servidor en una máquina diferente como si fuera local, facilitando la creación de aplicaciones y servicios distribuidos. Como en muchos sistemas RPC, gRPC se basa en definir un servicio, especificando los métodos que pueden ser llamados de manera remota con sus parámetros y tipos de retorno. Estos servicios gRPC se definen en archivos `proto`, con parámetros de método RPC y tipos de retorno especificados como mensajes de protobuf. El servidor implementa esta interfaz y corre un servidor gRPC para manejar las llamadas de los clientes, mientras que el cliente tiene un *stub* (denominado simplemente cliente en algunos idiomas) que provee los mismos métodos que el servidor.

Los clientes y servidores gRPC pueden correr y comunicarse entre sí en una variedad de entornos, desde servidores hasta computadoras personales, y pueden ser escritos en cualquiera de los lenguajes soportados por gRPC. Por ejemplo, se puede crear un servidor gRPC en Java con clientes en Go, Python o Ruby. [2] Se puede ver un ejemplo de diagrama de comunicación entre clientes y servidores gRPC en la Figura 2.18.

La API de P4Runtime es implementada por un programa que ejecuta un servidor gRPC. Este servidor se asocia con una implementación de la interfaz (generada automáticamente) del servicio P4Runtime. Este programa es llamado “P4Runtime Server”. [31]

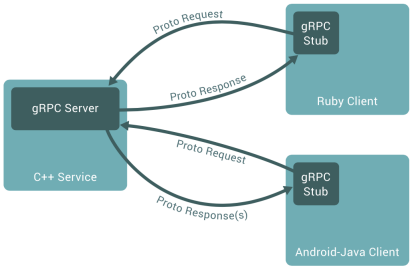


Figura 2.18: Ejemplo de diagrama de comunicación entre clientes y servidores gRPC [2]

2.3.3.2. Arquitectura

En la Figura 2.19 se puede apreciar la arquitectura de referencia de P4Runtime. El dispositivo o *target* a ser controlado se encuentra abajo (en violeta oscuro) y arriba se muestra un controlador. Puede haber más de un controlador, pero P4Runtime solo le da acceso de escritura a un controlador principal para cada entidad, a diferencia del acceso a lectura que lo tiene cualquier controlador. Los controladores tienen un cliente gRPC que llama de manera remota a los métodos provistos por el servidor gRPC que se encuentra en el *target*. Estos métodos son los que permiten el control de los elementos P4. Por ejemplo, para configurar el programa P4 de un *target* el controlador utiliza una RPC llamada `SetForwardingPipelineConfig`.

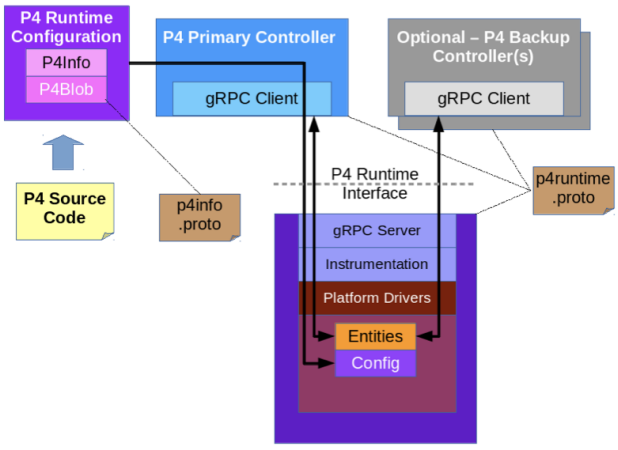


Figura 2.19: Arquitectura de referencia de P4Runtime [31]

El controlador puede acceder a las entidades P4 que están declaradas en los metadatos del archivo `P4Info`. La estructura de este archivo está definida por el archivo `p4info.proto`. El controlador también puede modificar la

configuración del *pipeline* de reenvío, lo cual equivale a instalar y ejecutar la salida del programa P4 compilado e instalar los metadatos de P4Info asociados. Además, el controlador puede consultar al *target* para obtener información sobre la configuración del dispositivo y el P4Info. En resumen, P4Runtime permite no solamente cargar distintos programas P4 en los dispositivos, sino que también permite consultar información de configuración del dispositivo (como las tablas del plano de control), junto con otros objetos (como son los contadores), además de poder modificar las reglas del plano de control, todo esto en tiempo de ejecución. [31]

2.3.3.3. Controladores

Como P4Runtime permite más de un controlador, existen distintos casos de uso involucrando distintos tipos de controladores [31]:

- Único controlador integrado: En la Figura 2.20 se puede ver este caso de uso, donde el *target* (dispositivo programable) tiene un controlador integrado que se comunica con un switch integrado a través de P4Runtime. Este caso podría ser apropiado para un dispositivo integrado que no está diseñado para casos de uso de SDN, ya que no se tiene un controlador remoto.

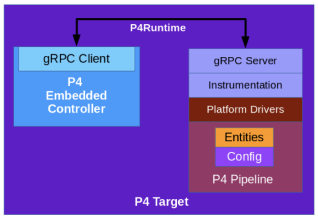


Figura 2.20: Caso de uso de P4Runtime con un solo controlador integrado [31]

- Único controlador remoto: En la Figura 2.21 se puede ver este caso de uso, donde el dispositivo no tiene control sobre el *pipeline*, solo aloja el servidor. Este caso es posible pero es probable que un caso híbrido resulte más práctico.

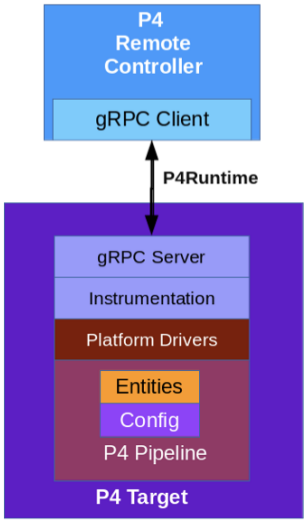


Figura 2.21: Caso de uso de P4Runtime con un solo controlador remoto [31]

- Combinaciones de los anteriores: es posible generar distintas combinaciones de los casos anteriores, como un controlador remoto y un controlador integrado, dos controladores remotos y un controlador integrado, entre otros. En todos los casos los controladores son clientes del mismo (y único) servidor (en el *target*), y se dividen cuál controlador se encarga de cuáles entidades.

Capítulo 3

Evaluación de herramientas

Debido a la popularidad de la programación del plano de datos y en particular del lenguaje P4, existen varios entornos y plataformas, tanto de software como de hardware capaces de ser programables con P4. También existen implementaciones interesantes de diferentes casos de uso desarrollados con este lenguaje. Sin embargo, varios son muy nuevos y carecen de un funcionamiento correcto, documentación apropiada, entre otros. Es por esto que resulta de interés realizar un relevamiento sobre el correcto funcionamiento de algunas herramientas y a qué casos se adaptan mejor. En este capítulo se presentan algunos entornos y *targets* (dispositivos P4) evaluados.

3.1. Controlador ONOS en Mininet con SR

Segment Routing (SR) es una técnica de enrutamiento basada en el origen, que agrega la información del estado de la red en los cabezales de los paquetes. Por esto, SR responde muy bien a cambios en la red, haciéndola más ágil y flexible que otras soluciones [46]. Debido a la adaptación a cambios en el tráfico que posee, resulta interesante probar una implementación de SR en P4. En esta línea, existen tanto una implementación de SR v6 (que es SR sobre IPv6) [30], así como un tutorial [29] de una versión simplificada de SR, que utiliza Mininet [21] para emular la red y ONOS [28] como controlador SDN. Se utilizó una máquina virtual del tutorial actualizado encontrado en [27], a la cual es necesario realizarle instalaciones como Mininet, ONOS, Stratum, luego hacer la conexión Mininet-ONOS, a través de una CLI de ONOS instalar la aplicación de SR, entre otros. A pesar de lograr establecer la conexión Mininet-

ONOS, se encontraron problemas para que Mininet levantara los switches P4, aunque lograba crearlos, por lo que no se logró poner en funcionamiento el ambiente.

3.2. NS4

Para diseñar y probar nuevos protocolos y entidades de red, muchas veces no es posible contar con infraestructuras físicas, ya que las mismas son muy costosas. Generalmente se usan entornos de simulación o emulación para modelar los comportamientos de redes, sin embargo, los emuladores son afectados por aspectos externos a los experimentos, como son los recursos del *host*. Es por esto que muchas veces se usan simuladores para validar y evaluar los diseños, ya que con estos es posible modelar los comportamientos de redes de gran tamaño. Para realizar una simulación de este tipo en un simulador de red tradicional (como lo es ns-3[1]), es necesario seguir los siguientes pasos:

- Desarrollar un modelo de comportamiento, que implemente el diseño del protocolo o entidad de red, como un módulo interno del simulador.
- Configurar la topología de la red y definir las tareas a realizar durante la simulación.
- Desencadenar la simulación para verificar si el modelo de comportamiento se está comportando como debería.
- Una vez verificado el paso anterior, los códigos de simulación deben reescribirse para ser desplegados en *testbeds* o en dispositivos de proveedores.

Durante este proceso, los simuladores de red tradicionales tienen algunos inconvenientes significativos:

- Desarrollar el modelo de comportamiento consume mucho tiempo, es propenso a errores y requiere conocimientos específicos al simulador.
- Los códigos de simulación son difíciles de portar a redes reales, ya que se encuentran muy acoplados al simulador específico. Un ejemplo sería traducir código de un lenguaje de propósito general (como C++) a un lenguaje de descripción de hardware. Esta traducción introduce código redundante y potenciales *bugs*.

[1] https://www.nsnam.org

- Los simuladores de red tradicionales no tienen soporte para el plano de datos programable. Los switches de software existentes que soportan P4 deben ser ejecutados en un emulador de red, que a diferencia de un simulador, se ve limitado por los recursos del *host*, por lo que no puede simular redes P4 a gran escala o ultrarrápidas.

NS4 es un simulador de red basado en P4, que soporta redes P4 a gran escala. Al introducir P4 en ns-3, NS4 desacopla el modelo de comportamiento de las plataformas de simulación subyacentes. Con NS4, los programadores solo necesitan definir comportamientos de procesamiento de paquetes, sin usar ninguna biblioteca específica del simulador. Además, la independencia de *target* de P4 permite que los modelos de comportamiento de NS4 se migren directamente a dispositivos reales habilitados para P4, lo que elimina la reescritura de código redundante. También es una herramienta útil que ayuda a los investigadores de P4 a simular y validar sus trabajos en contextos de red arbitrarios. [3]

Es una herramienta prometedora, que permitiría testear redes de grandes tamaños, sin la necesidad de conocer el simulador. Sin embargo, no se logró poner en funcionamiento la herramienta y tampoco se cuenta con soporte dado que el repositorio no tiene actividad reciente.

3.3. P4-OVS

Open vSwitch (OVS) es un switch virtual, multicapa y de producción. A diferencia de los dispositivos de red tradicionales, ya sean de software o hardware, que logran alto rendimiento a través de la especialización, OVS está diseñado para ser flexible y de uso general. Por ende, debe lograr alto rendimiento sin especialización, adaptándose a las diferentes plataformas soportadas, a la vez que comparte recursos con el hipervisor y sus cargas de trabajo [26][52]. Entonces, OVS es un switch de software de nivel de producción. La combinación de P4 con OVS resultó en P4-OvS [49], un switch de software eficiente programable con P4. P4-OvS se basa en las siguientes tecnologías:

- OVS como *framework* de procesamiento de paquetes
- uBPF como el motor de procesamiento de paquetes reconfigurable
- p4c-ubpf como el compilador que habilita la traducción de programas P4 a código C compatible con BPF

La versión actual de P4-OvS permite:

- Crear un único *bridge* P4 y proveer un programa P4 como parámetro de configuración, o usar P4Runtime
- Agregar puertos al *bridge*
- Recibir tráfico desde estos puertos y manejarlo en el *datapath* P4 (uBPF)
- Controlar las tablas a través de la herramienta `ovs-p4ctl`

La idea de P4-OvS es extender OVS con el soporte para el *datapath* compatible con P4 y la interfaz P4Runtime, pero manteniendo las características conocidas de OVS y el modelo de programación OF. Entonces, la arquitectura de P4-OvS está fuertemente basada en las abstracciones de OVS. En la Figura 3.1 se puede apreciar la arquitectura de P4-OvS.

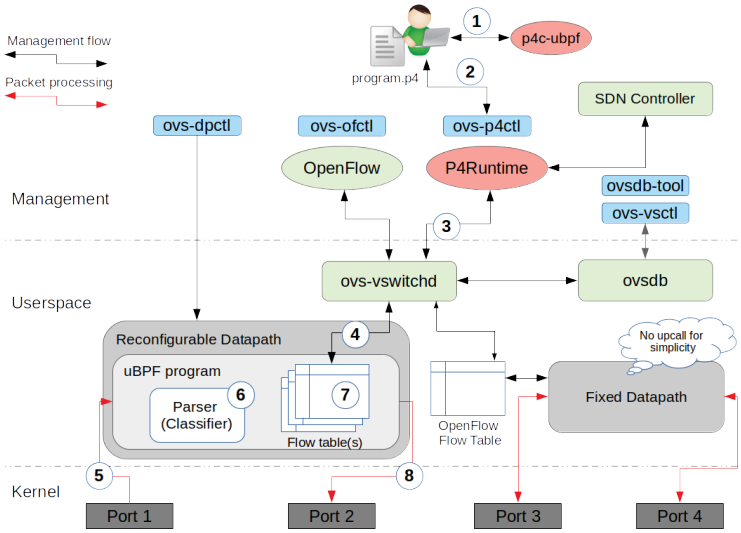


Figura 3.1: Arquitectura P4-OvS [49]

Desde el punto de vista funcional, P4-OvS extiende una base de OVS con cuatro bloques nuevos:

- *Datapath* reconfigurable (con P4): este nuevo componente es responsable de manejar los paquetes entrantes en puertos asociados con el *bridge* P4. Permite inyectar un nuevo *pipeline* de procesamiento de paquetes (generado a partir del programa P4) en tiempo de ejecución. El *datapath* reconfigurable expone interfaces para gestionar el programa P4 y controlar objetos P4 (como tablas, registros, etc.). El *datapath* configurable puede ser eBPF, XDP o uBPF.
- Interfaz P4Runtime: este bloque funcional provee una capa de abstracción entre el *datapath* reconfigurable y los controladores externos. En par-

ticular, implementa un servidor gRPC (con la definición de P4Runtime) y le permite a los usuarios controlar el *datapath* P4.

- Compilador P4: aunque esté implementado por separado (backend del compilador p4c) es parte de la arquitectura funcional. Los usuarios aprovechan el compilador para generar binarios específicos del *datapath* a partir del programa P4 y el archivo de metadatos P4Info para la interfaz P4Runtime.
- ovs-p4ctl: es la utilidad de gestión para el *bridge* P4. Gestiona los programas P4 y controla los objetos P4. Las otras tareas de gestión (por ejemplo, agregar un puerto) son implementadas por otras utilidades de OVS (por ejemplo, ovs-vsctl).

La versión Prueba de Concepto (POC) de P4-OvS implementa el *datapath* uBPF. De todos modos, debido al diseño modular, debería ser posible integrar nuevos *datapath* configurables (como FPGA). [49]

Esta herramienta es de gran valor, ya que permite tener un switch de software eficiente, a diferencia del switch de software de referencia de P4, llamado *Behavioral Model version 2* (BMv2), que no puede ser usado para medir *performance* debido a que no es de nivel de producción. Sin embargo, la documentación no es completa, faltando dependencias y arreglos en el código para lograr la instalación correcta. Una vez logrado, el repositorio cuenta con ejemplos para comprender el funcionamiento de P4-OvS, pero no fue posible reproducir los mismos, ni generar casos nuevos. Se encontraron varios errores, ya reportados, pero sin obtener una respuesta del autor. Por ende, a pesar de que teóricamente es una herramienta de gran utilidad, no fue posible utilizarla.

3.4. P4PI

P4Pi [39] surgió en agosto de 2021, como una plataforma para enseñar e investigar en el área de redes de computadoras. La misma permite diseñar algoritmos de plano de datos escritos en P4 usando una placa Raspberry Pi[1]. A diferencia de los dispositivos de hardware existentes compatibles con P4, que pueden ser caros, esta plataforma es de bajo costo y alta disponibilidad, además de ser libre tanto el software como el hardware.

[1] https://www.raspberrypi.org

P4Pi utiliza la placa Raspberry Pi 4 Model B, que viene con un procesador ARM64 de cuatro núcleos y se puede configurar con 2GB, 4GB u 8GB de RAM. Además, incluye un puerto Ethernet y una red inalámbrica integrada, que permite que el dispositivo actúe como un punto de acceso inalámbrico o un simple switch WiFi. La Figura 3.2 muestra una descripción general de alto nivel de la arquitectura de P4Pi.

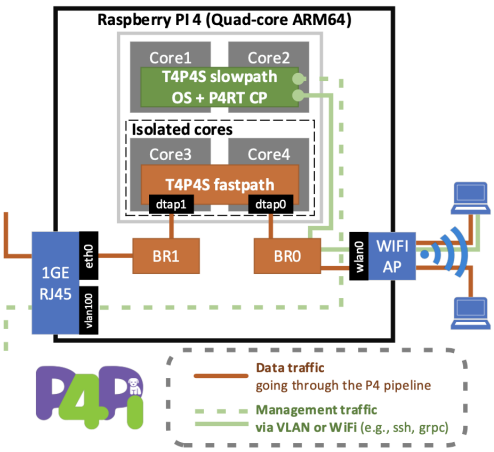


Figura 3.2: Descripción general de la arquitectura P4Pi [39]

La arquitectura se divide en *fast path* y *slow path*. En el *fast path*, dos núcleos de *Central Processing Unit* (CPU) son aislados y dedicados a ejecutar el *pipeline* de procesamiento de paquetes. Estos dos núcleos reciben todo el tráfico que se recibe en los dos puertos, la interfaz Ethernet y la interfaz inalámbrica. Ésta última está configurada en modo de punto de acceso y todo el tráfico recibido en esta interfaz, excepto el tráfico de administración, se conecta a través del *pipeline*. En el *slow path*, los otros dos núcleos restantes son usados para correr el sistema operativo, el servidor de P4Runtime, y la aplicación del plano de control local, en caso de ser necesario.

P4Pi utiliza el compilador de P4 T4P4S, que es un compilador *multitarget* de código abierto, que soporta programas solamente de $P4_{16}$ y las arquitecturas V1Model y PSA. P4Pi usa T4P4S para generar un switch de software basado en DPDK a partir del programa P4, que también incluye un servidor P4Runtime, que habilita aplicaciones del plano de control, como ONOS, a conectarse al switch a través de la API de gRPC. También sería posible utilizar el switch de software BMv2, pero el rendimiento es muy bajo para el uso práctico. El compilador p4c también ofrece un *backend* DPDK alternativo, pero cuan-

do se creó P4Pi, T4P4S proporcionaba un soporte de P4 más completo. Una ventaja adicional de T4P4S es que permite una mayor portabilidad (debido a bibliotecas de abstracción que utiliza) a otras plataformas de hardware, para un usuario que quisiera utilizar su programa P4 en otro dispositivo. En la configuración actual de P4Pi, el programa de switch generado crea dos interfaces virtuales (por ejemplo, TAP, PCAP o KNI) que se conectan con la interfaz inalámbrica o el puerto Ethernet de la placa. Para conectar las interfaces, se usan *bridges* de Linux con la función de aprendizaje de MAC deshabilitada. Esto garantiza que todas las tramas de capa 2 se dirijan al switch P4 y no a la pila de protocolos de red de Linux.

El equipo requerido para utilizar la plataforma es muy reducido, incluyendo la placa Raspberry Pi 4 Model B 4GB (también compatible con el modelo de 8GB), el cargador de la misma y una tarjeta micro SD de 16GB (aunque se recomienda de 32GB). Como equipamiento recomendado, se plantea un conjunto de disipador de calor, cable Ethernet (para la conexión cableada), como los más importantes. Luego, siguiendo los pasos presentados en el repositorio, la instalación es muy fácil y funciona de manera correcta. Una vez conectado a la corriente y encendido, P4Pi aparecerá como un punto de acceso. P4Pi cuenta con una interfaz web, que permite subir un programa P4, compilarlo y ejecutarlo, pero no permite aún modificar el plano de control. A su vez, es posible conectarse a la P4Pi tanto por SSH como utilizando una conexión Ethernet directa. Por defecto, la imagen de P4Pi contiene varios ejemplos, presentados y explicados en el repositorio, además se presenta la guía para crear nuevos programas P4, instalarlos y correrlos. Con el fin de poder modificar el plano de control, P4Pi utiliza P4Runtime-Shell [11], que es un *shell* de Python interactivo para P4Runtime. Esto permite modificar las reglas del plano de control en tiempo de ejecución, entre otros posibles comandos, como leer contadores. [39][51]

La plataforma P4Pi resulta muy interesante, ya que su bajo costo la hace accesible. De todos modos, es una plataforma muy nueva por lo que algunas funcionalidades no están implementadas de manera correcta. Por ejemplo, se tuvo problemas para acceder al tamaño de los paquetes (parte de los metadatos intrínsecos) a partir de registros, lo cual es necesario para desarrollar uno de los casos de uso presentados en Prueba de concepto. La plataforma es muy prometedora, ya que su precio accesible puede permitir desplegar redes de un tamaño considerable con hardware y se encuentra bien documentada.

3.5. BMv2

BMv2 es el switch de software de referencia de P4. El switch está escrito en C++11, y toma como entrada un archivo JSON, generado por un compilador P4 a partir de un programa P4, y lo interpreta para implementar el comportamiento de procesamiento de paquetes especificado por ese programa P4. BMv2 no está destinado a ser un switch de software de nivel de producción, sino que pretende ser usado como una herramienta para desarrollo, testeo y *debugging* de planos de datos P4 y software de plano de control correspondientes. Entonces, el rendimiento de BMv2, en términos de *throughput* y latencia, es significativamente menor que un switch de software de nivel de producción, como es OVS. El compilador p4c incluye un *backend* BMv2, y es el compilador recomendado.

Existen distintas variantes de este *target*:

- `simple_switch`: Este *target* es un switch de software, que se ejecuta en una CPU de propósito general como Intel, AMD, etc., que puede ejecutar la mayoría de los programas $P4_{14}$ y $P4_{16}$, solamente con algunas restricciones en las funciones utilizadas por los programas. Implementa las prestaciones correspondientes a la especificación de $P4_{14}$ (arquitectura única) y también de $P4_{16}$ con la arquitectura V1Model, que está definida para coincidir con la arquitectura de $P4_{14}$. Este *target* puede aceptar conexiones TCP de un controlador, donde el formato de los mensajes de control en esta conexión está definido por la API de Thrift[1], que fue diseñada específicamente para BMv2. El programa P4 cargado en este *target* puede cambiar mientras se está ejecutando, y no es necesario volver a compilar el ejecutable del *target* para usar los nuevos mensajes del plano de control para ese programa P4 nuevo, es decir que la API del plano de control es independiente del programa.
- `simple_switch_grpc`: Este *target* está basado en el *target* `simple_switch`. La diferencia principal es que este puede aceptar conexiones TCP de un controlador, donde el formato de los mensajes de control en esta conexión están definidos por la especificación de la API de P4Runtime.
- `psa_switch`: Es similar a `simple_switch` solo que no usa la arquitectura V1Model, sino que usa la arquitectura PSA.

[1] https://github.com/apache/thrift

- `simple_router` y `l2_switch`: Estos *targets* fueron introducidos para ilustrar cómo se puede aprovechar la biblioteca BMv2 para implementar diferentes arquitecturas. Admiten conjuntos mucho más pequeños de `standard_metadata` y `simple_router` no tiene un *pipeline* de salida. No se recomienda usarlos y no se pueden usar con programas $P4_{16}$.

`simple_switch` además cuenta con una CLI, invocada a través del comando `simple_switch_CLI`, que permite utilizar comandos para examinar/modificar el contenido de las tablas, leer/modificar registros, leer/reiniciar contadores, entre otros. [12]

Este *target* es muy utilizado, ya que cuenta con un tiempo de desarrollo razonable, funciona correctamente, implementa (salvo mínimas excepciones) todas las funcionalidades de P4 y se puede utilizar en conjunto con Mininet para emular redes de switches P4. Además, utiliza la arquitectura V1Model, que también es muy popular. Es importante tener en cuenta que el rendimiento de este *target* es limitado, por lo cual no debe ser usado en pruebas donde medir el mismo sea de interés.

Capítulo 4

Prueba de concepto

Se diseñaron distintos casos de uso con el objetivo de comprender en profundidad el lenguaje P4, junto con las herramientas necesarias para su correcto funcionamiento y puesta a prueba, además de poder demostrar en cierta medida las capacidades del lenguaje. Debido a que el *target* BMv2 es de las pocas herramientas que funcionan de manera correcta, y es de fácil integración con Mininet, permitiendo el desarrollo en redes de tamaños razonables, es que se optó por utilizar esta herramienta para desplegar los casos de uso. Con este fin, se utilizó como entorno de trabajo una máquina virtual brindada para el tutorial oficial del lenguaje P4. Esta máquina virtual cuenta con todas las dependencias y bibliotecas necesarias (como la biblioteca de P4Runtime) para poder desarrollar programas P4, emulando nodos que corran estos programas y redes con Mininet. Se eligió utilizar este entorno ya que cuenta con todas las instalaciones necesarias, y con *scripts* y *makefiles* (brindados para los ejercicios del tutorial) que hacen más fácil la ejecución de los casos de uso. Todo el código desarrollado se encuentra disponible en [8].

4.1. Entorno de trabajo

El entorno de trabajo es una máquina virtual con Ubuntu 16.04, que cuenta con Mininet instalado y Python 2. A su vez, cuenta con las bibliotecas necesarias para poder utilizar P4Runtime a través de programas de Python. Esta máquina virtual fue descargada del repositorio del tutorial oficial de P4 [14], a principios de 2021[1].

[1] https://github.com/p4lang/tutorials/tree/master/vm

Dentro de la máquina virtual (en la ruta `home/p4/tutorials/exercises`) se dispone de varias carpetas, donde cada una contiene los archivos necesarios para completar cada uno de los ejercicios del tutorial. En particular, estos cuentan con:

- varios archivos JSON (generalmente nombrados `sX-runtime.json`) que contienen las reglas de cada tabla del plano de control, para cada switch (indicado por X), según el formato de las tablas definidos por el programa P4. Estos archivos son los encargados de poblar las tablas con valores.
- un archivo JSON (generalmente nombrado `topology.json`) que contiene la topología a emular en Mininet, además de que establece las reglas que tendrá el plano de control de cada switch (indicando para cada switch qué archivo `sX-runtime.json` le corresponde). También contiene comandos necesarios para configurar los hosts.
- un archivo P4, que define los algoritmos de plano de datos que se desean instalar en los switches.
- un archivo `Makefile`, que se encarga de compilar el programa P4, levantar la topología en Mininet configurando todos los switches con el programa P4 apropiado, junto con las reglas de las tablas del plano de control y configurando todos los hosts con los comandos proporcionados.
- un archivo Python (generalmente nombrado `mycontroller.py`) que representa un controlador P4Runtime.

Para desarrollar nuevos casos de uso, se tomaron como base estos archivos.

4.1.1. Plano de control

El programa P4 define el procesamiento de paquetes, pero no define las reglas encontradas en las tablas, sino que esto es tarea del plano de control. Es por esto que en los casos de uso presentados a continuación, se definen las reglas para cada switch, en los archivos `sX-runtime.json`. Las reglas son definidas en base al formato de la tabla, es decir, si la clave definida es una dirección IP, la tabla tendrá para cada IP que elija, las acciones (que el plano de control debe elegir entre las definidas por el programa P4) a ejecutar. También debe proporcionar los valores de los parámetros definidos para cada acción.

4.2. Caso de uso - Monitorización

En este caso de uso, se plantea desarrollar un programa en P4 para monitorizar el tráfico en la red. La idea general consiste en diseñar un programa en P4 que reenvíe los paquetes según el destino (IP) de manera correcta, y que a su vez guarde distintos valores de utilidad, como por ejemplo, la cantidad de paquetes entrantes. Con este fin, se utiliza una topología simple de cuatro hosts y cuatro switches, dispuestos como se puede ver en la Figura 4.1.

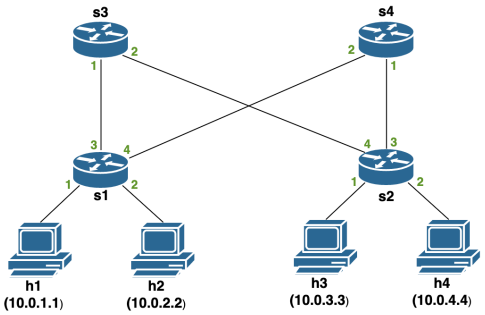


Figura 4.1: Topología simple para el caso de uso de monitorización

4.2.1. Estructura básica del programa P4

Como se mencionó en la sección Arquitectura V1Model, los programas P4 para la arquitectura V1Model cuentan con un *pipeline* de ingreso, el *traffic manager* y el *pipeline* de egreso.

Al igual que en el ejemplo presentado en la sección Un ejemplo: forwarding básico, se definen los *headers* Ethernet e IPv4, en el `parser` se realiza el *extract* de estos *headers* y se tiene el bloque *control* especializado para verificar la suma de comprobación, que se deja vacío.

Luego, se tiene un bloque *control* de ingreso, en el cual se realizará el reenvío y se guardarán datos de interés. Para esto se define una tabla: `ipv4_lpm`. Esta tabla será poblada por el plano de control y es la tabla que se utilizará para hacer el reenvío. Esta tabla es igual a la mostrada en el ejemplo presentado en la sección Un ejemplo: forwarding básico.

Además de definir la tabla y las acciones correspondientes, se definen ciertos contadores y registros, que serán de utilidad para la monitorización de la red. Estos se declaran al principio del bloque con tamaño *MAX_PORTS* (cantidad de puertos del switch más uno). Esto es porque los índices de los registros y contadores comienzan en cero, y los índices deben representar puertos (que

comienzan en uno). Todos los valores son actualizados dentro del `apply`. Los registros y contadores definidos son:

- Contador `pckt_counter_ingress`: este contador es del tipo `packets_and_bytes`, es decir que cuenta tanto de paquetes como de bytes. Con esto se obtiene la cantidad de paquetes y de bytes que han entrado al switch, separados según el puerto por el cual lo hicieron. Este valor es actualizado usando la función `count` para el índice correspondiente al puerto por el cuál ingresó el paquete. Este puerto se obtiene a partir de `standard_metadata.ingress_port`.
- Registro `time_lst_pckt_ingress`: este registro contendrá un *timestamp* en microsegundos de cuándo el último paquete apareció en *ingress* para cada puerto. Este valor se actualiza sobrescribiendo el valor actual del registro con el valor `standard_metadata.ingress_global_timestamp`.
- Registro `time_gap`: Este registro contendrá la diferencia de tiempo entre cuándo se recibió el paquete actual y el último, para cada puerto. Este registro es actualizado de la siguiente manera: primero se guarda el valor `standard_metadata.ingress_global_timestamp` que corresponde al *timestamp* de cuando ingresó el paquete. Luego se lee el registro `time_lst_pckt_ingress` que contendrá el *timestamp* de cuándo se recibió el último paquete (observar que este registro, mencionado anteriormente, debe ser actualizado luego de guardar su valor). Por último se calcula la diferencia, que se hace restando al *timestamp* del paquete recibido actualmente el *timestamp* del anterior recibido y se escribe en el registro.

Por último en este bloque, se chequea si el header IP es válido (con la función externa `isValid`) y si el TTL es mayor a cero (`hdr.ipv4.ttl`), y en tal caso se aplica la tabla `ipv4_lpm` al paquete que está siendo procesado.

Luego, en el *pipeline* egreso, se tiene un *control* para la actualización de la suma de comprobación como el presentado en Un ejemplo: forwarding básico.

Como siguiente bloque del egreso, se tiene un *control* de egreso, donde también se definen contadores y registros que solo pueden ser calculados en el egreso. Todos los registros y contadores se declaran con tamaño *MAX_PORTS* y son actualizados en el `apply` del bloque. Estos registros y contadores son:

- Contador `pckt_counter_egress`: este contador también es del tipo `packets_and_bytes`. Con esto se obtiene la cantidad de paquetes y cantidad de bytes que han salido del switch separados según el puerto por el

cual lo hicieron, que se obtiene a partir de `standard_metadata.egress_port`.

- Registro `time_lst_pckt_egress`: este registro contendrá un *timestamp* en microsegundos de cuando el último paquete comenzó el procesamiento de egreso. Este valor se actualiza sobrescribiendo el valor actual del registro para el índice correspondiente (puerto de egreso) con el valor `standard_metadata.egress_global_timestamp`.
- Registro `time_pckt_pipeline`: este registro contendrá la diferencia de tiempo entre cuando el paquete apareció en en el ingreso y cuando apareció en el egreso. Este registro es actualizado de la siguiente manera: primero se guarda el valor `standard_metadata.ingress_global_timestamp` que corresponde al *timestamp* de cuando llegó el paquete al ingreso. Luego, se lee el valor `standard_metadata.egress_global_timestamp` que corresponde al *timestamp* de cuando llegó el paquete al egreso. Por último se calcula la diferencia, que se hace restando al *timestamp* del egreso el *timestamp* del ingreso y se escribe en el registro.
- Registro `time_queue`: este registro contendrá la suma de los tiempos que cada paquete estuvo en la cola. Es actualizado obteniendo el valor actual del registro, sumándole el tiempo que el paquete actual estuvo en la cola, obtenido a partir de `standard_metadata.deq_timedelta` (tiempo en microsegundos que el paquete estuvo en la cola) y escribiendo el registro nuevamente. Observar que este dato podría ser útil, por ejemplo, para calcular el tiempo promedio que espera un paquete en la cola, haciendo la división de este tiempo entre la cantidad de paquetes. No es posible escribir este valor en un registro, ya que no se permite escribir valores que no se encuentran definidos en tiempo de compilación. Una alternativa sería calcular este promedio y escribirlo en un campo de un *header* del paquete para ser leído por un posible controlador, o calcular promedios desde el controlador de P4Runtime (solo en el caso de contadores) como se verá más adelante.

Por último se tiene el *deparser* donde se hace el *emit* de los *headers* Ethernet e IP.

4.2.2. Diseño de gráficas con P4Runtime

Para mostrar de forma gráfica los valores obtenidos en el programa de monitorización, se utiliza la biblioteca de P4Runtime a través de un programa Python, que actúa como controlador. No es posible leer registros con P4Runtime, ya que todavía no se encuentra implementado, pero si es posible leer contadores. Se plantea diseñar un controlador que lea los contadores que hay en el programa P4 (de ingreso y de egreso) y realice gráficas con estos valores. El usuario podrá elegir si quiere que los valores sean del ingreso o del egreso, si quiere desplegar el contador de bytes, el contador de paquetes, o el tamaño promedio de los paquetes (todos distinguidos por puerto), y por último el switch al cuál se quiere consultar estos valores. En Clases de utilidad de la biblioteca P4Runtime es posible consultar las funcionalidades en profundidad de las clases de la biblioteca de P4Runtime.

4.2.2.1. Estructura básica del programa del controlador

Primero, es necesario parsear los argumentos para elegir los valores mencionados para diseñar las gráficas (que serán input del usuario). Luego, es necesario instanciar un P4Runtime Helper (perteneciente a la biblioteca) que se usa para obtener los identificadores de los objetos P4 (necesarios para la API).

Para acceder a los contadores de los switches, es necesario establecer las conexiones entre el controlador y cada switch (conexión gRPC). Esto se hace instanciando la clase `Bmv2SwitchConnection` que extiende la clase `SwitchConnection`, la cual sirve para establecer las conexiones, y consultar/-modificar los elementos P4. Después de establecidas las conexiones, se envía un mensaje de actualización a cada switch para establecer el controlador como controlador *master*, usando la función llamada `MasterArbitrationUpdate`, sobre la conexión con cada switch. Observar que en este caso no es obligatorio declarar el controlador como *master*, ya que no se realiza ninguna operación de escritura por parte del controlador sobre el switch.

Luego, usando la biblioteca de Python Matplotlib [1] se realizan las gráficas según lo indicado por el usuario. Para leer los valores de los contadores, se usa la función `ReadCounters`, sobre las conexiones con los switches, pasando como parámetro el identificador del contador elegido (obtenido a través del

[1] https://matplotlib.org/stable/index.html

P4Runtime Helper) y el índice del contador a leer (en este caso el número de puerto). Iterando sobre los números de puerto, dentro de las respuestas recibidas del switch es posible leer la cantidad de bytes, la cantidad de paquetes y calcular el promedio de estos, para cada puerto. Al guardar estos datos, luego es muy sencillo realizar las gráficas.

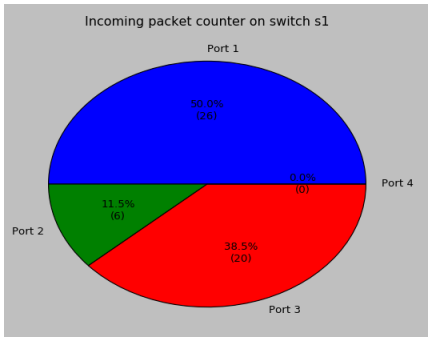
4.2.3. Pruebas realizadas

Para probar que los programas cumplen con lo esperado, alcanza con ver que las gráficas realizadas por el programa en Python (controlador) son correctas y que es posible consultar los valores tanto de los registros como de los contadores.

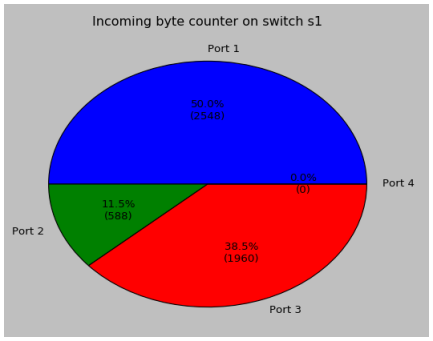
4.2.3.1. Correctitud de las gráficas

En primer lugar, se reinician los contadores en cada switch, ya que hay paquetes previos del sistema operativo y para hacer más claro el análisis resulta más sencillo borrarlos y solo tener los paquetes del tráfico generado para la prueba. Para esto se accede al `simple_switch_CLI` y se ejecuta el comando `counter_reset <nombre_del_contador>`, donde `<nombre_del_contador>` se deberá reemplazar por el contador a reiniciar (en este caso los dos contadores, de ingreso y egreso).

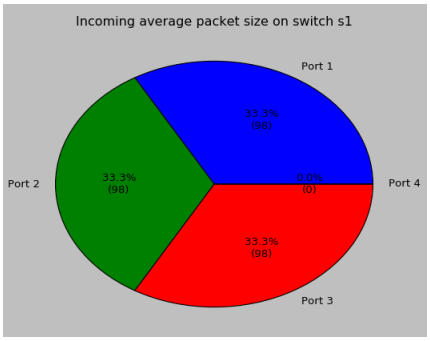
Para verificar las gráficas, se ejecuta un ping desde h1 a h2 con 6 paquetes, un ping desde h1 a h3 con 13 paquetes y un ping desde h1 a h4 con 7 paquetes. Luego de realizar los ping, al ejecutar el programa de Python (controlador) con los parámetros correspondientes, se podrán ver las gráficas deseadas. A modo de ejemplo, se presentan las gráficas para el switch s1, variando los diferentes parámetros posibles. Observar que esto se puede realizar para todos los switches. Las gráficas de las Figuras 4.2a, 4.2b y 4.2c pertenecen al contador de ingreso, presentado los valores de cantidad de paquetes, cantidad de bytes y tamaño promedio de paquetes respectivamente.



(a) Cantidad de paquetes entrantes



(b) Cantidad de bytes entrantes



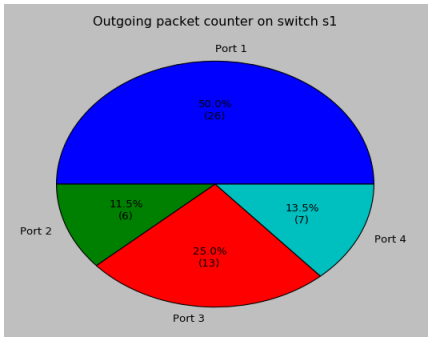
(c) Tamaño promedio de paquetes entrantes

Figura 4.2: Gráficas basadas en el contador de ingreso `pckt_counter_ingress` para el switch s1

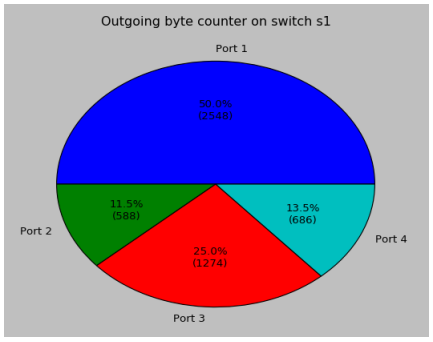
La Figura 4.2a representa los paquetes entrantes en el switch s1, divididos por puerto. En el caso del puerto 1, coincide con los paquetes salientes de todos los ping realizados desde h1 al resto de los hosts, que son la suma de los 6 paquetes del primer ping, 13 del segundo y 7 del tercero. En el caso del puerto 2, coincide con los paquetes de respuesta a los 6 paquetes del primer ping, y en el caso del puerto 3, a los paquetes de respuesta del segundo y tercer ping. También se puede ver que en el puerto 4 no hay ningún paquete entrante, lo cual se debe a las reglas de la tabla de reenvío del plano de control que determinan el camino que tomará cada paquete según el destino. La gráfica también muestra el porcentaje que representan los paquetes recibidos por cada puerto frente al total de paquetes recibidos en el switch. En la Figura 4.2b se puede apreciar la gráfica que representa la cantidad de bytes de paquetes que ingresaron al switch. Teniendo en cuenta que los paquetes que genera el ping tienen un tamaño por defecto de 98 bytes, es fácil ver que los números en la gráfica coinciden con el escenario planteado. Por último, en la Figura 4.2c, se puede ver que para los tres puertos mencionados el tamaño promedio es de 98

bytes, lo cual coincide con que los únicos paquetes que circulan en la red para esta prueba son de los ping realizados.

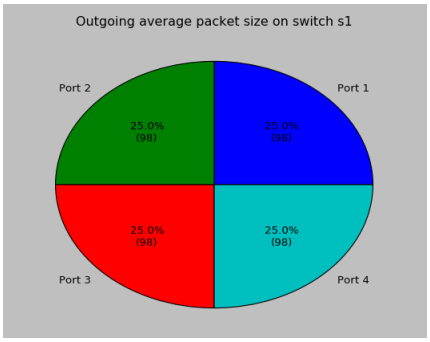
Análogamente, las gráficas de las Figuras 4.3a, 4.3b y 4.3c pertenecen al contador de egreso para el switch s1, presentado los valores de cantidad de paquetes, cantidad de bytes y tamaño promedio de paquetes respectivamente. Observar que en este caso todos los puertos tienen paquetes salientes, lo cual se debe nuevamente a las reglas del plano de control.



(a) Cantidad de paquetes salientes



(b) Cantidad de bytes salientes



(c) Tamaño promedio de paquetes salientes

Figura 4.3: Gráficas basadas en el contador de egreso `pckt_counter_egress` para el switch s1

Como alternativa, en el controlador también se pueden leer estos contadores y simplemente imprimirlos, o realizar otro tipo de gráficas. Por ejemplo, se podría desarrollar un programa que realice constantemente la lectura de estos contadores, cada cierta cantidad de tiempo.

4.2.3.2. Lectura de los valores

Como se mencionó anteriormente, también es necesario poder consultar los valores de los otros registros definidos, así como los contadores planteados

anteriormente. Para lograr esto, basta con acceder al `simple_switch_CLI` del switch, y utilizar los comandos correspondientes. En el caso de los registros, el comando para la lectura es:

```
register_read <nombre_registro> <indice>
```

Donde <nombre_registro> representa el nombre del registro que se desea leer, e <indice> representa el índice del registro al cuál se quiere acceder, en este caso particular sería un puerto. En el caso de los contadores, los comandos para lectura y reinicio son, respectivamente:

```
counter_read <nombre_contador> <indice>
counter_reset <nombre_contador>
```

Donde <nombre_contador> representa el nombre del contador que se desea leer, e <indice> representa el índice del contador al cuál se quiere acceder. A modo ilustrativo, en la Figura 4.4 se puede ver cómo se accede al `simple_switch_CLI` para el switch s1, donde se acceden a todos los registros y contadores para el índice (puerto) 1.

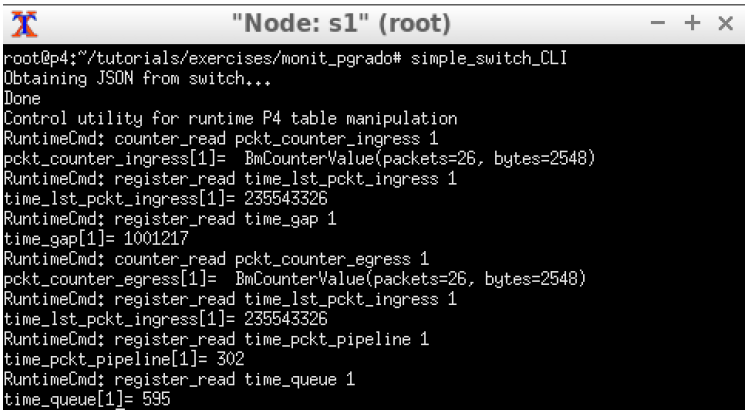


Figura 4.4: Acceso a registros y contadores del switch s1 con índice 1

4.2.4. Conclusiones sobre monitorización

Se diseñó un programa simple, para obtener valores de los paquetes procesados que son de mucho interés, siendo fácil acceder a los mismos y realizar distintas estadísticas con ellos. P4 permite desarmar por completo los paquetes, permitiendo una inspección detallada del tráfico, además de tener medidas dentro de los metadatos intrínsecos de fácil acceso. Es posible, por ejemplo, realizar estadísticas separadas por protocolo, simplemente actualizando los registros o contadores según el protocolo del paquete presente en el cabezal.

P4 junto con P4Runtime, hacen que monitorizar la red resulte sencillo. Escribiendo un programa relativamente corto en P4 y recolectando los datos en un controlador a través de P4Runtime, es posible brindar resultados gráficos y representativos que aporten a una comprensión más fácil y rápida para alguien que necesitara esta información. Cabe destacar que si fuera posible leer los registros, aumentarían las capacidades de P4, ya que se demostró que existen varios valores interesantes que solo se pueden escribir en registros. Además, P4Runtime permite calcular promedios lo cual dentro del programa P4 no es posible.

Por último, resulta de interés comparar la solución brindada con el enfoque tradicional para la monitorización de la red, como lo es utilizar Protocolo Simple de Gestión de Red (SNMP) [9]. SNMP es un protocolo estándar de Internet para administrar dispositivos en redes IP. Los dispositivos que admiten SNMP incluyen routers, switches, servidores, impresoras, entre otros [41]. Una principal diferencia que se puede ver es que la monitorización con P4 no se ve limitada a protocolos existentes, pudiendo monitorizar redes que no utilizan el stack de Internet. Por otro lado, SNMP permite monitorizar aspectos de *hardware*, como son temperaturas, además de funcionar en un amplio rango de dispositivos. En cuanto a los requerimientos para utilizar SNMP para monitorizar la red, es necesario realizar la instalación del servidor SNMP (ejecuta la aplicación que monitoriza los nodos) y el uso de Management Information Base (MIB) [54] (base de datos que contiene definiciones e información sobre las propiedades de los recursos administrados y los servicios que admiten los dispositivos) en los dispositivos [47]. Eventualmente, para obtener datos más personalizados se puede requerir el desarrollo de agregados a la MIB, en caso de que sea modificable el sistema operativo, sino se deberá trabajar con lo implementado por el proveedor. En el caso de P4, solo se requiere el desarrollo del algoritmo para monitorizar los datos deseados.

4.3. Caso de uso - Balanceo de carga

En este caso de uso, se plantea desarrollar dos programas en P4: un balanceador de carga del tipo *Round Robin* (RR) y otro balanceador que decida el destino según el protocolo de capa de transporte (TCP o *User Datagram Protocol* (UDP)). Además, se plantea desarrollar un controlador con P4Runtime que sea capaz de cambiar el programa P4 en tiempo de ejecución, según el

tamaño promedio de los paquetes. Con este fin, se utiliza una topología simple de tres hosts y tres switches, dispuestos en forma de triángulo, como se puede ver en la Figura 4.5.

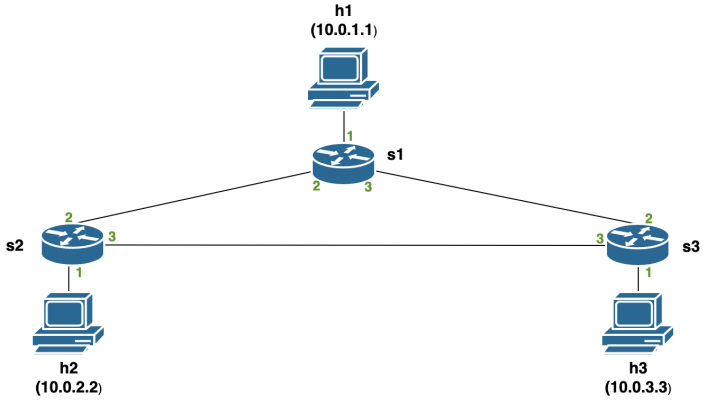


Figura 4.5: Topología simple en forma de triángulo para el caso de uso de balanceo de carga

En el caso del balanceador de carga con RR, se plantea un programa que obtiene, del plano de control, los puertos posibles por los cuales puede salir el paquete del switch, y alterna estos utilizando RR. En el caso del balanceador por protocolo, el programa obtendrá a partir del plano de control por qué puerto deberá salir el paquete, según el protocolo de capa de transporte del mismo y el destino.

4.3.1. Estructura básica de los programas P4

A continuación se describen los programas P4 para ambos balanceadores.

4.3.1.1. Balanceador RR

Al igual que en el caso de uso anterior, se definen los *headers* Ethernet e IPv4. Además, se define un campo del `metadata` (que son los metadatos definidos por el usuario). Este campo se llamará `selected_port` y será del tipo `bit<9>`, que será de utilidad más adelante. En el `parser`, se realiza el parseo de los *headers*, al igual que en el caso de uso de monitorización y también se deja vacío el bloque para verificar la suma de comprobación.

Luego, en el *control* de ingreso se declara un registro, utilizado para guardar según el destino, el último puerto utilizado. Este registro se llama `reg_last_port` (con tamaño MAX_PORTS definido al igual que en el caso de uso de

monitorización). Este registro será la memoria necesaria para implementar el RR, ya que se podrá saber cuál fue el último puerto utilizado para ese destino, para pasar al siguiente posible.

Siguiendo en el bloque se definen dos tablas, `port_group` y `set_nhop`. La tabla `port_group` es utilizada para obtener, según el destino, los puertos por los que es posible que salga el paquete. Por ejemplo, para la topología presentada en la Figura 4.5, un paquete que se debe reenviar desde el switch s1 hacia el switch s2, puede salir por el puerto 2 o por el 3.

Como se desea definir los posibles puertos de salida en base al destino, la clave de la tabla será la dirección de destino del cabezal IP, utilizando el algoritmo LPM y las posibles acciones a tomar serán `drop` (acción por defecto e igual que la presentada en Un ejemplo: forwarding básico), `select_port` y `select_port_host`. Las últimas dos acciones son las que obtienen los posibles puertos de salida del paquete. En el caso de `select_port_host` se trata de obtener el puerto cuando el switch está directamente conectado con el host de destino, por ejemplo, en caso de un paquete saliente del switch s1 hacia el host h1. Esta acción guarda el puerto obtenido (parámetro obtenido del plano de control) en el campo del `metadata` llamado `selected_port` mencionado anteriormente. Esto es necesario para que la otra tabla (`set_nhop`) pueda usarlo como clave.

La acción `select_port` se usa en el resto de los casos, cuando hay un rango posible de puertos salientes para el paquete. Los parámetros de esta acción (proporcionados por el plano de control) serán:

- `bit<9> port_base`: representa el puerto más bajo del rango de puertos por donde puede salir el paquete. Observar que esto asume que los puertos posibles de salida tendrán números contiguos.
- `bit<9> port_max`: representa el puerto más alto del rango anteriormente mencionado.
- `bit<32> register_id`: representa un identificador del destino. Con esto se guarda el último puerto usado para ese destino, y se usa como índice en el registro para acceder al valor.

Tomando como ejemplo el switch s1, una representación de las entradas de la tabla `port_group` se vería como la Figura 4.6. Recordar que es el plano de control quien rellena esta tabla, en base a la clave y acciones definidas por el programa P4.

hdr.ipv4.dstAddr	action	parameters
10.0.1.1/32	select_port_host	port_host: 1
10.0.2.2/32	select_port	port_base: 2, port_max: 3, register_id:1
10.0.3.3/32	select_port	port_base: 2, port_max: 3, register_id:2
default	drop	

Figura 4.6: Representación de las entradas de la tabla `port_group` para el switch s1

Dentro de la acción `select_port`, lo primero a realizar es leer el registro `reg_last_port`, a partir del identificador `register_id`. Según el valor leído se tienen las siguientes opciones:

- vale cero: esto significa que el registro no se usó aún, ya que no tiene ningún puerto guardado (se asume que no hay puerto 0). El puerto de salida en este caso debería ser el primero del rango, es decir `port_base`. También es necesario chequear que el puerto elegido no coincida con el puerto por donde vino el paquete. En caso de coincidir se elige el siguiente puerto del rango.
- coincide con el puerto máximo (parámetro `port_max`): en este caso es necesario "reiniciar" el RR ya que se llegó al tope máximo del rango de puertos posibles. En este caso el puerto a elegir será también `port_base`, volviendo al inicio del rango. Se realiza el mismo chequeo sobre el puerto elegido que el caso anterior, eligiendo el siguiente puerto en caso de ser necesario.
- otro caso: se realiza el RR sin "reiniciar", es decir que se pasa a elegir el siguiente puerto del rango. Nuevamente, es necesario realizar el chequeo del puerto. En caso de coincidir hay dos opciones, si el puerto es el máximo del rango, entonces se selecciona el mínimo, y si no es el máximo, simplemente se elige el siguiente puerto del rango.

Por último, también es necesario guardar el puerto elegido en el campo del `metadata selected_port`.

Luego, se define la tabla `set_nhop`, que es utilizada para hacer el reenvío en base al puerto seleccionado a través de las acciones mencionadas anteriormente. Primero se define la clave de la tabla, que será el puerto seleccionado, es decir `metadata.selected_port`, utilizando el algoritmo *exact*. Se definen también las posibles acciones a tomar, que son `drop` (acción por defecto e igual a todas

las descritas anteriormente) o `forward`. La acción `forward` es casi idéntica a la acción `ipv4_forward` mostrada en Un ejemplo: forwarding básico, a diferencia de que el puerto por donde debe salir el paquete en lugar de obtenerse a partir del plano de control (como un parámetro) se obtiene de `metadata.selected_port`, que es dónde se guardó anteriormente el puerto elegido.

Por último, en el bloque, se chequea si el cabezal IP es válido y si el TTL es mayor a cero, y en tal caso se aplica primero la tabla `port_group` y luego la tabla `set_nhop` al paquete que está siendo procesado.

Luego, en el egreso, se calcula la suma de comprobación en el bloque correspondiente. En el *control* de egreso, se define un contador, llamado `pckts_counter`, de paquetes y bytes. Este contador será de utilidad para realizar el cambio del programa P4 según el tamaño de los paquetes, por lo que deberá estar definido en ambos balanceadores de carga. Por último, en el *deparser* se hace el *emit* de los *headers* Ethernet e IP.

4.3.1.2. Balanceador por protocolo

Al igual que el programa anterior, se definen los *headers*, el campo del `metadata` (`selected_port`), se parsean los *headers* en el *parser* y se deja vacía la verificación de la suma de comprobación En el bloque *control* de ingreso se define una tabla, `port_protocol`, que es utilizada para obtener, según el destino y el protocolo de capa de transporte del paquete, el puerto por el cual debe salir del switch. En este caso la clave está definida por dos campos: la dirección de destino del header IP del paquete, utilizando el algoritmo LPM, y el protocolo del header IP del paquete (`hdr.ipv4.protocol`), con el tipo de coincidencia *exact*. Observar que en caso de no encontrar coincidencia en la tabla con ambas claves y sus respectivos algoritmos, se ejecuta la acción por defecto. Se definen también las acciones que pueden ser tomadas, que en este caso serán `drop` y `forward`. La primera es igual a todas las descritas anteriormente, y la acción `forward` es idéntica a la acción `ipv4_forward` presentada en Un ejemplo: forwarding básico. Por último en el bloque, se chequea si el header IP es válido y si el TTL es mayor a cero, y en tal caso se aplica la tabla al paquete que está siendo procesado.

Todos los bloques de egreso son exactamente iguales a los bloques de egreso del balanceador RR (cálculo de suma de comprobación, *control* de egreso y *deparser*).

4.3.2. Cambio de programa con P4Runtime

Como se ha mencionado, P4Runtime permite tanto cargar un programa P4 en un switch, como escribir las reglas de las tablas del plano de control en tiempo de ejecución. Además es posible leer los valores de los contadores del programa P4.

Con el fin de cambiar el programa P4 que ejecutan los switches, en base al tamaño promedio de los paquetes, se diseña un controlador P4Runtime que accede al valor del contador de paquetes de egreso. Si el tamaño promedio de los paquetes supera un umbral se instala en el switch el programa P4 de balanceador de carga por protocolo, en caso contrario se instala el balanceador de carga con RR.

4.3.2.1. Estructura básica del programa del controlador

Primero se instancian dos P4Runtime Helper, uno para cada programa P4. El siguiente paso es leer los archivos JSON que contienen las reglas del plano de control para cada switch. Como se tienen dos programas P4 con distintos formatos de tablas definidos, se tienen distintas reglas para estos programas, por lo que cada switch tendrá dos archivos JSON. Por ejemplo, se tendrá un JSON para el switch s1 con las reglas para las tablas del balanceador con RR (`port_group` y `set_nhop`) y otro JSON con las reglas para la tabla del balanceador por protocolo (`port_protocol`).

Luego, se deben crear las conexiones entre el controlador y los switches, al igual que en el caso de monitorización. Observar que en este caso es necesario declarar este controlador como *master*, ya que se realizan operaciones de escritura por parte del controlador al switch.

El siguiente paso es chequear, según el programa P4 que esté corriendo en el switch, si es necesario cambiar el mismo. Para cada switch, se lee el valor del contador (presente en ambos programas), usando la función `ReadCounters` e iterando en los índices (puertos). En cada iteración, se lee la cantidad de bytes por puerto y se suma a un total, lo mismo para la cantidad de paquetes. Al terminar el bucle se calcula el tamaño promedio de los paquetes, dividiendo la cantidad de bytes total entre la cantidad de paquetes total.

Una vez obtenido este promedio, se compara con el umbral deseado para decidir que programa deberá utilizar el switch. Por ejemplo, se elige el programa 1 (balanceador de carga con RR) si el tamaño promedio es menor o

igual a 300 bytes, y se elige el programa 2 (balanceador de carga por protocolo) en caso contrario. Primero se chequea que programa tiene actualmente el switch para evitar la carga del programa y de las tablas innecesariamente, y en caso de que el programa deba ser cambiado, se utiliza la función `SetForwardingPipelineConfig`, que se encarga de instalar el programa P4 en el switch. Al cambiar el programa P4, se borran automáticamente todas las reglas de las tablas del plano de control, por lo que es necesario cargarlas nuevamente. Para esto basta con obtener las entradas de los archivos JSON que contienen las reglas para el programa seleccionado. Estas entradas fueron obtenidas al principio del programa, y al iterar sobre ellas basta con utilizar la función `insertTableEntry` para escribir cada regla en su tabla correspondiente. Estas reglas también pueden ser cargadas desde el programa Python (sin utilizar un archivo JSON), utilizando el *shell* de P4Runtime, o a través del `simple_switch_CLI`.

Por último, se establece que el chequeo del contador sobre el tamaño promedio de los paquetes, y posible cambio de programa, se realiza cada 5 segundos a través de la función *sleep*, lo que claramente puede ser modificado.

4.3.3. Pruebas realizadas

4.3.3.1. Balanceo de carga con RR

Para probar que el balanceo de carga con RR funciona correctamente de manera individual, basta con chequear que en cada switch se hace RR en el puerto (dentro de los casos posibles). Para esto, es posible chequear que el registro que guarda el último puerto usado para el destino se actualice, y además verificar a través de capturas de tráfico que los paquetes salgan por los puertos correctos. Si se toma como ejemplo el switch s1, y se realiza un ping desde el host h1 (10.0.1.1) hacia el host h2 (10.0.2.2) de un solo paquete, al leer el registro `reg_last_port` a través del `simple_switch_CLI` del switch s1 con índice 1 (que según la tabla del plano de control es el índice que corresponde para guardar el último puerto usado para el destino 10.0.2.2/32), se podrá ver que el valor es 2 (indicando el puerto 2). Si se vuelve a realizar el mismo ping y la lectura, el puerto es 3, y al hacer lo mismo de nuevo vuelve a ser 2. Esto comprueba que el registro se actualiza de manera correcta, además de que el reenvío se realiza de manera correcta, tanto hacia los switches como a los hosts, ya que el ping se realiza sin problemas. Esto se puede ver en la Figura 4.7,

donde en la izquierda se puede ver una terminal en el host h1, que realiza los ping mencionados anteriormente, y a la derecha se puede ver una terminal en el switch s1, donde se ve como con cada ping se alterna el valor del registro de manera adecuada para el destino (indicado por el índice 1).

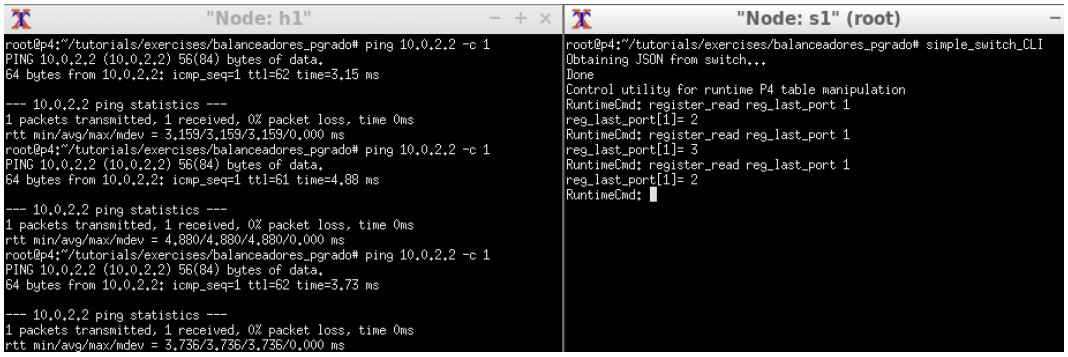


Figura 4.7: Salida de la prueba de actualización del registro que guarda el último puerto usado, según el destino en el balanceo de carga con RR para el switch s1

Para chequear que efectivamente los paquetes salen por los puertos indicados, alcanza con ver capturas de tráfico en los distintos puertos del switch. Al realizar un ping de dos paquetes desde h1 a h2, si se realizan capturas de tráfico en los puertos 2 y 3 del switch s1, se puede ver como un paquete va por un puerto y el otro paquete por el otro, teniendo el mismo destino. Esto se puede ver en la Figura 4.8, donde a la izquierda se puede ver una terminal en el host h1 realizando el ping hacia h2, y a la derecha se puede ver arriba una captura de tráfico en Wireshark del puerto 2 del switch s1, y abajo del puerto 3. Estas pruebas son análogas para todos los switches.

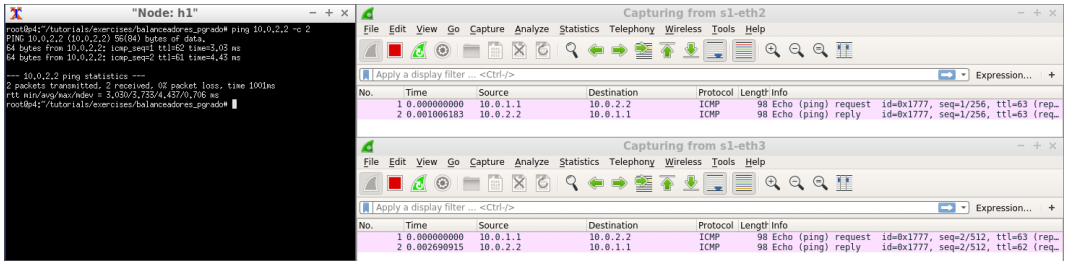


Figura 4.8: Capturas de tráfico de la prueba de puertos en el balanceo de carga con RR para el switch s1

4.3.3.2. Balanceo de carga por protocolo

Para probar que el balanceo de carga por protocolo funciona correctamente de manera individual, basta con chequear que en cada switch se envían los

paquetes, según su protocolo, al puerto indicado en las tablas del plano de control. A modo ilustrativo, en el diagrama mostrado en la Figura 4.9 se puede ver, para cada switch, a qué enlace debe dirigirse el tráfico según si el protocolo es TCP (flechas violetas) o UDP (flechas naranjas). La dirección que deben tomar los paquetes se corresponde con las reglas definidas en el plano de control y por ende pueden ser modificadas.

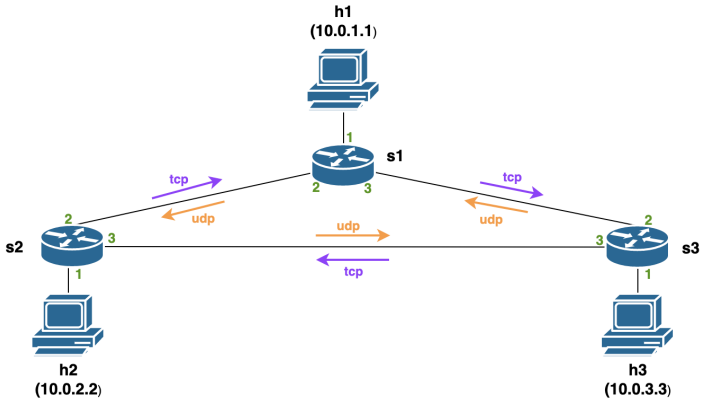


Figura 4.9: Dirección del tráfico según el protocolo para el balanceador de carga por protocolo

Primero se genera tráfico TCP, usando la herramienta *iperf*[1] (herramienta que mide el ancho de banda generando tráfico), entre los hosts h1 y h2. El tráfico TCP desde h2 a h1 debería llegar al switch s1 a través del puerto 2, mientras que el tráfico generado desde h1 a h2 debería salir del puerto 3 del switch s1 (pasando por el switch s3 para llegar a s2 y luego al host 2). Esto se verifica en la Figura 4.10, donde arriba a la izquierda se puede ver una terminal en el host h1, con el servidor *iperf*, abajo a la izquierda una terminal en el host h2 con el cliente *iperf*, arriba a la derecha una captura de tráfico en la interfaz 3 del switch s1, donde se confirma que los paquetes salientes de h1 (10.0.1.1) a h2 (10.0.2.2), salen del switch s1 a través del puerto 3, y por último en la figura, debajo a la derecha, se ve una captura de tráfico en la interfaz 2 del switch s1, donde se confirma que los paquetes salientes de h2 a h1, llegan al switch s1 a través del puerto 2.

[1]https://iperf.fr

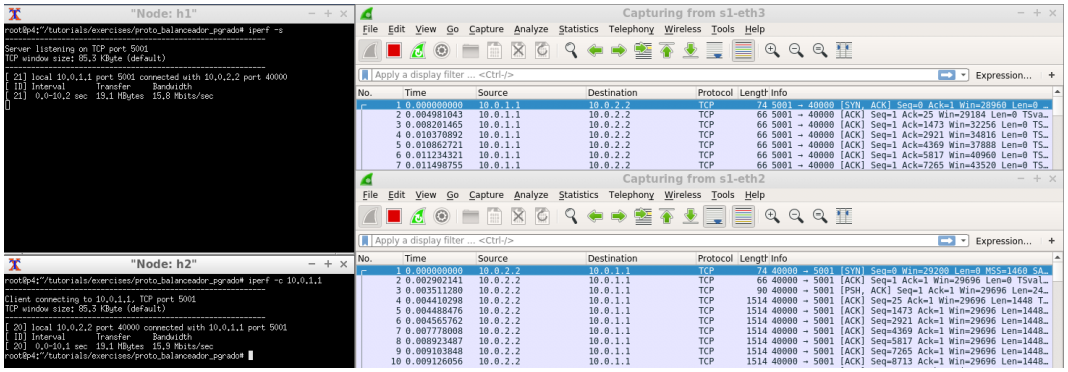


Figura 4.10: Capturas de tráfico de la prueba de puertos en el balanceo de carga por protocolo para el switch s1 con tráfico TCP

Por último, se genera tráfico UDP. El tráfico UDP generado desde h2 a h1 debería llegar al switch s1 a través del puerto 3 (ya que el tráfico debe salir del puerto 3 del switch s2 y luego salir del switch s3 a través del puerto 2), mientras que el tráfico generado desde h1 a h2 debería salir del puerto 2 del switch s1. Esto se verifica en la Figura 4.11, donde arriba a la izquierda se puede ver una terminal en el host h1, con el servidor, abajo a la izquierda una terminal en el host h2 con el cliente, arriba a la derecha una captura de tráfico en la interfaz 3 del switch s1, donde se confirma que los paquetes salientes de h2 (10.0.1.1) a h1 (10.0.2.2), llegan al switch s1 a través del puerto 3, y por último en la figura, debajo a la derecha, se ve una captura de tráfico en la interfaz 2 del switch s1, donde se confirma que los paquetes salientes de h1 a h2, salen del switch s1 a través del puerto 2.

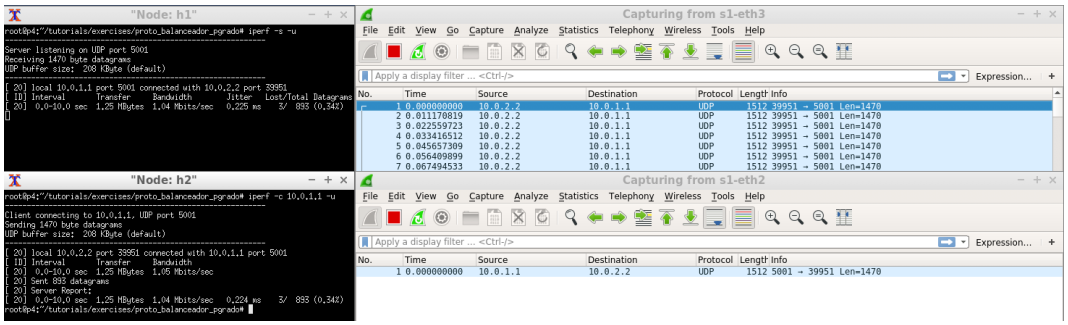


Figura 4.11: Capturas de tráfico de la prueba de puertos en el balanceo de carga por protocolo para el switch s1 con tráfico UDP

4.3.3.3. Cambio de programa

Por último dentro de las pruebas realizadas, es necesario probar que el cambio de programa se hace de manera correcta, además de chequear que se cargan las reglas del plano de control. Para probar esto, basta con generar tráficos con paquetes de distintos tamaños, de manera que el tamaño promedio de los paquetes varíe, haciendo que cambie el programa instalado en los switches. Una vez cambiado este programa, solo resta ejecutar las mismas pruebas mencionadas anteriormente, que confirman qué programa se está ejecutando. Para chequear que las reglas del plano de control efectivamente se cargan, es posible leer las mismas (a través del `simple_switch_CLI` o utilizando P4Runtime), o al chequear que los programas funcionan, ya se está chequeando que las reglas se cargaron ya que en caso contrario no se podrían reenviar los paquetes.

Al comienzo, los switches empiezan con el programa 1 (balanceador RR) instalado. Tomando como ejemplo el umbral de 300 bytes, tal que si el promedio de paquetes es menor (o igual) a 300 bytes el programa instalado deberá ser el programa 1, mientras que si es mayor deberá ser el programa 2 (balanceador por protocolo), si desde h1 se realiza un ping hacia h2, el programa no debería cambiar, ya que los paquetes del ping tienen tamaño 98 bytes. En cambio, si se genera un flujo TCP con la herramienta *iperf* desde h2 y h3 hacia h1, el programa debería cambiar, ya que los paquetes enviados tienen tamaño mayor a 300 bytes en promedio. Esto se puede confirmar en la Figura 4.12, donde arriba a la izquierda se puede ver una terminal en h1, abajo a la izquierda una terminal en h2, abajo a la derecha una terminal en h3 y arriba a la derecha una terminal corriendo el controlador P4Runtime. Primero se ejecuta el ping desde h1 a h2, donde se puede ver en el controlador que el programa no cambia. Cuando se genera el tráfico TCP con *iperf*, se puede ver en la terminal del controlador como se cambia para todos los switches el programa instalado, y se pasa a instalar el programa 2. A su vez, se puede ver que la herramienta sigue funcionando correctamente, por lo cual se instalan las reglas del plano de control.

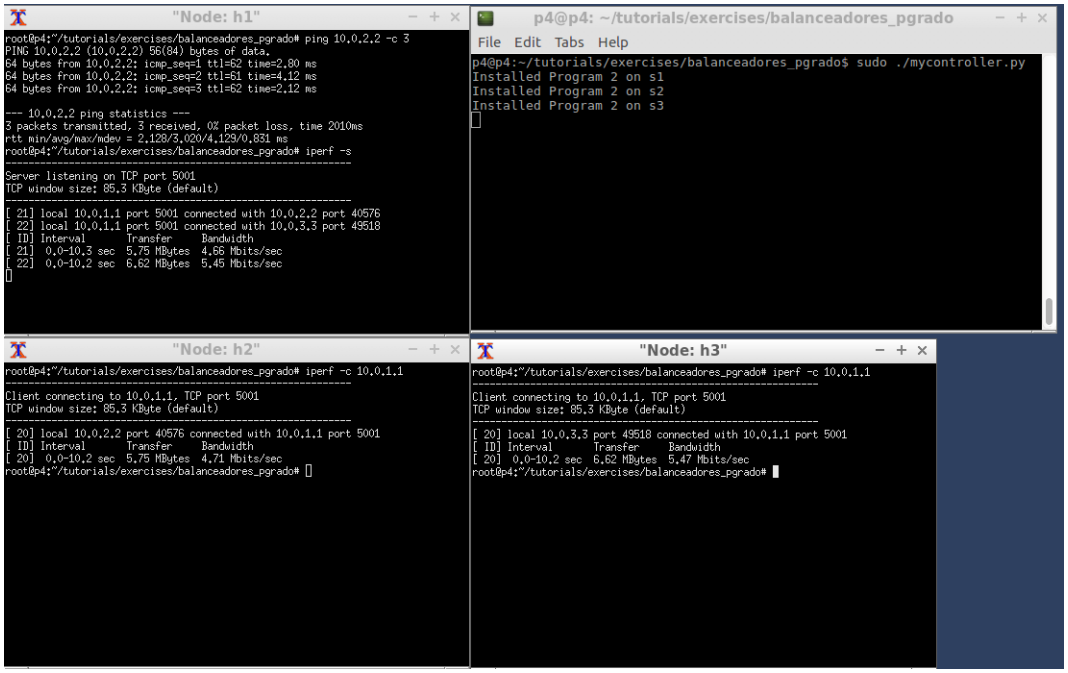


Figura 4.12: Cambio del programa 1 al programa 2 de balanceo de carga en los switches al aumentar el tamaño de los paquetes

Luego, para volver a cambiar el programa P4 en los switches al programa 1, es necesario reducir el tamaño promedio de los paquetes salientes, por lo cual es necesario generar paquetes más pequeños en la red. Generando varios flujos de tráfico con paquetes más pequeños, se logra reducir el promedio y por ende cambiar el programa P4. Esto se puede ver en la Figura 4.13. Observar que el cambio de programa es independiente para cada switch, y en este caso cada switch cambió su programa en tiempos diferentes.

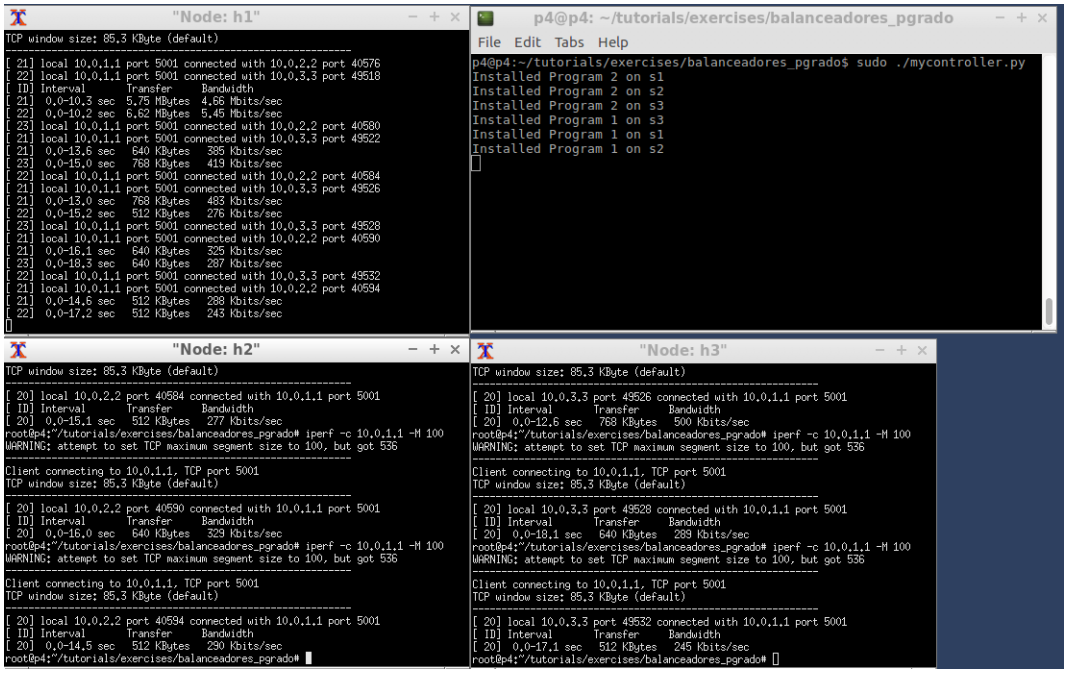


Figura 4.13: Cambio del programa 2 al programa 1 de balanceo de carga en los switches al disminuir el tamaño de los paquetes

4.3.4. Conclusiones sobre balanceo de carga

En primer lugar, se desarrollaron dos programas P4 que balancean el tráfico según diferentes criterios. En el caso de RR, se intentó diseñar un programa lo suficientemente general para que sea posible utilizarlo con diferentes topologías. Esto permite demostrar las grandes capacidades y flexibilidad de P4. Además, este caso permite ver como es posible mantener dentro del switch P4 un estado, teniendo memoria entre paquete y paquete, permitiendo consultar y utilizar este valor para tomar decisiones. También permite ver la variedad de tablas que se pueden diseñar, incluyendo claves a partir de metadatos definidos por el usuario. En el caso del balanceador por protocolo, el programa es un poco más dependiente de la topología y de las reglas del plano de control, por ejemplo, no se tiene en cuenta una solución (como si se tiene en el balanceo con RR) para casos donde un paquete quede circulando por el mismo link infinitamente. Esto muestra que, a pesar de ser responsabilidad del programador hacer buenos diseños, conociendo la topología y pudiendo escribir las reglas del plano de control a través del controlador de P4Runtime, estos problemas se solucionan. Este programa también permite ver otro tipo de tablas (con más de una clave), y cómo se pueden tomar decisiones de reenvío en base a otras atributos del

paquete, como en este caso lo es el protocolo de transporte que utiliza.

Con el controlador de P4Runtime, se pudo probar que es posible cambiar, en tiempo de ejecución, tanto el programa P4, como las reglas del plano de control, mostrando la capacidad de tomar decisiones en tiempo real. Además, estas decisiones pueden basarse en datos de monitorización recolectados con el programa P4, ya sea el tamaño de los paquetes, el tipo de protocolo, etc., todo en tiempo real, y con la facilidad que tiene un programa escrito en Python. Observar que para aprovechar esto al máximo sería necesario implementar la lectura de los registros en P4Runtime.

Por último, resulta de interés comparar la solución brindada con un posible enfoque tradicional. Para personalizar el algoritmo de reenvío, con cualquiera de los dos balanceadores de carga desarrollados, sería necesario implementarlo en el núcleo del sistema operativo, en caso de que sea libre. En otro caso, sería necesario pedirle al fabricante del dispositivo que incorpore estos algoritmos. Claramente, en la solución brindada esto no es necesario, ya que se puede personalizar completamente el algoritmo de plano de datos de manera fácil y sin necesidad de incorporar a los fabricantes. Además, es posible cambiar el algoritmo en tiempo real, en base a datos de monitorización, lo cual no es posible en un switch tradicional.

Capítulo 5

Consideraciones finales

Este proyecto fue desarrollado con el objetivo de conocer el estado del arte sobre la programabilidad de dispositivos de red, tanto investigando sobre modelos y lenguajes, como implementando casos de uso de interés con el fin de demostrar la utilidad de la programabilidad de estos dispositivos.

En primer lugar, fue posible conocer el concepto de la programabilidad de la red y las diferencias con los dispositivos configurables. Con el fin de comprender que elementos conforman la programabilidad de la red, se exploró la programabilidad del plano de control, junto con el análisis de sus capacidades y carencias. Luego se analizó en profundidad la programación del plano de datos, el otro componente de la programabilidad de la red, incluyendo los modelos más populares existentes para expresar sus algoritmos. Con este fin, se introdujo en profundidad el modelo PISA, con sus componentes y su arquitectura.

Se presentó el lenguaje P4, que es el lenguaje para expresar algoritmos del plano de datos más popular en la actualidad. Se indagó sobre qué cosas define P4, tales como los algoritmos e interfaces necesarias para la comunicación entre el plano de datos y de control, permitiendo conocer las diferencias entre un dispositivo programable con P4 y un dispositivo tradicional. También se investigaron las abstracciones, elementos y el modelo de arquitectura que conforman el lenguaje P4, tanto los bloques programables como aquellos que no lo son y necesitan ser importados de manera externa para el funcionamiento de un dispositivo de red.

Se logró comprender el flujo de trabajo para que un dispositivo utilice un código desarrollado en P4, incluyendo la definición de la arquitectura, el

rol de los fabricantes de dispositivos y los programadores, la necesidad de un compilador, y la comunicación entre los planos. Fue posible encontrar varios dispositivos programables P4, tanto de hardware como de software, y conocer el compilador de referencia de P4. Se ahondó en la arquitectura V1Model de P4, que es muy popular, viendo que elementos la conforman. Luego, se presentó P4Runtime, una parte del ecosistema P4 de gran poder. Se vio que herramientas utiliza, sus capacidades, así como su gran utilidad y las diferentes arquitecturas que se pueden presentar con controladores P4Runtime.

Luego, se evaluaron varias herramientas con el fin de encontrar ambientes para desarrollar redes programables con P4. Al tratarse de un concepto, y lenguaje tan reciente, varias de estas herramientas no cuentan con la madurez suficiente, por lo que no funcionan o carecen de funcionalidades importantes. Además, muchas veces se da que la documentación es muy pobre, o que se discontinúa el soporte de tales herramientas, por lo que a pesar invertir un tiempo prolongado intentando arreglarlas, no fue logrado. De todos modos, se encontraron varias herramientas prometedoras por lo que será interesante como trabajo a futuro seguir el progreso de las mismas, y hacer una eventual reevaluación. Por último en la evaluación de herramientas, se encontró un ambiente que funciona muy bien, permitiendo desplegar redes de tamaño razonable y además conteniendo todas las funcionalidades de P4 necesarias.

En base a ese mismo ambiente fue que se logró desplegar dos casos de uso de mucho valor. Además, para poner en marcha estos casos de uso, fue necesario aprender a utilizar y comunicar todas las herramientas, lo cual ayudó a comprender en profundidad como interactúan todos los componentes del ecosistema P4. En cuanto a la prueba de concepto, se logró desarrollar un caso de uso de monitorización de red. Esto es de gran valor, ya que permite la monitorización de la red en tiempo real, obteniendo medidas de utilidad y resultados gráficos representativos, siendo estos datos personalizables. Por otro lado, el caso de uso de balanceo de carga, también es un gran aporte. Fue posible desarrollar dos tipos de balanceo, definiendo distintos tipos de formato de tablas. Primero, se definió un tipo de balanceo más sencillo en cuanto al algoritmo de balanceo, pero se diseñó un programa genérico para que se pueda adaptar a otras topologías, aumentando la dificultad del código P4. El otro tipo de balanceo, no es tan sencillo, ya que se debe tomar la decisión en base a la dirección de destino y al protocolo, pero gracias a P4 y su capacidad de inspección del paquete, resultó en un código muy simple. Además, fue posible

no solamente cambiar el programa que ejecutan los dispositivos de red en tiempo real, sino que tomar esta decisión en base al tráfico entrante y de manera individual para cada switch. Esto demuestra el poder que tiene un controlador P4Runtime junto con dispositivos programables P4. Por último, todo esto prueba que, una vez superada la curva de aprendizaje tanto de P4 como de P4Runtime y los ambientes, los programas son sencillos de escribir.

Como trabajo a futuro, este proyecto contiene varias líneas. En primer lugar, queda como trabajo pendiente seguir investigando nuevas herramientas que no se relevaron, o que vayan surgiendo, con el fin de ir expandiendo el conjunto de herramientas útiles. En esto se incluye tanto dispositivos de hardware como de software. También es de interés seguir las actualizaciones de algunas herramientas ya relevadas que son interesantes pero que necesitan más desarrollo, como es el caso de P4-OvS para obtener un switch de software eficiente, o P4Pi, para poder probar una plataforma de hardware. En cuanto a P4Runtime, se vio su gran utilidad, pero se podría sacar aún más provecho si se lograra la implementación de lectura (y modificación) de los registros desde P4Runtime, aumentando el alcance del caso de uso de monitorización, entre otros.

En resumen, se logró comprender el tema en profundidad, probar herramientas y desplegar casos de uso de interés. Se pudo comprender la utilidad, reconociendo los beneficios y el poder que le brinda la programabilidad de la red a los usuarios, y a la sociedad en general, que podrá sacar provecho en un futuro de herramientas desarrolladas con este enfoque. Se encontró como desventaja que el tema es muy nuevo y le falta madurez, pero tiene como ventaja el potencial para investigar y desarrollar nuevas propuestas.

Bibliografía

[1] Carolyn Jane Anderson y col. "NetKAT: Semantic Foundations for Networks". En: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, págs. 113-126. ISBN: 9781450325448. DOI: 10.1145/2535838.2535862. URL: https://doi.org/10.1145/2535838.2535862.

[2] gRPC Authors. *Introduction to gRPC*. https://grpc.io/docs/what-is-grpc/introduction/. [Online; Accessed: Julio de 2021].

[3] Jiasong Bai y col. "NS4: Enabling Programmable Data Plane Simulation". En: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: Association for Computing Machinery, 2018. ISBN: 9781450356640. DOI: 10.1145/3185467.3185470. URL: https://doi.org/10.1145/3185467.3185470.

[4] Bi, Hao y Wang, Zhao-Hun. "DPDK-based Improvement of Packet Forwarding". En: *ITM Web Conf.* 7 (2016), pág. 01009. DOI: 10.1051/itmconf/20160701009. URL: https://doi.org/10.1051/itmconf/20160701009.

[5] Giuseppe Bianchi y col. "OpenState: Programming Platform-Independent Stateful Openflow Applications inside the Switch". En: *SIGCOMM Comput. Commun. Rev.* 44.2 (abr. de 2014), págs. 44-51. ISSN: 0146-4833. DOI: 10.1145/2602204.2602211. URL: https://doi.org/10.1145/2602204.2602211.

[6] Roberto Bifulco y Gábor Rétvári. "A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems". En: jun. de 2018, págs. 1-7. DOI: 10.1109/HPSR.2018.8850761.

[7] Pat Bosshart y col. “P4: Programming Protocol-Independent Packet Processors”. En: *SIGCOMM Comput. Commun. Rev.* 44.3 (jul. de 2014), págs. 87-95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[8] Belén Brandino. *Dispositivos de red programables*. https://gitlab.com/fing-mina/datacenters/p4. [Online; Accessed: Febrero de 2022].

[9] Jeffrey D. Case y col. *Simple Network Management Protocol (SNMP)*. STD 15. http://www.rfc-editor.org/rfc/rfc1157.txt. RFC Editor, mayo de 1990. URL: http://www.rfc-editor.org/rfc/rfc1157.txt.

[10] P4 Language Consortium. *$P4_{16}$ Language Specification*. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. [Online; Accessed: Enero de 2022].

[11] P4 Language Consortium. *A shell for P4Runtime*. https://github.com/p4lang/p4runtime-shell. [Online; Accessed: Enero de 2022].

[12] P4 Language Consortium. *BEHAVIORAL MODEL (bmv2)*. https://github.com/p4lang/behavioral-model. [Online; Accessed: Enero de 2022].

[13] P4 Language Consortium. *P4 Language Tutorial*. http://bit.ly/p4d2-2018-spring. [Online; Accessed: Enero de 2022].

[14] P4 Language Consortium. *P4 Tutorial*. https://github.com/p4lang/tutorials. [Online; Accessed: Enero de 2022].

[15] P4 Language Consortium. *p4c*. https://github.com/p4lang/p4c. [Online; Accessed: Enero de 2022].

[16] P4 Language Consortium. *P4Runtime Specification*. https://github.com/p4lang/p4runtime. [Online; Accessed: Julio de 2021].

[17] P4 Language Consortium. *The BMv2 Simple Switch target*. https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md. [Online; Accessed: Enero de 2022].

[18] P4 Language Consortium. *The P4 Language Specification*. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf. [Online; Accessed: Enero de 2022].

[19] P4 Language Consortium. *v1model.p4*. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4. [Online; Accessed: Enero de 2022].

[20] The P4 Language Consortium. *P4 Portable NIC Architecture (PNA)*. https://p4.org/p4-spec/docs/PNA.html. [Online; Accessed: Enero de 2022].

[21] Mininet Project Contributors. *Mininet An Instant Virtual Network on your Laptop (or other PC)*. http://mininet.org. [Online; Accessed: Enero de 2022].

[22] Google Developers. *Protocol Buffers*. https://developers.google.com/protocol-buffers. [Online; Accessed: Julio de 2021].

[23] doc.dpdk.org. *Overview*. https://doc.dpdk.org/guides-16.04/prog_guide/overview.html. [Online; Accessed: 29 de junio de 2021].

[24] FD.io. *VPP/What is VPP?* https://bit.ly/3I09TaT. [Online; Accessed: Enero de 2022].

[25] Andy Fingerhut. *demo1-heavily-commented.p4_16.p4*. https://github.com/jafingerhut/p4-guide/blob/master/demo1/demo1-heavily-commented.p4_16.p4. [Online; Accessed: Enero de 2022].

[26] Linux Foundation. *Production Quality, Multilayer Open Virtual Switch*. https://www.openvswitch.org. [Online; Accessed: 17 de junio de 2021].

[27] Open Networking Foundation. *Next-Gen SDN Tutorial (Advanced)*. https://github.com/opennetworkinglab/ngsdn-tutorial/tree/advanced. [Online; Accessed: Enero de 2022].

[28] Open Networking Foundation. *Open Network Operating System*. https://opennetworking.org/onos/. [Online; Accessed: Enero de 2022].

[29] Open Networking Foundation. *P4+ONOS SRv6 Tutorial*. https://github.com/opennetworkinglab/onos-p4-tutorial. [Online; Accessed: Enero de 2022].

[30] Netgroup Research Group. *SRv6 uSID (micro SID) implementation on P4*. https://github.com/netgroup/p4-srv6-usid. [Online; Accessed: Enero de 2022].

[31] The P4.org API Working Group. *P4Runtime Specification*. https://p4lang.github.io/p4runtime/spec/main/P4Runtime-Spec.pdf. [Online; Accessed: Julio de 2021].

[32] The P4.org Architecture Working Group. *$P4_{16}$ Portable Switch Architecture (PSA)*. https://p4.org/p4-spec/docs/PSA.html. [Online; Accessed: Enero de 2022].

[33] Mark Handley y col. "Re-architecting datacenter networks and stacks for low latency and high performance". En: *SIGCOMM* (2017), págs. 29-42.

[34] F. Hauser y col. "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research". En: *ArXiv* abs/2101.10632 (2021).

[35] Stephen Ibanez y col. "The P4->NetFPGA Workflow for Line-Rate Packet Processing". En: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, págs. 1-9. ISBN: 9781450361378. DOI: 10.1145/3289602.3293924. URL: https://doi.org/10.1145/3289602.3293924.

[36] Intel. *Intel® Tofino™ 2*. https://www.intel.es/content/www/es/es/products/network-io/programmable-ethernet-switch/tofino-2-series.html. [Online; Accessed: Enero de 2022].

[37] Eddie Kohler y col. "The Click Modular Router". En: *ACM Trans. Comput. Syst.* 18.3 (ago. de 2000), págs. 263-297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: https://doi.org/10.1145/354871.354874.

[38] James Kurose. *Redes de computadoras*. Ciudad de México: Pearson Educación, 2017. ISBN: 978-84-9035-528-2.

[39] Sándor Laki y col. "P4Pi: P4 on Raspberry Pi for Networking Education". En: *SIGCOMM Comput. Commun. Rev.* 51.3 (jul. de 2021), págs. 17-21. ISSN: 0146-4833. DOI: 10.1145/3477482.3477486. URL: https://doi.org/10.1145/3477482.3477486.

[40] Pilar Manzanares-Lopez, Juan Muñoz-Gea y Josemaria Malgosa. "Passive In-Band Network Telemetry Systems: The Potential of Programmable Data Plane on Network-Wide Telemetry". En: *IEEE Access* PP (ene. de 2021), págs. 1-1. DOI: 10.1109/ACCESS.2021.3055462.

[41] Douglas Mauro y Kevin Schmidt. “Essential SNMP / D.R. Mauro, K.J. Schmidt.” En: (ene. de 2001).

[42] Steven McCanne y Van Jacobson. “The BSD Packet Filter: A New Architecture for User-Level Packet Capture”. En: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX’93. San Diego, California: USENIX Association, 1993, pág. 2.

[43] Nick McKeown y col. “OpenFlow: Enabling Innovation in Campus Networks”. En: *SIGCOMM Comput. Commun. Rev.* 38.2 (mar. de 2008), págs. 69-74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[44] Masoud Moshref y col. “Flow-Level State Transition as a New Switch Primitive for SDN”. En: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, págs. 61-66. ISBN: 9781450329897. DOI: 10.1145/2620728.2620729. URL: https://doi.org/10.1145/2620728.2620729.

[45] NetFPGA. *SimpleSumeSwitch Architecture (v1.2.1 and Earlier)*. https://github.com/NetFPGA/P4-NetFPGA-public/wiki/SimpleSumeSwitch-Architecture-(v1.2.1-and-Earlier). [Online; Accessed: Enero de 2022].

[46] Juniper Networks. *What is segment routing?* https://www.juniper.net/us/en/research-topics/what-is-segment-routing.html. [Online; Accessed: Enero de 2022].

[47] Oracle. *SNMP MIB*. https://docs.oracle.com/cd/E13203_01/tuxedo/tux90/snmpmref/1tmib.htm. [Online; Accessed: Enero de 2022].

[48] Orange. *P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch using P4*. https://github.com/Orange-OpenSource/p4rt-ovs. [Online; Accessed: Enero de 2022].

[49] Tomasz Osiński. *P4-OvS - Bringing the power of P4 to OvS!* https://github.com/osinstom/P4-OvS. [Online; Accessed: Enero de 2022].

[50] Tomasz Osiński. *p4c-ubpf: a New Back-end for the P4 Compiler*. https://opennetworking.org/news-and-events/blog/p4c-ubpf-a-new-back-end-for-the-p4-compiler/. [Online; Accessed: Enero de 2022].

[51] P4 Education Workgroup at p4.org. *P4Pi*. https://github.com/p4lang/p4pi. [Online; Accessed: Enero de 2022].

[52] Ben Pfaff y col. "The Design and Implementation of Open vSwitch". En: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, mayo de 2015, págs. 117-130. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff.

[53] Salvatore Pontarelli y col. "Flowblaze: Stateful Packet Processing in Hardware". En: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI'19. Boston, MA, USA: USENIX Association, 2019, págs. 531-547. ISBN: 9781931971492.

[54] R. Presuhn. *Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)*. STD 62. http://www.rfc-editor.org/rfc/rfc3418.txt. RFC Editor, dic. de 2002. URL: http://www.rfc-editor.org/rfc/rfc3418.txt.

[55] Y. Rekhter, T. Li y S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. http://www.rfc-editor.org/rfc/rfc4271.txt. RFC Editor, ene. de 2006. URL: http://www.rfc-editor.org/rfc/rfc4271.txt.

[56] Muhammad Shahbaz y col. "PISCES: A Programmable, Protocol-Independent Software Switch". En: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, págs. 525-538. ISBN: 9781450341936. DOI: 10.1145/2934872.2934886. URL: https://doi.org/10.1145/2934872.2934886.

[57] Anirudh Sivaraman y col. "Packet Transactions: High-Level Programming for Line-Rate Switches". En: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, págs. 15-28. ISBN: 9781450341936. DOI: 10.1145/2934872.2934900. URL: https://doi.org/10.1145/2934872.2934900.

[58] Haoyu Song. “Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane”. En: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: Association for Computing Machinery, 2013, págs. 127-132. ISBN: 9781450321785. DOI: 10.1145/2491185.2491190. URL: https://doi.org/10.1145/2491185.2491190.

[59] SPAN. *BESS Berkeley Extensible Software Switch*. http://span.cs.berkeley.edu/bess.html. [Online; Accessed: Enero de 2022].

[60] MICHELE DI STEFANO. *BERKELEY PACKET FILTER: theory, practice and perspectives*. https://amslaurea.unibo.it/19622/1/berkeleypacketfilter_distefano.pdf. [Online; Accessed: 25 de junio de 2021].

[61] Laurent Vanbever. *Advanced Topics in Communication Networks Programming Network Data Planes*. https://adv-net.ethz.ch/pdfs/03_stateful.pdf. [Online; Accessed: Enero de 2022].

[62] Jonathan Vestin. *SDN-Enabled Resiliency, Monitoring and Control in Computer Networks*. Karlstad: Karlstads universitet, 2020. ISBN: 978-91-7867-075-8.

[63] Marcos Vieira y col. “Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications”. En: *ACM Computing Surveys (CSUR)* 53 (feb. de 2020), págs. 1-36. DOI: 10.1145/3371038.

[64] VMware. *p4c-xdp*. https://github.com/vmware/p4c-xdp. [Online; Accessed: Enero de 2022].

[65] Andrej Yemelianov. *Introduction to DPDK: Architecture and Principles*. https://blog.selectel.com/introduction-dpdk-architecture-principles/. [Online; Accessed: 29 de junio de 2021].

APÉNDICES

0.1. Clases de utilidad de la biblioteca P4Runtime

En la biblioteca de P4Runtime, se pueden encontrar las siguientes clases de Python (entre otras) que pueden ser de utilidad [14]:

- Clase `P4InfoHelper`: es usada para parsear los archivos `P4Info`. Proporciona métodos de traducción del nombre de la entidad desde/hacia el número de identificación (identificadores de los switches, acciones, etc.).
- Clase `SwitchConnection`: toma el código del cliente gRPC y establece conexiones con los switches. Además provee métodos auxiliares que construyen los mensajes protobuf de P4Runtime y realiza las llamadas de servicio gRPC de P4Runtime. Con estos métodos se pueden realizar las conexiones de switches con el controlador, establecer el controlador como *master*, setear la configuración de pipeline de reenvío (es decir el programa P4), escribir/leer entradas de tablas, leer contadores, entre otros.
- Clase `Bmv2SwitchConnection`: extiende la clase `SwitchConnection` y proporciona la carga útil del dispositivo específico BMv2 para cargar el programa P4.
- El archivo `convert.py`: proporciona métodos para codificar y decodificar desde *strings* y números amigables hacia *byte strings* requeridos para los mensajes de protobuf. Usada por la clase `P4InfoHelper`
- El archivo `simple_controller.py`: proporciona métodos para verificar las configuraciones de los switches (tablas), para insertar entradas en las tablas de los switches, para deserializar un archivo JSON en un objeto Python, entre otros.