

Programación de dispositivos del plano de datos con P4

Parte 1

Índice

1	Las redes definidas por software	3
1.1	Funcionamiento	3
1.2	El plano de control	4
1.3	El plano de datos	5
1.4	La arquitectura PISA	5
1.5	El Lenguaje P4	7
1.6	Ciclo de desarrollo con P4	8
2	Laboratorio	10
2.1	Paso 1. Instalación de Docker	10
2.2	Paso 2. Creación de plantillas de dispositivos	10
2.3	Paso 3. Creación y configuración de dispositivos	13
2.4	Paso 4. Configuración de carpetas de trabajo	13
2.5	Paso 5. Escritura de los programas P4	14
2.6	Paso 6. Compilación del programa P4	16
2.7	Paso 7. Despliegue del programa	16
2.8	Paso 8: Exportación/importación de proyectos en GNS3	18
3	Mejoras para el proyecto	19
3.1	Implementación de un reflector de tramas Ethernet	19
3.2	Implementación de un repetidor con tablas match-action	19
4	Referencias	20

1 Las redes definidas por software

En las redes tradicionales, a pesar de su adopción generalizada, los dispositivos de red integran en una misma unidad indivisible el plano de control, que es el que decide cómo manejar el tráfico a nivel general, y el plano de datos, que es el que reenvía el tráfico entre los puertos de un dispositivo. Bajo este modelo de funcionamiento tradicional se reduce de forma significativa la flexibilidad en la configuración y funcionamiento de las redes.

Además, la industria asociada a estos dispositivos de red tradicionales ha estado dominada por soluciones cerradas y propietarias (basadas en el uso de ASICs específicos), por lo que su ciclo de diseño se ha caracterizado por ser lento y cerrado tomando habitualmente años para la evolución e innovación, lo que contrasta drásticamente con la flexibilidad de la industria de desarrollo software.

Las redes definidas por software (SDN) vienen a salvar estas limitaciones de las redes tradicionales a través de la implementación separada del plano de control y el plano de datos. Esta desagregación permite, entre otras cosas, que los dispositivos de red tradicionales se conviertan en simples dispositivos de reenvío entre puertos y que la lógica de control se implemente en un nodo controlador centralizado y programable (basado en software).

De esta forma, se incrementa la flexibilidad de las redes al dividir el problema de control de la red en partes independientes y más manejables e introduciendo nuevas abstracciones que facilitan el desarrollo y evolución de nuevos servicios de red.

1.1 Funcionamiento

La arquitectura de una red definida por software (SDN) está formada por un conjunto de elementos, interfaces y protocolos que se separan en dos planos: el de control y el de datos (como se muestra en la siguiente figura).

El plano de datos es el responsable de tratar directamente con la mayor parte de los paquetes que transitan por la red y que atraviesan sus conmutadores (nombre con el que se designan a los dispositivos del plano de datos en una SDN).

Estos conmutadores se encargan de la recepción, modificación y/o transmisión de paquetes de acuerdo con el contenido de sus tablas de flujo que se ha establecido de acuerdo con el tratamiento deseado. Estas tablas contienen entradas en la que se establece un criterio de selección de paquetes y una acción a realizar en el supuesto de coincidencia (match).

Por otra parte, el plano de control es el encargado de mantener el contenido de las tablas de flujo en los conmutadores, haciendo uso de los protocolos de control necesarios, así como de mantener de forma centralizada la información de configuración y monitorización de la red. Para ampliar y/o hacer uso de forma programática de las funciones/servicios de la red es habitual que sobre los controladores sea posible la incorporación de aplicaciones de red específicas haciendo uso de las correspondientes API.

Bajo este paradigma la inteligencia de la red se traslada al controlador y los conmutadores únicamente se especializan en el reenvío de paquetes de acuerdo con la información existente en sus tablas de flujo.

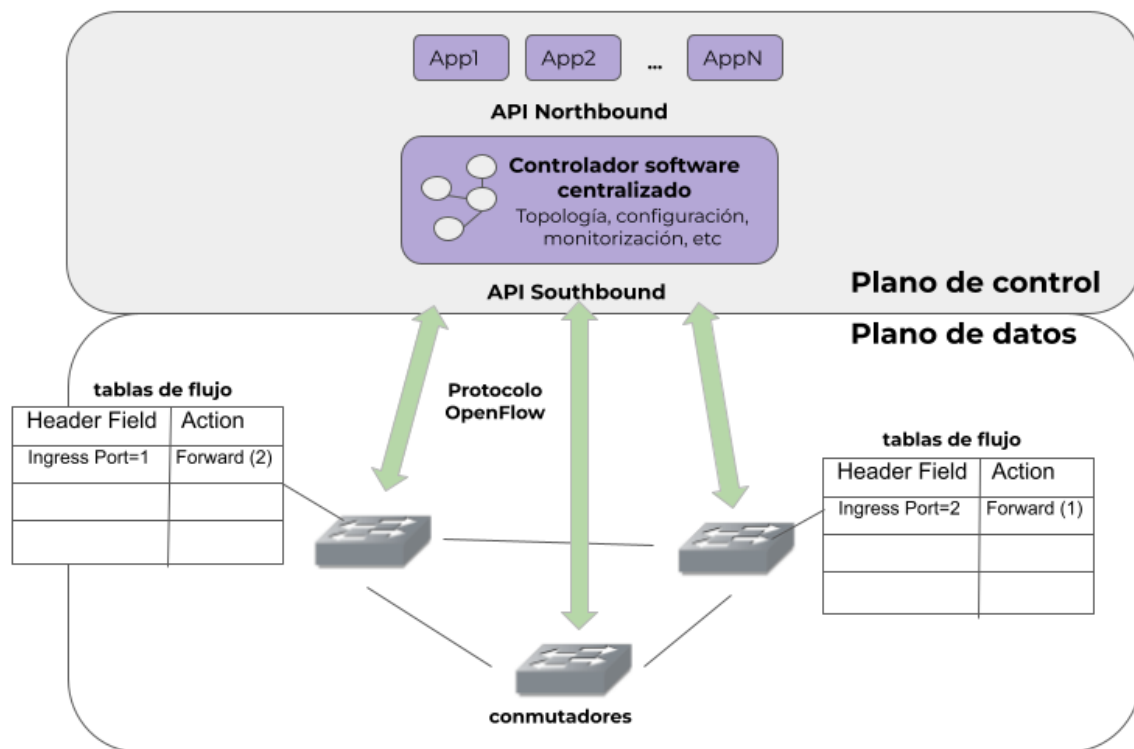


Figura 1. Arquitectura de una red definida por software (SDN)

1.2 El plano de control

Se implementa principalmente a partir de un nodo controlador programable, que es un servidor al uso, que se conecta a cada uno de los conmutadores (a través del protocolo de control correspondiente, habitualmente a través de OpenFlow) sobre una red de gestión (normalmente fuera de banda, es decir a través de una red separada/diferenciada) para realizar el intercambio de mensajes que permita la manipulación en el conmutador de las entradas de su tabla de flujo y de los paquetes que le atraviesan.

Cuando un conmutador no encuentra una coincidencia para un paquete en su tabla de flujo, éste se envía al controlador, para que tome una decisión sobre su tratamiento aplicando la inteligencia que tenga programada y en respuesta puede manipular las entradas de la tabla de flujo del controlador con el objetivo de que los paquetes que se reciban en el futuro y que coincidan con el mismo patrón se puedan tratar directamente en el conmutador sin tener que pasar de nuevo por el controlador (por motivos de eficiencia estas entradas pueden ser temporales).

Los elementos/componentes habituales de un nodo controlador SDN son los que se detallan a continuación (y que se muestran en la figura anterior):

- Una API hacia el norte (Northbound) para la comunicación con las aplicaciones que se despliegan sobre el controlador con el objetivo de ampliar la funcionalidad (personalizadas o predefinidas) de los servicios de red.
- Un conjunto de funcionalidades core:
 - Detección de dispositivos de usuario y de red.

- Gestión de la topología de la red: mantener información sobre los detalles de interconexión de dispositivos de red.
- Gestión de flujos: mantiene una base de datos de los flujos que gestiona el controlador y realizar toda la coordinación necesaria con los dispositivos.
- Monitorización de estado y tráfico de la red.
- Un conjunto de aplicaciones predefinidas que se pueden instalar opcionalmente con el objetivo de ampliar de forma rápida y/o mejorar sus prestaciones: soporte de una interfaz gráfica, implementación de enrutamiento básico, implementación de conmutación a nivel 2 básica, etc.
- Una API hacia el sur (Southbound) que se utiliza como interfaz de comunicación con los conmutadores que gestiona. Con frecuencia esta API se soporta sobre el protocolo OpenFlow.

1.3 El plano de datos

Se implementa principalmente a través de conmutadores hardware o software, y como en las primeras SDN no existía la capacidad de crear un nuevo protocolo y poder trabajar con sus encabezados en el plano de datos, ya que el mismo estaba limitado a las especificaciones de los protocolos de comunicación con el plano de control (protocolo OpenFlow) y a unas funciones fijas del plano de datos, con el tiempo se ha incorporado la capacidad de su programación como parte de la evolución necesaria y natural de las redes definidas por software (SDN).

Los conmutadores modernos son programables de forma que la definición sobre cómo deben procesar los paquetes se realiza aplicando las buenas prácticas de la ingeniería del software; codificando, probando, e implementando nuevas funcionalidades en ciclos de desarrollo mucho más cortos y en los que pueden participar perfiles muy diversos, como operadores, ingenieros, investigadores, etc., lo que facilita la innovación y la evolución hacia nuevos y mejores protocolos/servicios de red. La existencia de dispositivos con capacidades de programación a través de arquitecturas y lenguajes abiertos ha eliminado las barreras de entrada al diseño de redes, previamente reservada a los grandes fabricantes y/o proveedores de soluciones de networking.

1.4 La arquitectura PISA

Es un modelo estándar y abierto para el procesamiento de paquetes en el plano de datos. La arquitectura de conmutación independiente del protocolo (PISA) es independiente de los protocolos y se caracteriza por incluir un conjunto de abstracciones de elementos programables genéricos: un parser de paquetes, un pipeline de procesamiento basado en tablas secuenciales de match-action y un deparser de paquetes (estos elementos se muestran representados en la siguiente figura).

El parser permite definir los encabezados a tratar (según protocolos personalizados o estándar) y proceder a su extracción de los paquetes recibidos. El parser desencapsula los encabezados, convirtiendo el paquete original en una representación troceada. El programador declara los encabezados que deben ser reconocidos y su orden en el paquete. El parser se representa como una máquina de estado sin bucles (gráfico acíclico directo o DAG), con un estado inicial (start) y dos estados finales (aceptar o rechazar).

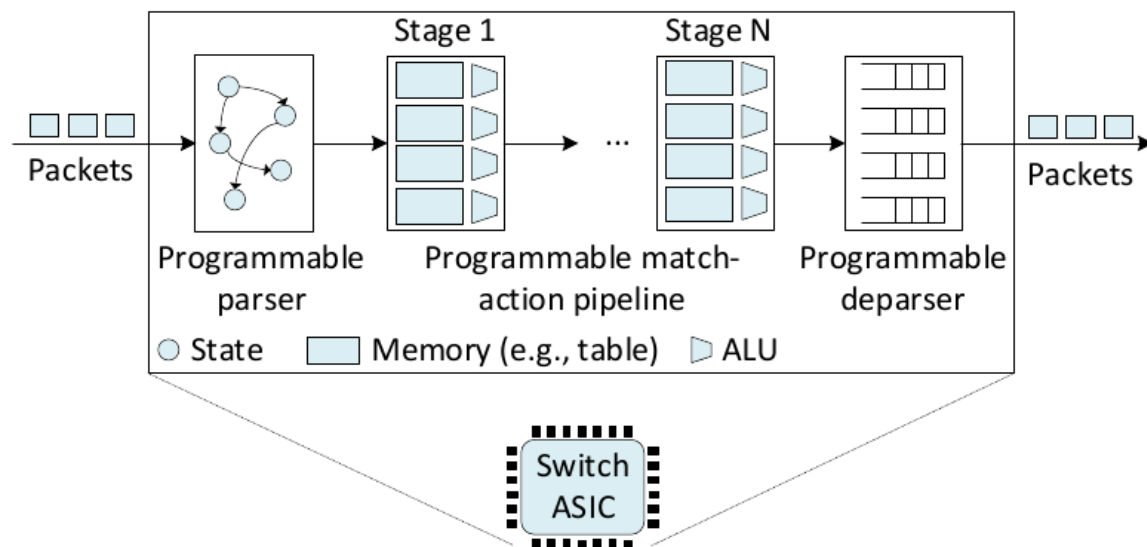


Figura 2. Arquitectura de conmutación independiente del protocolo (PISA)

El pipeline de tablas de match-action sirve para ejecutar operaciones sobre los encabezados seleccionados o sobre resultados intermedios. Este pipeline se divide en etapas, y cada una tiene múltiples bloques de memoria (tablas y registros) y unidades lógicas aritméticas (ALU) que permiten búsquedas y acciones simultáneas. Estas etapas se organizan de forma secuencial para soportar el procesamiento de datos dependientes. En la práctica incorporan entre 10/15 etapas entre la entrada y la salida.

Cada una de las etapas del pipeline de procesamiento puede contener múltiples tablas de match-action (por ejemplo 4 tablas en la figura anterior). Durante el procesamiento hay una fase inicial de match en la que se utiliza una de las tablas para buscar coincidencias por un encabezado del paquete entrante con sus entradas (p.e. por IP de destino). Como hay varias tablas, el conmutador puede realizar búsquedas en paralelo sobre diferentes encabezados. Una vez que se produce una coincidencia (match), la ALU realiza la acción definida (p.e.: modificar el encabezado, reenviar el paquete a un puerto de salida, descartar, etc).

El deparser se encarga de ensamblar de nuevo el paquete de salida combinando los encabezados procesados con la carga útil original y serializando el resultado para su transmisión por la red.

Este modelo de abstracción general, con las adaptaciones correspondientes, es el que se implementa en los conmutadores programables. En concreto, a partir de ahora, vamos a trabajar con la arquitectura VModel, ya que es una simplificación del modelo genérico PISA, y es una de las arquitecturas implementadas en el conmutador software programable que vamos a utilizar a nivel práctico.

La arquitectura VModel se compone de un parser, un pipeline de tablas de match-action de ingreso, un administrador de tráfico, un pipeline de tablas de match-action de egreso y un deparser. El único elemento nuevo es el administrador de tráfico (traffic manager) que se encarga de distribuir los paquetes entre los puertos de entrada y de salida y de realiza la replicación de paquetes cuando sea necesario (p.e.: multidifusión).

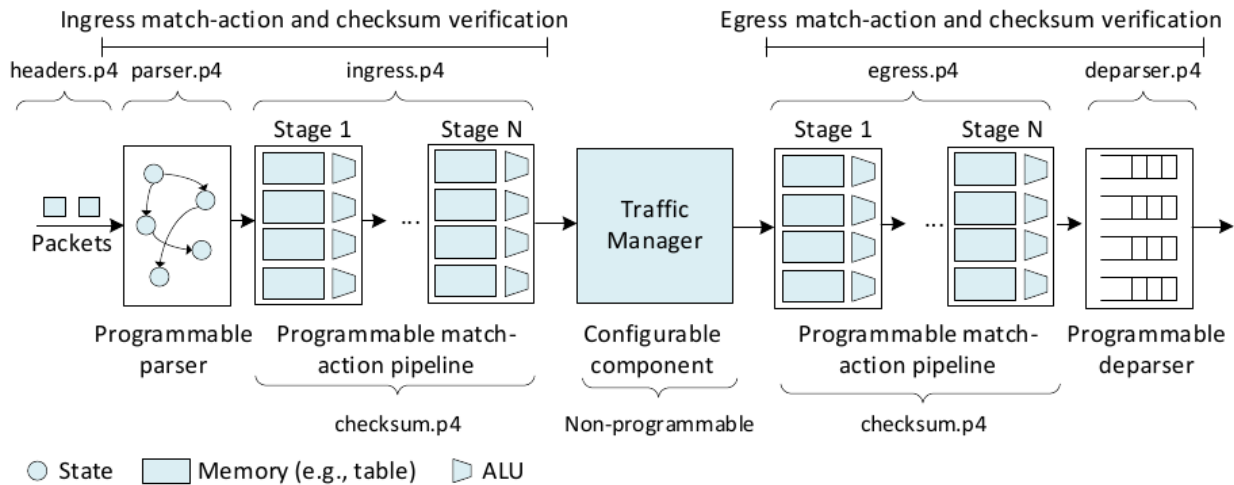


Figura 3. Arquitectura VModel

1.5 El Lenguaje P4

El estándar de facto para definir de forma programática el comportamiento de los conmutadores del plano de datos en las redes definidas por software es el lenguaje para la programación del procesamiento de paquetes independiente del protocolo o P4 [\[1\]](#) (*Protocol-independent Packet Processors*). Se trata de un lenguaje específico de dominio (DSL) diseñado para definir de forma programática el flujo de procesamiento de paquetes dentro de un conmutador (plano de datos). El uso de conmutadores programables con P4 permite incorporar las siguientes capacidades:

- Definición y análisis de nuevos protocolos
- Personalización de las funciones de procesamiento de paquetes
- Medir eventos que ocurren en el plano de datos con una resolución de nanosegundos
- Inspeccionar y analizar cada paquete (análisis por paquete), etcétera.

Un programa escrito en P4 se encarga principalmente de definir el formato de la tabla o tablas a emplear y las claves utilizadas para las operaciones de búsqueda que se van a realizar a partir de los encabezados de los paquetes. De forma complementaria, a lo especificado en el programa de P4, desde el plano de control se deberán poblar las entradas necesarias en las tablas, con las claves y los datos de acción requeridos para conseguir la funcionalidad deseada.

Un programa P4 puede implementar distintos modelos y/o arquitecturas de procesamiento, en nuestro caso vamos a realizar programas que implementen la arquitectura VModel, por tanto, tendrán los siguientes bloques:

- **headers:** contiene los encabezados y los metadatos a usar.
- **parser:** contiene el parser programable para extraer los encabezados a tratar.
- **ingress:** contiene el bloque de control de ingreso con las tablas de mach-action necesarias.

- egress: contiene el bloque de control de salida con las tablas de match-action necesarias.
- deparser: contiene la lógica deparser para el ensamblaje del paquete de salida.
- checksum: contiene el código que calcula y comprueba las sumas de verificación del paquete.

1.6 Ciclo de desarrollo con P4

La programación de un dispositivo del plano de datos (conmutador) con P4, ya sea hardware o software, requiere un entorno específico con los componentes genéricos que se muestran en la siguiente figura y que como mínimo debe incluir un compilador y un modelo de arquitectura concreta para mapear el código fuente escrito en P4 (programa P4), conceptualmente independiente del equipo en el que se va a ejecutar, a los artefactos necesarios por la plataforma en la que se va a ejecutar el programa. El modelo de arquitectura y el conmutador de destino son específicos de cada proveedor y deben ser proporcionados por el mismo.

El compilador de P4 producirá habitualmente dos artefactos a partir del código original. En primer lugar, se genera una configuración para el plano de datos que implementa lo especificado en el programa de entrada P4 (runtime del plano de datos). Esta configuración incluye las instrucciones y las asignaciones de recursos concretas para la plataforma destino específica. En segundo lugar, se genera una API para el tiempo de ejecución que será utilizada por el plano de control para interactuar con el plano de datos.

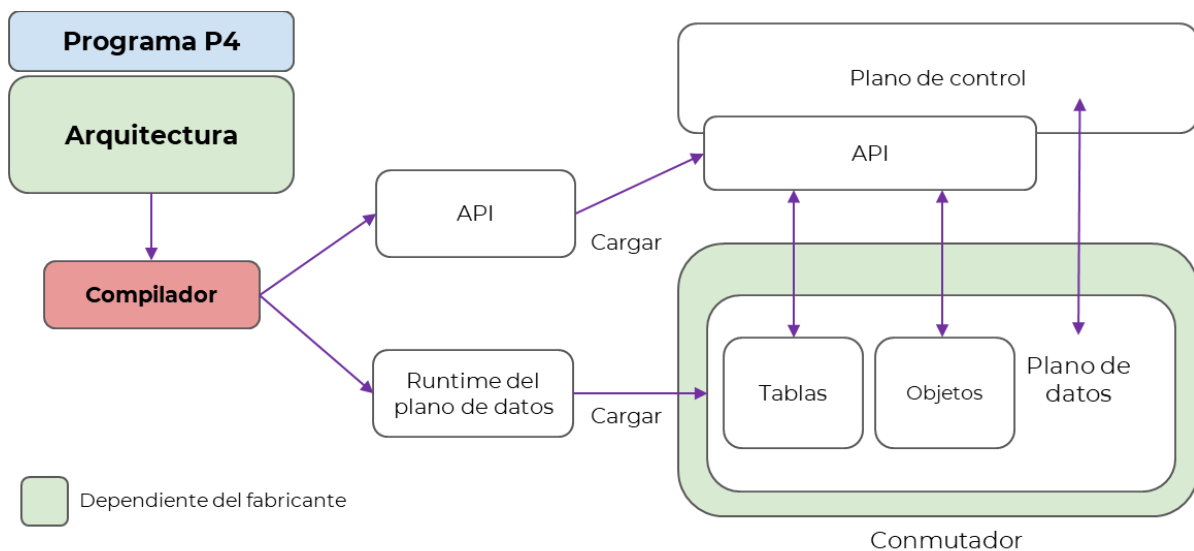


Figura 4. Arquitectura VModel

En nuestro caso, vamos a utilizar el siguiente entorno concreto de desarrollo:

- Emplearemos VSCode como IDE para la escritura de los programas en P4.
- Adoptaremos como conmutador de destino para probar y depurar los programas en P4 el conmutador software programable BMv2 [2] que ha sido concebido de forma específica para el desarrollo y prueba de programas en P4.
- El conmutador BMv2 soporta varios modelos de comportamiento como: simple_switch, simple_switch_grpc, psa_switch, etc, pero vamos a emplear por simplicidad el modelo de comportamiento simple_switch que sigue la arquitectura

VMModel.

- Utilizaremos el compilador estándar p4c que en este caso generará una salida JSON (.json) que utilizaremos como runtime del plano de datos a la hora de invocar el proceso/demonio del conmutador (simple_switch) y cuya información le permitirá implementar el procesamiento de paquetes definido en el programa P4 de partida.
- También utilizaremos la utilidad simple_switch_CLI en tiempo de ejecución para emular la manipulación de entradas de las tablas en el conmutador que debería realizar un nodo controlador del plano de control (no vamos a desplegar uno específico).

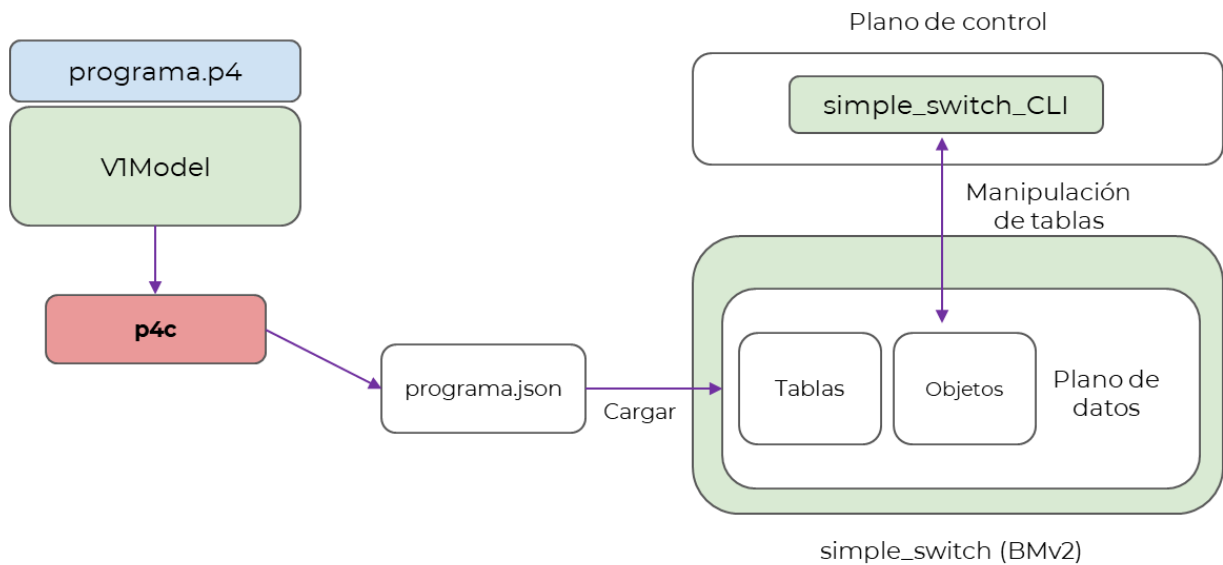


Figura 5. Entorno de desarrollo y ejecución de programas P4 que vamos a usar.

2 Laboratorio

Vamos a poner en práctica los pasos necesarios para configurar un entorno básico de desarrollo y simulación de funcionalidades sobre conmutadores software programables, haciendo uso del lenguaje P4 y del simulador GNS3.

Como caso de uso a implementar vamos a programar el comportamiento asociado a un repetidor.

2.1 Paso 1. Instalación de Docker

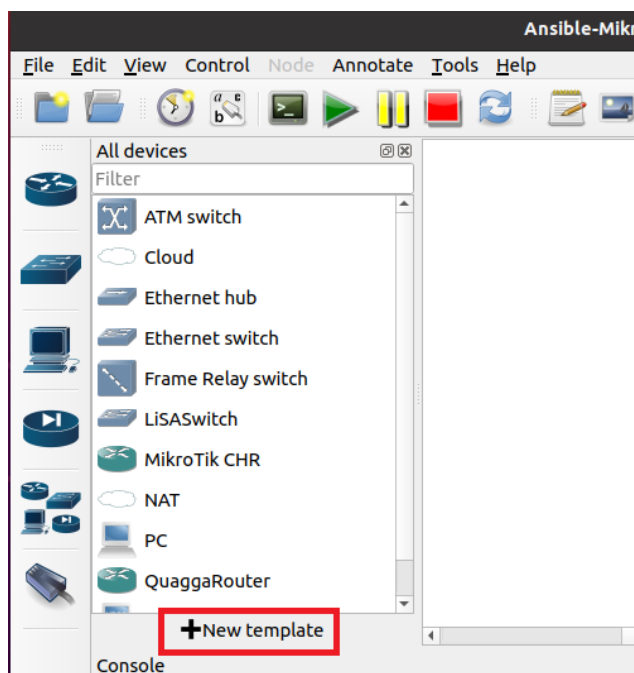
En un equipo en el que ya tengamos desplegado el entorno de trabajo con GNS3, vamos a instalar Docker con el objetivo de poder incorporar de forma sencilla los dispositivos que necesitemos como contenedores. Para realizar la instalación de Docker seguiremos los pasos indicados en [3](#).

2.2 Paso 2. Creación de plantillas de dispositivos

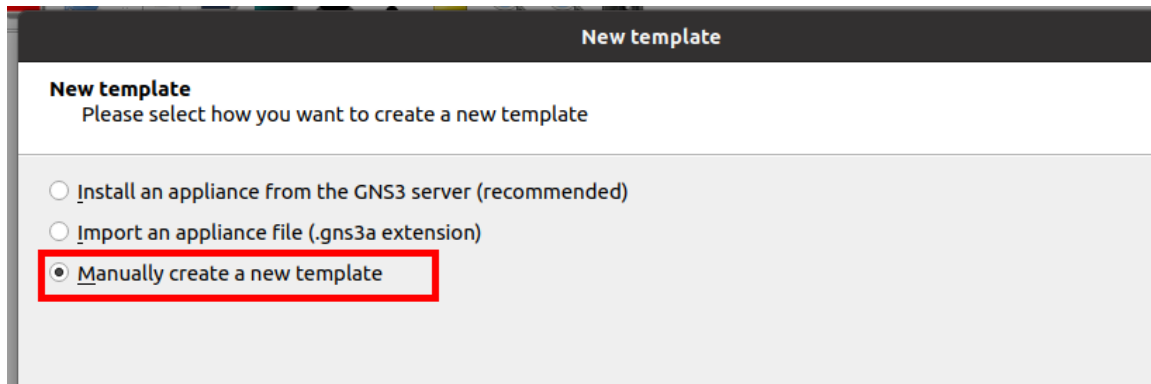
A continuación, vamos a abrir la herramienta GNS3, crearemos un nuevo proyecto y vamos a definir a continuación las plantillas que vamos a necesitar para la creación de dispositivos basados en contenedores Docker:

- *Nodo de desarrollo*: tendrá el software necesario para el desarrollo de programas con P4 (compilador p4c).
- *Conmutador software programable*: tendrá el software necesario para la ejecución del conmutador software BMv2 (principalmente simple_switch y simple_switch_CLI).

Ambas plantillas las vamos a crear de forma manual. Así que para ello, en ambos casos, vamos a la opción de crear nueva plantilla:



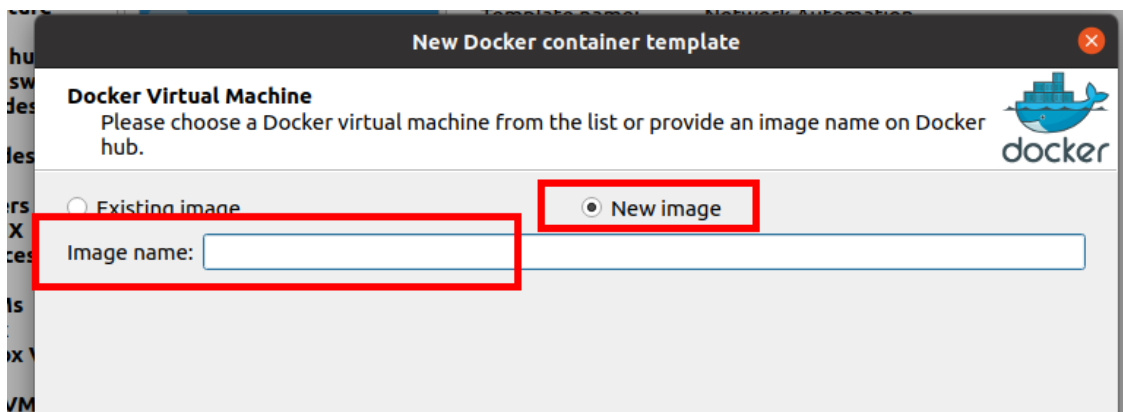
Seleccionaremos la opción de creación manual:



Haremos clic en tipo de dispositivo contenedor Docker y en el botón para nueva plantilla:



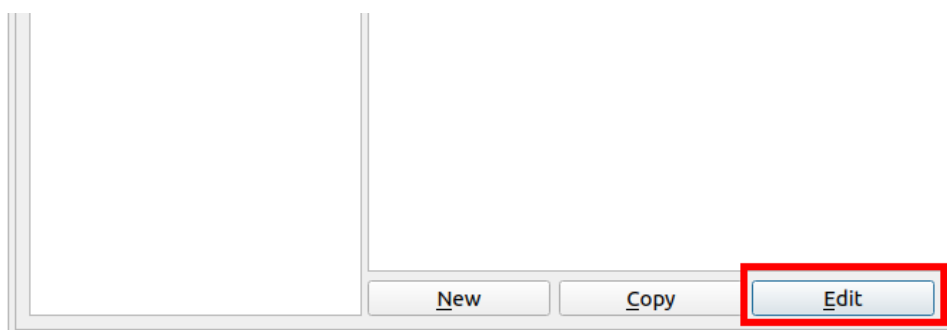
Marcaremos la opción de nueva imagen, y escribiremos a continuación en el campo correspondiente el nombre de la imagen base de Docker (en el repositorio Docker Hub) que vamos a emplear para el despliegue:



- Para el nodo de desarrollo: **p4lang/p4c**.
- Para el conmutador software: **p4lang/behavioral-model**.

Vamos a usar todas las opciones por defecto, salvo en la creación de la plantilla del conmutador software, que cuando nos consulte por el número de interfaces vamos a indicar que queremos crear los dispositivos con 8 interfaces (eth0 a eth7).

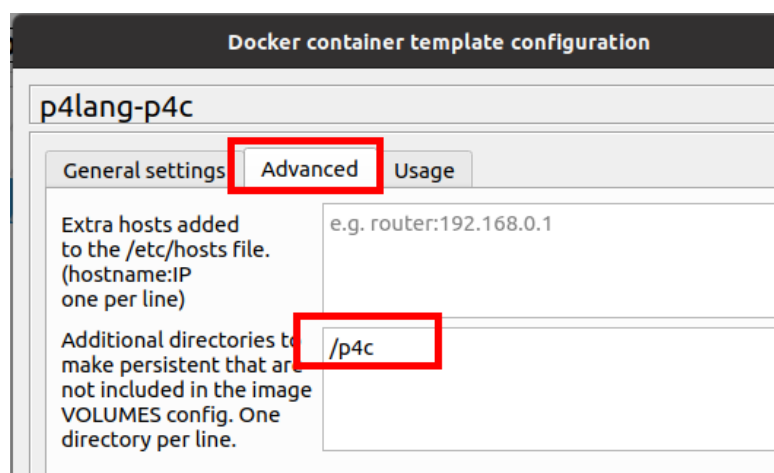
Una vez finalizado el asistente inicial al hacer clic en el botón siguiente (abajo a la derecha), ahora vamos a configurar en la plantilla que acabamos de crear la persistencia desde fuera del contenedor de las carpetas de trabajo dentro de cada contenedor. Para ello, seleccionaremos la plantilla creada y haremos clic en editar (para modificar su configuración por defecto).



Ahora, seleccionaremos la pestaña de opciones avanzadas y en el campo para la lista de directorios a persistir (como volumen del contenedor) añadimos la carpeta correspondiente en cada plantilla:

- Para el nodo de desarrollo: **/p4c**.
- Para el conmutador software: **/behavioral-model**.

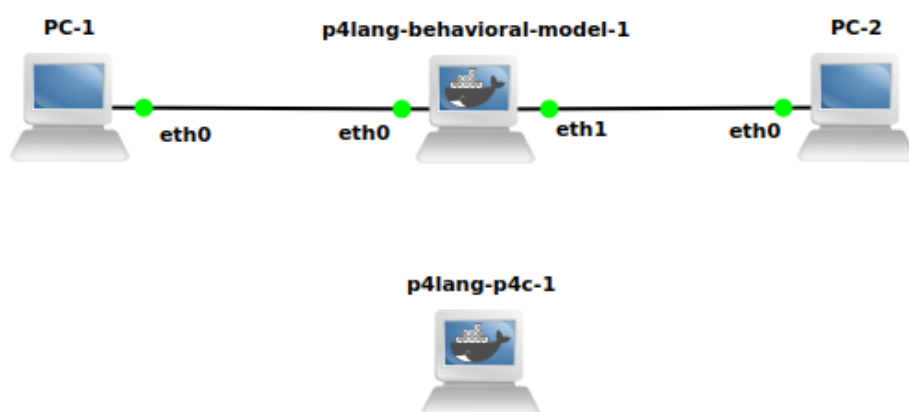
A continuación, se muestra este paso solamente para el nodo de desarrollo (se debe repetir siguiendo las mismas instrucciones para el conmutador software):



Para finalizar la creación de la plantilla haremos clic en Ok tanto en la ventana actual como en la siguiente (con esto deberíamos volver al proyecto).

2.3 Paso 3. Creación y configuración de dispositivos

Ahora vamos a incorporar al proyecto de GNS3 que creamos al principio; un dispositivo para cada una de las plantillas que creamos en el paso anterior y dos PC como dispositivos finales para las pruebas. Por último, vamos a conectar y configurar los elementos de la forma que se muestra a continuación:



Dispositivo	Descripción	Interfaz	Dirección IP
p4lang-behavioral-model-1	Conmutador software BMv2 basado en la imagen Docker p4lang/behavioral-model	-	-
p4lang-p4c-1	Nodo de desarrollo P4 basado en la imagen Docker p4lang/p4c	-	-
PC-1	Equipo de pruebas	eth0	10.0.0.1/24
PC-2	Equipo de pruebas	eth0	10.0.0.2/24

A continuación, iniciaremos la ejecución de todos los nodos del proyecto, y comprobaremos que de momento no se puede realizar con éxito un ping entre los dos PC de los extremos.

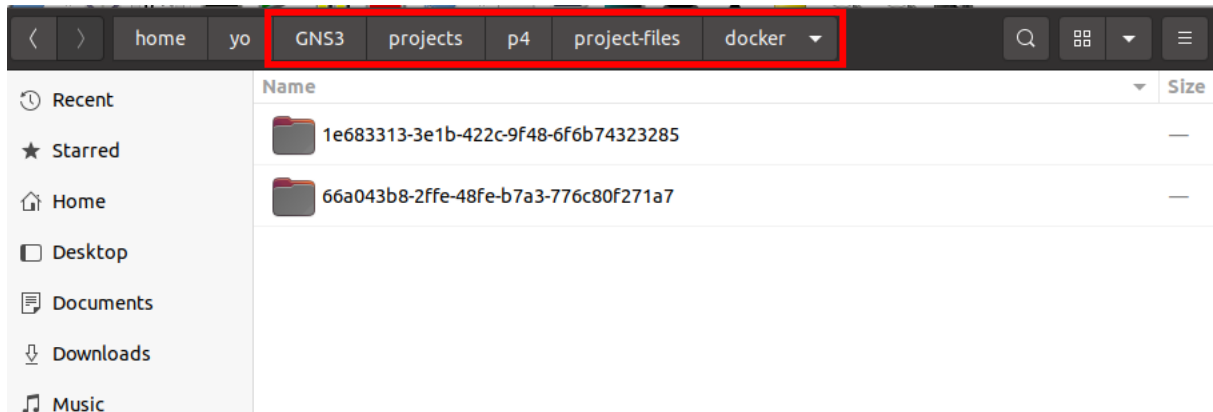
2.4 Paso 4. Configuración de carpetas de trabajo

A continuación, accederemos a la consola tanto del nodo de desarrollo como del conmutador, para modificar los permisos de las carpetas de trabajo /p4c y /behavioral_model, de forma que podamos acceder a su contenido de forma cómoda desde el exterior de los contenedores. Para ello, ejecutaremos los siguientes comandos (se muestra solamente para /p4c, repetir los mismos pasos para el nodo del conmutador):

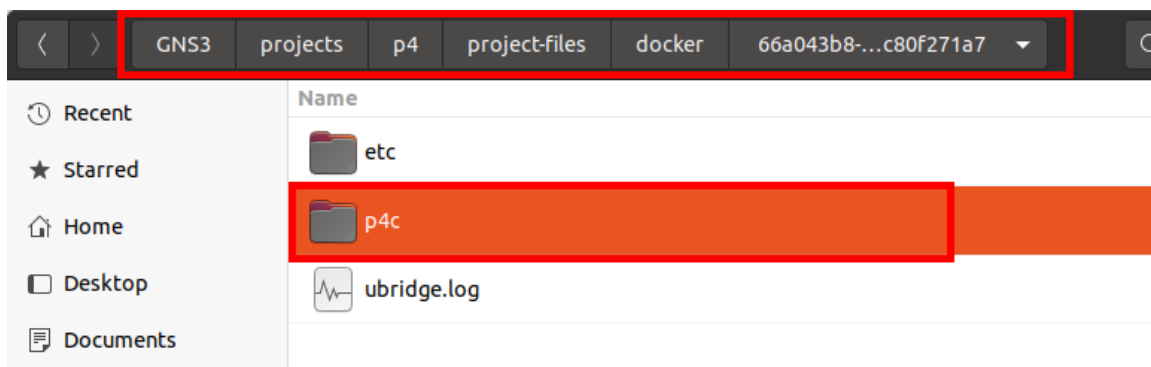
```
root@p4lang-p4c-1:/# cd /
root@p4lang-p4c-1:/# chmod 777 p4c
root@p4lang-p4c-1:/# cd /p4c
root@p4lang-p4c-1:/p4c# umask 000
```

A partir de este momento, ya podremos acceder desde nuestro ordenador (en donde se está ejecutando GNS3 y Docker) al contenido de estas carpetas en los nodos

contenerizados, simplemente navegando por el explorador de archivos. Para ello, debemos ir al directorio de instalación de GNS3, luego a la carpeta de proyectos, después a la carpeta del proyecto que acabamos de crear (en la figura p4), y a continuación navegamos a las subcarpetas project-files y luego docker, una vez desde allí, ya podremos acceder a las carpetas asociadas a cada contenedor (su nombre es su id):



Al acceder a alguna de estas carpetas, ya podremos ver las carpetas de trabajo cuya persistencia configuramos previamente y que ahora son accesibles desde el exterior del contenedor (en la siguiente figura vemos la carpeta /p4c dentro del nodo de desarrollo):



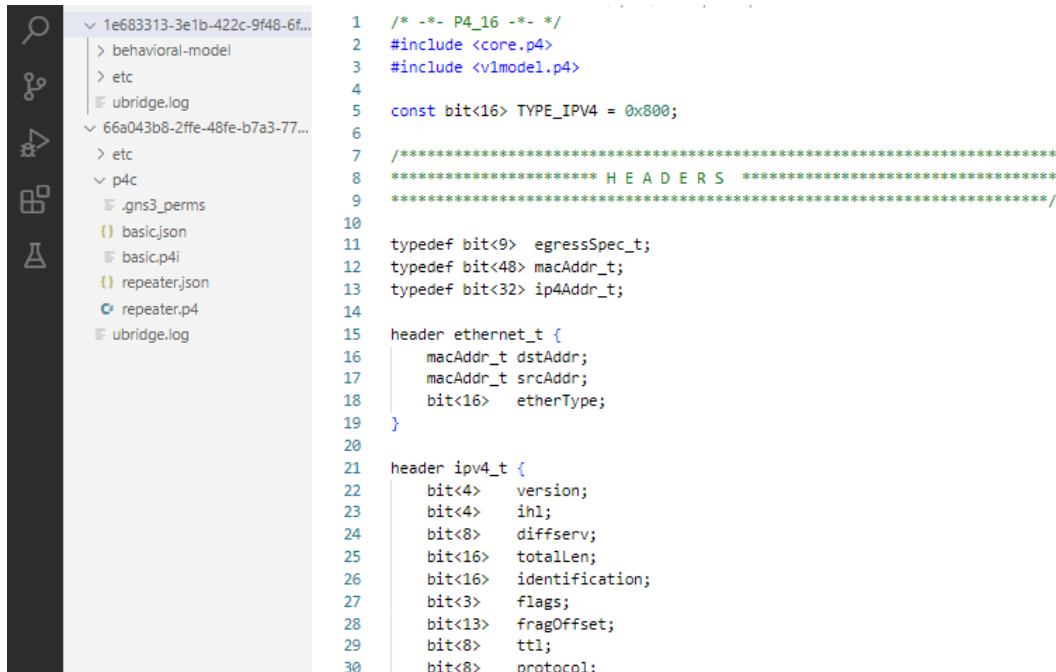
2.5 Paso 5. Escritura de los programas P4

Para el trabajo sobre los ficheros con código fuente en el lenguaje P4 (.p4) vamos a realizar la instalación y uso de un IDE. En nuestro caso, vamos a instalar y a usar Visual Studio Code (vscode). La instalación la podemos realizar directamente desde la utilidad Ubuntu Software del sistema (o podemos investigar y poner en práctica algún otro procedimiento alternativo).

A continuación, vamos a descargar, desde la carpeta de Anexos del proyecto en el aula virtual de la asignatura, el fichero repeater.p4 que contiene el código fuente del caso de uso que vamos a compilar y a desplegar.

Una vez descargado este fichero, lo vamos a copiar a la carpeta de trabajo /p4c en el nodo de desarrollo y lo vamos a abrir desde vscode (trabajaremos desde el entorno del ordenador en el que estamos utilizando GNS3). En vscode marcamos como lenguaje del fichero cpp para que se reconozca y se resalte la sintaxis asociada que es muy similar a la empleada por p4.

Como podemos ver, el contenido del fichero repeater.p4 sigue un patrón de diseño similar al que habíamos visto para la arquitectura VModel y por tanto incluye los siguientes bloques: HEADERS, PARSER, CHECKSUM VERIFICATION, INGRESS PROCESSING, EGRESS PROCESSING, CHECKSUM COMPUTATION, DEPARSER y SWITCH.



```

1  /* -*- P4_16 -*- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  const bit<16> TYPE_IPV4 = 0x800;
6
7  /***** HEADERS *****/
8  *****
9  *****
10
11  typedef bit<9>  egressSpec_t;
12  typedef bit<48> macAddr_t;
13  typedef bit<32> ip4Addr_t;
14
15  header ethernet_t {
16      macAddr_t dstAddr;
17      macAddr_t srcAddr;
18      bit<16>  etherType;
19  }
20
21  header ipv4_t {
22      bit<4>  version;
23      bit<4>  ihl;
24      bit<8>  diffserv;
25      bit<16> totalLen;
26      bit<16> identification;
27      bit<3>  flags;
28      bit<13> fragOffset;
29      bit<8>  ttl;
30      bit<8>  protocol;

```

El archivo repeater.p4 incluye a su inicio (además indica que la especificación del lenguaje que se va a usar es la última, es decir la P4_16) otros archivos que son específicos del lenguaje (core.p4) y de la arquitectura (v1model.p4). La inclusión de otras librerías se realiza con la directiva #include.

El bloque de HEADERS muestra los encabezados que se utilizarán en nuestro pipeline. Podemos ver que los encabezados de ethernet e IPv4 están definidos. También podemos ver cómo son agrupados en una estructura (struct headers). El nombre de los encabezados se utilizará en todo el programa al referirse a los mismos. Además, el archivo muestra cómo podemos usar typedef para proporcionar un nombre alternativo a un tipo.

Podemos ver que el parser ya está escrito con el nombre MyParser (en este caso solamente deja pasar). Este nombre se utilizará al definir la secuencia del pipeline al final del programa.

En este caso, el único bloque personalizado de forma importante es el de INGRESS PROCESSING, que describe el procesamiento de entrada. Podemos ver que está definido un bloque de control con el nombre MyIngress y en donde vemos que simplemente se realiza un intercambio del metadato del paquete del puerto de salida en función del puerto de entrada (se intercambia el tráfico entre los puestos 0 y 1 del conmutador).

Vemos a continuación que el resto de bloques: EGRESS PROCESSING, CHECKSUM COMPUTATION y DEPARSER, tienen secciones de control personalizadas que están vacías (dejan pasar el tráfico).

Por último, en el bloque SWITCH estamos definiendo la secuencia del pipeline de procesamiento de acuerdo con la arquitectura VModel, es decir, primero comenzamos por el analizador, luego verificamos la suma de verificación. Después, especifique el bloque de

entrada y el bloque de salida, y volvemos a calcular la suma de comprobación, y finalmente, especificamos el deparser, para volver a componer el paquete de salida (en este primer ejemplo no hay procesamiento en la mayoría de estos bloques).

```
68
69 /*****
70 ***** I N G R E S S   P R O C E S S I N G   *****
71 *****/
72
73 control MyIngress(inout headers hdr,
74                  inout metadata meta,
75                  inout standard_metadata_t standard_metadata) {
76
77     table test{
78         actions = {NoAction;}
79     }
80
81     apply {
82         test.apply();
83         if (standard_metadata.ingress_port == 0){
84             standard_metadata.egress_spec = 1;
85         }
86         else if (standard_metadata.ingress_port == 1){
87             standard_metadata.egress_spec = 0;
88         }
89     }
90 }
91
```

2.6 Paso 6. Compilación del programa P4

Una vez finalizada la edición del programa en P4, nos vamos a la consola del nodo de desarrollo (p4lang-p4c-1) y ejecutaremos los siguientes comandos para realizar la compilación y la generación de los artefactos correspondientes:

```
root@p4lang-p4c-1:/# cd /p4c
root@p4lang-p4c-1:/p4c# p4c repeater.p4
root@p4lang-p4c-1:/p4c# ls
repeater.json  repeater.p4  repeater.p4i
```

Ahora desde el visor de carpetas del ordenador en el que estamos ejecutando GNS3 vamos a copiar el fichero repeater.json (que se ha generado en la compilación) de la carpeta /p4c del nodo de desarrollo (p4lang-p4c-1) a la carpeta /behavioral-model del nodo conmutador (p4lang-behavioral-model-1).

2.7 Paso 7. Despliegue del programa

A continuación, desde la consola de p4lang-behavioral-model-1, vamos a ejecutar los siguientes comandos para lanzar el proceso que implementará la lógica que hemos programado en P4:

```
root@p4lang-behavioral-model-1:/# cd /behavioral-model
root@p4lang-behavioral-model-1:/behavioral-model#
root@p4lang-behavioral-model-1:/behavioral-model# simple_switch -i 0@eth0 -i 1@eth1 --nanolog
ipc:///tmp/bm-log.ipc repeater.json &
```



```
[1] 68
root@p4lang-behavioral-model-1:/behavioral-model# Calling target program-options parser
Adding interface eth0 as port 0
Adding interface eth1 as port 1
```

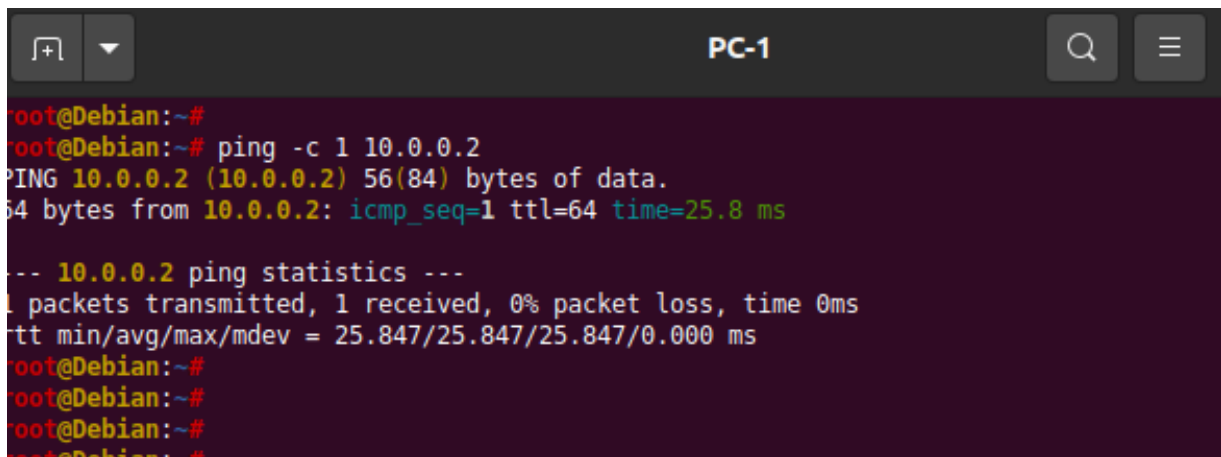
El comando `simple_switch` arranca un demonio que implementa un conmutador bajo la arquitectura VModel, con el mapeo de puertos entre el conmutador software y el nodo en el que se ejecuta que se indican con la opción `-i` (los puertos 0 y 1 del conmutador se mapean respectivamente a los puertos `eth0` y `eth1` del nodo) e implementando el runtime del plano de datos (archivo `repeater.json`) que se ha generado en la compilación de nuestro programa inicial en P4.

Con la opción `--nanolog` activamos el envío de los eventos de logs por IPC, con lo que podremos emplear el comando `bm_nanomsg_events`, sobre el puerto por defecto que utiliza el `simple_switch` para esta finalidad (el 9090), para consultar el debug del procesamiento de cada paquete tratado por el conmutador.

Para invocar el visor de eventos, una vez arrancado el conmutador, introducimos el siguiente comando:

```
root@p4lang-behavioral-model-1:/behavioral-model# bm_nanomsg_events --thrift-port 9090
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
```

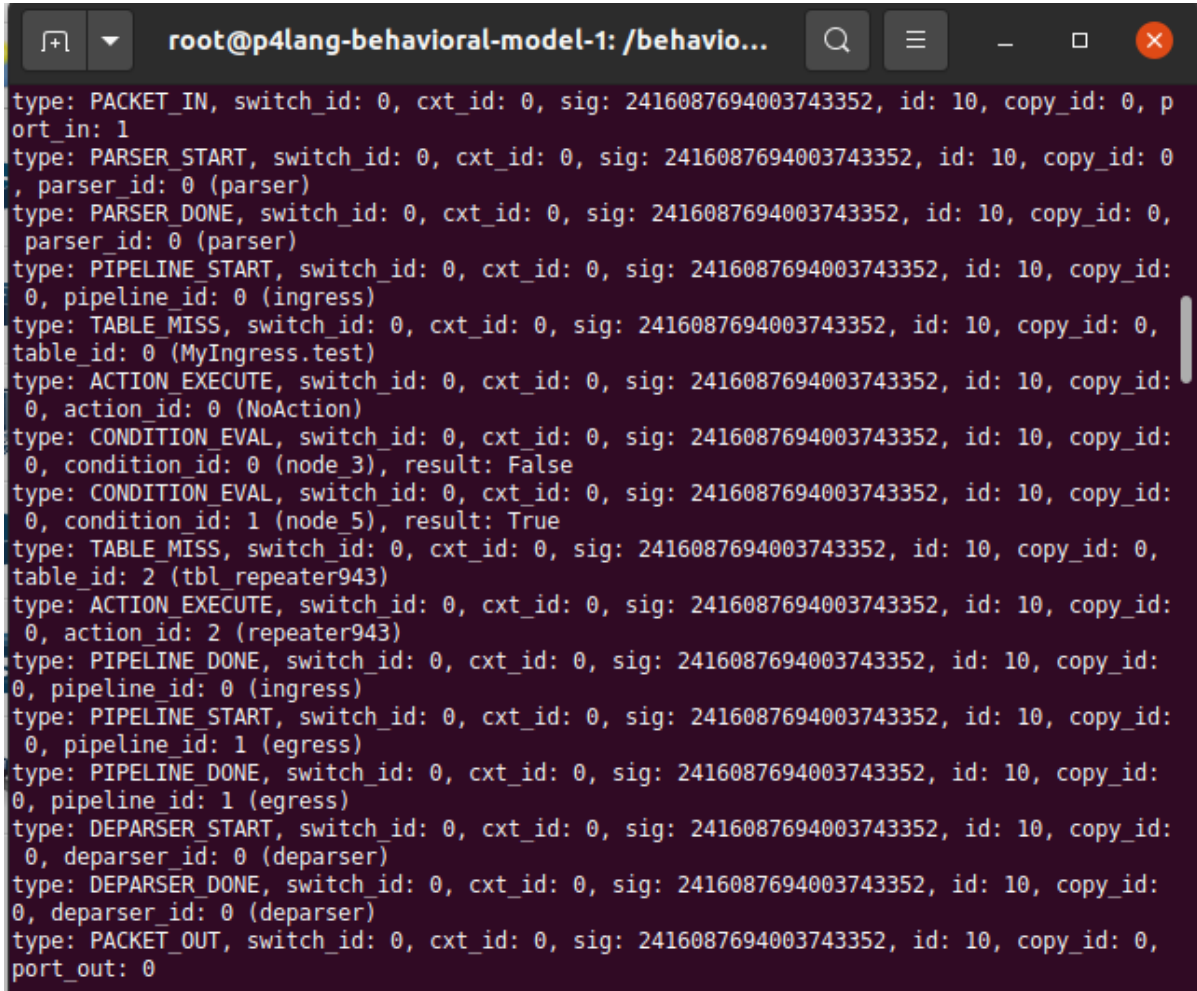
Ahora ya podremos comprobar que se puede hacer ping entre los PC de los extremos y podremos comprobar en la consola el detalle del procesamiento realizado:



```
root@Debian:~#
root@Debian:~# ping -c 1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=25.8 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 25.847/25.847/25.847/0.000 ms
root@Debian:~#
root@Debian:~#
root@Debian:~#
root@Debian:~#
```

Por último, podemos revisar la salida del log de eventos del conmutador, que se muestra a continuación. Podemos ver la lógica de procesamiento a medida que el paquete ingresa al conmutador. El paquete llega al puerto 1 (`port_in:1`), luego el parser comienza a extraer los encabezados. Una vez realizado el análisis, el paquete se procesa en los pipelines de entrada y salida. En este caso no hay match en la tabla pero hay dos evaluaciones de condiciones que son las que introducimos en el control `MyIngress`, la primera es `false`, pero la segunda es `true` (porque el paquete ha entrado por el puerto 1). Luego, se ejecuta la actualización de la suma de verificación y el analizador vuelve a ensamblar y emite el paquete usando el puerto 0 (`port_out: 0`), lo que significa que la lógica ha operado como esperábamos.

A terminal window titled 'root@p4lang-behavioral-model-1: /behavio...' displays a series of P4 behavioral model events. The events are listed line by line, each starting with 'type:'. The events include: PACKET_IN, PARSE_START, PARSE_DONE, PIPELINE_START, TABLE_MISS, ACTION_EXECUTE, CONDITION_EVAL, and PACKET_OUT. Each event includes various identifiers such as switch_id, cxt_id, sig, id, copy_id, parser_id, pipeline_id, condition_id, and action_id. The output is shown in a dark-themed terminal with a scrollbar on the right side.

```
type: PACKET_IN, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, port_in: 1
type: PARSE_START, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, parser_id: 0 (parser)
type: PARSE_DONE, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, parser_id: 0 (parser)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, pipeline_id: 0 (ingress)
type: TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, table_id: 0 (MyIngress.test)
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, action_id: 0 (NoAction)
type: CONDITION_EVAL, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, condition_id: 0 (node_3), result: False
type: CONDITION_EVAL, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, condition_id: 1 (node_5), result: True
type: TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, table_id: 2 (tbl repeater943)
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, action_id: 2 (repeater943)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, pipeline_id: 0 (ingress)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, pipeline_id: 1 (egress)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, pipeline_id: 1 (egress)
type: DEPARSER_START, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, deparser_id: 0 (deparser)
type: DEPARSER_DONE, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, deparser_id: 0 (deparser)
type: PACKET_OUT, switch_id: 0, cxt_id: 0, sig: 2416087694003743352, id: 10, copy_id: 0, port_out: 0
```

2.8 Paso 8: Exportación/importación de proyectos en GNS3

Investigue y realice pruebas para dominar el proceso necesario para la exportación e importación posterior de proyectos de GNS3 entre distintos equipos.

Para el desarrollo del proyecto se considera fundamental que se domine de forma adecuada el proceso necesario, de forma que se pueda garantizar la efectividad y la continuidad del trabajo tanto en el laboratorio como fuera del mismo.

3 Mejoras para el proyecto

3.1 Implementación de un reflector de tramas Ethernet

Desarrolle y despliegue un programa en P4 que implemente la lógica necesaria para implementar un reflector de tramas. El procesamiento principal que realizará será intercambiar la MAC de la trama Ethernet así como devolverla por el puerto por el que ha sido recibida.

Realice y documente las pruebas necesarias para demostrar su funcionamiento correcto. Explique y justifique lo observado.

3.2 Implementación de un repetidor con tablas match-action

Investigue sobre la definición y el uso de tablas de match-action en un programa en P4.

Desarrolle y despliegue a continuación un programa en P4 que implemente la lógica básica del repetidor descrita en el laboratorio anterior, pero haciendo uso esta vez de una tabla de match-action. Esta tabla deberá estar preparada para contener en cada entrada como clave el número del puerto de entrada, como acción posible en caso de coincidencia tendrá el procesamiento necesario para establecer un puerto determinado como de salida del paquete y como parámetro asociado a esta acción tendrá la opción de especificar el número de puerto concreto a usar como salida. La acción por defecto será no hacer nada.

Se deberá desplegar, introducir los valores necesarios en la tabla de match-action (emular el funcionamiento del plano de control) y realizar y documentar las pruebas necesarias para demostrar su funcionamiento correcto. Explique y justifique lo observado.

4 Referencias

- [1] Página oficial de P4: <https://p4.org/>
- [2] Conmutador software BMv2: <https://github.com/p4lang/behavioral-model>
- [3] Docker: <https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository>
- [4] Contenedor con compilador p4c: <https://hub.docker.com/r/p4lang/p4c>
- [5] Contenedor con software BMv2: <https://hub.docker.com/r/p4lang/behavioral-model>