Programación de dispositivos del plano de datos con P4

Parte 2

Índice

1 La arquitectura PISA

Es un modelo estándar y abierto para el procesamiento flexible de paquetes en el plano de datos de las redes definidas por software. La arquitectura de conmutación independiente del protocolo (PISA) es independiente de los protocolos y se caracteriza por incluir un conjunto de abstracciones genéricas en las que nos encontramos principalmente: un parser de encabezados, un pipeline de procesamiento basado en tablas de match-action y un deparser para reconstruir los 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 de esta forma los encabezados que deben ser reconocidos. 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).

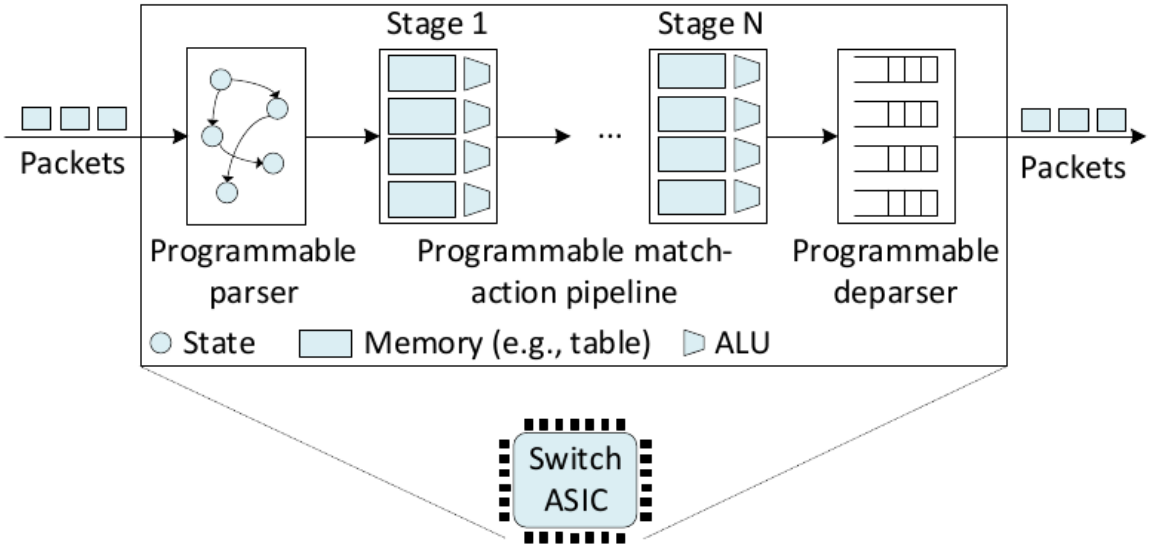


Figura 1. Arquitectura de conmutación independiente del protocolo (PISA)

El pipeline de procesamiento 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 con dependencias.

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 en sus entradas a partir de un determinado encabezado del paquete entrante (p.e. por IP de destino). Como hay varias tablas, el conmutador puede realizar búsquedas en paralelo empleando diferentes encabezados en cada una. 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 V1Model, 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 para el aprendizaje de estas tecnologías (el BMv2).

La arquitectura V1Model se compone de un parser, un pipeline de procesamiento de ingreso, un administrador de tráfico, un pipeline de procesamiento 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 realizar la replicación de paquetes cuando es necesario (p.e.: en supuestos de multidifusión).

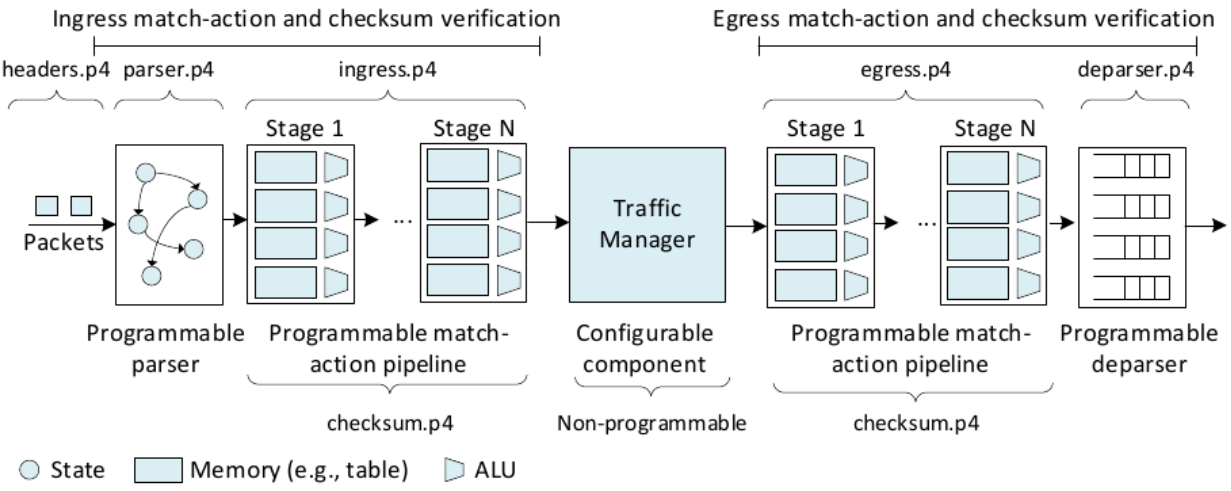


Figura 2. Arquitectura V1Model

1.1 El Lenguaje P4

El lenguaje para la programación del procesamiento de paquetes independiente del protocolo o P4 (*Protocol-independent Packet Processors*) es 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. P4 es un lenguaje específico de dominio (DSL) diseñado para definir de forma programática el flujo de procesamiento de paquetes dentro de un conmutador del plano de datos.

El uso de conmutadores programables con P4 permite incorporar las siguientes capacidades a los ingenieros de redes: 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), etc...

Un programa escrito en P4 se encarga principalmente de definir el formato de las tablas a emplear y las claves utilizadas para las operaciones de búsqueda sobre las mismas a partir de los encabezados de los paquetes. La incorporación de entradas en estas tablas se debe

realizar desde el plano de control, que es el responsable de incorporar las claves y los datos de acción necesarios para conseguir la funcionalidad deseada en cada momento (se trata de un comportamiento dinámico y muy flexible).

Un programa P4 puede implementar distintos modelos y/o arquitecturas de procesamiento, en nuestro caso vamos a desarrollar programas que implementarán la arquitectura V1Model, y por tanto, contendrán los siguientes bloques principales:

- headers: contiene los encabezados de los paquetes y las definiciones de los metadatos.
- parser: contiene la implementación del parser programable.
- ingress: contiene el bloque de control de ingreso que incluye las tablas de match-action
- egress: contiene el bloque de control de salida.
- deparser: contiene la lógica deparser o de reensamblaje de los paquetes.

1.2 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.

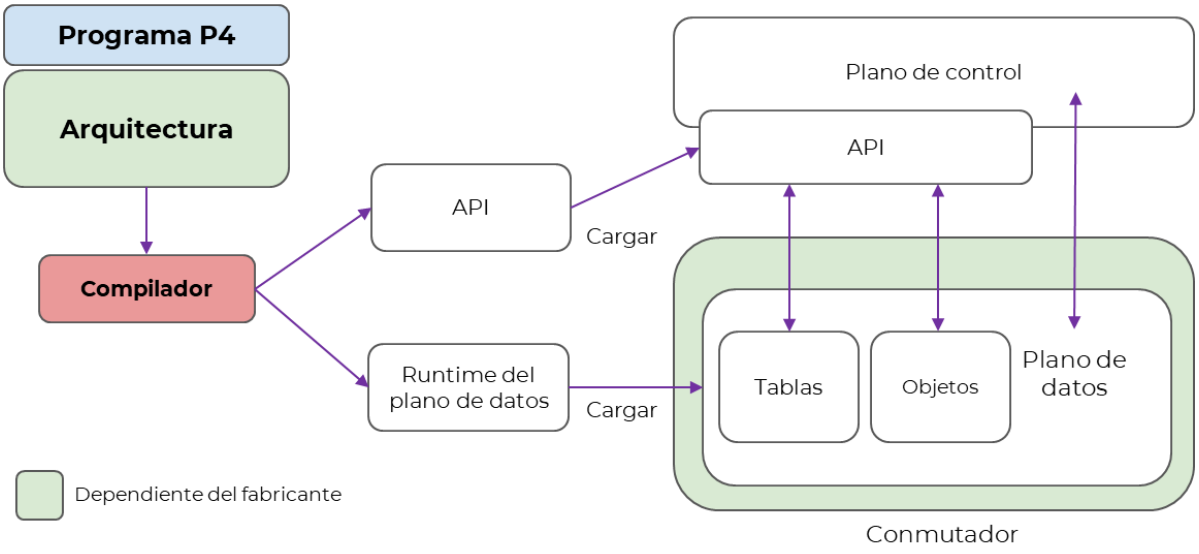


Figura 3. Arquitectura V1Model

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 [3] 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 V1Model.
- 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).

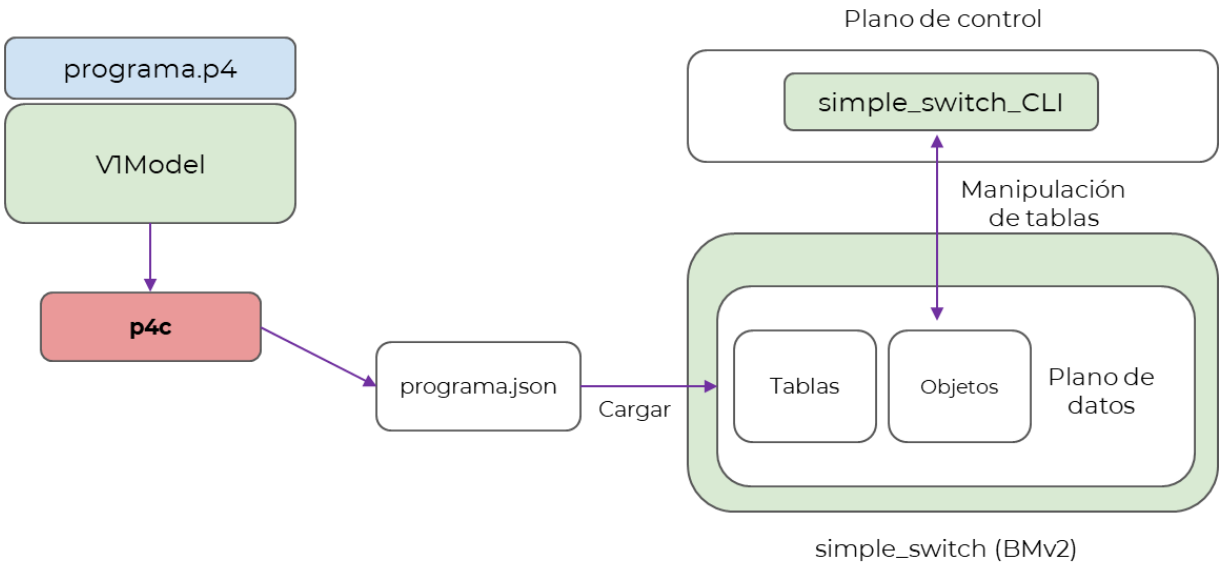


Figura 4. Entorno de desarrollo y ejecución de programas P4 que vamos a usar.

2 Laboratorio

Vamos a poner en práctica los pasos necesarios para el desarrollo y simulación de la funcionalidad de IP forwarding sobre un conmutador software programable (BMv2), haciendo uso del lenguaje P4 y del simulador GNS3. Vamos a utilizar el entorno de desarrollo y de pruebas configurado en el laboratorio anterior, y que cumple los siguientes requisitos:

- Tener instalada la solución Docker de gestión de contenedores software.
- Tener creadas en GNS3 las plantillas de dispositivos para las imágenes Docker siguientes:
  - *Nodo de desarrollo*: con el compilador p4c.
    - Imagen: **p4lang/p4c**.
    - Carpeta con persistencia como volumen: **/p4c**.
  - *Conmutador software programable*: con el software del conmutador software BMv2 (simple_switch y simple_switch_CLI).
    - Imagen: **p4lang/behavioral-model**.
    - Carpeta con persistencia como volumen: **/behavioral-model**.

2.1 Paso 1. Configuración de dispositivos

Vamos a reutilizar los dispositivos y configuraciones del proyecto de GNS3 que empleamos en el laboratorio anterior (vamos a arrancar todos los nodos y a actualizar las direcciones IP y direcciones MAC de los PC):

Dispositivo	Descripción	Interfaz	Dirección IP	MAC
p4lang-behavioral-model-1	Conmutador software BMv2	-	-	-
p4lang-p4c-1	Nodo de desarrollo P4	-	-	-
PC-1	Equipo de pruebas	eth0	10.0.0.1/8	00:00:00:00:00:01
PC-2	Equipo de pruebas	eth0	10.0.1.1/8	00:00:00:00:00:02

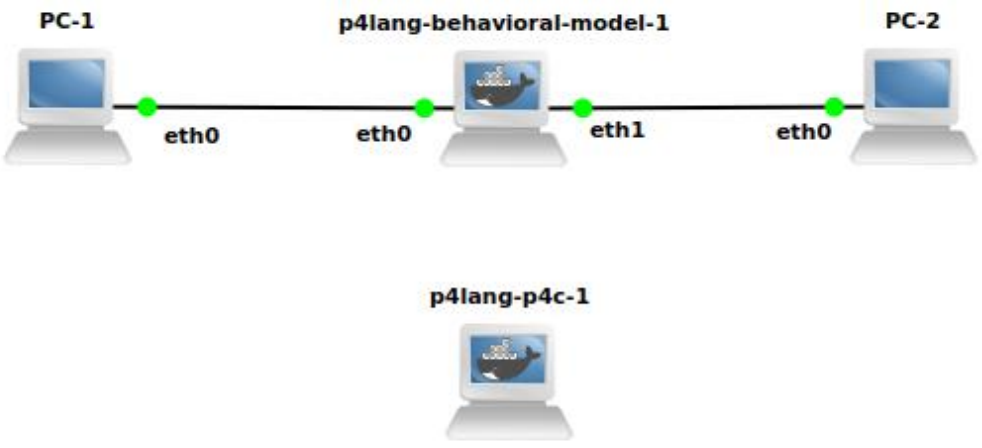


Figura 5. Esquema del montaje para la simulación.

2.2 Paso 2. Escritura del programa P4

A continuación, vamos a descargar, desde la carpeta de Anexos del proyecto en el aula virtual de la asignatura, el fichero basic.p4 y que contiene el código fuente del caso de uso que vamos a trabajar de implementación de una funcionalidad de IP Forwarding.

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 sintáxis asociada que es muy similar a la empleada por p4.

Como podemos ver, el contenido del fichero basic.p4 sigue el patrón de diseño asociado a la arquitectura V1Model y por tanto incluye los bloques: HEADERS, PARSER, CHECKSUM VERIFICATION, INGRESS PROCESSING, EGRESS PROCESSING, CHECKSUM COMPUTATION, DEPARSER y SWITCH.

El código comienza incluyendo el archivo core.p4 (línea 2) y que define algunos tipos comunes y variables utilizadas en todos los programas P4 (por ejemplo, los tipos packet_in y packet_out). Luego se incluye el archivo v1model.p4 (línea 3) para definir el uso de la arquitectura V1Model. La línea 5 crea una constante de 16 bits llamada TYPE_IPV4 con el valor 0x800.

```
1  /* -*- P4_16 -*- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  const bit<16> TYPE_IPV4 = 0x800;
6
```

2.2.1 Código del bloque HEADERS

En el bloque HEADERS, tenemos primero las declaraciones typedef (líneas 11 - 13) que se utilizan para asignar nombres alternativos a los tipos.

Posteriormente, las cabeceras y los metadatos definen las estructuras que se utilizarán en todo el programa.

Estos encabezados se pueden personalizar dependiendo de cómo el programador quiera que se analicen los paquetes. En este caso se definen los encabezados para Ethernet (líneas 15 a 19) y para IPv4 (líneas 21 a 34).

También podemos usar una estructura que representa los metadatos específicos del programa y que su utilidad es que pasan de un bloque a otro a medida que el paquete se propaga a través de la arquitectura.

Por simplicidad, este programa no requiere ningún metadato de usuario y, por lo tanto, lo definiremos como vacío.

```
6
7   /*************************************************************************
8   ********************** H E A D E R S  ***********************************
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;
31      bit<16>   hdrChecksum;
32      ip4Addr_t srcAddr;
33      ip4Addr_t dstAddr;
34  }
35
36  struct metadata {
37      /* empty */
38  }
39
40  struct headers {
41      ethernet_t   ethernet;
42      ipv4_t       ipv4;
43  }
44
```

2.2.2 Código del bloque PARSER

En este bloque se describe cómo funcionará el analizador de campos (parser) a la hora de procesar los paquetes (definimos en este caso uno específico llamado MyParser).

El parser se encarga de desencapsular los encabezados y se puede representar como una máquina de estados sin ciclos con un estado inicial (inicio) y dos estados finales (aceptar o rechazar).

En P4, un parser siempre comienza con el estado inicial start. En este caso (MyParser), primero hacemos la transición incondicional al estado parse_ethernet (los estados son bloques de código que describen el tratamiento a realizar en el mismo).

Dentro de cada estado podemos crear algunas condiciones para dirigir el parser y finalmente transitar a alguno de los estados finales de aceptación o de bloqueo. Si es de aceptación (accept) el paquete pasará al bloque de INGRESS PROCESSING, o en caso contrario se descartará.

Como vemos el paquete que entra al parser MyParser es una instancia de packet_in (tipo

específico de V1Model) y la ejecución se inicia en el estado inicial start. Dentro de este estado inicial se realiza una transición incondicional al estado llamado parser_ethernet, y dentro del mismo, se usa el método extract del paquete para extraer y copiar en las estructuras de datos asociados a los encabezados en el programa (hdr) los N bits definidos en cada caso (por ejemplo, para Ethernet extrae del paquete los 112 bits correspondientes y los copia a las estructuras de datos de encabezado del programa).

```
/*************************************************************************
*********************** P A R S E R  ***********************************
*************************************************************************/

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) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

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

}
```

Posteriormente, se realiza una transición condicionada al contenido del campo etherType en el encabezado Ethernet, para ello se usa la instrucción select.

El programa bifurca entonces al estado parse_ipv4 si el campo etherType corresponde a IPv4 y en caso contrario de acepta (se podría haber denegado). Por último, en el estado parse_ipv4, se extrae del paquete el encabezado IPv4 y se copia a las estructuras del programa y a continuación se acepta el tráfico (para que el paquete transite al siguiente bloque del pipeline).

Este código se puede representar a través del siguiente DAG (gráfico directo acíclico):

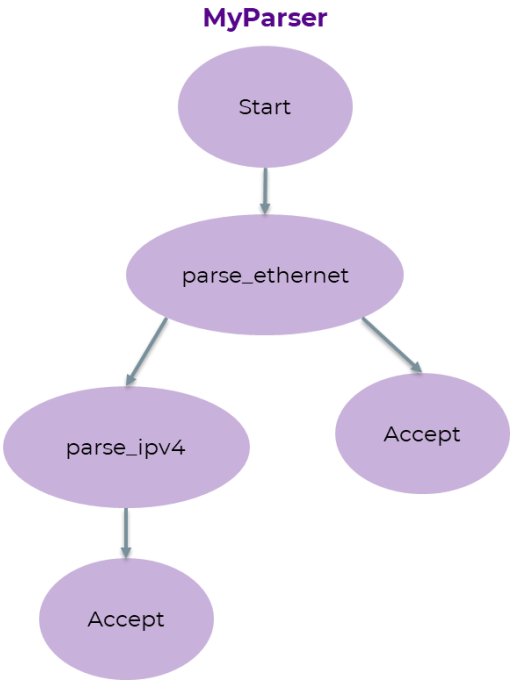


Figura 6. Representación del DAG asociado a la funcionalidad del parser.

2.2.3 Código de los bloques PROCESSING

Los bloques de control son las partes esenciales del procesamiento de un paquete en un programa de P4 y se usan en combinación con las tablas. Un bloque de control puede incluir acciones para reenviar un paquete cuando ocurre una coincidencia de una tabla determinada, y para descartar el paquete en caso contrario.

Las tablas son componentes esenciales que definen el comportamiento de procesamiento de un paquete dentro del conmutador. Cada tabla se especifica en el programa P4 y tiene una o más entradas (filas) que deben ser pobladas por el plano de control. Una entrada contiene una clave, una acción y los datos asociados a la acción.

- Clave: se utiliza para las operaciones de búsqueda. Se crea utilizando uno o más campos de encabezado.
- Acción: una vez que ocurre una coincidencia, la acción especificada en la entrada es realizada por la ALU. Son operaciones simples como modificar un campo de encabezado, reenvía el paquete a un puerto de salida y/o descarta el paquete.
- Datos de la acción: son parámetro/s utilizados junto con la acción. Por ejemplo, podría ser el número de puerto que el switch debe usar para reenviar el paquete.

Por ejemplo, para reenviar un paquete a nivel 3, un conmutador debe realizar una búsqueda en una tabla indexada por la dirección IP de destino.

Se pueden utilizar tres tipos de coincidencia con las entradas de una tabla (se pueden incorporar tipos adicionales como en V1Model, que se incorpora la coincidencia basada en rangos y en selectores):

- Coincidencia exacta: se utiliza para buscar un valor específico que coincide de forma completa con la clave indicada en una entrada de la tabla.

- Coincidencia de prefijo más largo (LPM): se utiliza para buscar un valor cuya coincidencia es parcial, y se elige de entre todas las posibilidades, aquella cuya coincidencia es mayor en número de bits (longest prefix match). Es muy útil para analizar las coincidencias entre direcciones IP y direcciones de subredes IP.
- Ternaria.

Como vemos dentro del bloque de INGRESS PROCESSING hemos declarado un bloque de control denominado MyIngress. Nuestro objetivo dentro de este tipo de bloques es definir las tablas P4 que vamos a necesitar y sus acciones asociadas.

```
/*************************************************************************
*************  I N G R E S S   P R O C E S S I N G   ******************
*************************************************************************/

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 dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        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();
        } else {
            if (standard_metadata.ingress_port == 0){
              standard_metadata.egress_spec = 1;
            } else if (standard_metadata.ingress_port == 1){
              standard_metadata.egress_spec = 0;
            }
        }
    }
}
```

Lo primero que vamos a hacer es definir las acciones posibles que se llamarán desde la/s tabla/s. En este caso solamente vamos a usar dos acciones:

- drop: desechar el paquete.
- Ipv4_forward: realiza el siguiente tratamiento en este caso:
    - Actualización del puerto de salida para que el paquete sea reenviado a su destino a través del puerto correcto.
    - Actualización de la MAC origen con la MAC del destino anterior del paquete.
    - Actualización de la MAC destino con la correspondiente al siguiente salto recuperado de la tabla.
    - Reduce el campo de tiempo de vida (TTL) en el encabezado IPv4.

La instancia standard_metadata es una estructura del V1Model y contiene metadatos intrínsecos que son útiles en el procesamiento de paquetes. Por ejemplo, sirve para determinar el puerto en el que un paquete llega (ingress_port) o en el que el paquete va a salir (egress_spec). La palabra clave default_action especifica en una tabla qué acción predeterminada se invocará siempre que no haya una coincidencia (en este caso es descartar el paquete).

La sección apply{} define el flujo secuencial de procesamiento de paquetes. Se requiere en cada bloque de control, o de lo contrario el programa no se compila. Describe en orden, la secuencia de tablas a invocar, entre otras instrucciones de procesamiento de paquetes.

Vemos que el procesamiento de la tabla que acabamos de crear se realiza solamente si hdr.ipv4.isValid(), esto se hace con la idea de que este procesamiento solamente aplique al tráfico IPv4, y en el resto de casos (como el tráfico ARP) la funcionalidad que se implementa simplemente es de repetidor entre los dos puertos.

La tabla ipv4_lpm tiene la estructura indicada y debería tener el contenido necesario para soportar el forwarding IP entre las dos redes y equipos que hemos interconectado en el proyecto. Estas entradas específicas deberían ser incorporadas en condiciones normales desde el plano de control.

2.3 Paso 3. 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 (en la subcarpeta basic.bmv2):

```
root@p4lang-p4c-1:/# cd /p4c
root@p4lang-p4c-1:/p4c# p4c -b bmv2 basic.p4 -o basic.bmv2
root@p4lang-p4c-1:/p4c# ls basic.bmv2/
basic.json  basic.p4i
```

Ahora desde el explorador de carpetas del ordenador en el que estamos ejecutando GNS3 vamos a copiar el fichero basic.json (que se ha generado en la compilación) de la carpeta /p4c/basic.bmv2 del nodo de desarrollo (p4lang-p4c-1) a la carpeta /behavioral-model del nodo conmutador (p4lang-behavioral-model-1) para realizar el despliegue de la funcionalidad.

2.4 Paso 4. Despliegue del programa

A continuación, desde la consola de p4lang-behavioral-model-1, vamos a ejecutar los siguientes comandos para lanzar el demonio del conmutador 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 basic.json &
[1] 68
```

El comando simple_switch arranca un demonio que implementa un conmutador bajo la arquitectura V1Model, con el mapeo de puertos entre el conmutador software (conceptuales) y el nodo en el que se ejecuta (reales) que se indican con la opción -i (los puertos 0 y 1 del conmutador se mapean respectivamente a los puertos eth0 y eth1 del servidor) e implementando el runtime del plano de datos (fichero basic.json) que se ha generado en la compilación de nuestro programa inicial en P4.

Con la opción -–nanolog activamos el envío de 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 (el 9090), para consultar el detalle del procesamiento de cada paquete tratado por el conmutador.

2.5 Paso 5. Población de tablas

Las tablas en los conmutadores están pensadas para ser populadas desde el plano de control, según las necesidades del procesamiento, y se realiza incluso de forma dinámica durante el procesamiento de cada paquete (desde el nodo controlador a través de alguno de los protocolos de control habituales en las redes SDN como OpenFlow).

En nuestro proyecto, por simplicidad, no vamos a incorporar un controlador SDN para asumir la gestión del plano de control, y en su lugar vamos a emular su presencia a través del uso del comando simple_switch_CLI que nos va a permitir manipular directamente el contenido de las tablas de conmutador.

Para nuestro caso de uso, deberíamos incorporar las siguientes entradas en nuestra tabla ip4_lvm para conseguir la comunicación haciendo forwarding ip entre los dos PC:

Key	action	data action
10.0.0.0/24	ipv4_forward	port 0, macAddr=00:00:00:00:00:01
10.0.1.0/24	ipv4_forward	port 1, macAddr=00:00:00:00:00:02
default	drop	

La clave de la primera entrada es 10.0.0.0/24 (que se traduce como 0a:00:00:00 en hexadecimal) y su acción es forward. Los parámetros de acción son 00:00:00:00:00:01 para la MAC de destino (es decir, la MAC de host PC1) y 0 para el puerto de salida (es decir, el puerto frente al PC1).

La clave de la segunda entrada es 10.0.1.0/24 (que se traduce como 0a:00:01:00 en hexadecimal) y su acción es forward. Los parámetros de acción son 00:00:00:00:00:02 para la MAC de destino (es decir, la MAC de host PC2) y 1 para el puerto de salida (es decir, el puerto frente al PC2).

Ahora vamos a usar la herramienta simple_switch_CLI para administrar los objetos P4 en tiempo de ejecución. La podemos invocar desde la consola una vez que hayamos arrancado previamente el demonio del conmutador (simple_switch):

```
root@p4lang-behavioral-model-1:/behavioral-model# simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd:
```

Podremos consulta la ayuda de cada comando e invocarlo (para ver la lista de comandos podemos usar ?) de la siguiente forma (en este ejemplo consultamos los puertos definidos):

```
RuntimeCmd: help show_ports
Shows the ports connected to the switch: show_ports
RuntimeCmd: show_
show_actions  show_ports    show_pvs      show_tables
RuntimeCmd: show_ports
  port #       iface name       status  extra info
==================================================
    0            eth0            UP
    1            eth1            UP
```

Con el comando show_tables, vemos las tablas que usa el conmutador:

```
RuntimeCmd: show_tables
MyIngress.ipv4_lpm             [implementation=None, mk=ipv4.dstAddr(lpm, 32)]
RuntimeCmd:
```

Para ver las acciones en uso por el conmutador escribimos:

```
RuntimeCmd: show_actions
MyIngress.drop                 []
MyIngress.ipv4_forward         [dstAddr(48),     port(9)]
NoAction                       []
```

Para ver información básica del conmutador usamos:

```
RuntimeCmd: switch_info
device_id                : 0
thrift_port              : 9090
notifications_socket     : ipc:///tmp/bmv2-0-notifications.ipc
elogger_socket           : ipc:///tmp/bm-log.ipc
debugger_socket          : None
```

Para ver la información de una tabla podemos usar el siguiente comando:

```
RuntimeCmd: table_info MyIngress.ipv4_lpm
MyIngress.ipv4_lpm             [implementation=None, mk=ipv4.dstAddr(lpm, 32)]
**********************************************************************
MyIngress.drop                 []
MyIngress.ipv4_forward         [dstAddr(48),     port(9)]
NoAction                       []
```

Para volcar el contenido de una tabla (volcado de contenido y número de entradas) usamos los siguientes comandos:

```
RuntimeCmd: table_dump MyIngress.ipv4_lpm
==========
TABLE ENTRIES
```

```
==========
Dumping default entry
Action entry: MyIngress.drop -
==========
RuntimeCmd: table_num_entries MyIngress.ipv4_lpm
0
```

Para añadir entradas usamos el comando table_add, que tiene la siguiente sintáxis:

```
RuntimeCmd: help table_add
Add entry to a match table: table_add <table name> <action name> <match fields> => <action parameters>
[priority]
```

- <table_name>: nombre de la tabla.
- <action name>: la acción asociada con la entrada.
- <match fields>: la clave utilizada para comparar con el paquete entrante.
- <action parameters>: el parámetro asociado con la entrada.
- [prioridad]: de la entrada.

Para añadir las entradas que hemos comentado debemos lanzar los siguientes comandos:

```
table_set_default ipv4_lpm drop
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.0/24 => 00:00:00:00:00:01 0
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.1.0/24 => 00:00:00:00:00:02 1
```

Para los dos últimos comandos, como ejemplo, las salidas serían:

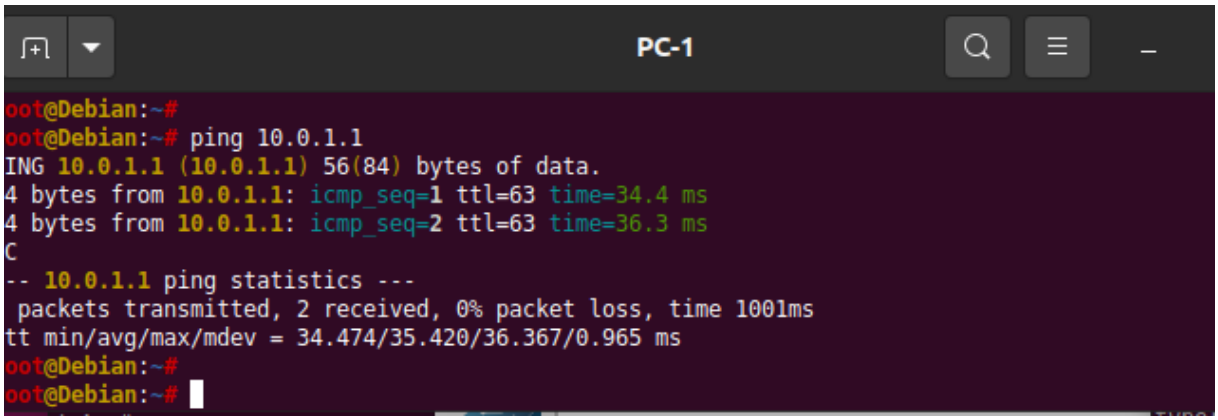
```
RuntimeCmd: table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.0/24 => 00:00:00:00:00:01 0
Adding entry to lpm match table MyIngress.ipv4_lpm
match key:           LPM-0a:00:00:00/24
action:              MyIngress.ipv4_forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0


RuntimeCmd: table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.1.0/24 => 00:00:00:00:00:02 1
Adding entry to lpm match table MyIngress.ipv4_lpm
match key:           LPM-0a:00:01:00/24
action:              MyIngress.ipv4_forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
```

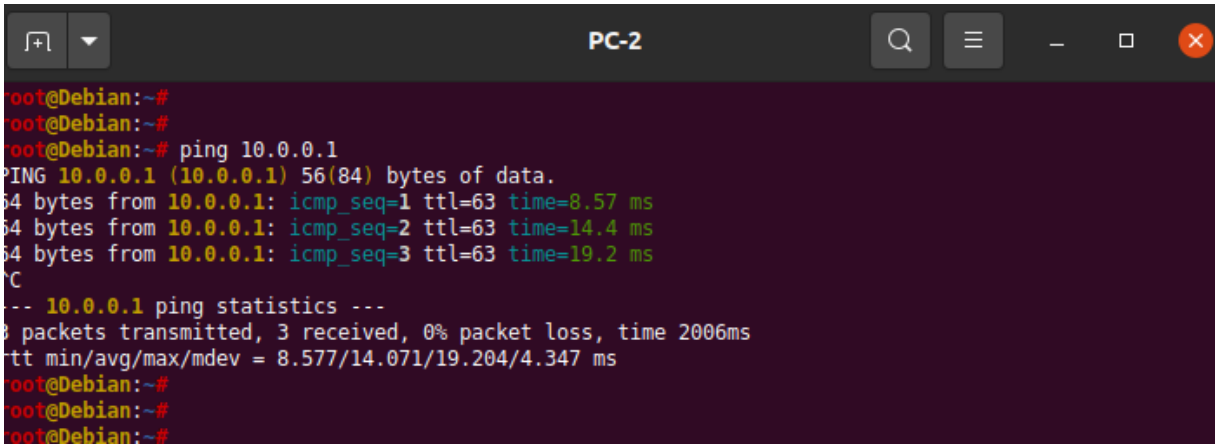
2.6 Paso 6. Comprobar la comunicación

Una vez arrancado el conmutador y creadas las entradas necesarias en las tablas, podemos invocar el visor de eventos para ver el detalle del procesamiento en el conmutador durante las pruebas:

```
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...
```

A continuación, ya podremos lanzar y comprobar que se puede hacer ping entre los PC de los extremos:





Además, podremos comprobar en la consola anterior el detalle del procesamiento realizado. En este caso, mostramos la salida para el procesamiento de los paquetes ICMP request y replay:

```
type: PACKET_IN, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, port_in: 0
type: PARSER_START, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, parser_id: 0 (parser)
type: PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, header_id: 2 (ethernet)
type: PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, header_id: 3 (ipv4)
type: PARSER_DONE, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, parser_id: 0 (parser)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, condition_id: 0 (node_2),
result: True
type: TABLE_HIT, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, table_id: 0 (MyIngress.ipv4_lpm),
entry_hdl: 1
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, action_id: 2
(MyIngress.ipv4_forward)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, pipeline_id: 0 (ingress)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, pipeline_id: 1 (egress)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, pipeline_id: 1 (egress)
type: DEPARSER_START, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, header_id: 2 (ethernet)
type: DEPARSER_EMIT, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, header_id: 3 (ipv4)
type: DEPARSER_DONE, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, deparser_id: 0 (deparser)
type: PACKET_OUT, switch_id: 0, cxt_id: 0, sig: 6187611452747271376, id: 19, copy_id: 0, port_out: 1

type: PACKET_IN, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, port_in: 1
type: PARSER_START, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, parser_id: 0 (parser)
type: PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, header_id: 2 (ethernet)
type: PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, header_id: 3 (ipv4)
type: PARSER_DONE, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, parser_id: 0 (parser)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, condition_id: 0 (node_2),
result: True
type: TABLE_HIT, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, table_id: 0 (MyIngress.ipv4_lpm),
entry_hdl: 0
```

```
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, action_id: 2
(MyIngress.ipv4_forward)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, pipeline_id: 0 (ingress)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, pipeline_id: 1 (egress)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, pipeline_id: 1 (egress)
type: DEPARSER_START, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, header_id: 2 (ethernet)
type: DEPARSER_EMIT, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, header_id: 3 (ipv4)
type: DEPARSER_DONE, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, deparser_id: 0 (deparser)
type: PACKET_OUT, switch_id: 0, cxt_id: 0, sig: 7292157593906687331, id: 20, copy_id: 0, port_out: 0
```

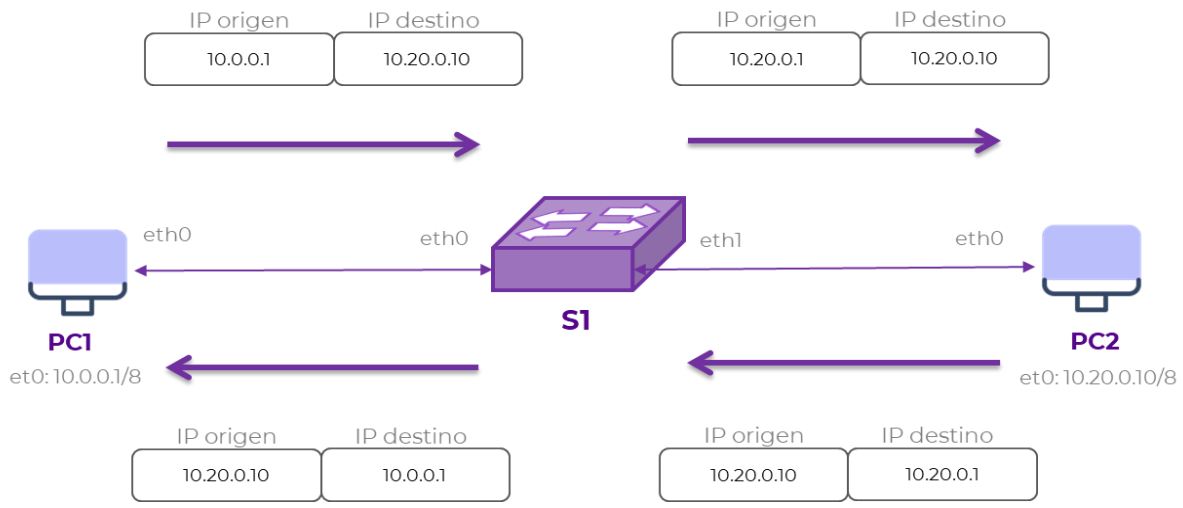
A continuación, también mostramos la captura de tráfico entre el conmutador y PC2 durante las pruebas:

	Time	Source	Destination	Protocol	Info
1	0.000000	00:00:00_00:00:01	Broadcast	ARP	Who has 10.0.1.1? Tell 10.0.0.1
2	0.002379	00:00:00_00:00:02	00:00:00_00:00:01	ARP	10.0.1.1 is at 00:00:00:00:00:02
3	0.021702	10.0.0.1	10.0.1.1	ICMP	Echo (ping) request id=0x02ff, seq=1/256, ttl
4	0.023793	10.0.1.1	10.0.0.1	ICMP	Echo (ping) reply id=0x02ff, seq=1/256, ttl
5	1.009910	10.0.0.1	10.0.1.1	ICMP	Echo (ping) request id=0x02ff, seq=2/512, ttl
6	1.015329	10.0.1.1	10.0.0.1	ICMP	Echo (ping) reply id=0x02ff, seq=2/512, ttl
7	5.036057	00:00:00_00:00:02	00:00:00_00:00:01	ARP	Who has 10.0.0.1? Tell 10.0.1.1
8	5.043968	00:00:00_00:00:01	00:00:00_00:00:02	ARP	10.0.0.1 is at 00:00:00:00:00:01

3 Mejoras para el proyecto

3.1 Implementación de NAT

El objetivo es desarrollar un programa en P4 que permita a un conmutador implementar el siguiente escenario simplificado de traducción de direcciones de red (Network Address Traducción o NAT):



El conmutador modificará la dirección IP de origen de un paquete procedente del host PC1, y si el paquete proviene del host PC2, el conmutador modificará la dirección IP de destino. Se propone implementar la lógica a través de dos tablas, una primera que se encargue de la traducción de direcciones y una segunda que se encargue del reenvío del tráfico.

Se debe comprobar que se ha conseguido el comportamiento indicado. Realice y documente las pruebas necesarias para demostrar su funcionamiento correcto. Explique y justifique lo observado.

3.2 Implementación libre

Se deberá definir, diseñar, implementar y probar (explicando y justificando el funcionamiento resultado) un programa en P4 con el procesamiento que estime más oportuno y haciendo uso de al menos dos tablas de match-action.

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