Redes Definidas por Software
Data plane - tutorial

Version: 1

Departamento de Informática
Universidade do Minho
Fev 2024

João Fernandes Pereira

---

1 Goal

The goal of this tutorial is to introduce you to Data plane programming with Programming Protocol-independent Packet Processors (P4), a domain-specific language for network devices, specifying how data plane devices (switches, NICs, routers, filters, etc.) process packet.

2 System Requirements

- Operative System = ubuntu family – Ubuntu, Xubuntu, Mint, Pop ...
- 8.5GB of free space

3 Required Software

- git, iperf3, Wireshark, Mininet
- p4c
- behavioral model version 2 (bmv2)

4 Software

This section details the required software and how to install it. One important note is that this guide is made only for *ubuntu* systems. The installation of *mininet* is not covered in this document, but you can check it here.

4.1 Setup for installation

```
$ mkdir ~/opt && cd ~/opt
$ echo "export PATH=~/.local/bin:$PATH" >> ~/.bashrc
$ source ~/.bashrc
```

All software, p4c and bmv2, should be installed in ~/opt.

4.2 Install dependencies (p4c + bmv2)

```
$sudo apt install make cmake g++ git automake libtool libgc-dev bison flex \
   libfl-dev libboost-dev libboost-iostreams-dev libbpf-dev\
   libboost-graph-dev llvm pkg-config python3 python3-pip \
   tcpdump python3-protobuf protobuf-compiler libprotoc-dev libprotobuf-dev\
   libgmp-dev libpcap-dev libboost-test-dev libboost-program-options-dev \
   libboost-system-dev libboost-filesystem-dev libboost-thread-dev \
   libevent-dev libtool flex bison libssl-dev unzip curl
```

4.3 Behavioral Model version 2 - bmv2

Behavioral Model version 2 is the second version of the reference P4 software switch, a software switch that runs P4 programs. The software switch takes as input a JSON file generated from your P4 program by a P4 compiler and interprets it to implement the packet-processing behavior specified by that P4 program.

bmv2 is not meant to be a production-grade software switch. Instead, it is intended to be used to develop, test, and debug P4 software. As such, the performance of bmv2 - throughput and latency - is significantly less than that of a production-grade software switch like Open vSwitch.

4.3.1 Installation

Based on: `https://github.com/p4lang/behavioral-model` [1]

Clone repo

[in ~/opt]

```
$ git clone https://github.com/p4lang/behavioral-model.git
```

Install Thrift server - for run time communication with the software switch

```
$ cd behavioral-model/ci
$ sudo ./install-thrift.sh
```

Install nanomsg library - for log messaging

[in ~/opt/behavioral-model/ci]

```
$ sudo ./install-nanomsg.sh
```

Install nnpy - python bindings for nanomsg, allows the usage of nanomsg in mininet CLI

[in ~/opt/behavioral-model/ci]

```
$ sudo ./install-nnpy.sh
```

Refresh the shared library cache

```
$ sudo ldconfig
```

Building bmv2

```
$ cd ~/opt/behavioral-model
$ ./autogen.sh
$ ./configure
$ make
$ sudo make install
```

Refresh the shared library cache - again

```
$ sudo ldconfig
```

4.4 P4 and P4 Compiler - p4c

p4c is a reference compiler for the P4 programming language. Before P4, vendors had total control over the functionality supported in the network. And since hardware architecture determined much of the possible behavior, hardware vendors controlled the rollout of new features (e.g., VXLAN), and rollouts took years.

P4 turns the traditional model on its head. Application developers and network engineers can now use P4 to implement specific behavior in the network, and changes can be made in minutes instead of years. Therefore, P4 is the language that allows us to program the Data plane in SDNs.

A P4 program (prog.p4) classifies packets by header and the actions to take on incoming packets (e.g., forward, drop).

P4 programs and compilers are target-specific. For example, the target can be hardware-based (FPGA, Programmable ASICs) or software (running on x86 - BMv2).

A P4-programmable switch differs from a traditional switch in two essential ways:

- The data plane functionality is not fixed in advance but is defined by a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program, and has no built-in knowledge of existing network protocols.
- The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

4.4.1 Installation

From: `https://github.com/p4lang/p4c` [2]

For the installation of p4c you have two options.

Option 1 - from package.
Only works for ubuntu 20.04, 21.04, 21.10, 22.04

```
$ . /etc/os-release
$ echo \
  "deb http://download.opensuse.org/repositories/home:/p4lang/xUbuntu_${VERSION_ID}/ /" \
  | sudo tee /etc/apt/sources.list.d/home:p4lang.list
$ curl -L \
  "http://download.opensuse.org/repositories/home:/p4lang/xUbuntu_${VERSION_ID}/Release.key" \
  | sudo apt-key add -
$ sudo apt-get update
$ sudo apt install p4lang-p4c
```

Option 2 - from source.

```
$ cd ~/opt
$ git clone --recursive https://github.com/p4lang/p4c.git
$ cd p4c
$ pip3 install --user -r requirements.txt
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
$ sudo make install
```

Refresh the shared library cache - again

```
$ sudo ldconfig
```

4.5 Checking installation

Here we want to see if things are installed. Getting versions and help menus is enough to test that.

```
$ cd ~
$ simple_switch -v # mininet will invoke this to create the software switch container
$ simple_switch_CLI -h # you will use this to populate tables
$ p4c-bm2-ss --version # to compile your p4 code
```

If you obtain responses, version numbers or help menus, for the commands above, everything should be OK.

5 Software components - Overview

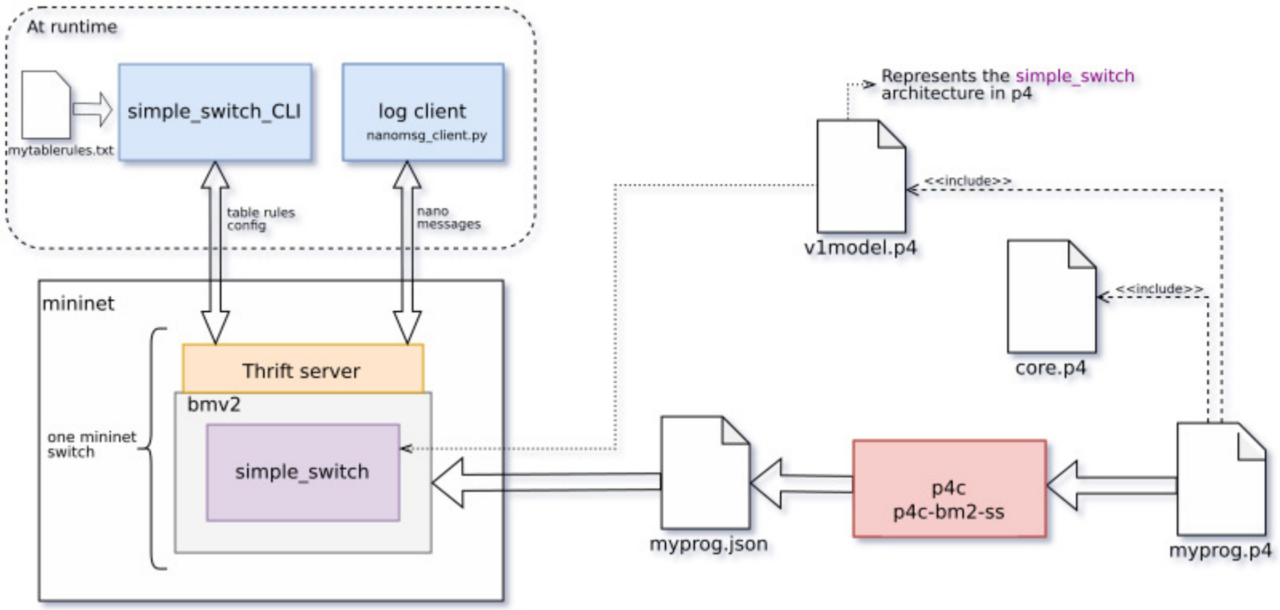


Figure 1: Relation between software components.

Figure 1 shows the relation of the installed software. A simple workflow can be described as:

1. write your mininet topology script, it receives as argument the path of the *myprog.json*.
2. write your p4 program, *myprog.p4*.

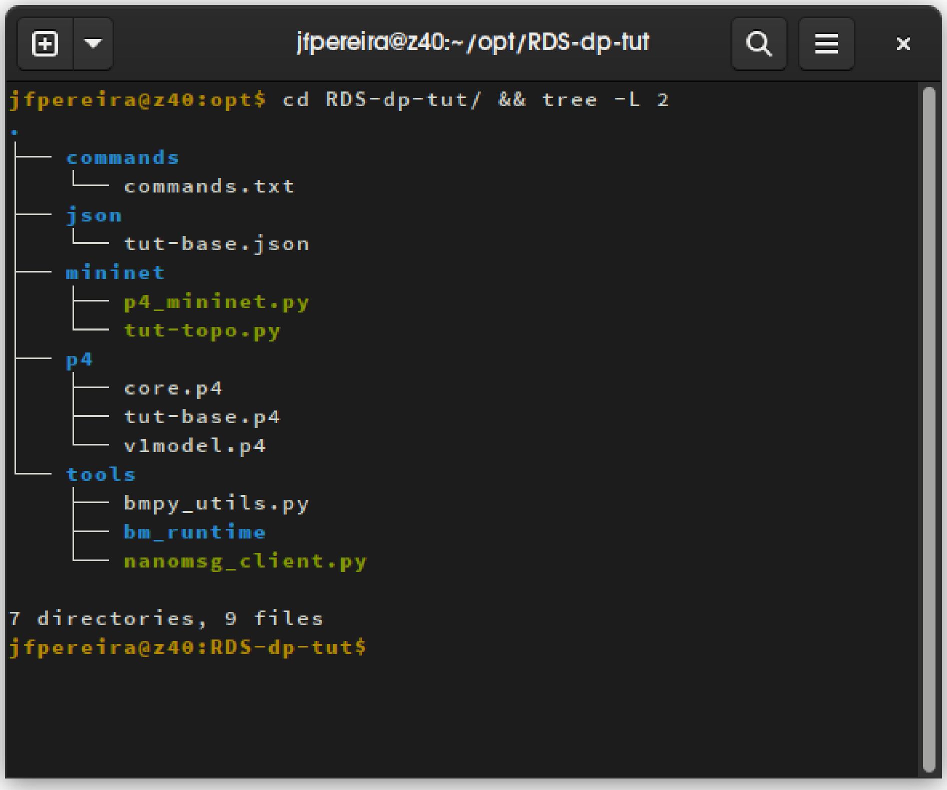
3. compile your p4 program, the output is *myprog.json*.
4. write your table entries in a txt doc, *mytablerules.txt*.
5. run the mininet topology script, with the path to the *myprog.json* as argument (or hard-coded).
6. in a terminal run the log client *nanomsg_client.py* to see the logs made by the software switch.
7. in a terminal, use the *simple_switch_CLI* program to add your table entries to the software switch.

In this document a tutorial will guide through the steps presented above. The tutorial covers the development of a simple router, but before we do that you need a workspace.

6 Workspace

The tutorial will use some relative paths. Specifying the workspace as is here instructed is one less source of errors.

```
$ cd ~
$ git clone https://github.com/jfpereira-uminho/RDS-dp-tut.git
$ rm -rf RDS-dp-tut/.git # remove this git repo, you may want to create your own
$ cd RDS-dp-tut/ && tree -L 2
```


jfpereira@z40:~/opt/RDS-dp-tut

```
jfpereira@z40:opt$ cd RDS-dp-tut/ && tree -L 2
.
├── commands
│   └── commands.txt
├── json
│   └── tut-base.json
├── mininet
│   ├── p4_mininet.py
│   └── tut-topo.py
├── p4
│   ├── core.p4
│   ├── tut-base.p4
│   └── v1model.p4
└── tools
    ├── bmpy_utils.py
    ├── bm_runtime
    └── nanomsg_client.py

7 directories, 9 files
jfpereira@z40:RDS-dp-tut$
```

Figure 2: $ cd RDS-dp-tut/ && tree

Figure 2 show the result of the command *$ cd RDS-dp-tut/ && tree*. You should have the same result.

- **commands/commands.txt** – here, you will write your table entries.
- **json/** – here, you will store the output of the p4 compiler.
- **mininet/p4_mininet.py** – for import reasons, it contains the P4Host and P4Switch class. Friendly classes for the integration of bmv2 with mininet.

- **mininet/tut-topo.py** – the mininet topology for this Tutorial.
- **p4/core.p4** – defines some standard data-types and error codes. This doesn't need to be here, is just for consulting.
- **p4/tut-base.p4** – our p4 program for this tutorial.
- **p4/v1model.p4** – the representation of bmv2-simple_switch architecture in p4. This allows us to write a p4 program capable of running in that same architecture, capable of running inside mininet. This doesn't need to be here, is just for consulting.
- **tools/nanomsg_client.py** – a nano message client to check the switch logs at runtime.

7 Tutorial - Simple Router

Before we start, a small disclaimer. As you learned so far, in SDNs there is a separation of the control plane and the data plane. In this document, we cover data plane development. Part of the data plane development is the implementation of how the device interacts with the controller. That topic is not covered in this document.

The focus of this tutorial is the development of the data plane of a simple router. This tutorial starts by presenting the mininet script that creates the topology. It just identifies what differs from the previous mininet scripts that you wrote. The tutorial then introduces you to the p4 language via the v1model. There, it covers the definition of headers, creation of parses, actions, and tables. The tutorial then covers the creation of table entries, mainly their syntax. It finishes with tests.

7.1 Mininet

The code given to you in *~/RDS-dp-tut/mininet/tut-topo.py* is complete and ready to be used. That doesn't mean that you don't have to understand it. The code is comprehensively commented. You should read that code and comments. Besides the parser, there are two classes that you have never seen, P4Host and P4Switch. These classes are defined in *~/RDS-dp-tut/mininet/p4_mininet.py*.

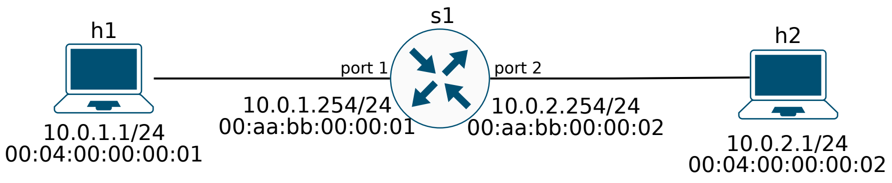


Figure 3: Topology for this tutorial.

The P4Switch can receive a few arguments, but only two are mandatory. It needs to receive the architecture model, var *args.behavioral_exe* which has the default value of *simple_switch*. It would be best if you did not change this. The other mandatory argument is the path to the JSON file (the output of the compilation of your p4 program), var *args.json*. Its value is passed as an argument when running the script. The var *args.thrift_port* refers to the port number where the Thrift server will run. A small note on the Thrift server is that we need a Thrift server per P4Switch. Here we will have no problem because we are only creating one P4Switch. Otherwise, your script must use different ports for which switch.

We want to create a simple router. To avoid dealing with ARP requests, the script populates the ARP table of the hosts with the <IP, MAC> addresses of the default route.

```
h.setARP(sw_addr[n], sw_mac[n])
h.setDefaultRoute("dev eth0 via %s" % sw_addr[n])
```

The topology created by this script is shown in Figure 3.

7.2 P4 - V1Model

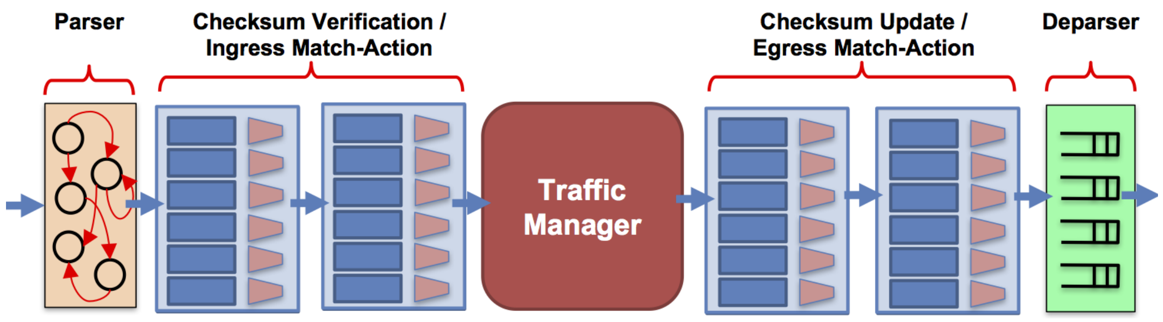

Parser
Checksum Verification / Ingress Match-Action
Traffic Manager
Checksum Update / Egress Match-Action
Deparser

Figure 4: V1Model architecture, [3].

The V1Model consists of six P4 programmable components:

- Parser
- Checksum verification control block
- Ingress Match-Action processing control block
- Egress Match-Action processing control block
- Checksum update control block
- Deparser

Figure 4, presents the blocks described above.

P4.org has defined this target architecture in a file called *v1model.p4* and added support for it to the open source P4 front end compiler called p4c. This allows p4c to compile P4 programs written for the V1Model into a json file that can be used to configure the BMV2 software switch, in our case the *bmv2 simple_switch*.

You can find additional information at [4] (bmv2 simple switch documentation), and information about $P4_{16}$ Language Specification [5]

7.3 Development of simple router p4

For the development of our simple router, you will use the base code given to you in *~/RDS-dp-tut/p4/tut-base.p4*. The code has been comprehensively commented. It will help if you read it.

Lets make a copy of that file and work on it.

```
$ cp ~/RDS-dp-tut/p4/tut-base.p4 ~/RDS-dp-tut/p4/tut-simple-router.p4
```

Now lets program some of the blocks presented in Figure 4.

7.3.1 Parser

The first block that we want to program is the parser block. But first, we need to define what we want to parse. Which protocol headers are helpful to us in order to make a simple router. Let's exclude ICMP. Our simple router will not answer to pings. Therefore, we only need **Ethernet** and **IPv4** headers. Let's remove what we don't need. You can remove everything **TCP**, the *header tcp_t* and from the *headers* struct.

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

Now let's begin with the parser. Our parser is called *MyParser* and is explicitly defined by the keyword *parser*. A P4 parser describes a state machine with one start state and two final states. The start state is always named *start*. The two final states are named *accept* (indicating successful parsing) and *reject* (indicating a parsing failure).

The parser reads its input from a *packet_in*, which is a pre-defined object that represents an incoming packet, declared in the core.p4. The parser writes its output (the out keyword) into the *headers* argument. The *metadata* and *standard_metadata_t* arguments represent structures capable of passing information between blocks. The first one is user-defined, the second one is defined by the v1model. The keyword *inout* indicates that these parameters are both an input and an output.

We already have a parser state for the Ethernet header. *state parse_ethernet*. Let's create the *state parse_ipv4* where we are going to extract the IPv4 header.

In Listing 1, we can see that if the Ethernet type is IPv4, then the *state parse_ipv4* is called and the IPv4 header is extracted from the package. Add this code (line 3 to 14) to your parser.

```
1  const bit<16> TYPE_IPV4 = 0x800;
2  ...
3  state parse_ethernet {
4     packet.extract(hdr.ethernet); // extract function populates the ethernet header
5     transition select(hdr.ethernet.etherType) {
6        TYPE_IPV4: parse_ipv4;
7        default: accept;
8     }
9  }
10 state parse_ipv4 {
11    packet.extract(hdr.ipv4); // extract function populates the ipv4 header
12    transition accept;
13 }
```

Listing 1: Parser.

7.3.2 Ingress

The Ingress block, or top pipeline, is a *control* block, which means that it is used to control the flow, manipulate and transform the headers. The body of a control block resembles a traditional imperative program, but like a parser block, they have no return values. In this Ingress block we will define two types of core components of a control block. Actions and Tables. Actions are code fragments that can read and write the data being processed. A table describes a match-action unit. Our simple router behavior is defined here by the means of actions and tables.

As we seen before in TP1, a router needs to find the next hop, change the source MAC address and change the destination MAC address. It finds the next hop with the IPv4 header, and manipulates the Ethernet header to change the MAC addresses. That is what we are going to do in the Ingress block.

Now we want to define a table that finds the next hop. To do that, the table needs to construct the following relation:

destination-IP-addr => next-hop-IP-addr   outgoing-port

Looking to our topology, Figure 3, we would have:

10.0.1.1 => 10.0.1.1   1

10.0.2.1 => 10.0.2.1   2

To understand the code in Listing 2, you should first look to the table. Basically, we are defining a table named *ipv4_lpm*. The *key* indicates which element will be used to search the table (the search key), and the search method. In this case, the key is the destination IP address, and the search method is the longest prefix match (lpm). The *actions* indicate the possible actions that an entry on this table can apply in case of a match. Remember that tables are populated by the control plane.

Now look at the code of the *action*, we can see that the action is named *ipv4_fwd* and receives as arguments the IP address of the next hop and the *egress* port, the outgoing port. These arguments are the data from the table.

The action then takes its course, saving the IP address of the next hop in the user-defined metadata *meta.next_hop_ipv4* and storing the outgoing port in the V1model metadata, *standard_metadata.egress_spec*. Finally, it decreases the *ttl* of the IPv4 Header.

You should add the code (line 5 to 18) in the Listing 2 to your *MyIngrees* function.

```
1  typedef bit<9> egressSpec_t;
2  typedef bit<32> ip4Addr_t;
3  ...

5  action ipv4_fwd(ip4Addr_t nxt_hop, egressSpec_t port) {
6     meta.next_hop_ipv4 = nxt_hop;
7     standard_metadata.egress_spec = port;
8     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
9  }
10 table ipv4_lpm {
11    key = { hdr.ipv4.dstAddr : lpm; }
12    actions = {
13       ipv4_fwd;
14       drop;
15       NoAction;
16    }
17    default_action = NoAction(); // NoAction is defined in v1model - does nothing
18 }
```

Listing 2: Finding the next hop IP address.

Now we want to create a table and action capable of rewriting the Ethernet source MAC address. The outgoing packet must have the MAC address of the outgoing port as its source MAC address.

The relation that we want to build in this table is:

outgoing-port => mac-addr

According to Figure 3:

1 => 00:aa:bb:00:00:01
2 => 00:aa:bb:00:00:02

Listing 3 shows the code to achieve that.

The action *rewrite_src_mac* just modifies the *hdr.ethernet.srcAddr* value to the one that received from the table (argument *macAddr_t src_mac*). In the table *src_mac*, the variable used as search key is *standard_metadata.egress_spec*, and its value comes from the *ipv4_lpm* table, see Listing 2. The type of match, in this case, is *exact*.

You can add the code in Listing 3 to your Ingress block.

```
action rewrite_src_mac(macAddr_t src_mac) {
   hdr.ethernet.srcAddr = src_mac;
}

table src_mac {
   key = { standard_metadata.egress_spec : exact; }
   actions = {
      rewrite_src_mac;
      drop;
   }
   default_action = drop;
}
```

Listing 3: Change MAC source.

The next step of our simple router is to change the destination MAC address. The IP address of the next hop was obtained from the table *ipv4_lpm* and is stored in the user-defined metadata *meta.next_hop_ipv4*. The table in Listing 4 uses that value as a key, the action *rewrite_dst_mac* receives a MAC address and changes the Ethernet header, *hdr.ethernet.dstAddr = dst_mac*.

Therefore this table needs to create a relation between IP addresses and MAC address.

ip-addr => mac-addr

According to Figure 3:

10.0.1.1 => 00:04:00:00:00:01
10.0.2.1 => 00:04:00:00:00:02

Go ahead and add the code in Listing 4 to your Ingress block.

```
action rewrite_dst_mac(macAddr_t dst_mac) {
   hdr.ethernet.dstAddr = dst_mac;
}

table dst_mac {
   key = { meta.next_hop_ipv4 : exact; }
   actions = {
      rewrite_dst_mac;
      drop;
   }
   default_action = drop;
}
```

Listing 4: Change MAC destination.

Every block defined by the keyword *control* needs to implement the *apply* statement. There we define the conditions and order that our code should be applied. The code presented in Listing 5 shows how our *MyIngress* function should apply the tables. You can add this code.

```
apply {
   if (hdr.ipv4.isValid()) {
      ipv4_lpm.apply();
      src_mac.apply();
      dst_mac.apply();
   }
}
```

Listing 5: Apply statement.

You can find more information on the supported kinds of Table match here, [4].

You can find more information on the V1model meta fields (Standard metadata) in here, [4].

7.3.3 Deparser

In the Deparser block we simply add the headers, extrated in the Parser, to the *packet_out packet*. In our case the function *MyDeparser* should be like is shown in Listing 6

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

Listing 6: MyDeparser function.

7.4 Table Entries

In this section we will create the table entries for the following tables:

- *ipv4_lpm*
- *src_mac*
- *dst_mac*

Entry structure:

```
table_set_default <table name> <action name>
table_add <table name> <action name> <match fields> => <action parameters> [priority]
```

The file *~/RDS-dp-tut/commands/commands.txt* contains our table entries. Copy the following Listings to it.

First lets define the default behavior for each table.

```
table_set_default ipv4_lpm drop
table_set_default src_mac drop
table_set_default dst_mac drop
```

The conceptual definition of the following table entries was already explained in the Ingress section, but we can take a closer look at them. The entry:

table_add ipv4_lpm ipv4_fwd 10.0.1.1/32 => 10.0.1.1 1

can be read as:

when table *pv4_lpm* matches *10.0.1.1/32* execute the action *ipv4_fwd*, passing *10.0.1.1* and *1* as arguments.
In a more formal reading we can say that:

when the destination IP is *10.0.1.1/32* the next hop is *10.0.1.1*, the destination itself, and is connected via port *1*.

One aspect that may confuse you is the mask of the IP address in the match field, the */32*. Remember that the table *ipv4_lpm* uses longest prefix match (lpm) to find a match. As we are dealing with a terminal system, we use mask */32* to indicate that a full match is needed.

```
table_add ipv4_lpm ipv4_fwd 10.0.1.1/32 => 10.0.1.1 1
table_add ipv4_lpm ipv4_fwd 10.0.2.1/32 => 10.0.2.1 2
table_add src_mac rewrite_src_mac 1 => 00:aa:bb:00:00:01
table_add src_mac rewrite_src_mac 2 => 00:aa:bb:00:00:02
table_add dst_mac rewrite_dst_mac 10.0.1.1 => 00:04:00:00:00:01
table_add dst_mac rewrite_dst_mac 10.0.2.1 => 00:04:00:00:00:02
```

Don't live blank lines in your *commands.txt* file.

You can find more information about table entries in [6]

7.5 Emulation and Tests

Now that we have our *tut-simple-router.p4* and *commands.txt* files completed, we can try to run the emulation. First, we need to compile our p4 program.

```
$ cd ~/RDS-dp-tut
$ p4c-bm2-ss --p4v 16 p4/tut-simple-router.p4 -o json/tut-simple-router.json
```

If everything went well you should have the following file *json/tut-simple-router.json*

Now let's run our mininet script

```
$ cd ~/RDS-dp-tut
$ sudo python mininet/tut-topo.py --json json/tut-simple-router.json
```

and let's check if everything is OK, compare with Figure 3:

- h1 and h2 must have the correct default route
- h1 and h2 must have the correct ARP table entry
- s1 must have the correct MAC addr on each interface

```
mininet> xterm h1 h2 s1
"Node:h1": route -n
"Node:h1": arp -n
"Node:h2": route -n
"Node:h2": arp -n
"Node:s1": ifconfig
```

Now, in a new terminal lets capture the nano messages of our software switch (logs).

```
 $ cd ~/RDS-dp-tut/tools/
 & sudo ./nanomsg_client.py --thrift-port 9090
```

In the mininet CLI do a ping and see the logs in the new terminal window:

```
 mininet> h1 ping h2
```

If you stop the ping (CTRL-C), and check the logs, you will see that the switch is dropping all the packets (MyIngress.drop)

Now let's inject our table entries. Open new terminal and execute the following:

```
$ cd ~/RDS-dp-tut/commands/
$ simple_switch_CLI --thrift-port 9090 < commands.txt
```

Once again, on the mininet CLI do a ping and see the logs:

```
mininet> h1 ping h2
```

In the logs, you will see all of your table actions being executed (MyIngress.ipv4_fwd), (MyIngress.rewrite_src_mac) and (MyIngress.rewrite_dst_mac).

As a final test, let's check if the TCP traffic is forwarded correctly. For that lets run an **iperf3** server on h2, on port 5555, and a client on h1.

```
"Node:h2": iperf3 -s -p 5555
"Node:h1": iperf3 -c 10.0.2.1 -p 5555
```

If you capture the traffic with **wireshark**, at interface **s1-eth1**, you will see many TCP Retransmissions, it is not a problem. Another think that you will see is that the **iperf3** server always uses the port 5555, but the client uses at least two random port numbers. This observation is handy for what comes next.

References

[1] *Behavioral Model (bmv2)*. [Online]. Available: https://github.com/p4lang/behavioral-model (visited on 05/18/2023).

[2] *P4c P4 Compiler*. [Online]. Available: https://github.com/p4lang/p4c (visited on 05/18/2023).

[3] *Working with P4 in Mininet on BMV2*. [Online]. Available: https://usi-advanced-networking.github.io/deliverables/p4-mininet/ (visited on 05/18/2023).

[4] *Simple Switch (bmv2)*. [Online]. Available: https://github.com/p4lang/behavioral-model/blob/f16d0de3486aa7fb2e1fe554aac7d237cc1adc33/docs/simple_switch.md (visited on 05/18/2023).

[5] *P4 v16 Language Specification*. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.2.html (visited on 05/18/2023).

[6] *Runtime CLI (bmv2)*. [Online]. Available: https://github.com/p4lang/behavioral-model/blob/f16d0de3486aa7fb2e1fe554aac7d237cc1adc33/docs/runtime_CLI.md (visited on 05/18/2023).