Network Virtualization TP3 - Data plane

Erikson Tomas$^{[pg44102]}$, Eduardo Cunha$^{[pg45515]}$, Miguel Peixoto$^{[pg45517]}$, and I-Ping Lin$^{[pg41017]}$

Engineering of Computer Networks and Telematic Services, School of Engineering,
University of Minho, Braga, Portugal

Abstract. This assignment is about data plane programming that uses a packet processor (P4) which is the domain-specific language of the network device and programs how the data plane device such as network interface cards, switches, filters, and routes process the packets. We will follow the tutorial provided by the teacher and answer a firewall exercise.

Keywords: Network Virtualization · Software Defined Networks · SDN · OpenFlow · OVSwitch · Programming Protocol independent Packet Processors · P4 · Wireshark · Mininet · iperf3 · p4c.

1 Introduction

This assignment is a data plane programming that uses a packet processor (P4) that does not depend on the programming protocol, which is the domain-specific language of the network device and programs how the data plane device such as network interface cards, switches, filters, and routes process the packets. To develop this work, it was first necessary to prepare the environment with some required tools (git, iperf3, Wireshark, Mininet, p4c, and behavioral model version 2.

Programmable forwarding can be seen as a natural evolution of the Software-Defined Network (SDN), where software that describes the behavior of how packets are processed can be designed, tested, and deployed in a much shorter time frame by operators, engineers, researchers, and professionals in general. The de facto standard for defining forwarding behavior is the P4 language, which stands for Programming Protocol-independent Packet Processors [1]. Essentially, P4 programmable switches removed the barrier to entry to network design, previously reserved for network vendors.

This subsection demonstrates the P4 workflow that will be used in this work. We used the Visual Studio Code (VS Code) as the editor to modify the tp3-base.p4 program. Then, we used the p4c compiler with the V1Model architecture to compile the user-supplied P4 program (tp3-base.p4). The compiler generates a JSON output (i.e., firewall.json) which will be used as the data plane program by the switch daemon (i.e., simple-switch). Finally, we used the simple-switch-CLI at runtime to populate and manipulate table entries in our P4 program. The target switch (vendor supplied) used in this work for testing and debugging P4 programs is the behavioral model version 2 (BMv2).

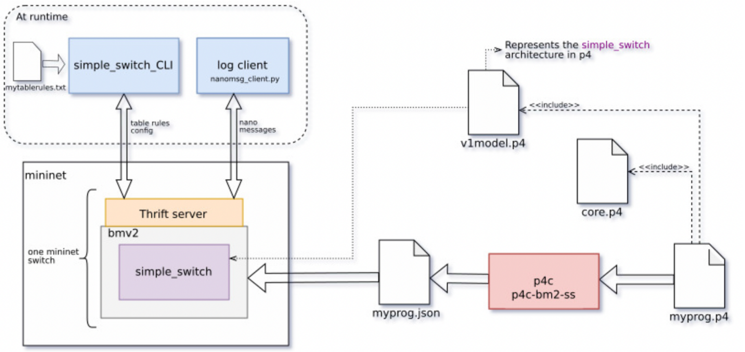


Fig. 1. Relation between software components.

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

- write mininet topology script, it receives as argument the path of the myprog.json.
- write p4 program, myprog.p4.
- compiler p4 program, the output is myprog.json.
- write table entries in a txt doc, mytablerules.txt.
- run the mininet topology script, with the path to the myprog.json as argument (or hard-coded).
- in a terminal run the log client nanomsg-client.py to see the logs made by the software switch.
- in a terminal, use the simple_switch_CLI program to add table entries to the software switch

2 Firewall

2.1 Implementation

We start by installing the required programs according to the tutorial available in Practice Statement 3 (TP3). Then we checked if everything was ok, running in the terminal the following commands.

```
simple_Switch –v && p4c–bm2—ss —version
```

Beforehand we want to mention that tp3-simple-router.P4 and tp3-simple-router.json files are not part of this exercise. We prefer to leave them because they served as a starting point for the realization of this work.

```
cd rds_tp3_g11 && tree .
```

- commands/commands.txt – here, we'll write our table entries;

- json/ – here, we'll store the output of the p4 compiler; It means after writing our firewall.p4 we'll compile the file (firewall.p4) and the output will be the file with extension .json;
- mininet/p4_mininet.py – here, contains the P4Host and P4Switch class; Friendly classes for the integration of bmv2 with mininet;
- mininet/tp3-topo.py – Here contains the ready topology written by the teacher ;
- p4/core.p4 – defines some standard data types and error codes. This file is just for consulting;
- 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;
- tools/nanomsg_client.py – a nano message client to check the switch logs at runtime.

The topology used in this work is shown in figure 2. It was given by the teacher and it was only necessary to understand it.

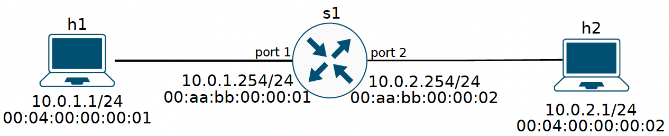


Fig. 2. Topology

Our starting point for accomplishing this task was the tutorial. After having done it little by little, we had practically walked done. To complete our exercise, we first made a copy of the tutorial executed in the terminal command: cp ~/rds_tp3_g11/p4/simple-router.p4 /rds_tp3_g11/p4/tp3-firewall.p4

After that, we add some missing fields between them:

- header tcp
- parse tcp header
- table — action and apply
- Deparser tcp

Since the added parameters are already in the sentence statement, we will not detail what each one does in the report. But it should be noted that the code is also commented on.

2.2 Table Entries

Tables and input are defined in the Ingress field. We created the four (4) table entries for the following tables:

- table ipv4_lpm
- table src_mac
- table dst_mac
- table firewall_tcp

We would like to mention that the first three tables were taken advantage of the tutorial and we had only the need to add a new table that we called firewall_tcp. This table received 4 keys as arguments.

The keys of the table serve as a search and the methods of it. They contain the source address, the destination address, and two values between src_range and dst_range. In addition, to define these key parameters we must apply them.

To modify the entries we create an extension file .txt. We begin by adding the existing ones in the teacher's tutorial and therefore add the entries for our table (firewall_tcp). We start by setting the tables by default to drop all tables. Then we apply the same tables with other steels that we wanted as shown in the figure

```
reset_state
table_set_default ipv4_lpm drop
table_set_default src_mac drop
table_set_default dst_mac drop
table_set_default firewall_tcp drop
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
table_add firewall_tcp NoAction 10.0.1.1 10.0.2.1/32 1->65535 5555->5555 => 1
table_add firewall_tcp NoAction 10.0.2.1 10.0.1.1/32 5555->5555 1->65535 => 1
```

2.3 Tests

First, we need to compile our P4 program. For this, we run in the terminal the following command:

```
p4c-bm2-ss --p4v 16 p4/tp3-firewall.p4 -o json/tp3_firewall.json
```

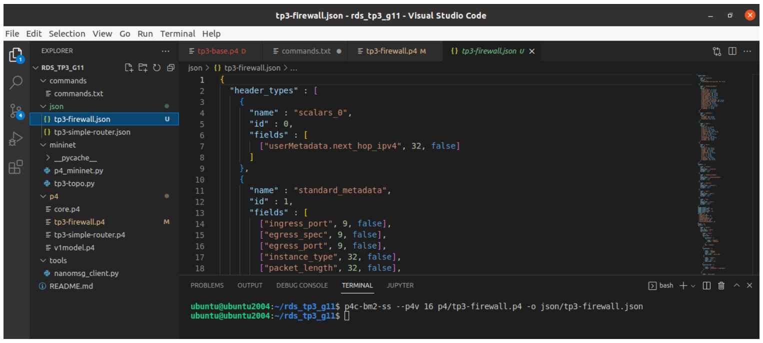
As a result of the command, we get the file firewall.json as we can see in the figure 3

Next, we run the mininet script. To do that, we run in the terminal the following command:

```
sudo python3 mininet/tp3-topo.py --json json/tp3-firewall.json
```

The output of this command opened a terminal of mininet. In terminal we opened 3 xterms:

```
xterm h1 h2 s1
```



```
{
  "header_types" : [
    {
      "name" : "scalars_0",
      "id" : 0,
      "fields" : [
        ["userMetadata.next_hop_ipv4", 32, false]
      ]
    },
    {
      "name" : "standard_metadata",
      "id" : 1,
      "fields" : [
        ["ingress_port", 9, false],
        ["egress_spec", 9, false],
        ["egress_port", 9, false],
        ["instance_type", 32, false],
        ["packet_length", 32, false],
```

```
ubuntu@ubuntu2004:~/rds_tp3_g11$ p4c-bm2-ss --p4v 16 p4/tp3-firewall.p4 -o json/tp3-firewall.json
ubuntu@ubuntu2004:~/rds_tp3_g11$ 
```

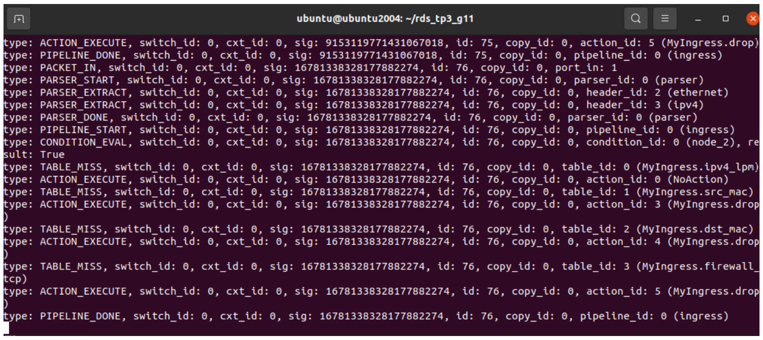
Fig. 3. firewall.json

In the terminals we can check the information, that is, in H1 and H2 we can see the respective mac addresses and on the switch, it is possible to confirm these addresses in interfaces 1 and 2.

Next, we will observe the nano messages of the switch. For this we open another terminal and execute the following command:

```
sudo ./nanomsg_client.py --thrift-port 9090
```

At the mininet terminal, we'll try to ping from H1 to H2. As a result, we return to the terminal we ran earlier and we can see the switch dropping all packets (Myingress.drop).



```
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 9153119771431067018, id: 75, copy_id: 0, action_id: 5 (MyIngress.drop)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 9153119771431067018, id: 75, copy_id: 0, pipeline_id: 0 (ingress)
type: PACKET_IN, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, port_in: 1
type: PARSER_START, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, parser_id: 0 (parser)
type: PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, header_id: 2 (ethernet)
type: PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, header_id: 3 (ipv4)
type: PARSER_DONE, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, parser_id: 0 (parser)
type: PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, condition_id: 0 (node_2), re
sult: True
type: TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, table_id: 0 (MyIngress.ipv4_lpm)
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, action_id: 0 (NoAction)
type: TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, table_id: 1 (MyIngress.src_mac)
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, action_id: 3 (MyIngress.drop
)
type: TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, table_id: 2 (MyIngress.dst_mac)
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, action_id: 4 (MyIngress.drop
)
type: TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, table_id: 3 (MyIngress.firewall_
tcp)
type: ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, action_id: 5 (MyIngress.drop
)
type: PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 16781338328177882274, id: 76, copy_id: 0, pipeline_id: 0 (ingress)
```

Fig. 4. Packet Dropping by switch

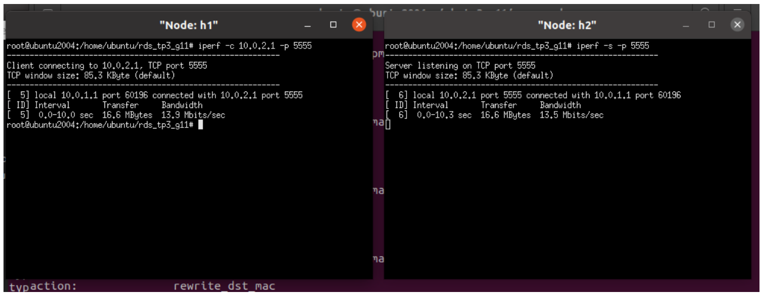
Now we'll inject our table entries. To do that, we'll execute the following command on the terminal:

```
simple_switch_CLI --thrift-port 9090 < commands.txt
```

Let's then check if the parameters defined are with the desired behavior using iperf and the log client nanomsg for that verification opening two xterms on the mininet terminal [xterm h1 h2].

```
h2: iperf —s —p 5555
h1: iperf —c 10.0.2.1 —p 5555
```

Figure 5 shows that the switch allows tcp traffic through port 5555.



```
root@ubuntu2004:/home/ubuntu/rds_tp3_g11# iperf -c 10.0.2.1 -p 5555
------------------------------------------------------------
Client connecting to 10.0.2.1, TCP port 5555
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  5] local 10.0.1.1 port 60196 connected with 10.0.2.1 port 5555
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec  16.6 MBytes  13.9 Mbits/sec
root@ubuntu2004:/home/ubuntu/rds_tp3_g11#
```

```
root@ubuntu2004:/home/ubuntu/rds_tp3_g11# iperf -s -p 5555
------------------------------------------------------------
Server listening on TCP port 5555
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.2.1 port 5555 connected with 10.0.1.1 port 60196
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.3 sec  16.6 MBytes  13.5 Mbits/sec
```

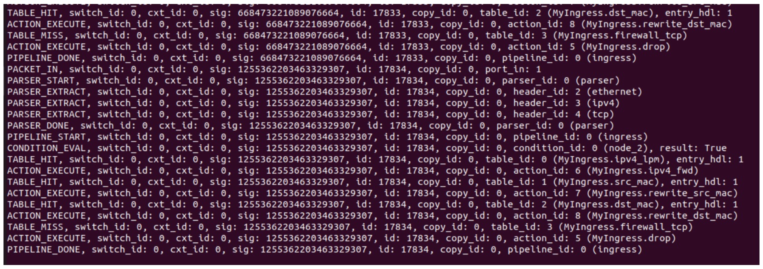
```
action:                rewrite_dst_mac
```

Fig. 5. Iperf tcp traffic port 5555

Through the nanomsg we can notice that the package with ID: 17813 has the Noaction action action and the package is Deparser.

Now let's try again with a different port.

```
h2: iperf —s —p 3773
h1: iperf —c 10.0.2.1 —p 3773
```



```
TABLE_HIT, switch_id: 0, cxt_id: 0, sig: 668473221089076664, id: 17833, copy_id: 0, table_id: 2 (MyIngress.dst_mac), entry_hdl: 1
ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 668473221089076664, id: 17833, copy_id: 0, action_id: 8 (MyIngress.rewrite_dst_mac)
TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 668473221089076664, id: 17833, copy_id: 0, table_id: 3 (MyIngress.firewall_tcp)
ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 668473221089076664, id: 17833, copy_id: 0, action_id: 5 (MyIngress.drop)
PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 668473221089076664, id: 17833, copy_id: 0, pipeline_id: 0 (ingress)
PACKET_IN, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, port_in: 1
PARSER_START, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, parser_id: 0 (parser)
PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, header_id: 2 (ethernet)
PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, header_id: 3 (ipv4)
PARSER_EXTRACT, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, header_id: 4 (tcp)
PARSER_DONE, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, parser_id: 0 (parser)
PIPELINE_START, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, pipeline_id: 0 (ingress)
CONDITION_EVAL, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, condition_id: 0 (node_2), result: True
TABLE_HIT, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, table_id: 0 (MyIngress.ipv4_lpm), entry_hdl: 1
ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, action_id: 6 (MyIngress.ipv4_fwd)
TABLE_HIT, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, table_id: 1 (MyIngress.src_mac), entry_hdl: 1
ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, action_id: 7 (MyIngress.rewrite_src_mac)
TABLE_HIT, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, table_id: 2 (MyIngress.dst_mac), entry_hdl: 1
ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, action_id: 8 (MyIngress.rewrite_dst_mac)
TABLE_MISS, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, table_id: 3 (MyIngress.firewall_tcp)
ACTION_EXECUTE, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, action_id: 5 (MyIngress.drop)
PIPELINE_DONE, switch_id: 0, cxt_id: 0, sig: 1255362203463329307, id: 17834, copy_id: 0, pipeline_id: 0 (ingress)
```

Fig. 6. Dropped trafic port 3737

Through the figure 6 we can notice that the package with ID: 17834 has the Drop action action and the pipeline is terminated without forwarding.

3 Conclusion

This practical work was an introduction to the Data Plane programming in Software Defined Networks. Although the proposed firewall exercise was simple and short, the main challenge was to set up the environment and understand all the components of the P4. Without the tutorial provided by the teacher, this work would have been much more difficult and would have taken more time to complete. Overall, we found this exercise to be a good introduction to data plane programming.

References

1. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., ... & Walker, D. (2014). P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3), 87-95.