



## **INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES**

### **Lab 5: Introduction to Match-action Tables (Part 1)**

Document Version: **01-25-2022**



Award 2118311

"CyberTraining on P4 Programmable Devices using an Online Scalable  
Platform with Physical and Virtual Switches and Real Protocol Stacks"

## Contents

Overview .....	3
Objectives.....	3
Lab settings .....	3
Lab roadmap .....	3
1 Introduction to control blocks .....	3
1.1 Tables .....	4
1.2 Match types.....	4
1.3 Exact match .....	4
2 Lab topology.....	6
2.1 Starting host h1 and host h2 .....	7
3 Defining a table with exact match lookup .....	8
3.1 Loading the programming environment.....	8
3.2 Programming the exact table in the ingress block.....	9
4 Loading the P4 program.....	15
4.1 Compiling and loading the P4 program to switch s1 .....	15
4.2 Verifying the configuration .....	16
5 Configuring switch s1 .....	17
5.1 Mapping P4 program's ports.....	17
5.2 Loading the rules to the switch .....	19
6 Testing and verifying the P4 program.....	19
References .....	23

## Overview

This lab describes match-action tables and how to define them in a P4 program. It then explains the different types of matching that can be performed on keys. The lab further shows how to track the misses/hits of a table key while a packet is received on the switch.

## Objectives

By the end of this lab, students should be able to:

1. Understand what match-action tables are used for.
2. Describe the basic syntax of a match-action table.
3. Implement a simple table in a P4.
4. Trace a table's misses/hits when a packet enters to the switch.

## Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1. Credentials to access Client machine.

Device	Account	Password
Client	admin	password

## Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction to control blocks.
2. Section 2: Lab topology.
3. Section 3: Defining a table with exact match lookup.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Testing and verifying the P4 program.

### 1 Introduction to control blocks

Control blocks are essential for processing a packet. For example, a control block for layer-3 forwarding may require a forwarding table that is indexed by the destination IP address. The control block may include actions to forward a packet when a hit occurs, and to drop

the packet otherwise. To forward a packet, a switch must perform routing lookup on the destination IP address. Figure 1 shows the basic structure of a control block.

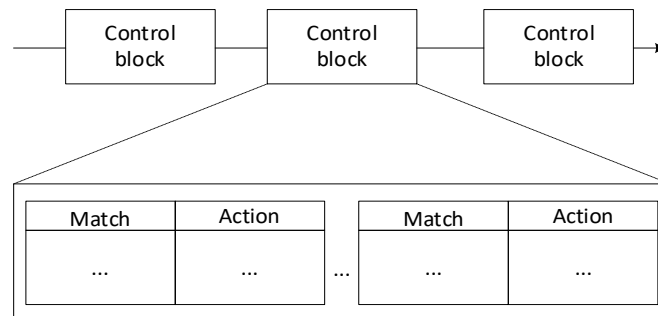


Figure 1. Control blocks.

## 1.1 Tables

Tables are essential components that define the processing behavior of a packet inside the switch. A table is specified in the P4 program and has one or more entries (rows) which are populated by the control plane. An entry contains a key, an action, and action data.

- **Key:** it is used for lookup operations. The switch builds a key for the incoming packet using one or more header fields (e.g., destination IP address) and then lookups for that value in the table.
- **Action:** once a match occurs, the action specified in the entry is performed by the arithmetic logic unit. Actions are simple operations such as modify a header field, forward the packet to an egress port, and drop the packet. The P4 program contains the possible actions.
- **Action data:** it can be considered as parameter/s used along with the action. For example, the action data may represent the port number the switch must use to forward the packet. Action data is populated by the control plane.

## 1.2 Match types

There are three types of matching: exact match, Longest Prefix match (LPM), and ternary match. They are defined in the standard library (*core.p4*<sup>1</sup>). Note that architectures may define and implement additional match types. For example, the V1Model<sup>2</sup> also has matching based on ranges and selectors. In this lab we will discuss exact match.

## 1.3 Exact match

Assume that the exact match lookup is used to search for a specific value of an entry in a table. Assume that Table 2 matches on the destination IP address. If an incoming packet has 10.0.0.2 as the destination IP address, then it will match against the second entry and the P4 program will forward the packet using port 2 as the egress port.

Table 2. Exact match table.

Key	Action	Action data
10.0.0.1	forward	port 1
10.0.0.2	forward	port 2
default	drop	

Figure 2 shows the ingress control block portion of a P4 program. Two actions are defined, `drop` and `forward`. The `drop` action (lines 5 - 7) invokes the `mark to drop` primitive, causing the packet to be dropped at the end of the ingress processing. The `forward` action (lines 8 - 10) accepts as input (i.e., action data) the destination port. This parameter is inserted by the control plane and updated in the packet during the ingress processing. In line 9, the P4 program assigns the egress port defined by the control plane to the `standard metadata` egress specification field (i.e., the field that the traffic manager looks at to determine which port the packet will be sent to). Lines 11-21 implement a table named `ipv4_exact`. The match is against the destination IP address using the exact lookup method. The actions associated with the table are forward and drop. The default action which is invoked when there is a miss is drop. The maximum number of entries a table can support is configured manually by the programmer (i.e., 1024 entries, see line 19). Note, however, that the number of entries is limited by the amount of memory in the switch.

The control block starts executing from the `apply` statement (see lines 22-26) which contains the control logic. In this program, the `ipv4_exact` table is enabled when the incoming packet has a valid IPv4 header.

```

1:  /*****INGRESS PROCESSING*****/
2:  control MyIngress(inout headers hdr,
3:                    inout metadata meta,
4:                    inout standard_metadata_t standard_metadata){
5:      action drop(){
6:          mark_to_drop(standard_metadata);
7:      }
8:      action forward(egressSpec_t port) {
9:          standard_metadata.egressSpec = port;
10:     }
11:     table ipv4_exact {
12:         key = {
13:             hdr.ipv4.dstAddr:exact;
14:         }
15:         actions = {
16:             forward;
17:             drop;
18:         }
19:         size = 1024;
20:         default_action = drop();
21:     }
22:     apply {
23:         if (hdr.ipv4.isValid()){
24:             ipv4_exact.apply();
25:         }
26:     }
27: }

```

Figure 2. Ingress control block portion of a P4 program. The code implements a match-action table with exact match lookup.

## 2 Lab topology

Let us get started with creating a simple Mininet topology using MiniEdit.

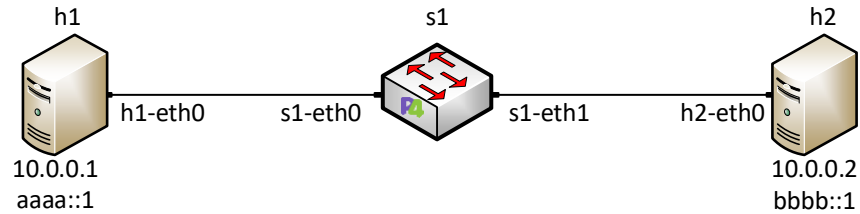


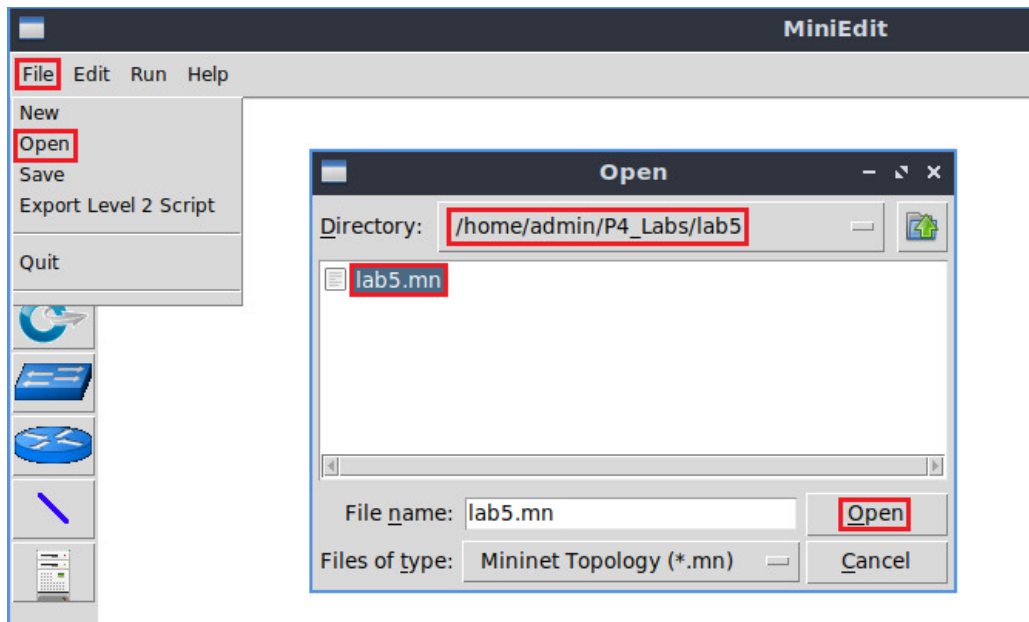
Figure 3. Lab topology.

**Step 1.** A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.



Figure 4. MiniEdit shortcut.

**Step 2.** In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab5* folder and search for the topology file called *lab5.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

Figure 5. MiniEdit's *Open* dialog.

**Step 3.** The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

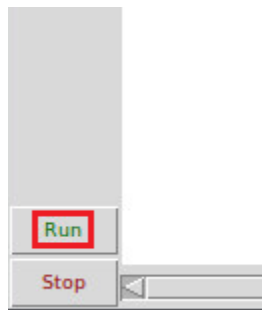


Figure 6. Running the emulation.

## 2.1 Starting host h1 and host h2

**Step 1.** Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

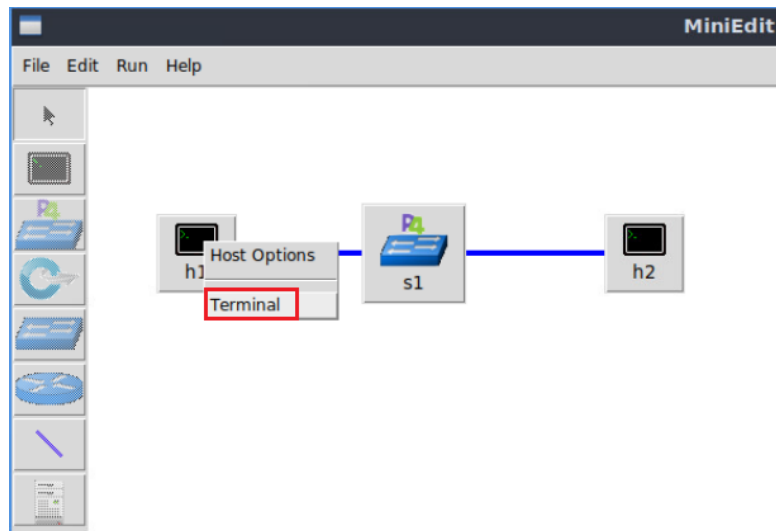


Figure 7. Opening a terminal on host h1.

**Step 2.** Test connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```

Figure 8. Connectivity test using `ping` command.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded on the switch.

### 3 Defining a table with exact match lookup

This section demonstrates how to implement a simple table in P4 that uses exact matching on the destination IP address of the packet. When there is a match, the switch forwards the packet from a certain port. Otherwise, the switch drops the packet.

#### 3.1 Loading the programming environment

**Step 1.** Launch a Linux terminal by double-clicking on the icon located on the desktop.



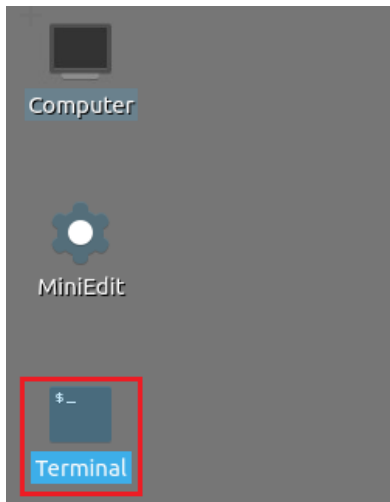


Figure 9. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI).

**Step 2.** In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab5
```

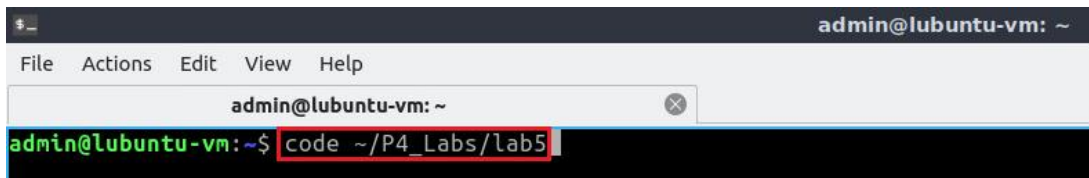


Figure 10. Launching the editor and opening the lab5 directory.

### 3.2 Programming the exact table in the ingress block

**Step 1.** Click on the *ingress.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.

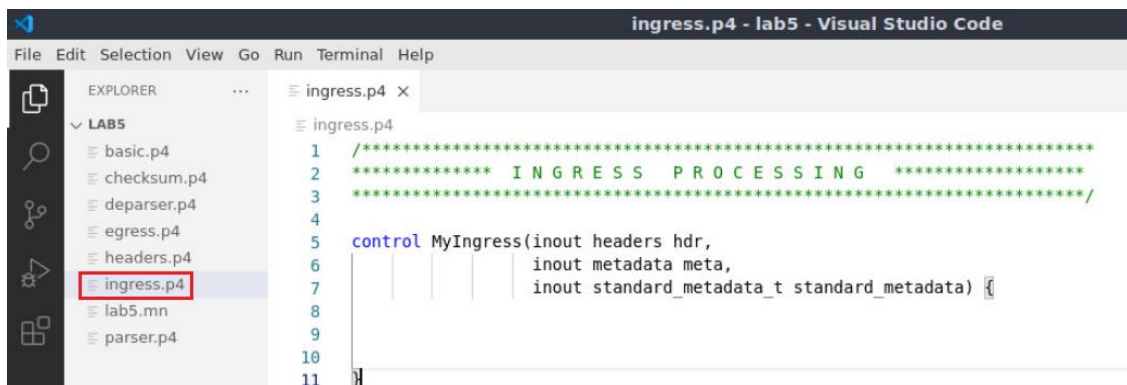


Figure 11. Opening the ingress processing block.

We can see that the *ingress.p4* declares a control block named *MyIngress*. Note that the body of the control block is empty. Our objective is to define a P4 table, its actions, and then invoke them inside the block.

**Step 2.** We will start by defining the possible actions that a table will call. In this simple forwarding program, we have two actions:

- `forward`: this action will be used to forward the packet out of a switch port.
- `drop`: this action will be used to drop the packet.

**Step 3.** Now we will define the behavior of the `forward` action. Insert the code below inside the *MyIngress* control block.

```
action forward (egressSpec_t port) {
    standard_metadata.egress_spec = port;
}
```

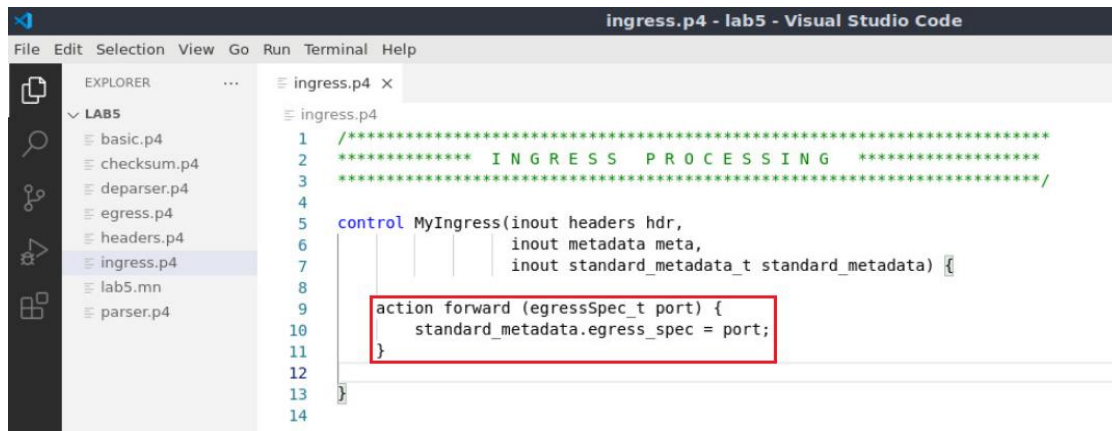


Figure 12. Defining the `forward` action.

The action `forward` accepts as parameters the port number (`egressSpec_t port`) to be used by the switch to forward the packet. Note that `egressSpec_t` is just a typedef that corresponds to `bit<9>`. It is defined in the *headers.p4* file.

The `standard_metadata` is an instance of the `standard_metadata_t` struct provided by the V1Model. This struct contains intrinsic metadata that are useful in packet processing and in more advanced features. For example, to determine the port on which a packet arrives, we can use the `ingress_port` field in the `standard_metadata`. If we want to specify the port to which the packet must be sent to, we need to use the `egress_spec` field of the `standard_metadata`.

Now that we know what `standard_metadata` is, the egress port (which will be passed through the control plane) is specified by `egress_spec` field (i.e., the port to which the packet must be sent to) of the `standard_metadata`.

In summary, when the forward action is executed, the packet will be sent out of the port number specified as parameter.

**Step 4.** Now we will define the drop action. Insert the code below inside the *MyIngress* control block.

```
action drop() {
    mark_to_drop(standard_metadata);
}
```

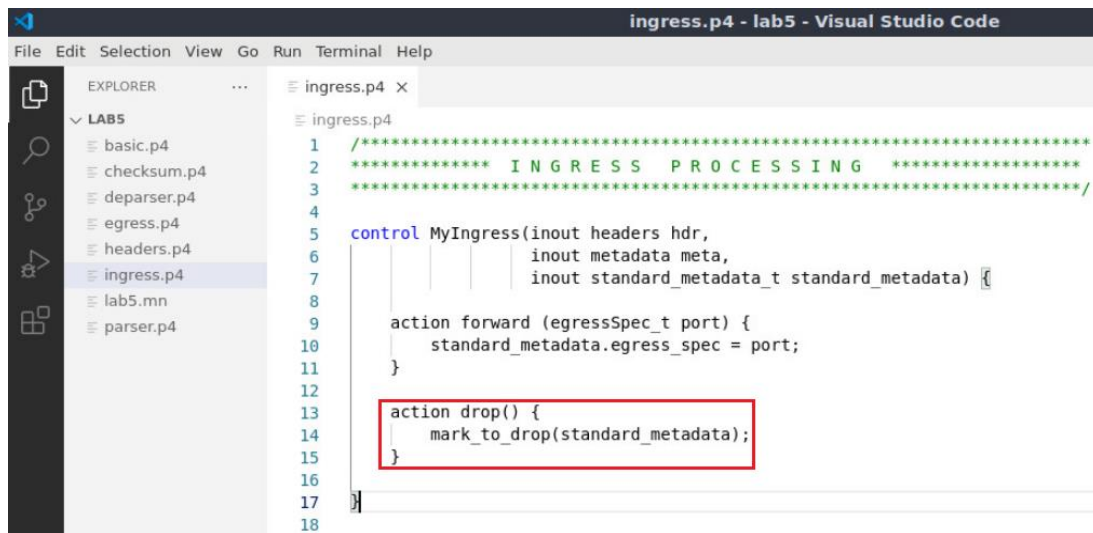
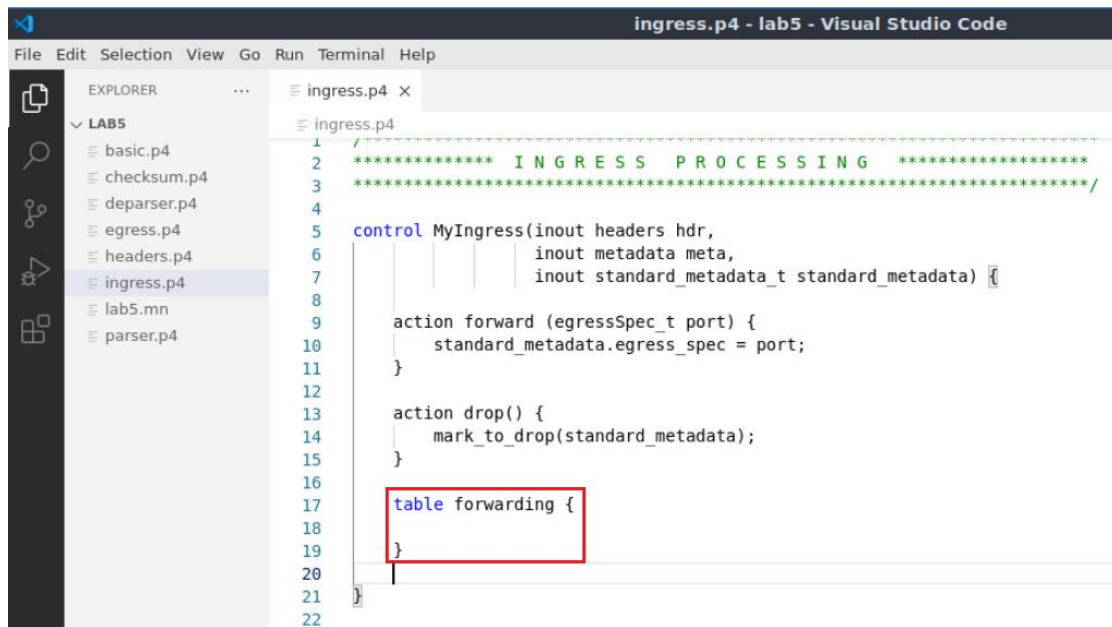


Figure 13. Defining the `drop` action.

The `drop()` action invokes a primitive action `mark_to_drop()` that modifies the `standard_metadata.egress_spec` to an implementation-specific special value that causes the packet to be dropped.

**Step 5.** Now we will define the table named `forwarding`. Write the following piece of code inside the body of the *MyIngress* control block.

```
table forwarding {
}
```

Figure 14. Declaring the `forwarding` table.

Tables require keys and actions. In the next step we will define a key.

**Step 6.** Add the following code inside the forwarding table.

```

key = {
    hdr.ipv4.dstAddr: exact;
}

```

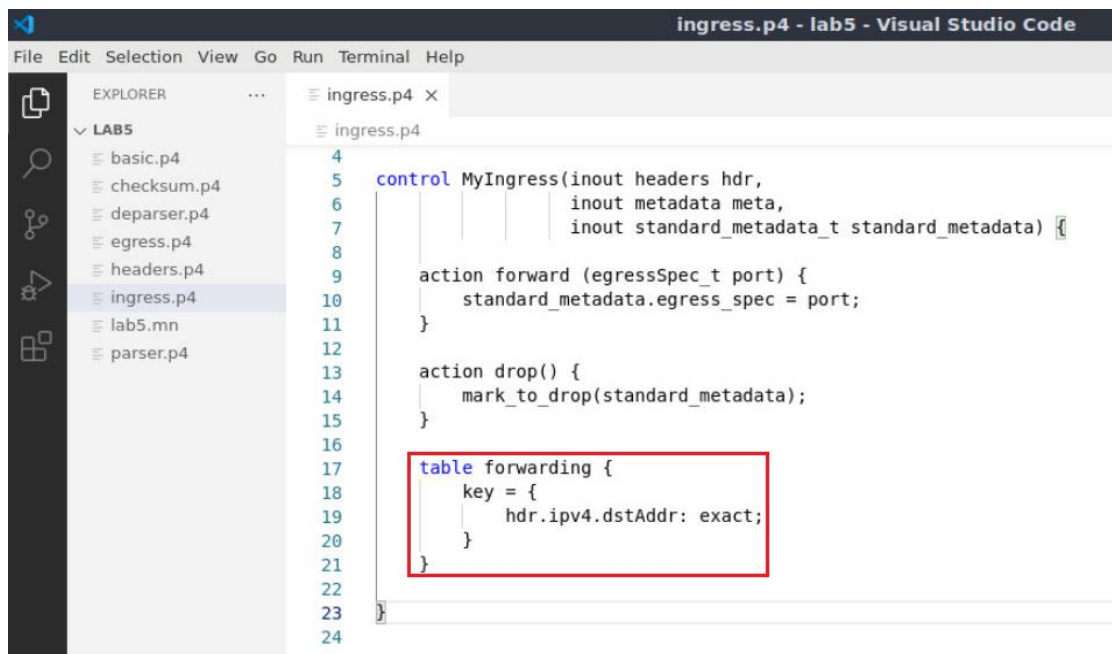


Figure 15. Specifying the key and the match type.

The inserted code specifies that the destination IPv4 address of a packet (`hdr.ipv4.dstAddr`) will be used as a key in the table. Also, the match type is `exact`,

denoting that the value of the destination IP address will be matched as is against a value specified later in the control plane.

**Step 7.** Add the following code inside the forwarding table to list the possible actions that will be used in this table.

```
actions = {
    forward;
    drop;
}
```

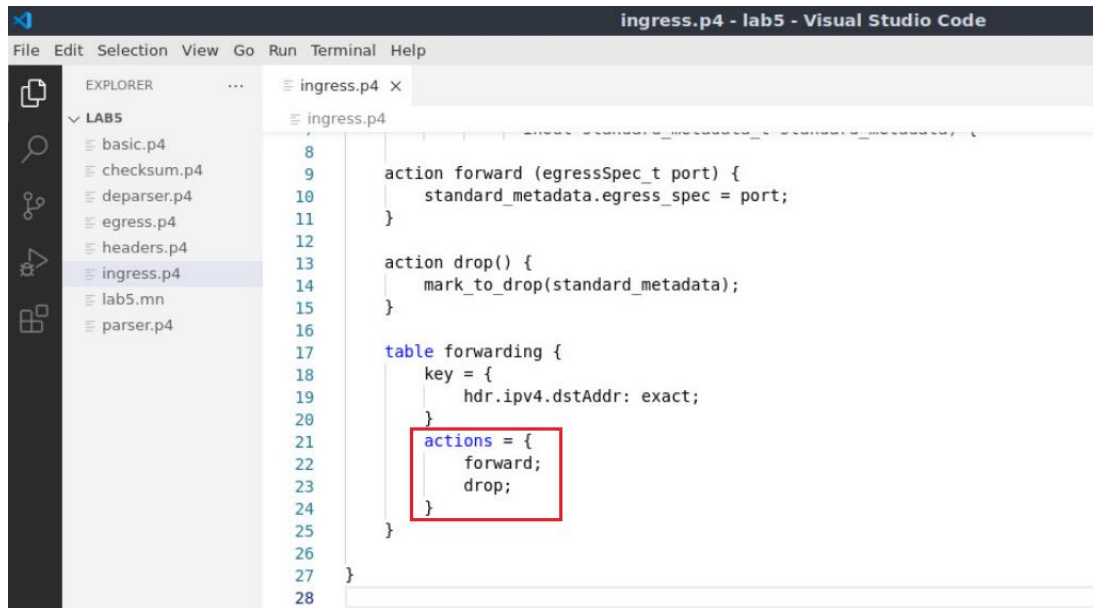
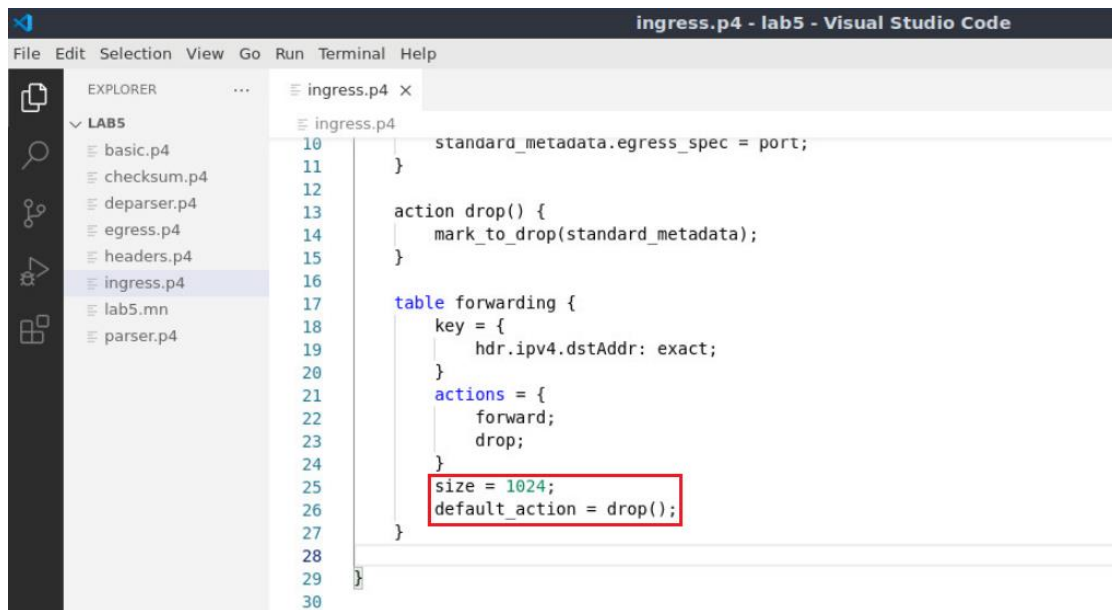


Figure 16. Adding the actions to the `forwarding` table.

The code above defines the possible actions.

**Step 8.** Add the following code inside the forwarding table. The `size` keyword specifies the maximum number of entries that can be inserted into this table from the control plane. The `default_action` keyword specifies which default action to be invoked whenever there is a miss.

```
size = 1024;
default_action = drop();
```

Figure 17. Specifying the size and default action of the `forwarding` table.

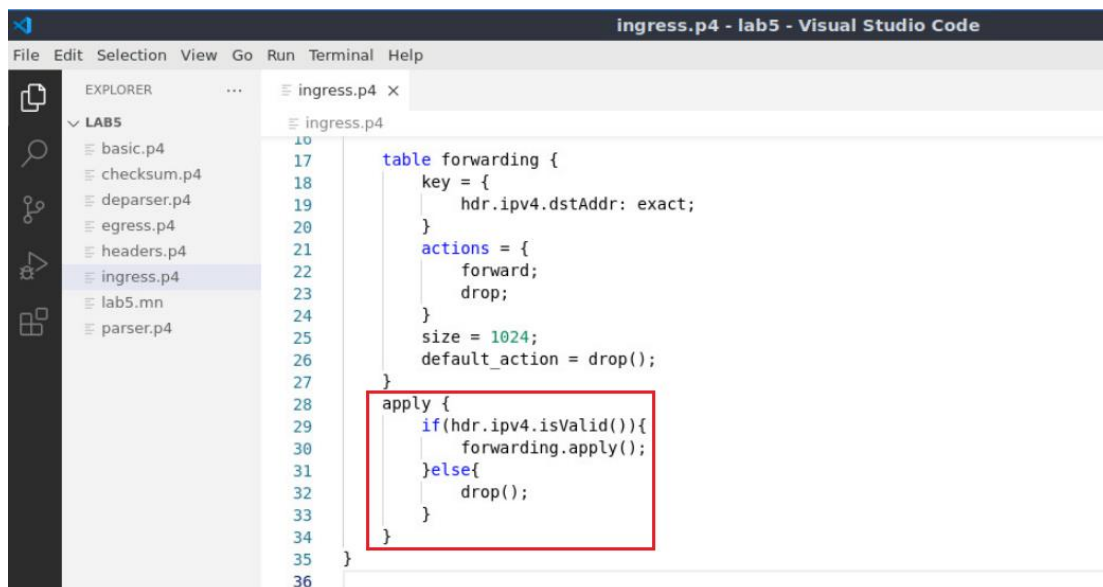
The code above denotes that a maximum of 1024 rules can be inserted into the table, and the default action to take whenever we have a miss is the `drop()` action.

**Step 9.** Add the following code inside the *MyIngress* block. The *apply* block defines the sequential flow of packet processing. It is required in every control block, otherwise the program will not compile. It describes in order, the sequence of tables to be invoked, among other packet processing instructions.

```

apply {
    if(hdr.ipv4.isValid()) {
        forwarding.apply();
    }else{
        drop();
    }
}

```

Figure 18. Defining the `apply` block.

In the code above, we are calling the table forwarding (`forwarding.apply()`) only if the IPv4 header is valid (`if (hdr.ipv4.isValid())`), otherwise the packet is dropped. The validity of the header is set if the parser successfully parsed said header (see *parser.p4* for a recap on the parser details). Note that if we received an IPv6 packet, the if-statement that checks for the validity of the IPv4 header will evaluate to false, and the forwarding table won't be applied.

**Step 10.** Save the changes to the file by pressing `Ctrl + s`.

## 4 Loading the P4 program

### 4.1 Compiling and loading the P4 program to switch s1

**Step 1.** Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```

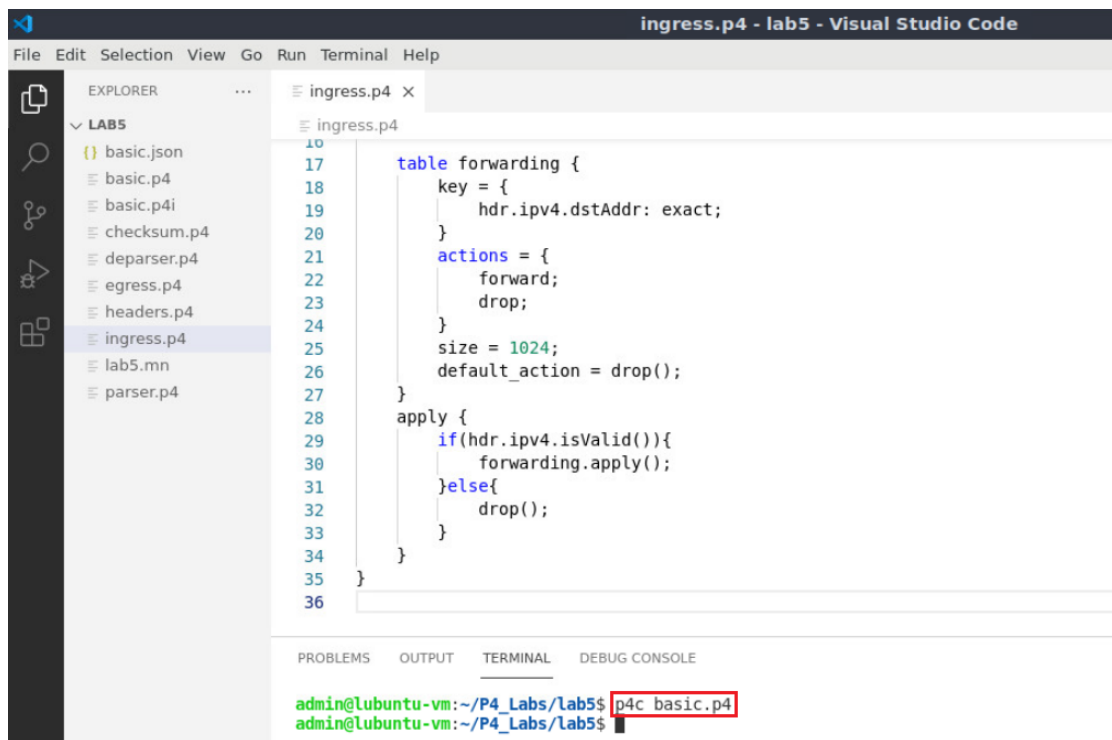


Figure 19. Compiling a P4 program.

**Step 2.** Type the command below in the terminal panel to push the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the p4c compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```

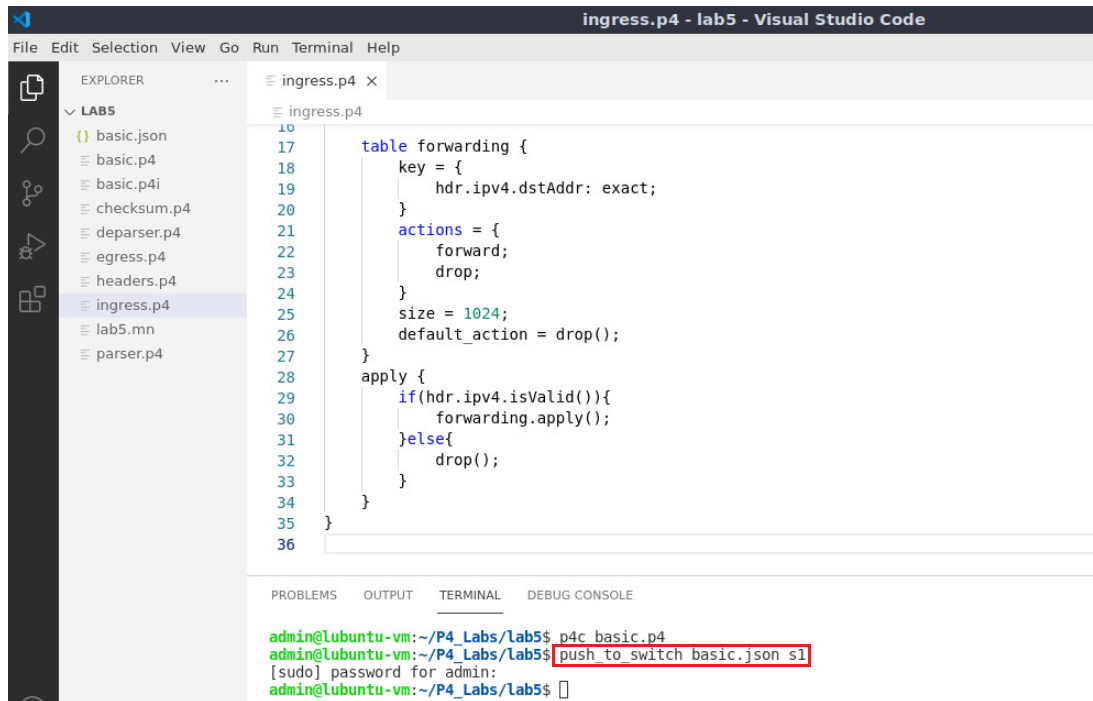


Figure 20. Pushing the *basic.json* file to switch s1.

## 4.2 Verifying the configuration

**Step 1.** Click on the MiniEdit tab in the start bar to maximize the window.



Figure 21. Maximizing the MiniEdit window.

**Step 2.** Right-click on the P4 switch icon in MiniEdit and start the *Terminal*.

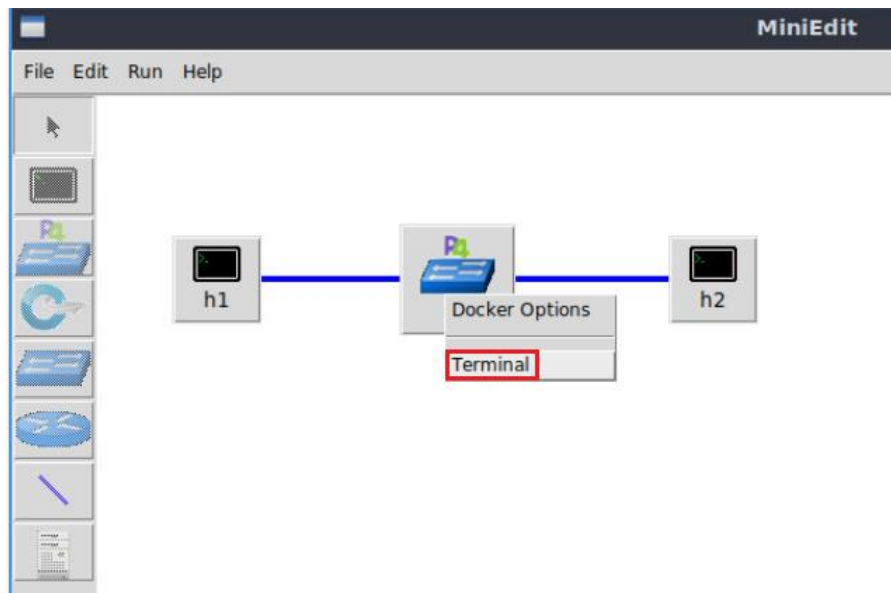


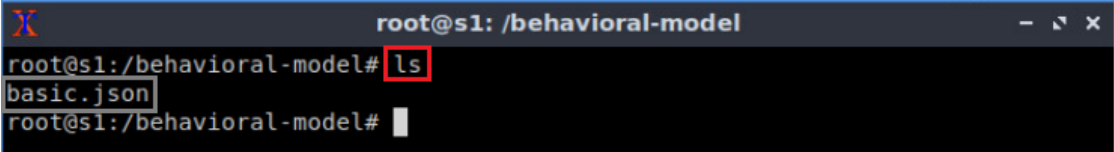
Figure 22. Starting the terminal on the switch.



Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

**Step 3.** Issue the command `ls` on the terminal of the switch s1 that was opened in the previous step.

```
ls
```



```
root@s1: /behavioral-model
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model#
```

Figure 23. Displaying the contents of the current directory in the switch s1.

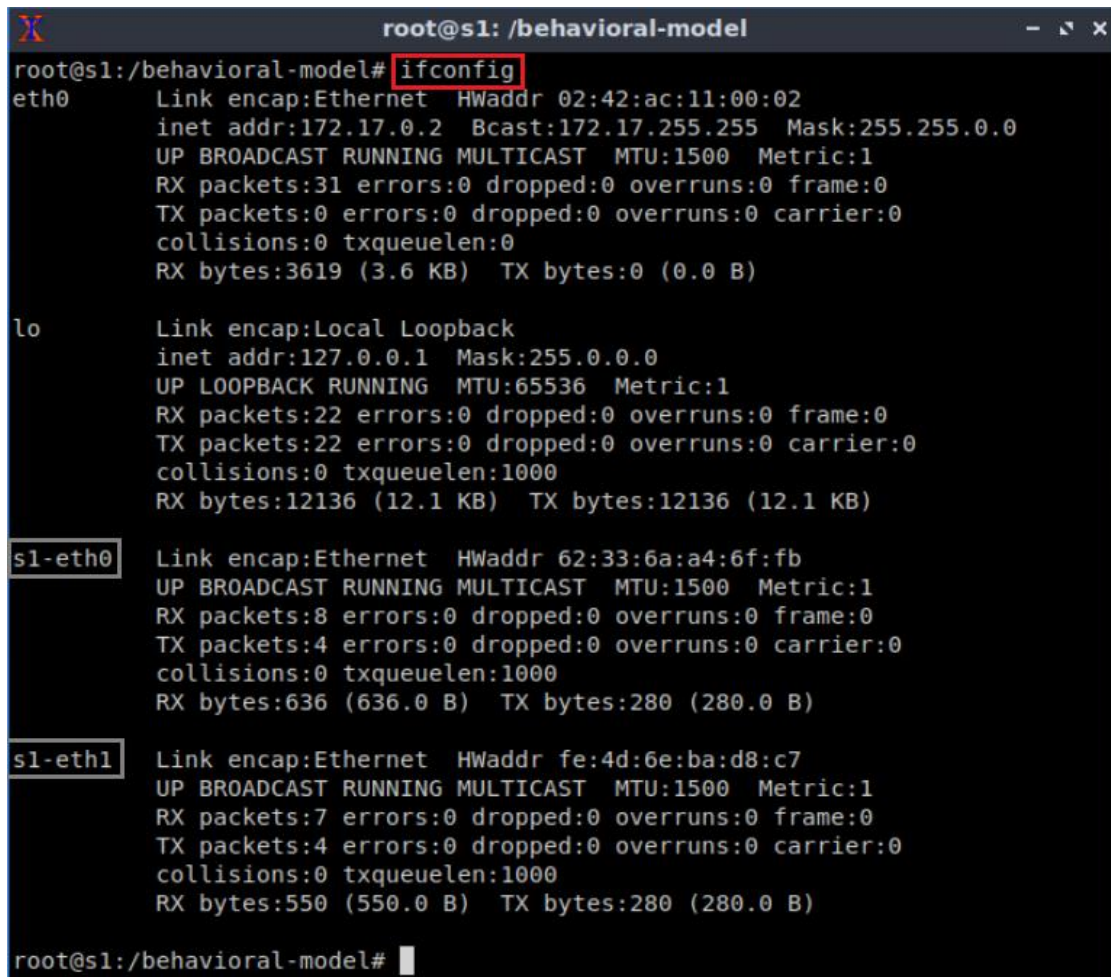
We can see that the switch contains the *basic.json* file that was pushed previously after compiling the P4 program.

## 5 Configuring switch s1

### 5.1 Mapping P4 program's ports

**Step 1.** Issue the command `ifconfig` on the terminal of the switch s1.

```
ifconfig
```



```

root@s1: /behavioral-model
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3619 (3.6 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12136 (12.1 KB)  TX bytes:12136 (12.1 KB)

s1-eth0   Link encap:Ethernet  HWaddr 62:33:6a:a4:6f:fb
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:636 (636.0 B)  TX bytes:280 (280.0 B)

s1-eth1   Link encap:Ethernet  HWaddr fe:4d:6e:ba:d8:c7
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:550 (550.0 B)  TX bytes:280 (280.0 B)

root@s1:/behavioral-model#

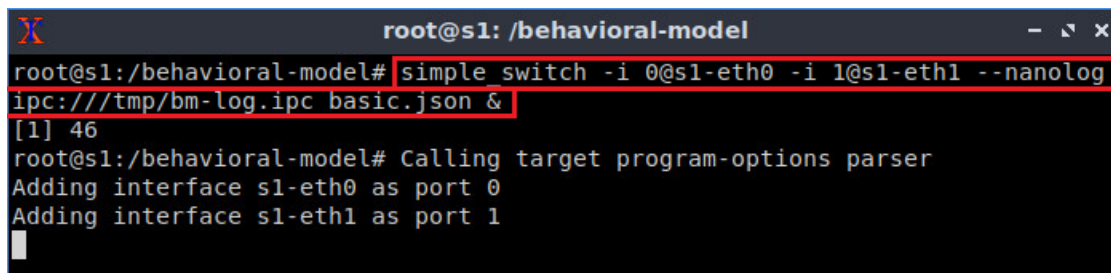
```

Figure 24. Displaying switch s1 interfaces.

We can see that the switch has the interfaces *s1-eth0* and *s1-eth1*. The interface *s1-eth0* on the switch s1 connects host h1. The interface *s1-eth1* on the switch s1 connects host h2.

**Step 2.** Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog ipc:///tmp/bm-log.ipc
basic.json &
```



```

root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 46
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

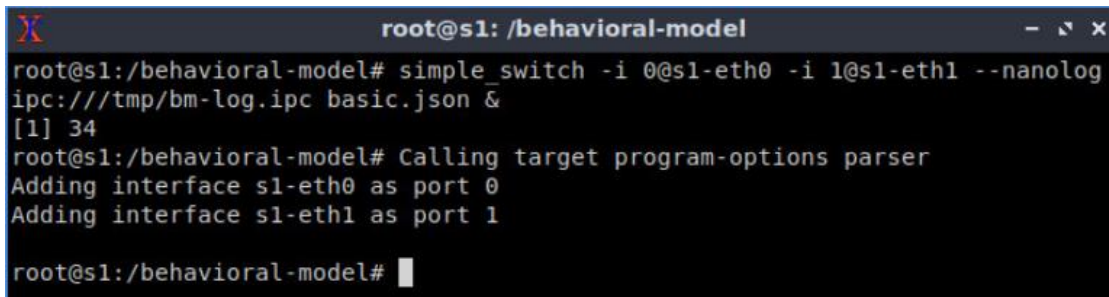
```

Figure 25. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

The `--nanolog` option is used to instruct the switch daemon that we want to see the logs of the switch.

## 5.2 Loading the rules to the switch

**Step 1.** In switch s1 terminal, press *Enter* to return the CLI.



```

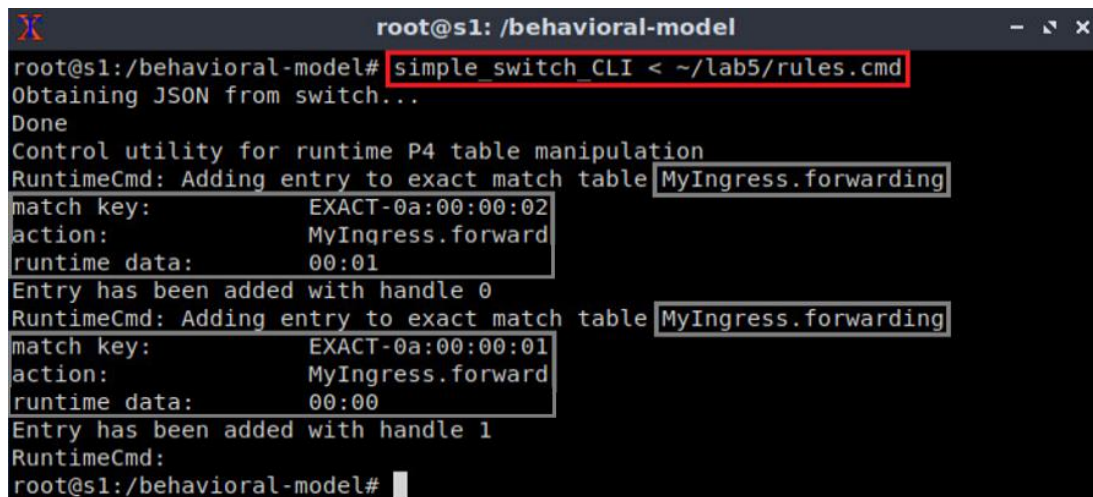
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 34
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
root@s1:/behavioral-model#

```

Figure 26. Returning to switch s1 CLI.

**Step 2.** Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab5/rules.cmd
```



```

root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab5/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:      EXACT-0a:00:00:02
action:         MyIngress.forward
runtime data:   00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:      EXACT-0a:00:00:01
action:         MyIngress.forward
runtime data:   00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#

```

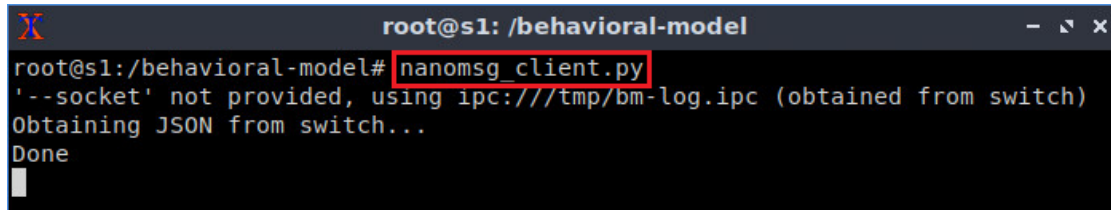
Figure 27. Populating the forwarding table into switch s1.

The script above pushes the rules to the switch daemon. We can see that we added two entries to the `forwarding` table. The key of the first entry is 10.0.0.2 (which translates to 0a:00:00:02 in hexadecimal as shown in the figure above, next to match key), its action is forward, and its action data is `00:01`, which specifies port 1. Similarly, the key of the second entry is 10.0.0.1 (which translates to 0a:00:00:01 in hexadecimal as shown in the figure above, next to match key), its action is forward, and its action data is `00:00`, which specifies port 0.

## 6 Testing and verifying the P4 program

**Step 1.** Type the following command to display the switch logs.

```
nanomsg_client.py
```



```

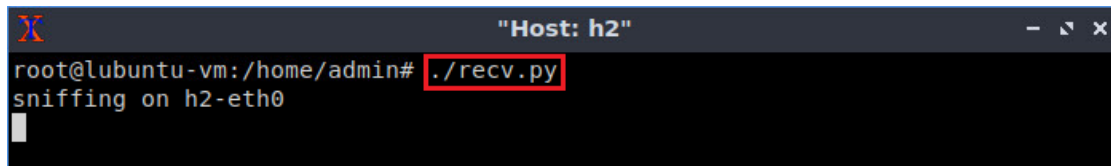
root@s1: /behavioral-model
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done

```

Figure 28. Displaying switch s1 logs.

**Step 2.** On host h2's terminal, type the command the command below so that, the host starts listening for packets.

```
./recv.py
```



```

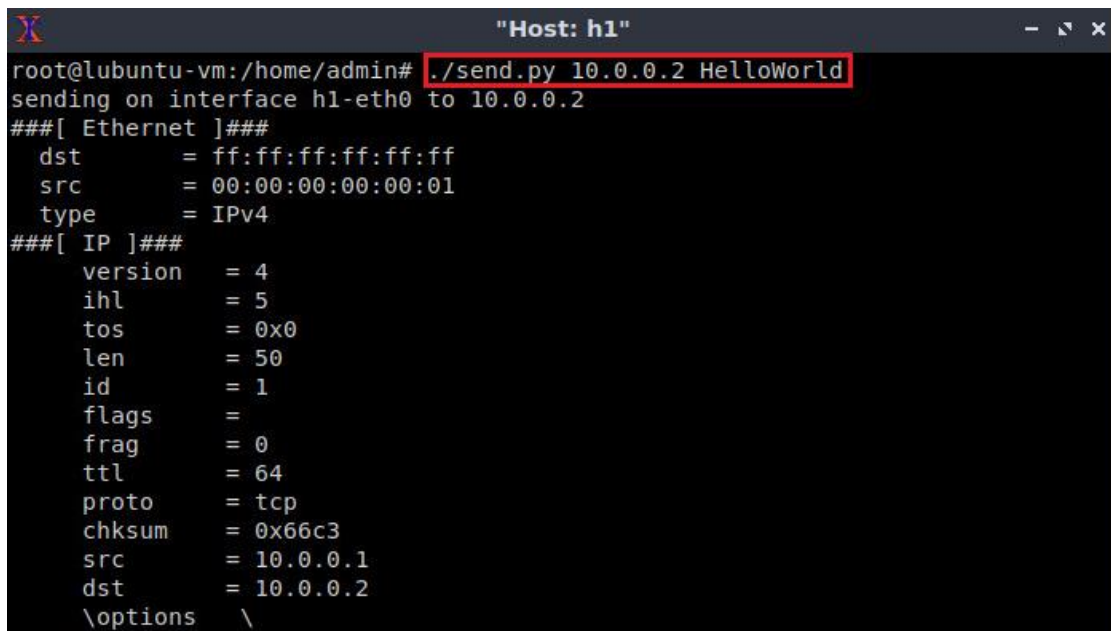
"Host: h2"
root@lubuntu-vm:/home/admin# ./recv.py
sniffing on h2-eth0

```

Figure 29. Listening for incoming packets in host h2.

**Step 3.** On host h1's terminal, type the following command.

```
./send.py 10.0.0.2 HelloWorld
```



```

"Host: h1"
root@lubuntu-vm:/home/admin# ./send.py 10.0.0.2 HelloWorld
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 00:00:00:00:00:01
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 50
  id       = 1
  flags    =
  frag     = 0
  ttl      = 64
  proto    = tcp
  chksum   = 0x66c3
  src      = 10.0.0.1
  dst      = 10.0.0.2
  \options \

```

Figure 30. Sending a test packet from host h1 to host h2.

**Step 4.** Inspect the logs on switch s1 terminal.

```

root@s1: /behavioral-model
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET IN, port in: 0
type: PARSE_START, parser_id: 0 (parser)
type: PARSE_EXTRACT, header_id: 2 (ethernet)
type: PARSE_EXTRACT, header_id: 3 (ipv4)
type: PARSE_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, condition_id: 0 (node 2), result: True
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 0 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port out: 1

```

Figure 31. Inspecting the logs in switch s1.

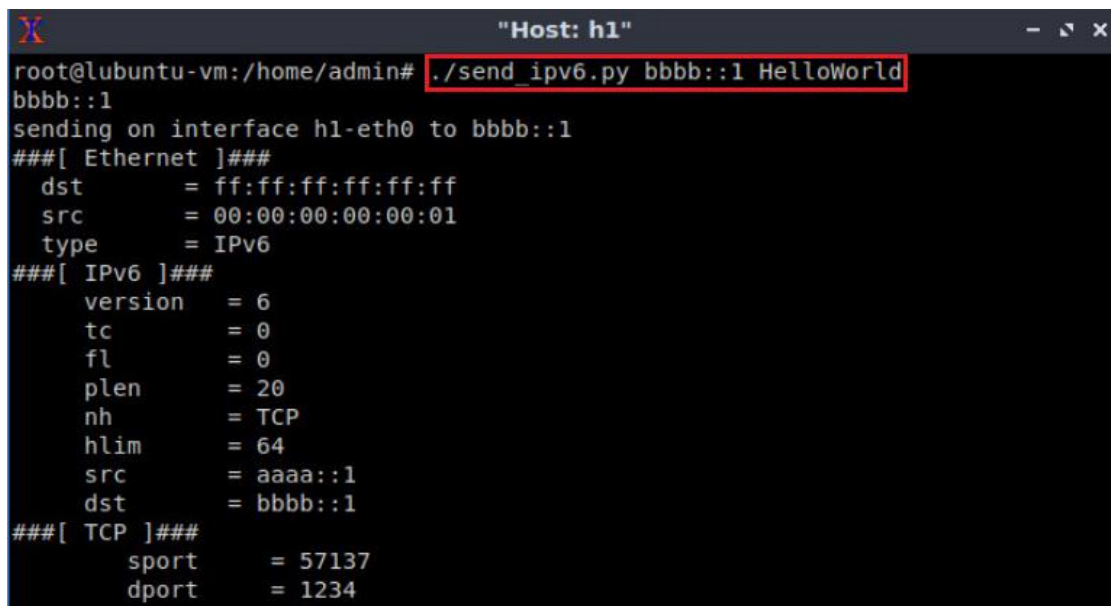
Note how the parser parsed the IPv4 header since the packet is IPv4. Also, we can see that the condition evaluated to True (the condition here refers to `if (hdr.ipv4.isValid())` in the P4 program). Consequently, the table forwarding was applied, and because we have a hit on the destination IP address (i.e., 10.0.0.2, inserted through the script), the packet was forwarded to host h2.

**Step 5.** Verify that the packet was received on host h2.

**Step 6.** On host h1's terminal, type the following command to send an IPv6 packet to host h2.

```
./send_ipv6.py bbbb::1 HelloWorld
```





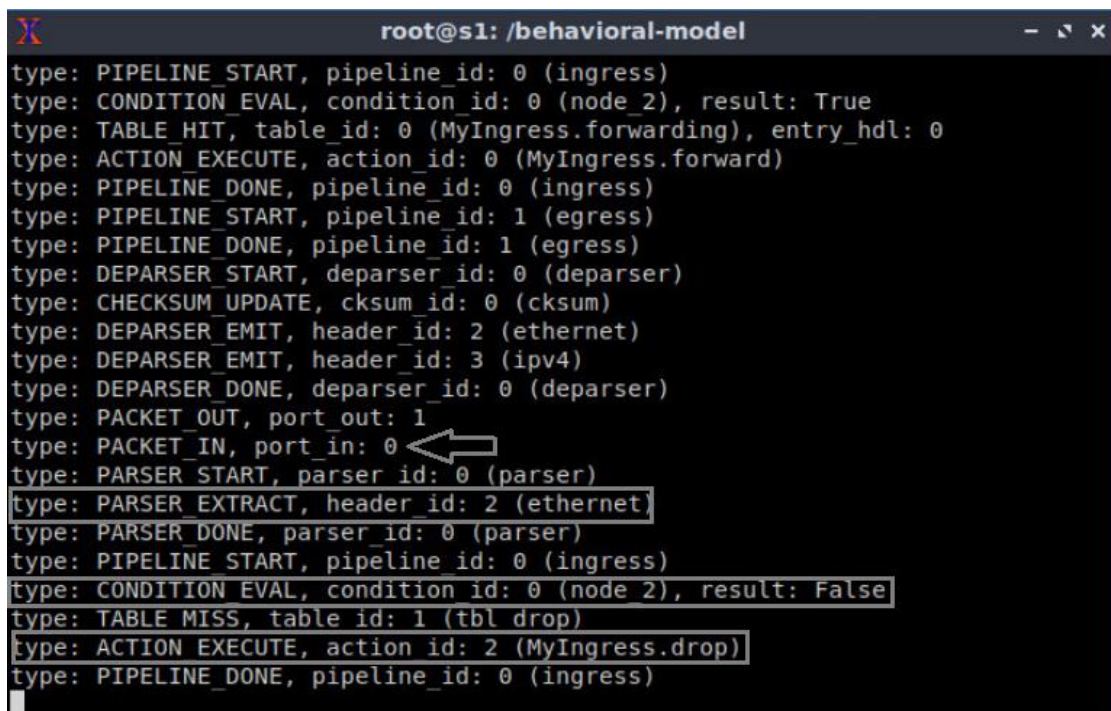
```

Host: h1
root@lubuntu-vm:/home/admin# ./send_ipv6.py bbbb::1 HelloWorld
bbbb::1
sending on interface h1-eth0 to bbbb::1
###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 00:00:00:00:00:01
  type     = IPv6
###[ IPv6 ]###
  version  = 6
  tc       = 0
  fl       = 0
  plen     = 20
  nh       = TCP
  hlim     = 64
  src      = aaaa::1
  dst      = bbbb::1
###[ TCP ]###
  sport    = 57137
  dport    = 1234

```

Figure 32. Sending an IPv6 test packet from host h1 to host h2.

**Step 7.** Inspect the logs on switch s1 terminal. The arrow indicates where the logs of the new packet starts.



```

root@s1: /behavioral-model
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, condition_id: 0 (node 2), result: True
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 0 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
type: PACKET_IN, port_in: 0 ←
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, condition_id: 0 (node 2), result: False
type: TABLE_MISS, table_id: 1 (tbl drop)
type: ACTION_EXECUTE, action_id: 2 (MyIngress.drop)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)

```

Figure 33. Inspecting the logs in switch s1.

Note how the parser now did not parse IPv4 since the packet is IPv6. Also, we can see that the condition evaluated to False (the condition here refers to `if (hdr.ipv4.isValid())` in the P4 program) and the packet is dropped. Consequently, the table was not applied, and the packet was not forwarded to host h2.

This concludes lab 5. Stop the emulation and then exit out of MiniEdit.

## References

1. “p4c core.p4”. [Online]. Available: <https://github.com/p4lang/p4c/blob/main/p4include/core.p4>.
2. P4 Language Tutorial. [Online]. Available: <https://tinyurl.com/2p9cen9e>.