

Overview

This lab describes the building blocks and the general structure of a P4 program. It maps the program's components to the Protocol-Independent Switching Architecture (PISA), a programmable pipeline used by modern whitebox switching hardware. The lab also demonstrates how to track an incoming packet as it traverses the pipeline of the switch. Such capability is very useful to debug and troubleshoot a P4 program.

Objectives

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

1. Understand the PISA architecture.
2. Understand on high-level the main building blocks of a P4 program.
3. Map the P4 program components to the components of the programmable pipeline.
4. Trace the lifecycle of a packet as it traverses the pipeline.

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: The PISA architecture.
2. Section 2: Lab topology.
3. Section 3: Navigating through the components of a basic P4 program.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Testing and verifying the P4 program.

1 The PISA architecture

1.1 The PISA architecture

The Protocol Independent Switch Architecture (PISA)¹ is a packet processing model that includes the following elements: programmable parser, programmable match-action pipeline, and programmable deparser, see Figure 1. The programmable parser permits the programmer to define the headers (according to custom or standard protocols) and to parse them. The parser can be represented as a state machine. The programmable match-action pipeline executes the operations over the packet headers and intermediate results. A single match-action stage has multiple memory blocks (e.g., tables, registers) and Arithmetic Logic Units (ALUs), which allow for simultaneous lookups and actions. Since some action results may be needed for further processing (e.g., data dependencies), stages are arranged sequentially. The programmable deparser assembles the packet headers back and serializes them for transmission. A PISA device is protocol independent. The P4 program defines the format of the keys used for lookup operations. Keys can be formed using packet header's information. The control plane populates table entries with keys and action data. Keys are used for matching packet information (e.g., destination IP address) and action data is used for operations (e.g., output port).

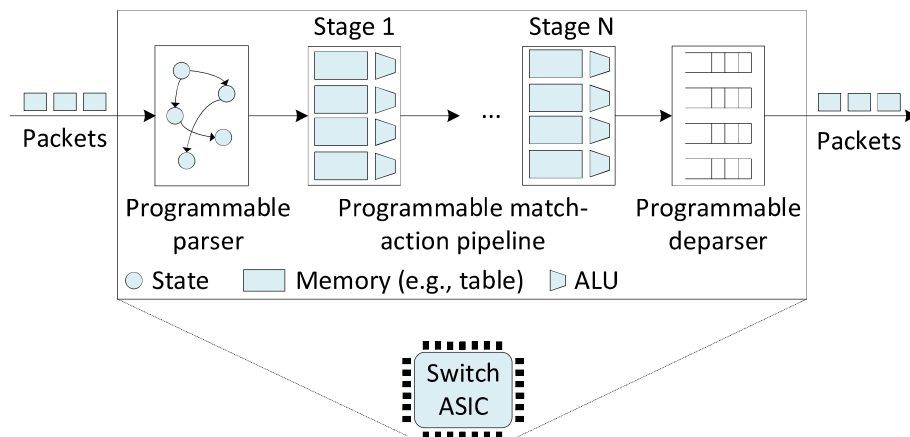


Figure 1. A PISA-based data plane.

Programmable switches do not introduce performance penalty. On the contrary, they may produce better performance than fixed-function switches. When compared with general purpose CPUs, ASICs remain faster at switching, and the gap is only increasing.

1.2 Programmable parser

The programmable parser permits the programmer to define the headers (according to custom or standard protocols) and to describe how the switch should process those headers. The parser de-encapsulates the headers, converting the original packet into a parsed representation of the packet. The programmer declares the headers that must be recognized and their order in the packet. The parser can be represented as a state machine without cycles (direct acyclic graph), with one initial state (start) and two final states (accept or reject).

1.3 Programmable match-action pipeline

The match-action pipeline implements the processing occurring at a switch. The pipeline consists of multiple identical stages (N stages are shown in Figure 1). Practical implementations may have 10/15 stages on the ingress and egress pipelines. Each stage contains multiple match-action units (4 units per stage in Figure 1). A match-action unit has a match phase and an action phase. During the match phase, a table is used to match a header field of the incoming packet against entries in the table (e.g., destination IP address). Note that there are multiple tables in a stage (4 tables per stage in Figure 1), which permit the switch to perform multiple matches in parallel over different header fields. Once a match occurs, a corresponding action is performed by the ALU. Examples of actions include: modify a header field, forward the packet to an egress port, drop the packet, and others. The sequential arrangement of stages allows for the implementation of serial dependencies. For example, if the result of an operation is needed prior to perform a second operation, then the compiler would place the first operation at an earlier stage than the second operation.

1.4 Programmable deparser

The deparser assembles back the packet and serializes it for transmission. The programmer specifies the headers to be emitted by the deparser. When assembling the packet, the deparser emits the specified headers followed by the original payload of the packet.

1.5 The V1Model

Figure 2 depicts the V1Model² architecture components. The V1Model architecture consists of a programmable parser, an ingress match-action pipeline, a traffic manager, an egress match-action pipeline, and a programmable deparser. The traffic manager schedules packets between input ports and output ports and performs packet replication (e.g., replication of a packet for multicasting). The V1Model architecture is implemented on top BMv2's simple_switch target³.

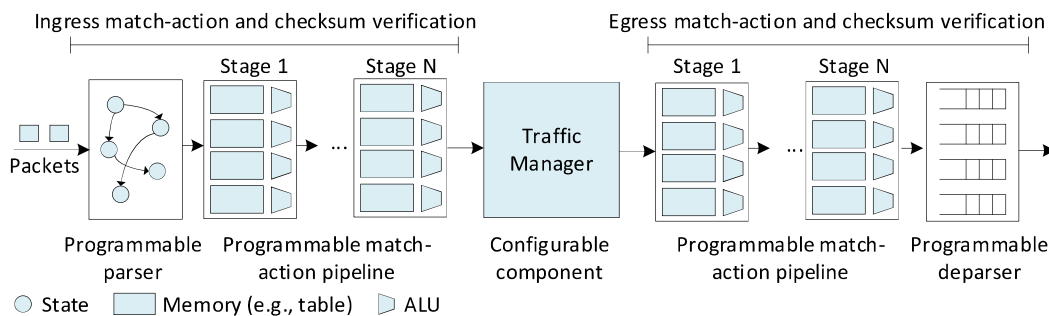


Figure 2. The V1Model architecture.

1.6 P4 program mapping to the V1Model

The P4 program used in this lab is separated into different files. Figure 3 shows the V1Model and its associated P4 files. These files are as follows:

- *headers.p4*: this file contains the packet headers' and the metadata's definitions.
- *parser.p4*: this file contains the implementation of the programmable parser.
- *ingress.p4*: this file contains the ingress control block that includes match-action tables.
- *egress.p4*: this file contains the egress control block.
- *deparser.p4*: this file contains the deparser logic that describes how headers are emitted from the switch.
- *checksum.p4*: this file contains the code that verifies and computes checksums.
- *basic.p4*: this file contains the starting point of the program (main) and invokes the other files. This file must be compiled.

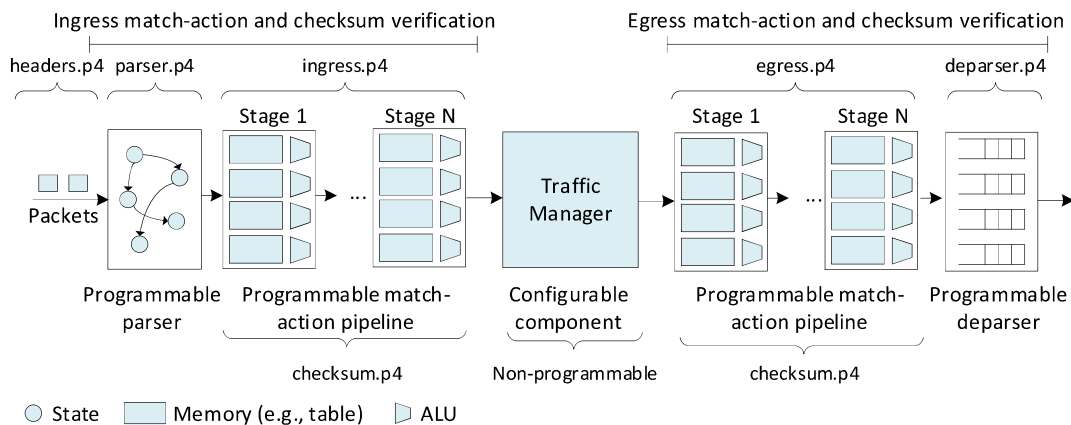


Figure 3. Mapping of P4 files to the V1Model's components.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

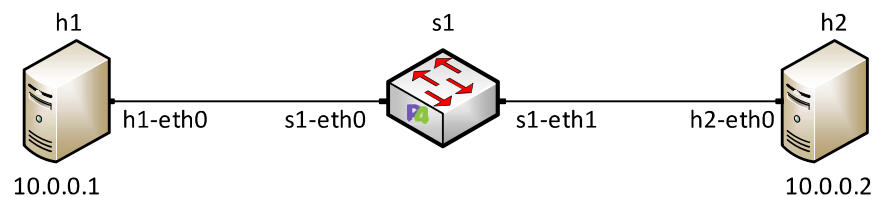


Figure 4. Lab topology.

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 5. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the lab3 folder and search for the topology file called *lab3.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

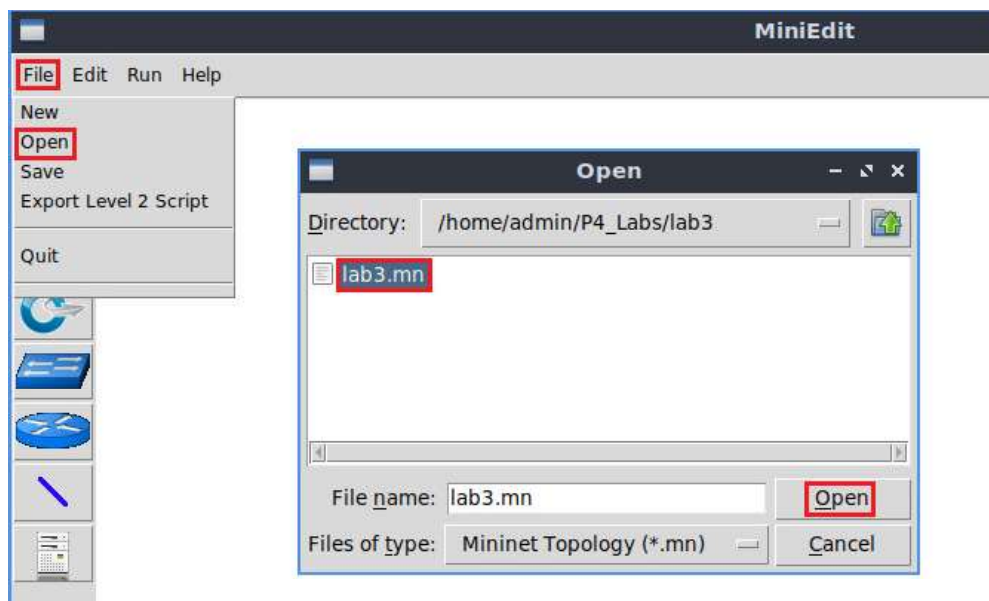


Figure 6. Opening a topology in MiniEdit.

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 7. 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.

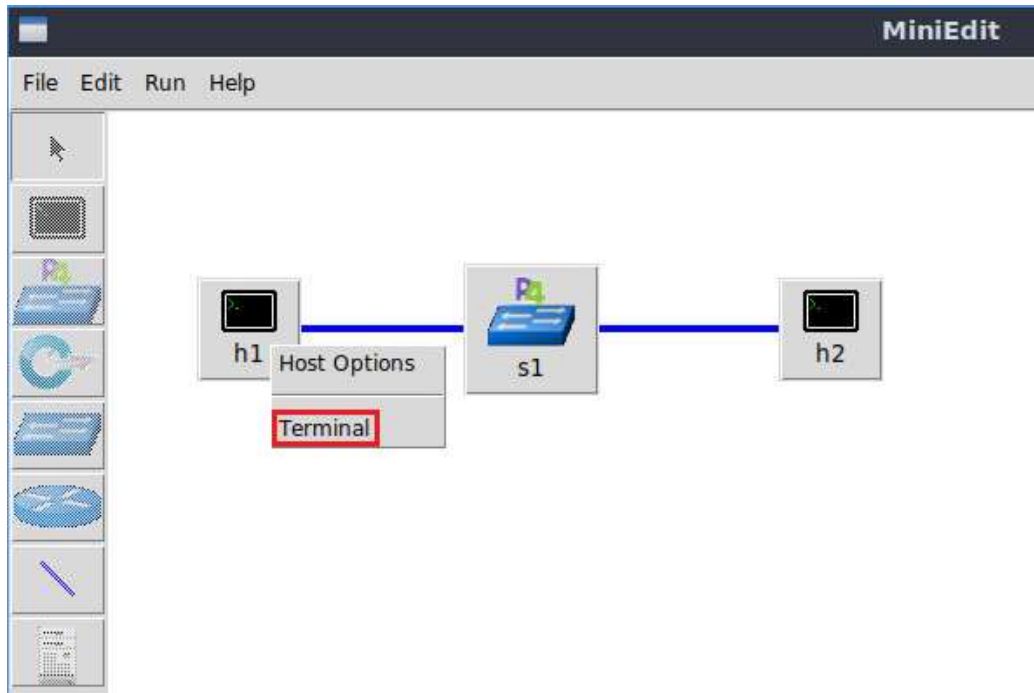


Figure 8. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

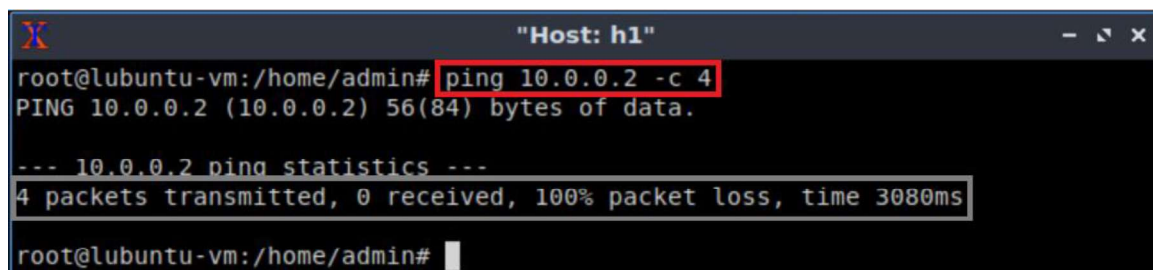


Figure 9. Performing a connectivity test between host h1 and host h2.

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

3 Navigating through the components of a basic P4 program

This section shows the steps required to compile the P4 program. It illustrates the editor that will be used to modify the P4 program, and the P4 compiler that will produce a data plane program for the software switch.

3.1 Loading the programming environment

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

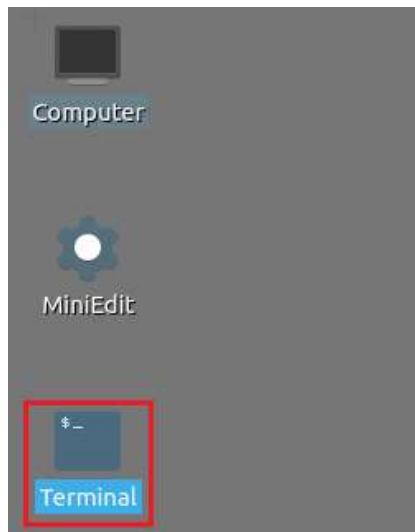


Figure 10. Shortcut to open a Linux terminal.

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.

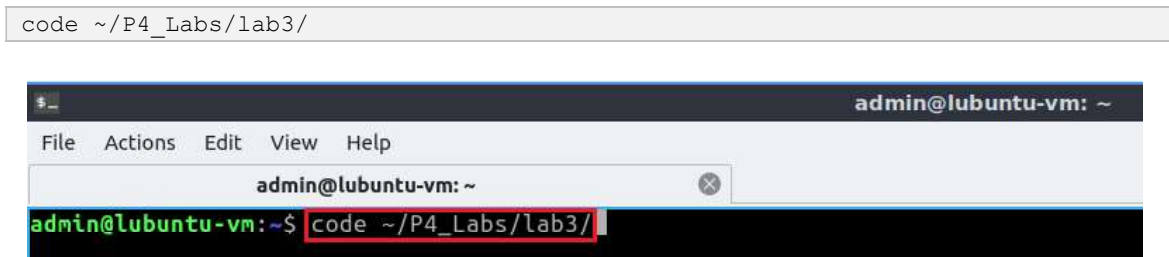


Figure 11. Launching the editor and opening the lab3 directory.

3.2 Describing the components of the P4 program

Step 1. Once the previous command is executed, VS Code will start. Click on *basic.p4* in the file explorer panel on the left hand side to open the P4 program in the editor.

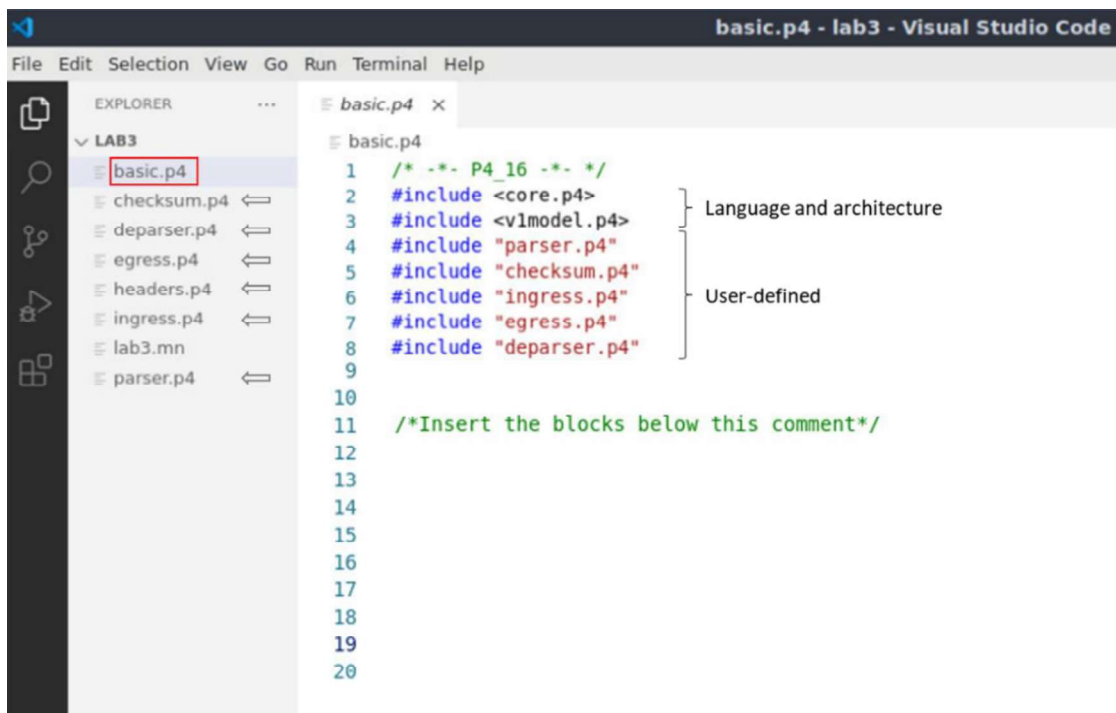


Figure 12. The main P4 file and how it includes other user-defined files.

The *basic.p4* file includes the starting point of the P4 program and other files that are specific to the language (*core.p4*) and to the architecture (*v1model.p4*). To make the P4 program easier to read and understand, we separated the whole program into different files. Note how the files in the explorer panel correspond to the components of the V1Model. To use those files, the main file (*basic.p4*) must include them first. For example, to use the parser, we need to include the *parser.p4* file (`#include "parser.p4"`).

We will navigate through the files in sequence as they appear in the architecture.

Step 2. Click on the *headers.p4* file to display the content of the file.


```

1  const bit<16> TYPE_IPV4 = 0x800;
2
3  /*****
4  ***** HEADERS *****/
5  */
6
7  typedef bit<9> egressSpec_t;
8  typedef bit<48> macAddr_t;
9  typedef bit<32> ip4Addr_t;
10
11  header ethernet_t {
12      macAddr_t dstAddr;
13      macAddr_t srcAddr;
14      bit<16> etherType;
15  }
16
17  header ipv4_t {
18      bit<4> version;
19      bit<4> ihl;
20      bit<8> diffserv;
21      bit<16> totalLen;
22      bit<16> identification;
23      bit<3> flags;
24      bit<13> fragOffset;
25      bit<8> ttl;
26      bit<8> protocol;
27      bit<16> hdrChecksum;
28      ip4Addr_t srcAddr;
29      ip4Addr_t dstAddr;
30  }
31
32  struct metadata {
33      /* empty */
34  }
35
36  struct headers {
37      ethernet_t ethernet;
38      ipv4_t ipv4;
39  }

```

Figure 13. The defined headers.

The *headers.p4* above shows the headers that will be used in our pipeline. We can see that the ethernet and the IPv4 headers are defined. We can also see how they are grouped into a structure (`struct headers`). The `headers` name will be used throughout the program when referring to the headers. Furthermore, the file shows how we can use `typedef` to provide an alternative name to a type.

Step 3. Click on the *parser.p4* file to display the content of the parser.

```

1  #include "headers.p4"
2
3  /*****
4  ***** P A R S E R *****
5  *****/
6
7  parser MyParser(packet in packet,
8                out headers hdr,
9                inout metadata meta,
10               inout standard_metadata_t standard_metadata) {
11
12     state start {
13         transition parse_ethernet;
14     }
15
16     state parse_ethernet {
17         packet.extract(hdr.ethernet);
18         transition select(hdr.ethernet.etherType) {
19             TYPE_IPV4: parse_ipv4;
20             default: accept;
21         }
22     }
23 }

```

Figure 14. The parser implementation.

The figure above shows the content of the `parser.p4` file. We can see that the parser is already written with the name `MyParser`. This name will be used when defining the pipeline sequence.

Step 4. Click on the `ingress.p4` file to display the content of the file.

```

3  /*****
4  ***** INGRESS PROCESSING *****
5  *****/
6
7  control MyIngress(inout headers hdr,
8                  inout metadata meta,
9                  inout standard_metadata_t standard_metadata) {
10
11     action drop() {
12         mark_to_drop(standard_metadata);
13     }
14
15     action forward(egressSpec_t port) {
16         standard_metadata.egress_spec = port;
17     }
18
19     table forwarding {
20         key = {
21             standard_metadata.ingress_port:exact;
22         }
23         actions = {
24             forward;
25             drop;
26             NoAction;
27         }
28         size = 1024;
29         default_action = drop();
30     }
31
32     apply {
33         forwarding.apply();
34     }
35 }

```

Figure 15. The ingress component.

The figure above shows the content of the *ingress.p4* file. We can see that the ingress is already written with the name *MyIngress*. This name will be used when defining the pipeline sequence.

Step 5. Click on the *egress.p4* file to display the content of the file.



```

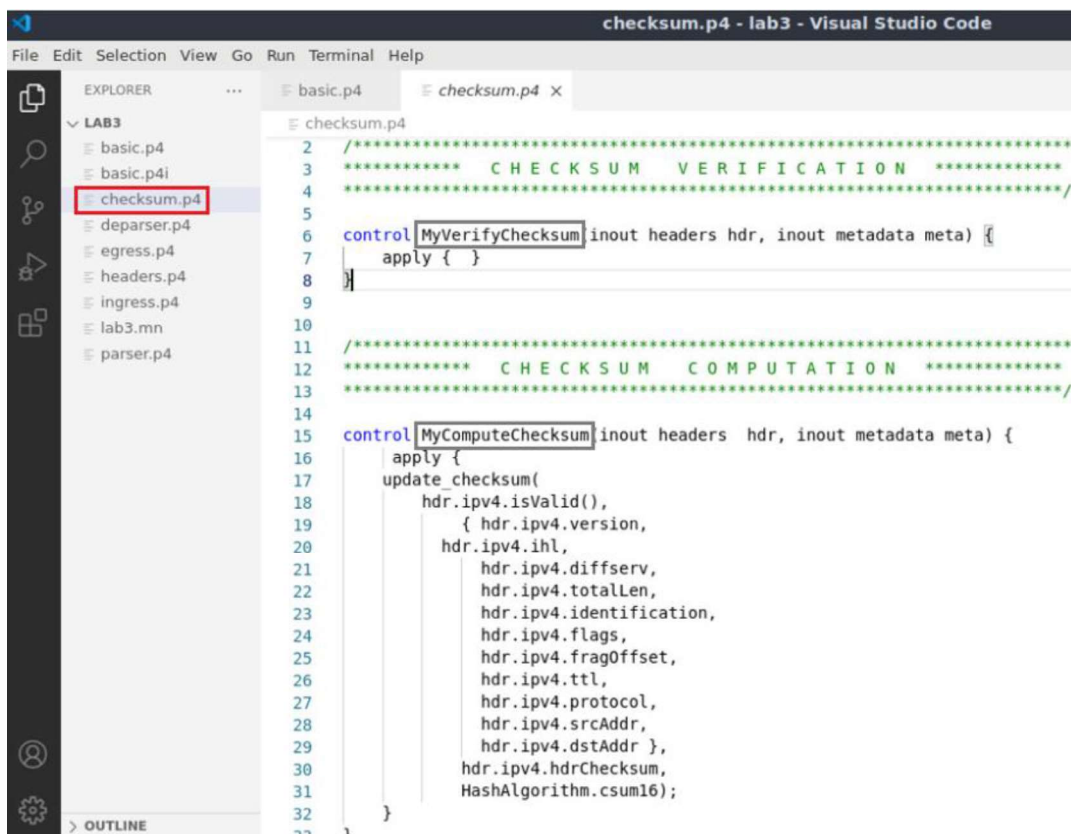
1
2 /*****
3  ***** EGRESS PROCESSING *****
4  *****/
5
6 control MyEgress {
7     inout headers hdr,
8     inout metadata meta,
9     inout standard_metadata_t standard_metadata) {
10     apply { }
11 }

```

Figure 16. The egress component.

The figure above shows the content of the *egress.p4* file. We can see that the egress is already written with the name *MyEgress*. This name will be used when defining the pipeline sequence.

Step 6. Click on the *checksum.p4* file to display the content of the file.



```

2 /*****
3  ***** CHECKSUM VERIFICATION *****
4  *****/
5
6 control MyVerifyChecksum {
7     inout headers hdr,
8     inout metadata meta) {
9     apply { }
10 }
11
12 /*****
13  ***** CHECKSUM COMPUTATION *****
14  *****/
15 control MyComputeChecksum {
16     inout headers hdr,
17     inout metadata meta) {
18     apply {
19         update_checksum(
20             hdr.ipv4.isValid(),
21             { hdr.ipv4.version,
22               hdr.ipv4.ihl,
23               hdr.ipv4.diffserv,
24               hdr.ipv4.totalLen,
25               hdr.ipv4.identification,
26               hdr.ipv4.flags,
27               hdr.ipv4.fragOffset,
28               hdr.ipv4.ttl,
29               hdr.ipv4.protocol,
30               hdr.ipv4.srcAddr,
31               hdr.ipv4.dstAddr },
32             hdr.ipv4.hdrChecksum,
33             HashAlgorithm.csum16);
34     }
35 }

```

Figure 17. The checksum component.

The figure above shows the content of the *checksum.p4* file. We can see that the checksum is already written with two control blocks: `MyVerifyChecksum` and `MyComputeChecksum`. These names will be used when defining the pipeline sequence. Note that `MyVerifyChecksum` is empty since no checksum verification is performed in this lab.

Step 7. Click on the *deparser.p4* file to display the content of the file.



Figure 18. The deparser component.

The figure above shows the content of the *deparser.p4* file. We can see that the deparser is already written with two instructions that reassemble the packet.

3.3 Programming the pipeline sequence

Now it is time to write the pipeline sequence in the *basic.p4* program.

Step 1. Click on the *basic.p4* file to display the content of the file.



Figure 19. Selecting the *basic.p4* file.

Step 2. Write the following block of code at the end of the file

```

V1Switch (
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

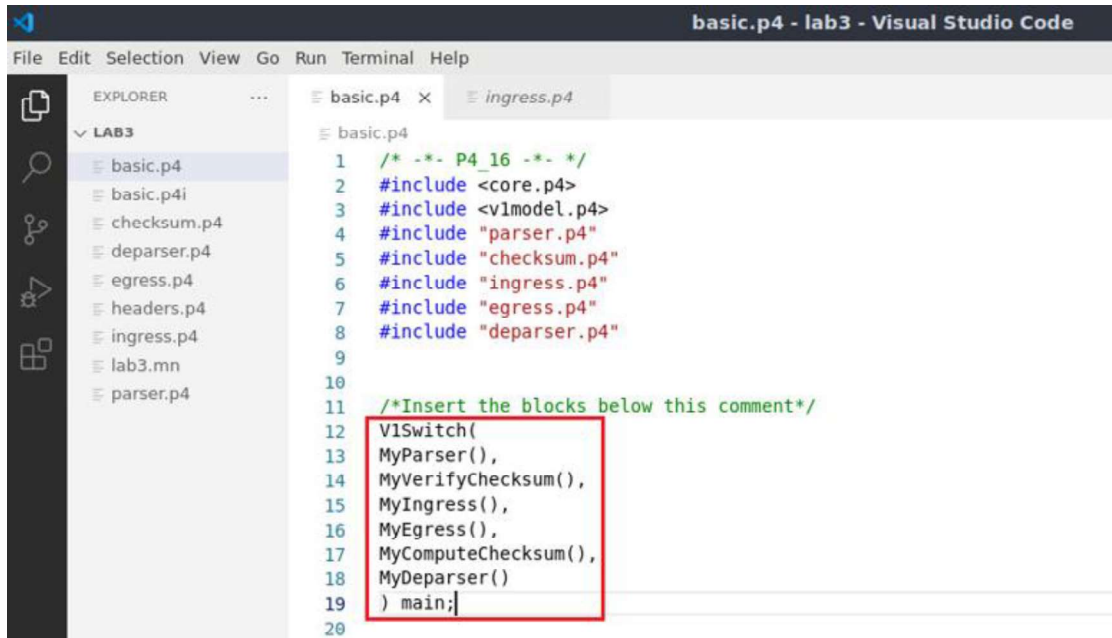


Figure 20. Writing the pipeline sequence in the `basic.p4` program

We can see here that we are defining the pipeline sequence according to the V1Model architecture. First, we start by the parser, then we verify the checksum. Afterwards, we specify the ingress block and the egress block, and we recompute the checksum. Finally, we specify the deparser.

Step 3. Save the changes 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
```

```
basic.p4 - lab3 - Visual Studio Code
File Edit Selection View Go Run Terminal Help

EXPLORER
LAB3
  basic.json
  basic.p4
  basic.p4i
  checksum.p4
  deparser.p4
  egress.p4
  headers.p4
  ingress.p4
  lab3.mn
  parser.p4

basic.p4
1  /* -*- P4_16 -*- */
2  #include <core.p4>
3  #include <vlmodel.p4>
4  #include "parser.p4"
5  #include "checksum.p4"
6  #include "ingress.p4"
7  #include "egress.p4"
8  #include "deparser.p4"
9
10
11 /*Insert the blocks below this comment*/
12 V1Switch(
13   MyParser(),
14   MyVerifyChecksum(),
15   MyIngress(),
16   MyEgress(),
17   MyComputeChecksum(),
18   MyDeparser()
19 ) main;
20

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
admin@lubuntu-vm:~/P4_Labs/lab3$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab3$
```

Figure 21. Compiling a P4 program.

Step 2. Type the command below in the terminal panel to download 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
```

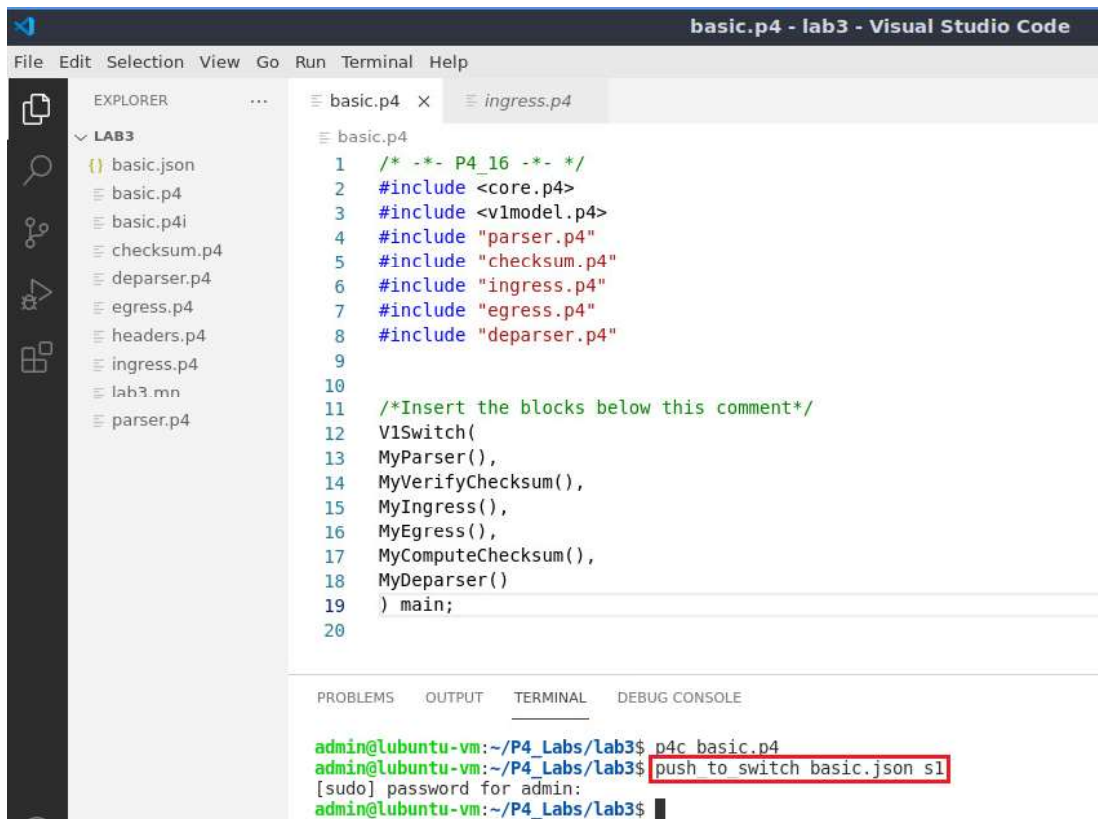



Figure 22. Downloading the P4 program to switch s1.

4.2 Verifying the configuration

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



Figure 23. Maximizing the MiniEdit window.

Step 2. In MiniEdit, right-click on the P4 switch icon and start the *Terminal*.

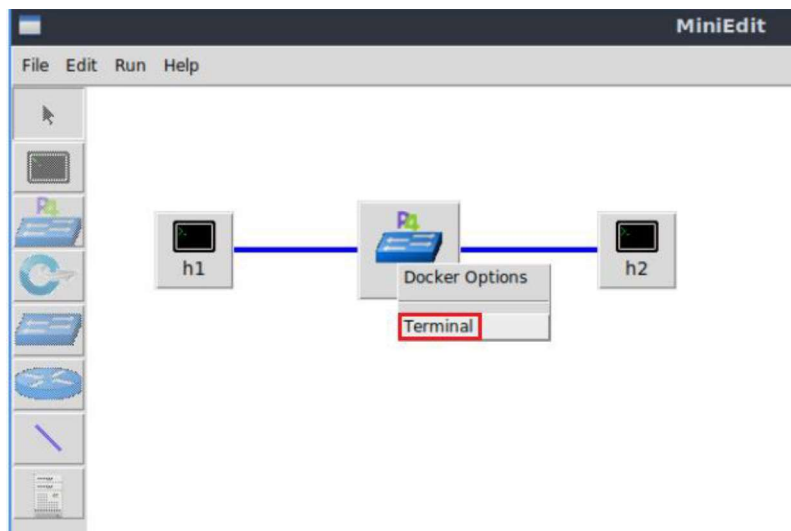
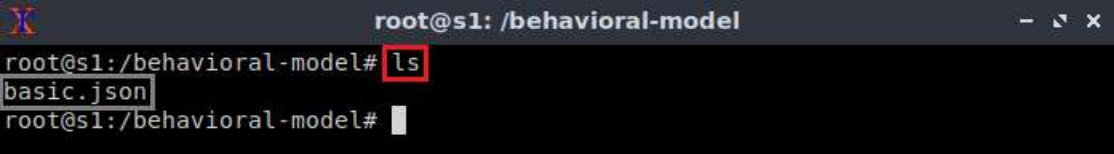


Figure 24. Starting the terminal on the switch.

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
```



The image shows a terminal window with a dark background. The title bar at the top reads 'root@s1: /behavioral-model'. The terminal prompt is 'root@s1:/behavioral-model#'. The user has entered the command 'ls', which is highlighted with a red box. The output of the command is 'basic.json', which is also highlighted with a red box. The prompt is now 'root@s1:/behavioral-model#' followed by a cursor.

Figure 25. Displaying the contents of the current directory in the switch s1.

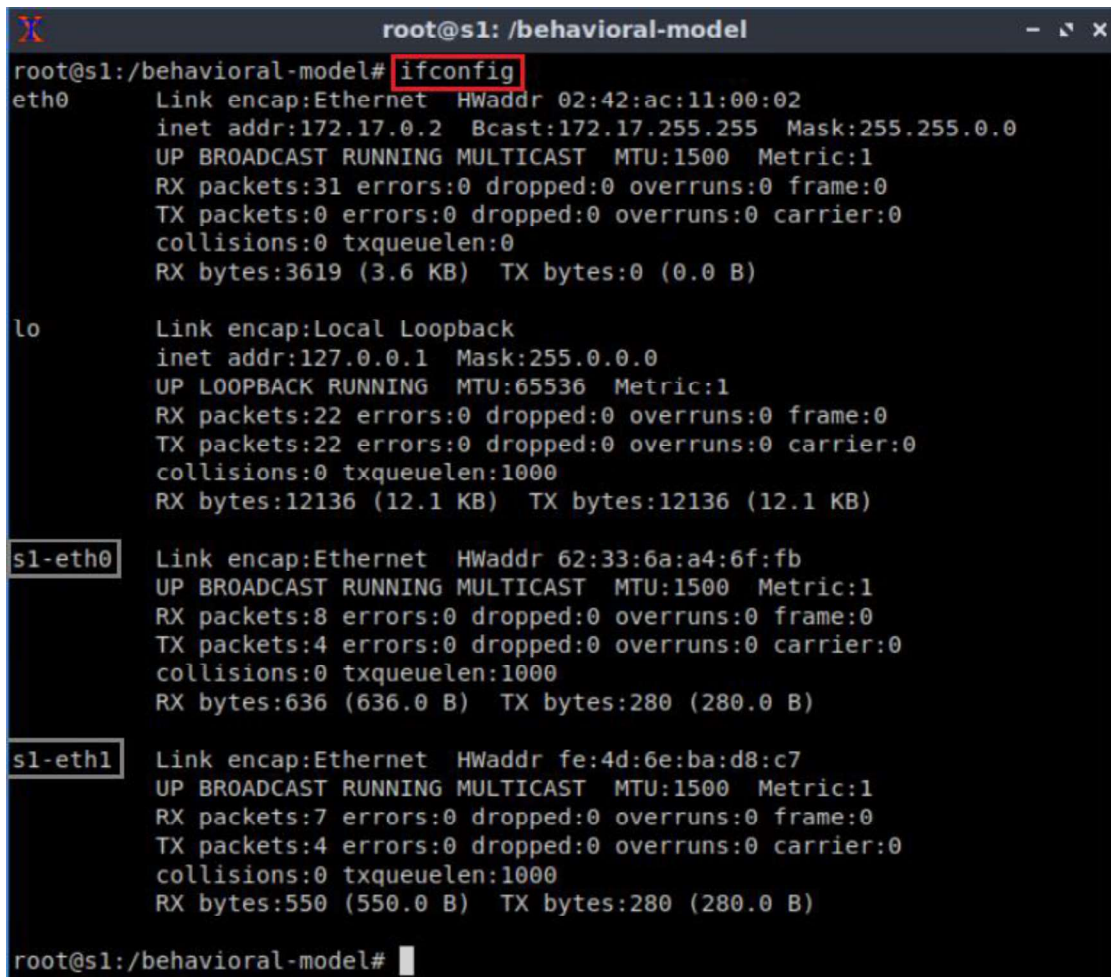
We can see that the switch contains the *basic.json* file that was downloaded to switch s1 after compiling the P4 program.

5 Configuring switch s1

5.1 Mapping the P4 program's ports

Step 1. Issue the following command to display the interfaces on the switch s1.

```
ifconfig
```

```

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#

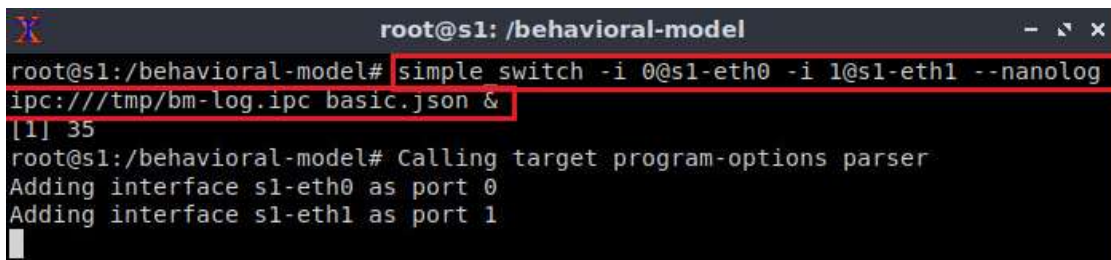
```

Figure 26. 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# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 35
root@s1: /behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

```

Figure 27. 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.

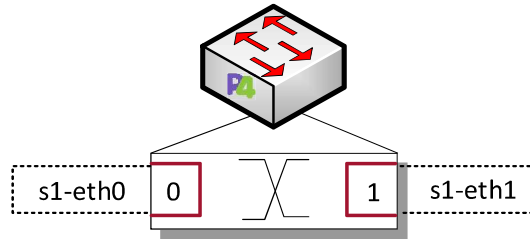


Figure 28. Mapping of the logical interface numbers (0, 1) to the Linux interfaces (*s1-eth0*, *s1-eth1*).

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 basic.json
&
[1] 33
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 29. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/lab3/rules.cmd
```

```

root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab3/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-00:00
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-00:01
action:         MyIngress.forward
runtime data:   00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#

```

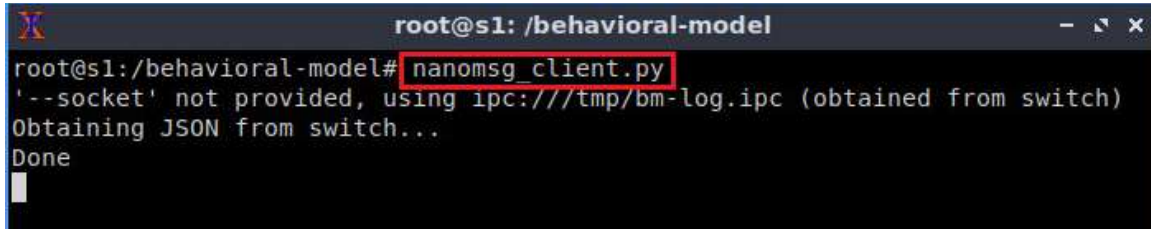
Figure 30. Loading the forwarding table entries into switch s1.

Now the forwarding table in the switch is populated.

6 Testing and verifying the P4 program

Step 1. Type the following command to initiate the `nanolog` client that will display the switch logs.

```
nanomsg_client.py
```

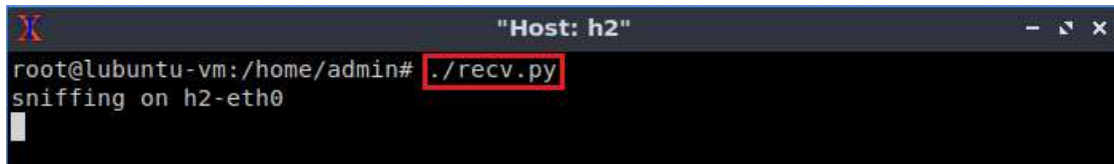
A terminal window titled "root@s1: /behavioral-model" with standard window controls. The prompt is "root@s1:/behavioral-model#". The command "nanomsg_client.py" is entered and highlighted with a red box. The output shows: "'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)", "Obtaining JSON from switch...", and "Done". A cursor is visible on the line following "Done".

```
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 31. Displaying switch s1 logs.

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

```
./recv.py
```

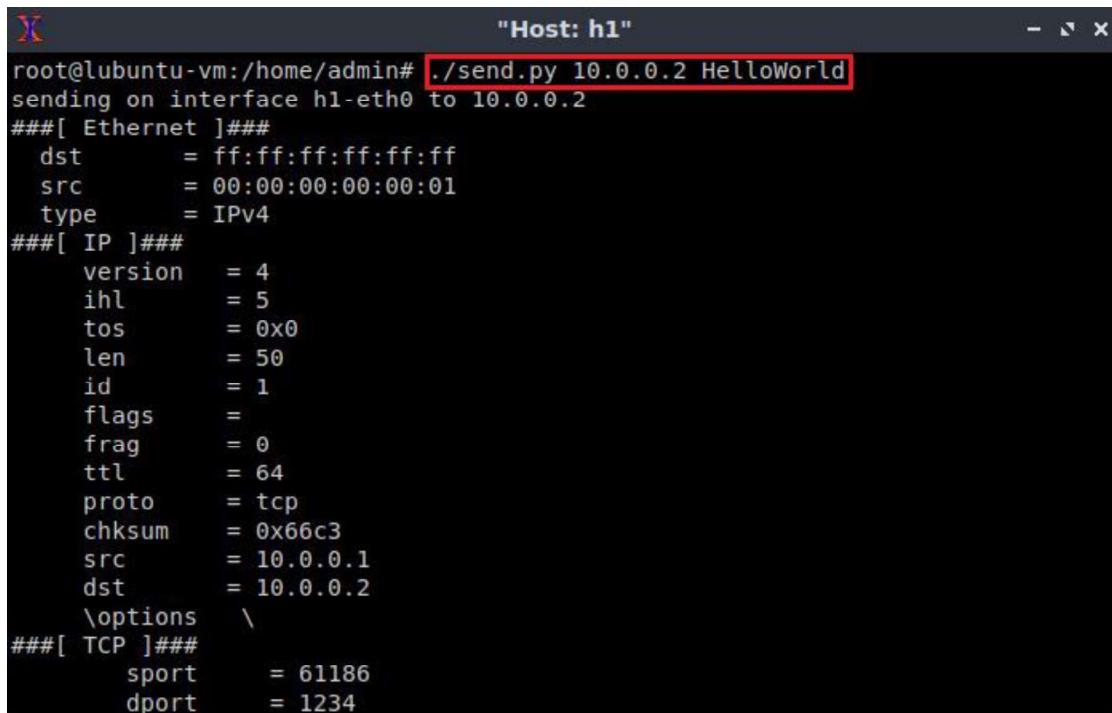
A terminal window titled "\"Host: h2\"" with standard window controls. The prompt is "root@lubuntu-vm:/home/admin#". The command "./recv.py" is entered and highlighted with a red box. The output shows "sniffing on h2-eth0". A cursor is visible on the line following the output.

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

Figure 32. Listening for incoming packets in host h2.

Step 3. On host h1's terminal, type the following command to send a packet to host h2.

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



```

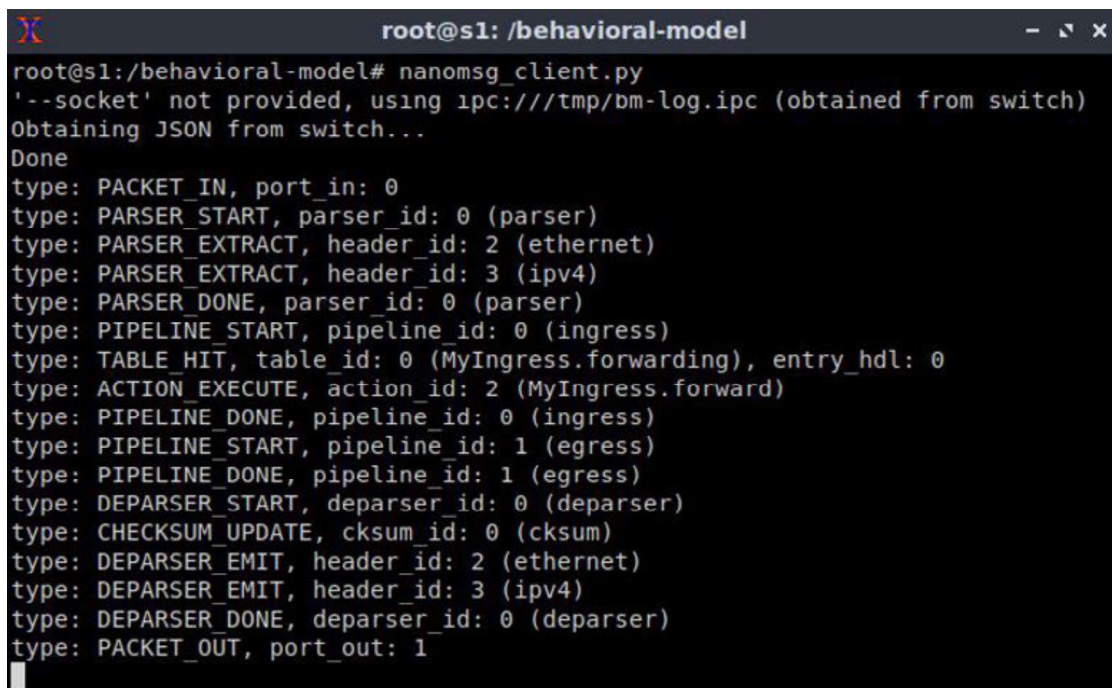
root@lubuntu-vm:/home/admin# ./send.py 10.0.0.2 HelloWorld
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 00:00:00:00:00:01
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 50
  id       = 1
  flags    =
  frag     = 0
  ttl      = 64
  proto    = tcp
  chksum   = 0x66c3
  src      = 10.0.0.1
  dst      = 10.0.0.2
  \options \
###[ TCP ]###
  sport    = 61186
  dport    = 1234

```

Figure 33. Sending a test packet from host h1 to host h2.

Now that the switch has a program with tables properly populated, the hosts are able to reach each other.

Step 4. Go back to switch s1 terminal and inspect the logs.



```

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: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (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 34. Inspecting the logs in switch s1.

The figure above shows the processing logic as the packet enters switch s1. The packet arrives on port 0 (`port in: 0`), then the parser starts extracting the headers. After the

parsing is done, the packet is processed in the ingress and in the egress pipelines. Then, the checksum update is executed and the deparser reassembles and emits the packet using port 1 (`port_out: 1`).

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

This concludes lab 3. Stop the emulation and then exit out of MiniEdit.

References

1. C. Cascaval, D. Daly. "P4 Architectures." [Online]. Available: <https://tinyurl.com/3zk8vs6a>.
2. P4 Language Tutorial. [Online]. Available: <https://tinyurl.com/2p9cen9e>.
3. P4lang/behavioral-model github repository. "*The BMv2 Simple Switch target.*" [Online]. Available: <https://tinyurl.com/vrasamm>.