UNIVERSITY OF
SOUTH CAROLINA

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Book Version: 06-25-2022

Principal Investigator: Jorge Crichigno

Award 2118311
"Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks"

Contents

Lab 1: Introduction to Mininet
Exercise 1: Building a Basic Topology
Lab 2: Introduction to P4 and BMv2
Exercise 2: Compiling and Running a P4 Program
Lab 3: P4 Program Building Blocks
Lab 4: Parser Implementation
Exercise 3: Parsing UDP and RTP
Lab 5: Introduction to Match-action Tables (Part 1)
Lab 6: Introduction to Match-action Tables (Part 2)
Exercise 4: Implementing NAT using Match-action Tables
Lab 7: Populating and Managing Match-action Tables at Runtime
Exercise 5: Configuring Match-action Tables at Runtime
Lab 8: Checksum Recalculation and Packet Deparsing
Exercise 6: Building a Packet Reflector

UNIVERSITY OF
SOUTH CAROLINA

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 1: Introduction to Mininet

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 Mininet 3
2 Invoke Mininet using the CLI 5
 2.1 Invoke Mininet using the default topology..... 5
 2.2 Test connectivity 9
3 Build and emulate a network in Mininet using the GUI 10
 3.1 Build the network topology 10
 3.2 Test connectivity 13
 3.3 Automatic assignment of IP addresses 16
 3.4 Save and load a Mininet topology 18
References 19

Overview

This lab provides an introduction to Mininet, a virtual testbed used for testing network tools and protocols. It demonstrates how to invoke Mininet from the command-line interface (CLI) utility and how to build and emulate topologies using a graphical user interface (GUI) application.

Objectives

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

1. Understand what Mininet is and why it is useful for testing network topologies.
2. Invoke Mininet from the CLI.
3. Construct network topologies using the GUI.
4. Save/load Mininet topologies using the GUI.

Lab settings

The information in Table 1 provides the credentials of the Client machine.

Table 1. Credentials to access the Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction to Mininet.
2. Section 2: Invoke Mininet using the CLI.
3. Section 3: Build and emulate a network in Mininet using the GUI.

1 Introduction to Mininet

Mininet is a virtual testbed enabling the development and testing of network tools and protocols. With a single command, Mininet can create a realistic virtual network on any type of machine (Virtual Machine (VM), cloud-hosted, or native). Therefore, it provides an inexpensive solution and streamlined development running in line with production networks[1]. Mininet offers the following features:

- Fast prototyping for new networking protocols.

- Simplified testing for complex topologies without the need of buying expensive hardware.
- Realistic execution as it runs real code on the Unix and Linux kernels.
- Open-source environment backed by a large community contributing extensive documentation.

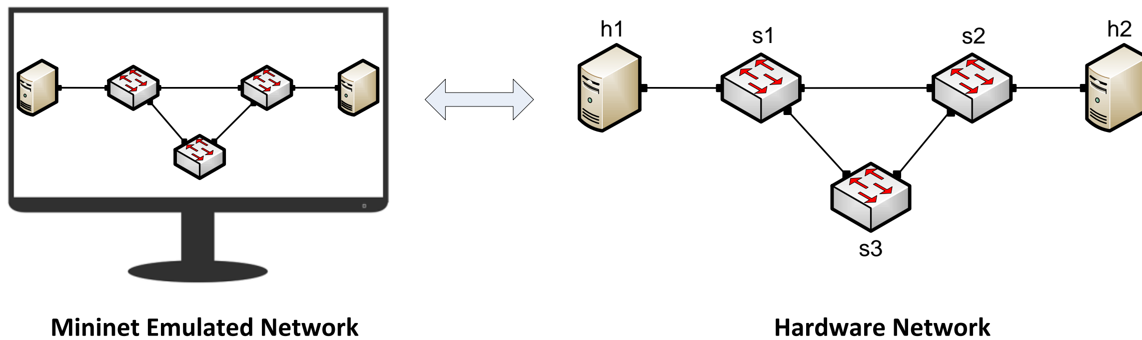


Figure 1. Hardware network vs. Mininet emulated network.

Mininet is useful for development, teaching, and research as it is easy to customize and interact with it through the CLI or the GUI. Mininet was originally designed to experiment with *OpenFlow*[2] and *Software-Defined Networking (SDN)*[3]. This lab, however, only focuses on emulating a simple network environment without SDN-based devices.

Mininet's logical nodes can be connected into networks. These nodes are sometimes called containers, or more accurately, *network namespaces*. Containers consume sufficiently fewer resources that networks of over a thousand nodes have created, running on a single laptop. A Mininet container is a process (or group of processes) that no longer has access to all the host system's native network interfaces. Containers are then assigned virtual Ethernet interfaces, which are connected to other containers through a virtual switch[4]. Mininet connects a host and a switch using a virtual Ethernet (veth) link. The veth link is analogous to a wire connecting two virtual interfaces, as illustrated below.

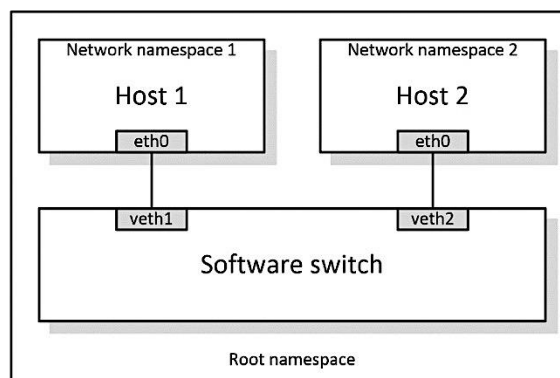


Figure 2. Network namespaces and virtual Ethernet links.

Each container is an independent network namespace, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and Address Resolution Protocol (ARP) tables.

Mininet provides network emulation opposed to simulation, allowing all network software at any layer to be simply run *as is*; i.e. nodes run the native network software of the physical machine. On the other hand, in a simulated environment applications and protocol implementations need to be ported to run within the simulator before they can be used.

2 Invoke Mininet using the CLI

In following subsections, you will start Mininet using the Linux CLI.

2.1 Invoke Mininet using the default topology

Step 1. Launch a Linux terminal by clicking on the Linux terminal icon in the task bar.

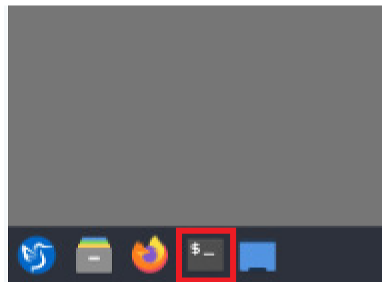


Figure 3. Linux terminal icon.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). A CLI is a program that takes commands from the keyboard and sends them to the operating system for execution.

Step 2. To start a minimal topology, enter the command shown below. When prompted for a password, type `password` and hit enter. Note that the password will not be visible as you type it.

```
sudo mn
```

```
admin@lubuntu-vm: ~
File  Actions  Edit  View  Help
admin@lubuntu-vm: ~
admin@lubuntu-vm:~$ sudo mn
[sudo] password for admin:
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:150:
"?
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:450:
 "!="?
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:150:
"?
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:450:
 "!="?
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet>
```

Figure 4. Starting Mininet using the CLI.

The above command starts Mininet with a minimal topology, which consists of a switch connected to two hosts as shown below.

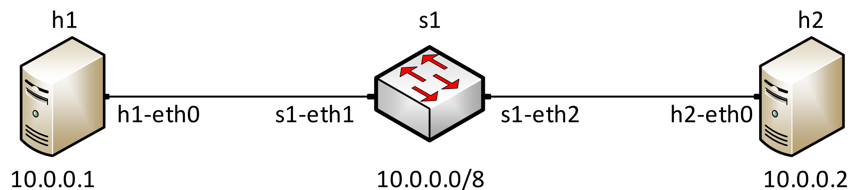


Figure 5. Mininet's default minimal topology.

When issuing the `sudo mn` command, Mininet initializes the topology and launches its command line interface which looks like this:

```
containernet>
```

Step 3. To display the list of Mininet CLI commands and examples on their usage, type the following command:

```
help
```

```
admin@lubuntu-vm: ~
File  Actions  Edit  View  Help
admin@lubuntu-vm: ~
containernet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperfudp  nodes        pingpair      py      switch
dpctl  help   link      noecho       pingpairfull  quit    time
dump   intfs  links     pingall      ports         sh      x
exit   iperf  net       pingallfull  px            source  xterm

You may also send a command to a node using:
  <node> command {args}
For example:
  mininet> h1 ifconfig

The interpreter automatically substitutes IP addresses
for node names when a node is the first arg, so commands
like
  mininet> h2 ping h3
should work.

Some character-oriented interactive commands require
noecho:
  mininet> noecho h2 vi foo.py
However, starting up an xterm/gterm is generally better:
  mininet> xterm h2

containernet>
```

Figure 6. Mininet's `help` command.

Step 4. To display the available nodes, type the following command:

```
nodes
```

```
admin@lubuntu-vm: ~
File  Actions  Edit  View  Help
admin@lubuntu-vm: ~
containernet> nodes
available nodes are:
c0 h1 h2 s1
containernet>
```

Figure 7. Mininet's `nodes` command.

The output of the `nodes` command shows that there is a controller (c0), two hosts (host h1 and host h2), and a switch (s1).

Step 5. It is useful sometimes to display the links between the devices in Mininet to understand the topology. Issue the command shown below to see the available links.

```
net
```

```
containernet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
containernet>
```

Figure 8. Mininet's `net` command.

The output of the `net` command shows that:

1. Host h1 is connected using its network interface *h1-eth0* to the switch on interface *s1-eth1*.
2. Host h2 is connected using its network interface *h2-eth0* to the switch on interface *s1-eth2*.
3. Switch s1:
    a. Has a loopback interface *lo*.
    b. Connects to *h1-eth0* through interface *s1-eth1*.
    c. Connects to *h2-eth0* through interface *s1-eth2*.
4. Controller c0 does not have any connection.

Mininet allows you to execute commands on a specific device. To issue a command for a specific node, you must specify the device first, followed by the command.

Step 6. To proceed, issue the command:

```
h1 ifconfig
```

```
containernet> h1 ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 3a:63:b8:06:23:9c  txqueuelen 1000  (Ethernet)
        RX packets 30  bytes 3449 (3.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

containernet>
```

Figure 9. Output of `h1 ifconfig` command.

This command `h1 ifconfig` executes the `ifconfig` Linux command on host h1. The command shows host h1's interfaces. The display indicates that host h1 has an interface *h1-eth0* configured with IP address 10.0.0.1, and another interface lo configured with IP address 127.0.0.1 (loopback interface).

2.2 Test connectivity

Mininet's default topology assigns the IP addresses 10.0.0.1/8 and 10.0.0.2/8 to host h1 and host h2 respectively. To test connectivity between them, you can use the command `ping`. The `ping` command operates by sending Internet Control Message Protocol (ICMP) Echo Request messages to the remote computer and waiting for a response or reply. Information available includes how many responses are returned and how long it takes for them to return.

Step 1. On the CLI, type the command shown below. The command `h1 ping 10.0.0.2` tests the connectivity between host h1 and host h2. To stop the test, press `Ctrl+c`. The figure below shows a successful connectivity test. Host h1 (10.0.0.1) sent four packets to host h2 (10.0.0.2) and successfully received the expected responses.

```
h1 ping 10.0.0.2
```

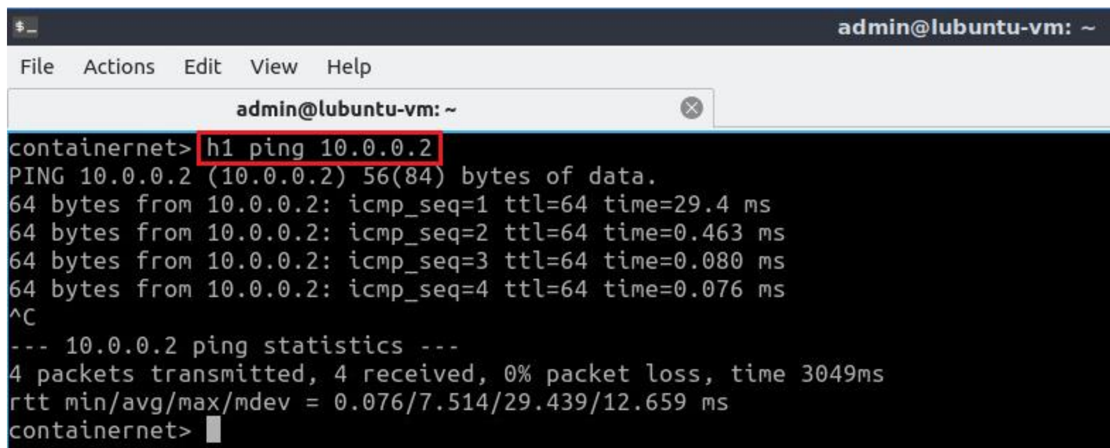


Figure 10. Connectivity test between host h1 and host h2.

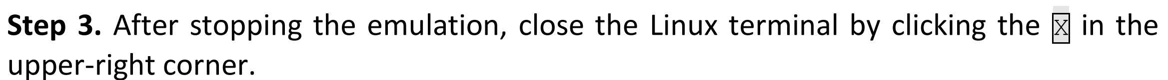
Step 2. Stop the emulation by typing the following command:

```
exit
```

```
admin@lubuntu-vm: ~
File  Actions  Edit  View  Help
admin@lubuntu-vm: ~
containernet> exit
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 619.612 seconds
admin@lubuntu-vm:~$
```

Figure 11. Stopping the emulation using `exit`.

If Mininet were to crash for any reason, the `sudo mn – c` command can be utilized to clean a previous instance. However, the `sudo mn -c` command is often used within the Linux terminal and not the Mininet CLI.



Step 3. After stopping the emulation, close the Linux terminal by clicking the `X` in the upper-right corner.

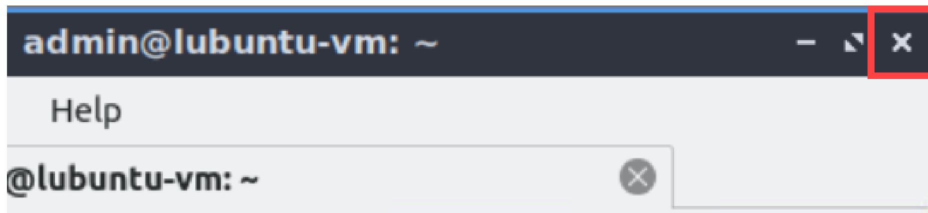


Figure 12. Closing the Linux CLI.

3 Build and emulate a network in Mininet using the GUI

In this section, you will use the application MiniEdit to deploy the topology illustrated below. MiniEdit is a simple GUI network editor for Mininet.

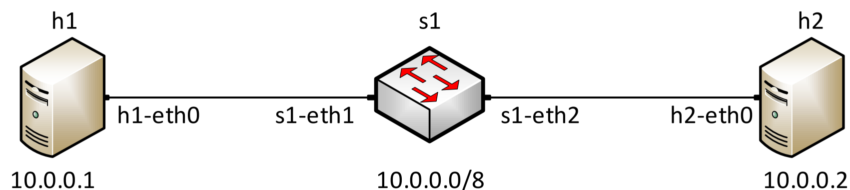


Figure 13. Lab topology.

3.1 Build the network 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`. MiniEdit will start, as illustrated below.

Figure 14. MiniEdit Desktop shortcut.

MiniEdit will start, as illustrated below.

MiniEdit

File Edit Run Help

(1) Select

(2) Host

(3) P4 switch (Docker)

(4) OpenFlow switch

(5) Legacy switch

(6) Legacy router

(7) Link

(8) Controller

(9) Run

(10) Stop

Run

Stop

Figure 15. MiniEdit Graphical User Interface (GUI).

The main buttons are:

1. Select: allows selection/movement of the devices. Pressing *Del* on the keyboard after selecting the device removes it from the topology.
2. Host: allows addition of a new host to the topology. After clicking this button, click anywhere in the blank canvas to insert a new host.
3. P4 switch (Docker): allows the addition of P4 switch. After clicking this button, click anywhere in the blank canvas to insert the P4 switch.
4. OpenFlow switch: allows the addition of a new OpenFlow-enabled switch. After clicking this button, click anywhere in the blank canvas to insert the switch.
5. Legacy switch: allows the addition of a new Ethernet switch to the topology. After clicking this button, click anywhere in the blank canvas to insert the switch.
6. Legacy router: allows the addition of a new legacy router to the topology. After clicking this button, click anywhere in the blank canvas to insert the router.
7. Link: connects devices in the topology (mainly switches and hosts). After clicking this button, click on a device and drag to the second device to which the link is to be established.
8. Controller: allows the addition of a new OpenFlow controller.
9. Run: starts the emulation. After designing and configuring the topology, click the run button.
10. Stop: stops the emulation.

Step 2. To build the topology illustrated in Figure 13, two hosts and one switch must be deployed. Deploy these devices in MiniEdit, as shown below.

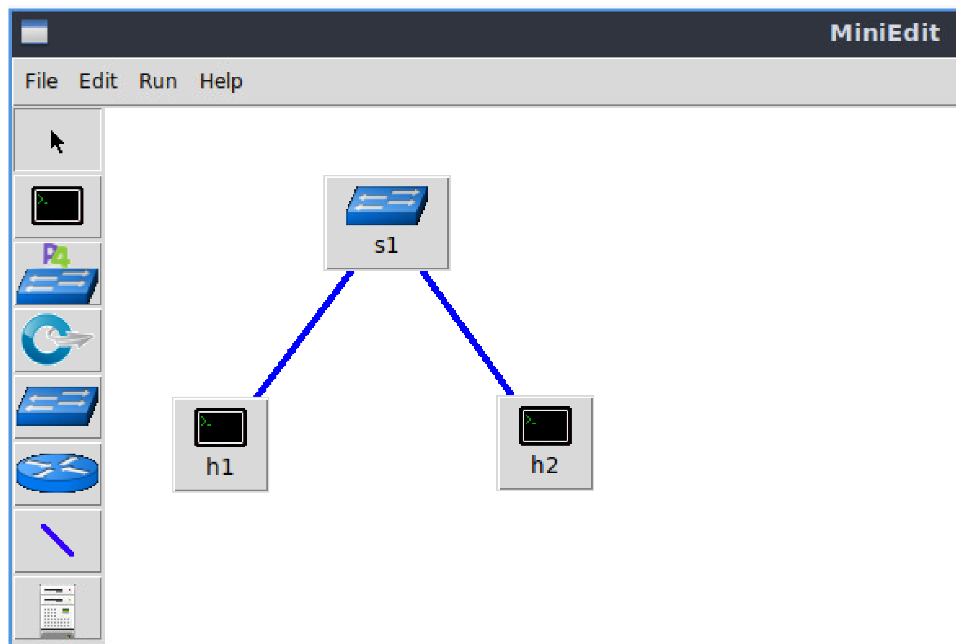


Figure 16. MiniEdit's topology.

Use the buttons described in the previous step to add and connect devices. The configuration of IP addresses is described in Step 3.

Step 3. Configure the IP addresses of host h1 and host h2. Host h1's IP address is 10.0.0.1/8 and host h2's IP address is 10.0.0.2/8. A host can be configured by holding the right click and selecting properties on the device. For example, host h2 is assigned the IP address 10.0.0.2/8 in the figure below. Click *OK* for the settings to be applied.

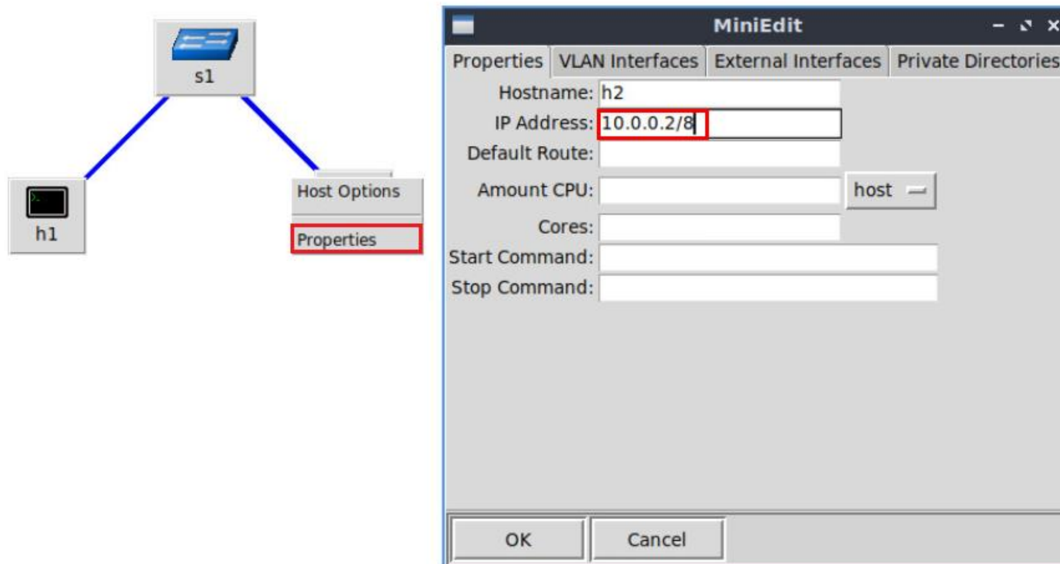


Figure 17. Configuration of a host's properties.

3.2 Test connectivity

Before testing the connection between host h1 and host h2, the emulation must be started.

Step 1. Click the *Run* button to start the emulation. The emulation will start and the buttons of the MiniEdit panel will gray out, indicating that they are currently disabled.

Figure 18. Starting the emulation.

Step 2. Open a terminal by right-clicking on host h1 and select *Terminal*. This opens a terminal on host h1 and allows the execution of commands on the host h1. Repeat the procedure on host h2.

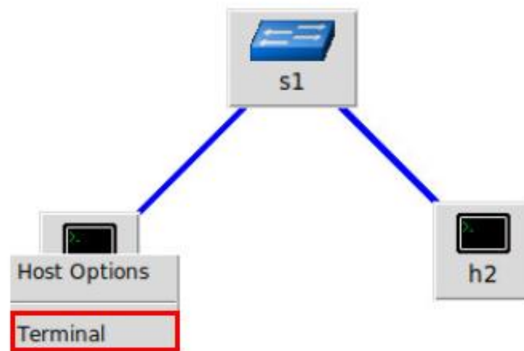


Figure 19. Opening a terminal on host h1.

The network and terminals at host h1 and host h2 will be available for testing.

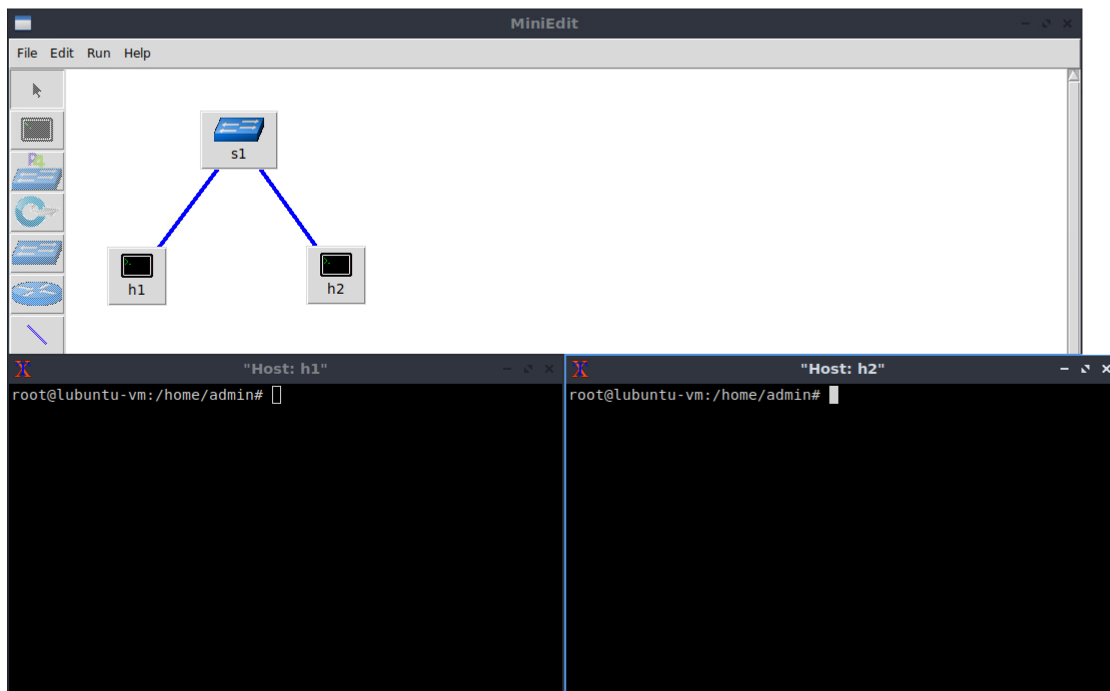


Figure 20. Terminals at host h1 and host h2.

Step 3. On host h1's terminal, type the command shown below to display its assigned IP addresses. The interface *h1-eth0* at host h1 should be configured with the IP address 10.0.0.1 and subnet mask 255.0.0.0.

```
ifconfig
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 22:6b:8e:fc:b9:0c  txqueuelen 1000  (Ethernet)
        RX packets 28  bytes 3272 (3.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin#
```

Figure 21. Output of `ifconfig` command on host h1.

Repeat Step 3 on host h2. Its interface *h2-eth0* should be configured with IP address 10.0.0.2 and subnet mask 255.0.0.0.

Step 4. On host h1's terminal, type the command shown below. This command tests the connectivity between host h1 and host h2. To stop the test, press `Ctrl+c`. The figure below shows a successful connectivity test. Host h1 (10.0.0.1) sent six packets to host h2 (10.0.0.2) and successfully received the expected responses.

```
ping 10.0.0.2
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.694 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.073 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.067/0.228/0.694/0.268 ms
root@lubuntu-vm:/home/admin#
```

Figure 22. Connectivity test using `ping` command.

Step 5. Stop the emulation by clicking on the *Stop* button.

Figure 23. Stopping the emulation.

3.3 Automatic assignment of IP addresses

In the previous section, you manually assigned IP addresses to host h1 and host h2. An alternative is to rely on Mininet for an automatic assignment of IP addresses (by default, Mininet uses automatic assignment), which is described in this section.

Step 1. Remove the manually assigned IP address from host h1. Right-click on host h1 and select *Properties*. Delete the IP address, leaving it unassigned, and press the *OK* button as shown below. Repeat the procedure on host h2.

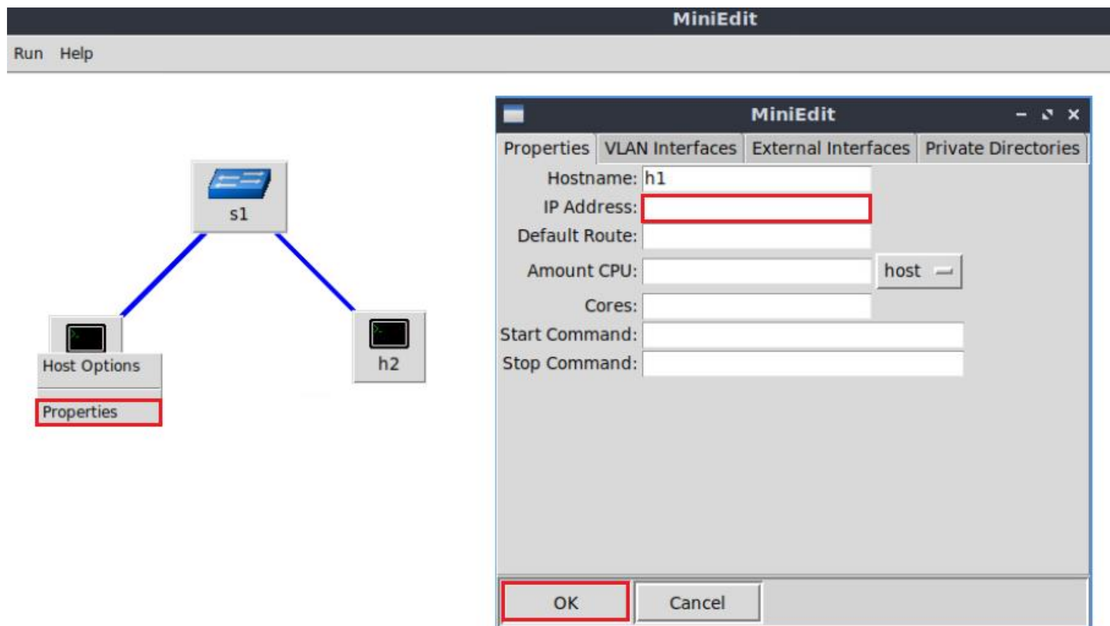


Figure 24. Host h1 properties.

Step 2. In the MiniEdit application, navigate to *Edit > Preferences*. The default IP base is 10.0.0.0/8. Modify this value to 15.0.0.0/8, and then press the *OK* button.

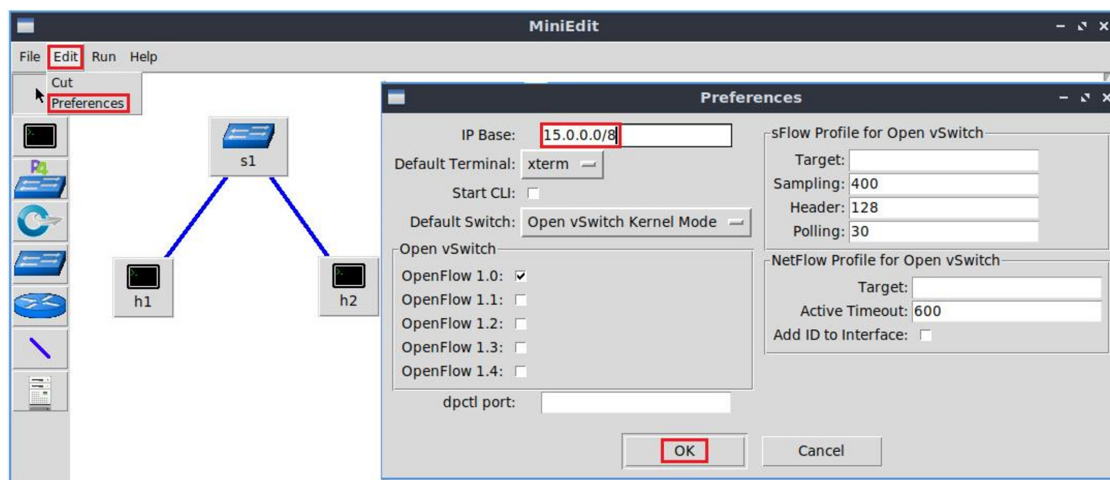


Figure 25. Modification of the IP Base (network address and prefix length).

Step 3. Run the emulation again by clicking on the *Run* button. The emulation will start and the buttons of the MiniEdit panel will be disabled.

Figure 26. Starting the emulation.

Step 4. Open a terminal by right-clicking on host h1 and select *Terminal*.

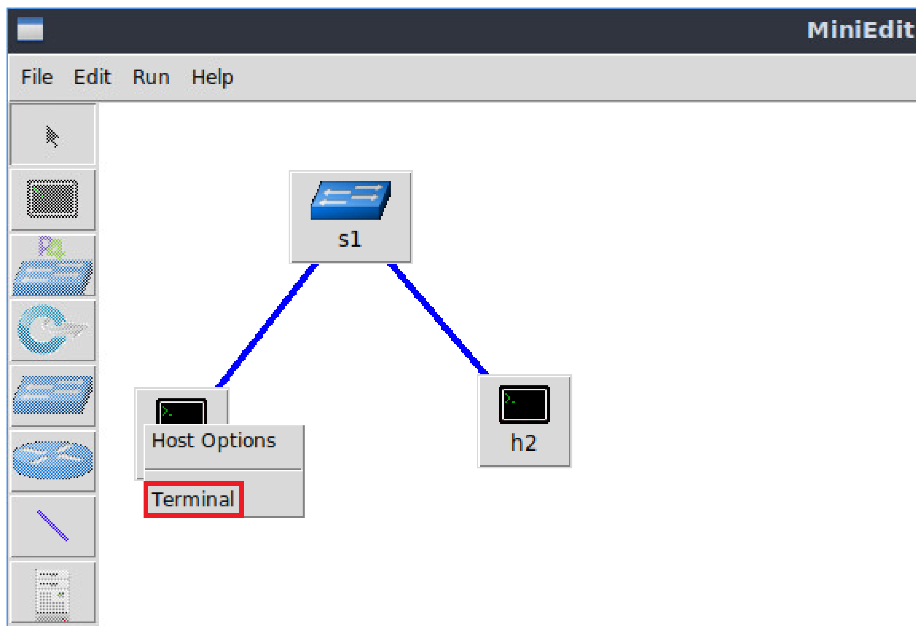


Figure 27. Opening a terminal on host h1.

Step 5. Type the command shown below to display the IP addresses assigned to host h1. The interface *h1-eth0* at host h1 now has the IP address 15.0.0.1 and subnet mask 255.0.0.0.

```
ifconfig
```

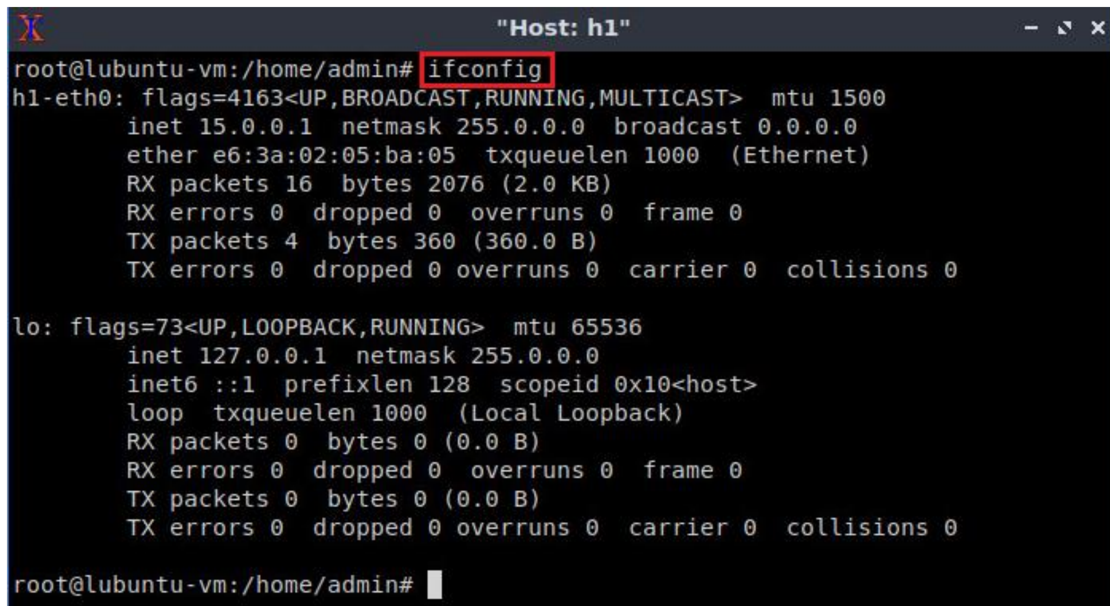


Figure 28. Output of `ifconfig` command on host h1.

You can also verify the IP address assigned to host h2 by repeating Steps 4 and 5 on host h2's terminal. The corresponding interface *h2-eth0* at host h2 has now the IP address 15.0.0.2 and subnet mask 255.0.0.0.

Step 6. Stop the emulation by clicking on *Stop* button.

Figure 29. Stopping the emulation.

3.4 Save and load a Mininet topology

In this section you will save and load a Mininet topology. It is often useful to save the network topology, particularly when its complexity increases. MiniEdit enables you to save the topology to a file.

Step 1. In the MiniEdit application, save the current topology by clicking *File*. Provide a name for the topology and notice *myTopology* as the topology name. Ensure you are in the *lab1* folder and click *Save*.

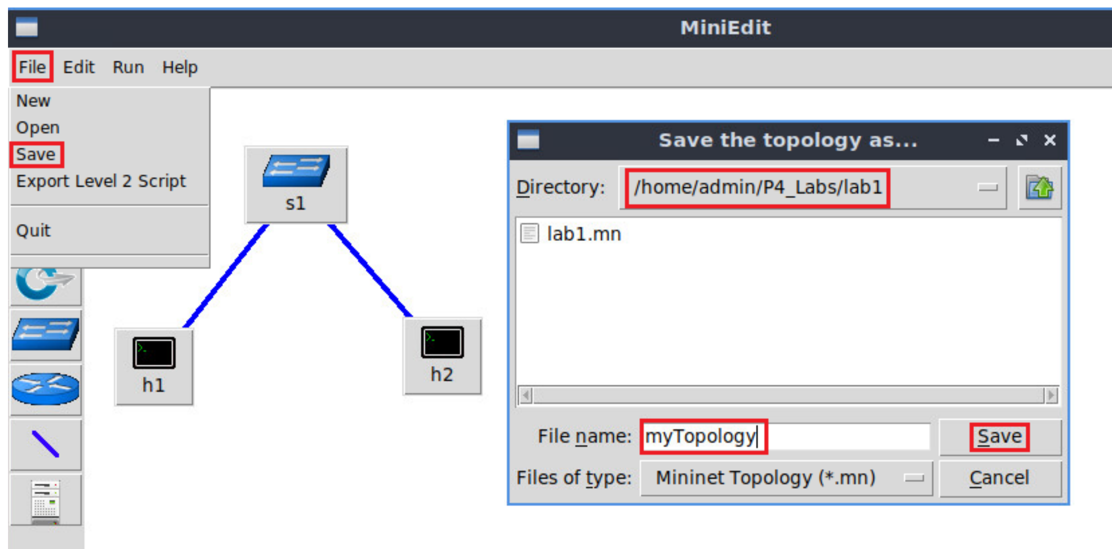


Figure 30. Saving the topology.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab1* folder and search for the topology file called *lab1.mn* and click on Open. A new topology will be loaded to MiniEdit.

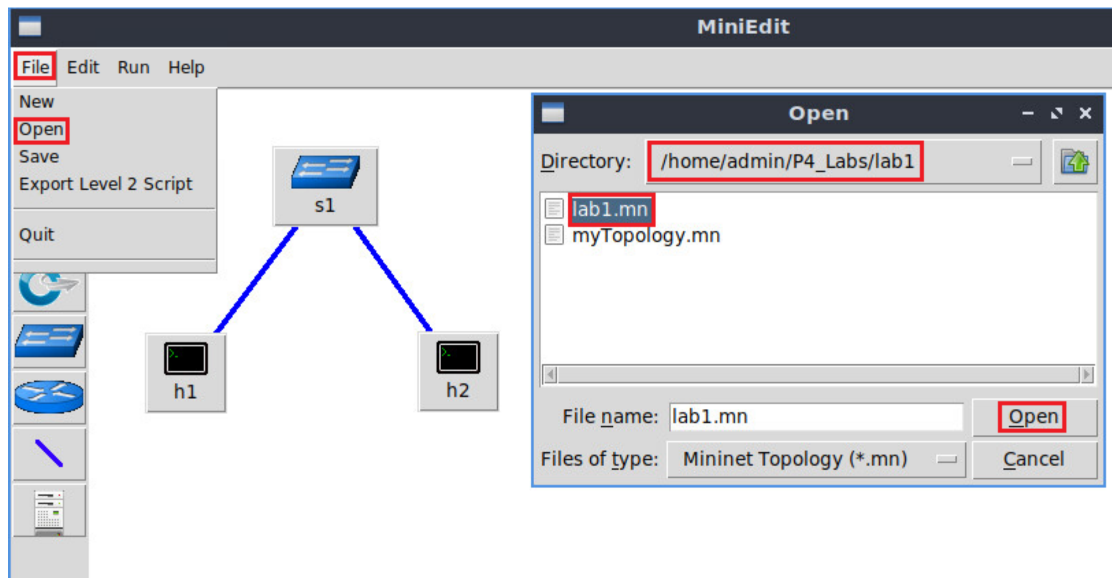


Figure 31. Opening a topology.

This concludes lab 1. Stop the emulation and then exit out of MiniEdit and the Linux terminal.

References

1. Mininet walkthrough. [Online]. Available: http://Mininet.org.
2. Mckeown N., Anderson T., Balakrishnan H., Parulkar G., Peterson L., Rexford J., Shenker S., Turner J., "*OpenFlow*," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, p. 69, 2008.

3. Esch J., "*Prolog to, software-defined networking: a comprehensive survey*," Proceedings of the IEEE, vol. 103, no. 1, pp. 10–13, 2015.
4. Dordal P., "*An Introduction to computer networks*,". [Online]. Available: https://intronetworks.cs.luc.edu/.
5. Lantz B., Gee G. "*MiniEdit: a simple network editor for Mininet.*" 2013. [Online]. Available: https://github.com/Mininet/Mininet/blob/master/examples.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Exercise 1: Building a Basic Topology

Document Version: 01-14-2022

Award 2118311
“Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks”

Contents

1 Exercise description 3
1.1 Credentials 3
1.2 Exercise topology 3
2 Deliverables.. 3

1 Exercise description

In this exercise, you will build a topology and run Mininet commands to verify the configuration. Additionally, you will perform a connectivity test.

1.1 Credentials

The information in Table 1 provides the credentials to access the Client's virtual machine.

Table 1. Credentials to access the Client's virtual machine.

Device	Account	Password
Client	admin	password

1.2 Exercise topology

The topology comprises two legacy switches and two end hosts.

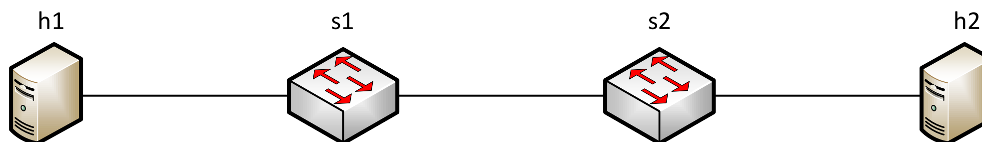


Figure 1. Exercise topology.

2 Deliverables

Follow the steps below to complete the exercise.

a) Open MiniEdit by double-clicking the shortcut on the desktop. If a password is required type `password`.

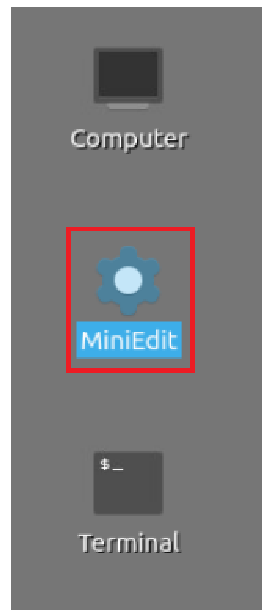


Figure 2. MiniEdit shortcut.

b) Using end hosts and legacy switches, build the topology presented in Figure 1. Those devices are highlighted in the figure below.

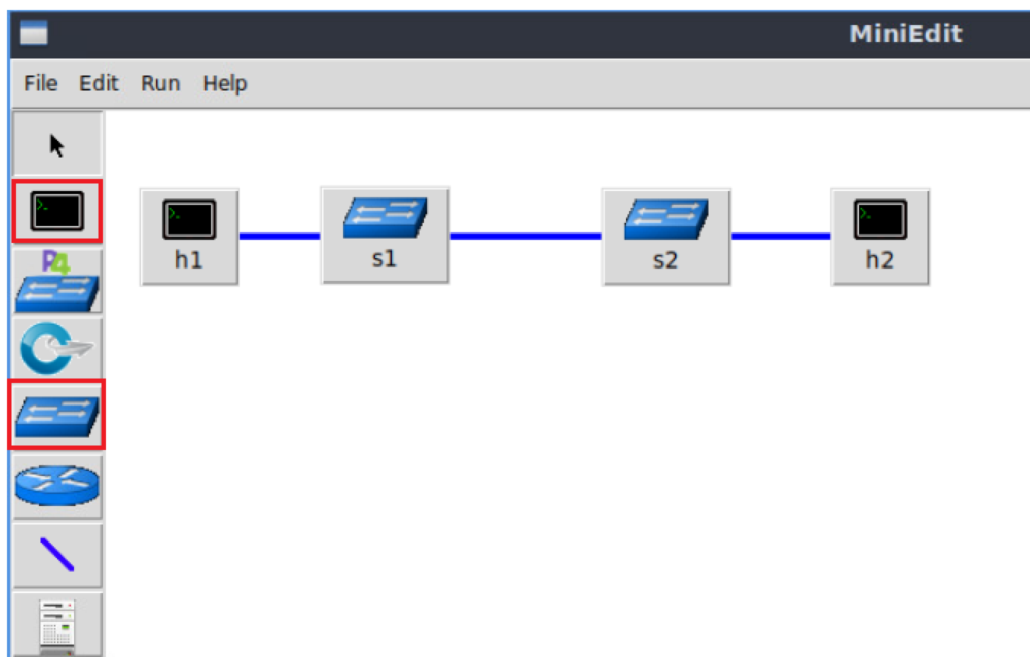


Figure 3. Building a topology using end hosts and legacy switches available in MiniEdit.

c) Enable Mininet's CLI navigating into *Edit->Preferences* and set the *Start CLI* box.

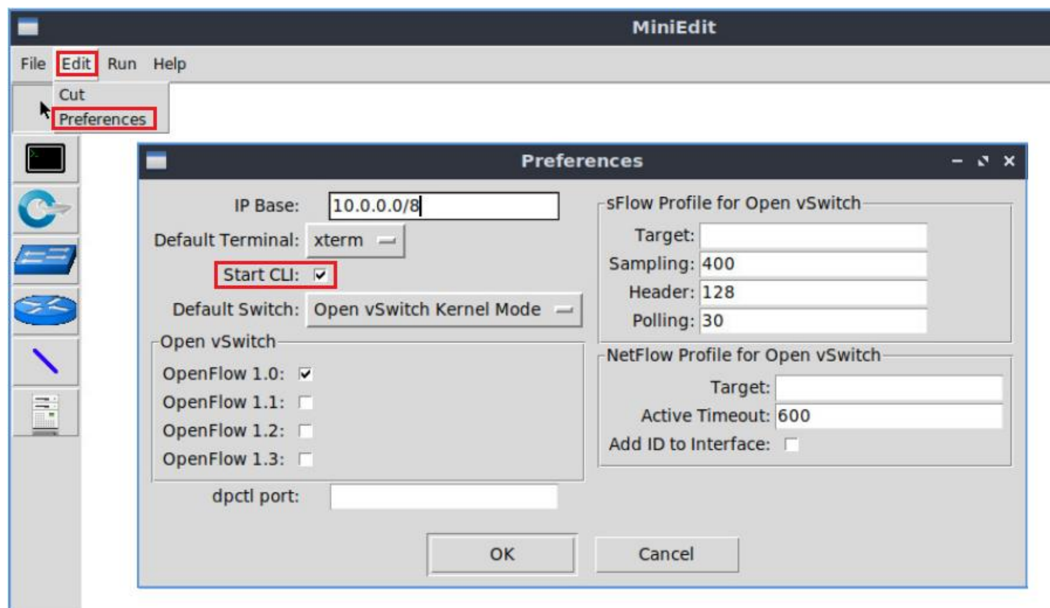


Figure 4. Enabling Mininet’s CLI.

d) In the Mininet CLI, run the corresponding commands to verify the name of the interfaces, links, and nodes in the topology. Which interface in switch s1 connects to switch s2?

e) In the hosts’ CLI, verify the IP and MAC addresses. Report the MAC address of host h2.

f) In a host’s terminal, perform a connectivity test between host h1 and host h2. Is the test successful?

UNIVERSITY OF
SOUTH CAROLINA

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 2: Introduction to P4 and BMv2

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 3
  1.1 Workflow of a P4 program 4
  1.2 Workflow used in this lab series 5
2 Lab topology 6
  2.1 Verifying connectivity between host h1 and host h2 7
3 Loading the P4 program 8
  3.1 Loading the programming environment 9
  3.2 Compiling and loading the P4 program to switch s1 11
  3.3 Verifying the configuration 13
4 Configuring switch s1 14
  4.1 Mapping P4 program's ports 14
  4.2 Loading the rules to the switch 16
References 17

Overview

This lab introduces programmable data plane switches and their role in the Software-defined Networking (SDN) paradigm. The lab introduces the Programming Protocol-independent Packet Processors (P4), the de facto programming language used to describe the behavior of the data planes of programmable switches. The focus of this lab is to provide a high-level overview of the general lifecycle of programming, compiling, and running a P4 program on a software switch.

Objectives

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

1. Define the need for SDN and data plane programmability.
2. Understand the structure of a P4 program.
3. Compile a simple P4 program and deploy it to a software switch.
4. Start the switch daemon and allocate virtual interfaces to the switch.
5. Perform a connectivity test to verify the correctness of the program.

Lab settings

Table 1 contains the credentials of the virtual machine used for this lab.

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.
2. Section 2: Lab topology.
3. Section 3: Loading the P4 program.
4. Section 4: Configuring switch s1.

1 Introduction

Since the emergence of the world wide web and the explosive growth of the Internet in the 1990s, the networking industry has been dominated by closed and proprietary

hardware and software. The progressive reduction in the flexibility of protocol design caused by standardized requirements, which cannot be easily removed to enable protocol changes, has perpetuated the status quo. This protocol ossification[1, 2] has been characterized by a slow innovation pace at the hand of few network vendors. As an example, after being initially conceived by Cisco and VMware[3], the Application Specific Integrated Circuit (ASIC) implementation of the Virtual Extensible LAN (VXLAN)[4], a simple frame encapsulation protocol, took several years, a process that could have been reduced to weeks by software implementations. The design cycle of switch ASICs has been characterized by a lengthy, closed, and proprietary process that usually takes years. Such process contrasts with the agility of the software industry.

The programmable forwarding can be viewed as a natural evolution of Software-Defined Networking (SDN), where the software that describes the behavior of how packets are processed, can be conceived, tested, and deployed in a much shorter time span by operators, engineers, researchers, and practitioners in general. The de-facto standard for defining the forwarding behavior is the P4 language[5], which stands for Programming Protocol-independent Packet Processors. Essentially, P4 programmable switches have removed the entry barrier to network design, previously reserved to network vendors.

1.1 Workflow of a P4 program

Programming a P4 switch, whether a hardware or a software target, requires a software development environment that includes a compiler. Consider Figure 1. The compiler maps the target-independent P4 source code (P4 program) to the specific platform. The compiler, the architecture model, and the target device are vendor specific and are provided by the vendor. The P4 source code on the other hand is supplied by the user.

The compiler generates two artifacts after compiling the P4 program. First, it generates a data plane configuration (Data plane runtime) that implements the forwarding logic specified in the P4 input program. This configuration includes the instructions and resource mappings for the target. Second, it generates runtime APIs that are used by the control plane / user to interact with the data plane. Examples include adding/removing entries from match-action tables and reading/writing the state of extern objects (e.g., counters, meters, registers). The APIs contain the information needed by the control plane to manipulate tables and objects in the data plane, such as the identifiers of the tables, fields used for matches, keys, action parameters, and others.

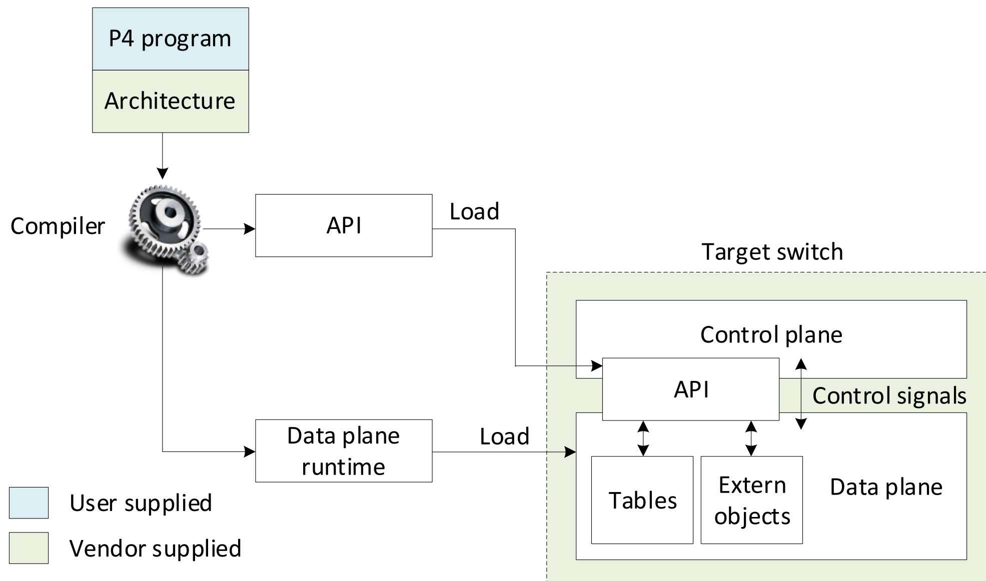


Figure 1. Generic workflow design. The compiler, the architecture model, and the target switch are provided by the vendor of the device. The P4 source code is customized by the user. The compiler generates a data plane runtime to be loaded into the target, and the APIs used by the control plane to communicate with the data plane at runtime.

1.2 Workflow used in this lab series

This section demonstrates the P4 workflow that will be used in this lab series. Consider Figure 2. We will use the Visual Studio Code (VS Code) as the editor to modify the *basic.p4* program. Then, we will use the *p4c* compiler with the V1Model architecture to compile the user supplied P4 program (*basic.p4*). The compiler will generate a JSON output (i.e., *basic.json*) which will be used as the data plane program by the switch daemon (i.e., simple_switch). Finally, we will use the `simple_switch_CLI` at runtime to populate and manipulate table entries in our P4 program. The target switch (vendor supplied) used in this lab series for testing and debugging P4 programs is the behavioral model version 2 (BMv2)[6].

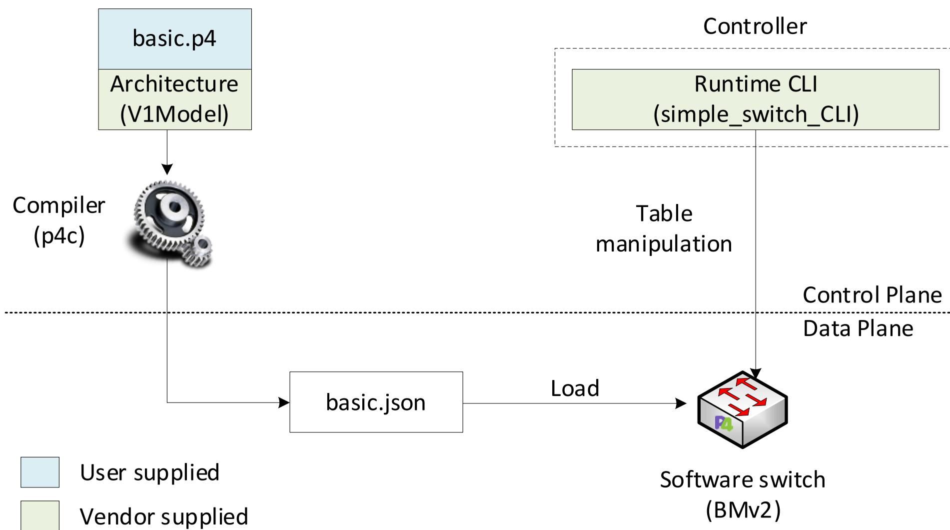


Figure 2. Workflow used in this lab series.

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.

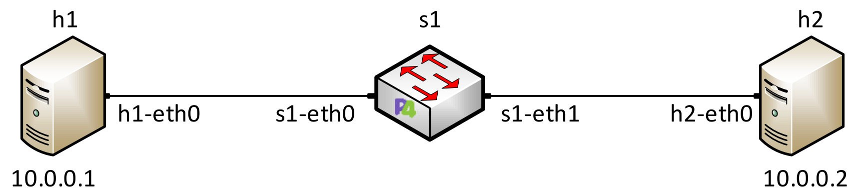


Figure 3. Lab topology.

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. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab2*, select the file *lab2.mn,* and click on *Open*.

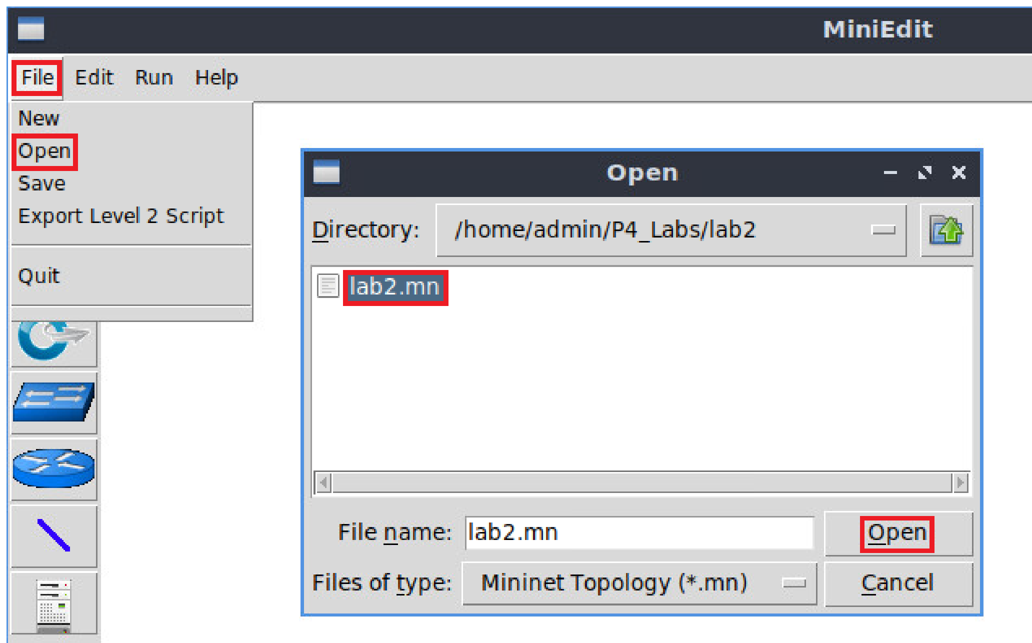


Figure 5. Opening a topology in MiniEdit.

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 6. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

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

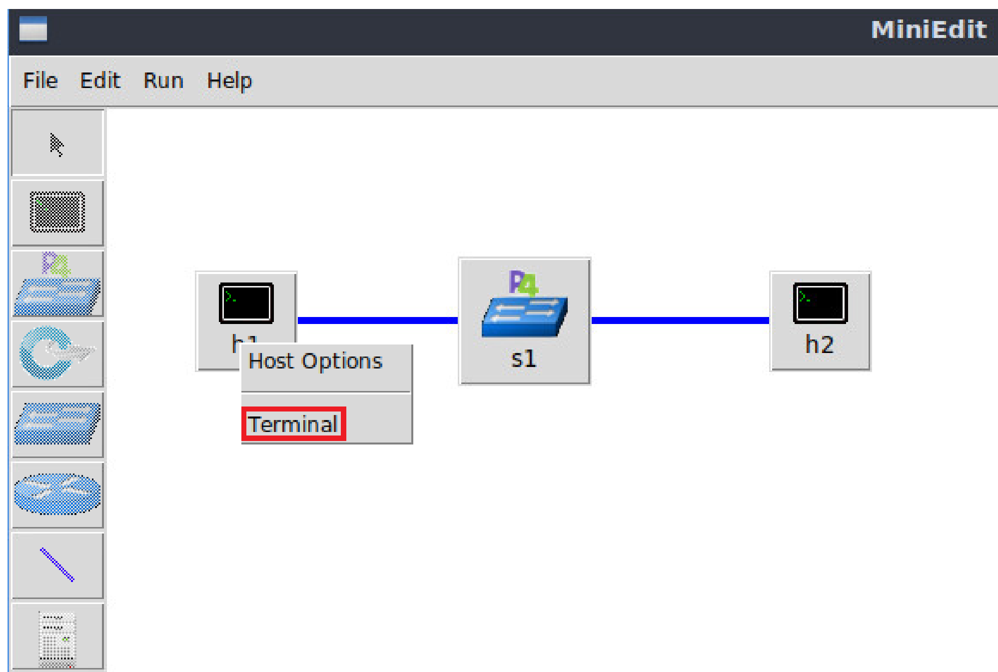


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3067ms
pipe 4
root@lubuntu-vm:/home/admin#
```

Figure 8. 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 into the switch.

3 Loading the P4 program

This section shows the steps required to implement a P4 program. It describes 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.

VS Code will be used as the editor to modify P4 programs. It highlights the syntax of P4 and provides an integrated terminal where the P4 compiler will be invoked. The P4 compiler that will be used is *p4c*, the reference compiler for the P4 programming language.

p4c supports both $P4_{14}$ and $P4_{16}$, but in this lab series we will only focus on $P4_{16}$ since it is the newer version and is currently being supported by major programming ASIC manufacturers[7].

3.1 Loading the programming environment

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

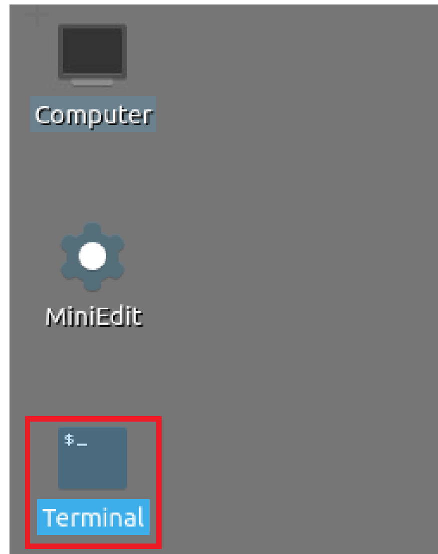


Figure 9. Shortcut to open a Linux terminal.

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

```
code P4_Labs/lab2
```

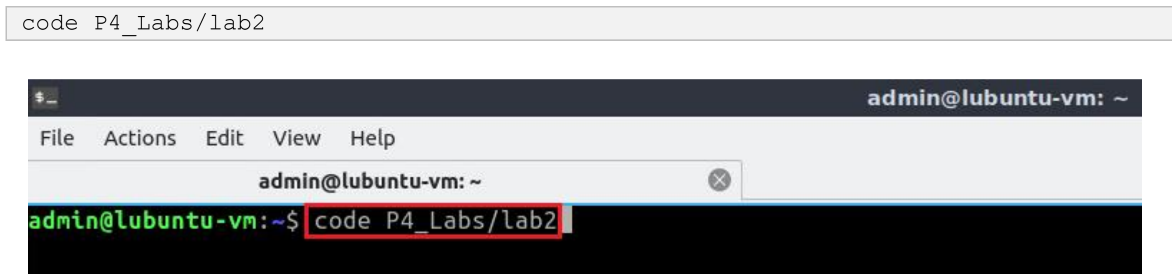


Figure 10. Launching the editor and opening the lab2 directory.

Step 3. 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.

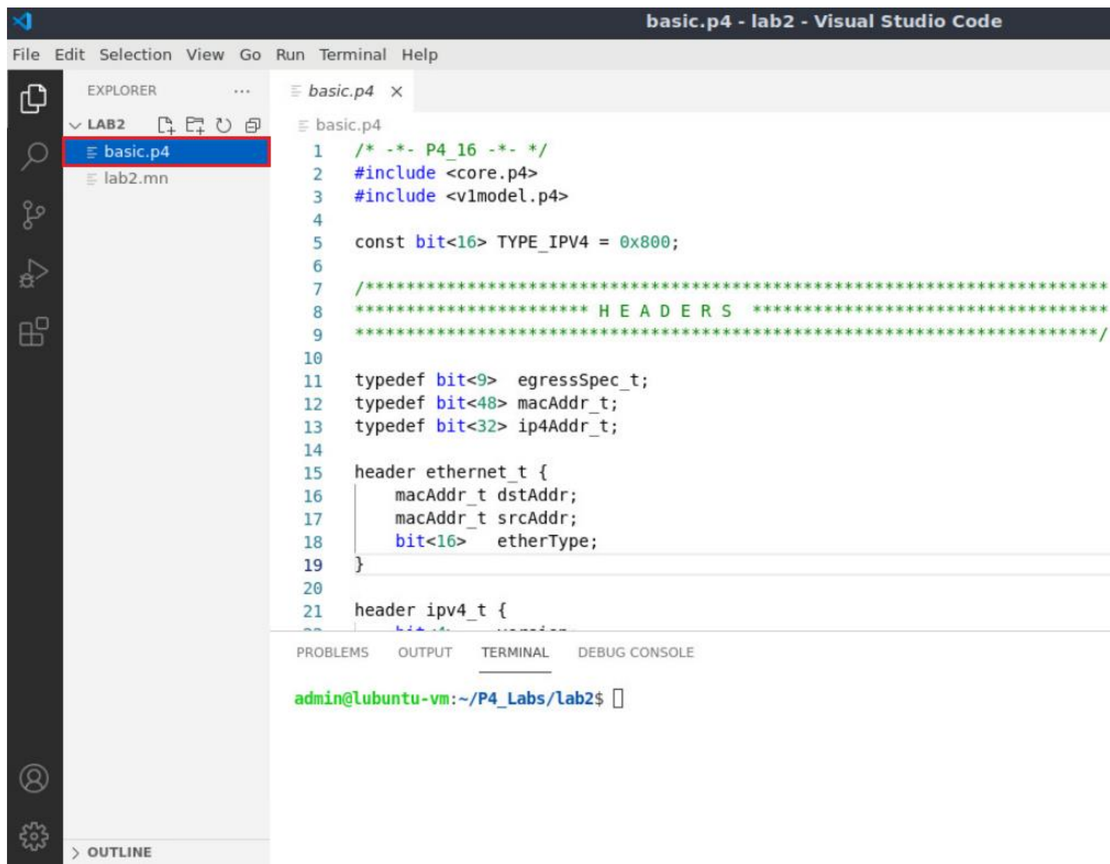


Figure 11. Opening the programming environment in VS Code.

Step 4. Identify the components of VS Code highlighted in the grey boxes.

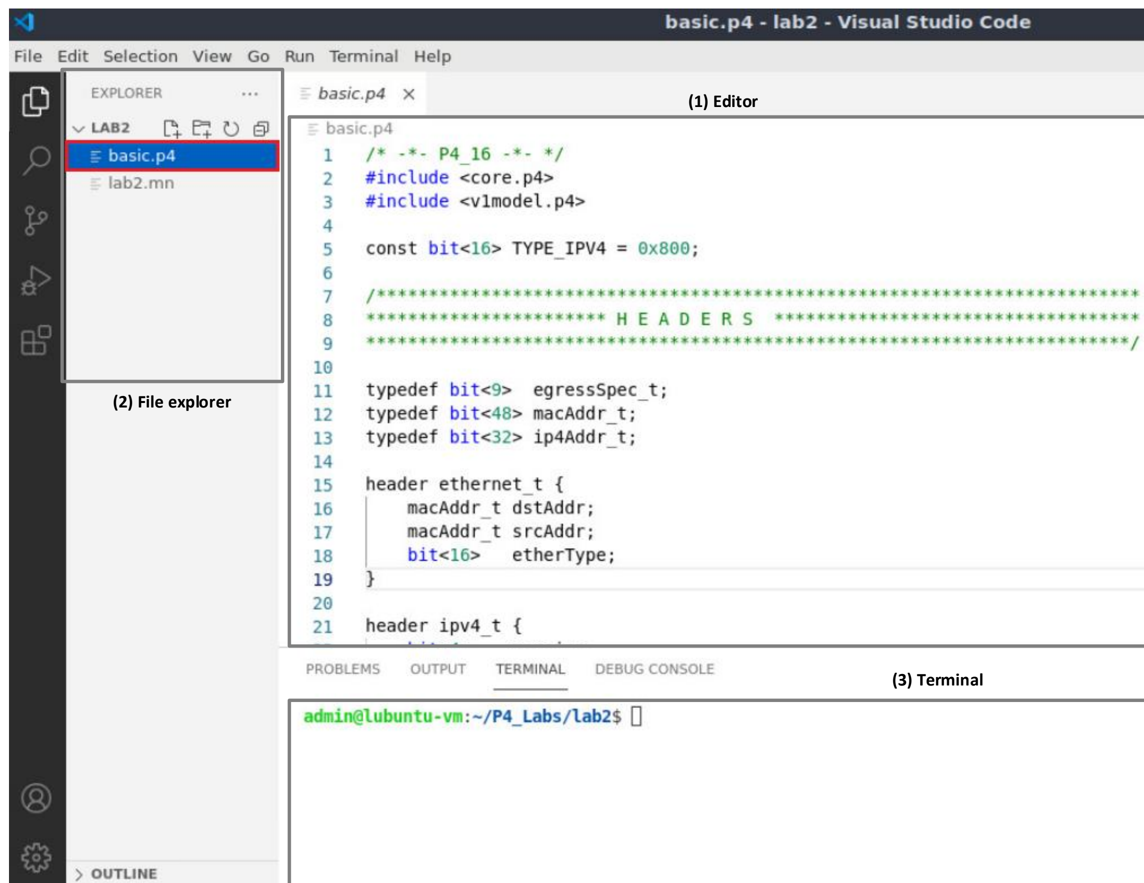


Figure 12. VS Code graphical interface components.

The VS Code interface consists of three main panels:

1. Editor: the editor panel will display the content of the file selected in the file explorer. In the figure above, the *basic.p4* program is shown in the Editor.
2. File explorer: this panel contains all the files in the current directory. You will see the *basic.p4* file which contains the P4 program that will be used in this lab, and the topology file for the current lab (i.e., *lab2.mn*).
3. Terminal: this is a regular Linux terminal integrated in the VS Code. This is where the compiler (*p4c*) is invoked to compile the P4 program and generate the output for the switch.

3.2 Compiling and loading the P4 program to switch s1

Step 1. In this lab, we will not modify the P4 code. Instead, we will just compile it and download it to the switch s1. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```

Figure 13. Compiling the P4 program using the VS Code terminal.

The command above invokes the *p4c* compiler to compile the *basic.p4* program. After executing the command, if there are no messages displayed in the terminal, then the P4 program was compiled successfully. You will see in the file explorer that two files were generated in the current directory:

- *basic.json*: this file is generated by the *p4c* compiler if the compilation is successful. This file will be used by the software switch to describe the behavior of the data plane. You can think of this file as the binary or the executable to run on the switch data plane. The file type here is JSON because we are using the software switch. However, in hardware targets, most probably this file will be a binary file.
- *basic.p4i*: the output from running the preprocessor of the compiler on your P4 program.

At this point, we will only be focusing on the *basic.json* file.

Now that we have compiled our P4 program and generated the JSON file, we can download the program to the switch and start the switch daemon.

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

```
push_to_switch basic.json s1
```

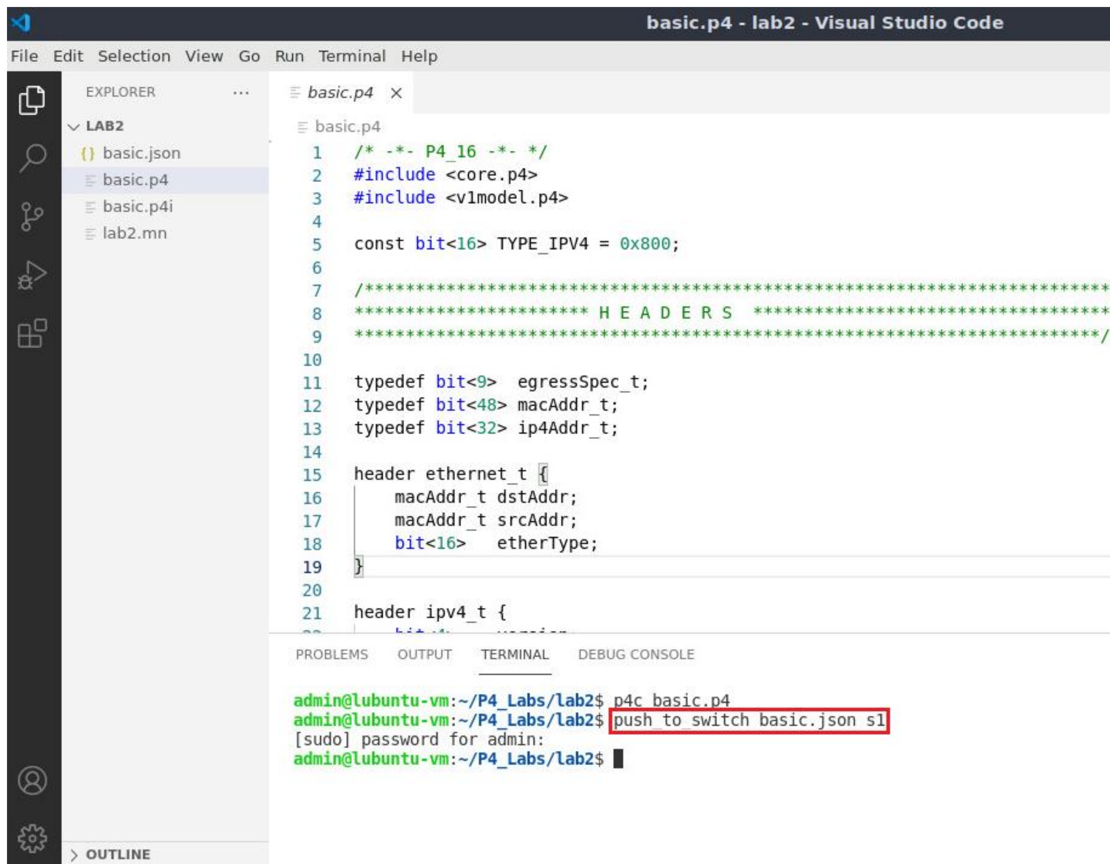


Figure 14. Downloading the compiled program to switch s1.

3.3 Verifying the configuration

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

Figure 15. Maximizing the MiniEdit window.

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

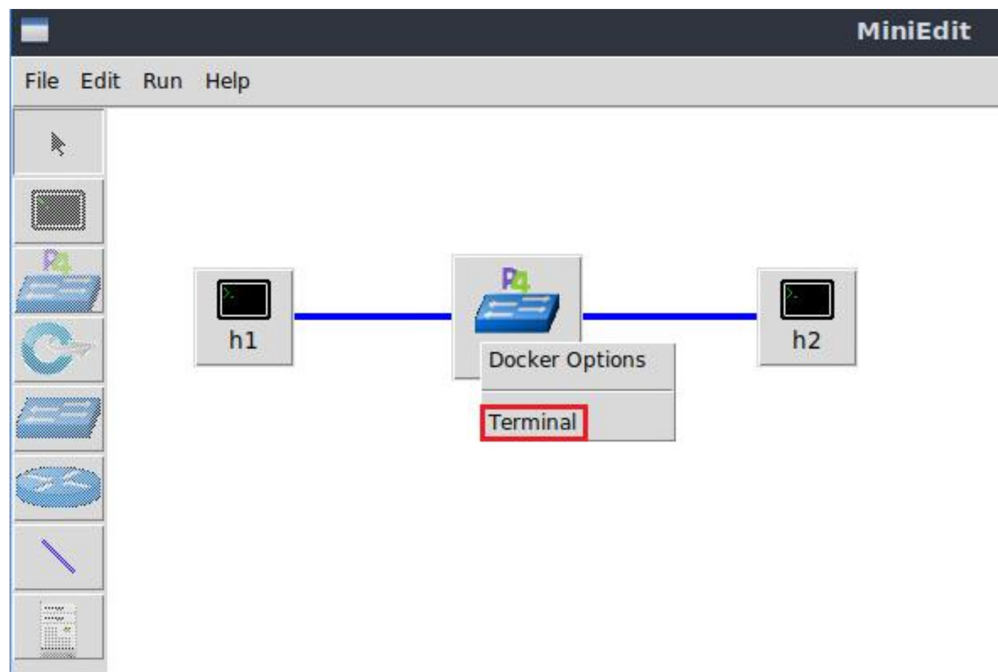


Figure 16. Starting the terminal on switch s1.

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's terminal.

Step 3. Issue the following command to list the files in the current directory.

```
ls
```

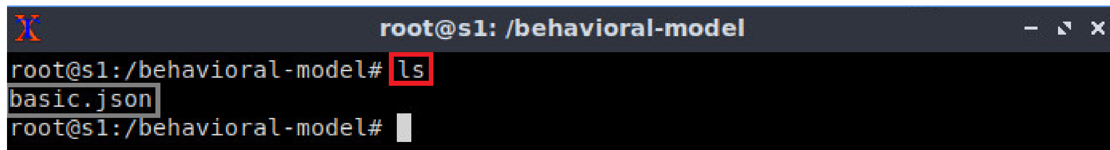


Figure 17. Displaying the contents of the current directory in the switch s1.

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

4 Configuring switch s1

4.1 Mapping P4 program's ports

Step 1. Issue the following command to display the interfaces in 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 18. 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 to the host h1. The interface *s1-eth1* on the switch s1 connects to the host h2.

Step 2. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
```

```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 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
```

Figure 19. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

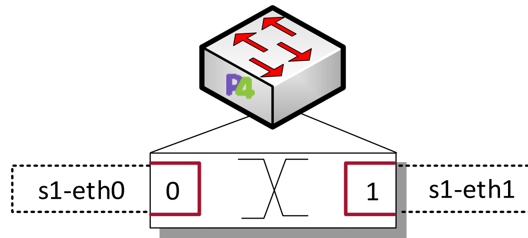


Figure 20. Ports 0 and 1 are mapped to the interfaces *s1-eth0* and *s1-eth1* of switch s1.

4.2 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 21. Returning to switch s1 CLI.

Step 2. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab2/rules.cmd
```

```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab2/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 22. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

Step 3. Go back to host h1 terminal to test the connectivity between host h1 and host h2 by issuing the following command.

```
ping 10.0.0.2 -c 4
```

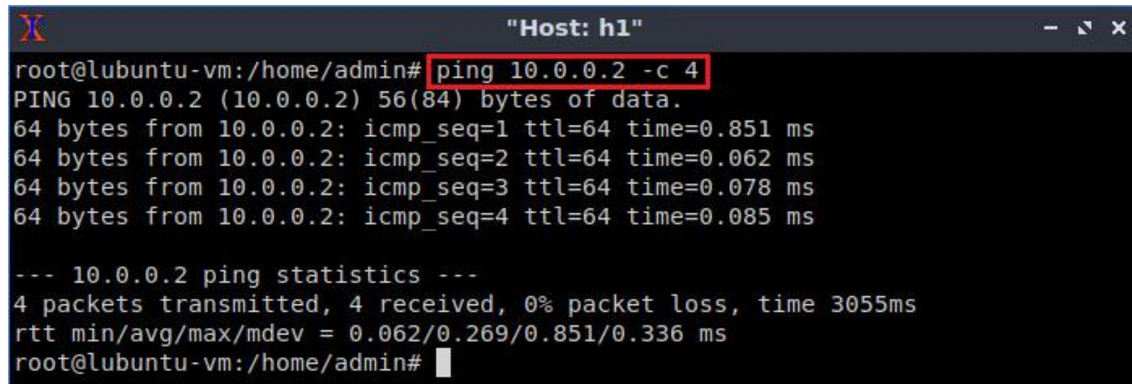


Figure 23. Performing a connectivity test between host h1 and host h2.

Now that the switch has a program with tables properly populated, the hosts can ping each other.

This concludes lab 2. Stop the emulation and then exit out of MiniEdit.

References

1. B. Trammell, M. Kuehlewind. "*RFC 7663: Report from the IAB workshop on stack evolution in a middlebox internet (SEMI).*" 2015. [Online]. Available: https://tools.ietf.org/html/rfc7663.
2. G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, S. Mangiante. ''*De-ossifying the internet transport layer: A survey and future perspectives*,'' IEEE Communications. Surveys and Tutorials., 2017.
3. The Register. "*VMware, Cisco stretch virtual LANs across the heavens*." 2011. [Online]. Available: https://tinyurl.com/y6mxhqzn.
4. M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "*Virtual eXtensible Local Area Network (VXLAN): a framework for overlaying virtualized layer 2 networks over layer 3 networks*," RFC7348. [Online]. Available: http://www. rfc-editor.org/rfc/rfc7348.txt
5. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, ''*P4: Programming protocol-independent packet processors,*'' ACM SIGCOMM Computer Communications. 2014.
6. P4lang. "*Behavioral model*". [Online]. Available: https://github.com/p4lang/behavioral-model.
7. V. Gurevich, A. Fingerhut, "*$P4_{16}$ for Intel Tofino™ using Intel P4 Studio™*". 2021 P4 Workshop, ONF. [Online]. Available: https://tinyurl.com/yckzkybf.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Exercise 2: Compiling and Running a P4 Program

Document Version: 01-14-2022

Award 2118311
"Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks"

Contents

1 Exercise description 3
1.1 Credentials 3
1.2 Exercise topology 3
2 Setting the environment 3
3 Deliverables 5

1 Exercise description

In this exercise, you will compile and run a P4 program on two P4 switches in the same topology. Then, you will push the table entries to the switches at runtime.

1.1 Credentials

The information in Table 1 provides the credentials to access the Client's virtual machine.

Table 1. Credentials to access the Client's virtual machine.

Device	Account	Password
Client	admin	password

1.2 Exercise topology

The topology comprises two P4 switches and two end hosts.

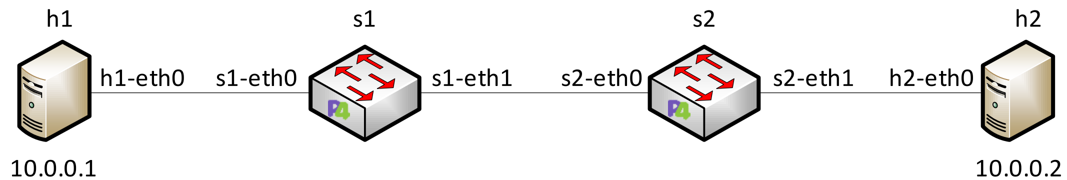


Figure 1. Exercise topology.

2 Setting the environment

Follow the steps below to set the exercise's environment.

Step 1. Open MiniEdit by double-clicking the shortcut on the desktop. If a password is required type `password`.

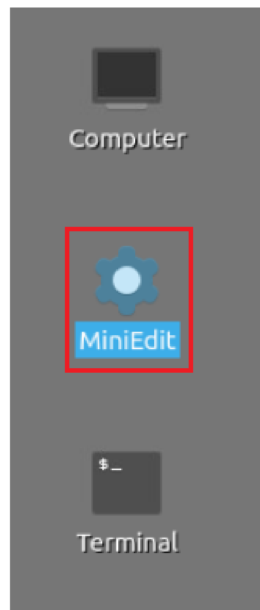


Figure 2. MiniEdit shortcut.

Step 2. Load the topology located at */home/admin/P4_Exercises/Exercise2/*.

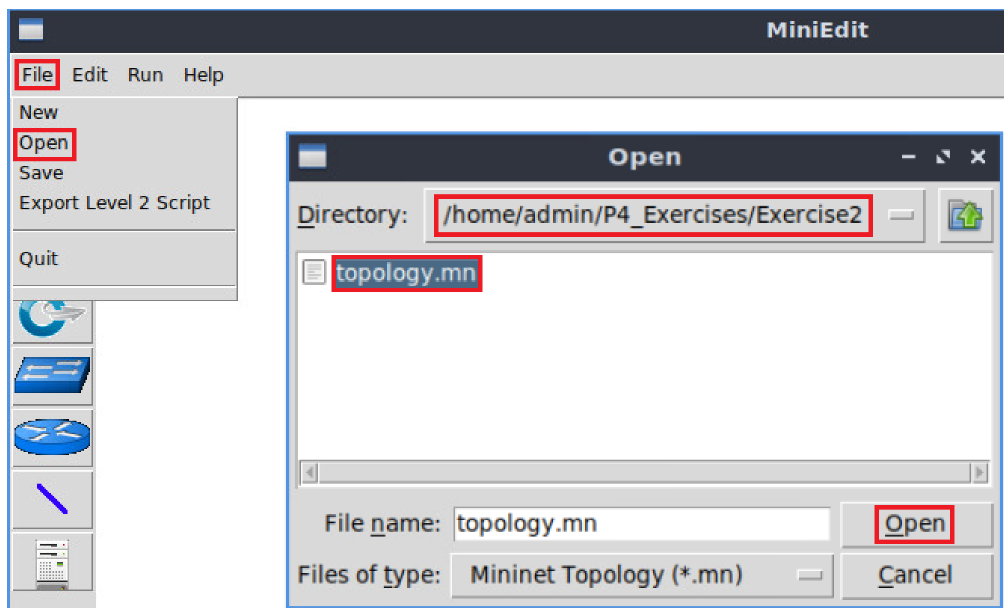


Figure 3. Opening the exercise topology.

Step 3. Run the emulation by clicking on the button located on the lower left-hand side.

Figure 4. Running the emulation.

Step 4. In the Linux terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this exercise is located.

```
code P4_Exercises/Exercise2/
```

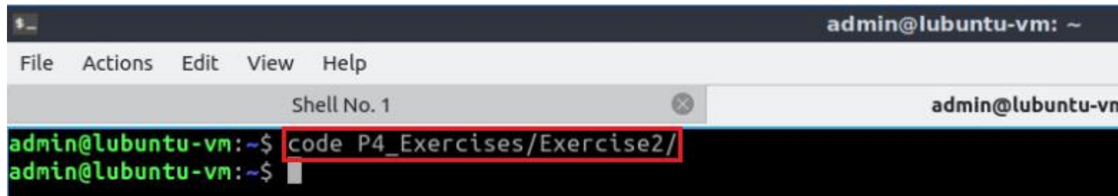


Figure 5. Opening the working directory.

3 Deliverables

Follow the steps below to complete the exercise.

a) Compile the *basic.p4* in the VS Code. Which files were generated?

b) Push the output file of the compiler to both switches s1 and s2.

c) Start the daemon on both switches and map the ports to the corresponding interfaces (see Figure 6). Will there be connectivity between the hosts at this point?

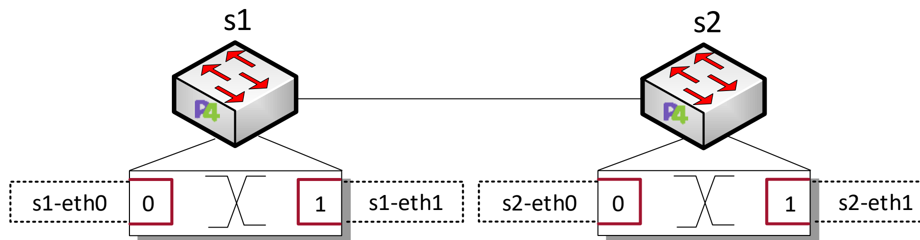


Figure 6. Port mapping.

d) Push the table entries to the switches. The files *rules_s1.cmd* and *rules_s2.cmd* for switches s1 and s2, respectively, are located in ~/exercise2/.

e) Run a connectivity test between the hosts using ping. Is there connectivity?

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 3: P4 Program Building Blocks

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 The PISA architecture 3
1.1 The PISA architecture 4
1.2 Programmable parser 4
1.3 Programmable match-action pipeline 5
1.4 Programmable deparser 5
1.5 The V1Model 5
1.6 P4 program mapping to the V1Model 6
2 Lab topology 6
2.1 Starting host h1 and host h2 8
3 Navigating through the components of a basic P4 program 8
3.1 Loading the programming environment 9
3.2 Describing the components of the P4 program 9
3.3 Programming the pipeline sequence 14
4 Loading the P4 program 15
4.1 Compiling and loading the P4 program to switch s1 15
4.2 Verifying the configuration 17
5 Configuring switch s1 18
5.1 Mapping the P4 program's ports 18
5.2 Loading the rules to the switch 20
6 Testing and verifying the P4 program 21
References 23

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).

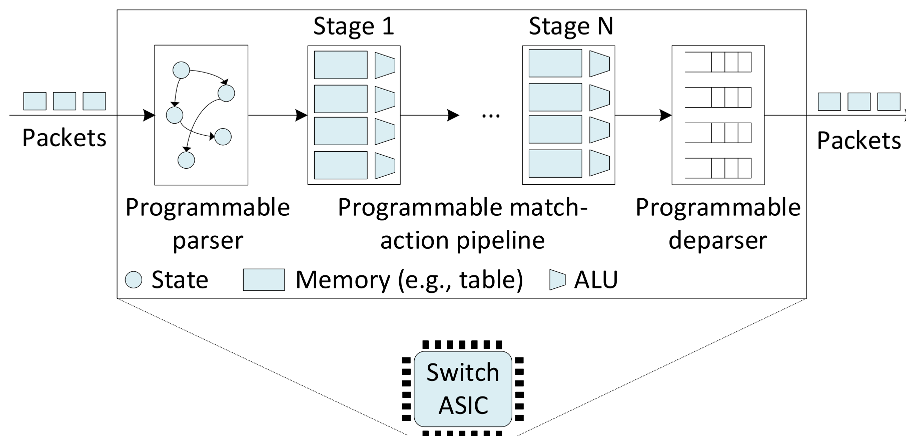


Figure 1. A PISA-based data plane.

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 V1Model[2] 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 target[3].

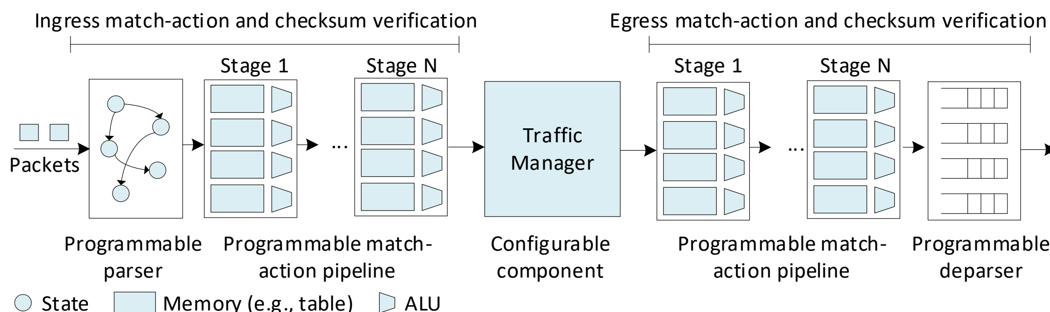


Figure 2. The V1Model architecture.

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.

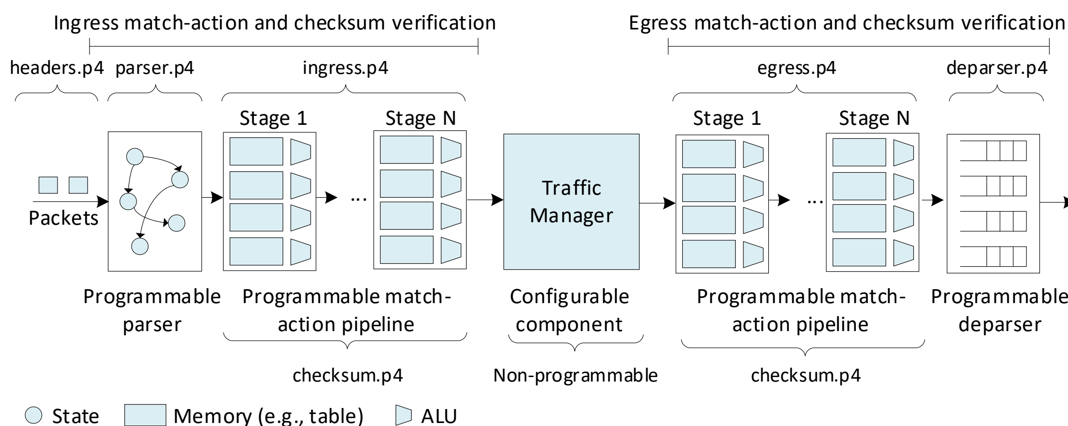


Figure 3. Mapping of P4 files to the V1Model's components.

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.

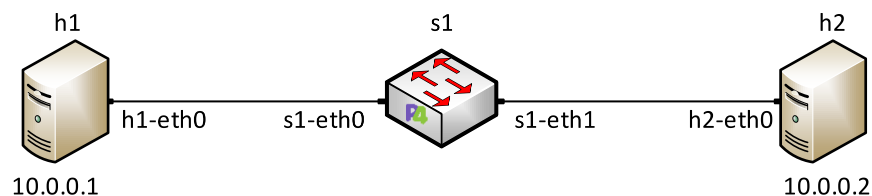


Figure 4. Lab topology.

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.

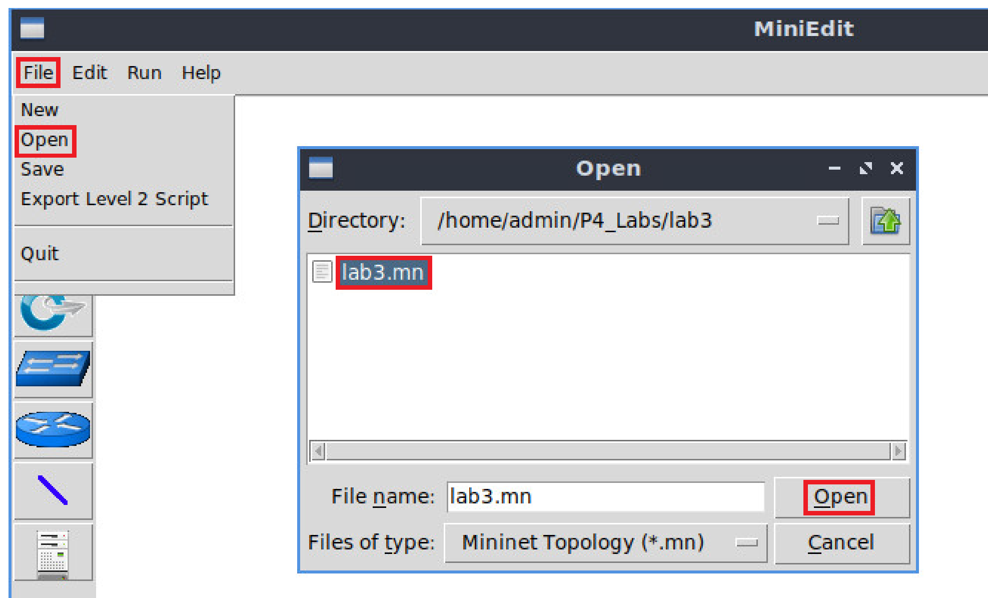


Figure 6. Opening a topology in MiniEdit.

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.

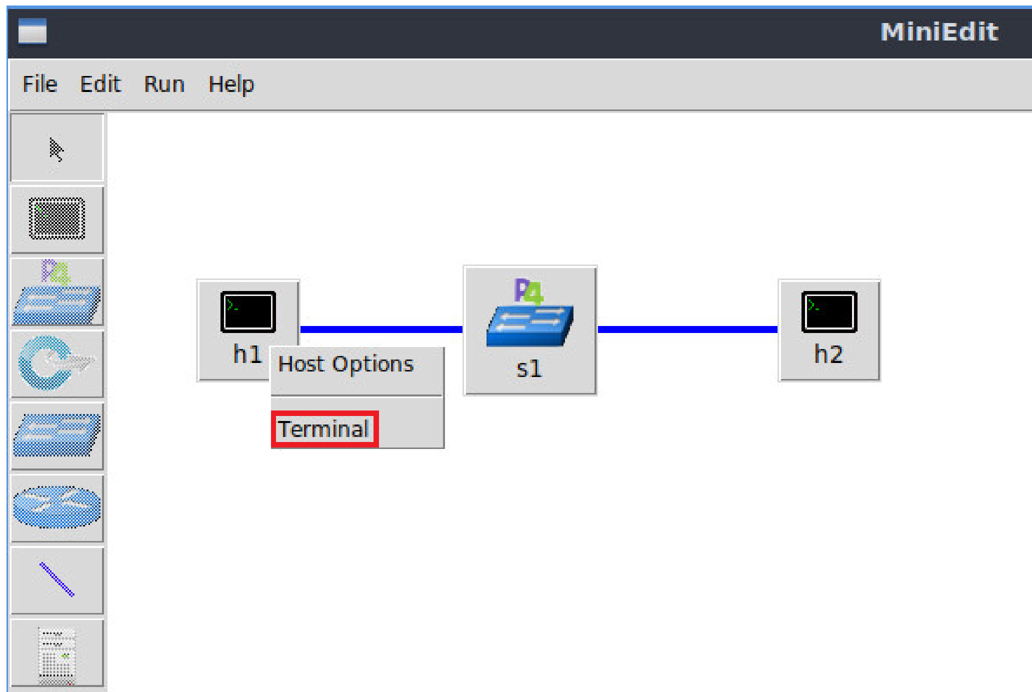


Figure 8. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

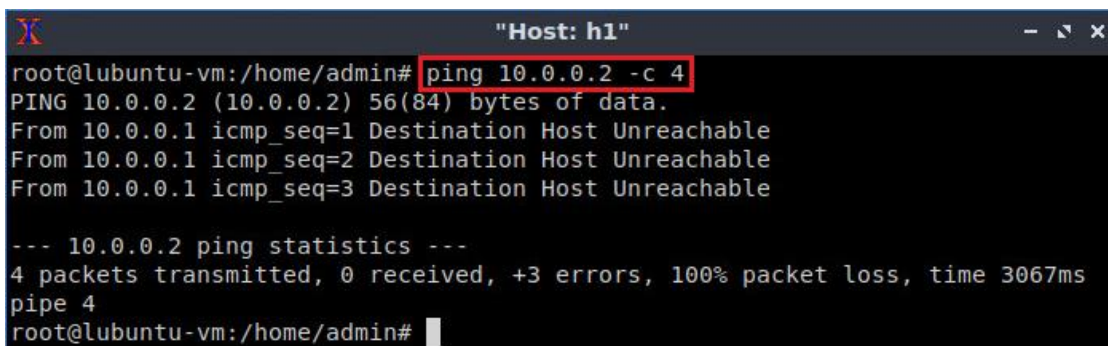


Figure 9. Performing a connectivity test between host h1 and host h2.

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.

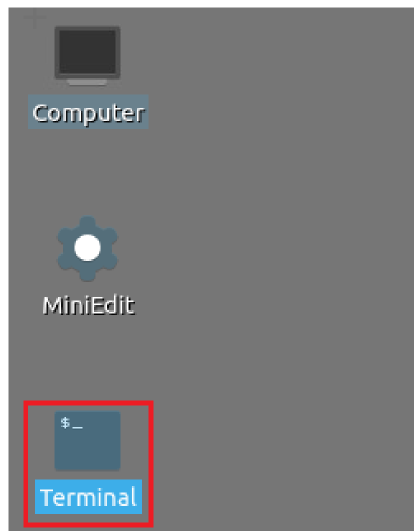


Figure 10. Shortcut to open a Linux terminal.

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.

```
code ~/P4_Labs/lab3/
```

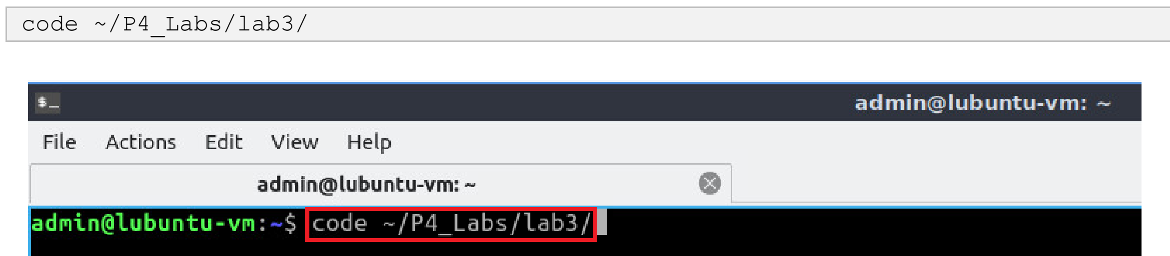


Figure 11. Launching the editor and opening the lab3 directory.

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.

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/
```

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.

headers.p4 - lab3 - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER ...

LAB3
- basic.p4
- checksum.p4
- deparser.p4
- egress.p4
- headers.p4
- ingress.p4
- lab3.mn
- parser.p4

basic.p4 | headers.p4 ×

headers.p4

```
1  const bit<16> TYPE_IPV4 = 0x800;
2
3  /*************************************************************************
4  *********************** H E A D E R S  ***********************************
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;
   }
```

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.

```
parser.p4 - lab3 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB3: basic.p4, basic.p4i, checksum.p4, deparser.p4, egress.p4, headers.p4, ingress.p4, lab3.mn, parser.p4

basic.p4   parser.p4
parser.p4
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     }
```

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.

```
ingress.p4 - lab3 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB3: basic.p4, basic.p4i, checksum.p4, deparser.p4, egress.p4, headers.p4, ingress.p4, lab3.mn, parser.p4

basic.p4   ingress.p4
ingress.p4
3  /*************************************************************************
4  ************** I N G R E S S   P R O C E S S I N G   *******************
5  *************************************************************************/
6
7  control MyIngress(inout headers hdr,
8                    inout metadata meta,
9                    inout standard_metadata_t standard_metadata) {
10     action drop() {
11         mark_to_drop(standard_metadata);
12     }
13
14     action forward(egressSpec_t port) {
15         standard_metadata.egress_spec = port;
16     }
17
18     table forwarding {
19         key = {
20             standard_metadata.ingress_port:exact;
21         }
22         actions = {
23             forward;
24             drop;
25             NoAction;
26         }
27         size = 1024;
28         default_action = drop();
29     }
30
31     apply {
32         forwarding.apply();
33     }
34 }
```

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.

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

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}
```

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.

```
/*************************************************************************
************   C H E C K S U M    V E R I F I C A T I O N   *************
*************************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {  }
}


/*************************************************************************
*************   C H E C K S U M    C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
     apply {
    update_checksum(
        hdr.ipv4.isValid(),
            { hdr.ipv4.version,
          hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
```

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;
```

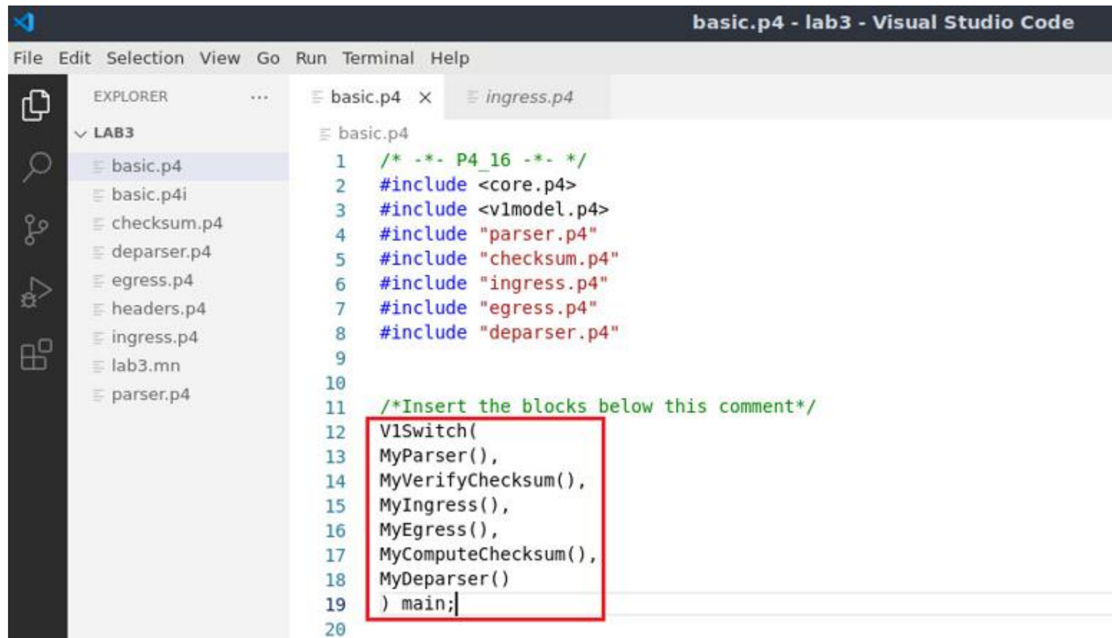


Figure 20. Writing the pipeline sequence in the *basic.p4* program

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
```

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
```

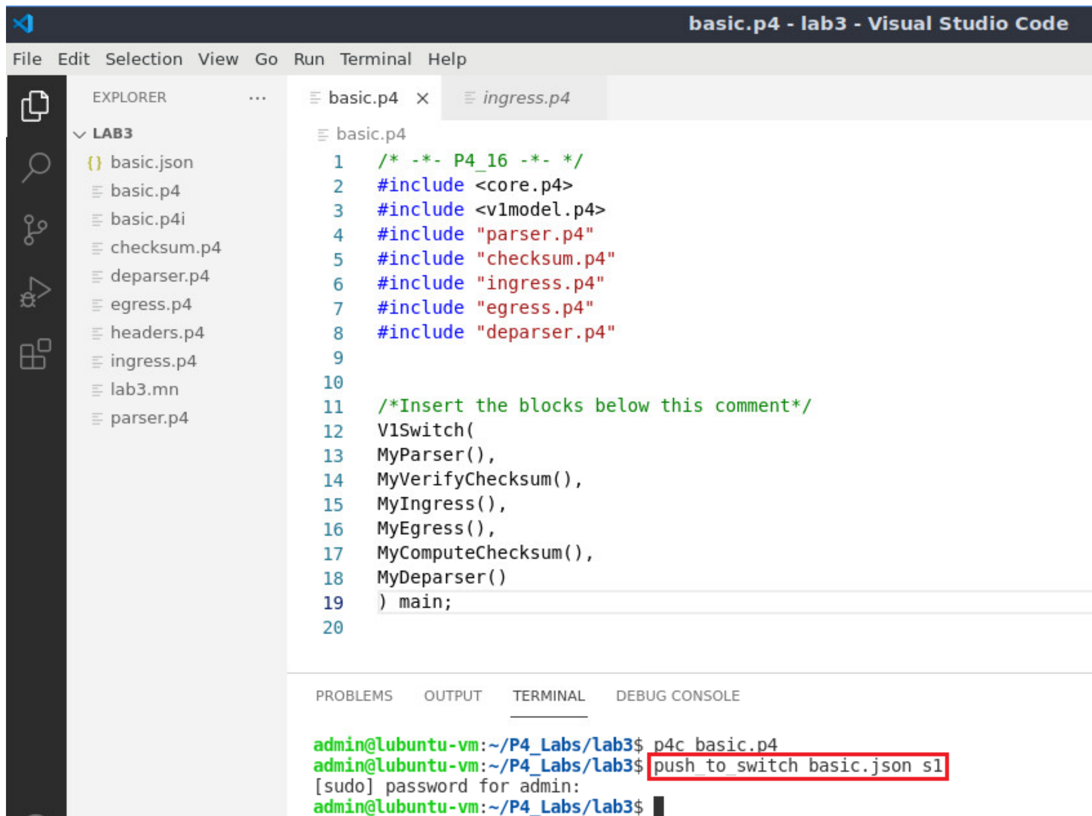


Figure 22. Downloading the P4 program to switch s1.

4.2 Verifying the configuration

Step 1. Click on the MinEdit 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*.

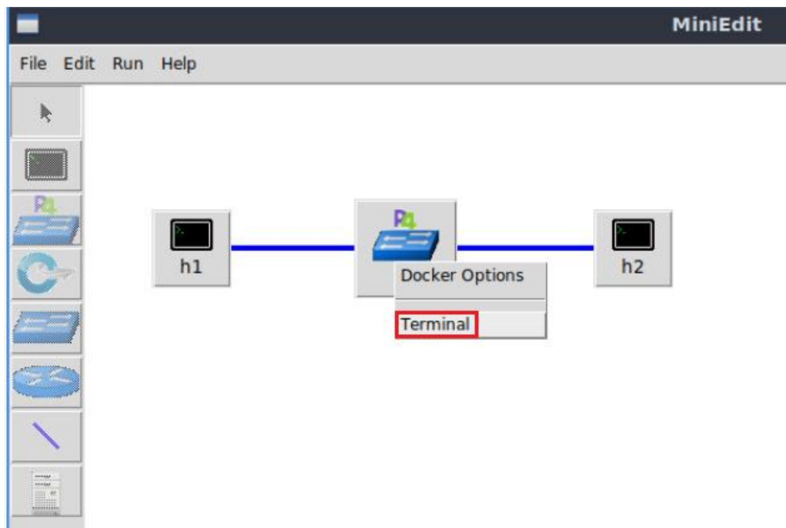


Figure 24. Starting the terminal on the switch.

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
```

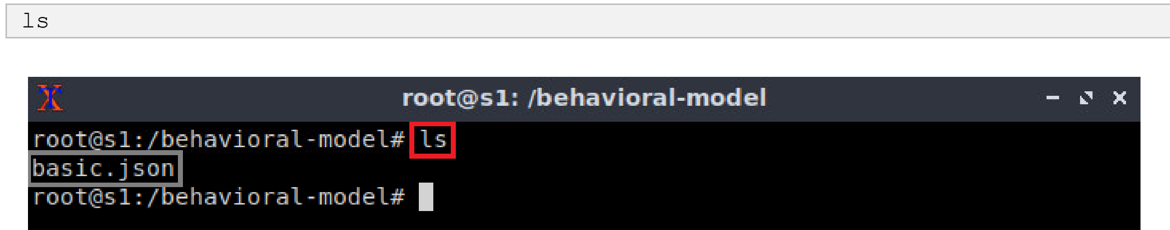


Figure 25. Displaying the contents of the current directory in the switch s1.

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
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
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.

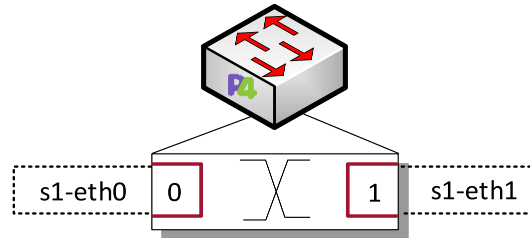


Figure 28. Mapping of the logical interface numbers (0, 1) to the Linux interfaces (*s1-eth0*, *s1-eth1*).

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
```

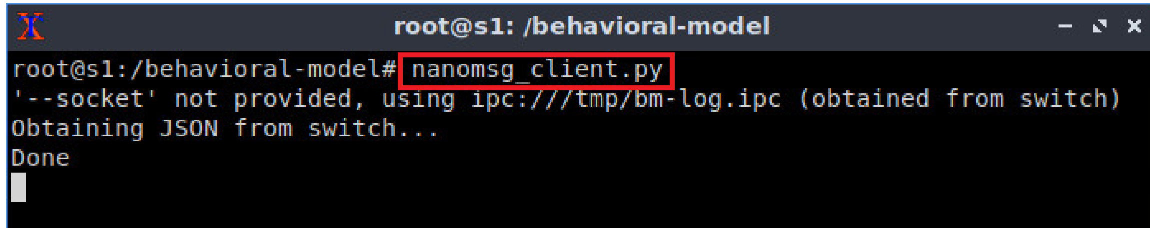


Figure 31. Displaying switch s1 logs.

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

```
./recv.py
```

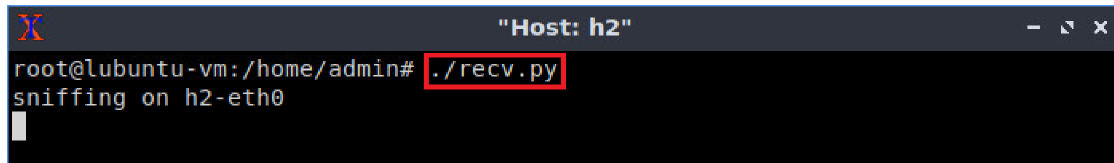


Figure 32. Listening for incoming packets in host h2.

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
```

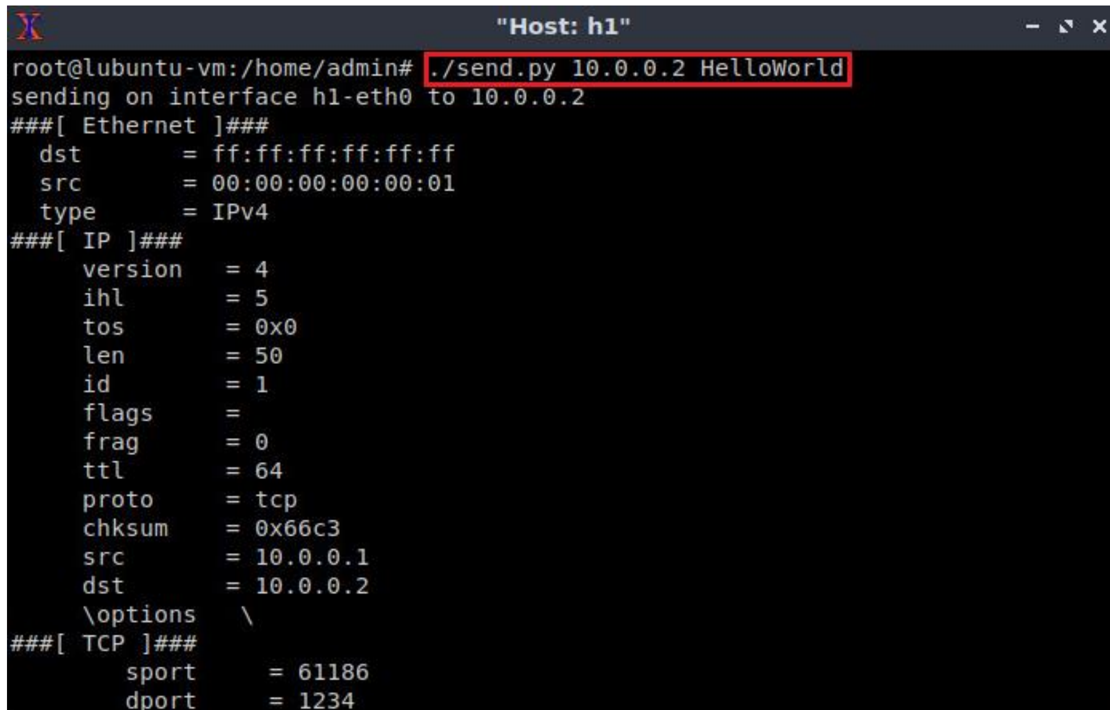


Figure 33. Sending a test packet from host h1 to host h2.

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.

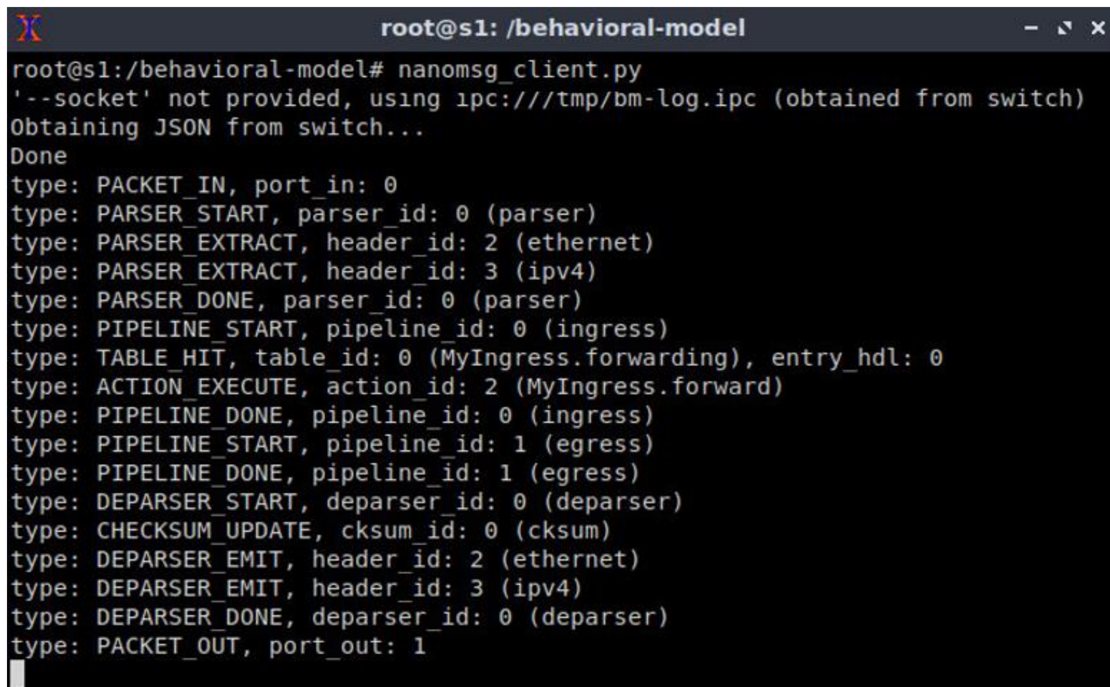


Figure 34. Inspecting the logs in switch s1.

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.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 4: Parser Implementation

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 3
1.1 Program headers and definitions 4
1.2 Programmable parser 6
2 Lab topology 7
2.1 Starting host h1 and host h2 9
3 Defining the program's headers 10
3.1 Loading the programming environment 10
3.2 Coding header's definitions into the *headers.p4* file 11
4 Parser Implementation 14
5 Loading the P4 program 17
5.1 Compiling and loading the P4 program to switch s1 17
5.2 Verifying the configuration 19
6 Configuring switch s1 20
6.1 Mapping P4 program's ports 20
6.2 Loading the rules to the switch 22
7 Testing and verifying the P4 program 22
8 Augmenting the P4 program to parse IPv6 24
9 Testing and verifying the augmented P4 program 28
References 31

Overview

This lab starts by describing how to define custom headers in a P4 program. It then explains how to implement a simple parser that parses the defined headers. The lab further shows how to track the parsing states of a packet inside the software switch.

Objectives

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

1. Define custom headers in a P4 program.
2. Understand how the parser transitions between states and how it extracts the headers from the packets.
3. Implement a simple parser in P4.
4. Trace the parsed states 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.
2. Section 2: Lab topology.
3. Section 3: Defining the headers.
4. Section 4: Parser implementation.
5. Section 5: Loading the P4 program.
6. Section 6: Configuring switch s1.
7. Section 7: Testing and verifying the P4 program.
8. Section 8: Augmenting the P4 program to parse IPv6.
9. Section 9: Testing and verifying the augmented P4 program.

1 Introduction

1.1 Program headers and definitions

For several decades, the networking industry operated in a bottom-up approach. At the bottom of the system are the fixed-function Application Specific Integrated Circuits (ASICs), which enforce protocols, features, and processes available in the switch. Programmers and operators are limited to these capabilities when building their systems. Consequently, systems have features defined by ASIC vendors that are rigid and may not fit the network operators' needs. Programmable switches and P4 represent a disruption of the networking industry by enabling a top-down approach for the design of network applications. With this approach, the programmer or network operator can precisely describe features and how packets are processed in the ASIC, using a high-level language, P4.

With the Protocol Independent Switch Architecture (PISA)[1], the programmer defines the headers and corresponding parser as well as actions executed in the match-action pipeline and the deparser. The programmer has the flexibility of defining custom headers (i.e., a header not standardized). Such capability is not available in non-programmable devices.

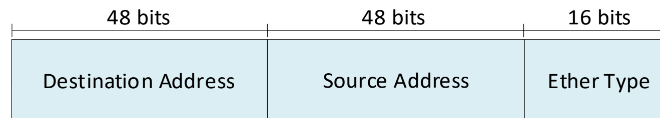


Figure 1. Ethernet header.

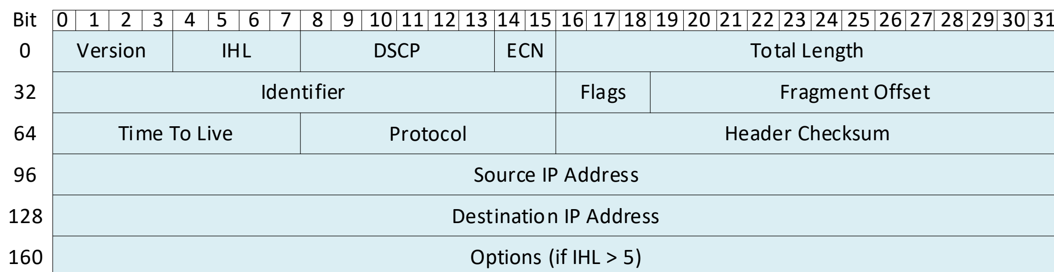


Figure 2. IPv4 header.

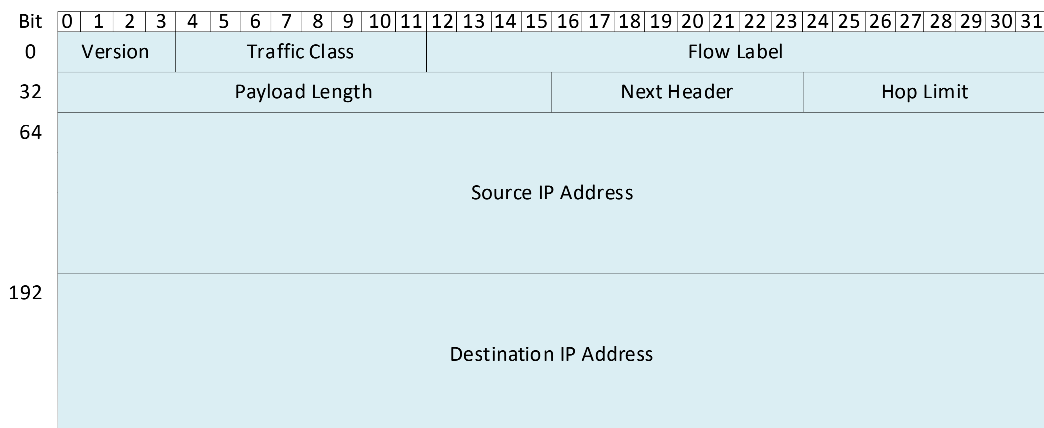


Figure 3. IPv6 header.

Figure 4 shows an excerpt of a P4 program where the headers are defined. This is typically written at the top of the program before the parsing starts. We can see that the programmer defined a header corresponding to Ethernet (lines 11-15). The Ethernet header fields are shown in Figure 1.

The programmer also defined an IPv4 header (lines 26-40). The IPv4 header format is shown in Figure 2 and the IPv6 header is shown in Figure 3.

```
 1:  #include <core.p4>
 2:  #include <v1model.p4>
 3:  const bit<16> TYPE_IPV4 = 0x800;
 4:
 5:  /***********************HEADERS***********************/
 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:  struct metadata {
18:     /* empty */
19:  }
20:
21:  struct headers{
22:     ethernet_t ethernet;
23:     ipv4_t ipv4;
24:  }
25:
26:  header ipv4_t {
27:     bit<4> version;
28:     bit<4> ihl;
29:     bit<6> DSCP;
30:     bit<2> ECN;
31:     bit<16> totalLen;
32:     bit<16> identification;
33:     bit<3> flags;
34:     bit<13> fragOffset;
35:     bit<8> ttl;
36:     bit<8> protocol;
37:     bit<16> hdrChecksum;
38:     ip4Addr_t srcAddr;
39:     ip4Addr_t dstAddr;
40:  }
```

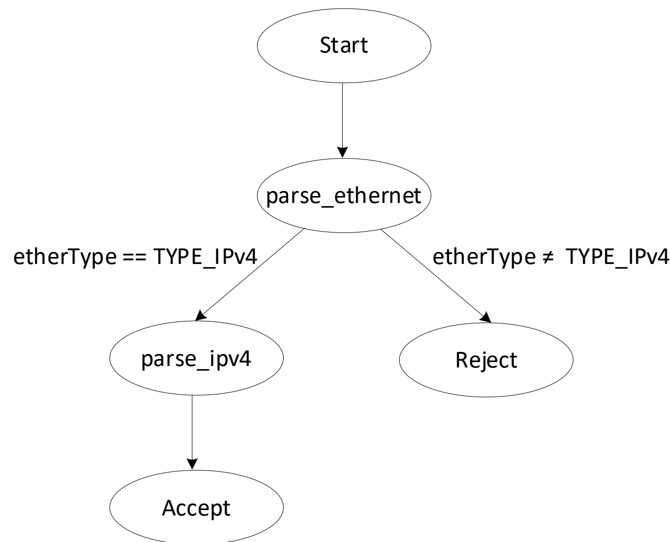
Figure 4. Program headers and definitions.

The code starts by including the *core.p4* file (line 1) which defines some common types and variables used in all P4 programs. For instance, the `packet_in` and `packet_out` extern types which represent incoming and outgoing packets, respectively, are declared in *core.p4*[2]. Next, the *v1model.p4*[3] file is included (line 2) to define the V1Model architecture[4] and all its externs used when writing P4 programs. Line 3 creates a 16-bit

constant `TYPE_IPV4` with the value 0x800. This means that `TYPE_IPV4` can be used later in the P4 program to reference the value 0x800. The typedef declarations (lines 7 - 9) are used to assign alternative names to types. Subsequently, the headers and the metadata structs that will be used in the program are defined. These headers are customized depending on how the programmer wants the packets to be parsed. The program in Figure 1 defines the Ethernet header (lines 11-15) and the IPv4 header (lines 26-40). The declarations inside each header are usually written after referring to the standard specifications of the protocol. Note in the `ethernet_t` header the `macAddr_t` is used rather than using a 48-bit field. Lines 17 - 19 show how to declare user-defined metadata, which are passed from one block to another as the packet propagates through the architecture. For simplicity, this program does not require any user metadata.

1.2 Programmable parser

The programmable parser permits the programmer to describe how the switch will process the packet. The parser de-encapsulates the headers, converting the original packet into a parsed representation of 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).



(a)

```
1:  /************************HEADERS************************/
2:  parser MyParser( packet_in packet, out headers hdr,
3:                   inout metadata meta,
4:                   inout standard_metadata_t standard_metadata ){
5:     state start {
6:         transition parse_ethernet;
7:     }
8:     state parse_ethernet {
9:         packet.extract(hdr.ethernet);
10:        transition select(hdr.ethernet.etherType) {
11:            TYPE_IPV4: parse_ipv4;
12:            default: reject;
13:        }
14:    }
15:    state parse_ipv4 {
16:        packet.extract(hdr.ipv4);
17:        transition accept;
18:    }
19: }
```

(b)

Figure 5. Example of a parser. (a) Graphical representation of the parser. (b) In P4, the parser always starts with the initial state called `start`. First, we transition unconditionally to `parse_ethernet`. Then, we can create some conditions to direct the parser. Finally, when we transition to the `accept` state, the packet is moved to the ingress block of the pipeline. A packet that reaches the `reject` state will be dropped.

Figure 5a shows the graphical representation of the parser and Figure 5b its corresponding P4 code. Note that packet is an instance of the `packet_in` extern (specific to V1Model) and is passed as a parameter to the parser. The `extract` method associated with the packet extracts N bits, where N is the total number of bits defined in the corresponding header (for example, 112 bits for Ethernet). Afterwards, the `etherType` field of the Ethernet header is examined using the select statement, and the program branches to the `parse_ipv4` state if the `etherType` field corresponds to IPv4. The state transitions to the `reject` if it is not an IPv4 header, as shown in the figure above (Line 12). In the `parse_ipv4` state, the IPv4 header is extracted, and the program unconditionally transitions to the `accept` state.

2 Lab topology

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

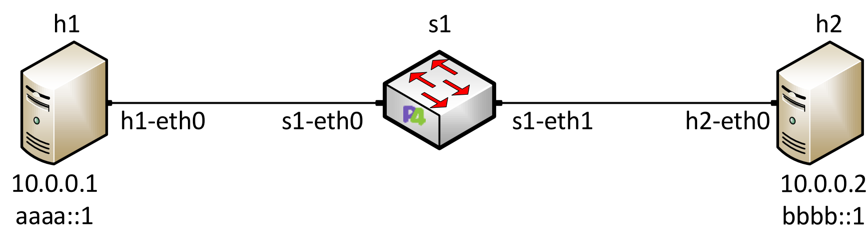


Figure 6. Lab topology.

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 7. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab4* folder and search for the topology file called *lab4.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

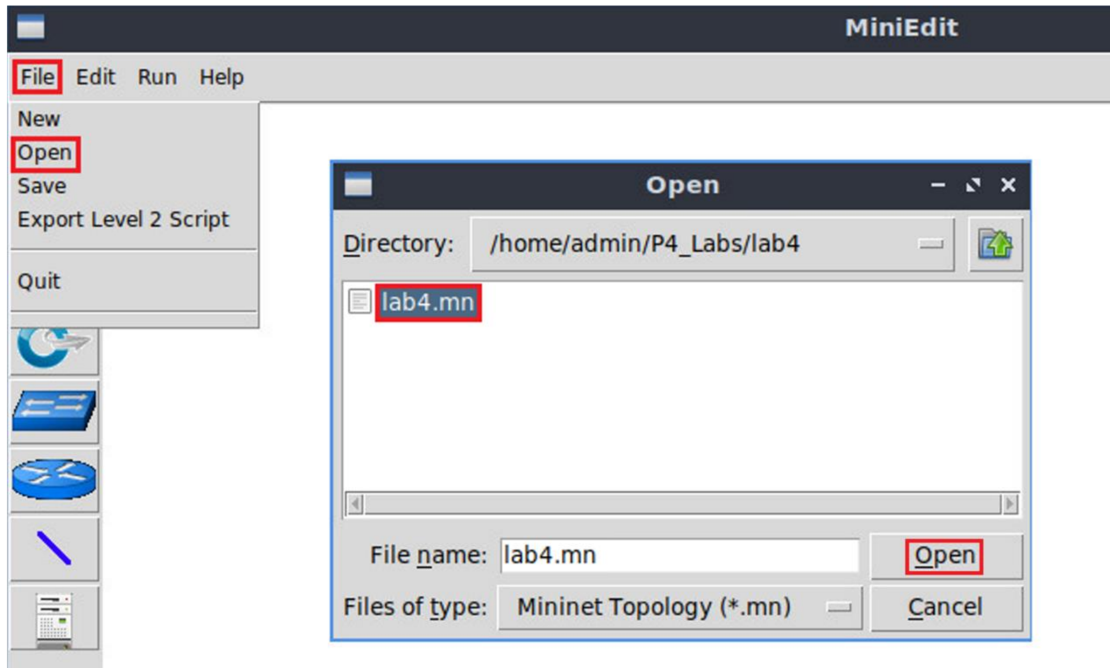


Figure 8. MiniEdit's *Open* dialog.

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 9. 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.

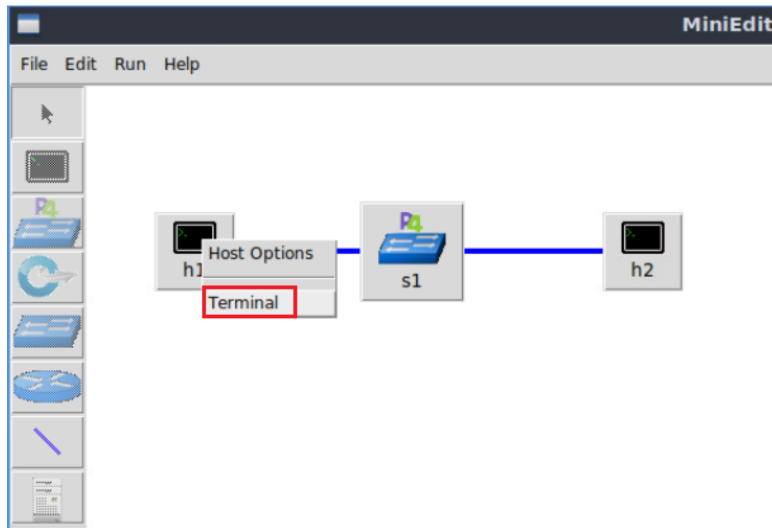


Figure 10. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

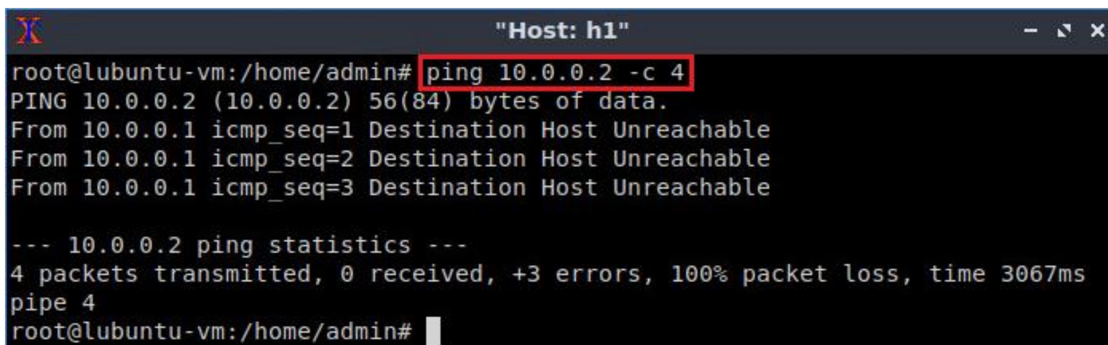


Figure 11. Connectivity test using `ping` command.

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

3 Defining the program's headers

This section demonstrates how to define custom headers in a P4 program. It also shows how to use constants and typedefs to make the program more readable.

3.1 Loading the programming environment

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

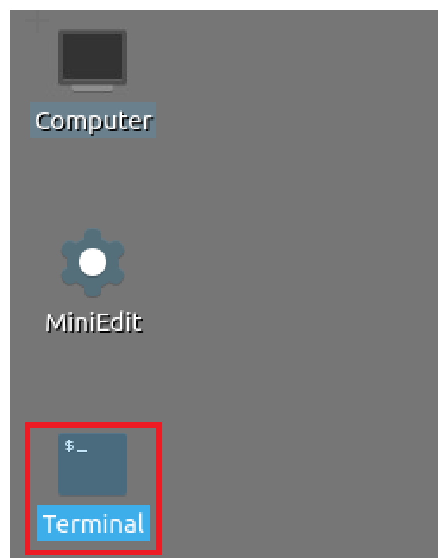


Figure 12. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). A CLI is a program that takes commands from the keyboard and sends them to the operating system to perform.

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/lab4
```

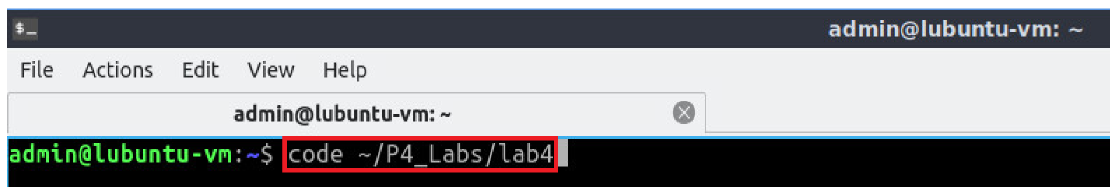


Figure 13. Launching the editor and opening the lab4 directory.

3.2 Coding header's definitions into the *headers.p4* file

Step 1. Click on the *headers.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.

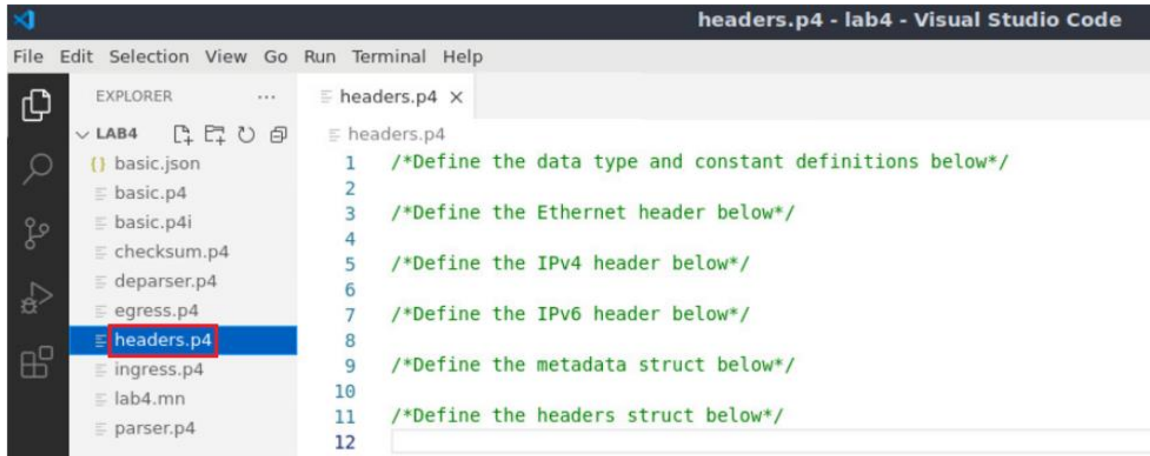


Figure 14. Inspecting the *headers.p4* file.

We can see that the *headers.p4* is empty and we have to fill it.

Step 2. We will start by defining some typedefs and constants. Write the following in the *headers.p4* file.

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;
```

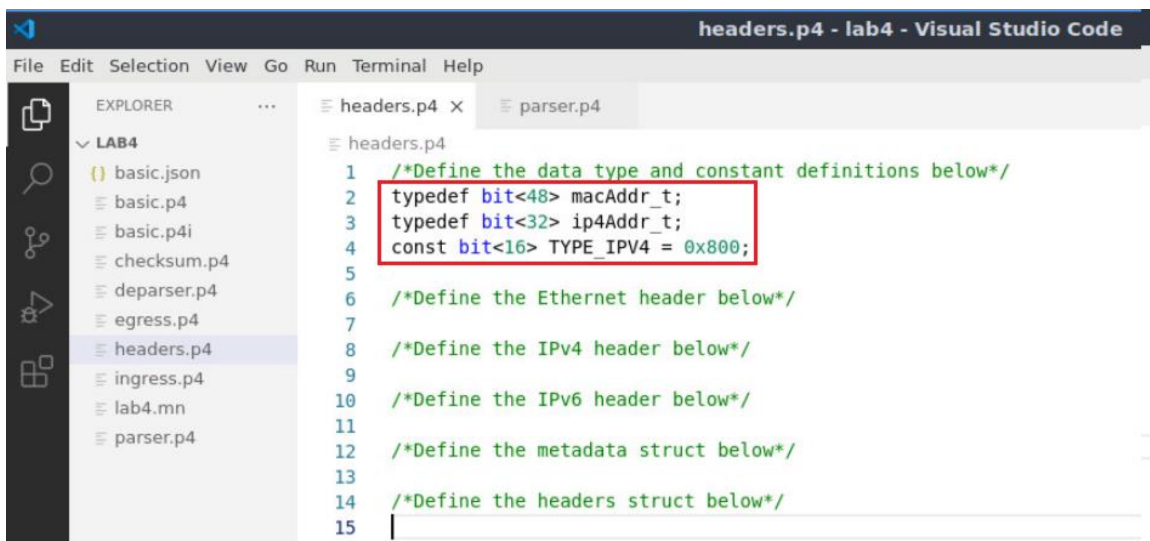


Figure 15. Data types and constant definitions.

In the figure above the typedef declarations used (lines 2 - 3) are used to assign alternative names to types. Here we are saying that `macAddr_t` can be used instead of `bit<48>`, and `ip4Addr_t` instead of `bit<32>`. We will use those typedefs when defining the headers.

Line 4 shows how to define a constant with the name `TYPE_IPV4` and a value of `0x800`. We will use this value in the parser implementation.

Step 3. Now we will define the Ethernet header. Add the following code to *the headers.p4* file.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}
```

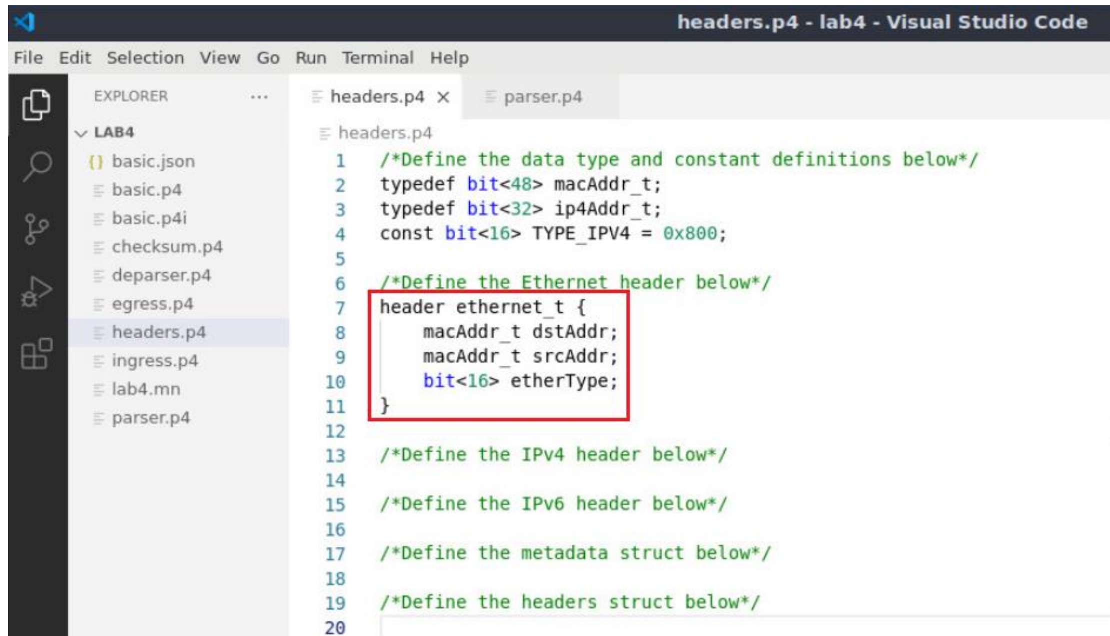


Figure 16. Adding the Ethernet header definition.

Note how we used the typedef `macAddr_t` which corresponds to `bit<48>` when defining the destination MAC address field (`dstAddr`) and the source MAC address field (`srcAddr`).

Step 4. Now we will define the IPv4 header. Add the following to the *headers.p4* file.

```
header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

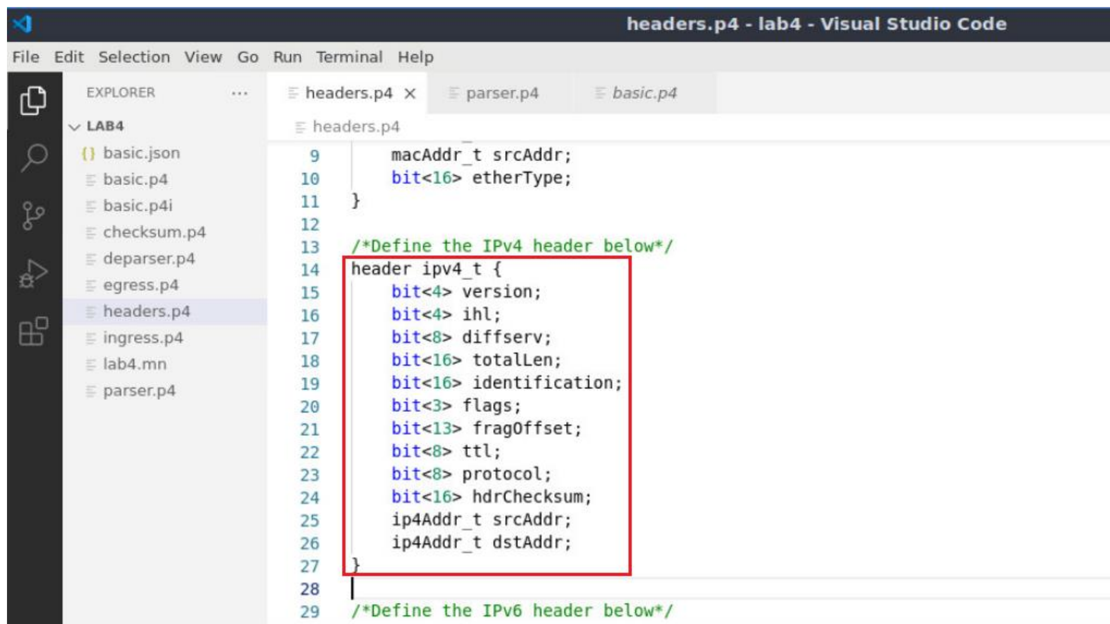


Figure 17. Adding the IPv4 header definition.

Consider the figure above. Note how we used the typedef `ip4Addr_t` which corresponds to `bit<32>` when defining the source IP address field (`srcAddr`) and the destination IP address field (`dstAddr`). Also, note how we are mapping the fields to those defined in the standard IPv4 header (see Figure 3).

Step 5. Now we will create a struct to represent our metadata. Metadata are passed from one block to another as the packet propagates through the architecture. For simplicity, this program does not require any user metadata, and hence we will define it as empty with no fields. Add the following to the *headers.p4* file.

```
struct metadata {
    /* empty */
}
```

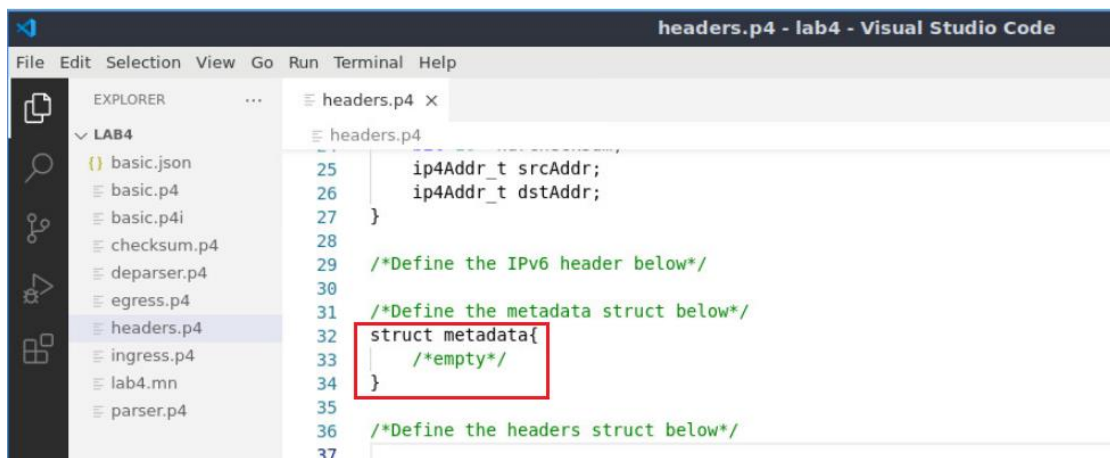


Figure 18. Adding the metadata structures.

Step 6. Now we will create a struct to contain our headers (Ethernet and IPv4). Append the following code to the *headers.p4* file.

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

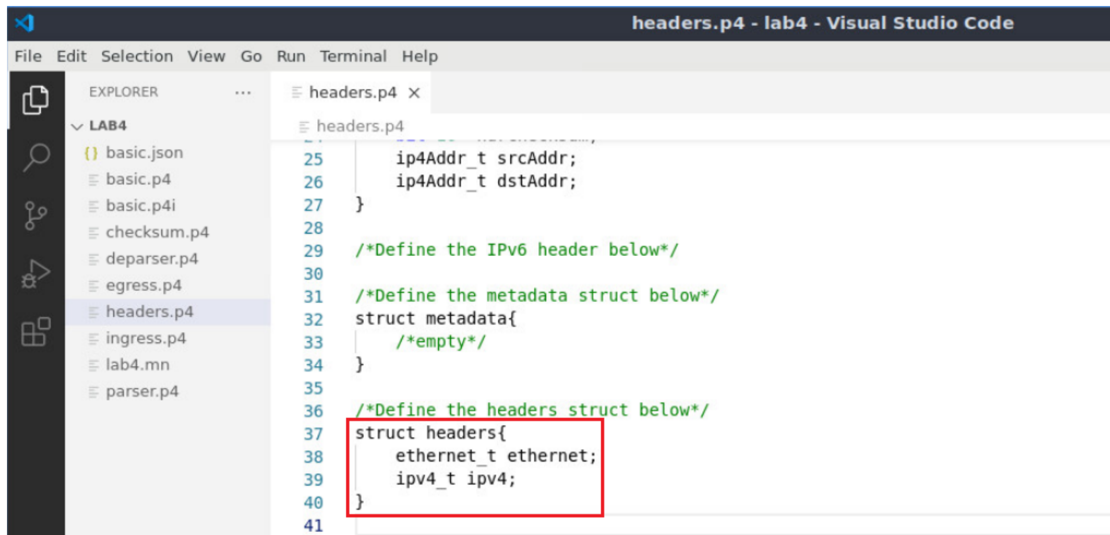


Figure 19. Appending the headers' data structure to the *headers.p4* file.

Step 7. Save the changes by pressing `Ctrl+s`.

4 Parser Implementation

Now it is time to define how the parser works.

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

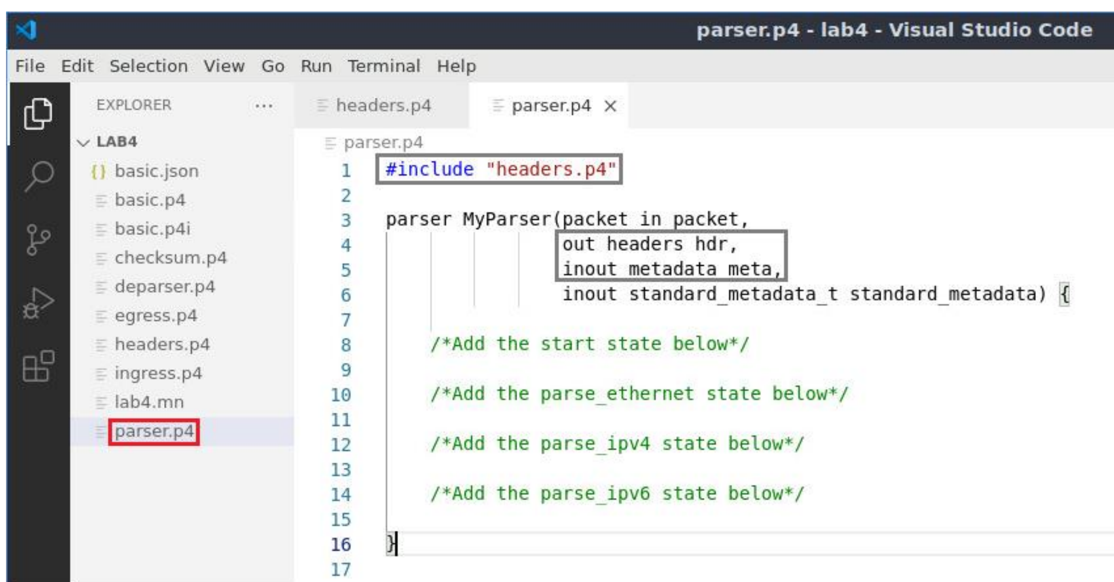


Figure 20. Inspecting the *parse.p4* file.

We can see that the *headers.p4* file that we just filled is included here in the parser. The file also includes a starter code which declares a parser named *MyParser*. Note how the headers and the metadata structs that we defined previously are passed as parameters to the parser.

Step 2. Add the `start` state inside the parser by inserting the following code.

```
state start {
     transition parse_ethernet;
}
```

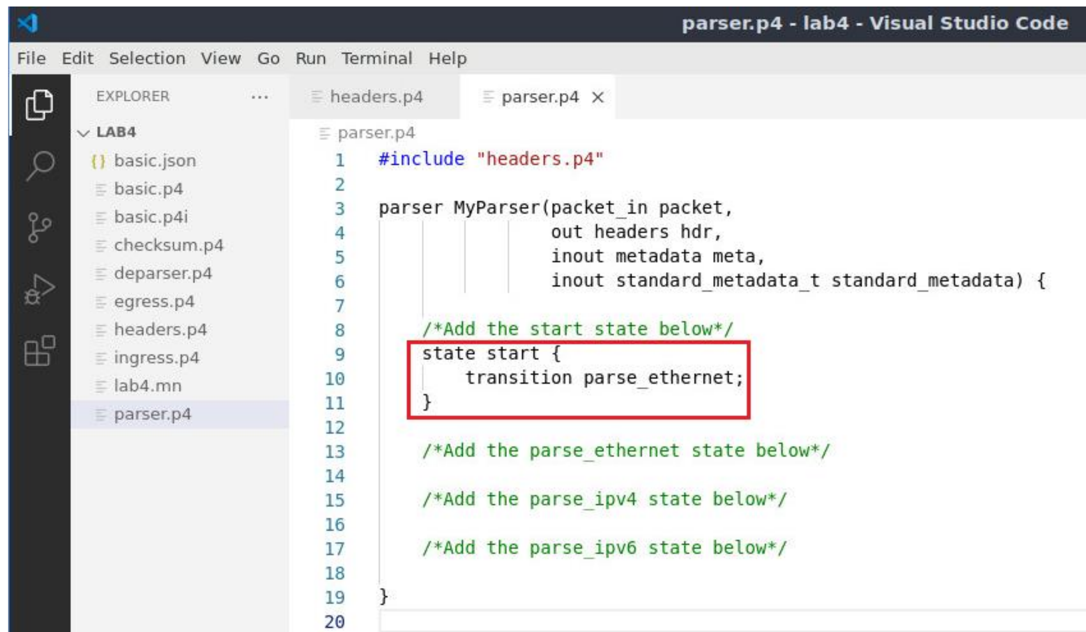


Figure 21. Adding `start` state to the *parser.p4* file.

The `start` state is the state where the parser begins parsing the packet. Here we are transitioning unconditionally to the `parse_ethernet` state.

Step 3. Add the `parse_ethernet` state inside the parser by inserting the following code.

```
state parse_ethernet {
   packet.extract(hdr.ethernet);
   transition select(hdr.ethernet.etherType) {
      TYPE_IPV4: parse_ipv4;
      default: accept;
   }
}
```

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

    /*Add the start state below*/
    state start {
        transition parse_ethernet;
    }

    /*Add the parse_ethernet state below*/
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    /*Add the parse_ipv4 state below*/
```

Figure 22. Adding `parse_ethernet` state to the *parser.p4* file.

The `parse_ethernet` state extracts the Ethernet header and checks for the value of the header field `etherType`. Note how we reference a header field by specifying the header to which that field belongs (i.e., `hdr.ethernet.etherType`). If the value of `etherType` is `TYPE_IPV4` (which corresponds to 0x800 as defined previously), the parser transitions to the `parse_ipv4` state. Otherwise, the execution of the parser terminates.

Step 4. Add the `parse_ipv4` state inside the parser by inserting the following code.

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

Figure 23. Adding `parse_ipv4` state to the *parser.p4* file.

The `parse_ipv4` state extracts the IPv4 header and terminates the execution of the parser.

Step 5. Save the changes to the file by pressing `Ctrl + s`.

5 Loading the P4 program

5.1 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
```

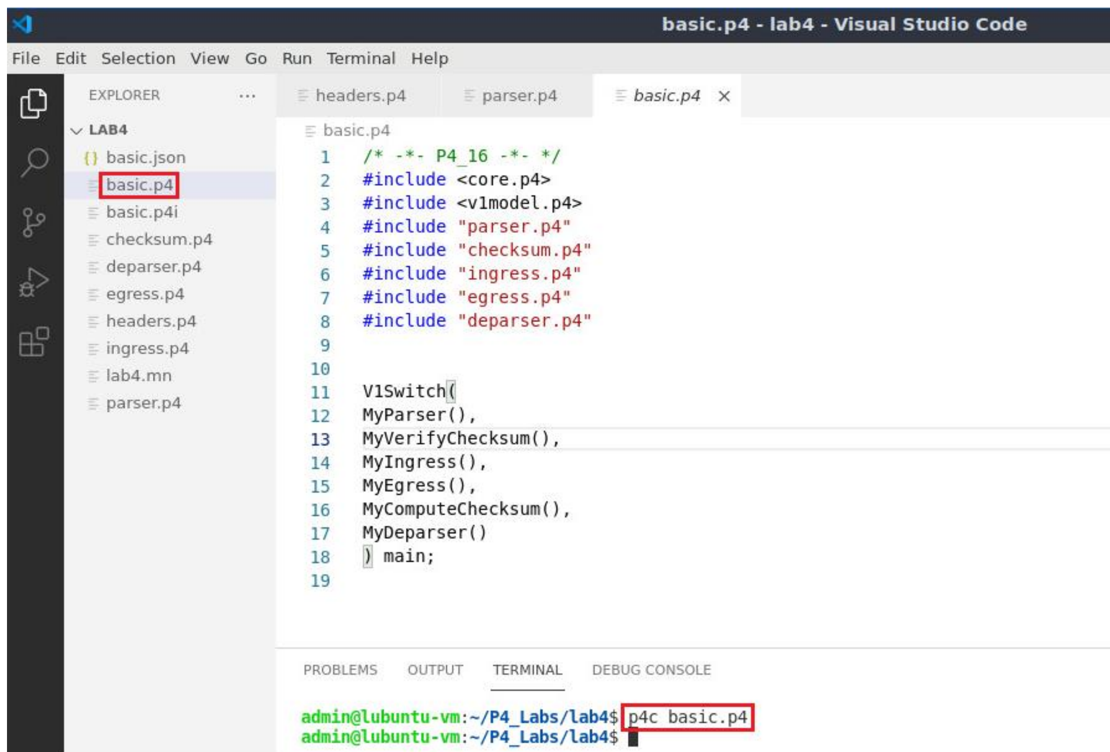


Figure 24. Compiling the code.

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
```

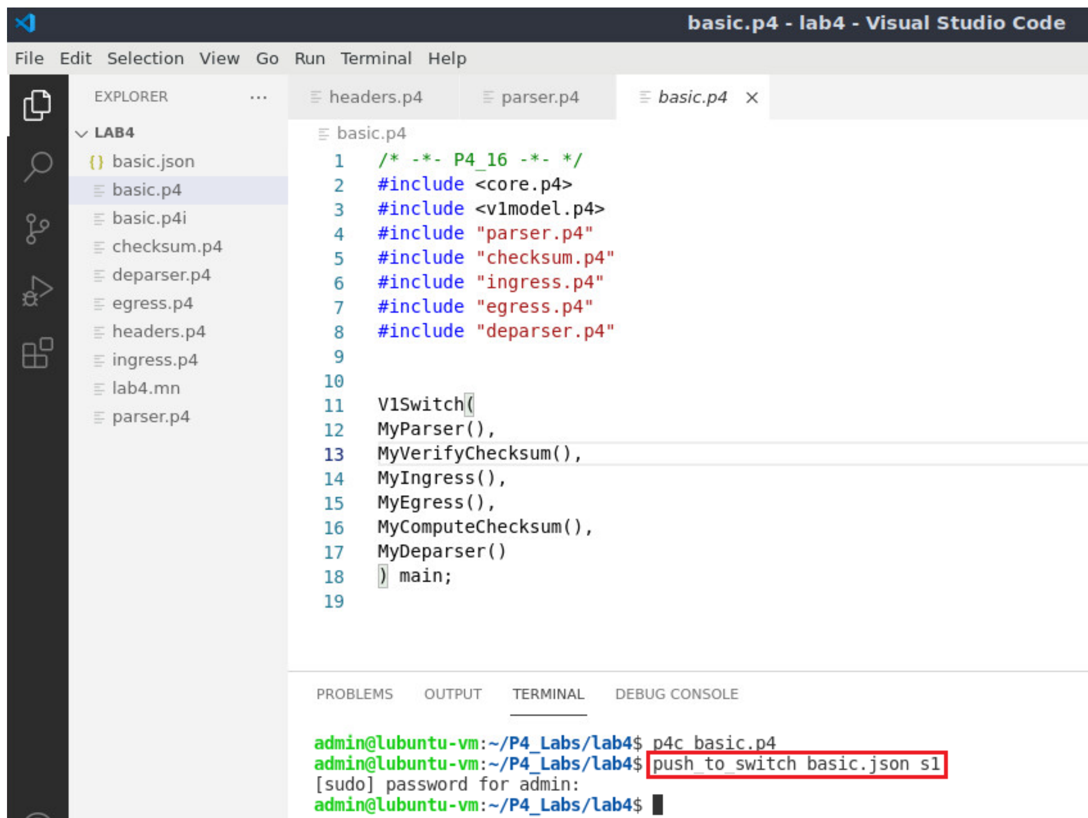


Figure 25. Pushing the P4 program to switch s1.

5.2 Verifying the configuration

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

Figure 26. Maximizing the MiniEdit window.

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

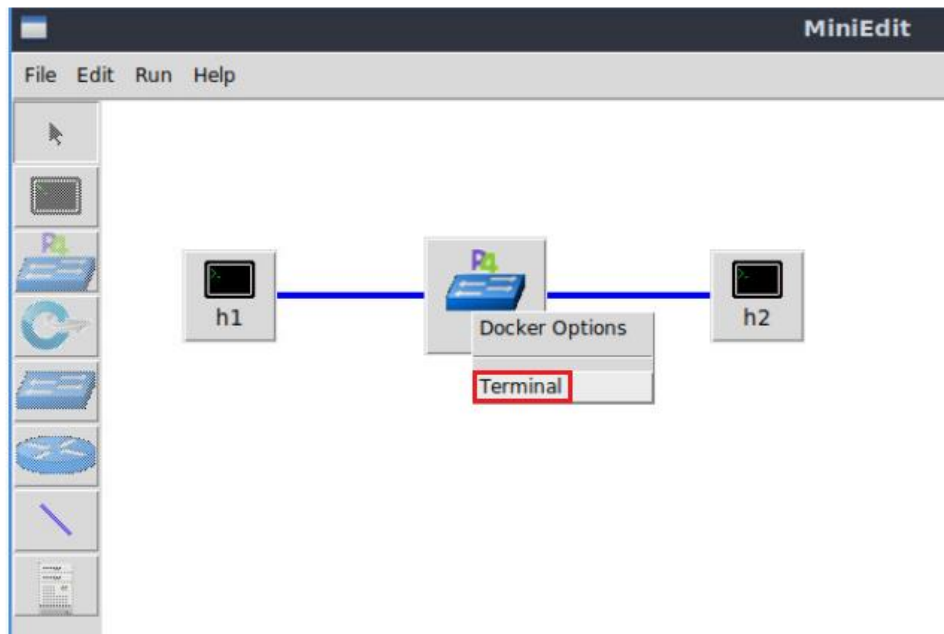


Figure 27. Starting the terminal on the switch.

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
```

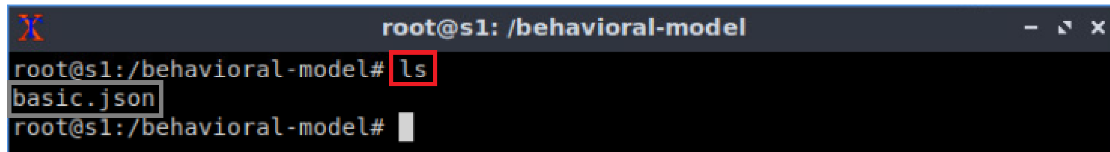


Figure 28. Displaying the contents of the current directory in the switch s1.

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

6 Configuring switch s1

6.1 Mapping P4 program's ports

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

```
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 29. 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] 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 30. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

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

6.2 Loading the rules to the switch

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

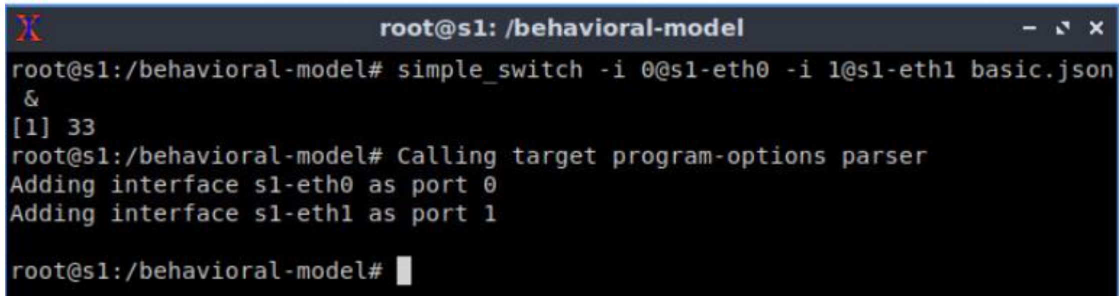


Figure 31. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/lab4/rules.cmd
```

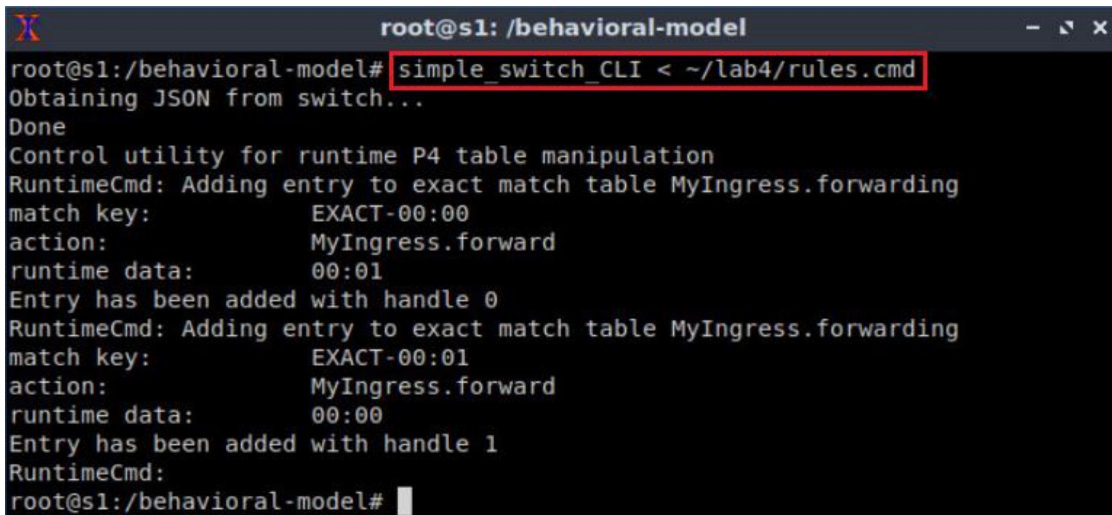


Figure 32. Populating the forwarding table into switch s1.

7 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
```

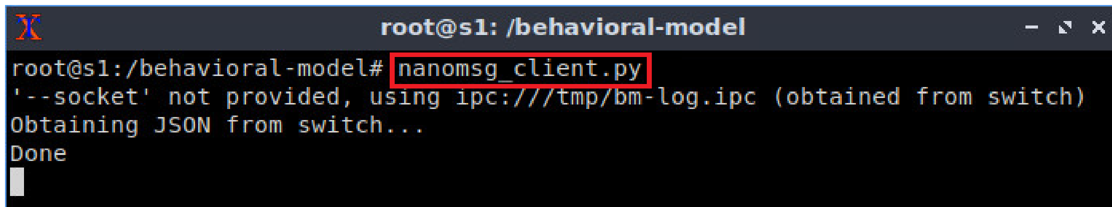


Figure 33. Displaying switch s1 logs.

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

```
./recv.py
```

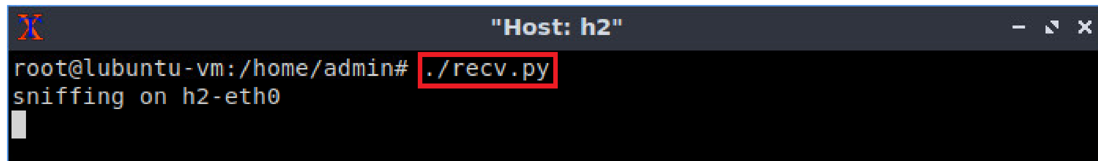


Figure 34. Listening for incoming packets in host h2.

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
```

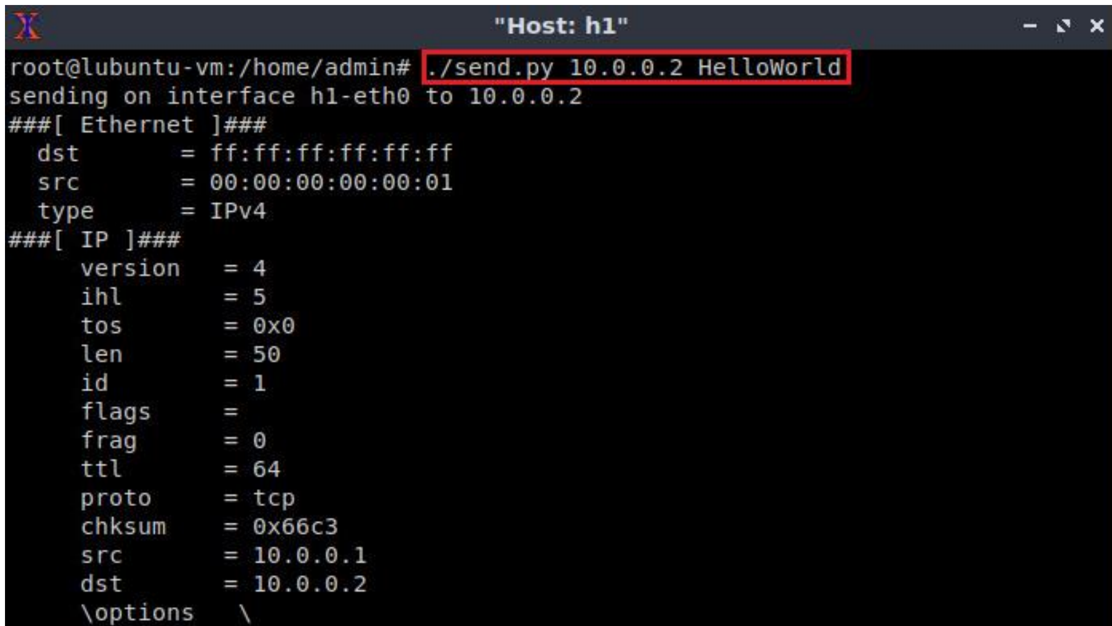


Figure 35. Sending a test packet from host h1 to host h2.

Step 4. Inspect the logs on switch s1 terminal.

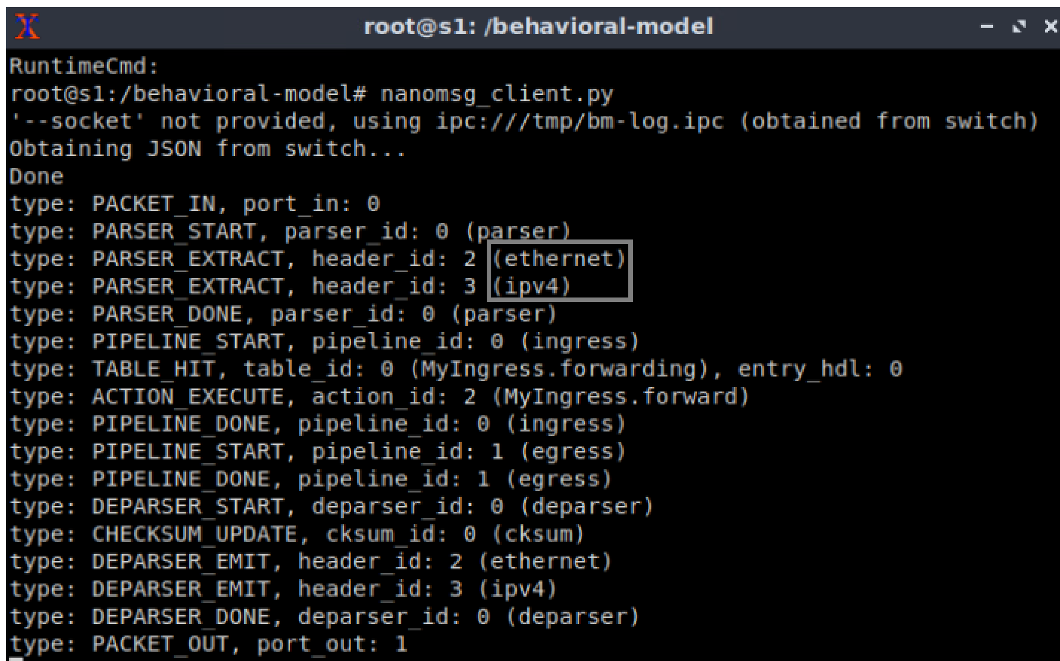


Figure 36. Inspecting the logs in switch s1.

The figure above shows that the Ethernet and IPv4 header are extracted.

8 Augmenting the P4 program to parse IPv6

Now we will augment the program to parse IPv6 packets. Figure 4 shows the IPv6 header fields.

Step 1. Go back to the *headers.p4* file and add the following constant definition.

```
const bit<16> TYPE_IPV6 = 0x86dd;
```

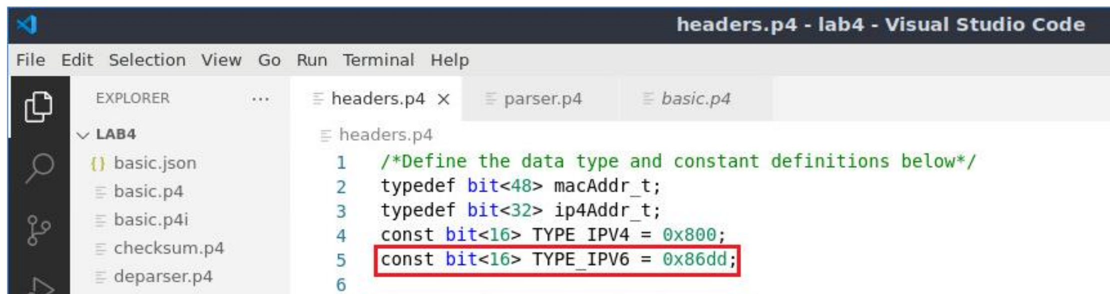


Figure 37. Adding the IPv6 type definition.

Step 2. Add the IPv6 header definition as shown below.

```
header ipv6_t{
       bit<4> version;
       bit<8> trafficClass;
       bit<20> flowLabel;
       bit<16> payloadLen;
       bit<8> nextHdr;
       bit<8> hopLimit;
```

```
        bit<128> srcAddr;
        bit<128> dstAddr;
}
```

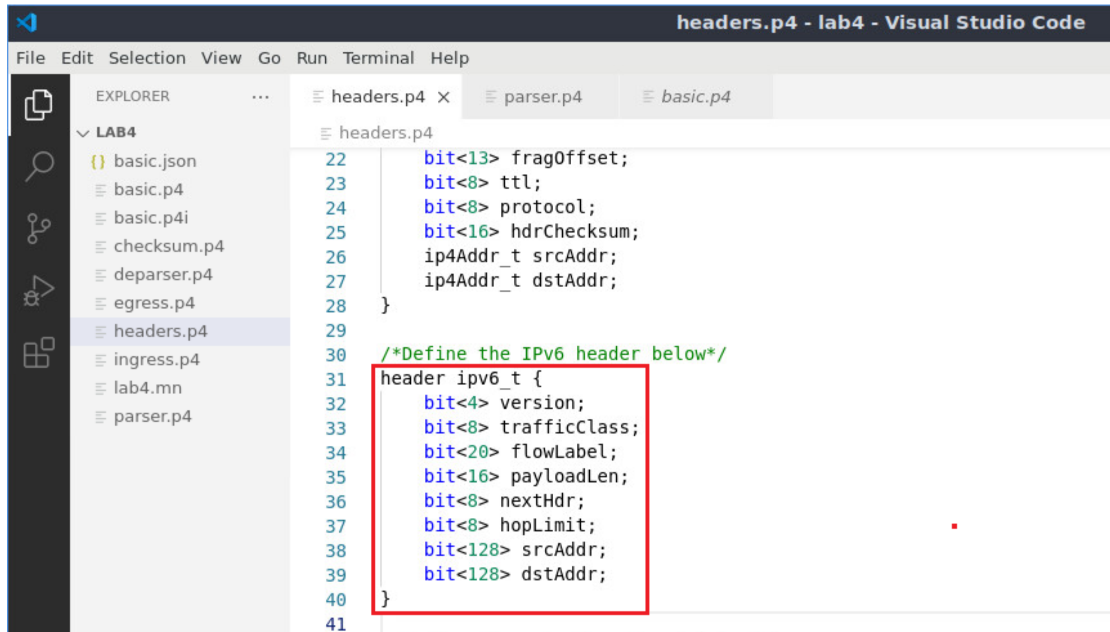


Figure 38. Adding the IPv6 header definition.

Step 3. Append the IPv6 header to the header's data structure.

```
ipv6_t ipv6;
```

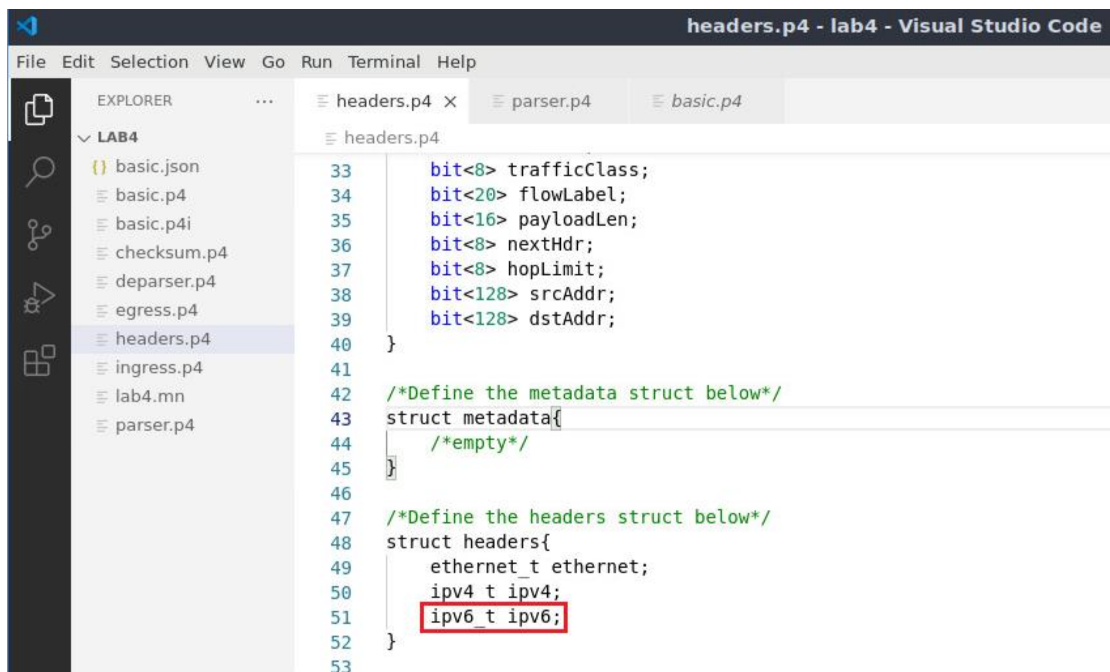


Figure 39. Adding IPv6 type to the header data structure.

Step 4. Go to the *parser.p4* file and add the following line to the `parse_ethernet` state.

```
TYPE_IPV6: parse_ipv6;
```

Figure 40. Including the IPv6 state transition into the `parse_ethernet` state.

Step 5. Add the `parse_ipv6` state inside the parser by inserting the following code.

```
state parse_ipv6 {
       packet.extract(hdr.ipv6);
       transition accept;
}
```

Figure 41. Adding `parse_ipv6` state to the *parser.p4* file.

Step 6. Save the changes by pressing `Ctrl+s`.

Step 7. Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```

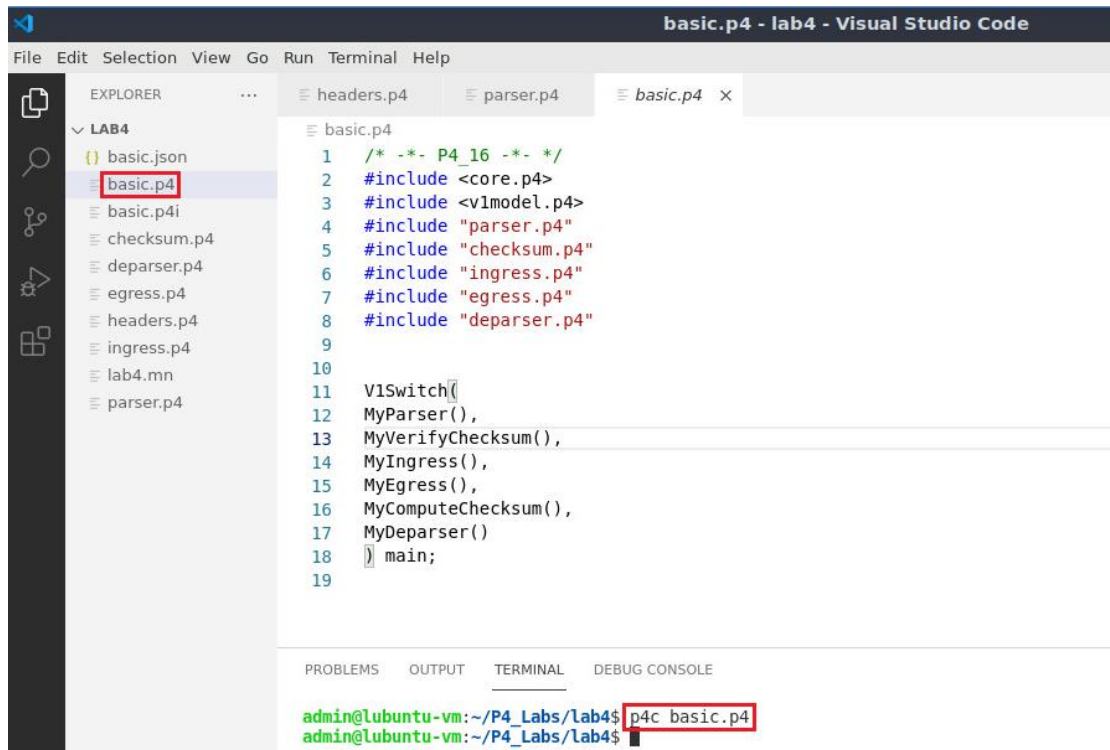


Figure 42. Compiling the P4 program.

Step 8. 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
```

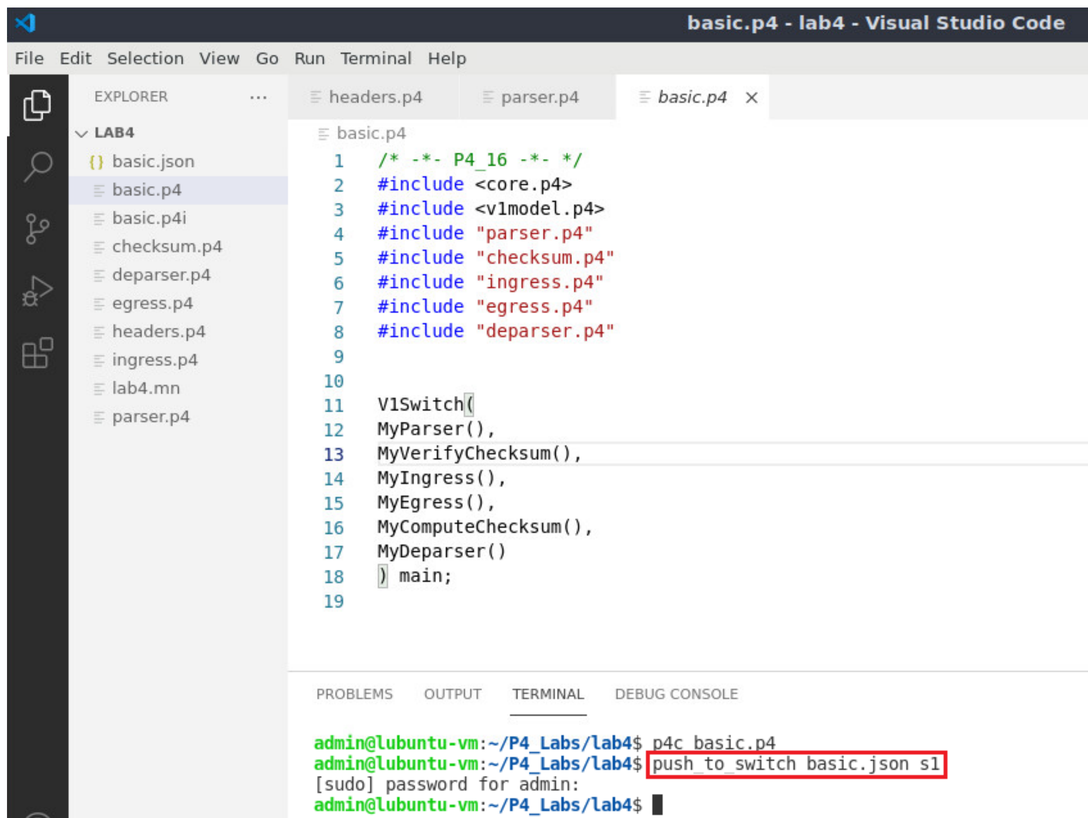


Figure 43. Pushing the P4 program to switch s1.

9 Testing and verifying the augmented P4 program

Step 1. In switch s1 terminal, press `Ctrl + c` to return to the CLI. The figure below shows the output after executing the command.

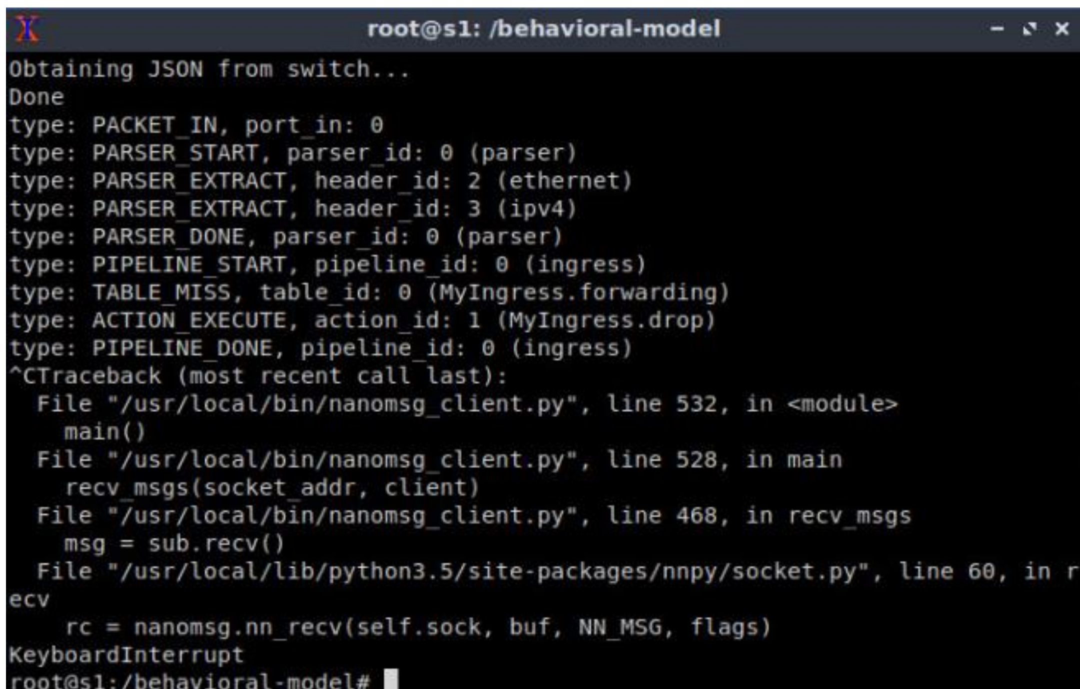


Figure 44. Returning to the CLI.

Step 2. Type the command below in the terminal of switch s1 to stop the running daemon.

```
pkill simple_switch
```

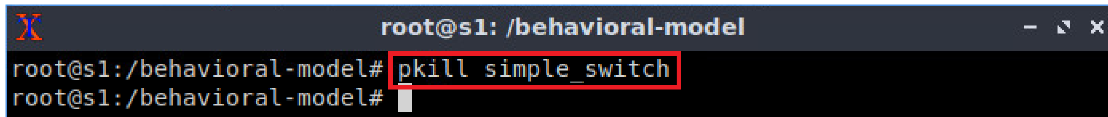


Figure 45. Ending switch s1 P4 process.

Step 3. Type the command below in the terminal of the switch s1 to start the daemon with the new P4 program.

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

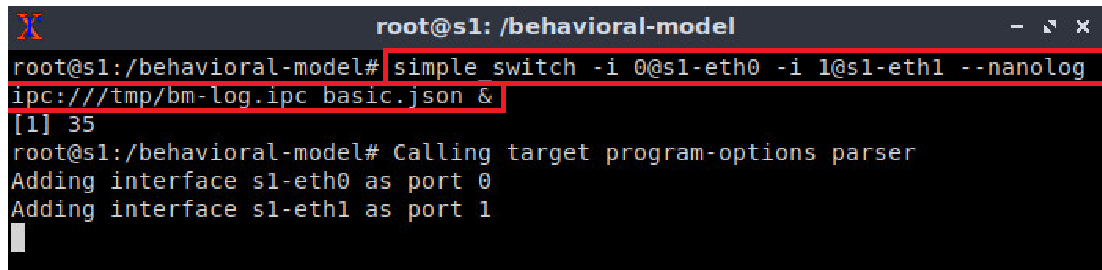


Figure 46. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 4. In switch s1 terminal, press *Enter* to return the CLI.

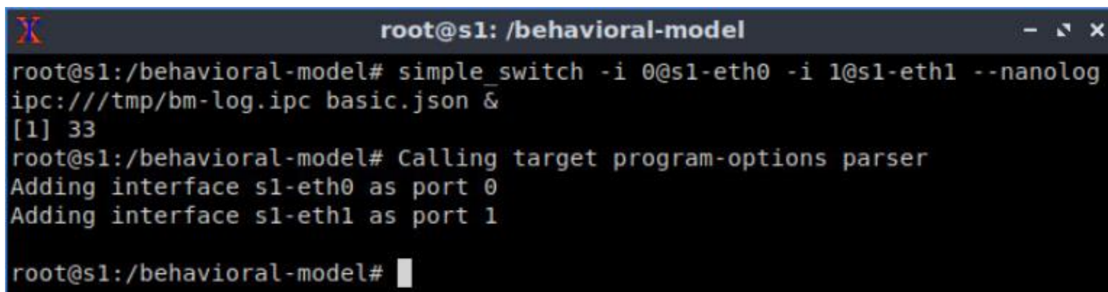


Figure 47. Returning to switch s1 CLI.

Step 5. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab4/rules.cmd
```

```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab4/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 48. Populating the forwarding table into switch s1.

Step 6. 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 49. Inspecting the logs in switch s1.

Step 7. On host h1's terminal, type the following command to send an IPv6 packet to host h2. Note that `bbbb::1` is IPv6 address of 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 50. Sending an IPv6 test packet from host h1 to host h2.

Step 8. Go back to switch s1 and inspect the logs.

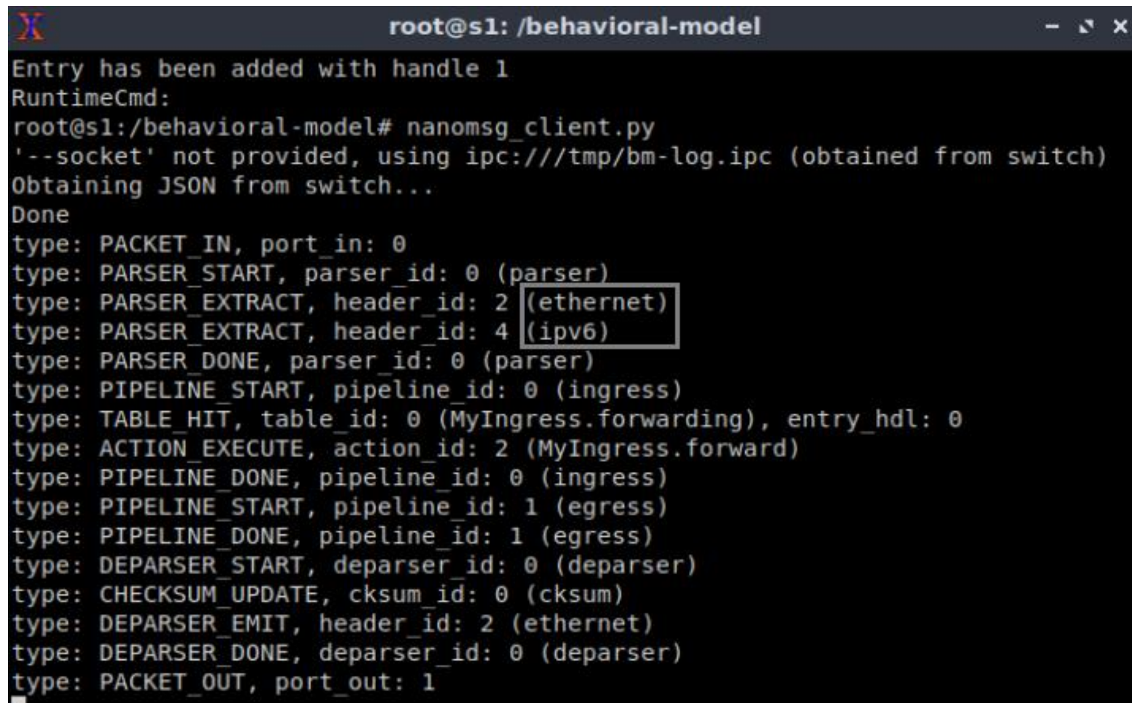


Figure 51. Inspecting the logs in switch s1.

The figure above shows that the Ethernet and IPv6 header are extracted.

This concludes lab 4. Stop the emulation and then exit out of MiniEdit.

References

1. C. Cascaval, D. Daly. "P4 Architectures." [Online]. Available: https://tinyurl.com/3zk8vs6a.
2. “p4c core.p4”. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/core.p4.
3. “p4c v1model.p4”. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.
4. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Exercise 3: Parsing UDP and RTP

Document Version: 01-14-2022

Award 2118311
"Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks"

Contents

1 Exercise description 3
1.1 Ethernet header 3
1.2 IPv4 header 3
1.3 UDP header 3
1.4 RTP header 4
1.5 Exercise topology 4
1.6 Credentials 4
2 Setting the environment 4
3 Deliverables 6

1 Exercise description

In this exercise, you will implement the parser for the User Datagram Protocol (UDP) and the Real-time Transport Protocol (RTP). RTP is used to deliver audio and video over IP networks. The figure below shows the headers of packets arriving to the switch.

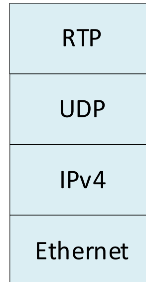


Figure 1. Packet header to be implemented in this exercise.

The header definitions are shown below.

1.1 Ethernet header

Ethernet determines that the next header is IPv4 if the value of *EtherType* is 0x0800.

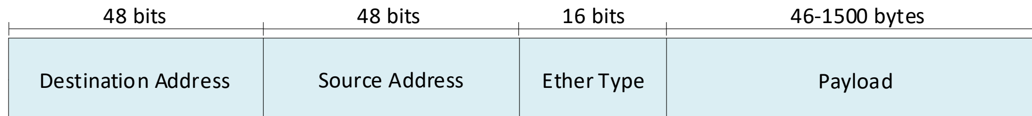


Figure 2. Ethernet header.

1.2 IPv4 header

Bit	0–3	4–7	8–13	14–15	16–18	19–31
0	Version	IHL	DSCP	ECN	Total Length	
32	Identifier				Flags	Fragment Offset
64	Time To Live		Protocol		Header Checksum	
96	Source IP Address					
128	Destination IP Address					
160	Options (if IHL > 5)					

Figure 3. IPv4 header.

The switch can determine that the next header after IPv4 is UDP by inspecting the protocol field of the IPv4 header. The protocol field corresponding to UDP is 17 (i.e., 0x11 in hexadecimal).

1.3 UDP header

We will assume that after parsing UDP, the switch can determine that the next header is RTP by inspecting the destination port of UDP. If the value is 5004 (i.e., 0x138C in hexadecimal), then the next header is RTP.

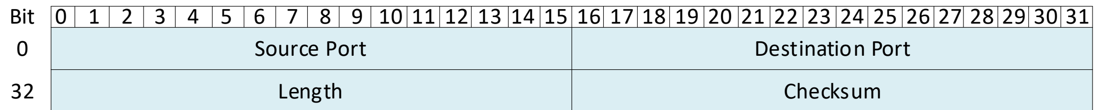


Figure 4. UDP header.

1.4 RTP header

The RTP header format is as follows:

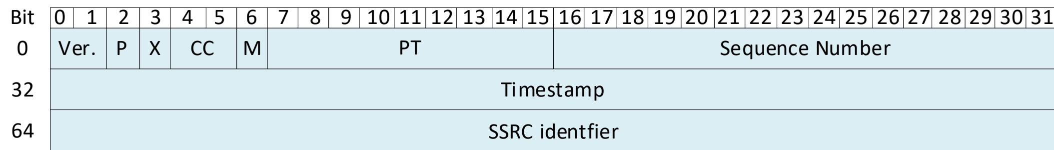


Figure 5. RTP header.

1.5 Exercise topology

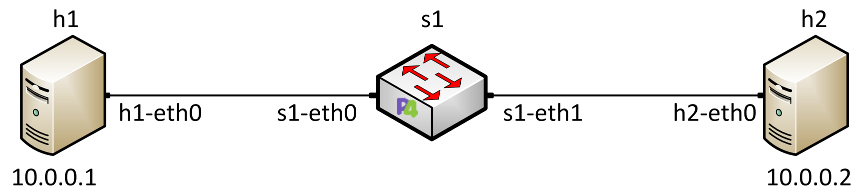


Figure 6. Lab topology.

1.6 Credentials

The information in Table 1 provides the credentials to access the Client's virtual machine.

Table 1. Credentials to access the Client's virtual machine.

Device	Account	Password
Client	admin	password

2 Setting the environment

Follow the steps below to set the exercise's environment.

Step 1. Open MiniEdit by double-clicking the shortcut on the desktop. If a password is required type `password`.

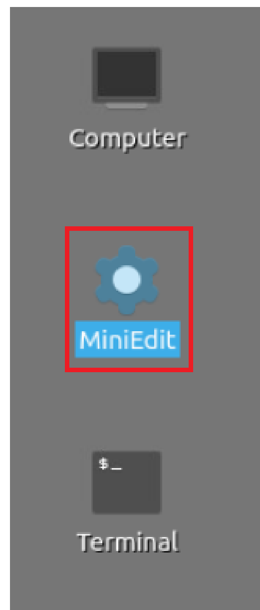


Figure 7. MiniEdit shortcut.

Step 2. Load the topology located at */home/admin/P4_Exercises/Exercise3/*.

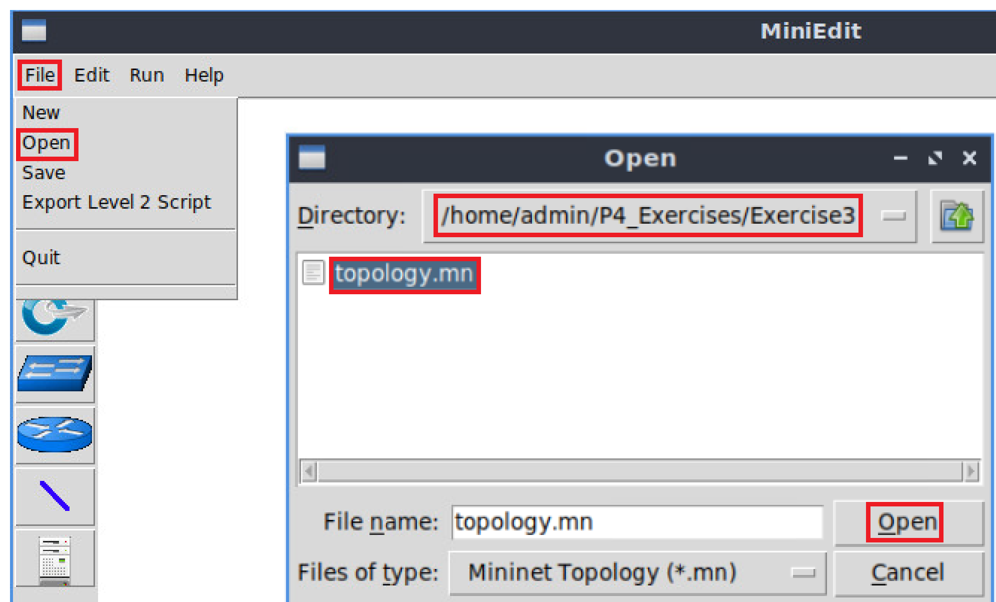


Figure 8. Opening the exercise's topology.

Step 3. Run the emulation by clicking on the button located on the lower left-hand side.

Figure 9. Running the emulation.

Step 4. In the terminal, type the command below. This command launches the Visual Studio Code and opens the directory where the P4 program for this exercise is located.

```
code ~/P4_Exercises/Exercise3/
```

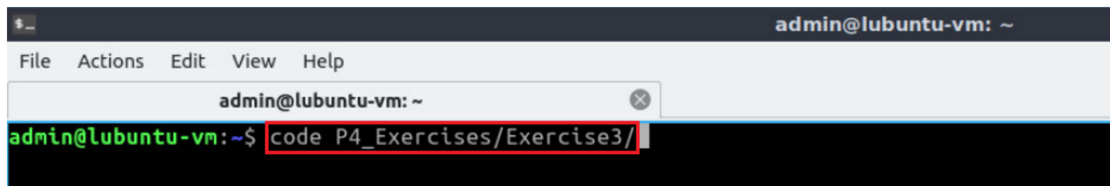


Figure 10. Opening the working directory.

3 Deliverables

Follow the steps below to complete the exercise.

a) Implement the headers for Ethernet, IPv4, UDP, and RTP in the *headers.p4* file.

b) Implement the parser.

c) Compile the *basic.p4* in the Visual Studio Code terminal. Push the output file of the compiler to the switch s1.

d) Start the switch daemon and load the rules located in *~/exercise3/*.

e) In switch s1 terminal, run the *nanomsg_client.py* program to log the events in the switch.

f) Send a packet using the following command.

```
./send_rtp.py 10.0.0.2 HelloWorld
```

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.

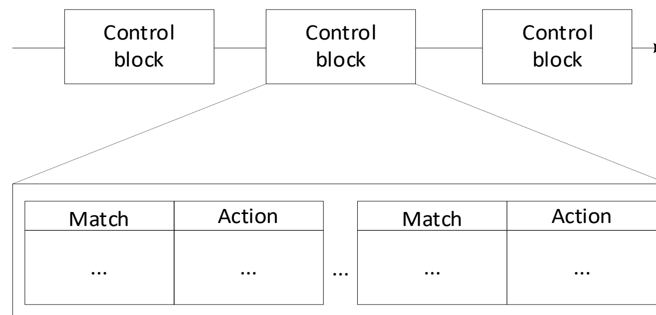


Figure 1. Control blocks.

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 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*[1]). Note that architectures may define and implement additional match types. For example, the V1Model[2] 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.

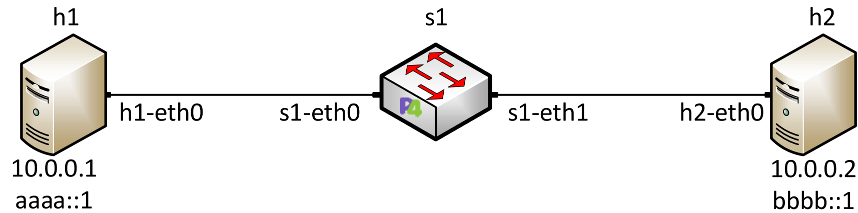


Figure 3. Lab topology.

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.

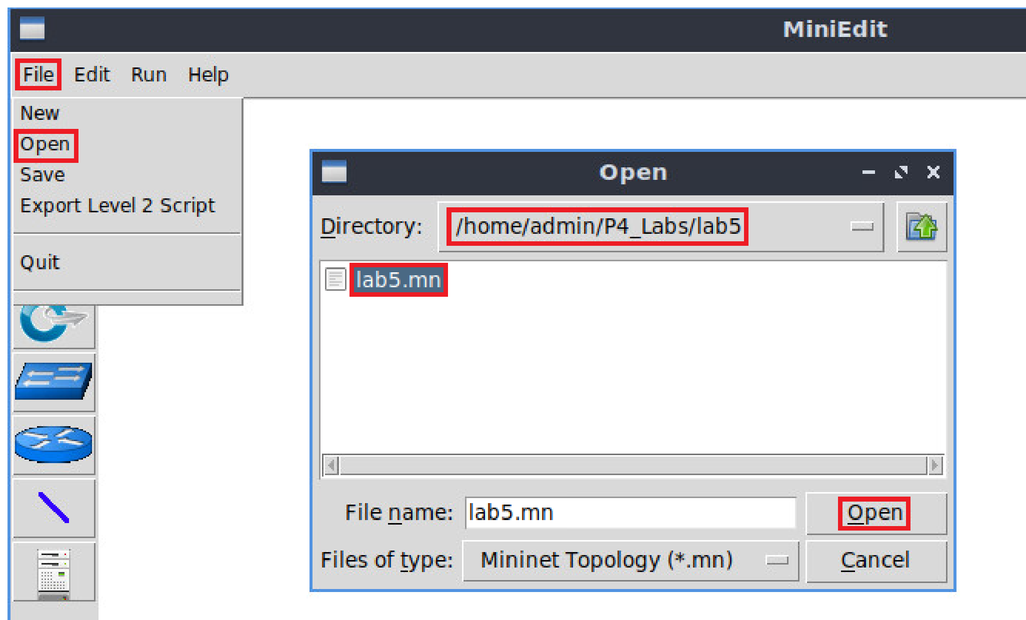


Figure 5. MiniEdit's *Open* dialog.

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.

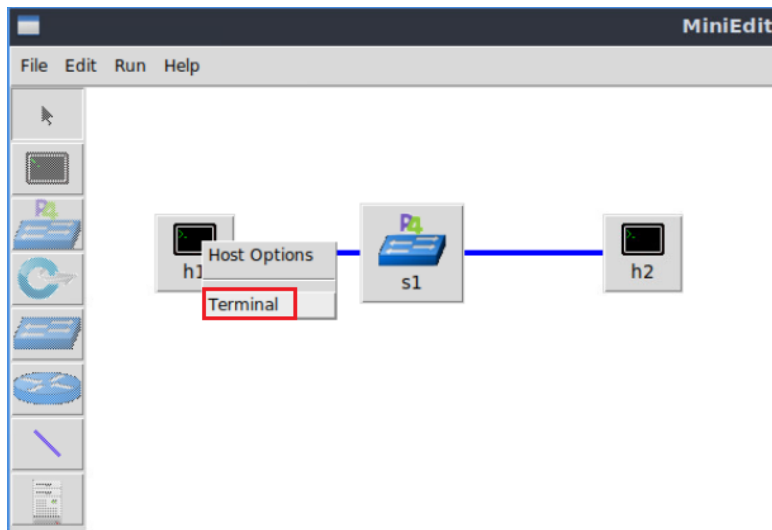


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

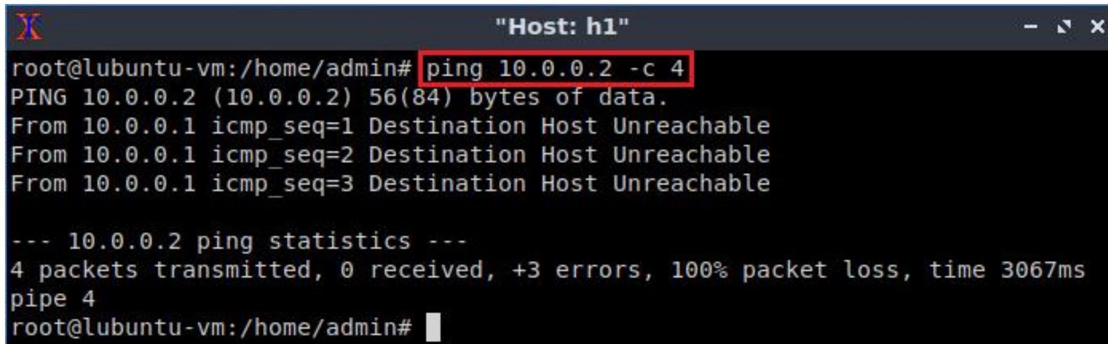


Figure 8. Connectivity test using `ping` command.

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.

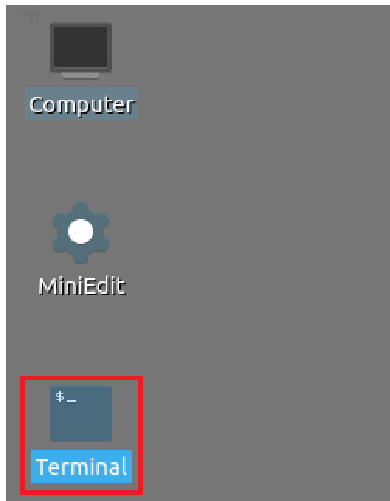


Figure 9. Shortcut to open a Linux terminal.

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
```

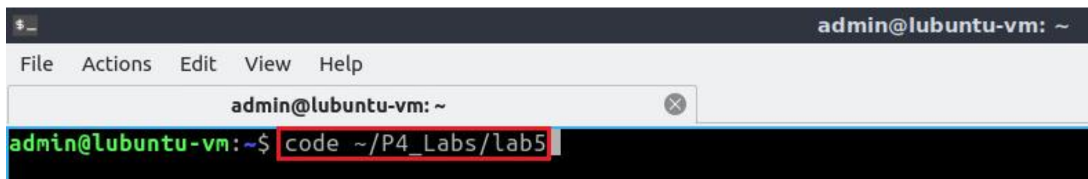


Figure 10. Launching the editor and opening the lab5 directory.

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;
}
```

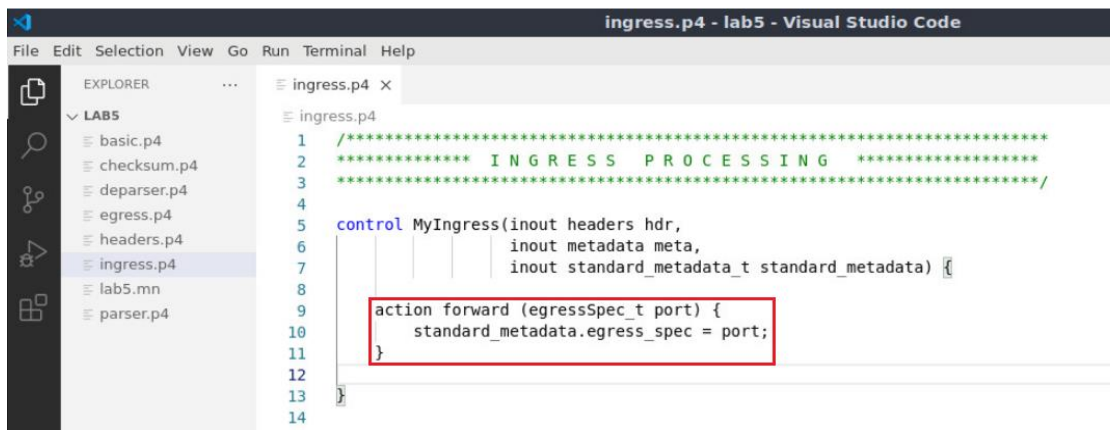


Figure 12. Defining the `forward` action.

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);
}
```

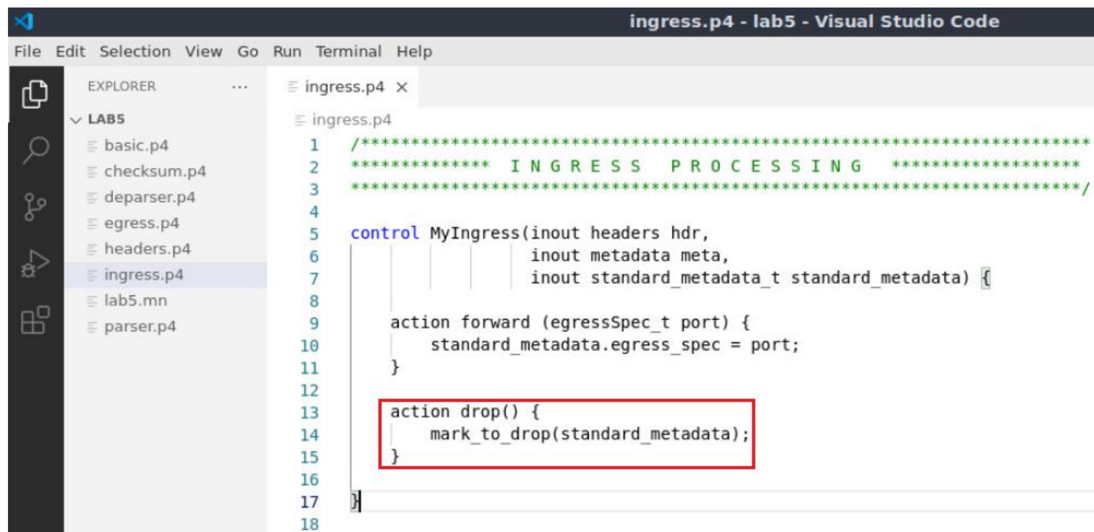


Figure 13. Defining the `drop` action.

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 {

}
```

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

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

    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;
}
```

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table forwarding {
        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;
}
```

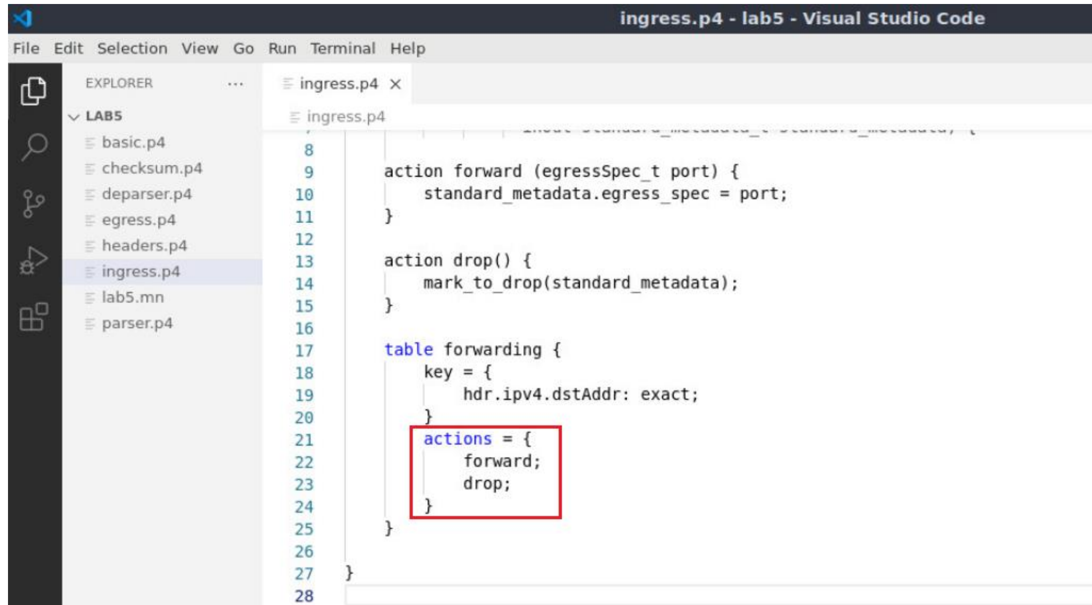


Figure 16. Adding the actions to the `forwarding` table.

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
```

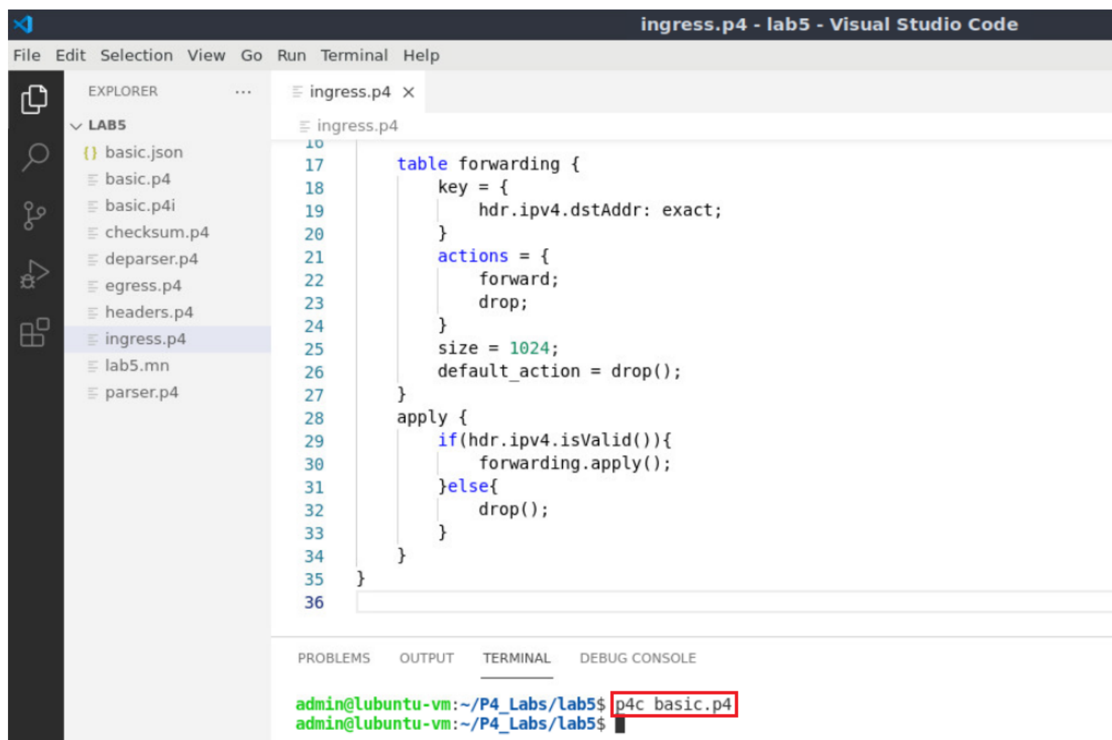


Figure 19. Compiling a P4 program.

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
```

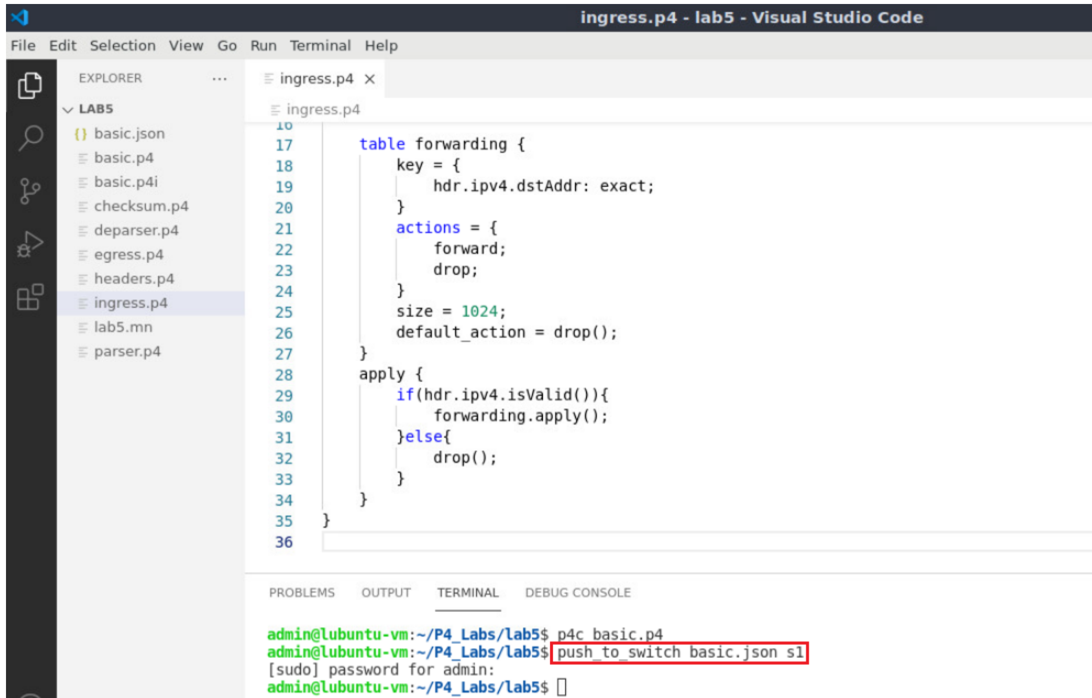


Figure 20. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

Step 1. Click on the MinEdit 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*.

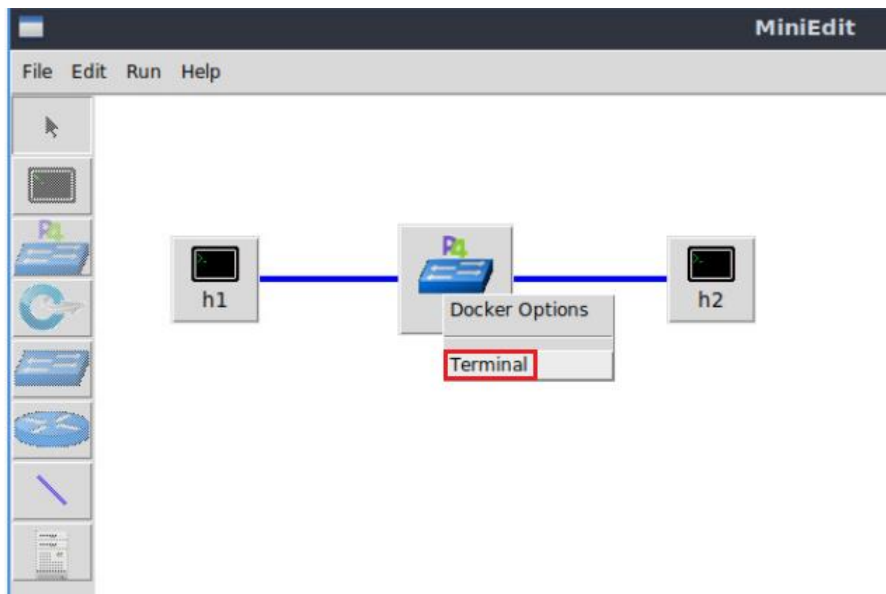


Figure 22. Starting the terminal on the switch.

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
```

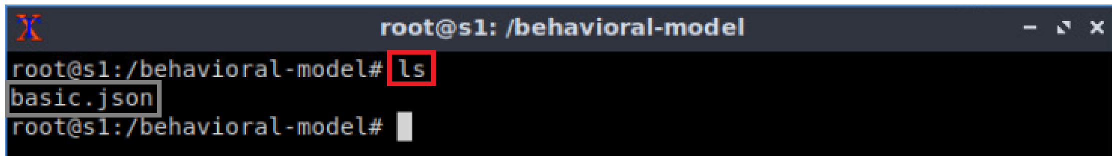


Figure 23. Displaying the contents of the current directory in the switch s1.

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# 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: 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.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 6: Introduction to Match-action Tables (Part 2)

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 3
1.1 Longest prefix match (LPM) 3
2 Lab topology 5
2.1 Starting end hosts 7
3 Defining a table with LPM matching 8
3.1 Loading the programming environment 8
3.2 Programming the ingress block 8
4 Loading the P4 program 13
4.1 Compiling and loading the P4 program to switch s1 13
4.2 Verifying the configuration 15
5 Configuring switch s1 15
5.1 Mapping P4 program's ports 16
5.2 Loading the rules to the switch 17
6 Testing and verifying the P4 program 18
References 21

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.
2. Section 2: Lab topology.
3. Section 3: Defining a table with LPM matching.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Testing and verifying the P4 program.

1 Introduction

1.1 Longest prefix match (LPM)

Table 2 is an example of a match-action table that uses LPM. Assume that the key is formed with the destination IP address. If an incoming packet has the destination IP address 172.168.3.5, two entries match. The first entry matches because the first 29 bits in the entry are the same as the first 29 bits of the destination IP. The second entry also matches because the first 16 bits in the entry are the same as the first 16 bits of the destination IP. The LPM algorithm will select 172.168.3.0/29 because of the longest prefix preference.

Table 2. Match-action table using LPM as the lookup algorithm.

Key	Action	Action data
172.168.3.0/29	forward	port 1, macAddr=00:00:00:00:00:01
172.168.0.0/16	forward	port 2, macAddr=00:00:00:00:00:02
default	drop	

Figure 1 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 - 11) accepts as input (action data) the port and the destination MAC address. These parameters are inserted by the control plane and updated in the packet during the ingress processing.

In line 9, the P4 program assigns the new egress port to the `standard_metadata` egress port field (i.e., the field that the traffic manager looks at to determine which port the packet must be sent to). Line 10 assigns the destination MAC address passed as parameter to the packet's new destination address.

Lines 12-22 implement a table named `ipv4_lpm`. The table is matching against the destination IP address using the LPM type. The actions associated with the table are `forward` and `drop`. The default action is invoked when there is a miss. The maximum number of entries is defined by the programmer (i.e., 1024 entries, see line 20).

The control block starts executing from the apply statement (see lines 23-27) which contains the control logic. In this program, the `ipv4_lpm` table is activated in case 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, macAddr_t dstAddr) {
9:          standard_metadata.egressSpec = port;
10:         hdr.ethernet.dstAddr = dstAddr;
11:     }
12:     table ipv4_lpm {
13:         key = {
14:             hdr.ipv4.dstAddr:lpm;
15:         }
16:         actions = {
17:             forward;
18:             drop;
19:         }
20:         size = 1024;
21:         default_action = drop();
22:     }
23:     apply {
24:         if (hdr.ipv4.isValid()){
25:             ipv4_lpm.apply();
26:         }
27:     }
28: }
```

Figure 1. Ingress control block portion of a P4 program. The code implements a match-action table with LPM lookup.

2 Lab topology

Let's get started by opening a simple Mininet topology using MiniEdit. The topology comprises three end hosts and one P4 programmable switch.

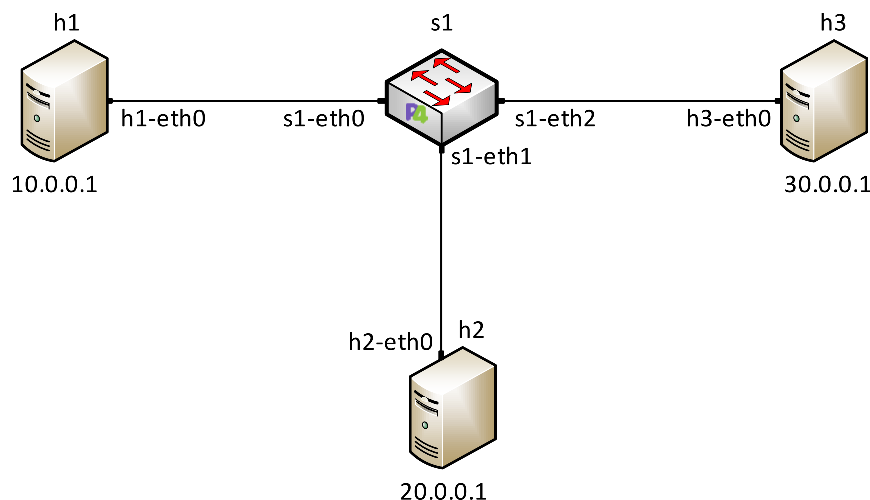


Figure 2. Lab topology.

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 3. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab6* folder and search for the topology file called *lab6.mn* and click on Open. A new topology will be loaded to MiniEdit.

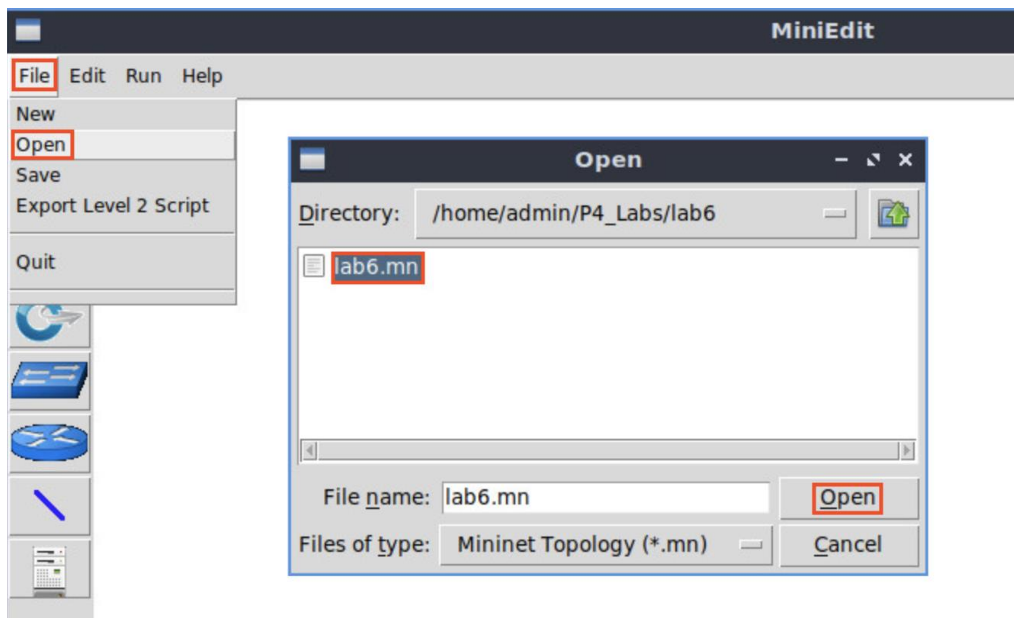


Figure 4. MiniEdit's *Open* dialog.

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 5. Running the emulation.

2.1 Starting end hosts

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.

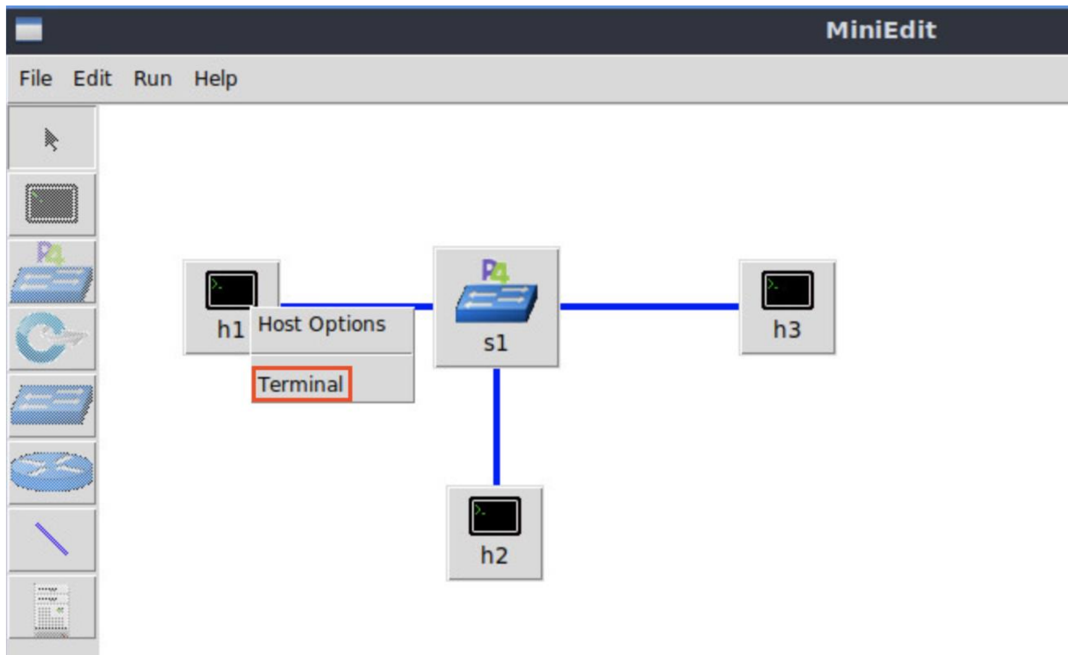


Figure 6. Opening a terminal on host h1.

Step 2. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin#
```

Figure 7. Verifying the configuration host h1 interfaces.

3 Defining a table with LPM matching

This section demonstrates how to implement a simple table in P4 that uses LPM matching on the packet's destination IP address. 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.

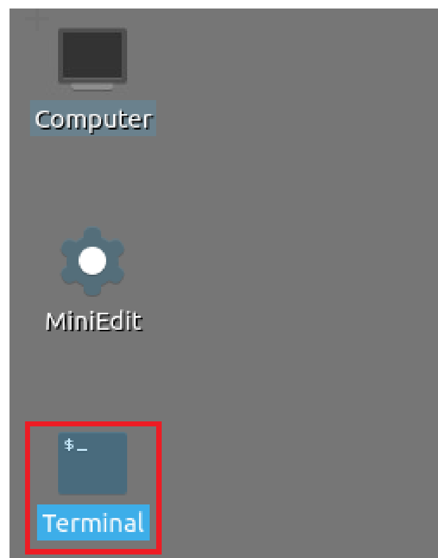


Figure 8. Shortcut to open a Linux terminal.

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.

```
code ~/P4_Labs/lab6
```

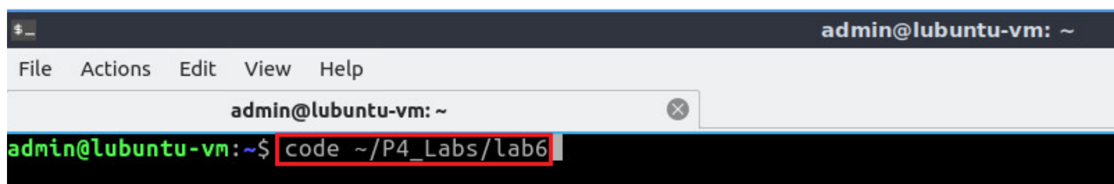


Figure 9. Launching the editor and opening the lab6 directory.

3.2 Programming 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.

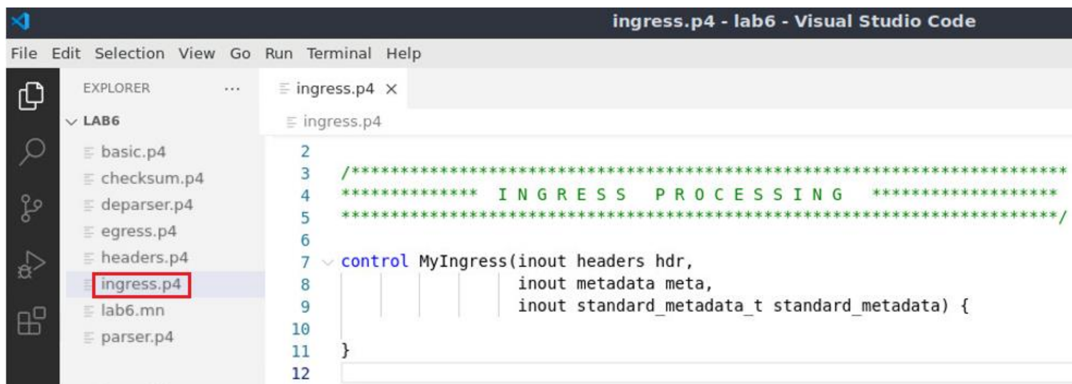


Figure 10. Opening the ingress processing block.

We can see that the *ingress.p4* declares a control block named `MyIngress`. Inside the block, we will define a table `ipv4_host` that is used to match on the destination IP address and forward/drop the packet. There are two actions that will be invoked in this table: `forward` and `drop`.

- `forward`: This action defines a set of basic operations on a packet header. Such operations are defined as follows: 1) Updating the egress port so the packet is forwarded to its destination through the correct port. 2) Updating the source MAC address with the packet's previous destination MAC address. 3) Changing the destination MAC address of the packet with the one corresponding to the next hop. 4) Decrementing the time-to-live (TTL) field in the IPv4 header.
- `drop`: this action will be used to drop the packet.

Step 2. The following code fragment describes the behavior of the `forward` action. Insert the code below inside the *MyIngress* control block.

```
action forward(macAddr_t dstAddr, egressSpec_t port){
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

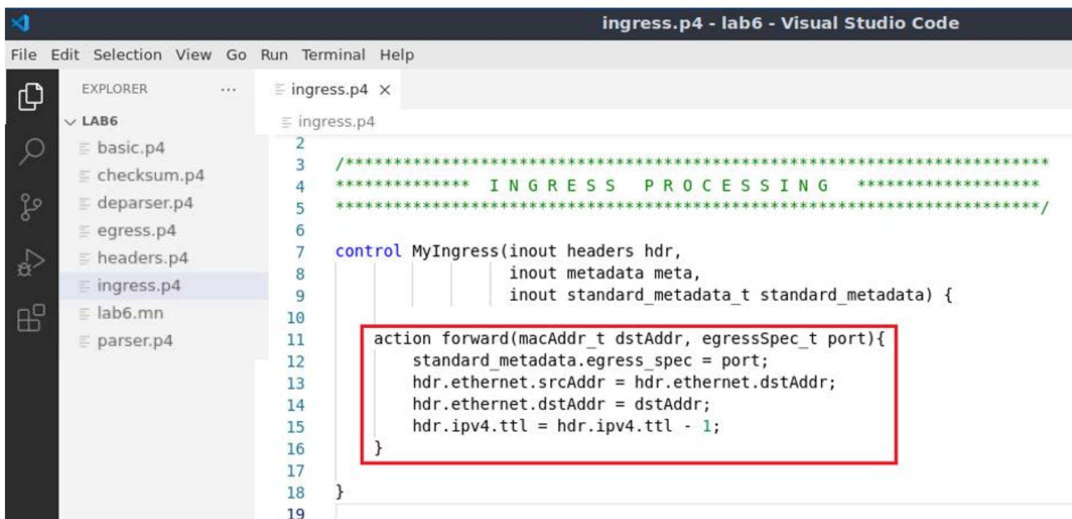


Figure 11. Defining the `forward` action.

The action `forward` accepts as parameters the next hop's MAC address (i.e., `macAddr_t dstAddr`) and the port number (i.e., `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>` and `macAddr_t` is a typedef that corresponds to `bit<48>`. These types are defined in the *headers.p4* file.

The `standard_metadata` is an instance of the `standard_metadata_t` struct provided by the V1Model[1]. This struct contains intrinsic metadata used 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` (i.e., `standard_metadata.ingress_port`). Similarly, the egress port `egress_spec` field of the `standard_metadata` defines the egress port. Line 12 shows how to assign the egress port to forward an incoming packet to its destination.

To modify header fields inside the packet, we refer to the field name based on where it exists inside the headers. Recall that the names of the headers and the fields are defined by the programmer. The file *headers.p4* defines the program's headers. Line 13 shows how we are assigning the destination MAC address of the packet (i.e., `hdr.ethernet.dstAddr`) to be the new source MAC of the packet (i.e., `hdr.ethernet.srcAddr`). Line 14 shows how we are assigning the destination MAC address which is provided as a parameter (assigned later in the control plane) to be the new destination MAC of the packet.

It is possible in P4 to perform basic arithmetic operations on header fields and other variables. In line 15, we are decrementing the TTL value of the header field.

Step 3. Define the drop action by appending the following code into the *MyIngress* control block.

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

Figure 12. Defining the `drop` action.

Step 4. Define an exact match table by appending the following piece of code.

```
table ipv4_exact {
    key = {
        hdr.ipv4.dstAddr: exact;
    }
    actions = {
        forward;
        drop;
    }
    size = 1024;
    default_action = drop();
}
```

Figure 13. Defining the table `ipv4_exact` implementing exact match lookup.

Step 5. Now we will define a table that performs a LPM on the destination IP address of the packet. The table will be invoking the forward and the drop actions, and hence, those actions will be listed inside the table definition.

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        forward;
        drop;
    }
    size = 1024;
    default_action = drop();
}
```

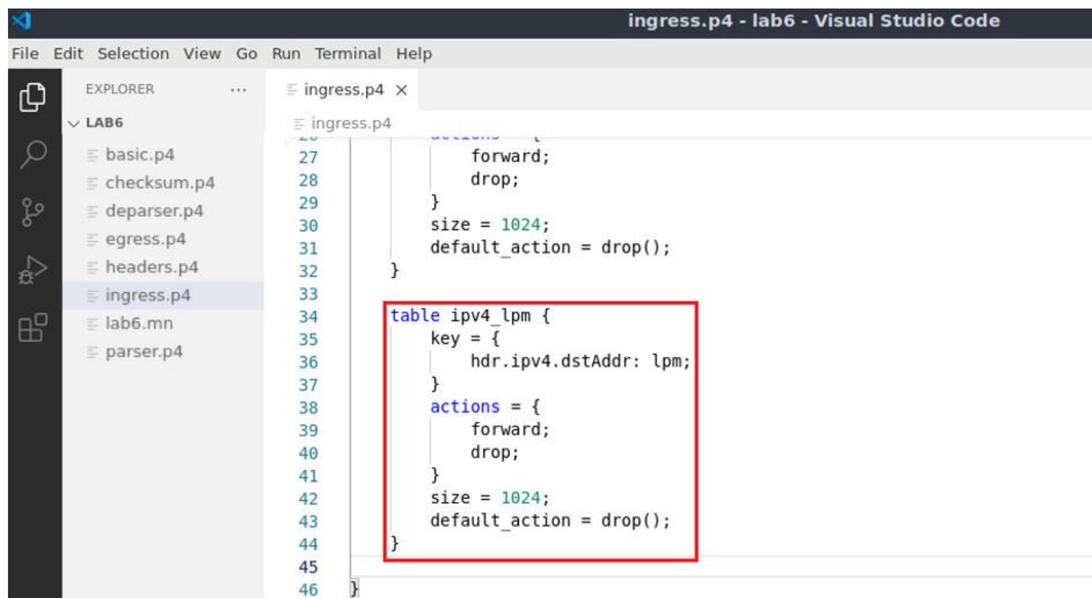


Figure 14. Defining the table `ipv4_lpm` implementing LPM lookup.

Step 6. Add the following code at the end of 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 the sequence of tables to be invoked, in addition to other packet processing instructions.

```
apply {
    if(hdr.ipv4.isValid()) {
        if(ipv4_exact.apply().miss) {
            ipv4_lpm.apply();
        }
    }
}
```

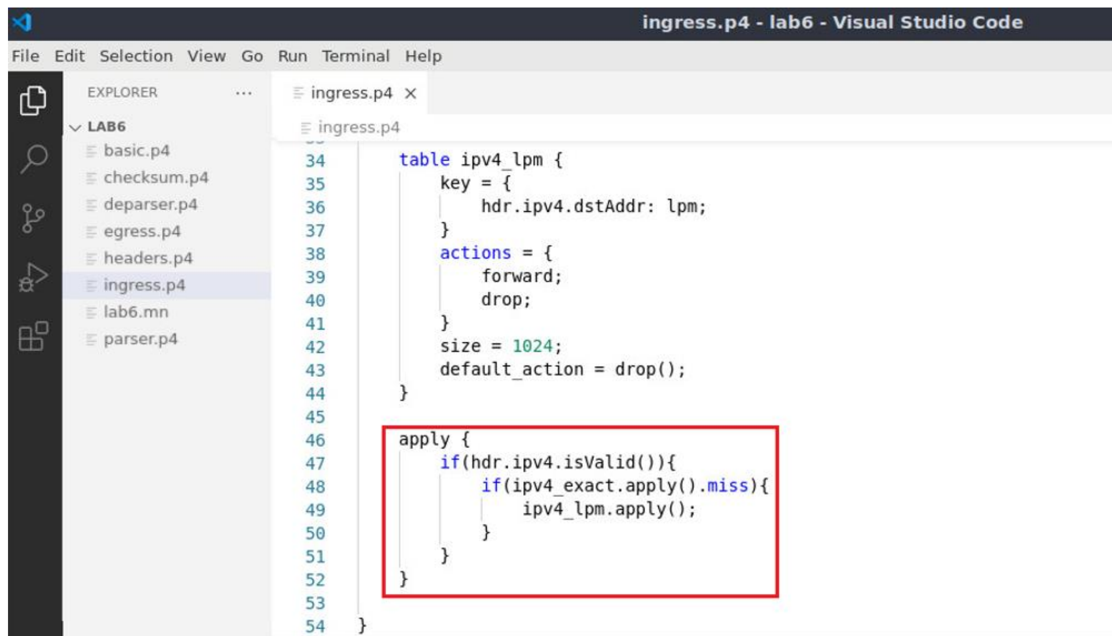


Figure 15. Defining the `apply` block.

The logic of the code above is as follows: if the packet has an IPv4 header, apply the `ipv4_exact` table which performs an exact match lookup on the destination IP address. If there is no *hit* (i.e., the table does not contain a rule that corresponds to this IPv4 address, denoted by the *miss* keyword), apply the `ipv4_lpm` table, which matches the destination IP address of the packet against a network address.

Step 7. 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 VS Code to compile the program.

```
p4c basic.p4
```

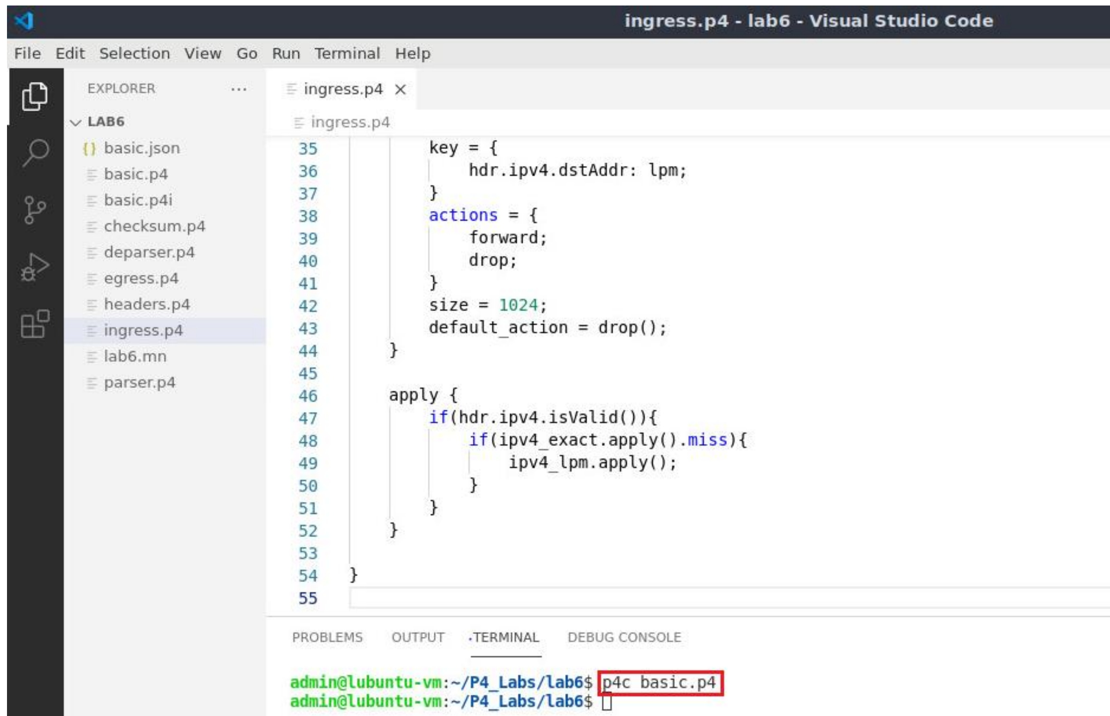


Figure 16. Compiling a P4 program.

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
```

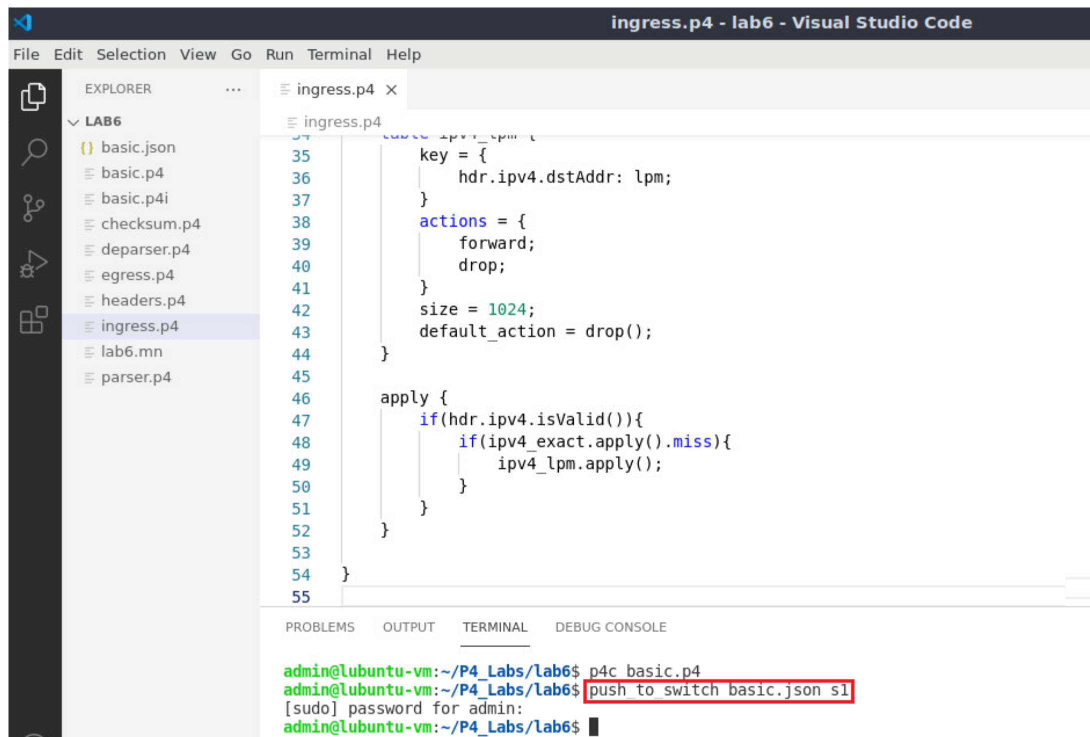


Figure 17. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

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

Figure 18. Maximizing the MiniEdit window.

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

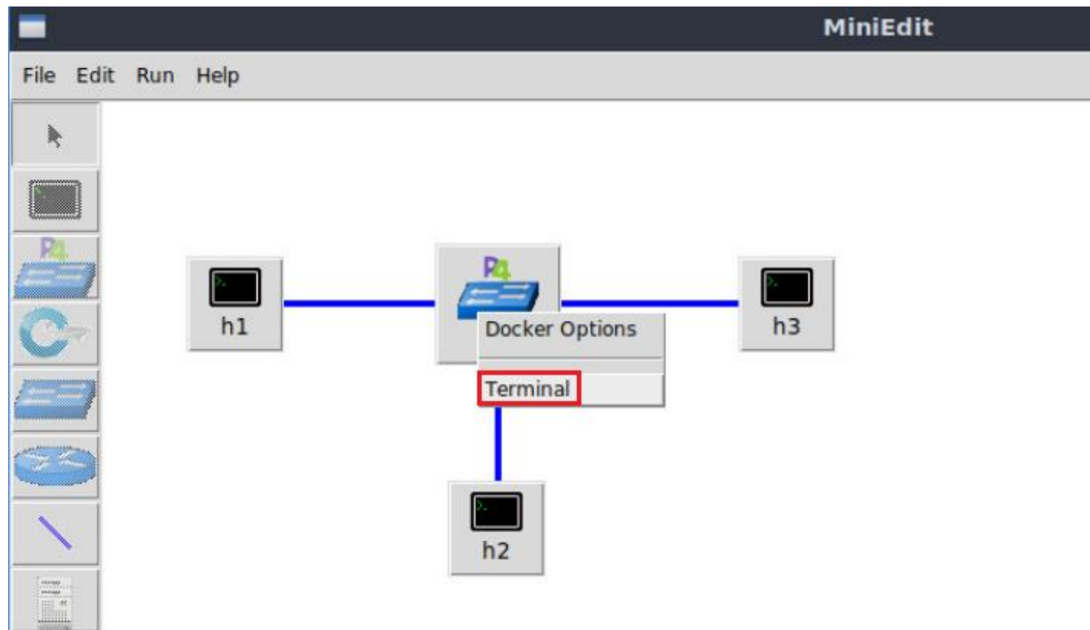


Figure 19. Opening switch s1 terminal.

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 following command on switch s1 terminal to inspect the content of the current folder.

```
ls
```

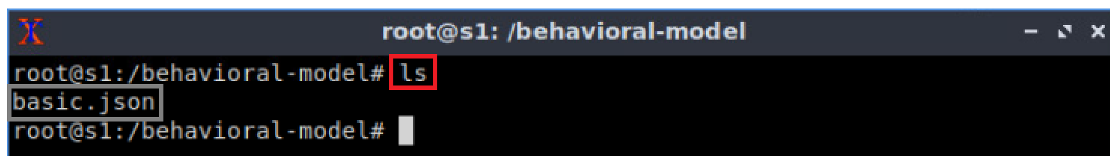


Figure 20. Displaying the content of the current directory in the switch s1.

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 following command on 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:27 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:3265 (3.2 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:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

s1-eth0   Link encap:Ethernet  HWaddr 0e:7e:48:32:53:a3
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:356 (356.0 B)  TX bytes:0 (0.0 B)

s1-eth1   Link encap:Ethernet  HWaddr 9e:c5:42:78:07:16
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)

s1-eth2   Link encap:Ethernet  HWaddr 26:15:f3:b2:b1:d4
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)
```

Figure 21. Displaying switch s1 interfaces.

The output displays switch s1 interfaces (i.e., *s1-eth0*, *s1-eth1* and *s1-eth2)*. The interface *s1-eth0* on the switch s1 connects to the host h1. The interface *s1-eth1* on the switch s1 connects to the host h2 and *s2-eth2* is connected to host h3.

Step 2. Start the switch daemon and map the logical interfaces (i.e., ports) to the switch's interfaces by issuing the following command. The `--nanolog` parameter is used to instruct the switch daemon to provide the switch's logs.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog ipc:///tmp/bm-log.ipc  basic.json &
```

Figure 22. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

5.2 Loading the rules to the switch

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

Figure 23. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/lab6/rules.cmd
```

Figure 24. Populating the forwarding table into switch s1.

The script above pushes the rules to the switch daemon. We can see that we added three entries to the `ipv4_exact` and `ipv4_lpm` tables.

- The key of the first entry is 10.0.0.0/8 (which translates to 0a:00:00:00 in hexadecimal as shown in the figure above, next to match key) and its action is forward. This entry is added to the `ipv4_lpm` table. The action parameters or runtime data are 00:00:00:00:00:01 for the destination MAC (i.e., host h1's MAC address) and 0 for the output port (i.e., the port facing host h1).
- The key of the second entry is 20.0.0.0/8 (which translates to 14:00:00:00 in hexadecimal as shown in the figure above, next to match key) and its action is forward. This entry is added to the `ipv4_lpm` table. The action parameter or runtime data are 00:00:00:00:00:02 for the destination MAC (i.e., host h2's MAC address) and 1 for the output port (i.e., the port facing host h2).
- The key of the third entry is 30.0.0.1 (which translates to 1e:00:00:01 in hexadecimal as shown in the figure above, next to match key) and its action is forward. This entry is added to the `ipv4_host` table. The action values are 00:00:00:00:00:03 for the destination MAC (i.e., host h3's MAC address) and 2 for the output port (i.e., the port facing host h3).

6 Testing and verifying the P4 program

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

```
nanomsg_client.py
```

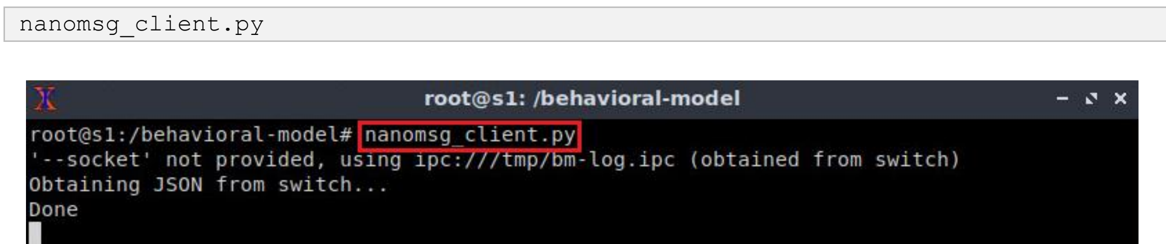


Figure 25. Displaying switch s1 logs.

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

```
./recv.py
```

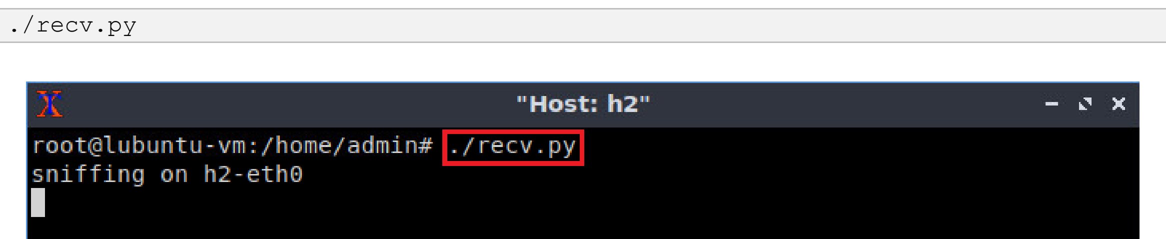


Figure 26. Listening for incoming packets in host h2.

Step 3. On host h1's terminal, type the following command to send a message to host h2. The output will show the Ethernet, IP and TCP header fields and their values. The payload is `HelloWorld`.

```
./send.py 20.0.0.1 HelloWorld
```

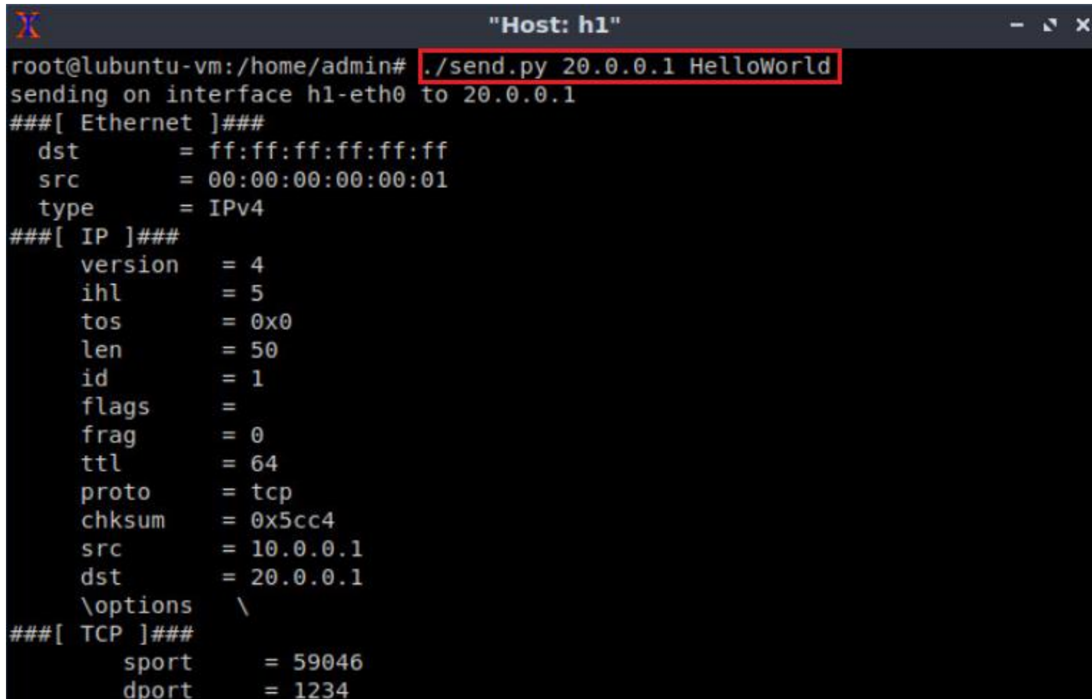


Figure 27. Sending a test packet from host h1 to host h2.

Step 4. Inspect the logs on switch s1 terminal.

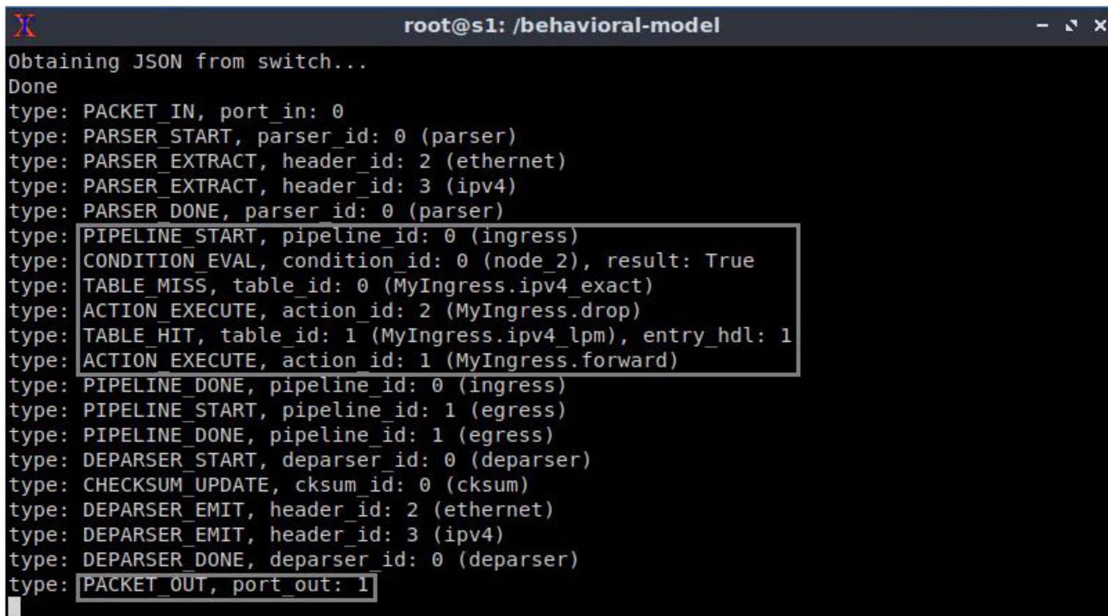


Figure 28. Inspecting the logs in switch s1.

Results show that there is a miss in the `ipv4_exact` table, but there is a hit on the `ipv4_lpm` table. Then, the packet is forwarded through port 1, which is connected to host h2. This behavior corresponds to the logic described by the `apply` block in the ingress processing.

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

Step 6. On host h1's terminal, type the following command to send a message to host h3.

```
./send.py 30.0.0.1 HelloWorld
```

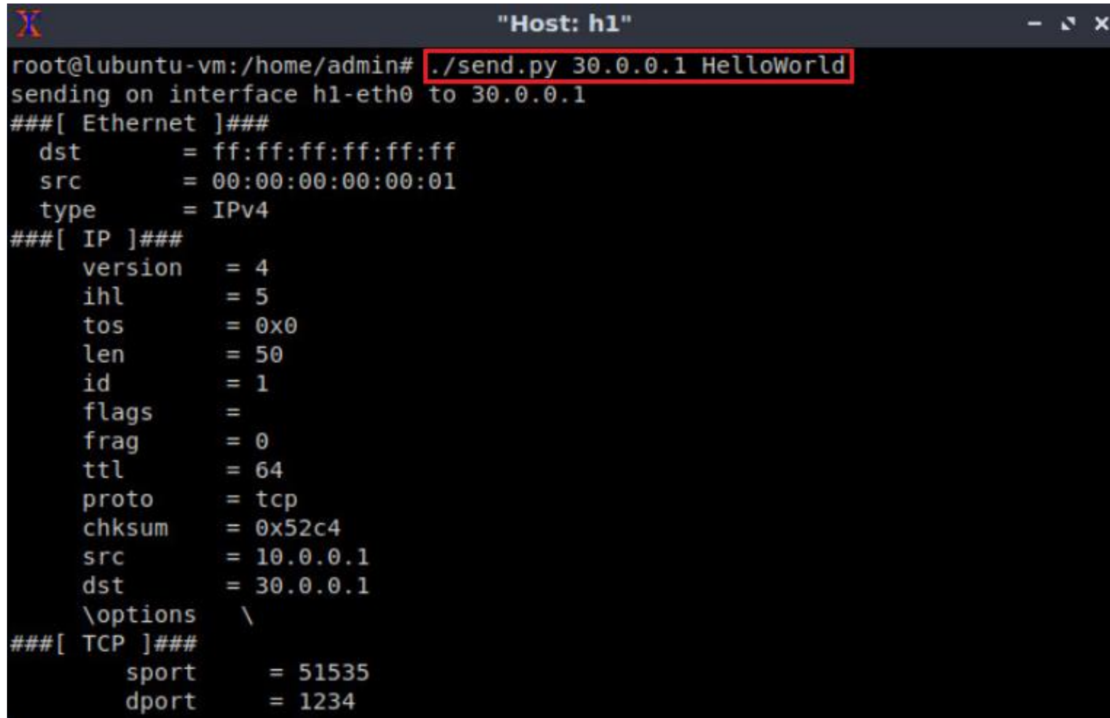


Figure 29. Sending a test packet from host h1 to host h3.

Step 7. Inspect the logs on switch s1 terminal.

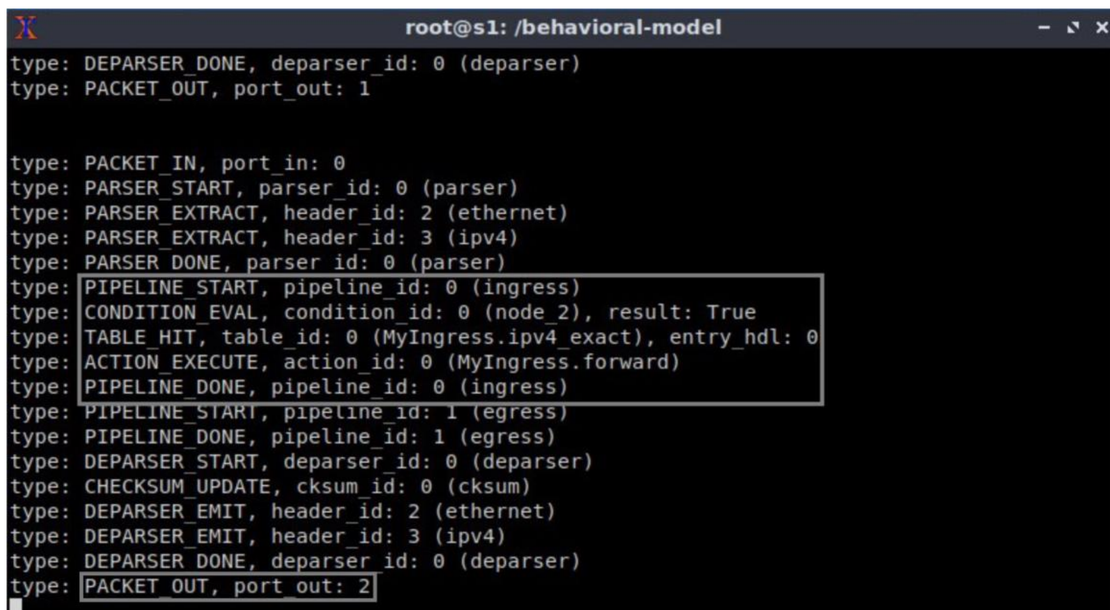


Figure 30. Inspecting the logs in switch s1.

The figure above shows that there is a hit in the `ipv4_exact` table. Then, the packet is forwarded through port 2, which is connected to host h3.

This concludes lab 6. Stop the emulation and then exit out of MiniEdit.

References

1. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.
2. Mininet walkthrough. [Online]. Available: http://Mininet.org.
3. M. Peuster, J. Kampmeyer, H. Karl. "*Containernet 2.0: A rapid prototyping platform for hybrid service function chains.*" 4th IEEE Conference on Network Softwarization and Workshops (NetSoft). 2018.
4. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
5. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Exercise 4: Implementing NAT using Match-Action Tables

Document Version: 01-14-2022

Award 2118311
“Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks”

Contents

1 Exercise description 3
1.1 Credentials 3
2 Setting the environment 4
3 Deliverables 5

1 Exercise description

In this exercise, you will implement match-action tables that perform IP address translation. The translation resembles the one performed with Network Address Translation (NAT).

Consider the figure below. The P4 switch s1 modifies the source IP address of a packet coming from host h1. On the other hand, if the packet is coming from host h2, switch s1 modifies the destination IP address.

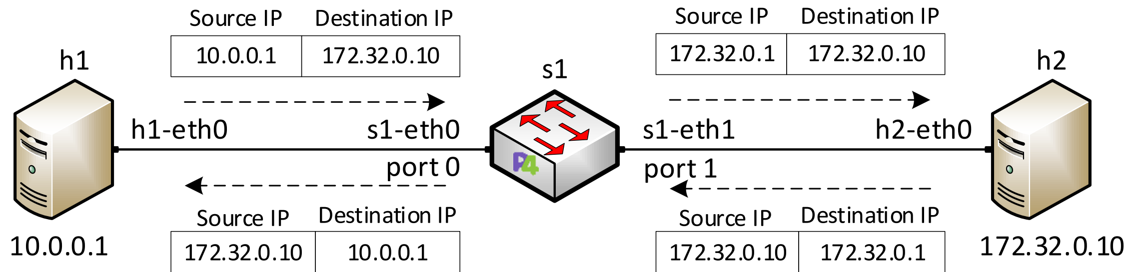


Figure 1. Lab topology.

Implement the table `translate_address` for this exercise: The key and actions of the table are as follows:

- Matches on the destination IP of the packet using `exact` matching.
- Invokes an action `change_source` that modifies the source IP address of a packet coming from host h1.
- Invokes an action `change_destination` that modifies the destination IP address of a packet coming from host h2.

Implement another table called `forwarding` with the following key and actions.

- Matches on the destination IP using `exact` matching.
- Invokes the `forward` action to forward a packet.

1.1 Credentials

The information in Table 1 provides the credentials to access the Client's virtual machine.

Table 1. Credentials to access the Client's virtual machine.

Device	Account	Password
Client	admin	password

2 Setting the environment

Follow the steps below to set the exercise's environment.

Step 1. Open MiniEdit by double-clicking the shortcut on the desktop. If a password is required type `password`.

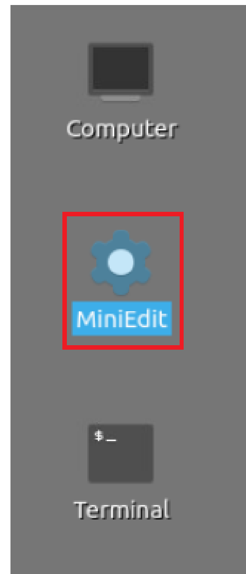


Figure 2. MiniEdit shortcut.

Step 2. Load the topology located at */home/admin/P4_Exercises/Exercise4/*.

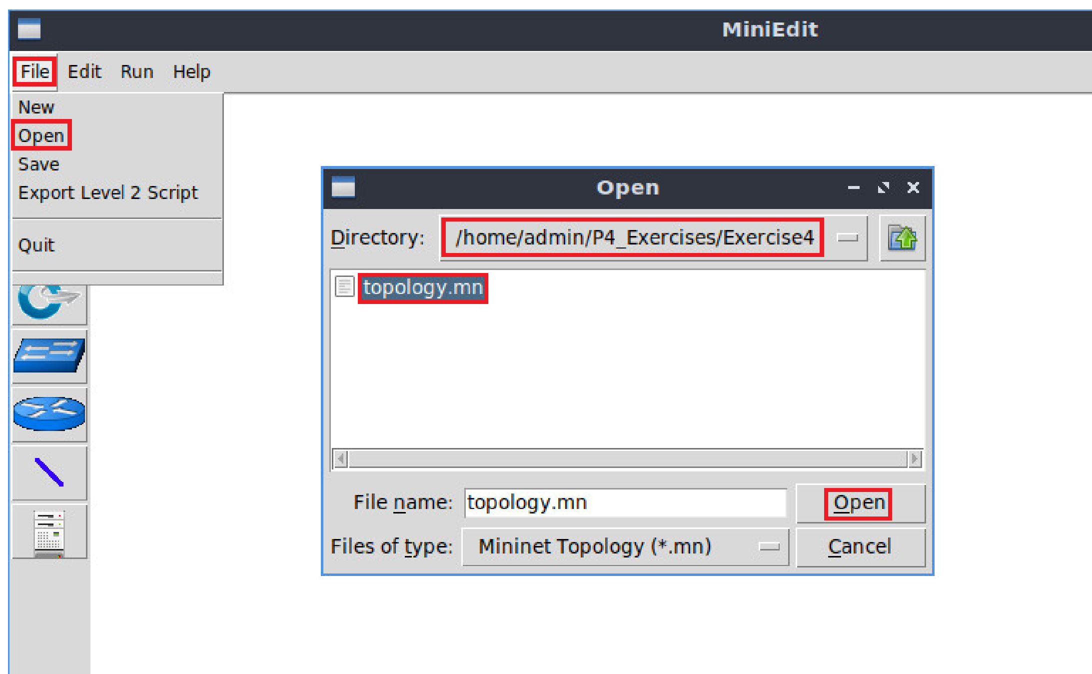


Figure 3. Opening exercise's topology.

Step 3. Run the emulation by clicking on the button located on the lower left-hand side.

Figure 4. Running the emulation.

Step 4. In the terminal, type the command below. This command launches the Visual Studio Code and opens the directory where the P4 program for this exercise is located.

```
code P4_Exercises/Exercise4/
```

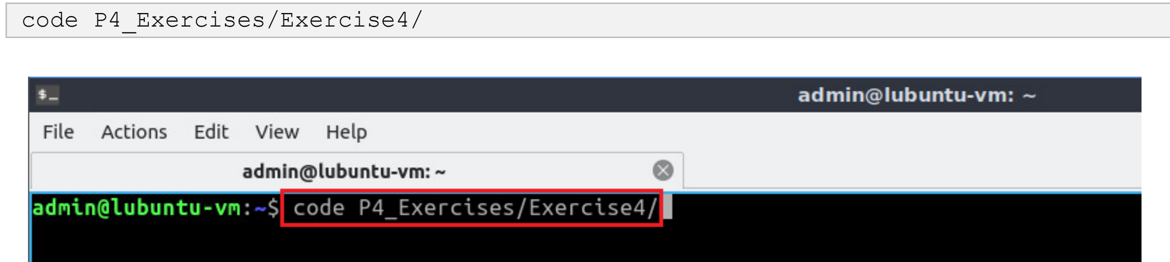


Figure 5. Opening the working directory.

3 Deliverables

Follow the steps below to complete the exercise.

a) Implement the table `translate_address` with the following actions: `change_source` and `change_destination`.

b) Implement the table `forwarding`.

c) Compile the *basic.p4* in the Visual Studio Code. Push the output file of the compiler to the switch s1.

d) Start the switch daemon, then push the table entries to the switches. The file *rules.cmd* is in the directory ~/exercise4/.

e) From host h1, send a packet using the *send.py* program. Verify which table this packet is hitting by inspecting the logs of the switch using the `nanomsg` tool.

f) Similarly, from host h2 send a packet using the script *send.py* and verify which table is the packet hitting.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 7: Populating and Managing Match-action Tables at Runtime

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 3
1.1 Runtime 3
2 Lab topology..... 4
2.1 Starting end hosts 6
3 Navigating the switch's CLI 7
3.1 Loading the programming environment..... 8
3.2 Compiling and loading the P4 program to switch s1 8
3.3 Verifying the configuration 10
4 Configuring switch s1..... 11
4.1 Navigating the switch's CLI..... 13
4.2 Displaying ports, tables, and actions..... 14
5 Populating match-action tables using the switch's CLI 16
5.1 Displaying the table's basic information..... 16
5.2 Manipulating a match-action table with exact lookup 17
5.3 Manipulating a match-action table with LPM lookup..... 20
References 21

Overview

This lab describes how to populate and manage match-action tables at runtime. It then explains a tool (simple_switch_CLI) that is used with the software switch (BMv2) to manage the tables.

Objectives

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

1. Understand how to populate match-action tables.
2. Describe the basic syntax of the simple_switch_CLI tool.
3. Verify the insertion of rules in the tables.

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.
2. Section 2: Lab topology.
3. Section 3: Navigating the switch's CLI.
4. Section 4: Configuring switch s1.
5. Section 5: Populating match-action tables using the switch's CLI.

1 Introduction

1.1 Runtime

Once a P4 program is compiled into a target-specific configuration, the output is loaded into the data plane of the device. Then, the behavior of the P4 target can be managed at runtime by the control plane via data plane Application Programming Interface (APIs).

Runtime operations include inserting, updating, and deleting entries in P4 tables as well as controlling other entities of the program such as counters, meters, etc.

Runtime APIs can be divided into program-dependent and program-independent APIs. Program-dependent APIs comprise functions whose names are derived from the P4 program itself. Thus, any changes to the P4 program would modify the names and the definitions of the APIs' functions. Program-independent APIs comprise a set of fixed functions that are independent of the P4 program. Therefore, changes in the P4 programs do not affect those APIs.

The control plane that manages the data plane tables and externs can be remote or local on the device. Remote control planes invoke API calls through Remote Procedure Calls (RPCs) while relying on asynchronous message frameworks such as Thrift[1] and gRPC[2]. Such frameworks use protocol buffers (protobuf) to define service API and message, and HTTP/2.0 and TLS for transport. On the other hand, a local control plane runs on the Central Processing Unit (CPU) of the device and invokes API calls locally. It is implemented by the driver of the device.

Figure 1 shows the runtime environment used in this lab series to control the P4 target (BMv2). The control plane uses the `simple_switch_CLI` tool to interact with the data plane. The `simple_switch_CLI` includes a program-independent CLI and a Thrift client which connects to the program-independent control server residing on the BMv2 switch.

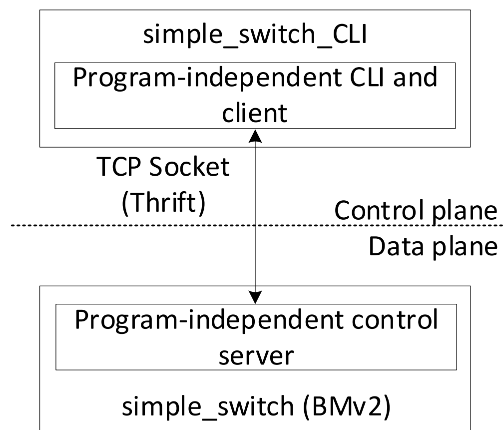


Figure 1. Runtime management of a P4 target (BMv2).

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology comprises three end hosts and one P4 programmable switch.

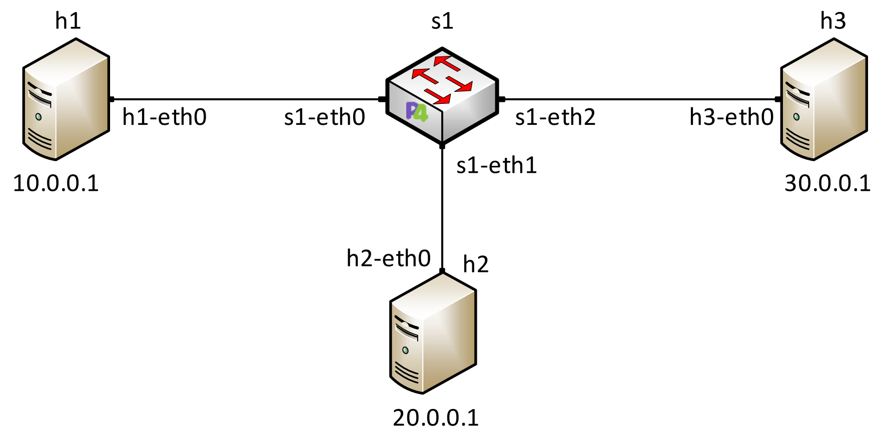


Figure 2. Lab topology.

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 3. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab7* folder and search for the topology file called *lab7.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

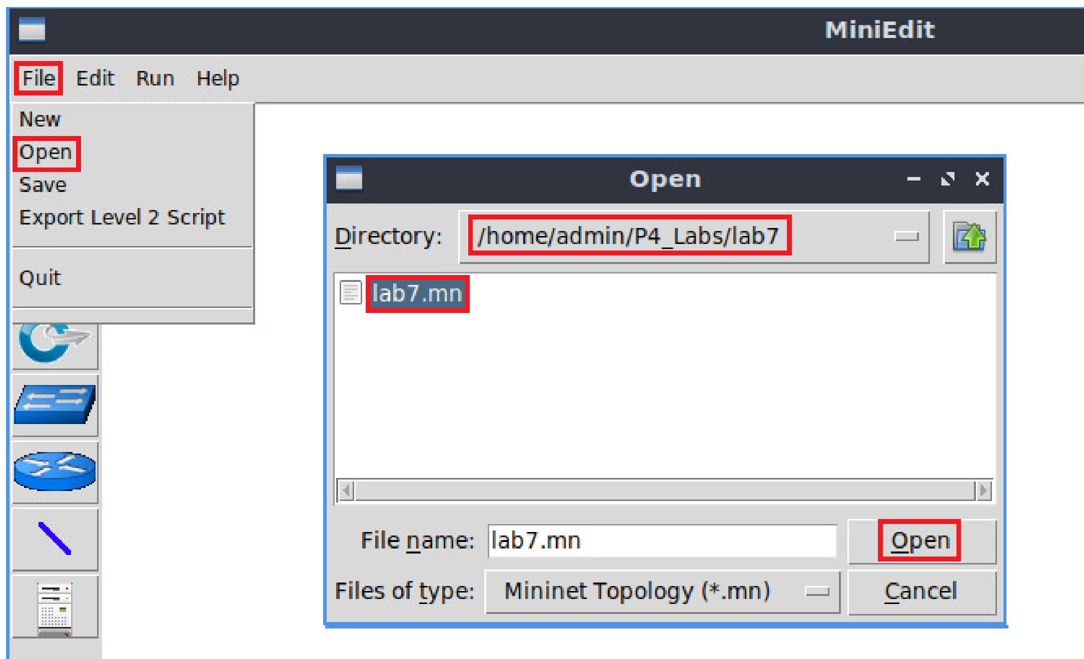


Figure 4. MiniEdit's *Open* dialog.

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 5. Running the emulation.

2.1 Starting end hosts

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.

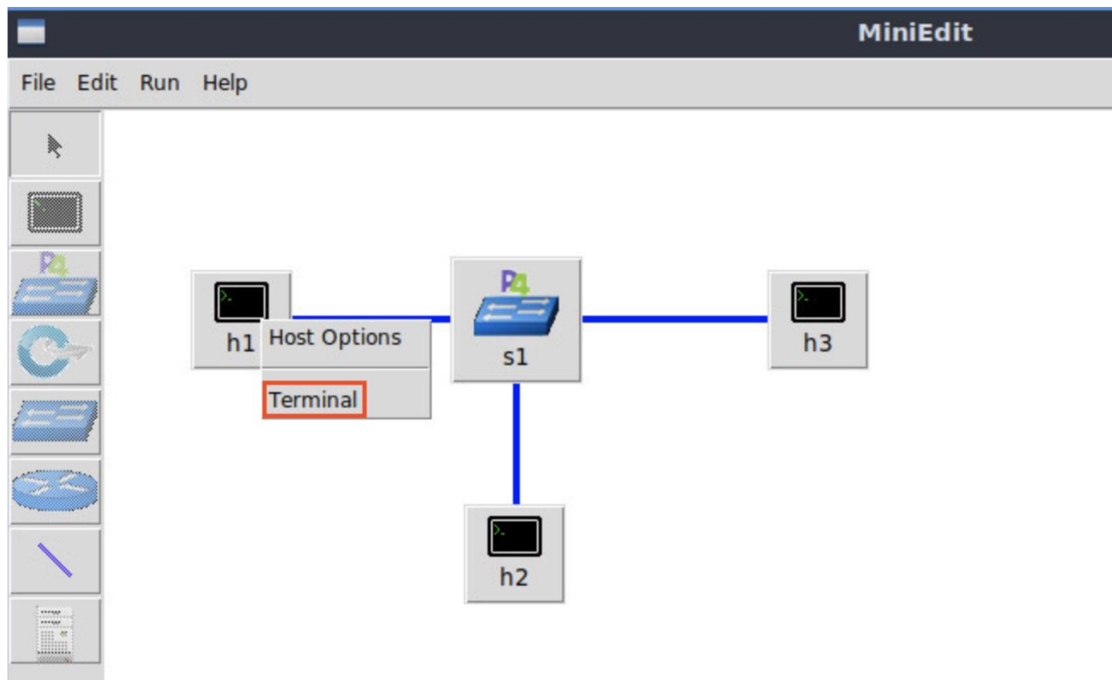


Figure 6. Opening a terminal on host h1.

Step 2. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin#
```

Figure 7. Verifying the configuration host h1 interfaces.

3 Navigating the switch's CLI

This section demonstrates how to navigate the switch's CLI using the `simple_switch_CLI` tool. This tool is used to manage P4 objects at runtime. This tool

works with the BMv2 software switch. Other targets have their own tools (e.g., Intel Tofino targets use the Barefoot Runtime).

3.1 Loading the programming environment

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

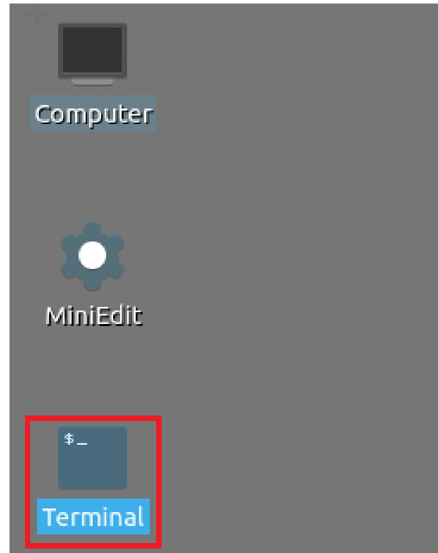


Figure 8. Shortcut to open a Linux terminal.

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.

```
code ~/P4_Labs/lab7
```

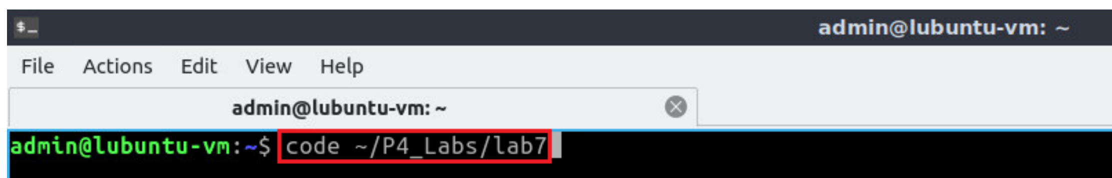


Figure 9. Launching the editor and opening the lab7 directory.

3.2 Compiling and loading the P4 program to switch s1

Step 1. We will not modify the P4 source code. The P4 program is already written. Issue the following command in the terminal panel inside the VS Code to compile the program.

```
p4c basic.p4
```

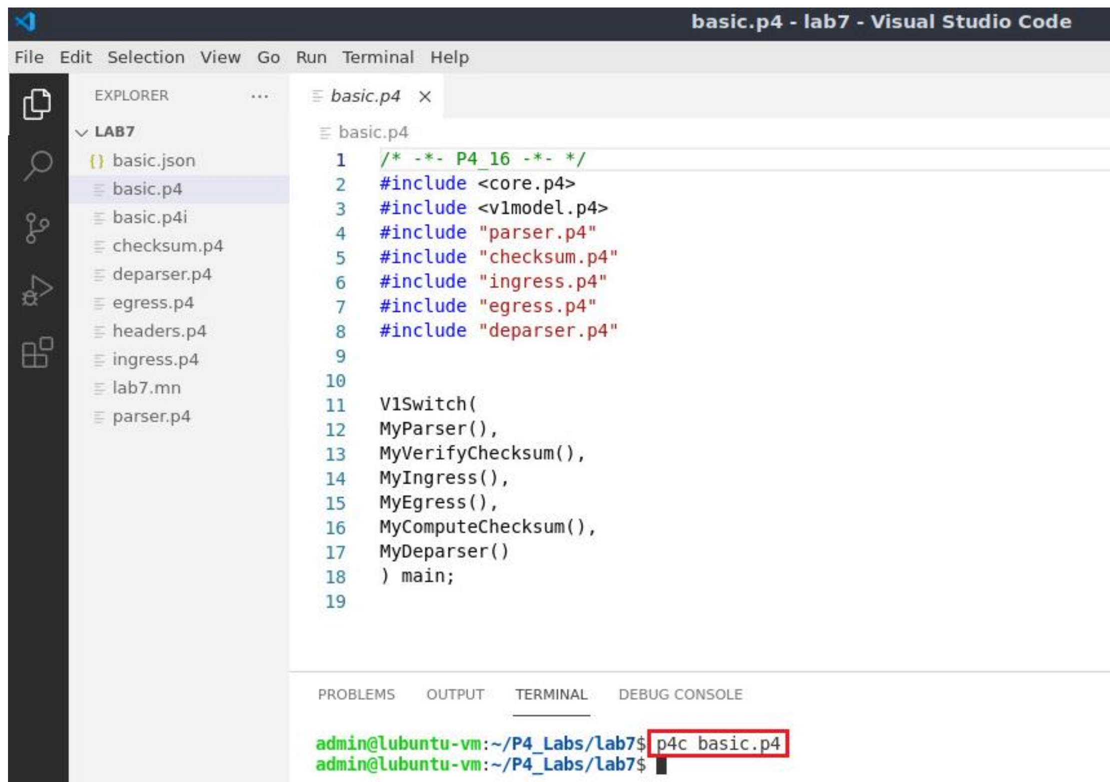


Figure 10. Compiling a P4 program.

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 11. Pushing the *basic.json* file to switch s1.

3.3 Verifying the configuration

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

Figure 12. Maximizing the MiniEdit window.

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

Figure 13. 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
```

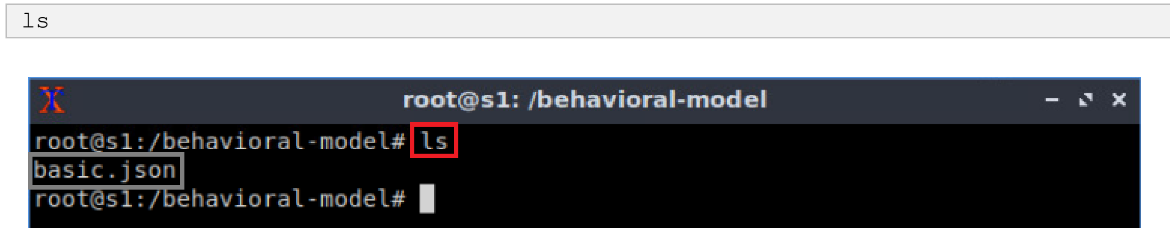


Figure 14. Displaying the contents of the current directory in the switch s1.

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

4 Configuring switch s1

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:27 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:3265 (3.2 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:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

s1-eth0   Link encap:Ethernet  HWaddr 0e:7e:48:32:53:a3
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:356 (356.0 B)  TX bytes:0 (0.0 B)

s1-eth1   Link encap:Ethernet  HWaddr 9e:c5:42:78:07:16
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)

s1-eth2   Link encap:Ethernet  HWaddr 26:15:f3:b2:b1:d4
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)
```

Figure 15. Displaying switch s1 interfaces.

The output displays switch s1 interfaces (i.e., *s1-eth0*, *s1-eth1* and *s1-eth2)*. The interface *s1-eth0* on the switch s1 connects to the host h1. The interface *s1-eth1* on the switch s1 connects to the host h2 and *s2-eth2* is connected to host h3.

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

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --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 -i 2@s1-eth2 --nanolog
 ipc:///tmp/bm-log.ipc basic.json &
[1] 39
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 16. 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.

4.1 Navigating the switch's CLI

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

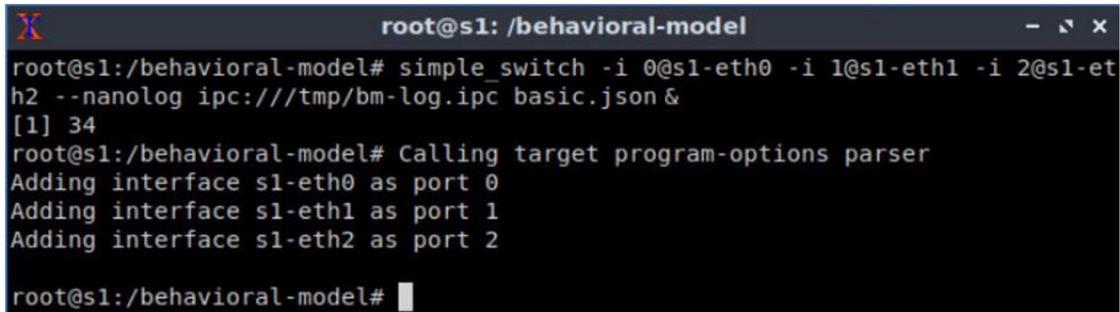


Figure 17. Returning to the CLI.

Step 2. Start switch s1 CLI tool by typing the following command.

```
simple_switch_CLI
```

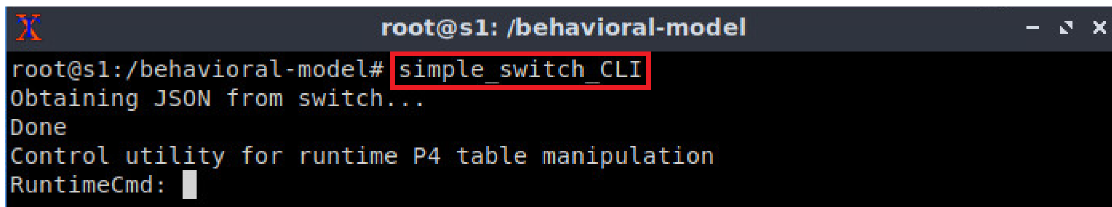


Figure 18. Starting the `simple_switch_CLI` runtime program.

Step 3. Type a question mark (`?`) to see the available commands in the tool.

```
?
```

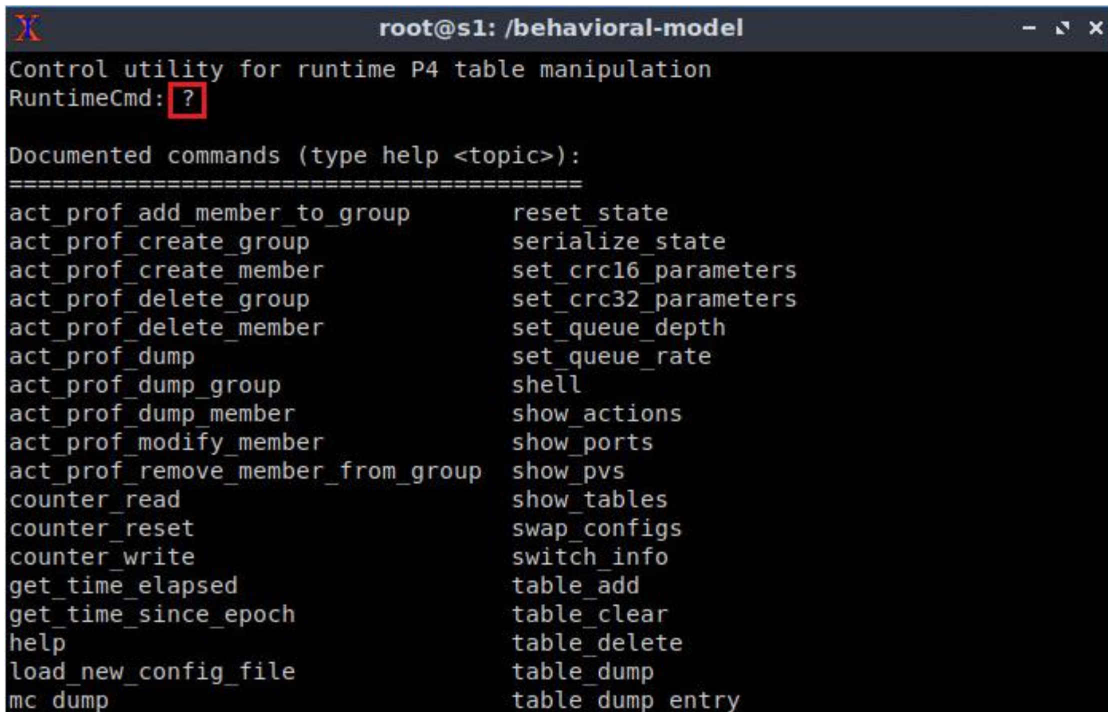


Figure 19. Displaying the available commands in the `simple_switch_CLI`.

Use the `TAB` key to autocomplete a command.

Step 4. To get help on a specific command, type `help <topic>`, where `<topic>` is the command that you would like to explore. For example, to see the syntax of table_dump, type the following.

```
help table_dump
```

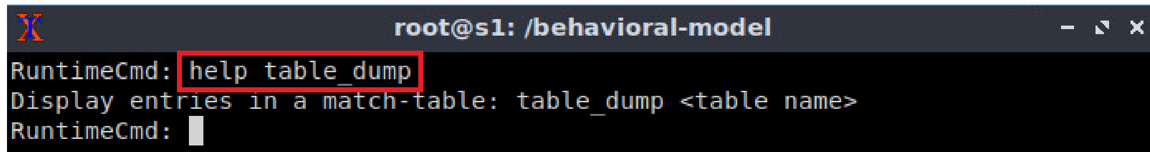


Figure 20. Displaying the syntax of the `table_dump` command.

4.2 Displaying ports, tables, and actions

Step 1. To display the list of ports in our switch and their mapping to Linux interface names, type the following command.

```
show_ports
```

```
root@s1: /behavioral-model
RuntimeCmd: show_ports
  port #         iface name       status   extra info
=====================================================
    0             s1-eth0           UP
    1             s1-eth1           UP
    2             s1-eth2           UP
RuntimeCmd:
```

Figure 21. Displaying the ports used by switch s1.

Step 2. To display the list of tables defined in the P4 program, type the following command.

```
show_tables
```

```
root@s1: /behavioral-model
RuntimeCmd: show_tables
MyIngress.ipv4_host            [implementation=None, mk=ipv4.dstAddr(exact, 32)]
MyIngress.ipv4_lpm             [implementation=None, mk=ipv4.dstAddr(lpm, 32)]
RuntimeCmd:
```

Figure 22. Showing the tables defined in the ingress block.

Step 3. List the actions defined in the P4 program by issuing the command below.

```
show_actions
```

```
root@s1: /behavioral-model
RuntimeCmd: show_actions
MyIngress.drop                 []
MyIngress.forward              [dstAddr(48),    port(9)]
RuntimeCmd:
```

Figure 23. Showing the actions defined in the P4 program.

Notice that the `MyIngress.drop` action does not have any action data whereas the action `MyIngress.forward` modifies the destination MAC address (i.e., `dstAddr(48)`) and the egress port (i.e., `port(9)`).

Step 4. To display basic information about the switch, type the following command.

```
switch_info
```

```
root@s1: /behavioral-model
RuntimeCmd: switch_info
device_id                  : 0
thrift_port                : 9090
notifications_socket       : ipc:///tmp/bmv2-0-notifications.ipc
elogger_socket             : ipc:///tmp/bm-log.ipc
debugger_socket            : None
RuntimeCmd:
```

Figure 24. Displaying switch's information.

Step 5. To display the time since the switch was turned on, type the following command.

```
get_time_elapsed
```

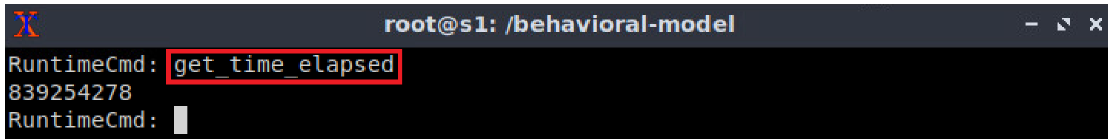


Figure 25. Showing the uptime of switch s1.

The command above displays the time since the switch was turned on in microseconds.

5 Populating match-action tables using the switch's CLI

This section demonstrates how to manage and populate the tables using the switch CLI tool.

5.1 Displaying the table's basic information

Step 1. To display information about a table in the P4 program, type the following command.

```
table_info MyIngress.ipv4_host
```

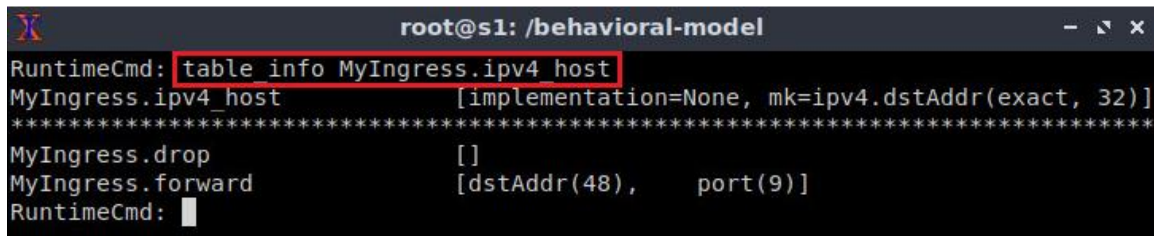


Figure 26. Showing information about the table `MyIngress.ipv4_host`.

Step 2. Issue the following command to display the actions corresponding to a table. The output shows the actions contained in the table `MyIngress.ipv4_host`.

```
table_show_actions MyIngress.ipv4_host
```

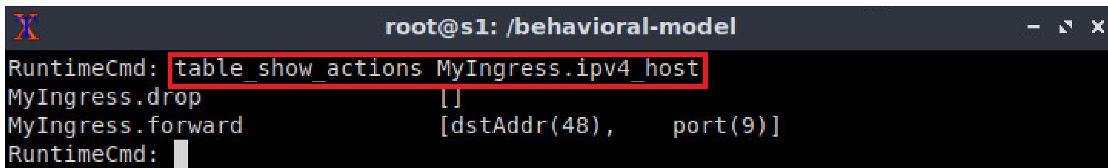


Figure 27. Displaying the actions associated with table `MyIngress.ipv4_host`.

Step 3. Dump the entries of `MyIngress.ipv4_host` table by issuing the following command.

```
table_dump MyIngress.ipv4_host
```

```
root@s1: /behavioral-model
RuntimeCmd: table_dump MyIngress.ipv4_host
==========
TABLE ENTRIES
==========
Dumping default entry
Action entry: MyIngress.drop -
==========
RuntimeCmd:
```

Figure 28. Showing the entries of the table `MyIngress.ipv4_host`.

The output above shows that the table has not been populated.

Step 4. Display the number of rules/entries in the `MyIngress.ipv4_host` table by issuing the following command. The output will show that there are no entries added so far.

```
table_num_entries MyIngress.ipv4_host
```

```
root@s1: /behavioral-model
RuntimeCmd: table_num_entries MyIngress.ipv4_host
0
RuntimeCmd:
```

Figure 29. Displaying the number of entries in the table `MyIngress.ipv4_host`.

5.2 Manipulating a match-action table with exact lookup

Step 1. Issue the following command to display the syntax of `table_add`.

```
help table_add
```

```
root@s1: /behavioral-model
RuntimeCmd: help table_add
Add entry to a match table: table_add <table name> <action name> <match fields>
=> <action parameters> [priority]
RuntimeCmd:
```

Figure 30. Showing the syntax of `table_add`.

The parameters of the `table_add` can be summarized as follows:

- <table_name>: name of the P4 table that we would like to add rules to. The list of tables can be displayed using the `show_tables` command.
- <action name>: the action associated with the entry.
- <match fields>: the key used to match against the incoming packet.
- <action parameter>: the parameter associated with the entry.
- `[priority]`: the priority of the entry.

Step 2. Add an entry/rule to the table `MyIngress.ipv4_host` by issuing the following command.

```
table_add MyIngress.ipv4_host MyIngress.forward 30.0.0.1 => 00:00:00:00:00:03 2
```

```
root@s1: /behavioral-model
RuntimeCmd: table_add MyIngress.ipv4_host MyIngress.forward 30.0.0.1 => 00:00
:00:00:00:03 2
Adding entry to exact match table MyIngress.ipv4_host
match key:           EXACT-1e:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 0
RuntimeCmd:
```

Figure 31. Adding an entry to the table `MyIngress.ipv4_host`.

The output shows the details of the new table entry. The match key is 0x1e:00:00:01 (i.e., the hexadecimal value of the IP address 30.0.0.1) and the lookup mechanism is exact. The action executed when this entry is hit will be the one defined in `MyIngress.forward`. The action data associated with the entry is the MAC address of the destination host (i.e., 00:00:00:00:00:03) and the egress port (i.e., 00:02).

Step 3. Issue the following command to show the entries in the table `MyIngress.ipv4_host`.

```
table_dump MyIngress.ipv4_host
```

```
root@s1: /behavioral-model
RuntimeCmd: table_dump MyIngress.ipv4_host
==========
TABLE ENTRIES
**********
Dumping entry 0x0
Match key:
* ipv4.dstAddr        : EXACT     1e000001
Action entry: MyIngress.forward - 03, 02
==========
Dumping default entry
Action entry: MyIngress.drop -
==========
RuntimeCmd:
```

Figure 32. Showing the entries of the table `MyIngress.ipv4_host`.

Step 4. Display the number of entries in the table `MyIngress.ipv4_host` by typing the following command.

```
table_num_entries MyIngress.ipv4_host
```

```
root@s1: /behavioral-model
RuntimeCmd: table_num_entries MyIngress.ipv4_host
1
RuntimeCmd:
```

Figure 33. Displaying the number of entries in the table `MyIngress.ipv4_host`.

Step 5. We can also display the entry in a table by using its match key as follows.

```
table_dump_entry_from_key MyIngress.ipv4_host 0x1e000001
```

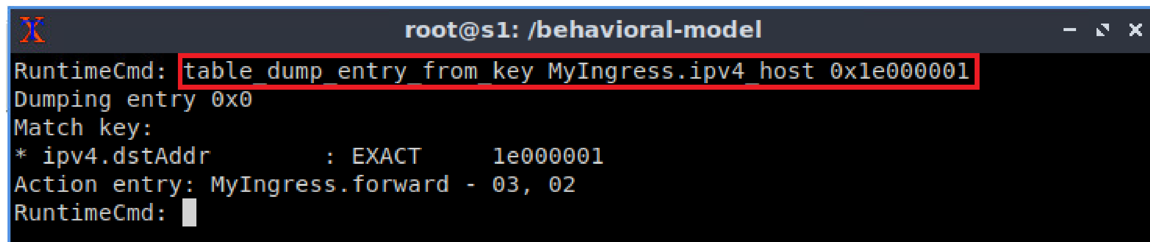


Figure 34. Dumping a table entry by specifying the key.

Match-action tables can contain too many entries and dumping the whole table will produce a large output that is hard to read.

Step 6. Another way to display the entry in a table is by specifying the entry handle, which in this case is 0. Issue the following command to show the table entry using the handle of the entry.

```
table_dump_entry MyIngress.ipv4_host 0
```

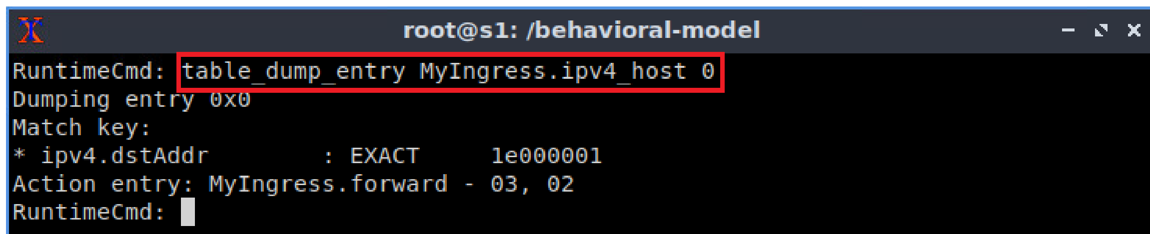


Figure 35. Dumping a table entry by specifying the handle.

Step 7. Issue the following command to modify an existing entry.

```
table_modify MyIngress.ipv4_host MyIngress.forward 0 00:00:00:00:00:05 5
```

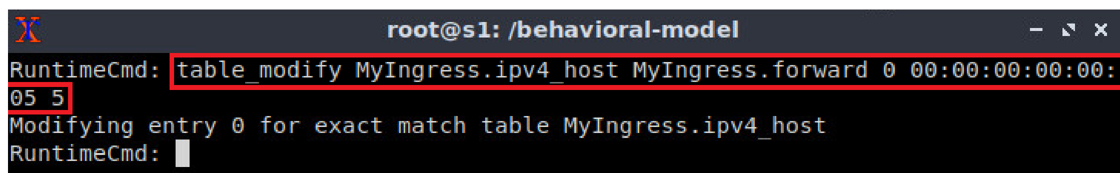


Figure 36. Modifying a table's entry.

The parameters of `table_modify` are described as follows:

- `MyIngress.ipv4_host`: refers to the table that implements an exact lookup.
- `MyIngress.forward`: specifies the action.
- `0`: the table's entry handle.
- `00:00:00:00:00:05`: the new MAC address.

- `5`: the new egress port.

Step 8. Dump the content of the table `MyIngress.ipv4_host` by typing the following command.

```
table_dump_entry MyIngress.ipv4_host 0
```

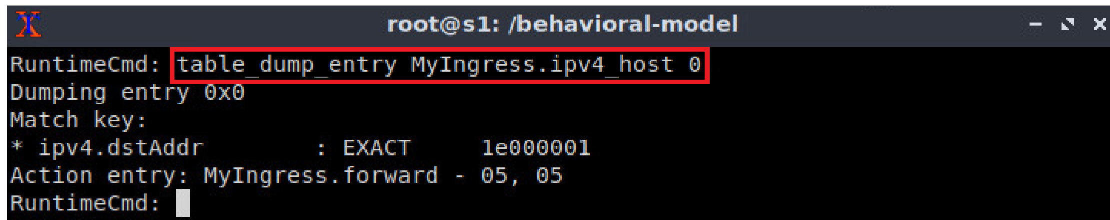


Figure 37. Dumping a table entry by specifying the handle.

The output shows the details of the modified entry. The match key is 0x1e:00:00:01 (i.e., the hexadecimal value of the IP address 30.0.0.1) and the lookup mechanism is exact. The action executed when this entry is hit will be the one defined in `MyIngress.forward`. The action data specifies 5 (i.e., the hexadecimal value 00:00:00:00:00:05) as the new destination MAC address and port 5 as the egress.

5.3 Manipulating a match-action table with LPM lookup

Step 1. Add an entry/rule to the `MyIngress.ipv4_lpm` table by issuing the following command.

```
table_add MyIngress.ipv4_lpm MyIngress.forward 20.0.0.0/8 => 00:00:00:00:00:02
1
```

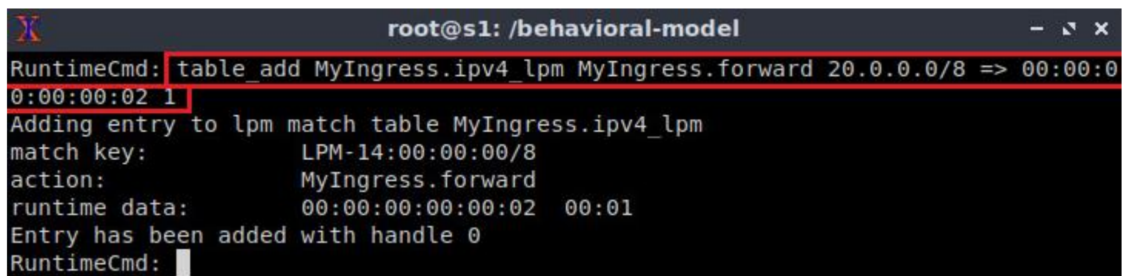


Figure 38. Adding an entry to the table `MyIngress.ipv4_lpm`.

The parameters of `table_add` are described as follows:

- `MyIngress.ipv4_lpm`: refers to the table that implements LPM lookup.
- `MyIngress.forward`: specifies the action.
- `20.0.0.0/8`: is entry's key.
- `00:00:00:00:00:02`: the destination MAC address.
- `1`: specifies the egress port.

Step 2. To delete a specific entry in a P4 table, type the following command.

```
table_delete MyIngress.ipv4_lpm 0
```

```
root@s1: /behavioral-model
RuntimeCmd: table_delete MyIngress.ipv4_lpm 0
Deleting entry 0 from MyIngress.ipv4_lpm
RuntimeCmd:
```

Figure 39. Removing an entry from the table `MyIngress.ipv4_lpm`.

This command deletes the entry with the handle 0 in the `MyIngress.ipv4_lpm` table.

Step 3. It is also possible to delete all entries from a match action table by issuing the following command.

```
table_clear MyIngress.ipv4_lpm
```

```
root@s1: /behavioral-model
RuntimeCmd: table_clear MyIngress.ipv4_lpm
RuntimeCmd:
```

Figure 40. Removing all the entries from the table `MyIngress.ipv4_lpm`.

Step 4. Verify that the table `MyIngress.ipv4_lpm` is cleared by issuing the following command. The output will show that the table `MyIngress.ipv4_lpm` is empty.

```
table_num_entries MyIngress.ipv4_lpm
```

```
root@s1: /behavioral-model
RuntimeCmd: table_num_entries MyIngress.ipv4_lpm
0
RuntimeCmd:
```

Figure 41. Displaying the number of entries in the table `MyIngress.ipv4_lpm`.

This concludes lab 7. Stop the emulation and then exit out of MiniEdit.

References

1. Apache. "*Apache Thrift.*" [Online]. Available: https://thrift.apache.org/.
2. Google. "*gRPC.*" [Online]. Available: https://grpc.io/.
3. Google. "*Protocol Buffers.*" [Online]. Available: https://developers.google.com/protocol-buffers.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Exercise 5: Configuring Match-action Tables at Runtime

Document Version: 01-14-2022

Award 2118311
“Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks”

Contents

1 Exercise description ... 3
1.1 Credentials ... 3
2 Setting the environment ... 3
3 Deliverables ... 5

1 Exercise description

In this exercise, you will populate and manage the tables of the P4 switches by using the runtime interface.

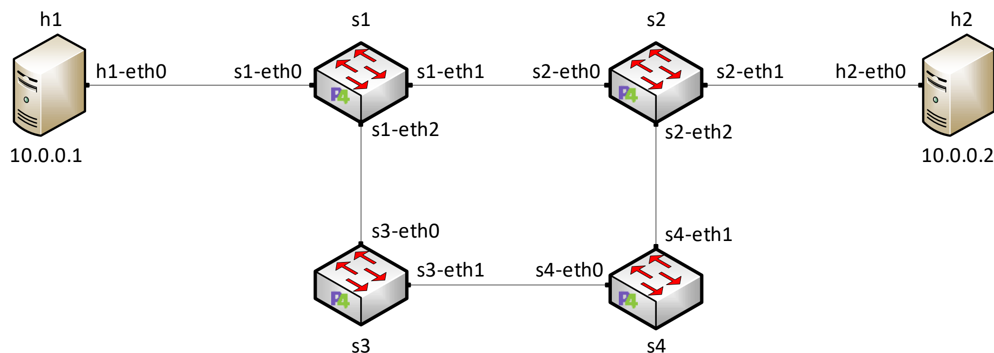


Figure 1. Lab topology.

1.1 Credentials

The information in Table 1 provides the credentials to access the Client's virtual machine.

Table 1. Credentials to access the Client's virtual machine.

Device	Account	Password
Client	admin	password

2 Setting the environment

Follow the steps below to set the exercise's environment.

Step 1. Open MiniEdit by double-clicking the shortcut on the desktop. If a password is required type `password`.

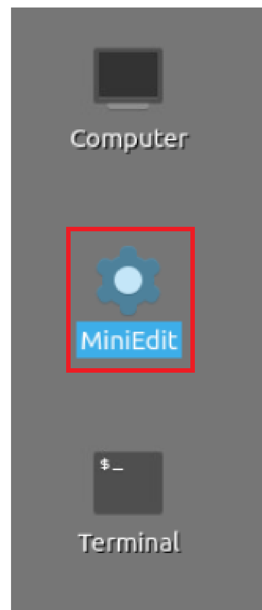


Figure 2. MiniEdit shortcut.

Step 2. Load the topology located at */home/admin/P4_Exercises/Exercise5/*.

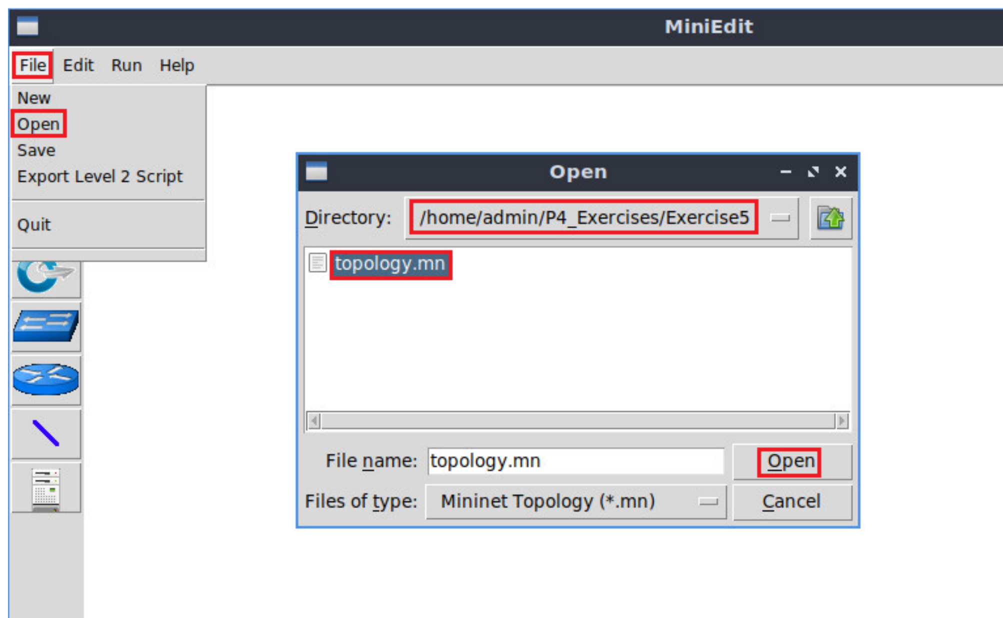


Figure 3. Opening the exercise's topology.

Step 3. Run the emulation by clicking on the button located on the lower left-hand side.

Figure 4. Running the emulation.

Step 4. In the terminal, type the command below. This command launches the Visual Studio Code and opens the directory where the P4 program for this exercise is located.

```
code P4_Exercises/Exercise5/
```

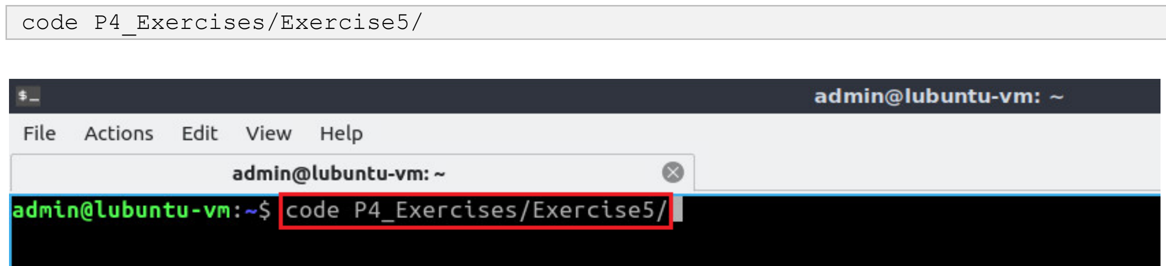


Figure 5. Opening the working directory.

3 Deliverables

Follow the steps below to complete the exercise.

a) Compile the *basic.p4* in the Visual Studio Code. Push the output file of the compiler to all switches s1, s2, s3, and s4.

b) Start the daemon on all switches and associate the interfaces to their corresponding ports and enable `nanolog`.

c) In switch s1, use the CLI to determine the table names on the switches. Then, list the actions associated with the tables. What are the actions and what parameters do they accept?

d) Push the table entries to the switches so that a packet sent from h1 to h2 traverses switches s1-s2.

e) Initiate the *nanomsg_client.py* program in switches s1 and s2.

f) Modify the path so that the packet traverses the switches s1-s3-s4-s2.

g) Delete all the rules on the switches. Write the rules that create a loop in the switches s1-s2-s4-s3-s1-s2-s4-s2… Verify that the packet is being transmitted infinitely between the switches. Suggest a solution for breaking the loop.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 8: Checksum Recalculation and Packet Deparsing

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 checksums 4
1.1 Checksums in P4 4
1.2 Deparsing 5
2 Lab topology 6
2.1 Starting the end hosts 7
3 Implementing checksum calculation in P4 8
3.1 Loading the programming environment 8
3.2 Inspecting the P4 code 9
4 Loading the P4 program 10
4.1 Compiling and loading the P4 program to switch s1 10
4.2 Verifying the configuration 11
5 Configuring switch s1 12
5.1 Mapping P4 program's ports 12
5.2 Loading the rules to the switch 14
6 Manipulating the checksum and deparser 14
6.1 Sending a packet without checksum update 14
6.2 Sending a packet with checksum update 16
6.3 Updating the deparser in the P4 code 21
References 25

Overview

This lab describes how to recompute the checksum of a header. Recomputing the checksum is necessary if the packet header was modified by the P4 program. The lab also describes how a P4 program performs deparsing to emit headers.

Objectives

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

1. Understand checksums and the need to recompute them if the header was modified.
2. Implement checksum update in the P4 program.
3. Validate the checksum of a header.
4. Understand deparsing and how to implement a deparser in P4.

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 checksums.
2. Section 2: Lab topology.
3. Section 3: Implementing checksum calculation in P4.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Manipulating the checksum and deparser.

1 Introduction to checksums

Several protocols use checksums to validate the integrity of the packet headers. A checksum is a small value derived from another data block, often through a checksum algorithm such as the Cyclic Redundancy Check (CRC). The checksum calculation and verification start with the sender calculating the checksum of the data before transmitting the packet. Then, the checksum value is inserted into the packet header. Upon receiving the packet, the receiver computes the checksum of the received packet using the same algorithm as the one used by the sender. If the calculated checksum value matches the one in the packet header, the packet is verified; otherwise, a transmission error has occurred. Incorrect checksums typically lead to dropping the packet by the switch.

1.1 Checksums in P4

In a P4 program, the developer may change the packet headers. For example, if the program is implementing a routing function, then header fields such as the Time-to-live (TTL) must be modified. Any change to the header fields will cause the checksum value to change. Therefore, it is necessary to recompute the checksum in the P4 program in case modifications are made to the header fields.

Figure 1 shows an example of computing the checksum in a P4 program.

```
 1:  /*********************CHECKSUM COMPUTATION*********************/
 2:  control MyComputeChecksum(inout header hdr, inout metadata meta){
 3:     apply{
 4:         update_checksum(
 5:             hdr.ipv4.isValid(),
 6:             { hdr.ipv4.version,
 7:               hdr.ipv4.ihl,
 8:               hdr.ipv4.diffserv,
 9:               hdr.ipv4.totalLen,
10:               hdr.ipv4.identification,
11:               hdr.ipv4.flags,
12:               hdr.ipv4.fragOffset,
13:               hdr.ipv4.ttl,
14:               hdr.ipv4.protocol,
15:               hdr.ipv4.srcAddr,
16:               hdr.ipv4.dstAddr },
17:             hdr.ipv4.hdrChecksum,
18:             HashAlgorithm.csum16);
19:     }
20:  }
```

Figure 1. Updating the checksum of IPv4 header.

The syntax for updating the checksum in P4 (V1Model) is as follows:

```
update_checksum(condition, data, checksum_output, algorithm)
```

- `condition`: a condition that is evaluated before updating the checksum. If the condition is true, the checksum is updated. Otherwise, the checksum remains as

it is in the packet. Here we often check if the header is valid (i.e., it was parsed or set to be valid by the programmer). For example, in Figure 1, the IPv4 header is checked if valid.

- `data`: the data whose checksum is to be computed. This typically includes the header fields of the protocol which uses the checksum. The example above shows the header fields of IPv4.
- `checksum_output`: the parameter that the checksum will be written to once it has been computed. In the example above, we are writing the resulting checksum value to the `hdrChecksum` field of IPv4.
- `algorithm`: the algorithm used by the protocol to compute the checksum. For example, for IPv4, the IETF RFC 791[1] state that the checksum field is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header. This checksum is implemented in the V1Model using the `HashAlgorithm.csum16` hash function.

It is also possible in P4 to verify the checksum. The V1Model provides the checksum verification extern function `verify_checksum` which sets the `checksum_error` bit in the standard metadata in case the verification fails, causing the packet to be dropped. The syntax for the `verify_checksum` is the same as that of `update_checksum` described above.

```
verify_checksum(condition, data, checksum_output, algorithm)
```

1.2 Deparsing

The P4 program includes a deparser that specifies which headers are to be emitted. The deparser emits the headers and the payload of the original packet. Note that only the valid headers are emitted. A header is considered valid after it has been parsed in the P4 program or after the program explicitly validates the header with the function `setValid()`.

The deparser is defined as a control block and is executed after finishing the packet processing by the other control blocks. Consider Figure 2. The deparser has a `packet_out` type in its parameters. The `packet_out` type includes the `emit` method which accepts the headers to be reassembled when the deparser constructs the outgoing packet. Note that the order of emitting packets' headers is important, and the headers are only emitted in case they are valid.

```
1:  /*********************DEPARSER*********************/
2:  control MyDeparser(packet_out packet, in headers hdr){
3:      apply{
4:          packet.emit(hdr.ethernet);
5:          packet.emit(hdr.ipv4);
6:      }
7:  }
```

Figure 2. Deparser implementation.

2 Lab topology

Let's get started by loading a simple Mininet topology using MiniEdit. The topology comprises three end hosts and one P4 programmable switch.

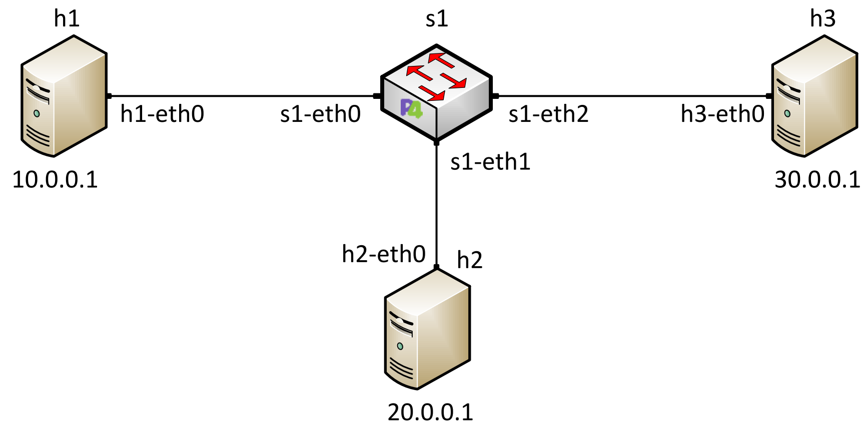


Figure 3. Lab topology.

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`.

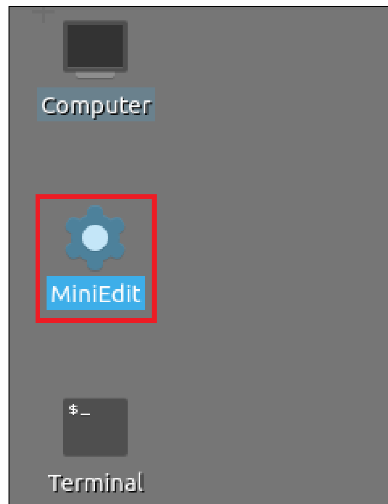


Figure 4. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab8* folder and search for the topology file called *lab8.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

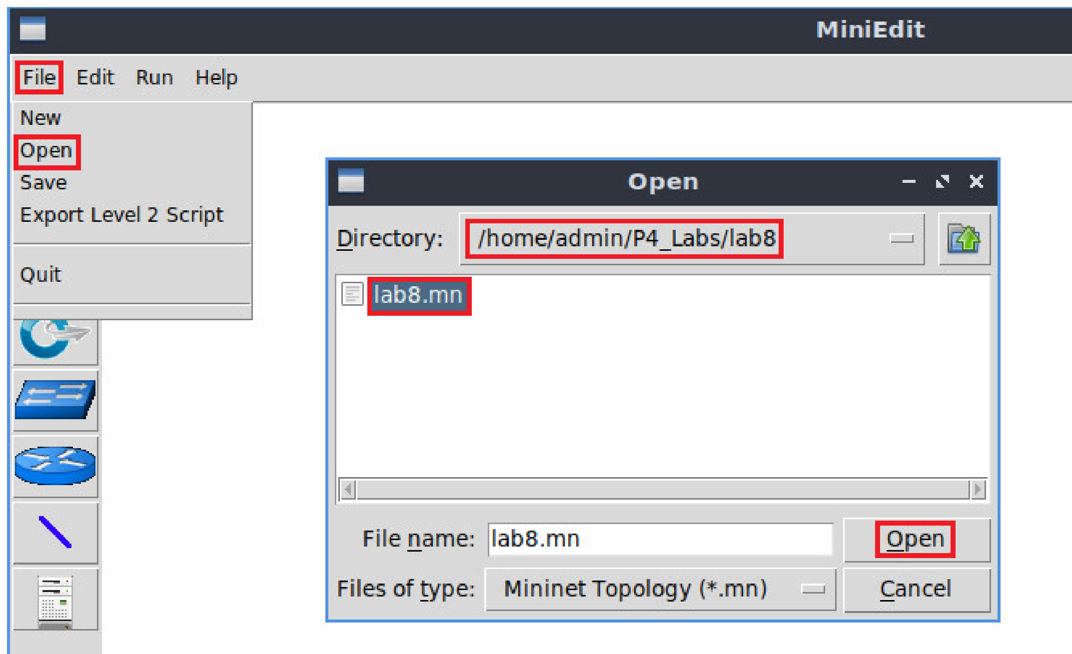


Figure 5. MiniEdit's *Open* dialog.

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 the end hosts

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.

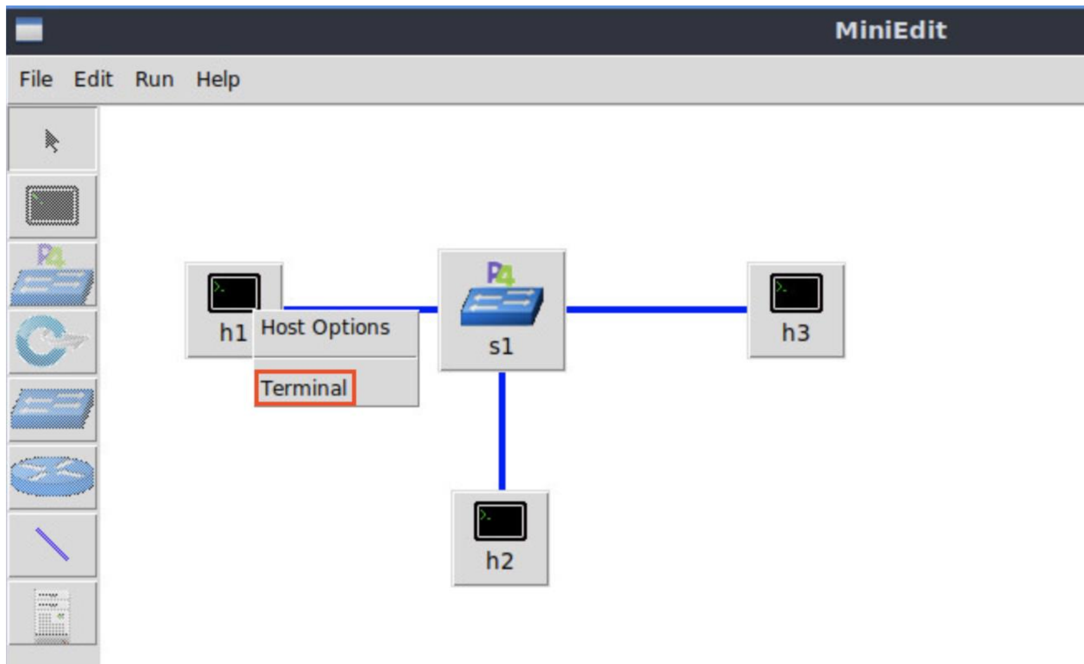


Figure 7. Opening a terminal on host h1.

3 Implementing checksum calculation in P4

This section demonstrates how to update the checksum of an IPv4 packet after being modified by the P4 program. We will be using the P4 program that implements the routing function.

3.1 Loading the programming environment

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

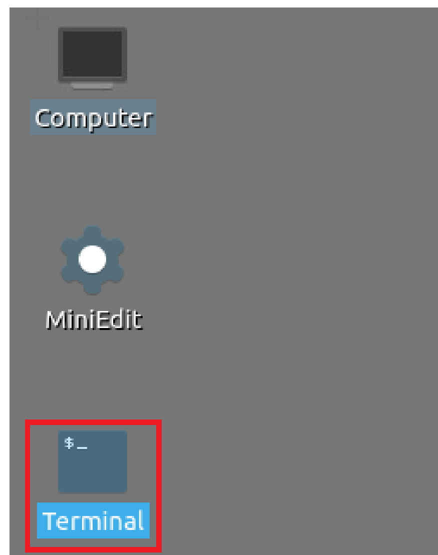


Figure 8. Shortcut to open a Linux terminal.

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.

```
code ~/P4_Labs/lab8
```

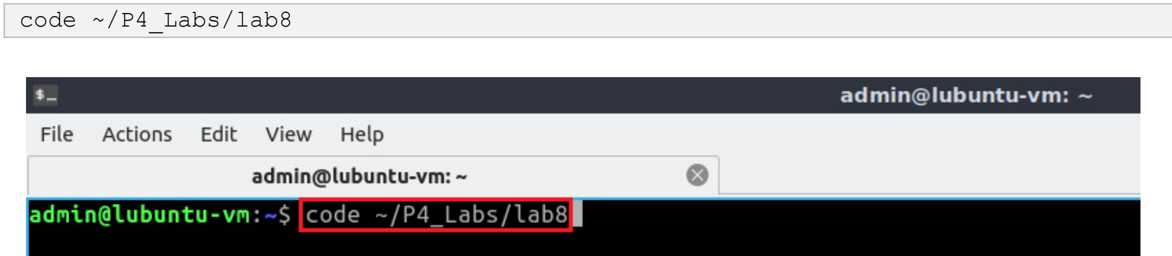


Figure 9. Launching the editor and opening the lab8 directory.

3.2 Inspecting the P4 code

Step 1. Inspect the content of the *ingress.p4* file before implementing the checksum calculation. Navigate into the file by clicking on *ingress.p4* in the file explorer

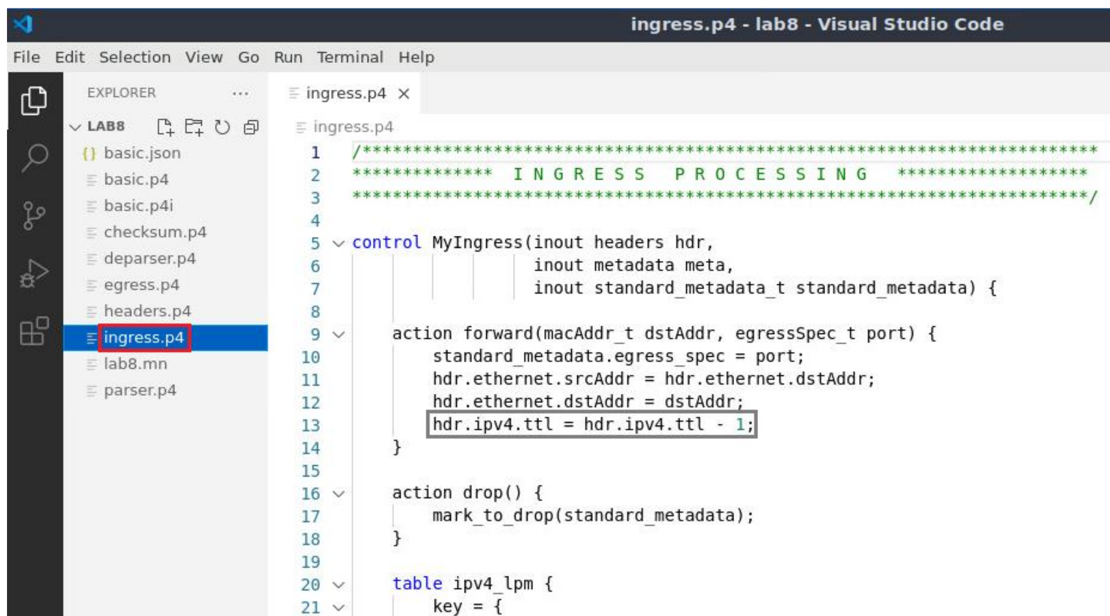


Figure 10. Inspecting the ingress block.

Note how the action `forward` is modifying the TTL value in the IPv4 header. Since the program is modifying the header fields, it is necessary to recompute and update the checksum of the header.

Step 2. Navigate into the checksum file by clicking on *checksum.p4* in the file explorer.

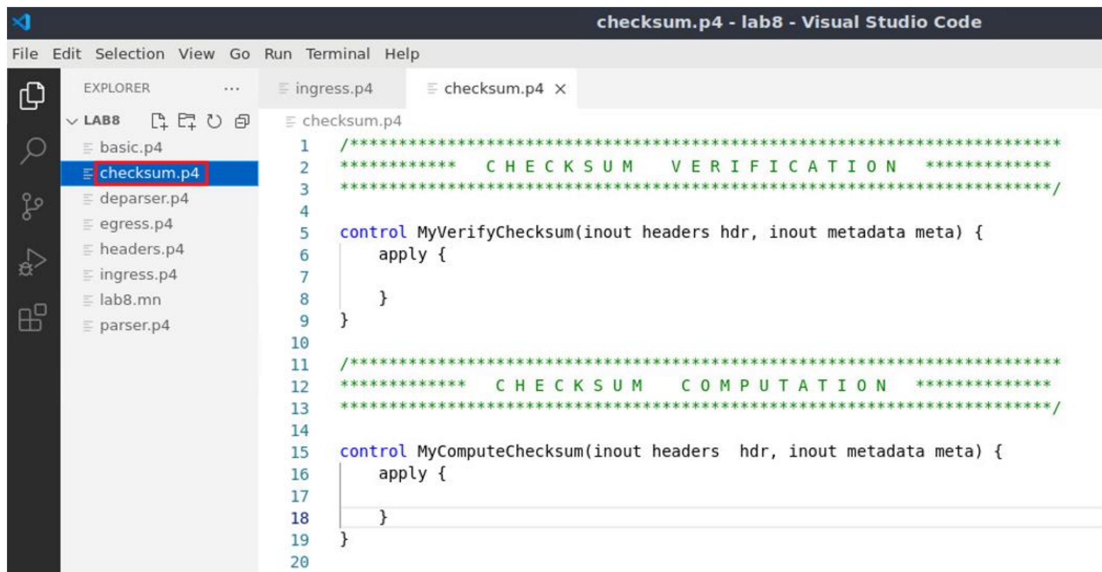


Figure 11. Inspecting the checksum block.

The figure above shows that two empty control blocks exist in the *checksum.p4* file. The first control `MyVerifyChecksum` verifies the checksum for the packet. The second control block `MyComputeChecksum` updates the checksum of the packet. We will only focus for now on computing the checksum. The upcoming steps show what happens when the checksum is not updated after modifying the IPv4 header (i.e., decrementing the TTL).

4 Loading the P4 program

4.1 Compiling and loading the P4 program to switch s1

Step 1. Compile the program by issuing the following command in the terminal.

```
p4c basic.p4
```

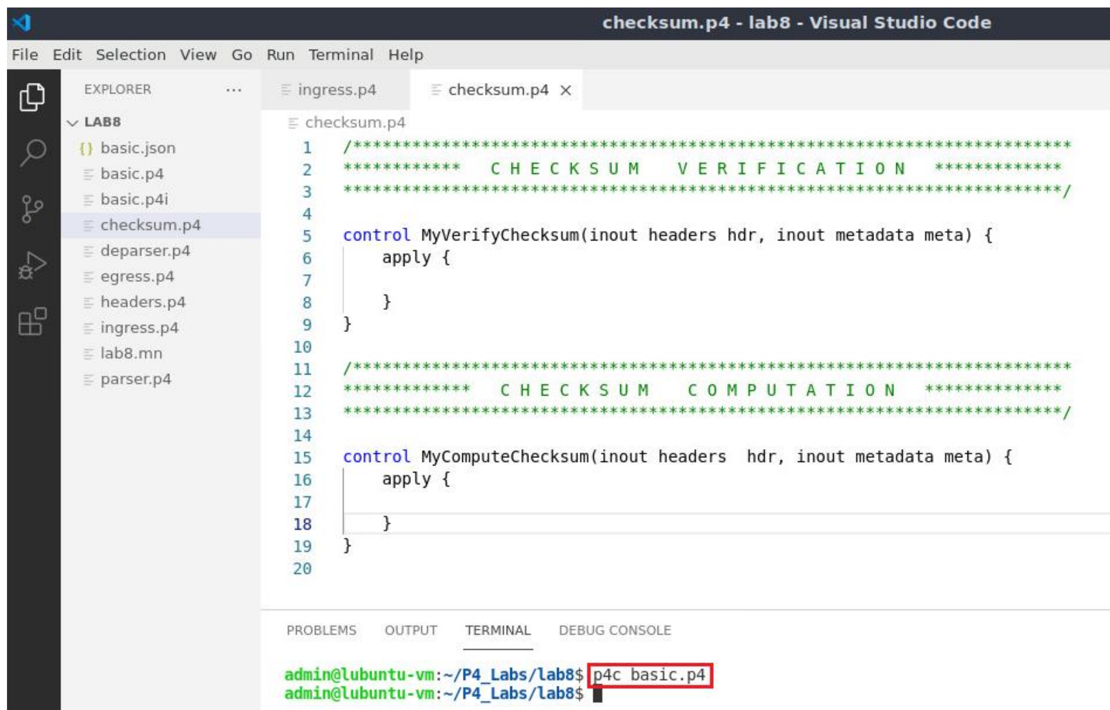


Figure 12. Compiling a P4 program.

Step 2. Push the output of the compiler to the switch by using the following command.

```
push_to_switch basic.json s1
```

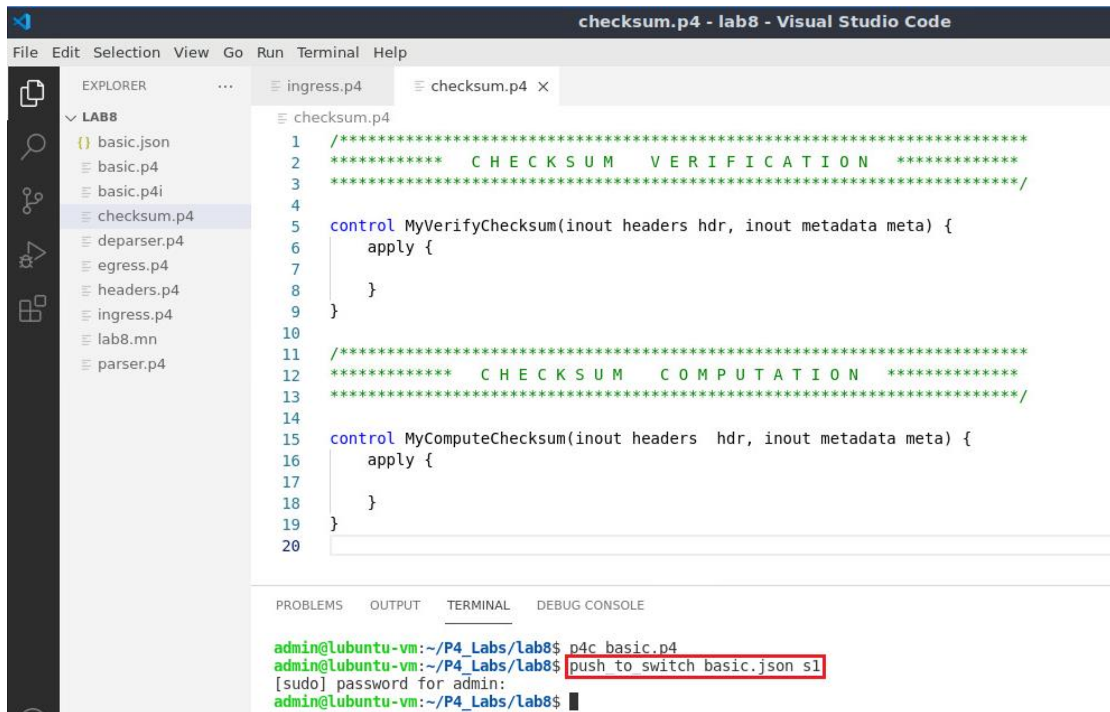


Figure 13. Downloading the *basic.json* file to switch s1.

4.2 Verifying the configuration

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

Figure 14. Maximizing the MiniEdit window.

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

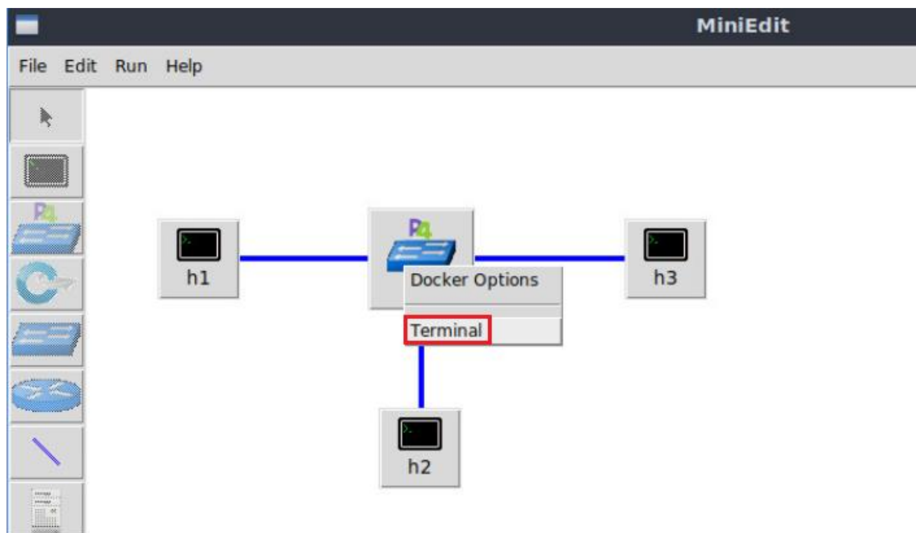


Figure 15. Starting the terminal on the switch.

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 following command on switch s1 terminal to inspect the content of the current folder.

```
ls
```

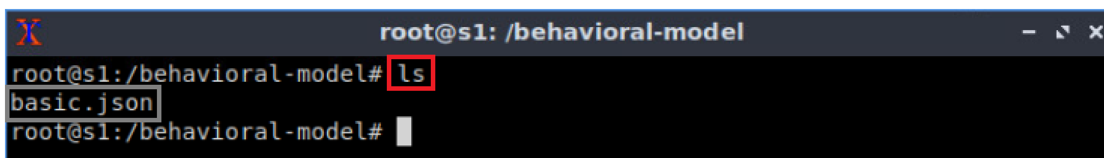


Figure 16. Displaying the contents of the current directory in the switch s1.

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

5 Configuring switch s1

5.1 Mapping P4 program's ports

Step 1. Issue the following command on 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:27 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:3265 (3.2 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:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

s1-eth0   Link encap:Ethernet  HWaddr 0e:7e:48:32:53:a3
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:356 (356.0 B)  TX bytes:0 (0.0 B)

s1-eth1   Link encap:Ethernet  HWaddr 9e:c5:42:78:07:16
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)

s1-eth2   Link encap:Ethernet  HWaddr 26:15:f3:b2:b1:d4
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)
```

Figure 17. Displaying switch s1 interfaces.

The output displays switch s1 interfaces (i.e., *s1-eth0*, *s1-eth1* and *s1-eth2)*. The interface *s1-eth0* on the switch s1 connects to the host h1. The interface *s1-eth1* on the switch s1 connects to the host h2 and *s2-eth2* is connected to host h3.

Step 2. Start the switch daemon by typing the following command, then press *Enter*.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog ipc:///tmp/bm-
log.ipc  basic.json &
```

```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog
 ipc:///tmp/bm-log.ipc basic.json &
[1] 39
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 18. 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.

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

```
simple_switch_CLI < ~/lab8/rules.cmd
```

```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab8/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to lpm match table MyIngress.ipv4_lpm
match key:           LPM-0a:00:00:00/8
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to lpm match table MyIngress.ipv4_lpm
match key:           LPM-14:00:00:00/8
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-1e:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 0
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 19. Populating the forwarding table into switch s1.

6 Manipulating the checksum and deparser

6.1 Sending a packet without checksum update

Step 1. On host h3's terminal, type the command below to launch Wireshark. Wireshark is a network analyzer used to inspect the content of network packets.

```
wireshark
```

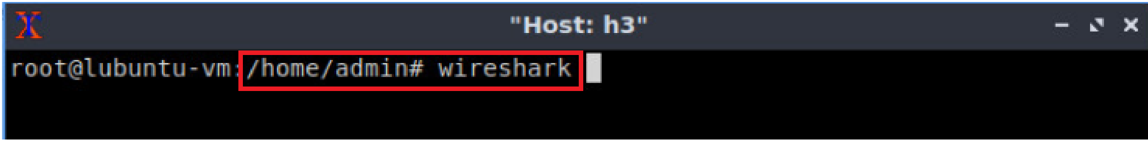


Figure 20. Opening Wireshark from host h3.

Step 2. Select the interface *h3-eth0* and start capturing packets by clicking on the icon located in the upper left-hand side.

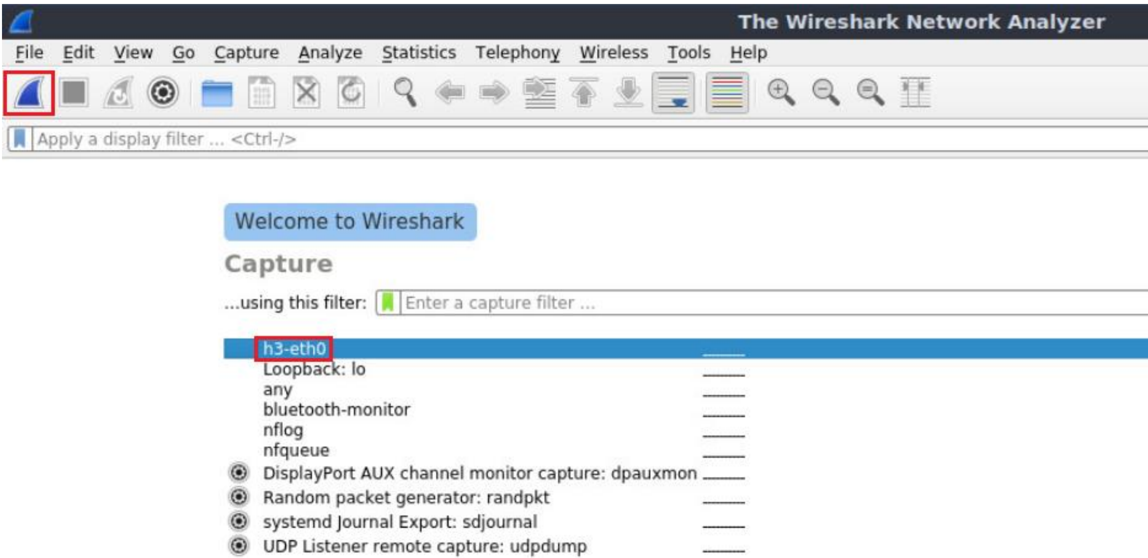


Figure 21. Starting packet capture.

Step 3. Apply the following filter in Wireshark by issuing the following keyword in the filter box, then press *Enter.*

```
tcp
```

Figure 22. Filtering TCP packets only.

Step 4. On host h1's terminal, send a packet to host h3 by issuing the following command.

```
./send.py 30.0.0.1 HelloWorld
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ./send.py 30.0.0.1 HelloWorld
sending on interface h1-eth0 to 30.0.0.1
###[ 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    = 0x52c4
     src       = 10.0.0.1
     dst       = 30.0.0.1
     \options   \
###[ TCP ]###
        sport     = 51535
        dport     = 1234
```

Figure 23. Sending a test packet from host h1 to host h3.

Step 5. Navigate back to the Wireshark window and click on the packet to see information about the received packet from h1.

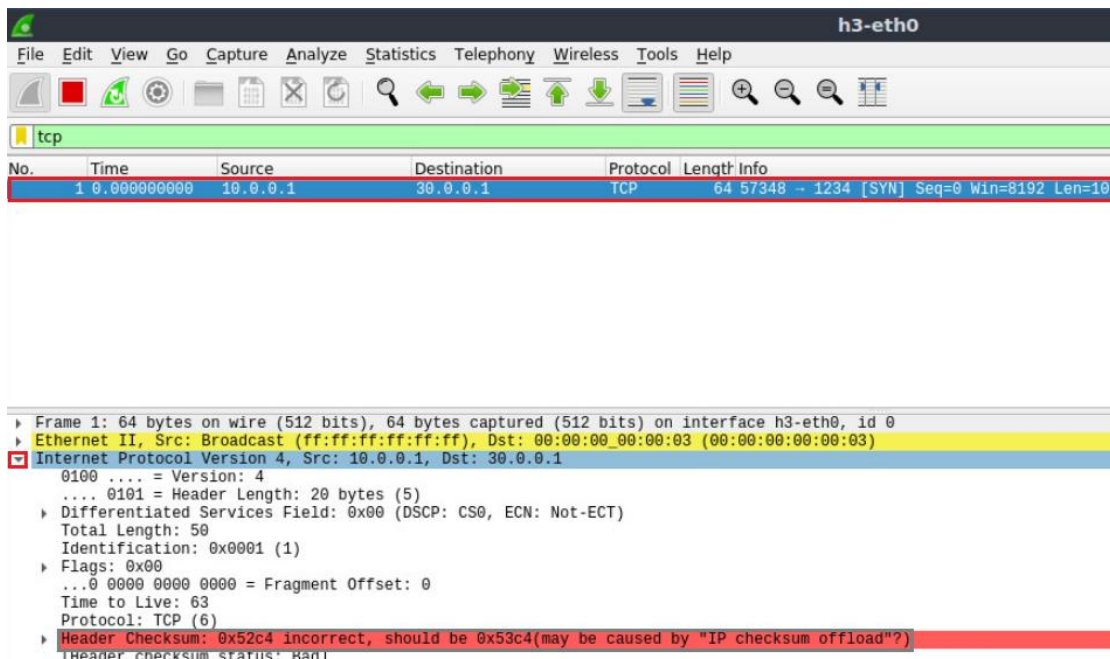


Figure 24. Inspecting the content of the IPv4 header.

We can see that the checksum in that packet is incorrect. This is because the P4 program changed the header field value (i.e., TTL), but did not update the checksum in the packet.

6.2 Sending a packet with checksum update

Step 1. Navigate back to the VS Code window and add the following code in the `MyComputeChecksum` block of the *checksum.p4* file.

```
update_checksum(
         hdr.ipv4.isValid(),
           {
             hdr.ipv4.version,
             hdr.ipv4.ihl,
             hdr.ipv4.diffserv,
             hdr.ipv4.totalLen,
             hdr.ipv4.identification,
             hdr.ipv4.flags,
             hdr.ipv4.fragOffset,
             hdr.ipv4.ttl,
             hdr.ipv4.protocol,
             hdr.ipv4.srcAddr,
             hdr.ipv4.dstAddr },
           hdr.ipv4.hdrChecksum,
           HashAlgorithm.csum16);
```

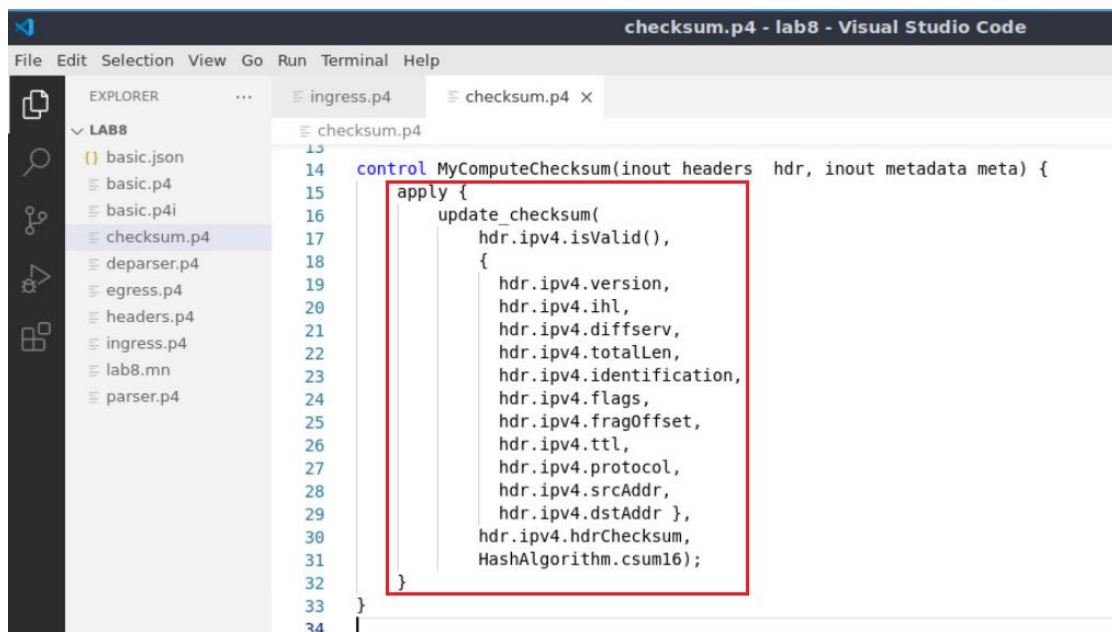


Figure 25. P4 code to update the checksum.

Step 2. Press `Ctrl + s` to save the changes.

Step 3. Issue the following command in the terminal panel inside the VS Code to compile the program.

```
p4c basic.p4
```

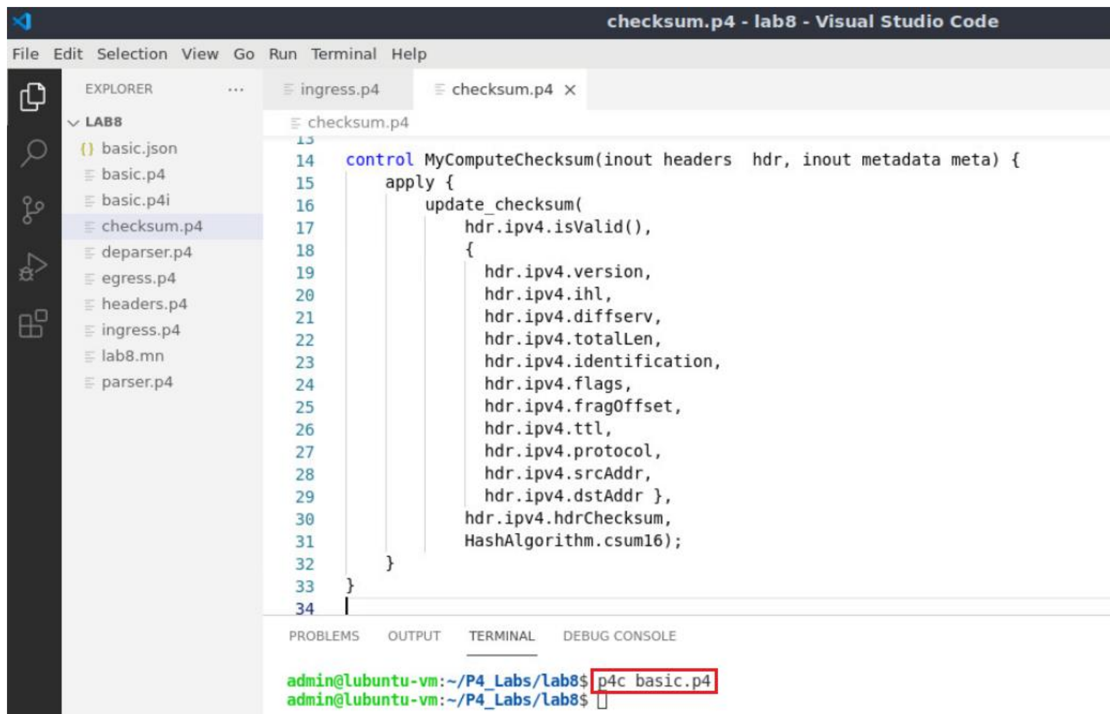


Figure 26. Compiling a P4 program.

Step 4. 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
```

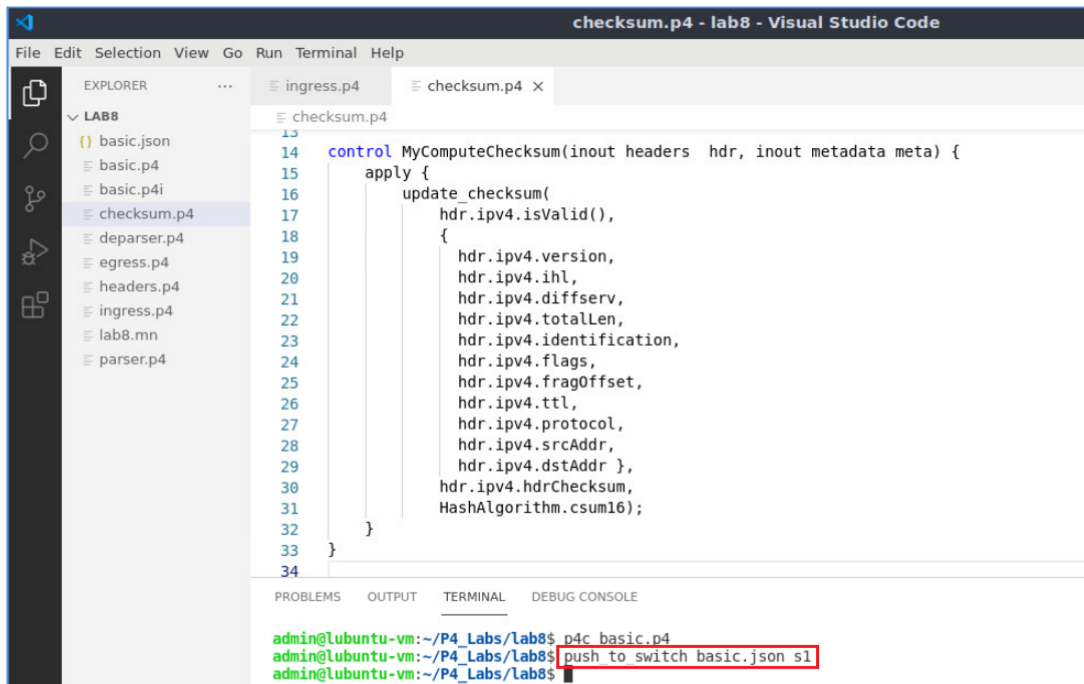


Figure 27. Pushing the *basic.json* file to switch s1.

Step 5. Navigate to the window of the switch s1 and stop the daemon of the switch by using the following command.

```
pkill switch
```

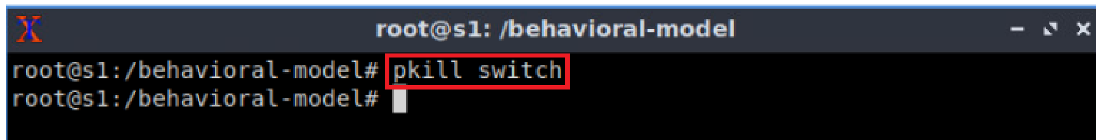


Figure 28. Terminating switch s1 process.

Step 6. Start the switch daemon by typing the following command, then press *Enter*.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog ipc:///tmp/bm-log.ipc  basic.json &
```

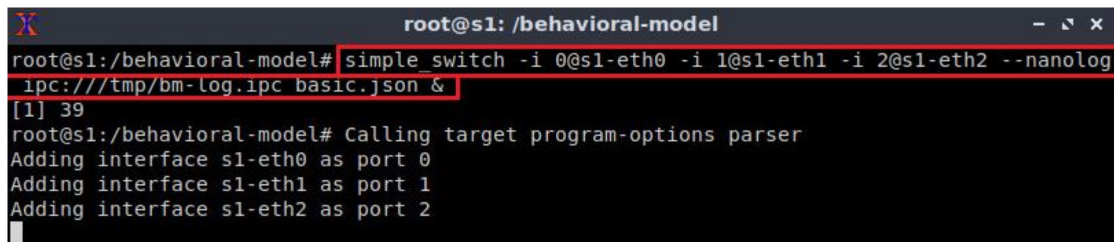


Figure 29. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 7. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab8/rules.cmd
```

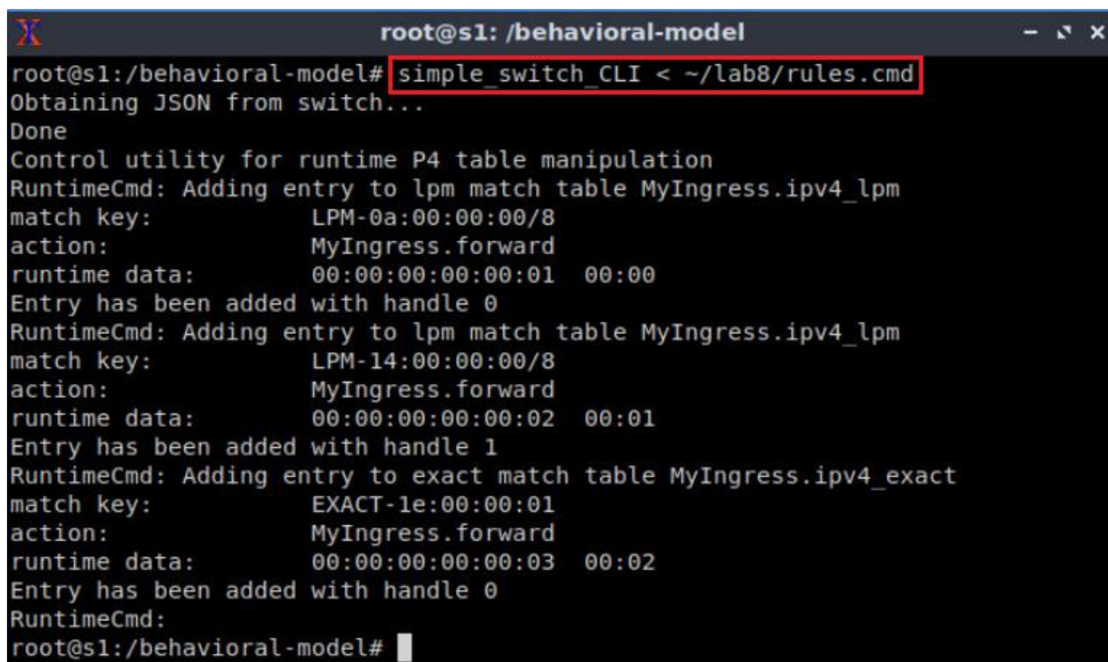


Figure 30. Populating the tables in switch s1.

Step 8. On host h1's terminal, send a packet to host h3 by issuing the following command.

```
./send.py 30.0.0.1 HelloWorld
```

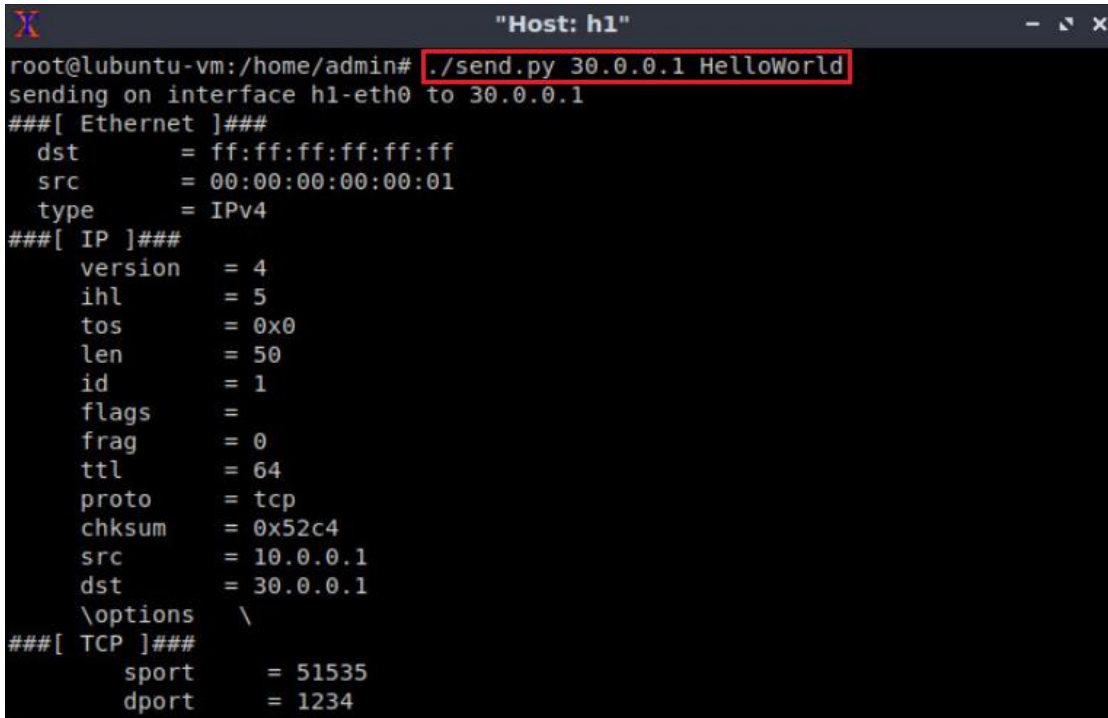


Figure 31. Sending a test packet from host h1 to host h3.

Step 9. Navigate back to the Wireshark window and click on the packet to see information about the received packet from h1.

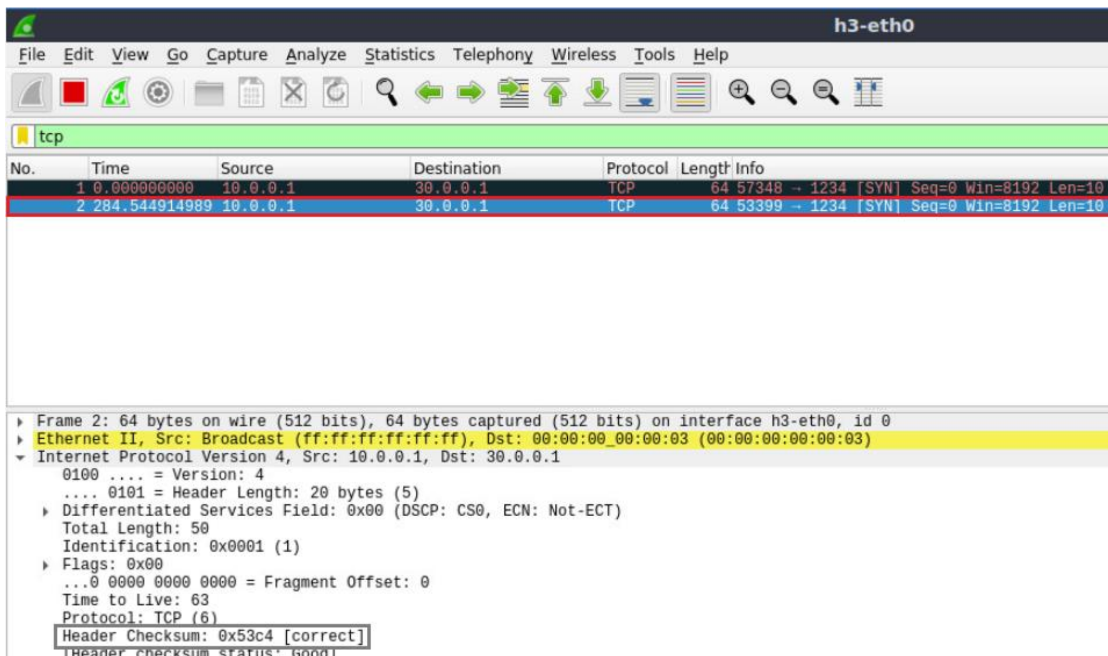


Figure 32. Inspecting the content of the IPv4 header.

We can see that the checksum in that packet is now correct. This is because the P4 program updated the checksum in the IPv4 header after changing the header field value (i.e., TTL).

6.3 Updating the deparser in the P4 code

In this section we will update the code of the deparser so that the IPv4 header is not emitted. We will then verify this operation using Wireshark.

Step 1. Navigate into the deparser block by clicking on *deparser.p4* in the file explorer of the VS Code application.

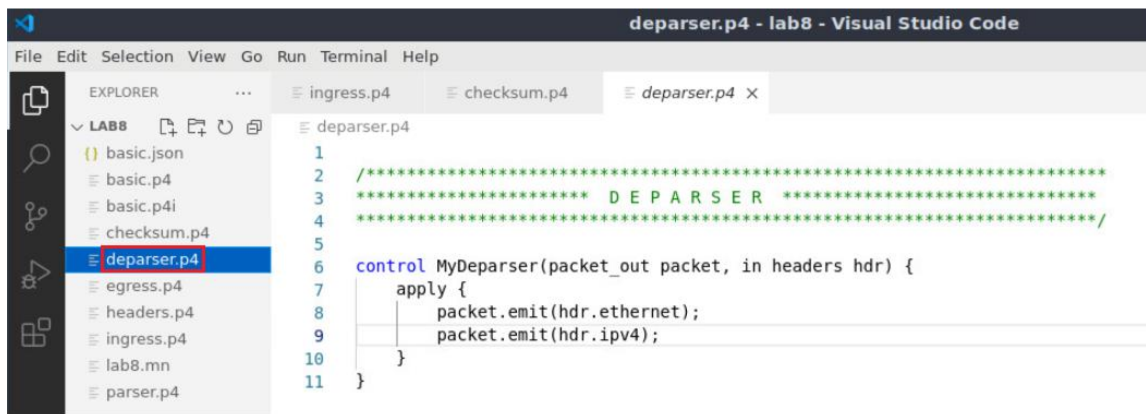


Figure 33. Inspecting the deparser block.

We can see in the figure above that the deparser is emitting both the Ethernet header and the IPv4 header. We verified in the previous step that Wireshark was able to recognize both headers.

Step 2. Update the deparser code to emit only the Ethernet header and save the file.

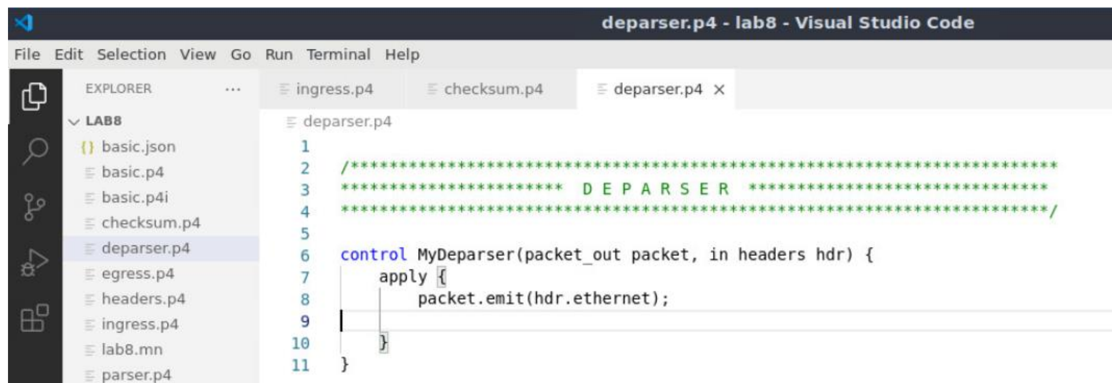


Figure 34. Emitting only the Ethernet header.

Step 3. Issue the following command in the terminal panel inside the VS Code to compile the program.

```
p4c basic.p4
```

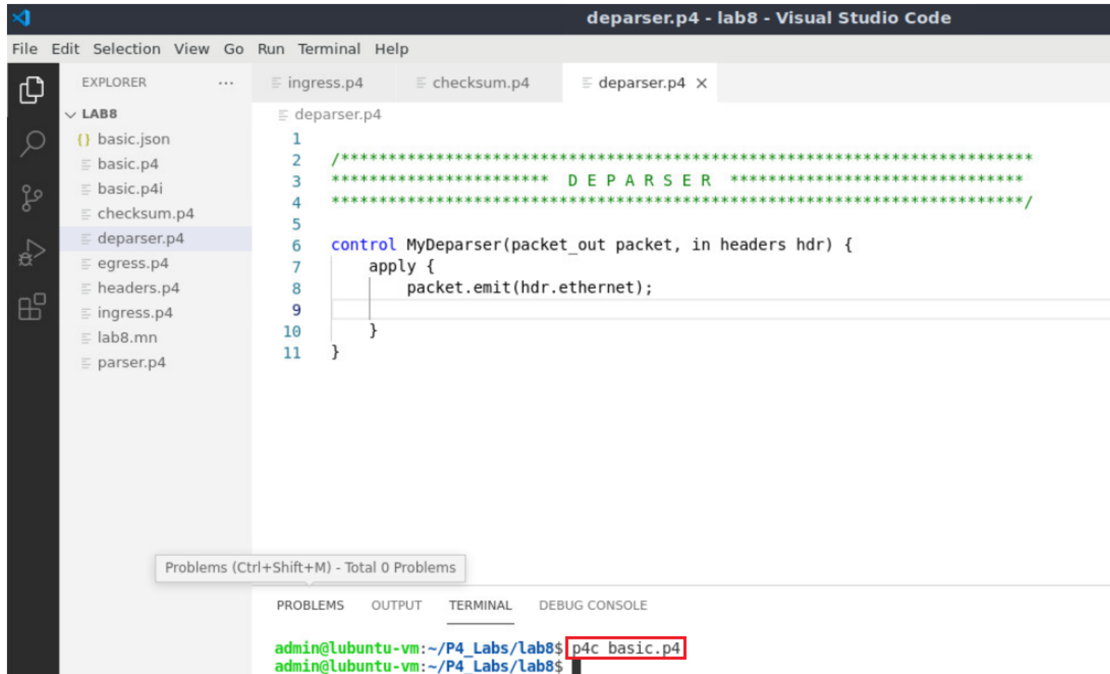


Figure 35. Compiling a P4 program.

Step 4. 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
```

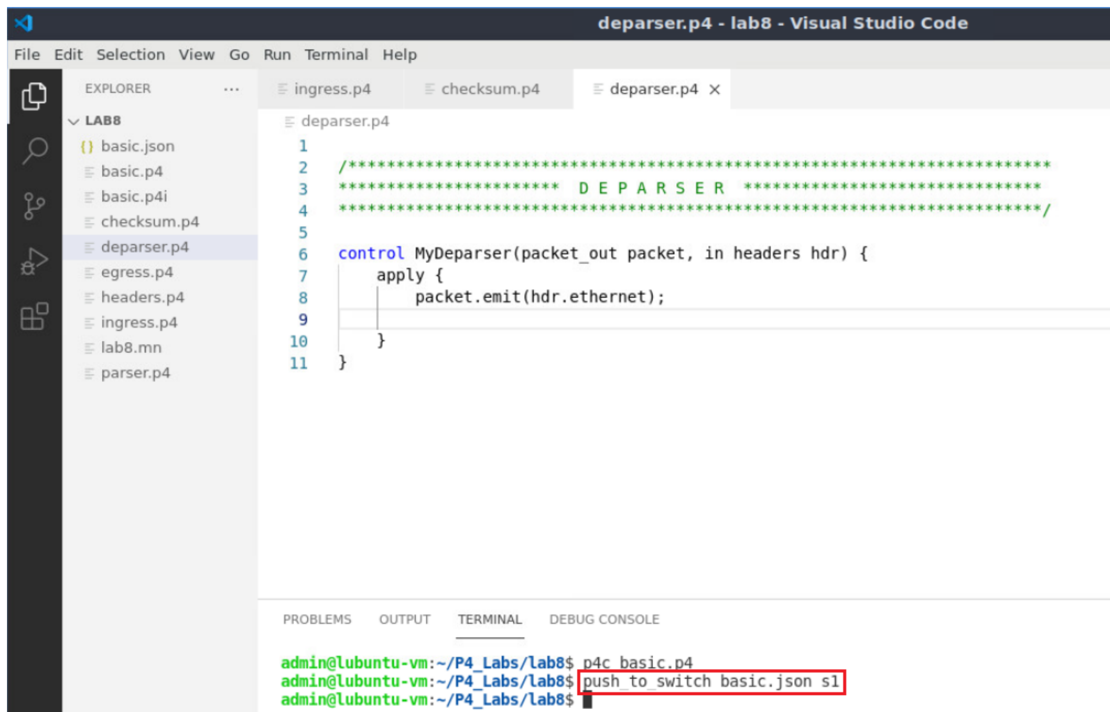


Figure 36. Downloading the *basic.json* file to switch s1.

Step 5. Navigate to the window of the switch s1 and stop the daemon of the switch by using the following command.

```
pkill switch
```

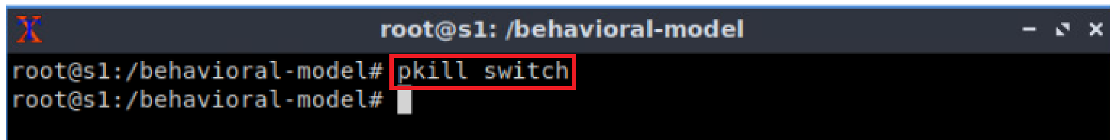


Figure 37. Terminating switch s1 process.

Step 6. Start the switch daemon by typing the following command, then press *Enter*.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog ipc:///tmp/bm-log.ipc  basic.json &
```

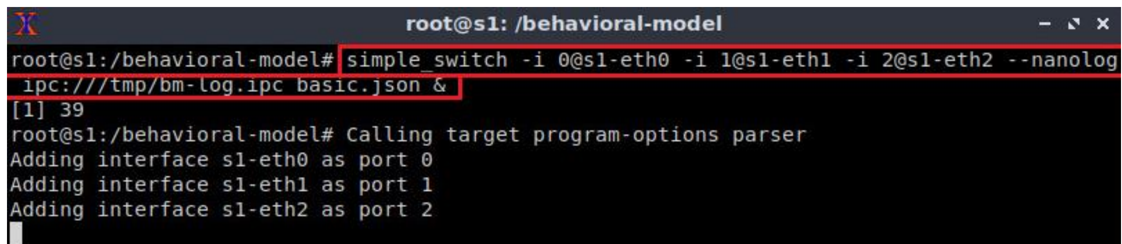


Figure 38. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 7. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab8/rules.cmd
```

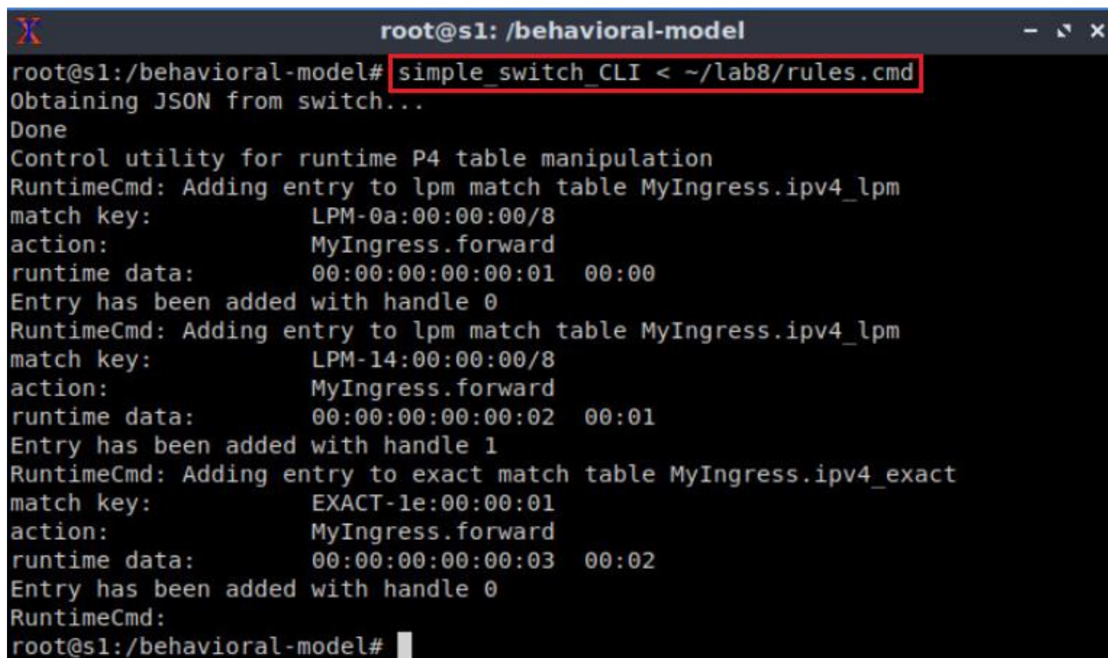


Figure 39. Populating the tables in switch s1.

Step 8. On host h1's terminal, send a packet to host h3 by issuing the following command.

```
./send.py 30.0.0.1 HelloWorld
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# ./send.py 30.0.0.1 HelloWorld
sending on interface h1-eth0 to 30.0.0.1
###[ 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    = 0x52c4
     src       = 10.0.0.1
     dst       = 30.0.0.1
     \options   \
###[ TCP ]###
        sport     = 51535
        dport     = 1234
```

Figure 40. Sending a test packet from host h1 to host h3.

Step 9. Navigate back to the Wireshark window and remove the *tcp* keyword from the filter and press *Enter.*

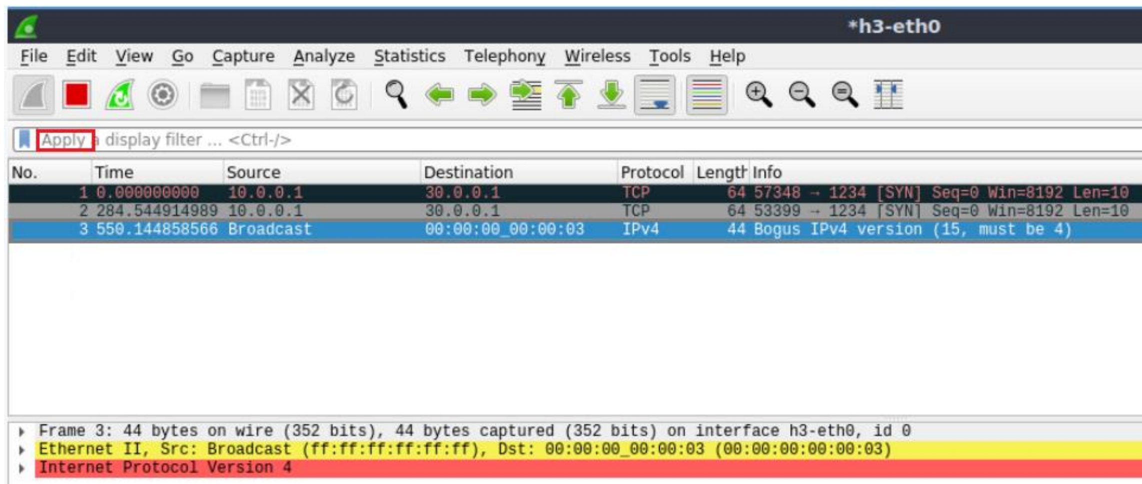


Figure 41. Removing the TCP filter from Wireshark.

Step 10. Click on the new packet (No. 3) to inspect its headers.

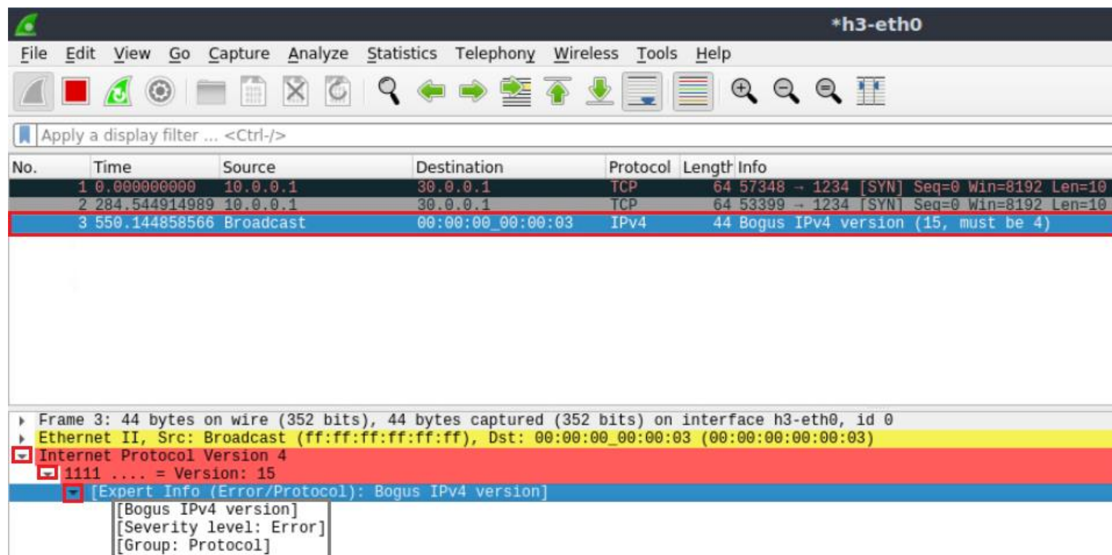


Figure 42. Inspecting the content of the IPv4 header.

The result above indicates that the IPv4 header is bogus. This error could occur due to the following reasons:

- The Ethernet header field *etherType* contains the value 0x0800 which indicates that the next header is of type IPv4.
- Since the P4 program did not emit the IPv4 header, the packet contains only the Ethernet header followed by the original payload of the packet.
- Wireshark is assuming that the next header is IPv4 (because of the *etherType* field of the Ethernet header), and hence it is interpreting the payload of the packet as the IPv4 header. Thus, Wireshark is alerting that the IPv4 header is bogus.

The IPv4 header must be emitted in the P4 program to solve this issue. Note that if you defined your own custom header in the P4 program and emit it in the deparser, Wireshark will not recognize it and might produce error messages.

This concludes lab 8. Stop the emulation and then exit out of MiniEdit.

References

1. RFC 791. "*Internet Protocol.*" 1981.
2. Mininet walkthrough. [Online]. Available: http://Mininet.org.
3. M. Peuster, J. Kampmeyer, H. Karl. "*Containernet 2.0: A rapid prototyping platform for hybrid service function chains.*" 4th IEEE Conference on Network Softwarization and Workshops (NetSoft). 2018.
4. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
5. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.

INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Exercise 6: Building a Packet Reflector

Document Version: 06-15-2022

Award 2118311
“Cybertraining on P4 Programmable Devices using an Online Scalable Platform with Physical and Virtual Switches and Real Protocol Stacks”

Contents

1 Exercise description 3
1.1 Ethernet header 3
1.2 IPv4 header........ 3
1.3 IPv6 header........ 3
1.4 Exercise topology 3
1.5 Credentials 4
2 Setting the environment........ 4
3 Deliverables........ 5

1 Exercise description

In this exercise, you will implement a P4 program that acts as a packet reflector. This means that the switch will bounce back a packet to the port the packet came from. You will be implementing the whole P4 program. This includes the headers definition, the parser, the control blocks, and the checksum update.

The header definitions are shown below.

1.1 Ethernet header

Ethernet determines that the next header is IPv4 if the value of *EtherType* is 0x0800.

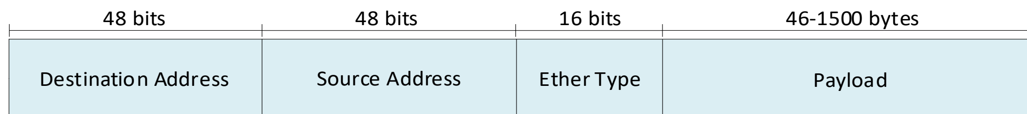


Figure 1. Ethernet header.

1.2 IPv4 header

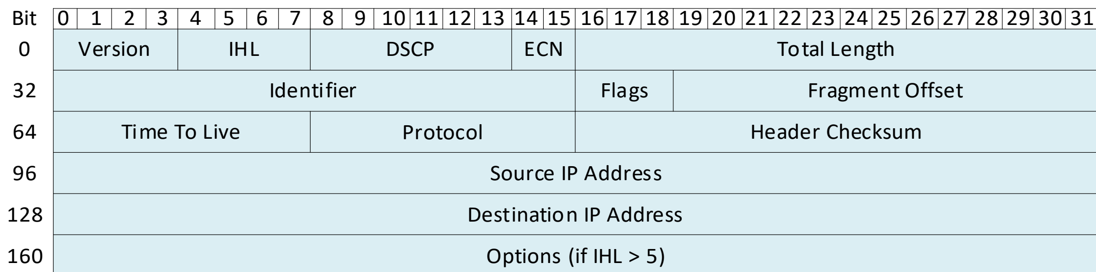


Figure 2. IPv4 header.

1.3 IPv6 header

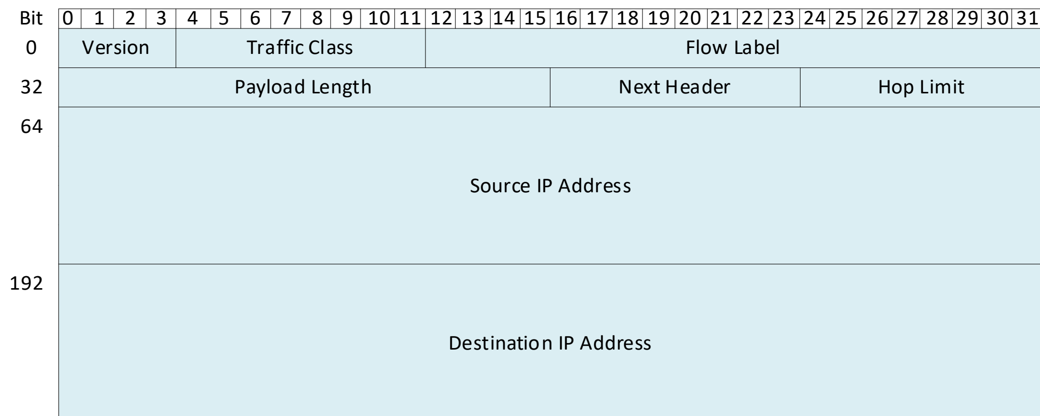


Figure 3. IPv6 header.

1.4 Exercise topology

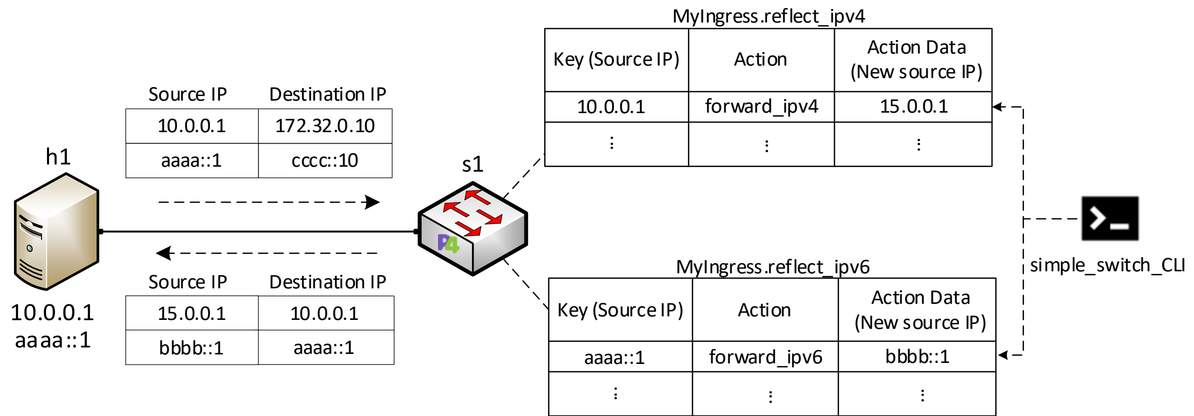


Figure 4. Lab topology.

1.5 Credentials

The information in Table 1 provides the credentials to access the Client's virtual machine.

Table 1. Credentials to access the Client's virtual machine.

Device	Account	Password
Client	admin	password

2 Setting the environment

Follow the steps below to set the exercise's environment.

Step 1. Open MiniEdit by double-clicking the shortcut on the desktop. If a password is required type `password`.

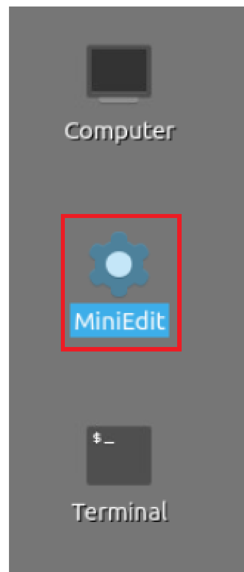


Figure 5. MiniEdit shortcut.

Step 2. Load the topology located at */home/admin/P4_Exercises/Exercise6/*.

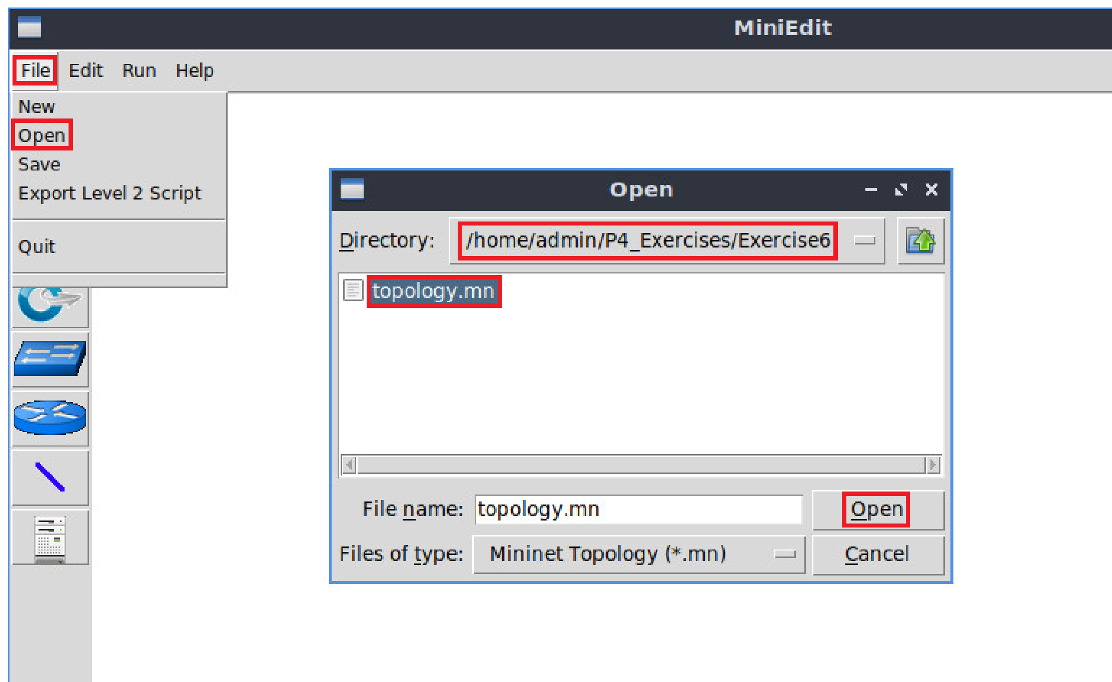


Figure 6. Opening exercise's topology.

Step 3. Run the emulation by clicking on the button located on the lower left-hand side.

Figure 7. Running the emulation.

Step 4. In the terminal, type the command below. This command launches the Visual Studio Code and opens the directory where the P4 program for this exercise is located.

```
code ~/P4_Exercises/Exercise6/
```

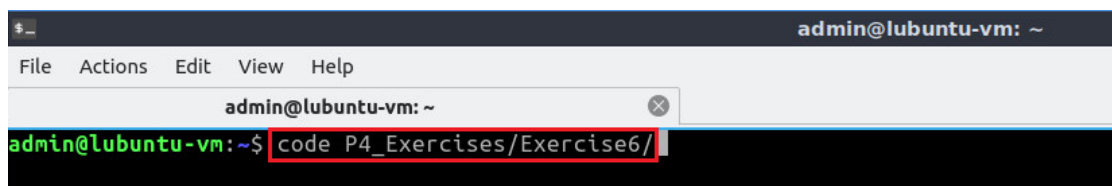


Figure 8. Opening the working directory.

3 Deliverables

Follow the steps below to complete the exercise.

a) Define the headers for the Ethernet, IPv4, and IPv6 protocols.

b) Implement the parser. Recall that the Ethernet header uses the *Ethertype* field to identify the next header. The *Ethertype* is 0x800 for IPv4 and 0x86DD for IPv6.

c) Implement the match-action tables `reflect_ipv4` and `reflect_ipv6` in the ingress pipeline. The actions in this table are `forward_ipv4` and `forward_ipv6`.

The action `forward_ipv4` must execute the following operations:

- Set the egress port the same as the ingress port. Note that the ingress port is available in the standard metadata `standard_metadata.ingress_port`.
- Set the source IPv4 address as the destination IPv4 address. Note that the packet is sent back to host h1.
- Assign a new destination IPv4 address.
- Decrement the TTL field in the IPv4 header.

Similarly, the action `forward_ipv6` must execute the following operations:

- Set the egress port the same as the ingress port. Note that the ingress port is available in the standard metadata `standard_metadata.ingress_port`.
- Set the source IPv6 address as the destination IPv6 address. Note that the packet is sent back to host h1.
- Assign a new destination IPv6 address.
- Decrement the hop limit field in the IPv6 header.

d) Implement the `drop` action.

e) Write the code to update the checksum of IPv4. Note that no checksum is needed for IPv6.

f) Start the daemon in switch s1 and associate the interfaces to their corresponding ports and enable `nanolog`.

g) Populate the tables at runtime using the `simple_switch_CLI`.

h) Open a terminal in host h1 and run the following command:

```
./recv.py
```

i) Using another terminal, send an IPv4 packet using the *send.py* program. Verify that the tables are matching by inspecting the logs of the switch using the `nanomsg` tool. Verify that the TTL field and hop limit field are decremented.

j) Repeat **i)** using the *send_ipv6.py* program.