UNIVERSITY OF SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Book Version: 06-13-2022

Principal Investigator: Jorge Crichigno

NSF

Contents

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 1: Introduction to Mininet

Document Version: 01-25-2022

Contents

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.

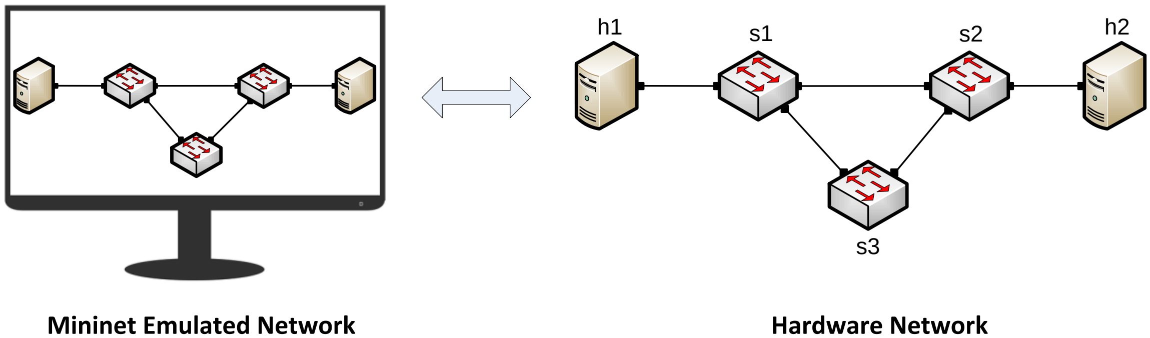


Figure 1. Hardware network vs. Mininet emulated network.

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.

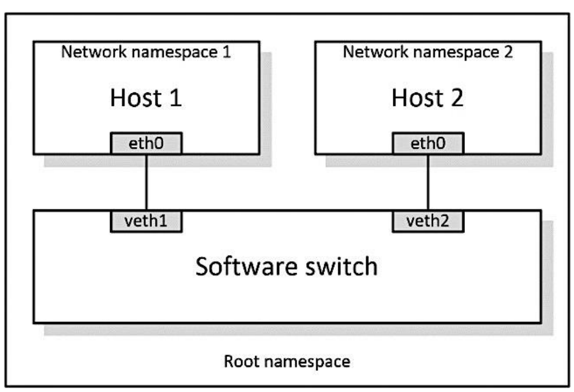


Figure 2. Network namespaces and virtual Ethernet links.

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.

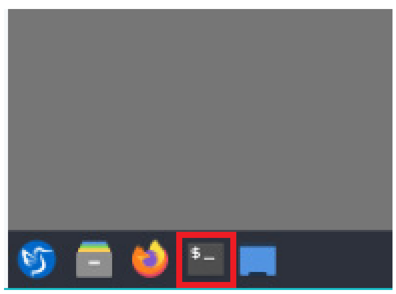


Figure 3. Linux terminal icon.

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
```

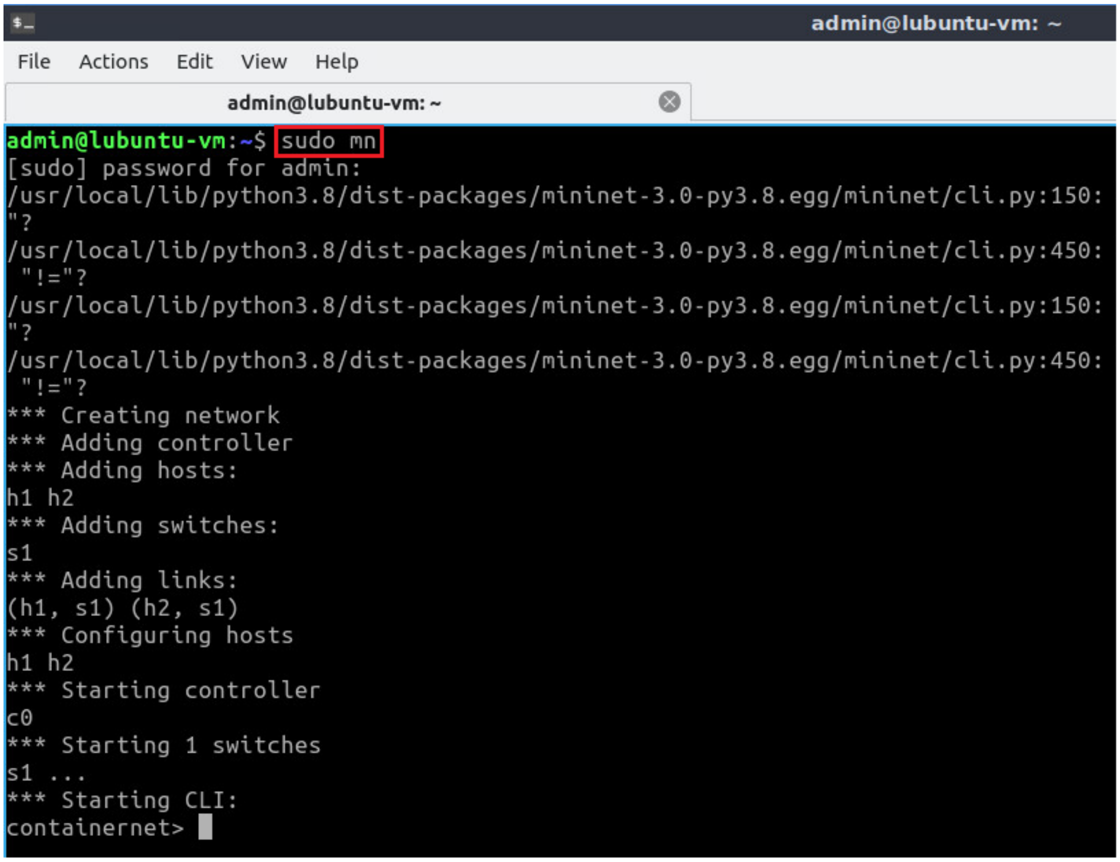


Figure 4. Starting Mininet using the CLI.

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

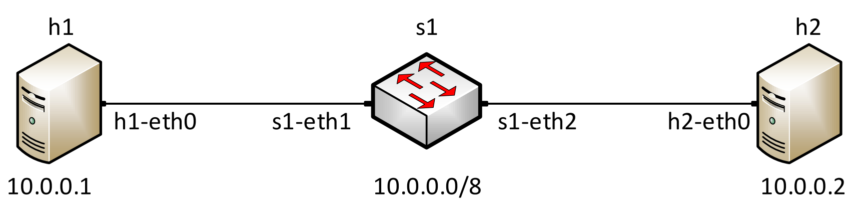


Figure 5. Mininet's default minimal topology.

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
```

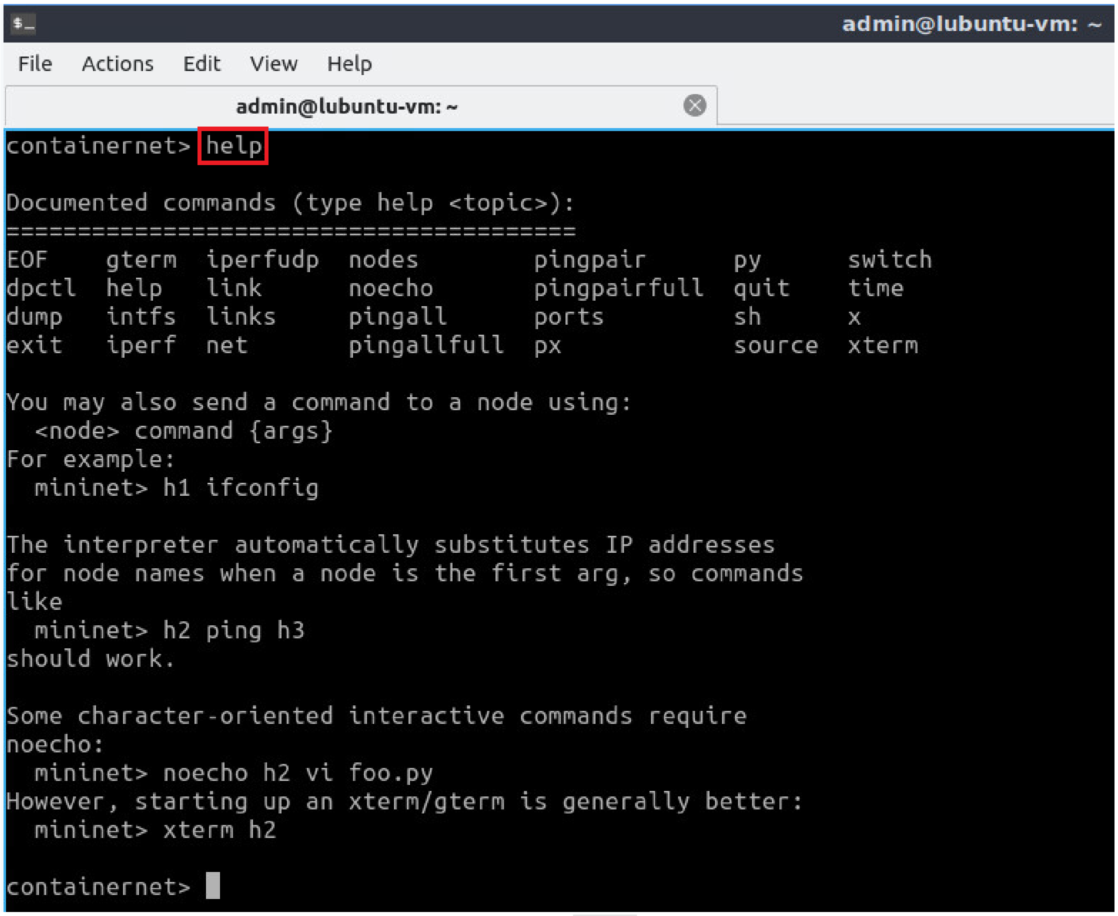


Figure 6. Mininet's `help` command.

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

```
nodes
```

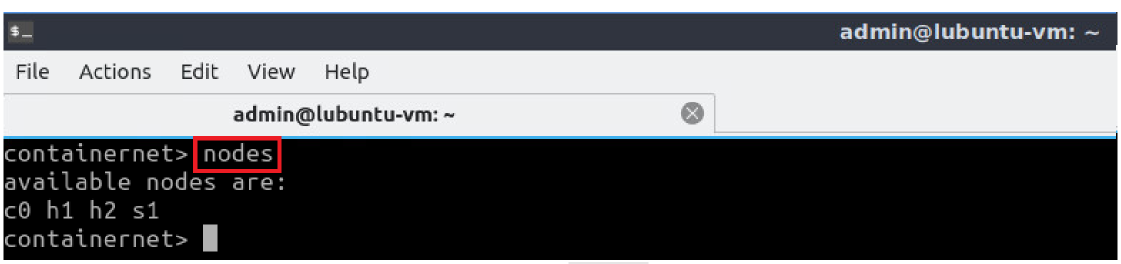


Figure 7. Mininet's `nodes` command.

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
```

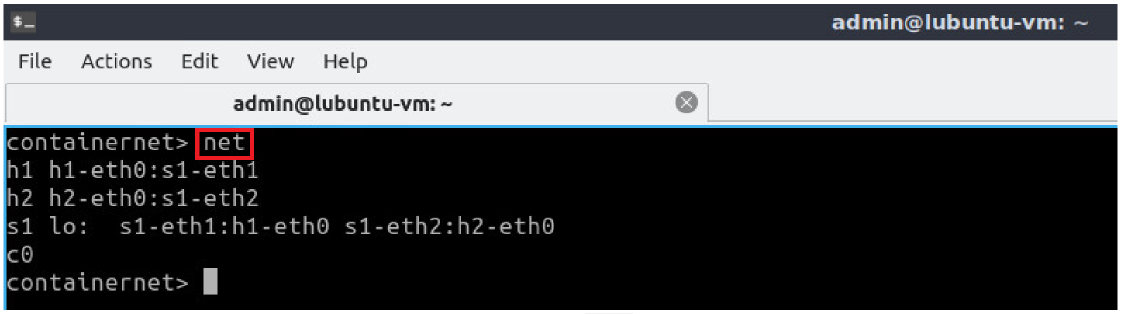


Figure 8. Mininet's `net` command.

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
```

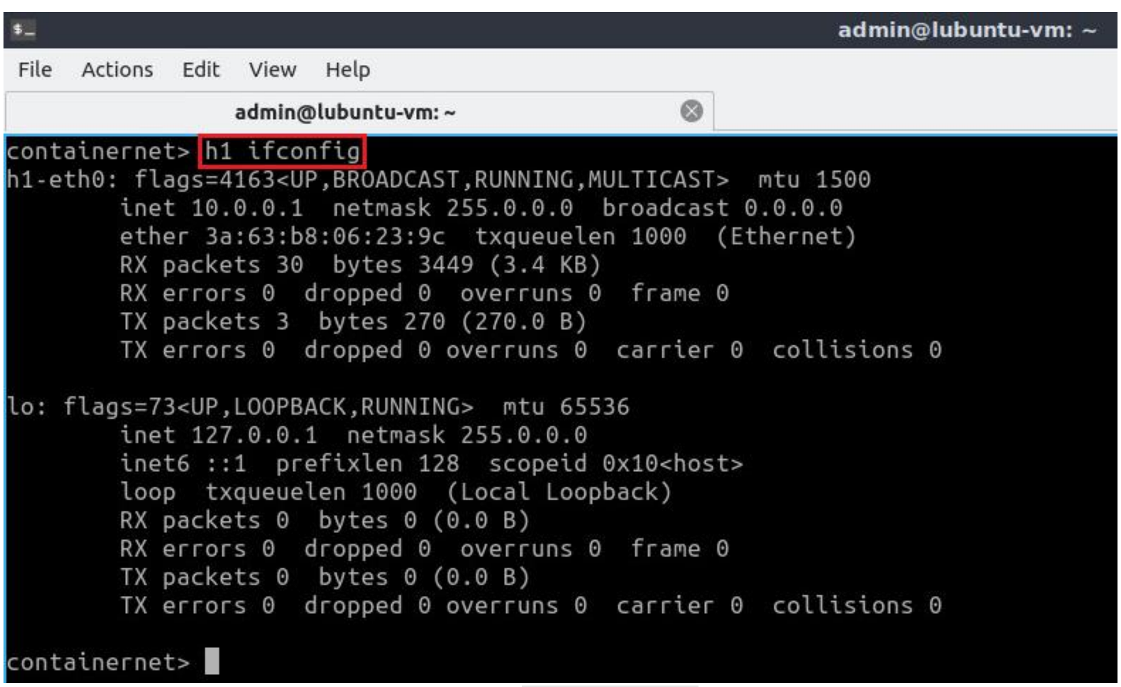


Figure 9. Output of `h1 ifconfig` command.

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
```

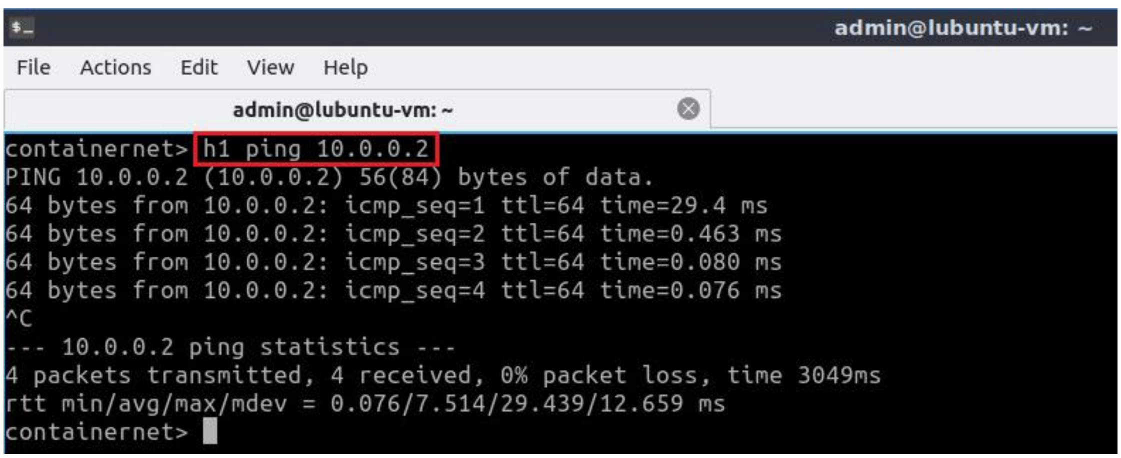


Figure 10. Connectivity test between host h1 and host h2.

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

```
exit
```

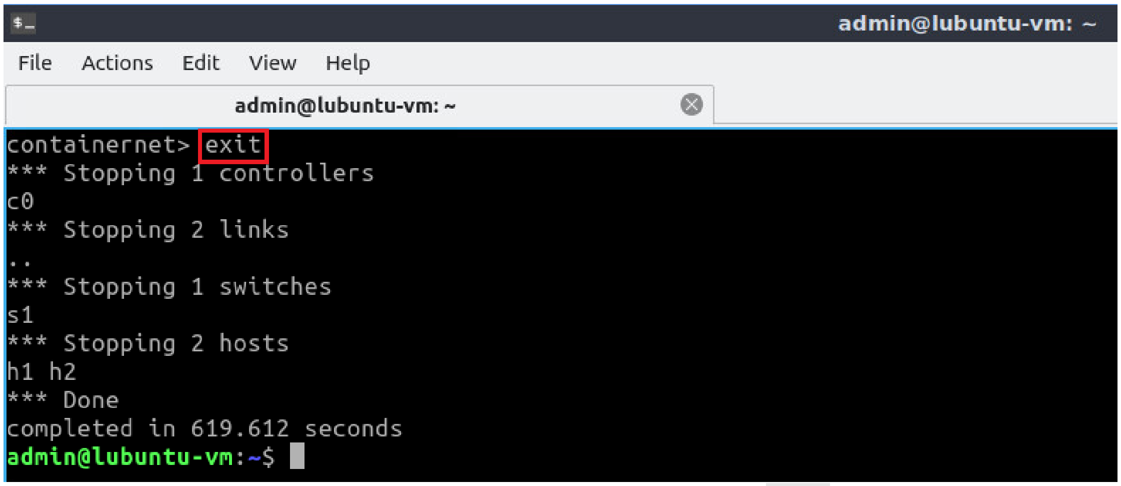


Figure 11. Stopping the emulation using `exit`.

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.

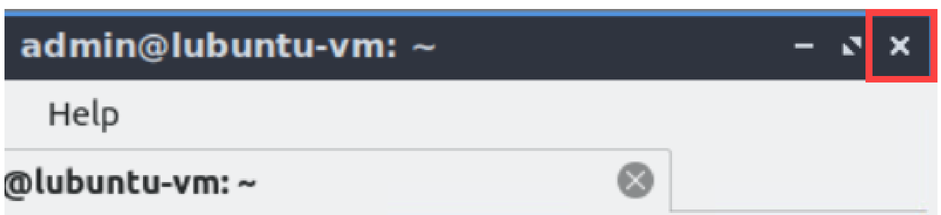


Figure 12. Closing the Linux CLI.

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.

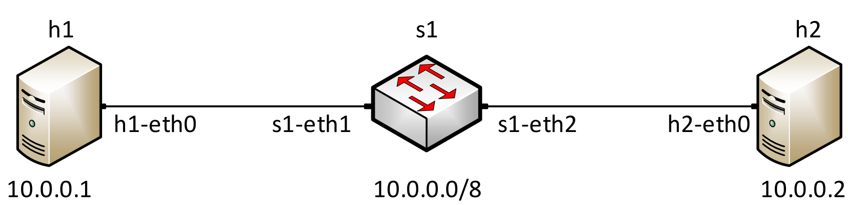


Figure 13. Lab topology.

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.

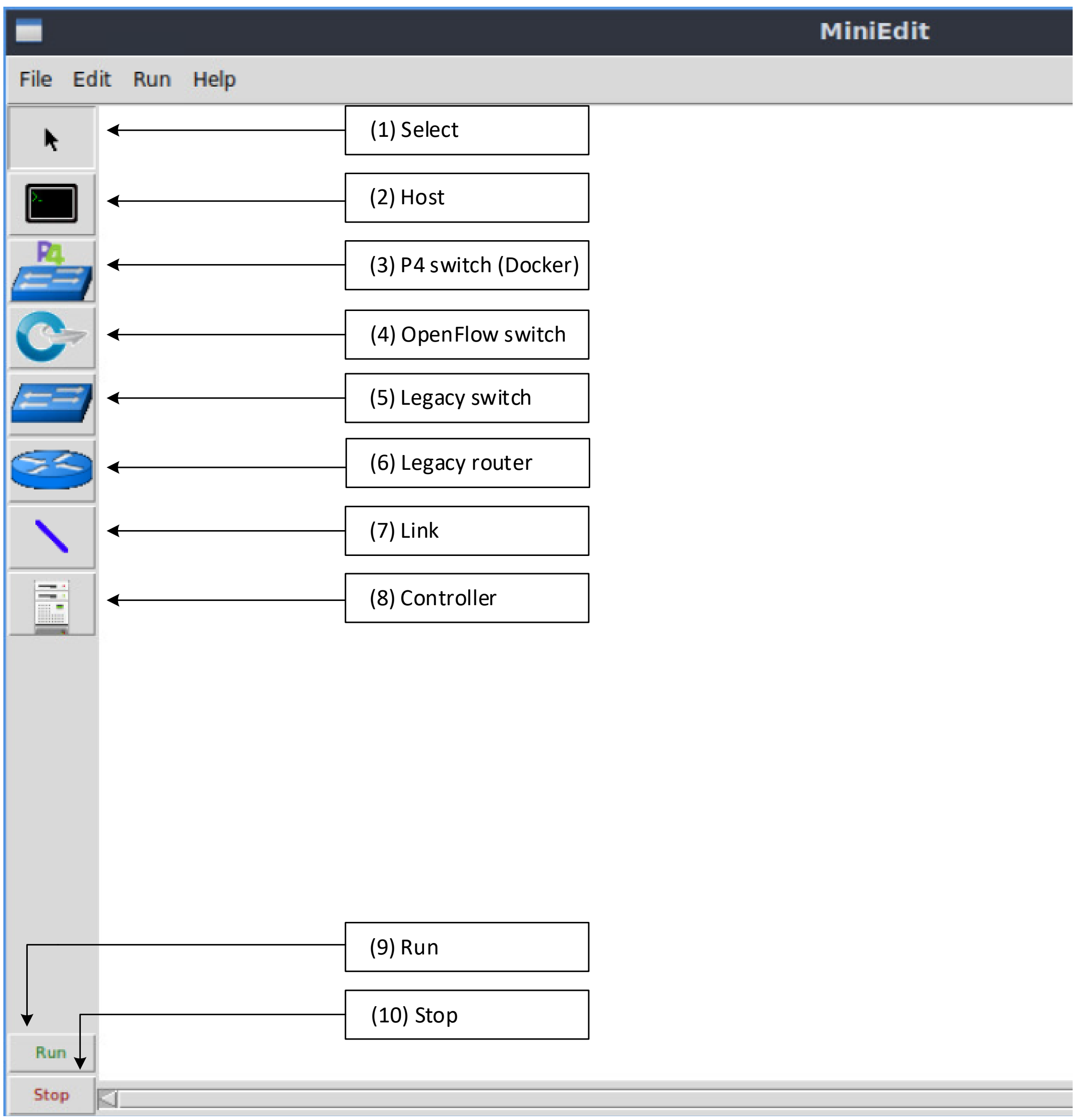


Figure 15. MiniEdit Graphical User Interface (GUI).

The main buttons are:

1. Select: allows selection/movement of the devices. Pressing *Delete* 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.

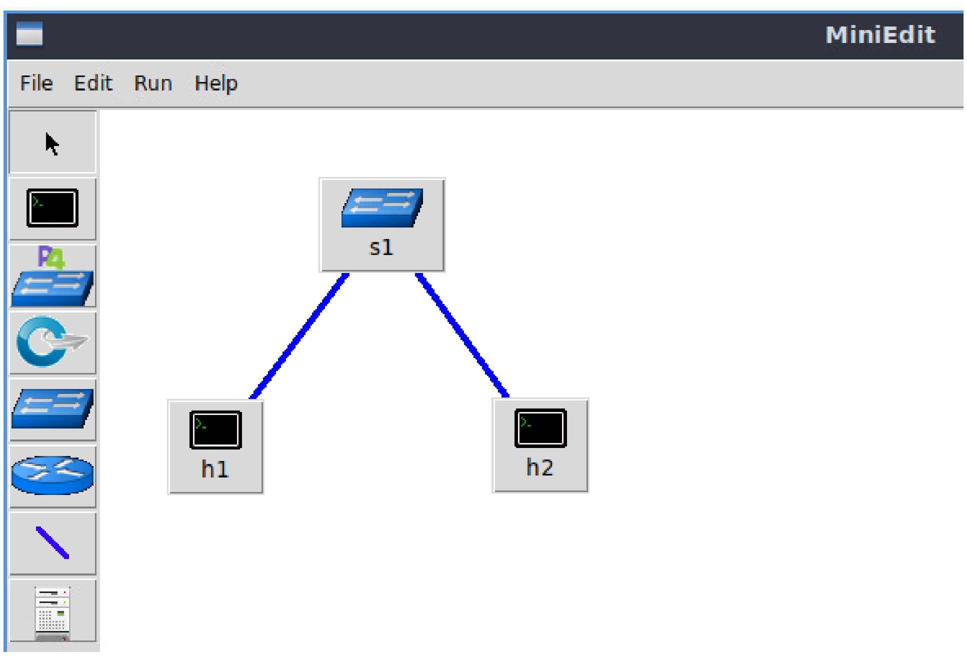


Figure 16. MiniEdit's topology.

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.

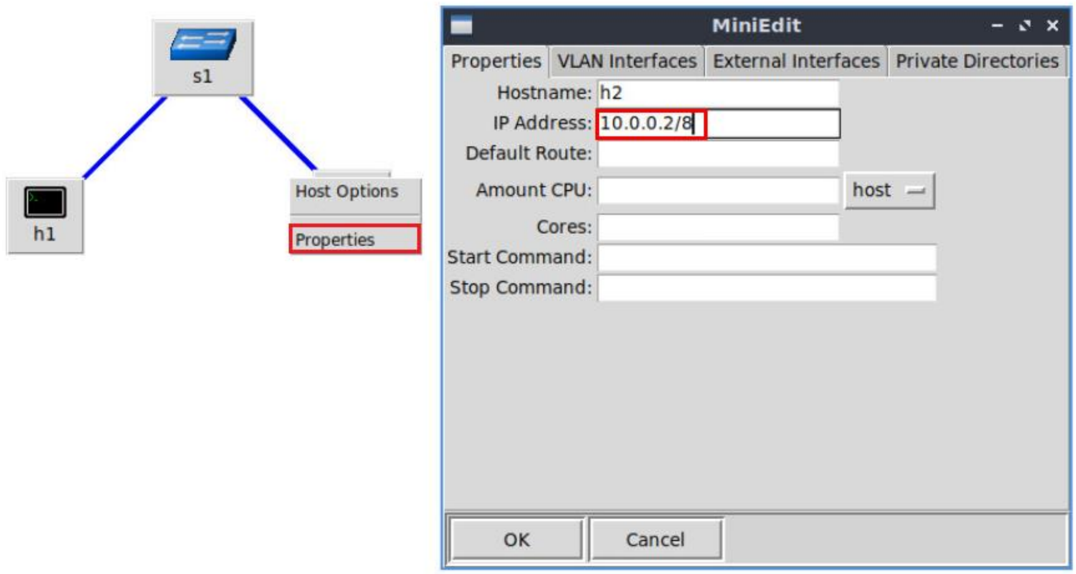


Figure 17. Configuration of a host's properties.

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.

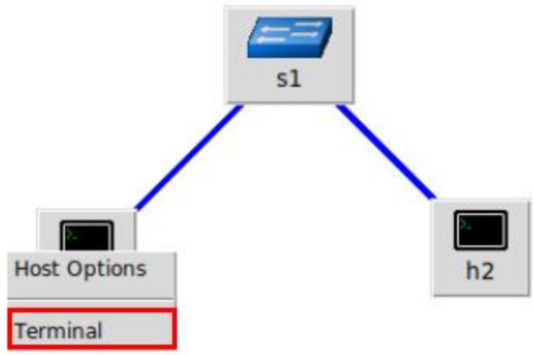


Figure 19. Opening a terminal on host h1.

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

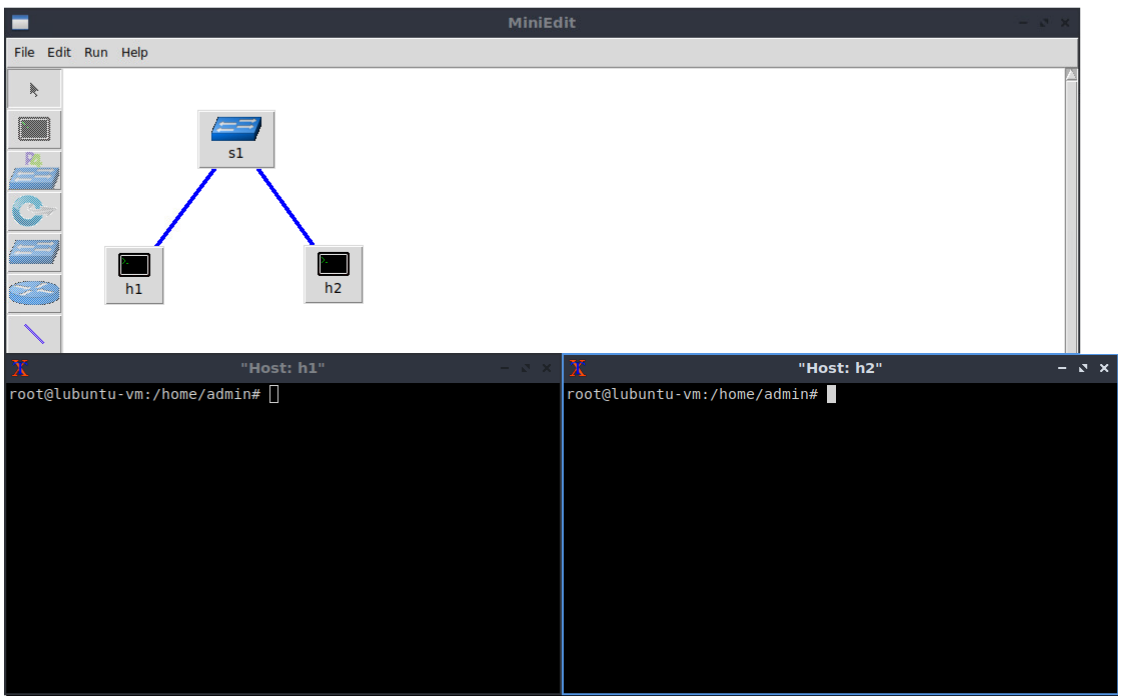


Figure 20. Terminals at host h1 and host h2.

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
```

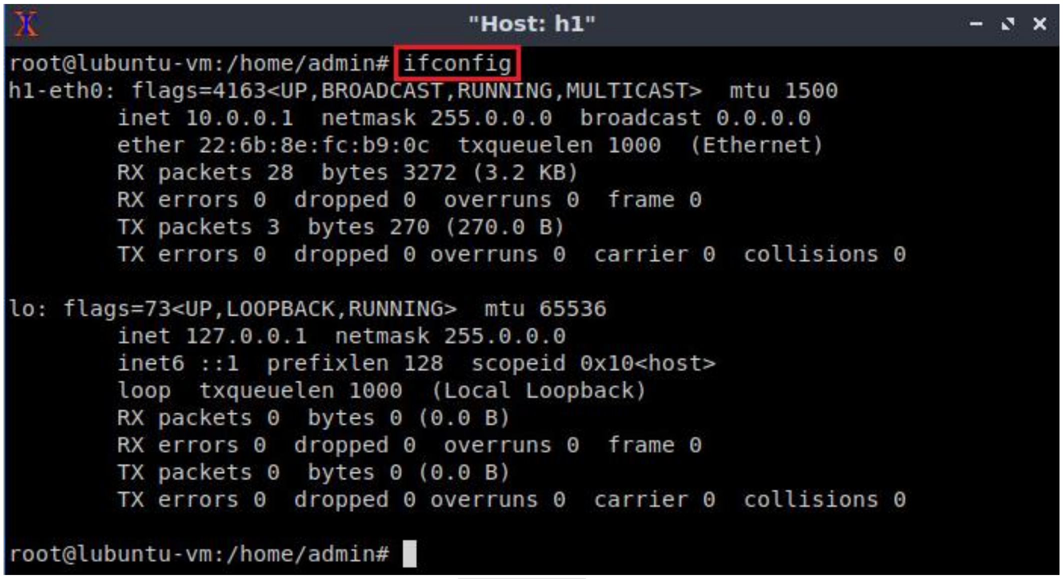


Figure 21. Output of `ifconfig` command on host h1.

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
```

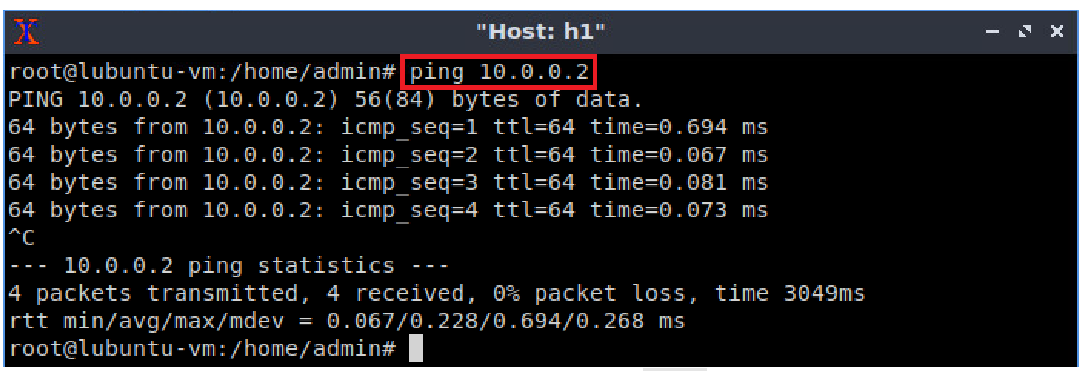


Figure 22. Connectivity test using `ping` command.

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

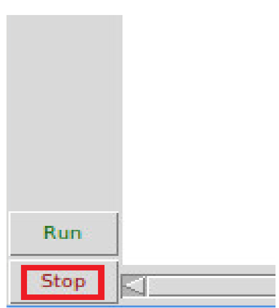


Figure 23. Stopping the emulation.

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.

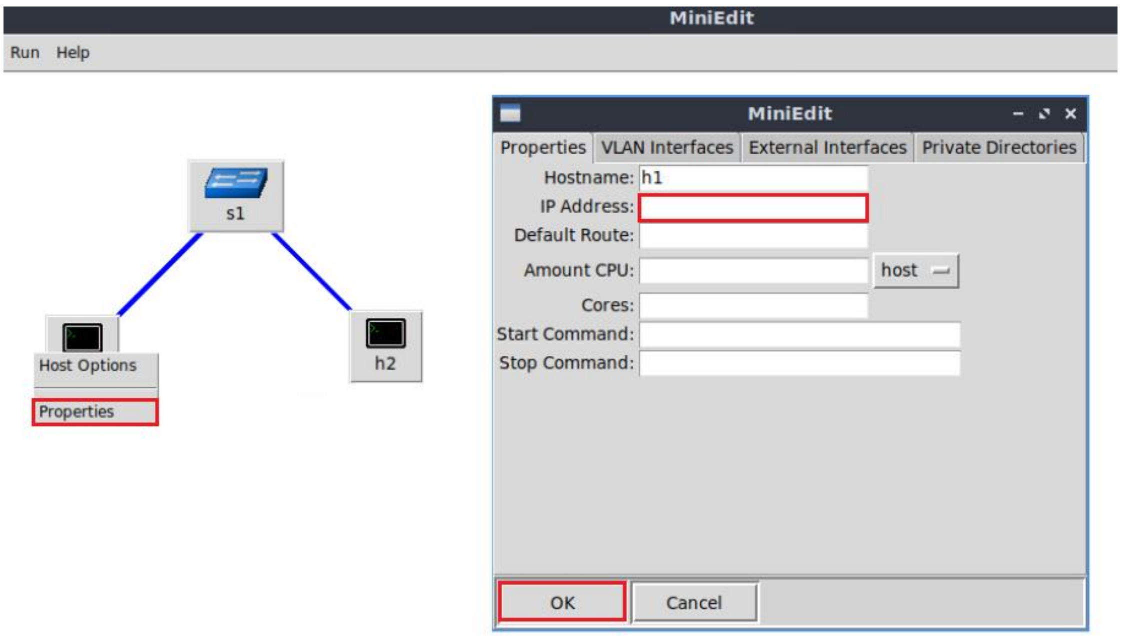


Figure 24. Host h1 properties.

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.

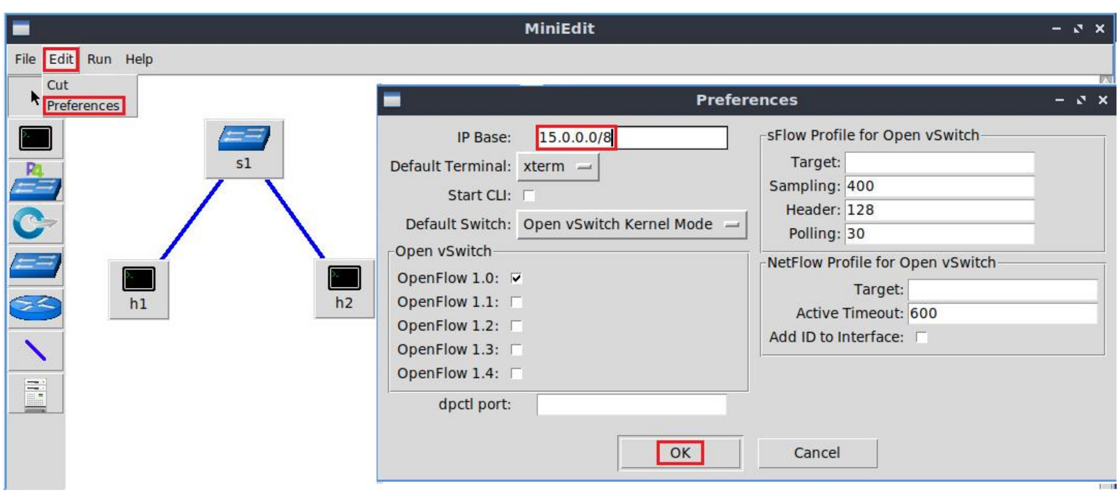


Figure 25. Modification of the IP Base (network address and prefix length).

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*.

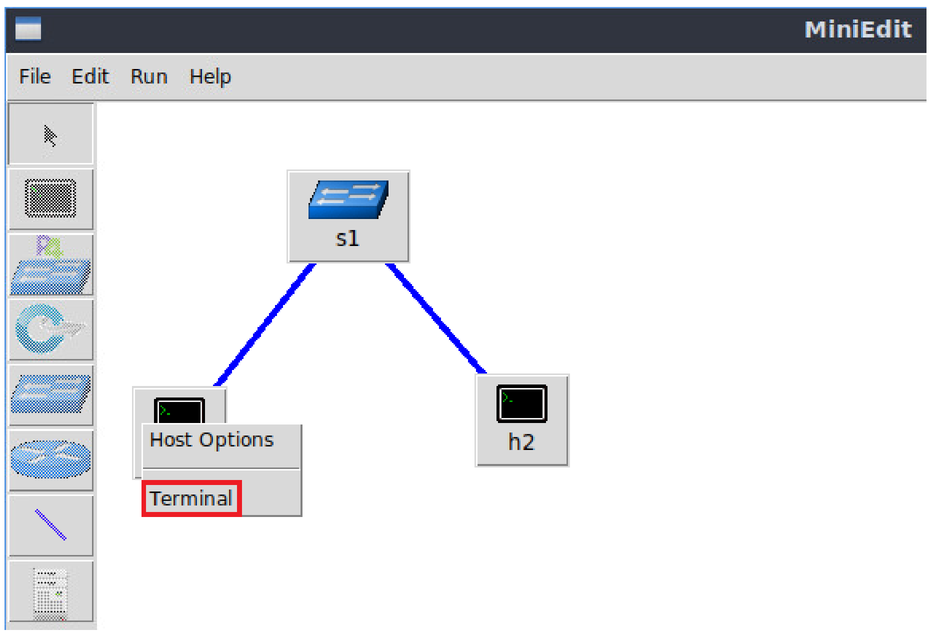


Figure 27. Opening a terminal on host h1.

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
```

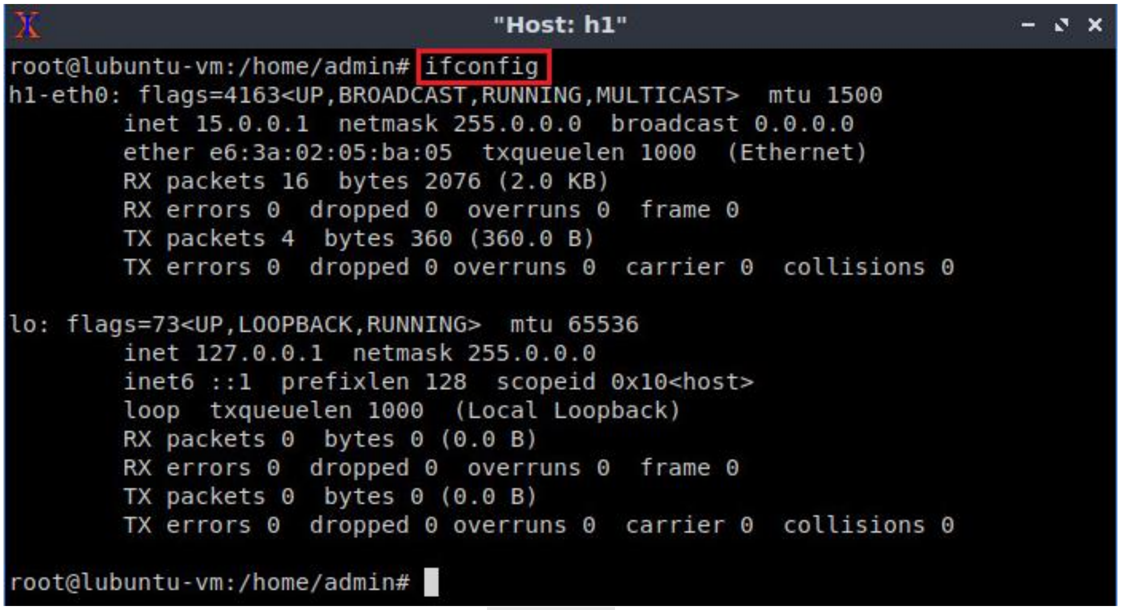


Figure 28. Output of `ifconfig` command on host h1.

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*.

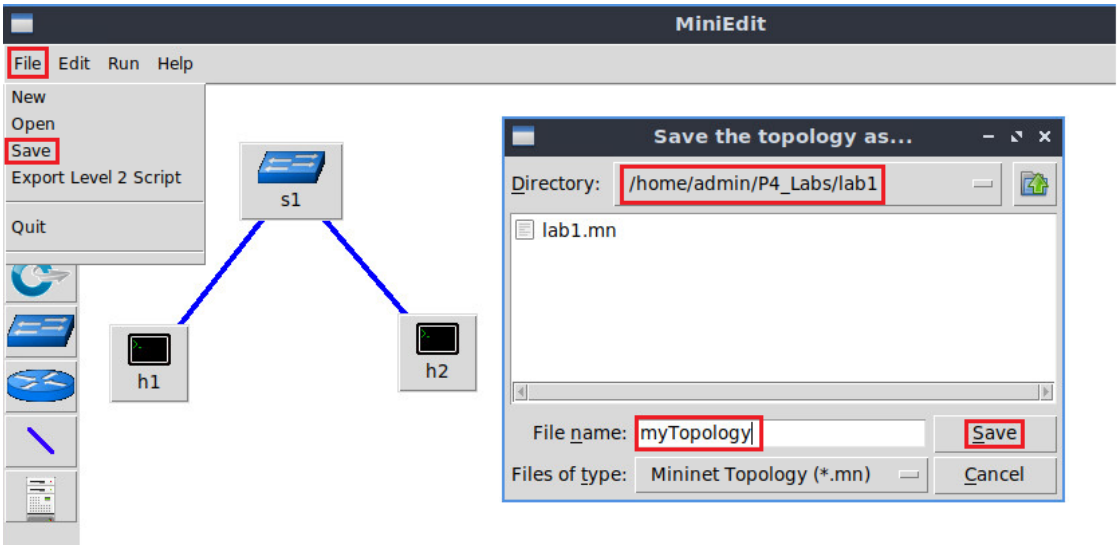


Figure 30. Saving the topology.

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.

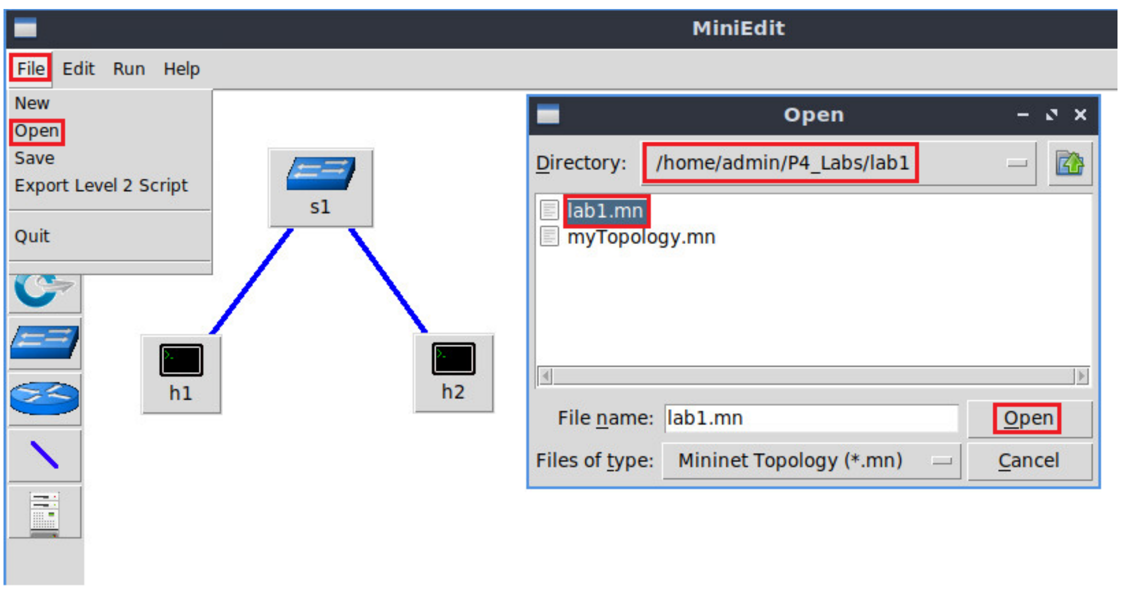


Figure 31. Opening a topology.

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.

UNIVERSITY OF SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 2: Introduction to P4 and BMv2

Document Version: 01-25-2022

Contents

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.

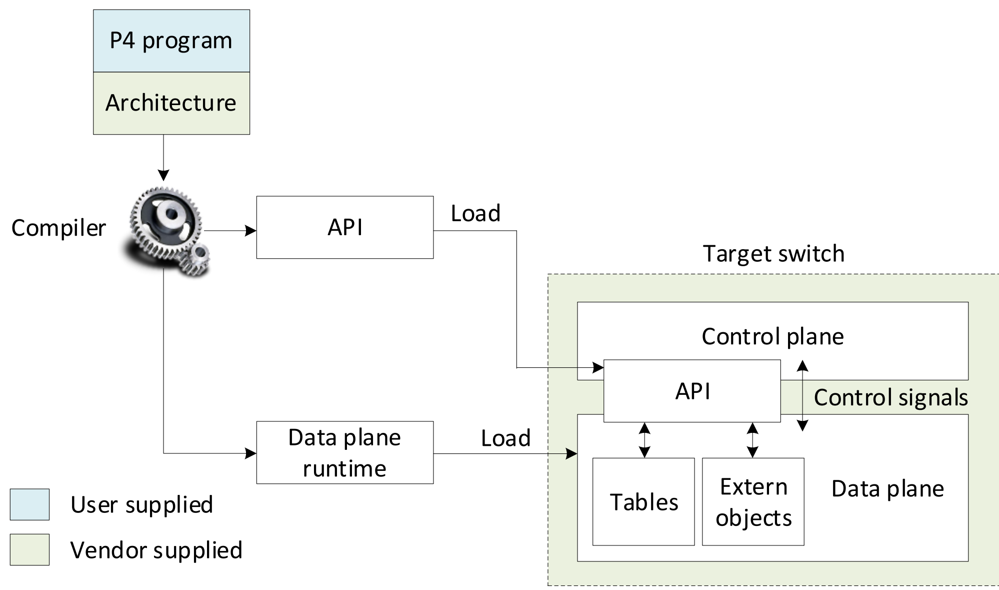


Figure 1. Generic workflow design. The compiler, the architecture model, and the target switch are provided by the vendor of the device. The P4 source code is customized by the user. The compiler generates a data plane runtime to be loaded into the target, and the APIs used by the control plane to communicate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

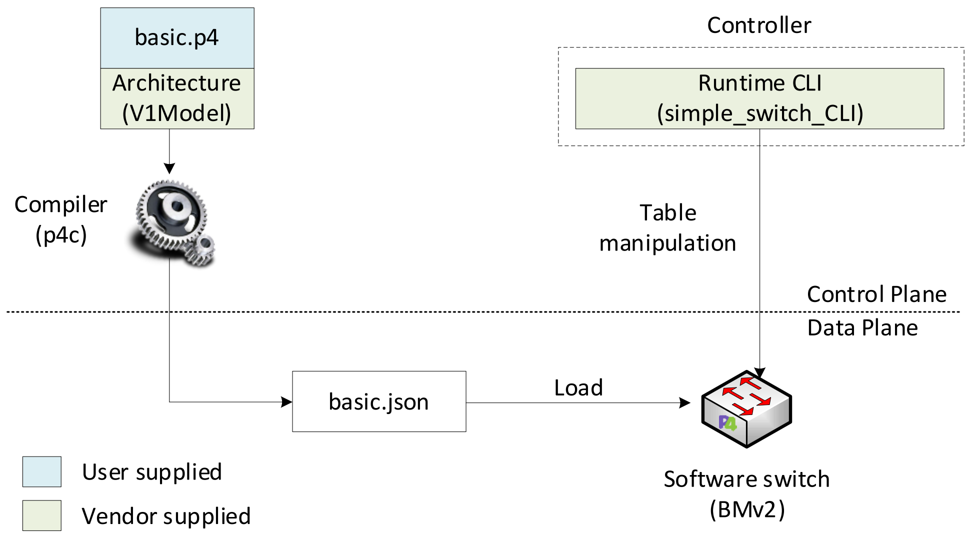


Figure 2. Workflow used in this lab series.

2 Lab topology

Let us 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.

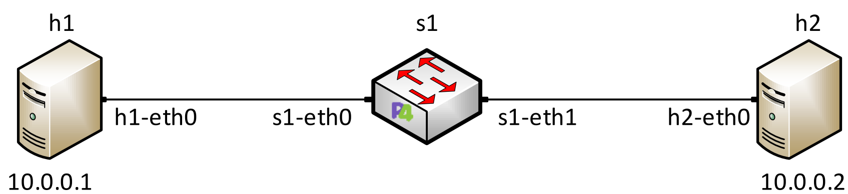


Figure 3. Lab topology.

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`.

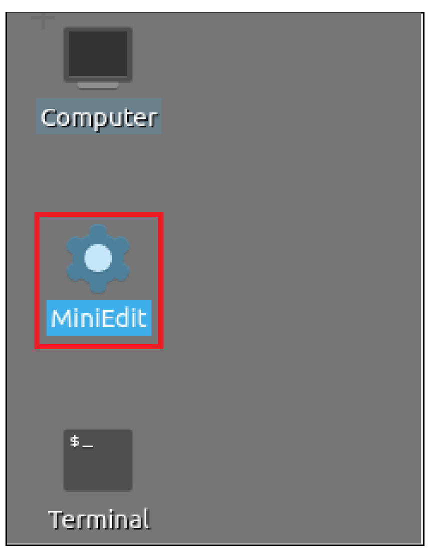


Figure 4. MiniEdit shortcut.

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*.

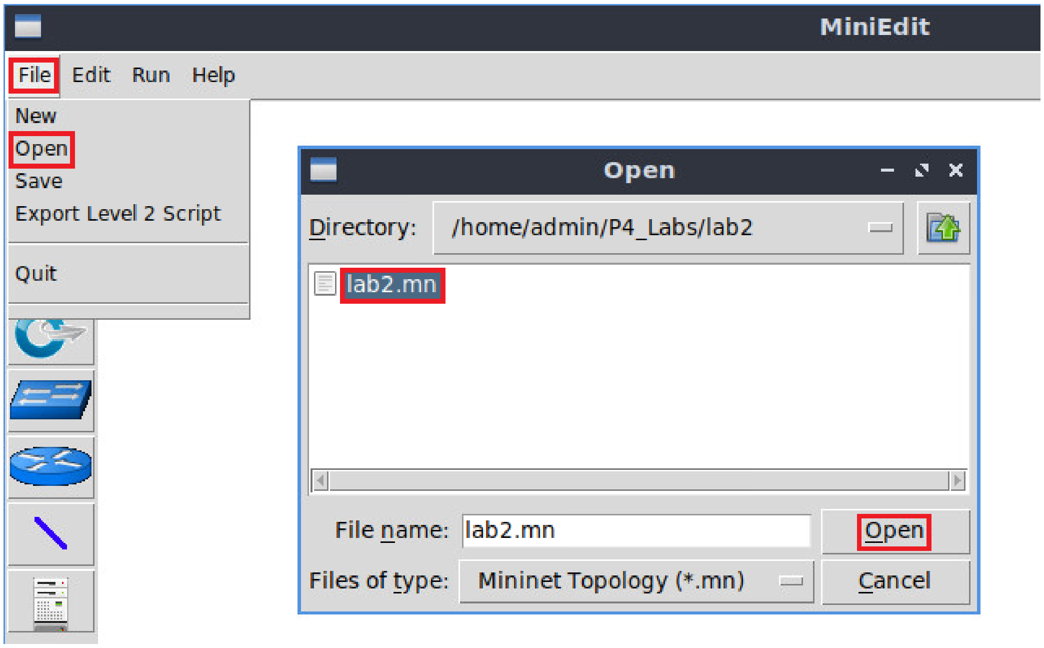


Figure 5. Opening a topology in MiniEdit.

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.

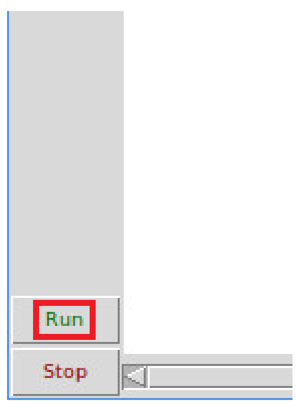


Figure 6. Running the emulation.

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.

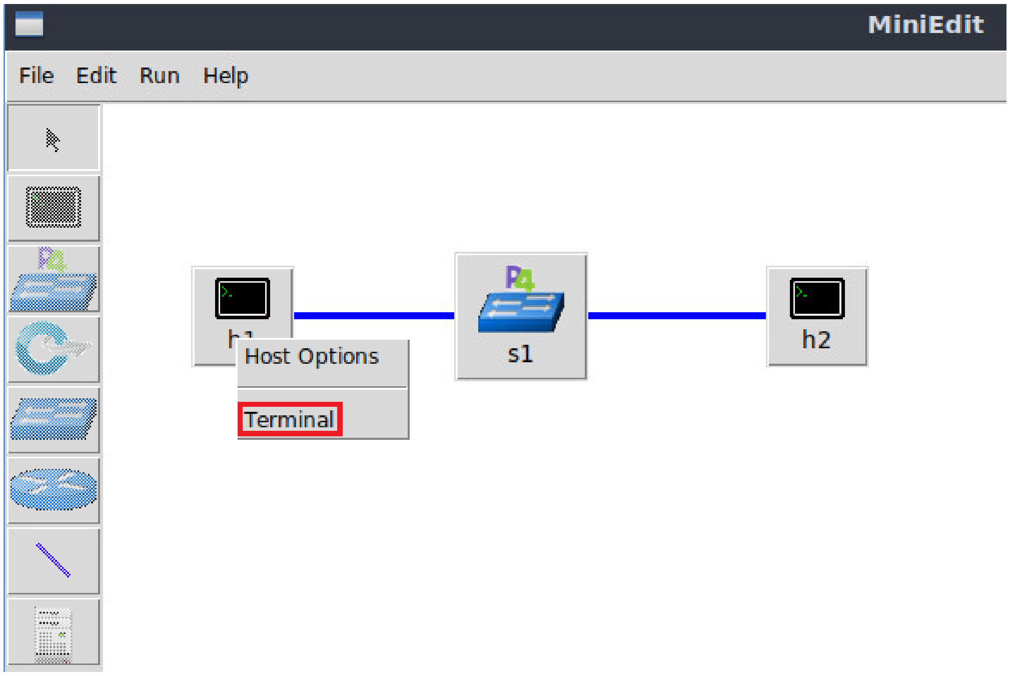


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

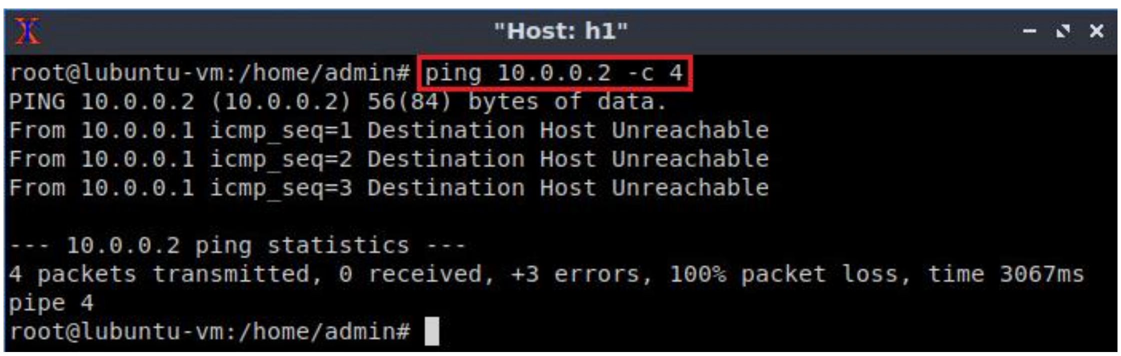


Figure 8. Performing a connectivity test between host h1 and host h2.

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.

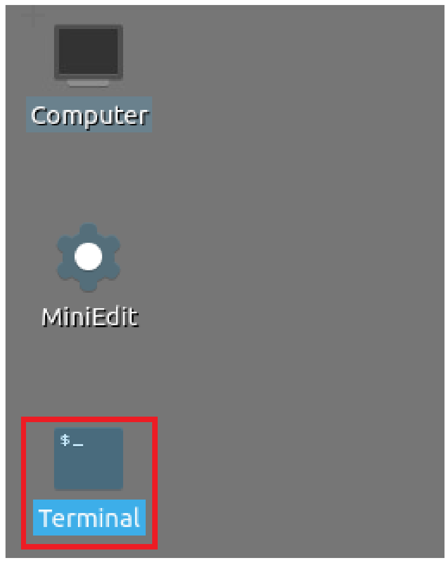


Figure 9. Shortcut to open a Linux terminal.

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
```

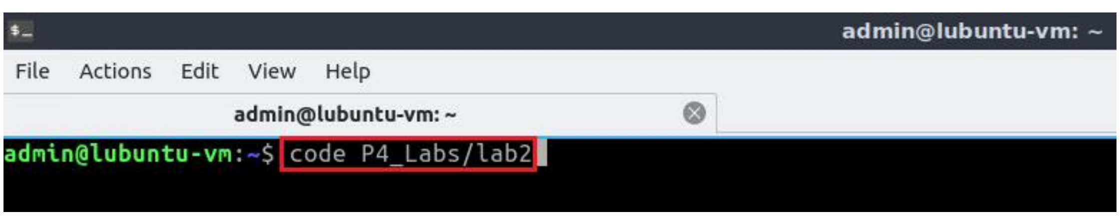


Figure 10. Launching the editor and opening the lab2 directory.

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.

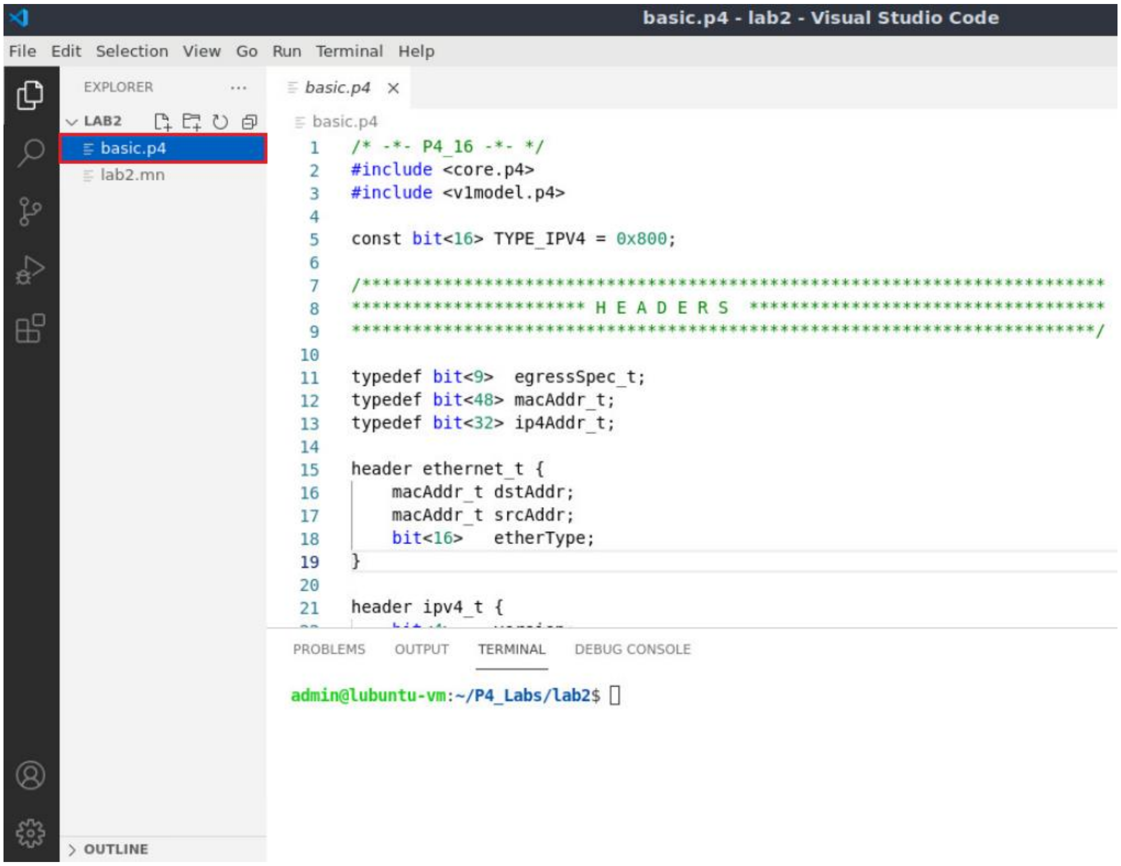


Figure 11. Opening the programming environment in VS Code.

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

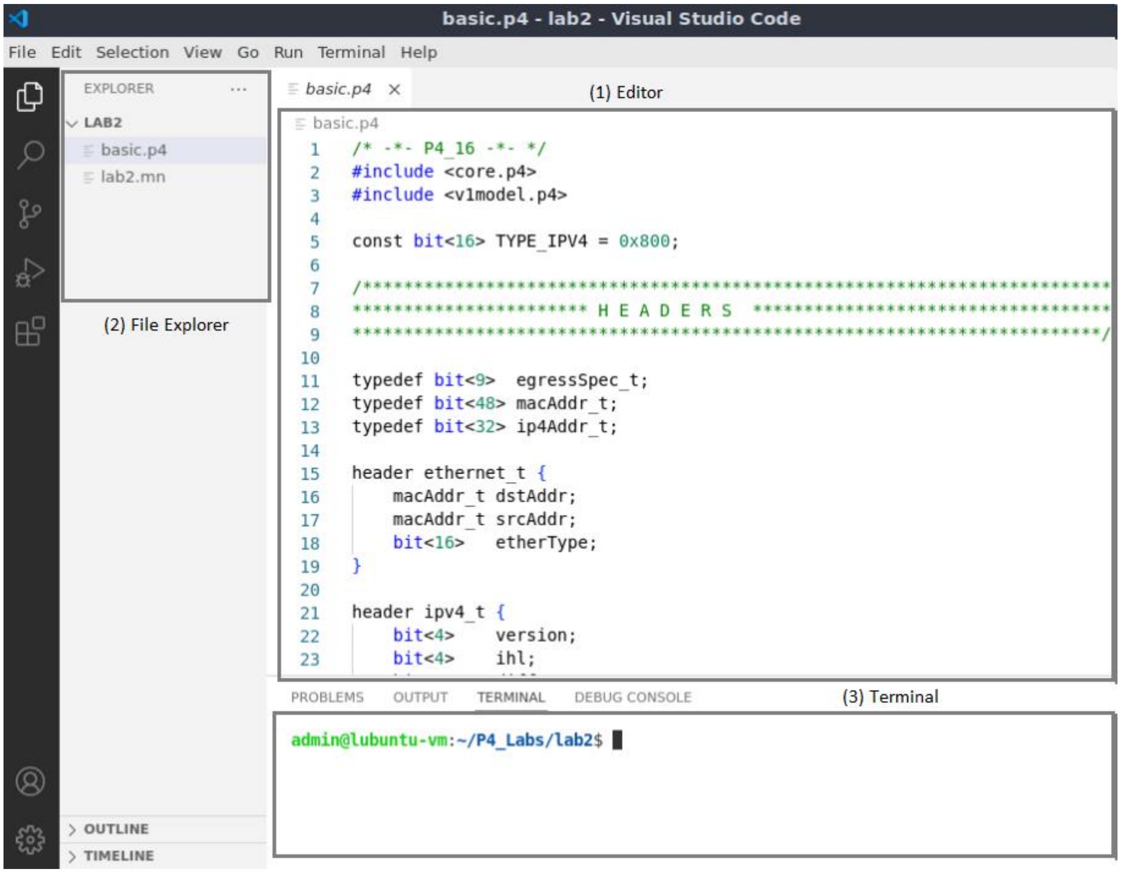


Figure 12. VS Code graphical interface components.

The VS Code interface consists of three main panels:

1. Editor: the editor panel displays 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., basic.p4 and *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
```

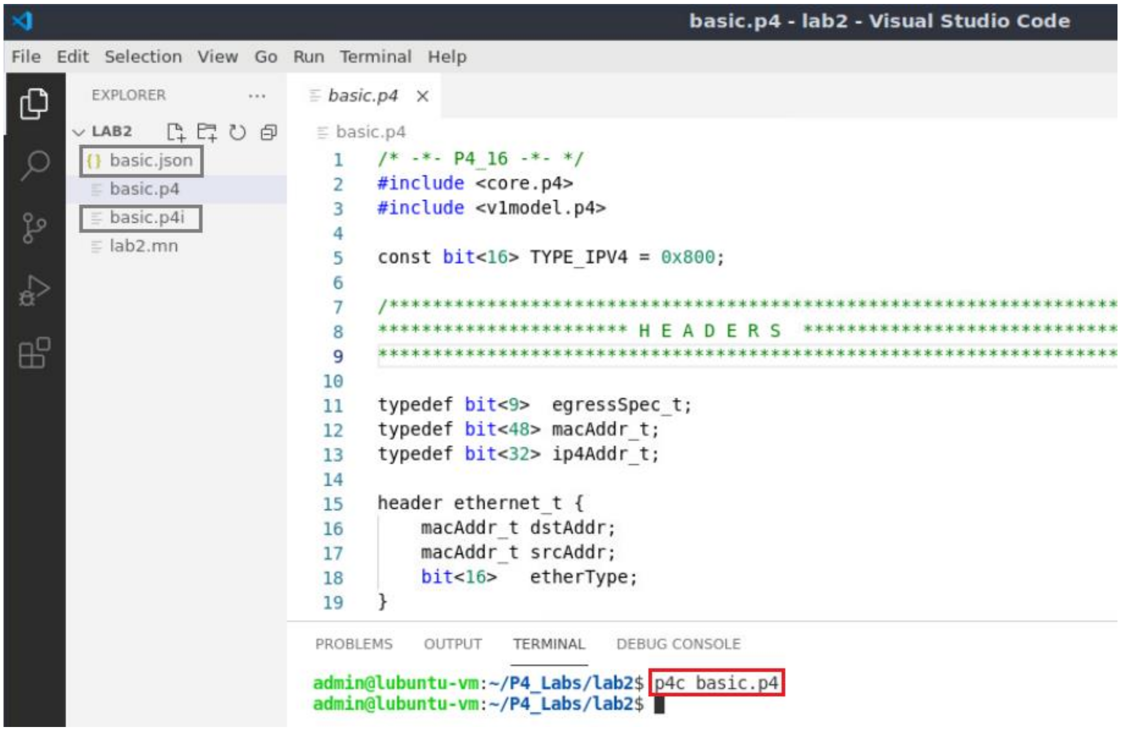


Figure 13. Compiling the P4 program using the VS Code terminal.

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
```

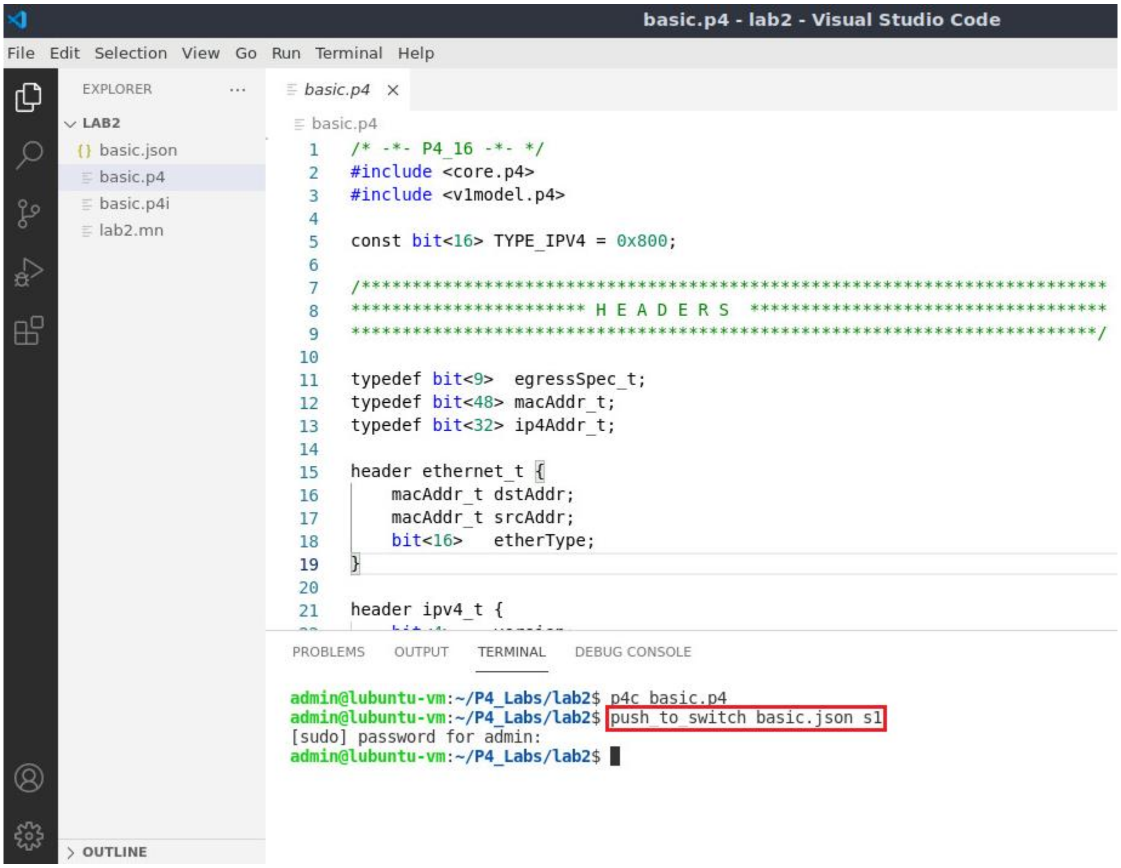


Figure 14. Downloading the compiled program to switch s1.

3.3 Verifying the configuration

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

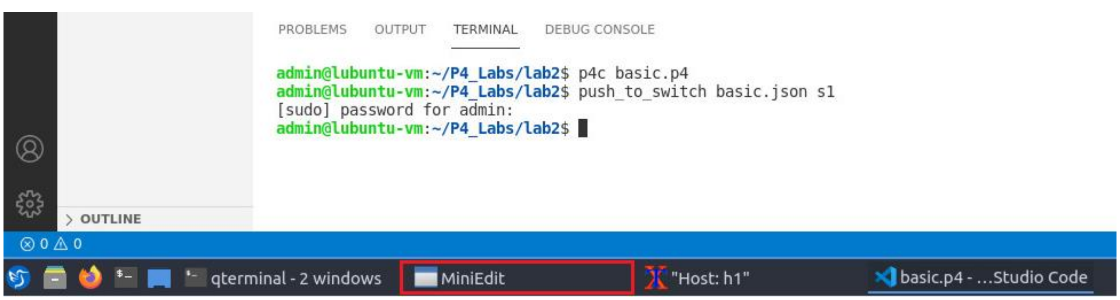


Figure 15. Maximizing the MiniEdit window.

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

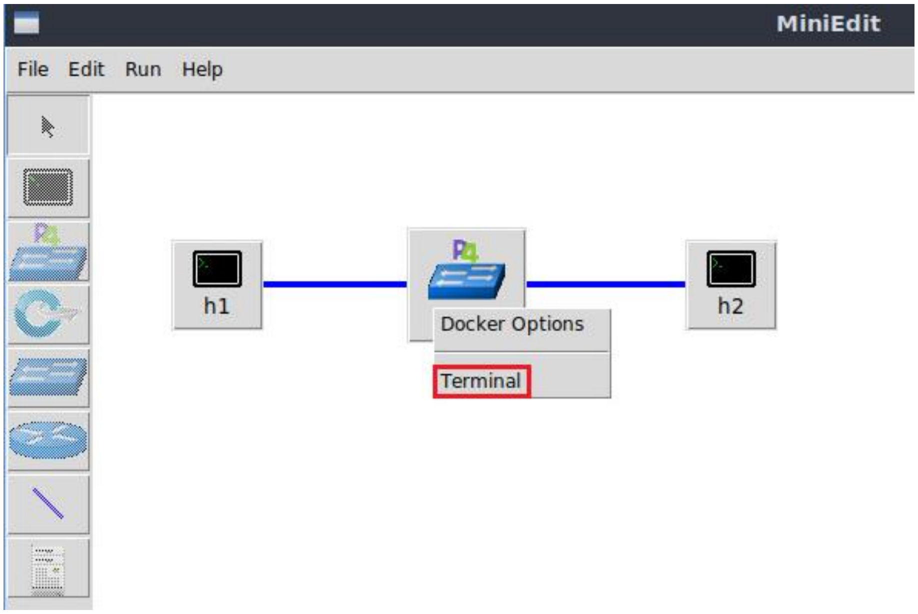


Figure 16. Starting the terminal on switch s1.

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
```

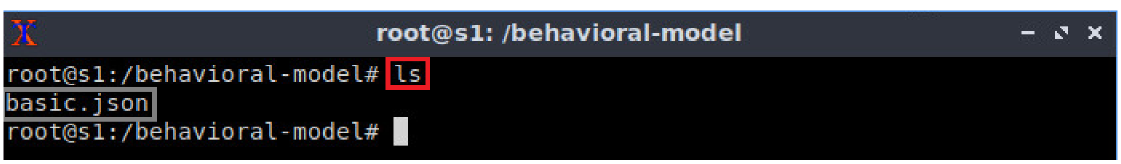


Figure 17. Displaying the contents of the current directory in the switch s1.

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
```

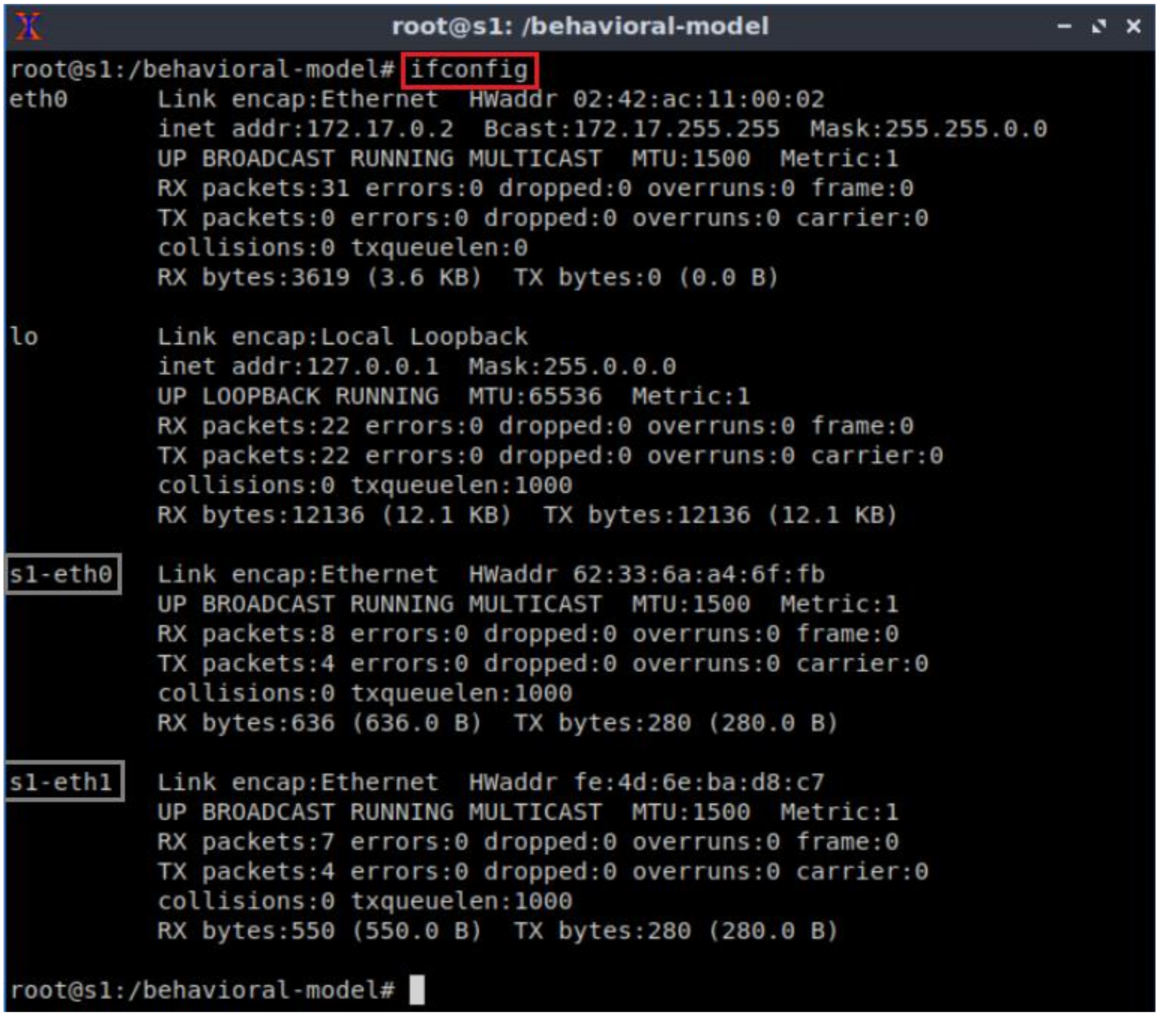


Figure 18. Displaying switch s1 interfaces.

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 &
```

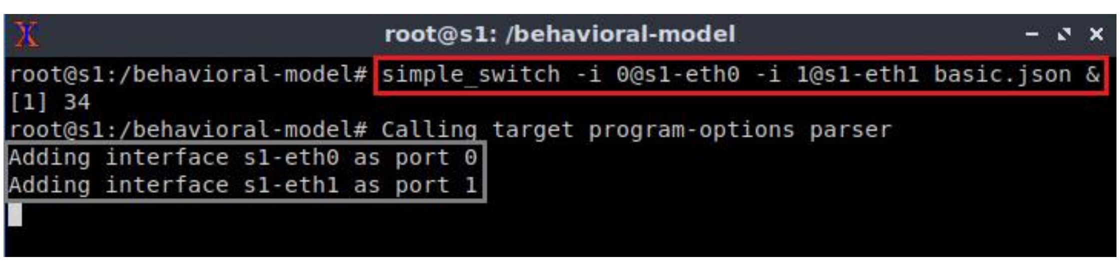


Figure 19. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

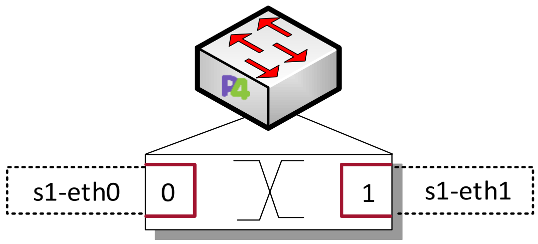


Figure 20. Ports 0 and 1 are mapped to the interfaces *s1-eth0* and *s1-eth1* of switch s1.

4.2 Loading the rules to the switch

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

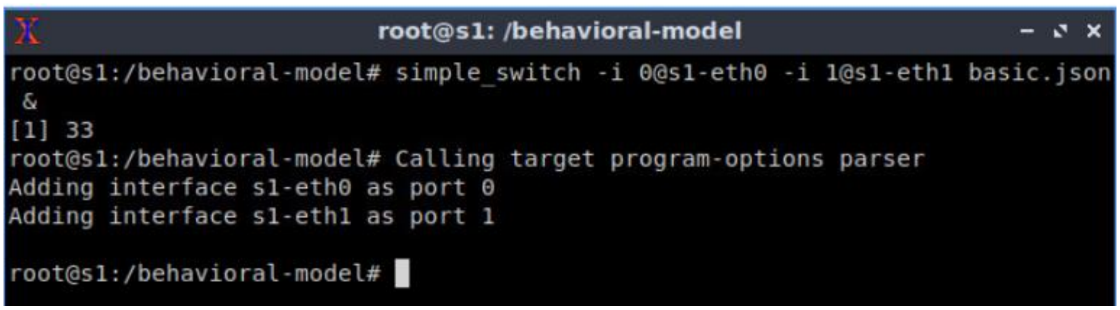


Figure 21. Returning to switch s1 CLI.

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

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

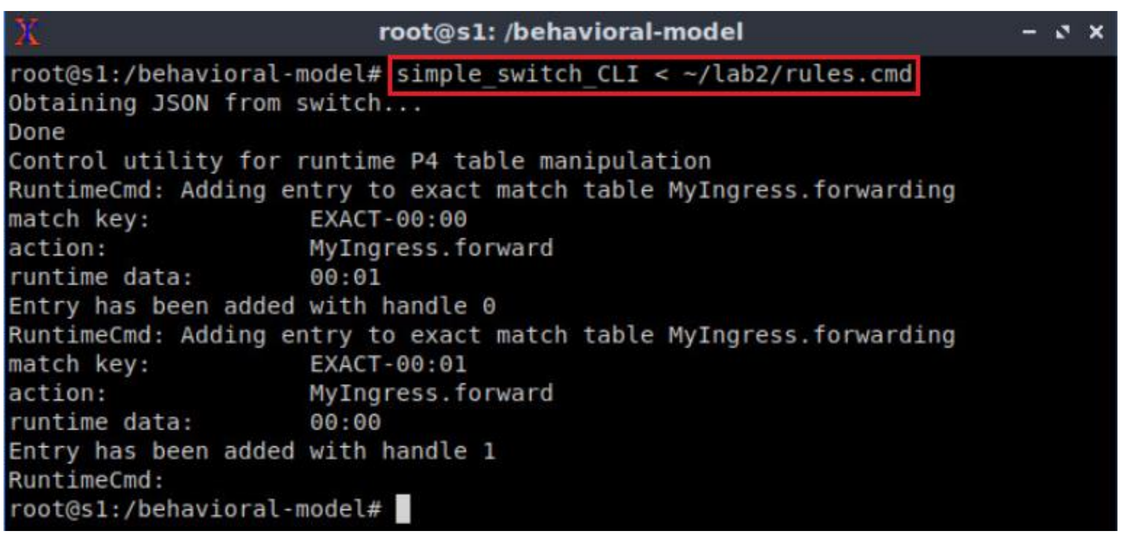


Figure 22. Loading table entries to switch s1.

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
```

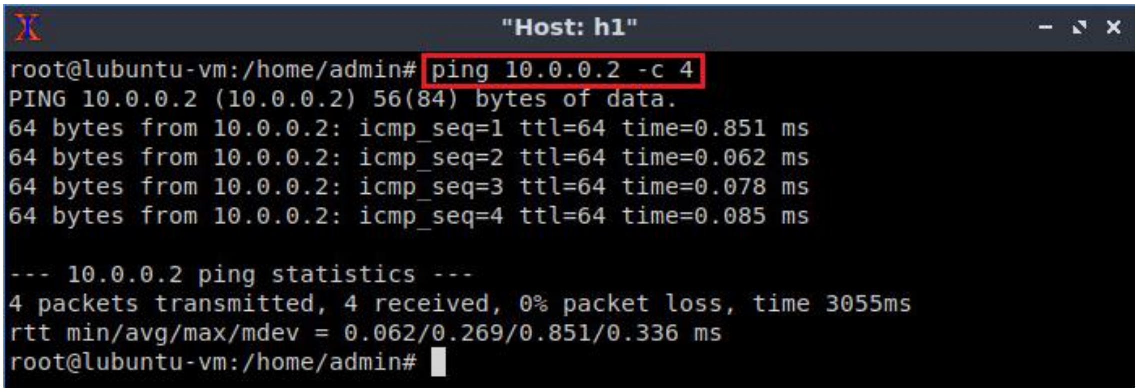


Figure 23. Performing a connectivity test between host h1 and host h2.

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^{TM} using Intel P4 Studio^{TM}*". 2021 P4 Workshop, ONF. [Online]. Available: https://tinyurl.com/yckzkybf.

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 3: P4 Program Building Blocks

Document Version: 01-25-2022

Contents

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).

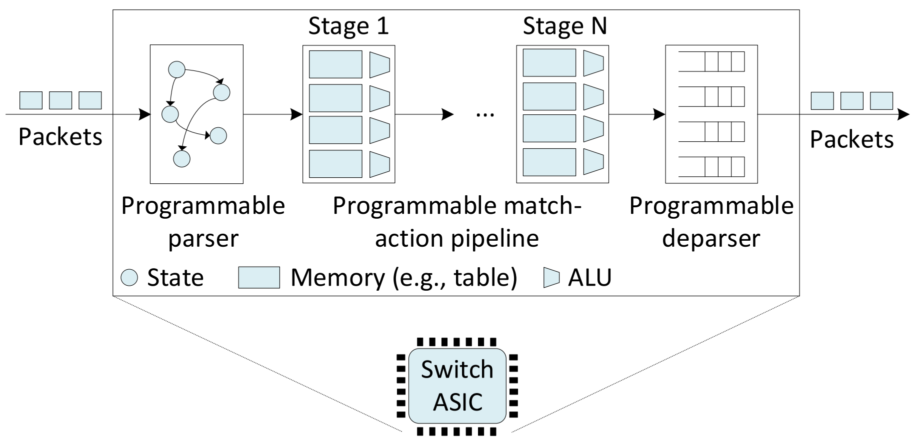


Figure 1. A PISA-based data plane.

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 of BMv2's simple_switch target[3].

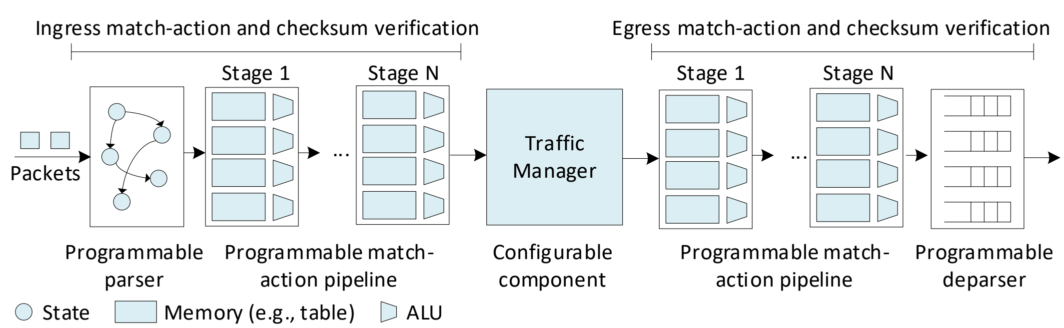


Figure 2. The V1Model architecture.

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.

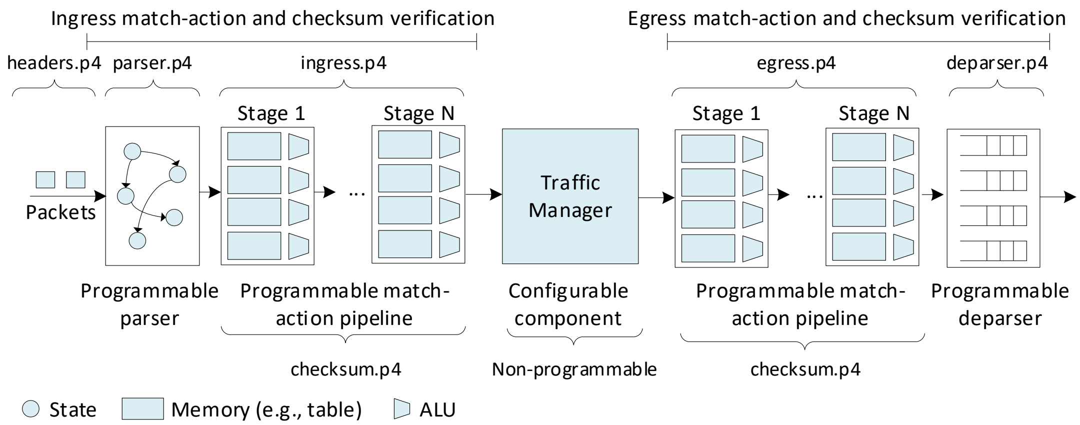


Figure 3. Mapping of P4 files to the V1Model's components.

2 Lab topology

Let us 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.

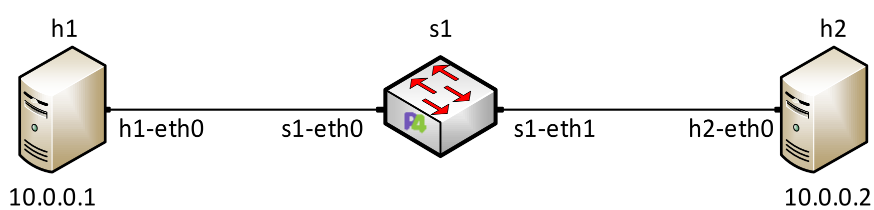


Figure 4. Lab topology.

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`.

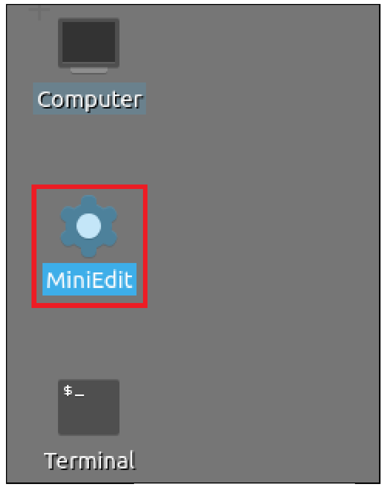


Figure 5. MiniEdit shortcut.

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.

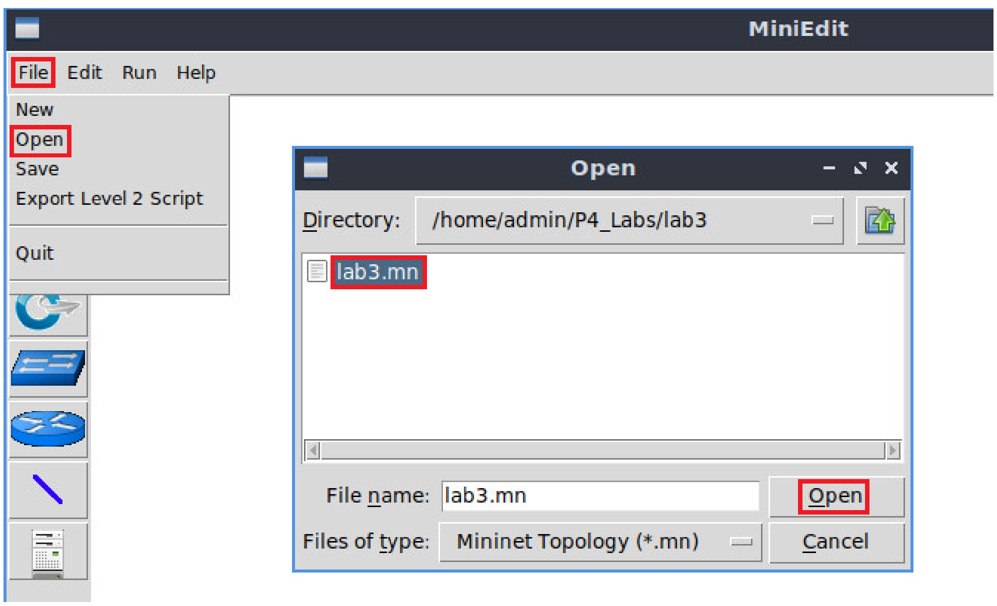


Figure 6. Opening a topology in MiniEdit.

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.

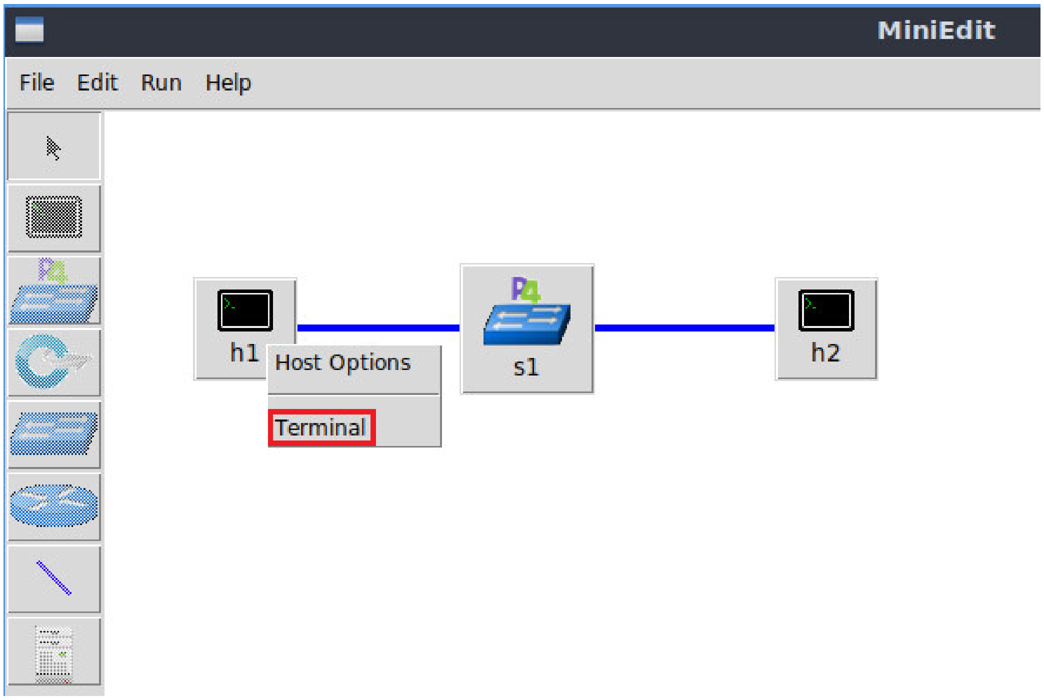


Figure 8. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

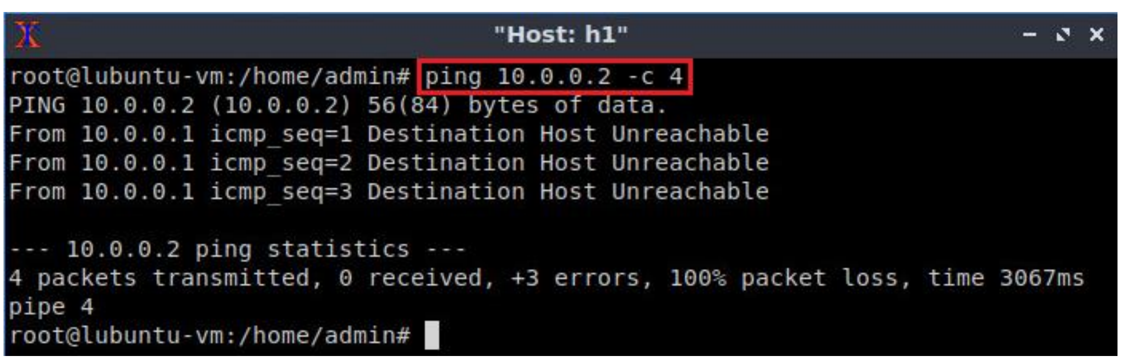


Figure 9. Performing a connectivity test between host h1 and host h2.

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.

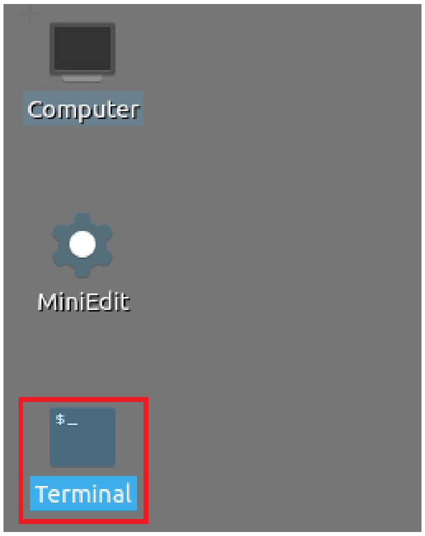


Figure 10. Shortcut to open a Linux terminal.

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/
```

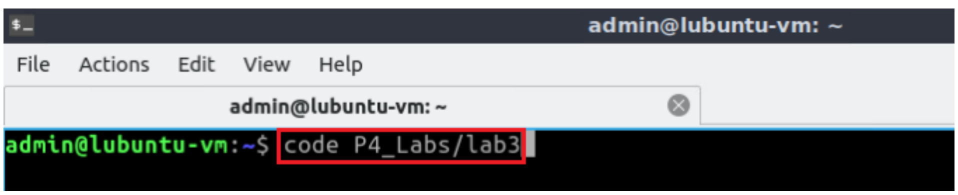


Figure 11. Launching the editor and opening the lab3 directory.

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.

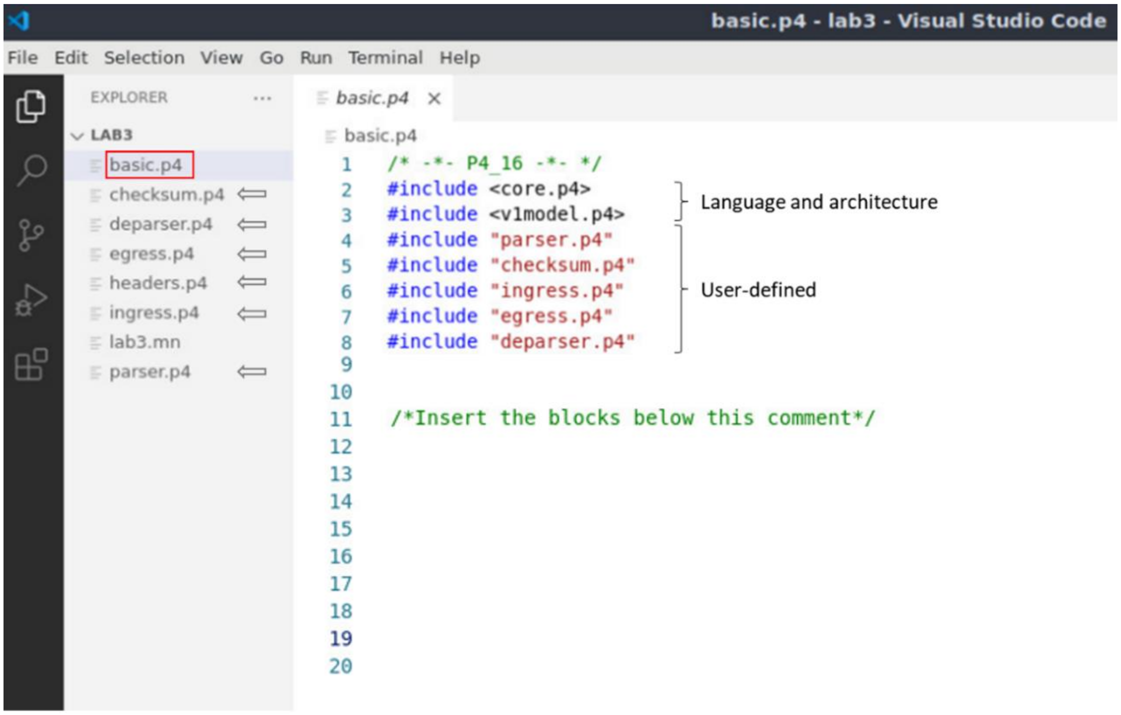


Figure 12. The main P4 file and how it includes other user-defined files.

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.

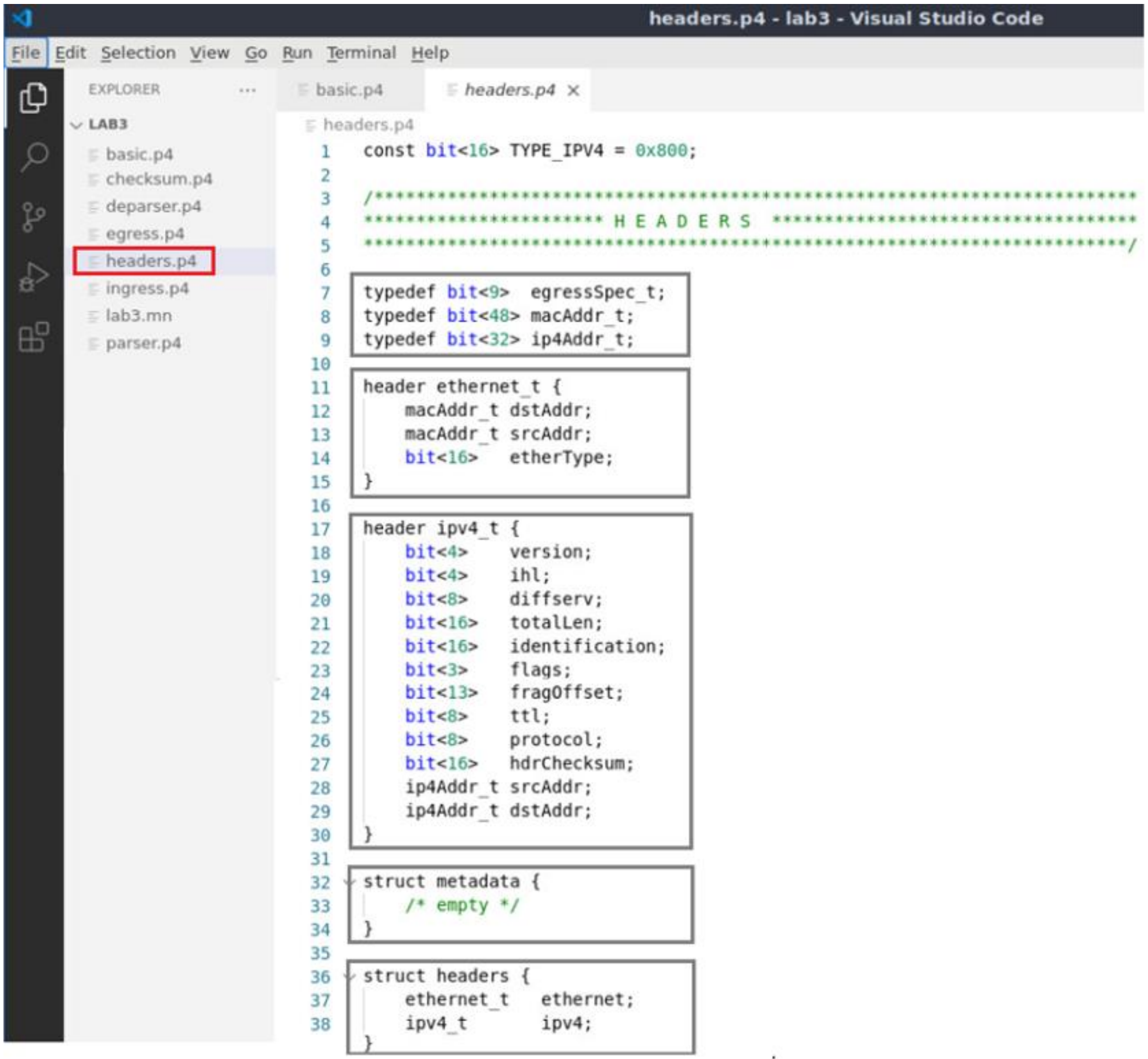


Figure 13. The defined headers.

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.

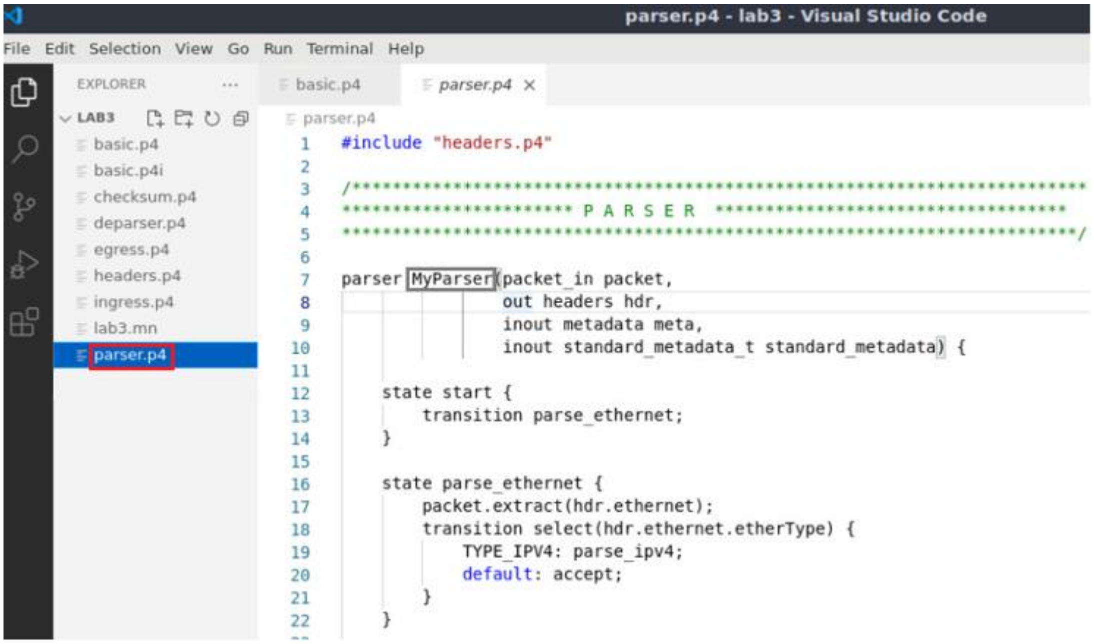


Figure 14. The parser implementation.

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.

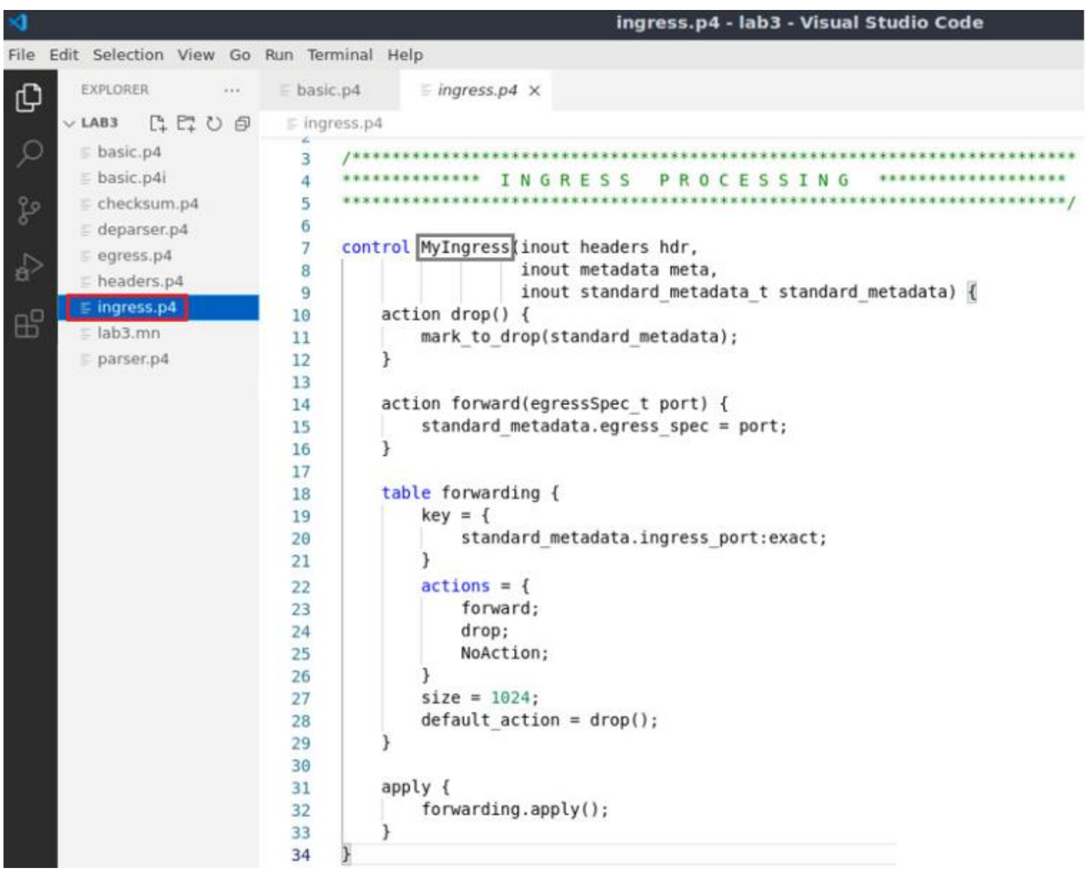


Figure 15. The ingress component.

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.

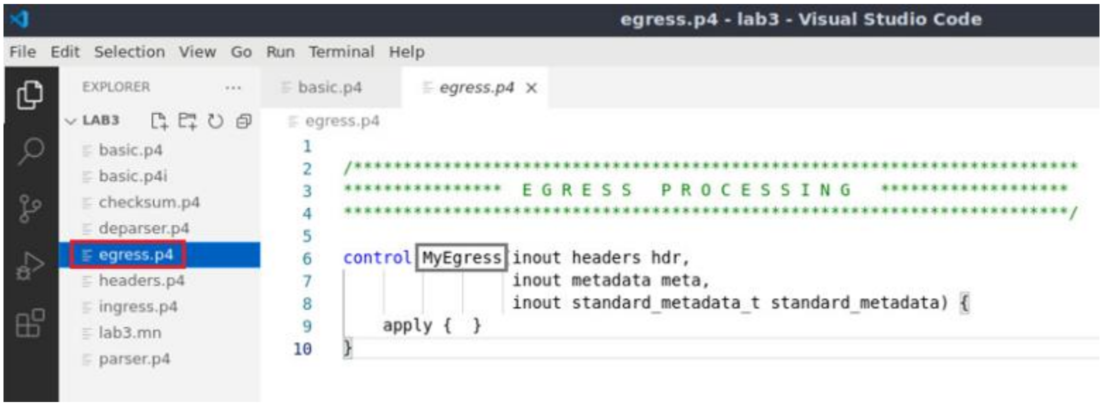


Figure 16. The egress component.

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.

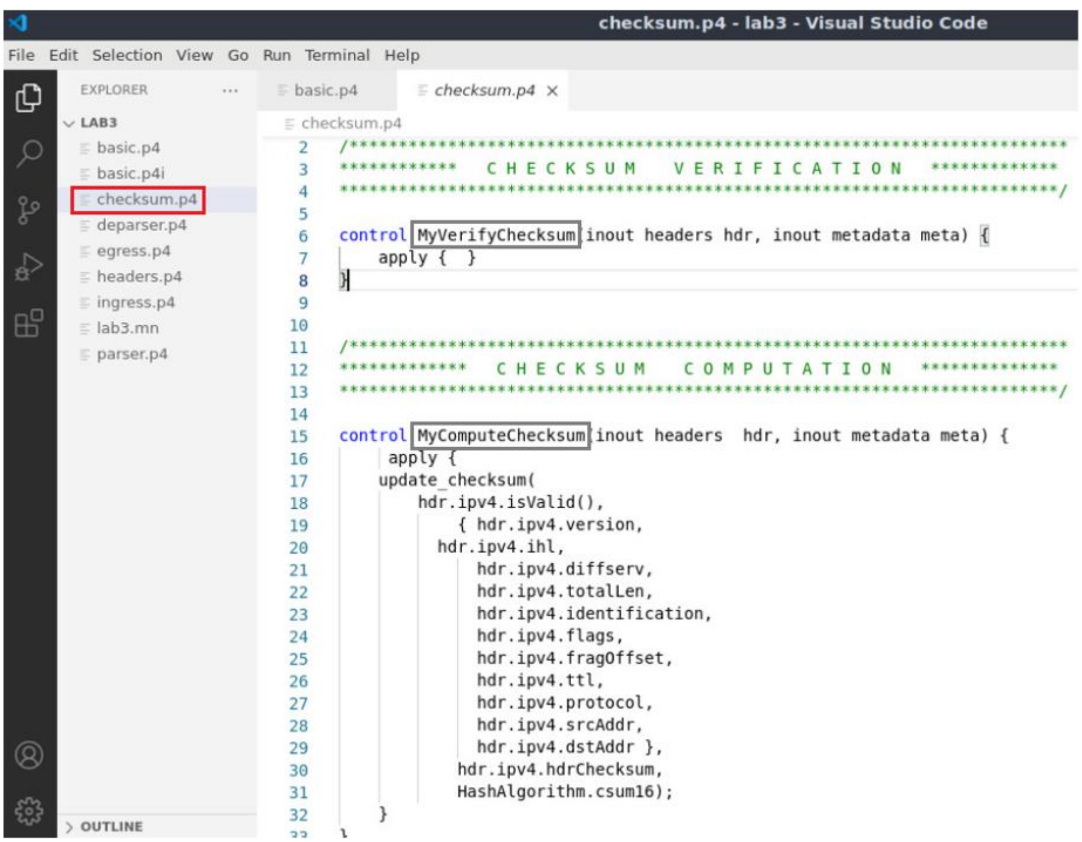


Figure 17. The checksum component.

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.

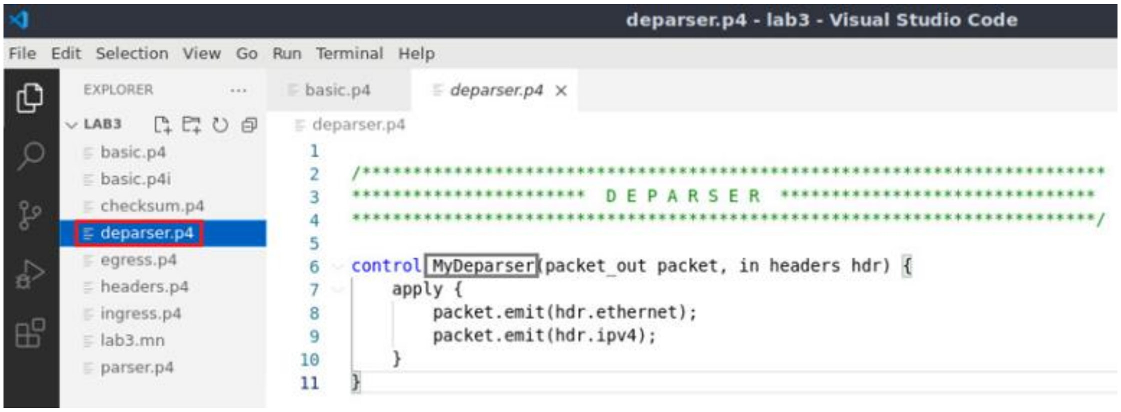


Figure 18. The deparser component.

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.

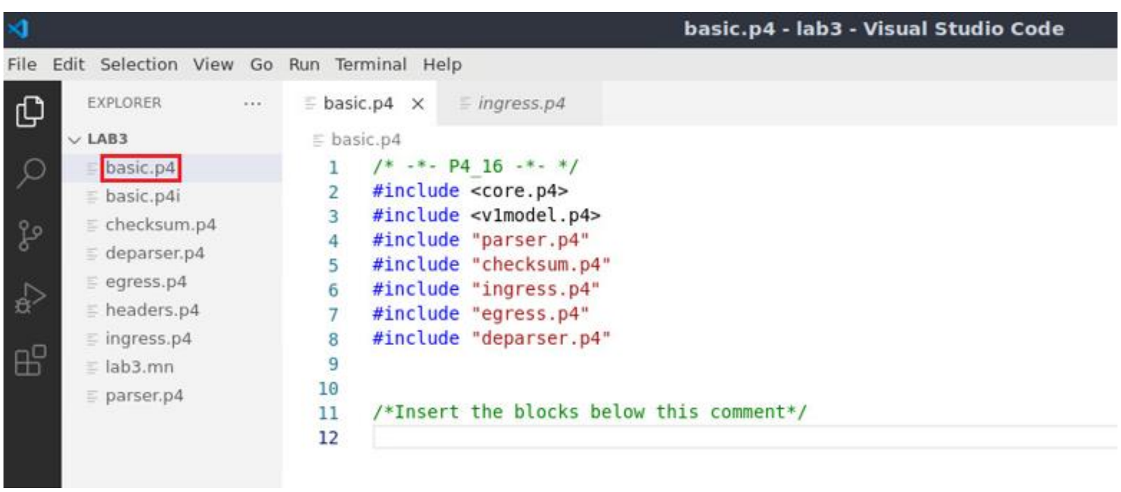


Figure 19. Selecting the *basic.p4* file.

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

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

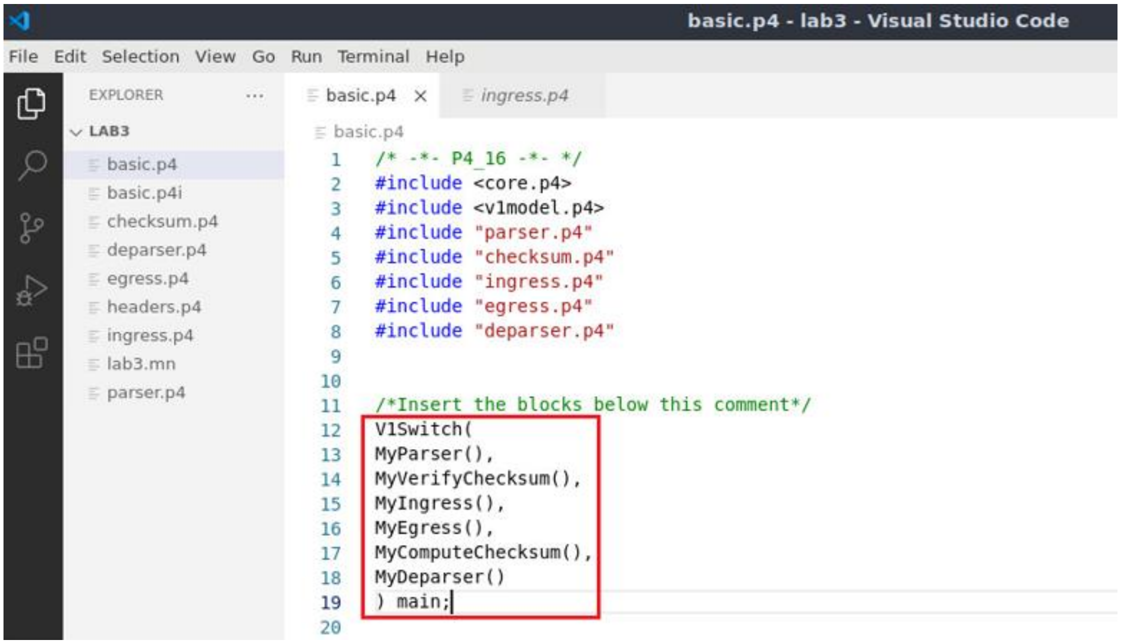


Figure 20. Writing the pipeline sequence in the *basic.p4* program

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
```

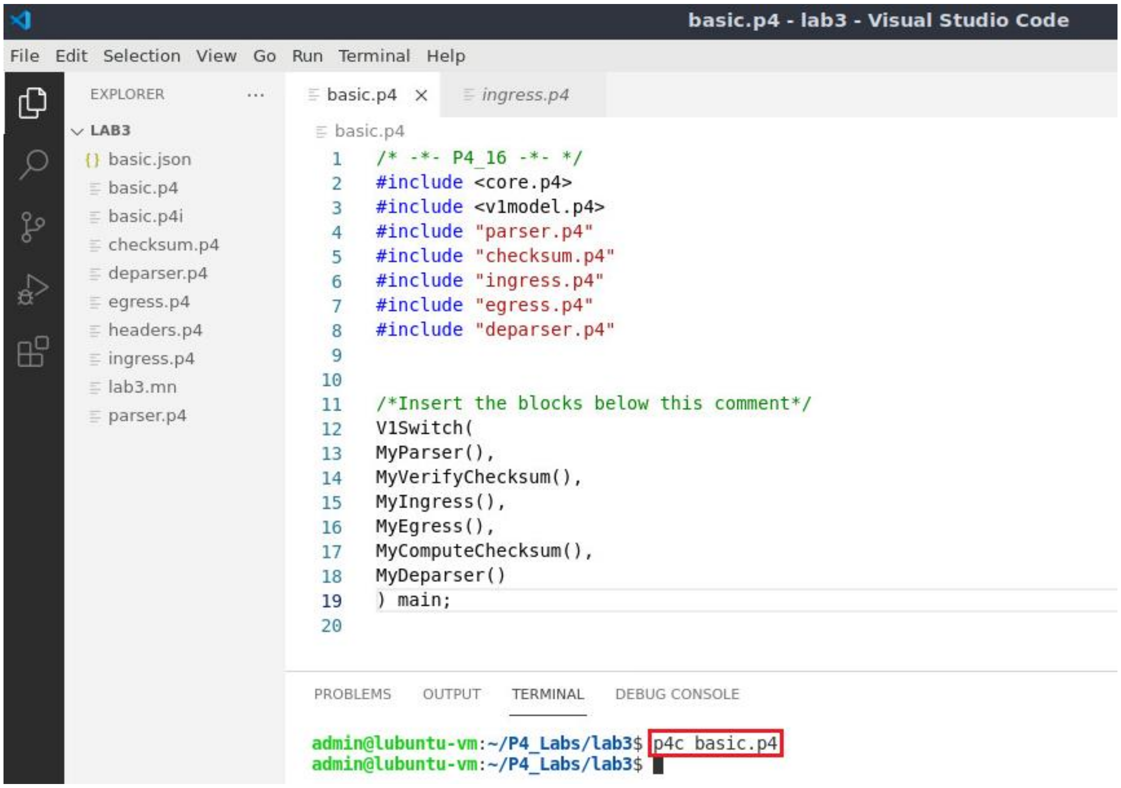


Figure 21. Compiling a P4 program.

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
```

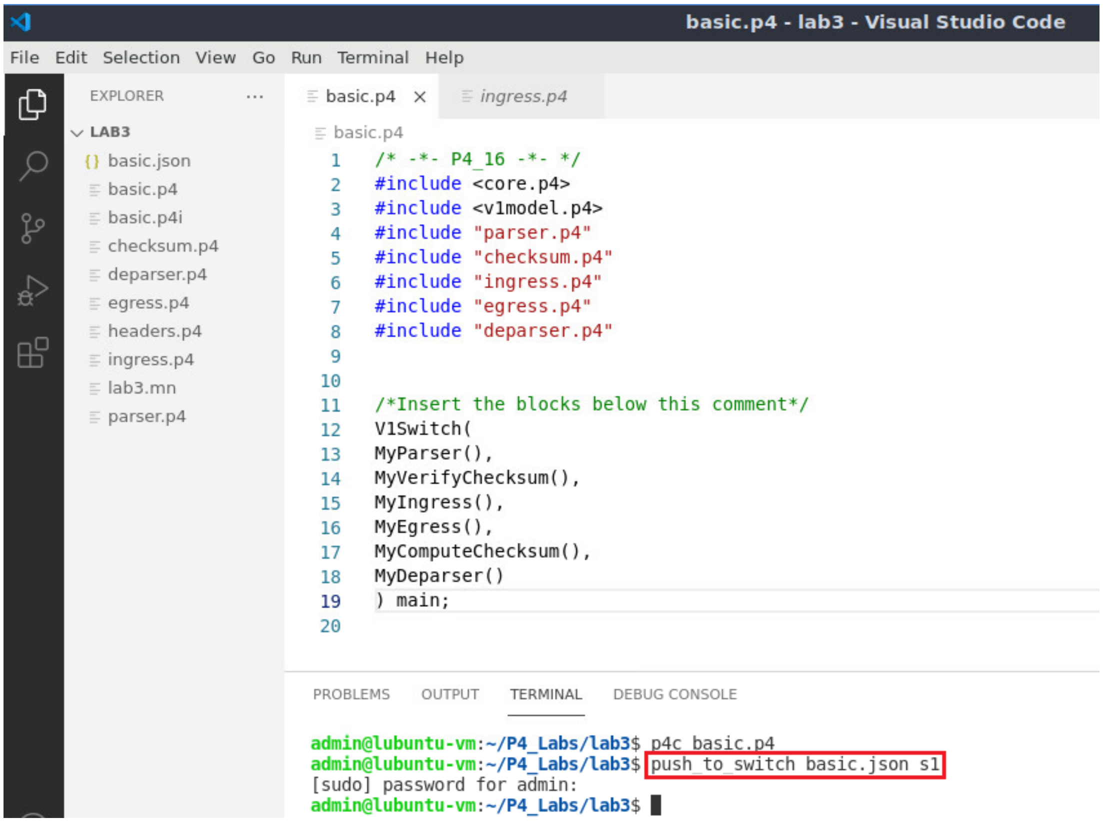


Figure 22. Downloading the P4 program to switch s1.

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*.

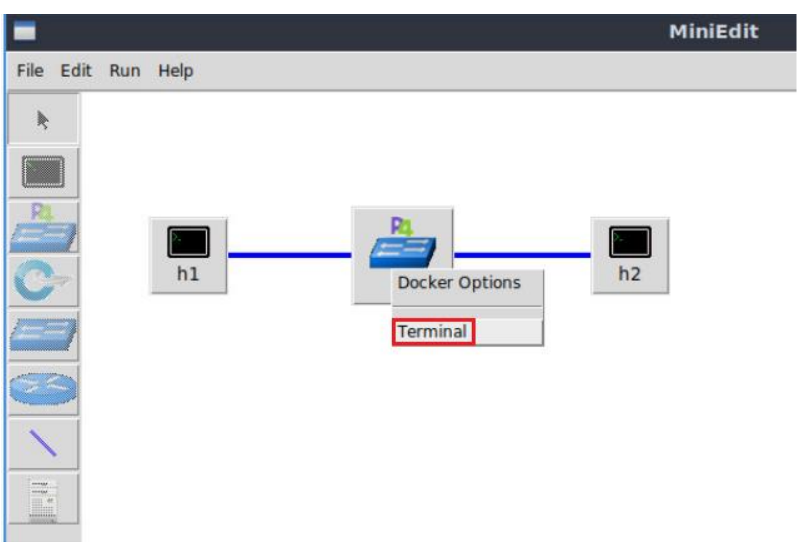


Figure 24. Starting the terminal on the switch.

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
```

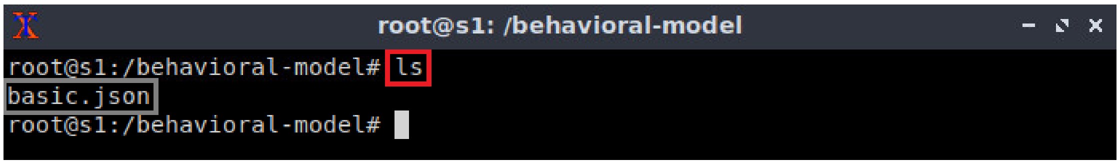


Figure 25. Displaying the contents of the current directory in the switch s1.

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
```

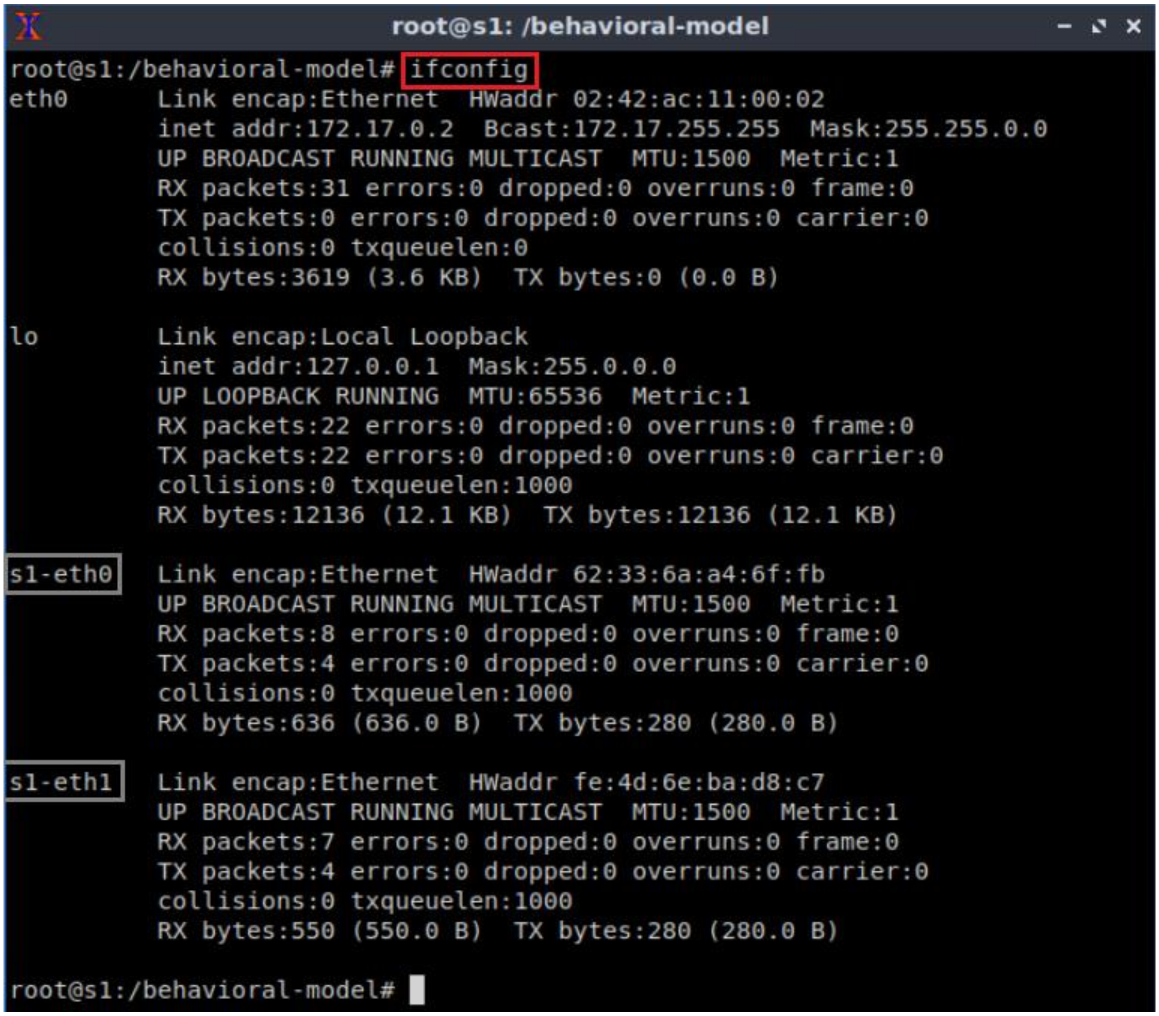


Figure 26. Displaying switch s1 interfaces.

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 &
```

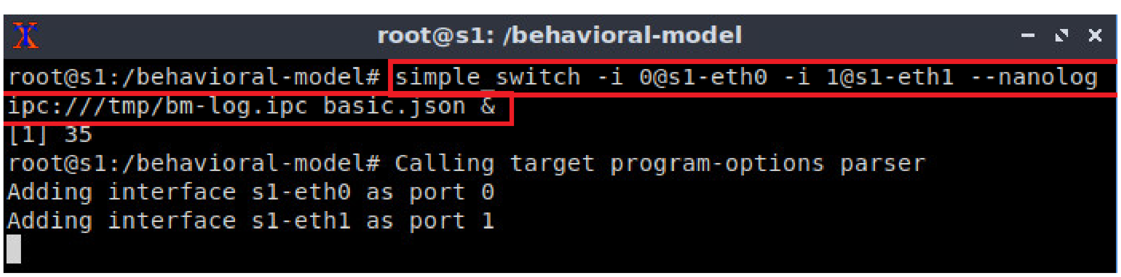


Figure 27. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

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

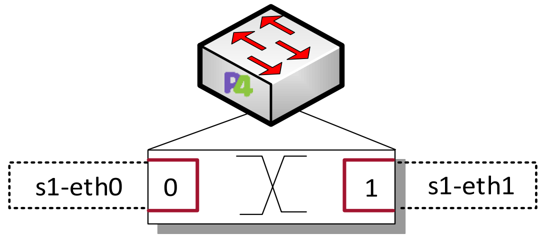


Figure 28. Mapping of the logical interface numbers (0, 1) to the Linux interfaces (*s1-eth0*, *s1-eth1*).

5.2 Loading the rules to the switch

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

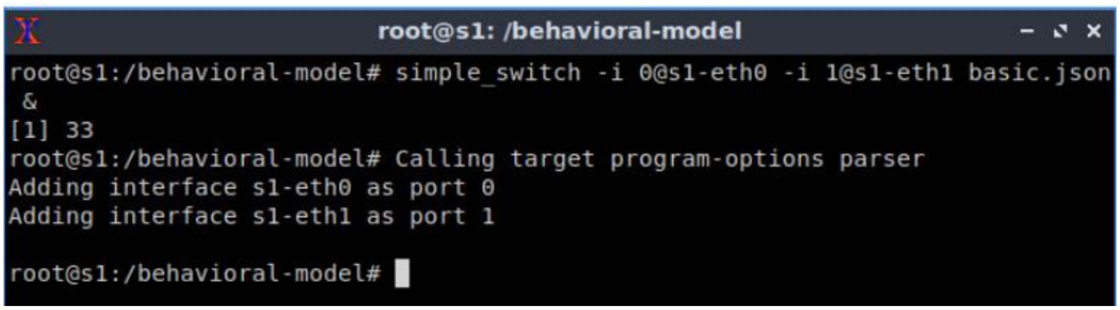


Figure 29. Returning to switch s1 CLI.

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

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

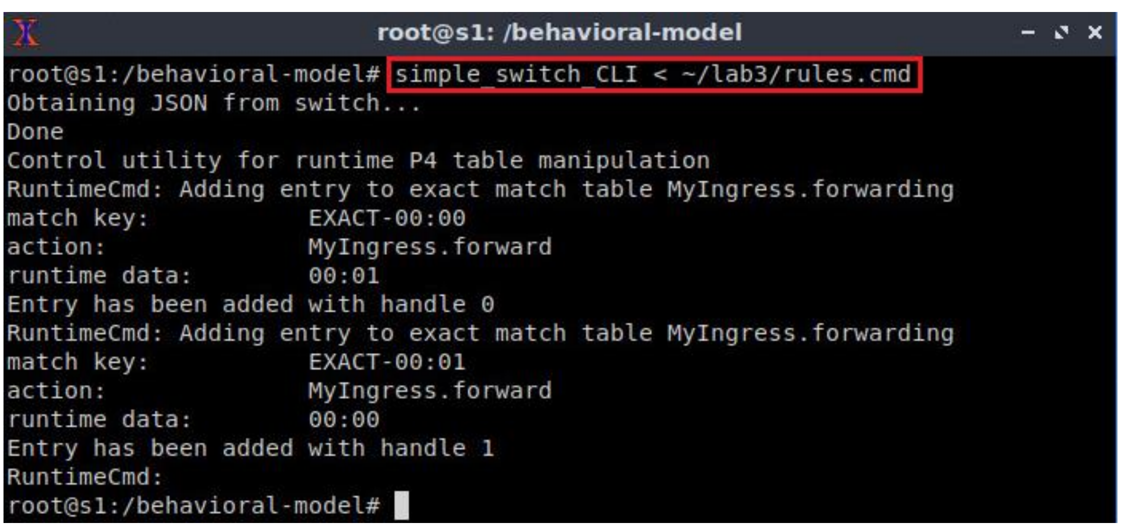


Figure 30. Loading the forwarding table entries into switch s1.

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
```

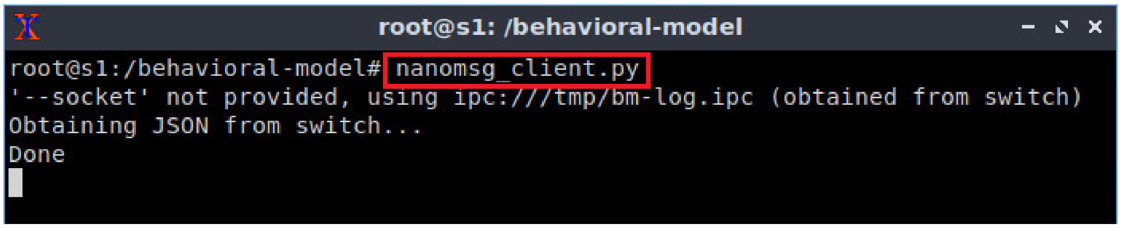


Figure 31. Displaying switch s1 logs.

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

```
recv.py
```

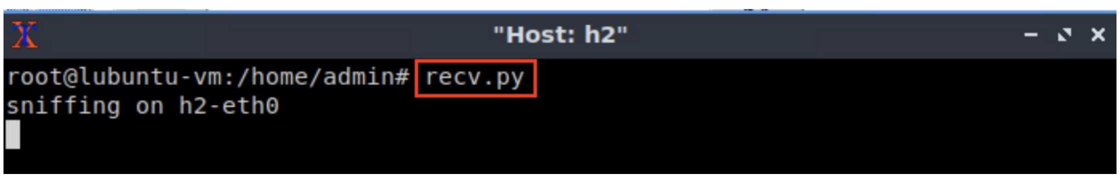


Figure 32. Listening for incoming packets in host h2.

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
```

```
"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       = 00:00:00:00:00:02
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 30
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = hopopt
     chksum    = 0x66dd
     src       = 10.0.0.1
     dst       = 10.0.0.2
     \options   \
###[ Raw ]###
        load      = '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.

```
root@s1: /behavioral-model
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 34. Inspecting the logs in switch s1.

The figure above shows the processing logic as the packet enters switch s1. The packet arrives on port 0 (`port_in: 0`), then the parser starts extracting the headers. After the parsing is done, the packet is processed in the ingress and in the egress pipelines. Then, the checksum update is executed and the deparser reassembles and emits the packet using port 1 (`port_out: 1`).

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

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

References

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

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 4: Defining and Processing Custom Headers

Document Version: 03-01-2022

Contents

Overview

This lab shows the steps to define, parse, and process a packet that contains a custom header. This custom header includes the ingress port, the egress port, and the packet length. Such information is available in the V1Model standard metadata, which is used to interface the fixed-function components of the target switch (BMv2) with the P4 code. At the end of this lab, the user will learn to insert the switch's metadata into a custom header and read that information from an end host.

Objectives

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

1. Define a custom header.
2. Parse a custom header.
3. Include metadata into a custom header.
4. Program the egress 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: Introduction to the V1Model standard metadata.
2. Section 2: Lab topology.
3. Section 3: Defining and parsing a custom header.
4. Section 4: Processing a custom header.
5. Section 5: Loading the P4 program.
6. Section 6: Configuring switch s1.
7. Section 7: Testing and verifying the P4 program.

1 Introduction to the V1Model standard metadata

When a packet arrives at a switch's ingress port, the information contained in the headers' field is available for processing. With this information, the switch can determine how to forward a packet. For example, a switch can use the destination MAC address to determine which port a packet will take to reach another host. Note that the MAC addresses are part of the Ethernet frame. On the other hand, information such as the ingress port and the packet length are not available in the packet headers. They are part of the switch's metadata.

The P4 language provides a data structure that contains the packet metadata. Figure 1 shows the metadata available in the V1Model, known as standard metadata. The metadata includes the ingress port (see line 3), egress port (see line 5), packet length (see line 10), and others. A programmer can use the switch's metadata to build custom programs. Note that the metadata available in a P4 target is vendor-specific.

```
1: /************************STANDARD METADATA*************************/
2:  standard_metadata_t {
3:        bit<9> ingress_port;
4:        bit<9> egress_spec;
5:        bit<9> egress_port;
6:        bit<32> clone_spec;
7:        bit<32> instance_type;
8:        bit<1> drop;
9:        bit<16> recirculate_port;
10:       bit<32> packet_length;
11:       bit<32> enq_timestamp;
12:       bit<19> enq_qdepth;
13:       bit<32> deq_timedelta;
14:       bit<19> deq_qdepth;
15:       bit<48> ingress_global_timestamp;
16:       bit<48> egress_global_timestamp;
17:       bit<32> lf_field_list;
18:       bit<16> mcast_grp;
19:       bit<32> resubmit_flag;
20:       bit<16> egress_rid;
21:       bit<1> checksum_error;
22:       bit<32> recirculate_flag;
23: }
```

Figure 1. The V1Model standard metadata.

Figure 2 represents the V1Model pipeline components. This figure shows that the packet and its metadata traverse the pipeline through the data bus and the metadata bus, respectively. Note that depending on the metadata that the P4 program is using, there are values such as the egress timestamp (see line 16 in Figure 1) or the queue depth (see line 12 in Figure 1) that are only available at the egress block after the packet passed through the traffic manager. This characteristic implies that the programmer must process the egress timestamp and queue depth at the egress pipeline.

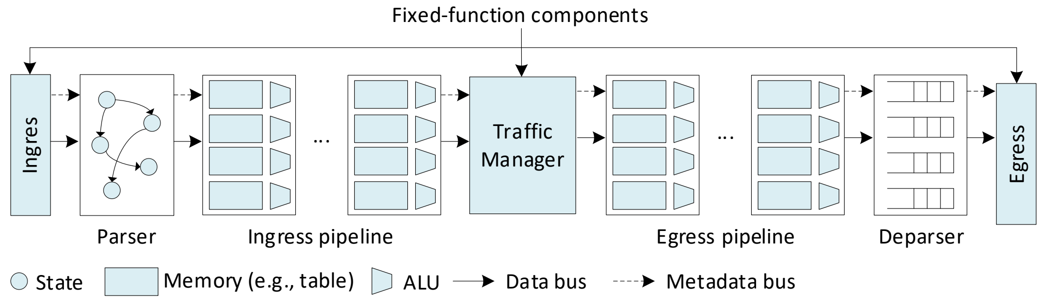


Figure 2. The V1Model architecture.

2 Lab topology

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

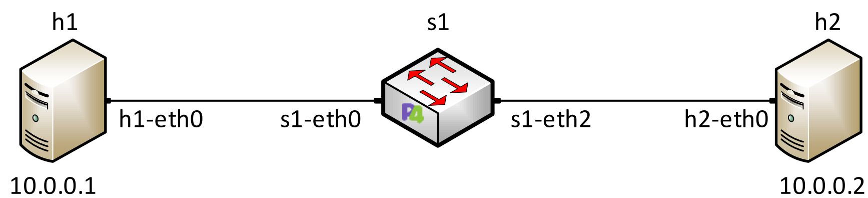


Figure 3. Lab topology.

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`.

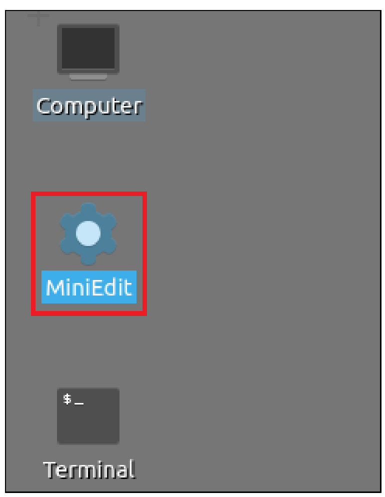


Figure 4. MiniEdit shortcut.

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

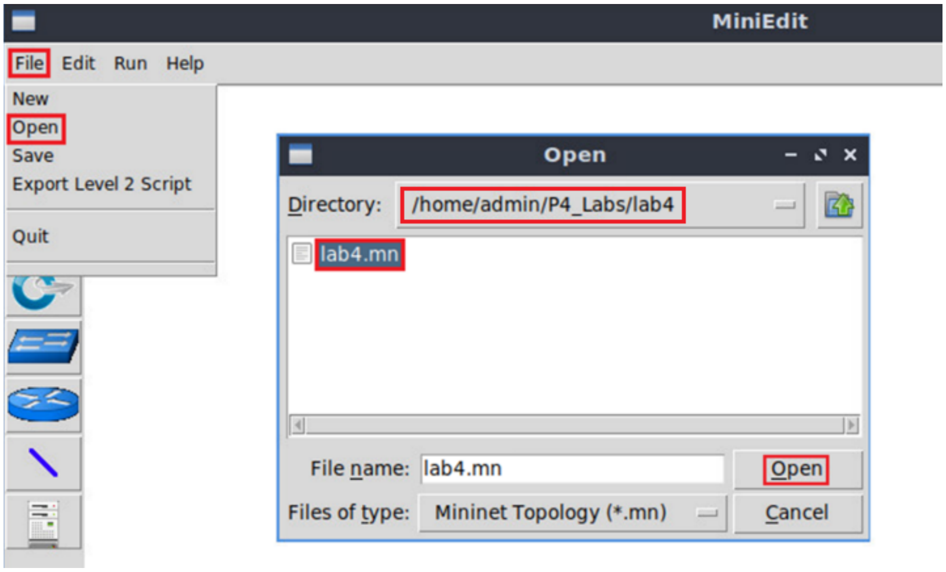


Figure 5. MiniEdit's *Open* dialog.

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.

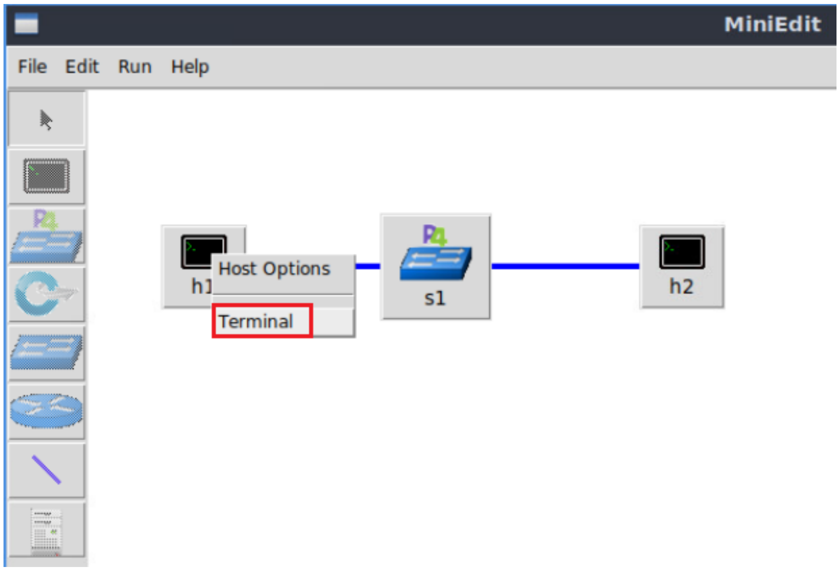


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

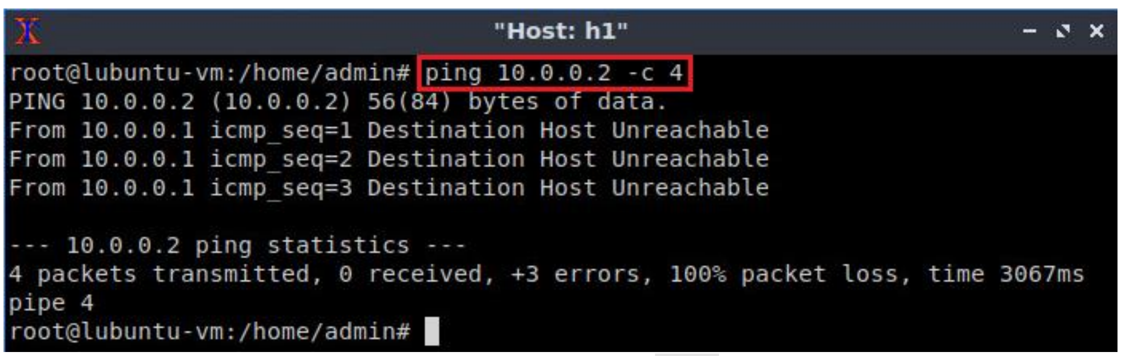


Figure 8. Connectivity test using `ping` command.

The figure above shows unsuccessful connectivity between host h1 and host h2. This result happens because there is no P4 program loaded on the switch.

3 Defining and parsing a custom header

In this section, you will learn how to create a custom header, which will contain the ingress port, egress port, and packet length values. Then, you will specify the parser's behavior to extract the fields from the Ethernet, the IPv4, and the custom headers.

3.1 Loading the programming environment

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

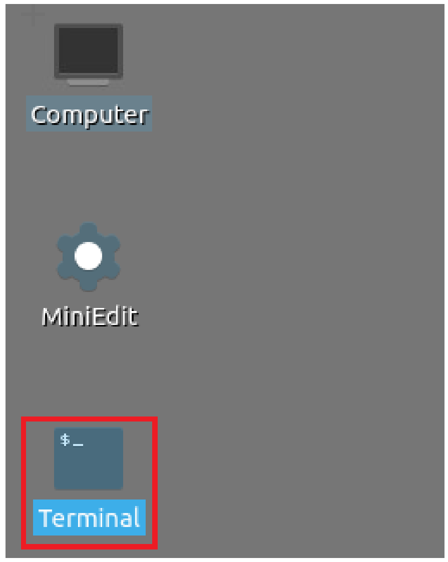


Figure 9. Shortcut to open a Linux terminal.

Step 2. Type the command below to open the working directory with Visual Studio Code (VS Code). With VS Code you will edit the *.p4* files, compile the source code, and load the binary to the switch.

```
code P4_Labs/lab4/
```

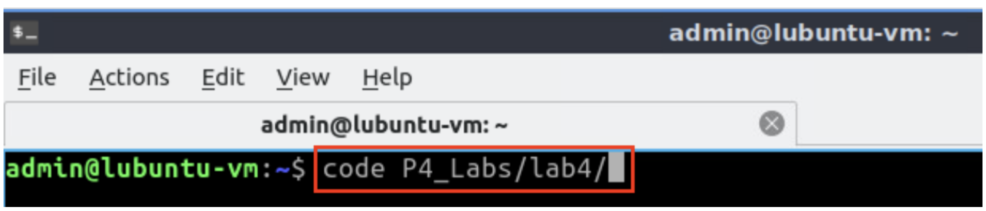


Figure 10. Loading the development environment.

3.2 Defining a custom header

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.

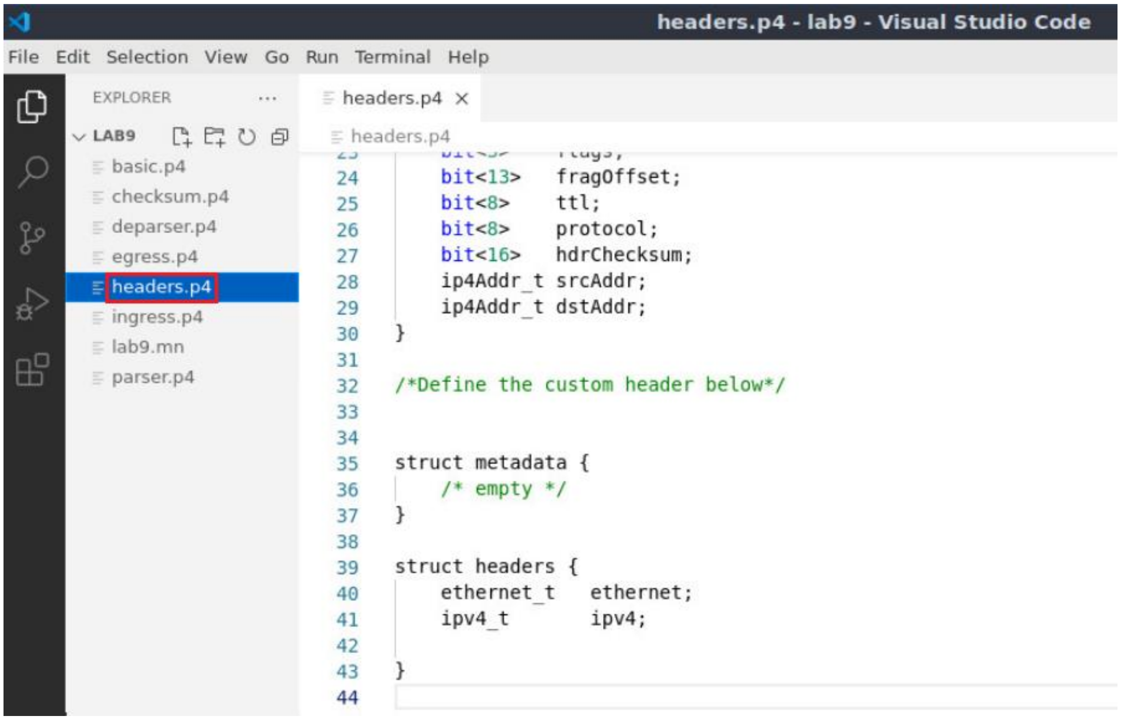


Figure 11. Inspecting the *headers.p4* file.

Step 2. Define a custom header type by adding the code shown below. Note the fields specified in the custom header will contain the ingress port, the egress port, and the packet length.

```
header my_custom_header_t {
       bit<16> ingress_port;
       bit<16> egress_port;
       bit<32> packet_length;
}
```

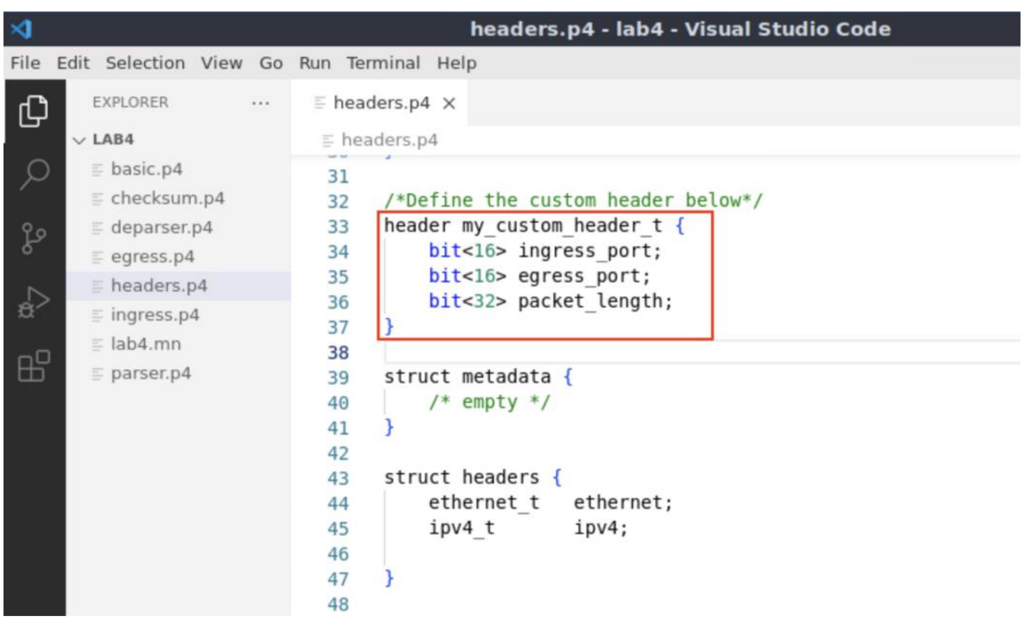


Figure 12. Defining a custom header type.

Step 3. Append the custom header to current packet headers consisting of the Ethernet and the IPv4 headers by adding the following line of code.

```
my_custom_header_t my_custom_header;
```

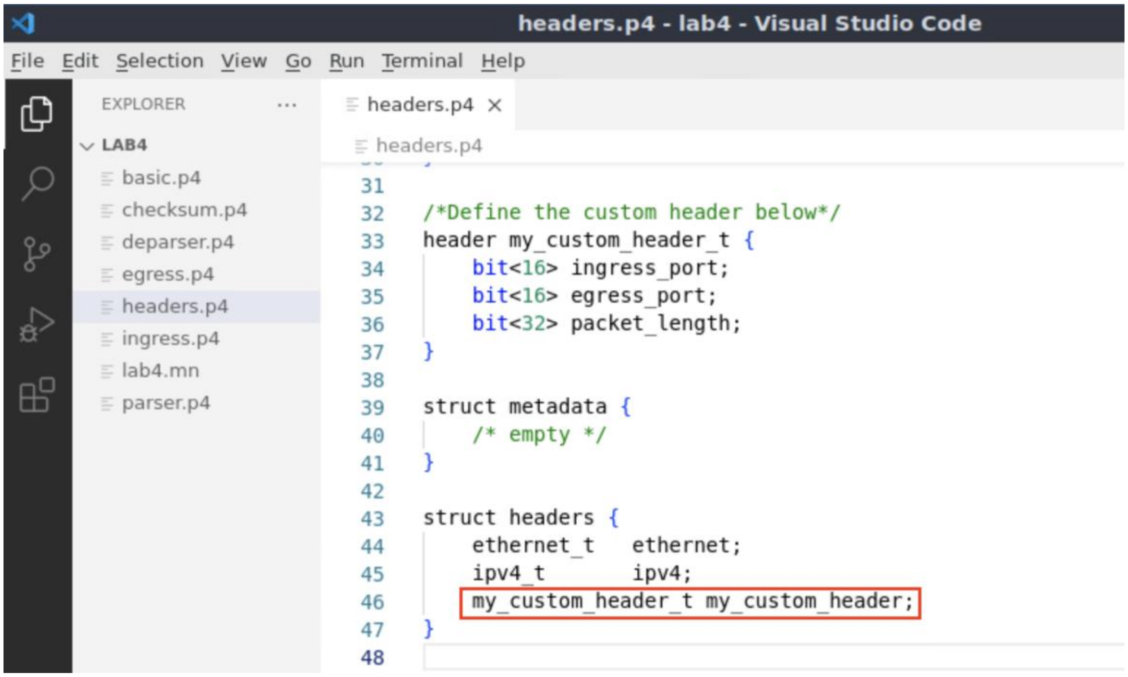


Figure 13. Defining a custom header.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

3.3 Parsing a custom header

Step 1. Click on the *parser.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.

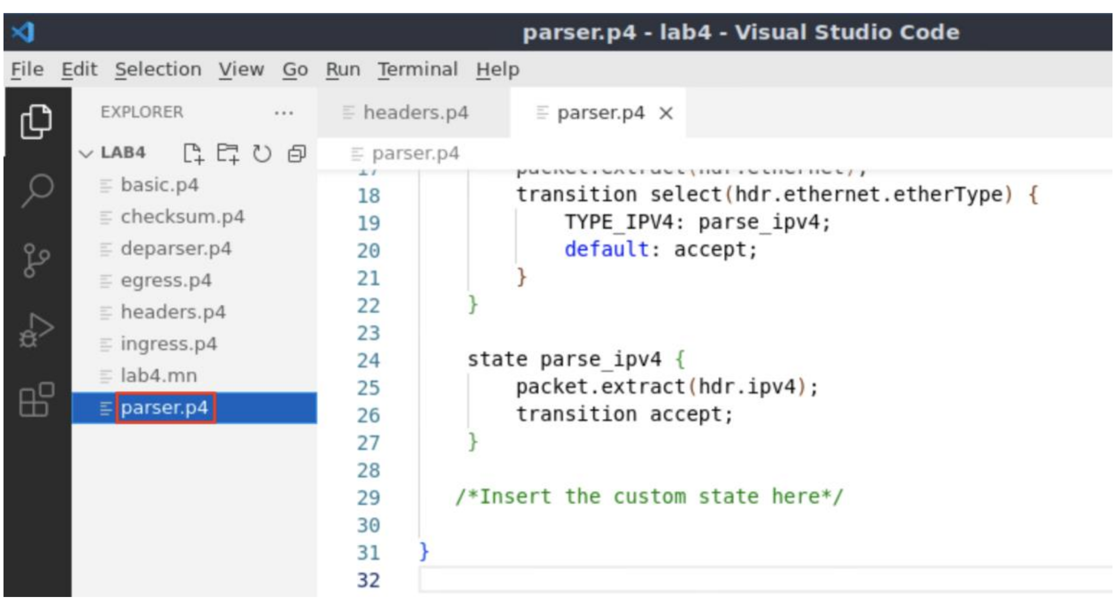


Figure 14. Inspecting the *parser.p4* file.

Step 2. Define a state to parse the custom header `my_custom_header` by adding the following piece of code.

```
state parse_my_custom_header{
       packet.extract(hdr.my_custom_header);
       transition accept;
}
```

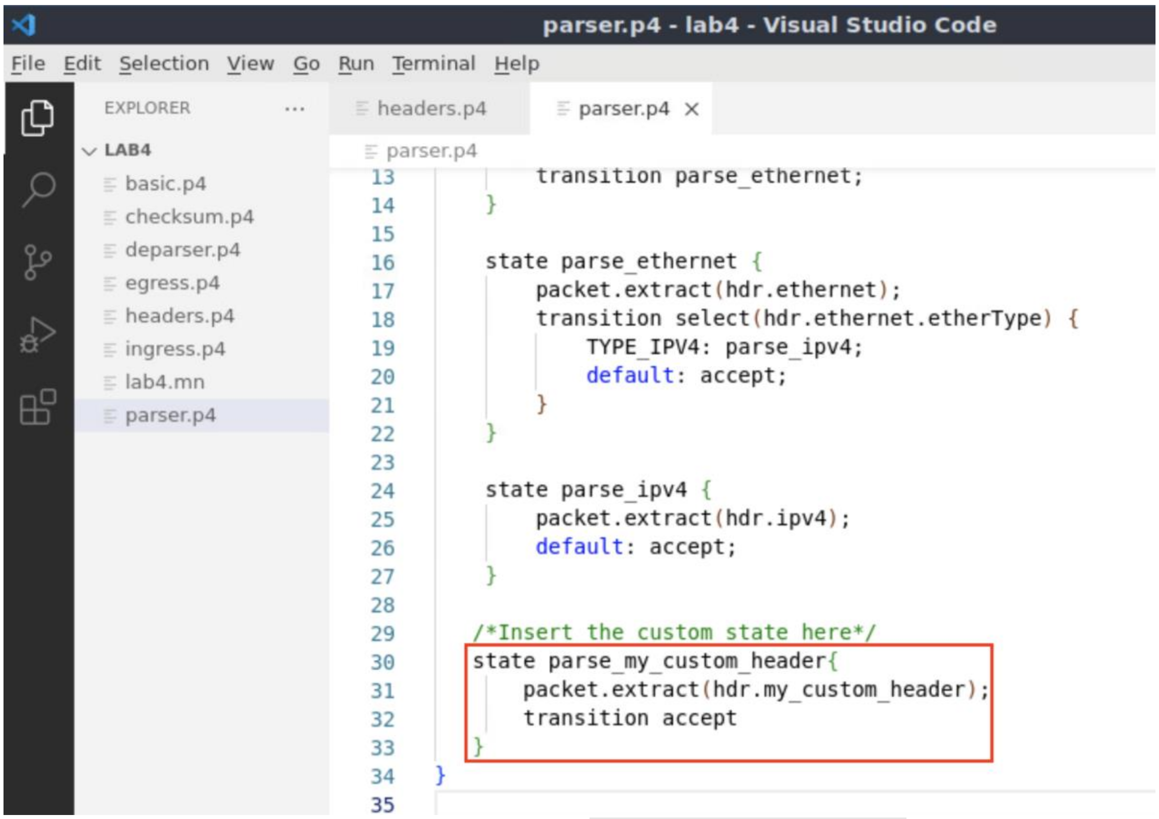


Figure 15. Defining the state `parse_my_custom_header`.

Step 3. Modify the transition statement in the `parse_ipv4` state by adding the following line of code. Instead of transitioning to the accept state after parsing the IP header, the parser will parse the custom header.

```
transition parse_my_custom_header;
```

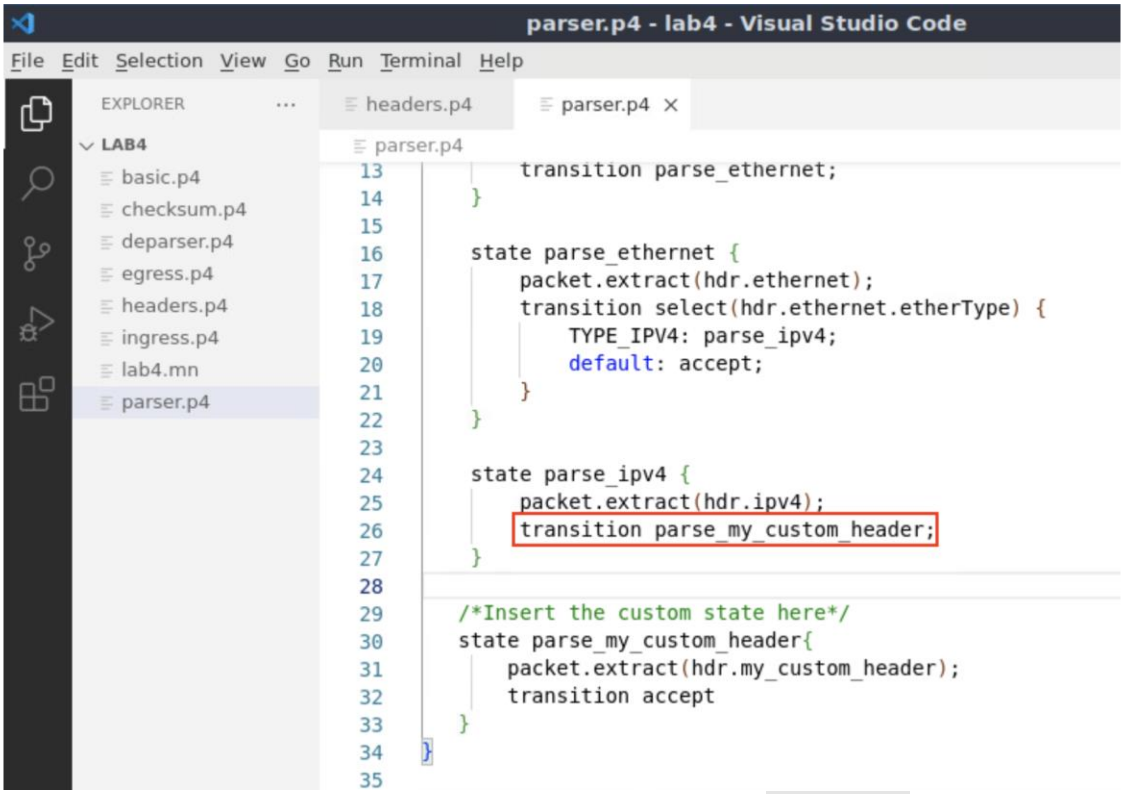


Figure 16. Modifying the transition statement in the `parse_ipv4` state.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

4 Processing a custom header

In this section, you will define the ingress pipeline’s behavior with a match-action table. This match-action table has the ingress port as the key and the actions to forward, drop and ignore packets. Then, you will process the custom header in the egress pipeline. The custom header will contain metadata such as the ingress port, the egress port, and the packet length. Finally, you will resemble and emit the Ethernet, the IPv4, and the custom header by programming the deparser.

4.1 Programming the ingress pipeline to forward a packet

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.

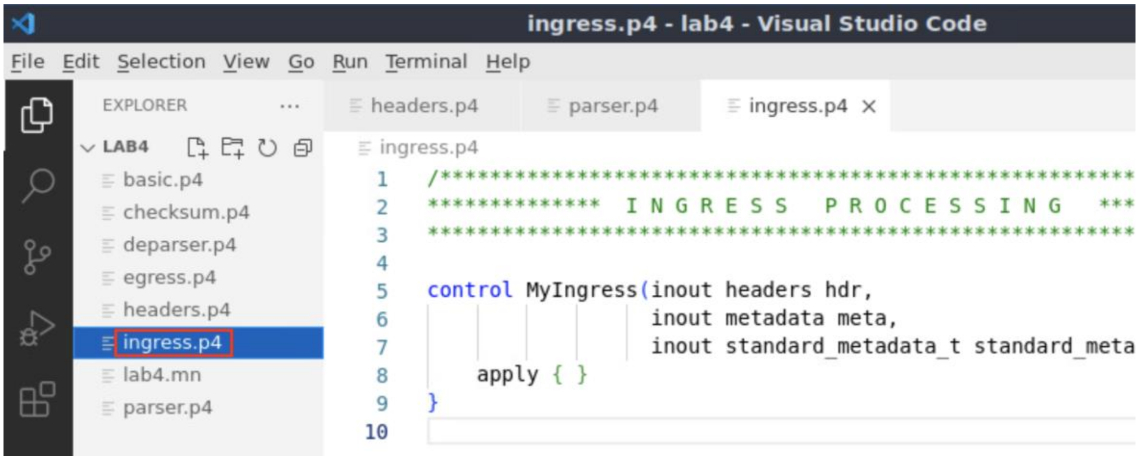


Figure 17. Opening the ingress processing block.

The *ingress.p4* file declares a control block named *MyIngress*. The *MyIngress* control block processes the headers (i.e., Ethernet, IPv4, and custom), the custom metadata (not used in this lab), and the standard metadata. The body of the control block is empty. You will define the actions that the match-action table will call as follows:

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

Step 2. Define the behavior of the `forward` action by inserting the code below inside the *MyIngress* control block.

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

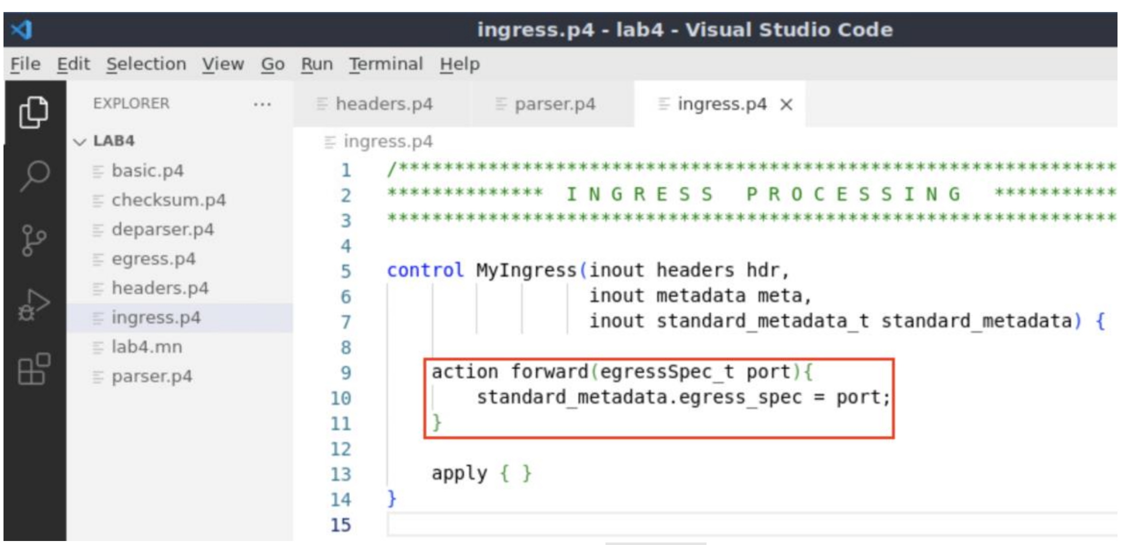


Figure 18. Defining the `forward` action.

The action `forward` takes as parameters the port number of data type `egressSpec_t` `port`. The switch uses the value of `port` to determine the egress port of a packet. The `egressSpec_t` is a user-define data type that corresponds to `bit<9>` specified in the *headers.p4* file.

The `standard_metadata` is an instance of the `standard_metadata_t` struct provided by the V1Model. Consider the figure below. In line 10, the `standard_metadata.egress_spec` determines the egress port. The value of `port` is populated from the control plane as action data.

Step 3. Now you will define the drop action by inserting the following code.

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

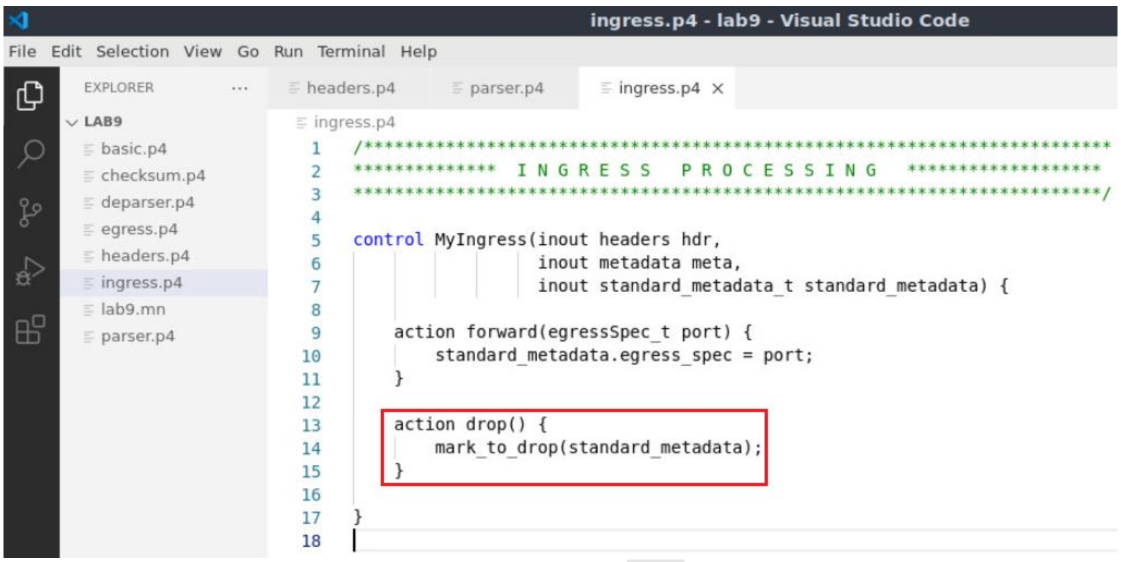


Figure 19. Defining the `drop` action.

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 4. Now you will define a table named `forwarding` by adding the following piece of code inside the control block *MyIngress*.

```
table forwarding {
        key = {
                standard_metadata.ingress_port: exact;
        }
        actions = {
                forward;
                drop;
                NoAction;
        }
        size = 1024;
        default_action = drop();

}
```

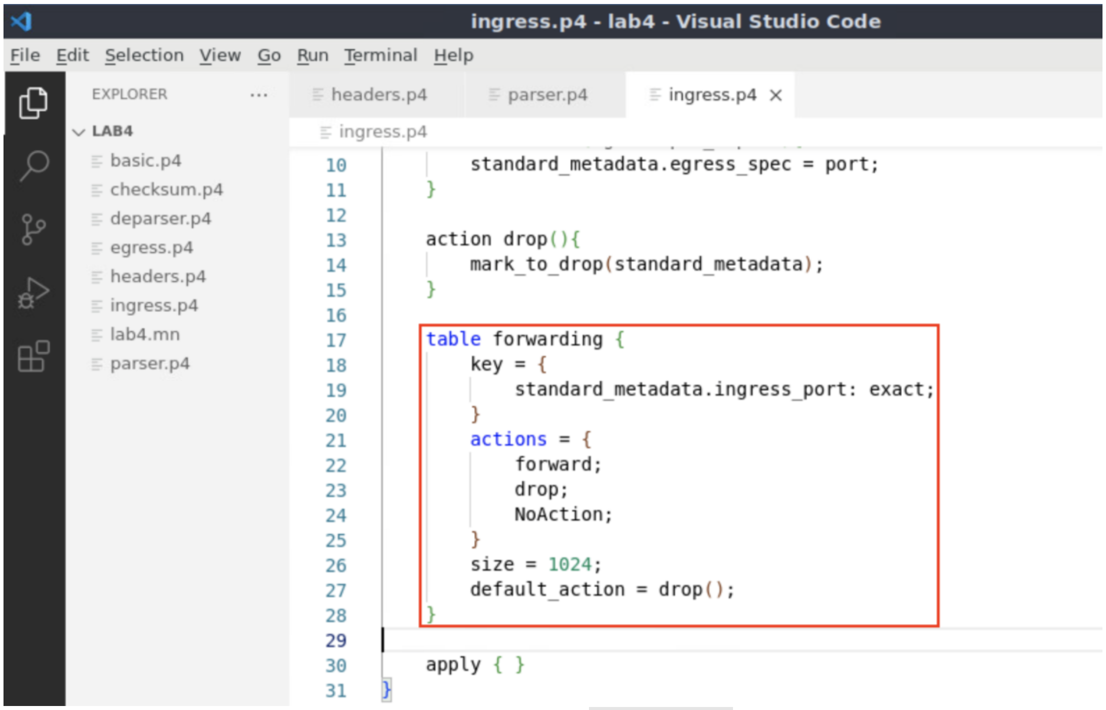


Figure 20. Declaring the `forwarding` table.

The `forwarding` table matches at the ingress port using an exact match. The actions include forward, drop, and NoAction. The table can contain up to 1024 entries, and the default action invokes the `drop` action.

Step 5. Add the following code inside the *MyIngress* block. The code below describes the ingress pipeline logic by sequentially invoking the tables, applying conditional statements (e.g., if-else statements), among other packet processing instructions.

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

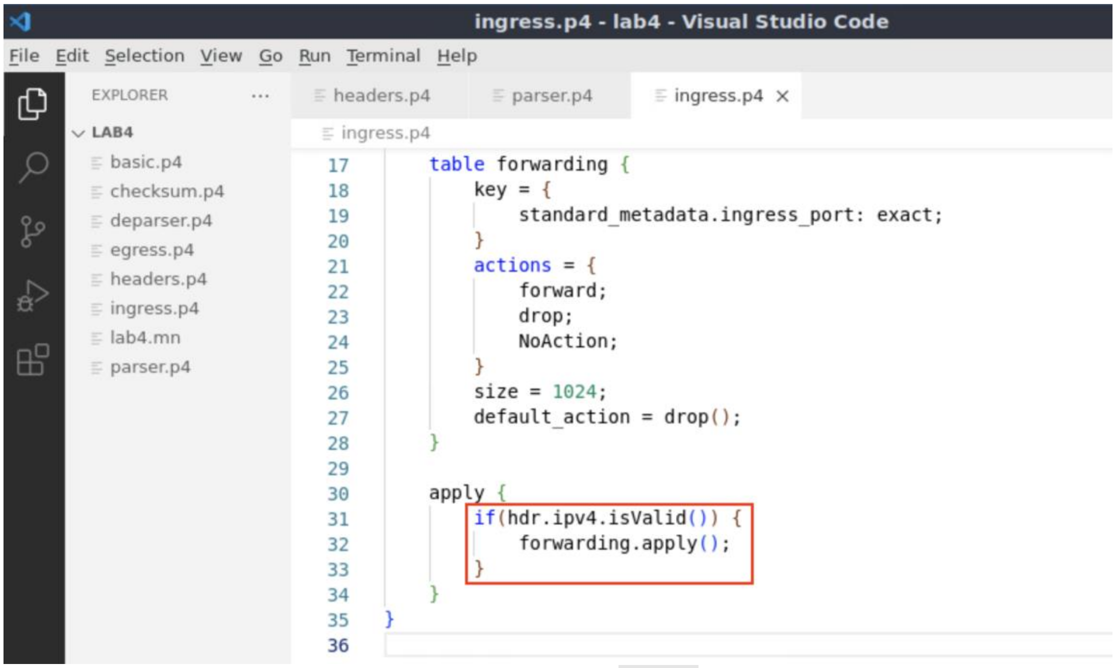


Figure 21. Defining the `apply` block.

The `apply` statement defines the sequential flow of packet processing. It is required in every control block, otherwise the program will not compile. The code above applies the table `forwarding` if the IPv4 header is valid (see line 31). Note that if the switch receives 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 6. Save the changes to the file by pressing `Ctrl + s`.

4.2 Programming the egress pipeline to modify a custom header

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

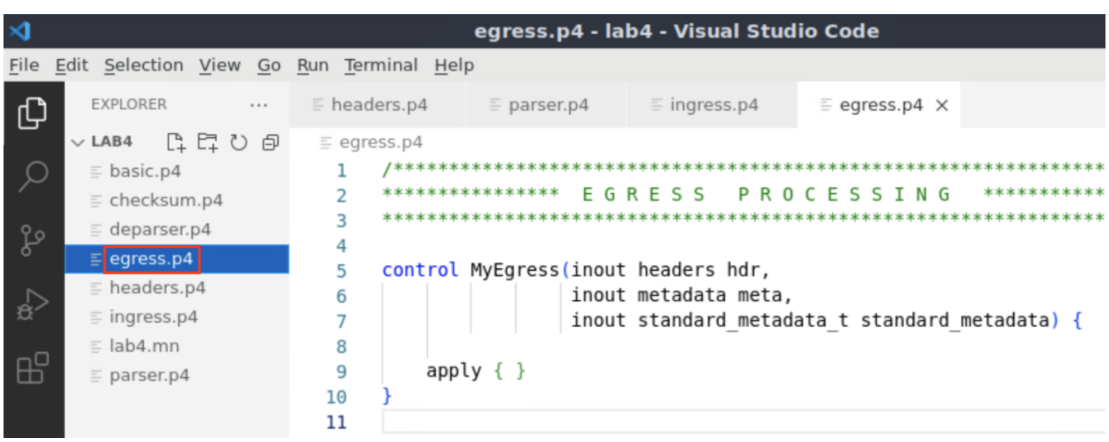


Figure 22. Opening the egress processing block.

Step 2. Define the modify action by adding the following piece of code.

```
action modify(){
  hdr.my_custom_header.ingress_port = (bit<16>)standard_meatadata.ingress_port;
  hdr.my_custom_header.egress_port = (bit<16>)standard_meatadata.egress_port;
  hdr.my_custom_header.packet_length = standard_metadata.packet_length;
}
```

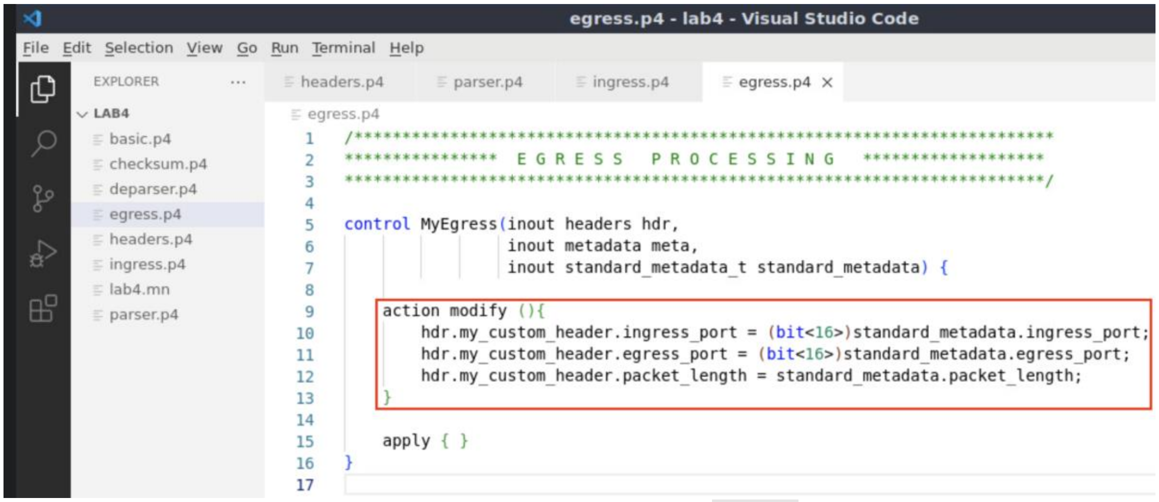


Figure 23. Defining the action `modify`.

The action defined above stores information from the standard metadata. Note that the length of `standard_metadata.ingress_port` and `standard_metadata.egress_port` is 9 bits. However, in $P4_{16}$ the header fields must be byte aligned. Thus, in lines 10 and 11, you cast the values to 16-bits numbers.

Step 3. Define the table `modify_custom_header` by adding the following piece of code.

```
table modify_custom_header {
       actions = {
              modify;
              NoAction;
       }
       size = 1;
       default_action = modify();
}
```

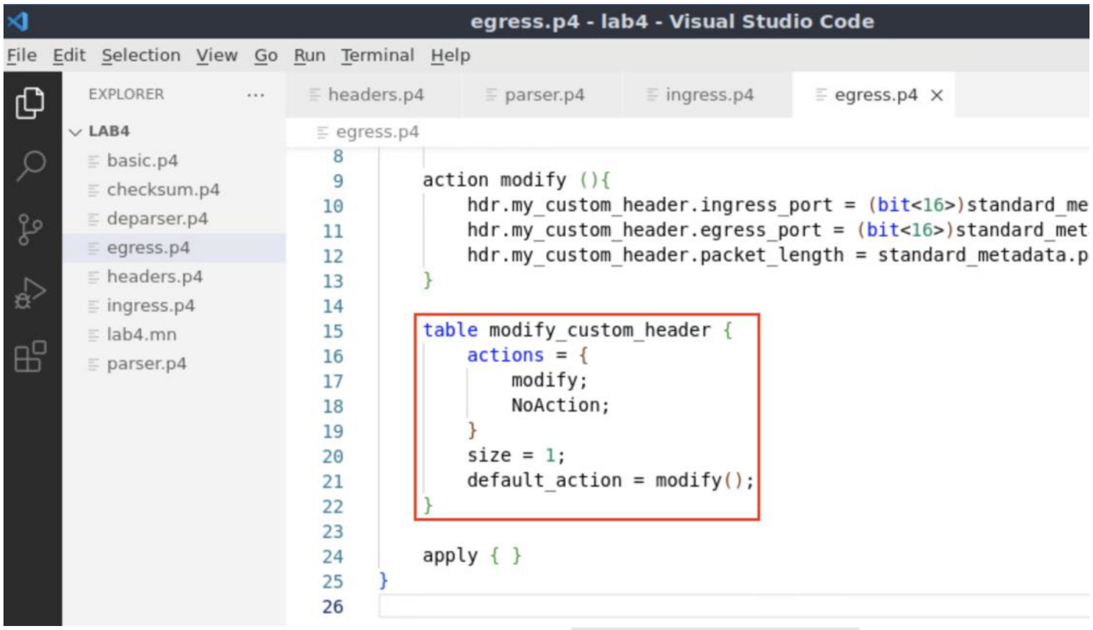


Figure 24. Defining the table `modify_custom_header`.

Note that the table `modify_custom_header` does not contain any key, which means it will not include any entries. Although the table has two actions, they are never invoked. Instead, it always executes the default action (i.e., `modify()`).

Step 4. Apply the egress logic by adding the following piece of code. Note that the table `modify_custom_header` is applied only if the custom header is valid.

```
apply {
    modify_custom_header.apply();
}
```

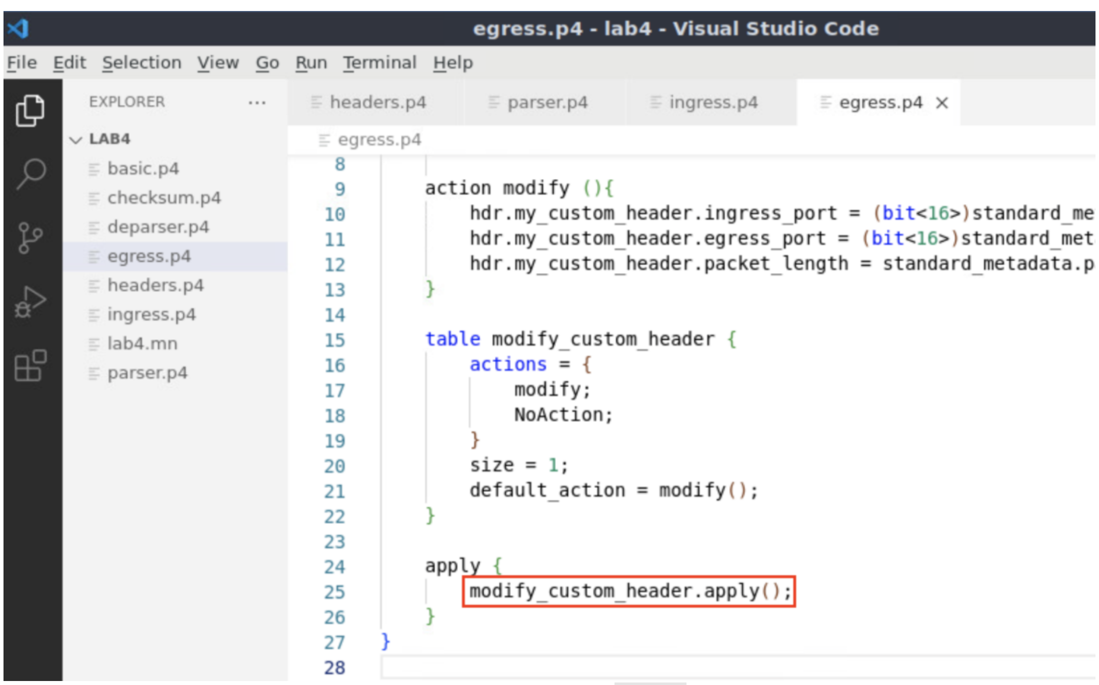


Figure 25. Defining the `apply` logic.

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

4.3 Programing the deparser

Step 1. Click on the *deparser.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. You will observe that the Ethernet and the IPv4 headers are already deparsed.

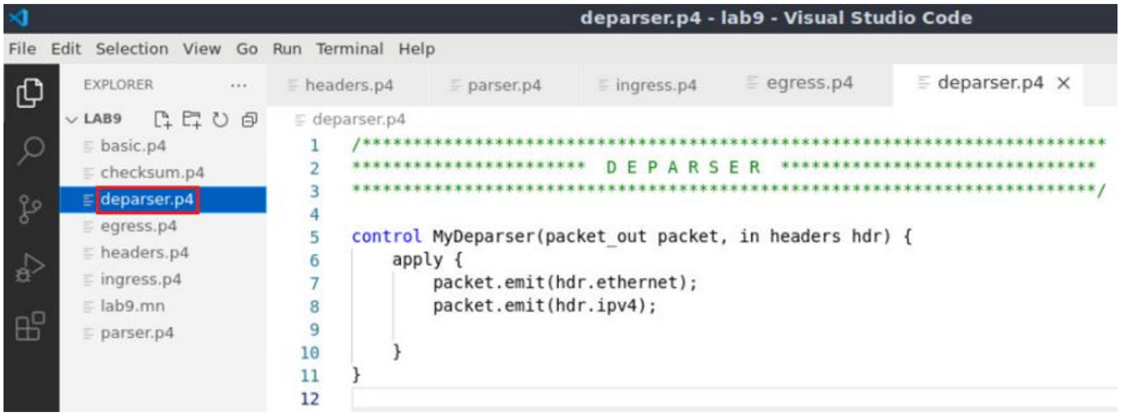


Figure 26. Opening the deparser processing block.

Step 2. Add the following line of code to emit the custom header.

```
packet.emit(hdr.my_custom_header);
```

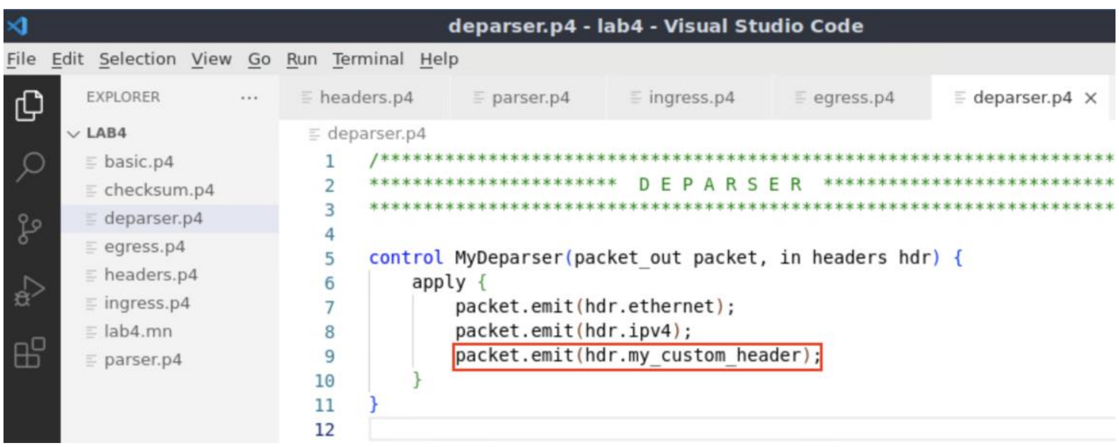


Figure 27. Emitting a custom header.

Step 3. Save the changes to the file by pressing `Ctrl + s`.

At this point, you created a P4 program that parses and processes a custom header.

5 Loading the P4 program

In this section, you will compile and load the P4 binary into switch s1. You will also verify that the binary resides in switch s1 filesystem.

5.1 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
```

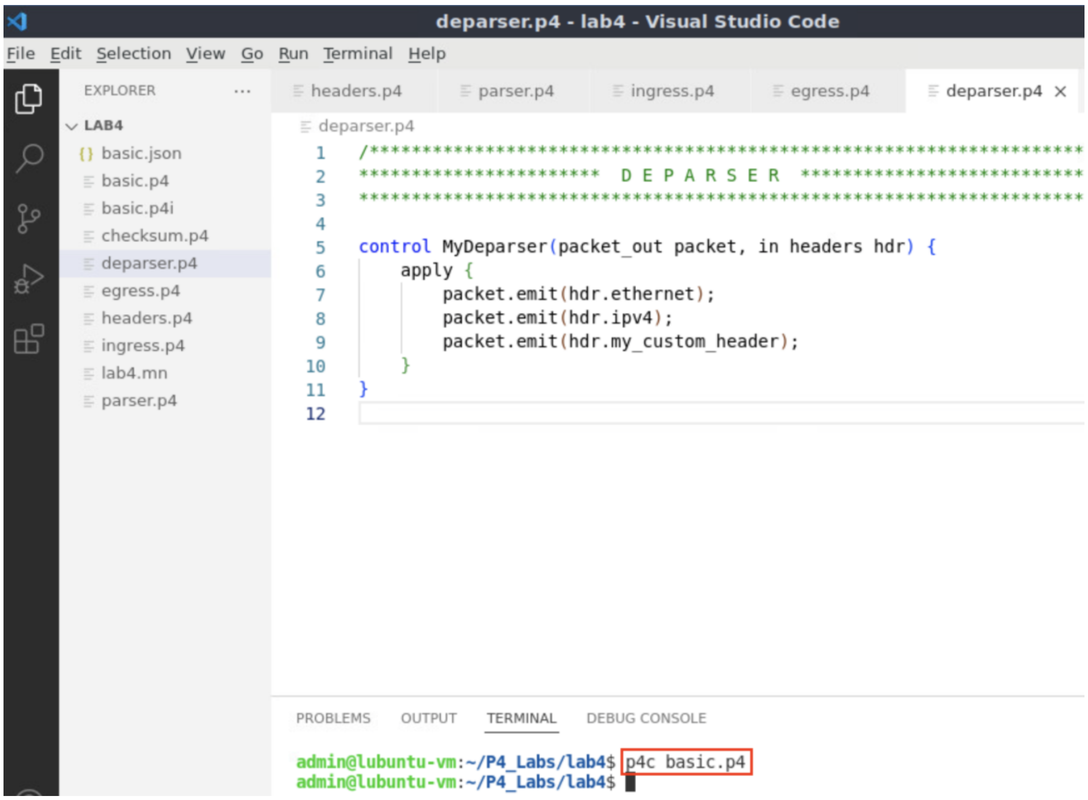


Figure 28. Compiling a P4 program.

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
```

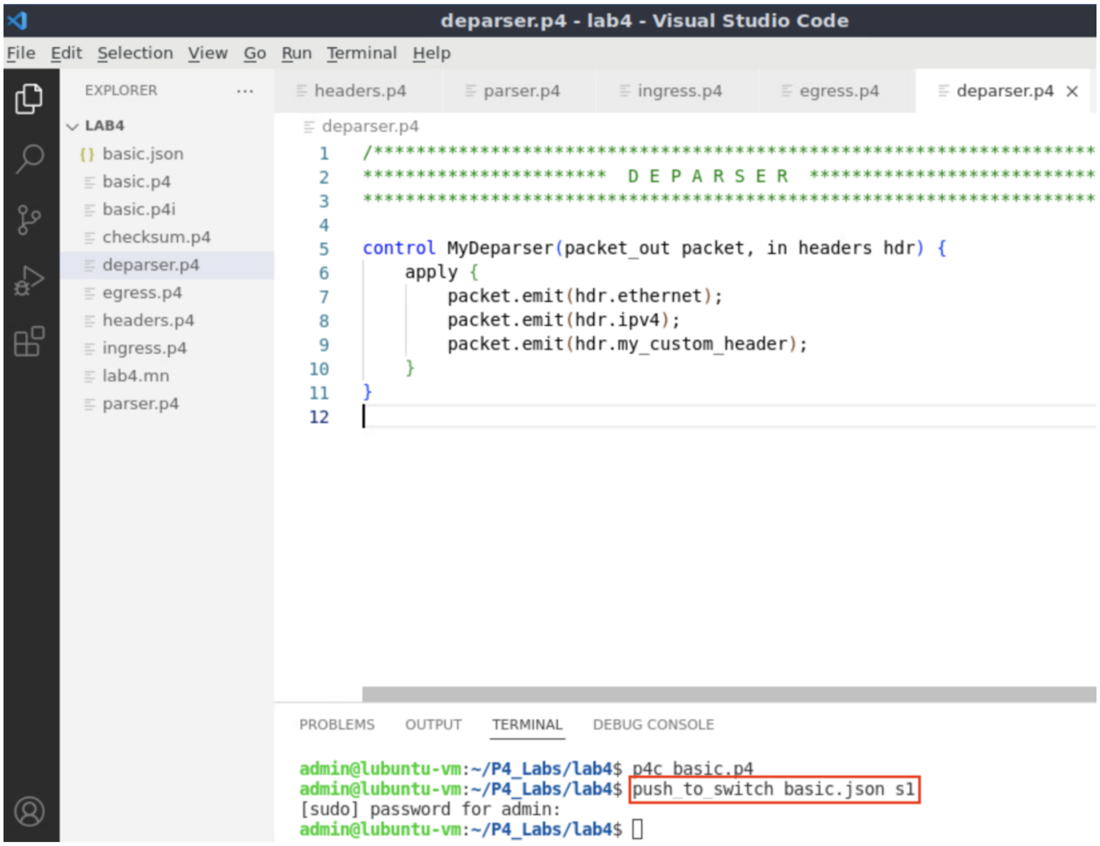


Figure 29. Pushing the *basic.json* file to switch s1.

5.2 Verifying the configuration

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

Figure 30. Maximizing the MiniEdit window.

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

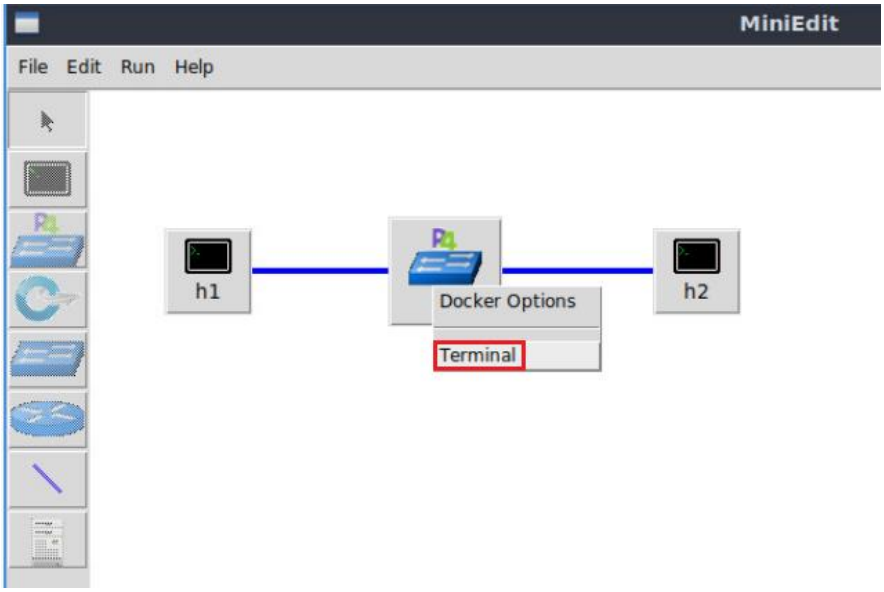


Figure 31. Starting the terminal on the switch.

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

Step 3. Issue the command `ls` on switch s1 terminal to verify that the filesystem contains the P4 program binary (i.e., *basic.json*)

```
ls
```

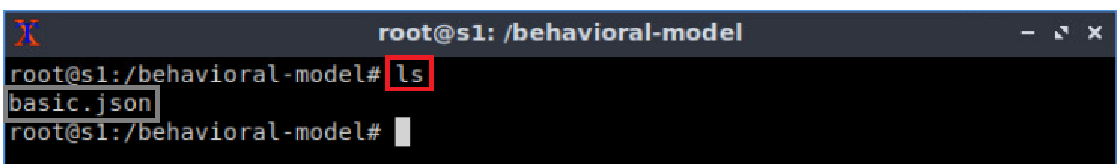


Figure 32. Displaying the contents of the current directory in the switch s1.

6 Configuring switch s1

In this section, you will observe and understand the purpose of the interfaces available in switch s1. You will map those interfaces to the ports in the P4 program and start the switch's daemon. Note that the switch's logs are enabled to see the tables and actions that packets hit across the pipeline. Finally, you will load the rules to populate the match action tables.

6.1 Mapping P4 program's ports

Step 1. Issue the command below on the terminal of the switch s1 to see the available interfaces in switch s1.

```
ifconfig
```

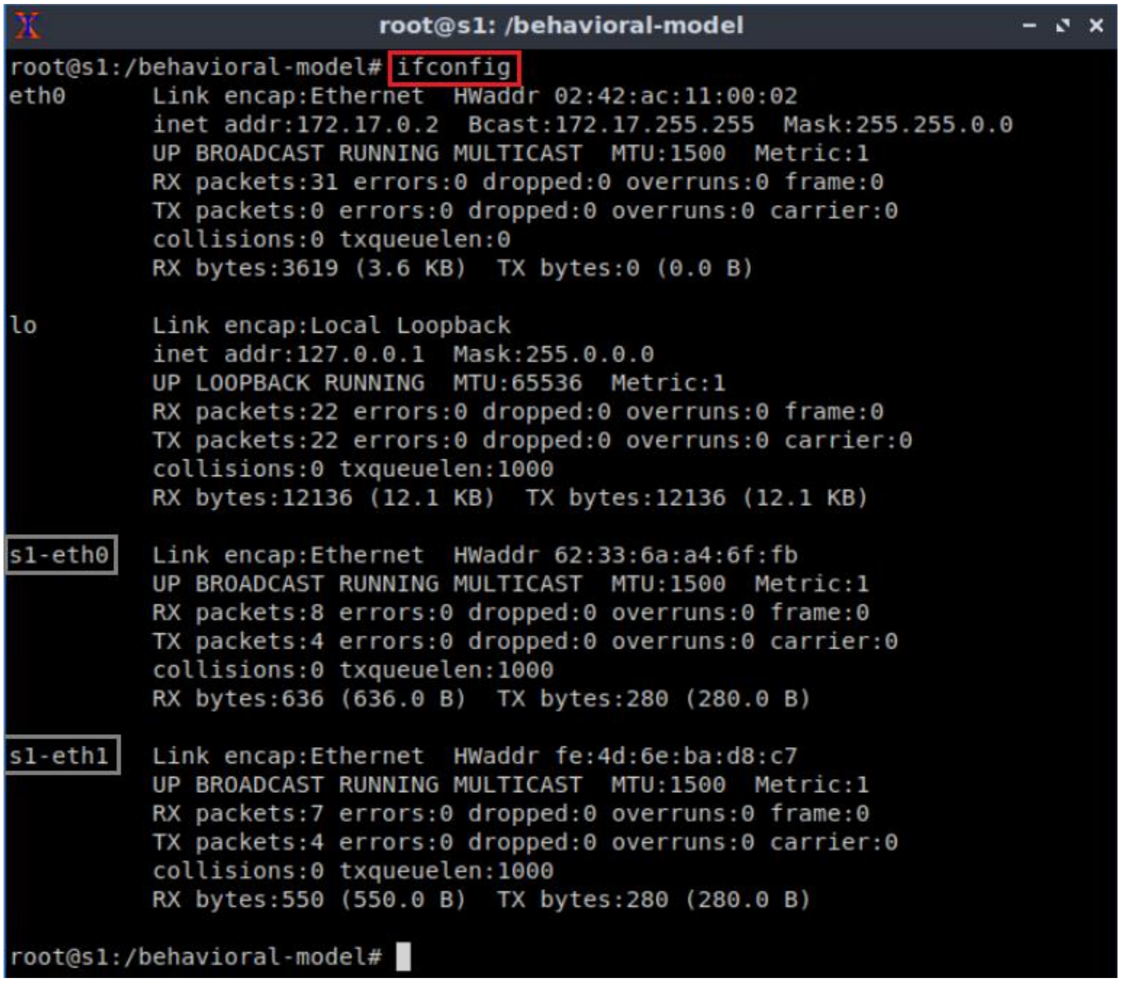


Figure 33. Displaying switch s1 interfaces.

You can observe that the switch has four interfaces: *eth0*, *lo*, *s1-eth0*, and *s1-eth1*. The interface *eth0* is used to communicate with the container, and *lo* is the loopback interface. None of these interfaces are used by the P4 program. On the other hand, interfaces *s1-eth0* and *s1-eth1* are used by the P4 program because they connect to hosts h1 and h2. Interface *s1-eth0* connects host h1, and interface *s1-eth1* connects to 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 &
```

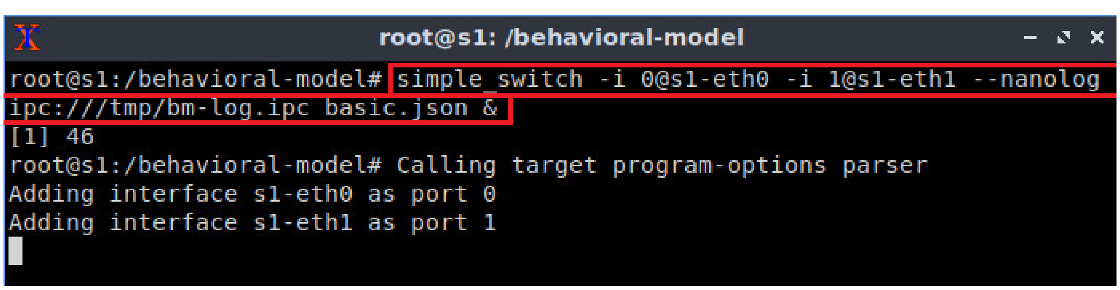


Figure 34. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

The `--nanolog` option is used to instruct the switch daemon to display the switch's logs.

6.2 Loading the rules to the switch

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

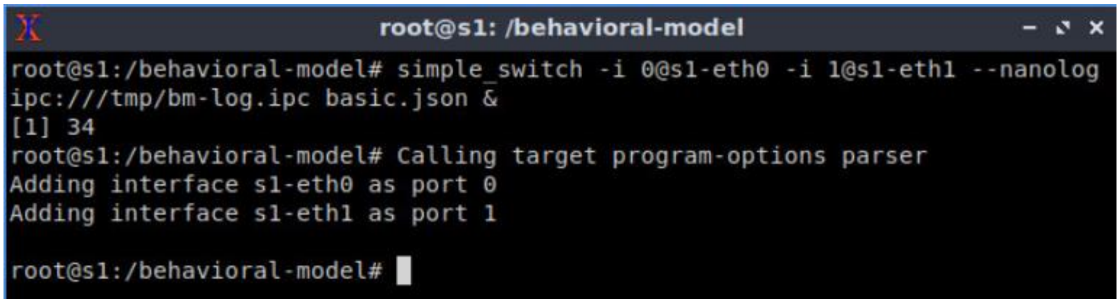


Figure 35. Returning to switch s1 CLI.

Step 2. Populate the table entries by typing the following command.

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

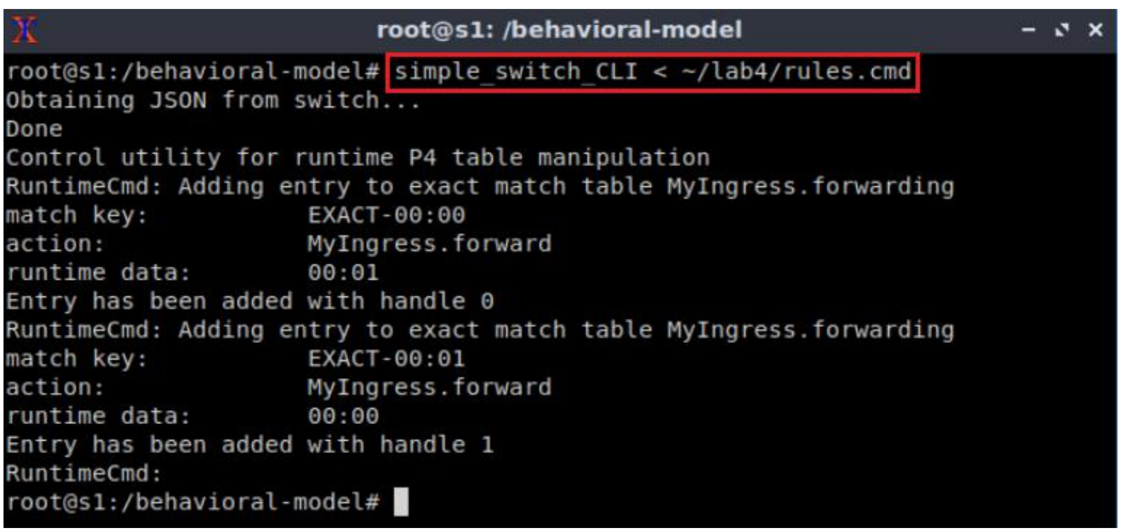


Figure 36. Populating the forwarding table into switch s1.

The script above populates the entries in the `forwarding` table defined in the P4 program. The first entry matches the key value of `00:00`, executes the action `forward`, and loads the action data with `00:01`. The handle of this entry is `0`. Similarly, the second entry matches the key value of `00:01`, executes the action `forward`, and loads the action data with `00:00`. The handle of this entry is `1`.

7 Testing and verifying the P4 program

In this section, you will test the P4 program by sending custom packets from host h1 to host h2. You will run the *nanomsg* client application to log the pipeline stages and observe how the packet looks like when it reaches its destination (i.e., host h2).

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

```
nanomsg_client.py
```

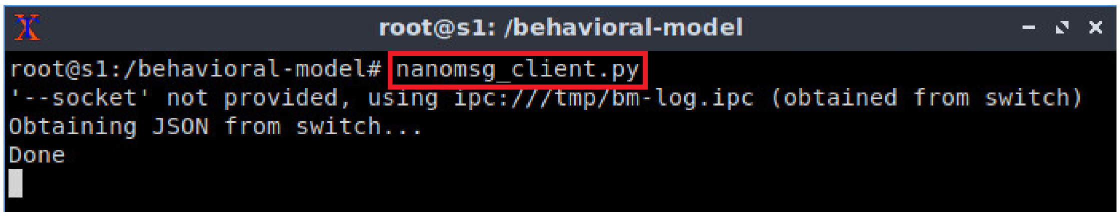


Figure 37. Displaying switch s1 logs.

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

```
recv.py -p custom
```

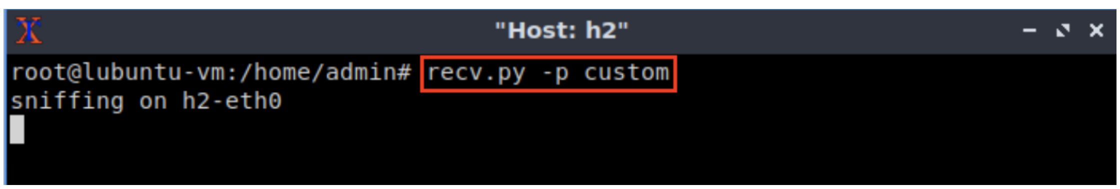


Figure 38. Listening for incoming packets in host h2.

The script above receives the following parameters:

- `-p`: enables listening to a specific protocol.
- `custom`: the type of protocol.

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

```
send.py 10.0.0.2 HelloWorld -p custom
```

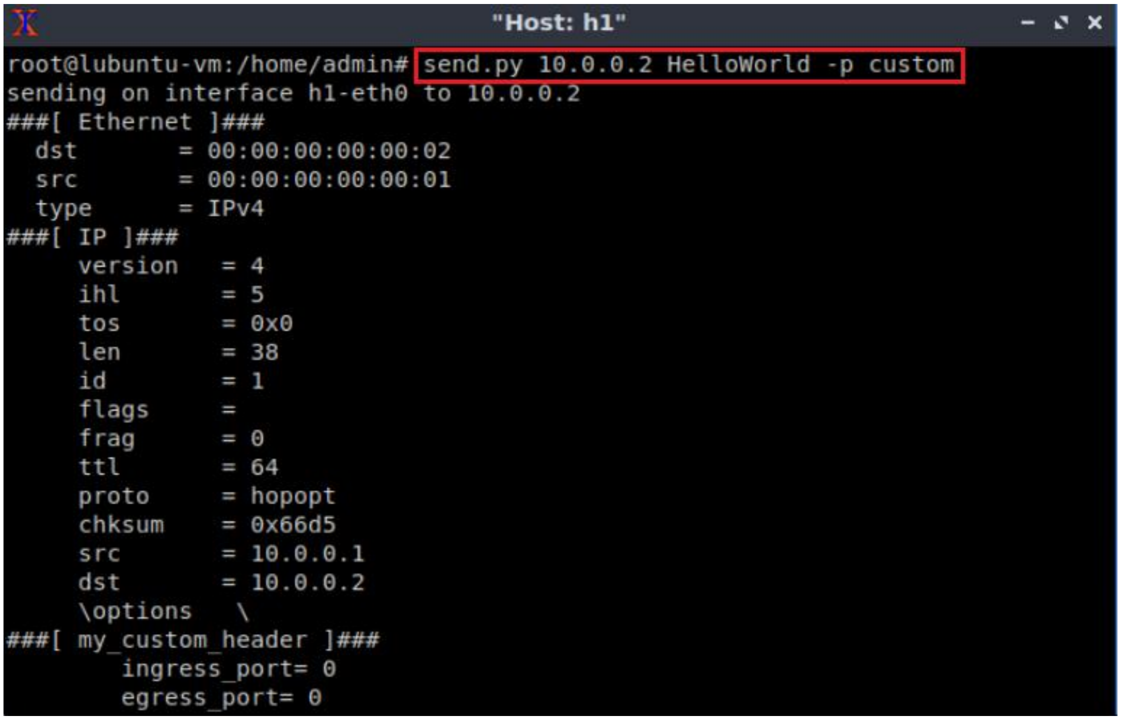
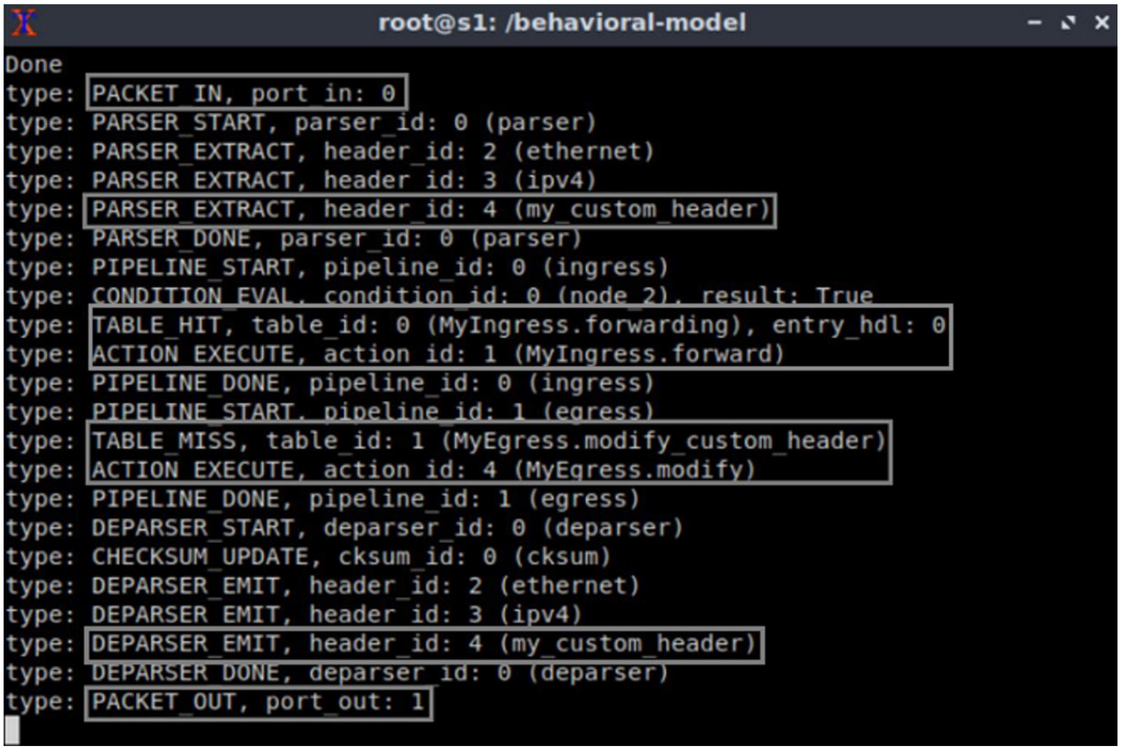


Figure 39. Sending a test packet from host h1 to host h2.

Similarly, the script above receives the following parameters:

- `10.0.0.2`: the destination IPv4 address.
- `HelloWorld`: the packet payload.
- `-p`: enables listening to a specific protocol.
- `custom`: the type of protocol.

Step 4. Inspect the logs on switch s1 terminal.



```
root@s1: /behavioral-model
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_EXTRACT, header_id: 4 (my_custom_header)
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: 1 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: TABLE_MISS, table_id: 1 (MyEgress.modify_custom_header)
type: ACTION_EXECUTE, action_id: 4 (MyEgress.modify)
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_EMIT, header_id: 4 (my_custom_header)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 40. Inspecting the logs in switch s1.

The switch's log shows that a packet is received in port `0`. Then, the parser extracts the Ethernet, the IPv4, and the custom header defined as `my_custom_header`. In the egress pipeline, note that a hit in the `forwarding` table which invokes the action `forward`. Then, there is a miss in the table `modify_custom_header` in the egress pipeline, which invokes the default action `modify`. Finally, the packet is deparsed and emitted through port `1`.

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

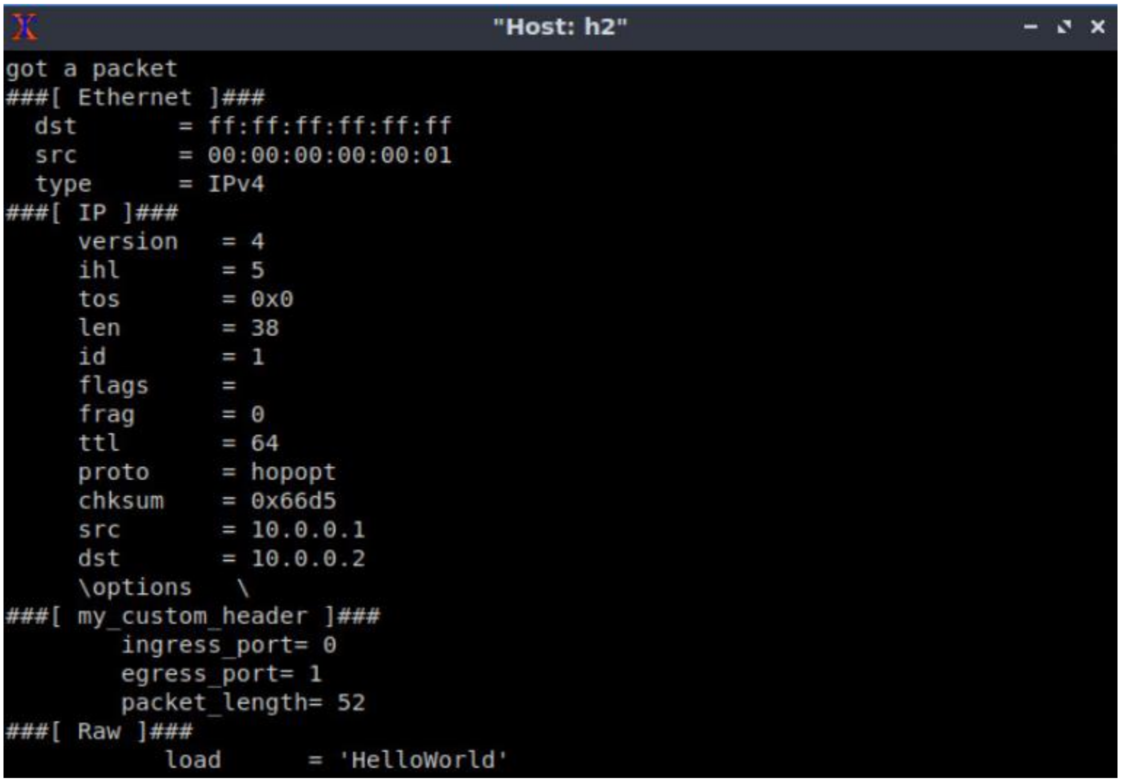


Figure 41. Packet received on host h2.

The figure above shows that the custom packet was received on host h2. The custom packet comprises the Ethernet, the IPv4, and the custom headers. The custom header contains the ingress port, which value is 0, the egress port, which value is 1, and the packet length, which is 52 bytes. The length of each header is summarized in the following table.

Table 2. Header lengths.

Header/Payload	Length (bytes)
Ethernet	14
IPv4	20
my_custom_header	8
Payload (HelloWorld)	10
Total	**52**

This concludes lab 4. 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.

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 5: Monitoring the Switch's Queue using Standard Metadata

Document Version: 08-08-2022

Contents

Overview

This lab is an introduction to queue monitoring using P4 standard metadata. The user will create a P4 program to obtain the queue length, the enqueueing timestamp, and the dequeuing timestamp. Then, the user will insert these values into a custom header to observe the evolution of the queueing delay and queue length from an end host.

Objectives

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

1. Understand how to obtain queueing delay from the switch's metadata.
2. Insert queueing metadata into a custom header.
3. Visualize the values of the queue length and queueing delay.

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 and parsing a custom header.
4. Section 4: Processing a custom header.
5. Section 5: Loading the P4 program.
6. Section 6: Configuring switch s1.
7. Section 7: Testing and verifying the P4 program.

1 Introduction to queueing delay

As a packet travels from the sender to the receiver, it experiences several types of delays at each node (router/switch) along the path. The most significant delays are processing, queuing, transmission, and propagation delay (see Figure 1).

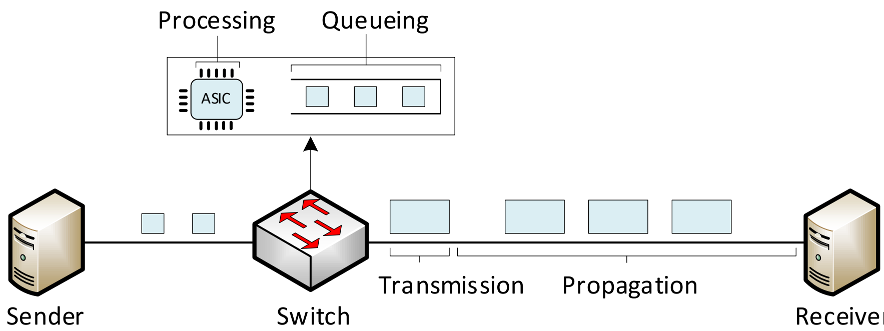


Figure 1. Delay components: processing, queueing, transmission, and propagation delays.

- Processing delay: The time required to examine the packet's header and determine where to direct the packet. For high-speed switches, this delay is on the order of microseconds or less.

- Transmission delay: The time required to put the bits on the wire. It is given by the packet size (in bits) divided by the bandwidth of the link (in bps). For example, for a 10 Gbps and 1,500-byte packet (12,000 bits), the transmission time is T = 12,000 / 10×10^9 = 0.0012 milliseconds or 1.2 microseconds.

- Queueing delay: The time a packet waits for transmission onto the link. The length of the queuing delay of a packet depends on the number of earlier-arriving packets that are queued and waiting for transmission onto the link. Queuing delays can be on the order of microseconds to milliseconds.

- Propagation delay: Once a bit is placed into the link, it needs to propagate to the other end of the link. The time required to propagate across the link is the propagation delay. In local area networks (LANs) and datacenter environments, this delay is small (microseconds to few milliseconds); however, in Wide Area Networks (WANs) / long-distance connections, the propagation delay can be on the order of hundreds of milliseconds.

1.1 Computing the queueing delay using standard metadata

Consider Figure 2. Switch s1 is a P4 programmable device with a bottleneck link bandwidth of 100 Mbps. Suppose a scenario where host h3 starts a data transfer to host h4. If the link between host h3 and switch s1 operates at a higher rate than the bottleneck link, a queue is formed at the egress interface of switch s1. Therefore, host h1 will experience an increased delay when communicating with host h2.

Switch s1's standard metadata contains the enqueueing and dequeuing timestamps. The enqueueing timestamp (`standard_metadata.enq_timestamp`) indicates when a packet enters the traffic manager (TM), and the dequeing timestamp (`standard_metadata.egress_global_timestamp`) denotes the time when the packet enters the egress pipeline. Note that these values are given with respect to the global switch's timer. With this information, the programmer can calculate the difference between the timestamps and obtain the queueing delay. Additionally, the programmer can obtain the queue length (`standard_metadata.enq_qdepth`) that indicates how many packets are occupying the switch's queue.

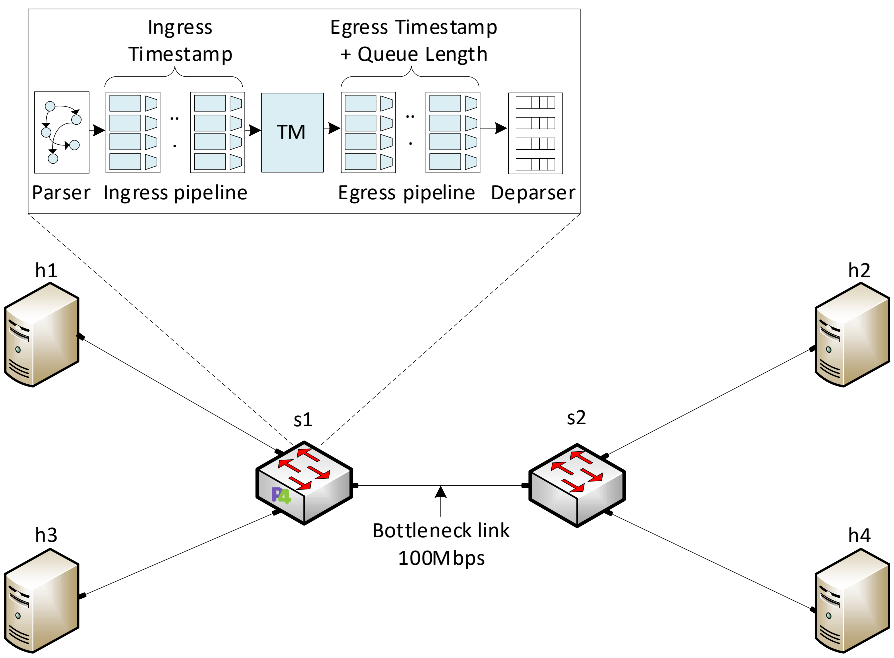


Figure 2. Measuring the queueing delay with switch s1.

2 Lab topology

Let us get started by loading a simple Mininet topology using MiniEdit. The topology comprises four end hosts, one P4 programmable switch, and one legacy switch.

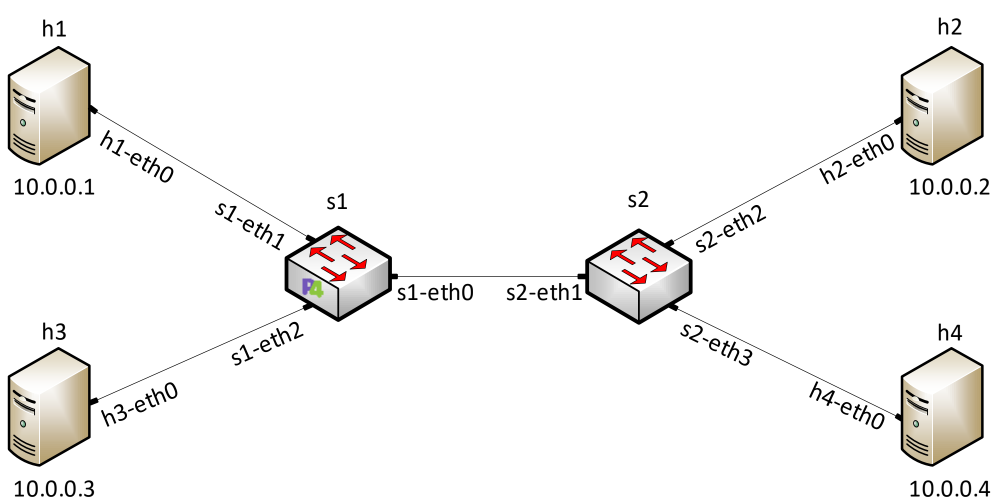


Figure 3. Lab topology.

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 *lab5* folder and search for the topology file called *lab5.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

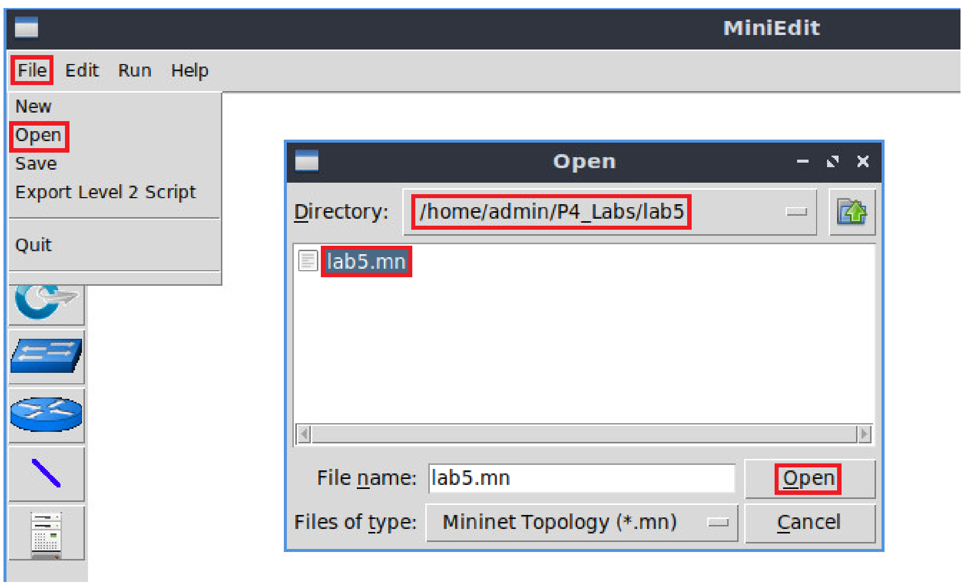


Figure 5. MiniEdit's *Open* dialog.

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.

3 Defining and parsing a custom header

3.1 Loading the programming environment

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

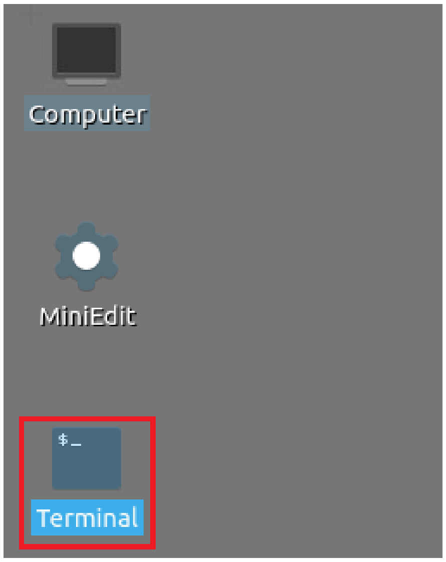


Figure 7. Shortcut to open a Linux terminal.

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 execute.

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
```

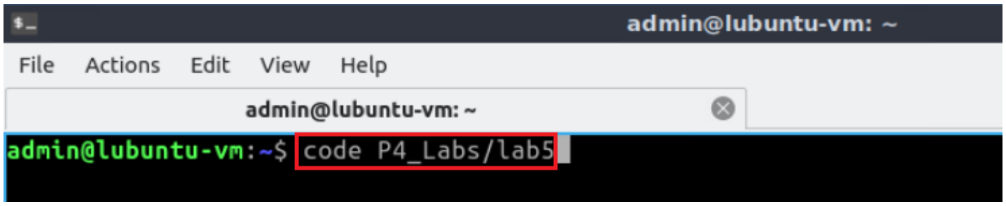


Figure 8. Loading the development environment.

3.2 Defining a custom header

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.

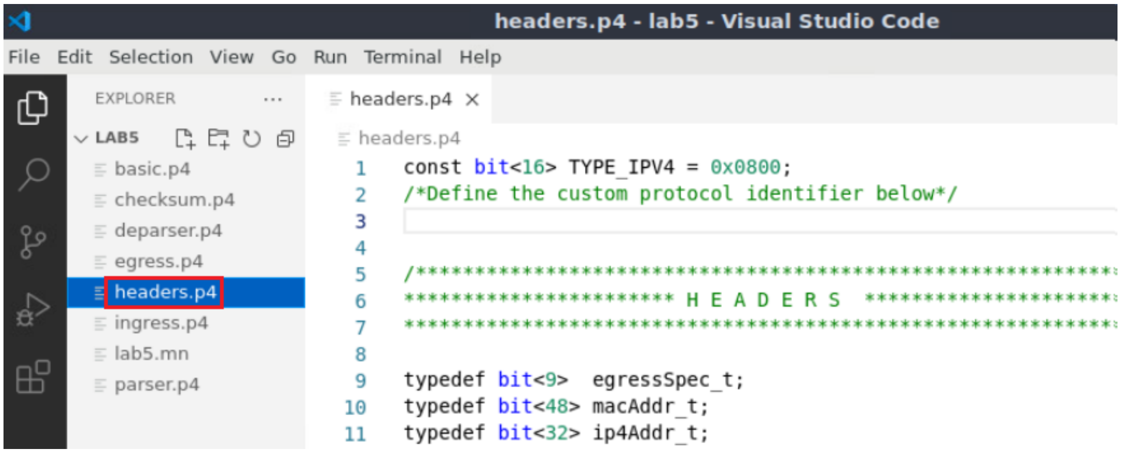


Figure 9. Inspecting the *headers.p4* file.

Step 2. Define the custom header identifier by issuing the following command. This constant valued indicates that the next header over IPv4 will be the one we defined.

```
const bit<8> TYPE_CUSTOM = 0xFD;
```

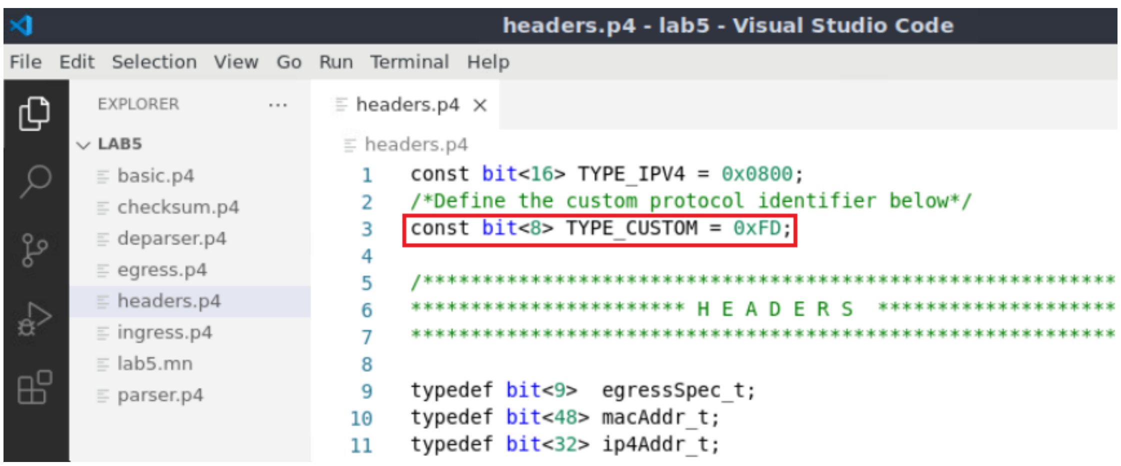


Figure 10. Defining the custom header identifier.

Step 3. Define the following custom header by adding code shown below.

```
header switch_stats_t {
       bit<8>  switch_ID;
       bit<32> enq_timestamp;
       bit<48> deq_timestamp;
       bit<48> q_delay;
       bit<24> q_depth;
}
```

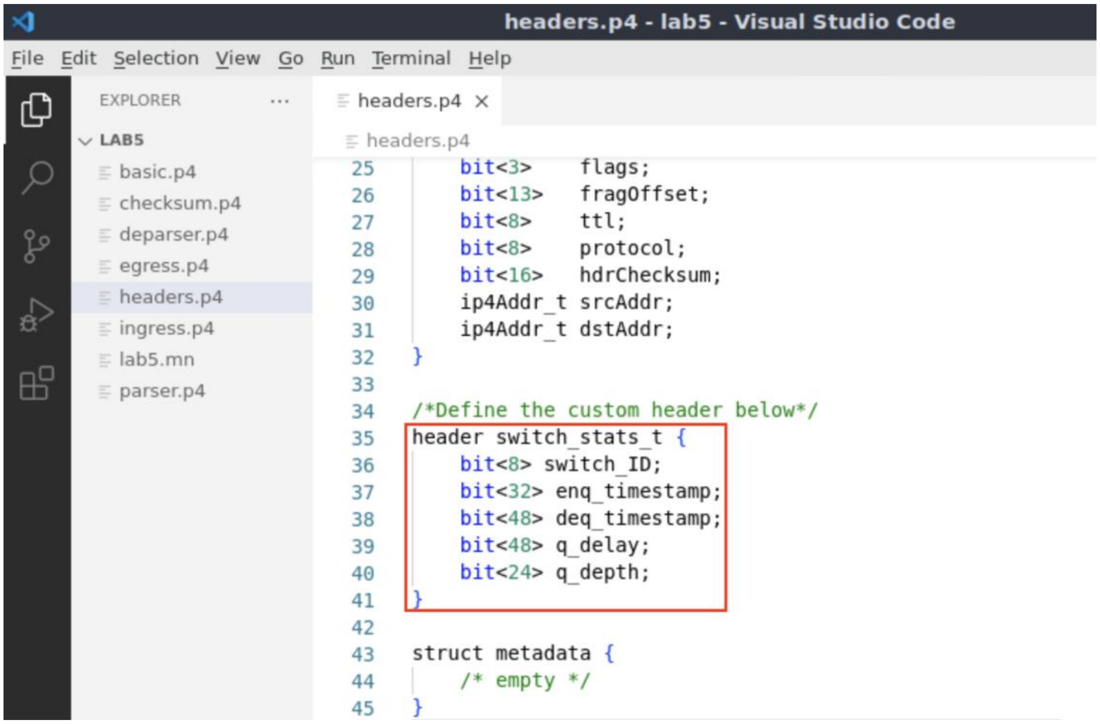


Figure 11. Defining a custom header type.

Step 4. Append the custom header to current Ethernet and IPv4 headers by inserting the following line of code.

```
switch_stats_t switch_stats;
```

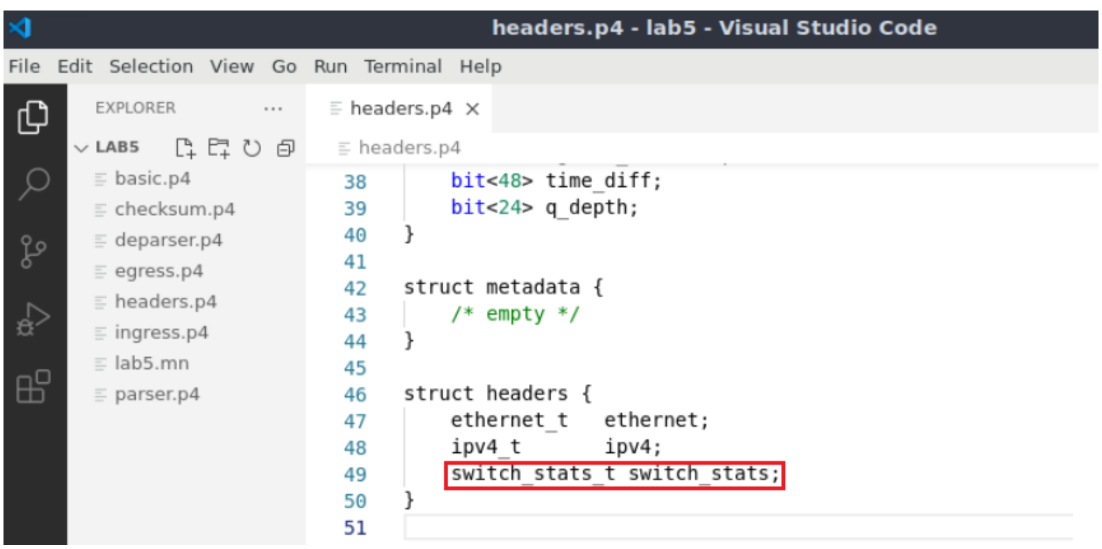


Figure 12. Defining a custom header.

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

3.3 Parsing a custom header

Step 1. Click on the *parser.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.

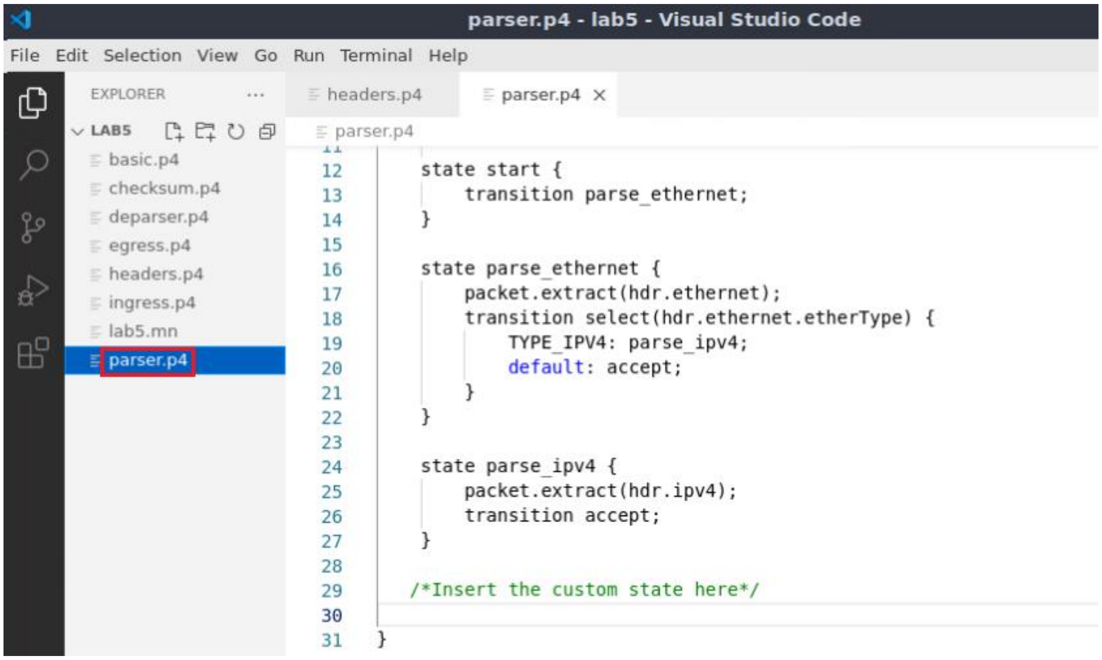


Figure 13. Inspecting the *parser.p4* file.

Step 2. Define a state to parse the custom header `switch_stats` by adding the following piece of code.

```
state parse_switch_stats{
       packet.extract(hdr.switch_stats);
       transition accept;
}
```

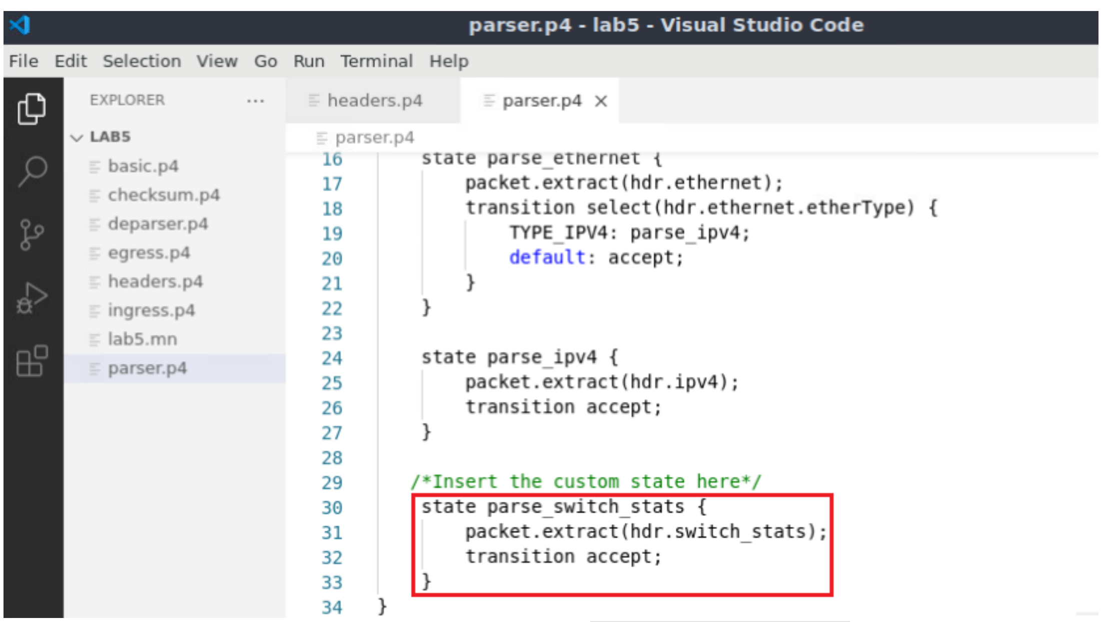


Figure 14. Defining the state `parse_switch_stats`.

Step 3. Modify the transition statement in the `parse_ipv4` state by adding the following line of code.

```
transition select(hdr.ipv4.protocol){
       TYPE_CUSTOM: parse_switch_stats;
       default accept;
}
```

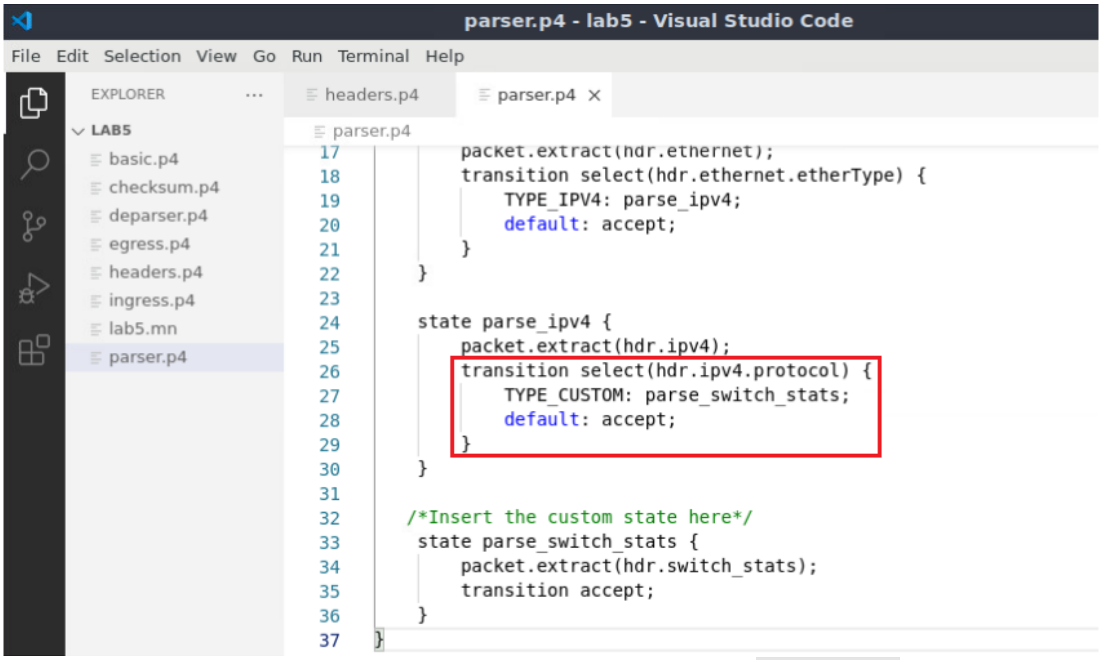


Figure 15. Modifying the transition statement in the `parse_ipv4` state.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

4 Processing a custom header

In this section, the user will program the egress pipeline to collect statistics such as the ingress timestamp, egress timestamp, the difference between the ingress and egress timestamps, and the queue length. All these values are obtained from the switch's metadata and computed using a match-action table. Finally, the user will emit the custom header by programming the deparser.

4.1 Programming the egress pipeline

Step 1. Click on the *egress.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.

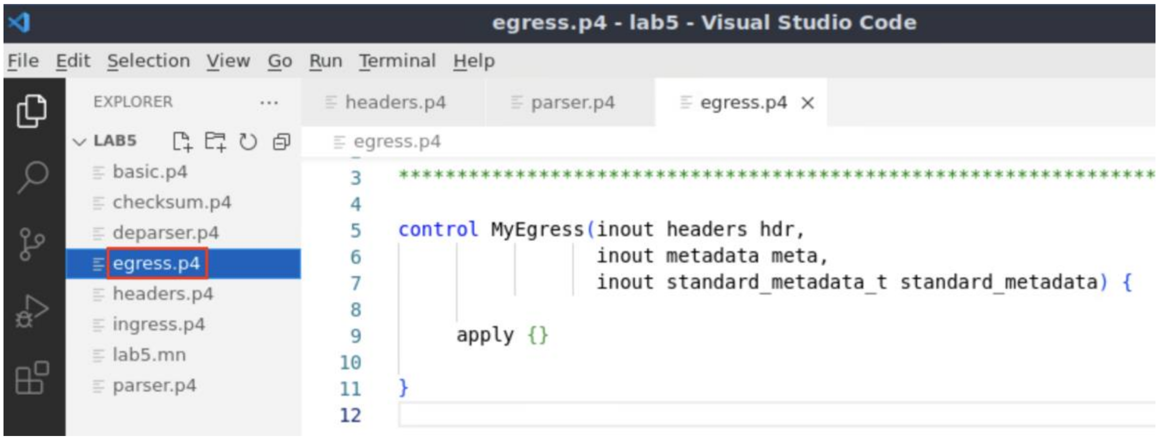


Figure 16. Inspecting the egress processing block.

Step 2. Define the `modify` action by adding the following piece of code.

```
action modify(){
  hdr.switch_stats.switch_ID = 1;
  hdr.switch_stats.enq_timestamp = standard_metadata.enq_timestamp;
  hdr.switch_stats.deq_timestamp = standard_metadata.egress_global_timestamp;
  hdr.switch_stats.q_delay = standard_metadata.egress_global_timestamp
  - (bit<48>)standard_metadata.enq_timestamp;
  hdr.switch_stats.q_depth = (bit<24>)standard_metadata.enq_qdepth;

}
```

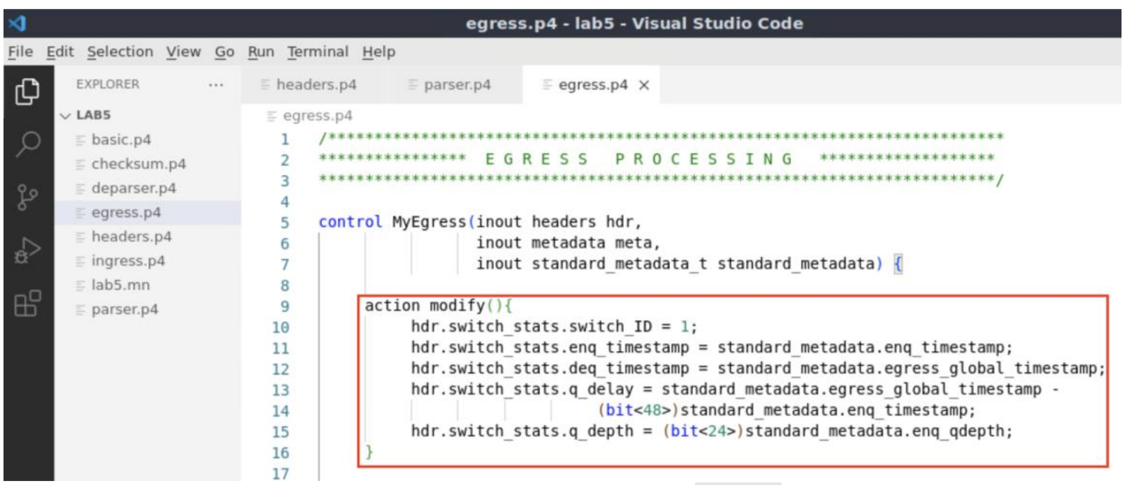


Figure 17. Defining the action `modify`.

Step 3. Apply the egress logic by adding the following piece of code.

```
apply {
        modify();
}
```

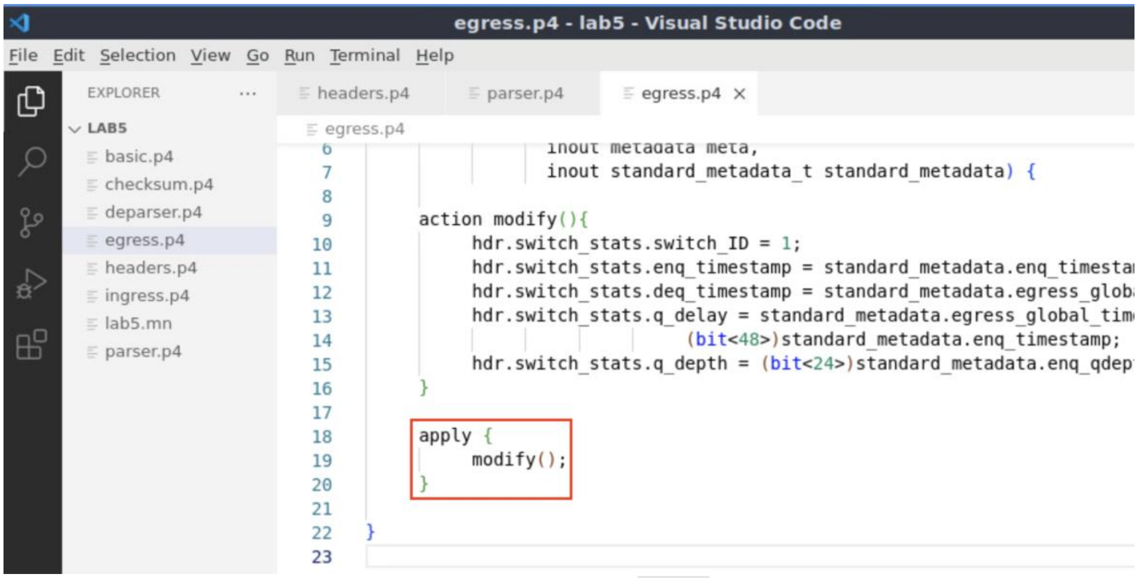


Figure 18. Defining the `apply` logic.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

4.3 Programing the deparser

Step 1. Click on the *deparser.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.

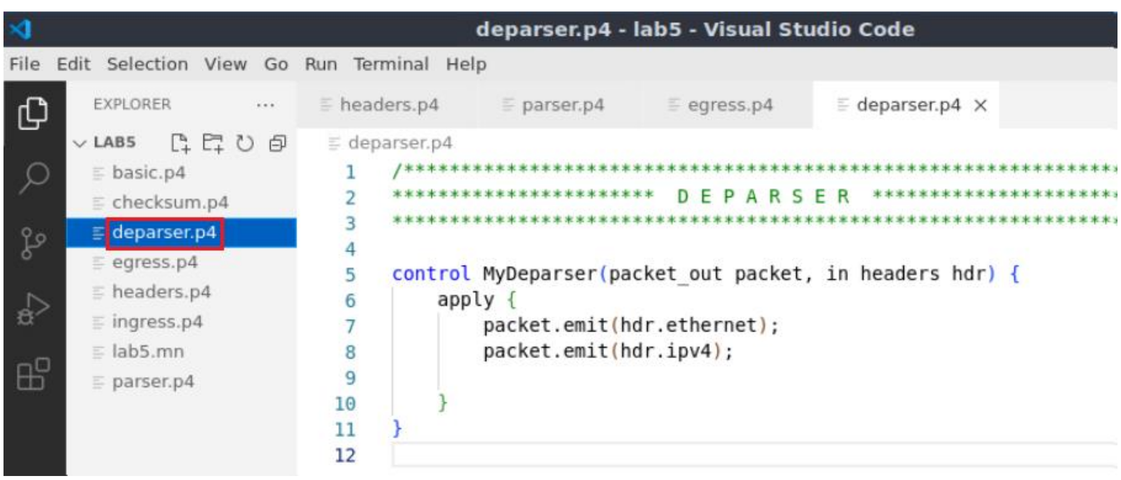


Figure 19. Opening the deparser processing block.

You will observe that the Ethernet and IPv4 header are already deparsed.

Step 2. Add the following line of code to emit the custom header.

```
packet.emit(hdr.switch_stats);
```

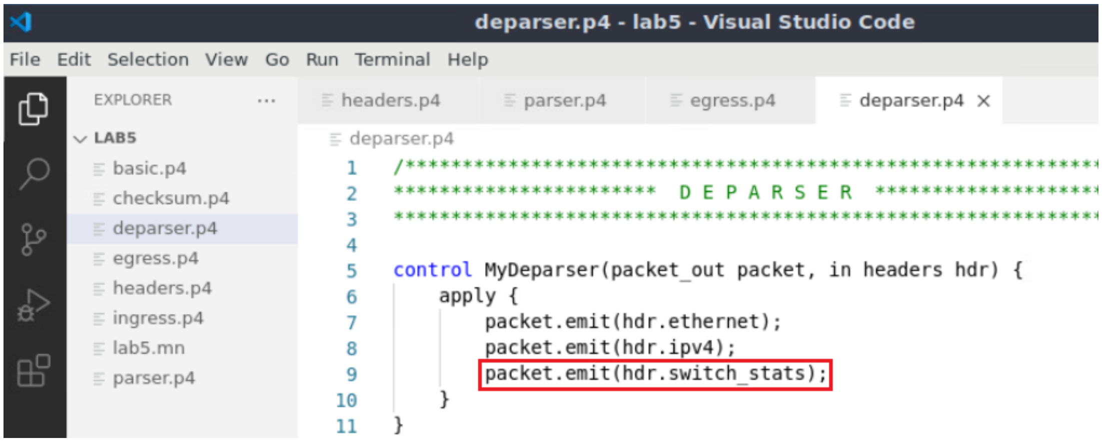


Figure 20. Emitting a custom header.

Step 3. 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 VS Code to compile the program.

```
p4c basic.p4
```

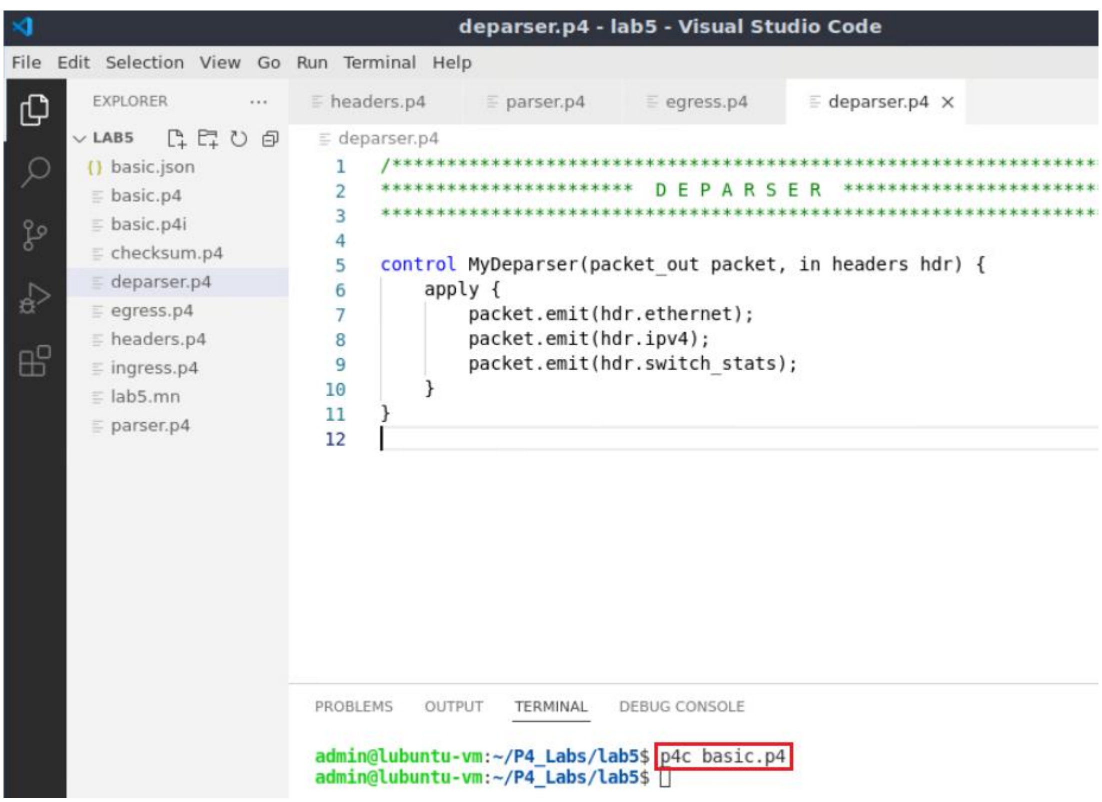


Figure 21. Compiling a P4 program.

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

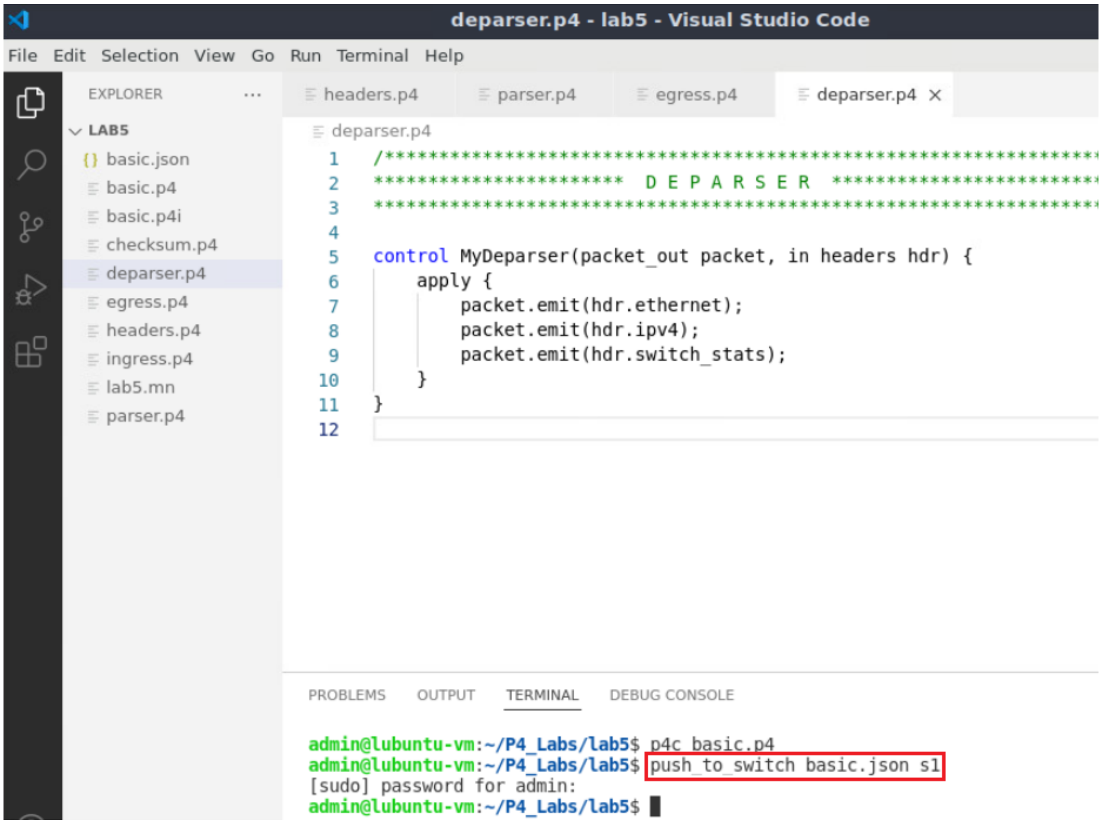


Figure 22. Pushing the *basic.json* file to switch s1.

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

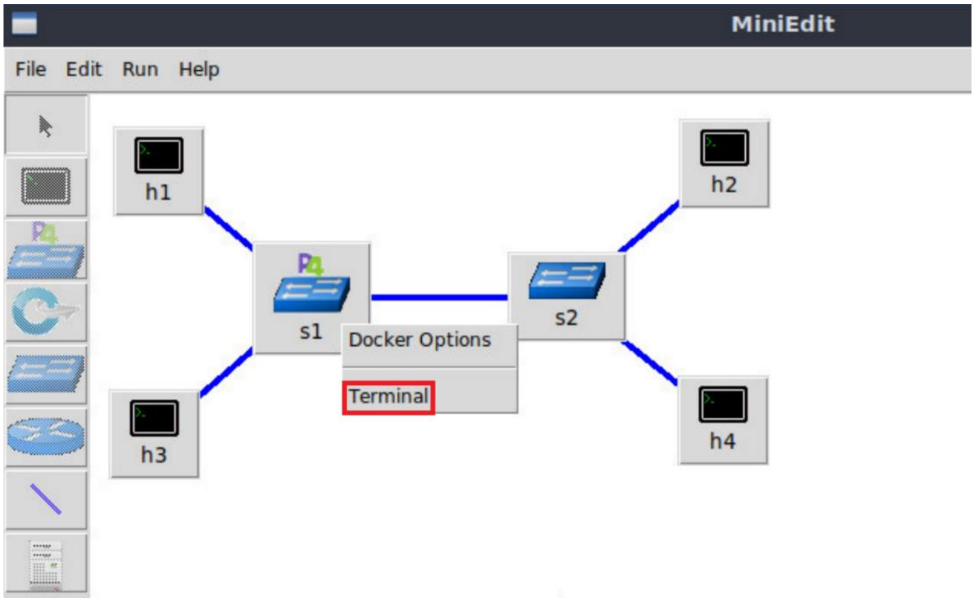


Figure 24. Starting the terminal on the switch.

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
```

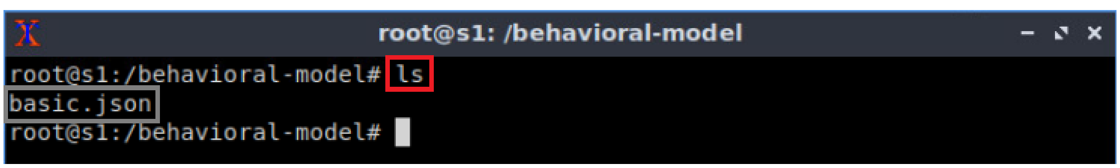


Figure 25. Displaying the contents of the current directory in the switch s1.

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

6 Configuring switch s1

6.1 Mapping P4 program's ports

Step 1. Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```

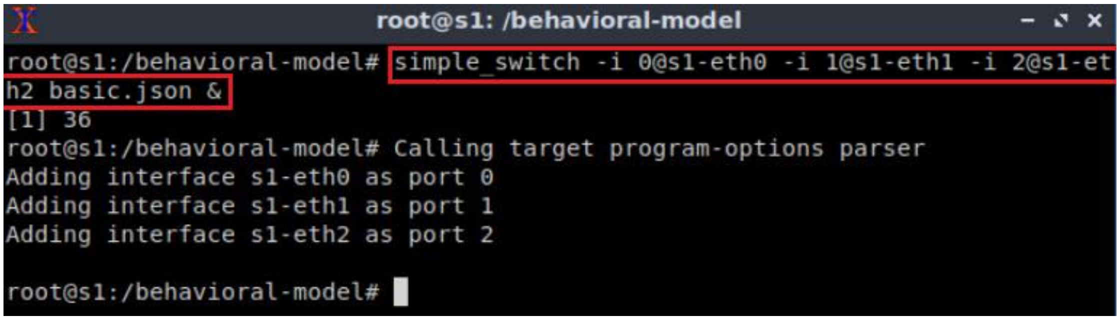


Figure 26. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

6.2 Loading the rules to the switch

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

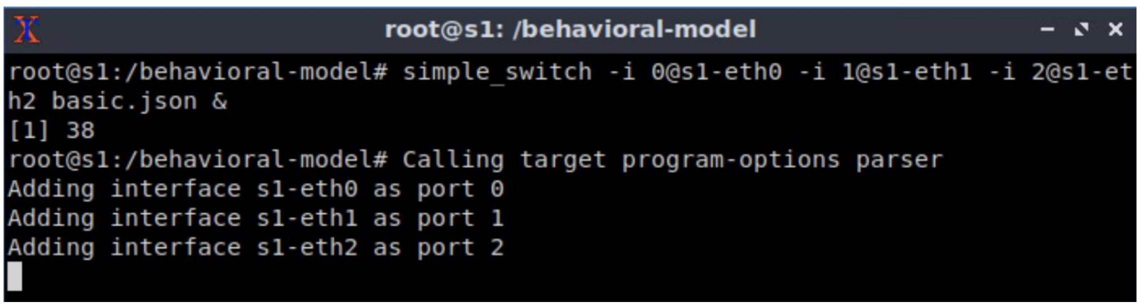


Figure 27. Returning to switch s1 CLI.

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

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

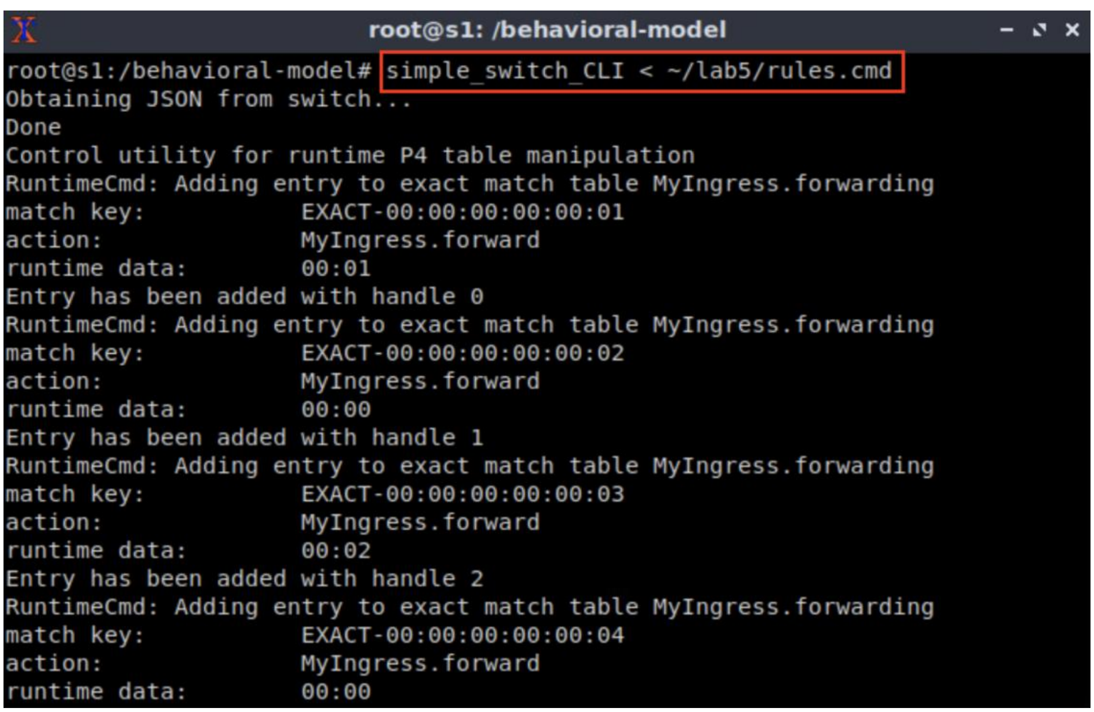


Figure 28. Populating the forwarding table into switch s1.

The script above pushes the rules into the match-action table `forwarding`.

7 Testing and verifying the P4 program

In this section, the user will test the P4 program by generating background traffic and sending a packet with the custom header `switch_stats`. The purpose of the background traffic is to fill the switch's queue. Then, the P4 program will insert queueing information into the custom header. The values in the custom headers are observed from a receiver.

7.1 Setting the queue length

Step 1. Type the following command to start switch s1's CLI.

```
simple_switch_CLI
```

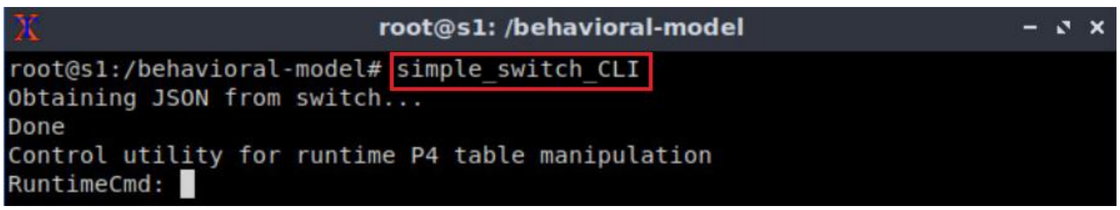


Figure 29. Starting switch s1's CLI.

Step 2. Set the queue rate by issuing the following command.

```
set_queue_rate 8333
```

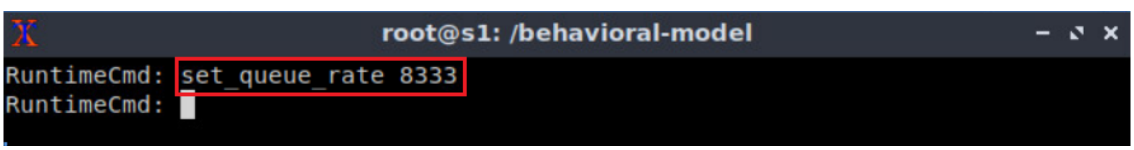


Figure 30. Setting the queue rate in switch s1.

Note that the queue rate value 8333 packets per second. This value is calculated as follows: Consider the the maximum rate is 100 Mbps and the Maximum Transmission Unit (MTU) is 1500 bytes/packet (i.e., 12,000 bits/packet). Thus, the number of packet that the queue must serve per second corresponds to the following value.

$$q_{rate} = \frac{Maximum\ rate\ [bits/s]}{Packet\ size\ [bits/packet]} = \frac{100{,}000{,}000\ [bits/s]}{12{,}000[bits/packet]} \approx 8333\ packets/s$$

With this value, the sending rate is 100Mbps.

Step 3. Set switch s1's buffer size (queue depth) by issuing the following command.

```
set_queue_depth 1666
```

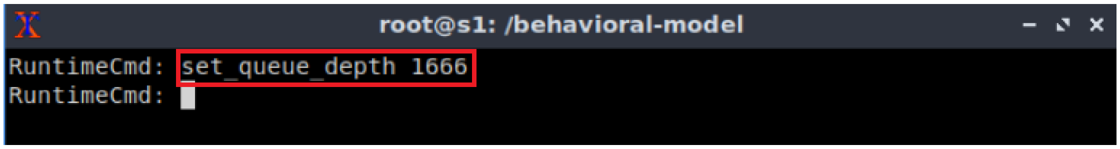


Figure 31. Setting the queue rate in switch s1.

In the figure above, the buffer size is set to 1666 packets (i.e., ~2.5Mbytes), which correspond to ten Bandwidth-Delay Product (BDP)[2]. The BDP value was calculated considering a bandwidth of 100Mbps and a maximum delay of 20ms.

$$BDP = BW * delay = 100{,}000{,}000[bits/s] * 0.02[s] = 2{,}000{,}000\ [bits]$$
$$= 250{,}000\ [bytes] \approx 166\ [packets]$$

$$10 * BDP = 1666\ [packets]$$

7.2 Testing the configuration

Step 1. Go back to MiniEdit and open a terminal in host h4 and start an iperf3 server by issuing the following command.

```
iperf3 -s
```

Figure 32. Starting an iperf3 server in host h4.

Step 2. Open a terminal in host h3 and run the following command to start an iperf3 client that will send data to the iperf3 server in host h4.

```
iperf3 -c 10.0.0.4
```

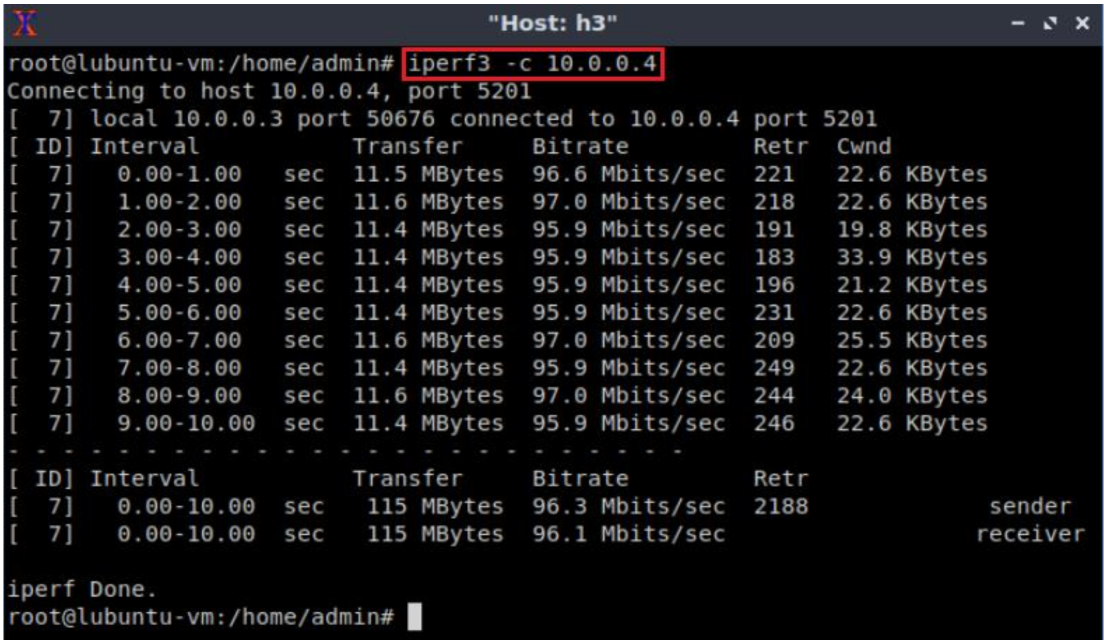


Figure 33. Starting an iperf3 client in host h3.

Note in the figure above that the bitrate of the data transfer is approximately 96.3Mbps which is close to the link bandwidth 100Mbps.

7.3 Starting the probing scripts

Step 1. Go back to MiniEdit and open a terminal on host h2. Issue the following command so that, host h2 starts listening for packets.

```
recv.py -p probe
```

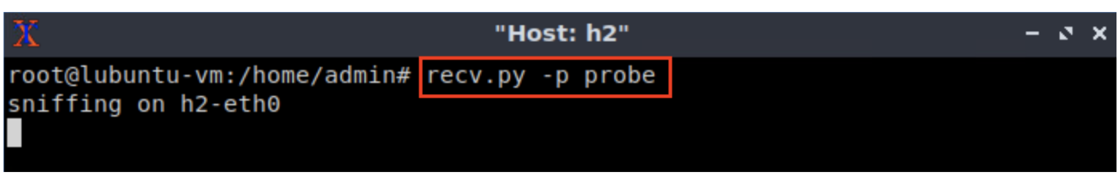


Figure 34. Listening for incoming packets in host h2.

The script above receives the following parameters:

- `-p`: enables listening to a specific protocol.
- `probe`: the protocol type.

Step 2. Open a terminal in host h1’s terminal, type the following command.

```
send.py 10.0.0.2 HelloWorld -p probe
```

```
"Host: h1"
root@lubuntu-vm:/home/admin# send.py 10.0.0.2 HelloWorld -p probe
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst       = 00:00:00:00:00:02
  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     = 253
     chksum    = 0x65cc
     src       = 10.0.0.1
     dst       = 10.0.0.2
     \options   \
###[ queue_statistics ]###
        switch_ID = 0
```

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

Similarly, the script above receives the following parameters:

- `10.0.0.2`: the destination IPv4 address.
- `HelloWorld`: the packet payload.
- `-p`: enables listening to a specific protocol.
- `probe`: the protocol type. Note that this protocol sends a custom packet every 10 milliseconds.

Step 3. Verify that the packet was received on host h2.

```
"Host: h2"
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 50
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = 253
     chksum    = 0x65cc
     src       = 10.0.0.1
     dst       = 10.0.0.2
     \options   \
###[ queue_statistics ]###
        switch_ID = 1
        enq_timestamp= 1157080600
        deq_timestamp= 1157080645
        q_delay    = 45
        q_length   = 0
###[ Padding ]###
           load      = 'HelloWorld'
```

Figure 36. Packet received on host h2.

Note that the value of the enqueueing timestamp (`enq_timestamp`) is 1,157,080,600 microseconds and the dequeuing timestamp (`deq_timestamp`) is 1,157,080,645 microseconds. The time difference (45 microseconds) indicates the processing time of the pipeline, and the queue length is zero.

7.5 Measuring the queue length with background traffic

Step 1. In host h3 and run the following command.

```
iperf3 -c 10.0.0.4 -t 120 -P 30
```

Figure 37. Starting an iperf3 client in host h3.

Step 2. Go back to host h2 and observe the evolution of the `time_diff` and `q_length` fields.

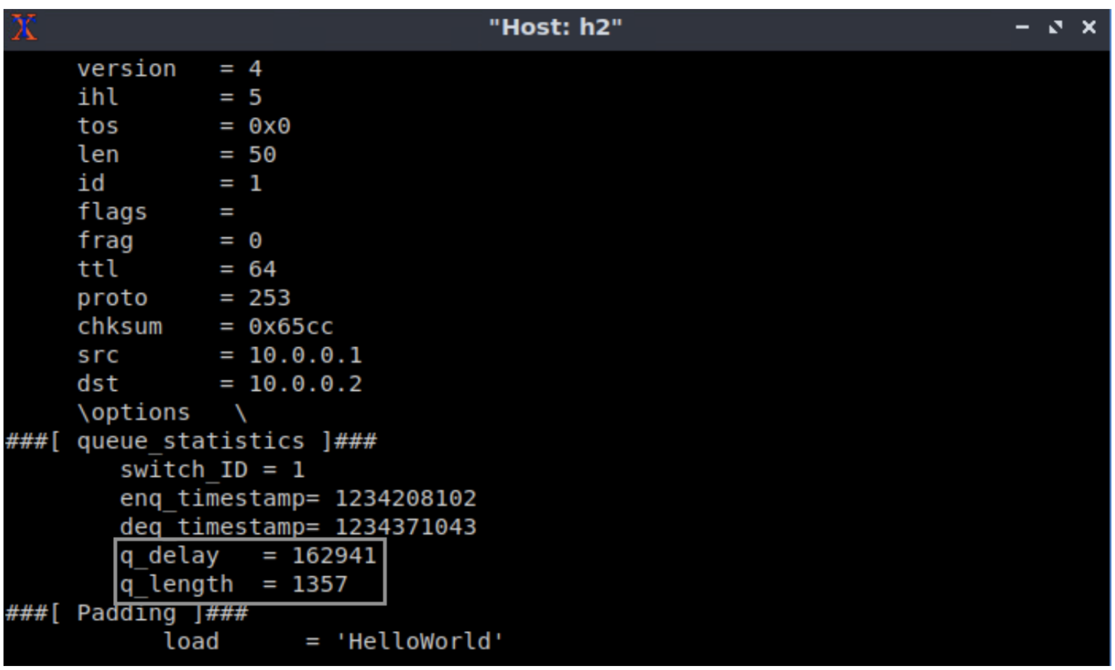


Figure 38. Visualizing the evolution of the processing time and queue length.

The figure above shows that the queuing delay (`q_delay`) is 162,941 microseconds (~162 milliseconds). Note that queue length is greater than zero while the iperf3 test is running.

Step 3. Go back to MiniEdit and open another terminal in host h1 and run a ping test.

```
ping 10.0.0.4
```

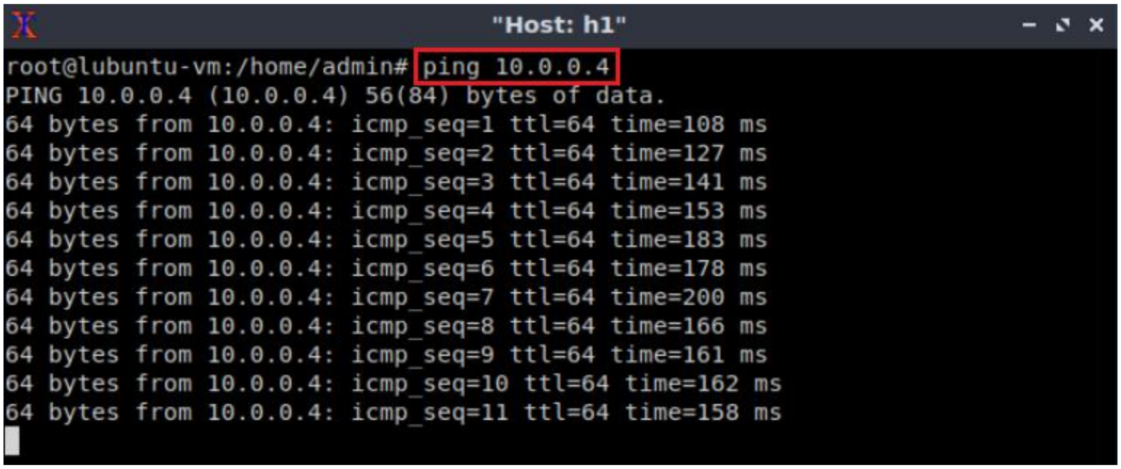


Figure 39. Measuring the round-trip time (RTT) between host h1 and host h4.

Note that RTT between host h1 and host h4 is up to 200 milliseconds due to bufferbloat[2].

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

References

1. RFC 791. "*Internet Protocol.*" 1981.
2. J. Crichigno, E. Kfoury, E. Bou-Harb, N. Ghani. "*High-Speed Networks: A Tutorial.*" [Online]. Available: https://tinyurl.com/3dkbf7d7
3. Mininet walkthrough. [Online]. Available: http://Mininet.org.
4. 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.
5. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28.*" [Online]. Available: https://tinyurl.com/rruscv3.
6. P4lang/behavioral-model github repository. "*The BMv2 simple switch target.*" [Online]. Available: https://tinyurl.com/vrasamm.

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 6: Collecting Queueing Statistics using a Header Stack

Document Version: 04-06-2022

Contents

Overview

This lab introduces P4 header stacks for collecting queue statistics. A header stack represents an array of headers that can be described in P4. This lab shows how to define, parse, and compute header stacks.

Objectives

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

1. Define header stacks in P4.
2. Parse headers with different lengths.
3. Append queue statistics into a custom header.
4. Visualize the evolution of the queue metrics in various switches.

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 and parsing a header stack.
4. Section 4: Processing a header stack.
5. Section 5: Loading the P4 program.
6. Section 6: Configuring the switches.
7. Section 7: Testing and verifying the P4 program.

1 Introduction to header stacks in P4

P4 provides the constructs to define, parse, and process header stacks. A header stack is an array of headers that a P4 programmable switch can parse. This capability enables applications to collect information from the switches that a packet transits.

1.1 Lab scenario

Figure 1 shows a topology with two end hosts and three P4 programmable switches. Consider a scenario where a packet departing from host h1 (sender) can reach host h2 (receiver) taking two paths: 1) h1-s1-s2-h2 and, 2) h1-s1-s3-s2-h2. Along the way, the packet collects information from the switches. This information includes:

- Switch ID.
- Ingress timestamp.
- Egress timestamp.
- Time difference between egress and ingress timestamps.
- Queue length.

The receiver host h2 can observe two or three headers in the stack depending on the path taken by the packet. In this lab, the user will create a P4 program that uses header stacks to insert the information listed above in a packet.

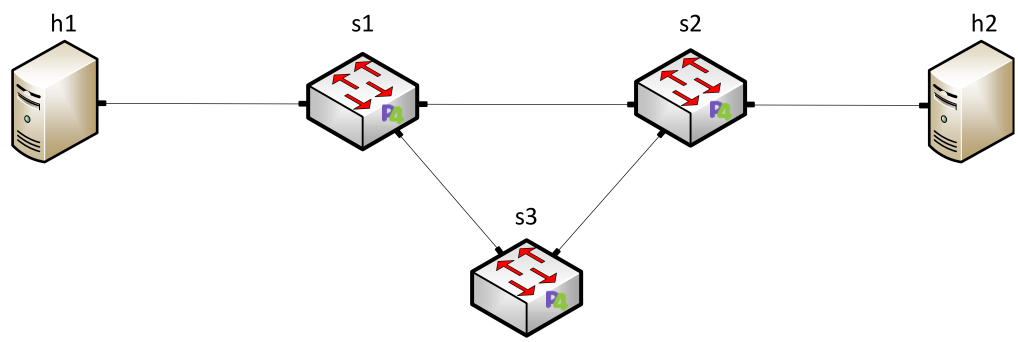


Figure 1. Collecting queue statistics with P4 switches.

1.2 Defining a header stack

Figure 2 shows the definition of a header stack. Assume that the Ethernet and IPv4 headers are already defined, and the header stack will be over the UDP header (see lines 3-8). The definitions in the P4 code are explained as follows:

- Line 10-12: defines the custom header type `layer_t`. This header will be used to store the number of headers in the stack.
- Line 14-20: defines a custom header that will collect information from each switch.
- Line 22-24: defines the parser metadata header type.
- Line 26-28: declares the metadata header used to store the parser's metadata.
- Line 30-34: declares the header used by the P4 program. Note that the header stack `sw_stats` is declared as an array of `MAX_HOPS` elements.

```
#define MAX_HOPS 8

header udp_t {
    port_t srcPort;
    port_t dstPort;
    bit<16> len;
    bit<16> checksum;
}

header layer_t {
    bit<16> count;
}

header sw_stats_t {
    switch_ID_t switch_ID;
    bit<48> ingress_timestamp;
    bit<48> egress_timestamp;
    bit<48> time_diff;
    bit<24> q_depth;
}

struct parser_metadata_t {
    bit<16> remaining;
}

struct metadata {
    parser_metadata_t parser_metadata;
}

struct headers {
    ethernet_t              ethernet;
    ipv4_t                  ipv4;
    udp_t                   udp;
    layer_t                 layers;
    sw_stats_t[MAX_HOPS]    sw_stats;
}
```

Figure 2. Defining a header stack.

1.3 Parsing a header stack

Figure 3a shows a P4 code fragment that parses a header stack. Consider that the Ethernet and IPv4 headers are already parsed, thus the following code starts with UDP.

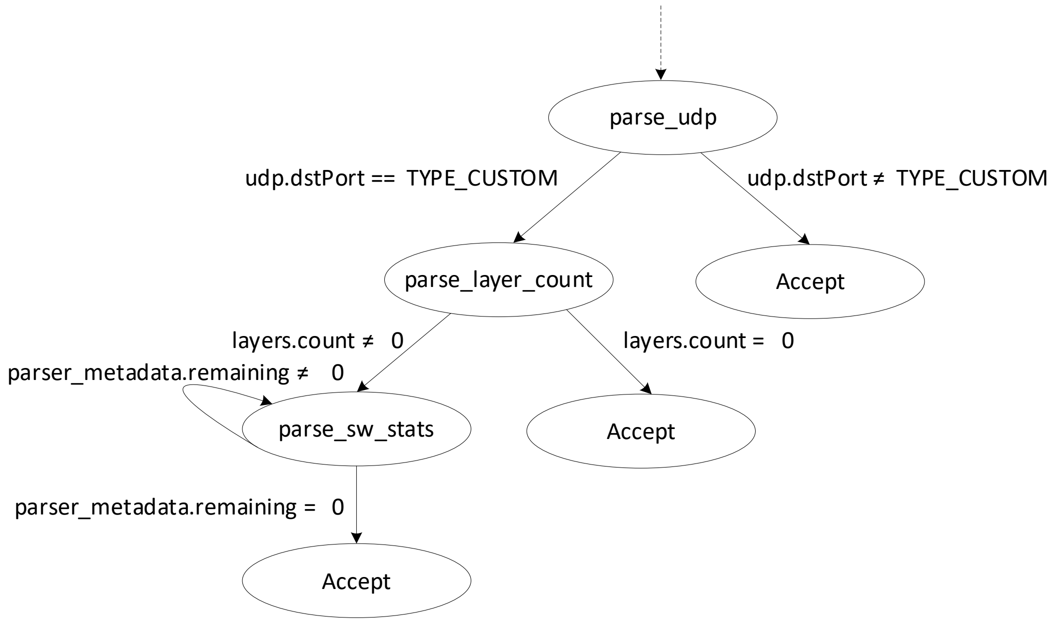
- Line 1: defines the state `parse_udp`.
- Line 2: extracts the values in the UDP header.
- Line 3: selects the next state based on the destination UDP port.
- Line 4: transitions to the state `parse_layer_count` when the destination port value is `TYPE_CUSTOM`.
- Line 5: specifies the default transition.
- Line 9: defines the state `parse_layer_count`.
- Line 10: extracts the values in the header `layers`.
- Line 11: stores the current layer count in the parser metadata.
- Line 12: selects the transition based on the layer count.
- Line 13: accepts the packet if the layer count is zero.

- Line 14: transitions to the state `parse_layer_count` when the destination port value is `TYPE_CUSTOM`.
- Line 18: defines the state `parse_layer_count`.
- Line 19: extracts the top header in the stack.
- Line 20-21: decrements the number of headers in the stack.
- Line 22: selects the transition based in the number of the remaining headers in the stack.
- Line 23: accepts the packet if the remaining headers in the stack are zero.
- Line 24: invokes the state `parse_sw_stats`. Note that this statement is reclusively called until all the headers in the stack is parsed.

Figure 3b summarizes the states and transitions described in the P4 code fragment.

```
 1: state parse_udp {
 2:         packet.extract(hdr.udp);
 3:         transition select(hdr.udp.dstPort) {
 4:             TYPE_CUSTOM: parse_layer_count;
 5:             default: accept;
 6:         }
 7:     }
 8:
 9: state parse_layer_count {
10:     packet.extract(hdr.layers);
11:     meta.parser_metadata.remaining = hdr.layers.count;
12:     transition select(hdr.layers.count){
13:         0: accept;
14:         default: parse_sw_stats;
15:     }
16: }
17:
18: state parse_sw_stats {
19:     packet.extract(hdr.sw_stats.next);
20:     meta.parser_metadata.remaining =
21:                 meta.parser_metadata.remaining - 1;
22:     transition select(meta.parser_metadata.remaining){
23:         0 : accept;
24:         default: parse_sw_stats;
25:     }
26: }
```

(a)



(b)

Figure 3. Parsing a header stack. (a) Fragment of a P4 code that parses a header stack. (b) Graphical representation of the states, transitions, and conditions in the parser.

2 Lab topology

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

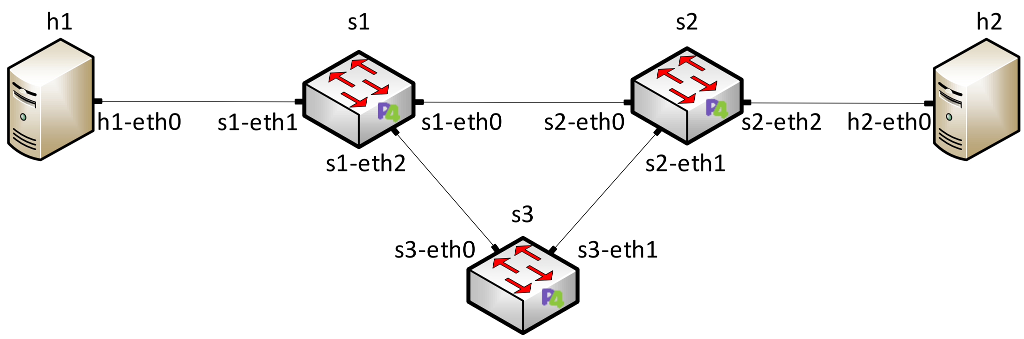


Figure 4. Lab topology.

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 *lab6* folder and search for the topology file called *lab6.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

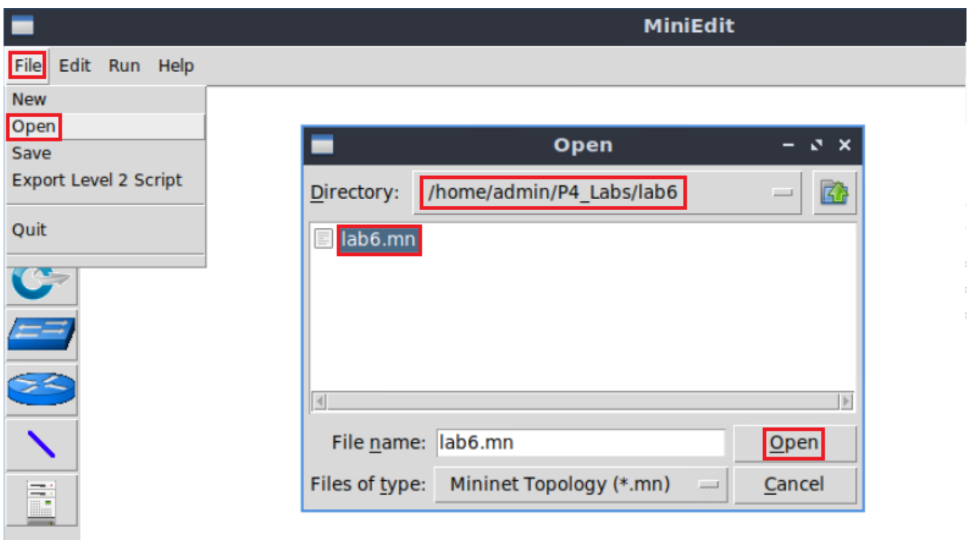


Figure 6. MiniEdit's *Open* dialog.

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 7. 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.

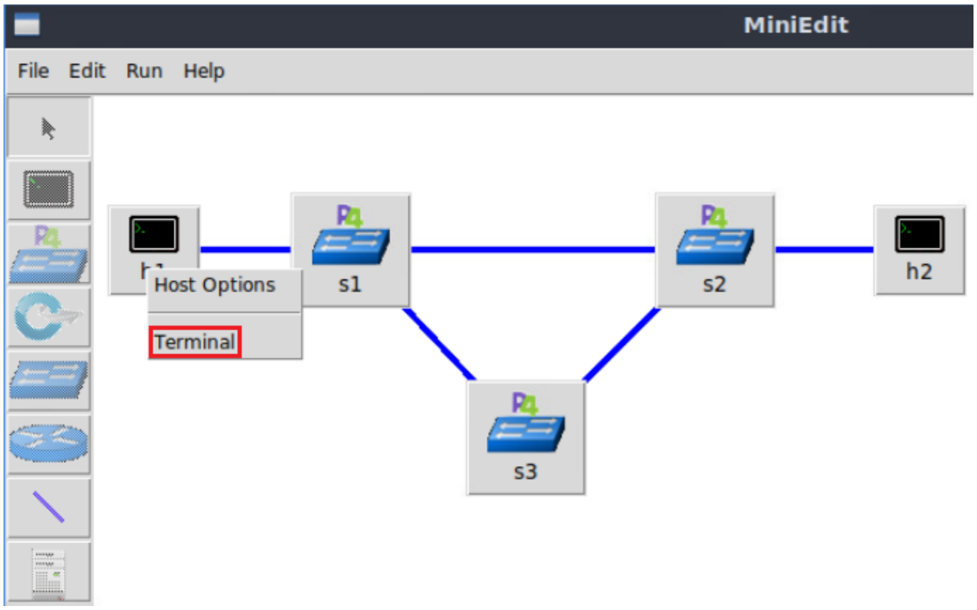


Figure 8. Opening a terminal on host h1.

Step 2. Run the following command to display the information of the interfaces on host h1.

```
ifconfig
```

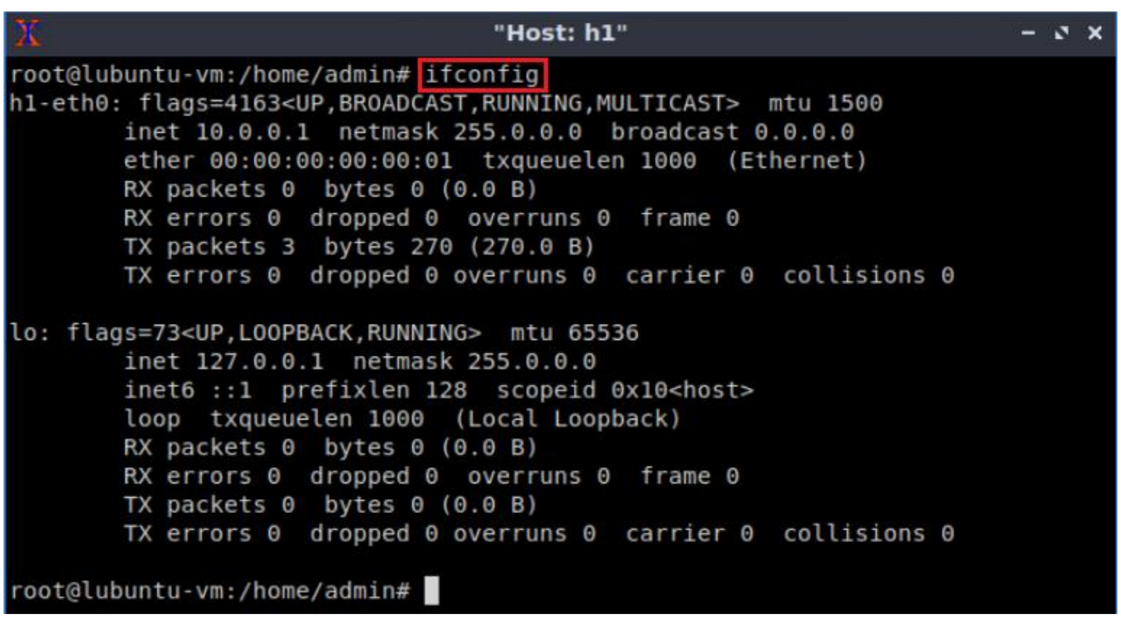


Figure 9. Displaying interfaces' information on host h1.

3 Defining and parsing a header stack

In this section, you will define and parse a header stack. The header stack stores queue statistics consisting of the switch ID, the ingress timestamp, the egress timestamp, the time difference between the previous timestamps, and the queue length. These values are part of the standard metadata collected from each switch. Then, you will define the parsing logic, which follows the graph described in Figure 3b. Note that a new header is appended every time a packet traverses a switch.

3.1 Loading the programming environment

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

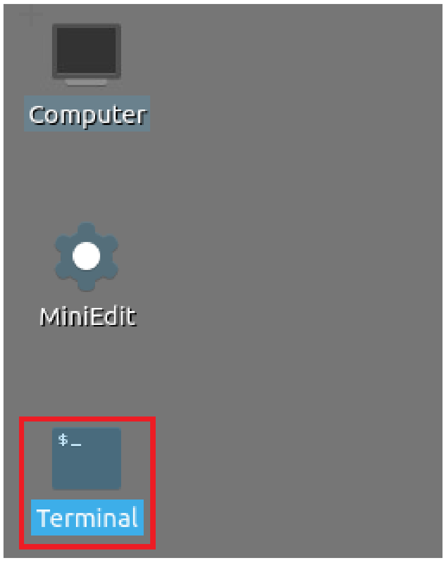


Figure 10. Shortcut to open a Linux terminal.

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 execute.

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
```

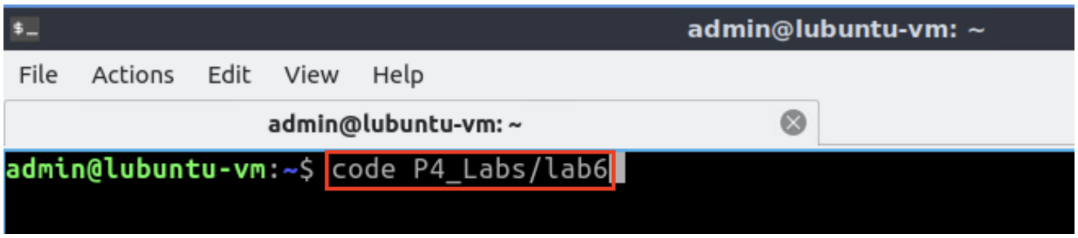


Figure 11. Loading the development environment.

3.2 Defining a header stack

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.

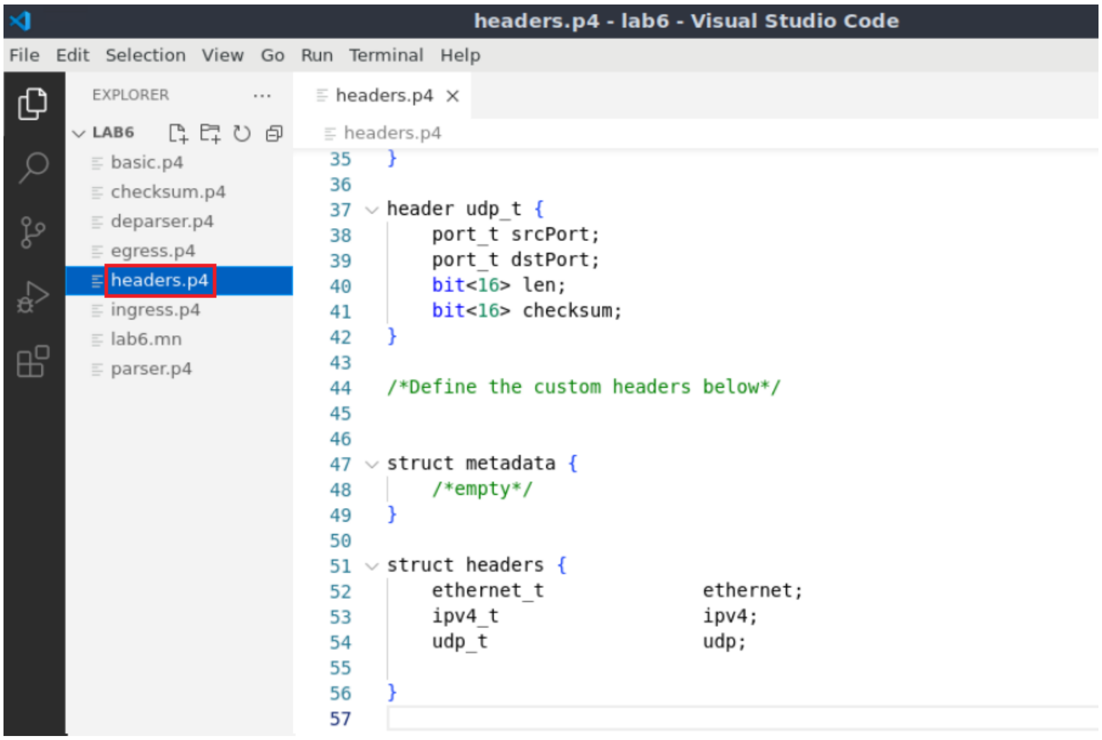


Figure 12. Inspecting the *headers.p4* file.

Step 2. Define the following header by inserting the next code into the *headers.p4*. file. The field will specify the number of custom headers added to the packet.

```
header layer_t {
        bit<16> count;
}
```

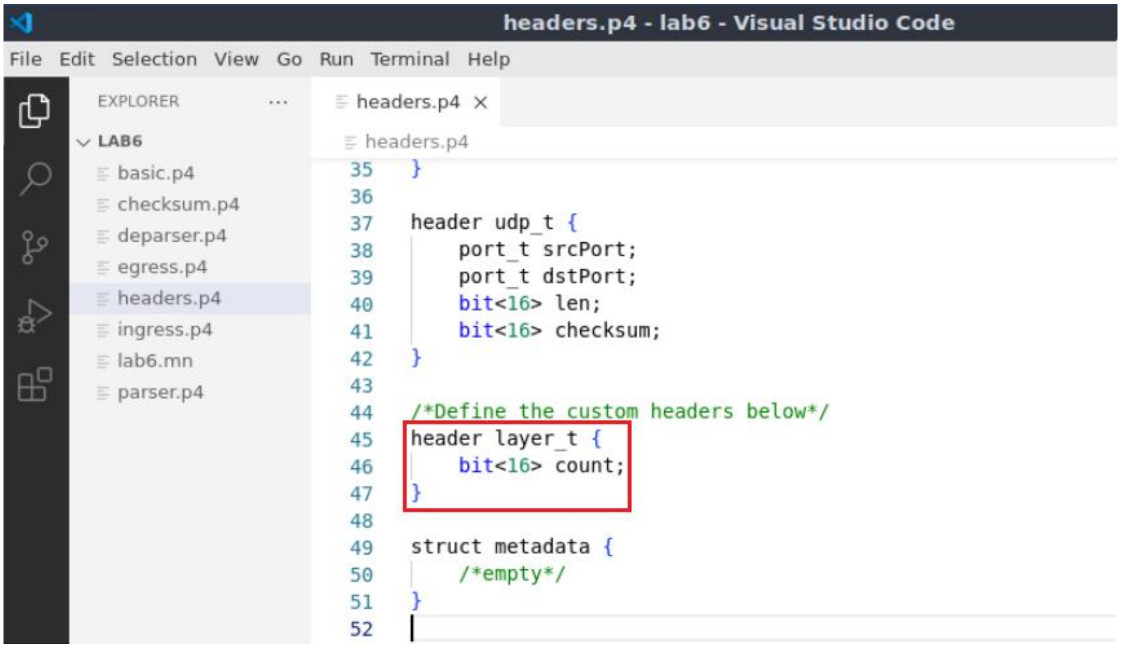


Figure 13. Defining a custom header.

Step 3. Define a custom header type by inserting the code shown below. This header consists of the switch ID, the ingress timestamp, the difference between the egress and the egress timestamps, and the queue length.

```
header sw_stats_t {
       bit<8>  switch_ID;
       bit<48> ingress_timestamp;
       bit<48> egress_timestamp;
       bit<48> time_diff;
       bit<24> q_depth;
}
```

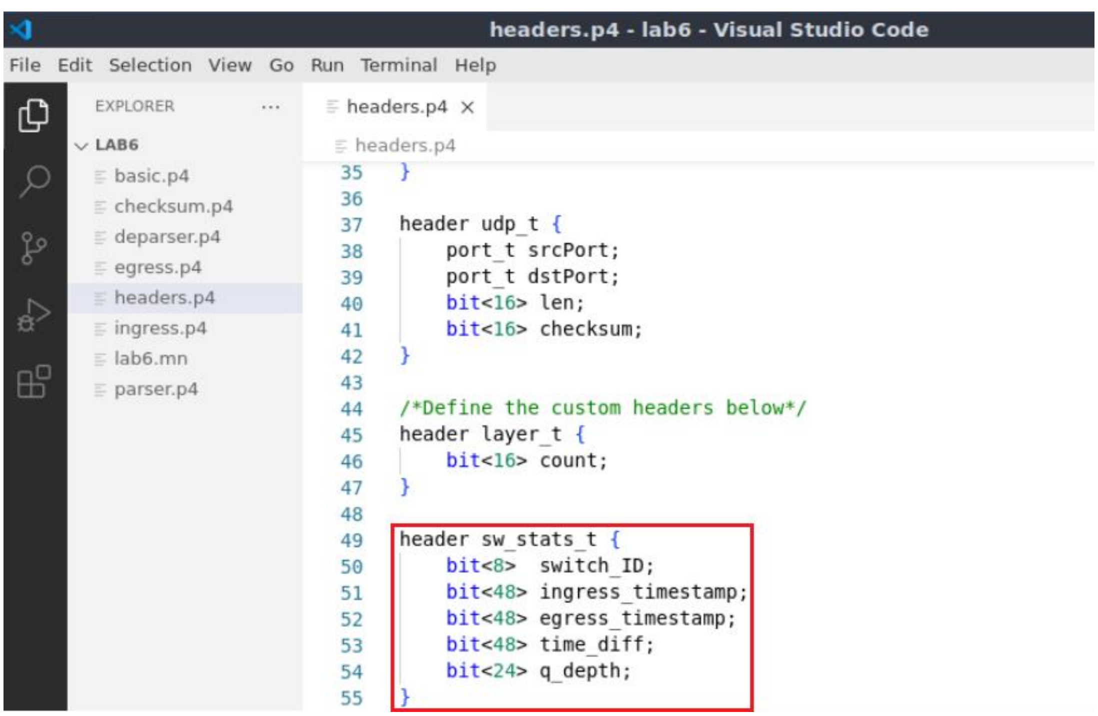


Figure 14. Defining a custom header data structure.

Step 4. Define a custom metadata type by adding the following code. This metadata stores the number of remaining headers to be parsed.

```
struct parser_metadata_t {
       bit<16> remaining;
}
```

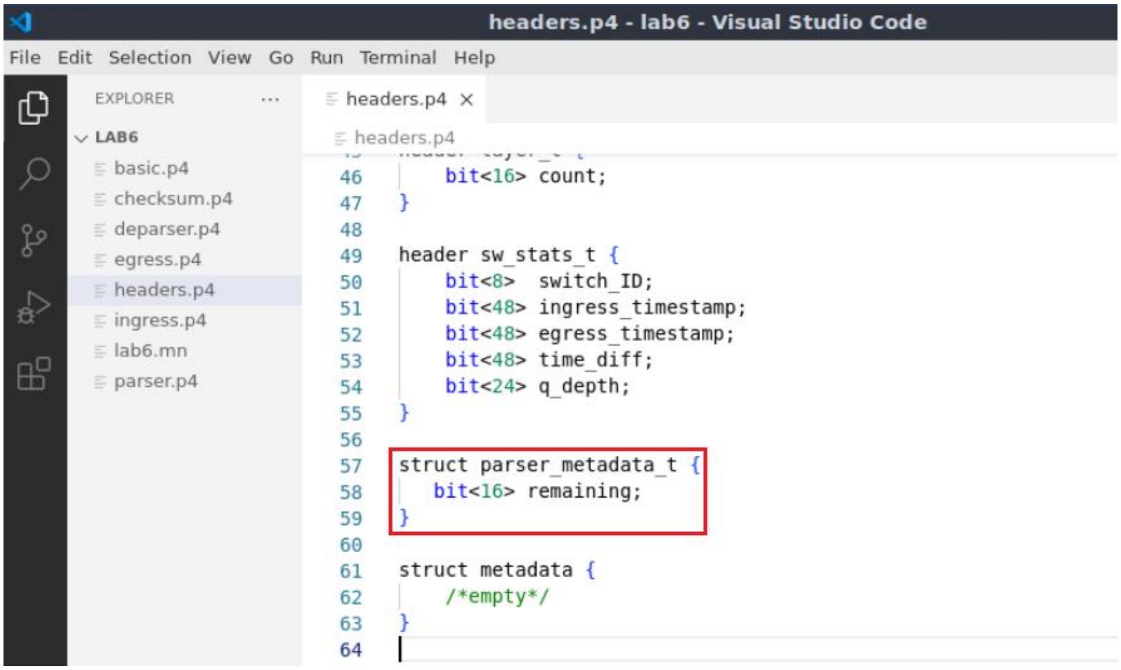


Figure 15. Defining a custom metadata type.

Step 5. Include the custom metadata `parser_metadata_t` into the metadata data structure by adding the following line.

```
parser_metadata_t parser_metadata;
```

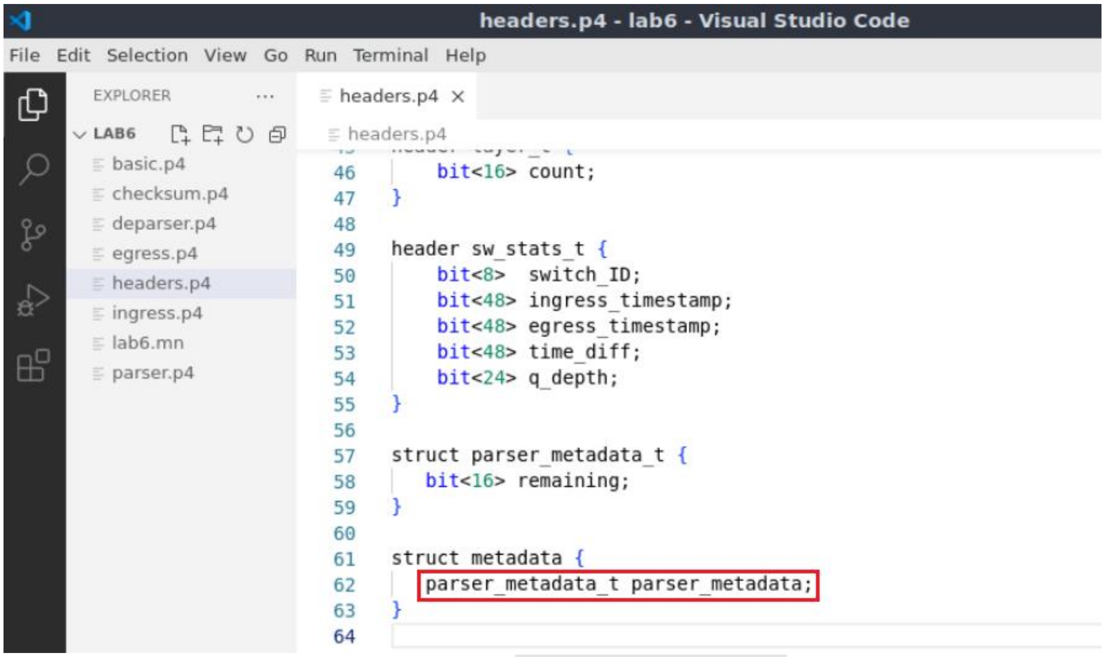


Figure 16. Including the custom metadata `parser_metadata_t` into the metadata struct.

Step 6. Add the custom headers to the packet header definition by including the following lines. Note that the header `sw_stats_t` has `MAX_HOPS` elements.

```
layer_t                                 layers;
sw_stats_t[MAX_HOPS]                    sw_stats;
```

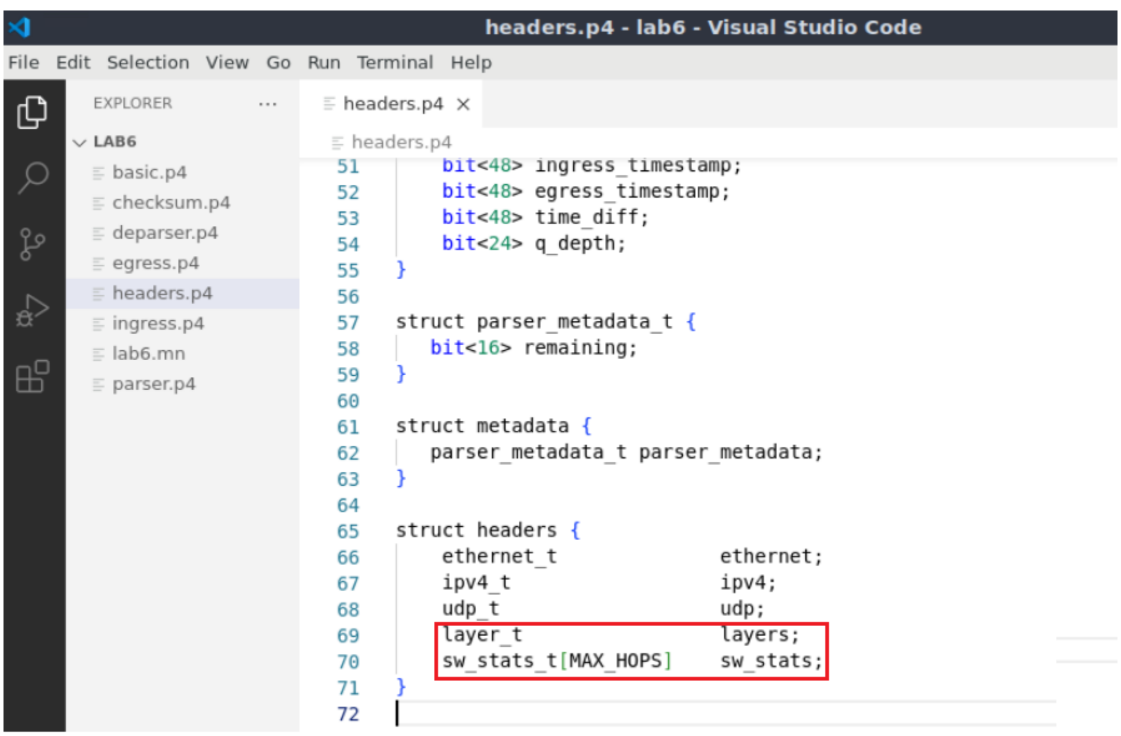


Figure 17. Adding the custom headers to the header’s definition.

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

3.3 Parsing a custom header

Step 1. Click on the *parser.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.

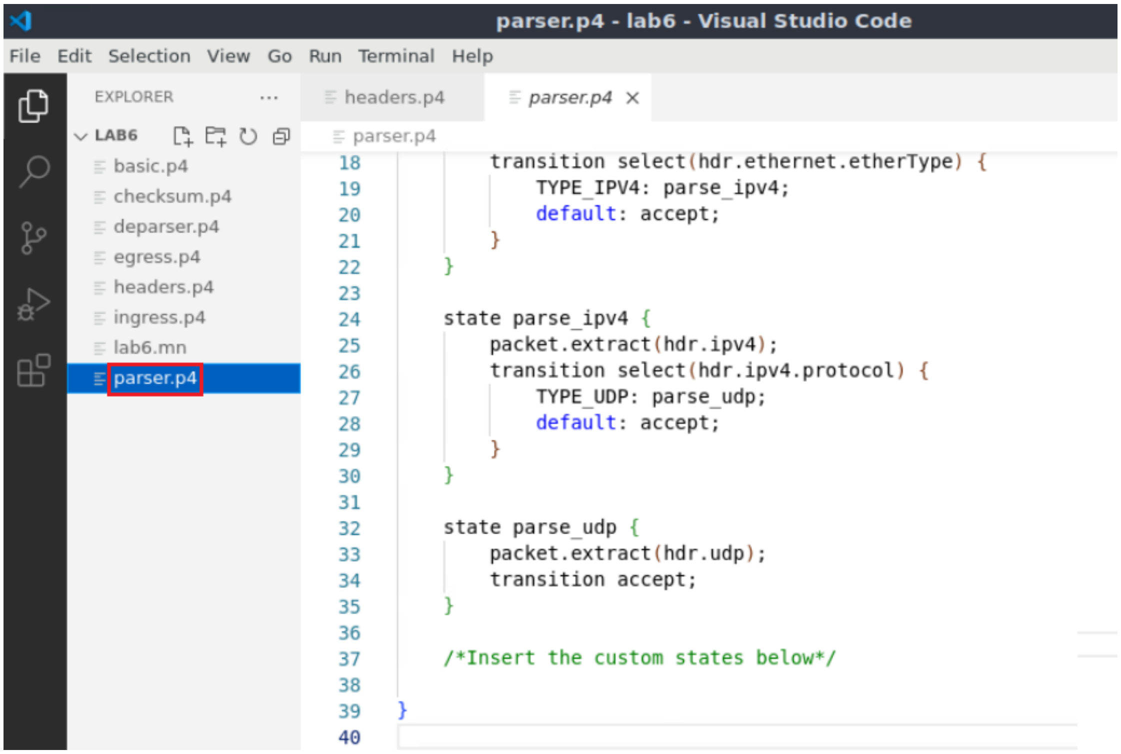


Figure 18. Inspecting the *parser.p4* file.

Step 2. Define the state `parser_layer_count` to parse the custom header by adding the following piece of code.

```
state parse_layer_count {
        packet.extract(hdr.layers);
        meta.parser_metadata.remaining = hdr.layers.count;
        transition select(hdr.layers.count){
                0: accept;
                default: parse_sw_stats;
        }
}
```

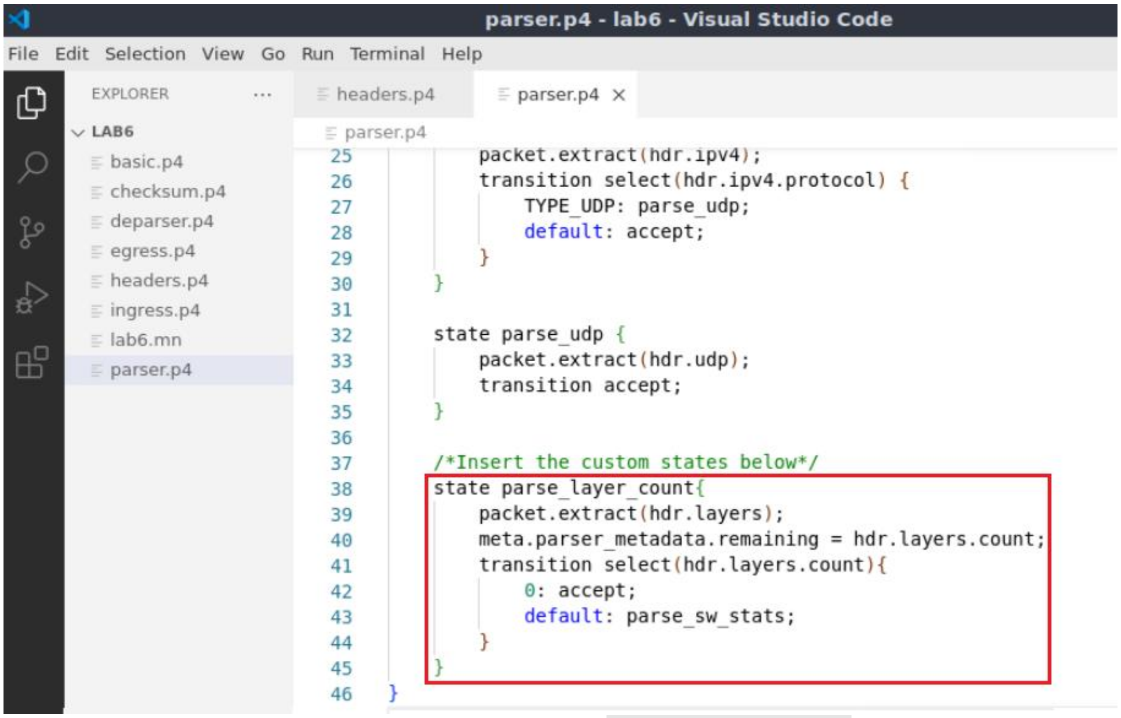


Figure 19. Defining the state `parse_layer_count`.

The code in the figure above extracts the value of `hdr.layers` which contains the number of custom headers in the packet. Such value is stored in the parser metadata `meta.parser_metadata.remaining`. The packet is accepted if the number is zero; otherwise, the parser transitions to the `parse_sw_stats` state.

Step 3. Define another state to parse the header stack `parser_sw_stats` by adding the following piece of code.

```
state parse_sw_stats {
	packet.extract(hdr.sw_stats.next);
	meta.parser_metadata.remaining =
			meta.parser_metadata.remaining - 1;
	transition select(meta.parser_metadata.remaining){
		0: accept;
		default: parser_sw_stats;
	}
}
```

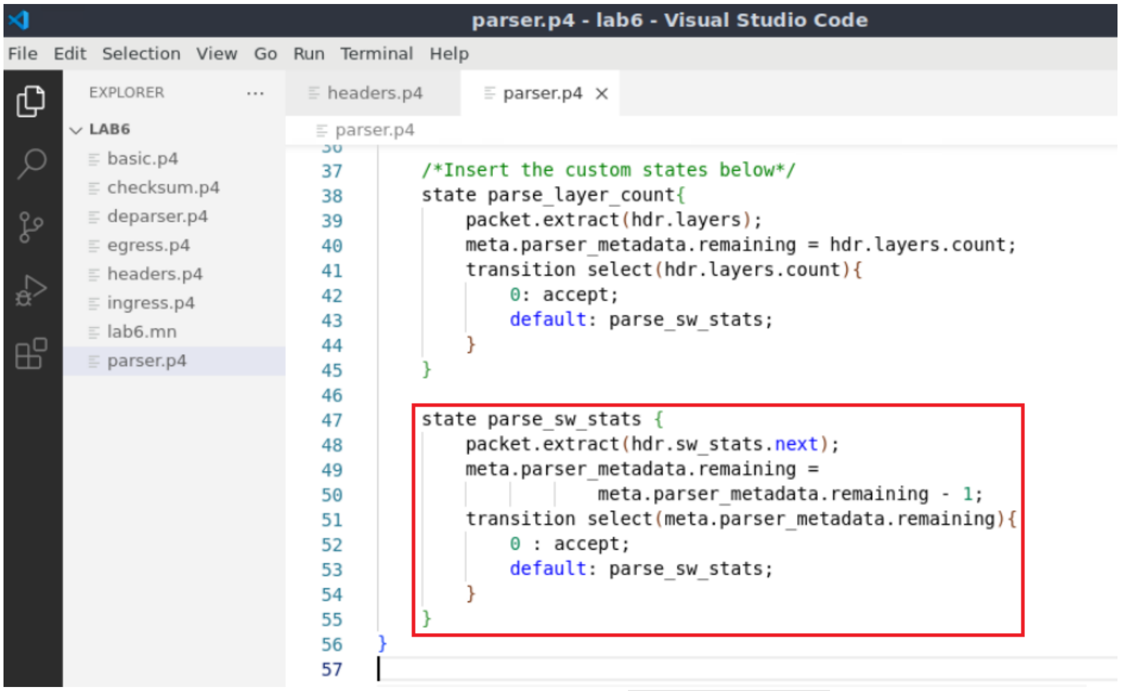


Figure 20. Defining the state `parse_sw_stats`.

The code in the figure above extracts the information from the header `stack` `hdr.sw_stats` by using the `.next` statement. After extracting the header, the state decrements the `remaining` field. The transition depends on the `remaining` value. If zero, the packet is accepted, meaning that there are no more layers to extract; otherwise, the process is repeated recursively.

Step 4. Now that the parsing states for the header stack are defined, you will modify the UDP parser to transition to `parse_layer_count` and `parse_sw_stats`. Scroll up and change the transition statement in the `parse_udp` state by adding the following statements.

```
transition select(hdr.udp.dstPort){
       TYPE_CUSTOM: parse_layer_count;
       default accept;
}
```

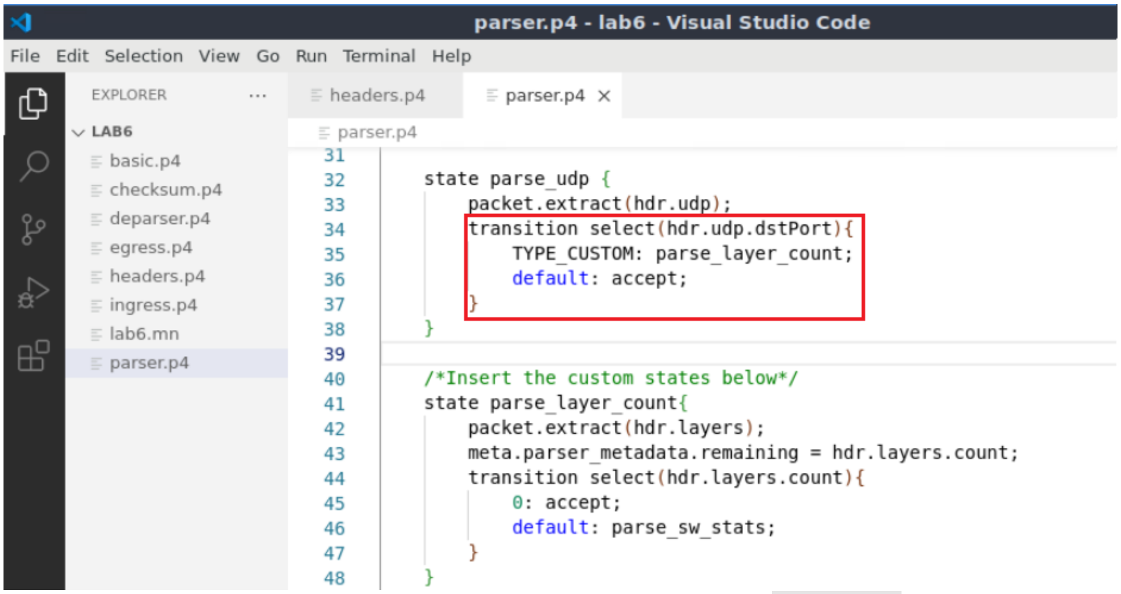


Figure 21. Modifying the transition statement in the `parse_udp` state.

Note that the transition from `parse_udp` to `parse_layer_count` is defined by the UDP destination port (i.e., `hdr.udp.dstPort`).

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

4 Processing a header stack

In this section, you will program the egress pipeline to assign the values of the switch ID, the egress timestamp, the ingress timestamp, the time difference, and the queue length to the fields in the header stack. These values are available in the switch's standard metadata. You will also perform an arithmetic operation between the egress and ingress timestamp to obtain the time difference when a packet finished the ingress block and started the egress block.

4.1 Programming the egress pipeline

Step 1. Click on the *egress.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.

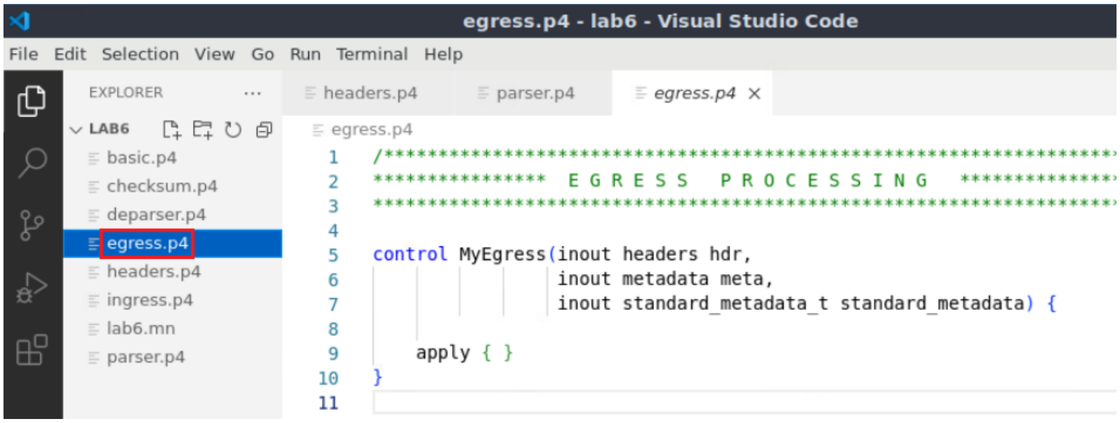


Figure 22. Inspecting the egress processing block.

Step 2. Define the `add_sw_stats` action by adding the following piece of code. Note that the action parameter is the switch ID.

```
action add_sw_stats(switch_ID_t ID){
  hdr.layers.count = hdr.layers.count + 1;
  hdr.sw_stats.push_front(1);
  hdr.sw_stats[0].setValid();
  hdr.sw_stats[0].ingress_timestamp = standard_metadata.ingress_global_timestamp;
  hdr.sw_stats[0].egress_timestamp = standard_metadata.egress_global_timestamp;
  hdr.sw_stats[0].time_diff = standard_metadata.egress_global_timestamp
                                     - standard_metadata.ingress_global_timestamp;
  hdr.sw_stats[0].q_depth = (bit<24>)standard_metadata.enq_qdepth;
  hdr.sw_stats[0].switch_ID = ID;
  hdr.ipv4.totalLen = hdr.ipv4.totalLen + 22;
  hdr.udp.len = hdr.udp.len + 32;
}
```

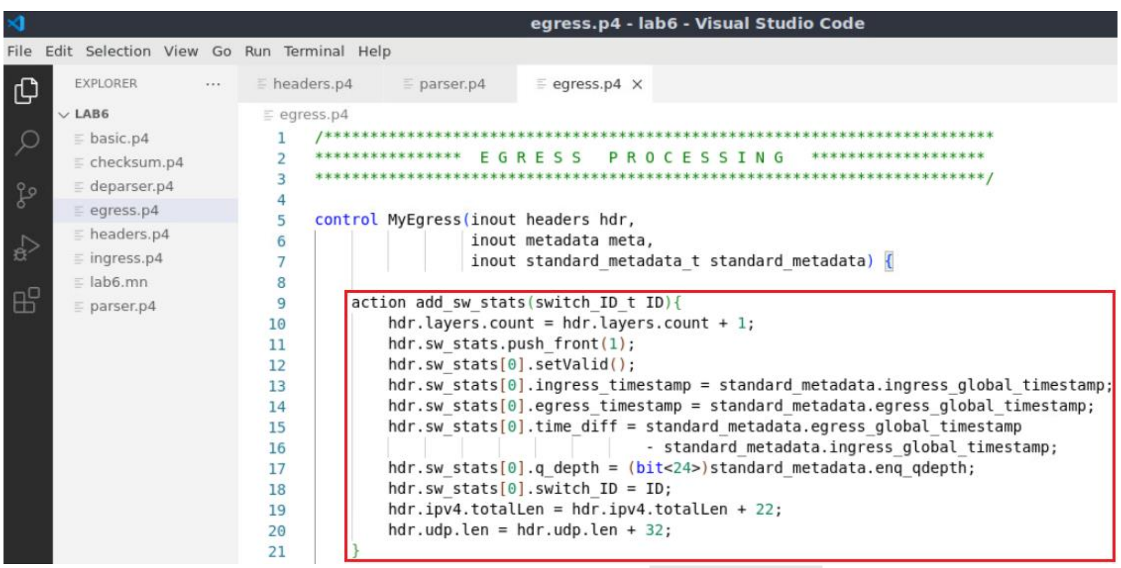


Figure 23. Defining the action `add_sw_stats`.

The code in the figure inserts metadata information into the header stack. It starts by increasing the layer count (see line 10) and adding it to the header stack (see line 11). After validating the new header (`hdr.sw_stats[0].setValid()`), the metadata information is added to the header stack (see lines 13-18). Finally, the IPv4 and UDP header lengths are updated to consider the new header.

Step 3. Define the table `add_queue_statistics` by adding the following piece of code.

```
table add_queue_statistics {
        key = {
                hdr.udp.dstPort: exact;
        }
        actions = {
                add_sw_stats;
                NoAction;
        }
        size = 32;
        default_action = NoAction;
}
```

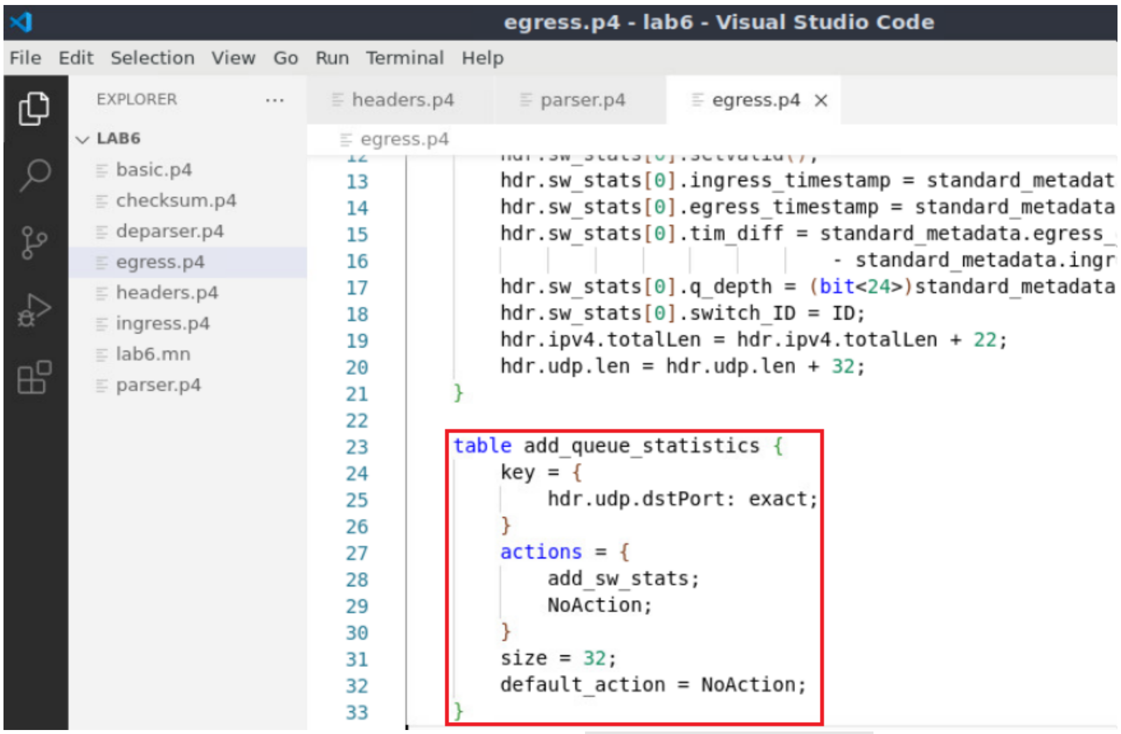


Figure 24. Defining the table `add_queue_statistics`.

The table in the figure above matches the UDP destination port. It can execute two actions `add_sw_stats` and `NoAction`. The table allocates 32 entries and the default action, which occurs when the key value is not present in the table, is `NoAction`.

Step 4. Apply the egress logic by adding the following piece of code. This code applies the `add_queue_statistics` table when the `layers` header is valid.

```
apply {
    if(hdr.layers.isValid()) {
        add_queue_statistics.apply();
    }
}
```

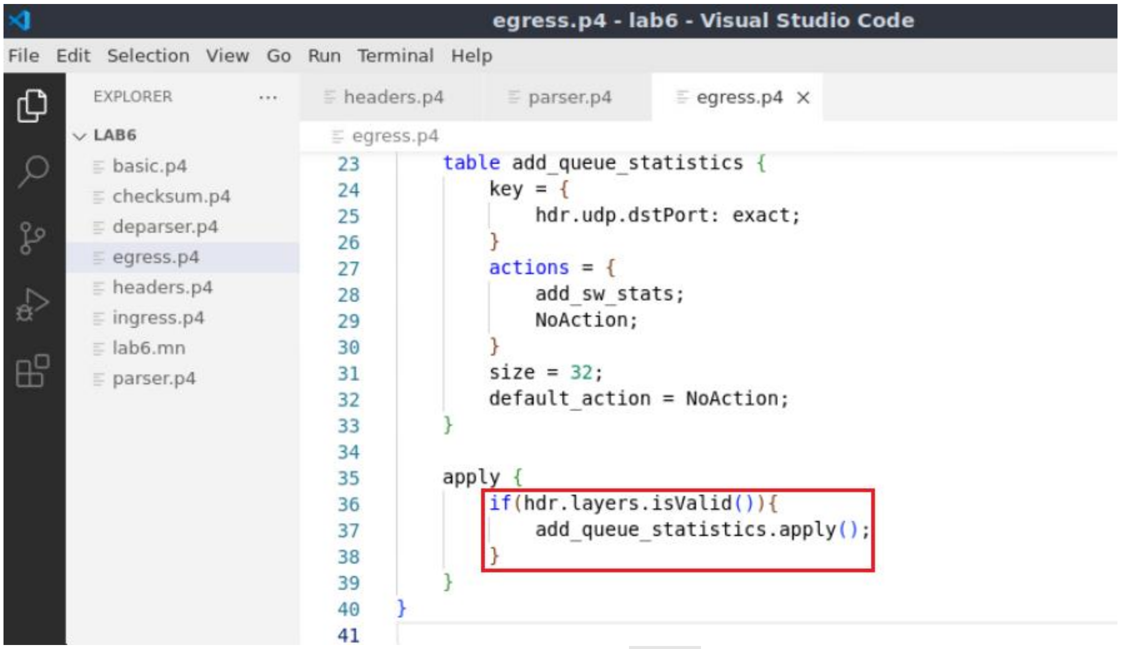


Figure 25. Defining the `apply` logic.

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

4.2 Programing the deparser to emit a custom header

Step 1. Click on the *deparser.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.

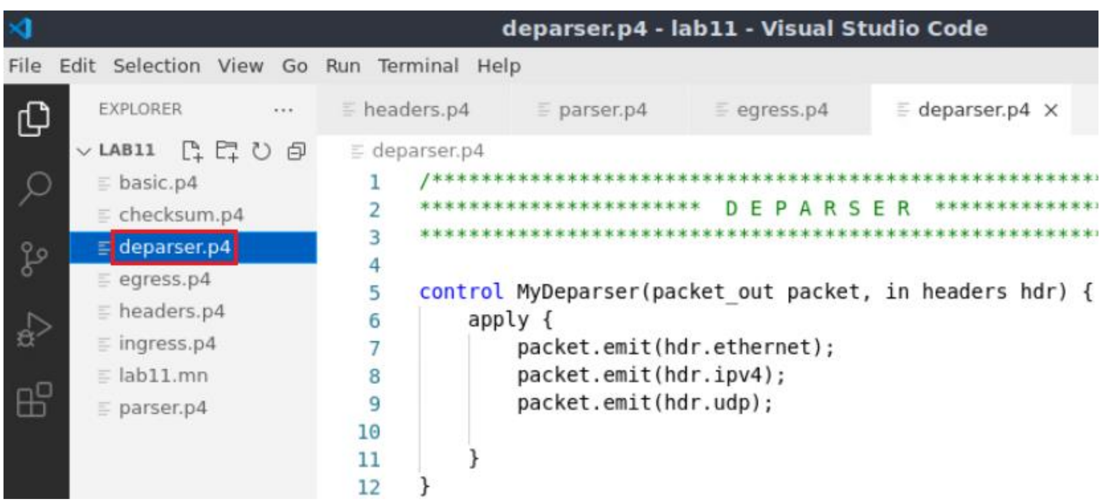


Figure 26. Opening the deparser processing block.

You will observe that the Ethernet and IPv4 header are already deparsed.

Step 2. Add the following lines of code to emit the custom headers.

```
packet.emit(hdr.layers);
packet.emit(hdr.sw_stats);
```

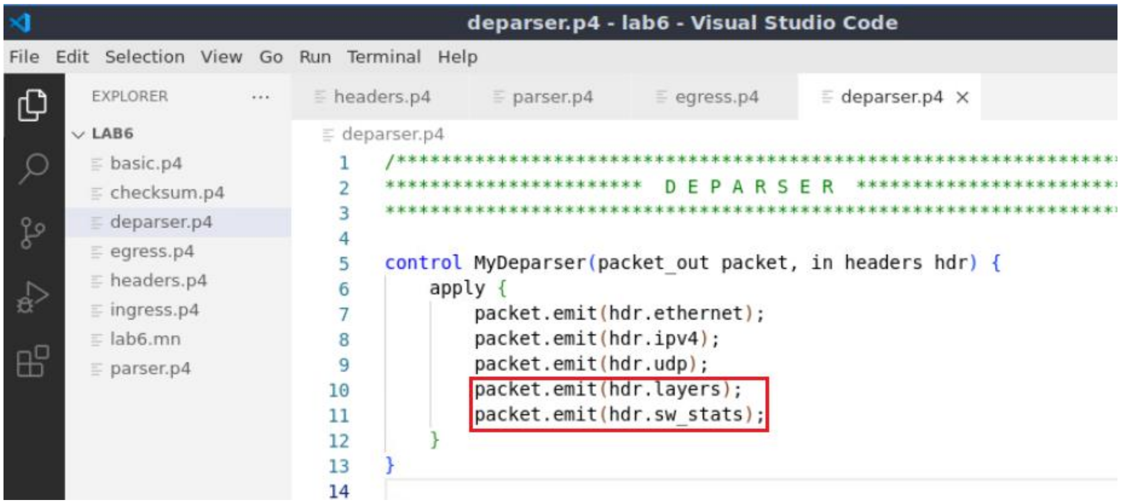


Figure 27. Emitting a custom header.

Note that the custom headers `layers` and `sw_stats` will only be emitted if they are valid.

Step 3. Save the changes to the file by pressing `Ctrl + s`.

5 Loading the P4 program

In this section, you will compile and load the P4 binary into the switches. You will also verify that the binaries reside in switches' filesystem.

5.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside VS Code to compile the program.

```
p4c basic.p4
```

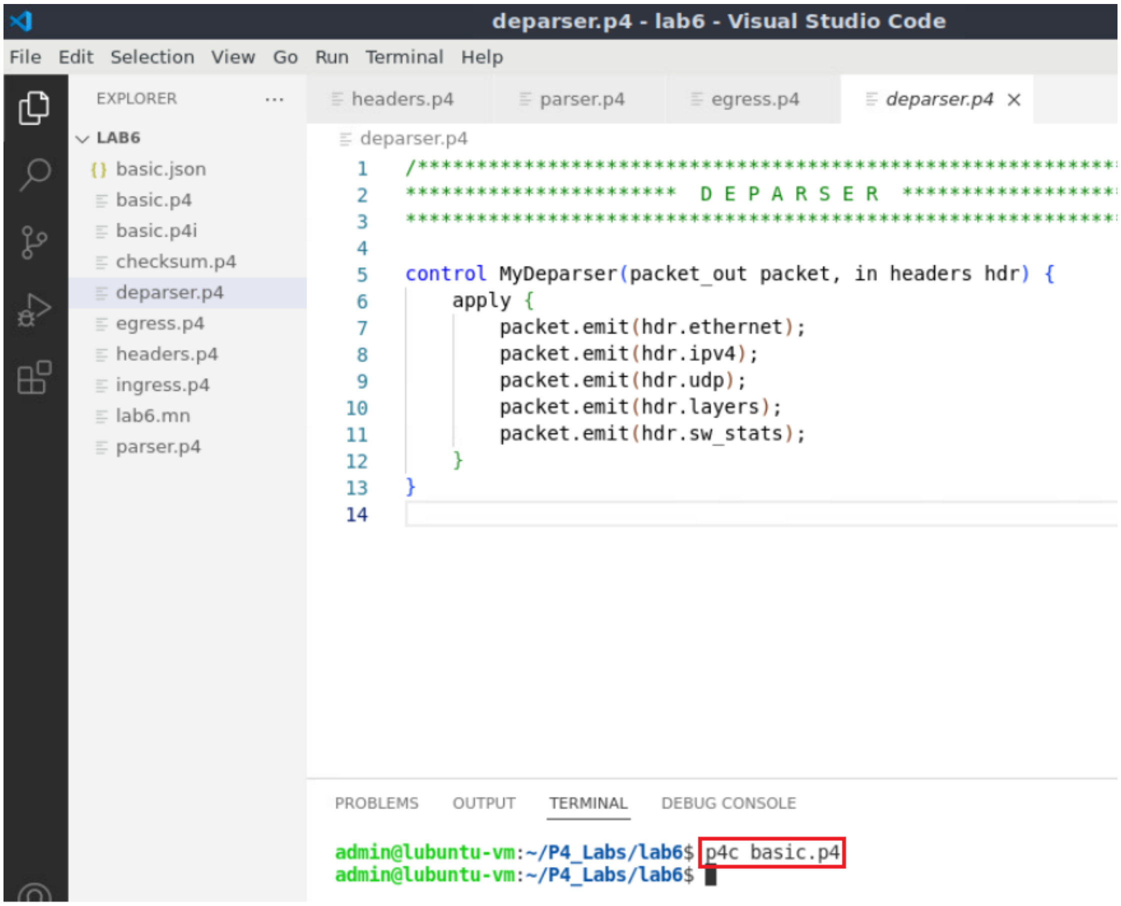


Figure 28. Compiling a P4 program.

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
```

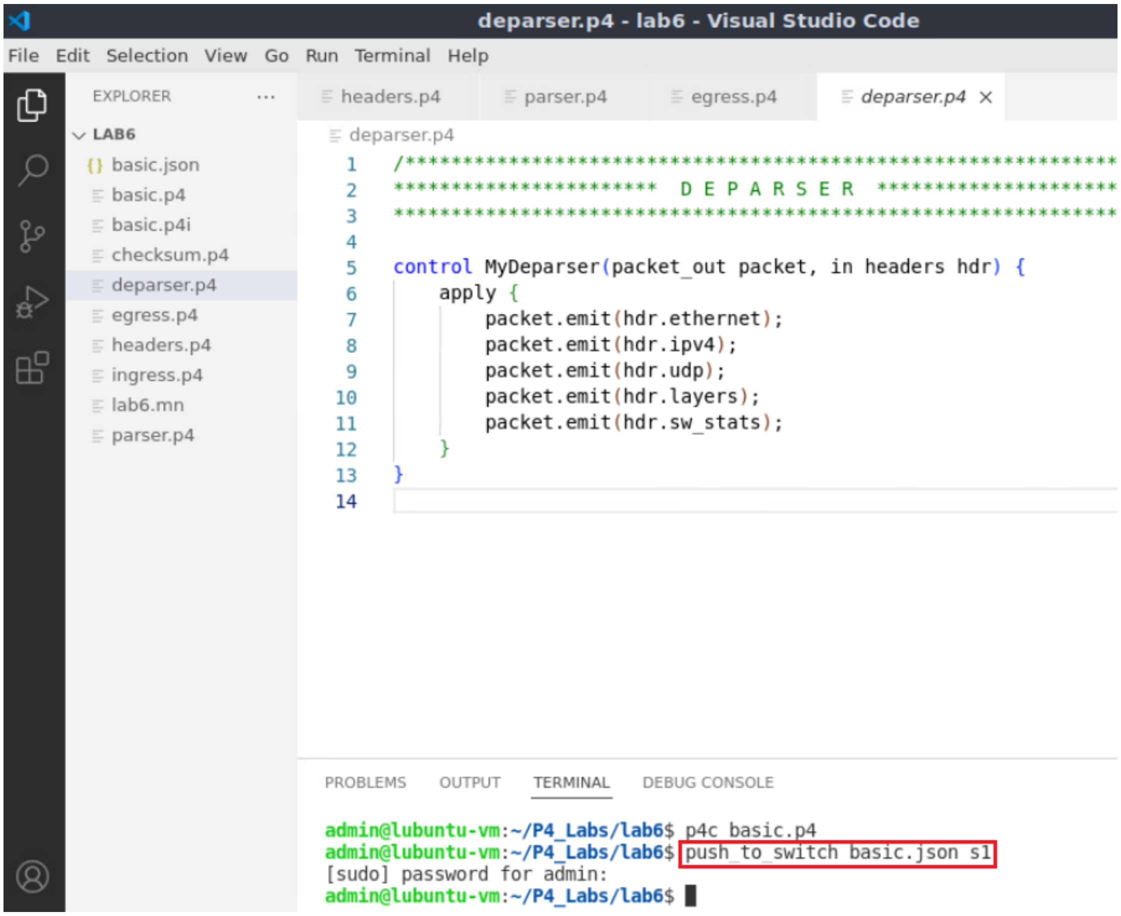


Figure 29. Pushing the *basic.json* file to switch s1.

Step 3. Similarly, type the command below in the terminal panel to push the *basic.json* file into switches s2 and s3 filesystems. Note that the same P4 program is used by the three switches.

```
push_to_switch basic.json s2
push_to_switch basic.json s3
```

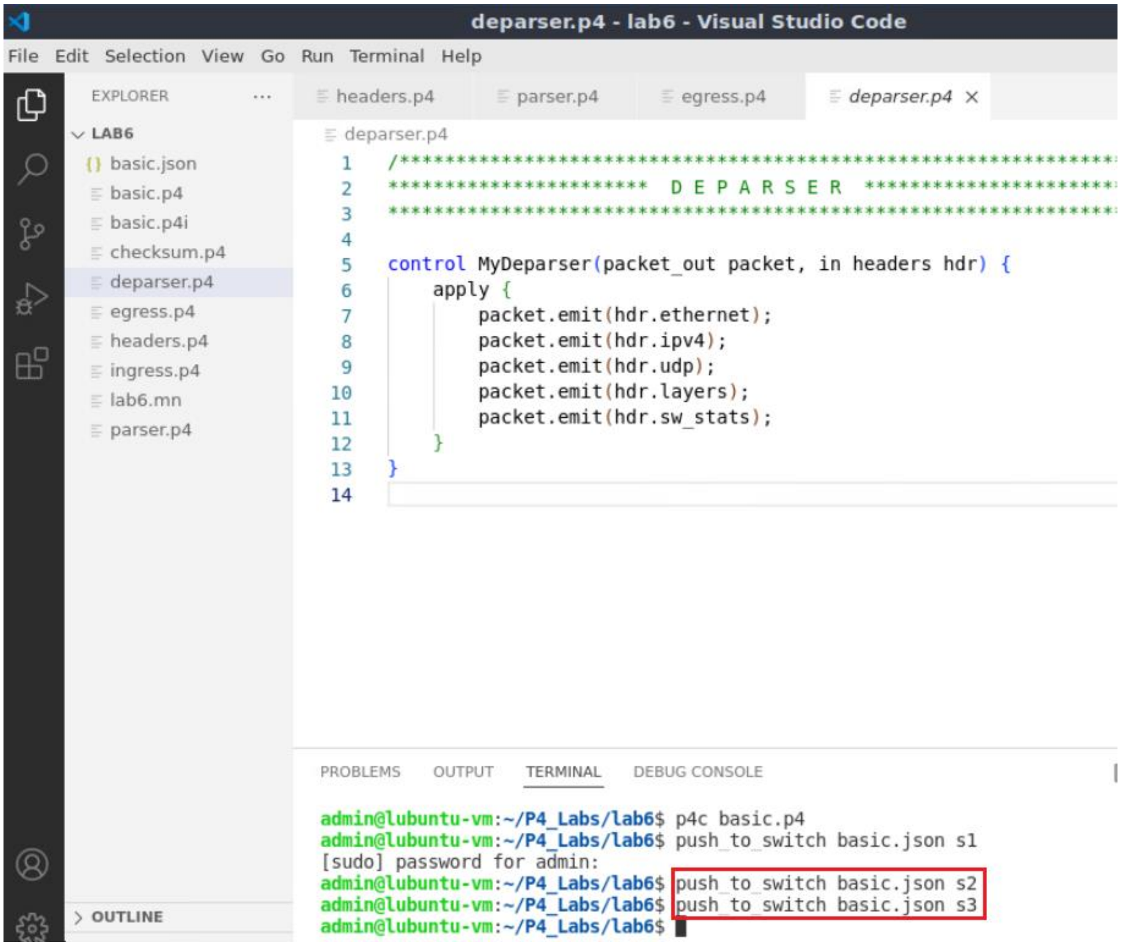


Figure 30. Pushing the *basic.json* file to switches s2 and s3.

5.2 Verifying the configuration

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

Figure 31. Maximizing the MiniEdit window.

Step 2. Right-click on the switch s1 icon and select *Terminal*.

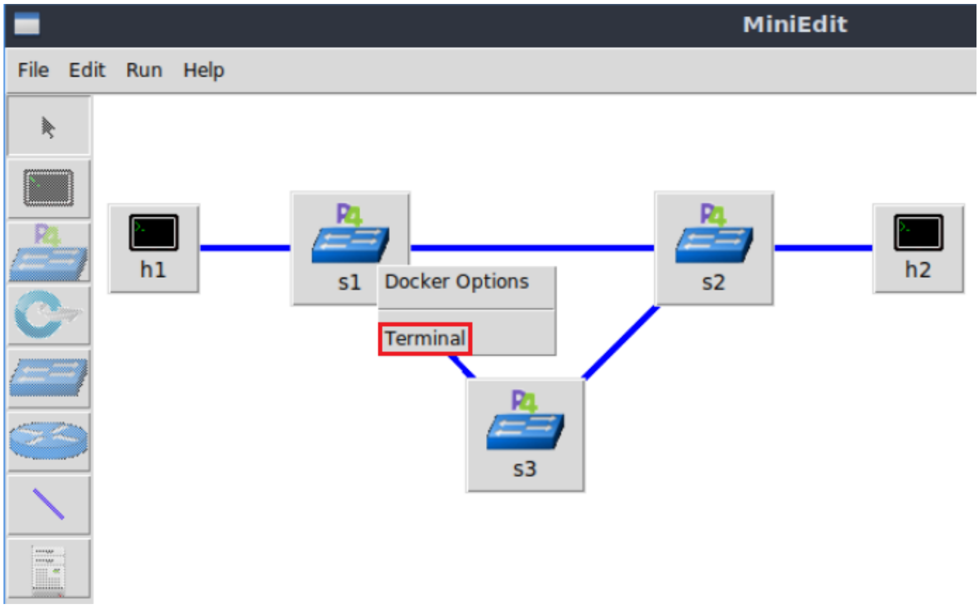


Figure 32. Starting the terminal on switch s1.

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 terminal.

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

```
ls
```

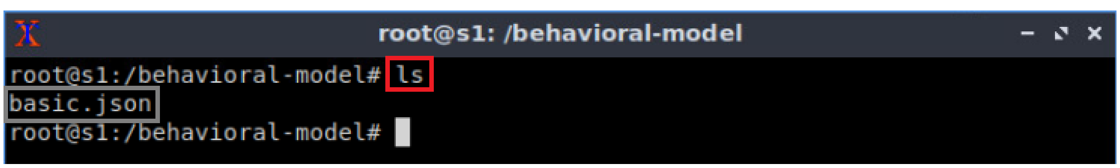


Figure 33. Displaying the contents of the current directory in the switch s1.

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

Step 4. Similarly, repeat steps 2 and 3 in switches s2 and s3, and verify that the *basic.json* file is present.

6 Configuring the switches

In this section, you will observe and understand the purpose of the interfaces available in the switches. You will map those interfaces to the ports in the P4 program and start the switch daemon. Then, you will load the rules to populate the match action tables.

6.1 Running the switch's daemon and mapping the ports

Step 1. In switch s1 terminal, start the switch daemon and map the logical interfaces to Linux interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```

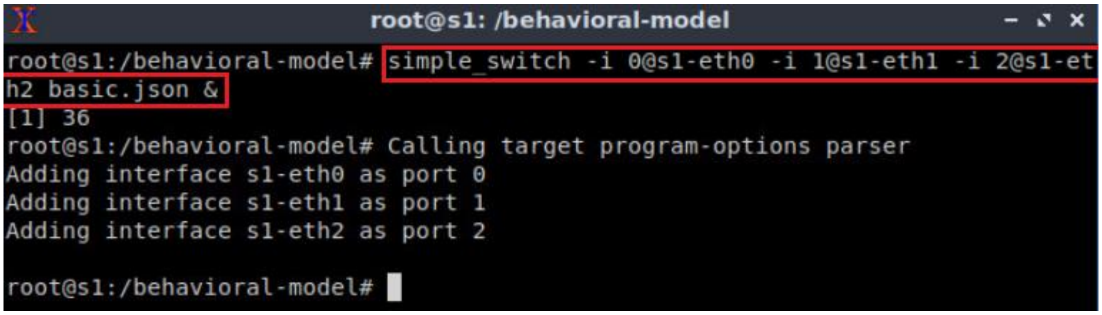


Figure 34. Starting switch s1 daemon and mapping the logical interfaces to Linux interfaces.

Step 2. In switch s2 terminal, start the switch daemon and map the logical interfaces to Linux interfaces by typing the following command.

```
simple_switch -i 0@s2-eth0 -i 1@s2-eth1 -i 2@s2-eth2 basic.json &
```

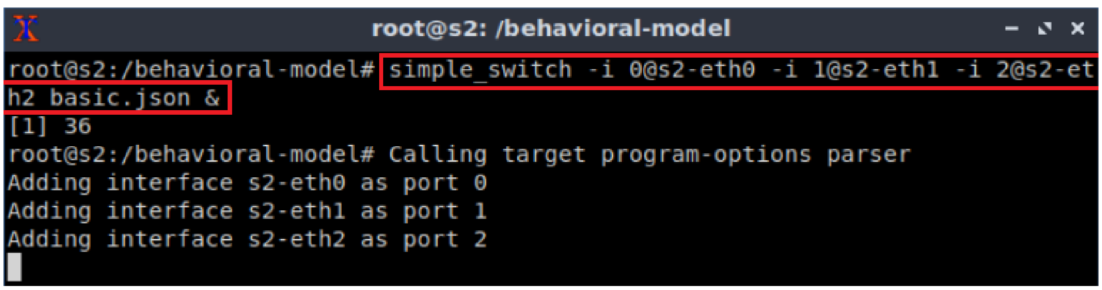


Figure 35. Starting switch s2 daemon and mapping the logical interfaces to Linux interfaces.

Step 3. In switch s3 terminal, start the switch daemon and map the logical interfaces to Linux interfaces by typing the following command.

```
simple_switch -i 0@s3-eth0 -i 1@s3-eth1 basic.json &
```

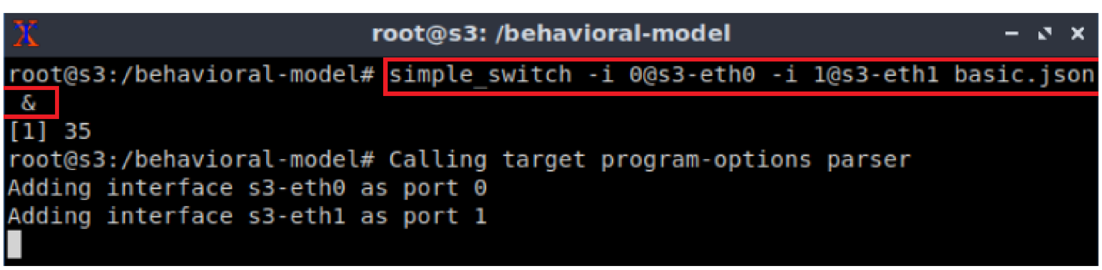


Figure 36. Starting switch s3 daemon and mapping the logical interfaces to Linux interfaces.

6.2 Loading the rules to the switch

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

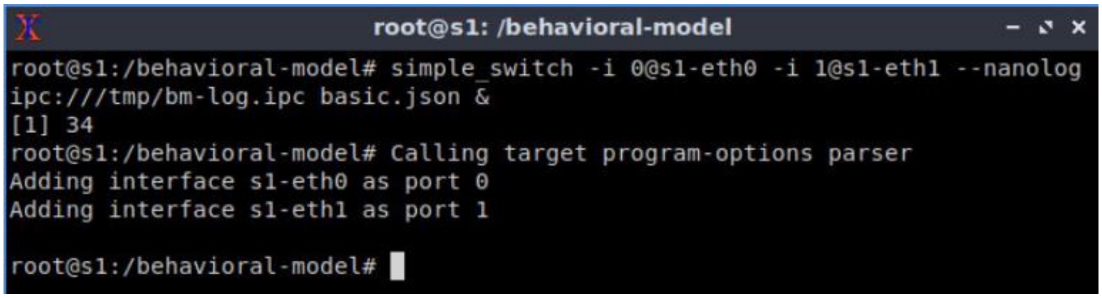


Figure 37. Returning to switch s1 CLI.

Step 2. Push the table entries to switch s1 by typing the following command.

```
simple_switch_CLI < ~/lab6/rules_s1.cmd
```

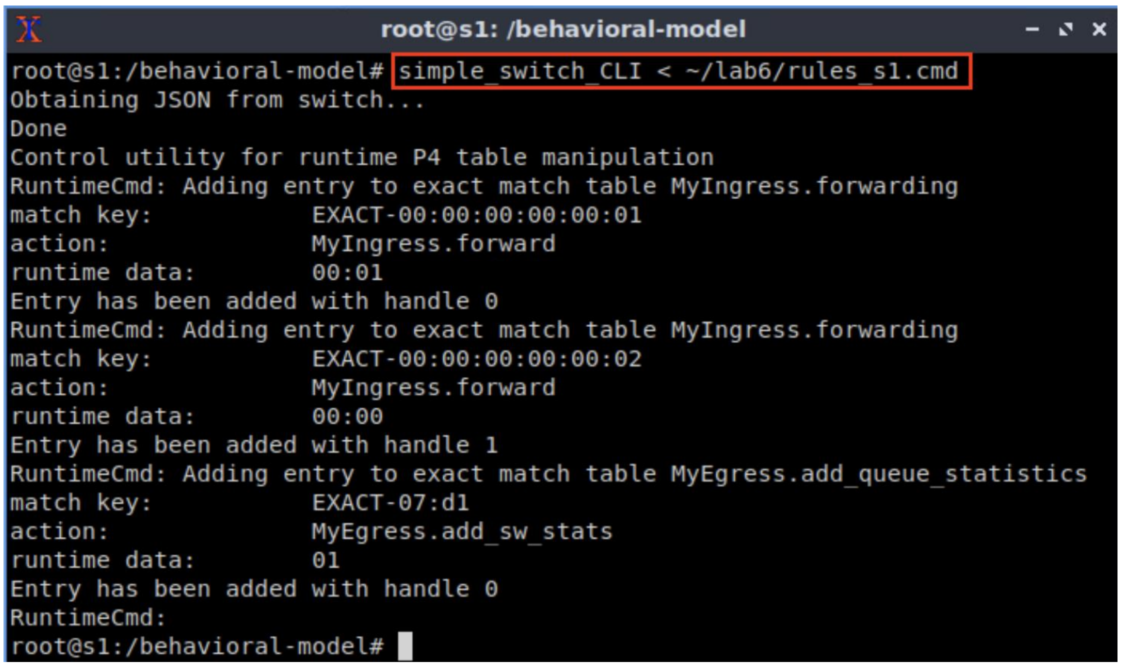


Figure 38. Populating the forwarding table into switch s1.

Step 3. Press *Enter,* then push the table entries to switch s2 by typing the following command.

```
simple_switch_CLI < ~/lab6/rules_s2.cmd
```

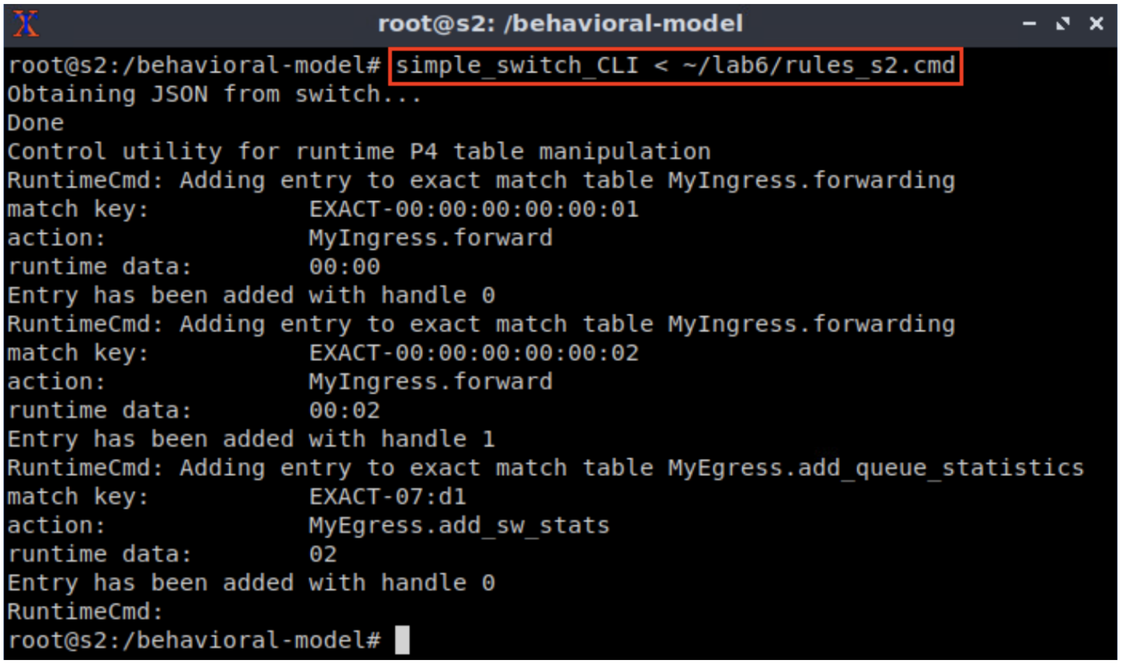


Figure 39. Populating the forwarding table into switch s2.

Step 4. Similarly, press *Enter,* then push the table entries to switch s3 by typing the following command.

```
simple_switch_CLI < ~/lab6/rules_s3.cmd
```

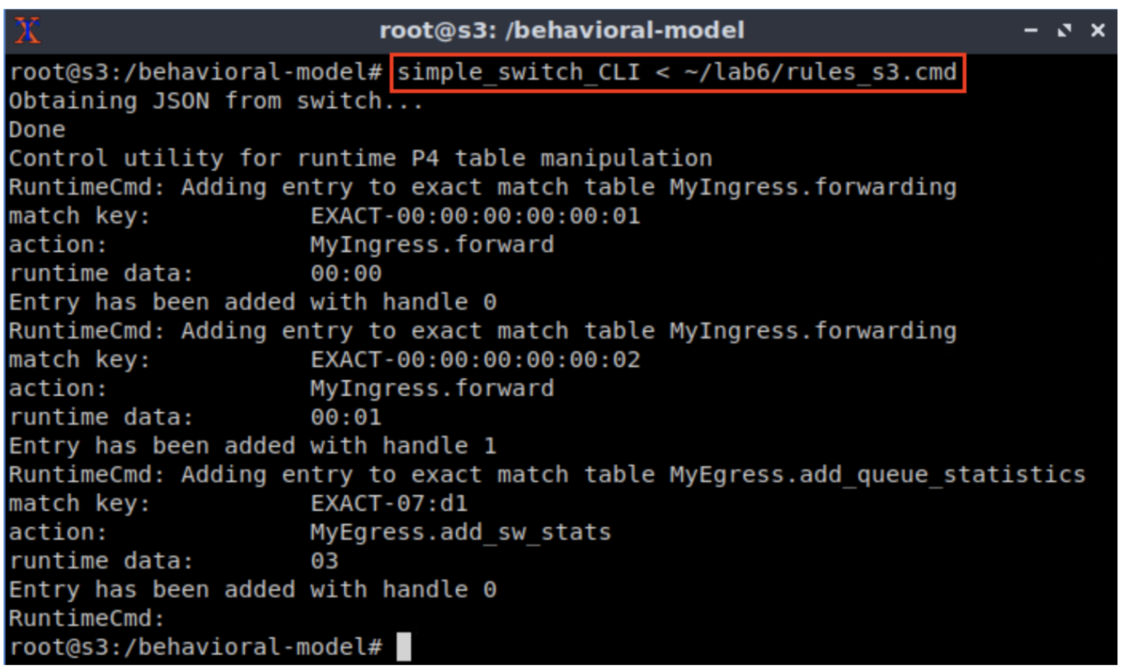


Figure 40. Populating the forwarding table into switch s3.

7 Testing and verifying the P4 program

In this section, you will set the queue rate and queue length of the three switches to send packets at 100 Mbps. Then, you will start sending probing packets to collect the queue statistics from the switches the packet traverses. Then, you will add background traffic by running an iperf3 test between host h1 and host h2 to observe the evolution of the queue length. Finally, you will change the forwarding rules in switch s1 to have the packet traversing three switches.

7.1 Setting the queue length

Step 1. Type the following command to start switch s1's CLI.

```
simple_switch_CLI
```

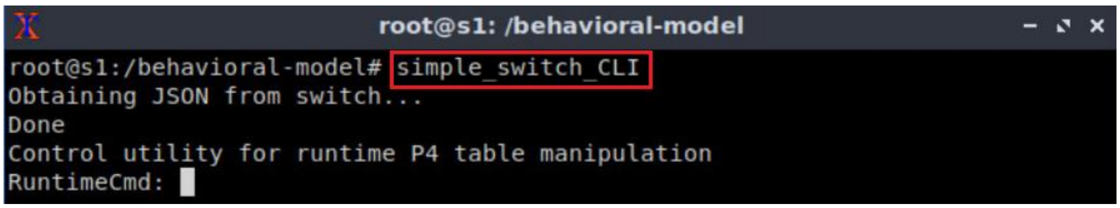


Figure 41. Starting switch s1's CLI.

Step 2. Set the queue rate by issuing the following command.

```
set_queue_rate 8333
```

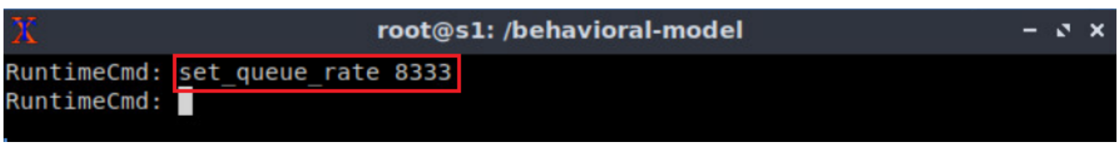


Figure 42. Setting the queue rate in switch s1.

Note that the queue rate value 8333 packets per second. This value is calculated as follows: Consider that we want to set the maximum rate to 100 Mbps and the Maximum Transmission Unit (MTU) is 1500 bytes/packet (i.e., 12,000 bits/packet). Thus, the number of packet that the queue must serve per second corresponds to the following value.

$$q_{rate} = \frac{Maximum\ rate\ [bits/s]}{Packet\ size\ [bits/packet]} = \frac{100{,}000{,}000\ [bits/s]}{12{,}000[bits/packet]} \approx 8333\ packets/s$$

Step 3. Set switch s1's buffer size (queue depth) by issuing the following command.

```
set_queue_depth 16666
```

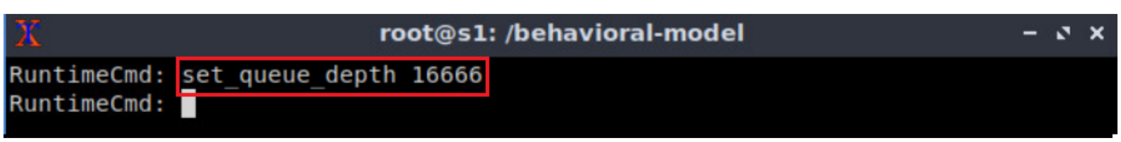


Figure 43. Setting the queue rate in switch s1.

In the figure above, the buffer size is set to 16666 packets (i.e., ~25Mbytes), which correspond to ten Bandwidth-Delay Product (BDP). The BDP value was calculated considering a bandwidth of 100Mbps and a maximum delay of 200ms.

$$BDP = BW * delay = 100{,}000{,}000 * 0.2 = 20{,}000{,}000\ [bits] = 2{,}500{,}000\ [bytes] \approx 1666\ [packets]$$

$$10 * BDP = 16{,}666\ [packets]$$

Step 4. Repeat from step 1 to step 3 in switches s2 and s3.

7.2 Testing the configuration

Step 1. Open a terminal in host h2 and start an iperf3 server by issuing the following command.

```
iperf3 -s
```

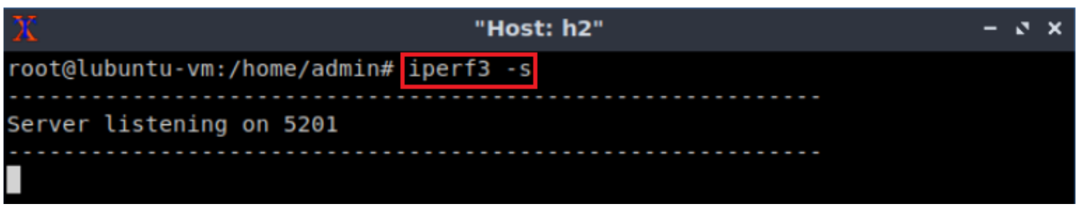


Figure 44. Starting an iperf3 server in host h2.

Step 2. In host h1, run the following command to start an iperf3 client that will send data to the iperf3 server in host h4.

```
iperf3 -c 10.0.0.2
```

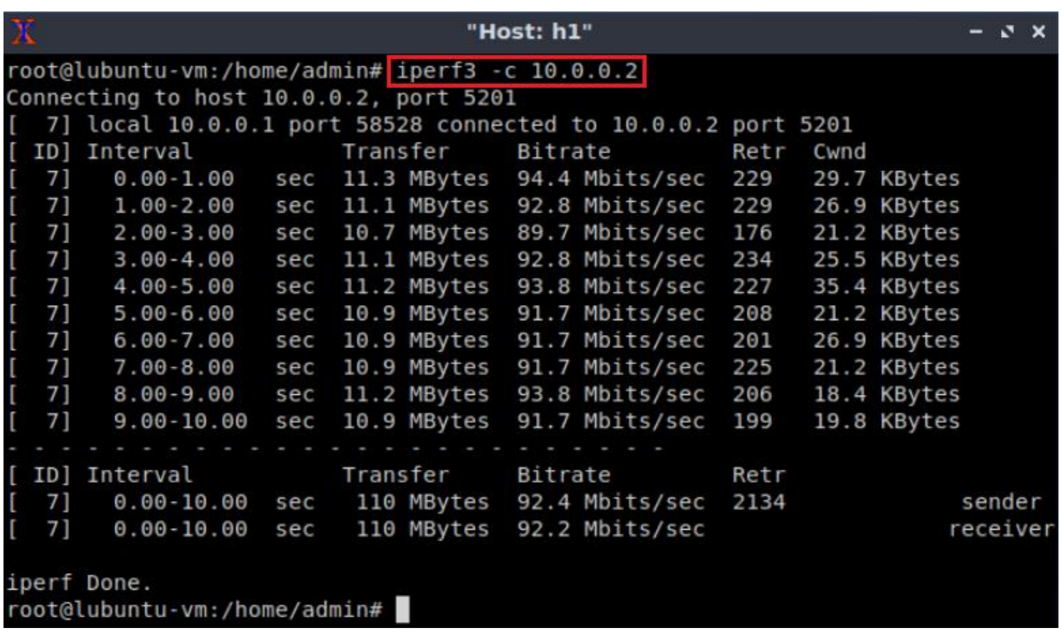


Figure 45. Starting an iperf3 client in host h1.

Note in the figure above that the bitrate of the data transfer is approximately 92.4Mbps which is close to the link bandwidth 100Mbps.

7.3 Starting the probing protocol

Step 1. Open another terminal in host h2 and type the command, so that, the host starts listening for the custom packets.

```
recv.py -p stack
```

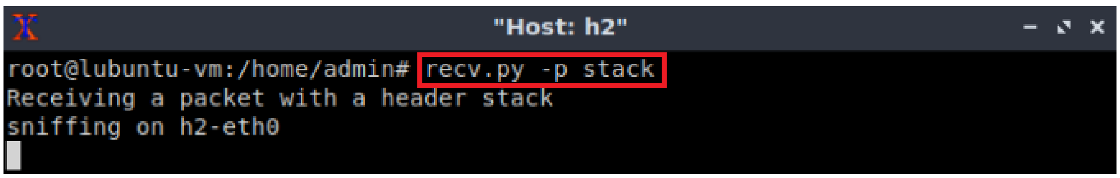


Figure 46. Listening for incoming packets in host h2.

The script above receives the following parameters:

- `-p`: enables listening to a specific protocol.
- `stack`: the protocol type.

Step 2. On host h1's terminal, type the following command.

```
send.py 10.0.0.2 HelloWorld -p stack
```

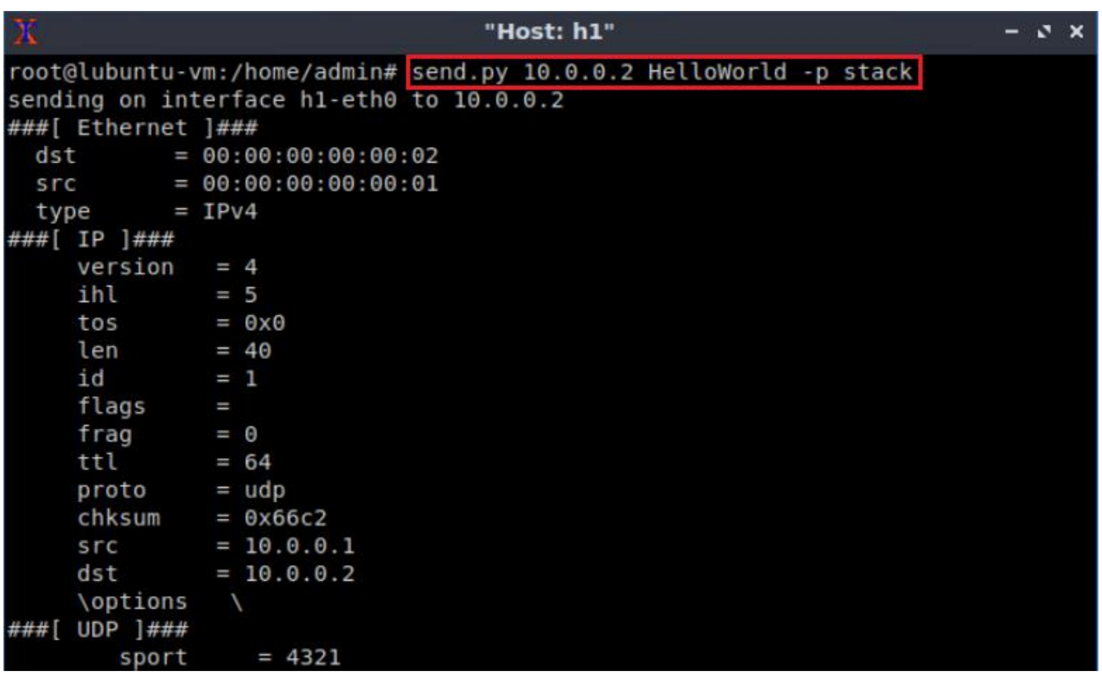


Figure 47. Sending a test packet from host h1 to host h2.

Similarly, the script above receives the following parameters:

- `10.0.0.2`: the destination IPv4 address.
- `HelloWorld`: the packet payload.
- `-p`: enables listening to a specific protocol.
- `stack`: the protocol type. Note that this protocol sends a custom packet every 10 milliseconds.

Step 3. Verify that the custom packet is being received on host h2.

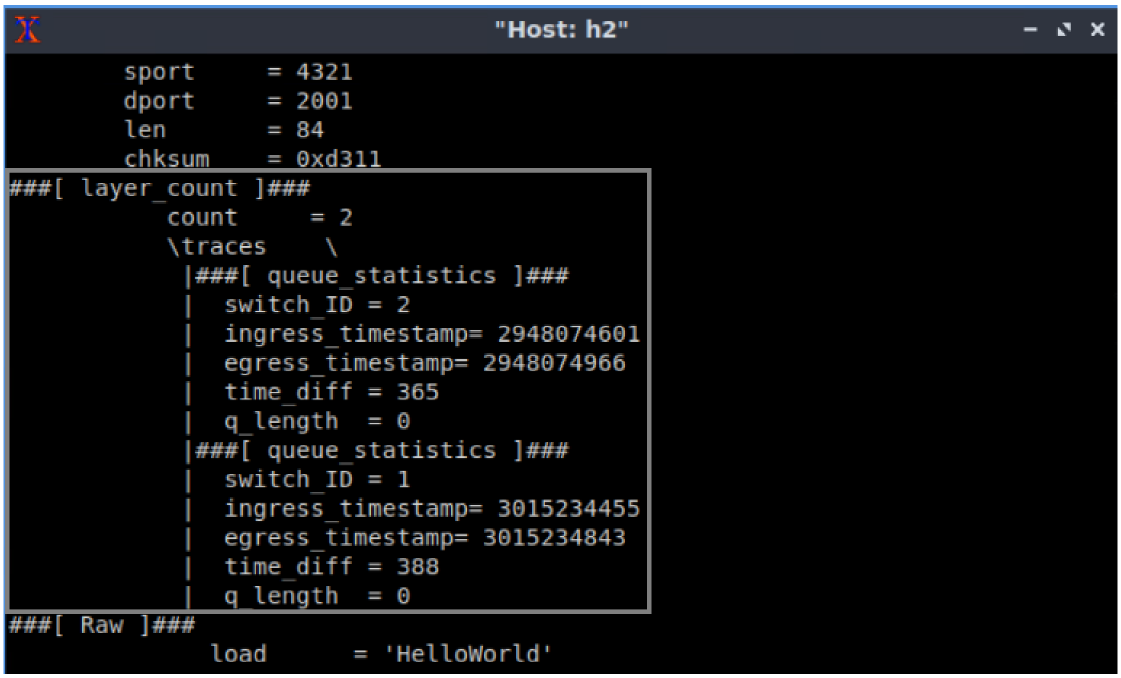


Figure 48. Packet received on host h2.

The figure above shows that the layer count and two headers with the queue statistics have been added over the UDP header. In this test, the layer count is 2, and the traces contain the queue statistics corresponding to switches 2 and 1, respectively. Note that the queue length is zero.

7.4 Measuring the queue length with background traffic

Step 1. Open another terminal in host h1 and run the following command.

```
iperf3 -c 10.0.0.2 -t 120
```

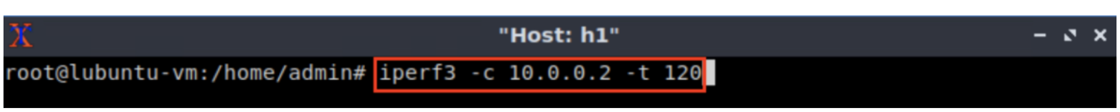


Figure 49. Starting an iperf3 client in host h1.

Step 2. Go back to host h2 and observe that two traces have been added. The first trace corresponds to the queue statistics of switch s2 and switch s1 respectively.

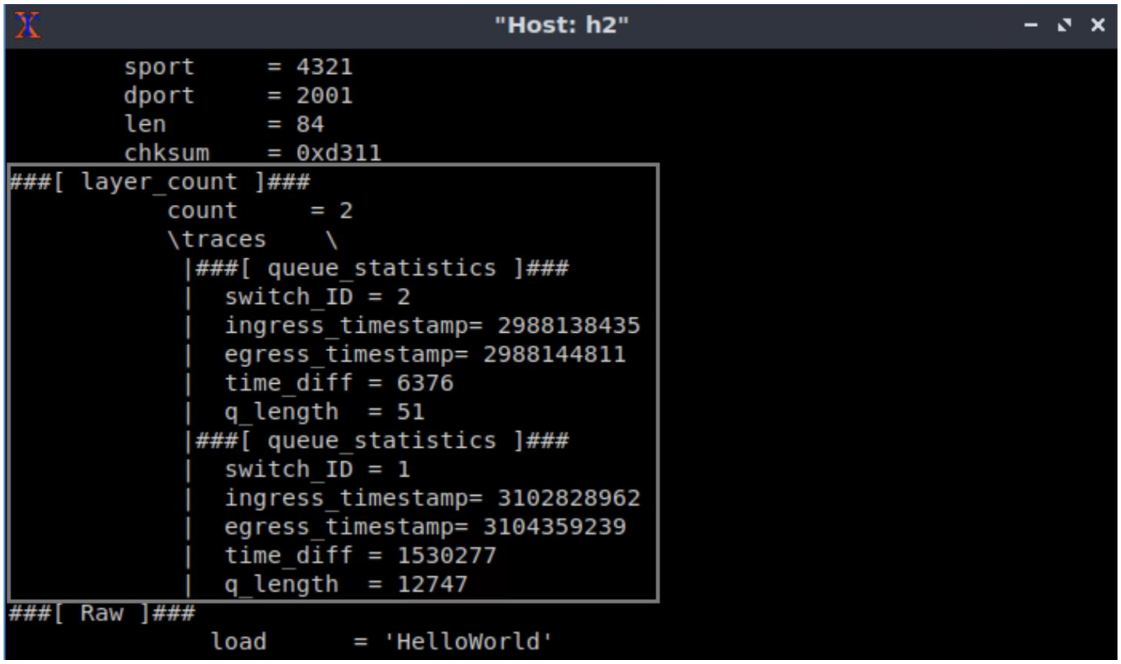


Figure 50. Visualizing the evolution of the processing time and queue length.

After adding background traffic by running an iperf3 test, you can observe that queues are formed in both switches.

7.5 Steering the traffic towards switch s3

Step 1. In switch s1 terminal, issue the following command to show the table entries. Note that the second entry (0x1) specifies that traffic going to host h2 (with MAC address 00:00:00:00:00:02) uses the egress port 0.

```
table_dump MyIngress.forwarding
```

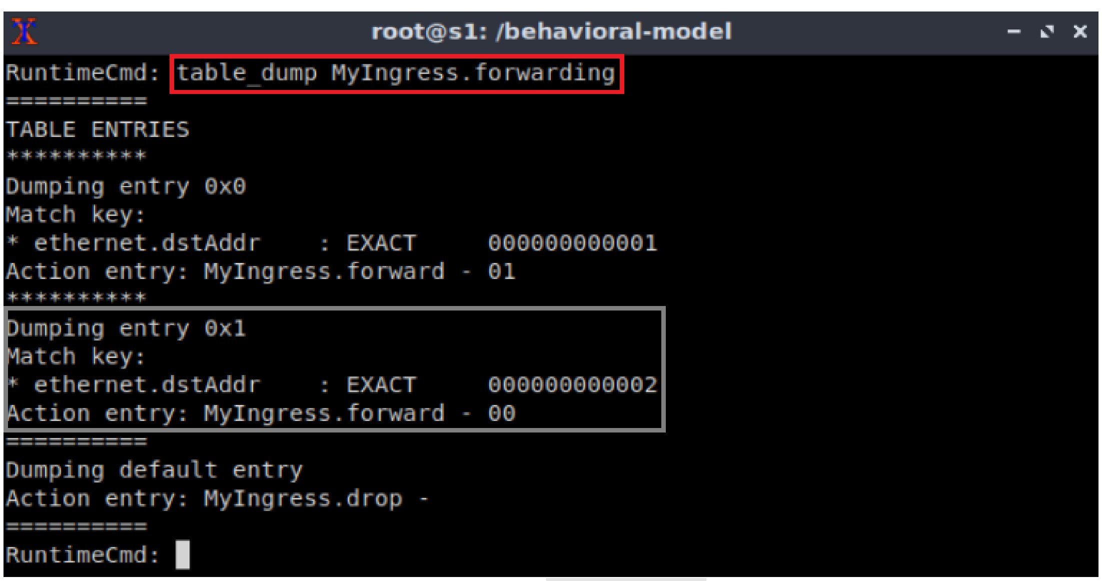


Figure 51. Displaying the entries of table `forwarding` located in the ingress block.

Step 2. Delete the second entry by issuing the following command.

```
table_delete MyIngress.forwarding 1
```

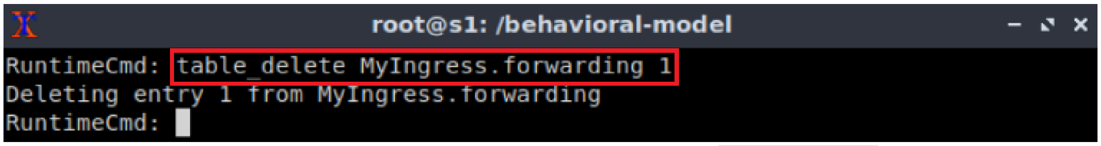


Figure 52. Deleting entry 1 from the table `forwarding`.

Step 3. Insert a new entry by issuing the command below. This entry will specify that egress port will be port 2, the one facing switch s3.

```
table_add MyIngress.forwarding MyIngress.forward 00:00:00:00:00:02 => 2
```

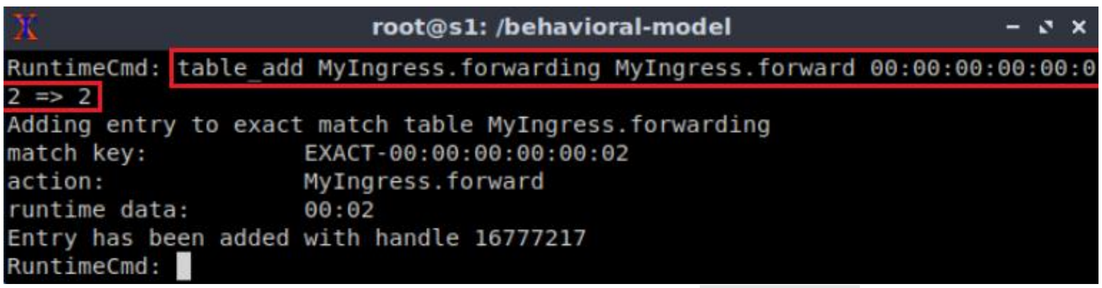


Figure 53. Adding a new entry to table `forwarding`.

Step 4. Go back to host h1 and stop the iperf3 test, if this is still running, by pressing `Ctrl+c`.

Step 5. Go back to host h1 and run another iperf3 test by issuing following command.

```
iperf3 -c 10.0.0.2 -t 120
```

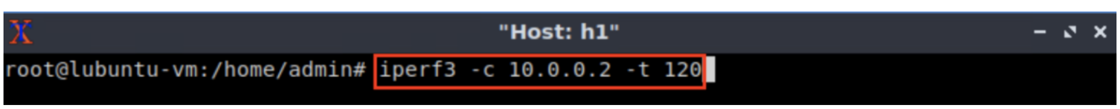


Figure 54. Starting an iperf3 client in host h1.

Step 6. In host h2 verify that three switch traces are being received.

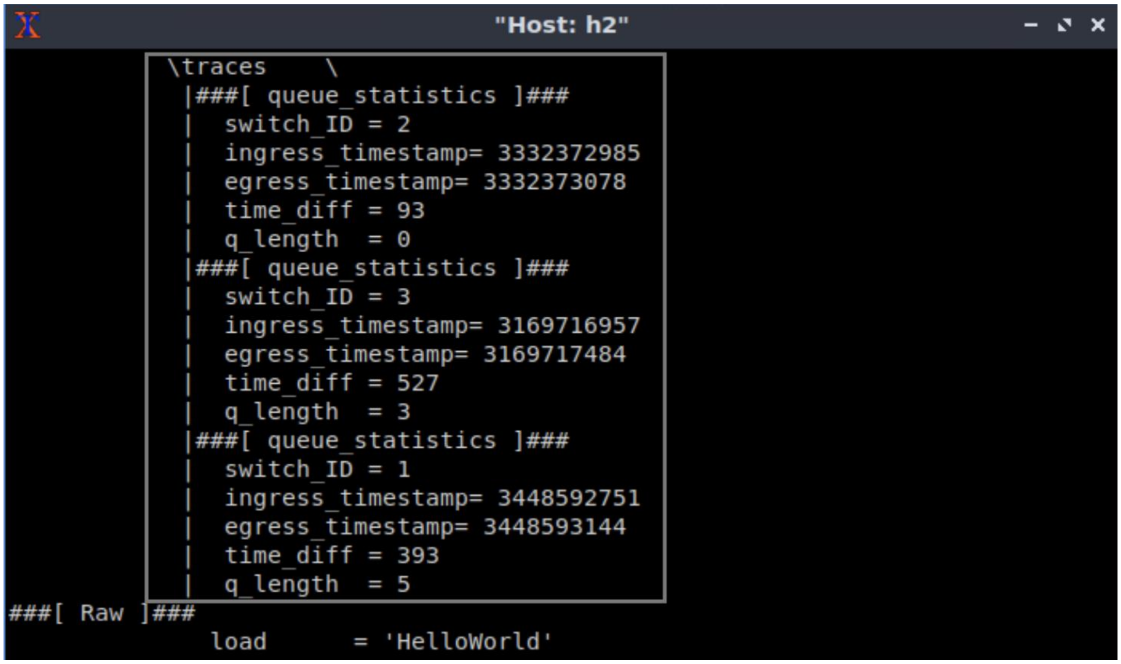


Figure 55. Starting an iperf3 client in host h2.

The figure above shows that the layer count and three headers with the queue statistics have been added over the UDP header. In this case, the layer count is 3, and the traces contain the queue statistics corresponding to switches 3, 2 and 1, respectively.

This concludes lab 6. 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.

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 7: Measuring Flow Statistics using Direct and Indirect Counters

Document Version: 04-24-2022

Contents

Overview

Counters are stateful elements to collect statistics in the data plane. This lab introduces the user to counters in P4 by showing how to create a monitoring application that counts the bytes per flow. Moreover, this lab demonstrates how to read the counter value from the control plane. Finally, the user will generate traffic to test the P4 program.

Objectives

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

1. Define direct and indirect counters.
2. Refer indirect counters to match-action table entries.
3. Read counter values from the control plane.

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: Implementing counters at the ingress pipeline.
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 counters in P4

Counters are stateful elements used for monitoring tasks, such as collecting statistics from flows, enforcing Quality of Service (QoS) policies, and implementing security features (e.g., detecting and blocking Denial of Service (DoS) attacks). The V1Model[1] provides counters as extern objects that can be invoked using the P4 language. A P4 program can update counters but cannot read them. The control plane can read counter values and use them to implement applications. Counters in P4 support packet counters, byte counters, and the combination of both. In P4, there are two types of counters[2]:

- Direct counters: which are directly associated to a match-action table.
- Indirect counter: independent counters that can be referred to specific entries or group of entries in a match-action table.

1.1 Lab scenario

Consider the topology in Figure 2. This topology comprises 8 hosts and two switches, a P4 switch (i.e., s1) and a legacy switch (i.e., s2). Switch s1 connects to host h1 via *port 1*, host h2 via *port 2*, etc. Note that switch s1 is linked to switch s2 via *port 0*. The lab scenario consists of creating a P4 program to monitor the flows traversing switch s1. The P4 program counts the number of bytes per flow going to a destination host using direct and indirect counters. Then, the user will load and test the P4 program by initiating data transfers from any pair of hosts. Finally, the user will verify the count values from the control plane.

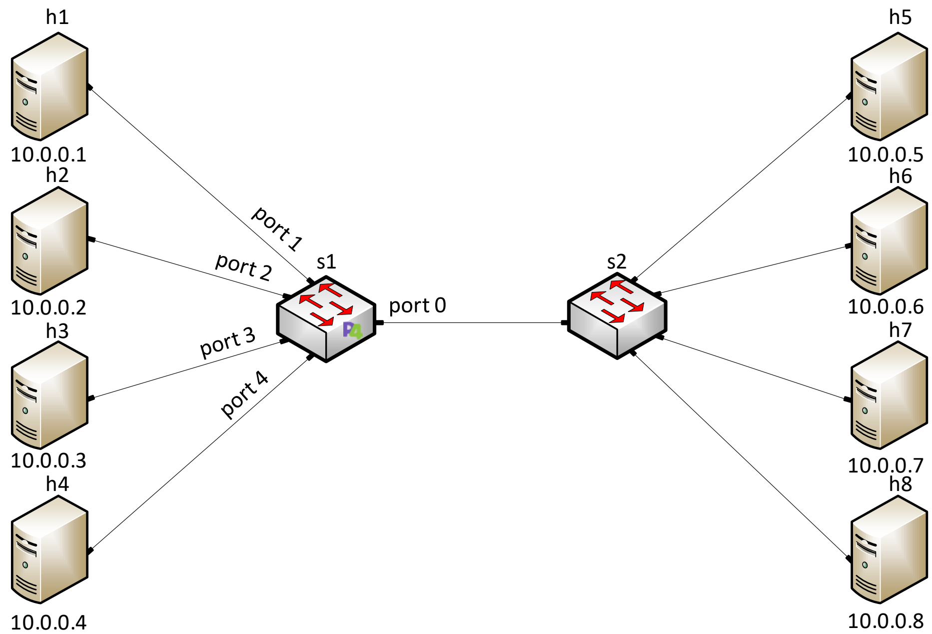


Figure 1. Example topology. This topology comprises 8 hosts, a P4 switch, and a legacy switch.

1.2 Defining a direct counter in P4

Figure 2 show an example P4 code that uses a direct counter to count how many times a rule in a match-action table was hit. The code describes the behavior of the ingress pipeline, which has implemented the table `forwarding` (see line 14). This table matches on the destination IPv4 address (see line 15-17), and has the following actions: `forward`, `drop`, and `NoAction` (see lines 18-22). The size of the table is 32 entries (see line 23), and the default action is `drop` (see line 24). Note that the direct counter `my_direct_counter` defined in line 4, is invoked in the table forwarding (see line 25). The forwarding logic is specified in the `apply` block, which invokes the table `forwarding` when the switch receives a valid IPv4 packet.

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

direct_counter(counterType.packets) my_direct_counter;

action forward(egressSpect_t port){
      standard_meadata.egress_spec = port;
}

action drop(){
      mark_to_drop(standard_metadata);
}

table forwarding {
      key = {
          hdr.ipv4.dstAddr : exact;
       }
      actions = {
          forward;
          drop;
          NoAction;
      }
      size = 32;
      default_action = drop();
      counters = my_direct_counter;
}
apply {
      if(hdr.ipv4.isValid()){
         forwarding.apply();
      }
}
```

Figure 2. Defining a direct counter in the ingress block.

Figure 3 shows the table `forwarding` described in the P4 program in Figure 2. Notice that each entry in the table refers to an index in the counter `my_direct_counter`. The number of indexes in the counter `my_direct_counter` is defined by the size of the table, which in this case is 32 entries. Each count value in the counter `my_direct_counter` is accessed with an index. In this P4 program, the value of each count indicates how many packets matched on an entry independently of the executed action. The count values are in packets and bytes. For example, notice that the last entry in the table `forwarding` (i.e., key = 10.0.0.32) counted 49 packets or 73,500 bytes assuming that each packet has 1500 bytes.

Note that the direct counter `my_direct_counter` assigns an index `Idx` to each entry in the table `forwarding`, meaning that the table can have up to 32 independent counters.

forwarding			my_direct_counter		
Key	Action	Action Data	Idx.	Count	
				Packets	Bytes
10.0.0.1	forward	egress port = 1	0	0	0
10.0.0.2	forward	egress port = 2	1	71	106,500
10.0.0.3	forward	egress port = 3	2	23	34,500
10.0.0.4	forward	egress port = 4	3	52	78,000
10.0.0.5	forward	egress port = 0	4	84	126,000
10.0.0.6	forward	egress port = 0	5	11	16,500
10.0.0.7	forward	egress port = 0	6	0	0
10.0.0.8	forward	egress port = 0	7	37	55,500
⋮	⋮	⋮	⋮	⋮	⋮
10.0.0.32	drop	egress port = 0	31	49	73,500

Figure 3. Representation of a match-action table referred to a direct counter. The counter `my_direct_counter` has an index associated with each entry in the table `forwarding`.

1.3 Defining an indirect counter in P4

Indirect counters in P4 are invoked by the extern `counter`. An example of an indirect counter is presented in Figure 4. The indirect counter `my_indirect_counter` defined in line 4 is a packet counter with three indexes. This counter increases the count value at index `index` when the forward action is invoked (see line 8). Note that the control plane provides the index value to the action `forward` as part of the action data (see line 6).

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

counter(3,counterType.packets) my_indirect_counter;

action forward(egressSpect_t port, bit<32> index){
      standard_meadata.egress_spec = port;
      my_indirect_counter.count(index);
}
action drop(){
      mark_to_drop(standard_metadata);
}

table forwarding {
      key = {
          hdr.ipv4.dstAddr : exact;
       }
      actions = {
          forward;
          drop;
          NoAction;
      }
      size = 32;
      default_action = drop();
}

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

Figure 4. Defining an indirect counter in the ingress block.

In contrast to the direct counter, the indirect counter `my_indirect_counter` is invoked inside the action `forward`. Note that the indirect counter can be invoked also in other actions such as `drop` and in the `apply` block. Another difference is that the programmer specifies the number of indexes of an indirect counter.

Figure 5 shows that the first three entries in the table `forwarding` are associated with index 0 in the counter `my_indirect_counter`. Index 1 is associated to the 4th, 5th, and 7th entries. Similarly, index 2 is associated with the 6th, 8th, and 32nd entries.

forwarding

Key	Action	Action Data
10.0.0.1	forward	egress port = 1, Idx = 0
10.0.0.2	forward	egress port = 2, Idx = 0
10.0.0.3	forward	egress port = 3, Idx = 0
10.0.0.4	forward	egress port = 4, Idx = 1
10.0.0.5	forward	egress port = 0, Idx = 1
10.0.0.6	forward	egress port = 0, Idx = 2
10.0.0.7	forward	egress port = 0, Idx = 1
10.0.0.8	forward	egress port = 0, Idx = 2
⋮	⋮	⋮
10.0.0.32	drop	egress port = 0, Idx = 2

my_indirect_counter

Idx.	Count	
	Packets	Bytes
0	23	34,500
1	17	25,500
2	42	63,000

Figure 5. Representation of a match-action table referred to a direct counter. The counter `my_indirect_counter` has indexes associated with some entries in the table `forwarding`. Note that a counter can have associated multiple entries.

2 Lab topology

Let us get started by loading a simple Mininet topology using MiniEdit. The topology comprises 8 end hosts, a P4 programmable switch (i.e., s1), and a legacy switch (i.e., s2).

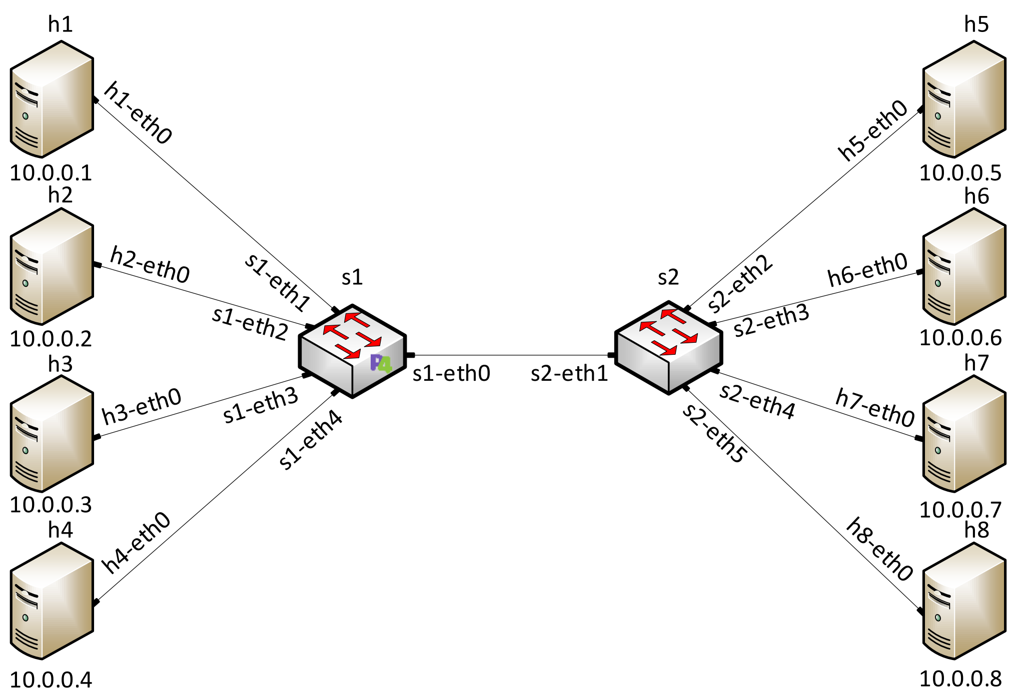


Figure 6. Lab topology.

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`.

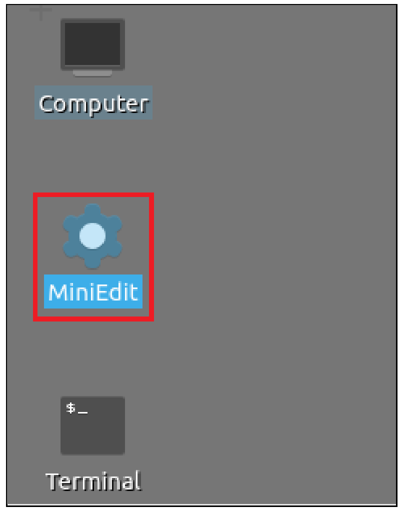


Figure 7. MiniEdit shortcut.

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

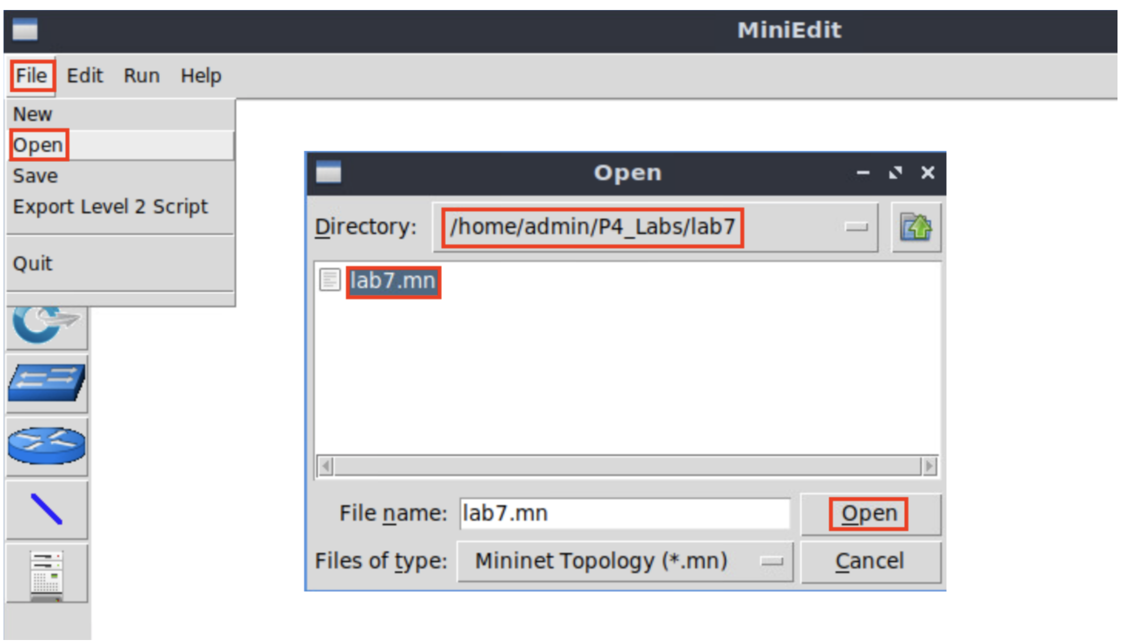


Figure 8. MiniEdit's *Open* dialog.

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 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.

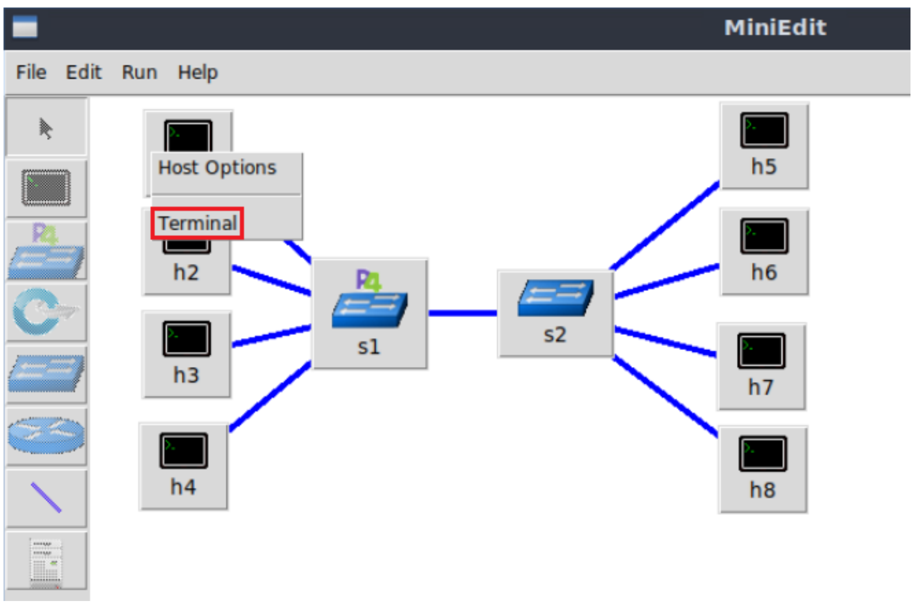


Figure 10. Opening a terminal on host h1.

Step 2. Run the following command to display interfaces' information on host h1.

```
ifconfig
```

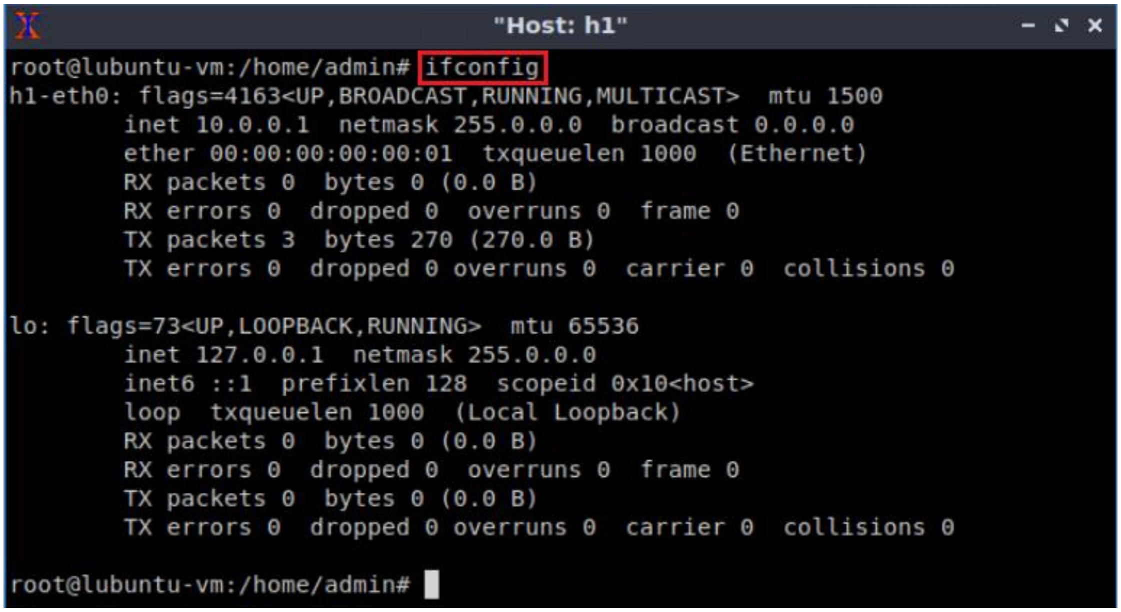


Figure 11. Displaying interfaces' information on host h1.

3 Implementing counters at the ingress pipeline

In this section, you will load the programming environment and define a match-action table that matches the destination IPv4 address to forwarding packets. This table will have a direct counter that will increase its count every time a packet matches a rule. Then you will define indirect counters. These counters will only count when a packet hits a specific entry in the match-action table.

3.1 Loading the programming environment

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

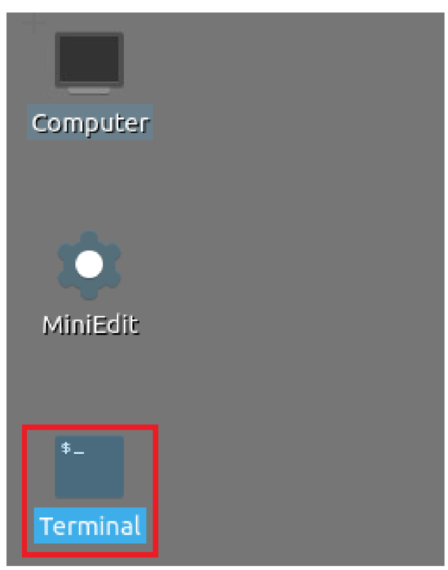


Figure 12. Shortcut to open a Linux terminal.

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 execute.

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
```

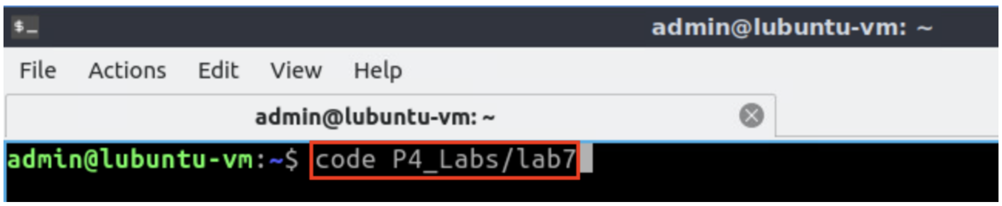


Figure 13. Loading the development environment.

3.2 Defining the forwarding behavior

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. You will observe that the block has no match-action table implemented.

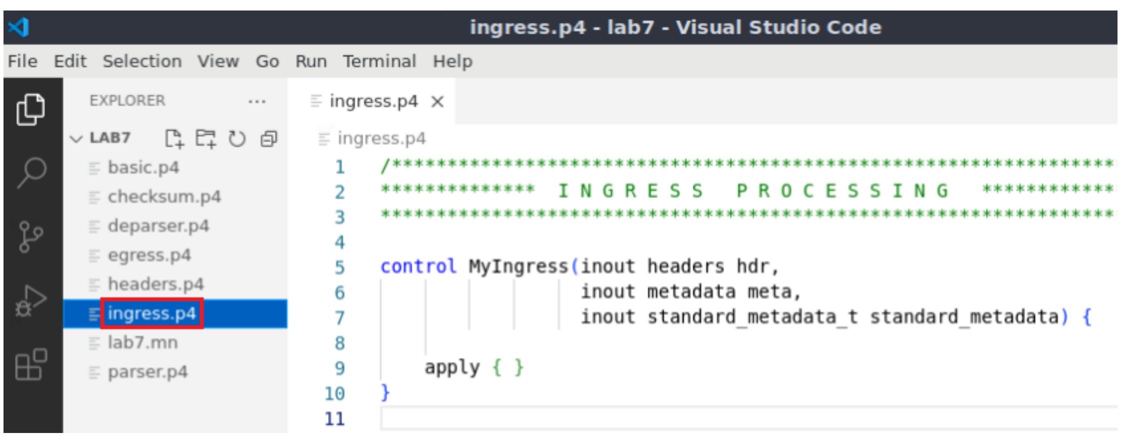


Figure 14. Inspecting the egress processing block.

Step 2. Define the `drop` action by adding the following code. This action is invoked to drop packets.

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

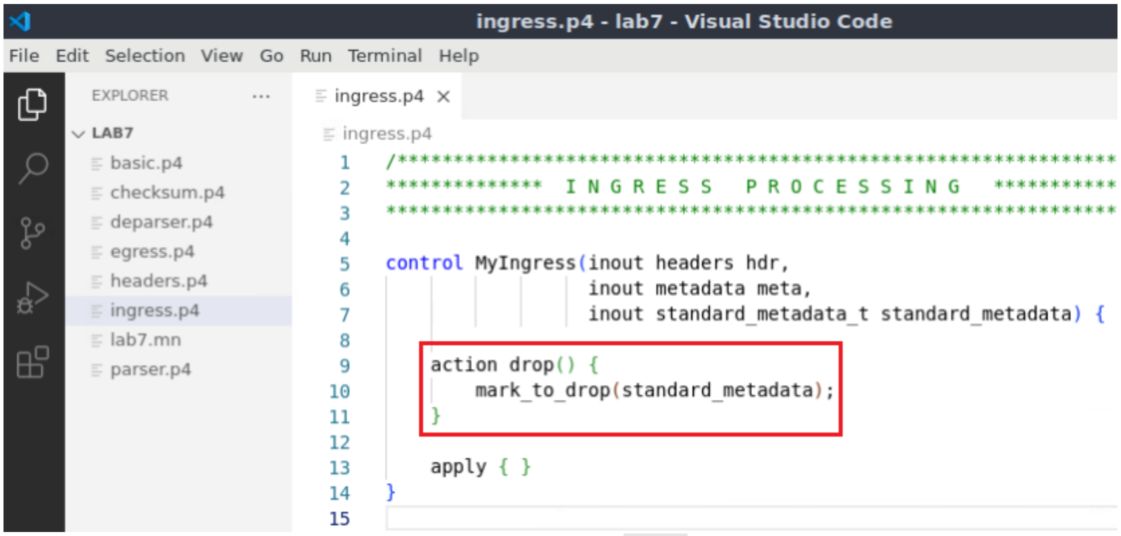


Figure 15. Defining the `drop` action.

Step 3. Define the `forward` action by adding the code shown below. This action forwards packets through an egress port specified by the control plane.

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

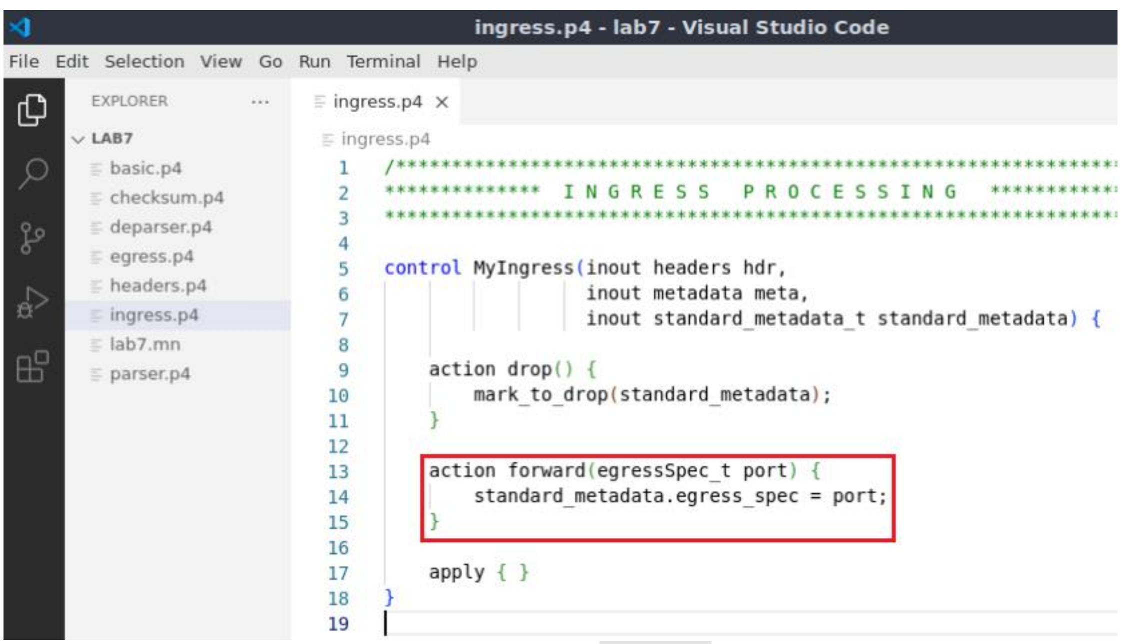


Figure 16. Defining the `forward` action.

Step 4. Define the table `forwarding` by adding the following piece of code.

```
table forwarding {
       key = {
              hdr.ipv4.dstAddr : exact;
       }
       actions = {
           forward;
           drop;
```

```
            NoAction;
        }
        size = 32;
        default_action = drop();

}
```

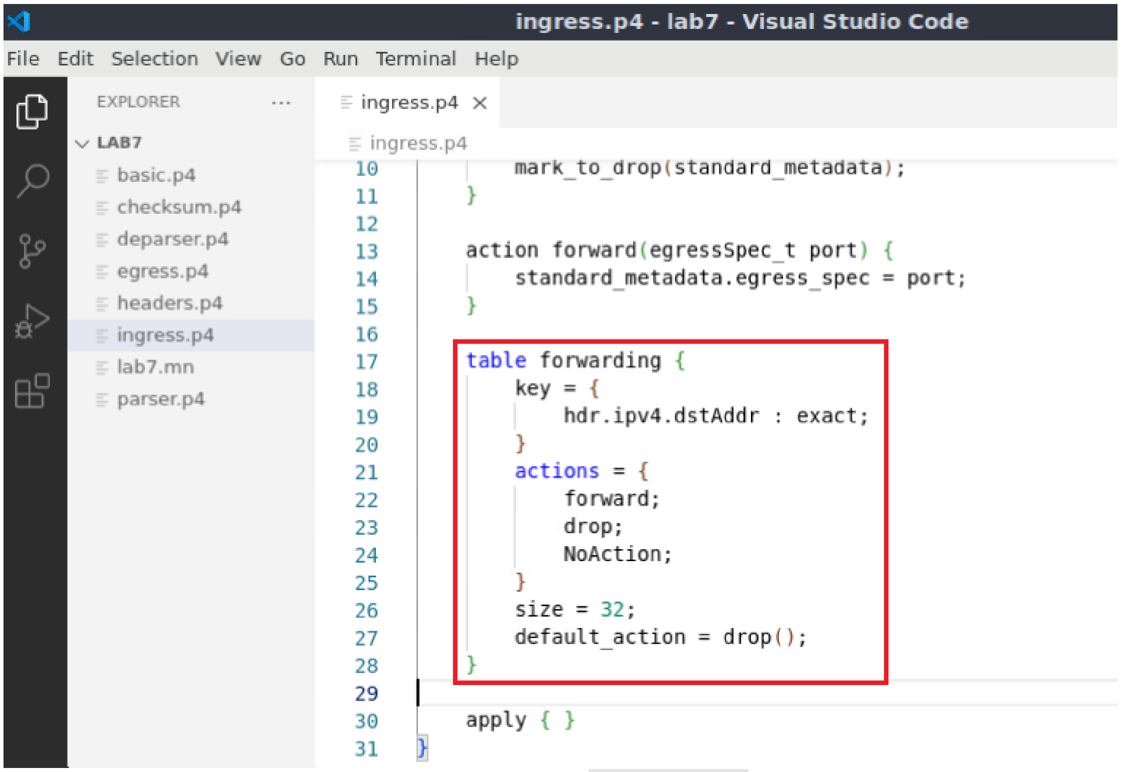


Figure 17. Declaring the `forwarding` table.

The table defined in the figure above matches the destination IPv4 address. The actions in this table can be `forward`, `drop`, or `NoAction`. Note that the table allows up to 32 entries (`size = 32`), and the default action is `drop`.

Step 5. Define the packet processing sequence by adding the following code inside the apply block.

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

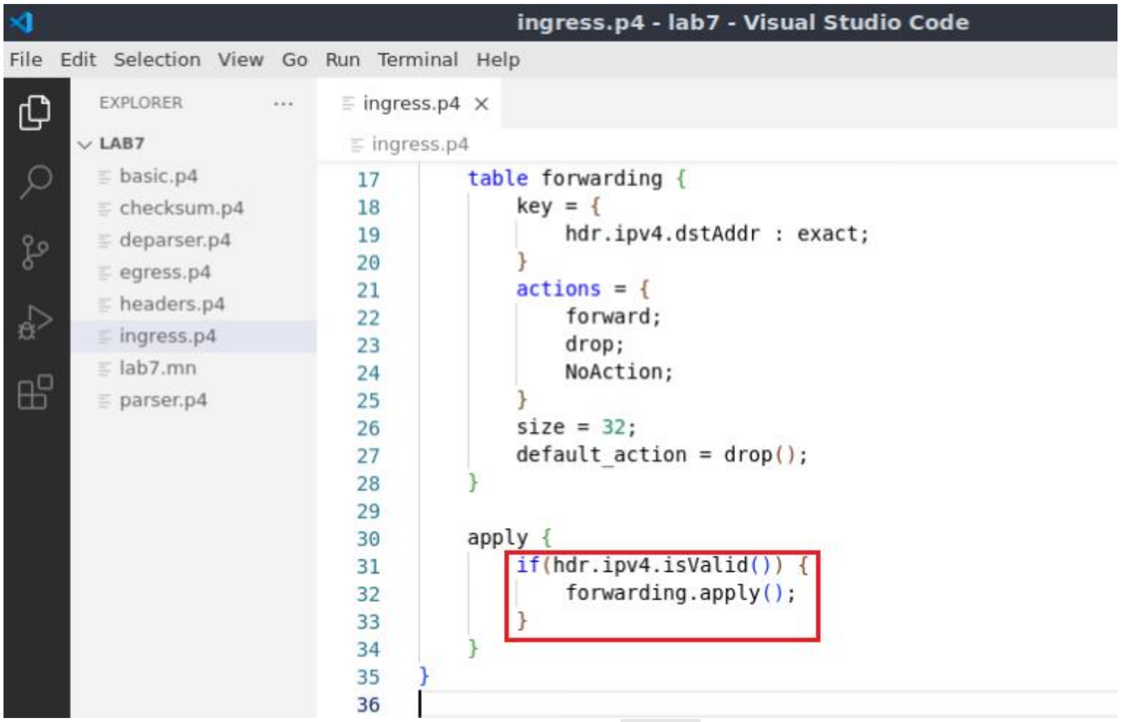


Figure 18. Defining the `apply` block.

Note that the block defined in the figure above applies the `forwarding` table every time there is a packet with a valid IPv4 header.

3.3 Defining a direct counter

Step 1. Declare the direct counter `my_direct_counter` by adding the following line of code. Note that this is a packet counter specified by the argument `CounterType.packets`.

```
direct_counter(CounterType.packets) my_direct_counter;
```

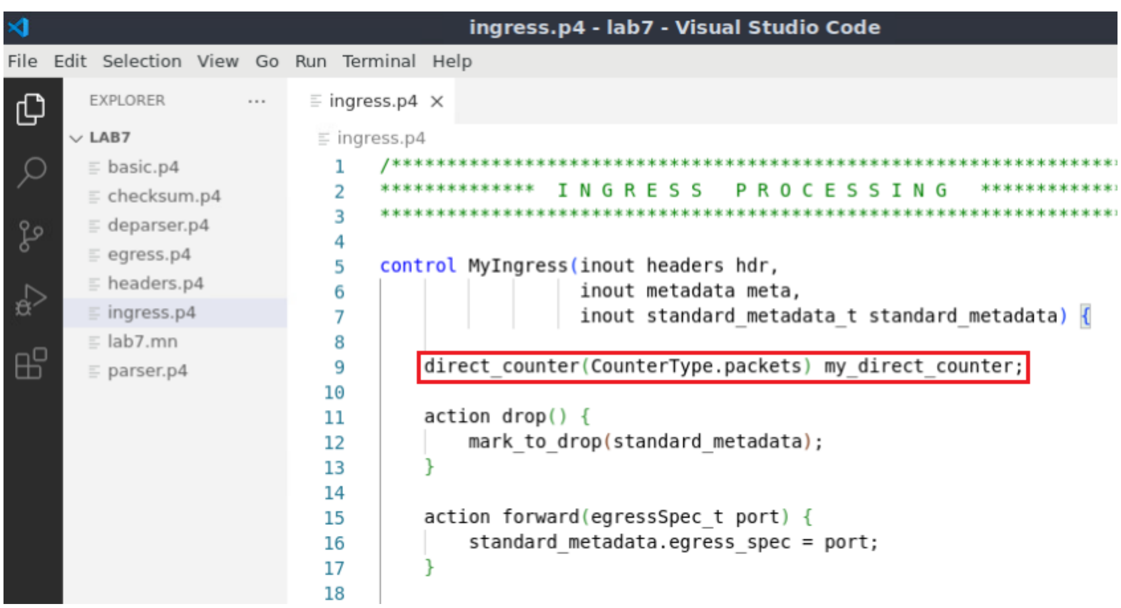


Figure 19. Defining a direct counter.

Step 2. Refer the direct counter in the `forwarding` table by adding the following line.

```
counters = my_direct_counter;
```

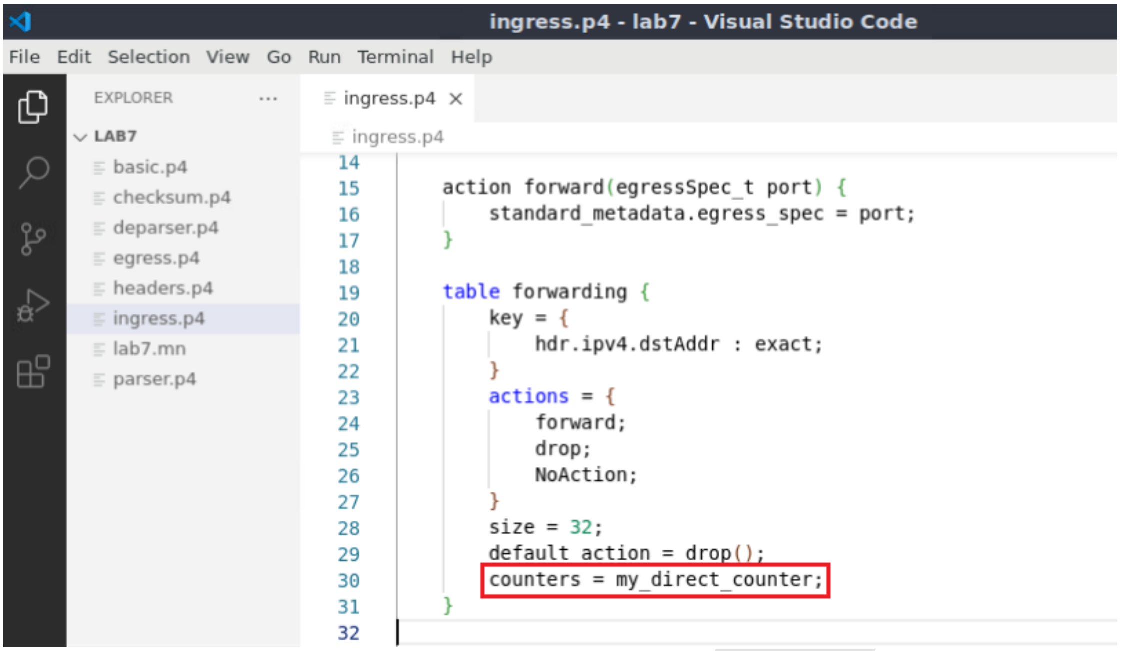


Figure 20. Referring the direct counter to the `forwarding` table.

The statement above will increase the count of `my_direct_counter` by one every time there is a match in the `forwarding` table.

3.4 Defining an indirect counter

Step 1. Declare the indirect counter `my_indirect_counter` by adding the following line of code.

```
counter(3, CounterType.packets) my_indirect_counter;
```

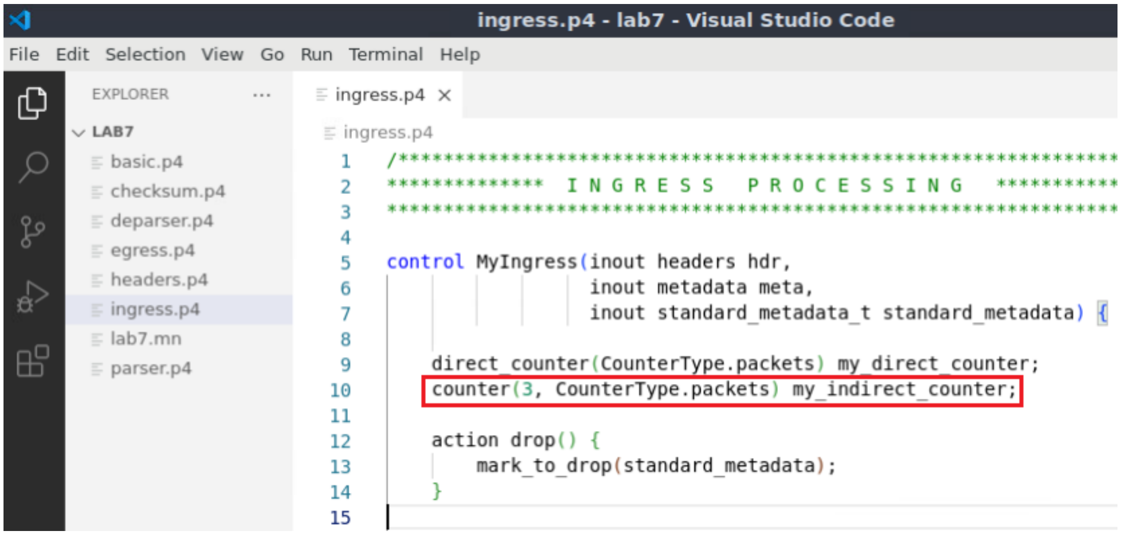


Figure 21. Defining an indirect counter.

The first argument in `counter` specifies the number of independent values of the counter, which in this case is `3`. Note also that this is a packet counter specified by the argument `CounterType.packets`.

Step 2. Refer the indirect counter in the `forward` action by adding the following line of code.

```
my_indirect_counter.count(index);
```

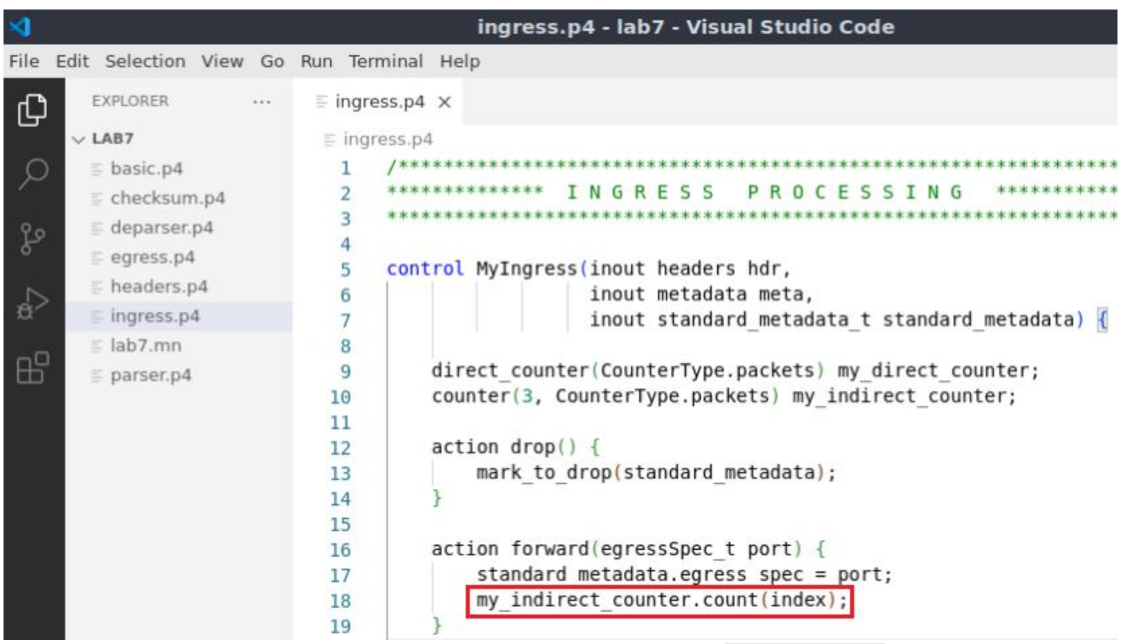


Figure 22. Referring the indirect counter to the `forwarding` table.

Step 3. Include the counter index in the `forward` action's parameter by adding the following statement.

```
bit<32> index
```

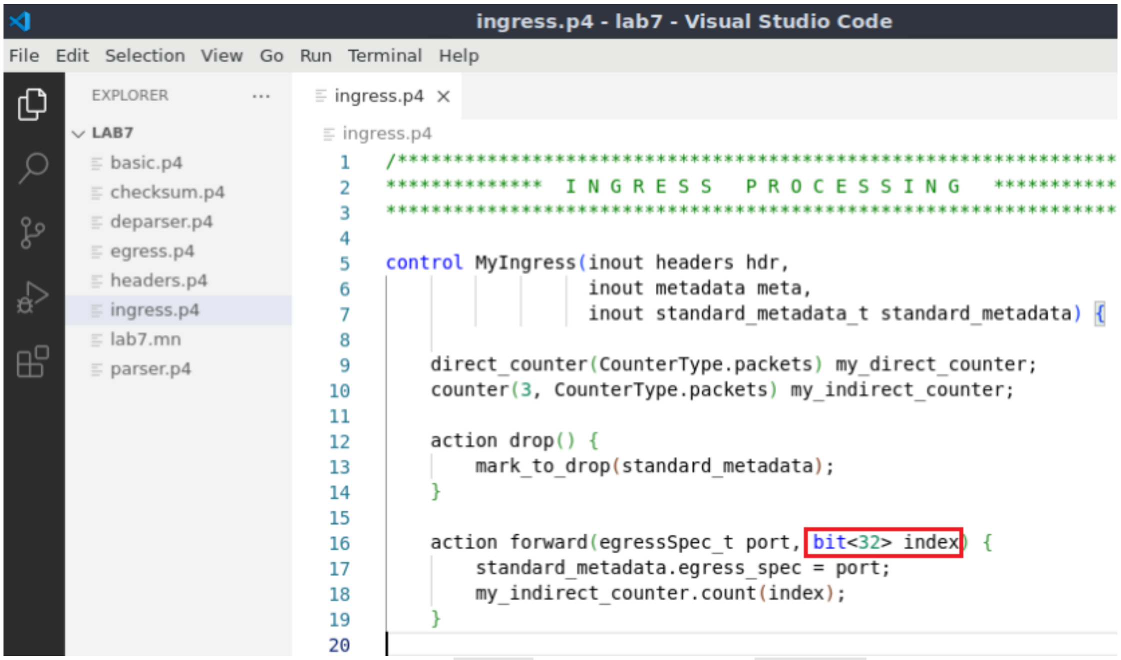


Figure 23. Including the `index` parameter to the `forward` action.

The value of index specifies a counter which can be associated with multiple entries in the `forwarding` table.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

4 Loading the P4 program

In this section, you will compile and load the P4 binary into switch s1. You will also verify that the binary resides in switch s1 filesystem.

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside VS Code to compile the program.

```
p4c basic.p4
```

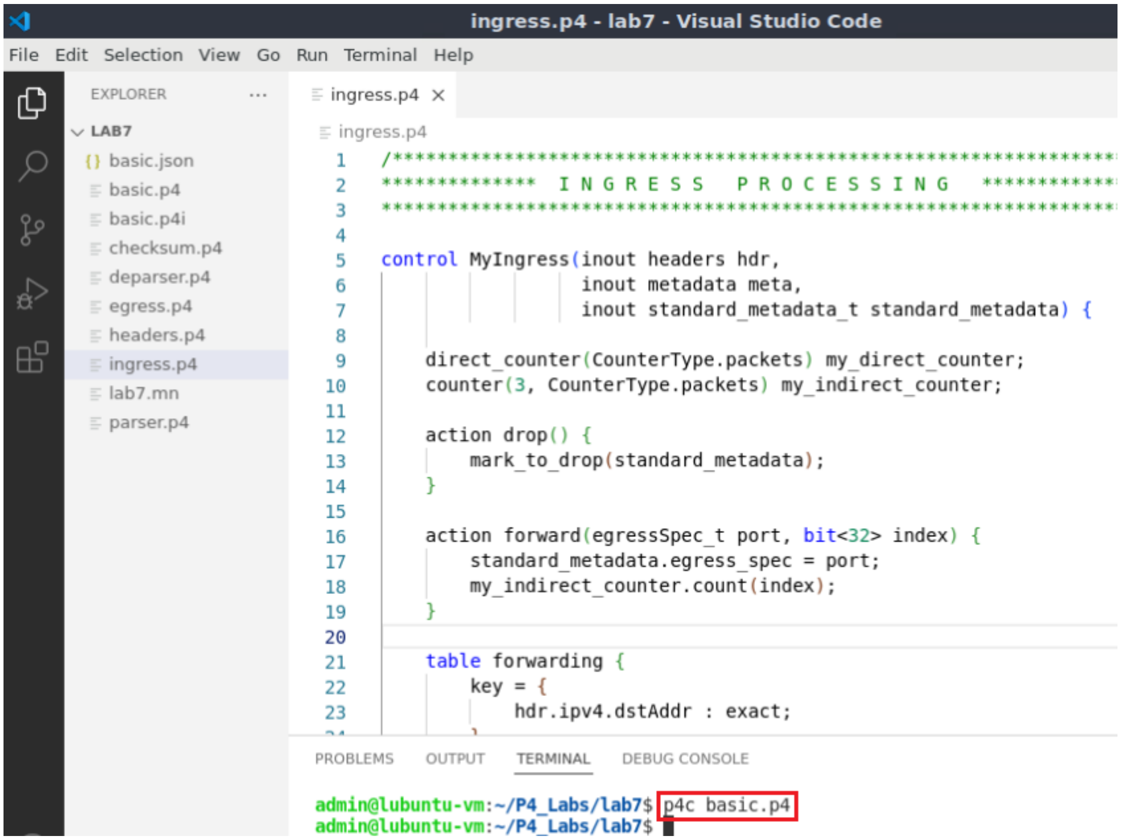


Figure 24. Compiling a P4 program.

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
```

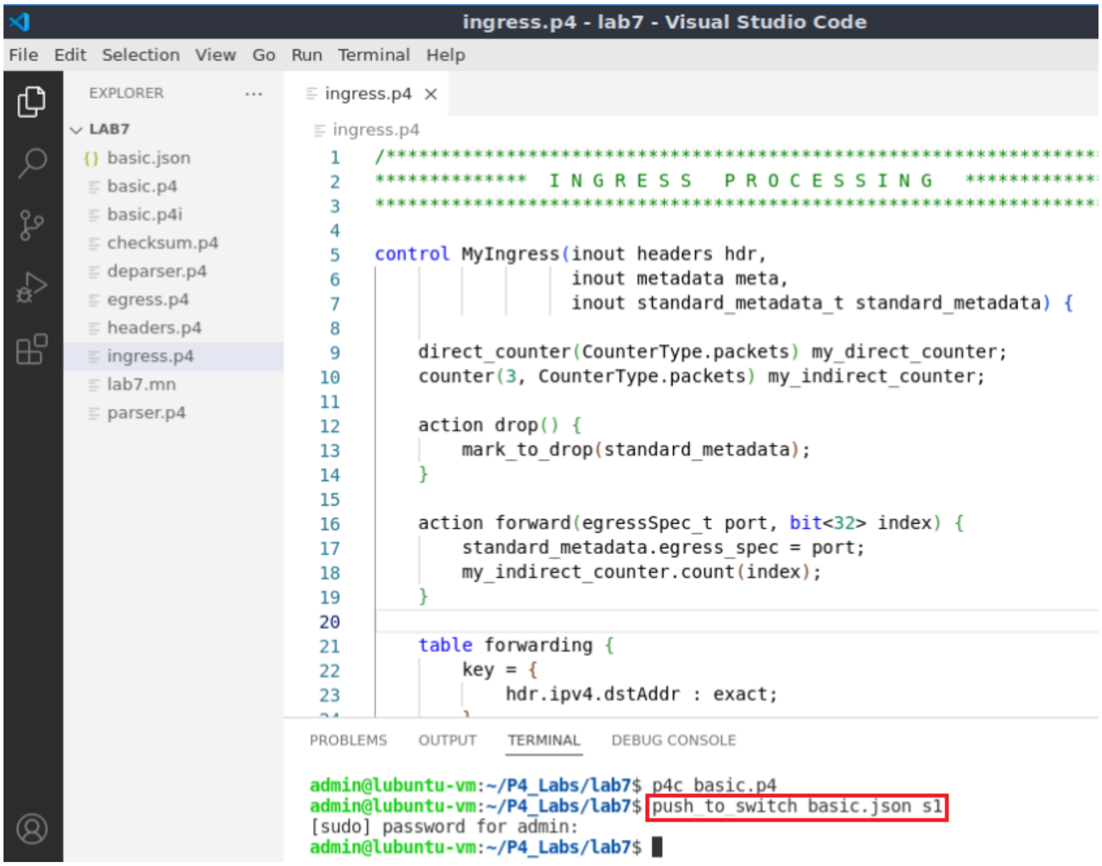


Figure 25. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

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

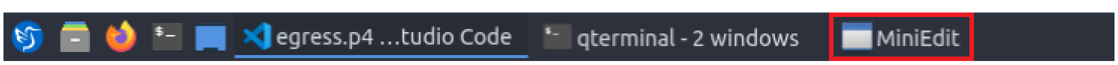


Figure 26. Maximizing the MiniEdit window.

Step 2. Right-click on the switch s1 icon and select *Terminal*.

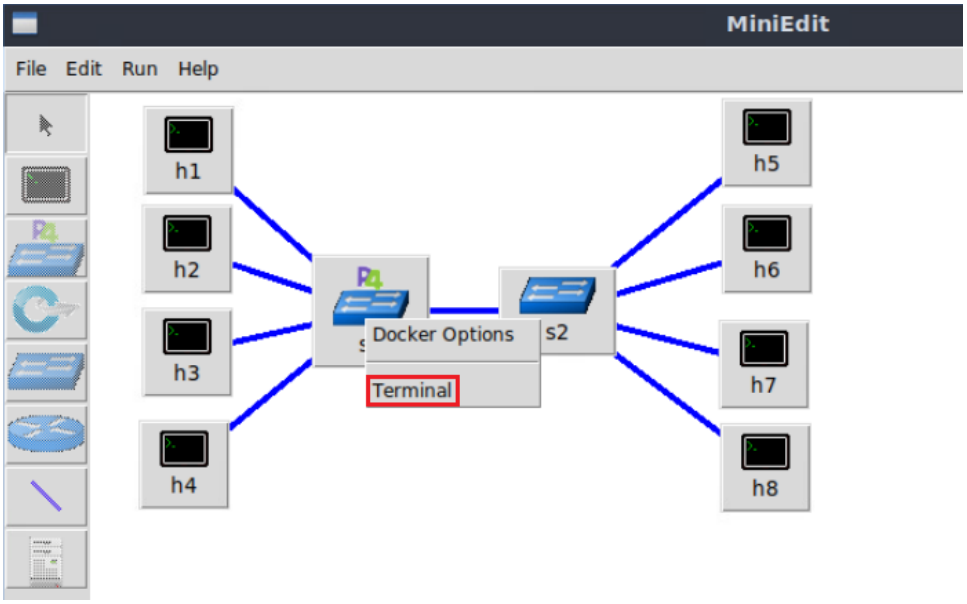


Figure 27. Starting the terminal on switch s1.

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 terminal.

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

```
ls
```

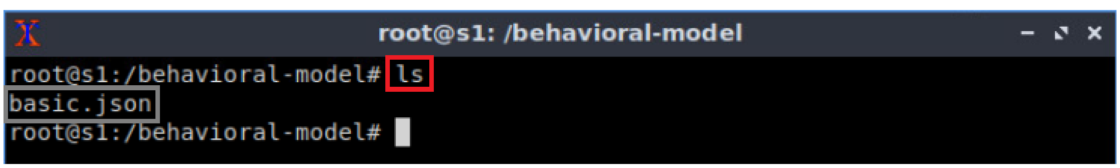


Figure 28. Displaying the contents of the current directory in the switch s1.

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

5 Configuring switch s1

In this section, you will map switch s1 interfaces to the ports in the P4 program and start the switch daemon. Then, you will inspect the switch's logs to see the tables and actions that packets hit and miss. Finally, you will load the rules to populate the match action tables.

5.1 Running the switch's daemon and mapping the ports

Step 1. In switch s1 terminal, start the switch daemon and map the logical interfaces to Linux interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 -i 3@s1-eth3 -i 4@s1-eth4
--nanolog ipc:///tmp/bm-log.ipc basic.json &
```

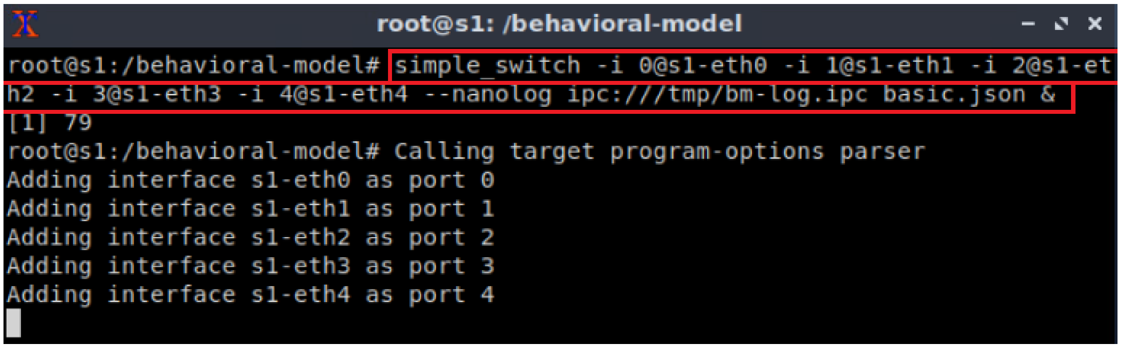


Figure 29. Starting switch s1 daemon and mapping the logical interfaces to Linux interfaces.

5.2 Loading the rules to the switch

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

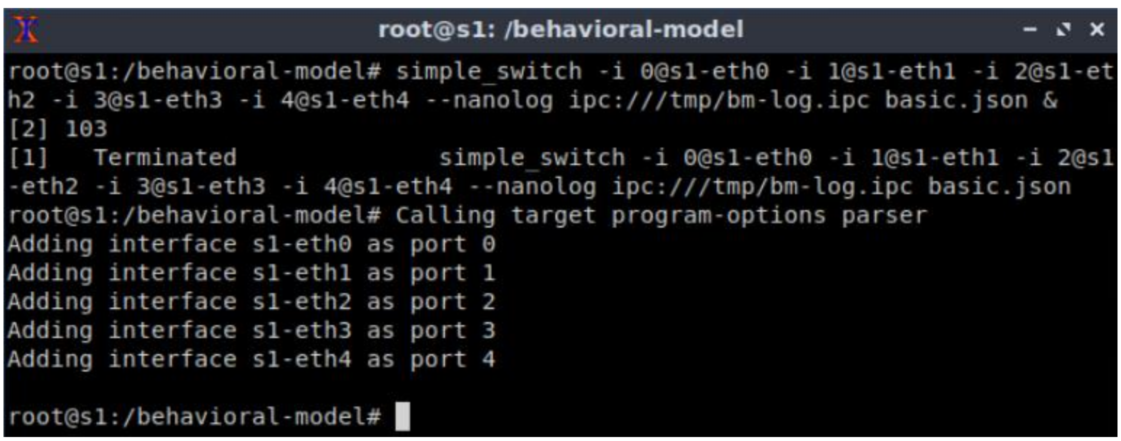


Figure 30. Returning to switch s1 CLI.

Step 2. Inspect the rules' file by issuing the following command.

```
cat ~/lab7/rules.cmd | nl
```

Figure 31. Inspecting the entries of the `forwarding` table.

The output of the figure above shows the rules to be populated in the table forwarding. Notice that each key has two action data: the egress port and the index of the indirect counter. For example, entry 3 has 10.0.0.3 as the key, 3 as the egress port, and 0 as the index of the indirect counter.

Step 3. Push the table entries to switch s1 by typing the following command.

```
simple_switch_CLI < ~/lab7/rules.cmd
```

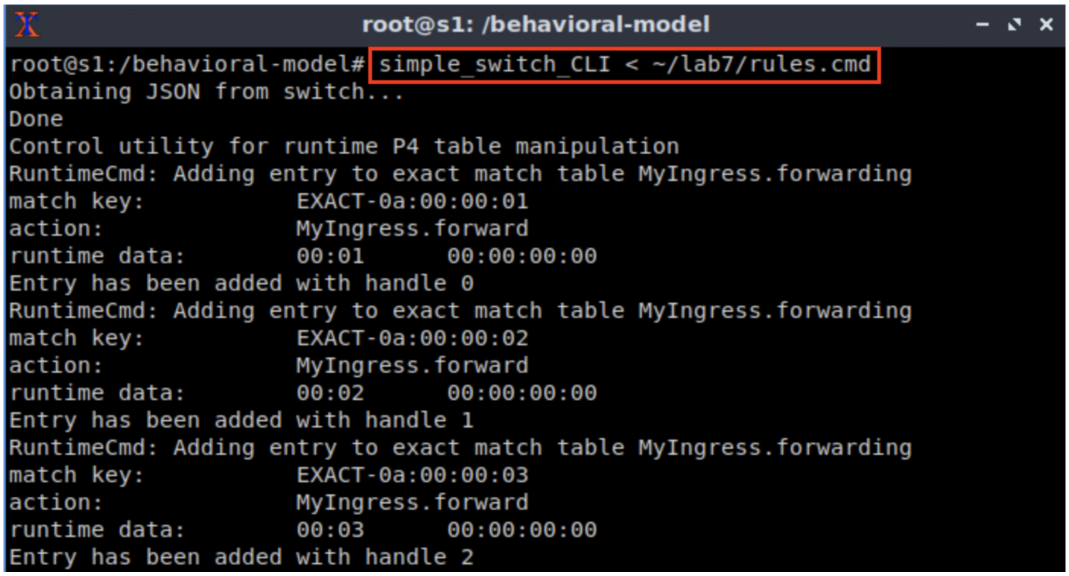


Figure 32. Populating the forwarding table into switch s1.

6 Testing and verifying the P4 program

In this section, you will run iperf3 tests between hosts to test the P4 program loaded to switch s1. You will verify the count of the direct counter from the control plane. Then, you will refer the indexes of the indirect counters to specific entries of the forwarding table. Finally, you will verify the count values of the indirect counters.

6.1 Running iperf3 tests between end hosts

Step 1. Open a terminal in host h5 and start the iperf3 server by issuing the following command.

```
iperf3 -s
```

Figure 33. Starting an iperf3 server in host h5.

Step 2. Repeat the previous step in hosts h6, h7, and h8.

Step 3. Go back to host h1 terminal and start the iperf3 client by issuing the following command.

```
iperf3 -c 10.0.0.5
```

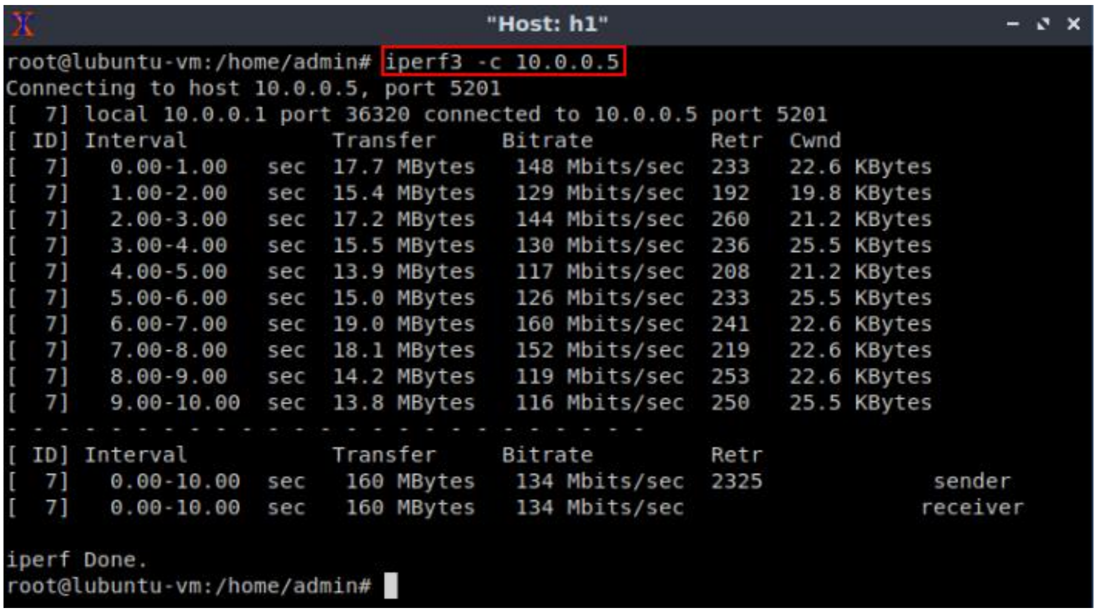


Figure 34. Running an iperf3 test between host h1 and host h5.

6.2 Verifying the counters' values

Step 1. In switch's s1 terminal, start the switch's CLI by issuing the following command.

```
simple_switch_CLI
```

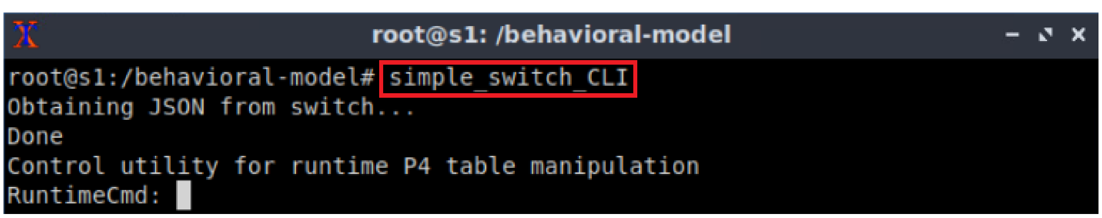


Figure 35. Starting the switch's CLI.

Step 2. Issue the following command to read the value of the direct counter associated to the destination IP address 10.0.0.5.

```
counter_read MyIngress.my_direct_counter 4
```

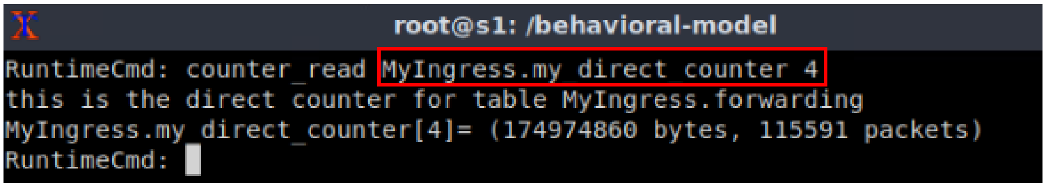


Figure 36. Reading the direct counter at index 4.

The figure above shows that the direct counter at index zero counted 174974860 bytes or 115591 packets.

Step 3. Issue the following command to read the value of the direct counter associated to the destination IP address 10.0.0.6.

```
counter_read MyIngress.my_direct_counter 5
```

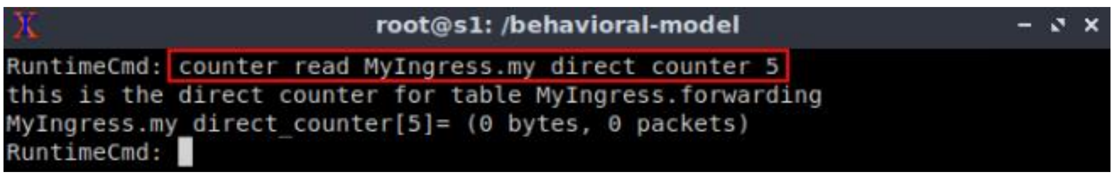


Figure 37. Reading the direct counter at index 5.

Note that the counter at index 5 in the figure above is zero because there are no packets sent to the destination IP 10.0.0.6 yet.

Step 4. Open a terminal in host h2 and start the iperf3 client by issuing the following command.

```
iperf3 -c 10.0.0.6
```

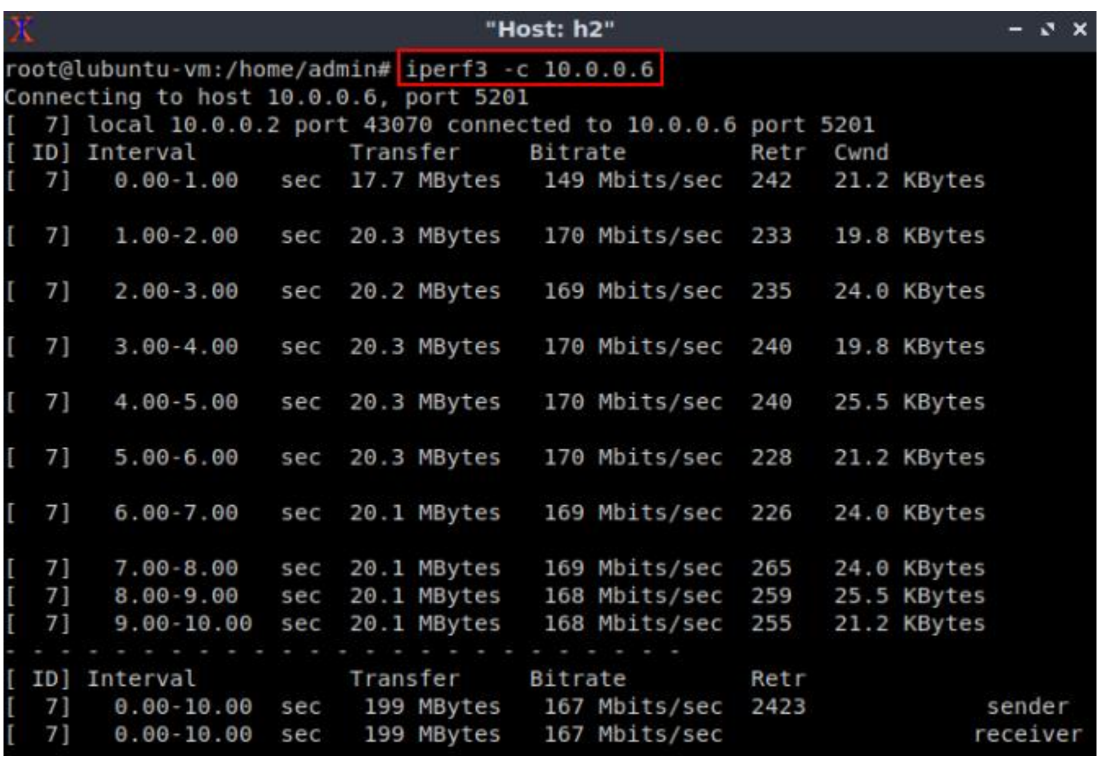


Figure 38. Running an iperf3 test between host h2 and host h6.

Step 5. Issue the following command to read the value of the direct counter associated to the destination IP address 10.0.0.6.

```
counter_read MyIngress.my_direct_counter 5
```

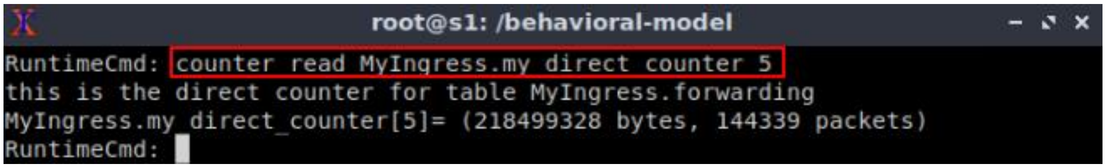


Figure 39. Reading the direct counter with index 5.

After finishing the iperf3 test between host h2 and host h6, the counter at index 5 increased.

6.3 Referring indirect counter indexes

Step 1. In the switch's CLI, issue the following command to refer index 1 of the indirect counter to the flow with destination IP address 10.0.0.7.

```
table_modify MyIngress.forwarding MyIngress.forward 6 0 1
```

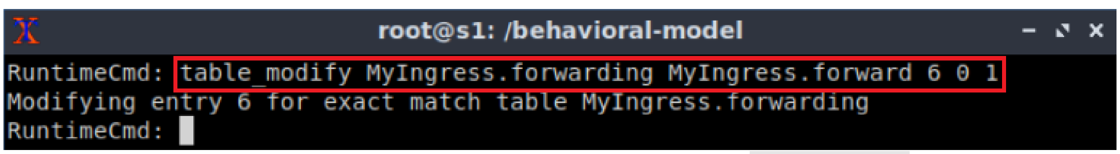


Figure 40. Modifying the entry with handle 6 in the `forwarding` table.

The command and its parameters in the figure above are explained as follows:

- `table_modify`: enables modifying a table entry.
- `MyIngress.forwarding`: the name of the table.
- `MyIngress.forward`: the action.
- `6`: the handle of the entry.
- `0`: the egress port.
- `1`: the counter's index.

Step 2. Similarly, issue the following command to refer the index 2 of the indirect counter to the flow with destination IP address 10.0.0.8.

```
table_modify MyIngress.forwarding MyIngress.forward 7 0 2
```

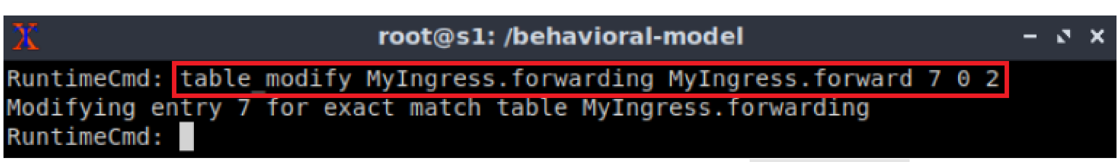


Figure 41. Modifying the entry with handle 7 in the `forwarding` table.

The command and its parameters in the figure above are explained as follows:

- `table_modify`: enables modifying a table entry.
- `MyIngress.forwarding`: the name of the table.
- `MyIngress.forward`: the action.
- `7`: the handle of the entry.
- `0`: the egress port.
- `2`: the counter's index.

Step 3. Open a terminal in host h3 and start the iperf3 client by issuing the following command.

```
iperf3 -c 10.0.0.7
```

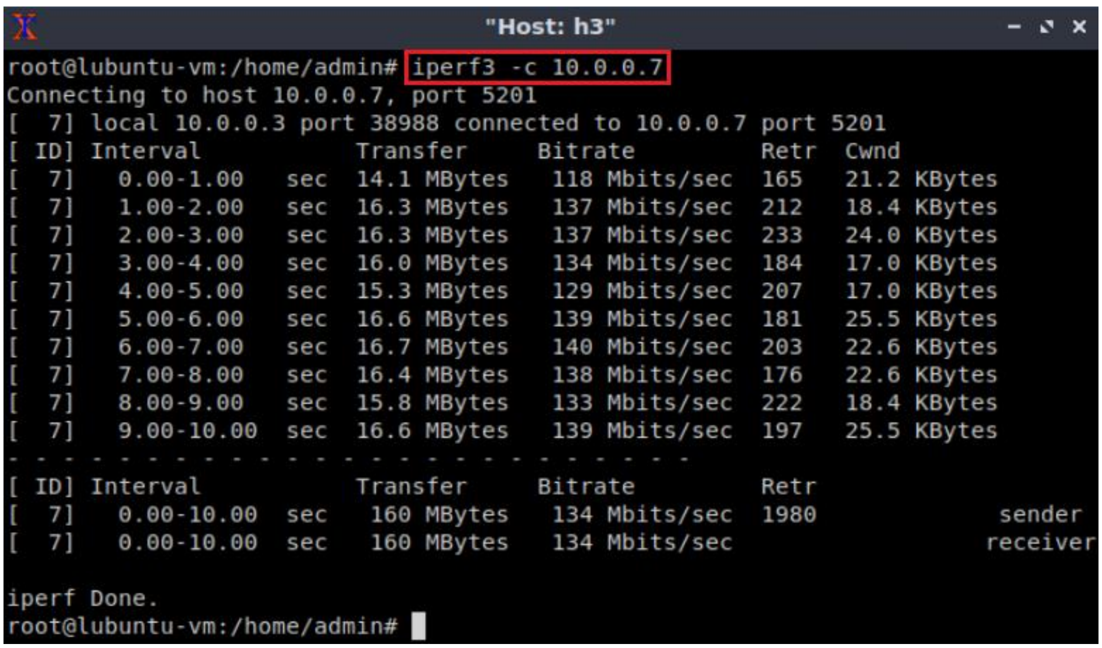


Figure 42. Running an iperf3 test between host h3 and host h7.

Step 4. Issue the following command to read the value of the indirect counter associated to the destination IP address 10.0.0.7.

```
counter_read MyIngress.my_indirect_counter 1
```

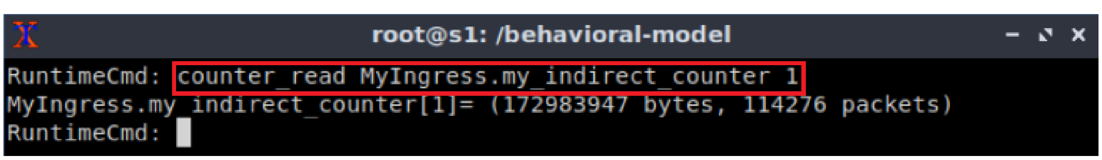


Figure 43. Reading the indirect counter with index 1.

The figure above shows that the indirect counter at index one counted 172,983,947 bytes or 114,276 packets.

Step 5. Similarly, open a terminal in host h4 and start the iperf3 client by issuing the following command.

```
iperf3 -c 10.0.0.8
```

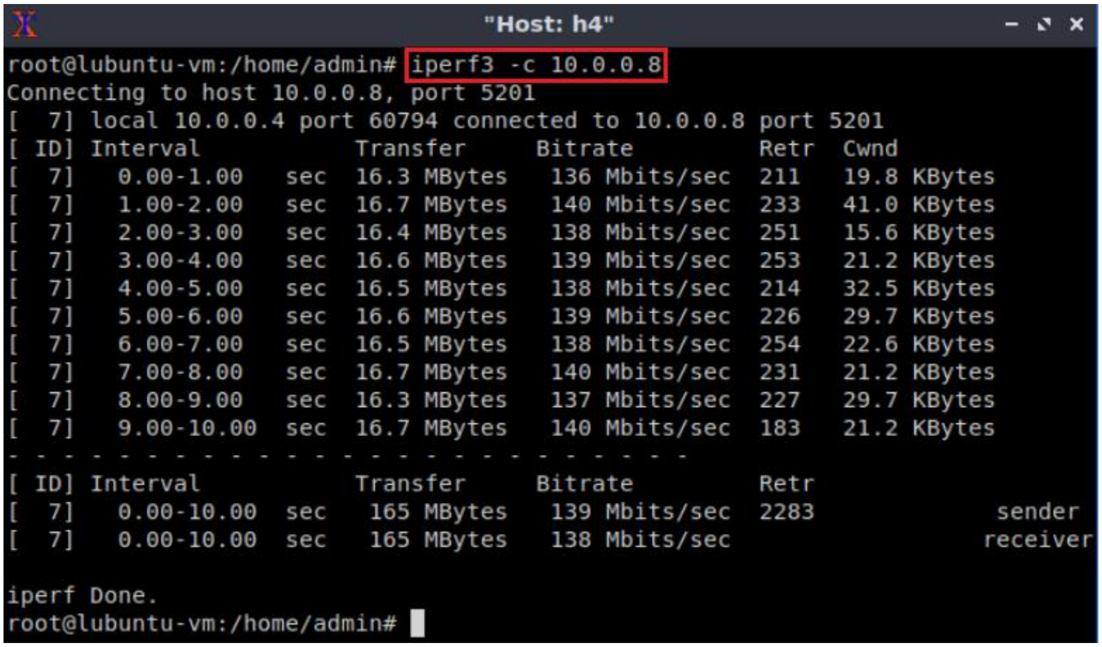


Figure 44. Running an iperf3 test between host h4 and host h8.

Step 6. Issue the following command to read the value of the indirect counter associated to the destination IP address 10.0.0.8.

```
counter_read MyIngress.my_indirect_counter 2
```

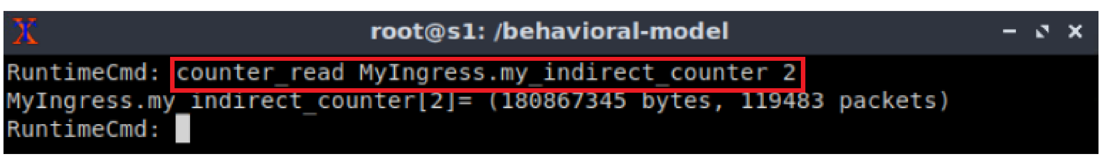


Figure 45. Reading the indirect counter with index 2.

Step 7. Finally, issue the following command to read the value of the indirect counter at index 0. Note that this value aggregates the packet count of all the flows with IP destination addresses 10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4, 10.0.0.5, and 10.0.0.6 that passed through switch s1. Similarly, the indirect counters at indices 1 and 2 aggregate the packet count of all the flows with IP destination addresses 10.0.0.7 and 10.0.0.8, respectively.

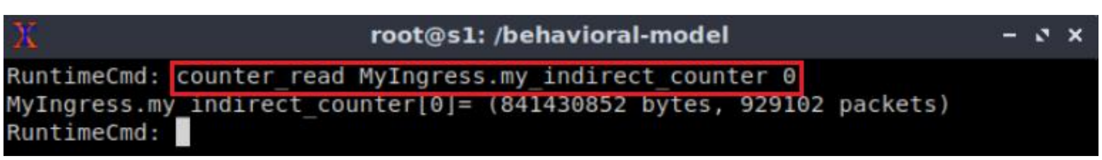


Figure 46. Reading the indirect counter with index 0.

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

References

1. The P4 Language Consortium. "*The BMv2 Simple Switch target*." [Online]. Available: https://tinyurl.com/mr3m59ph
2. The P4 Architecture Working Group. "*P4$_{16}$ Portable Switch Architecture (PSA).*" [Online]. Available: https://tinyurl.com/2wnkc6d2
3. Mininet walkthrough. [Online]. Available: http://Mininet.org.
4. 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.
5. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
6. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.

UNIVERSITY OF SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 8: Rerouting Traffic using Meters

Document Version: 04-25-2022

Contents

Overview

This lab introduces the reader to meters which are stateful elements used in P4 to measure and mark the rate of incoming traffic. In this lab, the user will create a P4 program to reroute traffic based on the measurement provided by the meter using a match-action table. Finally, the user will verify the P4 program by conducting throughput tests at different rates.

Objectives

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

1. Declare a custom metadata to store the meter's color.
2. Define a direct meter.
3. Refer a meter in a match-action table.
4. Reroute traffic based on the meter's color.

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 meters.
2. Section 2: Lab topology.
3. Section 3: Implementing a meter at the ingress pipeline.
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 meters

Meters[1] are provided as stateful objects by a P4 switch. Consider Figure 1 which shows the architecture of a meter. There are two token buckets *p* and *c*, with sizes peak burst size (PBS) and committed burst size (CBS), respectively. The buckets *p* and *c* are filled with tokens at rates peak information rate (PIR) and committed information rate (CIR), respectively. Upon receiving a packet of size *b* bytes at time *t*, the meter checks if *Tp(t) - b < 0*, where *Tp(t)* is the token count of bucket *p* at time *t*. In other words, the meter is checking if the packet rate is exceeding PIR, causing the bucket *p* to become empty. If the condition is met, the meter outputs the color *red*'. Otherwise, the meter checks if *Tc(t) - b < 0*, where *Tc(t)* is the token count of bucket *c* at time *t*. If the condition is met, the meter outputs the color *yellow*; otherwise, the meter outputs the color *green*.

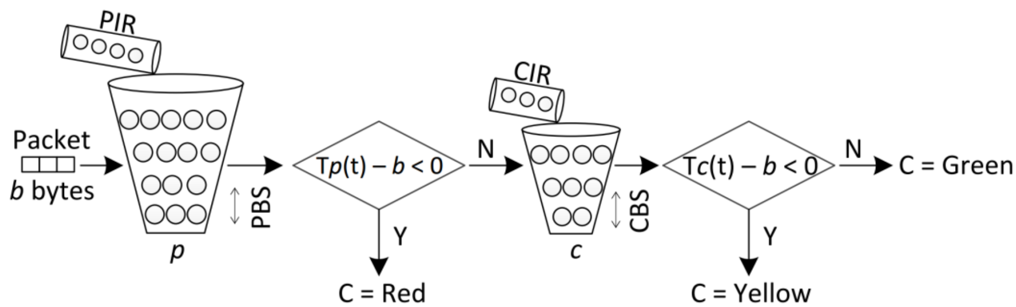


Figure 1. Scheme of the two-rate three-color marker (trTCM) meter[1].

1.1 Declaring and invoking meters in P4

Figure 2 shows a P4 code that describes how to declare and invoke a direct meter in P4. The code describes the ingress pipeline which has a forwarding table and two actions (i.e., `forward` and `drop`). The direct meter `my_direct_meter` is declared in line 5. Note that the width of the meter is 32 bits, and the output gives the byte rate in bytes/microsecond[3]. Then, the direct meter is invoked in the forwarding table (see line 26).

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

direct_meter<bit<32>>(MeterType.bytes) my_direct_meter;

action forward(egressSpect_t port){
      standard_meadata.egress_spec = port;
}

action drop(){
      mark_to_drop(standard_metadata);
}

table forwarding {
      key = {
          hdr.ethernet.dstAddr : exact;
       }
      actions = {
          forward;
          drop;
          NoAction;
      }
      size = 32;
      default_action = drop();
      meters = my_direct_meter;
}
apply {
      if(hdr.ipv4.isValid()){
         forwarding.apply();
      }
}
```

Figure 2. Declaring and invoking a direct meter in the ingress pipeline.

The programmer can invoke the following function to read the color of the meter `my_direct_meter`, where `result` will store the value of the color.

```
my_direct_meter.read(result);
```

The meter colors are encoded as follows:

- 0: Green
- 1: Yellow
- 2: Red

1.2 Lab scenario

The topology in Figure 3 describes the lab scenario where the P4 switch (i.e., switch s1) uses meters to determine the sending rate of host h1 and reroute the traffic according to the following rules.

- Route 1: if the sending rate is less than 100Mbps.
- Route 2: if the sending rate is between 100Mbps and 500Mbps.
- Route 3: if the sending rate is greater than 500Mbps.

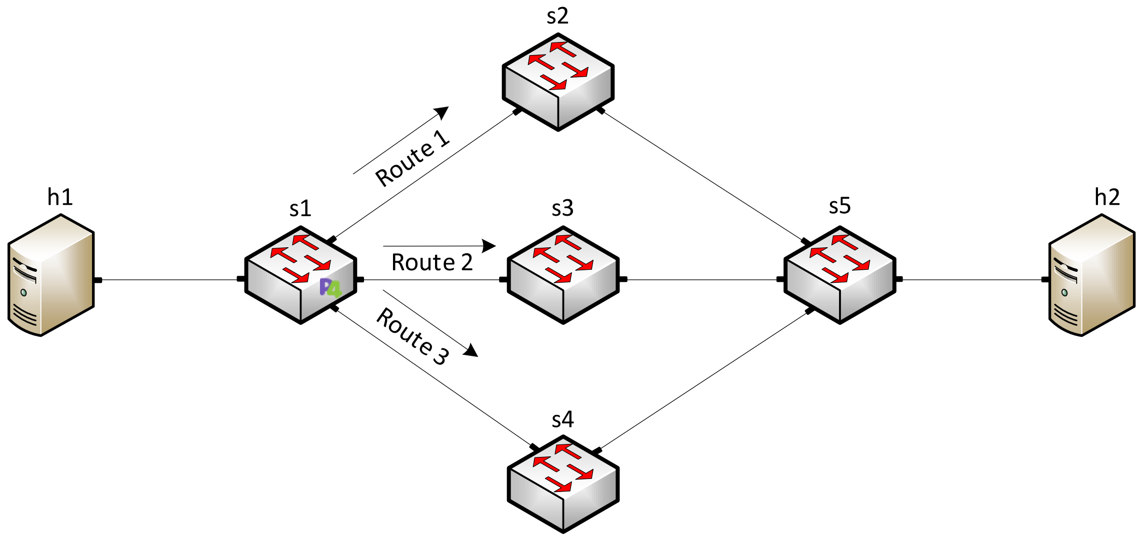


Figure 3. Lab scenario.

2 Lab topology

Let us get started by loading a simple Mininet topology using MiniEdit. The topology comprises two end hosts, one P4 switch, and four legacy switches.

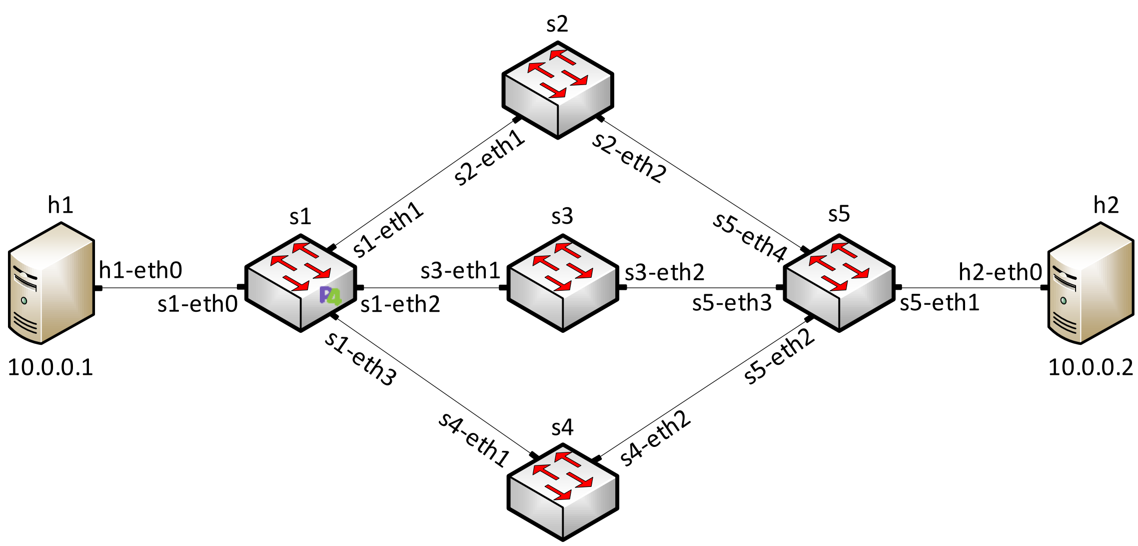


Figure 4. Lab topology.

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 *lab8* folder and search for the topology file called *lab8.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

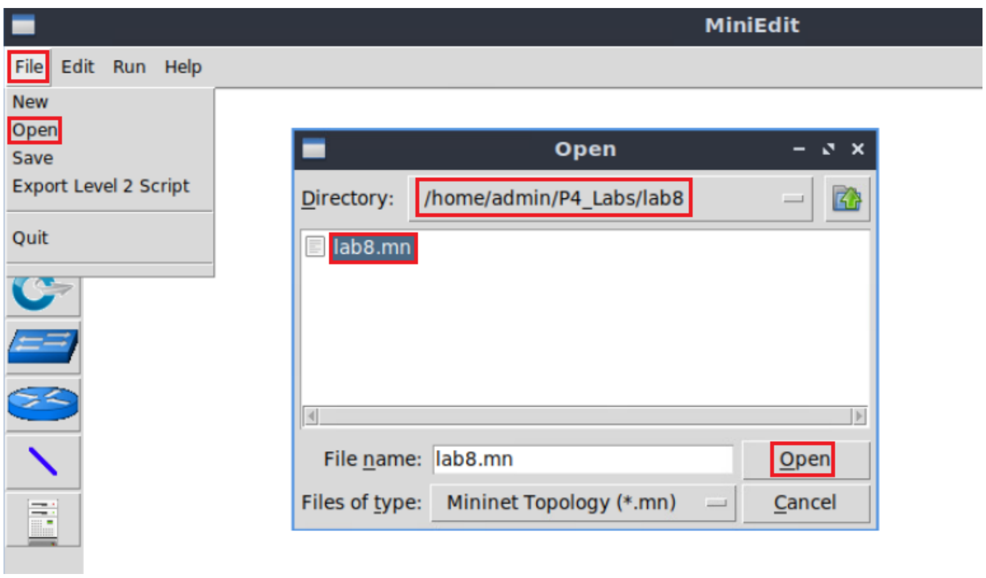


Figure 6. MiniEdit's *Open* dialog.

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 7. 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.

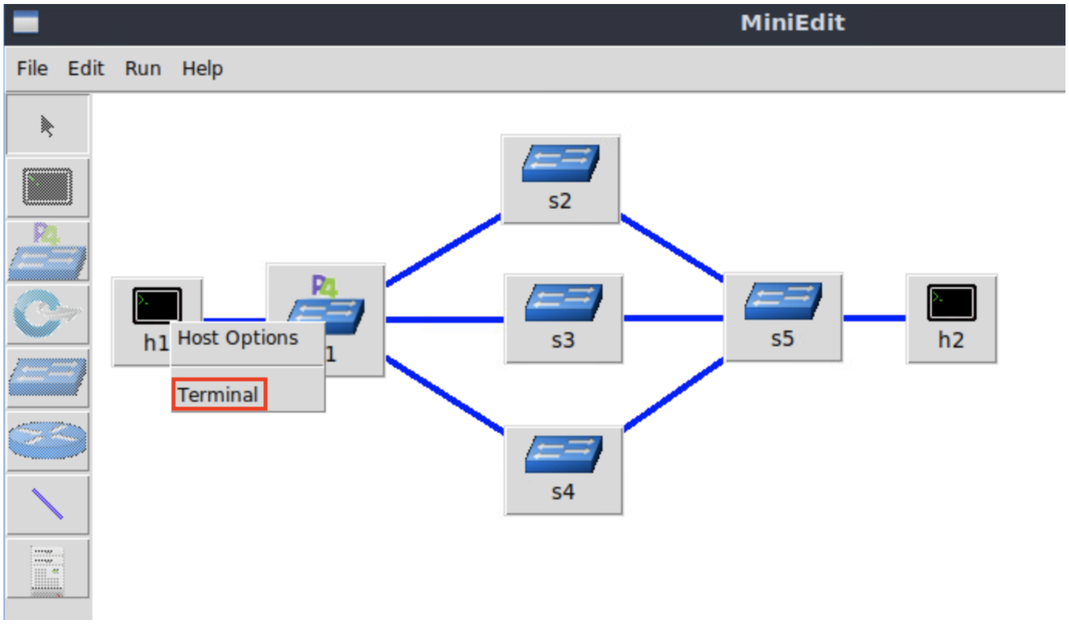


Figure 8. Opening a terminal on host h1.

Step 2. Run the following command to display interfaces' information on host h1.

```
ifconfig
```

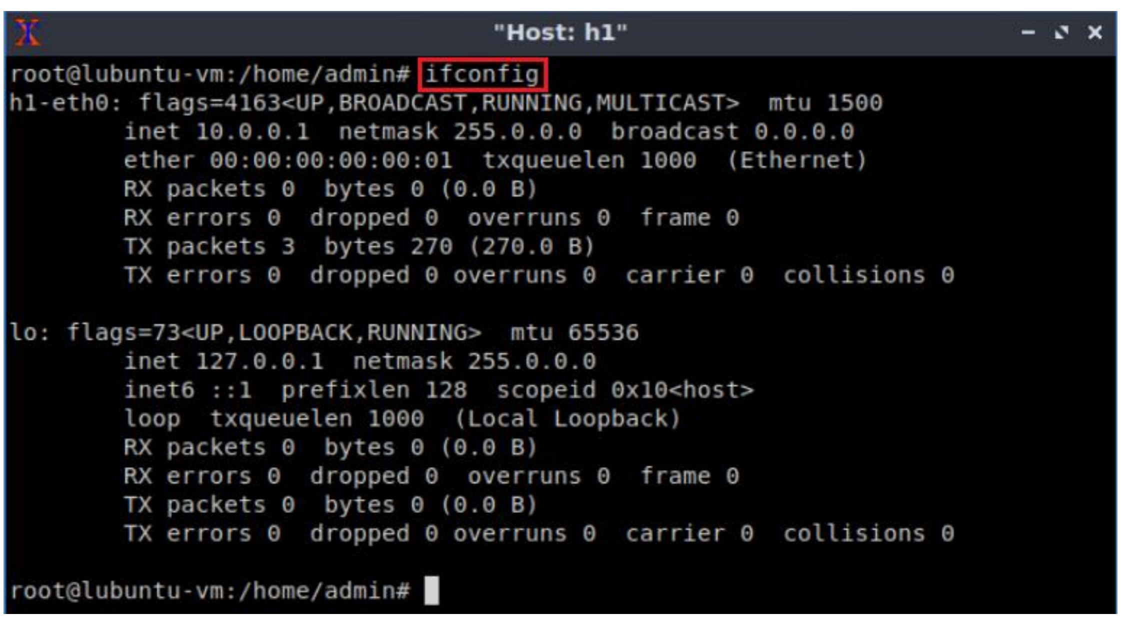


Figure 9. Displaying interfaces' information on host h1.

3 Implementing a meter at the ingress pipeline

In this section, you will load the programming environment, declare custom metadata to store the meter's color, and define a match-action table that matches the destination IPv4 address to forwarding packets. This table will have a direct meter associated with an entry. Then, you will define a table that will reroute traffic based on the meter's color.

3.1 Loading the programming environment

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

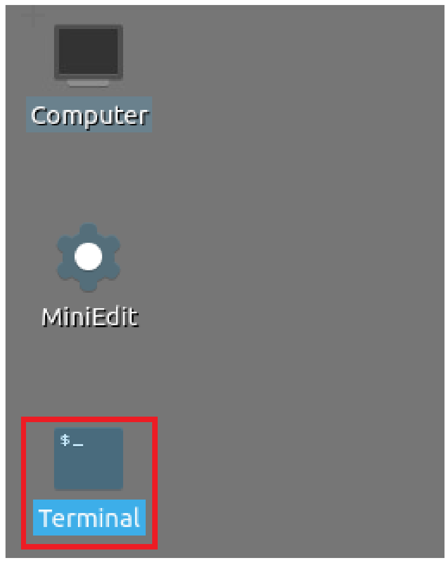


Figure 10. Shortcut to open a Linux terminal.

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 execute.

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
```

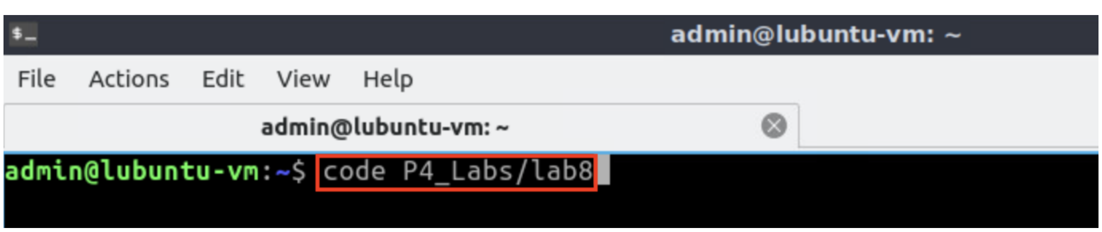


Figure 11. Loading the development environment.

3.2 Defining a custom metadata

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.

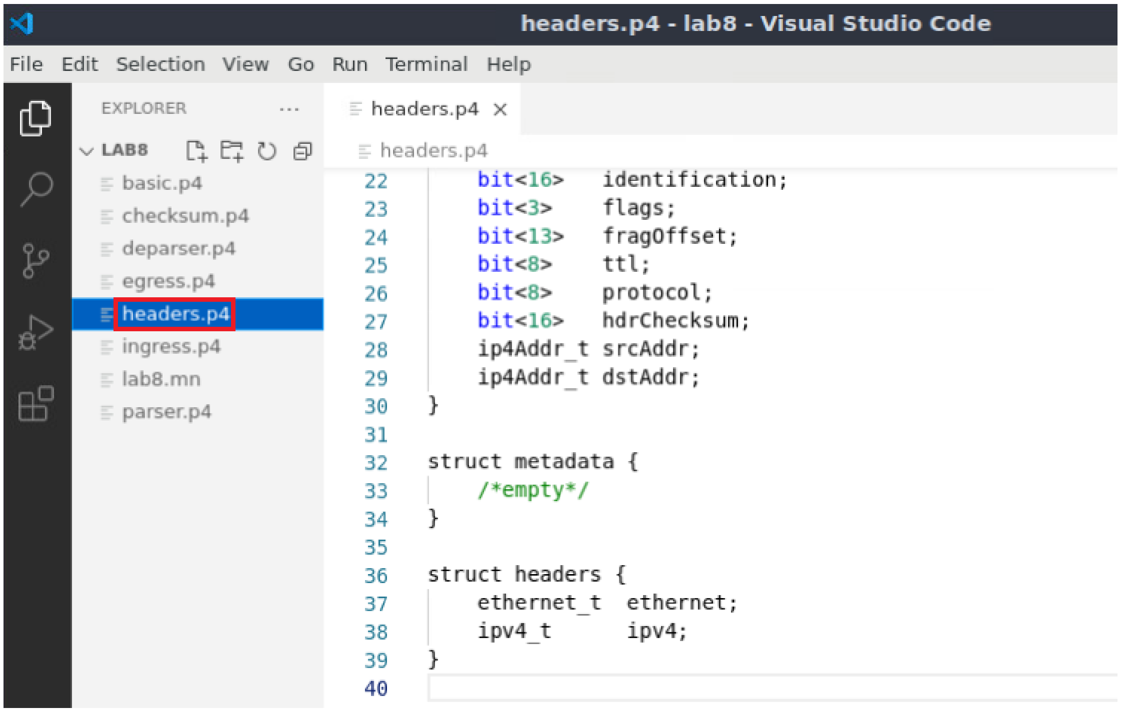


Figure 12. Inspecting the *headers.p4* file.

Step 2. Define the following custom metadata by adding the following line of code.

```
bit<32> meter_color;
```

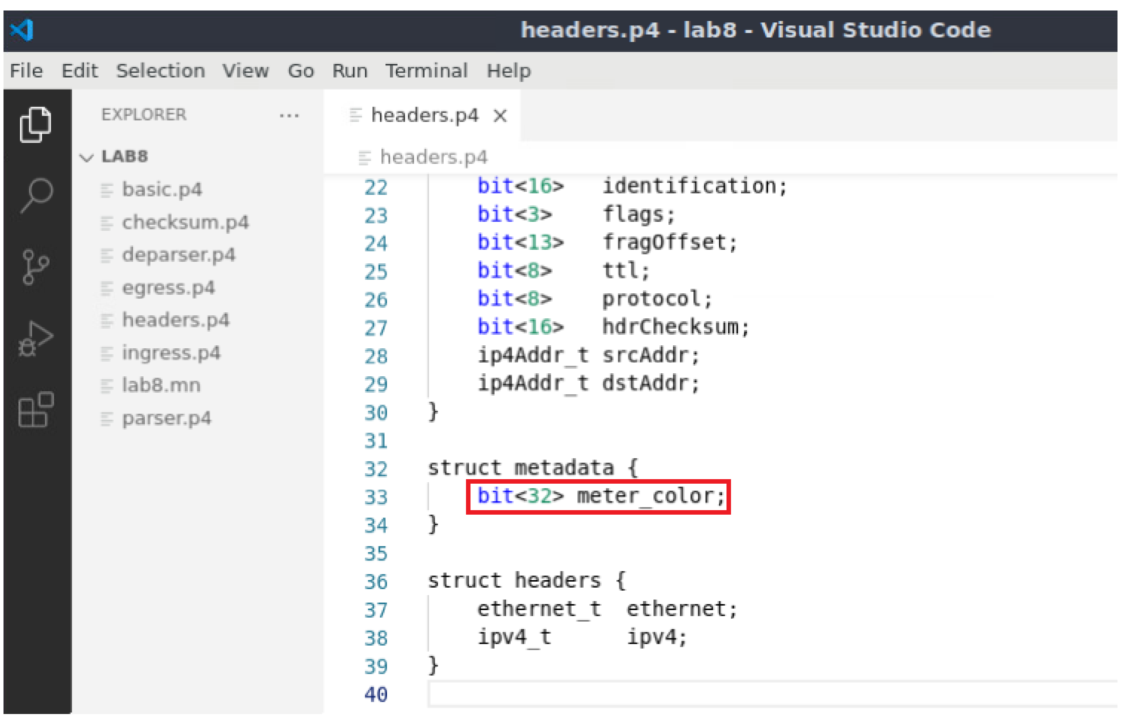


Figure 13. Defining a custom metadata.

The custom metadata in the figure above will store the value of the meter's color: 0 for green, 1 for yellow, and 2 for red.

Step 3. Save the changes to the file by pressing `Ctrl + s`.

3.3 Defining the forwarding behavior

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. You will observe that the block has no match-action table implemented.

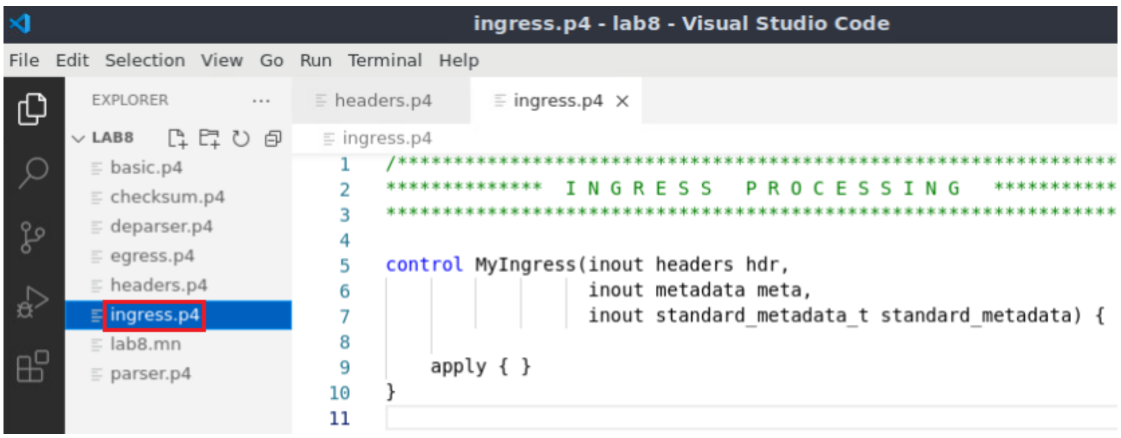


Figure 14. Inspecting the ingress processing block.

Step 2. Define the `drop` action by adding the following code. This action is invoked to drop packets.

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

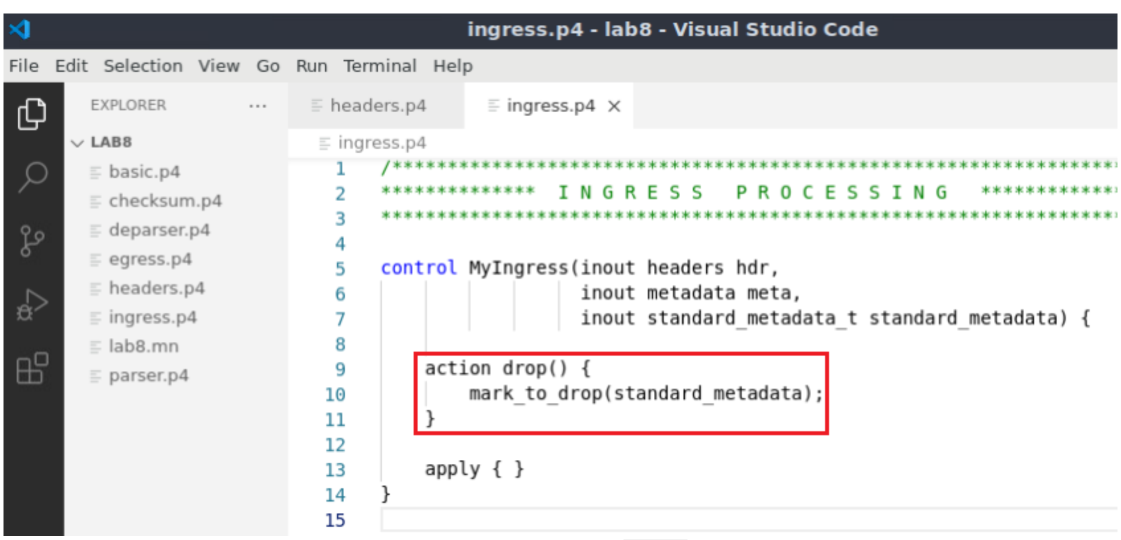


Figure 15. Defining the `drop` action.

Step 3. Define the `forward` action by adding the code shown below. This action forwards packets through an egress port specified by the control plane.

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

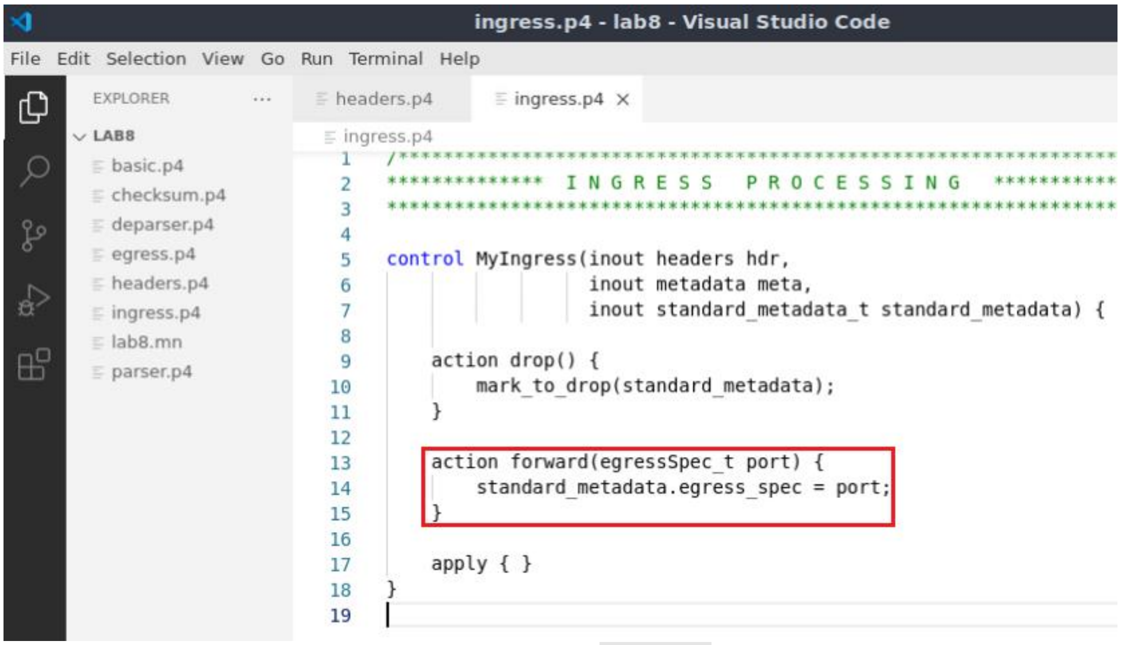


Figure 16. Defining the `forward` action.

Step 4. Define the table `forwarding` by adding the following piece of code.

```
table forwarding {
       key = {
              hdr.ipv4.dstAddr : exact;
       }
       actions = {
           forward;
           drop;
           NoAction;
       }
       size = 32;
       default_action = drop();

}
```

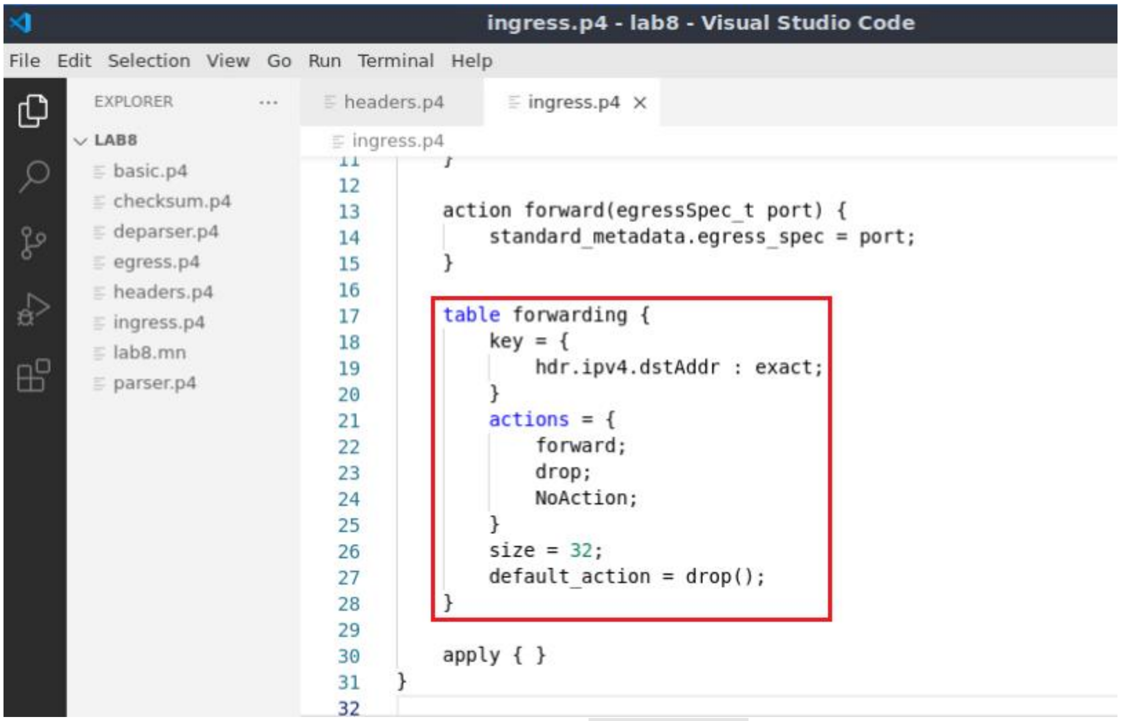


Figure 17. Declaring the `forwarding` table.

The table defined in the figure above matches the destination IPv4 address. The actions in this table can be `forward`, `drop`, or `NoAction`. Note that the table allows up to 32 entries (`size = 32`), and the default action is `drop`.

Step 5. Define the packet processing sequence by adding the following code inside the apply block.

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

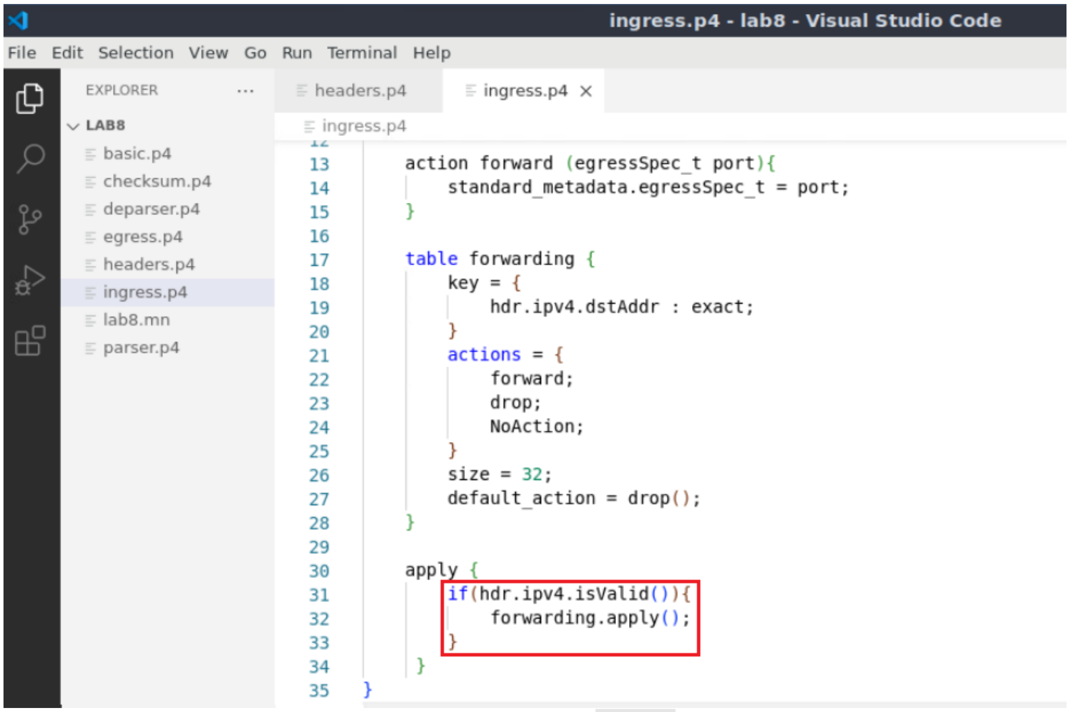


Figure 18. Defining the `apply` block.

Note that the block defined in the figure above applies the `forwarding` table every time there is a packet with a valid IPv4 header.

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

3.4 Defining a direct meter

Step 1. Declare the direct meter `my_direct_meter` by adding the following line of code.

```
direct_meter<bit<32>>(MeterType.bytes) my_direct_meter;
```

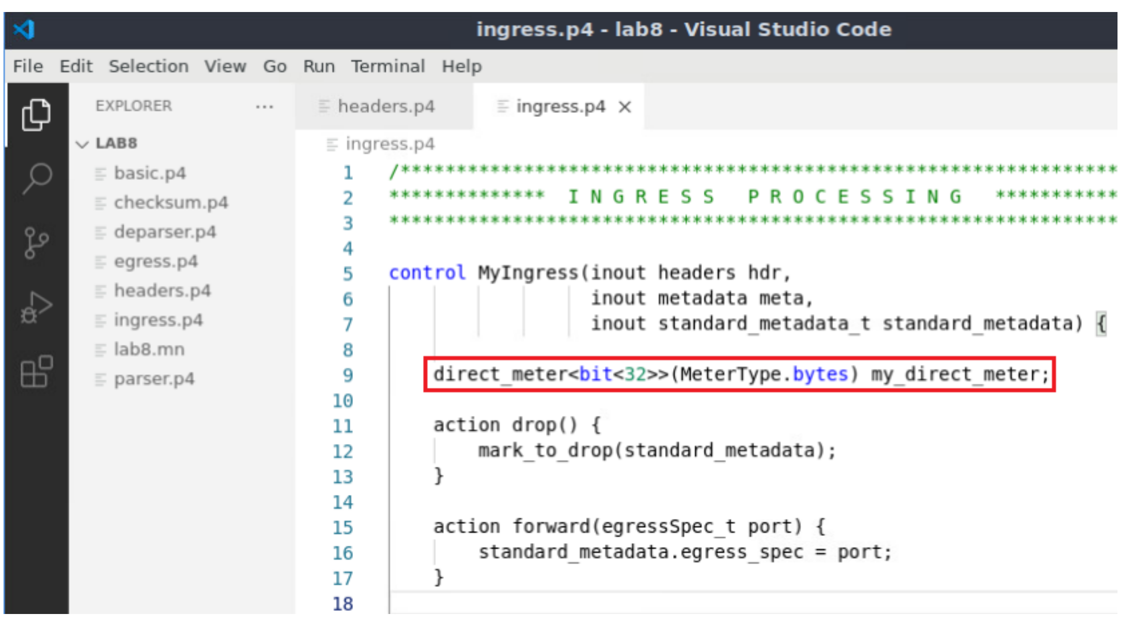


Figure 19. Declaring a direct meter.

Step 2. Refer `my_direct_meter` to the forwarding table by adding the following line of code.

```
meter = my_direct_meter;
```

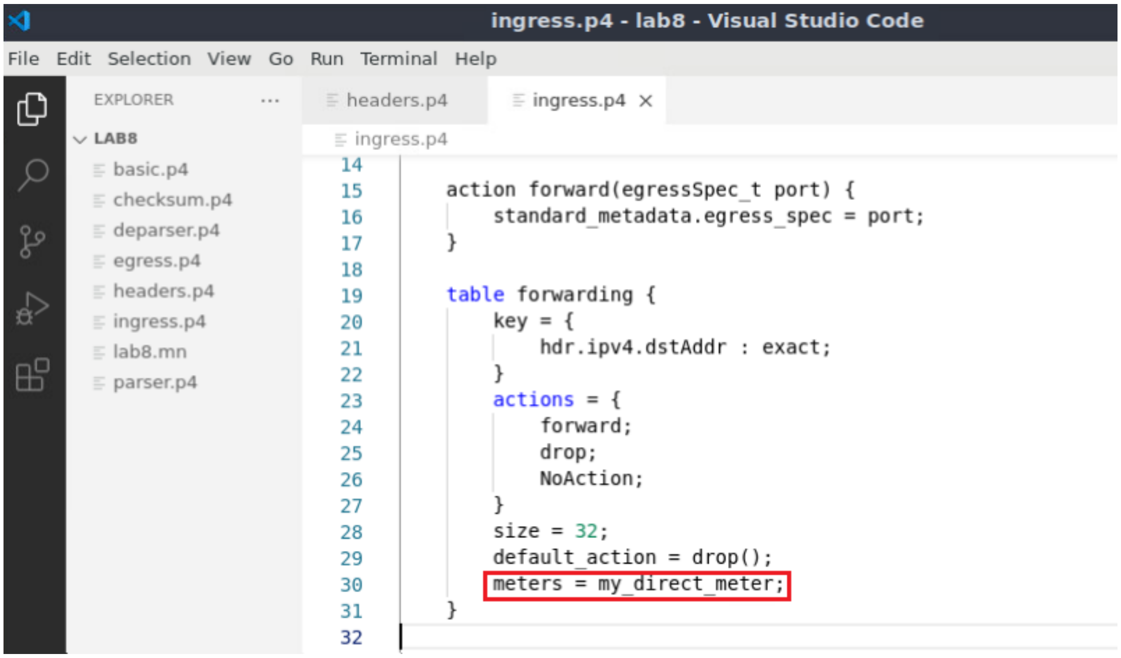


Figure 20. Referring a direct meter in the forwarding table.

Step 3. Add the following line of code to the forward action to read the color of the meter.

```
my_direct_meter.read(meta.meter_color);
```

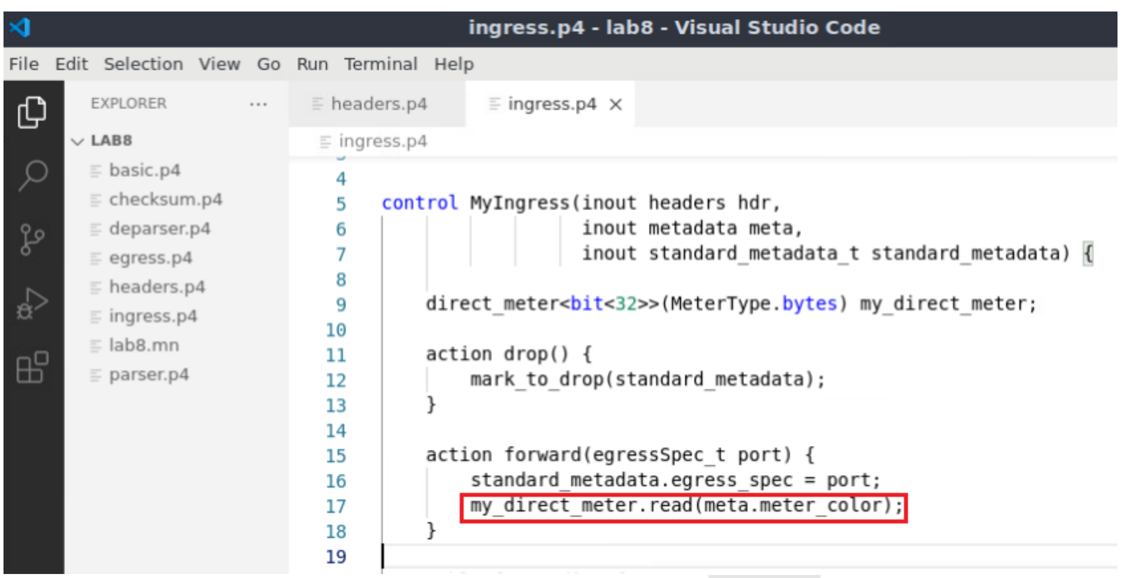


Figure 21. Reading the meter color in the `forward` action.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

3.5 Defining the rerouting table

Step 1. Define the table `rerouting` by adding the following piece of code.

```
table rerouting {
        key = {
                meta.meter_color : exact;
        }
        actions = {
                reroute;
                NoAction;
        }
        size = 32;
        default_action = NoAction;
}
```

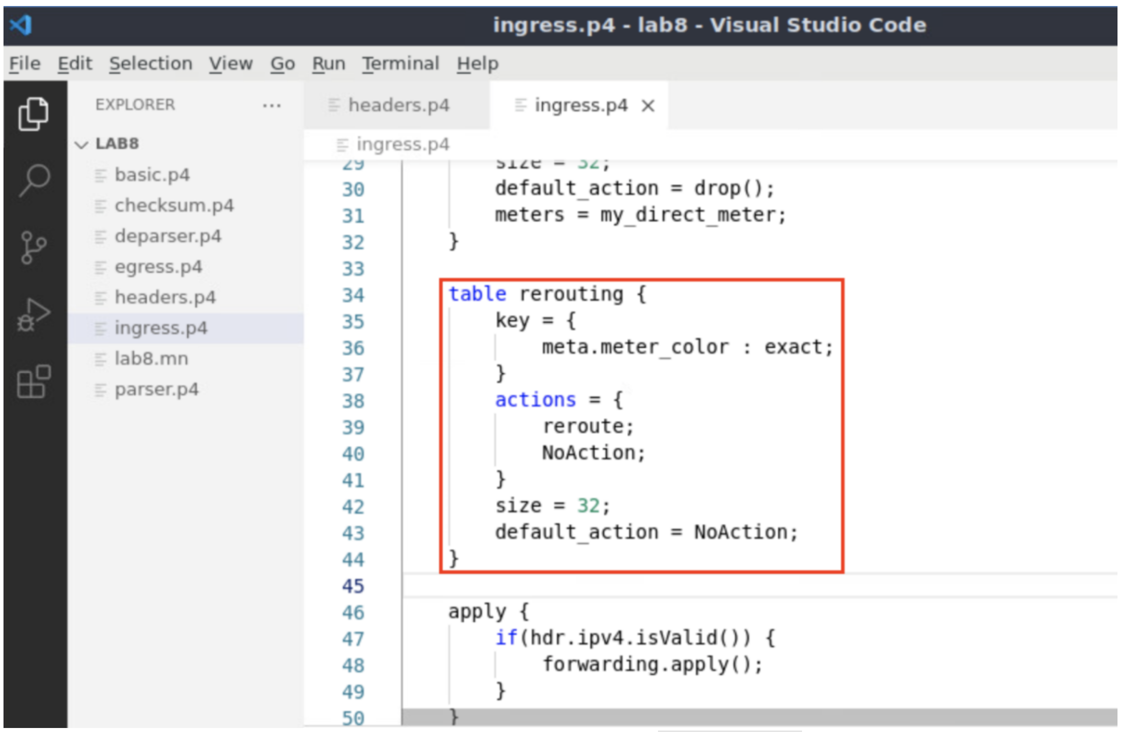


Figure 22. Defining the table `rerouting`.

The table defined in the figure above matches the destination meter's color. The actions in this table can be `reroute` or `NoAction`. Note that the table allows up to 32 entries (`size = 32`), and the default action is `NoAction`.

Step 2. Define the action `reroute` by adding the following piece of code. This action receives as a parameter a new destination port from the control plane.

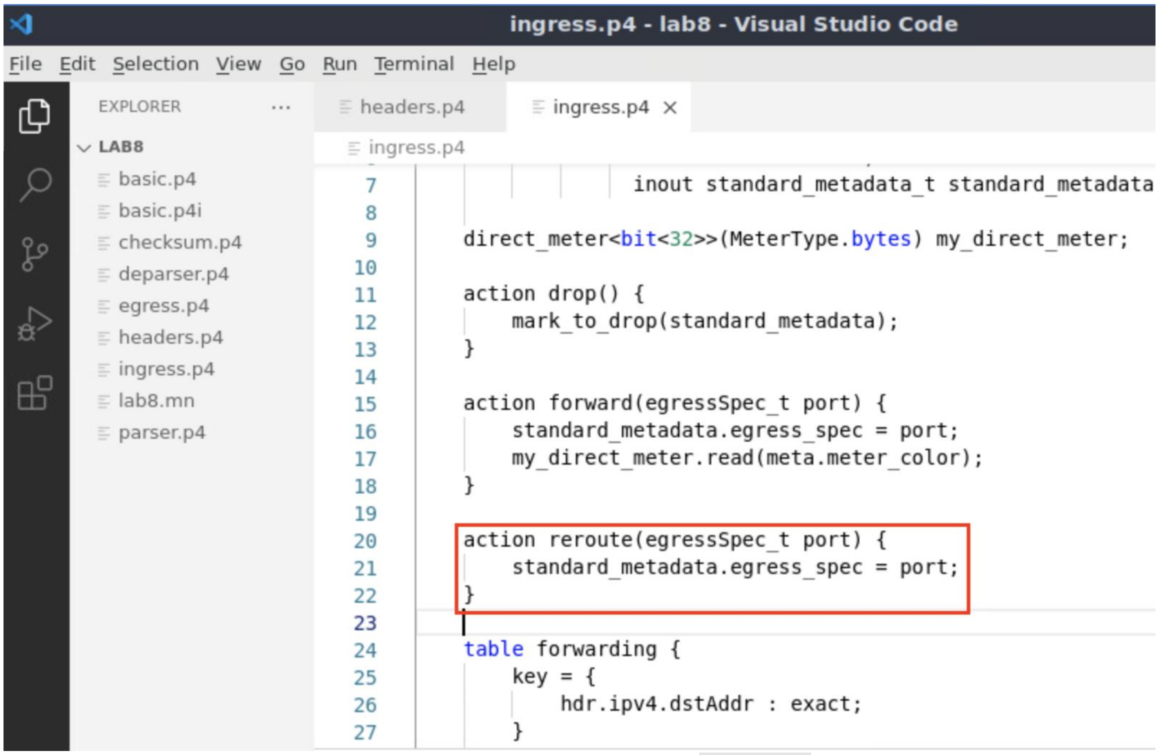


Figure 23. Defining the action `reroute`.

Step 3. Invoke the table `rerouting` in the apply block by adding the following line of code.

```
rerouting.apply();
```

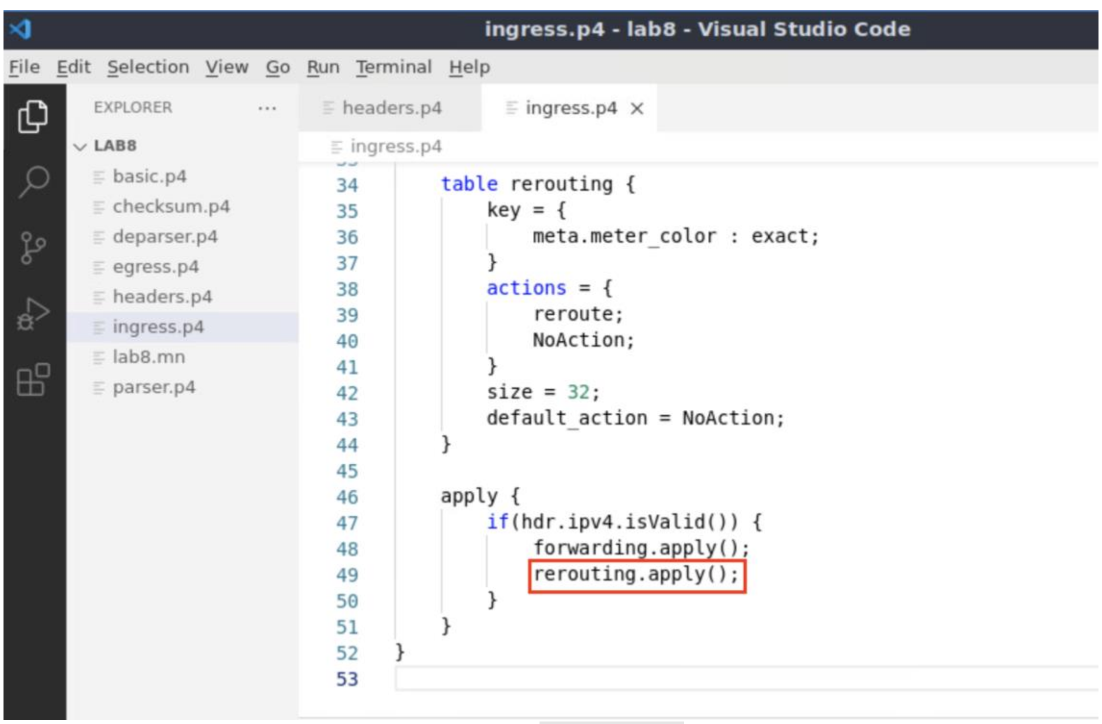


Figure 24. Invoking the table `rerouting` in the apply block.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

4 Loading the P4 program

In this section, you will compile and load the P4 binary into switch s1. You will also verify that the binary resides in switch s1 filesystem.

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside VS Code to compile the program.

```
p4c basic.p4
```

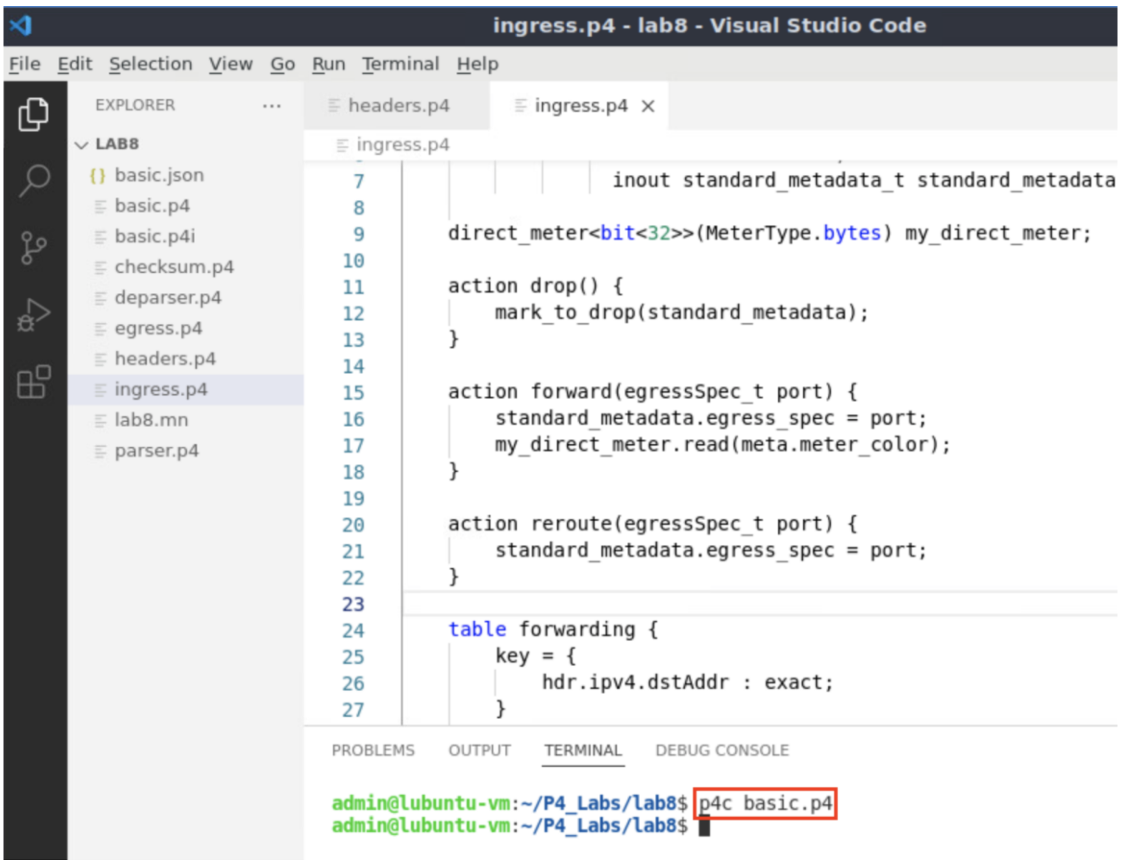


Figure 25. Compiling a P4 program.

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
```

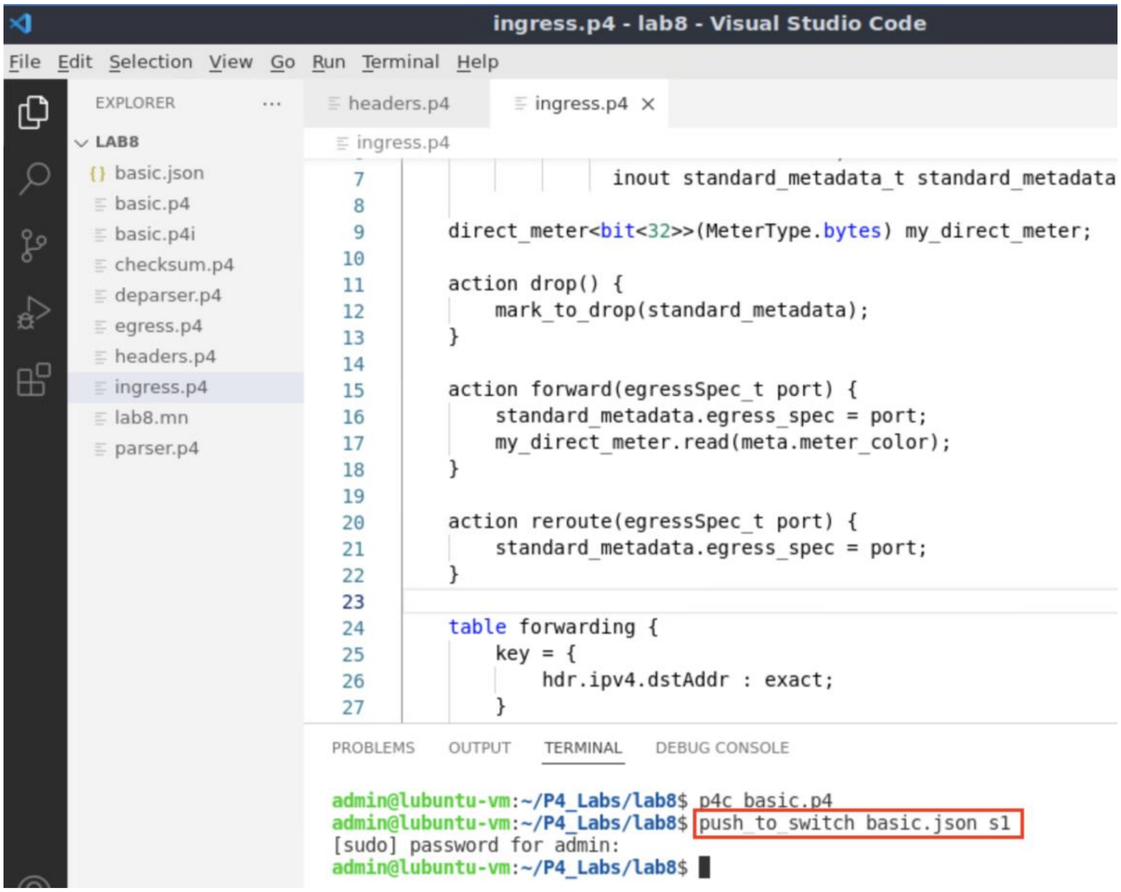


Figure 26. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

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

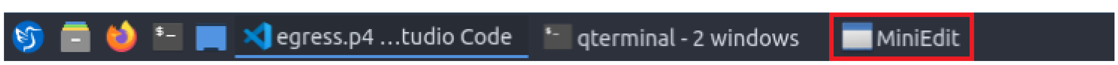


Figure 27. Maximizing the MiniEdit window.

Step 2. Right-click on the switch s1 icon and select *Terminal*.

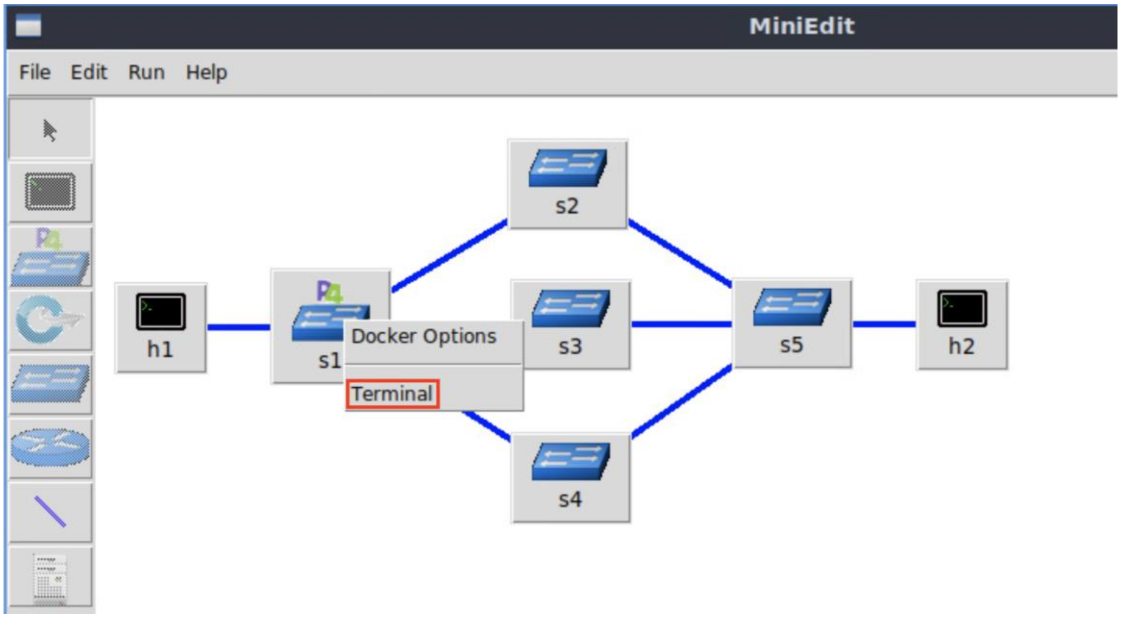


Figure 28. Starting the terminal on switch s1.

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 terminal.

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

```
ls
```

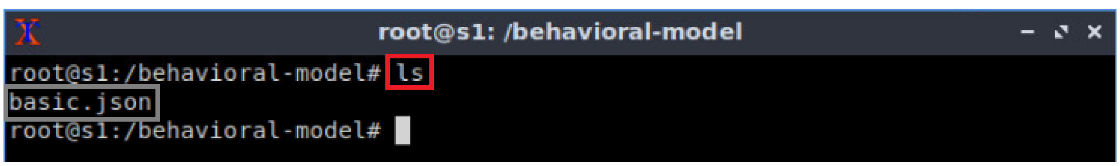


Figure 29. Displaying the contents of the current directory in the switch s1.

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

5 Configuring switch s1

In this section, you will map switch s1 interfaces to the ports in the P4 program and start the switch daemon. Note that the switch's logs are enabled to see the tables and actions that packets hit across the pipeline. Finally, you will load the rules to populate the table `forwarding`.

5.1 Running the switch's daemon and mapping the ports

Step 1. In switch s1 terminal, start the switch daemon and map the logical interfaces to Linux interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 -i 3@s1-eth3 basic.json &
```

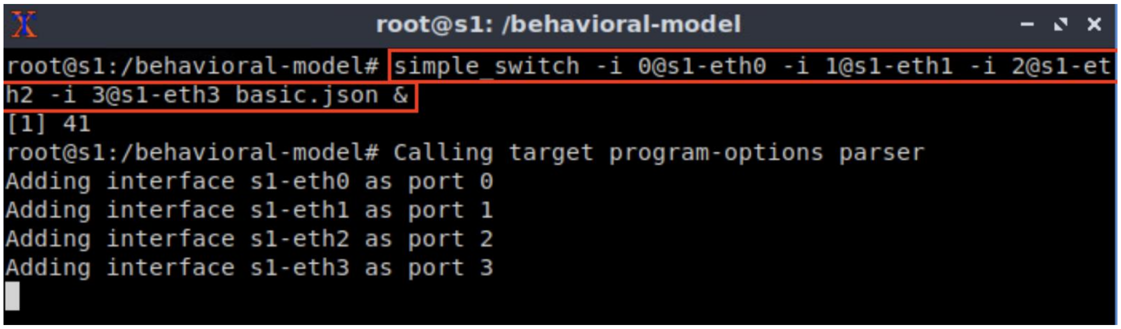


Figure 30. Starting switch s1 daemon and mapping the logical interfaces to Linux interfaces.

5.2 Loading the rules to the switch

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

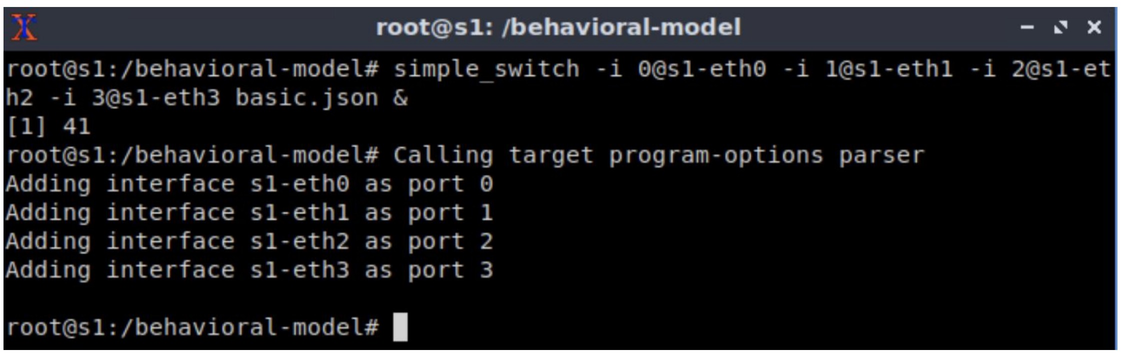


Figure 31. Returning to switch s1 CLI.

Step 2. Inspect the rules' file by issuing the following command.

```
cat ~/lab8/rules.cmd
```

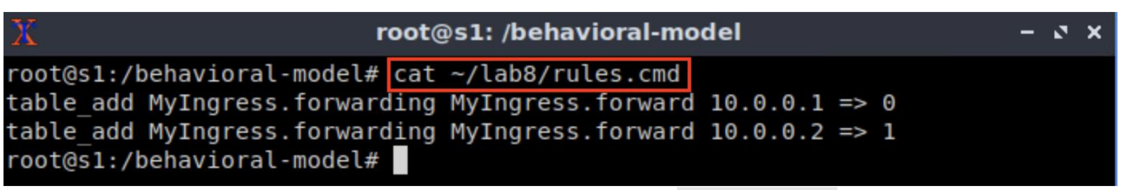


Figure 32. Inspecting the entries of the `forwarding` table.

Consider the first output in the figure above:

- `table_add`: adds an entry to a match-action table.
- `MyIngress.forwarding`: the name of the table.
- `MyIngress.forward`: the action.
- `10.0.0.1`: the key to the table.
- `0`: the egress port.

Step 3. Push the table entries to switch s1 by typing the following command.

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

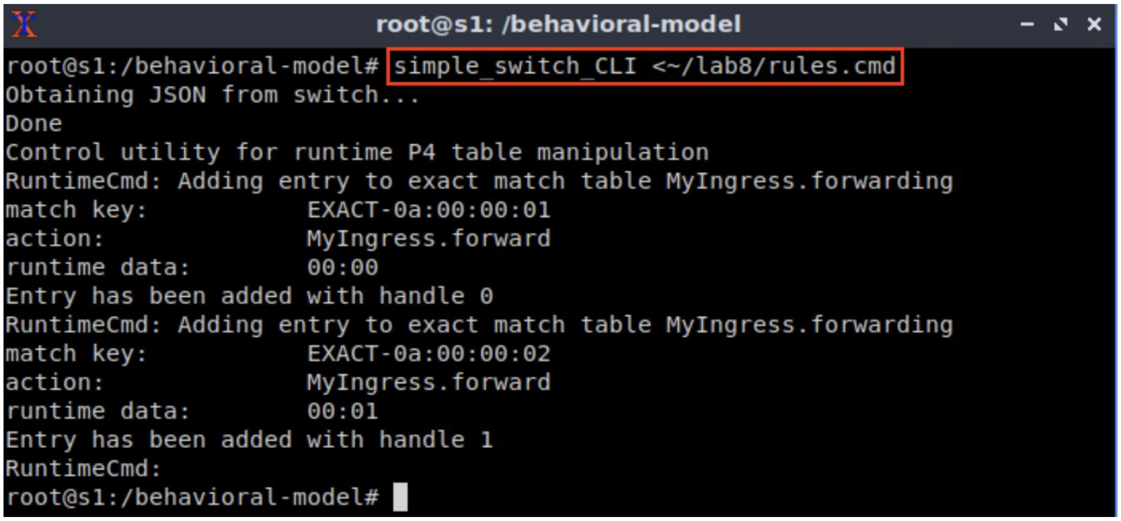


Figure 33. Populating the forwarding table into switch s1.

6 Testing and verifying the P4 program

In this section, you will run iperf3 tests between hosts to test the P4 program loaded to switch s1. You will set two rates of the meter to 100Mbps and 500Mbps. Then, you will populate the entries in the table `rerouting`, by matching the color of the meter. If the rate is below 100Mbps, the color is green (with value 0). If the color is between 100Mbps and 500Mbps, the color of the meter is yellow (with value 1). Finally, if the rate is over 500Mbps, the color of the meter is red (with value 2).

6.1 Running iperf3 tests between end hosts

Step 1. Open a terminal in host h2 and start an iperf3 server by issuing the following command.

```
iperf3 -s
```

Figure 34. Starting an iperf3 server in host h2.

Step 2. Go back to host h1 terminal and start an iperf3 client by issuing the following command.

```
iperf3 -c 10.0.0.2
```

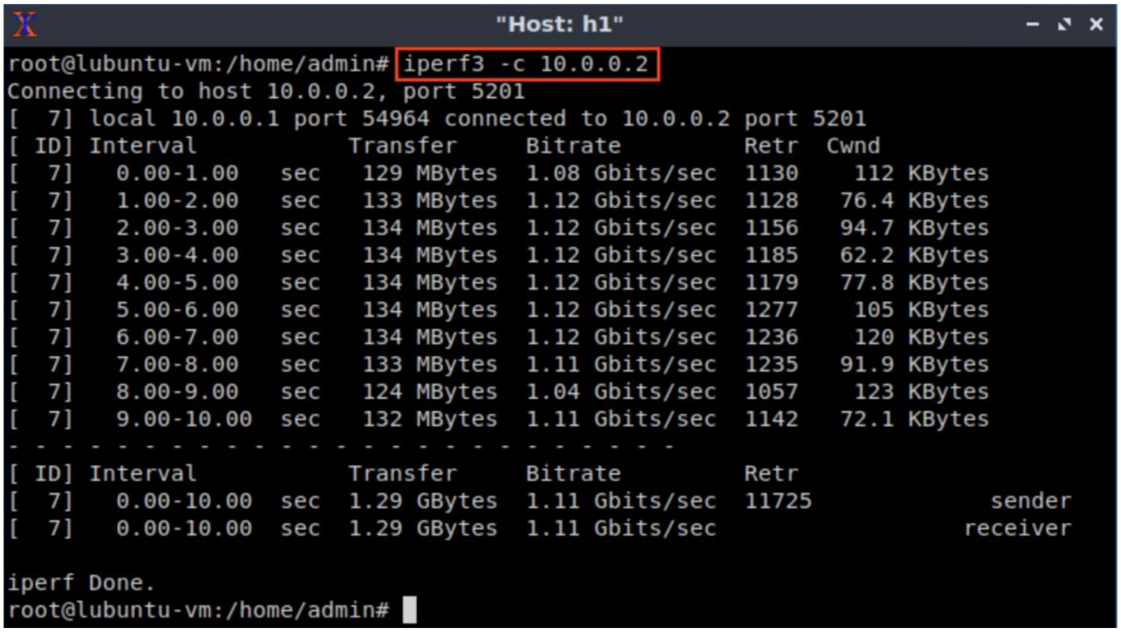


Figure 35. Running an iperf3 test between host h1 and host h2.

The figure above shows that the maximum throughput is around 1.11 Gbps.

6.2 Setting the meter's rate

Step 1. Go back to the switch's terminal and start the CLI by issuing the following command.

```
simple_switch_CLI
```

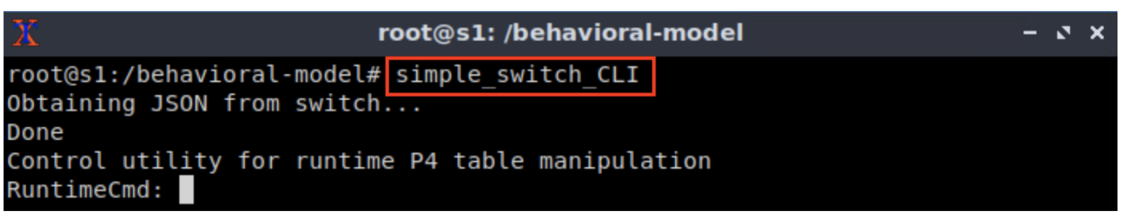


Figure 36. Starting the switch's CLI.

Step 2. Set the meter's rate by issuing the following command.

```
meter_set_rates MyIngress.mydirect_meter 1 12:400000 63:2000000
```

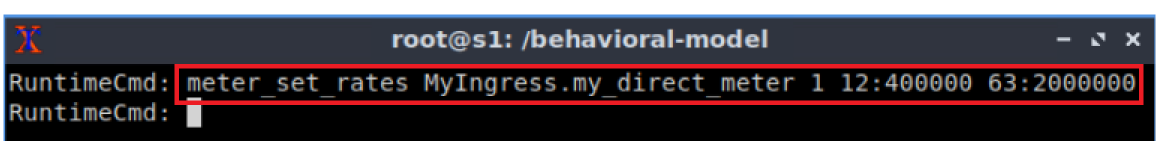


Figure 37. Setting the meter's rate.

The command and its parameters in the figure above are explained as follows:

- `meter_set_rates`: command to set the meter's rate.
- `MyIngress.my_direct_meter`: the name of the meter.
- `1`: the index of the meter. Note that this index is associated with the second entry in the `forwarding` table.
- `12:400000`: the rate (in packets/microseconds) and the burst values in bytes/second.
- `63:2000000`: the rate (in packets/microseconds) and the burst values in bytes/second.

Note that 12 packets per microsecond correspond to ~100Mbps and 63 packets per microsecond correspond to ~500Mbps. The burst values result from dividing the by 250 the rate in bytes. For example, 100Mbytes/second divided by 250 is 400,000, establishing the maximum burst size. Note that 250 specifies the bytes/Hz, where Hz is the system's clock rate.

6.3 Populating the rerouting table

Step 1. Add the following entry to the table `rerouting` by issuing the following command.

```
table_add MyIngress.rerouting MyIngress.reroute 1 => 2
```

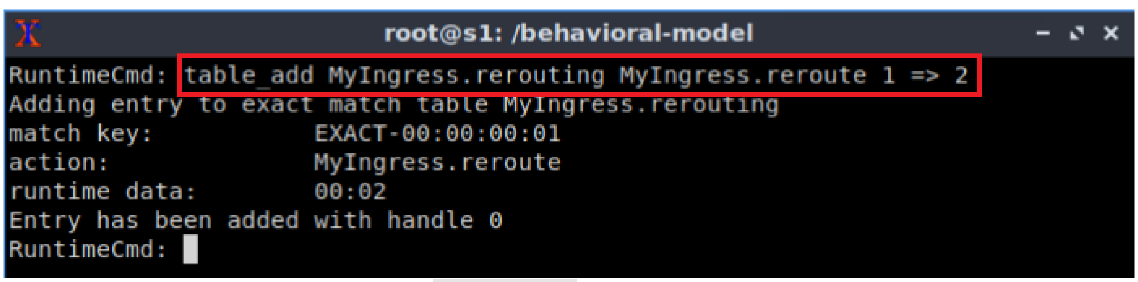


Figure 38. Populating the table `rerouting` with the value of the yellow color (i.e., 1).

Note the entry in the figure above will match the yellow color (with value 1) to reroute traffic through port 2.

Step 2. Similarly, add the second entry to the table `rerouting` by issuing the following command.

```
table_add MyIngress.rerouting MyIngress.reroute 2 => 3
```

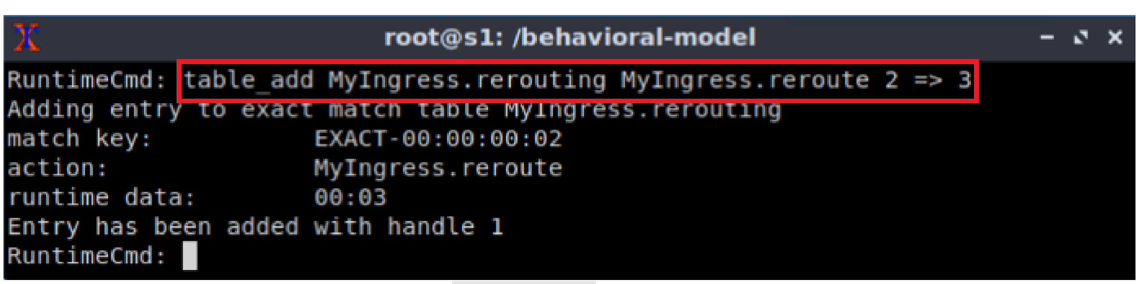


Figure 39. Populating the table `rerouting` with the value of the red color (i.e., 2).

Note the entry in the figure above will match the red color (with value 2) to reroute traffic through port 3.

6.4 Verifying the meter rate

Step 1. Go back to the Linux terminal by clicking the icon in the taskbar.

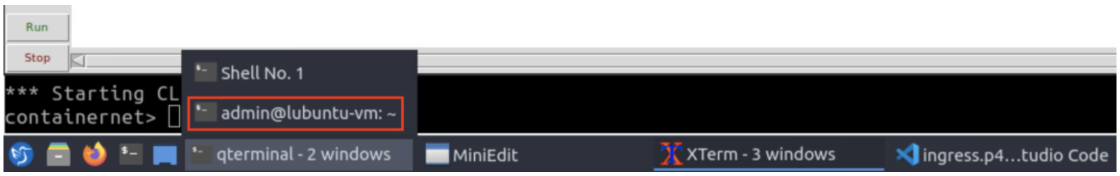


Figure 40. Opening the Linux terminal.

Step 2. Start the `nload` monitoring tool by issuing the command below. Note that `the` `-m` parameter specifies multiple interfaces. In this case, the tool is monitoring the rate of the egress ports of switches s2, s3, and s4.

```
nload -m s2-eth2 s3-eth2 s4-eth2
```

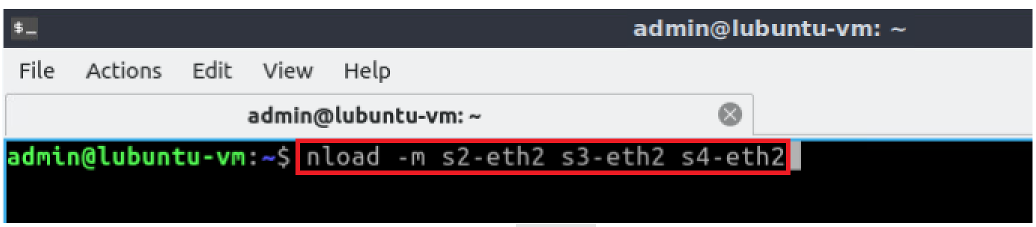


Figure 41. Running the `nload` monitoring tool.

Step 3. Go back to host h1 terminal and issue the following command.

```
iperf3 -c 10.0.0.2 -u -b 50mbit -t 300
```

Figure 42. Running an iperf3 test between h1 and h2 at 50Mbps.

Step 4. Go back to the Linux terminal and observe the rate at interface *s2-eth2*.

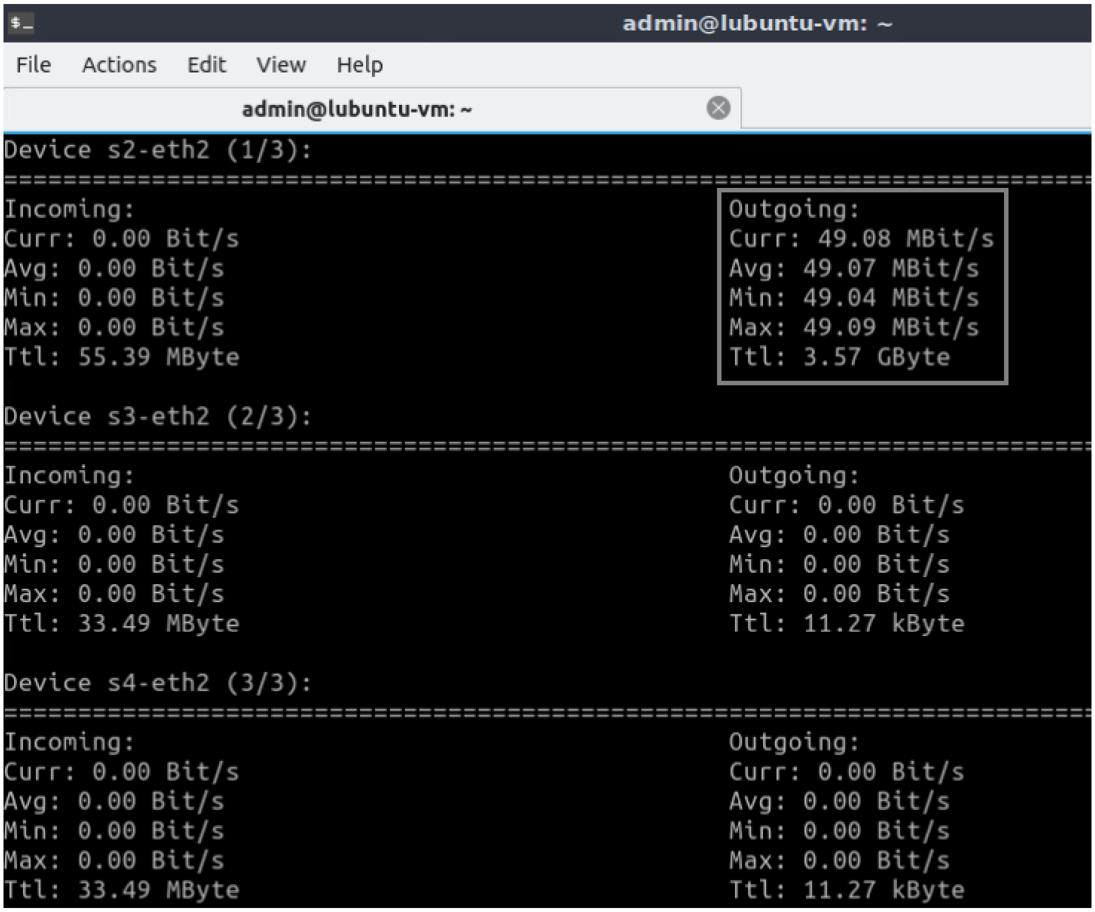


Figure 43. Inspecting the sending rate on the interface *s2-eth2*.

Step 5. Go back to host h1 terminal and stop the iperf3 test by pressing `Ctrl+c`.

Step 6. Start another iperf3 test by issuing the command below. Note that this time the rate is set to 300Mbps.

```
iperf3 -c 10.0.0.2 -u -b 300mbit -t 300
```

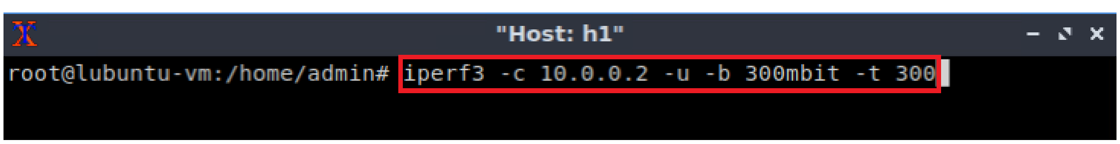


Figure 44. Running an iperf3 test between h1 and h2 at 300Mbps.

Step 7. Go back to the Linux terminal and observe the rate at interface *s3-eth2*.

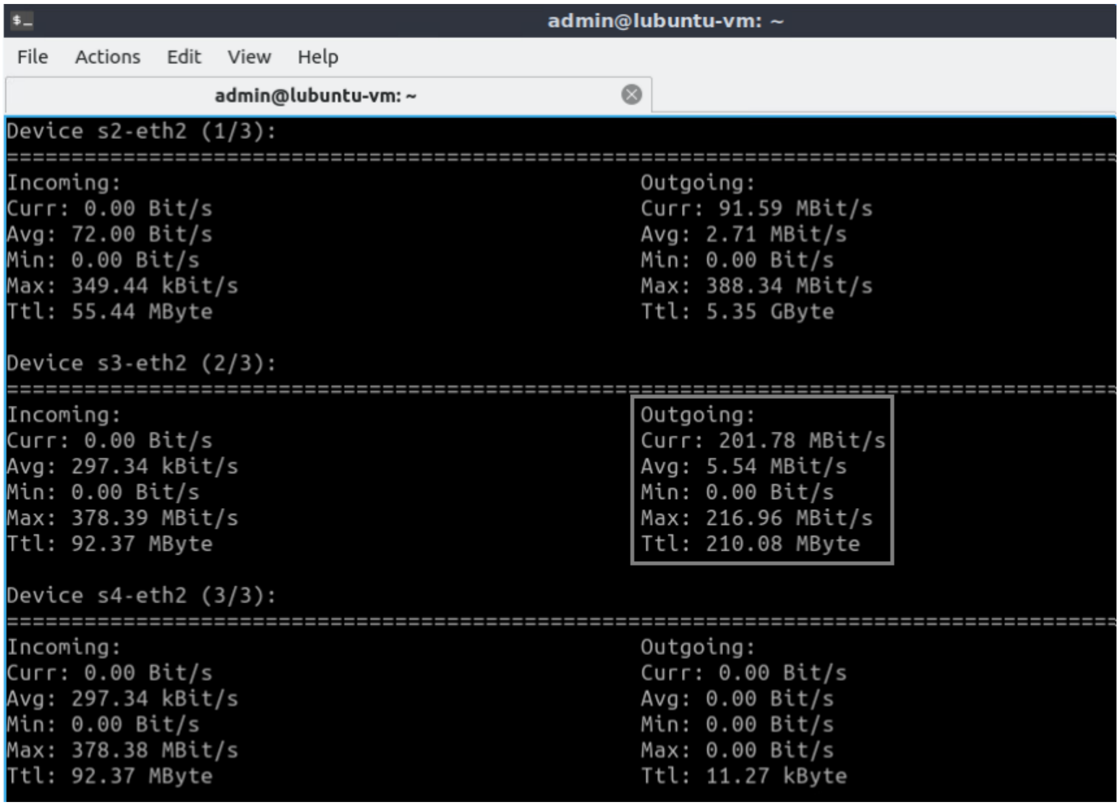


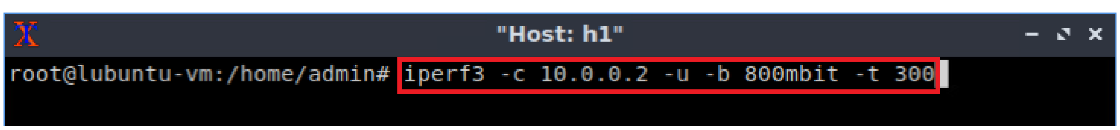
Figure 45. Inspecting the sending rate on the interface *s3-eth2*.

Note that part of throughput is shared with the interface *s2-eth2*.

Step 8. Go back to host h1 terminal and stop the iperf3 test by pressing `Ctrl+c`.

Step 9. Start another iperf3 test by issuing the command below. Note that this time the rate is set to 800Mbps.

```
iperf3 -c 10.0.0.2 -u -b 800mbit -t 300
```



Running an iperf3 test between h1 and h2 at 800Mbps.

Step 10. Go back to the Linux terminal and observe the rate at interface *s4-eth2*.

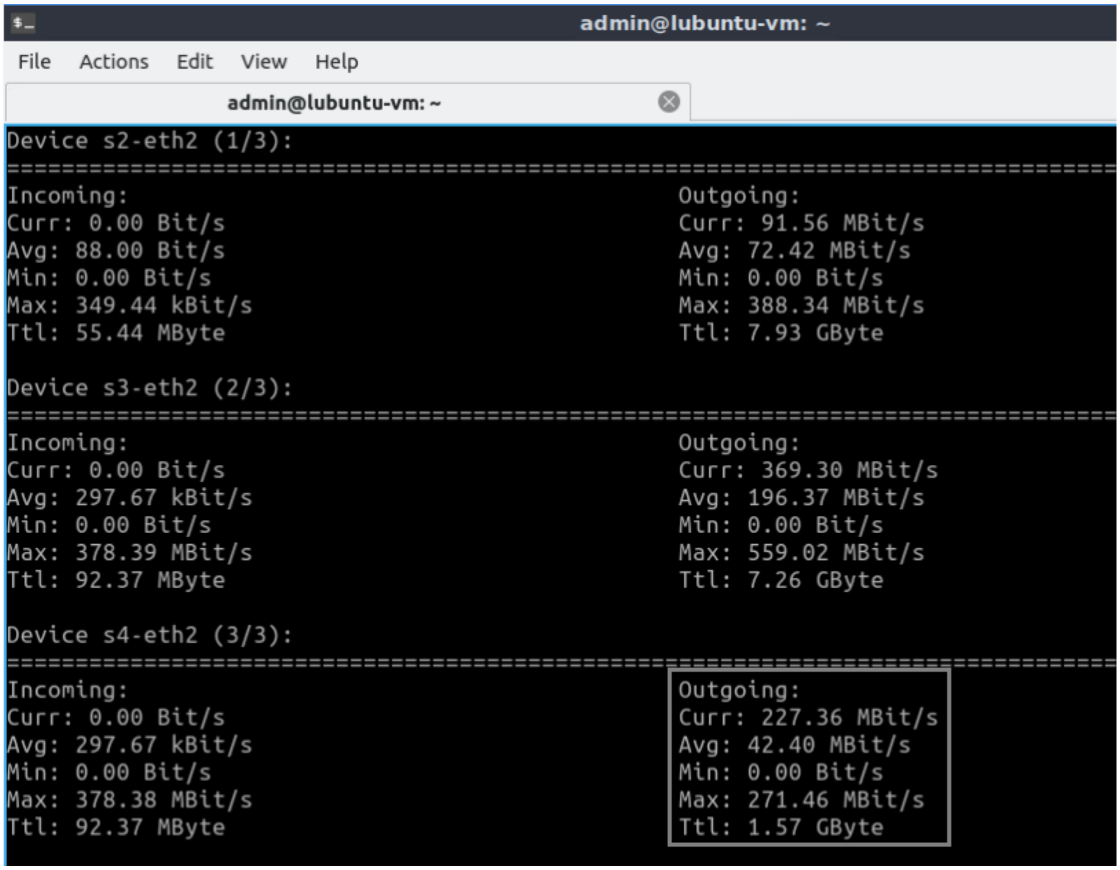


Figure 46. Inspecting the sending rate on the interface *s4-eth2*.

Note that part of throughput is shared with the interfaces *s2-eth2 and s3-eth2*.

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

References

1. H. Juha, R. Guerin. "*RFC2698: A two rate three color marker.*" 1999.
2. The P4 language Consortium. "*The V1Model*." [Online]. Available: https://tinyurl.com/bdzfarvy
3. The P4 language Consortium. "*Runtime CLI specification*."[Online]. Available: https://tinyurl.com/49n4wyrt
4. Mininet walkthrough. [Online]. Available: http://Mininet.org.
5. 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.
6. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
7. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 9: Storing Arbitrary Data using Registers

Document Version: 05-12-2022

Contents

Overview

Programmable data planes use registers to store arbitrary information that can be accessed by multiple packets traversing the switch. This lab describes how to use registers by showing the user the steps to create a P4 program that stores the IP address of the last flow. Moreover, it presents the control plane commands to read, write, and clear values of a register. Registers can be read and written from both the control and the data planes.

Objectives

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

1. Declare a single cell register in a P4 program.
2. Store an arbitrary value into a register.
3. Interact with registers from the control plane and the data plane.

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 single cell register.
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 P4 registers

The P4 language provides registers to save arbitrary data. Registers are stateful elements used to store values longer than the time it takes to process a packet. This feature allows the creation of P4 programs where multiple packets can access registers. Registers in P4 are organized into named arrays of cells. These cells are referred to by an index that defines a value's location. Registers can be read and written by both the control and the data plane. In P4, registers are global memory resources meaning that any match-action tables can reference them.

1.1 Declaring and using a single cell register

The syntax below shows how to declare a single cell register in P4. Register R1 contains a single cell that stores a value of `N` bits.

```
register<bit<N>>(1) R1;
```

The functions `write` and `read` are used to store and retrieve values from a register. For example, the programmer invokes the following function to store the value `val` in register R1.

```
R1.write(0,val)
```

Similarly, the user invokes the function below to read a value stored in R1. Note that the retrieved value is stored in the variable `res`.

```
R1.read(res,0)
```

1.2 Lab scenario

This lab shows the steps to create a P4 program that stores the IP address of the last flow. The user will access the register's values from the control plane. Moreover, the user will read, write, and reset a register from the control plane.

2 Lab topology

Let us get started by loading a simple Mininet topology using MiniEdit. The topology comprises four end hosts, a P4 programmable switch, and a legacy switch.

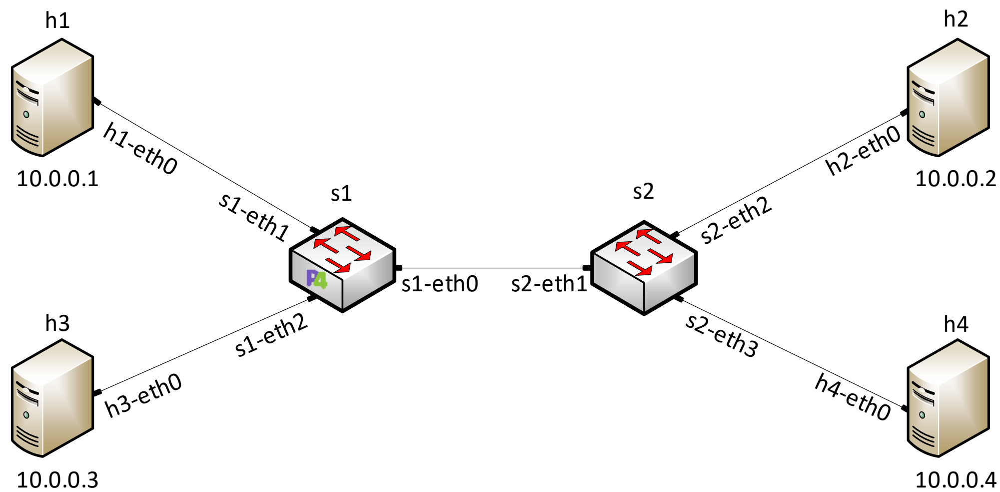


Figure 1. Lab topology.

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`.

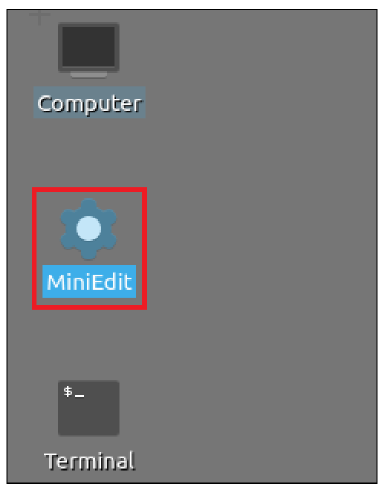


Figure 2. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab9* folder and search for the topology file called *lab9.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

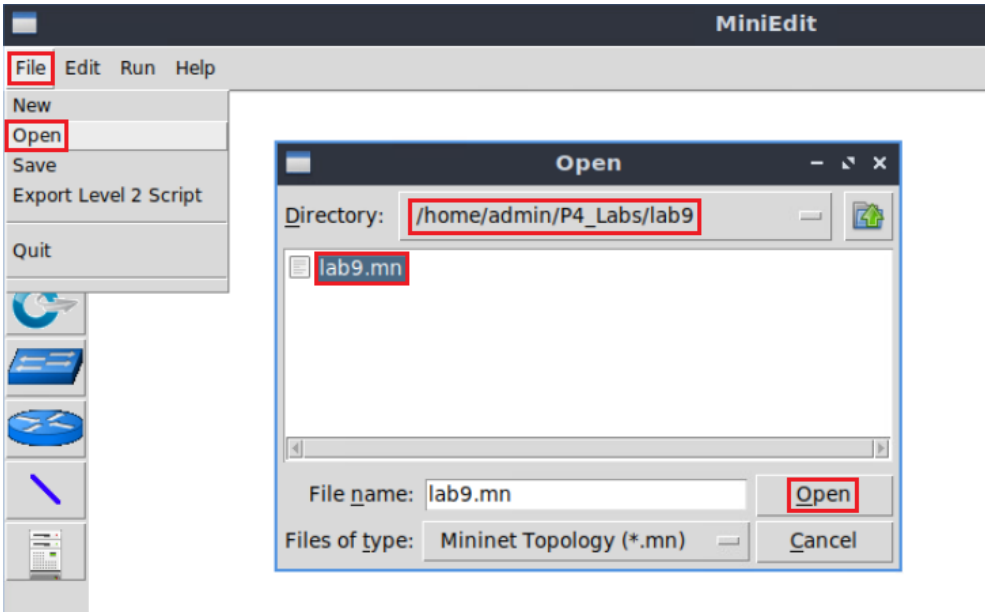


Figure 3. MiniEdit's *Open* dialog.

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 4. 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.

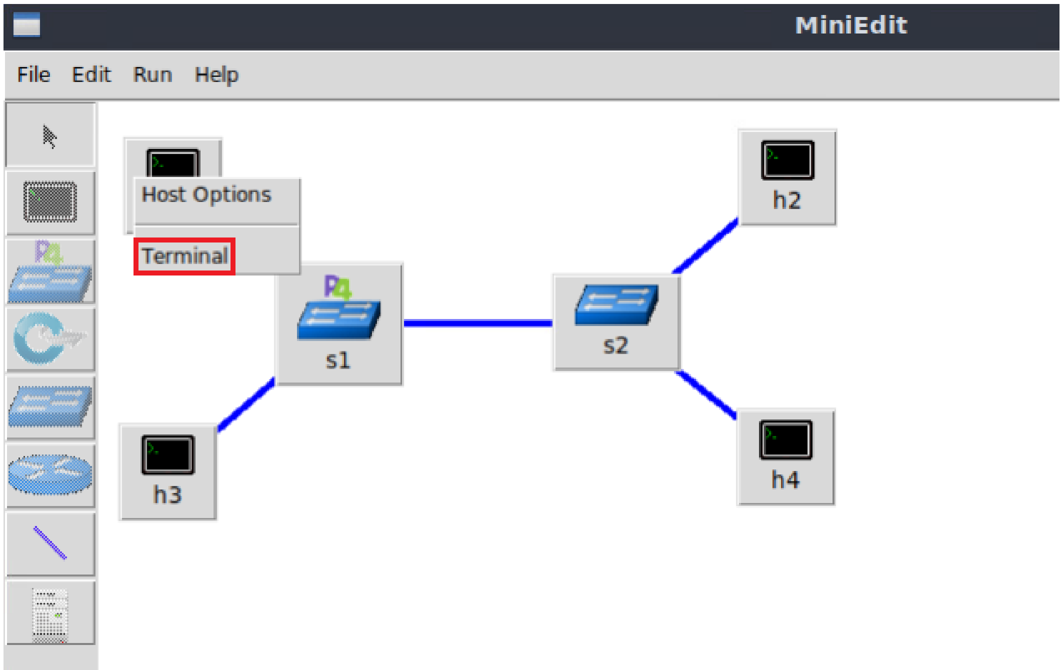


Figure 5. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

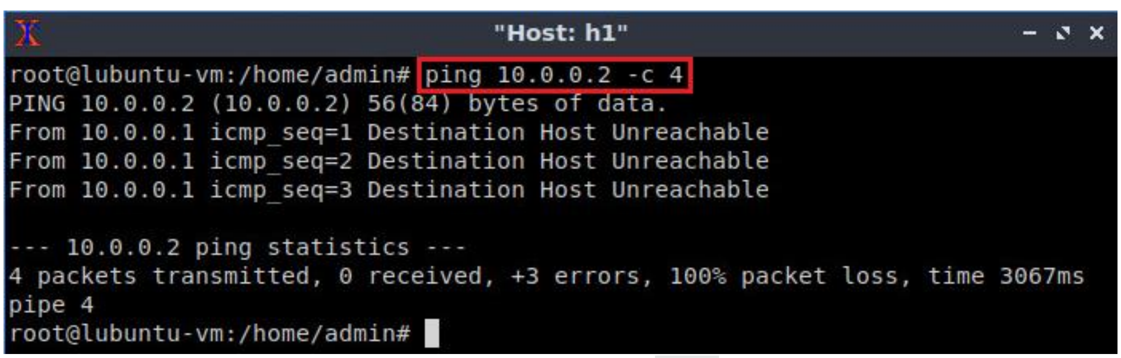


Figure 6. Connectivity test using `ping` command.

The figure above shows unsuccessful connectivity between host h1 and host h2. This result happens because there is no P4 program loaded on the switch.

3 Defining a single cell register

In this section, you will load the programming environment and define a single cell register in the ingress pipeline. This register will store the last destination IP address of any flow. Then, you will define an action that performs the storing operation. This action is invoked in the `apply` block.

3.1 Loading the programming environment

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

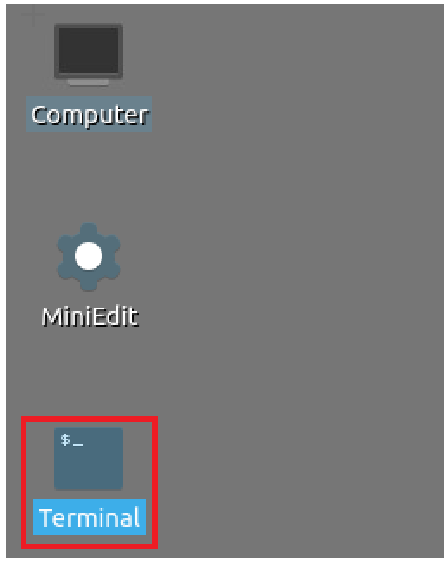


Figure 7. Shortcut to open a Linux terminal.

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/lab9
```

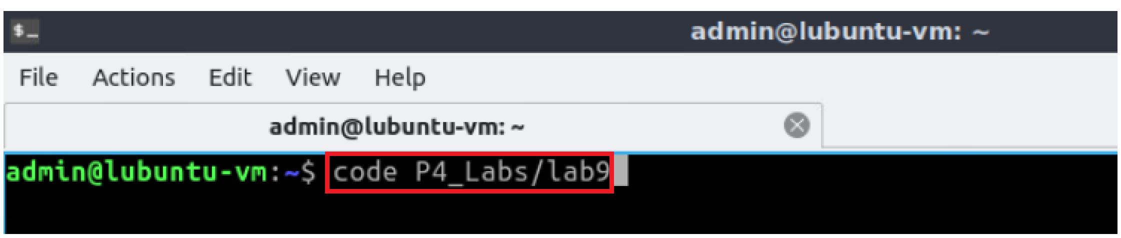


Figure 8. Loading the development environment.

3.2 Defining a register in the ingress pipeline

Step 1. Click on the in*gress.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.

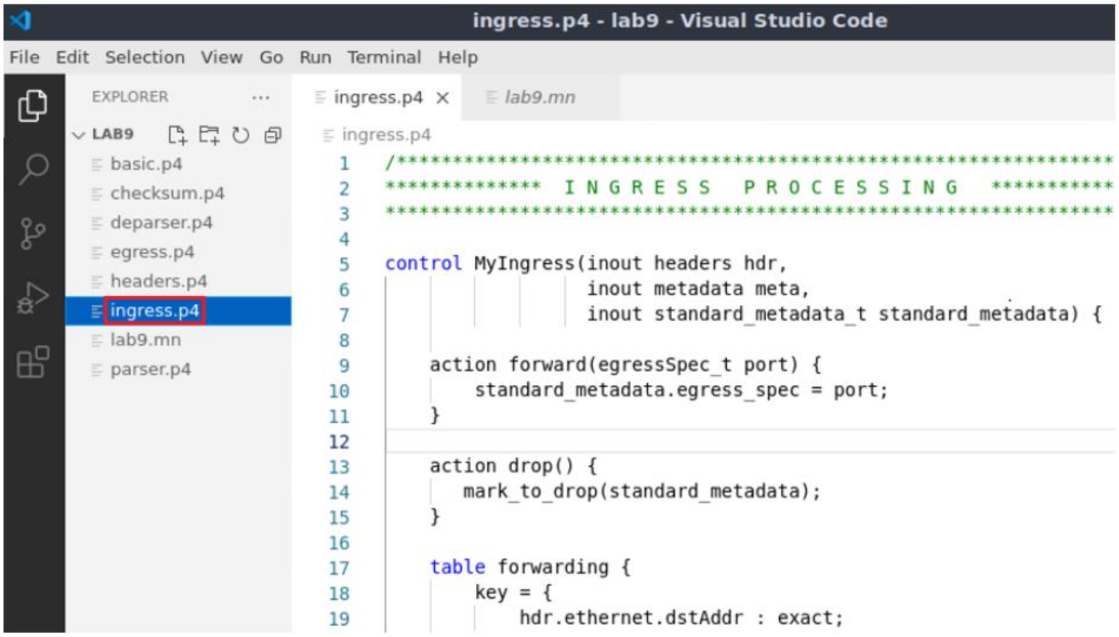


Figure 9. Inspecting the ingress processing block.

Step 2. Add the following line of code in the *ingress.p4* file to declare a local variable that we will use to store the flow identifier.

```
register<bit<32>>(1) last_src_IP;
```

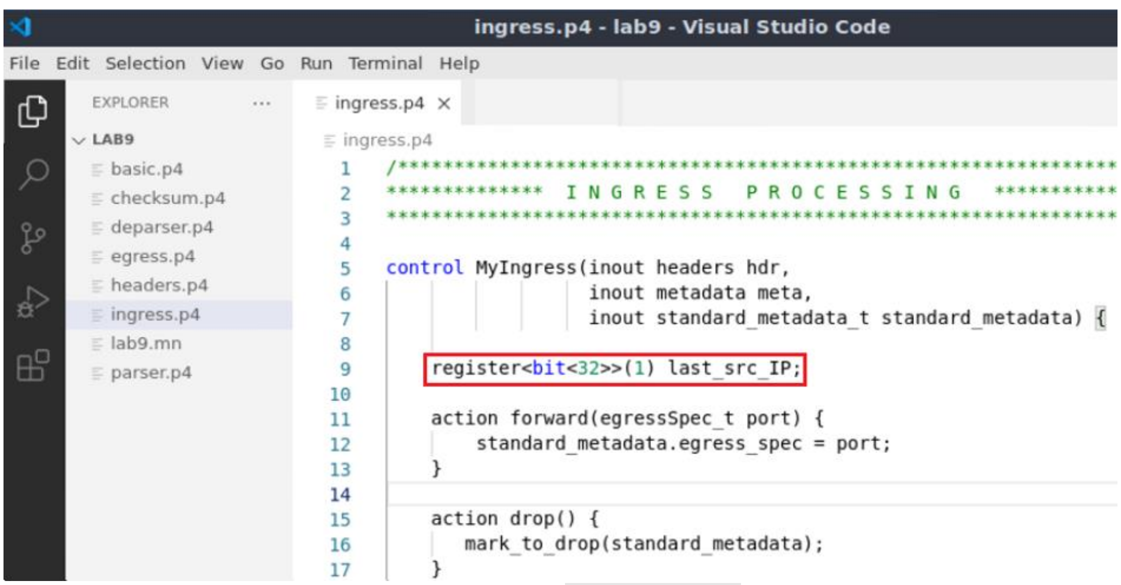


Figure 10. Declaring the `last_src_IP` register.

The statement above creates a 32-bit register with a single cell to store an IP address. The register name is `last_src_IP`. This register will be used to record the source IP of the last packet that traversed the switch.

Step 3. Define the following action by adding the piece of code shown below.

```
action update_last_src_IP() {
      last_src_IP.write(0, hdr.ipv4.srcAddr);
```

```
}
```

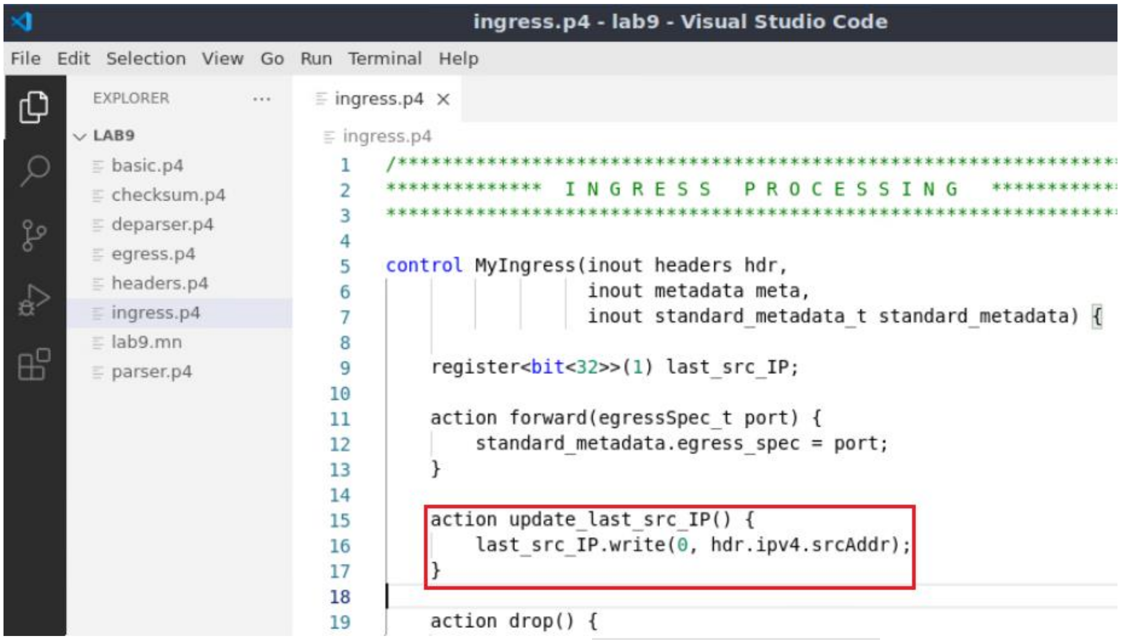


Figure 11. Defining the action `update_last_sct_IP`.

Since we are using a single cell register, we will write the value of the source IP to the cell indexed 0.

Step 4. Define the packet processing sequence by adding the following line of code inside the apply block.

```
update_last_src_IP();
```

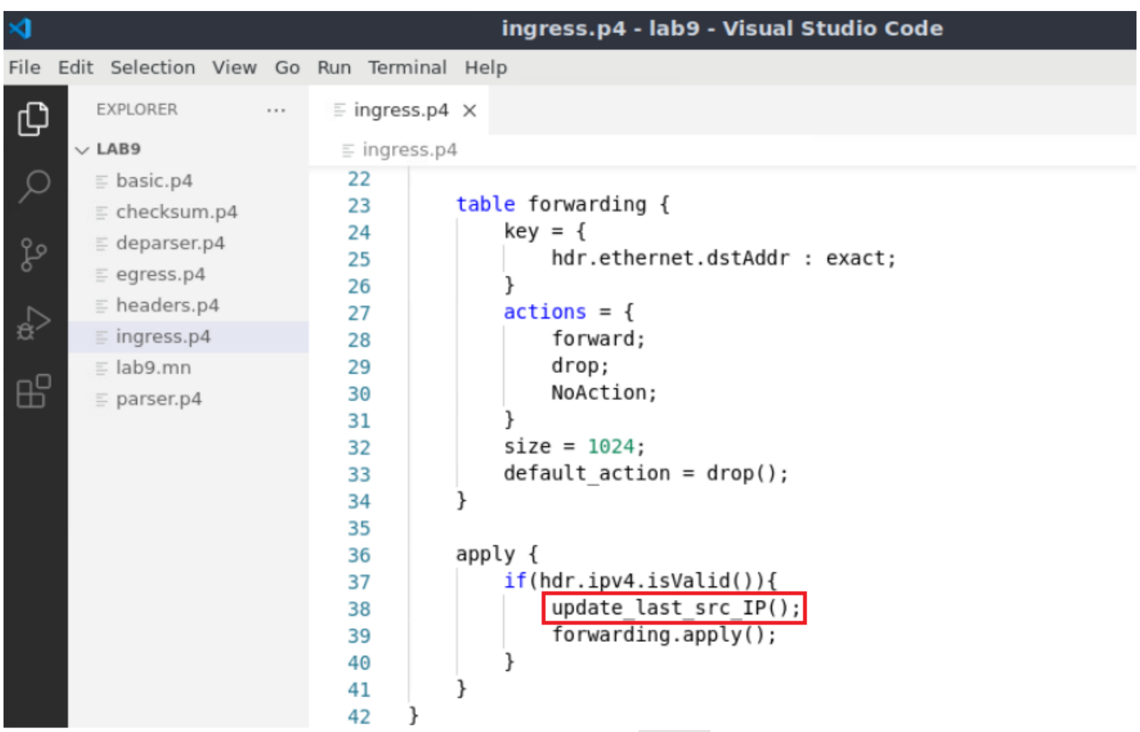


Figure 12. Defining the `apply` logic.

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

4 Loading the P4 program

In this section, you will compile and load the P4 binary into switch s1. You will also verify that the binary resides in switch s1 filesystem.

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
```

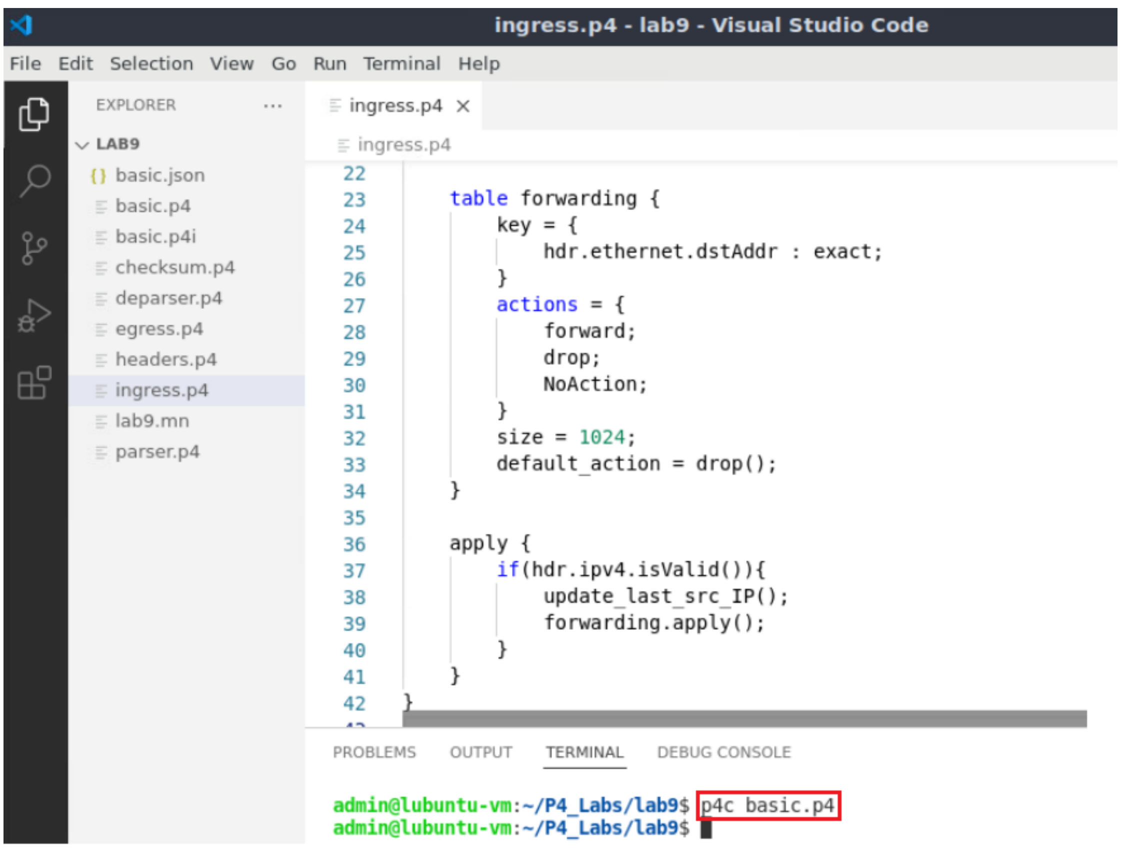


Figure 13. Compiling a P4 program.

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
```

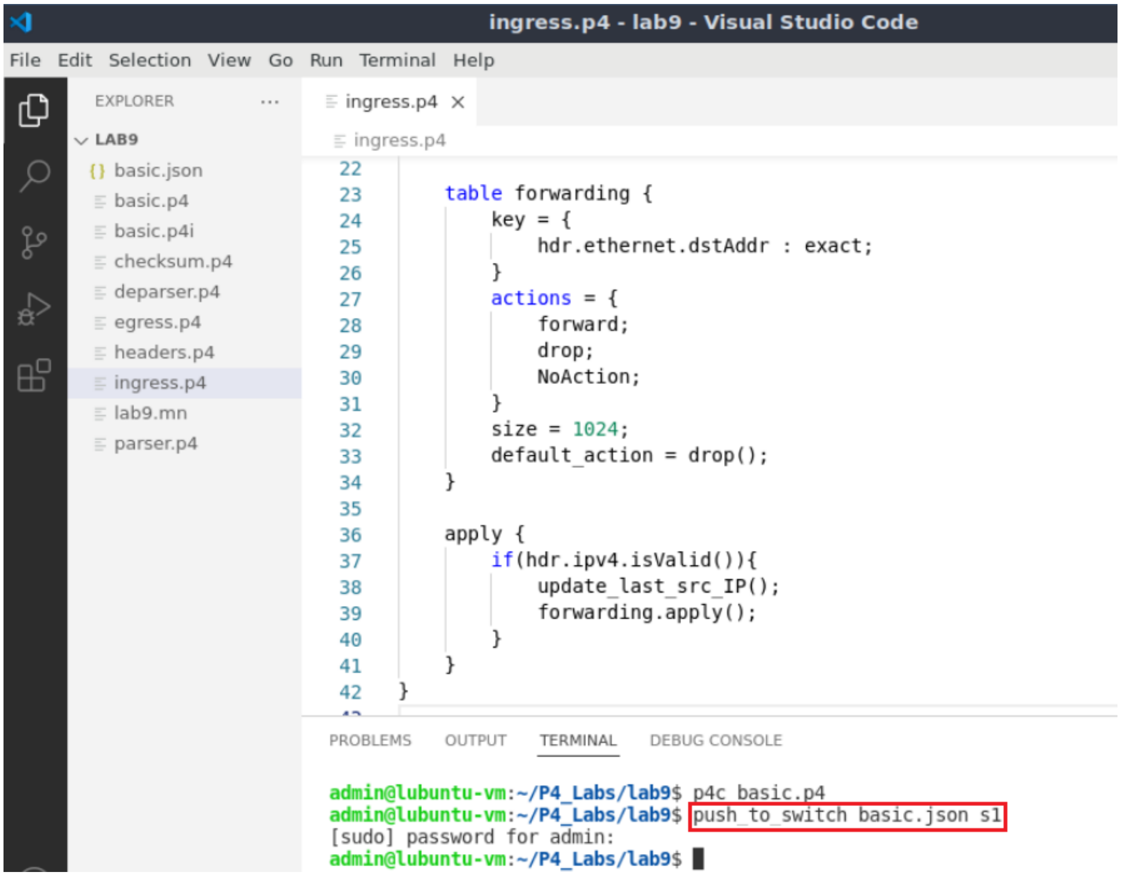


Figure 14. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

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

Figure 15. Maximizing the MiniEdit window.

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

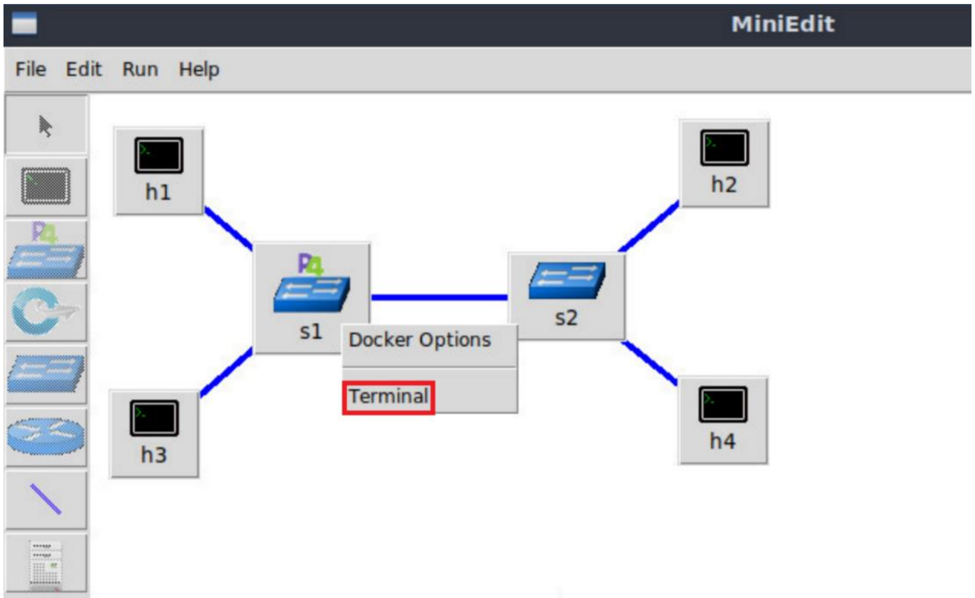


Figure 16. Starting the terminal on the switch.

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
```

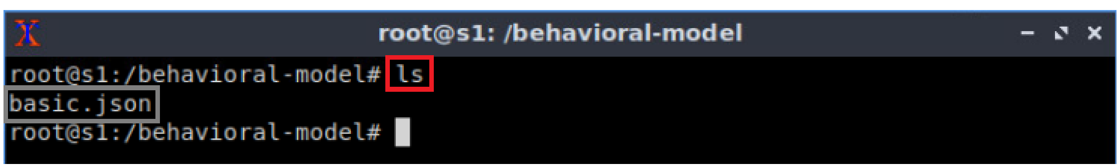


Figure 17. Displaying the contents of the current directory in the switch s1.

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

5 Configuring switch s1

In this section, you will map switch s1 interfaces to the ports in the P4 program and start the switch daemon. Then, you will load the rules to populate the match action tables.

5.1 Mapping P4 program's ports

Step 1. Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```

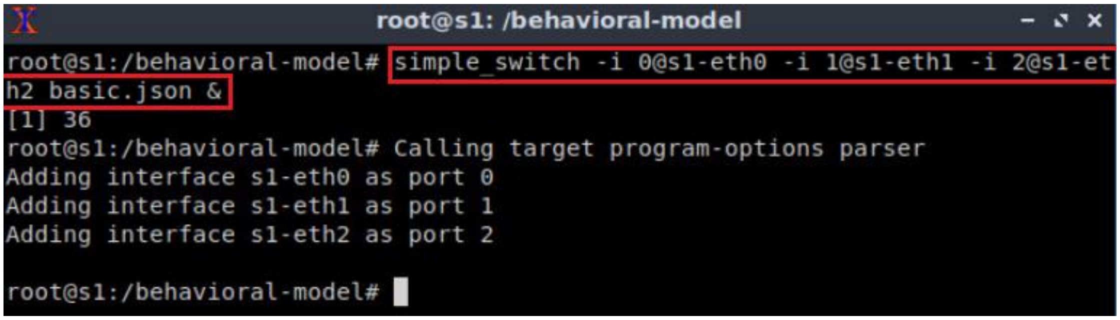


Figure 18. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

5.2 Loading the rules to the switch

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

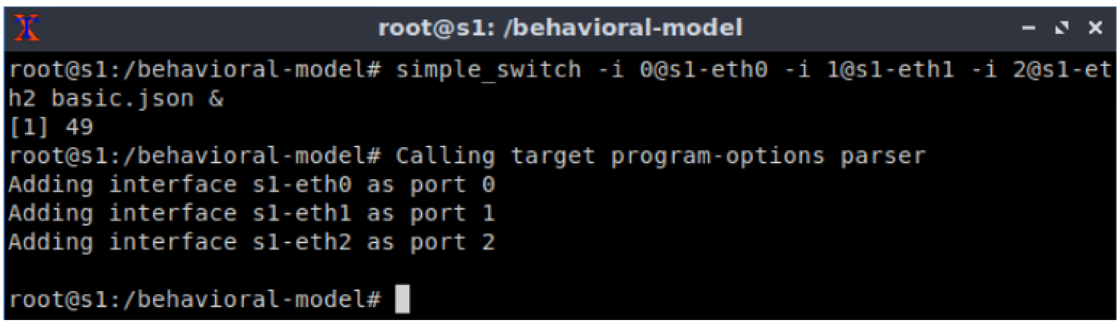


Figure 19. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/lab9/rules.cmd
```

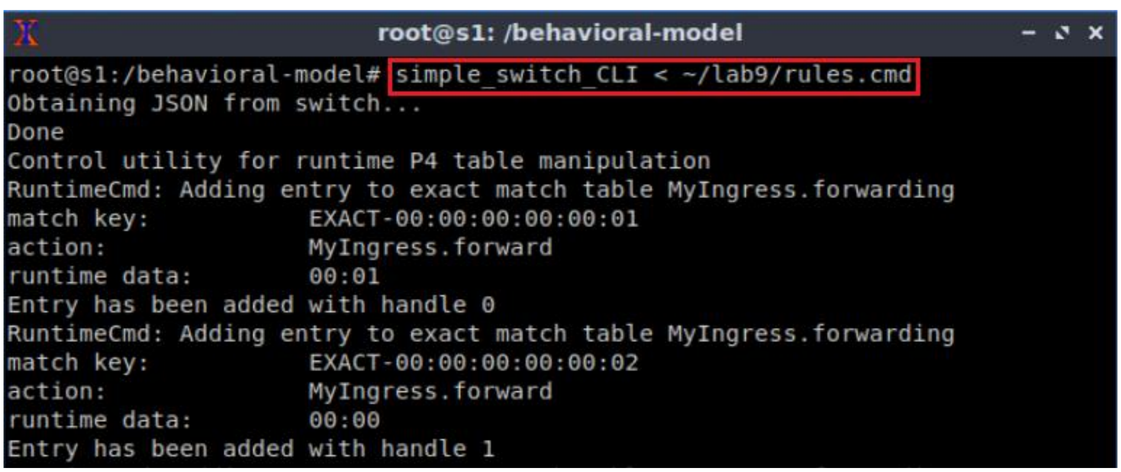


Figure 20. Populating the forwarding table into switch s1.

The script above pushes the rules into the match-action table `forwarding`.

6 Testing and verifying the P4 program

6.1 Sending a custom packet from host h1 to host h2

Step 1. Go back to MiniEdit and open a terminal on host h2's terminal. Issue the following command so that, host h2 starts listening for packets.

```
recv.py
```

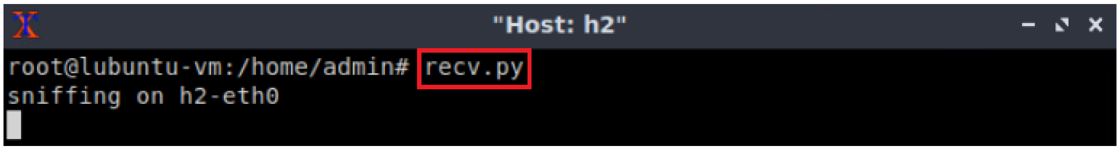


Figure 21. Listening for incoming packets in host h2.

Step 2. On host h1's terminal, type the following command.

```
send.py 10.0.0.2 HelloWorld
```

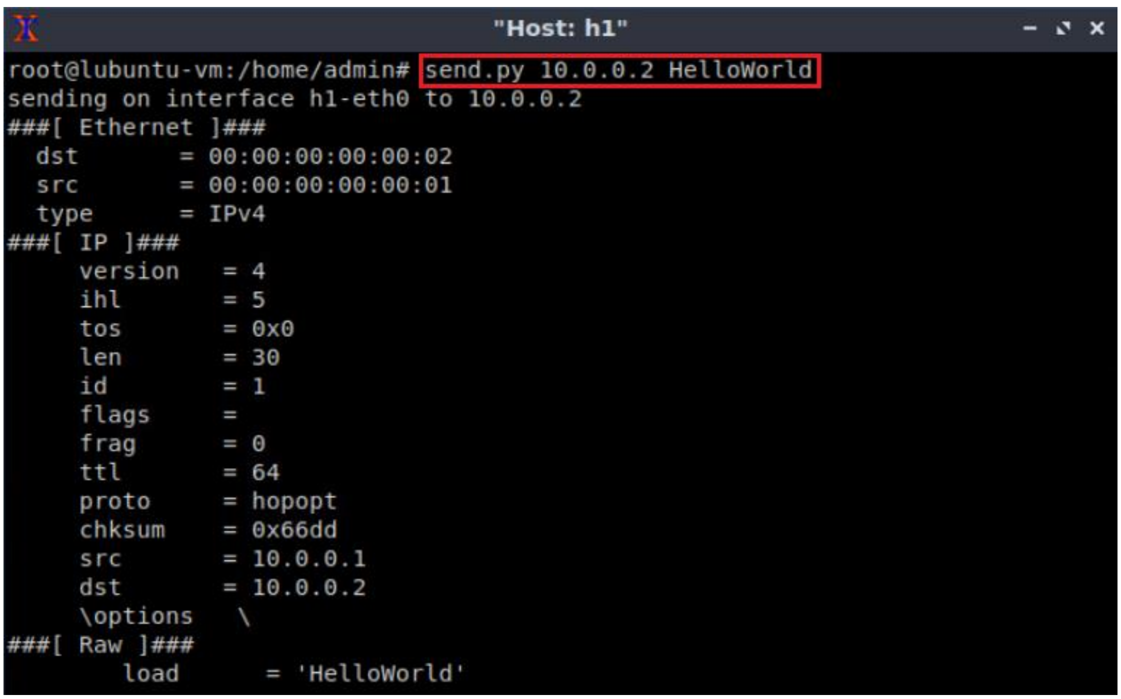


Figure 22. Sending a packet from host h1 to host h2.

Step 3. Verify that the packet was received on host h2.

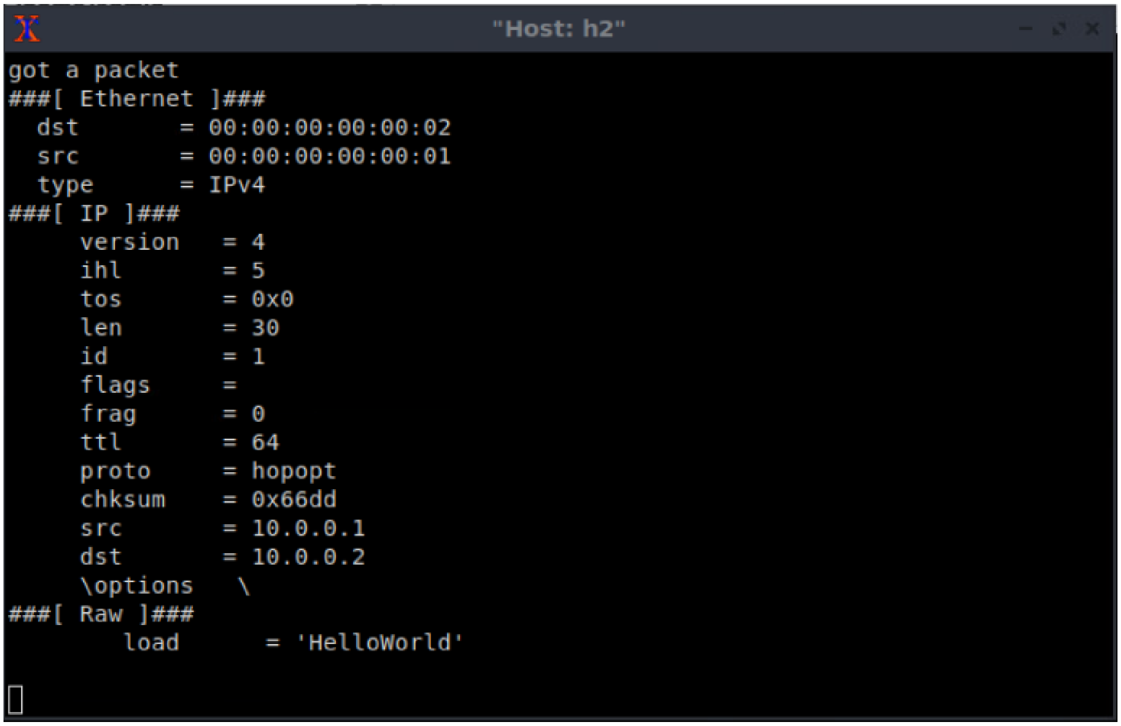


Figure 23. Packet received on host h2.

6.2 Reading the register's value

Step 1. Go back to the switch's terminal and start the CLI by issuing the following command.

```
simple_switch_CLI
```

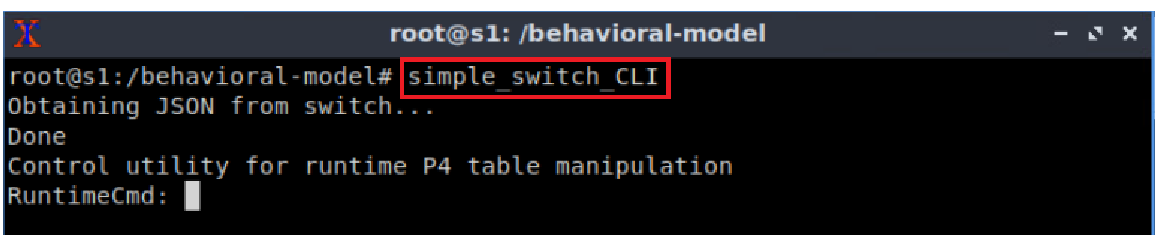


Figure 24. Starting the switch's CLI.

Step 2. Read the value of the `last_src_IP` register at index 0 by issuing the command shown below. This register contains the last source IP address.

```
register_read MyIngress.last_src_IP 0
```

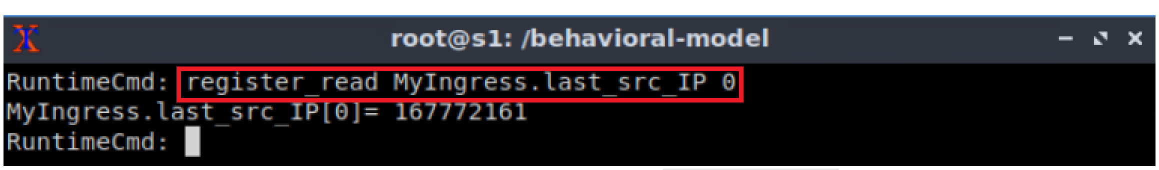


Figure 25. Reading the value of register `last_src_IP` at index 0.

Note that the decimal value 167772161 correspond to the IP address 10.0.0.1.

6.3 Sending a custom packet from host h3 to host h4

Step 1. Open a terminal in host h3 and issue the following command.

```
send.py 10.0.0.4 HelloWorld
```

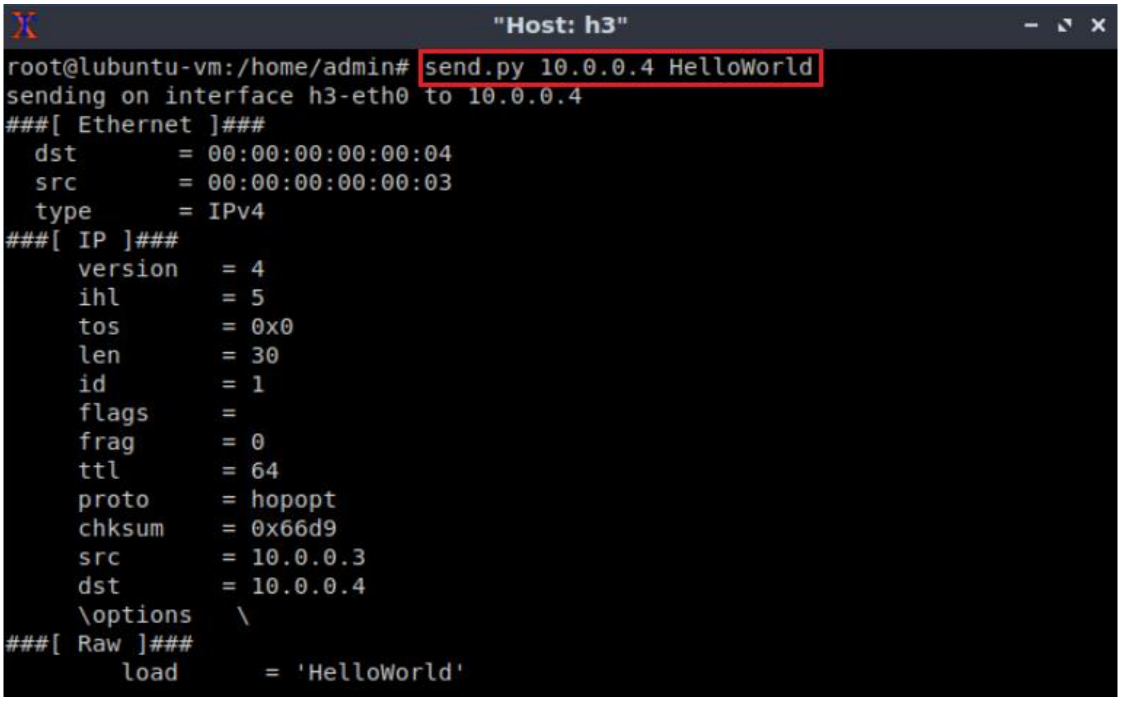


Figure 26. Sending a packet from host h3 to host h4.

Step 2. Similarly, read the value of the `last_src_IP` register at index 0 by issuing the command shown below. This register contains the last source IP address.

```
register_read MyIngress.last_src_IP 0
```

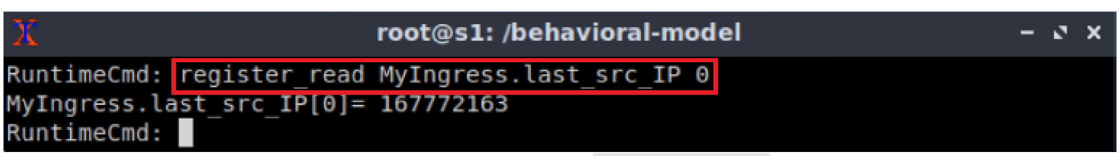


Figure 27. Reading the register `last_src_IP` at index 0.

Note that the decimal value 167772163 correspond to the IP address 10.0.0.3.

6.4 Manipulating registers from the control plane

Step 1. Write the following value into the register `last_src_IP` by issuing the following command.

```
register_write MyIngress.last_src_IP 0 321
```

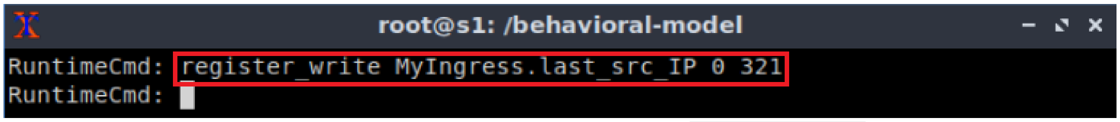


Figure 28. Writing a value to register `last_scr_IP`.

Step 2. Read the value stored in register `last_src_IP` by issuing the command below.

```
register_read MyIngress.last_src_IP 0
```

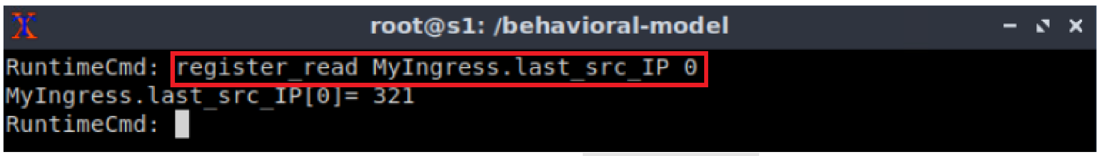


Figure 29. Reading the register `last_src_IP` at index 0.

The figure above shows that the value 321 was stored in register `last_src_IP`.

Step 3. Clear to zero register `last_src_IP` by issuing the following command.

```
register_reset MyIngress.last_src_IP
```

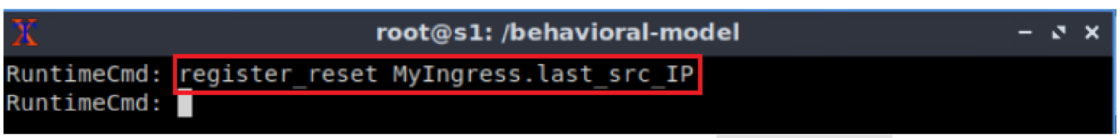


Figure 30. Removing values from register `last_scr_IP`.

Step 4. Read the value stored in register `last_src_IP` by issuing the command below.

```
register_read MyIngress.last_src_IP
```

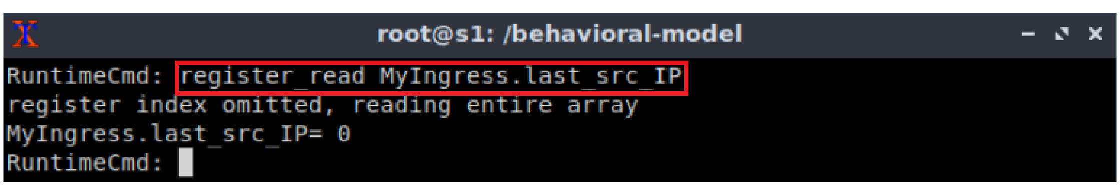


Figure 31. Reading the register array.

Note that the value stored in register `last_src_IP` was cleared to zero.

This concludes lab 9. 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.

UNIVERSITY OF
SOUTH CAROLINA

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 10: Calculating Packets Interarrival Times using Hashes and Registers

Document Version: 04-28-2022

Contents

Overview

Programmable data planes are capable of storing arbitrary information that can be accessed by multiple packets traversing the switch. This lab describes how to read and write information to the switch using stateful components known as registers. Registers can be written and read from both the control and the data planes. The use case demonstrated in this lab describes how to use registers to compute the inter-arrival time between packets belonging to the same flow.

Objectives

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

1. Understand how to declare registers in a P4 program.
2. Read and write data to registers using both the control and the data planes.
3. Identify unique flows by leveraging hashing functions.
4. Read the values of the inter-arrival times from the packet headers.

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 P4 registers.
2. Section 2: Lab topology.
3. Section 3: Creating a P4 program to calculate the interarrival time.
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 P4 registers

P4 targets implementations implement registers to save arbitrary data[1]. Registers are stateful elements used to store values longer than the time it takes to process a packet[2]. This feature allows the creation of P4 programs where multiple packets can access registers. Registers in P4 are organized into named arrays of cells. These cells are referred to by an index that indicates the location of a value. Registers can be read and written by both the control and the data plane. In P4, registers are global memory resources meaning that any match-action tables can reference them.

1.1 Declaring and using a register array

The syntax below shows how to declare a register array in P4. The register array R1 contains `M` values of `N` bits.

```
register<bit<N>>(M) R1;
```

Figure 1 depicts a graphical representation of the register `R1`. The functions `write` and `read` are used to store and retrieve values from a specific position, where an index specifies the position. For example, the programmer invokes the following function to store the value `val` in position 0 in the register array R1.

```
R1.write(0,val)
```

Similarly, the user invokes the function shown below to read a value stored in position 3. Note that the retrieved value is stored in the variable `res`.

```
R1.read(res,3)
```

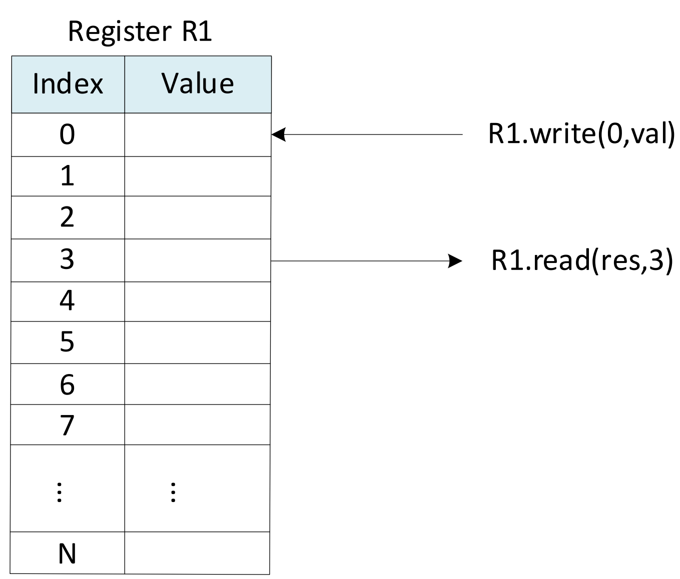


Figure 1. Register array R1. The register array contains N entries of M bits. The index indicates the position of the value. Using the functions `read` and `write`, programmers can retrieve and modify values in the register array.

1.2 Hashes in P4

P4 targets implement hash functions to map arbitrary data to a hash value. For example, the V1Model implements hash functions as externs[1]. The following code shows how to call a hash function in P4.

```
hash(hash_val, algo, min_val, {val_1, val_2, ..., val_N}, (n_bits, max_val))
```

The parameters of the hash function are as follows:

- `hash_val`: variable used to store the hash value.
- `algo`: indicates the hashing algorithm. For example, the V1Model supports crc16, crc32, universal hashing (i.e., random), xor32, and others.
- `min_val`: establishes the minimum hash value.
- `{val_1,val_2,…,val_N}`: values to be hashed.
- `n_bit`: number of bits of the output (i.e., width).
- `max_val`: maximum hash value.

1.3 Lab scenario

This lab shows how to compute the interarrival time of packets belonging to the same flow using registers. The interarrival time is the time difference between two consecutive packets. In this lab, the user will create a P4 program to store the timestamps of two consecutive packets and calculate the difference between them, obtaining the value of the interarrival time. The P4 program will use hashes to identify packets belonging to a flow.

The P4 program presented in this lab will implement the following steps to calculate the interarrival time. Figure 2 summarizes these steps.

1- Identify a flow by hashing the source and destination IP addresses. The hash value will be used as an index for the register array.
2- Extract the previous timestamp from the register array using the index calculated in step 1.
3- Compute the difference between the current timestamp and the previous timestamp. The result is the interarrival time.
4- Update the cell referenced in step 2 with the current timestamp.
5- Insert the current interarrival time in a custom header.

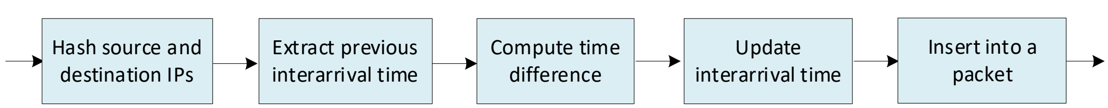


Figure 2. Interarrival processing block diagram.

2 Lab topology

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

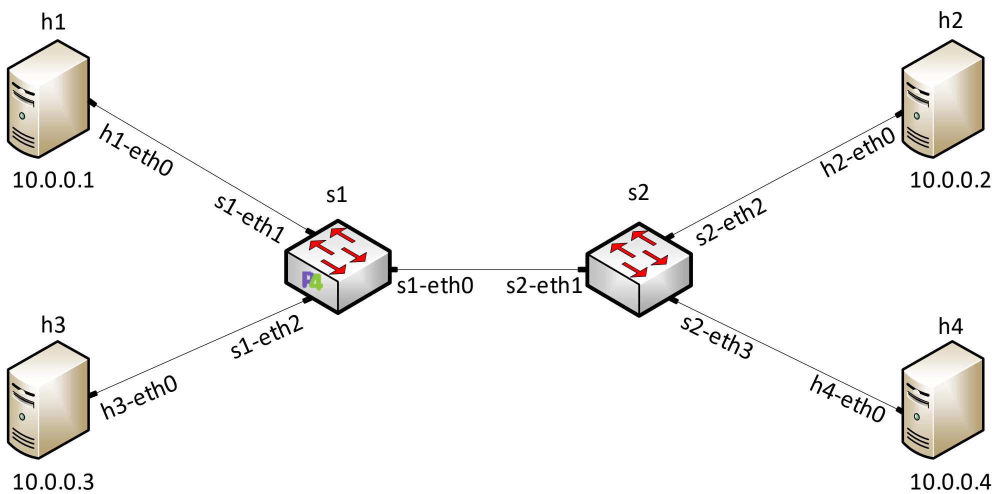


Figure 3. Lab topology.

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`.

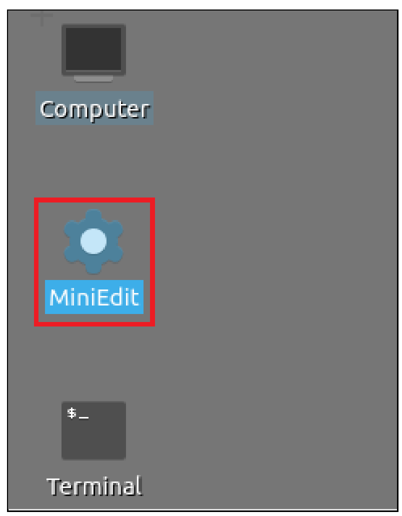


Figure 4. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab10* folder and search for the topology file called *lab10.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

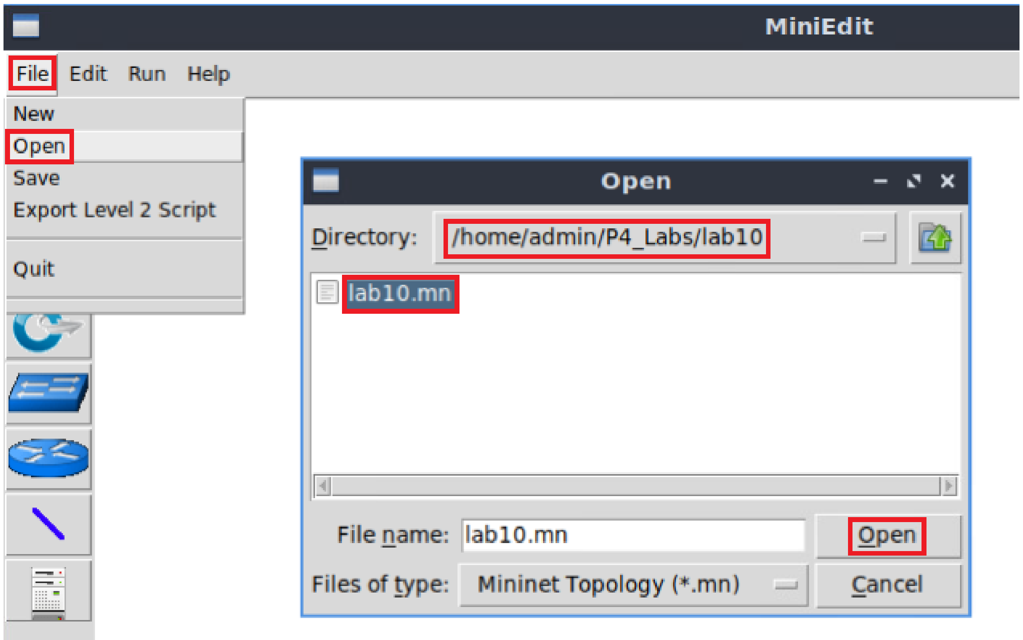


Figure 5. MiniEdit's *Open* dialog.

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.

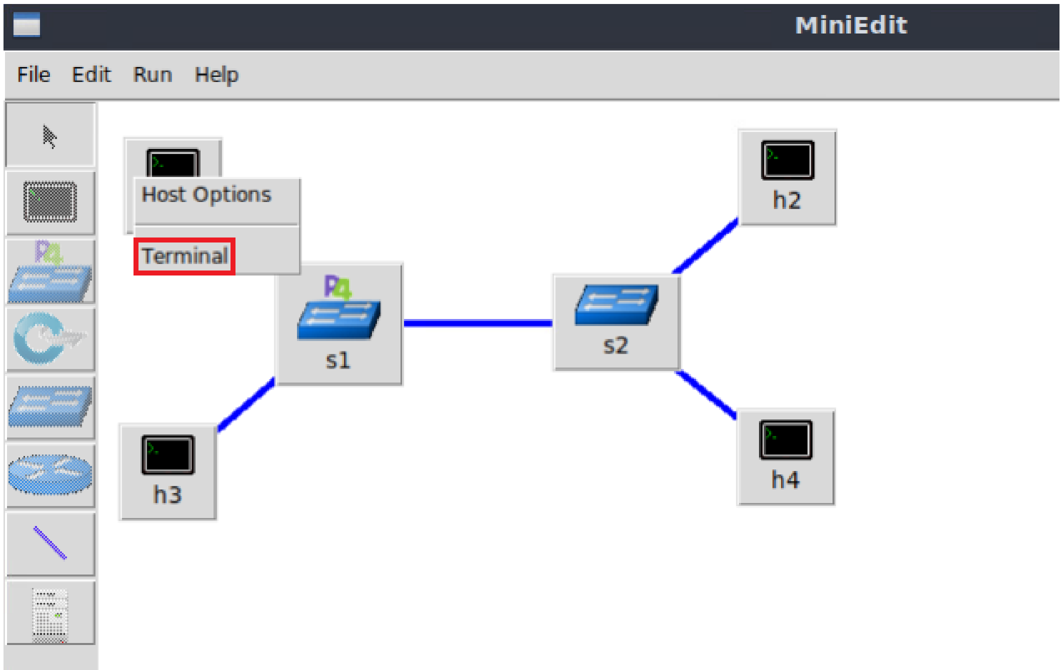


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

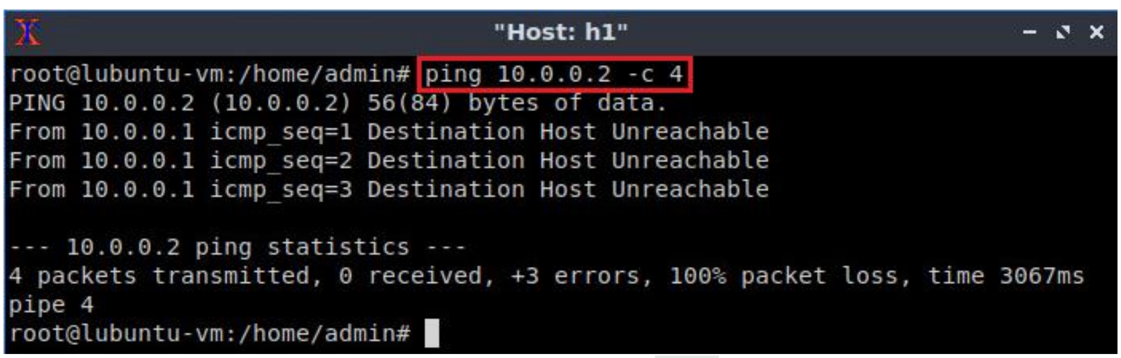


Figure 8. Connectivity test using `ping` command.

The figure above shows unsuccessful connectivity between host h1 and host h2. This result happens because there is no P4 program loaded on the switch.

3 Creating a P4 program to calculate the interarrival time

In this section, you will create a P4 program to compute the interarrival time. First, you will load the programming environment. Then, you will define a custom header to store the interarrival time. Following, you will create the actions to compute the flow ID and get the interarrival time. The flow ID is produced by a hashing algorithm that computes the source and destination IPv4 addresses to produce an index. This index indicates the position in the last interarrival time in a register array. Finally, you will define the action to calculate the interarrival time.

3.1 Loading the programming environment

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

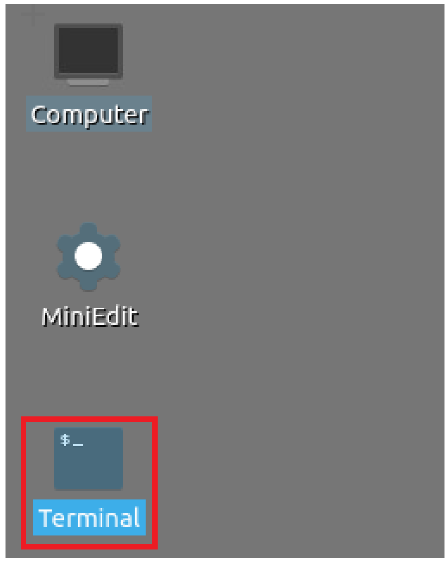


Figure 9. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). 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/lab10
```

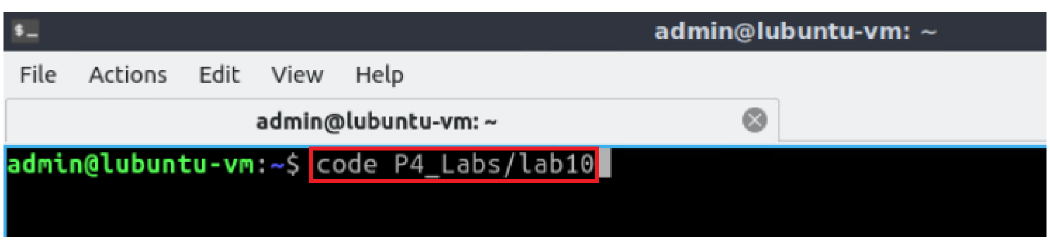


Figure 10. Loading the development environment.

3.2 Defining a custom header

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.

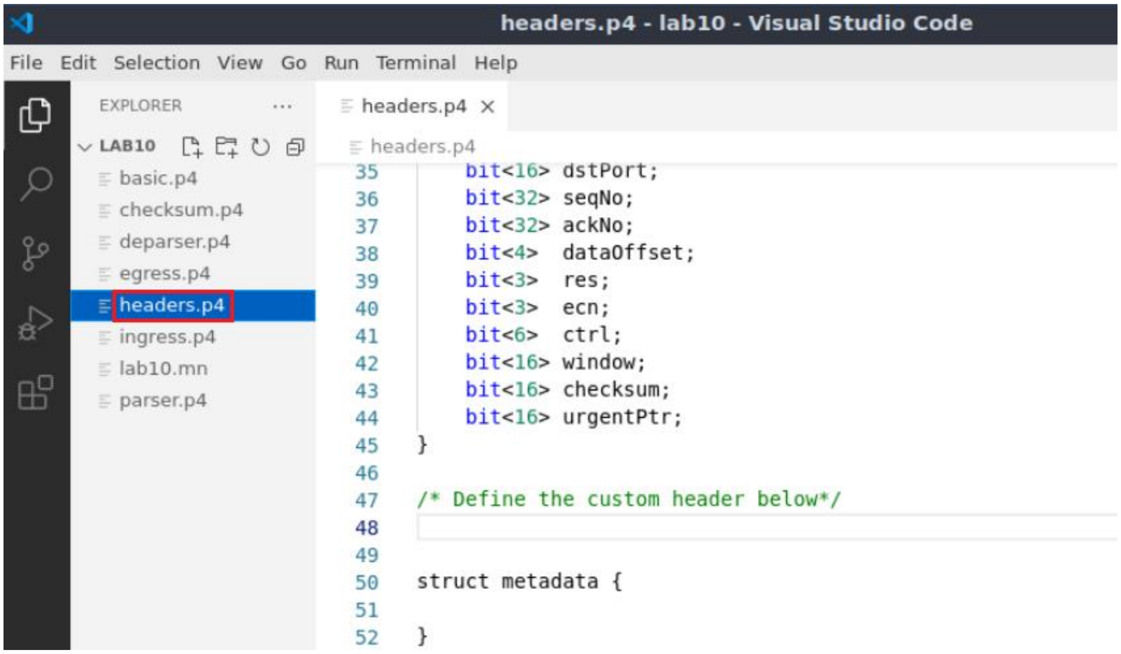


Figure 11. Inspecting the *headers.p4* file.

Step 2. Define the following custom header by adding code shown below.

```
header interarrival_t {
       bit<48> interarrival_value;
}
```

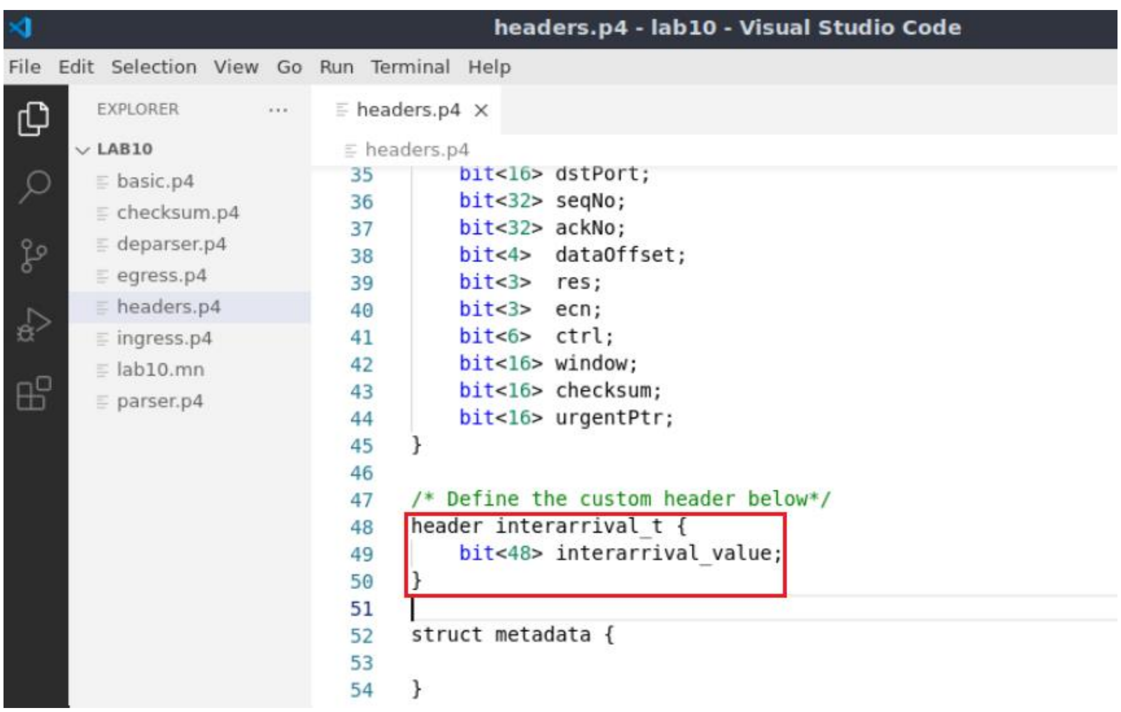


Figure 12. Defining a custom header type.

Step 3. Append the custom header to current Ethernet and IPv4 headers by inserting the following line of code.

```
interarrival_t interarrival;
```

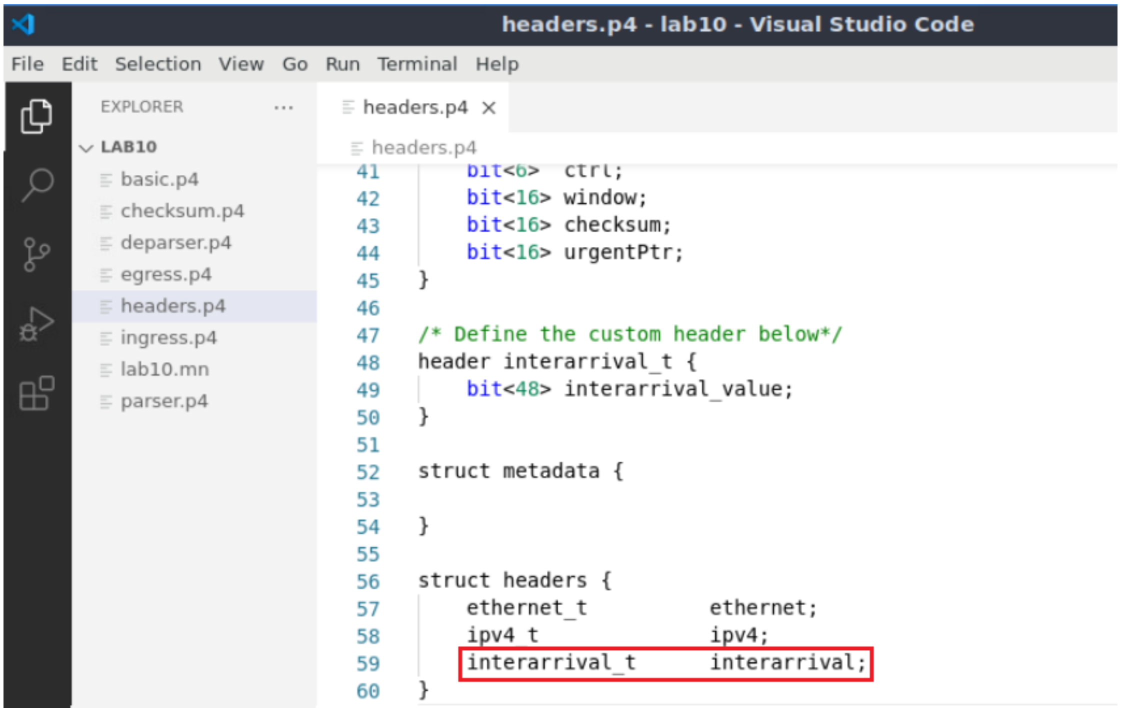


Figure 13. Defining a custom header.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

3.3 Classifying flows by hashing the source and the destination IPs

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. You will observe that the forwarding table is already defined.

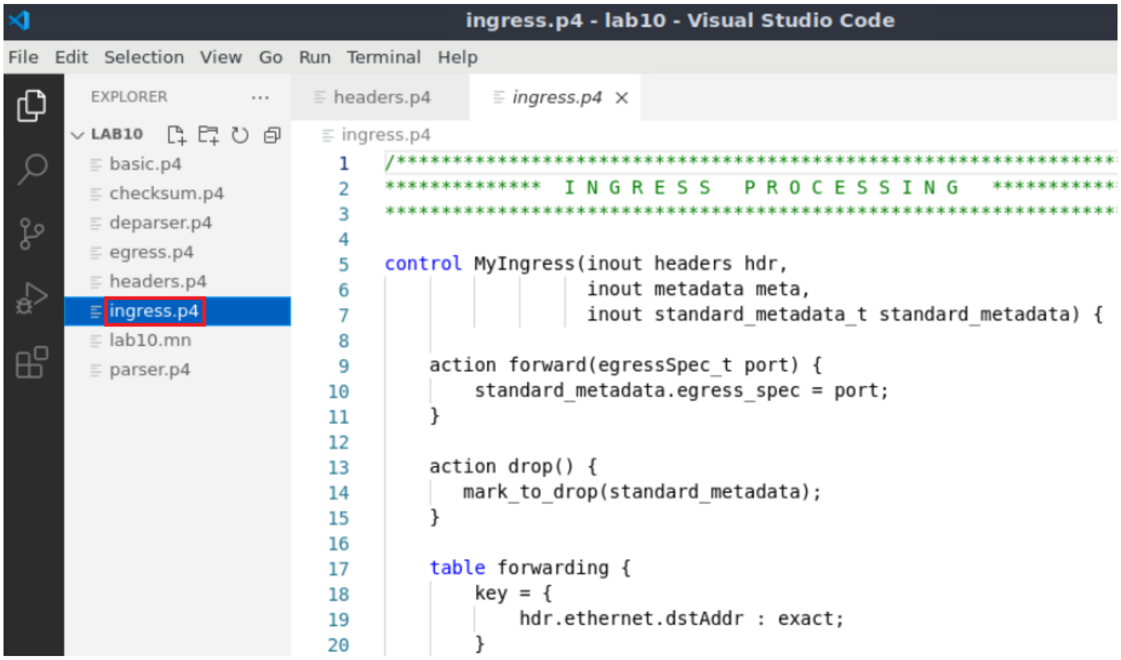


Figure 14. Inspecting the *ingress.p4* file.

Step 2. Add the following code in the *ingress.p4* file below the forwarding table. This creates a local variable that we will use to store the flow identifier.

```
bit<16> flow_id;
```

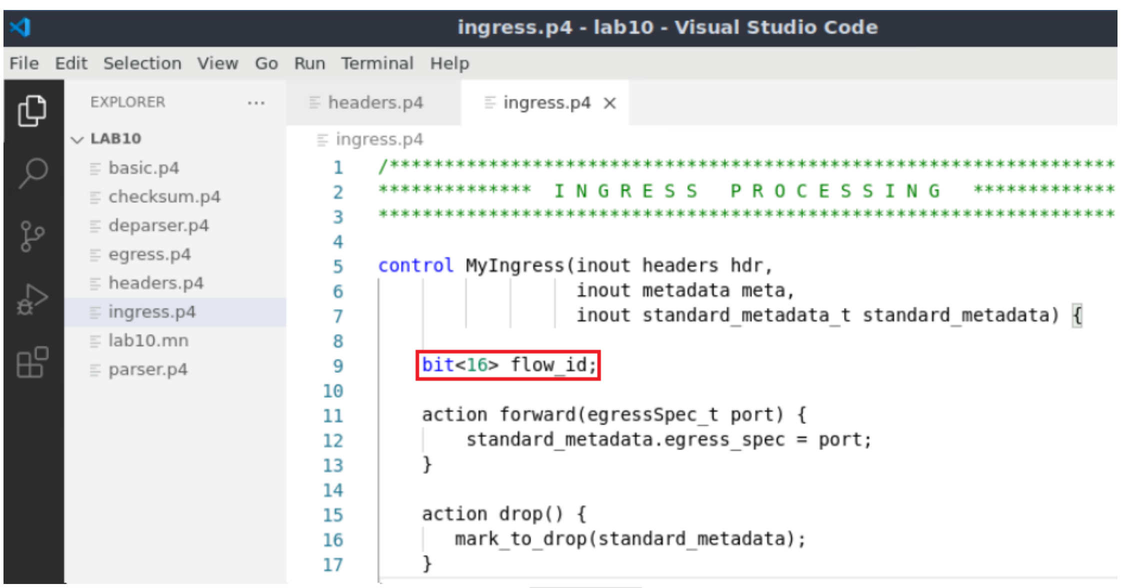


Figure 15. Defining the variable `flow_id` to store the flow identifier.

Step 3. Define the action `compute_flow_id` by adding the following piece of code.

```
action compute_flow_id() {
        hash(
               flow_id,
               HashAlgorithm.crc16,
               (bit<1>)0,
               {
                hdr.ipv4.srcAddr,
```

```
                hdr.ipv4.dstAddr
               },
               (bit<16>)65535);
}
```

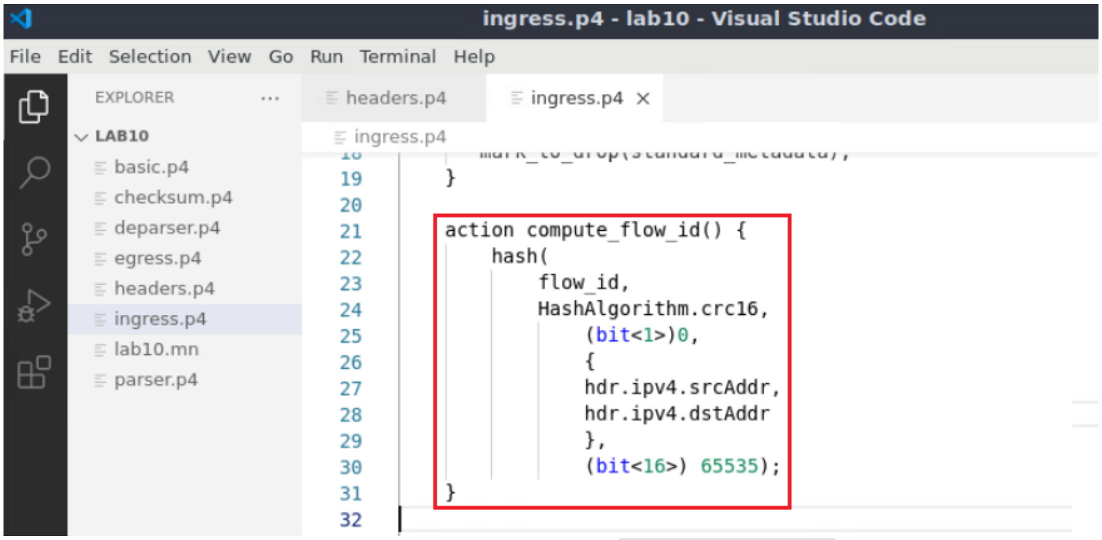


Figure 16. Defining the action `compute_flow_id`.

The code in the figure above hashes flows based on their source and destination IP addresses. The hash function `hash` produces a 16-bits output using the following parameters:

- `flow_id`: The variable used to store the output.
- `HashAlgorithm.crc16`: the hash algorithm.
- `bit<1>0`: the minimum (or base) value produced by the hash algorithm.
- `hdr.ipv4.srcAddr` and `hdr.ipv4.dstAddr`: the data to be hashed.
- `bit<16>65535`: the maximum value produced by the hash algorithm

Step 4. Save the changes to the file by pressing `Ctrl + s`.

3.4 Computing the interarrival time

Step 1. In the *ingress.p4*, define the register array by adding the code below.

```
register<bit<48>>(65535) last_timestamp_reg;
```

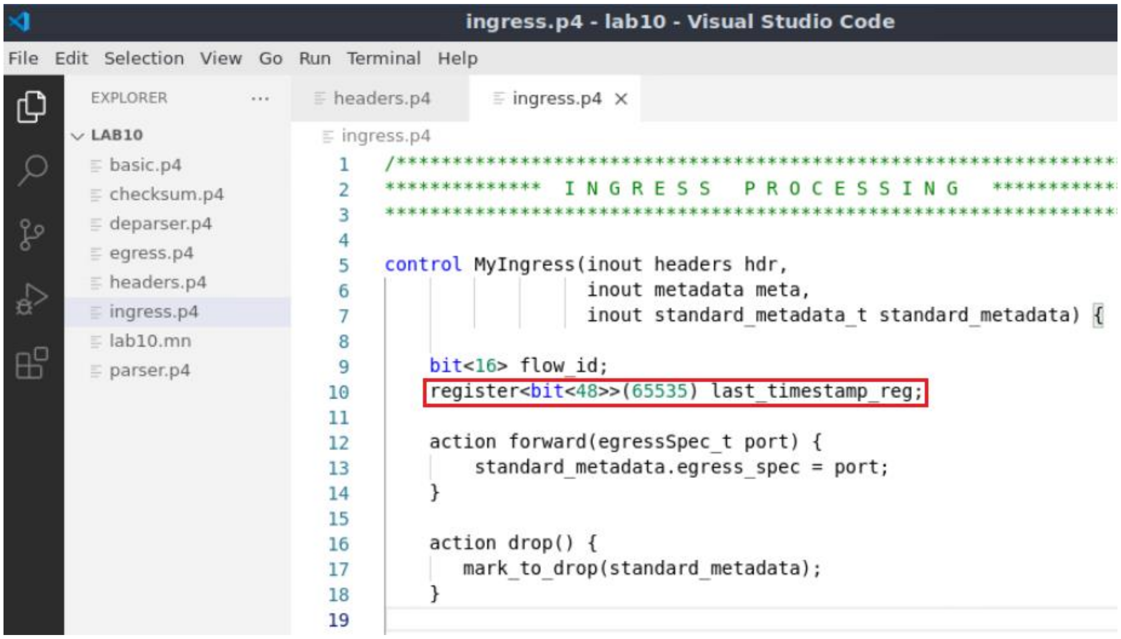


Figure 17. Defining a register array.

Step 2. Define the local variable to store the interarrival time.

```
bit<48> interarrival_value;
```

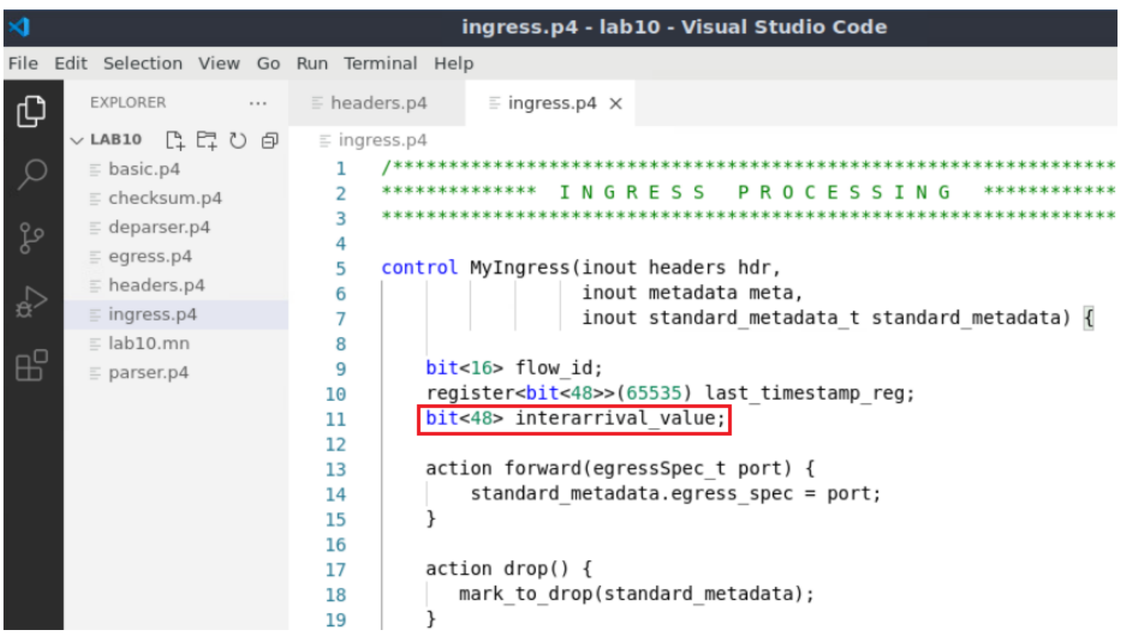


Figure 18. Defining a local variable.

Step 3. Define the action `get_interarrival_time` by adding the code below.

```
action get_interarrival_time (){
       bit<48> last_timestamp;
       bit<48> current_timestamp;

       last_timestamp_reg.read(last_timestamp, (bit<32>)flow_id);
       current_timestamp = standard_metadata.ingress_global_timestamp;

       if(last_timestamp != 0){
```

```
                interarrival_value = current_timestamp - last_timestamp;
        } else {
                interarrival_value = 0;
        }
        last_timestamp_reg.write((bit<32>)flow_id, current_timestamp);
}
```

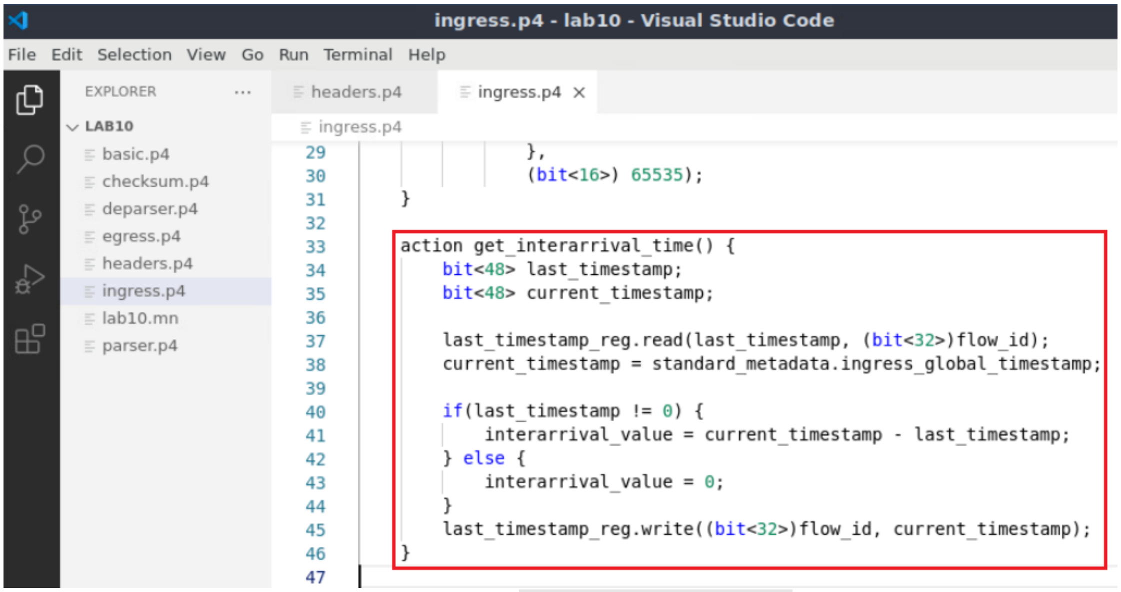


Figure 19. Defining the `get_interrival_time` action.

The code in the figure above is explained as follows:

- Line 34: declares the local variable `last_timestamp`, which will store the last timestamp.
- Line 35: declares the local variable `current_timestamp`, which will store the current timestamp.
- Line 37: reads the last timestamp stored in the register at index `last_timestamp`. This index was calculated using the action `compute_flow_id`.
- Line 38 assign to the `current_timestamp` variable the switch’s global ingress timestamp from `standard_metadata.ingress_global_timestamp`.
- Line 40-44: executes the following conditional statement: if the last timestamp stored in the register was not equal to zero, compute the interarrival time by subtracting the last timestamp from the current timestamp. Otherwise, the interarrival value will have the value of 0.
- Line 45: Update the register value at index `flow_id` with the current timestamp.

Step 4. Apply the ingress logic by adding the following piece of code.

```
if(hdr.ipv4.isValid()){
        if(hdr.interarrival.isValid()){
                compute_flow_id();
                get_interarrival_time();
                hdr.interarrival.interarrival_value = interarrival_value;
        }
forwarding.apply();
}
```

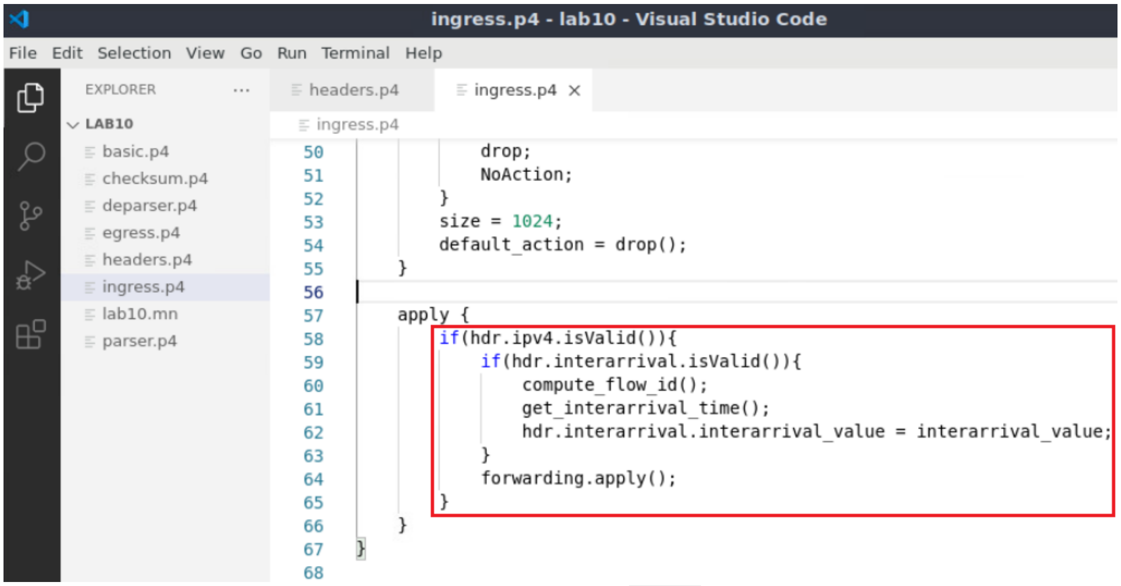


Figure 20. Defining the `apply` logic.

The code in the figure above applies the ingress pipeline logic if the packet has a valid IPv4 header. Then, if the interarrival header is valid, the actions `compute_flow_id` and `get_interarrival_time` are invoked. Lastly, the previous value in the interarrival header is updated with `interarrival_value`.

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

4 Loading the P4 program

In this section, you will compile and load the P4 binary into the switches. You will also verify that the binaries reside in switches' filesystem.

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside VS Code to compile the program.

```
p4c basic.p4
```

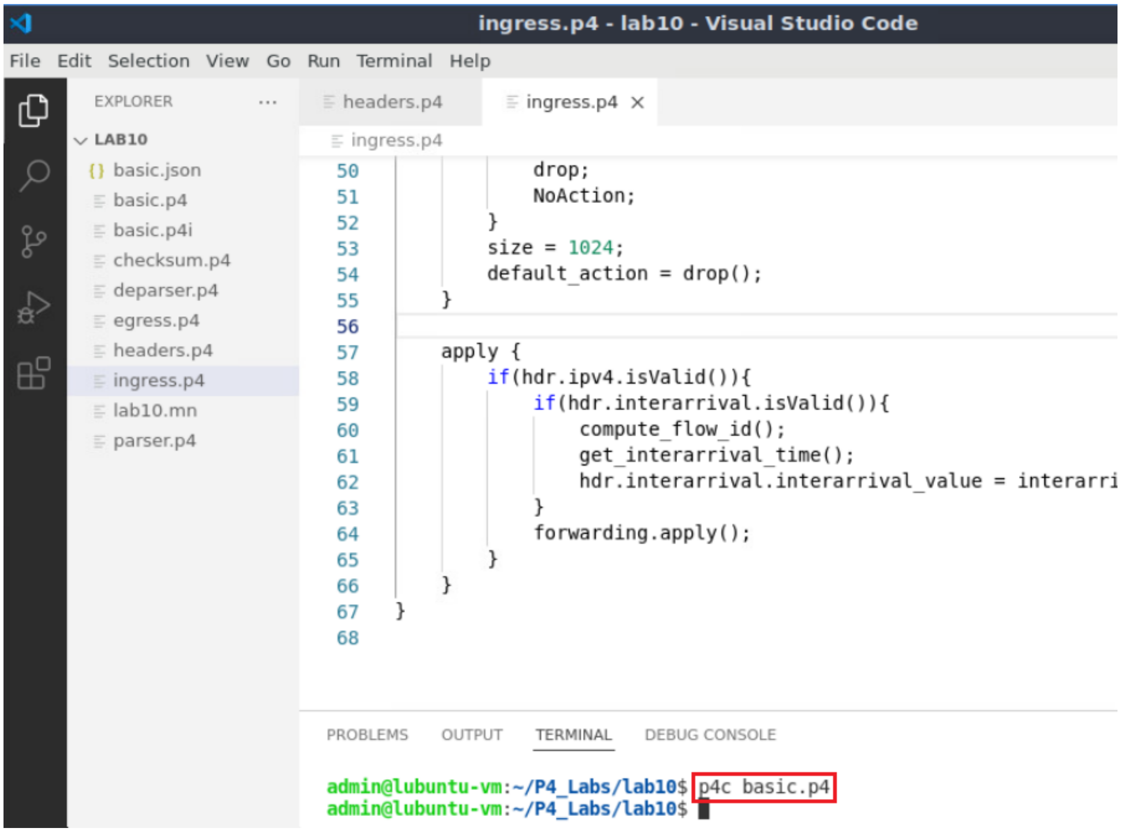


Figure 21. Compiling a P4 program.

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

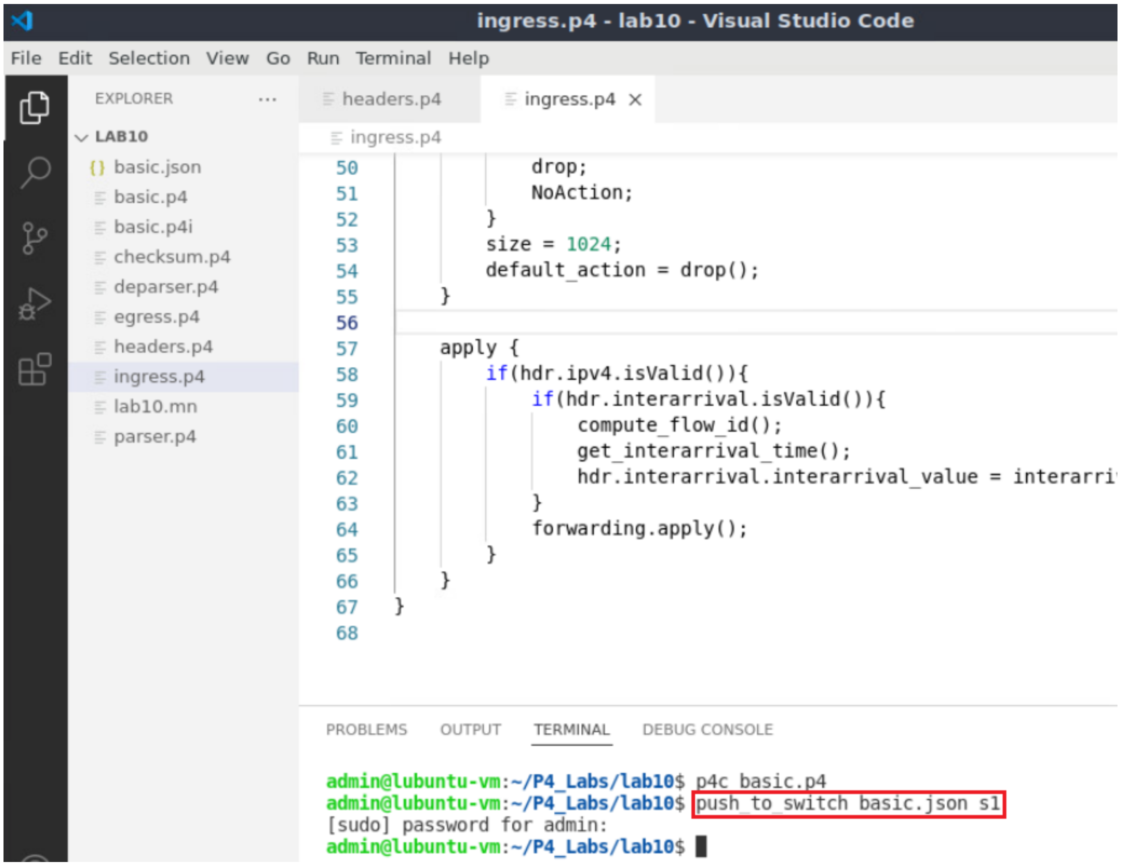


Figure 22. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

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

Figure 23. Maximizing the MiniEdit window.

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

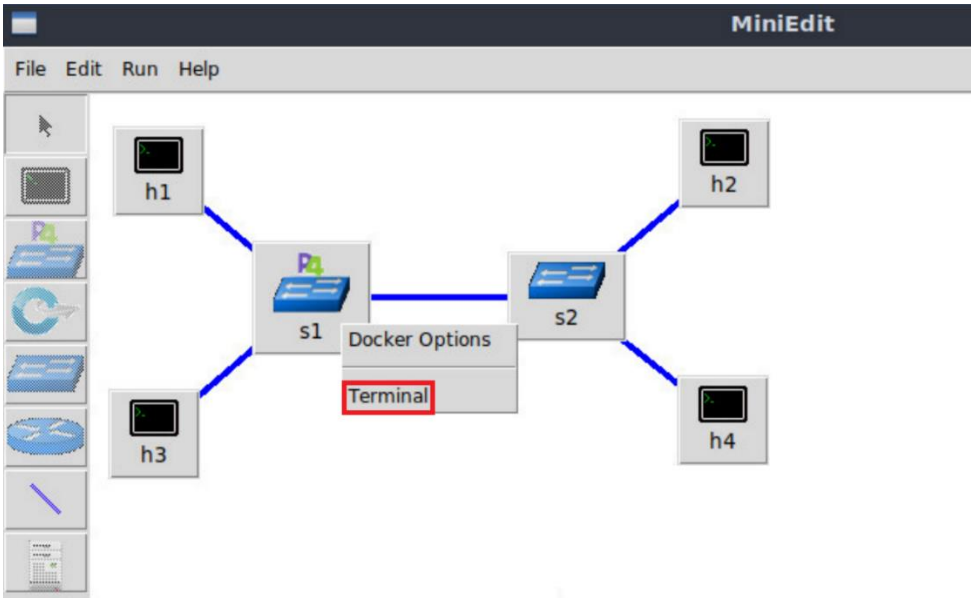


Figure 24. Starting the terminal on the switch.

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
```

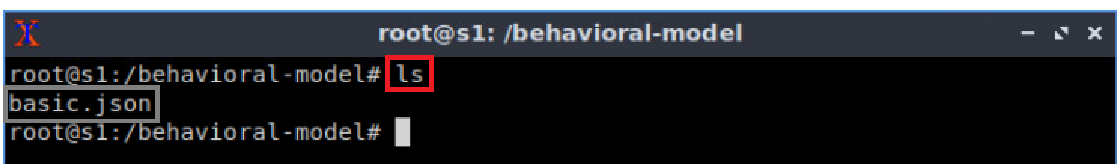


Figure 25. Displaying the contents of the current directory in the switch s1.

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

5 Configuring switch s1

In this section, you will map switch s1 interfaces to the ports in the P4 program and start the switch daemon. Then, you will load the rules to populate the match action tables.

5.1 Mapping the P4 program's ports

Step 1. Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```

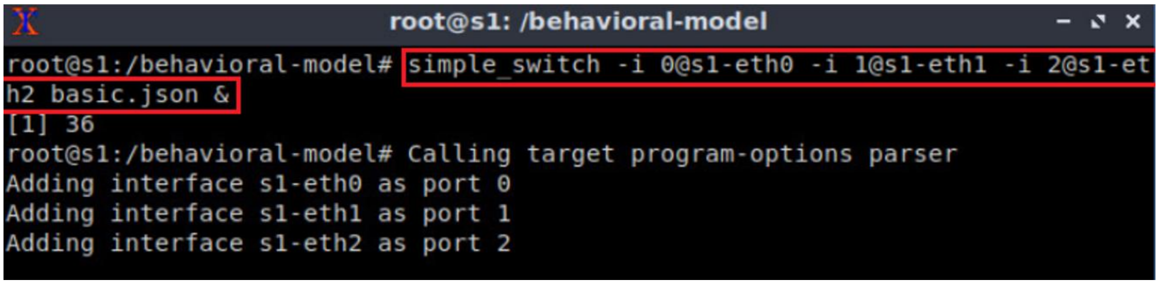


Figure 26. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

5.2 Loading the rules to the switch

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

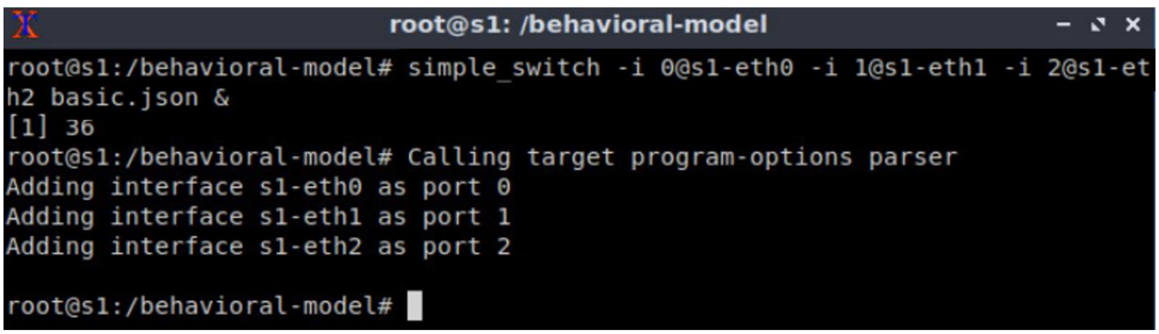


Figure 27. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/lab10/rules.cmd
```

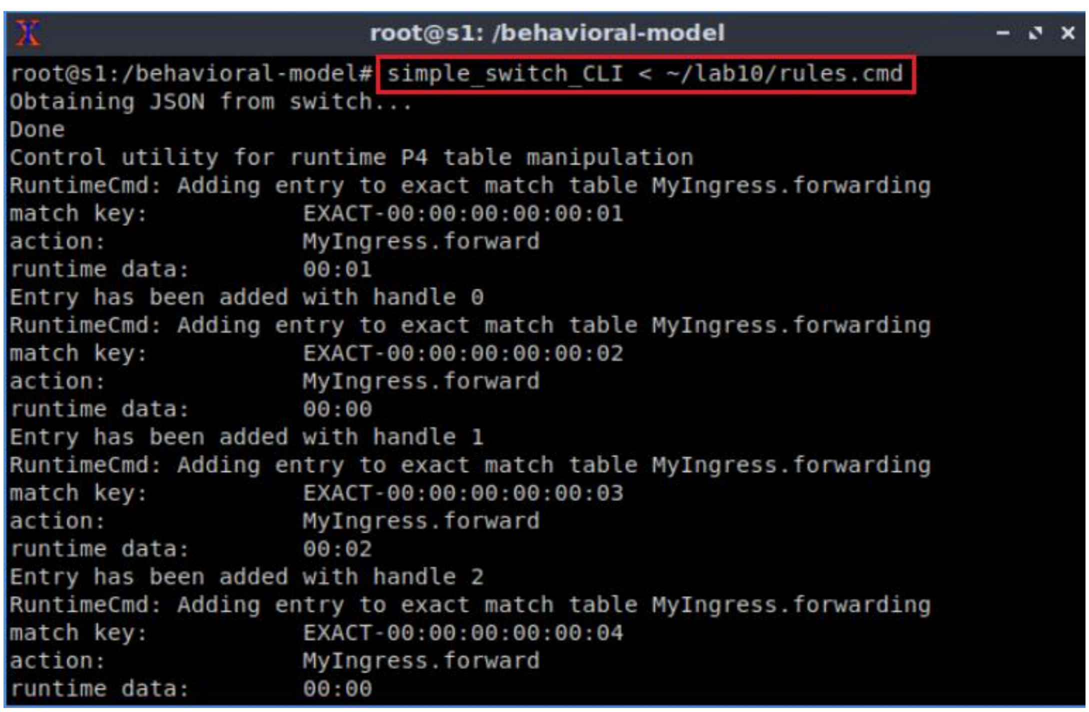


Figure 28. Populating the forwarding table into switch s1.

The script above pushes the rules into the match-action table `forwarding`. This table forwards packets matching the destination IPv4 address.

6 Testing and verifying the P4 program

This section shows the steps to send and receive packets at a specific rate. From host h1, you will send 10 packets per second, whereas, from host h2, you will send 20 packets per second. Then, you will observe different interarrival times corresponding to each flow.

6.1 Generating traffic at 10 packets per second

Step 1. Go back to MiniEdit and open a terminal on host h2's terminal. Issue the following command so that, host h2 starts listening for packets.

```
recv.py -p interarrival
```

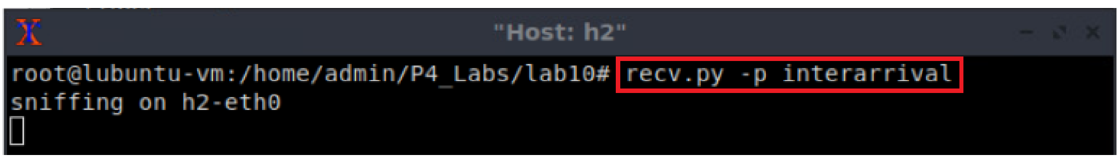


Figure 29. Listening for incoming packets in host h2.

The script above receives the following parameters:

- `-p`: enables listening to a specific protocol.
- `interarrival`: the protocol type.

Step 2. On host h1's terminal, type the following command.

```
send.py 10.0.0.2 10 -p interarrival
```

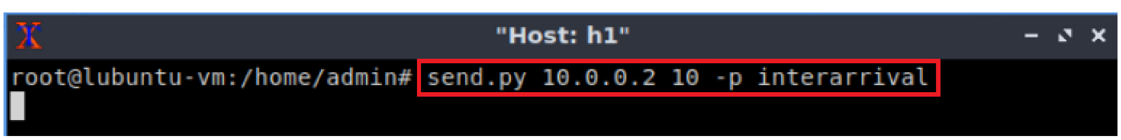


Figure 30. Sending 10 packets per second from host h1 to host h2.

The script above receives the following parameters:

- `10.0.0.2`: the destination IPv4 address.
- `10`: number of packets per second.
- `-p`: enables listening to a specific protocol.
- `interarrival`: the protocol type.

Step 3. Go back to host h2 terminal and verify the interarrival time.

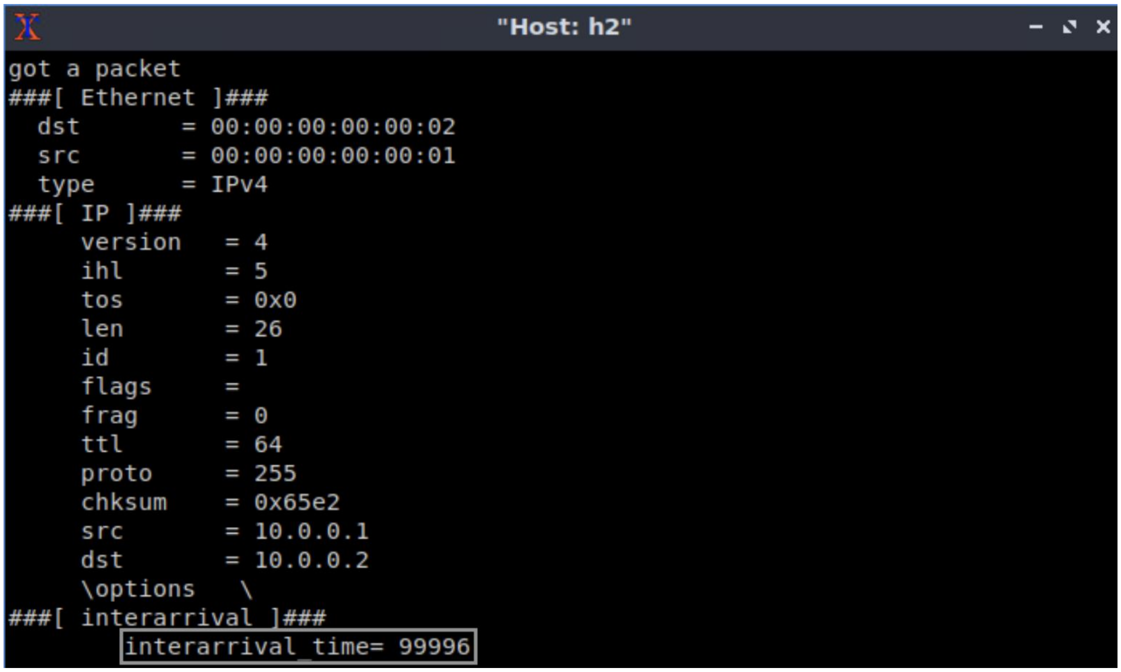


Figure 31. Verifying the interarrival time on host h2.

By sending 10 packets per second, the expected interarrival time is around 100 milliseconds, or 100,000 microseconds as observed in the figure above.

6.2 Generating traffic at 20 packets per second

Step 1. Go back to MiniEdit and open a terminal on host h4. Issue the following command so that, host h4 starts listening for packets.

```
recv.py -p interarrival
```

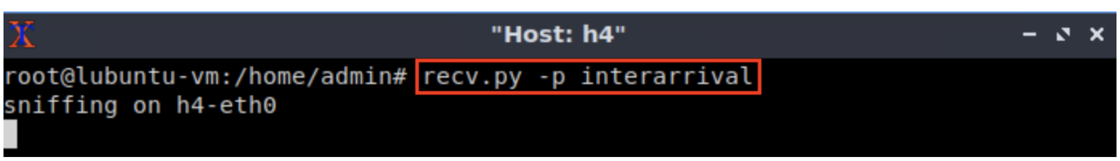


Figure 32. Listening for incoming packets in host h4.

The script above receives the following parameters:

- `-p`: enables listening to a specific protocol.
- `interarrival`: the protocol type.

Step 2. Go back to MiniEdit and open a terminal on host h3. Issue the following command so that, host h4 starts sending 20 packets per second.

```
send.py 10.0.0.4 20 -p interarrival
```

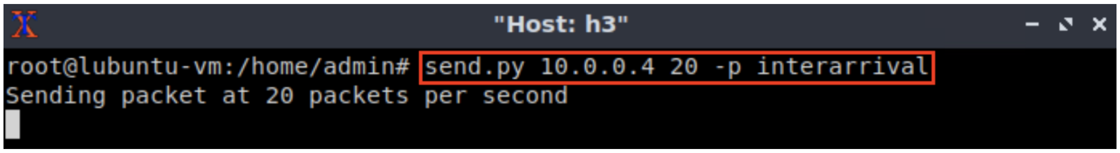


Figure 33. Sending 20 packets per second from host h3 to host h4.

The script above receives the following parameters:

- `10.0.0.4`: the destination IPv4 address.
- `20`: number of packets per second.
- `-p`: enables listening to a specific protocol.
- `interarrival`: the protocol type.

By sending 20 packets per second, the expected interarrival time should be approximately 50 milliseconds, or 50,000 us.

Step 3. Go back to host h4 terminal and verify the interarrival time.

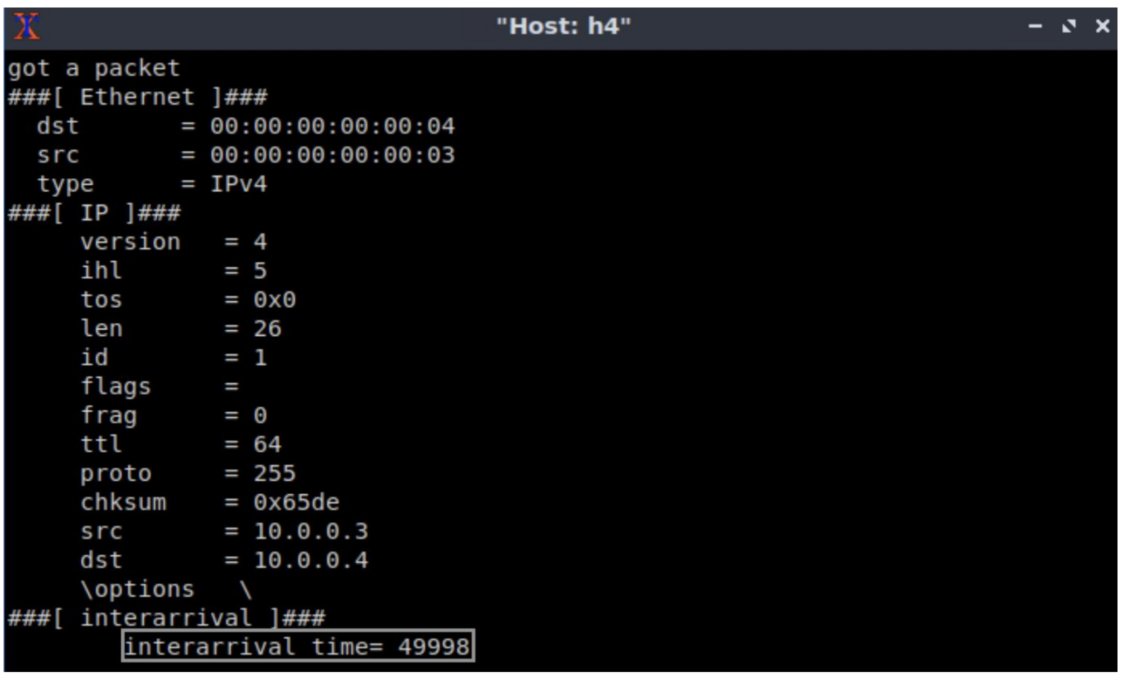


Figure 34. Verifying the interarrival time on host h4.

By sending 20 packets per second, the expected interarrival time is around 50 milliseconds, or 50,000 microseconds as observed in the figure above.

Step 4. Go back to host h2 and compare the interarrival time with the figure above. You will observe that the interarrival time is performed in a per flow basis

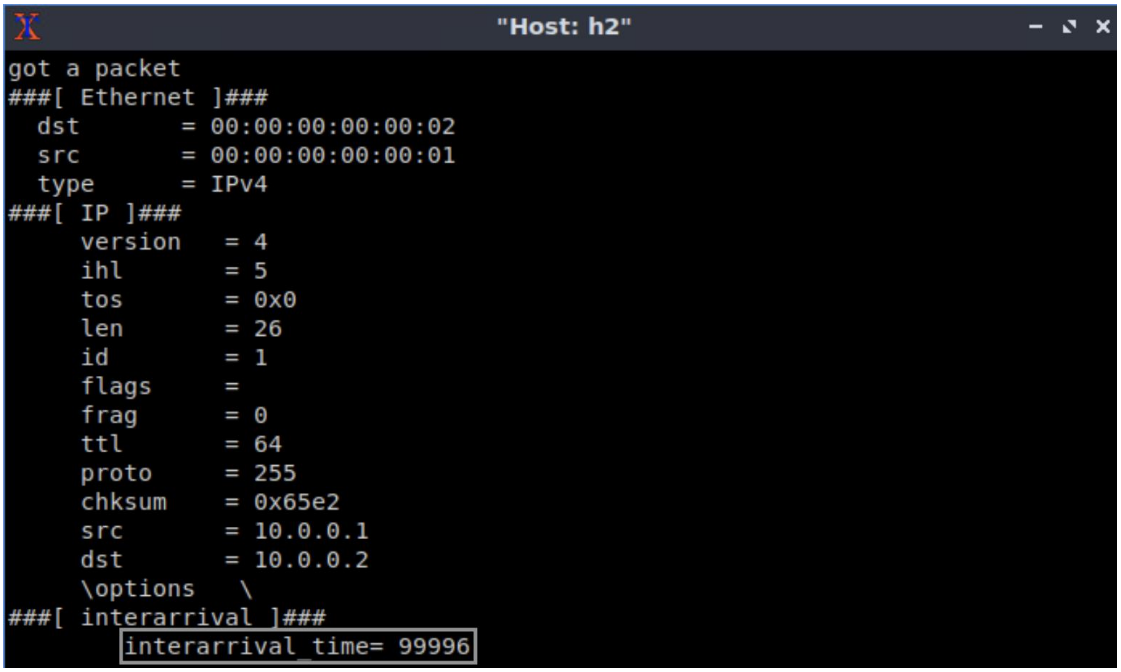


Figure 35. Verifying the interarrival time on host h2.

This concludes lab 10. Stop the emulation and then exit out of MiniEdit.

References

1. The P4 language Consortium. "*The V1Model*." [Online]. Available: https://tinyurl.com/bdzfarvy
2. The P4 Architecture Working Group. "*P4$_{16}$ Portable Switch Architecture (PSA)*." [Online]. Available: https://tinyurl.com/2wnkc6d2
3. Mininet walkthrough. [Online]. Available: http://Mininet.org.
4. 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.
5. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
6. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.

P4 PROGRAMMABLE DATA PLANES: APPLICATIONS, STATEFUL ELEMENTS, AND CUSTOM PACKET PROCESSING

Lab 11: Generating Notification Messages from the Data Plane using Digests

Document Version: 04-28-2022

Contents

Overview

This lab demonstrates how to use digests in P4. A digest is a communication mechanism used by the data plane to send values to the control plane. These values are then processed by the control plane to implement applications. In this lab, the user will create a P4 program and a controller that uses digests to implement a MAC learning application. The data plane produces a digest with the source MAC address and the ingress port. This digest is processed by the control plane to populate the forwarding table and provide connectivity between end hosts.

Objectives

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

1. Understand how to create digests in a P4 program.
2. Write a control plane application to receive the digests sent from the data plane.
3. Parse the digest and install forwarding rules in a match-action table.
4. Implement a basic MAC learning application on a P4 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 packet digests.
2. Section 2: Lab topology.
3. Section 3: Creating packet digests in P4.
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 packet digests

A digest consists of a mechanism to send a message from the data plane to the control plane. Digests contain data plane values such as packet headers or metadata to be processed by a program in the control plane (i.e., a controller). The controller can implement applications in programming languages such as C/C++, Java, or Python. Moreover, the controller can process multiple digests and communicate with the data plane using runtime APIs[1]. The controller can use these APIs to add, delete, or modify an entry in a match-action table, read registers, reset counters, change meter rates, etc.

1.1 Lab scenario

Figure 1 depicts an example of a controller application that implements MAC learning. The topology comprises two end hosts and a P4 switch. In the initial state, switch s1 does not have the forwarding rules to establish connectivity between host h1 and host h2. Therefore, a P4 program produces a digest with the source MAC address and ingress port with the first packet arriving to switch s1 from host h1. This digest is sent to the control plane, where a controller (i.e., controller.py) uses the source MAC address and ingress port to create a forwarding rule. Then, the controller populates the forwarding table in the data plane. Similarly, a new entry in the forwarding table is created when a packet is received from host h2. Figure 2 shows the resulting forwarding table.

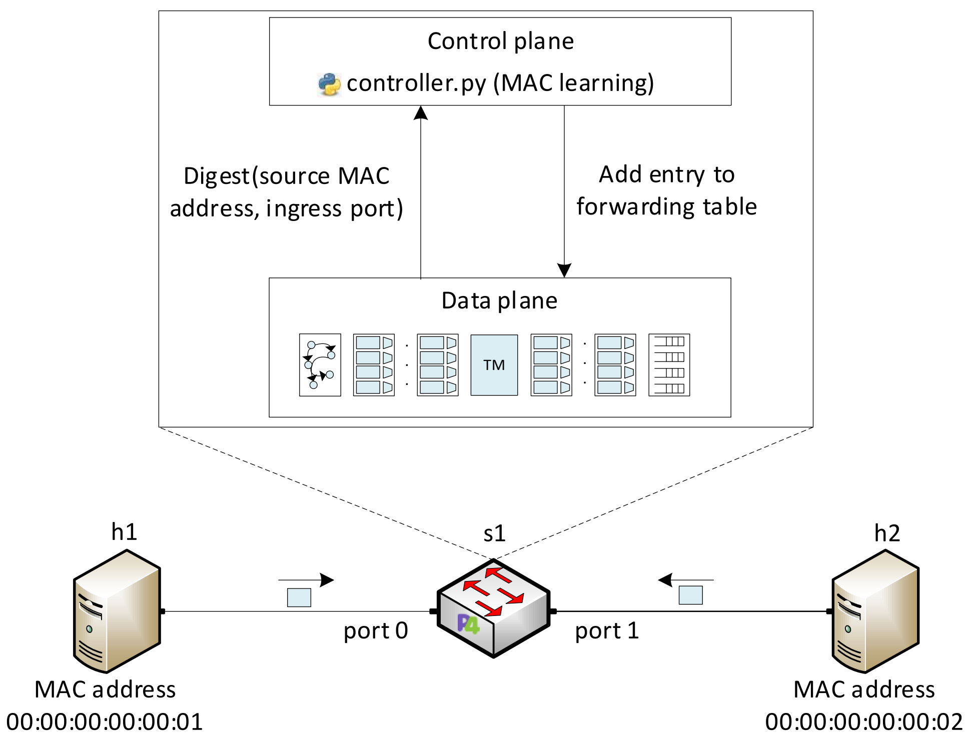


Figure 1. Lab scenario. Initially, switch s1 has an empty forwarding table, so that host h1 and host h2 cannot communicate. Host h1 sends a packet to switch s1. The data plane creates a digest with the source MAC address and ingress port and sends it to the control plane. Then the data plane populates the forwarding table in the data plane.

Once switch s1 learns the MAC addresses of host h1 and host h2, they can establish connectivity.

Forwarding table

Key	Action	Action Data
00:00:00:00:00:01	forward	egress port = 0
00:00:00:00:00:02	forward	egress port = 1

Figure 2. Forwarding table. The control plane populates the entries in the forwarding table. Then, host h1 and host h2 can establish connectivity.

2 Lab topology

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

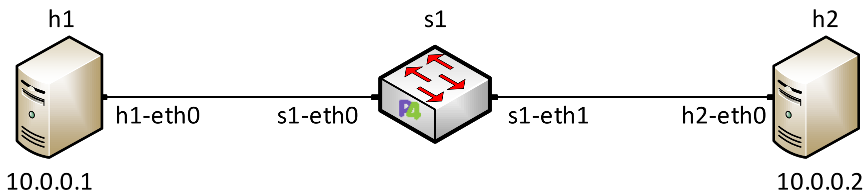


Figure 3. Lab topology.

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`.

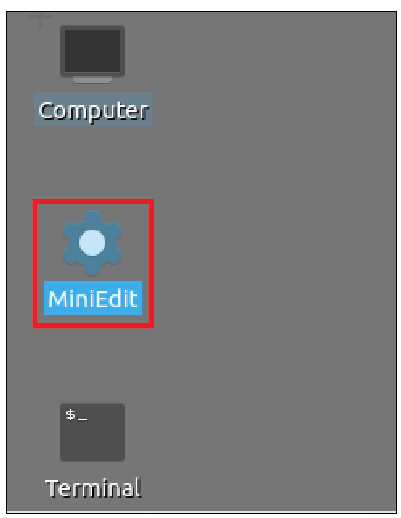


Figure 4. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab11* folder and search for the topology file called *lab11.mn* and click on *Open*. A new topology will be loaded to MiniEdit.

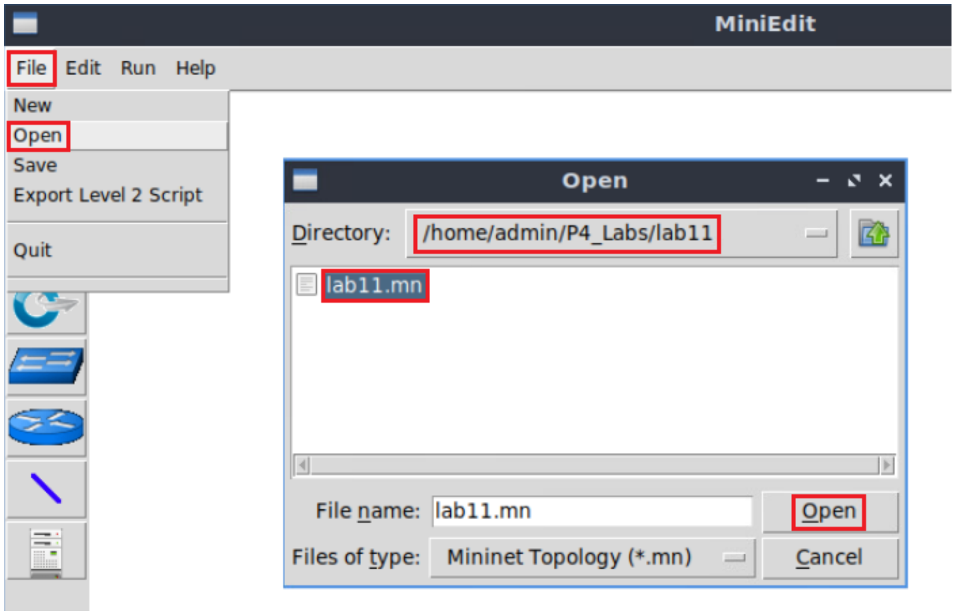


Figure 5. MiniEdit's *Open* dialog.

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.

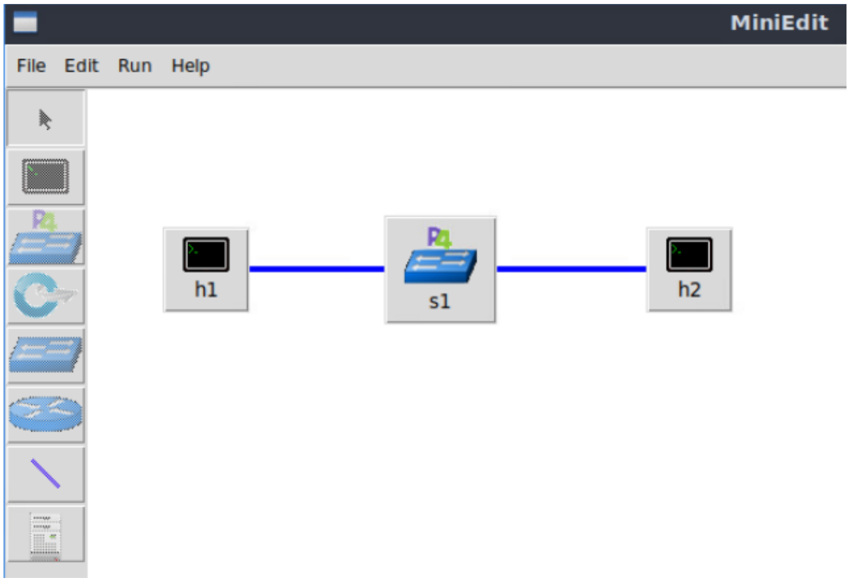


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

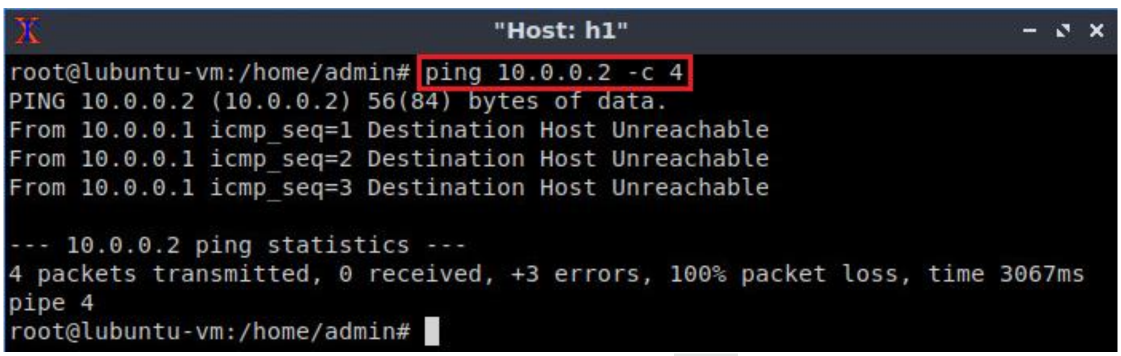


Figure 8. Connectivity test using `ping` command.

The figure above shows unsuccessful connectivity between host h1 and host h2. This result happens because there is no P4 program loaded on the switch.

3 Creating packet digests in P4

This section shows how to create a P4 program to generate a packet digest. A digest is a mechanism to send a message from the data plane to the control plane. The P4 program will produce a digest using the MAC address and the ingress port of an incoming packet and send it to the control plane. Then, a controller application in the control plane will process incoming digests and create the entries to populate the forwarding table.

3.1 Loading the programming environment

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

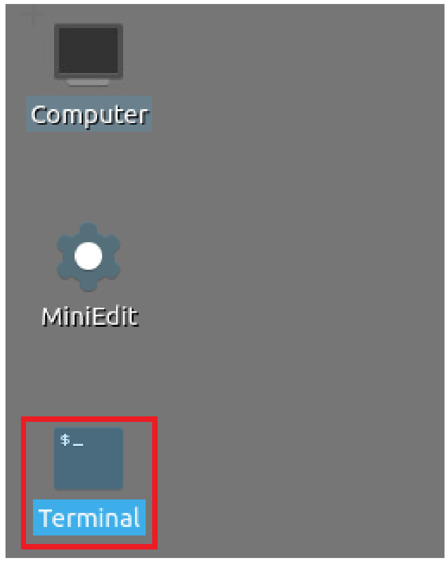


Figure 9. Shortcut to open a Linux terminal.

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

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/lab11
```

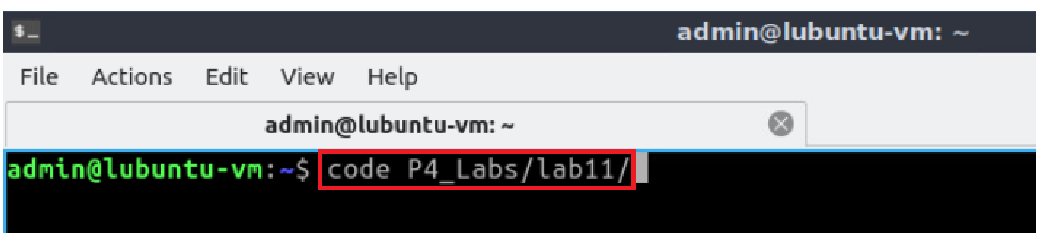


Figure 10. Loading the development environment.

3.2 Defining a custom header

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. The code in the figure below defines the Ethernet header.

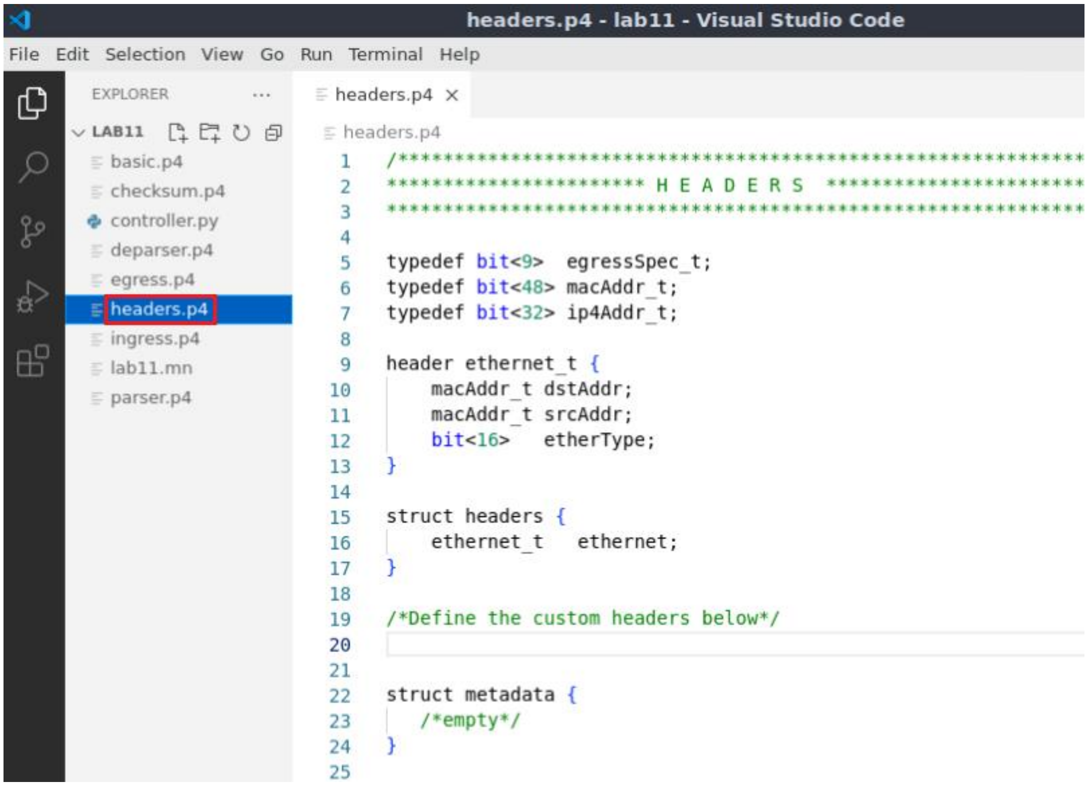


Figure 11. Inspecting the *headers.p4* file.

Step 2. Define the following custom header type by adding the code below.

```
struct digest_t {
   bit<48> srcAddr;
   bit<9>  in_port;
}
```

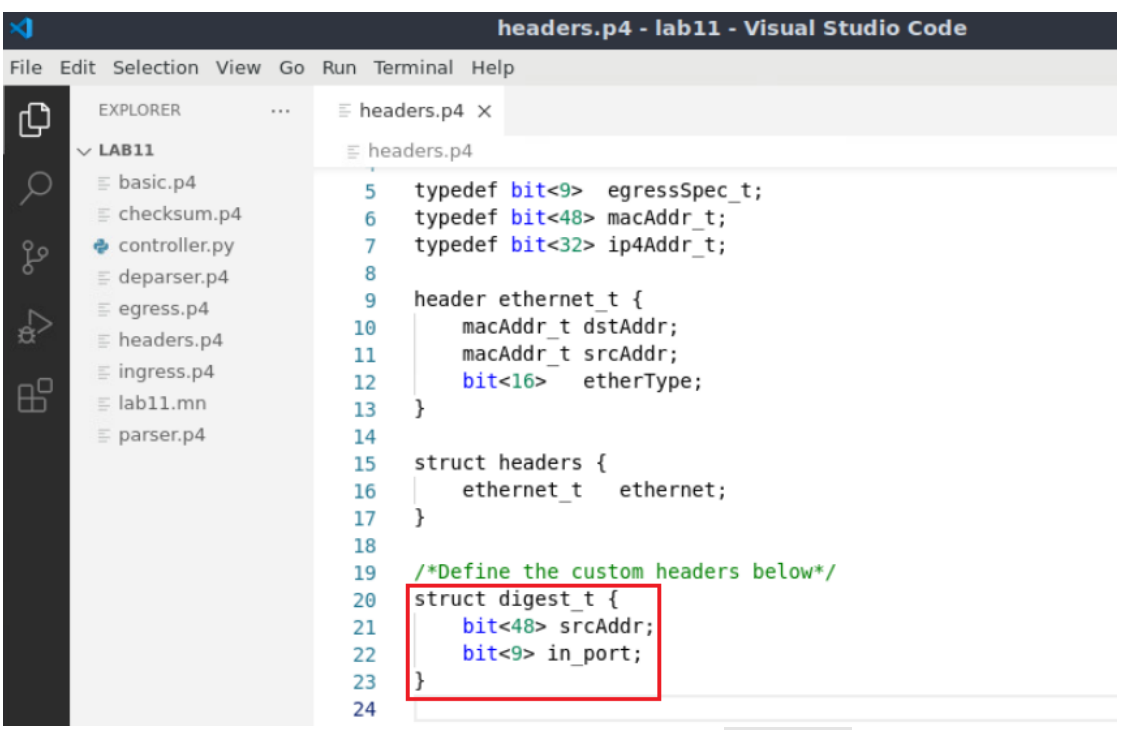


Figure 12. Defining the custom header `digest_t`.

The header type in the figure above contains the source MAC address and the ingress port.

Step 3. Define the following metadata structure by adding the code shown below.

```
struct metadata {
   digest_t mac_learn_digest;
}
```

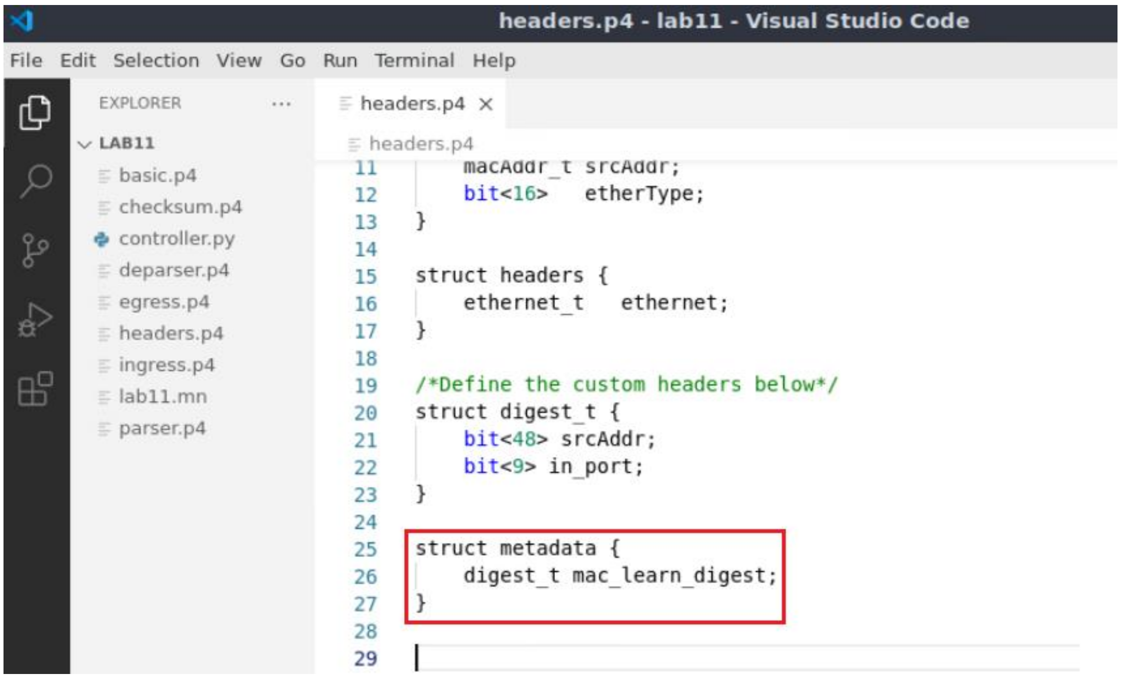


Figure 13. Defining the custom metadata struct.

The metadata defined in the figure above contains the custom header `mac_learn_digest` used to capture the source MAC address and ingress port.

Step 4. Press `Ctrl+s` to save the changes.

3.3 Programming the ingress pipeline

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. You will observe that the forwarding logic (i.e., the forwarding table, the actions, the apply block) is already defined.

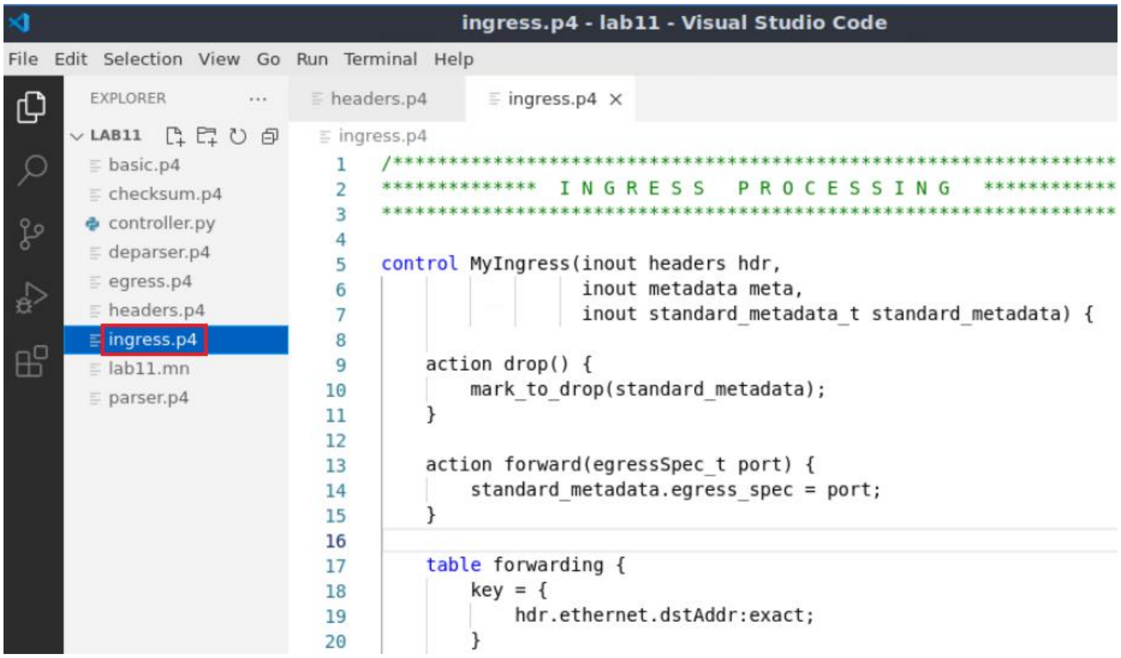


Figure 14. Inspecting the *ingress.p4* file.

Step 2. Define the action `learn_mac` by adding the following code.

```
action learn_mac() {
     meta.mac_learn_digest.srcAddr = hdr.ethernet.srcAddr;
     meta.mac_learn_digest.in_port = standard_metadata.ingress_port;
     digest(1, meta.mac_learn_digest);
}
```

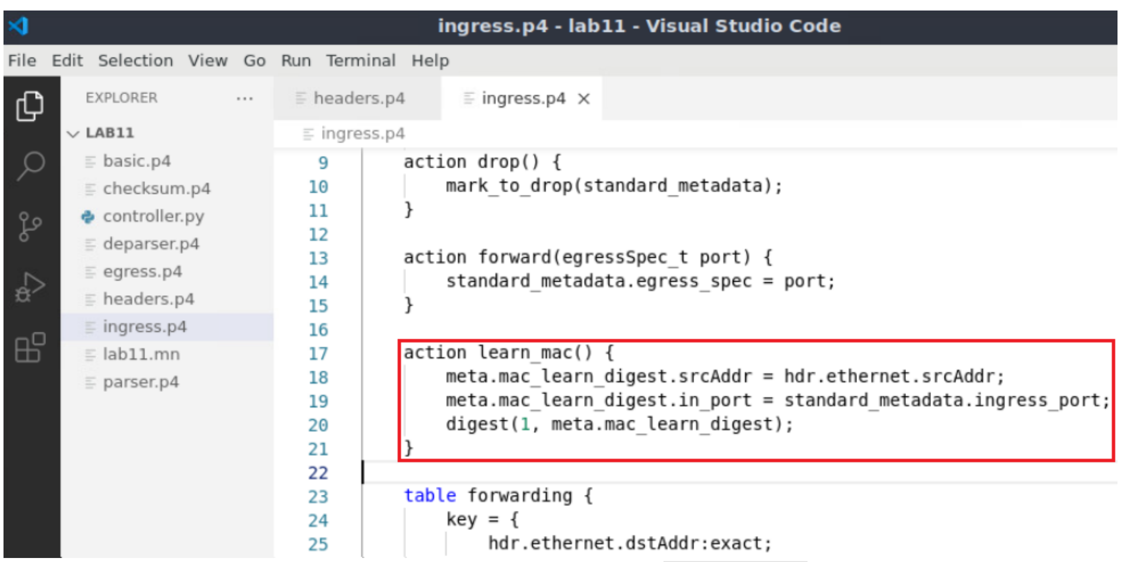


Figure 15. Defining the action `learn_mac`.

The code in the figure above is explained as follows:

- Line 17: defines the `learn_mac` action.
- Line 18: stores the source MAC address from an incoming packet into the custom header defined in the metadata.

- Line 19: stores the ingress port from an incoming packet into the custom header defined in the metadata.
- Line 20: sends a digest with content of the header `mac_learn_digest` to the control plane.

Step 3. Define the table `mac_learn` by adding the following code.

```
table mac_learn {
   key = {
       hdr.ethernet.srcAddr: exact;
   }
   actions = {
      learn_mac;
      NoAction;
   }
   size = 32;
   default_action = learn_mac();
}
```

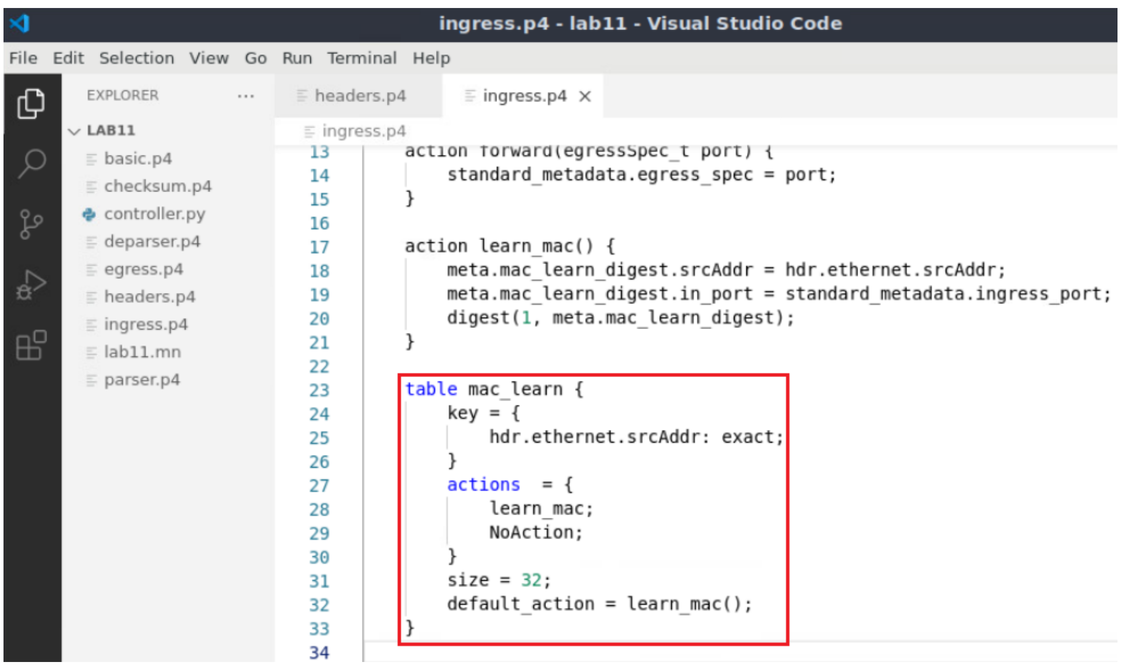


Figure 16. Inspecting the *ingress.p4* file.

The table in the figure above matches the source mac address and executes the actions `learn_mac` and `NoAction`.

Step 4. Add the following line to apply the table `mac_learn`.

```
mac_learn.apply();
```

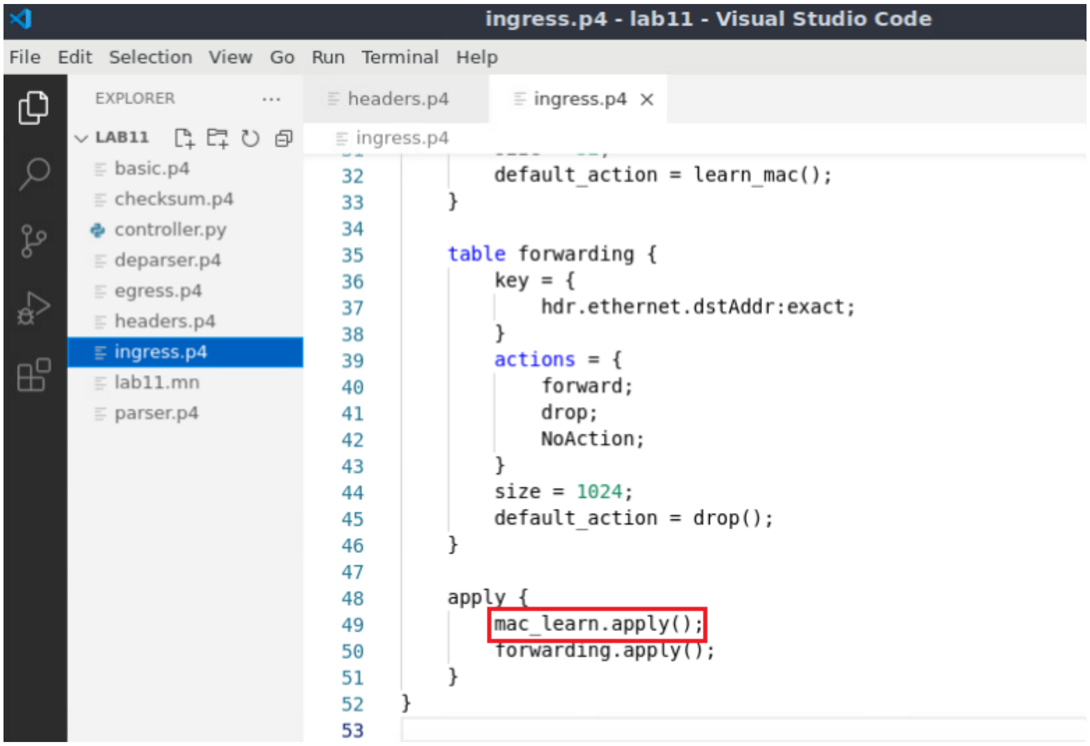


Figure 17. Applying the ingress pipeline logic.

Step 5. Press `Ctrl+s` to save the changes.

3.4 Creating the controller application

Step 1. Click on the *controller.py* file to display its content. Use the file explorer on the left-hand side of the screen to locate the file.

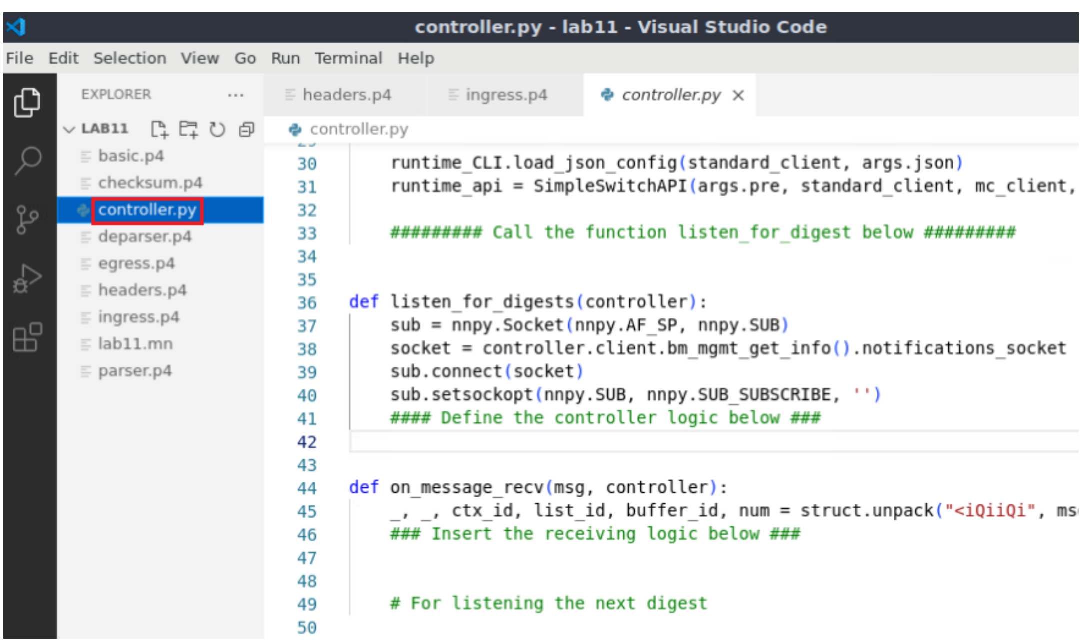


Figure 18. Inspecting the *controller.py* file.

Step 2. Scroll down to the function `listen_for_digests` and define the controller logic by adding the following lines.

```
while True:
    message = sub.recv()
    on_message_recv(message, controller)
```

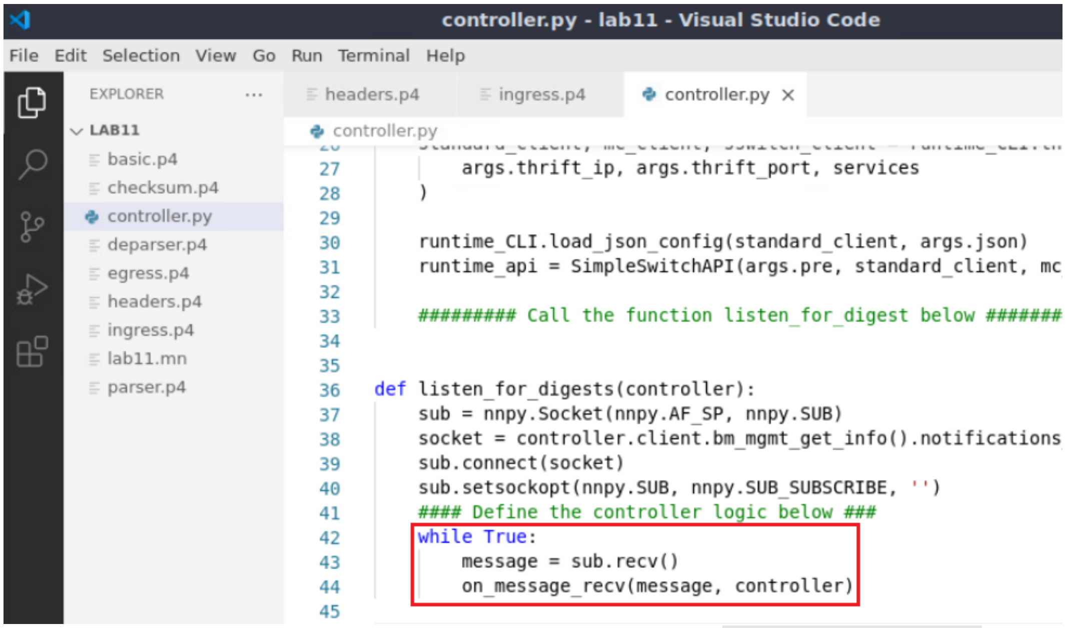


Figure 19. Defining the controller logic in the function `listen_for_digests`.

The code in the figure above implements a loop that listens for incoming digests (see line 43) and calls the `function on_message_recv` (see line 44). Note that function `sub.recv` will halt the execution until it receives a digest.

Step 3. Scroll down to the function `on_msg_recv` and define the following variables.

```
msg = msg[32:]
offset = 8
```

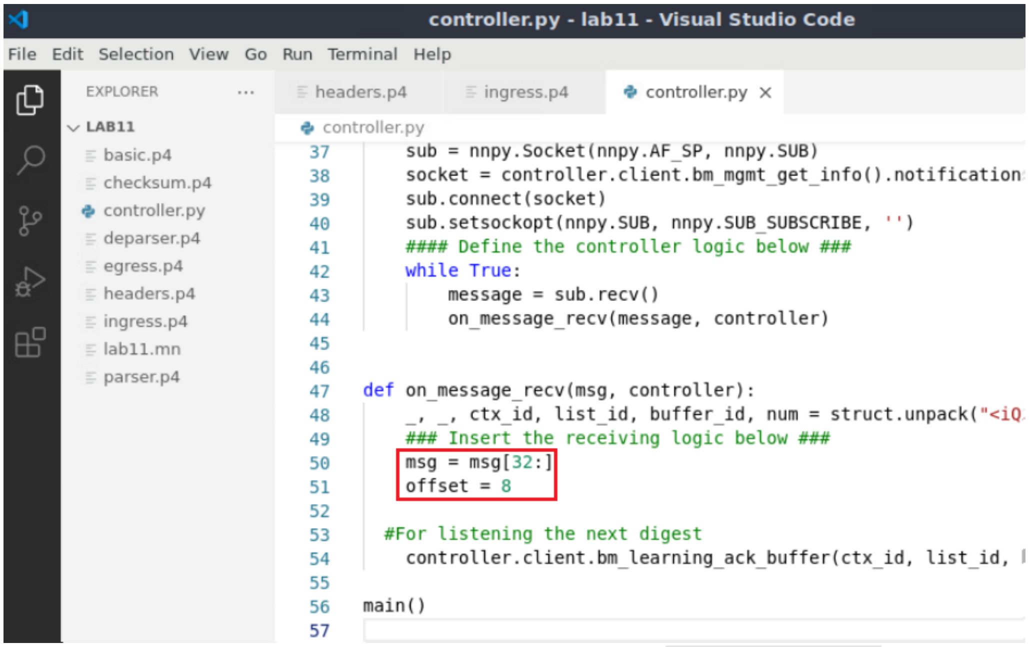


Figure 20. Defining variables in the function `on_message_recv`.

The variables in the figure above correspond to the digest and the offset. Note that the first 32 bytes are skipped because they store some metadata related to the digest and the switch. The offset value indicates the number of bytes corresponding to the MAC address (i.e., 48 bits) and the port number (i.e., 16 bits). Note that the 9-bits metadata `egressPort_t` represents the port number. However, this value is cast to a 16-bits variable.

Step 4. Define the receiving logic by adding the following code.

```
for m in range(num):
       mac1, mac2, port = struct.unpack("!LHH", msg[0:offset])
       mac_address = (mac1 << 16) + mac2
       print("mac address:", str(mac_address), 'port:', str(port))
       msg = msg[offset:]
       controller.do_table_add("mac_learn NoAction "
                              + str(mac_address) + " => ")
       print("forwarding forward " + str(mac_address) +" =>" + str(port))
       controller.do_table_add("forwarding forward "
                              + str(mac_address) +" => " + str(port) + "  ")
```

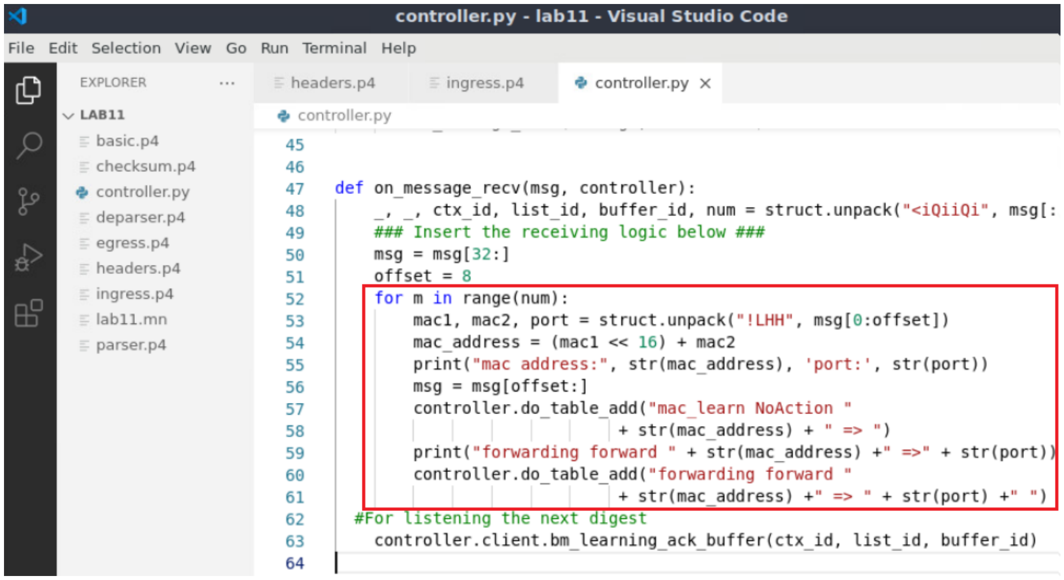


Figure 21. Defining receiving logic.

The code in the figure above is explained as follows:

- Line 53: Unpacks from the digest the source MAC address and ingress port. Note that the MAC address has 48 bits, thus, the value is stored in a 16-bits variable (i.e., `mac1`) and a 32-bits variable (i.e., `mac2`). Note that these values are contained in the first 8 bytes of the variable `msg`.
- Line 54: Shifts to the left 16-bits of `mac1` and `mac2`. The result is stored in `mac_address`.
- Line 55: Prints the received MAC address and ingress port.
- Line 56: Points to the next 8 bytes in `msg` to avoid reading the same digest in case there are two or more messages sent to the control plane simultaneously (see Figure 22).
- Line 57-58: Adds an entry to the table `mac_learn` that matches the MAC address and executes the action `NoAction`.
- Line 59: Prints the entry to be added to the table `forwarding`.
- Line 60-61: Adds an entry to the table `forwarding` that matches the MAC address and executes the action `forward`. The action data is the ingress port `port`.

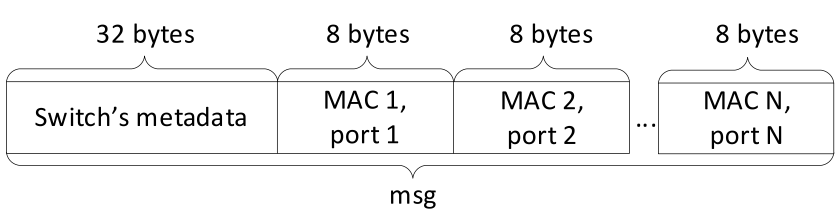


Figure 22. Defining receiving logic.

The figure above explains the data contained in the variable `msg`. This variable stores the digest sent from the data plane. The first 32 bytes contain the switch's metadata, followed by 8 bytes chunks that include the new MAC address (i.e., MAC 1, MAC 2, …, MAC N) and the ingress port numbers (i.e., port 1, port 2, …, port N).

Step 5. Scroll up and call the function `listen_for_digests` from the main function by adding the line below.

```
listen_for_digests(runtime_api)
```

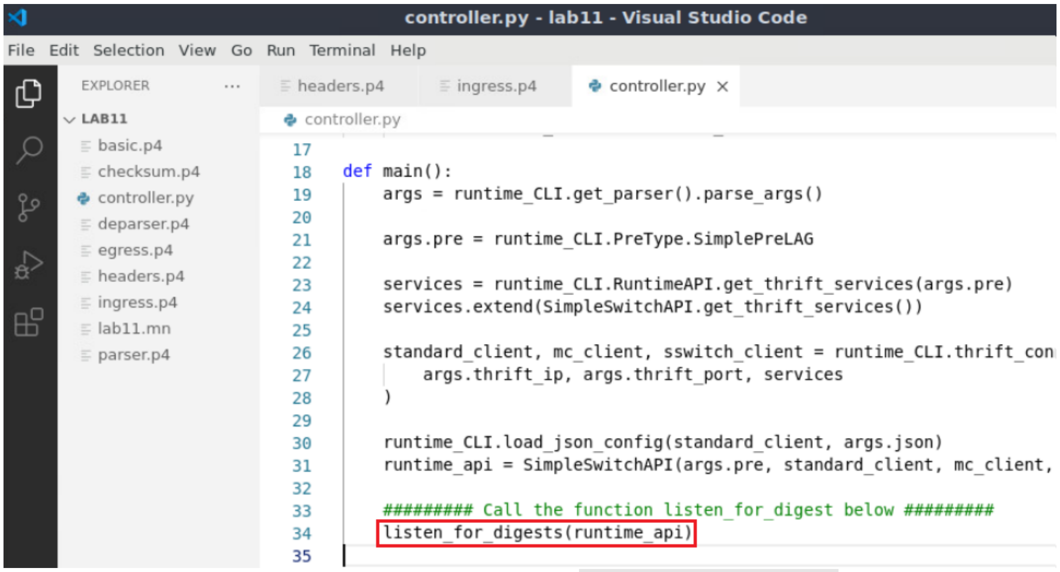


Figure 23. Calling the function `listen_for_digests`.

Step 6. Press `Ctrl+s` to save the changes.

4 Loading the P4 program

In this section, you will compile and load the P4 binary and the controller program in switch s1. You will also verify that the files reside in switch filesystem.

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside VS Code to compile the program.

```
p4c basic.p4
```

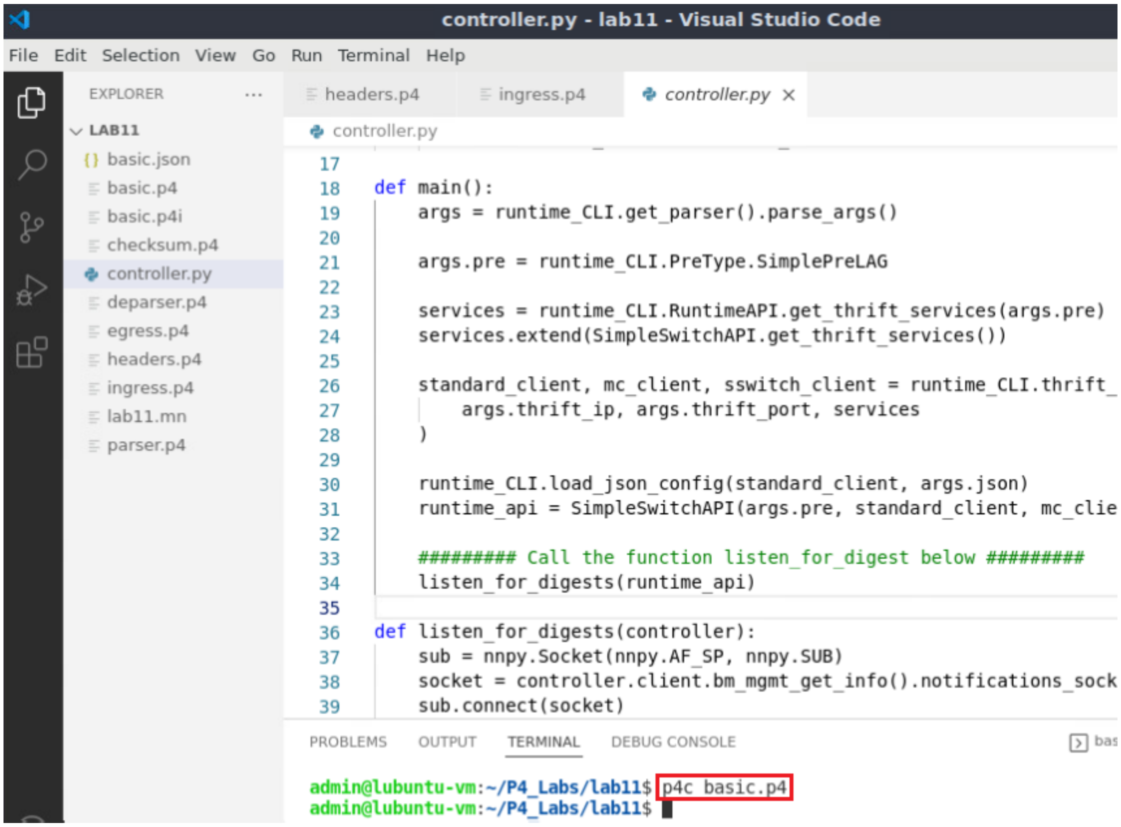


Figure 24. Compiling a P4 program.

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
```

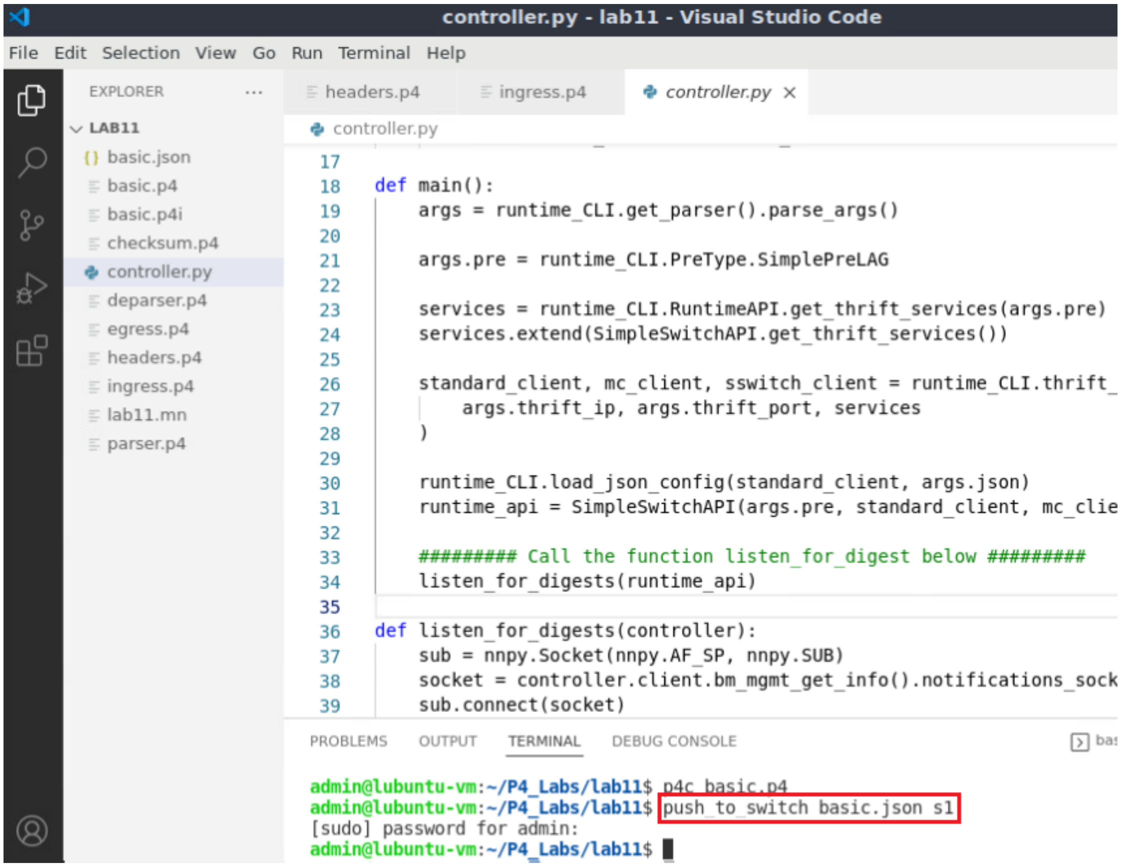


Figure 25. Pushing the *basic.json* file to switch s1.

Step 3. Type the command below in the terminal panel to push the *controller.py* file to the switch s1's filesystem.

```
push_to_switch controller.py s1
```

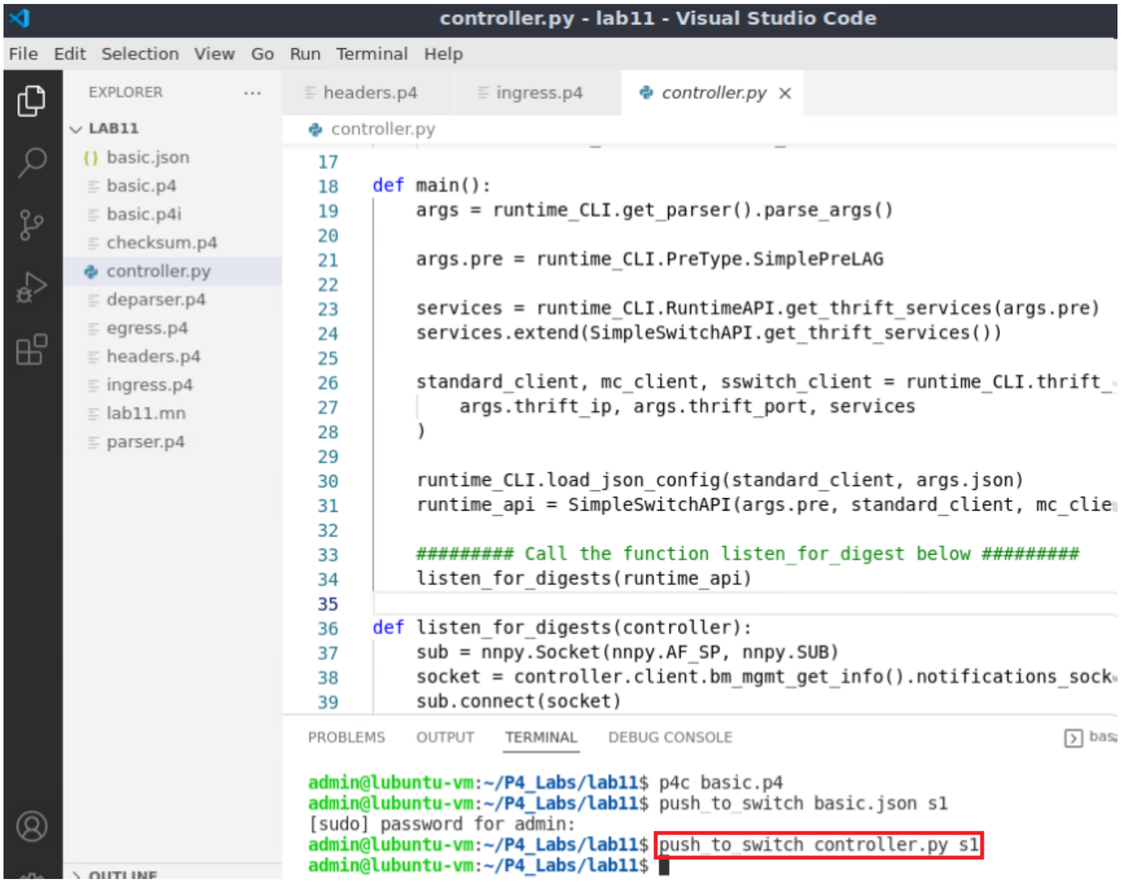


Figure 26. Pushing the *controller.py* file to switch s1.

4.2 Verifying the configuration

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

Figure 27. Maximizing the MiniEdit window.

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

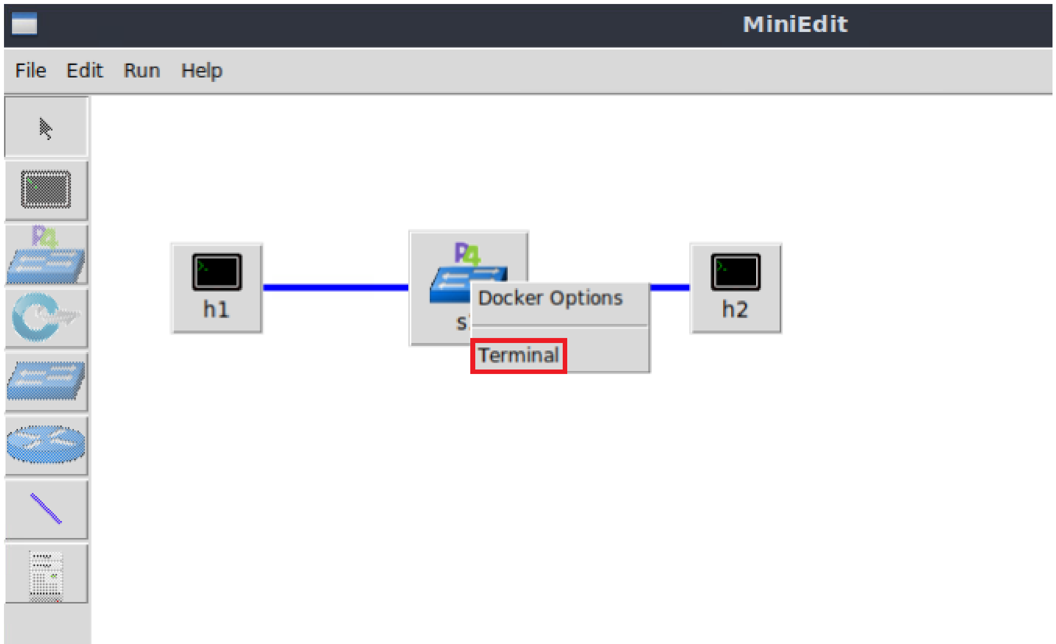


Figure 28. Starting the terminal on the switch.

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
```

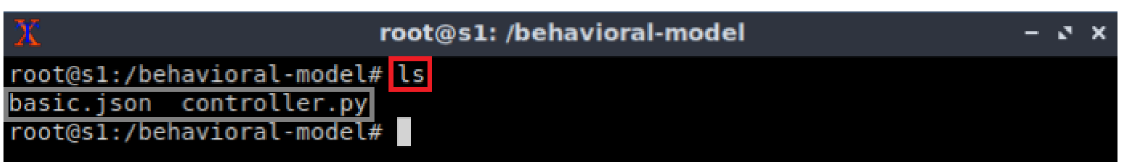


Figure 29. Displaying the contents of the current directory in the switch s1.

The figure above shows that the switch contains the *basic.json* and *controller.py* files that were pushed after compiling the P4 program and creating the controller application.

5 Configuring switch s1

In this section, you will map switch s1 interfaces to the ports in the P4 program and start the switch daemon. Then, you will load the rules to populate the match action tables.

5.1 Mapping the P4 program's ports

Step 1. Start the switch daemon by typing the following command.

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

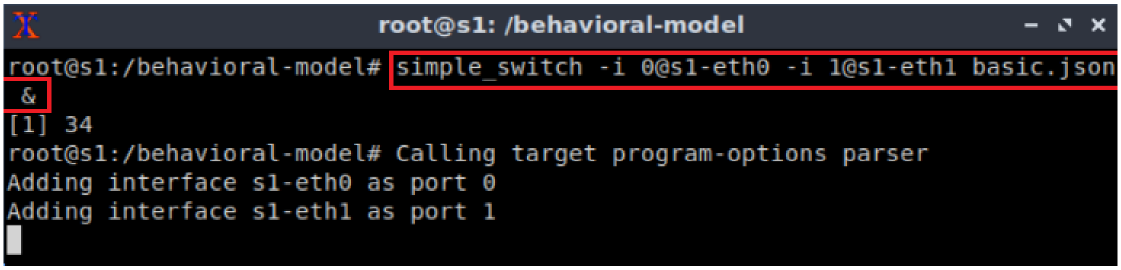


Figure 30. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

5.2 Loading the rules to the switch

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

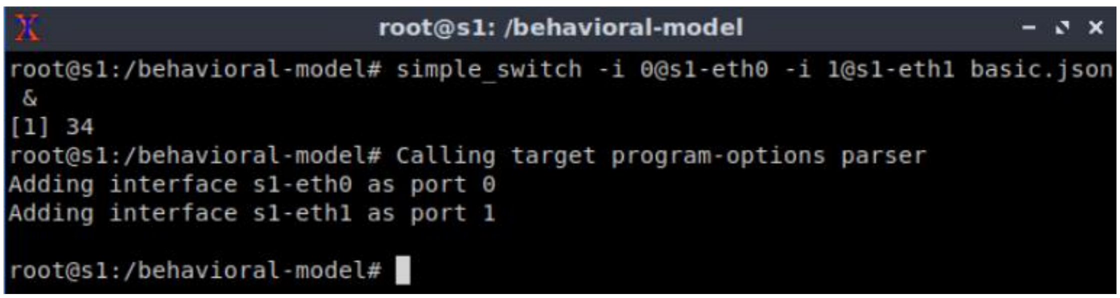


Figure 31. Returning to switch s1 CLI.

6 Testing and verifying the P4 program

This section shows the steps run a controller and observe how the MAC learning application populates the forwarding table in switch s1.

6.1 Starting the controller application

Step 1. In switch s1 terminal, start the controller by running the following command.

```
python controller.py
```

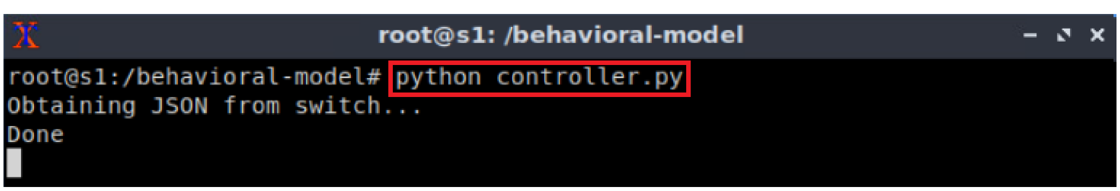


Figure 32. Starting the controller in switch s1.

6.2 Sending a packet from host h1 to host h2

Step 1. On host h1's terminal, type the following command.

```
send.py 10.0.0.2 HelloWorld
```

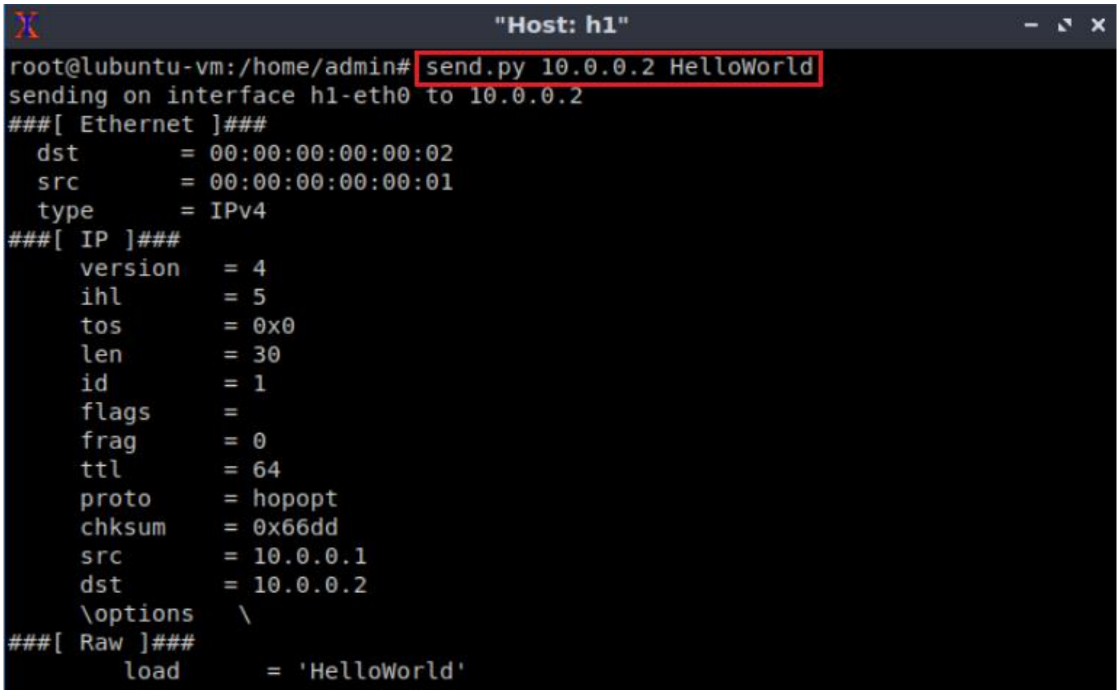


Figure 33. Sending a packet from host h1 to host h2.

Step 2. Go back to switch s1 terminal and inspect the output.

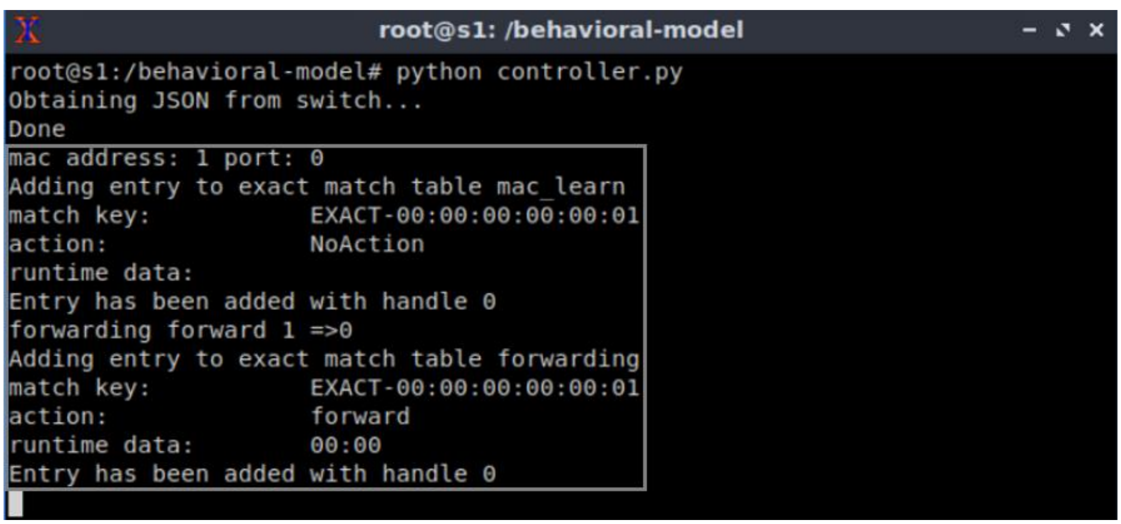
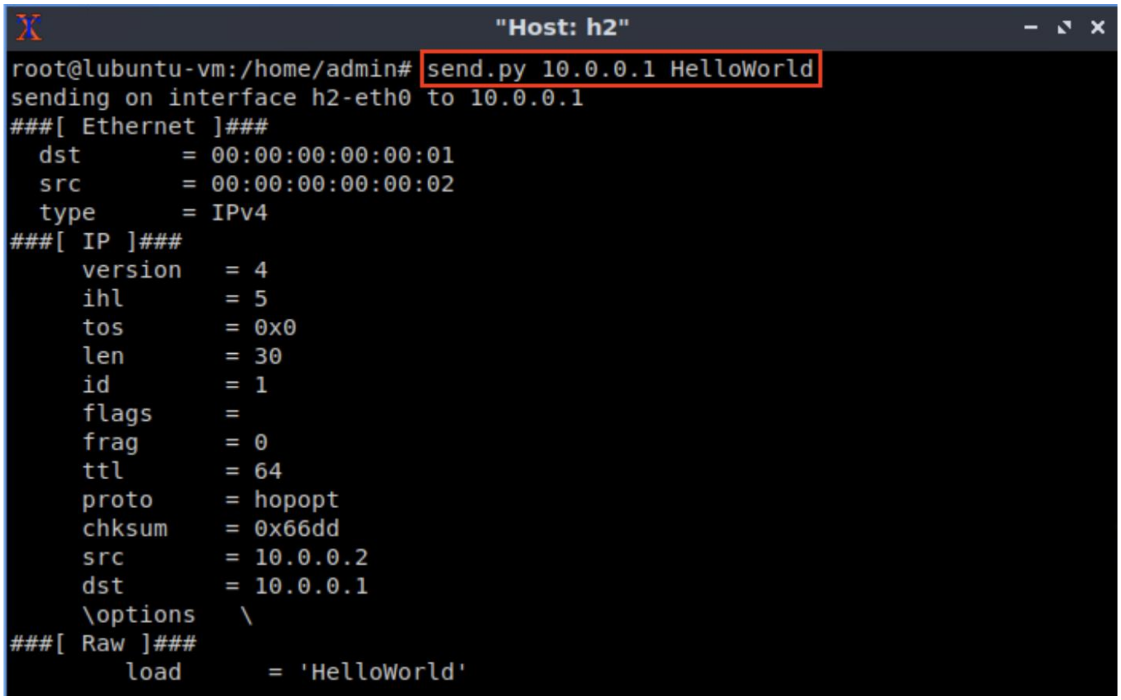


Figure 34. Inspecting the controller's log in switch s1.

6.3 Sending a packet from host h2 to host h1

Step 1. On host h2's terminal, type the following command.

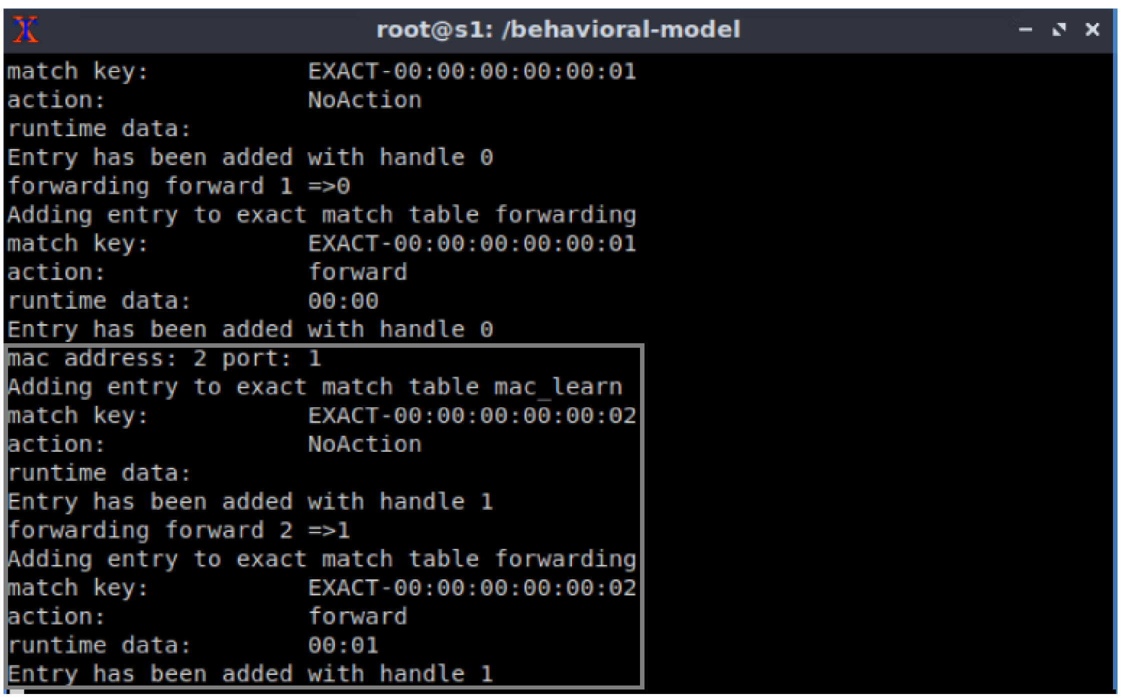
```
send.py 10.0.0.1 HelloWorld
```



```
"Host: h2"
root@lubuntu-vm:/home/admin# send.py 10.0.0.1 HelloWorld
sending on interface h2-eth0 to 10.0.0.1
###[ Ethernet ]###
  dst       = 00:00:00:00:00:01
  src       = 00:00:00:00:00:02
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 30
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = hopopt
     chksum    = 0x66dd
     src       = 10.0.0.2
     dst       = 10.0.0.1
     \options   \
###[ Raw ]###
        load      = 'HelloWorld'
```

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

Step 2. Go back to switch s1 terminal and inspect the output.



```
root@s1: /behavioral-model
match key:           EXACT-00:00:00:00:00:01
action:              NoAction
runtime data:
Entry has been added with handle 0
forwarding forward 1 =>0
Adding entry to exact match table forwarding
match key:           EXACT-00:00:00:00:00:01
action:              forward
runtime data:        00:00
Entry has been added with handle 0
mac address: 2 port: 1
Adding entry to exact match table mac_learn
match key:           EXACT-00:00:00:00:00:02
action:              NoAction
runtime data:
Entry has been added with handle 1
forwarding forward 2 =>1
Adding entry to exact match table forwarding
match key:           EXACT-00:00:00:00:00:02
action:              forward
runtime data:        00:01
Entry has been added with handle 1
```

Figure 36. Inspecting the controller's log in switch s1.

6.4 Verifying connectivity between host h1 and host h2

Step 1. Go back to host h2 and start the receiver by issuing the following command.

```
recv.py
```

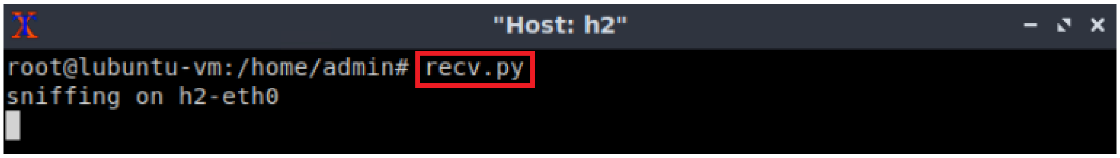


Figure 37. Starting the receiver in host h2.

Step 2. Go back to host h2 and start the receiver by issuing the following command.

```
send.py 10.0.0.2 HelloWorld
```

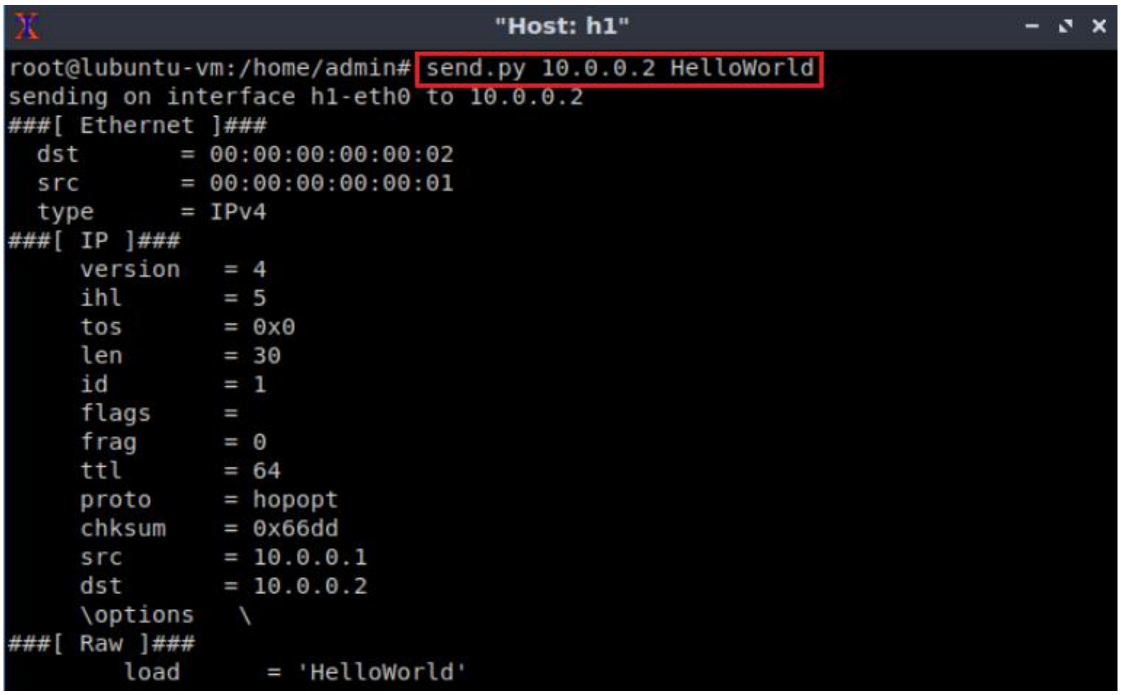


Figure 38. Sending a packet from host h1 to host h2.

Step 3. Inspect the output on host h2 to verify that the packet was received.

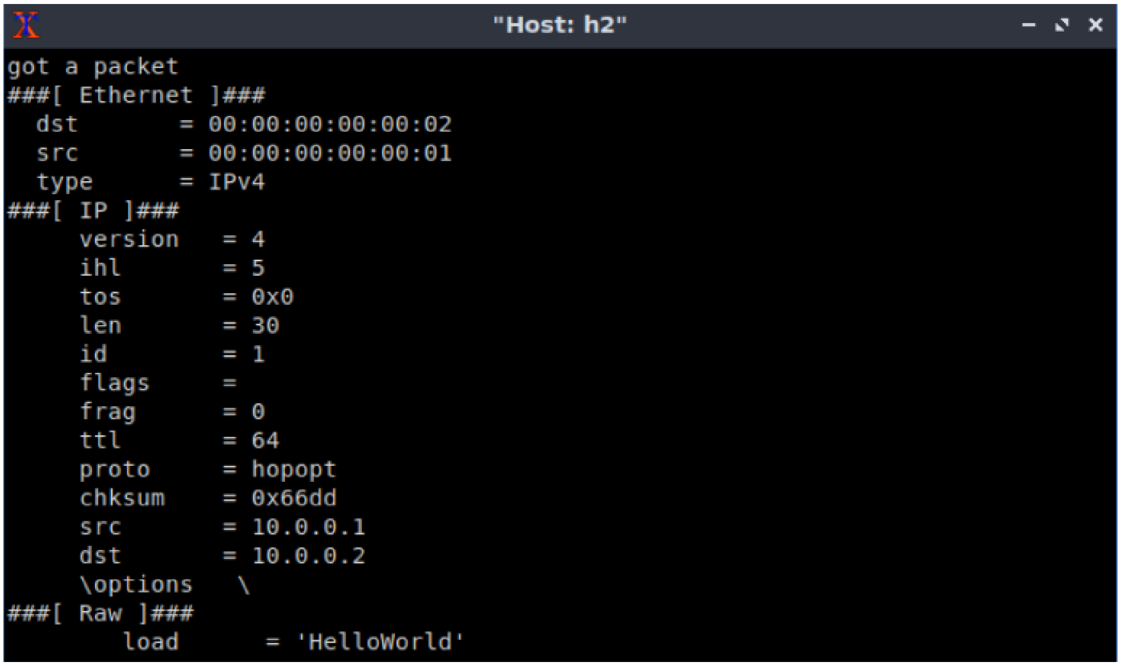


Figure 39. Inspecting the output in host h2.

6.5 Verifying the rules in the control plane

Step 1. Go back to switch s1 terminal and press `Ctrl+c` to stop the controller.

Step 2. Issue the following command to start the CLI.

```
simple_switch_CLI
```

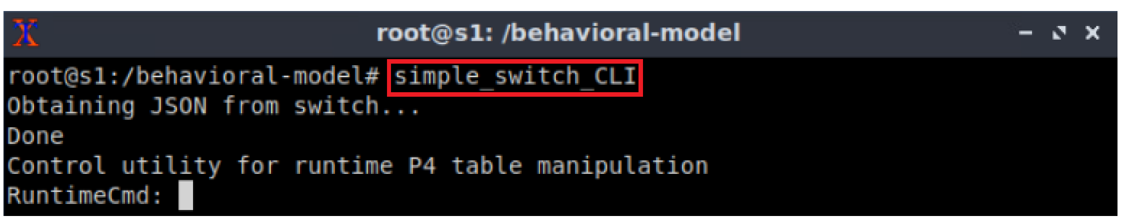


Figure 40. Starting the switch CLI.

Step 3. Issue the following command to see content of the table `forwarding`.

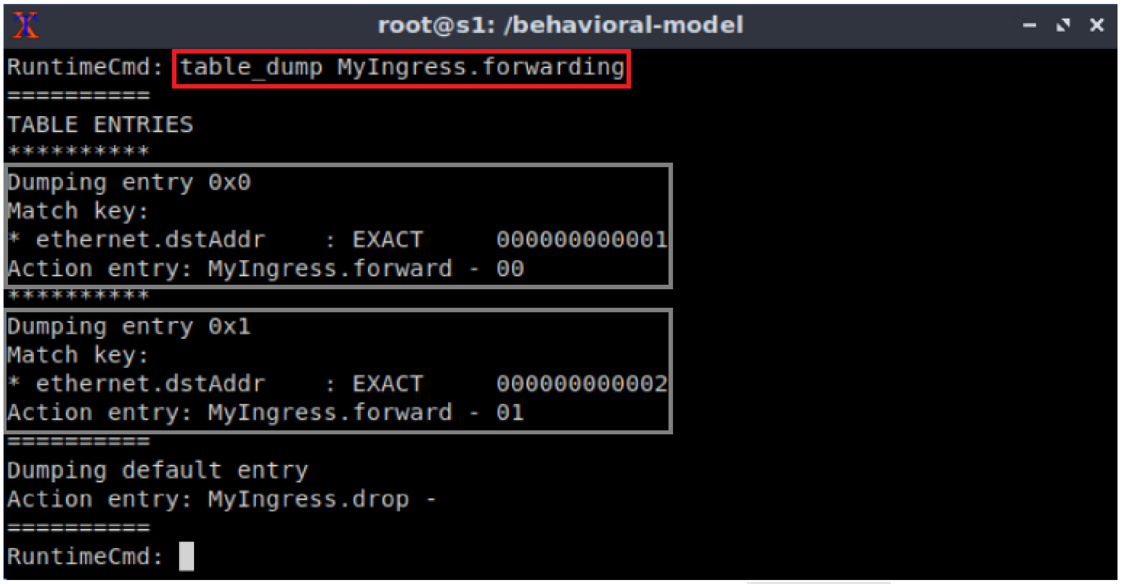


Figure 41. Showing the content of the table `forwarding`.

The output in the figure above shows that the table `forwarding` was populated with two entries. Entry 1 matches packets with destination MAC address 00:00:00:00:00:01 and forwards them through port 0. Similarly, entry 2 matches packets with destination MAC address 00:00:00:00:00:02 and forwards them through port 1.

Step 4. Issue the following command to show the content of the table `mac_learn`.

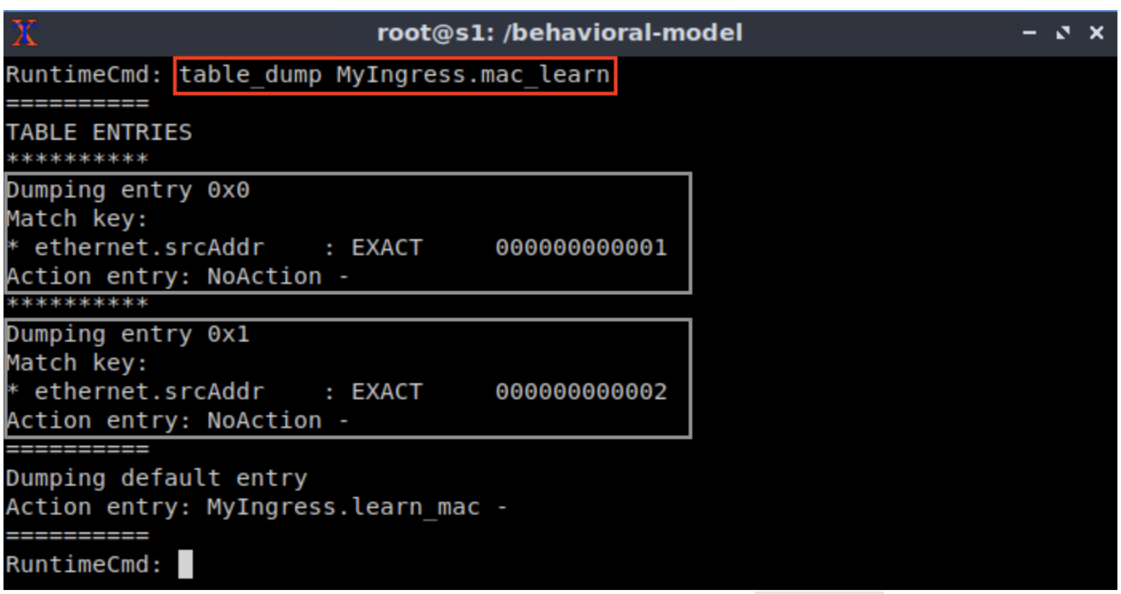


Figure 42. Showing the content of the table `mac_learn`.

Note that the table `mac_learn` matches the destination mac address and executes the action `NoAction`. The logic of this table consists of applying the default action `learn_mac` when there is a new MAC address to learn.

This concludes lab 11. Stop the emulation and then exit out of MiniEdit.

References

1. The P4 language Consortium. "*Behavioral model: The runtime CLI application.*" [Online]. Available: https://tinyurl.com/28fptt6z
2. The P4 Architecture Working Group. "*$P4_{16}$ Portable Switch Architecture (PSA)*." [Online]. Available: https://tinyurl.com/2wnkc6d2
3. Mininet walkthrough. [Online]. Available: http://Mininet.org.
4. 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.
5. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
6. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.