



INTRODUCTION TO P4 PROGRAMMABLE DATA PLANES

Lab 2: Introduction to P4 and BMv2

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



Award 2118311

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

Contents

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

Overview

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

Objectives

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

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

Lab settings

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

Table 1. Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Loading the P4 program.
4. Section 4: Configuring switch s1.

1 Introduction

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

hardware and software. The progressive reduction in the flexibility of protocol design caused by standardized requirements, which cannot be easily removed to enable protocol changes, has perpetuated the status quo. This protocol ossification^{1, 2} has been characterized by a slow innovation pace at the hand of few network vendors. As an example, after being initially conceived by Cisco and VMware³, 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 language⁵, which stands for Programming Protocol-independent Packet Processors. Essentially, P4 programmable switches have removed the entry barrier to network design, previously reserved to network vendors.

1.1 Workflow of a P4 program

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

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

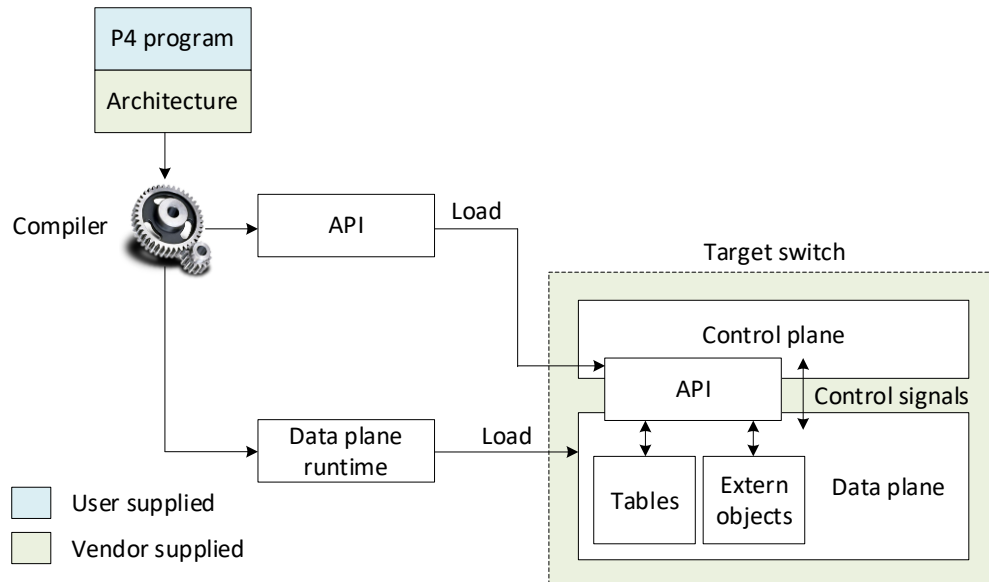


Figure 1. Generic workflow design. The compiler, the architecture model, and the target switch are provided by the vendor of the device. The P4 source code is customized by the user. The compiler generates a data plane runtime to be loaded into the target, and the APIs used by the control plane to communicate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

1.2 Workflow used in this lab series

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

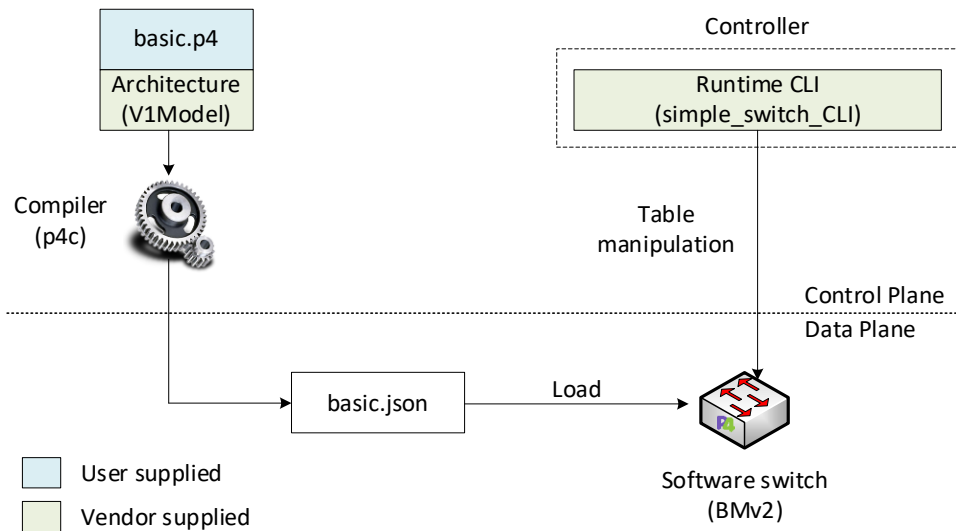


Figure 2. Workflow used in this lab series.

2 Lab topology

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

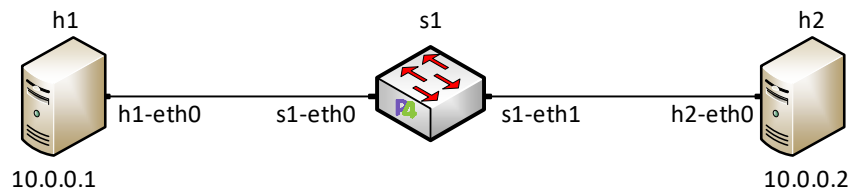


Figure 3. Lab topology.

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



Figure 4. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab2*, select the file *lab2.mn*, and click on *Open*.

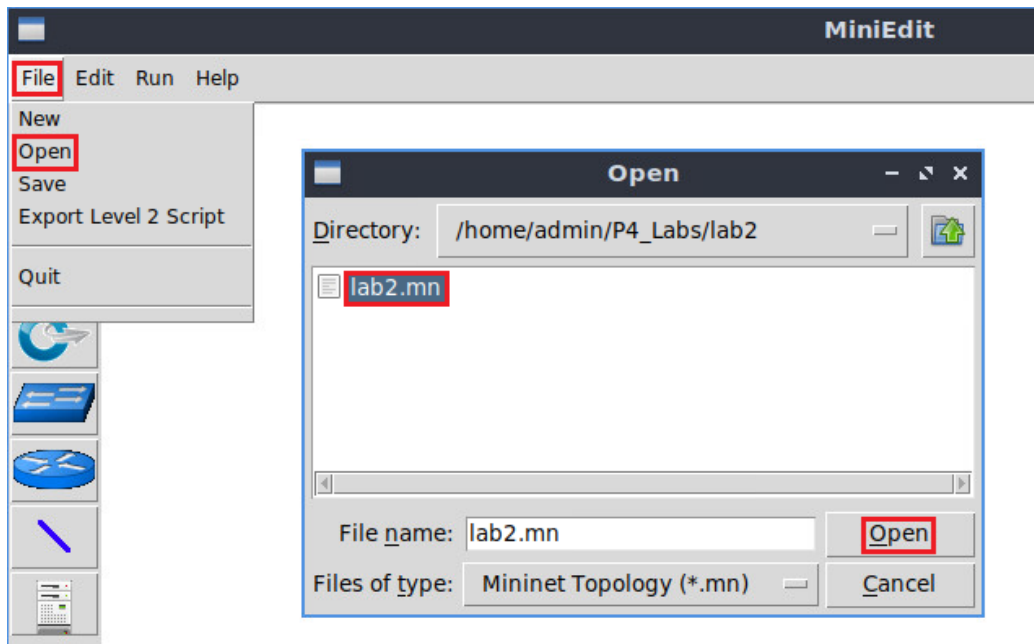


Figure 5. Opening a topology in MiniEdit.

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



Figure 6. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

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

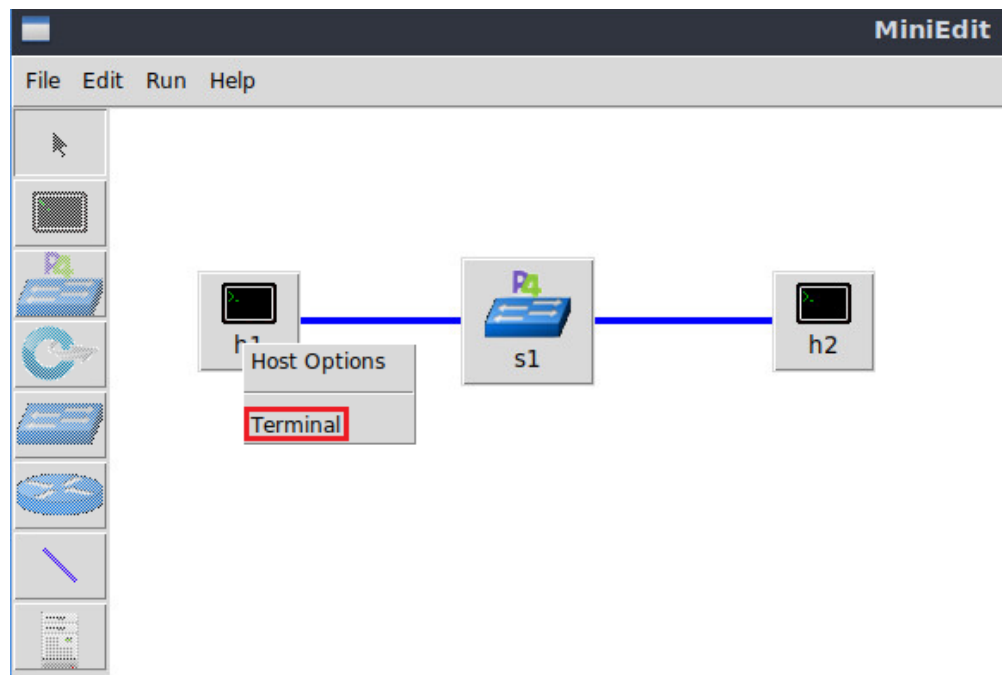


Figure 7. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```

```

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

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

```

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

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

3 Loading the P4 program

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

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

p4c supports both P4₁₄ and P4₁₆, but in this lab series we will only focus on P4₁₆ since it is the newer version and is currently being supported by major programming ASIC manufacturers⁷.

3.1 Loading the programming environment

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

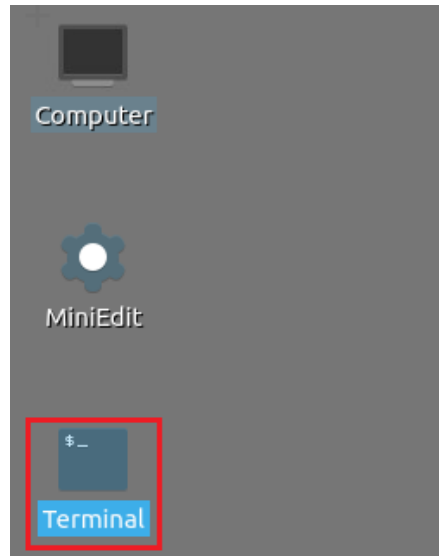


Figure 9. Shortcut to open a Linux terminal.

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.

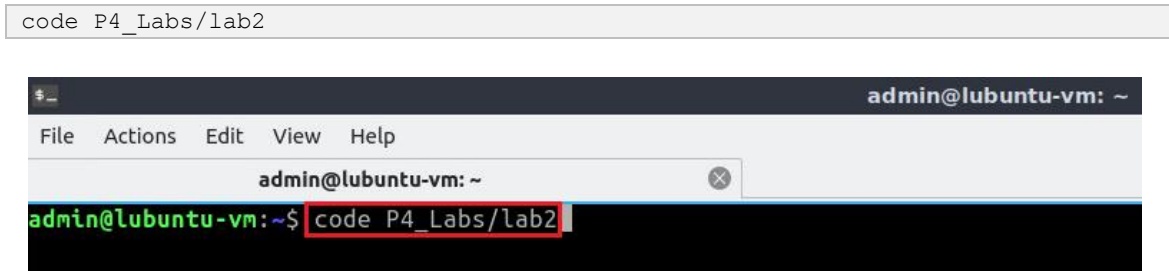


Figure 10. Launching the editor and opening the lab2 directory.

Step 3. Once the previous command is executed, VS Code will start. Click on *basic.p4* in the file explorer panel on the left hand side to open the P4 program in the editor.

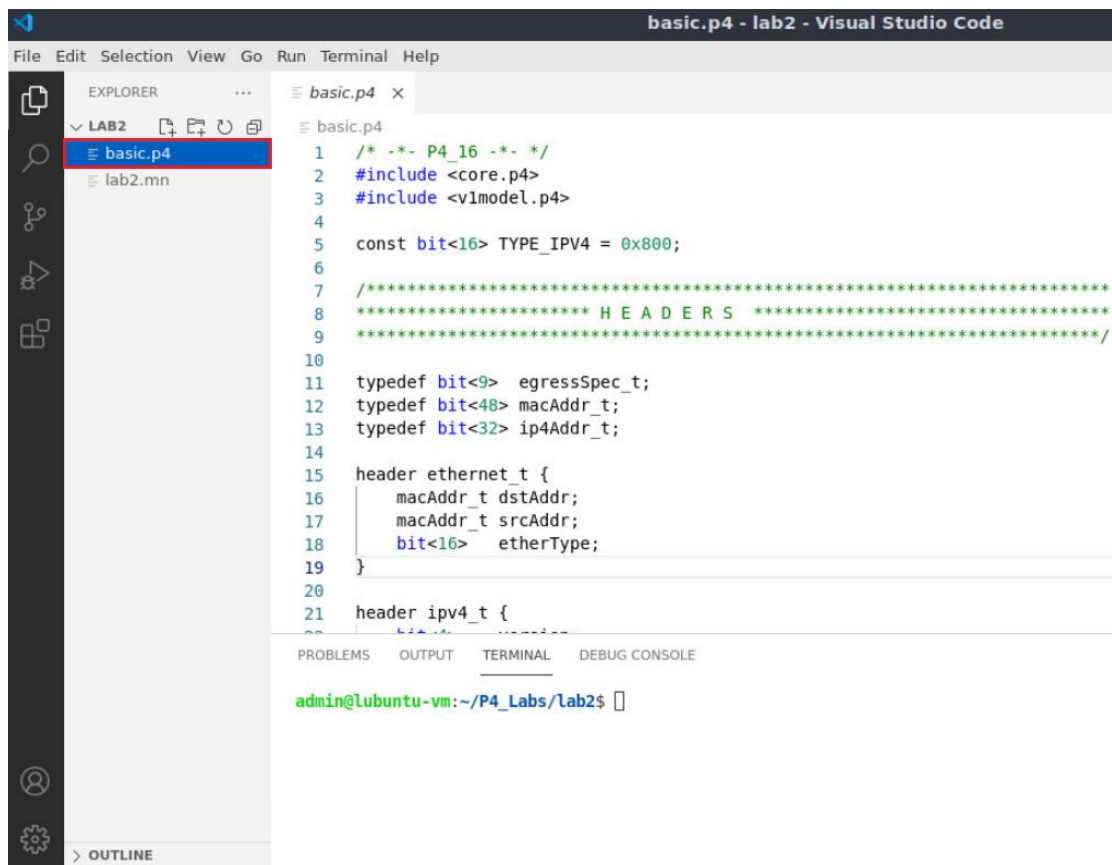


Figure 11. Opening the programming environment in VS Code.

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

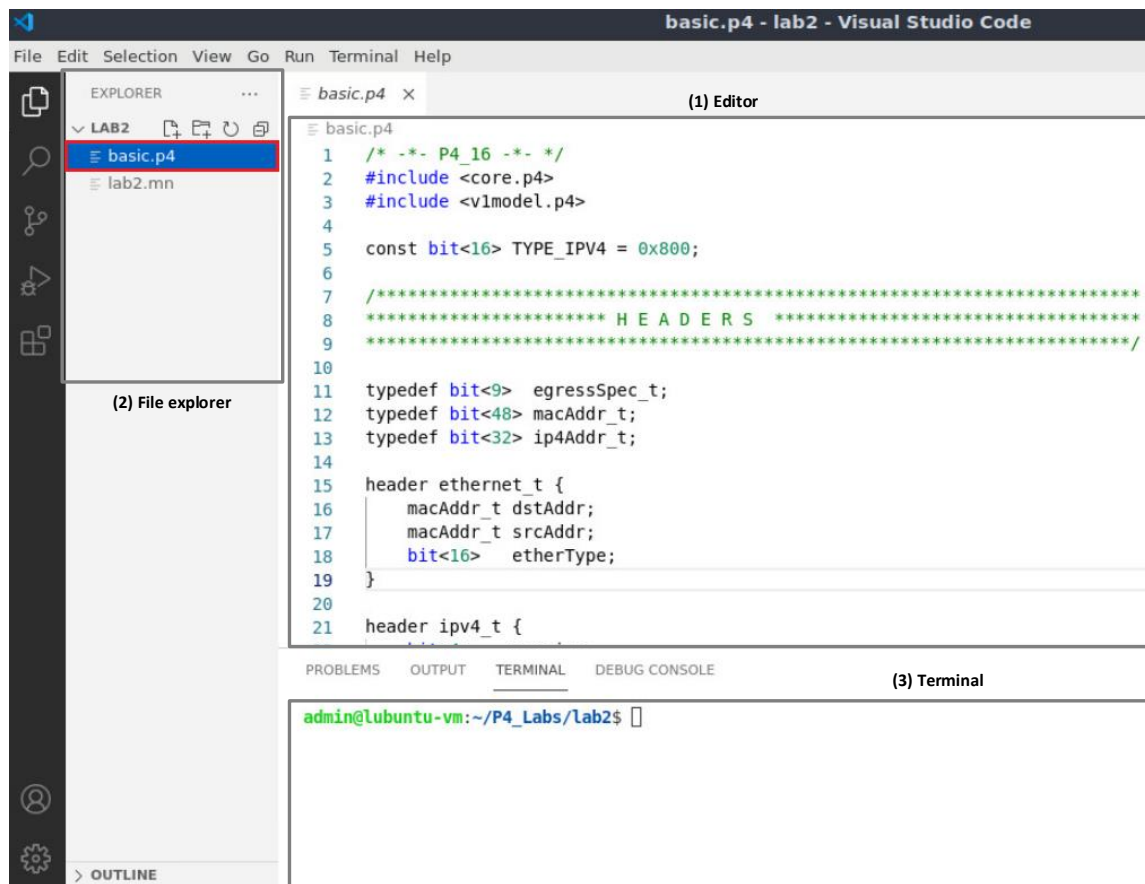


Figure 12. VS Code graphical interface components.

The VS Code interface consists of three main panels:

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

3.2 Compiling and loading the P4 program to switch s1

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

```
p4c basic.p4
```

```

1  /* -*- P4_16 -*- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  const bit<16> TYPE_IPV4 = 0x800;
6
7  /**
8   * ***** H E A D E R S *****
9   * *****
10
11  typedef bit<9> egressSpec_t;
12  typedef bit<48> macAddr_t;
13  typedef bit<32> ip4Addr_t;
14
15  header ethernet_t {
16      macAddr_t dstAddr;
17      macAddr_t srcAddr;
18      bit<16> etherType;
19  }

```

```

admin@lubuntu-vm:~/P4_Labs/lab2$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab2$

```

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
```

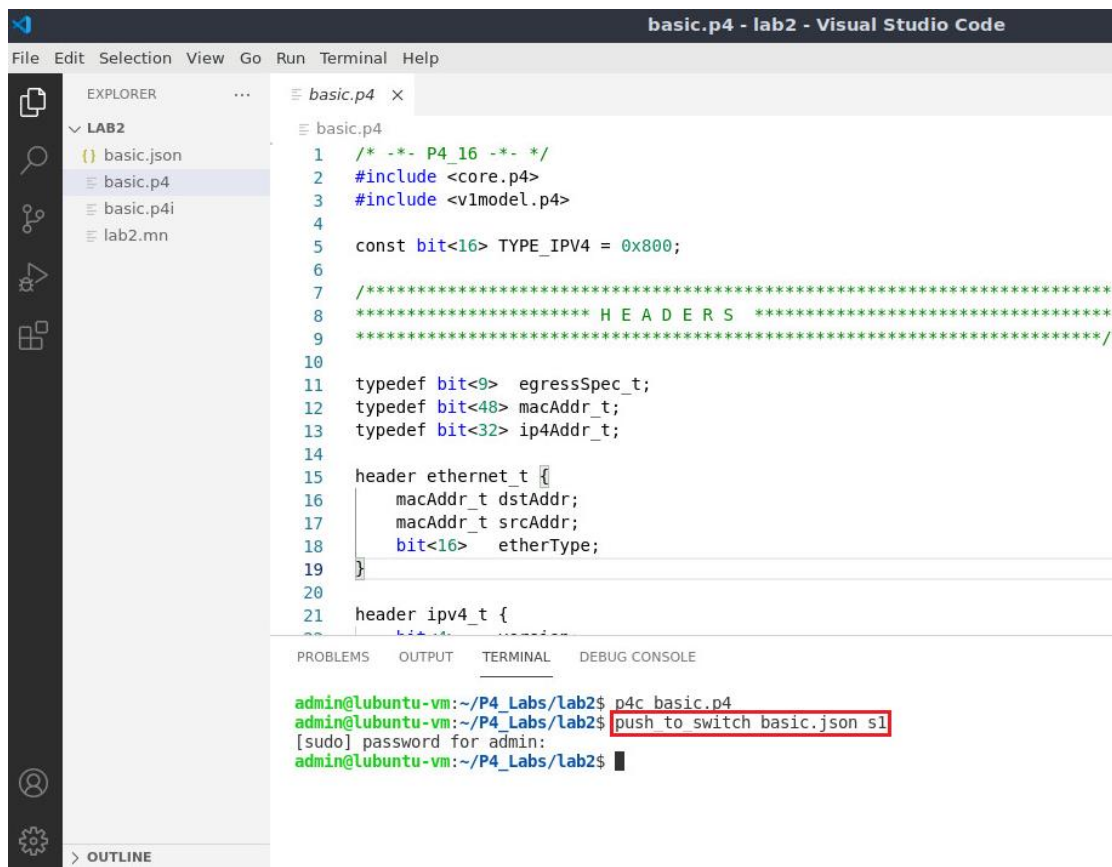


Figure 14. Downloading the compiled program to switch s1.

3.3 Verifying the configuration

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

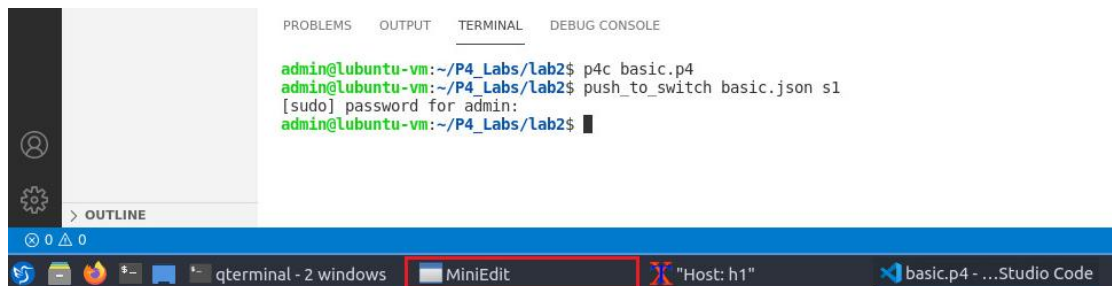


Figure 15. Maximizing the MiniEdit window.

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

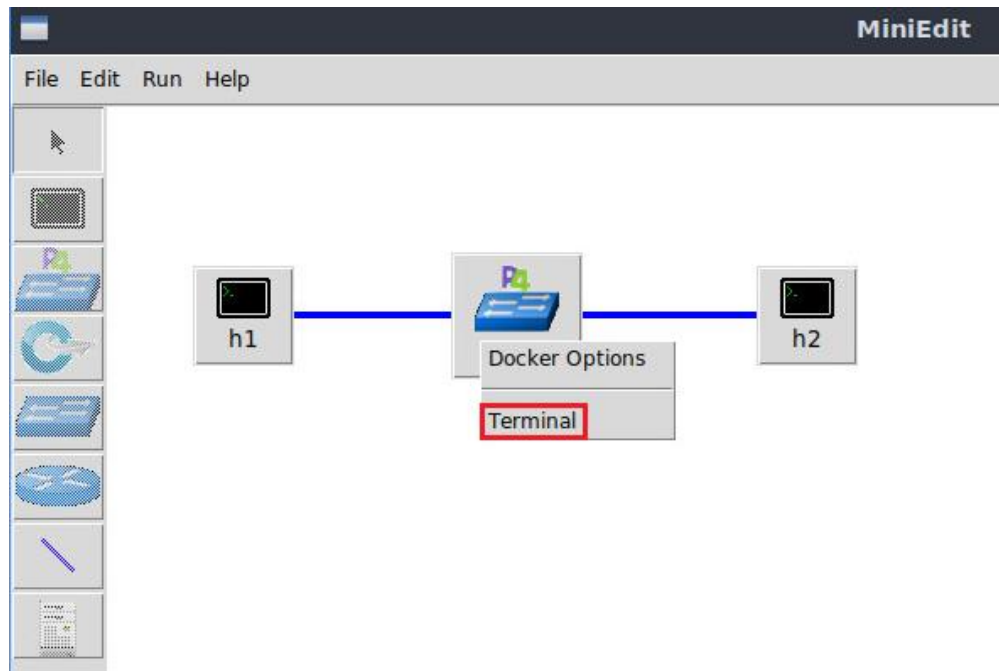


Figure 16. Starting the terminal on switch s1.

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
```

 The image shows a terminal window titled 'root@s1: /behavioral-model'. The prompt is 'root@s1:/behavioral-model#'. The user has entered 'ls' and pressed enter. The output is 'basic.json'. The 'ls' command and the output 'basic.json' are highlighted with red rectangular boxes.

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

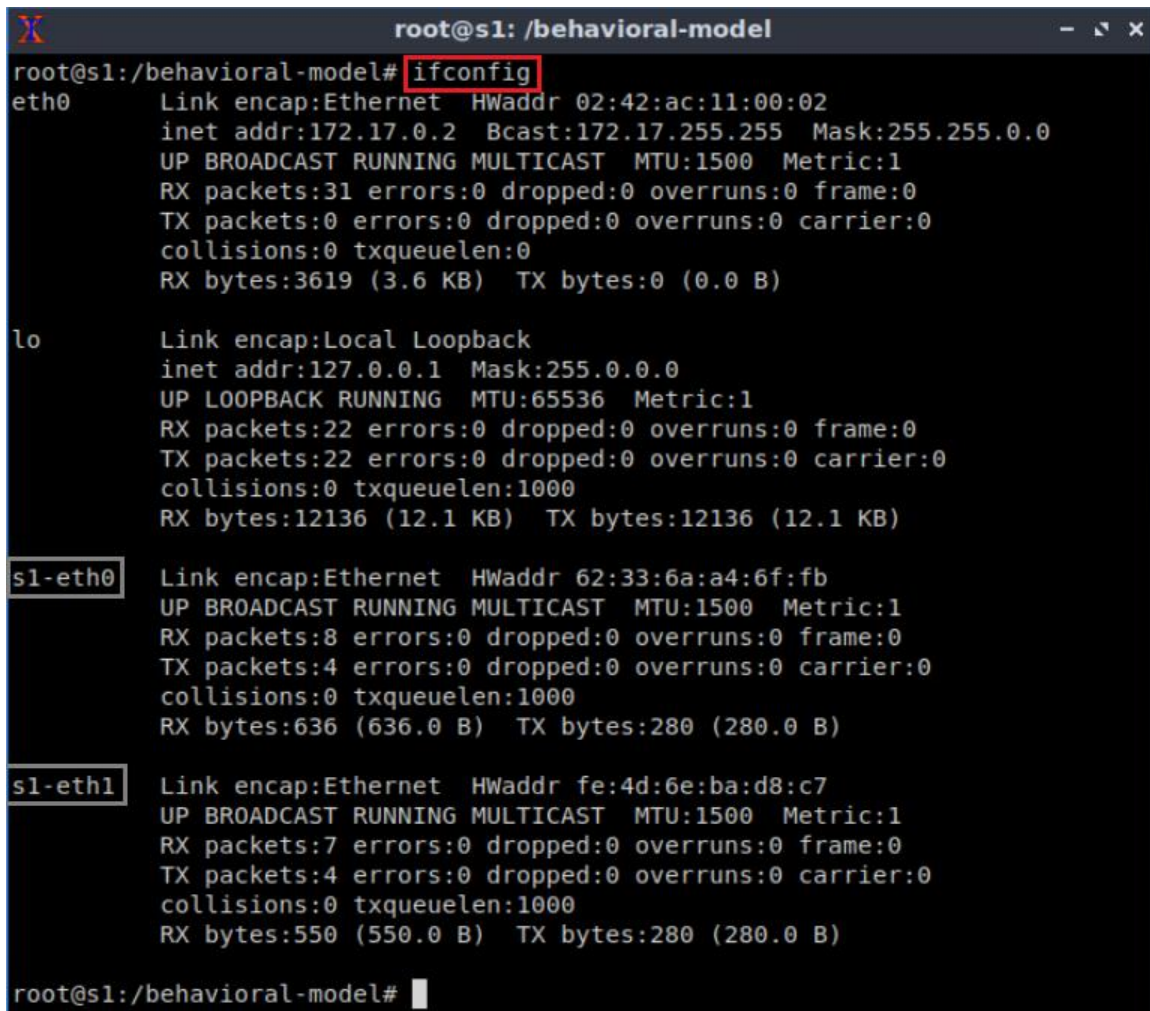
We can see that the switch contains the *basic.json* file that was downloaded after compiling the P4 program.

4 Configuring switch s1

4.1 Mapping P4 program's ports

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

```
ifconfig
```

```

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

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

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

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

root@s1:/behavioral-model#

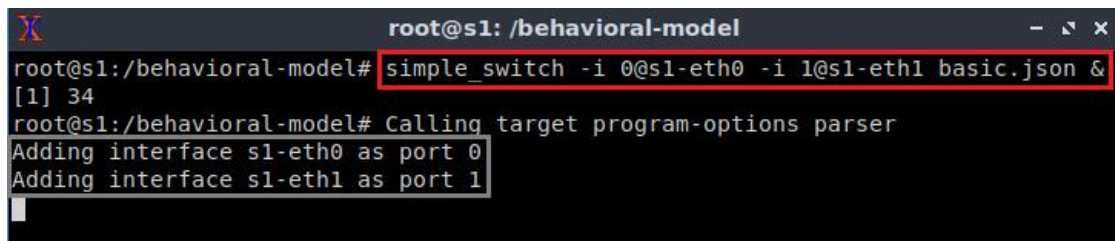
```

Figure 18. Displaying switch s1 interfaces.

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

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

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



```

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

```

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

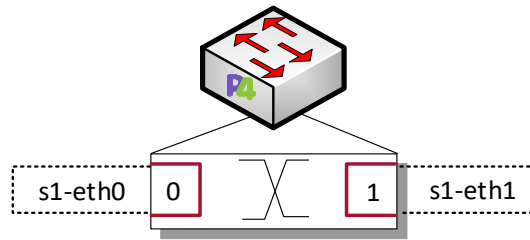


Figure 20. Ports 0 and 1 are mapped to the interfaces *s1-eth0* and *s1-eth1* of switch s1.

4.2 Loading the rules to the switch

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

```

root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
&
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
root@s1:/behavioral-model#

```

Figure 21. Returning to switch s1 CLI.

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

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

```

root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab2/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:      EXACT-00:00
action:         MyIngress.forward
runtime data:   00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:      EXACT-00:01
action:         MyIngress.forward
runtime data:   00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#

```

Figure 22. Loading table entries to switch s1.

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

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


```
ping 10.0.0.2 -c 4
```

```

Host: h1
root@ubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.851 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.062 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.078 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.085 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3055ms
rtt min/avg/max/mdev = 0.062/0.269/0.851/0.336 ms
root@ubuntu-vm:/home/admin#

```

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, "P4₁₆ for Intel Tofino™ using Intel P4 Studio™". 2021 P4 Workshop, ONF. [Online]. Available: <https://tinyurl.com/yckzkybf>.