UNIVERSITY OF
SOUTH CAROLINA

P4-PERFSONAR LAB SERIES

Book Version: 08-07-2024

Principal Investigator: Jorge Crichigno
Co-Principal Investigator: Elie Kfoury
Developers: Ali Mazloum, Jose Gomez

Contents

Lab 1: Introduction to Mininet
Lab 2: P4 Program Building Blocks
Lab 3: Measuring Flow's Throughput
Lab 4: Monitoring the RTT of TCP Flows using P4
Lab 5: Configuring Regular Tests Using pScheduler CLI
Lab 6: Connecting perfSONAR to Grafana Dashboard
Lab 7: Retrieving Per-flow Statistics from the Data Plane
Lab 8: Collecting P4 Measurements using perfSONAR's Archiver

P4-PERFSONAR LAB SERIES

Lab 1: Introduction to Mininet

Document Version: 06-12-2024

Contents

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

Overview

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

Objectives

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

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

Lab settings

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

Table 1**.** Credentials to access the Client machine.

Device	**Account**	**Password**
Client	admin	password

Lab roadmap

This lab is organized as follows:

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

1 Introduction to Mininet

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

- Fast prototyping for new networking protocols.

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

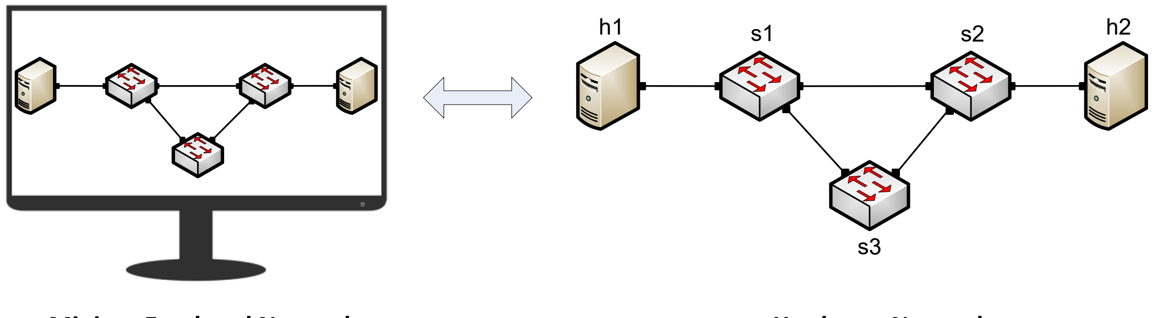


Figure 1. Hardware network vs. Mininet emulated network.

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

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

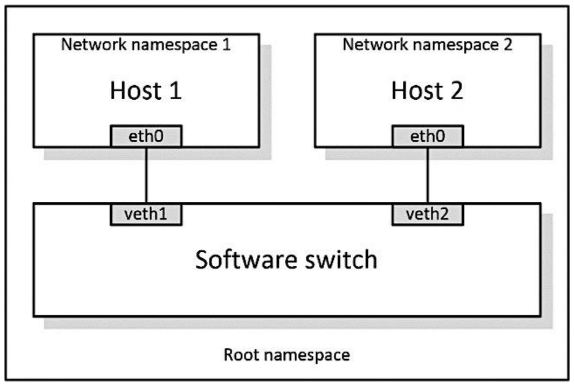


Figure 2. Network namespaces and virtual Ethernet links.

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

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

2 Invoke Mininet using the CLI

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

2.1 Invoke Mininet using the default topology

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

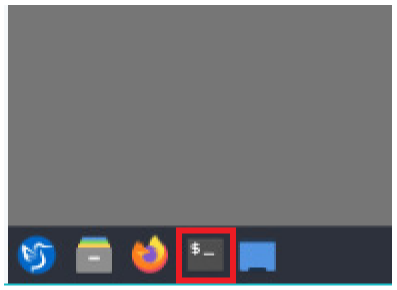
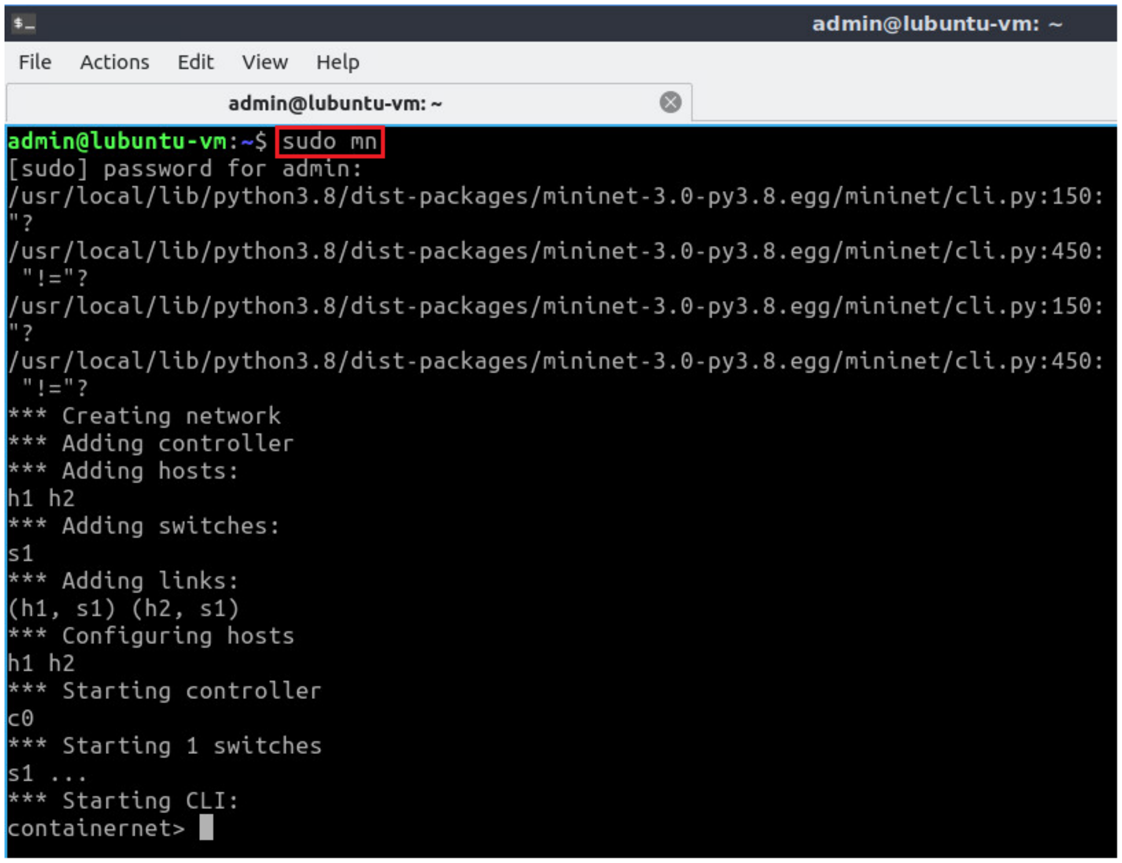


Figure 3. Linux terminal icon.

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

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

```
sudo mn
```



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

Figure 4. Starting Mininet using the CLI.

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

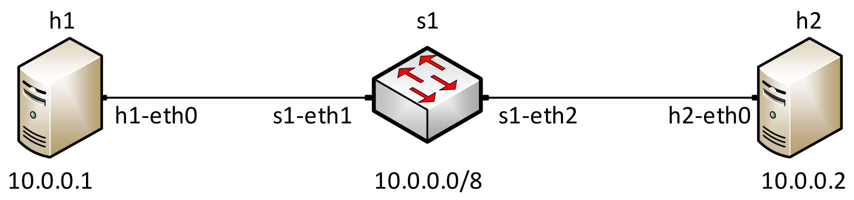


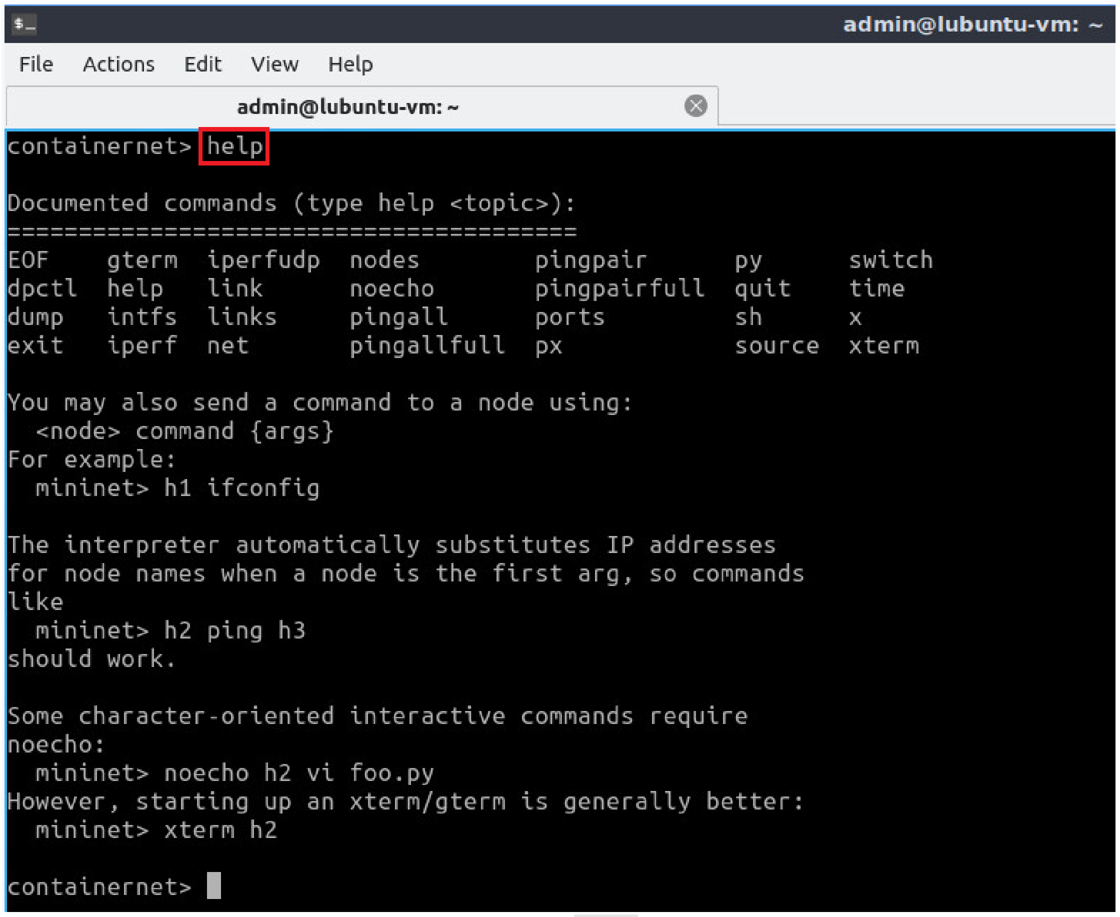
Figure 5. Mininet's default minimal topology.

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

```
containernet>
```

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

```
help
```

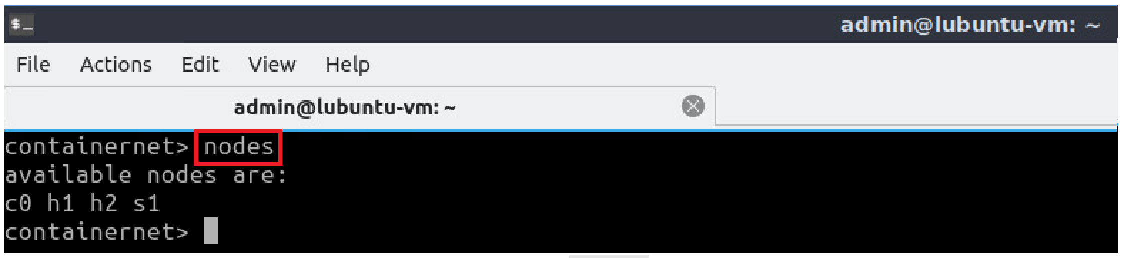


```
containernet> help

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

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

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

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

containernet> 
```

Figure 6. Mininet's `help` command.

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

```
nodes
```



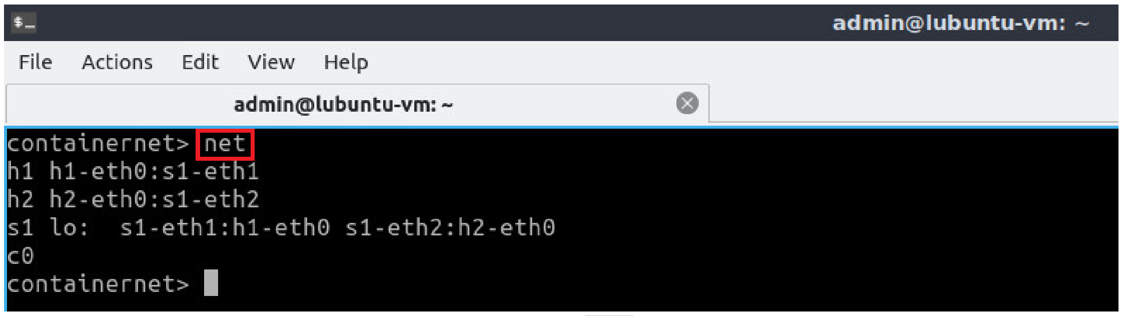
```
containernet> nodes
available nodes are:
c0 h1 h2 s1
containernet> 
```

Figure 7. Mininet's `nodes` command.

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

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

```
net
```



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

Figure 8. Mininet's `net` command.

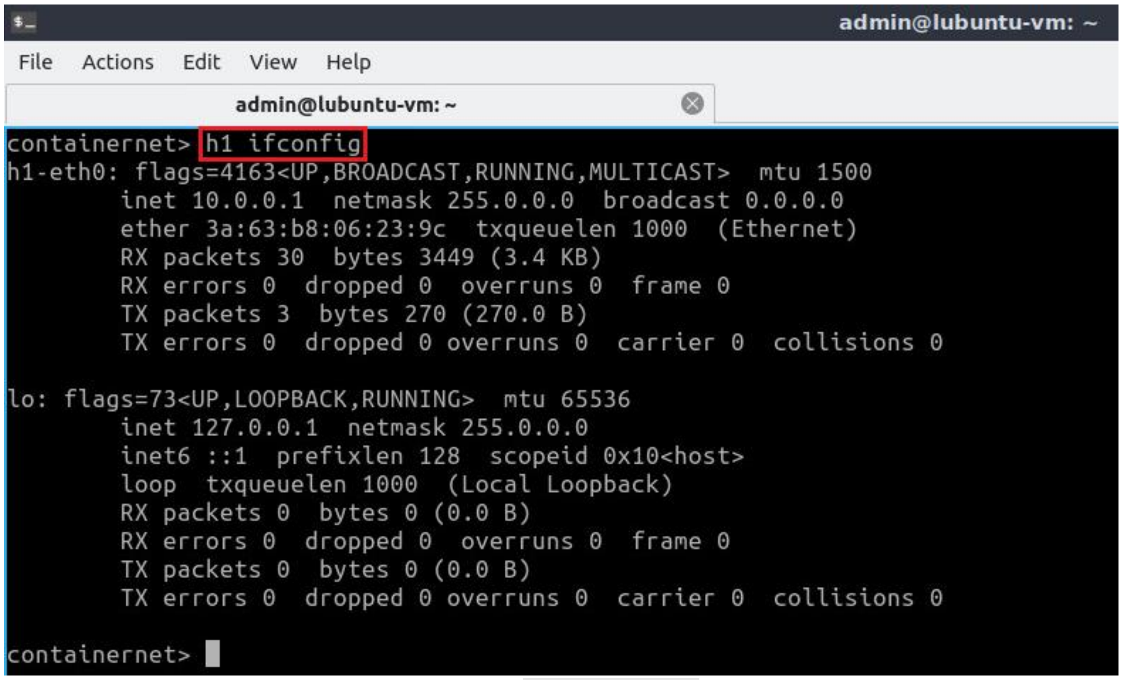
The output of the `net` command shows that:

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

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

Step 6. To proceed, issue the command:

```
h1 ifconfig
```



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

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

containernet>
```

Figure 9. Output of `h1 ifconfig` command.

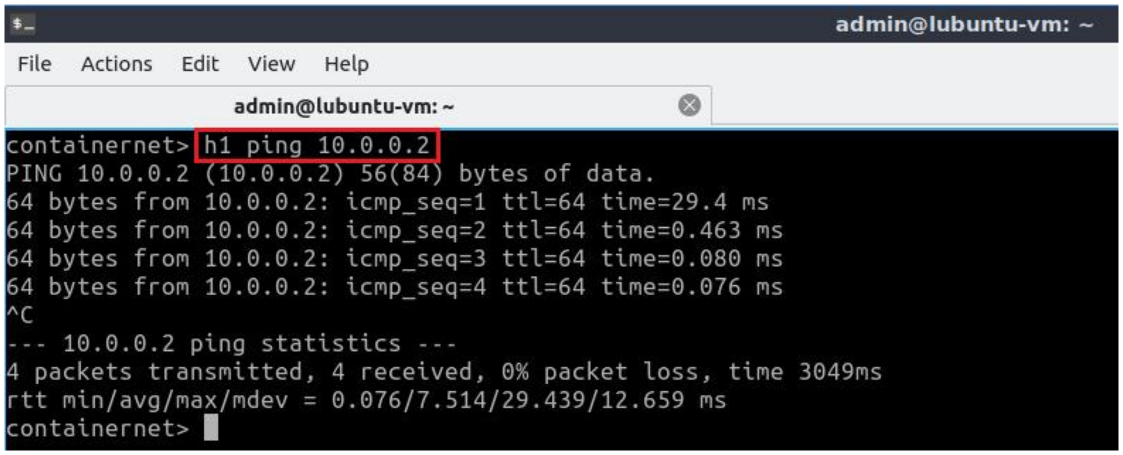
This command `h1 ifconfig` executes the `ifconfig` Linux command on host h1. The command shows host h1's interfaces. The display indicates that host h1 has an interface *h1-eth0* configured with IP address 10.0.0.1, and another interface lo configured with IP address 127.0.0.1 (loopback interface).

2.2 Test connectivity

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

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

```
h1 ping 10.0.0.2
```

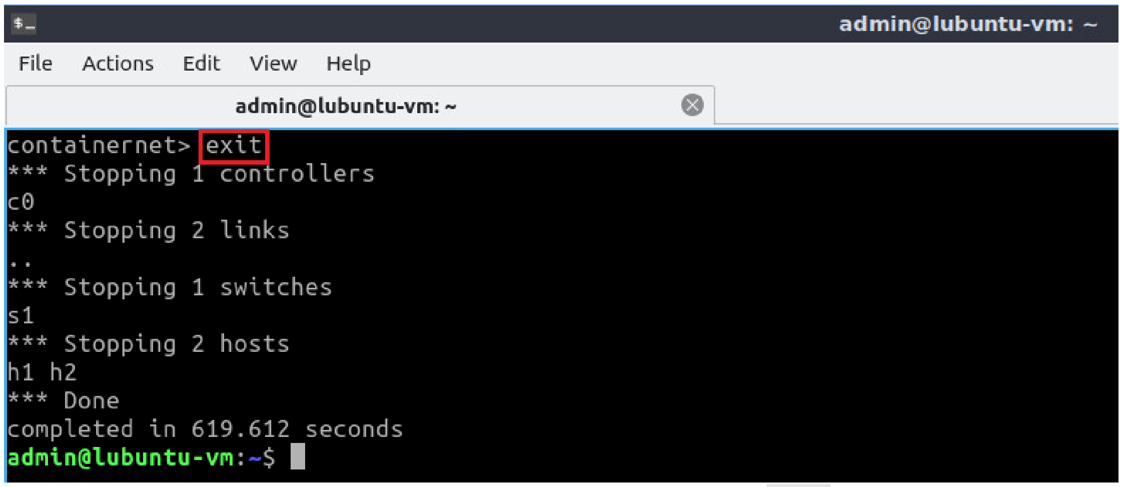


```
containernet> h1 ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=29.4 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.463 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.076 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.076/7.514/29.439/12.659 ms
containernet> 
```

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

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

```
exit
```



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

Figure 11. Stopping the emulation using `exit`.

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

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

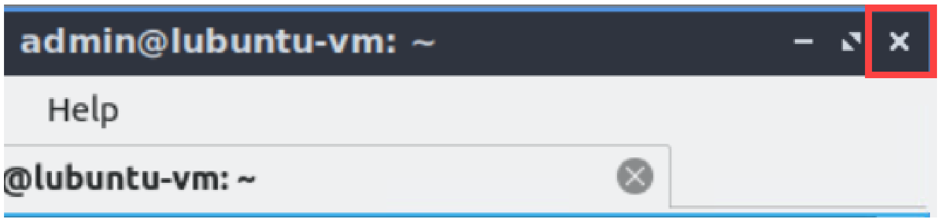


Figure 12. Closing the Linux CLI.

3 Build and emulate a network in Mininet using the GUI

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

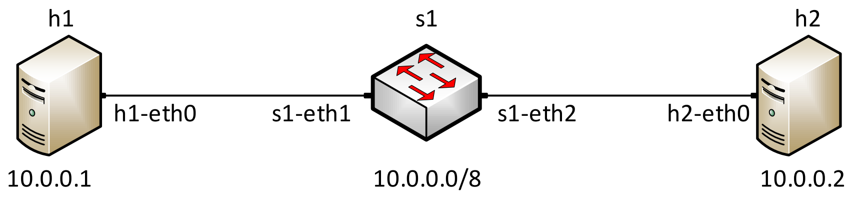


Figure 13. Lab topology.

3.1 Build the network topology

Step 1. A shortcut to MiniEdit is located on the machine's Desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`. MiniEdit will start, as illustrated below.

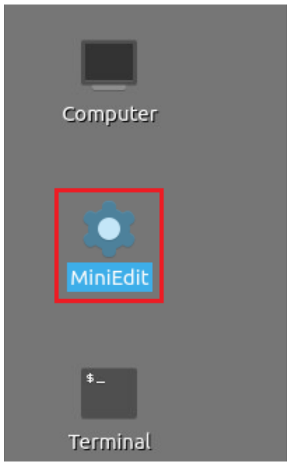


Figure 14. MiniEdit Desktop shortcut.

MiniEdit will start, as illustrated below.

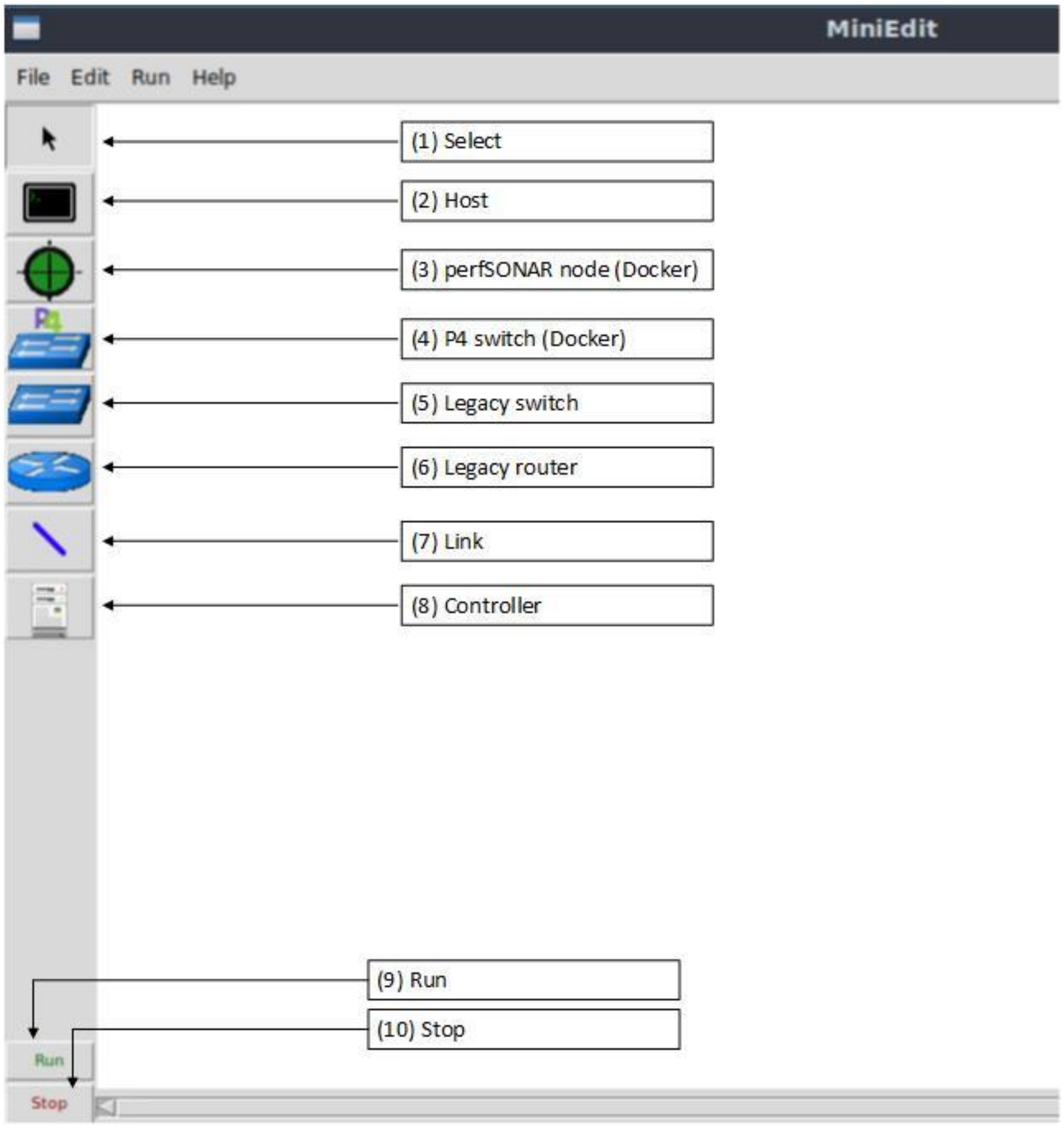


Figure 15. MiniEdit Graphical User Interface (GUI).

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. perfSONAR node (Docker): allows the addition a perfSONAR node. After clicking this button, click anywhere in the blank canvas to insert a perfSONAR node.
4. P4 switch (Docker): allows the addition of P4 switch. After clicking this button, click anywhere in the blank canvas to insert the P4 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.

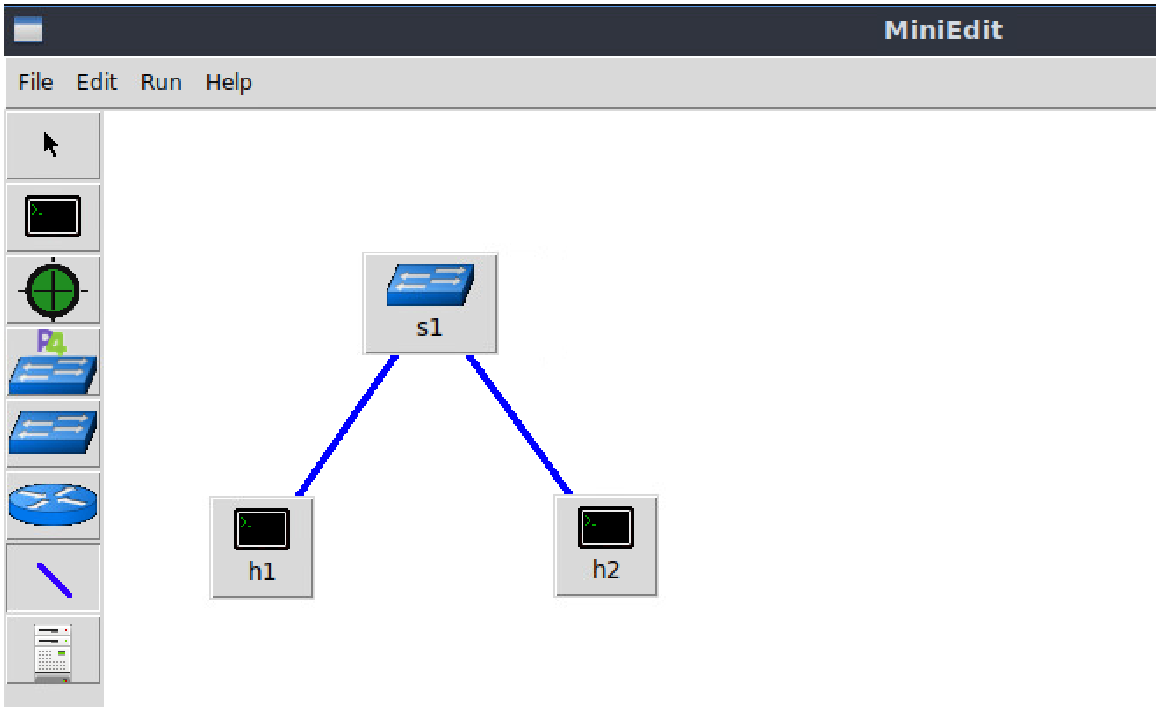
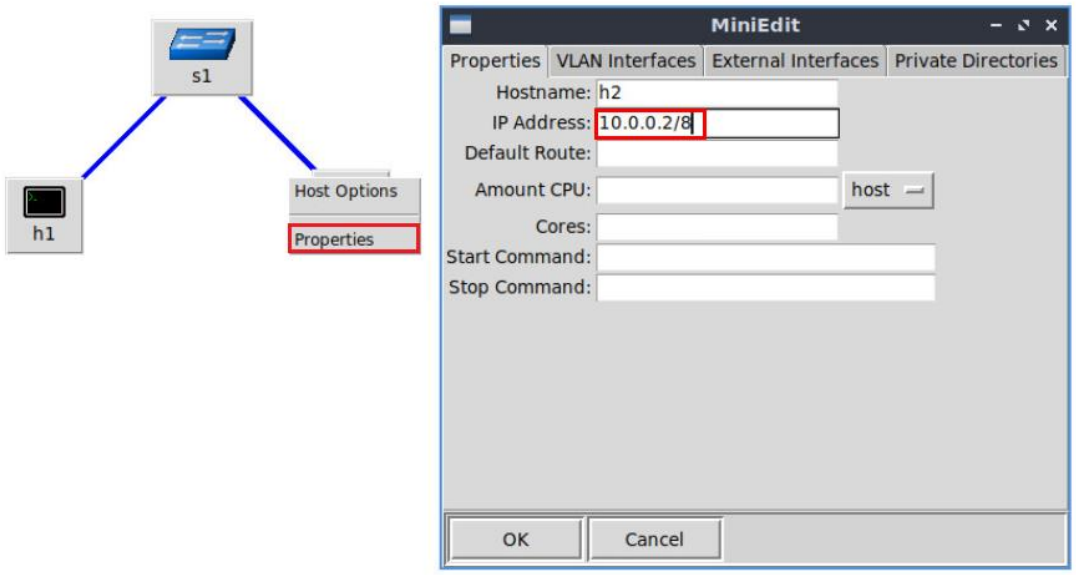


Figure 16. MiniEdit's topology.

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.



Hostname: h2
IP Address: 10.0.0.2/8
Default Route:

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.

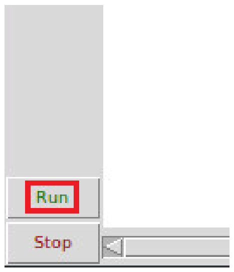


Figure 18. Starting the emulation.

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.

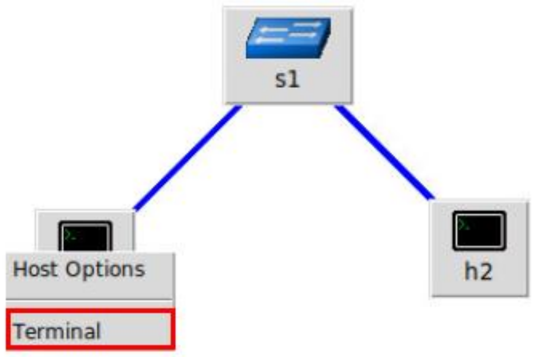
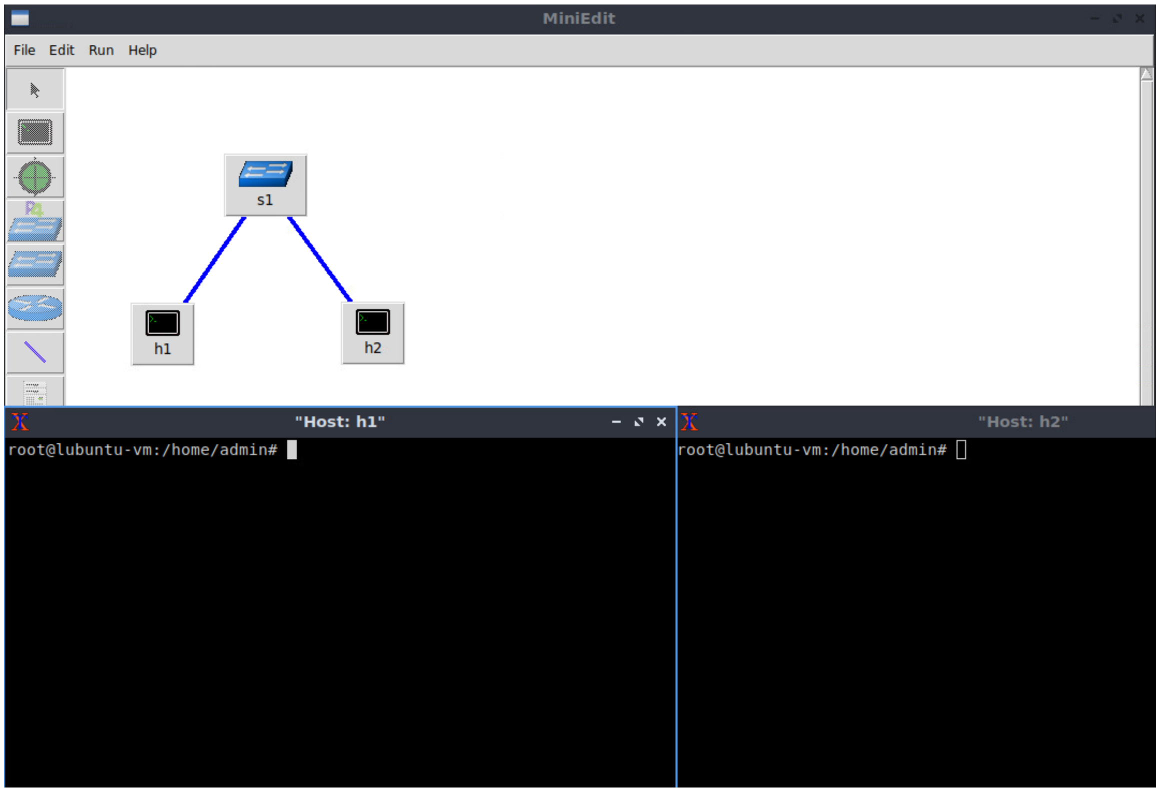


Figure 19. Opening a terminal on host h1.

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



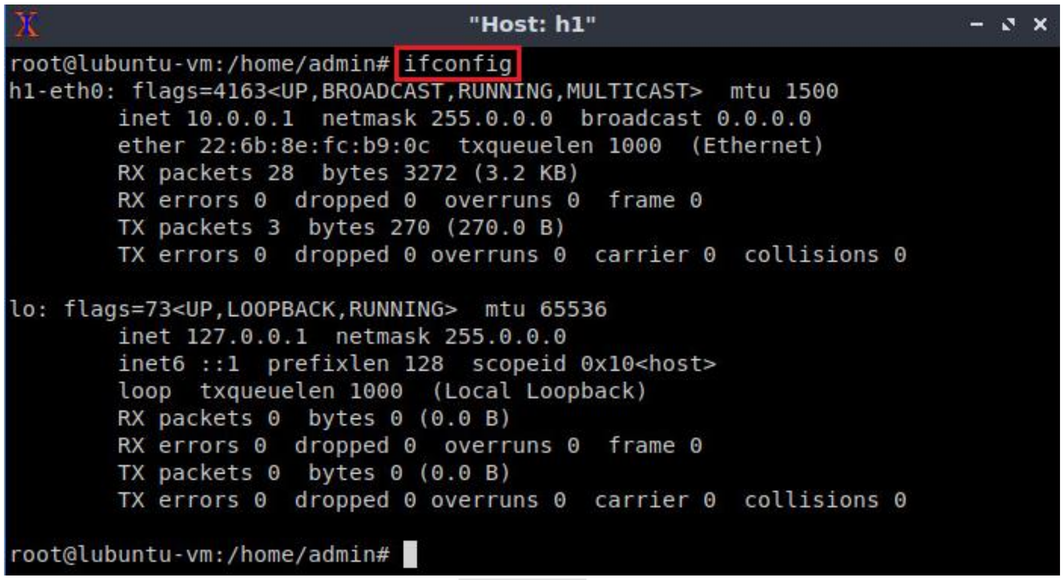
```
root@lubuntu-vm:/home/admin#
```

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

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
```



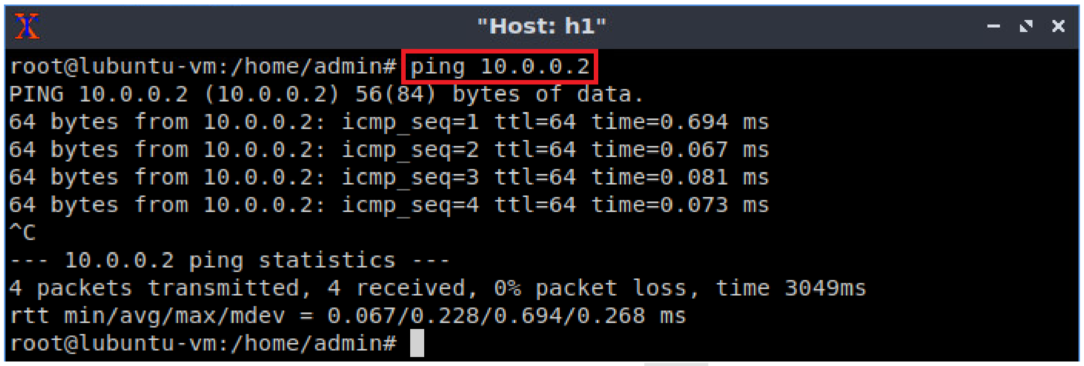
```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 22:6b:8e:fc:b9:0c  txqueuelen 1000  (Ethernet)
        RX packets 28  bytes 3272 (3.2 KB)
        RX errors 0  dropped 0  overruns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

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

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

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

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

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

```
ping 10.0.0.2
```



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

Figure 22. Connectivity test using `ping` command.

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

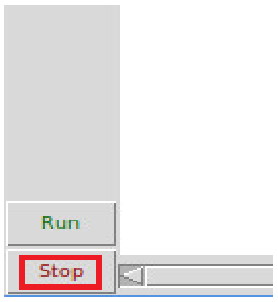
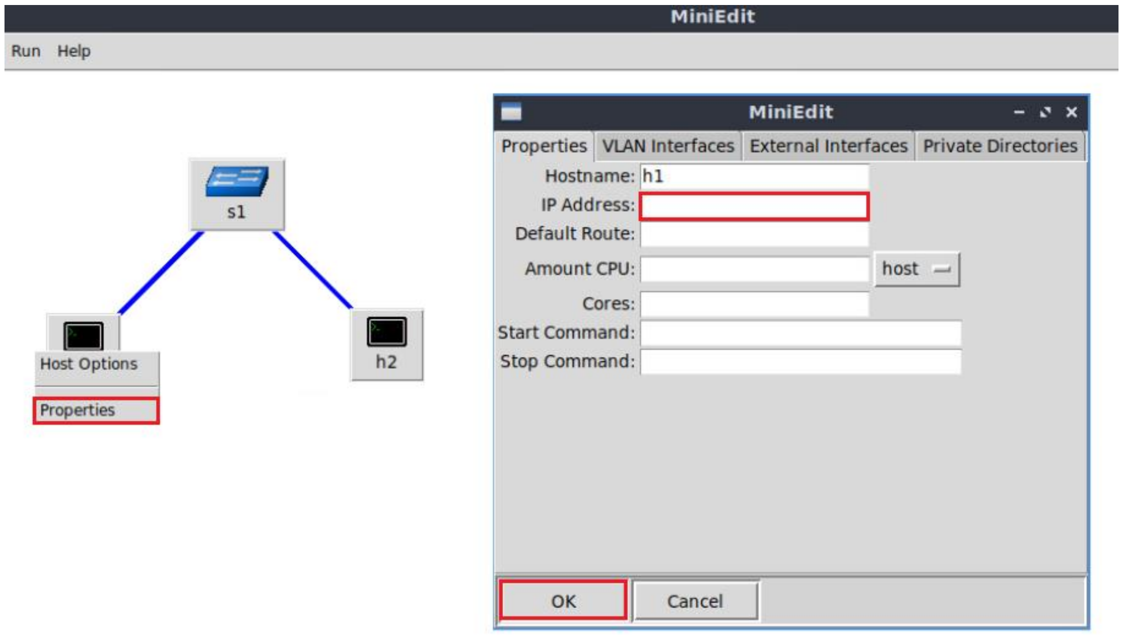


Figure 23. Stopping the emulation.

3.3 Automatic assignment of IP addresses

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

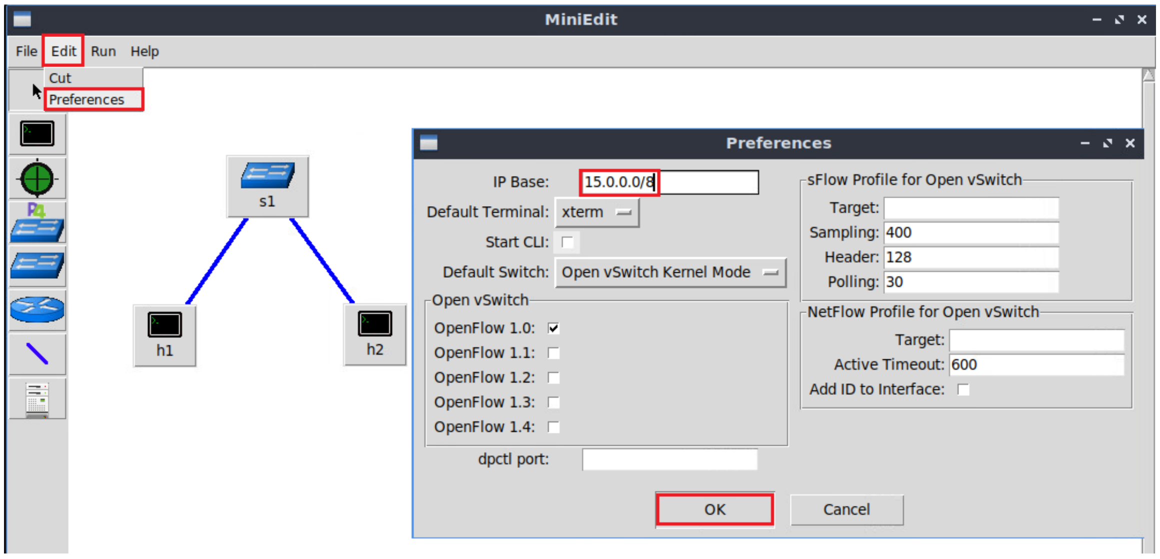
Step 1. Remove the manually assigned IP address from host h1. Right-click on host h1 and select *Properties*. Delete the IP address, leaving it unassigned, and press the *OK* button as shown below. Repeat the procedure on host h2.



Hostname: h1
IP Address:
Default Route:
Amount CPU: host
Cores:
Start Command:
Stop Command:

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.



IP Base: 15.0.0.0/8
Default Terminal: xterm
Start CLI: ☐
Default Switch: Open vSwitch Kernel Mode

Open vSwitch
OpenFlow 1.0: ☑
OpenFlow 1.1: ☐
OpenFlow 1.2: ☐
OpenFlow 1.3: ☐
OpenFlow 1.4: ☐
dpctl port:

sFlow Profile for Open vSwitch
Target:
Sampling: 400
Header: 128
Polling: 30

NetFlow Profile for Open vSwitch
Target:
Active Timeout: 600
Add ID to Interface: ☐

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.

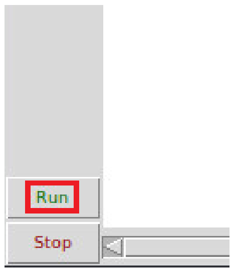


Figure 26. Starting the emulation.

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

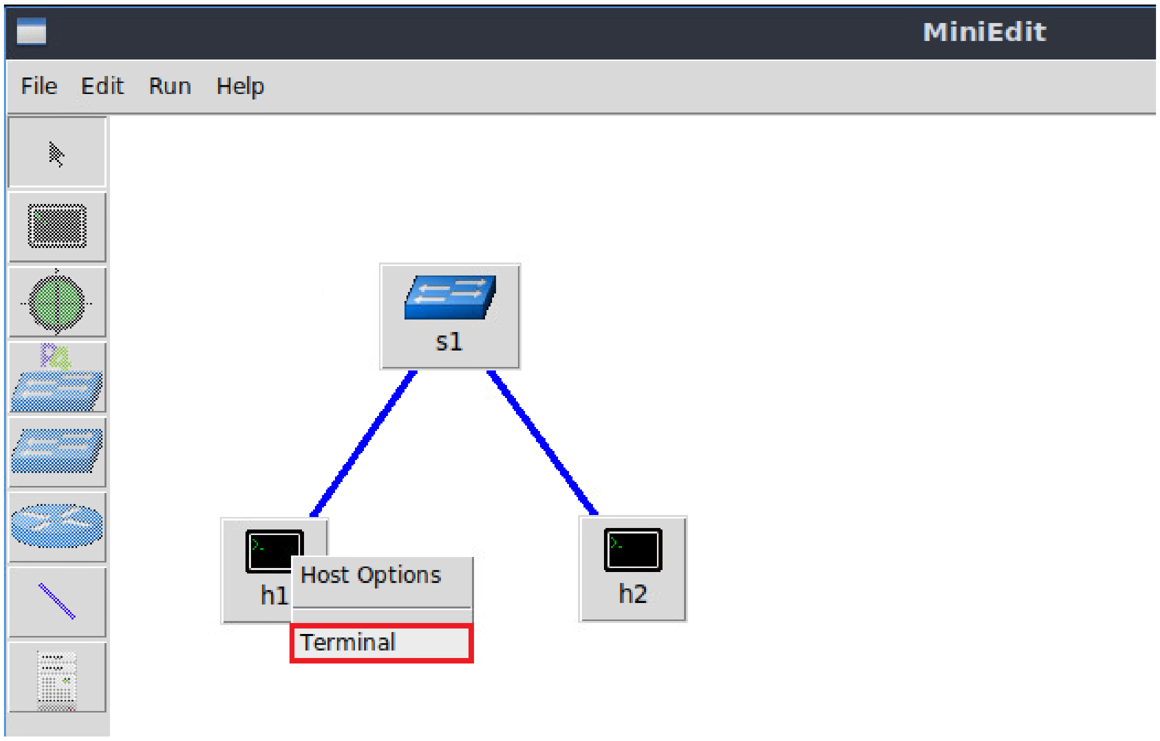
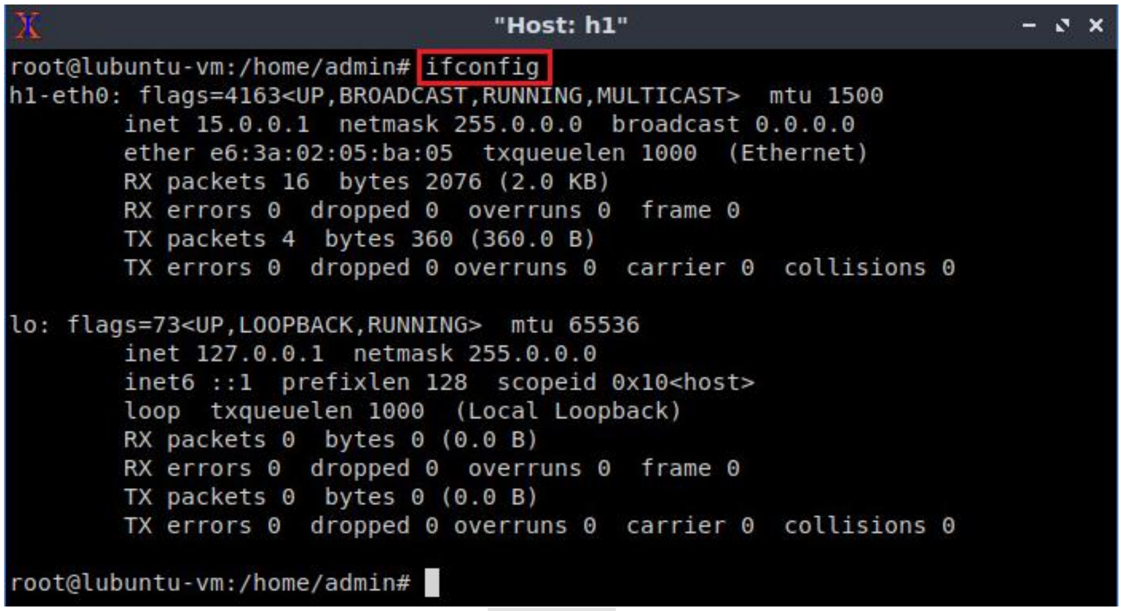


Figure 27. Opening a terminal on host h1.

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
```



```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 15.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether e6:3a:02:05:ba:05  txqueuelen 1000  (Ethernet)
        RX packets 16  bytes 2076 (2.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 360 (360.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

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

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

Figure 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.

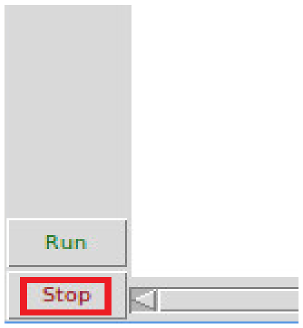
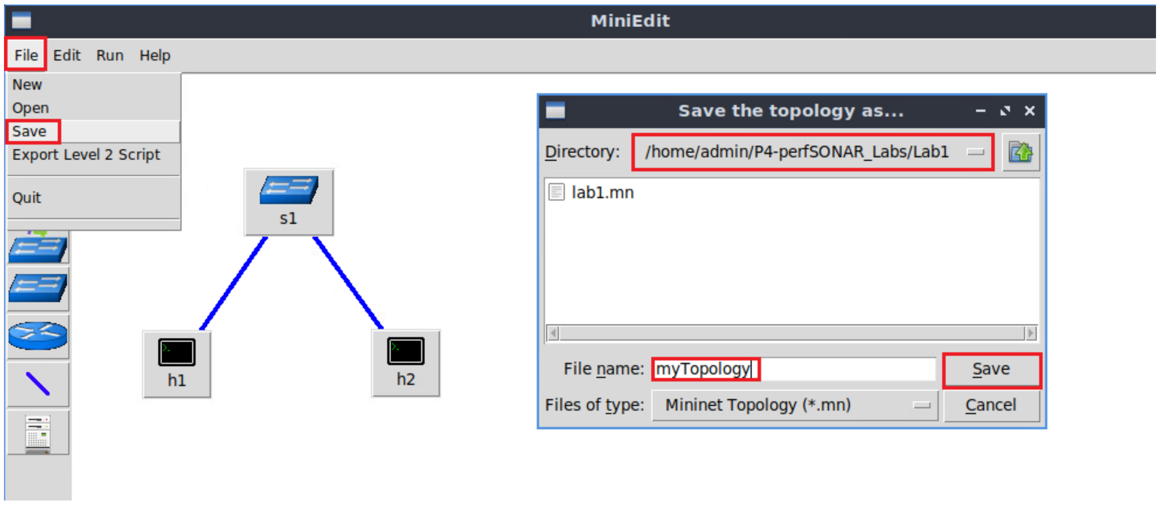


Figure 29. Stopping the emulation.

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*.



Directory: /home/admin/P4-perfSONAR_Labs/Lab1

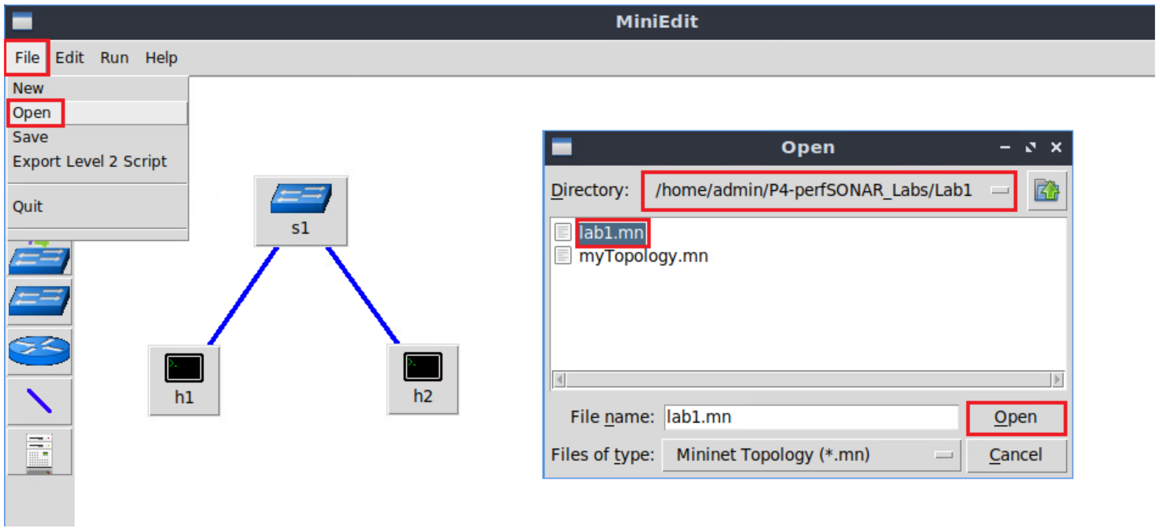
lab1.mn

File name: myTopology

Files of type: Mininet Topology (*.mn)

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.



Directory: /home/admin/P4-perfSONAR_Labs/Lab1

lab1.mn
myTopology.mn

File name: lab1.mn

Files of type: Mininet Topology (*.mn)

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-PERFSONAR LAB SERIES

Lab 2: P4 Program Building Blocks

Document Version: 01-25-2022

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

Contents

Overview 3
Objectives..... 3
Lab settings 3
Lab roadmap 3
1 The PISA architecture..... 3
1.1 The PISA architecture 4
1.2 Programmable parser 4
1.3 Programmable match-action pipeline 5
1.4 Programmable deparser 5
1.5 The V1Model 5
1.6 P4 program mapping to the V1Model 6
2 Lab topology..... 6
2.1 Starting host h1 and host h2 8
3 Navigating through the components of a basic P4 program..... 9
3.1 Loading the programming environment..... 9
3.2 Describing the components of the P4 program..... 10
3.3 Programming the pipeline sequence 15
4 Loading the P4 program..... 16
4.1 Compiling and loading the P4 program to switch s1 16
4.2 Verifying the configuration 17
5 Configuring switch s1..... 18
5.1 Mapping the P4 program's ports 18
5.2 Loading the rules to the switch..... 20
6 Testing and verifying the P4 program..... 21
References 23

Overview

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

Objectives

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

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

Lab settings

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

Table 1. Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: The PISA architecture.
2. Section 2: Lab topology.
3. Section 3: Navigating through the components of a basic P4 program.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Testing and verifying the P4 program.

1 The PISA architecture

1.1 The PISA architecture

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

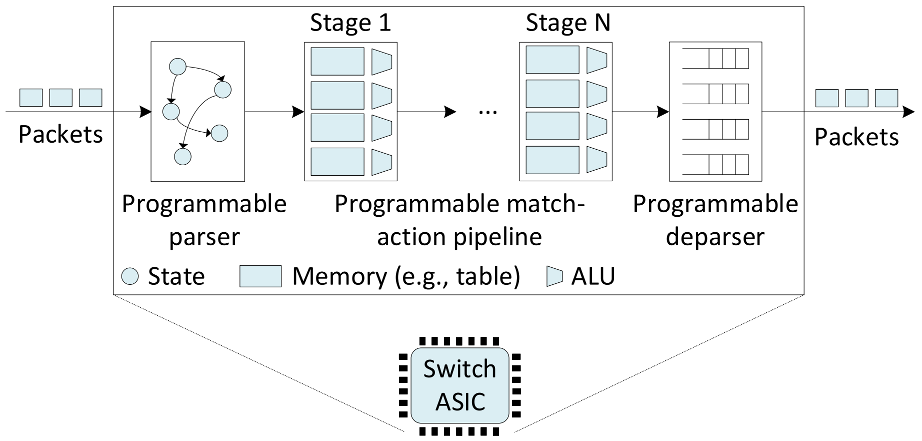


Figure 1. A PISA-based data plane.

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

1.2 Programmable parser

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

1.3 Programmable match-action pipeline

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

1.4 Programmable deparser

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

1.5 The V1Model

Figure 2 depicts the V1Model[2] architecture components. The V1Model architecture consists of a programmable parser, an ingress match-action pipeline, a traffic manager, an egress match-action pipeline, and a programmable deparser. The traffic manager schedules packets between input ports and output ports and performs packet replication (e.g., replication of a packet for multicasting). The V1Model architecture is implemented on top BMv2's simple_switch target[3].

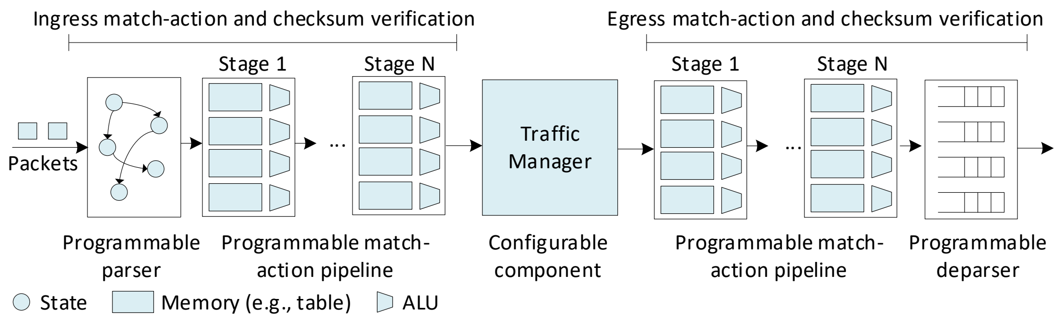

Ingress match-action and checksum verification
Egress match-action and checksum verification
Stage 1
Stage N
Stage 1
Stage N
Packets
...
Traffic Manager
...
Programmable parser
Programmable match-action pipeline
Configurable component
Programmable match-action pipeline
Programmable deparser
State
Memory (e.g., table)
ALU

Figure 2. The V1Model architecture.

1.6 P4 program mapping to the V1Model

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

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

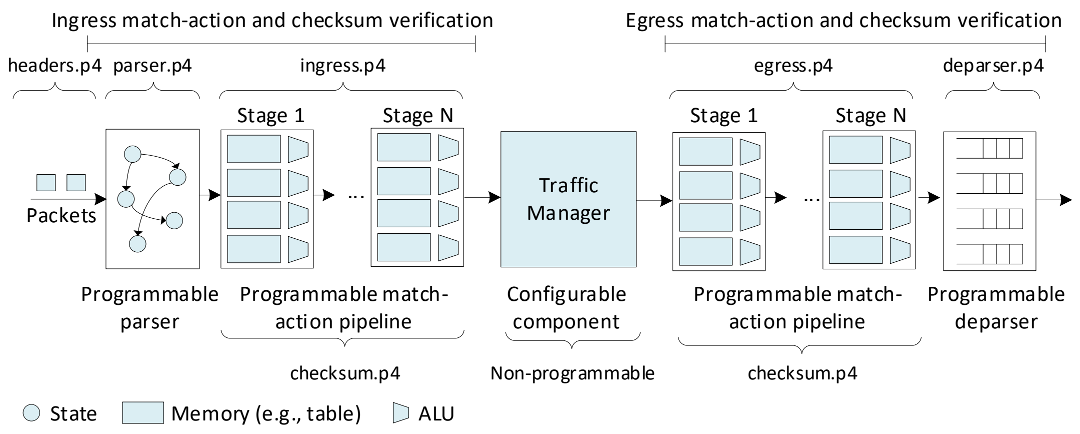


Figure 3. Mapping of P4 files to the V1Model's components.

2 Lab topology

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

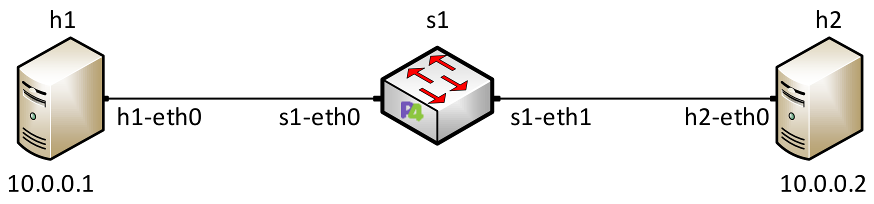


Figure 4. Lab topology.

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

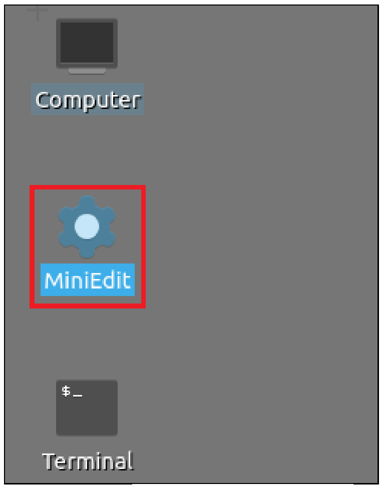
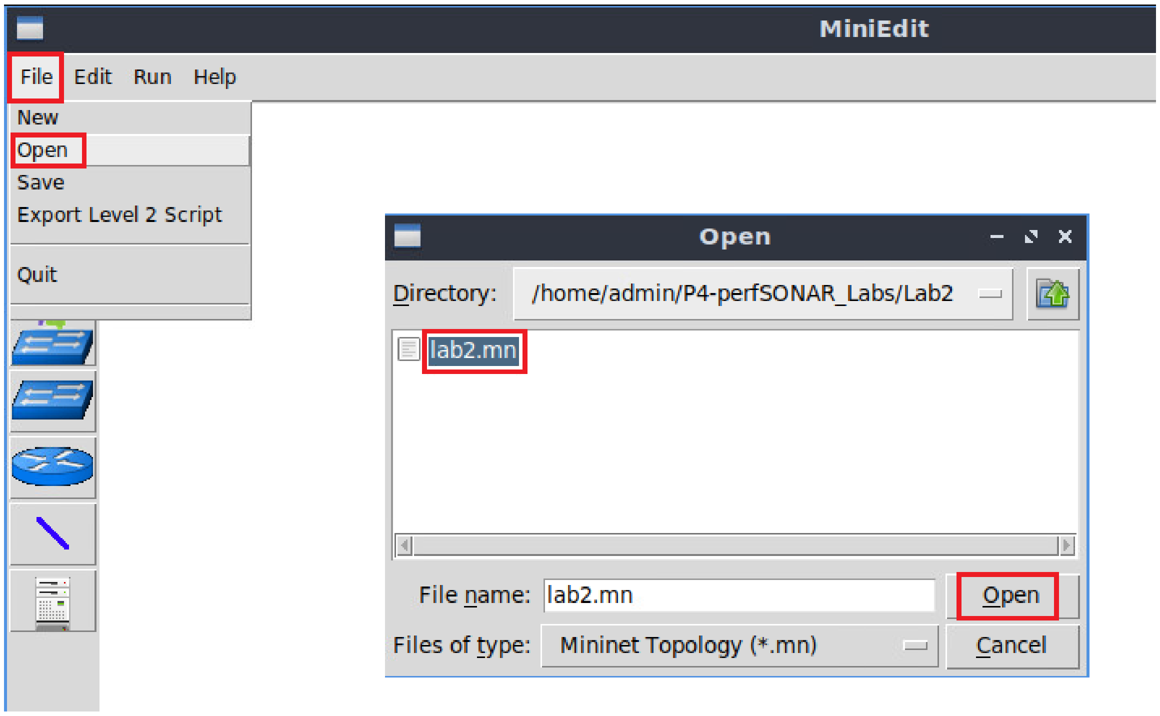


Figure 5. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the lab2 folder and search for the topology file called *lab2.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4-perfSONAR_Labs/Lab2

lab2.mn

File name: lab2.mn

Files of type: Mininet Topology (*.mn)

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.

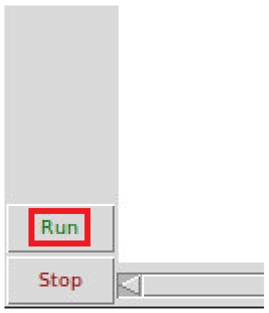


Figure 7. Running the emulation.

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.

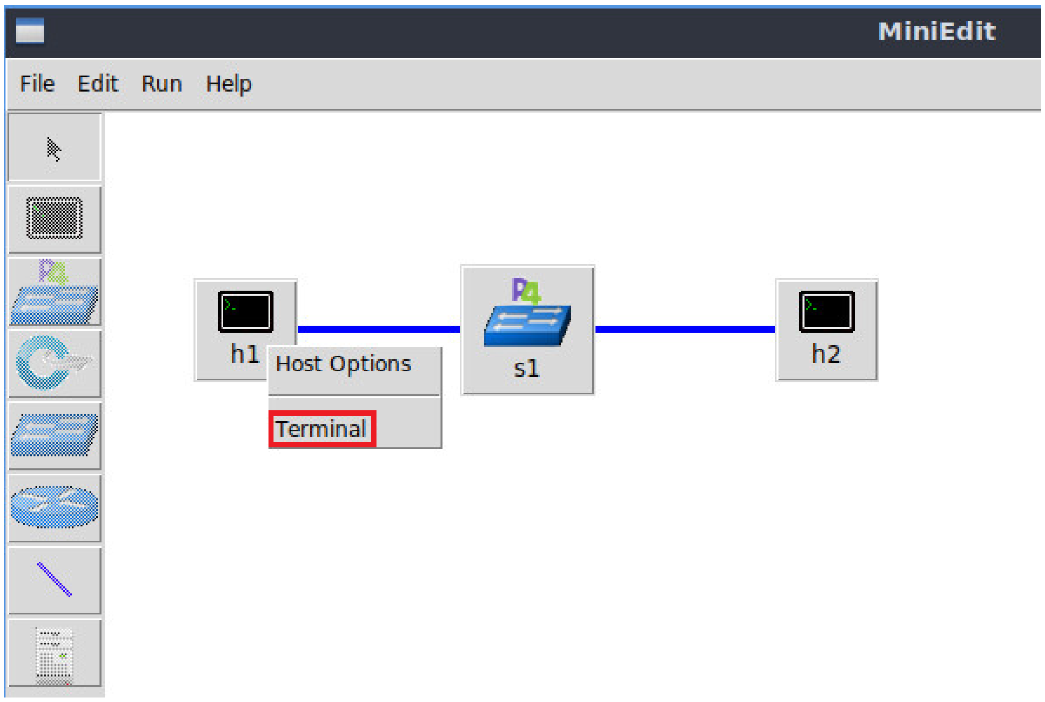
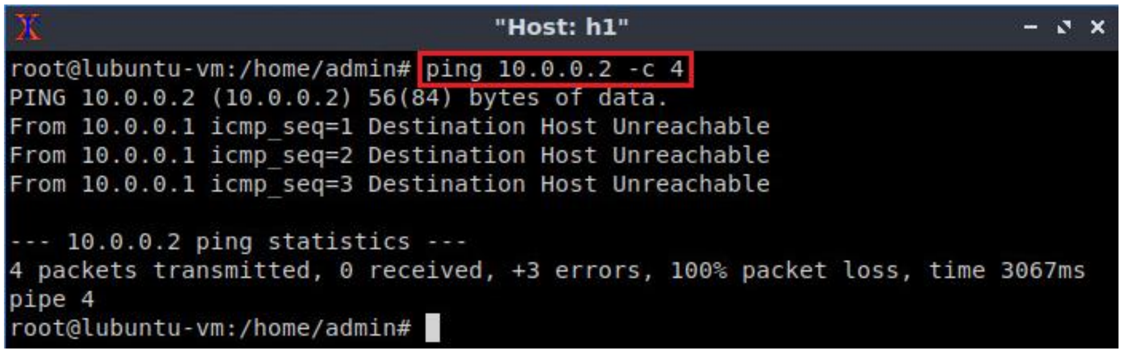


Figure 8. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```



```
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 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.

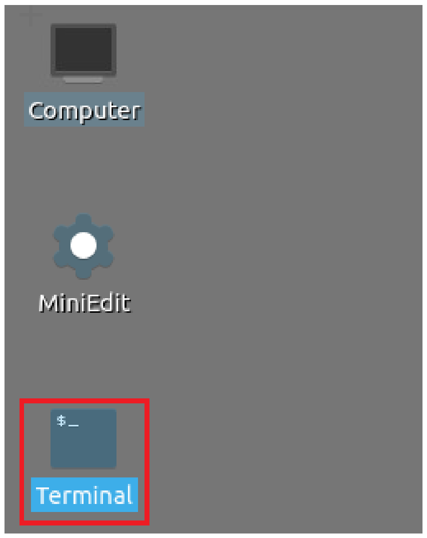
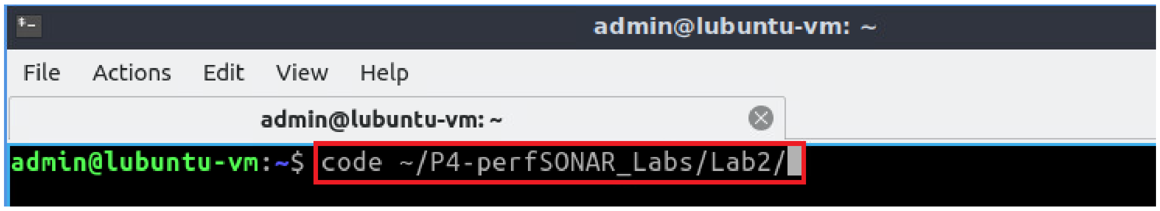


Figure 10. Shortcut to open a Linux terminal.

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-perfSONAR_Labs/lab2/
```

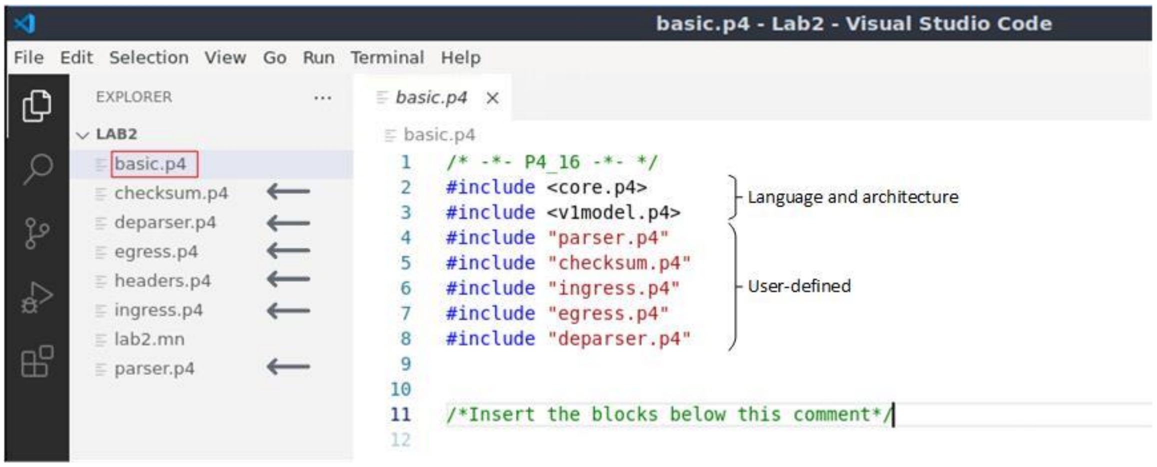


```
admin@lubuntu-vm:~$ code ~/P4-perfSONAR_Labs/Lab2/
```

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.


basic.p4 - Lab2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB2
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab2.mn
parser.p4
Language and architecture
User-defined

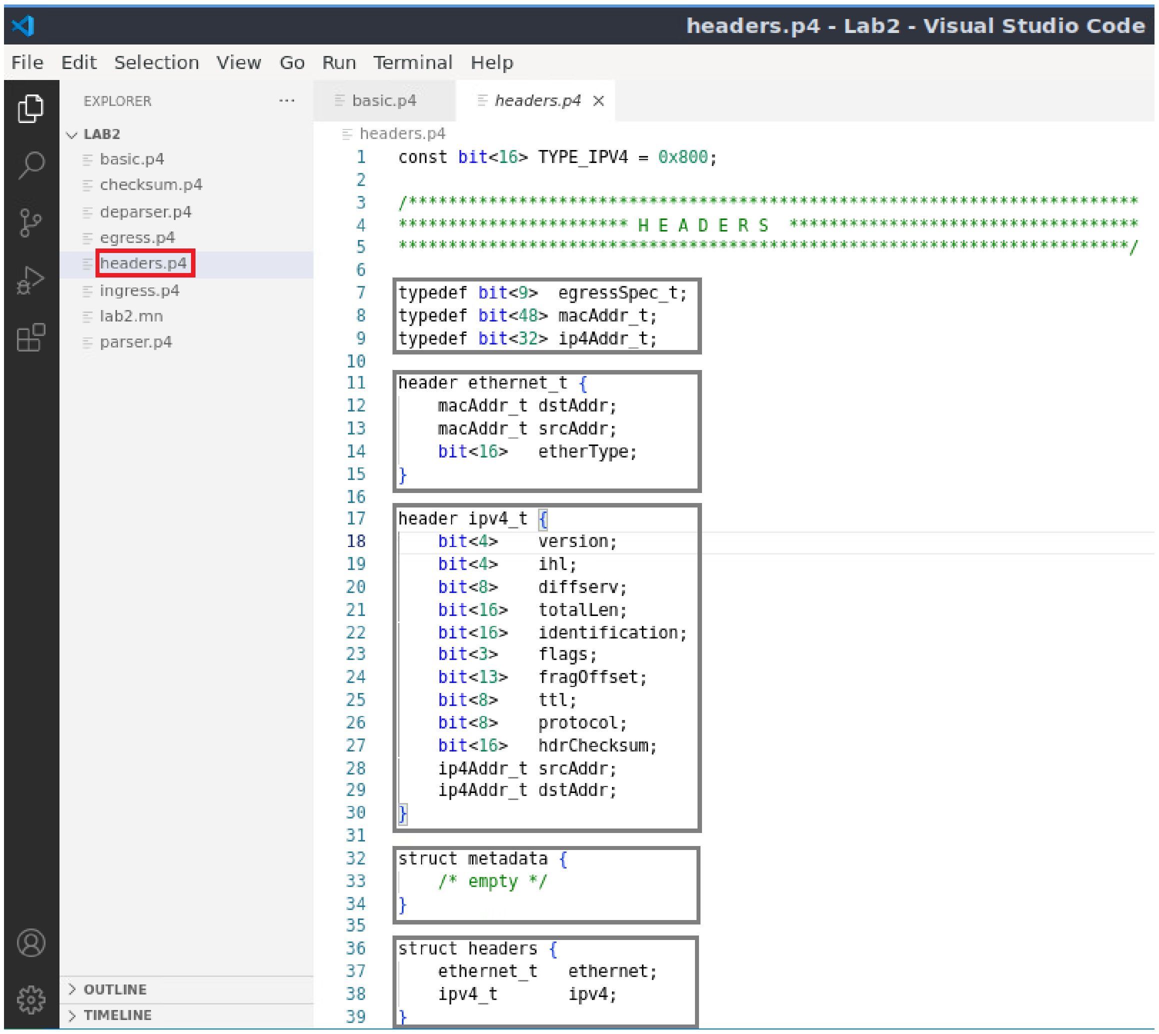
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/

```

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

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

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

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

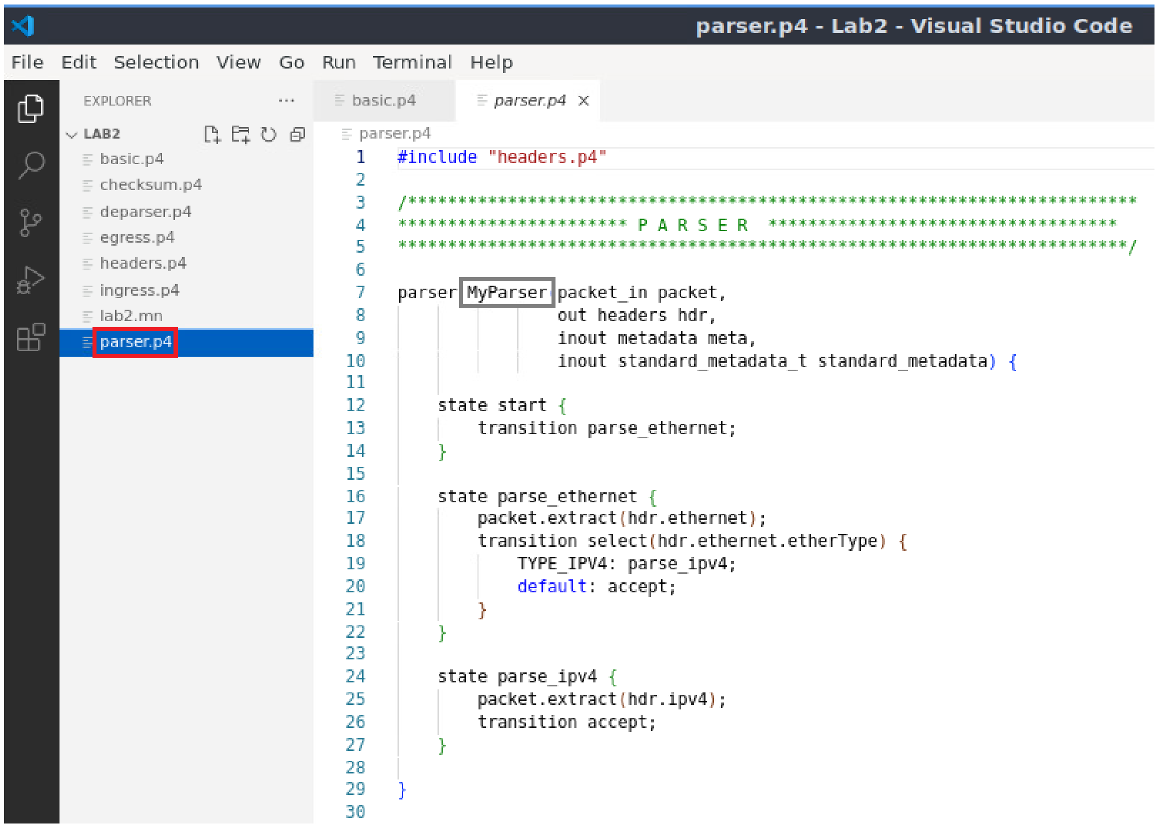


```
const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

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

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

struct metadata {
    /* empty */
}

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

Figure 13. The defined headers.

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

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

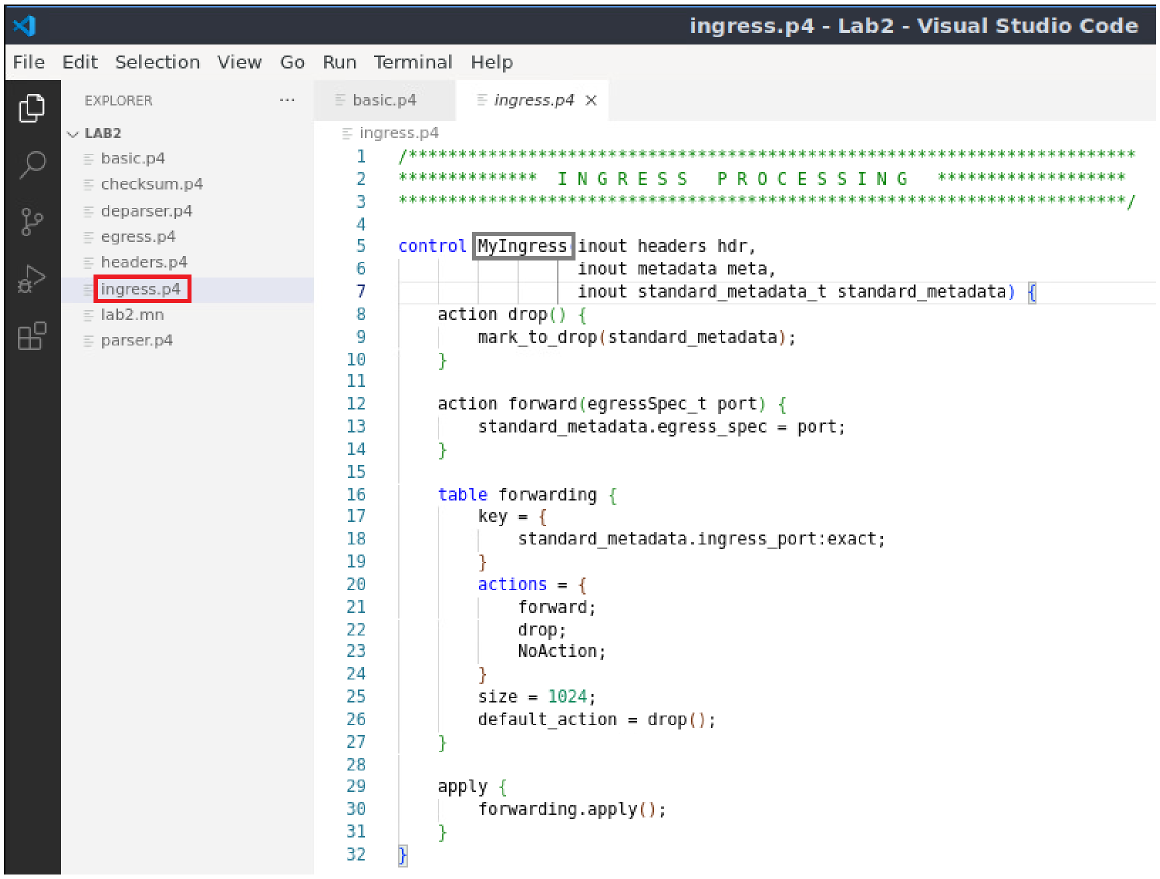


```
#include "headers.p4"

/*************************************************************************
*********************** P A R S E R  ***********************************
*************************************************************************/

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

    state start {
        transition parse_ethernet;
    }

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

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

}
```

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.

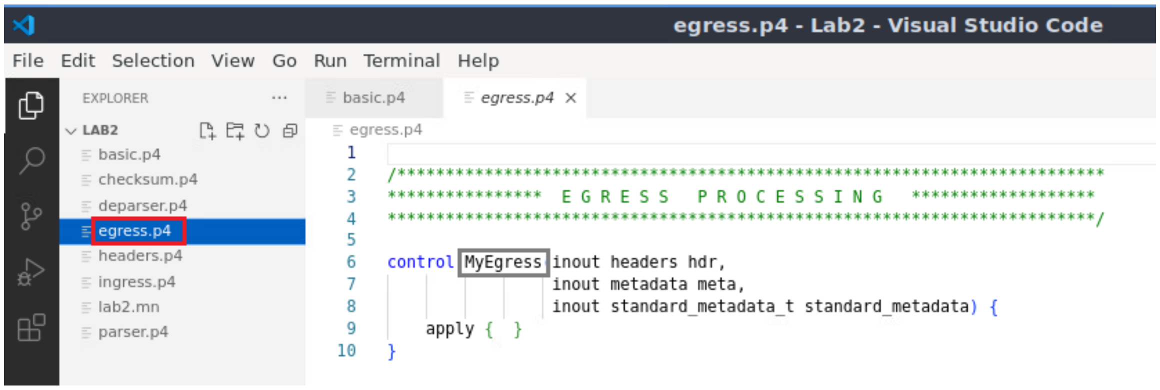


```
/*************************************************************************
*************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

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

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

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

Figure 15. The ingress component.

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

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

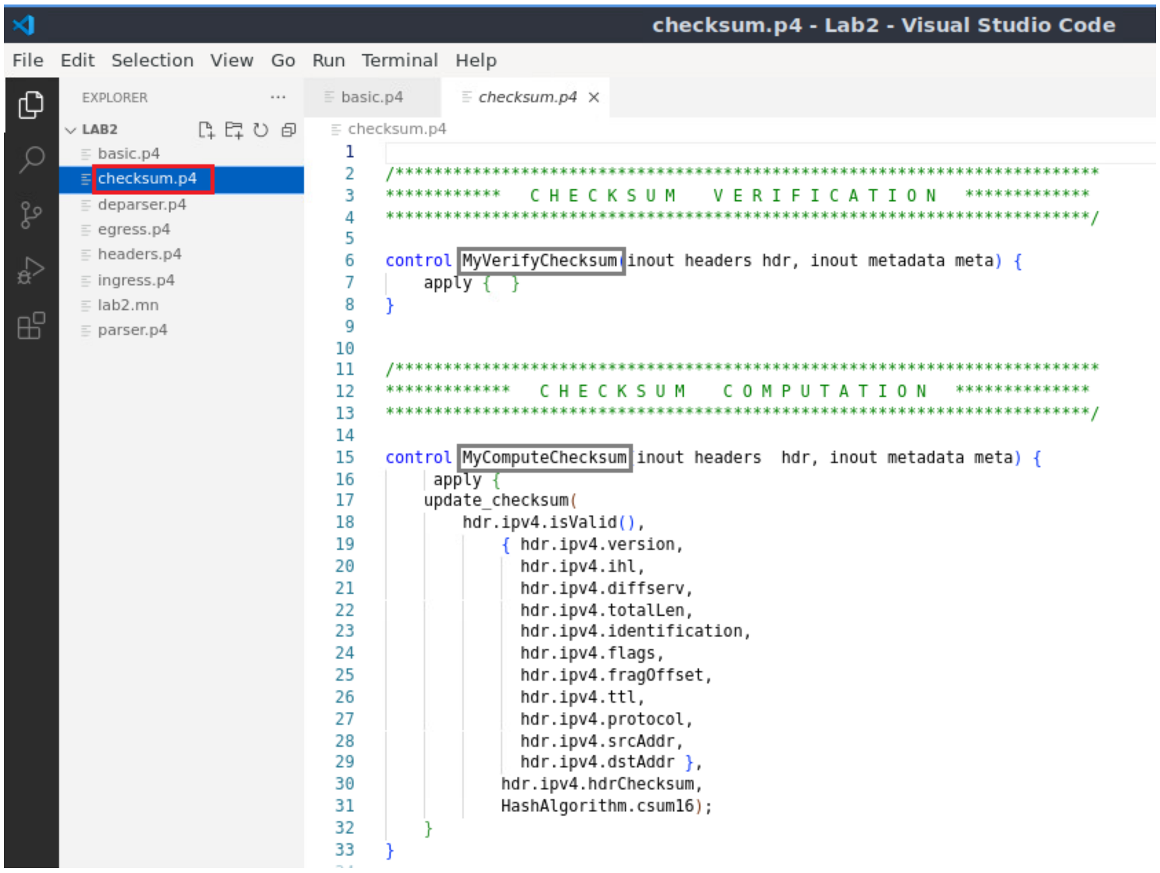


```

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

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

Figure 16. The egress component.

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

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

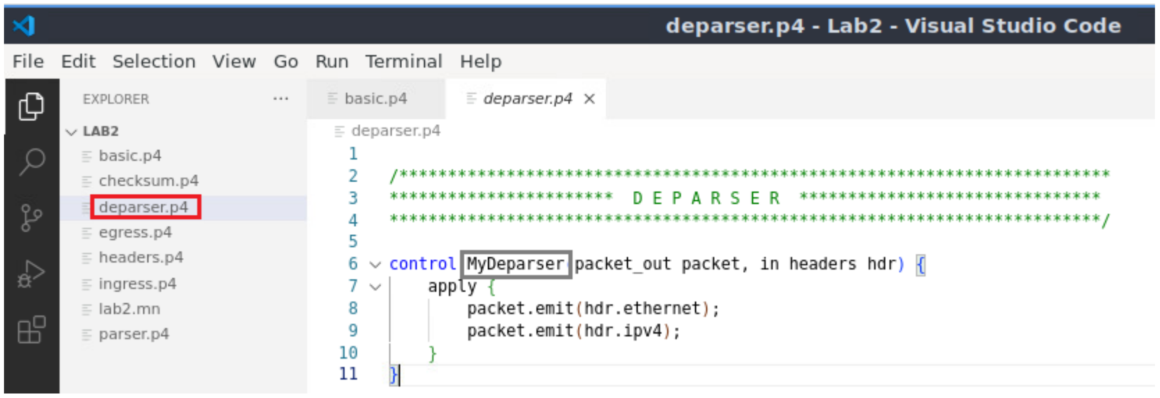


```
/*************************************************************************
************   C H E C K S U M    V E R I F I C A T I O N   *************
*************************************************************************/

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

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

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

Figure 17. The checksum component.

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

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



```
/*************************************************************************
***********************  D E P A R S E R  *******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

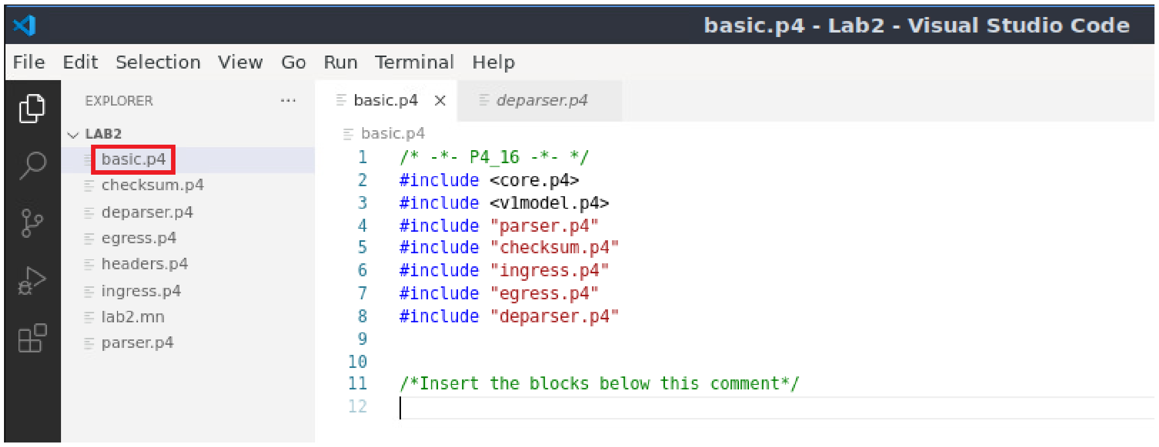
Figure 18. The deparser component.

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.

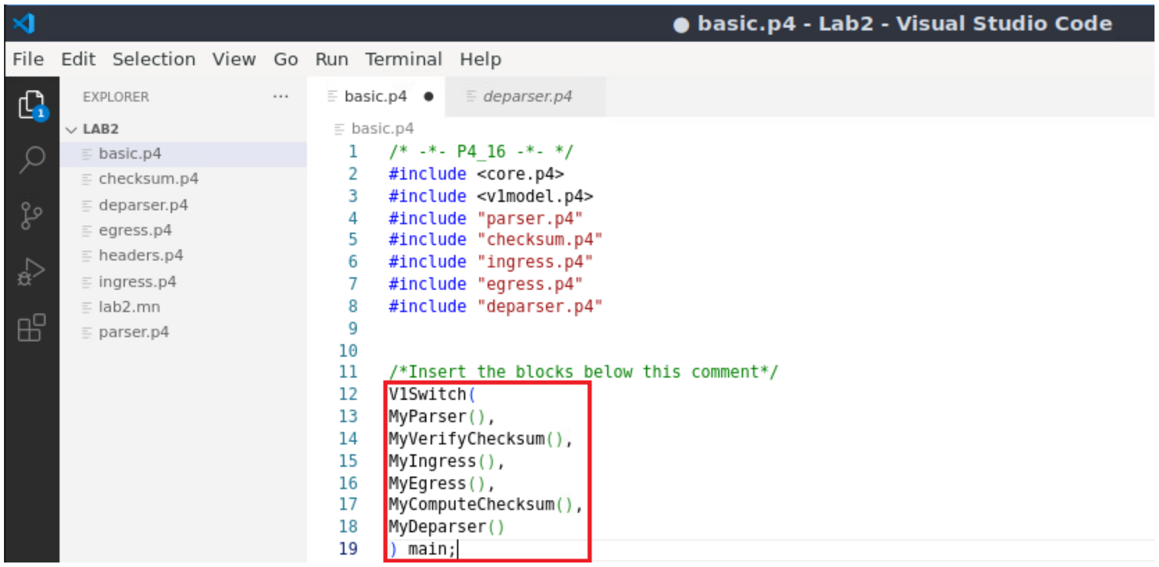


```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/

```

Figure 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;
```



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


/*Insert the blocks below this comment*/
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.

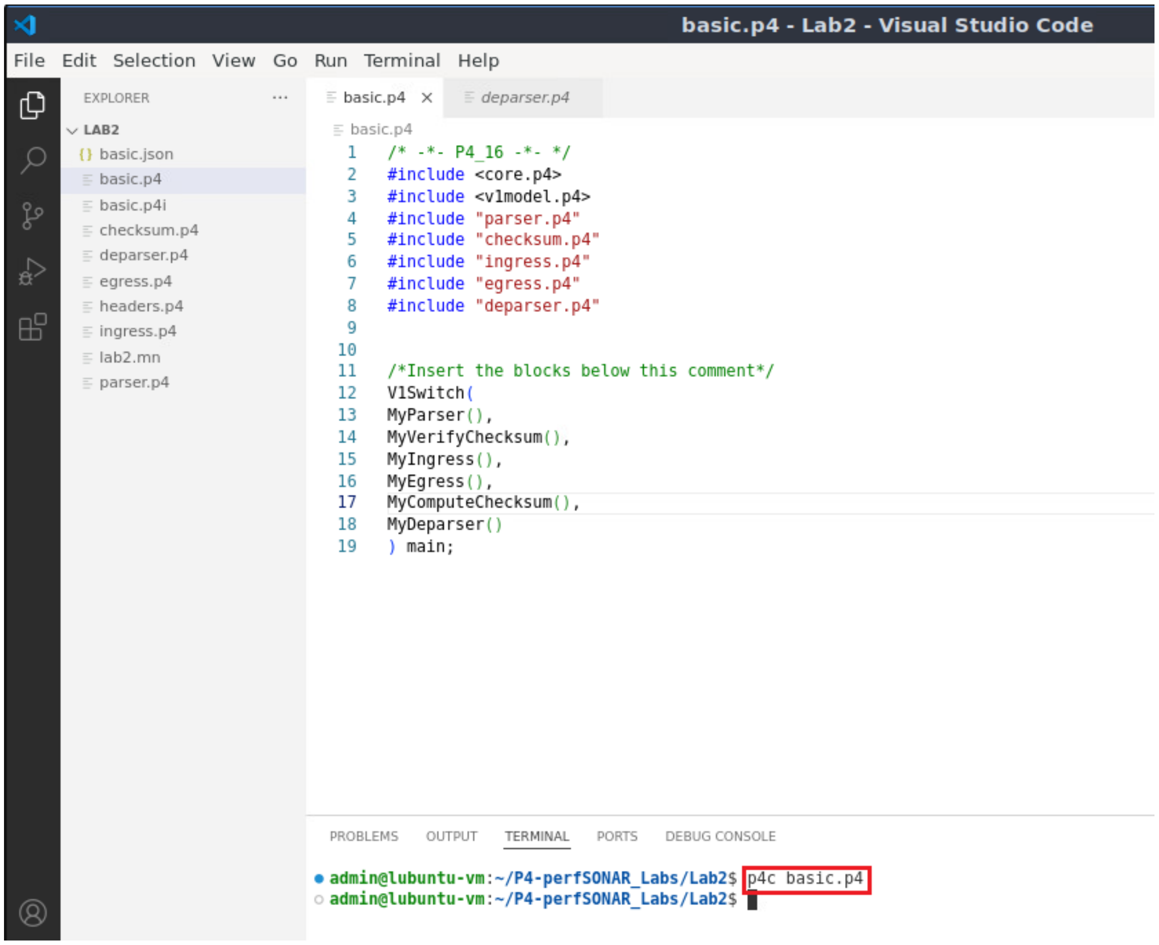
Step 3. Save the changes by pressing `Ctrl+s`.

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
```



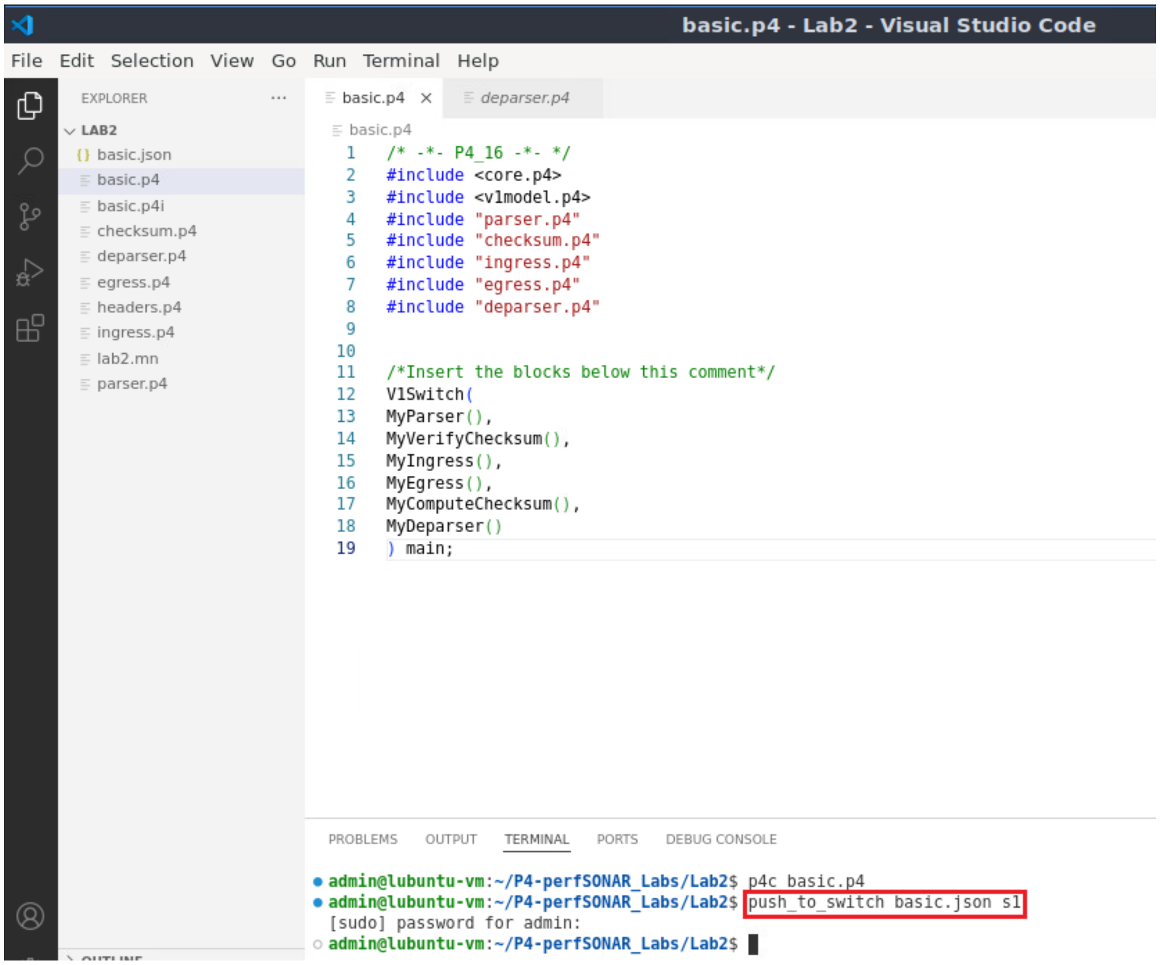
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab2$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab2$
```

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
```


basic.p4 - Lab2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB2
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab2.mn
parser.p4
basic.p4
deparser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

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


/*Insert the blocks below this comment*/
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab2$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab2$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab2$
```

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.

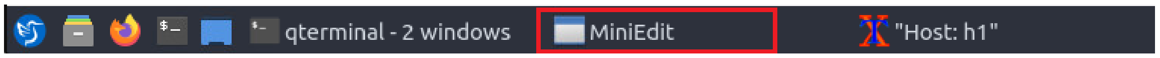


Figure 23. Maximizing the MiniEdit window.

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

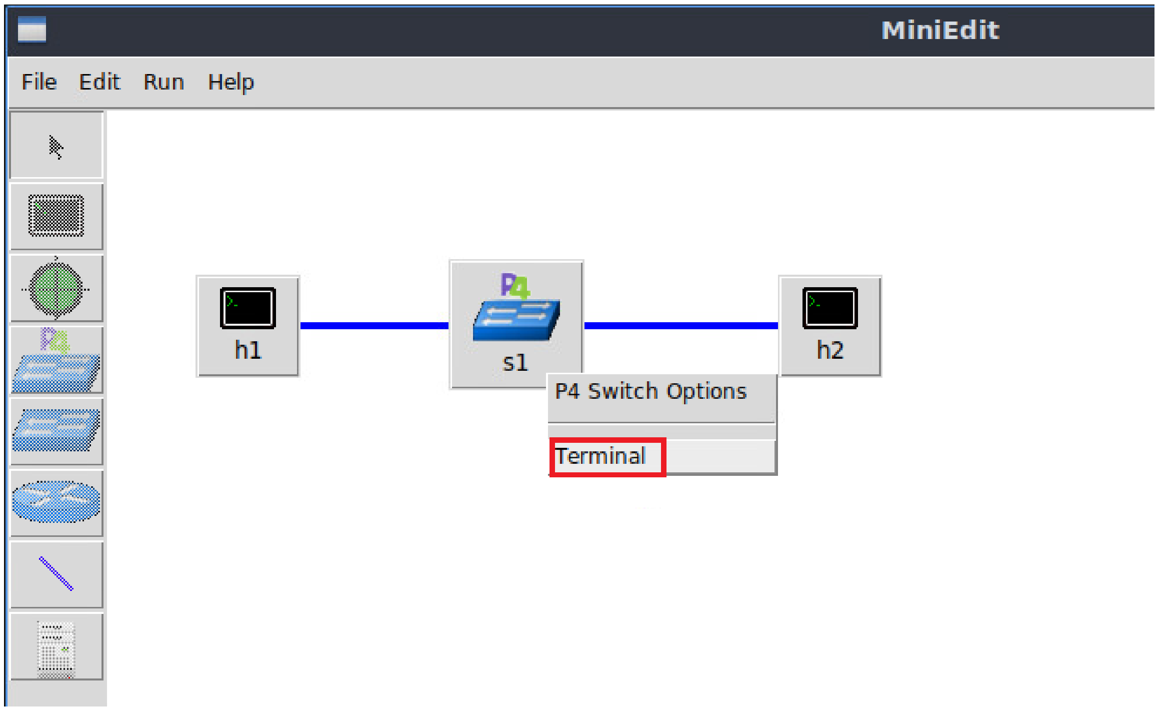
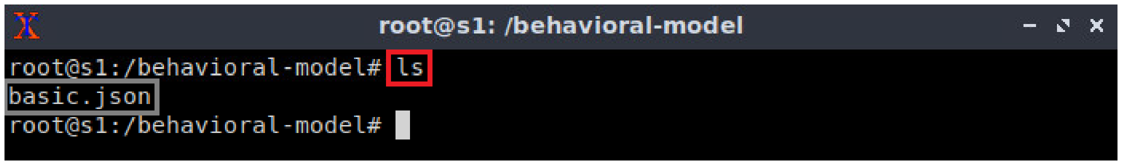


Figure 24. Starting the terminal on the switch.

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
```



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

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

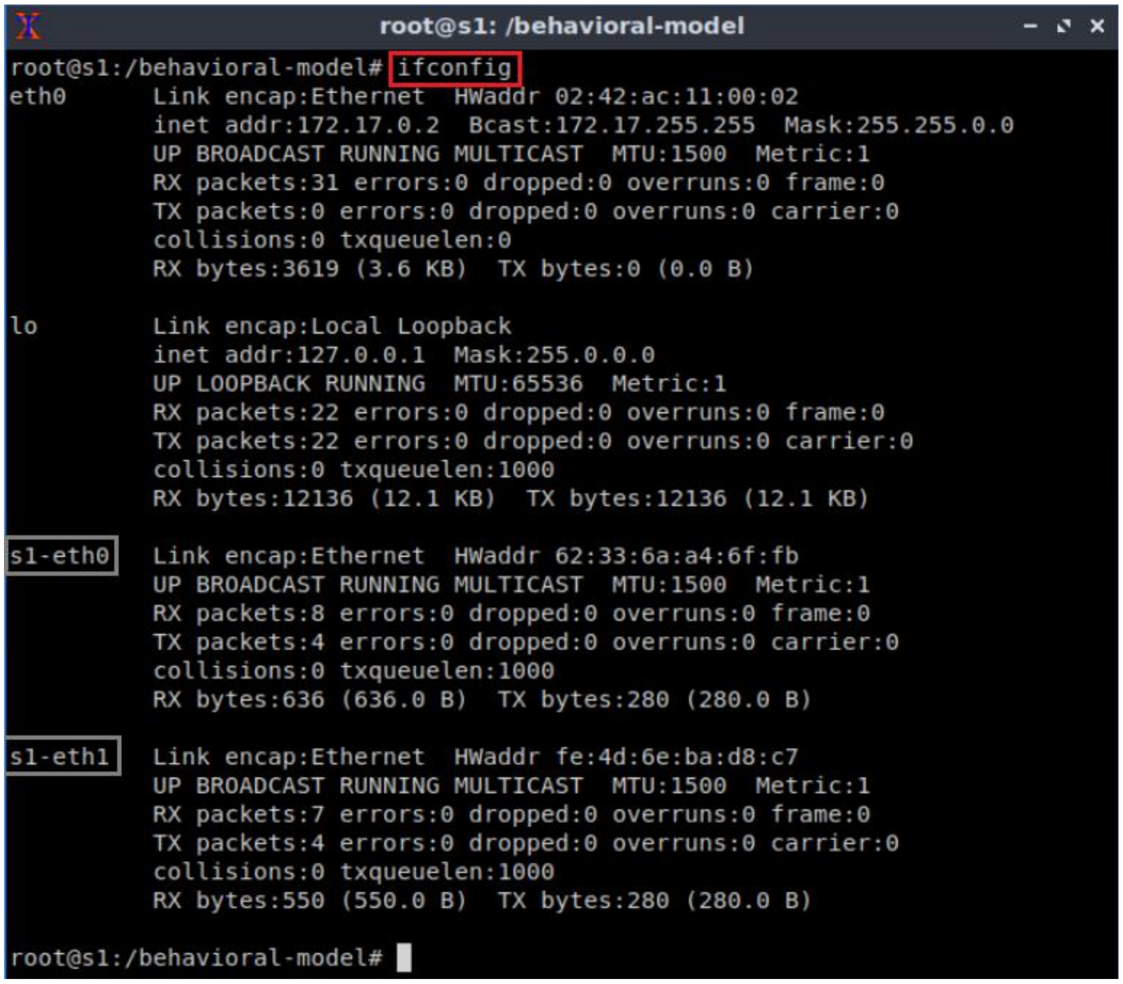
We can see that the switch contains the *basic.json* file that was downloaded to switch s1 after compiling the P4 program.

5 Configuring switch s1

5.1 Mapping the P4 program's ports

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

```
ifconfig
```



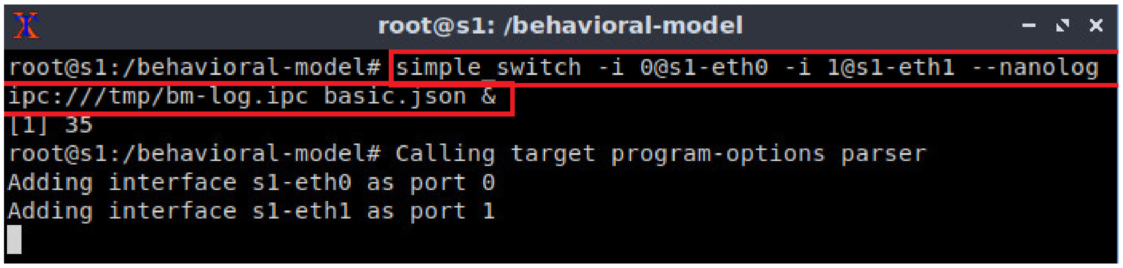
```
root@s1: /behavioral-model
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31 errors:0 dropped:0 overruns:0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

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

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

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

root@s1:/behavioral-model#
```

Figure 26. Displaying switch s1 interfaces.

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

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

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



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

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

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

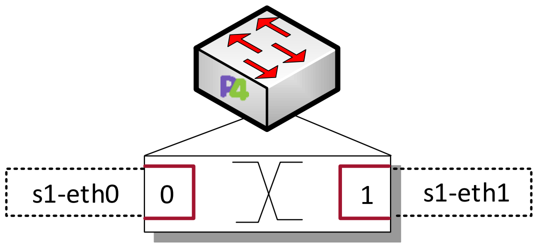
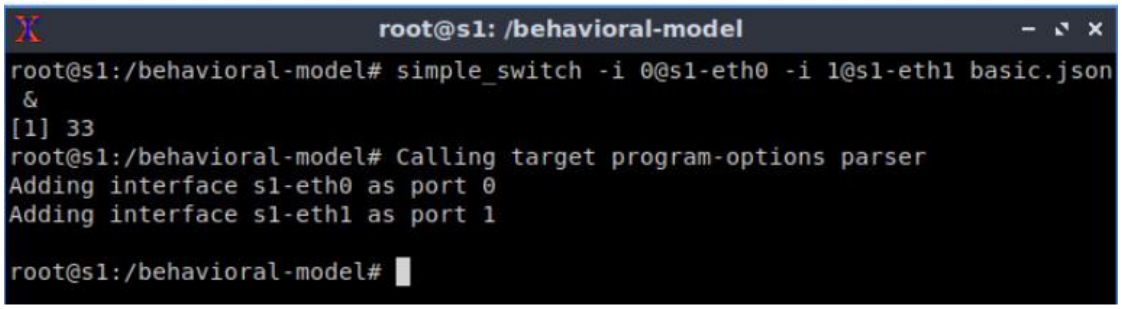


Figure 28. Mapping of the logical interface numbers (0, 1) to the Linux interfaces (*s1-eth0*, *s1-eth1*).

5.2 Loading the rules to the switch

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

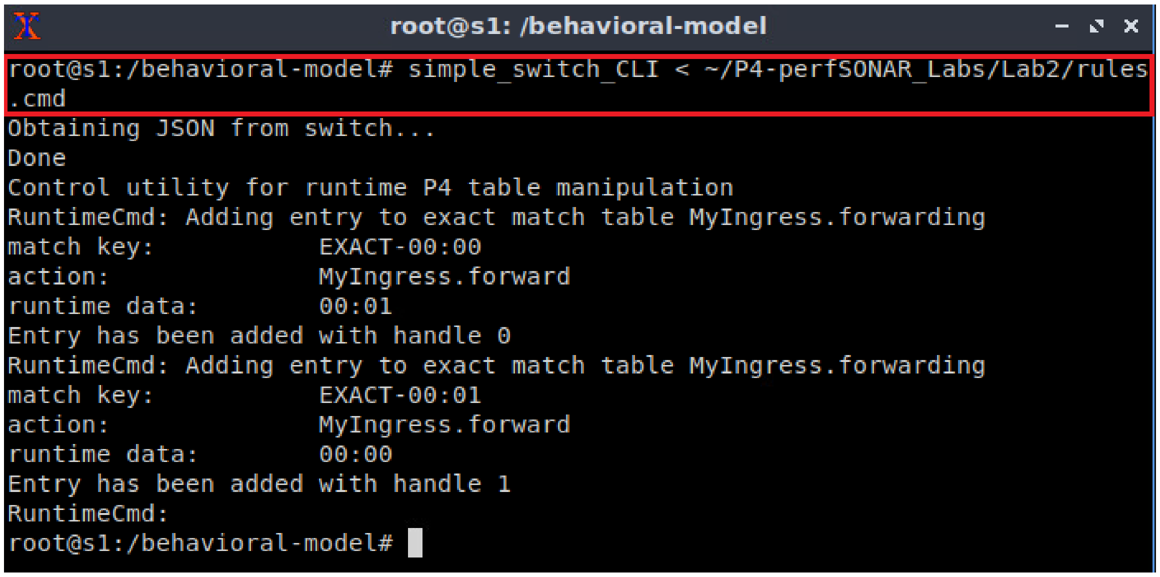


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 29. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/P4-perfSONAR_Labs/Lab2/rules.cmd
```



```
root@s1:/behavioral-model# simple_switch_CLI < ~/P4-perfSONAR_Labs/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# 
```

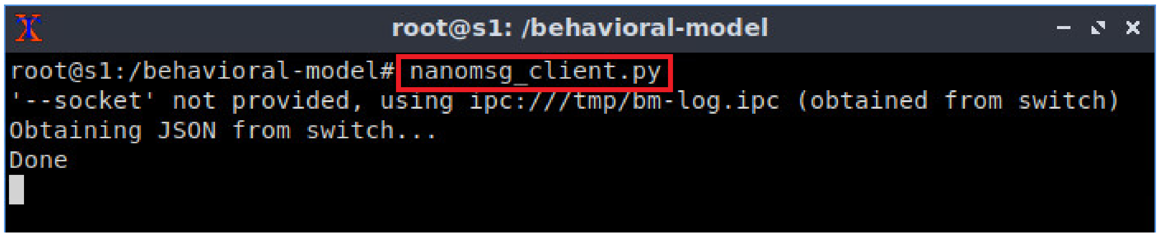
Figure 30. Loading the forwarding table entries into switch s1.

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
```

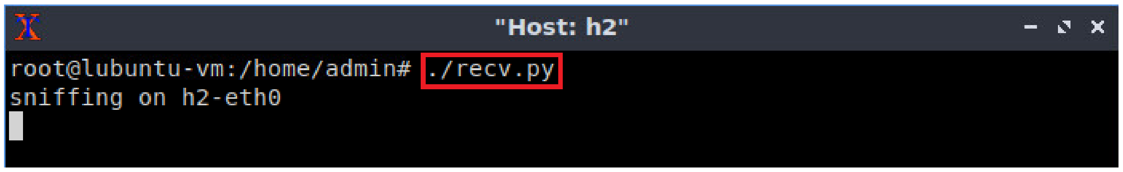


```
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 31. Displaying switch s1 logs.

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

```
./recv.py
```

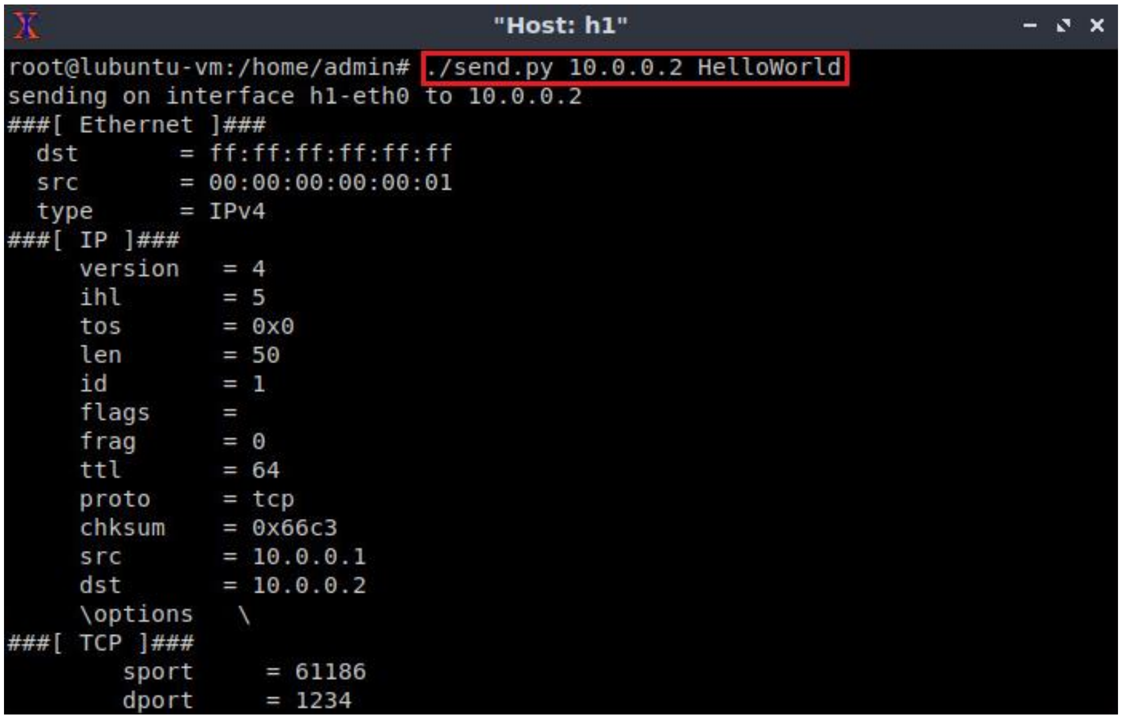


```
root@lubuntu-vm:/home/admin# ./recv.py
sniffing on h2-eth0
```

Figure 32. Listening for incoming packets in host h2.

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

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

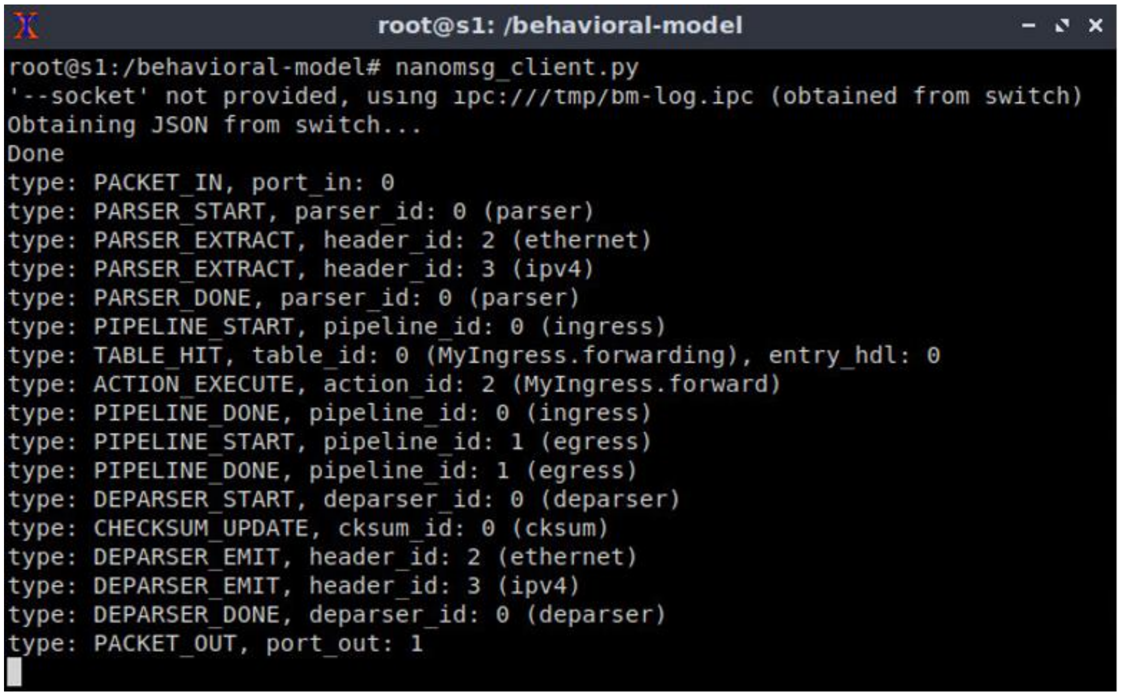


```
"Host: h1"
root@lubuntu-vm:/home/admin# ./send.py 10.0.0.2 HelloWorld
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len   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

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

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

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



```
root@s1: /behavioral-model
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 2. 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.

P4-PERFSONAR LAB SERIES

Lab 3: Measuring Flow's Throughput

Document Version: 06-13-2024

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 Lab scenario 4
2 Lab topology..... 5
2.1 Verifying connectivity between host h1 and host h2 7
3 Developing a P4 program to report per-flow throughput..... 8
3.1 Loading the programming environment..... 8
3.2 Defining a custom header 9
3.3 Performing the cloning operation every one second 11
3.4 Processing and mirroring packets to the collector 17
4 Loading the P4 program..... 25
4.1 Compiling and loading the P4 program to switch s1 25
4.2 Verifying the configuration 27
5 Configuring switch s1 28
5.1 Mapping the P4 program’s ports 28
5.2 Loading the rules to the switch..... 29
5.3 Defining packet mirroring 30
6 Testing and verifying the P4 program..... 31
6.1 Starting the collector 31
6.2 Testing the P4 program..... 32
References 33

Overview

Programmable data planes provide fine-grained monitoring capabilities, allowing for statistics to be reported on a per-flow basis. In this lab, participants will implement a P4 program that utilizes the mirroring operation to report throughput to a collector. For each flow, the switch will clone one packet every second, add the flow's bit count to the cloned packet, and then send it to the collector for analysis. This approach enables detailed flow-level monitoring and measurement of throughput within the network.

Objectives

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

1. Understand how packet cloning works in P4.
2. Understand how packet mirroring works in P4.
3. Report measurements using time intervals.
4. Collect measurements on a per-flow basis.

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 cloning in P4.
2. Section 2: Lab topology.
3. Section 3: Developing a P4 program to report per-flow throughput.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Testing and verifying the P4 program.

1 Introduction

Clone operation generates a new version of a packet, allowing the user to process the cloned packet differently from the original packet. The cloning operation does not disturb the ongoing connection as the original packet can be forwarded to its destination, while the cloned packet is directed to the ingress or egress blocks for additional processing. There are four types of cloning: 1) ingress to ingress; 2) ingress to egress; 3) egress to ingress; and 4) egress to egress.

The user can specify a list of metadata to be preserved by the cloned packets. Multiple lists can be defined in the metadata data, where the order of the defined lists specify their IDs. The order of the lists starts from 0 (i.e., first list has ID 0, second list has ID 1 and so on). Upon cloning, the user can specify which list to use by including its ID in the cloned function. If no lists are defined, the ID of value 0 is used to indicate that no metadata should be preserved. Another parameter of the cloning function is the session ID. The session ID field groups the packets into groups, where different actions can be performed based on the value of this field.

One function of packet cloning is mirroring. Mirroring (also known as port mirroring) is a standard networking functionality used to send a copy of a packet received on a specific port to a networking monitoring system (e.g., collector) on another port[2,3]. To implement the mirroring functionality in P4, the user should identify which packets to be monitored, generate the mirrored instances of the identified packets, and specify the actions to be performed on those instances.

Specifying which packets to be monitored is application dependent. For example, suppose the application should report the flows' throughput every one-second interval. In that case, the switch should calculate the time difference between the arrival time of the current packet and the time the last report of the current flow has been sent. If the time difference is larger than one second, the switch should clone the packet, append the counted bits in the last second to the cloned instances, and forward them to a collector. It is important to differentiate between packet cloning and packet mirroring. The term "clone" is used instead of "mirror" to emphasize that it solely generates a new packet version without requiring additional configuration for mirroring[2].

1.1 Lab scenario

In this lab, participants will develop a P4 application to measure and report the throughput of individual flows in real-time. The application will run on a switch and periodically send the throughput values to a designated collector for display. The switch will use the 5-tuple (source IP, destination IP, source port, destination port, and protocol) to classify packets into distinct flows. For each flow, the switch will calculate the number of bits transmitted within a one-second interval, representing the flow's throughput.

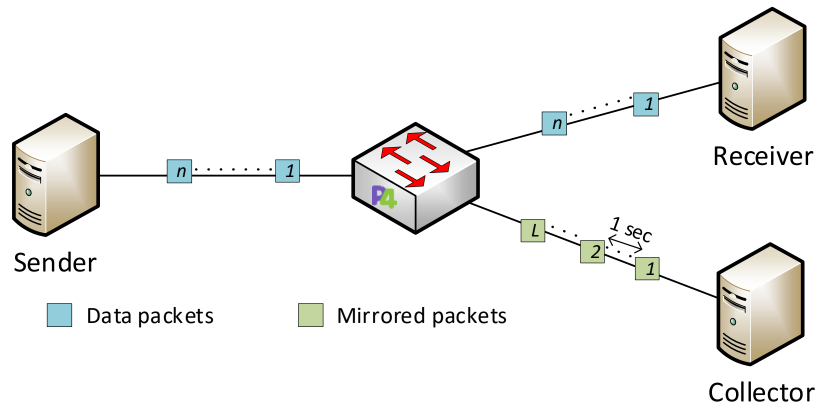


Figure 1. Lab scenario.

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.

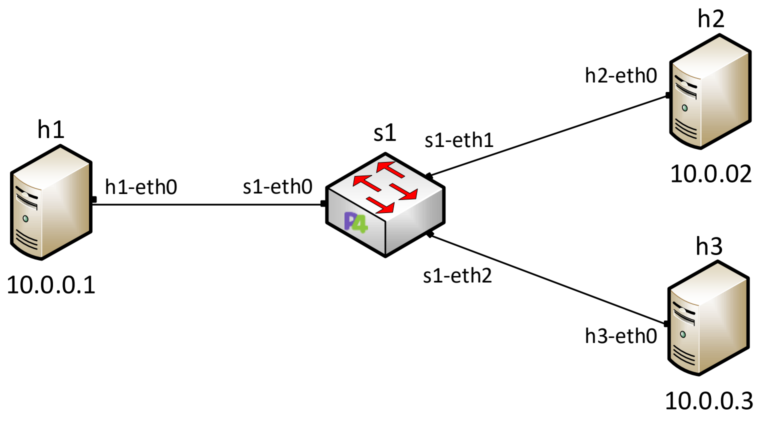


Figure 2. Lab topology.

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

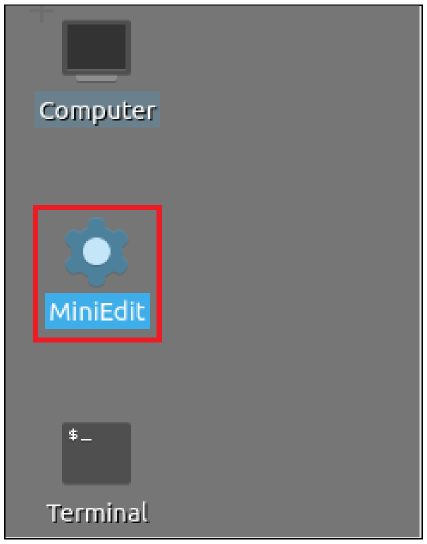


Figure 3. MiniEdit shortcut.

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

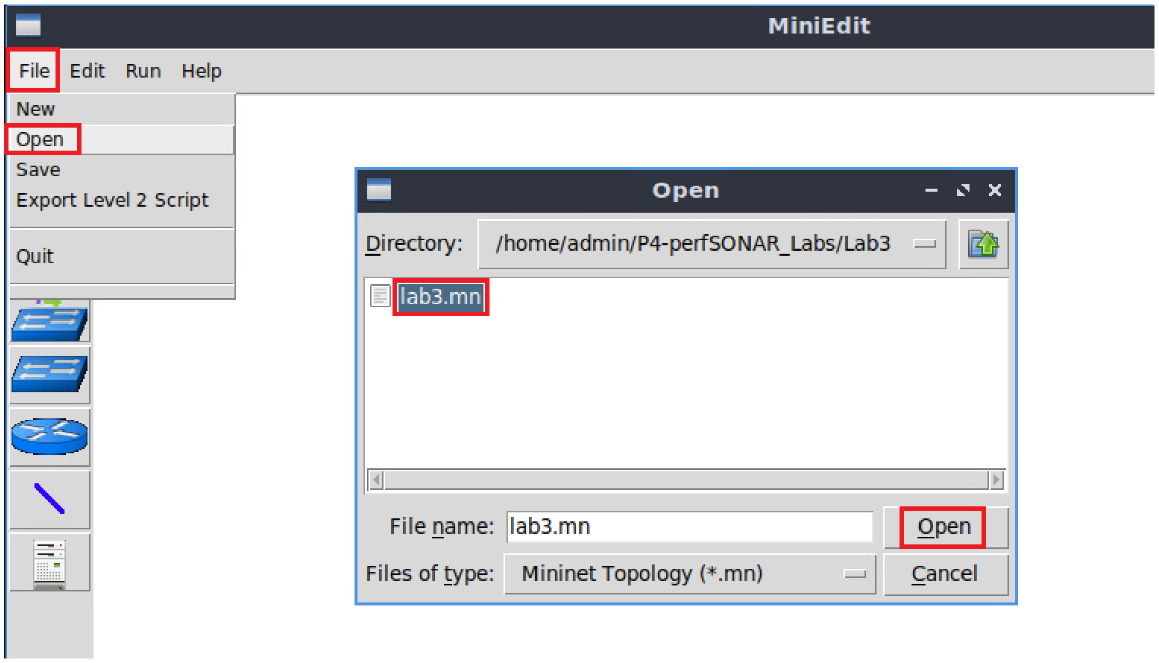


Figure 4. MiniEdit's *Open* dialog.

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

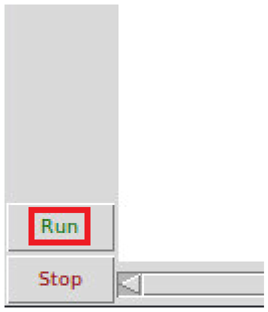


Figure 5. Running the emulation.

2.1 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.

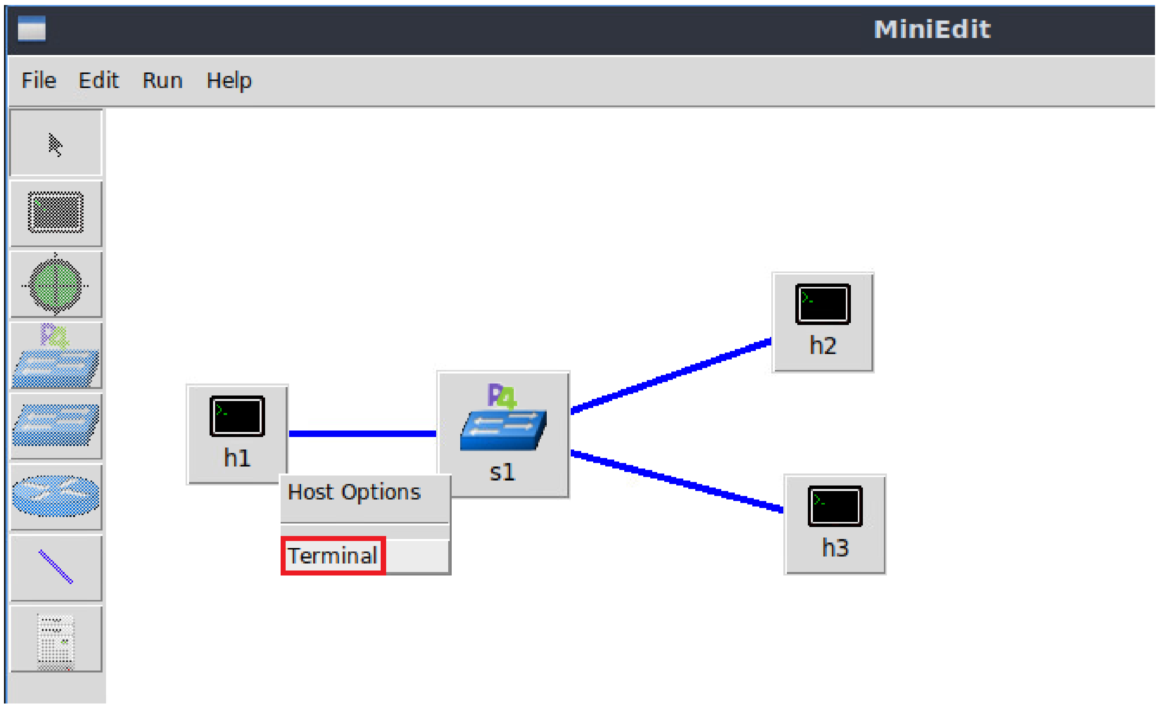
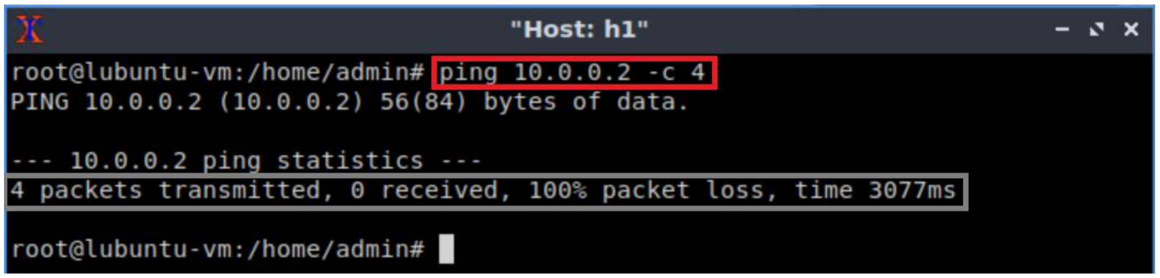


Figure 6. Opening a terminal on host h1.

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.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3077ms

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

Figure 7. 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. Note that there will be no connectivity between any two hosts in the topology before loading a P4 program to the switch.

3 Developing a P4 program to report per-flow throughput

In this section, you will create a P4 program to report per-flow throughput values to the collector. For each flow, the switch will report the counted number of bits per second.

3.1 Loading the programming environment

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

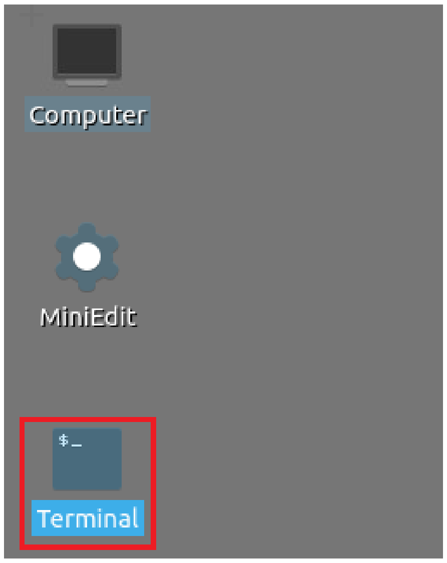
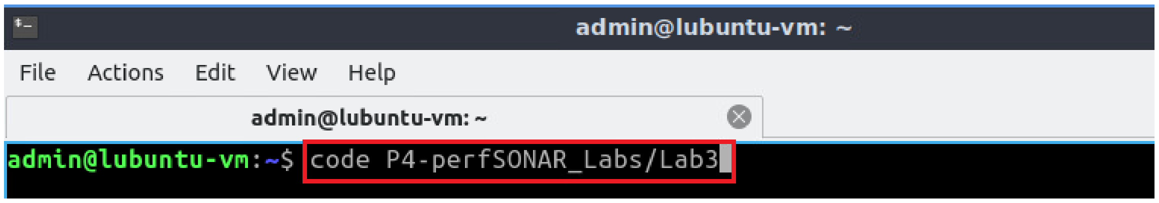


Figure 8. Shortcut to open a Linux terminal.

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-perfSOANR_Labs/Lab3
```

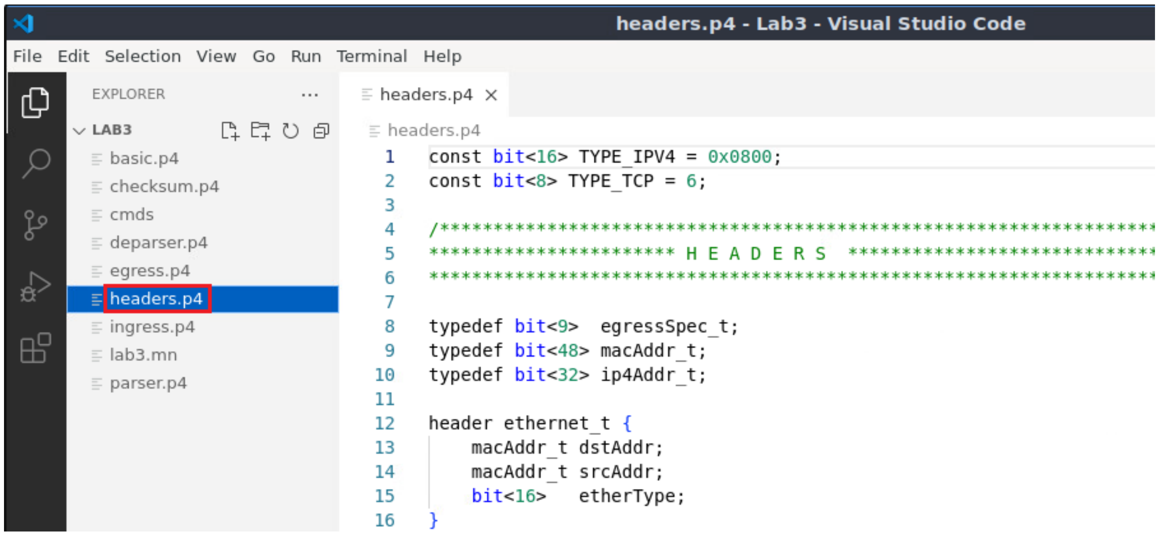


```
admin@lubuntu-vm:~$ code P4-perfSONAR_Labs/Lab3
```

Figure 9. 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.

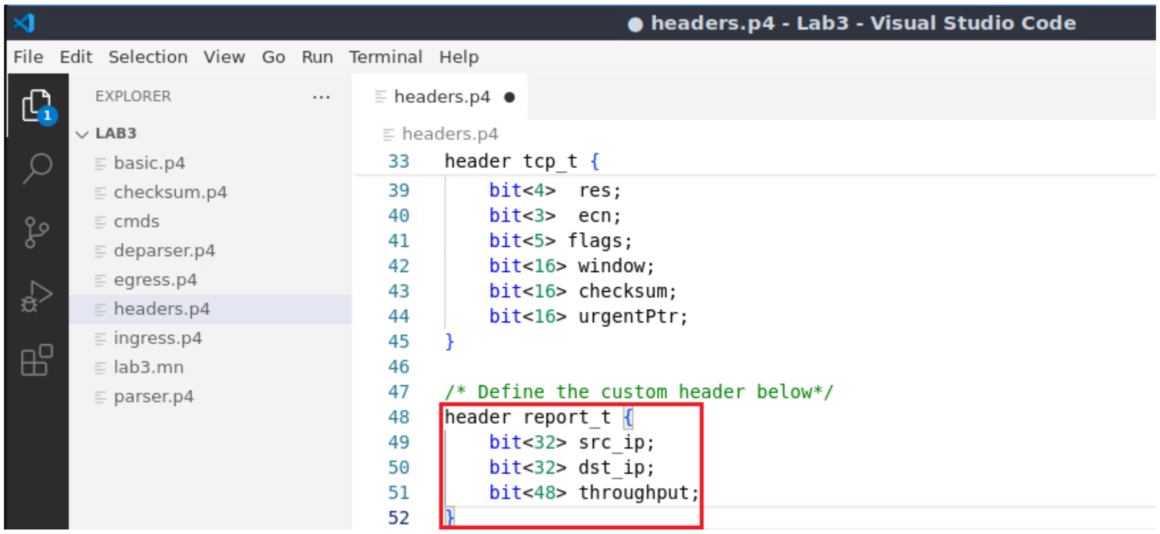


```
const bit<16> TYPE_IPV4 = 0x0800;
const bit<8> TYPE_TCP = 6;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

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

Figure 10. Inspecting the *headers.p4* file.

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

```
header report_t {
        bit<32> src_ip;
        bit<32> dst_ip;
        bit<48> throughput;
}
```

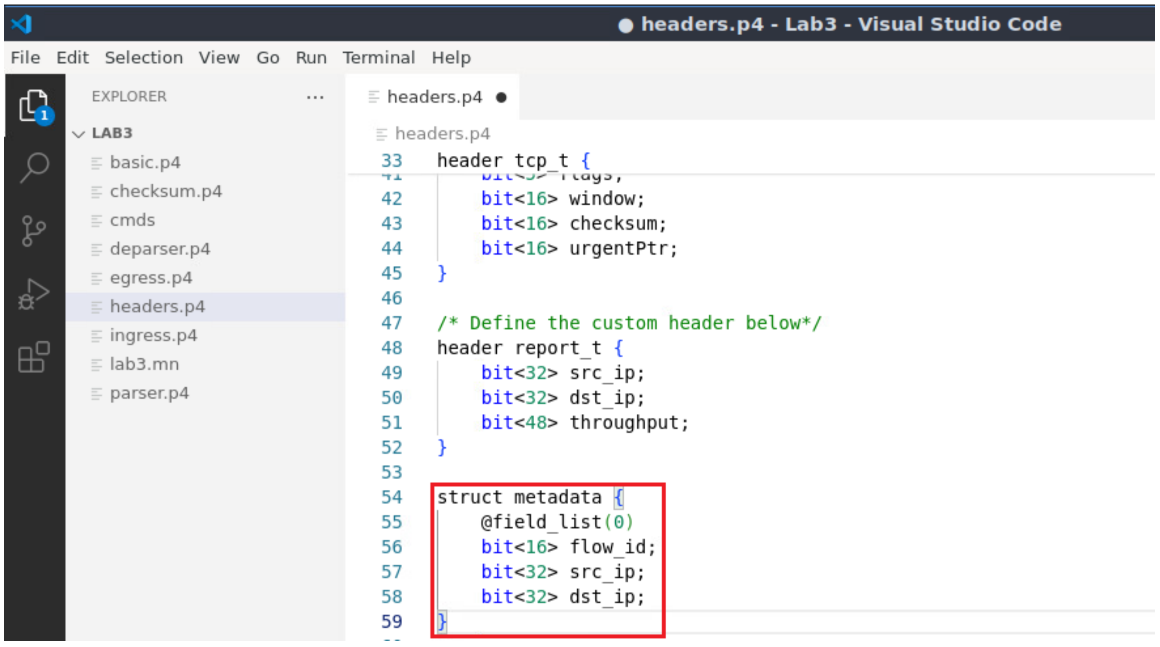


```
header tcp_t {
    bit<4>  res;
    bit<3>  ecn;
    bit<5> flags;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

/* Define the custom header below*/
header report_t {
    bit<32> src_ip;
    bit<32> dst_ip;
    bit<48> throughput;
}
```

Figure 11. Defining a custom header type.

Step 3. Define the following field list by adding the code below to the metadata struct.

```
@field_list(0)
bit<16> flow_id;
bit<32> src_ip;
bit<32> dst_ip;
```



```
header tcp_t {
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

/* Define the custom header below*/
header report_t {
    bit<32> src_ip;
    bit<32> dst_ip;
    bit<48> throughput;
}

struct metadata {
    @field_list(0)
    bit<16> flow_id;
    bit<32> src_ip;
    bit<32> dst_ip;
}
```

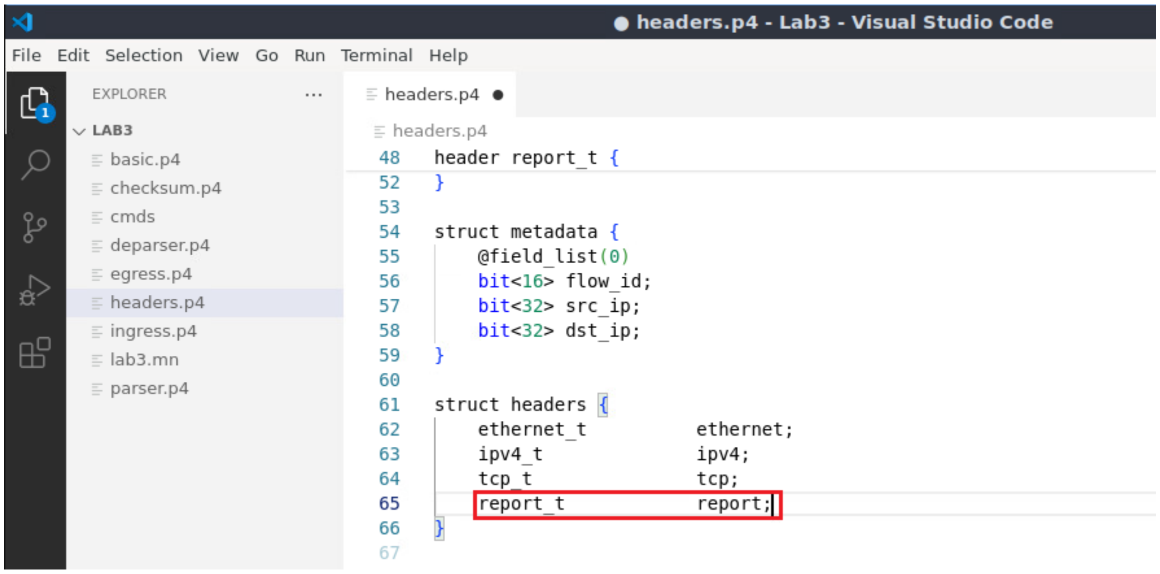
Figure 12. Defining a field list.

The clone function uses a field list to preserve a list of metadata fields after cloning a packet. By including the ID of a field list in the cloning function, the switch preserves all the fields defined by the field list.

In the code above, `@field_list(0)` defines a field list with ID `0`. This list has three metadata fields that will be used to preserve the flow ID the packet belongs to and the source IP address and the destination IP address of the flow.

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

```
report_t report;
```



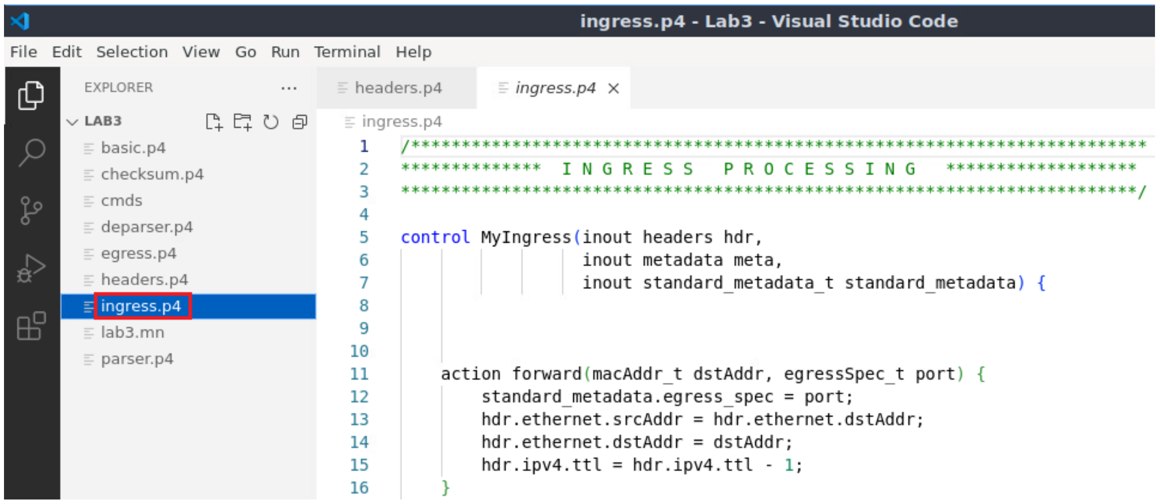
```
header report_t {
}

struct metadata {
    @field_list(0)
    bit<16> flow_id;
    bit<32> src_ip;
    bit<32> dst_ip;
}

struct headers {
    ethernet_t          ethernet;
    ipv4_t              ipv4;
    tcp_t               tcp;
    report_t            report;
}
```

Figure 13. Appending the custom header.

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

3.3 Performing the cloning operation every one second

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.

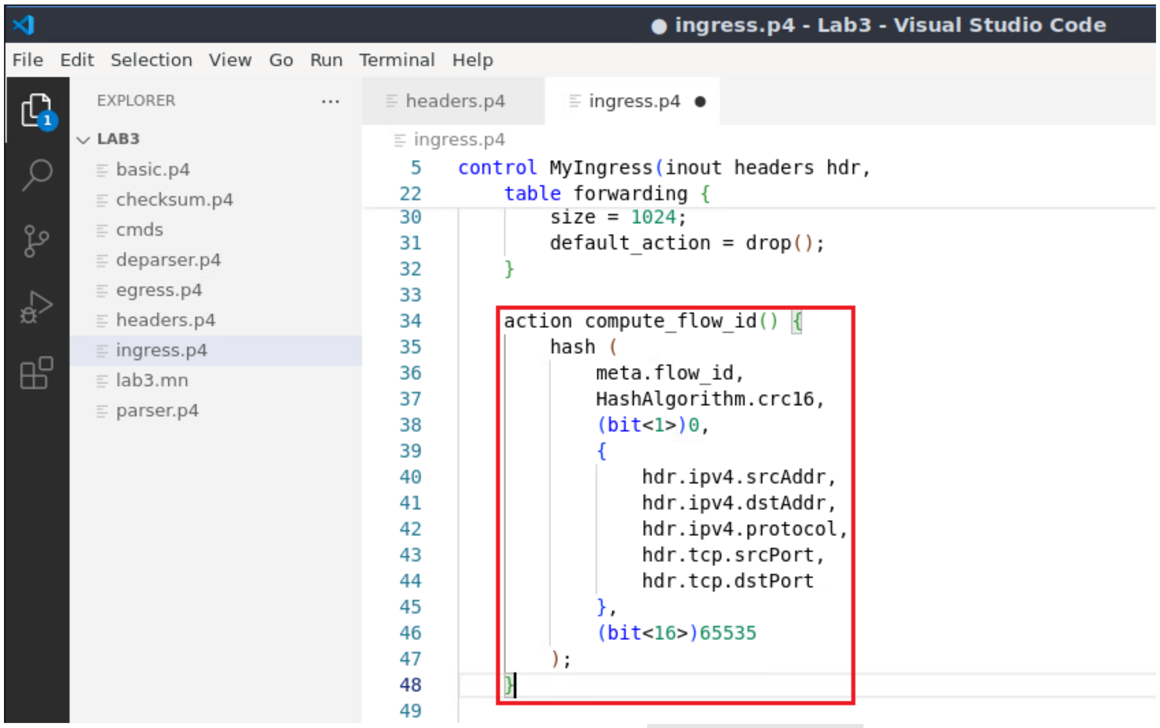


```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

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


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

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

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

```
action compute_flow_id() {
    hash (
         meta.flow_id,
         HashAlgorithm.crc16,
         (bit<1>)0,
         {
          hdr.ipv4.srcAddr,
          hdr.ipv4.dstAddr,
          hdr.ipv4.protocol,
          hdr.tcp.srcPort,
          hdr.tcp.dstPort
          },
          (bit<16>)65535
    );
}
```



```
control MyIngress(inout headers hdr,
    table forwarding {
        size = 1024;
        default_action = drop();
    }

    action compute_flow_id() {
        hash (
            meta.flow_id,
            HashAlgorithm.crc16,
            (bit<1>)0,
            {
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.ipv4.protocol,
                hdr.tcp.srcPort,
                hdr.tcp.dstPort
            },
            (bit<16>)65535
        );
    }
```

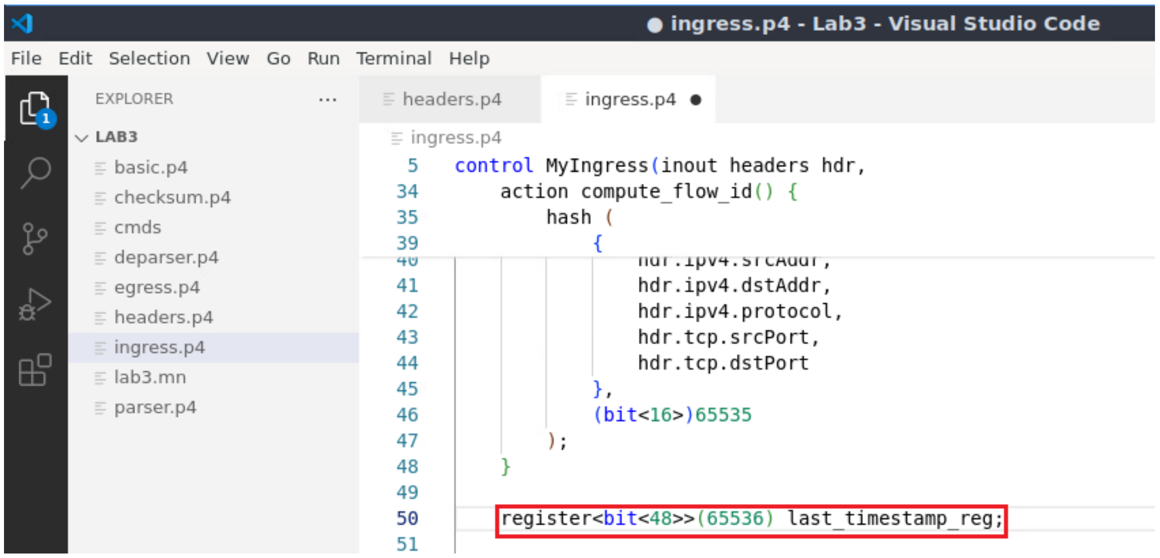
Figure 15. Defining the action `compute_flow_id`.

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

- `meta.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`, `hdr.ipv4.dstAddr`, `hdr.ipv4.protocol`, `hdr.tcp.srcPort`, `hdr.tcp.dstPort`: the data to be hashed.
- `bit<16>65535`: the maximum value produced by the hash algorithm.

Step 3. Add the following code to define a register array that will store the timestamp of the last report sent.

```
register<bit<48>>(65536) last_timestamp_reg;
```

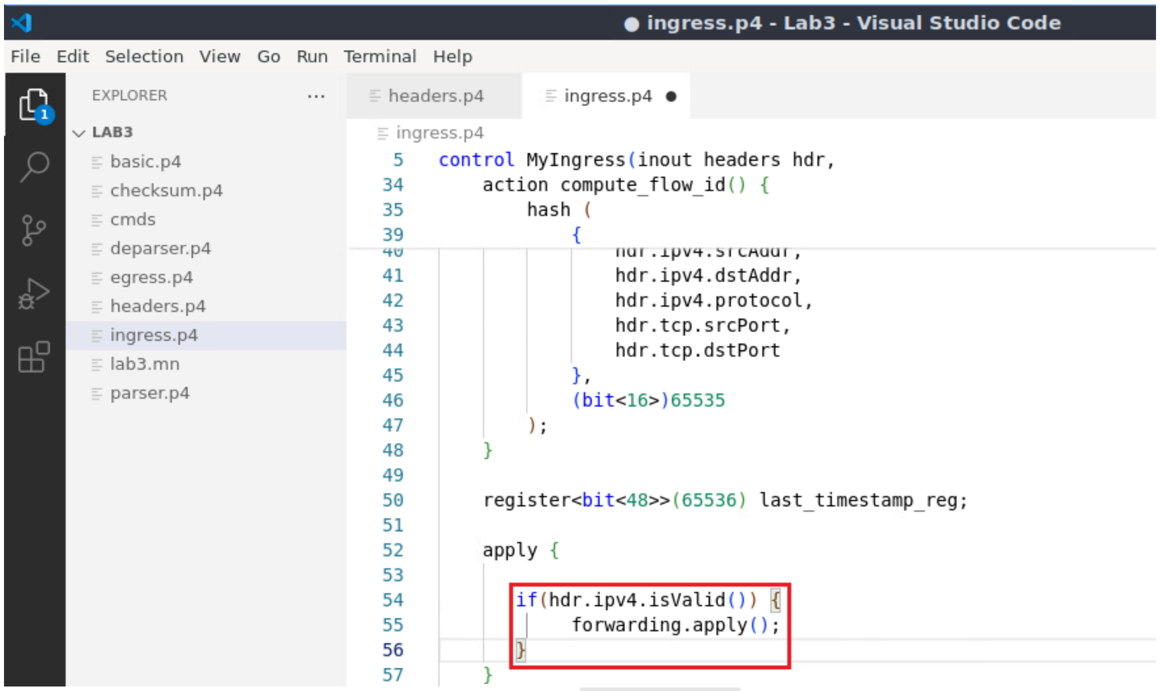


```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
        hash (
            {
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.ipv4.protocol,
                hdr.tcp.srcPort,
                hdr.tcp.dstPort
            },
            (bit<16>)65535
        );
    }

    register<bit<48>>(65536) last_timestamp_reg;
```

Figure 16. Defining a register array.

Step 4. Add the following code inside the *apply* block to apply the `forwarding` table if the packet has the IPv4 header.

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

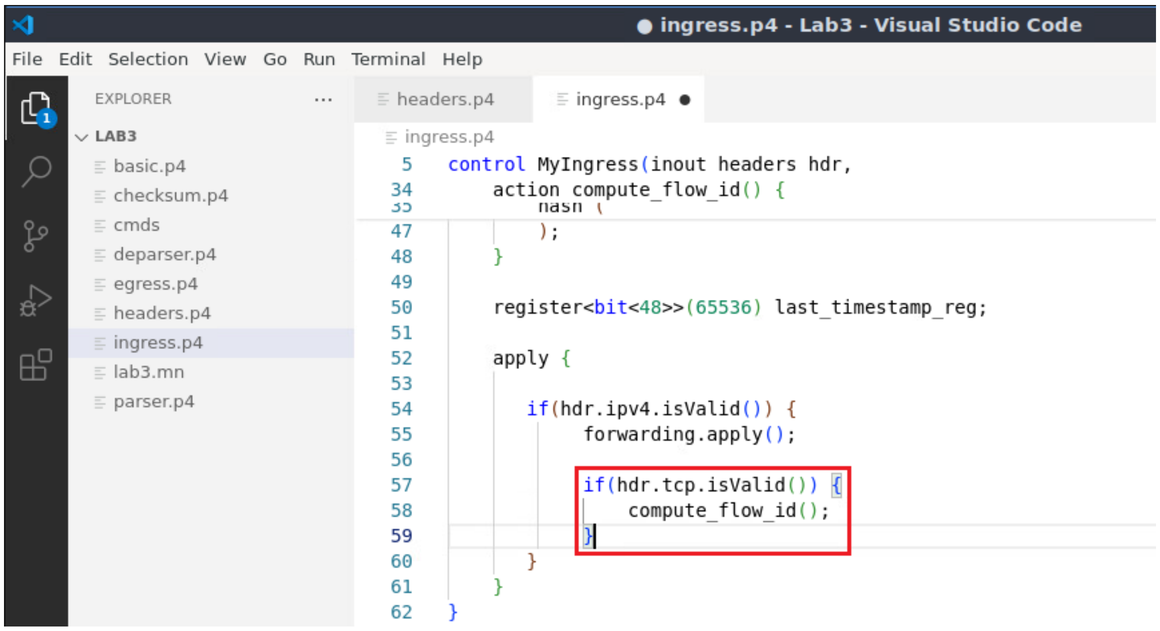


```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
        hash (
            {
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.ipv4.protocol,
                hdr.tcp.srcPort,
                hdr.tcp.dstPort
            },
            (bit<16>)65535
        );
    }

    register<bit<48>>(65536) last_timestamp_reg;

    apply {

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

Figure 17. Applying `forwarding` table.

Step 5. Add the following code to compute the ID of the flow if the packet has the IPv4 and the TCP headers.

```
if(hdr.tcp.isValid()){
    compute_flow_id();
}
```

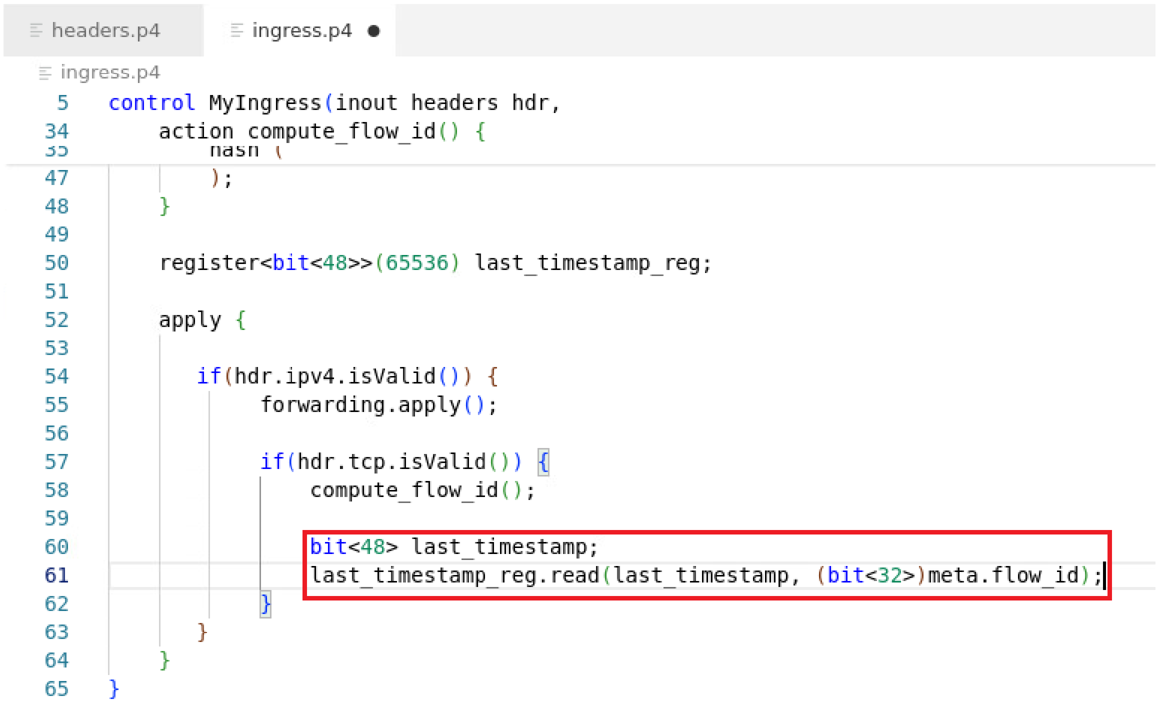


```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
        hash (
        );
    }

    register<bit<48>>(65536) last_timestamp_reg;

    apply {

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

            if(hdr.tcp.isValid()) {
                compute_flow_id();
            }
        }
    }
}
```

Figure 18. Computing flow ID.

Step 6. Add the following code to define a new variable, retrieve the timestamp of the last report sent from the register array, and store it in the defined variable.

```
bit<48> last_timestamp;
last_timestamp_reg.read(last_timestamp, (bit<32>)meta.flow_id);
```

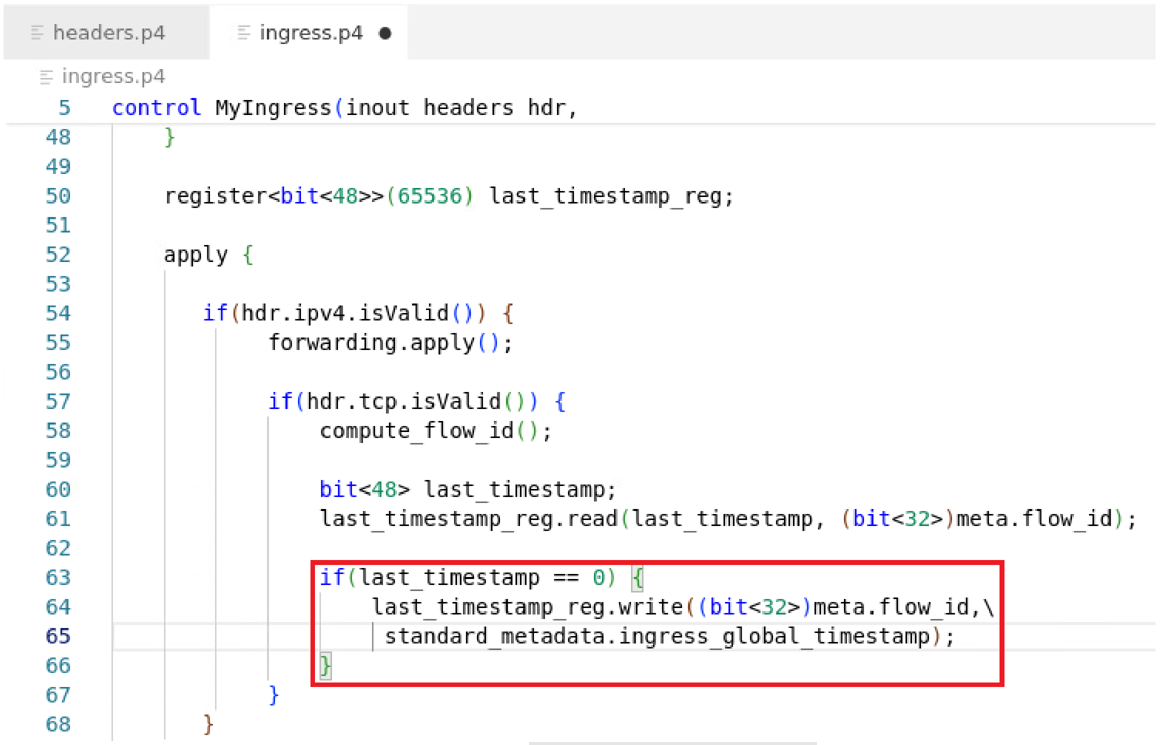


```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
        hash (
        );
    }

    register<bit<48>>(65536) last_timestamp_reg;

    apply {

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

            if(hdr.tcp.isValid()) {
                compute_flow_id();

                bit<48> last_timestamp;
                last_timestamp_reg.read(last_timestamp, (bit<32>)meta.flow_id);
            }
        }
    }
}
```

Figure 19. Retrieving the timestamp of the last report.

Step 7. Add the following code to modify the `last_timestamp_reg` to the current timestamp if the packet does not belong to an existing flow (i.e., it is the first packet of the flow).

```
if(last_timesatmp == 0){
    last_timestamp_reg.write((bit<32>)meta.flow_id,\
     standard_metadata.ingress_global_timestamp);
}
```



```
control MyIngress(inout headers hdr,
    }

    register<bit<48>>(65536) last_timestamp_reg;

    apply {

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

            if(hdr.tcp.isValid()) {
                compute_flow_id();

                bit<48> last_timestamp;
                last_timestamp_reg.read(last_timestamp, (bit<32>)meta.flow_id);

                if(last_timestamp == 0) {
                    last_timestamp_reg.write((bit<32>)meta.flow_id,\
                     standard_metadata.ingress_global_timestamp);
                }
            }
        }
```

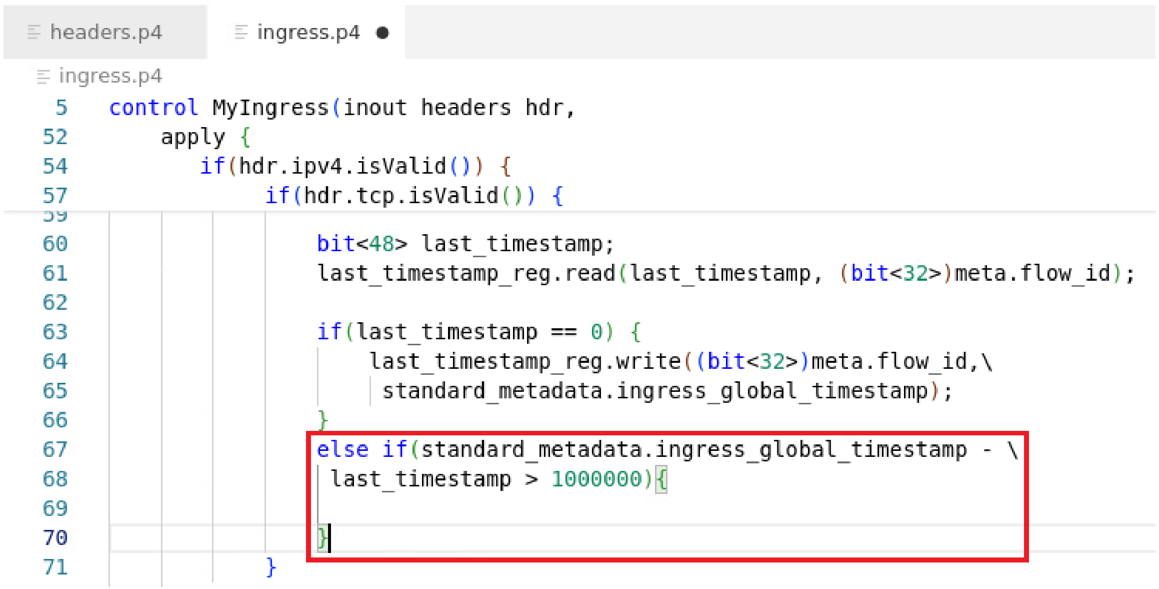
Figure 20. Setting the value of `last_timestamp_reg` for new flows.

The default value of register in P4 is 0. The code above checks if the value of the register `last_timestamp_reg` at index `meta.flow_id` is 0. If yes, then the current packet is the first packet of the flow. The `last_timestamp_reg` value will be set to the current time using the `standard_metadata.ingress_global_timestamp`.

If the value of the `last_timestamp_reg` at the index `meta.flow_id` is not 0, then the program should check the time difference between the last report sent and current time. If the time difference is larger than a predefined threshold, a new report should be sent.

Step 8. Add the following code to check if 1 second has passed since the last report sent. Note that `1000000` is in microseconds (i.e., it is equivalent to 1 second).

```
else if(standard_metadata.ingress_global_timesatmp -\
 last_timestamp > 1000000){
}
```



headers.p4 ingress.p4 ●

ingress.p4

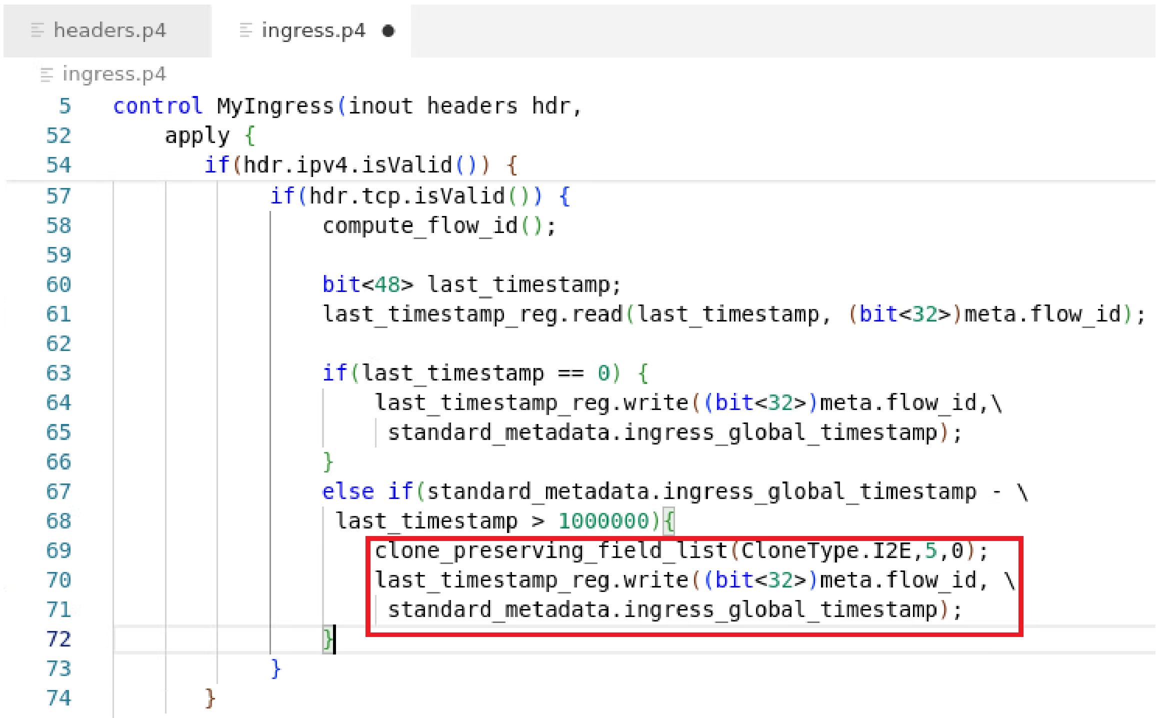
```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            if(hdr.tcp.isValid()) {

                bit<48> last_timestamp;
                last_timestamp_reg.read(last_timestamp, (bit<32>)meta.flow_id);

                if(last_timestamp == 0) {
                    last_timestamp_reg.write((bit<32>)meta.flow_id,\
                     standard_metadata.ingress_global_timestamp);
                }
                else if(standard_metadata.ingress_global_timestamp - \
                 last_timestamp > 1000000){

                }
            }
```

Figure 21. Checking if 1 second has passed since the last sent report.

If the condition of the *else if* statement is satisfied (i.e., 1 second has passed since the last duration), then a new report should be sent and the value of the `last_timestamp_reg` should be set to the current time.

Step 9. Add the following code inside the `else if` statement to clone the current packet and set the timestamp of the last report sent to the current time.

```
clone_preserving_field_list(CloneType.I2E, 5, 0);
last_timestamp_reg.write((bit<32>)meta.flow_id,\
 standard_metadata.ingress_global_timestamp);
```



headers.p4 ingress.p4 ●

ingress.p4

```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            if(hdr.tcp.isValid()) {
                compute_flow_id();

                bit<48> last_timestamp;
                last_timestamp_reg.read(last_timestamp, (bit<32>)meta.flow_id);

                if(last_timestamp == 0) {
                    last_timestamp_reg.write((bit<32>)meta.flow_id,\
                     standard_metadata.ingress_global_timestamp);
                }
                else if(standard_metadata.ingress_global_timestamp - \
                 last_timestamp > 1000000){
                    clone_preserving_field_list(CloneType.I2E,5,0);
                    last_timestamp_reg.write((bit<32>)meta.flow_id, \
                     standard_metadata.ingress_global_timestamp);
                }
            }
        }
```

Figure 22. Cloning the packet and modifying the timestamp of the last report sent.

The action `clone_preserving_field_list` has three parameters:

- `Clone type`: this parameter indicates the cloning type (e.g., ingress to egress).
- `Session ID`: this parameter indicates the session ID to be attached to the cloned packets. The user defines the mirroring port of the cloned packets using their session IDs.
- `ID of the field list`: this parameter indicates which list of metadata fields to preserve after cloning the packet. A field list should be defined in the metadata header.

In the code above, the parameters have the following values:

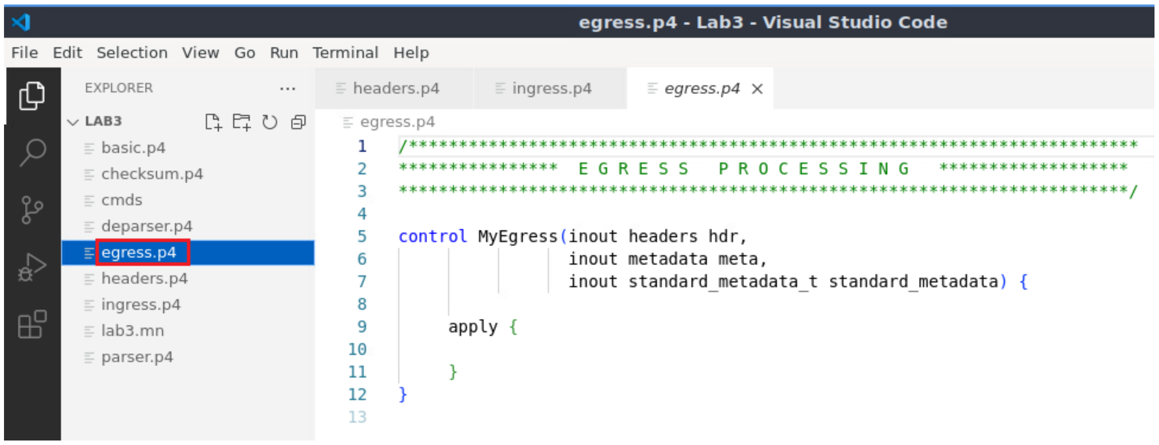
- `Clone type` is `CloneType.I2E` indicating that the cloning will be from the ingress to the egress.
- `Session ID` is `5` indicating that the cloned packets will have session ID of value 8.
- `ID of the field list` is `0` indicating that the fields of the field list with ID 0 will be preserved.

This P4 program will use cloned packets to send the throughput reports to the collector. The egress block will process the cloned packets to include the number of bits of their flow within the last second, and then mirror them to the collector.

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

3.4 Processing and mirroring packets to the collector

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.

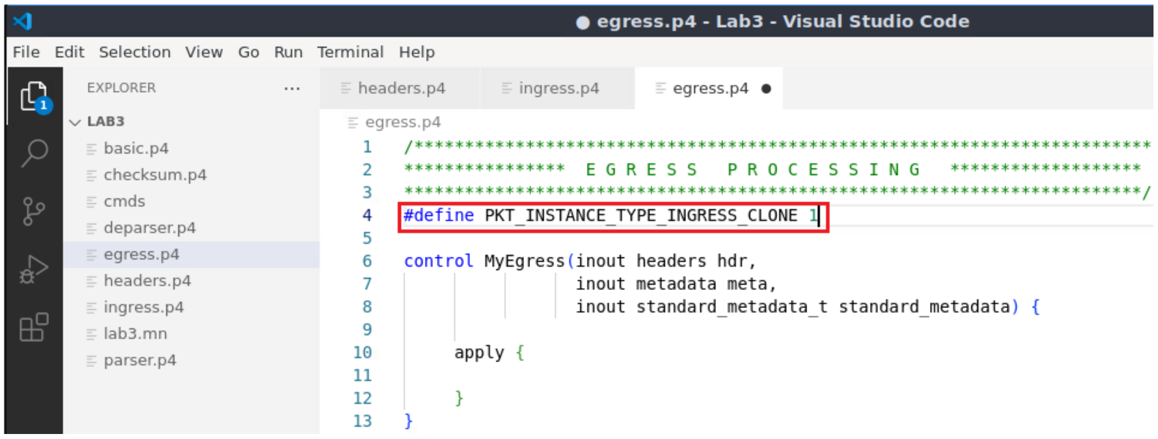


```
/*************************************************************************
***************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

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

    apply {

    }
}
```

Figure 23. Inspecting the egress.*p4* file.

Step 2. Add the following code in the *egress.p4* file to assign value `1` to the variable `PKT_INSTANCE_TYPE_INGRESS_CLONE`.

```
#define PKT_INSTANCE_TYPE_INGRESS_CLONE 1
```

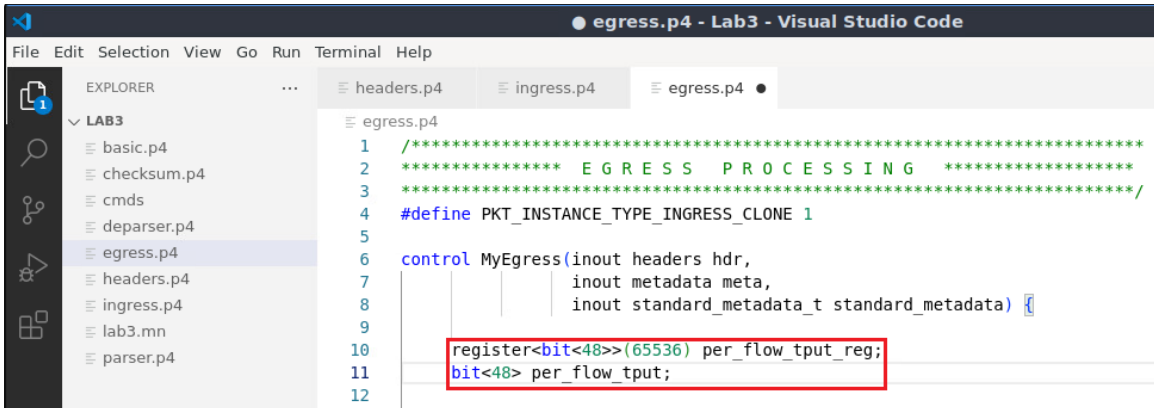


```
/*************************************************************************
****************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/
#define PKT_INSTANCE_TYPE_INGRESS_CLONE 1

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

    apply {

    }
}
```

Figure 24. Defining the variable *PKT_INSTANCE_TYPE_INGRESS_CLONE* with value 1.

Step 3. Add the following code to define a register array and a variable.

```
register<bit<48>>(65536) per_flow_tput_reg;
bit<48> per_flow_tput;
```

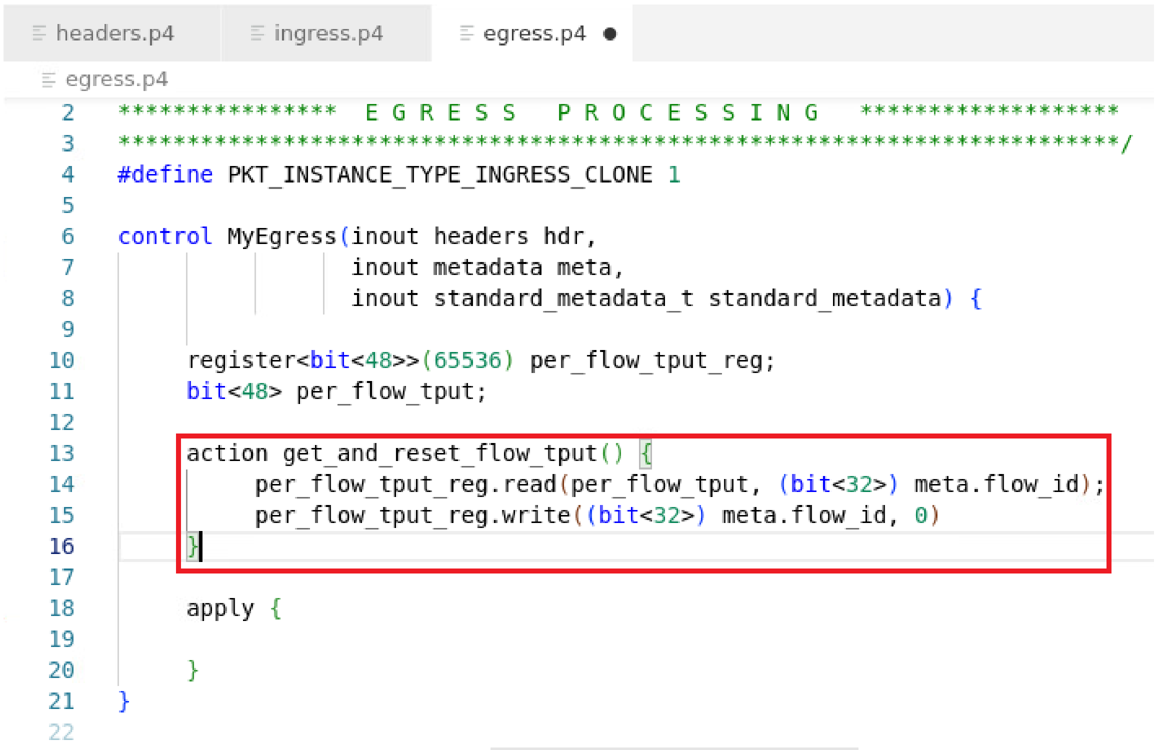


```
/*************************************************************************
****************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/
#define PKT_INSTANCE_TYPE_INGRESS_CLONE 1

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

    register<bit<48>>(65536) per_flow_tput_reg;
    bit<48> per_flow_tput;

```

Figure 25. Defining a register and a variable.

Step 4. Add the following code to define the action `get_and_reset_flow_tput`.

```
action get_and_reset_flow_tput(){
   per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
   per_flow_tput_reg.write((bit<32>) meta.flow_id, 0);
}
```



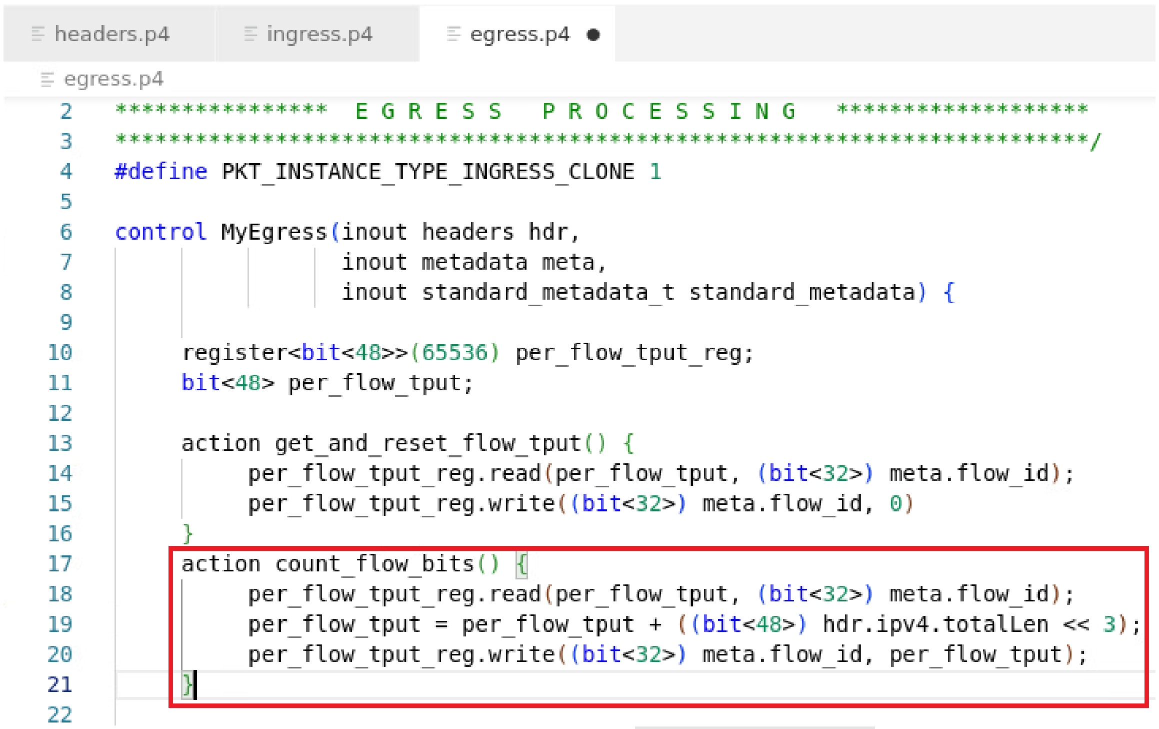
```
****************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/
#define PKT_INSTANCE_TYPE_INGRESS_CLONE 1

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

    register<bit<48>>(65536) per_flow_tput_reg;
    bit<48> per_flow_tput;

    action get_and_reset_flow_tput() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, 0)
    }

    apply {

    }
}
```

Figure 26. Defining the `get_and_reset_flow_tput` action.

In the code above, the action `get_and_reset_flow_tput` first retrieves the throughput of the flow from the register `per_flow_tput_reg` and stores it in the variable `per_flow_tput`, then it resets the value of the register array at the same index to 0. Note that `meta.flow_id` metadata is used to index the register array in the retrieving and in the resetting processes.

Step 5. Add the following code to define the action `count_flow_bits`.

```
action count_flow_bits(){
   per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
   per_flow_tput = per_flow_tput + ((bit<48>)hdr.ipv4.totalLen << 3);
   per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
}
```



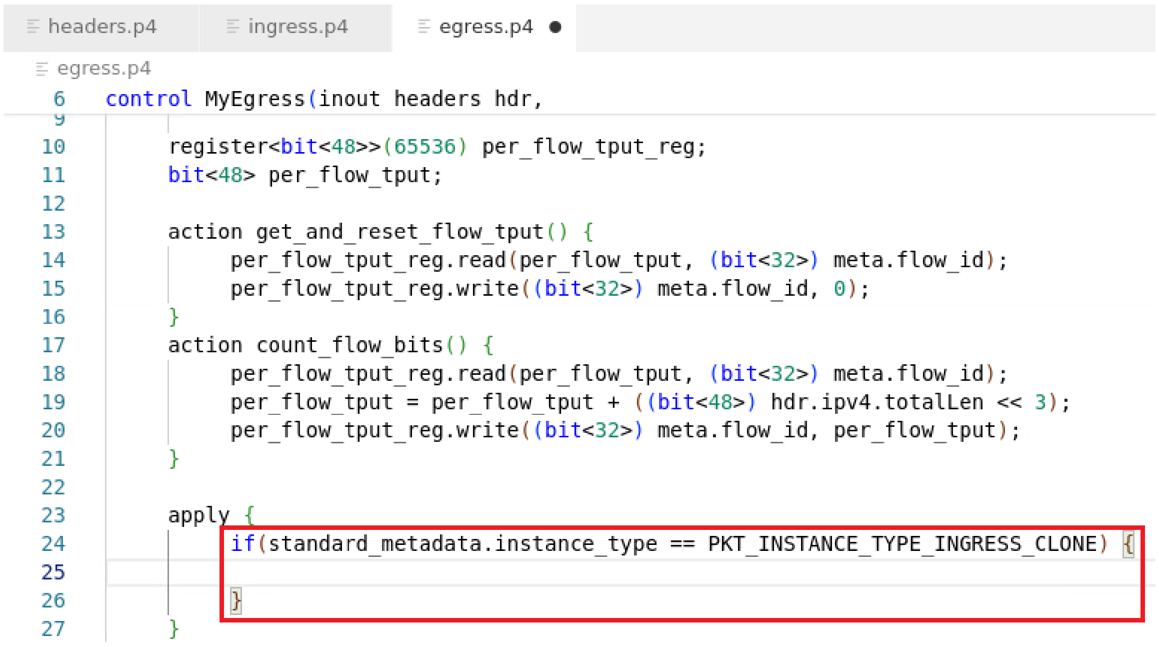
```
****************  E G R E S S   P R O C E S S I N G   *******************
**********************************************************************/
#define PKT_INSTANCE_TYPE_INGRESS_CLONE 1

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

    register<bit<48>>(65536) per_flow_tput_reg;
    bit<48> per_flow_tput;

    action get_and_reset_flow_tput() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, 0)
    }
    action count_flow_bits() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput = per_flow_tput + ((bit<48>) hdr.ipv4.totalLen << 3);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
    }

```

Figure 27. Defining the action `count_flow_bits`.

In the code above, the action `count_flow_bits` first retrieves the number of bits of the current flow from the register `per_flow_tput_reg` and stores it inside the variable `per_flow_tput`. Then, the action adds the number of bits of the current packet to the variable `per_flow_tput`. Finally, the action stores back the updated number of bits in the register. Note that the value of `hdr.ipv4.totalLen` is shifted by 3 (i.e., `hdr.ipv4.totalLen << 3`) to transform the value from bytes to bits.

Step 6. Add the following code inside the *apply* block to check if the current packet is a cloned instance.

```
if(standard_metadata.instance_type == PKI_INSTANCE_TYPE_INGRESS_CLONE){
}
```



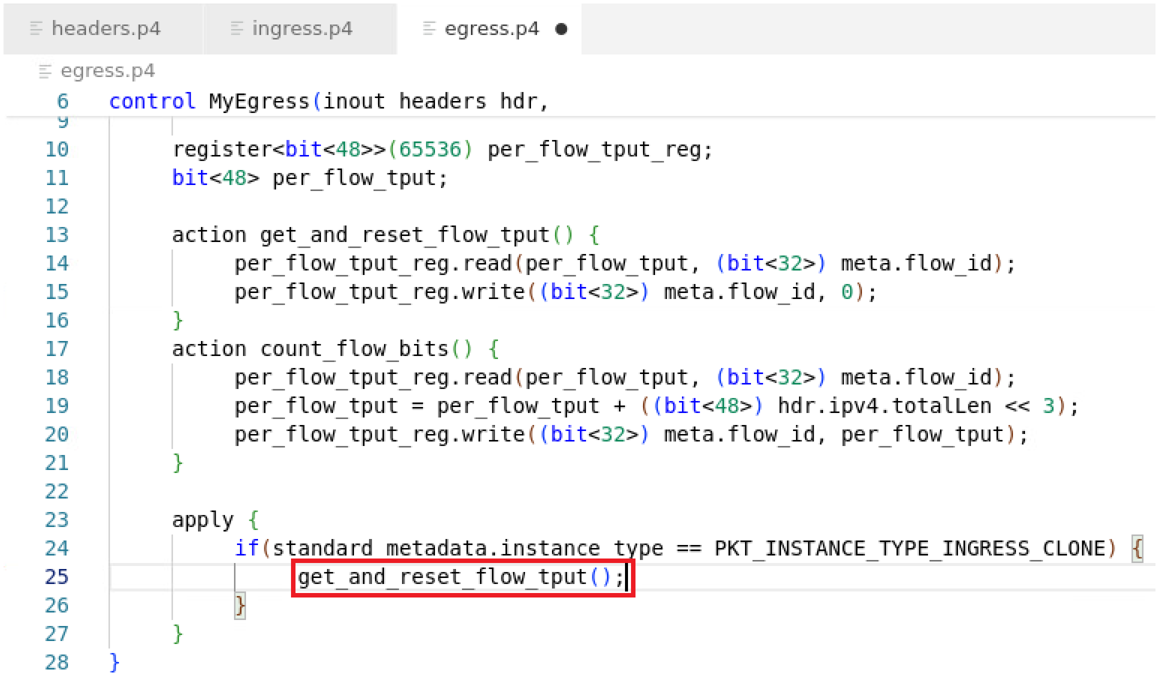
```
control MyEgress(inout headers hdr,

    register<bit<48>>(65536) per_flow_tput_reg;
    bit<48> per_flow_tput;

    action get_and_reset_flow_tput() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, 0);
    }
    action count_flow_bits() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput = per_flow_tput + ((bit<48>) hdr.ipv4.totalLen << 3);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {

        }
    }
```

Figure 28. Checking the instance type of the packet.

In the code above, `standard_metadata.instance_type` specifies if the packet is an original instance or a cloned instance. If the instance type is 1, then the instance is a cloned packet. Otherwise, the instance is an original packet. Thus, the *if statement* checks if the current packet is cloned instance.

Step 7. Add the following code inside the *if statement* to apply the action `get_and_reset_flow_tput`.

```
get_and_reset_flow_tput();
```



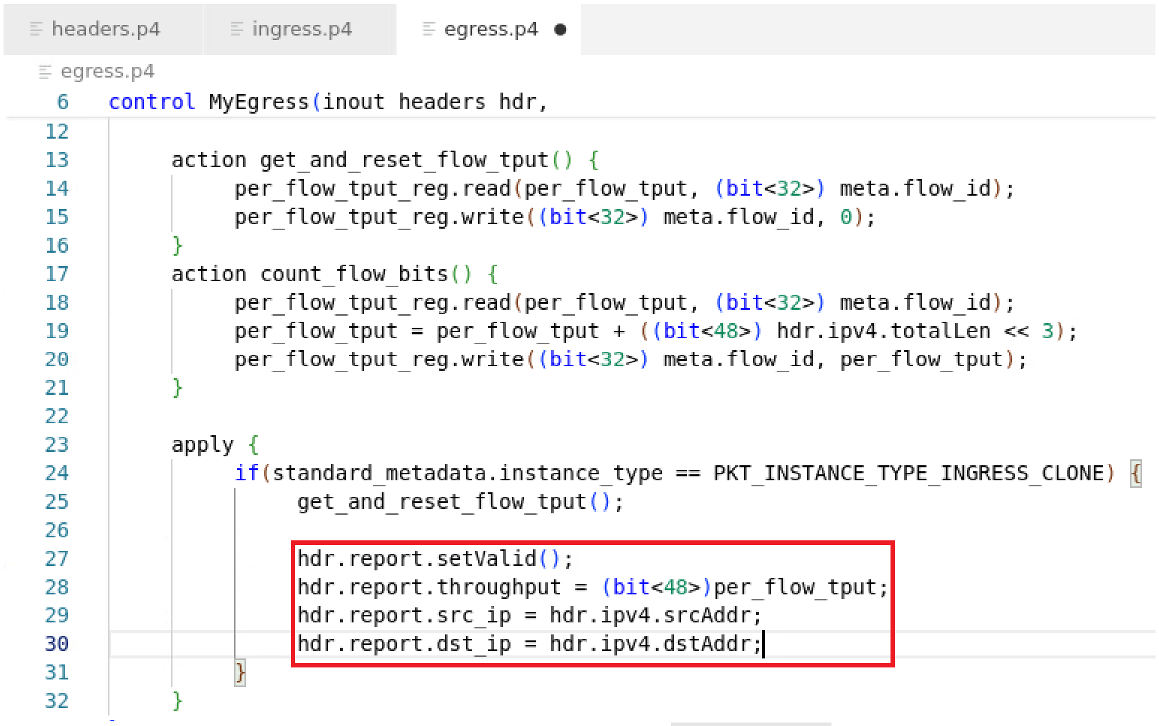
```
control MyEgress(inout headers hdr,

    register<bit<48>>(65536) per_flow_tput_reg;
    bit<48> per_flow_tput;

    action get_and_reset_flow_tput() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, 0);
    }
    action count_flow_bits() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput = per_flow_tput + ((bit<48>) hdr.ipv4.totalLen << 3);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {
            get_and_reset_flow_tput();
        }
    }
}
```

Figure 29. Applying `get_and_reset_flow_tput` on cloned packets.

Recall that the ingress pipeline will clone a packet to the egress pipeline only if a new report should be sent (i.e., 1 second has passed since the last sent report to the collector). Thus, upon receiving a cloned packet, the egress pipeline should retrieve the number of bits of the flow collected over the last second in order to append it to the cloned packet before mirroring it to the collector. After that, the egress pipeline should reset the number of bits stored for the flow.

Step 8. Add the following code to set the header `hdr.report` to valid and to populate the header.

```
hdr.report.setValid();
hdr.report.throughput = (bit<48>)per_flow_tput;
hdr.report.src_ip = hdr.ipv4.srcAddr;
hdr.report.dst_ip = hdr.ipv4.dstAddr;
```



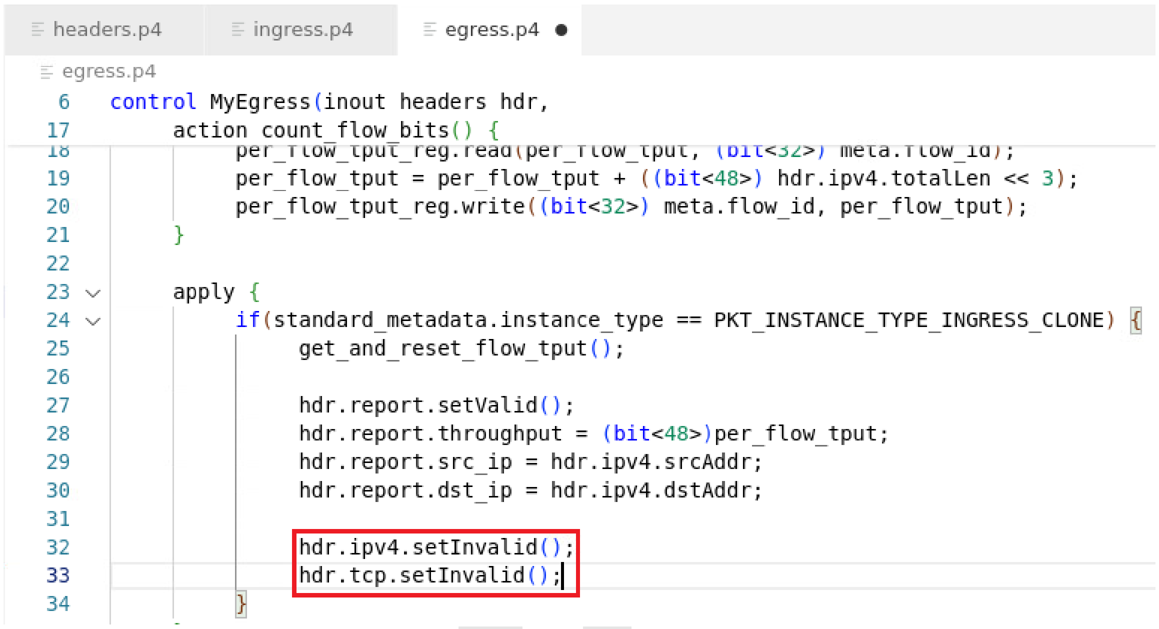
```
control MyEgress(inout headers hdr,

    action get_and_reset_flow_tput() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, 0);
    }
    action count_flow_bits() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput = per_flow_tput + ((bit<48>) hdr.ipv4.totalLen << 3);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;
        }
    }
```

Figure 30. Populating header `hdr.report`.

In the code above, the header `hdr.report` is set to valid so that the header can be assembled with the packet in the deparser. The number of bits (which is stored in `per_flow_tput`) is assigned to the throughput field of the header (i.e., `hdr.report.througput`). The source and destination addresses of the flow are assigned to the `src_ip` and `dst_ip` fields of the report header.

Step 9. Add the following code to discard the `ipv4` and `tcp` headers of the cloned packets.

```
hdr.ipv4.setInvalid();
hdr.tcp.setInvalid();
```



```
control MyEgress(inout headers hdr,
    action count_flow_bits() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput = per_flow_tput + ((bit<48>) hdr.ipv4.totalLen << 3);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;

            hdr.ipv4.setInvalid();
            hdr.tcp.setInvalid();
        }
```

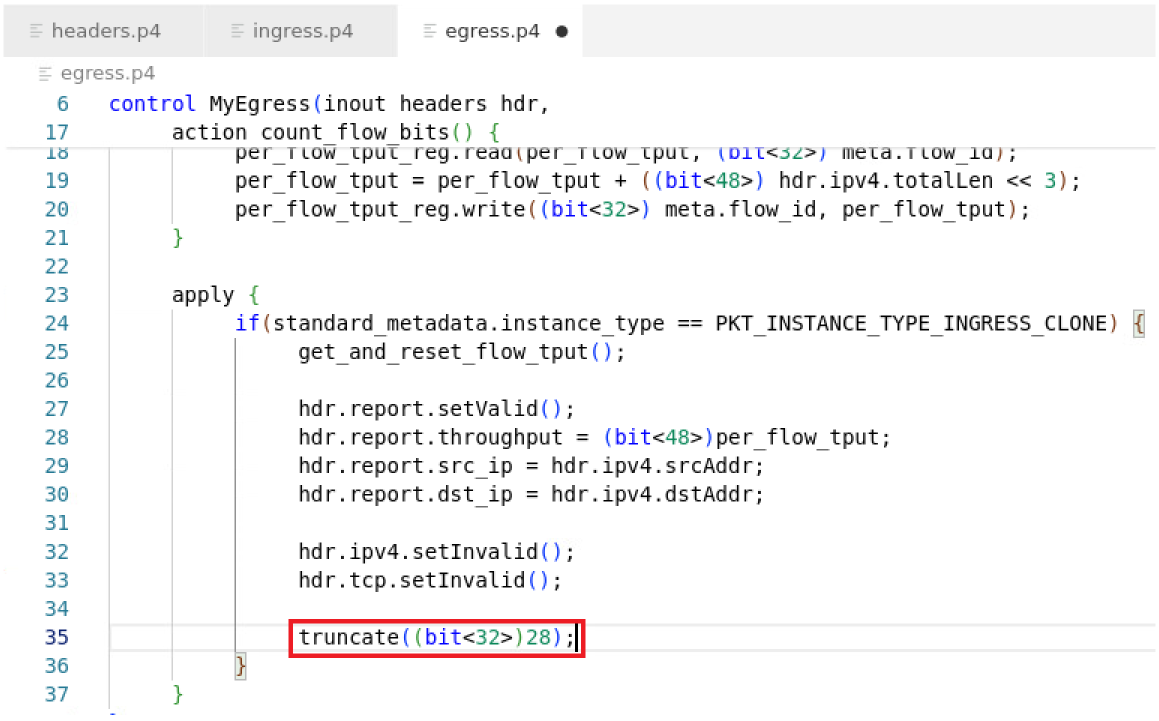
Figure 31. Discarding the `ipv4` and `tcp` headers of the cloned packets.

In the code above, `hdr.ipv4.setInvalid()` action discards the IPv4 header, and consequently, discards all the data stored inside the header. This reduces the size of the cloned packet, reducing the needed storage at the collector. `hdr.tcp.setInvalid()` action discards the TCP header.

Note that the Ethernet header is not discarded because it is used to route the packets to the collector. Because the collector and the switch are at the same network, the cloned packets can be routed using the Ethernet header only. However, if the collector was at a different network than the switch, then the IPv4 should be used to route the packet.

Step 10. Add the following code inside the else statement to discard the cloned packets' payload.

```
truncate((bit<32>28);
```



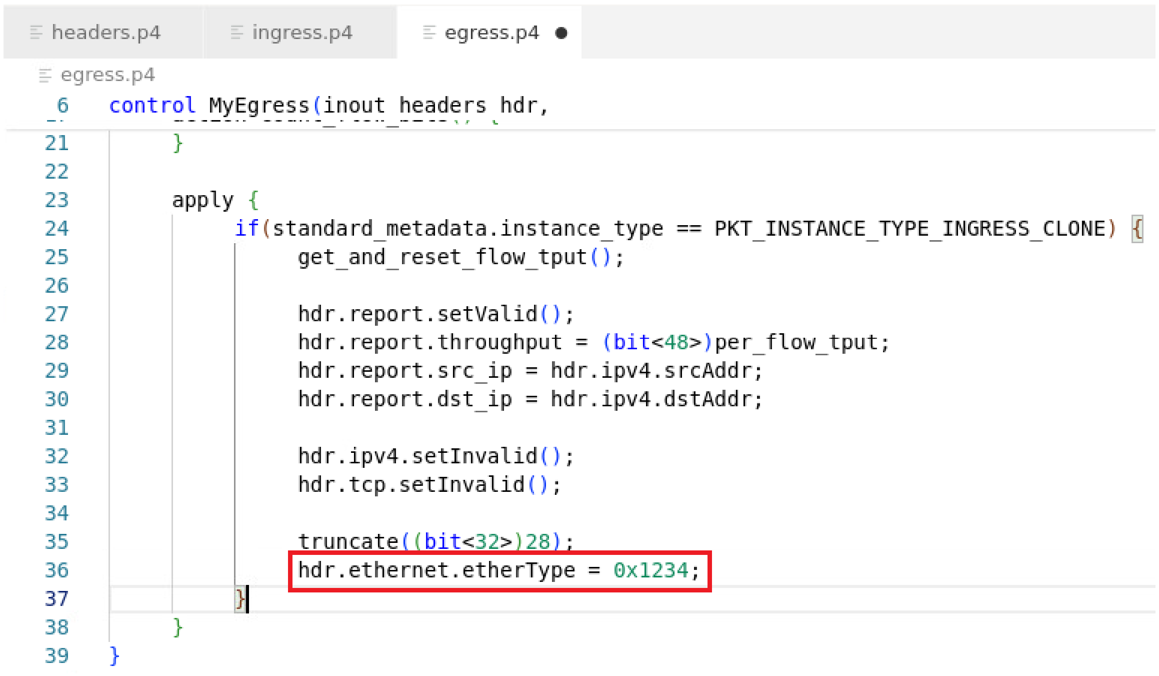
```
headers.p4    ingress.p4    egress.p4 ●
egress.p4
control MyEgress(inout headers hdr,
    action count_flow_bits() {
        per_flow_tput_reg.read(per_flow_tput, (bit<32>) meta.flow_id);
        per_flow_tput = per_flow_tput + ((bit<48>) hdr.ipv4.totalLen << 3);
        per_flow_tput_reg.write((bit<32>) meta.flow_id, per_flow_tput);
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;

            hdr.ipv4.setInvalid();
            hdr.tcp.setInvalid();

            truncate((bit<32>)28);
        }
    }
```

Figure 32. Discarding the packets' payload.

In the code above, `truncate((bit<32>)28)` leaves the first 28 bytes of the packets and drops everything else. The 28 bytes represent the Ethernet and report headers as follows: 6 bytes for the source MAC address, 6 bytes for the destination MAC address, 2 bytes to the Ethernet Type, 6 bytes for the throughput field of the report header, 4 bytes to the src_ip field of the report header, and 4 bytes to the dst_ip of the report header.

Step 11. Add the following code to modify the Ethernet type of the cloned packets.

```
hdr.ethernet.etherType = 0X1234;
```



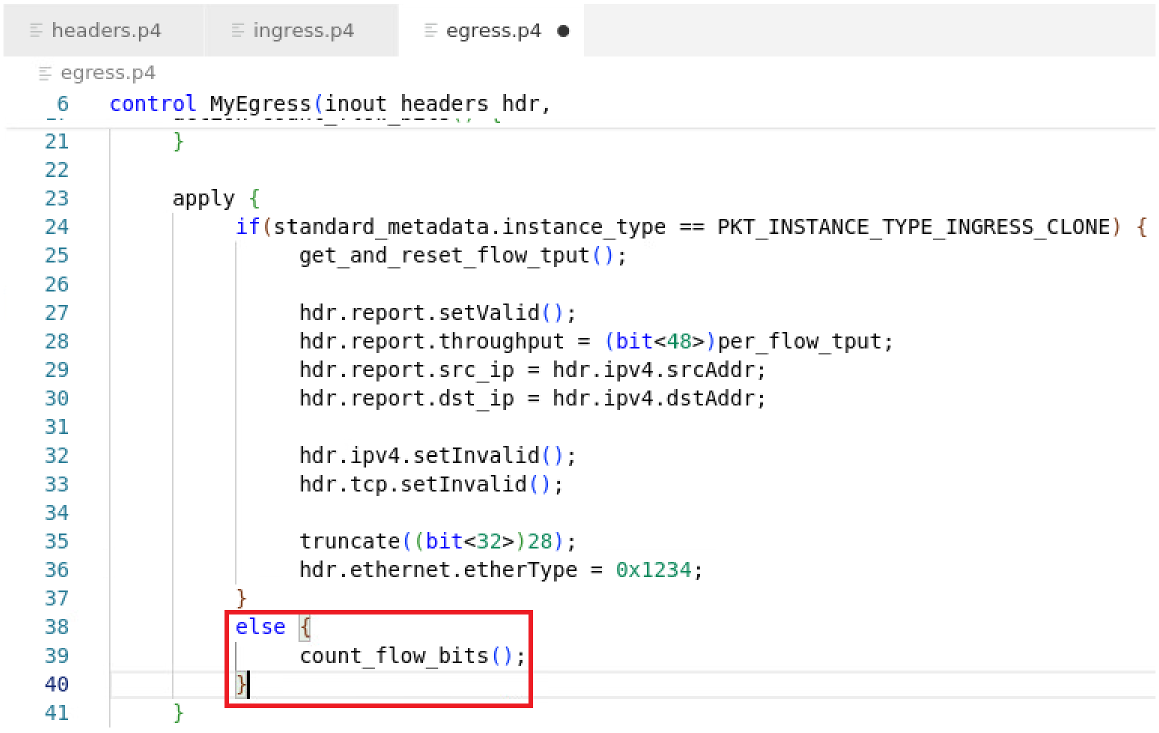
```
headers.p4    ingress.p4    egress.p4 ●
egress.p4
control MyEgress(inout headers hdr,
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;

            hdr.ipv4.setInvalid();
            hdr.tcp.setInvalid();

            truncate((bit<32>)28);
            hdr.ethernet.etherType = 0x1234;
        }
    }
}
```

Figure 33. Modifying the Ethernet type field of the cloned packets.

It is necessary to modify the Ethernet type field (`hdr.ethernet.etherType`) so that the collector can process the cloned packets. The cloned packets have the custom header `report`. Because the `report` header is after the Ethernet header, the Ethernet Type field should be modified to indicate that the next header is `report` and not IPv4 or IPv6. The value of the Ethernet Type field should not be preserved by any protocol (e.g., 0x800 is preserved to IPv4 header and cannot be used).

Step 12. Add the following code to update the number of bits of the flow if the packet is not a cloned instance.

```
else {
      count_flow_bits();
}
```


headers.p4 | ingress.p4 | egress.p4 ●

egress.p4

```
control MyEgress(inout headers hdr,
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE) {
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;

            hdr.ipv4.setInvalid();
            hdr.tcp.setInvalid();

            truncate((bit<32>)28);
            hdr.ethernet.etherType = 0x1234;
        }
        else {
            count_flow_bits();
        }
    }
```

Figure 34. Updating the number of bits of original instances.

Step 13. Save the changes to the file by pressing `Ctrl + s`.

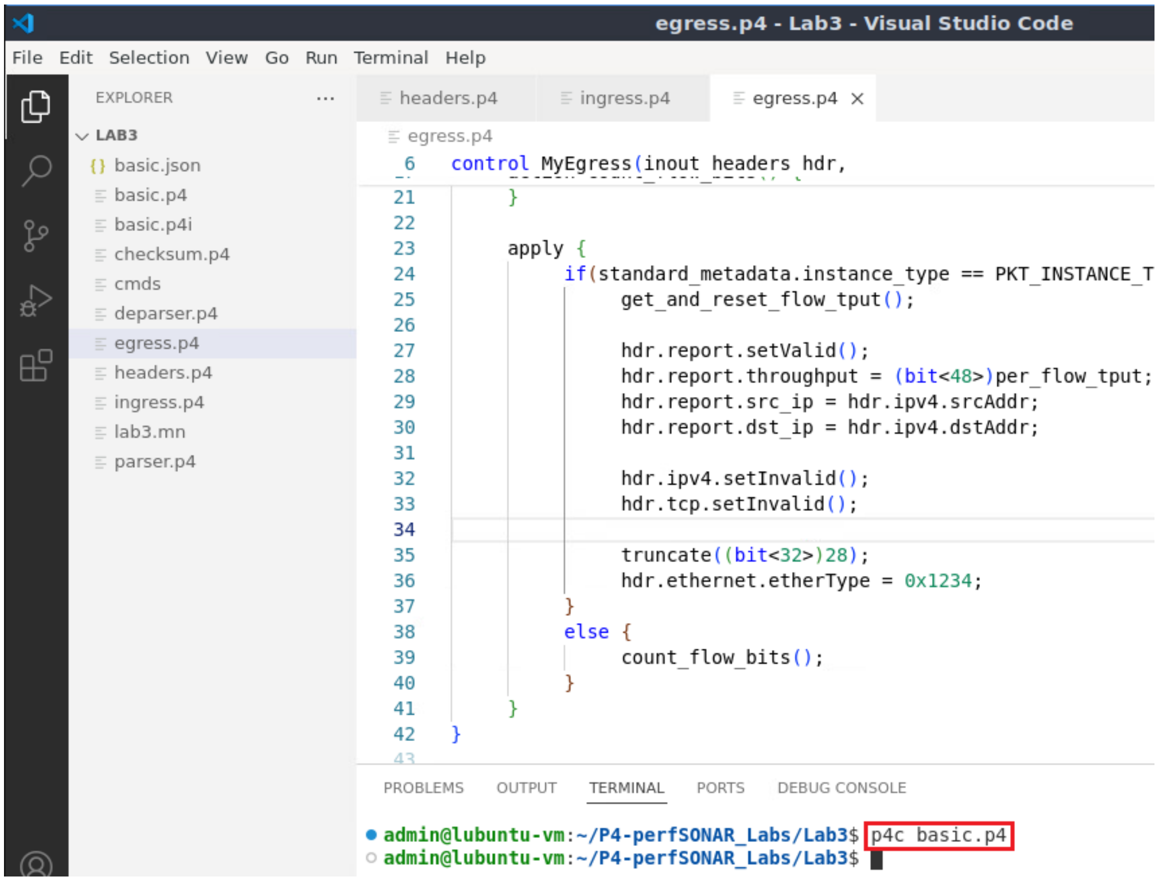
4 Loading the P4 program

In this section, you will compile and load the P4 binary into the switches.

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
```



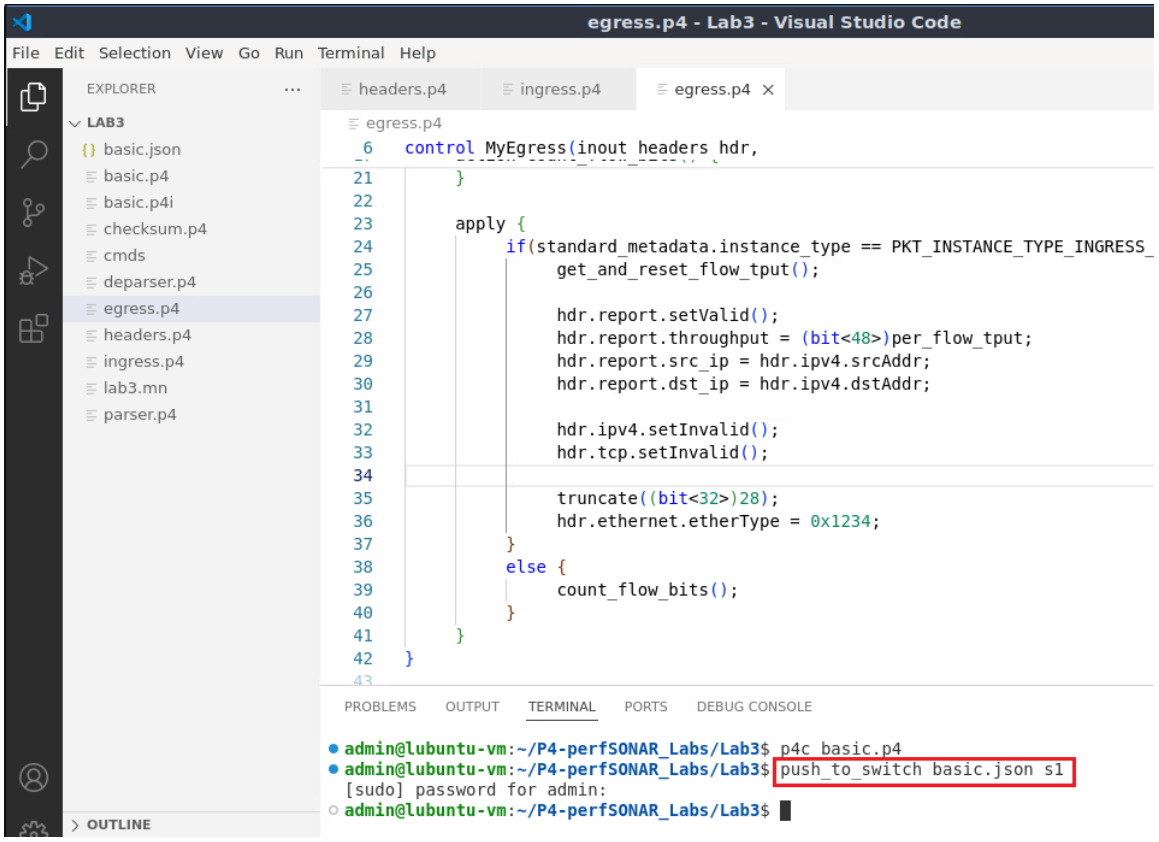
```
control MyEgress(inout headers hdr,
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_T
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;

            hdr.ipv4.setInvalid();
            hdr.tcp.setInvalid();

            truncate((bit<32>)28);
            hdr.ethernet.etherType = 0x1234;
        }
        else {
            count_flow_bits();
        }
    }
}
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab3$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab3$
```

Figure 35. 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
```


egress.p4 - Lab3 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB3
basic.json
basic.p4
basic.p4i
checksum.p4
cmds
deparser.p4
egress.p4
headers.p4
ingress.p4
lab3.mn
parser.p4
OUTLINE
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
control MyEgress(inout headers hdr,
    }

    apply {
        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_
            get_and_reset_flow_tput();

            hdr.report.setValid();
            hdr.report.throughput = (bit<48>)per_flow_tput;
            hdr.report.src_ip = hdr.ipv4.srcAddr;
            hdr.report.dst_ip = hdr.ipv4.dstAddr;

            hdr.ipv4.setInvalid();
            hdr.tcp.setInvalid();

            truncate((bit<32>)28);
            hdr.ethernet.etherType = 0x1234;
        }
        else {
            count_flow_bits();
        }
    }
}
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab3$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab3$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab3$
```

Figure 36. 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.

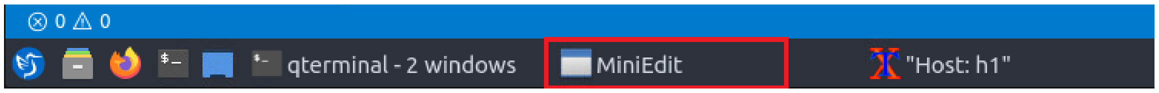


Figure 37. Maximizing the MiniEdit window.

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

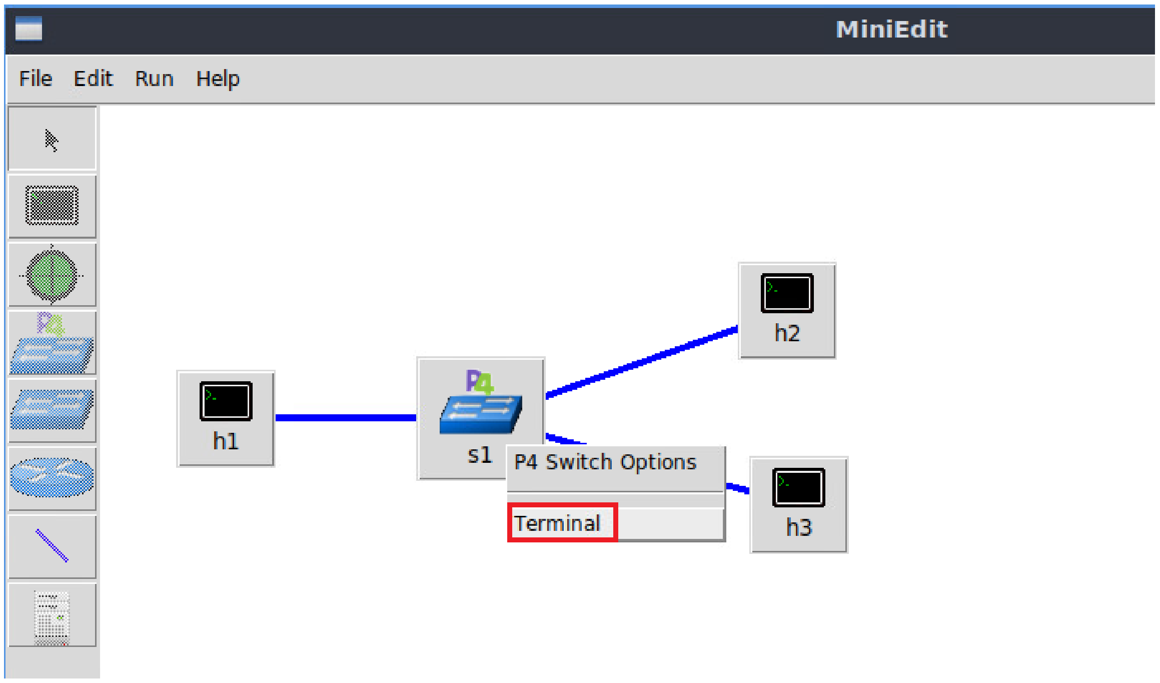
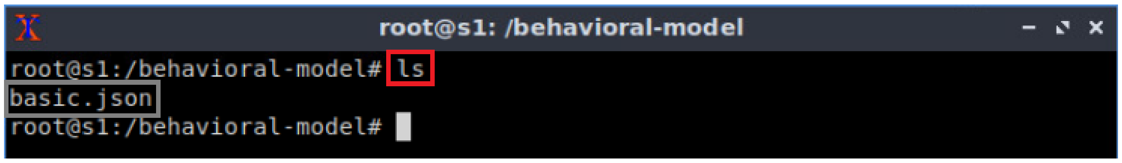


Figure 38. Starting the terminal on the switch.

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

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

```
ls
```



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

Figure 39. 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.

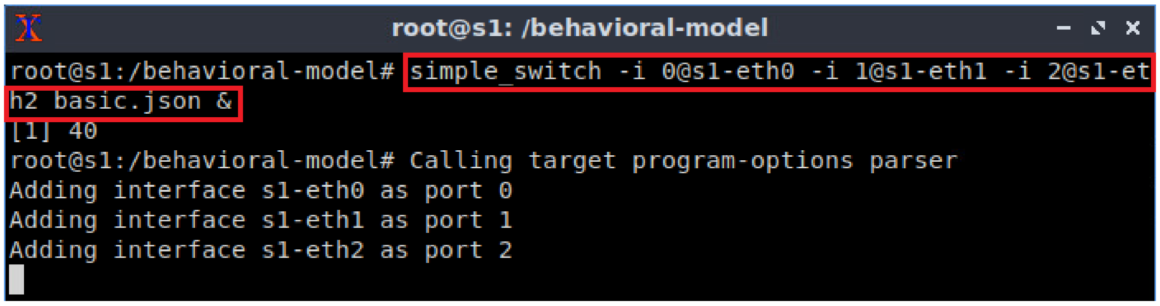
5 Configuring switch s1

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. After that, you will set the packet mirroring.

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 &
```

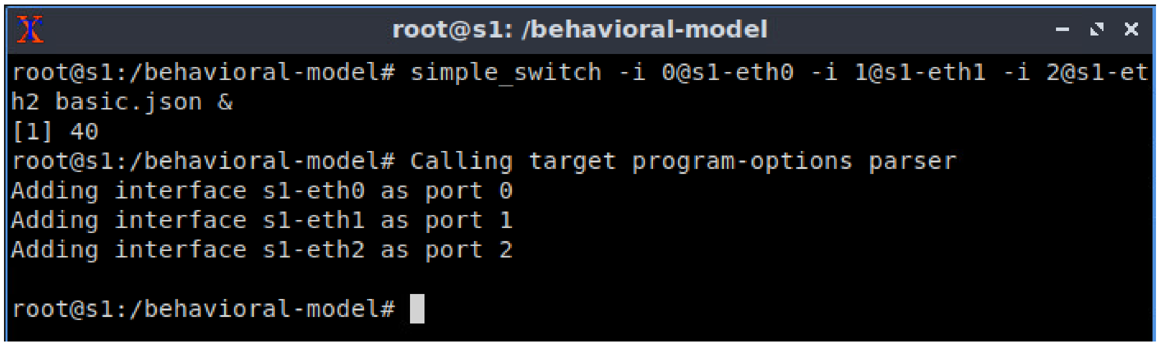


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 40
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 40. 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.

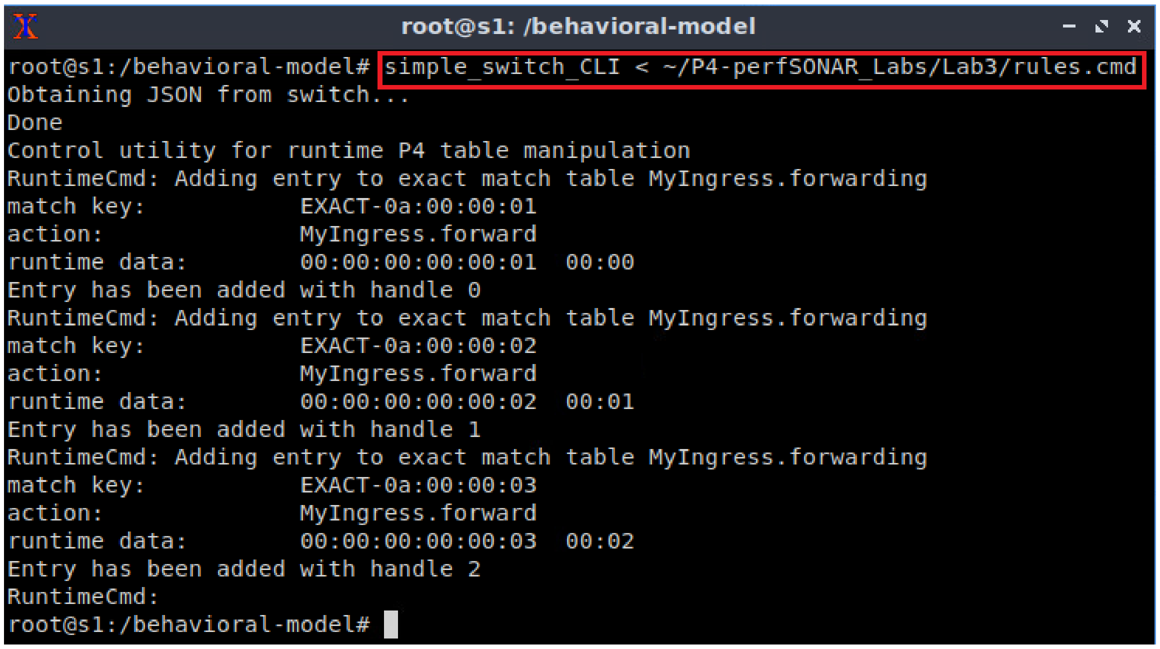


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 40
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model# 
```

Figure 41. Returning to switch s1 CLI.

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

```
simple_switch_CLI < ~/P4-perfSONAR_Labs/Lab3/rules.cmd
```



```
root@s1:/behavioral-model# simple_switch_CLI < ~/P4-perfSONAR_Labs/Lab3/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:02
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:03
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 2
RuntimeCmd:
root@s1:/behavioral-model#
```

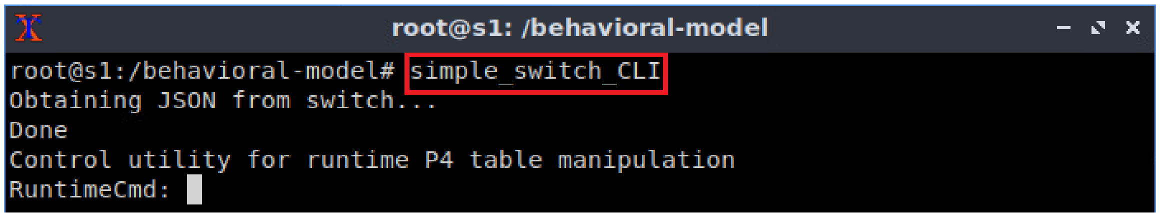
Figure 42. Populating the forwarding table into switch s1.

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

5.3 Defining packet mirroring

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

```
simple_switch_CLI
```

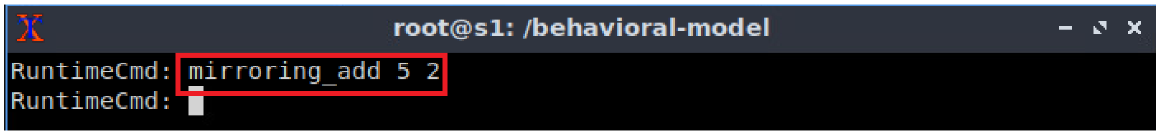


```
root@s1:/behavioral-model# simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd:
```

Figure 43. Starting switch s1's CLI.

Step 2. In switch s1's CLI, type the command below to forward the mirrored packets with session `5` from port `2`.

```
mirroring_add 5 2
```



```
RuntimeCmd: mirroring_add 5 2
RuntimeCmd:
```

Figure 44. Defining packet mirroring.

6 Testing and verifying the P4 program

In this section, the user will use the collector to display the throughput measurements collected from the switch.

6.1 Starting the collector

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

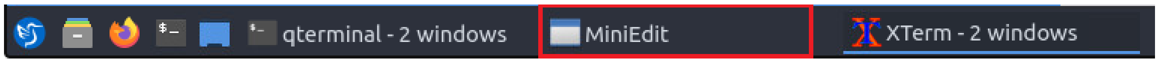


Figure 45. Maximizing the MiniEdit window.

Step 2. Hold the right-click on host h3 and select *Terminal*. This opens the terminal of host h3 and allows the execution of commands on that host.

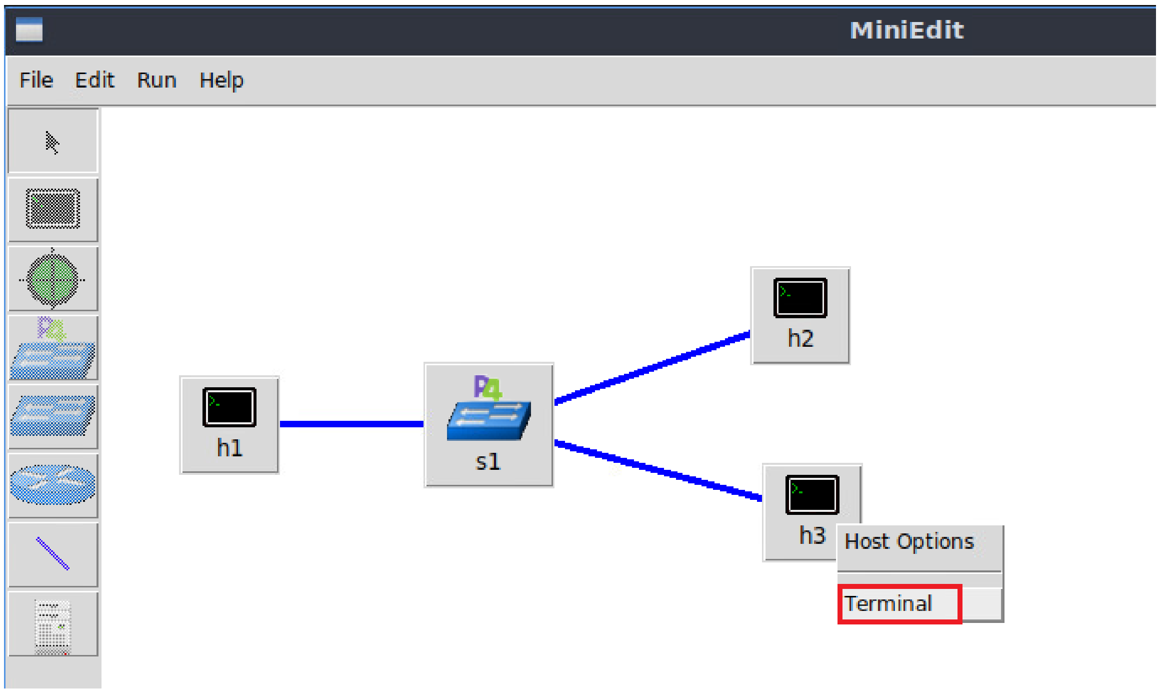
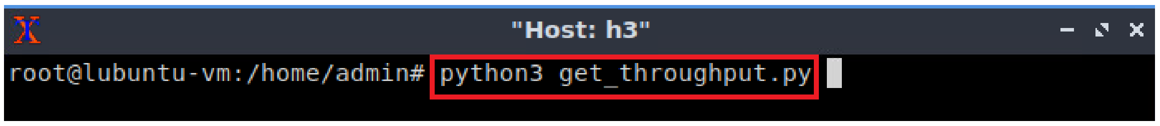


Figure 46. Opening a terminal on host h3.

Step 3. Run the collector by issuing the command bellow.

```
python3 get_throughput.py
```



```
root@lubuntu-vm:/home/admin# python3 get_throughput.py
```

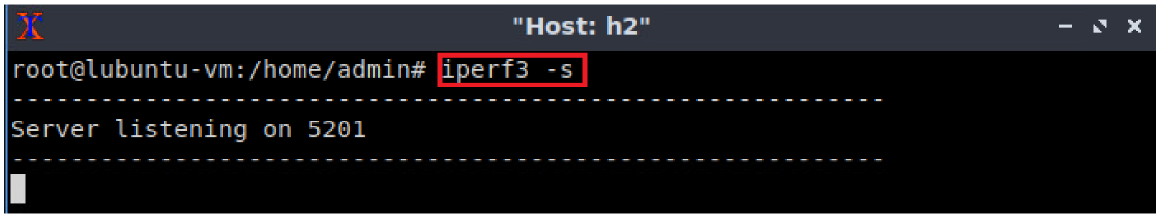
Figure 47. Starting the collector script.

The Python script `get_througput.py` collects, processes, and displays the throughput measurements from the packets mirrored by the switch.

6.2 Testing the P4 program

Step 1. Open a terminal in host h2 and start the Iperf3 server by issuing the following command.

```
iperf3 -s
```

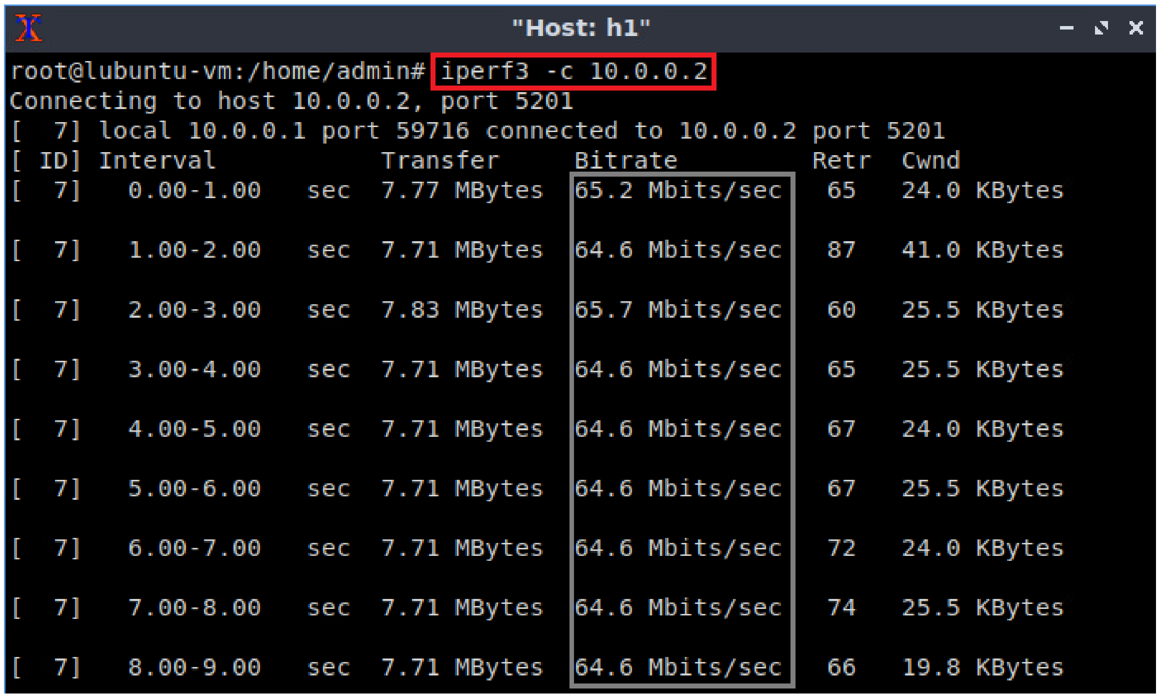


```
"Host: h2"
root@lubuntu-vm:/home/admin# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 48. Starting Iperf3 server on h2.

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

```
iperf3 -c 10.0.0.2
```

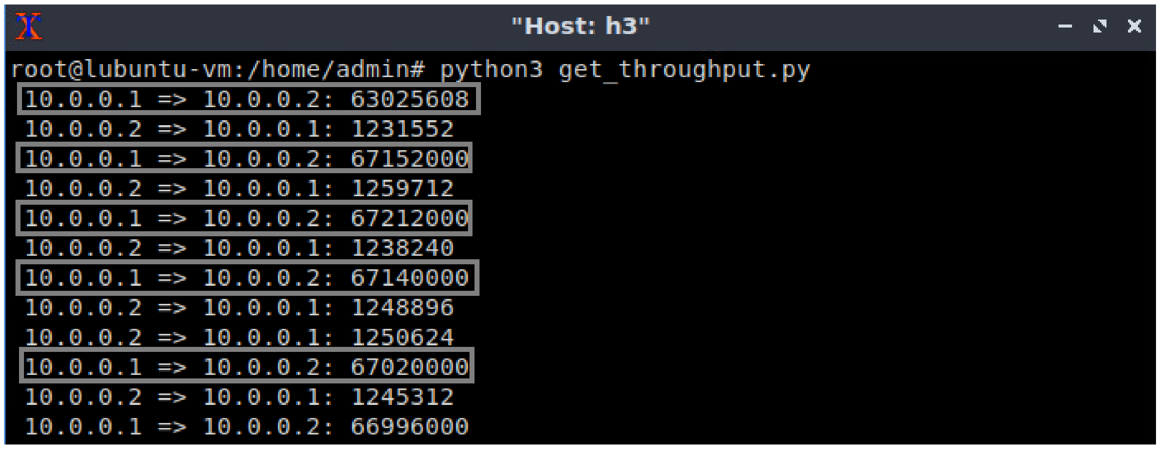


```
"Host: h1"
root@lubuntu-vm:/home/admin# iperf3 -c 10.0.0.2
Connecting to host 10.0.0.2, port 5201
[  7] local 10.0.0.1 port 59716 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  7]   0.00-1.00   sec  7.77 MBytes  65.2 Mbits/sec   65   24.0 KBytes

[  7]   1.00-2.00   sec  7.71 MBytes  64.6 Mbits/sec   87   41.0 KBytes

[  7]   2.00-3.00   sec  7.83 MBytes  65.7 Mbits/sec   60   25.5 KBytes

[  7]   3.00-4.00   sec  7.71 MBytes  64.6 Mbits/sec   65   25.5 KBytes

[  7]   4.00-5.00   sec  7.71 MBytes  64.6 Mbits/sec   67   24.0 KBytes

[  7]   5.00-6.00   sec  7.71 MBytes  64.6 Mbits/sec   67   25.5 KBytes

[  7]   6.00-7.00   sec  7.71 MBytes  64.6 Mbits/sec   72   24.0 KBytes

[  7]   7.00-8.00   sec  7.71 MBytes  64.6 Mbits/sec   74   25.5 KBytes

[  7]   8.00-9.00   sec  7.71 MBytes  64.6 Mbits/sec   66   19.8 KBytes
```

Figure 49. Running an iperf3 test between host h1 and host h2.

The figure above shows that the bitrate is around 65 Mbits/sec.

Step 3. Go back to h3 terminal and inspect the throughput reported by the switch.



```
root@lubuntu-vm:/home/admin# python3 get_throughput.py
 10.0.0.1 => 10.0.0.2: 63025608
 10.0.0.2 => 10.0.0.1: 1231552
 10.0.0.1 => 10.0.0.2: 67152000
 10.0.0.2 => 10.0.0.1: 1259712
 10.0.0.1 => 10.0.0.2: 67212000
 10.0.0.2 => 10.0.0.1: 1238240
 10.0.0.1 => 10.0.0.2: 67140000
 10.0.0.2 => 10.0.0.1: 1248896
 10.0.0.2 => 10.0.0.1: 1250624
 10.0.0.1 => 10.0.0.2: 67020000
 10.0.0.2 => 10.0.0.1: 1245312
 10.0.0.1 => 10.0.0.2: 66996000
```

Figure 50. Monitoring the throughput reported by the switch.

In the figure above, the throughput reported by the switch is around 67 Mbits/sec. Note that iperf3 reports the goodput[5] (i.e., the payload of the packets without considering the size of the header). However, the switch reports the throughput. For this, there is slight difference between the measurement reported by iperf3 and the measurement reported by the switch.

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

References

1. Qalcafe. "What is a network microburst and how can you detect them?" [Online]. Available: https://tinyurl.com/3yyudn2k
2. The P4 Language Consortium . " The P4 Language Specification." [Online]. Available: https://tinyurl.com/4j37n4mj
3. WIKIPEDIA. "Port mirroring." [Online]. Available: https://tinyurl.com/bdujd6ty
4. P4lang/p4-guide github repository. *"p4-guide/v1model-special-ops*." [Online]. Available: https://tinyurl.com/mrkw3wfu
5. "TCP/IP with iperf3."[Online]. Available: https://tinyurl.com/bdcr3xfd

UNIVERSITY OF
SOUTH CAROLINA

P4-PERFSONAR LAB SERIES

Lab 4: Monitoring the RTT of TCP Flows with P4

Document Version: 07-03-2024

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 RTT calculation process ... 4
1.2 Lab scenario ... 4
2 Lab topology ... 5
2.1 Verifying connectivity between host h1 and host h2 ... 7
3 Measuring the RTT of TCP flows in the data plane ... 8
3.1 Loading the programming environment ... 8
3.2 Inspecting the ingress pipeline ... 9
3.2.1 Understanding the forwarding logic ... 9
3.2.2 Understanding the components of the RTT calculation logic ... 11
3.2.3 Understanding how the forwarding and RTT calculation logic are applied . 15
4 Compiling and loading the P4 program in the software switch ... 18
4.1 Compiling the P4 program ... 18
4.2 Loading the P4 program ... 18
4.3 Verifying the configuration ... 19
5 Configuring switch s1 ... 20
5.1 Mapping the P4 program's ports ... 20
5.2 Loading the rules to the switch ... 21
6 Testing and evaluating the P4 program ... 22
References ... 24

Overview

In this lab, you will learn how Round-Trip Time (RTT) is measured using a P4-programmable data plane (PDP). The lab details the RTT calculation process, which is performed using a hash table implemented on a P4 switch. You will generate traffic between a sender and a receiver and observe the RTT measurements collected by a monitoring system.

Objectives

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

1. Understand how packets are paired to generate an RTT sample.
2. Implement the forwarding and RTT calculation logic in a PDP.
3. Evaluate the system by generating traffic between two end hosts.

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: Measuring the RTT of TCP flows in the data plane.
4. Section 4: Compiling and loading the P4 program in the software switch.
5. Section 5: Testing and evaluating the P4 program.

1 Introduction

The Round-Trip Time (RTT) is an essential metric to evaluate network performance. Increasing or fluctuating RTT values can degrade user Quality of Experience (QoE) and signal potential network performance or security issues like congestion or routing alterations. While end hosts often have access to RTT statistics, Internet Service Providers

(ISPs) lack direct insights into the latency their customers experience. Even in data centers, continuously tracking RTTs across all hosts is computing intensive. Therefore, continuous RTT monitoring can enhance an ISP's ability to evaluate network security and performance, covering aspects like BGP routing security, IP spoofing detection, Service-Level Agreement (SLA) compliance, and QoE. Traditionally, network operators measure RTT using active tools such as perfSONAR[2], typically in response to client-reported service quality issues. On the other hand, passive measurement tools generally provide RTT samples based only on the initial three-way TCP handshake. These tools fail to capture latency variations during prolonged TCP connections, such as video streaming.

With the emergence of P4-programmable data planes (PDPs), it is possible to calculate the RTT of TCP flows in a scalable way. This approach has various benefits such as detecting abnormal behaviors, identifying congested links, and enabling the development of applications that mitigate RTT-related issues at line rate[3,4]. Additionally, this approach can complement active measurement tools such as perfSONAR to provide a more accurate visibility of network events[5].

1.1 RTT calculation process

Figure 1 shows the process of calculating RTT on a per-flow basis in the data plane. This technique links the TCP sequence number (SEQ) and acknowledgment (ACK) numbers in incoming and outgoing packets. The system determines the RTT by measuring the time difference between these packets. This calculation occurs in the data plane using flow identification (FID) and the expected acknowledgment (eACK) of outgoing packets as the key for timestamp (Tstamp) values. The eACK is obtained by adding the packet length to the SEQ. When an incoming packet matches the expected acknowledgment, an RTT sample is generated by computing the difference between the timestamps.

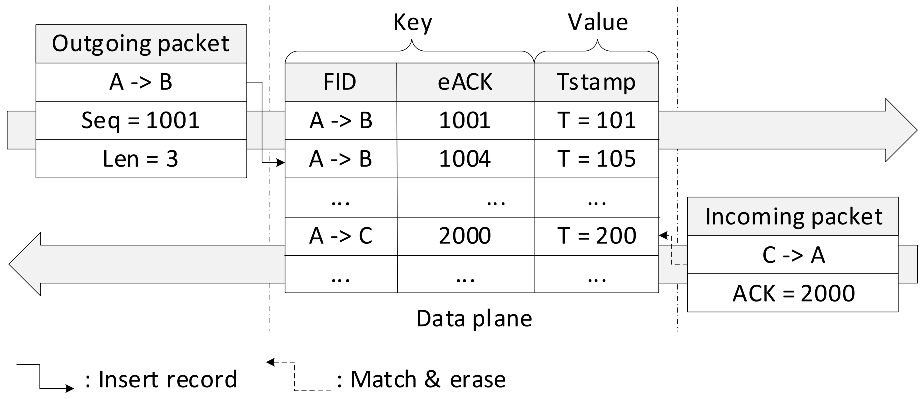


Figure 1. RTT calculation. When a TCP packet is received, its timestamp is compared with the timestamp of the corresponding outgoing packet. This calculation uses a flow identifier (FID) and the expected acknowledgment (eACK) as the key to retrieve the stored timestamp (Tstamp) value[1].

1.2 Lab scenario

In this lab, you will initiate a data transfer over TCP between two hosts. The P4 switch will run a P4 program designed to calculate the RTT and send the RTT samples to a collector. The collector will then display the RTT values calculated by the P4 switch.

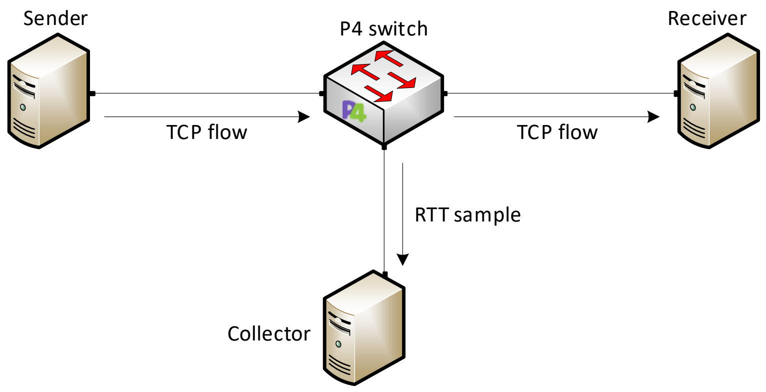


Figure 2. Lab scenario.

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.

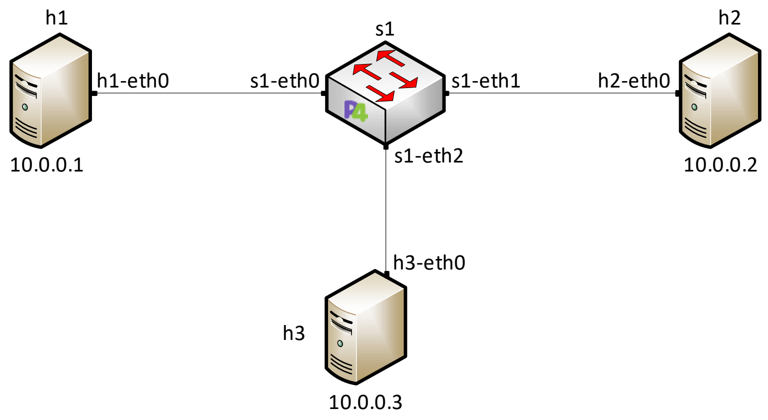


Figure 3. Lab topology.

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`.

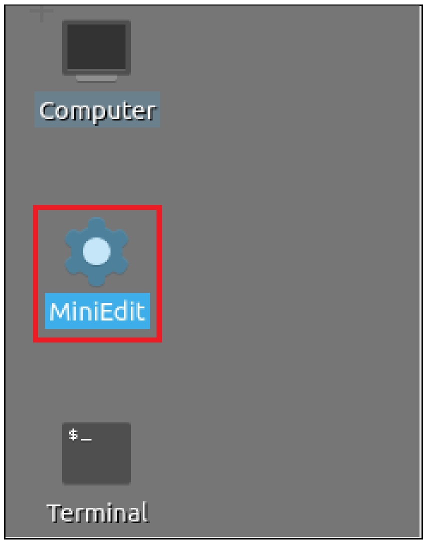


Figure 4. MiniEdit shortcut.

Step 2. 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.

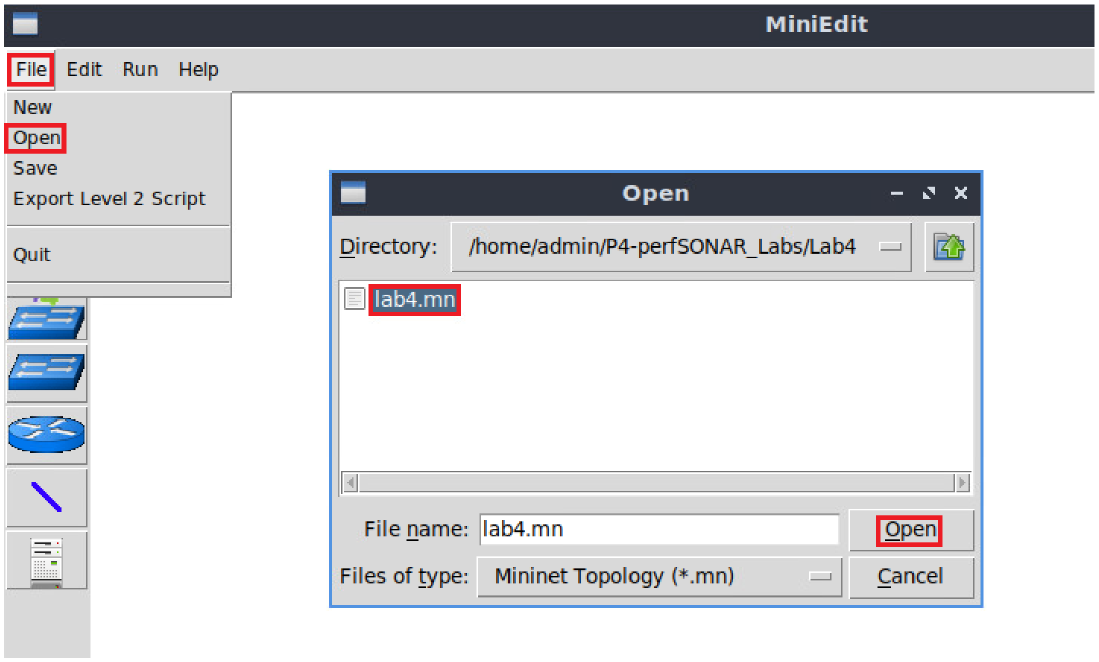


Figure 5. MiniEdit's *Open* dialog.

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.

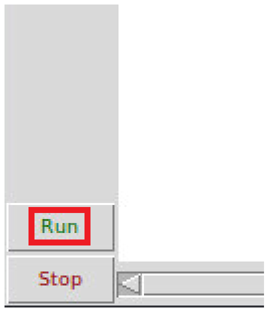


Figure 6. Running the emulation.

2.1 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.

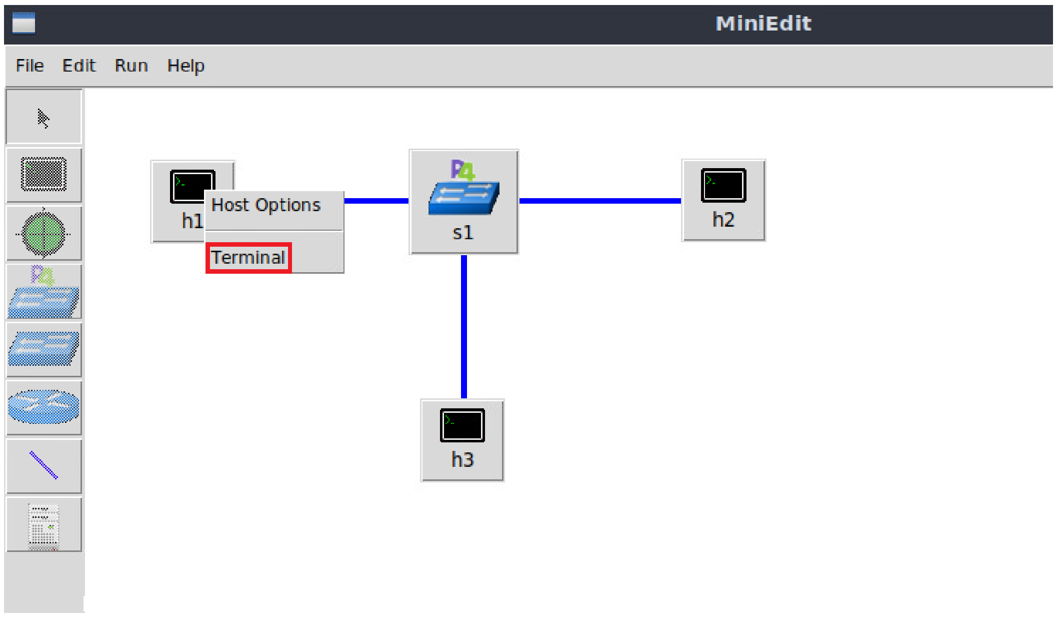
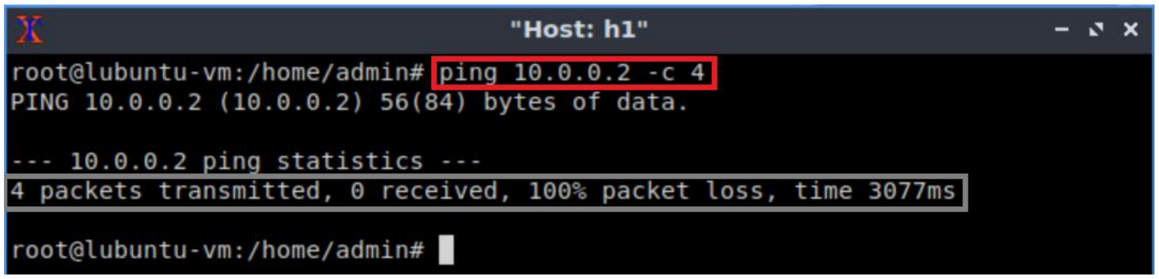


Figure 7. Opening a terminal on host h1.

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.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3077ms

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. Note that there will be no connectivity between any two hosts in the topology before loading a P4 program to the switch.

3 Measuring the RTT of TCP flows in the data plane

In this section, you will examine the ingress pipeline of the P4 program used to compute the RTT of TCP flows. The ingress pipeline implements the forwarding and the RTT calculation logics.

3.1 Loading the programming environment

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

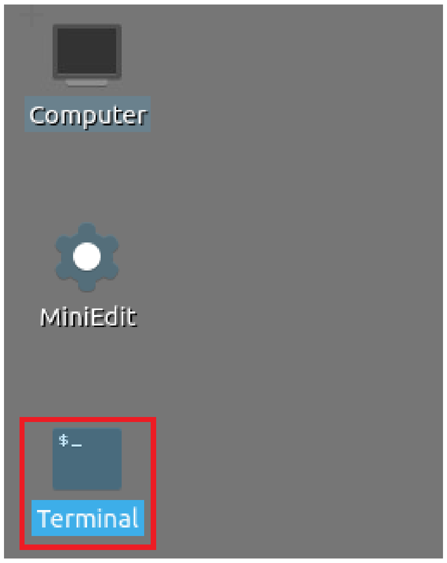
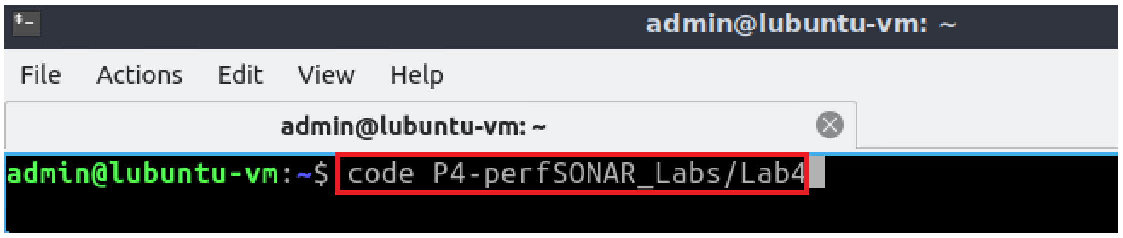


Figure 9. Shortcut to open a Linux terminal.

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-perfSOANR_Labs/Lab4
```



```
admin@lubuntu-vm:~$ code P4-perfSONAR_Labs/Lab4
```

Figure 10. Loading the development environment.

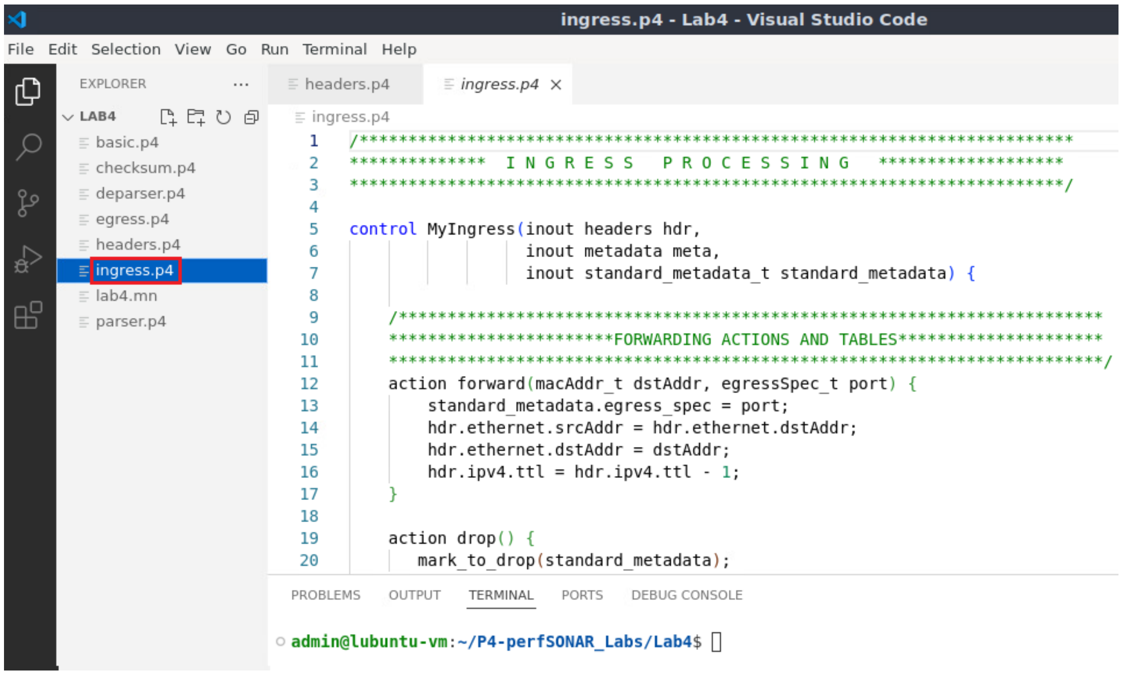
3.2 Inspecting the ingress pipeline

The ingress pipeline implements the forwarding and the RTT calculation logic. It uses a hash table to index the timestamps to calculate the RTT values as explained in section 1.

3.2.1 Understanding the forwarding logic

The actions and table needed to implement the forwarding logic are already provided in this lab. The goal of this section is to understand its behavior.

Step 1. Click on the *ingress.p4* file in the file explorer on the left-hand side.

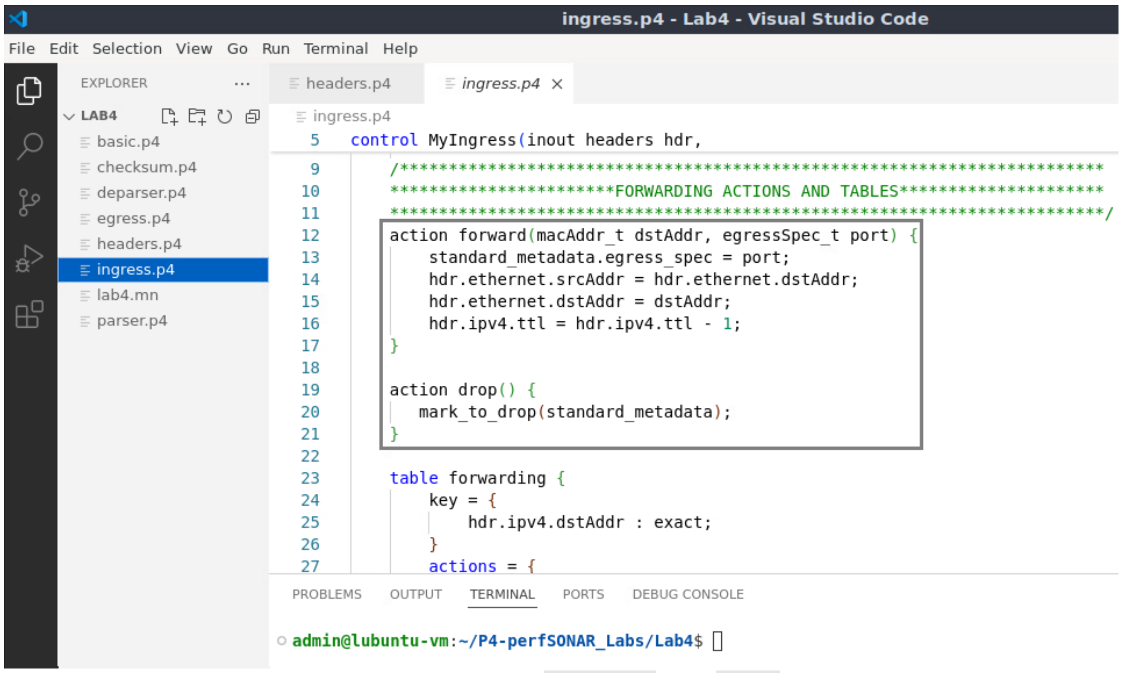


```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

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

    /*************************************************************************
    *********************FORWARDING ACTIONS AND TABLES********************
    *************************************************************************/
    action forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

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

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$ 
```

Figure 11. Navigating into the *ingress.p4* file.

Step 2. Inspect the `forward` and `drop` actions.


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
headers.p4
ingress.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
control MyIngress(inout headers hdr,
    /*************************************************************************
    **********************FORWARDING ACTIONS AND TABLES*********************
    *************************************************************************/
    action forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    action drop() {
       mark_to_drop(standard_metadata);
    }

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

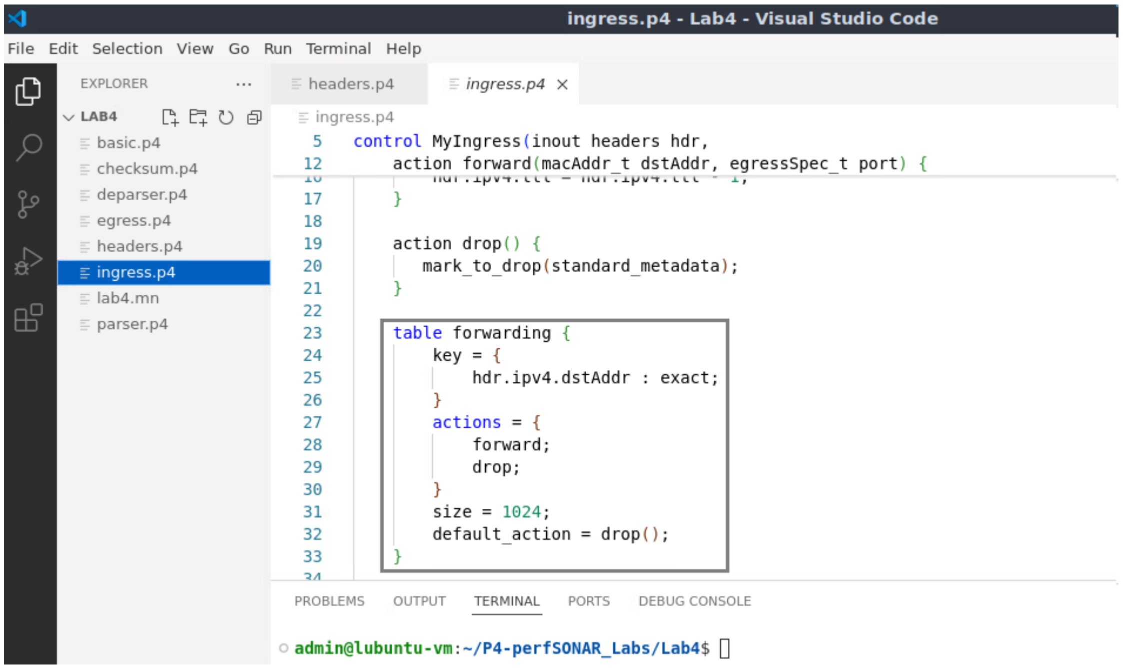
```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 12. Inspecting the `forward` and `drop` actions.

These actions are explained as follows:

- `forward()`: this action receives as parameters the destination MAC address (i.e., `macAddr_t dstAddr`) and the egress port (i.e., `egressSpec_t port`). The values of these parameters are provided by the control plane. The behavior of the action specifies the egress port and performs the swapping between the source and destination of the MAC addresses. Finally, this action decreases the TTL value in the IPv4 header.
- `drop()`: this action uses the `mark_to_drop()` primitive to indicate through the standard metadata that the current packet must be dropped.

Step 3. Inspect the `forwarding` table.



```
control MyIngress(inout headers hdr,
    action forward(macAddr_t dstAddr, egressSpec_t port) {
    }

    action drop() {
        mark_to_drop(standard_metadata);
    }

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

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 13. Inspecting the forwarding table.

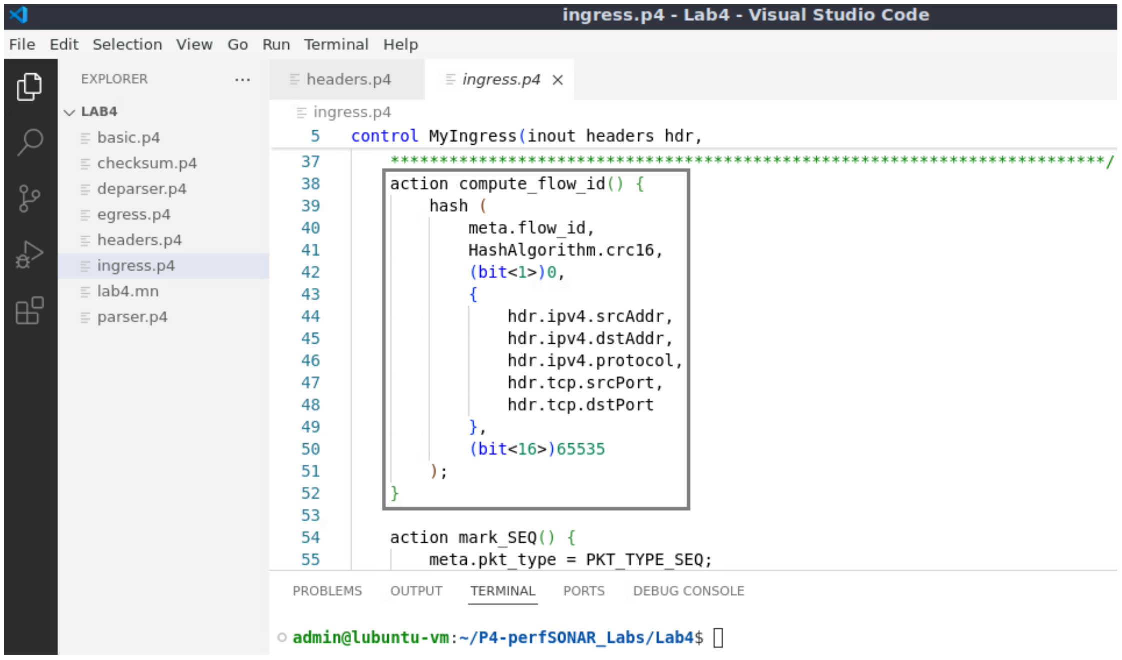
This match-action table has the following components:

- `key`: which in this example is the destination IPv4 address. The match type is exact.
- `actions`: the list of actions that this table implements.
- `size`: the maximum number of entries that the table will allocate.
- `default_action`: the action that will be executed when there is a miss in the table i.e., when the IPv4 destination address is not present in the table.

3.2.2 Understanding the components of the RTT calculation logic

The RTT calculation logic consist of calculating the difference between the time stamps of two types of TCP segments: 1) the SEQ and 2) the corresponding ACK. This is performed by implementing a hash table where the keys are the flow ID (i.e., source and destination IPv4 addresses and port numbers, and the protocol specified in the IPv4 header) and the expected ACK (i.e., eACK). This logic is explained with more details in Section 1.

Step 1. Consider lines 38-52 to see how the flow ID is computed.


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
headers.p4
ingress.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

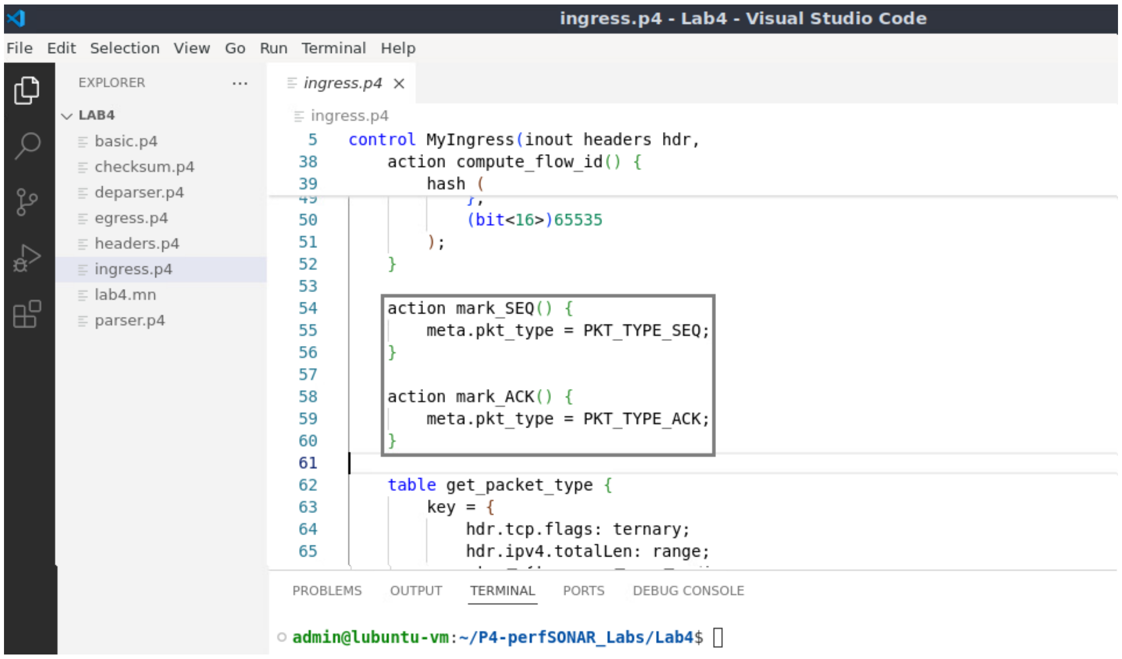
```
control MyIngress(inout headers hdr,
    *************************************************************************/
    action compute_flow_id() {
        hash (
            meta.flow_id,
            HashAlgorithm.crc16,
            (bit<1>)0,
            {
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.ipv4.protocol,
                hdr.tcp.srcPort,
                hdr.tcp.dstPort
            },
            (bit<16>)65535
        );
    }

    action mark_SEQ() {
        meta.pkt_type = PKT_TYPE_SEQ;
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 14. Inspecting flow ID computation action.

This action computes a unique identifier (`meta.flow_id`) for the flow using a CRC16 hash of the source and destination IP addresses, the protocol, and the source and destination TCP ports.

Step 2. Consider the actions `mark_SEQ()` and `mark_ACK()`. These actions mark packets as either SEQ (sequence) or ACK (acknowledgment) based on their type.

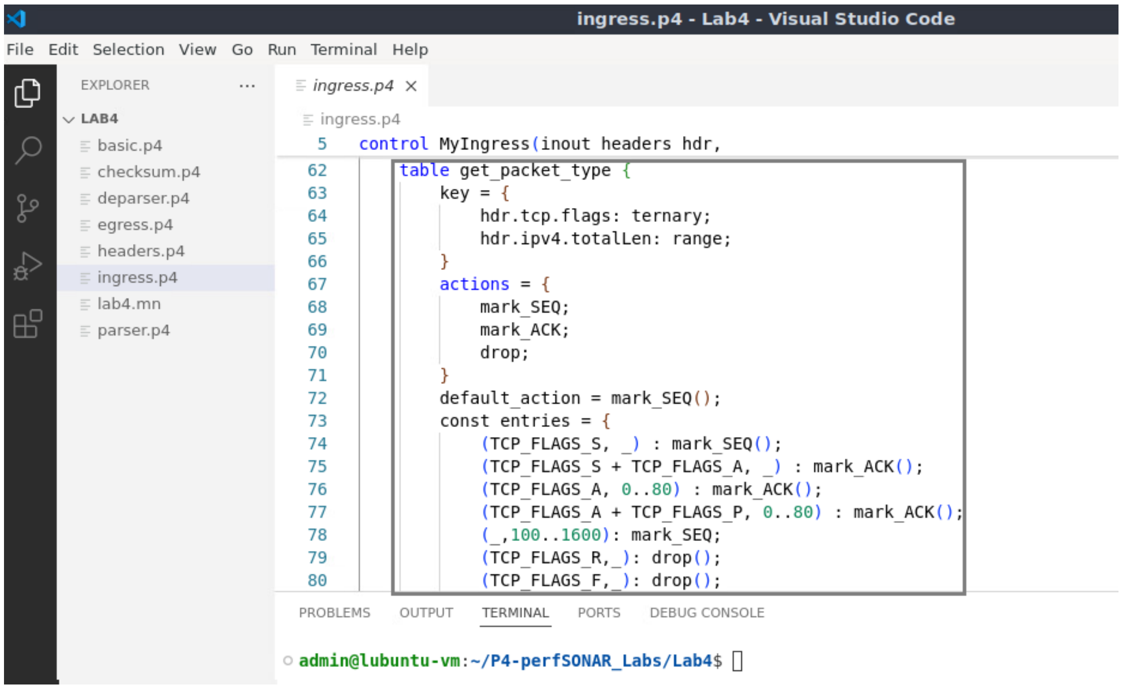

ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
ingress.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
        hash (
            (bit<16>)65535
        );
    }

    action mark_SEQ() {
        meta.pkt_type = PKT_TYPE_SEQ;
    }

    action mark_ACK() {
        meta.pkt_type = PKT_TYPE_ACK;
    }

    table get_packet_type {
        key = {
            hdr.tcp.flags: ternary;
            hdr.ipv4.totalLen: range;
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 15. Inspecting the packet type marking actions.

Step 3. Scroll down and consider lines 62-80 and inspect the table `get_packet_type`.


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

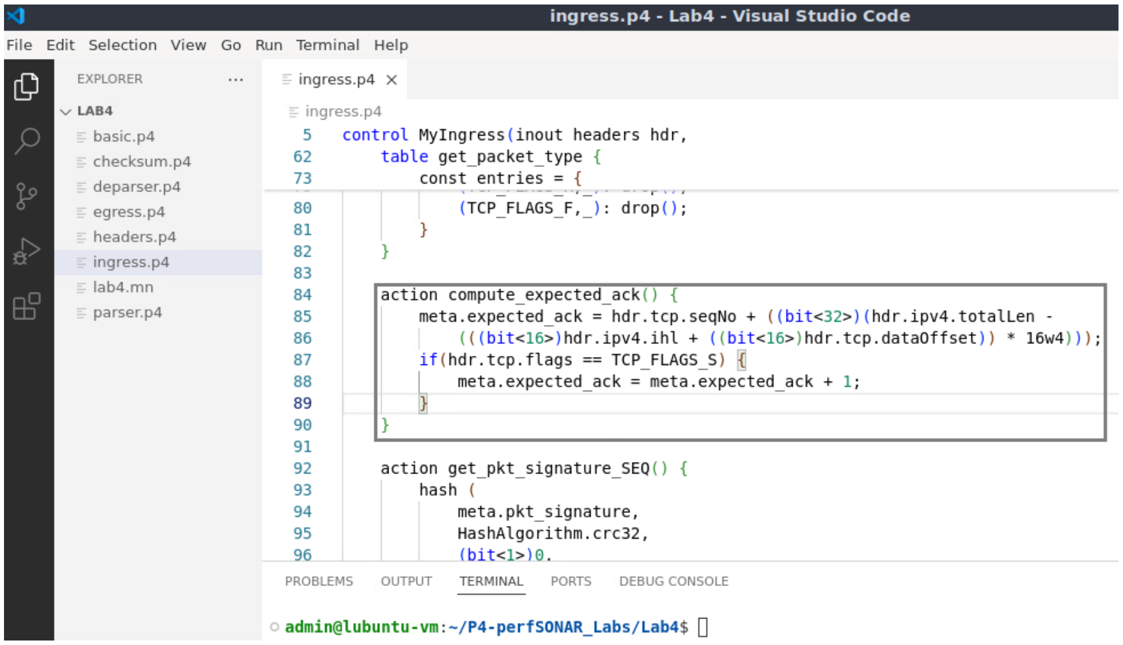
```
control MyIngress(inout headers hdr,
table get_packet_type {
    key = {
        hdr.tcp.flags: ternary;
        hdr.ipv4.totalLen: range;
    }
    actions = {
        mark_SEQ;
        mark_ACK;
        drop;
    }
    default_action = mark_SEQ();
    const entries = {
        (TCP_FLAGS_S, _) : mark_SEQ();
        (TCP_FLAGS_S + TCP_FLAGS_A, _) : mark_ACK();
        (TCP_FLAGS_A, 0..80) : mark_ACK();
        (TCP_FLAGS_A + TCP_FLAGS_P, 0..80) : mark_ACK();
        (_,100..1600): mark_SEQ;
        (TCP_FLAGS_R,_): drop();
        (TCP_FLAGS_F,_): drop();
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 16. Inspecting the packet type match-action table.

This table classifies packets as SEQ or ACK based on their TCP flags and total length. SEQ packets are marked with `mark_SEQ()` and ACK packets with `mark_ACK()`. Certain TCP flags like RST and FIN result in the packet being dropped.

Step 4. Inspect lines to understand how the expected ACK is computed.


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

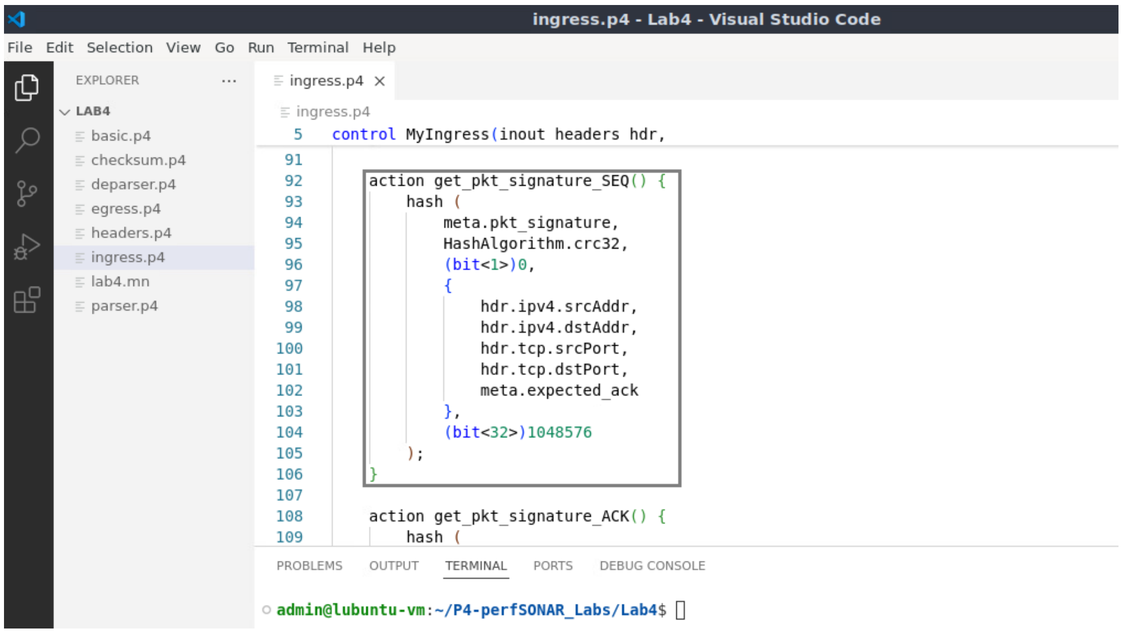
```
control MyIngress(inout headers hdr,
    table get_packet_type {
        const entries = {
            (TCP_FLAGS_F,_): drop();
        }
    }

    action compute_expected_ack() {
        meta.expected_ack = hdr.tcp.seqNo + ((bit<32>)(hdr.ipv4.totalLen -
            (((bit<16>)hdr.ipv4.ihl + ((bit<16>)hdr.tcp.dataOffset)) * 16w4)));
        if(hdr.tcp.flags == TCP_FLAGS_S) {
            meta.expected_ack = meta.expected_ack + 1;
        }
    }

    action get_pkt_signature_SEQ() {
        hash (
            meta.pkt_signature,
            HashAlgorithm.crc32,
            (bit<1>)0,
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 17. Inspecting the expected ACK computation action.

This action calculates the expected ACK number (`meta.expected_ack`) for a SEQ packet. It accounts for the total length of the IPv4 packet and adjusts based on the header lengths of IPv4 and TCP. If the packet is a SYN, the expected ACK is incremented by 1.

Step 5. Inspect lines 92-106 to understand how the SEQ signature is calculated.


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

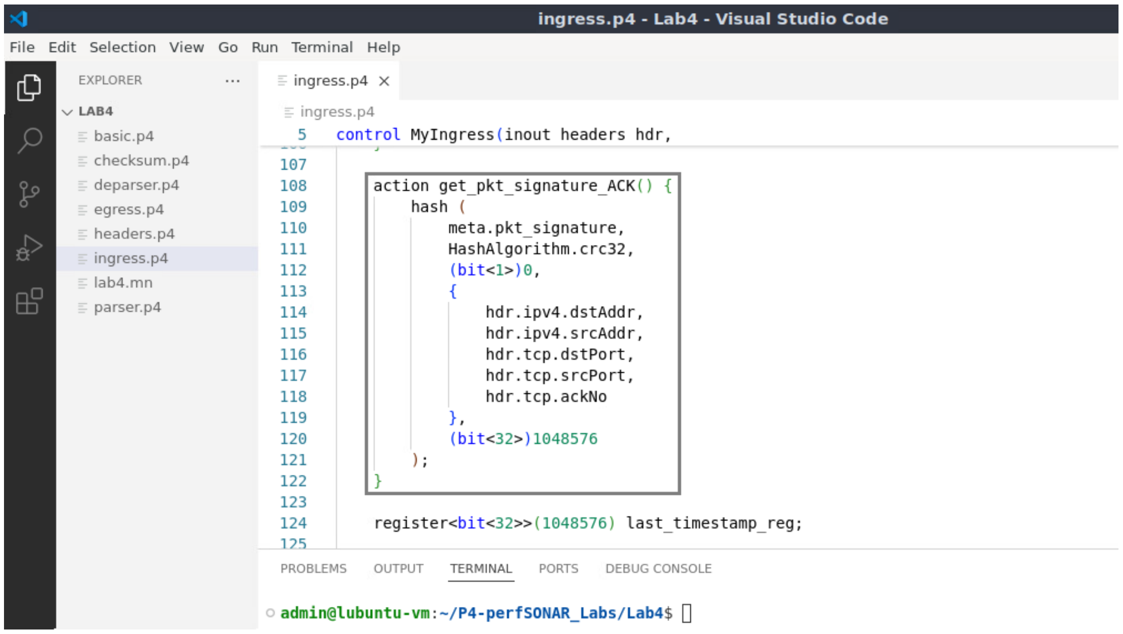
```
ingress.p4
5   control MyIngress(inout headers hdr,
91
92      action get_pkt_signature_SEQ() {
93          hash (
94              meta.pkt_signature,
95              HashAlgorithm.crc32,
96              (bit<1>)0,
97              {
98                  hdr.ipv4.srcAddr,
99                  hdr.ipv4.dstAddr,
100                 hdr.tcp.srcPort,
101                 hdr.tcp.dstPort,
102                 meta.expected_ack
103             },
104             (bit<32>)1048576
105         );
106     }
107
108     action get_pkt_signature_ACK() {
109         hash (
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 18. Inspecting the packet signature computation action (SEQ).

These actions compute a unique signature for SEQ and ACK packets using a CRC32 hash. The signature for SEQ packets includes the source and destination IP addresses and ports, along with the expected ACK number. For ACK packets, the signature includes the same fields but swaps the source and destination addresses and ports and uses the actual ACK number.

Step 6. Similarly, inspect lines 108-122 to understand how the ACK signature is calculated.

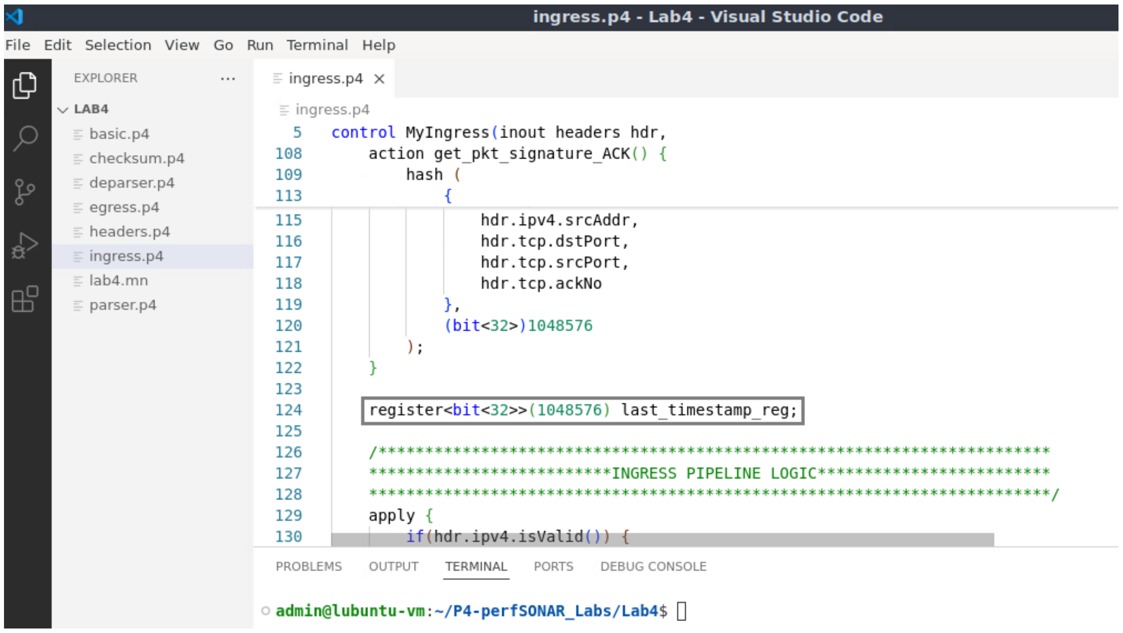

ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
ingress.p4
5   control MyIngress(inout headers hdr,
107
108     action get_pkt_signature_ACK() {
109         hash (
110             meta.pkt_signature,
111             HashAlgorithm.crc32,
112             (bit<1>)0,
113             {
114                 hdr.ipv4.dstAddr,
115                 hdr.ipv4.srcAddr,
116                 hdr.tcp.dstPort,
117                 hdr.tcp.srcPort,
118                 hdr.tcp.ackNo
119             },
120             (bit<32>)1048576
121         );
122     }
123
124     register<bit<32>>(1048576) last_timestamp_reg;
125
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 19. Inspecting the get packet signature action (ACK).

Step 7. Consider the following register. This register stores timestamps, indexed by the packet signatures.


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.p4
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
control MyIngress(inout headers hdr,
    action get_pkt_signature_ACK() {
        hash (
            {
                hdr.ipv4.srcAddr,
                hdr.tcp.dstPort,
                hdr.tcp.srcPort,
                hdr.tcp.ackNo
            },
            (bit<32>)1048576
        );
    }

    register<bit<32>>(1048576) last_timestamp_reg;

    /*************************************************************************
    *************************INGRESS PIPELINE LOGIC*************************
    *************************************************************************/
    apply {
        if(hdr.ipv4.isValid()) {
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

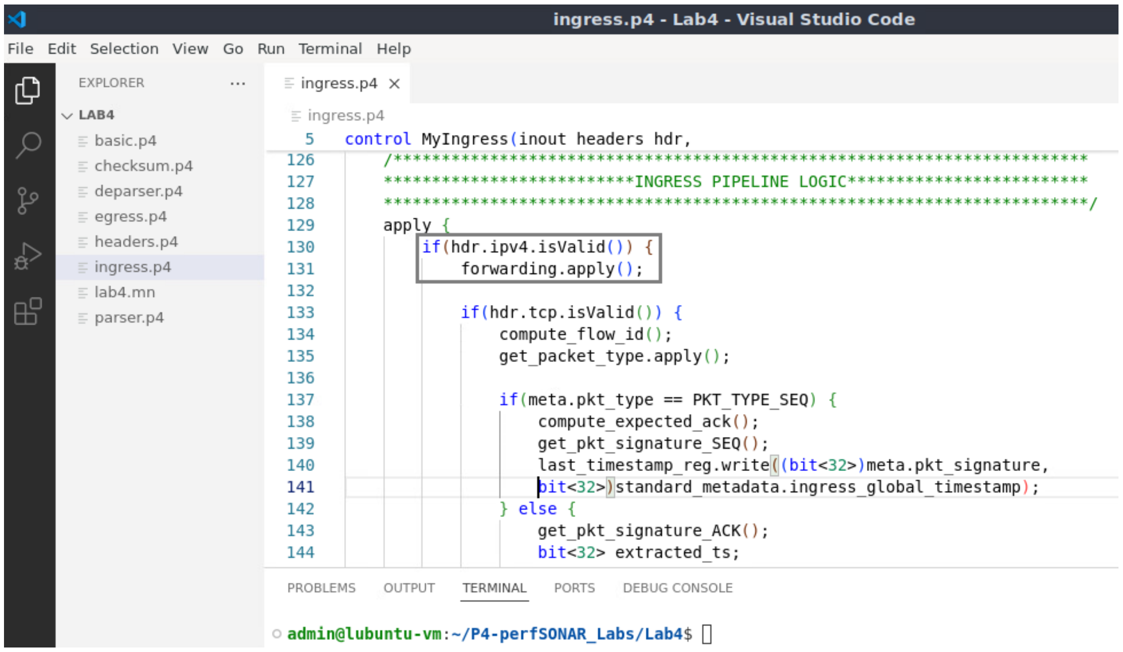
Figure 20. Inspecting the register that stores the last timestamp.

3.2.3 Understanding how the forwarding and RTT calculation logic are applied

In this section you will understand how the forwarding and RTT calculation logics are applied. The forwarding logic consists of applying the `forwarding` table. This logic is implemented in the `apply` block in the ingress pipeline. The RTT calculation logic applies the following steps:

1. When a SEQ packet arrives, the flow ID and expected ACK are computed. A signature is then created using these values.
2. The timestamp of the SEQ packet is stored in the register using the packet signature as the index.
3. When an ACK packet corresponding to a previously seen SEQ packet arrives, its signature is computed similarly.
4. The current time is retrieved, and the stored timestamp for the corresponding SEQ packet is fetched from the register.
5. The RTT is calculated by subtracting the stored timestamp from the current time.

Step 1. Inspect the ingress pipeline logic by scrolling down until the `apply` block. Line 130 contains the statement that verifies if the IPv4 header is valid, and the following line applies the forwarding logic.

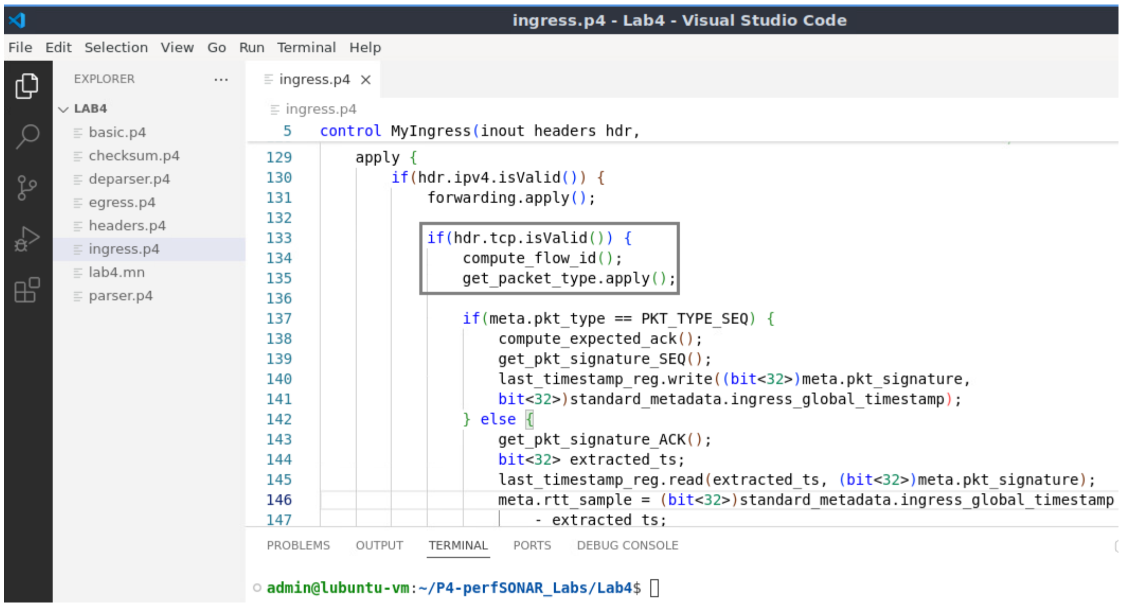


```
control MyIngress(inout headers hdr,
    /*************************************************************************
    ************************INGRESS PIPELINE LOGIC************************
    *************************************************************************/
    apply {
        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()) {
                compute_flow_id();
                get_packet_type.apply();

                if(meta.pkt_type == PKT_TYPE_SEQ) {
                    compute_expected_ack();
                    get_pkt_signature_SEQ();
                    last_timestamp_reg.write((bit<32>)meta.pkt_signature,
                    bit<32>)standard_metadata.ingress_global_timestamp);
                } else {
                    get_pkt_signature_ACK();
                    bit<32> extracted_ts;
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 21. Inspecting how the forwarding logic is applied.

Step 2. Consider lines 133-135 that applies TCP header validation, flow ID computation, and packet type determination.

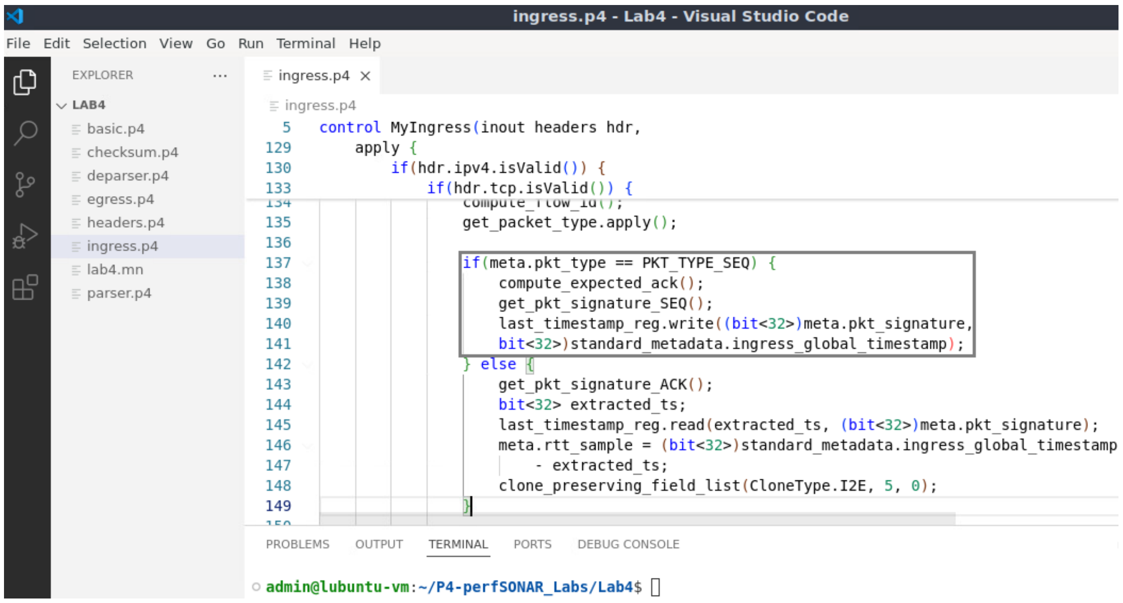


```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()) {
                compute_flow_id();
                get_packet_type.apply();

                if(meta.pkt_type == PKT_TYPE_SEQ) {
                    compute_expected_ack();
                    get_pkt_signature_SEQ();
                    last_timestamp_reg.write((bit<32>)meta.pkt_signature,
                    bit<32>)standard_metadata.ingress_global_timestamp);
                } else {
                    get_pkt_signature_ACK();
                    bit<32> extracted_ts;
                    last_timestamp_reg.read(extracted_ts, (bit<32>)meta.pkt_signature);
                    meta.rtt_sample = (bit<32>)standard_metadata.ingress_global_timestamp
                        - extracted_ts;
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 22. Inspecting how the flow ID and packet types are obtained.

Step 3. The code fragment below applies the SEQ packet handling.



```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            if(hdr.tcp.isValid()) {
                compute_flow_id();
                get_packet_type.apply();

                if(meta.pkt_type == PKT_TYPE_SEQ) {
                    compute_expected_ack();
                    get_pkt_signature_SEQ();
                    last_timestamp_reg.write((bit<32>)meta.pkt_signature,
                    bit<32>)standard_metadata.ingress_global_timestamp);
                } else {
                    get_pkt_signature_ACK();
                    bit<32> extracted_ts;
                    last_timestamp_reg.read(extracted_ts, (bit<32>)meta.pkt_signature);
                    meta.rtt_sample = (bit<32>)standard_metadata.ingress_global_timestamp
                        - extracted_ts;
                    clone_preserving_field_list(CloneType.I2E, 5, 0);
                }
```

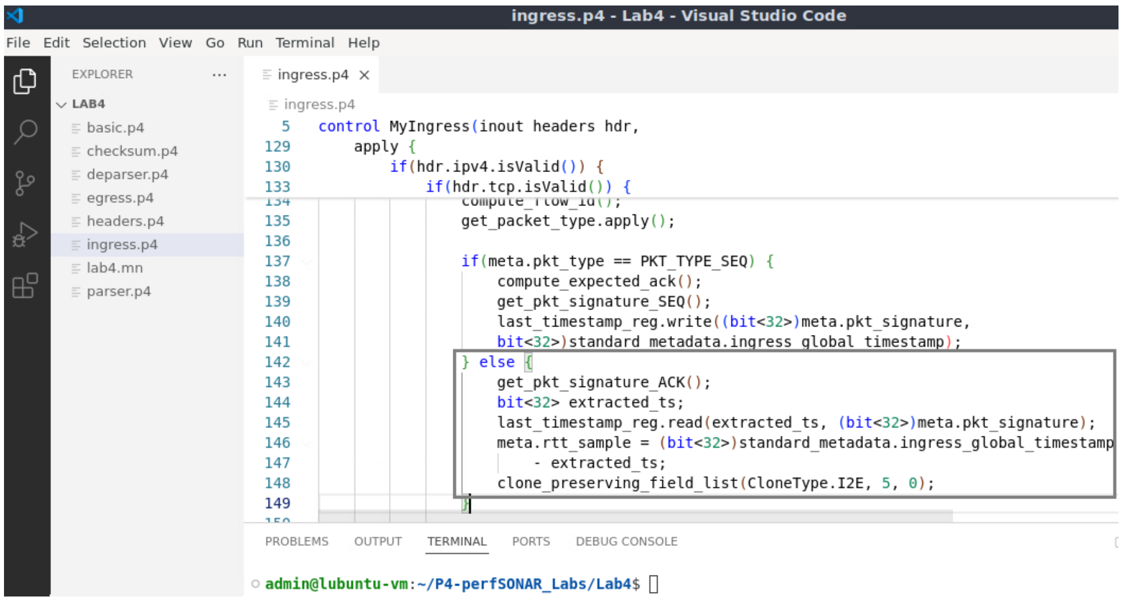
```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 23. Inspecting how the sequence number is collected.

If the packet is a SEQ packet:

- `compute_expected_ack()` calculates the expected ACK number.
- `get_pkt_signature_SEQ()` computes a unique signature for the SEQ packet.
- The current timestamp is stored in the `last_timestamp_reg` register, indexed by the packet signature.

Step 4. The code fragment below applies the ACK packet handling.



```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            if(hdr.tcp.isValid()) {
                compute_flow_id();
                get_packet_type.apply();

                if(meta.pkt_type == PKT_TYPE_SEQ) {
                    compute_expected_ack();
                    get_pkt_signature_SEQ();
                    last_timestamp_reg.write((bit<32>)meta.pkt_signature,
                    bit<32>)standard_metadata.ingress_global_timestamp);
                } else {
                    get_pkt_signature_ACK();
                    bit<32> extracted_ts;
                    last_timestamp_reg.read(extracted_ts, (bit<32>)meta.pkt_signature);
                    meta.rtt_sample = (bit<32>)standard_metadata.ingress_global_timestamp
                        - extracted_ts;
                    clone_preserving_field_list(CloneType.I2E, 5, 0);
                }
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

Figure 24. Inspecting how the expected ACK is computed.

If the packet is an ACK packet:

- `get_pkt_signature_ACK()` computes a unique signature for the ACK packet.

- The timestamp of the corresponding SEQ packet is read from the `last_timestamp_reg` register using the computed signature.
- The RTT sample (`meta.rtt_sample`) is calculated as the difference between the current timestamp and the extracted timestamp.
- The packet is cloned and sent to the control plane for further analysis or action using `clone_preserving_field_list(CloneType.I2E, 5, 0)`, where `5` is the session ID that determines how the packet is cloned and `0` is the field list ID that specifies which packet fields to preserve in the clone.

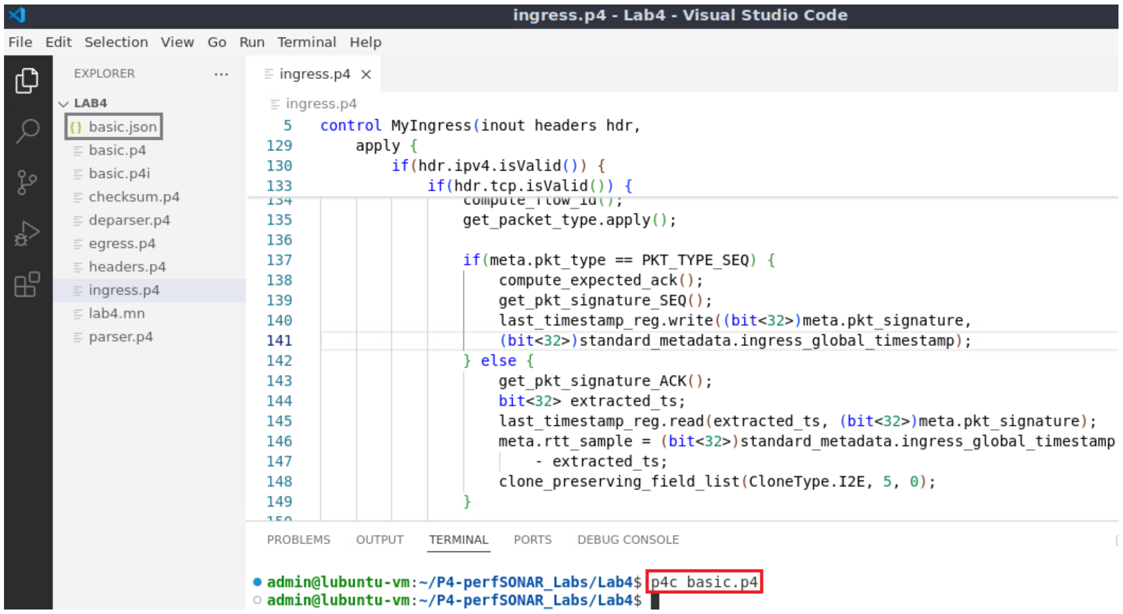
4 Compiling and loading the P4 program in the software switch

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

4.1 Compiling the P4 program

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

```
p4c basic.p4
```


ingress.p4 - Lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
ingress.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            if(hdr.tcp.isValid()) {
                compute_flow_id();
                get_packet_type.apply();

                if(meta.pkt_type == PKT_TYPE_SEQ) {
                    compute_expected_ack();
                    get_pkt_signature_SEQ();
                    last_timestamp_reg.write((bit<32>)meta.pkt_signature,
                    (bit<32>)standard_metadata.ingress_global_timestamp);
                } else {
                    get_pkt_signature_ACK();
                    bit<32> extracted_ts;
                    last_timestamp_reg.read(extracted_ts, (bit<32>)meta.pkt_signature);
                    meta.rtt_sample = (bit<32>)standard_metadata.ingress_global_timestamp
                        - extracted_ts;
                    clone_preserving_field_list(CloneType.I2E, 5, 0);
                }
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

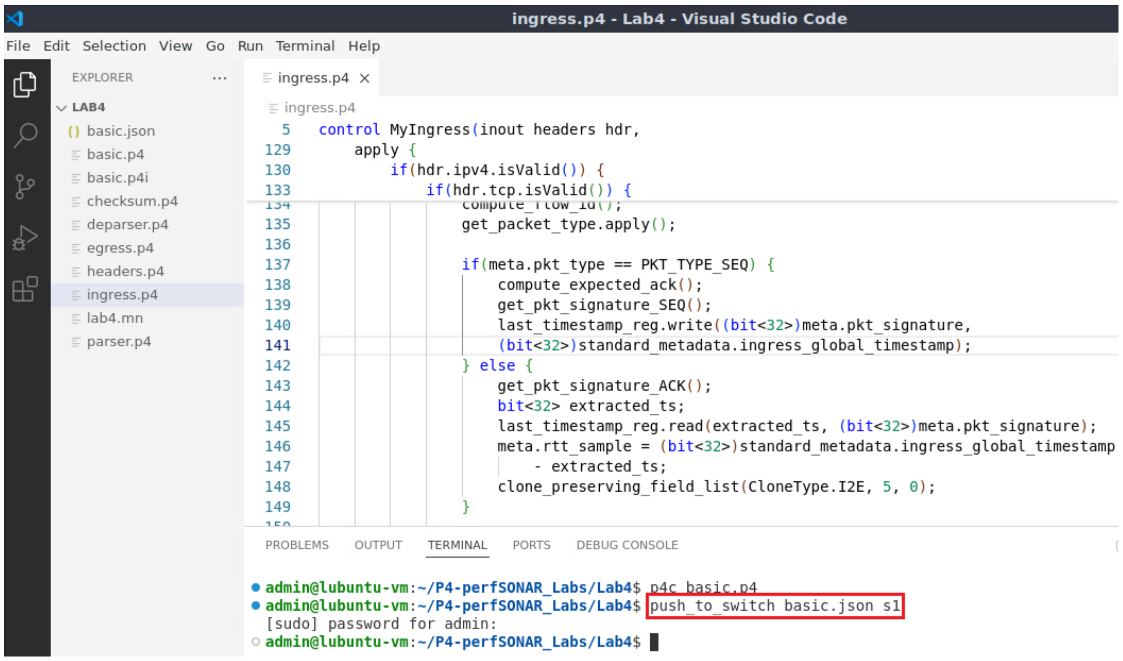
Figure 25. Compiling a P4 program.

As a result, the compilation process will produce a .json file (i.e., *basic.json*), which is similar to a binary that is interpreted by the software switch.

4.2 Loading the P4 program

Step 1. 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
```



```
control MyIngress(inout headers hdr,
    apply {
        if(hdr.ipv4.isValid()) {
            if(hdr.tcp.isValid()) {
                compute_flow_id();
                get_packet_type.apply();

                if(meta.pkt_type == PKT_TYPE_SEQ) {
                    compute_expected_ack();
                    get_pkt_signature_SEQ();
                    last_timestamp_reg.write((bit<32>)meta.pkt_signature,
                    (bit<32>)standard_metadata.ingress_global_timestamp);
                } else {
                    get_pkt_signature_ACK();
                    bit<32> extracted_ts;
                    last_timestamp_reg.read(extracted_ts, (bit<32>)meta.pkt_signature);
                    meta.rtt_sample = (bit<32>)standard_metadata.ingress_global_timestamp
                        - extracted_ts;
                    clone_preserving_field_list(CloneType.I2E, 5, 0);
                }
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab4$
```

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

4.3 Verifying the configuration

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

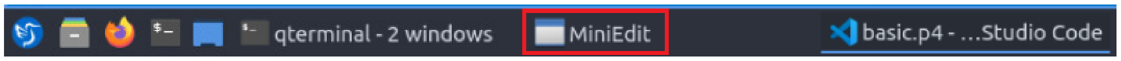


Figure 27. Maximizing the MiniEdit window.

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

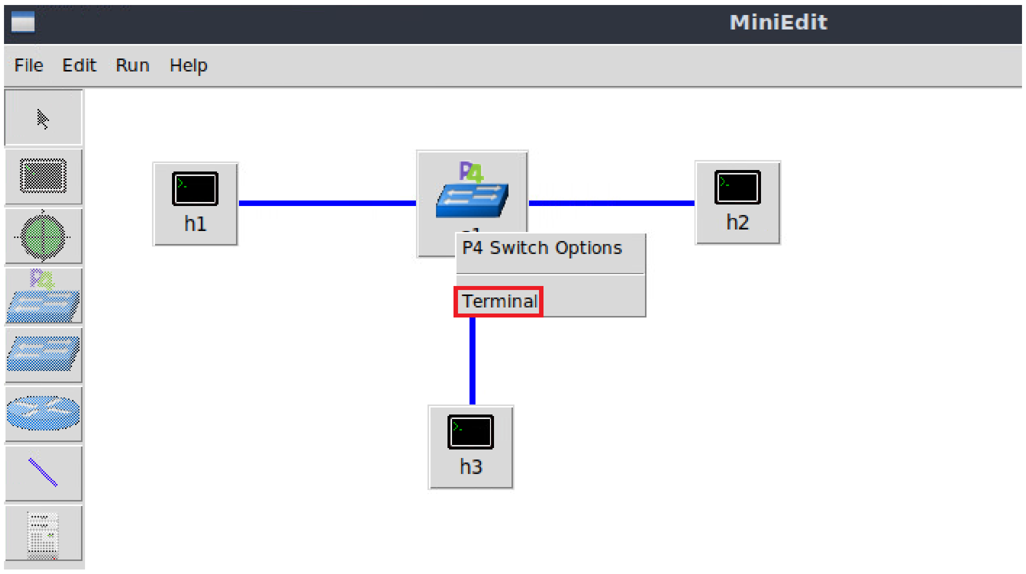
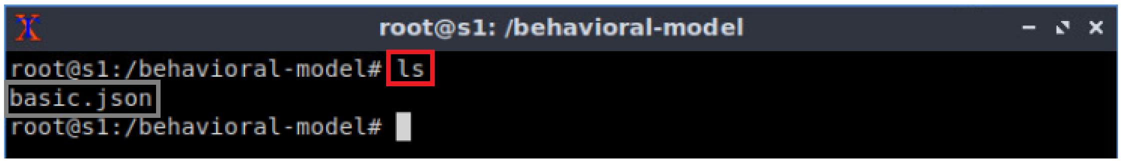


Figure 28. Starting the terminal on the switch.

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
```



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

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.

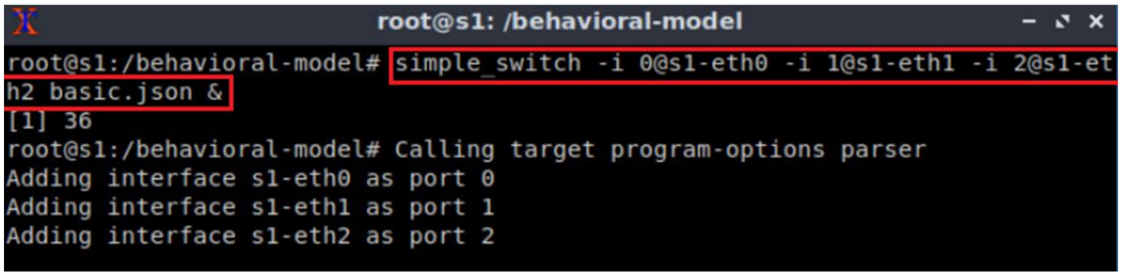
5 Configuring switch s1

In this section, you will map switch s1 interfaces to the ports in the P4 program and start the switch daemon. 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 &
```

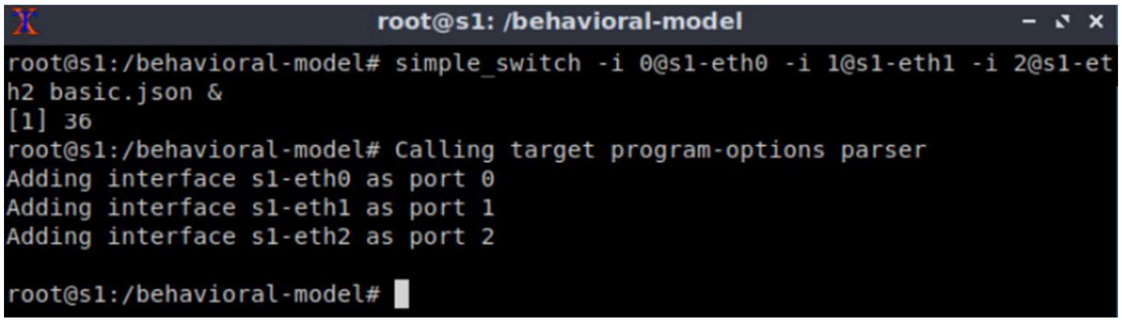


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 36
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 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.

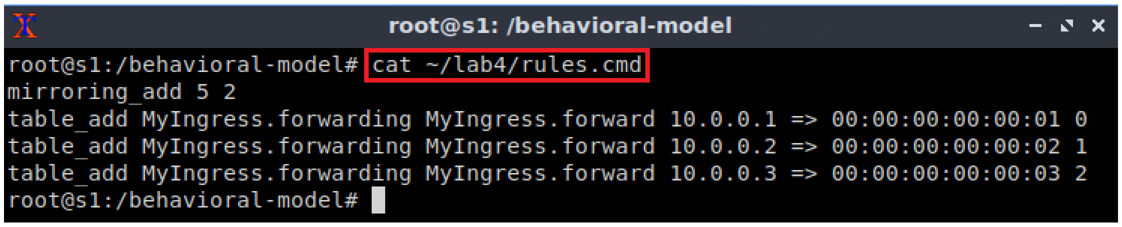


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 36
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model# 
```

Figure 31. Returning to switch s1 CLI.

Step 2. Inspect the control plane rules by issuing the following command:

```
cat ~/lab4/rules.cmd
```

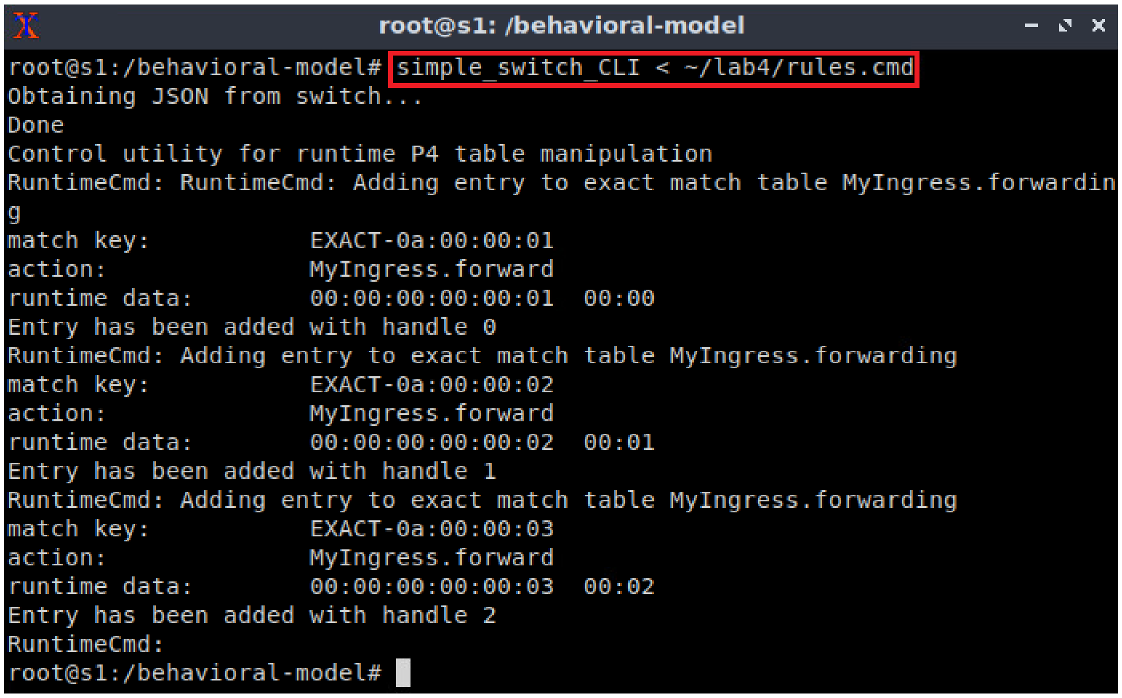


```
root@s1:/behavioral-model# cat ~/lab4/rules.cmd
mirroring_add 5 2
table_add MyIngress.forwarding MyIngress.forward 10.0.0.1 => 00:00:00:00:00:01 0
table_add MyIngress.forwarding MyIngress.forward 10.0.0.2 => 00:00:00:00:00:02 1
table_add MyIngress.forwarding MyIngress.forward 10.0.0.3 => 00:00:00:00:00:03 2
root@s1:/behavioral-model# 
```

Figure 32. Displaying the content of the file that contains the control plane rules.

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

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



```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab4/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: RuntimeCmd: Adding entry to exact match table MyIngress.forwardin
g
match key:           EXACT-0a:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:02
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:03
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 2
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 33. 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 evaluating the P4 program

In this section you will initiate a data transfer over TCP between host h1 and host h2. Then, you will observe the RTT values in host h3. You will run an iPerf3 client and server to reproduce the data transfer and a python script on host h3 to capture the RTT samples.

Step 1. Navigate to host h2 terminal by clicking on the icon in the taskbar as shown below.

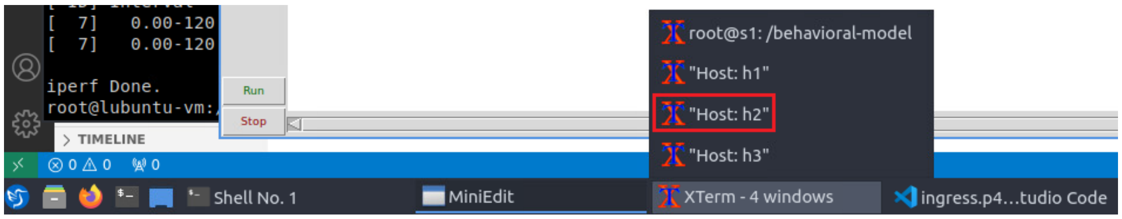
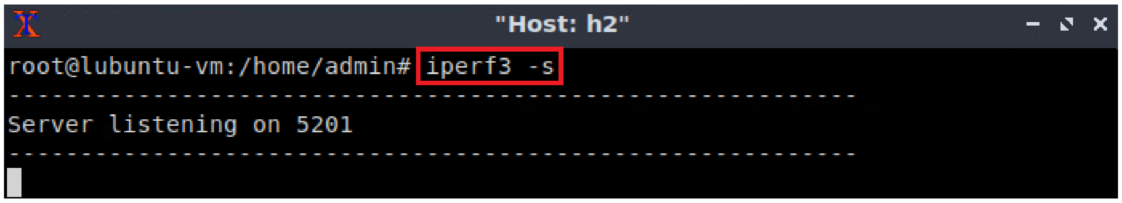


Figure 34. Navigating into host h2 terminal.

Step 2. Start the iPerf3 server in host h2 by issuing the following command:

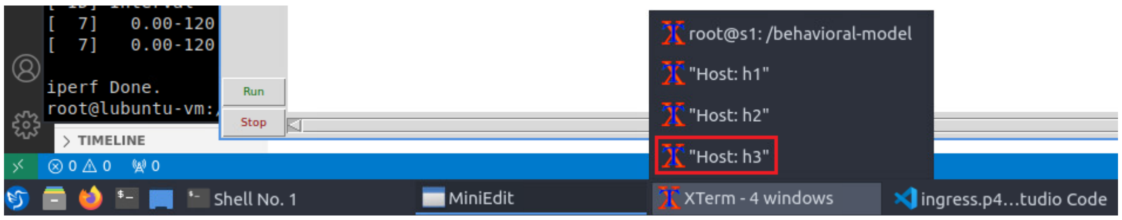
```
iperf3 -s
```



```
root@lubuntu-vm:/home/admin# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 35. Starting the iPerf3 server on host h2.

Step 3. Navigate to host h3 terminal by clicking on the icon in the taskbar as shown below.

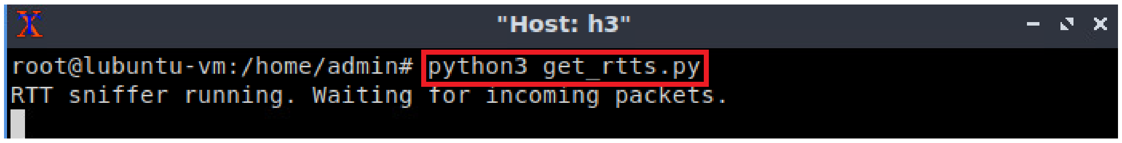


```
[  7]   0.00-120
[  7]   0.00-120

iperf Done.
root@lubuntu-vm:
```

Figure 36. Navigating into host h3 terminal.

Step 4. Start the sniffing script on host h3 by issuing the following command:

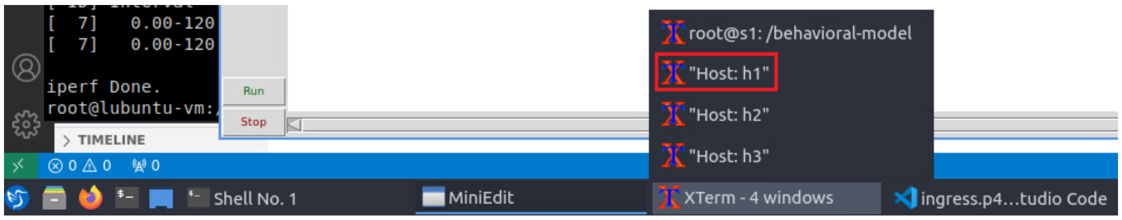
```
python3 get_rtt.py
```



```
root@lubuntu-vm:/home/admin# python3 get_rtts.py
RTT sniffer running. Waiting for incoming packets.
```

Figure 37. Starting the sniffing script on host h3.

Step 5. Navigate to host h1 terminal by clicking on the icon in the taskbar as shown below.

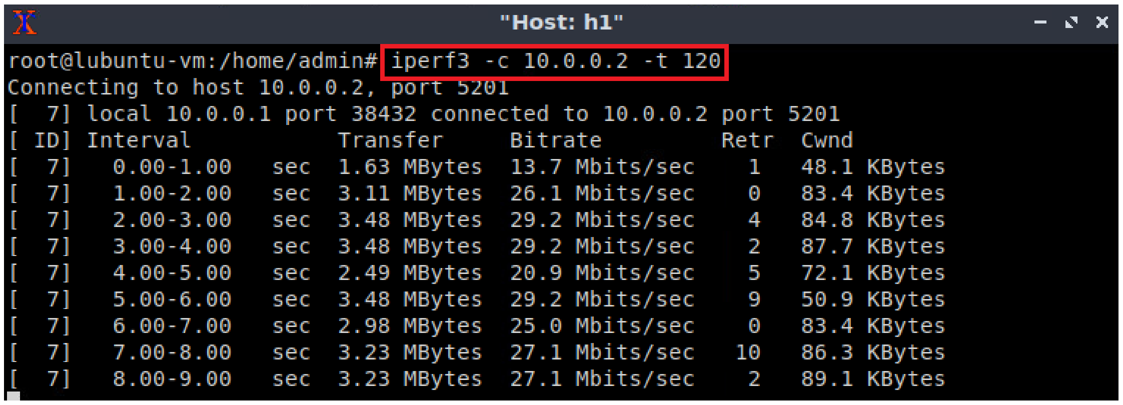


```
[  7]   0.00-120
[  7]   0.00-120

iperf Done.
root@lubuntu-vm:
```

Figure 38. Navigating into host h1 terminal.

Step 6. Start the iPerf3 client by issuing the following command. The `-c` parameters specify that iPerf3 runs in client mode, 10.0.0.2 is IP address of host h2, and `-t 120` determines that the test duration is 120 seconds.

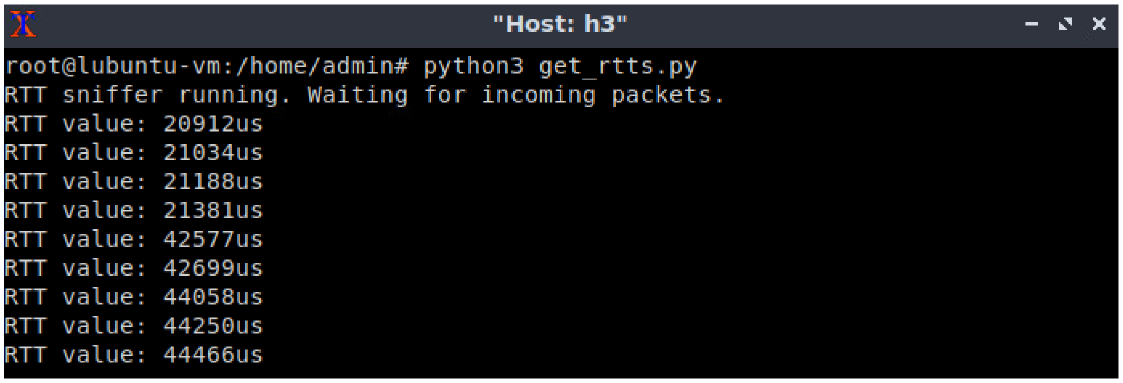
```
iperf3 -c 10.0.0.2 -t 120
```



```
"Host: h1"
root@lubuntu-vm:/home/admin# iperf3 -c 10.0.0.2 -t 120
Connecting to host 10.0.0.2, port 5201
[  7] local 10.0.0.1 port 38432 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  7]   0.00-1.00   sec  1.63 MBytes  13.7 Mbits/sec    1   48.1 KBytes
[  7]   1.00-2.00   sec  3.11 MBytes  26.1 Mbits/sec    0   83.4 KBytes
[  7]   2.00-3.00   sec  3.48 MBytes  29.2 Mbits/sec    4   84.8 KBytes
[  7]   3.00-4.00   sec  3.48 MBytes  29.2 Mbits/sec    2   87.7 KBytes
[  7]   4.00-5.00   sec  2.49 MBytes  20.9 Mbits/sec    5   72.1 KBytes
[  7]   5.00-6.00   sec  3.48 MBytes  29.2 Mbits/sec    9   50.9 KBytes
[  7]   6.00-7.00   sec  2.98 MBytes  25.0 Mbits/sec    0   83.4 KBytes
[  7]   7.00-8.00   sec  3.23 MBytes  27.1 Mbits/sec   10   86.3 KBytes
[  7]   8.00-9.00   sec  3.23 MBytes  27.1 Mbits/sec    2   89.1 KBytes
```

Figure 39. Starting the iPerf3 client on host h1.

Step 7. Go back to host h3 and observe the RTT values of the TCP flows. You will note that the value in microseconds is around 20,000, which is around 20ms.



```
"Host: h3"
root@lubuntu-vm:/home/admin# python3 get_rtts.py
RTT sniffer running. Waiting for incoming packets.
RTT value: 20912us
RTT value: 21034us
RTT value: 21188us
RTT value: 21381us
RTT value: 42577us
RTT value: 42699us
RTT value: 44058us
RTT value: 44250us
RTT value: 44466us
```

Figure 40. Observing the RTT values on host h3.

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

References

1. C. Xiaoqi, H. Kim, J. Aman, W. Chang, M. Lee, and J. Rexford, "*Measuring TCP round-trip time in the data plane*," In Proceedings of the Workshop on Secure Programmable Network Infrastructure, 2020.
2. perfSONAR Project, “*perfSONAR installation options*,” [Online]. Available: https://docs.perfsonar.net/install_options.html
3. J. Gomez, E. Kfoury, J. Crichigno, G. Srivastava, “*Reducing the impact of RTT unfairness using P4-Programmable data planes*”. The 2024 IEEE International Conference on Communications, Denver, CO, June 2024.
4. J. Gomez, E. Kfoury, J. Crichigno, G. Srivastava, “Improving TCP fairness in non-programmable networks using P4-programmable data planes,” IEEE International Black Sea Conference on Communications and Networking, Tbilisi, Georgia, June 2023.
5. A. Mazloum, J. Gomez, E. Kfoury, and J. Crichigno, "*Enhancing perfSONAR measurement capabilities using P4 programmable data planes*" In Proceedings of

the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, Denver, CO, 2023.

6. B. Lantz, G. Gee, "*MiniEdit: a simple network editor for Mininet*," [Online]. Available: https://github.com/Mininet/Mininet/blob/master/examples
7. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, "P4: programming protocol-independent packet processors," ACM SIGCOMM Computer Communication Review 44, no. 3, 2014.

UNIVERSITY OF
SOUTH CAROLINA

P4-PERFSONAR LAB SERIES

Lab 5: Configuring Regular Tests Using pScheduler CLI

Document Version: 07-17-2024

Contents

Overview 3
Objectives..... 3
Lab settings 3
Lab roadmap 3
1 Introduction to pScheduler commands 3
1.1 The pScheduler command 5
2 Loading and running the lab topology 6
2.1 Setting IPv4 addresses, static routes, and link conditions..... 8
3 Throughput tests..... 9
3.1 iPerf3 9
3.2 Nuttcp 11
4 Latency tests 12
4.1 One-way ping 12
4.2 Two-way ping 14
4.3 Round-Trip Time (RTT) 16
5 Traceroute tests 17
5.1 Traceroute..... 17
5.2 Tracepath 19
5.3 Paris-traceroute 20
6 Running tests on remote perfSONAR nodes 21
7 Repeating tasks 22
8 Exporting and importing tasks 23
9 Displaying the schedule 24
9.1 pScheduler monitor 25
9.2 pScheduler schedule 26
10 Canceling tasks..... 27
References 28

Overview

This lab is an introduction to pScheduler commands and their utilization for conducting latency, throughput, and trace tests. In this lab, the user will use both the default and specific tools available in pScheduler for running network measurement tests.

Objectives

By the end of this lab, the user will:

1. Understand pScheduler commands.
2. Measure latency using *owamp*, *twamp* and ping tools.
3. Run throughput tests using *iperf3* and *nuttcp* tools.
4. Use *traceroute*, *tracepath* and *paris-tracecoute* tools to identify the hops from a source to a destination.

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 pScheduler commands.
2. Section 2: Loading and running the lab topology.
3. Section 3: Throughput tests.
4. Section 4: Latency tests.
5. Section 5: Traceroute tests.
6. Section 6: Running tests on remote perfSONAR nodes.
7. Section 7: Repeating tasks.
8. Section 8: Exporting and importing tasks.
9. Section 9: Displaying the schedule.
10. Section 10: Cancelling tasks.

1 Introduction to pScheduler commands

pScheduler commands are used to manage network measurements and tasks within the perfSONAR framework. When it comes to executing network measurements in perfSONAR, the responsibility lies with pScheduler command-line operations. As depicted in Figure 1, pScheduler forms an integral part of the scheduling layer, which encompasses the following key functions:

- Scheduling Conflict Resolution: One of the primary tasks of the scheduling layer is to identify suitable timeslots for running measurement tools while avoiding conflicts that could potentially impact the accuracy and reliability of the results. By carefully managing the scheduling process, pScheduler ensures that measurements are executed in a controlled and optimized manner.
- Execution and Result Collection: pScheduler takes charge of executing the designated measurement tools and collecting the corresponding results. This includes coordinating with the relevant endpoints and managing any necessary daemon setup. Through its robust execution capabilities, pScheduler streamlines the process and ensures the seamless retrieval of measurement outcomes.
- Archiving Integration: In cases where long-term storage of results is required, pScheduler facilitates the seamless transmission of collected data to the archiving layer. By offering a convenient plugin architecture, it enables the integration of various storage systems, allowing users to customize the data flow as per their specific requirements.

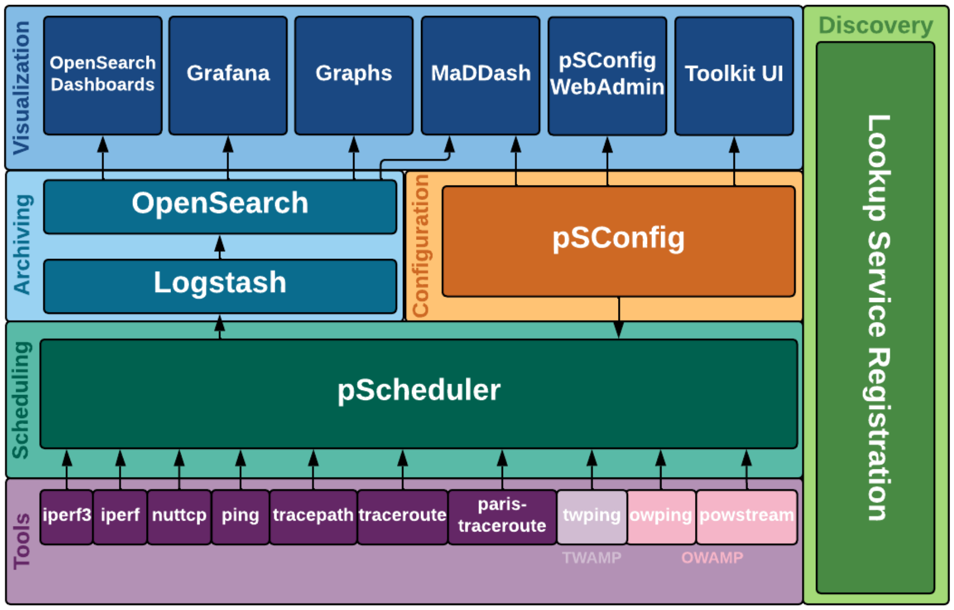


Figure 1. perfSONAR layers[3].

While many of the tools employed within perfSONAR can operate independently, pScheduler provides additional features that enhance their functionality. The following features stand out:

- Measurement Integrity: pScheduler maintains a comprehensive schedule of all measurements to be executed, ensuring that no conflicting measurements run

simultaneously. This meticulous approach prevents any undesirable resource competition that could significantly impact the accuracy of results. For instance, it intelligently avoids running multiple throughput tests concurrently, while latency tests can be run simultaneously due to their low resource consumption.
- Simplified Coordination: Beyond simplifying task execution coordination, pScheduler proactively handles interactions with end devices, seamlessly managing daemon setup as required. Moreover, its plugin architecture facilitates the effortless transmission of results to alternate storage systems or analysis tools upon completion of measurements.
- Access Control: pScheduler incorporates a robust limits system, enabling administrators to define rules governing the types of measurements permitted for execution, as well as enforcing restrictions on test duration and node-specific testing criteria. This ensures controlled access and adherence to predefined measurement policies.
- Diagnostics: pScheduler equips users with powerful visualization tools to gain insights into the schedule of executed, ongoing, and upcoming tasks. It provides detailed information about task execution timestamps and stores outcome data for a certain duration. This diagnostic capability proves invaluable when troubleshooting network issues.

In addition to these fundamental features, pScheduler extends its capabilities by allowing the development of plugins for new tests, tools, and archiving systems. This extensibility empowers users to introduce novel measurement techniques, incorporate additional functionalities, and seamlessly integrate with diverse storage and analysis platforms.

1.1 The pScheduler command

In the realm of perfSONAR, the primary means of interacting with the system is through the utilization of the pScheduler command. This command serves as the gateway from the command-line to create new pScheduler tasks, which are essential for conducting measurements. The basic syntax for executing pScheduler commands is as follows:

```
pscheduler command [args]                                                  (1)
```

Here's a breakdown of the components involved:

- `pScheduler`: This command is used to initiate interactions with perfSONAR, serving as the entry point for task creation and management.
- `command`: Each pScheduler command corresponds to a specific type of test or serves administrative and diagnostic purposes. These commands are accompanied by their respective lists of arguments (`args`). The task-related commands are as follows:

  - `task`: This command allows users to provide pScheduler with a task, which involves conducting one or more measurements.
  - `result`: With this command, users can fetch and display the results of a previously concluded run by specifying its URL.

  - `watch`: By attaching to a task identified by its URL, users can monitor the run results in real-time as they become available.
  - `cancel`: This command halts any future runs associated with a task, effectively canceling its execution.

In addition to the task-related commands, pScheduler offers the following commands for diagnostic and administrative purposes:

- `ping`: This command aids in determining if pScheduler is running on a specific host.
- `clock`: By employing the clock command, users can check and compare the clocks on pScheduler hosts for synchronization purposes.
- `debug`: This command enables debugging functionality on the internal components of pScheduler, facilitating troubleshooting activities.
- `diags`: With the `diags` command, users can generate a diagnostic dump, which can be shared with the perfSONAR team for assistance in resolving any encountered issues.

For more comprehensive information about pScheduler tasks, diagnosis, and administrative commands, users can access the help section by typing the following command:

```
pscheduler --help
```

To get more details about a specific command, using the format of the command (1) type:

```
pscheduler [command] --help
```

These resources provide extensive guidance and support, ensuring users can leverage the full potential of pScheduler for their measurement and administrative needs within perfSONAR.

2 Loading and running the lab topology

The topology is loaded using MiniEdit[6], which is the graphical tool used to create topologies in Mininet. During this lab, the user will access perfSONAR CLI to run network measurement tests.

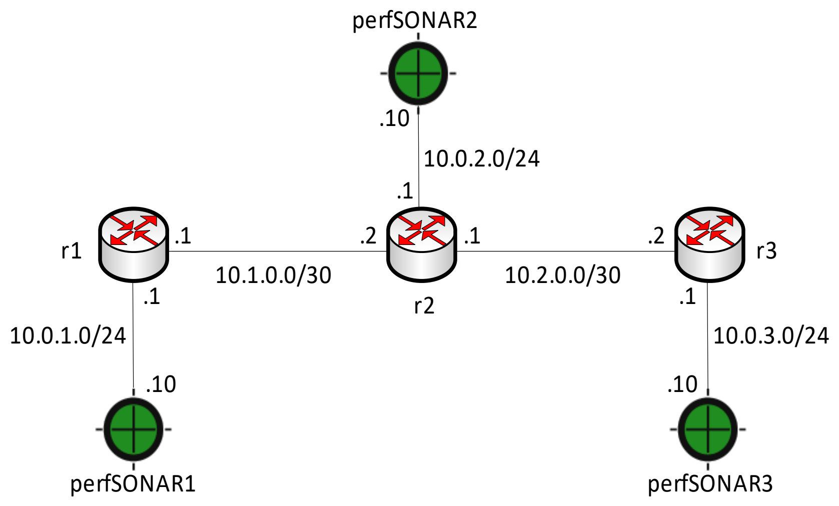


Figure 2. Lab topology.

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

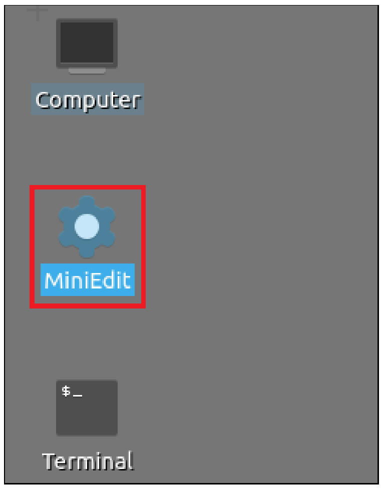


Figure 3. MiniEdit shortcut.

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

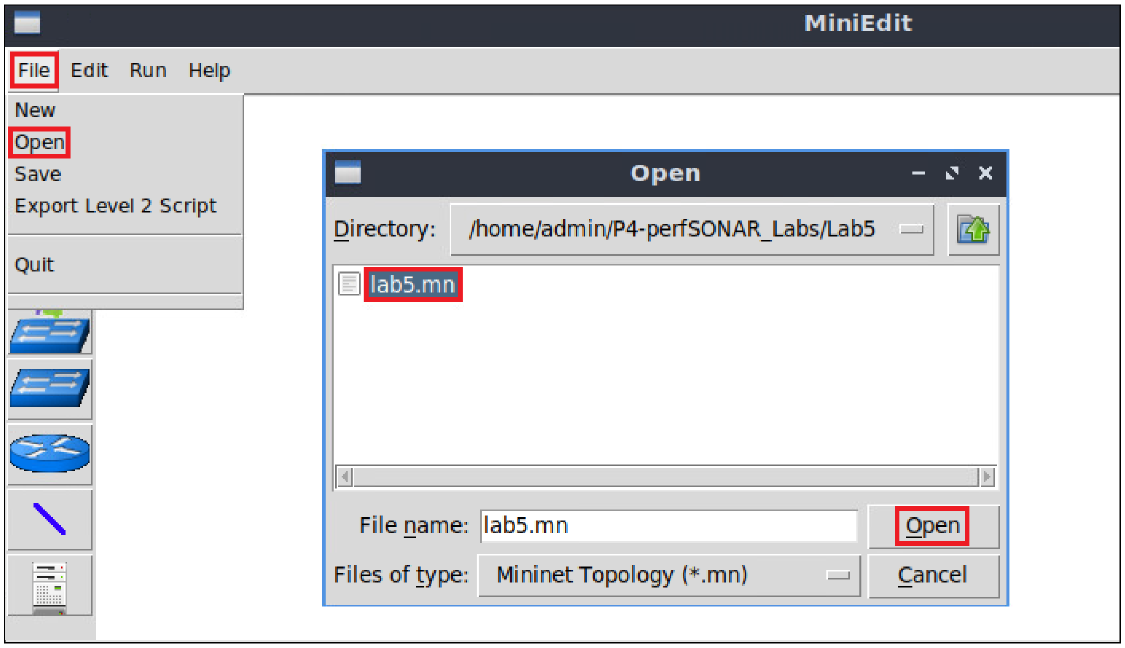


Figure 4. Opening a topology in MiniEdit.

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.

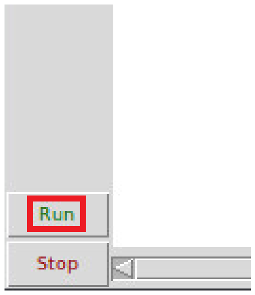


Figure 5. Running the emulation.

Wait for 20 seconds to allow the perfSONAR nodes to initialize within the network topology. Once the topology is fully operational, MiniEdit's toolbar on the left-hand side will grey out.

2.1 Setting IPv4 addresses, static routes, and link conditions

Step 1. Open a Linux terminal by clicking on the icon located on the taskbar.

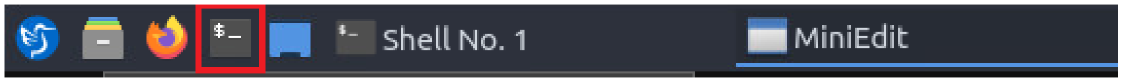
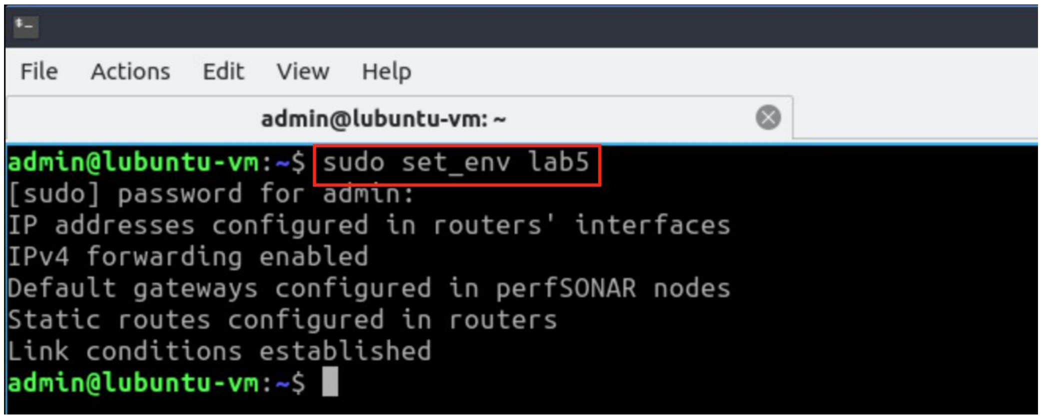


Figure 6. Opening a Linux terminal.

Step 2. To set up the lab environment, execute the following command in the Linux terminal. This script will configure the IPv4 addresses of the routers, define static routes, and establish specific link conditions, such as bandwidth limitations and minimum delays.

```
sudo set_env lab5
```



```
admin@lubuntu-vm:~$ sudo set_env lab5
[sudo] password for admin:
IP addresses configured in routers' interfaces
IPv4 forwarding enabled
Default gateways configured in perfSONAR nodes
Static routes configured in routers
Link conditions established
admin@lubuntu-vm:~$
```

Figure 7. Setting IPv4 addresses, static routes, links' bandwidths, and delays.

3 Throughput tests

This section shows how to conduct throughput measurements utilizing pScheduler tools. These tools, including iperf3 and nuttcp, are visually represented in the tools layer depicted in Figure 1. Users will execute throughput tests using the default tool, iperf3, and select nuttcp to conduct another test.

3.1 iPerf3

iPerf3 is a widely used network performance testing tool that allows users to measure the maximum achievable bandwidth and throughput between two endpoints in a network. It is designed to provide accurate and reliable measurements of TCP and UDP performance. With iperf3, users can assess network performance by generating synthetic traffic and measuring the resulting throughput, Round-trip Time (RTT), and packet loss. It supports both client-server and peer-to-peer testing modes, making it versatile for various network testing scenarios. iperf3 offers several key features, including the ability to set various testing parameters such as packet size, TCP window size, and bandwidth limits. It provides detailed statistics and reports on throughput, jitter, and packet loss, aiding in network troubleshooting and optimization.

Step 1. Hold the right click on perfSONAR1 and select *Terminal.* The perfSONAR1 CLI will emerge.

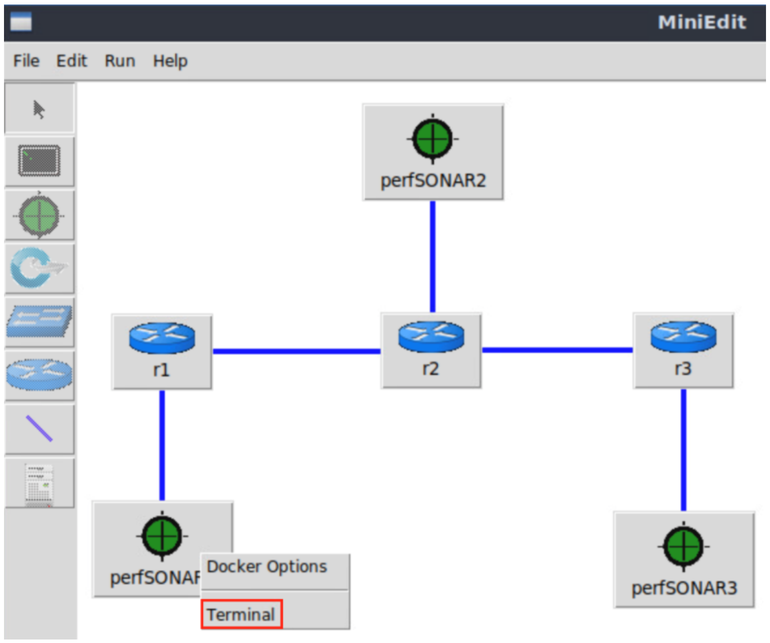
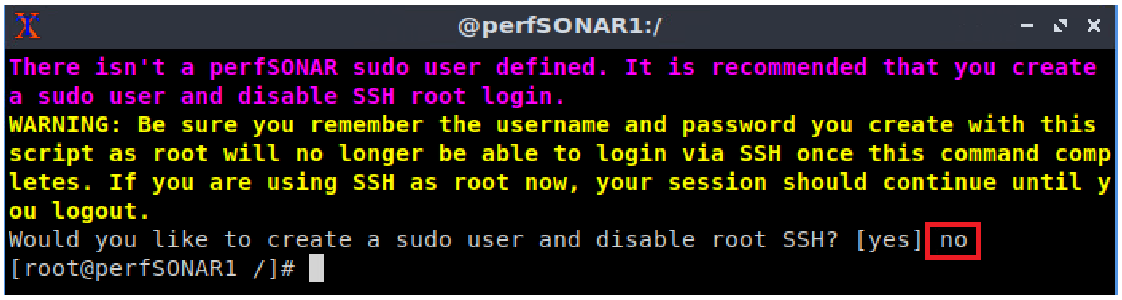


Figure 8. Opening perfSONAR1 CLI.

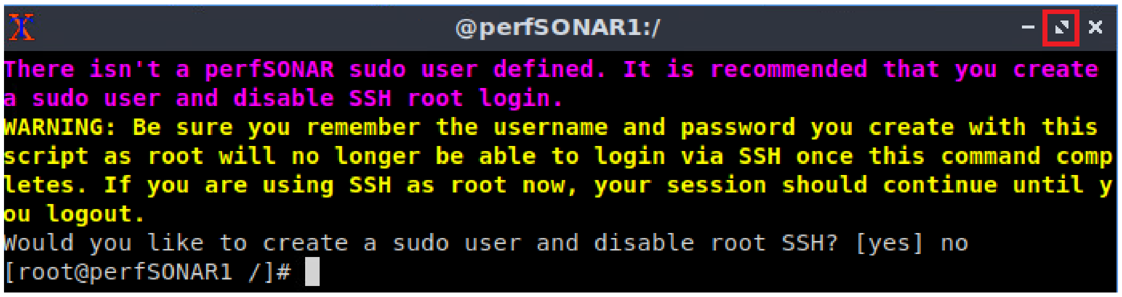
Step 2. In perfSONAR terminal, type `no` and press *Enter* to enable the CLI.



```
@perfSONAR1:/
There isn't a perfSONAR sudo user defined. It is recommended that you create
a sudo user and disable SSH root login.
WARNING: Be sure you remember the username and password you create with this
script as root will no longer be able to login via SSH once this command comp
letes. If you are using SSH as root now, your session should continue until y
ou logout.
Would you like to create a sudo user and disable root SSH? [yes] no
[root@perfSONAR1 /]#
```

Figure 9. Skipping initial configuration in perfSONAR1.

Step 3. Enlarge the terminal by clicking on the icon shown in the figure below.



```
@perfSONAR1:/
There isn't a perfSONAR sudo user defined. It is recommended that you create
a sudo user and disable SSH root login.
WARNING: Be sure you remember the username and password you create with this
script as root will no longer be able to login via SSH once this command comp
letes. If you are using SSH as root now, your session should continue until y
ou logout.
Would you like to create a sudo user and disable root SSH? [yes] no
[root@perfSONAR1 /]#
```

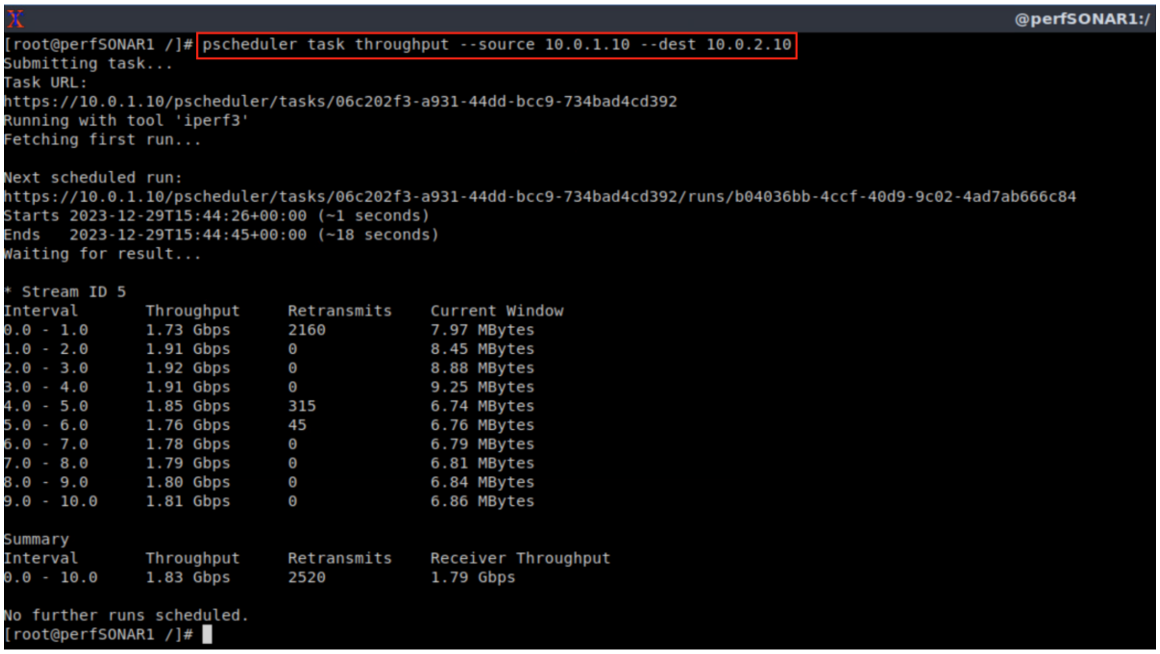
Figure 10. Enlarging perfSONAR1 CLI.

Step 4. In perfSONAR1 issue the following command to run a throughput test using the default tool. The nodes participating in this test are perfSONAR1 and perfSONAR2.

```
pscheduler task throughput --source 10.0.0.10 --dest 10.0.2.10
```

- `pscheduler`: command to interact with perfSONAR.

- `task`: pScheduler command.
- `throughput`: test type.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: destination node, in this case it is the perfSONAR2 node (*10.0.2.10*).



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task throughput --source 10.0.1.10 --dest 10.0.2.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/06c202f3-a931-44dd-bcc9-734bad4cd392
Running with tool 'iperf3'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/06c202f3-a931-44dd-bcc9-734bad4cd392/runs/b04036bb-4ccf-40d9-9c02-4ad7ab666c84
Starts 2023-12-29T15:44:26+00:00 (~1 seconds)
Ends   2023-12-29T15:44:45+00:00 (~18 seconds)
Waiting for result...

* Stream ID 5
Interval       Throughput     Retransmits    Current Window
0.0 - 1.0      1.73 Gbps      2160           7.97 MBytes
1.0 - 2.0      1.91 Gbps      0              8.45 MBytes
2.0 - 3.0      1.92 Gbps      0              8.88 MBytes
3.0 - 4.0      1.91 Gbps      0              9.25 MBytes
4.0 - 5.0      1.85 Gbps      315            6.74 MBytes
5.0 - 6.0      1.76 Gbps      45             6.76 MBytes
6.0 - 7.0      1.78 Gbps      0              6.79 MBytes
7.0 - 8.0      1.79 Gbps      0              6.81 MBytes
8.0 - 9.0      1.80 Gbps      0              6.84 MBytes
9.0 - 10.0     1.81 Gbps      0              6.86 MBytes

Summary
Interval       Throughput     Retransmits    Receiver Throughput
0.0 - 10.0     1.83 Gbps      2520           1.79 Gbps

No further runs scheduled.
[root@perfSONAR1 /]# 
```

Figure 11. Running a throughput test using the default tool (*iperf3*).

Shortly after starting the test submission, the user will see that the tool used to run the test is iperf3. The results above list the throughput every second (*Interval*), the number of retransmissions (*Retransmits*) and the congestion window size (i.e., *Current Window*). At the end, it is summarized the time interval when the test took place, in this case from 0 seconds to 10 seconds, the throughput is 1.83 Gbps and the number of retransmissions is 2520.

3.2 Nuttcp

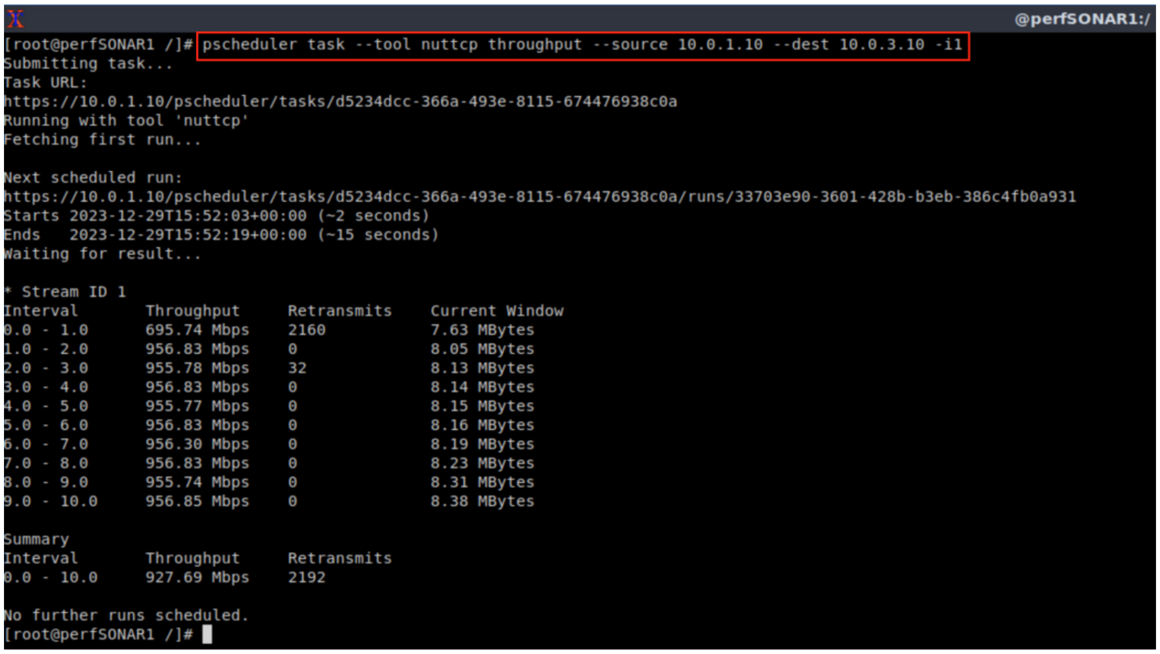
Nuttcp is a network performance testing tool that measures TCP and UDP throughput. It is designed to provide accurate and reliable measurements of network performance in terms of throughput and latency. *Nuttcp* offers various test modes, including bulk transfer, request/response, and parallel testing. These modes allow for more flexibility in conducting specific types of performance measurements.

Step 1. In perfSONAR1 issue the following command to run a throughput test using *nuttcp*. The nodes participating in this test are perfSONAR1 and perfSONAR3.

```
pscheduler task --tool nuttcp throughput --source 10.0.1.10 --dest 10.0.3.10
-i1
```

- `pscheduler`: command to interact with perfSONAR.

- `task`: pScheduler command.
- `--tool`: command to specify the tool.
- `nuttcp`: tool used to run the test.
- `throughput`: test type.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: destination node, in this case it is the perfSONAR3 node (*10.0.3.10*).
- `i1`: indicates the interval is 1 second.



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task --tool nuttcp throughput --source 10.0.1.10 --dest 10.0.3.10 -i1
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/d5234dcc-366a-493e-8115-674476938c0a
Running with tool 'nuttcp'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/d5234dcc-366a-493e-8115-674476938c0a/runs/33703e90-3601-428b-b3eb-386c4fb0a931
Starts 2023-12-29T15:52:03+00:00 (~2 seconds)
Ends   2023-12-29T15:52:19+00:00 (~15 seconds)
Waiting for result...

* Stream ID 1
Interval       Throughput     Retransmits    Current Window
0.0 - 1.0      695.74 Mbps    2160           7.63 MBytes
1.0 - 2.0      956.83 Mbps    0              8.05 MBytes
2.0 - 3.0      955.78 Mbps    32             8.13 MBytes
3.0 - 4.0      956.83 Mbps    0              8.14 MBytes
4.0 - 5.0      955.77 Mbps    0              8.15 MBytes
5.0 - 6.0      956.83 Mbps    0              8.16 MBytes
6.0 - 7.0      956.30 Mbps    0              8.19 MBytes
7.0 - 8.0      956.83 Mbps    0              8.23 MBytes
8.0 - 9.0      955.74 Mbps    0              8.31 MBytes
9.0 - 10.0     956.85 Mbps    0              8.38 MBytes

Summary
Interval       Throughput     Retransmits
0.0 - 10.0     927.69 Mbps    2192

No further runs scheduled.
[root@perfSONAR1 /]# 
```

Figure 12. Running a throughput test using *nuttcp*.

After starting the test submission, the user will see that the tool used to run the test is *nuttcp*. The results above list the throughput every second (*Interval*), the number of retransmissions (*Retransmits*) and the congestion window size (i.e., *Current Window*). At the end, it is summarized the time interval when the test took place, in this case from 0 seconds to 10 seconds, the throughput is 927.69 Mbps and the number of retransmissions is 2192.

4 Latency tests

In this section, users will conduct latency measurement tests utilizing the powerful pScheduler tools. These tools, including One-Way Ping (*owping*), Two-Way Ping (*twping*), and RTT, are harnessed by pScheduler to accurately measure network latency. The section commences with a latency test employing the default configuration, followed by the user's ability to specify a preferred tool for conducting a latency test.

4.1 One-way ping

Owping is a network performance testing tool that focuses on measuring one-way latency between network endpoints. It allows users to assess the time it takes for packets to travel from the sender to the receiver, providing insights into network performance and potential bottlenecks. *Owping* operates by sending ICMP (Internet Control Message Protocol) echo request packets from the sender to the receiver. The receiver records the arrival time of each packet and sends an echo reply to the sender. By comparing the sent and received timestamps, *owping* calculates the one-way latency. This tool is particularly useful for evaluating the performance of specific network paths or detecting issues that affect one-way traffic. It helps network administrators and operators pinpoint latency problems, identify network congestion points, and troubleshoot connectivity issues. *owping* provides real-time measurements and statistics, allowing users to monitor network latency and make informed decisions regarding network optimization and performance improvements.

Step 1. In perfSONAR1 issue the following command to run a throughput test using the default tool. The nodes participating in this test are perfSONAR1 and perfSONAR3.

```
pscheduler task latency --source 10.0.1.10 --dest 10.0.3.10
```

- `pscheduler`**:** command to interact with perfSONAR.
- `task`**:** pScheduler command to specify a measurement test.
- `latency`: test type. The default tool is owping.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: destination node, in this case it is the perfSONAR3 node (*10.0.3.10*).

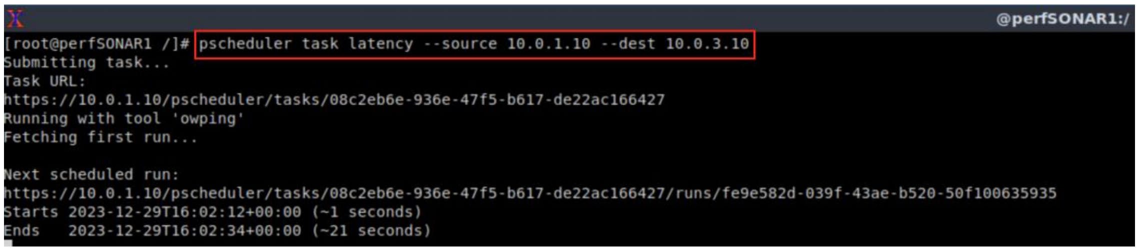
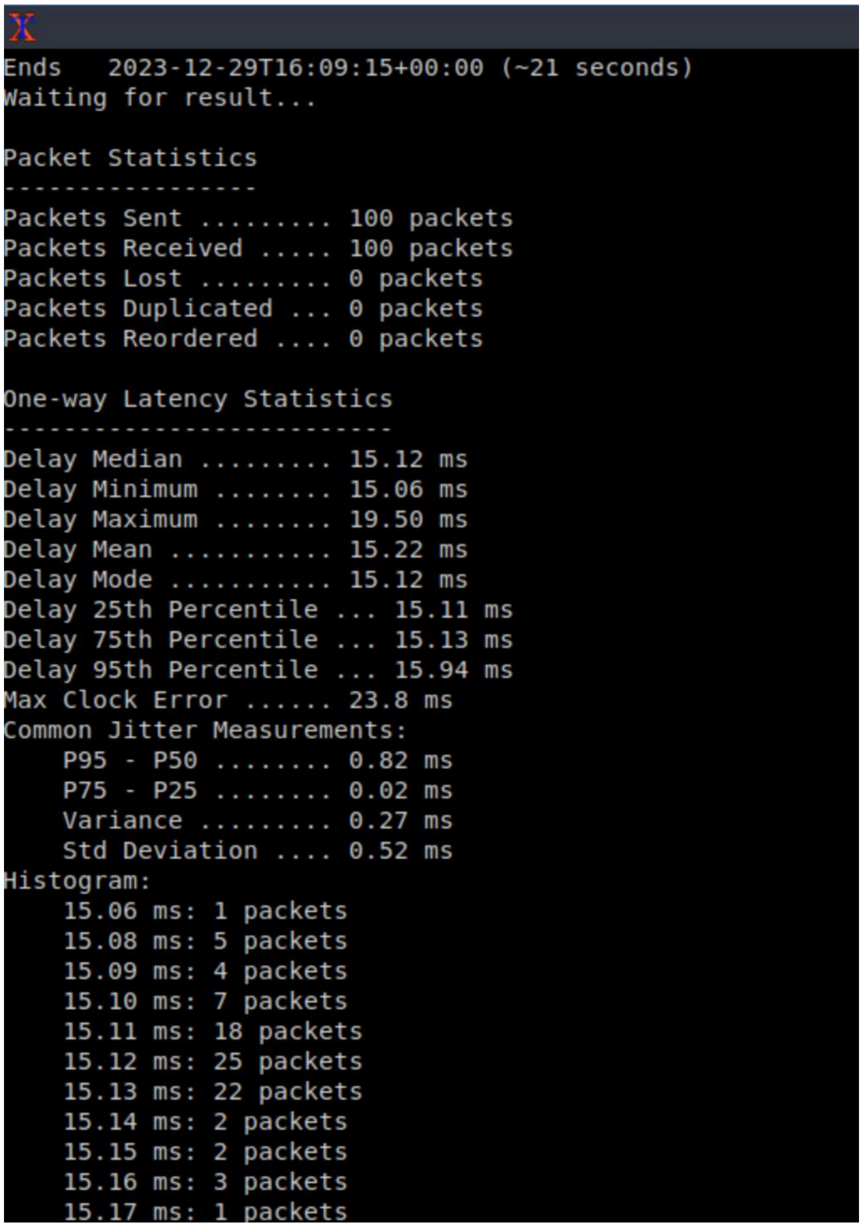

```
[root@perfSONAR1 /]# pscheduler task latency --source 10.0.1.10 --dest 10.0.3.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/08c2eb6e-936e-47f5-b617-de22ac166427
Running with tool 'owping'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/08c2eb6e-936e-47f5-b617-de22ac166427/runs/fe9e582d-039f-43ae-b520-50f100635935
Starts 2023-12-29T16:02:12+00:00 (~1 seconds)
Ends   2023-12-29T16:02:34+00:00 (~21 seconds)
```

Figure 13. Running a latency test using the default tool (*owping)*.

The default tool utilized by pScheduler for the default test is the one-way ping (*owping*) tool. Upon scheduling the task, the resulting report comprises three sections, each providing specific insights:

- Packet Statistics: This section presents a summary of the sent and received packets, along with details regarding packet loss, duplication, and reordering.
- One-way Latency Statistics: Here, the report offers a comprehensive overview of the one-way latency statistics. Additionally, a histogram is provided to visualize the distribution of delay values.
- TTL Statistics: The TTL (Time-to-Live) statistics section showcases the TTL values observed during the test.

Step 2. Scroll down to see the packet statistics, one-way latency statistics, common jitter measurements, histogram, and Time-to-Live (TTL) statistics.



```
Ends   2023-12-29T16:09:15+00:00 (~21 seconds)
Waiting for result...

Packet Statistics
-----------------
Packets Sent ......... 100 packets
Packets Received ..... 100 packets
Packets Lost ......... 0 packets
Packets Duplicated ... 0 packets
Packets Reordered .... 0 packets

One-way Latency Statistics
--------------------------
Delay Median ......... 15.12 ms
Delay Minimum ........ 15.06 ms
Delay Maximum ........ 19.50 ms
Delay Mean ........... 15.22 ms
Delay Mode ........... 15.12 ms
Delay 25th Percentile ... 15.11 ms
Delay 75th Percentile ... 15.13 ms
Delay 95th Percentile ... 15.94 ms
Max Clock Error ...... 23.8 ms
Common Jitter Measurements:
    P95 - P50 ........ 0.82 ms
    P75 - P25 ........ 0.02 ms
    Variance ......... 0.27 ms
    Std Deviation .... 0.52 ms
Histogram:
    15.06 ms: 1 packets
    15.08 ms: 5 packets
    15.09 ms: 4 packets
    15.10 ms: 7 packets
    15.11 ms: 18 packets
    15.12 ms: 25 packets
    15.13 ms: 22 packets
    15.14 ms: 2 packets
    15.15 ms: 2 packets
    15.16 ms: 3 packets
    15.17 ms: 1 packets
```

Figure 14. Visualizing the statistics of *owping.*

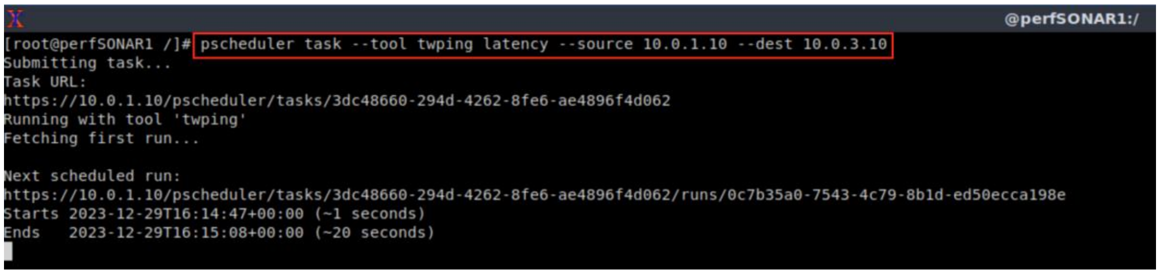
4.2 Two-way ping

Twping is a network performance testing tool that measures bidirectional latency between network endpoints. It allows users to evaluate the RTT of packets traveling between the sender and receiver. Unlike traditional ping, which only measures one-way latency, *twping* employs a bidirectional approach. It sends test packets from the sender to the receiver and back again, recording the time it takes for the round trip. By comparing the sent and received timestamps, *twping* calculates the RTT, providing insights into the overall latency of the network path. *twping* is useful for assessing the bidirectional performance of network connections and identifying any asymmetry in latency between the sender and receiver. It helps network administrators and operators troubleshoot latency-related issues, optimize network performance, and ensure a balanced and reliable communication path.

Step 1. In perfSONAR1 issue the following command to run a throughput test using the default tool. The nodes participating in this test are perfSONAR1 and perfSONAR3.

```
pscheduler task --tool twping latency --source 10.0.1.10 --dest 10.0.3.10
```

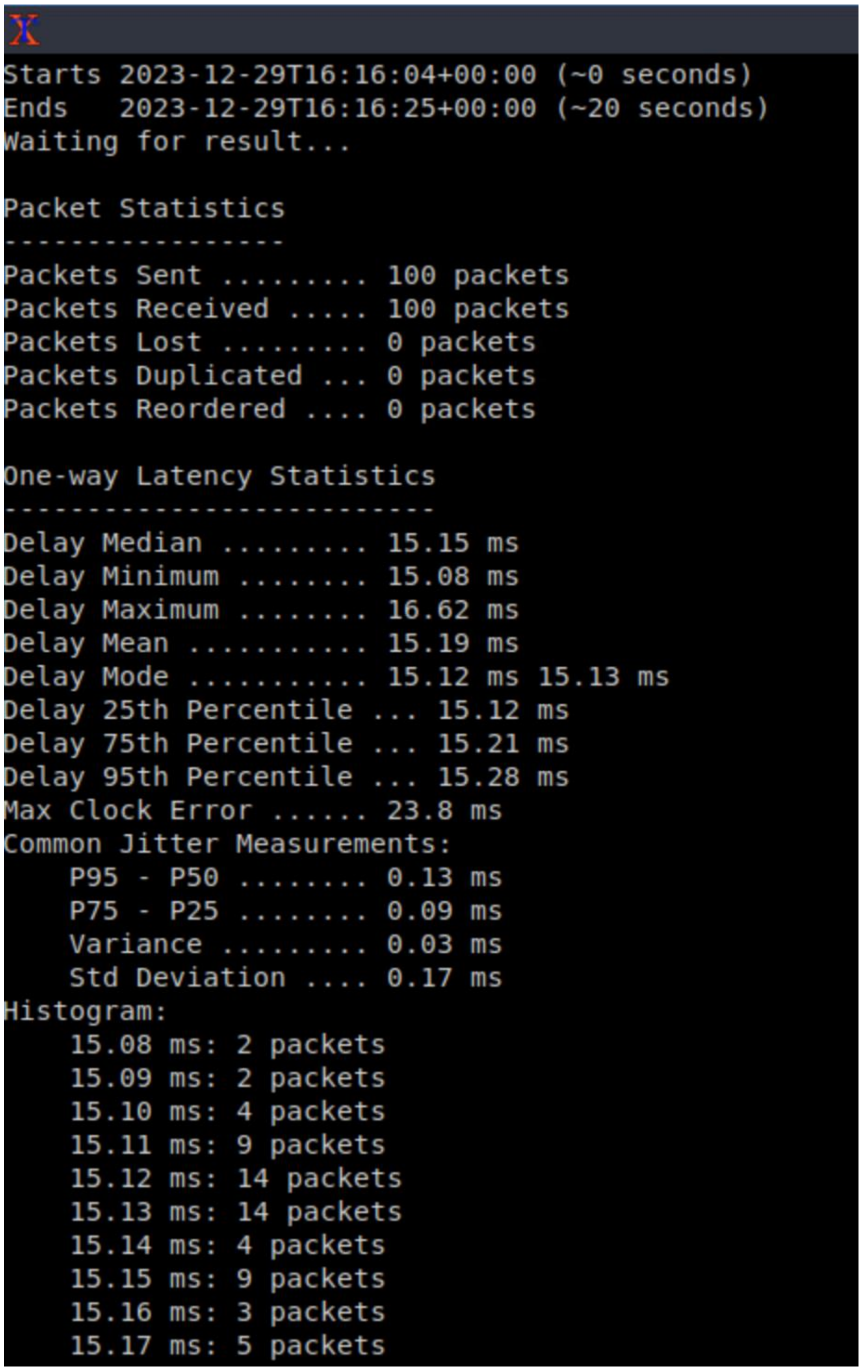
- `pscheduler`: command to interact with perfSONAR.
- `task`: pScheduler command to specify a measurement test.
- `--tool`: command to specify the tool.
- `twping`: tool for two-way ping measurement.
- `latency`: test type.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: the destination node, in this case it is the perfSONAR3 node (*10.0.3.10*).



```
[root@perfSONAR1 /]# pscheduler task --tool twping latency --source 10.0.1.10 --dest 10.0.3.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/3dc48660-294d-4262-8fe6-ae4896f4d062
Running with tool 'twping'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/3dc48660-294d-4262-8fe6-ae4896f4d062/runs/0c7b35a0-7543-4c79-8b1d-ed50ecca198e
Starts 2023-12-29T16:14:47+00:00 (~1 seconds)
Ends   2023-12-29T16:15:08+00:00 (~20 seconds)
```

Figure 15. Running a latency test using *twping.*

Step 2. Scroll down to see the packet statistics, one-way latency statistics, common jitter measurements, histogram, and Time-to-Live (TTL) statistics.



```
Starts 2023-12-29T16:16:04+00:00 (~0 seconds)
Ends   2023-12-29T16:16:25+00:00 (~20 seconds)
Waiting for result...

Packet Statistics
-----------------
Packets Sent ......... 100 packets
Packets Received ..... 100 packets
Packets Lost ......... 0 packets
Packets Duplicated ... 0 packets
Packets Reordered .... 0 packets

One-way Latency Statistics
--------------------------
Delay Median ......... 15.15 ms
Delay Minimum ........ 15.08 ms
Delay Maximum ........ 16.62 ms
Delay Mean ........... 15.19 ms
Delay Mode ........... 15.12 ms 15.13 ms
Delay 25th Percentile ... 15.12 ms
Delay 75th Percentile ... 15.21 ms
Delay 95th Percentile ... 15.28 ms
Max Clock Error ...... 23.8 ms
Common Jitter Measurements:
    P95 - P50 ........ 0.13 ms
    P75 - P25 ........ 0.09 ms
    Variance ......... 0.03 ms
    Std Deviation .... 0.17 ms
Histogram:
    15.08 ms: 2 packets
    15.09 ms: 2 packets
    15.10 ms: 4 packets
    15.11 ms: 9 packets
    15.12 ms: 14 packets
    15.13 ms: 14 packets
    15.14 ms: 4 packets
    15.15 ms: 9 packets
    15.16 ms: 3 packets
    15.17 ms: 5 packets
```

Figure 16. Visualizing the statistics of *twping*.

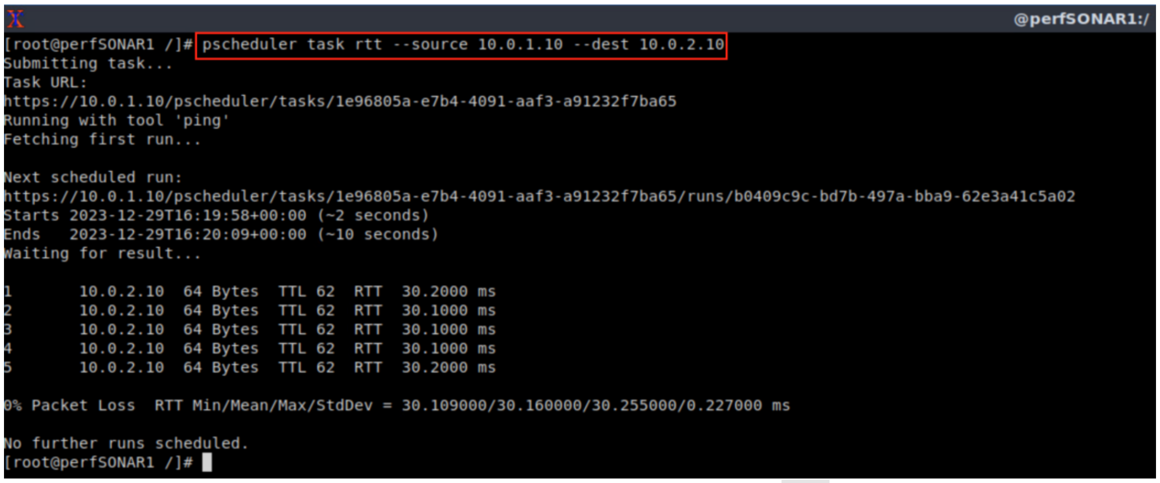
4.3 Round-Trip Time (RTT)

The RTT tool works by sending ICMP (Internet Control Message Protocol) echo request packets from the sender to the receiver and waiting for an echo reply. The sender records the time it takes for the packet to reach the receiver and return, calculating the RTT in milliseconds.

Step 1. In perfSONAR1 issue the following command to run a throughput test using the default tool. The nodes participating in this test are perfSONAR1 and perfSONAR2.

```
pscheduler task rtt --source 10.0.1.10 --dest 10.0.2.10
```

- `pscheduler`: command to interact with perfSONAR.
- `task`: pScheduler command to specify a measurement test.
- `rtt`: test type.
- `--source`: it specifies where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: destination node, in this case it is the perfSONAR2 node (*10.0.2.10*).



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task rtt --source 10.0.1.10 --dest 10.0.2.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/1e96805a-e7b4-4091-aaf3-a91232f7ba65
Running with tool 'ping'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/1e96805a-e7b4-4091-aaf3-a91232f7ba65/runs/b0409c9c-bd7b-497a-bba9-62e3a41c5a02
Starts 2023-12-29T16:19:58+00:00 (~2 seconds)
Ends   2023-12-29T16:20:09+00:00 (~10 seconds)
Waiting for result...

1       10.0.2.10  64 Bytes  TTL 62  RTT  30.2000 ms
2       10.0.2.10  64 Bytes  TTL 62  RTT  30.1000 ms
3       10.0.2.10  64 Bytes  TTL 62  RTT  30.1000 ms
4       10.0.2.10  64 Bytes  TTL 62  RTT  30.1000 ms
5       10.0.2.10  64 Bytes  TTL 62  RTT  30.2000 ms

0% Packet Loss  RTT Min/Mean/Max/StdDev = 30.109000/30.160000/30.255000/0.227000 ms

No further runs scheduled.
[root@perfSONAR1 /]#
```

Figure 17. Running a latency test using the `rtt` tool.

The result above indicates that all five packets were received successfully by perfSONAR2 (0% packet loss) and that the minimum, mean, maximum, and standard deviation of the RTT were 30.109, 30.16, 30.255 and 0.227 milliseconds respectively.

5 Traceroute tests

In this section, users will run traceroute tests using the powerful pScheduler tools. These tools, namely traceroute, tracepath, and Paris-traceroute, enable the analysis of network paths and routing. The section commences with a trace test utilizing the default configuration, followed by the user's ability to specify a preferred tool for conducting a customized trace test. Through this exploration, users can gain valuable insights into the network topology, identify potential bottlenecks, and analyze the routing behavior of their network.

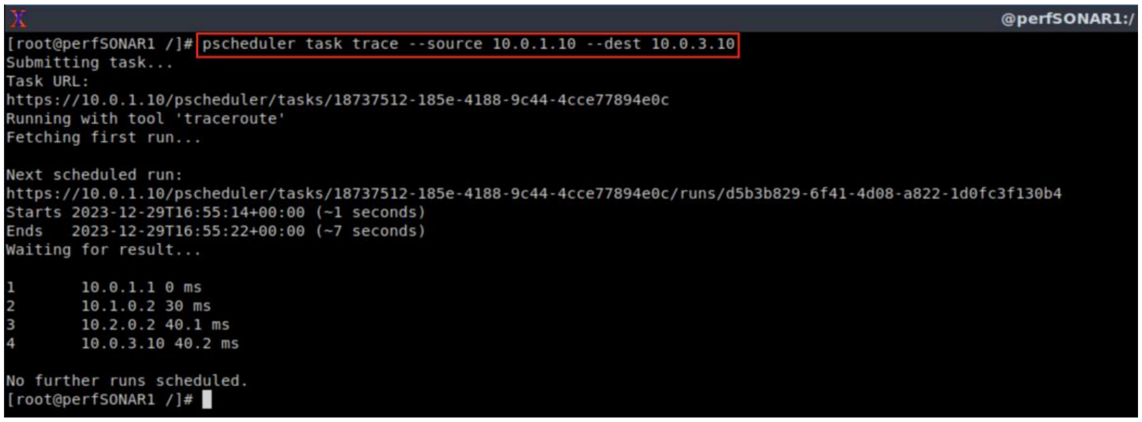
5.1 Traceroute

Traceroute is a fundamental network diagnostic tool used to trace the path that packets take from a source host to a destination host in a network. It provides valuable insights into network topology and helps identify the routers or nodes encountered along the route. By sending packets with incrementally increasing Time-to-Live (TTL) values, traceroute determines the intermediate hops by examining the ICMP Time Exceeded messages or ICMP Echo Reply messages received in response. It displays the IP addresses or domain names of the routers encountered at each hop and measures the RTT for the packets. *Traceroute* is widely used for troubleshooting network connectivity issues, analyzing routing behavior, and assessing network performance. It allows network administrators and operators to identify potential bottlenecks, latency problems, or routing inconsistencies.

Step 1. In perfSONAR1 issue the following command to run a trace test using the default tool (*traceroute)*. The nodes participating in this test are perfSONAR1 and perfSONAR3.

```
pscheduler task trace --source 10.0.1.10 --dest 10.0.3.10
```

- `pscheduler`**:** command to interact with perfSONAR.
- `task`**:** pScheduler command to specify a measurement test.
- `trace`: test type. The default tool is traceroute.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: destination node, in this case it is the perfSONAR3 node (*10.0.3.10*).



```
[root@perfSONAR1 /]# pscheduler task trace --source 10.0.1.10 --dest 10.0.3.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/18737512-185e-4188-9c44-4cce77894e0c
Running with tool 'traceroute'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/18737512-185e-4188-9c44-4cce77894e0c/runs/d5b3b829-6f41-4d08-a822-1d0fc3f130b4
Starts 2023-12-29T16:55:14+00:00 (~1 seconds)
Ends   2023-12-29T16:55:22+00:00 (~7 seconds)
Waiting for result...

1       10.0.1.1 0 ms
2       10.1.0.2 30 ms
3       10.2.0.2 40.1 ms
4       10.0.3.10 40.2 ms

No further runs scheduled.
[root@perfSONAR1 /]#
```

Figure 18. Running a trace test using the default tool (*traceroute*).

After submitting the test, the default tool used for tracing the route is *traceroute*. The report generated consists of rows divided into columns, each representing a hop along the route towards the destination.

- Hop number (first column): This column indicates the number of hops encountered along the route. In this case, it took five hops to reach the destination, with each hop representing a network node or router traversed.
- IP address (second column): The second column provides the IP address of the intermediary devices along the path to the destination. For the previous hop, it lists the IP address of the router that facilitated the connection. If available, the domain name associated with the IP address may also be displayed, providing additional context.
- RTT (third column): The subsequent column presents the RTT for the packet to travel from the source host to the current hop and return back. RTT is measured

in milliseconds and reflects the time taken for the packet's round trip. It serves as an indicator of the latency or delay experienced at each hop along the route.

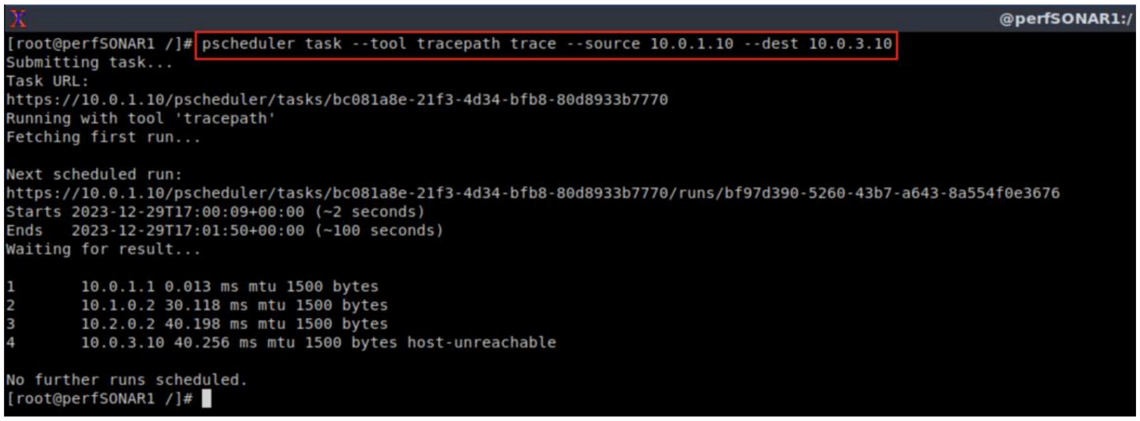
5.2 Tracepath

Tracepath is a network diagnostic tool that traces the path from a source to a destination while also discovering the Maximum Transmission Unit (MTU) along the route. It operates by sending UDP or random port packets to the destination and analyzing the responses received. Unlike *traceroute*, *tracepath* offers a simplified interface with fewer options, making it easier to use for basic path tracing and MTU discovery. It is particularly useful for non-superuser users, as it does not require elevated privileges to run tests. By utilizing tracepath, users can efficiently trace the network path, identify intermediate hops, and determine the MTU at each point. This information is valuable for troubleshooting network issues related to packet fragmentation and optimizing network performance.

Step 1. In perfSONAR1 issue the following command to run a tracepath test using *tracepath*. The nodes participating in this test are perfSONAR1 and perfSONAR3.

```
pscheduler task --tool tracepath trace --source 10.0.1.10 --dest 10.0.3.10
```

- `pscheduler`: command to interact with perfSONAR.
- `task`: pScheduler command to specify a measurement test.
- `--tool`: command to specify the tool.
- `tracepath`: tool used for the measurement.
- `trace`: test type.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: the destination node, in this case it is the perfSONAR3 node (*10.0.3.10*).



```
[root@perfSONAR1 /]# pscheduler task --tool tracepath trace --source 10.0.1.10 --dest 10.0.3.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/bc081a8e-21f3-4d34-bfb8-80d8933b7770
Running with tool 'tracepath'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/bc081a8e-21f3-4d34-bfb8-80d8933b7770/runs/bf97d390-5260-43b7-a643-8a554f0e3676
Starts 2023-12-29T17:00:09+00:00 (~2 seconds)
Ends   2023-12-29T17:01:50+00:00 (~100 seconds)
Waiting for result...

1       10.0.1.1 0.013 ms mtu 1500 bytes
2       10.1.0.2 30.118 ms mtu 1500 bytes
3       10.2.0.2 40.198 ms mtu 1500 bytes
4       10.0.3.10 40.256 ms mtu 1500 bytes host-unreachable

No further runs scheduled.
[root@perfSONAR1 /]#
```

Figure 19. Running a trace test using *tracepath.*

The first column shows the hop number, the second column shows the IP address or Domain name. The third column shows the RTT for the packet to reach that point and return to the source host. The last column shows the Maximum Transmission Unit (MTU) size.

5.3 Paris-traceroute

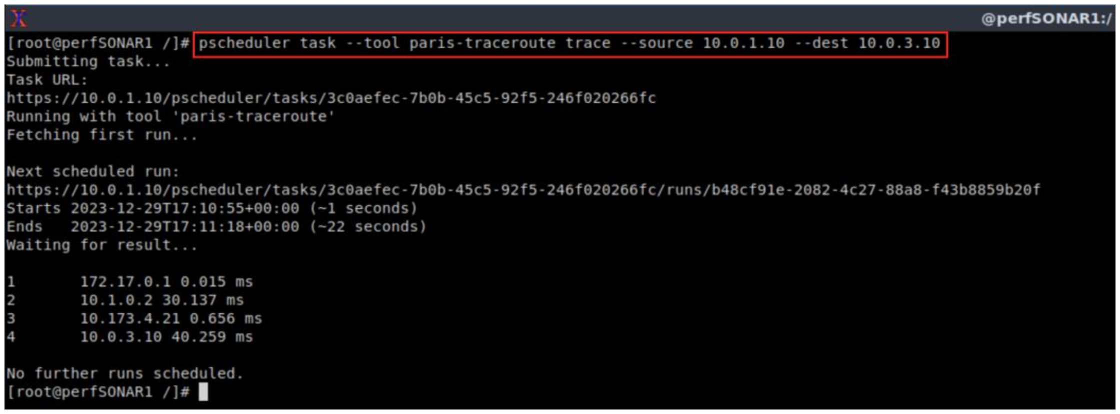
Paris-traceroute represents an enhanced version of the traceroute network diagnostic tool that specifically addresses issues related to load balancers. While the initial traceroute implementation may encounter challenges with load balancers, *Paris-traceroute* overcomes these obstacles.

By exerting control over the flow identifier of the probe packets, *Paris-traceroute* is capable of accurately following paths within networks that employ load balancers. It can identify and trace all the load-balanced paths leading to the destination, providing a comprehensive view of network routing.

Step 1. In perfSONAR1 issue the following command to run a tracepath test using *Paris-traceroute*. The nodes participating in this test are perfSONAR1 and perfSONAR3.

```
pscheduler task --tool paris-traceroute trace --source 10.0.1.10 --dest
10.0.3.10
```

- `pscheduler`: command to interact with perfSONAR.
- `task`: pScheduler command to specify a measurement test.
- `--tool`: command to specify the tool.
- `tracepath`: tool used for the measurement.
- `trace`: test type.
- `--source`: specify where the test should originate, in this case it is perfSONAR1 node (*10.0.1.10*).
- `--dest`: the destination node, in this case it is the perfSONAR3 node (*10.0.3.10*).



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task --tool paris-traceroute trace --source 10.0.1.10 --dest 10.0.3.10
Submitting task...
Task URL:
https://10.0.1.10/pscheduler/tasks/3c0aefec-7b0b-45c5-92f5-246f020266fc
Running with tool 'paris-traceroute'
Fetching first run...

Next scheduled run:
https://10.0.1.10/pscheduler/tasks/3c0aefec-7b0b-45c5-92f5-246f020266fc/runs/b48cf91e-2082-4c27-88a8-f43b8859b20f
Starts 2023-12-29T17:10:55+00:00 (~1 seconds)
Ends   2023-12-29T17:11:18+00:00 (~22 seconds)
Waiting for result...

1       172.17.0.1 0.015 ms
2       10.1.0.2 30.137 ms
3       10.173.4.21 0.656 ms
4       10.0.3.10 40.259 ms

No further runs scheduled.
[root@perfSONAR1 /]#
```

Figure 20. Running a trace test using *Paris-traceroute.*

Once the task is submitted, the user can observe that the selected tool is *Paris-traceroute*. The results obtained from the *Paris-traceroute* test can be interpreted in a similar manner to the traceroute test. The report consists of multiple rows divided into columns, representing the hops along the network route. To facilitate analysis, the results are reordered and presented in a table format.

Each row within the table corresponds to a specific hop in the route. The table consists of five columns, providing relevant information for each hop:

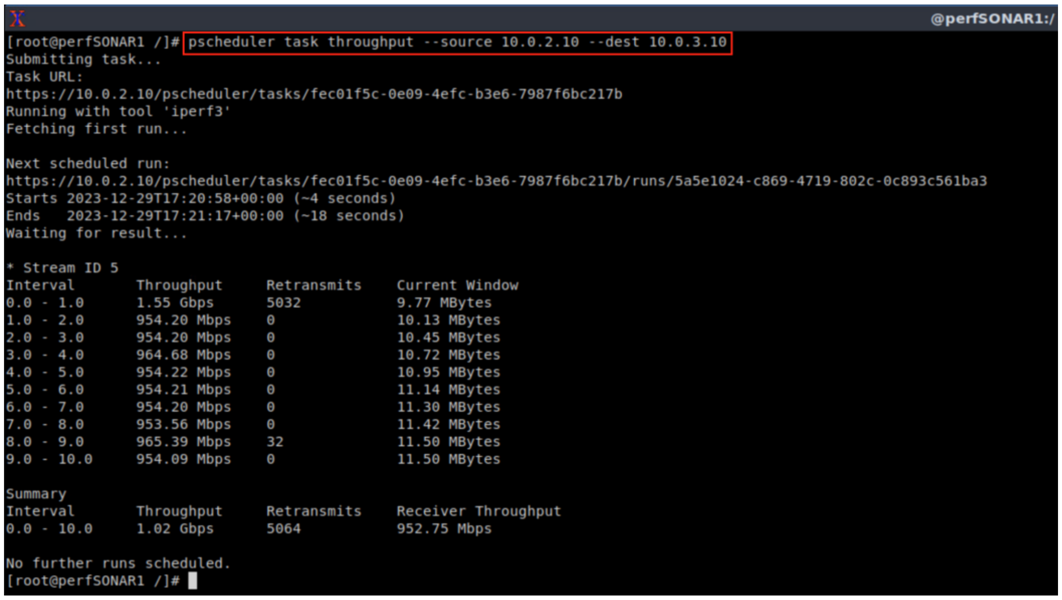
- Hop number (first column): This column denotes the number of hops encountered along the route. In this case, it takes five hops to reach the destination, and each hop is assigned a unique number.
- IP address (second column): The second column displays the IP address of either the destination or the router from the previous hop. If available, the corresponding domain name may also be listed, providing additional context.
- RTT (third column): The subsequent three columns represent the RTT for the packet to reach the specific hop and return to the source host. These RTT values are measured in milliseconds. Since *Paris-traceroute* sends three separate signal packets, there are three RTT columns. This allows for the evaluation of consistency, or the lack thereof, in the route by comparing the RTT values across the three packets.

6 Running tests on remote perfSONAR nodes

Step 1. In perfSONAR1 issue the following command to run a throughput test between perfSONAR2 and perfSONAR3.

```
pscheduler task throughput --source 10.0.2.10 --dest 10.0.3.10
```

- `pscheduler`**:** is the command to interact with perfSONAR.
- `task`**:** is a pScheduler command to specify a measurement test.
- `throughput`: specifies the test.
- `--source`: is to specify where the test should originate, in this case it is perfSONAR2 node (*10.0.2.10*).
- `--dest`: is the destination node, in this case is the perfSONAR3 node (*10.0.3.10*).



```
[root@perfSONAR1 /]# pscheduler task throughput --source 10.0.2.10 --dest 10.0.3.10
Submitting task...
Task URL:
https://10.0.2.10/pscheduler/tasks/fec01f5c-0e09-4efc-b3e6-7987f6bc217b
Running with tool 'iperf3'
Fetching first run...

Next scheduled run:
https://10.0.2.10/pscheduler/tasks/fec01f5c-0e09-4efc-b3e6-7987f6bc217b/runs/5a5e1024-c869-4719-802c-0c893c561ba3
Starts 2023-12-29T17:20:58+00:00 (~4 seconds)
Ends   2023-12-29T17:21:17+00:00 (~18 seconds)
Waiting for result...

* Stream ID 5
Interval       Throughput     Retransmits    Current Window
0.0 - 1.0      1.55 Gbps      5032           9.77 MBytes
1.0 - 2.0      954.20 Mbps    0              10.13 MBytes
2.0 - 3.0      954.20 Mbps    0              10.45 MBytes
3.0 - 4.0      964.68 Mbps    0              10.72 MBytes
4.0 - 5.0      954.22 Mbps    0              10.95 MBytes
5.0 - 6.0      954.21 Mbps    0              11.14 MBytes
6.0 - 7.0      954.20 Mbps    0              11.30 MBytes
7.0 - 8.0      953.56 Mbps    0              11.42 MBytes
8.0 - 9.0      965.39 Mbps    32             11.50 MBytes
9.0 - 10.0     954.09 Mbps    0              11.50 MBytes

Summary
Interval       Throughput     Retransmits    Receiver Throughput
0.0 - 10.0     1.02 Gbps      5064           952.75 Mbps

No further runs scheduled.
[root@perfSONAR1 /]# 
```

Figure 21. Running a throughput test between two remote perfSONAR nodes.

7 Repeating tasks

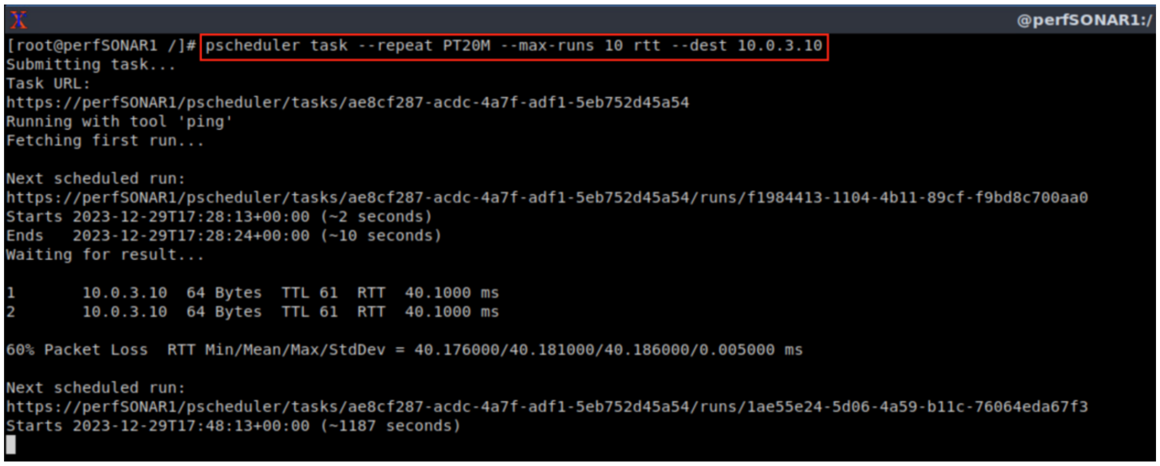
A task can be configured to run periodically. In this section, it is shown step by step how to repeat throughput and RTT tasks using pScheduler command. First the user will configure pScheduler to run a throughput task every 30 seconds. Then, the user will run an RTT task every 45 seconds. Any pScheduler task can be configured to run repeatedly by adding options to the task command:

- `--start TIMESTAMP`: it runs the first iteration of the task at `TIMESTAMP`.
- `--repeat DURATION`: Repeat runs at intervals of `DURATION`.
- `--max-runs N`: Allow the task to run up to N times.
- `--until TIMESTAMP`: Repeat runs of the task until `TIMESTAMP`.
- `--slip DURATION`: Allow the start of each run to be as much as `DURATION` later than their ideal scheduled time. If the environment variable `PSCHEDULER_SLIP` is present, its value will be used as a default. Failing that, the default will be PT5M. Notice that the slip value also applies to non-repeating tasks.
- `--sliprand`: Randomly choose a timeslot within the allowed slip instead of choosing earliest available.

Step 1. In perfSONAR1 command line, follow the command format (1) and type:

```
pscheduler task --repeat PT20M --max-runs 10 rtt --dest 10.0.3.10
```

- `pscheduler`: is the command to interact with perfSONAR.
- `task`: is a pScheduler command to specify a measurement test.
- `--repeat PT20M`: is a pScheduler command that configures the task to be repeated every 20 minutes.
- `--max-runs 10`: Allow the task to run up to 10 times.
- `rtt`: is the test type.
- `--dest` is the destination node, in this case it is perfSONAR3 node (*10.0.3.10*).



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task --repeat PT20M --max-runs 10 rtt --dest 10.0.3.10
Submitting task...
Task URL:
https://perfSONAR1/pscheduler/tasks/ae8cf287-acdc-4a7f-adf1-5eb752d45a54
Running with tool 'ping'
Fetching first run...

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/ae8cf287-acdc-4a7f-adf1-5eb752d45a54/runs/f1984413-1104-4b11-89cf-f9bd8c700aa0
Starts 2023-12-29T17:28:13+00:00 (~2 seconds)
Ends   2023-12-29T17:28:24+00:00 (~10 seconds)
Waiting for result...

1       10.0.3.10  64 Bytes  TTL 61  RTT  40.1000 ms
2       10.0.3.10  64 Bytes  TTL 61  RTT  40.1000 ms

60% Packet Loss  RTT Min/Mean/Max/StdDev = 40.176000/40.181000/40.186000/0.005000 ms

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/ae8cf287-acdc-4a7f-adf1-5eb752d45a54/runs/1ae55e24-5d06-4a59-b11c-76064eda67f3
Starts 2023-12-29T17:48:13+00:00 (~1187 seconds)
```

Figure 22. Repeating an RTT test between perfSONAR1 and perfSONAR3 every 20 minutes.

Notice that the source node is not explicit, this means that the source node is perfSONAR1 (*10.0.1.10*), i.e., the node where the command is executed.

The figure above shows the first measurement of the round-trip time. Notice that the task is going to be repeated 10 times in 20 minutes.

Step 2. To return to the CLI, press `Ctrl+C`. Notice that the task will keep running on the background.

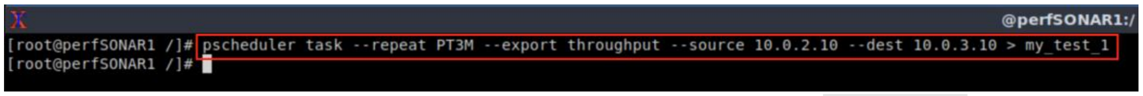
8 Exporting and importing tasks

During the usage of pScheduler, users have the capability to export a pScheduler task into a Java Script Object Notation (JSON) file. This export feature allows users to save a task specification in a structured and machine-readable format. The JSON representation of a task can be generated and displayed on the standard output without scheduling the task by utilizing the `--export` command.

Step 1. In perfSONAR1 command line, issue the following command to export a pScheduler task.

```
pscheduler task --repeat PT10M --export throughput --source 10.0.2.10 --dest
10.0.3.10 > my_test_1
```

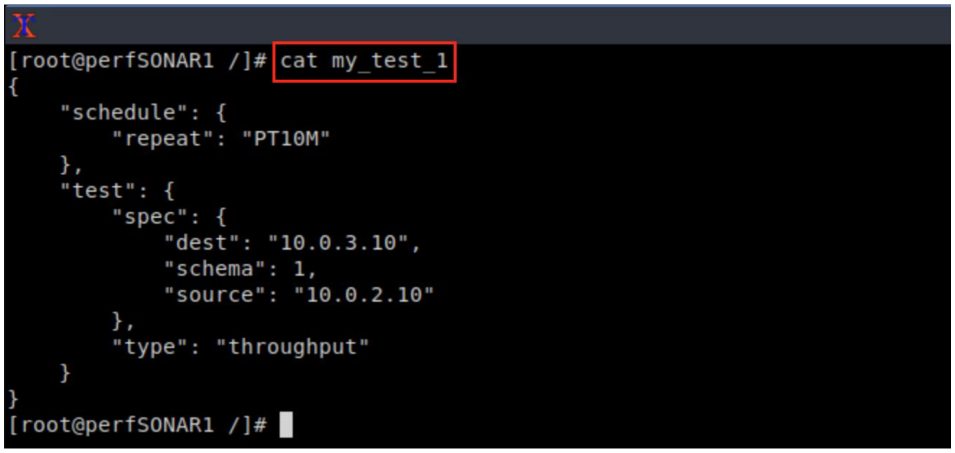
- `pscheduler`: is the command to interact with perfSONAR.
- `task`: is a pScheduler command to specify a measurement test.
- `--repeat PT10M`: is a pScheduler command that configures the task to be repeated every 10 minutes.
- `--export`: is to indicate that the task will not be executed but stored.
- `throughput`: is the test type.
- `--source`: is to specify where the test should originate, in this case it is perfSONAR2 node (*10.0.2.10*).
- `--dest` is the destination node, in this case it is perfSONAR3 node (*10.0.3.10*).
- `> my_task_1`: is to create a file where the task is going to be stored.



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task --repeat PT3M --export throughput --source 10.0.2.10 --dest 10.0.3.10 > my_test_1
[root@perfSONAR1 /]#
```

Figure 23. Exporting a throughput test through the file `my_test_1`.

Step 2. In order to visualize the file, type `cat my_test_1`. A JSON file will be displayed. This file contents a pScheduler task, however this task is not running. Notice also that the task might be invalid because tasks are not validated until they are submitted for scheduling.



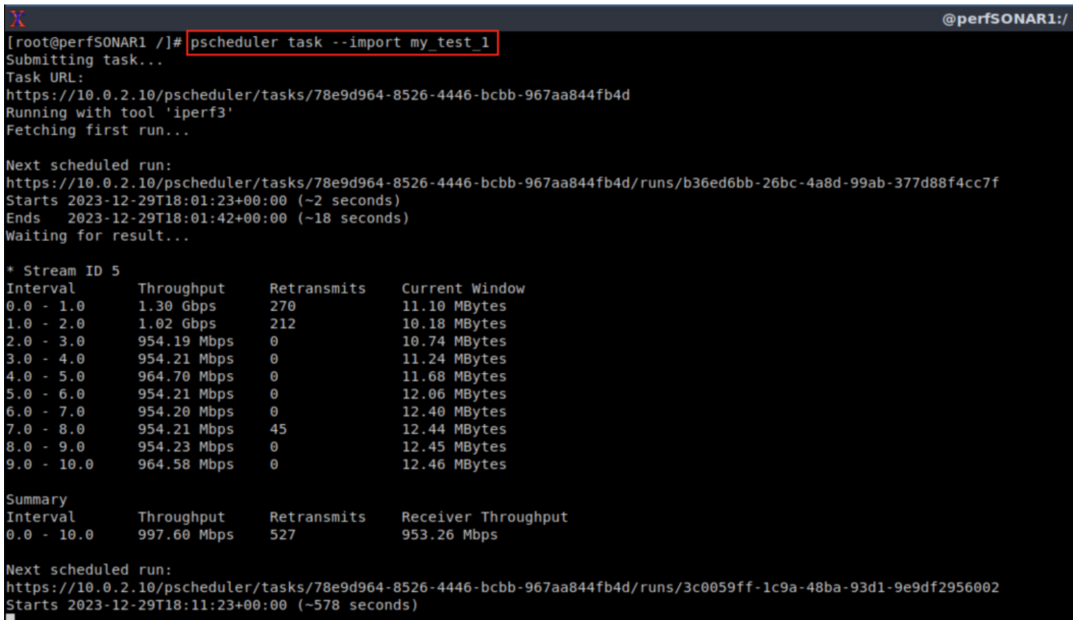
```
[root@perfSONAR1 /]# cat my_test_1
{
    "schedule": {
        "repeat": "PT10M"
    },
    "test": {
        "spec": {
            "dest": "10.0.3.10",
            "schema": 1,
            "source": "10.0.2.10"
        },
        "type": "throughput"
    }
}
[root@perfSONAR1 /]#
```

Figure 24. Visualizing the content of the exported file.

Step 3. A JSON file that was previously exported or generated can be imported using the `--import` command. In perfSONAR1 command line, issue the following command to import the task called `my_test_1`.

```
pscheduler task --import my_test_1
```

- `pscheduler`: is the command to interact with perfSONAR.
- `task`: is a pScheduler command to specify a measurement test.
- `--import`: specified the operation over the file.
- `my_test_1`: is the file that contains the task.



```
[root@perfSONAR1 /]# pscheduler task --import my_test_1
Submitting task...
Task URL:
https://10.0.2.10/pscheduler/tasks/78e9d964-8526-4446-bcbb-967aa844fb4d
Running with tool 'iperf3'
Fetching first run...

Next scheduled run:
https://10.0.2.10/pscheduler/tasks/78e9d964-8526-4446-bcbb-967aa844fb4d/runs/b36ed6bb-26bc-4a8d-99ab-377d88f4cc7f
Starts 2023-12-29T18:01:23+00:00 (~2 seconds)
Ends   2023-12-29T18:01:42+00:00 (~18 seconds)
Waiting for result...

* Stream ID 5
Interval       Throughput     Retransmits    Current Window
0.0 - 1.0      1.30 Gbps      270            11.10 MBytes
1.0 - 2.0      1.02 Gbps      212            10.18 MBytes
2.0 - 3.0      954.19 Mbps    0              10.74 MBytes
3.0 - 4.0      954.21 Mbps    0              11.24 MBytes
4.0 - 5.0      964.70 Mbps    0              11.68 MBytes
5.0 - 6.0      954.21 Mbps    0              12.06 MBytes
6.0 - 7.0      954.20 Mbps    0              12.40 MBytes
7.0 - 8.0      954.21 Mbps    45             12.44 MBytes
8.0 - 9.0      954.23 Mbps    0              12.45 MBytes
9.0 - 10.0     964.58 Mbps    0              12.46 MBytes

Summary
Interval       Throughput     Retransmits    Receiver Throughput
0.0 - 10.0     997.60 Mbps    527            953.26 Mbps

Next scheduled run:
https://10.0.2.10/pscheduler/tasks/78e9d964-8526-4446-bcbb-967aa844fb4d/runs/3c0059ff-1c9a-48ba-93d1-9e9df2956002
Starts 2023-12-29T18:11:23+00:00 (~578 seconds)
```

Figure 25. Importing a throughput test.

Step 4. To return to the CLI, press `Ctrl+C`. Notice that the task will keep running on the background.

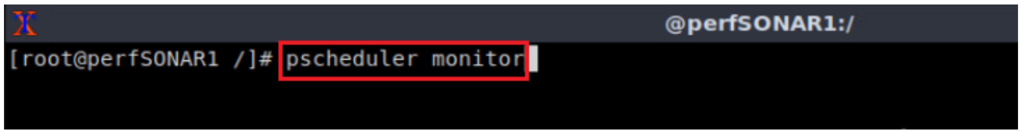
9 Displaying the schedule

In this section, the user will be introduced to two visualization tools: pScheduler Monitor and pScheduler Schedule. These tools are designed to provide users with insights into the scheduling and progress of tasks within the pScheduler framework. It is important to note that the tests scheduled in the previous lab section are still ongoing.

9.1 pScheduler monitor

Step 1. In perfSONAR1 command line, issue the following command to monitor the tasks.

```
pscheduler monitor
```



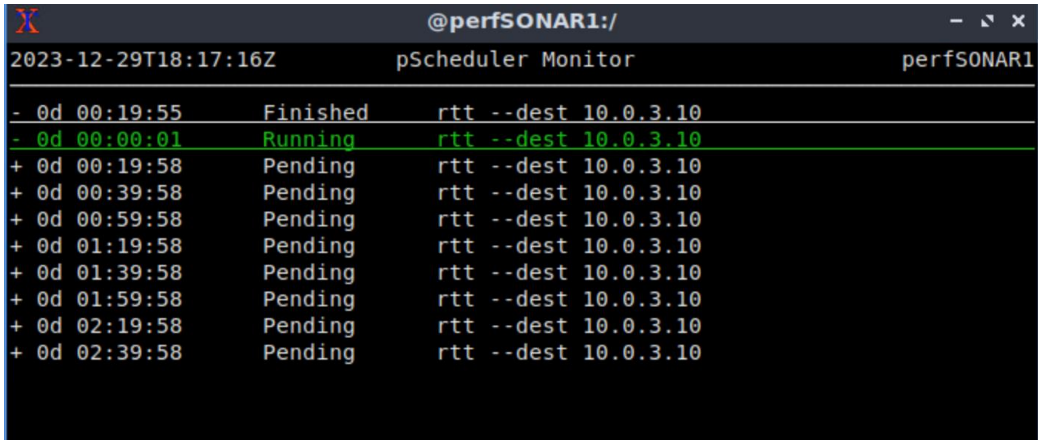
```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler monitor
```

Figure 26. Running the pScheduler monitor.

When viewing the scheduled tests using pScheduler commands and visualization tools, participants will encounter various status values that indicate the state of each test. These status values provide valuable information about the progress and outcome of the scheduled tests. The possible status values and their meanings are as follows:

- Pending: This status indicates that the test run is scheduled to be executed in the future but has not yet started.
- On Deck: When a test is in the "On Deck" status, it means that the execution of the test is about to begin imminently.
- Running: The "Running" status indicates that the test is currently in progress and actively being executed.
- Cleanup: After a test run is completed, it enters the "Cleanup" status. During this phase, any necessary final operations or clean-up tasks associated with the test are being performed.
- Finished: When a test run has been successfully executed and completed, it is assigned the "Finished" status.
- Overdue: If a test run was scheduled to start at a specific time in the past but did not commence, it is marked as "Overdue." The scheduler may still attempt to execute the test if it falls within a certain threshold.
- Missed: The "Missed" status is assigned to a test run that was scheduled but did not execute at its designated time. This can occur if the scheduler was not operational during the scheduled time or if the task was paused.
- Failed: If a test run fails to complete for any reason, it is labeled as "Failed." This status indicates that the test encountered an issue or encountered an error during execution.
- Non-Starter: When a test run cannot be scheduled due to constraints or the unavailability of suitable timeslots, it is categorized as "Non-Starter."

- Canceled: If a test is canceled before it is executed, it is assigned the "Canceled" status. This status indicates that the test will not be run as intended.



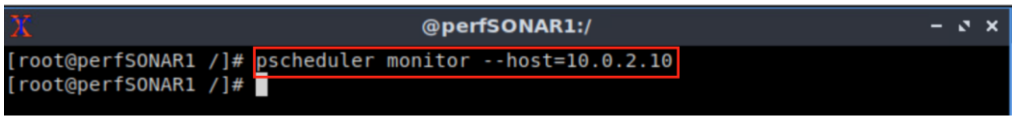
```
@perfSONAR1:/
2023-12-29T18:17:16Z          pScheduler Monitor                    perfSONAR1

- 0d 00:19:55      Finished      rtt --dest 10.0.3.10
- 0d 00:00:01      Running       rtt --dest 10.0.3.10
+ 0d 00:19:58      Pending       rtt --dest 10.0.3.10
+ 0d 00:39:58      Pending       rtt --dest 10.0.3.10
+ 0d 00:59:58      Pending       rtt --dest 10.0.3.10
+ 0d 01:19:58      Pending       rtt --dest 10.0.3.10
+ 0d 01:39:58      Pending       rtt --dest 10.0.3.10
+ 0d 01:59:58      Pending       rtt --dest 10.0.3.10
+ 0d 02:19:58      Pending       rtt --dest 10.0.3.10
+ 0d 02:39:58      Pending       rtt --dest 10.0.3.10
```

Figure 27. Visualizing the scheduled tasks in perfSONAR1.

Step 2. To exit from pScheduler monitor, press `Ctrl+C`.

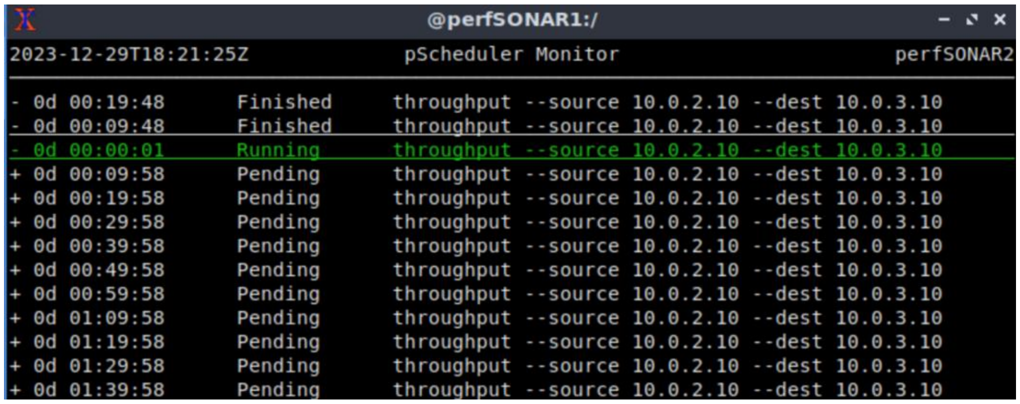
Step 3. In perfSONAR1, issue the following command to display the tasks in perfSONAR2.

```
pscheduler monitor --host=10.0.2.10
```



```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler monitor --host=10.0.2.10
[root@perfSONAR1 /]#
```

Figure 28. Running the pScheduler monitor.



```
@perfSONAR1:/
2023-12-29T18:21:25Z          pScheduler Monitor                    perfSONAR2

- 0d 00:19:48     Finished     throughput --source 10.0.2.10 --dest 10.0.3.10
- 0d 00:09:48     Finished     throughput --source 10.0.2.10 --dest 10.0.3.10
- 0d 00:00:01     Running      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 00:09:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 00:19:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 00:29:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 00:39:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 00:49:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 00:59:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 01:09:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 01:19:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 01:29:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
+ 0d 01:39:58     Pending      throughput --source 10.0.2.10 --dest 10.0.3.10
```

Figure 29. Visualizing the scheduled tasks in perfSONAR2 from perfSONAR1.

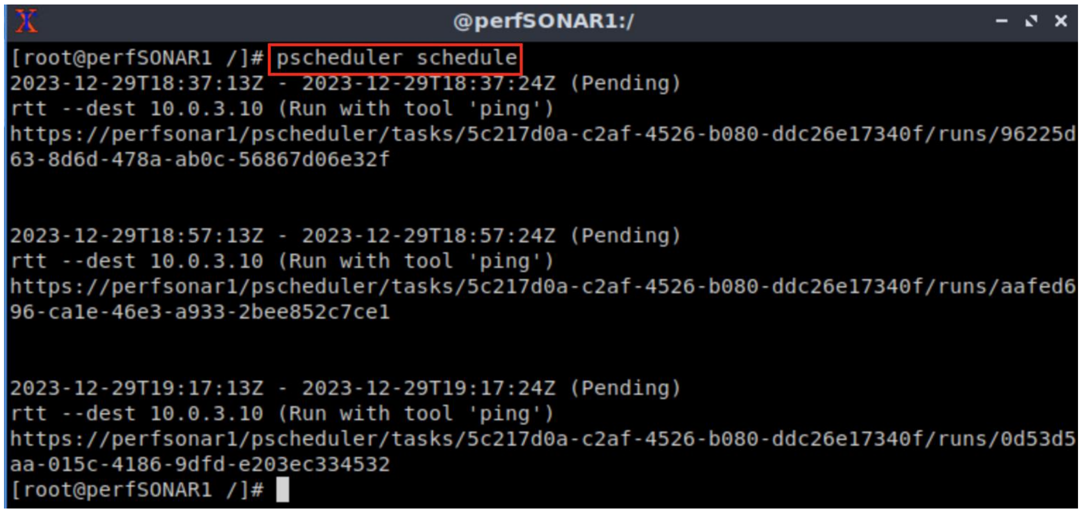
Step 4. To exit from pScheduler monitor, press `Ctrl+C`.

9.2 pScheduler schedule

The `pscheduler schedule` command asks pScheduler to fetch scheduled task runs from the past, present or future and display them as text.

Step 1. In perfSONAR1 command line, issue the following command to visualize the test schedule.

```
pscheduler schedule
```



```
[root@perfSONAR1 /]# pscheduler schedule
2023-12-29T18:37:13Z - 2023-12-29T18:37:24Z (Pending)
rtt --dest 10.0.3.10 (Run with tool 'ping')
https://perfsonar1/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f/runs/96225d
63-8d6d-478a-ab0c-56867d06e32f


2023-12-29T18:57:13Z - 2023-12-29T18:57:24Z (Pending)
rtt --dest 10.0.3.10 (Run with tool 'ping')
https://perfsonar1/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f/runs/aafed6
96-ca1e-46e3-a933-2bee852c7ce1


2023-12-29T19:17:13Z - 2023-12-29T19:17:24Z (Pending)
rtt --dest 10.0.3.10 (Run with tool 'ping')
https://perfsonar1/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f/runs/0d53d5
aa-015c-4186-9dfd-e203ec334532
[root@perfSONAR1 /]#
```

Figure 30. Visualizing the tests schedule in perfSONAR1.

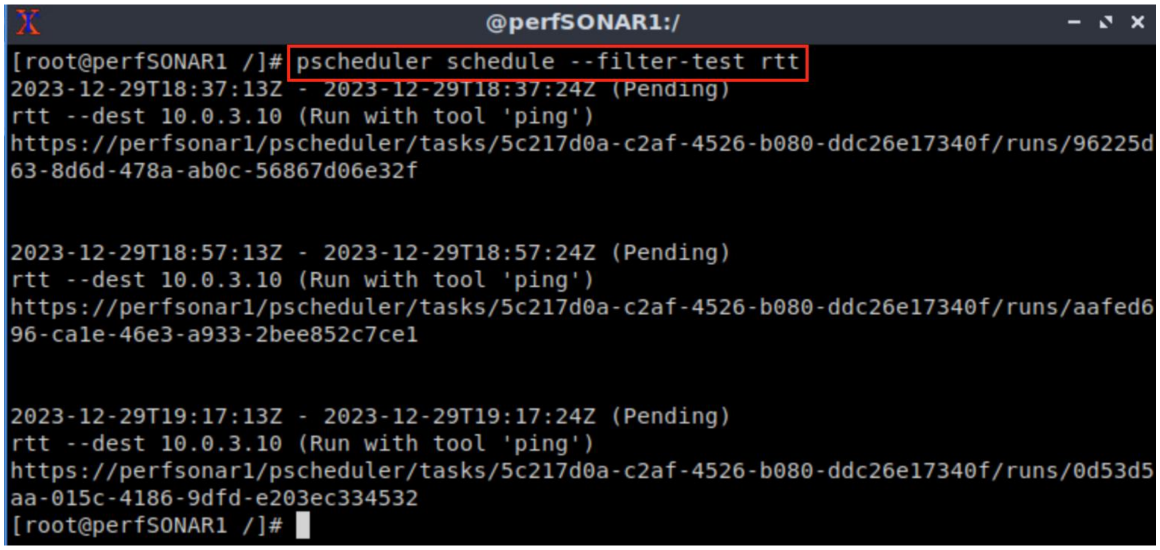
Step 2. To exit from pScheduler schedule, press `Ctrl+C`.

10 Canceling tasks

So far there are two pScheduler tasks running. In this section, the user will cancel the scheduled Round-Trip Time (RTT) and throughput tasks which are running.

Step 1. In perfSONAR1 command line, issue the following command to export a pScheduler task.

```
pscheduler schedule --filter-test rtt
```



```
[root@perfSONAR1 /]# pscheduler schedule --filter-test rtt
2023-12-29T18:37:13Z - 2023-12-29T18:37:24Z (Pending)
rtt --dest 10.0.3.10 (Run with tool 'ping')
https://perfsonar1/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f/runs/96225d
63-8d6d-478a-ab0c-56867d06e32f


2023-12-29T18:57:13Z - 2023-12-29T18:57:24Z (Pending)
rtt --dest 10.0.3.10 (Run with tool 'ping')
https://perfsonar1/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f/runs/aafed6
96-ca1e-46e3-a933-2bee852c7ce1


2023-12-29T19:17:13Z - 2023-12-29T19:17:24Z (Pending)
rtt --dest 10.0.3.10 (Run with tool 'ping')
https://perfsonar1/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f/runs/0d53d5
aa-015c-4186-9dfd-e203ec334532
[root@perfSONAR1 /]#
```

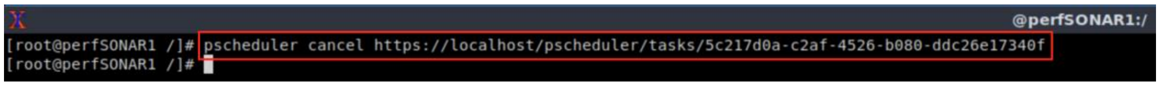
Figure 31. Filtering out RTT tests.

The user will see the scheduled task for Round-Trip Time (RTT) measurement.

Step 2. In perfSONAR1 command line, issue the following command to cancel a task.

```
pscheduler cancel https://localhost/pscheduler/tasks/[url]
```

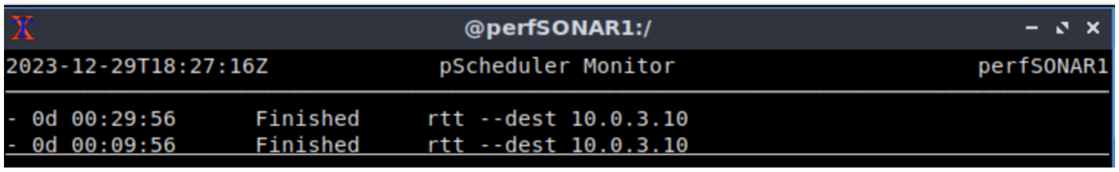
Replace `[url]` with the first three characters of the last task URL. In this example, the first three characters of the task are `5c21`, these characters may vary then, press `Tab` key to autocomplete the following characters. *Press Enter* to execute the command.



```
[root@perfSONAR1 /]# pscheduler cancel https://localhost/pscheduler/tasks/5c217d0a-c2af-4526-b080-ddc26e17340f
[root@perfSONAR1 /]#
```

Figure 32. Cancelling the RTT tasks.

Step 3. In perfSONAR1 command line, type the command `pscheduler monitor` to visualize the schedule.



```
2023-12-29T18:27:16Z          pScheduler Monitor                  perfSONAR1

- 0d 00:29:56      Finished     rtt --dest 10.0.3.10
- 0d 00:09:56      Finished     rtt --dest 10.0.3.10
```

Figure 33. Verifying the tasks in perfSONAR1.

Lab 5 has been completed. You may now end your reservation.

References

1. Mininet walkthrough, [Online]. Available: http://Mininet.org.
2. perfSONAR Project, "perfSONAR Installation Options," [Online]. Available: https://docs.perfsonar.net/install_options.html

3. perfSONAR Project. “What is perfSONAR?,” [Online]. Available: https://docs.perfsonar.net/intro_about.html
4. Docker Inc, “*What is a container,*” [Online]. Available: https://www.docker.com/resources/what-container/
5. Manuel Peuster, “*Containernet,*” [Online]. Available: https://containernet.github.io/
6. Brian Linkletter, “*How to use MiniEdit, Mininet’s graphical user interface*,”[Online]. Available: https://tinyurl.com/MiniEdit-PS
7. perfSONAR Project, “*perfSONAR Global NodeDirectory*”. Available: https://stats.es.net/ServicesDirectory/.

UNIVERSITY OF SOUTH CAROLINA

P4-PERFSONAR LAB SERIES

Lab 6: Connecting perfSONAR to Grafana Dashboard

Document Version: 06-19-2023

Contents

Overview 3
Objectives 3
Lab settings 3
Lab roadmap 3
1 Introduction 3
1.1 Lab scenario 4
2 Loading and running the lab topology 4
2.1 Setting IPv4 addresses, static routes, and link conditions 6
3 Running regular tests with pScheduler 6
3.1 Accessing perfSONAR1 CLI 6
3.2 Running RTT, throughput and trace tests 8
4 Configuring the communication between perfSONAR1 and Grafana 10
4.1 Running Grafana 10
4.2 Accessing Grafana 11
4.3 Configuring Grafana's data source 12
4.4 Configuring an OpenSearch data source 14
5 Displaying the measurement results in a dashboard 20
5.1 Visualizing throughput results 20
5.2 Visualizing RTT results 25
5.3 Visualizing retransmissions 29
References 34

Overview

This lab explains how to integrate Grafana with perfSONAR to display measurement results. Grafana is an open-source visualization platform for building dashboards. It supports multiple data sources, including the OpenSearch database, which is the default database implemented in perfSONAR. This lab shows how to configure a Grafana dashboard to show the results produced by a perfSONAR node.

Objectives

By the end of this lab, the user will:

1. Store pScheduler measurements in a local OpenSearch database.
2. Configure Grafana to access OpenSearch database.
3. Construct a Grafana dashboard to visualize pScheduler measurements.
4. Select the metrics to display in the Grafana dashboard.

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: Loading and running the lab topology.
3. Section 3: Running regular tests with pScheduler.
4. Section 4: Configuring the communication between perfSONAR1 and Grafana.
5. Section 5: Displaying the measurement results in a dashboard.

1 Introduction

Grafana[1] is an open-source visualization platform used to transform complex datasets into visually organized and insightful dashboards. It serves as a solution that enables professionals across various domains, from IT specialists to data analysts, to gain a comprehensive understanding of their data. Through flexible integration with diverse databases, Grafana proves suitable for a wide range of applications that require system performance monitoring. Leveraging customizable visualization options, such as charts, graphs, gauges, and heatmaps, users can creatively and effectively represent their data. Additionally, Grafana's alerting and notification features ensure prompt responses to critical events and anomalies.

Grafana integrates with perfSONAR[2] by displaying the measurement results stored in a node's database. This database is implemented using OpenSearch[3]. OpenSearch is a powerful open-source search and analytics engine, adept at indexing, searching, and visualizing large volumes of data. Its versatile functionalities cater to various applications, providing efficient data exploration, real-time monitoring, and alerting capabilities. With seamless integration of diverse data sources, OpenSearch offers rapid data retrieval and intuitive visualization.

1.1 Lab scenario

In this lab, the user will conduct regular tests on a perfSONAR node and store the results in a local database. Subsequently, the user will establish communication between Grafana, and the OpenSearch database hosted on a perfSONAR node. The final step involves presenting throughput, Round-trip Time (RTT), and retransmission data in a Grafana dashboard through multiple timing graphs.

2 Loading and running the lab topology

In this section, the user will load and run the lab topology using MiniEdit[6], which is the graphical tool used to create topologies in Mininet.

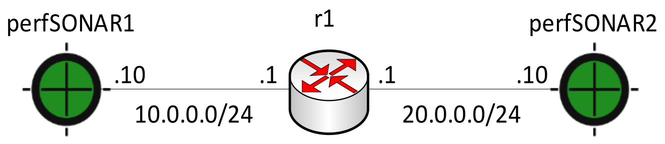


Figure 1. Lab topology.

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`.

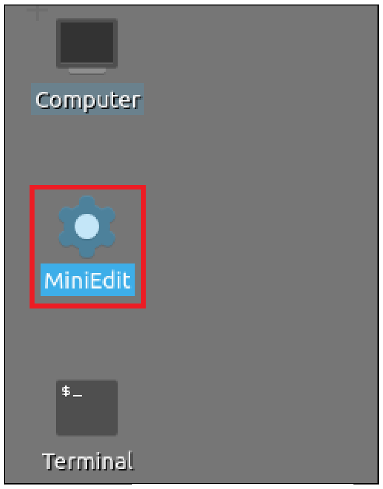
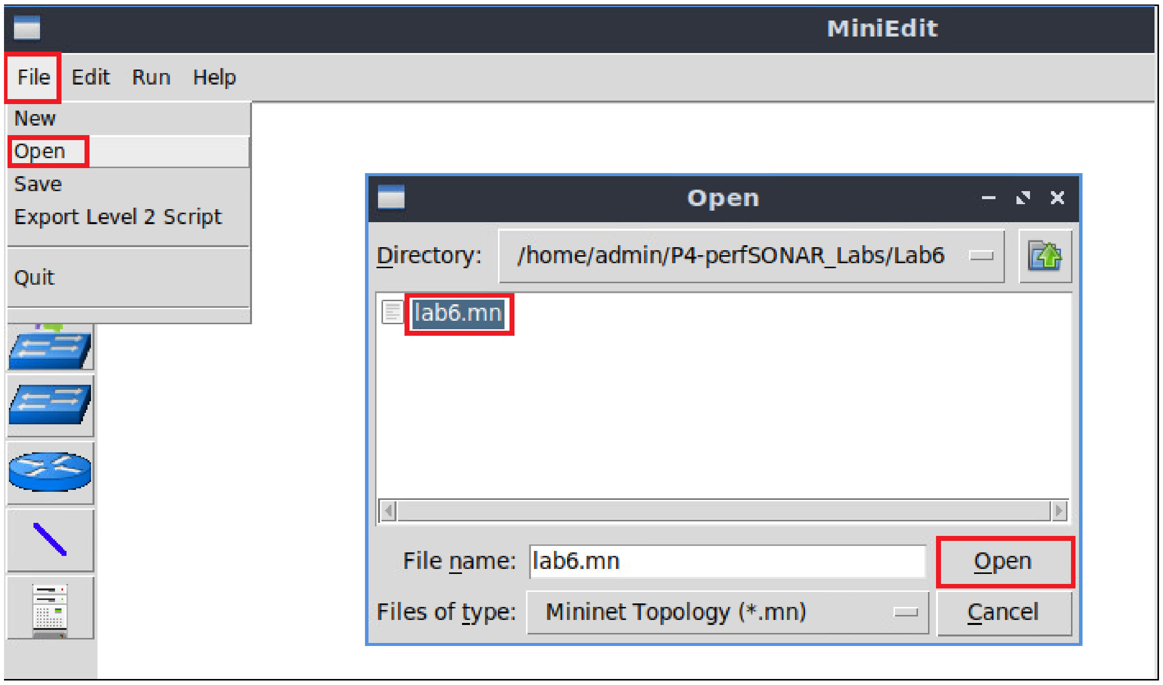


Figure 2. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the Lab6 folder and search for the topology file called *lab6.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4-perfSONAR_Labs/Lab6

lab6.mn

File name: lab6.mn

Files of type: Mininet Topology (*.mn)

Figure 3. 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.

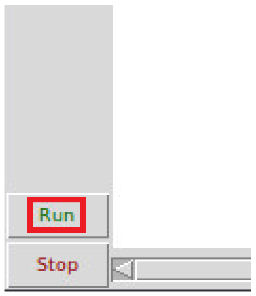


Figure 4. Running the emulation.

Wait for 10 seconds to allow the perfSONAR nodes to initialize within the network topology. Once the topology is fully operational, MiniEdit's toolbar on the left-hand side will become greyed out.

2.1 Setting IPv4 addresses, static routes, and link conditions

Step 1. Open a Linux terminal by clicking on the icon located on the taskbar.

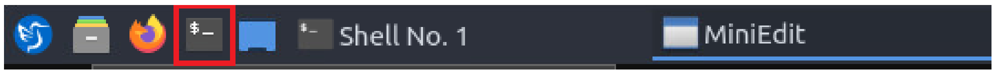
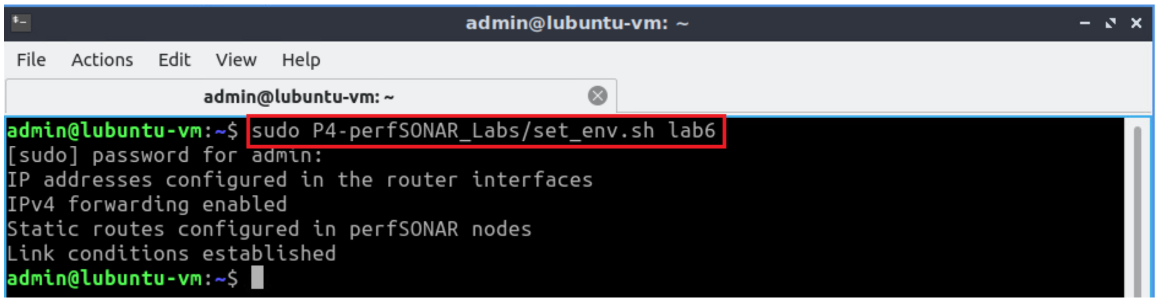


Figure 5. Opening a Linux terminal.

Step 2. To set up the lab environment, execute the following command in the Linux terminal. This script will configure the IPv4 addresses of the routers, define static routes, and establish specific link conditions, such as bandwidth limitations and minimum delays.

```
sudo P4-perfSONAR_Labs/set_env.sh lab6
```



```
admin@lubuntu-vm:~$ sudo P4-perfSONAR_Labs/set_env.sh lab6
[sudo] password for admin:
IP addresses configured in the router interfaces
IPv4 forwarding enabled
Static routes configured in perfSONAR nodes
Link conditions established
admin@lubuntu-vm:~$
```

Figure 6. Setting IPv4 addresses, static routes, links' bandwidths, and delays.

Keep the Linux CLI open, as it will be used in the upcoming sections.

3 Running regular tests with pScheduler

In this section, the user will run regular pScheduler tests from perfSONAR1 to perfSONAR2 and store the results in a local database.

3.1 Accessing perfSONAR1 CLI

Step 1. Go back to MiniEdit by clicking on the icon in the taskbar.

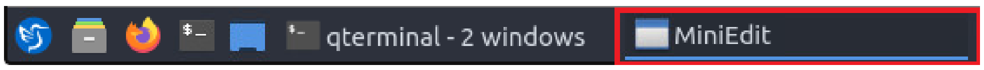


Figure 7. Opening MiniEdit.

Step 2. Hold the right click on perfSONAR1 and select *Terminal.* The perfSONAR1 CLI will emerge.

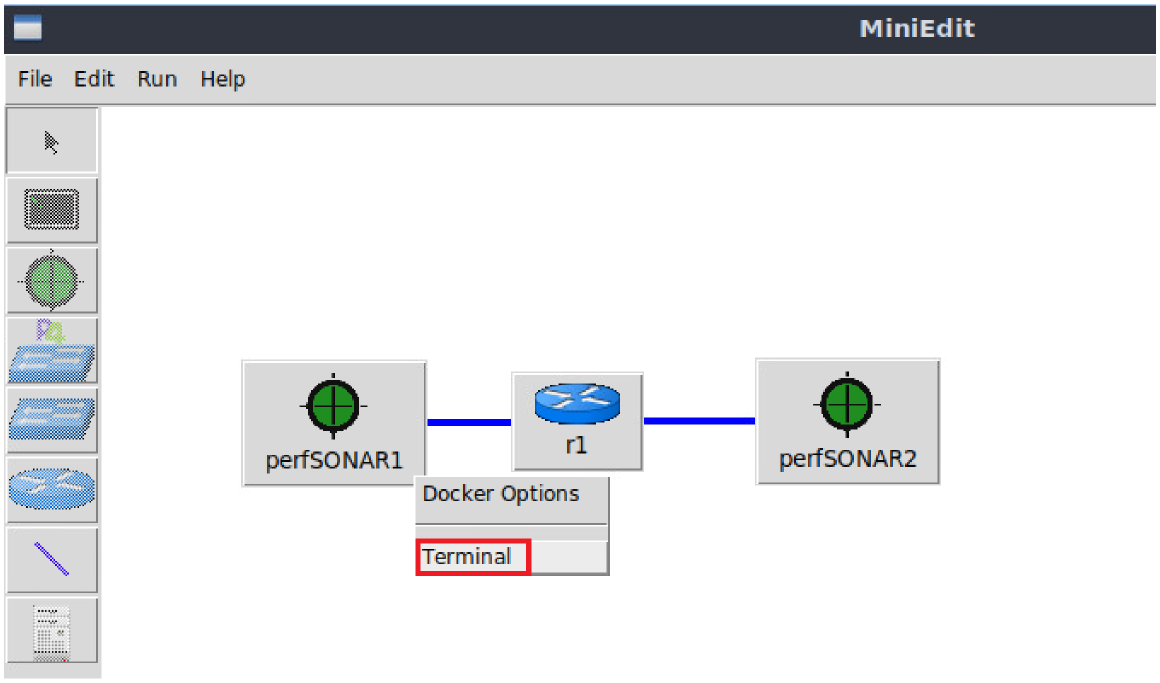
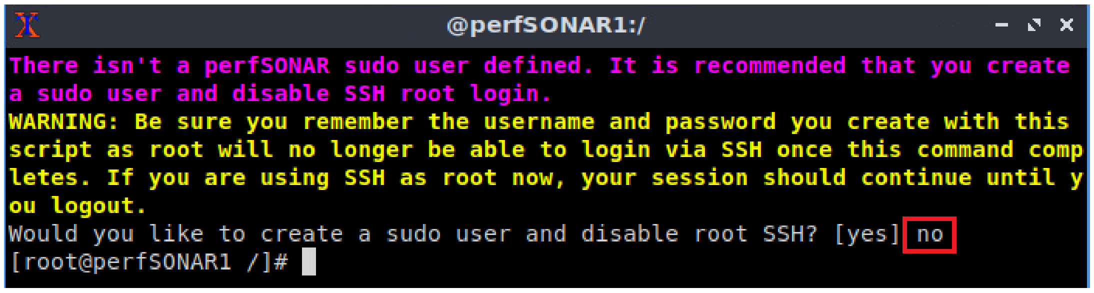


Figure 8. Opening perfSONAR1 CLI.

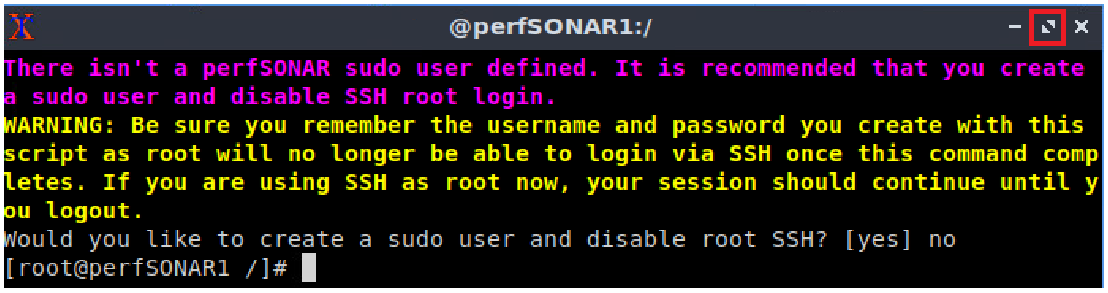
Step 3. In perfSONAR1 terminal, type `no` and press *Enter* to enable the CLI.



```
@perfSONAR1:/
There isn't a perfSONAR sudo user defined. It is recommended that you create
a sudo user and disable SSH root login.
WARNING: Be sure you remember the username and password you create with this
script as root will no longer be able to login via SSH once this command comp
letes. If you are using SSH as root now, your session should continue until y
ou logout.
Would you like to create a sudo user and disable root SSH? [yes] no
[root@perfSONAR1 /]#
```

Figure 9. Skipping initial configuration in perfSONAR1.

Step 4. Enlarge the terminal by clicking on the icon shown in the figure below.



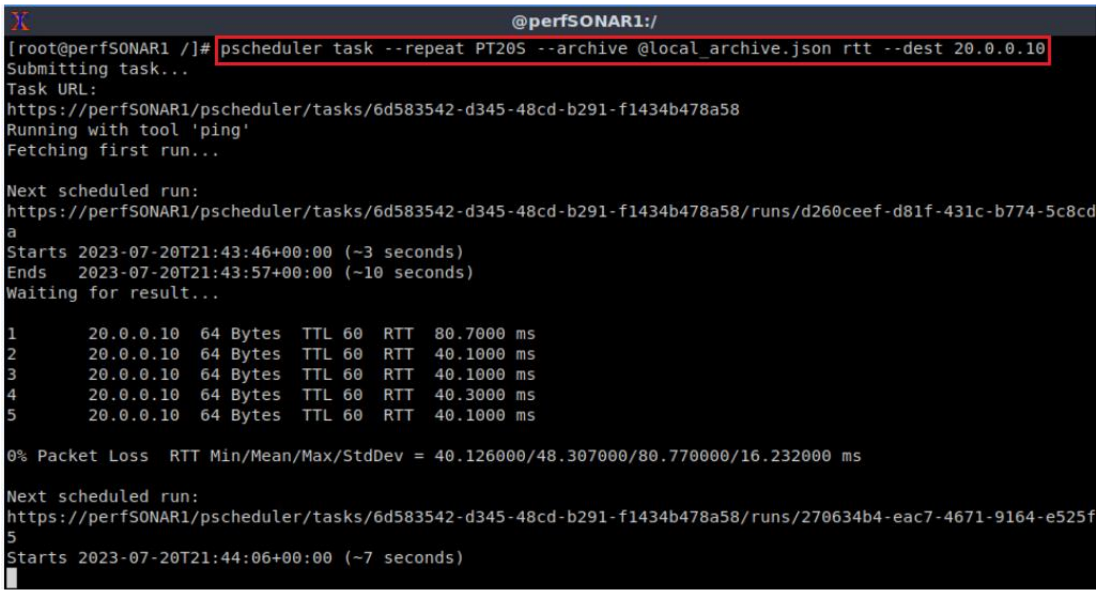
```
There isn't a perfSONAR sudo user defined. It is recommended that you create
a sudo user and disable SSH root login.
WARNING: Be sure you remember the username and password you create with this
script as root will no longer be able to login via SSH once this command comp
letes. If you are using SSH as root now, your session should continue until y
ou logout.
Would you like to create a sudo user and disable root SSH? [yes] no
[root@perfSONAR1 /]#
```

Figure 10. Enlarging perfSONAR1 CLI.

3.2 Running RTT, throughput and trace tests

Step 1. In perfSONAR1, issue the following command to run an RTT test every 20 seconds. Note that the results will be stored in the archive file called `local_archive.json`.

```
pscheduler task --repeat PT20S --archive @local_archive.json rtt --dest
20.0.0.10
```



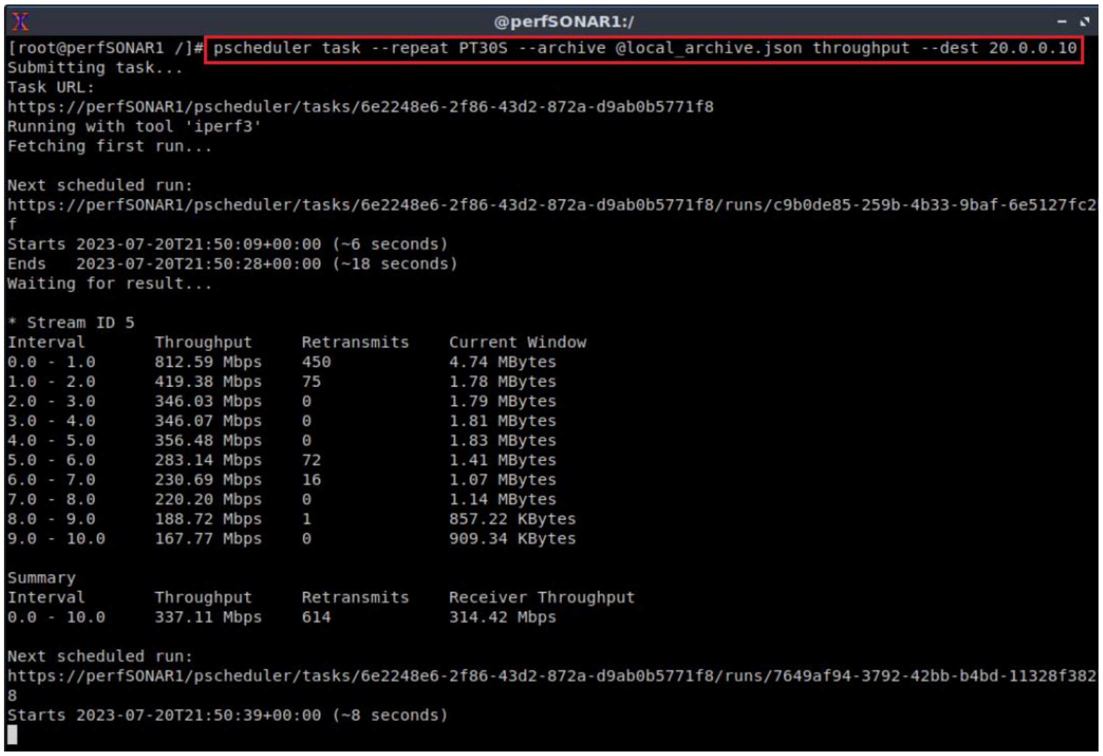
```
[root@perfSONAR1 /]# pscheduler task --repeat PT20S --archive @local_archive.json rtt --dest 20.0.0.10
Submitting task...
Task URL:
https://perfSONAR1/pscheduler/tasks/6d583542-d345-48cd-b291-f1434b478a58
Running with tool 'ping'
Fetching first run...

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/6d583542-d345-48cd-b291-f1434b478a58/runs/d260ceef-d81f-431c-b774-5c8cd
a
Starts 2023-07-20T21:43:46+00:00 (~3 seconds)
Ends   2023-07-20T21:43:57+00:00 (~10 seconds)
Waiting for result...

1       20.0.0.10  64 Bytes  TTL 60  RTT  80.7000 ms
2       20.0.0.10  64 Bytes  TTL 60  RTT  40.1000 ms
3       20.0.0.10  64 Bytes  TTL 60  RTT  40.1000 ms
4       20.0.0.10  64 Bytes  TTL 60  RTT  40.3000 ms
5       20.0.0.10  64 Bytes  TTL 60  RTT  40.1000 ms

0% Packet Loss  RTT Min/Mean/Max/StdDev = 40.126000/48.307000/80.770000/16.232000 ms

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/6d583542-d345-48cd-b291-f1434b478a58/runs/270634b4-eac7-4671-9164-e525f
5
Starts 2023-07-20T21:44:06+00:00 (~7 seconds)
```

Figure 11. Running an RTT test every 20 seconds.

Step 2. Press `Ctrl+C` to return to the CLI. The test will keep running in the background.

Step 3. In perfSONAR1, issue the following command to run a throughput test every 30 seconds. Note that the results will be stored in the archive file called `local_archive.json`.

```
pscheduler task --repeat PT30S --archive @local_archive.json throughput --dest
20.0.0.10
```



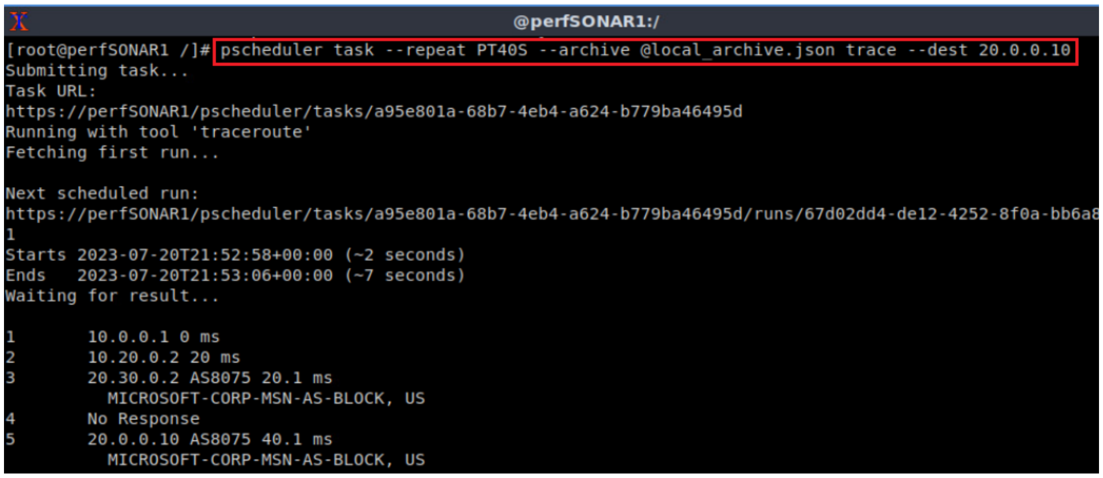
```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task --repeat PT30S --archive @local_archive.json throughput --dest 20.0.0.10
Submitting task...
Task URL:
https://perfSONAR1/pscheduler/tasks/6e2248e6-2f86-43d2-872a-d9ab0b5771f8
Running with tool 'iperf3'
Fetching first run...

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/6e2248e6-2f86-43d2-872a-d9ab0b5771f8/runs/c9b0de85-259b-4b33-9baf-6e5127fc2
f
Starts 2023-07-20T21:50:09+00:00 (~6 seconds)
Ends   2023-07-20T21:50:28+00:00 (~18 seconds)
Waiting for result...

* Stream ID 5
Interval       Throughput     Retransmits    Current Window
0.0 - 1.0      812.59 Mbps    450            4.74 MBytes
1.0 - 2.0      419.38 Mbps    75             1.78 MBytes
2.0 - 3.0      346.03 Mbps    0              1.79 MBytes
3.0 - 4.0      346.07 Mbps    0              1.81 MBytes
4.0 - 5.0      356.48 Mbps    0              1.83 MBytes
5.0 - 6.0      283.14 Mbps    72             1.41 MBytes
6.0 - 7.0      230.69 Mbps    16             1.07 MBytes
7.0 - 8.0      220.20 Mbps    0              1.14 MBytes
8.0 - 9.0      188.72 Mbps    1              857.22 KBytes
9.0 - 10.0     167.77 Mbps    0              909.34 KBytes

Summary
Interval       Throughput     Retransmits    Receiver Throughput
0.0 - 10.0     337.11 Mbps    614            314.42 Mbps

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/6e2248e6-2f86-43d2-872a-d9ab0b5771f8/runs/7649af94-3792-42bb-b4bd-11328f382
8
Starts 2023-07-20T21:50:39+00:00 (~8 seconds)
```

Figure 12. Running a throughput test every 30 seconds.

Step 4. Press `Ctrl+C` to return to the CLI. The test will keep running in the background.

Step 5. Similarly, issue the following command to run a trace test every 40 seconds. Note that the results will be stored in the archive file called `local_archive.json`.

```
pscheduler task --repeat PT40S --archive @local_archive.json trace --dest
20.0.0.10
```



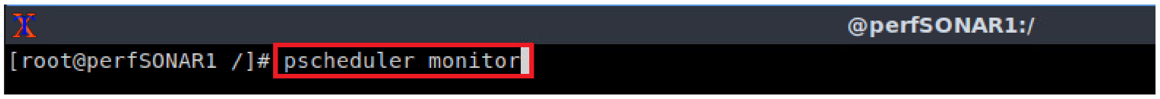
```
@perfSONAR1:/
[root@perfSONAR1 /]# pscheduler task --repeat PT40S --archive @local_archive.json trace --dest 20.0.0.10
Submitting task...
Task URL:
https://perfSONAR1/pscheduler/tasks/a95e801a-68b7-4eb4-a624-b779ba46495d
Running with tool 'traceroute'
Fetching first run...

Next scheduled run:
https://perfSONAR1/pscheduler/tasks/a95e801a-68b7-4eb4-a624-b779ba46495d/runs/67d02dd4-de12-4252-8f0a-bb6a8
1
Starts 2023-07-20T21:52:58+00:00 (~2 seconds)
Ends   2023-07-20T21:53:06+00:00 (~7 seconds)
Waiting for result...

1       10.0.0.1 0 ms
2       10.20.0.2 20 ms
3       20.30.0.2 AS8075 20.1 ms
          MICROSOFT-CORP-MSN-AS-BLOCK, US
4       No Response
5       20.0.0.10 AS8075 40.1 ms
          MICROSOFT-CORP-MSN-AS-BLOCK, US
```

Figure 13. Running a trace test every 40 seconds.

Step 6. Press `Ctrl+C` to return to the CLI. The test will keep running in the background.

Step 7. Verify that the RTT, throughput, and trace tests are scheduled correctly by issuing the following command.

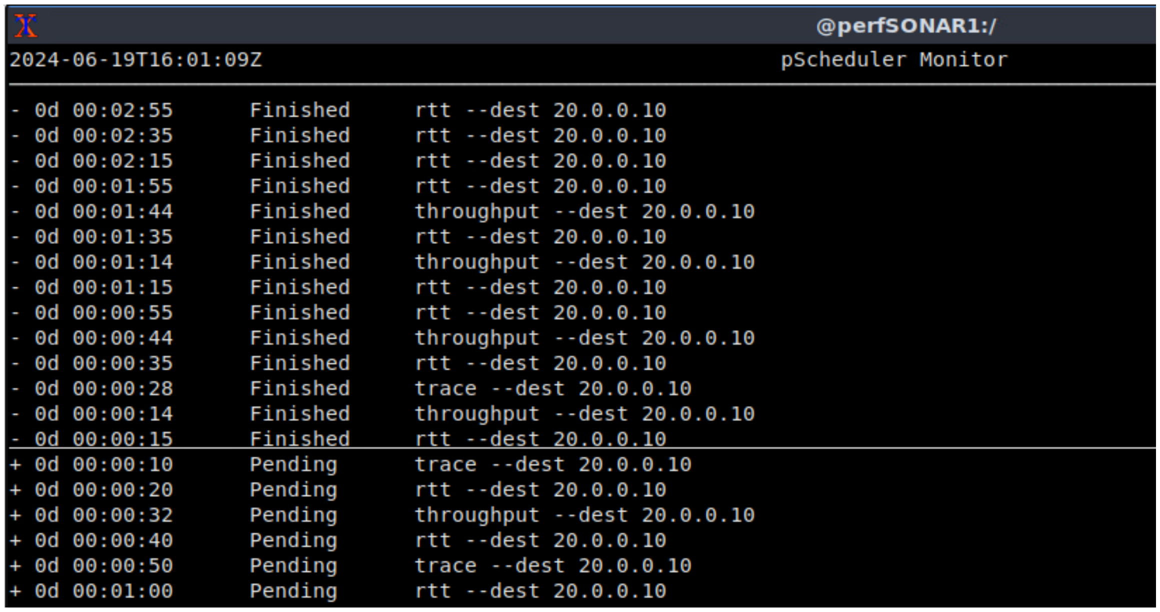
```
pscheduler monitor
```



```
[root@perfSONAR1 /]# pscheduler monitor
```

Figure 14. Running pScheduler monitor.

Step 8. Inspect the schedule.



```
2024-06-19T16:01:09Z                                   pScheduler Monitor

- 0d 00:02:55      Finished     rtt --dest 20.0.0.10
- 0d 00:02:35      Finished     rtt --dest 20.0.0.10
- 0d 00:02:15      Finished     rtt --dest 20.0.0.10
- 0d 00:01:55      Finished     rtt --dest 20.0.0.10
- 0d 00:01:44      Finished     throughput --dest 20.0.0.10
- 0d 00:01:35      Finished     rtt --dest 20.0.0.10
- 0d 00:01:14      Finished     throughput --dest 20.0.0.10
- 0d 00:01:15      Finished     rtt --dest 20.0.0.10
- 0d 00:00:55      Finished     rtt --dest 20.0.0.10
- 0d 00:00:44      Finished     throughput --dest 20.0.0.10
- 0d 00:00:35      Finished     rtt --dest 20.0.0.10
- 0d 00:00:28      Finished     trace --dest 20.0.0.10
- 0d 00:00:14      Finished     throughput --dest 20.0.0.10
- 0d 00:00:15      Finished     rtt --dest 20.0.0.10
+ 0d 00:00:10      Pending      trace --dest 20.0.0.10
+ 0d 00:00:20      Pending      rtt --dest 20.0.0.10
+ 0d 00:00:32      Pending      throughput --dest 20.0.0.10
+ 0d 00:00:40      Pending      rtt --dest 20.0.0.10
+ 0d 00:00:50      Pending      trace --dest 20.0.0.10
+ 0d 00:01:00      Pending      rtt --dest 20.0.0.10
```

Figure 15. Visualizing the schedule.

Step 8. Press q to return the CLI.

Note that the tests will keep running and the results of the measurement tests will be populated perfSONAR1's database.

4 Configuring the communication between perfSONAR1 and Grafana

In this section, the user will follow the process of configuring Grafana to access the measurement results stored in perfSONAR1. Grafana operates within a Docker container and can be accessed via a web interface. To enable this connection, the user will need to specify the data source, which, in this scenario, is the OpenSearch database hosting the pScheduler measurements. It's essential to note that the configured database is located in perfSONAR1.

4.1 Running Grafana

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

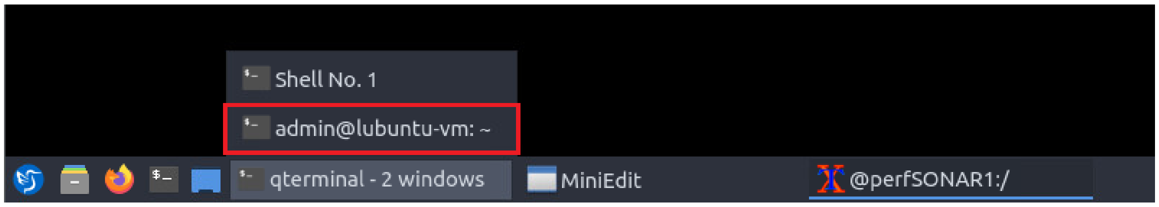
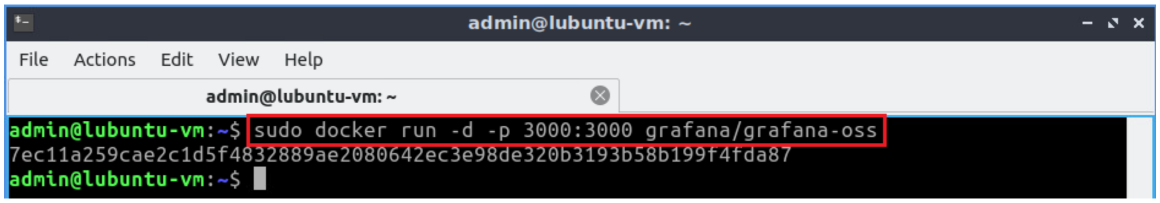


Figure 16. Returning to the Linux Terminal.

Step 2. In the Linux Terminal, issue the following command to start the Docker container that implements Grafana. If a password is required, type `password`.

```
sudo docker run -d -p 3000:3000 grafana/grafana-oss
```



```
admin@lubuntu-vm:~$ sudo docker run -d -p 3000:3000 grafana/grafana-oss
7ec11a259cae2c1d5f4832889ae2080642ec3e98de320b3193b58b199f4fda87
admin@lubuntu-vm:~$ 
```

Figure 17. Running Grafana.

4.2 Accessing Grafana

Step 1. Open the web browser by clicking on the icon located in the taskbar.

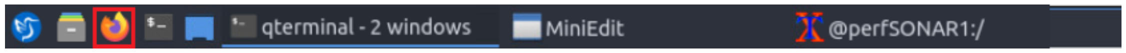


Figure 18. Opening the web browser.

Step 2. In the address bar type the following URL to access Grafana's web interface. An authentication webpage will be displayed.

```
localhost:3000
```

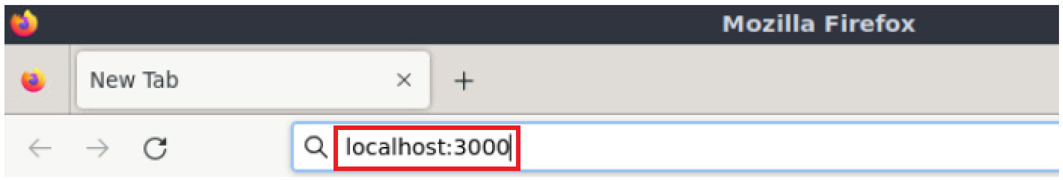


Figure 19. Accessing Grafana's web interface.

Step 3. Type `admin` as the username and `password` as the password. Then, click on *Log In*.

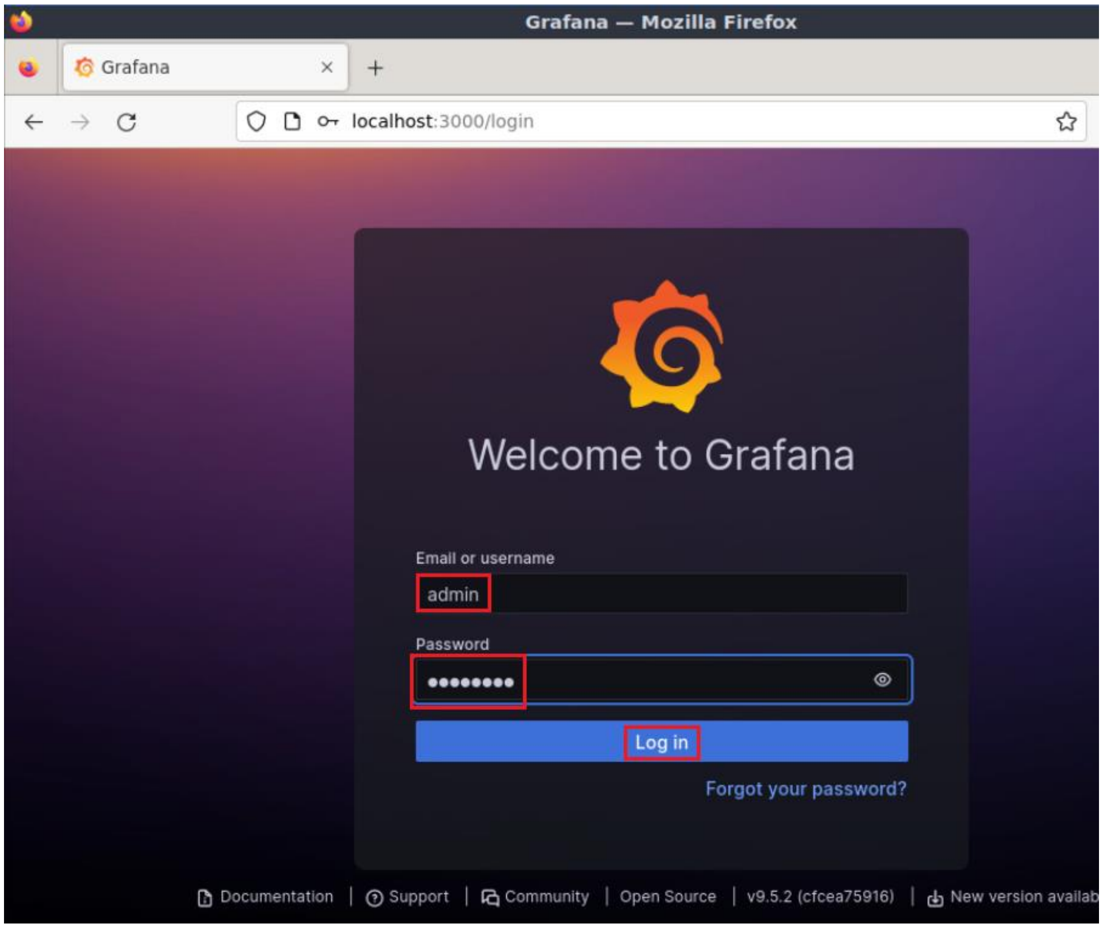

Grafana — Mozilla Firefox
Grafana
localhost:3000/login
Welcome to Grafana
Email or username
admin
Password
••••••••
Log in
Forgot your password?
Documentation | Support | Community | Open Source | v9.5.2 (cfcea75916) | New version availab

Figure 20. Authenticating the admin user in Grafana's web interface.

4.3 Configuring Grafana's data source

Step 1. Click on the three-line icon next to *Home* and select *Administration>Plugins* as shown in the figure below.

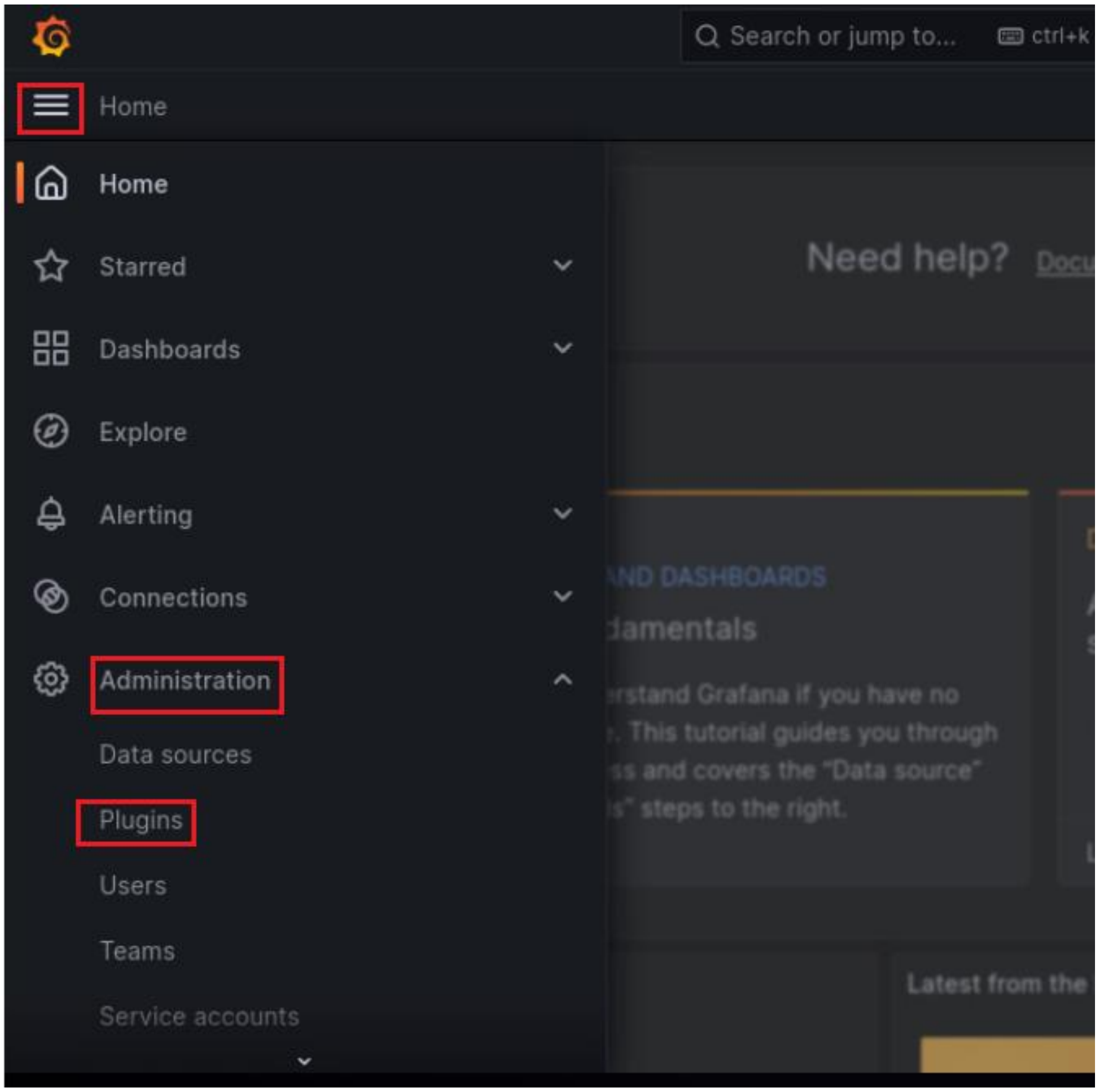


Figure 21. Configuring Grafana's data sources.

Step 2. In the search entry box, type *opensearch.* Then, click on the result as shown in the figure below.

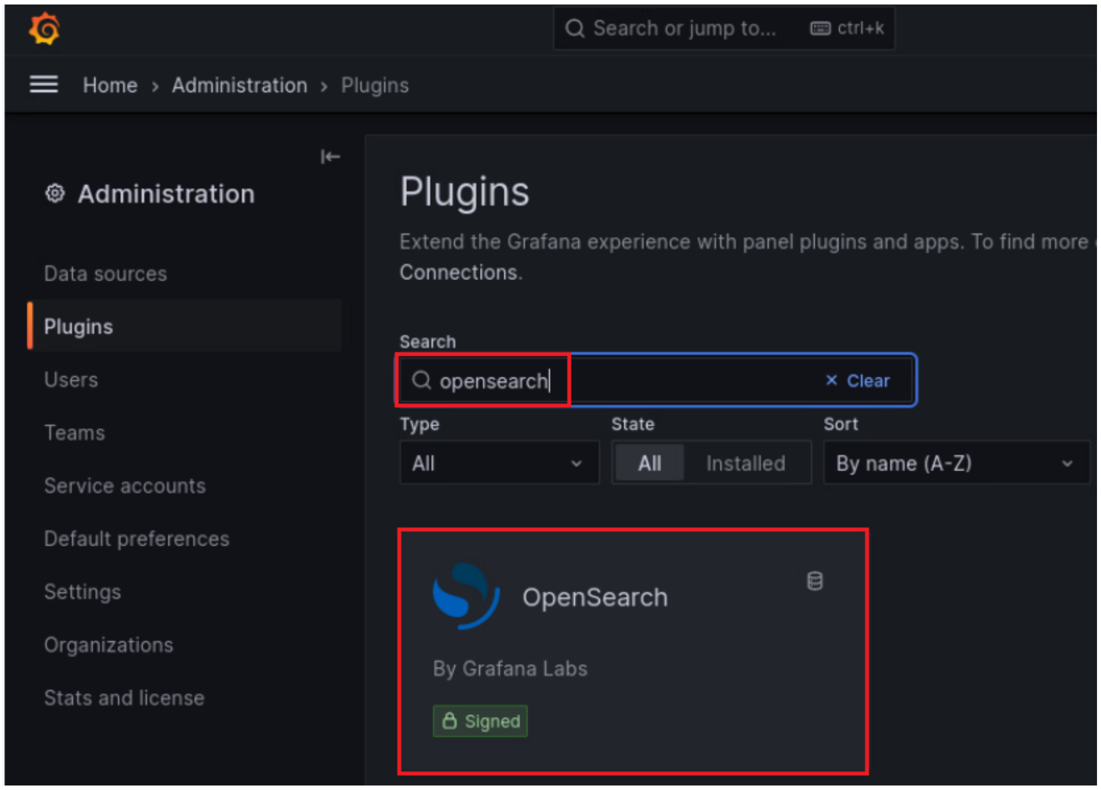


Figure 22. Searching for the OpenSearch plugin.

Step 3. Click on *Install* to install the plugin.

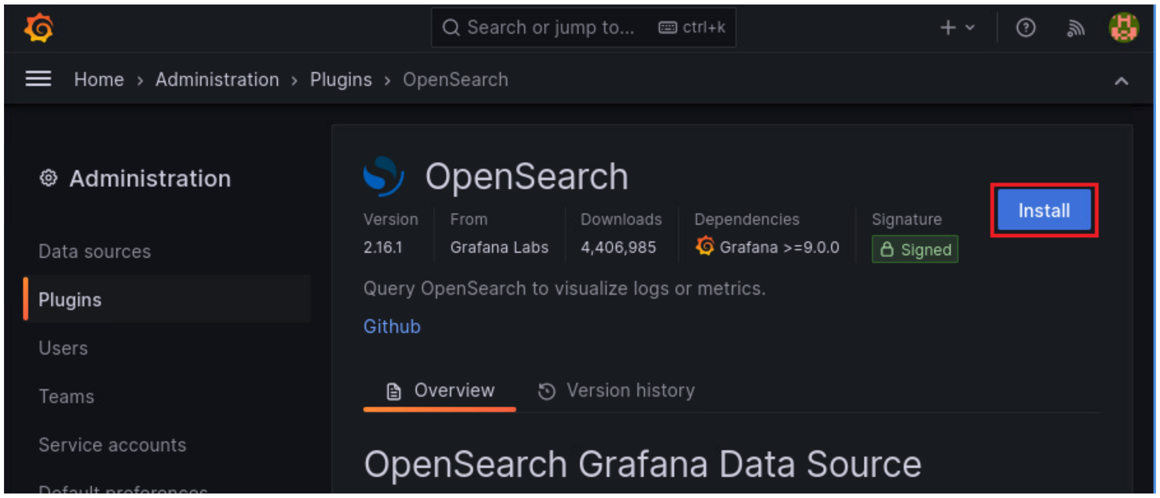


Figure 23. Installing the OpenSearch plugin.

4.4 Configuring an OpenSearch data source

Step 1. Once the installation is complete, click on *Create a OpenSearch data source.*

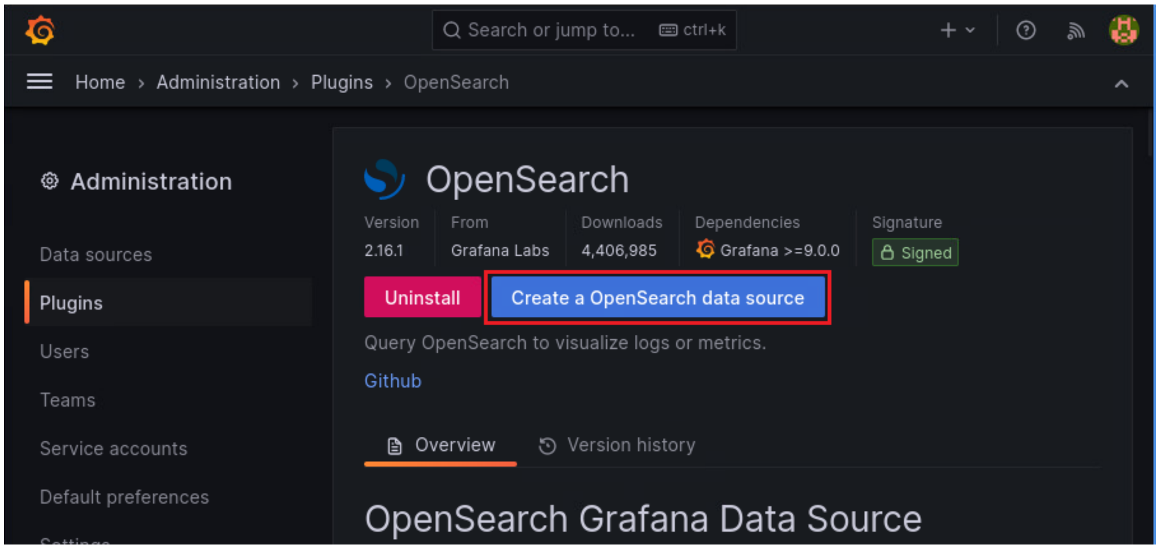
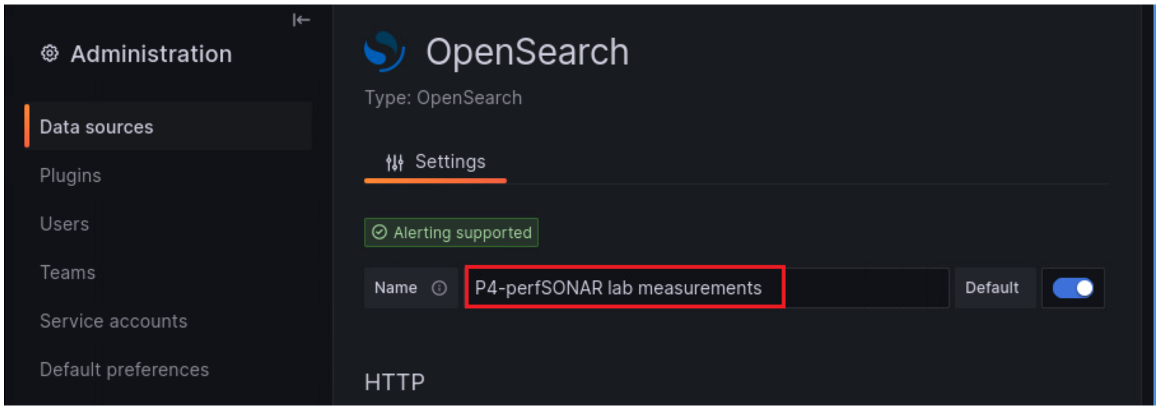


Figure 24. Creating an OpenSearch data source.

Step 2. Set the data source name as P4-*perfSONAR lab measurements* as shown in the figure below.

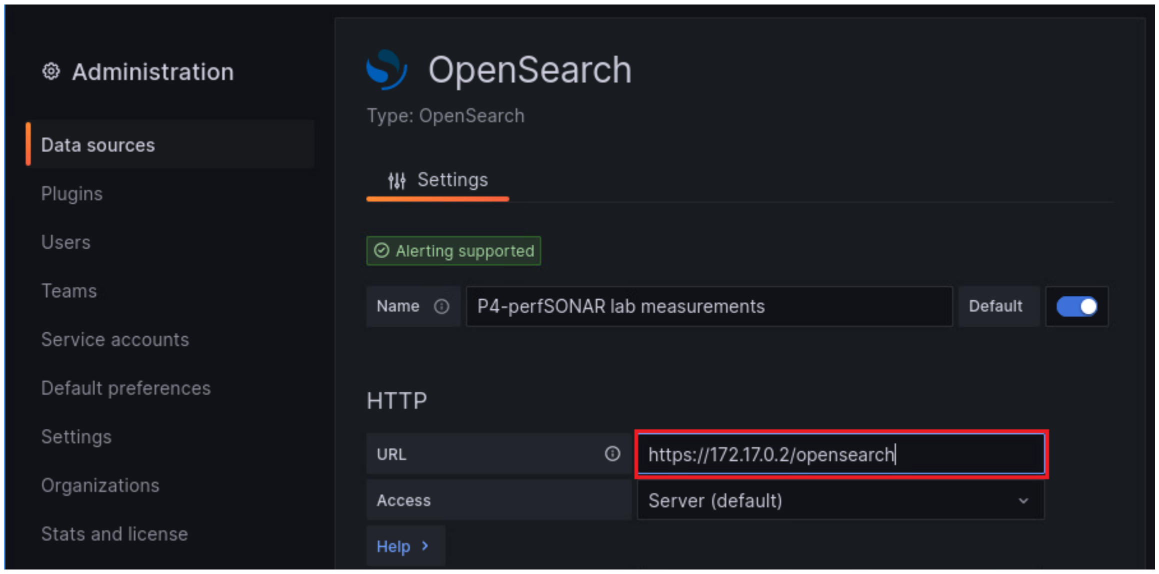


Name: P4-perfSONAR lab measurements
Default: on

Figure 25. Setting the data source name.

Step 3. Configure the following URL as shown in the figure below. This URL points towards the database of perfSONAR1.

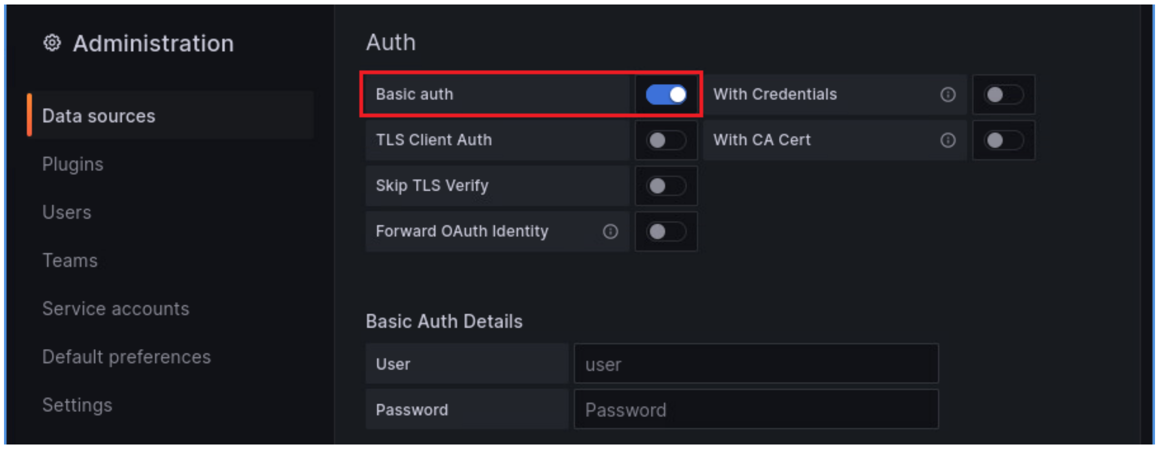
```
https://172.17.0.2/opensearch
```



Name: P4-perfSONAR lab measurements
Default: on
URL: https://172.17.0.2/opensearch
Access: Server (default)

Figure 26. Setting the data source URL.

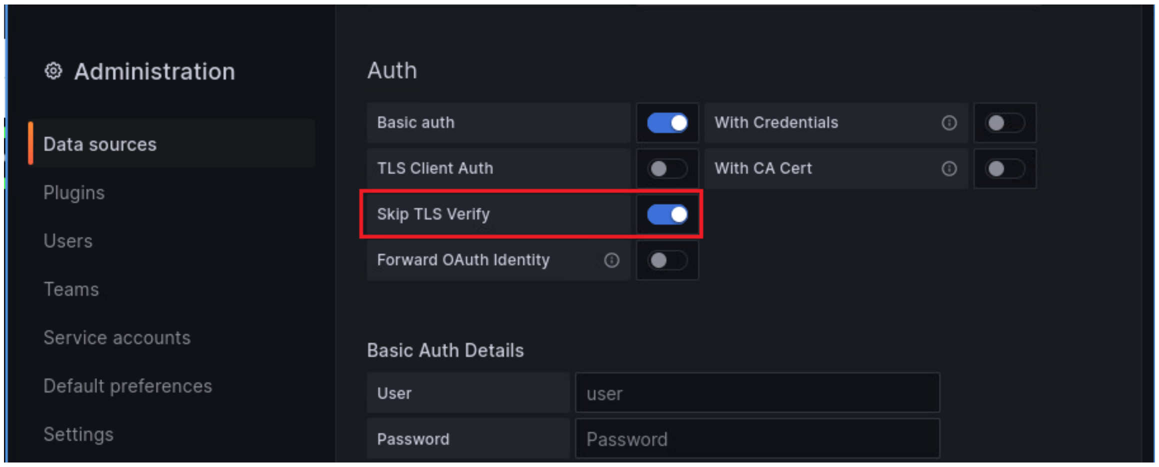
Step 4. Scroll down and enable the *Basic auth* option as shown in the figure below.



Basic auth: on
With Credentials: off
TLS Client Auth: off
With CA Cert: off
Skip TLS Verify: off
Forward OAuth Identity: off
User: user
Password: Password

Figure 27. Enabling basic authentication.

Step 5. In the *Auth* section, toggle on the *Skip TLS Verify* option.

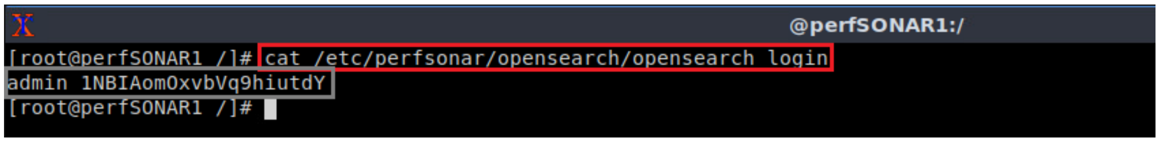


Basic auth: on
With Credentials: off
TLS Client Auth: off
With CA Cert: off
Skip TLS Verify: on
Forward OAuth Identity: off
User: user
Password: Password

Figure 28. Skipping TLS verification.

Step 6. Navigate to perfSONAR1 terminal and type the command below to display the Opensearch's authentication credentials.

```
cat /etc/perfsonar/opensearch/opensearch_login
```



```
[root@perfSONAR1 /]# cat /etc/perfsonar/opensearch/opensearch_login
admin 1NBIAomOxvbVq9hiutdY
[root@perfSONAR1 /]#
```

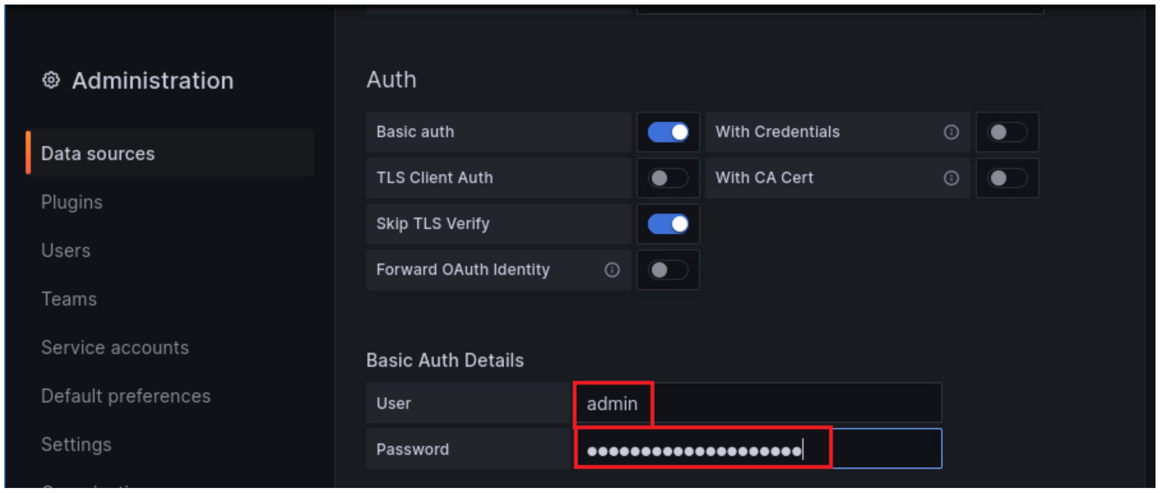
Figure 29. Displaying Opensearch's authentication credentials.

The figure above shows the credentials to access OpenSearch. The username is `admin` and the password is `1NBIAomOxvbVq9hiutdY`.

Step 7. In Grafana, input the credentials retrieved in the previous step as the username and password.

- User: `admin`
- Password: `1NBIAomOxvbVq9hiutdY`

Note that you can copy and paste this password from the file called *OpenSearch_password.txt* located on the Desktop.



Auth

Basic auth: on
With Credentials: off
TLS Client Auth: off
With CA Cert: off
Skip TLS Verify: on
Forward OAuth Identity: off

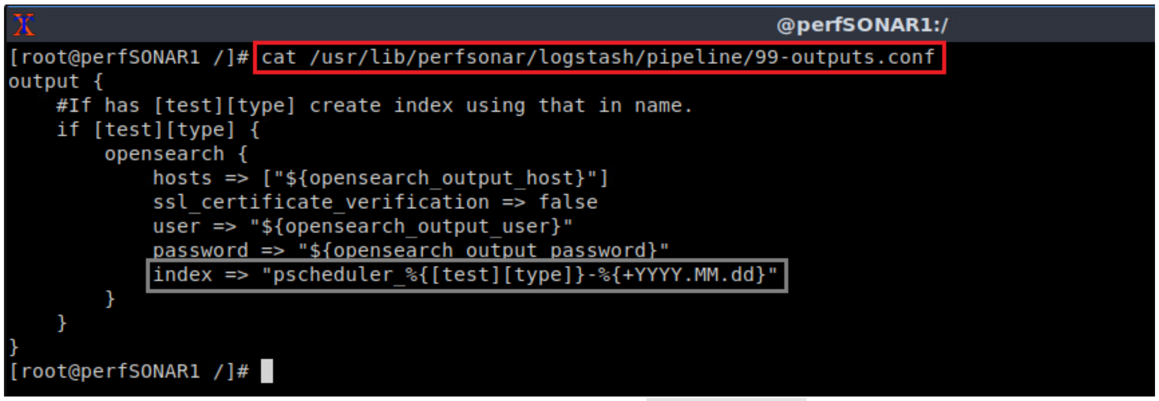
Basic Auth Details

User: admin
Password: ••••••••••••••••••••

Figure 30. Authenticating into the OpenSearch database.

Step 8. Navigate to perfSONAR1 terminal and type the command below to retrieve the *index* used by Logstash to store the data in OpenSearch.

```
cat /usr/lib/perfsonar/logstash/pipeline/99-outputs.conf
```



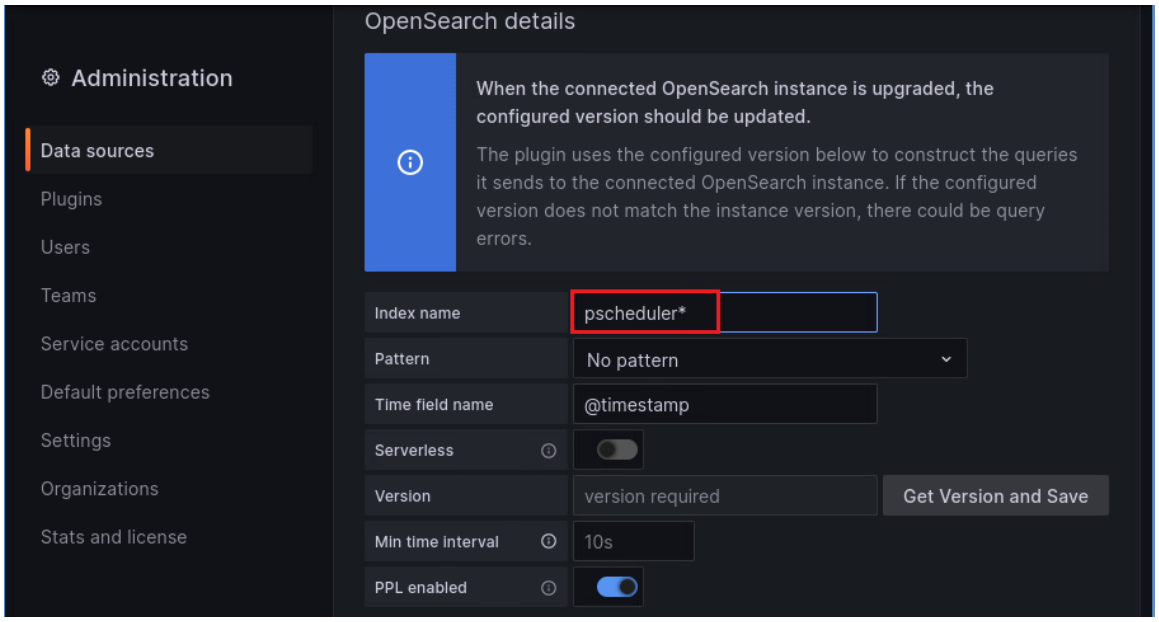
```
[root@perfSONAR1 /]# cat /usr/lib/perfsonar/logstash/pipeline/99-outputs.conf
output {
    #If has [test][type] create index using that in name.
    if [test][type] {
        opensearch {
            hosts => ["${opensearch_output_host}"]
            ssl_certificate_verification => false
            user => "${opensearch_output_user}"
            password => "${opensearch_output_password}"
            index => "pscheduler_%{[test][type]}-%{+YYYY.MM.dd}"
        }
    }
}
[root@perfSONAR1 /]#
```

Figure 31. Displaying the `index name`.

The index is the field used by OpenSearch to categorize the stored data. Querying the OpenSearch database requires specifying the index of the data.

The figure above shows that the index depends on the type of tests being performed and on the date the test is performed where `index = "pscheduler_%{[test][type]}-%{+YYYY.MM.dd}"`. Note that the index always starts with the keyword `pscheduler`.

Step 9. In Grafana, scroll down to the *OpenSearch details* section and type `pscheduler*` for the *index name* field. `pscheduler*` character implies that any index that starts with the keyword `pscheduler` should be considered.



When the connected OpenSearch instance is upgraded, the configured version should be updated.

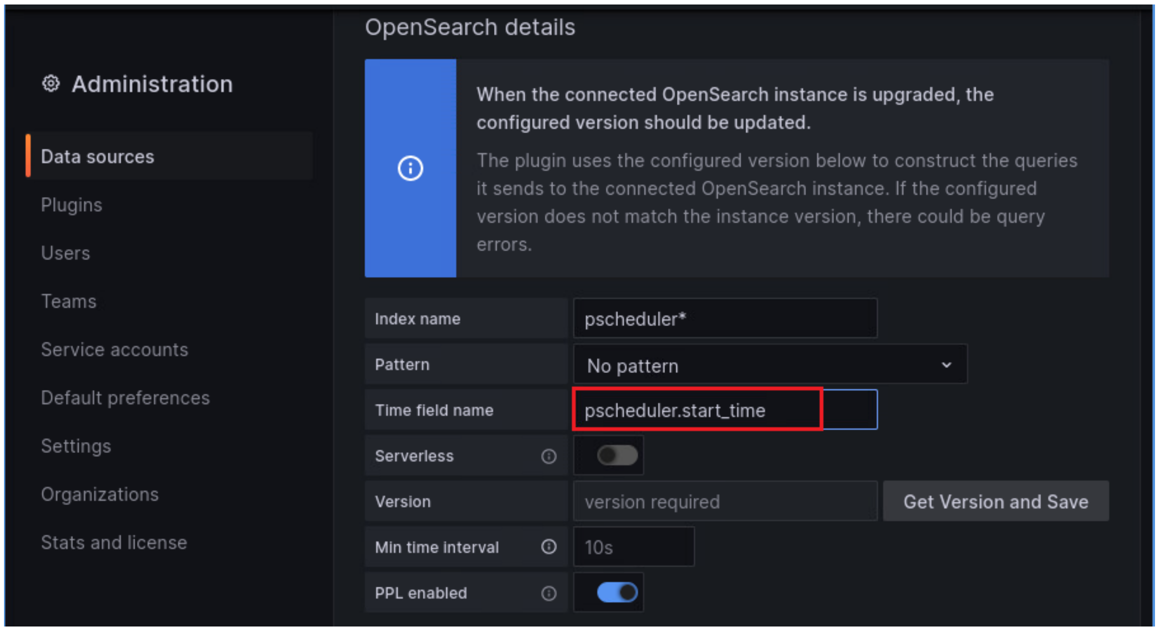
The plugin uses the configured version below to construct the queries it sends to the connected OpenSearch instance. If the configured version does not match the instance version, there could be query errors.

Index name	pscheduler*
Pattern	No pattern
Time field name	@timestamp
Serverless	
Version	version required
Min time interval	10s
PPL enabled	

Figure 32. Setting the index name.

Step 10. Type the following entry as the *Time field name.*

```
pscheduler.start_time
```



OpenSearch details

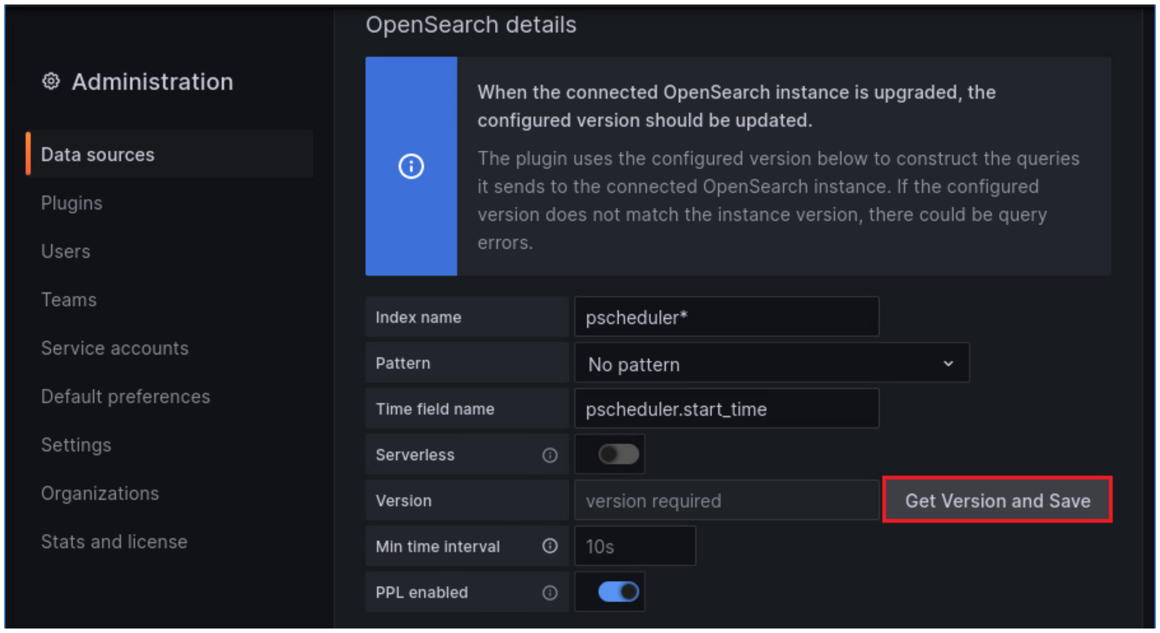
When the connected OpenSearch instance is upgraded, the configured version should be updated.

The plugin uses the configured version below to construct the queries it sends to the connected OpenSearch instance. If the configured version does not match the instance version, there could be query errors.

Index name	pscheduler*
Pattern	No pattern
Time field name	pscheduler.start_time
Serverless	
Version	version required
Min time interval	10s
PPL enabled	

Figure 33. Setting the time field name.

Step 11. Click on *Get Version and Save* to test the connection with the OpenSearch database. The output will indicate the current version.



OpenSearch details

When the connected OpenSearch instance is upgraded, the configured version should be updated.

The plugin uses the configured version below to construct the queries it sends to the connected OpenSearch instance. If the configured version does not match the instance version, there could be query errors.

Index name	pscheduler*
Pattern	No pattern
Time field name	pscheduler.start_time
Serverless	
Version	version required
Min time interval	10s
PPL enabled	

Figure 34. Getting the database version.

Step 12. Scroll down and click on *Save & test* to save the configured data source.

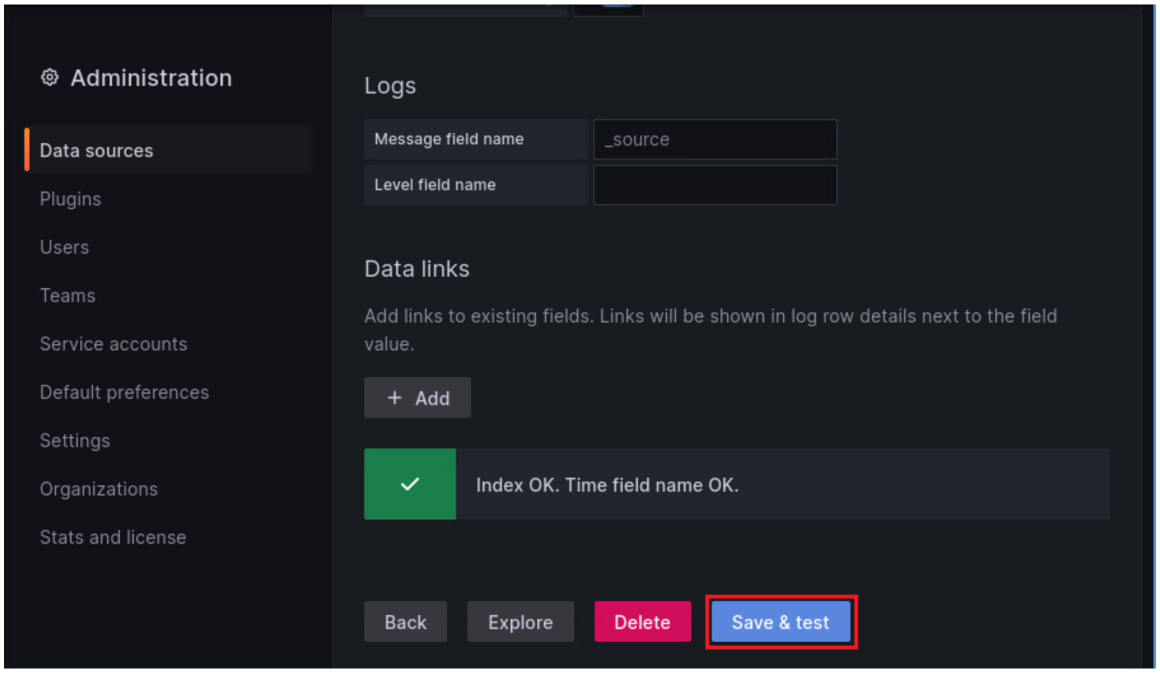


Figure 35. Saving the configuration.

5 Displaying the measurement results in a dashboard

In this section, the user will create a dashboard and retrieve measurements such as throughput, Round-trip Time (RTT), and retransmissions. The graphs will visually present these metrics over time, offering valuable insights into network performance trends. It's important to understand that the configuration provided here outlines the fundamental steps to display these results effectively. However, Grafana's flexibility allows for extensive customization, empowering users to tailor the views according to their specific needs and even set up alerts to monitor thresholds and trigger notifications if certain conditions are surpassed.

5.1 Visualizing throughput results

Step 1. Click on the + icon in the upper right of the screen and select *New dashboard*.

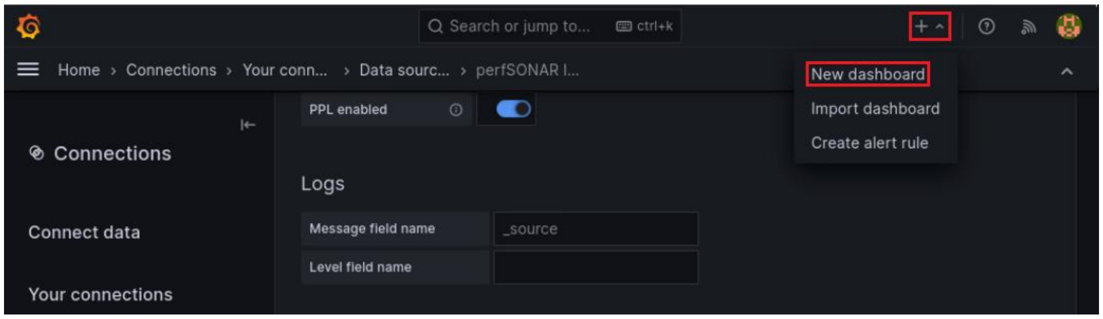


Figure 36. Creating a new dashboard.

Step 2. Click on *+ Add visualization.*

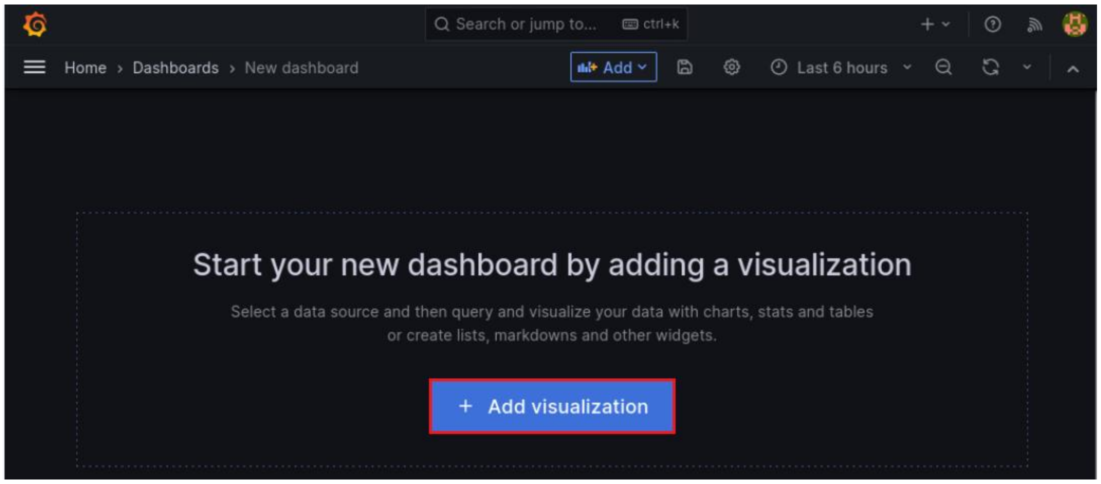
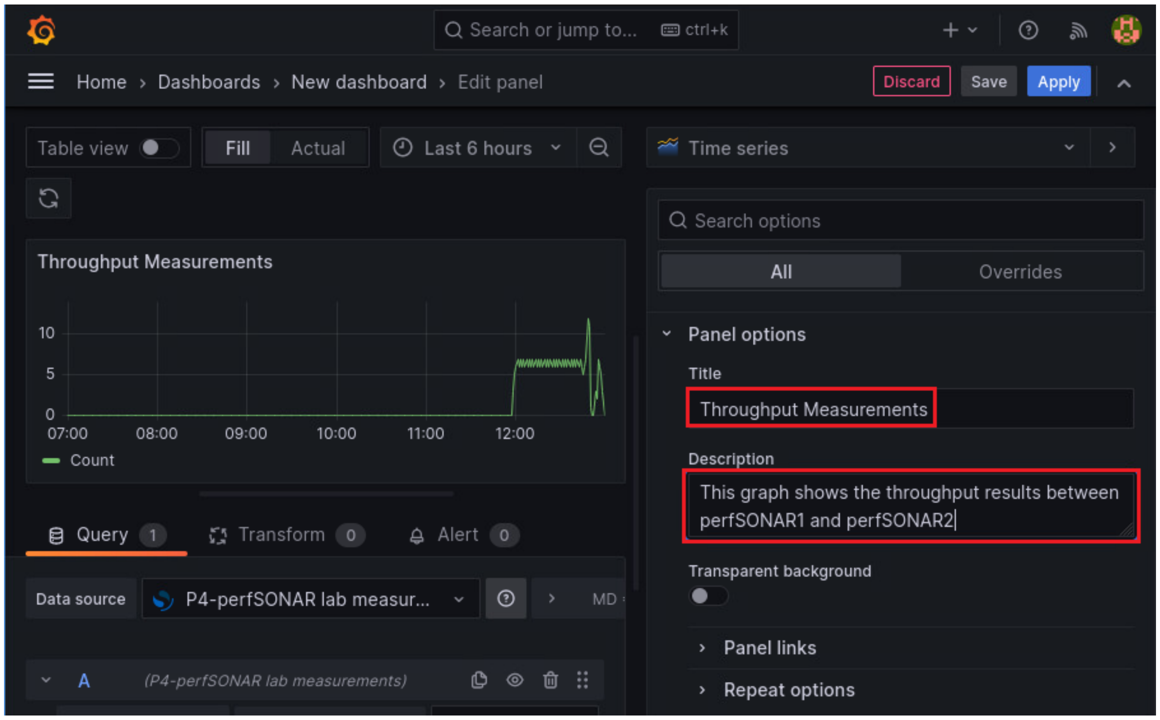

Search or jump to...
ctrl+k
Home › Dashboards › New dashboard
Add
Last 6 hours
Start your new dashboard by adding a visualization
Select a data source and then query and visualize your data with charts, stats and tables or create lists, markdowns and other widgets.
+ Add visualization

Figure 37. Adding a graph.

Step 3. On the right panel, in the *Panel options,* add a *Title* and a *Description*. This graph will display the throughput tests.



Title: Throughput Measurements

Description: This graph shows the throughput results between perfSONAR1 and perfSONAR2

Transparent background

Figure 38. Setting the graph title and the description.

Step 4. In the *Querry* panel, change the metric from *Count* to *Max*, as shown in the figure below.

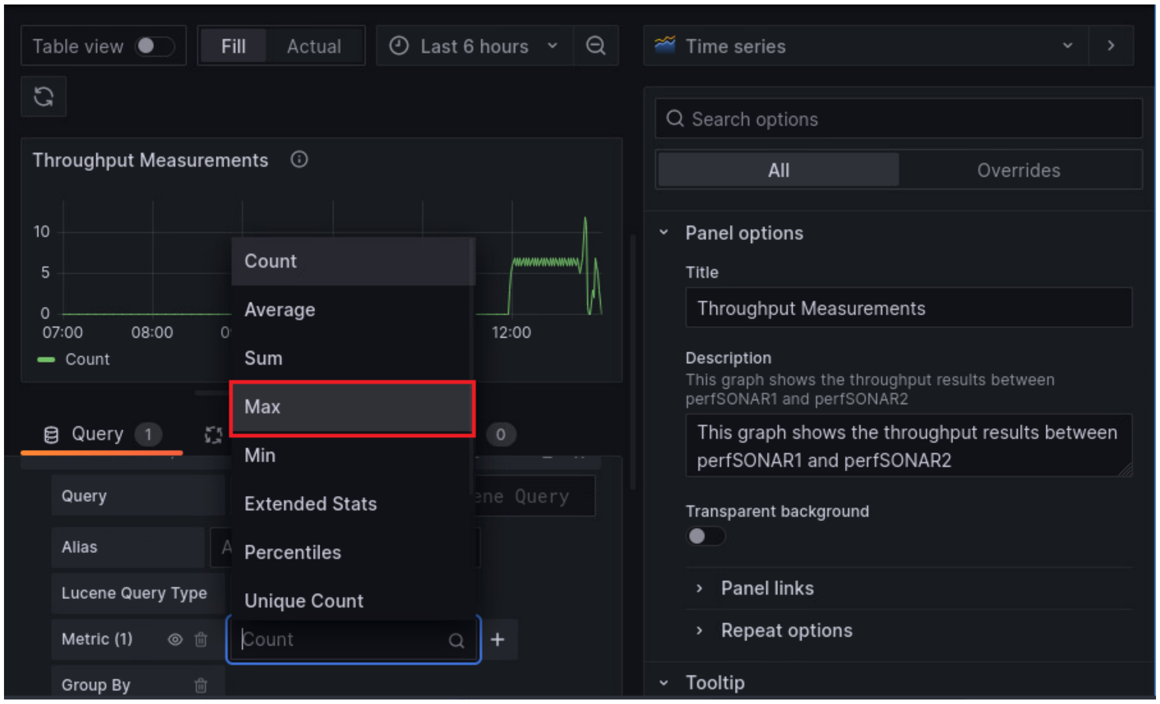


Figure 39. Displaying the maximum values.

Step 5. In the adjacent box, select *result.throughput* from the list.

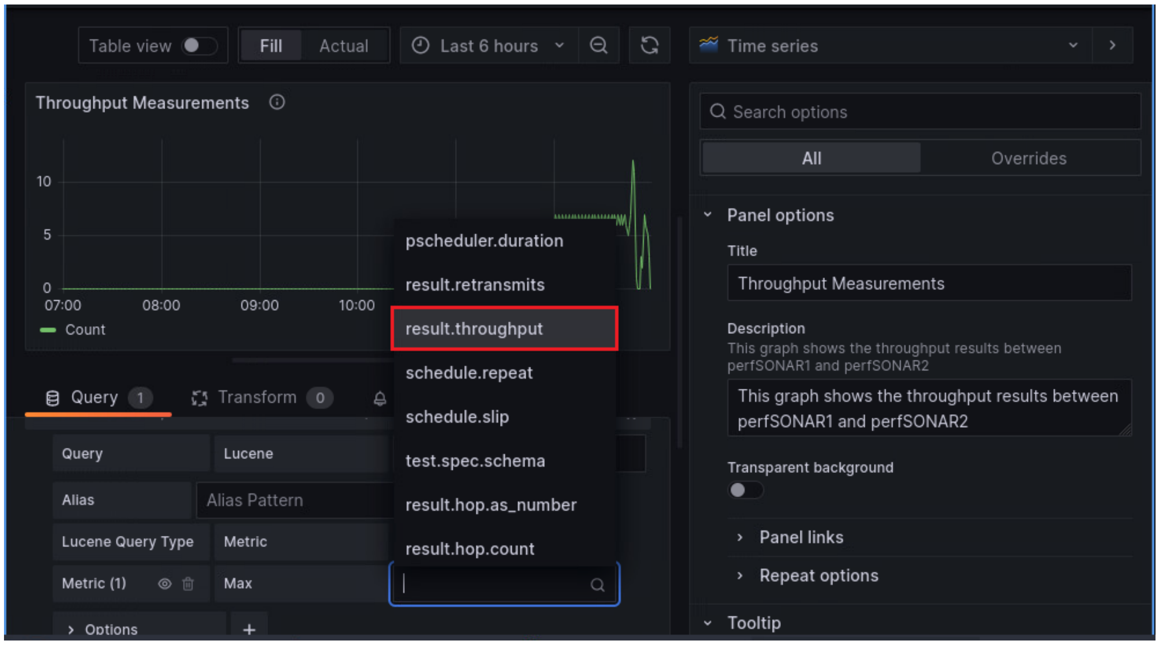


Figure 40. Selecting the throughput results.

Step 6. Change the time range by clicking on the icon located on the upper part and selecting the *Last 5 minutes* option.

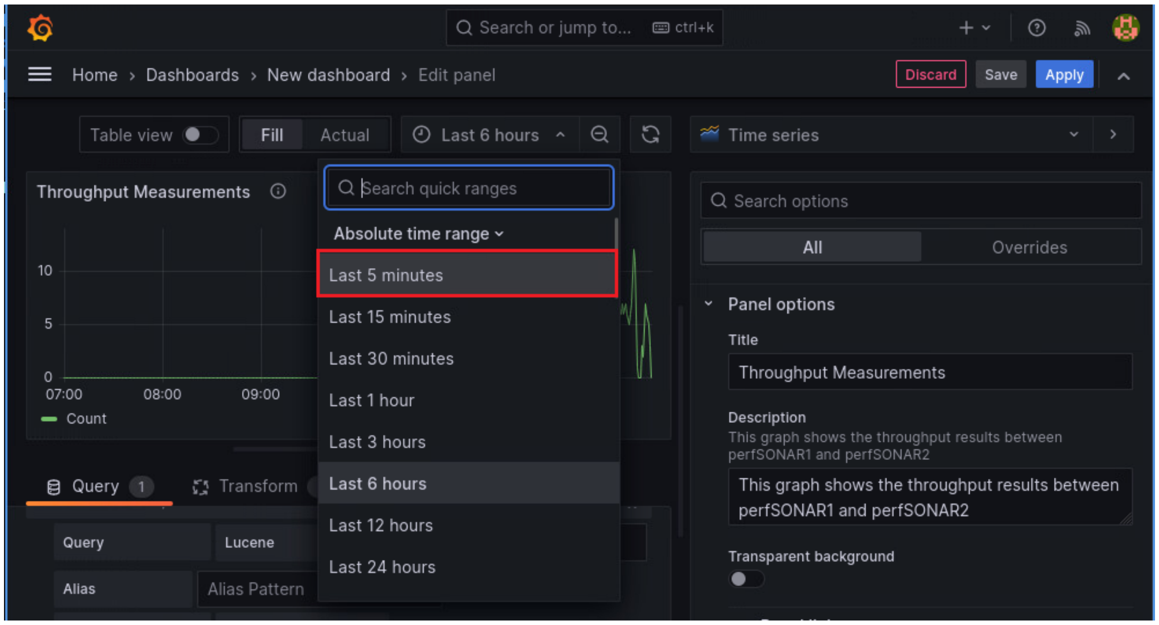


Figure 41. Adjusting the time range to 5 minutes.

Step 7. In the panel on the right, scroll down to reach the *Standard options* section. Select *Data /bits(SI)* option for the *Unit* field to change the unit to Gigabits (Gb).

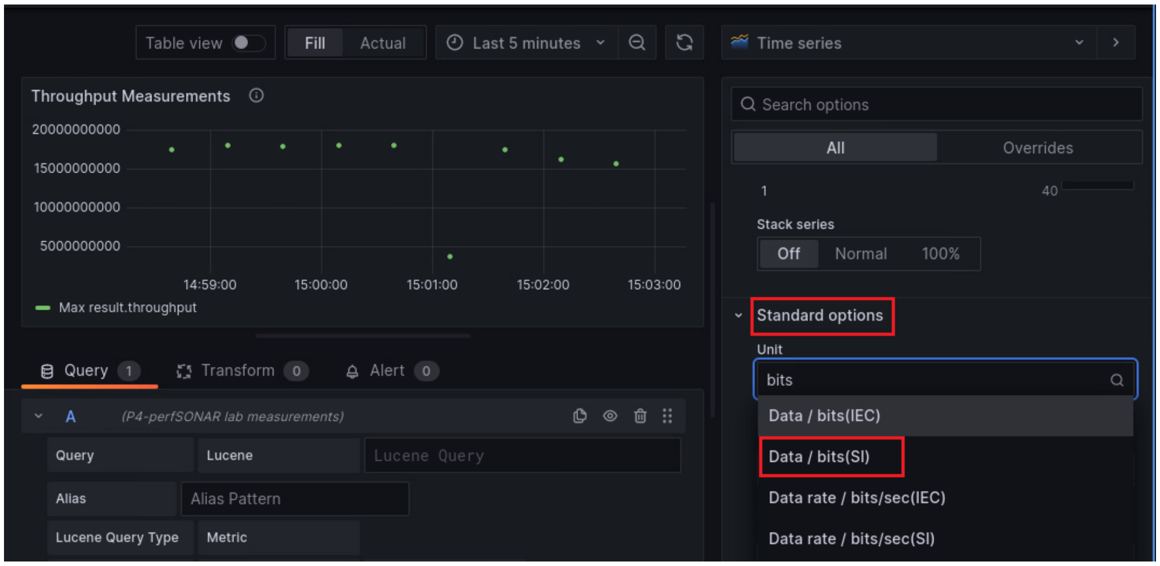


Figure 42. Adjusting unit to Gigabits.

Step 8. Click on *Apply* to save the changes.

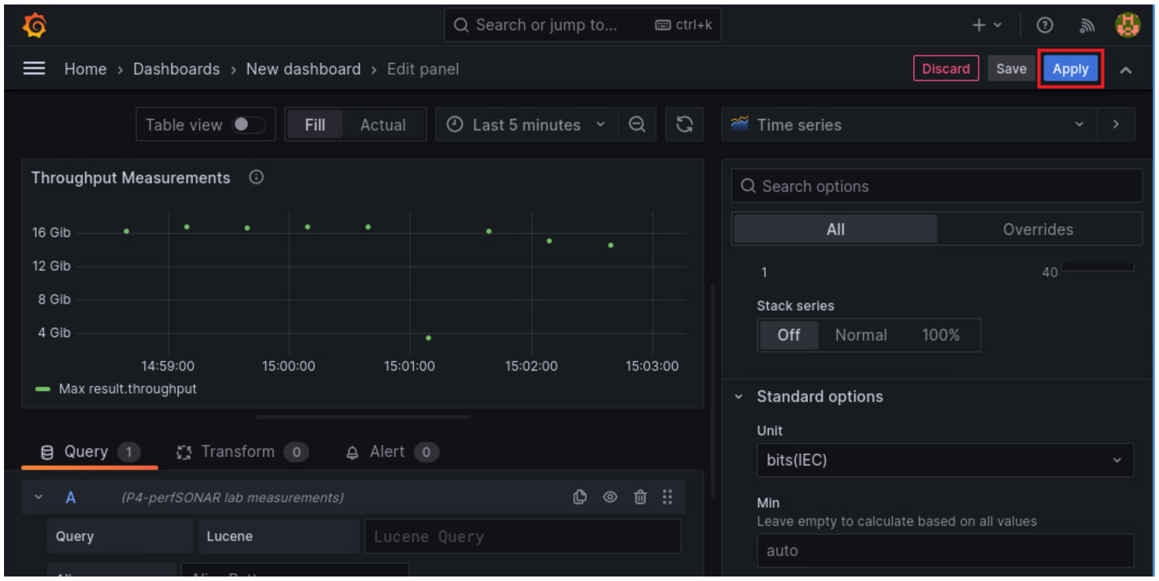


Figure 43. Applying the changes to the dashboard.

The throughput graph will appear on the Grafana dashboard.

Step 9. Save the dashboard by clicking on the icon on the upper part. A new panel will emerge on the right side.

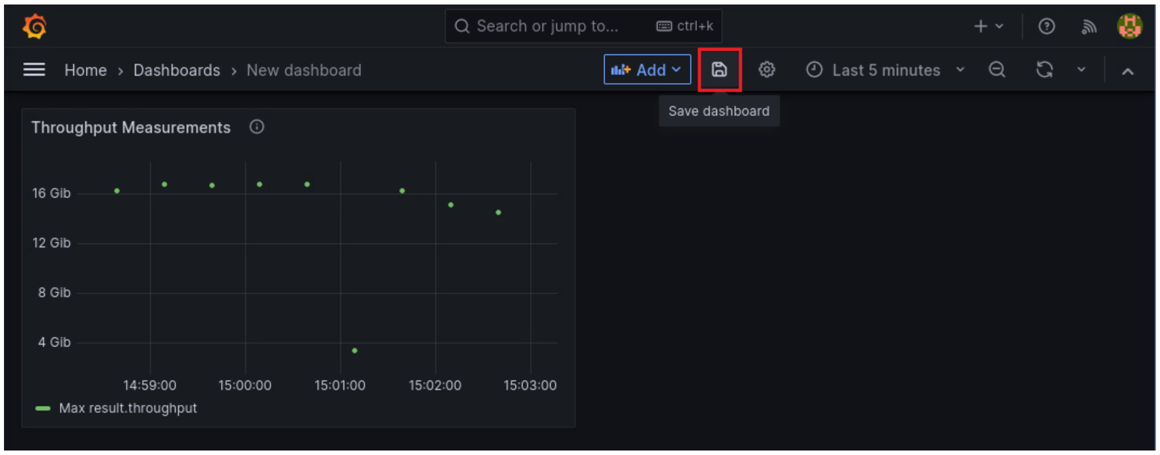


Figure 44. Saving the changes in the dashboard.

Step 10. Provide a name to the dashboard, then, click on *Save.*

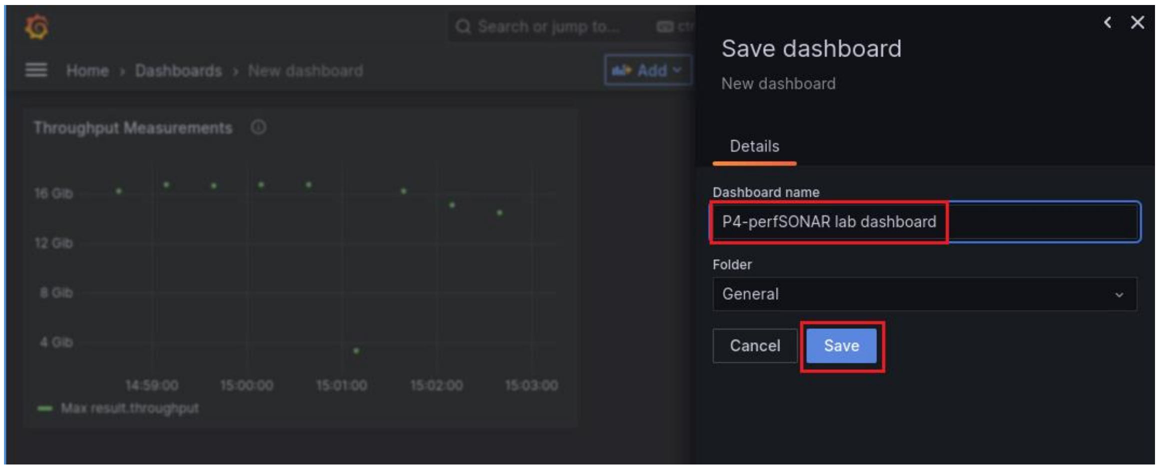


Figure 45. Saving the changes in the dashboard.

5.2 Visualizing RTT results

Step 1. Click on the *Add* icon located on the upper part and select *Visualization* as shown in the figure below.

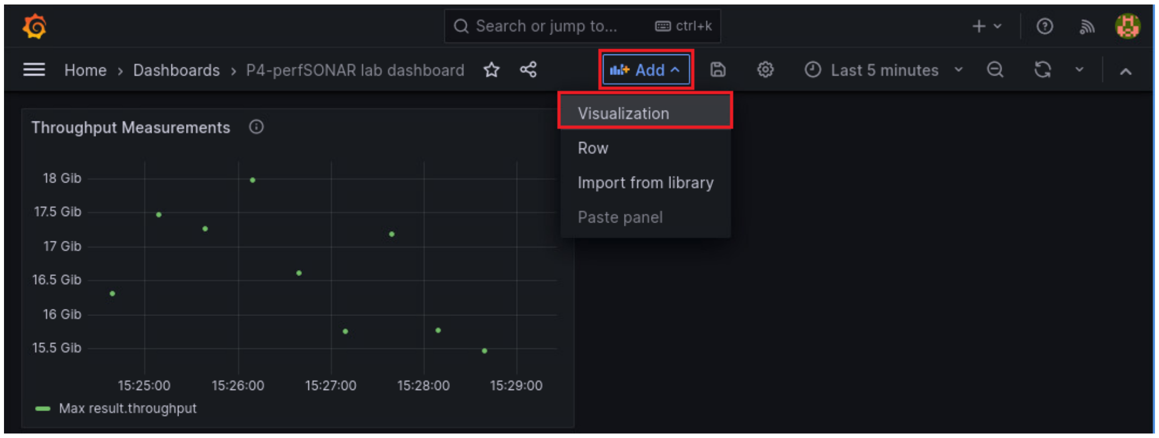
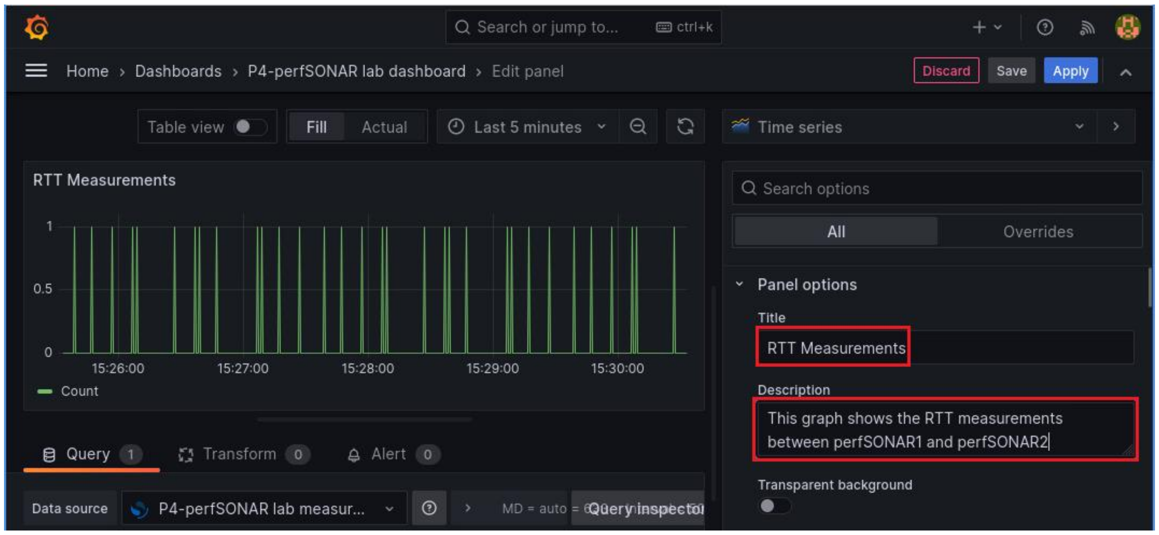


Figure 46. Adding a graph.

Step 2. On the right panel, in the *Panel options,* add a *Title* and a *Description*. This graph will display the RTT tests.

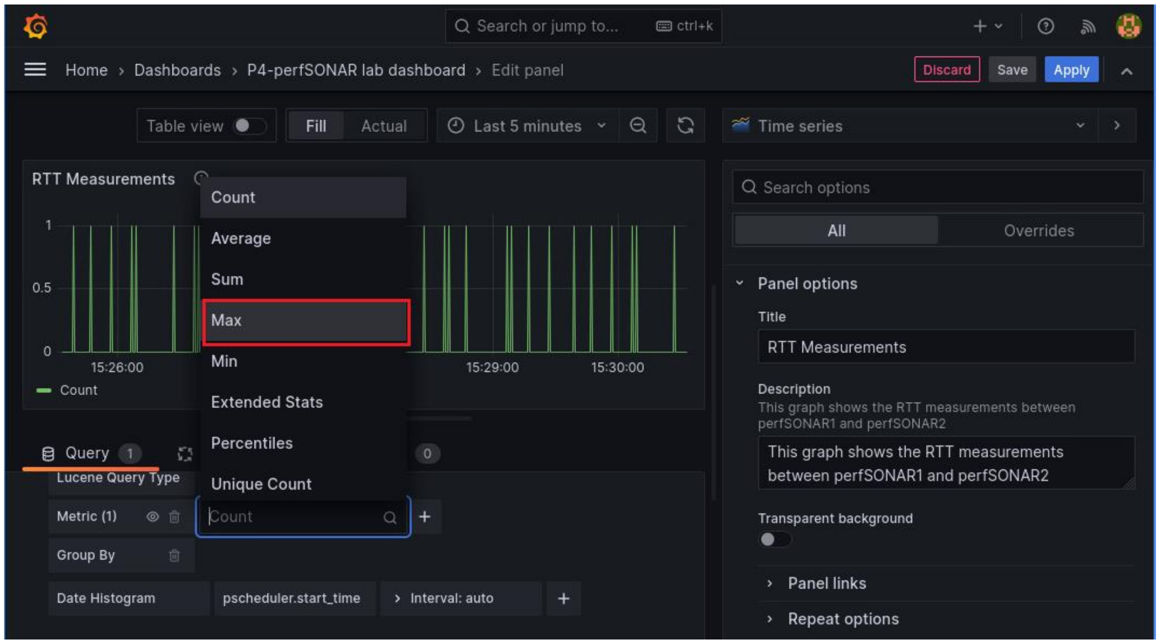


Title: RTT Measurements

Description: This graph shows the RTT measurements between perfSONAR1 and perfSONAR2

Figure 47. Setting the graph title and the description.

Step 3. In the *Querry* panel, change the metric from *Count* to *Max*, as shown in the figure below.



Title: RTT Measurements

Description: This graph shows the RTT measurements between perfSONAR1 and perfSONAR2

Figure 48. Displaying the maximum values.

Step 4. In the adjacent box, select *result.rtt.meant* from the list.

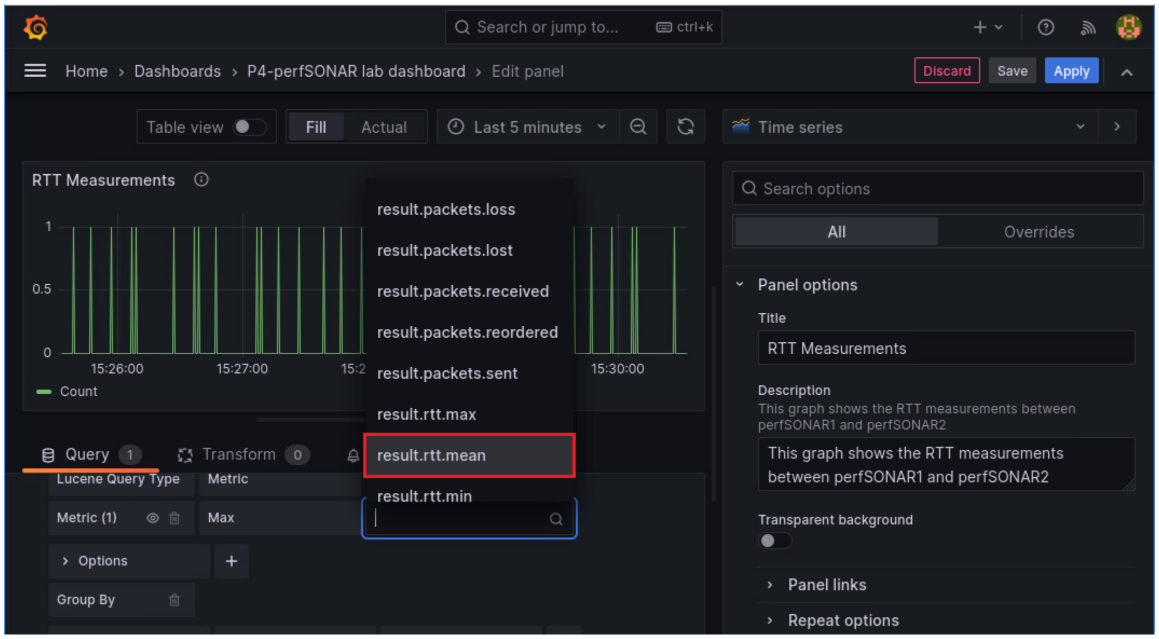


Figure 49. Selecting the RTT means values.

Step 5. Click on the icon located on the upper part to refresh the graph.

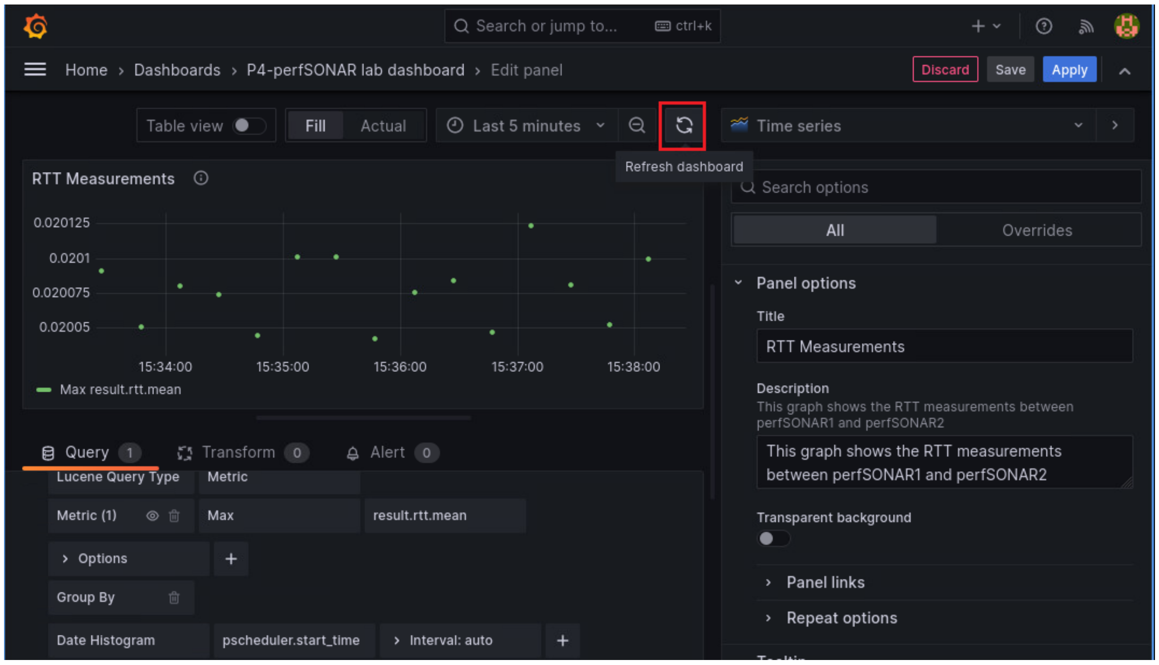


Figure 50. Refreshing the dashboard.

Step 6. In the panel on the right, scroll down to reach the *Standard options* section. Select *Time / second (s)* option for the *Unit* field to change the unit to seconds.

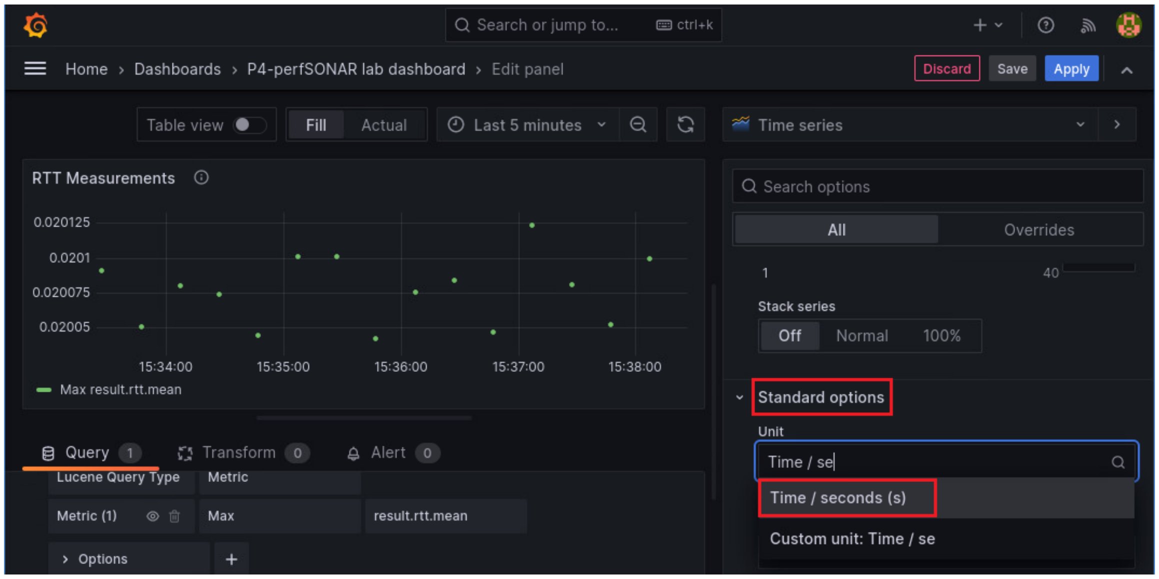


Figure 51. Adjusting unit to seconds.

Step 7. Click on *Apply* to save the changes.

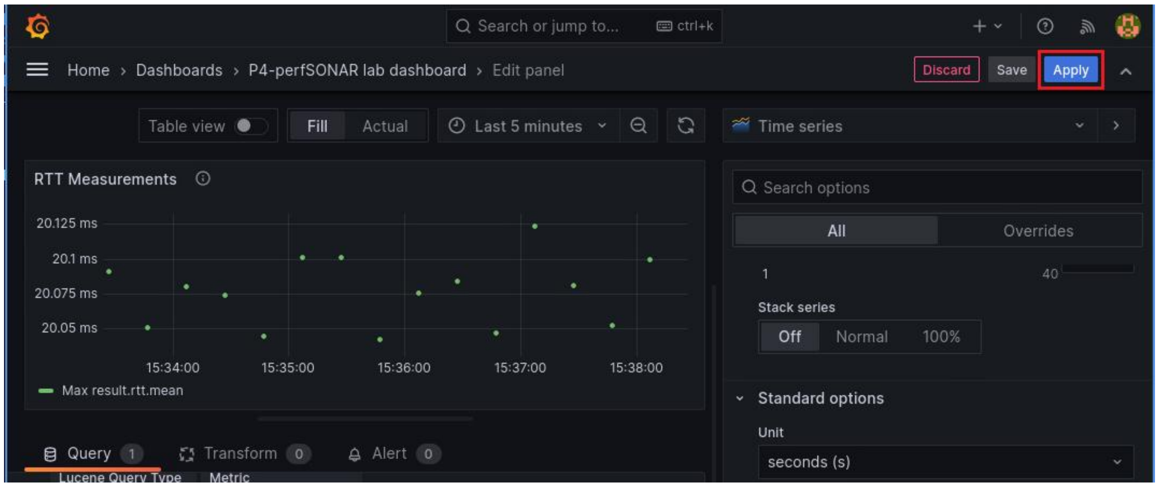


Figure 52. Applying the changes.

The RTT graph will appear on the Grafana dashboard.

Step 8. Save the changes by clicking on the icon on the upper part. A new panel will emerge on the right side.

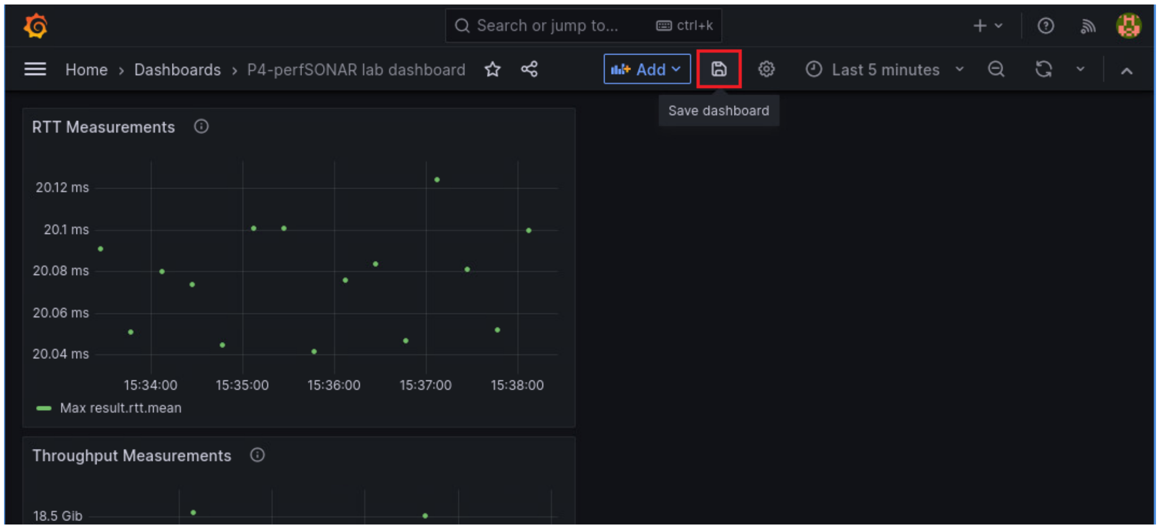


Figure 53. Saving the changes in the dashboard.

Step 9. Click on *Save* to save the changes.

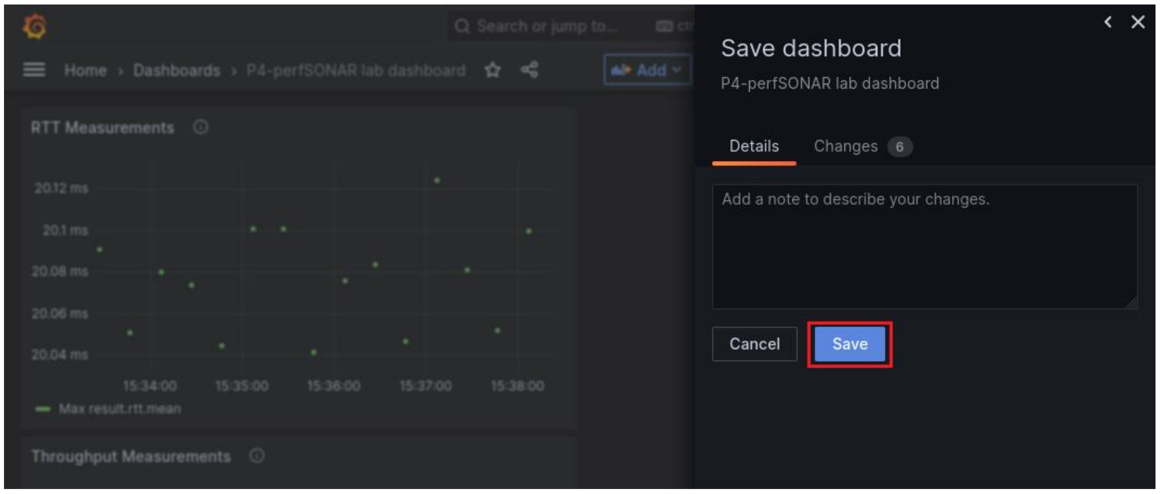

Save dashboard
P4-perfSONAR lab dashboard
Details
Changes 6
Add a note to describe your changes.
Cancel
Save

Figure 54. Saving the changes in the dashboard.

At this point, there are two graphs in the dashboard.

5.3 Visualizing retransmissions

Step 1. Click on the *Add* icon located on the upper part and select *Visualization* as shown in the figure below.

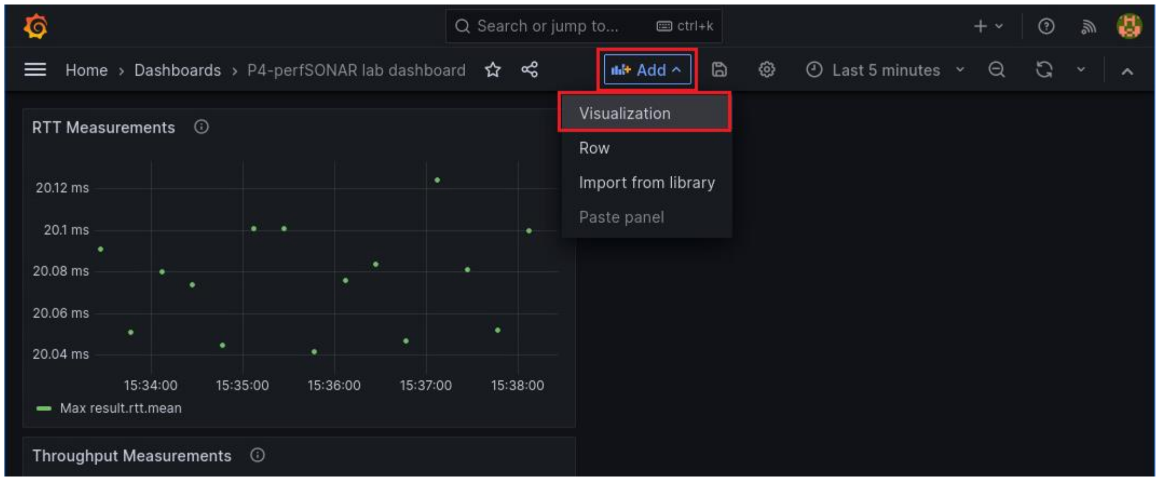
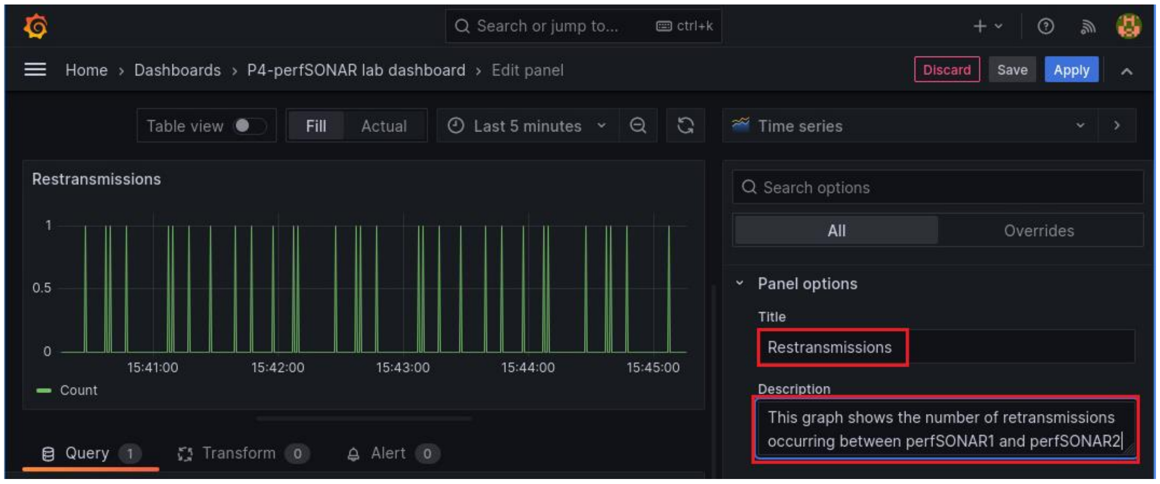


Figure 55. Adding a graph.

Step 2. On the right panel, in the *Panel options,* add a *Title* and a *Description*. This graph will display the throughput tests.



Title: Restransmissions

Description: This graph shows the number of retransmissions occurring between perfSONAR1 and perfSONAR2

Figure 56. Setting the title and the description.

Step 3. In the *Querry* panel, change the metric from *Count* to *Max*, as shown in the figure below.

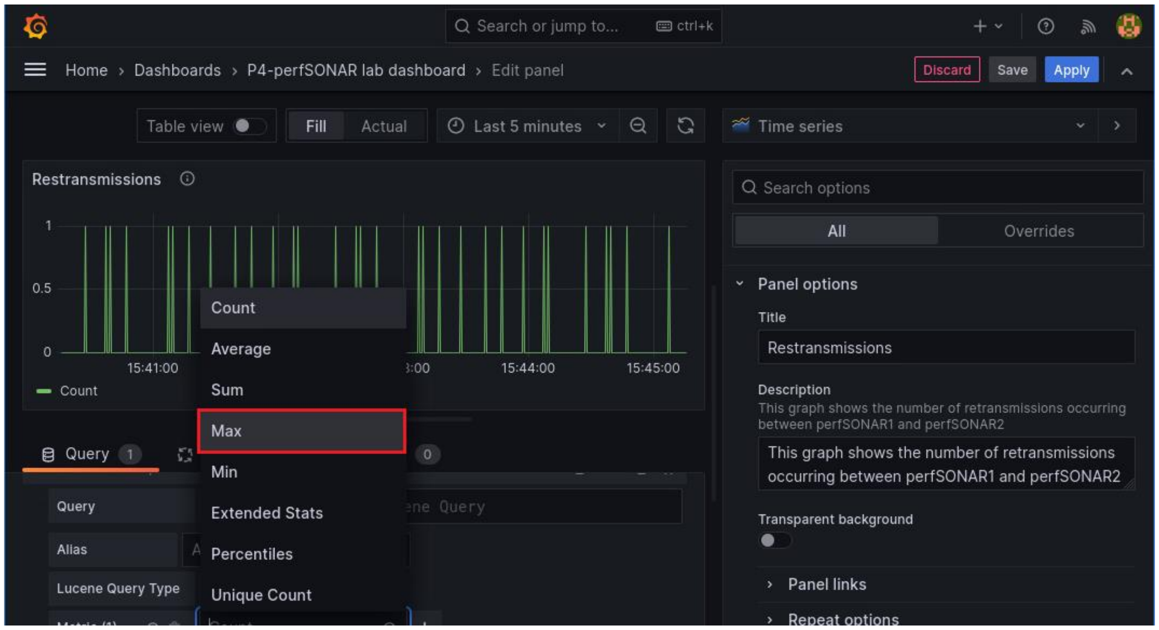


Figure 57. Displaying the maximum values.

Step 4. In the adjacent box, select *result.retransmits* from the list.

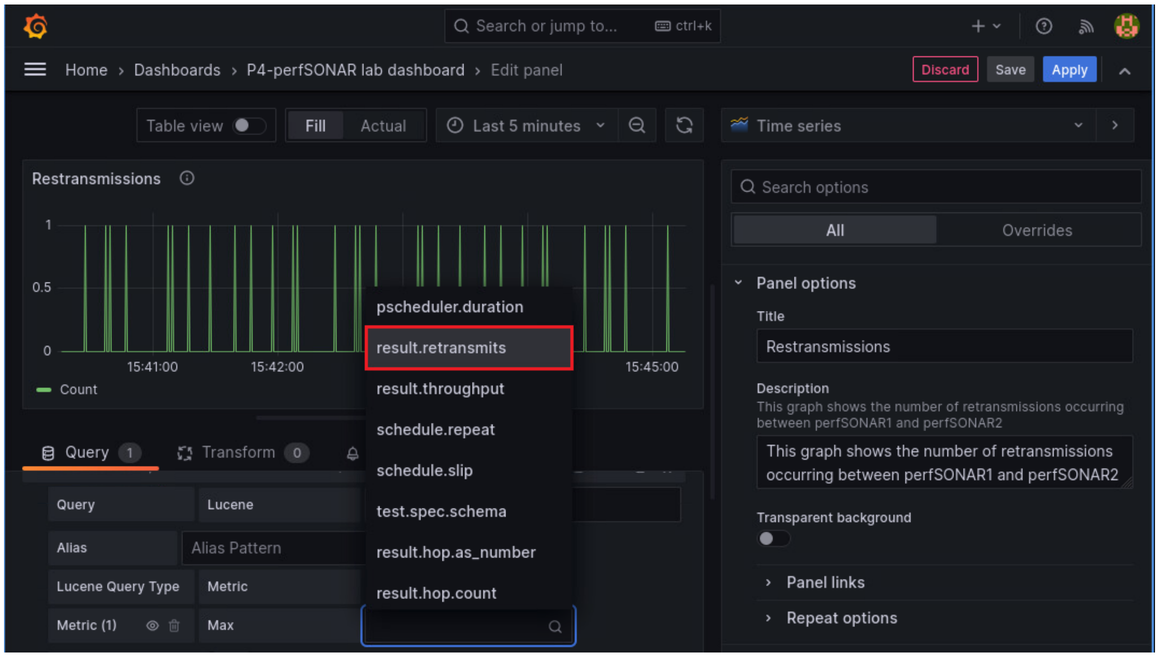
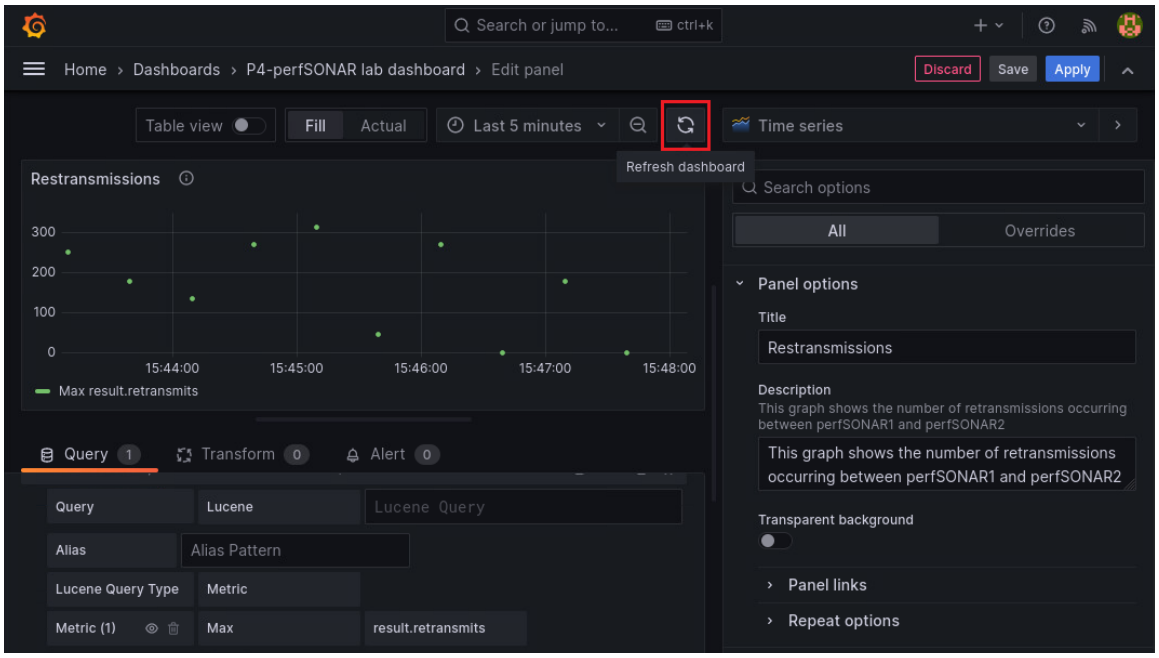


Figure 58. Selecting the number of retransmissions.

Step 5. Click on the icon located on the upper part to refresh the graph.



Query: Lucene — Lucene Query

Alias: Alias Pattern

Lucene Query Type: Metric

Metric (1): Max — result.retransmits

Title: Restransmissions

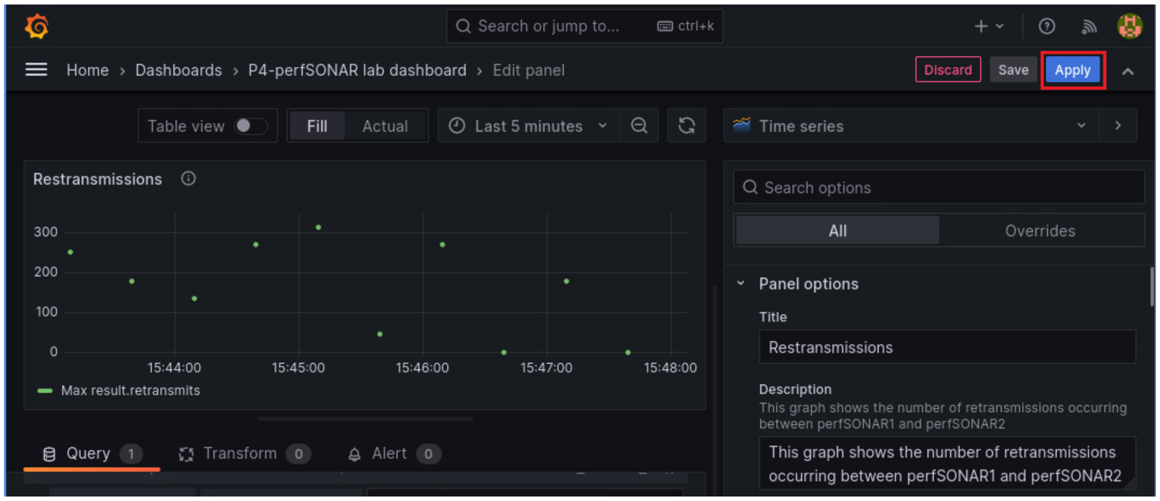
Description: This graph shows the number of retransmissions occurring between perfSONAR1 and perfSONAR2

This graph shows the number of retransmissions occurring between perfSONAR1 and perfSONAR2

Transparent background

Figure 59. Refreshing the graph.

Step 6. Click on *Apply* to save the changes.



Title: Restransmissions

Description: This graph shows the number of retransmissions occurring between perfSONAR1 and perfSONAR2

This graph shows the number of retransmissions occurring between perfSONAR1 and perfSONAR2

Figure 60. Applying the changes.

The Retransmissions graph will appear on the Grafana dashboard, with the throughput and RTT graphs.

Step 7. Drag the RTT graph next to the retransmissions graph as shown in the figure below.

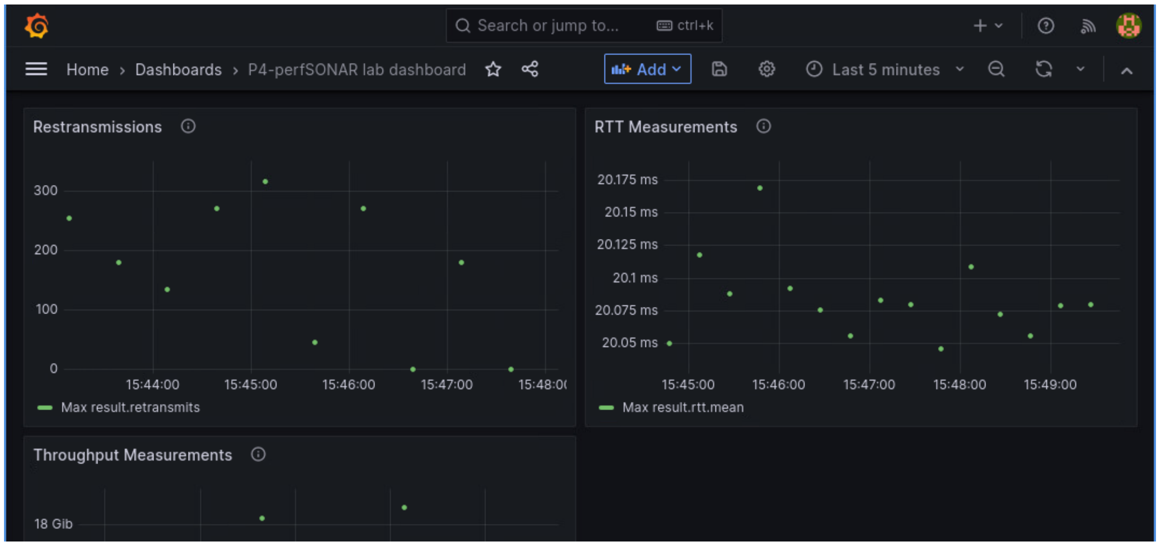


Figure 61. Relocating the graph in the dashboard.

Step 8. Save the changes by clicking on the icon on the upper part. A new panel will emerge on the right side.

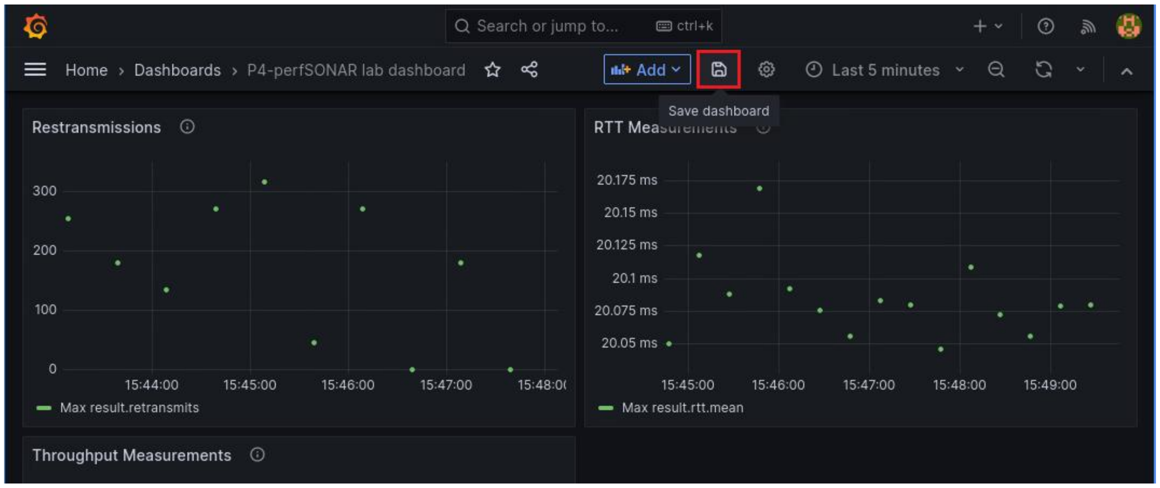


Figure 62. Saving the changes in the dashboard.

Step 9. Click on *Save* to save the changes.

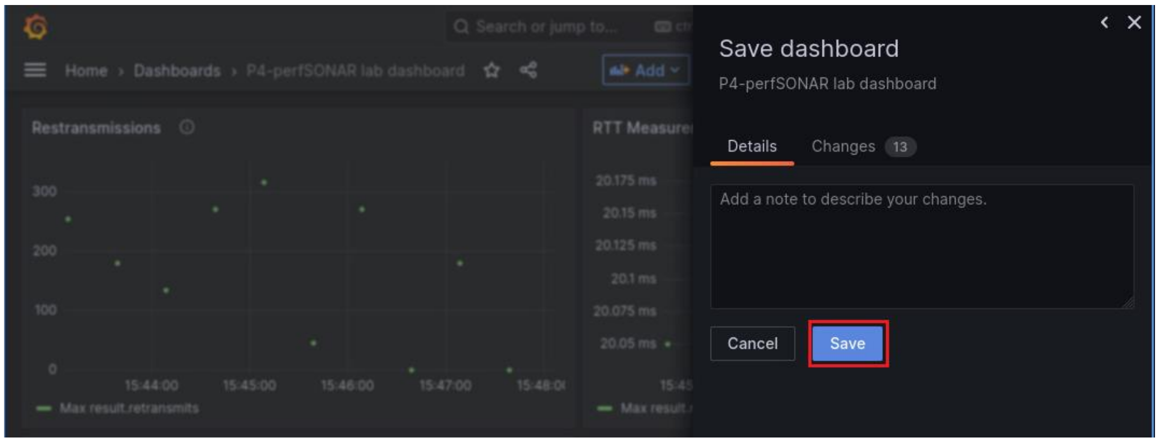

Save dashboard
P4-perfSONAR lab dashboard
Details
Changes 13
Add a note to describe your changes.
Cancel
Save

Figure 63. Saving the changes in the dashboard.

Lab 6 has been completed. You may now end your reservation.

References

1. Grafana Labs. "Grafana documentation," [Online]. Available: https://grafana.com/docs/grafana/latest/
2. perfSONAR Project. "Grafana perfSONAR Dashboard Cookbook," [Online]. Available: https://tinyurl.com/2nch38j8
3. OpenSearch contributors. "OpenSearch," [Online]. Available: https://opensearch.org/
4. perfSONAR Project. "*pScheduler limits*," [Online]. Available: https://tinyurl.com/pS-limits
5. Mininet walkthrough, [Online]. Available: http://Mininet.org.
6. perfSONAR Project, "perfSONAR Installation Options," [Online]. Available: https://docs.perfsonar.net/install_options.html
7. perfSONAR Project. "What is perfSONAR?," [Online]. Available: https://docs.perfsonar.net/intro_about.html
8. Docker Inc, "*What is a container,*" [Online]. Available: https://www.docker.com/resources/what-container/
9. Manuel Peuster, "*Containernet,*" [Online]. Available: https://containernet.github.io/
10. Brian Linkletter, "*How to use MiniEdit, Mininet's graphical user interface*,"[Online]. Available: https://tinyurl.com/MiniEdit-PS
11. perfSONAR Project, "*perfSONAR Global NodeDirectory*". Available: https://stats.es.net/ServicesDirectory/.

P4-PERFSONAR LAB SERIES

Lab 7: Retrieving per-flow Statistics from the Data Plane

Document Version: 06-06-2024

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

Contents

Overview 3
Objectives.. 3
Lab settings 3
Lab roadmap 3
1 Introduction to packet digests 3
1.1 Lab scenario 4
2 Lab topology.. 4
2.1 Starting the end hosts.. 6
3 Calculating per-flow number of bytes in P4 7
3.1 Loading the programming environment.. 7
3.2 Populate the metadata 8
3.3 Collecting per-flow number of bytes 9
4 Creating packet digests in P4 13
5 Creating the controller application.. 17
6 Loading the P4 program.. 23
6.1 Compiling and loading the P4 program to switch s1 23
6.2 Verifying the configuration 25
7 Configuring switch s1 26
8 Testing and verifying the P4 program.. 26
8.1 Starting the controller application.. 26
8.2 Starting data transfer between h1 and h2 27
8.3 Inspecting the measurements retrieved by the control plane.. 28
References 29

Overview

This lab demonstrates how to detect new flows in P4 and report flows' information using digests to the control plane. A digest is a communication mechanism used by the data plane to send values to the control plane. The control plane then processes these values to implement applications. After that, the user will utilize the flow information received from the digest to query the flow's statistics from the data plane.

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 rules in a match-action table.
4. Implement a basic monitoring 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: Calculating per-flow number of bytes in P4.
4. Section 4: Creating packet digests in P4.
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 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 a basic monitoring procedure. The topology comprises two end hosts and a P4 switch. The user will implement the logic to detect new flows by defining a new match-action table. The table will be responsible for detecting if an arriving packet belongs to a new flow. Packets are classified into flows by hashing the 5-tuple (source and destination IP addresses, source and destination ports, and protocol). The user will also utilize a stateful register to monitor the per-flow number of bytes.

For each new flow, the P4 program will produce a digest with the flow ID. This digest is sent to the control plane, where a controller (i.e., controller.py) uses flow ID to periodically query the number of bytes register. The throughput is then calculated by the control plane and presented to the user.

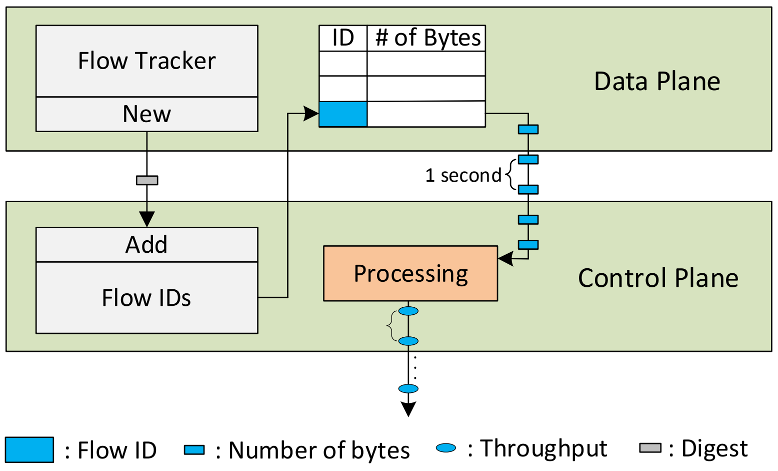


Figure 1. Lab scenario. The data plane detects new flows and reports them to the control plane using digests. After that, the data plane uses the flow ID to initialize a new register entry that maintains the number of bytes for that flow. Control plane uses the flow ID received from the digest to periodically query measurements from the data plane. The control plane calculates the throughput values from the number of bytes .

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.

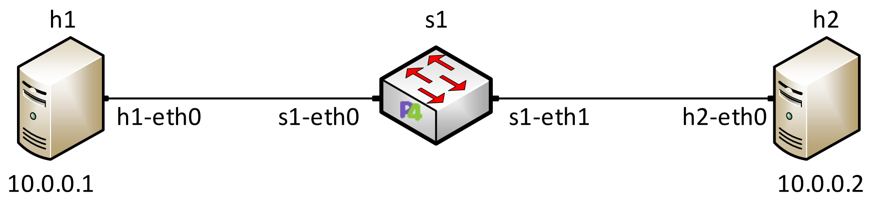


Figure 2. Lab topology.

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

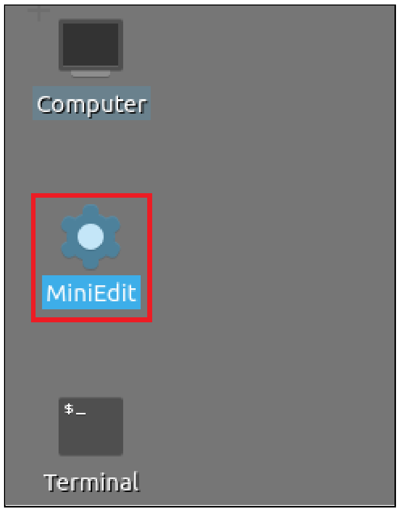


Figure 3. MiniEdit shortcut.

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

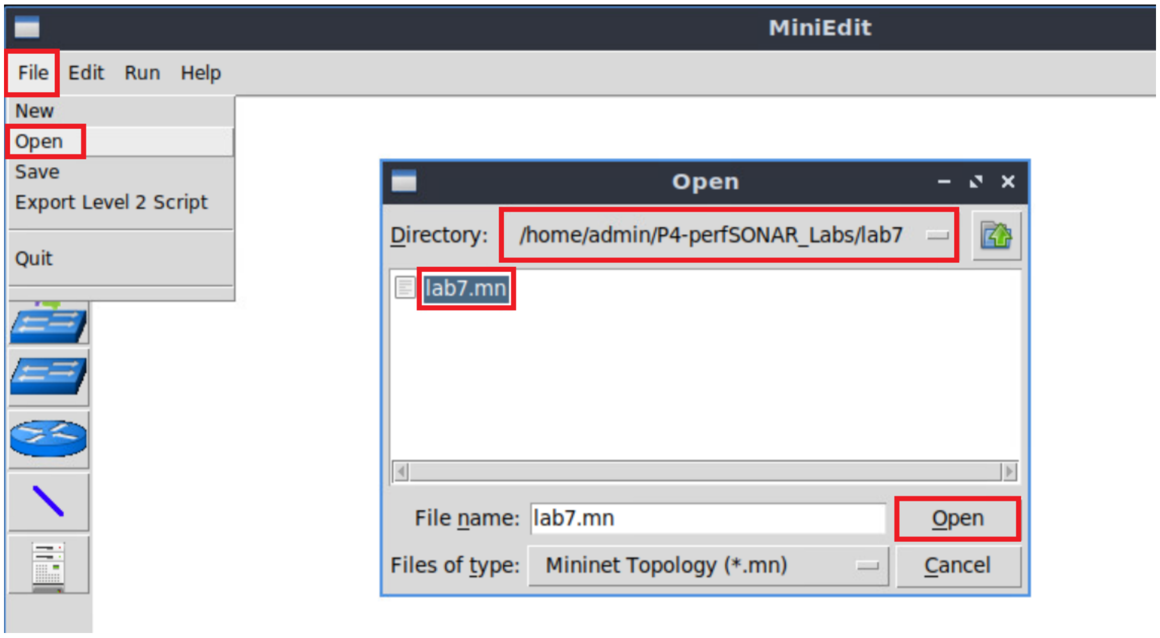


Figure 4. MiniEdit's *Open* dialog.

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

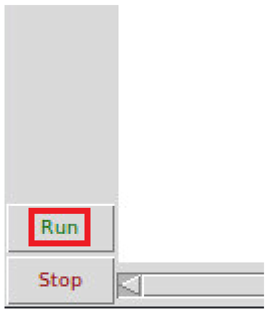


Figure 5. Running the emulation.

2.1 Starting 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.

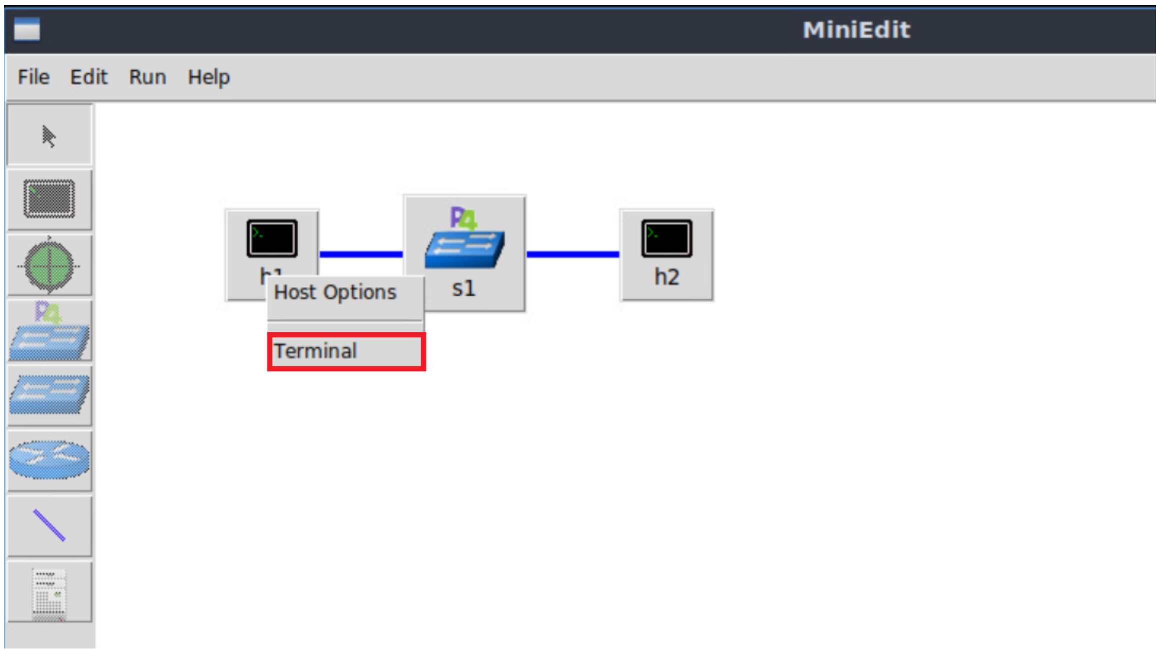
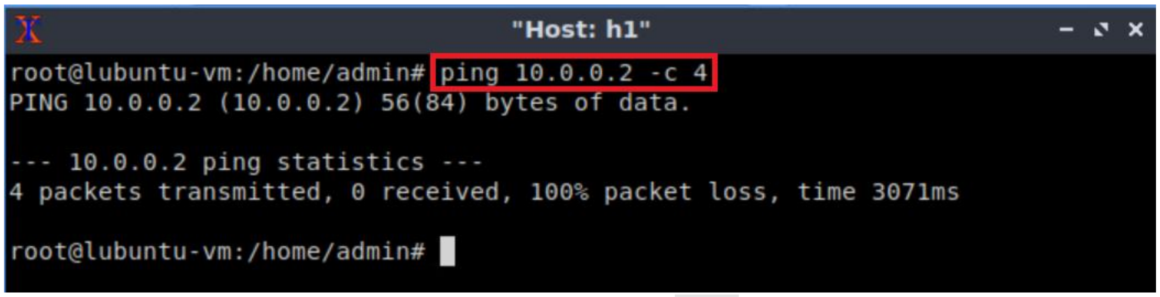


Figure 6. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```



```
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.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3071ms

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

Figure 7. 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 Calculating per-flow number of bytes in P4

3.1 Loading the programming environment

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

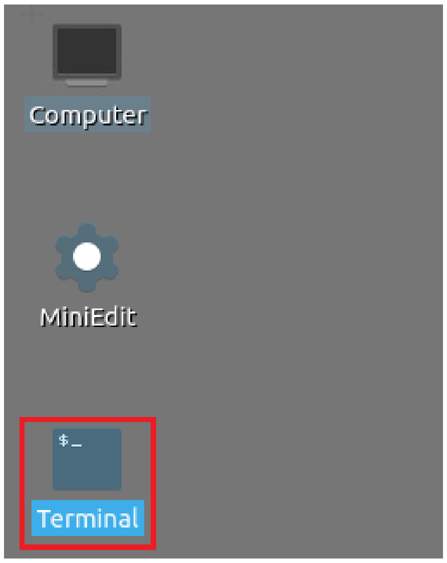
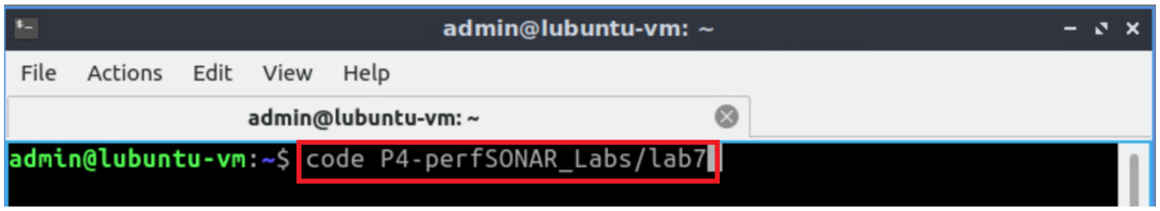


Figure 8. Shortcut to open a Linux terminal.

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-perfSOANR_Labs/Lab7
```

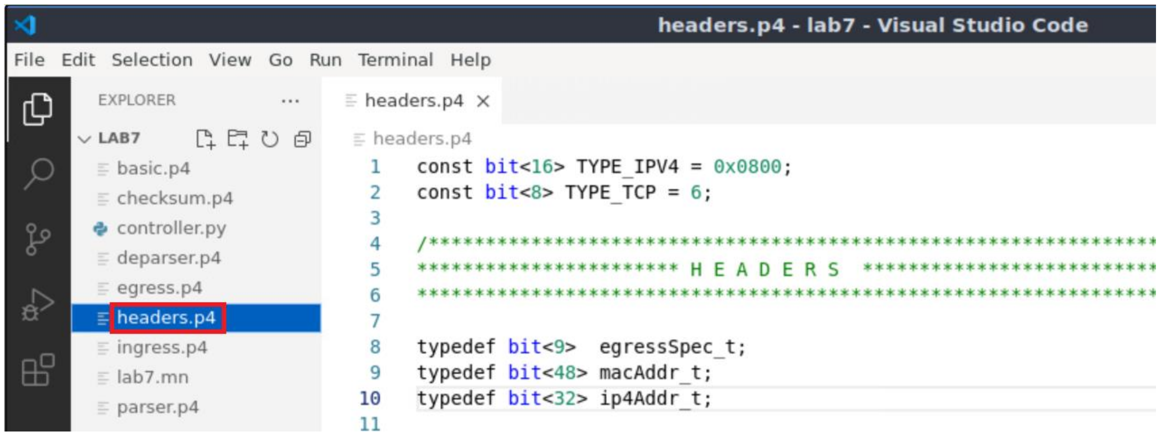


```
admin@lubuntu-vm:~$ code P4-perfSONAR_Labs/lab7
```

Figure 9. Loading the development environment.

3.2 Populate the 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.

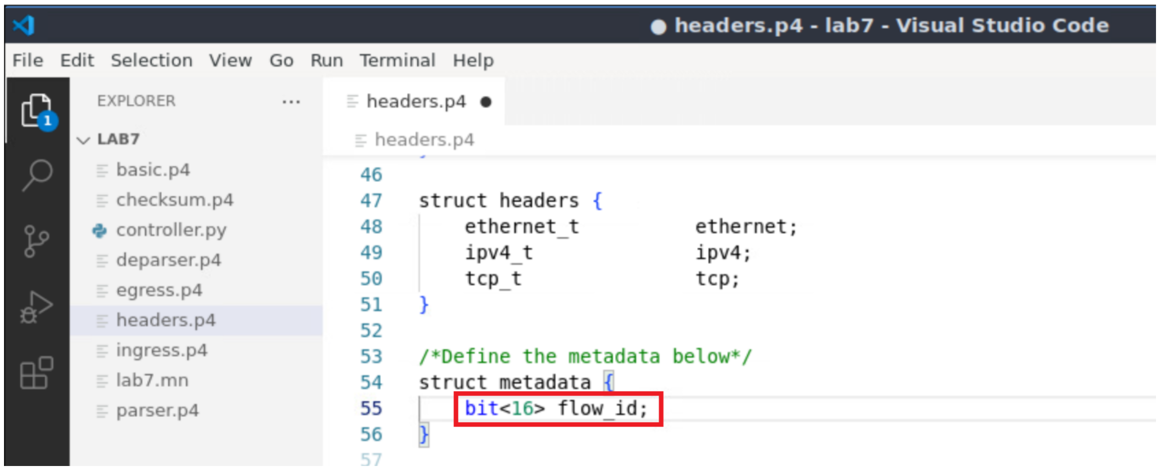


```
const bit<16> TYPE_IPV4 = 0x0800;
const bit<8> TYPE_TCP = 6;

/*************************************************************
*********************** H E A D E R S ************************
*************************************************************

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
```

Figure 10. Inspecting the *headers.p4* file.

Step 2. Define the `flow_id` metadata by adding the code below to the metadata struct.

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



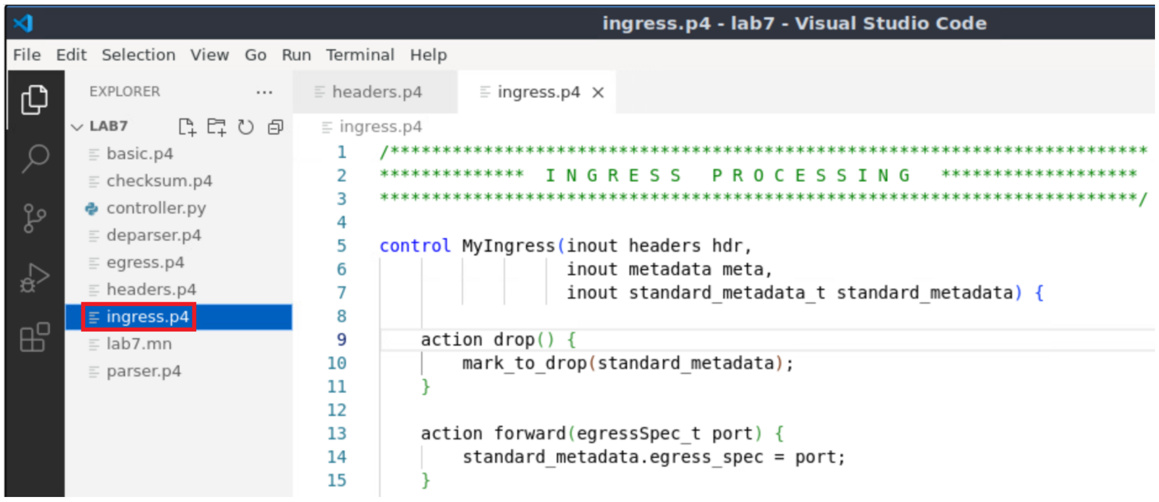
```
struct headers {
    ethernet_t          ethernet;
    ipv4_t              ipv4;
    tcp_t               tcp;
}

/*Define the metadata below*/
struct metadata {
    bit<16> flow_id;
}
```

Figure 11. Adding the metadata.

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

3.3 Collecting per-flow number of bytes

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.

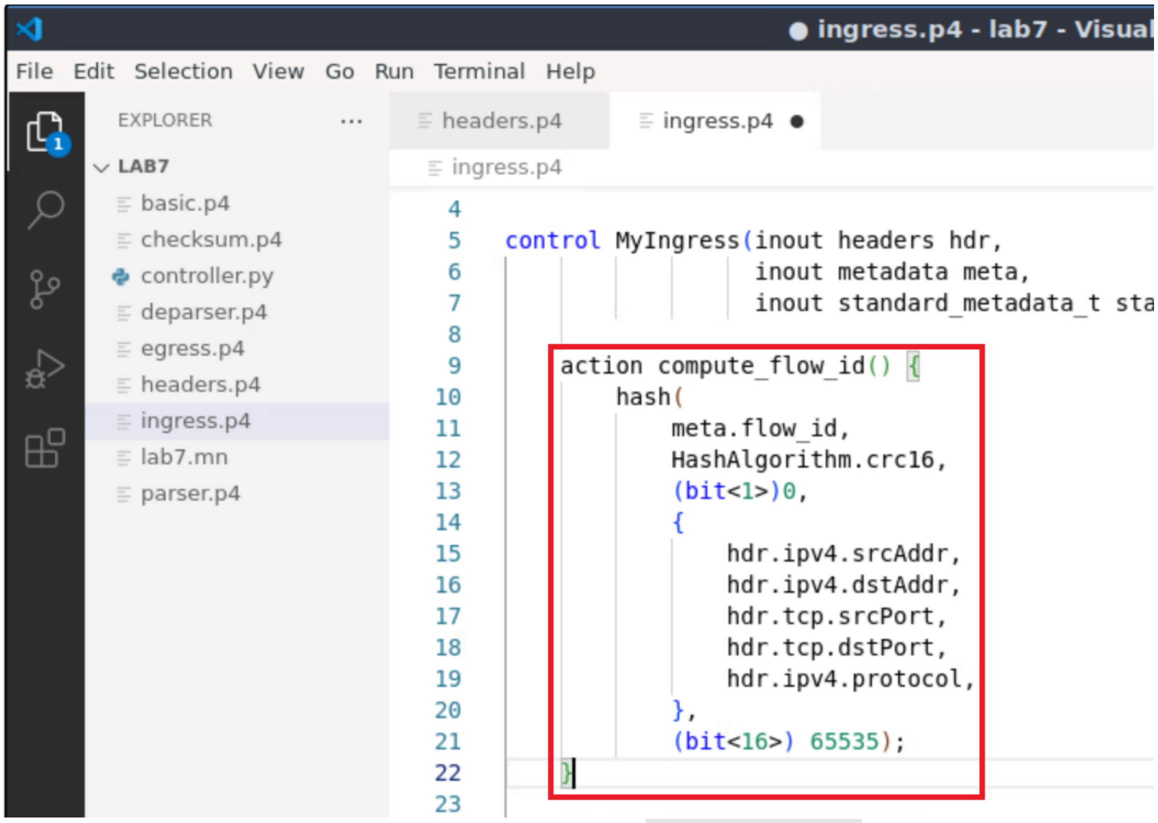


```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

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

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

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

```
action compute_flow_id() {
    hash (
         meta.flow_id,
         HashAlgorithm.crc16,
         (bit<1>)0,
         {
          hdr.ipv4.srcAddr,
          hdr.ipv4.dstAddr,
          hdr.tcp.srcPort,
          hdr.tcp.dstPort,
          hdr.ipv4.protocol,
          },
          (bit<16>)65535);
}
```



```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t sta

    action compute_flow_id() {
        hash(
            meta.flow_id,
            HashAlgorithm.crc16,
            (bit<1>)0,
            {
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.tcp.srcPort,
                hdr.tcp.dstPort,
                hdr.ipv4.protocol,
            },
            (bit<16>) 65535);
    }
```

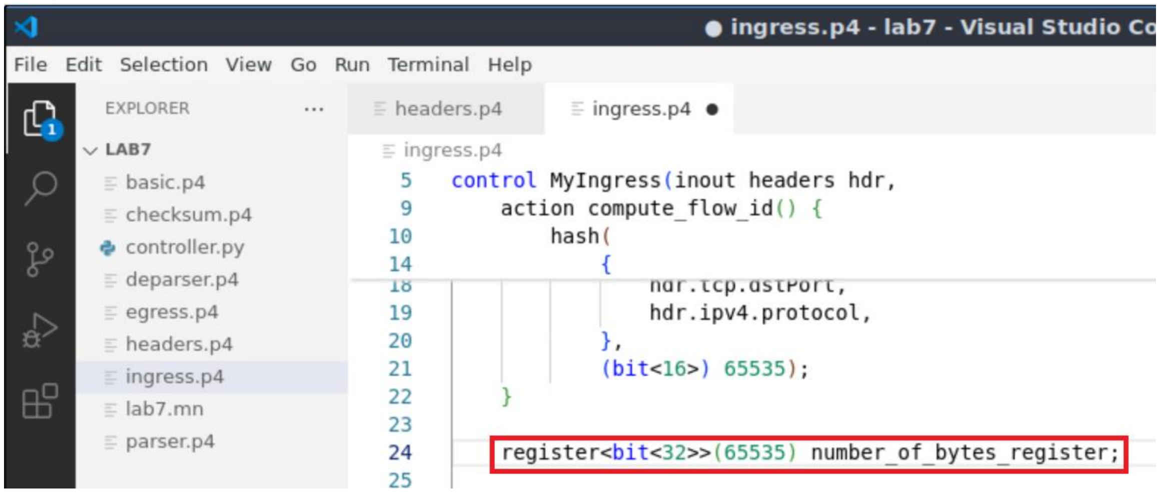
Figure 13. Defining the action `compute_flow_id`.

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

- `meta.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`, `hdr.ipv4.dstAddr`, `hdr.tcp.srcPort`, `hdr.tcp.dstPort`, `hdr.ipv4.protocol`: the data to be hashed.
- `bit<16>65535`: the maximum value produced by the hash algorithm.

Step 3. Add the following code to define a register array that will store per-flow number of bytes.

```
register<bit<32>>(65535) number_of_bytes_register;
```

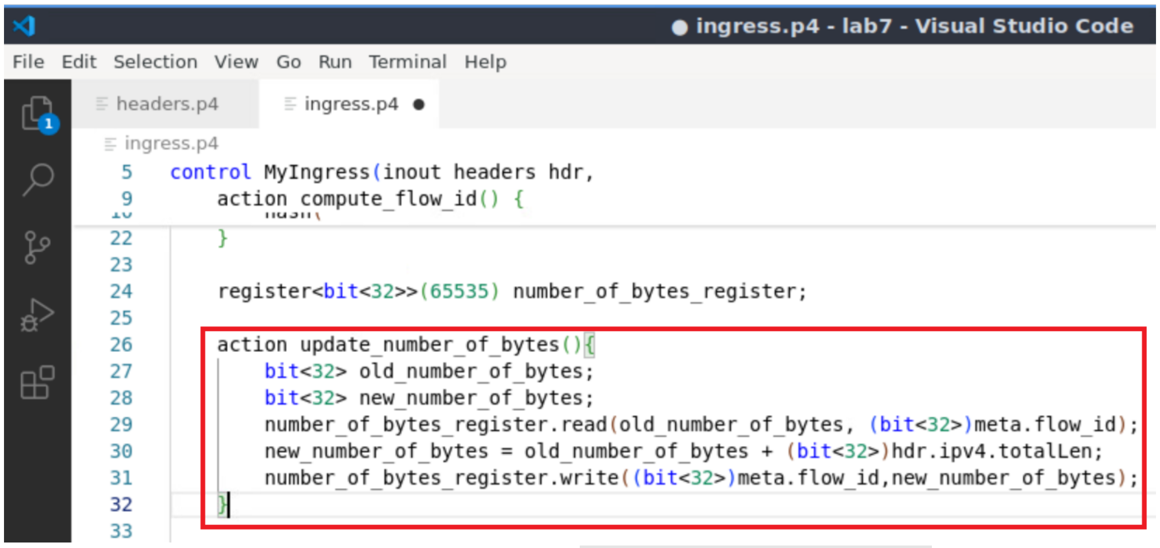


```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
        hash(
            {
            hdr.tcp.dstPort,
            hdr.ipv4.protocol,
            },
            (bit<16>) 65535);
    }

    register<bit<32>>(65535) number_of_bytes_register;
```

Figure 14. Defining a register array.

Step 4. Add the following code to define a register action.

```
action update_number_of_bytes(){
    bit<32> old_number_of_bytes;
    bit<32> new_number_of_bytes;
    number_of_bytes_register.read(old_number_of_bytes, (bit<32>)meta.flow_id);
    new_number_of_bytes = old_number_of_bytes + (bit<32>)hdr.ipv4.totalLen;
    number_of_bytes_register.write((bit<32>meta.flow_id, new_number_of_bytes);
}
```


ingress.p4 - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4

```
control MyIngress(inout headers hdr,
    action compute_flow_id() {
    }

    register<bit<32>>(65535) number_of_bytes_register;

    action update_number_of_bytes(){
        bit<32> old_number_of_bytes;
        bit<32> new_number_of_bytes;
        number_of_bytes_register.read(old_number_of_bytes, (bit<32>)meta.flow_id);
        new_number_of_bytes = old_number_of_bytes + (bit<32>)hdr.ipv4.totalLen;
        number_of_bytes_register.write((bit<32>)meta.flow_id,new_number_of_bytes);
    }
```

Figure 15. Defining the action `update_number_of_bytes`.

In the code above, the `update_number_of_bytes` action performs the following steps:

- Reading the current number of bytes: It retrieves the current number of bytes associated with the current flow from the `number_of_bytes_register` and stores this value in the variable `old_number_of_bytes`.
- Calculating the updated number of bytes: The action then adds the number of bytes from the current packet (obtained from `hdr.ipv4.totalLen`) to the value

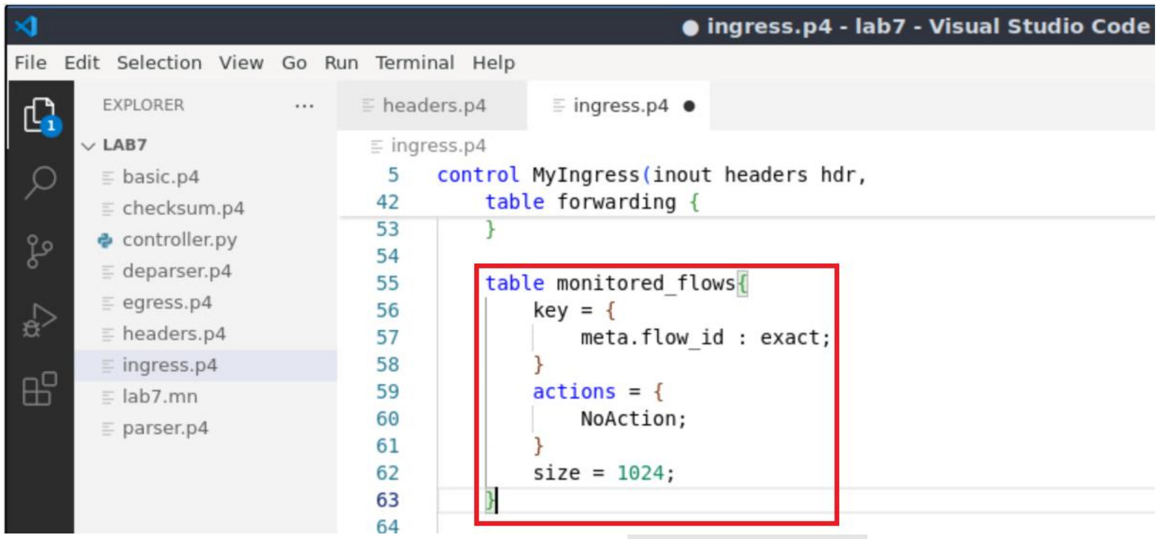
stored in `old_number_of_bytes`. The result is stored in the variable `new_number_of_bytes`.

- Writing back the updated value: Finally, the action writes the updated number of bytes (`new_number_of_bytes`) back into the `number_of_bytes_register`, ensuring to cast `meta.flow_id` into a 32-bit value because register indexes require 32-bit values.

4 Creating packet digests in P4

Step 1. Define the table `monitored_flows` by adding the following code.

```
table monitored_flows() {
    key = {
        meta.flow_id : exact;
    }
    actions = {
        NoAction;
    }
    Size = 1024;
}
```



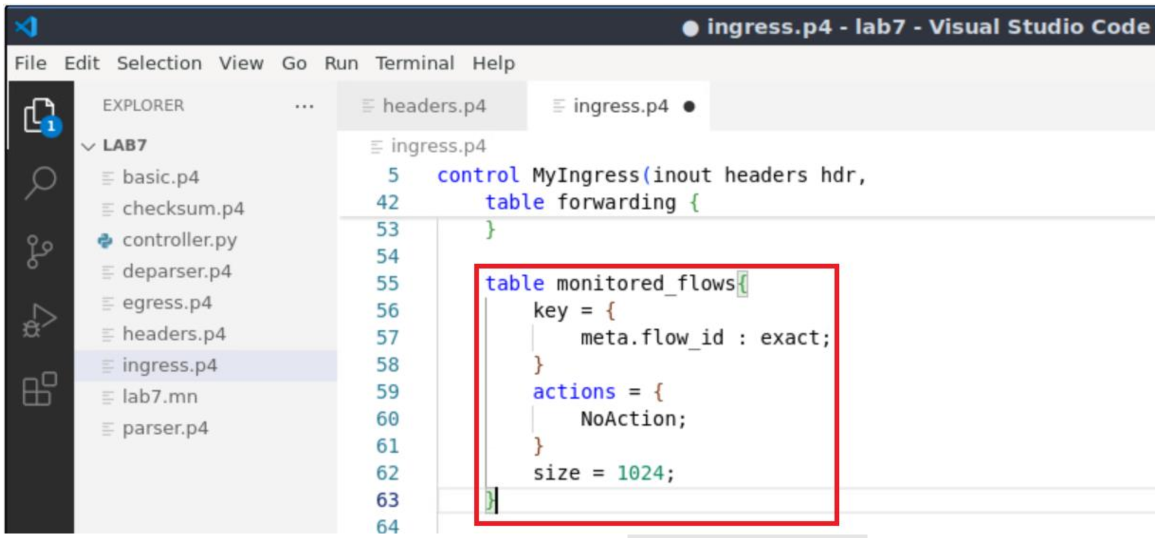
```
control MyIngress(inout headers hdr,
    table forwarding {
    }

    table monitored_flows{
        key = {
            meta.flow_id : exact;
        }
        actions = {
            NoAction;
        }
        size = 1024;
    }
```

Figure 16. Defining the table `monitored_flows`.

The table in the figure above matches the `flow_id` and executes the action `NoAction`. This table is responsible for detecting new flows by maintaining the flow IDs of previously detected flows. All flows with IDs not maintained by the table are considered new flows.

Step 2. Define the table `monitored_flows` by adding the following code.

```
table monitored_flows() {
    key = {
        meta.flow_id : exact;
    }
    actions = {
        NoAction;
    }
    Size = 1024;
}
```



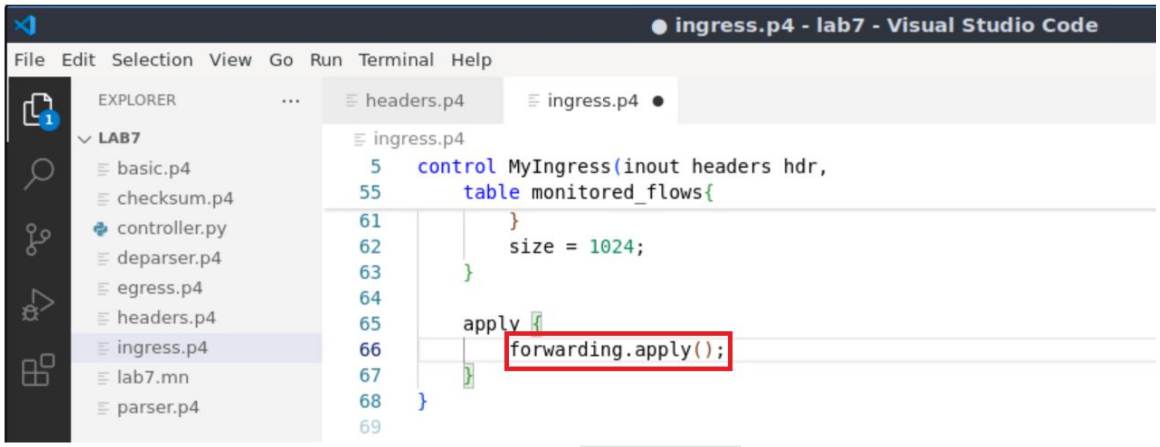
```
control MyIngress(inout headers hdr,
    table forwarding {
    }

    table monitored_flows{
        key = {
            meta.flow_id : exact;
        }
        actions = {
            NoAction;
        }
        size = 1024;
    }
```

Figure 17. Defining the table `monitored_flows`.

The table in the figure above matches the `flow_id` and executes the action `NoAction`. This table is responsible for detecting new flows by maintaining the flow IDs of previously detected flows. All flows with IDs not maintained by the table are considered new flows.

Step 3. Add the following code inside the *apply* block to apply the `forwarding`.

```
forwarding.apply();
```

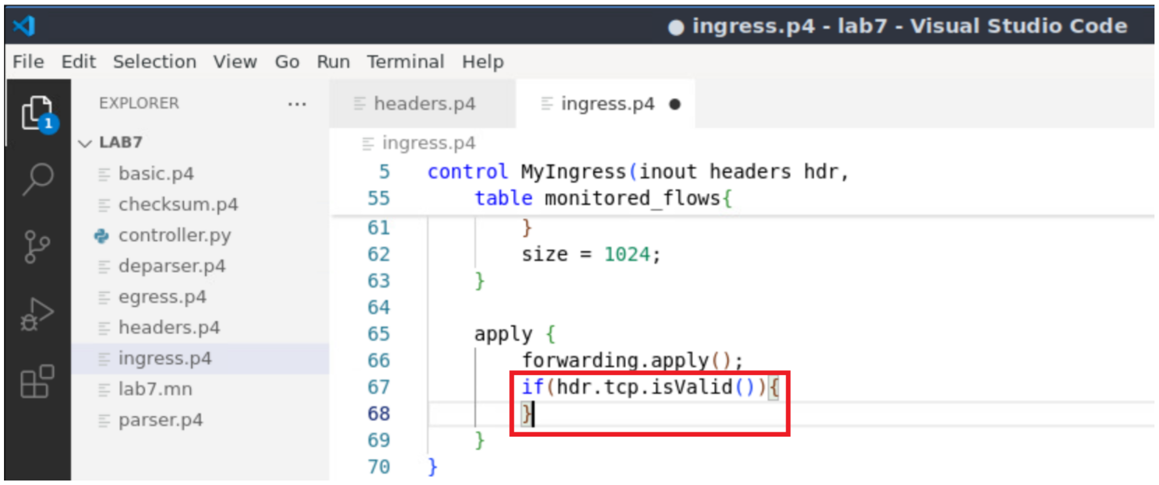


```
control MyIngress(inout headers hdr,
    table monitored_flows{
        }
        size = 1024;
    }

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

Figure 18. Applying `forwarding` table.

Step 4. Add the following code to check if the packets are TCP packets.

```
if(hdr.tcp.isValid()){
}
```

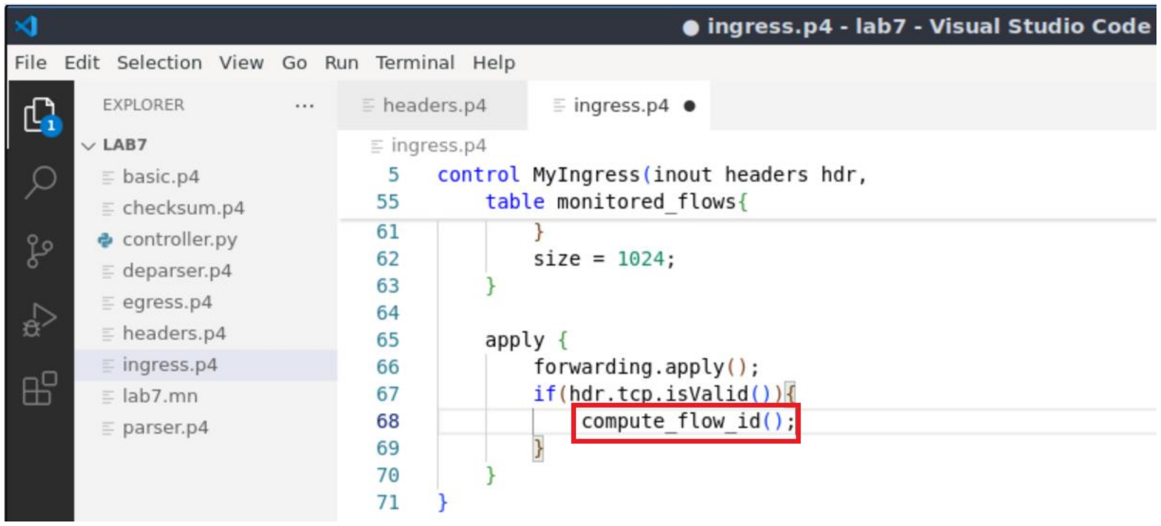


```
control MyIngress(inout headers hdr,
    table monitored_flows{
        }
        size = 1024;
    }

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

Figure 19. Checking the type of the packet.

Step 5. If a packet is a TCP packet, add the following code to compute the flow ID.

```
compute_flow_id();
```

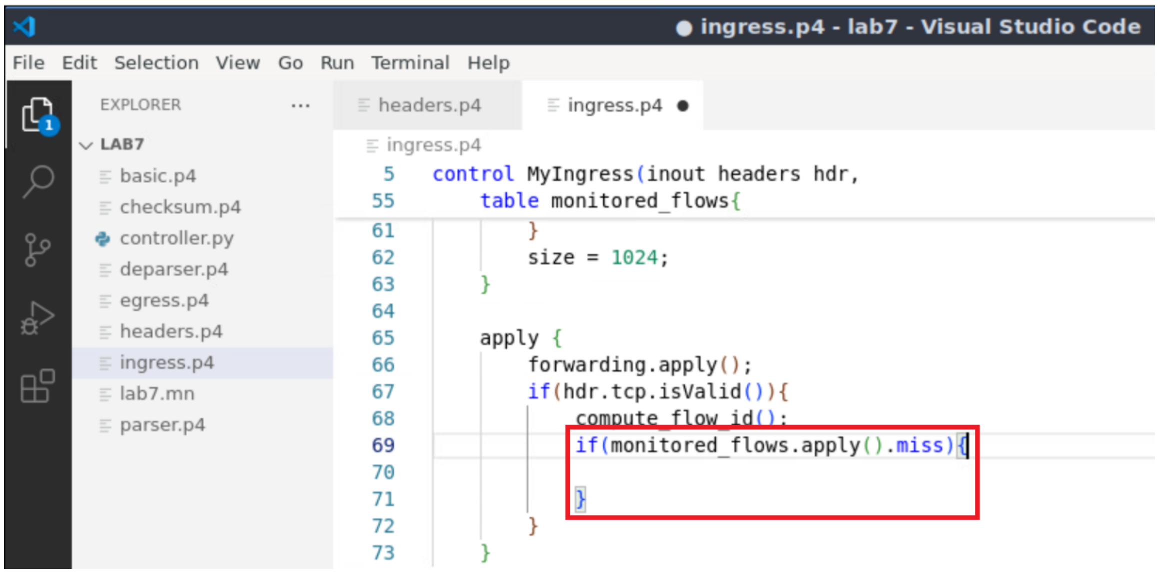


```
control MyIngress(inout headers hdr,
    table monitored_flows{
        }
        size = 1024;
    }

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

Figure 20. Computing the flow ID.

Step 6. Add the following code to identify if the flow is new. A flow with an ID not maintained by the table `monitored_flows` is a new flow.

```
if(monitored_flows.apply().miss){

}
```

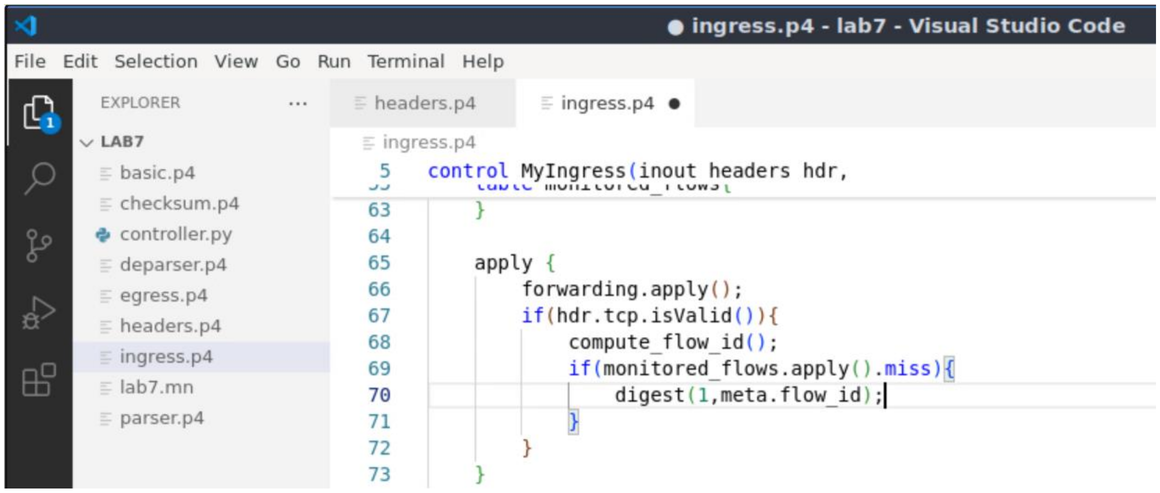


```
control MyIngress(inout headers hdr,
    table monitored_flows{
        }
        size = 1024;
    }

    apply {
        forwarding.apply();
        if(hdr.tcp.isValid()){
            compute_flow_id();
            if(monitored_flows.apply().miss){

            }
        }
    }
```

Figure 21. Identifying new flows.

Step 7. Add the following code to send a digest to the control plane if the flow is new.

```
digest(1,meta.flow_id);
```

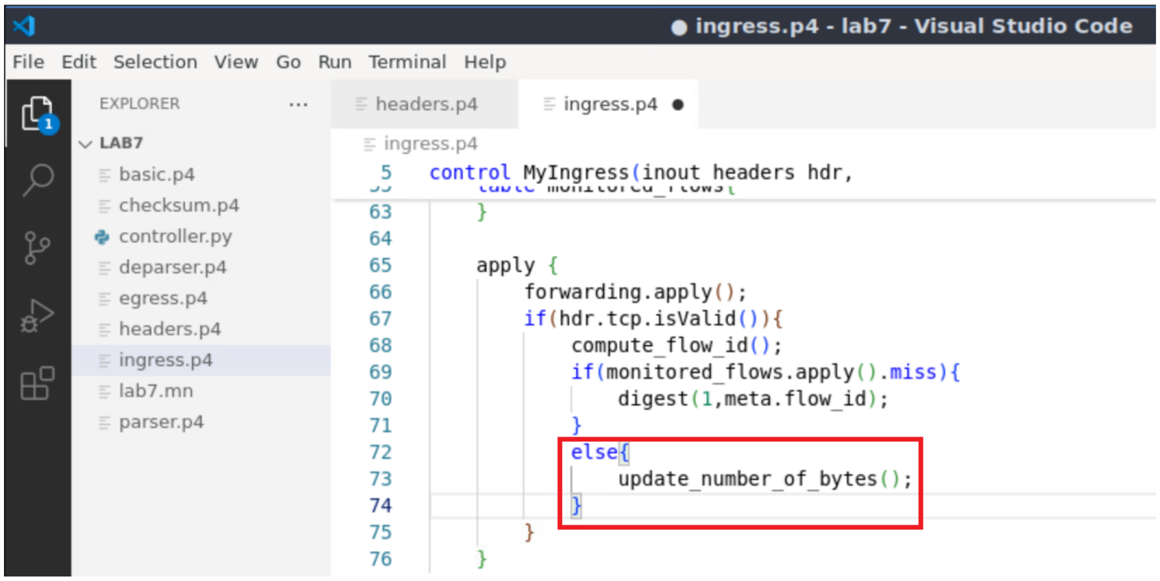


```
control MyIngress(inout headers hdr,
    }

    apply {
        forwarding.apply();
        if(hdr.tcp.isValid()){
            compute_flow_id();
            if(monitored_flows.apply().miss){
                digest(1,meta.flow_id);
            }
        }
    }
```

Figure 22. Notifying the control plane using a digest.

Step 8. Add the following code to update the flow's number of bytes if it is not new.

```
else{
    update_number_of_bytes();
}
```



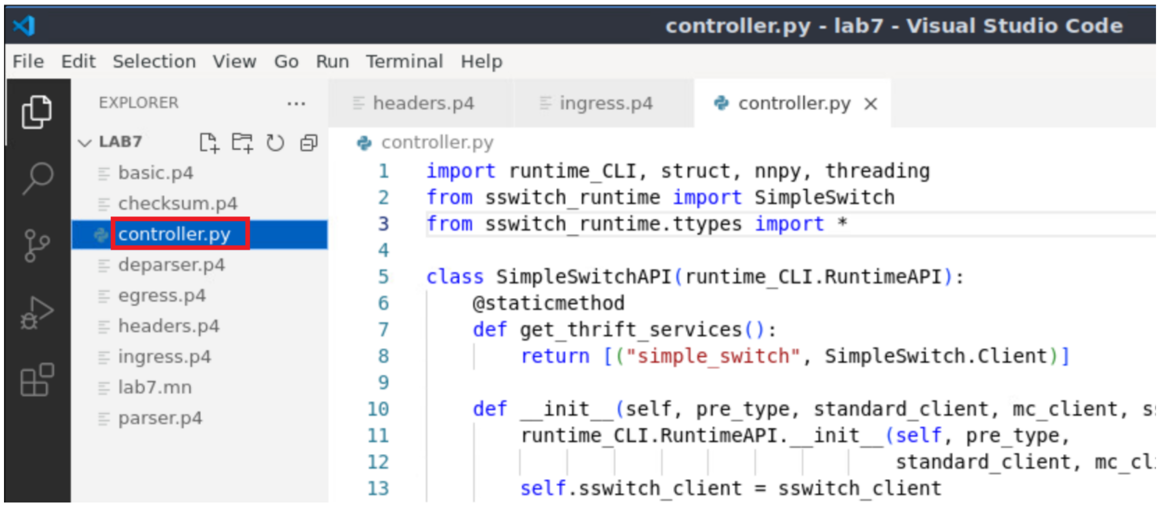
```
control MyIngress(inout headers hdr,
    }

    apply {
        forwarding.apply();
        if(hdr.tcp.isValid()){
            compute_flow_id();
            if(monitored_flows.apply().miss){
                digest(1,meta.flow_id);
            }
            else{
                update_number_of_bytes();
            }
        }
    }
```

Figure 23. Updating the number of bytes.

Step 9. Press `Ctrl+s` to save the changes.

5 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.

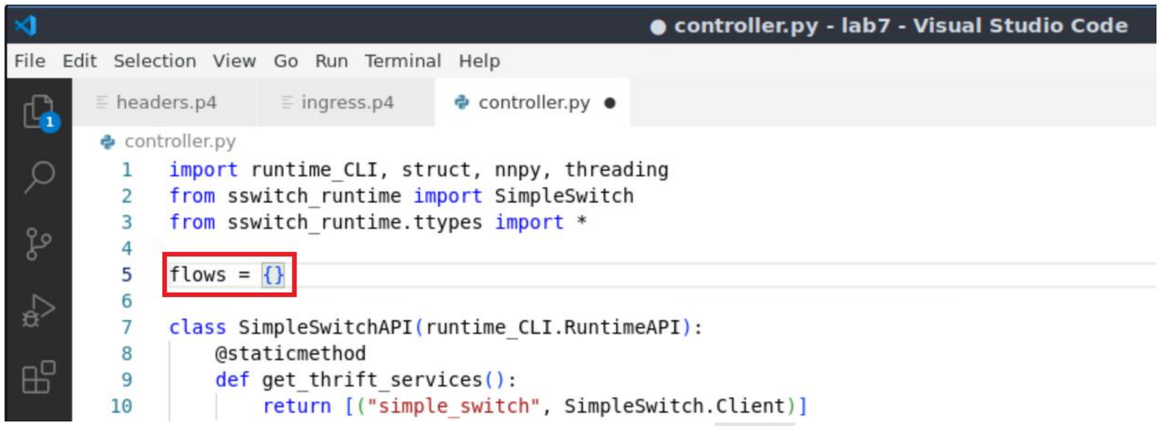


```
import runtime_CLI, struct, nnpy, threading
from sswitch_runtime import SimpleSwitch
from sswitch_runtime.ttypes import *

class SimpleSwitchAPI(runtime_CLI.RuntimeAPI):
    @staticmethod
    def get_thrift_services():
        return [("simple_switch", SimpleSwitch.Client)]

    def __init__(self, pre_type, standard_client, mc_client, s
        runtime_CLI.RuntimeAPI.__init__(self, pre_type,
                                        standard_client, mc_cl
        self.sswitch_client = sswitch_client
```

Figure 24. Inspecting the *controller.py* file.

Step 2. Define the dictionary `flows` by adding the code below. The dictionary will maintain the information of the flows.

```
flows = {}
```

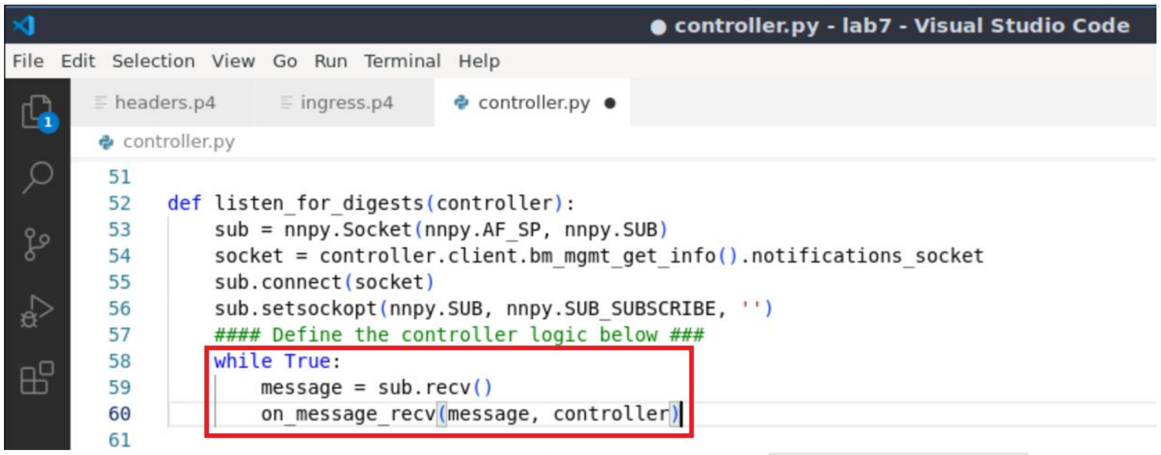

controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py

```
import runtime_CLI, struct, nnpy, threading
from sswitch_runtime import SimpleSwitch
from sswitch_runtime.ttypes import *

flows = {}

class SimpleSwitchAPI(runtime_CLI.RuntimeAPI):
    @staticmethod
    def get_thrift_services():
        return [("simple_switch", SimpleSwitch.Client)]
```

Figure 25. Defining dictionary `flows`.

Step 3. 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)
```


controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py

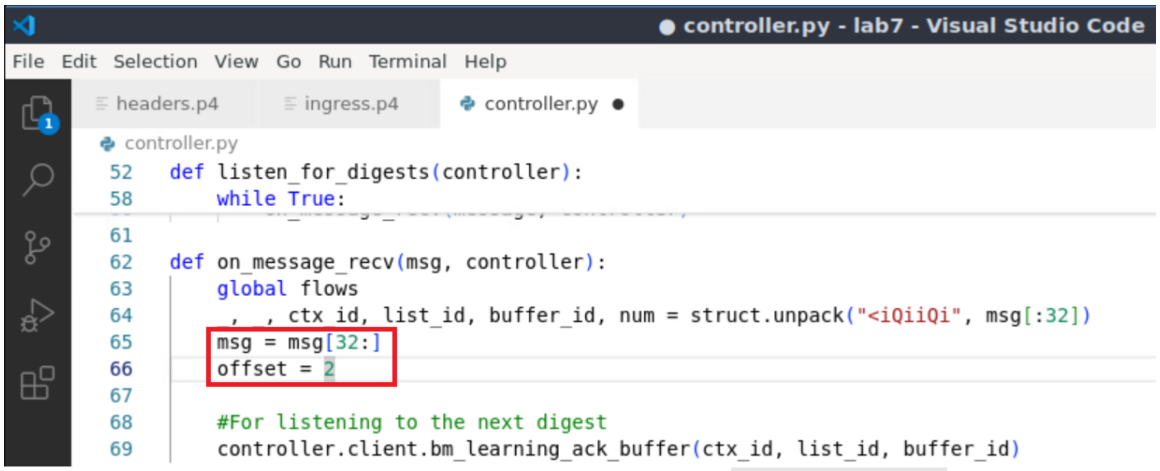
```
51
52 def listen_for_digests(controller):
53     sub = nnpy.Socket(nnpy.AF_SP, nnpy.SUB)
54     socket = controller.client.bm_mgmt_get_info().notifications_socket
55     sub.connect(socket)
56     sub.setsockopt(nnpy.SUB, nnpy.SUB_SUBSCRIBE, '')
57     #### Define the controller logic below ###
58     while True:
59         message = sub.recv()
60         on_message_recv(message, controller)
61
```

Figure 26. 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 4. Scroll down to the function `on_msg_recv` and define the following variables.

```
msg = msg[32:]
offset = 2
```


● controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py ●
controller.py

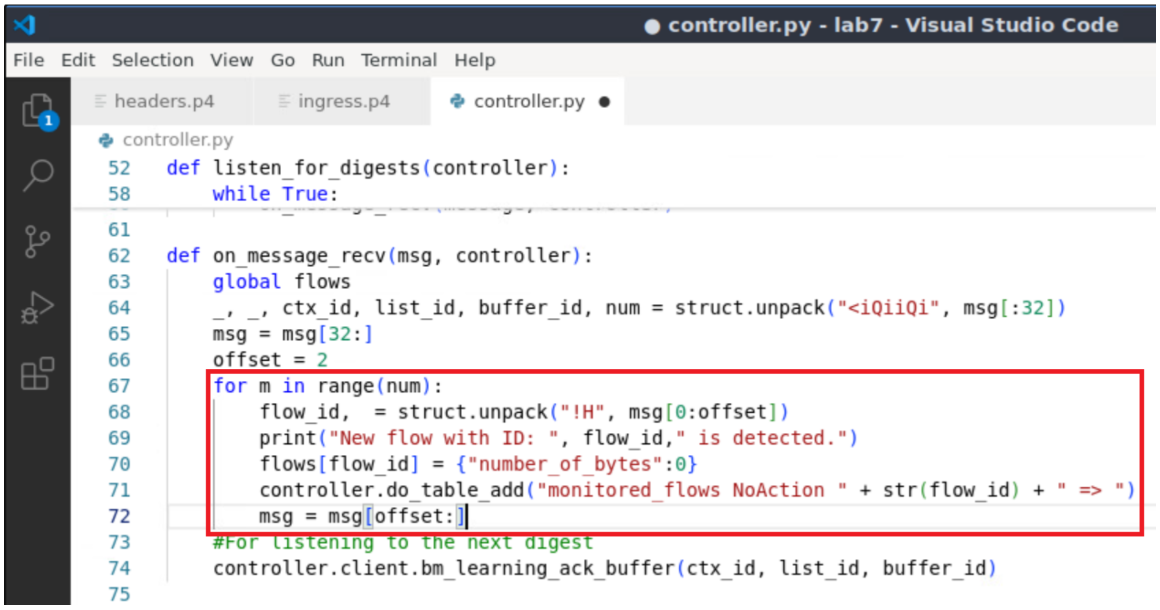
```
52  def listen_for_digests(controller):
58      while True:
61
62  def on_message_recv(msg, controller):
63      global flows
64      _, _, ctx_id, list_id, buffer_id, num = struct.unpack("<iQiiQi", msg[:32])
65      msg = msg[32:]
66      offset = 2
67
68      #For listening to the next digest
69      controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)
```

Figure 27. 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 flow ID (i.e., 16 bits).

Step 5. Define the receiving logic by adding the following code.

```
for m in range(num):
   flow_id, = struct.unpack(“!H”, msg[0:offset])
   print(“New flow with ID: ”, flow_id,” is detected.”)
   flows[flow_id] = {“number_of_bytes”:0}
   controller.do_table_add(“monitored_flows NoAction ” + str(flow_id) + “ => ”)
   msg = msg[offset:]
```


● controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py ●
controller.py

```
52  def listen_for_digests(controller):
58      while True:
61
62  def on_message_recv(msg, controller):
63      global flows
64      _, _, ctx_id, list_id, buffer_id, num = struct.unpack("<iQiiQi", msg[:32])
65      msg = msg[32:]
66      offset = 2
67      for m in range(num):
68          flow_id,  = struct.unpack("!H", msg[0:offset])
69          print("New flow with ID: ", flow_id," is detected.")
70          flows[flow_id] = {"number_of_bytes":0}
71          controller.do_table_add("monitored_flows NoAction " + str(flow_id) + " => ")
72          msg = msg[offset:]
73      #For listening to the next digest
74      controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)
75
```

Figure 28. Defining receiving logic.

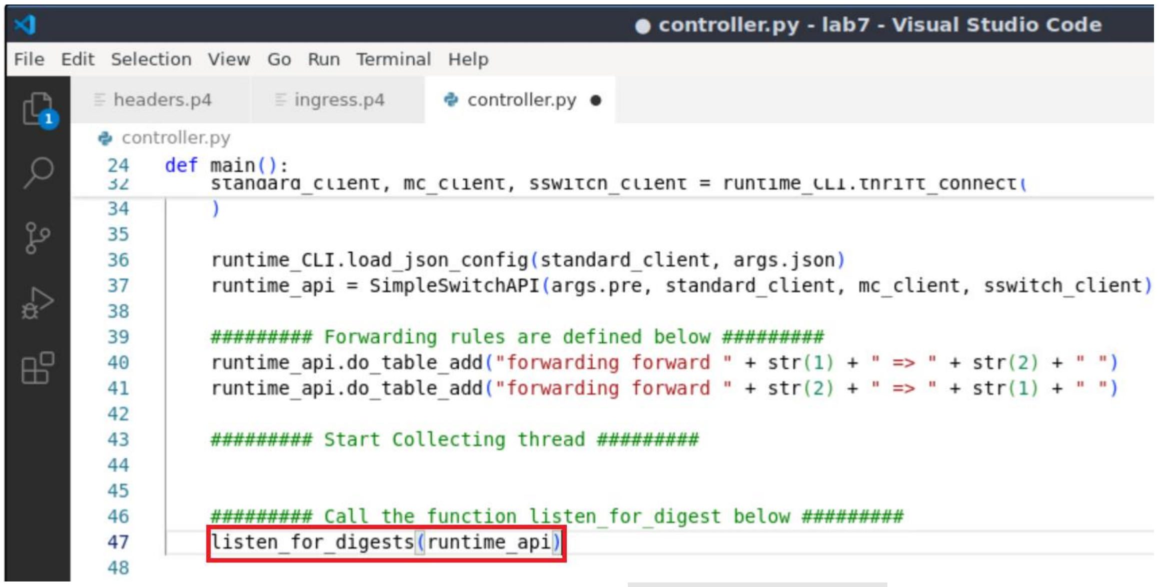
The code in the figure above is explained as follows:

- Line 68: Unpacks from the digest the flow ID.

- Line 69: Prints the received flow ID.
- Line 70: Define a new entry for the received flow in the `flows` dictionary. The flow ID (i.e., `flow_id`) is used as the key.
- Line 71: Adds an entry to the table `monitored_flows` that matches the flow ID and executes the action `NoAction`. By adding this entry, we inform the data plane that consequent packets with this flow ID do not belong to a new flow.
- Line 72: 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).

Step 6. Scroll up and call the function `listen_for_digests` from the main function by adding the line below.

```
listen_for_digests(runtime_api)
```

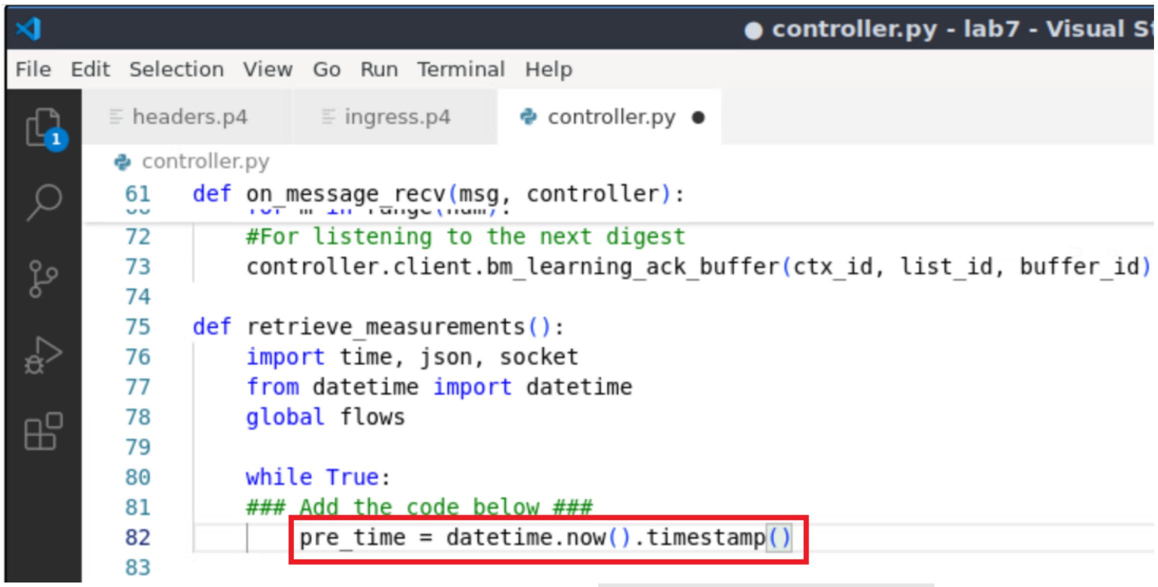

controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py
controller.py

```
def main():
    standard_client, mc_client, sswitch_client = runtime_CLI.thrift_connect(
    )

    runtime_CLI.load_json_config(standard_client, args.json)
    runtime_api = SimpleSwitchAPI(args.pre, standard_client, mc_client, sswitch_client)

    ######### Forwarding rules are defined below #########
    runtime_api.do_table_add("forwarding forward " + str(1) + " => " + str(2) + " ")
    runtime_api.do_table_add("forwarding forward " + str(2) + " => " + str(1) + " ")

    ######### Start Collecting thread #########


    ######### Call the function listen_for_digest below #########
    listen_for_digests(runtime_api)
```

Figure 29. Calling the function `listen_for_digests`.

Step 7. Scroll down and add the following code under the function `retrieve_measurements`. `pre_time` will store the current timestamp.

```
pre_time = datetime.now().timestamp()
```

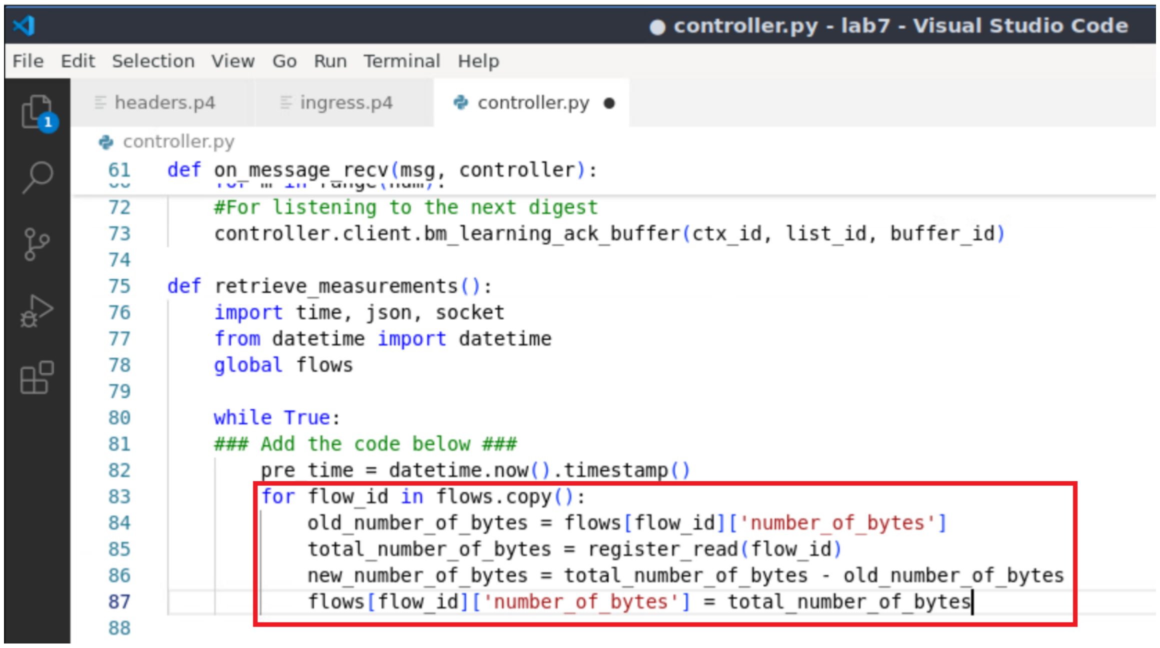

controller.py - lab7 - Visual St
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py

```
def on_message_recv(msg, controller):
        #For listening to the next digest
        controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)

def retrieve_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    while True:
    ### Add the code below ###
        pre_time = datetime.now().timestamp()
```

Figure 30. Defining the function `retrieve_measurements`.

Step 8. Add the code below to iterate over the monitored flows and retrieve the corresponding number of bytes.

```
for flow_id in flows.copy():
    old_number_of_bytes = flows[flow_id][‘number_of_bytes’]
    total_number_of_bytes = register_read(flow_id)
    new_number_of_bytes = total_number_of_bytes – old_number_of_bytes
    flows[flow_id][‘number_of_bytes’] = total_number_of_bytes
```

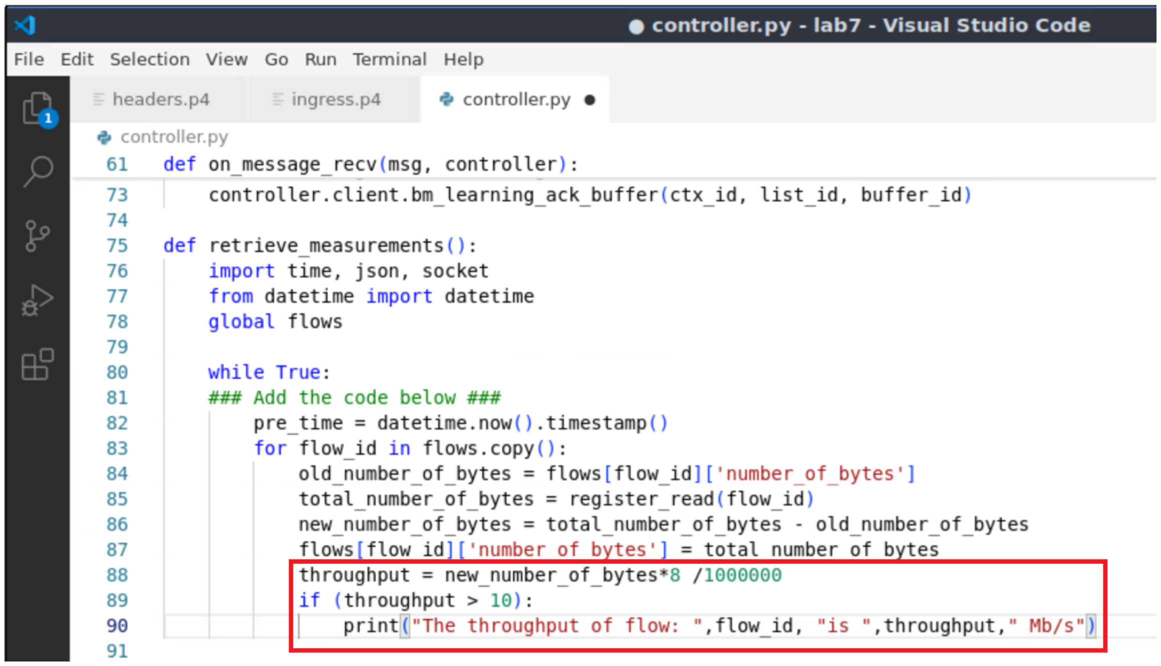

controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py

```
def on_message_recv(msg, controller):
        #For listening to the next digest
        controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)

def retrieve_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    while True:
    ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
```

Figure 31. Retrieving per-flow number of bytes.

Step 9. Add the code below to calculate and print the throughput. The `if` statement only prints the throughput if it is larger than 10 Mb/s.

```
thoughput = new_number_of_bytes*8/1000000
if(throughput > 10):
    print(“The throughput of flow: “,flow_id, “ is ”, throughput, “Mb/s”)
```

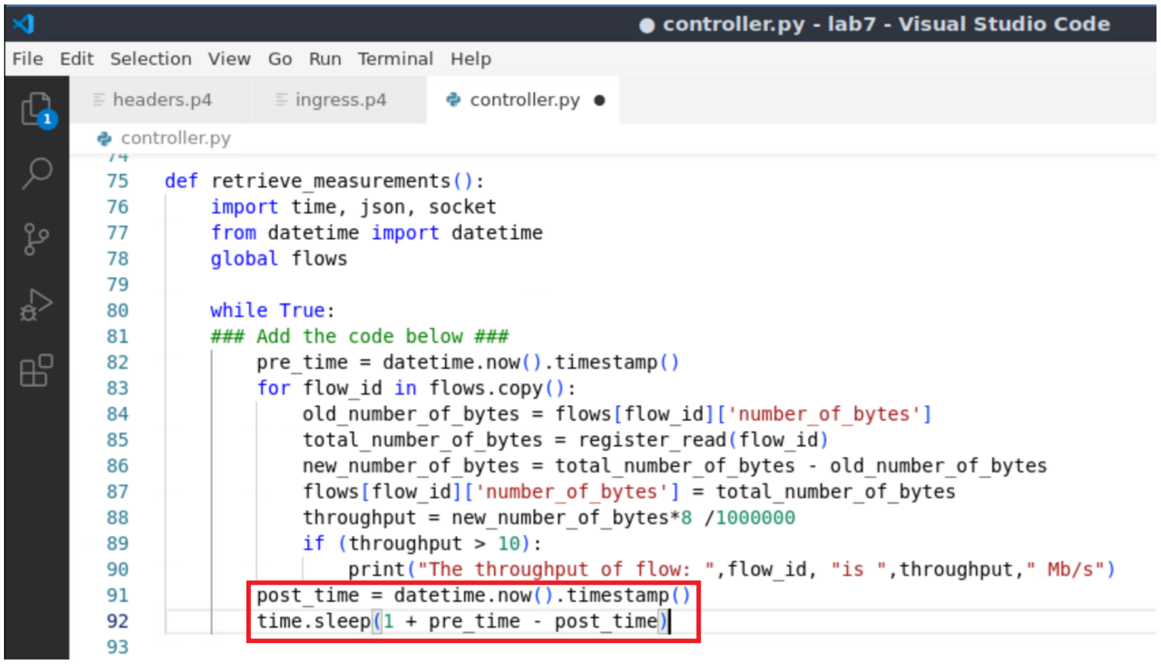

controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py
controller.py

```
def on_message_recv(msg, controller):
    controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)

def retrieve_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    while True:
    ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
            throughput = new_number_of_bytes*8 /1000000
            if (throughput > 10):
                print("The throughput of flow: ",flow_id, "is ",throughput," Mb/s")
```

Figure 32. Calculating and printing the throughput.

Step 10. Add the code below to retrieve the number of bytes once per second.

```
post_time = datetime.now().timestamp()
tmie.sleep(1 + pre_time – post_time)
```

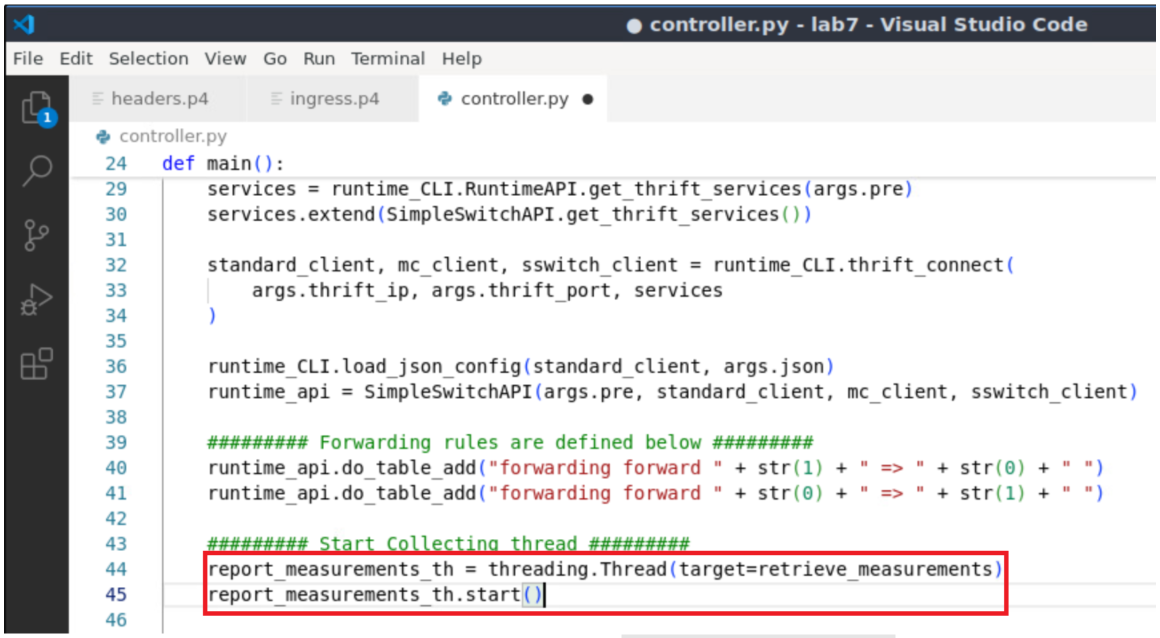

controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py
controller.py

```
def retrieve_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    while True:
    ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
            throughput = new_number_of_bytes*8 /1000000
            if (throughput > 10):
                print("The throughput of flow: ",flow_id, "is ",throughput," Mb/s")
        post_time = datetime.now().timestamp()
        time.sleep(1 + pre_time - post_time)
```

Figure 33. Setting the querying period to 1 second.

Step 11. Scroll up to the main function then define and start a new thread to call the function `retrieve_measurements`.

```
report_measurements_th = threading.Thread(targer=retrieve_measurements)
report_measurements_th.start()
```


● controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
headers.p4
ingress.p4
controller.py ●
controller.py

```
def main():
    services = runtime_CLI.RuntimeAPI.get_thrift_services(args.pre)
    services.extend(SimpleSwitchAPI.get_thrift_services())

    standard_client, mc_client, sswitch_client = runtime_CLI.thrift_connect(
        args.thrift_ip, args.thrift_port, services
    )

    runtime_CLI.load_json_config(standard_client, args.json)
    runtime_api = SimpleSwitchAPI(args.pre, standard_client, mc_client, sswitch_client)

    ######### Forwarding rules are defined below #########
    runtime_api.do_table_add("forwarding forward " + str(1) + " => " + str(0) + " ")
    runtime_api.do_table_add("forwarding forward " + str(0) + " => " + str(1) + " ")

    ######### Start Collecting thread #########
    report_measurements_th = threading.Thread(target=retrieve_measurements)
    report_measurements_th.start()
```

Figure 34. Calling the function `report_measurements`.

Step 12. Press `Ctrl+s` to save the changes.

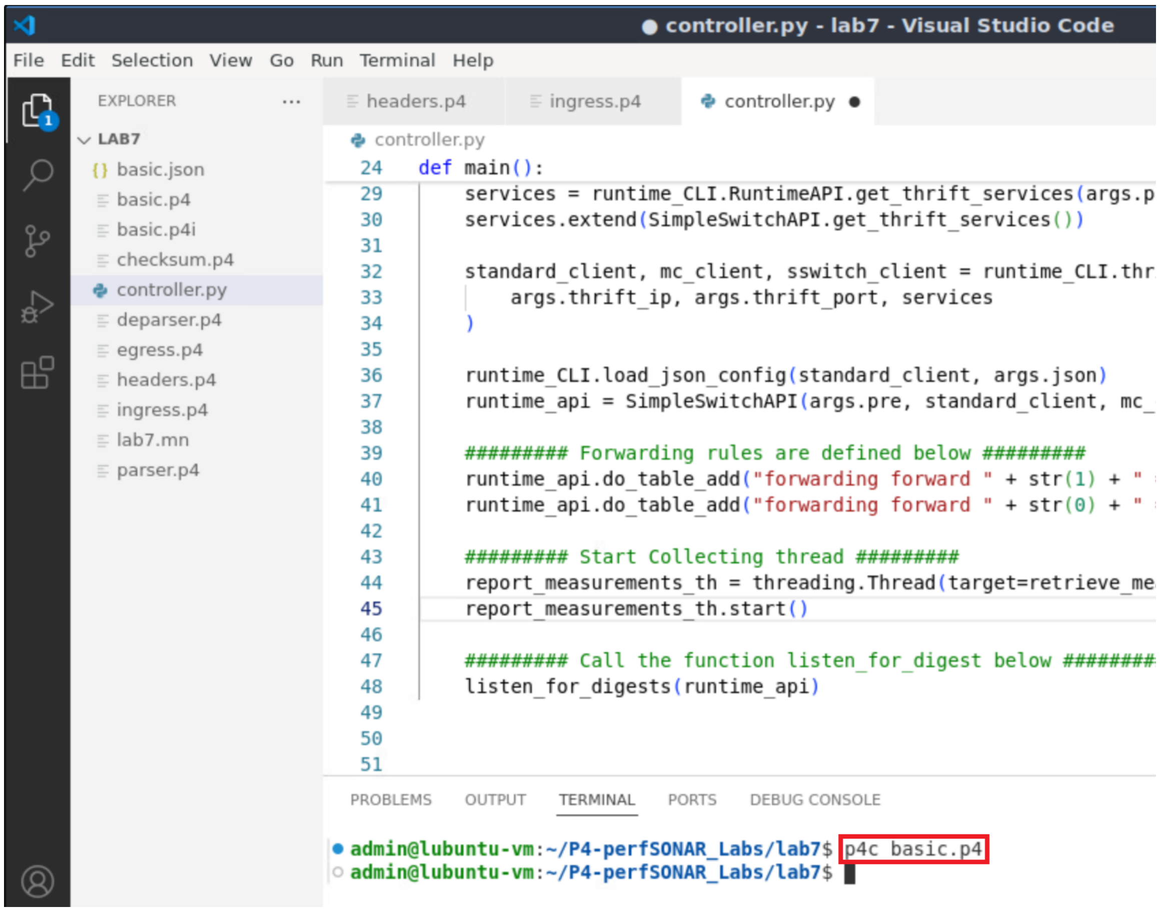
6 Loading the P4 program

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.

6.1 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
```



controller.py - lab7 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB7
basic.json
basic.p4
basic.p4i
checksum.p4
controller.py
deparser.p4
egress.p4
headers.p4
ingress.p4
lab7.mn
parser.p4
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

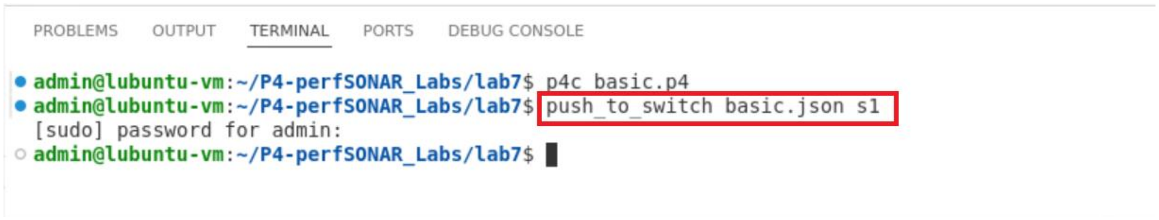
```
def main():
    services = runtime_CLI.RuntimeAPI.get_thrift_services(args.p
    services.extend(SimpleSwitchAPI.get_thrift_services())

    standard_client, mc_client, sswitch_client = runtime_CLI.thr
        args.thrift_ip, args.thrift_port, services
    )

    runtime_CLI.load_json_config(standard_client, args.json)
    runtime_api = SimpleSwitchAPI(args.pre, standard_client, mc_

    ######### Forwarding rules are defined below #########
    runtime_api.do_table_add("forwarding forward " + str(1) + "
    runtime_api.do_table_add("forwarding forward " + str(0) + "

    ######### Start Collecting thread #########
    report_measurements_th = threading.Thread(target=retrieve_me
    report_measurements_th.start()

    ######### Call the function listen_for_digest below ########
    listen_for_digests(runtime_api)
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ 
```

Figure 35. 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
```

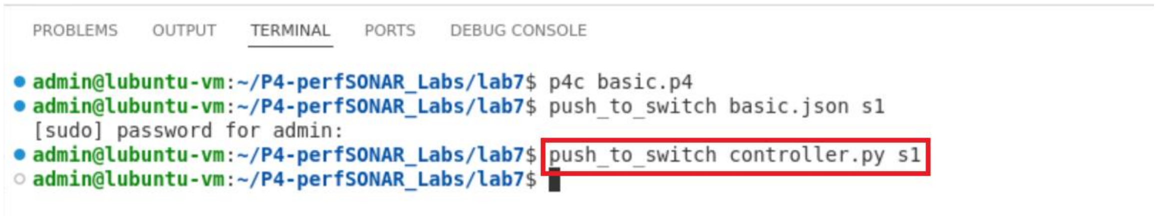


```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ 
```

Figure 36. 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
```



```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ p4c basic.p4
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$ push_to_switch controller.py s1
admin@lubuntu-vm:~/P4-perfSONAR_Labs/lab7$
```

Figure 37. Pushing the *controller.py* file to switch s1.

6.2 Verifying the configuration

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

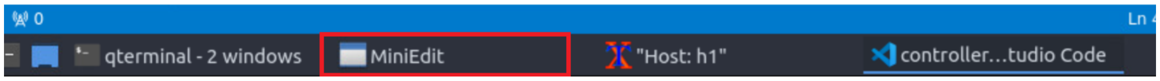


Figure 38. Maximizing the MiniEdit window.

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

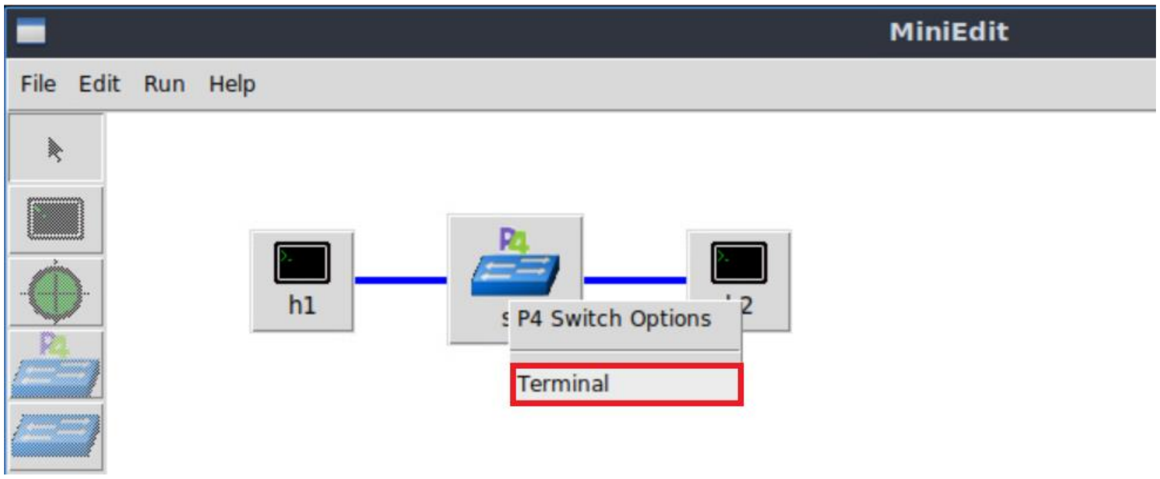
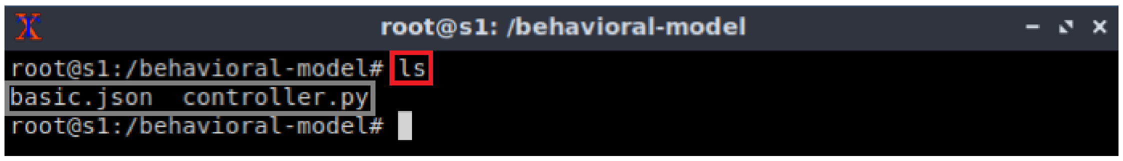


Figure 39. Starting the terminal on the switch.

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

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

```
ls
```



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

Figure 40. Displaying the contents of the current directory in the switch s1.

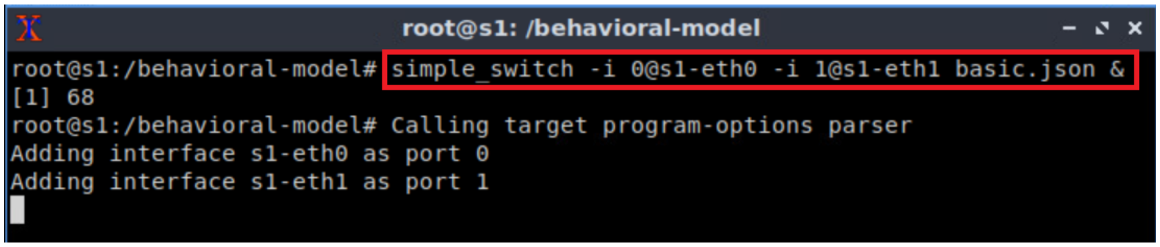
The figure above shows that the switch contains the *basic.json* and *controller.py* files that were pushed after compiling the P4 program and creating the controller application.

7 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.

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

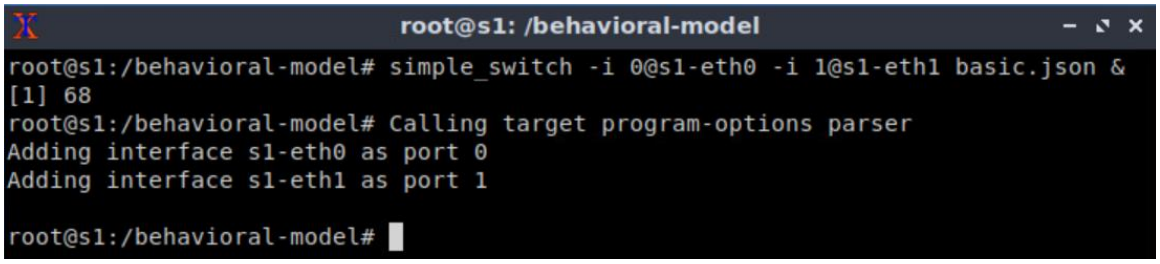
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
[1] 68
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 41. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 2. In switch s1 terminal, press *Enter* to return the CLI.



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
[1] 68
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 42. Returning to switch s1 CLI.

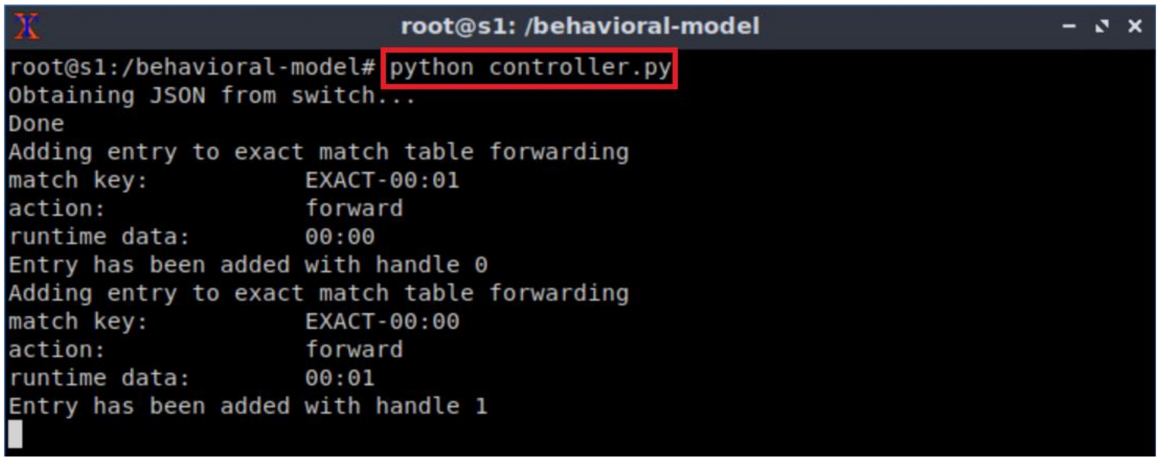
8 Testing and verifying the P4 program

This section shows the steps to run a controller and observe the pre-flow throughput monitored by the application.

8.1 Starting the controller application

Step 1. In switch s1 terminal, start the controller by running the following command.

```
python controller.py
```



```
root@s1:/behavioral-model# python controller.py
Obtaining JSON from switch...
Done
Adding entry to exact match table forwarding
match key:           EXACT-00:01
action:              forward
runtime data:        00:00
Entry has been added with handle 0
Adding entry to exact match table forwarding
match key:           EXACT-00:00
action:              forward
runtime data:        00:01
Entry has been added with handle 1
```

Figure 43. Starting the controller in switch s1.

8.2 Starting data transfer between h1 and h2

Step 1. Right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

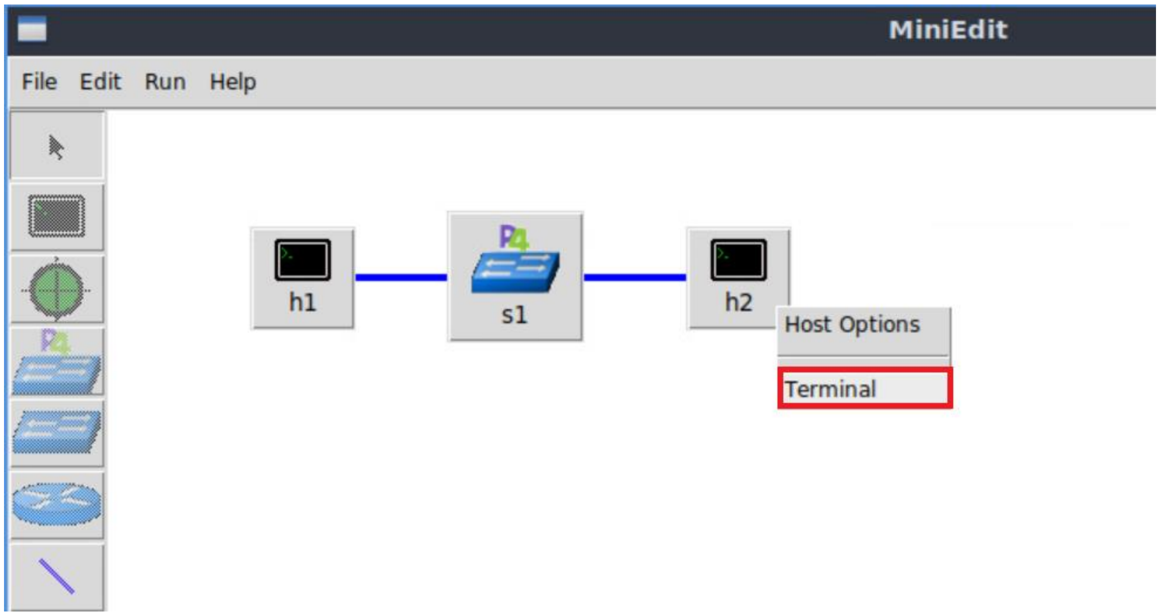
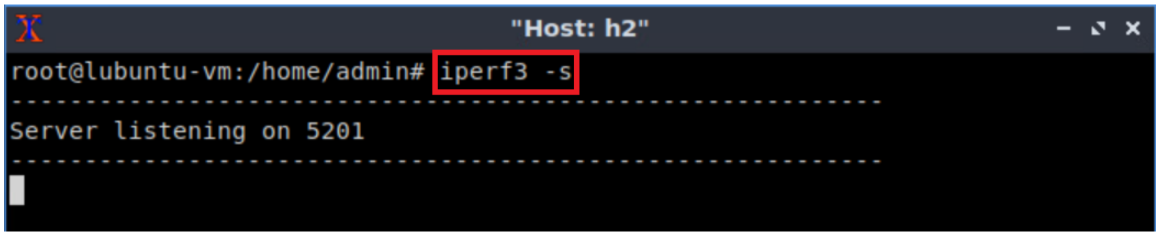


Figure 44. Opening a terminal on host h2.

Step 2. On host h2's terminal, type the following command to start Iperf3 server. `iperf3` tool is used to perform network throughput tests. `-s` option specifies that h2 will be acting as the server.

```
iperf3 -s
```

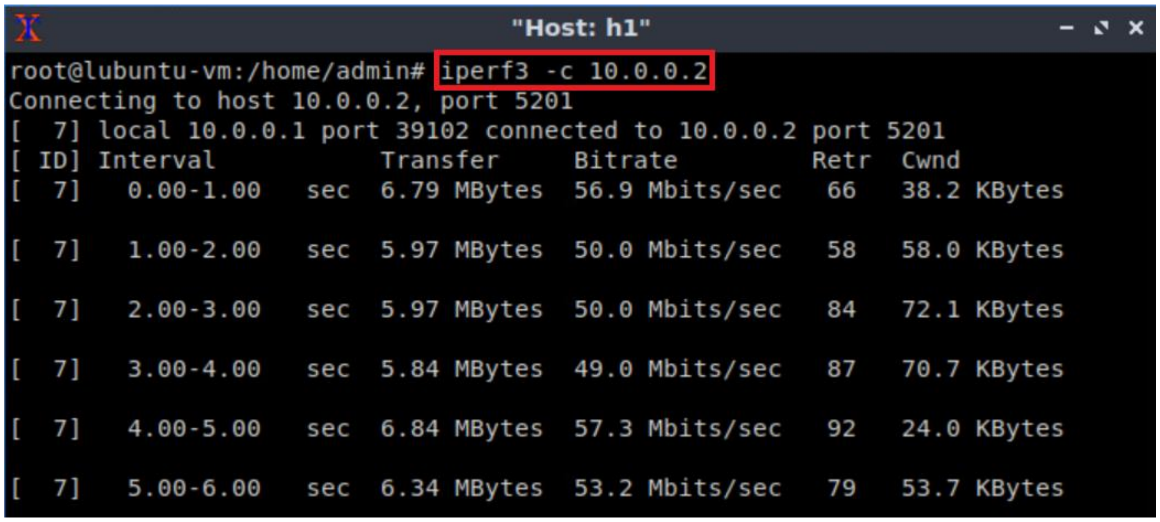


```
root@lubuntu-vm:/home/admin# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 45. Starting Iperf3 server on h2.

Step 3. On host h1's terminal, type the following command to start data transfer with h2. `-c` option specifies that h1 will be acting as the client.

```
iperf3 -c 10.0.0.2
```

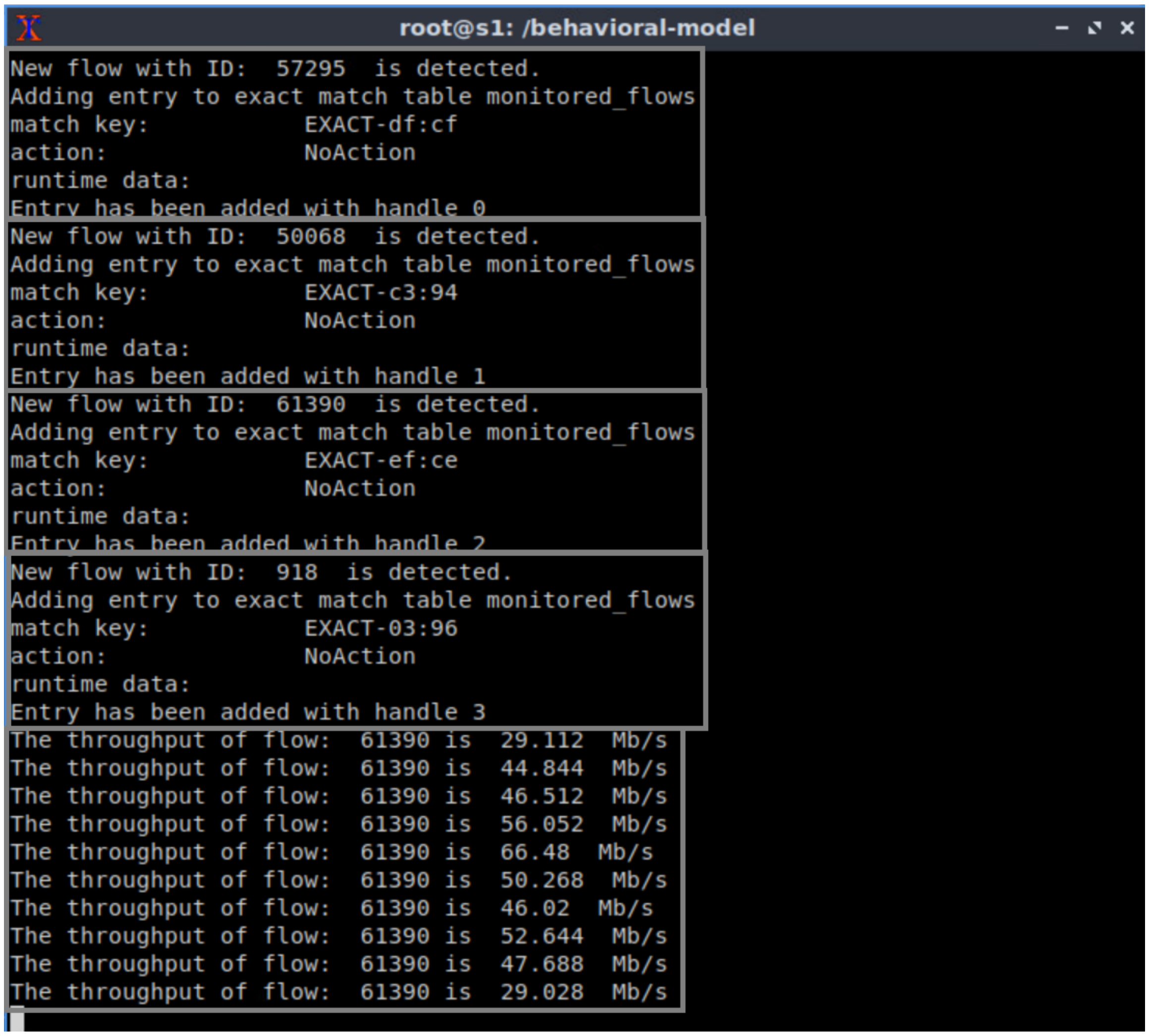


```
root@lubuntu-vm:/home/admin# iperf3 -c 10.0.0.2
Connecting to host 10.0.0.2, port 5201
[  7] local 10.0.0.1 port 39102 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  7]   0.00-1.00   sec  6.79 MBytes  56.9 Mbits/sec   66   38.2 KBytes

[  7]   1.00-2.00   sec  5.97 MBytes  50.0 Mbits/sec   58   58.0 KBytes

[  7]   2.00-3.00   sec  5.97 MBytes  50.0 Mbits/sec   84   72.1 KBytes

[  7]   3.00-4.00   sec  5.84 MBytes  49.0 Mbits/sec   87   70.7 KBytes

[  7]   4.00-5.00   sec  6.84 MBytes  57.3 Mbits/sec   92   24.0 KBytes

[  7]   5.00-6.00   sec  6.34 MBytes  53.2 Mbits/sec   79   53.7 KBytes
```

Figure 46. Sending data between h1 and h2.

8.3 Inspecting the measurements retrieved by the control plane

Step 1. Navigate back to switch s1 terminal and inspect the output.



```
New flow with ID:  57295  is detected.
Adding entry to exact match table monitored_flows
match key:           EXACT-df:cf
action:              NoAction
runtime data:
Entry has been added with handle 0
New flow with ID:  50068  is detected.
Adding entry to exact match table monitored_flows
match key:           EXACT-c3:94
action:              NoAction
runtime data:
Entry has been added with handle 1
New flow with ID:  61390  is detected.
Adding entry to exact match table monitored_flows
match key:           EXACT-ef:ce
action:              NoAction
runtime data:
Entry has been added with handle 2
New flow with ID:  918  is detected.
Adding entry to exact match table monitored_flows
match key:           EXACT-03:96
action:              NoAction
runtime data:
Entry has been added with handle 3
The throughput of flow:  61390 is  29.112  Mb/s
The throughput of flow:  61390 is  44.844  Mb/s
The throughput of flow:  61390 is  46.512  Mb/s
The throughput of flow:  61390 is  56.052  Mb/s
The throughput of flow:  61390 is  66.48  Mb/s
The throughput of flow:  61390 is  50.268  Mb/s
The throughput of flow:  61390 is  46.02  Mb/s
The throughput of flow:  61390 is  52.644  Mb/s
The throughput of flow:  61390 is  47.688  Mb/s
The throughput of flow:  61390 is  29.028  Mb/s
```

Figure 47. Inspecting the controller's log in switch s1.

The figure above shows that the 4 new flows are detected. This occurs because the packets from h1 to h2 are considered one flow and the packets from h2 to h1 are considered another flow. Moreover, the Iperf3 tool has two channels per connection, one data channel, and one control channel[7]. Thus, we have a total of 4 flows.

This concludes lab 7. 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.
7. Bruce A. Mah. *"The Iperf3 Protocol and State Machine*." [Online]. Available: https://tinyurl.com/54b6mbeu.

P4-PERFSONAR LAB SERIES

Lab 8: Collecting P4 measurements using perfSONAR's archiver

Document Version: 07-14-2024

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

Contents

Overview 3
Objectives 3
Lab settings 3
Lab roadmap 3
1 Introduction to perfSONAR's archiving layer 4
1.1 Lab scenario 4
2 Lab topology 5
2.1 Starting the end hosts 7
3 Configuring new pipeline in Logstash 8
3.1 Inspecting pScheduler pipeline 8
3.2 Creating P4 pipeline 11
3.3 Configuring P4 pipeline 12
4 Configuring the control plane of the switch 15
4.1 Loading the environment 16
4.2 Inspecting the controller 17
4.3 Configuring the connection with Logstash 18
4.4 Configuring the data retrieving and reporting process 20
5 Loading the P4 program 24
5.1 Compiling and loading the P4 program to switch s1 24
5.2 Verifying the configuration 25
5.3 Configuring switch s1 26
6 Connecting Grafana to OpenSearch 27
6.1 Running Grafana 27
6.2 Accessing Grafana 28
6.3 Configuring Grafana's data source 29
6.4 Configuring an OpenSearch data source 31
7 Testing and verifying the application 35
7.1 Starting the controller application 35
7.2 Starting data transfer between h1 and h2 36
7.3 Displaying the measurement results in a dashboard 37
References 43

Overview

This lab demonstrates how to collect the measurements calculated by P4 using a perfSONAR archiver. The user will start by defining a new pipeline in Logstash to ingest and process the measurements received from the switch. Logstash is a data processing pipeline that accepts data from multiple sources, provides flexible data processing, and forwards the processed data to the final destination. The user will then configure the control plane of the switch to connect and send the measurements to Logstash. After that, the user will configure Grafana to display the measurements.

Objectives

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

1. Define the workflow of the archiving layer of perfSONAR.
2. Configure and append new pipelines to Logstash.
3. Connect the control plane to Logstash.
4. Connect Logstash to OpenSearch.
5. Visualizing the measurements collected by P4 in a Grafana dashboard.

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 perfSONAR's archiving layer.
2. Section 2: Lab topology.
3. Section 3: Configuring new pipeline in Logstash.
4. Section 4: Configuring the control plane of the switch.
5. Section 5: Loading the P4 program.
6. Section 6: Connecting Grafana to OpenSearch.
7. Section 7: Testing and verifying the application.

1 Introduction to perfSONAR's archiving layer

perfSONAR's archiving layer is responsible for storing time-series data about the tests that were performed. This layer adopts the plug-in architecture, where the user can choose which software to use in this layer. The default archiver used by perfSONAR is the combination of Logstash and OpenSearch[1]. Logstash is a data processing pipeline that accepts data from multiple sources, provides flexible data processing, and forwards the processed data to the final destination[2]. OpenSearch is a search and analytical suite that supports data storage, real-time application monitoring, and log analytics[3].

The APIs used by perfSONAR layers to communicate are REST-based. REST APIs communicate via the HTTP protocol. When a test is performed in perfSONAR, the tools layer forwards the measurements to the scheduling layer. The scheduling layer formats the data in JavaScript Object Notation (JSON) and forwards it to Logstash. The input plugin used to connect the scheduling layer and Logstash is the HTTP plugin. Because the tools in perfSONAR produce different measurements, Logstash has a dedicated input-output path for each tool. After receiving a report from the input plugin, Logstash processes it using its filters such that each filter's output is the next filter's input. The processed report is then forwarded to the OpenSearch output plugin. The plugin appends the metadata required by OpenSearch to ingest and store the report.

1.1 Lab scenario

In this lab, the user will connect the P4 switch with Logstash to store the measurements. As depicted in Figure 1, the user will utilize the TCP input plugin. The plugin requires configuring the IP address and the port of the source application. The user will also configure a new pipeline in Logstash responsible for processing and storing the measurements received from the switch. The control plane will perform the necessary processing on the retrieved measurements from the data plane, eliminating the need for any new filters in Logstash. The measurements received from the P4 switch are directly forwarded from the TCP input plugin to the OpenSearch output plugin. The output plugin then adds the necessary metadata and stores the reports in OpenSearch. Finally, the user will connect Grafana to OpenSearch and visualize the collected measurements.

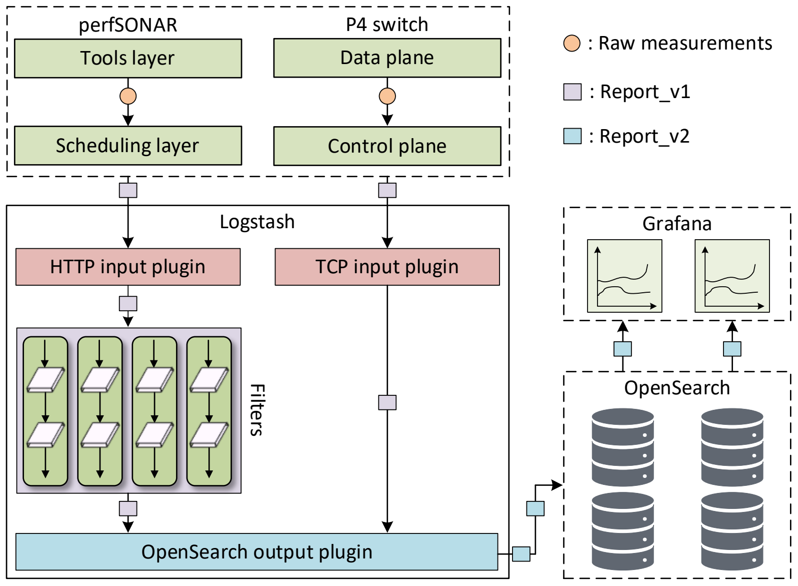


Figure 1. Lab scenario. The control plane formats the raw measurements extracted from the data plane to create structured reports (Report_v1). Logstash receives the reports via the TCP input plugin, adds the metadata required by the OpenSearch database, and forwards the new reports (Report_v2) to OpenSearch.

2 Lab topology

Let's get started by loading a simple Mininet topology using MiniEdit. The topology comprises two end hosts a P4 switch, and a perfSONAR's archiver. The control plane of the switch uses interface eth0 to send the measurements to the archiver.

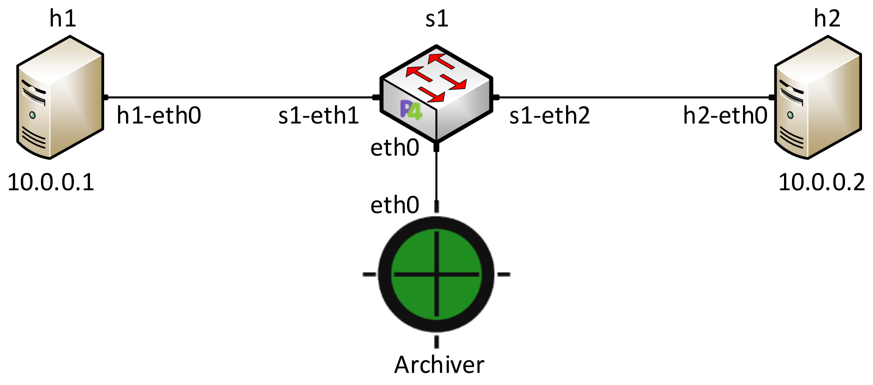


Figure 2. Lab topology.

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

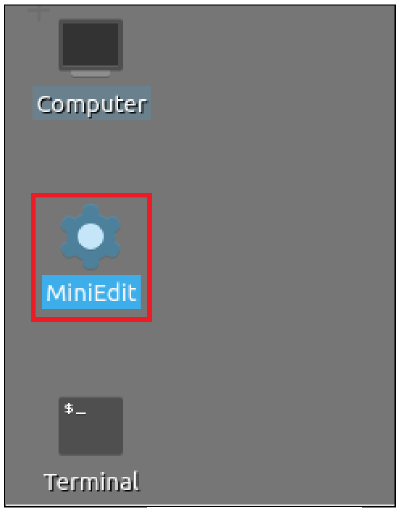
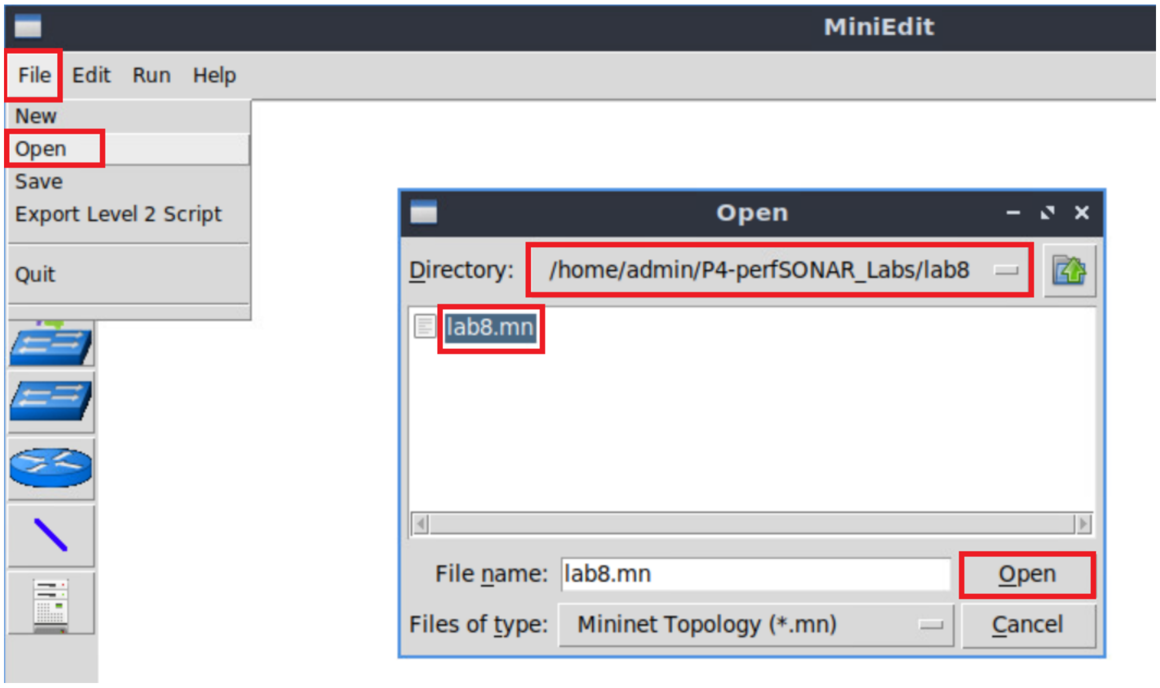


Figure 3. MiniEdit shortcut.

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



Directory: /home/admin/P4-perfSONAR_Labs/lab8

lab8.mn

File name: lab8.mn

Files of type: Mininet Topology (*.mn)

Figure 4. MiniEdit's *Open* dialog.

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

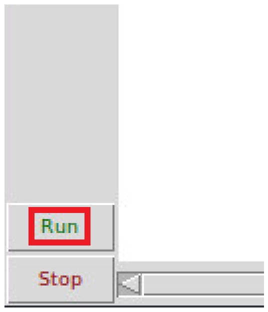


Figure 5. Running the emulation.

2.1 Starting 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.

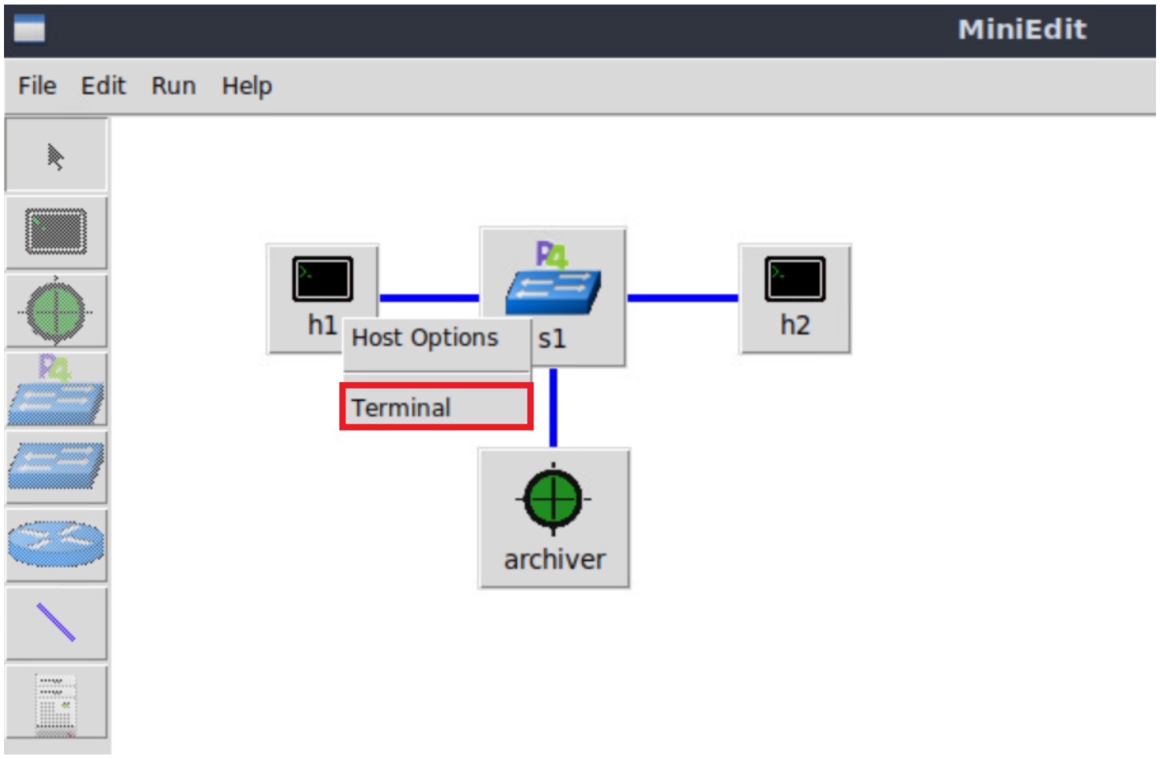
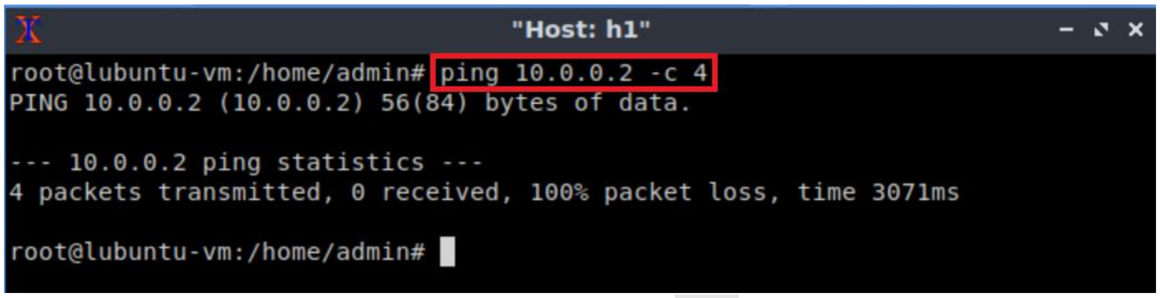


Figure 6. Opening a terminal on host h1.

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

```
ping 10.0.0.2 -c 4
```



```
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.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3071ms

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

Figure 7. 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 Configuring new pipeline in Logstash

In this section, you will implement a new Logstash pipeline to process the measurements arriving from the P4 switch. The user will first inspect the structure of the pipeline used by the pScheduler layer. You will then configure a pipeline for P4.

3.1 Inspecting pScheduler pipeline

Step 1. Right-click on the archiver and select *Terminal*. This opens the terminal of the archiver and allows the execution of commands on it.

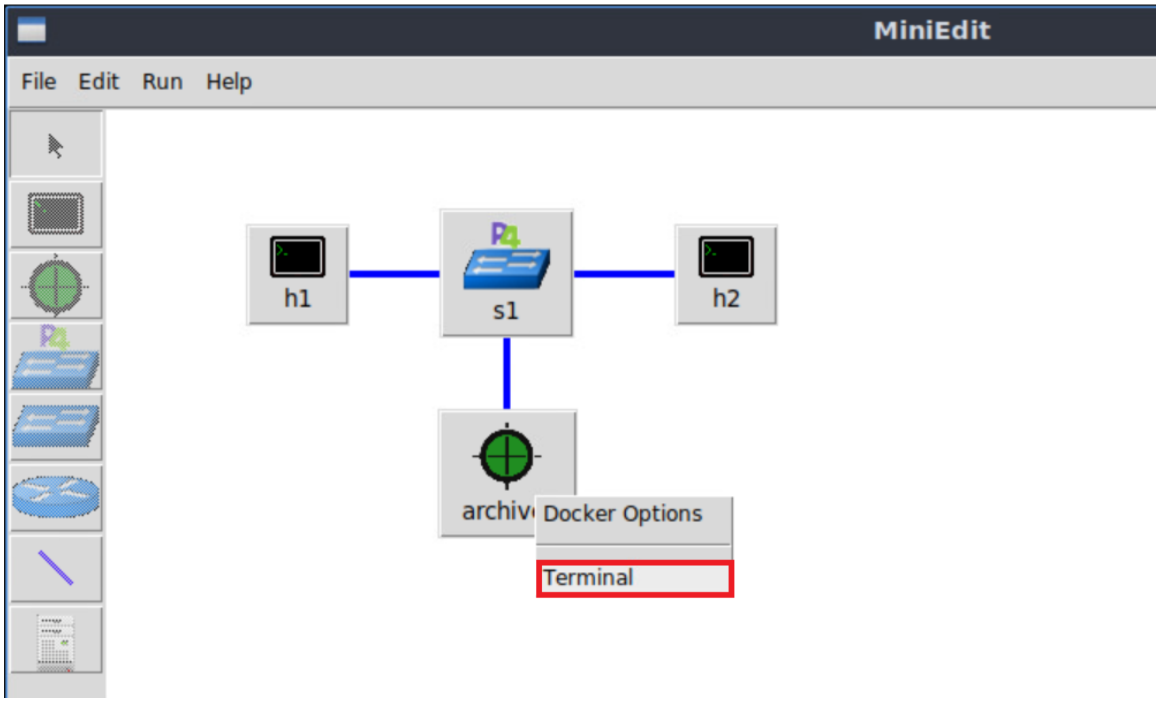
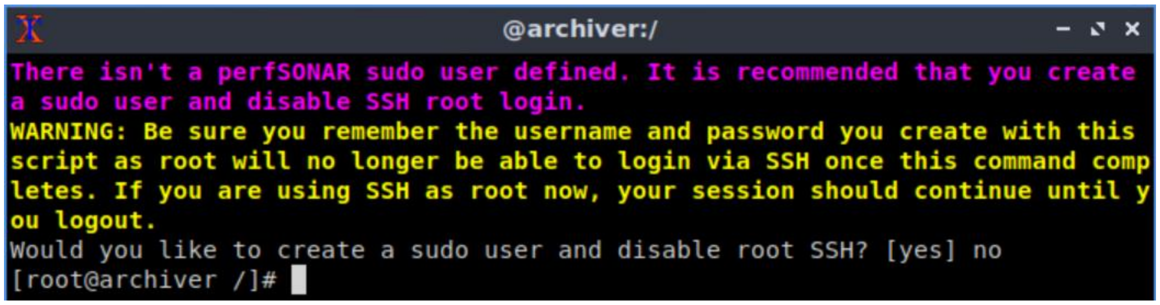


Figure 8. Opening a terminal on the archiver.

Step 2. On the opened terminal, type `no` and press *Enter*.

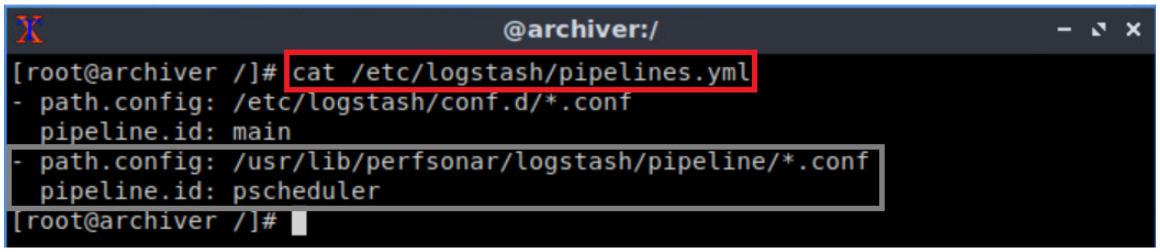


```
@archiver:/
There isn't a perfSONAR sudo user defined. It is recommended that you create
a sudo user and disable SSH root login.
WARNING: Be sure you remember the username and password you create with this
script as root will no longer be able to login via SSH once this command comp
letes. If you are using SSH as root now, your session should continue until y
ou logout.
Would you like to create a sudo user and disable root SSH? [yes] no
[root@archiver /]#
```

Figure 9. Skipping initial configuration in the archiver.

Step 3. Type the command below in the terminal. `cat` command prints the contents of the file to the standard output. The file `/etc/logstash/pipelines.yml` maintains the paths for the configuration files of Logstash's pipelines.

```
cat /etc/logstash/pipelines.yml
```



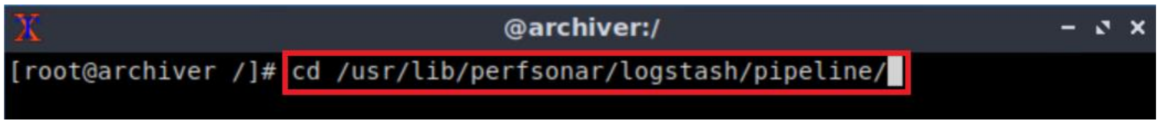
```
@archiver:/
[root@archiver /]# cat /etc/logstash/pipelines.yml
- path.config: /etc/logstash/conf.d/*.conf
  pipeline.id: main
- path.config: /usr/lib/perfsonar/logstash/pipeline/*.conf
  pipeline.id: pscheduler
[root@archiver /]#
```

Figure 10. Retrieving the path to the pipeline used by pScheduler.

The figure above shows that the configuration files of the pipeline with ID *pscheduler* are located under the directory `/usr/lib/perfsonar/logstash/pipeline/` and have `.conf` extension.

Step 4. Type the command below to navigate to the configuration files of the *pScheduler* pipeline.

```
cd /usr/lib/perfsonar/logstash/pipeline/
```

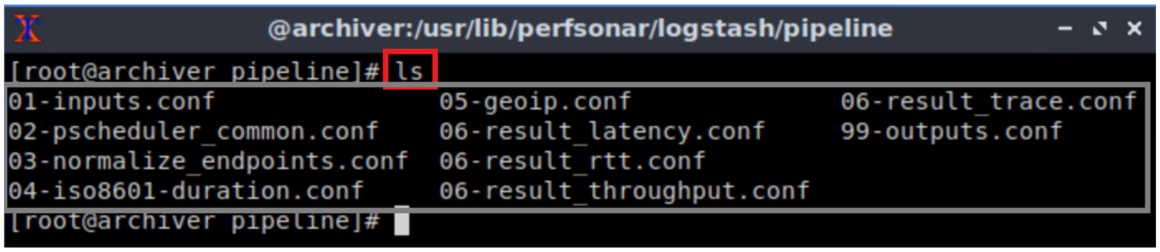


```
@archiver:/
[root@archiver /]# cd /usr/lib/perfsonar/logstash/pipeline/
```

Figure 11. Navigating to the configuration files.

Step 5. Type the command below to list the configuration files.

```
ls
```



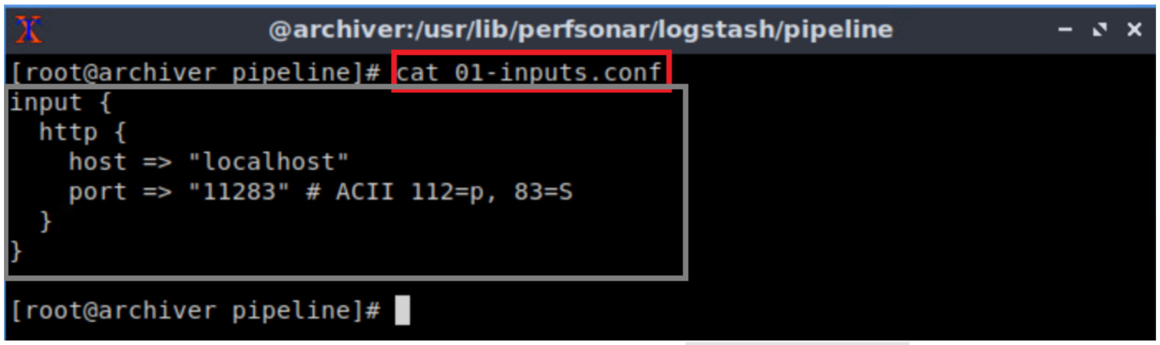
```
@archiver:/usr/lib/perfsonar/logstash/pipeline
[root@archiver pipeline]# ls
01-inputs.conf                 05-geoip.conf               06-result_trace.conf
02-pscheduler_common.conf      06-result_latency.conf      99-outputs.conf
03-normalize_endpoints.conf    06-result_rtt.conf
04-iso8601-duration.conf       06-result_throughput.conf
[root@archiver pipeline]#
```

Figure 12. listing the configuration files.

The figure above shows that *pscheduler pipeline* has a total of 10 configuration files. The file `01-inputs.conf` defines the input plug-ins used to receive data by the pipeline. The file `99-outpus.conf` defines the list of output plug-ins used to send the data out of the pipeline. The remaining files define the filters that process the data traversing the pipeline.

Step 6. Type the command below to inspect the contents of `01-inputs.conf`.

```
cat 01-inputs.conf
```



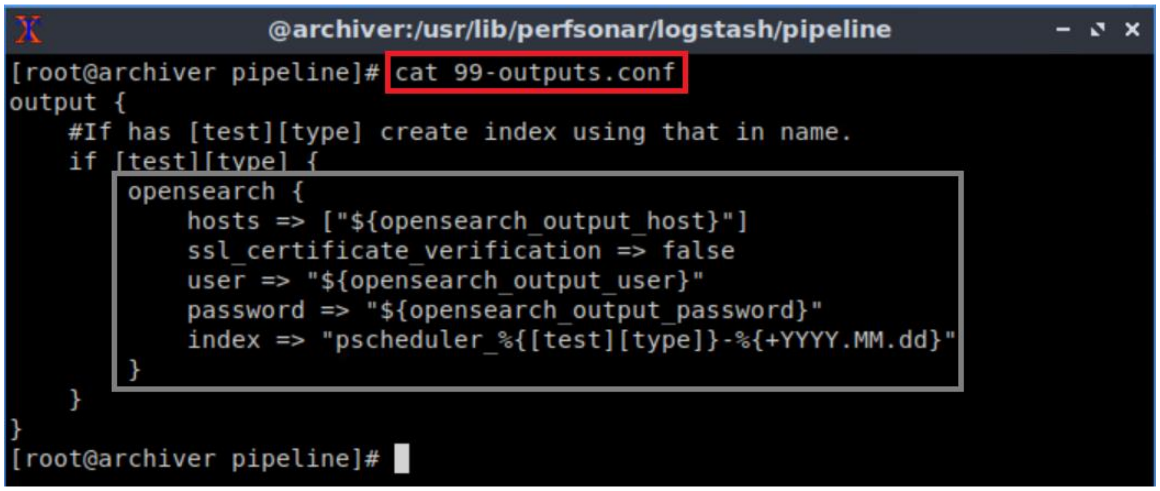
```
@archiver:/usr/lib/perfsonar/logstash/pipeline
[root@archiver pipeline]# cat 01-inputs.conf
input {
  http {
    host => "localhost"
    port => "11283" # ACII 112=p, 83=S
  }
}

[root@archiver pipeline]#
```

Figure 13. Inspecting the contents of `01-inputs.conf`.

The figure above shows that *the pscheduler pipeline uses the* `http` input plug-in and operates on `localhost`, port `11283`.

Step 7. Type the command below to inspect the contents of `99-outputs.conf`.

```
cat 99-outputs.conf
```



```
@archiver:/usr/lib/perfsonar/logstash/pipeline
[root@archiver pipeline]# cat 99-outputs.conf
output {
    #If has [test][type] create index using that in name.
    if [test][type] {
        opensearch {
            hosts => ["${opensearch_output_host}"]
            ssl_certificate_verification => false
            user => "${opensearch_output_user}"
            password => "${opensearch_output_password}"
            index => "pscheduler_%{[test][type]}-%{+YYYY.MM.dd}"
        }
    }
}
[root@archiver pipeline]#
```

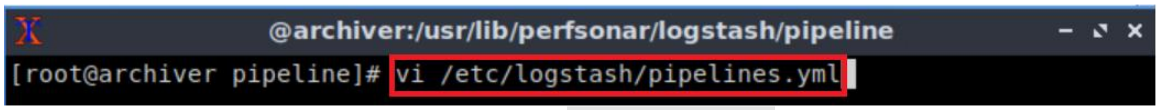
Figure 14. Inspecting the contents of `99-outputs.conf`.

The figure above shows that *the pscheduler pipeline uses the* `opensearch` output plug-in. The `index` field specifies the index at which the data is to be stored in OpenSearch. It index of the data is required to perform queries and is used while setting up the connection between Grafana and OpenSearch.

3.2 Creating P4 pipeline

Step 1. Type the command below in the terminal to edit the `pipelines.yml` file. `vi` is a text editor.

```
vi /etc/logstash/pipelines.yml
```



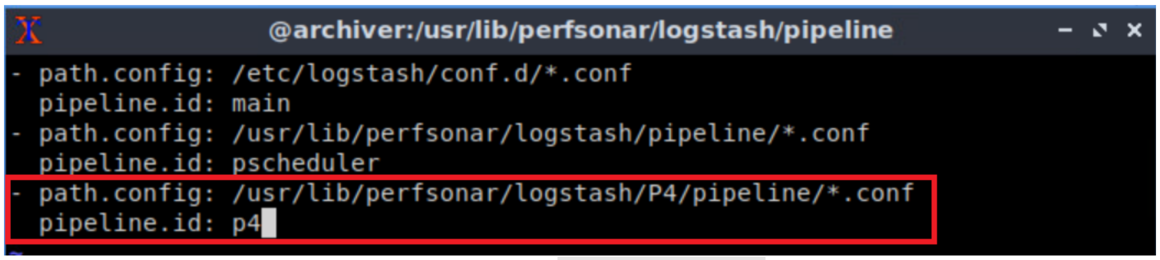
```
@archiver:/usr/lib/perfsonar/logstash/pipeline
[root@archiver pipeline]# vi /etc/logstash/pipelines.yml
```

Figure 15. Opening `pipelines.yml` file.

Step 2. After opening `pipelines.yml`, press `i` to enable editing.

Step 3. Add the code below to define the path to a new pipeline.

```
- path.config: /usr/lib/perfsonar/logstash/P4/pipeline/*.conf
  pipeline.id: p4
```

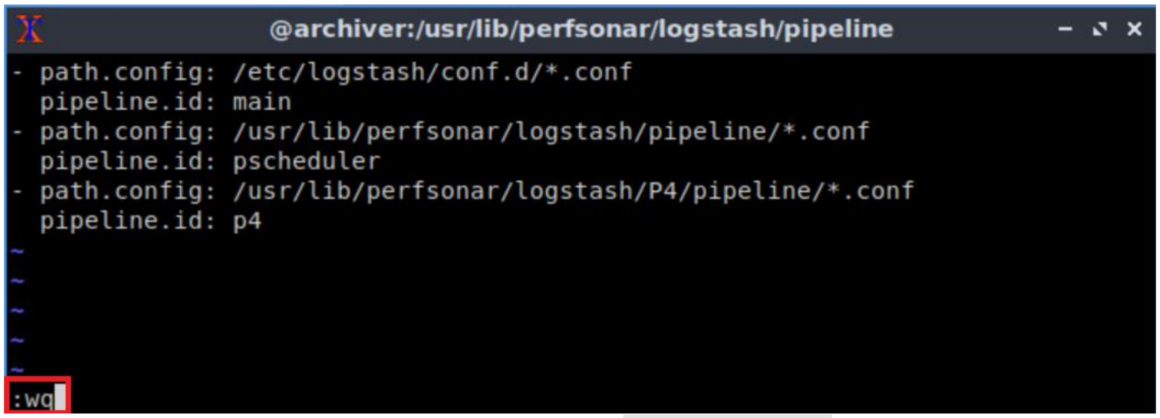


```
@archiver:/usr/lib/perfsonar/logstash/pipeline
- path.config: /etc/logstash/conf.d/*.conf
  pipeline.id: main
- path.config: /usr/lib/perfsonar/logstash/pipeline/*.conf
  pipeline.id: pscheduler
- path.config: /usr/lib/perfsonar/logstash/P4/pipeline/*.conf
  pipeline.id: p4
```

Figure 16. Editing `pipelines.yml` file.

In this step, a new path is added that points to the configuration files of the P4 pipeline. The ID of this pipeline is set to `p4`. Note that the directory `/usr/lib/perfsonar/logstash/P4/pipeline` does not exist yet. We will be creating it and adding the configuration files in the next steps.

Step 4. Press *Esc*, type `:wq`, then press *Enter* to save and exit the file.

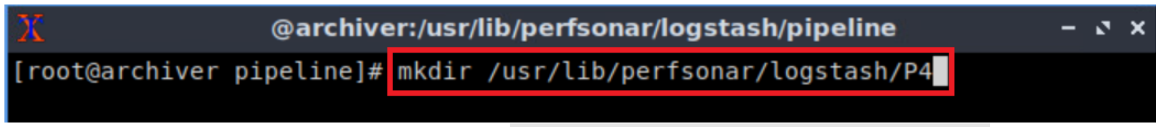


```
@archiver:/usr/lib/perfsonar/logstash/pipeline
- path.config: /etc/logstash/conf.d/*.conf
  pipeline.id: main
- path.config: /usr/lib/perfsonar/logstash/pipeline/*.conf
  pipeline.id: pscheduler
- path.config: /usr/lib/perfsonar/logstash/P4/pipeline/*.conf
  pipeline.id: p4
~
~
~
~
~
:wq
```

Figure 17. Saving and exiting `pipelines.yml` file.

Step 5. Type the command below to create the directory `/usr/lib/perfsonar/logstash/P4`.

```
mkdir /usr/lib/perfsonar/logstash/P4
```

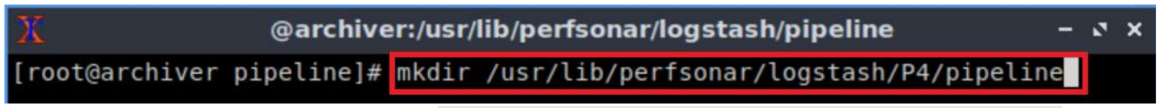


```
@archiver:/usr/lib/perfsonar/logstash/pipeline
[root@archiver pipeline]# mkdir /usr/lib/perfsonar/logstash/P4
```

Figure 18. Creating directory `/usr/lib/perfsonar/logstash/P4`.

Step 6. Type the command below to create the directory `/usr/lib/perfsonar/logstash/P4/pipeline`.

```
mkdir /usr/lib/perfsonar/logstash/P4/pipeline
```

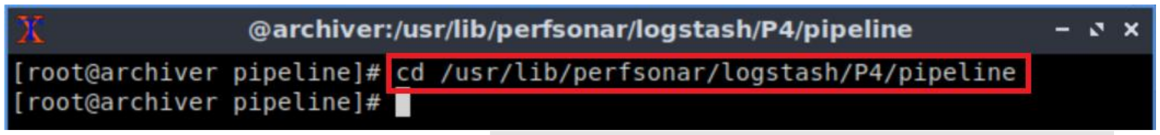


```
@archiver:/usr/lib/perfsonar/logstash/pipeline
[root@archiver pipeline]# mkdir /usr/lib/perfsonar/logstash/P4/pipeline
```

Figure 19. Creating directory `/usr/lib/perfsonar/logstash/P4/pipeline`.

Step 7. Type the command below to navigate to the directory `/usr/lib/perfsonar/logstash/P4/pipeline`.

```
cd /usr/lib/perfsonar/logstash/P4/pipeline
```



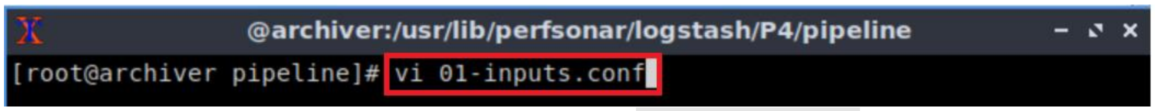
```
@archiver:/usr/lib/perfsonar/logstash/P4/pipeline
[root@archiver pipeline]# cd /usr/lib/perfsonar/logstash/P4/pipeline
[root@archiver pipeline]#
```

Figure 20. Navigating to the directory `/usr/lib/perfsonar/logstash/P4/pipeline`.

3.3 Configuring P4 pipeline

Step 1. Type the command below to create and open `01-inputs.conf` file.

```
vi 01-inputs.conf
```



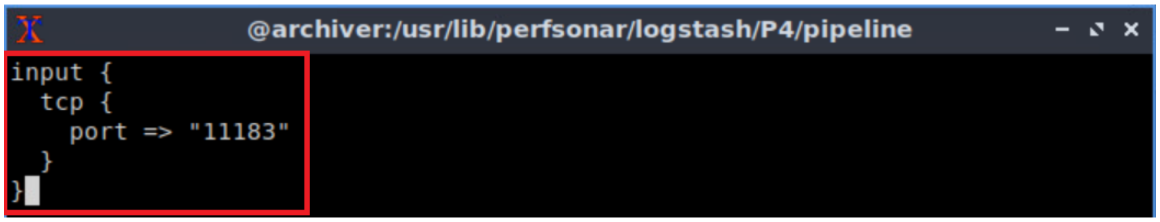
```
@archiver:/usr/lib/perfsonar/logstash/P4/pipeline
[root@archiver pipeline]# vi 01-inputs.conf
```

Figure 21. Creating and opening `01-inputs.conf` file.

Step 2. After opening `01-inputs.conf`, press `i` to enable editing.

Step 3. Type the code below in `01-inputs.conf` file.

```
input {
  tcp {
    port => "11183"
  }
}
```

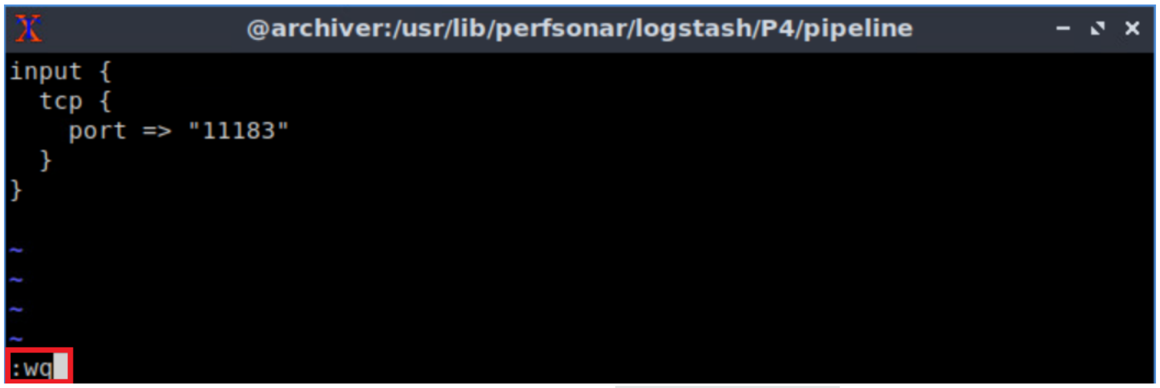


```
@archiver:/usr/lib/perfsonar/logstash/P4/pipeline
input {
  tcp {
    port => "11183"
  }
}
```

Figure 22. Configuring TCP input plugin.

In the code above, we configure the P4 pipeline to use the TCP input plugin and listen on port `11183`. Any requests received on port 11183 will be processed by the P4 pipeline. Note that any port not reserved by another process can be used.

Step 4. Press *Esc*, type `:wq`, then press *Enter* to save and exit the file.

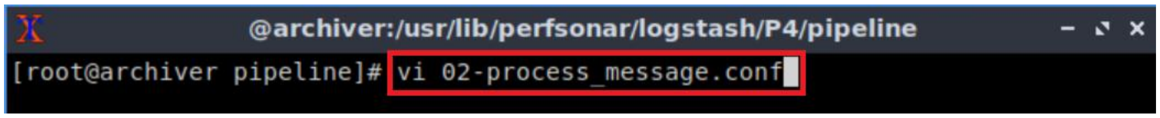


```
@archiver:/usr/lib/perfsonar/logstash/P4/pipeline
input {
  tcp {
    port => "11183"
  }
}

~
~
~
~
:wq
```

Figure 23. Saving and exiting `01-inputs.conf` file.

Step 5. Type the command below to create and open `02-process_message.conf` file.

```
vi 02-process_message.conf
```



```
@archiver:/usr/lib/perfsonar/logstash/P4/pipeline
[root@archiver pipeline]# vi 02-process_message.conf
```

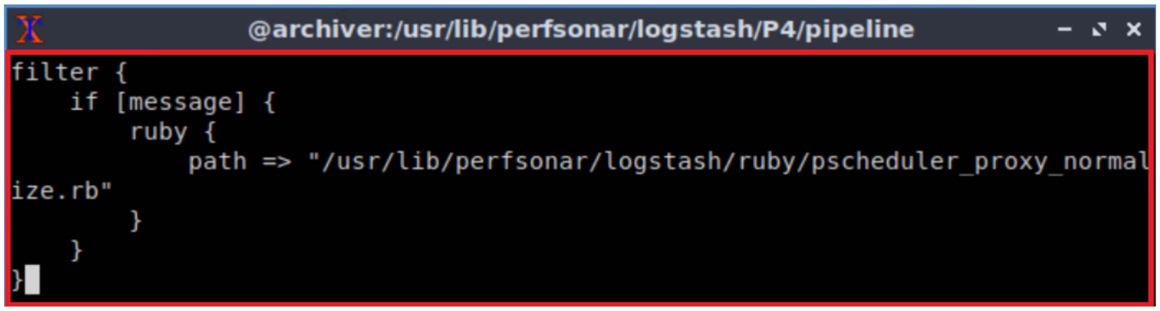
Figure 24. Creating and opening `02-process_message.conf` file.

This file will represent the only filter Logstash needs for processing the measurements from the control plane. Logstash wraps all received measurements under a *message* object and adds its own metadata (e.g., the time the measurement was received) as the headers of this message. This filter is responsible for unpacking the measurement from the message object. Note that we will be using the same script used by perfSONAR to unpack its measurements.

Step 6. After opening `02- process_message.conf`, press `i` to enable editing.

Step 7. Type the code below in `02- process_message.conf` file.

```
filter {
     if [message] {
          ruby {
             path => "/usr/lib/perfsonar/logstash/ruby/pscheduler_proxy_nomal
ize.rb"
          }
     }
}
```



```
filter {
     if [message] {
          ruby {
               path => "/usr/lib/perfsonar/logstash/ruby/pscheduler_proxy_normal
ize.rb"
          }
     }
}
```

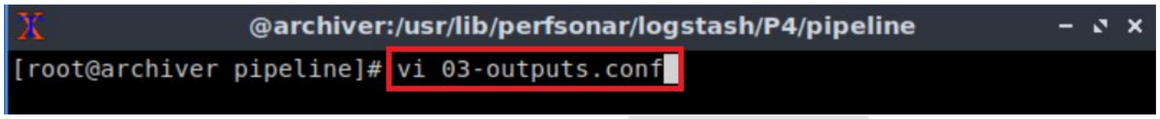
Figure 25. Configuring message processing filter.

The filter checks if the arriving measurement has the message object (i.e., has the message key). For the measurements that satisfy the condition, the filter applies the logic described in the Ruby application located at `/usr/lib/perfsonar/logstash/ruby/pscheduler_proxy_normalize.rb`.

Step 8. Press *Esc*, type `:wq`, then press *Enter* to save and exit the file.

Step 9. Type the command below to create and open `03-outputs.conf` file.

```
vi 03-outputs.conf
```



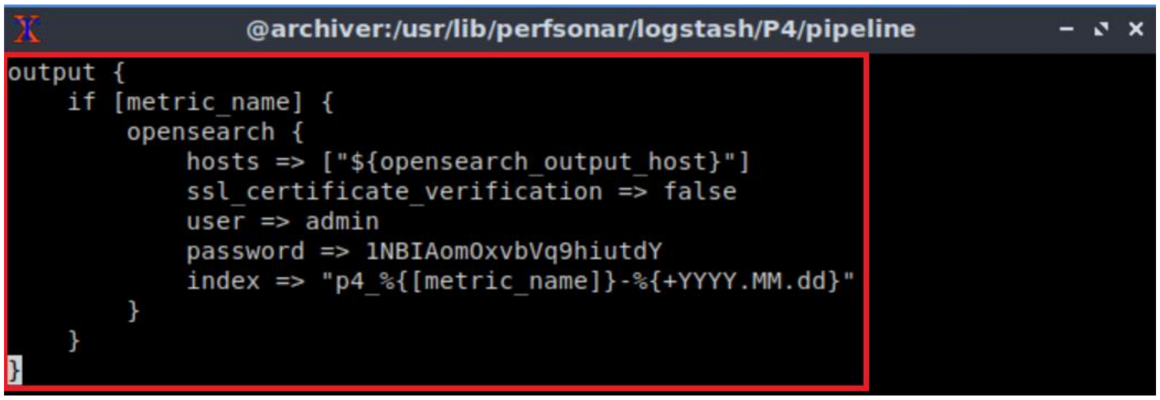
```
[root@archiver pipeline]# vi 03-outputs.conf
```

Figure 26. Creating and opening `03-outputs.conf` file.

Step 10. After opening `03-outputs.conf`, press `i` to enable editing.

Step 11. Type the code below in `03-outputs.conf` file.

```
output {
  if [metric_name] {
      opensearch {
          hosts => ["${opensearch_output_host}"]
          ssl_certificate_verification => false
          user => admin
          password => 1NBIAomOxvbVq9hiutdY
          index => "p4_%{[metric_name]}-%{+YYYY.MM.dd}"
  }
}
```



```
output {
    if [metric_name] {
        opensearch {
            hosts => ["${opensearch_output_host}"]
            ssl_certificate_verification => false
            user => admin
            password => 1NBIAomOxvbVq9hiutdY
            index => "p4_%{[metric_name]}-%{+YYYY.MM.dd}"
        }
    }
}
```

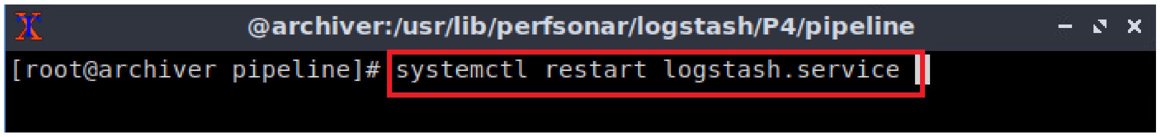
Figure 27. Configuring OpenSearch output plugin.

In the code above, we are using the `opensearch` output plugin. `host` field specifies the location of the OpenSearch service. Note that the variable `${opensearch_output_host}` is a Logstash environment variable defined in the file `/etc/perfsonar/logstash/logstash_sysconfig`. `usr` and `password` fields specify the credentials that will be used by Logstash to access OpenSearch. Their values are stored in `/etc/perfsonar/opensearch/opensearch_login`. `index` field specifies the index where the measurements are stored in OpenSearch. This index will be used by Grafana to point to the measurements collected by OpenSearch from the P4 switch. Note that different metrics collected by the P4 switch will have different index names because we are using the `metric_name` field of reports coming from the P4 switch as part of the index. The structure of the P4 report will be described in the next section.

Step 12. Press *Esc*, type `:wq`, then press *Enter* to save and exit the file.

Step 13. Type the command below to restart Logstash and load the P4 pipeline.

```
systemctl restart logstash.service
```



```
[root@archiver pipeline]# systemctl restart logstash.service
```

Figure 28. Restarting Logstash.

4 Configuring the control plane of the switch

In this section, you will configure the switch's control plane to retrieve the measurements from the data plane, process the measurements to create a report that Logstash can ingest, connect to Logstash, and periodically report the measurements to the archiver. The P4 program is already compiled for you.

4.1 Loading the environment

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

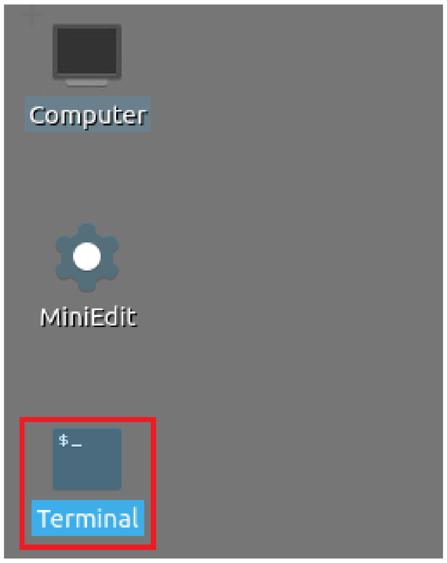
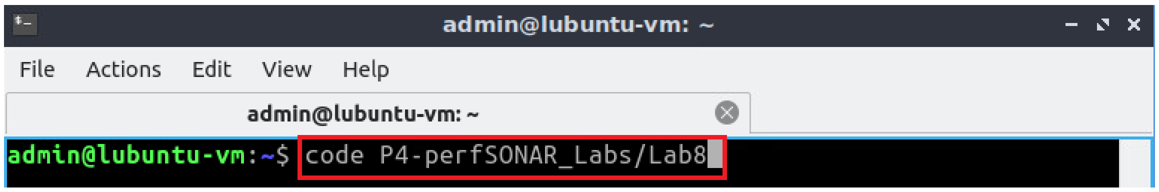


Figure 29. Shortcut to open a Linux terminal.

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.

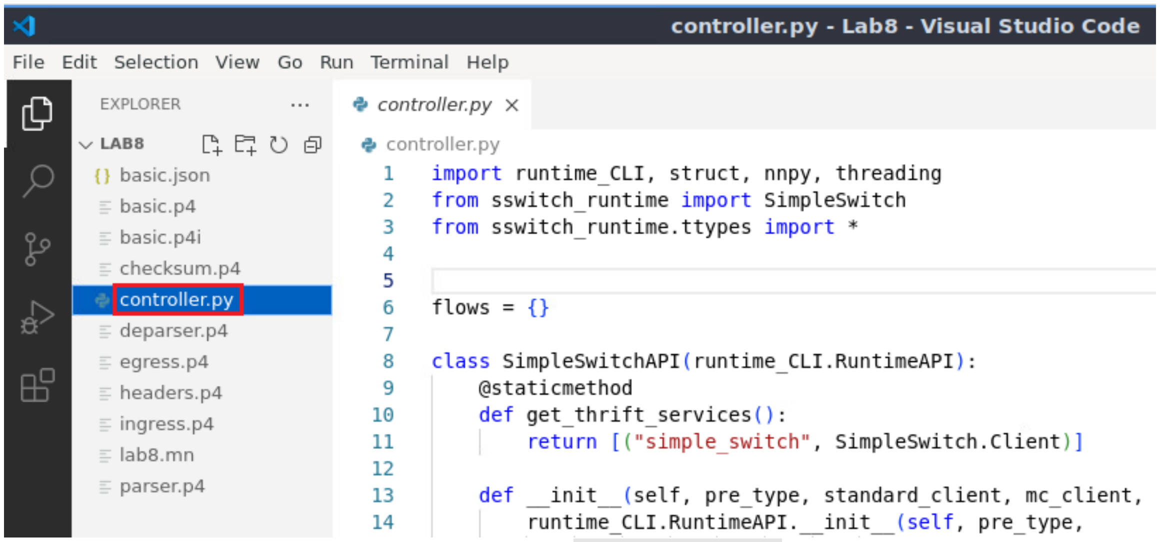
```
code P4-perfSOANR_Labs/Lab8
```



```
admin@lubuntu-vm:~$ code P4-perfSONAR_Labs/Lab8
```

Figure 30. Loading the development environment.

Step 3. 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. This file will run on the control plane of the switch.

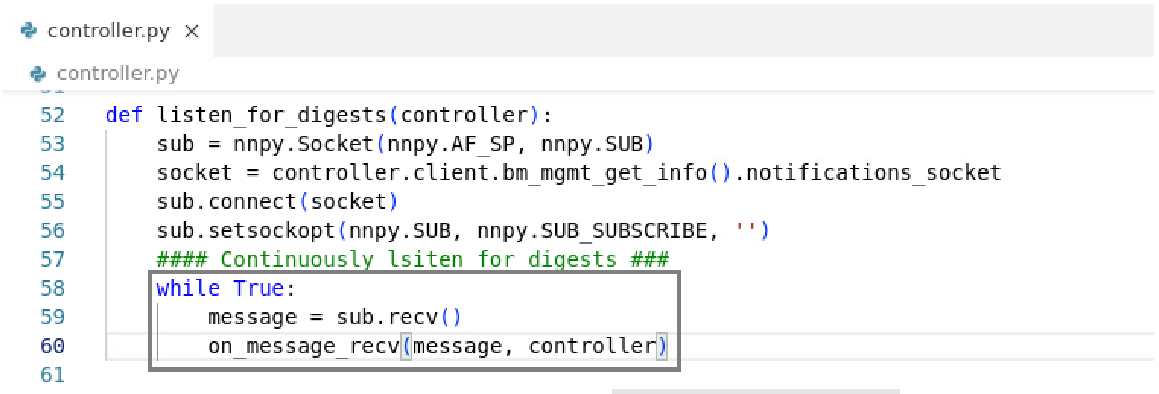


```
import runtime_CLI, struct, nnpy, threading
from sswitch_runtime import SimpleSwitch
from sswitch_runtime.ttypes import *


flows = {}

class SimpleSwitchAPI(runtime_CLI.RuntimeAPI):
    @staticmethod
    def get_thrift_services():
        return [("simple_switch", SimpleSwitch.Client)]

    def __init__(self, pre_type, standard_client, mc_client,
        runtime_CLI.RuntimeAPI.__init__(self, pre_type,
```

Figure 31. Accessing `controller.py` file.

4.2 Inspecting the controller

Step 1. Scroll down and inspect the function `listen_for_digests`.

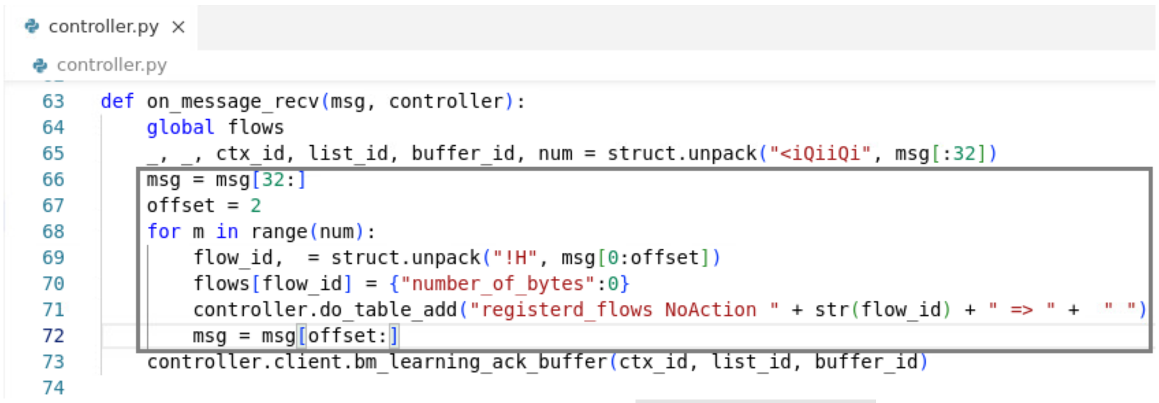


```
def listen_for_digests(controller):
    sub = nnpy.Socket(nnpy.AF_SP, nnpy.SUB)
    socket = controller.client.bm_mgmt_get_info().notifications_socket
    sub.connect(socket)
    sub.setsockopt(nnpy.SUB, nnpy.SUB_SUBSCRIBE, '')
    #### Continuously lsiten for digests ###
    while True:
        message = sub.recv()
        on_message_recv(message, controller)
```

Figure 32. Inspecting function `listen_for_digests`.

The function `listen_for_digests` continuously listens for digests from the data plane. For each digest, the function calls `on_message_recv`. `on_message_recv` is responsible for extracting the information from the digests.

Step 2. Scroll down and inspect the function `on_message_recv`.



```
63  def on_message_recv(msg, controller):
64      global flows
65      _, _, ctx_id, list_id, buffer_id, num = struct.unpack("<iQiiQi", msg[:32])
66      msg = msg[32:]
67      offset = 2
68      for m in range(num):
69          flow_id,  = struct.unpack("!H", msg[0:offset])
70          flows[flow_id] = {"number_of_bytes":0}
71          controller.do_table_add("registerd_flows NoAction " + str(flow_id) + " => " +  " ")
72          msg = msg[offset:]
73      controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)
74
```

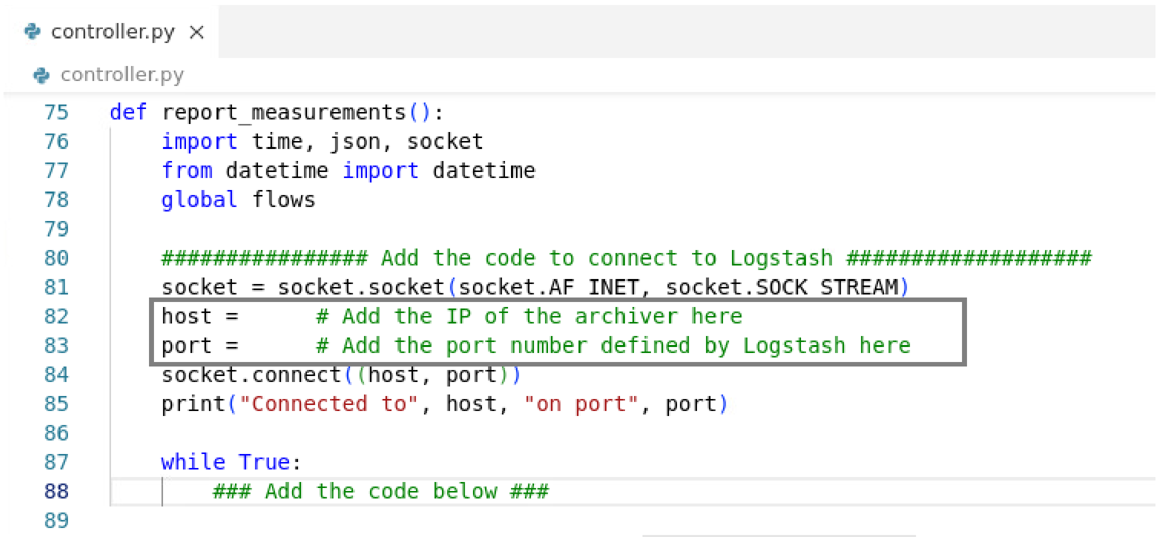
Figure 33. Inspecting function `on_message_recv`.

The code in the figure above is explained as follows:

- Line 66: Skip the first 32 bytes because they store some metadata related to the digest and the switch.
- Line 67: The offset value indicates the number of bytes corresponding to the flow ID (i.e., 16 bits).
- Line 69: Unpacks from the digest the flow ID.
- Line 70: Define a new entry for the received flow in the `flows` dictionary, a global dictionary responsible for maintaining information about the flows detected by the data plane. The flow ID (i.e., `flow_id`) is used as the key.
- Line 71: Adds an entry to the table `monitored_flows` that matches the flow ID and executes the action `NoAction`. By adding this entry, we inform the data plane that consequent packets with this flow ID do not belong to a new flow.
- Line 72: 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).

4.3 Configuring the connection with Logstash

Step 1. Scroll down and inspect the function `report_measurements`. This function is responsible for periodically extracting the measurements from the data plane and reporting them to the archiver.



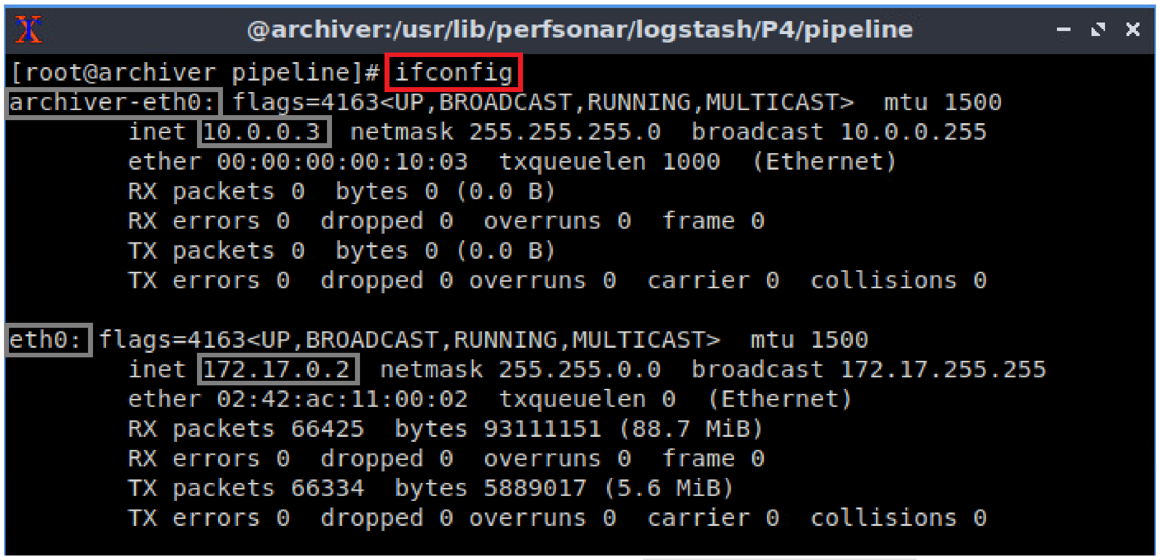
```
controller.py ×
controller.py
def report_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    ################ Add the code to connect to Logstash ###################
    socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host =      # Add the IP of the archiver here
    port =      # Add the port number defined by Logstash here
    socket.connect((host, port))
    print("Connected to", host, "on port", port)

    while True:
        ### Add the code below ###
```

Figure 34. Inspecting the function `report_measurements`.

The variable `host` and `port` in the code above are the IP address of the archiver and the port number defined previously by Logstash in the P4 pipeline (i.e., 11183).

Step 2. Navigate to the archiver and issue the command below to get the IP addresses of the interfaces.

```
ifconfig
```

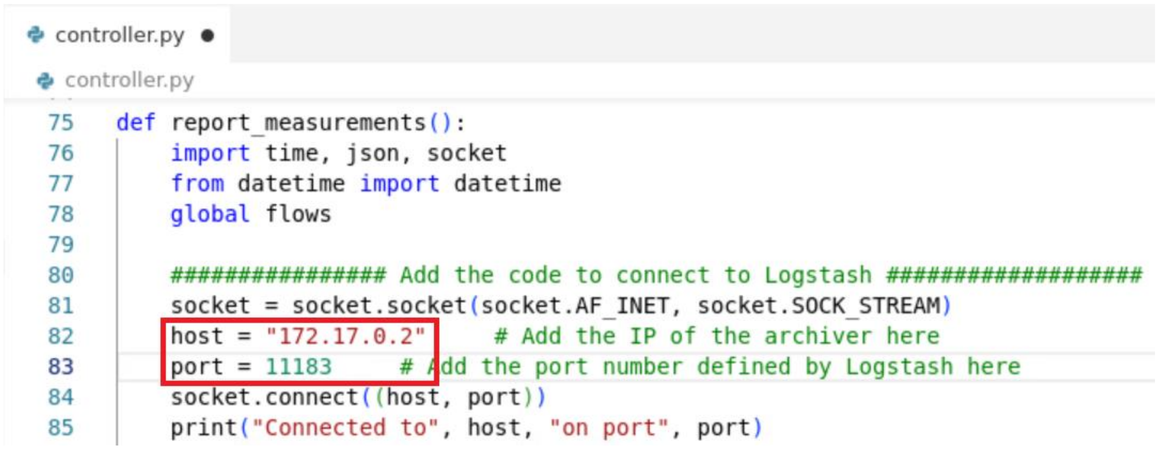


```
@archiver:/usr/lib/perfsonar/logstash/P4/pipeline
[root@archiver pipeline]# ifconfig
archiver-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.3  netmask 255.255.255.0  broadcast 10.0.0.255
        ether 00:00:00:00:10:03  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 66425  bytes 93111151 (88.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 66334  bytes 5889017 (5.6 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 35. Inspecting the function `report_measurements`.

The figure above shows two interfaces, `archiver-eth0`, and `eth0` (there is also the loopback interface, but it is not shown in the figure). Interface `archiver-eth0` connects the archiver with the data plane of the switch. Interface `eth0` connects the archiver to the control plane of the switch, and its IP address (i.e., `172.17.0.2`) will be used by the control plane of the switch to connect to Logstash.

Step 3. Navigate back to the controller and assign the value `"172.168.0.2"` to the `host` variable and `11183` to the `port` variable.



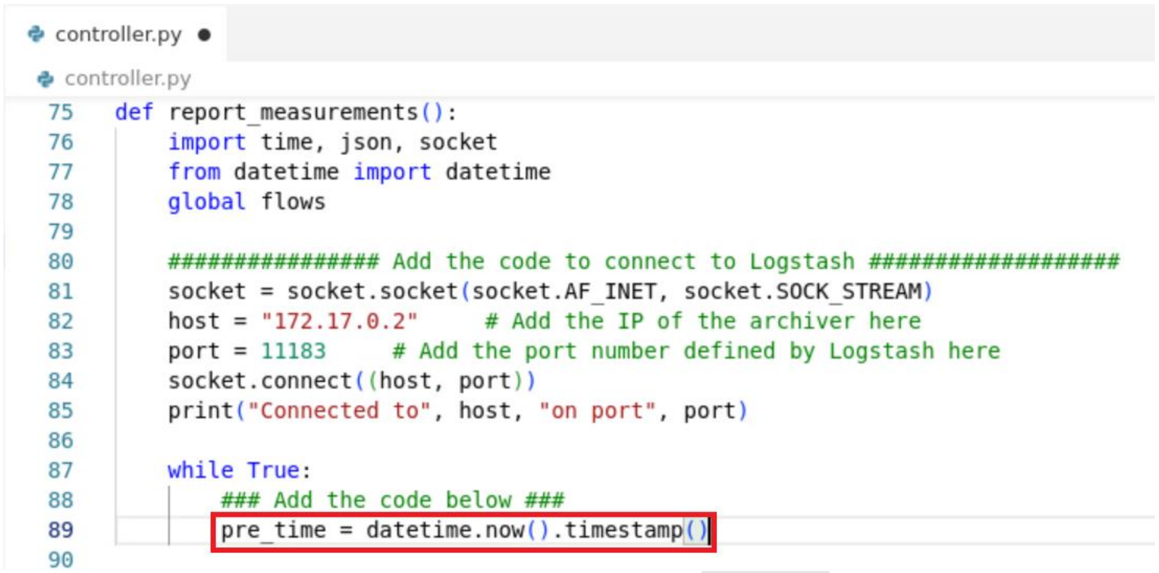
```
controller.py ●
controller.py
def report_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    ################ Add the code to connect to Logstash ###################
    socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = "172.17.0.2"      # Add the IP of the archiver here
    port = 11183     # Add the port number defined by Logstash here
    socket.connect((host, port))
    print("Connected to", host, "on port", port)
```

Figure 36. Appending the IP address and the port number.

4.4 Configuring the data retrieving and reporting process

Step 1. Type the code below inside the while loop. `pre_time` will store the current timestamp. It will be used to set the period of data retrieving from the data plane and reporting to the archiver.

```
pre_time = datetime.now().timestamp()
```

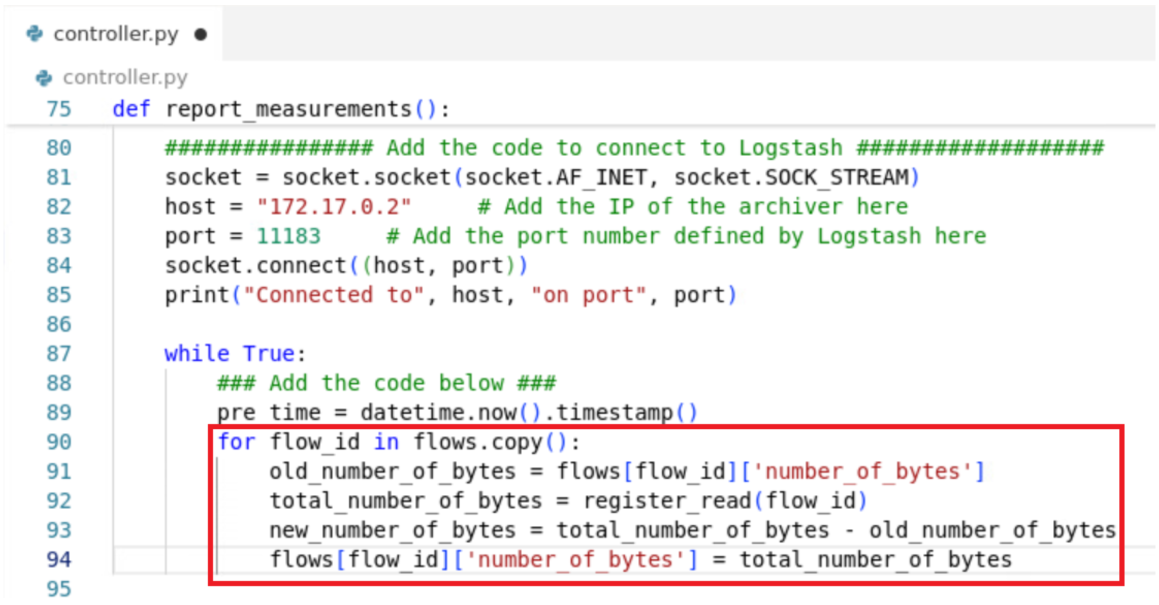


```
controller.py ●
controller.py
def report_measurements():
    import time, json, socket
    from datetime import datetime
    global flows

    ################ Add the code to connect to Logstash ###################
    socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = "172.17.0.2"      # Add the IP of the archiver here
    port = 11183     # Add the port number defined by Logstash here
    socket.connect((host, port))
    print("Connected to", host, "on port", port)

    while True:
        ### Add the code below ###
        pre_time = datetime.now().timestamp()

```

Figure 37. Defining the variable `pre_time`.

Step 2. Add the code below to iterate over the monitored flows and retrieve the corresponding number of bytes.

```
for flow_id in flows.copy():
    old_number_of_bytes = flows[flow_id]['number_of_bytes']
    total_number_of_bytes = register_read(flow_id)
    new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
    flows[flow_id]['number_of_bytes'] = total_number_of_bytes
```



controller.py ●

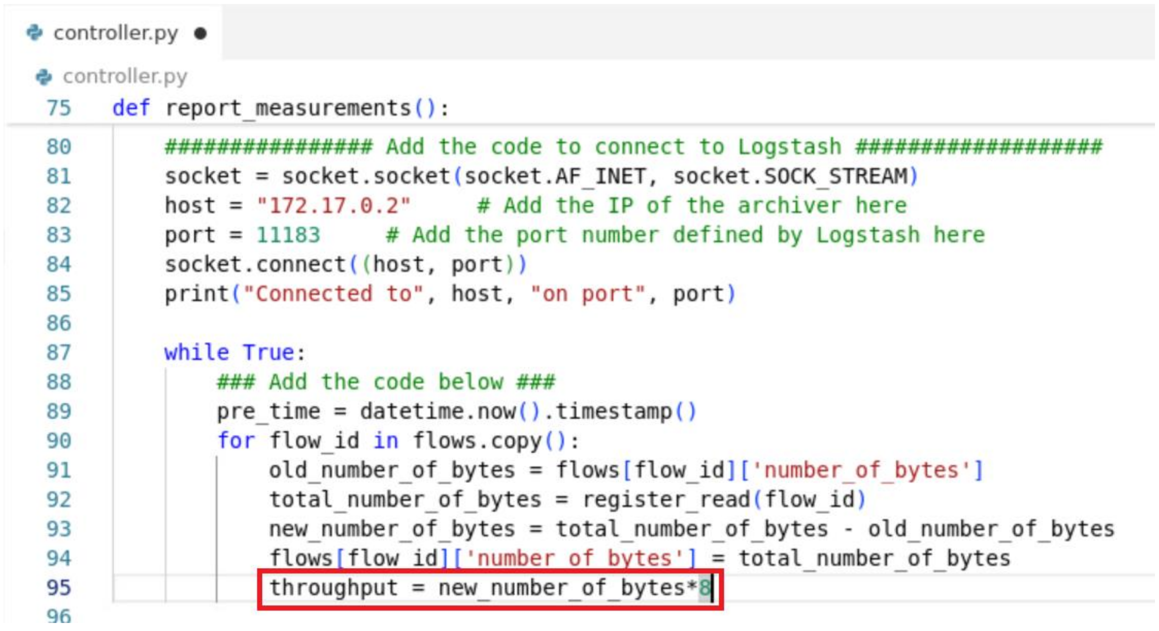
controller.py

```
def report_measurements():
    ################ Add the code to connect to Logstash ###################
    socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = "172.17.0.2"     # Add the IP of the archiver here
    port = 11183     # Add the port number defined by Logstash here
    socket.connect((host, port))
    print("Connected to", host, "on port", port)

    while True:
        ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
```

Figure 38. Retrieving per-flow number of bytes.

Step 3. Add the code below to calculate the throughput.

```
throughput = new_number_of_bytes*8
```



controller.py ●

controller.py

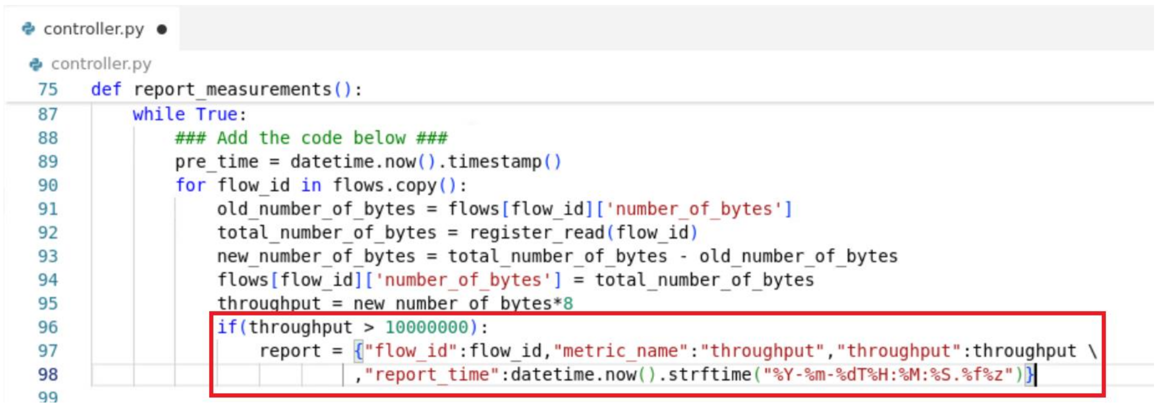
```
def report_measurements():
    ################ Add the code to connect to Logstash ###################
    socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = "172.17.0.2"     # Add the IP of the archiver here
    port = 11183     # Add the port number defined by Logstash here
    socket.connect((host, port))
    print("Connected to", host, "on port", port)

    while True:
        ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
            throughput = new_number_of_bytes*8
```

Figure 39. Calculating the throughput.

In the code above, the throughput is measured in bits per second as the number of bytes will be retrieved once per second from the data plane and then divided by 8.

Step 4. Add the code below to process the throughput measurement and create the report.

```
if(throughput  > 10000000):
     report={"flow_id":flow_id,"metric_name":"throughput","throughput":
throughput\ ,"report_time":datetime.now().strftime("%Y-%m-%dT%H:%M:%S.%f%z)}
```



```
controller.py ●
controller.py
def report_measurements():
    while True:
        ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
            throughput = new_number_of_bytes*8
            if(throughput > 10000000):
                report = {"flow_id":flow_id,"metric_name":"throughput","throughput":throughput \
                          ,"report_time":datetime.now().strftime("%Y-%m-%dT%H:%M:%S.%f%z")}
```

Figure 40. Creating the report from the throughput measurement.

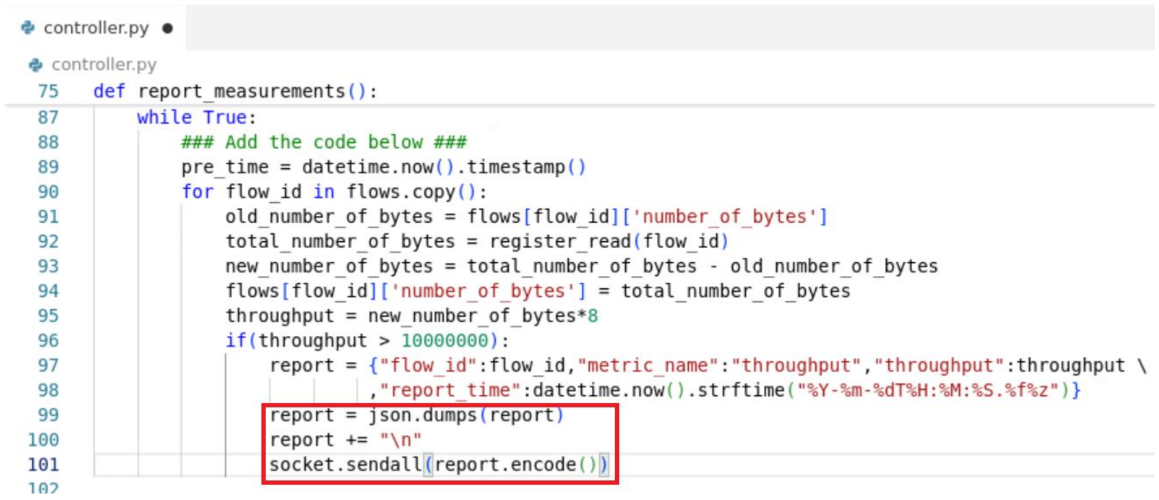
The if condition checks if the throughput is larger than 10 Mbps, and consequently, the control plane will only report the measurements of the flows with throughput larger than 10 Mbps.

The fields of the created report are utilized as follows:

- `“flow_id”:flow_id`: This key-value pair is used to specify the ID of the flow that this report belongs to. It will be used by Grafana to group the measurements.
- `“metric_name”:”throughput”`: This key-value pair is used by the OpenSearch output plugin to (1) validate that an arriving measurement is from the control plane of P4 and (2) define the index inside the OpenSearch database.
- `“throughput”:throughput:` This key-value pair stores the value of the throughput measurement.
- `”report_time”:datetime.now().strftime(“%Y-%m-%dT%H:%M:%S.%f%z)}:` This key-value pair is used by OpenSearch to sort the measurements by the time they were retrieved from the data plane. Note that we are casting the `report_time` to string format because we have to convert the report from a Python object to JSON formatted string before sending it to Logstash.

Step 5. Add the code below to convert the report into JSON format and send it to the archiver.

```
report = json.dumps(report)
report += “\n”
socket.sendall(report.encode())
```

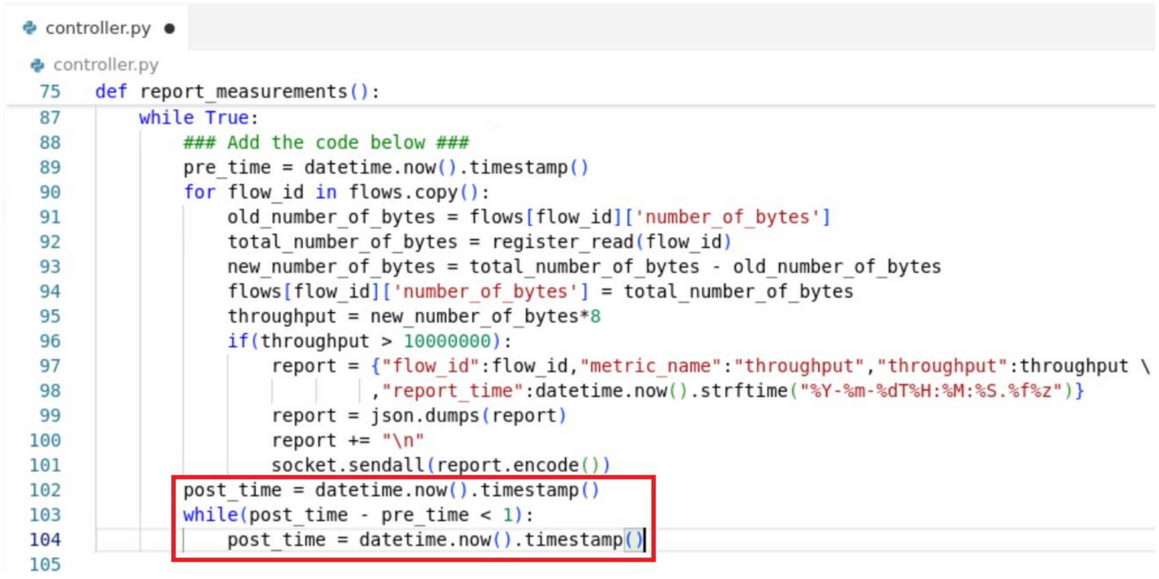


```
controller.py ●
controller.py
def report_measurements():
    while True:
        ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
            throughput = new_number_of_bytes*8
            if(throughput > 10000000):
                report = {"flow_id":flow_id,"metric_name":"throughput","throughput":throughput \
                        ,"report_time":datetime.now().strftime("%Y-%m-%dT%H:%M:%S.%f%z")}
                report = json.dumps(report)
                report += "\n"
                socket.sendall(report.encode())
```

Figure 41. Sending the report to the archiver.

Step 6. Add the code below to query the data plane once per second.

```
post_time = datetime.now().timestamp()
while(post_time – pre_time < 1):
    post_time = datetime.now().timestamp()
```

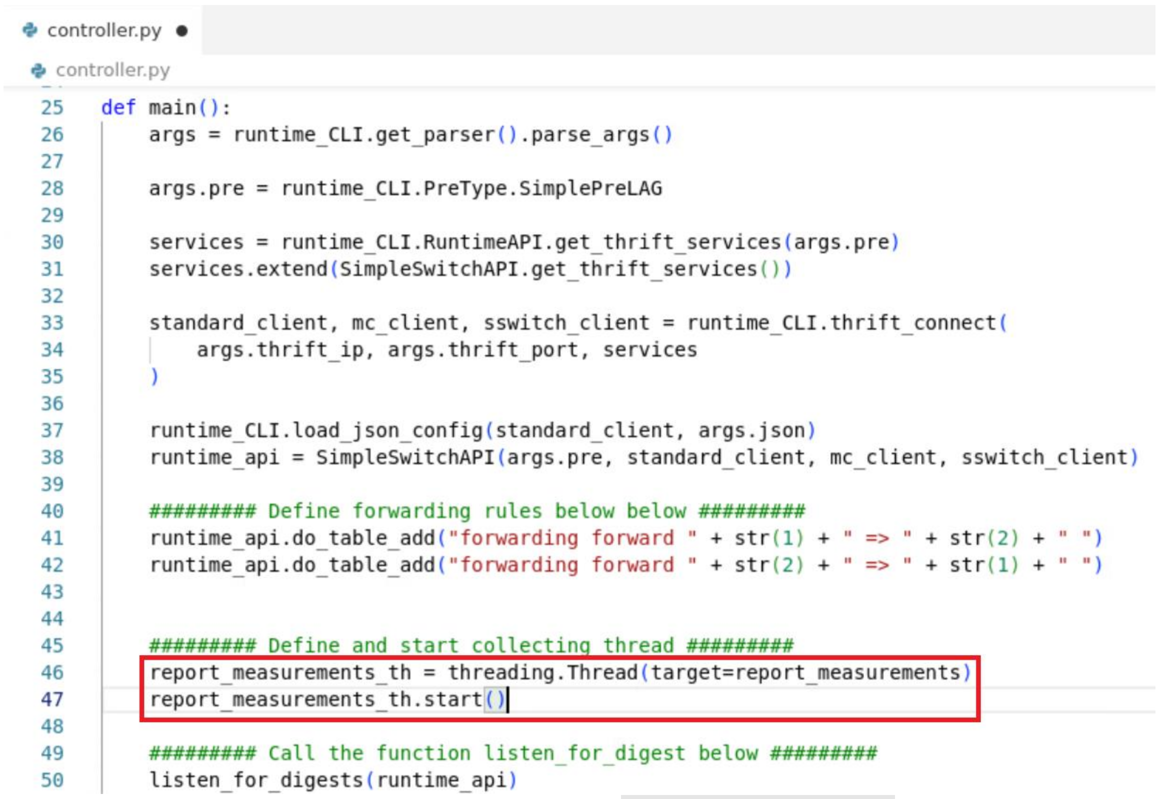


```
controller.py ●
controller.py
def report_measurements():
    while True:
        ### Add the code below ###
        pre_time = datetime.now().timestamp()
        for flow_id in flows.copy():
            old_number_of_bytes = flows[flow_id]['number_of_bytes']
            total_number_of_bytes = register_read(flow_id)
            new_number_of_bytes = total_number_of_bytes - old_number_of_bytes
            flows[flow_id]['number_of_bytes'] = total_number_of_bytes
            throughput = new_number_of_bytes*8
            if(throughput > 10000000):
                report = {"flow_id":flow_id,"metric_name":"throughput","throughput":throughput \
                        ,"report_time":datetime.now().strftime("%Y-%m-%dT%H:%M:%S.%f%z")}
                report = json.dumps(report)
                report += "\n"
                socket.sendall(report.encode())
        post_time = datetime.now().timestamp()
        while(post_time - pre_time < 1):
            post_time = datetime.now().timestamp()
```

Figure 42. Setting the querying period to 1 second.

Step 7. Scroll up to the main function then define and start a new thread to call the function `report_measurements`.

```
report_measurements_th = threading.Thread(targer=report_measurements)
report_measurements_th.start()
```


controller.py ●

controller.py

```
def main():
    args = runtime_CLI.get_parser().parse_args()

    args.pre = runtime_CLI.PreType.SimplePreLAG

    services = runtime_CLI.RuntimeAPI.get_thrift_services(args.pre)
    services.extend(SimpleSwitchAPI.get_thrift_services())

    standard_client, mc_client, sswitch_client = runtime_CLI.thrift_connect(
        args.thrift_ip, args.thrift_port, services
    )

    runtime_CLI.load_json_config(standard_client, args.json)
    runtime_api = SimpleSwitchAPI(args.pre, standard_client, mc_client, sswitch_client)

    ######### Define forwarding rules below below #########
    runtime_api.do_table_add("forwarding forward " + str(1) + " => " + str(2) + " ")
    runtime_api.do_table_add("forwarding forward " + str(2) + " => " + str(1) + " ")


    ######### Define and start collecting thread #########
    report_measurements_th = threading.Thread(target=report_measurements)
    report_measurements_th.start()

    ######### Call the function listen_for_digest below #########
    listen_for_digests(runtime_api)
```

Figure 43. Calling the function `report_measurements`.

Step 8. Press `Ctrl+s` to save the changes.

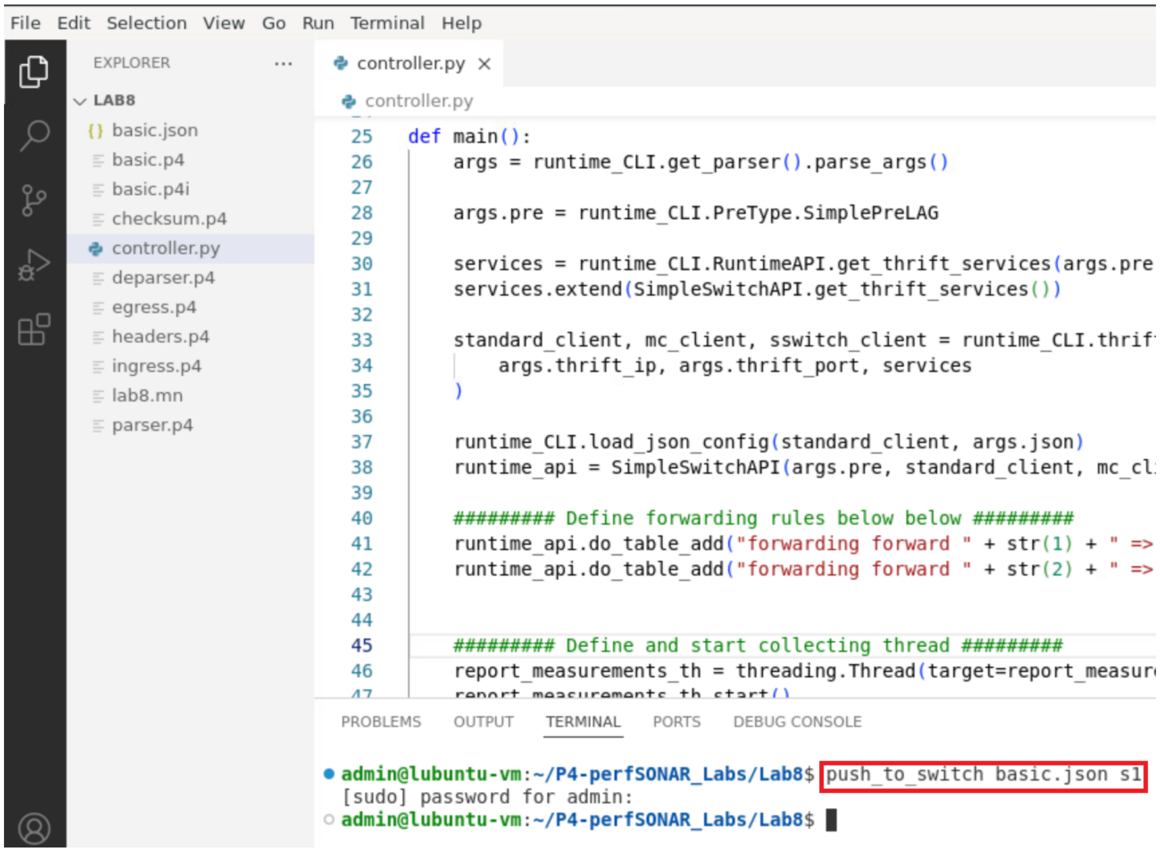
5 Loading the P4 program

In this section, you will load the P4 binary and the controller program in switch s1. You will also verify that the files reside in the switch filesystem.

5.1 Compiling and loading the P4 program to switch s1

Step 1. Type the command below in the terminal panel to push the *basic.json* file to 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
```



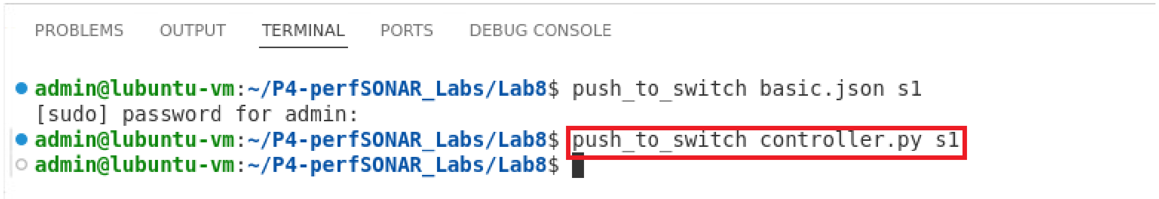
```
def main():
    args = runtime_CLI.get_parser().parse_args()

    args.pre = runtime_CLI.PreType.SimplePreLAG

    services = runtime_CLI.RuntimeAPI.get_thrift_services(args.pre
    services.extend(SimpleSwitchAPI.get_thrift_services())

    standard_client, mc_client, sswitch_client = runtime_CLI.thrif
        args.thrift_ip, args.thrift_port, services
    )

    runtime_CLI.load_json_config(standard_client, args.json)
    runtime_api = SimpleSwitchAPI(args.pre, standard_client, mc_cl

    ######### Define forwarding rules below below #########
    runtime_api.do_table_add("forwarding forward " + str(1) + " =>
    runtime_api.do_table_add("forwarding forward " + str(2) + " =>


    ######### Define and start collecting thread #########
    report_measurements_th = threading.Thread(target=report_measur
```

```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab8$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab8$
```

Figure 44. Pushing the *basic.json* file to switch s1.

Step 2. Type the command below in the terminal panel to push the *controller.py* file to switch s1's filesystem.

```
push_to_switch controller.py s1
```



```
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab8$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab8$ push_to_switch controller.py s1
admin@lubuntu-vm:~/P4-perfSONAR_Labs/Lab8$
```

Figure 45. Pushing the *controller.py* file to switch s1.

5.2 Verifying the configuration

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

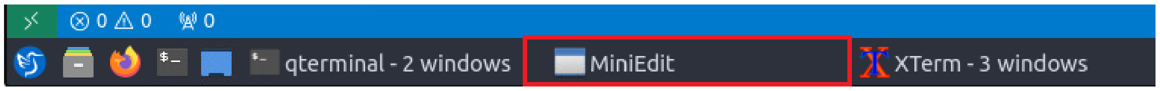


Figure 46. Maximizing the MiniEdit window.

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

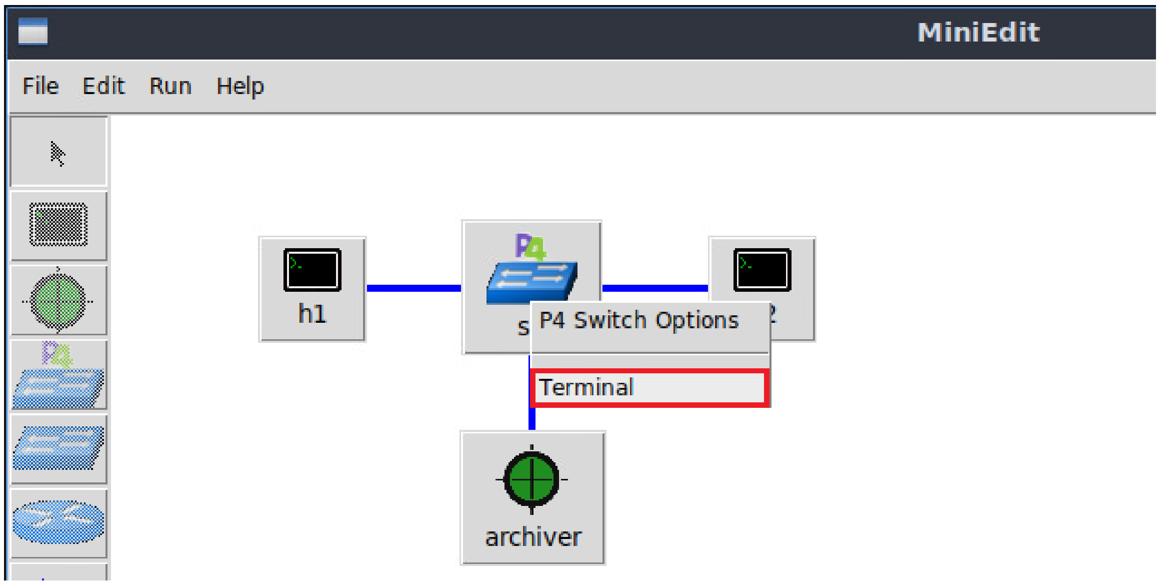
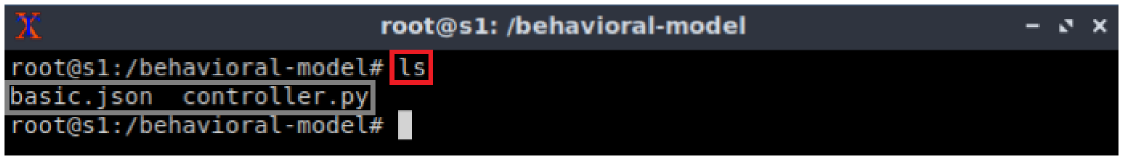


Figure 47. Starting the terminal on the switch.

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

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

```
ls
```



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

Figure 48. Displaying the contents of the current directory in the switch s1.

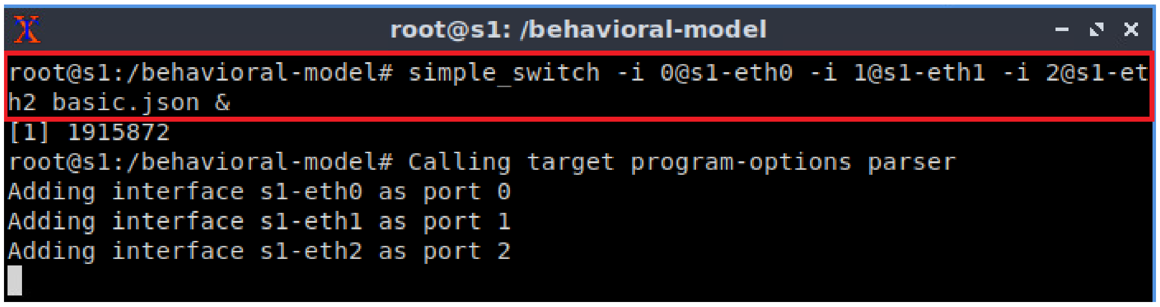
The figure above shows that the switch contains the *basic.json* and *controller.py* files that were pushed after compiling the P4 program and creating the controller application.

5.3 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.

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

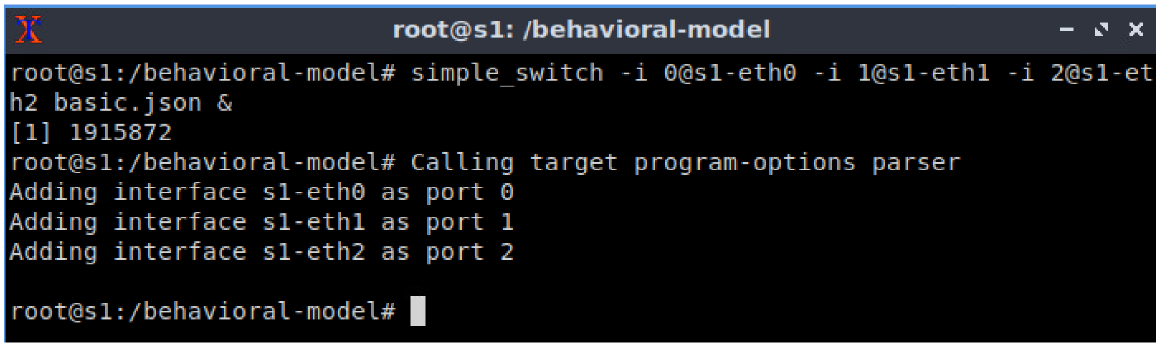
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 1915872
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 49. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 2. In switch s1 terminal, press *Enter* to return the CLI.



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 1915872
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model#
```

Figure 50. Returning to switch s1 CLI.

6 Connecting Grafana to OpenSearch

In this section, you will follow the process of configuring Grafana to access the measurement results stored in the archiver. Grafana operates within a Docker container and can be accessed via a web interface. To enable this connection, the user will need to specify the data source, which, in this scenario, is the OpenSearch database hosting the P4 measurements.

6.1 Running Grafana

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

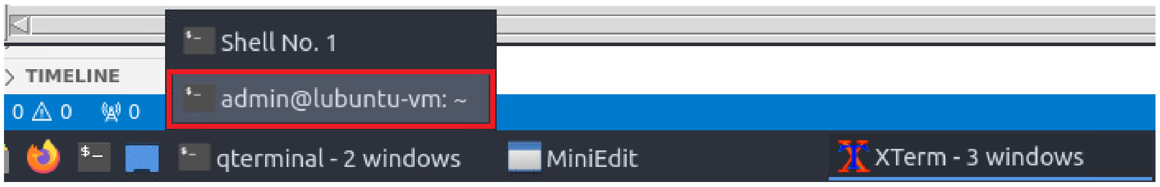
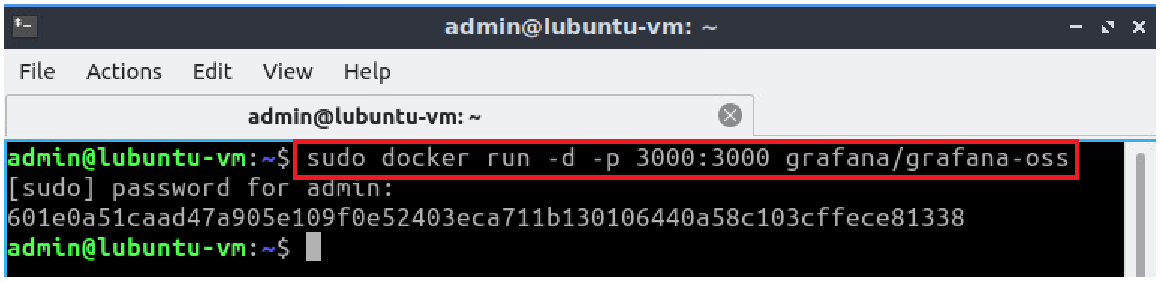


Figure 51. Returning to the Linux Terminal.

Step 2. In the Linux Terminal, issue the following command to start the Docker container that implements Grafana. If a password is required, type `password`.

```
sudo docker run -d -p 3000:3000 grafana/grafana-oss
```



```
admin@lubuntu-vm:~$ sudo docker run -d -p 3000:3000 grafana/grafana-oss
[sudo] password for admin:
601e0a51caad47a905e109f0e52403eca711b130106440a58c103cffece81338
admin@lubuntu-vm:~$
```

Figure 52. Running Grafana.

6.2 Accessing Grafana

Step 1. Open the web browser by clicking on the icon located in the taskbar.

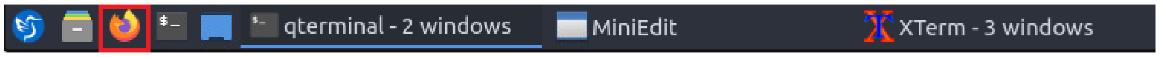


Figure 53. Opening the web browser.

Step 2. In the address bar type the following URL to access Grafana's web interface. An authentication webpage will be displayed.

```
localhost:3000
```

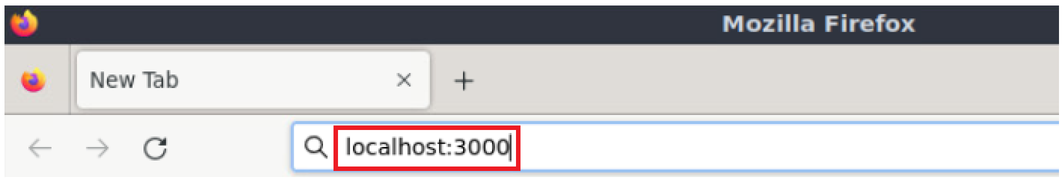


Figure 54. Accessing Grafana's web interface.

Step 3. Type `admin` as the username and `password` as the password. Then, click on *Log In*.

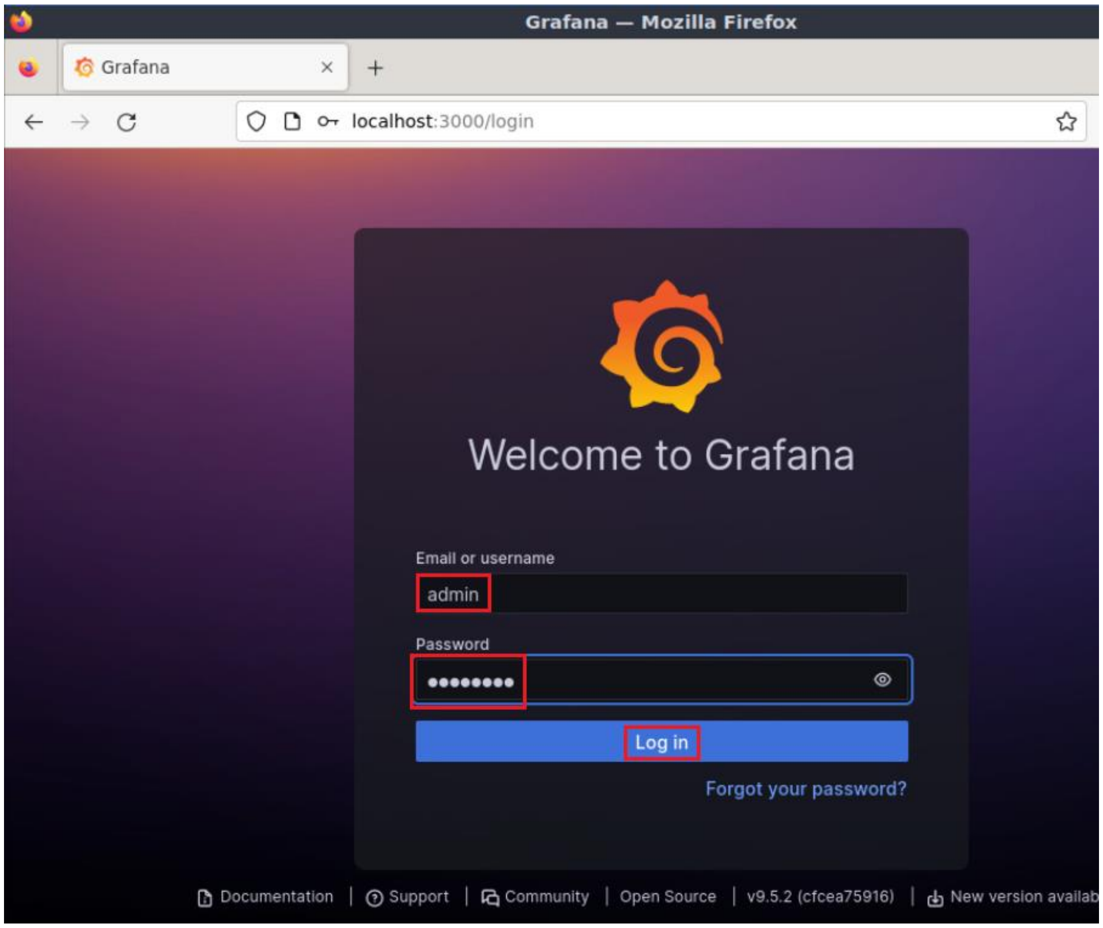

Grafana — Mozilla Firefox
Grafana
localhost:3000/login
Welcome to Grafana
Email or username
admin
Password
Log in
Forgot your password?
Documentation | Support | Community | Open Source | v9.5.2 (cfcea75916) | New version availab

Figure 55. Authenticating the admin user in Grafana's web interface.

6.3 Configuring Grafana's data source

Step 1. Click on the three-line icon next to *Home* and select *Administration>Plugins* as shown in the figure below.

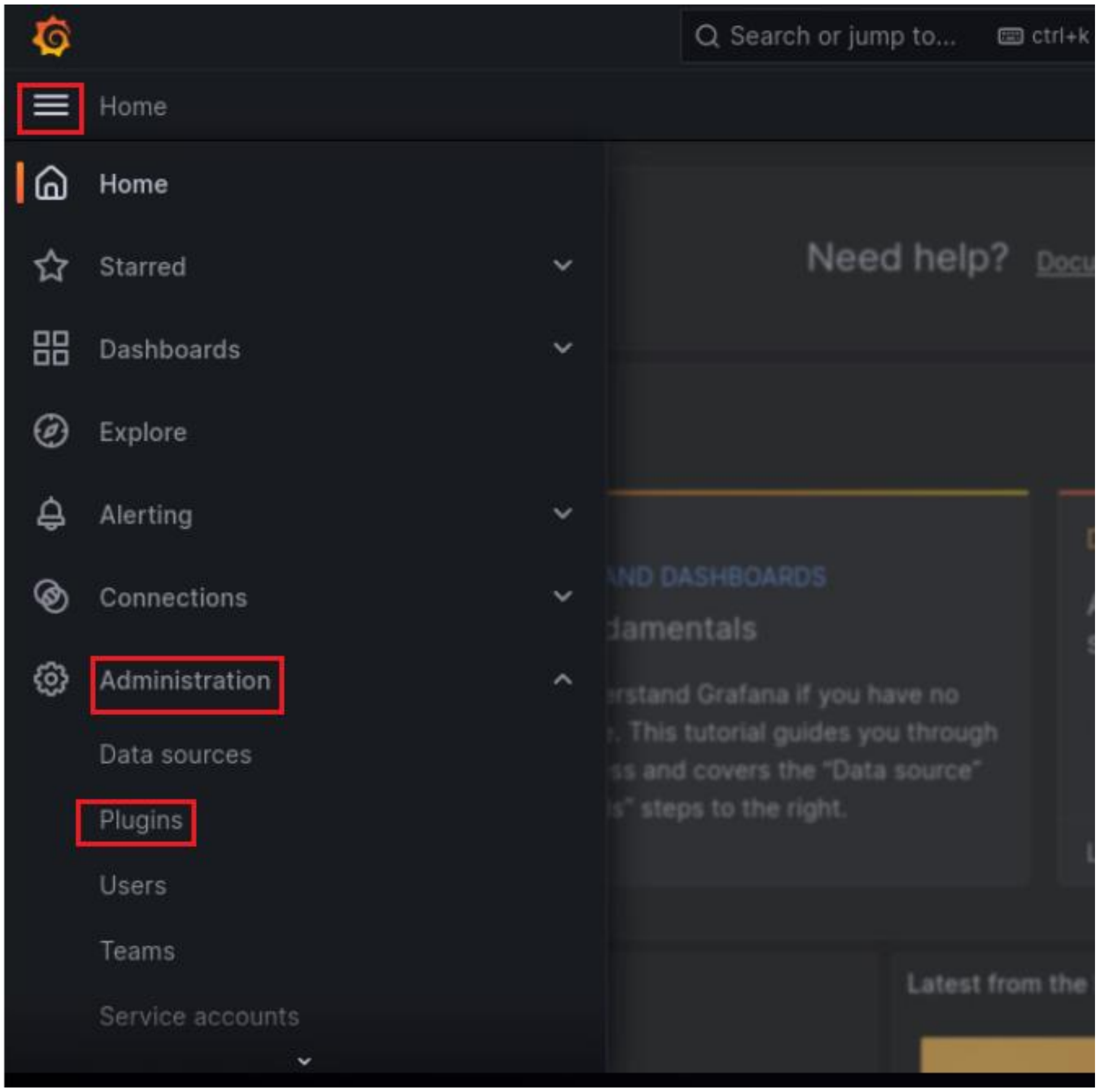


Figure 56. Configuring Grafana's data sources.

Step 2. In the search entry box, type *opensearch.* Then, click on the result as shown in the figure below.

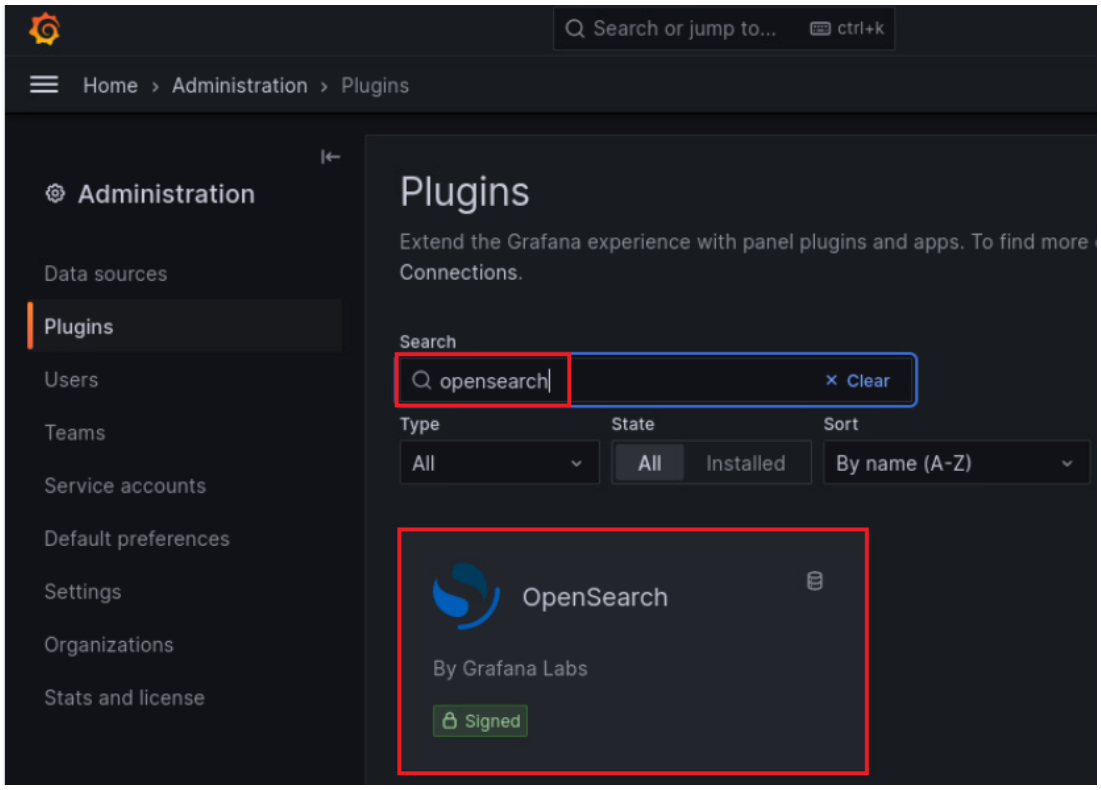


Figure 57. Searching for the OpenSearch plugin.

Step 3. Click on *Install* to install the plugin.

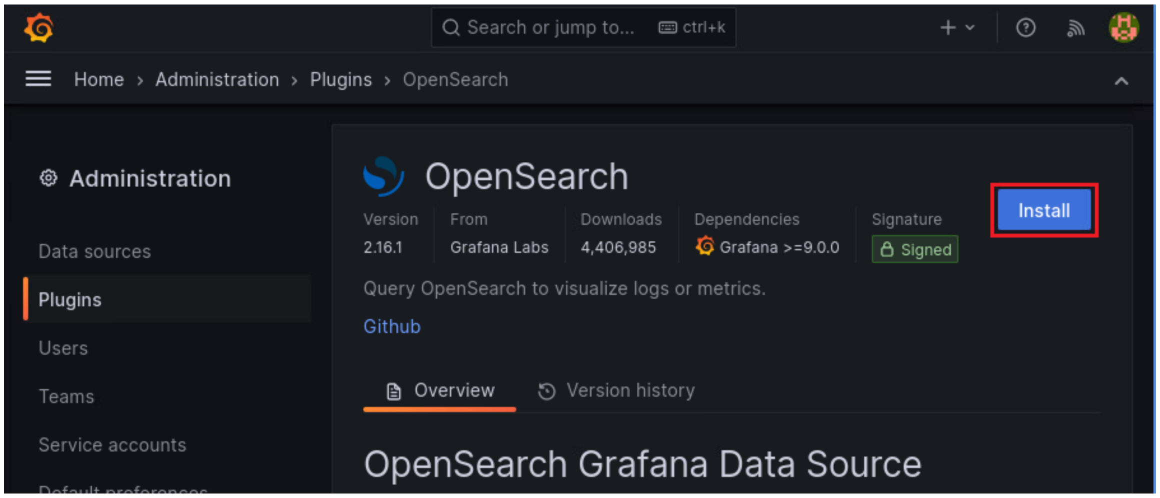


Figure 58. Installing the OpenSearch plugin.

6.4 Configuring an OpenSearch data source

Step 1. Once the installation is complete, click on *Create a OpenSearch data source.*

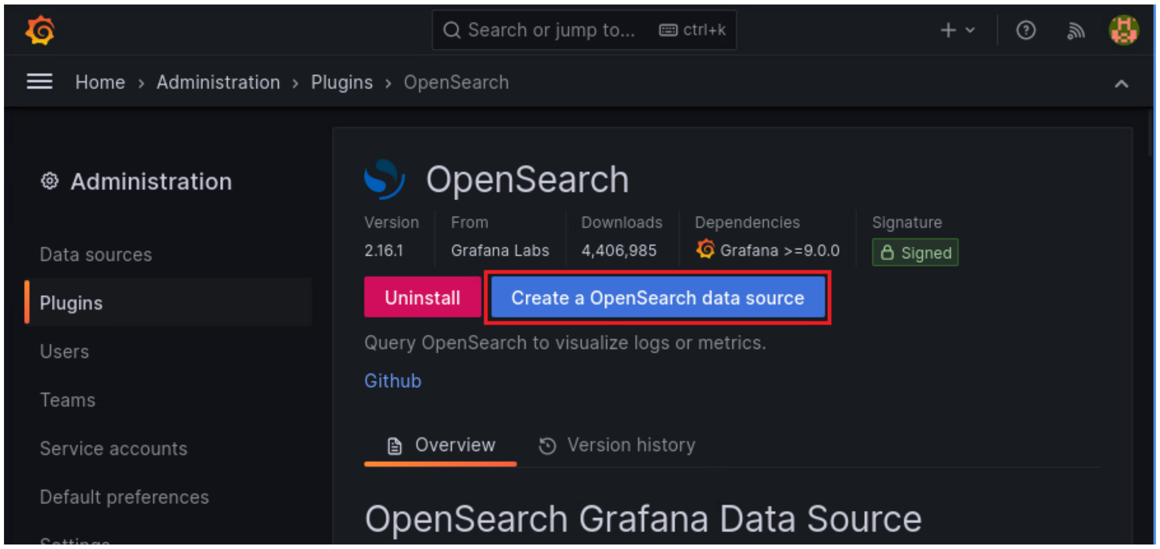


Figure 59. Creating an OpenSearch data source.

Step 2. Set the data source name as *P4 measurements* as shown in the figure below.

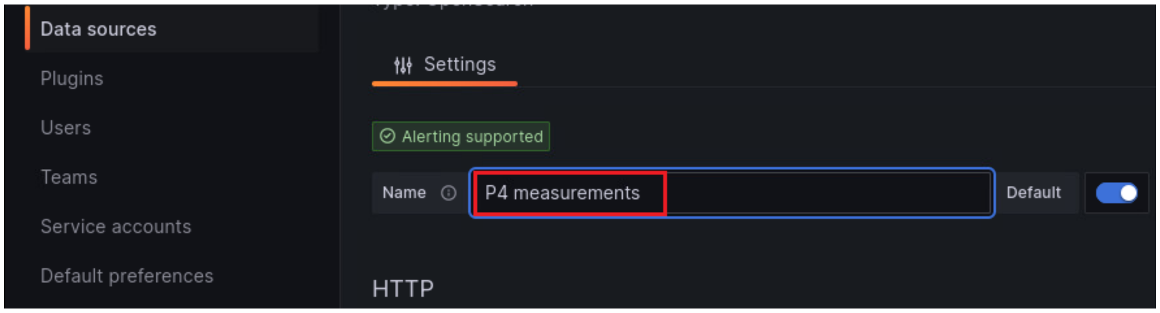


Figure 60. Setting the data source name.

Step 3. Configure the following URL as shown in the figure below. This URL points towards the database of perfSONAR1.

```
https://172.17.0.2/opensearch
```

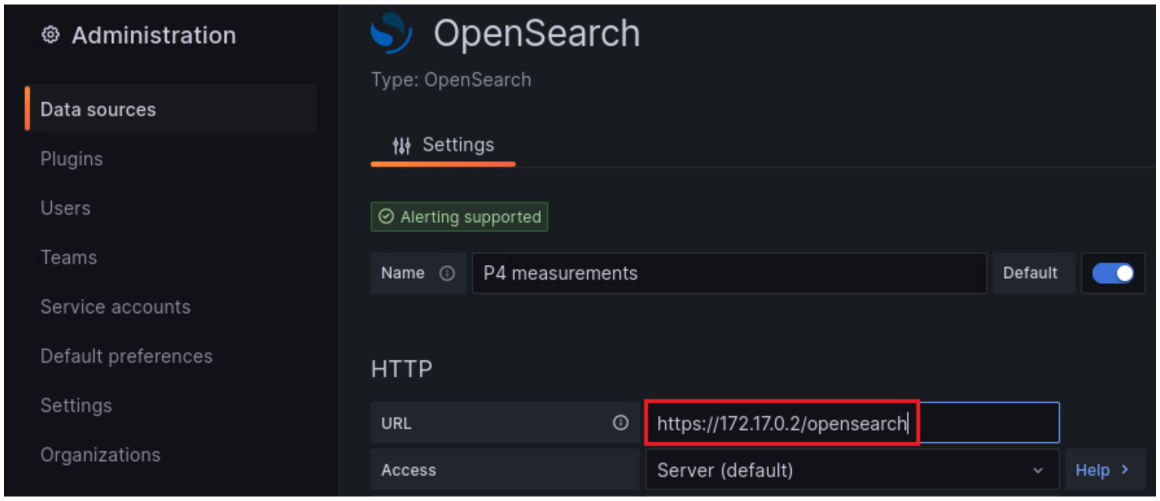


Figure 61. Setting the data source URL.

Step 4. Scroll down and enable the *Basic auth* option as shown in the figure below.

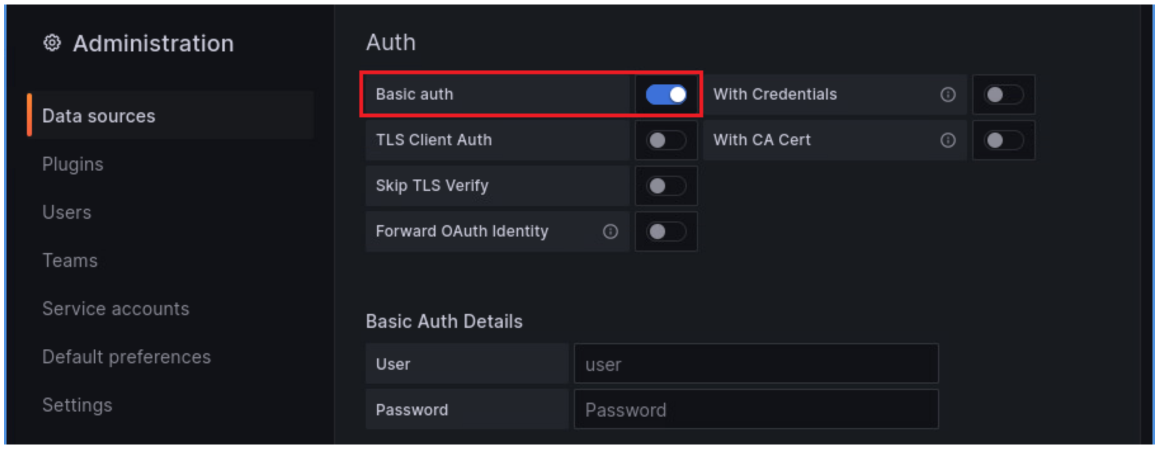


Figure 62. Enabling basic authentication.

Step 5. In the *Auth* section, toggle on the *Skip TLS Verify* option.

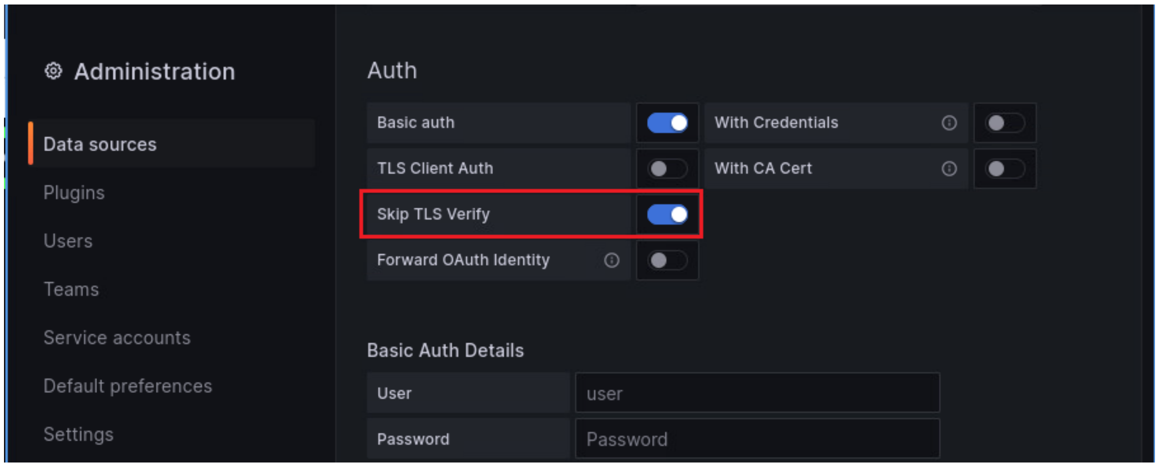
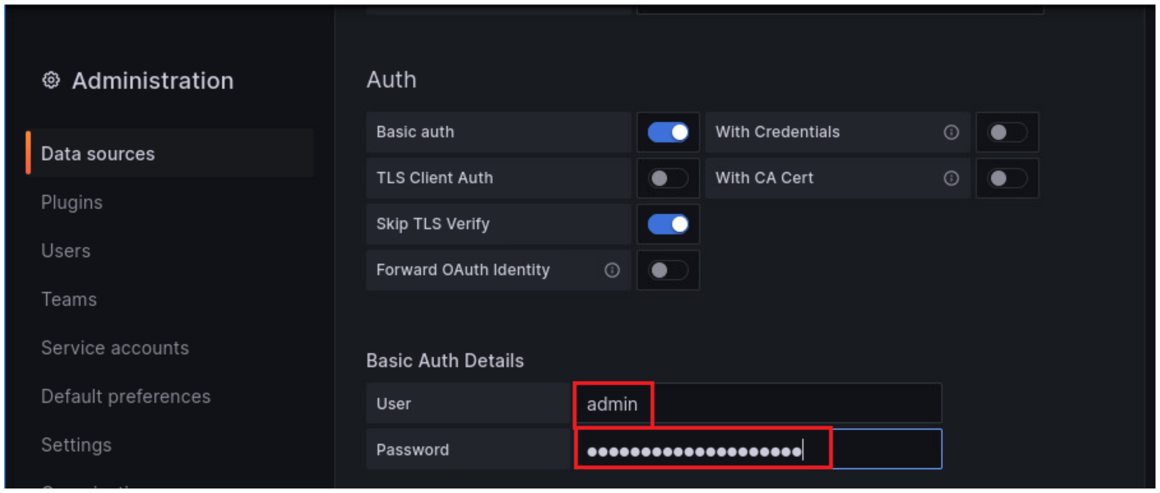


Figure 63. Skipping TLS verification.

Step 6. In Grafana, input the credentials as the username and password.

- User: `admin`
- Password: `1NBIAomOxvbVq9hiutdY`

Note that you can copy and paste this password from the file called *OpenSearch_password.txt* located on the Desktop.



Auth

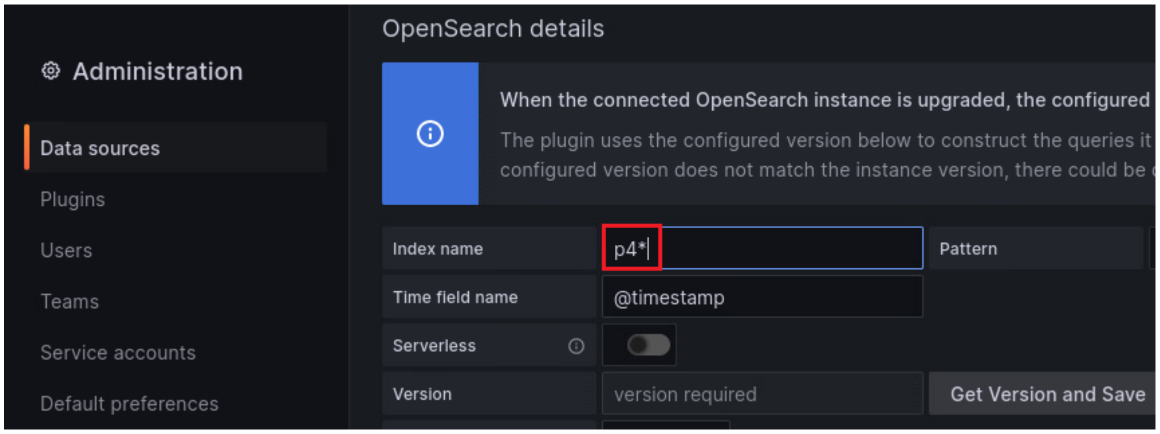
Basic auth	On	With Credentials	Off
TLS Client Auth	Off	With CA Cert	Off
Skip TLS Verify	On		
Forward OAuth Identity	Off		

Basic Auth Details

User	admin
Password	●●●●●●●●●●●●●●●●●●●●

Figure 64. Authenticating into the OpenSearch database.

Step 7. In Grafana, scroll down to the *OpenSearch details* section and type `p4*` for the *index name* field. `p4*` character implies that any index that starts with the pattern `p4` should be considered.



OpenSearch details

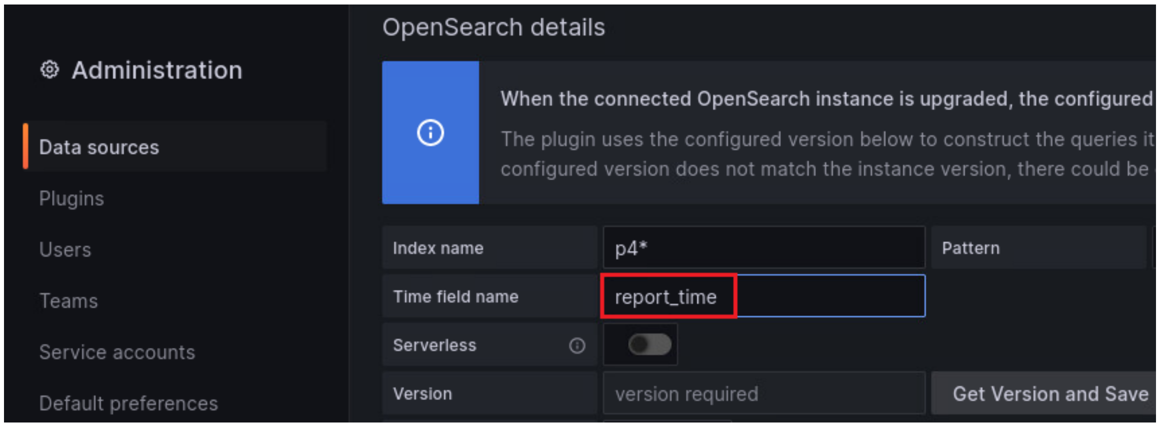
When the connected OpenSearch instance is upgraded, the configured
The plugin uses the configured version below to construct the queries it
configured version does not match the instance version, there could be

Index name	p4*
Time field name	@timestamp
Serverless	Off
Version	version required

Figure 65. Setting the index name.

Step 8. Type the following entry as the *Time field name.*

```
report_time
```



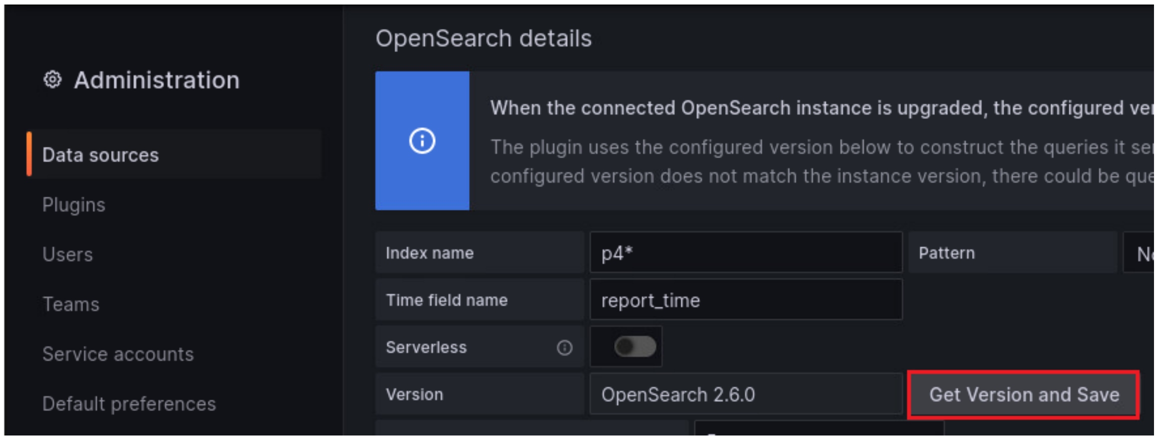
OpenSearch details

When the connected OpenSearch instance is upgraded, the configured
The plugin uses the configured version below to construct the queries it
configured version does not match the instance version, there could be

Index name	p4*
Time field name	report_time
Serverless	Off
Version	version required

Figure 66. Setting the time field name.

Step 9. Click on *Get Version and Save* to test the connection with the OpenSearch database. The output will indicate the current version.


OpenSearch details

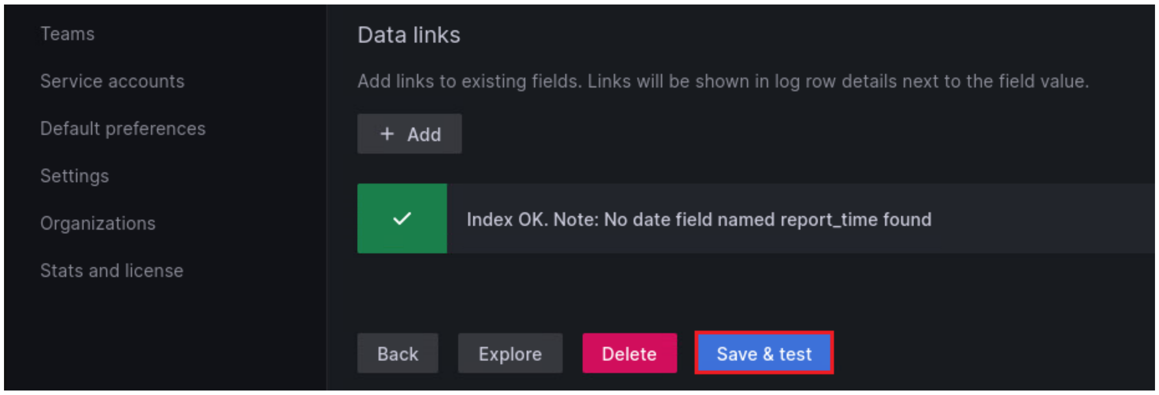
When the connected OpenSearch instance is upgraded, the configured ve

The plugin uses the configured version below to construct the queries it se configured version does not match the instance version, there could be que

Index name: p4*
Pattern: N
Time field name: report_time
Serverless
Version: OpenSearch 2.6.0

Figure 67. Getting the database version.

Step 10. Scroll down and click on *Save & test* to save the configured data source.


Data links

Add links to existing fields. Links will be shown in log row details next to the field value.

Index OK. Note: No date field named report_time found

Figure 68. Saving the configuration.

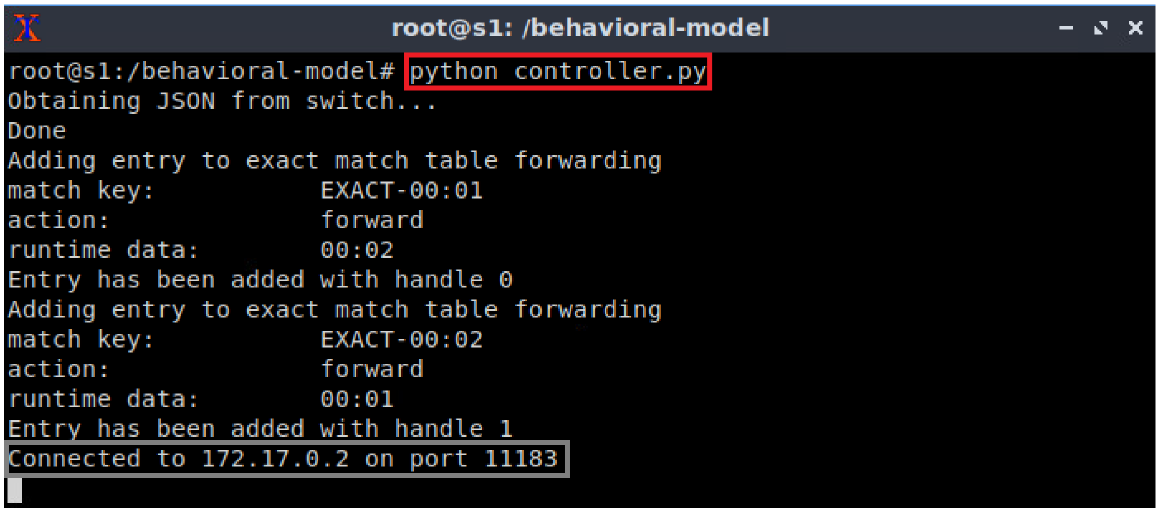
7 Testing and verifying the application

This section shows the steps to run a controller and observe the pre-flow throughput monitored by the application.

7.1 Starting the controller application

Step 1. In switch s1 terminal, start the controller by running the following command.

```
python controller.py
```



```
root@s1:/behavioral-model# python controller.py
Obtaining JSON from switch...
Done
Adding entry to exact match table forwarding
match key:           EXACT-00:01
action:              forward
runtime data:        00:02
Entry has been added with handle 0
Adding entry to exact match table forwarding
match key:           EXACT-00:02
action:              forward
runtime data:        00:01
Entry has been added with handle 1
Connected to 172.17.0.2 on port 11183
```

Figure 69. Starting the controller in switch s1.

The figure above shows that the control play has successfully connected to Logstash running on the archiver.

7.2 Starting data transfer between h1 and h2

Step 1. Right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

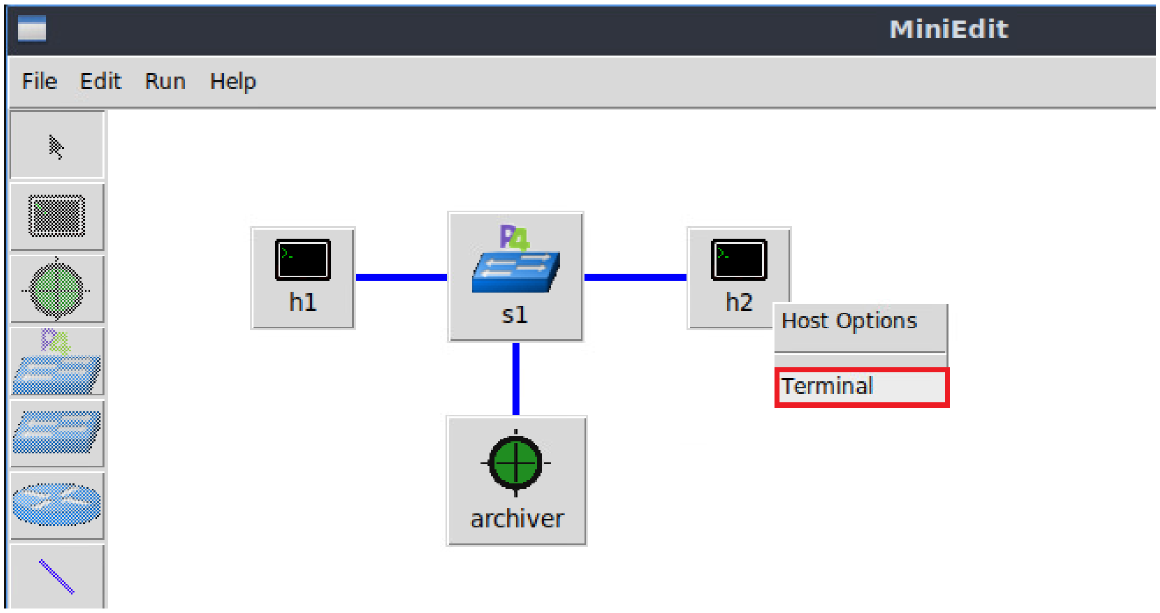
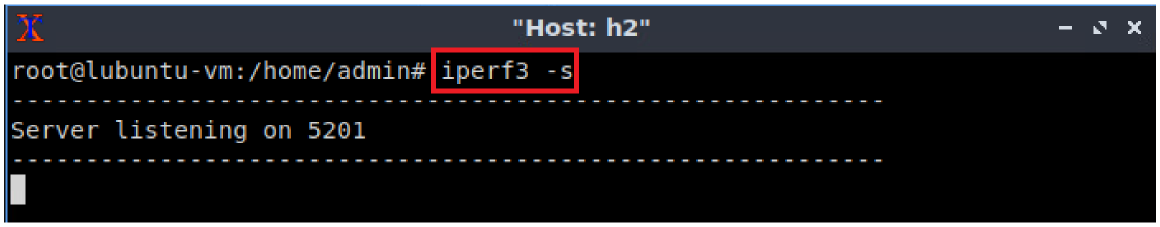


Figure 70. Opening a terminal on host h2.

Step 2. On host h2's terminal, type the following command to start Iperf3 server. `iperf3` tool is used to perform network throughput tests. `-s` option specifies that h2 will be acting as the server.

```
iperf3 -s
```

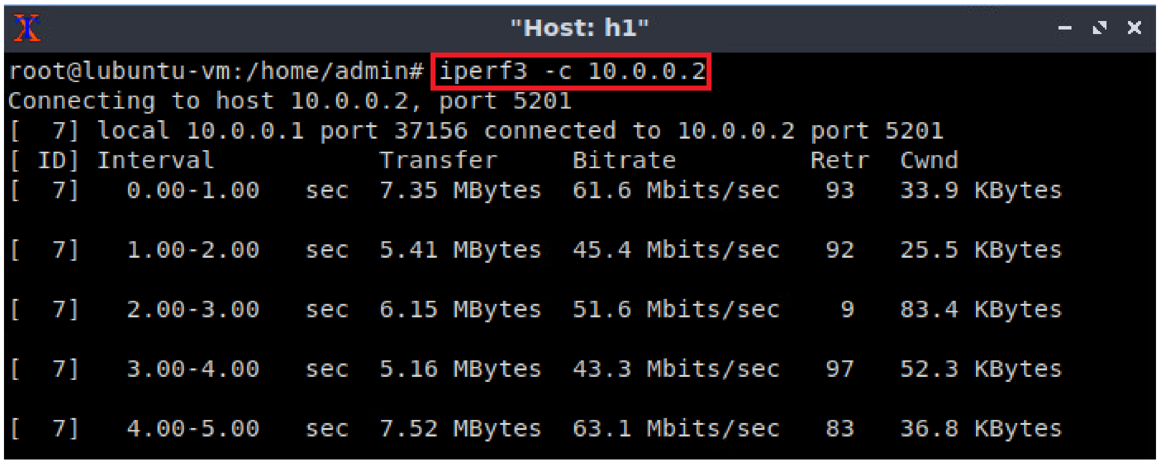


```
root@lubuntu-vm:/home/admin# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 71. Starting Iperf3 server on h2.

Step 3. On host h1's terminal, type the following command to start data transfer with h2. `-c` option specifies that h1 will be acting as the client.

```
iperf3 -c 10.0.0.2
```



```
root@lubuntu-vm:/home/admin# iperf3 -c 10.0.0.2
Connecting to host 10.0.0.2, port 5201
[  7] local 10.0.0.1 port 37156 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  7]   0.00-1.00   sec  7.35 MBytes  61.6 Mbits/sec   93   33.9 KBytes

[  7]   1.00-2.00   sec  5.41 MBytes  45.4 Mbits/sec   92   25.5 KBytes

[  7]   2.00-3.00   sec  6.15 MBytes  51.6 Mbits/sec    9   83.4 KBytes

[  7]   3.00-4.00   sec  5.16 MBytes  43.3 Mbits/sec   97   52.3 KBytes

[  7]   4.00-5.00   sec  7.52 MBytes  63.1 Mbits/sec   83   36.8 KBytes
```

Figure 72. Sending data between h1 and h2.

7.3 Displaying the measurement results in a dashboard

Step 1. Navigate back to Grafana.

Step 2. Click on the + icon in the upper right of the screen and select *New dashboard*.

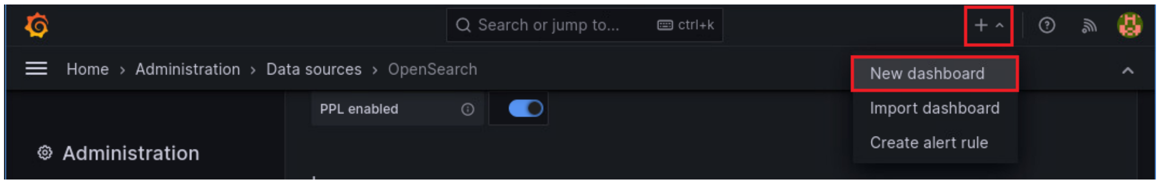


Figure 73. Creating a new dashboard.

Step 3. Click on *+ Add visualization.*

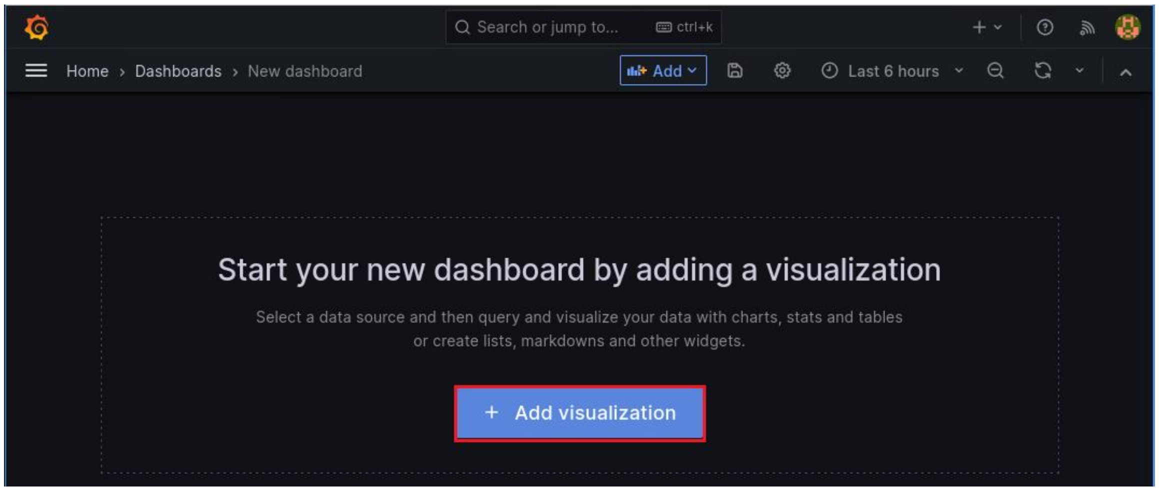
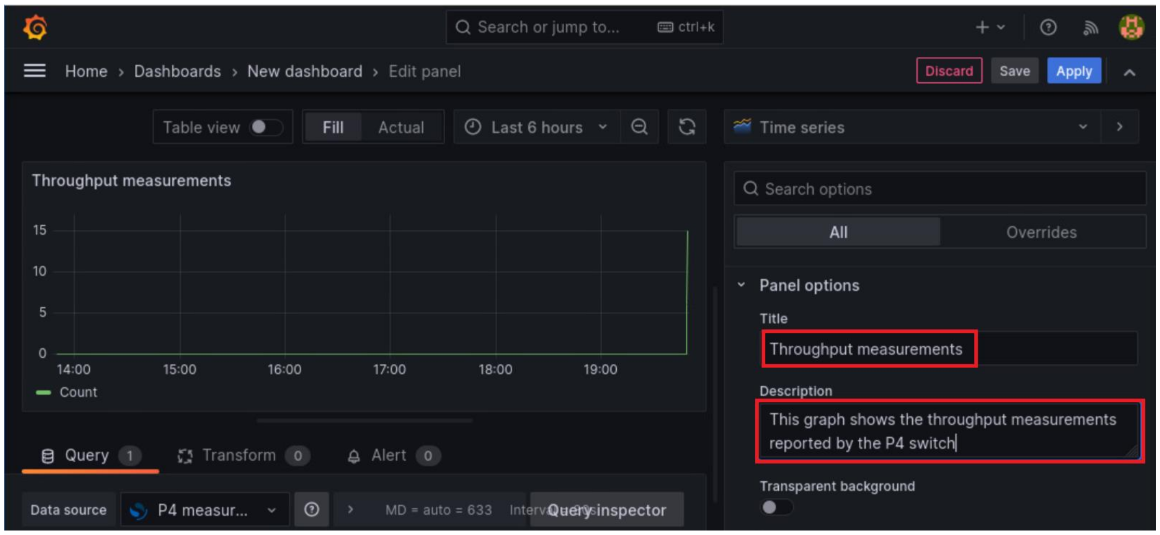

Search or jump to...
ctrl+k
Home › Dashboards › New dashboard
Add
Last 6 hours
Start your new dashboard by adding a visualization
Select a data source and then query and visualize your data with charts, stats and tables or create lists, markdowns and other widgets.
+ Add visualization

Figure 74. Adding a graph.

Step 4. On the right panel, in the *Panel options,* add a *Title* and a *Description*. This graph will display the throughput tests.



Title: Throughput measurements

Description: This graph shows the throughput measurements reported by the P4 switch

Figure 75. Setting the graph title and the description.

Step 5. Change the time range by clicking on the icon located on the upper part and selecting the *Last 5 minutes* option.

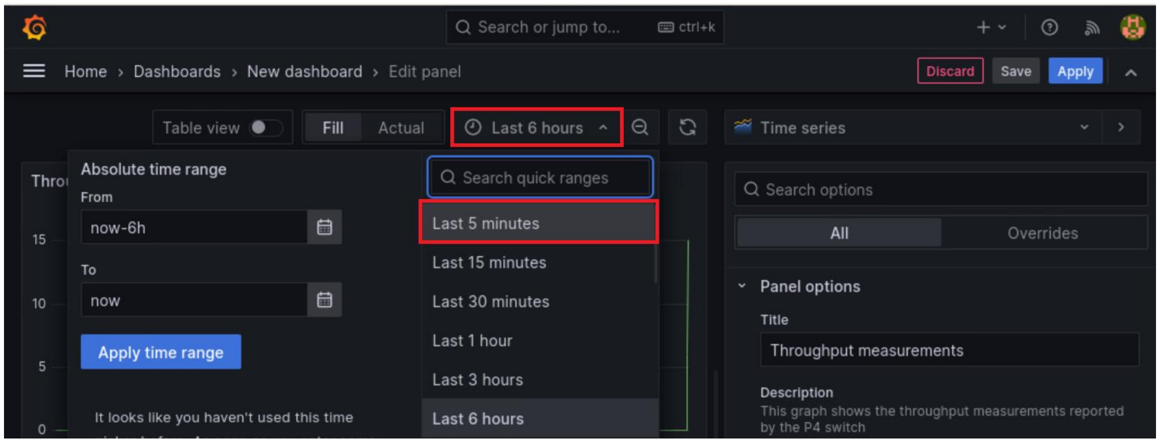


Figure 76. Adjusting the time range to 5 minutes.

Step 6. Press the refresh button to see the changes.

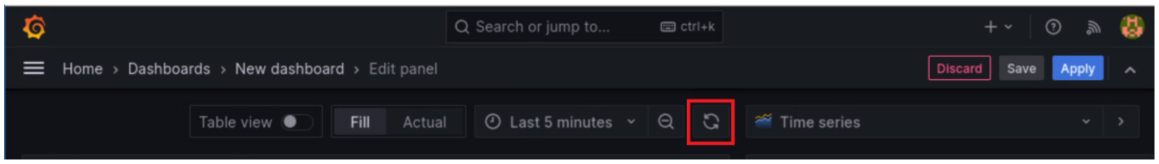


Figure 77. Refreshing the Panel.

Step 7. In the *Querry* panel, change the metric from *Count* to *Average*, as shown in the figure below.

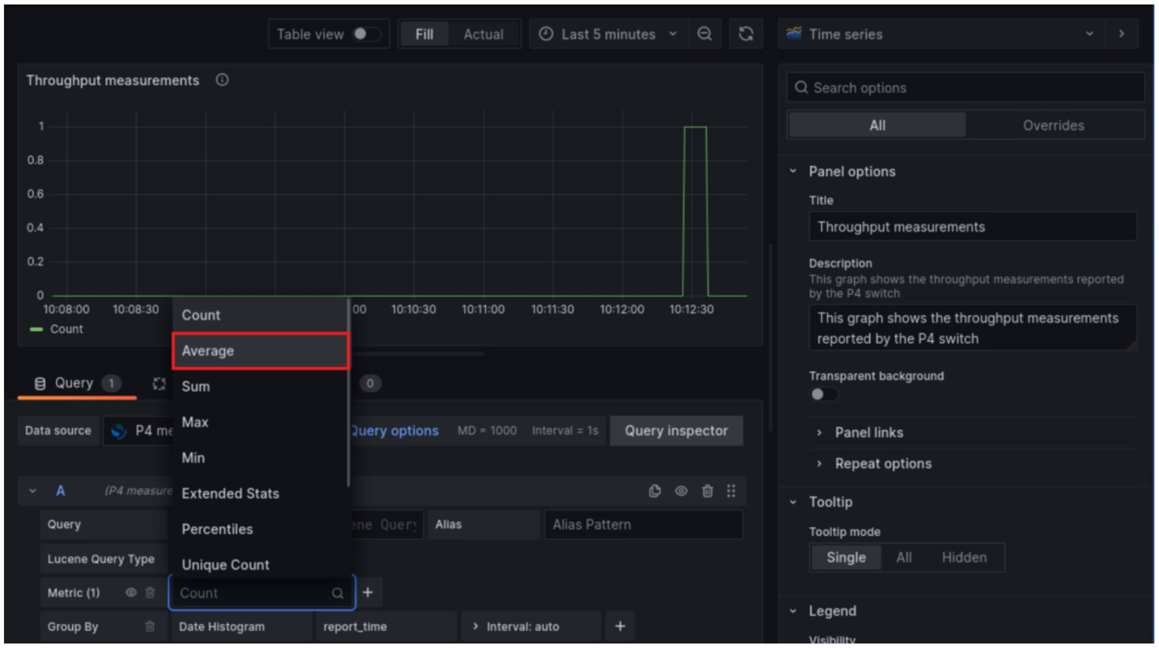


Figure 78. Displaying the maximum values.

Step 8. In the adjacent box, select *throughput* from the list.

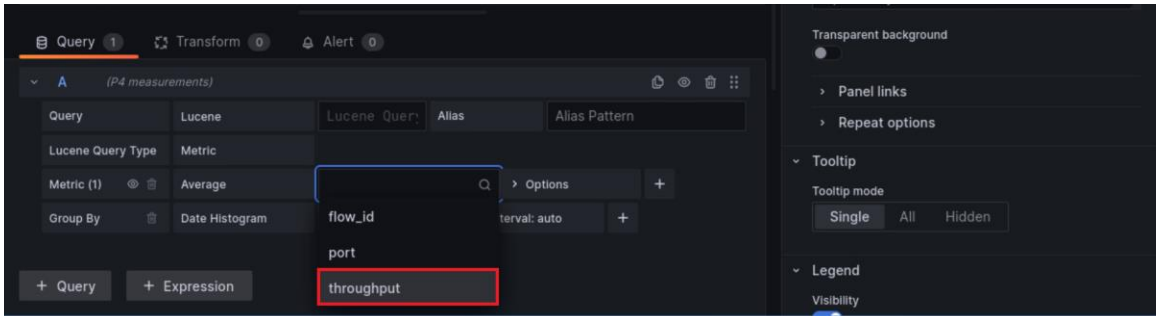


Figure 79. Selecting the throughput metric.

Step 9. Scroll down and click on the + next to the *Group by* row.

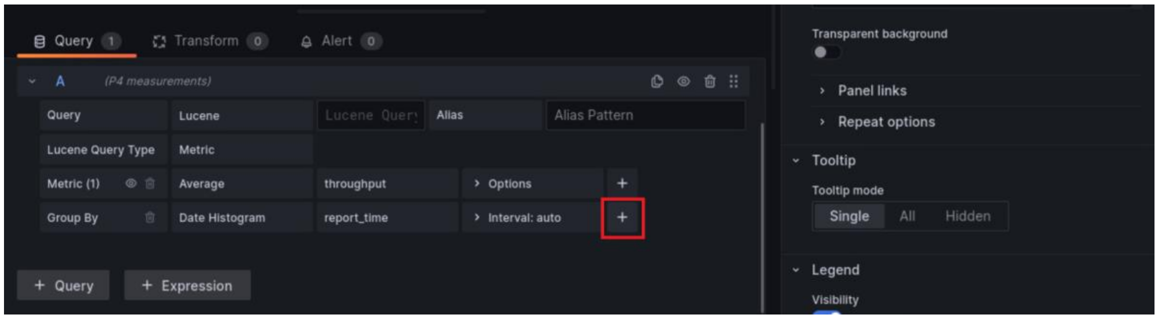


Figure 80. Selecting the throughput results.

Step 10. In the newly added row, select `flow_id` to group the measurements into their corresponding flows.

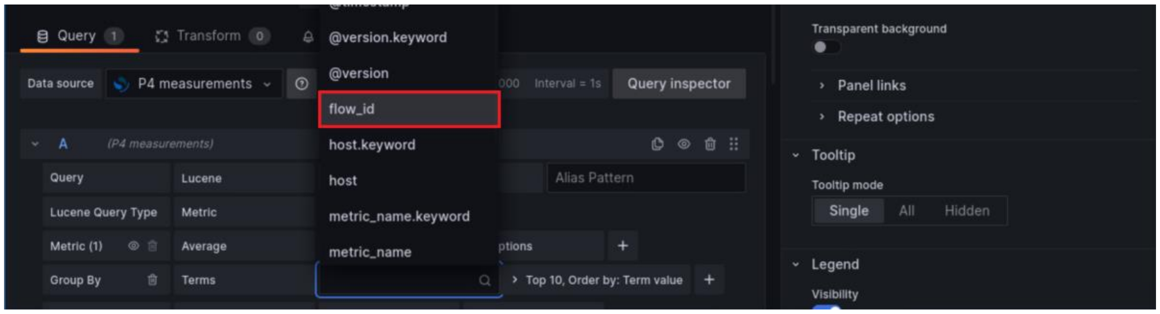


Figure 81. Selecting the throughput results.

Step 11. Click on the *Top 10, Order by: Term Value*, and set *Min Doc Count* to 1. Now Grafana will only display the flows with at least one reported measurement in the last 5 minutes.

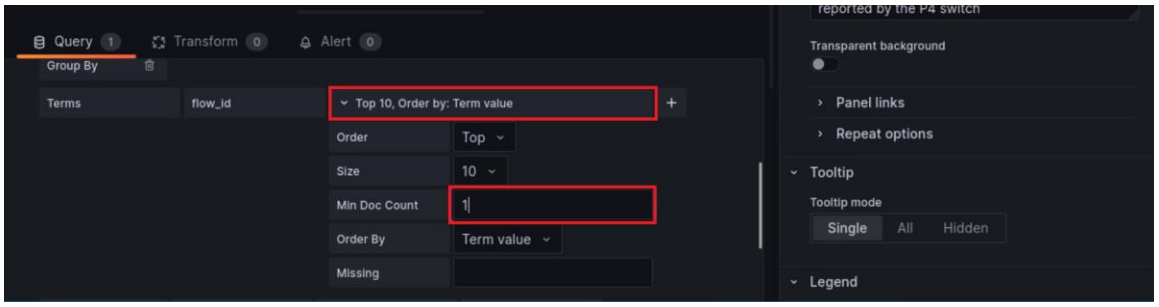
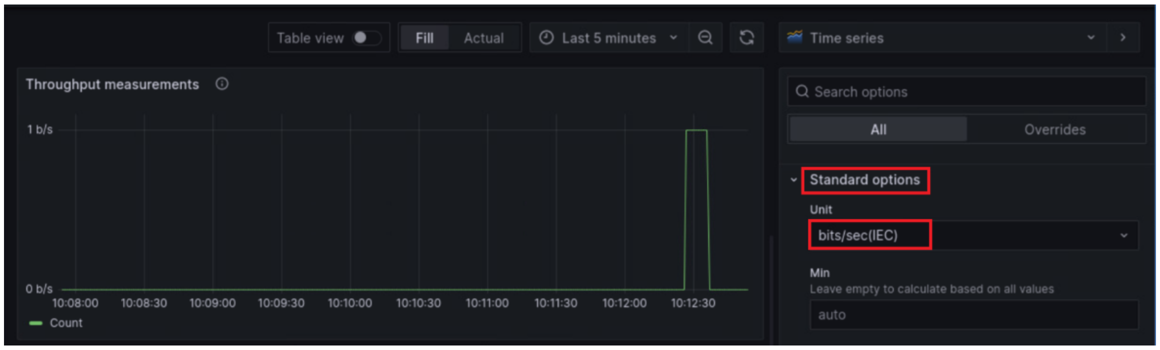


Figure 82. Setting *Min Doc Count* to 1.

Step 12. In the panel on the right, scroll down to reach the *Standard options* section. Select *Data rate/ bits/sec(IEC)* option for the *Unit* field to change the unit bits/second.



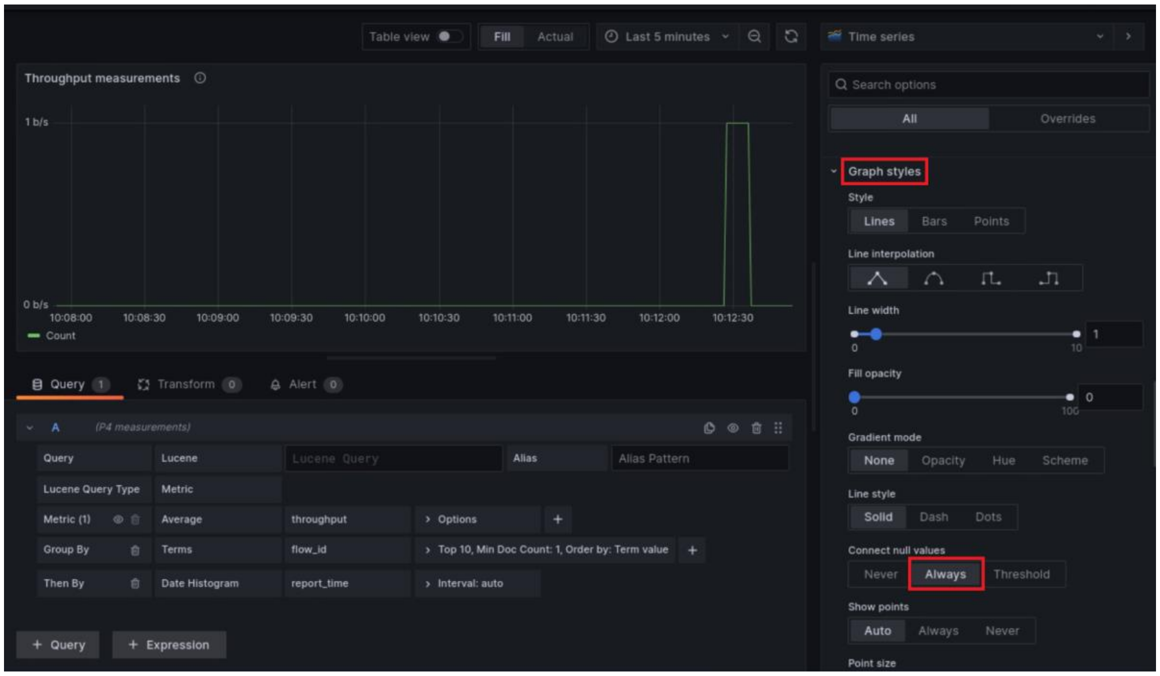
Standard options

Unit: bits/sec(IEC)

Min
Leave empty to calculate based on all values
auto

Figure 83. Adjusting unit to bits per second.

Step 13. In the panel on the right, Scroll up to reach the *Graph syles* section. Select *Always* option for the *Connect null values* field so the throughput measurements will be displayed as a graph and not as points.



A (P4 measurements)

Query	Lucene	Lucene Query	Alias	Alias Pattern
Lucene Query Type	Metric			
Metric (1)	Average	throughput	Options	
Group By	Terms	flow_id	Top 10, Min Doc Count: 1, Order by: Term value	
Then By	Date Histogram	report_time	Interval: auto	

Graph styles

Style: [✓ Lines] Bars Points

Line interpolation

Line width: 1

Fill opacity: 0

Gradient mode: [✓ None] Opacity Hue Scheme

Line style: [✓ Solid] Dash Dots

Connect null values: Never [✓ Always] Threshold

Show points: [✓ Auto] Always Never

Point size

Figure 84. Adjusting unit to bits per second.

Step 14. Click on *Apply* to save the changes.

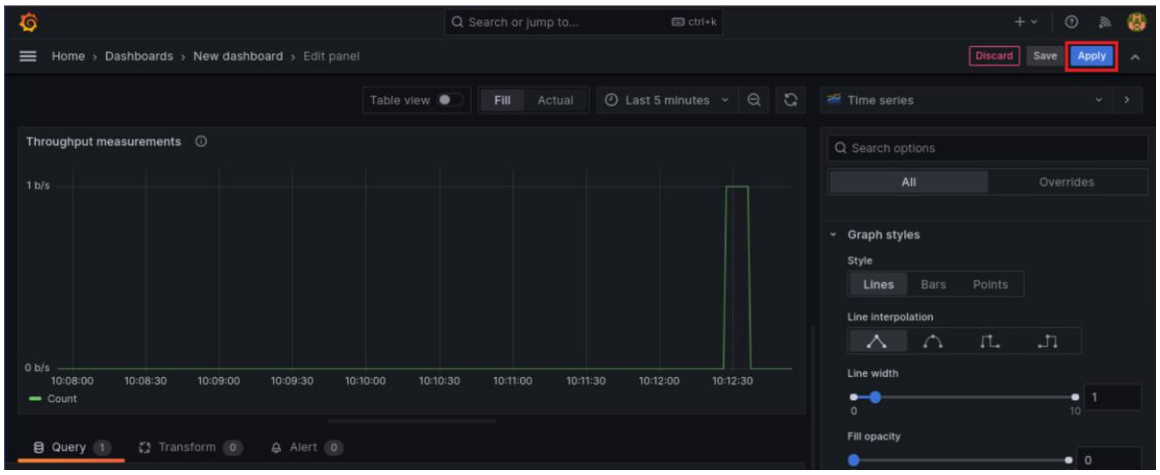


Figure 85. Applying the changes to the dashboard.

Step 15. Click on the ^ sign at the top right corner and select 5s. Grafana will now refresh the Panel every 5 seconds.

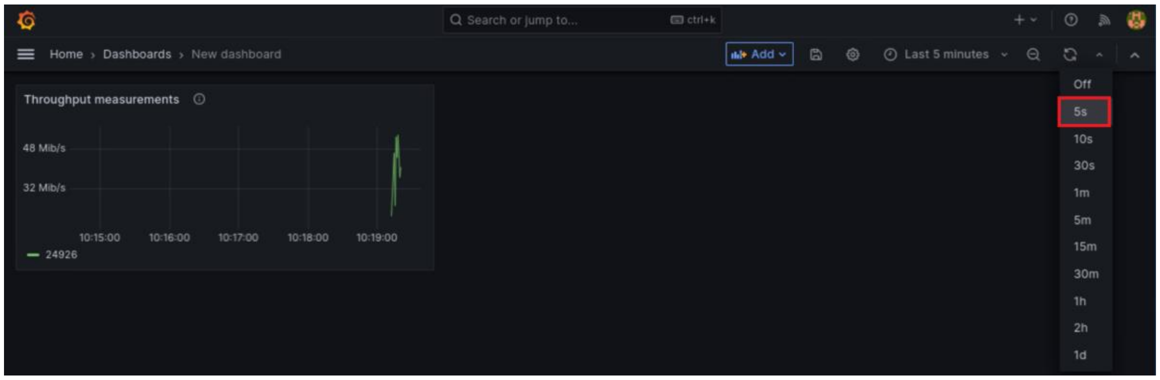


Figure 86. Setting auto-refresh interval.

Step 16. Extend the Panel to the maximum length possible.

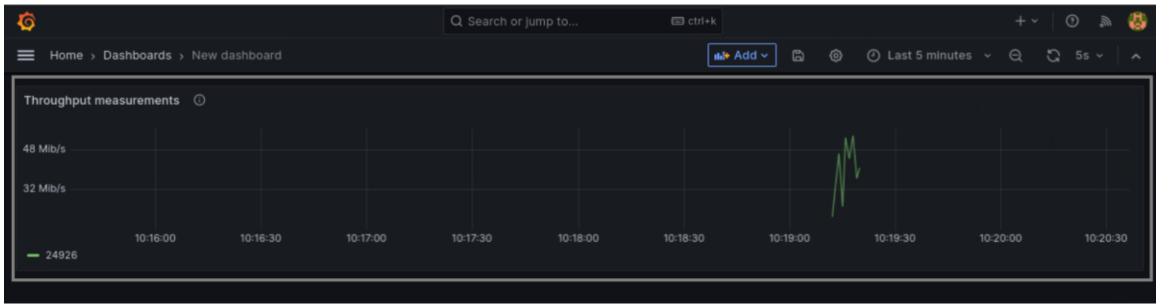
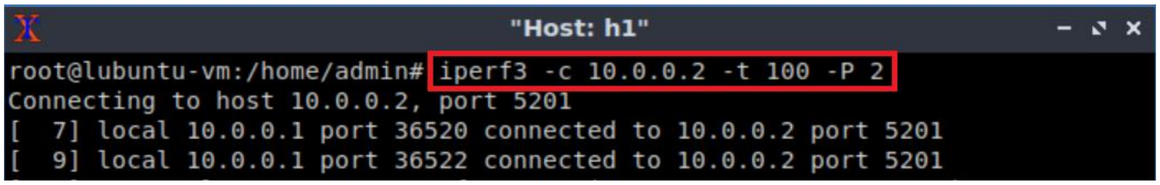


Figure 87. Extending panel's length.

Step 17. On the h1 terminal, type the command below to run an iperf3 test between h1 and h2. Use `-t 100` to set the duration of the set to 100 seconds. Use `-p 2` to set the number of data transfers to two.

```
iperf3 -c 10.0.0.2 -t 100 -P 2
```



```
root@lubuntu-vm:/home/admin# iperf3 -c 10.0.0.2 -t 100 -P 2
Connecting to host 10.0.0.2, port 5201
[  7] local 10.0.0.1 port 36520 connected to 10.0.0.2 port 5201
[  9] local 10.0.0.1 port 36522 connected to 10.0.0.2 port 5201
```

Figure 88. Starting two data transfers between h1 and h2.

Step 18. Navigate back to Grafana and monitor the throughput of the two data transfers.

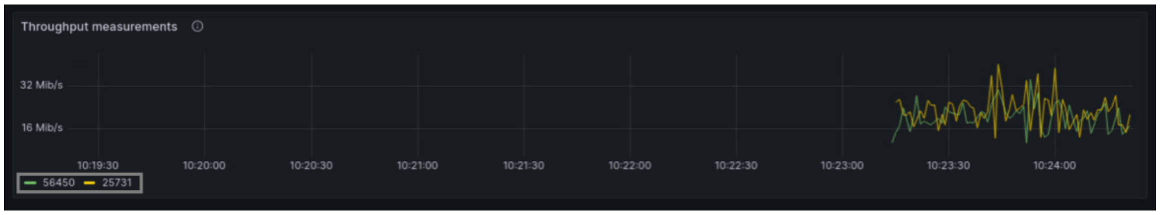


Figure 89. Monitoring the throughput of the iperf3 test.

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

References

1. perfSONAR Project . “Deploying a Central Measurement Archive.” [Online]. Available: https://docs.perfsonar.net/multi_ma_install.html
2. Logstash. [Online]. Available: https://www.elastic.co/logstash
3. OpenSearch. Available: https://opensearch.org/