Exercises High Performance Data Networking

Niels van Adrichem Chenxing Ji Fernando Kuipers[1] Jorik Oostenbrink
Belma Turkovic Adrian Zapletal

[1]Responsible lecturer.

Chapter 1

Lecture 1: Introduction to Mininet

Solution

The solution files are available here: https://surfdrive.surf.nl/files/index.php/s/v9x9Fn9bWPQ5naL

1.1 Important Notice

If you run into problems while going through any of the exercises, first try reading through the exercise again and fixing the problem by yourself. If this does not help, you can ask for assistance at the Q&A sessions or via the discussion forum.

At the end of the first and second chapters, you will find a summary of useful links and commands.

Another note: we have experienced that when copying commands or code directly from the reader (this document) into terminal, the terminal sometimes does not recognize certain special characters (e.g., ' or -).

1.2 Environment Setup

As you probably do not have access to a network of OpenFlow or P4 switches, you will run your programs on emulated networks instead. For this purpose we will use Mininet[1]. Mininet emulates switches using Open vSwitch, a popular and open-source virtual switch that is used in both hardware switches sold by vendors and software switches that can be installed on generic computer hardware. In this section, we will download and set up a Virtual Machine containing a pre-installed version of Mininet and all other software required for the course exercises. We use different tools and VMs for x86- and ARM-based systems. Inside the VM, there should be no differences. If you have a Mac with M1 or M2 CPU, follow the instructions for ARM, otherwise x86. If you want to install everything yourself and not use the provided VM, you can download: https://surfdrive.surf.nl/files/index.php/s/CSDP224IId0eFpK

x86 Install the open-source VirtualBox hypervisor, found here: https://www.virtualbox.org/wiki/Downloads.

The x86 VM image can be found here:
https://drive.google.com/file/d/1oKpnLdGdJ2ETfo29UBguhDnaMaMef-5N/view?usp=sharing

The Mininet and VirtualBox websites and communities provide ample information on installation issues. *We will not provide installation support for these tools.*

After downloading the image, start VirtualBox and import the VM by executing the following steps:

1. Import the VM by opening `File -> Import Appliance...` from the menu and selecting the image
2. Start the VM. The default username / password combination is: *hpdn* / *mininet*

ARM (Mac with M* CPU) Download the open-source QEMU-based hypervisor UTM (https://github.com/utmapp/UTM) from this link: https://github.com/utmapp/UTM/releases/latest/download/UTM.dmg

The ARM VM image can be found here:
https://drive.google.com/file/d/1deICH4h2LsXIXbhyQ95BRzllEjdI1mog/view?usp=sharing

After downloading and unzipping the image, start UTM and import the VM by executing the following steps:

1. Import the VM by selecting + and then *Open ...*; in this menu, select the downloaded image
2. Start the VM. The default username / password combination is: *hpdn* / *mininet*

[1] http://mininet.org/

1.3 Useful information about Mininet

Open a terminal, and start Mininet by running:

```
sudo mn
```

By default, Mininet will start a virtual network with 2 hosts, h1 and h2, connected via switch s1. Run `?` to view all possible commands. Feel free to play around with them, e.g. run `pingall` to confirm connectivity between all hosts.

Each host has a separate network namespace (but hosts share access to all other system resources, such as the filesystem). Within Mininet, you can run most Linux commands directly on any of the virtual hosts by prepending the command by its hostname. For example, you can ping h2 from h1 by running `h1 ping h2` (Mininet automatically replaces the second "h2" with h2's IP address). You can cancel the ping by pressing Ctrl+C. Analogously, you can also start different programs, services, or scripts on any of the virtual hosts, simply by prepending the relevant commands by a hostname.

Tip: By appending & to a command the process will run in the background and the Mininet window isn't blocked.

You can emulate different network topologies by adding the `--topo` option to the start command for Mininet. Mininet itself comes with the following built-in topologies:

- **minimal**: The default topology of 1 switch with 2 hosts. No further parameters apply.
- **single**: A star topology with a single switch and h hosts. This topology can be used by appending `--topo single,h` to the mn command, where h refers to the number of hosts.
- **reversed**: Equivalent to the single switch topology, except that hosts connect to the switch in reverse order (i.e. the highest host number gets the lowest switch port).
- **linear**: h switches connect in a line, and there is one host connected to each switch. To use this topology, append `--topo linear,h` to the mn command.
- **tree**: A binary tree topology of depth d. To use this topology, append `--topo tree,d` to the mn command.

By default, hosts are assigned randomly generated MAC addresses. By appending `--mac` to the mn command, you can make Mininet use more readable MAC and IP addresses. This can be very helpful when debugging your controller application.

You can exit Mininet by typing `exit` in the terminal.

If Mininet crashes or you terminated it without using the `exit` command (for example with Ctrl+C), you will need to clean the Mininet environment with the following command:

```
sudo mn −c
```

For more information on how to create a Mininet network, you can run the following command to view the manual of Mininet:

```
man mn
```

1.4 Creating custom topologies

One of the advantages of Mininet is that it enables you to create complex network topologies and run experiments on them without the need to physically create those networks. You can specify your own custom network topologies in Python using a combination of the *addHost*, *addSwitch*, and *addLink* methods of the class *Topo*. To emulate your topology within Mininet, simply use the `--custom` command to load your custom Python script and `--topo` to select the (custom) topology:

```
sudo mn −−custom /path_to_your_topology/topo_file.py −−topo topology_name
```

For example, the following Python script creates a topology of 2 connected switches, adding 1 host to each switch, while the last line maps the topology classes to topology names. These names can then be used with the `--topo` option. The script is provided in the VM in `/home/hpdn/HPDN_Exercises/week_1/custom_topo.py`.

```
from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."
```

```
        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )


topos = { 'mytopo': MyTopo }
```

It is also possible to map multiple topology classes to names in the same Python file, e.g.

```
topos = { 'mytopo': MyTopo, 'othertopo' : OtherTopo }
```

To start a network with the custom topology in this script, use:

```
sudo mn --custom /home/hpdn/HPDN_Exercises/week_1/custom_topo.py --topo mytopo
```

1.4.1 Exercise 1. - Complete graph & Square Lattice

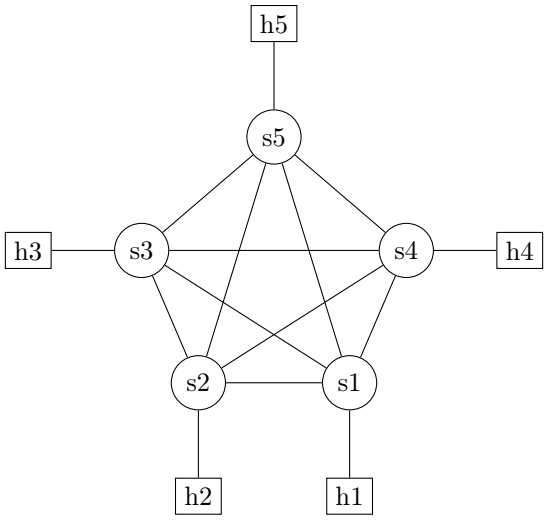


Figure 1.1: Complete graph ($n = 5$).

The following script creates a complete topology of n switches, each with a single host (as in Figure 1.1):

```
from mininet.topo import Topo

class Complete(Topo):
    #switches form a complete graph
    #One host connected to each switch

    #We set a default value for n,
    #so Mininet will not crash if someone creates this topology without specifying n
    def __init__(self, n = 5):
        Topo.__init__(self)

        switches = []
        for i in range(1,n+1):
```

```
            switch = self.addSwitch('s' + str(i))
            host = self.addHost('h' + str(i))
            self.addLink(switch, host)

            for s in switches:
                self.addLink(switch, s)

            switches.append(switch)

topos = {'complete': Complete}
```

The script is included in the VM: `/home/hpdn/HPDN_Exercises/week_1/custom_complete_topo.py`
For example, you can start a complete topology of 7 switches as follows:

```
sudo mn --custom /home/hpdn/HPDN_Exercises/week_1/custom_complete_topo.py --topo complete,7
```

Any parameters you provide for a custom topology (in this case 7) are passed on to `__init__`.
Note: Ping will not work on both the complete graph, as well as the square lattice topology you will construct yourself. (You will study this further in exercise 5 (Sec 1.8.2))!

EXERCISE

Create a custom script that can construct square lattice topologies of arbitrary size w x w. In a square lattice topology, all switches are aligned on a square grid and connected to their (up to 4) neighbors. Only add hosts to the four corner switches. See Figure 1.2 for an example of a square lattice topology of size 3x3.

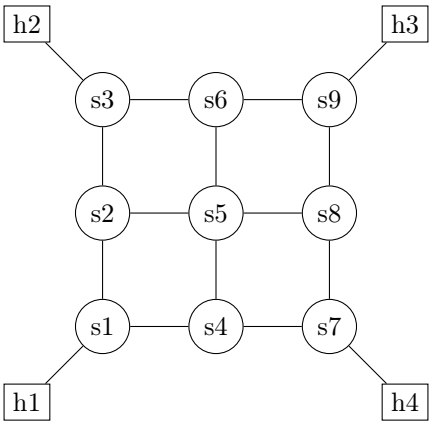


Figure 1.2: Square lattice graph ($w = 3$).

Solution

```
from mininet.topo import Topo

class SquareLattice(Topo):
    # Switches form a square lattice
    # One host at each corner

    # Note that we give a default value for w,
    # so mininet will not crash if someone creates this topology without specifying w
    def __init__(self, w = 3):
        Topo.__init__(self)

        switches = []
        for i in range(w):
            for j in range(w):
                switch_id = i * w + j + 1
                # create switch with id 's' + str(sid)
                switch = self.addSwitch('s' + str(switch_id))
```

```
                switches.append(switch)

                if j > 0:
                    # link current switch with switch below
                    self.addLink(switch, switches[-2])
                if i > 0:
                    # link current switch with switch to the left
                    self.addLink(switch, switches[-1 - w])

        # add the hosts
        host1 = self.addHost('h1')
        host2 = self.addHost('h2')
        host3 = self.addHost('h3')
        host4 = self.addHost('h4')

        # and link them to their switches
        self.addLink(switches[0], host1)
        self.addLink(switches[w-1], host2)
        self.addLink(switches[-1], host3)
        self.addLink(switches[-w], host4)

# create the mapping of custom topology names to topologies
topos = {'lattice': SquareLattice}
```

1.5 Introducing link properties

Mininet also has the capability to emulate network link parameters, such as bandwidth, delay, jitter, and loss. For example, if you want to set the bandwidth of all links in the network to 40 Mbps and their delay to 15 ms, you can run a command like the following. The default bandwidth unit is Mbps, and the default delay unit is μs:

```
sudo mn --link tc,bw=40,delay=15ms
```

It is possible to add these parameters in the custom Python files as well by specifying these options in the *addLink* function and using `TCLink` as shown below, and this allows us to set parameters of each link separately:

```
from mininet.link import TCLink
...
self.addLink(s1, s2, delay='5ms', bw=20, cls=TCLink)
```

or

```
from mininet.link import TCLink
...
self.addLink(s1, s2, delay=5000, bw=20, cls=TCLink)
```

Instead of importing `TCLink` and using `cls=TCLink`, you can also add the `--link tc` option when running mininet. `tc` stands for traffic control.

1.5.1 Exercise 2.1 - Add link properties

Create a simple loop-free topology using a topology script. Make sure there is a path from h1 to h2 in your topology. You can use a topology like the initial example in Section 1.4 where 2 hosts are connected via 2 switches (`custom_topo.py`). Modify your topology to set the delay of each link to a random value between 0 ms and 10 ms. In addition, set all link bandwidths to 10 Mbps.

Confirm your results using iperf and ping, for example:

```
h2 iperf -s &
h1 iperf -c h2
h1 ping h2
```

Keep in mind that the resulting bandwidth and delay values might not be entirely precise because Mininet itself can be a little imprecise and we are running it inside a VM, causing more loss of precision.

Solution

```
from mininet.topo import Topo
from mininet.link import TCLink
from random import random

class SimpleTCTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self._add_custom_link( leftHost, leftSwitch )
        self._add_custom_link( leftSwitch, rightSwitch )
        self._add_custom_link( rightSwitch, rightHost )

    def _add_custom_link(self, n1, n2):
        delay = random()*10
        self.addLink(n1, n2, delay=str(delay) + 'ms', bw=10, cls=TCLink)

topos = { 'tctopo': SimpleTCTopo }
```

Note that the delay is set to a new random value for each individual link.

When starting the topology, Mininet displays the bandwidth and delay of each link (in both directions) under "Adding links:". For example, `(10.00Mbit 9.77986975089ms delay) (10.00Mbit 9.77986975089ms delay) (h1, s1)`

If you start a ping between hosts `h1` and `h2`, the time should fluctuate around the sum of the link latencies between `h1` and `h2`. Similarly, the results of iperf should give a bandwidth of around 10 Mbps. (There can be some fluctuation here because you are running the simulation inside a VM.)

1.5.2 Exercise 2.2 - Congestion Control

In this exercise, we will use Mininet's capability to set link properties to experiment with congestion control. For this, you first need to create a bottleneck, i.e., a link that has lower capacity than other links. This bottleneck is where congestion will happen. If you use `custom_topo.py`, you can create a bottleneck between the two switches. Set the bandwidth of the link between the two switches to 10 Mbps. Set the bandwidth of the links between the hosts and the switches to 100 Mbps. Fix the delay on every link to 5 ms. Additionally, set the buffer size at the bottleneck link to 25 packets using the parameter `max_queue_size=25` in the addLink() function.

Start the network and set the congestion control algorithm on one host (in this case h1) to Reno using

```
h1 ip route change 10.0.0.0/8 dev h1-eth0 congctl reno
```

Run iperf to send a data stream from this host to another host. The bandwidth you see should be close to 10 Mbps, which is the bottleneck link speed.

Now change the buffer size to 5 packets and restart the network. As before, set the congestion control algorithm to Reno and run iperf. What can you observe?

Finally, change the congestion control algorithm to BBR using

```
h1 ip route change 10.0.0.0/8 dev h1-eth0 congctl bbr
```

Run iperf again. What happens? Can you explain why?

Solution

```
from mininet.topo import Topo
from mininet.link import TCLink
from random import random
```

```
class CongestionTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self._add_custom_link( leftHost, leftSwitch, 100 )
        self._add_custom_link( leftSwitch, rightSwitch, 10, 25 )
        self._add_custom_link( rightSwitch, rightHost, 100 )

    def _add_custom_link(self, n1, n2, bw, bufsize=None):
        delay = 5
        if bufsize:
            self.addLink(n1, n2, delay=str(delay) + 'ms',
                    bw=bw, max_queue_size=bufsize, cls=TCLink)
        else:
            self.addLink(n1, n2, delay=str(delay) + 'ms',
                    bw=bw, cls=TCLink)

topos = { 'cctopo': CongestionTopo }
```

When running iperf with Reno and a buffer size of 25 packets, iperf will report a bandwidth close to 10 Mbps (e.g., 9.5 Mbps). When you lower the buffer size to 5 packets, Reno can only achieve a lower bandwidth (e.g., 7.5 Mbps). With BBR, on the other hand, you should see a higher bandwidth even when the buffer is small (e.g., 9.3 Mbps).

This happens because upon packet loss, Reno backs off (Multiplicative Decrease) and then slowly raises its sending rate again (Additive Increase). If the buffer is too small, Reno backs off so often that its Additive Increase can never reach maximum throughput before there is another packet loss.

A well-known rule of thumb is that the ideal buffer size when using Reno is 1 Bandwidth-Delay Product (BDP). With 10 Mbps bandwidth and 30 ms delay (6 links with 5 ms delay each), one BDP is 37500 bytes, which is 25 packets (a typical IP packet is 1500 Bytes). When there is packet loss and Reno backs off, the buffer is still filled with 25 packets, which get forwarded while Reno sends at a lower rate. By the time the buffer is empty, Reno's Additive Increase has reached maximum throughput again. This effectively keeps up the throughput because packets are sent from the buffer even though Reno has reduced its sending rate. However, if the buffer is too small, it does not contain enough packets to keep up the effective throughput. BBR takes a fundamentally different approach to congestion control and therefore does not suffer from the same throughput reduction as Reno.

1.6 Automating Tasks

It is possible to automate certain tasks in Mininet, such as adding routes to the hosts, executing Python scripts, or tearing down links. To do so, simply create a file and put a single command on each line, just like in a shell script. These commands can then be executed in Mininet with `source file`, where `file` is the path to your automation script.

For example, the following script first adds multicast routes to hosts `h1`, `h2`, and `h3` and prints "routes setup". Next, it starts two multicast iperf servers on hosts `h1` and `h2` and connects them with a client on host `h3`. After `h3` has finished sending its multicast packets, the two servers are shut down.

```
h1 route add -net 224.0.0.0 netmask 240.0.0.0 dev h1-eth0
h2 route add -net 224.0.0.0 netmask 240.0.0.0 dev h2-eth0
h3 route add -net 224.0.0.0 netmask 240.0.0.0 dev h3-eth0
py "routes setup"
h1 iperf -s -B 224.0.0.14 -u -i 1 &
h2 iperf -s -B 224.0.0.14 -u -i 1 &
```

```
h3 iperf -c 224.0.0.14 -u -t 5
h1 kill %iperf
h1 kill %iperf
h2 kill %iperf
h2 kill %iperf
```

This script exists in `HPDN_Exercises/week_1/example_script`. You can create a single Mininet topology with 3 switches and run the script with:

```
sudo mn --topo single,3
```

```
mininet> source /home/hpdn/HPDN_Exercises/week_1/example_script
```

Important note: We put & after commands to start the process in the background. This way, the next command will execute immediately and would not have to wait until the earlier one is finished. In the case of iperf, we could alternatively use the `-D` option to run it as a daemon.

1.7 Custom Mininet commands

It is possible to create custom Mininet commands. For example, we can add the command "sleep" to Mininet by creating the following Python file and loading it with `--custom`:

```
from mininet.cli import CLI

from time import sleep

def custom_sleep(self, time):
    "custom sleep function"
    sleep(int(time))

CLI.do_sleep = custom_sleep
```

By specifying "CLI.do_*foo* = *bar*", you create the custom command *foo* that executes the function *bar*.

If we load this custom file we can use `sleep 10` to let the Mininet command line interface sleep for 10 seconds.

You can have multiple custom files. For example, if we have a topology file "custom_topo.py" and a commands file "custom_command.py", we can load both of them with:

```
--custom custom_topo.py,custom_command.py
```

1.7.1 Exercise 3. - Creating a Mininet script

Start Mininet with the linear topology with 4 switches (using `--topo=linear,4`) and the "custom_command.py" file created earlier, which contains the custom sleep command. Then, create an automation script that does the following:

- start a ping between h1 and h4
- sleep for 2 seconds
- tear down the link between switches s2 and s3 (`link s2 s3 down`)
- sleep for 60 seconds
- bring the link back up
- sleep for 2 seconds
- kill the ping process (`h1 kill %ping`)

Execute the script. What happens? What changes if you change the middle sleep command to 6 seconds instead of 60? Can you still deduce that the link was down from the ping messages?

Note that you only see the full output of ping **after** the process is killed.

Solution

```
h1 ping h4 &
sleep 2
link s2 s3 down
sleep 60
link s2 s3 up
sleep 2
h1 kill %ping
```

The output should look something like this:

```
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=51.9 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=1.01 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.208 ms
From 10.0.0.4 icmp_seq=40 Destination Host Unreachable
From 10.0.0.4 icmp_seq=41 Destination Host Unreachable
From 10.0.0.4 icmp_seq=42 Destination Host Unreachable
From 10.0.0.4 icmp_seq=43 Destination Host Unreachable
From 10.0.0.4 icmp_seq=44 Destination Host Unreachable
From 10.0.0.4 icmp_seq=45 Destination Host Unreachable
From 10.0.0.4 icmp_seq=46 Destination Host Unreachable
From 10.0.0.4 icmp_seq=47 Destination Host Unreachable
From 10.0.0.4 icmp_seq=48 Destination Host Unreachable
From 10.0.0.4 icmp_seq=49 Destination Host Unreachable
From 10.0.0.4 icmp_seq=50 Destination Host Unreachable
From 10.0.0.4 icmp_seq=51 Destination Host Unreachable
From 10.0.0.4 icmp_seq=52 Destination Host Unreachable
From 10.0.0.4 icmp_seq=53 Destination Host Unreachable
From 10.0.0.4 icmp_seq=54 Destination Host Unreachable
From 10.0.0.4 icmp_seq=55 Destination Host Unreachable
From 10.0.0.4 icmp_seq=56 Destination Host Unreachable
From 10.0.0.4 icmp_seq=57 Destination Host Unreachable
From 10.0.0.4 icmp_seq=58 Destination Host Unreachable
From 10.0.0.4 icmp_seq=59 Destination Host Unreachable
From 10.0.0.4 icmp_seq=60 Destination Host Unreachable
64 bytes from 10.0.0.4: icmp_seq=61 ttl=64 time=2042 ms
64 bytes from 10.0.0.4: icmp_seq=62 ttl=64 time=1042 ms
64 bytes from 10.0.0.4: icmp_seq=63 ttl=64 time=43.3 ms
64 bytes from 10.0.0.4: icmp_seq=64 ttl=64 time=0.995 ms
```

We can clearly see the link is down, as we get multiple `Destination Host Unreachable messages`. Now, if we replace the 60 seconds sleep with 6 seconds, the output will be similar to:

```
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=15.3 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=11.9 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.812 ms
64 bytes from 10.0.0.4: icmp_seq=10 ttl=64 time=16.2 ms
64 bytes from 10.0.0.4: icmp_seq=11 ttl=64 time=0.092 ms
```

We do not get any explicit messages that the host is unreachable anymore. It takes some time for these Destination Host Unreachable messages to appear because they only happen after the entry in the ARP cache times out, which is after 15–45 seconds. Nevertheless, even without Destination Host Unreachable messages, we can recognize that 6 ping requests did not receive a reply (corresponding to the 6 seconds the link was down) by *comparing the icmp_seq numbers.*

1.8 Mininet packet capture using Wireshark

Wireshark is a network protocol analyzer used to capture and store network packets. Monitoring packets can be very helpful in later exercises to help you debug your application. In this section, we will use Wireshark to monitor the traffic between switches.

1.8.1 Exercise 4. - Monitoring traffic

1. Start Mininet with the linear topology with 2 switches.

2. Start Wireshark as a background process on host 1 (`h1 wireshark &`). Mininet always starts processes as superusers. This causes the initial warning you will see when starting Wireshark, and you can safely ignore this warning.

3. On the left you can select any of h1's interfaces to monitor. We are only interested in $h1-eth0$, so select this one and press start. You should not see any packets yet, as there is currently no traffic in your emulated network.

4. Generate some traffic by starting a ping from host 1 to host 2. You should be able to spot 2 types of traffic: ARP packets (so h1 and h2 can learn each others' addresses) and ICMP packets.

5. Filter on the ARP protocol by writing `arp` in the filter at the top-left.

6. You can filter packets on a wide variety of properties. For example, we can filter out all packets with IP destination address 10.0.0.2 by typing in `ip.dst!=10.0.0.2`. You can test this along with other conditions like the following:

```
ip.dst!=10.0.0.2 and icmp
```

The interfaces of switches can be monitored in the same way. For example, you can start Wireshark on switch s1 with `s1 wireshark &`. The only difference to hosts is that Mininet switches do not run on their own separate network namespace, so the Wireshark client we started on switch s1 can also access the interfaces of all other switches.

You can capture the traffic of multiple interfaces by either selecting the interfaces and pressing start, or by starting multiple Wireshark processes at the same host/switch and starting multiple captures.

Hint: When capturing multiple interfaces at the same time, you can add an additional column to indicate the interface the traffic was captured on by following these instructions: `https://osqa-ask.wireshark.org/questions/30636/traces-from-multiple-interface`

1.8.2 Exercise 5. - Diagnosing Network Issues

Start a Mininet instance with your square lattice topology (change the default width and set $w = 2$). Try to ping host h4 from host h2. Use Wireshark to find out what goes wrong and explain why this problem occurs.

Hint 1: Learning Switches. Switches start with an empty forwarding table and incrementally build it through a learning process based on the processed packets. This process works as follows: the switch maintains a table of MAC addresses that have appeared as source addresses in packets and the interface the switch has received these packets on. Thus, the switch learns and knows how to reach these addresses if one of them later shows up as a destination address. If a switch does not have an entry for a particular destination address, it defaults to flooding, i.e., it forwards the packet out on every port except the one on which it received the packet.

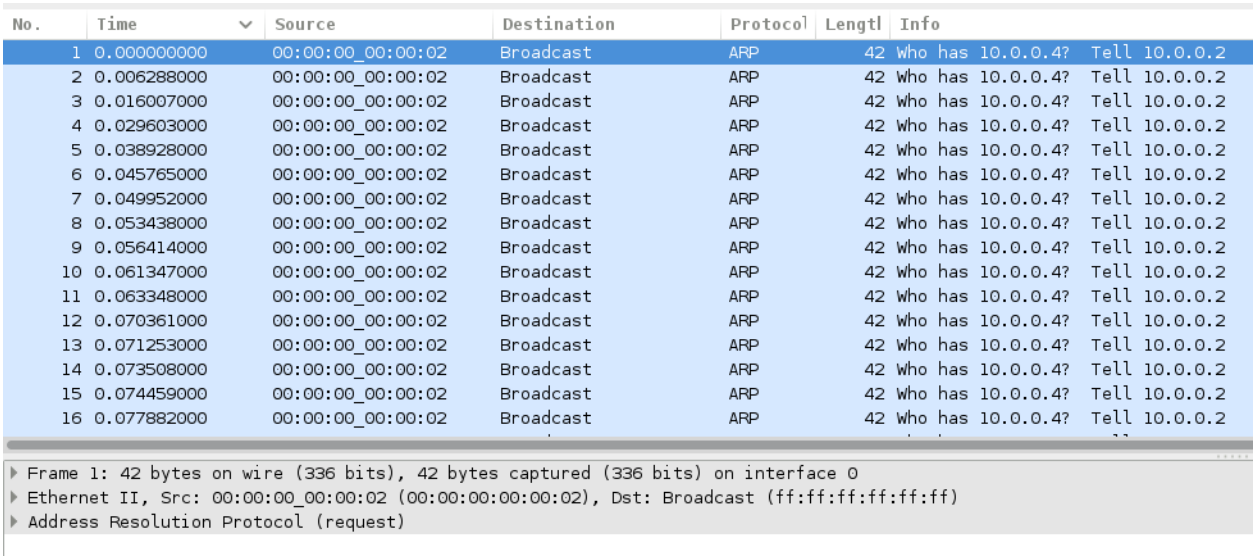
Hint 2: In previous exercises, we saw some network topologies where ping worked and others where it fails. What is a structural difference between the networks where ping succeeds and the ones where ping fails?

Hint 3: In Wireshark, order packets by timestamp. If your computer slows down too much during this exercise, restart the Mininet network.

Can you think of a method to fix the problem?

Solution

The ping does not seem to work, as it only outputs `Destination Host Unreachable` and gives a 100% packet loss rate. To figure out why this is happening, let's first capture the traffic on host `h2`, to see if it actually sends the ping request packets:

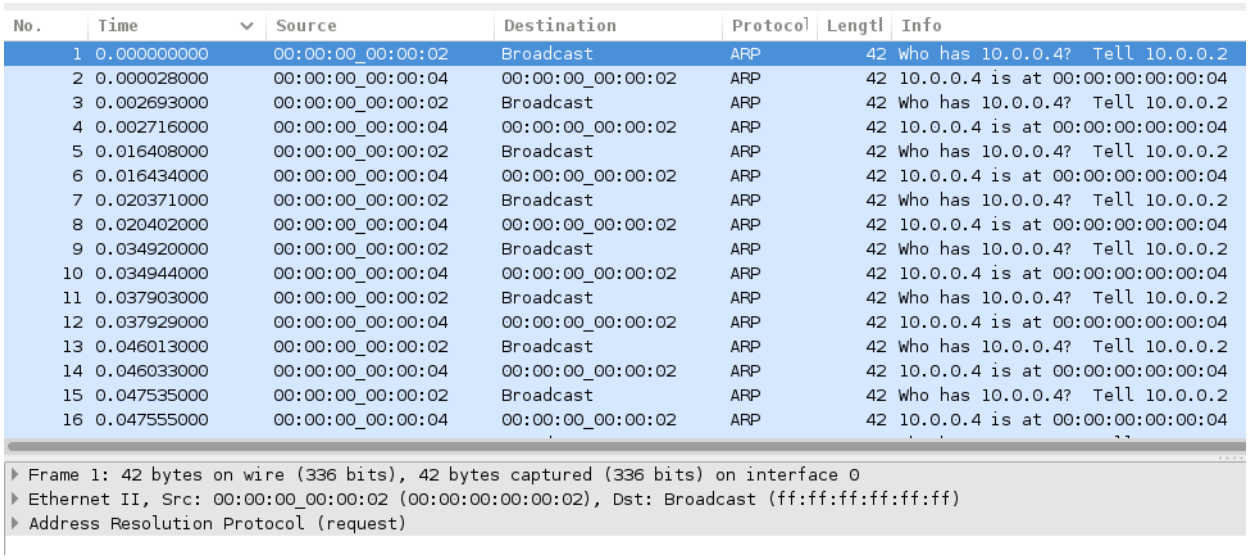


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
2	0.006288000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
3	0.016007000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
4	0.029603000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
5	0.038928000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
6	0.045765000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
7	0.049952000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
8	0.053438000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
9	0.056414000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
10	0.061347000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
11	0.063348000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
12	0.070361000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
13	0.071253000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
14	0.073508000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
15	0.074459000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
16	0.077882000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2

```
▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)
```

As we can see, `h2` is not sending out any ping requests to the network, as it does not have the Ethernet address of `h4` (10.0.0.4). To get this address `h2` continuously sends out ARP packets to the network, but it does not receive any replies. (As we will later learn, what we are seeing here are not only packets sent out by `h2`, but also some of its own packets that got sent back by the switch to `h2`. Unfortunately, we cannot separate these in Wireshark).

The next obvious step is to capture the traffic on `h4` as well, to see if it actually receives and replies to these ARP packets:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
2	0.000028000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
3	0.002693000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
4	0.002716000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
5	0.016408000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
6	0.016434000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
7	0.020371000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
8	0.020402000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
9	0.034920000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
10	0.034944000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
11	0.037903000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
12	0.037929000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
13	0.046013000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
14	0.046033000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
15	0.047535000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
16	0.047555000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04

```
▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)
```

Interestingly, `h4` does receive and reply to the ARP packets, but for some reason, the network is not sending these replies to `h2`.

To discover why the replies are not being sent to `h2`, we will have to capture the traffic on a switch itself. We capture the traffic on all interfaces of switch `s2` (the switch connected to `h2`). As we are capturing the traffic on multiple interfaces, it will be useful to sort the traffic on timestamp, as by default the packets are sorted by interface first and time of arrival second (to increase clarity, we added an interface id column):

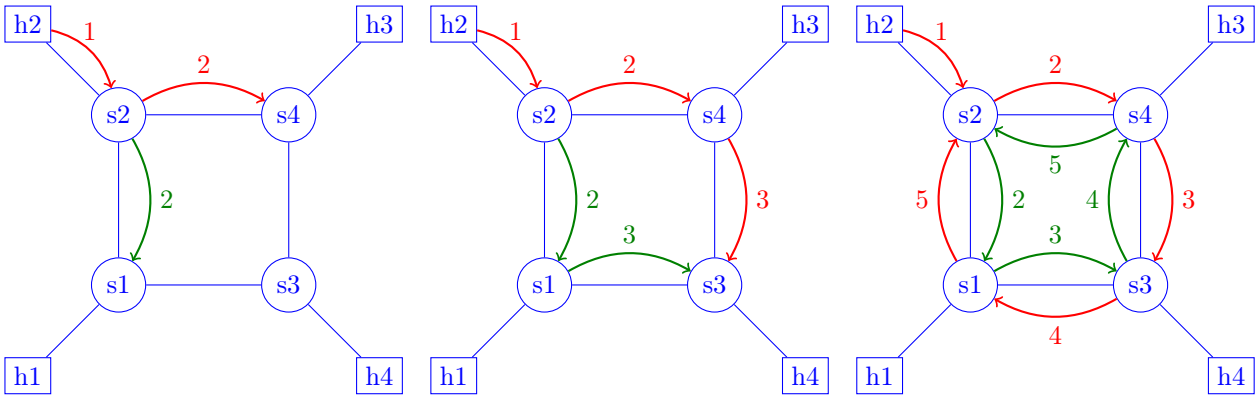
No.	Time	Source	Destination	Protocol	Lengtl	Interface id	Info
1	0.000000000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
22	0.000779000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
44	0.000785000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
23	0.003080000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
45	0.004790000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
2	0.004795000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
46	0.005251000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
24	0.006680000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
3	0.006686000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
25	0.011601000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
47	0.012347000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
48	0.012782000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
4	0.012787000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
26	0.014088000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
5	0.014092000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
27	0.014480000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2

```
▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)
```

`s2` does not receive the ARP reply either (**NOTE:** This differs per machine. On some machines, `s2` will receive the reply and on others it will not. However, the overall problem remains the same.), but this capture does give us all the information needed to figure out the reason ping does not work on this network. As you can see, `s2` does not only receive the ARP request on interface `0` (its connection to `h2`), but on all its interfaces. That is, the packets are looping in the network. Loops such as in our lattice topology are problematic for communication networks.

By default, Mininet switches act as learning switches. That is, they output packets addressed to Ethernet address A to the last port they received a packet from A on (or just output the packet to all ports if they have not received a packet from A yet). Now, because multiple packets are looping around in our network, this output port will, with some exceptions, not be the correct output port. In fact, in our case, a switch is dropping all ARP replies, because it receives them on the same port it should send them to.

In our network, every ARP request sent to switch `s2` results in two packets looping through the network in opposite directions, as broadcast packets are output to all ports (except the one they are received on). This is illustrated below:



If we were to use a square lattice with a larger width, a single broadcast packet would be duplicated infinitely. This is called a broadcast storm.

There are multiple protocols to solve this problem. One prominent example is the Spanning Tree Protocol (STP), which makes switches flood/broadcast packets over a spanning tree instead of to all output ports, by disabling all ports that are not part of the tree.

In an OpenFlow network, loops in the network are much less of a problem. The controller has a central view of the network, so it should not have to flood packets to all output ports of switches. Nevertheless, when handling broadcast packets, care should be taken to not loop these packets through the network. To handle these kinds of packets, a similar approach to the STP can be taken by installing a spanning tree in the network and redirecting all broadcast packets to this spanning tree. Alternatively, the controller can send broadcast packets to all hosts itself. However, in many cases, this is not recommended, as the controller will potentially be flooded and overburdened with broadcast packets.

1.9 Useful commands

`dump`: Dump information about all nodes
`help`: Display all options
`device_name command`: If the first typed string is a host, switch or controller name, the command is executed on that node, e.g. to display the IP address of a host h1 type `h1 ifconfig`

`link s1 s2 down/up`: Bring links down and up
`h1 ping h2`: Test connectivity between two hosts
`pingall`: Test connectivity by having all nodes ping each other
`h1 arp`: Display the ARP table of host h1
`h1 route`: Display the routing table of host h1
`h1 python -m SimpleHTTPServer 80 &`: Run a simple web server on host h1
`h1 kill \%python`: Shut the started web server down
`h2 wget -O - h1`: Send a request from node h2 to the web server running on node h1

You have now finished the HPDN Mininet exercises. For more information on Mininet, you can look through the official walkthrough: `http://mininet.org/walkthrough/`

Chapter 2

Lecture 2: Software-Defined Networking (SDN)

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/Q2IKWc8khoa5N7B` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_2` in which you can do the exercises.

Solution

The solution files are available here: `https://surfdrive.surf.nl/files/index.php/s/2P2NTxTabMR6PzL`

2.1 Important Notice

In our simulation of the SDNs, we utilize Ryu, a component-based software-defined networking framework. Ryu is the software that runs on the SDN controller. To communicate with switches, the controller uses the OpenFlow protocol.

If Ryu gives the following error (or similar) while running:

```
Traceback (most recent call last):
  File "./bin/ryu-manager", line 19, in <module>
    main()
  File "/home/hpdn/ryu/ryu/cmd/manager.py", line 98, in main
    app_mgr.load_apps(app_lists)
  File "/home/hpdn/ryu/ryu/base/app_manager.py", line 415, in load_apps
    cls = self.load_app(app_cls_name)
  File "/home/hpdn/ryu/ryu/base/app_manager.py", line 392, in load_app
    mod = utils.import_module(name)
  File "/home/hpdn/ryu/ryu/utils.py", line 104, in import_module
    return importlib.import_module(modname)
  File "/usr/lib/python2.7/importlib/__init__.py", line 37, in import_module
    __import__(name)
ImportError: Import by filename is not supported.
```

It can mean one of two things:

- You gave the wrong path or file name
- Your Python code contains a syntax error

To differentiate between these two problems (and find any syntax errors), you can simply run your script directly via the Python compiler to debug syntax errors:

```
python your_app.py
```

Mininet is able to connect to remote controllers. After starting Mininet, each OpenFlow switch in the network connects to the controller, which can run in the same VM or outside the VM. By default, Mininet runs a local reference OpenFlow controller to control its switches. In order to connect your Mininet environment to a specific controller, you should start Mininet with the `--controller=remote` option:

```
--controller=remote,ip=[controller IP],port=[controller listening port]
```

2.2 Exercise 1. - Connecting the switches to a custom controller

We will use Mininet, Ryu[1], and OpenFlow (OF) to emulate SDNs (Ryu is explained in more detail in section 2.4). Just like Mininet, Ryu is pre-installed on the provided VM. Follow the steps below to complete this exercise:

1. Open a new terminal and change the current directory to the directory where Ryu is installed:

   ```
   cd ~/ryu
   ```

2. Run a simple controller application called SimpleSwitch (for OpenFlow version 1.3) using this command:

   ```
   PYTHONPATH=. ./bin/ryu-manager ryu/app/simple_switch_13.py
   ```

   This will start the Ryu controller and instruct it to run the `simple_switch_13.py` app. By default, the Ryu controller listens for switches on port 6633. You can change this with the `--ofp-tcp-listen-port` option.

3. Open another terminal and start Mininet with a 5 node to 1 switch topology, adding the appropriate parameters to connect to the remote switch:

   ```
   sudo mn --topo single,5 --mac --controller=remote --switch ovs,protocols=OpenFlow13
   ```

   This command tells Mininet to look for the controller on the default address (127.0.0.1) and port (6633). We set the switch OF protocol (to OF 1.3) with `--switch ovs,protocols=OpenFlow13`.

   Note: Mininet and the controller are separate programs, so you will always need to start both Mininet and the controller itself.

4. Test the connectivity using the `pingall` command and check the output from the terminal that is running the controller app. You should see packets arriving at the controller. Try using the same command again: why does the controller receive no packets the second time?

   Note: you will see that the controller regularly receives messages related to IPv6 SLAAC and Neighbor Discovery. You can recognize these messages by destination MAC addresses that start with 33:33:* (MAC addresses used for IPv6 multicast). You can ignore these messages.

   Solution

   The first time you run the pingall command, the following messages should be printed by the controller:

   ```
   packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
   packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
   packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
   packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
   packet in 1 00:00:00:00:00:03 00:00:00:00:00:01 3
   packet in 1 00:00:00:00:00:01 00:00:00:00:00:03 1
   packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
   packet in 1 00:00:00:00:00:04 00:00:00:00:00:01 4
   packet in 1 00:00:00:00:00:01 00:00:00:00:00:04 1
   packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
   packet in 1 00:00:00:00:00:05 00:00:00:00:00:01 5
   .
   .
   .
   ```

   Let's analyze one set of messages:

   ```
   (1) packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
   (2) packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
   (3) packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
   ```

   (1) is the ARP request that is being broadcasted (as one can see from the broadcast MAC address). (2) is the ARP reply from host 2 to host 1; the controller registers a flow entry here (from 2 to 1). (3) is the ICMP echo request (for the ping) from host 1 to host 2; again, the controller registers a flow entry here (from 1 to 2). For the ICMP echo reply thereafter, there is already a matching flow entry (the one registered in step (2)). So, the reply is transferred directly without issuing a packet-in.

   The second time, you should see no new messages at all, as the controller will not receive any new packets.

[1] https://ryu-sdn.org/

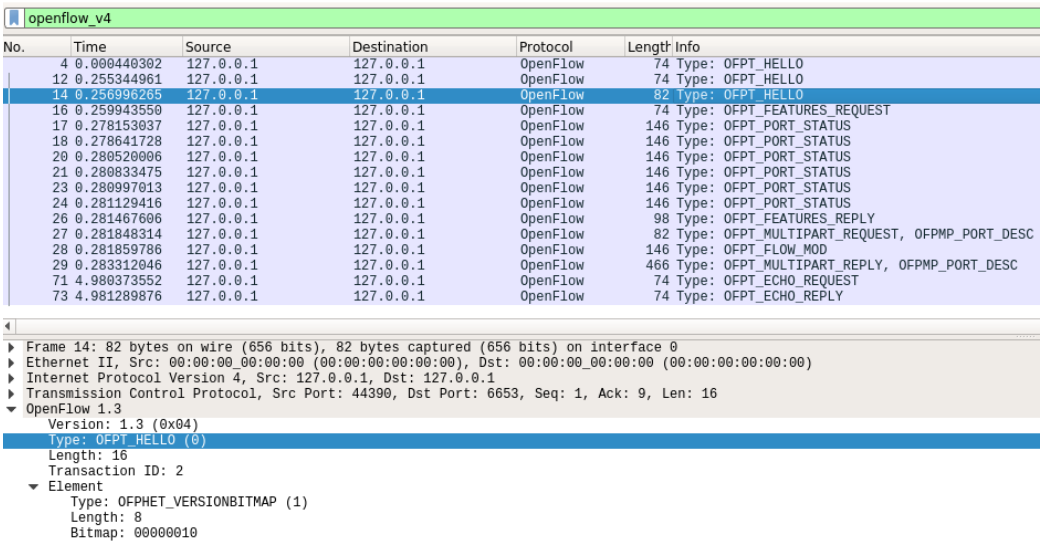
2.3 Exercise 2. - Capturing traffic between switches and controller

As mentioned previously, monitoring packets can be very helpful when debugging your controller application. For example, to test if a certain switch receives specific packets, or to inspect the OF messages between the controller and a switch. In this exercise, you will learn how to monitor the traffic between the controller and the switches.

1. Start the controller as explained in Section 2.2.
2. Start Wireshark in a second terminal (`sudo wireshark`). Note: Running this command as sudo will cause an initial warning, but you can safely ignore this warning.
3. In Wireshark, you can select any of the VM interfaces to monitor. We are only interested in the loopback interface (lo), because the controller is running on the same VM and is using the loopback interface. Select this interface and press start. You should not see any OpenFlow packets yet, as Mininet is not started yet.
4. Start Mininet with the single topology again, using the `--controller remote` option to connect the switches to the controller.
5. After starting Mininet, each OpenFlow switch in the network will connect to the controller. Observe the packets captured on the loopback adapter and describe the protocol handshake that occurs. In order to filter the OpenFlow 1.3 traffic in Wireshark, you can use `openflow_v4` as the filter. Note that the source and destination IP will always be `127.0.0.1`, as they are captured on the loopback interface. To differentiate between switches and the controller, you can look at the TCP source and destination ports instead.

   Note: The first connection you will capture is Mininet connecting to the controller and immediately closing the connection again after receiving an `OFPT_HELLO` message. You can ignore any packets related to this connection.

Solution



OpenFlow handshake:

(a) First, both the controller and the switch send each other an `OFPT_HELLO` message with the version set to the highest OF version they support. Optionally, they can also include a bitmap of supported OF versions. The version they choose to communicate with is the highest OF version they both support (in this case OF 1.3).

(b) Next, the controller sends an `OFPT_FEATURES_REQUEST` message to the switch, to which the switch replies with an `OFPT_FEATURES_REPLY`.

In this case, the switch also sent the controller an `OFPT_PORT_STATUS` message for each of its ports. This is not part of the handshake and is not actually required.

After this handshake, the Ryu controller sent an `OFPMP_PORT_DESC` message to get a description of all ports of the switch (port number, status, etc.) and installs the table-miss flow entry.

6. Open another terminal and use the `ovs-ofctl` command to print the flow entries of the switch:

   ```
   sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s1
   ```

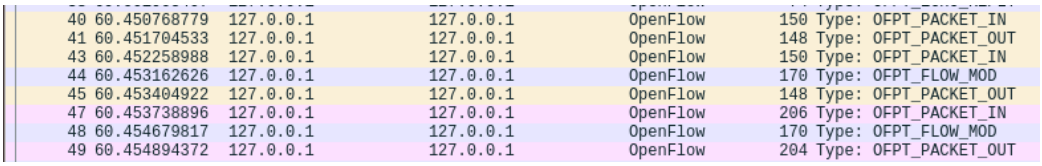
   Are there any entries present?

Solution

The switch has a single flow entry:
`cookie=0x0, duration=1535.775s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER:65535`
This is the table-miss flow entry and instructs the switch to send all packets that do not match any (other) flow entries to the controller.

7. Run the `h1 ping h2` command to generate traffic between hosts. When the first switch (s1) receives packets from h1, it is going to send a PacketIn message to the controller. Watch the packets captured on the loopback adapter and describe the actions taken by the controller.

Solution



The controller first receives an ARP request packet (embedded in the PacketIn message), which it simply floods to all ports of the switch (PacketOut). Next, it receives an ARP reply packet with destination `h1` (PacketIn). As a response, the controller installs a flow entry that matches packets with source `h2` and destination `h1` and outputs these packets to port `1` (FlowMod). In addition, the controller instructs the switch to output the ARP reply packet to this port as well (PacketOut). The final packet sent to the controller is the first ICMP echo request message transmitted by `h1` (PacketIn). As before, the controller installs a flow entry matching packets with source `h1` and destination `h2` and outputting these packets to port `2` (FlowMod), and instructs the switch to output the ICMP echo request packet to this port as well (PacketOut).

The switch does not transmit any other packets (between `h1` and `h2`) to the controller, as they will now match these flow entries.

8. Filter out the OpenFlow Flow Add messages. This can be done with the `openflow_v4.type == 14` filter. Analyze the captured packets.

9. Check the flow tables on the switches again. What changed compared to step 6.?

Solution

Two additional flow entries have been installed:

```
cookie=0x0, duration=1178.284s, table=0, n_packets=10, n_bytes=924, priority=1,in_port="s1-eth2",
dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
cookie=0x0, duration=1178.283s, table=0, n_packets=9, n_bytes=826, priority=1,in_port="s1-eth1",
dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
cookie=0x0, duration=3603.432s, table=0, n_packets=3, n_bytes=182, priority=0 actions=CONTROLLER:65535
```

These are exactly the flow entries we described earlier.

2.4 Exercise 3. - Creating an SDN Firewall

Ryu is a component based SDN framework and can be used to implement your own controller for an OpenFlow network using Python. It contains many useful components, such as topology discovery, packet generation, and a web GUI providing insight into the network topology.

In this exercise, you will implement an SDN controller-based firewall, and we will guide you toward creating your first own Ryu application step by step. The final app will be able to route IPv4 unicast packets through the network and will act as a basic firewall. The firewall will block all traffic except TCP traffic sent to port 80 or port 443.

The Firewall exercise contains two parts:

1. Create an app that connects to switches and acts as a basic firewall.

2. Add routing functionality.

2.4.1 Firewall

To create a Ryu app, you only need to extend the RyuApp class. The archive you downloaded contains a basic template for this exercise in `week_2/SDNBasicFirewallTemplate.py`. We recommend you to make a copy of the Python file an save it as `basicFirewallApp.py`. A Ryu application is built on an event based programming model. You can read more about the Ryu Application API using the following link: `https://ryu.readthedocs.io/en/latest/ryu_app_api.html`.

Connecting to a Switch

The small template is already a functioning Ryu app. You can run the app with

```
cd ~/ryu
PYTHONPATH=. ./bin/ryu-manager [path_to_app]/basicFirewallApp.py
```

Explanations of parts the template code

The provided basicFirewallApp class contains one class member and three different functions. Below we provide a short discussion about the things within the template code you might not have prior knowledge about.

1. *OFP_VERSIONS* should be a list of all the OpenFlow protocols the app supports. In this case the app only supports OF 1.3.

2. The *set_ev_cls* decorator is used to tell Ryu to execute a function during certain events. The first argument (in this case *ofp_event.EventOFPSwitchFeatures*) is the event during which the function should be called. Using the second argument you can limit during which switch negotiation phases the function can be called. The *CONFIG_DISPATCHER* phase is after the OF protocol version has been negotiated and a feature request message has been sent, but before the switch has responded with a list of features.

3. `@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)` tells Ryu to execute the decorated function after the controller has received the features reply message from a switch. This is the perfect moment to add a table-miss entry to the switch. Our *switch_features_handler* function simply instructs the switch to send all packets without matching flow entry to the controller. Note that the app does not yet handle these packets.

Note: If you have trouble understanding the template code, more information on Ryu can be found under the **Ryu Documentation**. Before starting the controller, first start a Wireshark capture on the loopback interface and filter on OF packets, like in the previous exercise (Section 2.3). Next, start both the controller and a simple Mininet network with a remote controller with the following command. You should now see your app connecting to the switches.

```
sudo mn --mac --controller=remote --switch ovs,protocols=OpenFlow13
```

Note: As before, the controller will register messages related to IPv6 SLAAC and Neighbor Discovery, which you can recognize by the destination MAC address being `33:33:*`. You can ignore these messages.

Questions

1. Try starting a ping from h1 to h2, what happens?

2. Where are all the packets reaching the network being sent to before reaching the destination?

Solution

1. `h1` can not reach `h2`. All the ARP request messages are sent to the controller that only logs them. You should see these log messages in the output of the controller. As before, you can recognize these messages by their destination MAC address being `ff:ff:ff:ff:ff:ff`, the broadcast MAC address.

2. The SDN controller / The RyuApp controller.

Setting Up the Basic Firewall

Part 1: Switch features handler

Currently, all traffic reaching the network is sent directly to the controller. However, we want to route valid TCP packets through the network. We do not want the switches to send the controller other types of traffic. Instead, they should simply drop the packets.

We can instruct switches to drop unmatched packets by installing an empty action list instead of the current action, i.e.:

```
actions = []
```

Unfortunately, setting actions to an empty list also drops all HTTP(S) packets. Instead, we want the switches to send these types of packets to the controller so that we can install flow entries to route this type of traffic through the network. In what follows, we explain the steps for adding the port 80 default flow entry.

1. Install two new flow entries that match both on the protocol (TCP) and the destination port.
2. To use the *tcp_dst* match field, OF first, requires us to match on TCP packets (`ip_proto=0x06`), which in turn requires us to match IPv4 (or IPv6) packets with the Ethernet type match field (*eth_type*).
3. Use the *tcp_dst* match field to match on the TCP destination port. The addition to our app will send all (otherwise unmatched) TCP packets sent to port 443 or 80 to the controller.

Note that we deliberately set the priority of the new entries higher than the table-miss entry, as otherwise the table-miss entry would be prioritized and all packets are dropped.

Now your code should look like the following:

```
#Switch connected
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    dp = ev.msg.datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser

    #Add table-miss flow entry
    match = parser.OFPMatch()
    actions = []
    instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
    cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
    dp.send_msg(cmd)

    #Add port 80 default flow entry
    match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=80)
    actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
    instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
    cmd = parser.OFPFlowMod(datapath=dp, priority=1, match=match, instructions=instr)
    dp.send_msg(cmd)

    #Add port 443 default flow entry
    #Fill in this part yourself
```

In the provided code, the functionality for adding the port 80 default flow entry is included but commented. Uncomment it, and the switches should send TCP packets sent to port 80 to the controller. Moreover, remember to change the action for the default table-miss flow entry to an empty list such that switches drop unmatched packets.

Exercise: Fill in the last part of the function (installing the default flow entry for packets sent to port 443) yourself.

Solution

```
#Switch connected
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    dp = ev.msg.datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser
```

```
        #Add table-miss flow entry
        match = parser.OFPMatch()
        actions = []
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)

        #Add port 80 default flow entry
        match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=80)
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=1, match=match, instructions=instr)
        dp.send_msg(cmd)

        #Add port 443 default flow entry
        match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=443)
        cmd = parser.OFPFlowMod(datapath=dp, priority=1, match=match, instructions=instr)
        dp.send_msg(cmd)
```

Part 2: Packet handler

To process these packets in the controller, we need to add a custom function to handle the packet-in event. If we decorate a function with `@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)`, it will be executed if the controller receives a packet from a switch (as long as the negotiation has completed).

For now, the code we provide will simply log and ignore all incoming packets:

```
    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg

        pkt = packet.Packet(msg.data)

        self.logger.info('Received packet:')
        self.logger.info(str(pkt))
```

Packet (`http://ryu.readthedocs.io/en/latest/library_packet.html`) is a Ryu library that can be used to parse and create packets.

If you have not stopped running yet, exit Mininet and stop the simple app you started earlier (`ctrl-c`). Start the new basicFirewallApp. Next, start Mininet again with the following command:

```
sudo mn --mac --arp --controller=remote --switch ovs,protocols=OpenFlow13
```

The addition of `--arp` flag will make Mininet pre-populate the ARP entries of each host. This is necessary because we want to immediately drop all ARP packets sent to the network.

Testing the firewall

1. First try to ping h2 from h1. You should not see any packets at the controller because ping packets are ICMP, which the switch drops. Note that you also no longer see the IPv6 Neighbor Discovery packets because their protocol is ICMPv6.

2. Test if TCP packets sent to port 443 and 80 are dropped directly (this should not be the case; they should arrive at the controller). You can test with the following command:

   ```
   h1 nc h2 443 -p 443
   ```

3. Note that since we haven't added routing functionality yet, the message itself does not reach h2. You can verify this by looking at the controller log.

Exercise

Check the entries the app installed on switch s1 by executing the following command in a new terminal:

```
sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s1
```

Solution

The output should be similar to the following:

```
cookie=0x0, duration=25.678s, table=0, n_packets=0, n_bytes=0, priority=1,tcp,tp_dst=80 actions=CONTROLLER:65535
cookie=0x0, duration=25.678s, table=0, n_packets=0, n_bytes=0, priority=1,tcp,tp_dst=443 actions=CONTROLLER:65535
cookie=0x0, duration=25.678s, table=0, n_packets=0, n_bytes=0, priority=0 actions=drop
```

2.4.2 Routing

We will now add some routing functionality to the controller.

Open `/home/hpdn/HPDN_Exercises/week_2/SDNBasicFirewallTemplate2.py` and save a copy as `FirewallApp.py`. Now, copy the `switch_features_handler` function from `basicFirewallApp.py` (from the previous exercise) to this new file such that the app drops all packets except for TCP packets sent to port 80 or 443.

Restarting Controller

By now, we have had to restart Mininet and the controller multiple times. When we start installing routes in the network, it would be nice if we did not have to restart the network every time we need to restart the controller. To make sure the controller starts with a clean slate every time we restart our app, we need to remove all flow and group entries from the switches during the negotiation phase.

We could add this functionality to the *switch_features_handler* function. However, the topology discovery module of Ryu (which we will start using next) adds an important flow entry to the switch after this function has already been called. As we do not want to remove this entry, we need to remove all existing entries earlier. We do so by using the *(ofp_event.EventOFPStateChange, CONFIG_DISPATCHER)* event, which is triggered just before *(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)*.

The following function in the template file removes any existing flow and group entries:

```
#This function gets triggered before
#the topology flow detection entries are added
#But late enough to be able to remove flows
@set_ev_cls(ofp_event.EventOFPStateChange, CONFIG_DISPATCHER)
def state_change_handler(self, ev):
    dp = ev.datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser

    #Delete any possible currently existing flow entry.
    del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL,
    out_port=ofp.OFPP_ANY, out_group=ofp.OFPG_ANY,
    command=ofp.OFPFC_DELETE)
    dp.send_msg(del_flows)

    #Delete any possible currently exising groups
    del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
    group_id=ofp.OFPG_ALL)
    dp.send_msg(del_groups)

    #Ensure deletion is finished before additional flows are added
    barrier_req = parser.OFPBarrierRequest(dp)
    dp.send_msg(barrier_req)
```

Sending a barrier request ensures that all previous flow modifications sent to the switch are executed before those you send after sending the barrier request.

Adding Topology Detection

To route packets through the network, the controller first needs to know the network topology. Fortunately, Ryu contains a built-in topology detection app. You can start this app by adding `--observe-links` when starting the controller:

```
PYTHONPATH=. ./bin/ryu-manager ~/path_to_app/FirewallApp.py --observe-links
```

The topology app will generate certain events, which, just as before, we can catch by adding decorators to our functions:

```
#Topology Events
@set_ev_cls(event.EventSwitchEnter)
def switchEnter(self,ev):

@set_ev_cls(event.EventSwitchLeave)
def switchLeave(self,ev):

@set_ev_cls(event.EventLinkAdd)
def linkAdd(self,ev):

@set_ev_cls(event.EventLinkDelete)
def linkDelete(self,ev):

@set_ev_cls(event.EventHostAdd)
def hostFound(self,ev):
```

These events can be imported by:

```
from ryu.topology import event, switches
```

Note that there is no host leave or host delete event, as the Ryu topology app can not detect the difference between a host leaving the network or simply not sending any packets for a while.

Our app will need to store all relevant topology information itself. To do so, we will use NetworkX (`https://networkx.github.io/`), a widely used Python graph library. A brief introduction on NetworkX is available at `https://networkx.github.io/documentation/stable/reference/introduction.html#networkx-basics`.

The following line in the constructor (`__init__(self, *args, **kwargs)`) creates a directional graph we can populate with all relevant topology information:

```
self.network = nx.DiGraph()
```

We can add switches, links and hosts to this graph as follows. Note that in this exercise we assume there will be no failures in the network and no devices will be removed from the network, so we do not implement the linkDelete and switchLeave functions.

Exercise: You need to add an edge between the source and destination to self.network using the ports. Finish the rest of `linkAdd` yourself:

```
#Topology Events
@set_ev_cls(event.EventSwitchEnter)
def switchEnter(self,ev):
    switch = ev.switch
    sid = switch.dp.id

    self.network.add_node(sid, switch = switch, flows= {}, host = False)

    self.logger.info('Added switch ' + str(sid))

@set_ev_cls(event.EventSwitchLeave)
def switchLeave(self,ev):
    switch = ev.switch
    sid = switch.dp.id

    self.logger.info('Received switch leave event: ' + str(sid))

@set_ev_cls(event.EventLinkAdd)
def linkAdd(self,ev):
    link = ev.link
    src = link.src.dpid
    dst = link.dst.dpid

    src_port = link.src.port_no
    dst_port = link.dst.port_no

    #Try to fill in the rest of this function yourself
```

```
        self.logger.info('Added link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventLinkDelete)
    def linkDelete(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        self.logger.info('Received link delete event: ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventHostAdd)
    def hostFound(self,ev):
        host = ev.host
        sid = host.port.dpid
        port = host.port.port_no
        mac = host.mac

        self.network.add_node(mac, host = True)
        self.network.add_edge(mac, sid, src_port = -1, dst_port = port)
        self.network.add_edge(sid, mac, src_port = port, dst_port = -1)

        self.logger.info('Added host ' + mac + ' at switch ' + str(sid))
```

Solution

```
    @set_ev_cls(event.EventLinkAdd)
    def linkAdd(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        src_port = link.src.port_no
        dst_port = link.dst.port_no

        self.network.add_edge(src, dst, src_port = src_port, dst_port = dst_port)

        self.logger.info('Added link from ' + str(src) + ' to ' + str(dst))
```

Test the app on any Mininet topology you want. You should see log messages of switches and links being added to the network. In addition, you will see a lot of LLDP packets arriving at the controller. LLDP stands for `Link-Layer Discovery Protocol`, and LLDP packets are used by the topology app to detect the links in the network.

You might notice that none of the hosts in your topology are detected. The topology app can only detect hosts when one of their packets is sent to the controller. Try generating some TCP traffic with netcat (nc) and you should see the hosts being detected by your app.

WARNING: Often your app will receive packets from hosts before the topology app will. This will cause the packet_in_handler to be called before the hostFound function. If your app depends on this ordering, you should implement the host detection yourself, without using the EventHostAdd event.

Routing Packets

We are finally ready to start routing TCP packets through the network. To do so, we need to add multiple flow entries to the network. It helps to create a single function allowing us to easily add simple flow entries to the network. By now, you should be able to write most of this function by yourself. You can find all possible match fields at `http://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#flow-match-structure` and all possible actions at `http://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#action-structures`.

Exercise: Implement the `_add_flow_entry` function in the template.

```
    def _add_flow_entry(self, sid, dst, port):
        """Adds flow entries on switch sid,
        outputting all (allowed) traffic with destination address dst to port.
```

```
        Arguments:
        sid: switch id
        dst: dst mac address
        port: output port
        """

        dp = self.network.nodes[sid]['switch'].dp

        #Try to finish this function yourself
```

Solution

```
    def _add_flow_entry(self, sid, dst, port):
        """Adds flow entries on switch sid,
        outputting all (allowed) traffic with destination address dst to port.

        Arguments:
        sid: switch id
        dst: dst mac address
        port: output port
        """

        dp = self.network.node[sid]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=443, eth_dst = dst)
        actions = [parser.OFPActionOutput(port)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=2,
        match=match, instructions=instr)
        dp.send_msg(cmd)

        match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=80, eth_dst = dst)
        cmd = parser.OFPFlowMod(datapath=dp, priority=2,
        match=match, instructions=instr)
        dp.send_msg(cmd)

        self.logger.info('ADDED␣FLOWS␣ON␣SWITCH␣' + str(sid) + "␣TO␣DESTINATION␣" + str(dst))
```

Since the `packet_in_handler` subscribes to the `PacketIn` event, all incoming packets will trigger *packet_in_handler*. Thus, packet handling and routing code are implemented in this function. *packet_in_handler* also receives all LLDP packets, which we will need to filter out to prevent spamming our log with needless information:

```
    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg

        pkt = packet.Packet(msg.data)

        eth = pkt[0]

        if eth.protocol_name != 'ethernet':
            #We should not receive non-ethernet packets,
            #as these are dropped at the switch
            self.logger.warning('Received␣unexpected␣packet:')
            self.logger.warning(str(pkt))
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return
```

```
        self.logger.info('Received␣ethernet␣packet')
        src = eth.src
        dst = eth.dst
        self.logger.info('From␣' + src + '␣to␣' + dst)
```

Exercise: Next, you need to install a path from the source switch to *dst* in the network (implement the `_install_path` and `_output_packet` functions yourself):

```
        self.logger.info('Received␣ethernet␣packet')
        src = eth.src
        dst = eth.dst
        self.logger.info('From␣' + src + '␣to␣' + dst)

        if eth.ethertype != 0x0800:
            #We should not receive non-IPv4 packets,
            #as these are dropped at the switch
            self.logger.warn('Packet␣ethertype␣is␣not␣IPv4')
            return

        ip = pkt[1]

        if ip.proto != 0x06:
            #We should not receive non-TCP packets,
            #as these are dropped at the switch
            self.logger.warn('Packet␣IP␣protocol␣is␣not␣TCP')
            return

        tcp = pkt[2]

        if tcp.dst_port != 443 and tcp.dst_port != 80:
            #We should not receive these packets,
            #as they are dropped at the switch
            self.logger.warn('Packet␣has␣blocked␣TCP␣dst␣port:␣' + tcp.dst_port)
            return

        if dst not in self.network:
            #We have not yet received any packets from dst
            #So we do not now its location in the network
            #Simply broadcast the packet to all ports without known links
            self._broadcast_unk(msg.data)
            return

        dp = msg.datapath
        sid = dp.id

        #Compute path to dst
        try:
            path = nx.shortest_path(self.network, source=sid, target=dst)
        except (nx.NetworkXNoPath, nx.NetworkXError):
            self.logger.warning('No␣path␣from␣switch␣' + str(sid) + '␣to␣' + dst)
            return False

        self._install_path(path)

        #Send packet directly to dst
        self._output_packet(path[-2], [self.network[path[-2]][path[-1]]['src_port']],
        msg.data)

    def _install_path(self, path):
        """Installs path in the network.
        path[-1] should be a host,
        all other elements of path are required to be switches
```

```
        Arguments:
        path: Sequence of network nodes
        """

        #Implement this function yourself

    def _output_packet(self, sid, ports, data):
        """Output packet to ports ports of switch sid

        Arguments:
        sid: switch id
        ports: output ports
        data: packet
        """

        self.logger.info('Outputing packet to ports ' + str(ports) + ' of switch ' + str(sid))

        #Implement this function yourself
```

Solution

```
    def _install_path(self, path):
        """Installs path in the network.
        path[-1] should be a host,
        all other elements of path are required to be switches

        Arguments:
        path: Sequence of network nodes
        """

        dst = path[-1]

        for i in range(0,len(path)-1):
            current = path[i]
            next = path[i+1]

            port = self.network[current][next]['src_port']

            self._add_flow_entry(current, dst, port)

    def _output_packet(self, sid, ports, data):
        """Output packet to ports ports of switch sid

        Arguments:
        sid: switch id
        ports: output ports
        data: packet
        """

        self.logger.info('Outputing packet to ports ' + str(ports) + ' of switch ' + str(sid))

        dp = self.network.node[sid]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        actions = [parser.OFPActionOutput(port) for port in ports]
        cmd = parser.OFPPacketOut(dp, buffer_id = ofp.OFP_NO_BUFFER,
        in_port=ofp.OFPP_CONTROLLER, actions=actions, data=data)
        dp.send_msg(cmd)
```

If the destination has not yet sent a valid IP packet to the network, the app does not know its location in the network. In this case we simply broadcast the packet to all output ports of the network to find out the location of the destination host:

```
def _broadcast_unk(self, data):
    """Output packet to all ports in the network without known links

    Arguments:
    data: packet
    """

    for node in self.network:
        if not self.network.node[node]['host']:
            switch = self.network.node[node]['switch']

            all_ports = [p.port_no for p in switch.ports]

            #If the number of links per switch is very large
            #it might be more efficient to generate a set instead of a list
            #of known ports
            known_ports = [self.network[node][neighbor]['src_port']
            for neighbor in self.network.neighbors(node)]

            unk_ports = [port for port in all_ports if port not in known_ports]

            self._output_packet(node, unk_ports, data)
```

2.4.3 Testing the App

To test the app, start Mininet with any topology with at least two hosts separated by more than 1 switch.

Choose two hosts, one client and one server. For the rest of this section, we assume the server is h1 and the client is h2. The following is an example for steps testing the App:

1. First, start a new window for h1:

   ```
   xterm h1
   ```

2. Start a netcat listener on port 80 of h1 with:

   ```
   nc -l -p 80
   ```

3. Start a netcat client on h2:

   ```
   h2 nc h1 80 -p 80
   ```

4. Type some text, this text should be received at the listener.

5. You should see packets successfully being received by h1. Use `ovs-ofctl` to view the flow tables on the path from h2 to h1. Check if the correct flow entries have been installed.

6. If the controller keeps receiving all TCP packets sent by the client, you might have set the priority of the flow entries installed by *_add_flow_entry* too low, try setting the priority to 2.

7. Stop the server.

8. Next, without resetting the controller or Mininet, repeat the steps above for ports 443 and 90. If the app has been configured correctly, you should receive traffic on port 443, but not on port 90.

9. If you do receive traffic on port 90, you might have forgotten to match on destination ports 443/80 in *_add_flow_entry*. By not filtering on these ports, the switches will forward all TCP packet sent to h1, instead of only the TCP packet sent to ports 443/80.

2.4.4 The Final Touch

You might have noticed that your app often tries to add flow entries to switches that already contain that entry. This can occur due to two reasons:

1. The controller receives many packets from the same source to the same destination after another, as multiple packets were received by the switch before the controller could install the first flow entry.

2. If multiple sources sent packets to the same destination, their paths may partially overlap, but the controller will install the whole path for all sources.

Try to change the app to only install new flow entries in a switch when required. Hint: You can use the map `self.network.nodes[sid]['flows']` to store all flow entries of switch *sid*.

Solution

We only need to make a small change to `_add_flow_entry`:

```
    def _add_flow_entry(self, sid, dst, port):
        """Adds flow entries on switch sid,
        outputting all (allowed) traffic with destination address dst to port.

        Arguments:
        sid: switch id
        dst: dst mac address
        port: output port
        """

        flows = self.network.nodes[sid]['flows']

        if dst not in flows:
            dp = self.network.nodes[sid]['switch'].dp
            ofp = dp.ofproto
            parser = dp.ofproto_parser

            match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=443, eth_dst = dst)
            actions = [parser.OFPActionOutput(port)]
            instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
            cmd = parser.OFPFlowMod(datapath=dp, priority=2, match=match, instructions=instr)
            dp.send_msg(cmd)

            match = parser.OFPMatch(eth_type = 0x0800, ip_proto=0x06, tcp_dst=80, eth_dst = dst)
            cmd = parser.OFPFlowMod(datapath=dp, priority=2, match=match, instructions=instr)
            dp.send_msg(cmd)

            flows[dst] = port

            self.logger.info('ADDED FLOWS ON SWITCH ' + str(sid) + " TO DESTINATION " + str(dst))
```

After you have made these changes, test the app again.

If the app still works at it is supposed to, congratulations! You have finished your first Ryu application and are now ready to create Ryu apps by yourself. `http://ryu.readthedocs.io/en/latest/ofproto_ref.html` should provide you with all the information you need on sending commands to the switches. If you are looking for specific details on the inner workings of OpenFlow, you can look up the official OpenFlow Switch Specification.

2.5 Useful Commands

```
sudo mn --custom path/filename.py --topo topology_name --mac --arp --controller=remote
        --switch ovs,protocols=OpenFlow13
```

Start a Mininet network with custom file and topology that connects to a remote controller on the default IP address (127.0.0.1) and port (6633). `--mac` tells Mininet to make MAC addresses more human-readable, and `--arp` to prepopulate the ARP tables of all hosts. The switch OF protocol is set to OF 1.3.

`PYTHONPATH=. ./bin/ryu-manager path/filename.py`
(In the ryu directory) Start the Ryu controller and instruct it to run the `path/filename.py` app.
Add `--observe-links` to start the controller with the topology detection module turned on.

`sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s*`
Show all flow entries of switch `s*` (assuming the switch is running OF 1.3)

`sudo ovs-ofctl --protocols=OpenFlow13 dump-groups s*`
Show all group entries of switch `s*` (assuming the switch is running OF 1.3)

`openflow_v4`
Wireshark OF 1.3 filter

`nc -l -p port`
Start a netcat listener on port `port`

`nc dst dst_port -p port`
Start a netcat client connecting to port `dst_port` of `dst` on port `port`
e.g. `nc h1 90 -p 80`

2.6 Useful Links

NetworkX
https://networkx.github.io/
https://networkx.github.io/documentation/stable/reference/introduction.html#networkx-basics

Ryu Packet parsing and creation library
https://ryu.readthedocs.io/en/latest/library_packet.html
https://ryu.readthedocs.io/en/latest/library_packet_ref.html
https://osrg.github.io/ryu-book/en/html/packet_lib.html

Ryu OpenFlow v1.3 Messages and Structures
https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html

Packet out
https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#packet-out-message

Modifying flow (group) entries with OFPFlowMod (OFPGroupMod)
https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#modify-state-messages

Flow match structure
https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#flow-match-structure

Actions
https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#action-structures

Chapter 3

Lecture 3: Quality of Service

Quality of Service (QoS) is the measurement of overall performance of a network service. SDN offers the opportunity to implement different Quality of Service (QoS) mechanisms in the network. The controller has a central view of the whole network, making it easier to react and adapt to changing network conditions. A real-time monitoring system is essential for this. In these exercises, you will create a controller app that monitors two important QoS metrics (throughput and delay) for flows in the network.

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/8op45oQrGvZB62F` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_3` in which you can do the exercises.

Solution

The solution files are available here: `https://surfdrive.surf.nl/files/index.php/s/jESqVQuB2nNhT2B`

3.1 Preparation - Ring Topology

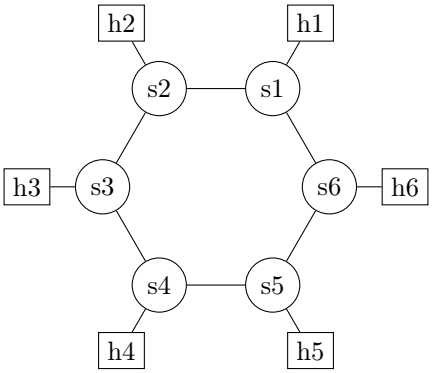


Figure 3.1: Ring topology ($n = 6$).

Exercise:
Write a Mininet Python script that can create ring topologies of any size n. In a ring topology, the switches form a single connected ring (all switches are connected and each switch is linked to exactly two other switches) and each switch connects to one host. See figure 3.1 for an example of a ring topology of $n = 6$ switches.

In addition, write a small automation script that starts an iperf server on host `h1` and an iperf client (connecting to the server) on host `h5`. The client should run for 20 seconds (this can be done using `iperf -t 20`), after which the server should be shut down again.

When running the script in the ring topology, iperf fails. Why?

Solution

The code for the ring topology:

```
from mininet.topo import Topo

class RingTopo(Topo):
    "Simple_topology_example."
```

```
    def __init__(self, n):
        "Create custom topo."
        # Initialize topology
        Topo.__init__(self)
        # Add hosts and switches
        switches = []
        hosts = []
        # connect switches to the hosts
        for i in range(n):
            switches.append(self.addSwitch('s%d' % (i+1)))
            hosts.append(self.addHost('h%d' % (i+1)))
            self.addLink(switches[i], hosts[i])

        for i in range(n):
            self.addLink(switches[i], switches[(i+1) % n])


topos = {'ring': RingTopo }
```

The code for the automation script:

```
h1 iperf -s &
h5 iperf -c 10.0.0.1 -t 20
h1 kill %iperf
```

To create the custom topology you should run the command:

```
sudo mn --custom ring.py --topo ring,5
```

After you run the custom script, you should get the following output:

```
connect failed: No route to host
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

As you can notice, a TCP connection cannot be established between hosts h1 and h5. The reason for this is the presence of loops in the topology (cf. chapter 1, exercise 5, Section 1.8.2). By default, Mininet switches act as learning switches and have no mechanisms to handle broadcast storms. Thus, in the next part of this exercise we will add an SDN controller to solve the issue.

3.2 Exercise 1. - Bandwidth monitoring

First, we will add an app that routes *all* Ethernet packets between Mininet hosts. Open the Ryu application in `Monitoring.py` (from the archive for this week that you downloaded).

This application forwards packets based on the source and destination MAC addresses. It is similar to the Firewall application from week 2; the main difference is the implementation of the `_add_flow_entry` and `packet_in_handler`.

```
        def _add_flow_entry(self, sid, src, dst, port, ethtype=""):
            """Adds flow entries on switch sid,
            outputting all (allowed) traffic with destination address dst to port.

            Arguments:
            sid: switch id
            src: src mac address
            dst: dst mac address
            port: output port
            ethtype: ethernet type
            """
            dp = self.network.nodes[sid]['switch'].dp
            ofp = dp.ofproto
            parser = dp.ofproto_parser
            # Add flow rule to match all other data packets
            match = parser.OFPMatch(eth_src=src, eth_dst=dst)
```

```
            actions = [parser.OFPActionOutput(port)]
            priority = 1

            instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
            cmd = parser.OFPFlowMod(datapath=dp, priority=priority, match=match, instructions=instr)
            dp.send_msg(cmd)

            self.logger.info('ADDED FLOWS ON SWITCH ' + str(sid) + " TO DESTINATION " + str(dst))

    # Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
            msg = ev.msg
            pkt = packet.Packet(msg.data)
            eth = pkt[0]
            src = eth.src
            dst = eth.dst
            if eth.protocol_name != 'ethernet':
                    # We should not receive non-ethernet packets
                    self.logger.warning('Received unexpected packet:')
                    self.logger.warning(str(pkt))
                    return

            # Don't do anything with LLDP, not even logging
            if eth.ethertype == ether_types.ETH_TYPE_LLDP:
                    return

            self.logger.info('Received ethernet packet')
            self.logger.info('From ' + src + ' to ' + dst)

            if dst not in self.network:
            # We have not yet received any packets from dst
            # So we do not know its location in the network
            # Simply broadcast the packet to all ports without known links
                    self._broadcast_unk(msg.data)
                    return

            dp = msg.datapath
            sid = dp.id
            if eth.ethertype == 0x0800:
            # Compute path to dst
                    try:
                            path = nx.shortest_path(self.network, source=sid, target=dst)
                    except (nx.NetworkXNoPath, nx.NetworkXError):
                            self.logger.warning('No path from switch ' + str(sid) + ' to ' + dst)
                            return False

            self._install_path(src, path)

            # Send packet directly to dst
            self._output_packet(path[-2], [self.network[path[-2]][path[-1]]['src_port']], msg.data)
```

While the Firewall application filtered packets based on the destination MAC address, in this application both source and destination MAC are used to identify separate flows (when `_add_flow_entry` triggers). Alternatively, you could use source and destination IP addresses or any other unique flow identifier. Additionally, the `packet_in_handler` was simplified to process all Ethernet packets. If you run into problems in this step, reread the **Routing packets** subsection from Chapter 2.

Exercise: As a simple test, create a ring topology of $n = 5$ switches and use the `pingall` command to verify connectivity (remember to start Mininet with `--arp` enabled). Additionally, run the automation script you previously created and compare the outputs with and without the controller.

Don't forget to run the Ryu controller with the `--observe-links` option to enable link discovery.

Solution

The output of the pingall command should look like this.

```
mininet> pingall
*** Ping: testing ping reachability
h0 -> h1 h2 h3 h4
h1 -> h0 h2 h3 h4
h2 -> h0 h1 h3 h4
h3 -> h0 h1 h2 h4
h4 -> h0 h1 h2 h3
*** Results: 0\% dropped (20/20 received)
```

The output of the automation script should look like:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.5 port 60732 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-20.0 sec  46.8 GBytes  20.1 Gbits/sec
[  4] local 10.0.0.1 port 5001 connected with 10.0.0.5 port 60732
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-20.0 sec  46.8 GBytes  20.1 Gbits/sec
```

Next, you will extend the app to make it monitor the throughput of each flow by periodically querying switches.

3.2.1 Collecting monitoring statistics

The OpenFlow protocol defines several messages that allow the controller to query switches for statistics about their current state, such as flow stats, port stats, and table stats (Fig. 3.2).

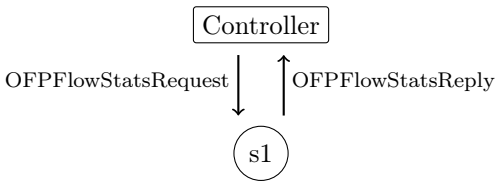


Figure 3.2: Polling statistics from an OpenFlow switch.

Information about the query messages supported by Ryu can be found at: `https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#multipart-messages`. For our purposes within the scope of this exercise, the following message types suffice:

- `OFPFlowStatsRequest` and `OFPFlowStatsReply` messages to request individual flow statistics (e.g. byte_count, duration_sec etc.)
- `OFPAggregateStatsRequest` and `OFPAggregateStatsReply` messages to request aggregate flow statictics.
- `OFPPortStatsRequest` and `OFPPortStatsReply` messages to request ports statistics (e.g. rx_packets, tx_packets, rx_errors, tx_errors, etc.).
- `OFPPortDescStatsRequest` and `OFPPortDescStatsReply` messages to query port descriptions (e.g. max_speed, and curr_speed).

To add monitoring functionality to the app, our first step is to create two functions:
`request_stats` and `flow_stats_reply`.
The functions are included in `Monitoring.py`, but you need to uncomment them. You may want to add a small change to `request_stats` to avoid that it polls stats for every flow from the switch, and you will have to add a chunk of code to `flow_stats_reply` (as part of this exercise).
`request_stats` will be used to send FlowStats messages to switches. When a switch receives this message, it will send back a FlowStatsReplay message. We process this message in the `flow_stats_reply` function.

```
def request_stats(self, sid):
    """ Send statistics request to a switch sid
    Arguments:
    sid: switch id
    """
    # You will have to change this function a bit
    dp = self.network.nodes[sid]['switch'].dp
    ofp = dp.ofproto
    parser = dp.ofproto_parser

    req = parser.OFPFlowStatsRequest(dp)
    dp.send_msg(req)

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def flow_stats_reply(self, ev):
    """Process flow stats reply info.
    Calculate flow speed and save it.
    """

    sid = ev.msg.datapath.id
    self.logger.info('Receiving statistics from switch ' + str(sid))

    body = ev.msg.body
    # Fill in this part yourself
```

The `OFPFlowStatsReply` message contains both the total amount of bytes sent through each flow entry, as well as the duration each entry has been installed for.

1. Implement a function called `monitor` that periodically (e.g., every second) polls all switches in the network and computes the average throughput of each flow entry in each switch since the last time they were polled.

2. Finally, create a separate thread to run `monitor` on by adding the following line to your init function:

   ```
   self.monitor_thread = hub.spawn(self.monitor)
   ```

   The above line creates a new attribute `monitor_thread` and calls the function hub.spawn() to create a new thread. Do not forget to import the hub module from ryu.lib (`from ryu.lib import hub`).

Hint: It will be helpful to create a new class "Flow" to create objects to store values in.
After you've completed your implementation, test your app as follows:

1. Start Mininet with your ring topology. If you want to, you can limit the bandwidth of all links, e.g. to 20 Mbps.

2. Start your monitoring app.

3. Use the automation script you prepared before to start an iperf server and client.

4. Check the output from the monitoring app. Compare the values computed by your app to those of iperf, and to the pre-configured bandwidth limit if you had set one. Are there any differences?

What would be advantages or disadvantages of shorter or longer monitoring intervals?

Solution

To store the flows we wish to monitor we will use an array. Additionally, we will specify a variable to represent the monitoring interval.

```
def __init__(self, *args, **kwargs):
    super(Monitoring, self).__init__(*args, **kwargs)
    self.network = nx.DiGraph()
    self.monitored_paths = [] #array to store information about flows we wish to monitor
    self.monitor_period = 1
    self.monitor_thread = hub.spawn(self._monitor)
```

Since we want to collect several matrices for each flow we can simply store them in a separate structure as shown below. We will extend this structure as we add more matrices (e.g. delay, loss, etc.).

```
class Flow():
        def __init__ (self, src, dst, did, port):
                #values needed to identify the flow (src mac, dst mac, first and last switch)
                self.destinationSwitch = did
                self.src_mac = src
                self.dst_mac =dst
                #port from the first switch specifying the port to which the probe packet will be sent out to
                self.sourcePort = port
                self.throughput = 0
                self.bytes_last = 0

        def __eq__(self, other):
        #override the default equal behaviour
                if isinstance(other, self.__class__):
                        return (self.src_mac == other.src_mac) and (self.dst_mac == other.dst_mac)
                else:
                        return False

        def __ne__(self, other):
        #override the default unequal behaviour
                if isinstance(other, self.__class__):
                        return (self.src_mac != other.src_mac) or (self.dst_mac != other.dst_mac)
                else:
                        return True

        def __str__ (self):
                return 'Sw: %d (%s - %s) - (%s kbps)' %
                    (self.destinationSwitch, self.src_mac, self.dst_mac, self.throughput)
```

For each of the monitored flows we need to remember the first and the last switch in the path as they carry the most information. For example, to calculate the throughput we only need to query the last switch in the path as all the packets that are received by the destination host are processed by this switch. Additionally, in the `packetIN` handler you need to add every new flow you wish to monitor to the `monitored_paths` array.

```
        flow = Flow(src, dst, path[len(path)-2], self.network[path[0]][path[1]]['src_port'])
        if flow not in self.monitored_paths:
            self.monitored_paths.append(flow)
```

The function to request the statistics from the switch is shown below. The only difference is to the function shown in the additional match argument. If the match field is not included, this function will poll all the flow stats from the switch, for every flow entry present. In our case, since the number of entries is small, this would not be a problem as we can simply filter out the stats we are interested in. However, in case of a switch that has hundreds of flow entries, the processing delay, as well as the received message would be too large and cause too much overhead. To avoid this, an additional match argument is specified and one request message is sent per each monitored flow.

```
        def request_stats(self, sid, src, dst):
                """ Send statistics request to a switch sid
                Arguments:
                sid: switch id
                src: src mac address
                dst: dst mac address
                """
                dp = self.network.nodes[sid]['switch'].dp
                ofp = dp.ofproto
                parser = dp.ofproto_parser
                #only ask for the stats for the flow we monitor
                #If this wasn't included we would request all the flows. This would
                #increase the processing delay at the switch
                match = parser.OFPMatch(eth_src = src, eth_dst = dst)
                req = parser.OFPFlowStatsRequest(dp, match=match)
                dp.send_msg(req)
```

The function to process the statistics received from the switch is shown below. The body of this message will contain the statistics for all the flows we requested in the previous message. As we only requested one (by specifying the match

field) this message will contain only one stats message.

```
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def flow_stats_reply(self, ev):
    """Process flow stats reply info.
    Calculate flow speed and save it.
    """

    sid = ev.msg.datapath.id
    body = ev.msg.body
    #self.logger.info('Receiving statistics from switch ' + str(sid))
    #process the replay message

    for stat in body:
        #the switch will return the stats for all flow entries that match src and dst address
        #for the first switch that will be just one entry, but for the second switch it will be 2
        #(one entry to match packets, one entry to match probe packets)
        #do not process probe packets
        if "eth_type" not in stat.match or stat.match["eth_type"]==2048:
            #find a flow from monitored_paths that corresponds to this replay
            matches = [flow for flow in self.monitored_paths if
                (flow.src_mac == stat.match["eth_src"]
                        and flow.dst_mac == stat.match["eth_dst"])]
            #there should be only one match as we sent the requests with the match field
            matches[0].throughput =
                    8*(stat.byte_count-matches[0].bytes_last)/(1000*self.monitor_period)
            matches[0].bytes_last = stat.byte_count
```

This function calculates the throughput in bits for the specified monitoring interval. The received *byte* count needs to be multiplied by 8 in order to receive the requested throughput in *bits* per second.

Finally, we need to create a monitoring function that is called by the previously defined monitoring thread:

```
def _monitor(self):
    """
    Main method for the monitoring actions.
    """
    while True:
        self.logger.info('Monitoring_stats:_num_flows:' + str(len(self.monitored_paths)))
        for m in self.monitored_paths:
            print m
            #save timestamps to measure delay from the controller to the switches
            self.request_stats(m.destinationSwitch, m.src_mac, m.dst_mac)
        hub.sleep(self.monitor_period)
```

The output of this app is shown below (for a maximum bandwidth of $\approx 40Mbps$). Two flows are detected. The second flow is representing the useful traffic sent from the source to the destination, while the first flow represents the ACK messages sent in the opposite direction (used protocol is TCP thus the traffic is always bidirectional). If the monitoring interval is small, we can observe the slow-start phase of the TCP connection, i.e. the sender starts slowly and doubles its sending rate every RTT until a loss is detected.

```
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (0 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (3372 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (13297537 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (22466 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (34428398 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (27281 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (39330670 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (33629 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (41421782 kbps)
Monitoring stats: num_flows:2
```

```
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (35722 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (41271127 kbps)
```

By changing the value of the monitoring period to 3 seconds:

```
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (0 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (20409 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (28069949 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (27639 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (36958272 kbps)
```

With a longer interval, we get less information, but we cause less additional load in the network.

3.3 Exercise 2. - Delay monitoring

Although the switches can provide the controller with a variety of statistics and information, some information can only be obtained by active measurement. For example, we cannot query OF switches for network delay. Thus, if we want to know how much delay the traffic in our network experiences, we need to actively send test packets to the network ourselves. In this final QoS exercise, you will learn how to use Ryu's capability to inject packets into the network.

Suppose we want to measure the delay of a flow travelling from switch `s1` to switch `s3`. To do so, your app should periodically send a probe to switch `s1`, which then should take the exact same path as the flow itself to switch `s3`. Now, by instructing `s3` to send these packets back to the controller, your app can determine the total latency of the path. This is demonstrated in Figure 3.3.

Theoretically, to determine the total latency over the path, the app needs to obtain three values:

- the delay of the probe packet t_{probe} (on the path from s1 to s3 in our example)
- the delay between the first switch and the controller t_1
- the delay between the last switch and the controller t_3

Then, the delay for the path can be calculated by

$$t_{probe} - t_1 - t_3$$

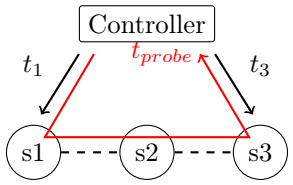


Figure 3.3: Calculating the delay of a path passing through switches s1 - s2 - s3.

For us, all the switches and the controller run in the same VM. So, values t_1 and t_3 should not have a significant contribution. Moreover, the calculation of t_1 and t_3 can be inaccurate and depends on the scheduling of different processes by the Linux kernel or the resources assigned to the VM. Thus, in this example, we will only calculate the t_{probe} value by keeping track of when the probe packet was sent to the first switch and received by the controller again.

Injecting packets into the network

To complete this exercise you will need to inject packets into the network. The following function can be used to send a probe packet out of a specific port on a switch:

```
def send_latency_probe_packet(self, sid, port):
    '''
    Injects latency probe packets in the network
    '''
    self.logger.info('Injecting latency probe packets')
    dp = self.network.nodes[sid]['switch'].dp
    actions = [dp.ofproto_parser.OFPActionOutput(port)]
```

```
        pkt = packet.Packet()
        pkt.add_protocol(ethernet.ethernet(ethertype=0xBBBB,
                                           dst=0xAAAAAAAAAAAA,
                                           src=0xBBBBBBBBBBBB))

        pkt.serialize()
        payload = '%d;%f' % (sid, time.time())
        # We must encode the payload to be able to append it to data
        # because data is a bytearray
        data = pkt.data + payload.encode()

        out = dp.ofproto_parser.OFPPacketOut(datapath=dp,
                                             buffer_id=dp.ofproto.OFP_NO_BUFFER,
                                             data=data,
                                             in_port=dp.ofproto.OFPP_CONTROLLER,
                                             actions=actions)
        dp.send_msg(out)
```

(Be careful with copy-pasting from this listing, as some characters (e.g. ') might not get recognized correctly.)

Do not forget to import all required modules:

```
from ryu.lib.packet import (packet, ethernet, ipv4, ipv6)
import time
```

Exercise:

Modify the function `send_latency_probe_packet` to fit your needs. Since you can customize all headers of the packet, make sure that your probe packet will match the same flow entries as the packets generated by the hosts.

Additionally, notice that you can embed the timestamp of the moment when the packet was sent in the payload of the probe packet. You can use this to simplify the calculation of t_{probe}. Modify your PacketIn handler (function `packet_in_handler`) to process the probe packets in addition to the useful data and calculate t_{probe}. Do not forget to install additional rules in the last switch that will forward the probe packet back to the controller.

Finally, make sure to call the function `send_latency_probe_packet` for every monitored path from your previously created monitoring thread.

Test your app as follows:

1. Extend the ring topology by adding bandwidth and delay properties to every link of the ring. There should be no delay between the hosts and the switches (only between switches).
2. Go through the same steps as in exercise 1.

Can you measure the delay?
How many additional flow entries did you need to install?

Solution

To calculate the delay we need to first extend the previously defined Flow class as shown below, while the instantiation of the Flow objects needs to be updated accordingly:

```
class Flow():
        def __init__ (self, src, dst, sid, did, port):
                #values needed to identify the flow (src mac, dst mac, first and last switch)
                self.destinationSwitch = did
                self.sourceSwitch = sid
                self.src_mac = src
                self.dst_mac =dst
                #port from the first switch specifing the port to which the probe packet will be sent out to
                self.sourcePort = port
                self.delay = 0 #delay of the probe packet
                self.throughput = 0
                self.bytes_last = 0

        def __eq__(self, other):
        #override the default equal behaviour
                if isinstance(other, self.__class__):
```

```
                return (self.src_mac == other.src_mac) and (self.dst_mac == other.dst_mac)
        else:
                return False

    def __ne__(self, other):
    #override the default unequal behaviour
        if isinstance(other, self.__class__):
                return (self.src_mac != other.src_mac) or (self.dst_mac != other.dst_mac)
        else:
                return False

    def __str__ (self):
        return 's%d-->s%d (%s - %s) - (%s kbps %s ms)' % (self.destinationSwitch,
                self.sourceSwitch, self.src_mac, self.dst_mac, self.throughput, self.delay*1000)
```

We will first explain how to calculate the delay t_{probe}. We first need to modify the `send_latency_probe_packet` function to fit our needs. To do this, we will add additional arguments to this function such as the switch ID (of the first switch for the monitored path) to send the probe packet to, as well as the output port on the first switch to forward this packet to. Additionally, we will use the source and destination MAC addresses of the monitored flow. This way, the probe packet will be forwarded using the same processing rules as the packets belonging to the flow.

```
    def send_latency_probe_packet(self, sid, port, src, dst):
        '''
        Injects latency probe packets in the network
        Arguments:
        sid: switch id
        src: src mac address
        dst: dst mac address
        port: output port

        '''
        #self.logger.info('Injecting latency probe packets')
    dp = self.network.nodes[sid]['switch'].dp
    actions = [dp.ofproto_parser.OFPActionOutput(port)]

    pkt = packet.Packet()
    # probe packet should have the same source and destination mac as the packets belonging to the monitored flow
    # these packets should be matched by the rules of the original flow on all the switches except the
    # first one
    pkt.add_protocol(ethernet.ethernet(ethertype=self.PROBE_ETHERTYPE, dst=dst, src=src))
    pkt.serialize()
    payload = '%d;%f' % (sid, time.time())
    # We must encode the payload to be able to append it to data
    # because data is a bytearray
    data = pkt.data + payload.encode()

    out = dp.ofproto_parser.OFPPacketOut(
        datapath=dp,
        buffer_id=dp.ofproto.OFP_NO_BUFFER,
        data=data,
        in_port=dp.ofproto.OFPP_CONTROLLER,
        actions=actions)
    self.logger.info("Probe sent!")
    dp.send_msg(out)
```

Additionally, in the `init` method of the app we need to add an argument self.PROBE_ETHERTYPE that will be used to differentiate the probe packets from regular packets belonging to the flow. To forward this packet back to the controller at the last switch we need to add one additional rule that will match on the ethtype.

```
    def _install_path(self, src, path):
        """Installs path in the network.
        path[-1] should be a host,
        all other elements of path are required to be switches
```

```
        Arguments:
        path: Sequence of network nodes
        """

        dst = path[-1]

        for i in range(0,len(path)-1):
            current = path[i]
            next = path[i+1]

            port = self.network[current][next]['src_port']
            self._add_flow_entry(current, src, dst, port)
            #add additonal rule for a probe packet at the last switch
            if i == len(path)-2:
                self._add_probe_flow_entry(current, src, dst, port)
```

The `_add_probe_flow_entry` will install a rule to forward the probe packets to the controller. To make sure that probe packets are processed by this rule at the last switch we need to configure a higher priority than the rule to process "normal" packets.

```
    def _add_probe_flow_entry(self, sid, src, dst, port):
        """Adds probe flow entry on switch sid,
        outputting all (allowed) traffic with destination address dst to port.

        Arguments:
        sid: switch id
        src: src mac address
        dst: dst mac address
        port: output port
        ethtype: ethernet type
        """
        dp = self.network.nodes[sid]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch(eth_src = src, eth_dst = dst, eth_type = self.PROBE_ETHERTYPE)
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        priority = 2

        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=priority, match=match, instructions=instr)
        dp.send_msg(cmd)

        self.logger.info('ADDED PROBE ENTRY ON SWITCH ' + str(sid))
```

Finally, this packet is received in the PacketIN handler (method `packet_in_handler`) and needs to be parsed to extract the delay information as shown below.

```
    if eth.ethertype == self.PROBE_ETHERTYPE:
        #self.logger.info('Received probe packet')
        # data (ip header + timestamp (when the probe packet was sent)) is the payload
        # We must decode it to get a str (data is a bytearray)
        data = pkt[1].decode()
        split_data = data.split(';')
        timestamp = split_data[len(split_data)-1] # last field is the timestamp
        matches = [flow for flow in self.monitored_paths if (flow.src_mac == src and flow.dst_mac == dst)]
        matches[0].delay = time.time() - float(timestamp)
        return
```

If we look at the flow rules at the switches we can notice the following:

```
hpdn@hpdn-VirtualBox:~/ryu$ sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s2
 cookie=0x0, duration=12.587s, table=0, n_packets=27, n_bytes=1620, priority=65535,
    dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=5.739s, table=0, n_packets=2, n_bytes=158, priority=2,
     dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03,dl_type=0x07c7 actions=CONTROLLER:65535
```

```
 cookie=0x0, duration=5.756s, table=0, n_packets=199988, n_bytes=13199456, priority=1,
     dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02 actions=output:"s2-eth2"
 cookie=0x0, duration=5.740s, table=0, n_packets=416287, n_bytes=21120858142, priority=1,
     dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03 actions=output:"s2-eth1"
 cookie=0x0, duration=12.626s, table=0, n_packets=2, n_bytes=134,
        priority=0 actions=CONTROLLER:65535
hpdn@hpdn-VirtualBox:~/ryu$ sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s1
 cookie=0x0, duration=14.414s, table=0, n_packets=34, n_bytes=2040, priority=65535,
     dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=7.547s, table=0, n_packets=3, n_bytes=237, priority=2,
     dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02,dl_type=0x07c7 actions=CONTROLLER:65535
 cookie=0x0, duration=7.547s, table=0, n_packets=199988, n_bytes=13199456, priority=1,
     dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth1"
 cookie=0x0, duration=7.533s, table=0, n_packets=416287, n_bytes=21120858142, priority=1,
     dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03 actions=output:"s1-eth3"
 cookie=0x0, duration=14.423s, table=0, n_packets=4, n_bytes=280,
        priority=0 actions=CONTROLLER:65535
```

We can notice the additional rules (with the match field `dl_type=0x88cc`) for two flows (between h1 and h2 and between h2 and h1) that are installed only on the last switches and that have the action to send the packet back to the controller. Finally, we need to call the function `send_latency_probe_packet` from the `_monitor` method once for every monitored path.

The output of the app (in case the delay on all the links was configured to 100ms) is shown below. The last value represents t_{probe}.

```
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (52 kbps  107.933044434 ms )
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (9532 kbps  104.384899139 ms )
```

By increasing the delay on the links, we can observe that t_{probe} reflects this.

```
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (0 kbps  506.588935852 ms  )
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps  504.956007004 ms )
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (1 kbps  505.844831467 ms )
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps  507.808923721 ms )
```

Chapter 4

Lecture 4: Multicast and Group Tables

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/Coe8agGmalUVt3I` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_4` in which you can do the exercises. You will also need a sample video for this exercise that is available here: `https://surfdrive.surf.nl/files/index.php/s/4UNli9HfzxQtYp9`

Solution

The solution files are available here: `https://surfdrive.surf.nl/files/index.php/s/NOBxYO9j1ci7rqG`

4.1 Implementing Multicast in OpenFlow

Multicast is an efficient method for sending the same data to a group of hosts, as the same packet is not transmitted through the same link multiple times. For example, multicast would be very effective for streaming video content to multiple devices. In data centers, it is applied to significantly reduce the bandwidth required for group communication.

SDN is nicely suited to multicast traffic, as the controller has a central view of the whole network and can thus easily compute and install multicast trees. In these exercises, you will implement a multicast controller app yourself and test it by streaming a video to multiple hosts.

A complete version of this application should have the following functionality:

- Receive all IGMPv3 packets sent to the network.
- Maintain multicast groups [1] automatically based on INCLUDE and EXCLUDE packets (we will ignore all other IGMPv3 packets).
- When a host starts sending (non-IGMP) packets to a multicast group, install flow and group entries in the network to send these packets to all members of the group. *Packets should not be sent over the same link multiple times!*
- When a host leaves a multicast group, ensure no packets from that group are being sent to it anymore by removing or updating installed flow and group entries.
- When a host joins a multicast group, ensure packets being sent to that group are also being sent to this host, by adding or updating installed flow and group entries.

4.2 Exercise 1. - Processing IGMPv3 packets

We provide you a basic Ryu controller app that keeps track of the network topology and logs all incoming packets. In addition, the app installs a default table-miss entry in each switch that sends any unmatched packets to the controller.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```

[1] In your app

```
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto
from ryu.lib.packet import igmp

from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link, get_host

import networkx as nx

class Multicast_Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Multicast_Controller, self).__init__(*args, **kwargs)

        self.network = nx.DiGraph()
        self.subscribers = {}

    def log(self, message):
        self.logger.info(message)
        return

    #This function is triggered before the topology controller flows are added
    #But late enough to be able to remove flows
    @set_ev_cls(ofp_event.EventOFPStateChange, [CONFIG_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Delete any possible currently existing flows.
        del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL, out_port=ofp.OFPP_ANY,
            out_group=ofp.OFPG_ANY, command=ofp.OFPFC_DELETE)
        dp.send_msg(del_flows)

        #Delete any possible currently exising groups
        del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
            group_id=ofp.OFPG_ALL)
        dp.send_msg(del_groups)

        #Make sure deletion is finished using a barrier before additional flows are added
        barrier_req = parser.OFPBarrierRequest(dp)
        dp.send_msg(barrier_req)

    #Switch connected
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Add default flow
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)

    #Topology Events
    @set_ev_cls(event.EventSwitchEnter)
    def switchEnter(self,ev):
```

```
        switch = ev.switch

        self.network.add_node(switch.dp.id, switch = switch, flows= {}, host = False)
        self.log('Added switch ' + str(switch.dp.id))

    @set_ev_cls(event.EventSwitchLeave)
    def switchLeave(self,ev):
        switch = ev.switch
        sid = switch.dp.id

        if sid in self.network:
            #NOTE: In actual applications one should include some code to properly handle switches leaving
            #and re-entering!

            self.network.remove_node(sid)
            self.log('Removed switch ' + str(sid))

    @set_ev_cls(event.EventLinkAdd)
    def linkAdd(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        self.network.add_edge(src, dst, src_port = link.src.port_no, dst_port = link.dst.port_no, live = True)
        self.log('Added link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventLinkDelete)
    def linkDelete(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        if (src,dst) in self.network.edges():
            #NOTE: In actual applications one should include some code to properly handle link failures!

            self.network.remove_edge(src, dst)
            self.log('Removed link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventHostAdd)
    def hostFound(self,ev):
        host = ev.host
        switch = host.port.dpid
        mac = host.mac

        self._hostFound(switch, host.port.port_no, mac)

    def _hostFound(self, switch_id, port, mac):
        if mac not in self.network:
            self.network.add_node(mac, host = True)
            self.network.add_edge(mac, switch_id, src_port = -1, dst_port = port, live = True)
            self.network.add_edge(switch_id, mac, src_port = port, dst_port = -1, live = True)
            self.log('Added host ' + mac + ' at switch ' + str(switch_id))

    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        pkt = packet.Packet(msg.data)
        eth = pkt[0]
```

```
        if eth.protocol_name != 'ethernet':
            #Ignore non-ethernet packets
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return

        self.log('Received ethernet packet')
        src = eth.src
        dst = eth.dst
        self.log('From ' + src + ' to ' + dst)

        #host detection
        if src not in self.network:
            self._hostFound(dp.id, msg.match['in_port'], src)

        if eth.ethertype == ether_types.ETH_TYPE_IP:
            self.process_ip(dp, pkt)
            return

    def process_ip(self, dp, pkt):
        eth_pkt = pkt[0]
        ip_pkt = pkt[1]

        #IGMP message
        if ip_pkt.proto == in_proto.IPPROTO_IGMP:
            igmp_pkt = pkt[2]
            self.processIGMP(eth_pkt.src, ip_pkt.src, igmp_pkt)
            return

    def processIGMP(self, eth_src, ip_src, igmp_pkt):
        self.log('IGMP packet')
        # Implement this function yourself
```

Now, to be able to support hosts running IGMPv3, the app should at the very least be able to parse IGMPv3 membership reports. Whenever a host wants to join or leave a multicast group, it will send a membership report to the network. An IGMPv3 membership report is basically just a list of *group records*, one for each group the host wants to leave or join (or report on). Each of these group records, in turn, contains a multicast address (the group!), a list of source addresses, and a record field type. By including a list of source addresses, IGMPv3 allows hosts to make their own selection of source hosts to listen to. However, in this exercise, we will assume hosts always want to listen to all sources in a multicast group. Thus, we only support group records with empty source address lists.

If a host wants to join a group, the group record will have the `CHANGE_TO_EXCLUDE_MODE` type and an empty source list (indicating it wants to exclude none of the sources, i.e., listen to the whole group). Conversely, when a host wants to leave a group, the group record will have the `CHANGE_TO_INCLUDE_MODE` type and an empty source list.

Exercise:

Extend your app with the ability to process IGMPv3 report messages. For now, you only have to keep track of which hosts are subscribed to which multicast groups. You can use the Packet library (`https://ryu.readthedocs.io/en/latest/library_packet_ref/packet_igmp.html`) to extract all relevant properties. You can determine if an (IPv4) packet is an IGMP packet by importing `ryu.lib.packet.in_proto` and testing if `ip_pkt.proto == in_proto.IPPROTO_IGMP` (where `ip_pkt` is the extracted IPv4 packet. For any IGMP message `igmp_pkt`, you would have to do the following:

- Check if the message is an IGMPv3 membership report (`igmp_pkt.protocol_name == 'igmpv3_report'`), do nothing if not.
- Ignore any IGMPv3 messages with non-empty source lists
- For each group record `record`
  - If `record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE`, add the source host of the packet to the multicast group
  - If `record.type_ == igmp.CHANGE_TO_INCLUDE_MODE`, remove the source host of the packet from the multicast group

- Log any changes you make to the multicast groups

Hint: use `self.subscribers` to store subscribers when implementing the `processIGMP` function.
You can test your app as follows:

1. Start Mininet with the binary tree topology of depth 3 and connect it to a Ryu controller running your app.
2. Add a multicast route to the route table of hosts `h1`, `h2`, `h7`, and `h8` (if you want, you can create an automation script for this, as you will need to repeat this step for the next two exercises as well):
   `h* route add -net 224.0.0.0 netmask 240.0.0.0 dev h*-eth0`
3. Connect hosts `h1`, `h2`, and `h8` to an imaginary multicast stream; for each host:
   (a) Start vlc (if VLC does not start properly or complains about being run in root mode, run the following command (outside mininet) first: `sudo sed -i 's/geteuid/getppid/' /usr/bin/vlc`)
   (b) Click on `media -> Open Network Stream`
   (c) Fill in `udp://@224.0.0.14` as the network address and press `play`
   (d) You should see the host being added to the group (one or multiple times, depending on how many messages VLC decides to send)
4. Shut down VLC on all hosts one by one. Check if the app has removed the hosts from the group again.

Solution

```
def processIGMP(self, eth_src, ip_src, igmp_pkt):
    self.log('IGMP packet')

    #Only support IGMPV3 membership reports
    if igmp_pkt.protocol_name != 'igmpv3_report':
        return

    records = igmp_pkt.records
    for record in records:
        if not record.srcs:
            address = record.address
            self.log('Record change for group ' + address)

            if record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE:
                if address not in self.subscribers:
                    self.subscribers[address] = set()

                self.subscribers[address].add(eth_src)

                self.log('Added ' + eth_src)

            elif record.type_ == igmp.CHANGE_TO_INCLUDE_MODE:
                if address not in self.subscribers:
                    self.subscribers[address] = set()

                self.subscribers[address].discard(eth_src)

                self.log('Removed ' + eth_src)
```

4.3 Exercise 2. - Routing multicast traffic

When a host starts sending traffic to a multicast group, the app should route this traffic to all subscribers of the group. For now, we will assume that all subscribers are known as soon as a source host starts streaming. Thus, your app will only have to install new flow and group entries when it receives a multicast packet from a new source host + multicast destination address combination. To support multicast, we will sometimes need to output a single packet over multiple ports. To do so, we will make use of the group table.

4.3.1 Group Table

Besides outputting packets directly to a port, OpenFlow also allows flow entries to indicate that packets should be processed by a *group* instead. A group is essentially an ordered list of *action buckets*: a set of actions and associated parameters. The way packets are processed by the group depends on its type:

- **all**: The all group, which we will use in this exercise, simply executes the actions of all its action buckets. Thus, if we insert an action bucket for each output port, the packet will be forwarded to each of these ports.
- **select**: The select group is quite interesting, as it allows the controller to divide traffic load over multiple paths. The select group executes one action bucket per packet. The trick is that it selects this bucket in such a way that overall, the load is equally shared across all action buckets in the group.
- **indirect**: The indirect group only allows the controller to specify one action bucket, and will simply execute this single bucket.
- **fast failover**: The fast failover group is used for rapidly switching over to a backup path after a link or node failure, without requiring any response from the controller. Each action bucket in a fast failover group is associated with a specific port (or even another group). The group executes the first bucket with a *live* port. This means that as soon as the port of the first bucket (forwarding packets over the primary path) goes down, it will automatically start executing the second bucket (forwarding packets over a backup path) instead.

You might wonder why we need the indirect, or even the all group, as we can also output packets to one or more ports by directly applying one or more `OFPActionOutput` actions. The main benefit of both of these group types is that we can point multiple flow entries to the same group. This not only allows us to change the behavior of multiple flow entries simultaneously but can also help from an organizational perspective. For example, a network operator could pre-install a spanning tree in the network, using the all group, and broadcast any packets to all nodes in the network simply by forwarding them to the pre-installed group entries.

Furthermore, not all switches necessarily support forwarding a packet to multiple ports simultaneously by applying multiple `OFPActionOutput` actions. In contrast, all OF 1.3 switches are required to support the all group type. However, the main reason for using group entries in these exercises is simply to learn how to install and use group entries, as they are an essential part of the OpenFlow protocol.

4.3.2 Installing OF Group Entries - Example

You can install an all group as follows:

```
dp = self.network.nodes[switch_id]['switch'].dp
ofp = dp.ofproto
parser = dp.ofproto_parser

ports = [1,2,5]

action_lists = [[parser.OFPActionOutput(port)] for port in ports]
buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list) for action_list in action_lists]

#Make sure to choose a unique group_id for each group entry!
cmd = parser.OFPGroupMod(dp, ofp.OFPGC_ADD, ofp.OFPGT_ALL, group_id, buckets)
dp.send_msg(cmd)
```

To send packets to the group, use the `OFPActionGroup` action:

```
match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=dst_address)
actions = [parser.OFPActionGroup(group_id)]
instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
cmd = parser.OFPFlowMod(datapath=dp, priority=2,
    match=match, instructions=instr)
dp.send_msg(cmd)
```

4.3.3 Exercise Instructions

Multicast packets can be recognized by their Ethernet address. If the least significant bit of the first octet is a 1, it is a multicast address. For any multicast packet,

- IGMP packets also have a multicast address, so first determine if the packet is an IGMP packet or "normal" multicast traffic.

- Extract the destination IP address (IGMP uses IP multicast addresses, and not Ethernet addresses)
- If the source host + multicast destination combination is already known, we can ignore the packet (as we already installed all required flow and group entries).
- Otherwise, the app needs to install the required flow and group entries to route traffic from the source host to all subscribers of the group. The easiest way to do so is to compute and install the shortest path between each subscriber and the source host (this will result in a Shortest-Path Tree (SPT)). We just need to make sure that we only install one flow and group entry in each switch and do not duplicate packets over the same link. To route packets from the source to one or more subscribers, you will need to check each switch on the shortest paths:
  - If the switch already forwards packets to the next switches on the path: do nothing
  - Otherwise, and if the switch already contains a group entry for this multicast group: modify this group entry (`OFPGroupMod(dp, ofp.OFPGC_MODIFY, ...)`) to add additional action bucket(s).
  - Otherwise, install both a new all group entry, as well as a new flow entry pointing to this group.

  Hint: You can either match on both the source address and the IP destination address, or only on the IP destination address. In the latter case, you only need to install up to a single flow and group entry in each switch for each multicast group (saving memory). While in the former case you need to install entries for each source host + multicast group combination.

Similarly to the `dump-flows s*` command, you can use
`sudo ovs-ofctl --protocols=OpenFlow13 dump-groups s*` to show all group entries of switch `s*`.

To test your app, you will stream a video from one host to the others. Download the video from here if you haven't done so yet: `https://surfdrive.surf.nl/files/index.php/s/4UNli9HfzxQtYp9`

Test your app as follows:

1. Start Mininet with the binary tree topology of depth 3 and connect it to a Ryu controller running your app.
2. Add a multicast route to the route table of hosts `h1`, `h2`, `h7`, and `h8`:
   `h* route add -net 224.0.0.0 netmask 240.0.0.0 dev h*-eth0`
3. connect hosts `h1`, `h2`, and `h8` to the stream; for each host:
   (a) Start vlc
   (b) Click on `media -> Open Network Stream`
   (c) Fill in `udp://@224.0.0.14` as the network address and press `play`
4. Start vlc on host `h7`
5. Start streaming from this host as follows:
   (a) Click on `media -> Stream...`
   (b) Add `TestVideo.mp4` (which you downloaded before) and press stream
   (c) Press `next`, select `UDP (legacy)` as destination and press `add`
   (d) Fill in `224.0.0.14` as the address and press `next`
   (e) Click through the rest of the dialog and start streaming
6. You should now be receiving the stream on all three hosts!

Note: It might take a few seconds until you can see the actual video playback.

Solution

In our solution, we first compute the whole SPT by computing the shortest paths from the source host to each subscriber. We then iterate over each switch in the tree and install an all table that forwards packets to all its outgoing links in the tree.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```

```
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto
from ryu.lib.packet import igmp

from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link, get_host

import networkx as nx

class Multicast_Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    LOWPRIO = 1
    MEDPRIO = 2
    HIGHPRIO = 3
    HIGHESTPRIO = 4

    def __init__(self, *args, **kwargs):
        super(Multicast_Controller, self).__init__(*args, **kwargs)

        self.network = nx.DiGraph()
        self.subscribers = {}
        self.sources = {}
        self.group_ids = {}

        self.max_group_id = 0

    def log(self, message):
        self.logger.info(message)
        return

    #This function is triggered before the topology controller flows are added
    #But late enough to be able to remove flows
    @set_ev_cls(ofp_event.EventOFPStateChange, [CONFIG_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Delete any possible currently existing flows.
        del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL, out_port=ofp.OFPP_ANY,
            out_group=ofp.OFPG_ANY, command=ofp.OFPFC_DELETE)
        dp.send_msg(del_flows)

        #Delete any possible currently exising groups
        del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
            group_id=ofp.OFPG_ALL)
        dp.send_msg(del_groups)

        #Make sure deletion is finished using a barrier before additional flows are added
        barrier_req = parser.OFPBarrierRequest(dp)
        dp.send_msg(barrier_req)

    #Switch connected
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Add default flow
        match = parser.OFPMatch()
```

```
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)

    #Topology Events
    @set_ev_cls(event.EventSwitchEnter)
    def switchEnter(self,ev):
        switch = ev.switch

        self.network.add_node(switch.dp.id, switch = switch, groups = {}, host = False)
        self.log('Added switch ' + str(switch.dp.id))

    @set_ev_cls(event.EventSwitchLeave)
    def switchLeave(self,ev):
        switch = ev.switch
        sid = switch.dp.id

        if sid in self.network:
            #NOTE: In actual applications one should include some code to properly handle switches leaving
            #and re-entering!

            self.network.remove_node(sid)
            self.log('Removed switch ' + str(sid))

    @set_ev_cls(event.EventLinkAdd)
    def linkAdd(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        self.network.add_edge(src, dst, src_port = link.src.port_no, dst_port = link.dst.port_no, live = True)
        self.log('Added link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventLinkDelete)
    def linkDelete(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        if (src,dst) in self.network.edges():
            #NOTE: In actual applications one should include some code to properly handle link failures!

            self.network.remove_edge(src, dst)
            self.log('Removed link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventHostAdd)
    def hostFound(self,ev):
        host = ev.host
        switch = host.port.dpid
        mac = host.mac

        self._hostFound(switch, host.port.port_no, mac)

    def _hostFound(self, switch_id, port, mac):
        if mac not in self.network:
            self.network.add_node(mac, host = True)
            self.network.add_edge(mac, switch_id, src_port = -1, dst_port = port, live = True)
            self.network.add_edge(switch_id, mac, src_port = port, dst_port = -1, live = True)
            self.log('Added host ' + mac + ' at switch ' + str(switch_id))

    def install_group(self, switch_id, dst_address, dsts, group_id):
```

```
        """Iff group group_id has not yet been installed at switch switch_id,
        install flow and group entry in switch switch_id that forwards packets to all dsts.

        Arguments:
        switch_id: id of switch to add flow to
        dst_address: IP address of the destination (i.e. the group address)
        dsts: switch or host ids to output packet to
        group_id: id of the all group associated with dst_address
        """
        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        ports = [self.network[switch_id][dst]['src_port'] for dst in dsts]

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups:
            #Add group entry
            action_lists = [[parser.OFPActionOutput(port)] for port in ports]
            buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
                for action_list in action_lists]
            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_ADD, ofp.OFPGT_ALL, group_id, buckets)
            dp.send_msg(cmd)

            groups[group_id] = set(ports)

            #Add flow entry pointing to group
            match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=dst_address)
            actions = [parser.OFPActionGroup(group_id)]
            instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
            cmd = parser.OFPFlowMod(datapath=dp, priority=self.MEDPRIO,
                match=match, instructions=instr)
            dp.send_msg(cmd)

        self.log('INSTALLED GROUP' + str(group_id) + ' AT SWITCH ' + str(switch_id))

    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        pkt = packet.Packet(msg.data)
        eth = pkt[0]

        if eth.protocol_name != 'ethernet':
            #Ignore non-ethernet packets
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return

        self.log('Received ethernet packet')
        src = eth.src
        dst = eth.dst
        self.log('From ' + src + ' to ' + dst)

        #host detection
        if src not in self.network:
            self._hostFound(dp.id, msg.match['in_port'], src)
```

```
        if eth.ethertype == ether_types.ETH_TYPE_IP and self.isMulticast(eth.dst):
            self.process_multicast(dp, pkt)
            return

    def process_multicast(self, dp, pkt):
        eth_pkt = pkt[0]
        ip_pkt = pkt[1]

        #IGMP message
        if ip_pkt.proto == in_proto.IPPROTO_IGMP:
            igmp_pkt = pkt[2]
            self.processIGMP(eth_pkt.src, ip_pkt.src, igmp_pkt)
            return

        if ip_pkt.protocol_name != 'ipv4':
            #ignore IPv6
            return

        ip_dst = ip_pkt.dst
        ip_src = ip_pkt.src
        eth_src = eth_pkt.src

        if (ip_dst not in self.sources
            or eth_src not in self.sources[ip_dst]):
            self.add_source(dp, eth_src, ip_dst)


    def add_source(self, dp, eth_src, multicast_address):
        if multicast_address not in self.sources:
            self.sources[multicast_address] = set()

            self.max_group_id = self.max_group_id + 1
            self.group_ids[multicast_address] = self.max_group_id

        group_id = self.group_ids[multicast_address]

        self.sources[multicast_address].add(eth_src)

        self.log('Adding source ' + eth_src + ' to multicast group ' + multicast_address)

        if multicast_address not in self.subscribers:
            #no subscribers
            self.install_group(dp.id, multicast_address, [], group_id)
            return

        next_hops = {}

        for subscriber in self.subscribers[multicast_address]:
            try:
                path = nx.shortest_path(self.network, source=eth_src, target=subscriber)

                for i in range(1,len(path)-1):
                    switch = path[i]

                    if switch not in next_hops:
                        next_hops[switch] = set()

                    next_hops[switch].add(path[i+1])

            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(eth_src) + ' to ' + subscriber)
```

```
        for switch in next_hops:
            #If the group entry has already been installed, we do not need to modify anything
            #So we can safely use this method
            self.install_group(switch, multicast_address, next_hops[switch], group_id)


    def processIGMP(self, eth_src, ip_src, igmp_pkt):
        self.log('IGMP packet')

        #Only support IGMPV3 membership records
        if igmp_pkt.protocol_name != 'igmpv3_report':
            return

        records = igmp_pkt.records
        for record in records:
            if not record.srcs:
                address = record.address
                self.log('Record change for group ' + address)

                if record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE:
                    if address not in self.subscribers:
                        self.subscribers[address] = set()

                    self.subscribers[address].add(eth_src)

                    self.log('Added ' + eth_src)

                elif record.type_ == igmp.CHANGE_TO_INCLUDE_MODE:
                    if address not in self.subscribers:
                        self.subscribers[address] = set()

                    self.subscribers[address].discard(eth_src)

                    self.log('Removed ' + eth_src)


    def isMulticast(self, eth_dst):
        bit = eth_dst[1]
        return int(bit, 16) % 2 == 1
```

4.4 Exercise 3. - Dynamic Groups

Our final goal is to support dynamic multicast groups as well. Hosts can join and leave a group at any time. Thus, the controller needs to modify the multicast tree on the fly, as new subscribers are joining or leaving the group.

Extend your IGMPv3 processing code to

- Add paths from all sources to the new subscriber if a host subscribes to a group. You can do so by computing a shortest path from each source to the new subscriber, and iterating over each switch on each of these paths:
  - If the switch already forwards packets to the next switch on the path: do nothing
  - Otherwise, and if the switch already contains a group entry for this multicast group: modify this group entry (`OFPGroupMod(dp, OFPGC_MODIFY, ...)`) to add an additional action bucket.
  - Otherwise, install both a new all group entry, as well as a new flow entry pointing to this group.
- Remove the paths from all sources to a host if it decides to leave a group. You are allowed to assume the network remains static after a subscriber joins a multicast group (which means that the same NetworkX shortest path computation should result in exactly the same path again). You can do so by computing a shortest path from each source to the subscriber. Unfortunately, we can not just start modifying group entries on all the switches on these paths, as parts of this path may also be used to route packets to other subscribers. Thus, for each of these paths, we move *backward* starting from the last switch on the path:

- If the switch still forwards packets to the next hop on the path, modify the group entry to not forward packets to this port anymore. If this would result in an empty group entry, remove the group entry using `OFPGroupMod(dp, ofp.ofpOFPGC_DELETE,...)`. Deleting a group entry also removes the associated flow entry automatically.
- If you removed the group entry in the previous step (or there was no group entry), that means the previous switch on the path also does not need to forward packets to this switch anymore, so we can move to this switch and start at step 1 again.
- If the current switch still contains a group entry for the multicast group, that means the previous switch on the path still needs to forward packets to the current switch, so we do not need to modify any other switches on this path.

Test your app as follows:

1. Start Mininet with the binary tree topology of depth 3 and connect it to a Ryu controller running your app.
2. Add a multicast route to the route table of hosts `h1`, `h2`, `h7`, and `h8`:
   `h* route add -net 224.0.0.0 netmask 240.0.0.0 dev h*-eth0`
3. Start vlc on host `h7`
4. Start streaming from this host as follows:
   (a) Click on `media -> Stream...`
   (b) Add `TestVideo.mp4` (which you downloaded before) and press stream
   (c) Press `next`, select `UDP (legacy)` as destination and press `add`
   (d) Fill in `224.0.0.14` as the address and press `next`
   (e) Click through the rest of the dialog and start streaming
5. connect hosts `h1`, `h2`, and `h8` to the stream; for each host:
   (a) Start vlc
   (b) Click on `media -> Open Network Stream`
   (c) Fill in `udp://@224.0.0.14` as the network address and press `play`
6. You should now be receiving the stream on all three hosts!
7. Disconnect and reconnect from the stream a few times on all receiving hosts. If the app is working properly, you should be able to see the video again after reconnecting to the stream. In addition, when you stop the stream at one host, it should continue at all the other hosts.
8. Check if the flow and group entries are properly removed/modified and no multicast packets are being received at the host anymore after exiting vlc with `dump-flows` and Wireshark.

If you receive an influx of packets at the controller after exiting VLC, your app is probably not modifying all group entries properly, and a switch is forwarding traffic to another switch that was removed from the multicast tree. Remember that the initial switch connected to the source host will always keep receiving packets, even if the group has no subscribers. To prevent the controller from being flooded by these packets, you can either install a low-priority flow entry in this switch that drops all packets to the multicast group or add logic to your controller to ensure that the group entry in this switch is never removed.

Solution

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto
from ryu.lib.packet import igmp
```

```
from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link, get_host

import networkx as nx

class Multicast_Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    LOWPRIO = 1
    MEDPRIO = 2
    HIGHPRIO = 3
    HIGHESTPRIO = 4

    def __init__(self, *args, **kwargs):
        super(Multicast_Controller, self).__init__(*args, **kwargs)

        self.network = nx.DiGraph()
        self.subscribers = {}
        self.sources = {}
        self.group_ids = {}

        self.max_group_id = 0

    def log(self, message):
        self.logger.info(message)
        return

    #This function is triggered before the topology controller flows are added
    #But late enough to be able to remove flows
    @set_ev_cls(ofp_event.EventOFPStateChange, [CONFIG_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Delete any possible currently existing flows.
        del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL, out_port=ofp.OFPP_ANY,
            out_group=ofp.OFPG_ANY, command=ofp.OFPFC_DELETE)
        dp.send_msg(del_flows)

        #Delete any possible currently exising groups
        del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
            group_id=ofp.OFPG_ALL)
        dp.send_msg(del_groups)

        #Make sure deletion is finished using a barrier before additional flows are added
        barrier_req = parser.OFPBarrierRequest(dp)
        dp.send_msg(barrier_req)

    #Switch connected
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Add default flow
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)
```

```
    #Topology Events
    @set_ev_cls(event.EventSwitchEnter)
    def switchEnter(self,ev):
        switch = ev.switch

        self.network.add_node(switch.dp.id, switch = switch, groups = {}, host = False)
        self.log('Added switch ' + str(switch.dp.id))

    @set_ev_cls(event.EventSwitchLeave)
    def switchLeave(self,ev):
        switch = ev.switch
        sid = switch.dp.id

        if sid in self.network:
            #NOTE: In actual applications one should include some code to properly handle switches leaving
            #and re-entering!

            self.network.remove_node(sid)
            self.log('Removed switch ' + str(sid))

    @set_ev_cls(event.EventLinkAdd)
    def linkAdd(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        self.network.add_edge(src, dst, src_port = link.src.port_no, dst_port = link.dst.port_no, live = True)
        self.log('Added link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventLinkDelete)
    def linkDelete(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        if (src,dst) in self.network.edges():
            #NOTE: In actual applications one should include some code to properly handle link failures!

            self.network.remove_edge(src, dst)
            self.log('Removed link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventHostAdd)
    def hostFound(self,ev):
        host = ev.host
        switch = host.port.dpid
        mac = host.mac

        self._hostFound(switch, host.port.port_no, mac)

    def _hostFound(self, switch_id, port, mac):
        if mac not in self.network:
            self.network.add_node(mac, host = True)
            self.network.add_edge(mac, switch_id, src_port = -1, dst_port = port, live = True)
            self.network.add_edge(switch_id, mac, src_port = port, dst_port = -1, live = True)
            self.log('Added host ' + mac + ' at switch ' + str(switch_id))

    def remove_forwarding_rule(self, switch_id, dst_address, dst, group_id):
        """Remove/Modify flow and group entry in switch switch_id to stop forwarding packets to dst.

        Returns True iff no group entry for group_id remains after performing this operation
        Arguments:
        switch_id: id of switch to add flow to
```

```
        dst_address: IP address of the destination (i.e. the group address)
        dst: switch or host id to output packet to
        group_id: id of the all group associated with dst_address
        """

        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        port = self.network[switch_id][dst]['src_port']

        self.log('REMOVING FLOW FROM SWITCH ' + str(switch_id) + ' TO PORT '
                 + str(port))
        self.log('DESTINATION = ' + str(dst_address))

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups or port not in groups[group_id]:
            return True

        if len(groups[group_id]) == 1:
            #Group entry only forwards to port
            #So we should remove the whole group

            del groups[group_id]

            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_DELETE, ofp.OFPGT_ALL, group_id, [])
            dp.send_msg(cmd)

            return True

        groups[group_id].discard(port)

        action_lists = [[parser.OFPActionOutput(port)] for port in groups[group_id]]
        buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
            for action_list in action_lists]
        cmd = parser.OFPGroupMod(dp, ofp.OFPGC_MODIFY, ofp.OFPGT_ALL, group_id, buckets)
        dp.send_msg(cmd)

        return False

    def install_forwarding_rule(self, switch_id, dst_address, dst, group_id):
        """Install/Modify flow and group entry in switch switch_id to forward packets to dst.

        Arguments:
        switch_id: id of switch to add flow to
        dst_address: IP address of the destination (i.e. the group address)
        dst: switch or host id to output packet to
        group_id: id of the all group associated with dst_address
        """

        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        port = self.network[switch_id][dst]['src_port']

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups:
            self.install_group(switch_id, dst_address, [dst], group_id)

        elif port not in groups[group_id]:
            groups[group_id].add(port)

            action_lists = [[parser.OFPActionOutput(port)] for port in groups[group_id]]
```

```
            buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
                for action_list in action_lists]
            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_MODIFY, ofp.OFPGT_ALL, group_id, buckets)
            dp.send_msg(cmd)

        self.log('ADDED FLOW FROM SWITCH ' + str(switch_id) + ' TO PORT '
                    + str(port))
        self.log('DESTINATION = ' + str(dst_address))

    def install_group(self, switch_id, dst_address, dsts, group_id):
        """Iff group group_id has not yet been installed at switch switch_id,
        install flow and group entry in switch switch_id that forwards packets to all dsts.

        Arguments:
        switch_id: id of switch to add flow to
        dst_address: IP address of the destination (i.e. the group address)
        dsts: switch or host ids to output packet to
        group_id: id of the all group associated with dst_address
        """
        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        ports = [self.network[switch_id][dst]['src_port'] for dst in dsts]

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups:
            #Add group entry
            action_lists = [[parser.OFPActionOutput(port)] for port in ports]
            buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
                for action_list in action_lists]
            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_ADD, ofp.OFPGT_ALL, group_id, buckets)
            dp.send_msg(cmd)

            groups[group_id] = set(ports)

            #Add flow entry pointing to group
            match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=dst_address)
            actions = [parser.OFPActionGroup(group_id)]
            instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
            cmd = parser.OFPFlowMod(datapath=dp, priority=self.MEDPRIO,
                match=match, instructions=instr)
            dp.send_msg(cmd)

        self.log('INSTALLED GROUP' + str(group_id) + ' AT SWITCH ' + str(switch_id))

    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        pkt = packet.Packet(msg.data)
        eth = pkt[0]

        if eth.protocol_name != 'ethernet':
            #Ignore non-ethernet packets
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return
```

```
        self.log('Received ethernet packet')
        src = eth.src
        dst = eth.dst
        self.log('From ' + src + ' to ' + dst)

        #host detection
        if src not in self.network:
            self._hostFound(dp.id, msg.match['in_port'], src)

        if eth.ethertype == ether_types.ETH_TYPE_IP and self.isMulticast(eth.dst):
            self.process_multicast(dp, pkt)
            return

    def process_multicast(self, dp, pkt):
        eth_pkt = pkt[0]
        ip_pkt = pkt[1]

        #IGMP message
        if ip_pkt.proto == in_proto.IPPROTO_IGMP:
            igmp_pkt = pkt[2]
            self.processIGMP(eth_pkt.src, ip_pkt.src, igmp_pkt)
            return

        if ip_pkt.protocol_name != 'ipv4':
            #ignore IPv6
            return

        ip_dst = ip_pkt.dst
        ip_src = ip_pkt.src
        eth_src = eth_pkt.src

        if (ip_dst not in self.sources
            or eth_src not in self.sources[ip_dst]):
            self.add_source(dp, eth_src, ip_dst)


    def add_source(self, dp, eth_src, multicast_address):
        if multicast_address not in self.sources:
            self.sources[multicast_address] = set()

            self.max_group_id = self.max_group_id + 1
            self.group_ids[multicast_address] = self.max_group_id

        group_id = self.group_ids[multicast_address]

        self.sources[multicast_address].add(eth_src)

        self.log('Adding source ' + eth_src + ' to multicast group ' + multicast_address)

        #With low priority, drop all packets of this multicast group
        #This prevents the controller from being flooded by packets once all subscribers leave the group
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=multicast_address)
        actions = []
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=self.LOWPRIO,
            match=match, instructions=instr)
        dp.send_msg(cmd)

        if multicast_address not in self.subscribers:
```

```
            #no subscribers
            return

        next_hops = {}

        for subscriber in self.subscribers[multicast_address]:
            try:
                path = nx.shortest_path(self.network, source=eth_src, target=subscriber)

                for i in range(1,len(path)-1):
                    switch = path[i]

                    if switch not in next_hops:
                        next_hops[switch] = set()

                    next_hops[switch].add(path[i+1])

            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(eth_src) + ' to ' + subscriber)

        for switch in next_hops:
            #If the group entry has already been installed, we do not need to modify anything
            #So we can safely use this method
            self.install_group(switch, multicast_address, next_hops[switch], group_id)


    def add_subscriber(self, eth_address, multicast_address):
        if multicast_address not in self.subscribers:
            self.subscribers[multicast_address] = set()

        self.subscribers[multicast_address].add(eth_address)

        self.log('Adding ' + eth_address)

        if multicast_address not in self.sources:
            #no sources
            return

        group_id = self.group_ids[multicast_address]

        for source in self.sources[multicast_address]:
            try:
                path = nx.shortest_path(self.network, source=source, target=eth_address)

                for i in range(1,len(path)-1):
                    switch = path[i]
                    self.install_forwarding_rule(switch, multicast_address, path[i+1], group_id)
            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(source) + ' to ' + eth_address)

    def remove_subscriber(self, eth_address, multicast_address):
        self.log('Removing ' + eth_address)

        if multicast_address not in self.subscribers or eth_address not in self.subscribers[multicast_address]:
            return

        self.subscribers[multicast_address].discard(eth_address)

        if multicast_address not in self.sources:
            #no sources
            return
```

```
        group_id = self.group_ids[multicast_address]

        for source in self.sources[multicast_address]:
            try:
                #this is exactly the path we also used to route traffic to the subscriber earlier
                #except if the network has changed in the meantime (but we assume the network is static)
                path = nx.shortest_path(self.network, source=source, target=eth_address)


                for i in range(1,len(path)-1):
                    switch = path[-i-1]

                    if not self.remove_forwarding_rule(switch, multicast_address, path[-i], group_id):
                        #If the switch still routes other traffic for this group,
                        #then we should not remove the rest of the path
                        break;

            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(source) + ' to ' + eth_address)

    def processIGMP(self, eth_src, ip_src, igmp_pkt):
        self.log('IGMP packet')

        #Only support IGMPV3 membership records
        if igmp_pkt.protocol_name != 'igmpv3_report':
            return

        records = igmp_pkt.records
        for record in records:
            if not record.srcs:
                address = record.address
                self.log('Record change for group ' + address)

                if record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE:
                    self.add_subscriber(eth_src, address)

                elif record.type_ == igmp.CHANGE_TO_INCLUDE_MODE:
                    self.remove_subscriber(eth_src, address)


    def isMulticast(self, eth_dst):
        bit = eth_dst[1]
        return int(bit, 16) % 2 == 1
```

Chapter 5

Lecture 5: P4

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/ftTe75WFx7n2JYk` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_5` in which you can do the exercises.

Solution

The solution files are available here: `https://surfdrive.surf.nl/files/index.php/s/ZoIoAyV7oNZsK1t`

5.1 Acknowledgement

This exercise is derived from the official P4 tutorial Github repository.

5.2 Intro to P4 Programming Language

Programming Packet-independent Packet Processing (P4) is a domain-specific language used for network devices to specify the data plane functionalities inside networks.

This chapter provides a brief introduction on how to use the P4 programming language to achieve simple network functionalities. It also gives you several exercises to practice the use of the language and to get familiar with the wide variety of functionalities you can implement. The following exercises are included in this chapter:

1. Basic Forwarding and Tunneling
2. In-band Network Telemetry
3. Stateful Network Function

5.2.1 Dependencies and exercise folder layout

The provided VM has all the dependencies installed to be able to run these exercises. If you are not using the VM we provided, you must install the following:

- P4 software switch (`https://github.com/p4lang/behavioral-model`) and all its dependencies
- P4 compiler (p4c, `https://github.com/p4lang/p4c`) and all its dependencies

For installation of all the dependencies, we recommend using the following GitHub repository that provides many helper scripts for installation `https://github.com/jafingerhut/p4-guide.git`.

The exercise for this week contains two folders, one called `utils` with helper functions for configuring the network and the programmable switches; and one called `exercises` that contains three exercise folders that correspond to each subsection of this exercise, namely, `basic_forwarding_and_tunneling`, `in_band_network_telemetry`, and `firewall`.

5.2.2 Architecture and Switch Binary

In this exercise, we will use the v1model architecture included inside the p4c compiler and the BMv2 software switch. More information on this architecture can be found at v1model and simple_switch. To be more precise, the `simple_switch_grpc` will be used, and is developed upon `simple_switch` such that it supports the P4Runtime API.

Generally, to write a P4 program with the v1model, you will need to implement the following six programmable blocks:

1. Parser
2. VerifyChecksum
3. Ingress
4. Egress
5. ComputeChecksum
6. Deparser

Furthermore, you will need to define all the metadata and header fields that you will use throughout your program at the beginning. Thus, the general outline of your program should look like this, and more details of what should be inside each code block will be given in the latter part of this document:

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

//*********************** H E A D E R S ***************************************************************
//header definitions and metadata

struct metadata {
}

struct headers {
}

//************************ P A R S E R ***************************************************************
parser MyParser(packet_in packet, out headers hdr, inout metadata meta,
                inout standard_metadata_t standard_metadata) {
}

//************ 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 {}
}

//************** 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) {
    apply {}
}

//*************** 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 {}
}

//************* 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 {}
}

//************************ D E P A R S E R ***************************************************************
control MyDeparser(packet_out packet, in headers hdr) {
    apply {}
}


V1Switch(
MyParser(),
MyVerifyChecksum(),
```

```
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

5.3 Exercise 1. - P4 Basic Forwarding and Tunneling

In this exercise, you will first implement basic layer-3 forwarding and then add tunneling functionality between two hosts. Since this is your first P4 program, we will guide you step by step. The file `basic.p4` contains the base code file that we provide for you to complete this exercise.

5.3.1 Basic Forwarding

Topology

This section explains a very simple topology used for this exercise that contains three different hosts and one switch. Figure 5.1 depicts the topology. Your task is to enable basic forwarding between $h1$, $h2$, and $h3$, as well as tunneling between $h1$ and $h3$ such that messages sent with the tunneling identifier are tunneled to $h3$.

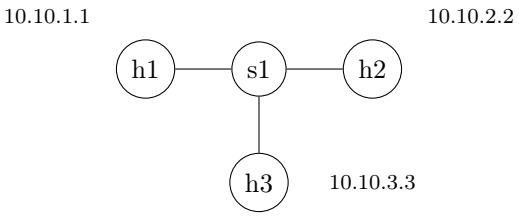


Figure 5.1: Topology - Exercise 1

Basic Forwarding

For this part of the exercise, you will be required to implement basic forwarding among the hosts, and we will guide you through the entire program for creating your very first P4 program.

Header definitions

The first step in writing a P4 program is to describe the headers that the switch will extract from an incoming packet. For our basic forwarding program, we only need to define Ethernet and IP as follows:

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> 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;
    bit<32> srcAddr;
    bit<32> dstAddr;
}
```

Each line contains a field name preceded by the field length in bits.

Additionally, to be able to use the previously defined headers, we must populate the struct "headers" with an instance of each previously defined header:

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

Metadata

Metadata contains local information about the packet that is not normally stored by the packet headers (e.g., ingress port, egress port, or a timestamp). Metadata is defined in a similar way as headers by creating a struct:

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

However, in this exercise, we will not need any custom metadata. Consequently, you can leave this struct empty. However, we will use the metadata supplied by the switch architecture (called `standard_metadata`). **Note that we don't need to explicitly define the standard metadata, since it is already defined inside the v1model**.

```
struct standard_metadata_t {

  bit<9> ingress_port;
  bit<9> egress_spec;
  bit<9> egress_port;
  bit<32> instance_type;
  bit<32> packet_length;
    .
    .
    .
}
```

The `standard_metadata` structure does not need to be defined in the P4 program, as it is implicitly defined and created at each packet reception. Out of all of these, while defining our custom processing blocks (e.g., actions, tables), we will only use the `egress_spec` field, defining which output port a packet will go to. More information on each of these fields can be found inside the v1model file located at `https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4`, and in the simple switch documentation `https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md`.

Parser

The next step in writing a P4 program is to define a parser. A parser defines the order in which the previously defined headers can be found in the packet. For example, consider the code provided below. First, each packet's Ethernet header is parsed (start state). As an output, the switch populates the Ethernet fields according to the bits specified in the header: `dstAddr`, `srcAddr`, and `etherType`. Next, depending on the value of `etherType`, the packet either exits the parser block (default accept) or continues to the next state (`parse_ipv4`). For this exercise, there is no transition after `parse_ipv4`.

```
parser MyParser(packet_in packet, out headers hdr, inout metadata meta,
                inout standard_metadata_t standard_metadata) {
        state start {
            packet.extract(hdr.ethernet);
            transition select(hdr.ethernet.etherType) {
                /* TODO: add logic for state transition */
                default: accept;
            }
        }
        state parse_ipv4{
            /* TODO: add logic for parsing ip header */
        }
    }
}
```

Exercise: Fill in the missing parts yourself (the state transition and the `parse_ipv4`).

Solution

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

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            0x0800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol){
            default: accept;
        }
    }

}
```

Verifying the Checksum

The VerifyChecksum block is executed after the previously defined parser and just before the Ingress control begins. In this block, you can choose to check the received packets' Checksums or leave the block empty (ignoring any potential problems). For example, using the code below, we can verify the checksum of the IPv4 packets.

```
control MyVerifyChecksum(inout headers hdr,

                         inout metadata meta) {
    apply {
        verify_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);
    }
}
```

However, in the v1model architecture, the switch will not automatically drop a packet having an incorrect checksum. Instead, it will update the value of the `checksum_error` field (in the standard metadata) to 1. Thus, you as the developer can specify the way the switch will handle these packets. However, since we do not intend to handle these packets differently, you can leave this block empty.

Ingress Control Flow

The next step to write a P4 program is to define the ingress control flow. The `ingress` block specifies all tables applied on a packet after the network node receives it (e.g. filter, determine the output port). Here is an overview of the function block that takes in a *header*, a *metadata* field and a *standard_metadata* field.

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

```
    apply {

    }
}
```

In the ingress control flow, we can also define match-action tables and control actions needed for the correct operation. These will be further explained in the next two subsections.

Match-Action tables

Similarly to OpenFlow flow entries, P4 tables associate lookup keys and actions. Lookup keys are any combination of header fields and metadata to be examined to find a match. Each match's outcome is an action that specifies operations to be performed on the packet. For example, the next code is a longest prefix-matching table that is based on the destination IP address parsed in the header. There are three match kinds defined by the P4 core library, namely, `exact`, `ternary`, and `lpm`.

```
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }
```

Actions

We defined two actions that can be the outcome of the lookup process: `drop_packet` and `ipv4_forward`. As the next step, we need to decide what exactly each of these actions will do. This can be done as follows:

```
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port){
        // TODO: fill this part for basic forwarding
    }
```

In P4, actions can only contain other actions (including some default primitive actions defined by the architecture). For more information on what statements you can have inside your actions, look at the P4 specification (`https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf`) and the v1model architecture: `https://raw.githubusercontent.com/p4lang/p4c/master/p4include/v1model.p4`

Exercise: Complete the `ipv4_forward` action yourself. You need to complete the following four steps:

1. Set the egress port for the next hop. Hint: the `egress_spec` field of the `standard_metadata` controls the egress port of the packet. **Notte:** You might want to refer to the `standard_metadata` struct.
2. Update the Ethernet destination address with the address of the next hop.
3. Update the Ethernet source address with the address of the switch.
4. Decrement the TTL.

Solution

```
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action ipv4_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;
    }
```

Now that you have completed the actions and match-action table part of the Ingress control block, you can proceed to finish the apply block inside the ingress control block. You would need to determine when to apply the match action table that you've created earlier.

Exercise: Complete the Ingress control block yourself. **Hint:** You only need to apply the match-action table when the IP header is valid and you can use the library function provided by the P4 core library, `isValid()`.

Solution

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

    //the previously defined tables and actions should be here!!

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

Egress control flow

In contrast to the ingress block, the `egress` block can be used to specify all tables applied at the output port before the packet is sent out (e.g., rewrite the MAC address). However, since this is a really simple program and no further actions are required to be applied on outgoing packets, this block can remain empty.

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

Updating the Checksum

Similarly to the verify checksum block, you can choose to update any of the checksums at the end of the packet processing (before the deparser). Thus, if your program modified any of the IPv4 header fields, you can update the checksum so that the following switch in the path or the end host would not potentially drop it, using the code below:

```
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);
    }
}
```

Since our firewall will not update any IP header fields, the packet's checksum field will contain the correct value. Consequently, this block should be left empty to simplify the program run at the switch.

Deparser

The deparser is the last part of our P4 program, we will add the code of the block that performs the exact opposite role of the parser. This block defines the order of the headers in the packet after it is sent. This block is restricted to emit methods only and a sequence of calls to the this method.

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        //Fill in this part yourself
    }
}
```

Exercise: Fill in the missing parts yourself. **Hint:** Since we only process the Ethernet and IP headers, we only need to emit those.

Solution

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

Running and testing your code

Congratulations! You have just completed your very first P4 program that enabled basic IP-layer forwarding. Now you can compile your program and test it in the Mininet environment. We've provided you with the `util` code and a `Makefile` for testing to be easier. You can compile your P4 code and start Mininet with the topology specified within the `pod-topo` folder with the following command:

```
make run
```

This will compile your p4 code and start the Mininet prompt, and you should now be able to test using a command such as:

```
mininet> h1 ping h2
```

You should be able to see ping messages being able to traverse among the three different hosts.

5.3.2 Basic Tunneling

This basic tunneling exercise develops based on the basic forwarding program that you have just completed earlier, and utilizes the same topology as the previous part. It allows a tunneling connection to $h3$, such that all the packets sent with the tunneling identified will be tunneled to $h3$.

Below we will guide you toward completing the tunneling functionality, but with less hints than the previous part.

Header modification

For implementing the tunneling functionality, the first step is to modify the header declaration and add tunneling headers to it such that there exists a way of identifying which packets need to be tunneled within the network. You can add the following header declarations to your header part of the program, and also modify the header struct such that tunnel header can be parsed correctly. We also recommend you to define a constant number for the type for tunneling messages.

Moreover, we use a 16-bit protocol id to indicate the type of packets being encapsulated within the tunnel packet. In the scope of this exercise, we only support ipv4 packets so you only need to handle ipv4 packets being encapsulated while implementing the parser.

```
const bit<16> myTunnel_id = 0x1212

header myTunnel_t {
    bit<16> proto_id;
    bit<16> dst_id;
}
```

Since tunneling packets and regular IP packets are different from each other, we can utilize the `etherType` field within the Ethernet header. Therefore, the size of the tunnel identifier can also be 16 bits.

Also modify the previously defined header struct to be like the following, such that the header has 3 different members:

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

Updating the Parser

The current parser does not support parsing the tunnel header. In order to incorporate with the new header type, the parser needs to be changed accordingly. You need to do the following:

1. Add transition in `parse_ethernet` based on the `etherType`. In this exercise, we'll use a special identifier `0x1212` (16 bits) for identifying tunneling packet.
2. Add a new parser state to extract the `myTunnel` header and transition based on the protocol id of the tunnel header to either `parse_ipv4` state or `accept`.

Exercise: Update the parser to be able to parse Tunnel header.

Solution

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

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            0x800: parse_ipv4;
            TYPE_MYTUNNEL: parse_myTunnel;
            default: accept;
        }
    }
    state parse_myTunnel{
        packet.extract(hdr.myTunnel);
        transition select(hdr.myTunnel.proto_id){
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4{
        packet.extract(hdr.ipv4);
        transition accept;
    }
```

Updating the Ingress

In the basic forwarding program, the packets will be forwarded based on the longest prefix-match rule to different ports. Now we need to take a different action, such that when the destination id matches correctly, the tunneling packets need to be all forwarded to a specific port. The following needs to be implemented:

1. Add an action called `myTunnel_forward` with parameter of `egressSpec_t` type called `port` to set the egress port.
2. Add a tunneling match-action table called `myTunnel_exact` and apply action called `myTunnel_forward` to forward the packet when the `dst_id` matches exactly.
3. Apply the tunneling table for tunneling packets and make sure normal IP packets are processed only by the IPv4 match-action table.

Update Deparser

When sending out the packet don't forget to emit the `myTunnel` header back to the packet.

Exercise: Update the Ingress block based on the previous description.

Solution

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

    action ipv4_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 myTunnel_forward(egressSpec_t port){
        standard_metadata.egress_spec = port;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    table myTunnel_exact{
        key = {
            hdr.myTunnel.dst_id: exact;
        }
        actions = {
            myTunnel_forward;
            NoAction;
        }
        size = 1024;
        default_action = NoAction;
    }

    apply {
         if(hdr.ipv4.isValid() && !hdr.myTunnel.isValid()){
            ipv4_lpm.apply();
         }
         if(hdr.myTunnel.isValid()){
             myTunnel_exact.apply();
         }
    }
}

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

```
        packet.emit(hdr.myTunnel);
        packet.emit(hdr.ipv4);
    }
}
```

Running and testing your solution

1. Run the following to compile and start a Mininet instance with the same topology as the previous `basic_forwarding` exercise.

```
make tunneling
```

2. You should see a Mininet prompt. Open three xTerm for `h1`, `h2` and `h3`:

```
mininet> xterm h1 h2 h3
```

3. On both `h2` and `h3`'s terminals start the listening server by:

```
./receive.py
```

4. First try testing the data without tunneling by sending a regular message from `h1` to `h2`, you should observe at `h2` that it did receive the message:

```
./send.py 10.0.2.2 "I␣love␣HPDN"
```

5. Then trying sending another message from `h1` to `h2`. However, this time we also include the tunneling functionality. You should be able to observe that even when the packet is sending for `h2`, the packet will actually received by `h3`.

```
./send.py 10.0.2.2 "I␣love␣HPDN␣;D" −−dst_id 3
```

5.4 Exercise 2. - P4 In-band Network Telemetry

Traditional SDN monitoring tools and protocols usually depend on a remote server to operate, which makes it hard to provide lower-than-millisecond accuracy and per-packet operation in the network. However, with the emergence of P4, this becomes feasible. In-band Network Telemetry has been a key application for programmable switches. In this exercise, you will implement the following functionality for `Explicit Congestion Notification`.

For this exercise, we are using the topology you can see below. **Note:** We have limited the bandwidth of the link between s1 and s2 to 512kbps in `topology.json`:

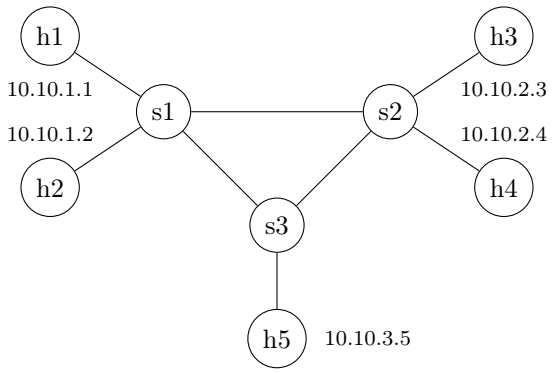


Figure 5.2: Topology - INT exercise.

There are basic templates for this exercise in the folder `inband_network_telemetry`

5.4.1 Explicit Congestion Notification

The objective of the explicit congestion notification is to extend the basic forwarding and allow end-to-end notification of network congestion without packets being dropped. As in the previous exercise, we have already handled and provided you with the control plane routing rules such that you only need to implement the data plane rules.

In order to achieve explicit congestion notification, the network utilizes the outdated `type of service (TOS)` field inside the IPv4 header and splits it into `DiffServ` and `ECN` fields. If the end host supports explicit congestion notification, the host set the value of 1 or 2 in the `ipv4.ecn` field to allow the network device to identify the data. Once congestion is detected, the network device can modify this value in the packet to notify the receiver of the packets. The receiver then copies and sends the same value to the original sender for it to lower the transmitting rate.

You will be developing the functionality described above using the `int.p4` base code that we provided to you which only supports L3-level forwarding at the moment.

Updating the headers

The very first step of completing this exercise is to update the IPv4 header definition and split the `tos` field into `diffServ` and `ecn` fields that are 6 bits and 2 bits correspondingly.

For simplicity of the program, we have defined a constant number at the beginning of the program as the threshold of the ECN that will be used later when comparing the queue length. The size of the number is chosen based on the size of the queue that is specified in the `standard_metadata` provided by the `v1model`.

```
const bit<19> ECN_threshold = 10; // NOTE: 10 being a temporary place holder
```

Exercise: Update the following IPv4 header

```
header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> tos; // TODO: split into 6 bits field of diffServ and 2 bits ecn
    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;
}
```

Solution

```
header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<6> diffServ;
    bit<2> ecn
    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;
}
```

Parser, Checksum, and Ingress

Since this application does not require any operation in these blocks, no changes are necessary within the `parser`, `verifyChecksum`, and `MyIngress`.

Updating Egress control block

In the previous explanation, we've described that we use three for indicating the network is congested and 1 or 2 for indicating that the host supports `ecn`. To correctly perform the operations needed for the egress control block, you need to implement the following:

1. Add an action to change the value of `ipv4.ecn` to 3, indicating the network is congested.
2. Apply the logic to first determine if the `ecn` value in the ipv4 header is 1 or 2, then check whether the queue length is above the threshold.
3. If the queue length is greater than the threshold, then take the action in step one.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {
        /*
         * TODO:
         * - if ecn is 1 or 2
         * - compare standard_metadata.enq_qdepth with threshold
         * and set hdr.ipv4.ecn to 3 if larger
         */
    }
}
```

Solution

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {
        action mark_ecn(){
            hdr.ipv4.ecn = 3;
        }
        apply{
            if (hdr.ipv4.ecn == 1 || hdr.ipv4.ecn == 2){
                if (standard_metadata.enq_qdepth >= ECN_THRESHOLD){
                    mark_ecn();
                }
            }
        }
    }
}
```

Updating the Checksums

Since the header of the IPv4 packet is modified, the way of computing `checksum` needs to be updated correspondingly to split the `tos` field of the IPv4 packets into `diffServ` and `ecn`.

Exercise: Update the Checksum computation.

Solution

```
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.ecn,
              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);
    }
}
```

Running and testing your solution

This section guides you on how you are able to run and test the P4 program that you've just created. For testing the `ecn`, we want to send a high rate of traffic from h2 to h3. In the meantime, we also use low-rate traffic from `h1` to `h4`. Since the `s1` to `s2` link is the bottleneck, we should be able to observe the correct `ecn` value at `h4`.

1. By running the following command, a Mininet instance with the topology specified above will be started.

   ```
   make
   ```

2. You should be seeing a Mininet prompt and you can run the following commands to start terminals for individual hosts, for example:

   ```
   mininet> xterm h1 h2 h3 h4
   ```

3. In `h4`'s terminal, start server that capture packets, and redirect the output to `h4_capture.log`. We've provide a helper file called `receive.py`:

   ```
   ./receive.py > h4_catpure.log
   ```

4. In `h3`'s terminal, start a iperf UDP server by running the following command

   ```
   iperf -s -u
   ```

5. In `h1`'s terminal, use the script that we provided, `send.py` (The script will set the `ecn` value to 1) to send one packet per second for 1 min:

   ```
   ./send.py 10.0.2.4 "I_love_HPDN" 60
   ```

6. In `h2`'s terminal, start a iperf UDP sender service by running the following command. This is to populate the network link between `s1` and `s2`.

   ```
   iperf -c 10.0.2.3 -u
   ```

7. Verify the change of `tos` value in the log file in a separate window by running:

   ```
   grep tos h4_catpure.log
   ```

Exercise: Run your solution with the instructions above and verify that your explicit congestion notification is working.

5.5 Exercise 3. - Simple P4 Firewall and Stateful P4 Firewall

In this exercise, you will need to use the knowledge you've learned so far about programmable switches and P4 to first create a simple firewall application, and secondly extend the application to become a more advanced stateful firewall application.

5.5.1 Simple P4 Firewall

In the first part of this exercise, you will write a simple P4 program that acts as a firewall that filters packets based on the IP source address and destination ports. For your implementation of the simple firewall, access should be granted only to:

- IP hosts under the 10.10.10.0/24 subnet establishing a TCP connection to the server with the IP address 10.0.2.3 using ports 80 (HTTP) or 443 (HTTPS).
- UDP packets with the destination port 53 (DNS)

There are basic templates for this exercise in the folder `firewall` of the P4 exercise folder.

Updating the headers, parser, and deparser once more

Since the firewall application is not only limited to permitting TCP connections but also UDP packets to support DNS queries, the parser and the deparser need to be updated to handle UDP data.

Exercise: Update parser and deparser to support both TCP and UDP traffic.

Solution

```
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol){
        6: parse_tcp;
        17: parse_udp;
        default: accept;
    }
}
state parse_tcp {
   packet.extract(hdr.tcp);
   transition accept;
}
state parse_udp {
   packet.extract(hdr.udp);
   transition accept;
}

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

The Ingress Control Flow

For the ingress control block, you need to define two match action tables. One will be used for TCP connections and the other for UDP packets. Then, apply the two tables you defined earlier when the header type is valid. This part of the exercise will come after you define the match-action tables.

Match-Action tables

In our firewall application, we want to filter TCP and UDP packets.

For these protocols, you need to operate on specific header fields (e.g., source IP address to allow hosts from the 10.10.10.0/24 subnet, destination TCP port to allow ports 80 and 443). These fields form a lookup key inside the match-action table. Furthermore, based on these values, the switch should either drop or forward the packet. Thus, the switch should implement two actions `drop_packet` and a `forward_packet` function. **Note:** The actions `drop_packet` and `forward_packet` have already been defined for you.

For UDP packets, we only want the UDP packets that reach port 53 for DNS queries. You need to follow the exercise instructions to complete these parts. Below is the part of the skeleton code that we provided.

```
table tcp_firewall {
    key = {
            hdr.ipv4.srcAddr: ternary;
        //Fill in this part yourself
    }
    actions = {
        drop_packet;
        forward_packet;
    }
    size = 1024;
    default_action = drop_packet();
```

```
    }

    table udp_firewall {
        //Fill in this part yourself
    }
```

Exercise: Fill in the missing parts according to the description below.

1. Update the lookup key of the `tcp_firewall` table. Keep in mind that TCP is a connection-oriented protocol, i.e., make sure that hosts are able to establish a connection.
2. Complete the table `udp_firewall` that will match the correct header fields and apply the correct actions for UDP traffic.

Solution

```
    table tcp_firewall {
        key = {
                hdr.ipv4.srcAddr: ternary;
                hdr.ipv4.dstAddr: ternary;
                hdr.tcp.srcPort: ternary;
                hdr.tcp.dstPort: ternary;
        }
        actions = {
            drop_packet;
            forward_packet;
        }
        size = 1024;
        default_action = drop_packet();
    }

    table udp_firewall {
        key = {
                hdr.udp.dstPort: exact;
        }
        actions = {
            drop_packet;
            forward_packet;
        }
        size = 1024;
        default_action = drop_packet();
    }
```

Applying the match action tables

Now that the tables are defined, these tables need to be properly applied in the ingress processing block.
Exercise: Apply the two match-action tables when the associated headers are valid.

```
if(hdr.ipv4.isValid(){
    // TODO: Check if header type is valid.
        // TODO: Apply your TCP connection table here
    // TODO: Check if header type is valid.
        // TODO: Apply your UDP table here.
}
```

Solution

```
if(hdr.ipv4.isValid(){
    if(hdr.tcp.isValid()){
        tcp_firewall.apply();
    }
    if(hdr.udp.isValid()){
        udp_firewall.apply();
    }
}
```

Egress Control Flow and Checksum

Since our firewall application does not need to apply any actions on outgoing packets or check the checksum of the packets, these blocks can be left as they are at the moment.

Control-plane rules

For SDN, the flow entries are installed by a centralized controller. Similarly, for programmable switches, P4 runtime service can be used to push control-plane rules (the table entries of match-action tables) onto the switches. If you would like to learn more about the P4 runtime services, you can read the specification at the following location: `https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html`.

In the previous parts of the P4 exercises, these rules have been provided by us, and for this part of the exercise, you need to complete these rules yourselves. Inside the `firewall` folder, the `topology.json` specifies the topology that Mininet will run, and `s1-runtime.json` specifies the runtime control plane rule for `s1` switch.
So far, we have defined the way our switch will process the incoming packets inside the P4 program. However, we have not yet associated how packets should be matched to specific actions. More specifically, what data to input into the action when a key is matched. Our next step will be to fill the match-action tables with rules that filter out any unwanted data, as specified at the start of this assignment.

Match-action table entries are filled at run-time. This can be done locally or by a centralized remote controller, depending on your use case. In this exercise, we will use statically defined rules to fill the table entries locally. You will need to complete specifying the control plane rules inside the file `s1-runtime.json`

The runtime control plane file uses JSON format for specifying the rules, you can refer to the runtime configuration file of previous P4 exercises for an example of the table entry. For example, the json format below is what you need for constructing a ternary match-kind table entry.

```
{
    "table": "table_name",
    "match":{
        "field_name": ["value", "ternary_mask"]
    },
    "action_name": "name_of_your_action",
    "action_params": {
        "name_of_the_parameter": value
    },
    "priority": non_zero_value
}
```

Note: Priority is required for the ternary match-kind. "Don't care" matches are omitted in the runtime json file.

Exercise: Finish adding all rules in `s1-runtime.json` to allow TCP traffic for IP hosts under 10.10.10.0/24 subnet establishing TCP connection to server at 10.0.2.3 using with port 80 and 443. Make sure to add all entries for both TCP, as well as UDP traffic for port 53.

Solution

```
{
    "target": "bmv2",
    "p4info": "build/firewall.p4.p4info.txt",
    "bmv2_json": "build/firewall.json",
    "table_entries":[
      {
        "table": "MyIngress.tcp_firewall",
        "default_action": true,
        "action_name": "MyIngress.drop_packet",
        "action_params": {}
      },
      {
        "table": "MyIngress.tcp_firewall",
        "match":{
            "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
            "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
            "hdr.tcp.dstPort": [80,65535]
        },
        "action_name": "MyIngress.forward_packet",
```

```
    "action_params": {
      "port" : 3,
      "dstAddr" : "08:00:00:00:02:02"
    },
    "priority": 1
  },
  {
    "table": "MyIngress.tcp_firewall",
    "match":{
      "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
      "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
      "hdr.tcp.dstPort": [443,65535]
    },
    "action_name": "MyIngress.forward_packet",
    "action_params": {
      "port" : 3,
      "dstAddr" : "08:00:00:00:02:02"
    },
    "priority": 2
  },
  {
    "table": "MyIngress.tcp_firewall",
    "match":{
      "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
      "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
      "hdr.tcp.srcPort": [80,65535]
    },
    "action_name": "MyIngress.forward_packet",
    "action_params": {
      "port" : 1
      "dstAddr" : "08:00:00:00:01:01"
    },
    "priority": 3
  },
  {
    "table": "MyIngress.tcp_firewall",
    "match":{
      "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
      "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
      "hdr.tcp.srcPort": [443,65535]
    },
    "action_name": "MyIngress.forward_packet",
    "action_params": {
      "port" : 1,
      "dstAddr" : "08:00:00:00:01:01"
    },
    "priority": 4
  },
  {
    "table": "MyIngress.tcp_firewall",
    "match":{
      "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
      "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
      "hdr.tcp.srcPort": [80,65535]
    },
    "action_name": "MyIngress.forward_packet",
    "action_params": {
      "port" : 2,
      "dstAddr" : "08:00:00:00:01:11"
    },
    "priority": 5
  },
```

```
    {
      "table": "MyIngress.tcp_firewall",
      "match":{
         "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
         "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
         "hdr.tcp.srcPort": [443,65535]
      },
      "action_name": "MyIngress.forward_packet",
      "action_params": {
         "port" : 2,
         "dstAddr" : "08:00:00:00:01:01"
       },
       "priority": 6
    },
    {
      "table": "MyIngress.udp_firewall",
      "default_action": true,
      "action_name": "MyIngress.drop",
      "action_params": {}
    },
    {
      "table": "MyIngress.udp_firewall",
      "match":{
          "hdr.udp.dstPort": 53
      },
      "action_name": "MyIngress.forward_packet",
      "action_params":{
         "port": 3,
         "dstAddr" : "08:00:00:00:02:02"
      }
    }
  ]
}
```

Testing your firewall Application

Same as previous exercises, you can run your firewall code and Mininet start with the topology specified in `topology.json`, by running the following command, and it will do the following:

```
make
```

1. Re-compile the P4 program
2. This will start a topology with three Mininet hosts (with IP addresses 10.10.10.1, 10.10.10.2, and 10.0.2.3) connected to one P4 switch that is running the previously implemented P4 firewall program (specified with the JSON flag) shown in Fig. 5.3.

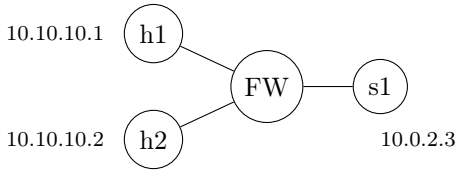


Figure 5.3: Topology - Exercise Simple firewall

3. Deploys control-plane rules specified in `s1-runtime.json` on the switches, populating the match-action tables.

Test your app as follows:

1. Start a new window for h3: `xterm h3`
2. On h3 start the netcat listener on port 80: `nc -l -p 80`

3. Start a new window for h1: `xterm h1`
4. Start a netcat client on h1: `nc 10.0.2.3 80`
5. Repeat the previous steps for different ports and IP addresses.
6. Send 5 UDP packets from h1 using the script UDPsend.py : `python UDPsend.py --dst-port 53`
7. Repeat step 6 by for different ports. Observe the output in h3 using `tcpdump -i eth0`

If the packets from steps 4 and 6 are the only packets received at `h3`, the app works as it is supposed to and you have finished your P4 application. If you also receive packets for other ports (other than 443, 80 and 53), or receive no traffic at all, the app is not functioning as it is supposed to.

`https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf` should provide you with all the information you need on writing more complex P4 applications. Additionally, the following two links will provide you with information you need to write control-plane rules using runtime_CLI and the software P4 switch you are using: `https://github.com/p4lang/behavioral-model/blob/master/docs/runtime_CLI.md` and `https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md`

Debugging options

You can use wireshark or tcpdump on host interfaces to examine the packets.

If both the switch is running correctly and the rules were added successfully, use the scripts: `TCPsend.py` and `UDPsend.py` to send custom TCP/UDP packets to the switch and observe the outcomes using tcpdump.

The Final Touch

If you are not using one single header for both TCP and UDP in the previous implementation, there is a way of combining the two headers into one. Rewrite the previous app by defining a single transport header (for both UDP and TCP)!

Hint: Analyze the header formats for TCP and UDP and create a general transport header. What UDP/TCP header fields are used by the firewall app? Where are they located? Remember that you can specify your custom header in P4.

Solution

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

//********************* H E A D E R S ***********************************************
//header definitions and metadata

header ethernet_t {
    bit<48> dstAddr;
    bit<48> 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;
    bit<32> srcAddr;
    bit<32> dstAddr;
}
```

```
header transport_t {
    bit<16> srcPort;
    bit<16> dstPort;
}

header tcp_t{
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4> dataOffset;
    bit<4> res;
    bit<1> cwr;
    bit<1> ece;
    bit<1> urg;
    bit<1> ack;
    bit<1> psh;
    bit<1> rst;
    bit<1> syn;
    bit<1> fin;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}
struct metadata {
}

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
    transport_t transport;
    tcp_t tcp;
}

//********************** 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 {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            0x0800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        packet.extract(hdr.transport);
        transition select(hdr.ipv4.protocol){
            6: parse_tcp;
            default: accpet;
        }
    }
    state parse_tcp {
       packet.extract(hdr.tcp);
       transition accept;
    }

}

//************ 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 {
```

```
        /*verify_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);*/
    }
}

//************** 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_packet(bit<9> port, bit<48> dstAddr) {
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        standard_metadata.egress_spec = port;
    }

    table firewall {
        key = {
                hdr.ipv4.srcAddr: ternary;
                hdr.ipv4.dstAddr: ternary;
                hdr.transport.srcPort: ternary;
                hdr.transport.dstPort: ternary;
                hdr.ipv4.protocol: exact;
        }
        actions = {
            drop;
            forward_packet;
        }
        size = 1024;
        default_action = drop();
    }


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

    }

}

//**************** 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 { }
}
```

```
//************* 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);*/
    }
}

//********************** 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);
        packet.emit(hdr.transport);
    }
}


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

Control-plane rules:

```
{
    "target": "bmv2",
    "p4info": "build/firewall.p4.p4info.txt",
    "bmv2_json": "build/firewall.json",
    "table_entries":[
      {
        "table": "MyIngress.firewall",
        "default_action": true,
        "action_name": "MyIngress.drop_packet",
        "action_params": {}
      },
      {
        "table": "MyIngress.firewall",
        "match":{
           "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
           "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
           "hdr.transport.dstPort": [80,65535],
           "hdr.ipv4.protocol": 6
        },
        "action_name": "MyIngress.forward_packet",
        "action_params": {
```

```
        "port" : 3,
        "dstAddr" : "08:00:00:00:02:02"
      },
      "priority": 1
    },
    {
      "table": "MyIngress.firewall",
      "match":{
        "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
        "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.transport.dstPort": [443,65535],
        "hdr.ipv4.protocol": 6
      },
      "action_name": "MyIngress.forward_packet",
      "action_params": {
        "port" : 3,
        "dstAddr" : "08:00:00:00:02:02"
      },
      "priority": 2
    },
    {
      "table": "MyIngress.firewall",
      "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6
      },
      "action_name": "MyIngress.forward_packet",
      "action_params": {
        "port" : 1
        "dstAddr" : "08:00:00:00:01:01"
      },
      "priority": 3
    },
    {
      "table": "MyIngress.firewall_",
      "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
        "hdr.transport.srcPort": [443,65535],
        "hdr.ipv4.protocol": 6
      },
      "action_name": "MyIngress.forward_packet",
      "action_params": {
        "port" : 1,
        "dstAddr" : "08:00:00:00:01:01"
      },
      "priority": 4
    },
    {
      "table": "MyIngress.firewall",
      "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6
      },
      "action_name": "MyIngress.forward_packet",
      "action_params": {
        "port" : 2,
        "dstAddr" : "08:00:00:00:01:11"
```

```
        },
        "priority": 5
      },
      {
        "table": "MyIngress.firewall",
        "match":{
            "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
            "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
            "hdr.transport.srcPort": [443,65535],
            "hdr.ipv4.protocol": 6
        },
        "action_name": "MyIngress.forward_packet",
        "action_params": {
            "port" : 2,
            "dstAddr" : "08:00:00:00:01:01"
        },
        "priority": 6
      },
      {
        "table": "MyIngress.firewall",
        "default_action": true,
        "action_name": "MyIngress.drop",
        "action_params": {}
      },
      {
        "table": "MyIngress.firewall",
        "match":{
            "hdr.transport.dstPort": 53,
            "hdr.ipv4.protocol": 17
        },
        "action_name": "MyIngress.forward_packet",
        "action_params":{
            "port": 3,
            "dstAddr" : "08:00:00:00:02:02"
        }
      }
    ]
}
```

5.5.2 Stateful Firewall

Extend your previous firewall program to allow packets to go through if they either match a known active connection or have the SYN flag set.

To complete this exercise, you will need to use two other P4 language elements, called **registers** and **hashes**.

Registers

Registers represent stateful memory. A register definition includes the length of the register cells in bits and the total number of cells. For example, a register table with 65535 rows of 4 bits can be defined as:

```
register<bit<4>>(65535) my_register;
```

Registers can be read from and written to by two specific actions:

```
my_register.read(meta.register_value, meta.index);
my_register.write(meta.index, meta.register_value)
```

Thus, updating a register always implies three actions:

1. Read the current register value into a metadata field
2. Modify the metadata value
3. Write the metadata value back in the register.

Hashes

Since this is the first time you encounter the hash function in P4, this portion of the document will demonstrate how to use a hash of some packet fields to calculate an index for a register array. The necessary steps are:

- Determining a field list you want to use for the calculation
- Defining the computation to be performed on it (the hash algorithm)
- Copying it into a user-defined metadata

For example, the code below calculates the CRC32 hash using the flow 5-tuple. Furthermore, to have the correct select value, it uses the modulo operation on the generated 32-bit hash and stores it in `meta.index` (metadata field):

```
hash(meta.index,
     HashAlgorithm.crc32, (bit<16>)0,
     {hdr.ipv4.srcAddr,
      hdr.ipv4.dstAddr,
      hdr.tcp.srcPort,
      hdr.tcp.dstPort,
      hdr.ipv4.protocol},
      (bit<32>)TABLE_WIDTH);
```

Note that the metadata header field needs to be added and the value of `ecmp_base` and `ecmp_count` are specified in the switch runtime configuration file.

Exercise

Implement a stateful firewall based on your previous simple firewall application. Note that the control plane rules also need to be changed if you modify the match-action table.

Hint: The basic idea is to use a hash of the 5-tuple and a register array to track existing TCP connections.

Test your app using the same procedure as in the simple firewall exercise. Additionally, use the script `TCPsynsend.py`. This script will send 10 packets and you can change the SYN flags options to try different scenarios.

Solution

The basic idea is to use a register array to keep track of the existing TCP connections (register `saved_connections`). To index this register array, we use a hash of the 5-tuple (source and destination IP, protocol, source, and destination port, see the ingress block). Thus, similarly to the previous exercise, our switch application first checks for every incoming TCP packet if the flow is allowed (using table `firewall`).

However, in contrast to the previous exercise, two additional actions are performed:

1. The value stored in the register array `saved_connections` is read into the metadata `is_existing`
2. For every new SYN packet, the register array `saved_connections` is updated by setting the stored value to 1. Furthermore, the metadata `is_existing` is also updated to 1.

Consequently, after a packet is processed in this table, the metadata `is_existing` will contain the information on whether the connection was established earlier.

UDP packets can be processed just the same as in the simple firewall exercise, since UDP does not save connections.

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

#define TABLE_WIDTH 256
//********************* H E A D E R S ***********************************************
//header definitions and metadata

header ethernet_t {
    bit<48> dstAddr;
    bit<48> 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;
    bit<32> srcAddr;
    bit<32> dstAddr;
}

header transport_t {
    bit<16> srcPort;
    bit<16> dstPort;
}

header tcp_t {
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4> dataOffset;
    bit<4> res;
    bit<8> flags;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

struct metadata {
    bit<1> is_existing; //metadata to store the value from the register array
    bit<32> index; //metadata field used to index the register array
}

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
    transport_t transport;
    tcp_t tcp;
}

//********************** 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 {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            0x0800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol){
            6: parse_tcp;
            17: parse_udp;
            default: accept;
```

```
        }
    }
    state parse_tcp {
        packet.extract(hdr.transport);
        packet.extract(hdr.tcp);
        transition accept;
    }
    state parse_udp {
        packet.extract(hdr.transport);
        transition accept;
    }

}

//************ 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 {
          /*verify_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);*/
    }
}

//************** 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) {

    /*register used to store existing connections
    a bit set to 1 indicates that a SYN packet for that 5-tuple was already received*/
    register<bit<1>>(65535) saved_connections;

    action drop_packet() {
        mark_to_drop(standard_metadata);
    }
    //used for UDP packets
    action forward_packet(bit<9> port,bit<48> dstAddr) {
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        standard_metadata.egress_spec = port;
            meta.is_existing = 1;
    }

    /*set the output port for the received TCP packet
    and read the current status of the connection:
    1-already established (SYN was received)
    0-not established*/
    action allow_packet(bit<9> port, bit<48> dstAddr) {
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        standard_metadata.egress_spec = port;
```

```
            /*save the value from the register connections at the index meta.index
            to the metadata meta.is_existing*/
            saved_connections.read(meta.is_existing, meta.index);
    }

    /*set the output port for the received TCP packet and
     set the current status of the connection to established (1).
     to be used if the SYN is set in the receiving packet*/
    action allow_packet_and_update(bit<9> port, bit<48> dstAddr) {
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
            standard_metadata.egress_spec = port;
        //set the value to the register connections at the index meta.index to 1
            saved_connections.write(meta.index, 1w1);
        meta.is_existing = 1;
    }


    /*table to check if the srcPort and dstPort are valid
     additionally check if the packet is a SYN:
     if the packet is a SYN - call allow_packet_and_update
     if the packet does not have the SYN flag - call allow_packet*/
    table firewall {
        key = {
                hdr.ipv4.srcAddr: ternary;
                hdr.ipv4.dstAddr: ternary;
                hdr.transport.srcPort: ternary;
                hdr.transport.dstPort: ternary;
                hdr.ipv4.protocol: exact;
                hdr.tcp.flags: ternary;
        }
        actions = {
            drop_packet;
            allow_packet;
            forward_packet;
            allow_packet_and_update;
        }
        size = 1024;
        default_action = drop_packet();
    }

    apply {

        if(hdr.ipv4.isValid()) {
        //meta.index = 5-tuple hash % 65535
            hash(meta.index,
                HashAlgorithm.crc32, (bit<16>)0,
                {hdr.ipv4.srcAddr,
                 hdr.ipv4.dstAddr,
                 hdr.transport.srcPort,
                 hdr.transport.dstPort,
                 hdr.ipv4.protocol},
                 (bit<32>)TABLE_WIDTH);

                    firewall.apply();
                if (meta.is_existing==0) drop_packet(); //drop nonexistent connections
        }
    }
}

//*************** 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 { }
}

//************* 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);*/
    }
}

//*********************** 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);
        packet.emit(hdr.transport);
        packet.emit(hdr.tcp);
    }
}


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

Control-rules that need to be added are listed here:

```
{
    "target": "bmv2",
    "p4info": "build/firewall.p4.p4info.txt",
    "bmv2_json": "build/firewall.json",
    "table_entries":[
      {
        "table": "MyIngress.firewall",
        "default_action": true,
        "action_name": "MyIngress.drop",
        "action_params": {}
      },
      {
        "table": "MyIngress.firewall",
        "match":{
           "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
```

```
        "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.transport.dstPort": [80,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [2, 7]
      },
      "action_name": "MyIngress.allow_packet_and_update",
      "action_params": {
        "port" : 3,
        "dstAddr": "08:00:00:00:02:02"
       },
       "priority": 1
    },
    {
      "table": "MyIngress.firewall",
      "match":{
        "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
        "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.transport.dstPort": [443,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [2, 7]
      },
      "action_name": "MyIngress.allow_packet_and_update",
      "action_params": {
        "port" : 3,
        "dstAddr": "08:00:00:00:02:02"
       },
       "priority": 2
    },
    {
      "table": "MyIngress.firewall",
      "match":{
        "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
        "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [0, 2]
      },
      "action_name": "MyIngress.allow_packet",
      "action_params": {
        "port" : 3,
        "dstAddr": "08:00:00:00:02:02"
       },
       "priority": 3
    },
    {
      "table": "MyIngress.firewall",
      "match":{
        "hdr.ipv4.srcAddr": ["10.10.10.0", "255.255.255.0"],
        "hdr.ipv4.dstAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.transport.srcPort": [443,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [0, 2]
      },
      "action_name": "MyIngress.allow_packet",
      "action_params": {
        "port" : 3,
        "dstAddr": "08:00:00:00:01:01"
       },
       "priority": 4
    },
    {
      "table": "MyIngress.firewall",
```

```
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [2, 7]
    },
    "action_name": "MyIngress.allow_packet_and_update",
    "action_params": {
        "port" : 1,
        "dstAddr": "08:00:00:00:01:01"
     },
     "priority": 3
  },
  {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
        "hdr.transport.srcPort": [443,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [2, 7]
    },
    "action_name": "MyIngress.allow_packet_and_update",
    "action_params": {
        "port" : 1,
        "dstAddr": "08:00:00:00:01:01"
     },
     "priority": 4
  },
  {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [2, 7]
    },
    "action_name": "MyIngress.allow_packet_and_update",
    "action_params": {
        "port" : 2,
        "dstAddr": "08:00:00:00:01:11"
     },
     "priority": 5
  },
  {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
        "hdr.transport.srcPort": [443,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [2, 7]
    },
    "action_name": "MyIngress.allow_packet_and_update",
    "action_params": {
        "port" : 2,
        "dstAddr": "08:00:00:00:01:11"
     },
     "priority": 6
  },
```

```
        {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [0, 2]
    },
    "action_name": "MyIngress.allow_packet",
    "action_params": {
        "port" : 1,
        "dstAddr": "08:00:00:00:01:01"
     },
     "priority": 3
  },
  {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.1", "255.255.255.255"],
        "hdr.transport.srcPort": [443,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [0, 2]
    },
    "action_name": "MyIngress.allow_packet",
    "action_params": {
        "port" : 1,
        "dstAddr": "08:00:00:00:01:01"
     },
     "priority": 4
  },
  {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
        "hdr.transport.srcPort": [80,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [0, 2]
    },
    "action_name": "MyIngress.allow_packet",
    "action_params": {
        "port" : 2,
        "dstAddr": "08:00:00:00:01:11"
     },
     "priority": 5
  },
  {
    "table": "MyIngress.firewall",
    "match":{
        "hdr.ipv4.srcAddr": ["10.0.2.3", "255.255.255.255"],
        "hdr.ipv4.dstAddr": ["10.10.10.2", "255.255.255.255"],
        "hdr.transport.srcPort": [443,65535],
        "hdr.ipv4.protocol": 6,
        "hdr.tcp.flags": [0, 2]
    },
    "action_name": "MyIngress.allow_packet",
    "action_params": {
        "port" : 2,
        "dstAddr": "08:00:00:00:01:11"
     },
```

```
        "priority": 6
      },
      {
        "table": "MyIngress.firewall",
        "match":{
            "hdr.transport.dstPort": 53,
            "hdr.ipv4.protocol": 17
        },
        "action_name": "MyIngress.forward_packet",
        "action_params": {
           "port": 3,
           "dstAddr": "08:00:00:00:02:02"
        },
        "priority": 7
      }
    ]
}
```

5.6 Useful links:

You can find many other examples on how to use P4 on `https://github.com/p4lang/tutorials`