

Exercises High Performance Data Networking

Jorik Oostenbrink Belma Turkovic Niels van Adrichem Fernando Kuipers

Chapter 1

Setup

Mininet¹ is a network simulator using a software implementation of Open vSwitch, a popular and open source virtual switch found in switches ranging from hardware switches sold by vendors to software switches installable on generic computer hardware. Open vSwitch implements the OpenFlow protocol, enabling the deployment of Software-Defined Networks. We will use Mininet together with the Ryu² OpenFlow controller framework to simulate Software-Defined Networks. In order to do this, you need to download and install Mininet and Ryu. In this chapter we will discuss downloading a Virtual Machine containing pre-installed versions of both.

1.1 VirtualBox

First install the open-source VirtualBox hypervisor, found here: <https://www.virtualbox.org/wiki/Downloads>.

If your PC supports hardware-assisted virtualization, make sure to enable it. This setting can usually be found under virtualization, VT-x, AMD-V or similar in your BIOS settings. Make sure to also enable it in VirtualBox itself.

1.2 Mininet VM

The Mininet VM image can be found here: <http://www.nas.ewi.tudelft.nl/tmp/Mininet-VM.ova>.

Alternatively, you can download a more up-to-date Mininet VM from <http://mininet.org/download/>.

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

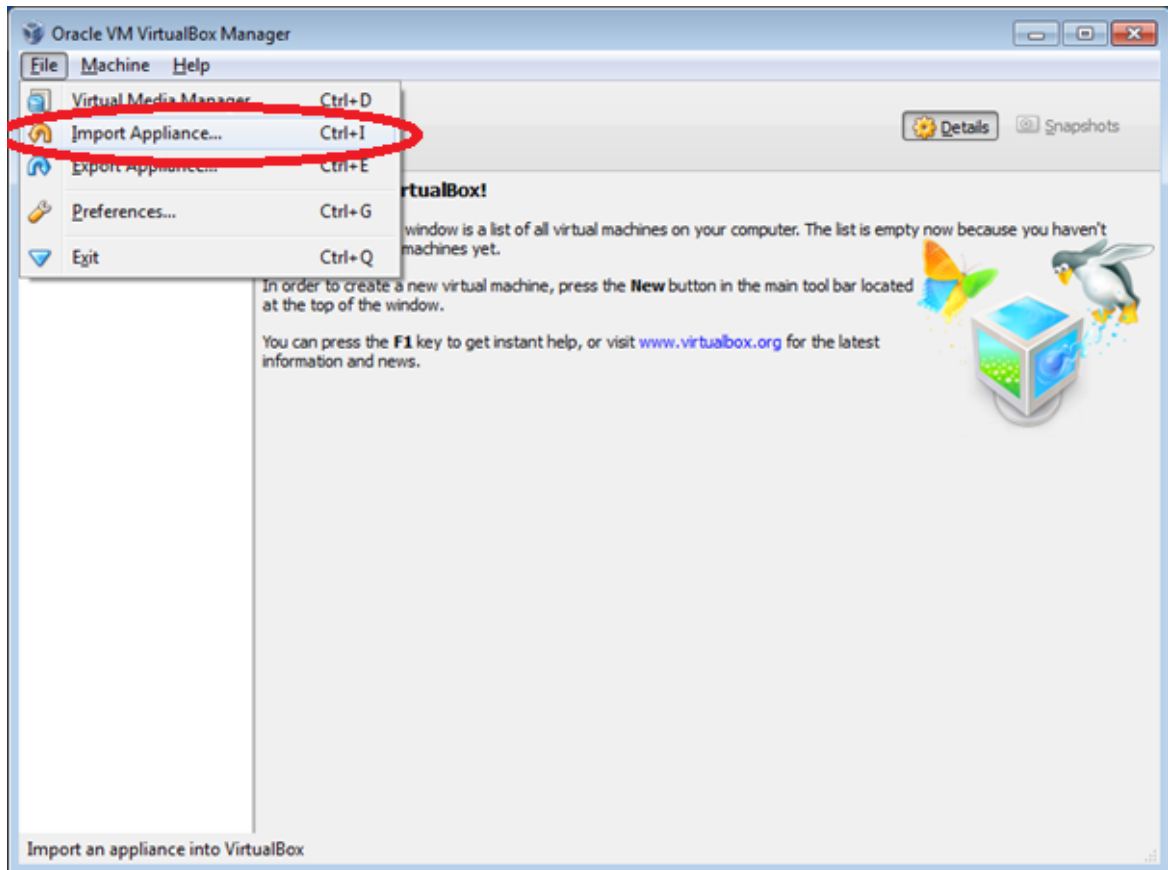
The following installation instructions are only meant as a guideline and, in case of problems, you are advised to closely follow the directions on the Mininet website (which are kept up-to-date).

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

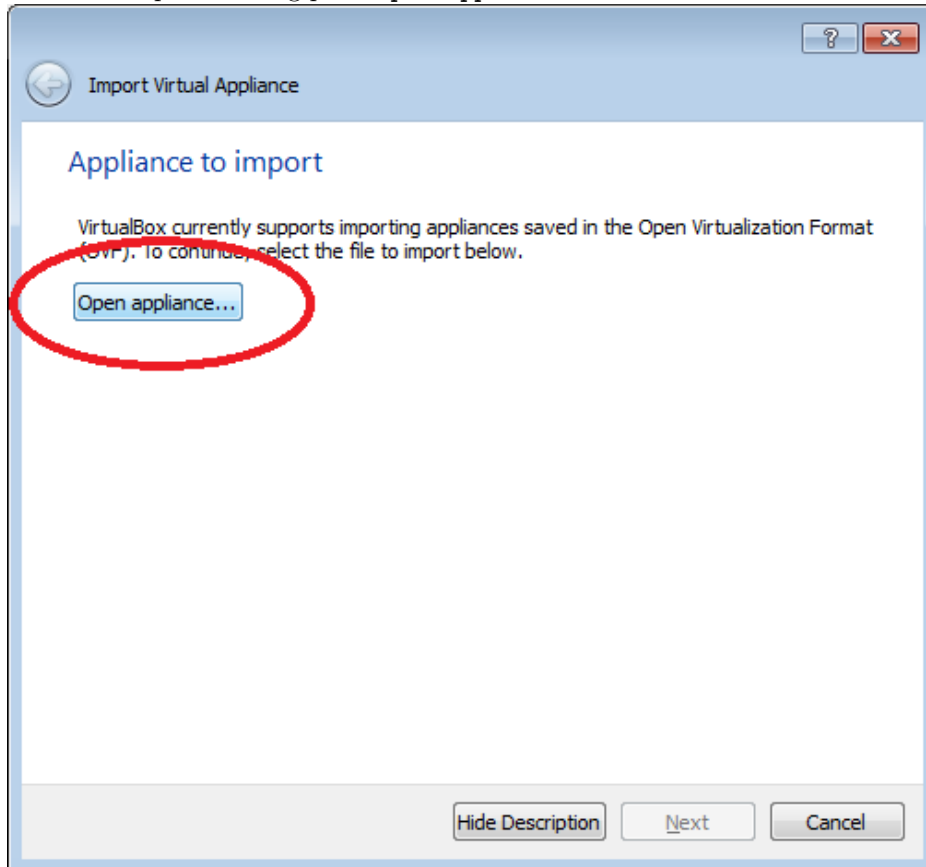
1. Import the VM by opening **File -> Import Appliance...** from the menu.

¹<http://mininet.org/>

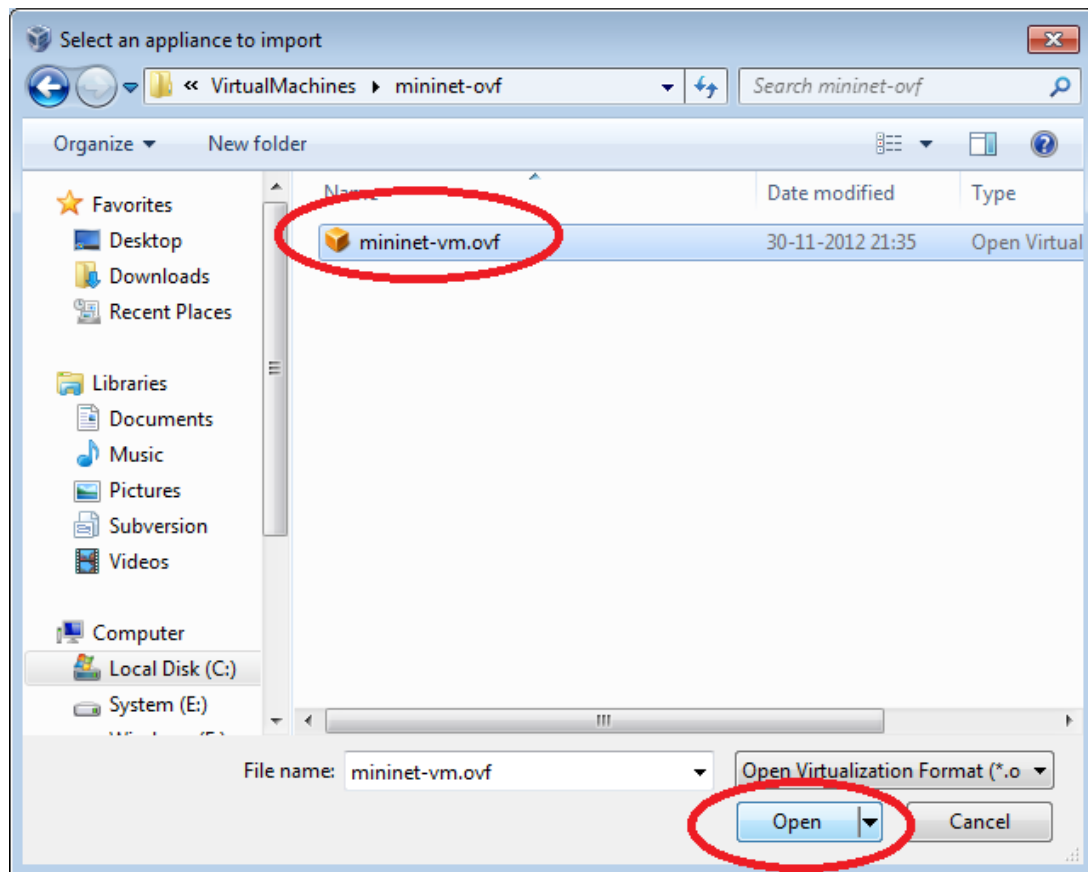
²<https://osrg.github.io/ryu/>



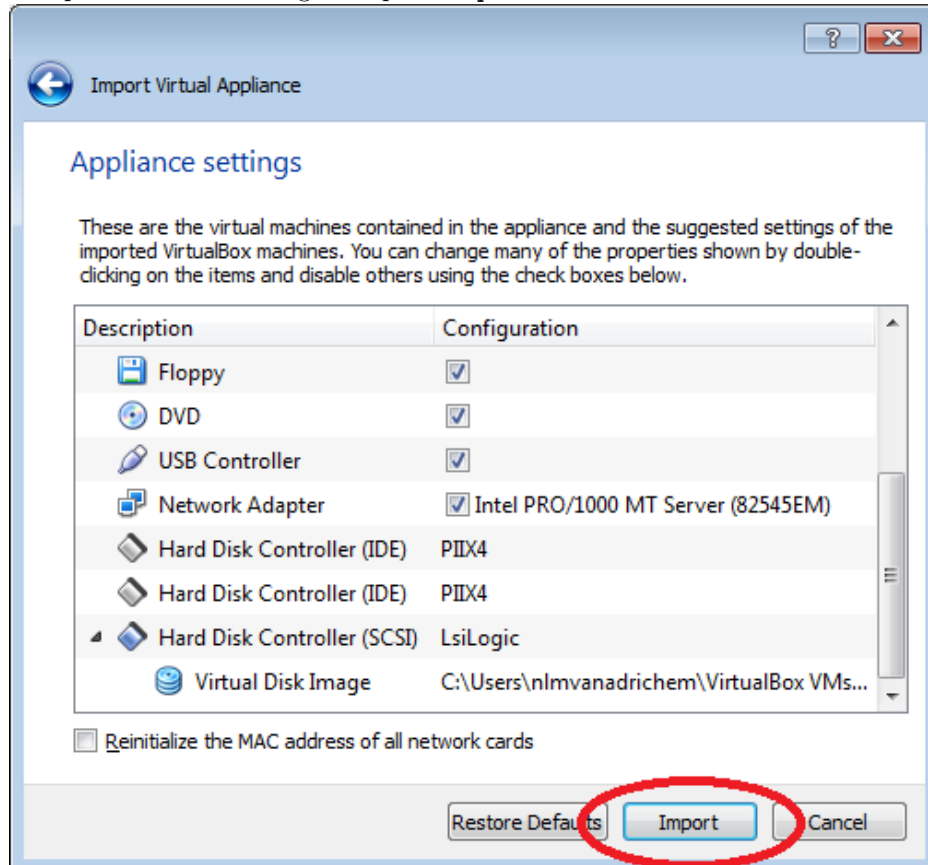
2. In the new opened dialog press `Open appliance....`



3. Select and open the file *Mininet-vm.ova* from the directory where you stored the Mininet VM.



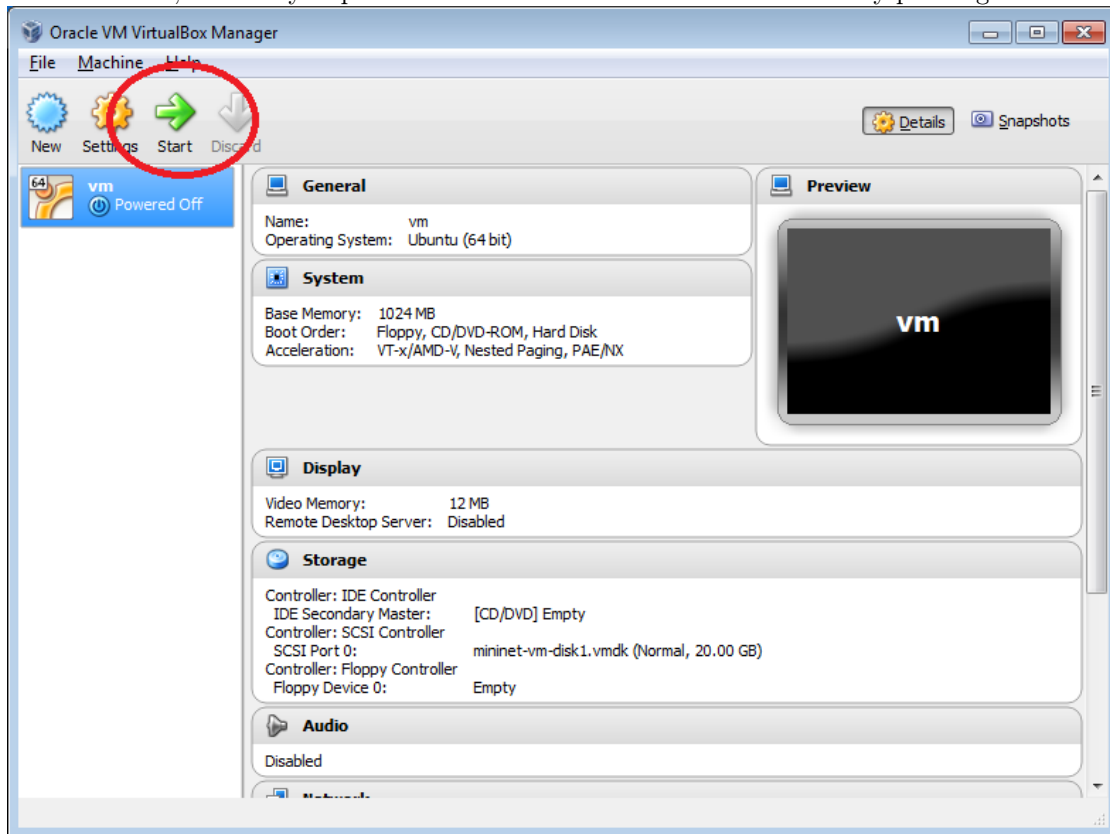
4. Accept the default settings and press Import.



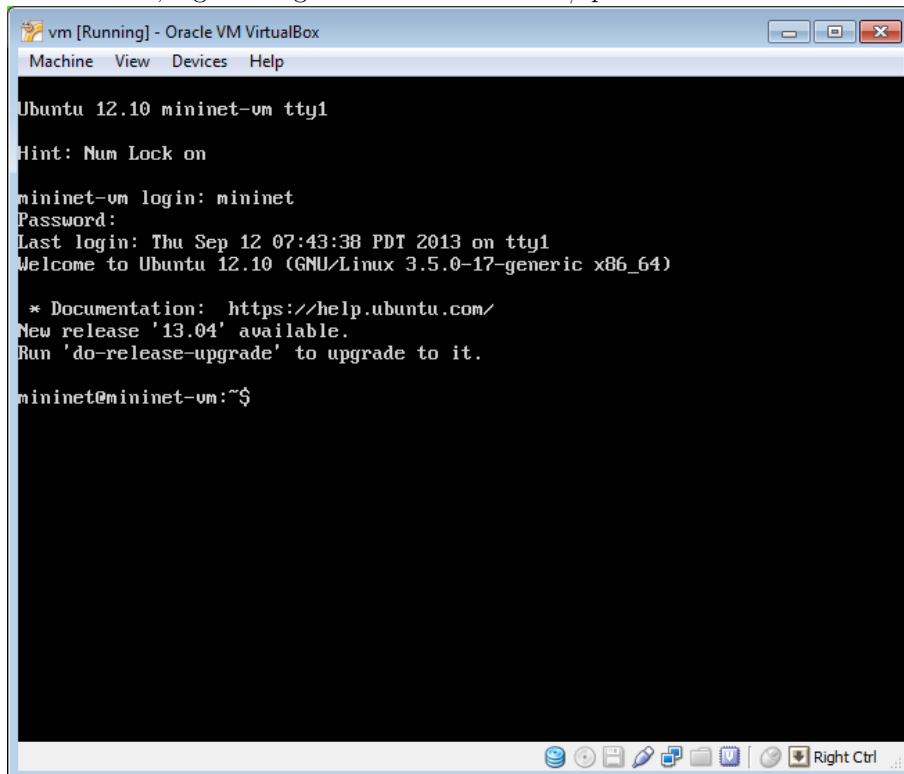
5. The VM is now being imported. This may take a couple of minutes, depending on the speed of

your computer.

- When finished, the newly imported VM is shown in VirtualBox. Start it by pressing **Start**.



- Once started, log in using the default username / password combination *mininet* / *mininet*.



- Run command, `sudo pip install networkx` to install networkx python library

Start the Desktop with the command 'startx' and open a terminal using the key combination Ctrl+Alt+T.

The Linux terminal is comparable to the Windows Command Prompt and offers a Command Line Interface in which commands can be executed.

Chapter 2

Mininet Exercises

2.1 Starting Mininet

In the terminal, start Mininet by running `sudo mn`. By default, Mininet will start a virtual network with 2 hosts `h1` and `h2` connected via OpenFlow switch `s1`. By default mininet will run a reference OF controller to control the switches.

Run `?` to view all possible commands. Feel free to play with them, e.g. run `pingall` to confirm connectivity between all hosts.

Each host has a separate network namespace. Within Mininet you can run most Linux commands directly on any of the virtual hosts by prepending it by its hostname. For example, you can run `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`. This way you can also start different programs, services, scripts etc. By appending `&` to a command the process will run in the background and the mininet window isn’t blocked.

You can emulate different topologies by adding the `--topo` option to the mininet start command. 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 single switch, `h` hosts topology. This topology can be used by appending `--topo single,h` to the `mn` command where `h` refers to the number of hosts.
- **Reversed:** Equal 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:** One switch for each host, switches are connected in a line. 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 start with randomly assigned 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 during the execution the Mininet script crashes or you terminate it (using for example `Ctrl+C`) you will need to clean the Mininet environment by typing `sudo mn -c`

2.2 Creating custom topologies

One of the advantages of Mininet is that it is possible to create complex network topologies and run experiments on them without the need to physically create those networks. It is possible to create python scripts to specify custom topologies. These custom topologies can be created using the methods `addHost`, `addSwitch` and `addLink` of the class `Topo`. You can use the `--custom` command to load your custom python script and the `--topo` to select the (custom) topology:

```
$ sudo mn --custom /path_to_your_topology/file.py --topo topology_name --test pingall
```

For example, the following python script creates a topology of 2 connected switches, adding 1 host to each switch:

```
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( 's3' )
        rightSwitch = self.addSwitch( 's4' )

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

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

The last line maps the created topology classes to topology names. These names can then be used with the `--topo` option.

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

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

If we save this file as `custom.py` we can use the following command to run the new topology:

```
$ sudo mn --custom /path_to_your_topology/custom.py --topo mytopo --test pingall
```

2.2.1 Exercise 1. - Complete graph

Make a custom python script that creates the topology shown in Fig. 2.1. Afterwards, try to adapt your

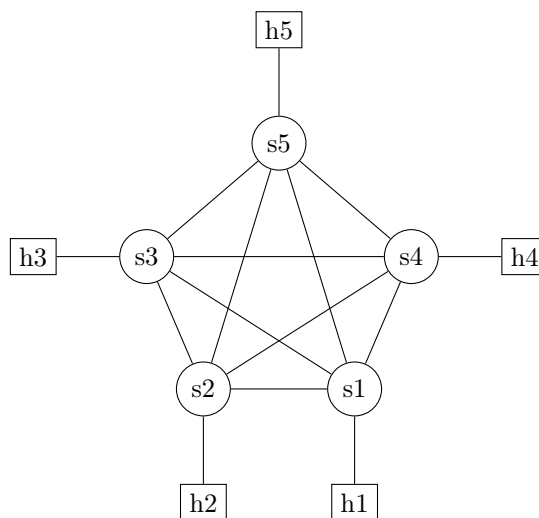


Figure 2.1: Complete graph

topology to 50 switches and hosts instead of 5.

2.2.2 Exercise 2. - Square lattice topology

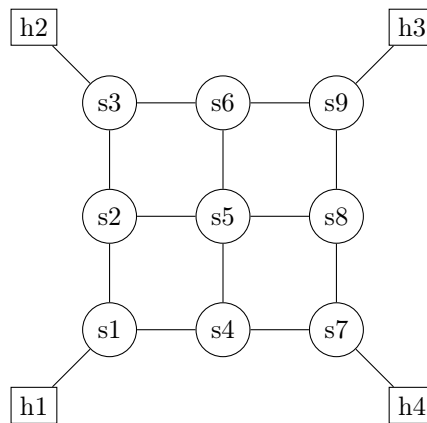


Figure 2.2: Complete graph

Create the topology from Fig. 2.2. Afterwards, try to increase the size of your custom topology to 6x6 nodes.

2.3 Introducing link properties

Mininet can also 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:

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

where bandwidth is specified in Mbps. It is possible to add these parameters in the custom python files too by specifying these options in the `addLink` function as shown below:

```
self.addLink( s1, s2, delay='5ms' , bw=20)
```

This way, you can set the parameters of all links individually. You need to add `--link tc` option when starting mininet.

2.3.1 Exercise 3. - Add link properties

Choose one of the topologies you created before and add a random link delay between 0-10ms to each link. In addition, set all link bandwidths to 22 Mbps (also in your custom topology file).

2.4 Create Mininet Scripts

It is possible to automate certain tasks in mininet, i.e. add routes to the hosts, execute python scripts or tear down links. To do so, simply create a file and put a single command on each line. These commands can then be executed in mininet with `source file`, where file is the path to your automation script.

For example, the following file first adds multicast routes to host h0, h1 and h2 and prints “routes setup”. Next, it starts a transmitter on h0 and receivers on h1 and h2 and prints “experiment started”.

```
h0 route add -net 224.0.0.0 netmask 240.0.0.0 dev h0-eth0
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
py "routes setup"
h0 python path_to_script/Transmitter.py &
h1 python path_to_script/Receiver.py &
h2 python path_to_script/Receiver.py &
py "experiment started"
```

Important note: It is very important to add `&` after calling each python script on the hosts, as this will start the processes in the background. This way, the next command will execute immediately and would not have to wait until the script is finished.

2.5 Create own 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 as sleep

def custom_sleep(self, time):
    "custom_sleep_function"
    sleep(int(time))

CLI.do_sleep = custom_sleep
```

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 “customTopology.py” and a commands file “customCommands.py”, we can load both of them with `-- custom customTopology.py, customCommands.py`.

2.5.1 Exercise 4. - Creating a mininet script

Start mininet with the linear topology with 4 switches. Then, create a script that will:

- start a ping between h1 and h4
- wait for 20 seconds
- tear down the link between switches s2 and s3 (`link s2 s3 down`)
- wait for 60 seconds
- bring the link back up
- wait for 20 seconds
- kill the ping process (`h1 kill %ping`)

Execute the script. What happens?

What changes if you change the sleep times to 1 second? Why?

2.5.2 Exercise 5. - Connecting the switches to a custom controller

After starting mininet, each OpenFlow switch in the network will be connected to the controller - which could be in the same VM or outside the VM. In order to connect your mininet environment to a specific controller you should start it with `-remote` option:

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

The downloaded VM has the Ryu controller already installed (Ryu is explained in more detail in section 4). In order to complete this exercise:

1. Open a new terminal, 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
```

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 --arp --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`.

4. Test the connectivity using the `ping all` command and check the output from the terminal that is running the controller app.

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

2.6 Useful commands

Dump information about all nodes: `dump`

Display all options : `help`

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`

Bring links up and down: `link s1 s2 down/up`

Test connectivity between two hosts: `h1 ping h2`

Test connectivity by having all nodes ping each other `pingall`

Display the ARP table of the host h1: `h1 arp`

Display the routing table of the host h1: `h1 route`

Run a simple web server on host h1: `h1 python -m SimpleHTTPServer 80 &`

Shuting down the started web serer: `h1 kill %python`

Send a request from the node h2 to the web server running on node h1: `h2 wget -O - - h1`

Chapter 3

Wireshark Exercises



Wireshark is a network protocol analyzer used to capture and store network packets. In this section we will use Wireshark to monitor the traffic between switches as well as the traffic between one of those switches and the controller.

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.

3.1 Monitoring network traffic

3.1.1 Exercise 1. - Capturing traffic between hosts

In this exercise we will show you how to monitor the traffic of a specific interface in your network.

1. Start mininet with the linear topology again. 
2. Start wireshark on host 1 (**h1 wireshark &**). Mininet always starts processes as superuser. This causes the initial warning you will see when starting wireshark, 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 topleft.
6. You can filter packets on a wide variety of properties. For example, we can filter out all packets with ip dst address 10.0.0.2 by typing in **ip.dst!=10.0.0.2**. Test this by filtering on **ip.dst!=10.0.0.2 and icmp**.
What type of traffic remains after filtering?

Monitoring the interfaces of switches can be done in the same way. For example, you can start wireshark on switch s1 with **s1 wireshark &**. The only difference is that 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.

3.1.2 Exercise 2. - Capturing traffic between switches and controller

In this exercise you will learn how to monitor the traffic between the controller and the switches.

1. Start the controller as explained in 2.5.2. The controller is going to run on the loopback interface in the VM (127.0.0.1)

2. Start wireshark in the second terminal (`sudo wireshark &`). Running this command as sudo will cause the initial warning, but you can safely ignore this warning.
3. On the left you can select any of the VM interfaces to monitor. We are only interested in loopback interface (lo), because the controller is running on that one. 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 linear topology again, but this time use the `-remote` option to connect the switches to the custom controller.
5. After starting mininet, each OpenFlow switch in the network will connect to the controller. Observe the packets captured on loopback adapter and describe the protocol handshake that occurs. In order to filter the OpenFlow traffic in Wireshark you can use `of` as the filter.
6. Open another terminal and use the `ovs-ofctl` command to find and print the flow table of the switches:

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

Are there any entries present?
7. Run the `h1 ping h2` command to generate traffic between hosts. When the first switch (s1) receives this 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.
8. Filter out the OpenFlow Flow Add messages. This can be done with the `of13.flow_add.type == 14` filter. Analyze the captured packets.
9. Check the flow tables on the switches again. What changed compared to step 6.?

Chapter 4

SDN Firewall Exercise

Ryu is a framework that can be used to implement your own OpenFlow network controller. It contains many useful components, such as topology discovery and a web GUI providing insight into network topology. Furthermore, it is shipped with prototype apps implementing various functions such as simple switches, spanning tree computation, firewalling, etc.

In this exercise we will guide you towards creating your first own Ryu app 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 blocks all traffic except TCP traffic being sent to port 80 or port 443.

4.1 Connecting to Switch

We start by creating a simple app. The app will only connect to the switches, and not install any flow entries yet.

To create a Ryu app you only need to extend the RyuApp class. Copy the following python script and save it as “simpleControllerApp.py”:

```
from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls

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

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

    #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 = [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)
```

This simple file is already a functioning Ryu app. You can run the app with

```
cd ~/ryu
PYTHONPATH=. ./bin/ryu-manager ~/path_to_app/simpleControllerApp.py
```

`OFF_VERSIONS` should be a list of all the OpenFlow protocols the app supports. In this case the app only supports OF 1.3.

The `set_ev_cls` decorator is used to tell ryu to execute certain functions during certain events. The first argument (in this case `off_event.EventOFFSwitchFeatures`) 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.

`@set_ev_cls(off_event.EventOFFSwitchFeatures, CONFIG_DISPATCHER)` tells Ryu to execute the decorated function after the controller has received 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.

Start Wireshark on the loopback interface and filter on OF packets, as in exercise 3.1.2. Next, start both the controller and a simple mininet network with a remote controller (`sudo mn --mac --controller=remote --switch ovs,protocols=OpenFlow13`). You should now see your app connecting to the switches.

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

4.2 Setting Up the Firewall

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

We can tell switches to drop unmatched packets by simply installing an empty action list instead of the current `parser.OFPActionOutput(off.OFPP_CONTROLLER, off.OFPCML_NO_BUFFER)` action, i.e.:

```
#Switch connected
@set_ev_cls(off_event.EventOFFSwitchFeatures, 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)
```

Unfortunately, now we are also dropping all HTTP(S) packets. We do want the switches to send these types of packets to the controller, so we can install flow entries to route this type of traffic through the network.

To do so, we need to install two flow entries that match both on protocol (TCP) and destination port.

We can use the `tcp_dst` match field to match on the TCP destination port. The following addition to our app will send all (otherwise unmatched) TCP packets sent to port 443 or 80 to the controller:

```
#Switch connected
@set_ev_cls(off_event.EventOFFSwitchFeatures, 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

```

To be able to use the `tcp_dst` match field, OF first requires us to match on only TCP packets (`ip_proto=0x06`), which in turn requires us to match IPv4 (or IPv6) packets with the ethernet type match field (`eth_type`).

Note that we set the priority of the new entries higher than the table-miss entry.

To process these packets in the controller, we need to add a 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 (after the negotiation has completed).

For now, we 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))

```

Thus, the first version of our basic firewall controller app will be:

```

from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
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.lib.packet import packet

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

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

    #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

#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))

```

Note how we added an import for *MAIN_DISPATCHER* and *ryu.lib.packet*. Packet is a ryu library that can be used to parse and create packets.

If you have not done so 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** will make mininet pre-populate the ARP entries of each host. This is necessary because we immediately drop all ARP packets sent to the network.

To test the firewall, first try to ping h2 from h1. Note that the controller logs no incoming packets.

Of course, TCP packets to port 443 and 80 should not be dropped, but send to the controller. To test this functionality, start a netcat client on h1:

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

This client will try to connect to a listener on port 443 of h2. Note that its messages will not even reach h2, because we do not yet route packets through the network. However, you should notice them being logged by the controller.

In order to check the entries added on the switch s1 by the app type in a new terminal:

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

4.3 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 anytime 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, but this is not the best possible event during which to ‘reset’ the flow and group tables. In section 4.4 we start using a ryu module to detect the topology of the network. This module adds a flow entry to every switch, so we need to remove all existing flow entries before this module adds these flow entries.

The *ofp_event.EventOFPSwitchFeatures*, *CONFIG_DISPATCHER* event is triggered just before *ofp_event.EventOFPSwitchFeatures*, *CONFIG_DISPATCHER*.

The following function 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.EventOFPPStateChange, 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 existing groups
    del_groups = parser.OFPGGroupMod(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 get executed before those you send after sending the barrier request.

4.4 Adding Topology Detection

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/basicFirewallApp.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 with:

```
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.

The app will need to store all relevant topology information itself. To do so, we will use NetworkX, which is a widely used python graph library.

To use this library to store our topology, we first need to import it (`import networkx as nx`) and add the following line to the constructor (`__init__(self, *args, **kwargs)`):

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

This will generate a directional graph we can use to store all relevant topology information.

We can add switches, links and hosts to this graph as follows:

```
#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))
```

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.

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 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 is 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 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.

4.5 Routing Packets

We are finally ready to actually 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 here and all possible actions here.

```
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

    #Try to finish this function yourself
```

Remember that incoming packets are sent to the `packet_in_handler`. First, we filter out all LLDP packets. We do not even log these packets, 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)
```

Remember to add `from ryu.lib.packet import ether_types` to be able to use `ether_types.ETH_TYPE_LLDP`.

Next, we need to install a path from the source switch to *dst* in the network:

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

    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
    """

    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 of switch sid

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

    self.logger.info('Outputting_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:

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)
```

4.6 Testing the App

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

Choose two hosts, one client and one server. In the rest of this section, we assume the server is h1 and the client h2.

First start a new window for h1: `xterm h1`

Start an netcat listener on port 80 of h1 with `nc -l -p 80`

Start an netcat client on h2: `h2 nc h1 80 -p 80`

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

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.

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.

Stop the server.

Next, without resetting the controller or mininet, perform the same process for ports 443 and 90. If the app has been configured correctly, you should receive traffic on port 443, but not on port 90.

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.

If you run into problems while testing your app, first try reading through the exercise again and fixing the problem by yourself. If this does not help, you can contact the TAs at `b.turkovic-2@tudelft.nl`, `j.oostenbrink@tudelft.nl`.

4.7 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.node[sid]['flows']` to store all flow entries of switch `sid`.

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 workings of OpenFlow, you can look up the official OpenFlow Switch Specification.

Chapter 5

Project

Try to come up with an app that you want to implement yourself on a network controller. Write down your goals as concise and clear as possible and send them to the lecturer for approval before December 12th. Implementing the app will form your assignment to finish the practical work of this course. Possible topics are:

1. Network Address Translation
2. Video streaming with bandwidth guarantees
3. Probing (to detect link failures)
4. Load balancing
5. Congestion detection and avoidance

However, you may deviate from the above-mentioned list of possible topics. If you want to add some additional difficulty to your project, you can try to make your app link/node fault tolerant as well.

It is also possible to do a project using the P4 language. For a simple example that combines the Mininet environment with a P4 software switch you can look at: <https://github.com/p4lang/behavioral-model>.

For more examples on what P4 can be used for, look at the examples on: <https://github.com/p4lang/tutorials/tree/master/examples>