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.

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.

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.

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?

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

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.

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.?

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?

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.

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
```

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))
```

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
```

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
```

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*.

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