ECE 4110 Internetwork Programming
Lab 8 Software Defined Networking: Mininet, OpenFlow, RYU

Date Issued: March 17, 2016
Date Due: April 7, 2016
Last Edited: March 12, 2016

Version: March 18, 2016

Caution: You will need to download a 3.3 gigabyte SDN Hub Virtual machine from **SDN Hub Tutorial VM** You will want to do that overnight well before you want to begin this lab. The file is too big to put on TSquare.

Portions of this lab are copied and modified from a TU Berlin SDN laboratory assignment as well as Mininet, OpenFlow, and RYU websites. Today we're leaving the safe realm of established network standards and looking at a new standard of tomorrow called OpenFlow. We will be investigating the basic concepts behind OpenFlow, using the Mininet network emulation tool. Using mininet, you will create network topologies, deploy a software controller, and build your own simple OpenFlow controller. You will watch a video from Professor Scott Shenker about the motivations behind OpenFlow and see how OpenFlow enables the network to become "Software Defined."

We have been working quite a bit with the managed switches/routers in our labs. Internally, those switches/routers consist of two core parts: An embedded processor running an operating system (Cisco IOS), for configuration and managing a forwarding table, and a specialized ASIC chip providing the forwarding path that actually forwards the packets from one port to another, based on some internal forwarding table. These two parts are tightly coupled together in a closed box, i.e., you cannot buy one without the other, and you can configure it, but not program it according to your wishes. If you want something that's not supported by the firmware you are very much out of luck.

Some vendors are living very well on this tight coupling between the hardware forwarding (switch box) and intelligence (firmware), but many users, for instance Telcos, and large datacenter owners, are unhappy with it. OpenFlow is a refactoring of the classical switch architecture to separate the mechanism of forwarding from the process of controlling the content of the forwarding table. Instead of an internal embedded processor in a closed OS, an OpenFlow switch exposes a forwarding table called the flow table via a standardized protocol to an external software controller, e.g., a piece of software running on a standard PC server. The controller makes the “intelligent” forwarding decisions, while the switch hardware performs the mechanism of forwarding the packets in hardware and at line rate.

OpenFlow is currently under active development and adoption. There's a mixed academy/industrial consortium which led to its development, originating at Stanford University. Today, the Open Networking Foundation is in charge of defining the future direction of the evolution of the OpenFlow protocol.

Note: The definition of a switch when talking about Software defined networking is a device that can perform switching and or routing all at the same time.

Question 1: OpenFlow Intro

Familiarize yourself with OpenFlow. Have a look at the whitepaper "OpenFlow: Enabling Innovation in College Networks" at http://archive.openflow.org/wp/documents/ Then answer the following questions:

(a) What is the function of an OpenFlow controller?
(b) Describe the difference between the data plane and the control plane in an Open Flow network. What traffic goes over the data plane? What traffic goes over the control plane?
(c) What is a flow table, what purpose does it serve? When does it get modified, and what modifies it?
(d) According to the old paper "OpenFlow: Enabling Innovation in College Networks", against which parts of an Ethernet frame can a flow table entry match in an old "Type 0" OpenFlow switch?
(e) According to http://flowgrammable.org/sdn/openflow/actions/#tab_ofp_1_0 "Actions can _____, _____, _____, or _____ the packet".
(f) What new problems might OpenFlow introduce into networks? Do you see any limitations? Pick 2 and explain.
(g) Could you build an IPv4 router using an OpenFlow v1.5 switch and an appropriate controller? Hint: Look at the supported flow-table actions for OF v1.5
https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf

Installing the SDN Hub Tutorial Virtual Machine

Source: http://sdnhub.org/tutorials/ryu/

This tutorial is intended for beginners interested in SDN application development for the RYU platform from NTT. RYU has support for several versions of OpenFlow, including OpenFlow versions 1.0 and 1.3 that have seen wide spread support from vendors.

1. Quickstart

- To get started, download and set up the **SDN Hub Tutorial VM** in Virtualbox or VMware Player. The instructions from that link are repeated here for convenience:

Jumpstart your SDN development through our all-in-one pre-built tutorial VM, built for you by SDN Hub. This is a 64-bit Ubuntu 14.04 image (3GB) that has a number of SDN software and tools installed.

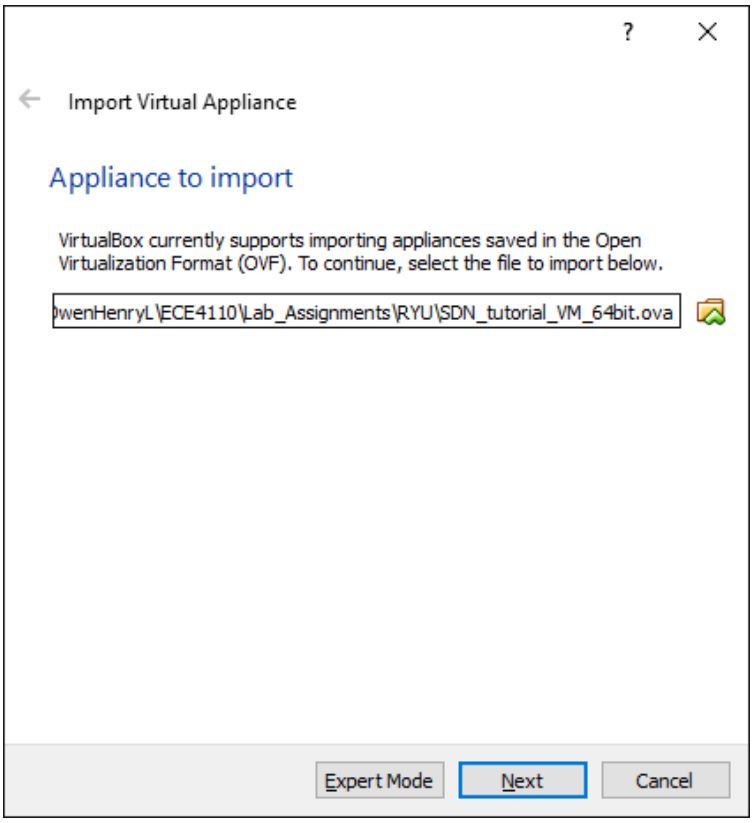
- SDN Controllers: OpenDaylight, ONOS, RYU, Floodlight, Floodlight-OF1.3, POX, and Trema
- Example code for a hub, L2 learning switch, traffic tap, and other applications
- Open vSwitch 2.3.0 with support for Openflow 1.2, 1.3 and 1.4, and LINC switch
- Mininet to create and run example topologies
- Pyretic
- Wireshark 1.12.1 with native support for OpenFlow parsing
- JDK 1.8, Eclipse Luna, and Maven 3.3.3

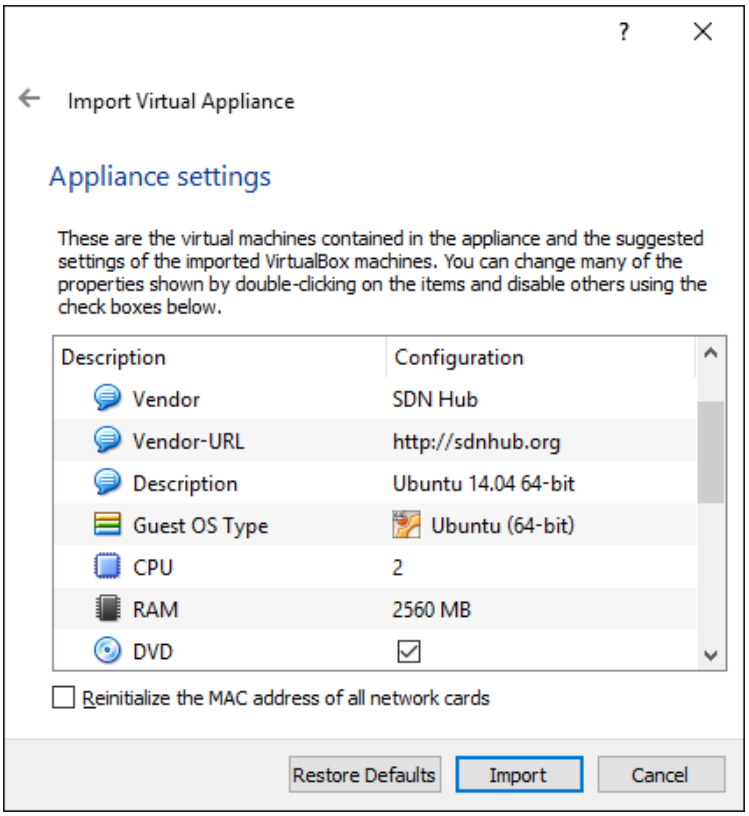
Download

1. You can directly download the OVA file from our file server: [64-bit | 32-bit]

Instructions to use the VM

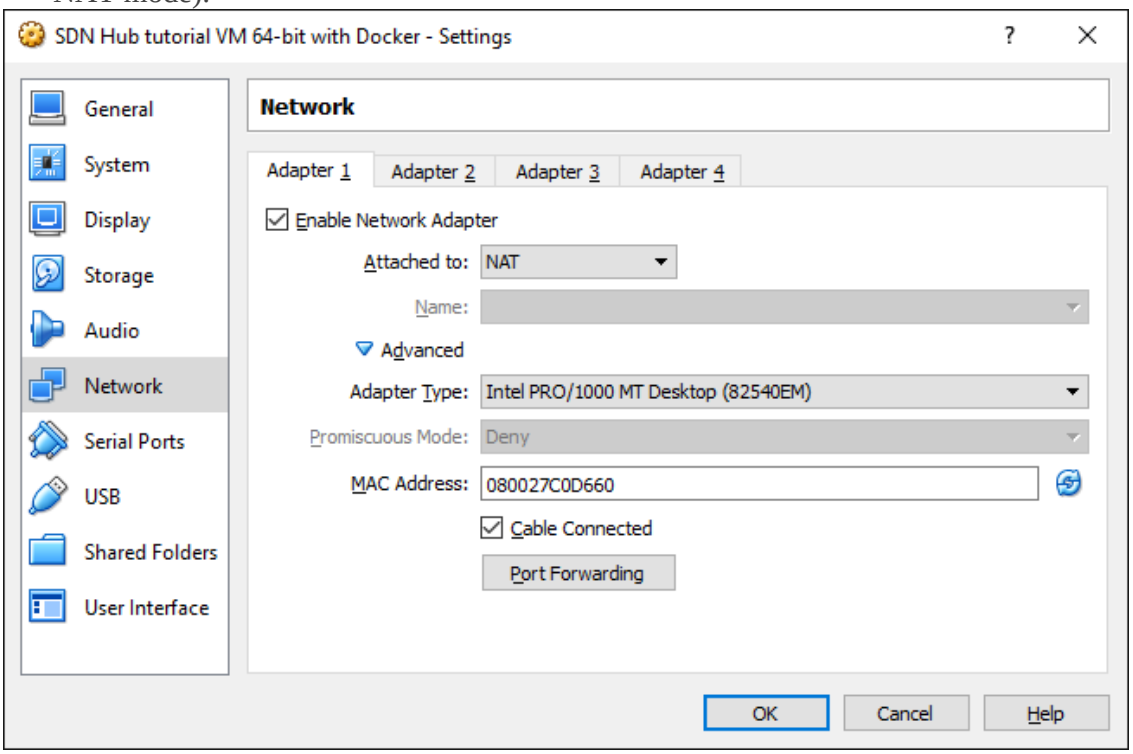
- Import the OVA (i.e., SDN Hub Tutorial VM) into Virtualbox or VMware Player and boot it. Feel free to change any of the VM attributes, but we highly recommend allocating at least 2 vCPUs and 2GB memory.



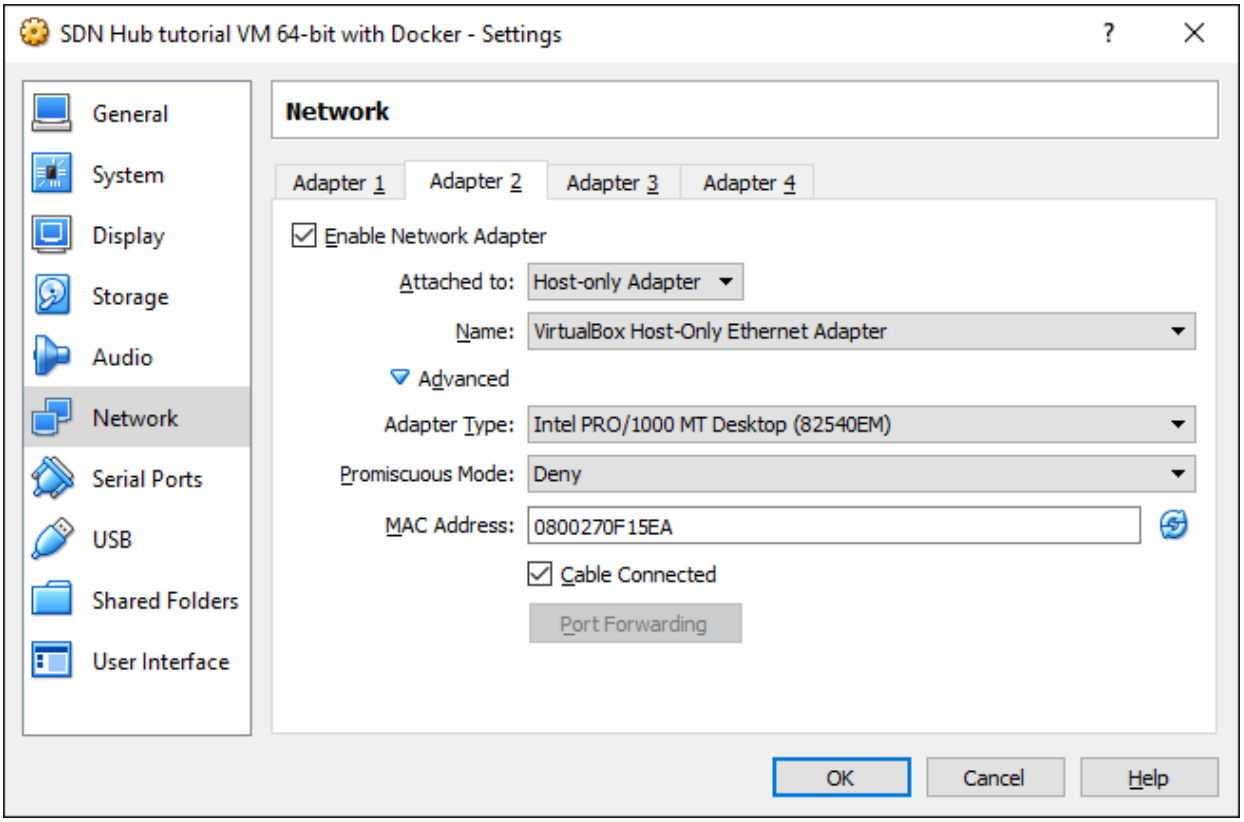


- In case the OVA (version 1.0) does not work on your VirtualBox or VMware player, unzip the OVA to extract the VMDK file. That file can be used to create a VM in your environment.

- Ensure you have connectivity to the Internet from the VM. If not, please ensure your Virtualbox/VMware network settings are correct for the VM's network adapter (should be in NAT mode).

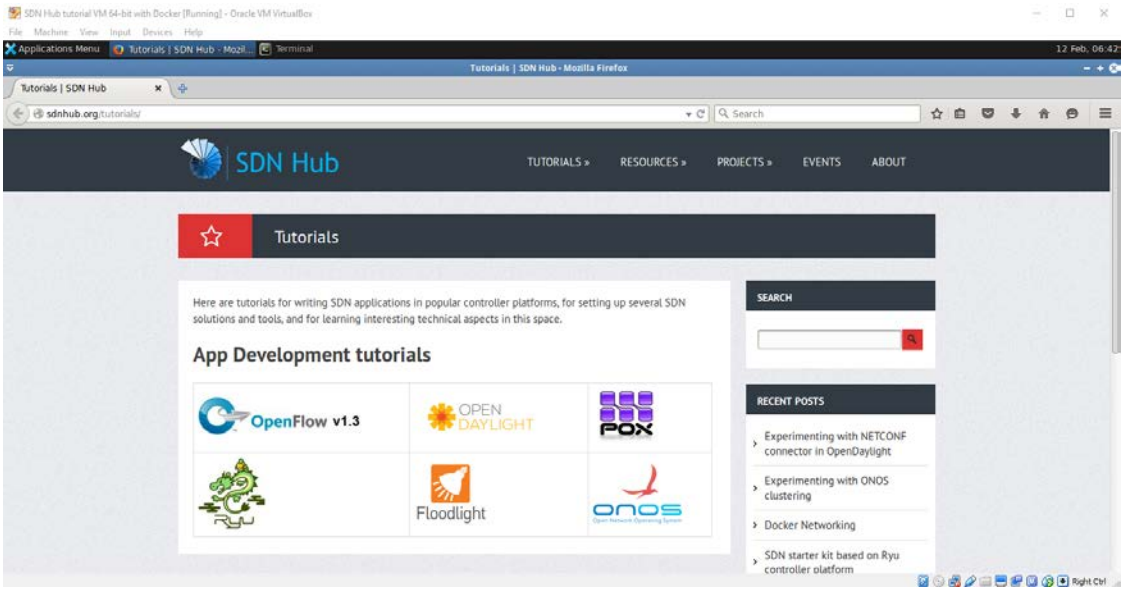


Add a second adaptor (which will be eth1) as a Host Only Adapter to allow use of PuTTY and WinSCP from your laptop to this new Ubuntu VM:

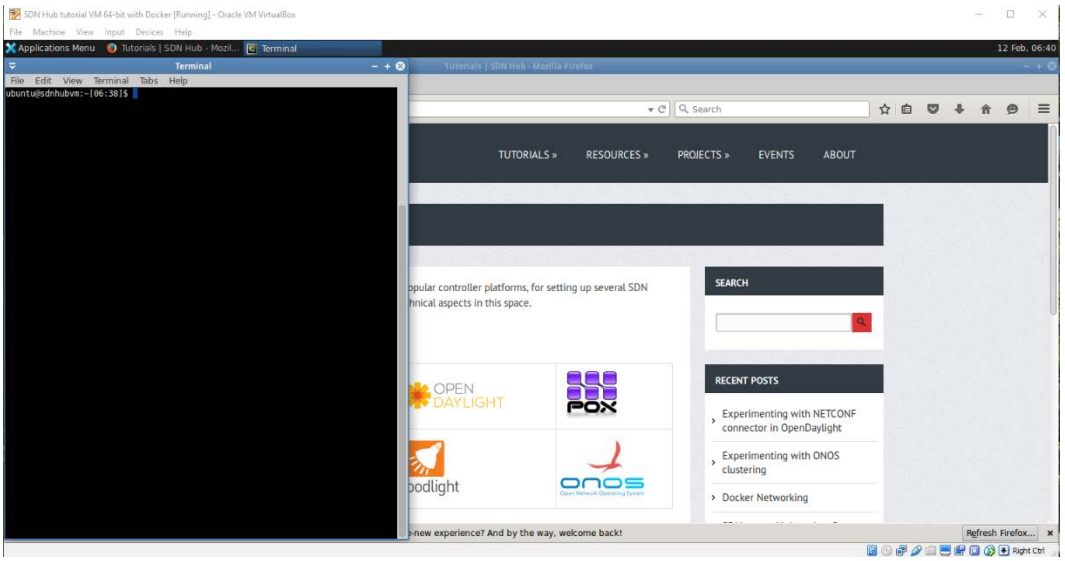


Note: To use this adapter you will have to:
sudo ifconfig eth1 192.168.56.101 netmask 255.255.255.0 up

After you power on the machine:



- You may now use a **Terminal Emulator** to create and run network topologies using mininet. A link is placed right on the desktop.



- **Username and passwd are both “ubuntu” but you are already logged on and in a terminal window if you opened the terminal window tab on the screen. When you use PuTTY or WinSCP instead of terminal windows you open up on the Ubuntu desktop, you will need this new username and password.**

****

Mininet Tutorial

After installing and getting set up, watch the following Introduction to Mininet Video (just watch do not do it as you watch) https://www.youtube.com/watch?v=jmlgXaocwiE

FYI only, no need to follow the next link: The next part of this lab is an excerpt from "Introduction to Mininet" which was copied from: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

What is Mininet?

Mininet is a *network emulator*, or perhaps more precisely a *network emulation orchestration system*. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine; you can `ssh` into it (if you start up `sshd` and bridge the network to your host) and run arbitrary programs (including anything that is installed on the underlying Linux system.) The programs you run can send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch, router, or middlebox, with a given amount of queueing. When two programs, like an `iperf` client and server, communicate through Mininet, the measured performance should match that of two (slower) native machines.

In short, Mininet's virtual hosts, switches, links, and controllers are the real thing – they are just created using software rather than hardware – and for the most part their behavior is similar to discrete hardware elements. It is usually possible to create a Mininet network that resembles a hardware network, or a hardware network that resembles a Mininet network, and to run the same binary code and applications on either platform.

Why is Mininet cool?

1. It's **fast** - starting up a simple network takes just a few seconds. This means that your run-edit-debug loop can be very quick.

2. You can **create custom topologies**: a single switch, larger Internet-like topologies, the Stanford backbone, a data center, or anything else.

3. You can **run real programs**: anything that runs on Linux is available for you to run, from web servers to TCP window monitoring tools to Wireshark.

4. You can **customize packet forwarding**: Mininet's switches are programmable using the OpenFlow protocol. Custom Software-Defined Network designs that run in Mininet can easily be transferred to hardware OpenFlow switches for line-rate packet forwarding.

5. You can **run Mininet on your laptop**, on a server, in a VM, on a native Linux box (Mininet is included with Ubuntu 12.10+!), or in the cloud (e.g. Amazon EC2.)

6. You can **share and replicate results**: anyone with a computer can run your code once you've packaged it up.

7. You can **use it easily**: you can create and run Mininet experiments by writing simple (or complex if necessary) Python scripts.

8. Mininet is an **open source project**, so you are encouraged to examine its source code onhttps://github.com/mininet, modify it, fix bugs, file issues/feature requests, and submit patches/pull requests. You may also edit this documentation to fix any errors or add clarifications or additional information.

9. Mininet is **under active development**. So, if it sucks, doesn't make sense, or doesn't work for some reason, please let us know on `mininet-discuss` and the Mininet user and developer community can try to explain it, fix it, or help you fix it. :-) If you find bugs, you are encouraged to submit patches to fix them, or at least to submit an issue on github including a reproducible test case.

What are Mininet's limitations?

Although we think Mininet is great, it does have some limitations. For example,

- Running on a single system is convenient, but it imposes resource limits: if your server has 3 GHz of CPU and can switch about 10 Gbps of simulated traffic,

those resources will need to be balanced and shared among your virtual hosts and switches.

- Mininet uses a single Linux kernel for all virtual hosts; this means that you can't run software that depends on BSD, Windows, or other operating system kernels. (Although you can attach VMs to Mininet.)

- Mininet won't write your OpenFlow controller for you; if you need custom routing or switching behavior, you will need to find or develop a controller with the features you require.

- By default your Mininet network is isolated from your LAN and from the internet - this is usually a good thing! However, you may use the `NAT` object and/or the `--nat` option to connect your Mininet network to your LAN via Network Address Translation. You can also attach a real (or virtual) hardware interface to your Mininet network (see `examples/hwintf.py` for details.)

- By default all Mininet hosts share the host file system and PID space; this means that you may have to be careful if you are running daemons that require configuration in /etc, and you need to be careful that you don't kill the wrong processes by mistake. (Note the `bind.py` example demonstrates how to have per-host private directories.)

- Unlike a simulator, Mininet doesn't have a strong notion of virtual time; this means that timing measurements will be based on real time, and that faster-than-real-time results (e.g. 100 Gbps networks) cannot easily be emulated.

An aside on performance: The main thing you have to keep in mind for network- limited experiments is that you will probably need to use slower links, for example 10 or 100 Mb/sec rather than 10 Gb/sec, due to the fact that packets are forwarded by a collection of software switches (e.g. Open vSwitch) that share CPU and memory resources and usually have lower performance than dedicated switching hardware. For CPU-limited experiments, you will also need to make sure that you carefully limit the CPU bandwidth of your Mininet hosts. If you mainly care about functional correctness, you can run Mininet without specific bandwidth limits - this is the quick and easy way to run Mininet, and it also provides the highest performance at the expense of timing accuracy under load.

With a few exceptions, many limitations you may run into will not be intrinsic to Mininet; eliminating them may simply be a matter of code, and you are encouraged to contribute any enhancements you may develop!

****

Mininet walkthrough tutorial
(Source: http://mininet.org/walkthrough/)

Complete the following mininet walkthrough tutorial included below.

Part 1: Everyday Mininet Usage

First, a (perhaps obvious) note on command syntax for this walkthrough:

- `$` preceeds Linux commands that should be typed at the shell prompt
- `mininet>` preceeds Mininet commands that should be typed at Mininet's CLI,
- `#` preceeds Linux commands that are typed at a root shell prompt

In each case, you should only type the command to the right of the prompt (and then press `return`, of course!)

Display Startup Options

Let's get started with Mininet's startup options.

Type the following command to display a help message describing Mininet's startup options:

```
$ sudo mn -h
```

This walkthrough will cover typical usage of the majority of options listed.

Interact with Hosts and Switches

Start a minimal topology by entering on the CLI in a terminal window:

```
$ sudo mn
```

The default topology is the `minimal` topology, which includes one OpenFlow kernel switch connected to two hosts, plus the OpenFlow reference controller. This topology could also be specified on the command line with `--topo=minimal`. Other topologies are also available out of the box; see the `--topo` section in the output of `mn -h`.

Three entities (2 host processes and one 1 switch process) are now running in the VM. (Aside: We do not yet have an “SDN controller” running, we are instead just using an internal Open Virtual Switch Bridge).

Display Mininet CLI commands:

```
mininet> help
```

Display nodes:

```
mininet> nodes
```

Display links:

```
mininet> net
```

Dump information about all nodes:

```
mininet> dump
```

You should see the switch and two hosts listed.

If the first string typed into the Mininet CLI is a host, switch or controller name, the command is executed on that node. Run a command on a host process:

```
mininet> h1 ifconfig -a
```

You should see the host’s `h1-eth0` and loopback (`lo`) interfaces. Note that this interface (`h1-eth0`) is not seen by the primary Linux system when `ifconfig` is run, because it is specific to the network namespace of the host process.
In contrast, the switch by default runs in the root network namespace, so running a command on the “switch” is the same as running it from a regular terminal:

```
mininet> s1 ifconfig -a
```

This will show the switch interfaces, plus the VM’s connection out (`eth0`).
For other examples highlighting that the hosts have isolated network state, run `arp` and `route` on both `s1` and `h1`.
It would be possible to place every host, switch and controller in its own isolated network namespace, but there’s no real advantage to doing so, unless you want to replicate a complex multiple-controller network. Mininet does support this; see the `--innamespace` option.
Note that *only* the network is virtualized; each host process sees the same set of processes and directories. For example, print the process list from a host process:

```
mininet> h1 ps -a
```

This should be the exact same as that seen by the root network namespace:

```
mininet> s1 ps -a
```

It would be possible to use separate process spaces with Linux containers, but currently Mininet doesn't do that. Having everything run in the "root" process namespace is convenient for debugging, because it allows you to see all of the processes from the console using `ps`, `kill`, etc.

Test connectivity between hosts

Now, verify that you can ping from host 0 to host 1:

```
mininet> h1 ping -c 1 h2
```

If a string appears later in the command with a node name, that node name is replaced by its IP address; this happened for h2.

An easier way to run this test is to use the Mininet CLI built-in `pingall` command, which does an all-pairs `ping`:

```
mininet> pingall
```

Run a simple web server and client

Remember that `ping` isn't the only command you can run on a host! Mininet hosts can run any command or application that is available to the underlying Linux system (or VM) and its file system. You can also enter any `bash` command, including job control (`&`, `jobs`, `kill`, etc..)
Next, try starting a simple HTTP server on `h1`, making a request from `h2`, then shutting down the web server:

```
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> h2 wget -O - h1
...
mininet> h1 kill %python
```

Exit the CLI:

```
mininet> exit
```

Cleanup

If Mininet crashes for some reason, clean it up:

```
$ sudo mn -c
```

Important ECE4110 Notes for troubleshooting:

To make sure that you are starting clean each time you change something you should use the following process to stop the controller (later in the lab we will use a controller) and to stop the mininet network:

Make sure that a controller is not running in the background (usually a control C in the window will stop it clean but just in case):

$ sudo killall controller

To shutdown Mininet and to make sure that everything is clean:

mininet> exit

$ sudo mn -c

Part 2: Advanced Startup Options

Run a Regression Test

You don't need to drop into the CLI; Mininet can also be used to run self-contained regression tests.

Run a regression test:

```
$ sudo mn --test pingpair
```

This command created a minimal topology, started up the OpenFlow reference switch, ran an all-pairs-`ping` test, and tore down both the topology.
Another useful test is `iperf` (give it about 10 seconds to complete):

```
$ sudo mn --test iperf
```

This command created the same Mininet, ran an iperf server on one host, ran an iperf client on the second host, and parsed the bandwidth achieved.

Changing Topology Size and Type

The default topology is a single switch connected to two hosts. You could change this to a different topo with `--topo`, and pass parameters for that topology's creation. For example, to verify all-pairs ping connectivity with one switch and three hosts:
Run a regression test:

```
$ sudo mn --test pingall --topo single,3
```

Another example, with a linear topology (where each switch has one host, and all switches connect in a line):

```
$ sudo mn --test pingall --topo linear,4
```

Parametrized topologies are one of Mininet's most useful and powerful features.

Link variations

Mininet 2.0 allows you to set link parameters, and these can even be set automatially from the command line:

```
$ sudo mn --link tc,bw=10,delay=10ms
mininet> iperf
...
mininet> h1 ping -c10 h2
```

If the delay for each link is 10 ms, the round trip time (RTT) should be about 40 ms, since the ICMP request traverses two links (one to the switch, one to the destination) and the ICMP reply traverses two links coming back.

You can customize each link using Mininet's Python API, but for now you will probably want to continue with the walkthrough.

Custom Topologies

Custom topologies can be easily defined as well, using a simple Python API, and an example is provided in /home/ubuntu/mininet/custom`/topo-2sw-2host.py`. This example connects two switches directly, with a single host off each switch:

```
"""Custom topology example
Two directly connected switches plus a host for each switch:
  host --- switch --- switch --- host
Adding the 'topos' dict with a key/value pair to generate our newly defined
topology enables one to pass in '--topo=mytopo' from the command line.
"""

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': ( lambda: MyTopo() ) }
```

When a custom mininet file is provided, it can add new topologies, switch types, and tests to the command-line. The above file connects host1 to switch 3, host2 to switch 4, and switch 3 to switch 4. For example, the following command line creates the topology and does a ping from h1 to h2 and also from h2 to h1:

```
$ sudo mn --custom ~/mininet/custom/topo-2sw-2host.py --topo mytopo --test pingall
```

ID = MAC

By default, hosts start with randomly assigned MAC addresses. This can make debugging tough, because every time the Mininet is created, the MACs change, so correlating control traffic with specific hosts is tough.

The `--mac` option is super-useful, and sets the host MAC and IP addresses to small, unique, easy-to-read IDs.
Before:

```
$ sudo mn
...
mininet> h1 ifconfig
h1-eth0  Link encap:Ethernet  HWaddr f6:9d:5a:7f:41:42
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:392 (392.0 B)  TX bytes:392 (392.0 B)
mininet> exit
```

After:

```
$ sudo mn --mac
...
mininet> h1 ifconfig
h1-eth0  Link encap:Ethernet  HWaddr 00:00:00:00:00:01
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
mininet> exit
```

In contrast, the MACs for switch data ports reported by Linux will remain random. This is because you can 'assign' a MAC to a data port using OpenFlow, as noted in the FAQ. This is a somewhat subtle point which you can probably ignore for now.

XTerm Display

For more complex debugging, you can start Mininet so that it spawns one or more xterms.

To start an `xterm` for every host and switch, pass the `-x` option:

```
$ sudo mn -x
```

After a second, the xterms will pop up, with automatically set window names.

Alternately, you can bring up additional xterms as shown below.

By default, only the hosts are put in a separate namespace; the window for each switch is unnecessary (that is, equivalent to a regular terminal), but can be a convenient place to run and leave up switch debug commands, such as flow counter dumps.

Xterms are also useful for running interactive commands that you may need to cancel, for which you'd like to see the output.

For example:

In the xterm labeled

"host: h1", run:

```
# ping 10.0.0.2
```

Close the setup, from the Mininet CLI:

```
mininet> exit
```

The xterms should automatically close.

Mininet Benchmark

To record the time to set up and tear down a topology, use test 'none':

```
$ sudo mn --test none
```

Part 3: Mininet Command-Line Interface (CLI) Commands

Display Options

To see the list of Command-Line Interface (CLI) options, start up a minimal topology and leave it running. Build the Mininet:

```
$ sudo mn
```

Display the options:

```
mininet> help
```

Python Interpreter

If the first phrase on the Mininiet command line is `py`, then that command is executed with Python. This might be useful for extending Mininet, as well as probing its inner workings. Each host, switch, and controller has an associated Node object.
At the Mininet CLI, run:

```
mininet> py 'hello ' + 'world'
```

Print the accessible local variables:

```
mininet> py locals()
```

Next, see the methods and properties available for a node, using the dir() function:

```
mininet> py dir(s1)
```

You can read the on-line documentation for methods available on a node by using the help() function:

```
mininet> py help(h1)
Press "q" to quit reading the documentation.
```

You can also evaluate methods of variables:

```
mininet> py h1.IP()
```

Link Up/Down

For fault tolerance testing, it can be helpful to bring links up and down.

To disable both halves of a virtual Ethernet pair:

```
mininet> link s1 h1 down
```

To bring the link back up:

```
mininet> link s1 h1 up
```

XTerm Display

To display an xterm for h1 and h2:

```
mininet> xterm h1 h2
mininet> exit
```

Now go back to the Introduction to Mininet
https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

And scan (do not spends lots of time just scan) the remainder of the web pages so you know what information is there in case you need it later.
****

Open-Flow Tutorial:
(Source: https://github.com/mininet/openflow-tutorial/wiki/Learn-Development-Tools)

Complete the following part of the open-flow tutorial which is copied here from the above source for your convenience:

Learn Development Tools

In this section, you'll bring up the development environment. In the process, you'll be introduced to tools that will later prove useful for turning the provided hub into a learning switch. You'll cover both general and OpenFlow-specific debugging tools.

The OpenFlowTutorial VM includes a number of OpenFlow-specific and general networking utilities pre-installed. Please read the short descriptions:

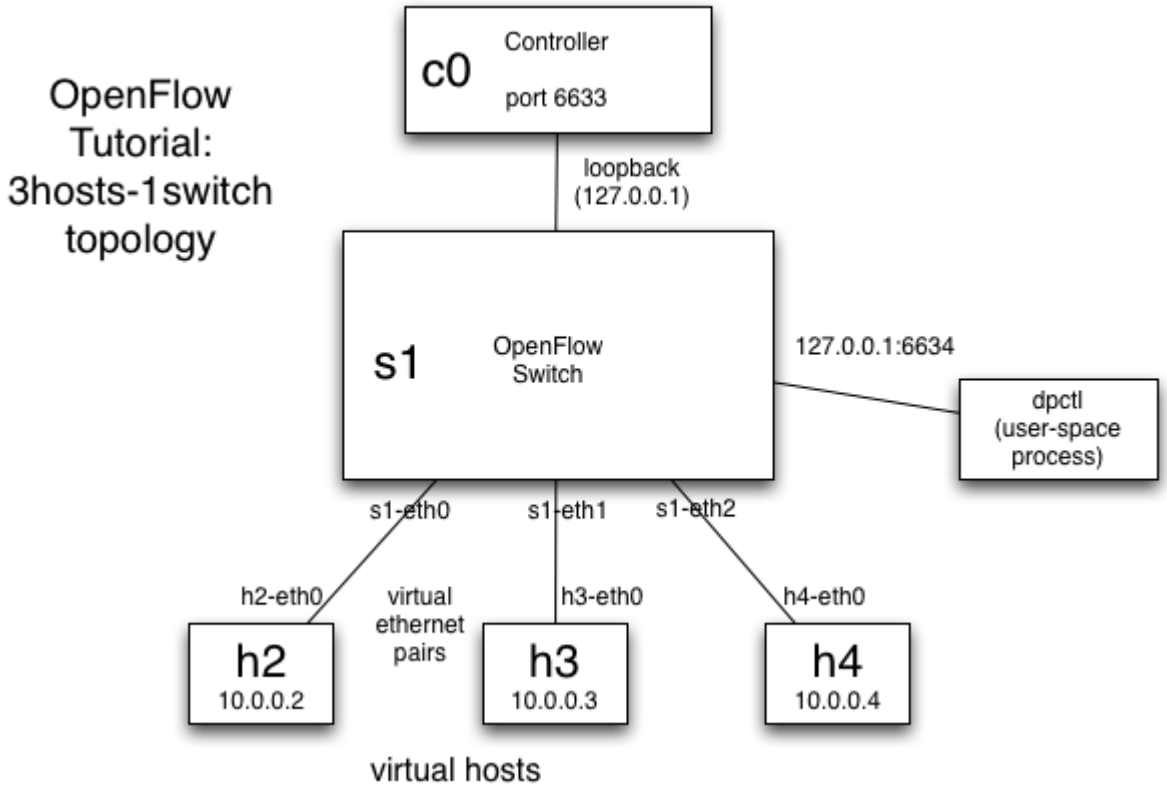
- **OpenFlow Controller**: sits above the OpenFlow interface. The OpenFlow reference distribution includes a controller that acts as an Ethernet learning switch in combination with an OpenFlow switch. You'll run it and look at messages being sent. Then, in the next section, you'll write our own controller on top of RYU (platforms for writing controller applications).
- **OpenFlow Switch**: sits below the OpenFlow interface. The OpenFlow reference distribution includes a user-space software switch. Open vSwitch is another software

but kernel-based switch, while there is a number of hardware switches available from Broadcom (Stanford Indigo release), HP, NEC, and others.
- **ovs-ofctl**: command-line utility that sends quick OpenFlow messages, useful for viewing switch port and flow stats or manually inserting flow entries.
- **Wireshark**: general (non-OF-specific) graphical utility for viewing packets. The OpenFlow reference distribution includes a Wireshark dissector, which parses OpenFlow messages sent to the OpenFlow default port (6633 or 6653) in a conveniently readable way.
- **iperf**: general command-line utility for testing the speed of a single TCP connection.
- **Mininet**: network emulation platform. Mininet creates a virtual OpenFlow network - controller, switches, hosts, and links - on a single real or virtual machine. More Mininet details can be found at the Mininet web page.
- **cbench**: utility for testing the flow setup rate of OpenFlow controllers.

From here on out, make sure to copy and paste as much as possible! For example, manually typing in ‘sudo dpctl show n1:0’ may look correct, but will cause a confusing error; the 'nl' is short for NetLink, not n-one.
Let's get started...

Start Network

The network you'll use for the first exercise includes 3 hosts and a switch (and, eventually, an OpenFlow controller, but we'll get to that later):



To create this network in the VM, in an SSH terminal, enter:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

This tells Mininet to start up a 3-host, single-(openvSwitch-based)switch topology, set the MAC address of each host equal to its IP, and point to a remote controller which defaults to the localhost.

Here's what Mininet just did:

- Created 3 virtual hosts, each with a separate IP address.
- Created a single OpenFlow software switch in the kernel with 3 ports.
- Connected each virtual host to the switch with a virtual ethernet cable.
- Set the MAC address of each host equal to its IP.
- Configure the OpenFlow switch to connect to a remote controller which does not yet exist.

Mininet Brief Intro

Since you'll be working in Mininet for the whole tutorial, it's worth REVIEWING a few Mininet-specific commands:

To see the list of nodes available, in the Mininet console, run:

```
mininet> nodes
```

To see a list of available commands, in the Mininet console, run:

```
mininet> help
```

To run a single command on a node, prepend the command with the name of the node. For example, to check the IP of a virtual host, in the Mininet console, run:

```
mininet> h1 ifconfig
```

The alternative - better for running interactive commands and watching debug output - is to spawn an xterm for one or more virtual hosts. In the Mininet console, run:

```
mininet> xterm h1 h2
```

You can close these windows now, as we'll run through most commands in the Mininet console.

If Mininet is not working correctly (or has crashed and needs to be restarted), first quit Mininet if necessary (using the `exit` command, or control-D), and then try clearing any residual state or processes using:

```
$ sudo mn -c
```

and running Mininet again.

NB: The prompt `mininet>` is for Mininet console, `$` is for terminal (normal user) and `#` is for terminal (root user) . Hereafter we follow with this rule.

Mininet has loads of other commands and startup options to help with debugging, and this brief starter should be sufficient for the tutorial. If you're curious about other options, follow the Mininet Walkthrough after the main tutorial.

ovs-ofctl Example Usage

(Just to remind you, we are assuming that you have already started up Mininet in another window using `sudo mn --topo single,3 --mac --switch ovsk --controller remote`, as directed above.)
ovs-ofctl is a utility that comes with Open vSwitch and enables visibility and control over a single switch's flow table. It is especially useful for debugging, by viewing flow state and flow counters. Most OpenFlow switches can start up with a passive listening port, from which you can poll the switch, without having to add debugging code to the controller.

Create a second SSH window if you don't already have one, and run:

```
$ sudo ovs-ofctl show s1
```

The `show` command connects to the switch and dumps out its port state and capabilities.

Here's a more useful command:

```
$ sudo ovs-ofctl dump-flows s1
```

Since we haven't started any controller yet, the flow-table should be empty.

Accessing remote OVS instances or the RYU reference switch

Note: In the above example, `ovs-ofctl` is talking to a local instance of Open vSwitch via a Unix domain socket which it is looking up by name. The actual socket is probably something like `/var/run/openvswitch/s1.mgmt` . If you were running another switch such as the Stanford reference switch or even a hardware OpenFlow switch, you would need to connect to a passive TCP port using a command like

```
$ sudo ovs-ofctl dump-flows tcp:{ip address}:{port}
```

where {ip address} is the IP address of the switch's management interface and {port} is the passive OpenFlow listening/management port. If Mininet is invoked with `--switch user`, it will open up a passive listening port for each switch at port (6633+n) where n is the number of the switch. This will enable `ovs-ofctl` to be used in commands like

```
$ sudo ovs-ofctl dump-flows tcp:127.0.0.1:6634
```

For the purposes of this tutorial, you will usually be using Open vSwitch so you can just use the simple form of the `ovs-ofctl` command.

Note that there are also `dpctl` and `ovs-dpctl` commands; usually these commands should not be used and `ovs-ofctl` should be used instead! With Open vSwitch, those commands can

allow you to examine OVS's kernel flow cache, which is usually a subset of the full OpenFlow flow table which you or the controller have programmed. Flows in OVS's flow cache will usually time out quickly, for example within 5 seconds. This allows OVS to support large flow tables reasonably efficiently without a huge kernel flow cache, but it can also introduce slowdowns when flows miss in the kernel flow cache and have to be reloaded from user space.

Ping Test

Now, go back to the mininet console and try to ping h2 from h1. In the Mininet console:

```
mininet> h1 ping -c3 h2
```

Note that the name of host h2 is automatically replaced when running commands in the Mininet console with its IP address (10.0.0.2).

Do you get any replies? Why? Why not?

As you saw before, switch flow table is empty. Besides that, there is no controller connected to the switch and therefore the switch doesn't know what to do with incoming traffic, leading to ping failure.

You'll use `ovs-ofctl` to manually install the necessary flows. In your SSH terminal:

```
$ sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2

$ sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
```

This will forward packets coming at port 1 to port 2 and vice-verca. Verify by checking the flow-table

```
$ sudo ovs-ofctl dump-flows s1
```

Run the ping command again. In your mininet console:

```
mininet> h1 ping -c3 h2
```

Do you get replies now? Check the flow-table again and look the statistics for each flow entry. Is this what you expected to see based on the ping traffic?

Examining OpenFlow Messages with Wireshark

Source: http://sdnhub.org/tutorials/openflow-1-3/

OpenFlow version 1.3 is the latest version of OpenFlow that has support from switch vendors. It is significantly different from OpenFlow version 1.0 (which was the previous version several vendors supported). Among others, the main features added since then are:

- **1.1**: Support for MPLS, Q-in-Q, VLANs, multipath, multiple tables, logical ports

- **1.2**: Support for extensible headers (in match, packet_in, set_field), IPv6
- **1.3**: Support for tunneling, per-flow traffic meters, Provider Backbone Bridging

In this part of the tutorial you will learn more about OpenFlow version 1.3 under the covers.

The Tutorial VM has wireshark and OFDissector installed for OpenFlow version 1.3. The dissector is based on CPqD's release. This enables us to inspect the exact syntax of the OpenFlow messages.

1. Quickstart

- Run Mininet on a terminal window using the following command. This starts a network emulation environment to emulate 1 switch with 3 hosts using protocol version 1.3.

```
sudo mn --topo single,3 --mac --controller remote --switch
ovsk,protocols=OpenFlow13
```

- Note that the above command will work in our patched mininet. For other mininet installations, you can run the following command to make a switch supports OF 1.3:

```
ovs-vsctl set bridge s1 protocols=OpenFlow13
```

- The Wireshark 1.11.3 that is part of the VM can parse OpenFlow 1.0, 1.1., 1.2, 1.3 and 1.4 messages. To start wireshark and view OpenFlow messages in a second terminal window:

```
sudo wireshark &
```

- Select Loopback: lo interface and click on the green start shark fin.
- In the wireshark filter window enter openflow_v4 and carriage return followed by clicking on Apply

- Next, start the RYU Controller from the second terminal window. The main folder where ryu is installed is in /home/ubuntu/ryu, The below command starts the controller by initiating the OpenFlow Protocol Handler and Simple Switch 1.3 application.

```
cd /home/ubuntu/ryu && ./bin/ryu-manager --verbose
ryu/app/simple_switch_13.py
```

- Next, check if the hosts in the mininet topology can reach each other

```
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=2.76 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.052 ms
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.051 ms
```

- Aside: You can now list the ongoing flows using the following command that is specific to OpenFlow 1.3 in a third terminal window:

```
sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

View Startup messages in Wireshark

You should see a bunch of messages displayed in Wireshark, from the Hello exchange onwards. As an example, click on the Features Reply message. Click on the triangle by the 'OpenFlow Protocol' line in the center section to expand the message fields. Click the triangle by Switch Features to display datapath capabilities - feel free to explore.

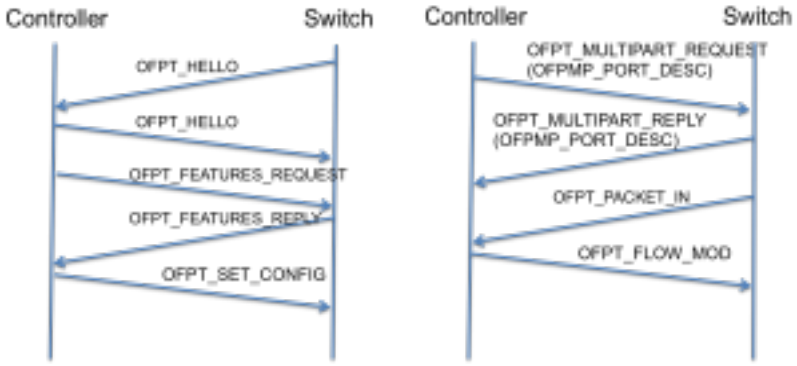
These messages include:

Message	Type	Description
Hello	Controller->Switch	following the TCP handshake, the controller sends its version number to the switch.
Hello	Switch->Controller	the switch replies with its supported version number.
Features Request	Controller->Switch	the controller asks to see which ports are available.
Set Config	Controller->Switch	in this case, the controller asks the switch to send flow expirations.
Features Reply	Switch->Controller	the switch replies with a list of ports, port speeds, and supported tables and actions.

Since all messages are sent over localhost when using Mininet, determining the sender of a message can get confusing when there are lots of emulated switches. However, this won't be an issue, since we only have one switch. The controller is at the standard OpenFlow port (6633), while the switch is at some other user-level port.

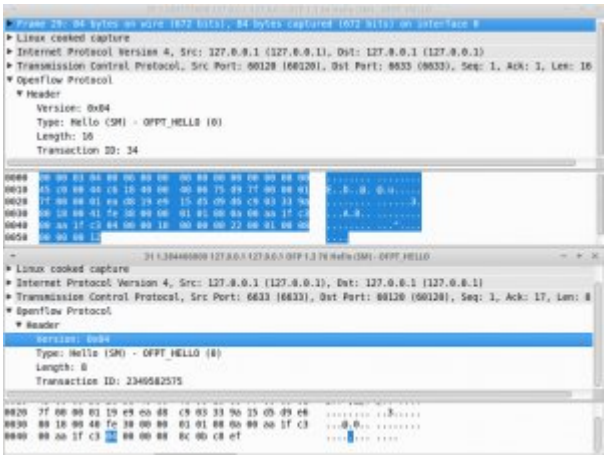
2. Understanding OpenFlow Messages

We now take a deep dive into understanding the set of OpenFlow messages exchanged between controller and switch, as shown in the following figure and captured in your wireshark session.



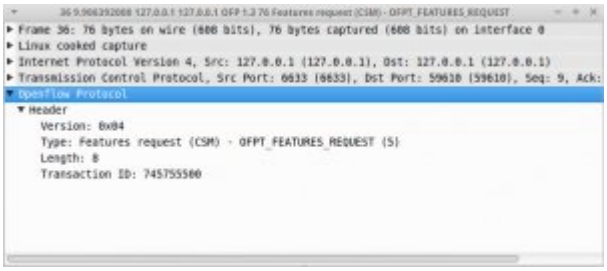
Connection Setup

The switch initiates a standard TCP (or TLS) connection to the controller. When an OpenFlow connection is established, each entity must send an OFPT_HELLO message with the protocol version set to the highest OpenFlow protocol version supported by the sender. In the below figure, we can see that OpenFlow version 1.3 has been negotiated.

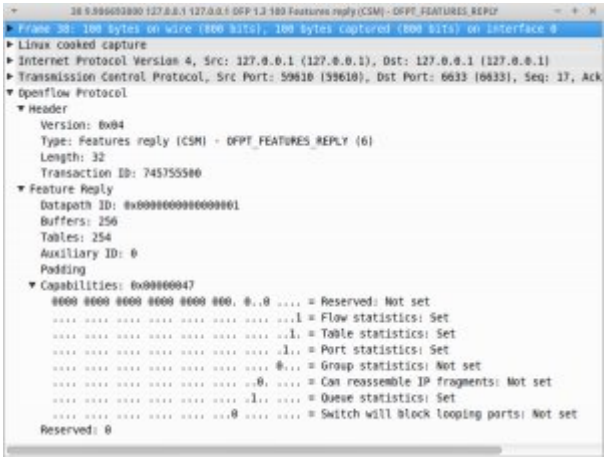


Feature Request – Reply

After successfully establishing a session, the controller sends an OFPT_FEATURES_REQUEST message. This message only contains an OpenFlow header and does not contain a body.

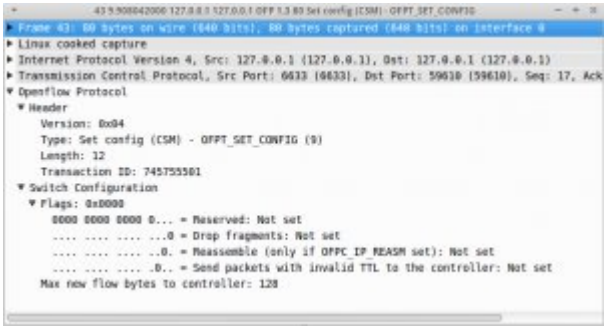


The switch responds with an OFPT_FEATURES_REPLY message. Notice the Datapath ID and the switch capabilities sent as part of the Feature reply message.



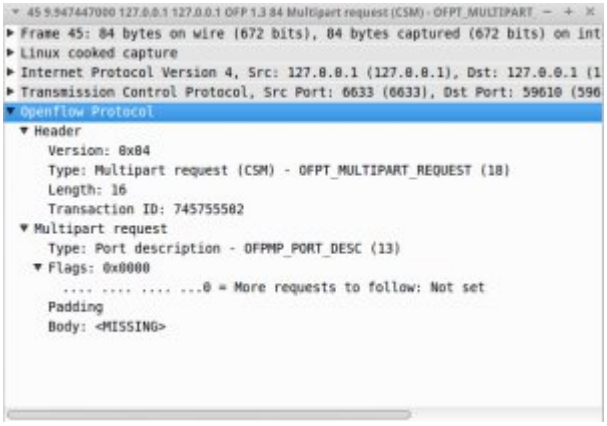
Set Configuration

Next, the controller sends the OFPT_SET_CONFIG message to the switch. This includes the set of flags and Max bytes of packet that datapath should send to the controller.

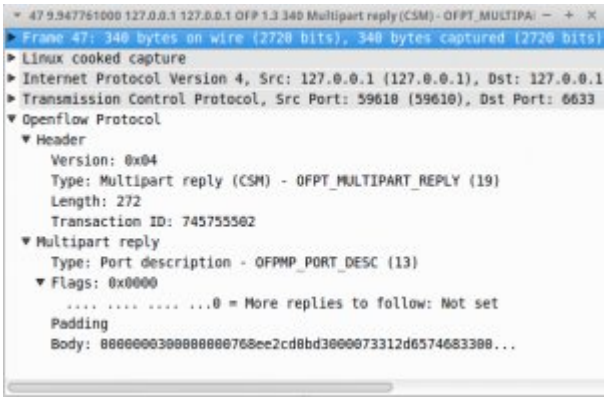


Multipart Request – Reply

The controller may request state from the datapath using the OFPT_MULTIPART_REQUEST message. The message types handled by this message include various statistics (FLOW/TABLE/PORT/QUEUE/METER etc) or description features (METER_CONFIG/TABLE_FEATURES/PORT_DESC etc). In our simple_switch_13.py, RYU internally sends a MULTIPART_REQUEST to request port description.



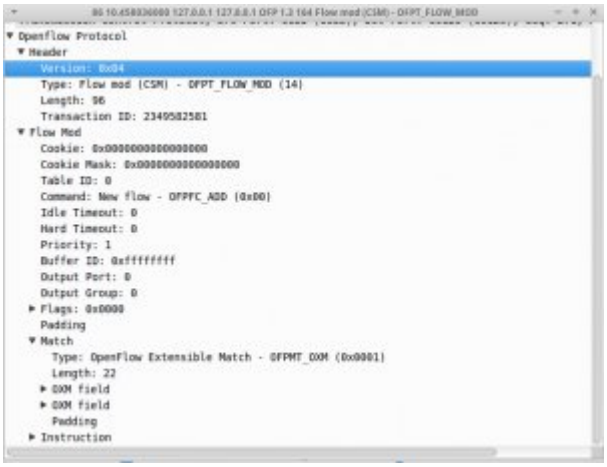
The switch replies with the PORT_DESCRIPTION of all active ports in the switch. Note: in OF 1.0, the port descriptions was returned as part of the FEATURE_REPLY message. Now this is handled separately as MULTIPART_* in OF 1.3.



Flow Mod

Flows can be proactively (e.g., pre-install flows like **TableMissFlow**) or reactively (e.g., react for packet_in messages) sent from the controller. Flow table modi

cation messages can have the following types:
OFPFC_ADD, OFPFC_DELETE, OFPFC_DELETE_STRICT, OFPFC_MODIFY, OFPFC_MOD IFY_STRICT.
In the following case, the controller installs a new flow, which shows that apart from the set of OF 1.0 parameters like priority, idle_timeout etc, the match and instruction structure reflect the new parameters specified in 1.3.



It is important to note that the switch does not positively acknowledge for FLOW_MOD messages. However, any error in the FLOW_MOD request will be replied
with OFPET_FLOW_MOD_FAILED.

Set Async Configuration Message

Asynchronous messages are sent from a switch to the controller. The set of messages supported by the OpenFlow protocol include “Packet-Ins, Flow-Removed, Port-Status or Error” messages. When the switch connects to the controller, the controller can set the type of messages that it wants to receive on its OpenFlow channel.

Above picture shows an async config message sent by the controller. Depending on the type of flags set, various async messages can be received from the switch.

View OpenFlow Messages for Ping

Now, we'll view messages generated in response to packets.

Before that update your wireshark filter to ignore the echo-request/reply messages (these are used to keep the connection between the switch and controller alive): Type the following in your wireshark filter, then press apply:

```
openflow_v4 && (openflow_v4.type != 3) && (openflow_v4.type != 2)
```

It's also recommended to clean up ARP cache on both hosts, otherwise you might not see some ARP requests/replies as the cache will be used instead:

```
mininet> h1 ip -s -s neigh flush all

mininet> h2 ip -s -s neigh flush all
```

Run a ping to view the OpenFlow messages being used. Do the ping in the Mininet console:

```
mininet> h1 ping -c1 h2
```

In the Wireshark window, you should see a number of new message types:

Message	Type	Description
Packet-In	Switch->Controller	a packet was received and it didn't match any entry in the switch's flow table, causing the packet to be sent to the controller.
Packet-Out	Controller->Switch	controller send a packet out one or more switch ports.
Flow-Mod	Controller->Switch	instructs a switch to add a particular flow to its flow table.
Flow-Expired	Switch->Controller	a flow timed out after a period of inactivity.

Not sure exactly what is going on here but here is a first guess:

First, you see an ARP request miss the flow table which generates a Packet In message from the switch to the controller. That event generates a Packet-Out message from the controller. Next, the ARP response comes back which generates a packet-In message from the switch to the controller. With both MAC addresses now known to the controller, the controller it can push down to the switch a Flow-Mod message. The switch allows flows for the ARP packets. Next a

ping request from H1 to H2 generates a packet in message from switch to controller. The controller sends back a Flow-Mod message allowing ICMP. Subsequent ping requests go straight through the datapath, and should incur no extra messages; with the flows connecting h1 and h2 already pushed to the switch, there is no additional controller involvement.

Re-run the ping, again from the Mininet console (hitting up is sufficient - the Mininet console has a history buffer):

```
mininet> h1 ping -c1 h2
```

If the ping takes the same amount of time, run the ping once more; the flow entries may have timed out while reading the above text.

This is an example of using OpenFlow in a *reactive* mode, when flows are pushed down in response to individual packets.

Alternately, flows can be pushed down before packets, in a *proactive* mode, to avoid the round-trip times and flow insertion delays.

Aside:

Not sure why if we remove the flows we cannot do another ping to add them back again. Instead no flows are allowed to be added. To remove any flows in the flow table in the switch between h1 and h2:

```
sudo ovs-ofctl del-flows s1 -O OpenFlow13
```

In theory we should be able to send new traffic, and repeat adding flows to the table but that does not work for me :>(

Benchmark Controller w/iperf

iperf is a command-line tool for checking speeds between two computers.

Here, you'll benchmark the reference controller; later, you'll compare this with the provided hub controller, and your flow-based switch (when you've implemented it).

In the mininet console run :

```
mininet> iperf
```

This Mininet command runs an iperf TCP server on one virtual host, then runs an iperf client on a second virtual host. Once connected, they blast packets between each other and report the results.

Question 2:

What value did you get for your network?

Exit Mininet:

```
mininet> exit
```

Question 3:

(a) Lets begin by creating a simple mininet topology.

```
sudo mn --topo single,2 --mac --switch ovsk --controller remote
```

(b) Next, start the RYU simple switch Controller from a second terminal window.

```
cd /home/ubuntu/ryu && ./bin/ryu-manager --verbose ryu/app/simple_switch_13.py
```

(c) What tcp:ip port is the OpenFlow controller and OpenVSwitch instance using? (hint: use "dump" command at the mininet prompt)

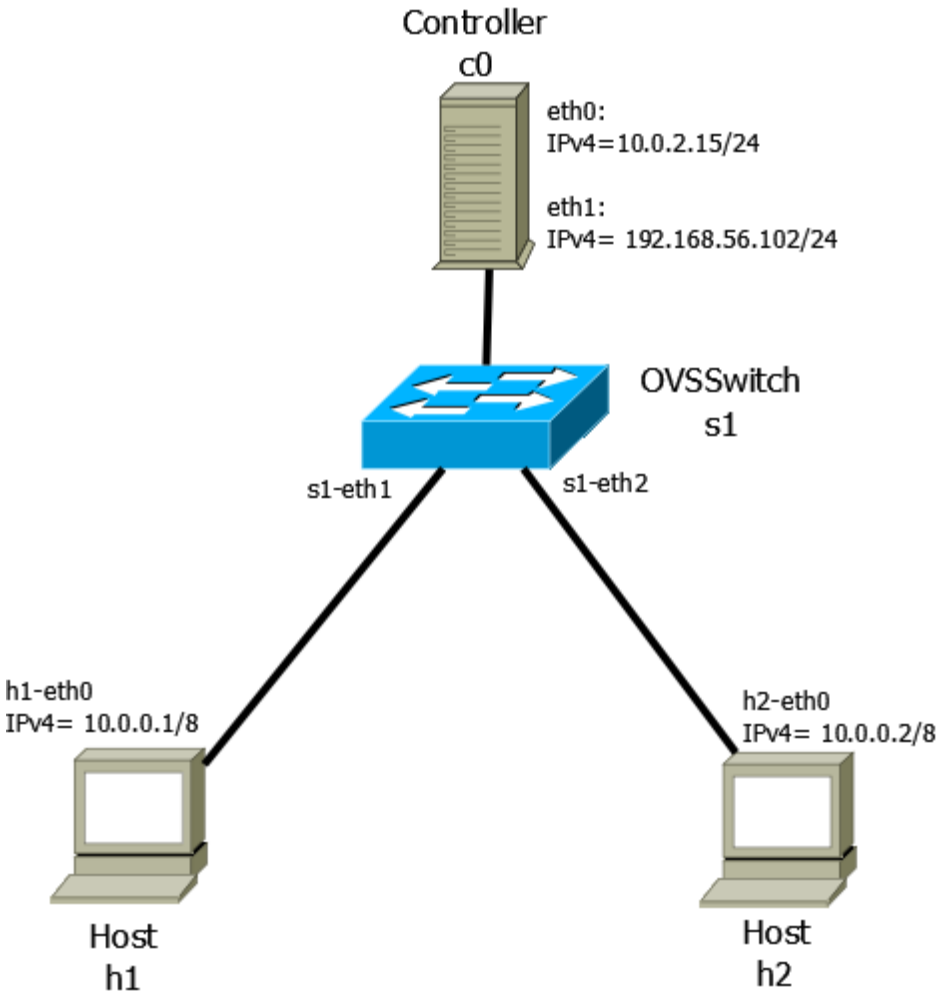
(d) Which network interfaces do you see from h1's namespace? (hint "h1 ifconfig")

(e) Which network interfaces do you see from h2's namespace? (hint "h2 ifconfig")

(f) Which network interfaces do you see from s1's namespace? (hint "s1 ifconfig")

(e) Aside: run netstat on the controller c0 with the command: "c0 netstat" and look at the first two lines. Just an FYI only, no need to do anything with this info.

(f) Draw the resulting topology and include the ip addresses and MAC assigned to each of the hosts. You may use the following diagram as your starting point and modify all incorrect info:



(g) From the Mininet prompt, execute the command

sh ovs-ofctl dump-flows s1

Explain what this command does. Include in your report what you see, and what does it mean?
(h) Now send a ping from h1 to h2

h1 ping –c1 h2

Execute the dump-flows command again. Include in your report what you now see, and what does it mean? Explain in detail.
(i) Also have a look at the flow statistics included in the output from the dump-flows command. What kind of stats are gathered?

(j) Exit Mininet (exit) and also stop the controller (control c in its window) start a new Mininet topology using the command

```
sudo mn –c
sudo mn --topo single,2 --mac --switch ovsk --controller none
```

Send a ping from h1 to h2. What happens and why?
h1 ping -c1 h2

(i) What needs to happen before pings will be forwarded again?
Hint: sh ovs-ofctl dump-flows s1

When finished exit mininet.

Question 4: (15 Points) Dissecting the OpenFlow protocol with tshark

Now let's take a closer look at what's going on in the open flow control channel. By using a non-graphical wireshark like tool.

```
sudo mn –c
sudo mn --topo single,2 --mac --switch ovsk --controller remote
```

Next, start the RYU simple switch Controller from a second terminal window.

cd /home/ubuntu/ryu && ./bin/ryu-manager --verbose ryu/app/simple_switch_13.py

Fire up tcpdump (in another window) on the loop back interface of the control plane c0, and have it capture all traffic where the src or dst port is 6633 and write out the entire packets (including payloads) to some file (You will want to use the option -s 0 to make sure tcpdump captures entire packets, not just headers).

```
sudo tcpdump -i lo -s 0 'tcp port 6633' -w dump.pcap
```

Send a single ping from h1 to h2.

h1 ping -c 1 h2

Exit tcpdump with a "control c". tcpdump should report that some packets have been captured.

(a) Use tshark with the OpenFlow dissector available in the mininet VM to read and analyze the dumpfile.

First to get an overview of what you captured and include this output in your report:
```
tshark -r dump.pcap
```

(d) Now using the OpenFlow dissector on the same data and include this output in your report:
```
tshark -O of -r dump.pcap
```

You should see the individual OF protocol messages now in clear text.

(e) Explain the individual messages that you see. You will want to reference the OpenFlow spec https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf .

(f) Why do some messages have to be flooded to all ports?

Question 5: From OpenFlow to Software Defined Networking.
So far, you've become familiar with OpenFlow, however this protocol is just a small architectural component of a much bigger picture: Software Defined Networking (SDN). View the hour long video lecture, “An Attempt to Motivate and clarify Software Defined Networking" from Professor Scott Shenker:
https://www.youtube.com/watch?v=WVs7Pc99S7w

While you watch this video, keep in mind, and then answer the following questions:
(a) What is the relationship between OpenFlow and SDN? What role specially does OpenFlow play?
(b) What are at least two motivating factors behind SDN?
(b) Around minute 39 of the talk, Professor Shenker mentions that SDN isn't just a better mechanism, but rather it is an instantiation of fundamental abstractions." Explain in detail what he means by this and give an example of one of the key abstractions that SDN builds upon.

RYU Controller Tutorial

Ryu is written fully in python script. So just open related files with your favorite editor and edit them. And restart ryu. ryu/app/simple_switch.py is a good starting point.

The First Application: A Hub

Source: http://ryu.readthedocs.org/en/latest/writing_ryu_app.html

Whetting Your Appetite

If you want to manage the network gears (switches, routers, etc) at your way, you need to write your Ryu application. Your application tells Ryu how you want to manage the gears. Then Ryu configures the gears by using OpenFlow protocol, etc.

Writing Ryu application is easy. It’s just Python scripts.

Start Writing

We show a Ryu application that make OpenFlow switches work as a dumb layer 2 hub.
Source: The First Application.

```
from ryu.base import app_manager

class L2Switch(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)
```

Ryu application is just a Python script so you can save the file with any name, extensions, and any place you want. The following file is named tutorial_l2_hub.py and is on the virtual machine in the directory /home/ubuntu/ryu/ryu/app

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls

# Hub behavior:
# For each packet received, do packet_out with FLOOD action

class L2Hub(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Hub, self).__init__(*args, **kwargs)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser

        actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
        out = ofp_parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id, in_port=msg.in_port,
            actions=actions)
        dp.send_msg(out)
```

A method ‘packet_in_handler’ is added to L2Hub class. This is called when Ryu receives an OpenFlow packet_in message. The trick is ‘set_ev_cls’ decorator. This decorator tells Ryu when the decorated function should be called.

The first argument of the decorator indicates an event that makes function called. As you expect easily, every time Ryu gets a packet_in message, this function is called.

The second argument indicates the state of the switch. Probably, you want to ignore packet_in messages before the negotiation between Ryu and the switch finishes. Using 'MAIN_DISPATCHER' as the second argument means this function is called only after the negotiation completes.

Next let's look at the first half of the 'packet_in_handler' function.

- ev.msg is an object that represents a packet_in data structure.
- msg.datapath is an object that represents a datapath (switch).
- dp.ofproto and dp.ofproto_parser are objects that represent the OpenFlow protocol that Ryu and the switch negotiated.

Ready for the second half.

- OFPActionOutput class is used with a packet_out message to specify a switch port that you want to send the packet out of. This application need a switch to send out of all the ports so OFPP_FLOOD constant is used.
- OFPPacketOut class is used to build a packet_out message.
- If you call Datapath class's send_msg method with a OpenFlow message class object, Ryu builds and send the on-wire data format to the switch.

Here, you have finished eamining your first Ryu application. You are ready to run this Ryu application. In the SDN Hub Virtual machine:

Open two terminal windows:

```
In a Terminal Window #1:
```

- Run Mininet on a terminal window using the following command. This starts a network emulation environment to emulate 1 switch with 3 hosts.

```
$ sudo mn -c
$ sudo mn --topo single,3 --mac --controller remote --switch ovsk
```

- The above command will spawn 1 switch that has support for both OpenFlow versions 1.0 and 1.3.

In Terminal Window #2:

```
$ sudo killall controller
```

To make this old tutorial code written for openflow v1.0 work, you will now need to tell the switch to only talk Openflow version 1.0:

```
$ sudo ovs-vsctl set bridge s1 protocols=OpenFlow10
```

•Next, start the RYU Controller. The main folder where ryu is installed is in /home/ubuntu/ryu. The following command starts the controller by initiating the OpenFlow Protocol Handler and Simple Switch application.

```
$ cd /home/ubuntu/ryu && ./bin/ryu-manager --verbose ryu/app/tutorial_l2_hub.py
```

In Terminal Window #1:

- Confirm the hosts in the mininet topology can reach each other

```
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=2.76 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.052 ms
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.051 ms
```

Control c to sop the ping.

FYI Only: Source for this was https://github.com/osrg/ryu/wiki/OpenFlow_Tutorial)

Verify Hub Behavior with tcpdump

Now that we have verified that hosts can ping each other, we will verify that all three hosts see the exact same traffic - the behavior of an **ethernet hub**. To do this, we'll create xterms for each host, and view the traffic in each host. In the Mininet console, start up three xterms:

```
mininet> xterm h1 h2 h3
```

Arrange each xterm so that they're all on the screen at once. This may require reducing the height of to fit a cramped laptop screen.

In the xterms for h2 and h3, run tcpdump, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

In the xterm for h1, send a ping:

```
# ping -c1 10.0.0.2
```

The ping packets are now going up to the controller, which then floods them out all interfaces except the sending one. You should see identical ARP and ICMP packets corresponding to the ping in both xterms running tcpdump.

Now, see what happens when a non-existent host doesn't reply. From h1 xterm:

```
# ping -c1 10.0.0.4
```

You should see three unanswered ARP requests in the tcpdump xterms. If your code is off later, three unanswered ARP requests is a signal that you might be accidentally dropping packets.

You can close the h1, h2, h3 xterms now.

Benchmark Hub Controller w/iperf

Here, you'll benchmark the provided hub code.

First, verify reachability. Mininet should be running, along with your Ryu tutorial controller. In the Mininet console, run:

```
mininet> pingall
```

This is just a sanity check for connectivity. Now, in the Mininet console, run:

```
mininet> iperf
```

Question 6: Iperf of an Ethernet hub with controller printing out events

At what speed does Iperf say your network runs?

You may now exit minet and control C the controller.

RYU Switch Controller

Follow and work through yourself the following detailed explanation simple_switch_13.py taken from the RYU documentation. This expands the simple non-switching hub to a learning Ethernet switch. The documentation calls it a switching hub.

Source: https://osrg.github.io/ryu-book/en/Ryubook.pdf
https://osrg.github.io/ryu-book/en/html/switching_hub.html

CHAPTER

ONE

SWITCHING HUB

This section uses implementation of a simple switching hub as a material to describe the method of implementing applications using Ryu.

1.1 Switching Hub

Switching hubs have a variety of functions. Here, we take a look at a switching hub having the following simple functions.

- Learns the MAC address of the host connected to a port and retains it in the MAC address table.
- When receiving packets addressed to a host already learned, transfers them to the port connected to the host.
- When receiving packets addressed to an unknown host, performs flooding.

Let's use Ryu to implement such a switch.

1.2 Switching Hub by OpenFlow

OpenFlow switches can perform the following by receiving instructions from OpenFlow controllers such as Ryu.

- Rewrites the address of received packets or transfers the packets from the specified port.
- Transfers the received packets to the controller (Packet-In).
- Transfers the packets forwarded by the controller from the specified port (Packet-Out).

It is possible to achieve a switching hub having those functions combined.

First of all, you need to use the Packet-In function to learn MAC addresses. The controller can use the Packet-In function to receive packets from the switch. The switch analyzes the received packets to learn the MAC address of the host and information about the connected port.

After learning, the switch transfers the received packets. The switch investigates whether the destination MAC address of the packets belong to the learned host. Depending on the investigation results, the switch performs the following processing.

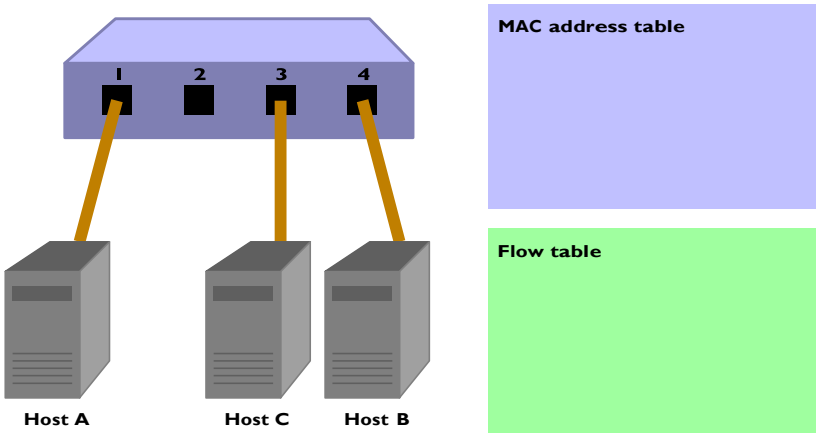
- If the host is already a learned host ... Uses the Packet-Out function to transfer the packets from the connected port.
- If the host is unknown host ... Use the Packet-Out function to perform flooding.

The following explains the above operation in a step-by-step way using figures.

1. Initial status

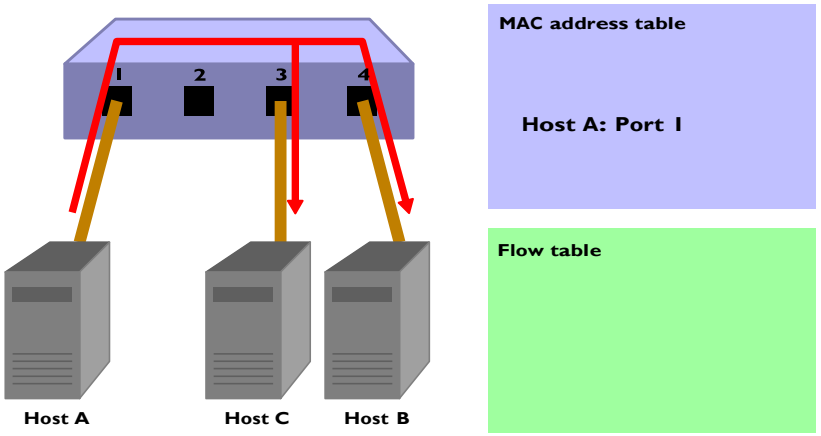
   This is the initial status where the flow table is empty.

   Assuming host A is connected to port 1, host B to part 4, and host C to port 3.



2. Host A -> Host B

   When packets are sent from host A to host B, a Packet-In message is sent and the MAC address of host A is learned by port 1. Because the port for host B has not been found, the packets are flooded and are received by host B and host C.



   Packet-In:

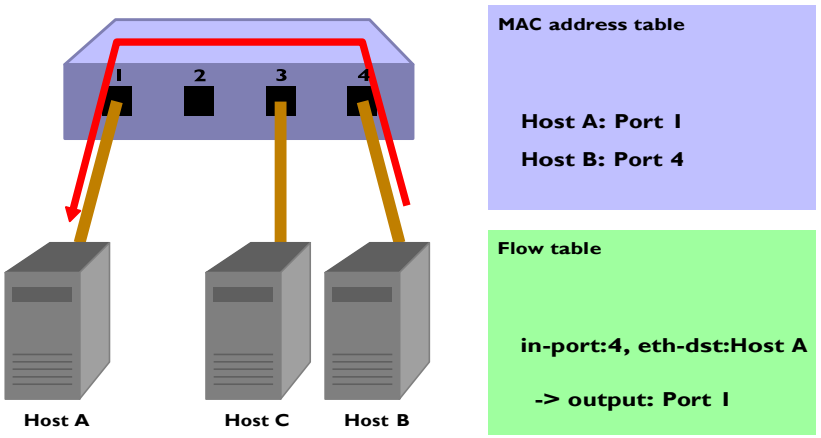
```
in-port: 1

eth-dst: Host B
eth-src: Host A
```

   Packet-Out:

```
action: OUTPUT:Flooding
```

3. Host B -> Host A

   When the packets are returned from host B to host A, an entry is added to the flow table and also the packets are transferred to port 1. For that reason, the packets are not received by host C.



Packet-In:

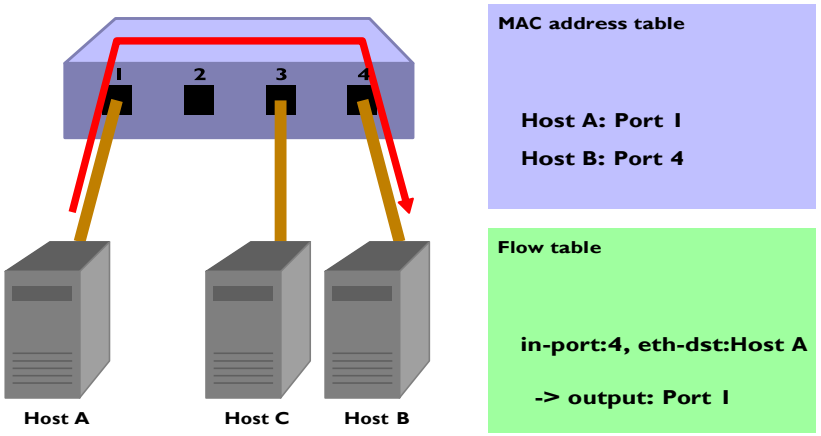
```
in-port: 4
eth-dst: Host A
eth-src: Host B
```

Packet-Out:

```
action: OUTPUT:Port 1
```

4. Host A -> Host B

   Again, when packets are sent from host A to host B, an entry is added to the flow table and also the packets are transferred to port 4.



Packet-In:

```
in-port: 1
eth-dst: Host B
eth-src: Host A
```

Packet-Out:

```
action: OUTPUT:Port 4
```

Next, let's take a look at the source code of a switching hub implemented using Ryu.

1.3 Implementation of Switching Hub Using Ryu

The source code of the switching hub is in Ryu's source tree.

/home/ubuntu/ryu/ryu/app/simple_switch_13.py

Other than the above, there are simple_switch.py(OpenFlow 1.0) and simple_switch_12.py(OpenFlow 1.2), depending on the version of OpenFlow but we take a look at implementation supporting OpenFlow 1.3.

The source code is short thus we shown the entire source code below.

```
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


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

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

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

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

        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})
```

```
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(datapath, 1, match, actions)

        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
```

Let's examine the respective implementation content.

1.3.1 Class Definition and Initialization

In order to implement as a Ryu application, ryu.base.app_manager.RyuApp is inherited. Also, to use OpenFlow 1.3, the OpenFlow 1.3 version is specified for `OFP_VERSIONS`.

Also, MAC address table mac_to_port is defined.

In the OpenFlow protocol, some procedures such as handshake required for communication between the OpenFlow switch and the controller have been defined. However, because Ryu's framework takes care of those procedures, thus it is not necessary to be aware of those in Ryu applications.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    # ...
```

1.3.2 Event Handler

With Ryu, when an OpenFlow message is received, an event corresponding to the message is generated. The Ryu application implements an event handler corresponding to the message desired to be received.

The event handler defines a function having the event object for the argument and uses the `ryu.controller.handler.set_ev_cls` decorator to decorate.

set_ev_cls specifies the event class supporting the received message and the state of the OpenFlow switch for the argument.

The event class name is `ryu.controller.ofp_event.EventOFP` + <OpenFlow message name>. For example, in case of a Packet-In message, it becomes `EventOFPPacketIn`. For details, refer to Ryu's document titled API Reference . For the state, specify one of the following or list.

Definition	Explanation
ryu.controller.handler.HANDSHAKE_DISPATCHER	Exchange of HELLO message
ryu.controller.handler.CONFIG_DISPATCHER	Waiting to receive SwitchFeatures message
ryu.controller.handler.MAIN_DISPATCHER	Normal status
ryu.controller.handler.DEAD_DISPATCHER	Disconnection of connection

Adding Table-miss Flow Entry

After handshake with the OpenFlow switch is completed, the Table-miss flow entry is added to the flow table to get ready to receive the Packet-In message.

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

    # ...
```

In `ev.msg`, the instance of the OpenFlow message class corresponding to the event is stored. In this case, it is `ryu.ofproto.ofproto_v1_3_parser.OFPSwitchFeatures`.

In `msg.datapath`, the instance of the `ryu.controller.controller.Datapath` class corresponding to the OpenFlow switch that issued this message is stored.

The Datapath class performs important processing such as actual communication with the OpenFlow switch and issuance of the event corresponding to the received message.

The main attributes used by the Ryu application are as follows:

Attribute name	Explanation
id	ID (data path ID) of the connected OpenFlow switch.

ofproto	Indicates the ofproto module that supports the OpenFlow version in use. At this point, it is one of the following modules.
	`ryu.ofproto.ofproto_v1_0`
	`ryu.ofproto.ofproto_v1_2`
	`ryu.ofproto.ofproto_v1_3`

ofproto_parser	Same as ofproto, indicates the ofproto_parser module. At this point, it is one of the following modules.
	`ryu.ofproto.ofproto_v1_0_parser`
	`ryu.ofproto.ofproto_v1_2_parser`
	`ryu.ofproto.ofproto_v1_3_parser`

The main methods of the Datapath class used in the Ryu application are as follows:

send_msg(msg)

Sends the OpenFlow message. msg is a sub class of `ryu.ofproto.ofproto_parser.MsgBase` corresponding to the send OpenFlow message.

```
def switch_features_handler(self, ev):
    # ...

    # install table-miss flow entry
    #
    # We specify NO BUFFER to max_len of the output action due to
    # OVS bug. At this moment, if we specify a lesser number, e.g.,
    # 128, OVS will send Packet-In with invalid buffer_id and
    # truncated packet data. In that case, we cannot output packets
    # correctly.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

The Table-miss flow entry has the lowest (0) priority and this entry matches all packets. In the instruction of this entry, by specifying the output action to output to the controller port, in case the received packet does not match any of the normal flow entries, Packet-In is issued.

Note: As of January 2014, Open vSwitch does not fully support OpenFlow 1.3 and as with versions prior to OpenFlow1.3, Packet-In is issued by default. Also, the Table-miss flow entry is not supported at this time and is handled as a normal flow entry.

An empty match is generated to match all packets. Match is expressed in the `OFPMatch` class.

Next, an instance of the OUTPUT action class (`OFPActionOutput`) is generated to transfer to the controller port. The controller is specified as the output destination and `OFPCML_NO_BUFFER` is specified to max_len in order to send all packets to the controller.

Finally, 0 (lowest) is specified for priority and the `add_flow()` method is executed to send the Flow Mod message. The content of the add_flow() method is explained in a later section.

Packet-in Message

Create the handler of the Packet-In event handler in order to accept received packets with an unknown destination.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # ...
```

Frequently used OFPPacketIn class attributes are as follows:

Attribute name	Explanation
match	`ryu.ofproto.ofproto_v1_3_parser.OFPMatch` class instance in which the meta information of received packets is set.
data	Binary data indicating received packets themselves.
total_len	Data length of the received packets.
buffer_id	When received packets are buffered on the OpenFlow switch, indicates its ID. If not buffered, `ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER` is set.

```
def _packet_in_handler(self, ev):
    # ...

    in_port = msg.match['in_port']

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

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    # ...
```

Get the receive port (`in_port`) from the OFPPacketIn match. The destination MAC address and sender MAC address are obtained from the Ethernet header of the received packets using Ryu's packet library.

Based on the acquired sender MAC address and received port number, the MAC address table is updated.

In order to support connection with multiple OpenFlow switches, the MAC address table is so designed to be managed for each OpenFlow switch. The data path ID is used to identify OpenFlow switches.

Judging the Transfer Destination Port

The corresponding port number is used when the destination MAC address exists in the MAC address table. If not found, the instance of the OUTPUT action class specifying flooding (`OFPP_FLOOD`) for the output port is generated.

```
def _packet_in_handler(self, ev):
    # ...

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)
```

If the destination MAC address is found, an entry is added to the flow table of the OpenFlow switch.

As with addition of the Table-miss flow entry, specify match and action, and execute add_flow() to add a flow entry.

Unlike the Table-miss flow entry, set conditions for match this time. Implementation of the switching hub this time, the receive port (in_port) and destination MAC address (eth_dst) have been specified. For example, packets addressed to host B received by port 1 is the target.

For the flow entry this time, the priority is specified to 1. The greater the value, the higher the priority, therefore, the flow entry added here will be evaluated before the Table-miss flow entry.

Based on the summary including the aforementioned actions, add the following entry to the flow table.

> Transfer packets addressed to host B (the destination MAC address is B) received by port 1 to port 4.

Hint: With OpenFlow, a logical output port called NORMAL is prescribed in option and when NORMAL is specified for the output port, the L2/L3 function of the switch is used to process the packets. That means, by instructing to output all packets to the NORMAL port, it is possible to make the switch operate as a switching hub. However, we implement each processing using OpenFlow.

Adding Processing of Flow Entry

Processing of the Packet-In handler has not been done yet but here take a look at the method to add flow entries.

```
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]

    # ...
```

For flow entries, set match that indicates the target packet conditions, and instruction that indicates the operation on the packet, entry priority level, and effective time.

In the switching hub implementation, Apply Actions is used for the instruction to set so that the specified action is immediately used.

Finally, add an entry to the flow table by issuing the Flow Mod message.

```
def add_flow(self, datapath, port, dst, actions):
    # ...

    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)
    datapath.send_msg(mod)
```

The class corresponding to the Flow Mod message is the `OFPFlowMod` class. The instance of the OFPFlowMod class is generated and the message is sent to the OpenFlow switch using the datapath.send_msg() method.

There are many arguments of constructor of the OFPFlowMod class. Many of them generally can be the default as is. Inside the parenthesis is the default.

datapath

This is the Datapath class instance supporting the OpenFlow switch subject to flow table operation. Normally, specify the one acquired from the event passed to the handler such as the Packet-In message.

cookie (0)

An optional value specified by the controller and can be used as a filter condition when updating or deleting entries. This is not used for packet processing.

cookie_mask (0)

When updating or deleting entries, if a value other than 0 is specified, it is used as the filter of the operation target entry using the cookie value of the entry.

table_id (0)

Specifies the table ID of the operation target flow table.

command (ofproto_v1_3.OFPFC_ADD)

Specify whose operation is to be performed.

Value	Explanation
OFPFC_ADD	Adds new flow entries.
OFPFC_MODIFY	Updates flow entries.
OFPFC_MODIFY_STRICT	Update strictly matched flow entries
OFPFC_DELETE	Deletes flow entries.
OFPFC_DELETE_STRICT	Deletes strictly matched flow entries.

idle_timeout (0)

Specifies the validity period of this entry, in seconds. If the entry is not referenced and the time specified by idle_timeout elapses, that entry is deleted. When the entry is referenced, the elapsed time is reset.

When the entry is deleted, a Flow Removed message is sent to the controller.

hard_timeout (0)

Specifies the validity period of this entry, in seconds. Unlike idle_timeout, with hard_timeout, even though the entry is referenced, the elapsed time is not reset. That is, regardless of the reference of the entry, the entry is deleted when the specified time elapsed.

As with idle_timeout, when the entry is deleted, a Flow Removed message is sent.

priority (0)

Specifies the priority order of this entry. The greater the value, the higher the priority.

buffer_id (ofproto_v1_3.OFP_NO_BUFFER)

Specifies the buffer ID of the packet buffered on the OpenFlow switch. The buffer ID is notified in the packet-In message and when the specified processing is the same as when two messages are sent,

i.e., the Packet-Out message for which OFPP_TABLE is specified for the output port and Flow Mod message. This is ignored when the command is OFPFC_DELETE or OFPFC_DELETE_STRICT.

When the buffer ID is not specified, set `OFP_NO_BUFFER`.

out_port (0)

If the command is OFPFC_DELETE or OFPFC_DELETE_STRICT, the target entry is filtered by the output port. If the command is OFPFC_ADD, OFPFC_MODIFY, or OFPFC_MODIFY_STRICT, it is ignored.

To disable filtering by the output port, specify `OFPP_ANY`.

out_group (0)

As with out_port, filters by the output group.

To disable, specify `OFPG_ANY`.

flags (0)

You can specify the following combinations of flags.

Value	Explanation
OFPFF_SEND_FLOW_REM	Issues the Flow Removed message to the controller when this entry is deleted.
OFPFF_CHECK_OVERLAP	When the command is OFPFC_ADD, checks duplicated entries. If duplicated entries are found, Flow Mod fails and an error is returned.
OFPFF_RESET_COUNTS	Resets the packet counter and byte counter of the relevant entry.
OFPFF_NO_PKT_COUNTS	Disables the packet counter of this entry.
OFPFF_NO_BYT_COUNTS	Disables the byte counter of this entry.

match (None)

Specifies match.

instructions ([])

Specifies a list of instructions.

Packet Transfer

Now we return to the Packet-In handler and explain about final processing.

Regardless whether the destination MAC address is found from the MAC address table, at the end the Packet-Out message is issued and received packets are transferred.

```
def _packet_in_handler(self, ev):
    # ...

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

The class corresponding to the Packet-Out message is `OFPPacketOut` class.

The arguments of the constructor of OFPPacketOut are as follows:

datapath

Specifies the instance of the Datapath class corresponding to the OpenFlow switch.

buffer_id

Specifies the buffer ID of the packets buffered on the OpenFlow. If not buffered, `OFP_NO_BUFFER` is specified.

in_port

Specifies the port that received packets. if it is not the received packet, `OFPP_CONTROLLER` is specified.

actions

Specifies the list of actions.

data

Specifies the binary data of packets. This is used when `OFP_NO_BUFFER` is specified for buffer_id. When the OpenFlow switch's buffer is used, this is omitted.

In the switching hub implementation, buffer_id of the Packet-In message has been specified for buffer_id. If the buffer-id of the Packet-In message has been disabled, the received packet of Packet-In is specified for data to send the packets.

This is the end of explanation of the source code of switching hub. Next, let's execute this switching hub to confirm actual operation.

1.4 Execution of Ryu Application

When you log in to the SDN Hub VM provided for this ECE4110 lab (using user Ubuntu and password Ubuntu), use the `mn` command to start the Mininet environment.

The environment to be built has a simple structure with three hosts and one switch.

mn command parameters are as follows:

Parameter	Value	Explanation
topo	single,3	Topology of one switch and three hosts
mac	None	Automatically sets the MAC address of the host
switch	ovsk	Uses Open vSwitch
controller	remote	Uses external OpenFlow controller
x	None	Starts xterm

An execution example is as follows:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet>
```

When executing the command, five xterm start on the desktop PC. Each xterm corresponds to hosts 1 to 3, the switch and the controller.

Execute the command from the xterm for the switch to set the OpenFlow version to be used. The xterm for which the window title is “switch:s1 (root)” is the one for the switch.

First of all, let’s take a look at the status of Open vSwitch.

switch: s1:

```
root@ryu-vm:~# ovs-vsctl show
fdec0957-12b6-4417-9d02-847654e9cc1f
Bridge "s1"
    Controller "ptcp:6634"
    Controller "tcp:127.0.0.1:6633"
    fail_mode: secure
    Port "s1-eth3"
        Interface "s1-eth3"
    Port "s1-eth2"
        Interface "s1-eth2"
    Port "s1-eth1"
        Interface "s1-eth1"
    Port "s1"
        Interface "s1"
            type: internal
ovs_version: "1.11.0"
root@ryu-vm:~# ovs-dpctl show
system@ovs-system:
        lookups: hit:14 missed:14 lost:0
        flows: 0
        port 0: ovs-system (internal)
        port 1: s1 (internal)
        port 2: s1-eth1
        port 3: s1-eth2
        port 4: s1-eth3
root@ryu-vm:~#
```

Switch (bridge) *s1* has been created and three ports corresponding to hosts have been added.

Next, set 1.3 for the OpenFlow version.

switch: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
root@ryu-vm:~#
```

Let's check the empty flow table.

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
root@ryu-vm:~#
```

The ovs-ofctl command needs to specify the OpenFlow version to be used as an option. The default is *OpenFlow10*. Note that is a capital O after the minus.

1.4.1 Executing the Switching Hub

Preparation is now done and we will run the Ryu application.

From the xterm for which the window title is "controller: c0 (root)", execute the following commands.

controller: c0:

```
root@ryu-vm:~# cd /home/ubuntu/ryu && ./bin/ryu-manager –verbose
ryu.app.simple_switch_13 loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler instantiating
app ryu.app.simple_switch_13 instantiating app
ryu.controller.ofp_handler BRICK SimpleSwitch13
  CONSUMES EventOFPSwitchFeatures CONSUMES
  EventOFPPacketIn
BRICK ofp_event
  PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])} PROVIDES
  EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])} CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPHello CONSUMES
  EventOFPEchoRequest
  CONSUMES EventOFPPortDescStatsReply CONSUMES
  EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x2e2c050> address:('127.0.0.1', 53937)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2e2a550> move onto
config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xff9ad15b OFPSwitchFeatures(auxiliary_id=0,
capabilities=71,datapath_id=1,n_buffers=256,n_tables=254)
move onto main mode
```

It may take time to connect to OVS but after you wait for a while, as shown above...

```
connected socket:<....
hello ev ...
...
move onto main mode
```

„„is displayed.

Now OVS has been connected, handshake has been performed, the Table-miss flow entry has been added and the switching hub is in the status waiting for Packet-In.

Confirm that the Table-miss flow entry has been added.

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=105.975s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER
:65535
root@ryu-vm:~#
```

The priority level is 0, no match, and CONTROLLER is specified for action, and transfer data size of 65535(0xffff = OFPCML_NO_BUFFER) is specified.

1.4.2 Confirming Operation

Not yet, follow the instruction below but we are about to execute ping from host 1 to host 2.

1. ARP request

   At this point, host 1 does not know the MAC address of host 2, therefore, before ICMP echo request, an ARP request is supposed to be broadcast. The broadcast packet is received by host 2 and host 3.

2. ARP reply

   In response to the ARP, host 2 returns an ARP reply to host 1.

3. ICMP echo request

   Now host 1 knows the MAC address of host 2, host 1 sends an echo request to host 2.

4. ICMP echo reply

   Because host 2 already knows the MAC address of host 1, host 2 returns an echo reply to host 1.

Communications like those above are supposed to take place.

Before executing the ping command, execute the tcpdump command so that it is possible to check what packets were received by each host.

host: h1:

```
root@ryu-vm:~# tcpdump -en -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

host: h2:

```
root@ryu-vm:~# tcpdump -en -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

host: h3:

```
root@ryu-vm:~# tcpdump -en -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

Use the console where the mn command is executed first, execute the following command to issue ping from host 1 to host 2.

```
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=97.5 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 97.594/97.594/97.594/0.000 ms
mininet>
```

ICMP echo reply has returned normally.

First of all, check the flow table.

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=417.838s, table=0, n_packets=3, n_bytes=182, priority=0 actions=
CONTROLLER:65535
 cookie=0x0, duration=48.444s, table=0, n_packets=2, n_bytes=140, priority=1,in_port=2,dl_dst
=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=48.402s, table=0, n_packets=1, n_bytes=42, priority=1,in_port=1,dl_dst
=00:00:00:00:00:02 actions=output:2
root@ryu-vm:~#
```

In addition to the Table-miss flow entry, tow flow entries of priority level 1 have been registered.

1. Receive port (in_port):2, Destination MAC address (dl_dst):host 1 -> Action (actions):Transfer to port 1
2. Receive port (in_port):1, Destination MAC address (dl_dst): host 2 -> Action (actions): Transfer to port 2

Entry (1) was referenced twice (n_packets) and entry (2) was referenced once. Because (1) is a communication from host 2 to host 1, ARP reply and ICMP echo reply must have matched. (2) is a communication from host 1 to host 2 and because ARP request is broadcast, this is supposed to be by ICMP echo request.

Now, let's look at the log output of simple_switch_13.

controller: c0:

```
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
```

The first Packet-In is the ARP request issued by host 1 and is a broadcast, the flow entry is not registered and only Packet-Out is issued.

The second one is the ARP reply returned from host 2 and because its destination MAC address is host 1, the aforementioned flow entry (1) is registered.

The third one is the ICMP echo request sent from host 1 to host 2 and flow entry (2) is registered.

The ICMP echo reply returned from host 2 to host 1 matches the already registered flow entry (1) thus is transferred to host 1 without issuing Packet-In.

Finally, let's take a look at the output of tcpdump executed on each host.

host: h1:

```
root@ryu-vm:~# tcpdump -en -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
20:38:04.625473 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42:
Request who-has 10.0.0.2 tell 10.0.0.1, length 28
20:38:04.678698 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42:
Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
20:38:04.678731 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800), length 98:
10.0.0.1> 10.0.0.2: ICMP echo request, id 3940, seq 1, length 64
20:38:04.722973 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800), length 98:
10.0.0.2> 10.0.0.1: ICMP echo reply, id 3940, seq 1, length 64
```

Host 1 first broadcast the ARP request and then received the ARP reply returned from host 2. Next, host 1 issued the ICMP echo request and received the ICMP echo reply returned from host 2.

host: h2:

```
root@ryu-vm:~# tcpdump -en -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

```
20:38:04.637987 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42:
Request who-has 10.0.0.2 tell 10.0.0.1, length 28
20:38:04.638059 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42:
Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
20:38:04.722601 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800), length 98:
10.0.0.1> 10.0.0.2: ICMP echo request, id 3940, seq 1, length 64
20:38:04.722747 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800), length 98:
10.0.0.2> 10.0.0.1: ICMP echo reply, id 3940, seq 1, length 64
```

Host 2 received the ARP request issued by host 1 and returned the ARP reply to host 1. Then, host 2 received the ICMP echo request from host 1 and returned the echo reply to host 1.

host: h3:

```
root@ryu-vm:~# tcpdump -en -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
20:38:04.637954 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42:
Request who-has 10.0.0.2 tell 10.0.0.1, length 28
```

Host 3 only received the ARP request broadcast by host 1 at first.

You may now type exit in the he console where the mn command was executed first. This closes all the terminals.

Aside from: http://sdnhub.org/tutorials/ryu/

This repeats some of what is in the tutorial above, but it may help clarify things you do not yet understand.

RYU Code Structure

The main controller code is organized under the /ryu/ folder (In our VM – /home/ubuntu/ryu/ryu/). Here we discuss the functionalities of the key components. It is important to become familiar with them.

- **app**/ – Contains set of applications that run on-top of the controller.
- **base**/ – Contains the base class for RYU applications. The RyuApp class in the app_manager.py file is inherited when creating a new application.
- **controller**/ – Contains the required set of files to handle OpenFlow functions (e.g., packets from switches, generating flows, handling network events, gathering statistics etc).
- **lib**/ – Contains set of packet libraries to parse different protocol headers and a library for OFConfig. In addition, it includes parsers for Netflow and sFlow too.
- **ofproto**/ – Contains the OpenFlow protocol specific information and related parsers to support different versions of OF protocol (1.0, 1.2, 1.3, 1.4)
- **topology**/: Contains code that performs topology discovery related to OpenFlow switches and handles associated information (e.g., ports, links etc). Internally uses LLDP protocol.

RYU Controller Code Essentials

Most controller platforms expose some native features to allow these key features:

- Ability to listen to asynchronous events (e.g., PACKET_IN, FLOW_REMOVED) and to observe events using ryu.controller.handler.set_ev_cls decorator.
- Ability to parse incoming packets (e.g., ARP, ICMP, TCP) and fabricate packets to send out into the network
- Ability to create and send an OpenFlow/SDN message (e.g., PACKET_OUT, FLOW_MOD, STATS_REQUEST) to the programmable dataplane.

With RYU you can achieve all of those by invoking set of applications to handle network events, parse any switch request and react to network changes by installing new flows, if required. For instance, creating a new application involves creating a subclass of RyuApp and building the required logic to listen for network events.

```
from ryu.base import app_manager

class L2Forwarding(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Forwarding, self).__init__(*args, **kwargs)
```

While the above code represents a valid RYU application, it doesn't have the logic to handle network events from OpenFlow switches. Next, to allow an application to receive packets sent by the switch to the controller, the class needs to implement a method which is decorated by EventOFPPacketIn.

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

The first argument of the decorator calls this function everytime a *packet_in* message is received. The second argument indicates the switch state. Any information about the switch and the protocol version supported by the switch can be deciphered using the following:

```
    msg = ev.msg                # Object representing a packet_in data
structure.
    datapath = msg.datapath     # Switch Datapath ID
    ofproto = datapath.ofproto # OpenFlow Protocol version the entities
negotiated. In our case OF1.3
```

Once the packet is received, you can decode the packet by importing the packet library under /ryu/lib:

```
    from ryu.lib.packet import packet
    from ryu.lib.packet import ethernet
```

We can inspect the packet headers for several packet types: ARP, Ethernet, ICMP, IPv4, IPv6, MPLS, OSPF, LLDP, TCP, UDP. For set of packet types supported refer to this link

```
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
```

I use the following two useful commands to extract Ether header details:

```
    dst = eth.dst
    src = eth.src
```

Similarly, the OFPPacketOut class can be used to build a packet_out message with the required information (e.g., Datapath ID, associated actions etc)

```
    out =
ofp_parser.OFPPacketOut(datapath=dp,in_port=msg.in_port,actions=actions)
#Generate the message
    dp.send_msg(out) #Send the message to the switch
```

Besides a PACKET_OUT, we can also perform a FLOW_MOD insertion into a switch. For this, we build the Match, Action, Instructions and generate the required Flow. Here is an example of how to create a match header where the in_port and eth_dst matches are extracted from the PACKET_IN:

```
    msg = ev.msg
    in_port = msg.match['in_port']
    # Get the destination ethernet address
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    dst = eth.dst
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
```

There are several other fields you can match, which are defined in line 1130. The supported set of actions is defined in line 230 and the instructions are defined in line 195. Here is an example of creating an action list for the flow.

```
    actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)] # Build the
required action
```

OpenFlow 1.3 associates set of instructions with each flow entry such as handling actions to modify/forward packets, support pipeline processing instructions in the case of multi-table, metering

instructions to rate-limit traffic etc. The previous set of actions defined in OpenFlow 1.0 are one type of instructions defined in OpenFlow 1.3.

Once the match rule and action list is formed, instructions are created as follows:

```
    inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
```

Given the above code, a Flow can be generated and added to a particular switch.

```
    mod = parser.OFPFlowMod(datapath=datapath, priority=0, match=match,
instructions=inst)
    datapath.send_msg(mod)
```

Question 7:
To help further understand the information in the appendix, let's look at a modified version of the learning switch named simple_switch_13.py which we will call simple_switch_13_ece4110.py This file is on T-Square and you will need to download it. The version has some print statements and some additional comments included in it.

Use WinSCP to copy the file simple_switch_13_ece4110.py to a new directory you create named /home/Ubuntu/ryu/ryu/ece4110

(a) Put in your report the output from the RYU controller in window #2 below after running the ping below
(b) Explain what the output is and what it means.

To run this version of the learning switch:

- Run Mininet on a terminal window #1 using the following command. This starts a network emulation environment to emulate 1 switch with 3 hosts.

```
$ sudo mn --topo single,3 --mac --controller remote --switch ovsk
```

- Next, start the RYU Controller in a terminal window #2. Assume that the main folder where ryu is installed is in /home/ubuntu/ryu The following command starts the controller by initiating the OpenFlow Protocol Handler and simple_switch_13_ece4110.py application.

```
$ cd /home/ubuntu/ryu && ./bin/ryu-manager --verbose
ryu/ece4110/simple_switch_13_ece4110.py
```

h1 ping –c1 h2

Question 8:
This part of the lab is very challenging (but very important if you want a job in networking now days). It is intended for students who want to be able to tell an interviewer that they have written an SDN controller application and are ready for using Software Defined Networking. Starting with the RYU learning switch simple_switch_13_ece4110.py, implement a controller for a single- switch, two-host mininet topology. The requirements of the controller are as follows: The controller must forward all pings (and learn) just as the default learning switch module does. In addition, no TCP traffic will be allowed to flow except for http requests from h1 to h2 will be allowed, (but not in the other direction). It must also allow ssh connections to be established from h2 to h1, but not in the other direction. Once you have built your controller, run it with the specified two host mininet topology. Demonstrate that your controller meets the ECE4110 lab requirements by performing pings, http and ssh connections between h1 and h2. Finally, provide the source code of your controller in your lab report highlighting the changes to the original code and outputs showing that it worked. This is really challenging but really cool stuff!

Suggested testing methodology (show outputs in report):

sudo mn –c
sudo mn --topo single,2 --mac --controller remote --switch ovsk
xterm h1 h2

cd /home/ubuntu/ryu && ./bin/ryu-manager ryu/ece4110/firewall.py

The controller must forward all pings just as the default learning switch module does. To test:
On h1:
ping -c1 10.0.0.2

On h2:
ping -c1 10.10.0.1

In addition, the controller must allow http requests from h1 to h2, but not in the other direction.

This should be allowed:
On h2:
nc -l -p 80 (the l stands for listen and is a lower case L)

On h1:
telnet 10.0.0.2 80
Do I see this typed line echoed over at host 2?
Control] (type control key and then at same time the back bracket key)
telnet> quit

--or—on host 1:
nc 10.0.0.2 80
Type “Do I see this typed line echoed over at host 2?”

To test the other direction (This should not be allowed):
On h1:
nc -l -p 80
control c

On h2:
telnet 10.0.0.1 80
Note you will not get a connection
Control]
telnet> quit
--or--
nc 10.0.0.1 80
Note you will not get a connection
Control c

It must also allow ssh connections to be established from h2 to h1, but not in the other direction.

This should be allowed:
On h1:
/usr/sbin/sshd -D
--or--
nc -l 22

On h2:
ssh 10.0.0.1 -l ubuntu
exit close the remote connection
-or-
nc 10.0.0.1 22
Do I see this typed line echoed over at host 1?

To test the other direction (This should not be allowed):
On h2:
/usr/sbin/sshd -D
--or--
nc -l 22

On h1:
ssh 10.0.0.2 –l ubuntu
am not able to get a connection
-or-
nc 10.0.0.2 22
am not able to get a connection

Here is the starting firewall.py code on T-Square you should start with and modify:

```
##################################################
# ECE4110 Spring 2016
# firewall.py
# Your Name:
# Date:
##################################################
"""
An OpenFlow 3.0 firewall
"""


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

import logging
import struct
from ryu.controller import mac_to_port
from ryu.lib.mac import haddr_to_bin


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    # this function adds a default rule with low priority 0
    # to send any packets that do not have a table entry
    #to the controller
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    #this function sends the rule from the controller to the switch
    #for entry in the switch table
    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

    #This function handles a packet in event in the controller
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
```

```
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)
    msg = ev.msg

    datapath = msg.datapath

    ofproto = datapath.ofproto

    parser = datapath.ofproto_parser

    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)

    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src


    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    ######################################################################
    #Add your code here to:
    1) to create matches on TCP packets using parser.OFPMatch(fill in fields here to match on)
    2) create actions using [parser.OFPActionOutput(port to send to goes in here)]
    3) call self.add_flow(datapath, a priority value where larger value is higher priority, a match rule from your step one, an action from your
step 2
    ######################################################################

    # if not packet taken care of above with a higher priority rule
    # add a lower priority rule to learn a mac address to avoid FLOOD next time.
    # creating a table for mapping ports to mac for a datapath

    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        # verify if we have a valid buffer_id, if yes avoid to send both
        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow(datapath, 1, match, actions)

    # For ping, data = None
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)

    #Send the message to the switch
    datapath.send_msg(out)
```