CS498 Systems and Networking Lab **Spring 2012**

Lab 7: Software Defined Networking

Instructor: Matthew Caesar *Due:*

In this assignment you will learn the basics of Software Defined Networking, and a few of its applications. We will be working with Openflow, a software defined networking standard developed by Stanford University in 2008. You will use a simulator called Mininet to simulate a network topology that uses Openflow switches. Then you will write the control logic for those switches to manipulate packets.

1 FAQ

1. Command Line Notation: There are different command lines that we will be working in.

   `mininet> command`

   Commands that are prefixed with this are to be run in the Mininet console

   `$ command`

   Commands that are prefixed with this are to be run at your VM, under user priviliges

   `# command`

   Commands that are prefixed with this are to be run at your VM, under *sudo* priviliges

2 Openflow Introduction

OpenFlow is an open interface for remotely controlling the forwarding tables in network switches, routers, and access points. Upon this low-level primitive, researchers can build networks with new high-level properties. For example, OpenFlow enables more secure default-off networks, wireless networks with smooth handoffs, scalable data center networks, host mobility, more energy-efficient networks and new wide-area networks to name a few.

You can find more information on Openflow at: http://www.openflow.org

3 Logging into the Environment

We will again be using vSphere and our Ubuntu VMs for this assignment. Each of the VMs already has the prerequisite materials installed on it, which will speed up the process to get started. Reprinted below are the instructions for accessing your VM. Remember, as in Lab(s) 3 and 6, these VMs are assigned to a group, and you should work with that group on this assignment as well.

3.1 Accessing the Virtual Machine

We have access to a service called vSphere, which will provide us with a number of local Virtual Machines for you to use. It should be noted that you do not HAVE to use these VMs if you wish. If you have your own VM setup, or even a personal computer, you may use that. Please be wary though that, this lab was setup and written using a recent installation of Ubuntu. If you are using a different operating system, we cannot gaurentee everything will work correctly.
To access your VM that we have setup for you, please follow these instructions.

- Speak with the TA (mark) to get permissions to a VM setup. (It is possible you may have already done this...good work!)
- If you are on a *Windows* machine, and you are on campus, you can download the vSphere client from *https://csil-vcenter.ad.uiuc.edu/*.
- If you are not on a Windows machine, you can still use vSphere. The campus Windows Terminal servers already have vSphere installed. The remote desktop host is *ews-windows-ts.ews.illinois.edu*. Your usename will be *UofI\NetID*. Be careful of the UofI domain name...it is the only one that will work. (I have no idea why...) If you are using Linux, you can use the rdesktop command from the command line to remote into the windows terminal server.
  rdesktop ews-windows-ts.ews.illinois.edu.
- Launch the vSphere client and connect to *csil-vcenter.ad.uiuc.edu*. You username and password should be your Netid and AD password.
- If you see a 'home' screen, click the 'VMs and Templates' icon. This will show you all the VMs you have access to.
- Start your VM with the green arrow at the top.
- You can pull up an X-console using the icon that looks like a computer with an arrow pointing out of it.
- The default user account if cs498class and the password is cs498class. Please change this password *Immediately!*. This account has sudo access.
- From this console, you can use this machine as you would any other. Please keep in mind that vSphere and your VM will only be accessible *inside* the UIUC network, or via VPN.

4 Mininet

Openflow requires multiple components in order to be effective. Primarily, we need a network which consists of hosts, links, and switches. It is this network where we will manipulate network traffic. Instead of setting up this network physically, we are going to use a simulator which will setup a virtual topology for us, complete with links, virtual hosts, and virtual openflow switches.
Mininet and your controller application will need to be run side by side, as they interact with one another. When working with Mininet, do so in a seperate terminal window.

Figure 1 depicts the network topology we will use in our first expirements. You may enter the following command to start Mininet with this topology.

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

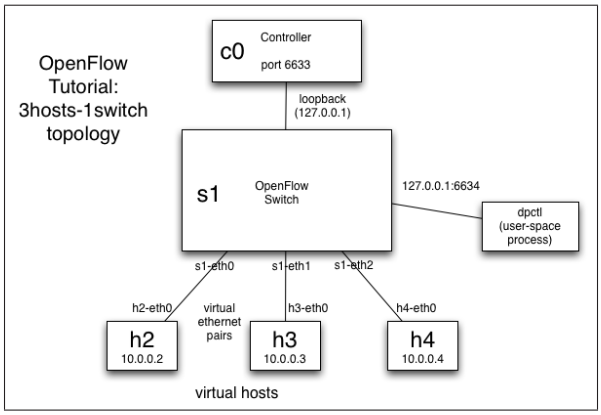

Figure 1: Mininet Topology

IF Mininet fails to start with the error FATAL: Module openvswitch_mod not found, run this command to fix it

```
/home/cs498class/mininet/util/install.sh -m
```

Command Breakdown:

- 'sudo mn': This starts mininet. Mininet always requires sudo to run
- '–topo single,3': This tells mininet to start using the topology of a 'single' switch and 3 hosts.
- '–mac': This tells mininet to assign each host a sequencial mac address, matching its IP address.
- '–switch ovsk': This tells mininet that the switches are to be of the type ovsk, this is the type for Openflow
- '–controller remote': This tells mininet that each Openflow switch is to talk to a controller, which is located at a remote location.

4.1 Mininet Basic Commands

Since you'll be working in Mininet for the whole tutorial, it's worth learning 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> h2 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 h2 h3
```

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.

5 dpctl

dpctl is a utility that comes with the OpenFlow reference distribution 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 (in your current setup this is 6634), 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:

```
$ dpctl show tcp:127.0.0.1:6634
```

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

Here's a more useful command:

```
$ dpctl dump-flows tcp:127.0.0.1:6634
```

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

Ping Test Now, go back to the mininet console and try to ping h3 from h2. In the Mininet console:

```
mininet> h2 ping -c3 h3
```

Note that the name of host h3 is automatically replaced when running commands in the Mininet console with its IP address (10.0.0.3).

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 dpctl to manually install the necessary flows. In your SSH terminal:

```
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
$ dpctl add-flow tcp:127.0.0.1:6634 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

```
$ dpctl dump-flows tcp:127.0.0.1:6634
```

Run the ping command again. In your mininet console:

```
mininet> h2 ping -c3 h3
```

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? NOTE: if you didn't see any ping replies coming through, it might be the case that the flow-entries expired before you start your ping test. When you do a "dpctl dump-flows" you can see an "idle_timeout" option for each entry, which defaults to 60s. This means that the flow will expire after 60secs if there is no incoming traffic. Run again respecting this limit, or install a flow-entry with longer timeout.

```
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=1,idle_timeout=120,actions=output:2
```

6 Work with the provided hub

Now, let's move on building a networking application. We provide you with starter code for a hub controller. After getting yourself familiar with it, you'll modify the provided hub to act as an L2 learning switch. In this application, the switch will examine each packet and learn the source-port mapping. Thereafter, the source MAC address will be associated with the port. If the destination of the packet is already associated with some port, the packet will be sent to the given port, else it will be flooded on all ports of the switch.

Later, you'll turn this into a flow-based switch, where seeing a packet with a known source and dest causes a flow entry to get pushed down.

NOX is an open-source platform that simplifies the creation of software for controlling or monitoring networks. Programs written within NOX (using either C++ or Python) have flow-level control of the network. This means that they can determine which flows are allowed on the network and the path they take.

You do not need to install NOX - it comes ready to run on the VM.

First, make sure Mininet is running under the correct settings:

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

6.1 Start the Hub Controller

Go to the directory holding the built NOX executable (/noxcore/build/src) in the other SSH window:

```
$ cd ~/noxcore/build/src
```

Then, in the same window, start the base Python hub code:

```
$ ./nox_core -v -i ptcp: pytutorial
```

This command told NOX to start the 'tutorial' application, to print verbose debug information, and to passively listen for new switch connections on the standard OpenFlow port (6633).

The switches may take a little bit of time to connect. When an OpenFlow switch loses its connection to a controller, it will generally increase the period between which it attempts to contact the controller, up to a maximum of 15 seconds. Since the OpenFlow switch has not connected yet, this delay may be anything between 0 and 15 seconds. If this is too long to wait, the switch can be configured to wait no more than N seconds using the –max-backoff parameter. Alternately, you exit Mininet to remove the switch(es), start the controller, and then start Mininet to immediately connect.

Wait until the application indicates that the OpenFlow switch has connected. When the switch connects, NOX will print something like this:

```
00039|nox|DBG:Registering switch with DPID = 1
```

If you see the switch print out a message like "sent error in response to capability reply, assuming no management support", this is OK. Open vSwitch has custom extensions to support a management database, but we're not enabling them on our OpenFlow switch.

6.2 Benchmark the Hub Controller

Here, you'll benchmark the provided hub.

First, verify reachability. Mininet should be running, along with the NOX tutorial in a second window. In the Mininet console, run:

```
mininet> pingall
```

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

```
mininet> iperf
```

What sort of speed are you seeing? The bandwidth of the virtual links in Mininet are very fast, so that is not a reason for slow speeds. Can you give a couple reasons, based on what you now know about Openflow and this hub, as to the slow speeds?

7 Create a basic Learning Switch

Next, we will take the provided hub and turn it into a learning switch. This can be done with a few simple Python commands, and is done to show the power of Openflow and NOX in working with router logic. Open Hub Code and Begin Go to your SSH terminal and stop the NOX hub controller using Ctrl-C. The file you'll modify is ~/noxcore/src/nox/coreapps/tutorial/pytutorial.py. Open this file in your favorite editor.

Most of the code will go in one function, learn_and_forward(). There isn't much code here, yet it's sufficient to make a complete hub and serve as an example of a minimal NOX app. You'll need to add roughly 10 lines to make a learning switch.

Each time you change and save this file, make sure to restart NOX, then use pings to verify the behavior of the combination of switch and controller as a (1) hub, (2) controller-based Ethernet learning switch, and (3) flow-accelerated learning switch.

In case you need some help with python, here are some basic Python resources:

http://docs.python.org/lib/built-in-funcs.html
http://docs.python.org/tutorial/

The subsections below give details about NOX APIs that should prove useful in the exercise.

7.1 Sending OpenFlow messages with NOX

We list a few functions here, that will be useful for your first steps into NOX.

```
Component.send_openflow( ... ) # send a packet out a port
Component.install_datapath_flow( ... ) # push a flow to a switch
```

These functions, part of the core NOX API, are defined in /noxcore/src/nox/lib/core.py. To save the need to look through this source code, the relevant documentation is below:

send_openflow(...)

```
def send_openflow(self, dp_id, buffer_id, packet, actions,
                  inport=openflow.OFPP_CONTROLLER):
   """
   Sends an openflow packet to a datapath.

   This function is a convenient wrapper for send_openflow_packet
   and send_openflow_buffer for situations where it is unknown in
   advance whether the packet to be sent is buffered.  If
   'buffer_id' is -1, it sends 'packet'; otherwise, it sends the
   buffer represented by 'buffer_id'.

   dp_id - datapath to send packet to
   buffer_id - id of buffer to send out
   packet - data to put in openflow packet
   actions - list of actions or dp port to send out of
   inport - dp port to mark as source (defaults to Controller
            port)
   """
```

Here's an example use, from the pytutorial.py starter code:

```
self.send_openflow(dpid, bufid, buf, openflow.OFPP_FLOOD, inport)
```

This code floods a packet cached at the switch (with the given bufid) out all ports but the input port. Replace openflow.OFPP_FLOOD with a port number to send a packet out a specific port, unmodified.

install_datapath_flow(....)

```
    def install_datapath_flow(self, dp_id, attrs, idle_timeout, hard_timeout,
                              actions, buffer_id=None,
                              priority=openflow.OFP_DEFAULT_PRIORITY,
                              inport=None, packet=None):
        """
        Add a flow entry to datapath

        dp_id - datapath to add the entry to

        attrs - the flow as a dictionary (described above)

        idle_timeout - # idle seconds before flow is removed from dp

        hard_timeout - # of seconds before flow is removed from dp

        actions - a list where each entry is a two-element list representing
        an action.  Elem 0 of an action list should be an ofp_action_type
        and elem 1 should be the action argument (if needed). For
        OFPAT_OUTPUT, this should be another two-element list with max_len
        as the first elem, and port_no as the second

        buffer_id - the ID of the buffer to apply the action(s) to as well.
        Defaults to None if the actions should not be applied to a buffer

        priority - when wildcards are present, this value determines the
        order in which rules are matched in the switch (higher values
        take precedence over lower ones)

        packet - If buffer_id is None, then a data packet to which the
        actions should be applied, or None if none.

        inport - When packet is sent, the port on which packet came in as input,
        so that it can be omitted from any OFPP_FLOOD outputs.
        """
```

Note that install_datapath_flow() takes in an attributes dictionary with parts of the OpenFlow match. Here's an example:

```
attrs = {}
attrs[core.IN_PORT] = inport
attrs[core.DL_DST] = packet.dst
```

install_datapath_flow also requires a list of actions. Here's another example:

```
actions = [[openflow.OFPAT_OUTPUT, [0, outport]]]
```

You will want to use exactly this action list for the tutorial. The format is a list of actions; we've defined one action, which forwards to a single port (OFPAT = OpenFlow Action Type: Output). The [0, outport] part that follows is the set of parameters for the output action type: 0 is max_len, which is only defined for packets forwarded to the controller (ignore this), while the outport param specifies the output port.

The priority shouldn't matter, unless you have overlapping entries. For example, the priority field could be:

```
openflow.OFP_DEFAULT_PRIORITY
```

For more details on the format, see the NOX core Python API code in src/nox/core.py.

For more information about OpenFlow constants, see the main OpenFlow types/enums/structs file, openflow.h, in /openflow/include/openflow/openflow.h

Parsing Packets with the NOX packet libraries

The NOX packet parsing libraries are automatically called to parse a packet and make each protocol field available to Python.

The parsing libraries are in:

```
~/noxcore/src/nox/lib/packet/
```

Each protocol has a corresponding parsing file.

For the first exercise, you'll only need to access the Ethernet source and destination fields. To extract the source of a packet, use the dot notation:

```
packet.src
```

The Ethernet src and dst fields are stored as arrays, so you'll probably want to to use the mac_to_str() function or mac_to_int. I suggest mac_to_str, and it avoids the need to do any hex conversion when printing. The mac_to_str() function is already imported, and comes from packet_utils in /noxcore/src/nox/lib/packet/packet_utils.py

To see all members of a parsed packet object:

```
print dir(packet)
```

Here's what you'd see for an ARP packet:

```
['ARP_TYPE', 'IP_TYPE', 'LLDP_TYPE', 'MIN_LEN', 'PAE_TYPE', 'RARP_TYPE',
 'VLAN_TYPE', '__doc__', '__init__', '__len__', '__module__', '__nonzero__',
 '__str__', 'arr', 'dst', 'err', 'find', 'hdr', 'hdr_len', 'msg', 'next', 'parse',
 'parsed', 'payload_len', 'prev', 'set_payload', 'src', 'tostring', 'type',
 'type_parsers']
```

Many fields are common to all Python objects and can be ignored, but this can be a quick way to avoid a trip to a function's documentation.

Using these specifications, first convert the Hub into a packet-based learning switch. That is to say, the controller should attempt to learn the source of each packet, and forward it on to its destination or flood it out if the location of the destination is unknown.
Once your learning switch is finished, re-run 'iperf' on it and measure its throughput.
Next, convert your packet-based switch into a flow-based switch. That means, when a packet arrives, attempt to learn the source. If a destination is known, push a 'Flow' entry down to the switch and then forward the packet. This will allow future packets that match to automatically be forwarded without being sent to the controller for analysis. Reference the install_datapath_flow command above.
Run 'iperf' on this new version of the learning switch and compare its performance to the hub and packet-based switch.

8 Support Multiple Switches

Your controller so far has only had to support a single switch, with a single MAC table. In this section, you'll extend it to support multiple switches.

Start mininet with a different topology. In the Mininet console:

```
mininet> exit
$ sudo mn --topo linear --switch ovsk --controller remote
```

The topology you created is shown in Figure 2.

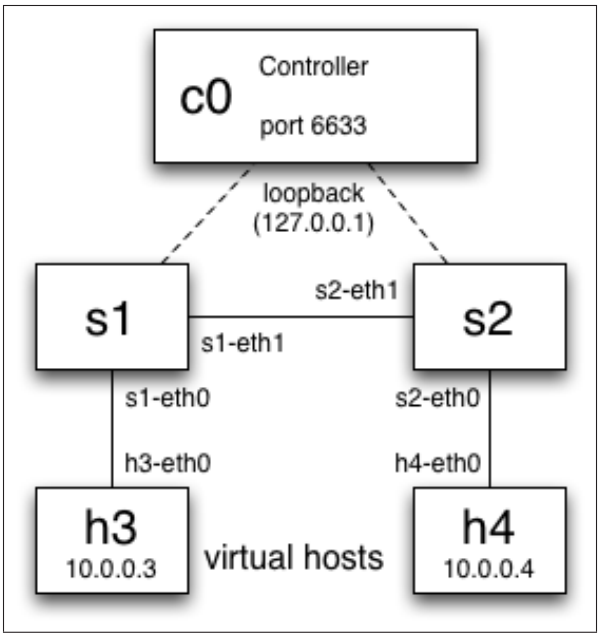

Figure 2: Mininet Topology, Multiple Switches

This will create a 2-switch topology where each switch has a single connected host.

Now, modify your switch so that it stores a MAC-to-port table for each DPID. The DPID (if you haven't already figured it out) is the unique identifier for each Openflow Switch. This strategy only works on spanning tree networks, and the delay for setting up new paths between far-away hosts is proportional to the number of switches between them. Other modules could act smarter. For example, one can maintain an all-pairs shortest path data structure and immediately push down all flow entries needed to previously-seen sources and destinations. This is out-of-scope for this assignment.

After the mods, to verify that your controller works, in the Mininet console, run:

```
mininet> pingall
```

9 Report

For you report for this lab, please include the following materials and answer the following questions:

1. Please include the source code for you Packet-Based learning switch as well as your Flow-Based learning switch
2. Please include the performance of the Hub controller, Packet-based switch controller, and Flow-based switch controller.
3. Openflow and NOX can handle a lot more than just switching packets, as we did in the lab. Think of at least 2 examples of other tasks you could make Openflow do.
4. As researchers and networking students, why do you think Software Defined Networking is an important topic? What can we do with Openflow that we aren't able to do with traditional off the shelf switches?
5. For companies with large infrastructures, what reasons would they have to migrate to a software defined network, as opposed to the traditional hardware based one?
6. Are there any issues with SDN, as we have presented it here, that might impede its acceptance? How should those issues be addressed?