Search

- Home
- Videos
- Documents
- News
- Research
- About

Views

- Page
- Discussion
- View source
- History

OpenFlow Tutorial

From OpenFlow Wiki

Jump to: navigation, search

Welcome to the OpenFlow tutorial!

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.

This tutorial is your opportunity to gain hands-on experience with the platforms and debugging tools most useful for developing network control applications on OpenFlow.

Innovate in your network!

After completing this tutorial, please fill out the feedback form.

Active ONS Tutorial Slides (4/16/2012):

- Main Slides pptx, pdf
- Virtualization
- Controller Showdown
- Deployment Experiences

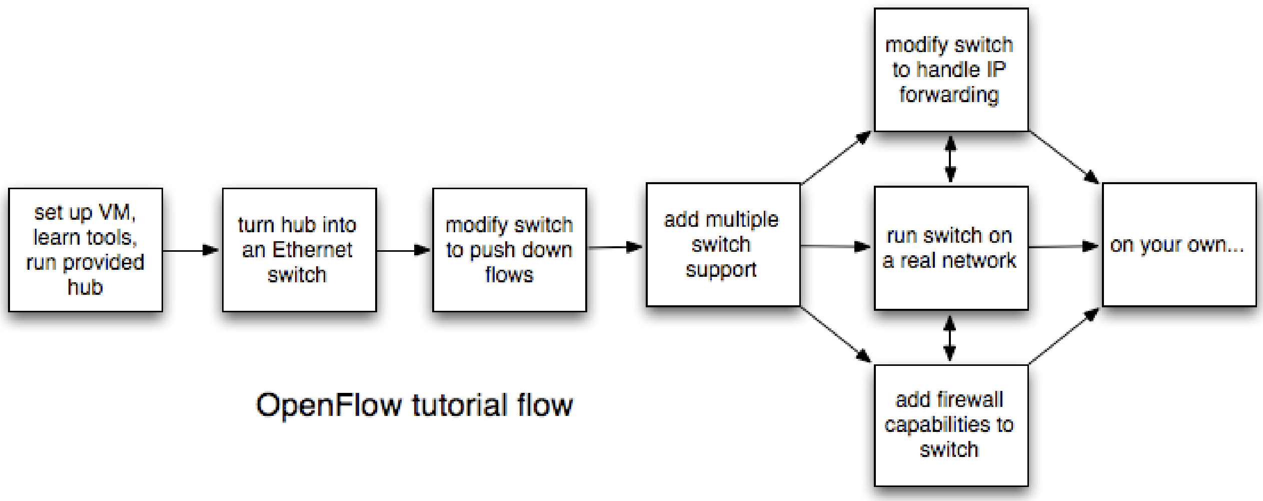
Archived Tutorial Slides:

- OpenNetSummit Tutorial (10/19/2011)
  - Main Slides: pdf, pptx
  - Deployment Forum:
    - Johan van Reijendam (Stanford): tgz
    - David Erickson (Stanford): pptx
    - Subhasree Mandal (Google): pdf

Contents

[hide]

Overview


set up VM, learn tools, run provided hub
turn hub into an Ethernet switch
modify switch to push down flows
add multiple switch support
modify switch to handle IP forwarding
run switch on a real network
add firewall capabilities to switch
on your own...
OpenFlow tutorial flow

In this tutorial, you'll turn the provided hub controller into a controller-based learning switch, then a flow-accelerated learning switch, and extend this from a single-switch network to a multiple-switch multiple-host network. You can also find guidelines for further extensions. Along the way, you'll learn the full suite of OpenFlow debugging tools. You will:

- view flow tables with dpctl
- dissect OpenFlow message with Wireshark
- simulate a multi-switch, multi-host network with Mininet
- benchmark your controller with cbench

After the tutorial, you can apply what you've learned to physical networks based on software switches, NetFPGAs, or even hardware switches at line rate.

To get you started quickly, we provide a preconfigured virtual machine with the needed software.

Pre-requisites

You will need a computer with at least 1GB (preferably 2GB+) of RAM and at least 5GB of free hard disk space (more preferred). A faster processor may speed up the virtual machine boot time, and a larger screen may help to manage multiple terminal windows.

These instructions consider Linux, OS X, and Windows. Linux and OS X are preferred - there's less to install.

You will need administrative access to the machine.

The tutorial instructions require no prior knowledge of OpenFlow. The OpenFlow Learn More page is a concise introduction.

Stuck? Found a bug? Questions?

Email [openflow-discuss - at - lists.stanford.edu] if you're stuck, think you've found a bug, or just want to send some feedback.

If you are asking a question or reporting a bug, try to include as many details about your setup as possible. Include your OS, virtualization software info, X11 and ssh software you're using, VM image you're using, memory size, and the step you're on.

Install Required Software

You will need to download these files individually.

The files include virtualization software, a SSH-capable terminal, an X server, and the VM image.

The tutorial image is distributed as a compressed VirtualBox image (vdi). VirtualBox enables you to run a virtual machine inside a physical machine, and is free and available for Windows, Mac and Linux. You can export the VirtualBox image to vmdk format and use it with VMWare using the instructions below.

The following instructions assume the use of VirtualBox, but the instructions should apply regardless of virtual software after you complete the initial setup.

Download Files

You'll need to download the files corresponding to your OS, plus the tutorial VM.

Start now with downloading a compressed VM image:

- Virtual Machine Image (OVF format, 64-bit, Mininet 2.0)

Important: For this VM image, the user name is 'mininet' with password 'mininet'.

The OVF format can be imported into VirtualBox, VMware, or other popular virtualization programs.

If this does not work for you, you can also try our older VM image that uses Mininet 1.0:

- VirtualBox VM Image (zipped VM image, 32-bit, Mininet 1.0)

Important: For this VM image, the user name is 'openflow' with password 'openflow'.

The download will take a while - it's on the order of 2GB in size.

You will also need virtualization software, an X server, and an ssh-capable terminal emulator:

OS Type	OS Version	Virtualization Software	X Server	Terminal
Windows	7+	VirtualBox	Xming	PuTTY
Windows	XP	VirtualBox	Xming	PuTTY
Mac	OS X 10.7-10.8 Lion/Mountain Lion	VirtualBox	download and install XQuartz	Terminal.app (built in)
Mac	OS X 10.5-10.6 Leopard/Snow Leopard	VirtualBox	X11 (install from OS X main system DVD, preferred), or download XQuartz	Terminal.app (built in)
Linux	Ubuntu 10.04+	VirtualBox	X server already installed	gnome terminal + SSH built in

Install and Verify

After you have downloaded the appropriate software and VM images, make sure that each column item (X server, Virtualization software, and SSH terminal) is installed and working for your platform, and that the VM image loads and runs correctly for your configuration.

Set up Virtual Machine

Import Virtual Machine Image

If you downloaded the .ovf image,

- **Start up VirtualBox, then select File>Import Appliance and select the .ovf image that you downloaded.**

You may also be able to simply double-click the .ovf file to open it up in your installed virtualization program.

- **Next, press the "Import" button.**

This step will take a while - the unpacked image is about 3 GB.

- **Continue with Finish VM Setup below.**

Note: If you downloaded the older .zip archive instead of the .ovf image, setup requires a few more steps:

OS X Users: Double-click the virtual machine image to extract it.

Linux Users (and OS X users who prefer the command line): from a terminal, unzip the virtual machine image, e.g.:

```
$ unzip OpenFlowTutorial-101311.zip
```

(If you downloaded a different VM image, make sure you use its correct file name.)

Windows Users: unzip the image in Windows Explorer.

For the .zip archive, you need to set up a new VirtualBox VM. Open VirtualBox.

- Select New
- Press Continue in the next prompt.
- Name your VM OpenFlowTutorial, Operating System Linux, Version Ubuntu. Click Continue.
- Set the memory at 512MB and click Continue
- At that point VB should ask you to Create a new hard disk, or use the existing one. Select "Use existing hard disk".
- Click the icon to select the hard disk. This will open the Virtual Media Manager Window.
- Press Add, and find the extracted OpenFlowTutorial*.vdi from the previous steps. Click Select and then Continue.
- Your VM installation is complete. Press Done.

Finish VM Setup

You will need to complete **one more step before you are done with the VM setup**.

Select your VM and go to the Settings Tab. Go to Network->Adapter 2. Select the "Enable adapter" box, and attach it to "host-only network".(**Sidenote**: on a new VirtualBox installation you may not have any "host-only network" configured yet. To have one select File menu/Preferences/Network and "Add host-only network" button with default settings. Then you can try the attach.) **This will allow you to easily access your VM through your host machine.**

At that point you should be ready to start your VM. Press the "Start" arrow icon or double-click your VM within the VirtualBox window.

In the VM console window, log in with the user name and password for your VM.

Note that this user is a sudoer, so you can execute commands with root permissions by typing `sudo` *`command`*, where *`command`* is the command you wish to execute with root permission.

Choose Preferred Editor

Nano, Vim, Emacs, and Gedit come installed on the OpenFlowTutorial VM. Brief instructions for each:

Nano: You can immediately modify a file. When you're done, hit 'ctrl-x', then say 'Yes' to the prompt, to save and quit.

Vim: to modify a file, type 'i' to enter Insert mode, then use the arrow keys to navigate and edit. When you're done, hit 'esc', type ':wq', then press enter, to save and quit.

Highly recommended for the NOX tutorial: add the following to ~/.vimrc in the VM:

```
set tabstop=4
set expandtab
```

Emacs: you can immediately modify a file. When you're done, hit 'ctrl-x', 'ctrl-s', then hit 'ctrl-x', 'ctrl-c' to exit.

Gedit: a graphical text editor, no instructions needed.

Eclipse: Eclipse and its dependencies would require about 500MB extra space on the VM image, so it's not shipped by default. If you have Eclipse installed on the host VM, using the Remote Systems Explorer can be a convenient way to access and modify text files on the VM, with many of the advantages of Eclipse, such as syntax highlighting.

If you have another preferred text editor, feel free to install it now:

```
$ sudo apt-get install <editor>
```

Command Prompt Notes

In this tutorial, commands are shown along with a command prompt to indicate what subsystem they are intended for . For example,

```
$ ls
```

indicates that the `ls` command should be typed at a Unix (e.g. Linux or OS X) command prompt (which generally ends in `$` if you are a regular user or `#` if you are root.

Other prompts used in this tutorial include

```
mininet>
```

for commands entered in the Mininet console and

```
C:>
```

for code entered into a Windows command window.

Set Up Network Access

The tutorial VM is shipped without a desktop environment, to reduce its size. All the exercises will be done through X forwarding, where programs display graphics through an X server running on the host OS.

To start up the X forwarding, you'll first need to find the guest IP address.

VirtualBox

If you are running VirtualBox, you should make sure your VM has **two network interfaces**. One should be a **NAT interface** that it can use to access the Internet, and the other should be a **host-only interface** to enable it to communicate with the host machine. For example, your NAT interface could be eth0 and have a 10.x IP address, and your host-only interface could be eth1 and have a 192.168.x IP address. You should ssh into the **host-only interface** at its associated IP address. Both interfaces should be configured using DHCP. If they are not already configured, you may have to run dhclient on each of them, as described below.

Access VM via SSH

In this step, you'll verify that you can connect from the host PC (your laptop) to the guest VM (OpenFlowTutorial) via SSH.

From the virtual machine console, log in to the VM, then enter:

```
$ ifconfig -a
```

You should see three interfaces(eth0, eth1, lo), Both eth0 and eth1 should have IP address assigned. If this is not the case, type

```
$ sudo dhclient ethX
```

Replacing ethX with the name of a downed interfaces; sometimes the eth ports appear as eth2 or eth3, you can fix this by editing /etc/udev/rules.d/70-persistent-net.rules and removing the existing configuration lines.

Note the IP address (probably the 192.168 one) for the **host-only** network; you'll need it later. Next, log in, which will depend on your OS.

Mac OS X and Linux

Open a terminal (Terminal.app in Mac, Gnome terminal in Ubuntu, etc). In that terminal, run:

```
$ ssh -X [user]@[Guest IP Here]
```

Replace [user] with the correct user name for your VM image.

Replace [Guest IP Here] with the IP you just noted. If ssh does not connect, make sure that you can ping the IP address you are connecting to.

Enter the password for your VM image. Next, try starting up an X terminal using

```
$ xterm
```

and a new terminal window should appear. If you have succeeded, you are done with the basic setup. Close the xterm. If you get a 'xterm: DISPLAY is not set error', verify your X server installation from above.

Windows

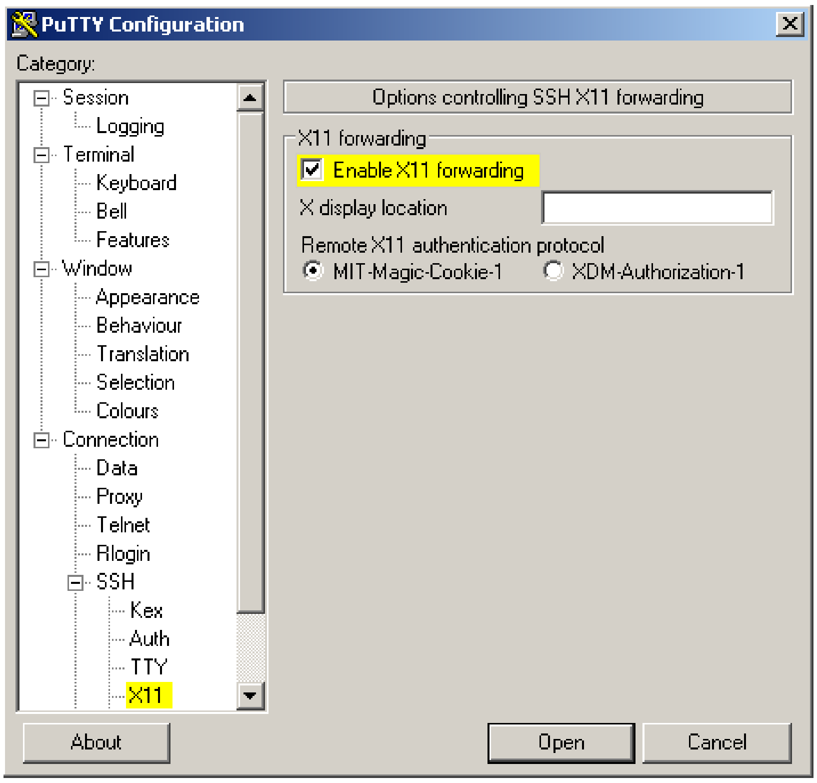
In order to use X11 applications such as xterm and wireshark, the Xming server must be running, and you must make an ssh connection with X11 forwarding enabled.

First, start Xming (e.g. by double-clicking its icon.) No window will appear, but if you wish you can verify that it is running by looking for its process in Windows' task manger.

Second, make an ssh connection with X11 forwarding enabled.

If you start up puTTY as a GUI application, you can connect by entering your VM's IP address and enabling X11 forwarding.

To enable X11 forwarding from puTTY's GUI, click puTTY->Connection->SSH->X11, then click on Forwarding->"Enable X11 Forwarding", as shown below:


PuTTY Configuration

Category:

Options controlling SSH X11 forwarding

X11 forwarding

[✓] Enable X11 forwarding

X display location

Remote X11 authentication protocol

[● MIT-Magic-Cookie-1] XDM-Authorization-1

You can also run putty (with the -X option for X11 forwarding) from the Windows command line:

Open a terminal: click the Windows 'Start' button, 'run', then enter 'cmd'.

Change to the directory where you saved putty.

```
C:> cd <dir>
```

Run:

```
C:> putty.exe -X openflow@[Guest IP Here]
```

Replace [Guest IP Here] with the IP you just noted.

If putty cannot connect, try pinging the VM's IP address to make sure you are connecting to the correct interface.

```
C:> ping [Guest IP Here]
```

Once the ssh connection succeeds or a terminal window for the VM pops up, log in to the VM. Now, type:

```
$ xterm -sb 500
```

to start an X terminal (the -sb 500 is optional but gives 500 lines of scrollback.)

A white terminal window should appear. If you have succeeded, you are done with the basic setup. Close the xterm.

If the xterm window does not appear, or if you get an error like "xterm: DISPLAY is not set," make sure that Xming is running in Windows and that you have correctly enabled X11 forwarding.

Alternative: Run X11 in the VM console window

As an alternative to running X11 on your host machine, you may find it useful or convenient to install X11 into the VM itself. To install X11 and a simple window manager, log in to the **VM console window** - *not* via an ssh session! - using the correct user name and password for your VM, and type:

```
$ sudo apt-get update && sudo apt-get install xinit flwm
```

At this point, you should be able to start an X11 session in the VM console window by typing:

```
$ startx
```

If you are familiar with Linux, you may wish to install another desktop manager (e.g. `lxde` or even the full `ubuntu-desktop`) or any other GUI packages that you prefer.

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.

Let's define some terminology, starting with terminal types:

- **VirtualBox** console terminal: connects to OpenFlowTutorial. This is the one created when you started up the VM. You can't copy and paste from this tutorial page to the console terminal, so it's a bit of a pain. **Minimize this NOW, if you haven't already done so**. Once you've used it to set up networking, it won't be needed.
- **SSH** terminal: connects to OpenFlowTutorial. Created by using putty on Windows or SSH on OS X / Linux, as described in the previous section. Copy and paste should work on this terminal.
- **xterm** terminal: connects to a host in the virtual network. Created in the next section when you start up the virtual network. Will be labeled at the top with the name of the host.

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 NOX or Beacon (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.
- **dpctl**: command-line utility that sends quick OpenFlow messages, useful for viewing switch port and flow stats, plus 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) in a

conveniently readable way.

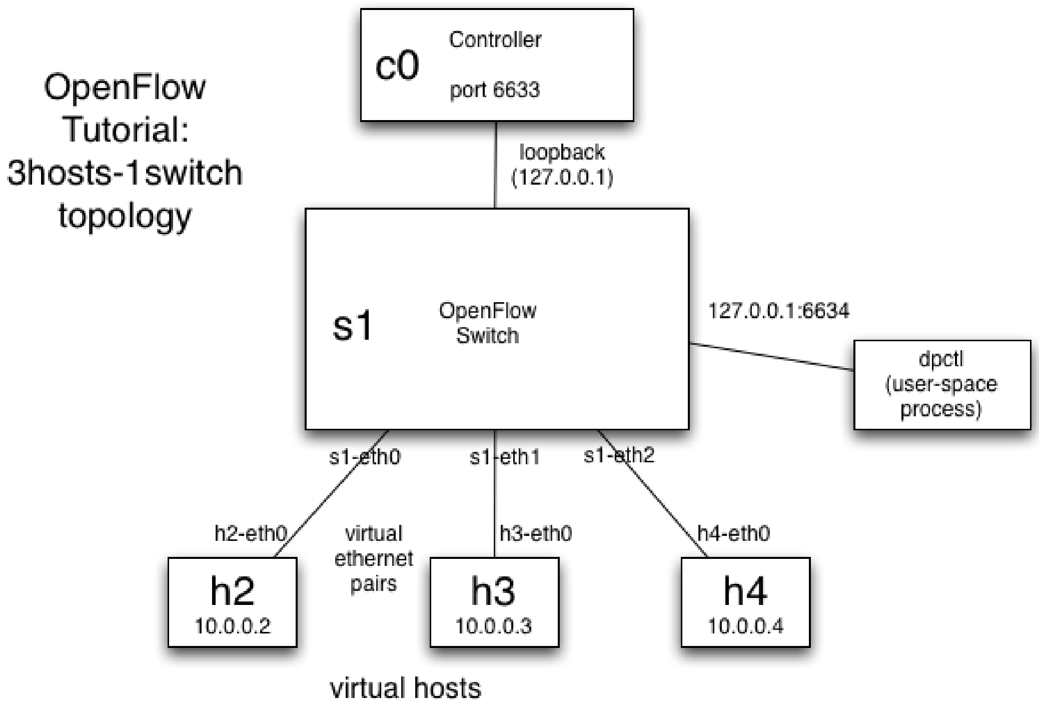
- **iperf**: general command-line utility for testing the speed of a single TCP connection.
- **Mininet**: network emulation platform. Mininet creates a virtual OpenFlow network - controller, switches, hosts, and links - on a single real or virtual machine. More Mininet details can be found at the Mininet 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.

Mininet Brief Intro

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> 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 SSH terminal (normal user) and `#` is for SSH terminal (root user) (See Command Prompt Notes). 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.

dpctl Example Usage

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 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 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> 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? 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
```

Start Wireshark

The VM image includes the OpenFlow Wireshark dissector pre-installed. Wireshark is extremely useful for watching OpenFlow protocol messages, as well as general debugging.

Start a new SSH terminal and connect to the VM with X11 forwarding.

(Reminder: here are the command-line commands to do so:

If you're using MAC OS X or Linux, enter:

```
$ ssh -X openflow@[guest ip address]
```

If you're using putty.exe from the Windows command-lineshell, enter:

```
C:> putty.exe -X openflow@[guest ip address]
```

If you're using putty's GUI, make sure X11 forwarding is enabled.)

Now open Wireshark:

```
$ sudo wireshark &
```

You'll probably get a warning message for using wireshark with root access. Press OK.

Click on Capture->Interfaces in the menu bar. Click on the Start button next to 'lo', the loopback interface. You may see some packets going by.

Now, set up a filter for OpenFlow control traffic, by typing 'of' in Filter box near the top:

```
of
```

Press the apply button to apply the filter to all recorded traffic.

Start Controller and view Startup messages in Wireshark

Now, with the Wireshark dissector listening, start the OpenFlow reference controller. In your SSH terminal:

```
$ controller ptcp:
```

This starts a simple controller that acts as a learning switch without installing any flow-entries.

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.
Port Status	Switch->Controller	enables the switch to inform that controller of changes to port speeds or connectivity. Ignore this one, it appears to be a bug.

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.

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:

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

Run a ping to view the OpenFlow messages being used. 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.

First, you see an ARP request miss the flow table, which generates a broadcast Packet-Out message. Next, the ARP response comes back; with both MAC addresses now known to the controller, it can push down a flow to the switch with a Flow-Mod message. The switch does then pushes flows for the ICMP packets. 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 was no 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.

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.

Now compare with the user-space switch. In the mininet console:

```
mininet> exit
```

Start the same Mininet with the user-space switch:

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

Run one more iperf test with the reference controller:

```
mininet> iperf
```

See a difference? With the user-space switch, packets must cross from user-space to kernel-space and back on every hop, rather than staying in the kernel as they go through the switch. The user-space switch is easier to modify (no kernel oops'es to deal with), but slower for simulation.

Exit Mininet:

```
mininet> exit
```

Create Learning Switch

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.

For this tutorial you will need to choose a controller platform to build on top of. The best option is likely the controller written in the language you are most familiar with. Plus, after making your controller, rewriting it on another controller platform should go quickly and be a useful exercise. Your options are:

- **Java**: Beacon, Floodlight
- **Python**: POX, Ryu, NOX (Deprecated)
- **Ruby**: Trema

Controller Choice: POX (Python)

NOTE: The POX version of the tutorial is quite new, and feedback on the pox-dev mailing list is very welcome!

One option for the first exercise is to use POX. POX is a Python-based SDN controller platform geared towards research and education. For more details on POX, see About POX or POX Documentation on NOXRepo.org.

We're not going to be using the reference controller anymore, which may still be running (do '`ps -A | grep controller`' if you're unsure), so you should either press Ctrl-C in the window running the controller program, or kill it from the other SSH window:

```
$ sudo killall controller
```

You should also run `sudo mn -c` and restart Mininet to make sure that everything is "clean" and using the faster kernel switch. From your Mininet console:

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

You then need to download the POX code from the POX repository on github into your VM:

```
$ git clone http://github.com/noxrepo/pox
$ cd pox
```

> **OPTIONAL**: The above should get you the latest release branch of POX. If you like the bleeding edge, you might like to switch to the latest development branch. This is documented on the POX website, but as of this writing, it's the "betta" branch:
> `$ git checkout betta`

Now you can try running a basic hub example:

```
$./pox.py log.level --DEBUG misc.of_tutorial
```

This tells POX to enable verbose logging and to start the of_tutorial component which you'll be using (which currently acts like a hub).

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, POX will print something like this:

```
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01
DEBUG:samples.of_tutorial:Controlling [Con 1/1]
```

The first line is from the portion of POX that handles OpenFlow connections. The second is from the tutorial component itself.

Verify Hub Behavior with tcpdu mp

Now we verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host and view the traffic in each. 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. This is how a hub works; it sends all packets to every port on the network.

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

```
# ping -c1 10.0.0.5
```

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 xterms now.

Benchmark Hub Controller w/iperf

Here, you'll benchmark the provided of_tutorial hub.

First, verify reachability. Mininet should be running, along with the POX hub 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
```

Now, compare your number with the reference controller you saw before. How does that compare?

Hint: every packet goes up to the controller now.

Open Hub Code and Begin

Go to your SSH terminal and stop the tutorial hub controller using Ctrl-C. The file you'll modify is **pox/misc/of_tutorial.py**. Open this file in your favorite editor.

The current code calls act_like_hub() from the handler for packet_in messages to implement switch behavior. You'll want to switch to using the act_like_switch() function, which contains a sketch of what your final learning switch code should look like.

Each time you change and save this file, make sure to restart POX, 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. For (2) and (3), hosts that are not the destination for a ping should display no tcpdump traffic after the initial broadcast ARP request.

Learning Python

This section introduces Python, giving you just enough to be dangerous.

Python:

- is a dynamic, interpreted language. There is no separate compilation step - just update your code and re-run it.
- uses indentation rather than curly braces and semicolons to delimit code. Four spaces denote the body of a for loop, for example.
- is dynamically typed. There is no need to pre-declare variables and types are automatically managed.
- has built-in hash tables, called dictionaries, and vectors, called lists.
- is object-oriented and introspective. You can easily print the member variables and functions of an object at runtime.
- runs slower than native code because it is interpreted. Performance-critical controllers may want to distribute processing to multiple nodes or switch to a more optimized language.

Common operations:

To initialize a dictionary:

```
mactable = {}
```

To add an element to a dictionary:

```
mactable[0x123] = 2
```

To check for dictionary membership:

```
if 0x123 in mactable:
    print 'element 2 is in mactable'
if 0x123 not in mactable:
    print 'element 2 is not in mactable'
```

To print a debug message in POX:

```
log.debug('saw new MAC!')
```

To print an error message in POX:

```
log.error('unexpected packet causing system meltdown!')
```

To print all member variables and functions of an object:

```
print dir(object)
```

To comment a line of code:

```
# Prepend comments with a #; no // or /**/
```

More Python resources:

- List of built-in functions
- Official Python tutorial

The subsections below give details about POX APIs that should prove useful in the exercise. There is also other documentation available in the appropriate section of POX's website.

Sending OpenFlow messages with POX

```
connection.send( ... ) # send an OpenFlow message to a switch
```

When a connection to a switch starts, a ConnectionUp event is fired. The example code creates a new Tutorial object that holds a reference to the associated Connection object. This can later be used to send commands (OpenFlow messages) to the switch.

ofp_action_output class

This is an action for use with ofp_packet_out and ofp_flow_mod. It specifies a switch port that you wish to send the packet out of. It can also take various "special" port numbers. An example of this would be OFPP_FLOOD which sends the packet out all ports except the one the packet originally arrived on.

Example. Create an output action that would send packets to all ports:

```
out_action = of.ofp_action_output(port = of.OFPP_FLOOD)
```

ofp_match class

Objects of this class describe packet header fields and an input port to match on. All fields are optional -- items that are not specified are "wildcards" and will match on anything.

Some notable fields of ofp_match objects are:

- dl_src - The data link layer (MAC) source address
- dl_dst - The data link layer (MAC) destination address
- in_port - The packet input switch port

Example. Create a match that matches packets arriving on port 3:

```
match = of.ofp_match()
match.in_port = 3
```

ofp_packet_out OpenFlow message

The ofp_packet_out message instructs a switch to send a packet. The packet might be one constructed at the controller, or it might be one that the switch received, buffered, and forwarded to the controller (and is now referenced by a buffer_id).

Notable fields are:

- buffer_id - The buffer_id of a buffer you wish to send. Do not set if you are sending a constructed packet.
- data - Raw bytes you wish the switch to send. Do not set if you are sending a buffered packet.
- actions - A list of actions to apply (for this tutorial, this is just a single ofp_action_output action).
- in_port - The port number this packet initially arrived on if you are sending by buffer_id, otherwise OFPP_NONE.

Example. of_tutorial's send_packet() method:

```
 def send_packet (self, buffer_id, raw_data, out_port, in_port):
   """
   Sends a packet out of the specified switch port.
   If buffer_id is a valid buffer on the switch, use that.  Otherwise,
   send the raw data in raw_data.
   The "in_port" is the port number that packet arrived on.  Use
   OFPP_NONE if you're generating this packet.
   """
   msg = of.ofp_packet_out()
   msg.in_port = in_port
   if buffer_id != -1 and buffer_id is not None:
     # We got a buffer ID from the switch; use that
     msg.buffer_id = buffer_id
   else:
     # No buffer ID from switch -- we got the raw data
     if raw_data is None:
       # No raw_data specified -- nothing to send!
       return
     msg.data = raw_data

   action = of.ofp_action_output(port = out_port)
   msg.actions.append(action)

   # Send message to switch
   self.connection.send(msg)
```

ofp_flow_mod OpenFlow message

This instructs a switch to install a flow table entry. Flow table entries match some fields of incoming packets, and executes some list of actions on matching packets. The actions are the same as for ofp_packet_out, mentioned above (and, again, for the tutorial all you need is the simple ofp_action_output action). The match is described by an ofp_match object.

Notable fields are:

- idle_timeout - Number of idle seconds before the flow entry is removed. Defaults to no idle timeout.
- hard_timeout - Number of seconds before the flow entry is removed. Defaults to no timeout.
- actions - A list of actions to perform on matching packets (e.g., ofp_action_output)
- priority - When using non-exact (wildcarded) matches, this specifies the priority for overlapping matches. Higher values are higher priority. Not important for exact or non-overlapping entries.
- buffer_id - The buffer_id of a buffer to apply the actions to immediately. Leave unspecified for none.
- in_port - If using a buffer_id, this is the associated input port.
- match - An ofp_match object. By default, this matches everything, so you should probably set some of its fields!

Example. Create a flow_mod that sends packets from port 3 out of port 4.

```
fm = of.ofp_flow_mod()
fm.match.in_port = 3
fm.actions.append(of.ofp_action_output(port = 4))
```

For more information about OpenFlow constants, see the main OpenFlow types/enums/structs file, **openflow.h**, in ~/openflow/include/openflow/openflow.h You may also wish to consult POX's OpenFlow library in pox/openflow/libopenflow_01.py and, of course, the OpenFlow 1.0 Specification.

Parsing Packets with the POX packet libraries

The POX packet library is used to parse packets and make each protocol field available to Python. This library can also be used to construct packets for sending.

The parsing libraries are in:

```
pox/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 pox.lib.addresses.EthAddr objects. These can easily be converted to their common string representation (`str(addr)` will return something like "01:ea:be:02:05:01"), or created from their common string representation (`EthAddr("01:ea:be:02:05:01")`).

To see all members of a parsed packet object:

```
print dir(packet)
```

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

```
['HW_TYPE_ETHERNET', 'MIN_LEN', 'PROTO_TYPE_IP', 'REPLY', 'REQUEST', 'REV_REPLY',
 'REV_REQUEST', '__class__', '__delattr__', '__dict__', '__doc__', '__format__',
 '__getattribute__', '__hash__', '__init__', '__len__', '__module__', '__new__',
 '__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_init', 'err',
 'find', 'hdr', 'hwdst', 'hwlen', 'hwsrc', 'hwtype', 'msg', 'next', 'opcode',
 'pack', 'parse', 'parsed', 'payload', 'pre_hdr', 'prev', 'protodst', 'protolen',
 'protosrc', 'prototype', 'raw', 'set_payload', 'unpack', 'warn']
```

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.

Now, skip ahead to Testing Your Controller below.

Controller Choice: Beacon (Java)

Beacon is a Java-based OpenFlow controller platform. For more details and tutorials see the Beacon Homepage.

Beacon is very easily developed using the Eclipse Integrated Development Environment which runs on any operating system (OS).

Therefore, for this tutorial you will you need to download Beacon and run Eclipse on your host machine, rather than inside the Mininet virtual machine.

The next section will help you do that.

Setting up a Beacon Development Envir onment

This section is a modified copy of the original Beacon quick-start guide.

Prerequisites

You will need the following installed in your **host machine** :

- Java 6 JDK & JRE (Java 7 JDK is fine on OSX)

Next, download the Beacon tutorial package for your **host machine's operating system**, the file names begin with beacon-tutorial-eclipse:

- Tutorial Packages

After the download is complete extract it to your desktop, or somewhere else that is convenient. For guide purposes, the folder you extracted it to will be referred to by <path to>/beacon-tutorial-1.0.2

Setup

Launch Eclipse by running the eclipse executable inside <path to>/beacon-tutorial-1.0.2/eclipse/

(Optional) Create a new Eclipse workspace if you are not starting from a fresh Eclipse install

- File -> Switch Workspace -> Other, pick a new folder to host the workspace

Set Eclipse's compliance level to 1.6

- Window (or Eclipse for MAC) -> Preferences -> Java -> Compiler then under JDK Compliance, change Compiler compliance level to 1.6.

Import Beacon and OpenFlowJ projects

- File -> Import -> General -> Existing Projects into Workspace, Select <path to>/beacon-tutorial-1.0.2/beacon-tutorial-1.0.2/src as the root directory, click ok, then select all the projects, ensure copy projects into workspace is not checked and click finish.

Note at this point you will see many Java errors, the libraries Beacon depends on will be installed in the next step which will resolve the errors, do not panic!

Set the target (download libraries)

- Open the Beacon Main Target project, double click the main-local.target file.
- Click Set as Target Platform in the top right corner of the main.target window. (Note if you click before it has been resolved, you will receive an error). Wait a few seconds, at this point all compilation errors should be gone.

Import the Beacon code style settings

- Click Window -> Preferences. Then in the left column click Java -> Code Style -> Formatter, then click the Import button, and select <path to>/beacon-tutorial-1.0.2/src/beacon-1.0.2/beacon_style_settings.xml and hit ok, then ensure the Active profile is Beacon.

That's it! Your Beacon controller is ready to rock'n'roll.

Running the Tutorial Controller

Back in your VM, We're not going to be using the reference controller anymore, which is running in the background (do 'ps -A | grep controller' if you're unsure).

The switches are all trying to connect to a controller, and will increase the period of their 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.

Make sure the reference controller used before is not running:

```
$ sudo killall controller
```

You will need to tell the Mininet virtual switches to connect to your host machine's IP address. To get that, from your Mininet ssh window run the command sudo route:

```
openflow@openflowtutorial:~$ sudo route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.206.0   *               255.255.255.0   U     0      0        0 eth0
default         192.168.206.2   0.0.0.0         UG    0      0        0 eth0
```

Look for the line that starts with default, and typically your host's IP address will be the IP in the Gateway column. If that IP address ends with .1, and later you are unable to connect Mininet to Beacon, try the same IP but swapping the ending .1 with .2.

You should also restart Mininet to make sure that everything is "clean". From your Mininet console:

```
mininet> exit
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote,ip=<your_host_ip>
```

Make sure you do not add extra spaces around the comma.

Note: The above syntax is for Mininet 2.0 - for Mininet 1.0, use the following syntax instead:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote --ip <your_host_ip>
```

The ip parameter defines in which IP your controller is located. Note that this is the IP your host uses to communicate with the VM, as discussed above. In my case, the VM's IP is 192.168.206.4 and the host's IP is 192.168.206.2 . Thus, I started mininet using

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote,ip=192.168.206.2
```

Now it's time to start Beacon.

Launching Beacon from Eclipse in debug mode

- Run -> Debug Configurations
- Look for the OSGi Framework on the left, expand its children and select 'beacon Tutorial LearningSwitch', then click Debug.

This command will start Beacon, including the 'tutorial' bundle, listening to incoming connection from switches on the standard OpenFlow port (6633).

Wait until the application indicates that the OpenFlow switch has connected. When the switch connects, your Eclipse console should print something like this:

```
01:07:31.095 [pool-2-thread-2] INFO n.b.core.internal.Controller - Switch connected from
java.nio.channels.SocketChannel[connected local=/192.168.206.2:6633 remote=/192.168.206.4:54994]
```

Verify Hub Behavior with tcpdu mp

Now we verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host, and view the traffic in each. 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 10.0.0.5
```

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 xterms now.

Benchmark Hub Contr oller w/iperf

Here, you'll benchmark the provided hub code, part of the Tutorial bundle.

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

```
mininet> pingall
```

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

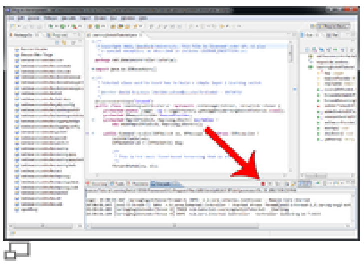
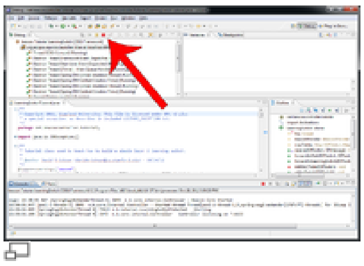
```
mininet> iperf
```

Now, compare your number with the reference controller you saw before. How does that compare?

Open Tutorial Code and Begin

Go to Eclipse and stop Beacon (you should see a square, red button in the console window, or the top left section of Eclipse if you are in the Debug perspective).

NOTE : You can run only one controller at the same time (otherwise you'll get a port conflict). Make sure that you stop any running instance of Beacon before you start another one. With Eclipse this might be tricky. To check the running programs open the Debug perspective (Window->Open Perspective->Debug). On the top-left corner you can see the running programs, click on it then hit the red square to terminate the redundant ones...

The file you'll modify is **net.beaconcontroller.tutorial/src/main/java/net/beaconcontroller/tutorial/LearningSwitchTutorial.java**. Find this file in the left pane of Eclipse and double-click it to open it in the editor window.

Take a quick look through the file, the key things to notice:

- The receive method is where packets initially arrive from switches, by default this method calls forwardAsHub. Once you have implemented the code in forwardAsLearningSwitch you should change receive by commenting out the call to forwardAsHub, and uncommenting the call to forwardAsLearningSwitch.
- The forwardAsHub method behaves as a broadcast hub, and has code that gives you an example of how to send an OFPacketOut message.
- The forwardAsLearningSwitch method is where you will be writing all of your code (with the exception of the receive method mentioned earlier). It should take you around 30 lines of code in this method to implement a learning switch.

Wireshark will be very useful to you in getting your code working. Try starting Beacon and doing some pings with Mininet while capturing with Wireshark. You should see OFPacketIn and OFPacketOut messages, and if you explore the OFPacketOut messages you should see their action is to flood.

Each time you change and save the LearningSwitchTutorial file, make sure to stop and start your running instance, then use pings to verify hub or Ethernet learning switch behavior. Hosts that are not the destination should display no tcpdump traffic after the initial broadcast ARP request.

There is a significant number of comments in the LearningSwitchTutorial file to help you create the proper functionality, and tips provided in the subsequent sections of this tutorial (**read these!**). At a high level there are two phases of work for you to do.

Phase 1

- Build an OFMatch object based on the incoming OFPacketIn
- Learn that the source MAC address is at the incoming port and store it in the Map

- Retrieve the port the destination MAC address is at, if it is known
- If the destination port is known, send an OFPacketOut directly and only to that port, else forward using the flooding method in the forwardAsHub method

At the end of this phase you should be able to send pings back and forth, and in Wireshark you should see an initial OFPacketOut in both directions with its action set to flood, subsequent OFPacketOut messages should have their action set to send to a single port.

Phase 2

- If the destination port is known, instead of sending an OFPacketOut, send an OFFlowMod so that subsequent pings are matched on the switch.

At the end of phase 2, when pinging and viewing the controller traffic in Wireshark you should see an initial OFPacketOut that floods for the request/response, then an OFFlodMod being sent for both request/response, then no further traffic for the same ping.

Learning Java and Beacon

Here are a few Java tips you might find useful down the road.

To add an element to a HashMap:

```
macTable.put(mac_address_key, port)
```

To convert a MAC address in a byte[] to a Long:

```
Ethernet.toLong(match.getDataLayerSource()/match.getDataLayerDestination())
```

To check for an element in your HashMap:

```
if (macTable.containsKey(mac_address_key))
   log.debug("mac_address_key is in hashmap")
```

To get an element from the HashMap:

```
learned_port = macTable.get(mac_address_key)
```

To print a debug message in Beacon:

```
log.debug("Learned new MAC!")
```

You can change the logging level using info or error instead of debug.

The subsections below give details about a few Beacon Classes that should prove useful in the exercise.

Useful Beacon Classes

In your beacon-tutorial-1.0.2 directory, there is an apidocs folder which contains Javadoc documentation for all essential classes in this assignment. Open your browser and point to <your beacon-tutorial-1.0.2 directory>/apidocs/index.html and look for the classes mentioned below, which should help while implementing your code.

To send a message to an OpenFlow switch, look at IOFSwitch class. The starter forward_as_hub function sends packets out using:

```
sw.getOutputStream().write(po);
```

Look at OFMatch and OFPacketIn to get header information from an OpenFlow packet-in. You should find the following command particularly useful:

```
match.loadFromPacket(pi.getPacketData(), pi.getInPort());
```

Down the road, you may want to print a mac address, or use it as a key to a HashMap. OFMatch will give you the address as a byte array. Use the functions/classes below in case needed:

```
HexString.toHexString(byte[] bytes) // Convert a string of bytes to a ':' separated hex string
Long mac_address_key = Ethernet.toLong(byte array); // Creates a Long key from a byte array.
```

Finally, at some point you will have to install a flow in the network. Use OFFlowMod and OFActionOutput to do that. Here is a way to initialize a flow-mod message for a specific switch:

```
OFFlowMod fm = new OFFlowMod();
```

Use the appropriate setters to construct the flow-mod you need. Assuming you know the outgoing port for this flow, this is a snippet on how to include this in your flow-mod message:

```
OFActionOutput action = new OFActionOutput(outPort);
fm.setActions(Collections.singletonList((OFAction)action));
```

Controller Choice: Floodlight (Java)

Floodlight is a Java-based OpenFlow controller platform. For more details and tutorials see the FloodLight OpenFlowHub page.

Install Prerequisites

Note: these instructions are a customized version of those on the Floodlight Download page

Make sure that you have Internet access:

```
ping -c1 www.stanford.edu
```

If not, ensure that each ethX interface on your VM has an IP assigned via DHCP:

```
ifconfig -a
```

Run for each interface without an IP assigned, replacing ethX as necessary:

```
sudo dhclient ethX
```

Update apt sources:

```
sudo apt-get update
```

Install prereqs:

```
time sudo apt-get install build-essential default-jdk ant python-dev
```

Getting Started

While the dependencies are installing on your VM, follow the Floodlight Getting Started instructions.

Developing

Once through with the getting started part, work through Developing on Floodlight.

Controller Choice: Trema (Ruby)

Trema is a Ruby-based OpenFlow controller platform and testing environment. For more details and tutorials see the Trema GitHub page.

Install Prerequisites

Make sure that you have Internet access:

```
ping -c1 www.stanford.edu
```

If not, ensure that each ethX interface on your VM has an IP assigned via DHCP:

```
ifconfig -a
```

Run for each interface without an IP assigned, replacing ethX as necessary:

```
sudo dhclient ethX
```

Update apt sources:

```
sudo apt-get update
```

Install prereqs:

```
sudo apt-get install rubygems1.8
```

Getting Started

Follow the instructions on the main Trema page, which has links to tutorial videos and more.

Controller Choice: Ryu (Python)

Ryu is a python-based OpenFlow controller. For details, please visit the Ryu Official Site

Ryu VM Image

The VM image set up for Ryu is available at

- Virtual Machine Image(zipped OVF format, 64-bit, Mininet 2.0).

This can be import into virtualbox, VMWare or other popular virtualization programs.

Important: For this VM image, the user name is 'ryu' with password 'ryu'.

Supplement on VM Setup

If you have troubles to connect into guest with ssh, please try

- Add host-only network (If your virtual box is not so-new, the host-only network is already created by default. In that case, skip to the next step) file menu/Preferences/Network and "Add host-only network" button with default settings.

- select your VM and go to the Setting Tab. Go to Network->Adapter 2. Select the "Enable Adapter" box and attach it to "host-only network" which was created at the step 1.

Install Prerequisites

You will need the following installed in your environment that runs Ryu

- python-gevent>=0.13
- python-routes
- python-webob
- python-paramiko

Make sure that you have internet access:

```
  ping -c www.stanford.edu
```

If not, ensure that each ethX interface on your VM has an IP assigned via DHCP:

```
  ifconfig -a
```

Run for each interface without an IP assigned, replacing ethX as necessary:

```
  sudo dhclient ethX
```

update apt soruces:

```
  sudo apt-get update
```

install prereqs:

```
  time sudo apt-get install python-gevent python-routes python-webob python-paramiko
```

Getting Started and developing

Please visit the OpenFlow Tutorial page.

Controller Choice: NOX w/Python

NOTE: The version of NOX used in this tutorial is no longer current. In fact, all Python-based development in NOX is now deprecated.

DEVELOPERS WISHING TO USE PYTHON ARE ENCOURAGED TO USE POX INSTEAD.

One option for the first exercise is to use NOX, a controller platform that allows you to write a controller on top, in Python, C++, or some combination of the two. From noxrepo.org:

> 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.

For more details, see the NOX overview.

If NOX classic is not already installed in your Mininet 2.0 VM, you can install it by typing

```
 cd ~
 mininet/util/install.sh -x
```

This may take some time (20 minutes), so it is recommended that you do this in advance if you are planning to use NOX classic. The older Mininet 1.0 VM includes NOX classic.

We're not going to be using the reference controller anymore, which is running in the background (do 'ps -A | grep controller' if you're unsure).

Make sure the reference controller used before is not running, so that NOX can use port 6633. Press Ctrl-C in the window running the controller program, or in the other SSH window run:

```
$ sudo killall controller
```

You should also run `sudo mn -c` and restart Mininet to make sure that everything is "clean" and using the faster kernel switch. From your Mininet console:

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

NOTE: The following usage instructions conforms perfectly with NOX Classic (bundled with Mininet 1.0 VM). However, there may be some minor variations with other versions.

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

```
$ cd ~/nox/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.

Verify Hub Behavior with tcpdu mp

Now we verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host and view the traffic in each. 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. This is how a hub works; it sends all packets to every port on the network.

Now, see what happens when a non-existent host doesn't reply. From h2 xterm:

```
# ping -c1 10.0.0.5
```

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 xterms now.

Benchmark Hub Contr oller w/iperf

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
```

Now, compare your number with the reference controller you saw before. How does that compare?

Hint: every packet goes up to the controller now.

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 **~/nox/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. For (2) and (3), hosts that are not the destination for a ping should display no tcpdump traffic after the initial broadcast ARP request.

Learning Python

This section introduces Python, giving you just enough to be dangerous.

Python:

- is a dynamic, interpreted language. There is no separate compilation step - just update your code and re-run it.
- uses indentation rather than curly braces and semicolons to delimit code. Four spaces denote the body of a for loop, for example.
- is dynamically typed. There is no need to pre-declare variables and types are automatically managed.
- has built-in hash tables, called dictionaries, and vectors, called lists.
- is object-oriented and introspective. You can easily print the member variables and functions of an object at runtime.
- runs slower than native code because it is interpreted. Performance-critical controllers may want to distribute processing to multiple nodes or switch to a more optimized language.

Common operations:

To initialize a dictionary:

```
mactable = {}
```

To add an element to a dictionary:

```
mactable[0x123] = 2
```

To check for dictionary membership:

```
if 0x123 in mactable:
    print 'element 2 is in mactable'
```

To print a debug message in NOX:

```
log.debug('saw new MAC!')
```

To print an error message in NOX:

```
log.error('unexpected packet causing system meltdown!')
```

To print all member variables and functions of an object:

```
print dir(object)
```

To comment a line of code:

```
# Prepend comments with a #; no // or /**/
```

More Python resources:

- List of built-in functions
- Official Python tutorial

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

Sending OpenFlow messages with NOX

NOX API documentation can be found here. Please look there for the appropriate calls. We also 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 /home/openflow/nox/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 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 below)

        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:

```
~/nox/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 ~/nox/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.

Now, skip ahead to Testing Your Controller below.

Testing Your Controller

To test your controller-based Ethernet switch, first verify that when all packets arrive at the controller, only broadcast packets (like ARPs) and packets with unknown destination locations (like the first packet sent for a flow) go out all non-input ports. You can do this with tcpdump running on an xterm for each host.

Once the switch no longer has hub behavior, work to push down a flow when the source and destination ports are known. You can use dpctl to verify the flow counters, and if subsequent pings complete much faster, you'll know that they're not passing through the controller. You can also verify this behavior by running iperf in Mininet and checking that no OpenFlow packet-in messages are getting sent. The reported iperf bandwidth should be much higher as well, and should match the number you got when using the reference learning switch controller earlier.

Let the instructor know when your learning switch works!

Support Multiple Switches

If you are working with POX, your code should already be able to pass the following tests because it creates one instance per switch and each instance maintains its own MAC table. Otherwise, your controller so far may have probably only supported 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
```

If you are using Beacon and Mininet 2.0, use the ip= option:

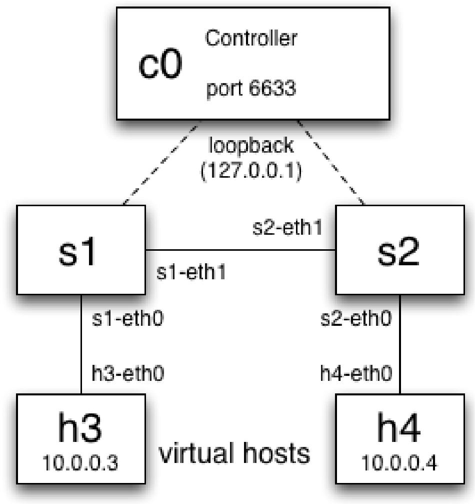
```
mininet> exit
$ sudo mn --topo linear --switch ovsk --controller remote,ip=<your_host_ip>
```

If you are using Beacon and Mininet 1.0, use the --ip option:

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

Your created topology looks like this:

Note: for Mininet 2.0, the hosts are h1/10.1 and h2/10.2



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. 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
```

Congratulations on getting this far. If you are doing this as part of the CS244 assignment, you are done!

If you want to continue, you have three options:

- run your code over real networking gear (if you are on a live tutorial)
- add layer-3 forwarding capabilities to your switch
- add firewall capabilities to your switch

Proceed to the section you prefer.

Control a Slice of a real Network

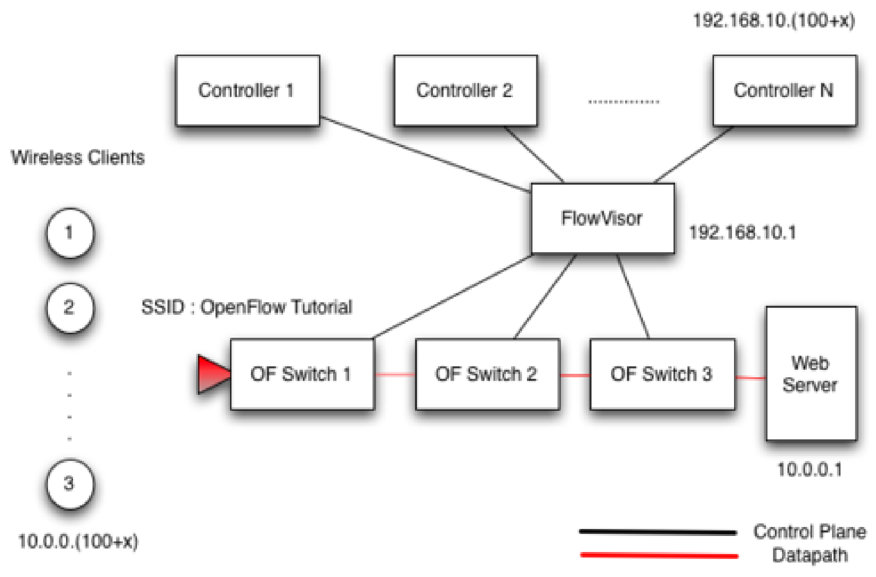
NOTE: This part is still in beta, so be prepared that things could break down the road...

In this exercise you'll use the code you have so far to control a slice of a real network consisting of 3 OpenFlow switches, a server and a wireless client. Your goal is to access a website and post a message. The interesting part is that your controller should allow traffic to go from client to server and vice versa. The setup includes:

1. 3 OpenFlow-enabled LinkSys Access Points.
2. FlowVisor (to virtualize the network and create slices)
3. 1 controller/team
4. 1 Webserver
5. 1 Wireless device/team (could be your laptop's WiFi, your iPhone, Droid etc)

Ideally, you shouldn't write any code for this experiment. You will have to form teams though. When you are ready to start, ask the instructors to put you in a team.

Here is the topology that we will use:



Setting up your Network

Here you'll work with a real network, so we won't use mininet. Go to your mininet console and type

```
mininet> exit
```

You will have to deal with two network configurations.

1. The control network which will allow your controller to talk with the FlowVisor and OpenFlow switches (192.168.10. subnet).
2. The actual network we want to control, i.e. your wireless client to access the Web Server (10.0.0. subnet).

Control Network Configuration

- Ask the instructor for an ethernet cable. This will go to a standard (non-OpenFlow) switch and connects to the FlowVisor.
- At your local machine, configure the interface where you plugged your ethernet cable with the following characteristics:

```
IP address : 192.168.10.X (where X = 100+TeamIndex, e.g. X=101 for Team1)
Netmask : 255.255.255.0
```

For Linux you'll have to do something like :

```
$ sudo ifconfig eth0 192.168.10.X netmask 255.255.255.0
```

For MAC

```
$ sudo ifconfig en0 192.168.10.X netmask 255.255.255.0
```

For Windows

```
Go to your Control Panel->Network Settings and configure your LAN device.
```

Now your local machine can access FlowVisor. To verify, ping FlowVisor from a terminal on your local machine:

```
ping 192.168.10.1
```

But what we really need is the guest VM - where the controller is - to be able to communicate with the FlowVisor. We will use port-forwarding to do that.

First, note the IP address of your VM guest. At your VM's shell, type:

```
$ ifconfig eth1
```

Now, we need to configure port-forwarding.

Linux and Mac users

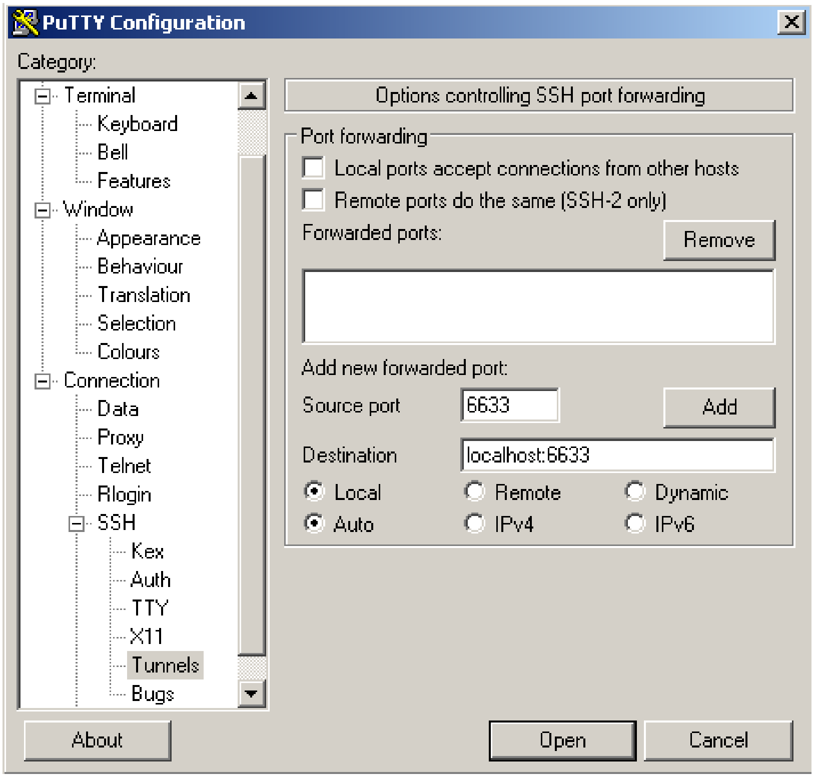
From your local machine:

```
ssh openflow@<guestIP> -L192.168.10.X:6633:<guestIP>:6633
```

where <guestIP> is the IP address of the eth1 interface of the VM, and X is 100 + your team number.

Windows users

To enable port forwarding on the Putty, please enter the appropriate fields during connection setup



Local ports accept connections from other hosts [ ]
Remote ports do the same (SSH-2 only) [ ]
Forwarded ports:
Add new forwarded port:
Source port: 6633
Destination: localhost:6633
[✓ Local] Remote Dynamic
[✓ Auto] IPv4 IPv6

Verify reachability

To verify that everything worked, use tcpdump at your guest VM and see whether you receive any incoming OpenFlow-related packets

```
$ sudo tcpdump -i lo port 6633
```

Note: The reason we specify the interface 'lo' not 'eth1' here is that SSH port forwarding arrives to the interface eth1 tcp port 22 and forwarded to the local interface 'lo' tcp port 6633.

Due to reconnection backoff, it might take up to one minute before you see any packets coming in. If you don't get any packets after 1 minute, ask one of the instructors for help.

Wireless Client Configuration

You will have to configure your wireless client's network interface. First login to the wireless network with SSID "OpenFlow-Tutorial". Depending on your OS use ControlPanel/ifconfig or any other similar utility :

For Linux:

```
$ sudo ifconfig wlan0 10.0.0.X netmask 255.255.255.0
```

For Mac:

```
$ sudo ifconfig en1 10.0.0.X netmask 255.255.255.0
```

(Or configure it using Network Preferences.)

For Windows, go to Control Panel->Network Settings and configure it manually.

Write down your wireless card MAC address. We'll use this shortly.

Try to ping the WebServer at 10.0.0.1. Does it work? If not, why?

FlowVisor Configuration

Several teams will be using the same switches at the same time, so we have to virtualize our network and give each team a separate slice. Each controller will have a limited view of the network, as this is dictated by its slice configuration. In this exercise, each team will control all the traffic that comes from and goes to its wireless client. To do that we will need the MAC address of your team's wireless client. Hopefully you already have this from the previous step. Give this MAC address to the instructor to update your slice configuration at the FlowVisor.

Here is how a slice definition looks at the flowvisor configuration :

```
Name: team1
ID: 1
Host: tcp:192.168.10.101:6633

FlowSpace: allow: dl_src: 00:22:41:66:13:1c
FlowSpace: allow: dl_dst: 00:22:41:66:13:1c
```

Note that 192.168.10.101:6633 points to this slice's OF controller, 00:22:41:66:13:1c is the MAC address of the wireless client, and every packet with that ethernet_src or ethernet_dst is assigned to this controller.

For more details on FlowVisor, look at the wikipage.

Run your controller

Go to your directory and start your controller.

```
$ cd ~/noxcore/build/src/
$ ./nox_core -i ptcp: -v pytutorial
```

What do you see? Are there any switches connecting to your controller? Use your debug messages or wireshark for that.

Accessing the WebServer

By now, you should have a functional network between your client and the webserver.

Try again to ping the server:

```
$ ping 10.0.0.1
```

Does it work now? How packets travel from client to the server? Check the switch flow-tables using dpctl.

```
$ dpctl dump-flows tcp:192.168.10.X (X=11,12,13)
```

What do you see? Do you recognize all flows?

If ping works, you are ready to go to the final step. Fire your browser and go to http://10.0.0.1:8000 Write and post a short message and make sure you fill your team's number.

Look what others said at http://10.0.0.1:8000/posts

Create Router

In this exercise, you'll make a static layer-3 forwarder/switch. It's not exactly a router, because it won't decrement the IP TTL and recompute the checksum at each hop (so traceroute won't work). Actions to do TTL and checksum modification are expected in the upcoming OpenFlow version 1.1. However, it will match on masked IP prefix ranges, just like a real router.

From RFC 1812:

> An IP router can be distinguished from other sorts of packet switching devices in that a router examines the IP protocol header as part of the switching process. It generally removes the Link Layer header a message was received with, modifies the IP header, and replaces the Link Layer header for retransmission.

To simplify the exercise, your "router" will be completely static. With no BGP or OSPF, you'll have no need to send or receive route table updates.

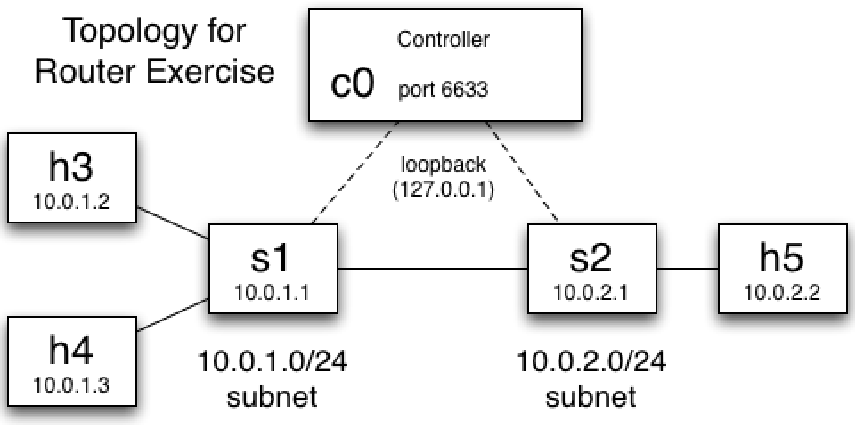
Each network node will have a configured subnet. If a packet is destined for a host within that subnet, the node acts as a switch and forwards the packet with no changes, to a known port or broadcast, just like in the previous exercise. If a packet is destined for some IP address for which the router knows the next hop, it should modify the layer-2 destination and forward the packet to the correct port.

Our hope is that this exercise will show that with an OpenFlow-enabled forwarding device, the network is effectively layerless; you can mix switch, router, and higher-layer functionality.

Please note that this is an advanced exercise, and given that most implementation details are up to you, will be harder.

Create Topology

You'll need a slightly different topology, something like this: **Note: For Mininet 2.0, the hosts have been renumbered h1-h3 and 10.1-10.3.**



… which will need to be described in a way that Mininet will understand.

There's an example custom topology at:

```
~/mininet/custom/topo-2sw-2host.py
```

First, copy the example to a new file:

```
$ cp ~/mininet/custom/topo-2sw-2host.py mytopo.py
```

To run a custom topology, pass Mininet the custom file and pass in the custom topology:

```
$ sudo mn --custom mytopo.py --topo mytopo --mac
```

Then, in the Mininet console, run:

```
mininet> pingall
```

Now, modify your topology file to match the picture and verify full host connectivity with pingall in the Mininet console.

Set up hosts

Set up IP configuration on each virtual host to force each one to send to the gateway for destination IPs that are outside of their configured subnet.

You'll need to configure each **host** with a subnet, IP, gateway, and netmask.

It may seem obvious, but we will warn you anyway: do not attempt to assign IP addresses to the interfaces belonging to switches s1 and s2. If you need to handle traffic "to" or "from" a switch, do so using OpenFlow.

There are several ways that this can be done:

- **Unix commands via Mininet CLI**

From within the Mininet CLI, you can send commands to hosts to configure them:

```
mininet> h1 ifconfig h2-eth0 10.0.1.1/24
mininet> h1 route add default gw 10.0.1.1
mininet> h1 route -n
```

- **Regular Unix commands via an xterm**

Another way of configuring hosts is to open up an xterm

```
mininet> xterm h1
```

and then to type into that xterm window to configure the host:

```
# ifconfig h1-eth0 10.0.1.1/24
```

- **Running a configuration script**

You can also run scripts on the hosts themselves, for example, if you have a script called config_script, you can run it on host h2 from the Mininet CLI:

```
mininet> h1 config_script
```

If you write such a script, you can either have it automatically know what to do (e.g. based on MAC address, which you can assign based on node number with --mac) or pass in additional parameters as desired.

- **CLI scripting**

To avoid repetitive CLI typing, it is possible to create a file consisting of multiple CLI commands, and then run that script in Mininet:

```
mininet> source my_cli_script
```

or

```
# mn --pre my_cli_script
```

A sample CLI script might look like:

```
py "Configuring network"
h1 ifconfig h3-eth0 10.0.1.1/24
h2 ifconfig h4-eth0 10.0.1.2/24
h3 ifconfig h5-eth0 10.0.2.3/24
h1 route add default gw 10.0.1.1
```

```
h2 route add default gw 10.0.1.1
h3 route add default gw 10.0.2.1
py "Current network:"
net
dump
```

CLI scripting is designed to save typing rather than to be a full programming environment (although you can evaluate Python expressions from the CLI, as shown above using the 'py' command!) For advanced tasks, the Mininet API provides flexible access to all Mininet capabilities.

- **Mininet API**

Mininet's Python API provides a much more powerful means of creating and configuring networks programatically, but is probably beyond the scope of this tutorial. Some examples of using the Mininet API may be found in `mininet/examples`, and they are described in `mininet/examples/README`.

- **Changing the node IDs in the topology**

It is also possible to change the default IP addresses by changing node IDs in a topology.

The node ID in a topology is the default IP address in base 256. For example, setting node ID 258 will result in a default IP address of 10.0.1.2.

The default IP prefix length may be changed with the command-line option --prefixlen. For example,

```
# mn --prefixlen 24
```

will use /24 as the default prefix length (i.e. netmask of 255.255.255.0).

For the this tutorial, you may find that using CLI scripting or changing the node IDs (and setting the prefix length to 24) may be most convenient.

Once you have your new topology configured and connected to your pytutorial controller, you should be able to ping between hosts on the same subnet, but not between hosts on different subnets. Your job is to create a new (or revised) controller that installs rules to forward packets between the two subnets!

ARP

A router generally has to respond to ARP requests. You will see ethernet broadcasts which will (initially at least) be forwarded to the controller.

Your controller should probably construct ARP replies and forward them out the appropriate ports.

It may be possible to avoid dealing with ARP (at least initially) by adding static ARP entries in each host for all of the nodes on its subnet.

Static Routing

Once ARP has been handled, you will need to handle routing for the static configuration. Since we know what is connected to each port, we can match on IP address (or prefix, in the case of the remote subnet) and forward out the appropriate port.

In the case of the left switch, for example, we will want to forward packets destined for an attached host to the appropriate port, while packets destined for the remote subnet should be forwarded to the right switch.

ICMP

Additionally, your controller may receive ICMP echo (ping) requests for each switch, which it should respond to.

Lastly, packets for unreachable subnets should be responded to with ICMP network unreachable messages.

Testing your router

If the router works properly:

- attempts to send from 10.0.1.2 to an unknown address range like 10.99.0.1 should yield an ICMP destination unreachable message.
- packets sent to hosts on the same subnet should be treated like before.
- packets sent to hosts on a known address range should have their MAC dst field changed to that of the next-hop router.
- the router should be pingable, and should generate an ICMP echo reply in response to an ICMP echo request.

The exercise is meant to give more practice with packet parsing and show how to use OpenFlow to modify packets.

Let the instructor know when you get stuck or when you complete the exercise. This one is more open, and you'll need different actions (MAC dst modify) and matching (IP dst/prefix combinations), for which NOX core.py should help.

Create Firewall

In this exercise, you'll modify your switch to reject connection attempts to specific ports, just like a firewall.

This exercise is meant to show how OpenFlow can even do basic layer-4 tasks. Other uses could even extend into layer 7; cookie-based load balancing, for example.

Testing your Firewall

To run iperf on xterms, on the xterm for h2, start up a server:

```
$ iperf -s
```

Then on the xterm for h3 start up a client:

```
$ iperf -c 10.0.0.2
```

Your task is to prevent this from happening, and to block all flow entries with this particular port.

You can change the port used by iperf with the -p option. Both server and client will need this option specified. Again, NOX core.py will be useful for figuring out how to specify a flow entry that matches on wildcard ports. Also note that an empty action list will drop a packet; there's no explicit drop action.

Learn More

OpenFlow

To learn more about OpenFlow in general, consult the main OpenFlow page. There are videos, blog entries, and more. Check the wiki for link to OpenFlow-based projects and demos.

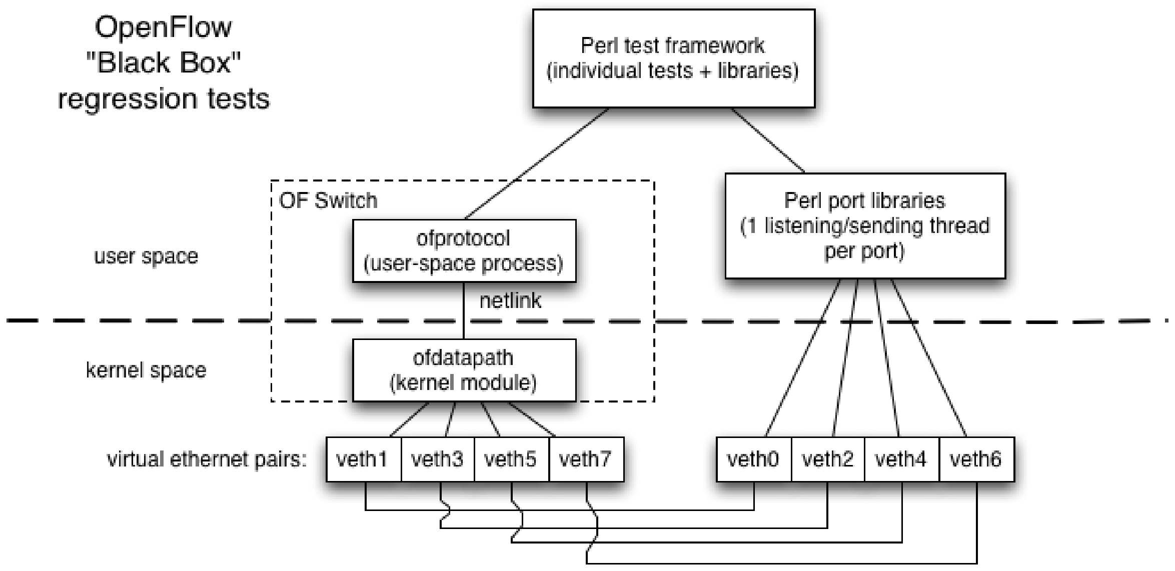
Additional Tools

- Mininet.
- FlowVisor.

Old Regression Tests

Note: you will need to install additional stuff to get the old regression tests working. See the install instructions for your platform, referenced at the top of the OpenFlow wiki

The OpenFlow reference distribution includes a set of tests to verify that an OpenFlow switch correctly sends and receives packets. In this section, you'll run through some of the "Black Box" tests, where the regression switch acts as a controller and verifies that a switch responds properly to OpenFlow messages, as well as properly forwards packets sent to the input ports of the switch. The Black Box regression tests use this layout:


OpenFlow "Black Box" regression tests
Perl test framework (individual tests + libraries)
OF Switch
user space
ofprotocol (user-space process)
netlink
Perl port libraries (1 listening/sending thread per port)
kernel space
ofdatapath (kernel module)
virtual ethernet pairs:
veth1
veth3
veth5
veth7
veth0
veth2
veth4
veth6

First, exit from the Mininet console:

```
mininet> exit
```

To run the tests, log in as root, from the same SSH terminal:

```
$ sudo su
```

Run a script to start up virtual ethernet pairs locally.

```
# openflow/regress/bin/veth_setup.pl
```

Veth0 connects to veth1, for example, and anytime a packet is sent to veth0, it'll pop out veth1, and vice versa. The regression suite will allocate four veth pair halves to a software OpenFlow switch and the other four to the test suite, which sends packets and checks the contents and locations of responses.

In an SSH terminal:

```
# ifconfig -a
```

You should see 8 new virtual interfaces, veth0..veth7.

Configure Wireshark to show OpenFlow messages and TCP resets in red (which delimit individual tests). In the Wireshark filter box, enter:

```
of || tcp.flags.reset == 1
```

Start running the tests, with Wireshark running on lo (loopback interface):

```
# openflow/regress/bin/of_kmod_veth_test.pl
```

You may see OpenFlow messages for each test - if not, hit Ctrl-C to stop the tests, and clean up any leftover state:

```
# openflow/regress/bin/of_kmod_veth_teardown.pl
```

Sometimes on the first run, the tests fail; we're not sure why this is happening. Restart after tearing down and they should work.

Then, re-run the tests:

```
# openflow/regress/bin/of_kmod_veth_test.pl
```

When you get bored, stop the tests and check the messages. Then clean up any state:

```
# openflow/regress/bin/of_kmod_veth_teardown.pl
```

The regression suite is useful for verifying new switch features, as well as diagnosis divergent behavior between different switches. The suite is generally used with 4 physical ports on hardware switches. We won't do anything more with it in this tutorial, but feel free to peruse the code. See the openflow/regress/projects/black_box directory, which has a file for each test.

To teardown virtual ethernet pairs, run:

```
# openflow/regress/bin/veth_teardown.pl
```

A replacement test suite is in the making, and is expected for mid 2010. The new suite will be based on Python and reorganized to make switch-specific tests and port configurations much easier to define and debug.

Exit back to regular username:

```
# exit
```

Credits

This tutorial was written by Brandon Heller and Yiannis Yiakoumis and beta-tested by Bob Lantz, KK Yap and Masayoshi Kobayashi.

Notes

VM Creation Notes

For the current Tutorial VM setup, see:

https://github.com/mininet/mininet/wiki/VM-Creation-Notes

Frequently Asked Questions (FAQ)

How can you convert a V irtualBox VM/.vdi image to a VM or .v mdk disk image usable by VMware?

If you have VMware installed, you may find that it is advantageous to run the VM under VMware rather than VirtualBox.

For VMware Player on Windows, use the "Export Appliance..." menu command from inside VirtualBox. Make sure you select the "Write Manifest file" option. This will create a .ova virtual machine archive which can be imported into VMware.

To import the .ova into VMware Player on Windows, open it using the "Open..." menu command.

For VMware Fusion on the Mac, it's slightly more complicated, but not terribly so. Use the "Export Appliance..." menu command from inside VirtualBox, making sure to save an .ovf (and .vmdk) file by pressing "Choose..." and selecting "Open Virtualization Format (*.ovf)" from the "Files of type:" pop-up menu. This will create a .ovf file, which you can ignore (or import using VMware's ovf tool if you have it installed), and a .vmdk disk image, which is exactly what you need.

(If you create an .ova file by mistake, you can either try again or rename the .ova file to a .tar file and extract it to get the .vmdk image.)

Next, create a new VM in VMware Fusion, delete its default hard disk (if any), and add a new hard disk, specifying the .vmdk file as the existing disk image to use. Once the VM has a single hard drive specified as the .vmdk file, it should be able to boot and run.

For VMware player on Ubuntu, you should be able to use the above approach that works for Windows, but I haven't tested it yet.

Alternately, you can create a .vmdk image from the .vdi file by using `qemu-img` from the `qemu` package; for example:

```
qemu-img convert -f vdi tutorial.vdi -O vmdk tutorial.vmdk
```

You can then attach it to a new VM as described in the VMware Fusion instructions.

How does one scroll an xterm?

Start it with the flag -sb 500 to store 500 lines of output, e.g.

```
xterm -sb 500 &
```

then use a scroll wheel, trackpad scrolling, or the middle mouse button to scroll with the quaint Athena scroll widget (i.e. grey bar on the left side of the window.)

If you want a more advanced terminal, you might wish to try installing gnome-terminal, e.g.

```
sudo apt-get update
sudo apt-get install gnome-terminal
gnome-terminal &
```

Why can't I ssh into my VirtualBox VM?

You may not be able to connect to a VirtualBox VM that only has NAT networking enabled. Make sure that you have followed the instructions above and **configured host-only networking on at least one interface in the VM settings**, and make sure that the **host-only interface** is configured and that you are connecting to its correct IP address.

If this does not work, or for more advanced setup, see below.

I can't start WireShark or xterm - help!

You are probably getting an error like "cannot open display."

This probably means that you have not successfully connected to the VM with ssh and X11 forwarding enabled.

Make sure that you have carefully followed our instructions above. If you are on Windows, make sure that

- You have **installed** Xming and PuTTY
- You have **started up Xming** (important!) and it is **still running**
- You have enabled X11 forwarding
- You have verified that you can ssh in with X11 forwarding and start up an xterm

If you cannot start up an xterm, you will not be able to run wireshark!

VirtualBox-specific Instructions

VirtualBox Networking

The easiest way to ssh into a VirtualBox VM is to use a **host-only** network interface, as we suggest.

To allow network access in the VM, execute:

```
sudo dhclient eth1
```

which grabs an IP address. If eth1 is a **host-only** network interface, you are all set and should be able to ssh to eth1's IP address from your host.

Otherwise, you can change the default, NAT interface as shown below and execute *sudo dhclient eth0*.

In order to log in via a NAT interface, we need to set up port forwarding. This may also enable users on your LAN to log in to the VM, which you may or may not want.

OS X / Linux Users Run the VBoxManage commands below from a local terminal (not the VM window).

Windows Users If you are using Windows Vista, first close Virtual box, to ensure that config files aren't locked. Then open a command line prompt, by click on 'Start', then 'Run', then typing 'cmd' and hitting enter. In the terminal that appears, change to the VirtualBox directory:

```
cd "C:\Program Files\Sun\xVM VirtualBox"
```

All OSes: From a terminal on your local machine (NOT the VM/VirtualBox), run:

```
VBoxManage setextradata OpenFlowTutorial "VBoxInternal/Devices/pcnet/0/LUN#0/Config/ssh/HostPort" 2222
VBoxManage setextradata OpenFlowTutorial "VBoxInternal/Devices/pcnet/0/LUN#0/Config/ssh/GuestPort" 22
VBoxManage setextradata OpenFlowTutorial "VBoxInternal/Devices/pcnet/0/LUN#0/Config/ssh/Protocol" TCP
```

Verify the settings:

```
VBoxManage getextradata OpenFlowTutorial enumerate
```

The three values you added above should be shown.

Save your network settings for future reboots, by powering down the VM, from the VM window:

```
sudo poweroff
```

Restart the OpenFlowTutorial VM once again, by clicking 'Start' at the top of VirtualBox. Login again with the correct user name and password.

Windows Users You may need to change the VM configuration. Edit /etc/network/interfaces. Check that it says:

```
allow hot-plug eth1
iface eth1 inet dhcp
```

If it says eth0, change it to eth1; note that to write the file you'll need to use sudo. If you changed the interfaces file, poweroff the machine (sudo poweroff) and then start it again.

VirtualBox SSH Connections

Mac OS X and Linux

To SSH from the local machine to the VM, on the local machine run:

```
ssh -Y -l  <user name> -p 2222 localhost
```

with the correct user name and password. Test the X11 forwarding by running on the local machine:

```
xterm
```

and a new terminal window should appear. If you have succeeded, you are done with the basic setup. Close the xterm.

Windows XP

Start Xming; nothing exciting will happen but it will show an icon on the right of the toolbar.

Open a terminal: click 'Start', 'run', then enter 'cmd'.

Change to the directory where you saved putty.

```
cd <dir>
```

Run:

```
putty.exe -X -P 2222 -l <user name> localhost
```

A new window will pop up; type in the password. Now, type:

```
xterm
```

A white terminal window should appear. If you have succeeded, you are done with the basic setup. Close the xterm.

Make sure you use the correct user name and password for your VM image.

Retrieved from "http://archive.openflow.org/wk/index.php?title=OpenFlow_Tutorial&oldid=12981"

Quick Navigation

- OpenFlow Specs
- Bug Tracking
- Wiki
- Legal
- Log in

OpenFlow White Paper

Download the OpenFlow Whitepaper (PDF)

OpenFlow Specification

Download v1.1.0 Implemented (PDF)

Wiki Tools

Personal tools

- Log in

Navigation

- OpenFlow.org
- OpenFlow wiki
- Recent changes
- Random page
- Help

Projects

- SIGCOMM Demo
- GENI Demo

Toolbox

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link