geni
Exploring Networks of the Future
EXPLORE FUTURE INTERNETS AT SCALE
NETWORK SCIENCE AND ENGINEERING
UNDERSTAND
TRANSFORM
INNOVATE

Programming OpenFlow Resources

http://groups.geni.net/geni/wiki/GENIExperimenter/Tutorials/OpenFlowOVS
<20150306>

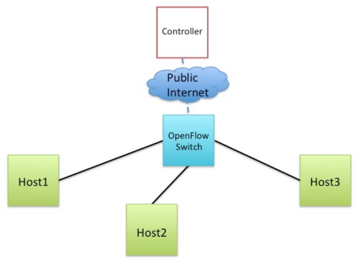
Intro to OpenFlow Tutorial

Overview:

This is a simple OpenFlow tutorial that will guide you through the writing of simple OpenFlow controllers to showcase some of the OpenFlow capabilities. We are going to write three different controllers:

1. Write a controller that will **duplicate all the traffic** of the OpenFlow switch out a specific port
2. **TCP Port Forward** controller. Divert all traffic destined to host A on TCP port X to TCP port Y
3. **Proxy Controller**. Write a controller that will divert all traffic destined to host A, TCP port X to host B, TCP port Y

In this tutorial we have a choice of using an ***OpenFlow Software Switch***, Open vSwitch (OVS), or using an ***OpenFlow-Capable Hardware Switch***. The general topology is as pictured below. In general, the controller just needs to have a public IP address, so that it can exchange messages with the OpenFlow switch. The controller for the switch can run anywhere in the Internet. For this tutorial we are going to use a POX based controller, which is just one example of many controller frameworks.



Prerequisites:

- A GENI account, if you don't have one sign up!
- Familiarity with how to reserve GENI resources with any of the GENI Tools (GENI Experimenter Portal, Omni, Flack). If you don't know you can take any of the tutorials:
    - Reserving resources using Flack tutorial
    - Reserving resources using Omni tutorial
- Familiarity with logging in to GENI compute resources.
- Basic understanding of OpenFlow. If you are doing this tutorial at home, flip through the tutorial's slides
- Familiarity with the Unix Command line
- Familiarity with the python programming language. We are going to use the POX controller, which is just one example of many controller frameworks, and POX is written in python.

Tools:

- Open vSwitch. OVS will be be installed. Installation was completed as described here.
- POX controller. POX controller is installed as part of the resource reservation.

Where to get help:

- If you need help with GENI, email help@geni.net
- If you have questions about OpenFlow, OVS, Pox you can subscribe to openflow-discuss or any of the other mailing lists listed.

Resources:

- Learn more about OpenFlow
- POX wiki
- Learn more about OVS

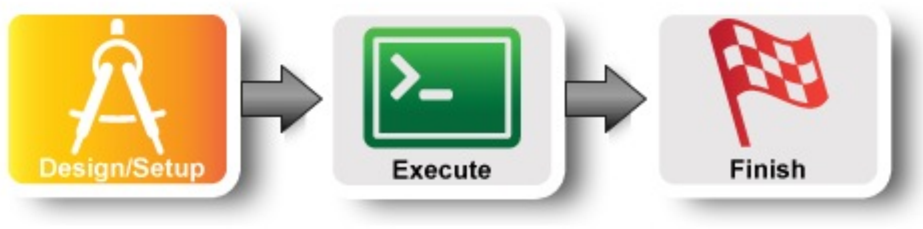
Tutorial Instructions

- Part I: Design/Setup
  - Step 1: Reserve Resources
    - OpenFlow using Open vSwitch (OVS): Reserve topology in one rack [Recommended]
    - OpenFlow using a Hardware Switch: Reserve topology in one rack using the HW OF switch
  - Step 2: Configure and Initialize Services
- Part II: Execute
  - Step 3: Execute Experiment
- Part III: Finish
  - Step 4: Teardown Experiment

Attachments

- IntroToOpenFlow_140123.pptx (2.6 MB) - added by *sedwards@bbn.com* 13 months ago.
  "Intro To Openflow slides"

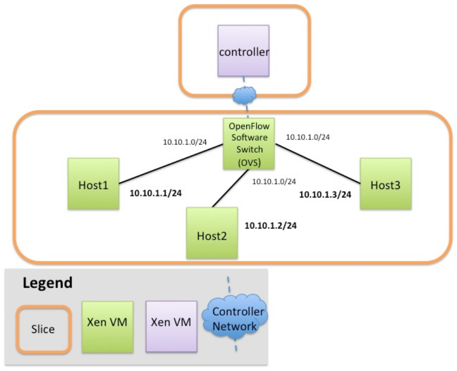
Intro to OpenFlow Tutorial (OVS)



Step 1. Obtain resources

This tutorial can use compute resources from any InstaGENI rack. For a list of available InstaGENI racks see the GENI Production Resources page. If doing this outside a tutorial, use *Utah DDC InstaGENI*. The experiment will need:

- 1 Xen VM with a public IP to run an OpenFlow controller
- 1 Xen VM to be the OpenFlow switch
- 3 Xen VMs as hosts



In this tutorial we are going to use Open vSwitch (OVS) as an OpenFlow switch connected to three hosts. OVS is a software switch running on a compute resource. The other three hosts can only communicate through the OVS switch.

If you are attending a Tutorial, the resources might have been reserved for you, check with your instructor and skip this step. You can use any reservation tool you want to reserve this topology. We will need two slices for this tutorial:

- A slice with a single VM that runs your OpenFlow controller
- A slice with your compute resources including a VM with OVS installed.

To reserve resources use your favorite resource reservation tool (Omni, Portal, jFed):

1. In your slice that will run the OpenFlow controller: Reserve a VM running the controller using the request RSpec http://www.gpolab.bbn.com/exp/OpenFlowOVS/pox-controller.rspec. This RSpec is available in the Portal and is called **XEN VM POX Ctrl**
2. In the slice that will run your hosts: Reserve the topology using the request rspec http://www.gpolab.bbn.com/experiment-support/OpenFlowOVS/openflowovs-all-xen.rspec.xml. This RSpec is available in the Portal and is called **OpenFlow OVS all XEN**

Step 2. Configure and Initialize

Although OVS is installed and initialized on the host that is meant to act as a software switch, it has not been configured yet. There are two main things that need to be configured: *(1) configure your software switch with the interfaces as ports* and *(2) point the switch to an OpenFlow controller*.

In order to configure the OVS switch, we first login to the host that will be used as an OpenFlow switch.

Depending on which tool and OS you are using there is a slightly different process for logging in. If you don't know how to SSH to your reserved hosts learn how to login.

2a. Configure the Software Switch (OVS Window)

Now that you are logged in, we need first to configure OVS. To save time in this tutorial, we have already started OVS and we have added an Ethernet bridge that will act as our software switch. Try the following to show the configured bridge:

```
sudo ovs-vsctl list-br
```

You should see only one bridge `br0`. Now we need to add the interfaces to this bridge that will act as the ports of the software switch.

1. List all the interfaces of the node
   - `ifconfig`

   Write down the interface names that correspond to the connections to your hosts. This information will be needed for one of the exercises.
   The correspondence is:
   - Interface with IP "10.10.1.11" to host1 - ethX
   - Interface with IP "10.10.1.12" to host2 - ethY
   - Interface with IP "10.10.1.13" to host3 - ethZ

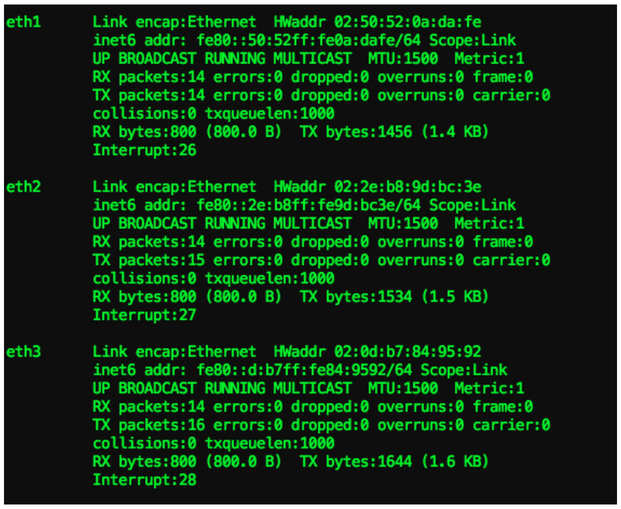
- Be careful **not to bring down eth0**. This is the control interface, if you bring that interface down you **won't be able to login** to your host. For all interfaces other than `eth0` and `l0`, remove the IP from the interfaces (your interface names may vary):

  - `sudo ifconfig ethX 0`

  - `sudo ifconfig ethY 0`

  - `sudo ifconfig ethZ 0`

- Add all the data interfaces to your switch (bridge):Be careful **not to add interface eth0**. This is the control interface. The other three interfaces are your data interfaces. (Use the same interfaces as you used in the previous step.)
    - `sudo ovs-vsctl add-port br0 ethX`
    - `sudo ovs-vsctl add-port br0 ethY`
    - `sudo ovs-vsctl add-port br0 ethZ`



```
eth1      Link encap:Ethernet  HWaddr 02:50:52:0a:da:fe
          inet6 addr: fe80::50:52ff:fe0a:dafe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:14 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:800 (800.0 B)  TX bytes:1456 (1.4 KB)
          Interrupt:26

eth2      Link encap:Ethernet  HWaddr 02:2e:b8:9d:bc:3e
          inet6 addr: fe80::2e:b8ff:fe9d:bc3e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:800 (800.0 B)  TX bytes:1534 (1.5 KB)
          Interrupt:27

eth3      Link encap:Ethernet  HWaddr 02:0d:b7:84:95:92
          inet6 addr: fe80::d:b7ff:fe84:9592/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:800 (800.0 B)  TX bytes:1644 (1.6 KB)
          Interrupt:28
```

Congratulations! You have configured your software switch. To verify the three ports configured run:

```
sudo ovs-vsctl list-ports br0
```

2c. Point your switch to a controller

In the controller window, find the control interface IP of your controller, use *ifconfig* and note down the IP address of `eth0`.

An OpenFlow switch will not forward any packet unless instructed by a controller. Basically the forwarding table is empty, until an external controller inserts forwarding rules. The OpenFlow controller communicates with the switch over the control network and it can be anywhere in the Internet as long as it is reachable by the OVS host.

In order to point our software OpenFlow switch to the controller, in the *ovs* window, run:

```
sudo ovs-vsctl set-controller br0 tcp:<controller_ip>:6633
```

`standalone` vs `secure` mode

The OpenFlow controller is responsible for setting up all flows on the switch, which means that when the controller is not running there should be no packet switching at all. Depending on the setup of your network, such a behavior might not be desired. It might be best that when the controller is down, the switch should default back to being a learning layer 2 switch. In other circumstances however this might be undesirable. In OVS this is a tunable parameter, called `fail-safe-mode` which can be set to the following parameters:

- `standalone` [default]: in this case OVS will take responsibility for forwarding the packets if the controller fails
- `secure`: in this case only the controller is responsible for forwarding packets, and if the controller is down all packets are dropped.

In OVS when the parameter is not set it falls back to the `standalone` mode. For the purpose of this tutorial we will set the `fail-safe-mode` to `secure`, since we want to be the ones controlling the forwarding. Run:

```
sudo ovs-vsctl set-fail-mode br0 secure
```

You can verify your OVS settings by issuing the following:

```
sudo ovs-vsctl show
```

Prev: Introduction

Next: Execute

Intro to OpenFlow Tutorial

Step 3. Execute Experiment

Now that the switch is up and running we are ready to start working on the controller. For this tutorial we are going to use the POX controller. The software is already installed in the controller host for running POX and can also be found here.

3a. Login to your hosts

To start our experiment we need to ssh all of our hosts.

To get ready for the tutorial you will need to have the following windows open:

- one window with ssh into the controller
- four windows with ssh into OVS
- one window with ssh into host1
- two windows with ssh into host2
- one window with ssh into host3

Depending on which tool and OS you are using there is a slightly different process for logging in. If you don't know how to SSH to your reserved hosts learn how to login. Once you have logged in follow the rest of the instructions.

3b. Use a Learning Switch Controller

In this example we are going to run a very simple learning switch controller to forward traffic between `host1` and `host2`.

1. First start a ping from `host1` to `host2`, which should timeout, since there is no controller running.

```
ping host2 -c 10
```

2. We have installed the POX controller under `/tmp/pox` on the controller host. POX comes with a set of example modules that you can use out of the box. One of the modules is a learning switch. Start the learning switch controller which is already available by running the following two commands:

"l2" below uses the letter `l` as in level and is not the number one. And you should wait for the '''INFO ... connected''' line to ensure that the switch and the controller are communicating.

```
cd /tmp/pox
./pox.py --verbose forwarding.l2_learning
```

The output should look like this:

```
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.3/Apr 20 2012 22:39:59)
DEBUG:core:Platform is Linux-3.2.0-56-generic-x86_64-with-Ubuntu-12.04-prec
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[9e-38-3e-8d-42-42 1] connected
DEBUG:forwarding.l2_learning:Connection [9e-38-3e-8d-42-42 1]
```

In the event that you need to move the port of your controller, this is the command -

```
sudo ./pox.py --verbose openflow.of_01 --port=443 forwarding.l2_learning
```

Do not forget to tell the ovs switch that the controller will be listening on this new port, i.e change 6633 to 443 in Step 2c.

3. In the terminal of `host1`, ping `host2`:

```
[experimenter@host1 ~]$ ping host2
PING host2-lan1 (10.10.1.2) 56(84) bytes of data.
From host1-lan0 (10.10.1.1) icmp_seq=2 Destination Host Unreachable
From host1-lan0 (10.10.1.1) icmp_seq=3 Destination Host Unreachable
From host1-lan0 (10.10.1.1) icmp_seq=4 Destination Host Unreachable
64 bytes from host2-lan1 (10.10.1.2): icmp_req=5 ttl=64 time=23.9 ms
64 bytes from host2-lan1 (10.10.1.2): icmp_req=6 ttl=64 time=0.717 ms
64 bytes from host2-lan1 (10.10.1.2): icmp_req=7 ttl=64 time=0.654 ms
64 bytes from host2-lan1 (10.10.1.2): icmp_req=8 ttl=64 time=0.723 ms
64 bytes from host2-lan1 (10.10.1.2): icmp_req=9 ttl=64 time=0.596 ms
```

   Now the ping should work.

4. If you are using OVS, go back to your OVS host and take a look at the print outs. You should see that your controller installed flows based on the mac addresses of your packets.

There is no way to get this information from the OpenFlow-capable hardware switch.

5. If you are using OVS, to see the flow table entries on your OVS switch:

   ```
   sudo ovs-ofctl dump-flows br0
   ```

   You should see at least two table entries: One for ICMP Echo (icmp_type=8) messages from host1 to host2 and one for ICMP Echo Reply (icmp_type=0) messages from host2 to host1. You may also see flow entries for arp packets.

6. To see messages go between your switch and your controller, open a new ssh window to your controller node and run tcpdump on the `eth0` interface and on the tcp port that your controller is listening on usually 6633. (You can also run `tcpdump` on the `OVS` control interface if you desire. However, when using the hardware switch, you can only do the `tcpdump` on your controller host.)

   ```
   sudo tcpdump -i eth0 tcp port 6633
   ```

   You will see (1) periodic keepalive messages being exchanged by the switch and the controller, (2) messages from the switch to the controller (e.g. when there is a table miss) and an ICMP Echo message in, and (3) messages from the controller to the switch (e.g. to install new flow entries).

7. Kill your POX controller by pressing `Ctrl-C`:

   ```
   DEBUG:forwarding.l2_learning:installing flow for 02:c7:e8:a7:40:65.1 -
   INFO:core:Going down...
   INFO:openflow.of_01:[3a-51-a1-ab-c3-43 1] disconnected
   INFO:core:Down.
   ```

8. Notice what happens to your ping on host1.

9. If you are using OVS, check the flow table entries on your switch:

   ```
   sudo ovs-ofctl dump-flows br0
   ```

   Since you set your switch to "secure" mode, i.e. don't forward packets if the controller fails, you will not see flow table entries. If you see flow table entries, try again after 10 seconds to give the entries time to expire.

Soft vs Hard Timeouts

All rules on the switch have two different timeouts:

- **Soft Timeout**: This determines for how long the flow will remain in the forwarding table of the switch if there are no packets received that match the specific flow. As long as packets from that flow are received the flow remains on the flow table.
- **Hard Timeout**: This determines the total time that a flow will remain at the forwarding table, independent of whether packets that match the flow are received; i.e. the flow will be removed after the hard timeout expires.

Can you tell now why there were packets flowing even after you killed your controller?

Useful Tips for writing your controller

In order to make this first experience of writing a controller easier, we wrote some helpful functions that will abstract some of the particularities of POX away. These functions are located in `/tmp/pox/ext/utils.py`, so while you write your controller consult this file for details.

Functions that are implemented include:

- packetIsIP : Test if the packet is IP
- packetIsARP : Test if the packet is ARP
- packetIsRequestARP : Test if this is an ARP Request packet
- packetIsReplyARP : Test if this is an ARP Reply packet
- packetArpDstIp : Test what is the destination IP in an ARP packet
- packetArpSrcIp : Test what is the sources IP in an ARP packet
- packetIsTCP : Test if a packet is TCP
- packetDstIp : Test the destination IP of a packet
- packetSrcIp : Test the source IP of a packet
- packetDstTCPPort : Test the destination TCP port of a packet
- packetSrcTCPPort : Test the source TCP port of a packet
- createOFAction : Create one OpenFlow action
- getFullMatch : get the full match out of a packet
- createFlowMod : create a flow mod
- createArpRequest : Create an Arp Request for a different destination IP
- createArpReply : Create an Arp Reply for a different source IP

3c. Debugging your Controller

While you are developing your controller, some useful debugging tools are:

i. Print messages

Run your controller in verbose mode (add --verbose) and add print messages at various places to see what your controller is seeing.

ii. Check the status in the switch

If you are using an OVS switch, you can dump information from your switch. For example, to dump the flows:

```
sudo ovs-ofctl dump-flows br0
```

Two other useful commands show you the status of your switch:

```
sudo ovs-vsctl show
sudo ovs-ofctl show br0
```

iii. Use Wireshark to see the OpenFlow messages

Many times it is useful to see the OpenFlow messages being exchanged between your controller and the switch. This will tell you whether the messages that are created by your controller are correct and will allow you to see the details of any errors you might be seeing from the switch. If you are using OVS then you can use wireshark on both ends of the connection, in hardware switches you have to rely only on the controller view.

The controller host and OVS has wireshark installed, including the openflow dissector. For more information on wireshark you can take a look at the wireshark wiki.

Here we have a simple case of how to use the OpenFlow dissector for wireshark.

If you are on a Linux friendly machine (this includes MACs) open a terminal and ssh to your controller machine using the -Y command line argument, i.e.

```
ssh -Y <username>@<controller>
```

Assuming that the public IP address on the controller is eth0, run wireshark by typing:

```
sudo wireshark -i eth0&
```

When the wireshark window pops up, you might still have to choose eth0 for a live capture. And you will want to use a filter to cut down on the chatter in the wireshark window. One such filter might be just seeing what shows up on port 6633. To do that type *tcp.port eq 6633* in the filter window, assuming that 6633 is the port that the controller is listening on. And once you have lines, you can choose one of the lines and choose "Decode as ...." and choose the *OFP protocol*.

3d. Run a traffic duplication controller

In the above example we ran a very simple learning switch controller. The power of OpenFlow comes from the fact that you can decide to forward the packet anyway you want based on the supported OpenFlow actions. A very simple but powerful modification you can do, is to duplicate all the traffic of the switch out a specific port. This is very useful for application and network analysis. You can imagine that at the port where you duplicate traffic you connect a device that does analysis. For this tutorial we are going to verify the duplication by doing `tcpdump` on two ports on the OVS switch.

1. Use the interfaces that are connected to `host2` and `host3`.
    - Software Switch (OVS): If you have not noted them down you can use the manifest and the MAC address of the interfaces (ovs:if1 and ovs:if2) to figure this out. But you should have noted down the interfaces in Section 2 when you were configuring the software switch. Run tcpdump on these interfaces; one in each of the two ovs terminals you opened. This will allow you to see all traffic going out the interfaces.
    - Hardware Switch: Refer to this Section to figure out ports: UsefulTips. If you are using a hardware switch, you may not see the traffic on host3, but if you observe your controller output, you will notice that flows are being installed for forwarding to host2 and host3.

    To see that duplication is happening, on the ovs host, run:

    ```
    sudo tcpdump -i <data_interface_name>  [data_interface to host2]
    sudo tcpdump -i <data_interface_name>  [data_interface to host3]
    ```

You should see traffic from host1 to host2 showing up in the tcpdump window for host3. As a comparison, you will notice that no traffic shows up in that window when the controller is running the learning switch.

2. In the controller host directory `/tmp/pox/ext` you should see two files:

 i. myDuplicateTraffic.py : This is the file that has instructions about how to complete the missing information. Go ahead and try to implement your first controller.
 ii. DuplicateTraffic.py : This has the actual solution. You can just run this if you don't want to bother with writing a controller.

3. Run your newly written controller on the <data_interface_name> that corresponds to *OVS:if2* (which is connected to `host3`):

```
cd /tmp/pox
./pox.py --verbose myDuplicateTraffic --duplicate_port=?
```

4. To test it go to the terminal of host1 and try to ping host2:

```
ping 10.10.1.2
```

   If your controller is working, your packets will register in both terminals running tcpdump.

5. Stop the POX controller:

```
DEBUG:myDuplicateTraffic:Got a packet : [02:f1:ae:bb:e3:a8>02:c7:e8:a7
DEBUG:SimpleL2Learning:installing flow for 02:f1:ae:bb:e3:a8.2 -> 02:c

INFO:core:Going down...
INFO:openflow.of_01:[3a-51-a1-ab-c3-43 1] disconnected
INFO:core:Down.
```

3d. Run a port forward Controller

Now let's do a slightly more complicated controller. OpenFlow gives you the power to overwrite fields of your packets at the switch, for example the TCP source or destination port and do port forwarding. You can have clients trying to contact a server at port 5000, and the OpenFlow switch can redirect your traffic to a service listening on port 6000.

1. Under the `/tmp/pox/ext` directory there are two files PortForwarding.py and myPortForwarding.py that are similar like the previous exercise. Both of these controller are configured by a configuration file at `ext/port_forward.config`. Use myPortForwarding.py to write your own port forwarding controller.

2. To test your controller we are going to use netcat. Go to the two terminals of host2. In one terminal run:

```
nc -l 5000
```

   and in the other terminal run

```
nc -l 6000
```

3. Now, start the simple layer 2 forwarding controller. We are doing this to see what happens with a simple controller.

```
cd /tmp/pox
```

```
./pox.py --verbose forwarding.l2_learning
```

4. Go to the terminal of host1 and connect to host2 at port 5000:

   ```
   nc 10.10.1.2 5000
   ```

5. Type something and you should see it at the the terminal of host2 at port 5000.

6. Now, stop the simple layer 2 forwarding controller:

   ```
   DEBUG:forwarding.l2_learning:installing flow for 02:d4:15:ed:07:4e.3 -

   INFO:core:Going down...
   INFO:openflow.of_01:[36-63-8b-d7-16-4b 1] disconnected
   INFO:core:Down.
   ```

7. And start your port forwarding controller:

   ```
   ./pox.py --verbose myPortForwarding
   ```

8. Repeat the netcat scenario described above. Now, your text should appear on the other terminal of host2 which is listening to port 6000.

9. Stop your port forwarding controller:

   ```
   DEBUG:myPortForwarding:Got a packet : [02:aa:a3:e8:6c:db>33:33:ff:e8:6

   INFO:core:Going down...
   INFO:openflow.of_01:[36-63-8b-d7-16-4b 1] disconnected
   INFO:core:Down.
   ```

3e. Run a Server Proxy Controller

As our last exercise, instead of diverting the traffic to a different server running on the same host, we will divert the traffic to a server running on a different host and on a different port.

1. Under the `/tmp/pox/ext/` directory there are two files Proxy.py and myProxy.py that are similar like the previous exercise. Both of these controllers are configured by the configuration file `proxy.config`. Use myProxy.py to write your own proxy controller.

2. On the terminal of `host3` run a netcat server:

   ```
   nc -l 7000
   ```

3. On your controller host, open the /tmp/pox/ext/myProxy.py file, and edit it to implement a controller that will divert traffic destined for `host2` to `host3`. Before you start implementing think about what are the side effects of diverting traffic to a different host.
   - Is it enough to just change the IP address?
   - Is it enough to just modify the TCP packets?

     If you want to see the solution, it's available in file /tmp/pox
     /ext/Proxy.py file.

4. To test your proxy controller run:

```
cd /tmp/pox
./pox.py --verbose myProxy
```

5. Go back to the terminal of `host1` and try to connect netcat to `host2` port 5000

```
nc 10.10.1.2 5000
```

6. If your controller works correctly, you should see your text showing up on the terminal of `host3`.

4. Moving to a Hardware Switch

To try your controller with a GENI Hardware OpenFlow switch:

- Delete resources in your slice with the compute resources. **Do not** delete resources in your slice with the controller.
- Follow the instructions at OpenFlow Design and Setup for Hardware Switch

If you do not want to do the Hardware OpenFlow portion of the tutorial, proceed to Finish

---

Prev: Design and Setup for OVS

Prev: Design and Setup for Hardware Switch

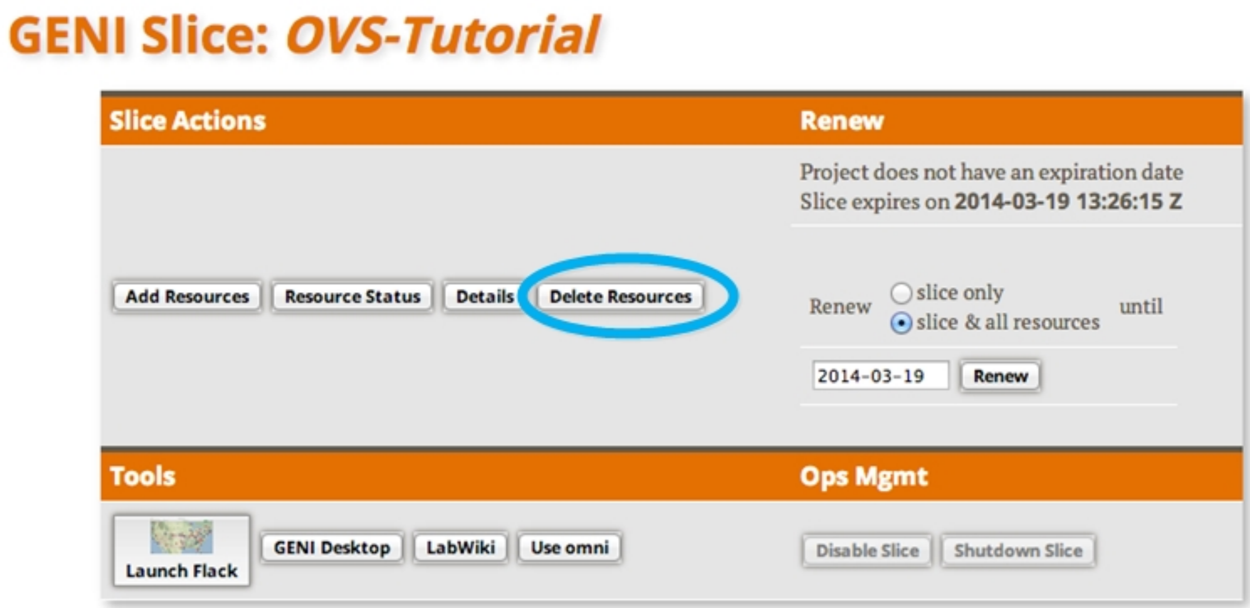
Next: Finish

```
#!/usr/bin/python

#----------------------------------------------------------------------
-
# Copyright (c) 2013 Raytheon BBN Technologies
#
# Permission is hereby granted, free of charge, to any person
obtaining
# a copy of this software and/or hardware specification (the "Work")
to
# deal in the Work without restriction, including without limitation
the
# rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Work, and to permit persons to whom the
Work
# is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Work.
#
# THE WORK IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
# OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
# HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
# WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE WORK OR THE USE OR OTHER DEALINGS
# IN THE WORK.
#----------------------------------------------------------------------

#File: /tmp/pox/ext/utils.py

from pox.core import core

import ConfigParser
import os
import sys

from pox.lib.packet.tcp import tcp
from pox.lib.packet.arp import arp
from pox.lib.packet.ipv4 import ipv4
from pox.lib.packet.ethernet import ethernet
from pox.lib.packet.ethernet import ETHER_BROADCAST

import pox.openflow.libopenflow_01 as of
from pox.lib.addresses import IPAddr

def getOpenFlowPort(connection, port_name) :
  phy_port = connection.ports[port_name]
  if phy_port is not None:
    return phy_port.port_no
  return -1

def readConfigFile(filename, logger) :
  config = None
  logger.debug("Looking for configuration file %s" % filename)
  filename = os.path.expanduser(filename)
  if not os.path.exists(filename):
    logger.error("Configuration file %s does not exist! Exit!" %
(filename))
    sys.exit(-1)
  logger.debug("Using configuration file %s" % filename)

  confparser = ConfigParser.RawConfigParser()
  try:
    confparser.read(filename)
  except ConfigParser.Error as exc:
    logger.warning("Config file %s could not be parsed: %s"
                    % (filename, str(exc)))

  # Create a dictionary from the configuration
  # - each section is a key in the dictionary that it's value
  # is a dictionary with (key, value) pairs of configuration
  # parameters
  config = {}
  for sec in confparser.sections():
    config[sec] = {}
    for (key,val) in confparser.items(sec):
      config[sec][key] = val

  logger.debug(config)
  return config

def packetIsIP(packet, logger) :
  if isinstance(packet, ethernet):
    return isinstance(packet.next, ipv4)
  return False

def packetIsARP(packet, logger) :
  if isinstance(packet, ethernet):
    return isinstance(packet.next, arp)
  return False

def packetIsRequestARP(packet, logger) :
  if packetIsARP(packet, logger) :
     if packet.next.opcode == arp.REQUEST:
       return True
  return False

def packetIsReplyARP(packet, logger) :
  if packetIsARP(packet, logger) :
     if packet.next.opcode == arp.REPLY:
       return True
  return False

def packetArpDstIp(packet, dstip, logger):
  if packetIsARP(packet, logger) :
     if packet.next.protodst == dstip:
       return True
  return False

def packetArpSrcIp(packet, srcip, logger):
  if packetIsARP(packet, logger) :
     if packet.next.protosrc == srcip:
       return True
  return False

def packetIsTCP(packet, logger) :
  if packetIsIP(packet, logger):
      return isinstance(packet.next.next, tcp)
  return False

def packetDstIp(packet, ipaddr, logger) :
  if packetIsIP(packet, logger):
    if not cmp(packet.next.dstip, ipaddr):
       return True
  return False

def packetSrcIp(packet, ipaddr, logger) :
  if packetIsIP(packet, logger):
    if not cmp(packet.next.srcip, ipaddr):
       return True
  return False

def packetDstTCPPort(packet, tcpport, logger) :
  if packetIsTCP(packet, logger):
    dsttcpportstr = packet.next.next.dstport
    if dsttcpportstr == tcpport :
       return True
  return False

def packetSrcTCPPort(packet, tcpport, logger) :
  if packetIsTCP(packet, logger):
    srctcpportstr = packet.next.next.srcport
    if srctcpportstr == tcpport :
       return True
  return False

def createOFAction(action_type, arg, logger) :
  if action_type == of.OFPAT_OUTPUT :
  # XXX Check if arg is a list
    logger.debug("Creating output action to %d" % arg)

    return of.ofp_action_output(port = arg)
  if action_type == of.OFPAT_SET_DL_SRC :
    return of.ofp_action_dl_addr.set_src(arg)
  if action_type == of.OFPAT_SET_DL_DST :
    return of.ofp_action_dl_addr.set_dst(arg)
  if action_type == of.OFPAT_SET_NW_SRC :
    return of.ofp_action_nw_addr.set_src(arg)
  if action_type == of.OFPAT_SET_NW_DST :
    return of.ofp_action_nw_addr.set_dst(arg)
  if action_type == of.OFPAT_SET_TP_SRC :
    return of.ofp_action_tp_port.set_src(arg)
  if action_type == of.OFPAT_SET_TP_DST :
    return of.ofp_action_tp_port.set_dst(arg)

  logger.warn("Type %d not supported" % action_type)
  return None


def getFullMatch(packet, inport) :
   return of.ofp_match.from_packet(packet, inport)

def createFlowMod(match, actions, hard_timeout, idle_timeout,
buffid=None) :
    msg = of.ofp_flow_mod(command=of.OFPFC_ADD,
                          idle_timeout=idle_timeout,
                          hard_timeout=hard_timeout,
                          buffer_id=buffid,
                          actions=actions,
                          match=match)
    return msg



def createArpRequest(packet, ip, logger):
  if not packetIsARP(packet, logger):
     logger.warn("Packet is not ARP")
     return
  origarp = packet.next
  arppkt = arp()
  arppkt.hwsrc      = origarp.hwsrc
  arppkt.hwdst      = origarp.hwdst
  arppkt.hwlen      = origarp.hwlen
  arppkt.opcode     = arp.REQUEST
  arppkt.protolen   = origarp.protolen
  arppkt.protosrc   = origarp.protosrc
  arppkt.protodst   = IPAddr(ip)
  pkt = ethernet()
  pkt.set_payload(arppkt)
  pkt.type = ethernet.ARP_TYPE
  pkt.src = arppkt.hwsrc
  pkt.dst = ETHER_BROADCAST
  return pkt

def createArpReply(packet, ip, logger):
  if not packetIsARP(packet, logger):
     logger.warn("Packet is not ARP")
     return
  origarp = packet.next
  arppkt = arp()
  arppkt.hwsrc      = origarp.hwsrc
  arppkt.hwdst      = origarp.hwdst
  arppkt.hwlen      = origarp.hwlen
  arppkt.opcode     = arp.REPLY
  arppkt.protolen   = origarp.protolen
  arppkt.protosrc   = IPAddr(ip)
  arppkt.protodst   = origarp.protodst
  pkt = ethernet()
  pkt.set_payload(arppkt)
  pkt.type = ethernet.ARP_TYPE
  pkt.src = arppkt.hwsrc
  pkt.dst = arppkt.hwdst
  return pkt
```

Intro to OpenFlow Tutorial

Step 4. Teardown Experiment

After you are done with this experiment release your resources. In the GENI Portal select the slice click on the "Delete Resources" button:



Project does not have an expiration date
Slice expires on **2014-03-19 13:26:15 Z**

Renew ○ slice only [✓ ● slice & all resources] until

2014-03-19 Renew

If you have used other tools to run this experiment than release resources as described in the Prerequisites for Tutorials on reservation tools pages.

Now you can start designing and running your own experiments!

Prev: Execute

Introduction