In this exercise, you will be working with **Pyretic**, a new programmer-friendly domain-specific language embedded in Python. It provides a runtime system that enables programmers to specify network policies at a high level of abstraction, compose them together in a variety of ways, and execute them on abstract topologies. As in week 4, this week you will be taken through the steps of writing network applications i.e., hub and layer 2 MAC learning on Pyretic and testing them using Mininet. The purpose of this exercise is to show you that there is no single way of writing such applications. New runtime systems are being developed, which provide richer abstractions and tools to express your network applications --- Pyretic is one such example.

After the walkthrough, You will be asked to re-implement and submit the Layer 2 firewall you encoded in POX last week using the higher-level constructs offered by Pyretic. More details on creating and submitting the code will be provided later on in the instructions. So, as always, make sure that you follow each step carefully.

Overview

The network you'll use in this exercise includes 3 hosts and a switch, this time with the Pyretic runtime to express your network applications.

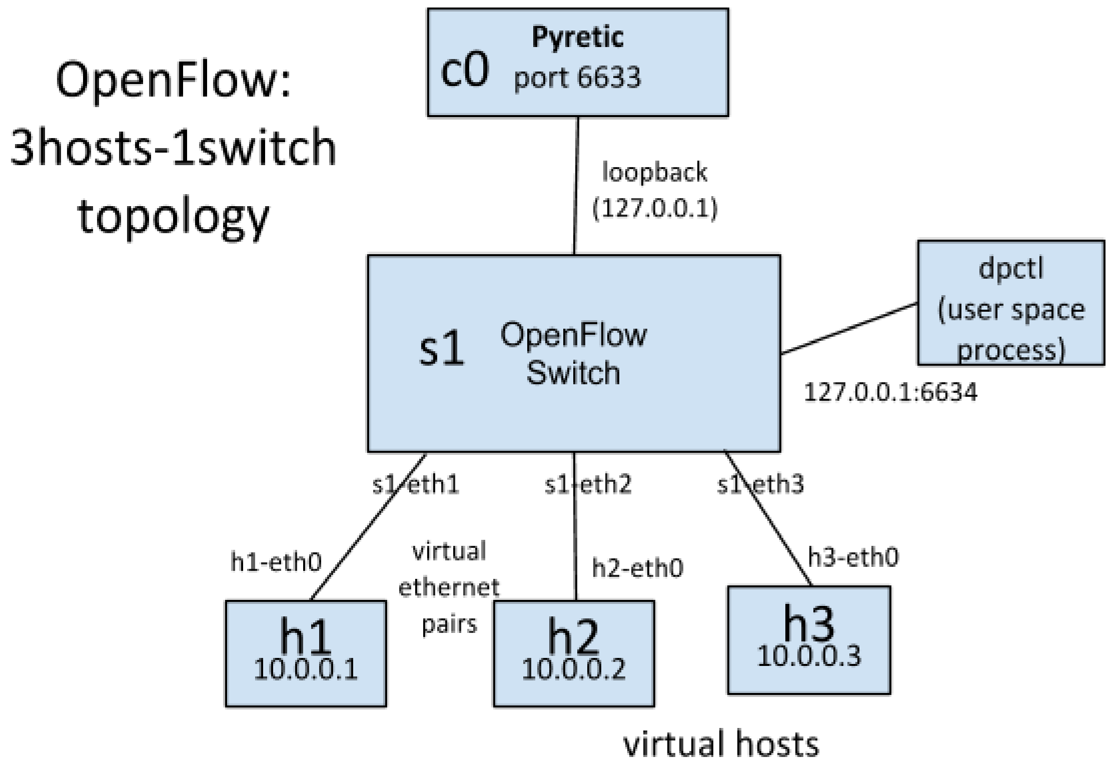

Figure 1: Topology for the Network under Test

As mentioned above, Pyretic is a new language and system that enables modular programming by:

- Defining composition operators and a library of policies for forwarding and querying traffic. Pyretic's parallel composition operator allows multiple policies to operate on the same set of packets, while itsl sequential composition operator allows one policy to process packets after another.

- Pyretic enables each policy to operate on an abstract topology that implicitly constrains what the module can see and do.

- Pyretic provides a rich abstract packet model that allows programmers to extend packets with virtual fields that may be used to associate packets with high-level meta-data.

For more details on Pyretic, see http://www.frenetic-lang.org/pyretic/.

We will be using the Pyretic runtime system, so make sure that the default or POX controller is not running in the background. Also, confirm that the port ‘6633’ used to communicate with OpenFlow switches by the runtime is not bounded:

```
$ sudo fuser -k 6633/tcp
```

This will kill any existing TCP connection, using this port.

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

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

The Pyretic runtime comes pre-installed with the provided VM image.

Verify Hub behavior with tcpdump

Now, run the basic hub example:

```
$ pyretic.py -v high pyretic.modules.hub
```

This tells Pyretic to enable verbose logging and to start the hub component. You’ll find the file `pyretic.py` in `~/pyretic` directory.

TIP: start pyretic first for quicker hookup w/ mininet

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, Pyretic will print something like this:

```
OpenFlow switch 1 connected
2013-07-21 13:19:11.276764  | clear_all
2013-07-21 13:19:11.278561  | clear_all
2013-07-21 13:19:11.280613  | clear_all
2013-07-21 13:19:11.281942  | clear_all
```

Now 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 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.

Now, lets look at the hub code (You can see the reduction in the amount of code needed to implement hub in Pyretic as compared to POX):

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *

   """Implement hub-like behavior --- send all packets to all ports on a network
    minimum spanning tree, except for the input port"""

hub = flood()

def main():
    return hub
```

Table 1. hub application

Interestingly, Pyretic’s “flood” operator computes a minimum spanning tree under the hood, so you don’t have to worry about loops in your topology when you call Pyretic’s flood operator.

Verify Switch behavior with tcpdump

This time, let’s verify that hosts can ping each other when the controller is behaving like a layer 2 learning switch. Kill the Pyretic runtime by pressing Ctrl-C in the window running the program.

Now, run the switch example:

```
$ pyretic.py -v high pyretic.modules.mac_learner
```

Like before, 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
```

Here, the switch examines 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.

You can close the xterms now.

Let’s have a look at the mac_learner code (it’s pretty self explanatory, and we went through it in lecture, as well):

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *
from pyretic.lib.query import *

class mac_learner(DynamicPolicy):
    """Standard MAC-learning logic"""
    def __init__(self):
        super(mac_learner,self).__init__()
        self.flood = flood()          # REUSE A SINGLE FLOOD INSTANCE
        self.set_initial_state()

    def set_initial_state(self):
        self.query = packets(1,['srcmac','switch'])
        self.query.register_callback(self.learn_new_MAC)
        self.forward = self.flood  # REUSE A SINGLE FLOOD INSTANCE
        self.update_policy()

    def set_network(self,network):
        self.set_initial_state()

    def update_policy(self):
        """Update the policy based on current forward and query policies"""
        self.policy = self.forward + self.query

    def learn_new_MAC(self,pkt):
        """Update forward policy based on newly seen (mac,port)"""
        self.forward = if_(match(dstmac=pkt['srcmac'],
                                switch=pkt['switch']),
                          fwd(pkt['inport']),
                          self.forward)
        self.update_policy()


def main():
    return mac_learner()
```

Table 2. switch application

Useful Pyretic policies

`match(f=v)`: filters only those packets whose header field f's value matches v
`~A`: negates a match
`A & B`: logical intersection of matches A and B
`A | B`: logical union of matches A and B
`fwd(a)`: forward packet out port a
`flood()`: send all packets to all ports on a network minimum spanning tree, except for the input port
`A >> B`: A's output becomes B's input
`A + B`: A's output and B's output are combined
`if_(M,A,B)`: if packet filtered by M, then use A, otherwise use B

These policies are explained in greater detail in Module 6.4.

Assignment

Background

A Firewall is a network security system that is used to control the flow of ingress and egress traffic usually between a more secure local-area network (LAN) and a less secure wide-area network (WAN). The system analyses data packets for parameters like L2/L3 headers (i.e., MAC and IP address) or performs deep packet inspection (DPI) for higher layer parameters (like application type and services etc) to filter network traffic. A firewall acts as a barricade between a trusted, secure internal network and another network (e.g. the Internet) which is supposed to be not very secure or trusted.

In this assignment, your task is to implement a layer 2 firewall that runs alongside the MAC learning module on the Pyretic runtime. The firewall application is provided with a list of MAC address pairs i.e., access control list (ACLs). When a connection establishes between the controller and the switch, the application installs static flow rule entries in the OpenFlow table to disable all communication between each MAC pair.

Network Topology

Your firewall should be agnostic of the underlying topology. It should take MAC pair list as input and install it on the switches in the network. To make things simple, we will implement a less intelligent approach and will install rules on **all** the switches in the network.

Understanding the Code

To start this exercise, download pyretic-firewall-assignment.zip It consists of three files:

> `pyretic_firewall.py`: a sekleton class which you will update with the logic for installing firewall rules.
> `firewall-policies.csv`: a list of MAC pairs (i.e., policies) read as input by the firewall application.
> `submit.py`: used to submit your code and output to the Coursera servers for grading.

You don't have to do any modifications in `firewall-policies.csv` and `submit.py`.

The `pyretic_firewall.py` is populated with a skeleton code. It consists of a `main` function and a global variable (`policy_file`) that holds the path of the `firewall-policies.csv` file. Whenever a connection is established between the Pyretic controller and the OpenFlow switch the `main` functions gets executed.

Your task is to read the policy file and update the `main` function. The function should install policies in the OpenFlow switch that drop packets whenever a matching src/dst MAC address (for any of the listed MAC pairs) enters the switch

Testing your Code

Once you have your code, copy the `pyretic_firewall.py` in the `~/pyretic/pyretic/examples` directory on your VM. Also in the same directory create the following file:

```
$ cd ~/pyretic/pyretic/examples
$ touch firewall-policies.csv
```

and copy the following lines in it:

```
id,mac_0,mac_1
1,00:00:00:00:00:01,00:00:00:00:00:02
```

This will cause the firewall application to install a flow rule entry to disable all communication between host (h1) and host (h2).

Run Pyretic runtime:

```
$ cd ~
$ pyretic.py pyretic.examples.pyretic_firewall &
```

Now run mininet:

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

In mininet try to ping host (h2) from host (h1):

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

What do you see? If everything has be done and setup correctly then host (h1) should not be able to ping host (h2).

Now try pinging host (h3) from host (h1):

```
mininet> h1 ping -c1 h3
```

What do you see? Host (h1) is able to ping host (h3) as there is no flow rule entry installed in the network to disable the communication between them.

Submitting your Code

The `pyretic_firewall.py` and the provided `firewall-policy.csv` should go into the `~/pyretic/pyretic/examples` directory on your VM.

Also copy the `submit.py` script in the HOME (~/) directory.

Run pyretic:

```
$ pyretic.py pyretic.examples.pyretic_firewall &
```

Method 1:

To submit your code, run the submit.py script:

```
$ sudo python submit.py
```

Your mininet VM should have internet access by default, but still verify that it has internet connectivity (i.e., eth0 set up as NAT). Otherwise `submit.py` will not be able to post your code and output to our coursera servers.

The submission script will ask for your login and password. This password is not the general account password, but an assignment-specific password that is uniquely generated for each student. You can get this from the assignments listing page.

Once finished, it will prompt the results on the terminal (either passed or failed).

Method 2: (alternative for those not able to submit using the above script)

Download and run the following script:

```
$ wget https://d396qusza40orc.cloudfront.net/sdn/srcs/m6-output.py
$ sudo python m6-output.py
```

This will create an `output.log` file in the same folder. Upload this log file on `coursera` using the `submit` button for Module 6 under the Week 5 section on the Programming Assignment page.

Once uploaded, it will prompt the results on the new page (either passed or failed).

Note, if during the execution `submit.py or m6-output.py` scripts crash for some reason or you terminate it using CTRL+C, make sure to clean mininet environment using:

```
$ sudo mn -c
```

Also, if it still complains about the controller running. Execute the following command to kill it:

```
$ sudo fuser -k 6633/tcp
```

---

* Part of these instructions are adapted from http://www.frenetic-lang.org/pyretic/.