Programming Assignment 2: Link & Network Layer Forwarding

Overview

For this assignment, you will implement the forwarding behavior of a switch and a router. Recall that a switch forwards packets based on MAC address, and a router forwards packets based on IP address.
Step 1: Getting Started
Step 2: Implement Virtual Switch
Step 3: Implement Virtual Router

Learning Outcomes

After completing this programming assignment, students should be able to:

- Construct a learning switch that optimally forwards packets based on link layer headers
- Determine the matching route table entry for a given IP address
- Develop a router that updates and forwards packets based on network layer headers

Step 1: Getting Started

You will be using Mininet, POX, and skeleton code for a simple router to complete the assignment. Mininet and POX are already installed in the virtual machine (VM) you used for Assignment 1. You should continue to use this VM for this project. You can always refer back to Step 2 of Assignment 1 if you have questions about using your VM.

Preparing Your Environment

Before beginning this project, there are some additional steps you need to complete to prepare your VM:

1. Install required packages

   ```
   sudo apt-get update
   sudo apt-get install -y python-dev python-setuptools flex
      bison ant openjdk-7-jdk git screen
   ```

2. Install ltprotocol

   ```
   cd ~
   git clone git://github.com/dound/ltprotocol.git
   cd ltprotocol
   sudo python setup.py install
   ```

3. Checkout the appropriate version of POX

```
cd ~
git clone https://github.com/noxrepo/pox
cd ~/pox
git checkout f95dd1
```

4. Get the starter codes and

```
cd ~/progAssign2
```

5. Symlink POX and configure the POX modules

```
cd ~/progAssign2
ln -s ../pox
./config.sh
```

Sample Configuration

The first sample configuration consists of a single switch (s1) and three emulated hosts (h1, h2, h3). The hosts are each running an HTTP server. When you have finished implementing your switch, one host should be able to fetch a web page from any other host (using `wget` or `curl`). Additionally, the hosts should be able to ping each other.
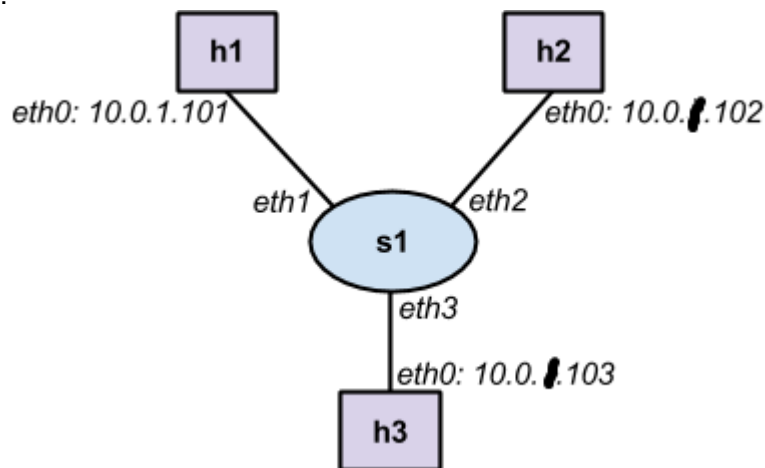
This topology is defined in the configuration file `topos/single_sw.topo`.

Running the Virtual Switch

1. Start Mininet emulation by running the following commands:

```
$ cd ~/progAssign2/
$ sudo python ./run_mininet.py topos/single_sw.topo
```
You should see output like the following:
```
*** Loading topology file topos/single_sw.topo
*** Writing IP file ./ip_config
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
```

```
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Configuring routing for h1
*** Configuring routing for h2
*** Configuring routing for h3
*** Writing ARP cache file ./arp_cache
*** Configuring ARP for h1
*** Configuring ARP for h2
*** Configuring ARP for h3
*** Starting CLI:
mininet>
```
Keep this terminal open, as you will need the mininet command line for
debugging. (Don't press ctrl-z.)

2. Open another terminal. Start the controller, by running the following
commands:

```
cd ~/progAssign2/
./run_pox.sh
```
You should see output like the following:
```
POX 0.0.0 / Copyright 2011 James McCauley
INFO:cs640.ofhandler:Successfully loaded VNet config file
{'h3-eth0': ['10.0.1.103', '255.255.255.0'], 'h1-eth0':
['10.0.1.101', '255.255.255.0'], 'h2-eth0': ['10.0.1.102',
'255.255.255.0']}
INFO:cs640.vnethandler:VNet server listening on
127.0.0.1:8888
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.6/Mar 22 2014 22:59:56)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY. This
program is free software,
and you are welcome to redistribute it under certain
conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on
0.0.0.0:6633
```
You must wait for Mininet to connect to the POX controller before you
continue to the next step. Once Mininet has connected, you will see output
like the following:
```
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-
01
DEBUG:cs640.ofhandler:Connection [Con 1/1]
DEBUG:cs640.ofhandler:dpid=1
INFO:cs640.vnethandler:VNetHandler catch
VNetDevInfo(ifaces={'eth3': (None, None, None, 3), 'eth2':
(None, None, None, 2), 'eth1': (None, None, None,
1)},swid=s1,dpid=1)
```

```
Ready.
POX>
```
Keep POX running. (Don't press `ctrl-z`.)

Note that POX is used "under the hood" in this assignment to direct packets between Mininet and your virtual switch and virtual router instances (i.e., Java processes). You **do not** need to understand, modify, or interact with POX in any way, besides executing the `run_pox.sh` script.

3. Open a third terminal. Build and start the virtual switch, by running the following commands:

```
cd ~/progAssign2/
ant
java -jar VirtualNetwork.jar -v s1
```
You should see output like the following:
```
Connecting to server localhost:8888
Device interfaces:
eth3
eth2
eth1
<-- Ready to process packets -->
```

4. Go back to the terminal where Mininet is running. To issue a command on an emulated host, type the hostname followed by the command in the Mininet console. Only the host on which to run the command should be specified by name; any arguments for the command should use IP addresses. For example, the following command sends 2 ping packets from h1 to h2:

```
mininet> h1 ping -c 2 10.0.1.102
```
The pings will fail because the virtual switch is not fully implemented. However, in the terminal where your virtual switch is running, you should see the following output:
```
*** -> Received packet:
    ip
    dl_vlan: untagged
    dl_vlan_pcp: 0
    dl_src: 00:00:00:00:00:01
    dl_dst: 00:00:00:00:00:02
    nw_src: 10.0.1.101
    nw_dst: 10.0.1.102
    nw_tos: 0
    nw_proto: 1
    icmp_type: 8
    icmp_code: 0
*** -> Received packet:
    ip
    dl_vlan: untagged
    dl_vlan_pcp: 0
    dl_src: 00:00:00:00:00:01
    dl_dst: 00:00:00:00:00:02
    nw_src: 10.0.1.101
    nw_dst: 10.0.1.102
    nw_tos: 0
```

```
nw_proto: 1
icmp_type: 8
icmp_code: 0
```

5. You can stop your virtual switch by pressing `ctrl-c` in the terminal where it's running. You can restart the simple router without restarting POX and mininet, but it's often useful to restart POX and mininet to ensure the emulated network starts in a clean state.

Note:
In order to run mininet, POX and the router/switch simultaneously, use the **screen** command. The key bindings for switching between screens and other actions can be found in the man page linked above.

Code Overview

The virtual network code consists of the following important packages and classes:

- `cs.sdn.vnet` — **no need** to modify code in this package

  - The main method (`Main`)
  - Classes representing a network device and interfaces (`Device`, `Iface`)
  - Code for creating a PCAP file containing all packets sent/received by a network device (`DumpFile`)

- `cs.sdn.vnet.rt` — **add/modify** code in this package to complete Step 3

  - Skeleton code for a virtual router (Router)
  - A complete implementation of an ARP cache (`ArpCache`, `ArpEntry`)
  - A partial implementation of a route table (`RouteTable`, `RouteEntry`)

- `cs.sdn.vnet.sw` — **add/modify** code in this package to complete Step 2

  - Skeleton code for a virtual switch (`Switch`)

- `net.floodlightcontroller.packet` — **no need** to modify code in this package

  - Code for parsing and manipulating packet headers

There are also several supporting packages and classes, which you **do not need to modify or understand**:

- `cs.sdn.vnet.vns` — code to communicate with POX
- `org.openflow.util` — code for manipulating special types

When your virtual switch or router receives a packet, the `handlePacket(...)` function in the `Switch` or `Router` class is called. When you want to send a packet, call the `sendPacket(...)` function in the `Device` class (which is a superclass of the `Switch` and `Router` classes).

Step 2: Implement Virtual Switch

For this step of the assignment, you will implement a learning switch which forwards packets at the link layer based on destination MAC addresses.

Forwarding Packets

You should complete the `handlePacket(...)` method in the `cs.sdn.vnet.sw.Switch` class to send a received packet out the appropriate interface(s) of the switch. You can use the `getSourceMAC()` and `getDestinationMAC()` methods in the `net.floodlightcontroller.packet.Ethernet` class to determine the source and destination MAC addresses of the received packet. You should call the `sendPacket(...)` function inherited from the `cs.sdn.vnet.Device` class to send a packet out a specific interface. To broadcast/flood a packet, you can call this method multiple times with a different interface specified each time. The `interfaces` variable inherited from the `Device` class contains all interfaces on the switch. The interfaces on a switch only have names; they do not have MAC addresses, IP addresses, or subnet masks. You will need to add structures and/or classes to track the MAC addresses, and associated interfaces, learned by your switch. You should timeout learned MAC addresses after 15 seconds. The timeout does not need to be exact; a granularity of 1 second is fine. The timeout for a MAC address should be reset whenever the switch receives a new packet originating from that address.

Testing (You must show this to TA face-to-face on time!)

You can test your learning switch by following the directions from Step 1.
You can also create your own topologies based on these examples, but do not create topologies with loops. Your virtual switch cannot handle loops in the topology because it does not implement spanning tree.

Step 3: Implement Virtual Router

For this Step of the assignment, you will implement a router which forwards packets at the network layer based on destination IP addresses.

For simplicity, your router will use a statically provided route table and a statically provided ARP cache. Furthermore, when your router encounters an error (e.g., no matching route entry), it will silently drop a packet, rather than sending an ICMP packet with the appropriate error message.

Route Lookups

Your first task is to complete the `lookup(...)` function in the `cs.sdn.vnet.rt.RouteTable` class. Given an IP address, this function should return the `RouteEntry` object that has the *longest prefix match* with the given IP address. If no entry matches, then the function should return null.

Checking Packets

Your second task is to complete the `handlePacket(...)` method in the `cs.sdn.vnet.rt.Router` class to update and send a received packet out the appropriate interface of the router.

When an Ethernet frame is received, you should first check if it contains an IPv4 packet. You can use the `getEtherType()` method in the `net.floodlightcontroller.packet.Ethernet` class to determine the type of packet contained in the payload of the Ethernet frame. If the packet is not IPv4, you do not need to do any further processing—i.e., your router should drop the packet.

If the frame contains an IPv4 packet, then you should verify the checksum and TTL of the IPv4 packet. You use the `getPayload()` method of the `Ethernet` class to get the IPv4 header; you will need to cast the result to `net.floodlightcontroller.packet.IPv4`.

The IP checksum should only be computed over the IP header. The length of the IP header can be determined from the header length field in the IP header, which specifies the length of the IP header in 4-byte words (i.e., multiple the header length field by 4 to get the length of the IP header in bytes). The checksum field in the IP header should be zeroed before calculating the IP checksum. You can borrow code from the `serialize()` method in the `IPv4` class to compute the checksum. If the checksum is incorrect, then you do not need to do any further processing—i.e., your router should drop the packet.

After verifying the checksum, you should decrement the IPv4 packet's TTL by 1. If the resulting TTL is 0, then you do not need to do any further processing—i.e., your router should drop the packet.

Now, you should determine whether the packet is destined for one of the router's interfaces. The `interfaces` variable inherited from the `Device` class contains all interfaces on the router. Each interface has a name, MAC address, IP address, and subnet mask. If the packet's destination IP address exactly matches one of the interface's IP addresses (not necessarily the incoming interface), then you do not need to do any further processing—i.e., your router should drop the packet.

Forwarding Packets

IPv4 packets with a correct checksum, TTL > 1 (pre decrement), and a destination other than one of the router's interfaces should be forwarded. You should use the `lookup(...)` method in the `RouteTable` class, which you implemented earlier, to obtain the `RouteEntry` that has the longest prefix match with the destination IP address. If no entry matches, then you do not need to do any further processing—i.e., your router should drop the packet.

If an entry matches, then you should determine the next-hop IP address and lookup the MAC address corresponding to that IP address. You should call the `lookup(...)` method in the `cs.sdn.vnet.rt.ArpCache` class to obtain the MAC address from the statically populated ARP cache. This address should be the new destination MAC address for the Ethernet frame. The MAC address of the outgoing interface should be the new source MAC address for the Ethernet frame. After you have correctly updated the Ethernet header, you should call the `sendPacket(...)` function inherited from the `cs.sdn.vnet.Device` class to send the frame out the correct interface.

Testing (You must show this to TA face-to-face on time!)

You can test your learning switch by following the directions from Step 1. However, when starting your virtual router, you must include the appropriate static route table and static ARP cache as arguments. For example:

```
          java -jar VirtualNetwork.jar -v r1 -r rtable.r1 -a
          arp_cache
```
You can use any of the following topologies (in the `~/progAssign2/topos` directory) to test your router:
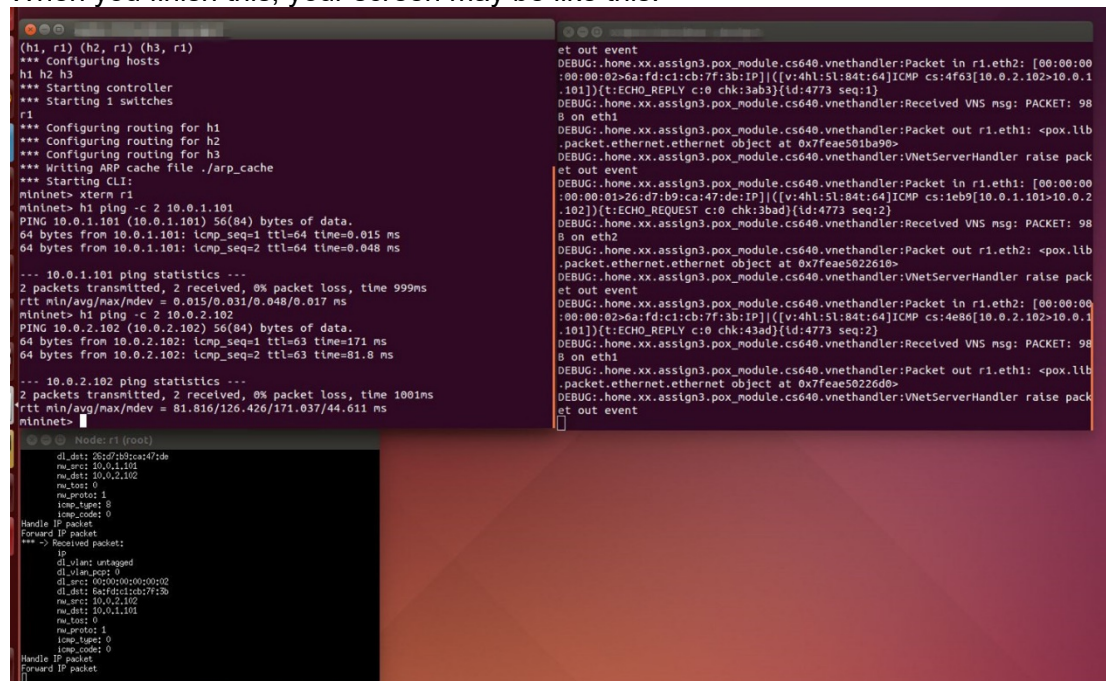
- `single_rt.topo`
- `pair_rt.topo`
- `triangle_rt.topo`

To test your switch and router implementations together, use any of the following topologies:

- `single_each.topo`
- `triangle_with_sw.topo`

You can also create your own topologies based on these examples.
When you finish this, your screen may be like this:

Submission Instructions

You must submit a single tar file of the `src` directory containing the Java source files for your virtual switch and router. Please submit the entire `src` directory; do not submit any other files or directories. Also, you should submit a readme file.
You must submit a single tar file named `assign2_username` containing the above. Please submit only one tar file per group.

Grading
1) Correctness 80%.

       Virtual Switch 40

       Virtual Router 40

2) Code quality 20% (Dont copy from others!)

      Make code more readable.

Acknowledgements