Lab 2: Link & Network Layer Forwarding
CS640 Fall 2022

Released: Tuesday Sep 27 2022
Due: Tuesday Oct 11 2022 11:59PM
Points: 60

Important notes

- All answers and code that you write for this lab must be your own work. Plagiarism is a serious violation of university statutes and all answers will be thoroughly checked for it.
- Ensure that your answer has proper headings and subheadings wherever necessary.
- Late policy:
    - Upto 30 minutes late — lose 0% of points
    - Upto 24 hours late — lose 10% of points
    - Upto 48 hours late — lose 30% of points
    - Upto 72 hours late — lose 60% of points
    - Beyond 72 hours — lose 100% of points

Overview

For this lab, 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.

Learning Outcomes

After completing this lab, 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

==========

Part 1: Getting Started

You will be using Mininet, POX (a platform for Software-Defined Networking control applications), and skeleton code for a simple router to complete the lab. Mininet and POX are

already installed in the virtual machine (VM) you used for lab 1. You should continue to use this VM for this project. You can always refer back to Part 2 (Mininet tutorial) of lab 1 if you have questions about using your VM.

Preparing Your Environment

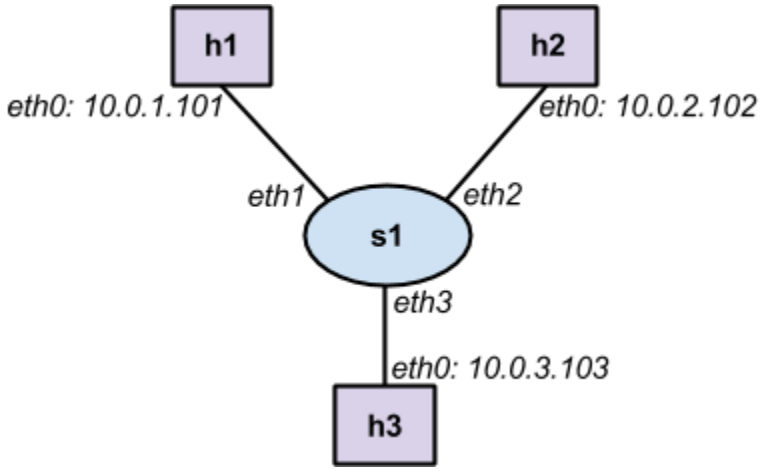
Before beginning this project, there are some additional steps you need to complete to prepare your VM. (We will use python2 for running some softwares and Java >= 8 for completing the assignment):

1. Run the Mininet VM with VirtualBox.
2. Install required packages
    - `sudo apt-get update`
    - `sudo apt-get install -y python-dev python-setuptools flex bison ant openjdk-8-jdk git screen`
3. Install `ltprotocol`
    - `cd ~`
    - `git clone https://github.com/dound/ltprotocol.git`
    - `cd ltprotocol`
    - `sudo pip install twisted`
    - `sudo python setup.py install`

    [NOTE: If you encounter an error with setuptools_scm.version.SetuptoolsOutdatedWarning: your setuptools is too old, please update setuptools
    sudo pip install --upgrade setuptools pip
    Install pip if necessary to do this]
4. Checkout the appropriate version of POX
    - `cd ~/pox`
    - `git checkout f95dd1`
5. Download the starter code from: https://pages.cs.wisc.edu/~mgliu/CS640/F22/labs/lab2/lab2.tgz
    - `cd ~`
    - `wget https://pages.cs.wisc.edu/~mgliu/CS640/F22/labs/lab2/lab2.tgz`
    - `tar xzvf lab2.tgz`
6. Symlink POX and configure the POX modules
    - `cd ~/lab2`
    - `ln -s ~/pox`
    - `sudo chmod +x config.sh && ./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 `lab2/topos/single_sw.topo`.

Running the Virtual Switch

1. Open a terminal. Start Mininet emulation by running the following commands:
   - `$ cd ~/lab2/`
   - `$ sudo python run_mininet.py topos/single_sw.topo -a`

   Exit the mininet emulation. We will restart this later in step 3. This is required to create some initial files that are required in the next step.

2. Open a terminal. Start the controller, by running the following commands:
   - `cd ~/lab2/`
   - `sudo chmod +x run_pox.sh && ./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
```

Keep POX running (Don't press `ctrl-z,ctrl-d or exit()`).

Note that POX is used “under the hood” in this lab 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.

You must wait for Mininet to connect to the POX controller (**Run Step 3 now**). 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>
```

3. Open another terminal. Start Mininet emulation by running the following commands:
   - `$ cd ~/lab2/`
   - `$ sudo python ./run_mininet.py topos/single_sw.topo -a`

   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 or exit`).

4. Open a third terminal. Build and start the virtual switch, by running the following commands:
   - `cd ~/lab2/`
   - `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 -->
   ```

5. 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
```

6. 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, you can 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:

- `edu.wisc.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`)
- `edu.wisc.cs.sdn.vnet.rt` — **add/modify** code in this package to complete Part 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`)

- `edu.wisc.cs.sdn.vnet.sw` — **add/modify** code in this package to complete Part 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:**

- `edu.wisc.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).

==========

Part 2: Implement Virtual Switch

For this part of the lab, you will implement a learning switch which forwards packets at the link layer based on destination MAC addresses. If you're not sure how learning switches work, you should read Section 3.1.4 of the textbook or review your notes from class.

Forwarding Packets

You should complete the `handlePacket(...)` method in the `edu.wisc.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 `edu.wisc.cs.sdn.vnet.Device` class to send a packet out to 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 can test your learning switch by following the directions from Part 1. You can use any of the following topologies (in the `~/lab2/topos` directory):

- `single_sw.topo`
- `linear5_sw.topo`
- `inclass_sw.topo`

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 a spanning tree.

==========

Part 3: Implement Virtual Router

For this part of the lab, you will implement a router which forwards packets at the network layer based on destination IP addresses. If you're not sure how IP packet forwarding works, you should read Section 3.2 of the textbook or review your notes from class.

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 `edu.wisc.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 `edu.wisc.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., multiply 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 `edu.wisc.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 `edu.wisc.cs.sdn.vnet.Device` class to send the frame out to the correct interface.

Testing

You can test your learning switch by following the directions from Part 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 `~/lab2/topos` directory) to test your router:

- `single_rt.topo`
- `pair_rt.topo`
- `triangle_rt.topo`
- `linear5_rt.topo`

Note:
Remember you must start the virtual router for each router in a new terminal. For eg, when testing `pair_rt.topo` -

- `java -jar VirtualNetwork.jar -v r1 -r rtable.r1 -a arp_cache`
- `java -jar VirtualNetwork.jar -v r2 -r rtable.r2 -a arp_cache`

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

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

Again, remember to run `VirtualNetwork.jar` separately for each router and each switch in the topology.

[OPTIONAL]

- In order run `VirtualNetwork.jar` in the same terminal for all routers and switches use the following cmd:
- Example:
    - For 2 routers-
        - `for i in {1..2}; do java -jar VirtualNetwork.jar -v r$i -r rtable.r$i -a arp_cache > junk_r$i.out & done`
    - For 2 switches -
        - `for i in {1..2}; do java -jar VirtualNetwork.jar -v s$i > junk_s$i.out & done`
    - Each of these cmds will create 2 processes and return 2 `pids`. Note them down. Use the below cmd to kill each of these processes once you are finished.
        - `kill -9 <pid>`

You can also create your own topologies based on these examples.

==========

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. To create the tar file, run the following command, replacing username1 and username2 with the CS username of each group member (or just username1 if you are working by yourself):

- `tar czvf username1_username2.tgz src`

Upload the tar file on the Lab2 tab on course's Canvas page. Please submit only one tar file per group.