COSC 465: Computer Networking
Spring 2014
Project 4: IP router 2: IP forwarding and ARP requests
Due: 6 March 2014, 11:59:59 pm

Overview

This project is the second in a series of projects that have the ultimate goal of creating the "brains" for a fully functional Internet IPv4 router. The basic functions of an Internet router are to:

1. Respond to ARP (address resolution protocol) requests for addresses that are assigned to interfaces on the router. (Remember that the purpose of ARP is to obtain the Ethernet MAC address associated with an IP address so that an Ethernet frame can be sent to another host over the link layer.)
2. Receive and forward packets that arrive on links and are destined to other hosts. Part of the forwarding process is to perform address lookups ("longest prefix match" lookups) in the forwarding table. We will just use "static" routing in our router rather than implement a dynamic routing protocol like RIP or OSPF.
3. Make ARP requests for IP addresses that have no known Ethernet MAC address. A router will often have to send packets to other hosts, and needs Ethernet MAC addresses to do so.
4. Respond to ICMP messages like echo requests ("pings").
5. Generate ICMP error messages when necessary, such as when an IP packet's TTL (time to live) value has been decremented to zero.

The goal of the second router project is to accomplish items 2 and 3 above.

What you need to do

Setup

The github repository for the IP router projects (https://github.com/jsommers/cosc465_iprouter) contains a new test case file for this stage of the router and a new `setup2.sh` to include those tests. If you cloned a repo directly from mine, you can simply run "git pull" to retrieve these updated and new files. If you forked my repo and cloned from there, you will need to pull from "upstream" to retrieve the changes (or simply clone my repo, copy the new files over, and go from there). Run setup2.sh to build shell scripts for running tests against your P4 code and to help run your router in Mininet. The files built are `runtests2.sh` and `runreal2.sh`.

Create a copy of your myrouter1.py file as myrouter2.py. The test script assumes your program is named myrouter2.py, so just to simplify things, start with a new file name and add it to git (which you should be using anyway!).

IP Forwarding Table Lookup

One of the key tasks to accomplish for this project is to do the fundamental thing that routers do: receive packets, match their destination addresses against a forwarding table, and forward them out the correct interface.

You will need to implement some kind of forwarding table, with each entry containing the following:

1. A network prefix (e.g., 149.43.0.0),
2. A network "mask" (e.g., 255.255.0.0),
3. The "next hop" IP address, if the destination network prefix is *not* for a directly attached network, and
4. The network interface name through which packets destined to the given network should be forwarded.

You will need to build the forwarding table from two source: the list of router interfaces you get through a call to the `net.interfaces()` method, and by reading in the contents of a file named `forwarding_table.txt`.

Note that for each interface object in the list obtained from `net.interfaces()`, the IP address assigned to the interface *and* the network mask are available (see the SRPY documentation for an example.)

The file `forwarding_table.txt` can be assumed to exist in the same directory where your router is starting up (this file is produced by the SRPY test scenario or by the Mininet startup script), and is structured such that each line contains 4 space-separated items: the network address, the subnet mask, the next hop address, and the interface through which packets should be forwarded. Here are two example lines:

```
172.16.0.0 255.255.0.0 192.168.1.2 router-eth0
172.16.128.0 255.255.192.0 10.10.0.254 router-eth1
```

In the first line, the network address is `172.16.0.0` and the subnet mask is `255.255.0.0` (16 1's followed by 16 0's). The next hop IP address is `192.168.1.2`, and the interface through which to forward packets is named `router-eth0`.

After you build the forwarding table (which should be done *once*, upon startup), destination addresses in IP packets received by the router should be matched against the forwarding table. Remember that in case of two items in the table matching, the **longest** prefix match should be used. Refer to the textbook and/or class notes if you don't recall how to do the longest prefix match.

Two special cases to consider:

1. If there is no match in the table, just drop the packet. (We'll fix this in a later project.)
2. If packet is for the router itself (i.e., destination address is an address of one of the router's interfaces), also drop/ignore the packet.

There are a couple functions and methods in the POX packet library that are helpful for building forwarding table entries and/or for matching destination IP addresses against forwarding table entries:

1. In the `pox.lib.addresses` module there is a `netmask_to_cidr` function that accepts a netmask as a string or IPAddr object, and returns the length of the netmask (i.e., the number of leading 1's in the network address). For example:

   ```
   from pox.lib.addresses import netmask_to_cidr, IPAddr
   netmask1 = IPAddr("255.255.0.0")
   netmask2 = IPAddr("255.255.252.0")
   print netmask_to_cidr(netmask1) # -> 16
   print netmask_to_cidr(netmask2) # -> 22
   ```

2. The IPAddr class has a `toUnsigned()` method that returns the 32-bit unsigned integer representation of an IP address. Remember that you can use bit-wise operations on Python integers (& = bitwise AND, | = bitwise OR, ~ = bitwise NOT, ^ = bitwise XOR).

Forwarding the Packet and ARP

Once you do the forwarding table lookup for an IP destination address, the next steps are to:

1. Decrement the TTL field in the IP header by 1 (you could do this prior to forwarding table lookup, too). You can assume for this project that the TTL value is greater than 0 after decrementing.
2. Create a new Ethernet header for the IP packet to be forwarded. To construct the Ethernet header, you need to know the destination Ethernet MAC address corresponding to the host to which the packet should be forwarded. The next hop host is either:
   a. the destination host, if the destination address is directly reachable through one of the router interfaces (i.e., the subnet that the destination address belongs to is directly connected to a router interface), or
   b. it is an IP address on a router through which the destination is reachable.

In either case, you will need to send an ARP query in order to obtain the Ethernet address corresponding to the next hop IP address. For handling ARP queries you should do the following:

- Send an ARP request for the IP address needing to be "resolved" (i.e., the IP address for which you need the corresponding Ethernet address).
- When an ARP reply is received, complete the Ethernet header for the IP packet to be forwarded, and send it along. You should also create a cache of IP addresses and the Ethernet MAC addresses that they correspond to. When you receive a response to an ARP query, add the IP address->Ethernet address mapping to the cache so that you can avoid doing an identical ARP query.

- If no ARP reply is received within 1 second in response to an ARP request, send another ARP request. Send up to (exactly) 5 ARP requests for a given IP address. If no ARP reply is received after 5 requests, give up and drop the packet (and do nothing else).

You will need to carefully structure your code to be able to receive and process incoming packets while you are waiting for replies to ARP requests. A suggested method is to create a queue that contains information about IP packets awaiting ARP resolution. Each time through the main `while` loop in your code, you can process the items in the queue to see whether an ARP request retransmission needs to be sent. If you receive an ARP reply packet, you could remove an item from the queue, update the ARP table, construct the Ethernet header, and send the packet. You might create a separate class to represent packets in the queue waiting for ARP responses, with the class containing variables to hold the most recent time an ARP request was sent, and the number of retries, among other things.

For keeping track of how long it has been since an ARP request has been sent, you can use the built-in time module. It has a time function that returns the current time in seconds (as a floating point value) (e.g., `time.time() # -> current time in seconds as a float`).

Lastly, refer to the [POX packet library documentation] for details and examples for how to parsing and constructing Ethernet, ARP, and IP packet headers.

When you're done, you should just submit your `myrouter2.py` file to Moodle.

Testing your code

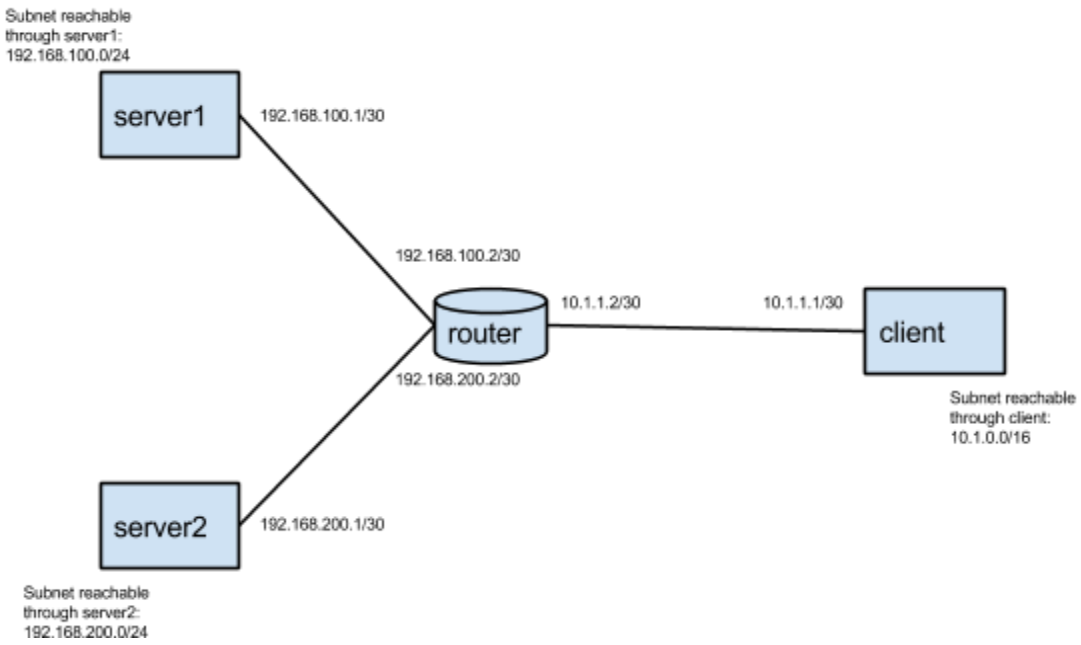
SRPY testing

For initial testing and debugging of your code, you can run the *runtests2.sh* script that is built as a side-effect of running *setup.sh*. The `runtests2.sh` script uses the test scenario in `routertests2.srpy`, which contains 31 separate test cases (yes, that many).

If you need to step through code to see what's going on, you can add calls to debugger() at any point in your code. When execution reaches that line, you'll get a Python debugger (pdb) command line at which you can inspect variables, call methods, etc., in order to understand what's happening. This kind of debugging will, in general, be much more effective than "printf" debugging.

Mininet ("live") testing

Once the SRPY tests pass, you should test your router in Mininet. There is a `start_mininet.py` script in the project git repo for building the following network topology:



(Note that the above topology is *not* the same as the one implied by the SRPY tests.)

To test your router in Mininet, open up a terminal on the virtual machine, and cd (if necessary) to the folder where your project files are located (or transfer them into the virtual machine). Then type:

```
$ sudo python start_mininet.py
```

Once Mininet starts up, you should open an xterm on the router node, and type `"./runreal2.sh"` to start your router.

At that point, you should be able to open another xterm on any one of the other nodes and send a ping (ICMP echo request) to *any* of the IP addresses configured on any node in the network. For example, if you open an xterm on client, you should be able to send a ping to 192.168.200.1 (on server2) and 192.168.100.1 (on server1). You should also be able to send a ping to any address in the subnets 192.168.100.0/24 and 192.168.200.0/24 from the client node, and the router should successfully forward them to either server1 or server. To test whether the router is correctly forwarding the packets, you can run wireshark on either (or both) of server1 and server2.