COSC 465: Computer Networking
Spring 2014
Project 3: IP router 1: ARP responder
Due: 20 February 2014, 11:59:59pm

Overview

This project is the first 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. 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.
3. 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 information base. We will eventually just use "static" routing in our router, rather than implement a dynamic routing protocol like RIP or OSPF.
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 this first router project is to accomplish item #1 above: respond to ARP requests.

What you need to do

In the git repository for this project (https://github.com/jsommers/cosc465_iprouter) there is a Python file to use as a starter template: `myrouter1.py`. This file contains the outline of a `Router` class, and currently contains a constructor (`__init__`) method and a `router_main` method.

After you clone the git repository for this project (or fork and clone, your choice), you will need to run `setup.sh` in the cloned repo:

```
$ ./setup.sh
```

This script will clone a couple additional repositories (one for the POX packet library, and the other for the SRPY module) and create a couple scripts for testing your code.

The main task for this project is to modify the `Router` class to do the following:

1. Upon receiving a packet, determine whether it is an ARP request.
    - For ARP packet headers, there are two *types* of addresses, as well as *source* and *destination* values for each address type, giving a total of four addresses.

For an ARP request, the source Ethernet and IP addresses are filled in, as well as the destination IP address. Note that the source and destination IP addresses are called `protosrc` and `protodst` in the ARP header, respective. The source and destination Ethernet addresses are called `hwsrc` and `hwdst`, respectively. The destination Ethernet address is *not* filled in: this is the address being requested.

- For each ARP request, you should determine whether the protodst field (IP address destination) in the ARP header is an IP address assigned to one of the interfaces on the router.
  - Remember that you can get a list of interfaces configured for the router by calling the `interfaces` method on the net object stored in the `self.net` attribute in the `Router` class. The SRPY documentation has details on what is returned by the `interfaces` method. You may wish to call this method in the constructor of the `Router` class and create some internal data structure for the `Router`, so it can keep track of its own interfaces.
- If the destination IP address is one that's assigned to one of the Router's interfaces, you should create and send an appropriate ARP reply. (If the destination IP address is *not* assigned to one of the router's interfaces, you should *not* respond with an ARP reply, even if you have enough information to do so.) The ARP reply should be sent out the same interface on which the ARP request arrived.

2. If a packet that you receive in the router is *not* an ARP request, you should ignore it (drop it) for now. In the next couple projects you'll handle more incoming packet types in your router.

Refer to the POX packet library documentation for details on parsing and constructing Ethernet and ARP packet headers. There are pretty good examples for how to construct Ethernet and ARP headers, and details on what's included in each header.

When you're done, you should just submit your `myrouter1.py` file to Moodle.

One more note

You'll eventually need to store a mapping in the `Router` between destination IP addresses and Ethernet MAC addresses (you can assume there is a one-to-one mapping). The reason is simple: when you send an IP packet to another host, you'll also need the Ethernet address associated with the destination IP address. If you "remember" any source IP/Ethernet MAC pairs from ARP requests that are received at the router, it may help you to avoid having to construct and send an ARP request to obtain the same information. This capability isn't strictly required for this project, but it may be helpful to have in place for future projects.

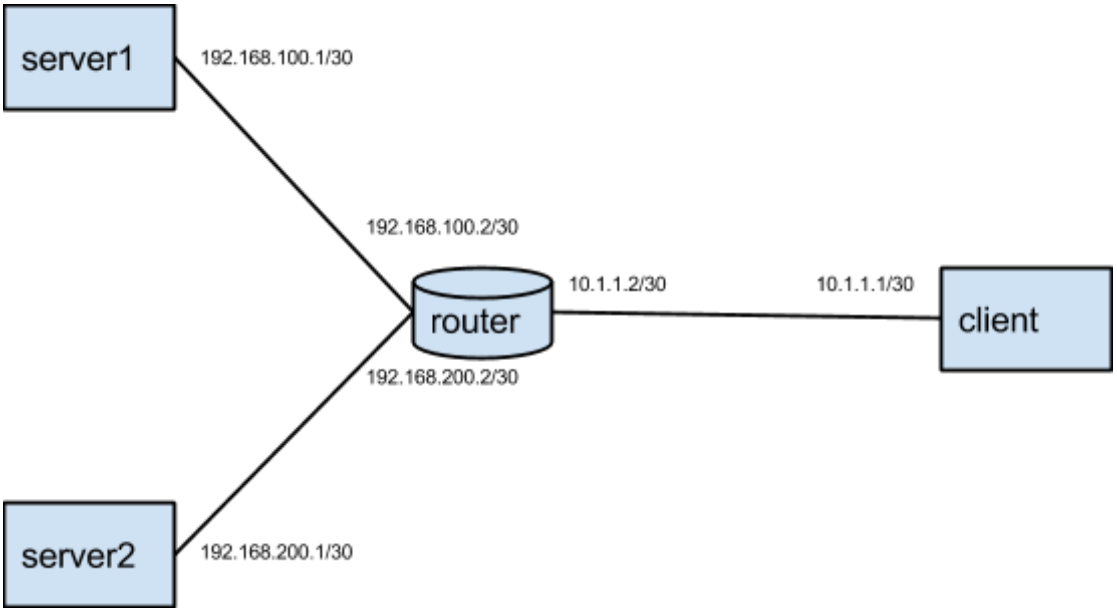
Testing your code

SRPY testing

For initial testing and debugging of your code, you can run the *runtests.sh* script that is built as a side-effect of running *setup.sh*. The `runtests.sh` script uses the test scenario in `routertests1.py`, which includes 6 test cases. You can also run srpy.py directly, but it's *much* easier to just run `./runtests.sh`.

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, you do the following:

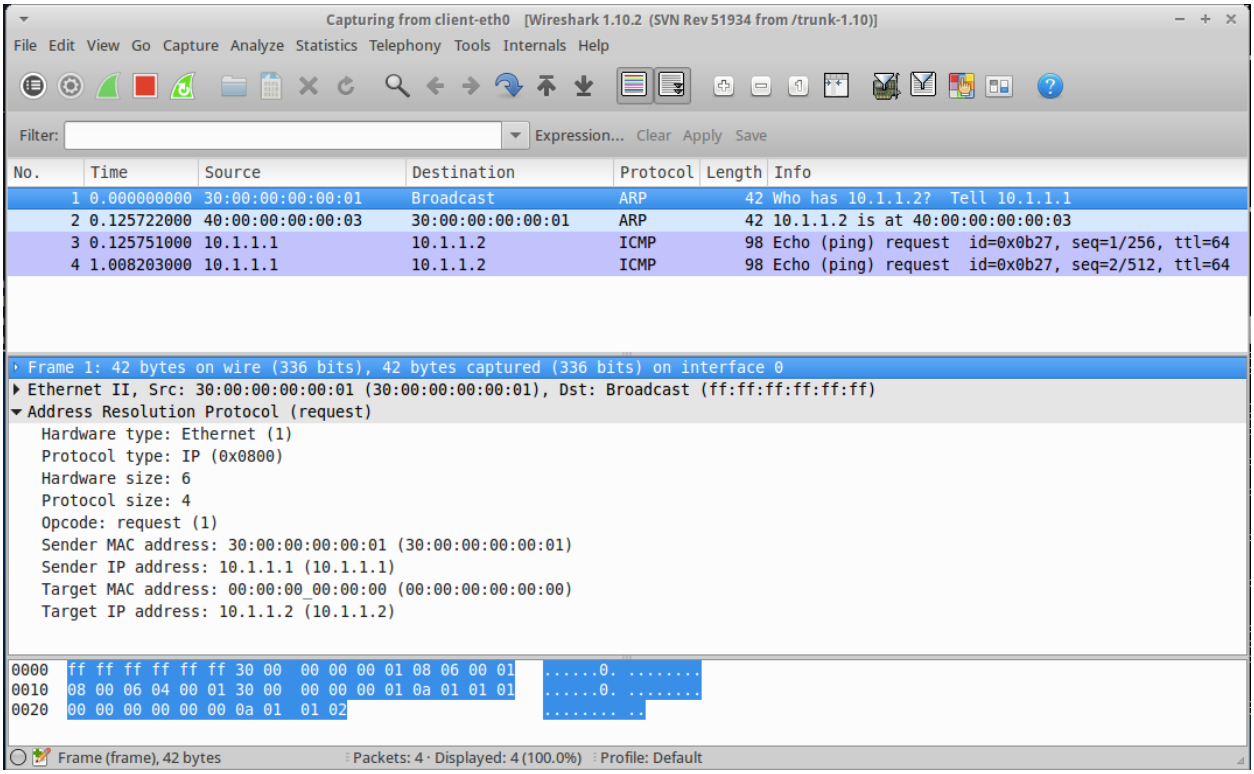
1. Open up a terminal on the virtual machine, and cd (if necessary) to the folder where your project files are located (or transfer them into the virtual machine). Then type the following to get Mininet started:
   ```
   $ sudo python start_mininet.py
   ```
2. Open up an xterm on the client node
   ```
   mininet> xterm client
   ```
3. Start up wireshark on the client. From the xterm running on the client, type:
   ```
   client# wireshark -k&
   ```
4. Open an xterm on the router node
   ```
   mininet> xterm router
   ```
5. Start your router:
   ```
   router# ./runreal.sh
   ```

6. Now, in the xterm running on the client, try to send an ICMP echo request to the IP address at the "other end" of the link between the client and the router.

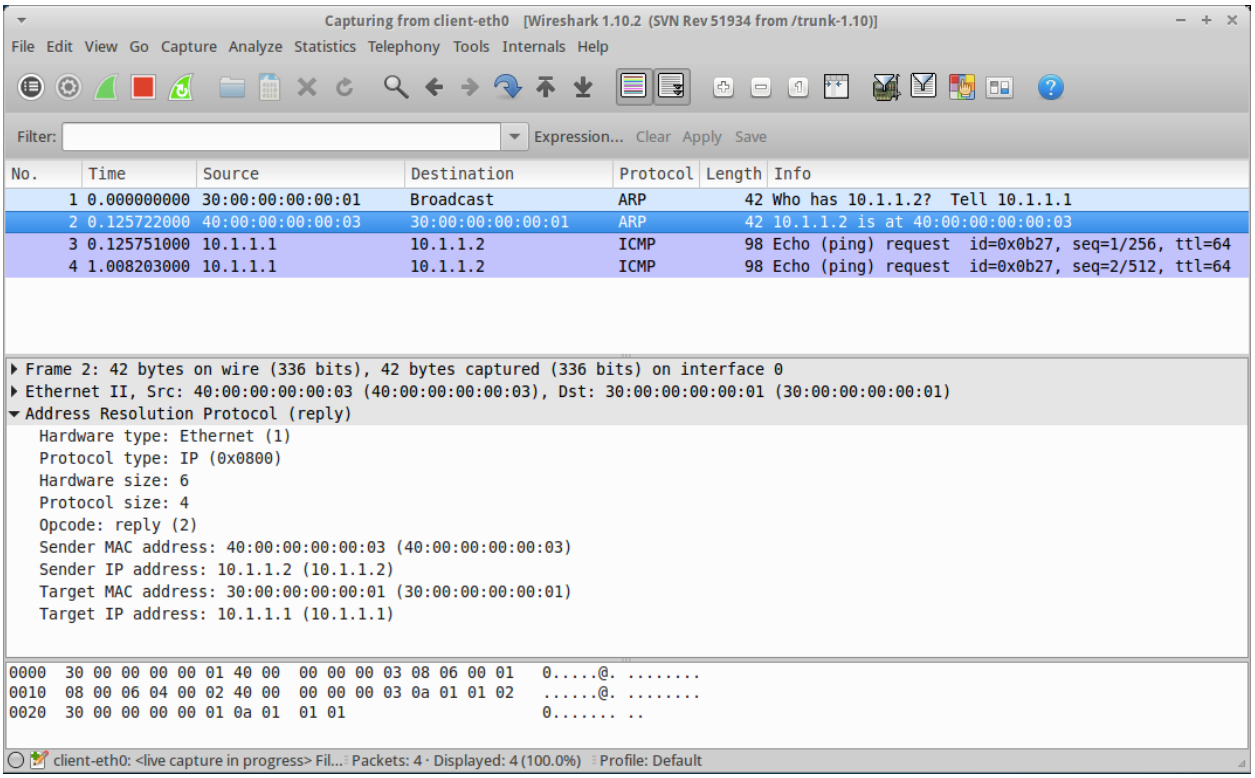
```
client# ping -c3 10.1.1.2
```

   The router *should* initially receive an ARP request for its own IP address (which your router will need to correctly respond to!), then it should receive an ICMP echo request. Since your router isn't yet programmed to respond to ping requests, nothing else should happen (i.e., you'll get ping requests, but they won't be responded to).

In wireshark, you should see the following details when you click on the ARP request packet the first line in the capture window). Notice that the "target MAC address" is currently all zeroes, since this is the address being requested.



Also in wireshark, you should see the following details when you click on the ARP response packet (second line in the capture window). Notice that all the addresses in the ARP header are now filled in (and that source and destination addresses are effectively swapped):


Capturing from client-eth0 [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: Expression... Clear Apply Save
No. Time Source Destination Protocol Length Info
1 0.000000000 30:00:00:00:00:01 Broadcast ARP 42 Who has 10.1.1.2? Tell 10.1.1.1
2 0.125722000 40:00:00:00:00:03 30:00:00:00:00:01 ARP 42 10.1.1.2 is at 40:00:00:00:00:03
3 0.125751000 10.1.1.1 10.1.1.2 ICMP 98 Echo (ping) request id=0x0b27, seq=1/256, ttl=64
4 1.008203000 10.1.1.1 10.1.1.2 ICMP 98 Echo (ping) request id=0x0b27, seq=2/512, ttl=64
Frame 2: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: 40:00:00:00:00:03 (40:00:00:00:00:03), Dst: 30:00:00:00:00:01 (30:00:00:00:00:01)
Address Resolution Protocol (reply)
Hardware type: Ethernet (1)
Protocol type: IP (0x0800)
Hardware size: 6
Protocol size: 4
Opcode: reply (2)
Sender MAC address: 40:00:00:00:00:03 (40:00:00:00:00:03)
Sender IP address: 10.1.1.2 (10.1.1.2)
Target MAC address: 30:00:00:00:00:01 (30:00:00:00:00:01)
Target IP address: 10.1.1.1 (10.1.1.1)
0000 30 00 00 00 00 01 40 00 00 00 00 03 08 06 00 01 0.....@. ....
0010 08 00 06 04 00 02 40 00 00 00 00 03 0a 01 01 02 ......@. ....
0020 30 00 00 00 00 01 0a 01 01 01 0....... ..
client-eth0: <live capture in progress> Fil... Packets: 4 · Displayed: 4 (100.0%) Profile: Default