

# Project 2a: Building a Simple Router: Basic ARP and ICMP

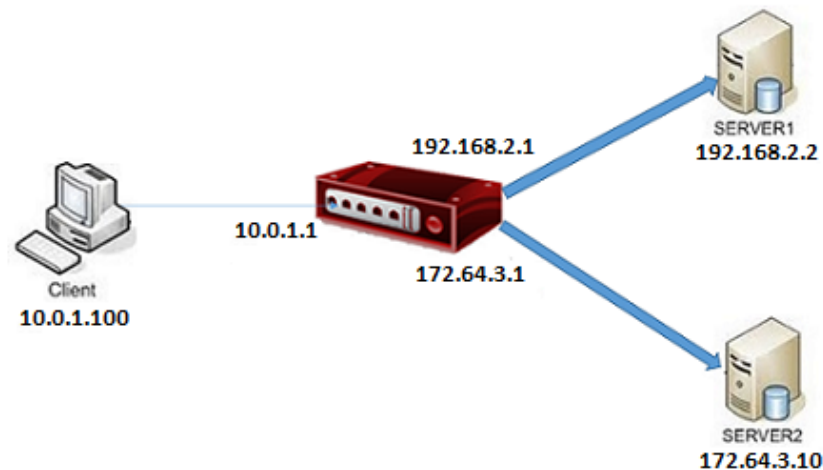
Assigned: 2022-02-07

Due: 2022-02-21, 10:00 PM Pacific Time

## Overview

In this assignment you will write a simple router given a static network topology and routing table. Your router will receive raw Ethernet frames. It will process these packets just like a real router and then forward them to the correct outgoing interface. You are responsible for implementing the logic for handling the incoming Ethernet frames (forward them, generate ICMP messages, drop them and more).

Your router will route real packets from an emulated host (client) to two emulated application servers (http server 1/2) sitting behind your router. A sample routing topology is shown below:



If the router is functioning correctly, you should be able to perform the following operations:

- Ping the router's interfaces (192.168.2.1, 172.64.3.1, 10.0.1.1) from the client.
- Ping the servers (192.168.2.2, 172.64.3.10) from the client.

Additional requirements are laid out in the [Required Functionality](#) section.

## Mininet

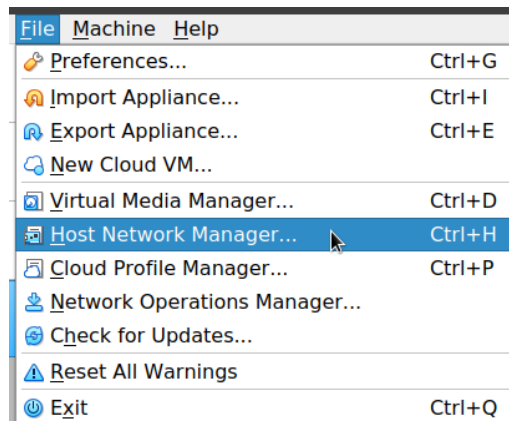
This assignment runs on top of Mininet which was developed at Stanford. Mininet allows you to emulate a topology on a single machine. It provides the needed isolation between the emulated

nodes so that your router node can process and forward real Ethernet frames between the hosts like a real router. You **don't** have to know how Mininet works to complete this assignment, but more information about Mininet (if you're curious) is available [here](#).

## Getting Started

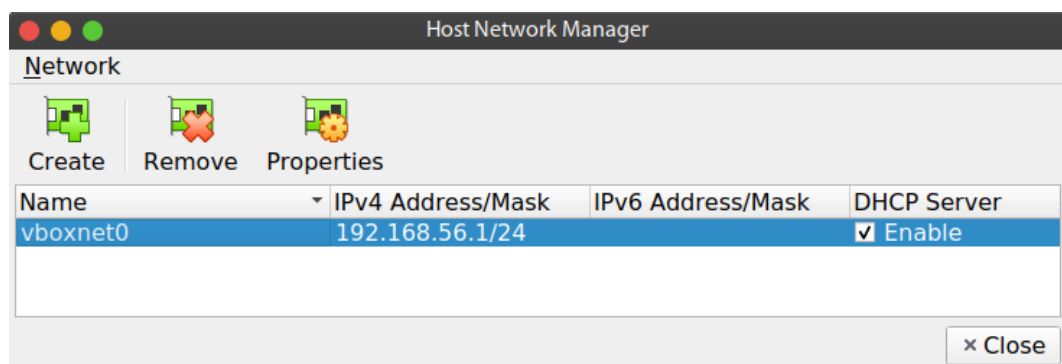
1. Download and install VirtualBox (<https://www.virtualbox.org/>)
  - a. **Windows Users:** Please see **Virtual Machine FAQ** section.
2. Download the virtual machine image zip file **cse-123-p2.ova.zip** from the ieng6 server:  
scp  
<account>@ieng6.ucsd.edu:/home/linux/ieng6/cs123wi22/public/cse-123-p2.ova.zip ./vm-image.zip
3. Configure VirtualBox with a **host-only** network:

- a. Open the host network manager from the File menu.



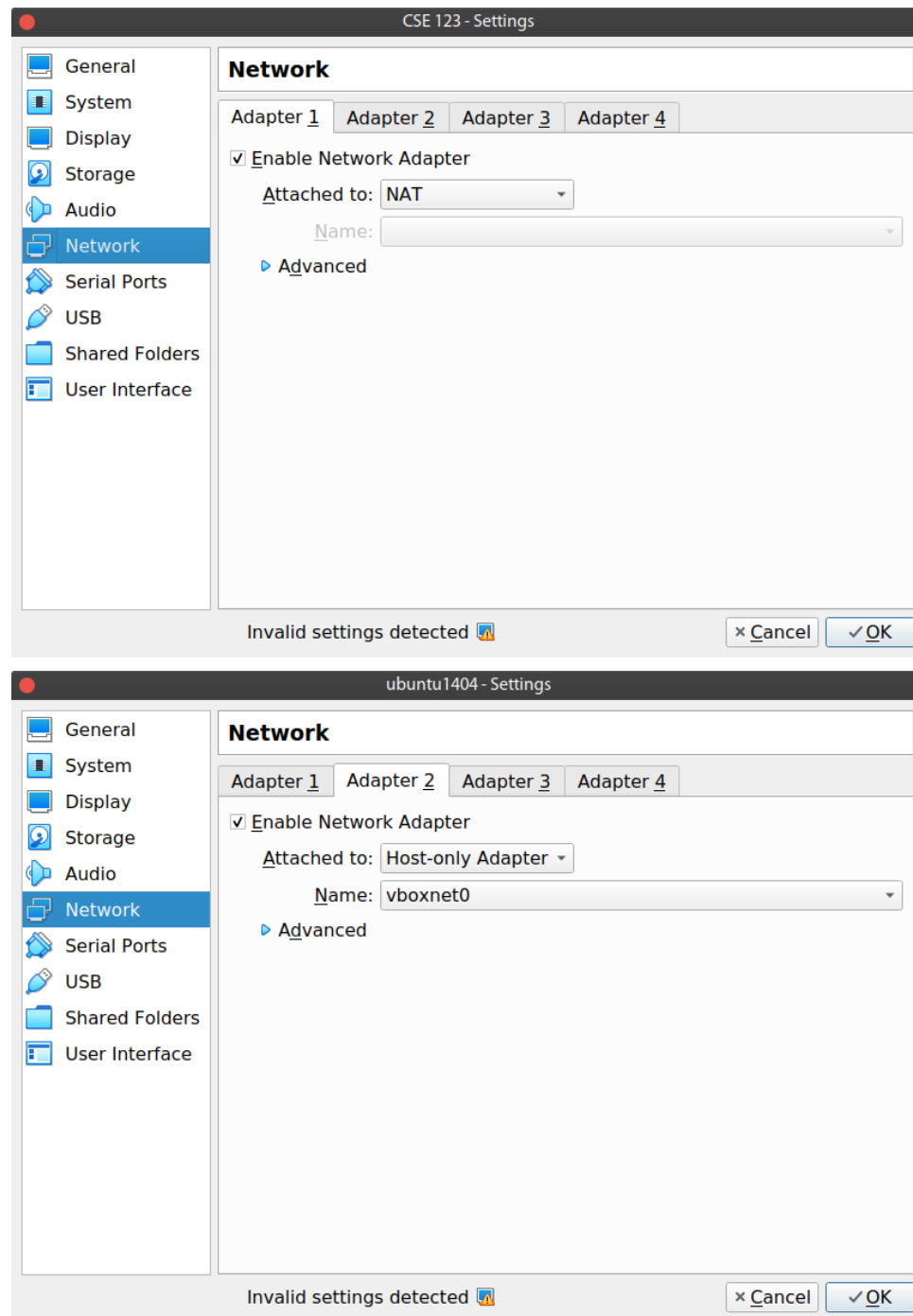
- b. Click on the Create button to create a host-only network.
- c. Check the **Enable** box in the DHCP Server column.

The window should look like this:



4. Unzip the VM image and open the .ova file in VirtualBox.
  - a. Keep the default settings and click on the Import button.
5. Check the settings for the newly imported virtual machine. The network section should show **two** enabled adapters.
  - a. Adapter 1 should be attached to **NAT**.

- b. Adapter 2 should be a **host-only** adapter connected to the host-only network you just created.



6. Now you should be able to run the virtual machine and log in with the **username mininet** and **password mininet**.
7. **NOTE:** After you log in to the virtual machine, run the command `sudo dhclient eth1` to get an IP address on the host-only network.

From here, you might find it more convenient to **ssh** into your VM instead of working inside of it directly (you'll be able to copy and paste, among other luxuries). To do this:

1. Run `ifconfig`.
2. Note down the IP address of the `eth1` interface.
3. Then, log out of your VM by typing in `exit`. This should take you back to the login screen. You can now **minimize** the VM window, but **do not close** the application.
4. Then, in a terminal window, type `ssh -X mininet@<IP address you noted>`.

**NOTE:** In case you are facing some issues with `-X`, you don't have X11 forwarding supported. You can either skip the `-X` (although you will not be able to inspect packets on the fly with Wireshark) or you can follow the steps mentioned [here](#) to get X11 forwarding setup.

## Can we access the VM from Visual Studio Code?

Yes! Please refer to the following official documentation from Microsoft:

<https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh>

## Virtual Machine FAQ

### Cannot Start VM On Windows VirtualBox

This may be related to having **Hyper-V** installed and enabled. Hyper-V is useful for many other applications such as Docker, but it may cause you some issues with this assignment. We recommend disabling Hyper-V while you work on the assignment. To do this, try the following:

- In the Control Panel, select Programs and Features.
- Select Turn Windows features on or off.
- Clear the **Hyper-V** check box (if you are on **Windows 10**) or the **Windows Hypervisor Platform** check-box (if you are on **Windows 11**).
- Restart your computer when prompted. You **may have to restart it one more time** after this if it still doesn't work (make sure you select restart and not shutdown).

### VERR\_NEM\_VM\_CREATE\_FAILED On Windows 11

You might encounter this error in Windows 11 if **you had VirtualBox installed and working on Windows 10 and then upgraded your OS to Windows 11**. The VirtualBox application will work fine, but when you start any VM, a window will pop up and then shortly after, it will be destroyed and the above error will be shown. To fix this, from an elevated (Run as administrator) Command Prompt window, execute the command `bcdedit /set hypervisorlaunchtype off` and then reboot your computer.

## Ensure Hardware Virtualization is Enabled on BIOS Settings

VirtualBox virtualization requires that the host machine has hardware virtualization enabled. While most computers have this feature, it might be disabled in the BIOS. You may have to reboot into the **system BIOS** and enable Virtualization Technology (VTx) in the system BIOS settings if it is not enabled on your computer. Please Google how to do this (update BIOS settings) depending on your computer's make and model.

## Configuration Files

To get your starter code please follow the next steps:

1. Find the GitHub Classroom invitation link on Canvas and create your repository.
2. In the virtual machine, create an SSH key and add the public key to your GitHub account by following these instructions:
  - a. <https://help.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>
  - b. <https://help.github.com/en/github/authenticating-to-github/adding-a-new-ssh-key-to-your-github-account>
3. Clone your repository.
  - `cd ~/cse123-p2/`
  - `git clone git@github.com:<link_to_your_repo>.git router`  
Do not miss the "router" at the end.

**NOTE:** If you run into an error along the lines of "**cannot resolve github.co**", try running: `sudo dhclient eth1` and then attempt the clone again.

There are two configuration files:

1. **IP\_CONFIG**: Lists the IP addresses assigned to the emulated hosts. It is used by mininet to emulate the topology shown earlier.
2. **rtable**: The static routing table used for the simple router.

The default **IP\_CONFIG** and **rtable** should look like the following:

- ```
cat ~/cse123-p2/IP_CONFIG
server1 192.168.2.2
server2 172.64.3.10
client   10.0.1.100
sw0-eth1 192.168.2.1
sw0-eth2 172.64.3.1
sw0-eth3 10.0.1.1
```

- `cat ~/cse123-p2/router/rtable`  

|             |             |                 |      |
|-------------|-------------|-----------------|------|
| 192.168.2.2 | 192.168.2.2 | 255.255.255.255 | eth1 |
| 172.64.3.10 | 172.64.3.10 | 255.255.255.255 | eth2 |
| 10.0.1.100  | 10.0.1.100  | 255.255.255.255 | eth3 |

**NOTE:** If nothing appears when you cat these files, you may have missed the “**router**” at the end of the git clone command. These files are symlinks to files that exist in the git repository which should be cloned to ~/cse123-p2/router.

## Test Connectivity of Your Emulated Topology

Mininet requires a controller, which is implemented in **POX**. To run the controller, use the following commands:

- `cd ~/cse123-p2/`
- `./run_pox.sh`

You should be able to see some output like the following:

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:.usr.local.lib.python2.7.dist-packages.cs144-0.0.0-py2.7.egg.cs
144.ofhandler:*** ofhandler: Successfully loaded ip settings for
hosts
{'server1': '192.168.2.2', 'sw0-eth3': '10.0.1.1', 'sw0-eth1':
'192.168.2.1', 'sw0-eth2': '172.64.3.1', 'client': '10.0.1.100',
'server2': '172.64.3.10'}

INFO:.usr.local.lib.python2.7.dist-packages.cs144-0.0.0-py2.7.egg.cs1
44.srhandler:created server
DEBUG:.usr.local.lib.python2.7.dist-packages.cs144-0.0.0-py2.7.egg.cs
144.srhandler:SRServerListener listening on 8888
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.4/Apr 19 2013 18:32:33)
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
INFO:openflow.of_01:[Con 1/139234599119694] Connected to
7e-a2-14-d2-2b-4e
DEBUG:.usr.local.lib.python2.7.dist-packages.cs144-0.0.0-py2.7.egg.cs
144.ofhandler:Connection [Con 1/139234599119694]
```

```

DEBUG:.usr.local.lib.python2.7.dist-packages.cs144-0.0.0-py2.7.egg.cs
144.srhandler:SRServerListener catch RouterInfo even, info={'eth3':
('10.0.1.1', 'c2:e3:78:bf:71:f9', '10Gbps', 3), 'eth2':
('172.64.3.1', '26:e4:1f:26:49:ab', '10Gbps', 2), 'eth1':
('192.168.2.1', '56:01:ec:fb:34:09', '10Gbps', 1)},
rtable=[('10.0.1.100', '10.0.1.100', '255.255.255.255', 'eth3'),
('192.168.2.2', '192.168.2.2', '255.255.255.255', 'eth1'),
('172.64.3.10', '172.64.3.10', '255.255.255.255', 'eth2')]
Ready.
POX>

```

**NOTE:** Please wait for a few seconds till mininet is able to connect to the POX controller before you continue to the next step.

**Keep POX running.** Now, **open a second terminal** to continue to the next step. You may want to use [screen](#) to run multiple shell sessions on the same window. For a cheat sheet of the different keyboard shortcuts available, consult the link or use manpages (Run `man screen` from the terminal). Few Basic Tips: Type `screen` to open a new terminal. Use `Ctrl+A+C` to create a new screen (terminal), `Ctrl+A+W` to list the different open screens (terminals) and `Ctrl+A+<Screen ID>` to switch to the required screen.

**NOTE:** If you are not comfortable with running multiple shell sessions from the same window, simply launch multiple SSH sessions from different terminals (you can use a SSH client like PuTTY or iTerm for this). Just ensure that you do not close these terminals while testing your code. You will require three such simultaneous sessions (as we shall discuss next):

1. POX
2. Mininet
3. Router

Next, we can start the **mininet emulation** by using the following commands:

- `cd ~/cse123-p2/`
- `./run_mininet.sh`

You should be able to see some output like the following:

```

*** Shutting down stale SimpleHTTPServers
*** Shutting down stale webservers
*** Successfully loaded ip settings for hosts
{'server1': '192.168.2.2', 'sw0-eth3': '10.0.1.1', 'sw0-eth1':
'192.168.2.1', 'sw0-eth2': '172.64.3.1', 'client': '10.0.1.100',
'server2': '172.64.3.10'}
*** Creating network
*** Creating network
*** Adding controller

```

```

*** Adding hosts:
client server1 server2
*** Adding switches:
sw0
*** Adding links:
(client, sw0) (server1, sw0) (server2, sw0)
*** Configuring hosts
client server1 server2
*** Starting controller
*** Starting 1 switches
sw0
*** setting default gateway of host server1
server1 192.168.2.1
*** setting default gateway of host server2
server2 172.64.3.1
*** setting default gateway of host client
client 10.0.1.1
*** Starting SimpleHTTPServer on host server1
*** Starting SimpleHTTPServer on host server2
*** Starting CLI:
mininet>

```

At the same time, you should be able to see the following output on the POX controller terminal:

```

POX> INFO:openflow.of_01:[Con 1/None] closing connection
INFO:openflow.of_01:[Con 2/196695499902282] Connected to
b2-e4-bd-66-8d-4a
DEBUG:build.bdist.linux-x86_64.egg.cs144.ofhandler:Connection [Con
2/196695499902282]
DEBUG:build.bdist.linux-x86_64.egg.cs144.srhandler:SRServerListener
catch RouterInfo even, info={'eth3': ('10.0.1.1',
'82:44:37:be:5c:ca', '10Gbps', 3), 'eth2': ('172.64.3.1',
'2e:3e:2a:9c:3b:07', '10Gbps', 2), 'eth1': ('192.168.2.1',
'72:25:54:8b:7b:1b', '10Gbps', 1)}, rtable=[('10.0.1.100',
'10.0.1.100', '255.255.255.255', 'eth3'), ('192.168.2.2',
'192.168.2.2', '255.255.255.255', 'eth1'), ('172.64.3.10',
'172.64.3.10', '255.255.255.255', 'eth2')]

```

**Keep this terminal open** as you will need the mininet command line for debugging. Next, **open a third terminal** (this is where you will run your router binary) to continue to the next step.

Now you are ready to test out the connectivity of the environment setup. To do so, run the following commands:

- `cd ~/cse123-p2/router`
- `./sr_solution2a` (`./sr_solution2b` for Project 2b)

*sr\_solution2a* and *sr\_solution2b* (for Project 2b) are the sample solution binaries that have been provided to you so that you can test your environment setup and also for you to check how the sample outputs should look like when you ping the hosts or the router's interfaces. If your setup is successful, you should be able to see some output like the following:

```
Using VNS sr stub code revised 2009-10-14 (rev 0.20)
Loading routing table from server, clear local routing table.
Loading routing table
-----
Destination      Gateway          Mask            Iface
192.168.2.2       192.168.2.2      255.255.255.255 eth1
172.64.3.10       172.64.3.10      255.255.255.255 eth2
10.0.1.100        10.0.1.100       255.255.255.255 eth3
-----
Client mininet connecting to Server localhost:8888
Requesting topology 0
successfully authenticated as mininet
Loading routing table from server, clear local routing table.
Loading routing table
-----
Destination      Gateway          Mask            Iface
192.168.2.2       192.168.2.2      255.255.255.255 eth1
172.64.3.10       172.64.3.10      255.255.255.255 eth2
10.0.1.100        10.0.1.100       255.255.255.255 eth3
-----
Router interfaces:
eth3      HWaddr c2:e0:f7:04:02:0c
          inet addr 10.0.1.1
eth2      HWaddr 9a:25:38:88:a3:24
          inet addr 172.64.3.1
eth1      HWaddr 1a:4c:9c:2b:b2:d9
          inet addr 192.168.2.1
<-- Ready to process packets -->
```

In this particular setup, 10.0.1.1 is the IP for the client, 192.168.2.2 is the IP for server1 and 172.64.3.10 is the IP for server2. You can find these IP addresses in your IP\_CONFIG file.

Now, back to the terminal where mininet is running. **To issue a command on the emulated host, type the host name followed by the command in the mininet console.** For example, the following command issues 3 pings from the client to the server2.

- mininet> client ping -c 3 172.64.3.10  
PING 172.64.3.10 (172.64.3.10) 56(84) bytes of data.  
64 bytes from 172.64.3.10: icmp\_seq=1 ttl=63 time=256 ms  
64 bytes from 172.64.3.10: icmp\_seq=2 ttl=63 time=102 ms

```

64 bytes from 172.64.3.10: icmp_seq=3 ttl=63 time=84.8 ms

--- 172.64.3.10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 84.857/147.942/256.825/77.315 ms

```

When you finish implementing your router code, it should make the above command work correctly, but with the starter code, it should not. To confirm this, next, run the starter code:

- `cd ~/cse123-p2/router`
- `make`
- `./sr`

You should see some output from like the following:

```

Loading routing table from server, clear local routing table.
Loading routing table
-----
Destination      Gateway          Mask      Iface
192.168.2.2       192.168.2.2     255.255.255.255 eth1
172.64.3.10      172.64.3.10    255.255.255.255 eth2
10.0.1.100       10.0.1.100     255.255.255.255 eth3
-----
Client mininet connecting to Server localhost:8888
Requesting topology 0
successfully authenticated as mininet
Loading routing table from server, clear local routing table.
Loading routing table
-----
Destination      Gateway          Mask      Iface
192.168.2.2       192.168.2.2     255.255.255.255 eth1
172.64.3.10      172.64.3.10    255.255.255.255 eth2
10.0.1.100       10.0.1.100     255.255.255.255 eth3
-----
Router interfaces:
eth3    HWaddr c2:e3:78:bf:71:f9
        inet addr 10.0.1.1
eth2    HWaddr 26:e4:1f:26:49:ab
        inet addr 172.64.3.1
eth1    HWaddr 56:01:ec:fb:34:09
        inet addr 192.168.2.1
<-- Ready to process packets -->

```

Moving back to the mininet terminal, since the starter code cannot handle ping requests, you will get unexpected ping output:

- mininet> client ping -c 3 172.64.3.10  
PING 172.64.3.10 (172.64.3.10) 56(84) bytes of data.  
From 10.0.1.100 icmp\_seq=1 Destination Host Unreachable  
From 10.0.1.100 icmp\_seq=2 Destination Host Unreachable  
From 10.0.1.100 icmp\_seq=3 Destination Host Unreachable  
  
--- 172.64.3.10 ping statistics ---  
3 packets transmitted, 0 received, +3 errors, 100% packet loss,  
time 2009ms

## Starter Code

You should now have all the pieces needed to build and run the router. You can build and run the starter code as follows:

- cd ~/cse123-p2/router
- make
- ./sr

## Logging Packets

Method 1:

You can log the packets received and generated by your ./sr program by using the "-l" parameter with your program. The file will be in .pcap format. You can use scp to copy the .pcap file over to your host machine and run Wireshark on it.

```
> ./sr -l logname.pcap
```

Method 2:

Besides the ./sr binary, you can also use mininet to monitor the traffic in and out of the emulated nodes. Mininet provides direct access to each emulated node. Taking server1 as an example, to see the packets in and out of it, go to the mininet CLI:

```
mininet> server1 sudo tcpdump -n -i server1-eth0
```

or you can bring up a terminal inside server1 by running

```
mininet> xterm server1
```

And then inside the newly popped xterm, run

```
> sudo tcpdump -n -i server1-eth0
```

Method 3:

You can even use Wireshark to inspect packets as they fly by. To get Wireshark running, go to the mininet CLI

```
mininet> xterm sw0
```

Now, in your new xterm window, run

```
> sudo wireshark &
```

**You can disregard the errors you'll get.** In Wireshark, click “Capture Options” and select “sw0-eth1”, “sw0-eth2”, and “sw0-eth3”. These correspond to the eth1, eth2, and eth3 interfaces on the router whose configurations were described above. Now click “Start”. Try pinging one of your router's interfaces. You should see some packets show up in this window.

**NOTE:** To get xterm working, you'll need to have an X server installed and running. Mac users should install and run [XQuartz](#). Windows users may need to install [Xming](#). Linux users likely don't need to do anything.

## ARP Handling

### ARP Request

There can be two scenarios:

- **Receive an ARP request:** These will be sent to the router by the hosts. The router must check if the request is for one of its IPs. If it is, an ARP reply must be constructed.
- **Send an ARP request:** This scenario will arise when the router receives an IP packet that must be forwarded and it needs to know the MAC address corresponding to the next-hop IP of the packet.

### ARP Reply

These will be sent by the hosts in response to the ARP request sent by the router. The router must send all the packets waiting on this ARP request towards its destination.

## IP Forwarding/Handling

## Packets to Other Hosts

Given a raw Ethernet frame, if the frame contains an IP packet that is not destined towards one of our router interfaces:

- Sanity-check the packet (meets version, minimum length and has correct checksum).
- Decrement the TTL by 1. If the TTL is 0 after decrementing its value, send an ICMP **Time Exceeded** message back to the source of the packet. Otherwise, recompute the packet checksum over the modified header and proceed to the next steps.
- Find an entry in the routing table that **exactly matches** the destination IP address (**do not perform longest-prefix matching in Project 2a**). Only forward if there is an exact match to the IP address.
  - If an entry exists, send an ARP request for the next-hop IP.
    - If the router gets an ARP reply within 5 attempts (one per second), send the packets waiting on this ARP request towards its destination. **Do not store the response in an ARP cache for Project 2a. Instead, you will have to send an ARP request for every packet.**
    - If an ARP reply is not received, send an ICMP Destination **Host Unreachable** message back to the source of the packet.
- If no matching entry is in the routing table, send an ICMP Destination **Net Unreachable** message back to the source of the packet.

## Packets to the Router

Given a raw Ethernet frame, if the frame contains an IP packet that is destined towards one of our router interfaces:

- Sanity-check the packet (meets version, minimum length and has correct checksum).
- If the packet is an ICMP echo request and its checksum is valid, send an ICMP echo reply to the sending host.
  - **NOTE:** The data field of an ICMP echo request does not have a fixed length. Its length is determined by the total length field of the IP header. The router should copy the complete data field of the echo request to the corresponding echo reply.
- Otherwise, ignore the packet.

## Protocols to Understand

For help with understanding the different fields in the headers of the different protocols, please refer to the following links:

- **ARP:** <http://www.networksorcery.com/enp/protocol/arp.htm>
- **IP:** <http://www.networksorcery.com/enp/protocol/ip.htm>
- **ICMP:** <http://www.networksorcery.com/enp/protocol/icmp.htm>

## Ethernet

You are given a raw Ethernet frame and you have to send raw Ethernet frames. You should understand the concept of source and destination MAC addresses and the idea that when we forward a packet one hop, we change the destination MAC address of the forwarded packet to the MAC address of the next hop's incoming interface.

## Internet Protocol (IP)

Before operating on an IP packet, you should verify its checksum (on the header only) and ensure that it meets the minimum length of an IP packet. Remember that you are only dealing with IPv4 packets in this assignment. If you determine that a datagram should be forwarded, you should correctly decrement the TTL field of the header and recompute the checksum over the changed header before forwarding it to the next hop. Remember that "For purposes of computing the checksum, the value of the checksum field is zero".

## Internet Control Message Protocol (ICMP)

ICMP is a simple protocol that can send control information to a host. In this assignment, your router will use ICMP to send messages back to a sending host. You will need to properly generate the following ICMP messages (including the ICMP header checksum) in response to the sending host under the following conditions:

- **Echo Reply (Type 0):** Sent in response to an echo request (ping) to one of the router's interfaces. (This is only for echo requests to any of the router's IPs. An echo request sent elsewhere should be forwarded to the next hop address as usual.)
- **Echo Request (Type 8):** Received as a ping message to either the interfaces of the router or forwarded to any other devices in the network. (This packet is not generated by our router.)
- **Destination Net Unreachable (Type 3, Code 0):** Sent if there is a non-existent route to the destination IP (No matching entry in the routing table when forwarding an IP packet).
- **Destination Host Unreachable (Type 3, Code 1):** Sent if 5 ARP requests were sent to the next-hop IP without an ARP reply.
- **Time Exceeded (Type 11, Code 0):** Sent if an IP packet is discarded during processing because the TTL field is 0.

The source address of an ICMP message can be the address of any of the router's interfaces as specified in RFC792. As mentioned earlier, the only incoming ICMP messages destined to the router's IPs that you have to explicitly process are the ICMP echo requests. You can drop any other ICMP messages directed to the router.

**NOTE:** You have already been provided with a `sr_icmp_ttl_hdr_t` struct in your starter code. You can use this struct for all the different types of ICMP messages. But if you want to create additional structs for ICMP messages for your convenience, make sure to use the

packed attribute so that the compiler doesn't try to align the fields in the struct to word boundaries.

**NOTE:** You can use the `-t` flag to test your ICMP Time Exceeded test case. This will allow you to set the TTL on your packets.

- `mininet> client ping -t 1 172.64.3.10`

**NOTE:** You can test the ICMP Destination Host Unreachable test case by adding a new entry to the routing table for a host which does not exist in the topology, say something like the following entry, and then ping that IP (172.64.3.100) from the client.

- `172.64.3.100 172.64.3.100 255.255.255.255 eth2`

**NOTE:** You can test the ICMP Destination Net Unreachable test case by pinging an IP from the client which does not exist in the routing table.

## Address Resolution Protocol (ARP)

ARP is needed to determine the next-hop MAC address that corresponds to the next-hop IP address stored in the routing table. Without the ability to generate an ARP request and process ARP replies, your router will not be able to fill out the destination MAC address field of the raw Ethernet frame you are sending over the outgoing interface. Analogously, without the ability to process ARP requests and generate ARP replies, no other router could send your router Ethernet frames. Therefore, your router must generate and process ARP requests and replies. When handling ARP requests, you should only send an ARP reply if the target IP address is one of the router's IP addresses. **For Project 2a, your router should send an ARP request for every packet it forwards. It should not cache the responses.**

**NOTE:** ARP **requests** are sent to the **broadcast MAC address** (ff-ff-ff-ff-ff-ff). ARP **replies** are sent directly to the **requester's MAC address**.

## Code Overview

### Basic Functions

Your router receives a raw Ethernet frame and sends raw Ethernet frames when sending a reply to the sending host or forwarding the frame to the next hop. The basic functions to handle these functions are:

```
void sr_handlepacket(struct sr_instance* sr, uint8_t * packet,
unsigned int len, char* interface)
```

This method, located in `sr_router.c`, is called by the router each time a packet is received. The `packet` argument points to the packet buffer which contains the full packet including the ethernet header. The name of the receiving interface is passed into the method as well.

```
int sr_send_packet(struct sr_instance* sr, uint8_t* buf, unsigned int
len, const char* iface)
```

This method, located in `sr_vns_comm.c`, will send an arbitrary packet of length `len` to the network out of the interface specified by `iface`.

**NOTE:** You **should not free the buffer given to you in `sr_handlepacket`** (this is why the buffer is labeled as being "lent" to you in the comments). You are responsible for doing correct memory management on the buffers that `sr_send_packet` borrows from you (i.e., `sr_send_packet` will not call `free` on the buffers that you pass to it). However, not cleaning up the memory you allocated (unless you have allocated absurd numbers) will not result in any penalty in this assignment. Do get your code working first before you get into the intricacies of memory cleanup.

```
void sr_arpcache_sweepreqs(struct sr_instance *sr)
```

The assignment requires you to send an ARP request about once every second until an ARP reply is received or your router has already sent 5 requests without a response. This function, defined in `sr_arpcache.c`, is **called every second**, and you should add the code here that iterates through the ARP request queue and re-sends any outstanding ARP requests that haven't been sent in the past second. If an ARP request has already been sent 5 times with no response, an ICMP Destination Host Unreachable message should be sent back to all the senders of packets that were waiting on a reply to this ARP request.

## Data Structures

Router (`sr_router.c/h`):

The full context of the router is housed in the struct `sr_instance` (`sr_router.h`). `sr_instance` contains information about the topology that the router is routing for as well as the routing table and the list of interfaces.

Interfaces (`sr_if.c/h`):

The stub code uses this to create a linked list of interfaces in the router instance at member `if_list`. Utility methods for handling the interface list can be found at `sr_if.h`.

Routing Table (sr\_rt.c/h):

The routing table in the stub code is read from a file (`rtable`) and stored in a linked list of routing entries in the current routing instance at member `routing_table`.

ARP Request Queue (sr\_arpcache.c/h):

You will need to queue the packets waiting on an ARP request. When an ARP reply arrives, you will have to forward any packets that were waiting on the response. Pseudocodes for these operations have been provided in `sr_arpcache.h`. You must fill out the `sr_arpcache_sweepreqs()` function in the `sr_arpcache.c` file that gets called every second to iterate through the ARP request queue and re-send ARP requests if necessary.

**NOTE:** For Project 2a, **you should not use the ARP cache**. To be clear, you can use some features of `sr_arpcache.c/h` to implement the ARP request queue, but you don't need to worry about caching the ARP replies (IP→MAC mapping) till Project 2b.

Protocol Headers (sr\_protocol.c/h):

Within the router framework you will be dealing directly with raw Ethernet packets. The stub code itself provides some data structures in `sr_protocols.h` which you may use to manipulate headers easily. There are a number of resources which describe the protocol headers in detail. Network Sorcery's RFC Sourcebook provides a condensed reference to the packet formats that you'll be dealing with (the links have been shared earlier in the [Protocols to Understand](#) section). For the actual specifications, you can also refer to the RFCs:

- IP - <https://datatracker.ietf.org/doc/html/rfc791>
- ICMP - <https://datatracker.ietf.org/doc/html/rfc792>
- ARP - <https://datatracker.ietf.org/doc/html/rfc826>

Debugging Functions:

We have provided you with some basic debugging functions in `sr_utils.h`. Feel free to use them to print out network header information from your packets. Below are some functions you may find useful:

- `print_hdrs(uint8_t *buf, uint32_t length)` - Prints out all possible headers starting from the Ethernet header in the packet.
- `print_addr_ip_int(uint32_t ip)` - Prints out a formatted IP address from a `uint32_t`. Make sure you are passing the IP address in the correct byte ordering.

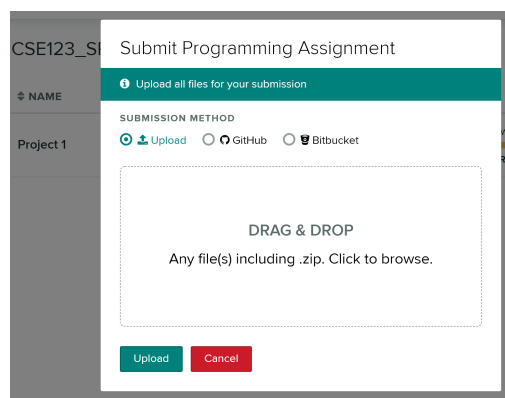
## Required Functionality

- The router must respond correctly to ICMP echo requests to its interfaces.

- The router must successfully route ICMP messages between the client and the application servers.
- The router must correctly handle ARP requests and replies.
- The router must send an ARP request **for each packet** it forwards.
- The router must queue all packets waiting for outstanding ARP replies. If a host does not respond to an ARP request within 5 seconds, the queued packets are dropped and an ICMP Destination Host Unreachable message is sent back to the source of the queued packets (one for each queued packet).
- The router must not needlessly drop packets (such as when waiting for an ARP reply).
- The router must enforce guarantees on timeouts, i.e., if an ARP request is not responded to within a fixed period of time, the ICMP Destination Host Unreachable message is generated even if no more packets arrive at the router. (You can guarantee this by implementing the `sr_arpcache_sweepreqs()` function in `sr_arpcache.c` correctly.)

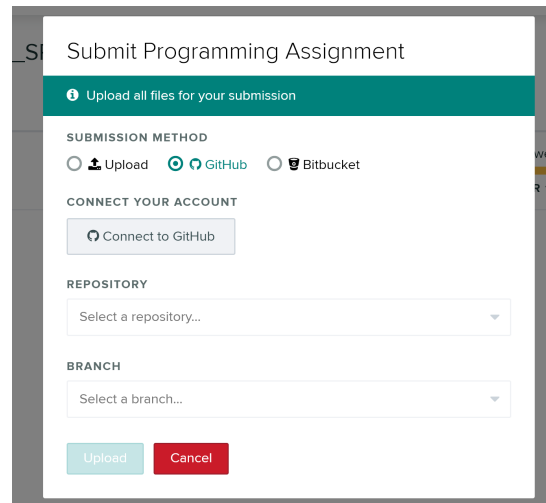
## Submission Instructions

1. Add your name and PID into the README file.
2. Add a short description/overview of your implementation in the README file.
3. Once you have modified the files, commit the changes to your private GitHub repo using:
  - a. `git add <files you have edited>`
  - b. `git commit -m <your commit message>`
4. Push/sync the changes up to your private GitHub repo using:
  - a. `git push origin master`
5. After you are done with your solution code, submit your repository to Gradescope using either of the following two options:
  - a. **Option 1 - Submit a zip:**
    - i. Zip your project folder.
    - ii. On Gradescope, choose “Upload” as the submission method.



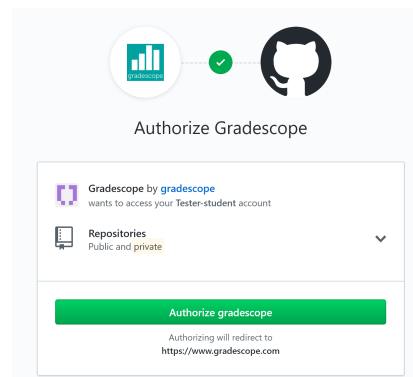
- iii. Upload your zip.
- b. **Option 2 - Submit through GitHub:**
    - i. Push your completed code to your GitHub repository.

- ii. On Gradescope, choose “GitHub” as the submission method.



The screenshot shows the 'Submit Programming Assignment' interface. At the top, there's a teal banner with the text 'Upload all files for your submission'. Below this, the 'SUBMISSION METHOD' section has three radio buttons: 'Upload', 'GitHub' (which is selected), and 'Bitbucket'. Underneath, the 'CONNECT YOUR ACCOUNT' section features a 'Connect to GitHub' button. The 'REPOSITORY' section has a dropdown menu labeled 'Select a repository...'. The 'BRANCH' section has a dropdown menu labeled 'Select a branch...'. At the bottom, there are two buttons: 'Upload' (light blue) and 'Cancel' (red).

- iii. Authorize Gradescope to access your GitHub repository by clicking “Connect to GitHub”.



- iv. Repository Menu: Select your Project 2 repository.  
v. Branch Menu: Select the `master` branch (unless you are certain that you pushed your code to some other branch that you created).