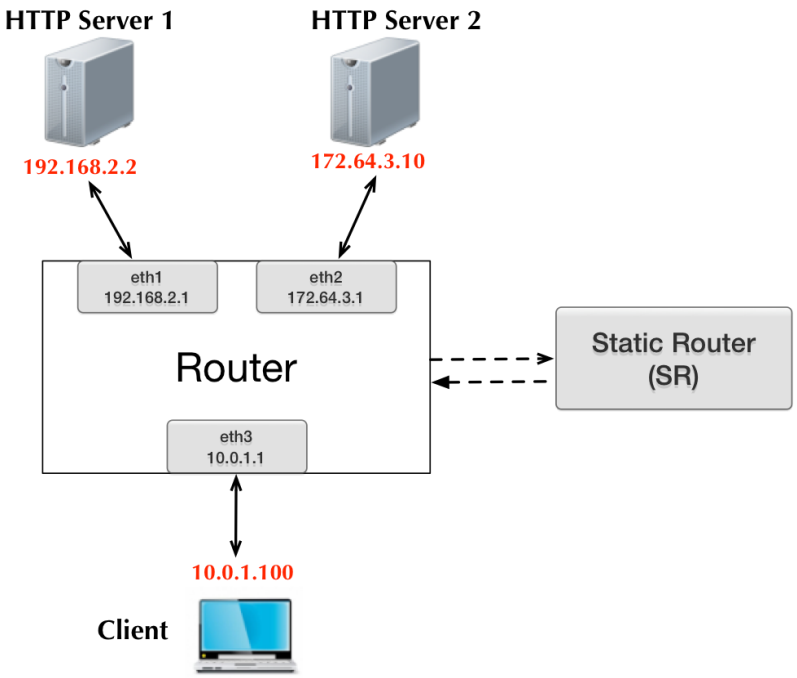

Introduction

In this lab assignment you will be writing a simple router configured under a static routing table and a static networking topology. Your router will receive raw Ethernet frames. It will process the packets just like a real router, then forward them to the correct outgoing interface. Your task is to implement the forwarding logic so packets go to the correct interface.

Your router will route real packets to HTTP servers sitting behind your router. When you have finished your router, you should be able to access these servers using regular client software (e.g., *wget* or *curl*). In addition, you should be able to *ping* and *traceroute* to and through a functioning Internet router. This is the topology you will be using for your development:



You will use Mininet to set up these topologies of emulated routers and process packets in them. Once your router is functioning correctly, you will be able to perform all of the following operations:

- Ping any of the router's interfaces from the VM (eth1, eth2, eth3);
- Traceroute to any of the router's interface IP addresses (eth1, eth2, eth3);
- Ping any of the HTTP servers from the VM (two HTTP servers);
- Traceroute to any of the HTTP server IP addresses (two HTTP servers);
- Download a file using HTTP from one of the HTTP servers (two HTTP servers);

Getting Started

Starter Code

All the code you need for this assignment is under **`lab1`** folder of your assigned git repo. You will implement all of your code inside this folder, and commit your lab using git.

Understanding a Routing Table File

Each line in the routing table (rtable) file is a routing entry and has the following format:

```
prefix        next_hop     netmask          interface
```

Here is the default routing table that you will find on the VM. The first entry is the default route.

```
0.0.0.0       10.0.1.100   0.0.0.0          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
```

Building and Running

The assignment relies on two tools: Mininet and POX. Mininet emulates a network with a single router and POX ensures that this router can communicate with your code. To

make your job easier, we have written scripts that start up Mininet and POX in the proper order. You simply need to run:

```
sudo ./run_all.sh
```

The script starts Mininet and POX in 2 different screen sessions. You can check that both are running correctly by attaching to each one.

For Mininet: Attach to the screen using `sudo screen -r mn`. You should see something like this:

```
*** Shutting down stale SimpleHTTPServers
*** Shutting down stale webservers
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
server2-eth1 172.64.3.17
server3-eth0 172.64.3.18
server2-eth2 172.64.3.33
server4-eth0 172.64.3.34
*** 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', 'serv
er4-eth0': '172.64.3.34', 'server3-eth0': '172.64.3.18', 'server2-eth1': '172.64
.3.17', 'server2-eth2': '172.64.3.33'}
*** Creating network
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** 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>
```

Detach the screen and return to the original terminal by typing Ctrl-A Ctrl-D (not just Ctrl-D).

For POX: `sudo screen -r pox` should show something like this:

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:.home.cs551.551-source.551-Labs.lab1.pox_module.cs144.ofhandler:*** ofhand
ler: 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', 'serv
er4-eth0': '172.64.3.34', 'server3-eth0': '172.64.3.18', 'server2-eth1': '172.64
.3.17', 'server2-eth2': '172.64.3.33'}

INFO:.home.cs551.551-source.551-Labs.lab1.pox_module.cs144.srhandler:created ser
ver
DEBUG:.home.cs551.551-source.551-Labs.lab1.pox_module.cs144.srhandler:SRServerLi
stener listening on 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
INFO:openflow.of_01:[Con 1/165919369135692] Connected to 96-e7-1d-0f-06-4c
DEBUG:.home.cs551.551-source.551-Labs.lab1.pox_module.cs144.ofhandler:Connection
 [Con 1/165919369135692]
DEBUG:.home.cs551.551-source.551-Labs.lab1.pox_module.cs144.srhandler:SRServerLi
stener catch RouterInfo even, info={'eth2': ('172.64.3.1', 'e2:a9:b2:39:76:c8',
'10Gbps', 2), 'eth1': ('192.168.2.1', 'd6:e1:e6:d1:c3:8a', '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', '255.255.25
5.255', 'eth2')]
Ready.
POX>
```

Once again, detach the screen with Ctrl-A Ctrl-D.

Now all that is left to do is run the router logic (i.e. the code you need to write). To check that the setup was done properly, you should start by running the reference solution binary that we provide (`lab1/sr_solution` in the VM). You can either run it in the ***current SSH terminal*** (which is not a good idea because then you will not have a prompt anymore and will not be able to run any tests), or in a ***new SSH terminal*** (simply open a new connection), or ***in a screen session***. Let's see how to do it with screen here (do "chmod a+x sr_solution" to make the solution binary an executable):

```
screen -S sr
./sr_solution
```

The output should look something like this: (there may be small differences, such as different HWaddrs and a slightly different Mask for the 172.64.3.0 network).

```
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
10.0.1.0                10.0.1.100      255.255.255.0   eth3
192.168.2.2             192.168.2.2     255.255.255.255 eth1
172.64.3.0              172.64.3.10     255.255.255.0   eth2
---------------------------------------------
Client root connecting to Server localhost:8888
Requesting topology 0
successfully authenticated as root
Loading routing table from server, clear local routing table.
Loading routing table
---------------------------------------------
Destination     Gateway         Mask    Iface
10.0.1.0                10.0.1.100      255.255.255.0   eth3
192.168.2.2             192.168.2.2     255.255.255.255 eth1
172.64.3.0              172.64.3.10     255.255.255.0   eth2
---------------------------------------------
Router interfaces:
eth3    HWaddr52:42:0a:08:f4:5a
        inet addr 10.0.1.1
eth2    HWaddr2a:ce:cc:02:7c:d8
        inet addr 172.64.3.1
eth1    HWaddra2:14:9c:a1:e4:d6
        inet addr 192.168.2.1
 <-- Ready to process packets -->
```

To detach the screen and return to the main terminal, use Ctrl-A Ctrl-D. You can return to the session anytime with **`screen -r sr`**. If you want to kill the session while attached to it, use Ctrl-D.

If everything is running correctly, you should be able to run the following commands from your **mininet screen**:

```
client ping server1
```

```
pingall
```

The first command runs a ping from **`client`** to **`server1`**, the second command runs a ping between each pair of **`client`**, **`server1`** and **`server2`**. When these commands run, you should be able to see messages on the terminal window where **`sr_solution`** runs.

Now, to build and test your lab code (see below on what you need to do for the lab), you can simply do as follows:

```
cd router

make

./sr
```

If your implementation is correct, you should see similar output (as our reference solution) when you run the ping commands listed above.

If you run into issues, try 'sudo ./killall.sh' and 'make clean' first.

Background: Routing

This section has an outline of the forwarding logic for a router, although it does not contain all the details. There are two main parts to this assignment: **IP forwarding** and **handling ARP**.

When an IP packet arrives at your router, it arrives inside an Ethernet frame. Your router needs to check if it is the final destination of the packet, and if not, forward it along the correct link based on its forwarding table. The forwarding table names the IP address of the next hop. The router must use ARP to learn the Ethernet address of the next hop IP address, so it can address the Ethernet frame correctly.

IP Forwarding and ARPs

Given a raw Ethernet frame, if the frame contains an IP packet whose destination is not one of the router's interfaces:

1. Check that the packet is valid (is large enough to hold an IP header and has a correct checksum).
2. Decrement the TTL by `1`, and recompute the packet checksum over the modified header.
3. Find out which entry in the routing table has the longest prefix match with the destination IP address.
4. Check the ARP cache for the next-hop MAC address corresponding to the next-hop IP. If it's there, send it. Otherwise, send an ARP request for the

next-hop IP (if one hasn't been sent within the last second), and add the packet to the queue of packets waiting on this ARP request.

This is a high-level overview of the forwarding process. More low-level details are below. For example, if an error occurs in any of the above steps, you will have to send an ICMP message back to the sender notifying them of an error. You may also get an ARP request or reply, which has to interact with the ARP cache correctly.

Protocols to Understand

Ethernet

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

Internet Protocol

Before operating on an IP packet, you should verify its checksum and make sure it is large enough to hold an IP packet. You should understand how to find the longest prefix match of a destination IP address in the routing table described in the "Getting Started" section. 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.

Internet Control Message Protocol

ICMP sends control information. 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).

- **Destination net unreachable (type 3, code 0)**

Sent if there is a non-existent route to the destination IP (no matching entry in routing table when forwarding an IP packet).

- **Destination host unreachable (type 3, code 1)**

  Sent after **five ARP requests** were sent to the next-hop IP without a response.

- **Port unreachable (type 3, code 3)**

  Sent if an IP packet containing a UDP or TCP payload is sent to one of the router's interfaces. This is needed for *traceroute* to work.

- **Time exceeded (type 11, code 0)**

  Sent if an IP packet is discarded during processing because the TTL field is `0`. This is also needed for *traceroute* to work.

Some ICMP messages may come from the source address of any of the router interfaces, while others must come from a specific interface: refer to RFC 792 for details. As mentioned above, the only incoming ICMP message destined towards the router's IPs that you have to explicitly process are ICMP echo requests. You may want to create additional structs for ICMP messages for convenience, but make sure to use the `packed` attribute so that the compiler doesn't try to align the fields in the struct to word boundaries: GCC Type Attributes. In addition to the RFC 792, you can also look at ICMP wiki page for detailed packet format.

Address Resolution Protocol

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 would 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.

To lessen the number of ARP requests sent out, you are required to cache ARP replies. Cache entries should timeout after `15` seconds to minimize staleness. The provided ARP cache class already times the entries out for you.

When forwarding a packet to a next-hop IP address, the router should first check the ARP cache for the corresponding MAC address before sending an ARP request. In the

case of a cache miss, an ARP request should be sent to a target IP address about once every second until a reply comes in. If the ARP request is sent five times with no reply, an ICMP destination host unreachable is sent back to the source IP as stated above. The provided ARP request queue will help you manage the request queue.

In the case of an ARP request, you should only send an ARP reply if the target IP address is one of your router's IP addresses. In the case of an ARP reply, you should only cache the entry if the target IP address is one of your router's IP addresses.

Note that 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.

IP Packet Destinations

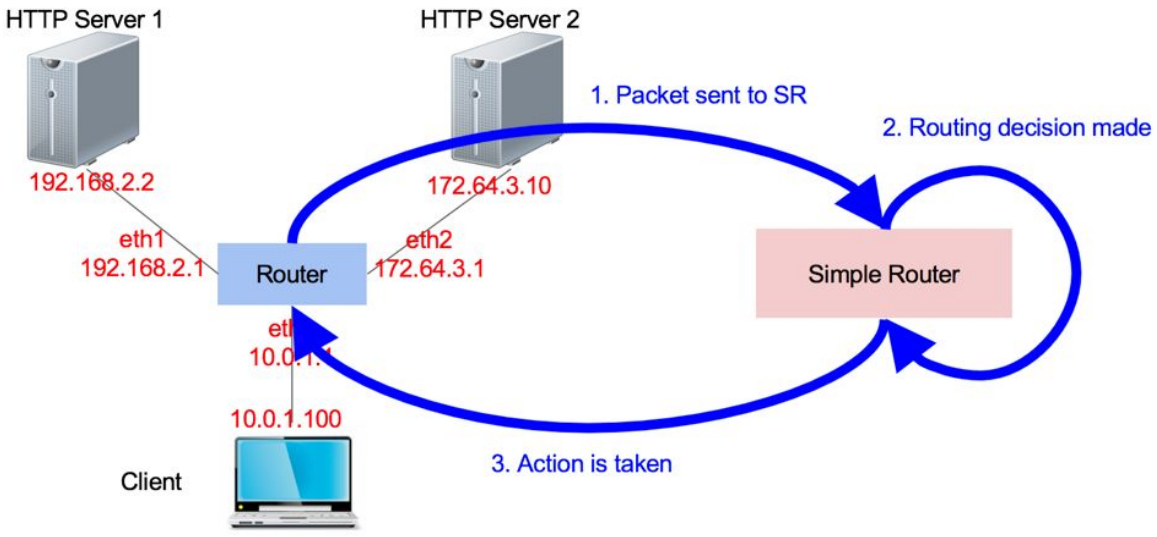
An incoming IP packet may be destined for one of your router's IP addresses, or it may be destined elsewhere. If it is sent to one of your router's IP addresses, you should take the following actions, consistent with the section on protocols above:

- If the packet is an ICMP echo request and its checksum is valid, send an ICMP echo reply to the sending host.
- If the packet contains a TCP or UDP payload, send an ICMP port unreachable to the sending host.
- Otherwise, ignore the packet.

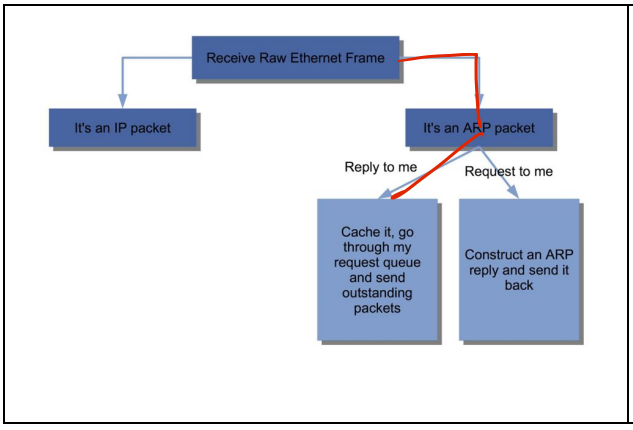
Packets destined elsewhere should be forwarded using your normal forwarding logic.

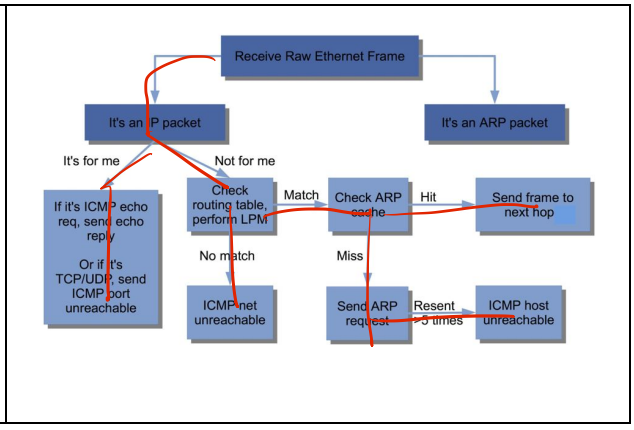
Building your router

Overview



The above figure shows the packet path inside the emulated Mininet environment for this assignment. Your code is responsible for making the routing decision: (a) look at the routing table; (b) figure out which interface to forward the packet to; (c) make necessary changes to packet. Specifically, your router will handle ARP and IP packets. The following two figures show the functional flow chart.





Code overview

Basic Functions

Your router receives and sends Ethernet frames. The basic functions to handle these functions are:

- **`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.

- **`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 `len` bytes of `buf` out of the interface specified by `iface`. The `buf` parameter should point to a valid Ethernet frame, and `len` should not go past the end of the frame. 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 (that is, `sr_send_packet` will not call `free()` on the buffers that you pass it).

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

  The assignment requires you to send an ARP request about once a second until a reply comes back or you have sent five requests. This function is defined in `sr_arpcache.c` and called every second, and you should add code 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 been sent 5 times with no response, a destination host unreachable should go back to all the sender of packets that were waiting on a reply to this ARP request.

Data Structures

The Router (sr_router.h)

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

Interfaces (sr_if.c/h)

After connecting, the server will send the client the hardware () information for that host. 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.c/h`**.

Note that The server and client here are in the context of the VNS system (which emulates the topology). You can refer to sr_add_interface (sr_if.c) and **`sr_handle_hwinfo(sr_vns_comm.c)`** for details.

The Routing Table (sr_rt.c/h)

The routing table in the stub code is read on from a file (default filename `rtable`, can be set with command line option `-r`) and stored in a linked list of routing entries in the current routing instance (the member name is `routing_table`).

The ARP Cache and ARP Request Queue (sr_arpcache.c/h)

You will need to add ARP requests and packets waiting on responses to those ARP requests to the ARP request queue. When an ARP response arrives, you will have to remove the ARP request from the queue and place it onto the ARP cache, forwarding any packets that were waiting on that ARP request. Pseudocode for these operations is provided in `sr_arpcache.h`. The base code already creates a thread that times out ARP cache entries 15 seconds after they are added for you. You must fill out the `sr_arpcache_sweepreqs()` function in `sr_arpcache.c` that gets called every second to iterate through the ARP request queue and re-send ARP requests if necessary. Psuedocode for this is provided in `sr_arpcache.h`.

Protocol Headers (sr_protocol.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 you'll be dealing with Ethernet, IP, ICMP, and ARP.

Debugging

Debugging is a critical skill in programming and systems programming specifically. Because your error might be due to some tiny, low-level mistake, trying to read through pages and pages of `printf()` output is a waste of time. While `printf()` is of course useful, you will be able to debug your code much faster if you also log packets and use *gdb*.

Protocols: Logging Packets

You can log the packets received and generated by your SR program by using the `-l` parameter. The file will be in `pcap` format, so you can use *tcpdump* to read it.

```
./sr -l logfile.pcap
```

Besides SR, you can also use Mininet to monitor the traffic that goes in and out of the emulated nodes, i.e., `router`, `server1` and `server2`. Mininet provides direct access to each emulated node. Using `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
```

Router

Using GDB

We encourage you to use *gdb* to debug any router crashes, and *valgrind* to check for memory leaks. These tools should make debugging your code easier and using them is a valuable skill to have going forward.

Debugging Functions

We have provided you with some basic debugging functions in `sr_utils.h`, `sr_utils.c`. 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.

Testing

To make sure that you have implemented things correctly, you should run the following command:

```
$ sh run_tests.sh
```

If you pass all of these tests, you will get a full score for this lab, otherwise, you will only get partial credit (depending on how many of the tests you pass).

There are two sets of tests, defined in **`base.xml`** and **`extended.xml`**. The latter file uses an extended topology, with two nodes (**`server3`**, **`server4`**) connected to **`server2`**. These tests include:

- Pings and traceroutes between different nodes in the topology
- Iperf to make sure your router implementation is not too slow
- Valgrind to check for memory leaks
- Downloading a large image from each of the HTTP servers
- Checking to see that your implementation sends ICMP host unreachable

You might want to read the XML test source files to understand how the tests work and what they are trying to do. The tests use scripts in the **`tester`** directory, feel free to look at files in that directory as well.

Some of these tests, in particular, some of the traceroute tests may take a while, so please be patient. If you think your program works correctly, but some of the tests fail, reboot your VM and re-run the tests again.

After executing the tests in each of **`base.xml`** and **`extended.xml`**, the testing software will print out your score for each set of tests.

Submission

- Final source code: Remember one of the goals of this assignment is for you to build a static router that can perform basic routing functions. Therefore, your final code MUST be runnable as a single shell command (“./router/sr”). We will test your code in the same setup.
- **Report.pdf**: This file should be 1-3 page document describing your **high-level design**, and **implementation**, together with any limitations in your implementation. You should also include this file in the “lab1” directory.

Once you have finished the lab, you should do:

```
git commit -a -m 'Lab 1 submission'
git push
```

You should also, of course, be committing to your github repo frequently to save your work.