CMPUT 313 – Computer Networks **Winter 2025**

Assignment 3 – Implementing a Network Router

This assignment is due **April 8, 2025, at 5:00pm** Mountain Time.

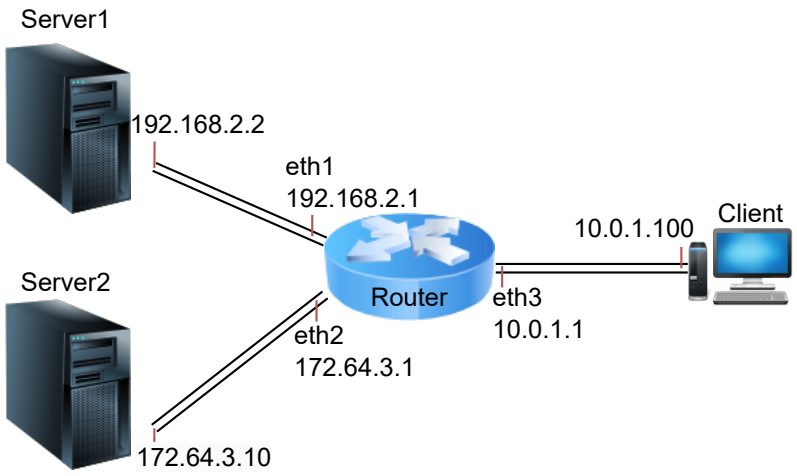
Objective

In this assignment, you will develop a simple router with a static routing table and learn about ICMP, IP, ARP, and Ethernet protocols. Upon receiving Ethernet frames, the router should process and forward them to the correct output link. The starter code provides the framework to receive Ethernet frames, your job is to implement the forwarding logic so that the packets are sent to the correct interface after updating the source and destination MAC addresses.

Overview

Your router must be capable of forwarding packets from one host to another host sitting on the other side of the router, e.g. from a client to a server and vice versa. Thus, once the forwarding logic is implemented, the client should be able to reach a server using standard network utilities, such as `ping`, `tracepath`, and `wget`. Moreover, each end host should be able to ping and trace the route to the router's interfaces.

We use Mininet to emulate a network on a single machine. The topology of this network is shown below. It consists of three emulated hosts, one client and two HTTP servers, and one emulated switch, which is our router.



If the router works correctly, all of these tests will be successful:

- Client pinging any of the router's interfaces, i.e. 192.168.2.1, 172.64.3.1, 10.0.1.1;
- Client tracing the route to any of the router's interfaces;
- Client pinging any of the servers (and vice versa);
- Client tracing the route to any of the servers (and vice versa);
- Client downloading a Web object, e.g. a file, using HTTP from either one of the servers;

Additional requirements are laid out in the "Required Functionality" section. Read them carefully.

Note: the above network is already defined in Mininet, so you will not write code using Mininet's Python API in this assignment. The router must be implemented in C, outside of Mininet.

Getting Started

It is required to use the CMPUT 313 VM to complete the assignment. The VM comes with a specific version of Mininet and POX, a SDN controller used to direct packets between the Mininet switch and the router implemented outside of Mininet. You do not need to understand how POX works to complete the assignment.

Environment Setup

The first step is to clone the assignment repository that you accepted in GitHub Classroom. Once this is done, run the following commands from a terminal window opened in the VM to install the assignment dependencies, configure the environment, and set up the modules for POX:

```
cd assignment3-simplerouter-x
sudo ./setup.sh
```

Note `x` must be replaced with your GitHub username.

Emulating the network

You need to have 3 terminal windows opened in the VM for running the controller, network emulator, and router.

1. **Controller:** Mininet requires a controller which is POX in this assignment. This controller transfers packets between the emulated switch in Mininet and your router. POX is installed in the VM and the modules were configured in the previous step. To start the controller, run the following command in the first terminal:

   ```
   ./run_pox.sh
   ```

   Wait until you see this output and the prompt before proceeding to the next step:

   ```
   POX 0.0.0 / Copyright 2011 James McCauley
   DEBUG:.home.mininet.assignment3.pox_module.cmput313.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:.home.mininet.assignment3.pox_module.cmput313.srhandler:created server
   DEBUG:.home.mininet.assignment3.pox_module.cmput313.srhandler:SRServerListener listening on 8888
   DEBUG:core:POX 0.0.0 going up...
   DEBUG:core:Running on CPython (2.7.18/Nov 21 2024 09:58:47)
   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
   Ready.
   POX>
   ```

   Note that POX is used "under the hood" to allow your router to process the packets generated by the hosts in Mininet. You do not need to understand how POX acts as an intermediary ("VNS server") sending and receiving these packets from your router. This functionality is implemented in `sr_vns_comm.c` and in the POX module. Thus, do not change anything in `sr_main.c`, `sr_vns_comm.c`, `run_pox.sh`, or `pox.py`. Let POX run and switch to the next terminal.

2. **Mininet:** In the second terminal, start Mininet by running the following command:

   ```
   ./run_mininet.sh
   ```

   If Mininet runs successfully, these lines will be printed to the terminal:

   ```
   *** 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
*** 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
c0
*** 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>
```

If you switch back to the first terminal, you will see that the mininet switch has connected to the controller:

```
INFO:openflow.of_01:[Con 1/None] closing connection
INFO:openflow.of_01:[Con 2/112186155550022] Connected to 66-08-5f-f3-45-46
DEBUG:.home.mininet.assignment3.pox_module.cmput313.ofhandler:Connection [Con 2/112186155550022]
DEBUG:.home.mininet.assignment3.pox_module.cmput313.srhandler:SRServerListener
catch RouterInfo even, info={
'eth3': ('10.0.1.1', '22:a3:c5:ee:c8:8d', '10Gbps', 3),
'eth2': ('172.64.3.1', '52:6b:b4:ec:09:af', '10Gbps', 2),
'eth1': ('192.168.2.1', 'f2:bc:da:fd:16:f3', '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 the emulator running and switch to the third terminal.

3. **Router:** In the last terminal, run the router that you have built. To help you debug your code and understand the expected behavior of the router, we have provided an executable binary of the router, called `sr_solution`. The ultimate goal in this assignment is to implement the router's forwarding logic and build an executable file like this. Then, you can run your executable file in this step.

   To get a sense of how the router should function before starting the implementation, run the provided binary:

   ```
   ./sr_solution
   ```

   It will output the following lines:

   ```
   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
   10.0.1.100    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
   ---------------------------------------------
   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
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
-----------------------------------------
Router interfaces:
eth3  HWaddr 8e:5a:1f:45:ce:90
        inet addr 10.0.1.1
eth2  HWaddr ca:67:4f:02:e4:6b
        inet addr 172.64.3.1
eth1  HWaddr ca:91:bf:78:1e:7d
        inet addr 192.168.2.1
<-- Ready to process packets -->
```

Connectivity Tests

Assuming the controller, router, and emulator are running, you can test out connectivity in the network. To find the IP address of each host and switch interface, refer to the `IP_CONFIG` file.

Let's switch to the terminal where Mininet is running. Type the hostname, e.g. `server1`, `server2`, or `client`, followed by a command in the Mininet console to run that command on that particular host.

- **Ping Test:** Test if a host can reach other hosts or the router using the `ping` command. For example, to test if `client` can ping `server1`, enter this command in the Mininet console:

  ```
  mininet> client ping -c 3 192.168.2.2
  PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
  64 bytes from 192.168.2.2: icmp_req=1 ttl=63 time=66.9 ms
  64 bytes from 192.168.2.2: icmp_req=2 ttl=63 time=49.9 ms
  64 bytes from 192.168.2.2: icmp_req=3 ttl=63 time=68.8 ms
  ```

  You may want to take a look at the output produced by the router and controller as a result of this command.

- **Tracepath Test:** Use the `tracepath` (or `traceroute`) command to check the route between two nodes:

  ```
  mininet> client tracepath -n 192.168.2.2
  1?:         [LOCALHOST]                                     pmtu 1500
  1:          10.0.1.1                                        38.445ms asymm 29
  1:          10.0.1.1                                        46.965ms asymm 29
  2:          192.168.2.2                                     130.594ms reached
          Resume: pmtu 1500 hops 2 back 2
  ```

- **Webserver Test:** Test if the web server is working properly at `server1` and `server2` by issuing an HTTP request using the `wget` or `curl` command:

  ```
  mininet> client wget http://192.168.2.2
  --2025-03-14 12:30:58--  http://192.168.2.2/
  Connecting to 192.168.2.2:80... connected.
  HTTP request sent, awaiting response... 200 OK
  Length: 161 [text/html]
  Saving to: 'index.html.2'

  index.html.2        100%[===================>]     161  --.-KB/s    in 0s

  2025-03-14 12:30:58 (658 KB/s) - 'index.html.2' saved [161/161]
  ```

Notice that if you terminate the execution of the router, the above tests will not succeed.

Your Task

Your task is to complete the implementation of the router. The code skeleton can be found in the `router` directory. You should write code in `sr_router.c`, `sr_rt.c`, `sr_arpcache.c`, and `sr_utils.c`. You are allowed to write helper functions and change the implementation of other functions in these files, if necessary. However, do not rename these files and do not change the header of a function that is already implemented in the starter code.

To build and run your router, use the makefile in the `router` directory as shown below:

```
cd router/
make
./sr
```

There are two configuration files that you should know about:

- `IP_CONFIG`: Lists the IP address assigned to each host or router interface.

```
> cat assignment3-simplerouter-x/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
```

- `router/rtable`: The static routing table used by the router. The columns, from left to right, represent the destination address, gateway address, subnet mask, and interface name.

```
> cat assignment3-simplerouter-x/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
```

Next, we discuss the two main tasks: packet forwarding and ARP handling.

Destination-based Forwarding Logic

Given a raw Ethernet frame, if the frame contains an IP datagram that is not destined for one of the router's interfaces, the router should:

- Parse and sanity-check the packet, verifying the IP header checksum and confirming that it meets the minimum length requirement.
- Decrement the TTL value by 1 and recompute the IP header checksum.
- Find out which entry in the routing table is the longest prefix match with the destination IP address. Note there might be overlapping subnets in the routing table (e.g. overlapping /16 and /24 addresses in `rtable`), hence more than one match. The correct implementation should always pick the longest prefix match.
- Check the ARP cache for the next-hop MAC address corresponding to the next-hop IP address. If it is there, retrieve it as the destination MAC address, assemble the packet and send it. Otherwise, send an ARP request for the next-hop IP if one has not been sent within the last second, and add the packet to the queue of packets waiting on this ARP request. The destination MAC address can be extracted from the Ethernet header of the ARP reply.

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

Protocols to Understand

Ethernet

The router gets raw Ethernet frames and must send raw Ethernet frames. You should understand source and destination MAC addresses, and how the router forwards a packet by changing its destination MAC address to the MAC address of the next hop.

Internet Protocol

Before operating on an IP datagram, you must verify its checksum, and ensure the "Header Length" is not less than the minimum length of the IPv4 header and the "Total Length" is not less than the specified header length. If an invalid checksum or length is detected, the router will silently drop the packet (i.e. without sending ICMP to source). Note that you do not need to worry about the maximum allowable MTU and IP fragmentation in this assignment. After performing these sanity checks, the longest prefix match of the destination IP address must be found in the routing table as described earlier. If you determine that a datagram should be forwarded, decrement the TTL field in the IP header and recompute the checksum over the changed header before forwarding it to the next hop.

Internet Control Message Protocol

The router will use the ICMP protocol to send an error message or a ping reply to a sending host. It must properly generate and send the following ICMP messages (including the ICMP header checksum) in response to the sending host under the specified conditions:

Name	Type	Code	Explanation
Echo reply	0	0	Sent in response to an echo request (ping) to one of the router's interfaces. Note an echo request sent elsewhere should be forwarded to the next hop address as usual.
Destination net unreachable	3	0	Sent if there is no matching entry in the routing table for the destination IP, when forwarding a packet.
Destination host unreachable	3	1	Sent if five ARP requests were sent to the next-hop IP without a response.
Port unreachable	3	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	11	0	Sent if an IP packet is discarded during processing because the TTL field is 0. This is also needed for traceroute to work.

The source address of an ICMP message can be the IP address of any of the input interfaces, as specified in RFC 792. As mentioned earlier, the only incoming ICMP message destined towards the router's IPs that you have to explicitly process is the echo request. Before operating on a received ICMP echo request, you should verify the ICMP checksum and sanity-check the IP packet that encapsulates it, as described earlier. You may want to create additional structures for ICMP messages for convenience; in that case, be sure to use the packed attribute so that the compiler does not try to align the fields in the structure to word boundaries.

Address Resolution Protocol

ARP is used to determine the MAC address corresponding to the IP address of the next hop stored in the routing table. Without the ability to generate an ARP request and process the ARP reply, the router would not be able to fill out the destination MAC address field of the raw Ethernet frame that must be sent out.

To reduce the number of ARP requests that are sent, ARP replies are cached. Cache entries should time out 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 event of a cache miss, an ARP request should be sent to a target IP address about once every second until a reply is received. If the ARP request is sent five times with no reply, an ICMP destination host unreachable message must be sent back to the source. The provided ARP request queue will help you manage the request queue.

When receiving an ARP request, you should only send an ARP reply if the target IP address is one of your router's interface addresses. When receiving 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.

Packets Destined for Router

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

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

As stated before, packets destined elsewhere must be forwarded using the forwarding logic.

Starter Code Overview

Key Data Structures

Router (`sr_router.h`):

The full context of the router is kept in the `sr_instance` structure, which is declared in this header file. Specifically, this structure contains information about the network topology, as well as the routing table, ARP cache, and the list of router interfaces. In `sr_router.h/c`, the `send_icmp_msg` function is partially implemented. You should implement other functionality of the router in `sr_router.h/c`, such as checking the ARP cache and handling IP/ARP packets. You may also do additional initialization operations in the `sr_init()` function.

Router's interfaces (`sr_if.h`):

Each router interface is a `sr_if` structure, which is declared in this header file. In `sr_if.c`, a linked list of these interfaces, called `if_list`, is created in the `sr_instance` structure. The methods for managing the

interface list are implemented in `sr_if.c`. You may also use the helper functions provided in `sr_if.c`, such as `sr_get_interface`, `sr_get_interface_by_ip`, and `sr_get_interface_by_mac`.

Router's routing table entries (`sr_rt.h`):

Each entry in the routing table is a `sr_rt` structure, which is declared in this header file. In `sr_rt.c`, the routing table is created by reading entries from a file (the default filename is `rtable`, which can be set with command line option -r) and adding them to a linked list of entries called `routing_table` in the `sr_instance` structure. You must complete the `longest_prefix_matching` function in `sr_rt.c`.

Router's ARP cache and ARP request queue (`sr_arpcache.h`):

The router's ARP cache is a `sr_arpcache` structure variable, which consists of an array of `sr_arpentry` structures and a linked list of pending ARP requests, each being a `sr_arpreq` structure.

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

Protocol headers (`sr_protocols.h`):

In the router, you will be dealing with raw Ethernet packets directly. The starter code provides some data structures in `sr_protocols.h`, which you may use to parse and update the protocol headers easily. For example, the raw Ethernet header contains source and destination MAC addresses, and the Ethertype field (see struct `sr_ethernet_hdr`).

The protocol headers are described in class and on Wikipedia. Here are some useful pointers:

- Ethernet
- IP
- ICMP
- ARP

For the actual specifications, refer to the RFC for ARP (RFC826), IPv4 (RFC791), and ICMP (RFC792).

Important Functions

Your router receives a raw Ethernet frame and sends a raw Ethernet frame when sending a reply to the sending host or forwarding the frame to the next hop. The functions that implement this functionality are listed below:

```
void sr_handlepacket(struct sr_instance* sr, uint8_t* buf, unsigned int len, char* iface)
```

This function, defined in `sr_router.c`, is called by the router upon the arrival of a packet to process and forward the packet if needed. The `buf` argument points to the buffer that contains the full packet including the Ethernet

header. The name of the receiving interface is passed to the method as `iface`. Do not free the buffer given to you in `sr_handlepacket`. This is why the buffer is labeled as "lent" in the comment.

```
void send_icmp_msg(struct sr_instance* sr, uint8_t* packet, unsigned int len, \
                                                    uint8_t type, uint8_t code)
```

This function, defined in `sr_router.c`, is called to send out an ICMP packet. We provided the implementation for constructing the headers and assembling the ICMP packet. However, this function is incomplete and you expected to finish the implementation by checking the ARP cache and sending an ARP request in case of a cache miss. Note that the starter code just initializes the source and destination MAC addresses as `00-00-00-00-00-00`. You should correctly set the source and destination MAC addresses when checking the ARP cache.

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

This function, defined in `sr_vns_comm.c`, sends out an arbitrary packet of length, `len`, via the interface `iface`. Thus, you must call it in your code everything that the router has to send out a packet. Note that you are responsible for doing memory management on the buffer that `sr_send_packet` borrows from you (i.e., `sr_send_packet` will not call free on the buffer passed to it).

```
void sr_arpcache_sweepreqs(struct sr_instance *sr)
void handle_arpreq(struct sr_instance *sr, struct sr_arpreq* request)
```

When there is a cache miss, the router will send an ARP request about once a second until a reply comes back or a total of five requests are sent. This behavior must be implemented in the `sr_arpcache_sweepreqs` and `handle_arpreq` functions, which are defined in `sr_arpcache.c`. More specifically, `sr_arpcache_sweepreqs`, which is called every second, must iterate through the ARP request queue and resend any outstanding ARP requests that have not been sent in the past second. If an ARP request has been sent 5 times with no response, an ICMP packet with the destination host unreachable code must be sent to the senders of the packets that were waiting on a reply to this ARP request. The comments in the header file `sr_arpcache.h` provide more information about the implementation.

```
struct sr_rt* longest_prefix_matching(struct sr_instance *sr, uint32_t ip)
```

This function is defined in `sr_rt.c` to find and return the entry in the routing table that is the longest prefix match with the destination IP address provided as the last argument.

```
int validate_ip(uint8_t* packet, unsigned int len)
```

This function is defined in `sr_utils.c` to perform sanity-check on the IP packet. It should return 0 if the incoming packet passed sanity-check, otherwise -1. Note the first argument is a buffer containing the full packet including the Ethernet header, and the second argument is the length of this packet.

```
int validate_icmp(uint8_t* packet, unsigned int len)
```

This function is defined in `sr_utils.c` to perform sanity-check on the ICMP packet. It should return 0 if the incoming packet passed sanity-check, otherwise -1. Note the first argument is a buffer containing the full packet including the Ethernet header, and the second argument is the length of this packet.

Required Functionality

- The router must successfully route packets between the emulated end hosts.
- The router must correctly handle ARP requests and replies.
- The router must correctly handle traceroutes through it (when it is not the destination) and to it (when it is the destination).

- The router must respond correctly to ICMP echo requests.
- The router must handle TCP/UDP packets sent to one of its interfaces. In this case, the router should respond with an ICMP port unreachable.
- The router must maintain an ARP cache whose entries are invalidated after a 15-second timeout period.
- The router must queue all packets waiting for outstanding ARP replies. If a host does not respond to 5 successive ARP requests, the queued packet is dropped and an ICMP host unreachable message is sent back to the source of the queued packet.
- The router must not needlessly drop packets (for example, when waiting for an ARP reply)
- The router must enforce guarantees on timeouts – that is, if an ARP request is not responded to within a fixed period of time, the ICMP host unreachable message is generated even if no more packets arrive at the router. (Note: you can ensure this by implementing the `sr_arpcache_sweepreqs` function in `sr_arpcache.c` correctly.)

Final Notes

Logging Packets

You can log the packets received and generated by your router program by using the "-l" option when running the program. The file will be in pcap format, i.e., you can use Wireshark or tcpdump to read it.

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

Debugging Functions

We have provided you with some basic debugging functions in `sr_utils.h/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 to pass the IP address in the correct byte ordering.

Length of Assignment

It takes a significant amount of time to complete the assignment and test your implementation properly, so start early and take advantage of the labs to ask questions from TAs. To give you a rough idea of the amount of code that must be written, there is around 400 lines of C, including empty lines and comments, in our reference solution.

Tools

There are several tools that can make your life considerably easier.

- `make` is a utility used to conditionally recompile your program based on the timestamps of the source code, object files, and executable. You may add additional targets in the provided Makefile, but they should not interfere with the existing targets.
- `gdb` is the GNU debugger. gdb is an interactive, source-level debugger. It allows you to trace through your code as it executes and examine the program state. If your program seg faults, it will show you where. Learning to use gdb effectively will save you many hours of debugging hell. To compile with gdb, simply

add the '-g' flag to the `gcc` command. You can then start gdb with your program with the command `gdb progname`. Type 'help' from the gdb prompt for more information.

- `valgrind` is a collection of debugging and profiling tools. Use Valgrind's Memcheck tool to ensure that your program is not leaking memory and that there are no potential segfaults lurking in your code. The basic procedure for running valgrind is to run the command `valgrind progname progarg1 progarg2...`, where `progname` is the name of the program and `progarg1` is an argument to the program. Run `valgrind --help` for additional options.

Grading

- Longest prefix matching [15 points]
- Implementation of ARP cache [10 points]
- Layer-3 validation of incoming packets [10 points]
- Adherence to other requirements [15 points]
  e.g. README is included, compiles cleanly, has no memory leaks, headers are intact for the functions implemented in the starter code
- Passing connectivity tests [50 points]

Submission

Submit all the required files via GitHub Classroom.

The following files must be included in your repository.

1. `README.md`, containing a header (see below), explains any important design decision that you have made, cites all sources that you used, and describes how your implementation was tested. You may use a markup language, such as Markdown, to format this plain text file.

   At the very top of the README file include **a header** with the following information: **your name, student ID (SID), and CCID**. For example:

   ```
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   # Name : Jane Doe
   # SID : 1234567
   # CCID : jdoe
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   ```

2. All files required to build your project including the header files and C files, and everything provided in the starter code.

Misc. Notes

- This assignment must be completed individually without consultation with anyone. Questions can be asked from TAs in the labs.
- The code must be written in C and compiled using `gcc` with the `-Wall` flag. Your code should compile cleanly, i.e. producing no warnings or errors.

- All code must be written in the VM image provided on Canvas. **Submissions must run in the VM, where they will be marked.** Be absolutely certain to test this.
- Your router does not have to support IPv6.
- It is recommended that you implement the IP and ICMP packet validation at the end, as under normal conditions, you router will function correctly without these checks.