CS 118: Computer Network Fundamentals - Fall 2018

Project 1: Build Your Own Router

❖Overview

In this project, you will be writing a simple router with a static routing table. Your router will receive raw Ethernet frames and process them just like a real router: forward them to the correct outgoing interface, create new frames, etc.

The starter code will provide the framework to receive Ethernet frames; your job is to create the frame processing, handling and forwarding logic.

You are allowed to use some high-level abstractions, including C++11 extensions, for parts that are not directly related to networking, such as string parsing, multi-threading, etc.

This is NOT a group project. Each student works on the project individually.

You are PROHIBITED to make your code public during the class or any time after the class. You are encouraged to host your code in PRIVATE repositories on GitHub, GitLab, or other places.

❖Task Description

This assignment runs on top of Mininet which was built at Stanford. Mininet allows you to emulate a network 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 if you're curious, you can learn more information about Mininet on its official website.

Your router will route real packets between emulated hosts in a single-router topology. The project environment and the starter code has the following default topology:

```
                                              +----------------+ server1-eth0
                                              |                  192.168.2.2/24
                                              +                   +-----------+
                                          192.168.2.1/24          |           |
                                          sw0-eth1                |  server1  |
+-----------+               +------------------+                  |           |
|           |               |                  |                  +-----------+
|  client   |               |   SimpleRouter   |
|           |               |      (sw0)       |
+-----------+               |                  |
   client-eth0              +------------------+                  +-----------+
   10.0.1.100/8           sw0-eth3        sw0-eth2                |           |
      +                 10.0.1.1/8        172.64.3.1/16           |  server2  |
      |                      +                +                   |           |
      |                      |                |                   +-----------+
      |                      |                |                  server2-eth0
      +----------------------+                +----------------+ 172.64.3.10/16
```

The corresponding routing table for the SimpleRouter `sw0` in this default topology:

Destination	Gateway	Mask	Interface
0.0.0.0	10.0.1.100	0.0.0.0	sw0-eth3
192.168.2.2	192.168.2.2	255.255.255.0	sw0-eth1
172.64.3.10	172.64.3.10	255.255.0.0	sw0-eth2

Do not hardcode any IP addresses, network, or interface information.

There are three main parts in this assignment:

1. Handle Ethernet frames
2. Handle ARP packets
3. Handle IPv4 packets

1. Ethernet Frames

A data packet on a physical Ethernet link is called an Ethernet packet, which transports an Ethernet frame as its payload.

The starter code will provide you with a raw Ethernet frame. Your implementation should read the ethernet header to find source and destination MAC addresses and properly dispatch the frame based on the protocol.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Destination Address                                       |
+         (48 bits)             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |          Source Address       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+             (48 bits)         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Type (16 bits)         |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               +
|                                                               |
|                                                               |
~                          Payload                              ~
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that actual Ethernet frame also includes a 32-bit Cyclical Redundancy Check (CRC). In this project, you will not need it, as it will be added automatically.

- `Type`: Payload type
  - `0x0806` (ARP)
  - `0x0800` (IPv4)

For your convenience, the starter code defines Ethernet header as an `ethernet_hdr` structure in `core/protocol.hpp`:

```
struct ethernet_hdr
{
  uint8_t  ether_dhost[ETHER_ADDR_LEN]; /* destination ethernet address */
  uint8_t  ether_shost[ETHER_ADDR_LEN]; /* source ethernet address */
  uint16_t ether_type;                  /* packet type ID */
} __attribute__ ((packed)) ;
```

Requirements

- Your router must **ignore** Ethernet frames other than ARP and IPv4.
- Your router must **ignore** Ethernet frames not destined to the router, i.e., when destination hardware address is neither the corresponding MAC address of the interface nor a broadcast address (`FF:FF:FF:FF:FF:FF`).
- Your router must **appropriately dispatch** Ethernet frames (their payload) carrying ARP and IPv4 packets.

❖ARP Packets

The Address Resolution Protocol (ARP) (RFC826) is a telecommunication protocol used for resolution of Internet layer addresses (e.g., IPv4) into link layer addresses (e.g., Ethernet). In particular, before your router can forward an IP packet to the next-hop specified in the routing table, it needs to use ARP request/reply to discover a MAC address of the next-hop. Similarly, other hosts in the network need to use ARP request/replies in order to send IP packets to your router.

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.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Hardware Type         |         Protocol Type         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  HW addr len  | Prot addr len |            Opcode             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                    Source hardware address                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                    Source protocol address                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                  Destination hardware address                 ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                  Destination protocol address                 ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- `Hardware Type`: `0x0001` (Ethernet)
- `Protocol Type`: `0x0800` (IPv4)
- `Opcode`:
  - `1` (ARP request)
  - `2` (ARP reply)
- `HW addr len`: number of octets in the specified hardware address. Ethernet has 6-octet addresses, so 0x06.
- `Prot addr len`: number of octets in the requested network address. IPv4 has 4-octet addresses, so 0x04.

For your convenience, the starter code defines the ARP header as an `arp_hdr` structure in `core/protocol.hpp`:

```c
struct arp_hdr
{
  unsigned short  arp_hrd;                  /* format of hardware address   */
  unsigned short  arp_pro;                  /* format of protocol address   */
  unsigned char   arp_hln;                  /* length of hardware address   */
  unsigned char   arp_pln;                  /* length of protocol address   */
  unsigned short  arp_op;                   /* ARP opcode (command)         */
  unsigned char   arp_sha[ETHER_ADDR_LEN]; /* sender hardware address      */
  uint32_t        arp_sip;                  /* sender IP address            */
  unsigned char   arp_tha[ETHER_ADDR_LEN]; /* target hardware address      */
  uint32_t        arp_tip;                  /* target IP address            */
} __attribute__ ((packed)) ;
```

Requirements

- Your router must properly process incoming ARP requests and replies:
  - Must properly respond to ARP requests for MAC address for the IP address of the corresponding network interface
  - Must ignore other ARP requests
- When your router receives an IP packet to be forwarded to a next-hop IP address, it should check ARP cache if it contains the corresponding MAC address:
  - If a valid entry found, the router should proceed with handling the IP packet
  - Otherwise, the router should queue the received packet and start sending ARP request to discover the IP-MAC mapping.
- When router receives an ARP reply, it should record IP-MAC mapping information in ARP cache (Source IP/Source hardware address in the ARP reply). Afterwards, the router should send out all corresponding enqueued packets.
- Your implementation should not save IP-MAC mapping based on any other messages, only from ARP replies!
- Your implementation can also record mapping from ARP requests using source IP and hardware address, but it is not required in this project.
- To reduce staleness of the ARP information, entries in ARP cache should time out after `30 seconds`. The starter code (`ArpCache` class) already includes the facility to mark ARP entries "invalid". Your task is to remove such entries.
- The router should send an ARP request about once a second until an ARP reply comes back or the request has been sent out at least `5 times`.
- If your router didn't receive ARP reply after re-transmitting an ARP request 5 times, it should stop re-transmitting, remove the pending request, and any packets that are queued for the transmission that are associated with the request.

❖IPv4 Packets

Internet Protocol version 4 (IPv4) (RFC 791) is the dominant communication protocol for relaying datagrams across network boundaries. Its routing function enables internetworking, and essentially establishes the Internet. IP has the task of delivering packets from the source host to the destination host solely based on the IP addresses in the packet headers. For this purpose, IP defines packet structures that encapsulate the data to be delivered. It also defines addressing methods that are used to label the datagram with source and destination information.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
~                          Payload                              ~
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

For your convenience, the starter code defines the IPv4 header as an `ip_hdr` structure in `core/protocol.hpp`:

```c
struct ip_hdr
{
  unsigned int ip_hl:4;              /* header length */
  unsigned int ip_v:4;               /* version */
  uint8_t ip_tos;                    /* type of service */
  uint16_t ip_len;                   /* total length */
  uint16_t ip_id;                    /* identification */
  uint16_t ip_off;                   /* fragment offset field */
  uint8_t ip_ttl;                    /* time to live */
  uint8_t ip_p;                      /* protocol */
  uint16_t ip_sum;                   /* checksum */
  uint32_t ip_src, ip_dst;           /* source and dest address */
} __attribute__ ((packed));
```

Requirements

- For each incoming IPv4 packet, your router should verify its checksum and the minimum length of an IP packet
  - Invalid packets must be discarded.
- Your router should classify datagrams into (1) destined to the router (to one of the IP addresses of the router), and (2) datagrams to be forwarded:
  - For (1), it should be discarded in this project.
  - For (2), your router should use the longest prefix match algorithm to find a next-hop IP address in the routing table and attempt to forward it there
- For each forwarded IPv4 packet, your router should correctly decrement TTL and recompute the checksum.

❖Environment Setup

The best way to guarantee full credit for the project is to do project development using a Ubuntu 16.04-based virtual machine.

You can easily create an image in your favourite virtualization engine (VirtualBox, VMware) using the Vagrant platform and steps outlined below. We suggest using VirtualBox as it is free.

1. Set Up Vagrant and Create VM Instance

Note that all example commands are executed on the host machine (your laptop), e.g., in `Terminal.app` (or `iTerm2.app`) on macOS, `cmd` in Windows, and `console` or `xterm` on Linux. After the last step (`vagrant ssh`) you will get inside the virtual machine and can compile your code there.

- Download and install your favourite virtualization engine, e.g., VirtualBox (Note that we are using Vagrant and Vagrant only supports VirtualBox versions 4.0.x, 4.1.x, 4.2.x, 4.3.x, 5.0.x, and 5.1.x) - use this link to download
- Download and install Vagrant tools for your platform, then reboot

> Do not start VM instance manually from VirtualBox GUI, otherwise you may have various problems (connection error, connection timeout, missing packages, etc.)

2. Steps to run

1. Clone project template
   ```
   > git clone https://github.com/ksb2043/cs118_fall18_project_1
   ~/cs118-proj1
   > cd ~/cs118-proj1
   ```
2. Initialize VM
   ```
   > vagrant up
   ```
3. To establish an SSH session to the created VM, run
   ```
   > vagrant ssh
   ```
4. In this project you will need to open at least two SSH sessions to the VM: to run Mininet to emulate topology and to run commands on emulated nodes, and to run your router implementation.

Notes

- If you want to open another SSH session, just open another terminal and run `vagrant ssh` (or create a new Putty session).
- If you are using Windows, read [this article](#) to help yourself set up the environment.
- If you are using Putty on Windows platform, `vagrant ssh` will return information regarding the IP address and the port to connect to your virtual machine.
- Work on your project
- All files in `~/cs118-proj1` folder on the host machine will be automatically synchronized with `/vagrant` folder on the virtual machine. For example, to compile your code, you can run the following commands:

❖Running Your Router

To run your router, you will need to run in parallel two commands on two different terminals:

1. Mininet process that emulates network topology
2. Your router app that will run your code.

For ease of debugging, can run them simply in separate SSH sessions:

- To run Mininet network emulation process

```
> vagrant ssh #  or   vagrant ssh -- -Y
> cd /vagrant
> sudo ./run.py # must be run as superuser
  ...
mininet>
```

- To run your router

```
> vagrant ssh
> cd /vagrant
  # implement router logic // see below
> make
> ./router
```

- Note If after start of the router, you see the following message

```
 Resetting SimpleRouter with 0 ports
```

```
Interface list empty
```

You should start or restart Mininet process. The expected initial output should be:

```
Resetting SimpleRouter with 3 ports
  sw0-eth1 (192.168.2.1, f6:fc:48:40:43:af)
  sw0-eth2 (172.64.3.1, 56:be:8e:bd:91:bf)
  sw0-eth3 (10.0.1.1, 22:69:6c:08:25:e9)
  ...
```

If your router is functioning correctly, the following operations should work:

- Transferring file from client to server(s) using the given client and server applications. Notice that outputs need to be redirected into files; otherwise, the client or server(s) may stop responding.
    - `mininet> server1 /vagrant/server 5678 /vagrant/ &`
    - `mininet> client /vagrant/client 192.168.2.2 5678 /vagrant/test.file &`

Starter Code Overview

Here is the overall structure of the starter code:

```
                    simple-router.hpp
                    +---------------+           core/protocol.hpp
                    |               |
                    | SimpleRouter  |           core/utils.hpp
                    |               |
                    +---+-----+-----+
         m_arp        1 | 1|   | 1       m_ifaces
       +----------------+  |  +------------------+
       |                   |                     |
       |                   |m_routingTable       |
       |                   |                     |
       |1                  |1                    | N (std::set)
       v                   v                     v
+------+------+     +-------+--------+      +-----+-----+
|             |     |                |      |           |
|  ArpCache   |     | RoutingTable   |      | Interface |
|             |     |                |      |           |
+-------------+     +----------------+      +-----------+
 arp-cache.hpp      routing-table.hpp     core/interface.hpp
```

❖Key Classes

- **`SimpleRouter`**
  Main class for your simple router, encapsulating `ArpCache`, `RoutingTable`, and as set of `Interface` objects.
- **`Interface`**
  Class containing information about router's interface, including router interface name (`name`), hardware address (`addr`), and IPv4 address (`ip`).
- **`RoutingTable` (`routing-table.hpp|cpp`)**
  Class implementing a simple routing table for your router. The content is automatically loaded from a text file with default filename is `RTABLE` (name can be changed using `RoutingTable` option in `router.config` config file)
- **`ArpCache` (`arp-cache.hpp|cpp`)**
  Class for handling ARP entries and pending ARP requests.

❖Key Methods

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 this functionality are:

- Need to implement **Method that receives a raw Ethernet frame (`simple-router.hpp|cpp`):**

```
/**
 * This method is called each time the router receives a packet on
 * the interface.  The packet buffer \p packet and the receiving
 * interface \p inIface are passed in as parameters.
 */
void
SimpleRouter::handlePacket(const Buffer& packet, const std::string& inIface);
```

`Pseudocode for handlePacket()`

1. Find input network interface using already given function findIfaceByName. Drop packet if interface is unknown
2. Read ethernet header and check the eth_type field. Ignore Ethernet frames other than ethernet type ARP and IPv4.
3. If eth_type is ARP, check ARP operation (ARP request/ARP response):
   a. If ARP Request packet:
      - Prepare an ARP response packet from the scratch and send back the packet
   b. If ARP Response packet:
      - record IP-MAC mapping information in ARP cache
      - send out all corresponding enqueued packets for the ARP entry corresponding to the ARP response (decrement TTL and recalculate checksum for the enqueued packets before sending)
4. If eth_type is IPv4 packet to be forwarded to a next-hop IP address:
   a. verify its checksum and the minimum length of an IP packet, discard invalid packets
   b. if packet is destined to the router (to one of the IP addresses of the router) - it should be discarded in this project
   c. for packets to be forwarded:
      i. use the Longest Prefix Match algorithm to find a next-hop IP address in the routing table

ii. check ARP cache if it has a MAC address mapped to the destination IP address for next-hop.
    1. If a valid entry is found: proceed with handling the IP packet.
    2. Else: the router should queue the received packet and send ARP request to discover the IP-MAC mapping.

- Implemented **Method to send raw Ethernet frames (`simple-router.hpp|cpp`):**

```
/**
 * Call this method to send packet \p packt from the router on interface \p outIface
 */
void
SimpleRouter::sendPacket(const Buffer& packet, const std::string& outIface);
```

- Need to implement **Method to handle ARP cache events (`arp-cache.hpp|cpp`):**

```
/**
 * This method gets called every second. For each request sent out,
 * you should keep checking whether to resend a request or remove it.
 */
void
ArpCache::periodicCheckArpRequestsAndCacheEntries();
```

- Implemented **Method to calculate the IP checksum of given data (`core/utils.hpp|cpp`):**

```
/*Calculates the IP Checksum via IPv4 specs*/
uint16_t
cksum(const void* _data, int len)
```

- Need to implement **Method to lookup entry in the routing table (`routing-table.hpp|cpp`):**

```
/**
 * This method should lookup a proper entry in the routing table
 * using "longest-prefix match" algorithm
 */
RoutingTableEntry
RoutingTable::lookup(uint32_t ip) const;
```

❖Debugging Functions

We have provided you with some basic debugging functions in `core/utils.hpp` (`core/utils.cpp`). Feel free to use them to print out network header information from your packets. They will print to the router ssh connection. Below are some functions you may find useful:

- `print_hdrs(const uint8_t *buf, uint32_t length)`, `print_hdrs(const Buffer& packet)`
  Print out all possible headers starting from the Ethernet header in the packet

- `ipToString(uint32_t ip)`, `ipToString(const in_addr& address)`
  Print out a formatted IP address from a `uint32_t` or `in_addr`. Make sure you are passing the IP address in the correct byte ordering

❖A Few Hints

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

- Sanity-check the packet (meets minimum length and has correct checksum).
- Decrement the TTL by 1, and recompute the packet checksum over the modified header.
- Find out which entry in the routing table has the longest prefix match with the destination IP address.
- 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.

If an incoming IP packet is destined towards one of your router's IP addresses, you should discard received packet.

❖Submission Requirements

To submit your project, you need to prepare:

1. A `README.md` file placed in your code that includes:
    - Name and UID
    - The high level design of your implementation
    - The problems you ran into and how you solved the problems
2. All your source code, `Makefile`, `README.md`, and `Vagrantfile`.
3. To create the submission, **use the provided Makefile** in the starter code. Just update `Makefile` to include your UCLA ID and then just type
4. `make tarball`
5. Then submit the resulting archive on CCLE project submission page.

Before submission, please make sure:

1. Your code compiles
2. Your implementation conforms to the specification
3. `.tar.gz` archive does not contain temporary or other unnecessary files. We will deduct points otherwise.

> Submissions that do not follow these requirements will not get any credit.

❖Grading

Grading Criteria

- Transfer a file from the client to the server1
- Transfer a file from the client to the server2

- Update ARP cache correctly
- Update IP header (Checksum and TTL) correctly

Deductions

(-5 pts) The submission archive contains temporary or other non-source code file, except `README.md`:q and `Vagrantfile`.

❖Acknowledgement

This project is based on the CS144 class project by Professor Philip Levis and Professor Nick McKeown, Stanford University.