[CS118: Computer Network Fundamentals - Fall 2022 (UCLA)]

Project 3: 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 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.

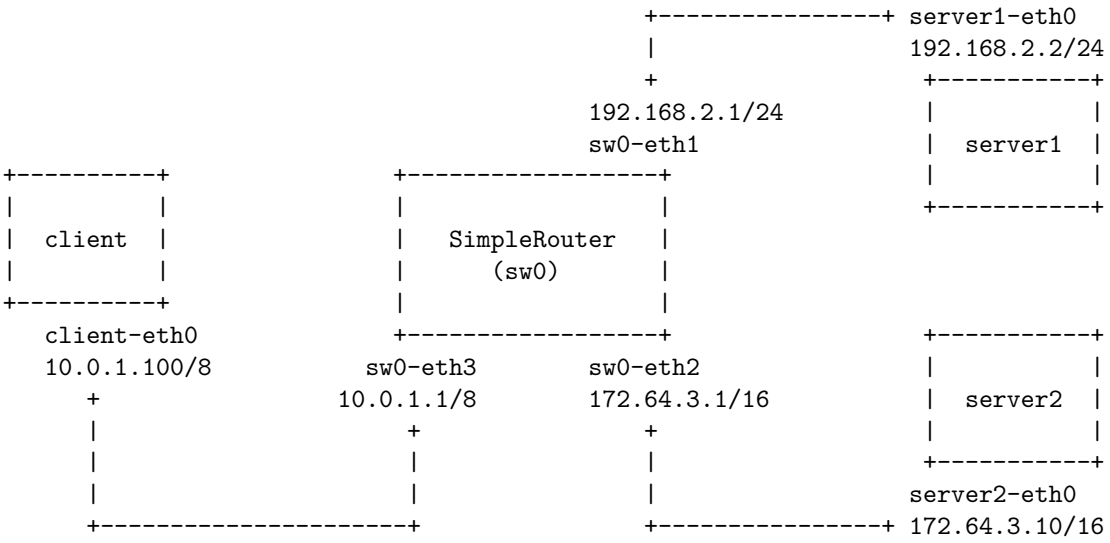
Task Description

There are four main parts in this assignment:

- Handle Ethernet frames
- Handle ARP packets
- Handle IPv4 packets
- Implement an Access Control List (ACL) in your router

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:



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

Destination	Gateway	Mask	Iface
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. We will be testing your code on other single-router topologies with different number of servers and clients, and different IP and network addresses.

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

- `ping` from the client to any of the router's interfaces:

  ```
  mininet> client ping 192.168.2.1
  ...
  ```

```
mininet> client ping 172.64.3.1
...
mininet> client ping 10.0.1.1
...
```

- `ping` from the client to any of the app servers:

```
mininet> client ping server1  # or client ping 192.168.2.2
...
mininet> client ping server2  # or client ping 172.64.3.10
...
```

- `traceroute` from the client to any of the router's interfaces:

```
mininet> client traceroute 192.168.2.1
...
mininet> client traceroute 172.64.3.1
...
mininet> client traceroute 10.0.1.1
...
```

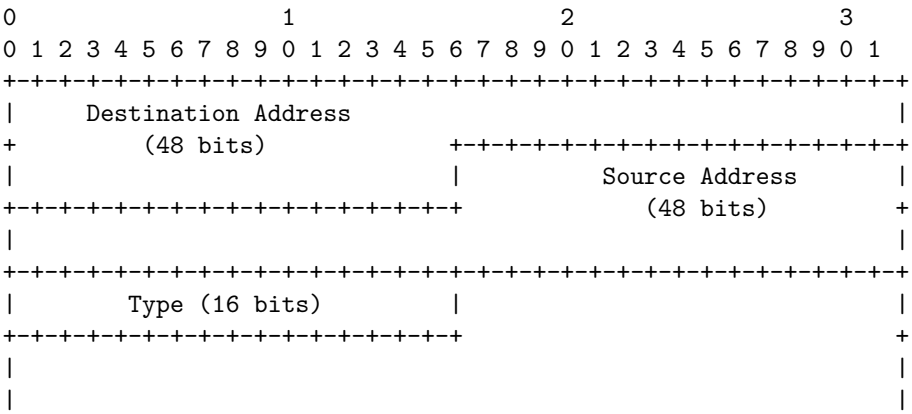
- Tracerouting from the client to any of the app servers:

```
mininet> client traceroute server1
...
mininet> client traceroute server2
...
```

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 understand source and destination MAC addresses and properly dispatch the frame based on the protocol.



```
~                              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

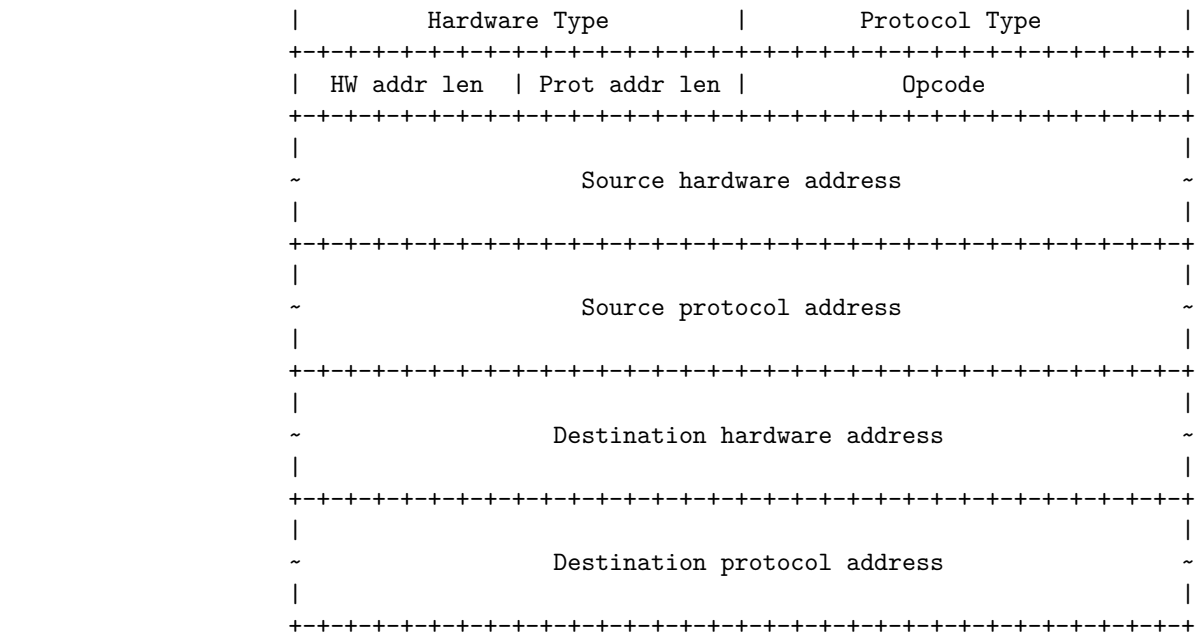
- Your router should ignore Ethernet frames other than ARP and IPv4.
- Your router must ignore Ethernet frames not destined to the router, i.e., when destination hardware address is neither the corresponding MAC address of the interface nor a broadcast address (`FF:FF:FF:FF:FF:FF`).
- Your router must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

```
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.

  Note Your implementation should not save IP-MAC mapping based on any other messages, only from ARP replies!

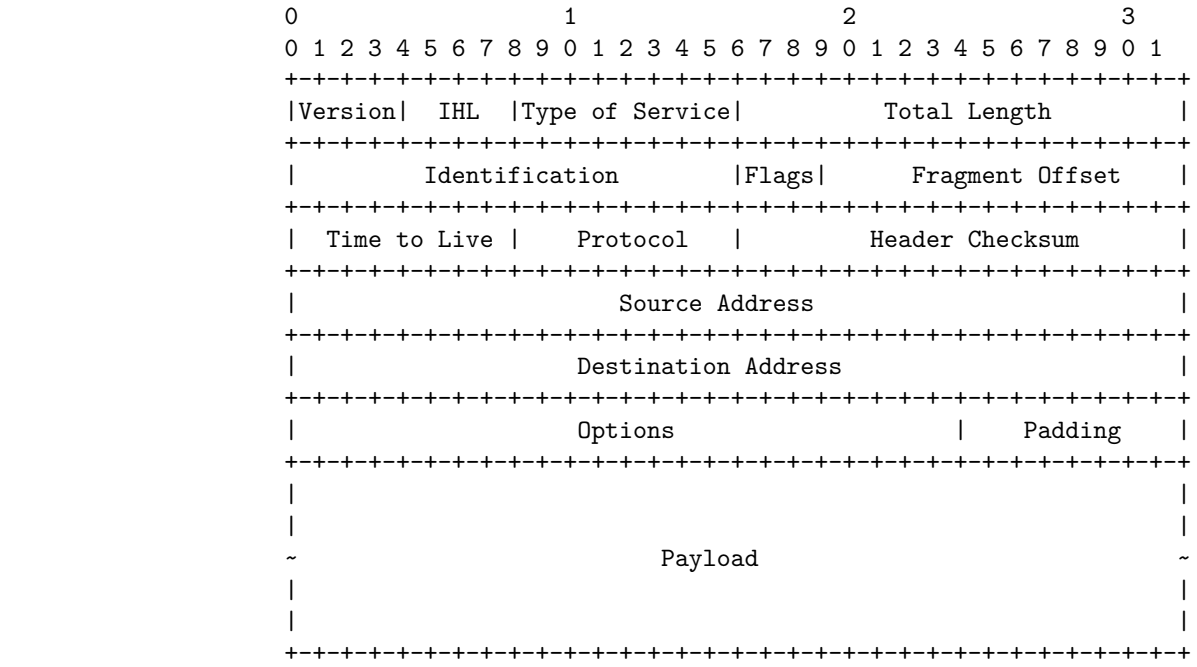
  Your implementation can also record mapping from ARP requests using source IP and hardware address, but it is not required in this project.
- To reduce staleness of the ARP information, entries in ARP cache should time out after `30 seconds`. The starter code (`ArpCache` class) already includes the facility to mark ARP entries "invalid". Your task is to remove such entries. If there is an ongoing traffic (e.g., client still pinging the server), then the router should go through the standard mechanism to send ARP request and then cache the response. If there is no ongoing traffic, then ARP cache should eventually become empty.
- 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.



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

```
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), packets should be discarded.
  - 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.

Access Control List

An Access Control List (ACL) is a list of permissions that defines what actions a router should take when it receives packets destined for specific IP addresses. For example, if could say that packets destined for some addresses should be dropped while packets destined for a second address should be forwarded. Your router must implement this functionality. The ACL for your router is defined in the file `ATABLE`. The contents of this file is as follows.:

```
c0a80202&ffff0000 ac40030a&ffff0000 0&0 0&0 0&0 1 deny
c0a80202&ffff0000 ac40030a&ffff0000 0&0 0&0 1&ff 2 permit
```

- The first field corresponds to source IP address and its mask
- The second field corresponds to destination IP address and its mask
- The third and fourth fields correspond to source and destination ports and their masks
- The fifth field corresponds to the IP protocol number field and its mask
- The sixth field specifies the priority of the rule
- The final field specifies the action to be taken when a packet matches one of the rules in the ACL. “Deny” means the packet should be dropped and “Permit” means it should be forwarded as normal

Requirements

- For each incoming packet, your router should check if any of the rules in the ACL apply to the packet (eg. check source/destination IP address, port, protocol, etc. against each entry in the ACL) and then perform the action described by the matched rule with the highest priority, if any exist. If no rules apply to the packet, then your router should just forward the packet as usual.

- When your router decides to drop a packet because of an ACL rule, it should log the ACL rule entry that causes the drop in file "./router-acl.log".
- Sample ACL Log:

```
c0a80202&ffff0000   ac40030a&ffff0000   0&0 0&0 1   2   deny
c0a80202&ffff0000   ac40030a&ffff0000   0&0 0&0 1   2   deny
c0a80202&ffff0000   ac40030a&ffff0000   0&0 0&0 1   2   deny
```

Environment Setup

Initial Setup

To run any of the code in this project, you will need a computer running Linux. We have specifically tested the code on Ubuntu 22.04, but it is likely to work on most Linux distributions, provided you install the dependencies correctly. If your machine does not run Linux natively, you have two options.

1. Use a program such as VirtualBox to create a linux VM. NOTE: The course staff are running VirtualBox version 6.1. Version 7.0 is new and might not work for this assignment.
2. Create a GCP instance using the credits provided to you by the course staff. Email Henry or Changrong if you would like to use this option and they can give you a code which you can redeem to add credits to your Google account.

After you have access to your Linux VM, you will need to download and install Docker. All the code for this assignment will run inside a Docker container that you build.

Once you have installed Docker, first download the starter code from BruinLearn and unzip.

Then, `cd` into the project directory and run the following command to build your Docker container. You can name your container whatever you'd like. The command will create a container with all the necessary dependencies for this project as specified in the `Dockerfile` located in the project's root directory.

```
sudo docker build -t <name_of_your_container> .
```

Finally, you will need to open a terminal window in your Docker container to run any necessary code.

```
sudo docker run -it --privileged <name_of_your_container> bash
```

Now you should be able to run the commands in Running Your Router without any issues.

It is worth noting that after making edits to any code on your local machine (eg. not inside the container), you will need to copy the files into the container using the Docker cp command.

Running Your Router

To run your router, you will need to run in parallel two commands: Mininet process that emulates network topology and your router app. For ease of debugging, can run them in `screen` (or `tmux`) environments or simply in separate SSH sessions:

- Make sure openvswitch is running in your container. If it is not running, execute the following command:

  ```
  service openvswitch-switch start
  ```

- To run Mininet network emulation process

  ```
  ./run.py
  ...
  mininet>
  ```

- Before you run your router, you need to run a pox script at the background using the following command:

  ```
  python2.7 -u /opt/pox/pox.py --verbose ucla_cs118 &
  ```

- To run your router

  ```
  #
  # 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
  ```

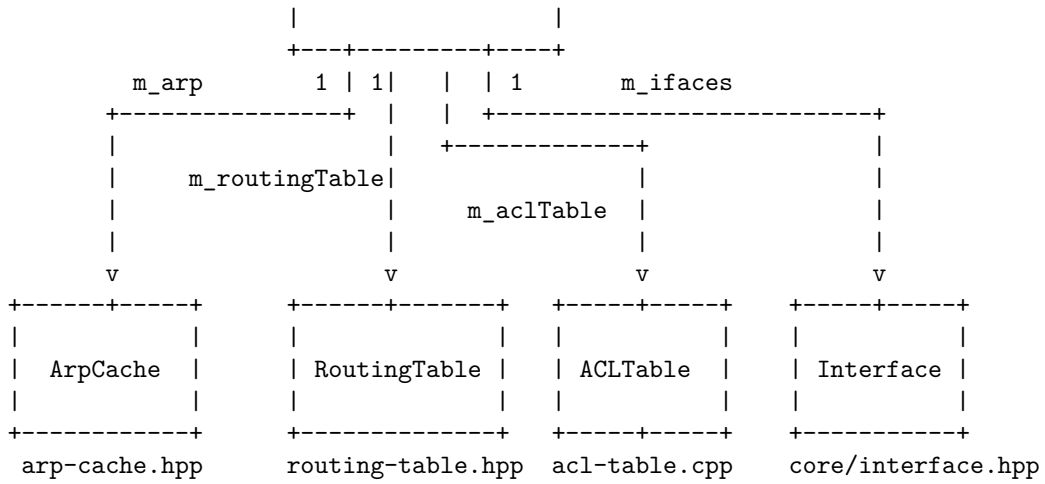
  You should start or restart Mininet process. The expected initial output should be:

  ```
  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)
  ...
  ```

Starter Code Overview

Here is the overal structure of the starter code:

```
simple-router.hpp
+------------------+           core/protocol.hpp
|                  |
| SimpleRouter     |           core/utils.hpp
```



- `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 these functions 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);
```

- 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();
```

- 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. 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

Logging Packets

You can use Mininet to monitor traffic that goes in and out of the emulated nodes, i.e., router, server1 and server2. For example, to see the packets in and

out of `server1` node, use the following command in Mininet command-line interface (CLI):

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

Alternatively, you can bring up a terminal inside `server1` using the following command

```
mininet> xterm server1
```

then inside the newly opened `xterm`:

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

Length of Assignment

This assignment is considerably hard, so get started early, not as many of you did for project 2.

In our reference solution, we added less than 520 lines of C/C++ code, including whitespace and comments. Of course, your solution may need fewer or more lines of code, but this gives you a rough idea of the size of the assignment to a first approximation.

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, ignore the packet.

Obviously, this is a very simplified version of the forwarding process, and the low-level details follow. For example, you may also get an ARP request or reply, which has to interact with the ARP cache correctly.

Submission Requirements

To submit your project, you need to prepare:

1. A `README.md` file placed in your code that includes:
   - Your name and UID
   - List of any additional libraries used
   - Acknowledgement of any online tutorials or code example you have been using.
2. All your source code, `Makefile` and `README.md` as a `.tar.gz` archive.

   To create the submission, **use the provided Makefile** in the starter code. Just update `Makefile` to include your UCLA ID and then just type

   ```
   make tarball
   ```

   Then submit the resulting archive to Gradescope.

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 automatically deduct points otherwise.

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

Grading

Grading Criteria

1. Ping tests
   - 1.1. (5 pts,) Pings from client to all other hosts (all pings expected to succeed), including non-existing host (error expected)
   - 1.2. (5 pts) Pings from server1 to all other hosts (all pings expected to succeed), including non-existing host (error expected)
   - 1.3. (5 pts) Pings from server2 to all other hosts (all pings expected to succeed), including non-existing host (error expected)
   - 1.4. (5 pts) Ping responses (from client) have proper TTLs
   - 1.5. (5 pts) Ping from client to server1, check ARP cache, there should be two entries
   - 1.6. (10 pts) Ping from client to server1, after 40 seconds, the ARP cache should be empty (+ no segfaults)
   - 1.7. (10 pts) Ping from client a non-existing IP, router sends proper ARP requests (+ no segfaults)
   - 1.8. (3 pts) Ping from client, receive host unreachable message
2. Traceroute tests

- 2.1. (10 pts) Traceroute from client to all other hosts, including a non-existing host
- 2.2. (10 pts) Traceroute from server1 to all other hosts, including a non-existing host
- 2.3. (10 pts) Traceroute from server2 to all other hosts, including a non-existing host
- 2.4. (2 pts) Traceroute from client to router's interfaces (get 1 line)

3. ACL Tests
    - 3.1. (10 pts) Pings from client to serverX where packet is allowed to be forwarded according to ACL. Check logs to see if correct ACL rule has been applied.
    - 3.2 (5 pts) Pings from client to serverX where packet is NOT allowed to be forwarded according to ACL. Check logs to see if correct ACL rule has been applied

Note that the router should work in other single-router network topologies / with different routing tables

Acknowledgement

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