44100113-0: COMPUTER NETWORKS
FALL 2025

Build Your Own Router

Assigned: November 26, 2025
Due Date: 23:59, December 21, 2025

1 ASSIGNMENT

You must work on this assignment individually.

This project is designed for you to:

- Learn to build a simple router.
- To be familiar with IPv4, ICMP and ARP.

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.

In the following contents, you will see

- Detailed description of this project in Section 2
- Guidelines to setup the environment in Section 3
- Overview of the starter code in Section 4
- Some helpful hints in Section 5
- How to submit your code in Section 6
- Grading criteria in Section 7

This assignment is considerably hard, so get started early, not as some of you did for FTP Project, to avoid missing the deadline.

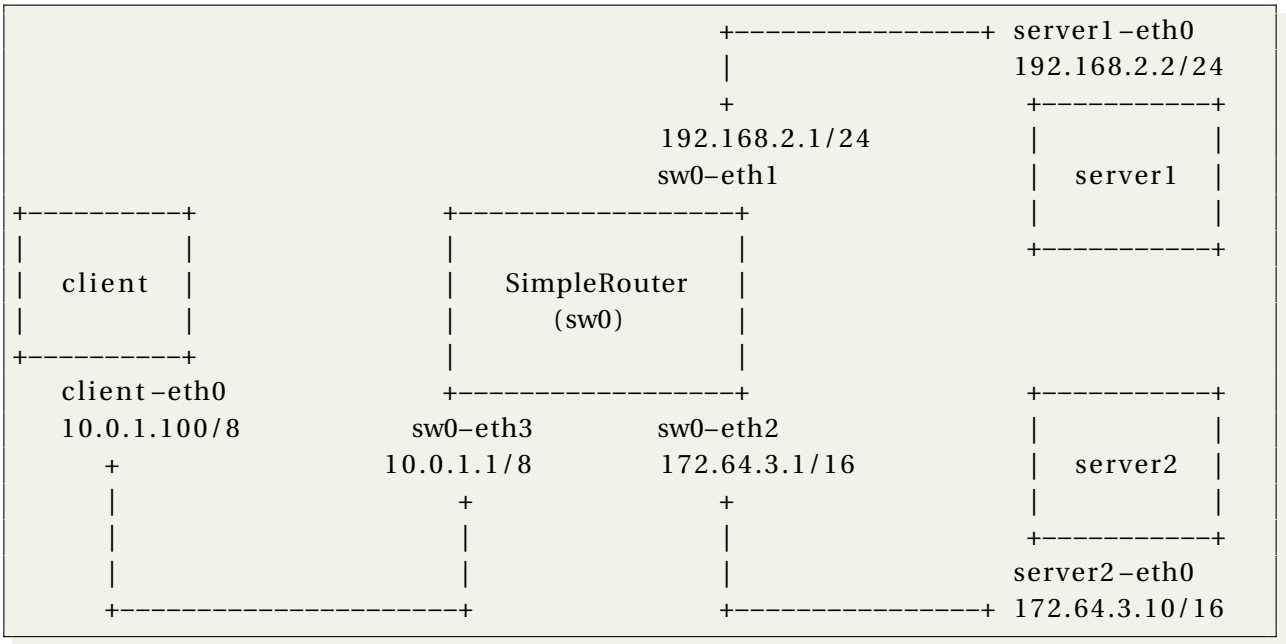
2 Project Description

There are four main parts in this assignment:

- Handle Ethernet frames
- Handle ARP packets
- Handle IPv4 packets
- Handle ICMP packets

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 (`http://mininet.org/`).

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	0.0.0.0	255.255.255.0	sw0-eth1
172.64.3.10	0.0.0.0	255.255.0.0	sw0-eth2

NOTE: 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. So it is STRONGLY RECOMMENDED that you test your implementations under different IP addresses and routing tables (by modifying the `IP_CONFIG` and `RTABLE` file respectively).

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
...
```

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

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

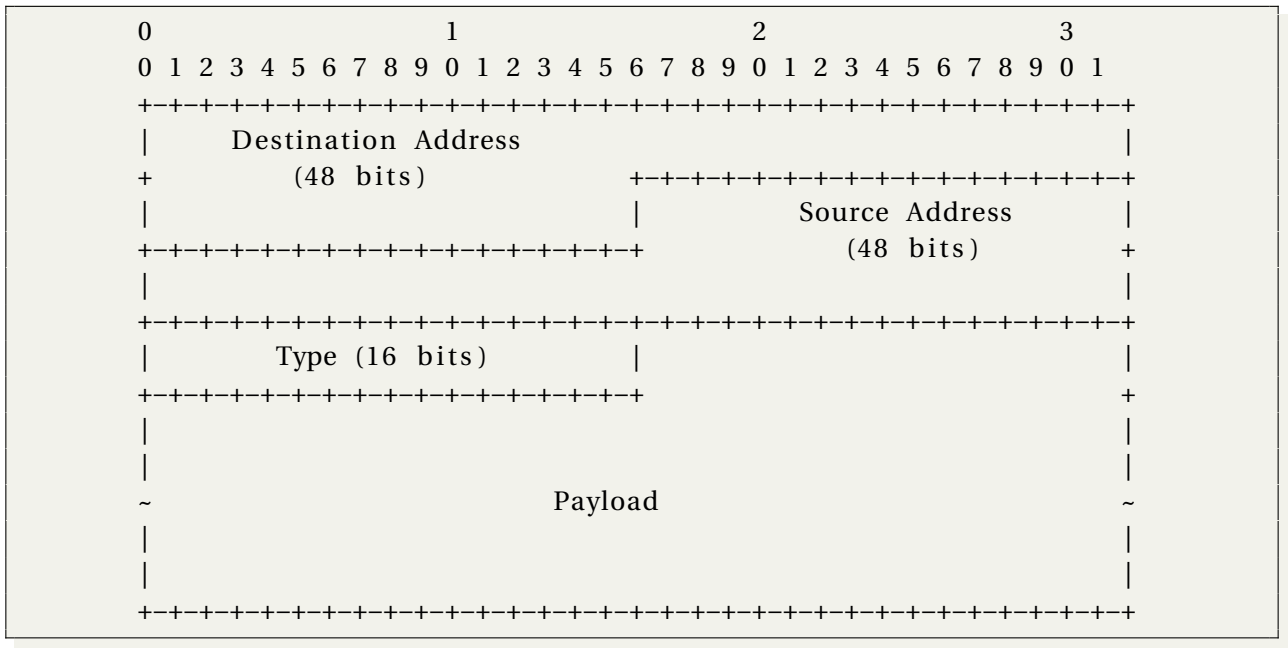
- Downloading files from http-server by `wget` command

```
mininet> client wget http://192.168.2.2/<filename>
...
mininet> client wget http://172.64.3.10/<filename>
...
```

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



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 types
  - `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

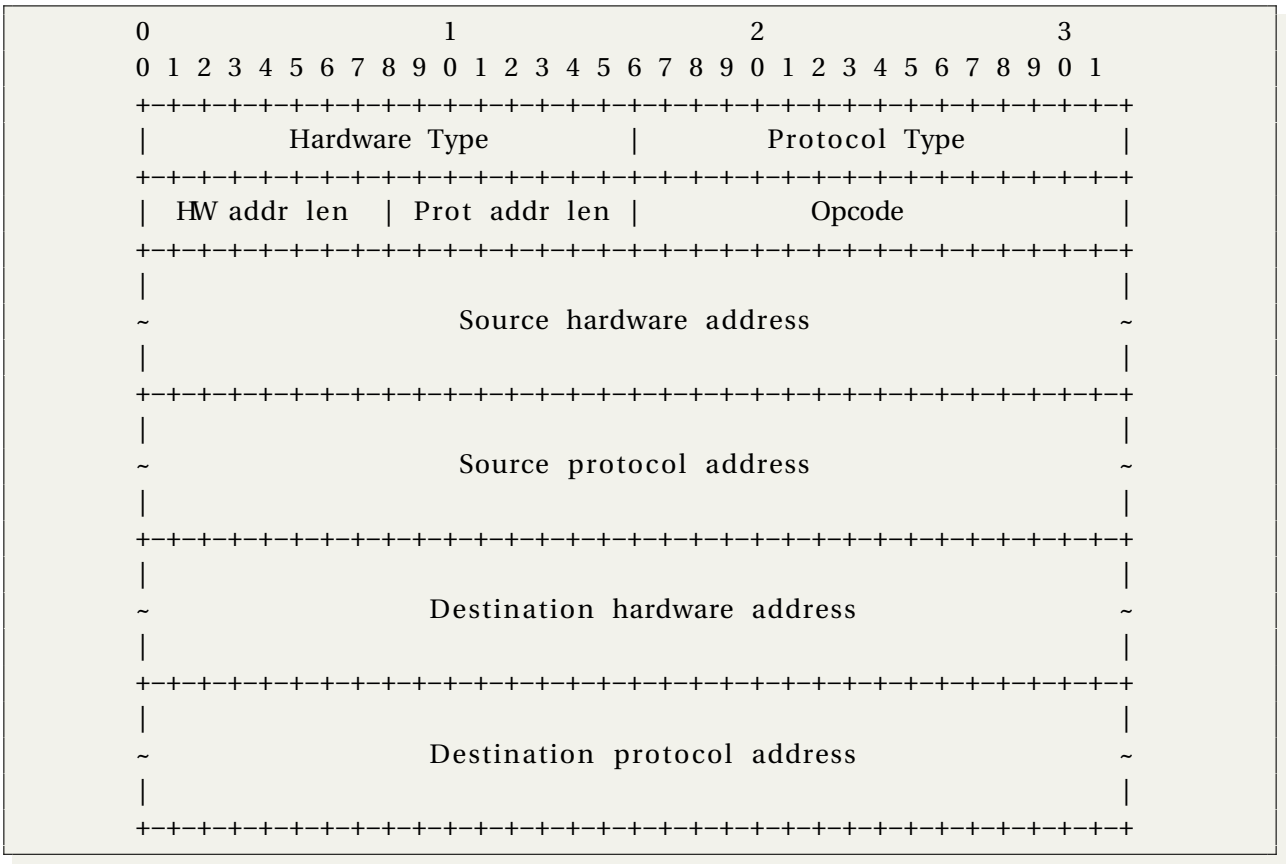
- Your router should ignore Ethernet frames other than ARP and IPv4.
- Your router must ignore Ethernet frames not destined to the router, i.e., when destination hardware address is neither the corresponding MAC address of the interface nor a broadcast address (`FF:FF:FF:FF:FF:FF`).
- Your router must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

2.2 ARP Packets

The Address Resolution Protocol (ARP) (RFC 826) 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.



- `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.

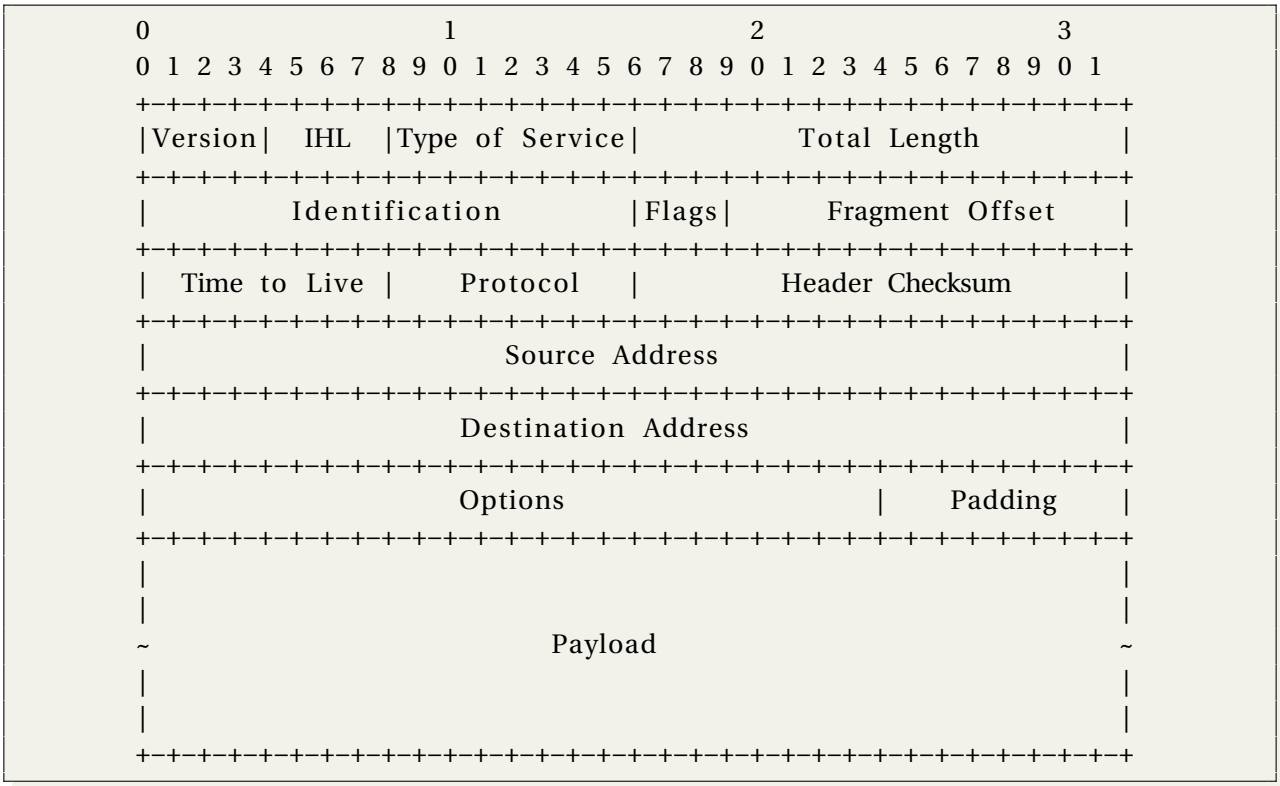
  NOTE: Your implementation should not save IP-MAC mapping based on any other messages, only from ARP replies!
- To reduce staleness of the ARP information, entries in ARP cache should time out after `30 seconds`. The starter code (`ArpCache` class) already includes the facility to mark ARP entries "invalid". 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.

  Your router should also send an ICMP Destination `Host Unreachable` message to the source IP.

2.3 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.

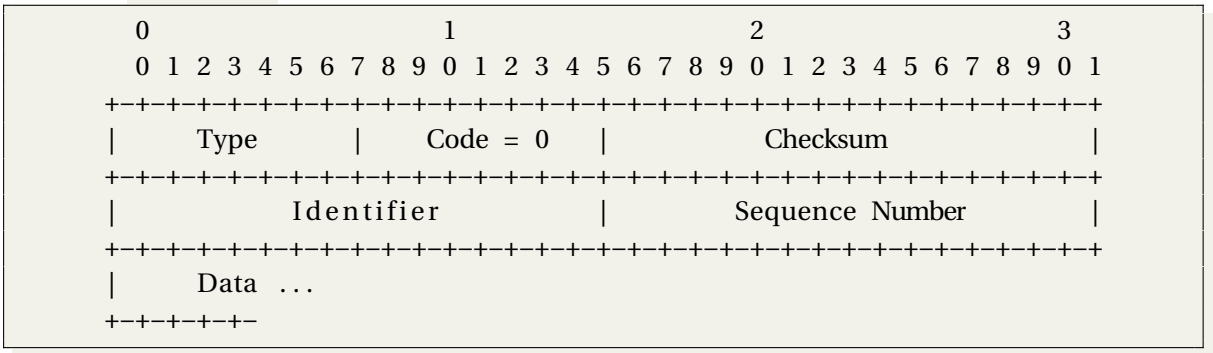
- Your router should classify datagrams into (1) destined to the router (to one of the IP addresses of the router), and (2) datagrams to be forwarded:
  - For (1), if packet carries ICMP payload, it should be properly dispatched. Otherwise, discarded (a proper ICMP error response is NOT required for 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.

2.4 ICMP PACKETS

Internet Control Message Protocol (ICMP) (RFC 792) is a simple protocol that can send control information to a host.

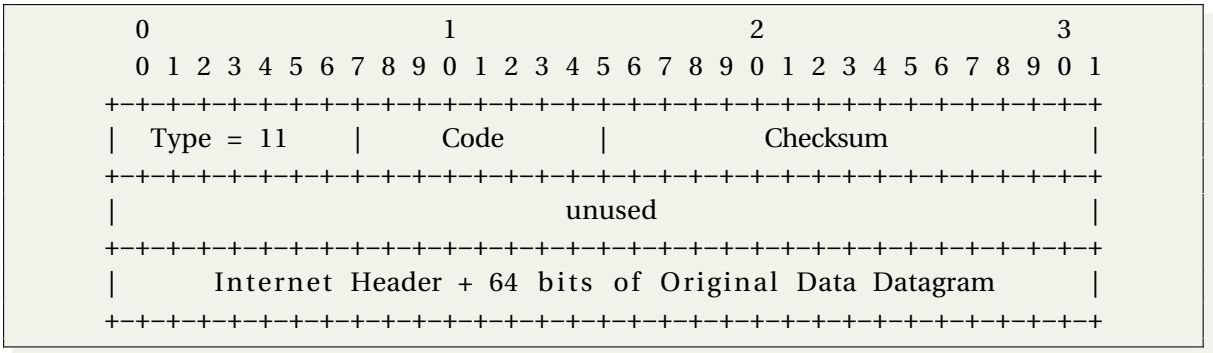
In this assignment, your router will use ICMP to send messages back to a sending host.

- `Echo` or `Echo Reply` message



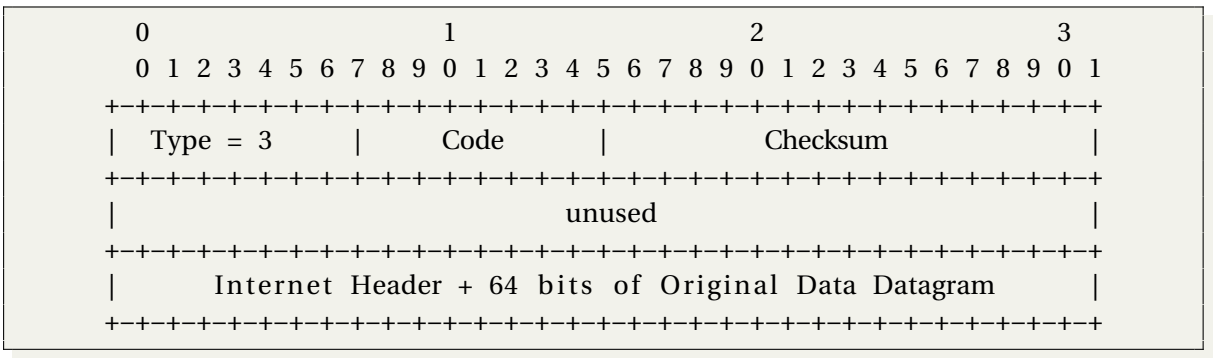
  - `Type`
    - `8` : echo message
    - `0` : echo reply message

- `Time Exceeded` message



  - `Code`
    - `0` : time to live exceeded in transit
    - `1` : fragment reassembly time exceeded (NOT required to implement)

- `Destination Unreachable` message



  - `Code`
    - `1` : Destination host unreachable
    - `3` : Destination port unreachable

When an ICMP message is composed by a router, the source address field of the internet header can be the IP address of any of the router's interfaces, as specified in RFC 792.

Note that `Time Exceeded` message is needed for `traceroute` to work properly.

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

```
struct icmp_hdr {
  uint8_t icmp_type;
  uint8_t icmp_code;
  uint16_t icmp_sum;
} __attribute__ ((packed));
```

You may want to create additional structs for ICMP messages, 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 (i.e., must use `__attribute__ ((packed))` annotation).

Requirements

Your router should properly generate the following ICMP messages, including proper ICMP header checksums:

- `Echo Reply` message (`type 0`):

  Sent in response to an incoming `Echo Request` message (ping) to one of the router's interfaces. Echo requests sent to other IP addresses should be forwarded to the next hop address as usual.

  In this project, `Echo Reply` message's initial TTL field in its IPv4 header should be `64` .

- `Time Exceeded` message (`type 11` , `code 0`):

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

- `Port Unreachable` message (`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.

3 Environment Setup

You are suggested to finish this project on Ubuntu 16.04.7. If your system is Windows or Mac OS, you can install Ubuntu 16.04.7 Virtual Machine in VMware. (Do not use Windows Subsystem for Linux (WSL), as Mininet is currently not supported by WSL.)

We provide `setup.sh` to setup the environment. Please run it in superuser mode:

```
sudo bash setup.sh
```

To run or test your code, please change work path to the project directory and open three terminals here, then

1. in the first terminal, start pox

   ```
   /opt/pox/pox.py --verbose ucla_cs118
   ```

2. in the second terminal, start Mininet

   ```
   chmod +x run.py
   sudo ./run.py
   ```

   To exit Mininet, type *exit* command in this terminal.

3. in the third terminal, start your router

   ```
   make
   ./router
   ```

Besides, you can use the `show-arp.py` script to print out your router's current arp cache or routing table in another terminal.

- To show arp cahce, run:

  ```
  ./show-arp.py arp
  ```

- To show routing table, run:

  ```
  ./show-arp.py routing
  ```

Here are some tips:

- DO NOT COPY directly from the project spec, as the pdf formatting may be problematic.
- If you get the following outputs after running router

  ```
  Got packet of size 42 on interface sw0-eth1
  Received packet, but interface is unknown, ignoring
  ```

  or

  ```
  Resetting SimpleRouter with 0 ports
    Interface list empty
  ```

Please restart Mininet.

- If you get the following outputs after running `run.py`

```
/usr/bin/env: 'python\r': No such file or directory
```

Maybe converting the format of `run.py` is helpful

```
sudo apt-get install dos2unix
dos2unix run.py
```

- When POX controller is restrated, the simpler router needs to be manually stopped and started again.
- If you get the following outputs after running `run.py` or `autograde.py`

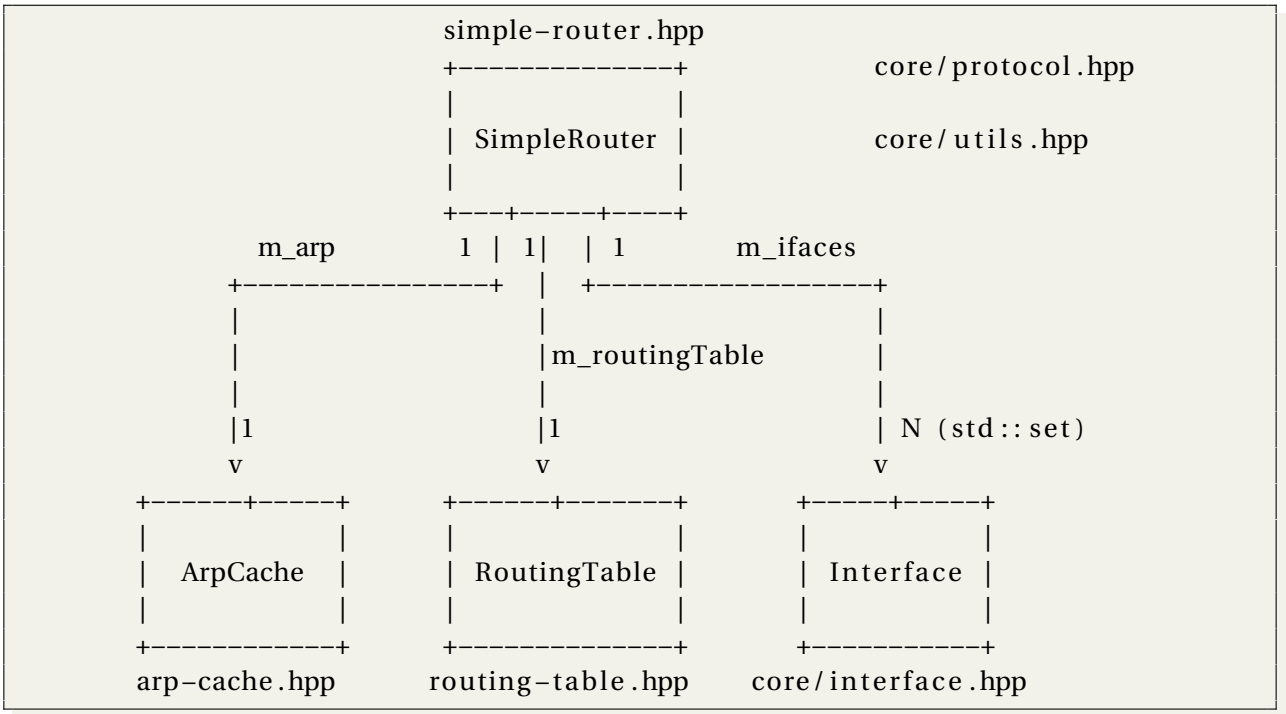
```
Exception: Error creating interface ... RTNETLINK answers: File exists
```

Try cleaning up mininet related environments by

```
sudo mn -c
```

4 Starter Code Overview

Here is the overal structure of the starter code:



- `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.

4.1 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;
```

4.2 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

- `macToString(const Buffer& macAddr)`

  Print out a formatted MAC address from a `Buffer` of MAC address

4.3 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
```

4.4 Grading Script

To ease debugging, we make a simplified version of grading script public with `autograde.py`. It contains all the public test cases (details in Section 7), which make up 45/85 of the total test score. To run the script, first start pox and your router, then use the following command:

```
chmod +x autograde.py
sudo ./autograde.py
```

The grading result and related information will be shown in standard output as well as the `details.log` file. You can also change the logging level to see more detailed information:

```
log.setLevel(logging.DEBUG)
```

If you get the following outputs from router:

```
Got packet of size 42 on interface sw0-eth1
Received packet, but interface is unknown, ignoring
```

Just turn off your router, grading script, and then start and exit Mininet with `run.py`. Afterwards, the grading script may work as expected.

Notice that the private, comprehensive grading script will be run on other single-router topology with different interfaces, IP/MAC addresses. Thus, **the output of provided script may be different from your final score, even on public test cases.**

5 Helpful 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 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.

Obviously, this is a very simplified version of the forwarding process, and the low-level details follow. 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.

In case you have difficulties, you can contact the TAs by email: xu-s25@mails.tsinghua.edu.cn, hanyuliu03@gmail.com or visit the TAs at 11-211, East Main Building.

6 Submission Requirements

To submit your project, you need to prepare:

1. A `report.pdf` file no more than THREE pages. (See details in Section 7)
2. All your source code, `Makefile` and `report.pdf` as a `.tar.gz` archive (and any files from extra credit part).

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

   ```
   make tarball
   ```

   Then submit the resulting archive to WEB LEARNING.

Before submission, please make sure:

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

7 Grading Guidelines

Your project will be graded based on the following parts (private tests are not provided in `autograde.py`):

1. Ping tests
   a) (5 pts, public) Pings from client to all other hosts (all pings expected to succeed), including non-existing host (error expected)
   b) (5 pts, public) Pings from server1 to all other hosts (all pings expected to succeed), including non-existing host (error expected)
   c) (5 pts, public) Pings from server2 to all other hosts (all pings expected to succeed), including non-existing host (error expected)
   d) (10 pts, private) Ping responses (from client) have proper TTLs
   e) (5 pts, public) Ping between selected hosts, check ARP cache, there should be a proper number of entries
   f) (5 pts, public) Ping from client to server1, after 40 seconds, the ARP cache should be empty (+ no segfaults)
   g) (5 pts, private) Ping from client a non-existing IP, router sends proper ARP requests (+ no segfaults)
   h) (5 pts, private) Ping from client, receive host unreachable message
2. Traceroute tests

a) (5 pts, public) Traceroute from client to all other hosts, including a non-existing host

b) (5 pts, public) Traceroute from server1 to all other hosts, including a non-existing host

c) (5 pts, public) Traceroute from server2 to all other hosts, including a non-existing host

d) (10 pts, private) Traceroute from client to router's interfaces (get 1 line)

3. File downloading tests

   a) (5 pts, public) Download a small file (≈ 1 KB) from any server through http

   b) (10 pts, private) Download a large file (≈ 10 MB) from any server through http

4. (20 pts) Code quality and Project Report (`report.pdf`)

   The project report could include:

   a) Your name and student ID

   b) The problems you have met in this project and how you solved them.

   c) List of any additional libraries used.

   d) Any advice on this project.

   Note that poor design, code structure, or report will probably reduce the credits you gained in this part.

8 ACKNOWLEDGEMENT

This project is based on the CS118 class project by Alexander Afanasyev, UCLA.