CS 118, Winter 2021 (UCLA): Build Your Own Router

Revisions

- Revision 2 (February 16, 2021): Correct link to Vagrant software.
- Revision 1 (February 15, 2021): Initial release.

Overview

In this project, you will need to write a simple router. Your router will receive raw Ethernet frames and process them just like a real router by forwarding them to the correct outgoing interface, performing longest-prefix matching lookups in the routing table, etc. The starter code provides a framework that receives Ethernet frames; it is your job to write the frame processing, handling, and forwarding logic. You are allowed to use some high-level abstractions, such as C++11 extensions, for parts of your code that are not directly related to networking, such as string parsing, multi-threading, etc.

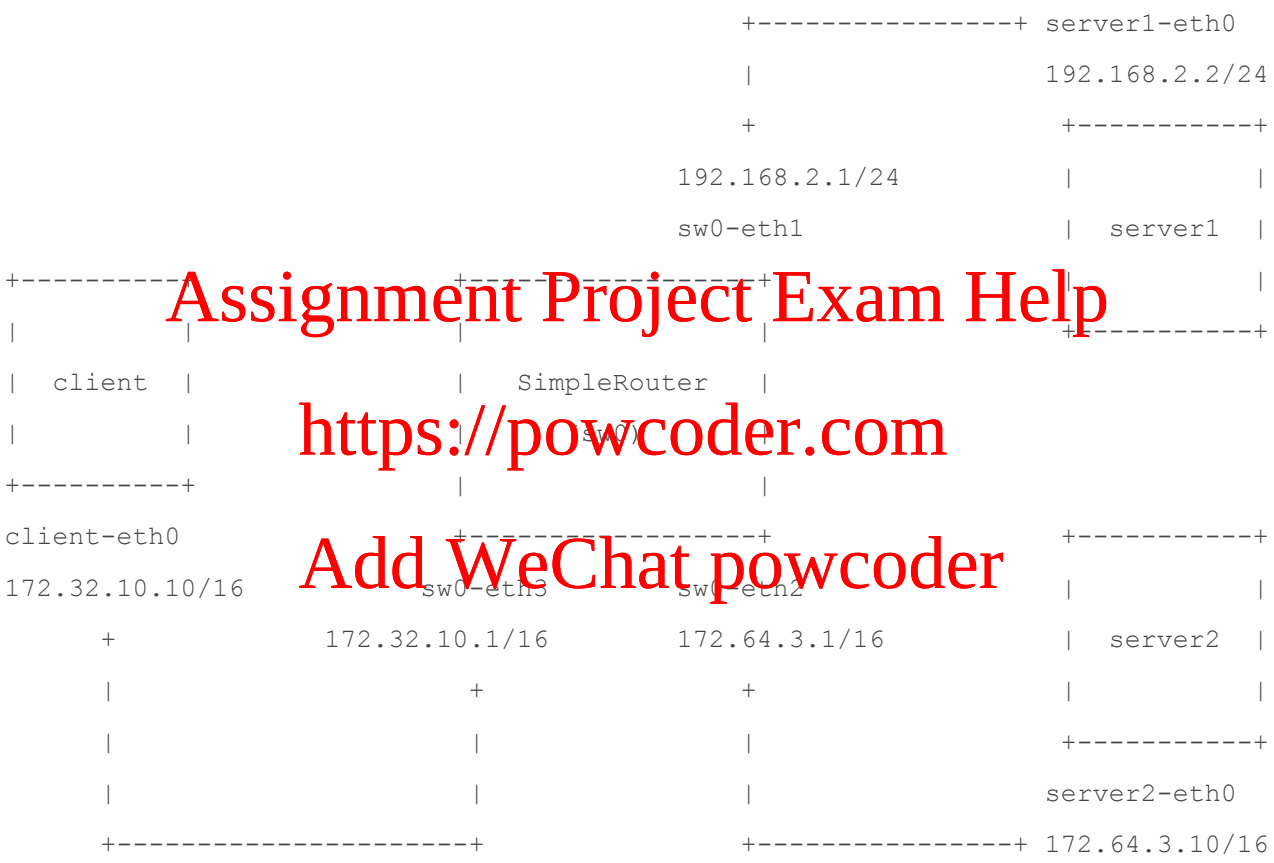
> **Each student or group of up to two students must work on the project individually, without collaboration with anyone else.**
>
> **If working on this project in a group of two people, please only submit one copy of the project per group. Please indicate in the CCLE submission notes the name of your partner, if any.**
>
> **You are encouraged to host your code in a private repository on GitHub, GitLab, or another similar site. At the same time, you are PROHIBITED from making your code for this project public either during the class or at any time thereafter.**

Project Overview

This assignment runs on top of Mininet. 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.

We have provided a virtual machine environment for your use when developing this project. This provides a standard environment for both development and grading to ensure there are no incompatibilities during grading. While it may be possible for you to develop and run your router on a different environment, we **very strongly encourage** you to use this environment. Instructions on how to set up and access this environment are provided in the Environment Setup section below.

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 is:

```
Destination  Gateway      Mask             Interface
------------ ------------ ---------------- ----------
0.0.0.0      172.32.10.10 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 in your router implementation. We will be testing your code on other single-router topologies with different number of servers and clients, and different IP and network addresses.**

Your simple router implementation is expected to:

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

If your implementation is functioning correctly, all of the following operations should succeed:

- `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 172.32.10.1
...
```

- `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
...
```

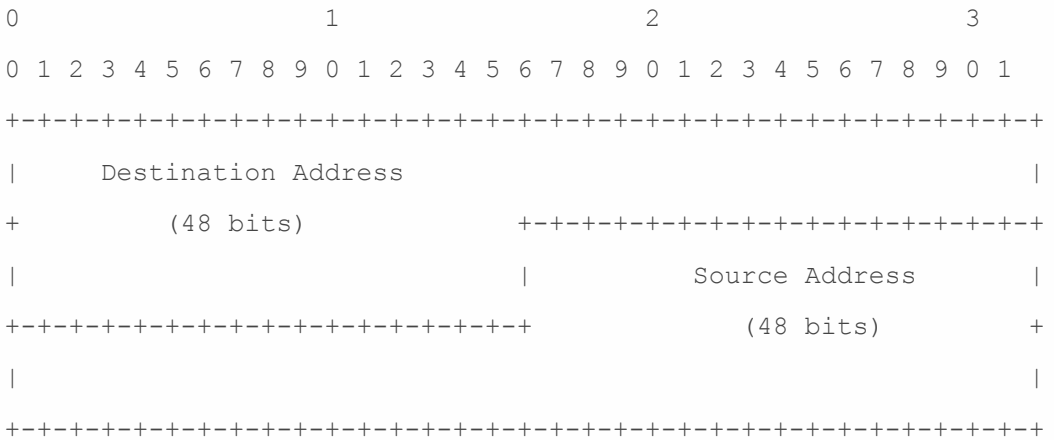
- Transferring a file from the client to one of the server(s) using the provided `client` and `server` applications.

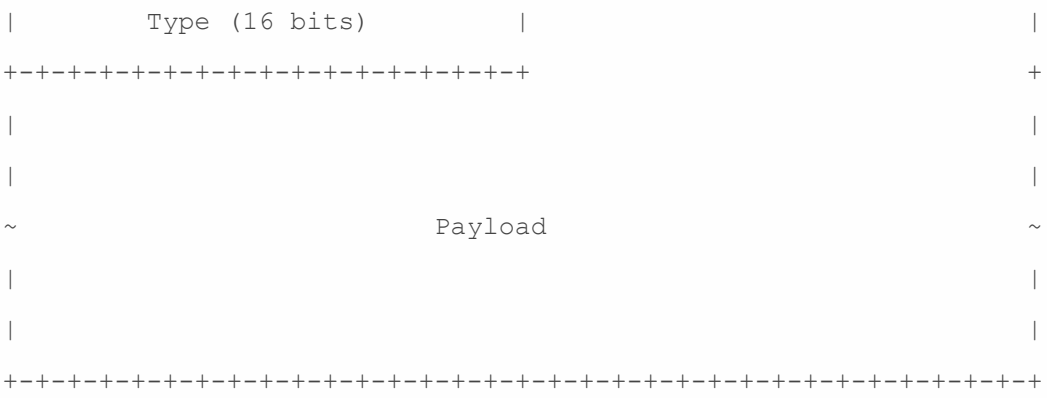
```
mininet> server1 /vagrant/server 5678 /vagrant &
...
mininet> client /vagrant/client 192.168.2.2 5678 /vagrant/test_small.file &
...
```


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 extract the source and destination MAC addresses, as well as as the Ethernet payload type, and properly dispatch the frame to the protocol indicated by the Ethernet payload type.





Note that the 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. Your router will need to support the following payload types:

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

For your convenience, the starter code defines the `ethernet_hdr` struct in `core/protocol.hpp`. This struct contains the following Ethernet header fields:

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


You can easily access the Ethernet header fields in a received packet or in a new packet (that will be sent out by your router) by casting the raw frame into a pointer of this type:

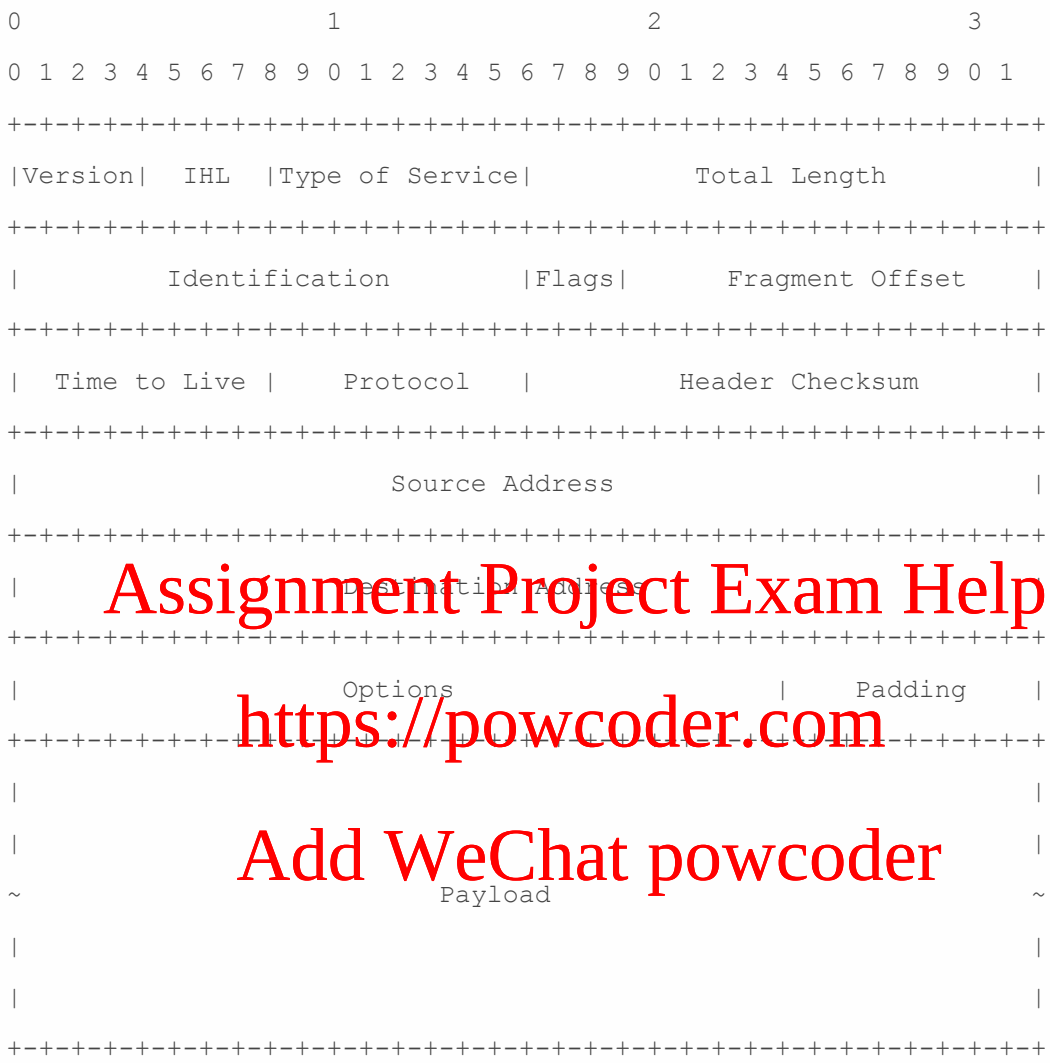
```
struct ethernet_hdr* ethHdr = (struct ethernet_hdr*)packet.data();

// The header fields can then be accessed and written to like so:
ethHdr->ether_dhost
ethHdr->ether_shost
ethHdr->ether_type
```

Requirements

- Your router **must** ignore Ethernet frames with a payload type other than IPv4.
- Your router **must** ignore Ethernet frames not destined to the router, meaning any frame where the destination hardware address is neither the corresponding MAC address of the interface nor the broadcast address (`FF:FF:FF:FF:FF:FF`).
- Your router **must** appropriately dispatch the payload of Ethernet frames when this payload contains an IPv4 packet.

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, which is the central principle behind the operation of the Internet. IP is tasked with delivering a packet from a source host to a destination host solely using IP addresses located in the packet header. To accomplish this, 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 `ip_hdr` struct in `core/protocol.hpp`. This struct contains the following IPv4 header fields:

```
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 */
    uint8_t ip_ttl;              /* Time-to-live (TTL) */
    uint8_t ip_p;                /* Protocol */
    uint16_t ip_sum;             /* Checksum */
```

```
    uint32_t ip_src;             /* Source IP address */
    uint32_t ip_dst;             /* Destination IP address */
} __attribute__ ((packed));
```

You can easily access the IP header fields in a received packet or in a new packet (that will be sent out by your router) by casting the raw frame into a pointer of this type:

```
struct ip_hdr* ipHdr = (struct ip_hdr*)(packet.data() + sizeof(struct ethernet_hdr));

// The header fields can then be accessed and written to like so:
ipHdr->ip_src
ipHdr->ip_dst
...
```

Requirements

- For each incoming IPv4 packet, your router **must** verify that its checksum is valid and minimum length is at least as long as required (20 bytes minimum).
    - Invalid packets **must** be discarded (a proper ICMP error response is **NOT** required for this project).
- Your router should classify datagrams into two categories: (1) those destined to the router (i.e., to one of the IP addresses of the router), and (2) those to be forwarded:
    - For (1), if the packet carries an ICMP payload, it should be properly dispatched. Otherwise, it **should** be discarded (a proper ICMP error response is **NOT** required for this project).
    - For (2), your router **must** use the longest prefix match algorithm to find a next-hop IP address in the routing table and attempt to forward the packet to that host.
- For each forwarded IPv4 packet, your router **must** correctly decrement its TTL and recompute its checksum.

ICMP Packets

The Internet Control Message Protocol (ICMP) (RFC 792) is a simple protocol that can be used to send control information to a host. In this assignment, your router will use ICMP to respond to “ping” messages sent to it by other hosts.

A “ping” exchange takes the form of an ICMP `Echo` message sent to a destination, followed by the destination responding to the sender with an ICMP `Echo Reply` message. `Echo` and `Echo Reply` ICMP messages are structured as follows (contained within the payload of an IPv4 packet):

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Code = 0   |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Identifier          |        Sequence Number        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Data ...
```

```
+-+-+-+-+-
```

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

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

Note that, unlike the IP header checksum, the ICMP header checksum covers **both** the ICMP header and the "Data" field.

For your convenience, the starter code defines the `icmp_hdr` struct in `core/protocol.hpp`. This struct contains the following IPv4 header fields:

```
struct icmp_hdr
{
    uint8_t icmp_type;      /* Type */
    uint8_t icmp_code;      /* Code */
    uint16_t icmp_sum;      /* Checksum */
    uint16_t icmp_id;       /* Identifier */
    uint16_t icmp_seq;      /* Sequence number */
} __attribute__ ((packed));
```


You can easily access the ICMP header fields in a received packet (or in a new packet that will be sent out by your router) by casting the raw frame into a pointer of this type:

```
struct icmp_hdr* icmpHdr = (struct icmp_hdr*) (packet.data() + sizeof(struct
ethernet_hdr)
    + sizeof(struct ip_hdr));

// The header fields can then be accessed and written to like so:
icmpHdr->icmp_type
icmpHdr->icmp_code
icmpHdr->icmp_sum
icmpHdr->icmp_id
icmpHdr->icmp_seq
```

You may wish to create additional structs for ICMP messages. If you do so, make sure to use the `packed` attribute (as above) so that the compiler doesn't try to align the fields in the struct to word boundaries.

Requirements

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

- For each incoming ICMP `Echo` message (`Type=8`) packet that is destined to one of the router's interfaces, your router **must** verify the ICMP checksum and discard the packet if the checksum is wrong. As mentioned above, please note that the ICMP checksum covers **both** the ICMP header and the data after the header.

- Echo requests sent to other IP addresses **must** be forwarded to their respective next hop addresses, just like TCP and UDP segments contained in IPv4 packets.

- Your router **must** generate an `Echo Reply` message (`Type=0`) in response to an incoming `Echo` message (`Type=8`) destined to one of the router's interface IP addresses.

- The `TTL` field in the IP header **must** be set to `64` when your router creates a new `Echo Reply` message.

Environment Setup

Initial Setup

For this project, you should use the provided Vagrant environment, as it installs several critical dependencies and starts all necessary daemons. You can follow the instructions in the `Vagrantfile` file to recreate the environment natively, but you do so at your own risk. Vagrant is a virtualization manager that greatly simplifies the process of creating and managing virtual machines. Vagrant requires that you have a virtualization engine installed – we recommend the open source VirtualBox platform for this purpose.

After installing both VirtualBox and Vagrant (in that order), launch a terminal and run the following commands:

1. Clone the project template repository:

   ```
   git clone https://github.com/eric-fcs118-router-project-1/cs118-router
   cd ~/cs118-router
   ```

2. Initialize the VM:

   ```
   vagrant up
   ```

3. Establish an SSH session to the created VM:

   ```
   vagrant ssh
   ```

   In this project you will need to open at least two simultaneous SSH sessions to the VM: one running Mininet to emulate the network topology and run commands on the emulated nodes, and another to build and run your router implementation. You can also accomplish this with `screen` (or `tmux`) – further details on this second method are provided in the "Running Your Router" section below.

   > **Do not start the VM instance manually from the VirtualBox GUI. Otherwise, you may encounter various issues, including connection errors, connection timeouts, missing packets, and so on.**

4. Work on your project:

   All files in the `~/cs118-router` directory on your host machine will be automatically synchronized with the `/vagrant` directory on your virtual machine. To compile your code, run the following commands in one of your SSH sessions:

   ```
   cd /vagrant
   make
   ```


Notes

- To stop the VM, one can run (from your host machine):

```
vagrant halt
```

- If your VM is encountering strange issues, you can reboot it by running (from your host machine):

```
vagrant reload
```

Running Your Router

To run your router, you will need to run in two commands in parallel: (1) the Mininet process that the emulates network topology and (2) your router. For ease of debugging, you can run them in `screen` (or `tmux`) environments or simply in separate SSH sessions (as discussed above):

- To run the Mininet network emulation process:

```
vagrant ssh #  or   vagrant ssh -- -Y
cd /vagrant
sudo python3 ./run.py # must be run as superuser
...
mininet>
```

- To run your router:


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


Note If after starting your router, you see the following message:

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

You need to start or restart the Mininet process. The expected initial output is something like the following:

```
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 (172.32.10.1, 22:69:6c:08:25:e9)
...
```

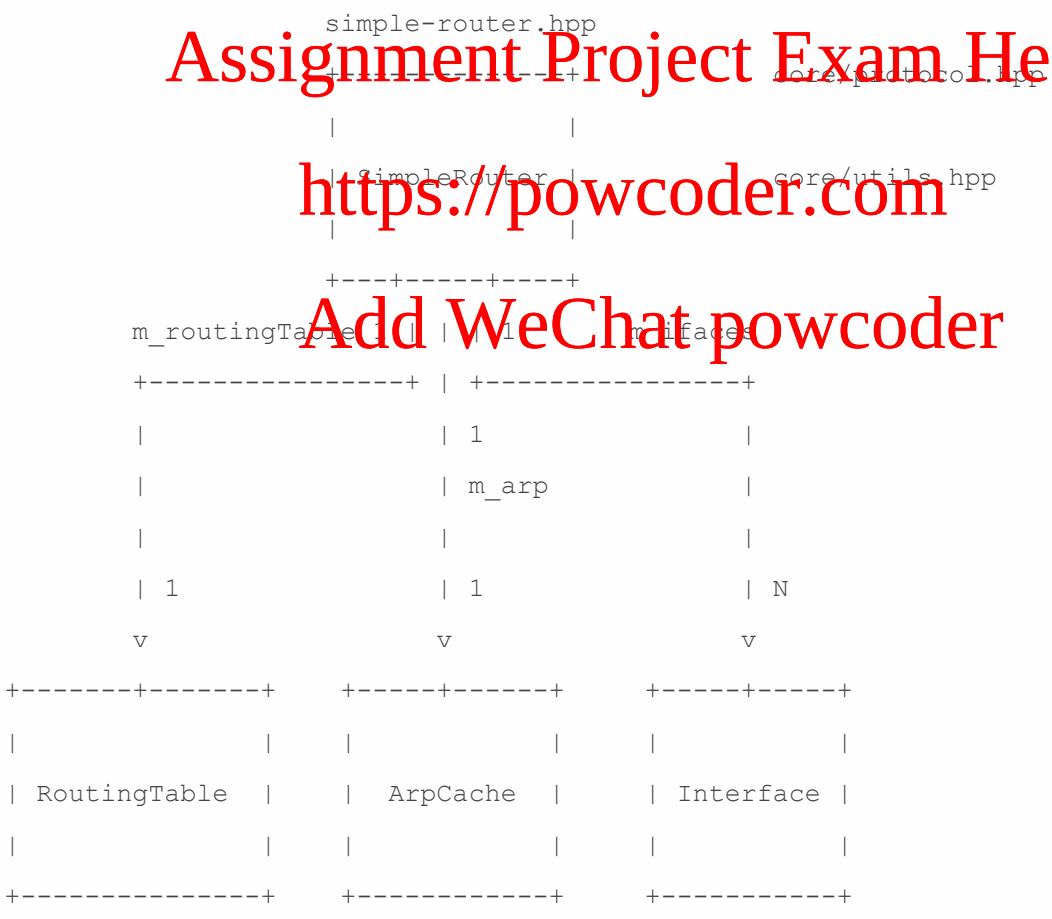
The VM environment you're using in this project runs a process that redirects packets from the Mininet-emulated switch to your router. If you want to see debug output of that redirector, you can use the `journalctl` command:

```
root@vagrant:/vagrant/# journalctl -f _SYSTEMD_UNIT=pox.service
-- Logs begin at Sat 2017-04-08 20:51:21 UTC. --
Apr 09 19:57:48 vagrant pox.py[8879]: POX 0.5.0 (eel) / Copyright 2011-2014 James
```

```
McCauley, et al.
Apr 09 19:57:48 vagrant pox.py[8879]: INFO:.usr.local.lib.python2.7.dist-
packages.ucla_cs118.pox_rpc_server:Starting packet redirector...
Apr 09 19:57:48 vagrant pox.py[8879]: -- 04/09/17 19:57:48.798 Network: listening for
tcp connections at 127.0.0.1:8888
Apr 09 19:57:48 vagrant pox.py[8879]: -- 04/09/17 19:57:48.798 Network: published
endpoints for object adapter `SimpleRouter':
Apr 09 19:57:48 vagrant pox.py[8879]:    tcp -h 127.0.0.1 -p 8888
Apr 09 19:57:48 vagrant pox.py[8879]: -- 04/09/17 19:57:48.799 Network: accepting tcp
connections at 127.0.0.1:8888
Apr 09 19:57:48 vagrant pox.py[8879]: INFO:core:POX 0.5.0 (eel) is up.
...
```

Starter Code Overview

Here is the overal structure of the starter code:



routing-table.hpp arp-cache.hpp core/interface.hpp

- `SimpleRouter` (`simple-router.hpp|cpp`)

  The main class for your simple router, encapsulating `RoutingTable` and a set of `Interface` objects, as well as other portions of the implementation that already have been implemented.

- `Interface` (`core/interface.hpp`)

  A class containing information about a specific interface of the router, including the interface name (`name`), hardware address (`addr`), and IPv4 address (`ip`).

- `RoutingTable` (`routing-table.hpp|cpp`)

  A class implementing a simple routing table for your router. The contents of the routing table are automatically loaded from a text file, with the default filename for this file being `RTABLE` (the name can be changed using the `RoutingTable` option in the `router.config` config file).

Key Methods

Your router receives raw Ethernet frames and sends raw Ethernet frames, either sending a reply to the sending host or forwarding its frame to the appropriate next hop. The key functions for these operations are as follows:

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

```
/**
 * This method is called each time the router receives a packet on
 * an 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 packet on interface \p outIface
 */
void
SimpleRouter::sendPacket(const Buffer& packet, const std::string& outIface);
```

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

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

- Need to implement Method to look up the best matching entry in the routing table (`routing-table.hpp|cpp`):

```
/**
```

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

Debugging Functions

We have provided you with some basic debugging functions in `core/utils.hpp`. 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 is sent and received by the emulated nodes, i.e., "router", "server1" and "server2". For example, to see the packets sent and received by the `server1` node, use the following command from the Mininet command-line interface (CLI):


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

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


```
mininet> xterm server1
```

and then from inside the newly opened `xterm`, one can run:

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

Length of Assignment

This assignment is considerably harder than Project 1, so get an early start and make use of Piazza! We highly recommend that this project be worked on in groups of two.

A Few Hints

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

- Sanity-check the packet (i.e., ensure it meets the minimum length and has a valid checksum).
- Decrement its TTL by 1 and then recompute the packet checksum over the modified header.
- Find out which entry in the routing table has the longest prefix match with the packet's destination IP address.

- Check the ARP cache for the next-hop MAC address corresponding to the next-hop IP using `ArpCache::lookup`. If a mapping is found, use it to generate an Ethernet frame and send the packet on the appropriate nexthop you obtained in the previous step. Otherwise, queue an ARP request for the next-hop IP using `ArpCache::queueRequest`.

If an incoming IP packet is destined toward 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.
- 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 excludes many low-level details.

Submission Requirements

To submit your project:

1. Create a `README.md` file placed in your code that includes:
    - Name and UID of each team member (up to 2 members in one team).
    - Acknowledgement of any online tutorials or code examples (except materials from this course) you used for this project.
2. All of your source code, `Makefile`, `README.md`, and `Vagrantfile`. **Make sure to update the `Makefile` to include the UCLA UIDs of any project members (leaving `USERID2` empty if you completed this project solo).**
3. Remove all custom test files (especially large test files) before creating your submission tarball.
4. Use the provided `Makefile` in the starter code to create a tarball.

    ```
    make tarball
    ```

5. Submit the resulting tarball using the CCLE project submission page.

Before submission, please make sure that:

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

> Submissions that do not compile will not receive any credit.

Grading

Grading Criteria

Note that test cases and points are subject adjustments.

1. Ping tests (50 points)
    - 1.1. (15 points) Pings from client to all other hosts (all pings expected to succeed).

- 1.2. (10 points) Pings from server1 to all other hosts (all pings expected to succeed).
- 1.3. (10 points) Pings from server2 to all other hosts (all pings expected to succeed).
- 1.4. (15 points) Ping responses (from client) have proper TTLs.

2. File transfer tests (40 points)
    - 3.1. (10 points) Transfer a small file (50KB) from client to server.
    - 3.2. (10 points) transfer a medium file (1MB) from client to server.
    - 3.3. (20 points) transfer a large file (10MB) from client to server.
3. Whether the submitted tarball includes extraneous files (10 points)

> Note that your router should work in other single-router network topologies with different routing tables, interface names, and IP addresses.

Acknowledgements

This project is based on an earlier CS 118 class project originally created by Prof. Alexander Afanasyev (now at Florida International University), which was in turn based on the Stanford CS 144 class project by Prof. Philip Levis and Prof. Nick McKeown.