CSC458/2209 Programming Assignment 1

Winter 2025

Packaging IP in Ethernet

The main goal of PA1 is to set the environment for your programming assignments[1], and use that environment to implement a simplified portion of the network stack: creating Ethernet frames from IP packets and sending them out. One of the key ideas of the networking stack is to hide details of lower layers from higher levels and make it possible to have different upper layers that rely on the same lower layer. In this assignment, we focus on the link layer which is responsible for creating and sending out Ethernet frames and also receiving and processing incoming Ethernet frames. You will complete a simplified network interface to perform these two tasks.

1. Send Ethernet frames and ARP requests

When the IP layer decides to send a packet, it identifies the next hop, which is the next machine/router on the network set to receive the packet[2]. This next hop must reside on the same physical network, enabling direct transmission of packets at the link layer, without intermediate routing decisions. Once the next hop address is determined, the IP layer invokes the link layer interface to transmit the packet to the next hop, providing it with 1) the IP packet and 2) the IP address of the next hop.

Although the IP layer supplies the next hop's IP address to the link layer, MAC addresses are necessary to reach other interfaces within the physical network. Therefore, the first job of the link layer interface is to determine the corresponding MAC address for the given IP address. The link layer protocol used to perform this task is the Address Resolution Protocol (ARP). To this end, the link layer maintains a mapping, known as the "ARP Table", that translates IP addresses to MAC addresses. It simply contains the MAC address associated with different IP addresses that the interface is aware of.

If the IP-MAC mapping for the next hop is available in the table when the link layer is invoked (i.e., the interface knows the corresponding MAC address for the next hop), it can readily proceed with sending the packet by creating an Ethernet frame that contains

- the entire given IP packet as the payload,
- the translated MAC address as the destination MAC address,
- and the other appropriate header sections (such as its own MAC address as the source).

[1] CSC458 programming assignments are based on some of the Stanford CS144 lab assignments.

[2] The routing process determines the next hop, which will be the focus of PA2. For now, we assume that the next hop for the packet is already known.

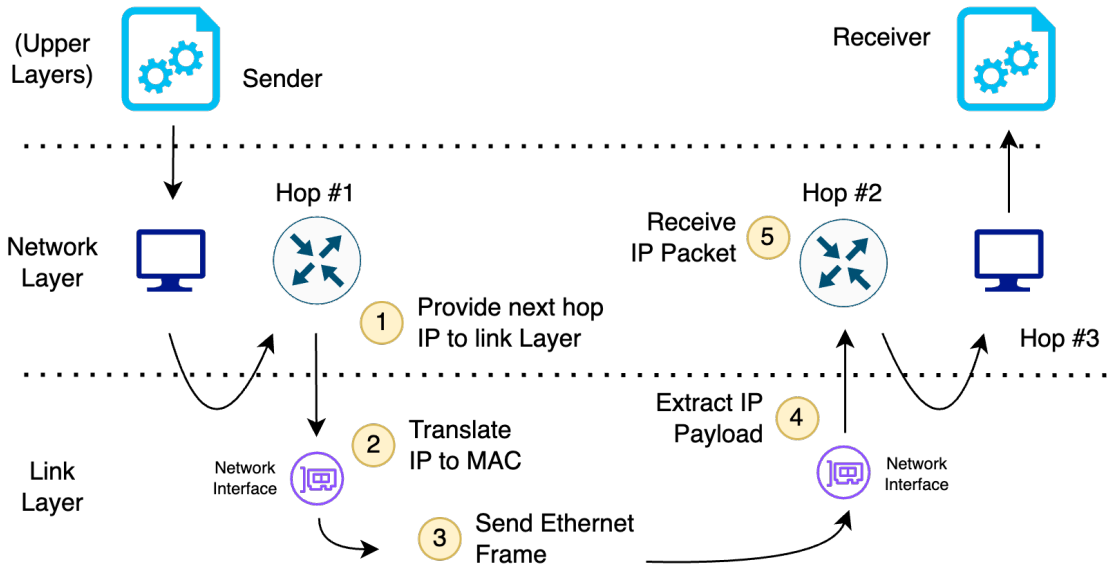

Figure 1: A typical communication workflow. You will implement steps 2, 3, and 4 in PA1.

If the next-hop IP address is not available in the table, the link layer must try to find it using the ARP protocol. It sends an ARP request to all interfaces in the local network, asking who has that specific IP address. When it receives a response, it records the answer in the table for future use and then sends the waiting packet. However, if it does not receive a response within a 5-second window, it assumes that the next hop is not accessible and drops the packet. Considering that computers can move between networks, the answers are only cached for a limited time (30 seconds), after which the entry is removed from the table. Hence, the table might also be referred to as the ARP cache.

2. Receive Ethernet frames and create ARP responses

The other task of the link layer interface is to process incoming Ethernet frames. When it receives an Ethernet frame addressed to this device, it will extract its payload and pass it to the upper layer (i.e., the IP layer). It also needs to process incoming ARP requests and respond to queries of its own IP address.

Getting Started

This section guides you to set up your development environment and test your code.

Setting up the environment

The programming assignments require Linux operating system and a recent C++ compiler. We highly recommend that you use the virtual machine (VM) that is created for CS144 at the Stanford University. Do not use your own GNU/Linux installation as you might risk losing points during the marking. The instructions to use that is available at

https://stanford.edu/class/cs144/vm_howto/

The instructions on that website walk you through the installation and use of VirtualBox to use the VM image they prepared. There are also alternative instructions for computers with Apple M series chips, which are based on UTM instead of VirtualBox. Setting up this environment is important for the two programming assignments in this course.

Note: If downloading the images from the Stanford servers is taking too long, you might want to try these local mirrors for the **VirtualBox** and **UTM** images.

Getting the code

You will be adding some lines of code to a simple network simulator to implement the requirements of this assignment. Follow these steps to get the starter code:

1. Clone the starter code:

   ```
   git clone https://github.com/yganjali/csc458-pa
   cd csc458-pa
   ```

2. Verify that your build system is properly set up (or set it up in case you start from scratch by cloning the repository into a new folder):

   ```
   cmake -S . -B build
   ```

3. Now you can try to build the starter code:

   ```
   cmake --build build
   ```

4. You are ready to start working on the source code to complete the assignment. Whenever you want to build your solution and run the tests, you can:

   ```
   cmake --build build --target pa1
   ```

   which shows you the tests that your code has passed. There are currently 10 test cases included in your starter code to run your code through various scenarios.

Functions to complete

There are 4 methods in the **NetworkInterface** class (**network_interface.cc** file) that you should implement.

1. **void NetworkInterface::send_datagram(const InternetDatagram& dgram, const Address& next_hop)**

   This is the function that is called by the IP layer to send an IP packet (**dgram**) to the next hop (with the given IP address **next_hop**). The function should be completed to perform the following tasks.

   - First, check to see if you know the MAC address of the next hop, by looking it up in your cached ARP table. If you know the MAC address, you can create the proper Ethernet frame and put it in the ready-to-be-sent queue (which will be discussed later). To do this, create an Ethernet frame, set the type as an IPv4 packet (**EthernetHeader::TYPE_IPv4**), properly set the source and destination MAC addresses, and put the serialized version of the **dgram** in it.
   - If you do not know the MAC address of the next hop, you should try to find it. This is done by broadcasting an ARP request and asking what MAC address currently has the next hop IP address. Then, you should wait for the response to that request. You should also place this packet in a queue so that you can prepare it for sending when you receive a reply. There is one additional thing that you need to consider here:
     - If you have sent an ARP request for the same IP address in the last 5 seconds, you should not send a new ARP request. Instead, you should append this packet to the queue that holds previous packets waiting for that IP address.

2. **optional<InternetDatagram> NetworkInterface::recv_frame(const EthernetFrame& frame)**

   This is the function that is called when the system receives an Ethernet frame. Its job is to process it as follows:

   - If the frame is not destined for this interface, discard it. A frame is destined for this interface if its destination MAC address matches the interface's MAC address or if it is broadcast to the whole network.
   - If this frame is destined for this interface and its payload contains an IPv4 packet, then try to parse the payload as an **InternetDatagram**. If the parse is successful, it should be returned so that the system can pass it to the higher IP layer in the network stack.
   - If this frame is destined for this interface and its payload is an ARP message, process it. To do this, first try to parse the payload as an **ARPMessage**. If it can be parsed properly, then learn the mapping between the packet's Sender IP address and its MAC address and cache this information in the ARP cache table for 30 seconds. Note that this is true for both ARP responses and requests.
   - If it is an ARP request that asks for our IP address, reply to it. To do this, create an ARP reply message that is destined for the sender and contains the appropriate information, including our IP and MAC address. Then, package this ARP message in an Ethernet frame and place it in the ready-to-be-sent queue.

3. **optional<EthernetFrame> NetworkInterface::maybe_send()**

   Whenever the physical layer of the network is ready to send out a frame, it will call this function to check if there is any frames ready to be sent. At this point, you should check your ready-to-be-sent queue to see if there are any frames in it. If so, remove the first frame (the oldest) waiting in the queue and return it. If there are no frames to be sent, simply return nothing.

   Note that there could be different types of Ethernet frames in the queue: datagrams passed by the IP layer, ARP requests to learn the MAC address of the next hop, and ARP replies to requests about our IP address.

4. **void NetworkInterface::tick(const size_t ms_since_last_tick)**

   This is the callback function that informs you about the passage of time. When this function is called, it means that **ms_since_last_tick** milliseconds have passed since the last time it was called. You should keep track of time and perform the following two tasks:

   - Expire any entry in the ARP cache table that was learned more than 30 seconds ago.
   - Terminate the wait for any pending ARP reply for a next-hop IP that was sent more than 5 seconds ago. You should also discard any packets waiting for that IP address from the queue.

In addition to completing these 4 functions, you can add extra helper functions to the **NetworkInterface** class, or any extra state information to it (*e.g.*, the ARP cache table). Therefore, you can modify both **network_interface.cc** and **network_interface.hh** files. Do not modify any other files, because you will only submit these two files (along with **writeups/pa1.md**).

Submission and grading

We will use the MarkUs submission system for this assignment. Please check the class webpage for the link and instructions. You need to submit the two source-code files for the **NetworkInterface** class (**network_interface.cc** and **network_interface.hh**). You should also submit the **writeups/pa1.md** file that contains a few questions that you should answer about the assignment. In addition to checking your submission with automated tests (to test its functionality), TAs will read your source code and assess its coding style.

- 90% of the assignment mark is for the automated tests (i.e., correct functionality). This 90% is divided into:
  - 50% that can be acquired by passing the publicly available tests (already part of the starter code, under **tests** directory),
  - Another 40%, that will be dedicated to private tests. This is to make sure your code doesn't run correctly only in the provided scenarios.
- 5% of the mark is for providing reasonable answers to the questions found in **pa1.md**.
- The remaining 5% is awarded for your coding style. The coding style includes factors such as inline comments, proper variable names, breaking complex functions into smaller functions, preventing duplicate code by defining helper functions, etc. You should be familiar with such marking from previous programming courses.

Some Notes

- **Important note**: Although it is advised to backup your work periodically using some version control method, **you must not make your solutions publicly available.** Make sure to keep your repositories private.
- The tests are run independently of each other. Each test case creates new instances of your `NetworkInterface` class. When a test fails, a trace of the events that led to the unexpected results will be printed, which can greatly guide you in debugging your issues.
- This assignment is to be done individually. Our sample solution added around 200 lines of code to the starter code to pass all the tests.