Lab Checkpoint 4: down the stack (the network interface)

Due: Monday, May 29, 11 a.m.

Collaboration Policy: Same as checkpoint 0. Please do not look at other students' code or solutions to past versions of these assignments. Please fully disclose any collaborators or any gray areas in your writeup—disclosure is the best policy.

0 Overview

In this week's lab, you'll go down the stack and implement a network interface: the bridge between Internet datagrams that travel the world, and link-layer Ethernet frames that travel one hop. This component can fit "underneath" your TCP/IP implementation from the earlier labs, but it will also be used in a different setting: when you build a router in Lab 6, it will route datagrams *between* network interfaces. Figure 1 shows how the network interface fits into both settings.

In past labs, you wrote a TCP implementation that can exchange **TCP segments** with any other computer that speaks TCP. How are these segments actually conveyed to the peer's TCP implementation? As we've discussed, there are a few options:

- **TCP-in-UDP-in-IP.** The TCP segments can be carried in the payload of a user datagram. When working in a normal (user-space) setting, this is the easiest to implement: Linux provides an interface (a "datagram socket", `UDPSocket`) that lets applications supply *only the payload* of a user datagram and the target address, and the kernel takes care of constructing the UDP header, IP header, and Ethernet header, then sending the packet to the appropriate next hop. The kernel makes sure that each socket has an exclusive combination of local and remote addresses and port numbers, and since the kernel is the one writing these into the UDP and IP headers, it can guarantee isolation between different applications.

- **TCP-in-IP.** In common usage, TCP segments are almost always placed directly inside an Internet datagram, without a UDP header between the IP and TCP headers. This is what people mean by "TCP/IP." This is a little more difficult to implement. Linux provides an interface, called a TUN device, that lets application supply an *entire* Internet datagram, and the kernel takes care of the rest (writing the Ethernet header, and actually sending via the physical Ethernet card, etc.). But now the application has to construct the full IP header itself, not just the payload.

- **TCP-in-IP-in-Ethernet.** In the above approach, we're still relying on the Linux kernel for part of the networking stack. Each time your code writes an IP datagram to the TUN device, Linux has to construct an appropriate link-layer (Ethernet) frame with the IP datagram as its payload. This means Linux has to figure out the next hop's Ethernet destination address, given the IP address of the next hop. If it doesn't know

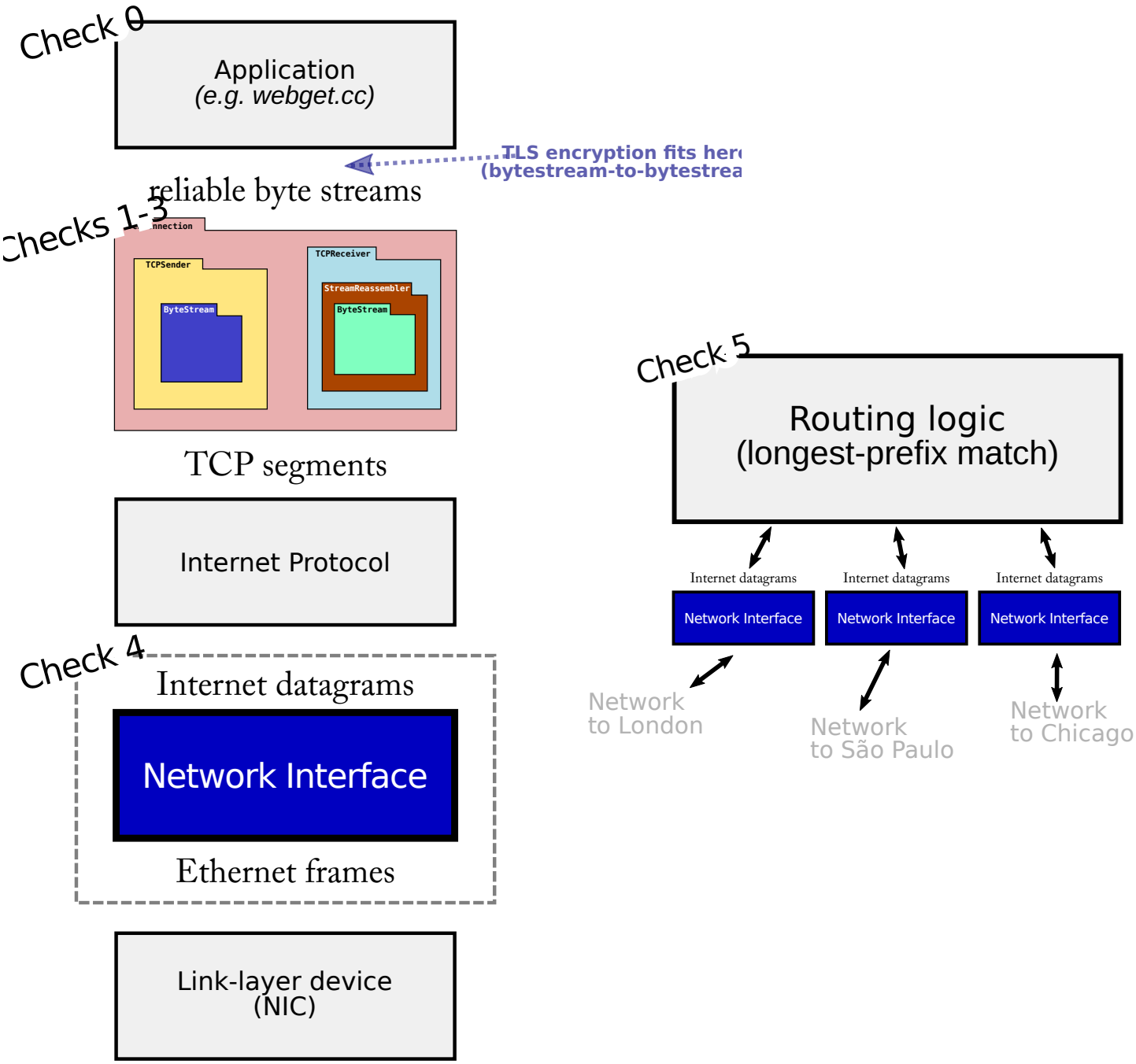

Figure 1: The network interface bridges the worlds of Internet datagrams and of link-layer frames. This component is useful as part of a host's TCP/IP stack (left side), and also as part of an IP router (right side).

this mapping already, Linux broadcasts a query that asks, "Who claims the following IP address? What's your Ethernet address?" and waits for a response.

These functions are performed by the *network interface*: a component that translates outbound IP datagrams into link-layer (e.g., Ethernet) frames and vice versa. (In a real system, network interfaces typically have names like `eth0`, `eth1`, `wlan0`, etc.) **In this week's lab**, you'll implement a network interface, and stick it at the very bottom of your TCP/IP stack. Your code will produce raw Ethernet frames, which will be handed over to Linux through an interface called a TAP device—similar to a TUN device, but more low-level, in that it exchanges raw link-layer frames instead of IP datagrams.

Most of the work will be in looking up (and caching) the Ethernet address for each next-hop IP address. The protocol for this is called the **Address Resolution Protocol**, or **ARP**.

We've given you unit tests that put your network interface through its paces. In Lab 6, you'll use the same network interface outside the context of TCP, as a part of an IP router.

1 Getting started

1. Make sure you have committed all your solutions to Checkpoint 1. Please don't modify any files outside the top level of the `src` directory, or `webget.cc`. You may have trouble merging the Checkpoint 4 starter code otherwise.
2. While inside the repository for the lab assignments, run `git fetch --all` to retrieve the most recent version of the lab assignment.
3. Download the starter code for Checkpoint 4 by running `git merge origin/check4-startercode`. (If you have renamed the "origin" remote to be something else, you might need to use a different name here, e.g. `git merge upstream/check4-startercode`.)
4. Make sure your build system is properly set up: `cmake -S . -B build`
5. Compile the source code: `cmake --build build`
6. Open and start editing the `writeups/check4.md` file. This is the template for your lab writeup and will be included in your submission.
7. Reminder: please make frequent **small commits** in your local Git repository as you work. If you need help to make sure you're doing this right, please ask a classmate or the teaching staff for help. You can use the `git log` command to see your Git history.

2 Checkpoint 4: The Address Resolution Protocol

Your main task in this lab will be to implement the three main methods of `NetworkInterface` (in the `network_interface.cc` file), maintaining a mapping from IP addresses to Ethernet addresses. The mapping is a cache, or "soft state": the NetworkInterface keeps it around

for efficiency's sake, but if it has to restart from scratch, the mapping will naturally be regenerated without causing a problem.

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

   This method is called when the caller (e.g., your `TCPConnection` or a router) wants to send an outbound Internet (IP) datagram to the next hop.[1] It's your interface's job to translate this datagram into an Ethernet frame and (eventually) send it.

   - *If the destination Ethernet address is already known*, send it right away. Create an Ethernet frame (with `type = EthernetHeader::TYPE_IPv4`), set the payload to be the serialized datagram, and set the source and destination addresses.
   - *If the destination Ethernet address is unknown*, broadcast an ARP request for the next hop's Ethernet address, and queue the IP datagram so it can be sent after the ARP reply is received.

   Except: You don't want to flood the network with ARP requests. If the network interface already sent an ARP request about the same IP address in the last five seconds, don't send a second request—just wait for a reply to the first one. Again, queue the datagram until you learn the destination Ethernet address.

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

   This method is called when an Ethernet frame arrives from the network. The code should ignore any frames not destined for the network interface (meaning, the Ethernet destination is either the broadcast address or the interface's own Ethernet address stored in the `_ethernet_address` member variable).

   - *If the inbound frame is IPv4*, parse the payload as an `InternetDatagram` and, if successful (meaning the `parse()` method returned `true`), return the resulting `InternetDatagram` to the caller.
   - *If the inbound frame is ARP*, parse the payload as an `ARPMessage` and, if successful, remember the mapping between the sender's IP address and Ethernet address for 30 seconds. (Learn mappings from both requests and replies.) In addition, if it's an ARP request asking for our IP address, send an appropriate ARP reply.

3. `std::optional<EthernetFrame> maybe_send();`

   This is the `NetworkInterface`'s opportunity to actually send a `EthernetFrame` if it wants to.

4. `void NetworkInterface::tick(const size_t ms_since_last_tick);`

   This is called as time passes. Expire any IP-to-Ethernet mappings that have expired.

[1] Please don't confuse the *ultimate* destination of the datagram, which is in the datagram's own header as the destination address, with the next hop. In this lab you're *only* going to care about the next hop's address.

You can test your implementation by running `cmake --build build --target check4`
This test does not rely on your TCP implementation.

3 Q & A

- *How much code are you expecting?*

  Overall, we expect the implementation (in `network_interface.cc`) will require about 100–150 lines of code in total.

- *How do I "send" an Ethernet frame?*

  Return it when `maybe_send()` is called.

- *What data structure should I use to record the mapping between next-hop IP address and Ethernet addresses?*

  Up to you!

- *How do I convert an IP address that comes in the form of an Address object, into a raw 32-bit integer that I can write into the ARP message?*

  Use the `Address::ipv4_numeric()` method.

- *What should I do if the NetworkInterface sends an ARP request but never gets a reply? Should I resend it after some timeout? Signal an error to the original sender using ICMP?*

  In real life, yes, both of those things, but don't worry about that in this lab. (In real life, an interface will eventually send an ICMP "host unreachable" back across the Internet to the original sender if it can't get a reply to its ARP requests.)

- *What should I do if an InternetDatagram is queued waiting to learn the Ethernet address of the next hop, and that information never comes? Should I drop the datagram after some timeout?*

  Again, definitely a "yes" in real life, but don't worry about that in this lab.

- *How do `parse()` and `serialize()` work?*

  `parse()` takes a `T& obj` and `vector<Buffer>& buffers`. On success, it fills `obj` with the result and returns `true`. Otherwise, it returns `false`.

  `serialize()` takes a `T& obj` and returns the result as a `vector<Buffer>`.

- *Where can I read if there are more FAQs after this PDF comes out?*

  Please check the website (https://cs144.github.io/lab_faq.html) and EdStem regularly.

4 Development and debugging advice

1. Implement the `NetworkInterface`'s public interface (and any private methods or functions you'd like) in the file `network_interface.cc`. You may add any private members you like to the `NetworkInterface` class in `network_interface.hh`.

2. You can test your code with `cmake --build build --target check4`.

3. Please re-read the section on "using Git" in the Checkpoint 0 document, and remember to keep the code in the Git repository it was distributed in on the `main` branch. **Make small commits, using good commit messages that identify what changed and why.**

4. Please work to make your code readable to the CA who will be grading it for style. Use reasonable and clear naming conventions for variables. Use comments to explain complex or subtle pieces of code. Use "defensive programming"—explicitly check preconditions of functions or invariants, and throw an exception if anything is ever wrong. Use modularity in your design—identify common abstractions and behaviors and factor them out when possible. Blocks of repeated code and enormous functions will make it hard to follow your code.

5 Submit

1. In your submission, please only make changes to the `.hh` and `.cc` files in the `src` directory. Within these files, please feel free to add private members as necessary, but please don't change the *public* interface of any of the classes.

2. Before handing in any assignment, please run these in order:

   (a) Make sure you have committed all of your changes to the Git repository. You can run `git status` to make sure there are no outstanding changes. Remember: make small commits as you code.

   (b) `cmake --build build --target format` (to normalize the coding style)

   (c) `cmake --build build --target check4` (to make sure the automated tests pass)

   (d) Optional: `cmake --build build --target tidy` (suggests improvements to follow good C++ programming practices)

3. Write a report in `writeups/check4.md`. This file should be a roughly 20-to-50-line document with no more than 80 characters per line to make it easier to read. The report should contain the following sections:

(a) **Program Structure and Design.** Describe the high-level structure and design choices embodied in your code. You do not need to discuss in detail what you inherited from the starter code. Use this as an opportunity to highlight important design aspects and provide greater detail on those areas for your grading TA to understand. You are strongly encouraged to make this writeup as readable as possible by using subheadings and outlines. Please do not simply translate your program into an paragraph of English.

(b) **Implementation Challenges.** Describe the parts of code that you found most troublesome and explain why. Reflect on how you overcame those challenges and what helped you finally understand the concept that was giving you trouble. How did you attempt to ensure that your code maintained your assumptions, invariants, and preconditions, and in what ways did you find this easy or difficult? How did you debug and test your code?

(c) **Remaining Bugs.** Point out and explain as best you can any bugs (or unhandled edge cases) that remain in the code.

4. Please also fill in the number of hours the assignment took you and any other comments.

5. Please let the course staff know ASAP of any problems at a lab session, or by posting a question on Ed. Good luck!