Lab Checkpoint 1: stitching substrings into a byte stream

Due: Sunday, January 19, 11:59 p.m. (late/extension deadline: Wed. Jan. 22 @ 7 p.m.)

Collaboration Policy: Same as checkpoint 0.

0 Overview

For Checkpoint 0, you used an *Internet stream socket* to fetch information from a website and send an email message, using Linux's built-in implementation of the Transmission Control Protocol (TCP). This TCP implementation managed to produce a pair of *reliable in-order byte streams* (one from you to the server, and one in the opposite direction), even though the underlying network only delivers "best-effort" datagrams. By this we mean: short packets of data that can be lost, reordered, altered, or duplicated. You also implemented the byte-stream abstraction yourself, in memory within one computer. Over the coming weeks, you'll implement TCP yourself, to provide the byte-stream abstraction between a pair of computers separated by an unreliable datagram network.

> ⋆*Why am I doing this?* Providing a service or an abstraction on top of a different less-reliable service accounts for many of the interesting problems in networking. Over the last 40 years, researchers and practitioners have figured out how to convey all kinds of things—messaging and e-mail, hyperlinked documents, search engines, sound and video, virtual worlds, collaborative file sharing, digital currencies—over the Internet. TCP's own role, providing a pair of reliable byte streams using unreliable datagrams, is one of the classic examples of this. A reasonable view has it that TCP implementations count as the **most widely used** nontrivial computer programs on the planet.

The lab assignments will ask you to build up a TCP implementation in a modular way. Remember the `ByteStream` you just implemented in Checkpoint 0? In the coming labs, you'll end up convey two of them across the network: an "outbound" `ByteStream`, for data that a local application writes to a socket and that your TCP will send *to* the peer, and an "inbound" `ByteStream` for data coming *from* the peer that will be read by a local application.

This checkpoint contains a "hands-on" component and an implementation component. You might prefer to start the implementation component before the lab session, and do the hands-on component at the lab session. If you are a CGOE student, please use EdStem to coordinate a time with another student to do the hands-on component.

The hands-on component is new this year and involves multiple moving parts—so there might be some glitches. Please bear with us at the lab session and we'll do our best to get it working for everybody. If you see an error message from the https://cs144.net website, please report it in a public post on EdStem and we'll take a look.

1 Getting started

Your implementation of TCP will use the same Minnow library that you used in Checkpoint 0, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Checkpoint 0. Please don't modify any files outside of the `src` directory, or `webget.cc`. You may have trouble merging the Checkpoint 1 starter code otherwise.
2. While inside the repository for the lab assignments, run `git fetch` to retrieve the most recent version of the lab assignments.
3. Download the starter code for Checkpoint 1 by running `git merge origin/check1-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/check1.md` file. This is the template for your lab writeup and will be included in your submission.

2 Hands-on component: a private network for the class

We have created a private network for the CS144 class. This will allow your VM to send datagrams directly to and from the VMs of other students in the class. To make your VM join this network:

1. On your VM, install the "wireguard" package by running `sudo apt install wireguard`
2. Visit https://cs144.net/wg and follow the instructions to join the CS144 private network.
3. Once you have joined the network, verify that you can connect by following the "ping" instructions on that page (the instructions appear after you have joined the network).
4. Every time you reboot your VM, you'll have to rejoin the network (if you want to be able to send datagrams to and from other students in this class). You don't have to register a new public key each time, but you do have to rerun the commands on that webpage. The commands will be the same each time.

2.1 Ping a friend and look at the datagrams

1. On your own computer (e.g. your Mac or Windows machine—not your VM), install the "wireshark" program by following the instructions at https://www.wireshark.org/. (If you are using Debian or Ubuntu GNU/Linux, the command is `sudo apt install wireshark`.

2. Ask a groupmate for their IP address (the one shown on the https://cs144.net/wg webpage **for them**). Using the `ping` command, send some "echo request" datagrams to your friend, and make sure that you get some "echo reply" datagrams back.

3. Tips:

   - You can end the "ping" program by typing `ctrl-C`.)
   - You can make the "ping" command go faster by including the argument `-i 0.2`. This will make it send an "echo request" every 0.2 seconds (5 times per second).
   - You can make the "ping" command print out a summary of the statistics so far (without ending it) by running this command in another terminal: `killall -QUIT ping`.

4. Begin a report in your writeup, including the following information:

   (a) What is the average round-trip delay between when your VM sends an "echo request" and when it receives an "echo reply" from your groupmate's VM?

   (b) What was the delivery rate (what percentage of "echo requests" received a corresponding "echo reply")? What was the loss rate (this is 100% minus the delivery rate)? Send at least 1,000 pings to get a reliable estimate. (This will take about three minutes if using `ping -i 0.2`.

   (c) Did you see any duplicated datagrams (ping will print "DUP")?

   (d) While the ping is running, you and your groupmate can capture some of the raw Internet datagrams by running `sudo rm /tmp/capture.raw; sudo tcpdump -n -w /tmp/capture.raw -i wg0 --print --packet-buffered`. This command will capture the datagrams on the "wg0" interface (the private class network) to a file ("/tmp/capture.raw"), while also printing them out to the screen. Make sure you see some "echo request" and "echo reply" lines printed—that indicates your groupmate is receiving your datagrams and replying to you.

   (e) Use the `wireshark` program to inspect the `/tmp/capture.raw` file on each of your VMs. You probably want to `scp` the `capture.raw` file to your own computer (e.g. a Mac or Windows machine) and then use wireshark to open this file, so you can use its graphical interface. Can you find the fields of the Internet datagram that were discussed in the Jan. 10 (and match the diagram at https://www.rfc-editor.org/rfc/rfc791.html#page-11)?

   (f) Are there any differences between the **same** datagrams when they were captured on your VM compared with when they were captured on your friend's VM? What?

2.2 Send an Internet datagram by hand

In the `apps/ip_raw.cc` file, write a program that sends an Internet datagram to your friend by using a raw socket, using the same method as the January 10 lecture. It's okay to adapt code from this lecture.

1. Send your groupmate an Internet datagram with IP protocol "5" (you'll have to use "sudo" to run the "./build/apps/ip_raw" program), and have your friend use `tcpdump` to make sure they receive the datagram. They can run `sudo tcpdump -n -i wg0 'proto 5'` to print out only datagrams matching protocol "5". Make sure they get it!

2. Send your groupmate a user datagram (with IP protocol "17"), using the "user datagram" header format in https://www.rfc-editor.org/rfc/rfc768. Have your groupmate **receive** this datagram *without using "sudo"*. They can use the "nc -u" program as was done in lecture, or a C++ program using the `UDPSocket` class—whatever they prefer!

3. Include the code for your "ip_raw.cc" in your submission to this checkpoint.

4. Do the same in reverse and receive a datagram from your groupmate.

3 Implementation: putting substrings in sequence

As part of the lab assignment, you are implementing a TCP receiver: the module that receives datagrams and turns them into a reliable byte stream to be read from the socket by the application—just as your `webget` program read the byte stream from the webserver in Checkpoint 0.

The TCP sender is dividing its byte stream up into short *segments* (substrings no more than about 1,460 bytes apiece) so that they each fit inside a datagram. But the network might reorder these datagrams, or drop them, or deliver them more than once. The receiver must reassemble the segments into the contiguous stream of bytes that they started out as.

In this lab you'll write the data structure that will be responsible for this reassembly: a `Reassembler`. It will receive substrings, consisting of a string of bytes, and the index of the first byte of that string within the larger stream. **Each byte of the stream** has its own unique index, starting from zero and counting upwards. As soon as the Reassembler knows the **next** byte of the stream, it will write it to the Writer side of a ByteStream— the same ByteStream you implemented in checkpoint 0. The Reassembler's "customer" can read from the Reader side of the same ByteStream.

Here's what the interface looks like:

```
// Insert a new substring to be reassembled into a ByteStream.
void insert( uint64_t first_index, std::string data, bool is_last_substring );

// How many bytes are stored in the Reassembler itself?
// This function is for testing only; don't add extra state to support it.
uint64_t count_bytes_pending() const;

// Access output stream reader
Reader& reader();
```

⋆*Why am I doing this?* TCP robustness against reordering and duplication comes from its ability to stitch arbitrary excerpts of the byte stream back into the original stream. Implementing this in a discrete testable module will make handling incoming segments easier.

The full (public) interface of the reassembler is described by the `Reassembler` class in the `reassembler.hh` header. Your task is to implement this class. You may add any private members and member functions you desire to the `Reassembler` class, but you cannot change its public interface.

3.1 What should the Reassembler store internally?

The `insert` method informs the `Reassembler` about a new excerpt of the ByteStream, and where it fits in the overall stream (the index of the beginning of the substring).

In principle, then, the `Reassembler` will have to handle three categories of knowledge:

1. Bytes that are the **next bytes** in the stream. The `Reassembler` should push these to the stream (`output_.writer()`) as soon as they are known.

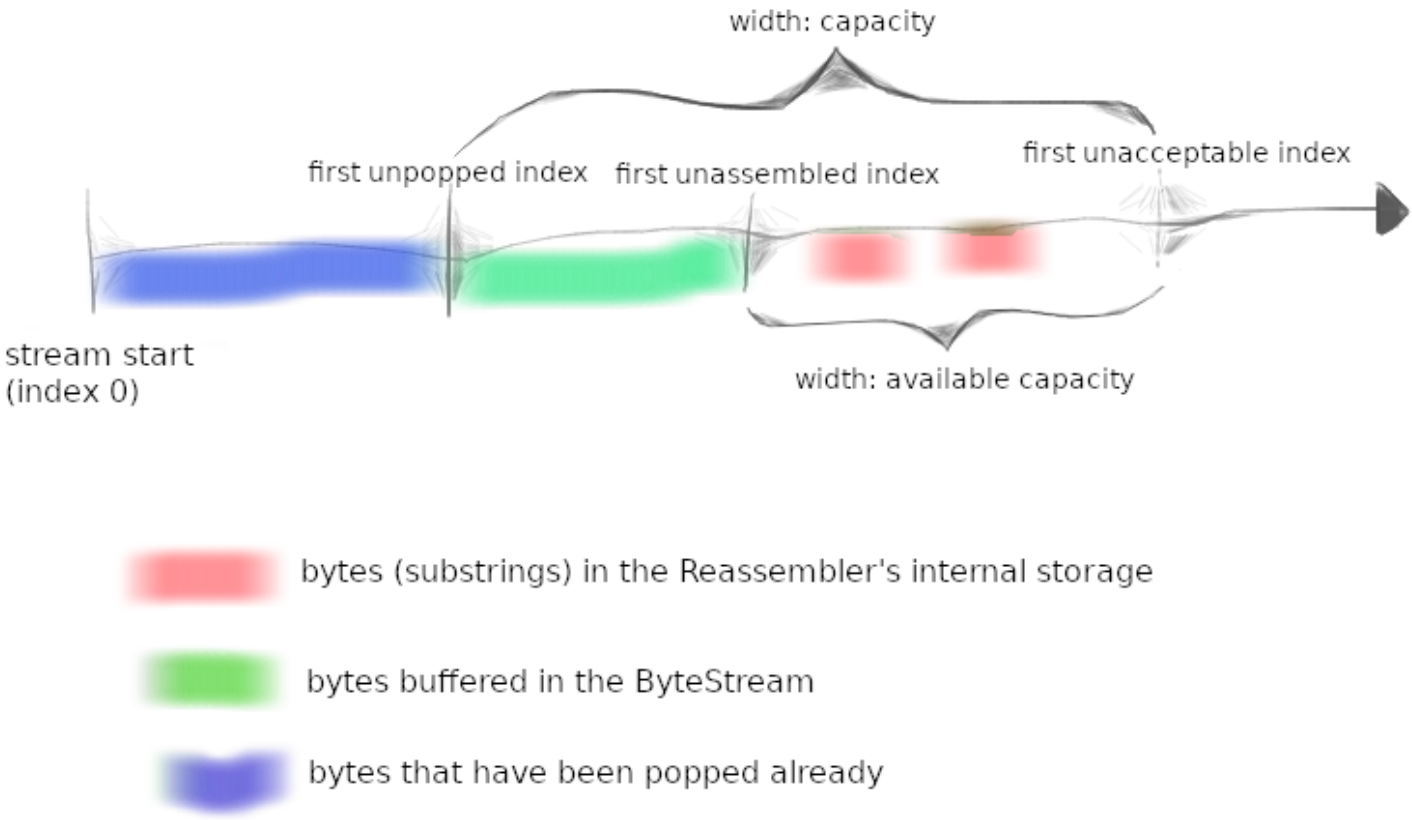
2. Bytes that fit within the stream's available capacity but can't yet be written, because earlier bytes remain unknown. These should be stored internally in the `Reassembler`.

3. Bytes that lie beyond the stream's available capacity. These should be discarded. The `Reassembler`'s will not store any bytes that can't be pushed to the ByteStream either immediately, or as soon as earlier bytes become known.

The goal of this behavior is to **limit the amount of memory** used by the `Reassembler` and `ByteStream`, no matter how the incoming substrings arrive. We've illustrated this in the picture below. The "capacity" is an upper bound on *both*:

1. The number of bytes buffered in the reassembled `ByteStream` (shown in green), and

2. The number of bytes that can be used by "unassembled" substrings (shown in red)



You may find this picture useful as you implement the `Reassembler` and work through the tests—it's not always natural what the "right" behavior is.

3.2 FAQs

- *What is the index of the first byte in the whole stream?* Zero.
- *How efficient should my implementation be?* The choice of data structure is again important here. Please don't take this as a challenge to build a grossly space- or time-inefficient data structure—the Reassembler will be the foundation of your TCP implementation. You have a lot of options to choose from.

  We have provided you with a benchmark; anything greater than 0.1 Gbit/s (100 megabits per second) is acceptable. A top-of-the-line Reassembler will achieve 10 Gbit/s.
- *How should inconsistent substrings be handled?* You may assume that they don't exist. That is, you can assume that there is a unique underlying byte-stream, and all substrings are (accurate) slices of it.
- *What may I use?* You may use any part of the standard library you find helpful. In particular, we expect you to use at least one data structure.

- *When should bytes be written to the stream?* As soon as possible. The only situation in which a byte should not be in the stream is that when there is a byte before it that has not been "pushed" yet.
- *May substrings provided to the* `insert()` *function overlap?* Yes.
- *Will I need to add private members to the Reassembler?* Yes. Substrings may arrive in any order, so your data structure will have to "remember" substrings until they're ready to be put into the stream—that is, until all indices before them have been written.
- *Is it okay for our re-assembly data structure to store overlapping substrings?* No. It is possible to implement an "interface-correct" reassembler that stores overlapping substrings. But allowing the re-assembler to do this undermines the notion of "capacity" as a memory limit. If the caller provides redundant knowledge about the same index, the `Reassembler` should only store one copy of this information.
- *Will the Reassembler ever use the Reader side of the ByteStream?* No—that's for the external customer. The Reassembler uses the Writer side only.
- *How many lines of code are you expecting?* When we run `./scripts/lines-of-code` on the starter code, it prints:

```
ByteStream:    82 lines of code
Reassembler:   26 lines of code
```

  and when we run it on our solutions, it prints:

```
ByteStream:   111 lines of code
Reassembler:   85 lines of code
```

  So a reasonable implementation of the `Reassembler` might be about 50–60 lines of code for the `Reassembler` (on top of the starter code).
- *More FAQs*: For more, please see https://cs144.github.io/lab_faq.html.

4 Development and debugging advice

1. You can test your code (after compiling it) with `cmake --build build --target check1`.
2. Please re-read the section on "using Git" in the Lab 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.
3. Please work to make your code readable to the CA who will be grading it for style and soundness. 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.

4. Please also keep to the "Modern C++" style described in the Checkpoint 0 document. The cppreference website (https://en.cppreference.com) is a great resource, although you won't need any sophisticated features of C++ to do these labs. (You may sometimes need to use the `move()` function to pass an object that can't be copied.)

5. If you get your builds stuck and aren't sure how to fix them, you can erase your `build` directory (`rm -rf build`—please be careful not to make a typo as this will erase whatever you tell it), and then run `cmake -S . -B build` again.

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 check1` (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/check1.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) **Structure and Design.** Describe the high-level structure and design choices embodied in your code. You don't 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. What data structures did you choose in your header file? Are any of them not *strictly* necessary? We'd like you to avoid redundant state if at all possible, unless you think and can justify that there's a serious performance penalty from doing so. 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) **Alternative design choices** that you considered or ideally evaluated in terms of their performance, difficulty to write (e.g., hours required to produce a bug-free implementation), difficulty to read (e.g., lines of code and their degree of subtlety or nonobvious correctness), and any other dimensions you think are interesting for the reader (or for your own past self before you did this assignment). Include any measurements if applicable.

(c) **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?

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

4. In your writeup, please also fill in the number of hours the assignment took you and any other comments.

5. The mechanics of "how to turn it in" will be announced before the deadline.

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