Lab Checkpoint 1: stitching substrings into a byte stream

Due: Thursday, October 1**0, 23:59.**

Collaboration Policy: Link in QQ Group.

0 Overview

For Checkpoint 0, you used an *Internet stream socket* to fetch information 1from 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.

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. Run `git remote add upstream https://github.com/CS144/minnow.git`, then run `git branch -r |grep upstream`, you will see:

```
liyu@Ubuntu22:~/CS144$ git branch -r |grep upstream
  upstream/check-unsanitized
  upstream/check1-startercode
  upstream/check2-startercode
  upstream/check3-startercode
  upstream/check4-startercode
  upstream/check5-startercode
  upstream/check6-startercode
  upstream/check7-startercode
  upstream/main
```

3. Download the starter code for Checkpoint 1 by running `git merge upstream/check1-startercode`.
4. Make sure your build system is properly set up: `cmake -S . -B build`.
5. Compile the source code: `cmake --build build`.

2 Putting substrings in sequence

As part of the lab assignment, you will implement 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?
uint64_t 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.

2.1 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.

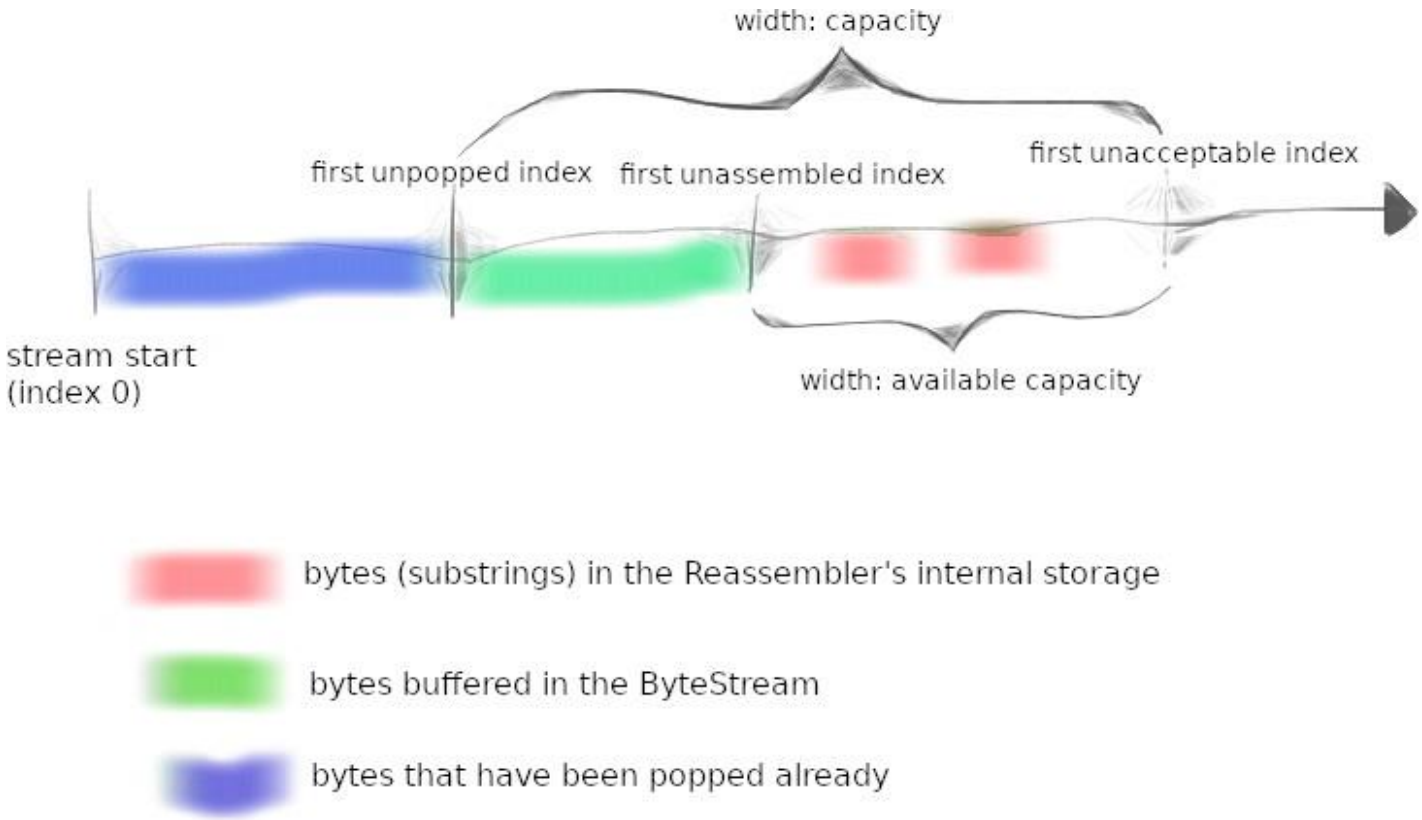
2. Bytes that fit within the stream's available capacity but can't yet be written, because earlier bytes remain unknown. These should be stored internally in the Reassembler.

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 thetests—it's not always natural what the "right" behavior is.

2.2 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).

3 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 TA 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.

4 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 tity (suggests improvements to follow good C++ programming practices)

3. 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.
4. Write a report in minnow/. This file should be a 3-5 pages pdf 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. We encourage discussions of alternatives considered or evaluated, and benefits of weaknesses of your design compared with alternatives – perhaps in terms of simplicity/complexity, risk of bugs, asymptotic performance, empirical performance, required implementation time and difficulty, and other factors. Include any

(b) measurements if applicable. You are strongly encouraged to make this writeup as readable as possible by using subheadings and outlines.

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

(e) **Experimental results and performance.**

5. Compress the code into a zip file and submit it in the QQ group.