Lab 2: stitching substrings into a byte stream

Due: To be determined

0 Collaboration Policy

The programming assignments must be your own work: You must write all the code you hand in for the programming assignments, except for the code that we give you as part of the assignment. Please do not copy-and-paste code from Stack Overflow, GitHub, or other sources. If you base your own code on examples you find on the Web or elsewhere, cite the URL in a comment in your submitted source code.

Working with others: You may not show your code to anyone else, look at anyone else's code, or look at solutions from previous years. You may discuss the assignments with other students, but do not copy anybody's code. If you discuss an assignment with another student, please name them in a comment in your submitted source code.

1 Overview

In Lab 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 next four weeks, you'll implement TCP, 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.
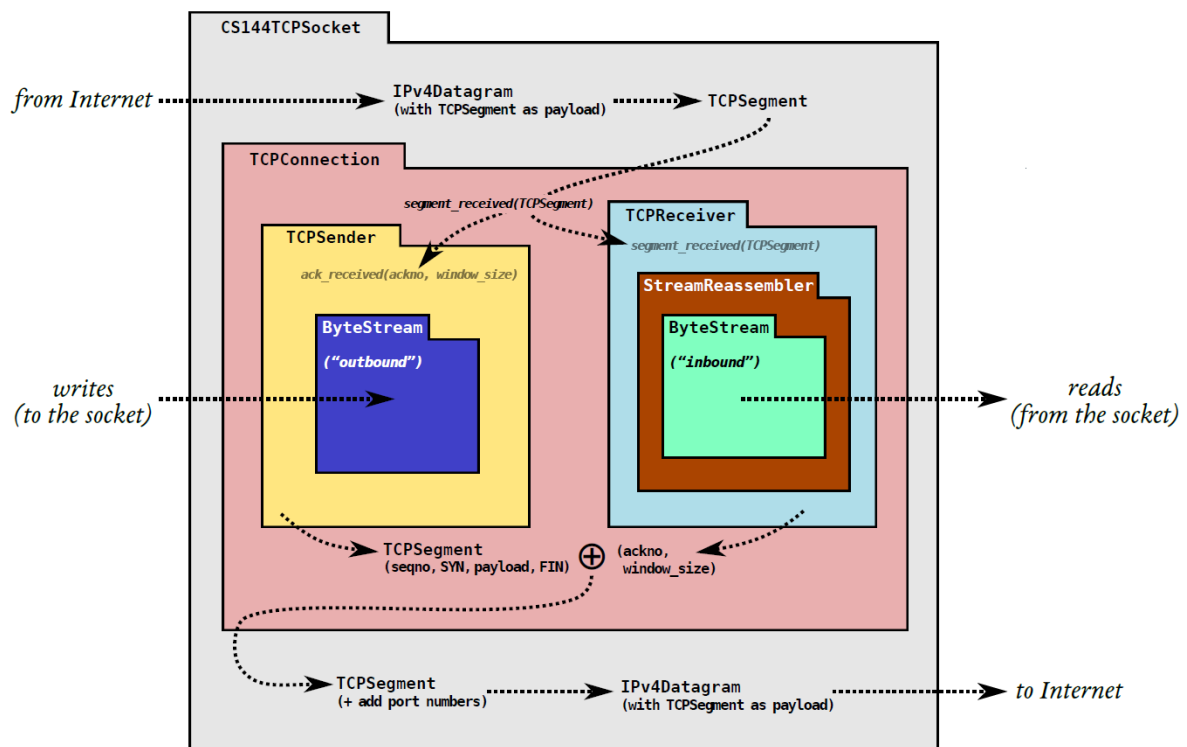
Figure 1: The arrangement of modules and dataflow in your TCP implementation. The
ByteStream is a part of Lab 2. The job of TCP is to convey two **ByteStreams** (one in each
direction) over an unreliable datagram network, so that bytes written to the socket on one side of
the connection emerge as bytes that can be read at the peer, and vice versa. Lab 2 is the
ByteStream and the **StreamReassembler**, and in Labs 3, 4, and 5 you'll implement the
TCPReceiver, **TCPSender**, and then the **TCPConnection** to tie it all together.

The lab assignments will ask you to build up a TCP implementation in a modular way. Figure 1
shows how the pieces fit together.

1. In Lab 2, you'll implement a *stream reassembler*--a module that stitches small pieces of the
   byte stream (known as substrings, or **segments**) back into a contiguous streamda of bytes in
   the correct sequence.
2. In Lab 3, you'll implement the part of TCP that handles the inbound byte-stream: the
   TCPReceiver. This involves thinking about how TCP will represent each byte's place in the
   stream--known as a "sequence number." The **TCPReceiver** is responsible for telling the
   sender (a) how much of the inbound byte stream it's been able to assemble successfully (this
   is called "acknowledgment") and (b) how many more bytes the sender is allowed to send
   right now ("flow control").
3. In Lab 4, you'll implement the part of TCP that handles the outbound byte-stream: the
   TCPSender. How should the sender react when it suspects that a segment it transmitted was
   lost along the way and never made it to the receiver? When should it try again and re-
   transmit a lost segment?
4. In Lab 5, you'll combine your work from the previous to labs to create a working TCP
   implementation: a **TCPConnection** that contains a **TCPSender** and **TCPReceiver**. You'll use
   this to talk to real servers around the world.

2 Getting started

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

1. Make sure you have committed all your solutions to Lab 0. Now clone the Lab 2 repository to
   your local workspace. Then make your changes to files in Lab0 be synchronized to Lab2

project. Specifically, you should copy the source code in file `webget.cc` from your Lab0 repository to the file `webget.cc` of Lab2 project, so that previous tests that passed on Lab0 can also passed on Lab2 after synchronization.

2. Just as Lab0, run `mkdir build`, `cd build`, `cmake ..` and `make` one by one.

3. Outside the `build` directory, open and start editing the `writeups/lab2.md` file. This is the template for your lab writeup and will be included in your submission.

3 An in-memory reliable byte stream

By now, you've seen how the abstraction of a reliable byte stream can be useful in communicating across the Internet, even though the Internet itself only provides the service of "best-effort" (unreliable) datagrams.

To finish off this week's lab, you will implement, in memory on a single computer, an object that provides this abstraction. Bytes are written on the "input" side and can be read, in the same sequence, from the "output" side. The byte stream is finite: the writer can end the input, and then no more bytes can be written. When the reader has read to the end of the stream, it will reach "EOF" (end of file) and no more bytes can be read.

Your byte stream will also be *flow-controlled* to limit its memory consumption at any given time. The object is initialized with a particular "capacity": the maximum number of bytes it's willing to store in its own memory at any given point. The byte stream will limit the writer in how much it can write at any given moment, to make sure that the stream doesn't exceed its storage capacity. As the reader reads bytes and drains them from the stream, the writer is allowed to write more. Your byte stream is for use in a single thread —— you don't have to worry about concurrent writers/readers, locking, or race conditions.

To be clear: the byte stream is finite, but it can be *almost arbitrarily long* before the writer ends the input and finishes the stream. Your implementation must be able to handle streams that are much longer than the capacity. The capacity limits the number of bytes that are held in memory (written but not yet read) at a given point, but does not limit the length of the stream. An object with a capacity of only one byte could still carry a stream that is terabytes and terabytes long, as long as the writer keeps writing one byte at a time and the reader reads each byte before the writer is allowed to write the next byte.

Here's what the interface looks like for the writer:

```
// Write a string of bytes into the stream. Write as many
// as will fit, and return the number of bytes written.
size_t write(const std::string &data);

// Returns the number of additional bytes that the stream has space for
size_t remaining_capacity() const;

// Signal that the byte stream has reached its ending
void end_input();

// Indicate that the stream suffered an error
void set_error();
```

And here is the interface for the reader:

```
// Peek at next "len" bytes of the stream
std::string peek_output(const size_t len) const;
```

```
    // Remove "len" bytes from the buffer
    void pop_output(const size_t len);

    // Read (i.e., copy and then pop) the next "len" bytes of the stream
    std::string read(const size_t len);

    bool input_ended() const; // "true" if the stream input has ended
    bool eof() const; // "true" if the output has reached the ending
    bool error() const; // "true" if the stream has suffered an error
    size_t buffer_size() const; // the maximum amount that can currently be
    peeked/read
    bool buffer_empty() const; // "true" if the buffer is empty

    size_t bytes_written() const; // Total number of bytes written
    size_t bytes_read() const; // Total number of bytes popped
```

Please open the **libsponge/byte stream.hh** and **libsponge/byte_stream.cc** files, and implement an object that provides this interface. As you develop your byte stream implementation, you can run the automated tests with `make check_lab2_1`.

4 Putting substrings in sequence

In this and the next lab, 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 Lab 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 **StreamReassembler**. 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. The **StreamReassembler** will own a **ByteStream** for the output: as soon as the reassembler knows the next byte of the stream, it will write it into the **ByteStream**. The owner can access and read from the **ByteStream** whenever it wants.

Here's what the interface looks like:

```
    // Construct a 'StreamReassembler' that will store up to 'capacity' bytes.
    StreamReassembler(const size_t capacity);

    // Receive a substring and write any newly contiguous bytes into the stream,
    // while staying within the memory limits of the 'capacity'. Bytes that would
    // exceed the capacity are silently discarded.
    //
    // 'data': the substring
    // 'index': indicates the index (place in sequence) of the first byte in 'data'
    // 'eof': the last byte of this substring will be the last byte in the entire
    // stream
    void push_substring(const string &data, const uint64_t index, const bool eof);

    // Access the reassembled ByteStream (your code from section 3)
    ByteStream &stream_out();
```

```
// The number of bytes in the substrings stored but not yet reassembled
size_t unassembled_bytes() const;

// Is the internal state empty (other than the output stream)?
bool empty() const;
```

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 much easier.

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

4.1 What's the "capacity"?

Your **push_substring** method will ignore any portion of the string that would cause the **StreamReassembler** to exceed its "capacity": a limit on memory usage, i.e. the maximum number of bytes it is ever allowed to store. This prevents the reassembler from using an unbounded amount of memory, no matter what the TCP sender decides to do. We've illustrated this in the picture below. The "capacity" is an upper bound on both:

1. The number of bytes in the reassembled **ByteStream** (shown in green below), and
2. The maximum number of bytes that can be used by "unassembled" substrings (shown in red)
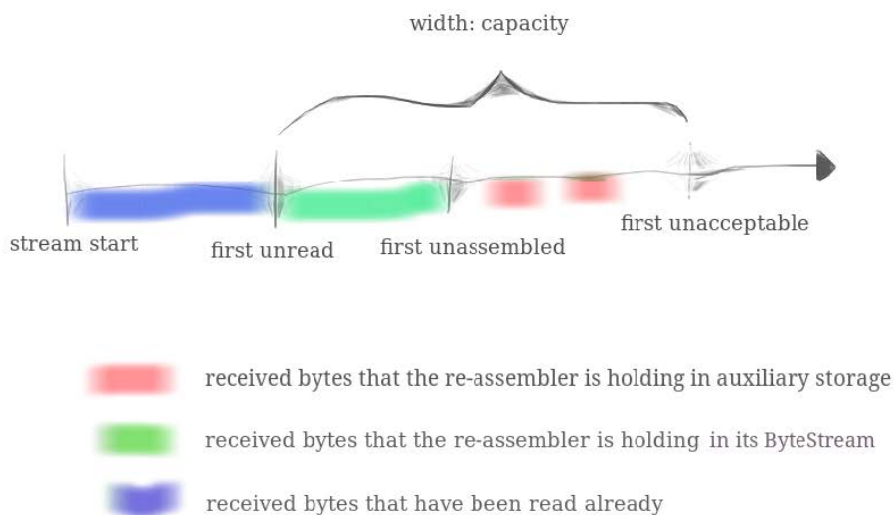
You may find this picture useful as you implement the **StreamReassembler** and work through the tests--it's not always natural what the "right" behavior is.

4.2 FAQs

- What is the index of the first byte in the whole stream? Zero.
- How efficient should my implementation be? Please don't take this as a challenge to build a grossly space- or time-inefficient data structure--this data structure will be the foundation of your TCP implementation. A ballpark expectation would be that each of the new Lab 2 tests can complete in less than half a second.
- 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 *push_substring()* function overlap? Yes.
- Will I need to add private members to the StreamReassembler? 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 reassembler to do this undermines the notion of "capacity" as a memory limit. We'll consider the storage of overlapping substrings to be a style violation when grading.

5 Development and debugging advice

1. You can test your code (after compiling it) with `make check_lab2_2` .
2. Please also keep to the "Modern C++" style described in the Lab 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.)
3. If you get a segmentation fault, something is really wrong! We would like you to be writing in a style where you use safe programming practices to make segfaults extremely unusual (no malloc(), no new, no pointers, safety checks that throw exceptions where you are uncertain, etc.).

6 Submit

1. In your submission, please only make changes to the **.hh** and **.cc** files in the top level of **libsponge**. 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` (to make sure the code compiles)
- (b) `make check_lab2_2` (to make sure the automated tests pass)

3. Write a report in `writeups/lab2.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:

  - **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.

- **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?
- **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. When ready to submit, please follow the instructions in Our QQ group. Please make sure you have committed everything you intend before submitting. We can only grade your code if it has been committed.

6. Please let the course staff know ASAP of any problems at the Friday-afternoon lab session, or by posting a question on QQ group. Good luck and welcome to NJU networking lab!