**Attribution:** This series of programming assignments, including codebase and documentation, is adopted from 'Sponge' of Stanford CS144 Introduction to Computer Networking by Prof. Keith Winstein https://cs144.github.io/.

# Programming Assignment 1: Stitching Substrings into a byte stream

Due: March 10, 2024. 23:59 pm

Late deadline: March 11, 2024. 23:59 pm (20% penalty)

## 1. Overview

*Suggestion: read the whole assignment document before implementing.*

In Assignment 0, you used an *Internet stream socket* to fetch information from a website, 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 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 kindsof 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 assignments will ask you to build up a TCP implementation in a modular way. Remember the `ByteStream` you just implemented in Assignment 0? In the next assignments, 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. Figure 1 shows how the pieces fit together.

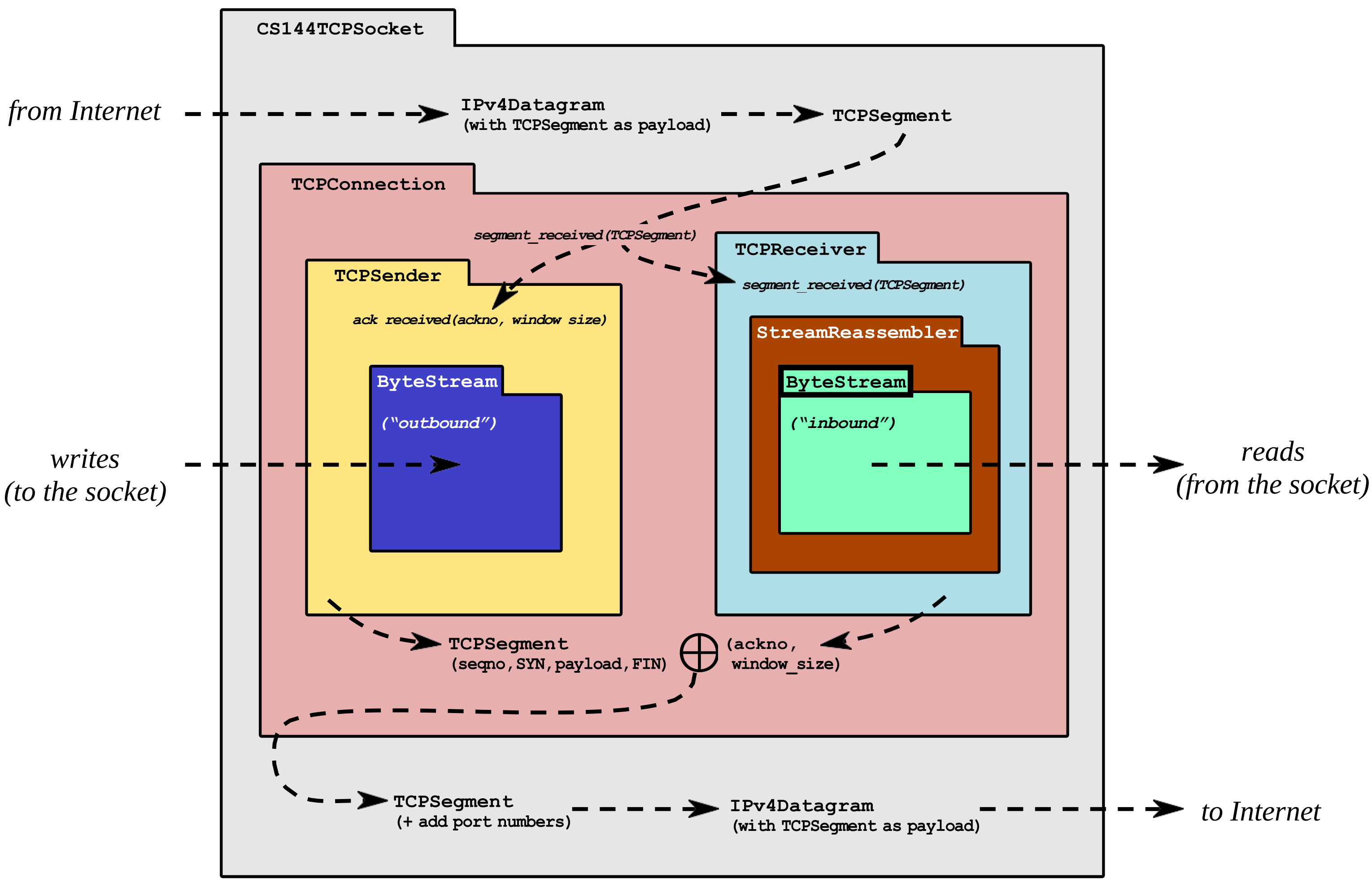



**Figure 1**: The arrangement of modules and dataflow in your TCP implementation. The `ByteStream` was Assignment 0. The job of TCP is to convey two `ByteStreams` (one in each direction) over an unreliable datagram network, so that bytes written to the socket on one side of the connection emerge as bytes that can be read at the peer, and vice versa. Assignment 1 is the `StreamReassembler`, and in Assignments 2, 3, and 4 you'll implement the `TCPReceiver`, `TCPSender`, and then the `TCPConnection` to tie it all together.

1. In Assignment 1, you'll implement a *stream reassembler* —a module that stitches small pieces of the byte stream (known as substrings, or **segments**) back into a continuous stream of bytes in the correct sequence.
2. In Assignment 2, 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 Assignment 3, 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 Assignment 4, 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 Assignment 0, with additional classes and test. To get started:

1. Make sure you have committed all your solutions to Assignment 0. Please don't modify any files outside the top level of the `libsponge` directory, or `webget.cc`. You may have trouble merging the Assignment 1 starter code otherwise.
2. While inside the repository for the assignments, run `git fetch` to retrieve the most recent version of the assignments.
3. Download the starter code for Assignment 1 by running `git merge origin/lab1-startercode`.
4. Within your `build` directory, compile the source code: `make` (you can run, e.g., `make -j4` to use four processors when compiling).
5. Outside the `build` directory, open and start editing the `writeups/assn1.md` file. This is the template for your assignment writeup and will be included in your submission.

## 3. Putting substrings in sequence

In this and the next 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 Assignment 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 assignment 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 Assignment 0)
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;
```

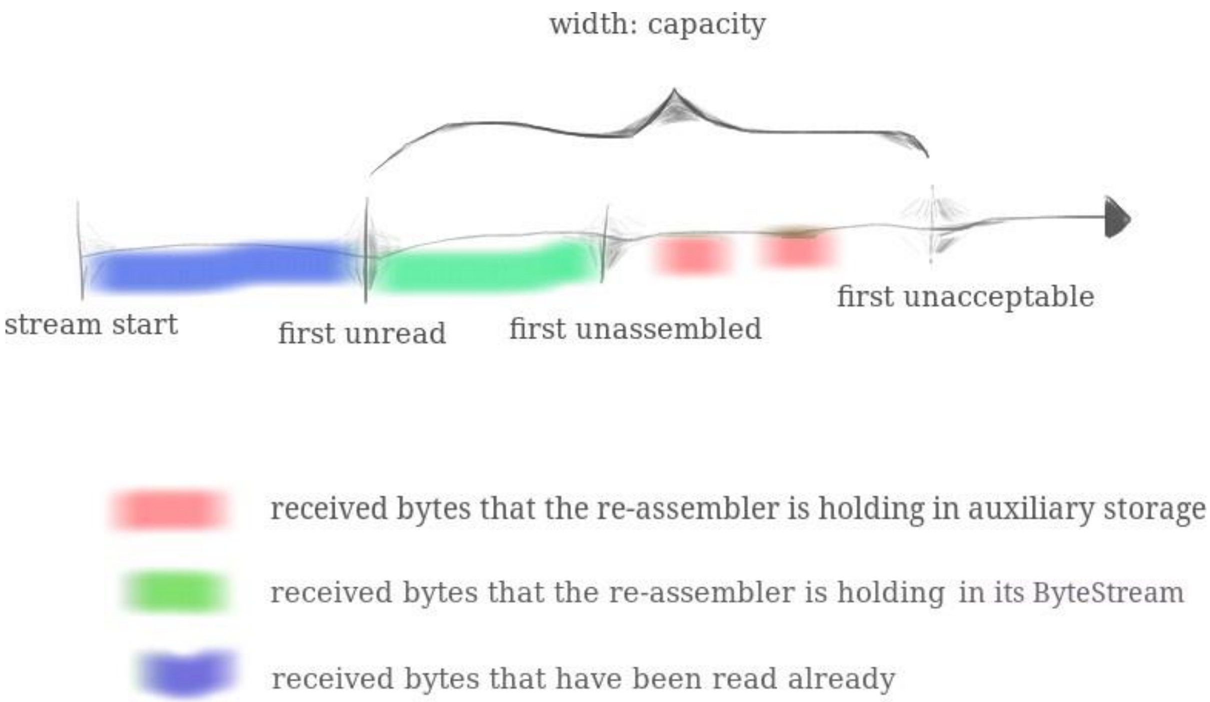
***Why am I doing this?*** TCP robustness against reordering and duplication comes from its ability to stitch arbitrary excerpts of the byte stream back into the original stream. Implementing this in a discrete testable module will make handling incoming segments *much* easier.

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.

## 3.1 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.

## 3.2 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 Assignment 1 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 re-assembler 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.
- *More FAQs*: For more, please see http://tomahawk.postech.ac.kr/csed353/assignments/faq.

# 4. Development and debugging advice

1. You can test your code (after compiling it) with `make check_lab1`.
2. Please re-read the section on "using Git" in the Assignment 0 document, and remember to keep the code in the Git repository it was distributed in on the `master` 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. 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 Assignment 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 assignments. (You may sometimes need to use the `move()` function to pass an object that can't be copied.)
5. 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). That said, to debug you can configure your build directory with `cmake .. -DCMAKE_BUILD_TYPE=RelASan` to enable the compiler's "sanitizers" to detect memory errors and undefined behavior and give you a nice diagnostic about when they occur. You can also use the `valgrind` tool. You can also configure with

`cmake .. -DCMAKE_BUILD_TYPE=Debug` and use the GNU debugger (`gdb`). But please remember that these options (especially the sanitizers) will slow down compilation and execution—you don't want to accidentally leave them in place!

6. You can reset the build system with `make clean` and `cmake .. -DCMAKE_BUILD_TYPE=Release`. Or if you get your builds really 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), make a new `build` directory, and `cmake ..` again.
7. This assignment may require you to perform string concatenations many times. C++ `string` offers multiple ways to concatenate strings, which can be found in https://en.cppreference.com/w/cpp/string/basic_string. Among them, note that `operator+=()` or `append()` often outperform `operator+()` in terms of speed by orders of magnitude. The formers often reuse the buffer of `this` (by appending) and thus a memory allocation may not occur every time. The latter creates a new string and thus a memory allocation occurs every time.

## 5. Submit

1. In your submission, please only make changes to `.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 format` (to normalize the coding style)
   (b) `make` (to make sure the code compiles)
   (c) `make check_lab1` (to make sure the automated tests pass)
3. Write a report in `writeups/assn1.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 a 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. In your writeup, please also fill in the number of hours the assignment took you and any other comments.

5. When you have finished your implementation, do not forget to **commit all changes you made!** This **should be done before creating an archive** of your git repository. If you create an archive before committing changes, the archive will include only part or none of the changes you made. This may result in a lower score than you anticipated, or worse zero.
   If you are unsure whether your archive includes all changes you made, please refer to our Q&A item: “How do I make sure the created bundle includes all my commits?”

6. To archive your git repository, use the following command from the git working directory on your VM:

   ```
   git bundle create /tmp/<your_student_id>.git --all
   ```

   **Important**: This command will create an archive of your git history, which means that your solution must be committed to your git repository! In addition, we will be grading only the master branch of your repository, so please be sure that branch corresponds to the solution you want to turn in.

7. Please upload your archive (bundle) file to the proper assignment entry on PLMS.