Lab 2: the TCP receiver

Due: Friday, October 11, 5 p.m.
Lab session: Tuesday, October 8, 7:30–10 p.m. in STLC114

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. Please refer to the course administrative handout for more details, and ask on Piazza if anything is unclear.

Piazza: Please feel free to ask questions on Piazza, but please don't post any source code.

1 Overview

In Lab 0, you implemented the abstraction of a *flow-controlled byte stream* (`ByteStream`).

In Lab 1, you created a module that accepts a series of substrings, all excerpted from the same byte stream, and reassembles them back into the original `ByteStream`, while limiting its memory consumption to a given amount (the `capacity`).

Now, in Lab 2, you'll implement the part of TCP that handles the inbound byte-stream: the `TCPReceiver`. **You've already done most of the "algorithmic" work involved in this** when you wrote the `StreamReassembler` and the `ByteStream`; this week is mostly about wiring those classes up to the format of TCP. This will involve 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) what range of bytes the sender is allowed to send right now (the "flow control window").

Next week, in Lab 3, you'll implement the part of TCP that handles the outbound byte-stream: the `TCPSender`. Finally, in Lab 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 a `TCPReceiver` will use the same Sponge library that you used in Labs 0 and 1, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Lab 1. Please don't modify any files outside the top level of the `libsponge` directory, or `webget.cc`. You may have trouble merging the Lab 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 Lab 2 by running `git merge origin/lab2-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/lab2.md` file. This is the template for your lab writeup and will be included in your submission.

3 Lab 2: The TCP Receiver

TCP is a protocol that reliably conveys a pair of flow-controlled byte streams (one in each direction) over unreliable datagrams. Two parties participate in the TCP connection, and *each party* acts as both "sender" (of its own outgoing byte-stream) and "receiver" (of an incoming byte-stream) at the same time. The two parties are called the "endpoints" of the connection, or the "peers."

This week, you'll implement the "receiver" part of TCP, responsible for receiving TCP segments (the actual datagram payloads), reassembling the byte stream (including its ending, when that occurs), and determining that signals that should be sent back to the sender for acknowledgment and flow control.

⋆*Why am I doing this?* These signals are crucial to TCP's ability to provide the service of a flow-controlled, reliable byte stream over an unreliable datagram network. In TCP, **acknowledgment** means, "What's the index of the *next* byte that the receiver needs so it can reassemble more of the `ByteStream`?" This tells the sender what bytes it needs to send or resend. **Flow control** means, "What range of indices is the receiver interested and willing to receive?" (usually as a function of its remaining capacity). This tells the sender how much it's *allowed* to send.

3.1 Sequence Numbers

As a warmup, we'll need to implement TCP's way of representing the index of each byte in the byte stream. Last week you created a `StreamReassembler` that reassembles substrings where each individual byte has a 64-bit index, starting from zero and counting up by one for each byte in the stream. In TCP, each byte's index in the stream is represented with a 32-bit "sequence number" (seqno) that adds a few complexities:

1. **Beginning and ending count as one place in sequence:** In addition to ensuring the receipt of all bytes of data, TCP must ensure that the beginning and ending of the stream are also received. Thus, in TCP the SYN (beginning-of-stream) and FIN (end-of-stream) flags are both assigned sequence numbers. Each of these counts as *one* place in sequence. Each byte of data in the stream also counts as one place in sequence.

2. **32-bit wrapping sequence numbers:** In our `StreamReassembler`, the indices always start at zero and have 64 bits—enough that we don't have to worry about using up all the possible 64-bit indices. However, in TCP, the sequence numbers transmitted are 32 bits and will wrap around if the stream is long enough. (Streams in TCP can be arbitrarily long—there's no limit to the length of a ByteStream that can be sent over TCP. So wrapping is pretty common.)

3. **TCP sequence numbers don't start at zero:** To improve security and avoid confusion between different connections, TCP tries to make sure sequence numbers can't be guessed and are less likely to repeat. So the sequence numbers for a stream don't start at zero. The first sequence number in the stream is generally a *random 32-bit number* called the Initial Sequence Number (ISN). This is the sequence number that represents the SYN (beginning of stream). The rest of the sequence numbers behave normally after that—e.g. the first byte of data will have the sequence number of the ISN+1, and the second byte of data will have the ISN+2, etc.

These sequence numbers are the "real" sequence numbers: the actual 32-bit values transmitted in the header of each TCP segment. It's also sometimes helpful to talk about the concept of an "absolute sequence number" (which always starts at zero and doesn't wrap), and about a "stream index" (what you've been using with your `StreamReassembler`: a number that starts at zero for the first actual byte in the stream).

To make these distinctions concrete, consider the byte stream containing just the three-letter string `'cat'`. If the SYN happened to have seqno $2^{32} - 2$, then the seqnos, absolute seqnos, and stream indices of each byte are:

element	SYN	c	a	t	FIN
seqno	$2^{32} - 2$	$2^{32} - 1$	0	1	2
absolute seqno	0	1	2	3	4
stream index		0	1	2	

The figure shows the three different types of indexing involved in TCP:

Sequence Numbers	**Absolute Sequence Numbers**	**Stream Indices**
• Start at the ISN	• Start at 0	• Start at 0
• Include SYN/FIN	• Include SYN/FIN	• Omit SYN/FIN
• 32 bits, wrapping	• 64 bits, non-wrapping	• 64 bits, non-wrapping
• "seqno"	• "absolute seqno"	• "stream index"

Converting between absolute sequence numbers and stream indices is easy enough—just add or subtract one. Unfortunately, converting between sequence numbers and absolute sequence numbers is a bit harder, and confusing the two can produce tricky bugs. To prevent these bugs systematically, we'll represent sequence numbers with a custom type: `WrappingInt32`[1], and write the conversions between it and absolute sequence numbers (represented with `uint64_t`).

We've defined the type for you and provided some helper functions (see `wrapping_integers.hh`), but you'll implement the conversions in `wrapping_integers.cc`:

1. `WrappingInt32 wrap(uint64_t n, WrappingInt32 isn)`

   Convert absolute seqno → seqno. Given an absolute sequence number (n) and an Initial Sequence Number (isn), produce the (relative) sequence number for n.

2. `uint64_t unwrap(WrappingInt32 n, WrappingInt32 isn, uint64_t checkpoint)`

   Convert seqno → absolute seqno. Given a sequence number (n), the Initial Sequence Number (isn), and an absolute *checkpoint* sequence number, compute the absolute sequence number that corresponds to n that is closest to the checkpoint.

   (Note: A **checkpoint** is required because any seqno corresponds to *many* absolute seqnos. E.g. the seqno "17" corresponds to the absolute seqno of 17, but also $2^{32} + 17$, or $2^{33} + 17$, or $2^{34} + 17$, etc. The checkpoint helps to resolve the ambiguity: it's a recently unwrapped absolute seqno that the user of this class knows is close to the desired unwrapped absolute seqno for *this* seqno. In your TCP implementation, you'll use the index of the last reassembled byte as the checkpoint.)

   Hint: *The cleanest/easiest implementation will use the helper functions provided in* `wrapping_integers.hh`. *The wrap/unwrap operations should preserve offsets—two seqnos that differ by 17 will correspond to two absolute seqnos that also differ by 17. So, try (a) wrapping the checkpoint, (b) computing the differences between the n and the checkpoint in the wrapped space, (c) adding that difference in the unwrapped space, and (d) handling an underflow (negative) result.*

You can test your implementation by running the `WrappingInt32` tests. From the `build` directory, run `ctest -R wrap`.

[1] `WrappingInt32` is an example of a *wrapper type*: a type that contains an inner type (in this case `uint32_t`) but provides a different set of functions/operators.

3.2 What's a "window" of acceptable bytes?

Congrats on getting the wrapping logic right! Before we dive into the `TCPReceiver` implementation, let's discuss the concept of the receiver's flow-control *window*.

The idea here is that the `TCPReceiver` is going to tell the `TCPSender` what contiguous range (or "window") of byte indices it's interested in and willing to accept. (And, let's be clear: your TCPReceiver is going to have your `StreamReassembler` as a member and use it for the heavy lifting. This week is mostly about wiring up the `StreamReassembler` to TCP.)

The receiver does this for flow control. The `TCPReceiver`'s `StreamReassembler` is going to ignore any bytes that would exceed the limit on its memory usage (the capacity). By telling the sender what range of data it's allowed and not allowed to send, the receiver can prevent the sender from wasting effort and cause the sender to send no faster than the application is reading from the reassembled `ByteStream`. (Remember that the `ByteStream` itself can be arbitrarily long—terabytes, exabytes, yottabytes—but the memory usage of the receiver, including the `StreamReassembler` and the `ByteStream` it's reconstructing, are bounded by the capacity. The sender is only going to be allowed to send more of the stream once the application reads and pops some data out of the receiver's ByteStream.)

So the "window" refers to a range of byte indices, or sequence numbers, that are currently acceptable to the receiver.

The **lower** (sometimes called "left") edge of the window is called the *ackno*: it's the index of the first byte that the `TCPReceiver` doesn't already know (and therefore would like to find out). For example, let's say the receiver has gotten a SYN with sequence number 24, and then four bytes of data with sequence numbers 25, 26, 27, and 35. The `ackno` will be 28: it's the sequence number of the first byte that the receiver *hasn't already been able to reassemble.*

The **higher** (or "right") edge of the window is the first index that the `TCPReceiver` is *not* willing to accept.

The **window size** is the higher edge minus the lower edge.

Your receiver is responsible for calculating the `ackno` (lower edge of window) and window size (difference between upper and lower edges of window) at any given point. These will eventually be put in the `TCPSegment`s sent from the receiver to the sender. In practice, your receiver will announce a window size equal to its capacity minus the number of bytes being held in its `StreamReassembler`'s `ByteStream`.

3.3 Implementing the TCP receiver

In the rest of this lab, you'll be implementing the `TCPReceiver`. It will (1) receive segments from its peer, (2) reassemble the `ByteStream` using your `StreamReassembler`, and calculate the (3) acknowledgment number (ackno) and (4) the window size.

First, please review the format of a TCP *segment*. This is the structure of the datagram payload that the two endpoints send each other. The ackno and window size are both fields in the `TCPSegment` structure. Please review the documentation for `TCPSegment` (https://cs144.github.io/doc/lab2/class_t_c_p_segment.html) and `TCPHeader` (https://cs144.github.io/doc/lab2/struct_t_c_p_header.html). You may be interested in the `length_in_sequence_space()` method, which calculates how many sequence numbers a segment occupies (including the fact that SYN and FIN each count as one place in sequence, along with each byte of the payload).

Next, let's talk about the interface that your `TCPReceiver` will provide:

```
// Construct a `TCPReceiver` that will store up to `capacity` bytes
TCPReceiver(const size_t capacity); // implemented for you in .hh file

// Handle an inbound TCP segment
//
// returns `true` if any part of the segment was inside the window
bool segment_received(const TCPSegment &seg);

// The ackno that should be sent to the peer
//
// returns empty if no SYN has been received
//
// This is the beginning of the receiver's window, or in other words,
// the sequence number of the first byte in the stream
// that the receiver hasn't received.
std::optional<WrappingInt32> ackno() const;

// The window size that should be sent to the peer
//
// Formally: this is the size of the window of acceptable indices
// that the receiver is willing to accept. It's the difference between
// (a) the sequence number of the end of the window (the first index that
//     won't be accepted by the receiver) minus
// (b) the sequence number of the beginning of the window (the ackno).
//
// Operationally: it's the capacity minus the number of bytes that the
// TCPReceiver is holding in the byte stream.
size_t window_size() const;

// number of bytes stored but not yet reassembled
size_t unassembled_bytes() const; // implemented for you in .hh file

// Access the reassembled byte stream
ByteStream &stream_out(); // implemented for you in .hh file
```

The `TCPReceiver` is built around your `StreamReassembler`. We've implemented the constructor and the `unassembled_bytes` and `stream_out` methods for you in the `.hh` file. Here's what you'll have to do for the others:

3.3.1 segment_received()

This is the main workhorse method! `TCPReceiver::segment_received()` will be called each time a new segment is received from the peer.

This method needs to:

- **Set the Initial Sequence Number if necessary.** The sequence number of the first-arriving segment that has the SYN flag set is the *initial sequence number*. You'll want to keep track of that in order to keep converting between 32-bit wrapped seqnos/acknos and their absolute equivalents. (Note that the SYN flag is just *one* flag in the header. The same segment could also carry data and could even have the FIN flag set.)
- **Push any data, or end-of-stream marker, to the `StreamReassembler`.** If the `FIN` flag is set in a `TCPSegment`'s header, that means that the stream ends with the last byte of the payload—it's equivalent to an EOF.
- **Determine if any part of the segment falls inside the window.** This method should return `true` if *any* part of the segment fell inside the window, and false otherwise.[2] Here's what we mean by that: A segment occupies a range of sequence numbers—a range starting with its sequence number, and with length equal to its `length_in_sequence_space()` (which reflects the fact that SYN and FIN each count for one sequence number, as well as each byte of the payload). A segment is *acceptable* (and the method should return `true`) if any of the sequence numbers it occupies falls inside the receiver's window.

  There are some special cases:

  - If the ISN hasn't been set yet, a segment is acceptable if (and only if) it has the SYN bit set.
  - If the window has size zero, then its size should be treated as one byte for the purpose of determining the first and last byte of the window.[3]
  - If the segment's length in sequence space is zero (e.g. just a bare acknowledgment with no payload and no SYN or FIN flag), then its size should be treated as one byte for the purpose of determining the first and last sequence number it occupies.

3.3.2 ackno()

Returns an `optional<WrappingInt32>` containing the sequence number of the first byte that the receiver doesn't already know. This is the windows's left edge: the first byte the receiver is interested in receiving. If the ISN hasn't been set yet, return an empty optional.

[2] If false, somehow the TCPSender has gotten confused and needs to be corrected about the window.

[3] This definition of acceptability is a slight adaptation of the definition in RFC 793, pp. 25 through 26.

3.3.3 window_size()

Please see §§ 3.2 for the definition of the window size.

4 Development and debugging advice

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

2. You can test your code (after compiling it) with `make check_lab2`.

3. 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 `master` 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. 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.)

6. 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`).

5 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 format` (to normalize the coding style)

(b) `git status` (to check for un-committed changes—if you have any, commit!)

(c) `make` (to make sure the code compiles)

(d) `make check_lab2` (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:

   (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. In your writeup, 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 at https://cs144.github.io/submit. Please make sure you have committed everything you intend before submitting.

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