Lab Checkpoint 2: the TCP receiver

Due: Tuesday, January 30, 3 p.m.

Collaboration Policy: Same as checkpoint 0. Please do not look at other students' code or solutions to past versions of these assignments. Please fully disclose any collaborators or any gray areas in your writeup—disclosure is the best policy.

0 Overview

Suggestion: read the whole lab document before implementing.

In Checkpoint 0, you implemented the abstraction of a *flow-controlled byte stream* (`ByteStream`). And in Checkpoint 1, you created a `Reassembler` that accepts a sequence of substrings, all excerpted from the same byte stream, and reassembles them back into the original stream.

These modules will prove useful in your TCP implementation, but nothing in them was specific to the details of the Transmission Control Protocol. That changes now. In Checkpoint 2, you will implement the `TCPReceiver`, the part of a TCP implementation that handles the incoming byte stream.

The `TCPReceiver` receives messages from the peer's sender (via the `receive()` method) and turns them into calls to a `Reassembler`, which eventually writes to the incoming `ByteStream`. Applications read from this `ByteStream`, just as you did in Lab 0 by reading from the `TCPSocket`.

Meanwhile, the `TCPReceiver` also generates messages that go back to the peer's sender, via the `send()` method. These "receiver messages" are responsible for telling the sender:

1. the index of the "first unassembled" byte, which is called the "acknowledgment number" or "**ackno**." This is the first byte that the receiver needs from the sender.
2. the available capacity in the output ByteStream. This is called the "**window size**".

Together, the **ackno** and **window size** describe describes the receiver's **window**: a **range of indexes** that the TCP sender is allowed to send. Using the window, the receiver can control the flow of incoming data, making the sender limit how much it sends until the receiver is ready for more. We sometimes refer to the ackno as the "left edge" of the window (smallest index the `TCPReceiver` is interested in), and the ackno + window size as the "right edge" (just beyond the largest index the `TCPReceiver` is interested in).

You've already done most of the algorithmic work involved in implementing the `TCPReceiver` when you wrote the `Reassembler` and `ByteStream`; this lab is about wiring those general classes up to the details of TCP. The hardest part will involve thinking about how TCP will represent each byte's place in the stream—known as a "sequence number."

1 Getting started

Your implementation of a `TCPReceiver` will use the same Minnow library that you used in Checkpoints 0 and 1, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Checkpoint 1. Please don't modify any files outside the top level 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 --all` to retrieve the most recent version of the lab assignment.
3. Download the starter code for Checkpoint 2 by running `git merge origin/check2-startercode`. (If you have renamed the "origin" remote to be something else, you might need to use a different name here, e.g. `git merge upstream/check2-startercode`.)
4. Make sure your build system is properly set up: `cmake -S . -B build`
   - **Note for arm64 (UTM) Mac users:** The g++ 13 "sanitizers" (bug checkers) seem to run *very* slow on arm64. Minnow uses these to run the tests. If you are on an arm64 Mac, please configure `cmake` to use a different compiler:
     `cmake -S . -B build -DCMAKE_CXX_COMPILER=clang++`
5. Compile the source code: `cmake --build build`
6. Open and start editing the `writeups/check2.md` file. This is the template for your lab writeup and will be included in your submission.

2 Checkpoint 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, or "peers," participate in the TCP connection, and *each peer* acts as both "sender" (of its own outgoing byte stream) and "receiver" (of an incoming byte stream) at the same time.

This week, you'll implement the "receiver" part of TCP, responsible for receiving messages from the sender, reassembling the byte stream (including its ending, when that occurs), and determining that messages 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?" (a function of its available capacity). This tells the sender how much it's *allowed* to send.

2.1 Translating between 64-bit indexes and 32-bit seqnos

As a warmup, we'll need to implement TCP's way of representing indexes. Last week you created a `Reassembler` that reassembles substrings where each individual byte has a 64-bit **stream index**, with the first byte in the stream always having index zero. A 64-bit index is big enough that we can treat it as **never overflowing**.[1] In the TCP headers, however, space is precious, and each byte's index in the stream is represented not with a 64-bit index but with a 32-bit "sequence number," or "seqno." This adds three complexities:

1. **Your implementation needs to plan for 32-bit integers to wrap around.** Streams in TCP can be arbitrarily long—there's no limit to the length of a ByteStream that can be sent over TCP. But 2^{32} bytes is only 4 GiB, which is not so big. Once a 32-bit sequence number counts up to $2^{32} - 1$, the next byte in the stream will have the sequence number zero.

2. **TCP sequence numbers start at a random value:** To improve robustness and avoid getting confused by old segments belonging to earlier connections between the same endpoints, TCP tries to make sure sequence numbers can't be guessed and are unlikely to repeat. So the sequence numbers for a stream don't start at zero. The first sequence number in the stream is a *random 32-bit number* called the Initial Sequence Number (ISN). This is the sequence number that represents the "zero point" or the SYN (beginning of stream). The rest of the sequence numbers behave normally after that: the first byte of data will have the sequence number of the ISN+1 (mod 2^{32}), the second byte will have the ISN+2 (mod 2^{32}), etc.

3. **The logical beginning and ending each occupy one sequence number:** In addition to ensuring the receipt of all bytes of data, TCP makes sure that the beginning and ending of the stream are received reliably. Thus, in TCP the SYN (beginning-of-stream) and FIN (end-of-stream) control flags are assigned sequence numbers. Each of these occupies *one* sequence number. (The sequence number occupied by the SYN flag is the ISN.) Each byte of data in the stream also occupies one sequence number. Keep in mind that SYN and FIN aren't part of the stream itself and aren't "bytes"—they represent the beginning and ending of the byte stream itself.

These sequence numbers (**seqnos**) are transmitted in the header of each TCP segment. (And, again, there are two streams—one in each direction. Each stream has separate sequence numbers and a different random ISN.) 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 already been using with your `Reassembler`: an index for each byte in the stream, starting at zero).

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:

[1] Transmitting at 100 gigabits/sec, it would take almost 50 years to reach 2^{64} bytes. By contrast, it takes only a third of a second to reach 2^{32} bytes.

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
- Start at the ISN
- Include SYN/FIN
- 32 bits, wrapping
- "seqno"

Absolute Sequence Numbers
- Start at 0
- Include SYN/FIN
- 64 bits, non-wrapping
- "absolute seqno"

Stream Indices
- Start at 0
- Omit SYN/FIN
- 64 bits, non-wrapping
- "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: `Wrap32`, and write the conversions between it and absolute sequence numbers (represented with `uint64_t`). `Wrap32` 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.

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. `static Wrap32 Wrap32::wrap( uint64_t n, Wrap32 zero_point )`

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

2. `uint64_t unwrap( Wrap32 zero_point, uint64_t checkpoint ) const`

   Convert seqno → absolute seqno. Given a sequence number (the `Wrap32`), the Initial Sequence Number (`zero_point`), and an absolute *checkpoint* sequence number, find the corresponding absolute sequence number that is **closest to the checkpoint.**

   Note: A **checkpoint** is required because any given seqno corresponds to *many* absolute seqnos. E.g. with an ISN of zero, the seqno "17" corresponds to the absolute seqno of 17, but also $2^{32}+17$, or $2^{33}+17$, or $2^{33}+2^{32}+17$, or $2^{34}+17$, or $2^{34}+2^{32}+17$, etc. The checkpoint helps resolve the ambiguity: it's an absolute seqno that the user of this class knows is "in the ballpark" of the correct answer. In your TCP implementation, you'll use the first unassembled index 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.*

   Hint #2: *We're expecting one line of code for wrap, and less than 10 lines of code for unwrap. If you find yourself implementing a lot more than this, it might be wise to step back and try to think of a different strategy.*

You can test your implementation by running the tests: `cmake --build build --target check2`.

2.2 Implementing the TCP receiver

Congratulations on getting the wrapping and unwrapping logic right! We'll shake your hand (or, post-COVID, elbow-bump) if this victory happens at the lab session. In the rest of this lab, you'll be implementing the `TCPReceiver`. It will (1) receive messages from its peer's sender and reassemble the `ByteStream` using a `Reassembler`, and (2) send messages back to the peer's sender that contain the acknowledgment number (ackno) and window size. We're expecting this to take **about 15 lines of code** in total.

First, let's review the format of a TCP "sender message," which contains the information about the `ByteStream`. These messages are sent *from* a `TCPSender` to its peer's `TCPReceiver`:

```
/*
 * The TCPSenderMessage structure contains five fields (minnow/util/tcp_sender_message.hh):
 *
 * 1) The sequence number (seqno) of the beginning of the segment. If the SYN flag is set,
 *    this is the sequence number of the SYN flag. Otherwise, it's the sequence number of
 *    the beginning of the payload.
 *
 * 2) The SYN flag. If set, this segment is the beginning of the byte stream, and the seqno field
 *    contains the Initial Sequence Number (ISN) -- the zero point.
 *
 * 3) The payload: a substring (possibly empty) of the byte stream.
 *
 * 4) The FIN flag. If set, the payload represents the ending of the byte stream.
 *
 * 5) The RST (reset) flag. If set, the stream has suffered an error and the connection
 *    should be aborted.
 */

struct TCPSenderMessage
{
  Wrap32 seqno { 0 };

  bool SYN {};
  std::string payload {};
  bool FIN {};

  bool RST {};

  // How many sequence numbers does this segment use?
  size_t sequence_length() const { return SYN + payload.size() + FIN; }
};
```

The `TCPReceiver` generates its own messages back to the peer's `TCPSender`:

```
/*
 * The TCPReceiverMessage structure contains three fields (minnow/util/tcp_receiver_message.hh):
 *
 * 1) The acknowledgment number (ackno): the *next* sequence number needed by the TCP Receiver.
 *    This is an optional field that is empty if the TCPReceiver hasn't yet received the
 *    Initial Sequence Number.
 *
 * 2) The window size. This is the number of sequence numbers that the TCP receiver is interested
 *    to receive, starting from the ackno if present. The maximum value is 65,535 (UINT16_MAX from
 *    the <cstdint> header).
 *
 * 3) The RST (reset) flag. If set, the stream has suffered an error and the connection
 *    should be aborted.
 */

struct TCPReceiverMessage
{
  std::optional<Wrap32> ackno {};
  uint16_t window_size {};
  bool RST {};
};
```

Your `TCPReceiver`'s job is to receive one of these kinds of messages and send the other:

```
class TCPReceiver
{
public:
  // Construct with given Reassembler
  explicit TCPReceiver( Reassembler&& reassembler ) : reassembler_( std::move( reassembler ) )

  // The TCPReceiver receives TCPSenderMessages from the peer's TCPSender.
  void receive( TCPSenderMessage message );

  // The TCPReceiver sends TCPReceiverMessages to the peer's TCPSender.
  TCPReceiverMessage send() const;

  // Access the output (only Reader is accessible non-const)
  const Reassembler& reassembler() const { return reassembler_; }
  Reader& reader() { return reassembler_.reader(); }
  const Reader& reader() const { return reassembler_.reader(); }
  const Writer& writer() const { return reassembler_.writer(); }

private:
  Reassembler reassembler_;
};
```

2.2.1 receive()

This is method will be called each time a new segment is received from the peer's sender. 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 message could also carry data or have the FIN flag set.)
- **Push any data to the `Reassembler`.** If the `FIN` flag is set in a `TCPSegment`'s header, that means that the last byte of the payload is the last byte of the entire stream. Remember that the `Reassembler` expects stream indexes starting at zero; you will have to unwrap the seqnos to produce these.

3 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 with `cmake --build build --target check2`.
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 `main` 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 your code harder to follow.
5. 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.

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 check2` (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/check2.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. Please let the course staff know ASAP of any problems at the lab session, or by posting a question on Ed. Good luck!

5 Extra Credit

Extra credit will be rewarded for improvements to the test suite. Add a test case to one of the files in the `tests` directory (e.g. `minnow/tests/recv_connect.cc`) that catches a **real bug** that somebody might reasonably make that isn't already caught by the existing test

suite. Please submit your test as a Pull Request (it's okay to make this public) so we can take a look and decide whether to add it to the overall testsuite. (This opportunity will remain open—e.g. if you find a good additional test for the `Reassembler` in week 7, that's great too.)