# Lab 4: the TCP sender

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.

1 Overview

In Labs 2 and 3, you implemented the abstraction of a *flow-controlled byte stream*(**ByteStream**), and the tools that translate from segments carried in unreliable datagrams to an incoming byte stream: the **StreamReassembler** and **TCPReceiver**.

Now, in Lab 4, you'll implement the other side of the connection. The **TCPSender** is a tool that translates from an outgoing byte stream to segments that will become the payloads of unreliable datagrams. Finally, 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 on the Internet.
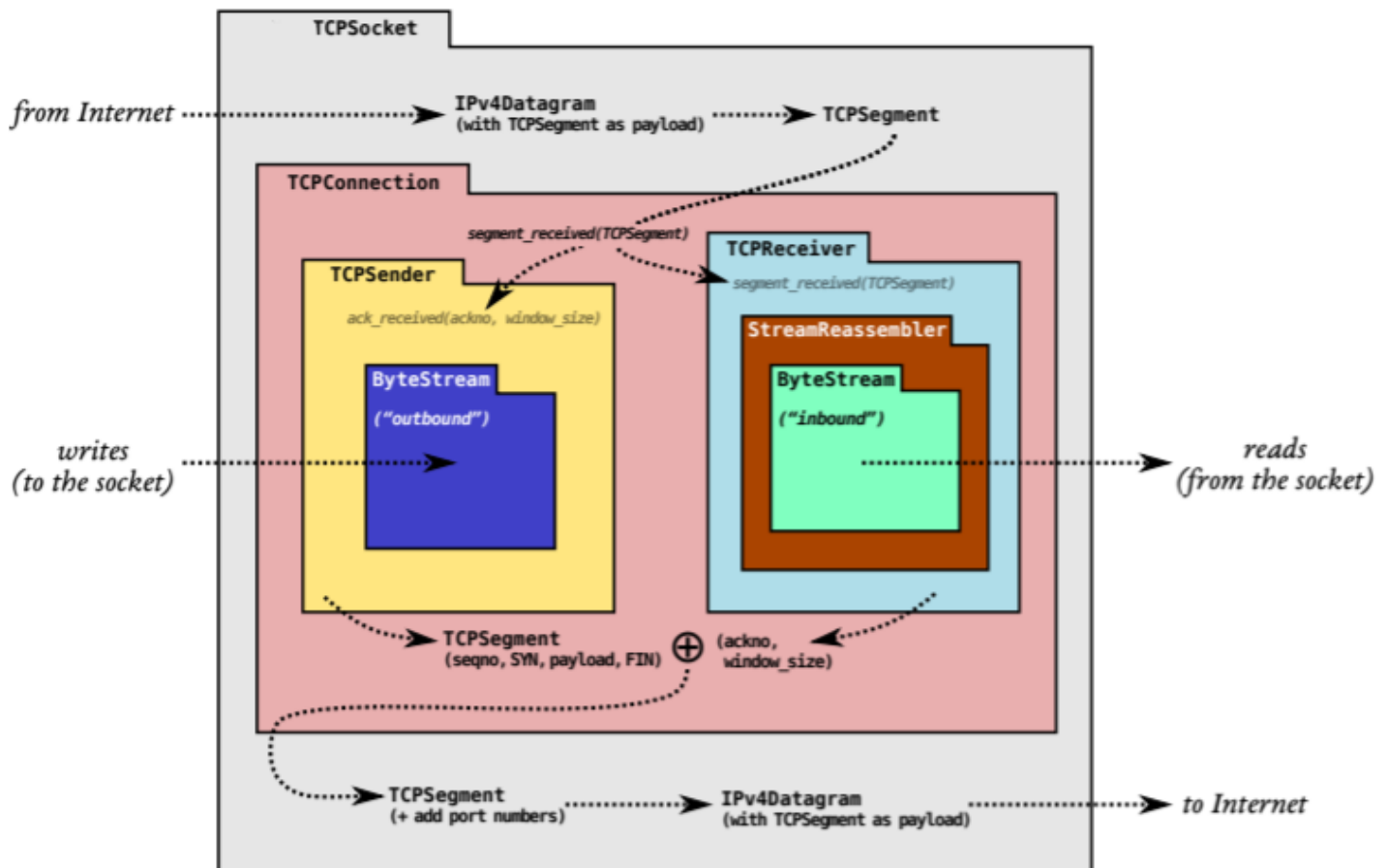
2 Getting started

Your implementation of a **TCPSender** will use the same Sponge library that you used in Labs 0,2,3, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Lab1 , Lab2 and Lab3. Now clone the Lab 3 repository to your local workspace. Then make your changes to files in Lab1 , Lab2 , Lab3 be synchronized to Lab3 project. Specifically, you modified `apps/webget.cc` in Lab1 and `libsponge/byte_stream.cc` , `libsponge/byte_stream.hh` , `libsponge/stream_reassembler.cc` , `libsponge/stream_reassembler.hh` in Lab2 and `libsponge/tcp_receiver.cc` , `libsponge/tcp_receiver.hh` in Lab3

2. Just as Lab1, run `mkdir build` , `cd build` , `cmake ..` and `make` (you can run, e.g., `make -j4` to use four processors when compiling) one by one.

3. Outside the `build` directory, open and start editing the `writeups/lab4.md` file. This is the template for your lab writeup and will be included in your submission.

3 Lab 4: The TCP Sender

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 "sender" part of TCP, responsible for reading from a ByteStream (created and written to by some sender-side application), and turning the stream into a sequence of outgoing TCP segments. On the remote side, a TCP receiver1 transforms those segments (those that arrive—they might not all make it) back into the original byte stream, and sends acknowledgments and window advertisements back to the sender.

The TCP sender and receiver are each responsible for a portion of the TCP segment. The TCP sender writes all of the fields of the **TCPSegment** that were relevant to the **TCPReceiver** in Lab 3: namely, the sequence number, the SYN flag, the payload, and the FIN flag. However, the TCP sender only **reads** the fields in the segment that are written by the receiver: the ackno and the window size. Here is the structure of a TCP segment, highlighting only the fields that will be read by the sender:

It will be your TCPSender's responsibility to:

- Keep track of the receiver's window (processing incoming **acknos** and **window sizes**)
- Fill the window when possible, by reading from the **ByteStream**, creating new TCP segments (including SYN and FIN flags if needed), and sending them. The sender should *keep sending segments* until either the window is full or the **ByteStream** is empty.
- Keep track of which segments have been sent but not yet acknowledged by the receiver— we call these "outstanding" segments
- Re-send outstanding segments if enough time passes since they were sent, and they haven't been acknowledged yet

> Why am I doing this? The basic principle is to send whatever the receiver will allow us to send (filling the window), and keep retransmitting until the receiver acknowledges each segment. This is called "automatic repeat request" (ARQ). The sender divides the byte stream up into segments and sends them, as much as the receiver's window allows. Thanks to your work last

> week, we know that the remote TCP receiver can reconstruct the byte stream as long as it receives each index-tagged byte at least once—no matter the order. The sender's job is to make sure the receiver gets each byte at least once.

3.1 How does the TCPSender know if a segment was lost?

Your **TCPSender** will be sending a bunch of **TCPSegments**. Each will contain a (possibly empty) substring from the outgoing **ByteStream**, indexed with a sequence number to indicate its position in the stream, and marked with the SYN flag at the beginning of the stream, and FIN flag at the end.

In addition to sending those segments, the **TCPSender** also has to *keep track of* its outstanding segments until the sequence numbers they occupy have been fully acknowledged. Periodically, the owner of the **TCPSender** will call the **TCPSender**'s tick method, indicating the passage of time. The **TCPSender** is responsible for looking through its collection of outstanding **TCPSegment**s and deciding if the oldest-sent segment has been outstanding for too long without acknowledgment (that is, without *all* of its sequence numbers being acknowledged). If so, it needs to be retransmitted (sent again).

Here are the rules for what "outstanding for too long" means. You're going to be implementing this logic, and it's a little detailed, but we don't want you to be worrying about hidden test cases trying to trip you up or treating this like a word problem on the SAT. We'll give you some reasonable unit tests this week, and fuller integration tests in Lab 5 once you've finished the whole TCP implementation. As long as you pass those tests 100% and your implementation is reasonable, you'll be fine.

> Why am I doing this? The overall goal is to let the sender detect when segments go missing and need to be resent, in a timely manner. The amount of time to wait before resending is important: you don't want the sender to wait too long to resend a segment (because that delays the bytes flowing to the receiving application), but you also don't want it to resend a segment that was going to be acknowledged if the sender had just waited a little longer—that wastes the Internet's precious capacity

1. Every few milliseconds, your TCPSender's tick method will be called with an argument that tells it how many milliseconds have elapsed since the last time the method was called. Use this to maintain a notion of the total number of milliseconds the TCPSender has been alive. **Please don't try to call any "time" or "clock" functions** from the operating system or CPU—the tick method is your only access to the passage of time. That keeps things deterministic and testable.
2. When the TCPSender is constructed, it's given an argument that tells it the "initial value" of the **retransmission timeout** (RTO). The RTO is the number of milliseconds to wait before resending an outstanding TCP segment. The value of the RTO will change over time, but the "initial value" stays the same. The starter code saves the "initial value" of the RTO in a member variable called **_initial_retransmission_timeout**.
3. You'll implement the retransmission **timer**: an alarm that can be started at a certain time, and the alarm goes off (or "expires") once the RTO has elapsed. We emphasize that this notion of time

passing comes from the **tick** method being called—not by getting the actual time of day.

4. Every time a segment containing data (nonzero length in sequence space) is sent (whether it's the first time or a retransmission), if the timer is not running, **start it running** so that it will expire after RTO milliseconds (for the current value of RTO). By "expire," we mean that the time will run out a certain number of milliseconds in the future.

5. When all outstanding data has been acknowledged, **stop** the retransmission timer.

6. If **tick** is called and the retransmission timer has expired:

    1. Retransmit the *earliest* (lowest sequence number) segment that hasn't been fully acknowledged by the TCP receiver. You'll need to be storing the outstanding segments in some internal data structure that makes it possible to do this.

    2. **If the window size is nonzero:**

        1. Keep track of the number of consecutive retransmissions, and increment it because you just retransmitted something. Your **TCPConnection** will use this information to decide if the connection is hopeless (too many consecutive retransmissions in a row) and needs to be aborted.

        2. Double the value of RTO. This is called "exponential backoff"—it slows down retransmissions on lousy networks to avoid further gumming up the works.

    3. Reset the retransmission timer and start it such that it expires after RTO milliseconds (taking into account that you may have just doubled the value of RTO!).

7. When the receiver gives the sender an **ackno** that acknowledges the successful receipt of *new* data (the **ackno** reflects an absolute sequence number bigger than any previous **ackno**):

    1. Set the RTO back to its "initial value."

    2. If the sender has any outstanding data, restart the retransmission timer so that it will expire after RTO milliseconds (for the current value of RTO).

    3. Reset the count of "consecutive retransmissions" back to zero.

We would suggest implementing the functionality of the retransmission timer in a separate class, but it's up to you. If you do, please add it to the existing files (**tcp_sender.hh** and **tcp_sender.cc**).

3.2 Implementing the TCP sender

Okay! We've discussed the basic idea of *what* the TCP sender does (given an outgoing **ByteStream**, split it up into segments, send them to the receiver, and if they don't get acknowledged soon enough, keep resending them). And we've discussed *when* to conclude that an outstanding segment was lost and needs to be resend.

Now it's time for the concrete interface that your **TCPSender** will provide. There are four important events that it needs to handle, each of which could end up sending a **TCPSegment**:

1. `void fill_window()`

The **TCPSender** is asked to *fill the window*: it reads from its input **ByteStream** and sends as many bytes as possible in the form of **TCPSegments**, *as long as there are new bytes to be read and space available in the window*.

You'll want to make sure that every **TCPSegmen**t you send fits fully inside the receiver's window. Make each *individual* **TCPSegment** as big as possible, but no bigger than the value given by **TCPConfig::MAX_PAYLOAD_SIZE** (1452 bytes).

You can use the **TCPSegment::length_in_sequence_space()** method to count the total number of sequence numbers occupied by a segment. Remember that the SYN and FIN flags also occupy a sequence number each, which means that t_hey occupy space in the window._

> *What should I do if the window size is zero?* If the receiver has announced a window size of zero, the fill window method should act like the window size is one. The sender might end up sending a single byte that gets rejected (and not acknowledged) by the receiver, but this can also provoke the receiver into sending a new acknowledgment segment where it reveals that more space has opened up in its window. Without this, the sender would never learn that it was allowed to start sending again.

1. `void ack_received( const WrappingInt32 ackno, const uint16_t window_size)`

   A segment is received from the receiver, conveying the new left (= ackno) and right (= ackno + window size) edges of the window. The TCPSender should look through its collection of outstanding segments and remove any that have now been fully acknowledged (the ackno is greater than all of the sequence numbers in the segment). The TCPSender should fill the window again if new space has opened up

2. `void tick( const size_t ms_since_last_tick ):`

   Time has passed — a certain number of milliseconds since the last time this method was called. The sender may need to retransmit an outstanding segment

3. `void send_empty_segment():`

   The **TCPSender** should generate and send a **TCPSegment** that has zero length in sequence space, and with the sequence number set correctly. This is useful if the owner (the **TCPConnection** that you're going to implement next week) wants to send an empty **ACK** segment.
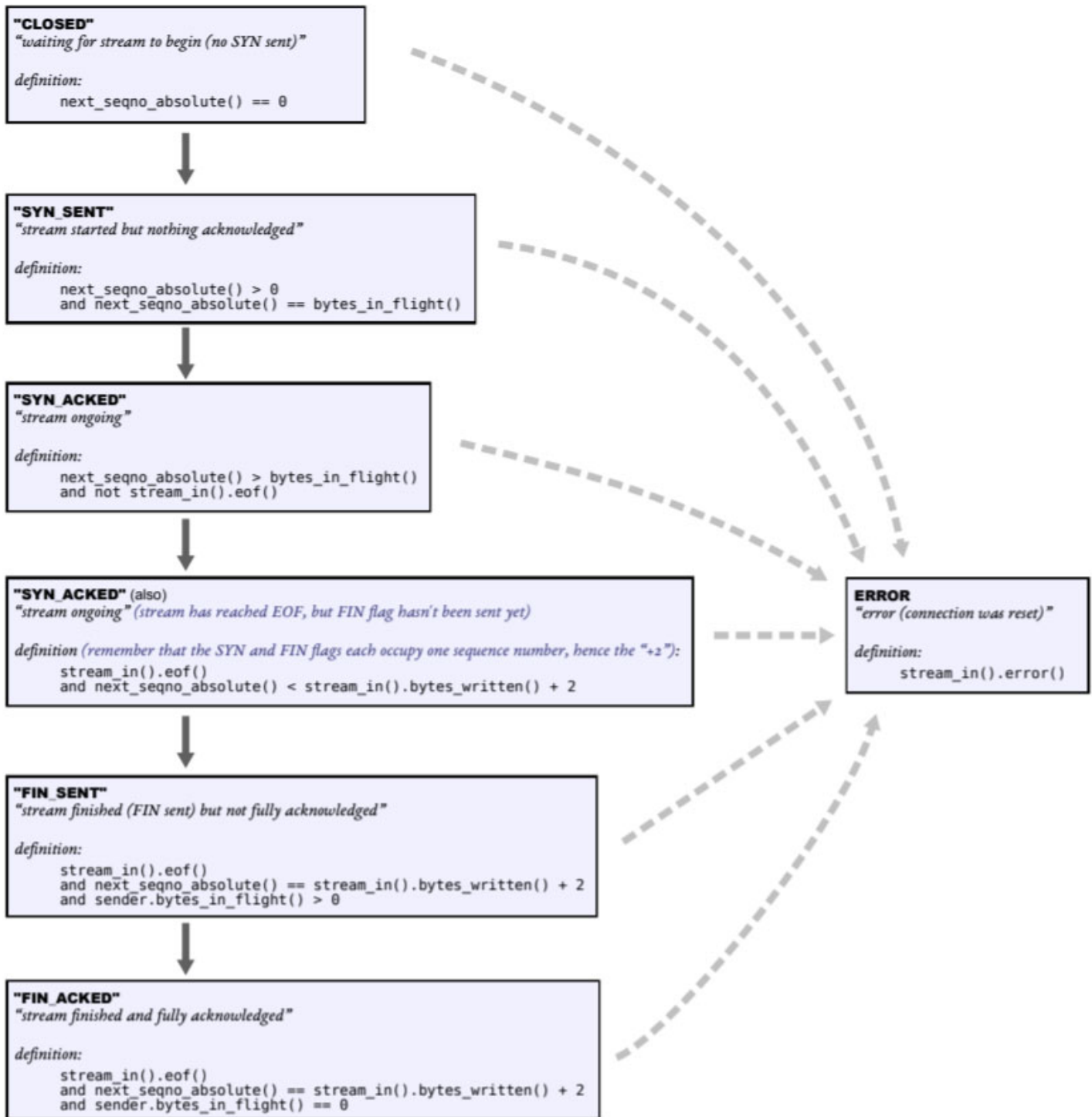
   > Note: a segment like this one, which occupies no sequence numbers, doesn't need to be kept track of as "outstanding" and won't ever be retransmitted.

To complete Lab 4, please review the full interface in the documentation, at
https://cs144.github.io/doc/lab0/index.html, and implement the complete TCPSender public interface

in the tcp sender.hh and tcp sender.cc files. We expect you'll want to add private methods and member variables, and possibly a helper class.

3.3 Theory of testing

To test your code, the test suite will expect it to evolve through a series of situations—from sending the first SYN segment, to sending all the data, to sending the FIN segment, and finally having the FIN segment acknowledged. **We don't think you want to make more state variables** to track these "states"—the states are simply defined by the already-exposed public interface of your **TCPSender** class. But to help you understand the test outputs, here is a diagram of the expected evolution of the **TCPSender** through the life of the stream. (You don't have to worry about the error state or the RST flag until Lab 5.)

Evolution of the TCP sender (as tested by the test suite)

"CLOSED"
"waiting for stream to begin (no SYN sent)"

definition:
```
next_seqno_absolute() == 0
```

"SYN_SENT"
"stream started but nothing acknowledged"

definition:
```
next_seqno_absolute() > 0
and next_seqno_absolute() == bytes_in_flight()
```

"SYN_ACKED"
"stream ongoing"

definition:
```
next_seqno_absolute() > bytes_in_flight()
and not stream_in().eof()
```

"SYN_ACKED" (also)
"stream ongoing" (*stream has reached EOF, but FIN flag hasn't been sent yet*)

definition (remember that the SYN and FIN flags each occupy one sequence number, hence the "+2"):
```
stream_in().eof()
and next_seqno_absolute() < stream_in().bytes_written() + 2
```

ERROR
"error (connection was reset)"

definition:
```
stream_in().error()
```

"FIN_SENT"
"stream finished (FIN sent) but not fully acknowledged"

definition:
```
stream_in().eof()
and next_seqno_absolute() == stream_in().bytes_written() + 2
and sender.bytes_in_flight() > 0
```

"FIN_ACKED"
"stream finished and fully acknowledged"

definition:
```
stream_in().eof()
and next_seqno_absolute() == stream_in().bytes_written() + 2
and sender.bytes_in_flight() == 0
```

3.4 FAQs and special cases

- *How do I "send" a segment?*
  Push it on to the _segments_out queue. As far as your **TCPSender** is concerned, consider it sent as soon as you push it on to this queue. Soon the owner will come along and pop it (using the public **segments_out()** accessor method) and really send it.

- *Wait, how do I both "send" a segment and also keep track of that same segment as being outstanding, so I know what to retransmit later? Don't I have to make a copy of each segment then? Is that wasteful?*
  When you send a segment that contains data, you'll probably want to push it on to the **segments_out** queue and also keep a copy of it internally in a data structure that lets you keep

track of outstanding segments for possible retransmission. This turns out not to be very wasteful because the segment's payload is stored as a reference-counted read-only string (a **Buffer** object). So don't worry about it—it's not actually copying the payload data.

- *What should my **TCPSender** assume as the receiver's window size before I've gotten an ACK from the receiver?*
  One byte.

- *What do I do if an acknowledgment only partially acknowledges some outstanding segment? Should I try to clip off the bytes that got acknowledged?*
  A TCP sender could do this, but for purposes of this class, there's no need to get fancy. Treat each segment as fully outstanding until it's been fully acknowledged—all of the sequence numbers it occupies are less than the **ackno**.

- *If I send three individual segments containing "a," "b," and "c," and they never get acknowledged, can I later retransmit them in one big segment that contains "abc"? Or do I have to retransmit each segment individually?*
  Again: a TCP sender could do this, but for purposes of this class, no need to get fancy. Just keep track of each outstanding segment individually, and when the retransmission timer expires, send the earliest outstanding segment again.

- *Should I store empty segments in my "outstanding" data structure and retransmit them when necessary?*
  No—the only segments that should be tracked as outstanding, and possibly retransmitted, are those that convey some data—i.e. that consume some length in sequence space. A segment that occupies no sequence numbers (no payload, SYN, or FIN) doesn't need to be remembered or retransmitted.

4 Development and debugging advice

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

2. After compiling, you can test your code with `make check_lab4`

3. 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.

4. Please also keep to the "Modern C++" style described in the Lab 1 document. The cpppreference website (https://en.cppreference.com) is a great resource, although you won't need any sophisticated features of C++ to do these labs.

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 ~ 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**). Both of these will slow down your code—don't forget to return to a "Release" build when done.

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:
   - `make` (to make sure the code compiles)
   - `make check_lab4` (to make sure the automated tests pass)
3. Write a report in `writeups/lab4.md` . This file should be a roughly 3-4 pages 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**: **Submit your test screenshots** from `make check_lab4` . 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!