Lab 3: the TCP sender

Due: Friday, October 18, 5 p.m.
Lab session: Tuesday, October 15, 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 Labs 1 and 2, you implemented the tools that translate *from* segments carried in unreliable datagrams *to* an incoming byte stream: the `StreamReassembler` and `TCPReceiver`.

Now, in Lab 3, you'll implement the other side of the connection: a tool that translates *from* an outgoing byte stream *to* segments sent in unreliable datagrams.

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 `TCPSender` will use the same Sponge library that you used in Labs 0–2, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Lab 2. Please don't modify any files outside the top level of the `libsponge` directory, or `webget.cc`. You may have trouble merging the Lab 3 starter code otherwise.

2. While inside the repository for the lab assignments, run $\boxed{\texttt{git fetch}}$ to retrieve the most recent version of the lab assignments.

3. Download the starter code for Lab 3 by running $\boxed{\texttt{git merge origin/lab3-startercode}}$.

4. Within your `build` directory, compile the source code: $\boxed{\texttt{make}}$ (you can run, e.g., $\boxed{\texttt{make -j4}}$ to use four processors when compiling).

5. Outside the `build` directory, open and start editing the `writeups/lab3.md` file. This is the template for your lab writeup and will be included in your submission.

3   Lab 3: 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 receiver[1] 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.

It will be your `TCPSender`'s responsibility to:

- Keep track of the receiver's window (processing incoming **ackno**s and **window size**s)
- Fill the window when possible, by reading from the `ByteStream`, creating new TCP segments (including SYN and FIN flags if needed), and sending them
- 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.

---

[1]It's important to remember that the receiver can be *any* implementation of a valid TCP receiver—it won't necessarily be your own `TCPReceiver`. One of the valuable things about Internet standards is how they establish a common language between endpoints that may otherwise act very differently.

3.1   When should the `TCPSender` conclude that a segment was lost and send it again?

Your `TCPSender` will be sending a bunch of `TCPSegment`s. 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.[2] 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 4 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

---

[2]These are based on a simplified version of the "real" rules for TCP: RFC 6298, recommendations 5.1 through 5.6. The version here is a bit simplified, but your TCP implementation will still be able to talk with real servers on the Internet.

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).

5. When all outstanding data has been acknowledged, turn off the retransmission timer.

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

   (a) 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.

   (b) If the window size is nonzero:

      i. 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.
      ii. Double the value of RTO.[3]

   (c) Start the retransmission timer, such that it expires after RTO milliseconds (for the value of RTO after the doubling operation outlined in the previous bullet point).

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`):

   (a) Set the RTO back to its "initial value."

   (b) 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).

   (c) Reset the count of "consecutive retransmissions" back to zero.

You may want to implement the functionality of the retransmission timer in a separate class—it's up to you. If you do, please add it to the existing files (`tcp_sender.hh` and `tcp_receiver.hh`).

---

[3]This is called "exponential backoff"—it slows down retransmissions on lousy networks to avoid further gumming up the works. We'll learn more about this later in class.

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. 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 TCPSegment 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. Your TCPSender maintains a member variable called _next_seqno that stores the outgoing absolute sequence numbers starting at zero. For every segment you send, you'll want to increment _next_seqno by the segment's length so you know the sequence number to put on the *next* segment.

2. ack_received: An acknowledgment message 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 may need to fill the window again if new space has opened up. This method returns false if the ackno appears invalid, that is, if it's acknowledging data that the sender hasn't sent yet.

3. tick: Time has passed; the TCPSender will check if the retransmission timer has expired and, if so, retransmit the outstanding segment with the lowest sequence number.[4]

4. 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 to _next_seqno. This is useful if the owner (the TCPConnection that you're going to implement next week) wants to send an empty ACK segment. This kind of segment—one that carries no data and occupies no sequence numbers—doesn't need to be kept track of as "outstanding" and won't ever be retransmitted.

To complete Lab 3, please review the full interface in the documentation, at https://cs144. github.io/doc/lab3/class_t_c_p_sender.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.

---

[4]Importantly, this decision to retransmit doesn't have to look at the receiver's window: this segment fell inside the window when it was first sent, and it hasn't been acknowledged, so it's still inside the receiver's window now. The receiver isn't supposed to "shrink" the right edge of the window—you can assume the right edge always stays the same or moves to the right.

3.3   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.

- *The receiver told me its window size was zero bytes. Should I just get stuck and never send any data again?*

  No. If the receiver tells you its window is zero bytes long, save that information as you would any other window advertisement, because it matters for the retransmission behavior described in §§ 3.1. But when it comes time to fill the window, act like the window size is one byte. This is called "zero window probing"—it's a way to periodically probe the receiver to see if it happens to have opened up some more space in the window since the last time we heard from them. The worst thing that can happen is that the receiver will ignore your one-byte segment.[5]

- *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?*

---

[5]In a more production-ready TCP implementation, zero-window-probing behavior is a little more complicated, but not by much.

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. An empty ACK doesn't need to be remembered or retransmitted.

- *Where can I read if there are more FAQs after this PDF comes out?*

  Please check the website (https://cs144.github.io/lab_faq.html) and Piazza regularly.

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. You can test your code (after compiling it) with `make check_lab3`.

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`). Remember to use these settings for debugging only—they dramatically slow down both compilation and execution of your programs. The most reliable/foolproof way to revert to "Release" mode is just to blow away the `build` directory and create a new one.

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_lab3` (to make sure the automated tests pass)

3. Write a report in `writeups/lab3.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. 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!