Lab Checkpoint 3: the TCP sender

Due: Sunday, Oct. 19, 11:59 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`). In Checkpoints 1 and 2, you implemented the tools that translate *from* segments carried in unreliable datagrams *to* an incoming byte stream: the `Reassembler` and `TCPReceiver`.

Now, in Checkpoint 3, you'll implement the other side of the connection. The `TCPSender` is a tool that translates *from* an outbound byte stream *to* segments that will become the payloads of unreliable datagrams. This will complete your implementation of the Transmission Control Protocol (the implementations of which are arguably the world's most prevalent computer program, period). You'll use this to talk to a classmate and to peers across the Internet—real servers that speak TCP.

1 Getting started

Your implementation of a `TCPSender` will use the same Minnow library that you used in Checkpoints 0–2, 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 3 by running `git merge origin/check3-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/check3-startercode`.)
4. Make sure your build system is properly set up: `cmake -S . -B build`
5. Compile the source code: `cmake --build build`
6. Open and start editing the `writeups/check3.md` file. This is the template for your lab writeup and will be included in your submission.

7. Reminder: please make frequent **small commits** in your local Git repository as you work. If you need help to make sure you're doing this right, please ask a classmate or the teaching staff for help. You can use the `git log` command to see your Git history.

2 Checkpoint 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 party participate in the TCP connection, and *each party* is a peer of the other. 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 "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 (receiving incoming `TCPReceiverMessage`s with their **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. The sender should *keep sending segments* until either the window is full or the outbound `ByteStream` has nothing more to send.
- 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.

2.1 How does the `TCPSender` know if a segment was lost?

Your `TCPSender` will be sending a bunch of `TCPSenderMessage`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 `TCPSenderMessage`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 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_RTO_ms_`.

[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.

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:

   (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. This is called "exponential backoff"—it slows down retransmissions on lousy networks to avoid further gumming up the works.

   (c) 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`):

   (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 might choose to implement 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`).

2.2 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:

1. `void push( const TransmitFunction& transmit );`

   The `TCPSender` is asked to *fill the window* from the outbound byte stream: it reads from the stream and sends as many `TCPSenderMessage`s as possible, *as long as there are new bytes to be read and space available in the window.* It sends them by calling the provided `transmit()` function on them.

   You'll want to make sure that every `TCPSenderMessage` you send fits fully inside the receiver's window. Make each *individual* message as big as possible, but no bigger than the value given by `TCPConfig::MAX_PAYLOAD_SIZE`.

   You can use the `TCPSenderMessage::sequence_length()` 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 *they 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 `push` method should pretend 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.
   > **This is the only special-case behavior your implementation should have for the case of a zero-size window.** The `TCPSender` shouldn't actually *remember* a false window size of 1. The special case is only within the push method. Also, N.B. that even if the window size is one (or 20, or 200), the window might still be **full**. A "full" window is not the same as a "zero-size" window.

2. `void receive( const TCPReceiverMessage& msg );`

   A 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).

3. `void tick( uint64_t ms_since_last_tick, const TransmitFunction& transmit );`

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; it can call the `transmit()` function to do this. (Reminder: please don't try to use real-world "clock" or "gettimeofday" functions in your code; the only reference to time passing comes from the `ms_since_last_tick` argument.)

4. `TCPSenderMessage make_empty_message() const;`

   The `TCPSender` should generate and send a zero-length message with the sequence number set correctly. This is useful if the peer wants to send a `TCPReceiverMessage` (e.g. because it needs to acknowledge something from the peer's sender) and needs to generate a `TCPSenderMessage` to go with it.

   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 Checkpoint 3, please review the full interface in `src/tcp_sender.hh` 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.

2.3 FAQs and special cases

- *What should my* `TCPSender` *assume as the receiver's window size before the receive method informs it otherwise?*

  One.

- *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 SYN, payload, or FIN) 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 Ed regularly.

3 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 with `cmake --build build --target check3`.
3. Please re-read the section on "using Git" in the Checkpoint 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 it hard to follow your code.

4 Hands-on activity

Congratulations—you have made a fully working implementation of the Transmission Control Protocol, implementations of which are arguably the most prevalent computer program on the planet. It's time to take a victory lap! You'll communicate with Linux's TCP and with a lab partner, and then you'll modify your webget (from checkpoint 0) to use *your* TCP implementation. In your writeup, describe what you did, answer the questions below, and try to find something interesting to discuss!

4.1 Experiments within your own VM

We've given you a client program (`./build/apps/tcp_ipv4`) that uses your TCPSender and TCPReceiver to speak TCP-over-IP over the Internet.[3] We've also given you a similar program (`./build/apps/tcp_native`) that uses a Linux `TCPSocket`.

The big question: Can your TCP implementation (`tcp_ipv4`) interoperate with Linux's TCP (`tcp_native`)?

[3] If you're curious how this program works, the `tcp_peer.hh` and `tcp_over_ip.cc` files are probably the interesting part of how we glued your TCPSender/TCPReceiver into a conforming TCP peer.

4.1.1 Have Linux's TCP talk to itself

- First, let's do the boring part of making sure Linux's TCP implementation can talk to itself. Run Linux's TCP as a "server" (the peer that waits for an incoming SYN segment), listening on port 9090. On your VM, run: `./build/apps/tcp_native -l 0 9090`
- Next, try using Linux's TCP as the "client": the peer that initiates the connection by sending the first SYN segment to the server. In another terminal window on your VM, run: `./build/apps/tcp_native 169.254.144.1 9090`
- If all goes well, the "server" will print something like `DEBUG: New connection from 169.254.144.1:36568` and the "client" will print something like `DEBUG: Connecting to 169.254.144.1:9090... DEBUG: Successfully connected to 169.254.144.1:9090.`
- Try typing into each window, and you will see the same bytes on the other window.
- To end a stream, type ctrl-D (on a line by itself) to close the ByteStream Writer in *that* direction. If all goes well, you'll see `Outbound stream...finished` on the terminal where you typed the ctrl-D, and `Inbound stream...finished` on the other terminal. Notice that the other peer can keep sending to the "closed" peer—each direction of the stream can be closed independently, without preventing the other direction from continuing.
- Now end the stream in the second direction by typing ctrl-D (on a line by itself) in the other terminal. If all went well, both programs will quit and bring you back to the command line in both terminals. This indicates the TCP connection has finished in both directions (as discussed in class, Linux will "linger" in the background before reusing one of the port numbers to reduce the chance of a "two general's problem").

4.1.2 Have *your* TCP talk to Linux's

Repeat the above steps, but connect *your* TCP implementation to Linux's. **First,** run `sudo ./scripts/tun.sh start 144` to give your implementation permission to send raw Internet datagrams without needing to be root. You'll have to rerun this command any time you reboot your VM.

Then, rerun the above experiment, replacing *one* of the programs (the client or server) with `tcp_ipv4` (which is *your* TCP implementation). Does the connection still get established as before, and can each peer still type at the other and have the text appear on the other peer's window? If so, pat yourself on the back (and we'll shake your hand)—you've earned it! If not... time to start debugging. You can capture the TCP segments with a command like `sudo rm -f /tmp/capture.raw; sudo tcpdump -n -w /tmp/capture.raw -i tun144 --print --packet-buffered`; the resulting `/tmp/capture.raw` file can be visualized in wireshark as before.

After you've typed a little in each direction, try closing one of the ByteStreams and keep typing a little in the other direction. Do both programs quit cleanly after both streams have

finished with a ctrl-D? They should—although you may need to see `tcp_ipv4` wait a little to reduce the chance of a "two general's problem." When does it need to wait (when it's the first to close or the second to close)? Does this match what was discussed in class?

4.1.3 Try to pass the "one megabyte challenge"

Once it looks like you can have a basic conversation, try sending a file between `tcp_ipv4` (your TCP) and `tcp_native` (Linux's TCP).

To **create** a random file that's 12345 bytes as "/tmp/big.txt":

```
dd if=/dev/urandom bs=12345 count=1 of=/tmp/big.txt
```

You can choose the direction of transmission—i.e. whether the client or server is the one to send the file.

To have the **server** send the file as soon as it accepts an incoming connection, redirect standard input to read from the file:

```
./build/apps/tcp_native -l 0 9090 < /tmp/big.txt
```

To have the client receive the file, close off its outgoing stream by redirecting from `/dev/null`, and redirect standard output to a second file named "/tmp/big-received.txt":

```
</dev/null ./build/apps/tcp_ipv4 169.254.144.1 9090 > /tmp/big-received.txt
```

Or to have the **server** receive the file:

```
</dev/null ./build/apps/tcp_native -l 0 9090 > /tmp/big-received.txt
```

Or to have the **client** send the file:

```
./build/apps/tcp_ipv4 169.254.144.1 9090 < /tmp/big.txt
```

To compare two files and make sure they're the same:

```
sha256sum /tmp/big.txt
```

or

```
sha256sum /tmp/big-received.txt
```

If the SHA-256 hashes match, you can be almost certain the file was transmitted correctly.

Try this with a tiny file (12 bytes), then 65534 bytes (a little less than 2^{16}), then 65537 bytes (a little omre than 2^{16}), then 200000 bytes, then the full megabyte (1000000 bytes). If they all match, give yourself an even bigger pat on the back! If not... time to debug (possibly with tcpdump and wireshark as described above).

4.2 Reach out and talk to a friend

If everything works above, try communicating with a labmate over the Internet! One of you will run `tcp_native` as a server, as above. The other will run `tcp_ipv4` as the client, connecting to the labmate's address on the CS144 private network (10.144....).

Can you type to each other and successfully end the two streams cleanly? And if so, can you pass the one-megabyte challenge (sending a random 1000000-byte file successfully over the Internet to your labmate's VM, with the SHA-256 hashes matching perfectly on both sides)? If so, congratulations... now trade places and try sending the file in the other direction!

What's the biggest file that you have the patience to successfully send to your labmate? In your lab report, include the sizes of the two files (the output of `ls -l /tmp/big.txt` for the sender and `ls -l /tmp/big-received.txt` for the receiver) and the results of `sha256sum /tmp/big.txt` (on the sender's VM) and `sha256sum /tmp/big-received.txt` (on the receiver's).

4.3 webget revisited

Remember your `webget.cc` that you wrote in Checkpoint 0? It used a TCP implementation (`TCPSocket`) provided by the Linux kernel. We'd like you to switch it to use your own TCP implementation without changing anything else. We think that all you'll need to do is:

- Replace `#include "socket.hh"` with `#include "tcp_minnow_socket.hh"`.
- Replace the `TCPSocket` type with `CS144TCPSocket`.
- At the end of your `get_URL()` function, add a call to `socket.wait_until_closed()`.

> ⋆*Why am I doing this?* Normally the Linux kernel takes care of waiting for TCP connections to reach "clean shutdown" (and give up their port reservations) even after user processes have exited. But because your TCP implementation is all in user space, there's nothing else to keep track of the connection state except your program. Adding this call makes the socket wait until the connection is fully closed.

Recompile, and run `make check_webget` to confirm that you've gone full-circle: you've written a basic Web fetcher on top of *your own complete TCP "stack"*, and it still successfully talks to a real webserver. If you have trouble, try running the program manually: `./build/apps/webget cs144.keithw.org /hasher/xyzzy`. You'll get some debugging output on the terminal that may be helpful.

5 Submit

1. In your submission, please only make changes to the `.hh` and `.cc` files in the `src` directory (and `apps/webget.cc`). 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 check3` (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/check3.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) **Alternative design choices** that you considered or ideally evaluated in terms of their performance, difficulty to write (e.g., hours required to produce a bug-free implementation), difficulty to read (e.g., lines of code and their degree of subtlety or nonobvious correctness), and any other dimensions you think are interesting for the reader (or for your own past self before you did this assignment). Include any measurements if applicable.

   (c) **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?

   (d) **Remaining Bugs.** Point out and explain as best you can any bugs (or unhandled edge cases) that remain in the code.

   (e) **Hands-on Activity.** Include answers to the questions and some thoughtful commentary on the hands-on activity above.

4. 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 a lab session, or by posting a question on Ed. Good luck!

6 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 post your test on EdStem (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 10, that's great too.)