Attribution: This series of programming assignments, including codebase and documentation, is adopted from 'Sponge' of Stanford CS144 Introduction to Computer Networking by Prof. Keith Winstein https://cs144.github.io/.

Programming Assignment 4: the summit (TCP in full)

Due: April 26, 2024. 23:59
Late deadline: April 27, 2024. 23:59 (20% penalty)

1. Overview

You have reached the summit.

In Assignment 0, you implemented the abstraction of a *flow-controlled byte stream* (`ByteStream`). In Assignment 1, 2, and 3, you implemented the tools that translate—*in both directions*—between that abstraction and the one the Internet provides: unreliable datagrams.

Now, in Assignment 4, you will make the overarching module, called `TCPConnection`, that combines your `TCPSender` and `TCPReceiver` and handles the global housekeeping for the connection. The connection's TCP segments can be encapsulated into the payloads of user (TCP-in-UDP) or Internet (TCP/IP) datagrams—letting your code talk to billions of other computers on the Internet that speak the same TCP/IP language. Figure 1 again shows the overall design.
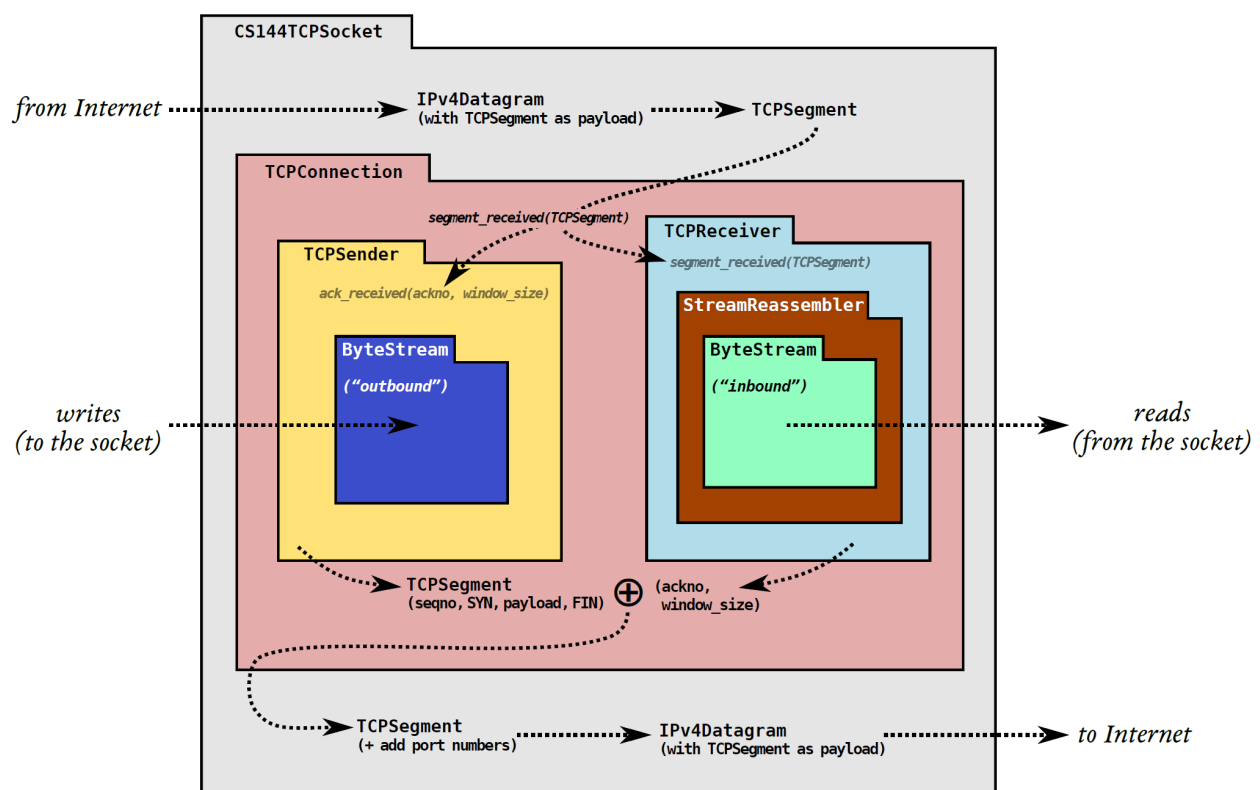

Figure 1 The arrangement of modules and dataflow in your TCP implementation.

A short note of caution: the `TCPConnection` *mostly* just combines the sender and receiver modules that you have implemented in the earlier assignments—the `TCPConnection` itself can be implemented in less than 100 lines of code. If your sender and receiver are robust, this will be a short assignment. If not, you may need to spend time debugging, aided by the test failure messages. (We'd discourage you from trying to read the test source code, unless it is as a last resort.) We **strongly encourage you to start early** and not leave this assignment until the night before the deadline.

2. Getting started

Your implementation of a `TCPConnection` will use the same Sponge library that you used in Assignment 0–3, with additional classes and tests. We're giving you support code that reads and writes TCP segments into the payloads of user and Internet datagrams. We're also giving you a class (`CS144TCPSocket`) that wraps your `TCPConnection` and makes it behave like a normal stream socket, just like the `TCPSocket` you used to implement `webget` back in Assignment 0. By the end of this assignment, you will slightly modify your `webget` to use *your* TCP implementation—a `CS144TCPSocket` instead of a `TCPSocket`. To get started:

1. Make sure you have committed all your solutions to Assignment 3. Please don't modify any files outside the top level of the `libsponge` directory, or `webget.cc`. You may have trouble merging the Assignment 4 starter code otherwise.

2. While inside the repository for the assignments, run `git fetch` to retrieve the most recent version of assignments.

3. Download the starter code for Assignment 3 by running `git merge origin/lab4-startercode`.

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

5. Outside the `build` directory, open and start editing the `writeups/assn4.md` file. This is the template for your assignment writeup and will be included in your submission.

3. Assignment 4: The TCP connection

In this assignment, you'll finish building a working TCP implementation. You've already done most of the work to get there: you've implemented the sender and the receiver. Your job here is to "wire them up" together into one object (a `TCPConnection`) and handle some housekeeping tasks that are global to the connection.

Recall: TCP reliably conveys a *pair* of flow-controlled byte streams, one in each direction. Two parties participate in the TCP connection, and *each party* acts as both "sender" (of its own outbound byte-stream) and "receiver" (of an inbound byte-stream) at the same time:
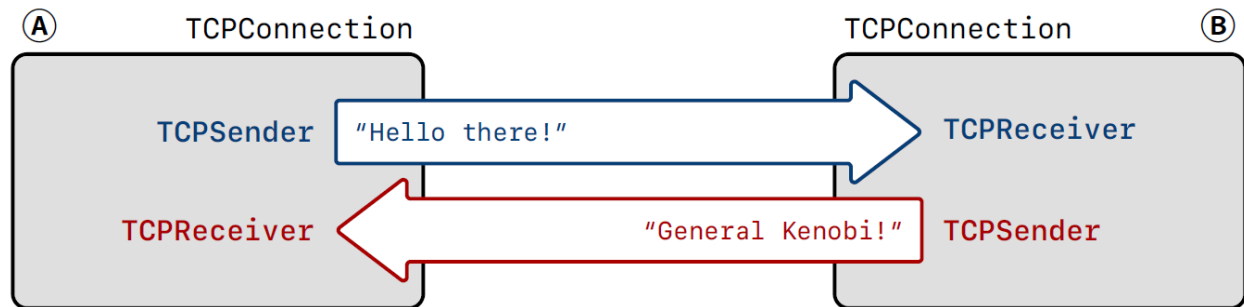
The two parties ("A" and "B" in the above diagram) are called the "endpoints" of the connection, or the "peers." Your TCPConnection acts as *one* of the peers. It's responsible for receiving and sending segments, making sure the sender and receiver are informed about and have a chance to contribute to *the fields they care about* for incoming and outgoing segments.

Here are the basic rules the TCPConnection has to follow:

Receiving segments. As shown on Figure 1, the TCPConnection receives TCPSegments from the Internet when its segment_received method is called. When this happens, the TCPConnection looks at the segment and:

- if the RST (reset) flag is set, sets both the inbound and outbound streams to the error state and kills the connection permanently. Otherwise it...

- gives the segment to the TCPReceiver so it can inspect the fields it cares about on incoming segments: seqno, SYN, payload, and FIN.

- if the ACK flag is set, tells the TCPSender about the fields it cares about on incoming segments: ackno and window_size.

- if the incoming segment occupied any sequence numbers, the TCPConnection makes sure that *at least one* segment is sent in reply, to reflect an update in the ackno and window size.

- There is one extra special case that you will have to handle in the TCPConnection's segment_received() method: responding to a "keep-alive" segment. The peer *may* choose to send a segment with an invalid sequence number to see if your TCP implementation is still alive (and if so, what your current window is). Your TCPConnection should reply to these "keep-alives" **even though they do not occupy any sequence numbers**. Code to implement this can look like this:

```
if (_receiver.ackno().has_value() and (seg.length_in_sequence_space() == 0)
    and seg.header().seqno == _receiver.ackno().value() - 1) {
    _sender.send_empty_segment();
}
```

Sending segments. The `TCPConnection` will send `TCPSegment`s over the Internet:

- **any time** the `TCPSender` has pushed a segment onto its outgoing queue, having set the fields it's responsible for on outgoing segments: (`seqno`, SYN, `payload`, and FIN).

- Before sending the segment, the `TCPConnection` will ask the `TCPReceiver` for the fields it's responsible for on outgoing segments: `ackno` and `window_size`. If there is an `ackno`, it will set the ACK flag and the fields in the `TCPSegment`.

When time passes. The `TCPConnection` has a `tick` method that will be called periodically by the operating system. When this happens, the `TCPConnection` needs to:

- tell the `TCPSender` about the passage of time.

- abort the connection, and send a reset segment to the peer (an empty segment with the RST flag set), if the number of consecutive retransmissions is more than an upper limit `TCPConfig::MAX_RETX_ATTEMPTS`.

- end the connection cleanly if necessary (please see Section 5).

As a result, the overall structure of each `TCPSegment` looks like this, with "sender-written" and "receiver-written" fields shown in different colors:

TCPSegment

Source Port Number (`sport`)	Destination Port Number (`dport`)
Sequence Number (`seqno`)	
Acknowledgement Number (`ackno`)	
Data Offset (`doff`) · URG ACK PSH RST SYN FIN	Window Size (`win`)
Checksum (`cksum`)	Urgent Pointer (`uptr`)
Options / Padding	
Payload	

The full interface for the `TCPConnection` is in the class documentation. Please take some time to read through this. **Much** of your implementation will involve "wiring up" the public API of the `TCPConnection` to the appropriate routines in the `TCPSender` and `TCPReceiver`. As much as possible, you want to defer any heavy lifting to the sender and receiver that you've already implemented. That said, not everything will be that simple, and there are some subtleties that involve the "global" behavior of the overall connection. The hardest part will be deciding when to fully terminate a `TCPConnection` and declare it no longer "active."

What follows are some FAQs and details of edge cases that you'll need to handle.

4. FAQs and special cases

- *How much code are you expecting?*

  Overall, we expect the implementation (in `tcp_connection.cc`) will require about 100–150 lines of code in total. When you're done, the test suite will extensively test your interoperability with your own implementation as well as the Linux kernel's implementation of TCP.

- *How should I get started?*

  Probably the best way to start is by wiring up some of the "ordinary" methods to the appropriate calls in `TCPSender` and `TCPReceiver`. This may include stuff like `remaining_outbound_capacity()`, `bytes_in_flight()`, and `unassembled_bytes()`.

  Then you may choose to implement the "writer" methods: `connect()`, `write()`, and `end_input_stream()`. Some of these methods may need to do something to the outbound `ByteStream` (owned by the `TCPSender`) **and** tell the `TCPSender` about it.

  You might choose to start running the test suite (`make check_lab4`) before you have fully implemented every method; the test failure messages can give you a clue or a guide about what to tackle next.

- *How does the application read from the inbound stream?*

  `TCPConnection::inbound_stream()` is implemented in the header file already. You don't have to do anything more to support the application reading.

- *Does the `TCPConnection` need any fancy data structures or algorithms?*

  No, it really doesn't. The heavy lifting is all done by the `TCPSender` and `TCPReceiver` that you've already implemented. The work here is really just about wiring everything up, and dealing with some lingering connection-wide subtleties that can't easily be factored in to the sender and receiver.

- *How does the `TCPConnection` actually send a segment?*

  Similar to the `TCPSender`—push it on to the `_segments_out` queue. As far as your `TCPConnection` 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.

- *How does the `TCPConnection` learn about the passage of time?*

  Similar to the `TCPSender`—the `tick()` method will be called periodically. Please don't use any other way of telling the time—the tick method is your only access to the passage of time. That keeps things deterministic and testable.

- *What does the `TCPConnection` do if an incoming segment has the* RST *flag set?*

  This flag ("reset") means instant death to the connection. If you receive a segment with RST, you should set the error flag on the inbound and outbound `ByteStream`s, and any

subsequent call to `TCPConnection::active()` should return false.

- *When should I **send** a segment with the* ⎡RST⎤ *flag set?*

There are two situations where you'll want to abort the entire connection:

1. If the sender has sent too many consecutive retransmissions without success (more than `TCPConfig::MAX_RETX_ATTEMPTS`, i.e., 8).

2. If the `TCPConnection` destructor is called while the connection is still active (`active()` returns true).

Sending a segment with ⎡RST⎤ set has a similar effect to receiving one: the connection is dead and no longer `active()`, and both `ByteStream`s should be set to the error state.

- *Wait, but how do I even make a segment that I can set the* ⎡RST⎤ *flag on? What's the sequence number?*

Any outgoing segment needs to have the proper sequence number. You can force the `TCPSender` to generate an empty segment with the proper sequence number by calling its `send_empty_segment()` method. Or you can make it fill the window (generating segments *if* it has outstanding information to send, e.g. bytes from the stream or SYN/FIN) by calling its `fill_window()` method.

- *What's the purpose of the* ⎡ACK⎤ *flag? Isn't there always an* `ackno`*?*

Almost every `TCPSegment` has an `ackno`, and has the ⎡ACK⎤ flag set. The exceptions are just at the very beginning of the connection, before the receiver has anything to acknowledge.

On outgoing segments, you'll want to set the `ackno` and the ⎡ACK⎤ flag whenever possible. That is, whenever the `TCPReceiver`'s `ackno()` method returns a `std::optional<WrappingInt32>` that has a value, which you can test with `has_value()`.

On incoming segments, you'll want to look at the `ackno` only if the ⎡ACK⎤ field is set. If so, give that `ackno` (and window size) to the `TCPSender`.

- *How do I decipher these "state" names (like "stream started" or "stream ongoing")?*

Please see the diagrams in the Assignment 2 and Assignment 3 handouts.

We want to emphasize again that the "states" are useful for testing and debugging, but **we're not asking you to materialize these states in your code.** You don't need to make more state variables to keep track of this. The "state" is just a function of the public interface your modules are already exposing.

- *What window size should I send if the* `TCPReceiver` *wants to advertise a window size that's bigger than will fit in the* `TCPSegment::header().win` *field?*

Send the biggest value you can. You might find the `std::numeric_limits` class helpful.

- *When is the TCP connection finally "done"? When can* `active()` *return false?*

Please see the next section.

5. The end of a TCP connection: consensus takes work

One important function of the `TCPConnection` is to decide when the TCP connection is fully "done." When this happens, the implementation releases its exclusive claim to a local port number, stops sending acknowledgments in reply to incoming segments, considers the connection to be history, and has its `active()` method return false.

There are two ways a connection can end. In an **unclean shutdown**, the `TCPConnection` either sends or receives a segment with the $\boxed{\text{RST}}$ flag set. In this case, the outbound and inbound `ByteStream`s should both be in the `error` state, and `active()` can return false immediately.

A **clean shutdown** is how we get to "done" (`active()` = false) without an error. This is more complicated, but it's a beautiful thing because it ensures as much as possible that *each* of the two `ByteStream`s has been reliably delivered *completely* to the receiving peer. In the next section (§§5.1), we give the practical upshot for when a clean shutdown happens, so feel free to skip ahead if you like.

Cool, you're still here. Because of the Two Generals Problem, it's *impossible to guarantee* that both peers can achieve a clean shutdown, but TCP gets pretty close. Here's how. From the perspective of one peer (one `TCPConnection`, which we'll call the "local" peer), there are four prerequisites to having a clean shutdown in its connection with the "remote" peer:

Prereq #1 The **inbound** stream has been fully assembled and has ended.

Prereq #2 The **outbound** stream has been ended by the local application *and* fully sent (including the fact that it ended, i.e. a segment with $\boxed{\text{FIN}}$) to the remote peer.

Prereq #3 The **outbound** stream has been fully acknowledged by the remote peer.

Prereq #4 The **local** `TCPConnection` is confident that the **remote** peer can satisfy prerequisite #3. This is the brain-bending part. There are two alternative ways this can happen:

- **Option A: lingering after both streams end.** Prerequisites #1 through #3 are true, and the remote peer *seems* to have gotten the local peer's acknowledgments of the entire stream. The local peer doesn't know this for sure—TCP doesn't deliver acks reliably (it doesn't ack acks). But the local peer is pretty confident that the remote peer has gotten its acks, because the remote peer doesn't seem to be retransmitting anything, and the local peer has waited a while to make sure.

   In specific, a connection is done when prereqs #1 through #3 are satisfied and **it has been at least 10 times the initial retransmission timeout (`_cfg.rt_timeout`) since the local peer has received *any* segments from the remote peer.** This is called "lingering" after both streams finish, to make sure the remote peer isn't trying to retransmit anything that we need to acknowledge. It does mean that a `TCPConnection` needs to stay alive for a while,[1] keeping an exclusive claim on a local port number and possibly sending acks in

---

[1] In a production TCP implementation, the linger timer (also known as the TIME-WAIT timer or twice the Maximum Segment Lifetime (MSL)) is typically something like 60 or 120 seconds. That can be long time to keep a port number reserved after a connection is effectively done, especially if you want to start a new server that binds to the same port number—nobody wants to wait two minutes. The SO_REUSEADDR socket option essentially makes Linux ignore the reservation and can be handy for debugging or testing.

response to incoming segments, even after the `TCPSender` and `TCPReceiver` are completely done with their jobs and both streams have ended.

- **Option B: passive close.** Prerequisites #1 through #3 are true, and the local peer is 100% certain that the remote peer can satisfy prerequisite #3. How can this be, if TCP doesn't acknowledge acknowledgments? Because the remote peer was **the first one to end its stream**.

> *Why does this rule work?* This is the brain-bender and you don't need to read further to complete this assignment, but it's fun to think about and gets to the deep reasons for the Two Generals Problem and the inherent constraints on reliability across an unreliable network. The reason this works is that *after* receiving and assembling the remote peer's FIN (prerequisite #1), the local peer sent a segment with a greater sequence number than it had ever sent before (at the very least, it had to send its own FIN segment to satisfy prerequisite #2), and that segment also had an `ackno` that acknowledged the remote peer's FIN bit. The remote peer acknowledged that segment (to satisfy prerequisite #3), which means that the remote peer *must have also seen the local peer's ack of the remote peer's* FIN. Which guarantees that the **remote peer** must be able to satisfy its own prerequisite #3. All this means the local peer can satisfy prerequisite #4 without having to linger.

The bottom line is that **if the `TCPConnection`'s inbound stream ends before the `TCPConnection` has ever sent a FIN segment, then the `TCPConnection` doesn't need to linger after both streams finish.**

5.1 The end of a TCP connection (practical summary)

Practically what all this means is that your `TCPConnection` has a member variable called `_linger_after_streams_finish`, exposed to the testing apparatus through the `state()` method. The variable starts out `true`. If the inbound stream ends before the `TCPConnection` has reached EOF on its outbound stream, this variable needs to be set to `false`.

At any point where prerequisites #1 through #3 are satisfied, the connection is "done" (and `active()` should return `false`) if `_linger_after_streams_finish` is false. Otherwise you need to linger: the connection is only done after enough time (10 × `_cfg.rt_timeout`) has elapsed since the last segment was received.

6. Testing

In addition to the automated tests, we encourage you to just "play around with" your TCP implementation, and **debug it with wireshark**. Here are some instructions:

Here's an example of how to run manually. You'll need two windows open, both in the `sponge/build` directory.

In one window, run: `./apps/tcp_ipv4 -l 169.254.144.9 9090`

This runs your `TCPConnection` as a server (with the segments going inside Internet (IPv4) datagrams), listening on a local address (169.254.144.9), port 9090. You should see:

`DEBUG: Listening for incoming connection...`

Now you may want to start capturing the segments sent and received so you can later inspect them in wireshark. In a different window, run `sudo tshark -Pw /tmp/debug.raw -i tun144`.[2] This will capture all TCP segments sent or received by the server, and save them to a file called "/tmp/debug.raw". You can later open this file with Wireshark (on your computer, not necessarily inside the VM) to see the full details of each segment.

In a third window, run a client: `./apps/tcp_ipv4 -d tun145 -a 169.254.145.9 169.254.144.9 9090`

This runs your `TCPConnection` as a client, connecting to the address that the server is listening on.

In the server window, you should now see something like:

`New connection from 169.254.144.1:52518.`

And in the client window, you should now see:

`Successfully connected to 169.254.144.9:9090.`

Now try typing something in either window (client or server) and hit ENTER. You should see the same text appear in the other window.

Now try ending one of the streams. In the client window, type Ctrl-D. This ends the client's outbound stream. Now the client window should look something like this:

`DEBUG: Outbound stream to 169.254.144.9:9090 finished (1 byte still in flight).`
`DEBUG: Outbound stream to 169.254.144.9:9090 has been fully acknowledged.`

and the server window should include something like this:

`DEBUG: Inbound stream from 169.254.144.1:52518 finished cleanly.`

Finally, in the server window, end the stream in that direction by typing Ctrl-D again.

The server window should now print something like the following and immediately return you to the command line (no lingering):

---

[2] You may need to install this program, with `sudo apt install tshark`.

```
DEBUG: Waiting for clean shutdown...
DEBUG: Outbound stream to 169.254.144.1:52518 finished (1 byte still in flight).
DEBUG: Outbound stream to 169.254.144.1:52518 has been fully acknowledged.
DEBUG: TCP connection finished cleanly.
done.
```

and the client window should print something like this:

```
DEBUG: Inbound stream from 169.254.144.9:9090 finished cleanly.
DEBUG: Waiting for lingering segments (e.g. retransmissions of FIN) from peer...
DEBUG: Waiting for clean shutdown...
```

after 10 seconds of lingering, the client window should print the rest and return you to the command line:

```
DEBUG: TCP connection finished cleanly.
done.
```

If one of these steps is going awry, that suggests a place to look in your termination logic (the decision about when to stop reporting active()=true).

Testing with a tiny window: if you're worried about whether your TCPSender is getting stuck when the receiver advertises a zero window case, try running the above commands and use a "-w 1" argument to the tcp_ipv4 program. This will make it use a TCPReceiver capacity of 1, which means that when you type "hello" into one side, only one byte can be sent and then the receiver will advertise a zero window.

Make sure your implementation doesn't get stuck there! It should successfully be able to send an arbitrary length string (e.g. "hello how are you doing") even when the receiver capacity is 1 byte.

7. Performance

After you've finished your TCP implementation, and after you are passing all of the tests run by make check_lab4, please commit! Then, measure the performance of your system and bring it up to at least 100 megabits per second.

From the build directory, run ./apps/tcp_benchmark. If all goes well, you'll see output that looks like this:

```
user@computer:~/sponge/build$ ./apps/tcp_benchmark
CPU-limited throughput               : 1.78 Gbit/s
CPU-limited throughput with reordering: 1.21 Gbit/s
```

To receive full credit on the assignment, your performance needs to be at least "0.10 Gbit/s" (100 megabits per second) on both lines. You may need to profile your code or reason about where it is slow, and you may have to improve your implementation of some of the critical modules (e.g., ByteStream or StreamReassembler) to get to this point.

In your writeup, please report the speed figures you achieved (with and without reordering).

If you would like, you're welcome to try to optimize your code as much as you want. However, we won't give extra points for performance that's faster than 100 Mbit/s—any improvements you do beyond this minimum are for your own satisfaction and learning only.

8. webget revisited

Time to take a victory lap! Remember your `webget.cc` that you wrote in Assignment 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"` to `#include "tcp_sponge_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 your `TCPConnection` reports `active()` = false.

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 implementation*, and it still successfully talks to a real webserver. If you have trouble, try running the program manually: `./apps/webget tomahawk.postech.ac.kr /hasher/xyzzy`. You'll get some debugging output on the terminal that may be helpful.

9. Development and debugging advice

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

2. We are expecting about 100–150 lines of code in total. You won't need any fancy data structures or algorithms.

3. You can test your code (after compiling it) with `make check_lab4`. This will run a fairly comprehensive test suite (159 tests). Many of the tests confirm that your TCP implementation can transfer files error-free with Linux's TCP implementation, or with

itself, over various combinations of packet loss and data transfer in each direction.[3]

4. Please re-read the section on "using Git" in the Assignment 0 document and in the online FAQs, 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.

5. Please work to make your code readable to the TA 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.

6. Please also keep to the "Modern C++" style described in the Assignment 0 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 assignments. (You may sometimes need to use the `move()` function to pass an object that can't be copied.)

7. 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.

10. 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!)

---

[3] In the test names, "c" means your code is the client (peer that sends the first `SYN`), and "s" means your code is the server. The letter "u" means it is testing TCP-over-UDP, and "i" is testing TCP-over-IP (TCP/IP). The letter "n" means it is trying to interoperate with Linux's TCP implementation. "S" means your code is sending data; "R" means your code is receiving data, and "D" means data is being sent in both directions. At the end of a test name, a lowercase "l" means there is packet loss on the receiving (incoming segment) direction, and uppercase "L" means there is packet loss on the sending (outgoing segment) direction.

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

(d) `make check_lab4` (to make sure the automated tests pass)

3. Write a report in `writeups/assn4.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 specify the usage of best-submission codes, if you have used **any**.

   Caution: When specifying your usage of best codes, you should include all of the best codes that exist in not only your current Assignment but **also in any part of your past Assignment that you are building upon**. The below figure shows examples of how to specify the usage of best codes.

   If your submission is not purely yours, your score will be capped at 90% of the full score, and your code is not a candidate for the best code selection. This penalty persists unless you replace all the best codes you used **with your implementation**.

5. In your writeup, please also fill in the number of hours the assignment took you and any other comments.

6. When you have finished your implementation, do not forget to **commit all changes you made!** This **should be done before creating an archive** of your git repository. If you create an archive before committing changes, the archive will include only part or none of

the changes you made. This may result in a lower score than you anticipated, or worse zero.

If you are unsure whether your archive includes all changes you made, please refer to our Q&A item: How do I make sure the created bundle includes all my commits?.

7. To archive your repository, use the following command to archive your git repository:

```
git bundle create /tmp/<your_student_id>.git --all
```

Important: this command will create an archive of your git history, which means that your solution must be committed to your git repository! In addition, we will be grading only the master branch of your repository, so please be sure that branch corresponds to the solution you want to turn in.

8. Please upload your archive (bundle) file to the proper assignment entry on PLMS.