DESIGN

Status | Done

Overall Program Design

💬 This file contains a high-level description of the program architecture and the functions in each file. The code files can be referenced for more detailed information.

Sections:

- Normal Program Run
- High Level Functions Overview
- How it Works
- Major Design Decisions

“Normal” Program Run 🏃

💬 A description of a successful file transfer between the client and the server. Other cases are discussed later in this document.

- Run udpl, tcpserver, and tcpclient, following the “How to run” section of the README

TCPClient Walkthrough

- Three way handshake:
    - The client will initiate the three-way handshake. It will send SYN segments until the server responds with a SYNACK.
    - The client will send an ACK back to complete the three-way handshake sequence. This ACK will have some data piggybacked on it— the filesize of the file being transferred.
    - At this point, the three-way handshake is finished in the client’s POV, but the “=====connection established======” message doesn’t print until the server ACKs this back, as the client’s ACK could have been lost.
- The client will then start sending data. Go-Back-N pipelining is used so not every data segment needs to be ACK’d before another is sent. To do this:
    - The client creates a window (which holds all the segments that are sent but not ACK’d) fill it up with tcp segments with (MSS- 20) bytes of the file data as the segments’ payload. After receiving an ACK for a segment in the window, it will move the window’s indices forward and send the next data from the file.
    - On 3 duplicate ACKs, the client will retransmit the first segment in the window
    - If a timeout occurs, all the segments in the window are resent.
- After the client reaches the end of the file, we can’t return yet because the data might not have been received. Instead, wait until the server sends a FIN to indicate that the file is finished being received.

- Upon receiving a FIN, the client will:
    - respond with an ACK (which may get lost. See code for comment on this)
    - send its own FIN and retransmit if there's a timeout
    - receive an ACK and send nothing, exiting.
- This entire time, the timeout values will be adjusted according to SampleRTTs and timeout adjustments as discussed in RFC 6298. This is also discussed later in this document.

TCPServer Walkthrough

- Three-way handshake:
    - The server will wait and listen for a SYN segment. To avoid listening forever and having a half open-connection, it will try MAX_RETRIES amount of times, setting the timeout interval accordingly.
    - If it receives one, it will respond with a SYNACK and wait until the client sends its final ACK to complete the connection. This ACK will contain the filesize.
- Data transmission:
    - Because the server needs to acknowledge that it has received data (filesize), it will send an ACK back, indicating to the client that it is ready to receive the file.
    - The server will start to receive the file from the client. It keeps track of the next in-order sequence number. When it receives a segment, it checks whether the segment has this sequence number. If so, it writes the data to the output file. Out of order segments are discarded. In both cases, an ACK is sent with the next sequence number that the server expects. When the segment is out of order, this ACK will be a duplicate.
    - The server will receive data until it receives the amount of bytes specified by the payload in the last leg of the handshake or if the client sends a FIN request.
- The server will send a FIN request, indicating that it is done reading the file. Then, the server will:
    - wait until it receives a FINACK from the client
    - after receiving the FINACK from the client, it will wait for the client's FIN segment and send its own FINACK.
    - wait TIME_WAIT seconds after sending the FINACK and exit.

Functions 👩‍💻

💬 See the docstrings in the actual code for a description of the functions, as well as the comments throughout the code. However, for convenience, I provided a very high-level description of the functions below:

utils.py

💬 contains useful functions and parameters used by both the tcpclient and tcpserver, including checksum calculations, segment and header creation, and header checking.

- `SimplexTCPHeader` — makes creating a header, setting the header fields, and attaching a checksum and payload easy.
    - `make_tcp_segment` — attaches a payload onto an existing SimplexTCPHeader object and returns a bytearray representing the segment
    - `_make_tcp_header_without_checksum` — internal method to make a bytearray representing the TCP header without the checksum. This method is necessary to compute the checksum later.

    - `unpack_segment` — breaks apart a tcp segment into its constituent parts; returns a tuple of (seq_num, ack_num, flags, recv_window, payload) from a segment
- `validate_args` — validates command line arguments for tcpsender and tcpreceiver
- `calculate_checksum` — used to fill the checksum field of a tcp header and to verify whether bits within a segment have been altered.
- `verify_checksum` — uses `calculate_checksum` to verify that the checksum in the header of the received segment matches the computed checksum of the received segment
- `are_flags_set` — given the 8-bit flag field, checks whether the expected flags bits are set via bitmasking

tcpclient.py

- `create_and_bind_socket` — creates the UDP socket to communicate with newudpl and binds it to the specified ack port
- `create_tcp_segment` — creates a TCP segment with a payload and flags (autofills the src port and destination port), uses the `make_tcp_segment` of utils to abstract from manipulating bytes
- `run` — runs the client functionality
- `update_timeout_on_rtt` — called after receiving a valid SampleRTT, updates the socket timeout values based on the formulas specified in RFC 7298. If it is the first time calling this function, the estimatedRTT and devRTT are initialized to sampleRTT and sampleRTT/2, respectively. The parameters to tweak the EWMA formulas' weights can be adjusted in utils.
- `update_timeout_on_timeout` — called when a timeout occurs, updates the timeout values for the client's socket by TIMEOUT_MULTIPLIER.
- `establish_connection` — takes care of the entire connection establishment process with the server (three way handshake). There are many cases where the client's messages get dropped during connection establishment, so there are helper functions to abstract away the ugliness. For detail about these cases, see the helper functions' comments, but otherwise the cases are covered at a high level in the "How it works: connection establishment" section.
    - `_send_syn_and_wait_for_synack`
    - `_send_ack_with_filesize`
- `send_fin` — called when the client needs to initiate TCP connection teardown. This is only called when the client has retransmitted a segment MAX_RETRIES times. When the client is the initiator, it needs to successfully send a FIN to the server, wait for the server to send its own FIN segment, send a FINACK to the server, wait TIME_WAIT seconds to make sure the FINACK was received, and then close the connection. There are many cases where the client's messages get dropped during teardown, so there are helper functions to abstract away the ugliness. For detail about these cases, see the helper functions' comments, but otherwise the cases are covered at a high level in the "How it works: connection teardown" section
    - `_send_fin_and_wait_for_ack`
    - `_wait_for_fin_and_send_ack`
- `respond_to_fin` — called when the client receives a FIN request from the server (after the server is finished receiving the file). The client will receive the FIN, respond with a FINACK (with retries), and exit. For detail about edge cases that can happen during the fin response, see the code comments.
- `send_file_gbn` — sends the file to the server via the GBN pipelining protocol.
    - The client will read MSS data from the file to break it into packets and send out (windowsize // MSS) packets. We keep track of the window's packets and how many retries the packet has taken.

    - One segment from the window will be chosen to measure a SampleRTT for so we can update the timeout values. If the segment has been retransmitted, we discard the SampleRTT and live with the fact that we won't have a SampleRTT for this RTT.
    - Then it will try receiving acks, verifying if the ack for a segment that is currently in the window.If it is, we can move the window forward.
    - If it does not receive a valid ack in time, it will timeout and retransmit all the segments in the window, incrementing the number of retransmissions that the segments have taken.
    - If a segment has been retransmitted too many times, it will begin the connection teardown sequence by sending a FIN to the server.
    - If it has received a triple duplicate ack, it will perform a fast retransmit and send the first segment in the window as it was probably lost.

tcpserver.py

- `create_and_bind_socket` — creates the
- `create_tcp_segment` — creates a TCP segment with a payload and flags (autofills the src port and destination port), uses the `make_tcp_segment` of utils to abstract from manipulating bytes
- `run` — runs the server functionality
- `update_timeout_on_rtt` — same as tcpclient's, but the RTT is different for the server as it sends segments directly to the client instead of through newudpl.
- `update_timeout_on_timeout` — see above.
- `send_fin` — called when the server needs to initiate TCP teardown. This is only called when it is done receiving the file from the client. The server will send a FIN to the client, try to receive an ack from the client, try to receive a FIN from the client, and send an ACK to the client. Because there are many cases where the client's messages could get dropped and an unsuccessful teardown could occur during this, there are helper methods, listed below. See their comments for more detail.
    - `_send_fin_and_wait_for_finack`
    - `_wait_for_fin_and_send_finack`
- `respond_to_fin` — called when the server receives a FIN from the client (when the client has tried to retransmit a segment too many times). The server will receive the FIN and respond with an ACK and its own FIN, both of which are guaranteed to get delivered. Then, it will receive an ACK and do nothing, entering the CLOSED state and terminate.
- `receive_file_gbn` — implements the GBN style of receiving for the client. The receiver will keep track of the expected sequence number to receive next, denoted by nextseqnum. Out of order and corrupted packets will be discarded. Upon receiving any packet, the receiver will send an ACK with the last in-order packet correctly received, resulting in duplicate ACKs for OOO packets.
- `establish_connection` — establishes a connection with the client address by first listening for a SYN segment and stashing the client's isn. Then it will send a SYNACK segment with its own randomized server isn. Then it will receive an ACK segment with the filesize as the payload piggybacked. Because things could go wrong at any moment (e.g. the client reaches the transmission limit for the SYN and has to send a FIN to terminate, the client's ACK doesn't come, etc), we again!! abstract this handling in the helper functions whose comments you can take a look at.
    - `_listen_for_syn`
    - `_send_and_wait_for_ack`

How it Works

💬 for grading convenience, this section is split according to the rubric sections. Some design decisions are discussed here, but I think the most interesting tradeoffs are discussed in the Design Decisions summary.

🌭 describe how it all fits together

- examples
    - process of reading data from the file
    - breaking it into packets
    - pipelining the packets
    - generating and handling acks
    - handling losses

Establishing a connection 🤝

- The conceptual details of this section are taken from pg 249 of K&R
- As TCP is connection-oriented, the two processes that are trying to communicate with each other must first set up a connection with each other via a procedure called the three-way handshake. This connection-establishment procedure is as follows:

1. The client process initiates by first sending a SYN with a random `client_isn` in the seq_num in the segment to the server.
2. After receiving the SYN (there is a possibility this doesn't happen), the server will respond with SYNACK segment to indicate that it has agreed to establish this connection. This SYNACK has `client_isn+1` in the header's ack field, and puts its own initial random sequence number `server_isn` in the seq num.
3. After the client has received the SYNACK segment, the client will send a final segment to complete the connection-establishment procedure. This final ACK segment is piggybacked on 4 bytes of data containing the file's size.

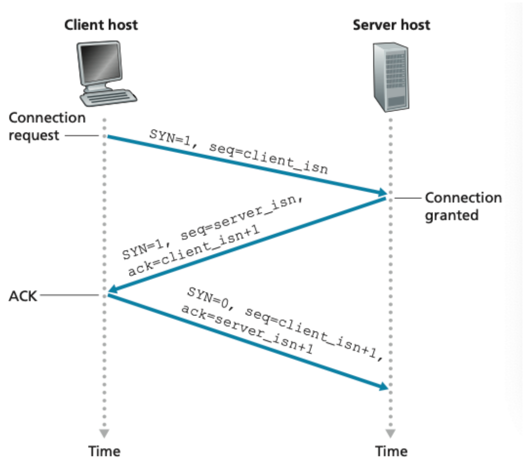


Figure 3.39 ♦ TCP three-way handshake: segment exchange

- This is the exchange if all goes well. But what happens if the SYN segment does not arrive?

packet loss during a three-way handshake

💬 packet loss can occur during connection establishment so the following design decisions were made to avoid a half-open connection for a prolonged period of time.

- Losing SYN packet: Following this Ed thread, the client will try sending the SYN segment MAX_RETRIES amount of times until it receives an SYNACK with the client_isn+1, resetting the timeout according to the tcp standard. It will also attempt to measure a SampleRTT if this SYN segment is not retransmitted.
    - If the client doesn't receive that SYNACK within MAX_RETRIES transmissions, it will just abort the connection.
    - The server will be listening for the SYN until it times out MAX_RETRIES times (multiplying the timeout every time), until it also aborts the connection.
- Losing the client's final ACK: See the comment in `establish_connection`. The client will send an ACK segment with the file size piggybacked to the server and wait for an ACK from the server before officially sending file data. This ACK is treated like all other segments (retransmitted until it hits the limit). If the ACK is not ACK'd back within the retransmission limit, the client will send a FIN and the server will respond.
    - This mechanism is to prevent a half-open connection if the last ACK of the three-way handshake gets lost. While the server will timeout waiting for its SYNACK to be ACK'd, the client will think that the connection is established and start sending data. By waiting for an additional ACK, we can ensure that the connection is fully established before sending file data.
    - The tradeoff is just a longer connection establishment time. This is implemented in `SimplexTCPClient._send_ack_with_filesize`

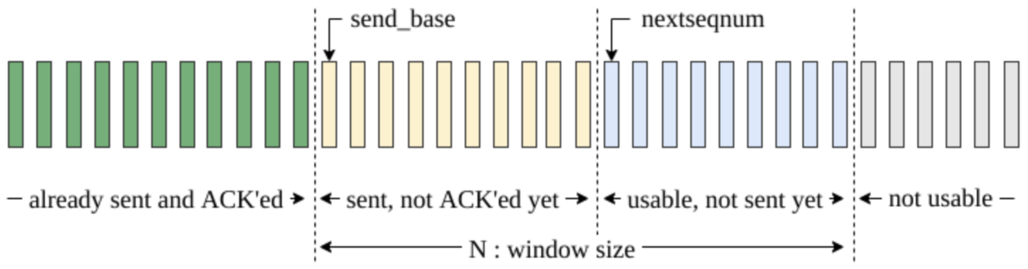
Data transmission and reception

Data transmission 🚀

- The file is divided in (file size in bytes)/(MSS in bytes) number of segments, with each segment being of size MSS
    - data transmission and reception is implemented in `SimplexTCPClient.send_file_gbn` and `SimplexTCPServer.receive_file_gbn`
- The client will read MSS data from the file to break it into packets and send out (windowsize // MSS) packets.
    - We keep track of the window's packets and how many retries the packet has taken with a list of tuples to take valid SampleRTT measurements.

```
# window format:
[(segment, num_retries), ...]
```

    - `send_base` and `next_seq_num` keep track of the window's indices:



- One segment from the window will be chosen to measure a SampleRTT for so we can update the timeout values. If the segment has been retransmitted, we discard the SampleRTT and just have to live with the fact that we won't have a SampleRTT for this RTT.
- Once the client has filled up its window, the client will try receiving acks, verifying if the ack for a segment that is currently in the window. If it is, we can move the window forward by removing the segment from the window and updating the `send_base`.
- To account for lost packets, if the client does not receive a valid ack in time, it will timeout and retransmit all the segments in the window, incrementing the number of retransmissions that the segments have taken.
- If a segment has been retransmitted too many times, it will begin the connection teardown sequence by sending a FIN to the server.
- Fast retransmit
  - If it has received a triple duplicate ack, it will perform a fast retransmit and send the first segment in the window as the packet was probably lost
  - Especially when window sizes get bigger and `newudpl` gets more packet loss, it can take a while for retransmissions to occur, so fast retransmit is useful.

Data reception 👐

- All the data that the client sends will be written to a file called `recvd_file` with no extensions so users can easily `diff` it with the original file. Data reception is implemented in `SimplexTCPServer.receive_file_gbn`.
- The receiver will keep track of two state variables:
  - the next expected sequence number to receive next, denoted by `next_seq_num`.
  - the number of `bytes_received` so far from the client. This is used so the server can detect when we're done transmitting. Recall that the client sends the size of the file to the server as part of the last third of the three-way handshake, so the server can just compare `bytes_received` with the filesize.
- The server will open a new file called `recvd_file` for writing and try receiving data from the client, updating the timeout accordingly.
- If the server receives something, the received segment can:
  - be out of order or corrupted —> the packet is discarded and a duplicate ack is sent (ack with the last in-order packet correctly received)
  - have the correct expected sequence number, in which case the ack for `next_seq_num` is sent and `next_seq_num` is incremented to denote that we received it (again, it's the ack with the last in-order packet correctly received)
- Terminating data reception
  - the server will either keep trying to receive data "infinitely" until it has received the `filesize` bytes. Then, it will send a FIN to the client.
  - or can terminate prematurely if the server has hit its retransmission limit (in which case it will begin the `_respond_to_fin()` sequence). Thus, there is a check in `receive_file_gbn` to check if the server received a FIN segment.
- Note that the seq_nums of the TCPServer and the ack_nums of the TCPClient don't have to be used because we're using a GBN policy!

Ending a connection (FIN request) 🏁

- Both the server and the client are able to initiate the teardown sequence. This is implemented in both file's `send_fin()`.

- The client will initiate the closing of a TCP connection when a segment hits its retransmission limit. This can happen while sending file data or during the last ACK of the three way handshake. This sequence depicted by the diagram below (from K&R textbook) will be kicked off.
    - The client will send a FIN (which can get lost). If the FIN segment hits the retransmission limit, there is nothing that the client can really do, so it just aborts and exits.
    - If the server successfully receives the FIN, it will respond with a FINACK and then a FIN. After this point, the server can try to receive the ack (which it does try, but only once), but it can safely ignore this ACK and wait TIME_WAIT seconds before closing the connection. This is safe because the server's FIN is guaranteed to arrive at the client so the client will begin the timed wait and close its side, resulting in both sides closed. This is implemented in `respond_to_fin()`

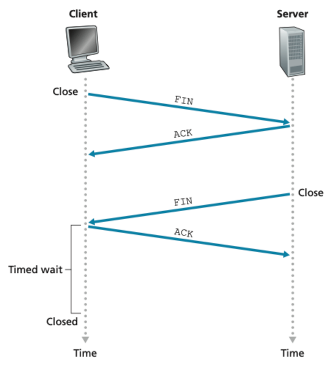


Figure 3.40 ♦ Closing a TCP connection

- The server will initiate the closing of a TCP connection when it has received the entire file from the client. This will kick off the same sequence as above, with the server and client's roles reversed.
    - The server will send a FIN to the client.
    - The client will respond with a FINACK, which can get lost. The client will also respond with a FIN, which can also get lost.
        - Because the client's FIN is what really kicks off the closing sequence for the client, we only try retransmitting the client's FIN segment MAX_RETRIES amount of times (and not the first FINACK). Technically, it should close the connection as soon as the client sends the FIN, but we make our best effort in telling the server that the client is also shutting down. This is implemented in `respond_to_fin()`
- To test this functionality:
    - test client initiation of connection termination by setting the MAX_RETRIES super low
    - test server initiation by making the file transmitted super small so the server will send the FIN.
- In my tests, there were a lot of times where the file was finished being received but some leftover in-flight packets were being received during the server shutdown sequence. In any case, it terminates gracefully.

Error logging ⚠️

- Python's logging module is used to record timeouts, duplicate acks, retransmissions, and connection teardown/setup states so there are timestamps recorded with everything. The logging levels can be adjusted per the README.txt, but it is pasted here for convenience:
- To get **more** verbose logging, you can set `LOGGING_LEVEL` in `utils.py` to logging.DEBUG. This may come at the expense of performance if the file is large and especially because the logs are being written to multiple streams.

- To get **less** verbose logging, you can set LOGGING_LEVEL to `logging.INFO` , `logging.WARNING`, `logging.ERROR`, etc. Higher levels will include all the logs from the levels below. For example, INFO will include all the logging.error logs. See the documentation for Python's logging module for more details.
    - `DEBUG` — lots of information: every packet and its ack_num, seq_num, payload, and flags, when a new sample RTT is measured / discarded when the packet being measured is a retransmission, window moving forward and which segment is being removed from it, the expected ack_num when a duplicate ack is received, etc.
    - `INFO` — timeout updates, states for connection teardown and establishment (e.g. CLOSE_WAIT, SYN_SENT, etc), file transfer messages, out of order segments, etc.
    - `WARNING` — maximum retransmissions reached —> beginning to terminate program, handling unexpected exceptions (haven't seen this one be invoked yet)
    - `ERROR` — checksum verification failures, unexpected flags in received segments, incorrect ack numbers during the handshake sequence, invalid arguments,
- For convenience, logs are written to `tcpclient.log` , `tcpserver.log` , as well as `stdout`

20-byte TCP header format 💆‍♀️

- The format of the TCP headers attached to the payload follows figure 3.29 from the textbook. There aren't many design decisions we can make here.

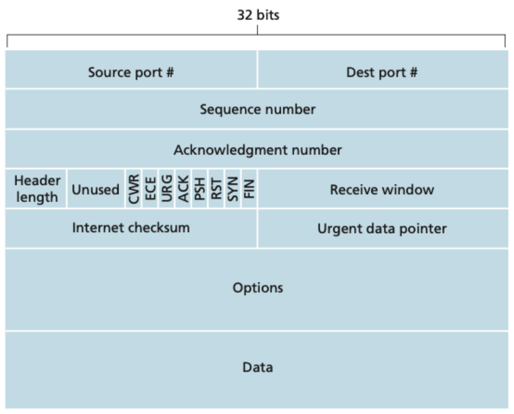
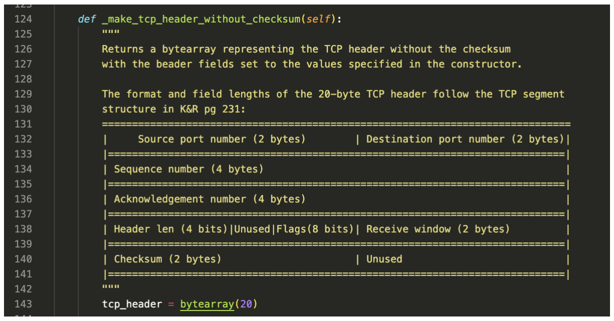


Figure 3.29 ♦ TCP segment structure

- The 20-byte TCP header a python bytearray, which we extend to add the payload.
- The header is created in `_make_tcp_header_without_checksum` in utils, as we cannot compute the checksum without the segment's payload (if it has a payload).
- There are some fields that are not used in this program (CWR, ECE, RST, urgent data pointer, options), so the header looks more like:



```
    def _make_tcp_header_without_checksum(self):
        """
        Returns a bytearray representing the TCP header without the checksum
        with the beader fields set to the values specified in the constructor.

        The format and field lengths of the 20-byte TCP header follow the TCP segment
        structure in K&R pg 231:
        ==============================================================================
        |     Source port number (2 bytes)        | Destination port number (2 bytes)|
        |============================================================================|
        | Sequence number (4 bytes)                                                   |
        |============================================================================|
        | Acknowledgement number (4 bytes)                                            |
        |============================================================================|
        | Header len (4 bits)|Unused|Flags(8 bits)| Receive window (2 bytes)         |
        |============================================================================|
        | Checksum (2 bytes)                      | Unused                            |
        |============================================================================|
        """
        tcp_header = bytearray(20)
```

Retransmission timer adjustment as per TCP standard ⏳⌛

- Figuring out how long the timeout should be before a segment gets retransmitted is tough; it must be larger than the RTT, otherwise there would be too many retransmissions. But it can't be too large, otherwise lost packets will not be quickly retransmitted.

The following bullets document some design choices concerning the retransmission timer.

- A maximum retransmission limit of 7 (can be tweaked in `utils`) was set because typical TCP implementations have a MAX_RETRIES of 5-7
- The socket timeouts are set via the below formula taken from the textbook. These implementations can be found in the tcpclient and tcpserver's `update_timeout_rtt`

$$TimeoutInterval = EstimatedRTT + 4DevRTT$$

  - where EstimatedRTT and DevRTT are measures of the "average sampleRTT" and " variability of the RTT" given by these formulas from RFC 6298:

$$EstimatedRTT = 0.875EstimatedRTT + 0.125SampleRTT$$

$$DevRTT = 0.75DevRTT + 0.25|SampleRTT - EstimatedRTT|$$

  - the `EstimatedRTT` formula is a weighted combination of the previous value of `EstimatedRTT` and `SampleRTT` from RFC 6298. The motivation behind using an "average" is that `SampleRTT`s vary a lot with network congestion and fluctuations:
- Having a timer per unACK'd packet would be too hefty for the program (there can be lots of in-flight packets). Like most other TCP implementations, the SimplexTCPClient only takes a `SampleRTT` measurement for only one of the transmitted but UNACK'd segments. This is because we only really need a new value of SampleRTT per RTT, it is unnecessary and slows down things if we take any more measurements.
- The client does not compute `SampleRTT` for retransmissions, as the client cannot distinguish between an ACK for the original segment vs an ACK for a retransmitted segment (see HW3). Measuring RTTs for retransmissions could cause a huge dip in the SampleRTT and lead to even faster transmissions, jamming the network more.
  - This is implemented by keeping track of how many times a packet has had to be retransmitted in the window.
  - when there's a lot of loss (e.g. L50), it takes a longer time to send as expected because the SampleRTTs aren't updating on a per-RTT basis since there's more retransmissions.
- Per RFC 6298, the initial timeoutInterval value is set to 1 second. Upon receipt of the first valid ACK, the EstimatedRTT is initialized to SampleRTT, and DevRTT=0.5*SampleRTT.

    - We try to measure valid ACKs in the connection setup sequence, but there is a chance that these segments also had to be retransmitted.
- I chose to not do any adjusting for the SampleRTT in the FIN sequence, as the connection is shutting down.
- Note that the server cannot measure as many SampleRTTs as the client, as it does not send data and wait for ACKs often. It only waits for a segment's ACK during connection setup and teardown, but I chose to only attempt to measure a SampleRTT during connection setup as having an accurate timeout value during teardown is not of the highest priority. If this SampleRTT was not representative of the actual average RTT of the server → receiver link, this decision could cause lots of timeouts.

Correct computation of the TCP checksum ✅

- The checksum is calculated in `calculate_checksum` and attached to the segment in `make_tcp_segment`
- The process is as follows:
    - make sure the segment's length is even. if not, pad a 0 byte.
    - set the checksum field to 0
    - sum all the 16-but words in the segment (header and data with the checksum field set to 0)
    - take the 1st complement of this sum and wrap the overflow around
- The verification process slightly deviates from the textbook. The textbook says that TCP verifies by checking if the received checksum + calculated checksum = 1111111111111111, but the program verifies by checking if the recomputed checksum of the segment received (with the checksum zero'd out) is equal to the checksum in the packet. Both are logically the same.

Design Decisions Summary 🎨

Using GBN instead of SR 📊

- When resending segments after a timeout, the client uses a Go-Back-N policy. It sends all segments in the window.
- This is as opposed to using a Selective Repeat, where the receiver individually acknowledges correctly received packets, so correctly received OOO packets will be buffered.
- Pros of using GBN
    - it simplifies receiver design a lot by removing receiver buffering altogether, so it the server only needs to know about `next_seq_num`. The code for this project was already super bulky so even though buffering OOO segments is the practical approach, I chose GBN for ease of implementation.
    - Also means that the seq_nums of the TCPServer and the ack_nums of the TCPClient don't have to be used! This made debugging a lot easier.
- Cons of using GBN
    - We throw away a correctly received packet, and a subsequent retransmission of that packet might be lost. This could cause more retransmissions.
    - For large window sizes and large packet loss, GBN is not great. A single packet error can cause GBN to retransmit all the packets in the window (unnecessarily for a lot of the packets), so the file transfer can terminate before the entire file is received, since the client hits the retransmission limit. Or, the file transfer just takes a really really long time to complete.

Retransmission timer adjustment ⏰

- The timeout multiplier (what is multiplied to the timeoutInterval after a timeout) is set to 1.1 instead of the recommended doubling of the timeout.
- Otherwise, the timeout increases too quickly and the file transfer will take a really long time (especially if there are a lot of retransmissions and the SampleRTTs aren't also adjusting the timeout intervals).

TIME_WAIT ⌚

- After receiving a FIN, the server/client will wait for 5 seconds before terminating, which is a lot shorter than what the textbook says (30 seconds). This made it easier for testing.