CSE351 Computer Networks

Fall, 2025, Programming Assignment #3
Due: November 9, 2025 (Sun), 23:59 KST

Instructor: Taesik Gong (taesik.gong@unist.ac.kr), TA: Daeun Lee (delee@unist.ac.kr)

==========

Submission Instructions. You shall submit this assignment as a single ZIP file named PA3_StudentNumber_Name.zip (e.g., PA3_20251234_HongGildong.zip) on the Blackboard. The ZIP must include transport.c and report.pdf

Collaboration Policy. You are welcome to discuss the homework with your classmates to understand the concepts, and you may also use code snippets from the provided guide links. However, if your report or code is found to be unreasonably identical to someone else's work, it will be considered plagiarism and will result in no points for all involved.

Project Overview.. In this project, you will implement your own socket layer supporting reliable data transfer. The project is composed of two large parts. First, you are going to implement 3-way and 4-way handshaking for connection setup and teardown. More specifically, you are going to create, destroy, send and receive packets. By doing this project, you will be able to understand the TCP finite state machine and the process of handshaking in TCP. Next, you will implement the data transfer protocol in the transport layer under the write() and read() system calls. You will transmit packets with payload, sequence number, and ack number. All implementations will be based on the assumption of a reliable environment, so you don't have to think about packet loss, reordering, retransmission, and timeout(not for delayed ACK). At the end of this project, your sample client should be able to download and write data from the server side.

What to Do. Your main objective in this assignment is to implement STCP (Simple TCP), a reliable, connection-oriented transport layer protocol. You will work within a provided skeleton framework that already includes a MYSOCK socket layer and dummy client/server applications. While the MYSOCK layer is complete, your specific task is to implement the core logic of STCP to replace the skeleton's bogus transport layer. Your STCP implementation must guarantee in-order, reliable data delivery and be full-duplex, allowing data to flow in both directions over a single connection. It must also treat application data as a transparent stream by managing all packetization and reassembly. Furthermore, implementing delayed ACK is a mandatory requirement. The ultimate goal is to enable the provided dummy client and server applications to function correctly and reliably using your completed STCP implementation.

==========

[Description 1] 3-way handshaking

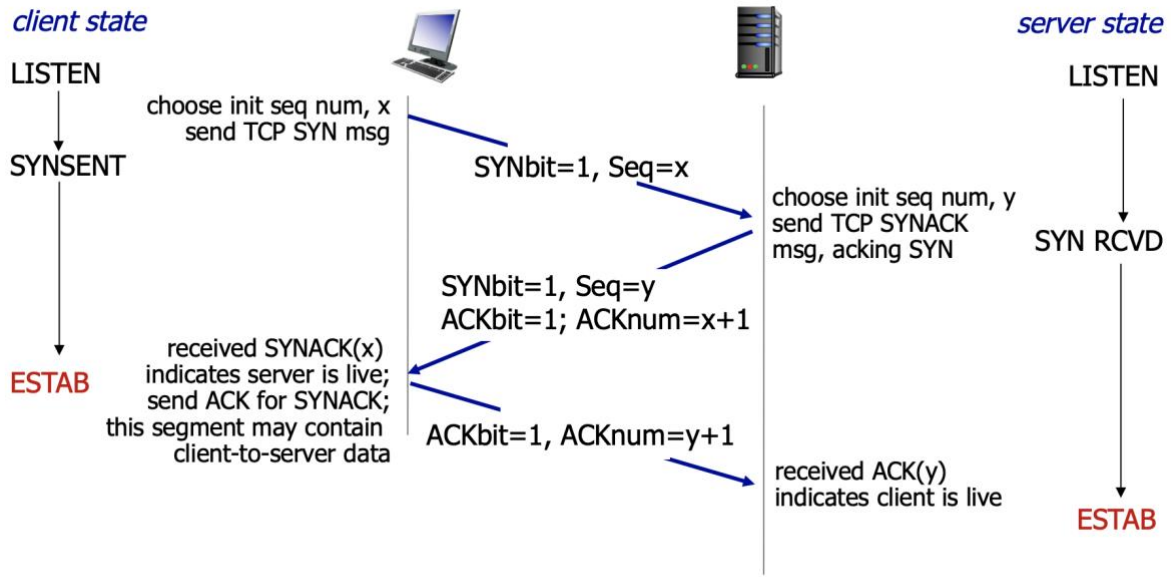
3-way handshaking is done when a new connection is established. Assume there are a client and a server. When the client calls connect() system call, it seems to simply return the system call and we were free to communicate with the server. However, more complicated things happen below the surface.

When a connect() system call is issued (this is called "active open"), a SYN packet is sent to the remote host. This SYN packet contains the initial sequence number of the client that will be used in the future. The socket state of the client changes from CLOSED to SYN_SENT. Also, because a typical client socket doesn't bind actively, we need to assign an arbitrary, unused port number to the socket (implicit bind). The server, when successfully receives this SYN packet, replies with SYNACK packet. This SYNACK packet simultaneously acknowledges the first SYN packet sent by the client and sends SYN of the server containing the server's initial sequence number in a single packet. The client finally receives this SYNACK packet and sends ACK packet to the server to acknowledge the SYN part of SYNACK packet sent by the server. The socket state of the client changes from SYN_SENT to ESTAB. In addition, the client returns from the connect() system call when it receives SYNACK packet.

Now let's observe the same scenario from the server's side. Typically, servers don't send SYN packets first. They wait for an incoming connection and start the connection when clients request. We call this "passive open". The server first calls listen() system call to change its socket state from CLOSED to LISTEN. Now a very special thing happens. When a SYN packet from the client arrives, the server doesn't modify the original socket. Instead, the server duplicates the listening socket and uses it for the connection. The original socket remains in its state as LISTEN, but a new socket changes its state to SYN_RCVD. This new socket then sends SYNACK to the client as described in the previous paragraph. The client will send ACK to acknowledge the server's SYNACK. When the server (more specifically, a socket that is newly created) receives this ACK, it changes the socket state from SYN_RCVD to ESTAB.

We have seen that as the handshaking process goes on, the state of the socket changes like a finite state machine. Your context structure should also contain state information. Also, there is something called "sequence numbers" and "ack numbers" on both sides of the connection. Although this value will be more important in the next part(when you implement data transmission), you still need to keep track of this value to send correct ACK packets.

The following diagram explains 3-way handshaking in chronological order, with socket states and correct sequence numbers.



*Borrowed from J. Kurose's slide

[Description 2] 4-way handshaking

4-way handshaking is done when a connection is closed. It can be initiated by any host (client or server) in the connection by calling close() system call. For convenience, we say that a host who receives the first FINACK packet in a connection does "passive close" while a host who decides to send the first FINACK packet in the connection does "active close." When both hosts consider themselves doing "active close", we call the situation "simultaneous close."

- **Why FINACK instead of FIN?**
  According to the official RFC 793 document (https://tools.ietf.org/html/rfc793, see page 16), once a connection is established, ACK field (ACK control bit & ACK number) must be set for every packet. FIN packet is not an exception. All real-world TCP implementations (e.g., Linux's) enable ACK field on their FIN packets. If the ACK number is not specified then last-sent data could be lost in unreliable network environments.
- **But my figure says just FIN?**
  You might be confused as TCP diagrams and documents say FIN packets while SYNACK packet

> explicitly specifies ″ACK″ field. You need to understand that those figures/docs simply omitted ACK because ACK is ALWAYS enabled after a connection establishment. In the case of SYNACK, ACK is explicitly specified because it's unusual to enable ACK field before a connection establishment. To be more clear, we've used FINACK instead of FIN in this document.

When a close() system call is issued by the client, a FINACK packet is sent to the remote host. The socket state of the client changes from ESTAB to FIN_WAIT_1, and in this new state, the client cannot send data anymore. In other words, when there's a request to send data from the application layer by write() system call, the request should be declined. Note that it can still receive data as the remote socket is not closed yet. The remote host will reply with an ACK packet to this FINACK packet. When the client receives this ACK packet, it changes its state from FIN_WAIT_1 to FIN_WAIT_2. This is the intermediate state where the client completely closed the socket but its remote host didn't. After a moment, the server will also close the socket. It will send a FINACK packet to the client. When the client receives the FINACK packet from the server, it will change its state from FIN_WAIT_2 to TIME_WAIT. Also, the client will reply to the server by sending an ACK packet.

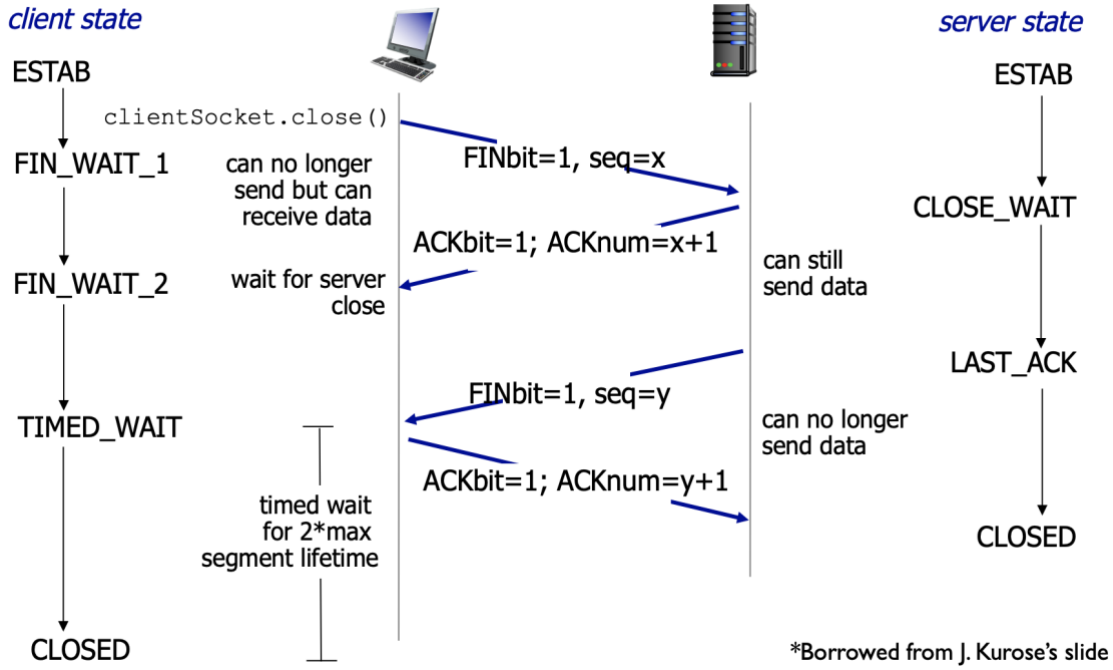
In the diagram, you will see that the client whose state is TIME_WAIT waits for 2 times maximum segment lifetime (typical MSL = 60 seconds). This is because the client doesn't know whether the server received the very last ACK sent by the client. If the server failed to receive this ACK packet, then the server will retransmit FINACK packet to the client. The (socket information of) client kindly remains alive for a moment to handle this case and makes sure the ACK packet is delivered to the server. If FINACK from the server is not retransmitted for the waiting time, the client assumes that the server safely received the ACK packet and finally removes all context information related to that socket.

Let's get back to the moment where the client sent out the first FINACK packet to the server in the very first stage of connection teardown. The scenario explained above is a very neat one, in the sense that 4-way handshaking is sequentially done from the client's FINACK then ACK and the server's FINACK then ACK. However, sometimes the order can be mixed. The server may send its FINACK before ACKing the client's previous FINACK, or the server may send its FINACK with ACK simultaneously. We need an additional state named CLOSING for this special case. For the details about this "simultaneous close", please refer to https://tools.ietf.org/html/rfc793 (page 38-39).

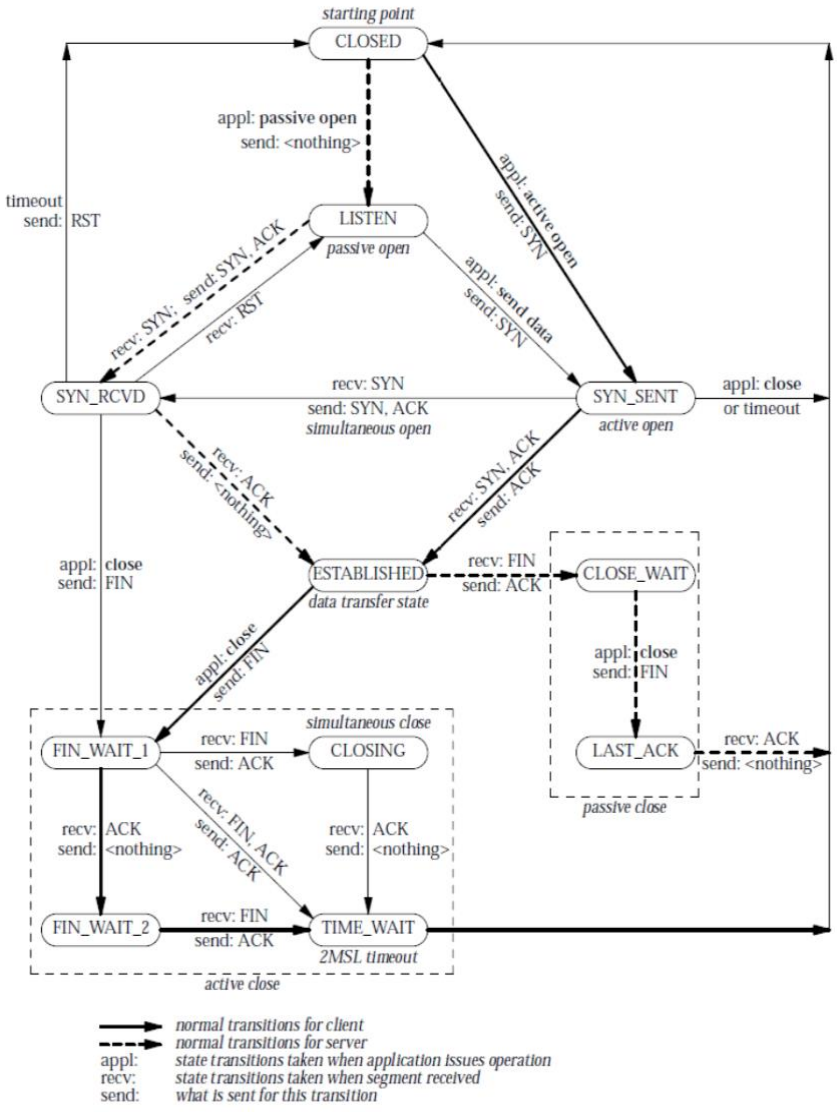
Now let's observe the same scenario from the server's side. When a FINACK packet arrives, the server changes its state from ESTAB to CLOSE_WAIT. At this point, the server can still send data but receive data that is only sent by the client before the client sends FINACK packet. This sentence will make more sense when you consider retransmission due to loss. The server should be able to handle data packets that are sent later than the FINACK packet in terms of time but earlier than the FINACK packet in terms of the sequence number. It is natural to think that the server will receive no extra data from the client because the fact that the client sent FINACK packet implies that the client can no longer send data. The server should send ACK to this FINACK to tell the client the receipt of the FINACK packet. (In this assignment, you don't have to consider retransmission.)

After a few seconds later, the server's application layer will also call close() system call. The server will send a FINACK packet to the client and change its state from CLOSE_WAIT to LAST_ACK. As the state name suggests, the server will wait for the last ACK sent by the client. At this point, the server also cannot send data anymore.

The following diagrams explain 4-way handshaking more in detail. The first diagram explains 4-way handshaking in chronological order, with socket states and correct sequence numbers. The second diagram explains socket state change as a finite state machine. The diagram also contains 3-way handshaking which was covered in the prior part.



*Borrowed from J. Kurose's slide



So far, you know how 3-way and 4-way handshaking protocol runs under connect() and close() application system calls. In the next part, you will implement data transfer over a reliable underlying channel that runs under write() and read() system calls. After receiving the application calls, your implementation should be able to send/receive data which is delivered through the payload in the packet. The sequence/ack number, length, and window field of the packet should follow the protocol described below.

[Description 3] Data transfer

After 3-way handshaking, end hosts can send and receive data as they desire. This is done by calling two system calls, write() and read(). We will first look at the sender and see what happens under the system call.

First, observe what happens in the transport layer with write() system call. Whenever there is data to be sent in the internal sender buffer, new data packets are created with an appropriate sequence number. In STCP, the application stacks data to be sent with write() call, and you can get the data that is requested from the application using stcp_app_recv() function.

In typical TCP, the packets are sent to the remote host if there is enough space in the receiver buffer. (Flow control) Otherwise, packets are kept on the sender side until the sender receives the ACK packet from the receiver, and the remaining receiver window size written in the ACK packet turns out to be large enough.

Now let's observe the receiver side, with read() system call. As the socket receives data packets from the sender, it should (first verify the checksum and) extract data from the packet, and store it in the internal receiver buffer. This internal receiver buffer stores the data until there's a request from the upper layer, a read() system call. In STCP, you don't have to implement the logic related to the receiver buffer, you can just call stcp_app_send() function instead. The application will get the values you stacked with the stcp_app_send() function with later read() calls.

To improve network efficiency, STCP requires the implementation of a Delayed ACK mechanism instead of acknowledging every data packet immediately. Upon receiving a data segment, the receiver should not send an ACK right away but should delay it for a short period (DELAYED_ACK_TIMEOUT). An ACK should be sent before this timer expires if either additional data arrives from the peer, or the application consumes the buffered data (triggering a read() call). If the timer expires before either of these events occurs, a standalone ACK must then be sent. This approach allows the receiver to acknowledge multiple incoming data segments with a single ACK packet. This logic can be implemented within the control_loop()'s NETWORK_DATA event handler by setting a flag to pend an ACK and utilizing the timeout feature of the stcp_wait_for_event() function.

When an ACK is sent, the algorithm for choosing acknowledgment numbers should follow the TCP's rule. The acknowledgment number represents the "next expected sequence number". Please note that this is different from the GBN or SR algorithm, as their acknowledgment number represents the "last successfully received sequence number". For example, if the sender sent a packet that contains 70 bytes of data with sequence number 100, the correct acknowledgment number to this packet is 170. The next packet will have sequence number 170.

The receiver should also implement flow control. The purpose of flow control is to prevent the overflow of the receiver buffer by telling the remaining receiver buffer size to the sender in ACK packet header. The sender should not send data packets that will cause the receiver's buffer overflow. Actually in STCP, the receiver window is always fixed to a constant value(3072). It will make your implementation much easier. (you don't have to consider about the flow control/congestion control in this assignment)

[Problem 1 STCP Transport Layer Implementation in transport.c (100 points)

The primary goal of this assignment is to implement the core functionalities of a simplified TCP protocol (STCP) within the transport.c file. You must implement connection management and reliable data transfer according to the specifications below, using the provided skeleton code. Your implementation will be tested for compatibility with a reference solution.

(a) **Connection Setup: 3-Way Handshake in transport_init (25 points)**
You must implement the logic for establishing a connection.

- In the transport_init function, implement the complete 3-way handshake process.

- Your code must correctly differentiate between an "active open" (when is_active is TRUE, initiating a connection like a client) and a "passive open" (when is_active is FALSE, waiting for a connection like a server).
- Upon successful connection establishment, your implementation must call the control_loop() function. The transport_init function should not return until the entire connection is terminated later in the control_loop.
- When the connection is successfully established or an error occurs during the process, you must call stcp_unblock_application() to unblock the myconnect() or myaccept() calls in the application layer.

(b) Connection Teardown: 4-Way Handshake in control_loop (25 points)
You must implement the logic for gracefully terminating a connection.

- Within the control_loop function, implement the complete 4-way handshake process.
- Handle the APP_CLOSE_REQUESTED event, which is triggered by an application's close() call. Upon this event, your code should initiate the connection teardown by sending a FINACK packet.
- When a FINACK packet is received from the peer (a NETWORK_DATA event), you must call stcp_fin_received() to correctly notify the API layer that the peer has initiated closing.
- Correctly terminate the control_loop by setting the ctx->done flag to 1 upon completion of the handshake. The TIME_WAIT state can be simplified (i.e., you can close immediately after), but you must ensure that both your FINACK and the final ACK for the peer's FINACK are properly transmitted.

(c) **Data Transfer: Sender Logic in control_loop** (25 points)
You must implement the logic for segmenting and sending application data.

- Handle the APP_DATA event, which indicates that the application has data to send via a write() call.
- Retrieve the application data using stcp_app_recv().
- Segment the data into packets. The payload size of a single packet must not exceed STCP_MSS (536 bytes).
- For each segment, construct the STCPHeader with the correct sequence number, flags, and window size. All multi-byte integer fields must be in network byte order.
- Send the complete segment (header + payload) using stcp_network_send().
- Implement sender-side flow control: The total amount of unacknowledged data (data sent but not yet ACKed) must not exceed the fixed window size of 3072 bytes. You must not retrieve more data from stcp_app_recv() than the currently available window size allows.
- When an ACK is received from the peer, correctly update the available sender buffer space based on the new acknowledgment number.

(d) Data Transfer: Receiver Logic in control_loop (25 points)
You must implement the logic for receiving and acknowledging data segments.

- Handle the NETWORK_DATA event. When a segment is received (via stcp_network_recv()), check if its size is greater than the header size to determine if it contains a payload.
- If a payload is present, pass the payload data (and only the payload) to the application layer using stcp_app_send().
- Upon receiving data, send an ACK packet back to the sender. To improve network efficiency, an ACK should be sent after a second data packet is received. If a second packet doesn't arrive, send an ACK when the DELAYED_ACK_TIMEOUT of 100ms expires. The acknowledgment number must be the "next expected sequence number" as per TCP RFC793 standard (i.e., last_received_seq_num + received_data_length).

[Problem 2] General Protocol and Code Requirements

These requirements apply to all parts of your implementation and will be graded as part of Problem 1.

(a) Initial Sequence Number: The initial sequence number for any new connection must be 1.

(b) Network Byte Order: All multi-byte integer fields in the STCPHeader (e.g., port numbers, sequence/ack numbers, window size) must be converted to network byte order before sending and converted back to host byte order after receiving.

(c) ACK Flag: All packets sent after the 3-way handshake is complete must have the ACK flag set in the header's th_flags field.

(d) Resource Management: You must free any dynamically allocated resources before the transport_init function returns to prevent memory leaks.

(e) Compatibility: Your implementation in transport.c must be compatible with the provided solution code. Your code will be tested by pairing your client logic with the solution server, and your server logic with the solution client. Passing tests with only your own implementation is not sufficient for grading.

[Appendix A] Grading Method and Packet Logging

(a) Grading Criteria: Grading will be based on the sent and received packet information captured from the stcp_network_recv and stcp_network_send functions.

(b) How to Enable Logging:

- To enable packet logging, change the LOG_PACKET value to TRUE in the 18th line of the stcp_api.c file.

(c) Log File Details:

- Filename: Log files are created in your working directory with the name format: pcap_from_[my_port_number]_to_[connected_port_number].
- Contents: The file records the flag, sequence number, ACK number, length, and window size for all sent/received packets in the corresponding order.
- Note: The logging system appends to existing files. If you run a new test using the same ports, the new log will be added to the end of the existing file, not overwrite it.

[Appendix B] How to Build and Run

(a) Build: Type the make all command in the project directory to build the server and client.

- $ make all

(b) Run the Server: Execute ./server to start the server. It will bind to a host and port, printing the address to standard output.

- $ ./server
  Server's address is 127.0.0.1:33451

(c) Run the Client: In another terminal, connect to the server using the format ./client server_address:server_port. A success message will appear if the 3-way handshake is implemented correctly.

- $ ./client 127.0.0.1:33451

[Appendix C] Testing Scenarios

(a) **Connection Teardown Test (close):**

- In the client terminal, press Ctrl+D to call the close() system call and test your 4-way handshake implementation.

(b) **File Transfer Test:**

- Run the client with the -f [filename] option to request a file from the server.
- $ ./client 127.0.0.1:33451 -f test.txt
- If successful, the received file will be saved as rcvd in the client's directory.
- You can verify the integrity of the transfer by comparing the files with the command: diff rcvd [original_filename].

(c) **Custom Testing:**

- You are free to modify server.c and client.c to create more detailed test scenarios.

[Appendix D] Environmental setting

(a) **Compatibility Test (Very Important):** Grading is performed using a [Solution Code - Your Implementation] pairing. Your client must be compatible with the solution server, and your server must be compatible with the solution client. **You will not receive points if your implementation only works with itself.**

(b) **Grading Environment:** All submissions will be graded in the Docker environment provided for PA2, with the addition of the following library
- apt install libnsl-dev

(c) **Environmental Differences:** It is the student's responsibility to handle any issues arising from differences between their personal environment and the grading server. **Regrade requests such as "It runs on my computer" will not be accepted.**

[Appendix E] Grading criteria

The grading will be done as below.

(a) Your code should be built with the default Makefile. If not, we cannot give you points.

(b) For all test cases, the order of SYN/SYNACK/ACK/FINACK flags, the corresponding sequence/ack numbers, length, and window size(fixed as 3072) should be correct. If any of them is wrong, we will score the case as 0.

(c) The cases testing close and data transfer are performed after the “active open” case. If you have a problem in an active open case, you will get 0 points for the later close cases

(d) (10%) Code: Active open

(e) (10%) Code: Passive open

(f) (10%) Code: Active close

(g) (10%) Code: Passive close

(h) (10%) Code: Simultaneous close

(i) (10%) Code: Receiver side data transfer (short file)

- This test will verify that the received file is correct and that acknowledgments are sent according to the **mandatory Delayed ACK rule (i.e., checking if a single, cumulative ACK is sent for multiple data packets).**

(j) (10%) Code: Receiver side data transfer (long file)

- This test will verify that the received file is correct and that acknowledgments are sent according to the **mandatory Delayed ACK rule (i.e., checking if a single, cumulative ACK is sent for multiple data packets).**

(k) (10%) Code: Sender side data transfer (short file)

(l) (10%) Code: Sender side data transfer (long file)

(m) (10%) Submission format (follow How to submit? section below)

(n) Any violations (wrong Makefile script, wrong file name, report is not PDF, etc.) will result in a penalty.

[Appendix F] How to submit
You need to submit the following items.

(a) transport.c

(b) report.pdf

- Submit a brief implementation report (≤ 3 pages, PDF). Cite any external sources used.