# Lab 3 Part 1 Report

I have basically created 3 structures to maintain the TCP state and I've typedef'd them into a new type for future use. These 3 typedef'd structures are namely: receiver_state_t, sender_state_t and wrapped_ctcp_segment_t. The first structure stores the last sequence number that got accepeted, whether we have received a segment with FIN flag set and a linked list of pending segments to be outputted. The second structure stores the most recently received acknowledgement, a flag whether we have received end of file on the standard input stream or not, most recent sequence number received and the one that is sent and finally a linked list of unacknowledged segments. The third and final structure maintains state info about number of times retransmission has occurred, the last send time and the ctcp segment. All of these typedef'd structures are instantiated in the ctcp_state to maintain sender and receiver state.

Then I create 2 helper functions namely: ctcp_send which sends whatever data we have and another one: ctcp_send_segment which sends an individual segment. There is another helper function which sends info segment by having the acknowledgement number as one plus the last sequence number that got accepted.

In the ctcp_init function we set the different values we have for the ctcp connection in various fields of the ctcp state such as receiver window size, sender window size, rto value etc. We also initialize various flow control variables such as sequence number beginning from 1, false to flag of not receiving end of file on standard input etc. In case of ctcp_destroy function, we use the linked list helper methods to free the memory occupied by unacknowledged segments and the pending segments that are yet to be outputted. Once that is done, we free the state pertaining to that particular connection. We use the ctcp_read function to read in data from a new segment that just arrived. In that function, we create a pointer to the new segment with the data we just read with max size as MAX_SEG_DATA_SIZE. We copy the data into our buffer using memcpy and update the most recent sequence number since in TCP everything is tracked using byte count and not using segment counts i.e., sequencing happens at byte granularity instead of segment granularity. After this is done, we send the segment with same data so that the same data is outputted on the terminal as well. If the read bytes count is -1 which means EOF was sent on the standard input stream then it means we have received ctrl D and we are supposed to send a FIN flag set segment. In case of ctcp_receive function, we compare the checksums i.e., the one we received and the one we calculate. The checksums should match, if not we discard the corrupted segments. Also, we discard the segment if the length of the segment is less than the advertised length. Then we get the data byte count and maintain the invariant such that the received data falls only within the current open window. We also check if the ACK flag was set or not, since piggybacked ACKs can be sent along with data and therefore if ACK flag was set we then update the most recent received acknowledgement too. Then we check whether received data was 0 or if the FIN flag was set in

the received segment, if yes, then we output the pending segments and update the linked list. In the ctcp_output function we output the data from a particular segment. In the ctcp_timer function we check if we got a segment with FIN flag set and EOF on standard input stream and there are no pending segments to be outputted and there are no pending segments to be acknowledged. If that is true then we wait for twice the maximum segment lifespan and then destroy the state.

The most challenging part was to get the state machine flow correct. In addition, the testing script used the same port for all the test cases so that behavior was erratic. Sometimes it used to pass all test cases and sometimes to used to fail some test cases without any reason. Therefore I made a slight modification so that each and every test case spawns client and server instance with random ports selected beforehand to get a fresh start on the communication that is taking place.