Project 2 Report

Contributors
Young Hoon Kang, 904469956
Server, some of the auxiliary functions in simpleTCP, TCPrto, and TCPutil
Michael Wang, 204458659
Client, some of the auxiliary functions in simpleTCP, TCPrto, and TCPutil

Description
In this project, we built a server and client that could be used to reliably transmit a file over the User Datagram Protocol (UDP), which is unreliable and connectionless. To transmit data reliably over UDP, we implemented some of the principles of reliable data transfer, or rdt. In particular, we used sequence numbers to indicate the sequence of data for ordered delivery, acknowledgments to confirm receipt of data, and retransmission timers in case the data failed to get to its destination. To do this, one of the things we did was to encapsulate the data inside the simpleTCP packet. The simpleTCP packet contains the sequence number, acknowledgement number, the receiver window size, and the three flags ACK, SYN, and FIN. To start the connection between client and server, the client initiates a three-way handshake as in TCP. After a connection has been established, the server then reads data from the requested file into a buffer, then sends data up to the size of the congestion window. Then, it waits for acknowledgments of the sent data from the client. If one of the sent data packets times out by not receiving a corresponding ACK packet during the retransmission timer, then the server assumes that the data packet was lost and restarts sending data packets from that packet. The client checks for ordered delivery and sends an ACK packet that corresponds to the last in order packet. The client buffers out of order packets, and when it receives the missing packets it puts them into the output file received.data. Finally, after sending all the data contained in the requested file, the server initiates a FIN connection teardown, terminating the connection between the server and client after both have sent a FIN packet. The retransmission timer in this project is adaptive, and implements the retransmission timeout based on the formula given in page 41 of RFC 793, which contains the implementation of TCP. The congestion window in this project implements TCP Tahoe, which uses slow start, congestion avoidance, and fast retransmit.

One of the biggest problems that we encountered while implementing our project was implementing data transfer from the server to the client. First, we were unable to send data as the congestion window logic was implemented incorrectly, leading to no actual data transfer. In addition, the server was not able to hold enough data about the packets it send with a simple map of time sent to packets, and therefore we created three data structures: a queue of unacknowledged packets, a hash table of packets with time sent, and a set of packets that are already sent. With these, we were able to fix our logic and configure the server to properly

retransmit lost packets. Then, we found that the server was sending messages twice in succession, which we were able to trace down to faulty while loop logic involving fast retransmission. Another problem that we found was that the server was retransmitting far too frequently even for low loss rates, which we were able to trace down to the congestion control constantly staying in congestion avoidance.

To test our code, we used virtualbox and vagrant tools to use the instance of linux provided in the project specifications. We then used ssh into two instances of vagrant, one being the server and the other being the client. First, we did not change any parameters of the network to make sure that our project was able to transmit data properly with no complications. We first tested just the SYN handshake to make sure that we were able to set up our server and client sockets properly, and to see that they could communicate. Then, we tested our SYN handshake and FIN connection teardown to check that we could add on the teardown as well onto the handshake. We then tested an empty file to make sure of the case when there was no file existing. Then, we tested the small and the large file provided. After these files worked under an ideal network, we first introduced a small delay of 10 ms, which we then increased, to test our retransmission timeout implementation. After this, we introduced packet loss ranging from 5% to 50%, making sure that the server retransmitted data properly. We then introduced packet reordering as well, to make sure that the client could send acknowledgments back properly. We tested all those conditions with both the small file and the large file, using the diff command to see if the output file received.data was identical to the input file.