Morgan Trembley

CSE 160 – Project 3 Design

I tried to follow the default transport interface file as closely as possible. I had to add a couple of my own functions and a few timers to get everything working reliably. In my implementation, I started by creating server and client start functions that would take in the cmd parameters and prepare for communication by binding the connections to unique sockets and initializing socket variables. The client attempts to initiate a connection with the server. For this, I follow the descriptions given in lecture and the transport interface. When noise was added I decided the easiest way to approach the connecting phase was to restart the connection attempt if the 3-way handshake failed. During this time, the send buffer can be written to for the transfer size included in the client command. The other design decision I made at this point was to keep a list of sockets and all the variables associated with that socket in an array. This takes up quite a bit of unused space for this purpose and it would have been better to use a list or hash table to only keep track of sockets that are in use.

Once the connection is established. A send timer starts that includes flow control. Right now, the window size changes as follows; If the buffer is less than ½ full, there is a 3-packet window; If the buffer is half to ¾ full, there is a 2 packet window; If the buffer is ¾ to $^{7}/_{8}$ full, there is a 1 packet window; and if it is full there is a 0 packet window, stop sending. In order to reliably transport with noise, each packet is added to a queue when sent. If a packet or its acknowledgement is lost, the packet will be resent. If the acknowledgement is received, any packet with a sequence number less than the acknowledgement is dropped from the queue. If a packet is received out-of-order, the packet is ignored and the last acknowledgement packet is resent to the client.

When data transfer begins, a periodic timer starts with a time of 2*RTT that was obtained during the connection phase. Each time this timer triggers, each packet in the queue is checked, If the sequence number is higher than the last acknowledgement, the packet is sent again. If it is lower or equal, the packet is dropped from the queue. This continues until all the data pertaining to a socket buffer is sent.

The last data packet initiates a close sequence that is similar to the connection sequence except, the packet queue is utilized since the first fin packet from each party needs acknowledgement in order to close properly. Once the sockets are officially closed, the socket variables are reset in preparation for a new connection.

My strategy for approaching this project was not the best way to go about things upon reflection. I started by making sure no noise worked with 1 connection, then multiple, then multiple with the same nodes. I should have implemented reliability before adding support for multiple connections while everything was less complicated. Which leads into the other error I made. Instead of breaking up all the receiving and processing of packets into their own functions, I let the built-in receive function handle everything so it was difficult to troubleshoot when I introduced more connections and the packet queue. The other mistake I made, since I didn't implement the packet queue until last, flow control was also one of the last things I addressed. This made it incredibly cumbersome to follow which packets went where, figure out how to properly edit the ad window, and insert bits of code in every place needed to ensure the flow control worked properly and all the values were updated in the socket appropriate list entry.