Christian Cam: 204446732
Hunter Gorczycki: 704461758
Group #43

CS118 Programming Project 2 Report

Instructions for Running the Program:
First, type `make` to compile our program. To begin the server, use the following command:
`./server <port number>`
Then, to run the client, use the following command:
`./client <hostname> <port number> <file name>`

To run the server/client with congestion control, use `serverCC` and `clientCC` instead of `server` and `client`, respectively.

Implementation Description:
For our headers, we decided to store the payload length, the sequence number, the window size, and the location in the file at which the payload begins. Additionally, we also stored whether the packet was a SYN or a FIN packet. The rationale for each header entry are as follows:

- Payload Length: We wanted the client to know how many payload bytes it got so that it could send an appropriate ACK number. This was mostly for the case of the last packet, where the packet size may not be the same as the maximum packet length.
- Sequence Number: We needed the server and client to know the sequence number or ACK number of the incoming packet.
- Window Size: We use this so that the client can advertise its flow control window.
- File Location Byte: We needed this information for out of order packets, where the client may receive payload bytes out of order, so this information helps the client figure out where the bytes belong in the file it's constructing.
- SYN/FIN: We needed a way for the client/server to communicate whether the packets are SYN/FIN packets.

To send our messages, we made a wrapper function for sendto so that it would also copy in the header bytes at the beginning of the packet (followed by the payload bytes). We also created a corresponding wrapper function for recvfrom, which would also copy the header bytes from the incoming packets into corresponding parameter variables (which were passed by reference).

When the client receives a data packet with sequence number X and packet size L, it will write the payload bytes to the received.data file at the header-specified location of the file simply immediately send an ACK with the number X + L.

In order to implement timeouts, we used timespec structures and the Select() function. We kept an array of information, one entry for every packet we send. This array would keep any information of the packet that would be needed in order to verify its ACK or to retransmit it. The data in the array included the sequence number, the expected ACK number, whether the packet was ACKed or not, the payload length, the file location in which the payload belongs, and the time at which the packet was sent/retransmitted. Using the send time and the Select() function, we were able to check whether a packet has timed out, and if it had, we would use the other information in the array to resend the packet. We would also use the Select() function to retransmit any needed SYN/FIN packets if we did not receive a response in time.

The array of packet information that we kept for sent packages was also the basis of our window-based protocol. Once we received messages, we would check if the first entry in the array had been ACKed. If it had been, we would shift all of the entries left until the first entry was not yet ACKed, effectively “sliding” the window to the right so that we can send new packets.

To begin a connection, we used a standard SYN/SYN-ACK protocol, where the client would first send a SYN, resulting in the server to send a SYN-ACK, after which the client would ACK that by sending the requested file name, causing the server to begin transferring data bytes.

To end a connection, we used a slightly modified FIN/FIN-ACK protocol. Once the last data packet was ACKed, the server would send a FIN message, telling the client that the file transfer was complete. After this, we used the standard FIN/FIN-ACK protocol, where the client would send a FIN, then the server would get the FIN and send a FIN-ACK. Once the client got the FIN-ACK, it would send an ACK and go into the TIME-WAIT state, during which it will ACK any incoming packets. Once the server received the last ACK, it would close the connection with the client and enter the standby state where it is waiting for a client to connect.

To calculate the TIME-WAIT duration, we had the client keep track of how many unique packets were ACKed and how long the entire connection took. Using that, we calculated the average RTT and used the maximum of 2 * RTO and 2 * average RTT to select our TIME-WAIT duration.

Difficulties:
It was a challenge to implement the timers and timeouts for individual packets. At first, I was using some clock_t functions to store when each packet was sent, but for some reason, it would break and not correctly find the difference in time when I used it along with select() functions. Then, I had tried to use a different structure for keeping time, but that structure could only handle an integer number of seconds (not allowing me to check in a millisecond granularity). I eventually arrived to using timespec functions to keep track of the times at which the packets were sent and how much time had elapsed since they were sent.

Another difficulty was figuring out how to know what exactly to send on retransmission (in order to regenerate the sent packet), so I had to just make arrays to store extra information about each packet, such as the starting byte of the file and the expected ACK number for the package. Initially, my header did not include the location in the file in which the payload belonged, so when I got to dealing with out of order packets, I had to implement that into my header, so that the client would know where the payload bytes belonged relative to the start of the received file.