CS 118 - Project 2

Jair Hinojosa, Kyle Romero

904771873, 204747283

Date Due: June 8, 2018

1.0 - Introduction

The goal of this project was to build a reliable data transfer protocol built atop of UDP, an existing unreliable transfer protocol. Our code takes inspiration from TCP, the most widely used reliable transfer protocol, and uses much of the same mechanisms such as the use of sequence numbers and acknowledgment numbers.

1.1 - Header Format

To ease the transfer of data between the client and the server, a Packet class is used which consists of both the header and the data. The header consists of the following values

int	**Int**	**int**	**bool**	**bool**	**int**
seq_num	ack_num	cwnd	syn_bit	fin_bit	data_size

Figure 1.1.1 - Packet Header

As mentioned earlier, much of the header values are taken from the TCP protocol. The seq_num and ack_num values are used to manage retransmissions and duplicate packets, two integral mechanisms in providing a reliable transport protocol. Furthermore, the syn_bit and fin_bit are used to identify two special types of packets that are used for connection setup and teardown. Lastly, the *data_size* variable is used to ease the reading and transferring of data. It is also worth mentioning that the header is of size 16 bytes which is consequently the size of a dataless packet.

1.2 - Messages

Essentially all of the messages going back and forth between the client and server are managed with the Connman class. It includes the all of the state that must be kept for the lifetime of the connection such as the sequence and ack numbers. Moreover, this class also contains all of the functions used to transport the data between the two parties.

The lifetime of a connection begins with the server listening for the arrival of the SYN packet in order to initiate the connection. This is part of the three-way-handshake used for the connection setup. After receiving the SYN packet, the server will acknowledge this packet, prompting the client to send the initial file request packet which includes the name of the file being requested. This three way handshake is managed by the *listen(), connect(),* and *accept()* functions.

After receiving the requested filename the server begins sending the file, assuming the name is valid. This involves continuously reading the file until reaching an EOF and packetizing the contents so they can be transported over the connection. At the client side, there is a continuous loop that is receiving the packetized file contents and waiting for the arrival of the FIN packet which informs the client that the full file has successfully been received. The sending and receiving of the file constitutes the bulk of the code and is covered primarily by the *sendFile()* and *receiveFile()* functions.

Once the full file has been transferred, the client and server proceed to the connection teardown procedure which began with the server's initial sending of the FIN packet. The client is then led to respond with a FINACK packet which is attempted a maximum of 10 times while waiting for an acknowledgement from the server. The client then closes its socket and its lifetime is over. On the server side, the server awaits the arrival of the FINACK and responds with the acknowledgement mentioned earlier. As per the TCP protocol, the server then waits for a 2*RTO interval before closing to make sure no packets floating in the network were lost.

As is evident by the high level description, much of the connection involves sending packets and waiting for an acknowledgement. This is accomplished through the use of the *poll()* function which waits one RTO for the ack before attempting a retransmission.

1.3 - Timeouts

Timeouts during the SYN and FIN procedure require special considerations and are handled on a case by case basis. These timeouts are specially made so as to enforce states on our client and server. The general structure of a SYN/FIN timeout is as follows: initialize variables for reading from our socket and polling, run a while loop that checks if we have received an ACK for our one SYN/ACK packet, send the packet, poll for a response of one RTO, continue to the next stage once the ACK has been received. It should be noted that this is a general structure and even more specific regulations must be imposed for special packets such as the first SYN or the last ACK packet.

During the main transmission of our file, unACK'd packets are kept track of in a linked list that holds important information regarding our packets. For example, we store a pointer to the packet and the last time that the packet was retransmitted at. In our main file transmission loop, we first poll for ACKs from the server and then check our list of outstanding packets to see if any of them have been ACK'd. If they have, we remove them from our list of outstanding packets. Then, we check the remainder of the outstanding packets to see if any of them have timed out using the "chronos" time library. Since we have stored a pointer to our dynamically allocated packets, retransmitting a packet is as easy as passing the pointer to a function that sends packets. Of course, one must also be careful to update the most recent time that the packets was retransmitted at.

1.4 - Window-based Protocol

Our congestion window prevents the server from overloading the network with too many packets. The base of our congestion window is determined by finding the oldest unACKed packet. We determine whether or not we are able to send out more packets into our congestion window immediately after we check to see if any packets have timed out and need to be resent. Using some boolean algebra and some assumptions regarding the distance between two sequence numbers, we are able to calculate whether a packet is able to wraparound and still be within the congestion window.

On the client-side, the congestion window is handled by ensuring that we can deal with out of order packets by buffering them in a linked list. We are able to sort this list by using the same boolean algebra and assumptions that we used to wrap around the congestion window. During transmission, if an in-order packet is received, then that packet is written immediately to our output file along with any in-order packets that follow it in the linked list that stores our buffered packets. When our server shuts down communication with the client, which it doesn't do until the client has ACK'd all packets of the file that the server has sent, the client ensures that all buffered packets are appended to the rest of the file in order. For all linked list implementations we utilized the C++ STL List structure but made some custom functions that are able to handle the data in this project well.

2.0 - Difficulties

The first issue we faced was in the serialization of the packet class and converting the class into a byte buffer. Luckily, we were able to find a somewhat elegant solution using the *reinterpret_cast()* function that eased the transfer of packet. Moreover, we had a lot of trouble getting the packets to be added to the file in the right order and in the handling of duplicate packets on the client side. For example, our group spent several hours debugging an issue that we later resolved by simply making sure that our client-side buffer did not already contain a copy of an incoming packet.

3.0 - Conclusion

During this lab we learned a lot about reliable data transfer and the TCP/UDP protocols. Our group faced several tough challenges in implementing some of the finer details of this lab but were we able to persevere and complete the entire required implementation. The scale of this project required us to utilize good programming practices such as Object Oriented Design and good error-checking/logging. We are grateful for the opportunity to learn about subjects such as these. Overall we found this project to be both challenging and rewarding and will never forget the time we spect working together on this delightful little puzzler.