CS 118 report

Xilai Zhang 804796478 Haiying Huang 804757410

1. **Header format**

Sequence number (uint16_t)	Ack number (uint16_t)	Cwnd size (uint16_t)	Data size (uint16_t)	Ack flag (boolean)	Syn flag (boolean)	Fin flag (boolean)

The header of each packet is designed to include the followings: a sequence number named seq_num of type uint16_t, an ack number named ack_num of type uint16_t, a congestion window size named cwnd of type uint_16, a data size named data_size of type unint16_t, an ack flag named ack_flag of type Boolean, a syn flag named syn_flag of type Boolean, and a fin flag named fin_flag of type boolean. We have four values of type uint_16t, which in total is 4*16=64 bits=8 bytes. And three Boolean types which is 3*1 byte=3 bytes in total. And since our header is declared as a struct, the alignment padding will make the size of it the closest multiple of 4, which is 12 bytes.

2. **Messages**

The server and client first establishe sockets initializations very similar to that we did in project1. After that the client and server will do a three-way handshake: sending syn packet, syn-ack packet and ack with or without payload. If the three-way handshake is successful, the server will assign a connection number to the current connection. This connection number will later be used to save the files transmitted by the client. For example, if the current connection is the second successful connection, the server will save everything into 2.file.

The messages are sent between the client and server using UDP functions recvfrom and sendto. The messages will be first stored in a buffer, and the buffer will later be converted to our self-defined structure Packet. From the packet we will then extract useful information such as sequence number and ack number. These information can be used to update parameter values such as next ack number, and also help us to print the required information to standard output.

In the case that server successfully receives all packets and stores them in [connection number].file, client and server will enter into fin stage. Client will send a FIN message and server will reply with FIN ACK followed by a FIN message. The client then knows the server want to shut down and sends an ACK message. The client will also wait for 2 seconds in case that the last ACK was lost. Note that the client can still shut down properly if the first FIN ACK from server is lost but the FIN from server arrived successfully.

3. **Timeouts**

For small values of timeouts, such as time out for a single packet in the handshake or shutdown process, we use poll structures. The time out value is specified in the call to poll function. When we are transmitting large amount of data, we used the timeout algorithms kindly provided by TA: (1) Every time a packet containing data is sent (including a retransmission), if the timer is not running, start it running so that it will expire after RTO seconds (for the current value of RTO). (2) When all outstanding data has been acknowledged, turn off the retransmission timer. (3) When an ACK is received that acknowledges new data, restart the retransmission timer so that it will expire after RTO seconds (for the current value of RTO).

4. **window-based protocol**

To achieve control over cwnd size and decide what to send, we used the deque structure available in C++. We maintain a two ended list of struct packets. We will push in new packets and remove old packets constantly. The first packet in the queue will be popped if its ACK is received. The correct number of new packets are then pushed to the queue based on calculation of cwnd size. These packets will then be sent. Next ack number and next seq number are also updated accordingly. We also have a variable that tracks the sequence number of the first packet. The receiver will also maintain a two sided queue to track its packets. All of these variables together implement the logic of slow start and congestion avoidance. We have also secured that the transfer is robust under conditions of packet loss using the tc command. For example, after executing “tc qdisc add dev lo root netem loss 1%”, the file can still successfully be transferred between client and server.

5. **Difficulties**

One of the difficulties we had was when implementing the three-way handshake. We were expecting the recvfrom function to fill in the sockaddr * of the source address. But it did not. We were confused because the linux man page says that recvfrom should have the ability to auto fill. We then discovered that it was because we set the addrlen (address length) to be 0 when we were transmitting the empty packet. It turned out the recvfrom will only be able to perform auto fill when addrlen is nonzero values.

Another difficulty we had was when pushing in new packets to the cwnd. Initially our logic compares the starting sequence number of the queue, add it with the cwnd length, and then take modulo to obtain the ending sequence number of the queue. The logic keep pushing packets into queue until the sequence number reaches the ending sequence number of the queue. However, if the sequence number wraps around, this logic will keep pushing packets into the queue forever. We realized this difficulty and designed a logic to first check if sequence number wrap around will happen. If the sequence number indeed wraps around, we will set a breakpoint at the wrap around place. We then push packets from after the breakpoint to warp around sequence number, and also push packets from starting sequence number to breakpoint.

There were also other difficulties, but the biggest difficulty we had was we thought the due date, June 8th 12:00 am means June 8th 24:00. But in fact it means June 7th 24:00. This changed our plan so we did not sleep and worked for a straight 48 hours starting from June 6th.