Program Description

This program implements a TCP-like protocol on top of UDP. It supports variable window size, and can recover from data loss, data corruption, and data delay. Basically, I implement the TCP 20 bytes header as follow:

[Source port][Dest port] [Sequence Number] [Acknowledgement Number] [Flags][Receive window] [Checksum *][Urgent] [Payload …]*

I run my sender, receiver and the link emulator on one machine. The link emulator tool we will use is called *newudpl*. The link emulator acts like a "proxy", i.e., the sender is configured to send packets there and the proxy is configured to send packets to the receiver and it can drop, corrupt, reorder and delay packets. The receiver process sends its acknowledgements directly to the data sender.

I use timeout mechanism for retransmission as per TCP (without fast retransmission), such as RTT estimation and update of timeout interval, to handle packet loss and use checksum in UDP/TCP to check if the packet is corrupted, and use ack number & sequence number to check if the packet is delivered in order or not. I use a *FIN* request to signal the end of the transmission. For supporting the variable window size, I implement my own mechanism in this program.

How it works:

A) The TCP Segment Structure

I used the Standard TCP Header to pack and unpack the payload of my packets. In other words, all my packets had 20 Byte headers that represented the packet's *source port(2 bytes), destination port(2 bytes), sequence number(4 bytes), acknowledgement number(4 bytes), header size (1 byte), flags (1 byte including **ACK** bit and **FIN** bit), window size (2 bytes), checksum (2 bytes), urgent pointer (2 bytes).*

B) Sender and receiver

My code works by:

(1) First having both the sender and receiver setting up the client segment sending socket, client ack socket, server segment sending socket and server ack socket.

(2) As well as setting up other descriptors like client/server log file.

(3) Sender then disassemble the original file into segments and stores them in buffer(an array) from which it will send later on.

(4) Sender then uses the UDP sockets to send segments to the receiving side of emulator, in the order of sequence number from the buffer.

(5) If the packet is not lost, uncorrupted and in order, and transmitted successfully to the receiver. Then the server will send an ACK that has a next expected sequence number to client.

If the receiver receives either out of order or corrupted packets, it will do nothing.

(6) The sending window only moves on to the next segment when a received ack belongs to the segments at the window base.

(7) Once all the packets are sent, close all the sockets.

C) Packet Corruption, Disorder and Loss Recovery mechanism

Set just one timer for all segments sent back-to-back at once (timeout interval is updated through the TCP estimated RTT formulas when receiving an ACK, if it does not belong to retransmitted segment).

If an ack is received within the timeout interval and it matches the sequence number of window base (left bound),

(1) Its reception is logged at the sending side and prompts the sending window to move right by one and send the non-sent packets in sending window.

(2) Reset the very one timer.

(3) Calculate the timeout interval through the TCP estimated RTT formulas when receiving an ACK, only if it does not belong to retransmitted segment.

However, if an ACK is not received in the timeout window,

(1) Double timeout interval.

(2) Resent all the packets in the sending window.

(3) Write log file and reset the timer.

D) Potential Problems

I checked my code in my local machine (Mac OSX with Apple Silicon), however I am unsure whether my code works elsewhere, like on other OS. Beyond that the code should work as expected. Please make sure to follow the instructions that I wrote for the sample run to make sure you are running the code appropriately.

Besides, my program will double the timeout interval once packet timeout occurs but I have not implemented fast retransmission. Thus, the timeout interval might be large (dozens of seconds or more if worse), and hence the waiting time to resend packets might be very long. I have tried to reduce the time by setting a relevant small delay in emulator(like 0.3 seconds).

Including files:

My program includes 3 program files and other files like *'source_file.txt'* used to transmit and *'log.txt'*:

- main program files: *Segment.py, client.py, server.py*

1. *Segment.py*:

Implement several handy utility functions in this file. $make\_segment()$ is used to assemble the segment, including the header and payload; $unpack\_segment()$ is responsible for extracting each fields from the segment, and also $total\_2\_bytes\_sum()$ is used to calculate the total 16-bit sum of the segment. Finally, $calculateCheckSum()$ is used for checksum fields in TCP header.

2. *client.py*

Read from the file and sends the packets to the designated receiver. Maintain the sending window and timeout mechanism for packet corruption, disorder and loss Recovery mechanism. And update timeout interval as per the TCP protocol.

3. *server.py*

Receive the sender's datagram packet, check whether the delivered packets are uncorrupted and in order. Send ACK to the sender if nothing wrong with the received packets.

- other files: *source_file.txt, received_file.txt, server_log.txt, client_log.txt*

1. *source_file.txt*: Original file used to be transmitted from client to server.
2. *received_file.txt*: File received by the server.
3. *server_log.txt*: Log file for server. Records the action of server.
4. *client_log.txt*: Log file for client. Records the action of client.