Nicole Niemiec
Programming Assignment 2: Analyzing packet traces (PCAP)
CSE 310, Spring 2022
Instructor: Aruna Balasubramanian
Due date: March 10 2022, 9.00pm

High Level Summary

To answer Part A, the code first takes in a .pcap file name and attempts to read it (lines 399-403). Using the pcap module from the dkpt library, we pass in the open file to the dp.pcap.reader and call our own method, read_pcap, in order to analyze the pcap file. In read_pcap, we go through each connection, checking if each packet contains Ethernet frame, grabbing the IP. Once we have the IP data, we can get the TCP data. Here, we have to check if the IP of the sender is equal to the expected sender IP address and same for the receiver. Here we can create a Packet object to wrap the packet with, which holds the ethernet information, the IP address, the TCP data, source port, destination port, source IP address, and destination address, and the timestamp. Then, according to their source and destination port numbers, we create assign them to a distinct flow. We can think of each flow as a list of all the packets going from one specific pair of ports. If there's already a flow for this pair of ports, we append the packet to the current flow. If there isn't, then we create a new flow. At the end, we will have TCPFlow objects that represent each distinct flow in the file. Our TCPFlow objects have the instance variables of the sender, the receiver, the flow, the window scaling factor, as well as the list of packets in the flow after we've removed the "handshake" in the beginning of the flow. We do this by a method called "__ignore_handshake()" where we identify the packet with SYN, the packet with SYN, ACK, and the first ACK after that one.

The read_pcap() method returns with the distinct flows and we then pass this into another function called "print_results()." This function is supposed to loop through all of the flows and print out the information requested in Part A and Part B. Inside the loop, first we grab all the identifying information of the flow (i.e., the source port number, destination port number, the source IP address, and the destination IP address) and print in a table-like manner, using a "PrettyTable()" object from prettytable.

Next, we need to look for the transactions in the flow. We call "get_transactions()" on our flow object. Here, we loop through the packets in the flow searching for the first two transactions from the sender to the receiver. Then, we loop through the flow again to grab the first two "ACKS" that are sent back to the sender as a result, giving us four items in our array. The array holds the source IP, destination IP, ACK number, sequence number, window size, and timestamp of each packet and we return this back to the caller. We then loop through these and add them as rows for our table, printing out the results.

Then, we have to find the sender throughput. We call our function "sender_throughput()" and first identify the last FIN,ACK packet. Once we find this, we find the index of it and essentially chop it off of the flow array. We then sum up the amount of data sent as we loop through the array of packets, returning both the sum of the data as well as the time period (in order to do some math in our caller function).

Next, we find the first three congestion window sizes. To do this, we call "estimate_congestion_win_size()" and here, we identify the first single" ACK" packet that matches the receiver's IP. We note down the timestamp of this packet and then note down the start time of the flow and take the difference as an RTT. Then, we grab all of the packet timestamps that match the sender's IP and append them in an array. After subtracting the start_time from each of these and finding the breakpoints where we can find the window sizes, we then append a list of window sizes only if the current indexed time is greater than the breakpoint. These are quantified in seconds. It is noted that there's a fixed sense of increasing between the congested window sizes. For example, we get [10, 20, 33] for the first flow, [10, 32, 43] for the second flow, and [10, 22, 33] for the last flow.

Lastly, we find the number of retransmissions due to triple duplicate ACK and the number of retransmissions due to timeout. We know that a packet has been "triple ack'd" if we can find more than three packets with the same ACK number. Likewise, we know that a packet is retransmitted as a result of timeout if the sequence number is repeated once or more. We use this in our method "find_fda_and_timeout_retransmissions()" in order to follow this logic and return a pair of counts: the triple ack count and the timeout count.