Joshua Landron
CSS432
Homework 3

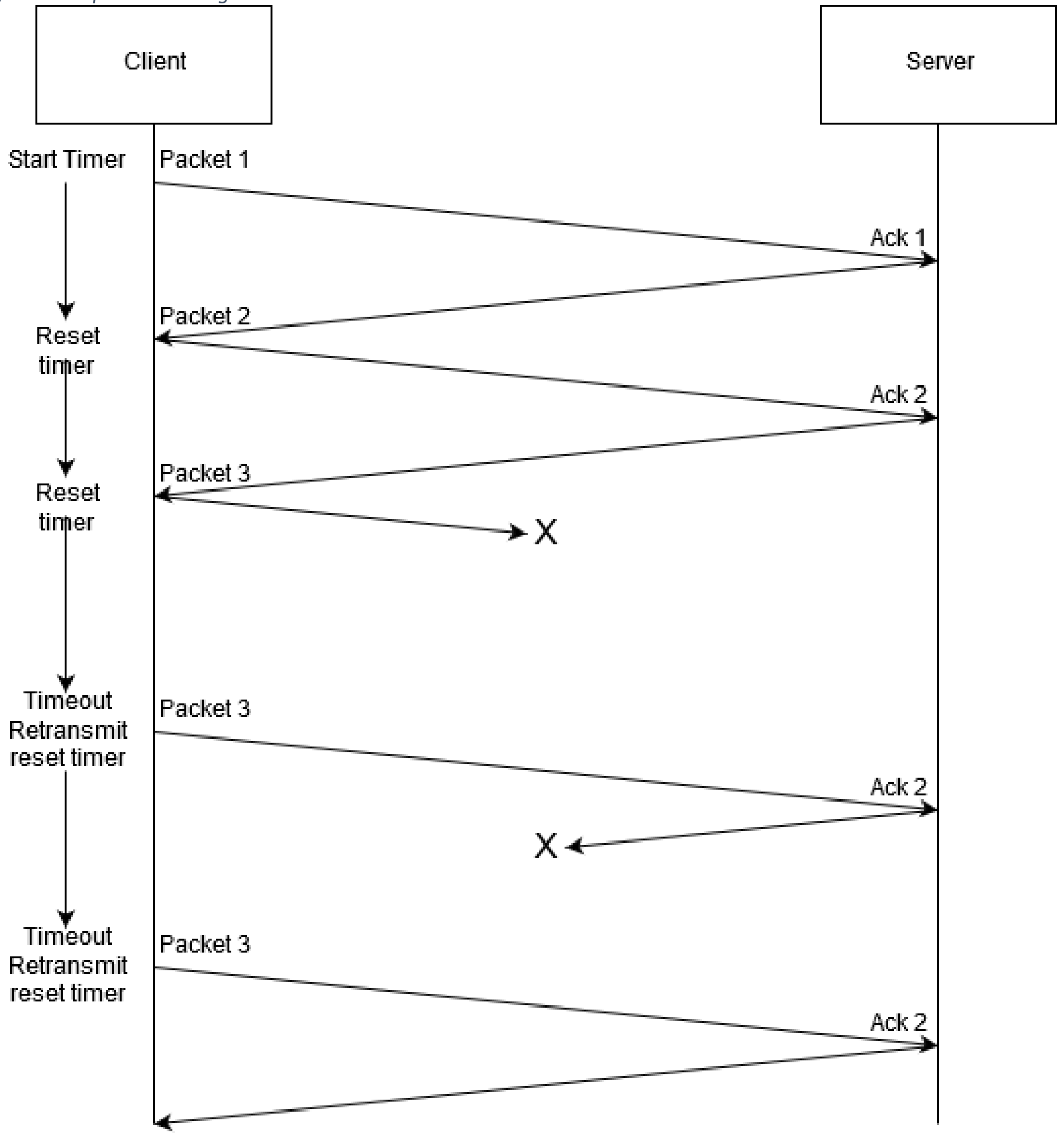
Contents

Algorithm descriptions

Stop-and-wait

The stop-and-wait client transmit a single packet and awaits an ack from the server regarding that packet. The server receives packets and if the packet received is in the correct order it sends and ack and awaits the next packet. This simple process is illustrated in Figure 1.

Figure 1: Stop-and-wait algorithm overview

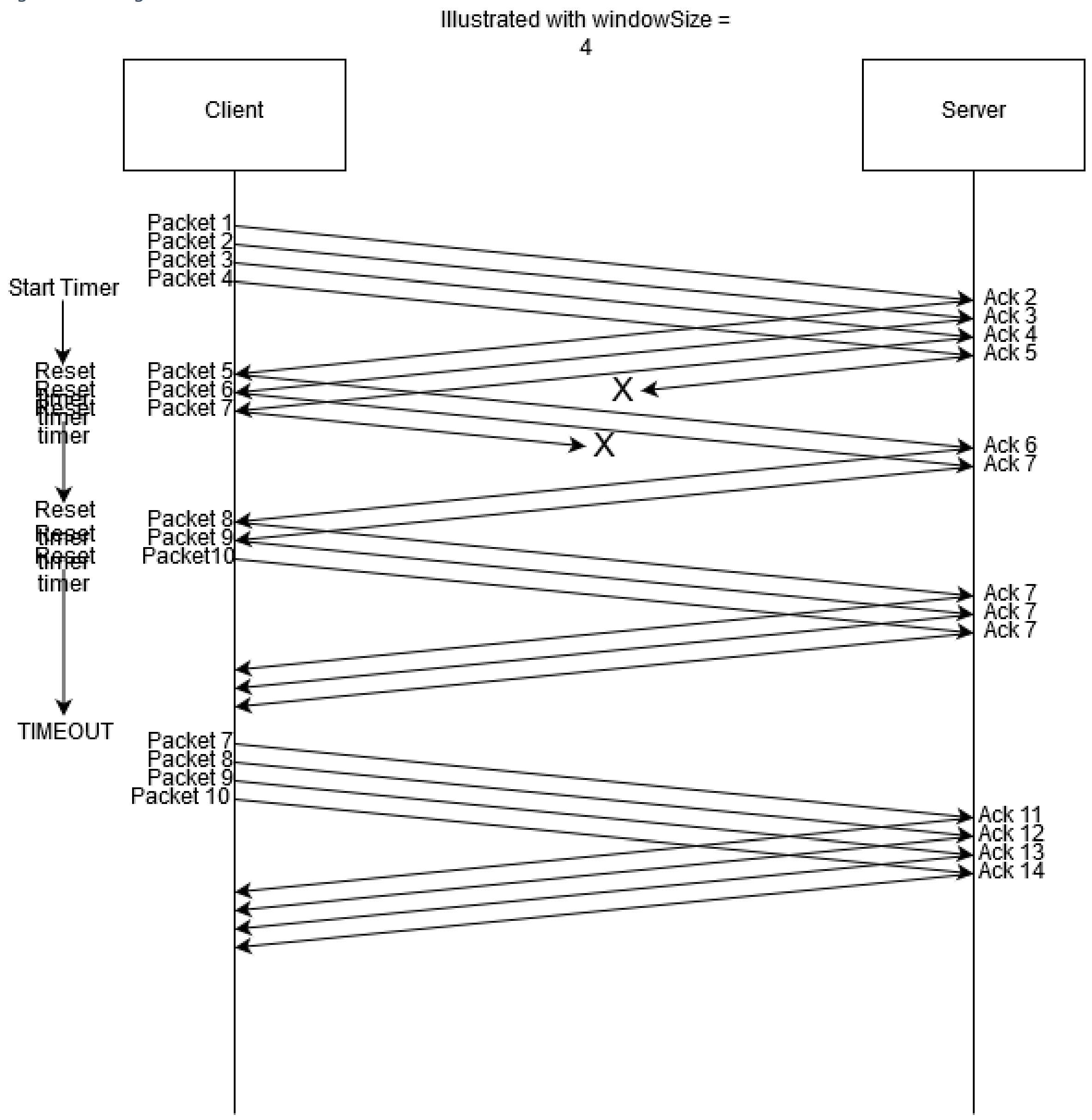


Sliding window

The sliding window algorithm that I implemented is very different from the stop and wait algorithm. The client sends the number of packets as the size of its window and marks these packets as sent. Then it waits for either a timeout or an ack from the server. Upon receipt of an ack, the client sets the base of the window to the value of the ack. This means that lost acks are not important as the server always sends acks for the next expected packet. This behavior can be seen in figure 2 which shows a packet loss and an ack loss.

The server in this algorithm waits for packets from the client and upon receipt of a packet sends an Ack and marks it as Ack'ed. The server keeps track of which packets it has received and always sends acks for the next expected packet. This method allows the client to move its window rapidly and ignore out of order acks.

Figure 2: Sliding window



Output snapshots

Example output of server is contained in ServerOutput.txt.

Example output of Client is contained in ClientOutput.txt.

Performance Evaluation

Stop-and-wait

The stop-and-wait algorithm completed very slowly due to the fact the client must wait for every ack to move forward and send the next packet. This operation is the simplest way to guarantee that all packets are sent, but in testing performed 13.5 times slower that unaltered unreliable data transfer. The average completion time for stop-and-wait over 20000 packets was 3,547,427 microseconds.

Sliding window

The sliding window approach to data transfer made large improvements in completion time over stop and wait once the window size increased beyond 1.

A window size of 1 acted exactly like stop and wait, with the same performance results as can be seen in figure 3. Adding a window where the client could send packets without waiting for a server response allowed the client to process data as fast as possible within its window, then move between waiting for acks and sending data. The fact that the server sends the next expected packet sequence number allows the client to move its window rapidly which led to great improvements in performance over a pure go-back-N approach.

Sliding window with random drops

When random packets drops were introduced the execution time increased as expected. The sliding window of size 30 dealt with this problem well and execution time increased linearly at a much slower rate than the window size of 30 which was essentially stop-and-wait. These execution times are shown in figure 6.

Discussion of results

Stop-and-wait vs Sliding window

Over all windows sizes greater than 1 sliding window shows a 5x improvement in completion times over stop and wait with the algorithm eventually settling into a state of gradually improving after the window size of 20. Figure 3 shows the average completion times against window sizes, while figure 4 shows the result of three separate test runs. Figure 4 shows interesting behavior resulting from the sliding window algorithm.

Influence of window size

The completion time quickly decreases as the window size increases from 1 to 7, than as the window size increases from 7 to 15, the completion time becomes less predictable and each run deviates from the others as seen in figure 4. This behavior starts to fall away after a window size of 15 and this seems to indicate that the client is running into issues with the window size being to small to generate the best results. The average completion time increases between a window size of 7 and 12. While this behavior may depend heavily on the packet size, all packets sent were the same size and no other sizes were tested in this process.

Figure 3: Average Completion times of after 3 runs of test 3

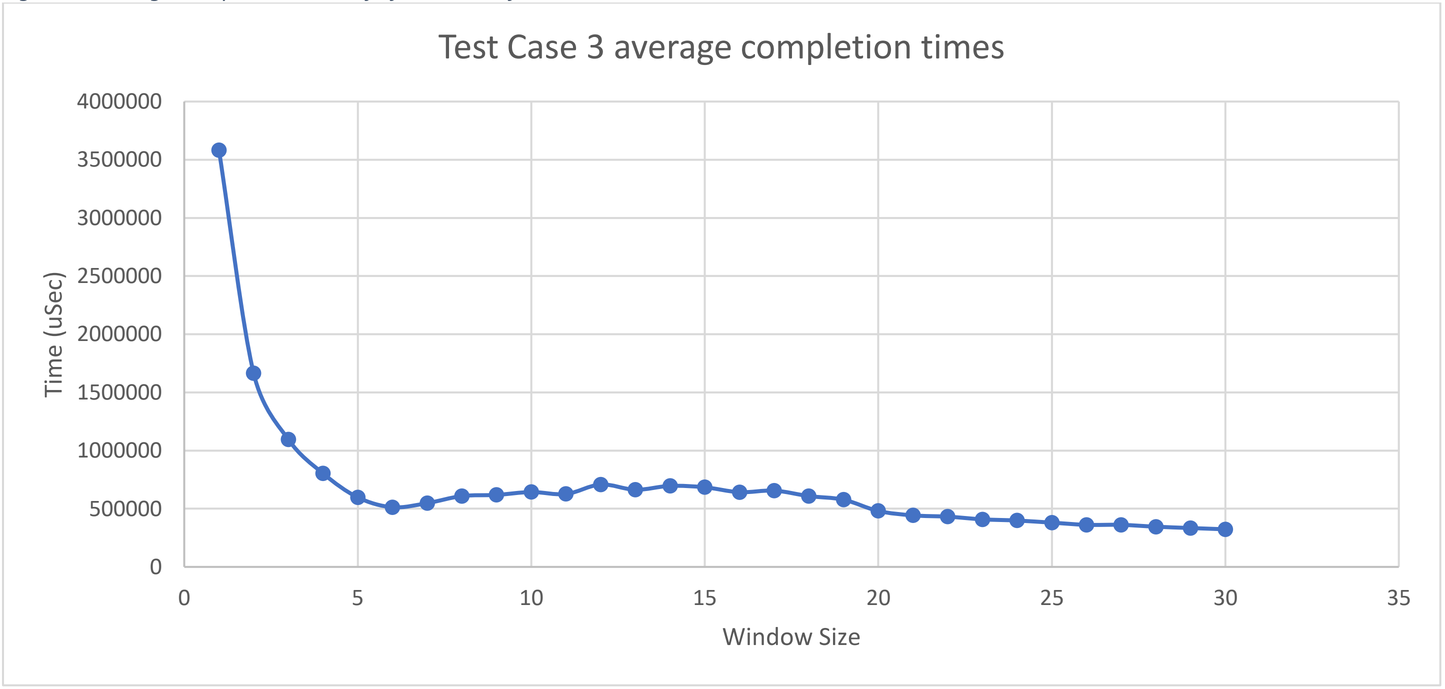
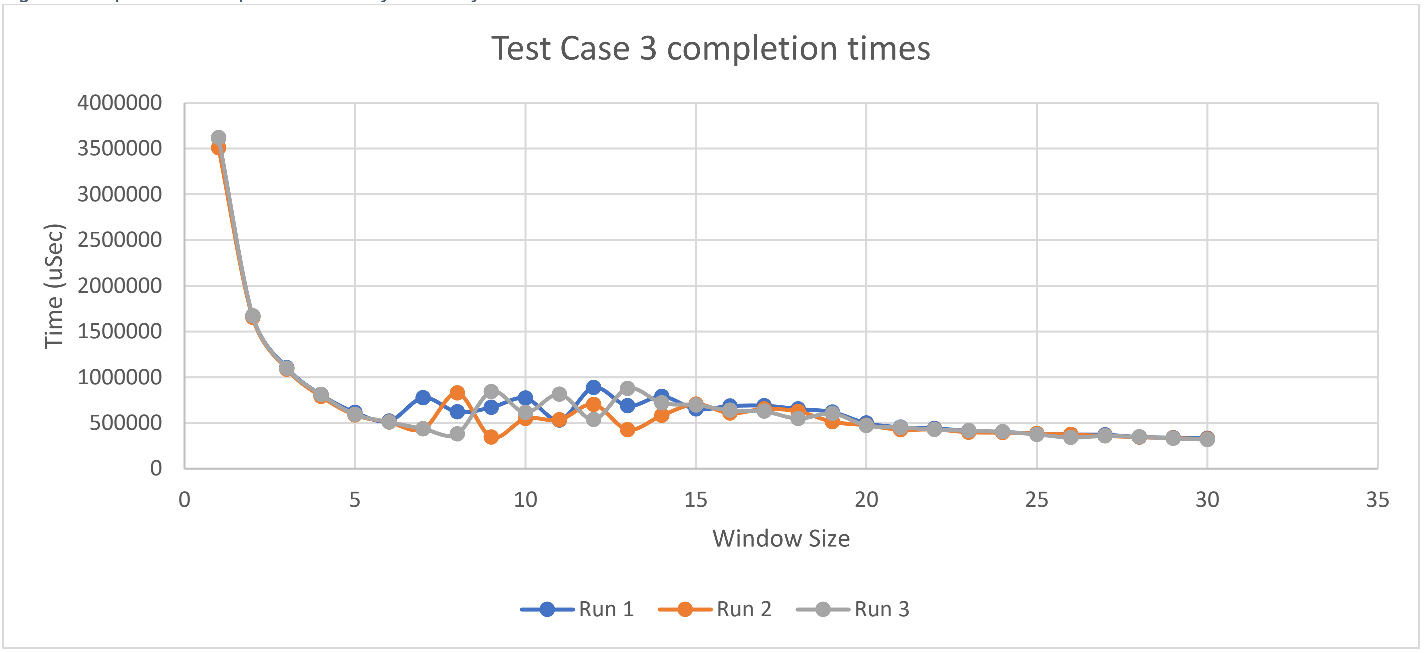


Figure 4: Separated completion times of 3 runs of test 3



Retransmissions and drop rates

Retransmissions were minimal in each algorithm during my testing and until I forced dropped packets to occur in case 4, this stayed consistent at between 0 and 10 transmissions being the most common occurrence. Figure 5 shows the retransmission occurrences plotted against the percent of packets dropped by the server. As packet loss approached a 10 percent chance, the stop-and-wait algorithm (sliding window with window size of 1) rapidly increased in execution time at a rate of about 1500ms per percent packet loss. The sliding window algorithm (window size = 30) did not have as much trouble dealing with these losses increasing at a rate of about 227ms per percent packet loss. A similar result happens when completion time Is plotted against percent change of packet loss as shown in figure 6.

Figure 5: Retransmission rates of packets

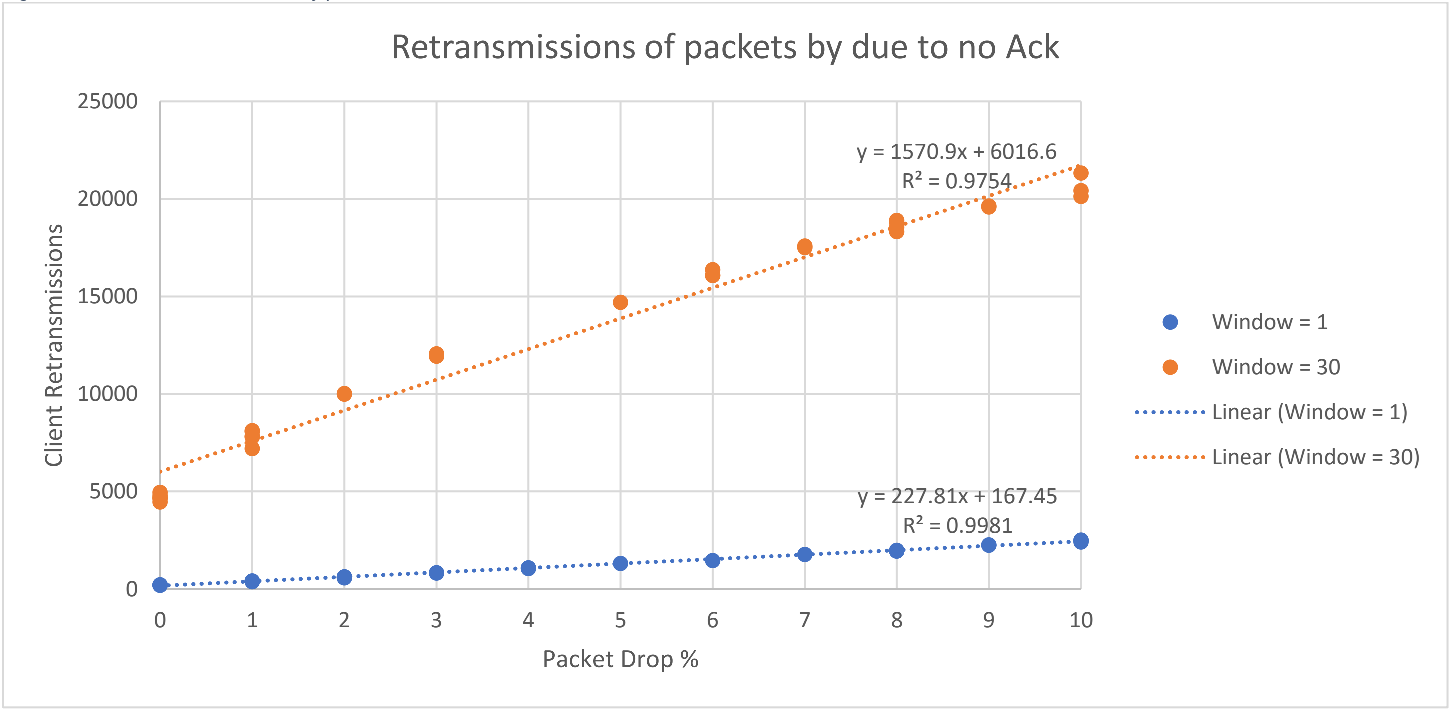
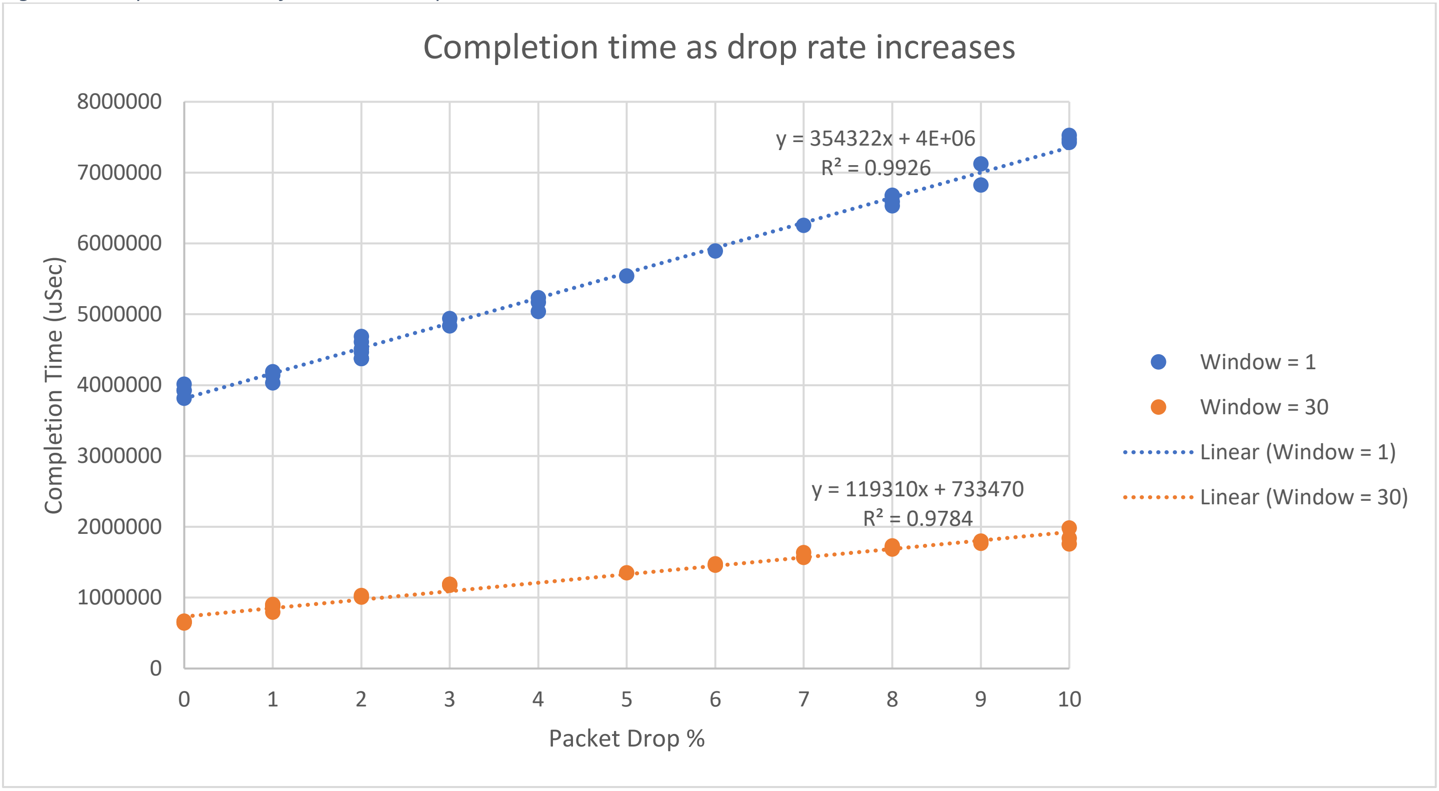


Figure 6: Completion times after random drops



Execution Notes

The .cpp files submitted depend on hw3.cpp, Timer.cpp, and UdpSocket.cpp being in the same directory to compile and run. I have included a compile.sh script to compile both udphw3.cpp and udphw3case4.cpp. Both of these files have included references to hw3.cpp and hw3case4.cpp that should not be included in the g++ command.