Supporting Documentation

I used RDT 3.0 as the primary resource for this project. The notes in the slides provided all of the necessary graphics and description to accomplish the objectives of the project. The diagram below describing the different window states was especially helpful.

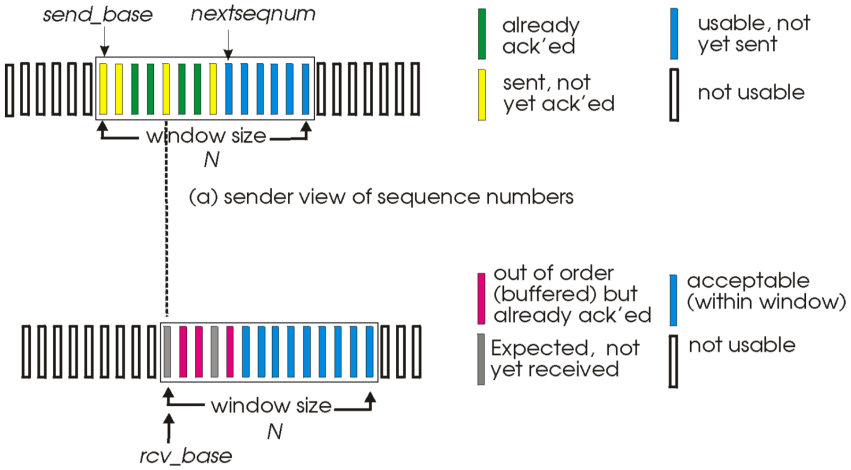

Figure 1: Select Repeat Window Description

TCP Header Selection

I only used a small subset of the TCP header elements. In order to minimize the overhead in the message and keep the transmission time to a minimum, I worked under the assumption that the sender and receiver would have prior knowledge of the other.

- Source and Destination Port
  - This is included in the UDP header that we are using so it was un-necessary to include this twice. I'm also working under the assumption that there is only one client attempting to send to the server.
- Sequence Number
  - I used a 1-byte field for the sequence number. Since the maximum window size is 64 messages and we only need twice that for the protocol to work properly, I used a smaller sequence number field than used in TCP
- Acknowledgement Number
  - I didn't use this field because the data is only being transferred from the client to the host and not the other direction. For this reason, I used the sequence field to hold the acknowledgement number from the server back to the client.
- Flags
  - We are not establishing a connection because there is only one client and one server so I was able to reduce the number of flags needed. I used a flag for ACK, NACK, and FIN. The server ACKs or NACKs the messages from the client and the client uses the FIN flag for the last message.

- Window Size
    - I worked under the assumption that the server and client have the same window size so it was not necessary to include this in the header.
- Checksum
    - Instead of the checksum I used a 16-bit CRC. I did this because the CRC has a much higher probability of detecting an error. It turned out to not be necessary because the lower layers provided this service.
- Urgent Pointer
    - I didn't use this field because all of the messages have the same priority.
- Optional Data
    - The data was pretty straightforward and all of the messages carry the same type of data. It's pretty much irrelevant for the client and server to know what type of data is being sent, so this field was un-necessary.

Results

To gather the results, I use a test fixture script to sweep the latency, latency variance, drop percentage, window size, and the minimum segment size. I aggregated all of the results on the bits per second and one other parameter to create the two-dimensional plots.

As you can see from the graphs, the bits per second increased linearly with the increase in window size and MSS. It's no surprise that this is the case for both the MSS and window size because they both increase the number of bytes that are in flight.

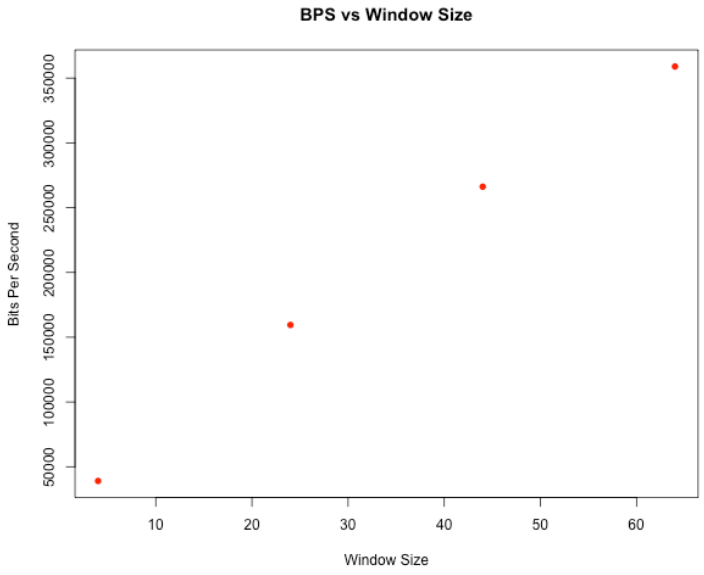

Figure 2: Bits per second versus window size

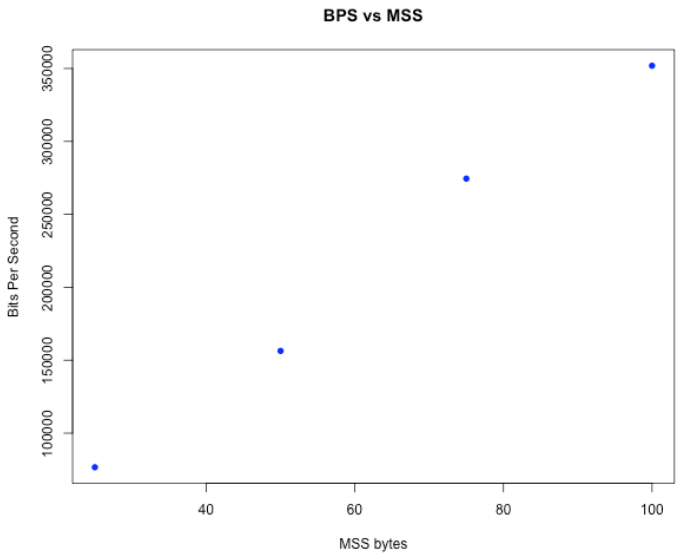

Figure 3: Bits per second versus MSS size

The decay in bits per second is more exponential as we can see in the following two figures. The latency decay is sharper than the loss delay. The throughput increases greatly as the propagation delay approaches zero.

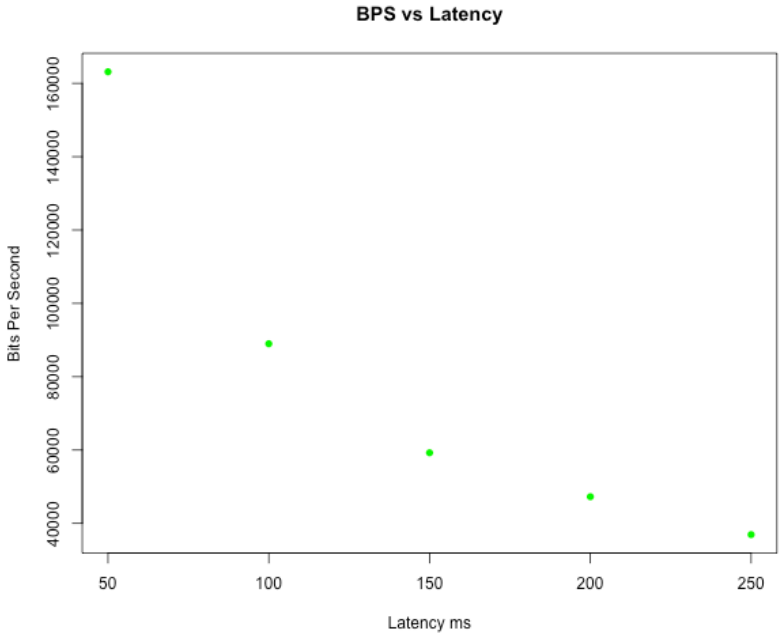

Figure 4: Bits per second versus latency

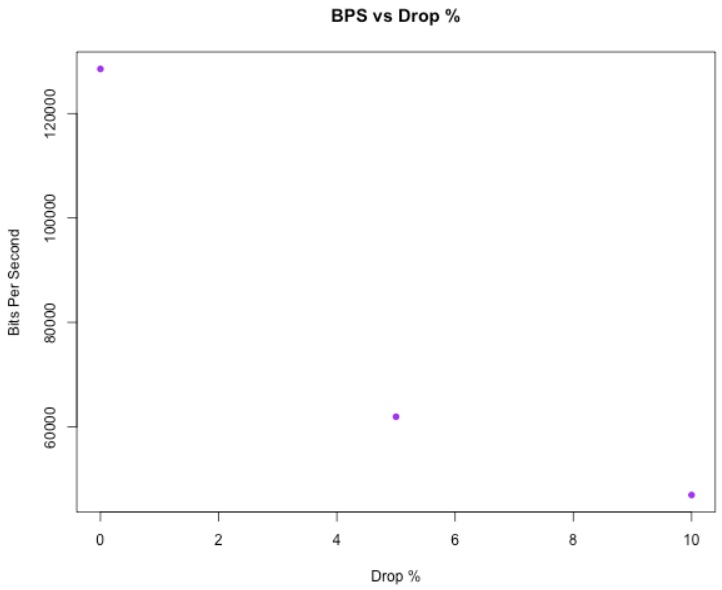

Figure 5: Bits per second versus drop percentage

Issues

The main issue that I had with the project was properly wrapping the sequence numbers and setting the acceptable range of sequence numbers that the server would accept. This was difficult to debug because it is hard to isolate the two asynchronous services and alter the behavior of the system without affecting the results. I could print out the sequence numbers and different status elements but the increased activity can alter how the system behaves.

Another smaller issue that was causing the server to deadlock was not advancing the receive base pointer with every packet. At first I was processing all of the pending messages from the client before advancing the receive base. I noticed that the receive base was not advancing and the server was nacking everything sent by the server.

The most elusive issue that I solved was the client saving the ACK messages from the server outside of the window size. At first I was recording all of the ACKs from server without checking the state of the sequence number in the window. I spent a lot of time debugging the server because the client was finishing the transmission but the server was reporting that it had not received all of the data. To solve the problem, I made sure the client was expecting an ACK with the given sequence number before saving the state.

Improvements

Assuming that we cannot improve the channel improving the window size or MSS of the sender and receiver would provide the most effective increase in bits per second. Some functional improvements could be made to increase the usability of the client and server. Sending the name of the file instead of assuming a file name would be a good improvement for the system. It would also be useful if the server could accept multiple different files from different server.