Analysis

TCP VS UDP

For the majority of development, I used TCP. With the channel already dropping packets, it seemed like adding to that by using UDP would be a negative. In addition, using TCP allowed me to send variable size data – UDP meant that simple ACK responses could be 50k Bytes.

Congestion control with TCP

Because TCP took care of its own congestion control, the protocol suffered from timeouts. With large packets and a large send window the receiver side was not able to keep up. It had no notion of this, as it just replied with ACKS, so it was left to the client to time out. Timeouts were costly performance wise, as my SRTT was set to 2-3X RTT. To avoid further congestion, the send window was set to minimum size on timeout.

Wrong ACK with TCP

A wrong ACK with TCP could only be the result of the wrong packets arriving. With the channel having a certain % drop rate, or delay rate skewing the packets out of order and leading to wrong ACK's, I kept the send window the same upon wrong ACK detection. With a small window and a high loss rate, timeouts would be more likely. In addition, the receiver buffered all new packets, so sending a larger window helped fill out that buffer.

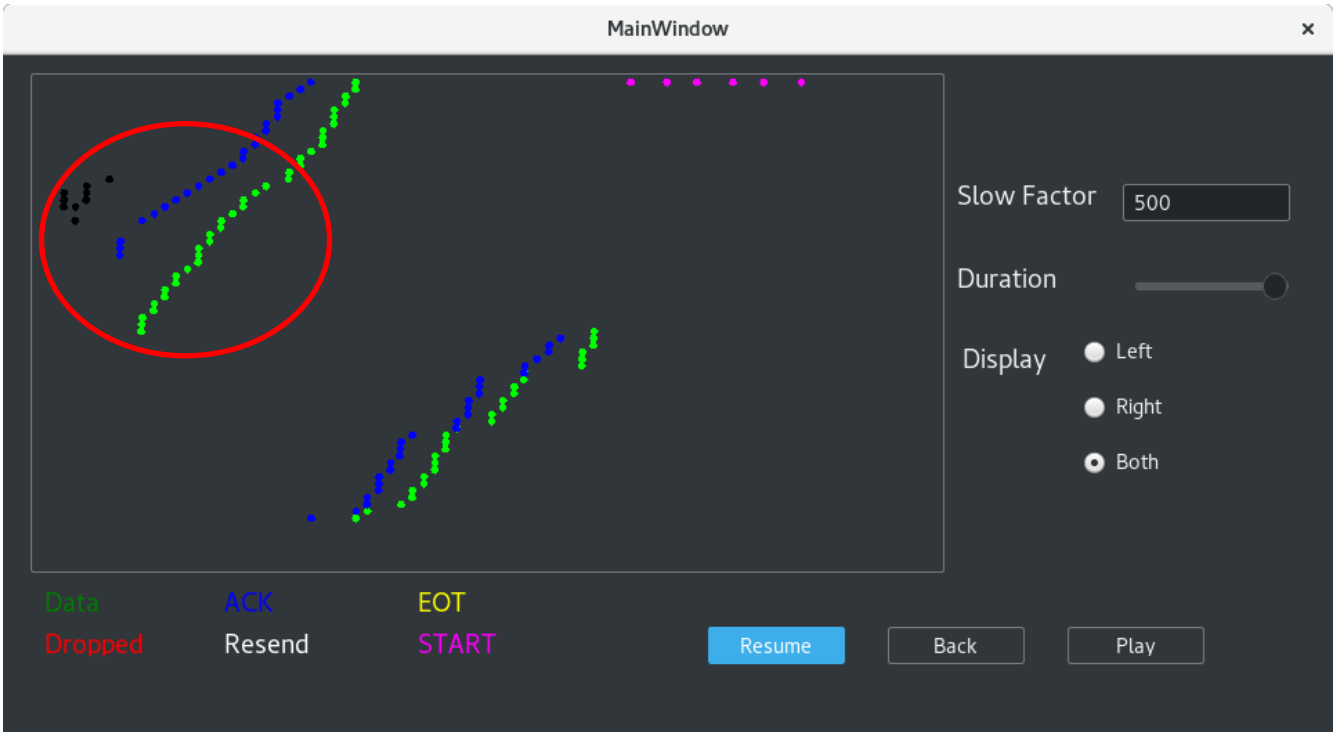
Overall Performance TCP

Overall, the performance was slow. On a no/low loss channel, timeouts killed performance. The receive side was not able to keep up to a large window of moderate sized packets – 24 window 10k packet, and TCP's congestion control kicked in, leading to timeouts.

Congestion with UDP

After switching to UDP, timeouts disappeared. Without the internal congestion control, packets were dropped. These drops caused the receiver to send back ACK messages, alerting the sender that it needed to resend. These responses happened before the timeout occurred, allowing resends to happen earlier. Because congestion now manifested through dropped packets, I was forced to add window resizing wrong ACK responses. It helped with congestion, but may not be ideal behaviour for dropped packets caused by the channel's drop chance.

Example of congestion avoidance:

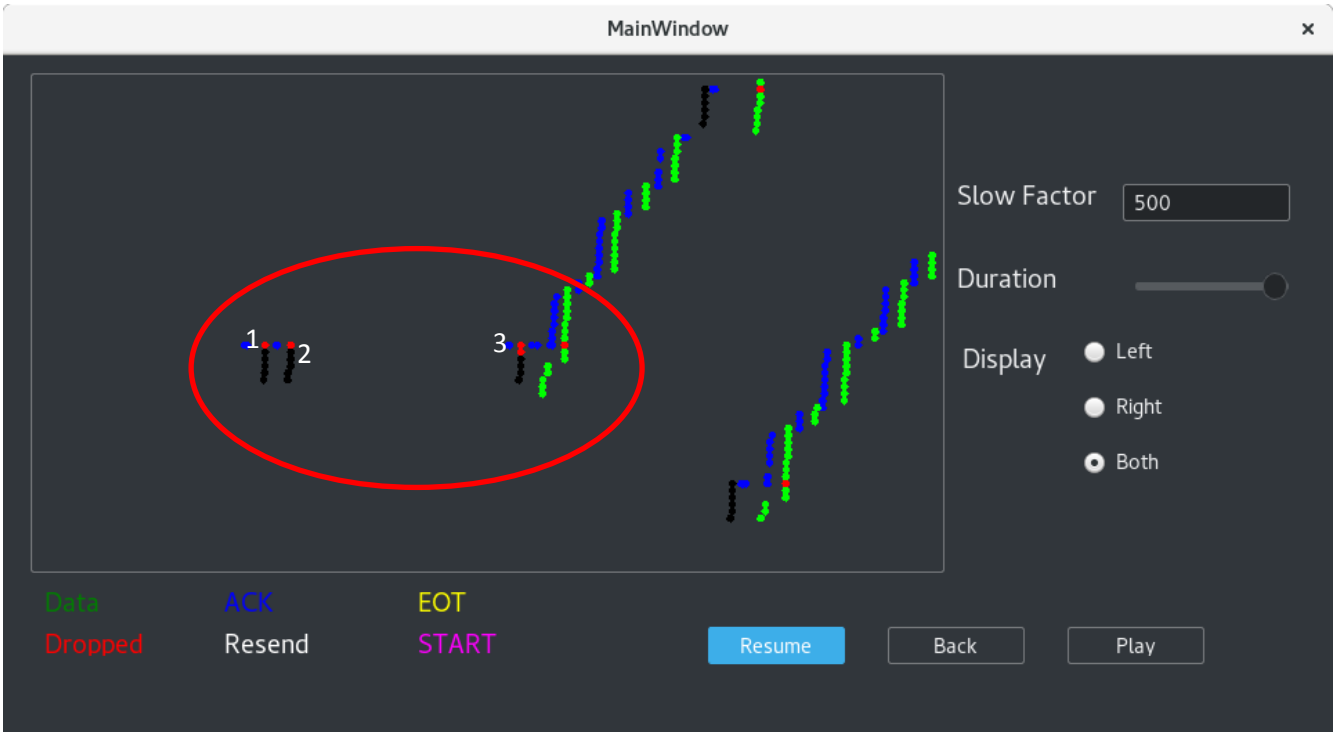


As the channel becomes congested due to sending a large buffer with a large window, the ACK slowly starts to be delayed. In the above, a packet was dropped due to the congestion causing a resend(in black). This resend reset the buffer to the minimum(6 for this transfer) to help avoid further congestion.

Wrong ACK with UDP

Wrong ACK's could now be caused by congestion or a drop by the channel. Because there was no way to differentiate, I was forced to cut the window on any wrong ACK.

Example of dropped packets(emulator, not congestion) causing a resend:



In the above picture, packets dropped by the emulator(red) caused a resend(3). The client was quick to react and send the resend because it received the required number of wrong ACK's. In the resend volley, the initial 2 packets were dropped, meaning that the remaining packets did not generate enough wrong ACK's to cause a resend. This caused the big delay in time as it led to a timeout. The resend because of timeout(2) also dropped a packet. The next packet sent in that volley was finally enough to trigger a resend, leading to the short time to resend the third volley(1).

Overall Performance UDP

The overall performance of UDP was much faster. The lack of congestion control was a benefit because it stopped or reduced timeouts. Dropped packets because of congestion alerted the sender before timeouts could happen, and started the resend process quicker than waiting on a timeout then resending.

Implementation

Writing

The writing is based around a timer and window. At the start of a transfer, an initial volley of packets is sent, and then the write loop starts. The write loop begins waiting on the timer of the first sent packet. During this time, the read loop is reading responses from the other client. These responses cause the timer of the packet currently being waited on to reset. This causes an end to the timer in the write loop. It returns with the response code – ACK, wrong ACK, timeout. These are handled individually:

ACKs

If an ACK is received, then 1 or more packets were successfully delivered. The window then fills the newly opened space with new packets, and then transmits them. The exception is if the window size has been reduced such that there are more outstanding packets than window size – in this case a resend is called. On an ACK, the window size is also incremented by 1.

Wrong ACKs

A wrong ACK is the result of an out of order ACK being received. This means either a packet was dropped or packets arrived out of order. In either case, a resend check is performed. This resend check will only return true – a resend, after 3 wrong ACK's were received. After returning true, it additionally sets a timer to ignore any future wrong ACK's for 1/2 RTT. This helps in the case of a large window, 40 or 60 packets, which had a packet drop early on from spamming 30 or 40 wrong ACK messages. Upon a retransmit caused by wrong ACK, the window is resized to minimum size. It is assumed that if a packet was dropped, there is a good chance that it could be due to congestion, therefore back off and send less data.(this effect can be seen in the above images)

Timeout

A timeout causes an immediate resend. The window is reduced to minimum size and a resend calls, sending a small number of packets.

Reading

The reader thread is always running. Upon received an SOT – a file may be opened and circular buffer thread started. The reader thread responds to all manner of header flags – ACK, EOT, SOT, and Data.

ACK

If an ACK is received, then the writer must be running. The current timer is reset with an ACK value, alerting the writer that it can send new data.

EOT

If an EOT is received, then the current send is done. Alert the writer thread that it needs to shut down and set final stats.

SOT

If SOT is received, then the other client is requesting a send. Respond to the SOT with the matching ACK number.

Data

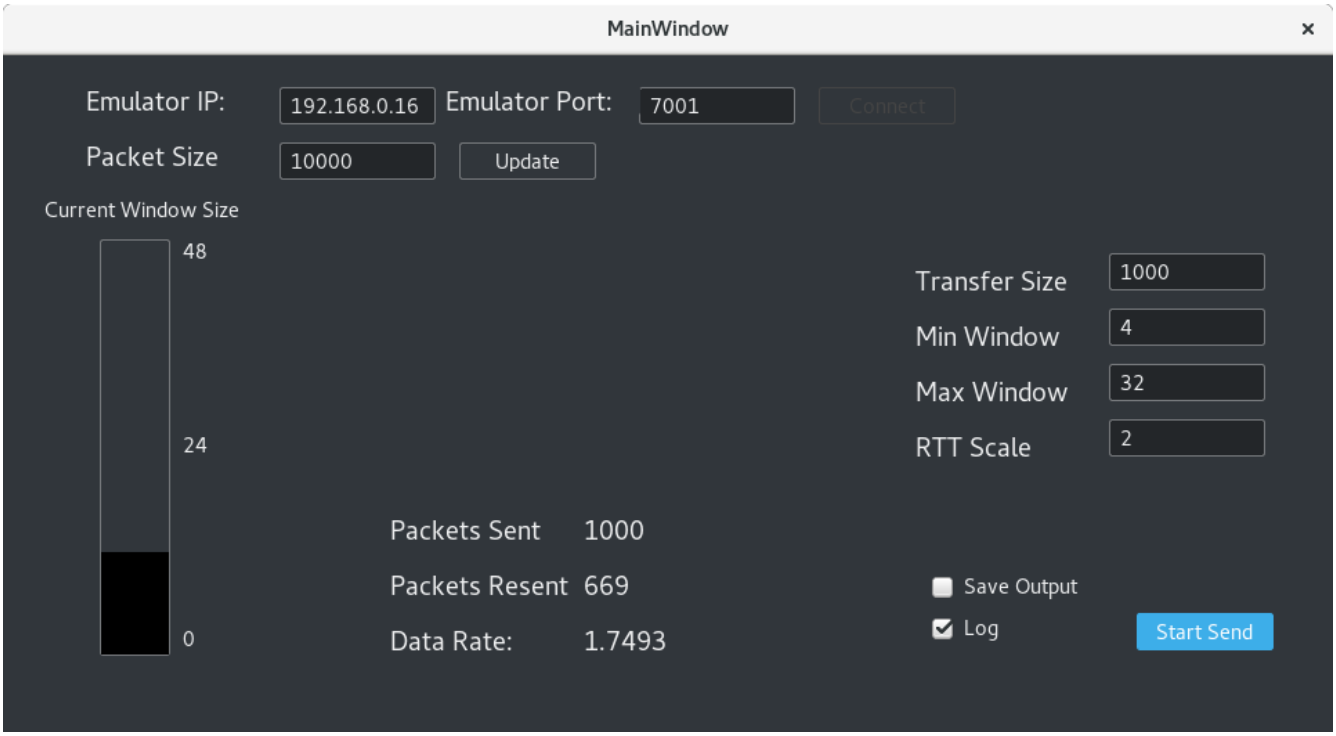
When data is received, the SYN number is pulled from the packet. This is checked against an array of all packets received. If the index of the current packet has not been flagged, the data is stored and the array flag is set. The data is also passed into a circular buffer for reading. After setting the received packet array, the array is scanned from the index of the current SYN to check for packets further in order that were already received. Upon finding an unset packet, it sets the currently expected ACK to that packet and sends the ACK back to the sender.

Full Duplex

The application was designed with the idea that full duplex might implemented later, if able, so I made a distinction in all receive methods – the responses for different header responses, between sending and not sending. In the end, few of these were needed. Upon receiving data, the SYN is checked and ACK may be updated. The ACK number is then passed to the function that normally handles ACK responses, and the rest of the program responds as it would normally. The newly sent or resent packets have the new ACK number attached to them and are sent off. The performance is significantly slower than a one way send, mostly as a result of increased timeouts and resends.

Sample Transfers:

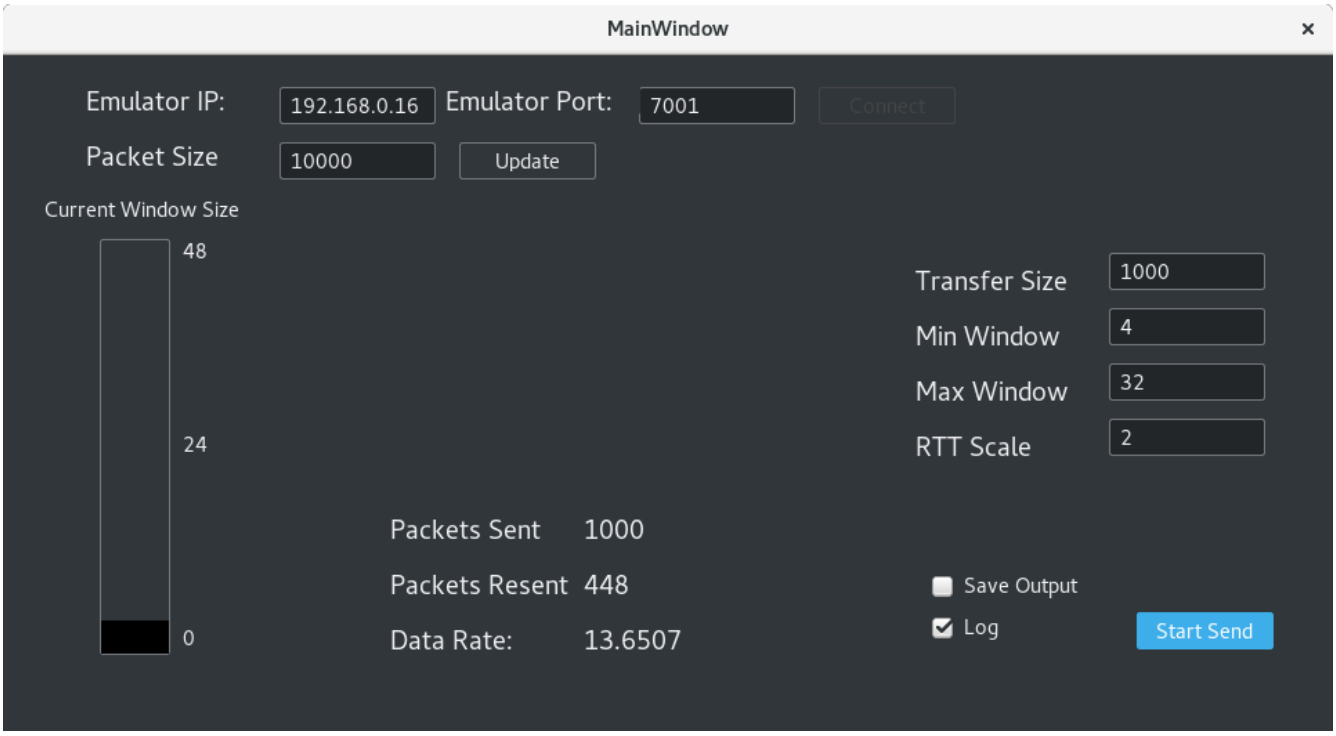
Transfer 1:



Transfer Parameters: 0% Drop Rate, 18-22ms delay.

The random delay with a 4ms variance caused the above transfer to resend a significant amount of packets, even though there was a 0% drop chance. With a window of up to 32, large volleys of packets would arrive out of order, causing multiple wrong ACK messages. In addition, with a 10KB packet size, congestion also played a role, leading to dropped packets due to congestion.

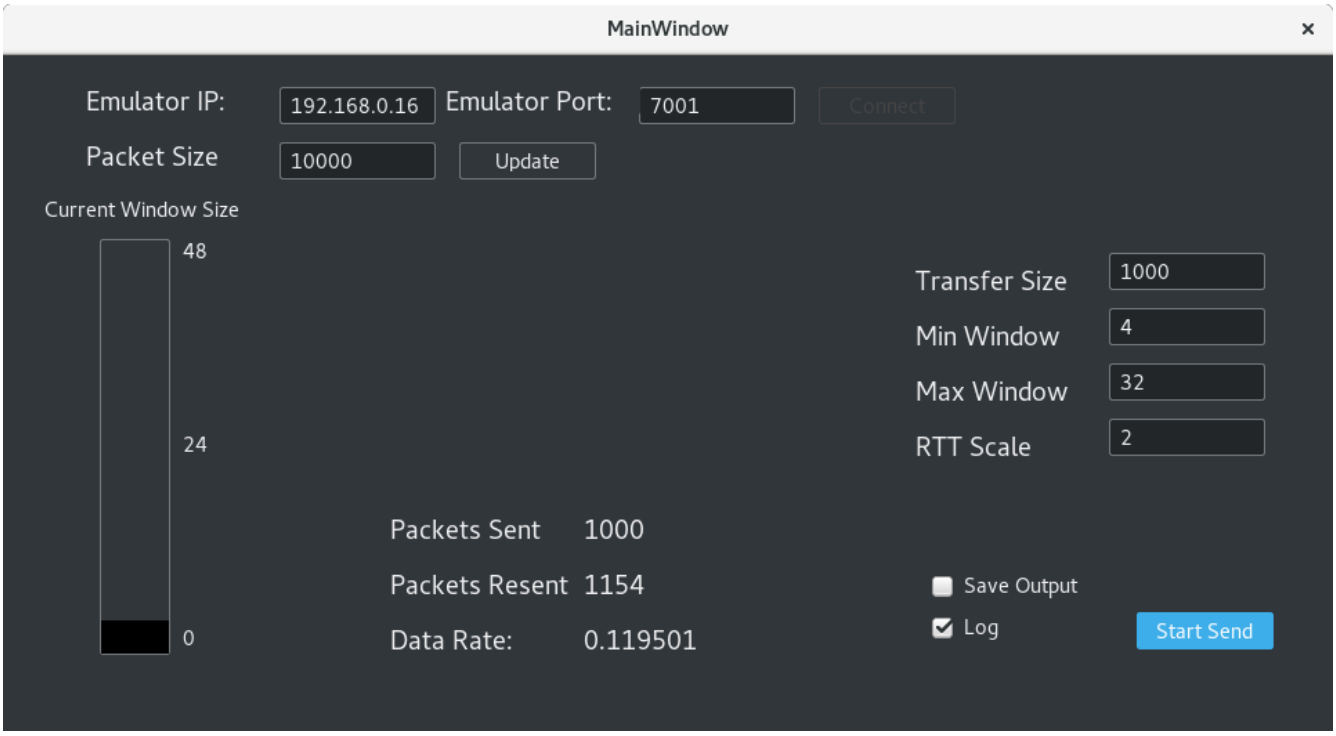
Transfer 2:



Transfer Parameters: 10% drop rate, no delay.

The above transfer had less resends than the previous one, despite having a drop rate of 10%. This was due to the lack of delay, allowing packets to arrive in order. Resends were then only caused by dropped packets by the emulator and dropped packets due to congestion(10KB window, 32 window size). The lack of delay allowed the transfer to complete at a decent rate, 13MB/s, compared to not even 2MB/s with delay in transfer 1 seen above.

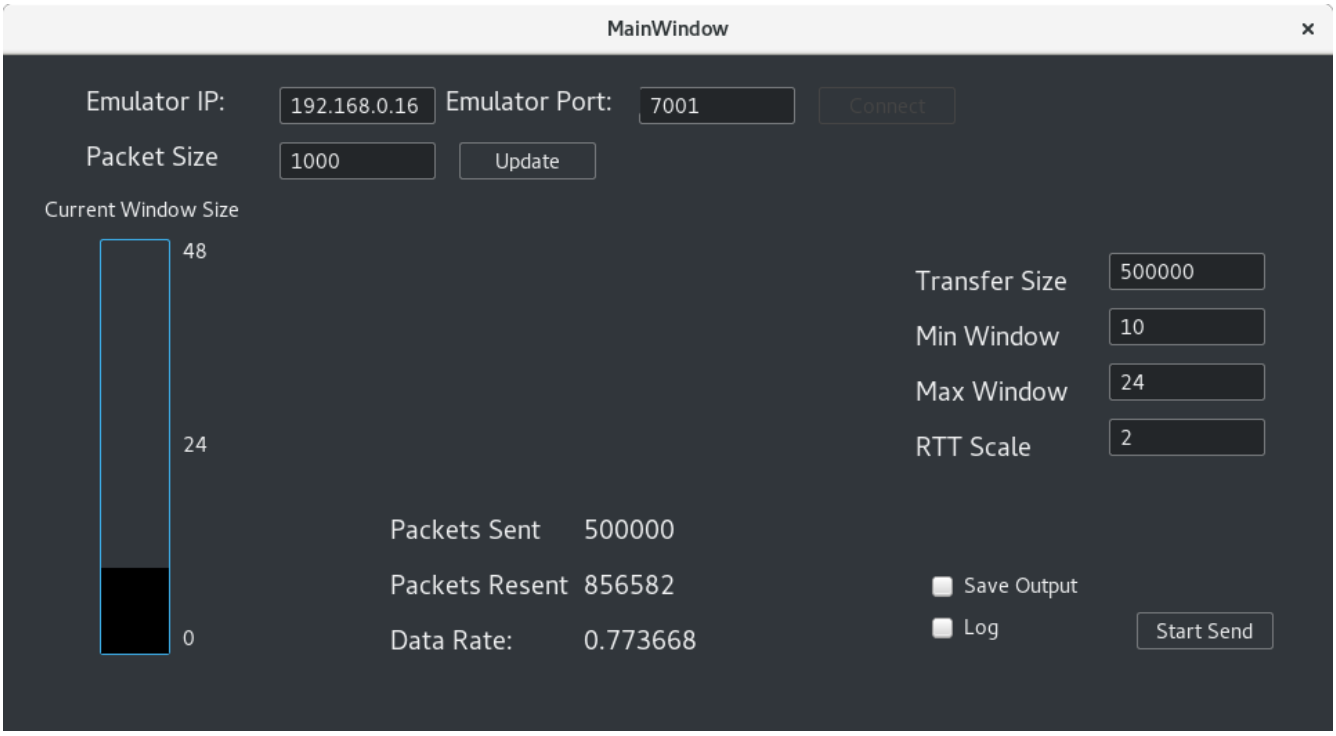
Transfer 3:



Transfer Parameters: 30% drop rate, 18-22ms delay.

This transfer was under extremely poor conditions. 30% drop chance on top of the large variance in delay lead to more resends than actual packets sent. The 18-22ms delay(both ways, so 36-44) also increased timeouts and slowed down the transfer to barely 100KB/s.

Transfer 4:



Transfer Parameters: 30% drop rate, no delay, 1000B packets, **FULL DUPLEX**.

This transfer was performed on two clients at once, with both sending the same 500,000 packets at the same time. The transfer featured a 30% drop rate at the emulator, which created the massive resend rate. Even with the high resend rate and lower packet size, it was still able to achieve a reasonable data rate(under such poor network conditions and with the increased traffic of full duplex communications)