Arpit Jasapara, Sahil Gandhi

UID: XXXXXXXXX, XXXXXXXXX

Project 2: Simple Window-based Reliable Data Transfer Report

Implementation Description:

We started by following the spec guidelines and first implemented simple one-packet UDP transmission. We did this by creating sockets on both the client and server side, accepting a port number, and sending a pre-determined file via a single packet to the client. We then implemented a client request mechanism where the client requests a file and the server processes this request and sends the appropriate file. Then, we divided the file into standard 1024 byte packets, and sent them one by one to the client, which compiled all the data received from the files into "received.data". At this point, since our file transmission was working correctly, we then decided to solve the problem of packet loss.

In order to combat packet loss, we realized we need to send some metadata about the packet along with the data to identify the packet and its contents. For this, we created a TCP_Packet class. This TCP_Packet class has a header of 9 bytes, with 2 bytes for sequence number, 2 bytes for acknowledgment number, 2 bytes for data length, and 3 bytes for flags (1 byte for SYN, ACK, FIN each). Therefore, each TCP_Packet is at max 1024 bytes long with a 9 byte header and 1015 bytes of data. We then initiate the TCP connection by sending a SYN from the client through the UDP connection, and upon receiving the SYN on the server side, we send back to the client a SYN-ACK. When the client receives this SYN-ACK, the client sends back an ACK and also attaches the filename of the requested file. At this point, we send the first 5 packets of the file out since that is the starting window size, while keeping each packet in a TCP_Packet array so we know which files we have sent out. On the client side, upon receiving this packet, the client sends back an ACK. The client maintains a rolling window of size 25 (30 – 5) to determine when a file segment that is outside its current scope (can only differ by a max of 5 since the server's window size is 5) and as such rotate the window while writing to a file buffer. When the server receives this ACK, it removes that packet if there are no unACKed packets before it in the array. It also removes any other consecutive ACKed packets, basically shifting the window for each consecutive ACK. Then the server sends additional packets as the window permits. Upon sending the final packet and receiving the ACK for that packet, the server sends a FIN, to which the client sends an ACK. After the client sends the ACK, it also sends a FIN, to which the server responds with an ACK.

At this point, we have set up a basic TCP-like handshake and connection, so we now introduce packet timeouts to combat packet loss. We use the clock_gettime to determine if packet loss has occurred for each packet, and upon timeout we retransmit that unACKed packet. We do this for each TCP_Packet, including the SYN, SYN-ACK, ACK, FIN, and regular data packets, so this ensures that each packet is ACKed or retransmitted until it is. On the client side, upon receiving any data packet, it first checks if it's a duplicate packet (due to retransmission). If it is a duplicate, you discard the packet (but still send over an ACK), and if it is not, you check if the

packet is the next in sequence for the data file. If it is, simply add the packet to the data file, and otherwise buffer the packet in the receiver window TCP_Packet array. Upon receiving another data packet, the client will check the buffer to determine if any out-of-sequence packets are now the next in sequence to add to the file. Thus, at this point, we have implemented the main portion of the project and ensured that reliable data transfer will occur.

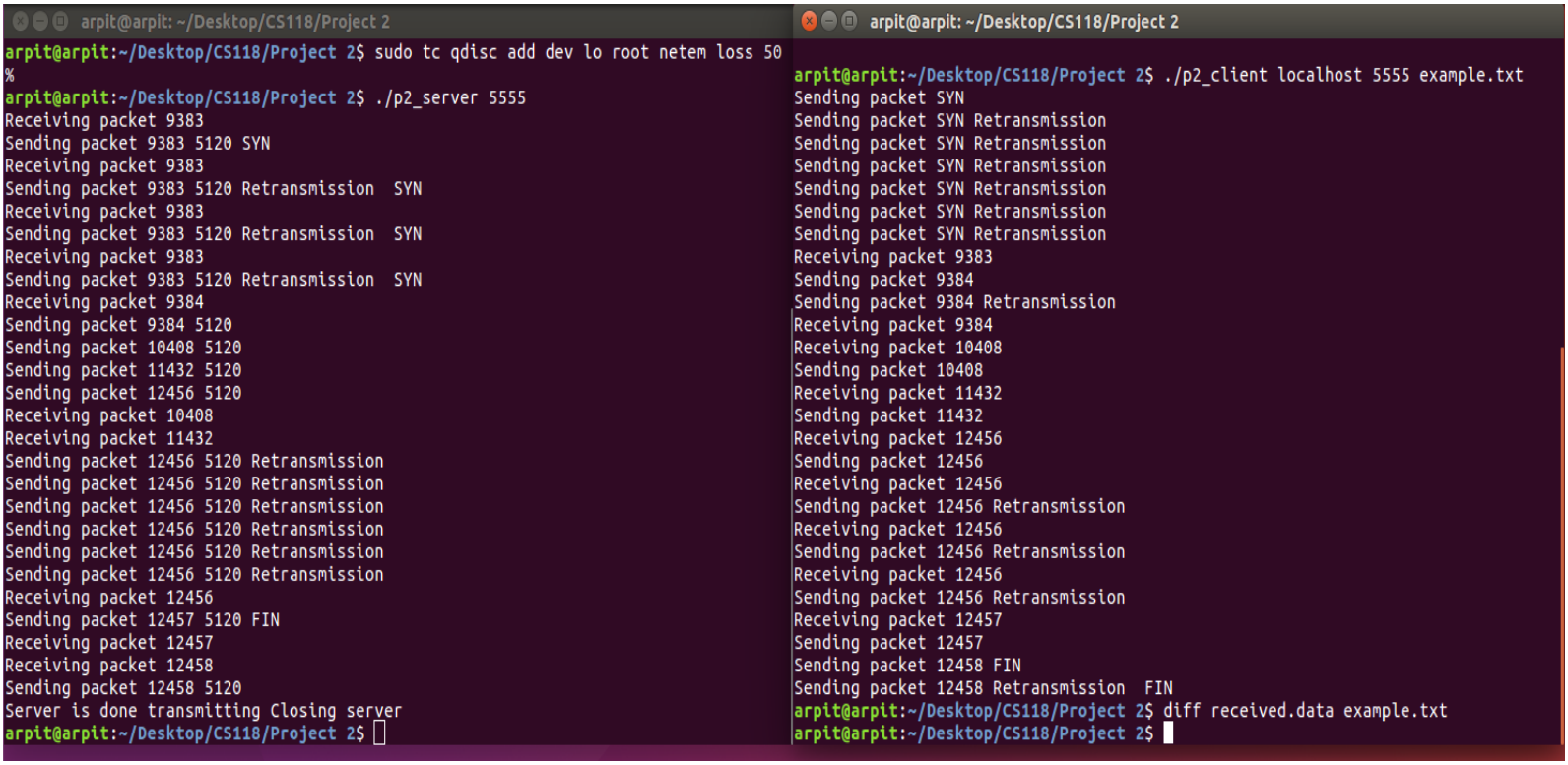

Figure 1. Example run of client/server programs with network loss enabled.

At this point, we used our current implementation and expanded on it to add congestion control for the extra credit. The congestion window size increases in accordance with slow start until packet loss occurs, at which point the congestion window is set to 1 MSS (1024 bytes). If the threshold is reached by the congestion window, then we go into congestion avoidance and calculate when to increase the congestion window by the appropriate amount. For this, we also changed the sequence and ack number size to 4 bytes each (up to 2^32) since the max sequence number is now 2^32 bytes.

Thus, we have successfully designed and implemented a reliable data transfer protocol analogous to TCP over an unreliable UDP connection.

Difficulties we faced and how we solved them:

The biggest difficulties we faced involved thinking through retransmission and the extra credit congestion control. For retransmission, we had to ensure the packet was sent out when the timeout occurred, but only if it is unACKed. However, the ACK could be lost, and we had to figure out how to retransmit the ACK so that the server knows it was received. We implemented this by sending the ACK upon receiving any packet, including duplicate ones, so the server knows to

disregard duplicate ACKs, but it ensures that each packet is ACKed. Similarly, we had to ensure that duplicate and out-of-order packets due to retransmission and packet loss did not affect the integrity of the requested file. We implemented this solution using a receiver buffer and the sequence numbers to ensure the right order. We had some issues for the regular credit where the sequence numbers could be out of order and could also wrap around since it wrapped back to 0 after reaching 30720, so we solved this by using a rotating window with just the right size so it is only rotated when segments out of the largest possible range are received. Lastly, for the congestion control window, we were having difficulty adjusting the window on both sides and ensuring that the right algorithms were followed. We implemented this by adjusting the server-side window since the client only transmits back the ACKs so there will never be more ACKs than the window size. Then, we used the packet timers as a tool to detect congestion, and accordingly adjusted the window size and the size increase/decrease algorithm (AIMD, resetting cwnd/ssthresh according to fast recovery, congestion avoidance, etc).

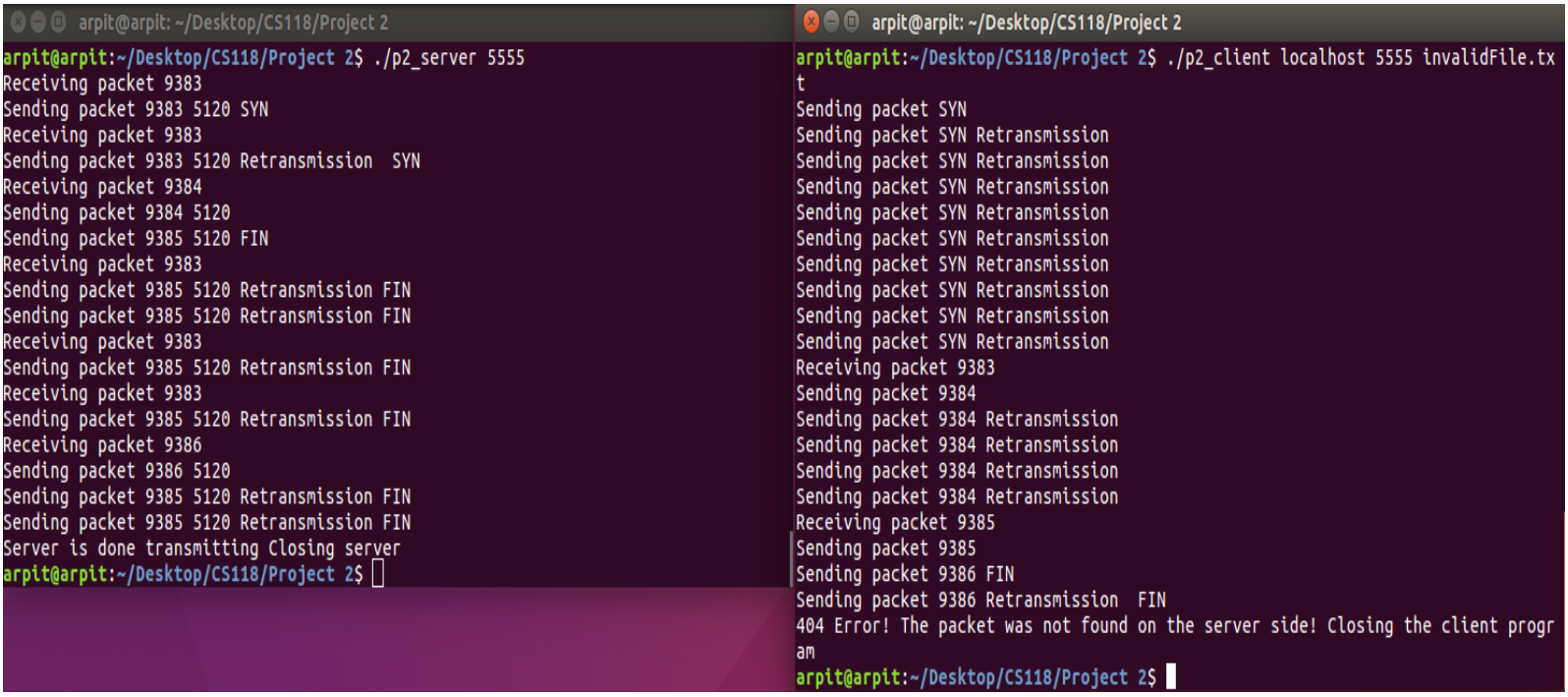

Figure 2. Client/server program communication with network loss on an invalid file, causing a 404 error.