**CS 118: Computer Network Fundamentals**

**Project 2**

**Report**

**Abhijoy Saha (Seas: abhijoy) – 004252758**

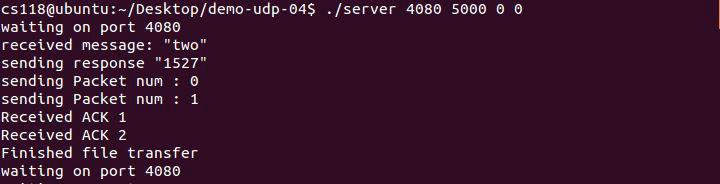
**&**

**Kenta Onimura (Seas: kenta) – 804296452**

**Implementation**

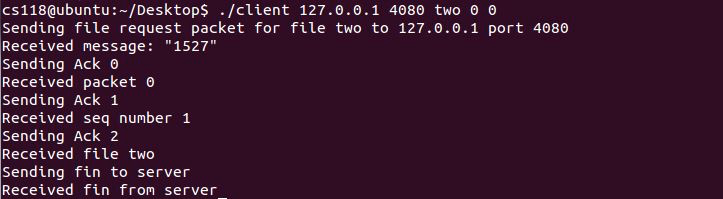
We implemented a reliable data transfer protocol on top of UDP by implementing sequence numbers, handshake and FIN mechanisms. We started with the skeleton code for UDP provided by Rutgers University at <https://www.cs.rutgers.edu/~pxk/417/notes/sockets/demo-udp-04.html>. This skeleton code only sets up the sockets on both the client and server and sends a few test messages between the client and server.

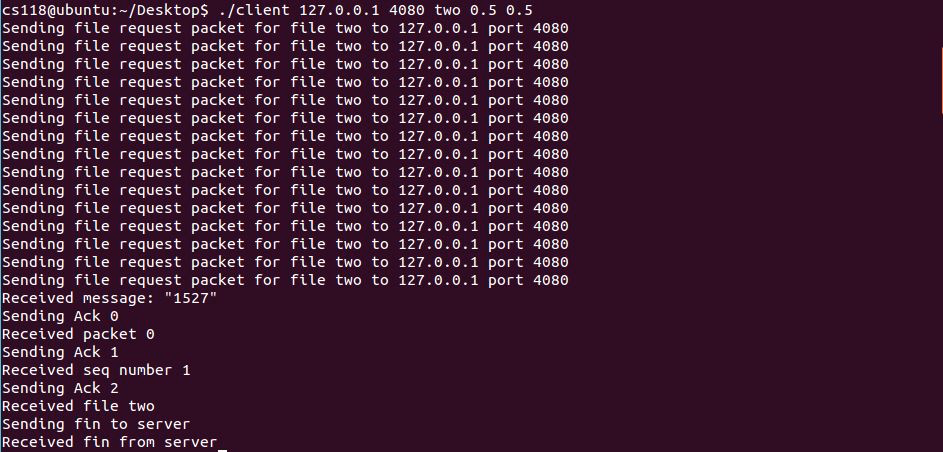
There are two classes we implemented for the headers, Packet and PacketStream. Packet contains a data array whose size is denoted by the macro PAYLOAD\_SIZE. PacketStream contains an array of Packets and provides getter and setter functions. When the server prepares the file data for transfer, PacketStream’s array of Packets is initialized with a sufficient number of packets to contain the entire file. Each of these packets is initialized with a sequence number for the client to ensure that it is not missing any Packets. The Packet also contains boolean fields to signal whether the packet is corrupted or lost at the destination. These values are set before each packet transmission and allow us to test for reliable data transfer and emulate packet loss and corruption.

We implemented a selective-repeat protocol with these classes. The server executable takes in a congestion window to set the window size. The server starts the file transfer by sending back the file size and all the initial packets in the window. The server keeps track of all its sent packets and respective timestamp in a linked list, and all of the received ACKs in a binary search tree (a set in C++). Every time the server receives an ACK, we update these two data structures to remove Packets that have been ACKed. If the number of packets waiting for ACKs is smaller than the congestion window and there are more packets available to send, the server will send packets until the window is full or there no more packets to send. Once the server receives a FIN packet from the server, it will send a FIN packet and will start the process again for a new file.

1: Server output for small file with two packets

On the client side, we also create a PacketStream to store all of the received packets. We also internally keep track of the received packets in a boolean array, whose size we receive before the file data. As we receive the packets with their sequence number, we mark the boolean array until the entire array reads true, at which point we have received the entire file and write the file to disk from PacketStream. We then send FIN packets (with a specified sequence number) until the server sends a FIN packet back, at which point we exit.



2: Client output for small file with two packets. Handshake and FIN.

3: Client output with packets from server being lost and corrupted. The handshake in the beginning does not terminate until it receives the file size.

**Difficulties**

We faced tremendous difficulty when splitting the file into a stream of packets. We initially did so using strcpy() and then after debugging realized that this did not allow us to copy the embedded null characters and binary information into our output file. This resulted in our file transfer only working for text files and not binary files. We solved this issue by using memcpy() which instantly solved out problem.

We faced difficulty in specifying when the client initiated and finished the handshake and the final fin. To specify this we made sure that sequence number is -1 for packets involved in the handshake except the packet for ACK 0 which tells the server to start sending the file packets. For fin, we set the sequence number to -2 allowing the fin packets to be recognized on both sides.

We at first considered using forking to create a timer for receiving a packet from the socket and timing out individual packets. We tried implementing this but it turned out to be overly complicated and we decided to move on from this idea. After hours and hours of research, we discovered the setsockopt() function and used its SO\_RCVTIMEO option to set a timer when receiving packets from a socket. This in conjunction with a while loop allowed us to keep receiving packets and at the same time send appropriate packets on time out. For time outs for individual packets, we created a new variable which initialized to timestamp when packet is transmitted and this timestamp is compared to the current time when the packet is received to determine time out.