CS 118, Winter 2017
Professor Songwu Lu
Project 2

Lauren Yeung, 104456482
Kelvin Zhang, 304569586

Implementation Description

Our implementation of Selective Repeat Protocol in a window based reliable data transfer project consisted of three primary programs: server.c, client.c, and packethandler.h. This was built on top of UDP socket programming, which did not guarantee reliable data transfer, thus leading to our implementation of buffers, timers, and other methods of ensuring data delivery.

The server is responsible for creating and binding a socket, and upon client request, sends a file in packets to the client. Every packet has a header with identifying information such as the sequence number to identify the packet that was just sent. Upon sending out the packet, a timer with 500 ms timeout is started while the server waits for an ACK to the corresponding packet.

The client also creates and binds to a socket, and establishes a connection with the server with a SYN. The client continues to wait and send SYNs until the server responds. For every packet of data sent to the client, the client sends an ACK, equal to the sequence number plus length, back to the client.

The packethandler header file is included in both programs, defining common macros such as maximum packet size and having a struct called Packet. Packet holds the header, including the type of Packet, ACK number,sequence number, timer, length, and finally a buffer of the actual file data.

The connection is initiated with a three way handshake. The client sends the SYN and starts the timer; the server sends a SYN-ACK. The client has a timer when the packet is sent, which resends the SYN at 500 ms, then restarts the clock and continues to transmit if there is packet loss. Once the SYN-ACK is received, the client sends a request packet with the desired filename to the server.

The Selective Repeat Protocol is implemented with a window size of 5 for each the client and the server. The client starts expecting sequence number 0, and has a integer for the maximum window size of 1024*5; the window shifts up by the size of each packet received. If the packet is of the expected sequence number, the packet is written to file, ACKed to the server, and the window is shifted up by the size of a packet. If the packet is within the window size but larger than the expected sequence number, the packet is either stored into an array buffer or discarded if the packet has been seen before in the window size, and an ACK will be sent back. If a consecutive number of packets have been buffered, and an expected packet arrives, the entire window is shifted up. The previous window of packet information are also stored so that ACKs can be sent to the server if prior ACK transmissions were lost.

For the server’s window, a timer is set for every packet as the first five packets are sent out. The window only shifts up if the lowest expected sequence number receives an ACK. For packets within the window range but higher than the lowest sequence number, the ACK can be stored until the lowest is ACKed, and the bottom of the window can be increased to the highest consecutive ACKed number. In case of packet loss, each timer that timed out would retransmit the corresponding packet. Even if a server’s packet reaches the client, the ACKs from the client can also be lost, so that the window cannot be shifted up. In that case, the timer for the particular packet would time out again and continue to resend.

Difficulties and Solutions

One of the most of the difficult parts was incrementally developing and debugging the project, since there were various areas that lead to segmentation faults. After the first project, we gained some experience with socket programming and using the functions “sendto” and “recvfrom.” However, this project dealt with more transmissions, time-outs, and the selective repeat protocol.

Our first roadblock involved refactoring code when we realized that using a wrapper for the packet handling made the process even more difficult. The first phase of the project, where the server sent packets and the client sent ACKs back worked. However, as we moved on to phase two for splitting a large file, the memory allocation, zeroing of buffers, and pointers made it inconvenient.to implement the handshake and multiple packet transmits. We opted to use only the header to store macros and the struct declaration.

For the struct Packet, we had to determine how we wanted to store distinguishing information. We chose to put the ACK and sequence number in there, as well as having types that could describe the packet as a SYN or ACK or FIN, highlighting whether other fields were pertinent. We also chose to associate a timer, so that once a packet was stored at an end party, the timer could be set.

For the client’s window, we used a set range of integers to keep track of the window for sequence numbers. For any out of order packets that were within the window, we had an array that held pointers to the packets, in order to buffer data that could be written to file after the lowest numbered packets arrived. The server’s window was more difficult to implement since timers had to be actively used for each packet. Not only did a particular sequence number have to be sent out, but it had to be individually timed until the following ACK returned. Server.c utilized a vector for the convenience of adding and dropping packets that were in the window.