Akikazu Miyazawa 504827264

Nicholas Zimmerman 804809254

Project 2 Report

Implementation Description

Server

Our server program begins by creating a socket and then binding it to the port given by the parameters. It then begins a loop and waits for a client to contact it. When it receives a packet from a client, it updates its global variable ‘filenum’ so it can create a new file for this connection. When the client eventually terminates the connection, the server returns to this loop with an updated filenum. The server then opens the file associated with the client number and creates a beginning sequence number for itself. It creates a packet to send to the client with the appropriate ack_num based on the client’s seq_num. It sets the ack and syn flags to 1 and then sends the packet to complete the connection establishment process.

While the client sends packets that contain data, the server checks the packet’s ack_num and seq_num to make sure packet loss has not occured. If something does not match, the server must retransmit the previous acknowledgment packet. If no packet loss has been detected, the server gets the packet, updates its internal counters of what sequence number to expect next, and moves the data from the client into the appropriate file. The server then constructs a response packet with an ack_num equal to the seq_num of the packet plus the number of bytes in the data array of the packet. It then sends this packet and awaits more.

When the server receives a packet that has the fin flag set, it knows to enter the end connection procedure. The server first constructs an acknowledgment packet for the fin packet

and sends it. It then constructs another packet with the fin flag set and sets this as well. When the server receives a packet from the client acknowledging the fin packet, it then terminates the connection.

Client

The client program begins by constructing a socket. It then creates a syn packet with a randomized initial seq_num, and sends it through the socket to the given port number and hostname. It then waits for a response from the server and checks the packet to ensure it has the proper ack_num and flags to begin.

Once the connection has been established, the client opens the given file and begins to read the file into a buffer 512 bytes at a time. It then constructs a packet with the buffer as the payload and fills in the correct seq_num and ack_num. It does this by keeping internal records of these values. It also uses these values to check response packets from the server to ensure no packets have been lost. If there is more data to read from the file, it repeats the process until the entire file has been sent.

When the entire file has been sent, the client constructs a FIN packet to eventually close the connection with the server. It first sends the initial FIN packet and waits for an acknowledgment packet from the server. After receiving this, the client waits for a corresponding FIN packet from the server while ignoring any other incoming packets. When it eventually receives this packet, it sends an acknowledgment packet with the appropriate seq_num and ack_num. It repeats this process of responding to FIN packets for two seconds until it finally closes the connection.

Difficulties

The first problem we ran into was successfully transmitting the data array between the client and server. We kept getting segmentation faults when accessing the data array on the server side even though we checked its value on the client side and found it to be correct. Through use of gdb's ability to look at the value of local variables, we realized that on the server side, the array pointed to an address value that was inaccessible. In order to fix this problem, we changed the structure of the packet by having the array equal to char data[payload] instead of char* data. This fixed the problem and we were able to access the array from the server side.

Another problem that we encountered was how to properly implement timeouts. At first we attempted to use clock() and measure the difference between clock() values taken at different points in the program. However, we kept running into infinite loops and we realized, once again through the use of gdb, that clock() kept returning 0 no matter when it was called. We tried different fixes for this problem but made little progress. We eventually switched from clock() to the struct timeval and the function gettimeofday(). We called this function on two different timeval structs and then subtracted the .tv_sec member variables. This seemed to work and we never figured out what was wrong with clock().