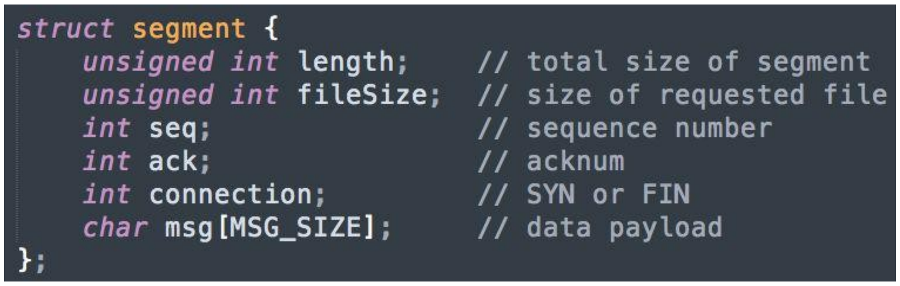
Implementation Description

Our implementation uses a 20 byte header as shown in **Figure 1**.We store the data payload in a character array. Since the maximum packet size is 1024 bytes, our data payload is a maximum of 1004 bytes. We included a 'length' header that specifies the total size of the TCP segment, a 'fileSize' header that indicates the total size of the file to be transmitted, a 'seq' header to specify the current sequence number, an 'ack' header to specify the current ACK number, and a 'connection' header to specify if the current packet is a SYN or FIN packet.



```
struct segment {
    unsigned int length;     // total size of segment
    unsigned int fileSize;   // size of requested file
    int seq;                 // sequence number
    int ack;                 // acknum
    int connection;          // SYN or FIN
    char msg[MSG_SIZE];      // data payload
};
```

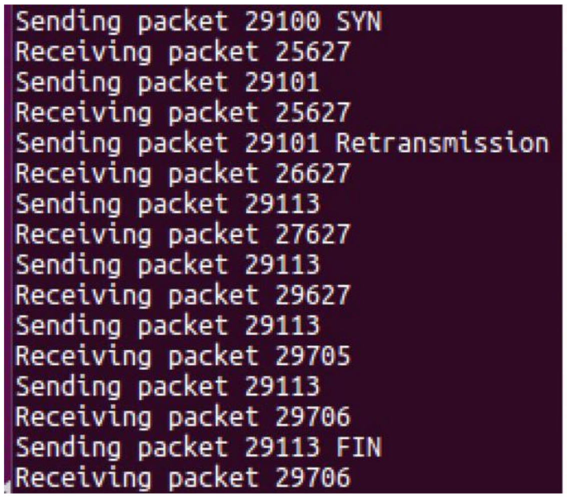
Figure 1: Our TCP header fields

The format of the messages in the data payload is very simple. The only time that the client sends data is when it sends the name of the requested file, and it simply fills the data portion with the characters in the filename. The messages that the server sends in the data portion are simply the bytes contained in the requested file. No additional information is included in the payload.

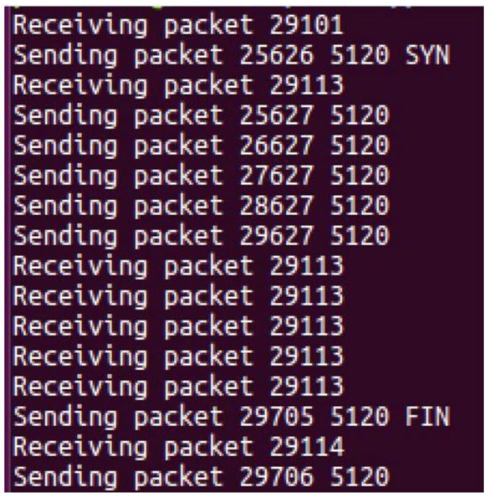
Our protocol begins with the client sending a connection request to the server with the 'connection' header field in the packet set to SYN. The server responds with a SYN-ACK packet: 'connection' is set to SYN and 'ack' is set to the client's initial sequence number + 1. Then, the client piggybacks an ACK for the server's SYN (server's sequence number + 1) with the name of the requested file contained in the data portion of the packet. The server then runs in a loop as it continually reads in data from the file and sends up to 5 packets (since cwnd is 5120) to the client if no ACKS are received. Our reliable data transfer protocol uses cumulative acknowledgments and a sliding window design, so any valid ACK for a sent packet will increase the sender's base window up to the received ACK. If no ACK arrives for the base packet in the sliding window after 500 ms, retransmission timeout occurs and the base packet is retransmitted to the client. When there is no more data to read from the file and the server has received an ACK for its last data packet, it breaks out of its loop and initiates the FIN procedure with the client.

The client also runs in a loop as it waits for the server to send it the requested file's data. The client buffers out of order data for packets that have a sequence number that is no greater than 4 maximum packet lengths larger than the next expected sequence number, so the receiver window has a total size of 5 packets. If the incoming packet contains the next expected sequence number, the client writes the payload data to the received.data file and increments the expected ack number based on the amount of data received. If any out of order packets have been buffered, the client checks if any of these packets have a sequence number that matches the expected ack number, writes its payload data to the file, and increments the expected ack by the length of data in this manner until either no buffered packets are remaining or their sequence numbers do not match the next expected ack number. The client then sends the expected ack to the server in an ACK packet. When the client receives a FIN from the server, it sends a FIN-ACK and waits for the final ACK from the server. When the ACK is received or multiple RTOs occur, the client exits.

Figure 2shows the messages that are printed to the Terminal by the client (left) and server (right) when the network is emulated to have 30% packet loss, 600ms delay, and 25% reordering.



```
Sending packet 29100 SYN
Receiving packet 25627
Sending packet 29101
Receiving packet 25627
Sending packet 29101 Retransmission
Receiving packet 26627
Sending packet 29113
Receiving packet 27627
Sending packet 29113
Receiving packet 29627
Sending packet 29113
Receiving packet 29705
Sending packet 29113
Receiving packet 29706
Sending packet 29113 FIN
Receiving packet 29706
```



```
Receiving packet 29101
Sending packet 25626 5120 SYN
Receiving packet 29113
Sending packet 25627 5120
Sending packet 26627 5120
Sending packet 27627 5120
Sending packet 28627 5120
Sending packet 29627 5120
Receiving packet 29113
Receiving packet 29113
Receiving packet 29113
Receiving packet 29113
Receiving packet 29113
Sending packet 29705 5120 FIN
Receiving packet 29114
Sending packet 29706 5120
```

Figure 2: Client (left) and server (right) messages to Terminal upon sending/receiving packets

<u>Difficulties Encountered and Solutions</u>

One of the biggest difficulties we faced was figuring out how to wrap the sequence and ACK numbers around the maximum sequence number value. Our initial implementation caused all sequence numbers to be buffered by the client even if they were not within the client's receive window. We fixed this issue by creating a function called Wrap_inc that takes an integer value and an amount to increment it by and returns the result modulo (MAX_SEQ+1). We use this function to increment all sequence numbers and acknowledgment numbers. After implementing this change, our sequence and acknowledgment numbers wrapped around the maximum sequence number correctly and our client correctly buffered out of order packets.

Manual for Compiling/Running Server and Client

1. Preparation: All files in the submission tarball (project2.tar) should be inside the current directory. If not, place the tarball in the current directory and then extract the files:
   ```
   $ tar -xzvf project2.tar
   ```
2. Compilation: Enter the 'make' command into a Terminal to compile all 4 source files (this will create 4 executable files: 'server', 'client', 'serverCC', 'clientCC'):
   ```
   $ make
   ```
3. Running the Server: Launch the server by executing the 'server' program
   ```
   $ ./server [port]
   ```
4. Running the Client: Launch the client by executing the 'client' program (hostname must be '**localhost**' or '**127.0.0.1**' without the single quotes):
   ```
   $ ./client [hostname] [port] [filename]
   ```
5. Running the Server/Client in Congestion Control: Repeat steps 3 and 4 above, except replace 'server' with 'serverCC' and 'client' with 'clientCC':
   ```
   $ ./serverCC [port]
   $ ./clientCC [hostname] [port] [filename]
   ```