Student 1:

Destin Raymundo, 904852907

Student 2:

Ethan Ward, 304635336

High-Level Project Description

Using C, we implemented a reliable transfer protocol built on UDP. Our implementation uses the Selective Repeat window-based protocol to ensure reliable data transfer. We wrote two programs, `client` and `server`. The client program acts like a client application requesting a file from the server. The server program opens the file in its local directory and attempts to send it back in packets. While the two programs communicate with each other through unreliable UDP, the Selective Repeat protocol ensures reliability.

Server Implementation

The implementation of the server can be broken down into a series of steps:

1) First, the server waits for the client to establish a connection and send a packet with a SYN flag.

2) Once it receives such a packet, it sends a SYN ACK flag over the socket.

3) Then, it waits for the client to send a packet with the filename it requests.

4) Once the server gets the filename, it looks for that file in its local directory. If it doesn't find it, it sends back a 404 message and exits. If it does find it, it will open a file descriptor for that file, read and send back the file at a packet granularity.

5) The server has a five slot queue data structure which acts as sending window in the SR protocol. Every time a packet is generated and sent, it is also pushed to queue. The server does this as long as EOF hasnt been read and as long as the window isn't full. (Note that the queue is implemented as an array of type `struct pwrapper`, which is a custom class that wraps the packet and stores metadata about the packet, like the epoch time it was generated and a flag to mark whether it's been received by the client)

6) Before the server attempts to receive any new packets, it checks the window to find the oldest packet whose age is over 500ms, the timeout period. (`currentTime - packet timestamp`) If it finds any such packet, it resends that packet.

7) Afterwards, the server attempts to receive a packet from the client. Once a packet is received, the server follows a mark-and-sweep approach to update the window. Within the window, the server looks for packets that has the same sequence number of the ACK that was just received from the client. Any packet that satisfies that requirement is marked as `complete`. In the sweeping phase, the server starts from the beginning of the queue and pops off completed packets until it runs into an incomplete packet.

8) Steps 5 to 7 are repeated until the entire final is read and transmitted.

Client Implementation

The steps the client follows are:

1) The client attempts to initiate a connection by sending a packet with the SYN flag.

2) When a SYN-ACK packet is received, the client sends the filename argument as the message of the next packet. The packet is sent repeatedly until it is acknowledged with the first data packet

3) The client now continues receiving packets and sending ACKs for them, storing received packets in a `struct recv_buffer` which has 5 slots for packets along with some context information including the number of bytes written to the received.data file. Each time a packet is received which has a sequence number equal to the next expected byte, the client writes its data to the file then iterates through the buffer writing data until the next expected byte cannot be found. The next slot to receive data into is chosen by finding the first packet in the buffer whose sequence number is lower than the number of bytes written, so writing a packet's data automatically "frees" its buffer space. If a packet is received that contains unwritten data but is a duplicate of an already buffered packet, its sequence number is manually set to -1 so its buffer space can be reused.

4) When a packet with FIN is received, the client calculates the final byte number and continues receiving packets as described above until the number of bytes written is equal to the final byte.

Issues and Roadblocks

- The project spec required that ACK and SEQ numbers are defined at a byte granularity rather than a packet granularity. This way of thinking is different from what we practiced in class, so we encountered some slowdowns while developing.

How to Run

```
make
./server <portno>
./client <server hostname> <server portnumber> <filename>
```