Alex Fountain

Josh Crum

Tristan

**Project 1 Documentation**

**Client Side:**

The client side connects to the server that is already running on a specified port. This port entered by the user. The IP address is also requested by the program for the user to input. If the server is not running the client will not connect and will close the program. If the server is running then the client will connect and be prompted for a file name that they would like to transfer. The client then sends the file name to the server. If the file does not exist the then the client will close the application. If the program does exist the client will receive the file from the server. The file transfer will be based on a sliding window packet transfer. The window size is set to five. The way we are accomplishing the sliding window is by using an array. When the packet is received, the first character of the packet is stripped off. This character is a number put on by the server. The program then places a 1 in the array for what position the packet was in, writes it to the user’s file, and then checks if the packet was the lowest in the window. If it was then it sends a confirmation to the server and increments the window by 1 and checks again to see if the next packet has arrived yet. If the packet has arrived it will again increment and continue the loop till a packet hasn’t been received. After a five second period when a packet is not received, the program prompts the server to resend the missing packet. This continues till the buffer reader is null. The packets received are checked by their sequence number to ensure that they are in order, if they are not they are placed in order and then written to the file. Corruption is handled as a dropped packet, the packets left variable is not decremented on corruption, however this does not work well and is sometimes uncaught, resulting in a corrupted file.

**Server Side:**

The server side of the program runs and waits for a client to connect. Every five seconds it times out and waits for another connection. Once the client connects to the server, the server receives a file name. The server then checks to see if the file requested exists. If the file does not exist it prints out that the file does not exist. It then waits for another client connection. If the file does exist it begins transferring the file. It reads the information into a buffered reader and then sends the number of packets in the packet window. It does not clear the buffered readers that are storing the packets in memory until a confirmation response from the client is received. It then stores a 1 if a confirmation is received in the confirmedTrack array. The lower and upper of the window is then incremented by 1. It then checks if the next spot in the array is a 1. If it is, it again increments the lower and upper, repeating this process until the next spot in the array is a zero. After five seconds without a response, the server will resend a packet. Packets are not always resent when they are dropped and sometimes fall out of sync with the client.

**Struggles:**

**Alex:**

I personally struggled the most with writing the C code. I have not had a lot of experience in writing C code so this was a challenge for me. I was able to come up with the process for us to use fairly quickly on the file transfer and how to implement the window. This ended up changing soon after we started trying to implement it. Another problem we ran into was that our one byte packet on the end of the data wasn’t large enough to store a large enough number for how many packets went through.

**Josh:**

The greatest issue that I had was resending lost packets. The goal was to reset the file pointer, I know now that this is not the correct way to do it, but sometimes the client would get out of sync with the server and incorrect parts of the file would be read. Another issue was working cross platform between my Mac and Linux desktop, some things would not function properly and I got segmentation faults on my desktop that were not present on my desktop.

**Tristan:**

I had the most issue with regards to formatting a data structure for the packet. In the end we found the best use was a structure with predefine size to ensure the size remains constant across every packet regardless if full or not. The biggest issue we had was bloated code, as ideas kept compounding into the next and creating a code that was difficult to edit and even more difficult to keep track of. This was resolved in the Python code, but the python code had the issue of data structures that was already resolved in C