1. I test the basic functionality of my program. I want to ensure that it is able to process a simple input file containing “hello world”. After starting the server with “./myserver 8080”, I run the following client sided command: “./myclient 127.0.0.1 8080 3 ../doc/input_file.txt ../doc/output_file.txt”. Since I am setting the MTU to a size of 3 (meaning file data will be a byte at a time, due to my custom payload having a size of 2 bytes) the packets should be sending and receiving a byte of information each. The expected value should be an exact copy of the input file to the output file.
2. I test for client side inputs that violate the rules of our MTU. If the MTU size is smaller then the custom payload, then we should return an immediate error message.
3. I test for the efficiency of my program by passing an extremely large file size as the input text file. Since I’m running my server to accept even local connections, packet dropping should be extremely minimal. I tested a file size of nearly 1Gb a total of 3 times using the following command: “./myclient 127.0.0.1 8080 3 ../doc/input_file.txt ../doc/output_file.txt”. Not only did my program not drop any packets during the tests, the files were exactly the same.
4. I tested if my program could handle more than one client. By running my client sided program on two different terminals, you can see that the server handles a packet at a time and ensures each client is handled on a first come first serve basis. The output file for each client is the same as the input file.
5. I tested if my client program times out if the server suddenly shuts down. When testing a large file, I set the MTU size to something small enough for there to be a long echoing of packets. During this process, if I terminate the server program the client program will hang for 60 seconds before timing out.

I create a custom payload struct to maintain a sequence number and a *uint8_t* data vector for each individual packet. This helps to maintain some order between echoed packets and debug to view packet information when handling small files. To begin chunking, I read the file in binary and create packets at a set *window size.* By doing this I help maintain some reliability for the *UDP connection*. This ensures we don’t overload the server with packets and nothing is dropped. After creating this *window-sized* vector of my custom struct, *dgram_t*, I pass it to my client send/request function. This section functions similarly to the stop-and-wait method typical for a UDP connection, however now we’re doing it at a set size (because of the *window_size*). This function will iterate through the passed vector containing the payload, it will then send them one at a time in a loop. It will then loop again, receiving the expected number of packets back. Then it will simultaneously write the received packets before returning to the original file reading function. In order to maintain ordering, while we are sending the packets to the server, we match the packets sequence number to its data value using C++’s *map* data structure.