Kevin Slachta

Nowshad Azimi

Vivek Rajan

Austin Tsai

Project 3 ReadMe

In our program, we designed it according to the project guidelines. The client will need to take in at least 6 parameters, which are the -p, -c and -h flags and their respective values. We assumed that the parameters coming after the flags were all valid. If the -p, -c and -h are not in the parameter, the program will print out “Invalid” and exit. The client will also be able to take in the other 2 flags in any order. The client will recursively traverse either the default directory or a target directory looking for valid csv files. When it finds a valid csv file, it will create a new thread for it. To determine that it’s a valid csv (noted by a .csv extension), it will then perform a further check on it to figure out whether it is in the correct format. The “correct format” is specified as having 28 column headings which are in the same exact order as they were in project 0. If the file is validated to be in the correct format, then we create a new socket, connect it to the port and send the file over. We first send the command “nowsh” followed by the column number. For example, if we want to sort by colors, we use column 0. Sorting by director name means that we send “nowsh 1”. Then we send the entire file over. Once a file has been sent, we wait for a response from the server saying that it has finished sorting that file. Then we join the child thread. Once all the threads have been joined, we sent a dump request to the server. After getting the dump request, we wait for the server to sort out the final csv and give it back to us, where we will then place it in the output directory.

The server will open a connection and wait for a response from the client. Whenever it receives a connection request, it will make a thread and first verify what sort of action to take. If the send request is “nowsh” followed by a column name, then the server will prepare to receive a file from the connection. After it has received the file, the server will then sort that file into a large BST on the server side and will send back “done” to the client thread once it has fully sorted the file into the BST. If the send request is “dump”, then the server will send the entire sorted file over to the client. It will then continue to run unless it is killed.

We had some difficulties on our end. At first, we were able to establish a connection between the server and client. However, we had errors when it came to sending files over. We also had seg faults when it came to running the client side. For some reason, our program would exit when it began to get the IP address of the host name. We first hard coded in the IP and it would work. After, we slowly went about the steps to figuring out how we would find the IP addresses. We also had issues at first about checking valid csvs. However, we soon found out that some of our test files were in an incorrect format. When it came to sending the file to the server, we had issues in transferring. We found that we needed to specify the file size. When it came to keeping the port and the host name, we decided to use a global variable to keep track so all our other methods are able to access it. When we finally fixed all our compilation errors, we encountered errors with the output. It was a simple error where there was one less line than it should have. We also had an error where there was no header on the output csv file.

In order to use our code, you need to compile the client with the following command “gcc -pthread sorterClient.c -o client”. The server can be compiled with the MakeFile. This will allow you to run our program. The actual running will be in the same format specified in the assignment description, where it can take in a variety of inputs from 6 – 10. However, the input variables should be an even number and in the specified format. Otherwise, the code will output an error message. We wrote our code under the assumption that the “-c”, “-d”, “-p”, “-h” and “-o” parameters are legitimate and what follows them is a valid parameter/directory. If the input or output directories do not exist, our code will exit with an error message.

We tested it with a very small test case with only 1 csv file. We then began to work with multiple csv files and see if it works. Then we tried performing the same test but with different directories. Finally, we tested it with multiple csv files and multiple directories and checked if it could sort correctly. We utilized the python code provided by a fellow student to test our code against thousands of csv files and hundreds of thousands of csv entries in the final directory file. We ran it and outputted the sorted csv file to multiple directories and took data from multiple directories.

Extra Credit

Our program should be able to handle a limited number of sockets. What we do is that if there is a specified number of maximum sockets when passed in as a parameter, our program will first create the maximum number of sockets. Once that's done, we will commence the file search as usual. However, we implemented a queue for the threads that need to send a file over the sockets. If all the sockets are taken up and they have not received the "done" command from the server, any new thread that requests access to the socket will be put into a queue. They remain in this queue until one of the threads receives the "done" command, where the first in thread in the queue will then connect to one of the sockets and proceed to transfer the file over to the server. Other than this implementation, our program runs the exact same way.