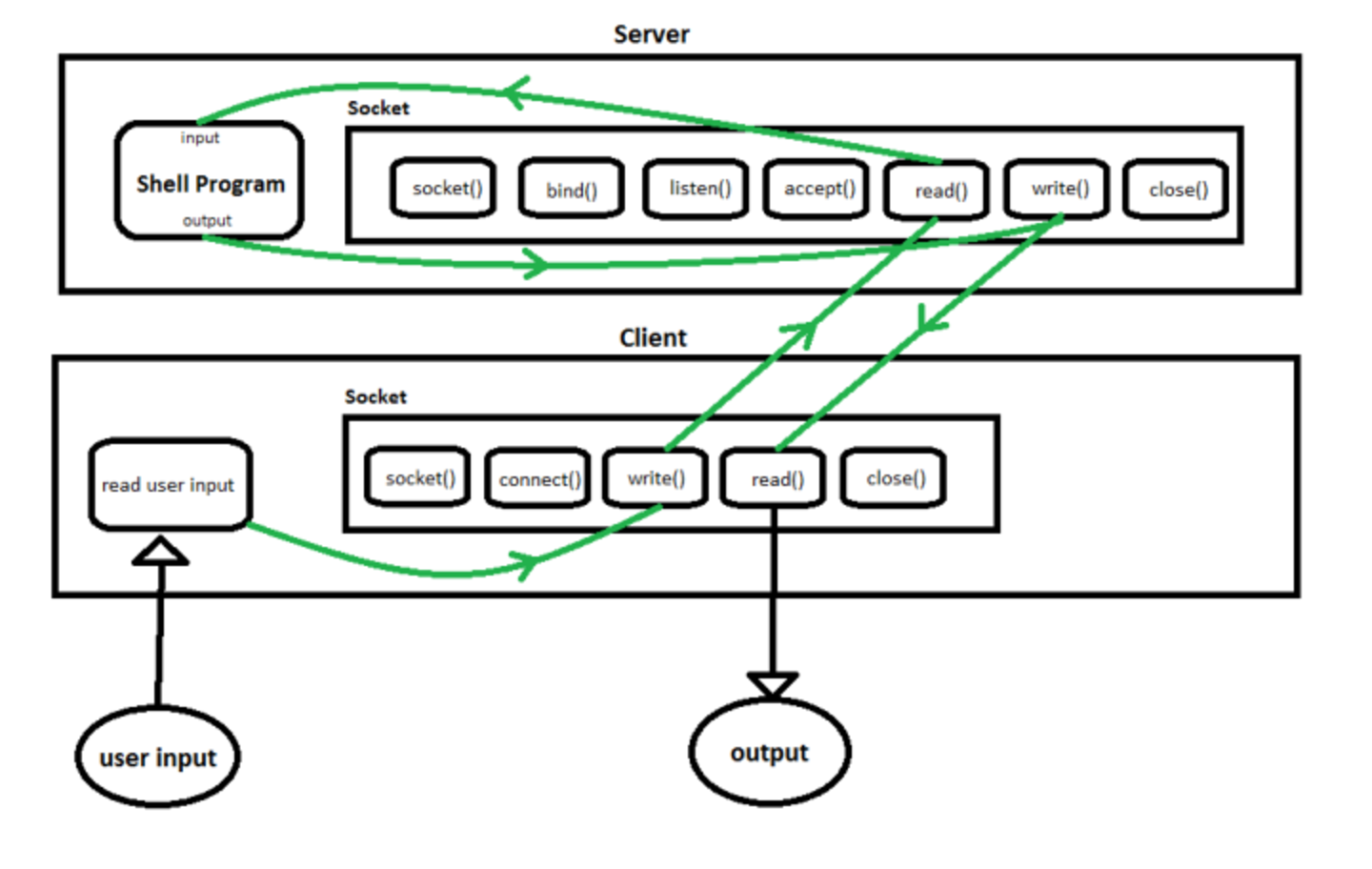
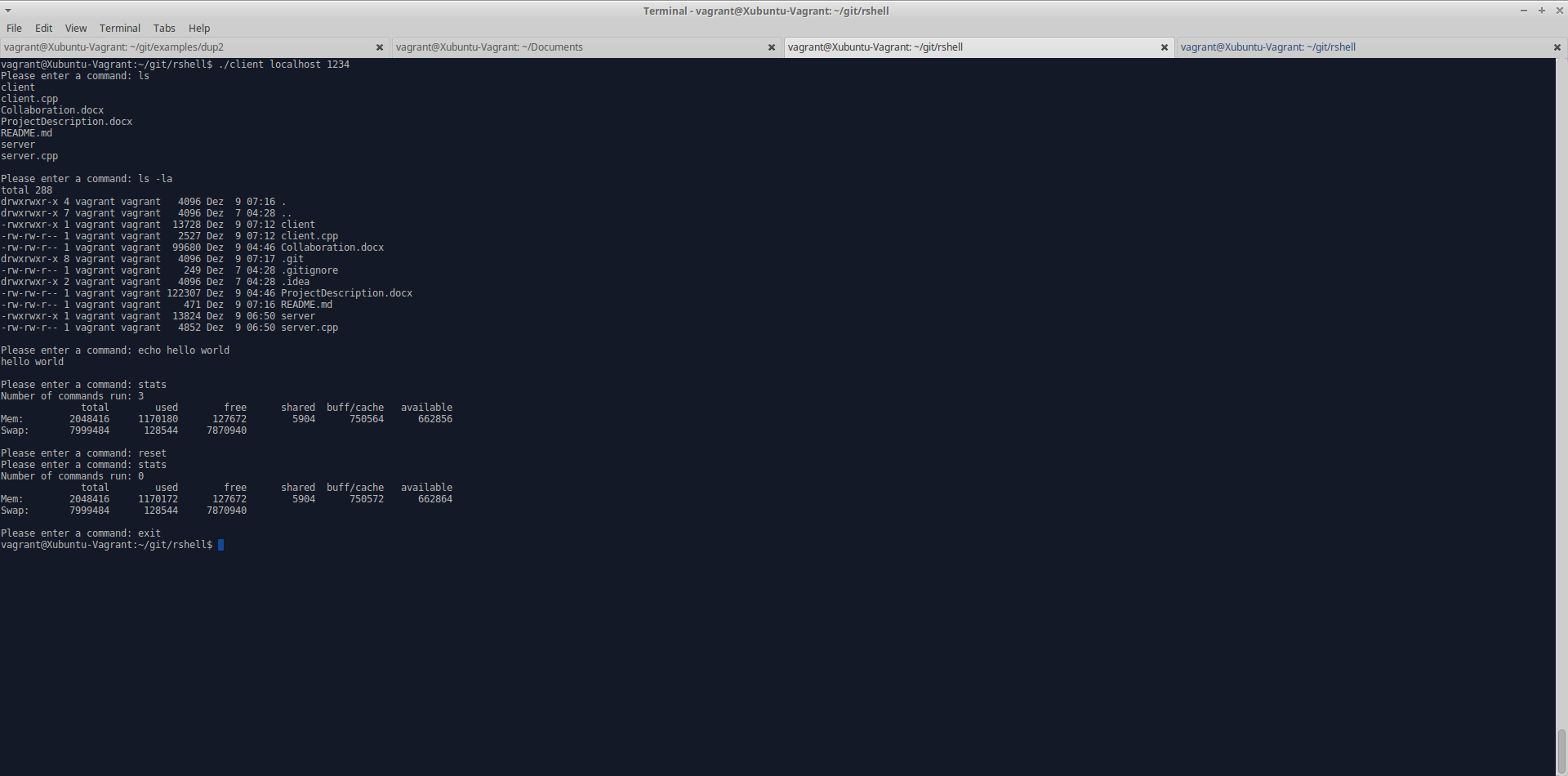
This project consisted of building a remote shell server and remote shell client. To build this remote shell server and client, what we needed was a combination of sockets and a shell program that will execute the commands that are sent to the server. The client side will read in the input from the user in the form of a command and some arguments, or just a command (in the case of ls). There will be a socket on the server side listening for this input. This input will be put into the shell program and the command will then execute. Using sockets again, the output will be redirected back to the client, where the console will show the results. The entire process can be visualized in the following diagram.



The client side of the project was simple. The most important things we had to do was create a file descriptor and associate the port number that was created by the server and link it with the client to establish the connection. In addition, we had to have the address of the server and a struct to store the host information. Once we have established a secure connection (if we didn’t, we throw an error), we finally move on to the user facing part of the application. We ask the user to enter a command and we place this command into a character array of size 256 called “buffer”. If the user entered “exit”, we stop the program and terminate execution. However, if they entered a viable command, we write that command to a socket. By writing to the socket, we are enabling the server side to listen to our input. Control is now transferred over to the server, where it “reads” our input.  
 In the server, we do much of the same setup for file descriptors and sockets as we did in the client. One key difference is that we are now listening for socket connections from the client. We will block anything else until this connection between client and server is established. We throw the appropriate error if this connection failed. Finally, we enter our while loop where we process the input. We read the bytes sent from the client. Again, we throw the appropriate error if reading from the socket failed. We then start actually processing the buffer. We use an interesting strategy where we count the number of spaces in the buffer because we want to know the length of the char pointer array we are about to build, called args. Args[0] will hold the command, and the rest of args will hold the arguments passed. We pass these two arguments as parameters in the execute function. Once we execute the command, we must be able to redirect this output back to the client so that we can display the results in the terminal. To do this, we use the dup2 command. More specifically we say “dup2(newsockfd, STDOUT\_FILENO)”. We changed the file descriptor to print to client rather than standard output.

Finally, in the client, we keep track of the statistics. We have a mechanism to keep track of the total number of commands entered. The user simply must type in “stats”, where a conditional will kick in and just print out the total number of commands entered up until that point. The total number of commands entered is a running count.



As shown in the image above, our code successfully executes any command entered. ls, ls –la, and echo work exactly as expected.