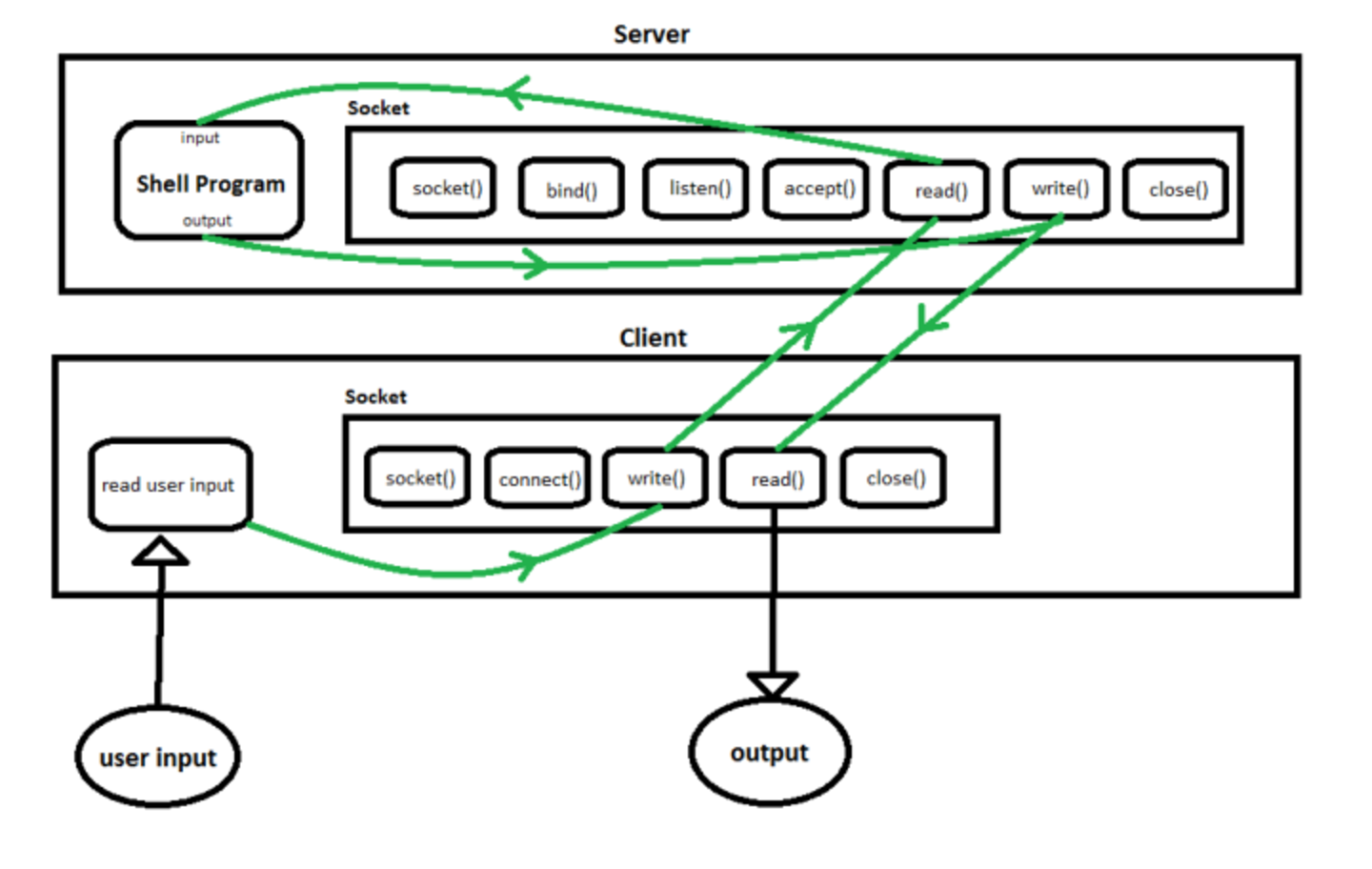
This project consisted of building a remote shell server and remote shell client. To build this remote shell server and client, we used a combination of sockets and a shell program that would execute the commands that were sent to the server. The client side would read in the input from the user in the form of a command and any possible arguments. The socket on the server side listened for this input, and executed the command that was passed to it. Using the dup2 function, the output of the server was set to the input of the client, and thus the results would print out to the client terminal. The entire process can be visualized in the following diagram:

**Control Flow**



**The Client Side**

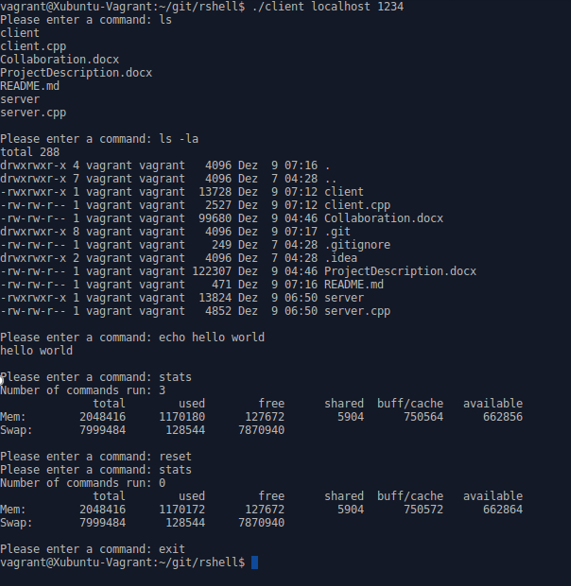
We approached the project first from the client side. A socket for the client was initialized by using the series of commands shown in the Client box in the Control Flow diagram above. Once the client socket had connected to the Server socket, the code entered into a while loop that iterated until the user asked to exit the program. In this while loop, we prompted for the user to enter a command to be sent to the server, and we placed this command into a character array called “buffer”. We checked buffer for values such as “exit”, “stats”, and “reset”, in case the user wanted to leave the program, see the program statistics, or reset the program statistics. If the character array found inside the buffer was not any of these predefined character arrays, then we allowed the output to be sent to the server by passing it as a parameter into the socket. Upon every iteration of the while loop, if “buffer” was found to be a command than a command counter would be incremented. If the buffer was found to contain “stats”, then this integer would be printed to the terminal along with other system information.

**The Server Side**

The server side started much the same way as the client side; a socket was initialized by the commands listed in the Control Flow diagram above. The server side socket would, however, block at accept until a client socket was found. Once a secure connection was established, (upon failure to do so an error was thrown), a file descriptor to the client socket was obtained and the program moved into a while loop in which input was repeatedly read and executed. Before the input from the buffer could actually be executed, it had to be briefly processed into a form that could be passed into the execvp function. Thus, the character array received was split into individual character array, which were all put into an array of character arrays. In order to generate a buffer that was the correct size to hold all of these words, we wrote code that would count the spaces in the character buffer and then added one to represent the total number of words in the string. A null pointer was then added at the end of the array so that Execvp knew where the list of arguments ended.

After using fork() to create another process, and Execvp takes two inputs, the command and the array that contained both the command and all the arguments. Therefore, we passed in the first element of the array as the first argument and the entire array itself as the second argument. In order to direct the output of the execvp function back to the client, the standard output file descriptor of the server file was replaced with the file descriptor that was obtained from the accept function earlier. To do this, we used the dup2 command.

**Example of Program Output**



As shown in the image above, our code successfully executes any command entered, such as ls, ls –la, and echo. Also shown is the functionality of the stats command, which first prints out the number of commands issued thus far, followed by system memory information. The reset command, as shown above, resets the command count back to zero. The exit command halts the program.

Code can be found at <https://github.com/christopher18/rshell> along with more information about how to run the program (in the README.md file).