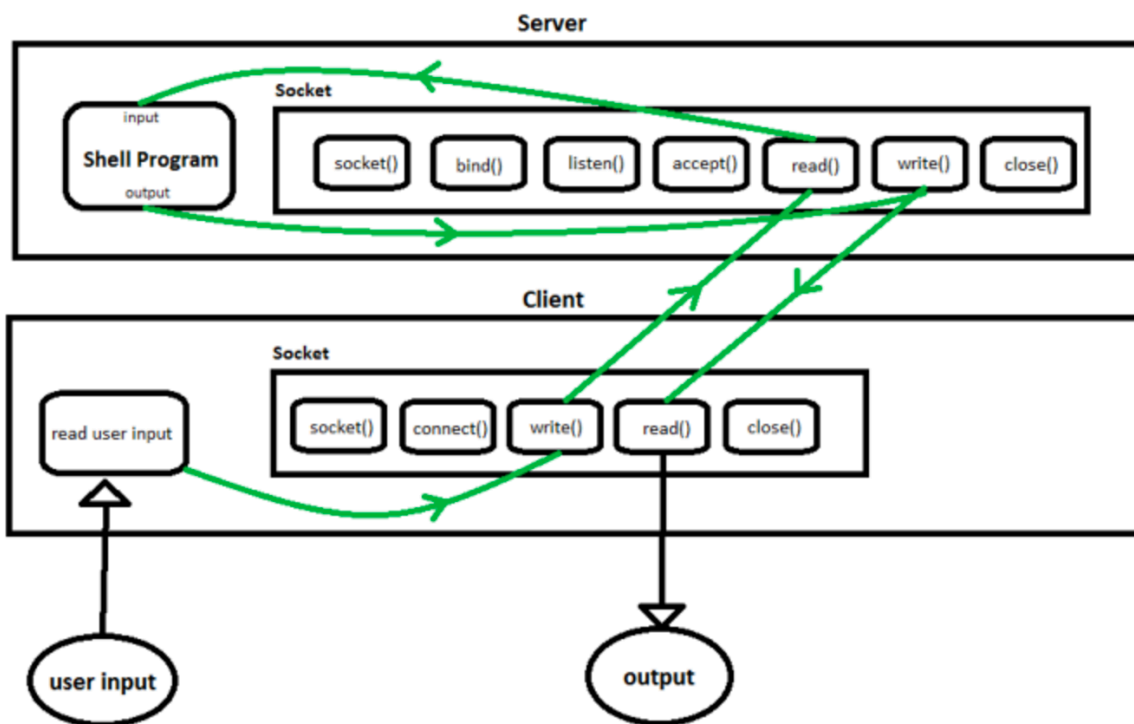


Dhruv Yadav

Chris Acker

<https://github.com/christopher18/rshell>

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.

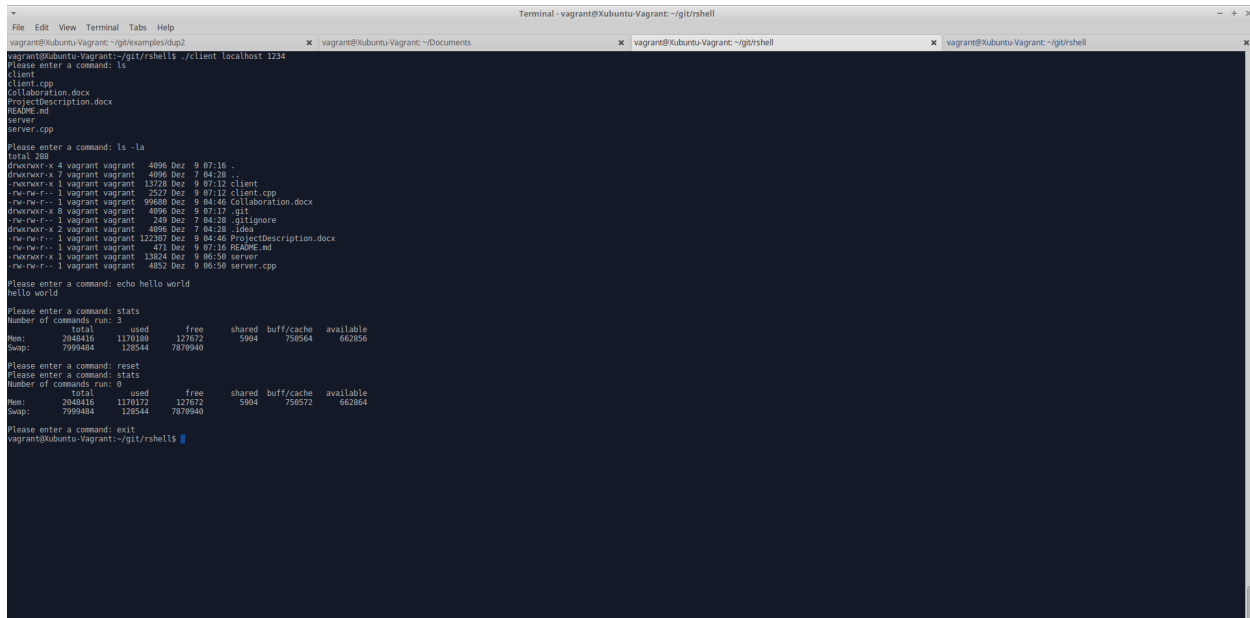


The client side of the project was simple. The most important things we had to do was create a file descriptor and associate the port number that was created by the server and link it with the client to establish the connection. In addition, we had to have the address of the

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.



```
Terminal - vagrant@Xubuntu-Vagrant: ~/git/rshell
vagrant@Xubuntu-Vagrant: ~/git/rshell$ ./client localhost 1234
Please enter a command: ls
client
client.cpp
Collaboration.docx
ProjectDescription.docx
README.md
server
server.cpp
Please enter a command: ls -la
total 288
drwxr-xr-x 4 vagrant vagrant 4096 Dez  9 07:16 .
drwxr-xr-x 7 vagrant vagrant 4096 Dez  7 04:28 ..
-rw-rw-r-- 1 vagrant vagrant 13728 Dez  9 07:12 client
-rw-rw-r-- 1 vagrant vagrant 2527 Dez  9 07:12 client.cpp
-rw-rw-r-- 1 vagrant vagrant 59888 Dez  9 04:46 Collaboration.docx
drwxr-xr-x 8 vagrant vagrant 4096 Dez  9 07:17 .git
-rwxr-xr-x 1 vagrant vagrant 349 Dez  7 04:28 gillipore
drwxr-xr-x 2 vagrant vagrant 4096 Dez  7 04:28 .idea
-rw-rw-r-- 1 vagrant vagrant 122397 Dez  9 04:46 ProjectDescription.docx
-rw-rw-r-- 1 vagrant vagrant 471 Dez  9 07:16 README.md
-rwxr-xr-x 1 vagrant vagrant 13824 Dez  9 06:50 server
-rw-rw-r-- 1 vagrant vagrant 4852 Dez  9 06:50 server.cpp
Please enter a command: echo hello world
hello world
Please enter a command: stats
Number of commands run: 3
      total      used      free      shared buff/cache   available
Mem:    2048416  1178189    127672    5904    758564    662856
Swap:    7209484    126344    7870940
Please enter a command: reset
Please enter a command: stats
Number of commands run: 0
      total      used      free      shared buff/cache   available
Mem:    2048416  1178172    127672    5904    758572    662884
Swap:    7209484    126344    7870940
Please enter a command: exit
vagrant@Xubuntu-Vagrant: ~/git/rshell$
```

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

Collaboration – Word Breakdown

Chris

- Majority of the client.cpp file
 - Wrote code on client for opening sockets
 - Wrote code to get hosts
 - Wrote code on getting the server addresses
 - Wrote the while loop to get the commands from the user and sent them through sockets to the server side
- Majority of the server.cpp file
 - Wrote code on forking and creating new processes
 - Wrote code on checking to see if the child terminated successfully or not
 - Wrote code on creating sockets and setting the server address values
 - Bound the socket to a particular port as specified by the user
 - Wrote code on listening for sockets and blocking all else until a connection was established between client and server
 - Read the bytes sent from the client using sockets
 - Did some char array string manipulation to get the total number of words in the command sent.
 - Created a char ** array that would put the first word of the buffer into an array at the 0th index, and put the rest of the buffer (the arguments) into the following array spots. These would serve as the parameters to pass in the execute function.
 - Did file descriptor stuff to send the output back to client.

Dhruv

- Some of the server.cpp file
 - Did some char array string manipulation to get the total number of words in the command sent.
 - Created a char ** array that would put the first word of the buffer into an array at the 0th index, and put the rest of the buffer (the arguments) into the following array spots. These would serve as the parameters to pass in the execute function.
- Worked on the final report
- Worked on the PowerPoint presentation slides
- Did the one page collaboration document

