Sidharth Gilela
sgilela@ucsc.edu
Final Project

Design Document

Goal:

The objective of this project was to be able to implement an ftp client and server application where the client and server are able to communicate with one another. The client sends certain requests to the server and the server is either able to or not able to process those requests. The server is concurrent in order to allow multiple clients to connect to it simultaneously. This project was great way to understand more about how the ftp protocol works which is widely used everywhere. I had a great time implementing this project.

Compile:

To compile my code run:

```
make all
```

Design:

I will explain the design of my code through a sample run between the client and the server. What I basically have in the client is an infinite while loop that continually takes input from the user and executes the command by sending the command to the server. The first thing the server does after accepting a connection from the client is fork itself so that way the server can implement concurrency. Inside the child, all the following logic is implemented. When the user types in quit, the while loop is broken and the client code is done executing. Now, the client is able to do 4 commands. The get, put, ls , and quit. Anytime any of these commands execute, the first thing the client does is send a PORT command to the server. The port number sent along with the PORT command is a random port generated and will be the port number of the data port on the client. The server processes this PORT command, creates a new data port which is its current port – 1 and sends back a PORT OK message to the client in the control channel. This PORT OK message is sent after it is able to connect with the data port that listening on the client side. This way the client knows that the PORT command was successful and that the data channel between the client and server is successful. If failing to do so, an error message is sent back, and the user gets told to enter a different command. Now, after this PORT command, a LIST, STOR, RETR, or QUIT request is sent to the server. LIST corresponds with ls , STOR corresponds with put, and RETR corresponds with get, and QUIT corresponds with quit. If LIST or RETR is sent to the server, the server takes in the buffer and converts the buffer to have an ls [filename] or a cat [filename] correspondingly. This way I can run popen on the command and send back the output to the client. All of the output that server executes is sent on the data port of the server and received on the data port of the client. However, if an error message occurs instead, then the error message is sent through the control port of the server and received on the control port of the client. This happens during a while send on the server so that the server can keep sending back 4096 bytes until it is done sending back the data. Meanwhile the client should be doing a while read to be able to constantly read sizes of 4096 bytes until the data is done being sent by the server. This is actually the tricky part where I have implemented select on the client side. The reason being is because the client does not know if it will get an error message on the control port or data on the data port. Thus, we implement select and have two ports ready, we can expect either output and then break out the select's while loop. If no error message is sent back, then I wait on the control port of the client to receive the 200 OK for the corresponding command that it has just executed. The put command works a little differently in my design because now the opposite occurs and the client is sending data to the server. Thus, in order to implement this, I basically do the reverse, meaning I do a popen on the client side and do a select on the server side. If successfully able to send all the data of the file to the server, it will send a 200 OK at the end to the server so that way the server knows that everything went well. For this implementation, a STOR request is sent to correspond to the put command. Lastly, the quit command was a lot simpler. I did not have to send the PORT request for PORT because the server and client do not need a data connection. No data will be sent back to the client in response to the QUIT request. Instead. I just do a read on the control port on the client side and send back a 200 OK from the server to the client when the server is successfully able to quit. After sending back the quit message, both control ports on the client and server are closed. The client exits its infinite while loop and is done executing. The server just continues to wait on accept, waiting for new connections to come to it.