OPERATING SYSTEMS & NETWORKS
ASSIGNMENT 2
- PROTOCOL DESIGN -

CANDIDATE NUMBER 52655

Introduction

This document consists of the designs and discussions for a file transfer protocol to be implemented using the Java programming language. The protocol involves the creation of both a client and a server that can communicate using file transfer tools. The server should be implemented to constantly run once executed, and the client implemented to take in specific commands from the user in order to communicate with the server which will automatically respond. Also, multiple clients should be able to connect to the one server at the same time.

File Directories

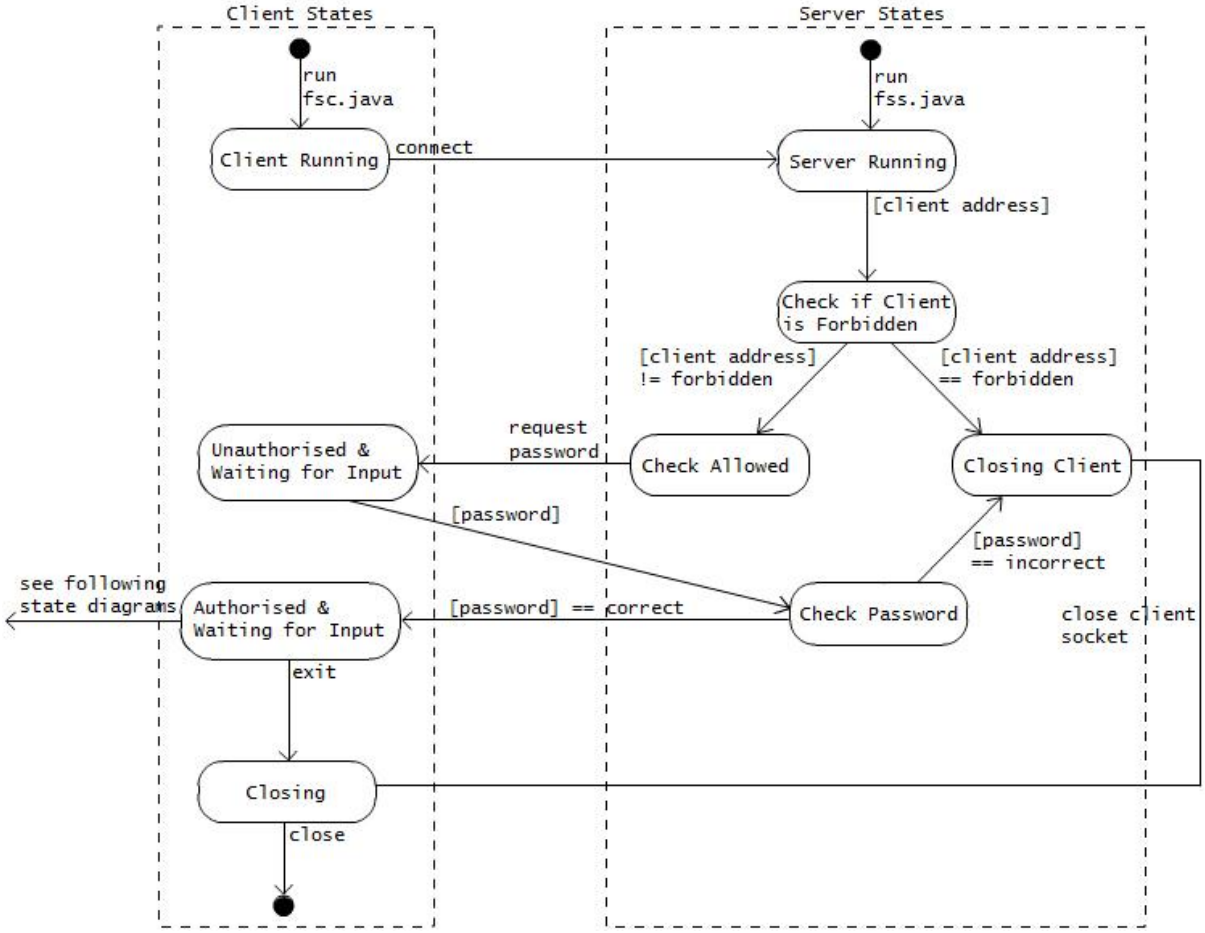
Both the client and server should have access to files stored in either of their current directories. So, the current directory of the client will be the directory in which it is stored and same for the server. Files will be added as objects to an array on both the client and server sides and the arrays should be updated with every iteration of the protocol.

Multithreading

In order for multiple clients to be able to connect and communicate with the server, the server will assign each connected client to its own thread.

Authorisation (password and forbidden.txt)

Once the server is run and a client connects to it, the first thing the server should do is check to see that the address of the client is not listed in the text file "forbidden.txt" which will be stored with the server. This file is used to ban connections from specific addresses. If the clients address isn't listed in the text file, the server then prompts for a password from the client. This should bring up a text field allowing the user to type, however no commands should work just yet. The user must enter the valid server password to progress to further states of the protocol. The state diagram below demonstrates the processes the protocol undergoes before the client gains authorised access to the server.



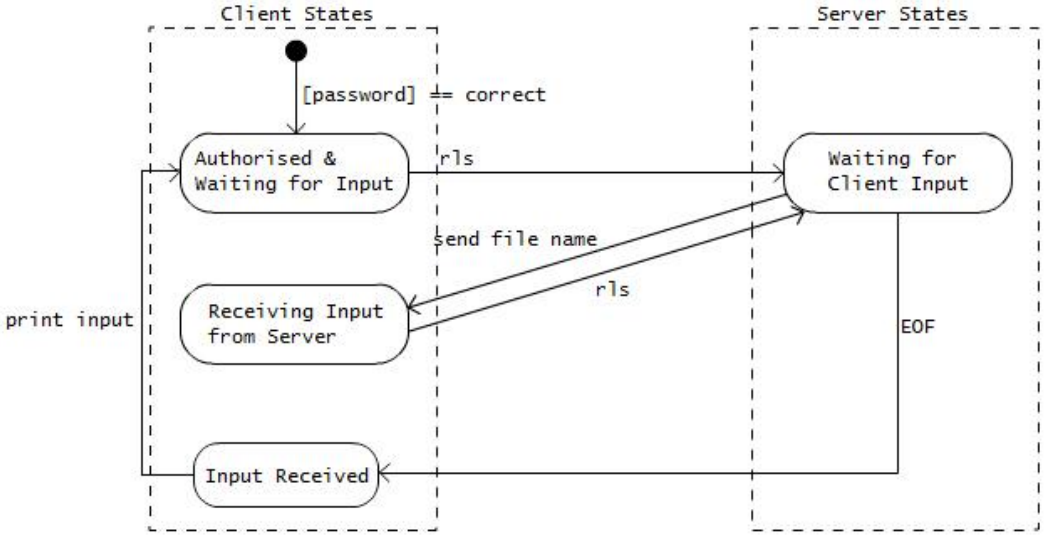
As the diagram demonstrates, a forbidden client address or an incorrect password will result in the client socket closing and therefore the connection terminating. Otherwise, the client will reach an authorised state and remain connected to the server; the client will then prompt for user input in the form of a command. The user can choose the type "enter" which will terminate the client, or one of four other commands that will be discussed on the following pages.

List Local Directory (Command: "lls")

The command "lls" simply reads through each file name in the file array of the client's current directory and prints it out to the terminal window.

List Remote Directory(Command: "rls")

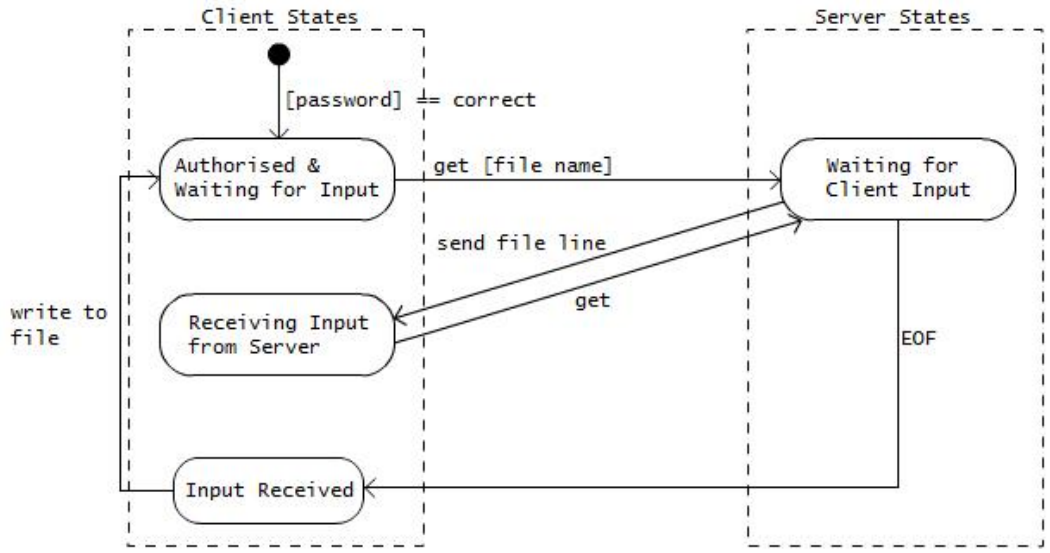
The command "rls" is slightly more complicated the "lls" in that it makes a request to the server for each file name in the file array of the server's current directory. This means the server must read through each file name and send the names, file by file, to the client. The state diagram below demonstrates how this should be implemented.



The diagram shows that if the user input is "rls", the server will send a single file name to the client, the client will then automatically send "rls" to the server again which will send the next file name to the client. This will continue until the server has read through to the last file in its file array, where it will then send an "EOF" (end-of-file) notification. The client will then print the input from the server to the terminal and then will return to the "Waiting for Input" state whereby commands can again be entered.

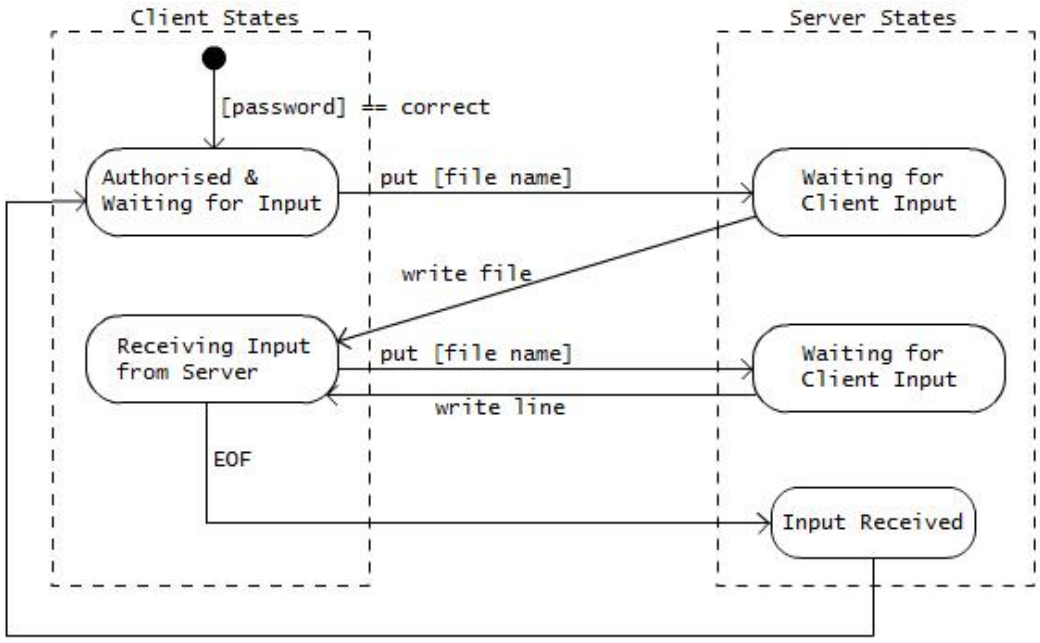
Get File(Command: "get [file name]")

The command "get [file name]" is concerned with downloading a file from the server to the client. The file name is specified in the command and the server will read through its file array to compare it with the specified file name. If it is found, the server will read the file, one line at a time, and send it to the client, one line at a time. It is a similar data flow to that of the "rls" command in that the client constantly requests the file as each line of the file is sent by the server. This continues until the server sends an "EOF" message and the client will then write these lines to a file. It then will return to the "Waiting for Input" state whereby commands can again be entered. The diagram below demonstrates how this is to be implemented.



Put File(Command: "put [file name]")

The command "put [file name]" is concerned with uploading a file from the client to the server. The file name is specified in the command and the client will read through its file array to compare it with the specified name. If it is found the server uses the file name to create the file. The client will then read the file, one line at a time, and send it to the server, one line at a time. For every line received by the server, it will write that line to the file created. This will continues until the client sends an "EOF" as opposed to more "put" commands. Once this is done, the client will return to the "Waiting for Input" state whereby commands can again be entered. The diagram below demonstrates this.



Exit(Command: "exit")

As discussed earlier, this command is will result in the client connection to the server being terminated. However, the server should still run.