CIS 457 – Winter 2024
Project 1: Building Multi-Threading FTP Server
Date Posted: 2/14/24.
Date Assigned: 2/14/24.
Due by: 3/15/24 by 10:00AM EST; late submission policy (20%/day for up-to 5 days including weekends/holidays)
Point Value: 100 pts

Project Demo and Work Protocol:

- For this project, (4) students will team up to work on this project as a group. The groups are already pre-formed as per the info posted in this spreadsheet https://docs.google.com/spreadsheets/d/1hvASVAsRFT22SzAhDwJJ8pEWgs7JMc_mc1QDJbnzRz4/edit#gid=0

- Students need to submit the following items for grading the project; otherwise, no grade will be given to the submitted work:

1. A 5-minute video to demo the project work. Students can take parts of the demo, or only one can demo the entire project work. **In the video, please state (i) the completion percentage of the project and (ii) what functions worked and what functions did not work.** The group will demo the project during our meeting on **Fri 3/15**
2. A copy of your project code **(multi-threading FTP application)** with a written report showing a screen capture demonstrating how **each service** provided by the FTP server works. The services to be provided by your code are: **connect, list, retr, stor and quit**.
3. The report should include the basic logic of the program, how you implemented each service, what problems you have encountered, and how you solved them. The programs should be submitted in an electronic version containing all your source codes.

   All materials listed in items (1)-(3) must be uploaded to BB using the link "SubmitYourProject01Here"

4. **The team members are responsible for:**
   a. Writing, debugging, and making the project work as required using any programming language the team deems appropriate.
   b. Sample Code is provided as part of the project materials to just show you the code's overall structure. You can use it as a guide to start building your code or simply ignore it and design your code from scratch.
   c. Dividing the project work equally among the team members as well as contributing fairly to the overall project deliverables.
   d. If this is not the case, you may consider leaving the group and forming another group, as well as informing the instructor about making this change. This action must occur within the first few days after assigning the project. Please, Do NOT wait until the last minute to make this decision.

5. Self and peer evaluation may be conducted randomly to get more info about how the project work was going during the course of the project.
6. You may use ChatGPT to get a starting code. Please document the use of ChatGPT and which sections of your project code are obtained

from ChatGPT. No more than 25% of the project code can be obtained from ChatGPT.

Introduction

Most network applications rely on file transfer protocols in one form or the other. For instance, the HTTP protocol used on the Web is a generic file transfer protocol. In this project, you will implement an FTP client program and an FTP server program for a simple file transfer. The server could handle one or more file transfers to the client (s) at any given time. The project implements an **active FTP** app, as described in the project assignment. However, you may choose to implement FTP running in passive mode, which is okay too. The implemented FTP application is assumed to support text file transfer. The client program presents a command line interface that allows a user to:

- Connect to a server
- List files located on the server.
- Get (retrieve) a file from the server.
- Put (store) a file from the client to the server.
- Terminate the connection to the server.

The server program binds to a port and listens for requests from a client. After a client connects to the server, the server waits for commands. When the client sends a terminate message (quit), the server terminates the connection and waits for the next connection.

Implementation

You can implement your project using any programming language as two independent programs: an FTP client called **ftp_client** and an FTP server called **ftp_server**. The ftp_client program presents a command line interface. The client and server communication needs to be done using TCP sockets. The client should be able to send FTP commands to the server. On receiving a command, the server should parse the command and perform the appropriate action. The format of the commands is as follows:

1. **CONNECT** <server name/IP address> <server port>: This command allows a client to connect to a server. The arguments are the name/IP address of the server – for instance, cis.gvsu.edu – and the port number on which the server is listening for connections. Once the control connection is established, a data connection with the server has to be set up for each data transaction operation (You can have this part hardcoded in the client and server code).
2. **LIST**: When this command is sent to the server, it returns a list of the files in the current directory on which it executes. The client should get the list and display it on the screen.
3. **RETR** <filename>: This command allows a client to get a file specified by its filename from the server.
4. **STOR** <filename>: This command allows a client to send a file specified by its filename to the server.
5. **QUIT**: This command allows a client to terminate the control connection. On receiving this command, the client should send it to the server and terminate the connection. When the ftp_server receives the quit command, it should close its end of the connection.

Client/Server Interaction

To implement the communication between the client and the server, you need to use 2 TCP connections (control and data connections) at both the client and the server.

- The control connection between the ftp_client and the ftp_server is used for sending control messages (commands sent by the client and their associated responses coming back from the server).
- The data connection should be used for sending/receiving data between the ftp_client and the ftp_server programs.

The client/server interaction also has another interesting aspect. The ftp_server program acts as the server on the control port, and the ftp_client acts as the server for the data port. By "acting as a server," I mean creating a socket, binding to the socket, and listening for and accepting connections on the socket. This interaction happens in two steps:

- First, the ftp_client connects to the ftp_server over the control port. After establishing the control connection, you need to establish a data connection. The data connection is established and torn down after each command.

- Second, after the ftp_client sends a command (list, retrieve, and store) to the server where it expects a response – it opens the data port as the server. The ftp_server program connects to the data port established by ftp_client and sends the response. When you receive an **EOF** (end-of-file) on the data connection, you know the server has no more data to send, and the connection can be terminated. In the case of the put command, you need to wait for a connection to be established by the ftp_server before you send any data.

Why all this complexity?

Why can't we just use one socket and use TCP's ability to do bi-directional communication on a socket to send data both ways. A little thought will show that you now have to worry about both ASCII control messages as well as binary data flowing over the socket. Think about the sequence of actions that happen on the store command. The client sends the message and then starts transferring the file. If you happen to transfer a file that has your grocery list in it and the first line of the file says list, the server will get confused. Also, if you use a single connection you would need to worry about the length of each command and response. If you use separate control and data connections, where the data connection is torn down after each command, you can use the **EOF** indication on the data connection to infer that there is no more data to send.

The next question to ask is what port numbers you should use for the control port and the data port. For the **control port (server port)** the port number should be the last 4 digits of your G#. If the last 4 digits of your G# are less than 1024, add 10000 to the last 4 digits. For example, if the last 4 digits of your G# are 0123, your control port number should be 10123. For the data port number add 2 to the control port number using the same calculation as above. For example, if your control port is 3478, your data port should be 3480. You may hard code the control port number in your server program.

Project Grading Policy

Please note the following grading policy:

- **Not** implementing a multithreaded server leads to deducting **20Pts** from the project score.
- **Not** implementing any FTP function correctly leads to deducting **20Pts** from the project score.
- Work turned in after the due date and time will receive a 20% late penalty per day, including the weekends/holidays, with no more than a total of 5 days of being late in your submission.

FTP Server Implementation Details

The basic logic behind the FTPClient is first when the program starts the user must enter the **connect** command followed by an IP address of the server, which they wish to be connected to and the port number the server is listening on. To do this the user must enter the command space ip address space, port number. The program will ask for the user to put in the input, from there it will find out if the command from the user starts with connect. If it does, then it will parse out the server ip and port number and establish a tcp connection with that server. This connection will be the control connection. After the control connection has been established then the other commands can be processed. Once there is a control connection established, the user can enter commands and get requests from the server.

The first command to implement is the **list** command. For the list command, the client program gets the input from the user, adds the port number that it will be listening on, and send the command to the server over the control connection. Once the server receives this command it looks in its current directory and lists all the files that are in it. Once the server is ready to list of the files in the directory, it creates a data connection on the port number specified by the client and sends the list followed by "**eof**" to indicate there are no more files. Once the "**eof**" is sent by the server and received by the client the data connection is closed, and the user sees the list of files on their screen.

The **retr** command sends a request for a particular file over the control connection. Once the server gets this command it looks through the directory it is currently running in and searches for the file. If the file is there the server send a status code of 200 command ok, and sends the file. If the file is not there it sends a status code of 550 and the user can enter another command. This is implemented by first the client program reads in the input from the user. Once it gets the input it appends a port number to the front of the command that the client program will be listening on to receive the file. The client program then creates a server socket, which will be used for the data connection. Then it sends the request over the control connection that has already been established. Once the server gets the request it separates the port number, command, and file that it needs to get. Once this is done if the file exists it creates a TCP connection with the client on the port the client specified and sends the requested file. It sends the file by reading it in line by line. Once one line of the file is read in, it sends the line over to the client. On the client side, it creates a file with the same file name and write the data in the file line by line. Once the file is finished being sent a string

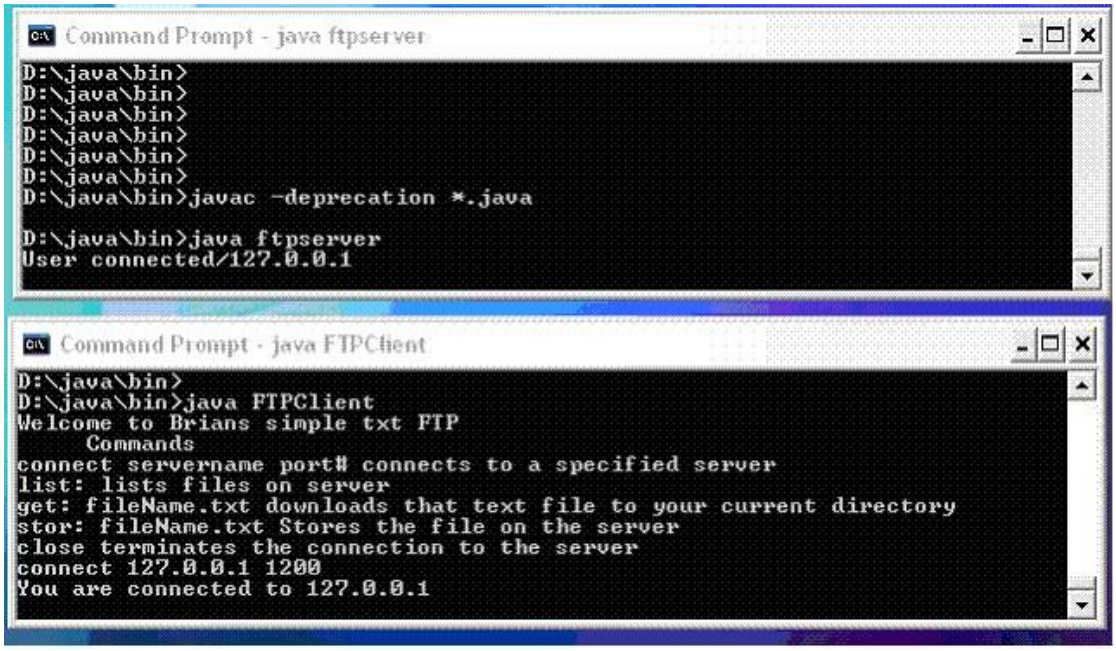
containing eof is sent to tell the client that it is the end of the file and both sides terminate the data connection but keep open the control connection.

The **stor** command is very similar to the retr command but in reverser. The stor command will first get the command from the user. Once it gets the command it will search through its directory to see if that file exists or not. If the file does exists then it will go ahead and send the port number it will be listing on and the command over to the server. Once the server gets this command it creates the file in the current directory and if that is successful it creates the data connection with the client. Once the connection is established, the client then read the file line by line and sends it over to the server, which writes it in the file line by line. This is implemented in the same way as get but in reverse because the client is sending the file and the server is receiving the file.

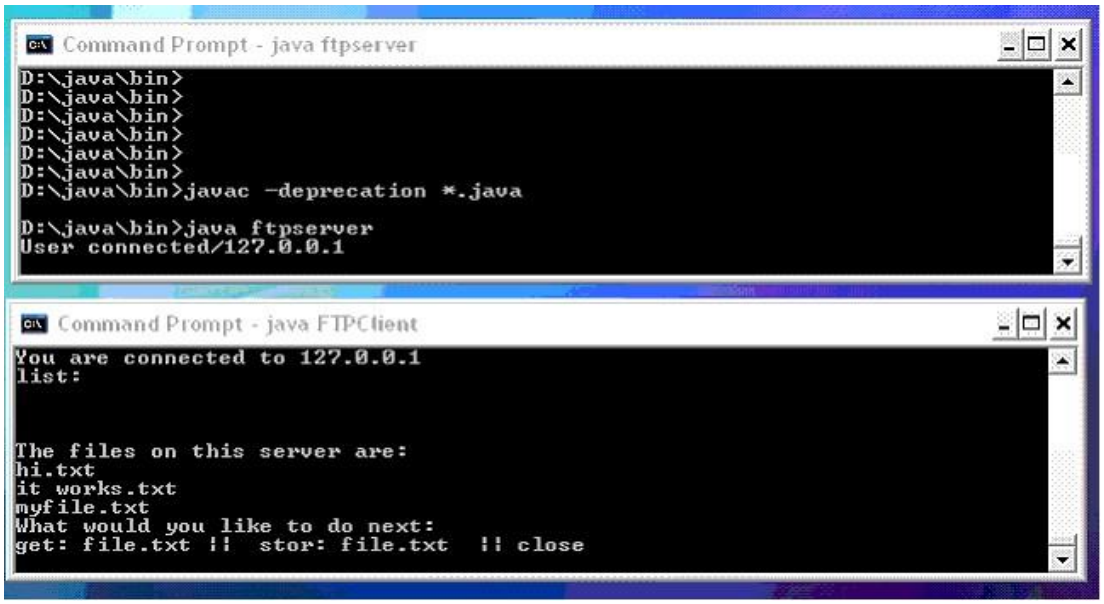
The last command to be implemented was the close command. This command is when the user wants to terminate the connection between the server. To do this is first the client will wait for the input from the user. If that input is equal to close the client sends the command to the server. Then server closes the control connection and so does the client.

Screen Shots of FTP Program

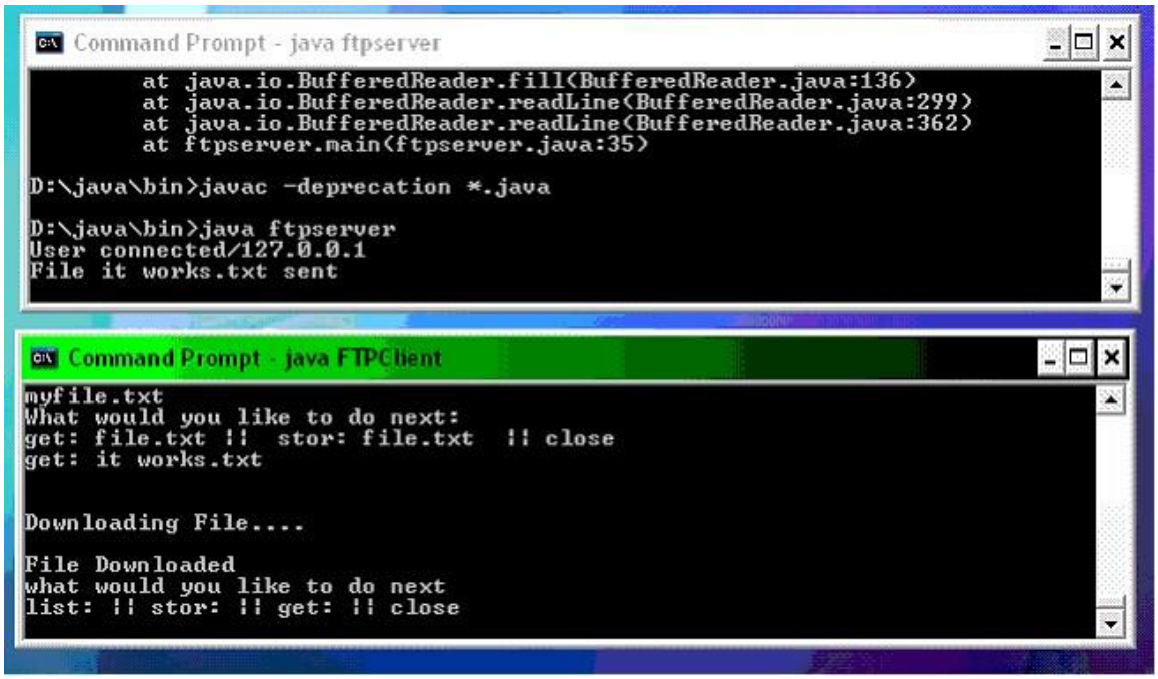
Command **Connect :** Connect server-IP Port



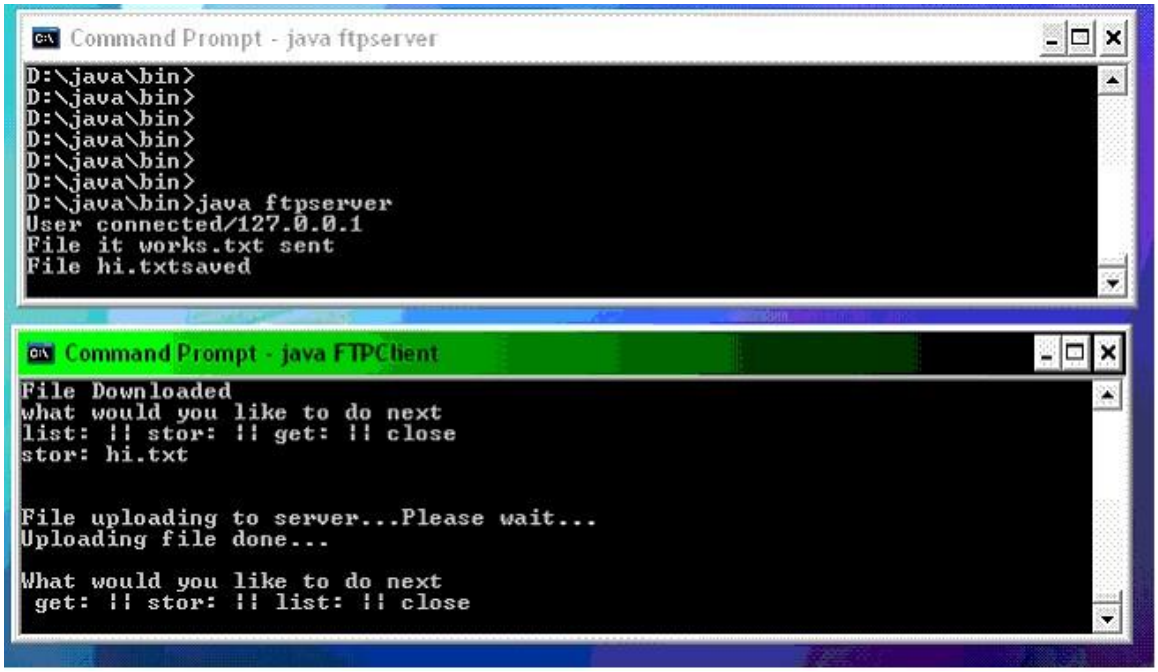
Command **List:**



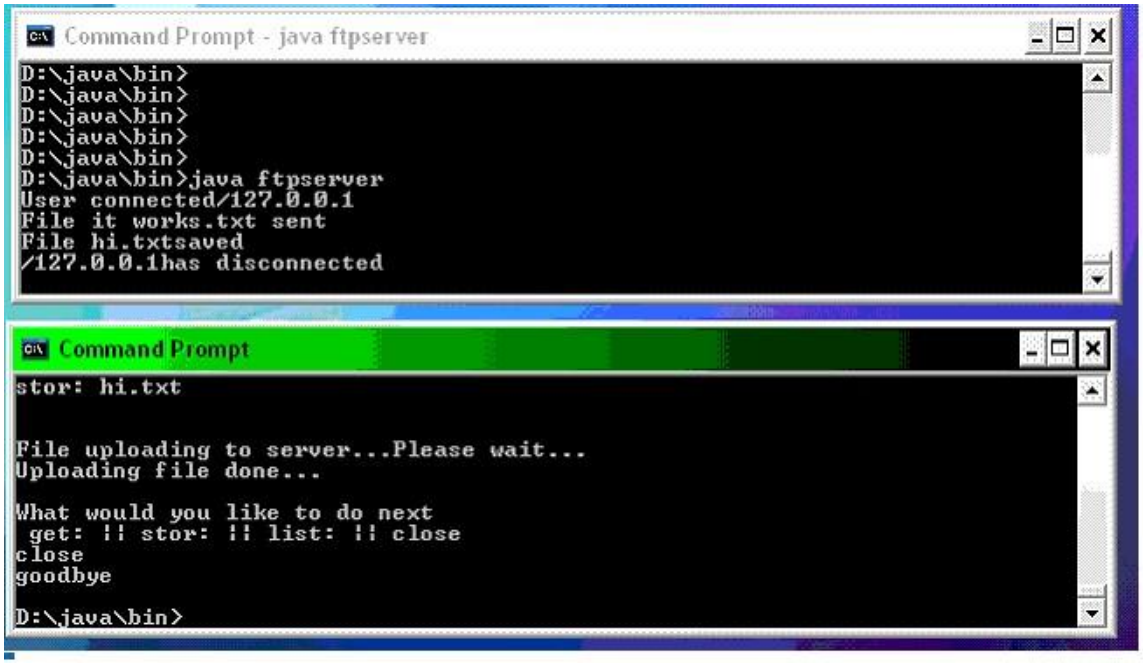
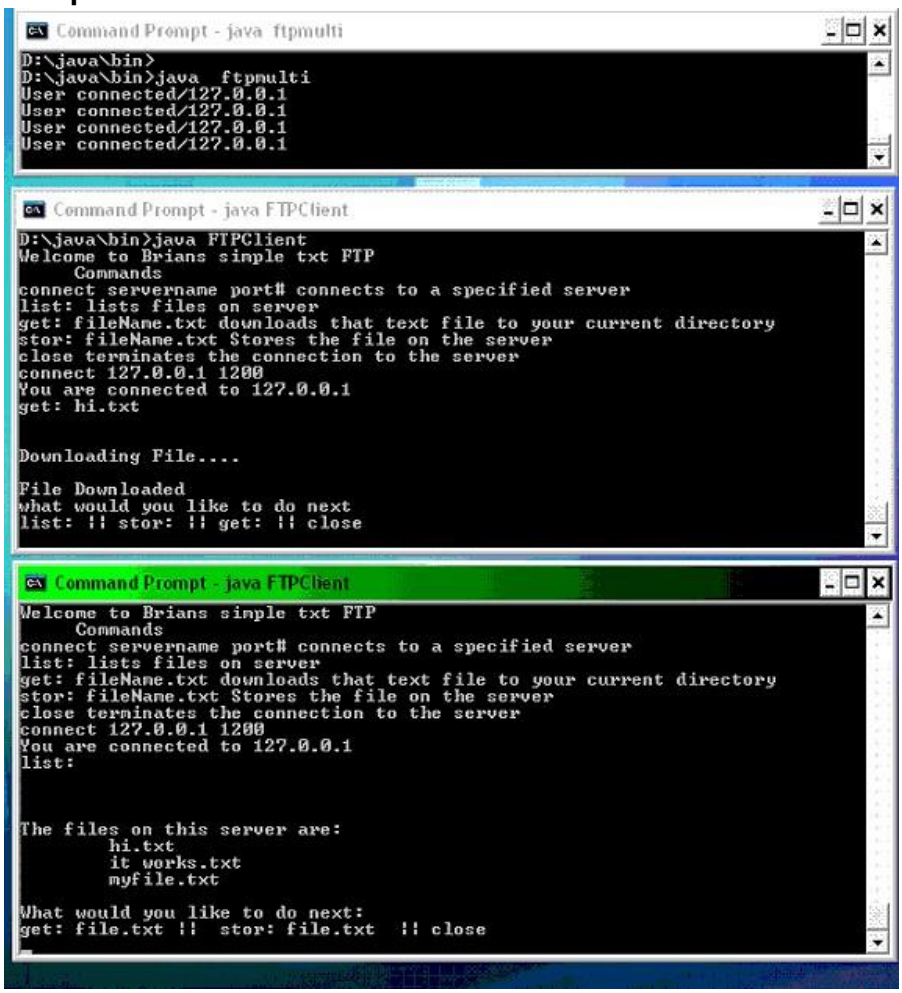
Command **Get/retr:** filename.txt



Command **Stor:**



Command **Close:**



Multiple Clients at once