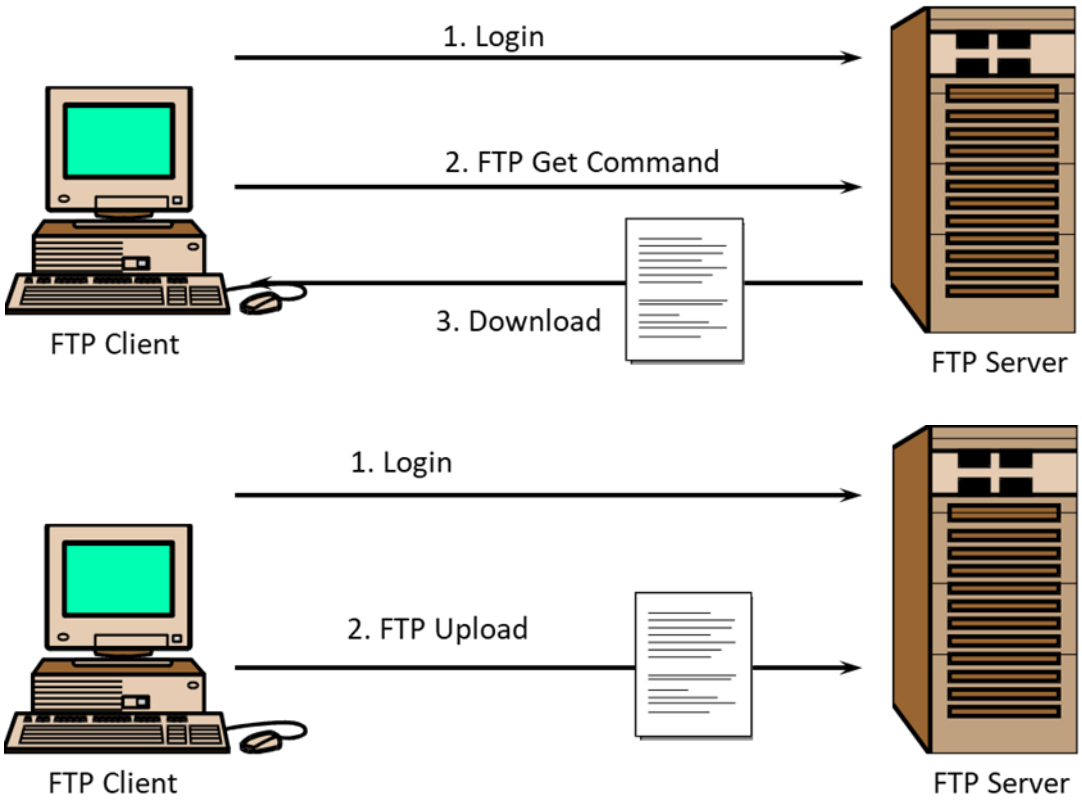
# Project 2: File Transfer Protocol (FTP) Server Software Application

## Objectives

1. To learn how to interpret and use a computer network protocol's Request for Comments (RFC) by Internet Engineering Taskforce (IETF) as an Internet Standard.
2. To understand File Transfer Protocol (FTP) from RFC 959.
3. To learn and use Unix Network or Socket programming API.
4. To implement FTP Server software according to Internet Standard outlined in RFC 959.
5. To implement FPT Server software using C++ programming language and Unix Network or Socket programming API.

## Specifications

You have been using **File Transfer Protocol** (FTP) to get or to store files from or to FTP servers. You use a FTP Client software application in your host computer to communicate with an FTP server. FTP server runs a FTP Server software application.



In this project, you are going to implement your own FTP Server software application in C++ programming language using Unix Network or Socket programming API.

FTP is a client-server protocol to transfer files to or from the servers. An FTP client sends FTP request messages to an FTP server. FTP server interprets the request message, takes appropriate action, and sends back response message to the client. **RFC 959** describes FTP and its request and response messages in detail. You need to read and understand **RFC 959** to complete this project. Although, **RFC 959** describes many request messages, you need to implement only the followings:

- **USER <username>**
- **PASS <password>**
- **PWD**
- **CWD <dirname>**
- **CDUP**
- **PASV**
- **NLST**
- **SIZE <filename>**
- **RETR <filename>**
- **QUIT**

Your FTP Server software must be compatible with the FTP Client software of your first project, which has a command-line user interface (UI) and the users can enter following user commands through that interface.

- **help**
- **user <username>**
- **pass <password>**
- **pwd**
- **dir**
- **cwd <dirname>**
- **cdup**
- **get <filename>**
- **quit**

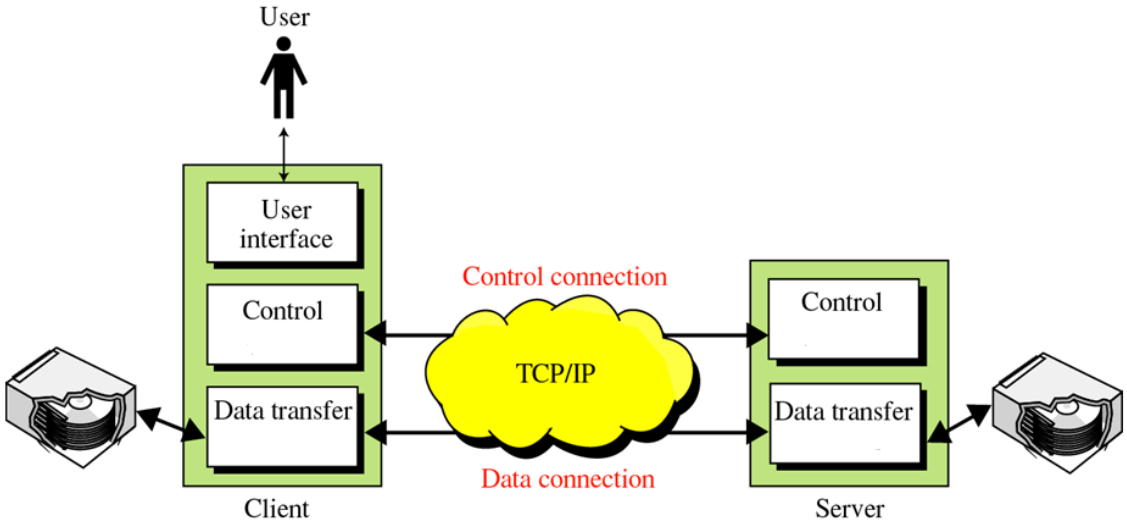
Some of the user commands have an argument. For example, command **user** has an argument <**username**>. A user command and its argument is always space separated.

– Command '**help**' displays the list of commands supported by this software application, their syntax, and meaning.

- Command '**user <username>**' sends username to the FTP server for authentication using '**USER <username>**' FTP request message. You need to support only one user with user name 'csci460' and password '460pass'.
- Command '**pass <password>**' sends the password to the FTP server for authentication using '**PASS <password>**' FTP request message.
- Command '**pwd**' sends '**PWD**' FTP request message to the FTP server in order to print the current working directory of the FTP server.
- Command '**dir**' sends '**PASV**' and '**NLST**' FTP request messages in order to list the contents of the current working directory of the FTP server.
- Command '**cwd <dirname>**' sends '**CWD <dirname>**' FTP request message to the FTP server in order to change the current working directory to another directory specified by <**dirname**>. If the specified directory is beyond current working directory of FTP server, an error is reported by the server.
- Command '**cdup**' sends '**CDUP**' FTP request message in order to change the current working directory to its parent directory. If the parent directory is beyond the root directory of FTP server process, an error is reported by the server.
- Command '**get <filename>**' sends '**SIZE <filename>**', '**PASV**', and '**RETR <filename>**' FTP request messages to FTP server in order to fetch the specified file from the current working directory of FTP server. If the specified file is not available an error is reported by the server.
- Command '**quit**' sends 'QUIT' FTP request message to FTP server in order to inform the FTP server that the client application is quitting, so that the server can close the connection gracefully. It also closes the client connection gracefully and terminates the software.

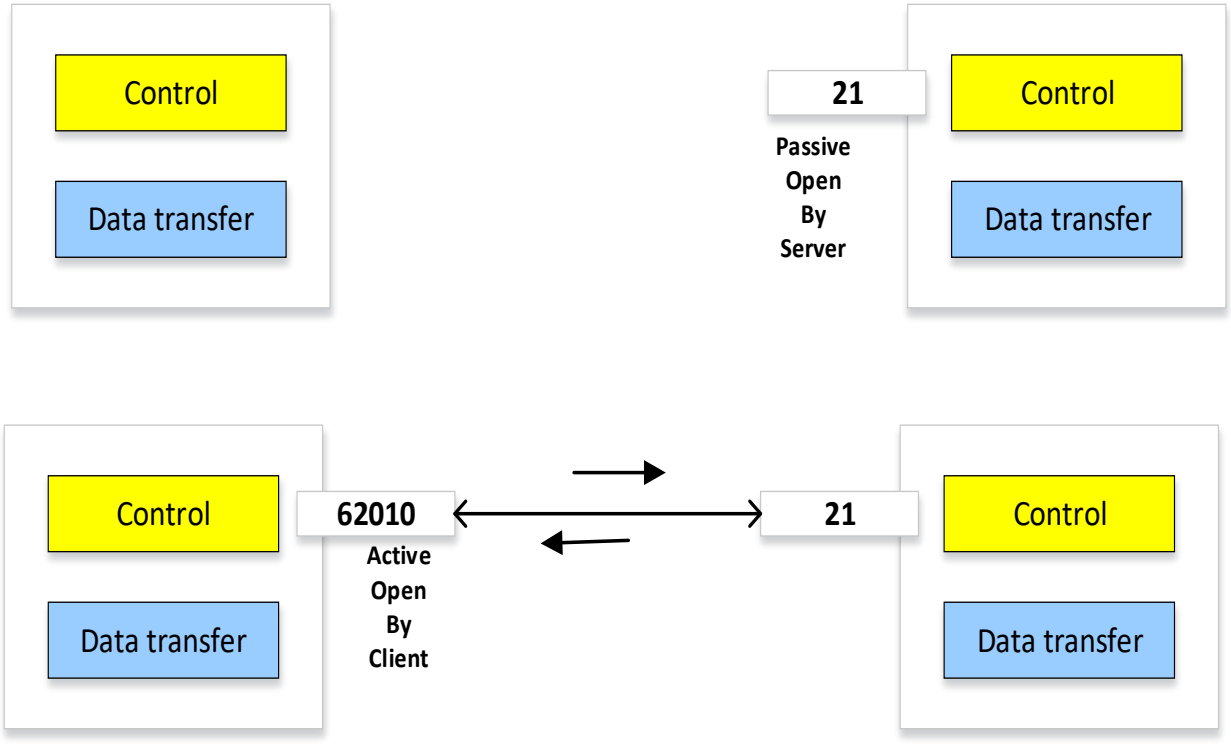
User commands of your FTP client application have different syntax than that of FTP request messages. User commands are more human readable. A user command has one or more corresponding FTP request message(s).

FTP client application interprets above user commands, translates them into appropriate FTP request messages, sends the FTP request messages to FTP server, receives the response messages from the server, and presents the response to the user in a user-friendly manner.



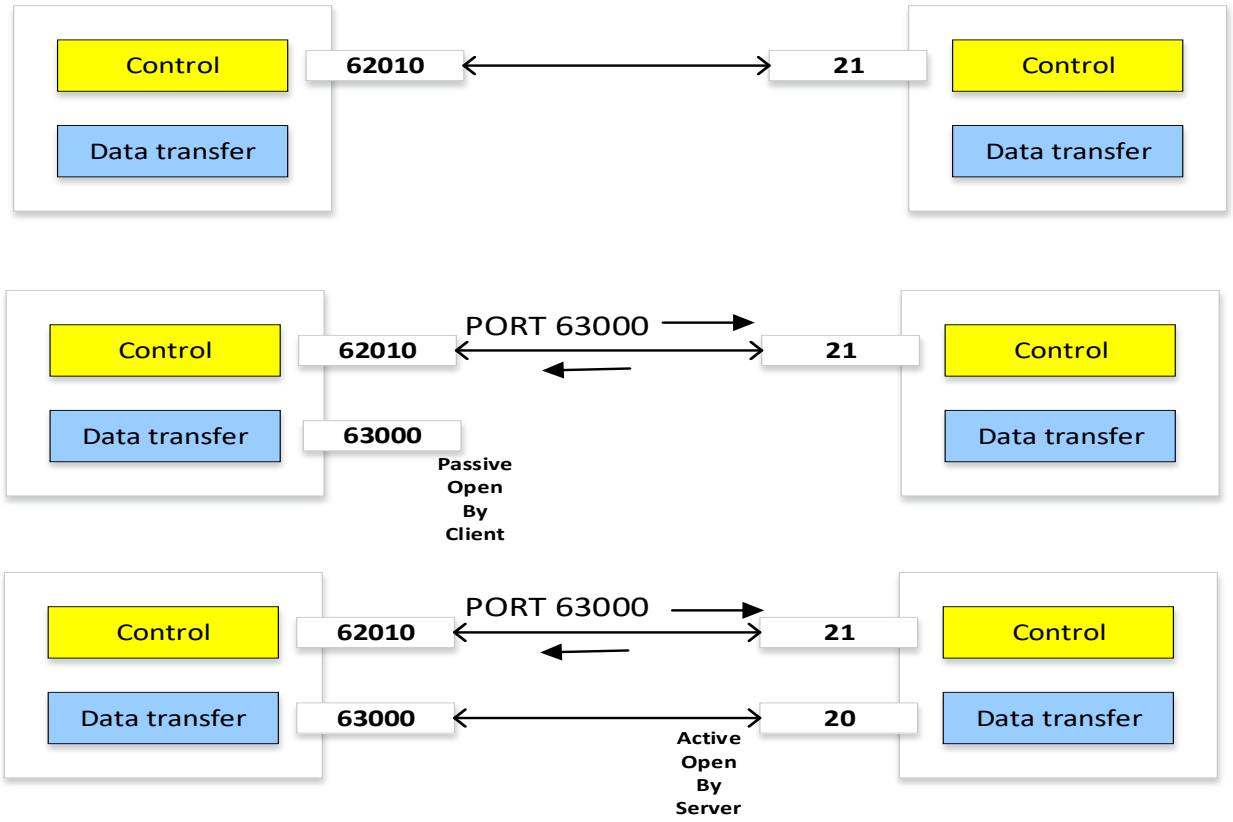

FTP Client and Server communicate request and response messages over a control connection.


FTP Server passively waits for a control connection and FTP Client actively opens the control connection.

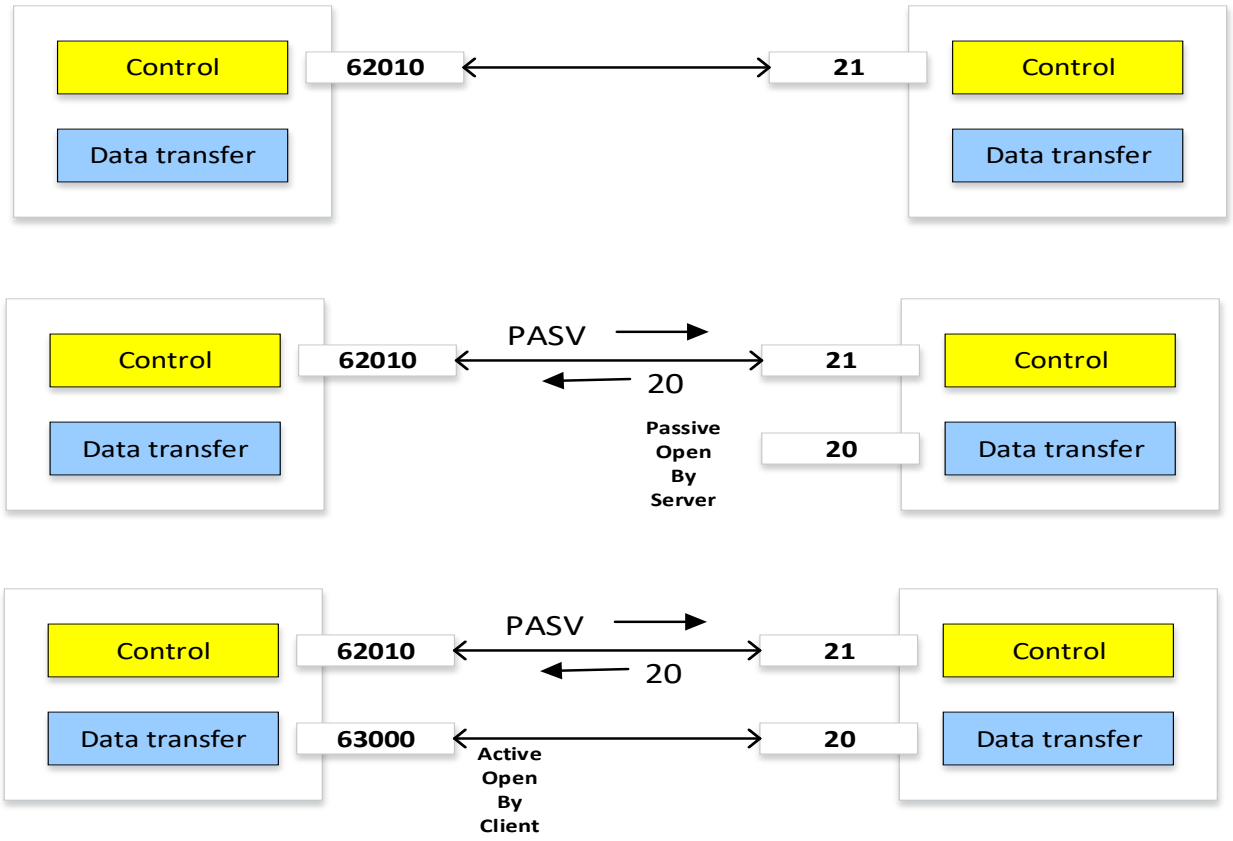



Some requests/responses involve a data transmission. For example, both NLST and RETR requests involve a data transmission to transfer data as the part of a successful response. FTP server uses a separate connection for each data transmission. A data connection is opened on demand and closed when a data transmission is complete.


FTP server can operate either in **active** or in **passive** mode to open a data connection. In active mode, FTP server opens the data connection with the client on demand. At first, FTP Client choose an ephemeral port and opens a connection listener on this port. FTP Client sends this port number to the server using PORT request message and waits for the server to open the connection. After receiving the port number, FTP server actively opens the data connection.




If the client is behind a firewall, FTP active mode fails to open a data connection. FTP passive mode has been proposed to solve this problem. In passive mode, when a data connection is required the server passively opens a connection listener so that client can send connection request to the listener to open a data connection. Client instructs the server to enter into passive mode by sending a PASV request message to the server. The server opens the connection listener on a port and sends the port number to the client in its PASV response. After receiving the PASV response, the client retrieves listener port number from the response and sends a connection request to the listener port in order to open a data connection.




In this project, you must implement passive mode and your FTP server software application, must handle a PASV request before any data request, such as RETR and NLST.

## Tasks

1. You will work on and submit this project using **GIT submission system** of the department. A central repository named **project2** has already been created for this project.

2. Create your own fork of **project2** on the central GIT repository using following command.

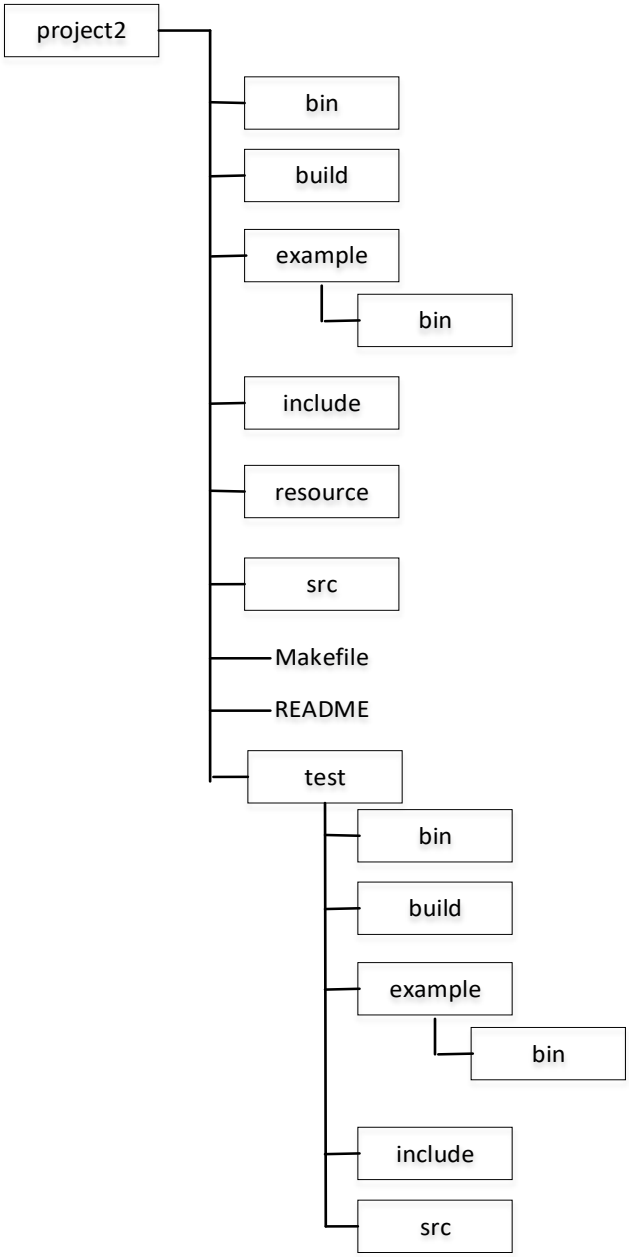
   *ssh csci fork csci460/project2 csci460/$USER/project2*

3. Go into your **csci460** folder that you have created in your project1.

4. Create a clone of your forked **project2** repository using following GIT command.

   *git clone csci:csci460/$USER/project2*

5. Repository **project2** has been organized as follows:




6. Continue your work in your cloned or local **project2** repository and **commit** and **push** your work to your central **project2** repository as it progresses**.** Instructor is expecting lots of incremental commits in the repository. Few number of commits is a red flag of **academic misconduct**. Instructor, will take further steps to verify the integrity of your work if there is any red flag of academic misconduct.

7. You will use supplied **Makefile** and **Unix make** utility to **build**, **run**, and **test** your server application. You don't need to and should not modify this Makefile.

8. A **README** file template has been placed in the root of the application development folder. The README file gives a general idea of the application, technologies used in developing the application, how to build and install the application, how to use the application, list of contributors to the application, and what type of license is given to the users of the application. You will need to complete the **README** file before your final submission.

9. Following header (***.h)** files have been supplied in **include** folder in the repository.
    a. **ftp_server_connection_listener.h**
    b. **ftp_server_connection.h**
    c. **ftp_server_net_util.h**
    d. **ftp_server_nlist.h**
    e. **ftp_server_passive.h**
    f. **ftp_server_request.h**
    g. **ftp_server_response.h**
    h. **ftp_server_retrieve.h**
    i. **ftp_server_session.h**

    You will need to implement the functions specified in the header files in corresponding source code files. For example, all functions in header file '**ftp_server_request.h**' should be implemented in '**ftp_server_request.cpp**' file. You are not allowed to change any header file.

**10.** All of your source (*.**cpp**) files should be in **src** folder of the project. The source code of **ftp_server.cpp** file has been supplied, you don't need to and should not change this code. Implement following **cpp** files in your **src** folder.
    a. **ftp_server_connection_listener.cpp**
    b. **ftp_server_connection.cpp**
    c. **ftp_server_net_util.cpp**
    d. **ftp_server_nlist.cpp**
    e. **ftp_server_passive.cpp**
    f. **ftp_server_request.cpp**
    g. **ftp_server_response.cpp**
    h. **ftp_server_retrieve.cpp**
    i. **ftp_server_session.cpp**
    j. **ftp_server.cpp**

11. The lists of internal dependencies of the **cpp** files in this project are as follows. You will need to include dependent header (***.h**) files in each of your source (***.cpp**) file.
    a. **ftp_server_net_util.cpp**
        i. ftp_server_net_util.h
    b. **ftp_server_connection.cpp**
        i. ftp_server_connection.h
    c. **ftp_server_connection_listener.cpp**
        i. ftp_server_connection_listener.h
        ii. ftp_server_net_util.h
    d. **ftp_server_session.cpp**
        i. ftp_server_session.h
        ii. ftp_server_net_util.h
        iii. ftp_server_connection.h
        iv. ftp_server_request.h
        v. ftp_server_response.h
    e. **ftp_server_request.cpp**
        i. ftp_server_request.h
        ii. ftp_server_net_util.h
        iii. ftp_server_session.h
        iv. ftp_server_connection.h
        v. ftp_server_passive.h
        vi. ftp_server_nlist.h
        vii. ftp_server_retrieve.h
        viii. ftp_server_response.h
    f. **ftp_server_nlist.cpp**
        i. ftp_server_nlist.h
    g. **ftp_server_passive.cpp**
        i. ftp_server_passive.h
        ii. ftp_server_connection_listener.h
        iii. ftp_server_connection.h
        iv. ftp_server_net_util.h
        v. ftp_server_response.h
    h. **ftp_server_retrieve.cpp**
        i. ftp_server_retrieve.h
        ii. ftp_server_connection.h
        iii. ftp_server_response.h
    i. **ftp_server.cpp**
        i. ftp_server_net_util.h
        ii. ftp_server_connection_listener.h
        iii. ftp_server_session.h

12. As mentioned earlier, you will **build** your FTP Server software application using build tool **make**. Build tool **make** will compile your source codes from **src** folder and save all object files in **build** folder. Build tool (**make)** will link all object files into a single executable and save the binary file ('**ftpserver**') of the application in **bin** folder. Build tool will also link necessary object files from **test/build** and **build** folders to create a test binary (**ftpservertest**) in **test/bin** folder.

13. An **example** FTP Server Application binary file (**ftpserver**) and an example FTP Client binary file (**ftpclient**) have been given in **example/bin** folder. An **example** FTP Server Test binary file (**ftpservertest**) has been given in **test/example/bin** folder. Make clean command will not delete these binary files and remember not to delete them yourself.

14. Test code file named **ftp_server_test.cpp** has been placed in **test/src** folder. Your instructor has implemented all the functions prototyped in all the header files in **test/include** folder in the corresponding source code files in **test/src** folder and these source files are not exposed to you for technical reasons. You will not need to write these test codes either. Compiled object files (**ftp_server_test.o**, **ftp_server_test_net_util.o**, and **ftp_client_test_command.o**) from your instructor's codes have been supplied in **test/build** folder and build tool **make** will use these object files and other object files from your source code to create test binary (**ftpservertest**) in **test/bin** folder. Command **make clean** will not delete object files (**ftp_server_test.o**, **ftp_server_test_net_util.o**, and **ftp_client_test_command.o**) from **test/build** folder and you must not delete them either.

15. To see how the example **ftpservertest** works for **deliverable1** enter following from your project root folder and observe console outputs.

    *make test-example-deliverable1*

    These outputs will give you the idea about how many test cases your own code has to pass. Your instructor recommends you to give close attention to these test cases. Your project evaluation will suffer if your code does not pass all the test cases.

16. To see how the example **ftpservertest** works for **deliverable2** enter following from your project root folder and observe console outputs.

    *make test-example-deliverable2*

17. To see how the example **ftpservertest** works for **deliverable3** enter following from your project root folder and observe console outputs.

*make test-example-deliverable3*

18. To see how the example **ftpservertest** works for **deliverable1**, **deliverable2**, and **deliverable3** together enter following from your project root folder and observe console outputs.

    *make test-example*

19. In order to see how the example **ftpserver** works, you need to run both example **ftpserver** and example **ftpclient** in separate shell terminals. You need to run the example **ftpserver** first in one terminal and then the example **ftpclient** in another terminal.
    a. Open two terminals and go into your project root directory in both terminals.
    b. In order to run example **ftpserver**, type following command in one terminal.

       *make run-example-server*

    **c.** In order to run example **ftpclient**, type following command in the other terminal.

       *make run-example-client*

    **d.** Example **ftpclient** will be connected to example **ftpserver** automatically and will be ready to accept user commands and send the appropriate FTP requests to the example server.
    **e.** Play with all the commands that the FTP client has to support. If you play enough with the example **ftplclient** and example **ftpserver**, it will give you good idea what is expected from you to develop in this project.

    Note that all the supplied binaries were built and tested only in Linux Debian machines of the lab. Run them in other machines at your own risks.

20. Type **make clean** command from the project root folder to clean up the artefacts (binary and object files) of the project and start with your own. Remember, not to delete any artefact yourself using other means, always use **make clean** to clean up old artefacts.

21. Once your code is ready, build your own binaries **ftpserver** and **ftpservertest** typing **make** from your project root folder.

22. Type **make test-deliverable1** to test the **deliverable 1** using your own code.
23. Type **make test-deliverable2** to test the **deliverable 2** using your own code.
24. Type **make test-deliverable2** to test the **deliverable 3** using your own code.
25. Type **make test** to test both **deliverable 1, deliverable 2,** and **deliverable 3** together using your own code.

26. In order to see how your own **ftpserver** performs, you will need to run it as well as the example **ftpclient** in two separate terminals.
    a) Open two terminals and go into your project root directory in both terminals.
    b) In order to run your **ftpserver**, type following command in one terminal.

        *make run-server*

    c) In order to run example **ftpclient**, type following command in the other terminal.

        *make run-example-client*

    d) Type user commands in example FTP Client application to see how your server responds against your commands.

27. **Organize** your code by appropriate **indentation** and add adequate **file** and **function comments** on your code. Type **your name**, **student number**, and **section number** at the beginning of each **cpp** file that you have contributed to. You can follow coding style shown in '**ftp_server.cpp**'.

28. Make sure you have deleted all debugging print codes that you were using to debug your code during your development time but not necessary in the final code. Also, make sure you have deleted all commented out codes from your final submission. The code with poor organization and comments will also get poor evaluation.

29. Complete the **README** file. You need to give the general description of the application, technologies that are used in the application, how a user can build (compile and link) and install the application, how a user can run the application after the installation. Mention instructor's name and your name in the list of contributors. Give GPL license to the users to use the application. You can google to find README examples if you are not sure how to write one.

30. **Commit** and **push** your **final code** before the **deadlines**.

31. Commit and push the filled up **contribution form** if you have done your project in a partnership with another student.

32. Your submission will be evaluated to zero if it does not compile and run.

33. Your submission will be tested by running automated test as well as by running as an independent application for evaluation.

## Deadlines and Submissions

This project has **3 incremental deliverables**. The deadlines for the deliverables are as follows:

| **Project 2** | **S25N01** | | **S25N02** | |
|---|---|---|---|---|
| | **Demonstration** | **Code Submission** | **Demonstration** | **Code Submission** |
| **Deliverable 1** | In the lab on **March 03, 2025** | **11:00 PM** on **March 05, 2025** | In the lab on **March 05, 2025** | **11:00 PM** on **March 05, 2025** |
| **Deliverable 2** | In the lab on **March 17, 2025** | **11:00 PM** on **March 19, 2025** | In the lab on **March 19, 2025** | **11:00 PM** on **March 19, 2025** |
| **Deliverable 3** | In the lab on **April 07, 2025** | **11:00 PM** on **April 09, 2025** | In the lab on **April 09, 2025** | **11:00 PM** on **April 09, 2025** |

**Late completion/submission of any deliverable will be evaluated as zero.**

**Commit** and **push** your work from your **local** repository to your **remote** repository regularly. Use following **git commands** to commit and push your work from your local repository to your remote repository from your project's root folder:

*git add --all*

*git commit –am"A meaningful commit message that describes what is being committed"*

*git push origin master*

Remember '**git add**' command has **double dashes** before '**all**' and '**git commit**' command has single dash before '**am**'. You can also use '**git add .**' **dot** option instead of '**all'** option. Both options do the same, **add** all **the new** and **the modified files** into the **commit index**. If you want to add only **a specific file** into the commit index, you can specify **the relative path of the file** instead of dot or 'all' option. For example, if you want to add only **main.cpp** file of the **src** folder into the commit index, you can use '**git add src/main.cpp**' command. You can also include multiple files or paths separated by space in the same 'git add' command instead of using multiple 'git add' commands. Command 'git add' is necessary before each 'git commit' command. If you skip 'git add' command before a 'git commit' command, it does not perform the actual commit of the new and modified files into the repository. Always type a **meaningful message** in your '**git commit**' command's '-am' option. For example, if you are committing after adding all the necessary comments into your 'Makefile', use 'git commit –am"Completed adding Makefile comments"'. It is not recommended to make a huge commit instead of a number of small commits. It is also recommended to avoid unnecessary commits. Commits should reflect the check points of your software development milestones. Command '**git push**' is necessary to push the local commits to the remote repository. If you skip 'git push' after a 'git commit', your local and remote repository will become unsynchronized. You **must keep your local** and **remote repositories synchronized** with each other.

You will find most useful **git** commands in this **git cheat sheet** from GitLab. You will be allowed to commit and push until the deadline is over. Incremental and frequent commits and pushes are highly expected and recommended in all assignments.

## Evaluation

| Project2 | Test Cases | FTP Request | Marks | Subtotal |
|---|---|---|---|---|
| **Deliverable 1** | *ftp_server_net_util* | *closeSocket* | 1.0 | |
| | | *getIP* | 2.0 | |
| | | *getPort* | 2.0 | |
| | | *getRemoteIP* | 2.0 | |
| | | *getRemotePort* | 2.0 | |
| | | *isSocketReady* | 5.0 | |
| | *ftp_server_connection_listener* | *startListener* | 7.0 | |
| | | *isListenerReady* | 0.5 | |
| | | *acceptConnection* | 4.0 | |
| | | *closeListener* | 0.5 | |
| | *ftp_server_connection* | *sendTo* | 2.0 | |
| | | *receiveFrom* | 2.0 | **30.0** |
| **Deliverable 2** | *ftp_server_session* | startSession | 5.0 | |
| | | stopSession | 2.0 | |
| | *ftp_server_request* | parse | 1.0 | |
| | | interpret | 3.0 | |
| | | unsupported | 2.0 | |
| | | QUIT | 2.0 | |
| | | notLoggedIn | 2.0 | |
| | | USER | 2.0 | |
| | | PASS | 2.0 | |
| | | PWD | 2.0 | |
| | | isValidPath | 1.0 | |
| | | CWD | 2.0 | |
| | | CDUP | 4.0 | **30.0** |
| **Deliverable 3** | *ftp_server_passive* | passiveResponse | 2.0 | |
| | | startPassiveListener | 1.0 | |
| | | stopPassiveListener | 1.0 | |
| | | enteringIntoPassive | 8.0 | |
| | *ftp_server_nlst* | listDirEntries | 4.0 | |
| | *ftp_server_retrieve* | fileSize | 2.0 | |
| | | sendFile | 5.0 | |
| | *ftp_server_request* | PASV | 1.0 | |
| | | NLST | 2.0 | |
| | | SIZE | 2.0 | |
| | | RETR | 2.0 | **30.0** |
| **Code Quality, Comments, README** | | | 15.0 | **15.0** |
| **Total** | | | | **105** |