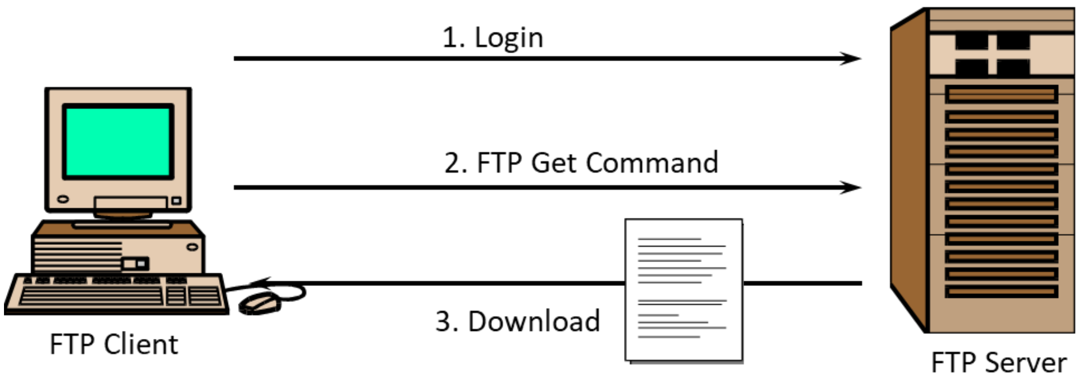
Project 1: File Transfer Protocol (FTP) Client Software Application

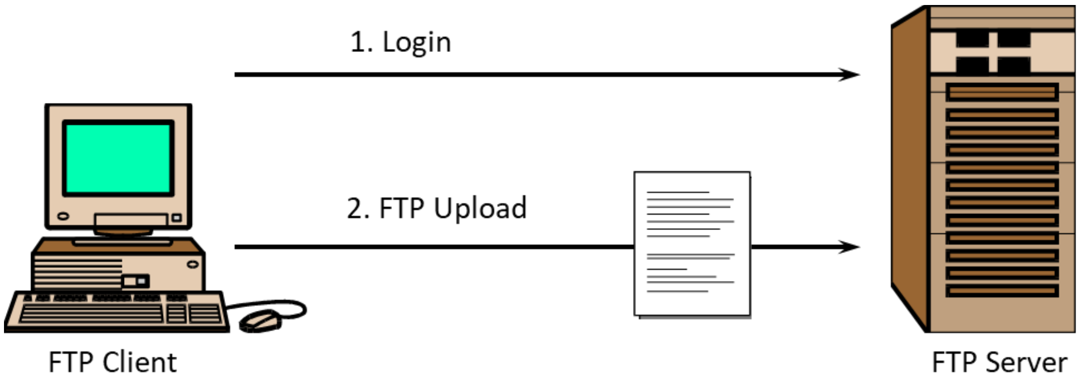
Objectives

1. To learn how a computer network protocol is documented in a 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 Client software according to Internet Standard outlined in RFC 959.
5. To implement FPT Client 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 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 Client software application in C++ programming language using Unix Network or Socket programming API.

FTP is a client-server protocol to transfer files to/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**
- **PASS**
- **PWD**
- **CWD**
- **CDUP**
- **PASV**
- **NLST**
- **RETR**
- **QUIT**

Your FTP Client software must have a command-line user interface (UI) so that the users can enter commands through this interface. User commands of your 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). Your software must accept and process following user commands.

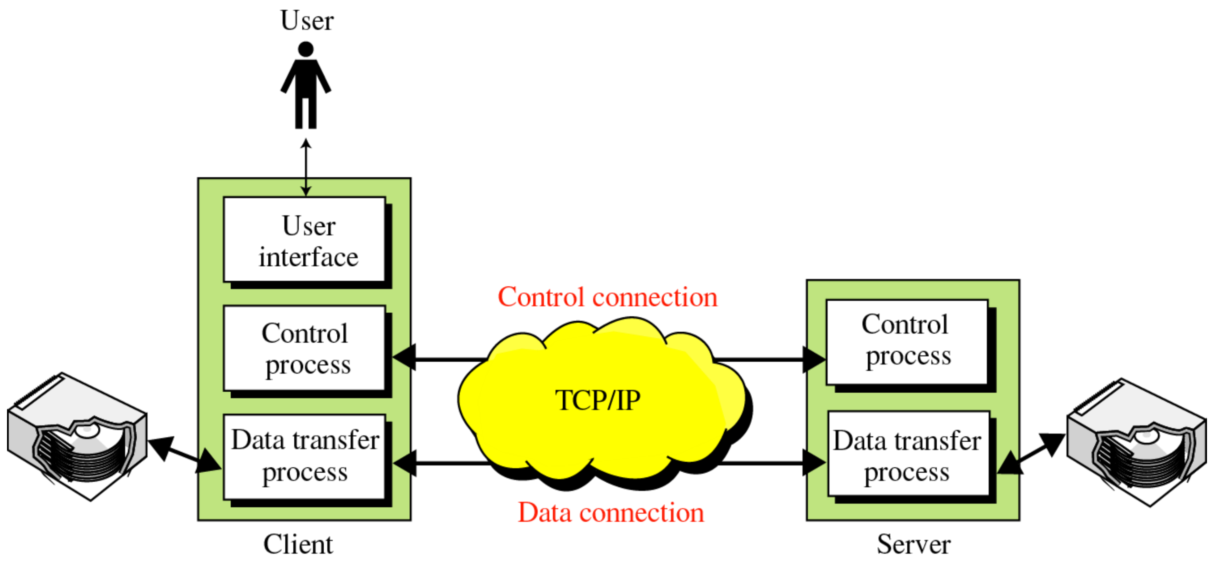
- **help**
- **user <username>**
- **pass <password>**
- **pwd**
- **dir**
- **cwd <dirname>**
- **cdup**
- **get <filename>**
- **quit**

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.

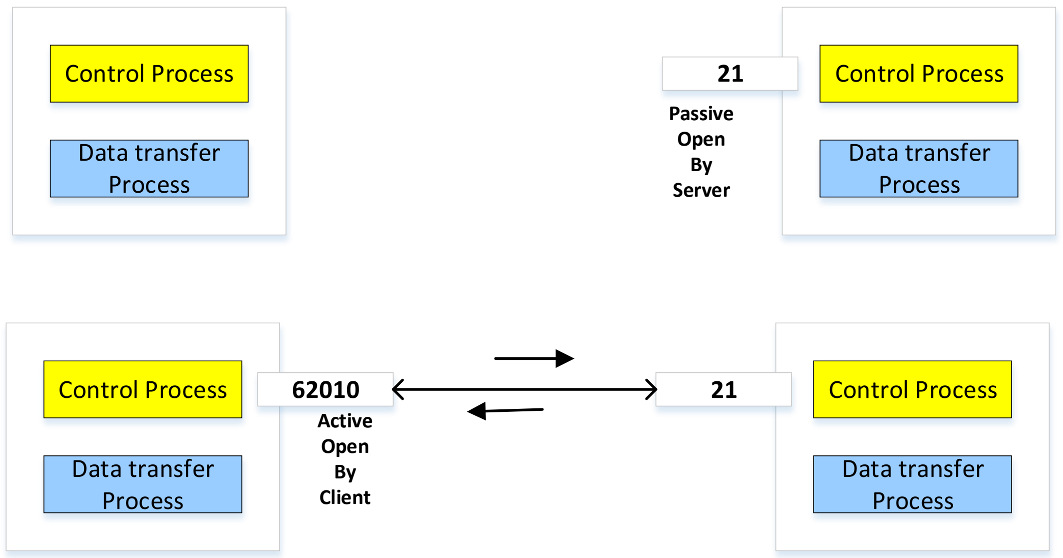
– Command '**help**' displays the list of commands supported by this software application, their syntax, and meaning.

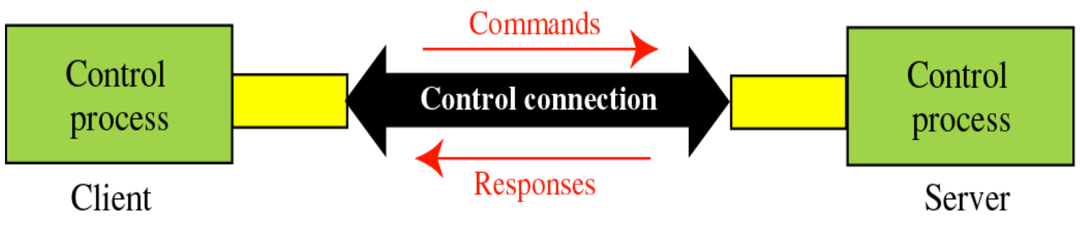
- Command ‘**user**’ sends username to the FTP server for authentication. You need to support only one user with user name ‘csci460’ and password ‘460pass’.
- Command ‘**pass**’ sends the password to the FTP server for authentication.
- Command ‘**pwd**’ prints the current working directory of the FTP server.
- Command ‘**dir**’ lists the contents of the current working directory of the FTP server.
- Command ‘**cwd**’ changes the current working directory to another directory specified in the argument. If the specified directory is beyond current working directory of FTP server, an error is reported by the server.
- Command ‘**cdup**’ changes the current working directory to its parent directory. If the parent directory is beyond server’s base directory, an error is reported by the server.
- Command ‘**get**’ fetches 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**’ informs 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.

Your FTP client software application must interpret above user commands, translate them into appropriate FTP request messages, send the FTP request messages to FTP server, receive the response messages from the server, and present the response to the user in a user-friendly manner.



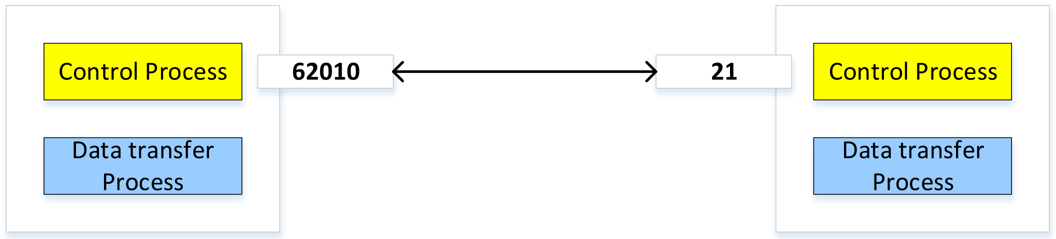
FTP Client and Server communicate request and response messages over a control connection. FTP Server passively wait for a control connection and FTP Client actively open the control connection.

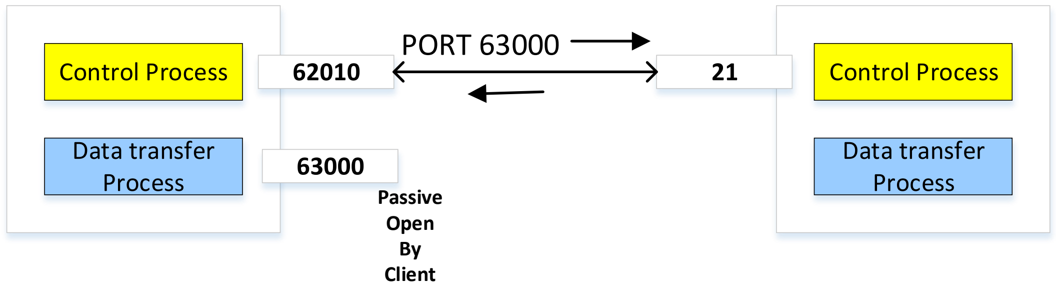


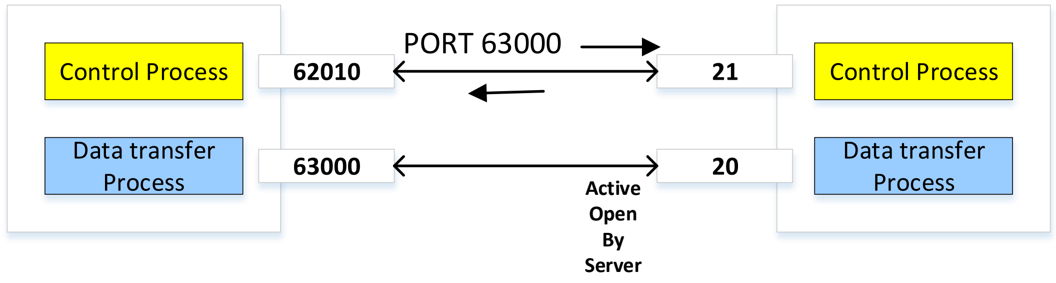


Some requests/responses involve a data transmission. For example, RETR request involves a data transmission to transfer file content as the part of a successful response. FTP server uses a separate connection for each data transmission. A data connection is opened on demand and closed when a data transmission is complete. FTP server can operate either in active or in passive mode to open a data connection.

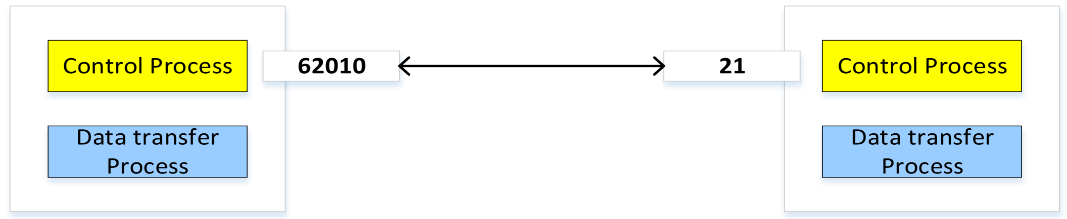
In active mode, FTP server opens the data connection with the client on demand. At first, FTP Client choose an ephemeral port and opens a connection listener on this port. FTP Client sends this port number to the server using PORT request message and wait for the server to open the connection. After receiving the port number, FTP server opens the data connection.

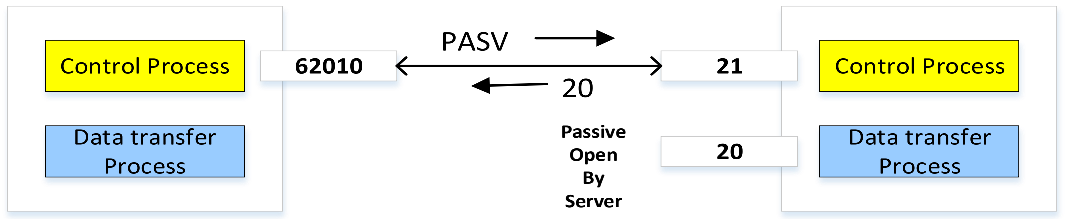


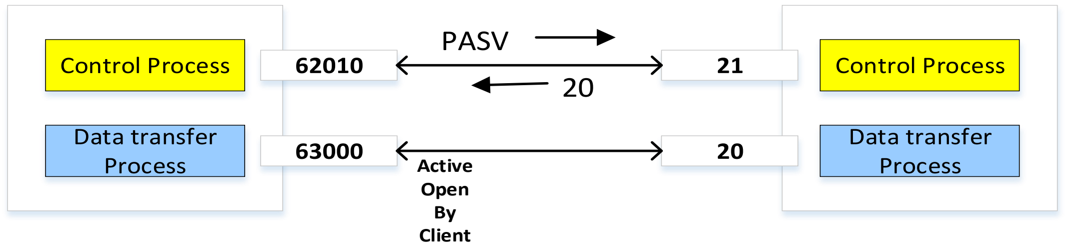




If the client is behind a firewall, FTP active mode fails to open a data connection. FTP passive mode has been proposed to solve this problem. In passive mode, when a data connection is required the server opens a connection listener so that client can send connection request to the listener and open 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 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 client software application, must send 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 **project1** has already been created for this project.

2. You are allowed to do this project alone or with another student. Your team has to be approved by your instructor before you proceed with your team work. If you are doing the project alone, create your own fork of **project1** on the central GIT repository using following command. You should skip this step if you are doing the project in a team or group.

   ssh csci fork csci460/project1 csci460/$USER/project1

If you are working in a team, a forked repository for your team has already been created. For example, the central repository for **team1** is **csci460/team1/project1**. In order to find out which team repository you have access to, type following command:

```
ssh csci info
```

3. Create a folder named **csci460** in your home folder and go into your **csci460** folder.

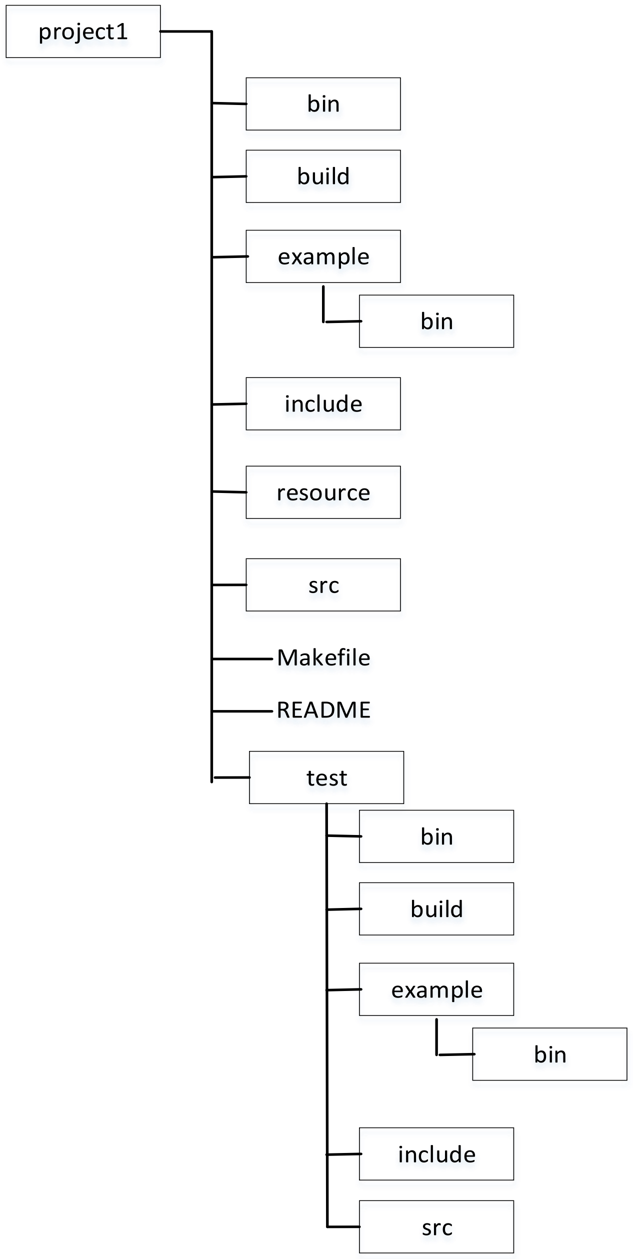
4. Create a clone of your forked **project1** repository using following command if you are working alone:

   ```
   git clone csci:csci460/$USER/project1
   ```

   Create a clone of your forked team **project1** repository using following command if you are working in a team (assuming you are working in team1, replace team1 with your own team):

   ```
   git clone csci:csci460/team1/project1
   ```

5. Repository **project1** has been organized as follows:



6. Continue your work in your cloned or local **project1** repository and **commit** and **push** your work to your central **project1** 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**. <u>Instructor, will take further steps to verify the integrity of your work if there is any red flag of academic misconduct</u>.
7. You will use supplied **Makefile** and Unix **make** utility to **build**, **run**, and **test** your application. You don't need to and should not modify this Makefile.

8. A **README** template file has been supplied. You will need to complete **README** file to guide your user about your application. Complete your README file once you have completed your project.

9. Following header (***.h)** files have been supplied in **include** folder in the repository.
   a. **ftp_client_command.h**
   b. **ftp_client_connection.h**
   c. **ftp_client_session.h**
   d. **ftp_client_ui.h**
   e. **ftp_server_response.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_client_command.h**' should be implemented in '**ftp_client_command.cpp**' file. You are not allowed to change any header file. If you modify any header file, your project will be evaluated to zero.

10. All of your source code (**cpp**) files should be in **src** folder of the project. The source code of **ftp_client.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_client_command.cpp**
   b. **ftp_client_connection.cpp**
   c. **ftp_client_session.cpp**
   d. **ftp_client_ui.cpp**

11. The lists of internal dependencies of the **cpp** files in this project are as follows.
   i. ***ftp_client.cpp:*** *ftp_client_session.h, ftp_client_ui.h, ftp_client_command.h*
   ii. ***ftp_client_ui.cpp:*** *ftp_client_ui.h*
   iii. ***ftp_client_connection.cpp:*** *ftp_client_connection.h*
   iv. ***ftp_client_sesion.cpp:*** *ftp_client_session.h, ftp_client_connection.h, ftp_client_command.h, ftp_client_ui.h, ftp_server_response.h*
   v. ***ftp_client_command.cpp:*** *ftp_client_command.h, ftp_client_ui.h, ftp_client_connection.h, ftp_client_session.h, ftp_server_response.h*

12. As mentioned earlier, you will **build** your FTP Client 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 ('**ftpclient**') 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 (**ftpclienttest**) in **test/bin** folder.

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

14. Test code file named **ftp_client_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 file (**ftp_client_test.o** and **ftp_client_test_net_util.o**) from your instructor's test code have been supplied in **test/build** folder and build tool **make** will use this object file and other object files from your source code to create your test binary (**ftpclienttest**) in **test/bin** folder. Make clean command will not delete this test object file and don't delete them yourself either.

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

    make test-example-deliverable2

17. To see how the example **ftpclienttest** works for **deliverable1** and **deliverable2** together enter following from your project root folder and observe console outputs.

    make test-example

18. In order to see how the example **ftpclient** 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.

19. 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.

20. Once your code is ready, build your own binaries **ftpclient** and **ftpclienttest** typing **make** from the project root folder.

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

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

       make run-example-server

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

       make run-client

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

25. **Organize your code nicely** and **add adequate 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_client.cpp**'. 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.

26. 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.

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

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

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

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

Partnership

You are allowed to complete this project with another student. In that case, your contributions must be marked in your source code. If you are the only contributor of a function, you should mark yourself as the sole author of that function before the function header. If you are the only contributor of all the functions in a file, you should mark yourself as the sole author of the file at the beginning of the file. If both you and your partner have contributed to write a function or all the functions of a file, both of you should mark yourselves as the authors at appropriate place in the code. You and your partner must complete, sign, and submit **contribution form**.

Deadlines and Submissions

This project has **2 incremental deliverables**. The deadlines for the deliverables are as follows:

Project 1	**S24N01**		**S24N02**	
	Demonstration	**Code Submission**	**Demonstration**	**Code Submission**
Deliverable 1	In the lab on **January 22, 2024**	**11:00 PM** on **January 24, 2024**	In the lab on **January 24, 2024**	**11:00 PM** on **January 24, 2024**
Deliverable 2	In the lab on **February 05, 2024**	**11:00 PM** on **February 07, 2024**	In the lab on **February 07, 2024**	**11:00 PM** on **February 07, 2024**

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 --all**' has double dashes before 'all'. 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 **prog.cpp** file of the src folder into the commit index, you can use '**git add src/prog.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 **'-m**' option. For example, if you are committing after adding all the necessary comments into your 'Makefile', use '**git commit –m"Added Makefile comments**"'. Remember there is a single dash before m in 'git commit' command. 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 commit 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 using '**git push**' after each '**git commit'**.

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 this assignment.

Evaluation

Project1	**Test Cases**	**Commands**	**Marks**	
Deliverable 1	*ftp_client_ui*		2	
	ftp_client_connection		20	
	ftp_client_session		5	
	ftp_client_command	*interpret*	3.0	
		help	2.5	
		simple	5	
		user	2.5	
		pass	2.5	
		quit	2.5	
				45
Deliverable 2	*ftp_client_command*	*pwd*	2.5	
		cwd	2.5	
		cdup	2.5	
		dir	15	
		get	15	
README			5	
Code Quality and Comments			12.5	55
Total				**100**