Implementation of a FTP server and client in Python

Sailen Nair 1078491
William Becerra Gonzalez 789146

***Abstract*—This report details the design and implementation of a FTP server and client in Python. The server and client both support the minimal required commands, and multiple more, as specified in the RFC 959 document. A basic graphical user interface was created for the client using the PyQt5 framework. The server and client were both tested independently with standard FTP applications such as FileZilla, thus allowed for independent development. The server implemented a range of response codes, as well as basic error handling. The server and client were able to communicate and transfer files between each other successfully. The server caters for a multi-threaded approach, with each client having their own user name and password. The server is able to log each users transactions between the client and server in a log file. The overall FTP application worked as expected, thus the final application is deemed a success.**

I. Introduction

In the information age and the rise of IoT, it has become increasingly important to share information between computer devices. The File Transfer Protocol (FTP) is a standard network application layer protocol that facilitates the transfer of files between computer devices [1]. The FTP specifications were first published in 1971 by Abhay Bhushan and modified until the latest standards were published in 1985 [2]. FTP provides a simple way to transfer files between devices. Users can log in using plain text, however if security is a concern user log in details can be protected using TLS/SSL [3].

This project aims to detail the implementation of a FTP server/client application with a simple client graphical user interface and an FTP server. There are good FTP applications available and therefore the project is about the FTP protocol implementation rather than providing an alternative to the already existing application. The project requires knowledge in socket programming and basic understanding of network protocols. The implementation was created using only pythons socket library and PyQt5 for the client graphical user interface.

Section II details the specifications of the project, assumptions, constraints as well as the success criteria for the project. The design and implementation of the client, and server are shown in Section III. The various commands and response codes that the FTP application caters for is detailed in Section IV. Section V contains the results of the implemented FTP application, with Section VI showing how to use the code. The critical analysis of the project as well as the future improvements are showcased in Section VII and Section VIII respectively. Finally Section IX details how the work load of the project was split between the two members of the group.

II. Background

The File Transfer Protocol is designed on a client-server architecture with two TCP connections, namely a command connection and data connection. The command connection is persistent were as the data connection is non-persistent. This means that one data connection is opened for a single data transfer and closed after the transfer is completed. The command connection is created when there is a request on server port 21, which is the standard FTP port, and it is closed after the user sends the 'QUIT' command or after some time of inactivity. The FTP client/server standard is described in the Request for Comments 959 (RFC 959) document [1]. The standard describes the minimal functionality and the default values for any FTP implementation. The default transfer type is ASCII non-print and default transfer mode is stream. All FTP servers must accept the defaults and provide support for at least the following commands: QUIT, USER, TYPE, MODE, STRU, RETR, STOR.

A. Project Specifications

An FTP server and client application is required. The server and client must be able to communicate with each other as well as with standard FTP applications, such as FileZilla Client and FileZilla Server. The implemented server needs to be multi-threaded, i.e. the server should be able to host multiple clients at the same time. The client application must have a command line interface as well as a graphical user interface (GUI). Further details can be found in the project brief [4].

B. Assumptions

The FTP application will support all minimal implementation requirements and some extra features assuming, any client will access the FTP server using the standard commands in the RFC 959. The client application assumes that the client wishes to access the server in passive mode. This assumption prevents possible firewall issues that can arise with active mode and does not limit the user experience. However both the server and client do cater for active mode but on start-up, they are both set to passive as default. The command line interface assumes the user has some knowledge of FTP as it is required to enter FTP commands. The client GUI application assumes very basic knowledge on FTP and no knowledge of the commands is required.

C. Constraints

The File Transfer Protocol is relatively old, and the official documentation is very abstract making FTP applications hard to implement for someone who has limited knowledge in networking. The project brief restricts the use of FTP libraries that abstract the implementation. The implementation can only make use of low level libraries that deal with the network transport layer.

D. Success Criteria

The final project will be deemed a success if the FTP server is able to communicate with both standard FTP clients as well as the client created for the project. Furthermore, the server should be able to handle multiple clients. The client application will be considered successful if it can communicate with a standard FTP server performing the minimal commands needed as specified by RFC 959.

E. Existing Solutions

FTP is a relatively old transfer protocol and therefore there are many applications that offer end user products. One example is FileZilla, a open source FTP client as well as server. Most web browsers support the FTP protocol making FTP accessible through URL.

III. Design and Implementation

A. Client

The client application was designed using the commands stated in the RFC 959 document, a single function was implemented per command. The design has an object-oriented approach creating a TCP class that handles the transport layer by setting up sockets for communication. The TCP class was created to keep the transport layer abstract as the project is more concerned with the application layer. The TCP class is initialized with the server address and the port number (port 21). The class has two main functions $transmit()$ and $receive()$. The transmit function takes in the data to be transmitted as a string. The $receive()$ function returns a string of what has been sent from the server to the client. The TCP class uses 'utf-8' encoding by default. The 'FTPClient' class is the implementation of an FTP Client where the specific command to be sent to the FTP server can be called by a function of the same name. This is with the exception for USER and PASS commands that are sent by the login function. For example, to send the RETR command, the function $retr()$ sends the command and handles the server response to that command. It is the same for all the implemented commands.

The client application uses passive mode by default to perform all FTP commands. This design choice prevents possible firewall issues [6]. For example, if the user is behind a NAT (Network Address Translation) router, incoming connections from outside the network may be blocked. Similarly a client's computer firewall might block incoming connections. The client GUI does not provide active mode capabilities however, the command line interface does provide support for active mode. The GUI gives users basic functionality and passive mode can provide all basic functionality while avoiding firewall issues that can hinder the user experience.

The CLI client application has a state machine design where each command that can be sent to the server is defined as a state. The user is prompted to input a command and the respective state is chosen by a series of if statements. Once the task associated with the state has been completed, the current state is changed to idle were the user is prompted again. This process iterates and the only way to exit the state machine is to select the quit command. Unsupported commands are ignored by the design and the user is prompted again for a command.

B. Server

The server application implements the minimum required commands as stated in RFC 959, as well as multiple others. The full list of commands supported by the server is detailed in Section IV-A below. Much like the client implementation, the server design has an object-orientated approach. The server makes use of two TCP classes, one for the main server socket, which will constantly listen for any clients wanting to gain access to the server. The second TCP class is for the client connection, it is used to create and control the command connection between a client and the server. The main function within the 'TCP_Server' class is the $acceptConnection()$ function, this function will accept any client that wants to access the server. The 'TCP_Client' class is similar to the TCP class mentioned above, where the two main functions are $transmit()$ and $receive()$ respectively. Where $transmit()$ encodes a message and sends it to the client and $receive()$ decodes and returns the message sent from the client.

The commands received from the client, are decomposed into two variables, the first being the actual command and the second being the command attribute. If the command does not require an attribute, then the second variable will be left as 'None'. All the commands that are accepted by the server are encapsulated in their own separate function, much like the client implementation. Hence when the server receives a command from the client, the function pertaining to that command will be executed. The appropriate function is executed by using the 'getattr()' function within the standard Python library. This function takes in two parameters, joins the two parameters and returns the named attribute of the object [7]. For example when the 'LIST' command is received from the client it is parsed into the function like 'getattr(self,LIST)', the output is the command 'self.LIST()'. All the functions are named after the command they represent. If the command has not been implemented by the server, the server will send an appropriate response to the client indicating that the command is not supported by the server.

The server is implemented according to RFC 959, hence all the commands received by the client are then responded to by use of standard reply codes. These reply codes are

specific to the command. There are multiple reply codes for each command, depending on the success or failure of the command. Section IV-B will detail the reply codes implemented by the server and to which commands each reply code is associated with.

The server makes use of two high level libraries, namely 'os' and 'logging'. The 'os' library is used to interact with the file system of the operating system, such that directories can be accessed and new directories can be made. Through the use of 'os' the server is able to run on multiple operating systems. The 'logging' library is used to log all the commands received by the server as well as all the responses sent from the server. For each client, a separate log file is created, and all the clients commands and servers responses will be stored, as well as the time and date of each command/response. This ensures a full history of the client interactions with the server is stored.

As mentioned in Section II-A the server needed to be multi-threaded. This was done by use of the 'threading' library. When a client tries to access the server through the server TCP socket, a new thread is created for that specific client. The client will then proceed to interact with the server as normal. The server can support multiple clients, all of which will be able to interact with the server concurrently. When a thread is created, a welcome message is sent from the server to the client, after which the client will login and proceed to interact with the server.

Section III-A details why the passive mode of data transfer is preferred over the active mode. However to satisfy the minimum requirements as listed in RFC 959, the 'PORT' command needed to be implemented. The 'PORT' command indicates that the mode of transfer has been set to active, hence the server will request the TCP connection and not the client as is the case with passive mode. Hence the server can operate under both passive mode as well as active mode. However as default, the mode for data transfer is set to passive.

IV. Commands and Response Codes

This section will detail the commands implemented by both the FTP server and client, as well as the reply codes implemented by the server.

A. Supported Commands

Below details the commands implemented by both the FTP server and client, for brevity only the minimal commands stated in RFC 959 are listed below. However extra commands have also been implemented. The extra commands implemented along with a brief description are shown in Table II in the appendix.

The RFC 959 document provides a detailed description on the function of each FTP command. Therefore the supported commands will be explained briefly in the context of this project.

1) USER: This command is used to send the user the user name of the client that is trying to access the FTP server. The server responds with a '331' code indicating the user name is correct and a password is required. The server can also respond with an error code 5xy[1] indicating the command could not be performed. The USER command is sent by the $login()$ function which handles the login procedure within the client implementation.

2) PASS: This command is used to send the server the password of the user defined by the USER command. The server responds using the command connection with code '230' for a successful log in. The server can also respond with a 5xy command for commands that could not be performed. The PASS command is also sent by the $login()$ function which handles the login procedure within the client implementation.

3) RETR: The retrieve file command can be used to download data from the server to the client. The retrieve command enters the connection into active/passive mode and exits active/passive mode as soon as the file transfer is complete. The $retr()$ function is associated with 4 reply codes, namely '530' when the user is not logged in, '550' when the file that has been chosen to download does not exist, '150' when the data channel is opened for file download, and lastly '226' for the successful transfer of the file.

4) STOR: The Store command can be used to upload data to the server. The store command initiates a data connection in passive/active mode and the connection is terminated once the upload has been completed. The $stor()$ function is associated with 3 reply codes name '530' when the user is not logged in, '150' when the data channel has opened, and lastly '226' once the data has been successfully transferred.

5) NOOP: No Operation command is used to get a server OK reply with response code '200'.

6) STRU: The Structure command specifies the file structure. The FTP standard describes three different structures, file, record and page structure. The default structure is the file structure and is a minimum requirement for an FTP server implementation. The implemented server only supports the file structure however, the server is able to respond with an appropriate code for any mode requested. The server replies code '200' for file structure request, code '504' for a record or page structure request and '500' for an unknown structure request.

7) PORT: The Port command is used to tell the server which port the client is listening for the data connection. This represents an active data connection, and tells the server to initiate a data connection to a given port. A reply code of '200' will be relayed back to client indicating that data will

[1] In code 5xy the 5 indicates a permanent negative completion reply and xy is used to indicate the command to which the server responded to

be transferred in active mode.

8) TYPE: The Type command indicates how data will be transferred, by default the server is set to transfer data in 'ASCII' mode, however this is not suitable for media files, therefore the server also supports binary transmission. Three reply codes are associated with the TYPE command namely '200' to indicate that a mode has been set, '504' to indicate that the mode chosen is not support by the server and lastly '500' for an unknown type of mode.

9) MODE: The Mode command specifies the format the data is transferred. The FTP standard defines block mode and stream mode. Stream mode is the default and is a requirement for minimal implementation of an FTP server. The FTP server implemented only supports stream mode however, the server is able to respond with an appropriate code for any mode requested. The client mode() function sends the MODE command to the server. The server replies code '200' for a stream mode request, '504' for a block mode request and '500' for an unknown mode requested.

10) QUIT: The Quit command is used to end an FTP section by logging out the user. The client will receive the reply code '221' indicating that the user has logged out.

B. Reply Codes

A various number of reply codes were implemented within the server. Table III in appendix gives a detailed list of reply codes implemented as well as which command each reply code is associated with. The table in appendix also shows the various error codes that the server can support. The final section of Table III lists the response codes for the various errors that the server can handle, such as if the file name does not exist.

C. Features not implemented

All the features specified within the project brief has been met. Additional commands have also been implemented, these commands are discussed in Section IV-A.

V. Results

This section will detail the results obtained from the FTP server and client implementation. To ensure that both the server and client worked correctly and according to RFC 959, they needed to be tested independently of each other.

Thus to test the server, the FileZilla Client application was used. It as assumed that the FileZilla Client application is fully RFC 959 compliant, thus if the server communicates successfully with it, the server will also be deemed FTP compliant in accordance to RFC 959. Figure 1 in the appendix shows a screen-shot of the FileZilla Client interface interacting with the FTP server. This interaction between the server and the FileZilla Client was observed though Wireshark (a packet sniffing tool) and can be seen in Figure 2. This shows that the server can interact with a standard FTP client. The interaction was done in the passive mode of data transfer. However Figure 3 in appendix shows the server operating in an active data transfer mode.

Due to the many available standard FTP servers, including one situated in the school of Electrical and Information Engineering at the University of the Witwatersrand. The choice of FTP servers to test with were abundant. The main server of choice to test was 'ftp.mirror.ac.za'. Figure 4 in appendix shows a Wireshark screen-shot of the interaction between the client and 'ftp.mirror.ac.za' server. The screen-shot shown in Figure 4 shows a passive mode of data transfer. The client can handle active mode, but as stated above, due to firewall restrictions active mode could not be tested with an external server. The successful testing of the client with an external standard FTP server, proves that the client is FTP compliant.

With both the server and client interacting with standard FTP clients and servers respectively, the two were made to communicate. The first test between client and server was conducted on the same machine. Thus no external routing was required. The results of the interaction was captured through Wireshark and is shown in Figure 5. The entire interaction between client and server is shown , which includes a 'RETR' (download) command as well as 'STOR' (upload) command. It is also worth noting that the 'RETR' command is executed whilst under a passive mode of data transfer, whereas the 'STOR' command is executed under an active mode of data transfer. This shows that the client and server are both equipped to handle both modes of data transfer. Figure 6 to Figure 11 show the packets that were transmitted between the client and server. The commands as well as the reply codes are clearly seen. This verifies that both the client and server implemented comply with the standard FTP.

The client and server were succefully connected across two computers in the network. This shows that the application can be used across multiple devices. Figure 12 shows a Wireshark screen-shot of the transactions between the client and server over two different IP address. As can be seen the IP addresses are different, however they are within the same network.

The multi-threaded implementation of the server was tested successfully. 5 different clients, all with different login details managed to connect to the server and interact with the server concurrently.

Finally Section A in appendix shows an example of the log file that is created by the server. The server keeps a separate log file for each client that accesses the server. All the clients requests as well as the servers response are stored in the log file. Each entry is also time stamped.

VI. Structure of code and how to use

A. Client

The client consists of three classes, the TCP_Client class, the FTP_Client class and the main.py class. The TCP_Client class is responsible for handling the transport layer by controlling the TCP socket connection. The FTP_client.py file contains the FTP client implementation within the FTPClass and it makes use of python idiomatic __name__ == ”main” to run the CLI FTP application. The main.py class is responsible for the GUI implementation and it imports functionality from both FTP_Client and TCP_Client. To be able to run the main.py class the user requires PyQt5 installed in their system. The application is packaged into an executable called main.exe. The executable was created using the cx_Freeze tool available in [5]. The executable is convenient as it removes the need of a user needing to install Python 3 and the PyQt libraries.

The program can therefore be run in different ways depending on user preferences. To run the CLI application the python script FTP_Client.py needs to be executed. For users who prefer a graphical interface or whom do not have the necessary tools pre-installed in their system, the executable can be run by double clicking the executable. Figure 13 of the Appendix depicts basic instructions on how to use the client GUI.

B. Server

The server implementation consists of the FTP_Server.py script. This script makes use of the python standard library module ’os’, ’threading’, ’sys’, ’random’, ’logging’ and it imports the TCP_Server class. The FTP_Server.py defines the FTP_Server class which defines all the FTP functions. The FTP_Server.py also defines the application main loop and therefore running the script will set up an FTP server. The server.exe provides an executable to decrease the systems software requirement. All the commands that the server can handle are packaged into individual functions within the main class. The server does not require any input from the user. Once the server is running, it will automatically direct the user to the users folder and start logging all the transactions between the user and the server. Hence the server does all the work without any direct input from the user.

Both executables can be downloaded from [10].

VII. Critical Analysis

The results show that both the server and client applications are able to communicate with each other as well as with standard FTP client/server applications. The applications meet more functionality than the standard minimal requirements, described in section 5.1 of the RFC 959 document. As discussed in Section III. The application offers GUI and CLI options that are packaged into an executable for user convenience. The full range of commands described in the RFC 959 are not supported. However, the application is able to handle unknown/unsupported commands by passing an appropriate message to the client. This project is far from an end-user application for the File Transfer Protocol but it does provide details on FTP and how to run FTP server/client in the application layer. The overall application provides minimalistic error handling and can sometimes crash. A specific case which causes the application to crash is in start-up, if there is an application such as FileZilla already using port 21 the application attempts to bind to port 21 and an OS error cause the FTP server application to crash.

VIII. Possible Improvements

The application supports more functionality commands than the minimal implementation for an FTP server as seen in section 5.1 of RFC959. However there a few features that can be implemented such as support for different modes such as block mode and compression mode, type and structure. The GUI implemented was minimalistic and lacks a proper file structure presentation, it requires the user to read the server responses in the server response screen. In a modern world a GUI based application is essential for any program to provide a competitive product. Another key improvement that can be made is to add a layer of security to the application. Most FTP clients and servers support a TLS (Transport Layer Security) layer. The application created does not support any form of security and thus can be addressed in future developments of the application.A time-out feature could be implemented for both server and client. This helps to avoid possible crashes on the application waiting for a response from the server/client.

IX. Division of Tasks

Both group partners actively collaborated throughout the project. The task divisions can be seen in Table I. For the software collaboration the GitHub account of the either partner can be checked [8], [9]. As can be seen from Table I the tasks were evenly split. Developing the client separate to the server (and vice versa) was possible due to the availability of FTP applications such as FileZilla.

TABLE I
Division of tasks between

Task	Contributor
FTP client	Both
FTP server	Sailen
Client GUI	William
Wireshark	Both
Documentation	Both

X. Conclusion

The FTP server and client applications successfully allows the transfer of files between two different computers. The implementation was tested with a number of standard FTP applications. It was shown that the FileZilla applications work seamlessly with the implemented FTP applications. The implementation does not support every single command detailed in the RFC 959. However, it supports a large number of features that allow any user to enjoy the full experience of the File Transfer Protocol. The report has detailed the design and implementation of the FTP application, it has shown the various commands and reply codes supported, as well as the possible improvements of the application. The FTP application worked as expected, and the final solution is deemed a success.

REFERENCES

[1] J. Reynolds and J. Postel, FILE TRANSFER PROTOCOL (FTP) 1985, https://tools.ietf.org/html/rfc959, Last Accessed: 31 March 2018

[2] Joe McKendrick, ZDnet ,FTP, first created in 1971, alive and well in service-oriented world 2011, http://www.zdnet.com/article/ftp-first-created-in-1971-alive-and-well-in-service-oriented-world/, Last Accessed: 31 March 2018

[3] unknown, IBM, Steps for customizing the FTP client for TLS, https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.halz002/ftp_cust_client_for_tls.htm, Last Accessed: 31 March 2018

[4] Dr. Asad Mahmood, ELEN4017 Networks Fundamentals Course Project 2018, School of Electrical and Information Engineering, University of the Witwatersrand

[5] Anthony Tuininga, cx_Freeze,https://anthony-tuininga.github.io/cx_Freeze/index.html, Last Accessed: 31 March 2018

[6] David Geer, Securing risky network ports 2017,https://www.csoonline.com/article/3191531/network-security/securing-risky-network-ports.html, Last Accessed: 31 March 2018

[7] Python By Programiz, https://www.programiz.com/python-programming/methods/built-in/getattr Last Accessed 31 March 2018

[8] William Becerra Gonzalez https://github.com/93will93

[9] Sailen Nair https://github.com/sailennair

[10] https://www.dropbox.com/sh/j9fjo4x5jl14b89/AABlUFS0Feu6Csq2bC6MQilba?dl=0

APPENDIX

TABLE II
TABLE SHOWING THE EXTRA COMMANDS IMPLEMENTED ON THE CLIENT/SERVER FTP APPLICATION

Command	Description
LIST	This command requests the server to transmit a list of all the files in the current working directory. The request is done through the command connection, but the list is transmitted from server to client through the data connection.The $list()$ function sends the LIST command to the FTP server and handles the list information that is received by the server. When the data channel is opened, a '150' code is transmitted to the client, once the list has been transferred fully, a '226' code is relayed back to the client indicating that the list has successfully been sent from the server.
CWD	This function takes in the directory defined by the user and sends the server the CWD command as well as the directory the user wants to change to. If the operation is successful, the server responds with an OK '250' code, if the operation cannot be done, a '550' error code will be relayed back to the client.
CDUP	The change to parent directory command moves to the parent directory of the current working directory. The server replies the message code '250' to show that the requested file action was successful. The function $cdup()$ sends the CDUP command to the server. The server responds with a '250' OK code message if the change to parent directory is successful.
MKD	The Make Directory command can be used to make a directory on the server. The $mkd()$ function sends the MKD command to the server. This function takes in an argument that specifies the path on the server where the new subdirectory is going to be created. A reply code '257' indicates that the directory has been made, whereas a reply code '550' indicates that the file could not be created.
DELE	The Delete command can be used to delete a file in the server. The $dele()$ function sends the DELE command to the server. It takes in as an argument the path of the file to be deleted on the server. If the file was successfully deleted a reply code '250' is received by the client or '550' if the file chosen to be deleted does not exist.
PASV	The Passive command tells the server to listen to a port for the data connection. This initiates the connection before the server receives a transfer command. A reply code of '227' is received indicating the server is now in passive mode and is listening to a particular port.
PWD	The PWD command will print the current directory the client is in. A successful reply code of '257' is received along with the directory.
RMD	The Remove Directory command can be used to remove a directory from a given path. The associated reply codes indicate if the directory was removed successfully (code '250') and if the directory could not be removed a permanent negative reply code '5xy' is received.
RNFR	The Rename From command specifies a file path of a file that is to be rename. This command has to be immediately followed by the Rename To (RNTO) command. The server replies code '350' and waits for RNTO command to successfully rename the file. If the file cannot be renamed, a '550' response code will be sent to the client.
RNTO	The Rename To command specifies the new name for the file selected in the command RNFR. It is associated with two commands, namely a successful '250' which indicates the file has been renamed, and a '553' code which indicates that the new file name is not allowed.
SYST	The system command indicates to the client what platform the server is running on. A '215' code is relayed back to the client.
HELP	The help command will tell the client what command codes are supported by the server, this is associated with the '214' reply code.

TABLE III
TABLE SHOWING THE REPLY CODES IMPLEMENTED ON THE SERVER OF THE FTP APPLICATION

Reply code	Description	Commands that use it
	Positive Preliminary Reply	
150	Used to indicate that a data connection is about to be opened	LIST, RETR, STOR
	Positive Completion reply	
220	Welcome message, indicating the service is ready for a new user.	Welcome message
230	User is logged in.	PASS
257	The file name is created/returned	PWD, MKD, RMD
250	File action is okay and/or has been completed	CWD, DELE, RNTO
200	Command Okay	CDUP, NOOP, TYPE, MODE, STRU, PORT
226	Data connection is closing, with a successful data transfer	LIST, RETR, STOR
227	Entering passive mode.	PASV
215	The system type.	SYST
214	Help message on how to use the server.	HELP
221	Logging out.	QUIT
	Positive Intermediate reply	
331	User name is okay, awaiting password.	USER
350	Requested file action pending further information.	RNFR
	Transient Negative Completion reply	
425	The data connection cannot be opened	When attempting to open a data connection
	Permanent Negative Completion reply	
502	Command is not implemented by the server.	When a command is not recognized
501	Syntax error in parameters or arguments.	USER, PASS
530	User is not logged in.	PASS, DELE, LIST, RETR, STOR
550	File is not available	CWD, MKD, RMD, DELE, RETR, RNFR
504	Command is not implemented for that specific command.	TYPE, MODE, STRU
500	Syntax error, command is not recognized.	TYPE, MODE, STRU
553	File name is not allowed.	RNTO

LOG ENTRIES

```
2018-03-23 15:54:45,661 INFO sailen Command USER sailen
2018-03-23 15:54:45,662 INFO sailen Response USER 331 User name okay, need password
2018-03-23 15:54:45,662 INFO sailen Command PASS *****
2018-03-23 15:54:45,662 INFO sailen Response PASS 230 User Logged in
2018-03-23 15:54:49,622 INFO sailen Command PWD
2018-03-23 15:54:49,622 INFO sailen Response PWD 257 C:\Users\Sailen\Dropbox\2018
Electrical\Network Fundamentals\Project\code-repo-final\elen4017_project
2018-03-23 15:54:51,447 INFO sailen Command PASV
2018-03-23 15:54:51,447 INFO sailen Response PASV 227 Entering passive
mode(127,0,0,1,48,65)
2018-03-23 15:54:51,448 INFO sailen Command LIST
2018-03-23 15:54:51,448 INFO sailen Response LIST 150 list is here
2018-03-23 15:54:51,449 INFO sailen Response LIST 226 List is done transferring
2018-03-23 15:55:07,046 INFO sailen Command PASV
2018-03-23 15:55:07,047 INFO sailen Response PASV 227 Entering passive
mode(127,0,0,1,228,255)
2018-03-23 15:55:07,047 INFO sailen Command TYPE I
2018-03-23 15:55:07,047 INFO sailen Response TYPE 200 Binary mode set
2018-03-23 15:55:07,048 INFO sailen Command RETR rfc.pdf
2018-03-23 15:55:07,048 INFO sailen Response RETR 150 Opening data channel
2018-03-23 15:55:07,099 INFO sailen Response RETR 226 Transfer of rfc.pdf successful
2018-03-23 15:55:10,206 INFO sailen Command PASV
2018-03-23 15:55:10,206 INFO sailen Response PASV 227 Entering passive
mode(127,0,0,1,191,209)
2018-03-23 15:55:10,207 INFO sailen Command LIST
2018-03-23 15:55:10,207 INFO sailen Response LIST 150 list is here
2018-03-23 15:55:10,208 INFO sailen Response LIST 226 List is done transferring
2018-03-23 15:55:12,599 INFO sailen Command QUIT
2018-03-23 15:55:12,599 INFO sailen Response QUIT 221 User logged out
```

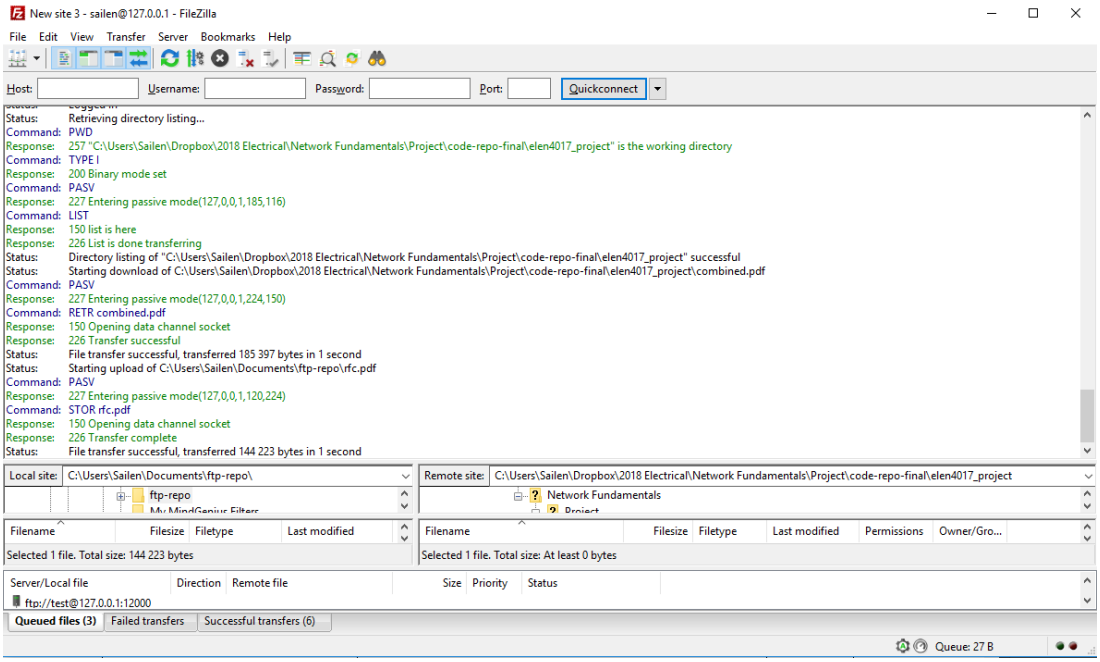


Fig. 1. The image above shows the FileZilla application interacting with the built FTP server in passive mode

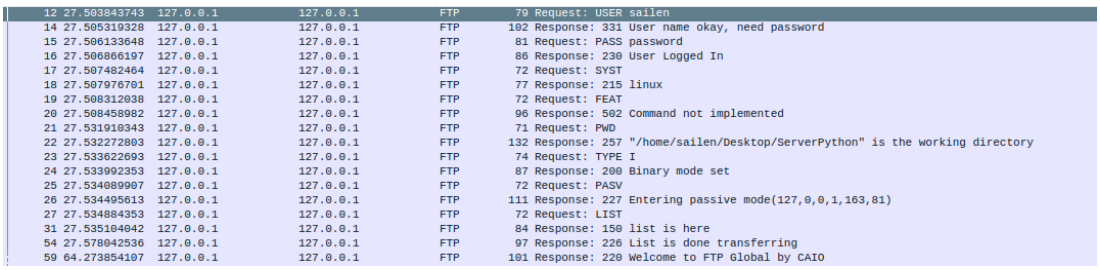


Fig. 2. The above image shows the packets transferred between the FileZilla client and the server. The packets were captured using Wireshark

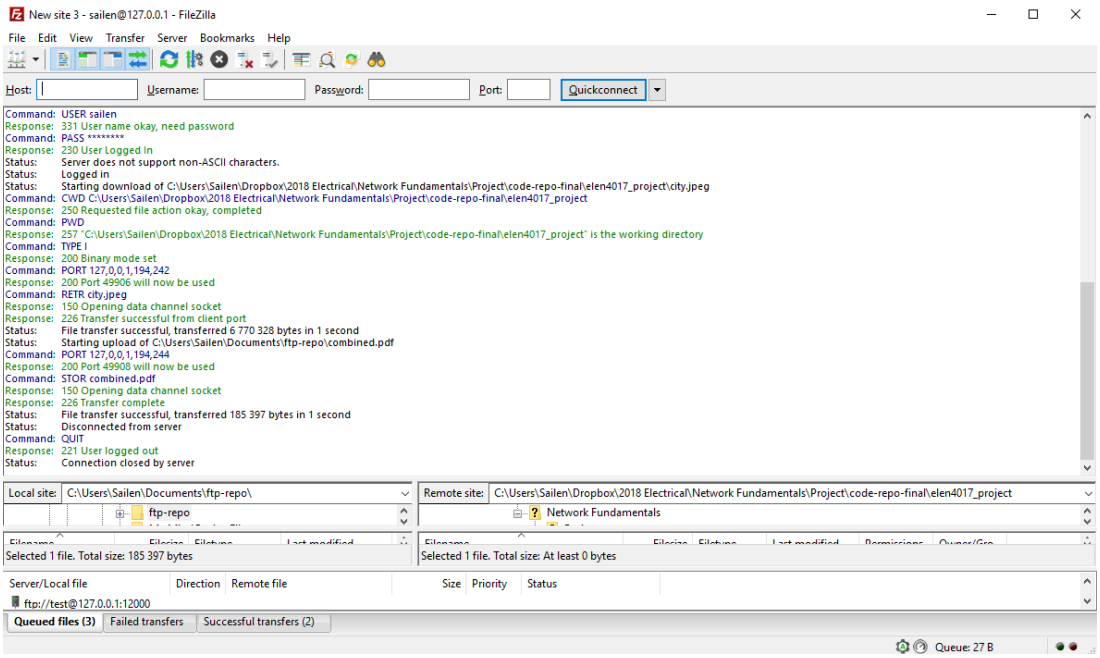


Fig. 3. The image above shows the FileZilla application interacting with the built FTP server in active mode

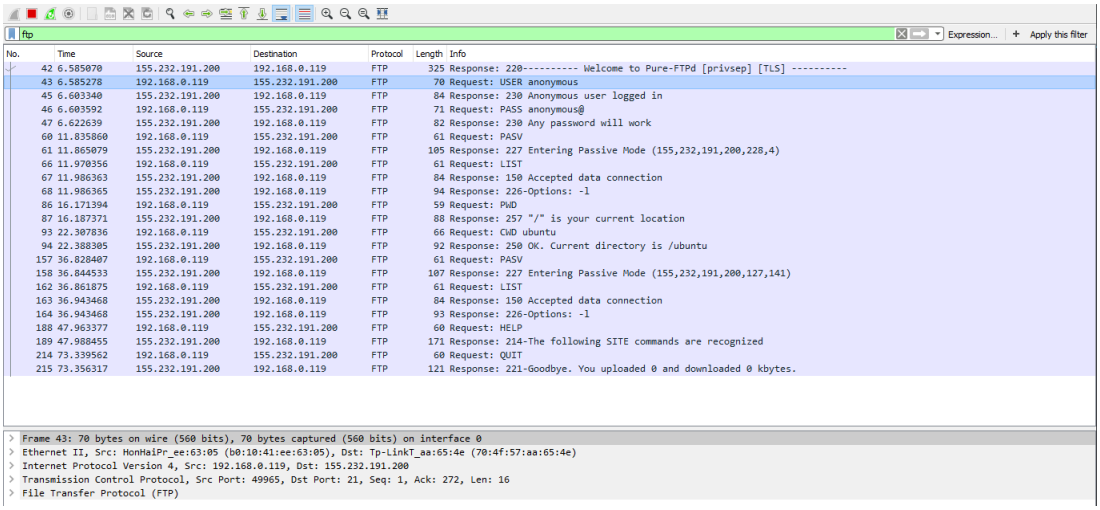
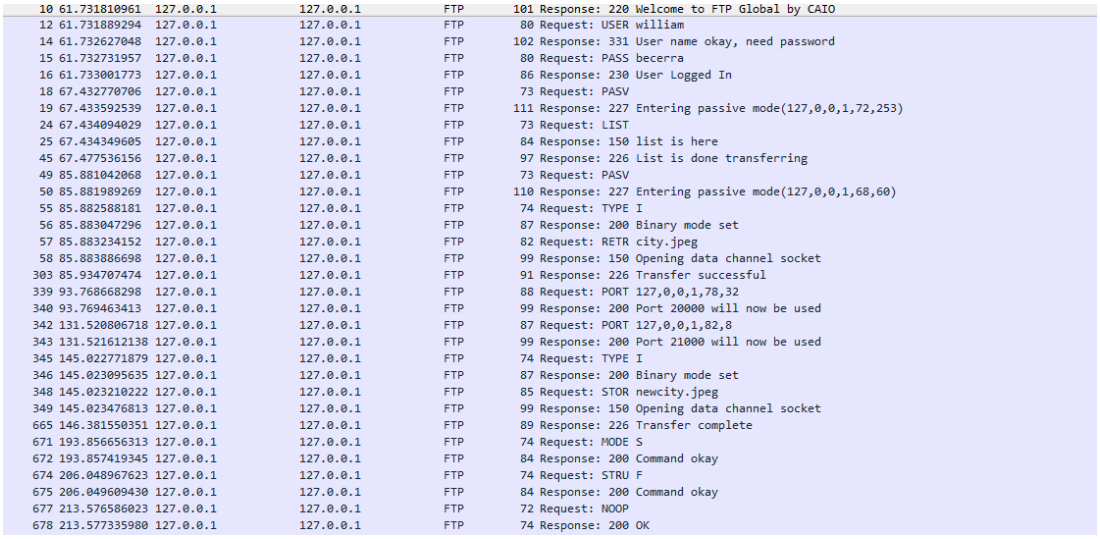


Fig. 4. This image shows the packets transferred between the built client and an outside FTP server, namely 'ftp.mirror.ac.za'.



10	61.731810961	127.0.0.1	127.0.0.1	FTP	101 Response: 220 Welcome to FTP Global by CAIO
12	61.731889294	127.0.0.1	127.0.0.1	FTP	80 Request: USER william
14	61.732627048	127.0.0.1	127.0.0.1	FTP	102 Response: 331 User name okay, need password
15	61.732731957	127.0.0.1	127.0.0.1	FTP	80 Request: PASS becerra
16	61.733001773	127.0.0.1	127.0.0.1	FTP	86 Response: 230 User Logged In
18	67.432770706	127.0.0.1	127.0.0.1	FTP	73 Request: PASV
19	67.433592539	127.0.0.1	127.0.0.1	FTP	111 Response: 227 Entering passive mode(127,0,0,1,72,253)
24	67.434094029	127.0.0.1	127.0.0.1	FTP	73 Request: LIST
25	67.434349605	127.0.0.1	127.0.0.1	FTP	84 Response: 150 list is here
45	67.477536156	127.0.0.1	127.0.0.1	FTP	97 Response: 226 List is done transferring
49	85.881042068	127.0.0.1	127.0.0.1	FTP	73 Request: PASV
50	85.881989269	127.0.0.1	127.0.0.1	FTP	110 Response: 227 Entering passive mode(127,0,0,1,68,60)
55	85.882588181	127.0.0.1	127.0.0.1	FTP	74 Request: TYPE I
56	85.883047296	127.0.0.1	127.0.0.1	FTP	87 Response: 200 Binary mode set
57	85.883234152	127.0.0.1	127.0.0.1	FTP	82 Request: RETR city.jpeg
58	85.883886698	127.0.0.1	127.0.0.1	FTP	99 Response: 150 Opening data channel socket
303	85.934707474	127.0.0.1	127.0.0.1	FTP	91 Response: 226 Transfer successful
339	93.768668298	127.0.0.1	127.0.0.1	FTP	88 Request: PORT 127,0,0,1,78,32
340	93.769463413	127.0.0.1	127.0.0.1	FTP	99 Response: 200 Port 20000 will now be used
342	131.520806718	127.0.0.1	127.0.0.1	FTP	87 Request: PORT 127,0,0,1,82,8
343	131.521612138	127.0.0.1	127.0.0.1	FTP	99 Response: 200 Port 21000 will now be used
345	145.022771879	127.0.0.1	127.0.0.1	FTP	74 Request: TYPE I
346	145.023095635	127.0.0.1	127.0.0.1	FTP	87 Response: 200 Binary mode set
348	145.023210222	127.0.0.1	127.0.0.1	FTP	85 Request: STOR newcity.jpeg
349	145.023476813	127.0.0.1	127.0.0.1	FTP	99 Response: 150 Opening data channel socket
665	146.381550351	127.0.0.1	127.0.0.1	FTP	89 Response: 226 Transfer complete
671	193.856656313	127.0.0.1	127.0.0.1	FTP	74 Request: MODE S
672	193.857419345	127.0.0.1	127.0.0.1	FTP	84 Response: 200 Command okay
674	206.048967623	127.0.0.1	127.0.0.1	FTP	74 Request: STRU F
675	206.049609430	127.0.0.1	127.0.0.1	FTP	84 Response: 200 Command okay
677	213.576586023	127.0.0.1	127.0.0.1	FTP	72 Request: NOOP
678	213.577335980	127.0.0.1	127.0.0.1	FTP	74 Response: 200 OK

Fig. 5. This image shows the packets transferred between the built client and the built FTP server.

```
     50 85.881989269   127.0.0.1             127.0.0.1             FTP      110    Response: 227 Enteri
passive mode(127,0,0,1,68,60)
Frame 50: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 21, Dst Port: 55384, Seq: 186, Ack: 50, Len: 44
    Source Port: 21
    Destination Port: 55384
    [Stream index: 0]
    [TCP Segment Len: 44]
    Sequence number: 186    (relative sequence number)
    [Next sequence number: 230    (relative sequence number)]
    Acknowledgment number: 50    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window size value: 342
    [Calculated window size: 43776]
    [Window size scaling factor: 128]
    Checksum: 0xfe54 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
    TCP payload (44 bytes)
File Transfer Protocol (FTP)
    227 Entering passive mode(127,0,0,1,68,60)\r\n
        Response code: Entering Passive Mode (227)
        Response arg: Entering passive mode(127,0,0,1,68,60)
        Passive IP address: 127.0.0.1
        Passive port: 17468
```

Fig. 6. The above image shows the packet sent from the server to the client when the 'PASV' command is sent.

```
    339 93.768668298   127.0.0.1            127.0.0.1            FTP      88     Request:
PORT 127,0,0,1,78,32
Frame 339: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 55384, Dst Port: 21, Seq: 74, Ack: 309, Len: 22
    Source Port: 55384
    Destination Port: 21
    [Stream index: 0]
    [TCP Segment Len: 22]
    Sequence number: 74    (relative sequence number)
    [Next sequence number: 96    (relative sequence number)]
    Acknowledgment number: 309    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window size value: 342
    [Calculated window size: 43776]
    [Window size scaling factor: 128]
    Checksum: 0xfe3e [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
    TCP payload (22 bytes)
File Transfer Protocol (FTP)
    PORT 127,0,0,1,78,32\r\n
```

Fig. 7. This figure shows the packet sent from the client to the server when the 'PORT' command is sent.

```
    343 131.521612138  127.0.0.1            127.0.0.1            FTP      99     Response: 200 Port
21000 will now be used
Frame 343: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 21, Dst Port: 55384, Seq: 342, Ack: 117, Len: 33
    Source Port: 21
    Destination Port: 55384
    [Stream index: 0]
    [TCP Segment Len: 33]
    Sequence number: 342    (relative sequence number)
    [Next sequence number: 375    (relative sequence number)]
    Acknowledgment number: 117    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window size value: 342
    [Calculated window size: 43776]
    [Window size scaling factor: 128]
    Checksum: 0xfe49 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
    TCP payload (33 bytes)
File Transfer Protocol (FTP)
    200 Port 21000 will now be used\r\n
```

Fig. 8. The above image shows the reply from the server indicating a successful 'PORT' command.

```
     57 85.883234152   127.0.0.1            127.0.0.1            FTP      82     Request: RETR city.j
Frame 57: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 55384, Dst Port: 21, Seq: 58, Ack: 251, Len: 16
    Source Port: 55384
    Destination Port: 21
    [Stream index: 0]
    [TCP Segment Len: 16]
    Sequence number: 58    (relative sequence number)
    [Next sequence number: 74    (relative sequence number)]
    Acknowledgment number: 251    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window size value: 342
    [Calculated window size: 43776]
    [Window size scaling factor: 128]
    Checksum: 0xfe38 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
    TCP payload (16 bytes)
File Transfer Protocol (FTP)
    RETR city.jpeg\r\n
        Request command: RETR
        Request arg: city.jpeg
```

Fig. 9. The above image shows the client sending the 'RETR' command to the server, the file to download is seen in the command header.

```
     58 85.883886698   127.0.0.1            127.0.0.1            FTP      99     Response: 150 Openin
data channel socket
Frame 58: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 21, Dst Port: 55384, Seq: 251, Ack: 74, Len: 33
    Source Port: 21
    Destination Port: 55384
    [Stream index: 0]
    [TCP Segment Len: 33]
    Sequence number: 251    (relative sequence number)
    [Next sequence number: 284    (relative sequence number)]
    Acknowledgment number: 74    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window size value: 342
    [Calculated window size: 43776]
    [Window size scaling factor: 128]
    Checksum: 0xfe49 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
    TCP payload (33 bytes)
File Transfer Protocol (FTP)
    150 Opening data channel socket\r\n
        Response code: File status okay; about to open data connection (150)
        Response arg: Opening data channel socket
```

Fig. 10. The first reply associated with 'RETR' from the server is indicated in the above image.

```
    303 85.934707474   127.0.0.1            127.0.0.1            FTP      91     Response: 226 Transf
successful
Frame 303: 91 bytes on wire (728 bits), 91 bytes captured (728 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 21, Dst Port: 55384, Seq: 284, Ack: 74, Len: 25
    Source Port: 21
    Destination Port: 55384
    [Stream index: 0]
    [TCP Segment Len: 25]
    Sequence number: 284    (relative sequence number)
    [Next sequence number: 309    (relative sequence number)]
    Acknowledgment number: 74    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window size value: 342
    [Calculated window size: 43776]
    [Window size scaling factor: 128]
    Checksum: 0xfe41 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
    TCP payload (25 bytes)
File Transfer Protocol (FTP)
    226 Transfer successful\r\n
        Response code: Closing data connection (226)
        Response arg: Transfer successful
```

Fig. 11. The second reply associated with 'RETR' from the server is indicated in the above image.

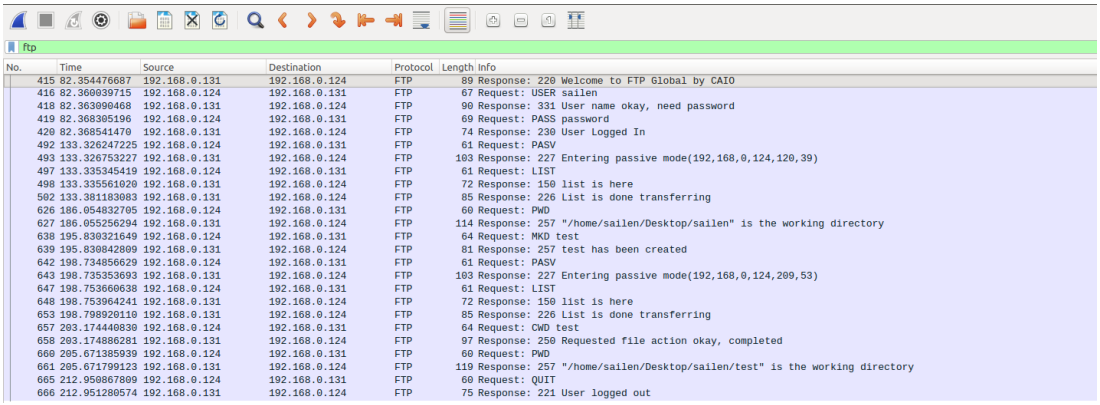


Fig. 12. The above image shows a wireshark screen-shot of the FTP client and server operating across two computers over the same network.

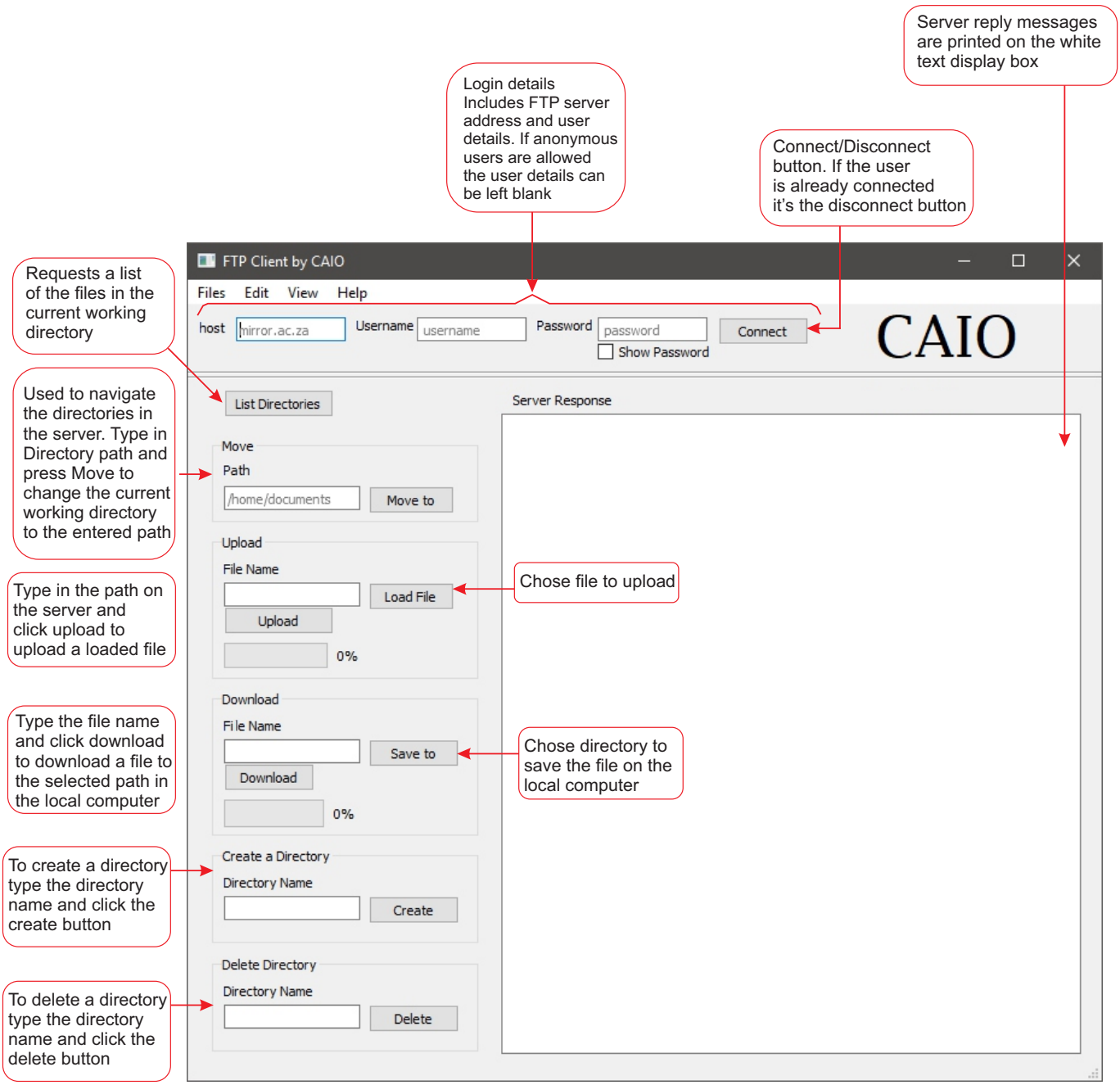


Fig. 13. The above image details the user interface and the functionality of each component