FTP Client Server Application

ELEN4017: Networks Fundamentals

Author: Diana Munyaradzi (1034382)

Abstract

The design, implementation and analysis of a file transfer application is presented. The final solutions consists of a client and a server which communicate through the transfer protocol implemented using the Python3 socket library. The client part of the application includes a GUI implemented using Qt Designer and Python. The application was designed and implemented on Ubuntu. The implemented algorithm implements multithreading that allows multiple clients to connect to one server. The project successfully implements a design that allows an FTP client to download and upload files as well as create and delete a directory on an FTP server. The server was also tested with FileZilla. It was concluded that the designed application implements the required basic functionality. However, multithreading can be implemented to enhance the transfer of files.

I. Introduction

Devices in a computer network communicate sequentially using various network protocols and standards, which are defined within the layers of the OSI Model. These devices can effectively send and receive data from each other through the use of the File Transfer Protocol (FTP), which falls within the Application layer and is built on a client-server architecture. A protocol is a set of rules that governs the communication between these devices [1]. Table 1 below shows examples of protocols found found in different OSI layers.

TABLE I: Examples of Protocols

Physical layer	Ethernet, WiFi
Network layer	IPv4, IPv6
Transport layer	TCP, UDP
Application layer	HTTP, FTP, SMTP, DNS

This paper presents the implementation of FTP using the basic Python socket methods and the protocols and standards of the network, transport and physical layers of the OSI model. Section II and III discuss the background and the algorithm designs for the FTP Client and Server, respectively. The results are presented and analysed in Section IV and the attached powerpoint presentation.

II. Background

FTP, a fundamental building block of data retrieval, is an RFC959 standardized file sharing protocol that allows the client user to exchange files with the local and remote storage installed on the server machine [1]. In addition to giving the user access to the files on the server (remote host), FTP also allows the client (local host) to perform operations such as renaming and deleting files as well as creating and deleting directories through a set of recognized commands. Figure 1 below shows an overview of the FTP client-server structure [2].

FTP uses 2 TCP connections which run in parallel, the data and control connection, to transmit data in segments known as packets. The control connection is based on the Telnet Protocol and it handles the communication between

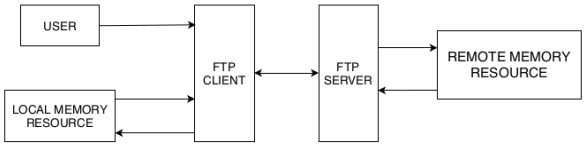

Fig. 1: FTP Client-Server Structure

the client and the server [3]. It is also responsible for sending commands, replies and control information such as the client's user identification and password. The data connection is used to transfer files in a specific mode and type[4]. The two types of modes are passive and active while the two types of transfer are binary and ASCII.

The client and server communicate sequentially in a request-messaging pattern where the client sends a request and the server responds. Once the server has been initiated, the client initiates a TCP connection using the server's IP address and port number [5]. Following the TCP connection, the server authenticates the user by requesting a username and the respective password. Figure 2 below shows a brief overview of how the TCP connection is established between the client and the server. Once the client has successfully logged in, the transfer file can begin. The following is the format used by client and server to communicate:

- Server: <Command Parameter Telnet Character >
- Client: : <Command Parameter Telnet Character >

The benefits of using FTP is that it allows both small and large types of files or folders to be transferred and it ensures that data is never lost. It also allows a transfer to be resumed in case an interruption occurs. However, since both the username and password are shared in plain text, it is susceptible to attacks because a hacker can access the credentials with little effort [2].

III. Design and Implementation

This project was implemented with the following functionality using Python 3:

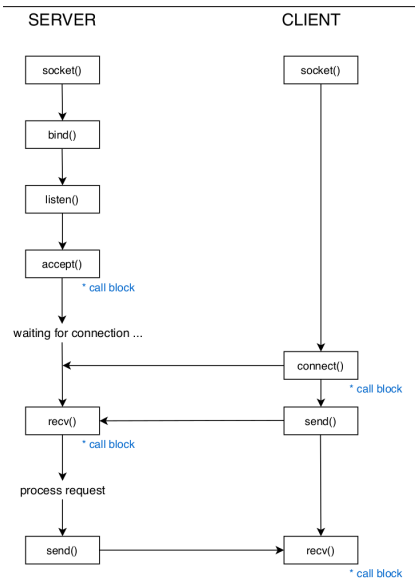

Fig. 2: Overview of TCP connection

- FTP server basic functionality
- FTP client basic functionality
- Client user interface
- Ability to deal with multiple files
- Implementation of multithreading
- Interaction between FTP client and server
- Ability to deal with different files

The application is developed using basic sockets taking into consideration the RFC959 standards. The program was designed and tested using Ubuntu. Although the memo has an extensive list of commmands, due to time constrains, this paper only focuses on a limited number of commands, which are shown in Table II along with their respective response codes. The implemented application consists of a simple client user interface, implemented using the IDE PyQt5, and the logic programs for both the client and the server.

A. FTP Server

The server hosts the files which can only be accessed via the client. Multithreading was implemented to allow a maximum of 5 clients to connect at a time, with each thread handling each TCP connection initiated by a client. The client thread runs in parallel with the main thread, which creates the socket commands, generates client threats and terminates the server. If the server accepts the handshake initiated by the client, the client is welcomed to the server with the 200 status which indicates that there is a live Protocol Interpreter (PI) connection between the server and the client. The PI connection exists in the `Main` function in the `serverThread` class. Figure 3 shows an overview of how the serverThread class functions.

Table 2 : Implemented Commands and Reply Codes

Command	Description	Success Reply Code	Error Reply Code
SYST	Describes the server's operating system	215	500
USER	Used by the user to specify their username	331	500, 530
PASS	Used by the user to specify their password	230	500, 530
QUIT	Closes the connection pipeline	221	500
STRU	Specifies the file structure	200	500, 504
MODE	Used to specify the data transfer mode	200	500, 504
NOOP	Asks the server to send OK	200	500
TYPE	Used to specify how data is represented	200	500, 501
PWD	Prints the current working directory	257	500
CWD	Changes the current working directory	250	500, 530, 550
PASV	Used to request a data connection	227, 225	425, 500, 530
PORT	Used to request a data connection	200, 225	425, 500, 530
LIST	Used to request a directory list	150, 200	500, 530
MKD	Creates a directory on the server	257	500, 530
STOR	Uploads files onto the server	150, 226	500, 530
RETR	Downloads files from the sever	150, 226	500, 530, 550

The `serverThread` class sends an error message if it receives an unrecognized command but if it receives a recognized one, it triggers the corresponding function. To protect the client's data, all the commands except for STRU, MODE, QUIT and SYS, use the `Username()` and `Password()` functions to confirm that the client is logged in first before executing the command. `Username()` checks if the entered username is registered in the database and if it is, `Password()` is triggered to check if the entered password matches the one registered for the user. If the user details the client enters are validated successfully, the client is given full access to the files on the remote server.

A Data Transfer Protocol (DTP) connection is used to facilitate the transfer of files between the remote server. If a client requests a DTP connection, it will be a passive connection and the serverThread class triggers the PASV() function after receiving the PASV command. After validating a client's log in status, the PASV() function generates a host name and a port number and sends them back to the client according to the RFC959 standard format shown below: Client: PASV
Server: 227 Passive Mode (A1, A2, A3, A4, a1, a2)
The hostname IP address is stored in A1-A4 and the PASV() function uses the Python split() function to extract all the components separately using "." as a delimiter character. The port number is computer as is shown in equations 1 and 2 below:

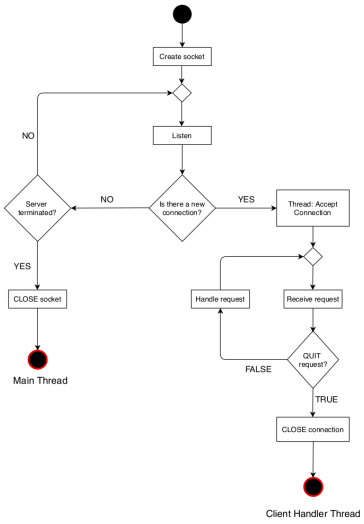

Fig. 3: Overview of `serverThread` functionality

$$a1 = PortNumber/256 \quad (1)$$

$$a2 = 256\%PortNumber \quad (2)$$

An active connection is established when the server initiates a data transfer. The `PORT()` function is triggered to generate and send the server's host name and port number using the format below:

Client: PORT A1, A2, A3, A4, a1, a2
Server: 227 OK

Once a DTP connection has been established, the client (or the server in the case of an active connection) either downloads a file, requests the directory list or stores a file using the RETR, LIST and STOR commands respectively. The commands trigger their respective functions and once the command has been executed, the connection is automatically closed and to repeat the process again, the connection has to be established. The application was designed to have the passive DTP connection as the default mode of transfer.

B. FTP Client

The client was designed with a logic layer, implemented using the FTPClient class and the user interface, implemented using the clientInterface class and the IDE Qt Designer. The interface was designed to mimic the FileZilla interface and it can be seen in Figure 1 of the Appendix. The interface was implemented because it makes it easier for the client to use compared to a user entering commands manually using the terminal. Once the interface is running, the user is required to enter their username, password and the server IP address and the port number. The two layers interact as shown by Figure 4 below.

When the login button is clicked the **loginButtonClicked** function in the `clientInterface` class is triggered. The function calls the `initialConnection()` and `login()` functions which from the FTPClient class. `initialConnection()` requests a TCP connection with the server using the IP addresses and the port number provided by the user and `login()` sends the username and password to the server once the TCP connection has been established.

The client interface runs an infinite while loop where it constantly polls to see if a button has been clicked. If a button has been clicked, the appropriate event is triggered via the client logic class FTPClient. When an event is triggered, the corresponding command is sent to the server. Figure 5 shows part of the client code showing the logic deployed to download a file.

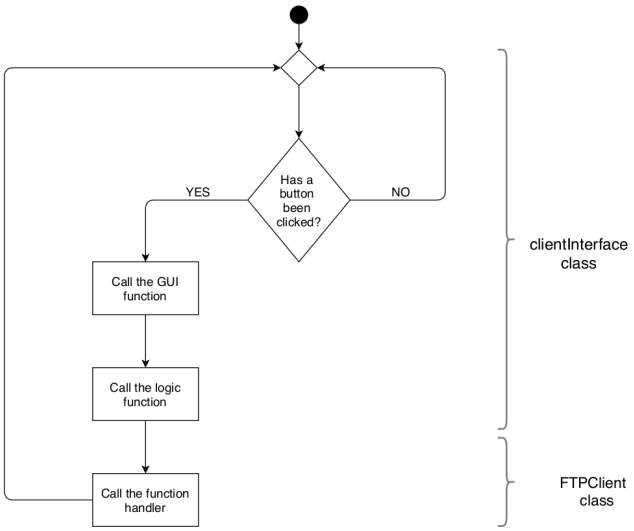

Fig. 4: Overview of the relationship between the Client GUI and Logic layers

```
def downloadFile(self,fileName):

    cmd = 'RETR ' +  fileName
    self.send(cmd) # Send download command
    self.printServerReply(self.getServerReply()) # Print the response from the server

    if not self.errorResp:

        # Create Downloads folder if not exist
        downloadFolder = 'Downloads'
        if not os.path.exists(downloadFolder):
            os.makedirs(downloadFolder)

        # Mode of data transfer
        if self.mode == 'I':
            outfile = open(downloadFolder + '/' + fileName, 'wb')
        else:
            outfile = open(downloadFolder + '/' + fileName, 'w')

        # Receive the data packets
        print('Receiving data...')
        self.statusMSG = 'Receiving data...'

        while True:
            data = self.DTPsocket.recv(8192)
            if not data:
                break
            outfile.write(data)
        outfile.close()

        print('Transfer Succesful')
        self.statusMSG = 'Transfer Successfull'
        self.DTPsocket.close()
        self.printServerReply(self.getServerReply())
```

Fig. 5: downloadFile() function from the Client Code

IV. Results and Analysis

The server was tested using both the FTP Client and FileZilla for the following functionality:

- Logging in using correct credentials
- Listing of the directory
- File upload
- File download
- Creating a new directory

The program encountered an authentication issue during the handshake stage between FileZilla and the FTP Client. The status response can be seen in Figure 3 in the Appendix. The FileZilla client sends the AUTH SSL command to the server to determine if the Secured Socket Layer is supported. To prevent the client from being stuck on the login page, the program was amended such that the server can send a response message to notify the client that SSL is not supported. A similar response message was included in the program to notify the client that ASCII is not supported. Therefore, the file transfer between FileZilla and the implemented server will only allow files to be read and written to in binary.

Another error in the implemented logic is that the server does not keep track of the users that are logged in. This causes a logical problem where the same user can log in using multiple client terminals at the same time. Although not tested, this might cause a race condition if one client terminal is attempting to delete a file that is being transferred by the other. Since the existence of the different client terminals is controlled through the use of multithreading, a mutex lock maybe be implemented to prevent the race condition as well.

The client GUI freezes when the client attempts to download or upload incredibly large files but FIleZilla completed the transfer successfully. This occurs the main thread will be handling the transfer of data whilst still polling for other client users. This could possibly be fixed by implementing threads in the client code to handle the upload and download of data in addition to threads being used to handle the clients.

The program achieves the basic functionality listed above successfully and this can be seen in the figures below that show screens of the GUI when different commands are called. One of the strengths of this program is that the program can be scaled easily because of the way the code was designed. Instead of having one big function that all the commands are in, the code has multiple classes and functions which makes it reusable and easily scaleable. One of the biggest drawbacks from the design is that a HELP command was not implemented. Adding such a command can improve the user experience that clients have with the program. The Appendix shows the screenshots when FileZilla is used as the client and the client and server responses as seen from the terminal.

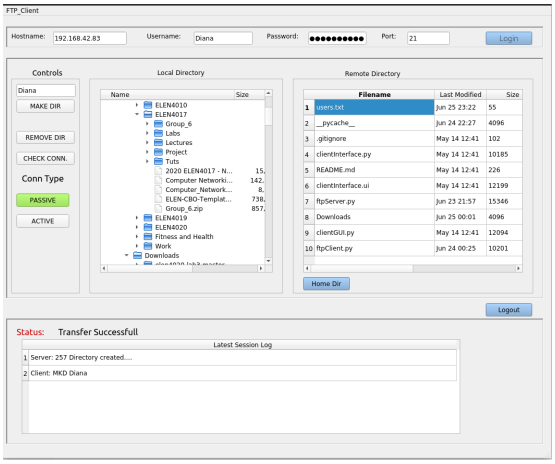

Fig. 6: Creating a new directory

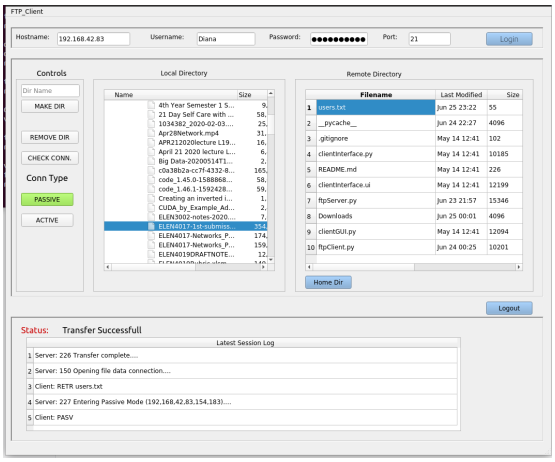

Fig. 7: Download Files

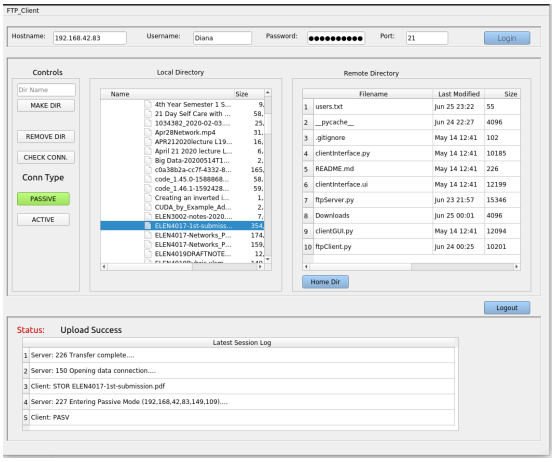

Fig. 8: Upload Files

V. Conclusion

An FTP application with a server and a client was successfully designed and implemented using basic python 3 socket methodology. The application included a very

useful client GUI that made navigating the client easier for clients with very little experience using terminals. The implemented solution implemented threads to manage the number of clients that had access to the server. The server was tested with the designed client as well as FileZilla. The application successfully implemented the basic transfer of files as well as the managing of directories. The final application could still be improved by using multithreading to enhance the upload and download of files. A better understanding of the FTP protocol was gained when working on this project. In conclusion, the basic functionality of the project was implemented successfully although more features could have been added.

References

[1] D. R. Winkelman, *Protocol*, Jun. 2020. [Online]. Available: http://fcit.usf.edu/network/chap2/chap2.htm.

[2] V. Glass, *Understanding key differences between sftp, ftps and ftp*, Jan. 2012. [Online]. Available: :%20http://www.jscape.com/blog/bid/75602/understanding-keydifferences-between-ftp-ftps-and-sftp.

[3] J. Kurose and K. Ross, *Computer networking, a top-down approach*, 2012.

[4] Nordea, *Ftp connectiona globalfile transfer protocol*, Apr. 2014. [Online]. Available: %E2 % 80 % 9D % 20https://www.nordea.com/Images/6-47406/File-transfer-FTP_fact_sheet..

[5] W. I. H. This? *File transfer protocol (ftp): Why this old protocol still matters*, Jan. 2020. [Online]. Available: https : / / www . whoishostingthis . com / resources/ftp/.

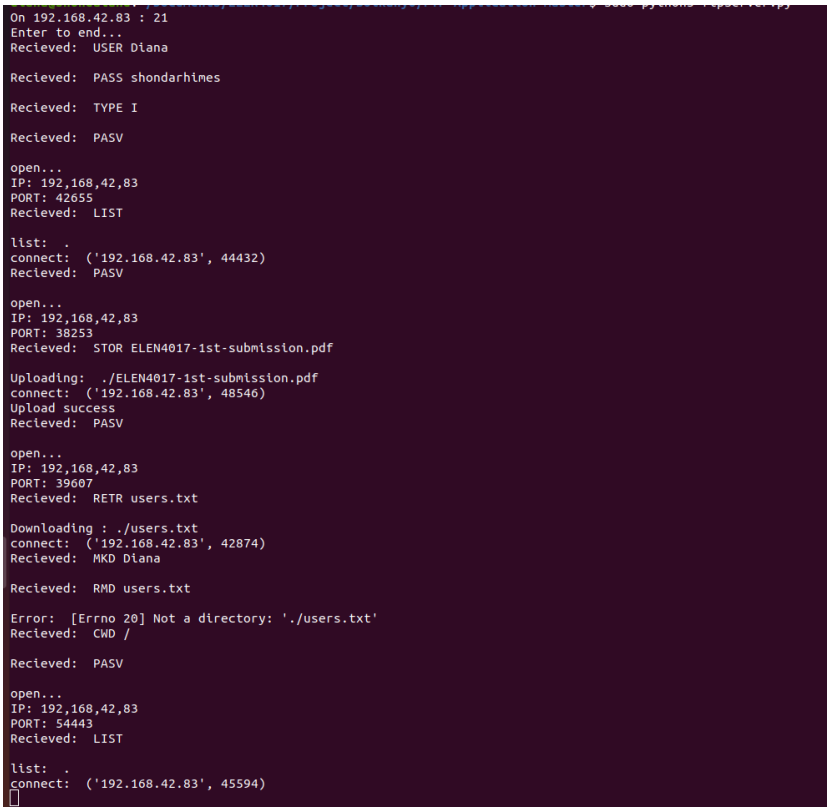

Fig. 9: Server Terminal

```
[sudo] password for diana:
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
220 Welcome!

Connected to Server ;)
Client:  USER Diana
Server : 331 User name okay, need password.

Server : 230 User logged in, proceed.

Login Success

Client:  TYPE I
Server : 200 Binary mode.

Client:  PASV
Server : 227 Entering Passive Mode (192,168,42,83,166,159).

192.168.42.83 42655
Passive Connection Success, Ready to receive

Client:  LIST
Server : 150 File status okay; about to open data connection.


Receiving Data...

-rw-rw-r--      1 user   group          55       Jun 25 23:22   users.txt

drwxrwxr-x      1 user   group          4096     Jun 24 22:27   __pycache__

-rw-rw-r--      1 user   group          102      May 14 12:41   .gitignore

-rw-rw-r--      1 user   group          10185    May 14 12:41   clientInterface.py

-rw-rw-r--      1 user   group          226      May 14 12:41   README.md

-rw-rw-r--      1 user   group          12199    May 14 12:41   clientInterface.ui

-rw-rw-r--      1 user   group          15346    Jun 23 21:57   ftpServer.py

drwxr-xr-x      1 user   group          4096     Jun 25 00:01   Downloads

-rw-rw-r--      1 user   group          12094    May 14 12:41   clientGUI.py

-rw-rw-r--      1 user   group          10201    Jun 24 00:25   ftpClient.py


Directory Listing Done!

Server : 200 Listing completed.

Dir..............
```

Fig. 10: Client Terminal Snippet

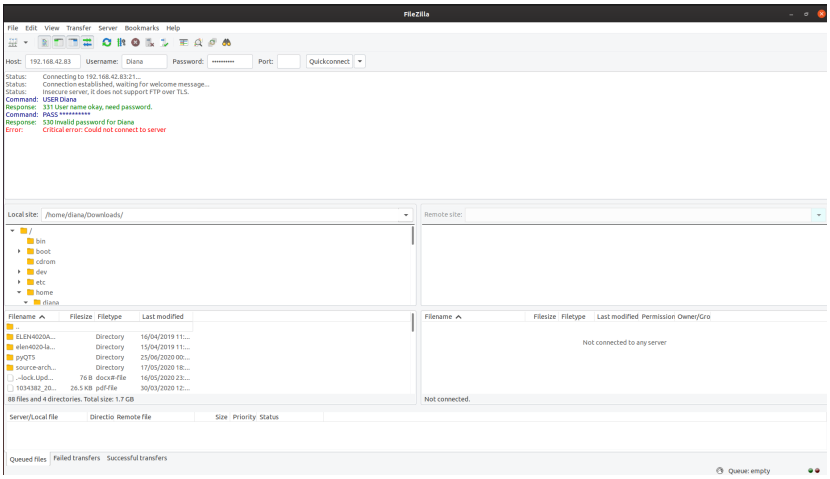

Fig. 11: FileZilla Client with incorrect log in details

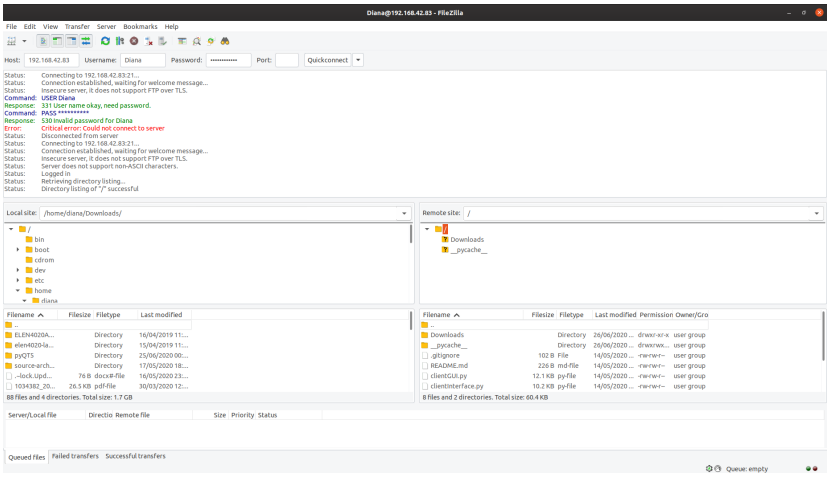

Fig. 12: FileZilla Client with correct log in details

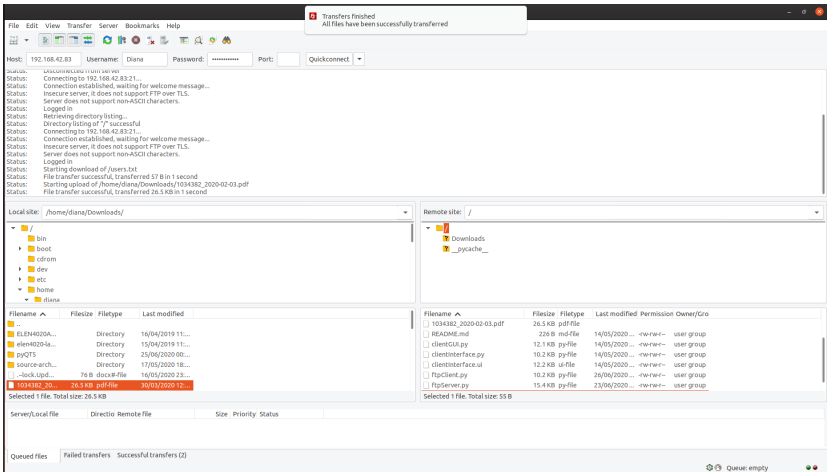

Fig. 13: FileZilla Client File Upload