DESIGN AND IMPLEMENTATION OF A FILE TRANSFER APPLICATION

Gerald K. Kirui (1615002)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: The purpose of this document is to report on the design and implementation of a file transfer application. The app allows a user to login, logout, upload a file, download a file, check the connection, query server OS, handle directories, delete a file and to list files in a directory. Nineteen FTP commands were implemented to support these features. The app was designed according to the standard FTP model. The app works successfully with FileZilla, and Wireshark can sniff packets transmitted by the app. The app uses multithreading to handle more than one user. However, the app is not secure because the passwords are sent as plaintext through the connection, and the UI is not easy to use. For future improvements, secure shell (SSH) authentication will be used to secure the password, and PyQt5 will be used to create a graphical user interface (GUI).

Key words: Client, File Transfer Protocol, Python, Server.

1. INTRODUCTION

The purpose of this report is to design and to implement a file transfer application which is based on the File Transfer Protocol (FTP). The scope of this document includes a discussion of the features implemented in the application, as well as the features which were omitted. The commands and the response messages used in the application will be outlined. Moreover, the structure of the code will be explained, and an analysis of the results will be carried out. Lastly, recommendations will be made to improve the application.

2. BASIC FEATURES

2.1. Login and logout

The application allows the user to login. To login, the user must use the command `login`, followed by the username. For increased security, the application requires a password – this feature will be explained in the next section (see Section 3.1).

The application allows the user to logout. To do so, the user should use the command `quit`. This feature terminates all connections between the server and the local site.

2.2. Upload files, download files and check connection

The application allows the user to upload files from the local site to the server. To do so, the user must first login to the local site. Next, the user must use the command `upload` followed by the filename and the extension. The application also allows the user to download files from server to the local site. To do so, the user must login to the app. Thereafter, the user must use the command `download` followed by the filename and the extension. The application allows the user to verify that the connection between the local site and the server is still active. To do this, the user must use the command `ping`.

3. EXTRA FEATURES

3.1. Security

For increased security, the application will only allow a user to login if the user provides the correct password after entering a valid username.

3.2. Query server OS

It is useful to know the operating system of the server. To query the server OS, the user must use the command `os`.

3.3. Handle directories

The user may find it useful to know which folder he/she is working in at any point in time. To know the current working directory, the user must use the command `this_dir`. The user may require to switch between existing folders to preserve the order in which files are stored. To change the working directory, the user must use the command `change_dir` followed by an existing pathname. Returning to the parent directory may be useful when a user has many folders. To return to the parent directory, the user must use the command `parent_dir`. It may be useful to create a folder directly from the application. To do so, the user must use the command `make_dir` followed by a folder name. If a user no longer requires a folder, it may be useful to delete the folder directly in the application. To do so, the user must use the command `remove_dir` followed by the folder to be deleted.

3.4. Delete a file

Apart from uploading and downloading files, a user may require to permanently delete a file from the server. To do so, the user must use the command `delete` followed by the file to be deleted.

3.5. List files on the server

Since functionality for handling directories was implemented, the user may end up having numerous folders, which makes it difficult to keep track of which files are placed in the different folders. To solve this issue, functionality was implemented to allow the user to see a list of all the files in the current working directory. To see the list, the user must use the command `list`.

4. FEATURES NOT IMPLEMENTED

4.1. Access control features

The account feature (ACCT) was not implemented. This feature allows a user to have more than one account on the server. The author did not implement this feature because the author deemed it sufficient for each user to have a single account on the server.

The structure mount (SMNT) feature was not implemented. This feature allows the user to mount a different structure without changing login status. The author did not implement this feature because the application is designed to operate only under the default structure (file) for the sake of limiting the scope of the application.

The reinitialization feature (REIN) was not implemented. This feature allows a user to return the application to its state before the action of logging in. Thus, different user information can be input. The reason for not implementing this feature is that one user usually uses one local site in a practical scenario, therefore there will be no need to reinitialise the application for the sake of a different user.

4.2. Transfer parameter features

The TYPE format EBCDIC in Carriage Control (ASA) form was not implemented. For minimum FTP implementation, the server must operate in ASCII Non-print format. Moreover, the server was designed for personal computers (particularly UNIX), therefore the author deemed it unnecessary to provide functionality for EBCDIC encoding style, which is used primarily in IBM systems [1].

The record structure and the page structure of STRU were not implemented. For minimum implementation, FTP does not require a record structure. By implementing the default file structure, the author limited the scope of the FTP application.

The author did not implement the block mode or compressed mode (which are types of mode) of file transfer for the FTP application, because such features add additional complexity with little benefit. Since the application is tested using small files, features such as compressed file transfer mode is not necessary.

4.3. Service features

The author did not implement storing a file in the server under a unique name (STOU) because the author did not see any advantage for functionality to name the same file using a different name in the server. In fact, the author considered that STOU may lead to problems because a user may confuse the name of a file on the local site and the server.

The append feature (APPE) was not implemented because the user can simply delete the file at the server site and upload an updated version with the appended information.

The feature of allocating storage at the server before an upload (ALLO) was not implemented because the application was designed under the assumption that a few users would be clients of the server. As such, there is no risk of the server reaching storage capacity.

The feature of restarting a download (REST) at the point of failure was not implemented. Since the size of the files to be shared on the FTP application is small, the user can simply reinitiate the entire download. If the file sizes were extremely large (gigabytes) then it would be a useful feature.

The feature of renaming filenames (RNFR and RNTO) was not implemented because the author assumed that the users would not need to constantly rename files on the server.

Functionality for cancelling the previous command was not implemented because the author already implemented complementary features, e.g. a user can delete a file that was uploaded, a user can switch back to the parent directory, etc.

The feature for returning the names of files in a directory in the form of a list (NLST) was not implemented because LIST was already implemented (NLST returns a list of names only, whereas LIST returns all information of the files in the specified directory).

The feature for requesting help with commands (SITE) was not implemented because the author implemented this feature in the user-interface, not as an FTP command.

The feature of requesting server status (STAT) was not implemented because the user can simply query the server for specific details such as the mode of operation as well as the file structure used (instead of receiving a list of all details at once).

4.4. External help

See Appendix A, Table 3 for a breakdown of the external help obtained from student 1599953 for the successful completion of the app.

5. COMMANDS AND RESPONSE MESSEGES

5.1. Basic FTP commands.

There are 10 basic commands that were implemented for FTP application, namely PORT, USER, NOOP, TYPE, QUIT, STRU, STOR, PASS, MODE and RETR. See Appendix A, Table 1 for a brief description of the response message(s) that correspond to each of the basic commands.

5.2. Extra FTP commands.

There are 9 extra commands that were implemented for the FTP application, namely PASV, CWD, CDUP, PWD, MKD, RMD, DELE, SYST and LIST. See Appendix A, Table 2 for a brief description of the response message(s) that correspond to each of the extra commands.

6. STRUCTURE OF CODE

The FTP application is based on the FTP model. The FTP model consists of the following main subsystems: user-interface (UI), user protocol interpreter (user PI), user data transfer protocol (user DTP), server protocol interpreter (server PI) and server data transfer protocol (server DTP).

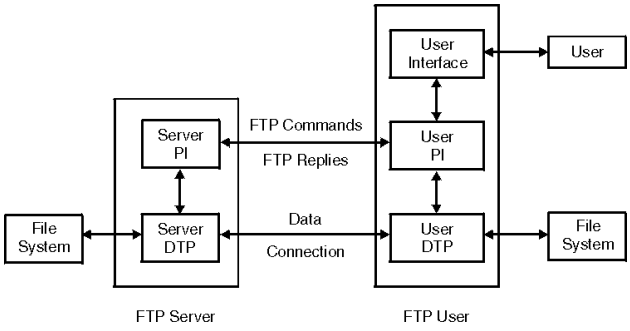


Figure 1: The FTP model.

6.1. User-interface (UI)

The user-interface is the middleman between the FTP app and the user. The UI is run from the ClientUI.py file, which imports the mainClient.py file. The UI makes use of the `Cmd` class, which is part of the `cmd` module. ClientUI.py contains one class called `class myPrompt(Cmd)`. The class contains 13 methods which are responsible for getting input in a user-friendly manner. Two examples of the methods contained by this class are `do_list(self)` and `do_login(self,user)`. The purpose of the first method mentioned above is to list the files in the current directory. In the command line, the user need only type the following in the command line: `list`. The output will be the list of all files in the current directory. The purpose of the second method given as an example is to login. In the command line, the user need only type `login` followed by a valid username, separated by a space. An example of using this function to login to the command line is as follows: `login` Gerald. The application will then prompt the user to enter a valid password. If the password is correct, the user is successfully logged in. An added feature of the UI is a help-feature. This feature allows the user to request a description for all the functions performed by the FTP app. To request a list of the functions supported by the app, the user must press '`?`' or `help` in the command line. To get an explanation for a specific function, the user must type `help` followed by the function in question. For example, `help delete` will give the user specific information on how to delete a file from the server.

6.2. User protocol interpreter (user PI)

The purpose of the user PI is to send commands from the UI to the server, and to interpret replies received from the server. The user PI is contained in the ClientPI.py file. The file imports from the socket module and the ClientDTP class (which will be explained later). The ClientPI.py file contains one class, `class ClientPI()`. The class contains 8 member variables, 7 private methods and 20 public methods. The member variables store useful information and set flags. For example, `clientIP` stores the IP address of the local site in the form of a string, e.g. "127.0.0.1", and `commandIsActive` is a Boolean value which stores the active status of the command connection. The private methods are responsible for the low-level transmission and interpretation of commands and replies between the server and the local system. For example, `isKeyValid(key)` sends the password to the server to check for validity, and `receiveCommand()` decodes and prints a reply from the server to the local system. The public functions are responsible for converting user input into commands. For example, `makeDirectory(dir)` sends the MKD command, along with the pathname, to the server, and `serverOS()` generates the SYST command.

6.3. User data transfer protocol (user DTP)

The purpose of the user DTP is to send and to receive data between server and the local computer. The user DTP is contained in the ClientDTP.py file. This file imports from the socket module, the random module and the os module. The file contains one class, `class ClientDTP()`. There are 11 member variables, 4 private methods and 14 public methods. The member variables store useful information such as folder names and flags. For example, the variable `sendFolder` stores the folder name "ToServer/", and `isConnectionPassive` stores a Boolean value indicating whether the data connection to the server is active or passive. The private methods establish a data connection between the server and the user. For example, `extractPassiveServerIP(addr)` extracts the IP address of server to make a passive connection, and `extractPassiveServerPort(addr)` extracts the port number of the server to establish a passive data connection. The public methods handle the transmission of data between the server and the client. For example, the

method `toServer(file)` converts a file to bytes and sends the bytes from the client to the server, and the method `fromServer(file)` receives bytes from the server and converts them to a file in the local site.

6.4. Server protocol interpreter (server PI)

The purpose of the server PI is to send replies from the server to the client. The server PI is contained in the ServerPI.py file. It imports from the threading module, the socket module and the ServerDTP class. ServerPI.py has one class, `class ServerPI(threading.Thread)`. This class has 14 member variables, 4 private methods and 19 public methods (corresponding to each of the implemented FTP functions). The member variables store useful server information and keep track of flags. For example, `currentMode` stores the mode of operation of the server (which is "S" for stream-mode for this specific FTP application) and `isUserRecognised` stores a Boolean value indicating whether the user has an account on the server. The private methods send replies to the user and process commands received from the user. For example, `sendData(reply)` sends a reply from the server to the user, and `commandLength(cmd)` extracts the FTP command from the data received from the user. The public methods implement the standard FTP functions. For example, `QUIT()` sends code 221 to the user before it terminates the data connection and the command connection, and `PWD()` sends code 200 to the user and prints the current working directory of the user.

6.5. Server data transfer protocol (server DTP)

The purpose of the server DTP is to send and receive data to/from the user. The server DTP is contained in the ServerDTP.py file. It imports from the following Python modules: socket, random, os, datetime and stat. It contains the definition of one class, `class ServerDTP()`. The class has 12 member variables, 8 private methods and 14 public methods. The member variables store useful information. For example, `dataPort` stores the port number of the data connection, and `bufferSize` stores the buffer size used when transmitting data. The private methods handle the data connection between the user and the server. For example, `closeDataConnection()` terminates the data connection, while the method called `activateConnection(addr)` establishes the data connection. The public methods transmit and handle data between the user and the server. For example, `doesFileExist(file)` checks the server file system to determine whether the file exists, and the method `makeDirectory(path)` creates a directory in the file system with the specified path.

7. DETAILED IMPLEMENTATION OF FEATURES

7.1. Login and logout

To implement the login feature, the `login` command in the UI passes the username to the `login(name)` method in the client side. This method sends the USER command together with the username to the server through the command connection. The server responds with code 331 if the user has an account, or 332 if the user does not have an account. If the user has an account, the user is then prompted to provide the password. The method `isKeyValid(key)` on the client side sends the PASS command together with the key, to the server. The server replies with code 230 (and successfully logs the user in) if the key is correct, or 501 if the password is incorrect.

To implement the logout feature, the `quit` command in the UI calls `exit()` method on the client side. This method sends the QUIT command to the server. The server replies with code 221 and closes the data connection and the command connection.

7.2. Upload and download files

To implement the upload of files, the `upload` command in the UI calls the `upload(file)` method on the client side. This method first checks the local system to determine whether the specified file exists. If the file exists, the method verifies that the data connection is active. If the data connection is open, the method sends the FTP command STOR together with the file name to the server. If the server replies with code 125, the method reads the bytes from the local file and sends the bytes through the data connection. After file transfer, the method closes the data connection.

To implement the download of files, the `download` command in the UI calls the `download(file)` method on the client side. Before sending the FTP command RETR together with the filename, the method verifies that the data connection is open. If the specified file exists on the server in the working directory, the server responds with code 125 and sends the bytes of the file to the client side. Thereafter, the method writes the bytes to a file in the local site. Lastly, the method terminates the data connection.

7.3. Check connection

The `ping` command in the UI calls the `ping()` method in ClientPI.py. This method sends the NOOP command to the server. The server responds with code 200 if the connection between the server and the client is OK.

7.4. Query OS

The feature to query the server OS uses the `os` command in the UI. This command calls the `serverOS()` method on the client side. This method sends the FTP command SYST

to the server. The server responds with code 215 and informs the client of the system type, which is UNIX for this application.

7.5. Handle directories

To handle directories, the UI requires the user to specify the name of the directory as an argument (except for printing the working directory and changing to parent directory, which do not take an argument). The relevant command in the UI calls the relevant method in the client side and passes the argument (if required) to the relevant method. For example, to implement printing the working directory, the command `this_dir` in the UI calls the method `thisDirectory`() on the client side. This method sends the FTP command PWD to the server. The server responds with code 200 and shows the client the working directory. The other features (changing directory, making a directory, etc) work in a similar way, albeit with different arguments and server responses.

7.6. Delete a file

To implement deleting a file, the `delete` command in the UI passes the name of the file entered by the user to the `delete`(`file`) method on the client side. The method sends the FTP command DELE together with the file name through the command connection to the server. First, the server checks if the file exists in the server at the working directory. If not, the server sends code 501 to inform the user that the file does not exist. Otherwise, the server deletes the file and sends code 250 to the user to inform the user of the successful deletion of the file.

7.7. List files

To implement listing the files in the working directory, the `list` command in the user-interface calls the method `updateRemoteDirectoryList`() on the client side. First, this method verifies that the data connection is open. Then, the method sends the FTP command LIST. The server will send code 125 to inform the client that the list of all files in the working directory and their details through the data connection are being sent in the form of bytes. Lastly, the server sends code 226 to confirm the successful operation. The UI obtains the list and formats the list in a user-friendly way.

8. RESULTS

8.1. Wireshark, FileZilla and multiple users

Wireshark was used to sniff packets sent between the server and client on the local site. The screenshot is shown in Appendix B, Figure B1. The server was also tested using FileZilla. The screenshot is shown in Appendix B, Figure B2. Lastly, the multi-threading capability of the file transfer app was tested on the local site. The results of these tests are also shown in the PowerPoint presentation.

9. ANALYSIS

9.1. Discussion

Although the file transfer application requires the user to enter a password, it is still not secure. By inspecting the Wireshark results, the password is clearly visible in packet number 9 (line number 5). This means that an observer can easily obtain the password of the users, because the packet sniffer can simply observe the packets shared between the client and the server. This is a security concern, which is one of the drawbacks of the file transfer protocol [2].

Although the UI has greatly eased the use of the file transfer app, it is still not user-friendly. The UI requires one to be well adept at the command line, and the interface is uninviting. This is not good for user experience.

9.2. Future recommendations

To improve the security of the file transfer application, the password should be encrypted on the client side and decrypted on the server side, which will ensure that the password plaintext is not sent through the data connection or the command connection. This can be accomplished with secure shell (SSH) authentication [3].

Instead of using the basic UI provided by the `Cmd` class of Python, a graphical user interface (GUI) can be included in the app for future improvement. The GUI can be created using PyQt5, which facilitates the design of UI for increased user experience [4].

10. CONCLUSION

The design and implementation of the file transfer app is successful. The app allows a user to login, logout, upload files, download files, check connection status, query server OS, handle directories, delete a file and list files in a folder. A total of 19 FTP commands were implemented to support these features, but many others were omitted in this app. The app was designed according to the standard FTP model. The app was tested successfully with Wireshark and FileZilla. Moreover, the app uses multithreading to handle more than one user simultaneously. The drawbacks of the app are that the password is sent as plaintext through the connection, and the UI is not easy to use. For future work, the password will be encrypted on the local site and decrypted on the server site using secure shell (SSH) authentication, and PyQt5 will be used to design a GUI.

REFERENCES

[1] McCormack, M. A., Schansman, T. T., & Womack, K. K. (1965). 1401 compatibility feature on the IBM System/360 model 30. *Communications of the ACM*, *8*(12), 773-776.

[2] Kurose, J. F. (2005). *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India.

[3] Ylonen, T., & Lonvick, C. (2006). *The secure shell (SSH) authentication protocol*. RFC 4252, January.
[4] Siahaan, V., & Sianipar, R. H. (2019). *Learning PyQt5 with MariaDB for Absolute Beginners: A Hands-On, Practical Database Programming*. SPARTA PUBLISHING.

APPENDICES

APPENDIX A: FTP COMMANDS AND REPLIES, AND EXTERNAL HELP BREAKDOWN

Table 1: FTP commands for minimum implementation.

	Reply	Description
PORT	225	Data connection is open.
	425	Cannot open data connection.
USER	331	Correct username, provide password.
	332	Username is not recognised.
NOOP	200	OK (positive response from server).
TYPE	200	ASCII/Image type has been selected.
	501	Invalid type requested
QUIT	221	Exiting, closing control connection.
STRU	200	File structure has been selected.
	504	Requested structure not implemented
	501	Invalid structure requested.
STOR	125	Transfer has begun.
	226	Transfer is complete.
	426	Transfer is unsuccessful.
PASS	530	User is not logged in.
	230	User has logged in and may proceed.
	501	Invalid password.
MODE	200	Stream mode selected.
	504	Requested mode is unimplemented.
	501	Invalid mode requested.
RETR	125	Transfer has begun.
	226	Transfer is complete.
	426	Transfer is unsuccessful.
	450	File is invalid.

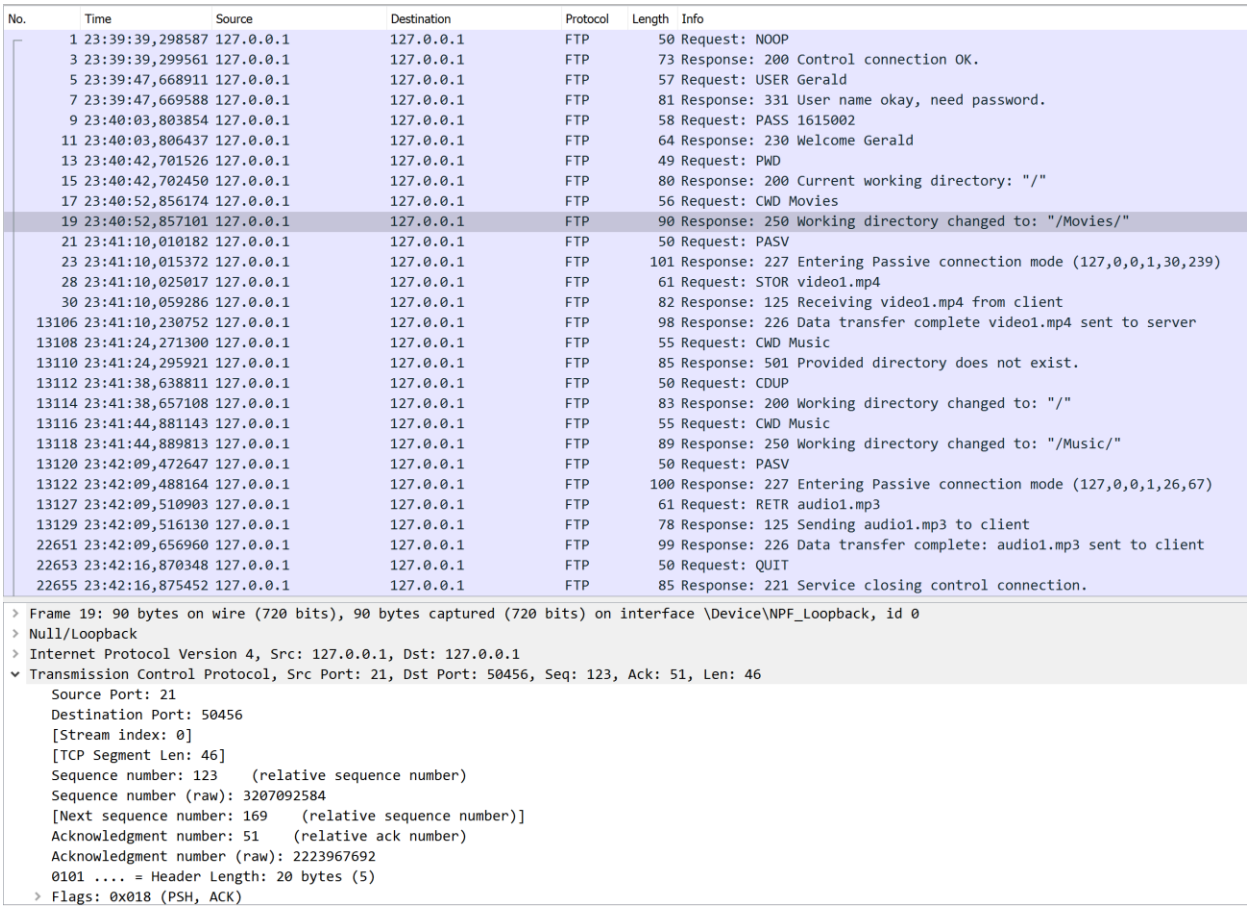
Table 2: Extra FTP commands.

	Reply	Description
PASV	227	Entering passive data mode.
	425	Cannot open data connection.
CWD	250	Working directory is changed.
	501	Directory does not exist.
CDUP	200	Directory is now parent directory.
PWD	200	Print working directory.
MKD	257	Directory successfully created.
	501	Directory already exists.
RMD	250	Directory successfully deleted.
	501	No such directory.
DELE	250	File is successfully deleted.
	501	File does not exist.
SYST	215	UNIX system type.
LIST	125	Transferring file list.
	226	Transfer complete.
	426	Transfer of file list is unsuccessful.
	450	Path is invalid

Table 3: External help obtained from ELEN4017 student 1599953 (lab partner).

Source	Server DTP	Server PI
Student 159953 (lab partner)	The student assisted in describing the structure and goal of the methods used to send and receive data to the client.	The student assisted in integrating the FTP commands with the response of the server DTP, e.g. “STOR instructs the server to place a file in the server file system.”

APPENDIX B: RESULTS



No.	Time	Source	Destination	Protocol	Length	Info
1	23:39:39,298587	127.0.0.1	127.0.0.1	FTP	50	Request: NOOP
3	23:39:39,299561	127.0.0.1	127.0.0.1	FTP	73	Response: 200 Control connection OK.
5	23:39:47,668911	127.0.0.1	127.0.0.1	FTP	57	Request: USER Gerald
7	23:39:47,669588	127.0.0.1	127.0.0.1	FTP	81	Response: 331 User name okay, need password.
9	23:40:03,803854	127.0.0.1	127.0.0.1	FTP	58	Request: PASS 1615002
11	23:40:03,806437	127.0.0.1	127.0.0.1	FTP	64	Response: 230 Welcome Gerald
13	23:40:42,701526	127.0.0.1	127.0.0.1	FTP	49	Request: PWD
15	23:40:42,702450	127.0.0.1	127.0.0.1	FTP	80	Response: 200 Current working directory: "/"
17	23:40:52,856174	127.0.0.1	127.0.0.1	FTP	56	Request: CWD Movies
19	23:40:52,857101	127.0.0.1	127.0.0.1	FTP	90	Response: 250 Working directory changed to: "/Movies/"
21	23:41:10,010182	127.0.0.1	127.0.0.1	FTP	50	Request: PASV
23	23:41:10,015372	127.0.0.1	127.0.0.1	FTP	101	Response: 227 Entering Passive connection mode (127,0,0,1,30,239)
28	23:41:10,025017	127.0.0.1	127.0.0.1	FTP	61	Request: STOR video1.mp4
30	23:41:10,059286	127.0.0.1	127.0.0.1	FTP	82	Response: 125 Receiving video1.mp4 from client
13106	23:41:10,230752	127.0.0.1	127.0.0.1	FTP	98	Response: 226 Data transfer complete video1.mp4 sent to server
13108	23:41:24,271300	127.0.0.1	127.0.0.1	FTP	55	Request: CWD Music
13110	23:41:24,295921	127.0.0.1	127.0.0.1	FTP	85	Response: 501 Provided directory does not exist.
13112	23:41:38,638811	127.0.0.1	127.0.0.1	FTP	50	Request: CDUP
13114	23:41:38,657108	127.0.0.1	127.0.0.1	FTP	83	Response: 200 Working directory changed to: "/"
13116	23:41:44,881143	127.0.0.1	127.0.0.1	FTP	55	Request: CWD Music
13118	23:41:44,889813	127.0.0.1	127.0.0.1	FTP	89	Response: 250 Working directory changed to: "/Music/"
13120	23:42:09,472647	127.0.0.1	127.0.0.1	FTP	50	Request: PASV
13122	23:42:09,488164	127.0.0.1	127.0.0.1	FTP	100	Response: 227 Entering Passive connection mode (127,0,0,1,26,67)
13127	23:42:09,510903	127.0.0.1	127.0.0.1	FTP	61	Request: RETR audio1.mp3
13129	23:42:09,516130	127.0.0.1	127.0.0.1	FTP	78	Response: 125 Sending audio1.mp3 to client
22651	23:42:09,656960	127.0.0.1	127.0.0.1	FTP	99	Response: 226 Data transfer complete: audio1.mp3 sent to client
22653	23:42:16,870348	127.0.0.1	127.0.0.1	FTP	50	Request: QUIT
22655	23:42:16,875452	127.0.0.1	127.0.0.1	FTP	85	Response: 221 Service closing control connection.

```
> Frame 19: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 21, Dst Port: 50456, Seq: 123, Ack: 51, Len: 46
    Source Port: 21
    Destination Port: 50456
    [Stream index: 0]
    [TCP Segment Len: 46]
    Sequence number: 123    (relative sequence number)
    Sequence number (raw): 3207092584
    [Next sequence number: 169    (relative sequence number)]
    Acknowledgment number: 51    (relative ack number)
    Acknowledgment number (raw): 2223967692
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x018 (PSH, ACK)
```

Figure B1: Testing the file transfer app with Wireshark.

```
Status:    Connecting to 127.0.0.1:21...
Status:    Connection established, waiting for welcome message...
Response:  220 Successful control connection
Status:    Plain FTP is insecure. Please switch to FTP over TLS.
Command:   USER Gerald
Response:  331 User name okay, need password.
Command:   PASS *******
Response:  230 Welcome Gerald
Command:   SYST
Response:  215 UNIX system type.
Command:   FEAT
Response:  502 Command not implemented
Status:    Server does not support non-ASCII characters.
Status:    Logged in
Status:    Retrieving directory listing...
Command:   PWD
Response:  200 Current working directory: "/"
Command:   TYPE I
Response:  200 Binary (I) Type selected
Command:   PASV
Response:  227 Entering Passive connection mode (127,0,0,1,28,129)
Command:   LIST
Response:  125 Sending file list
Response:  226 List successfully sent
Status:    Directory listing of "/" successful
Status:    Starting upload of C:\Users\user\Documents\4th Year\ELEN4017_Networks\File Transfer App\Client\ToServer\video1.mp4
Command:   PASV
Response:  227 Entering Passive connection mode (127,0,0,1,26,175)
Command:   STOR video1.mp4
Response:  125 Receiving video1.mp4 from client
Response:  226 Data transfer complete video1.mp4 sent to server
Status:    File transfer successful, transferred 6 690 957 bytes in 1 second
Status:    Retrieving directory listing of "/Docments"...
Command:   CWD Docments
Response:  250 Working directory changed to: "/Docments/"
Command:   PWD
Response:  200 Current working directory: "/Docments/"
Command:   PASV
Response:  227 Entering Passive connection mode (127,0,0,1,22,166)
Command:   LIST
Response:  125 Sending file list
Response:  226 List successfully sent
Status:    Directory listing of "/Docments" successful
Status:    Starting download of /Docments/file1.pdf
Command:   PASV
Response:  227 Entering Passive connection mode (127,0,0,1,23,207)
Command:   RETR file1.pdf
```

Figure B2: Testing the server of the file transfer app with FileZilla.