File Transfer Application Report

ELEN4017 - Network Fundamentals

Lecturer: Asad Mahmood

2018/04/02

Christopher Maree - 1101946 & Daniel Edwards - 1055301

***Abstract*— This paper presents the implementation of a socket level File Transport Protocol (FTP) client and server, implemented in Python. FTP is discussed and critiqued and then the implementation specific details are outlined. Code structure, testing of the implementation as well as critical analysis thereof are also discussed. Wireshark screen shots are presented to verify the protocol.**

I. Introduction

File Transfer Protocal, or FTP, is a ubiquitous protocol invented in 1971[1]. FTP was created first before TCP/IP existed and has survived the test of time, still used in many contexts today despite its age. FTP provides a mechanism for transferring files between a client and server. The protocol itself as well as its implementation will be discussed in this report. The project Github repo, with all code, screenshots, and this report can be found here.

II. Problems with the FTP protocol

There are a number of inherent problems with the FTP protocol, primarily due to its age. Despite these problems, it is still used in a multitude of contexts as a result of its first mover advantage and simplicity of usage[2]. FTP is primarily used today in antiquated systems that have not been updated to a more modern paradigm. All of FTPs functionality can now be been replaced by HTTP, with particular reference to WebDAV[2], a set of HTTP methods enabling all the functionality of FTP and more. The primary problems with FTP are as follows:

A. Protocol Insecurity

Usernames and passwords are sent over the network in plain text. This means that anyone performing any kind of network sniffing can pick up these packets. This makes the protocol especially vulnerable to packet capture attacks[3], [4].

B. Firewall Complexity

FTP utilizes random ports to establish connections. Strict firewalls make this a problem in establishing connections. This is dealt with through the use of passive mode and is discussed later on in this report in section III.

C. Protocol Unreliability

FTP struggles to gracefully deal with network transmission errors. This means that larger file transfers can fail if there are connectivity problems with the internet connection during the transfer.

III. Overview of the FTP Protocol inner-workings

The FTP protocol enables the transfer of data between a client and a server. It employees separate data and control connections for transfering data and relaying commands back and forth between the client and server. The protocol operates in one of two modes: active or passive, depending on the network configuration and setup. Both configuration involve the client creating a TCP connection from a random, unprivileged port to the FTP server on port 21.

Active mode entails the the client informing the server for an active listening port on which the server should connect. Passive mode, on the other hand, involves the server specifying an IP address and port onto which the client can bind from a random client port. The key difference here is that in active mode, the client tells the server where to connect and in in passive mode the server tells the client where to connect. Passive mode is primarily used as if the client is behind a firewall, it might be unable to accept incoming TCP connections from the server on the specified port. By letting the client establish the outbound connection in passive mode, this problem is avoided.

All instructions between the client and server are sent over the control connection, where the server uses standardized three-digit status codes to communicate responses to the client. These status codes represent particular messages and represent operations between the client and server.

Data transferred using FTP is encoded in one of four ways:

1) ASCII mode, primarily used for the transfer of plain text. All information transferred is converted into 8-bit ASCII before and after transmission.
2) Binary mode, also called Image mode, enables the sending of data in a byte by byte fashion. This data is reconstructed on the receiving side through the use of a byte stream. Most implementations use Binary mode for data transfer.
3) The two other transfer methods are EBCDI mode and Local mode, both of which are rarely used as they are more proprietary.

The process of data transfer is done in one of three ways:

1) Stream mode, where data is sent in a contiguous stream. In this transfer mode, all processing is done by the TCP protocol.
2) Block mode entails data being broken up into multiple chunks that are then transmitted using TCP. These blocks are re constructed on the receiving side.
3) Compressed mode facilitates basic data compression before transmission.

Now that the FTP protocol has been outlined in detail, the Python implementation presented in this report can be discussed.

IV. Overall System Implementation

The development process of the FTP client and server involved first building a dummy client and server using the Python `ftplib` library to gain an understanding of how the protocol worked. This dummy client had all the functionality of a

traditional FTP client and server, without needing to deal with the socket level networking. The client and server could then be used as a point of reference later on as additional testing frameworks. The development of the dummy client and server also provided insight into how the User interface should look in the final product.

Next, a client and server were built using basic socket level networking, without using any priority libraries. This client and server setup were built using the FTP RFC 959 documentation[1]. All required functionality to meet minimum implementation was added to both the client and server, as laid out in section 5.1 of the RFC 959[1]. A GUI was built for the client but this was later abandoned in favour of a multi-windowed, full-screen command line application as it was felt that this better showed the underlying networking communication between the client and server without being obscured by a GUI.

A. Implemented Functionality Overview

All functionality required to meet minimum implementation specified in the RFC 959 was implemented, as well as a few additional features. This functionality broken down at a high level enables the user to:

- Establish a control connection with a specified server
  - Can change the connection port but is specified as 21 by default
- Log into the FTP server with the client
  - Can use a username and password or log in anonymously
- Perform a number of commands on the server, such as:
  - List directory contents
  - List working directory
  - Change working directory
  - Change to parent directory
  - Uploading of files
  - Downloading of files
  - Printing commands help
- Perform a number of commands on the client file system, such as:
  - Changing working directory
  - Listing directory contents
  - Changing to parent directory
  - Viewing basic contents of text files
  - Printing commands help

The server is able to accommodate all aforementioned functionality, responding with appropriate responses to the client. The functionality outlined above involves the implementation of FTP commands in accordance with the FRC 959. Each of these implemented functions, as well as a description of usage, is outlined in section V.

B. Non-implemented Functionality

Core functionality for the minimum implementation requirements was implemented fully. Advanced error handling on the client side was, however, not implemented. For example, if the user tries to upload a file that already exists, they get blocked by the server by default. Ideally, the client should check if the file exists first and if it does it should tell the user. The user should then be able to pick if they want to override it, rename or cancel the upload. This responsibility should remain with the client, not the server. This kind of functionality ins not required for the minimum implementation requirements so was no implemented in this proof of concept implementation.

The FTP environment allows for data to be transferred three modes namely, stream, block and compression. The block and compression transfer modes were not implemented because the minimum requirements for FTP is that data must be transferred in a stream as stated in the FTP RFC document[1]. The data structure of the information transferred between the server and client can by represented in a file, record or page structure. Similarly to the transfer modes, the file structure is the default requirement and hence, this was the only data structure implemented.

V. Implemented Features Details

Each of the aforementioned features from section III.V are now discussed, along with the associated FTP commands. The order of discussed features follows a basic use case of a client establishing a connection, logging into the server, listing file directories, changing directories, uploading, downloading and then closing the connection.

A. Establishing a Socket Connection

Before any communication between the client and server can commence, a `controlSocket` connection is required to be established. This connection is used to transfer all commands between the client and server and is used in every subsequent implemented command. The `controlSocket` remains open until the session is terminated by the client (or server). The `controlSocket` is implemented using a standard Python socket, binding to a predefined server address and port number (both spesified by the user on opening the application).

B. Logging Into the Server

Once a `controlSocket` has been established, the client can log into the server. This process involves the client first sending the server the username, and then the password. Responses from the server define if the login attempt was successful (or not). The `USER` command is accompanied by the username and the `PASS` command specifies the password sent to the server. A response code of 331 indicates the username is accepted and a password is required. A response code of 332 represented the server needing an account for login. Finally, a response code of 230 indicates successful login to the server and 530 indicates that the client could not login.

Once the user has successfully logged into the server, there are two main kinds of operations that can be performed: commands requiring a `dataSocket` and those that run over the `controlSocket` exclusively. The latter will be discussed first for simplicity.

C. Printing Working Directory

The notion of a working directory is current file path of an application. The `PWD` command (same as Linux) is passed to the server over the `controlSocket` and the server responds with the current path on the `controlSocket`. The response code of 257 from the server indicates that the working directory was successfully sent to the client.

D. Changing Working Directory

Changing the working directory entails browsing to a new path on the server. This is done by specifying the `CWD` command (`cd` in Linux), followed by the desired path change. This path change can be to a subdirectory of the current working directory or the the parent directory, by changing to `..` as the new desired path. A subsequent response code of 250 is sent to the user if the working directory is changed. However, if the directory cannot be found, the server sends a response code of 550.

E. Change to Parent Directory

Change to parent directory moves the current working directory of the server one level up, to the parent of the current path. This is done through the use of the `CDUP` command (`cd ..` in Linux) and does not need any other parameters.

F. Make Directory

Make directory creates a new folder on the server. This is done with the `MKD` command (`mkdir` on Linux). The command is followed by the name of the new directory. If the directory already exists, the server sends a 521 response code to the client. Upon a successful creation of a directory, 257 is sent from the server to the client as the response code.

G. Delete Directory

Remove directory is use to delete directories on the server. This is done with the `RMD` command (`rm -rf` in Linux) followed by the directory name. It can only be used to delete directories, not files. Similarly to the change working directory command, a 550 response code is sent to the client if the specified directory cannot be found. A response code of 250 indicates that the directory was deleted successfully.

H. Delete Files

Files are deleted with the `DELE` command (`rm` in Linux). Verification of desire to delete should be done on the client side. A 250 response code is sent to the client when the file is deleted. A 450 response code indicates an error in deleting the file.

I. Quit Connection

At the end of the session, the client can terminate the connection with the `QUIT` command. This command only terminates the session if the current user is not in the progress of transferring files. If it is currently transferring files, the server will wait until the transfer finishes before closing the connection. The response code 221 indicates that the connection has been terminated.

Next, commands that require a data connection are discussed. The client and server establish an additional connection, as discussed in the FTP implementation in section III to transfer data. At this point, a passive or active connection to the server can be established. Passive was primarily used in testing to simplify firewall interactions. The set up of a data connection will first be discussed then its usage within different commands is outlined.

J. Establishing a Data Connection

The data connection acts as a normal Python socket, running on a predefined address and port. Before the connection is created, the client needs to find the correct port to establish the socket on. This is done in code through the `getPortNumber()` function. The client sends a `PASV` to the server, requesting passive connection information. The server responds with six comma separated values. The first four correspond to the IP address that the client should connect on and the last two are used to calculate the correct port for server connection. The port number is calculated by multiplying the 5th comma separated value by 256 and adding it to the 6th comma separated value. An example server response for the `PASV` command could be $(10, 30, 0, 50, 30, 60)$. From this example, the client knows the server IP address is `10.30.0.50` and the port to connect on is $30 * 256 + 60 = 7740$. The client can now connect to the server on a new socket on `10.30.0.50:7740`.

This data connection is used for uploading, downloading and listing files. For uploading and downloading, a new data connection is used for each file.

K. Uploading Files

Uploading involves transferring files from the client to the server over a data connection. The client first checks that the file the user has requested to upload does indeed exist on the client. If it does not exist, the upload is terminated, else it proceeds. Next, the client tells the server to change into Image (binary) mode through a `TYPE I` command. The server responds with a 200 code once the mode has changed to Image. Next, a `dataSocket` is established with the server, as discussed above. Once the socket is correctly created and connected, a data connection is established with the server. The server responds with a 150 code on successful connection. The client then informs the server that it wants to upload a file, using the `STOR` command, followed by the name that the file should have once uploaded to the server. The client then opens the file within Python and begins transmitting the file in chunks to the server. Once the whole file has been transmitted, the data connection is closed the server sends a 226 code, informing the client of successful file upload. Once upload is complete, the file is closed in Python.

L. Downloading Files

File download is similar to upload in implementation, except the direction of file transfer is now the other way: from the server to the client. The process begins with the client changing the server into Image (binary) mode, using the `TYPE I` command. A `dataAocket` is then connected and configured with the server. The client then uses the `RETR` command, followed by the file that the client wants to download, to initiate the download process. Once the data connection has been opened, the server responds with a 150 response. The client then starts receiving the file in chunks from the server and recombines them into one file on the client side. Once all the file parts have been received by the client, the server sends a 226 transfer successful code. The client then closes the data connection with the server and closes the new file within Python.

M. Listing files within a directory

The `LIST` command is used to request the files within a directory from the server. This command requires the use of a data socket to transfer the list of files from the server to the client. This process begins with the client sending the `LIST` command, to which the server responds with a 150 code. The data connection is then created and the server sends the file listing over this connection. After the transfer of file listing is

complete, the data connection is closed the the server sends a 226 over the controlling socket.

VI. System Error Handling

Incorrect information and commands are expected when communicating between nodes. Therefore an error handling scheme is established to ensure that both the client and the server can detect and manage errors. The FTP RFC document provides a list of response codes that correspond to various cases where the transferred command and/or information is incorrect. The error handling methods associated with the implemented functions is discussed below.

A. User Authentication

When a client attempts to login, a username and password is required. The server handles the event of an incorrect username and a username-password mismatch by sending a 530 error code to the client. The client handles this error by repeatedly asking the user for the username and password until it is accepted. This ensures that only authorized users can access the functionality of the server.

B. Directory Operations

Several commands revolve around directory manipulation. The server must ensure that the requested operations do not compromise the integrity of the hierarchy and information and stored on the server. Every directory manipulation command must check the existence of the path specified. If the client attempts to change to nonexistent or invalid path, the server rejects this requests and sends a 550 response code to the client. Similarly, if the client attempts to delete a nonexistent directory, a 550 response code is returned. Directory path names are unique and hence, the server must prevent the client from creating a directory with a name that matches to an existing directory. It does this by sending a response code of 521 to the client in this event.

C. File Operations

As stated above, the destination of the file being transferred is sent during uploading and downloading requests. If this destination already exists, the file is simply overwritten. To prevent this unnecessary loss of data, the server verifies that the file destination is not taken before starting the data transfer. If the destination does exists, a 550 response code is sent to the client indicating this error. In addition, the server prevents the client from deleting files that are unavailable. This includes files that do not exist and files that are currently in use. The client is notified of this error with a response code of 450 sent from the server.

D. Type Validation

The FTP implementation handles ASCII and binary data types which correspond to specific characters. If the server determines that the character sent by the client does not correspond with these data types, a 500 response code is sent to the client.

E. Command Validation

The correct command must be supplied to the server to perform each function. In the event that an unrecognized command is sent to the server, a 202 response code is sent to the client.

VII. Code Structure

Code written for this project was kept as simple as possible as to not get lost in programming complexities and rather to keep the focus of the project on the networking side. Both the client and server have a main while loop that Handel all interactions to act to keep the application running. The implementation of the client and server are discussed below.

A. Client Code Structure

The client was designed to clearly demonstrate all interactions with the server, while providing a mechanism to provide commands and see the associated outputs. This was done by creating a fullscreen terminal application, split into three equal sections corresponding to the: user inputs, client/server communications and system responses. Intelligent command predictive text was implemented to make the user inputs easier to process. History of the users commands was also recorded so common tasks can be re-run with ease.

All custom terminal functionality (full screen application, autocomplete, split terminal application and command history) was implemented using the `python-prompt-toolkit` framework, an open source python library for building command line application interfaces[5]. Client logic runs in a while loop that constantly polls the user input section of the application. When a new line is entered by the user(enter is pressed) the client then runs the corresponding command, feeding in the parameters specified by the user. The client ensures that only defined functions can be run by checking that each command that is run exists. Additionally, the client provides the user with a `help` section, informing the user of all the possible commands that can be sent to the server.

All aforementioned client functionality runs commands on the server. The client does however also enable the user to run commands directly on the client machine itself, by preceding the command with an exclamation mark (`!`). For example, to view the files within the current directory on the client, the `! ls` command can be given to the client. The client will then run the `ls` command on the local machine.

B. Server Code Structure

As discussed above, all of the processes are handled using a `controlConnection`. Therefore, a control socket is established on port 21. When a client contacts the server asking to begin a connection, a new connection is created and sent to an instance of the `FTPServer` class. The `FTPServer` class has threading capabilities which allows for multiple clients to connect to the server simultaneously. This is achieved by running each client socket in a separate thread. The data received from the client consists of a command and often additional information. The class utilizes the `getCommand()` and `getInfo()` function to separate the data received from the client.

Each of the FTP commands have a separate function which encapsulates and executes the procedure for the respective command. Initializing a data socket is required by several FTP commands. A data socket can be acquired by calling the `getDataSocket()` function. The separation of each FTP command into different functions ensures that each function can run independently. The user information is stored in a text file using a JSON format. The server verifies that both the

username exists, and that the password maps to the correct username.

VIII. Solution Testing

It is essential for both the server and the client to be robust which gives rise to a rigorous testing procedure. As stated above, the dummy server and client was used to verify that the constructed server and client were indeed functioning correctly. This verification of functionality is discussed below.

Initially, a testing client was created, with no option for user interaction, to verify each of the FTP commands. These commands were tested on an existing FTP server to ensure that any abnormalities in the client-server interactions were localized to the developed client commands. Furthermore, this allow for specific commands to tested to isolate any errors that occurred. The commands were deemed to be successfully implemented when the response codes received from the server matched the expected response codes specified in the FTP RFC document.

Once all the commands in the testing client were marked as working, the server was constructed to match these commands. Both the testing and dummy client were used to verify that the server was functioning correctly by executing the commands with the client and server hosted locally. In addition to the normal cases sent to the server, the error case were also tested to verify that the server handled these erroneous situations correctly.

Once a correctly working testing client was created, a client with a user interface could be developed, enabling user interaction with the FTP protocol. This was again tested against a FTP server to ensure correctness of implementation. The next step in the procedure was to verify that the server and the client were able to communicate remotely. This was done by running the server and client code on separate machines on the same network and analyzing the response codes.

Developing and testing the application in this way ensured that no errors creped into the final implementations, and enabled the final solution to be well refined.

IX. Solution Critical Analysis

All required functionality was implemented in accordance with the RFC minimum implementation requirements. Both the server and client were able to communicate over a LAN, with all implemented functions correctly working. That being said, some functionality could be greatly improved in future iterations of the application.

While the client does provide a clear representation of the interaction with the server, showing all commands and associated outputs, it is less intuitive than a GUI. it does not provide a mechanism for performing bulk uploads/downloads of files. The commands specified for the client mimic that of a Linux system to keep the interface as simple as possible. This however could pose to be challenging for non Linux users.

The formatting of system output within the client could be improved as long running lines can sometimes be hard to distinguish. Additionally, the usage of colour within the terminal output could make it easier to read the application outputs.

A server interface would also have been useful. This would enable the server administrator to see the current connections to the server and manage them accordingly. A watchdog could also have been setup to ensure that the server starts up again, in the event that it goes down[6].

Error handling, on both the server and client, could also have been improved. Common errors were dealt with in the final implementation but more intelligent, contextual handling could have been done, like informing the user of existing files on download and checking of adequate disk space before upload/download.

A more streamlined installation process would have made the application usability easier, removing some of the library dependencies. This could have been achieved through packaging the application as a release application and then deploying it to a package manager, such as pip[7].

X. Wireshark Results

All basic commands results were captured using wireshark. These can be seen in the Appendix of this report and within the submitted files.

XI. Division of Tasks and Individual Contributions

The project work load was split as evenly as possible, playing to each members strengths. The Specific tasks each member took on are outlined in the table below.

XII. Division of Labor within Project

TABLE I: Project Devision of Labor

Task	**Student**
Dummy server and dummy client	Chris Maree
Research into FTP commands	Chris Maree, Daniel Edwards
Server and client using sockets	Daniel Edwards
Initial GUI	Chris Maree
Command line interface	Chris Maree
Interface and back-end integration	Chris Maree, Daniel Edwards
Report writing	Chris Maree, Daniel Edwards
WireShark testing	Daniel Edwards
Report editing and formatting	Chris Maree

XIII. How to Use the application

The setup process to run the client and server is relatively straight forward, requiring few dependencies. The client, however, does require a small setup to correctly configure the `python-prompt-toolkit` library. A full detailed installation guide is presented in the submitted directory as README.md (or an exported pdf version under README.pdf) and can be found on the project Github repo here.

XIV. Conclusion

This report presented the FTP protocol and analyzed its usefulness in a modern context. A high level technical overview of the protocol was presented, followed by the python specific implementation created for the project. Testing was preformed to ensure the correctness of the implementation. The solution was analyzed critically and future improvements were presented.

APPENDIX I
WIRESHARK RESULTS

Screen shots taken from Wireshark demonstrate the correctness of the FTP implementation.
Only the key screenshots are shown below, with the github repo (and submitted zip file) containing many more showing all client interactions.

Figure 1, 2, 3, 4, and Figure 5 illustrate the interaction between the server and the client when the client attempts to login.

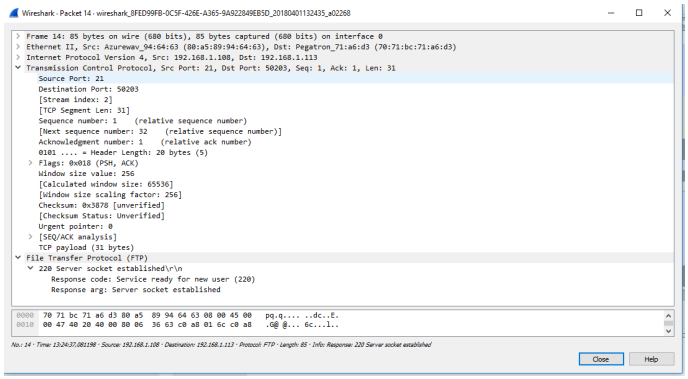

Fig. 1: Welcome Message

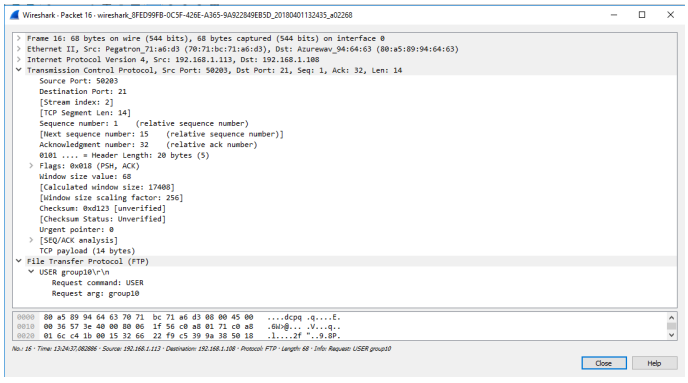

Fig. 2: Client Sends Username

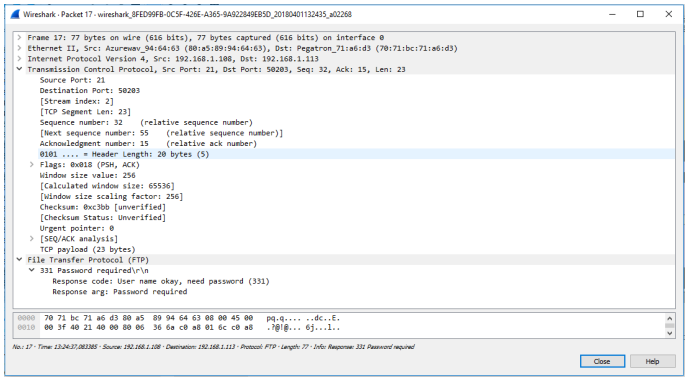

Fig. 3: Server Requests a Password

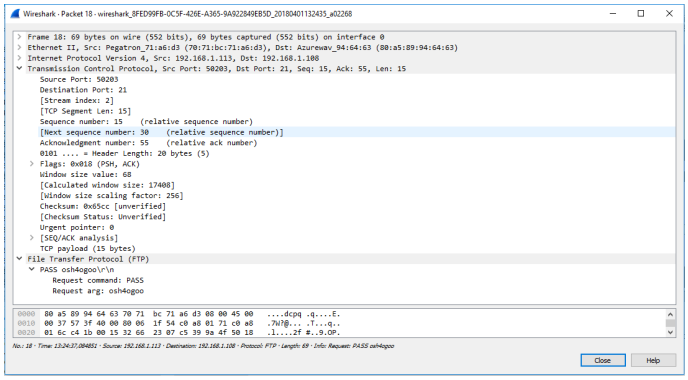

Fig. 4: Client Sends the Password

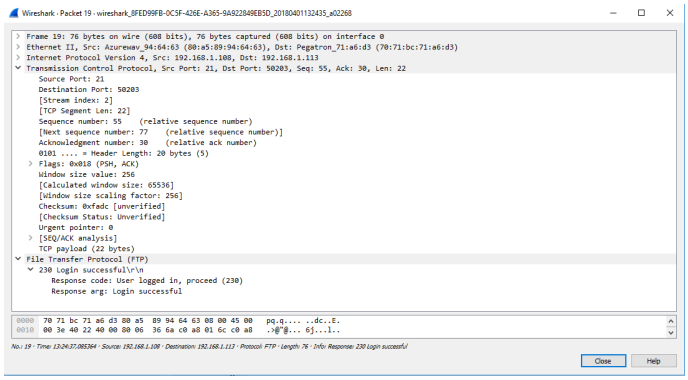

Fig. 5: Login Successful

The *Wireshark* reports below correspond to the interactions between the client and the server for directory manipulations.

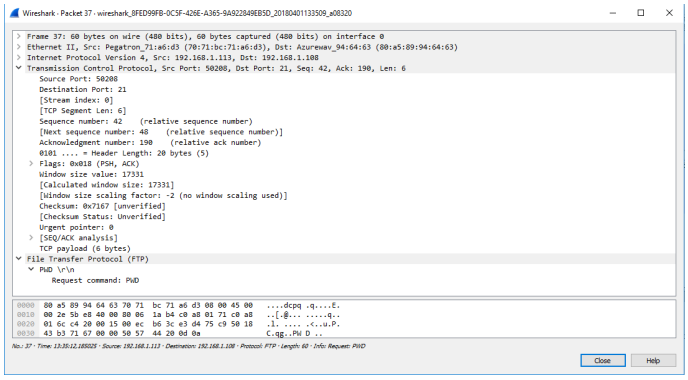

Fig. 6: PWD Request

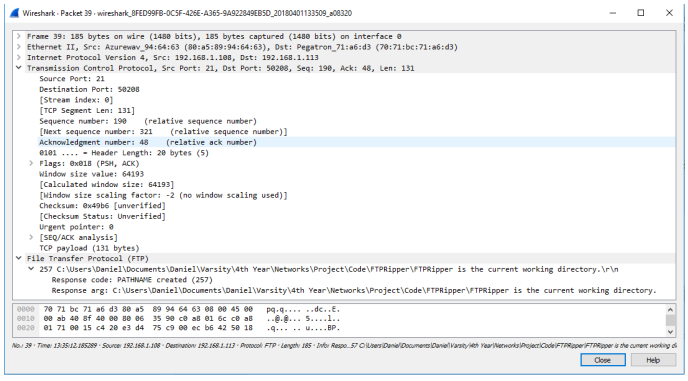

Fig. 7: PWD Response

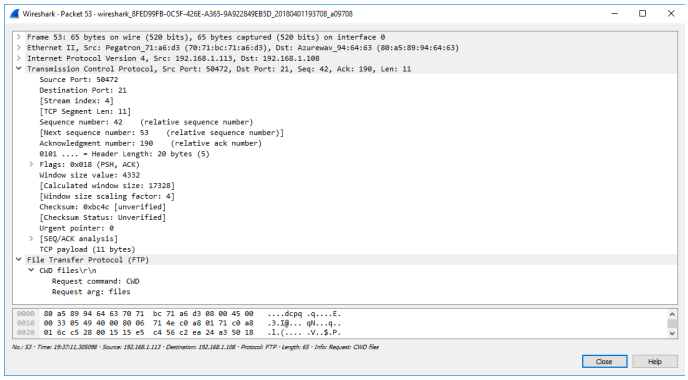

Fig. 8: CWD Request

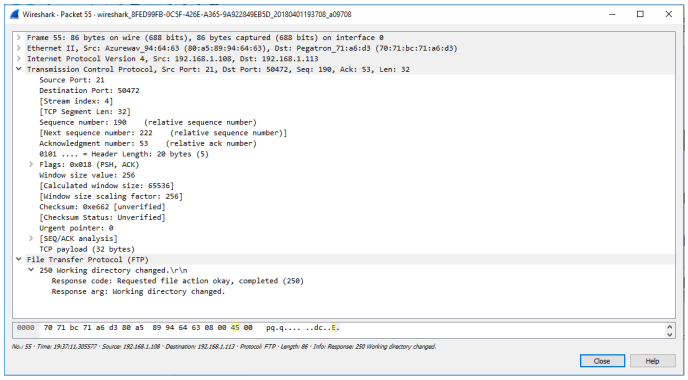

Fig. 9: CWD Response

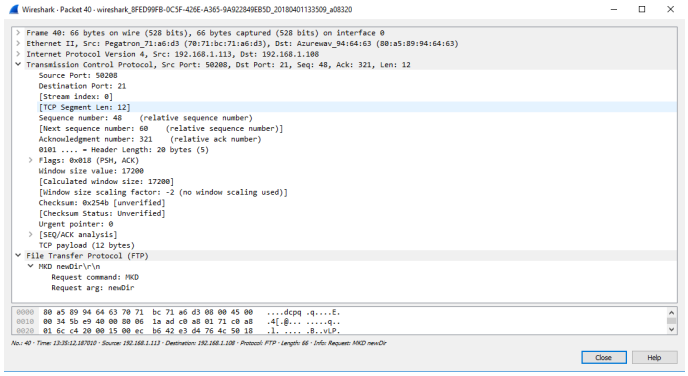

Fig. 10: MKD Request

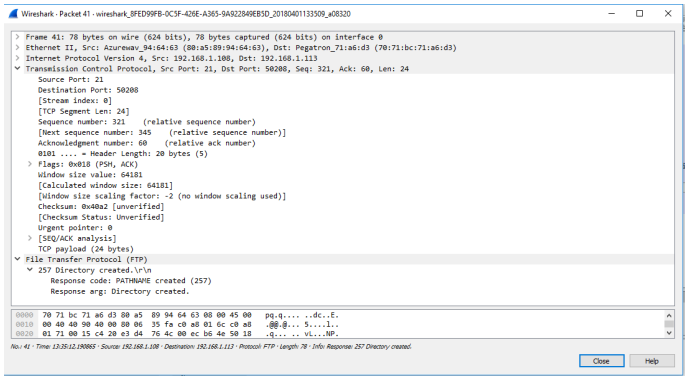

Fig. 11: MKD Response

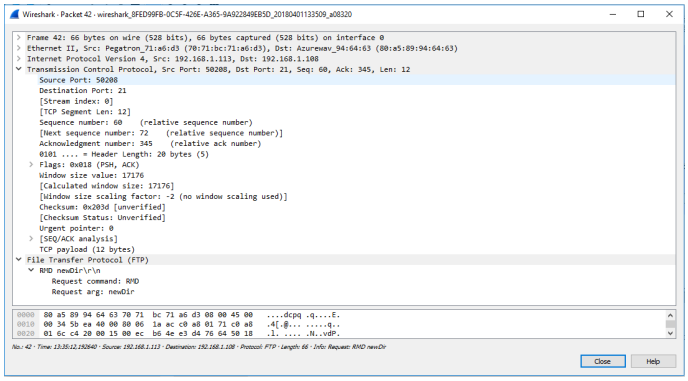

Fig. 12: RMD Request

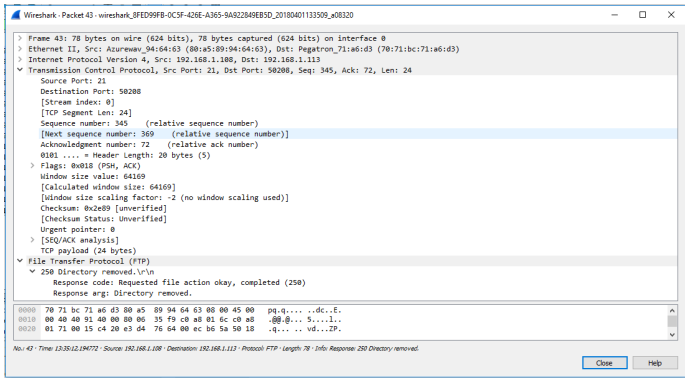

Fig. 13: RMD Response

The following figures illustrates the *Wireshark* screenshots of file operations which include deleting, uploading and downloading a file.

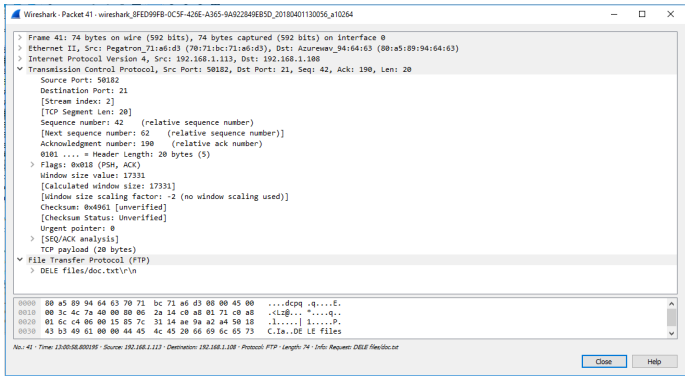

Fig. 14: DELE Request

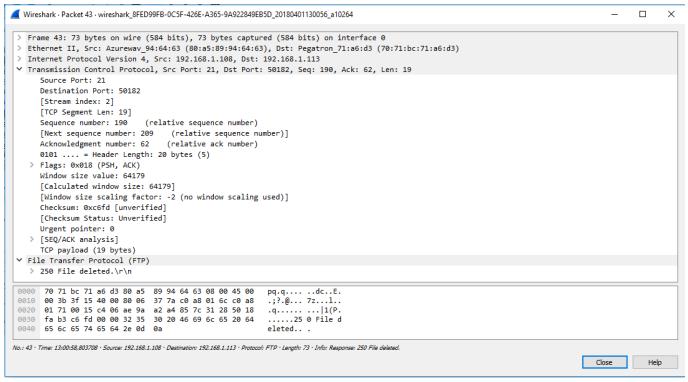

Fig. 15: DELE Response

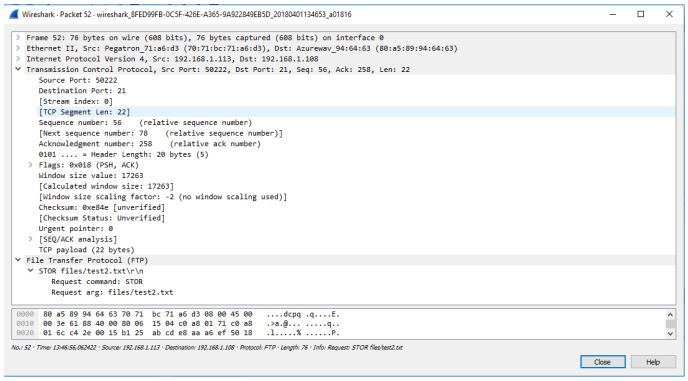

Fig. 16: STOR Request

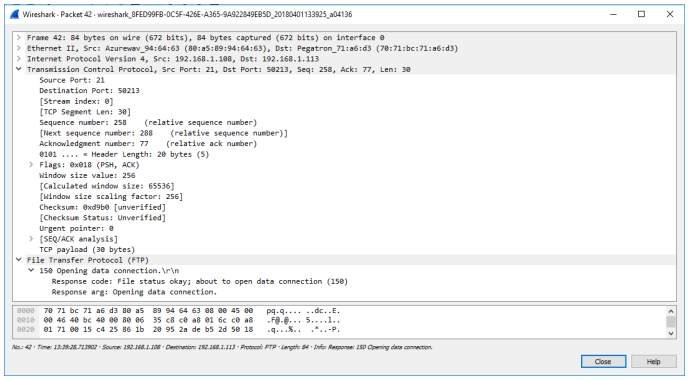

Fig. 17: STOR Response

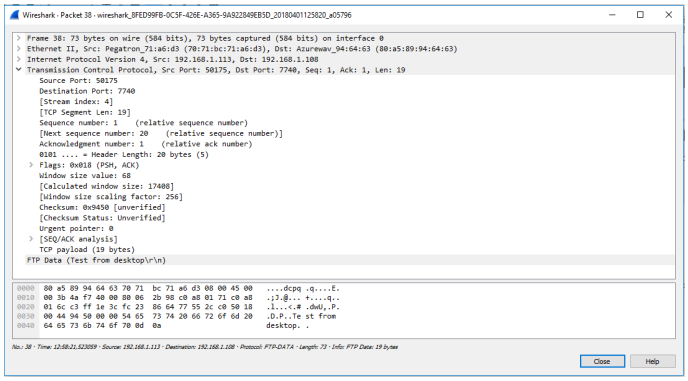

Fig. 18: STOR Data Transfer

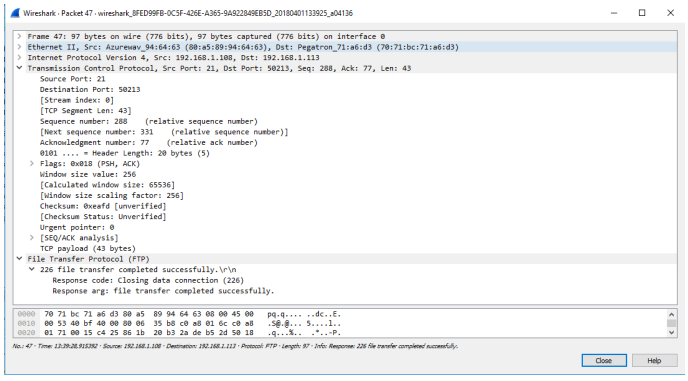

Fig. 19: Uploading Complete

The downloading response sequence to client from the server is almost the same as the upload sequence. The only call that differs is the client request which is given in Figure 20.

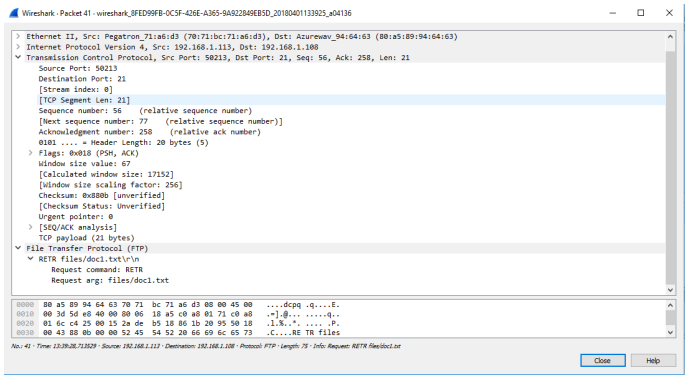

Fig. 20: Download Request

REFERENCES

[1] J. Postel and J. Reynolds, "File Transfer Protocol," 1985. [Online]. Available: https://tools.ietf.org/html/rfc959

[2] AlBlue, "Why do people still use FTP?" 2009. [Online]. Available: http://alblue.bandlem.com/2009/02/why-do-people-still-use-ftp.html

[3] Cisco Academy, "Lab -Using Wireshark to Examine FTP and TFTP Captures," 2013. [Online]. Available: http://static-course-assets.s3.amazonaws.com/IntroNet50ENU/files/7.2.4.3Lab-UsingWiresharktoExamineFTPandTFTPCaptures.pdf

[4] Brian Prince, "Should Organizations Retire FTP for Security? — SecurityWeek.Com," 2012. [Online]. Available: https://www.securityweek.com/should-organizations-retire-ftp-security

[5] Jonathanslenders, "python-prompt-toolkit," 2018. [Online]. Available: https://github.com/jonathanslenders/python-prompt-toolkit

[6] L. die.net, "watchdog(8): software watchdog daemon - Linux man page." [Online]. Available: https://linux.die.net/man/8/watchdog

[7] Digital Ocean, "How To Package And Distribute Python Applications — DigitalOcean." [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-package-and-distribute-python-applications