

Design and Implementation of a File Transfer Protocol Client and Server using Python Socket Library

ELEN4017 - Network Fundamentals

Kavilan Nair (1076342) & Iordan Tchaporov (1068874)

03/04/2018

Abstract: This document outlines the design and implementation of a File Transfer Protocol client and server using Python3.6. These programs adhere to the minimum implementation detailed in the RFC959 document, with the exception of the MODE and STRU commands. The commands as well as the reply codes implemented are presented and a brief description is provided. The server was able to handle multiple concurrent users through multi-threading. The FTP client and server were only able to perform data transfer in active mode when run on two separate computers connected over a wireless network.

1. INTRODUCTION

This report details the implementation of a File Transfer Protocol (FTP) client and server. These programs make use of the protocol to send and receive different files between computers on a network or on the internet. The implemented client and server must abide to RFC959, a document that outlines the standard for FTP programs so that they can communicate with other FTP clients and servers universally [1]. The programs were created using Python 3.6, PyQt5 and QT Designer by a group of 2 students. This report gives background to the file transfer protocol and its working, explains the commands and replies sent by client and server respectively, discusses the code implementation, documents the results of client and server communication and critically analyses the 2 programs.

2. BACKGROUND

The internet protocol hierarchy is made up of 5 layers which are called the Application layer, Transport layer, Network layer, Link layer and Physical layer [2]. The file transfer protocol is part of the Application layer as it makes use of network communications. The following sections are specific to the structure of FTP, how it works and the different types of data connections it utilizes.

2.1 Structure of FTP

FTP, on the Application layer, runs on top of the Transmission Control Protocol (TCP) which is part of the Transport layer. TCP is a type of protocol used to connect two devices together so that the Internet Protocol (belonging to the Network layer) can send data packets between the devices. A file that is going to be transmitted is divided up into data packets with each packet numbered [3]. While the Internet Protocol is responsible for sending the data packets, it is the responsibility of TCP to verify that all packets have been received correctly [3]. This is done by checking that it has received the correct number of packets based on their numbering. If any number is missing, it requests that the specific packet(s) are re-transmitted. Once all packets are received, they are

then put together at the destination host and the file is re-created. This makes TCP/IP secure and reliable for data transfer.

FTP makes use of two TCP connections to transmit and receive data [2]. One TCP connection is called the control connection and it is where the commands are sent by the client and where the replies are sent by the server to each other. The other TCP connection is called the data connection and is used to transmit the files between server or client when uploading or downloading. The control connection is constant from the moment the client connects to the server until that client logs out (which terminates the connection) while the data connection is only established once a file needs to be sent or received by the client or server. The data connection sends one file or piece of data and then terminates. It is then re-established whenever another file needs to be sent.

2.2 FTP Procedure

The following steps detail how the file transfer protocol works:

- A client attempts to establish a control connection with a server on port 21 of the server. This port is reserved for FTP.
- Once the connection is created, the client can now send commands to the server and the server can send replies to the client.
- Once a file or other data (like a directory list) is requested by the client to either be downloaded or uploaded, the client first sends a command to begin establishing the data connection.
- Based on the type of data connection requested by the client, the server then establishes the appropriate data connection and the data packets are transmitted.
- These are received by the corresponding party and the data connection is terminated.
- The client is then free to send more commands.
- Once the client has finished communicating with the server, a command is sent that terminates the control connection.

2.3 Passive and Active Data Connection

FTP has two different methods to establish the data connection which are called active and passive mode [4]. These two modes are explained below and have their own advantages and disadvantages.

2.3.1 Passive Mode - For passive mode, the client sends a command in which the server replies with its IP address and a port for the client to connect to. The client then establishes its data connection by connecting to the address of server and the specific port that was sent. Figure 1 illustrates this process.

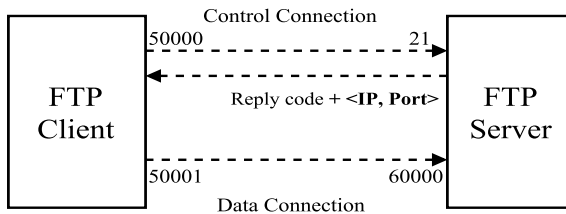


Figure 1 : Establishing a passive data connection.

2.3.2 Active Mode - For active mode, the client sends a command with its IP address and a port for the server to connect to. The server receives this, sends a reply code and then connects to the client's address and the specific port that was sent. Figure 2 illustrates this process.

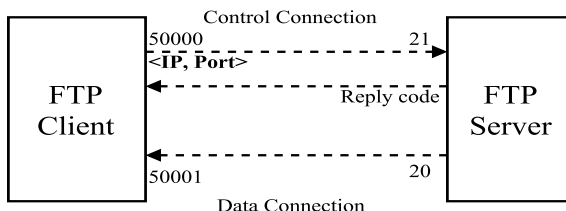


Figure 2 : Establishing a active data connection.

The processes are different but achieve the same result: the creation of a data connection. The use of Active Mode has depreciated due to the fact that routers now have more secure firewalls so unless the port sent during active mode is forwarded on the router's firewall, an incoming connection will be rejected [5]. Viruses and malicious users were connecting to clients this way and as a result, the safer passive mode has become the most used default in FTP servers and clients.

3. FTP IMPLEMENTATION

The procedure that File Transfer Protocol works on is that of a client sending command codes with arguments, a server receiving those commands, processing the desired operation and then sending replies back to the client.

3.1 Command Codes

Commands are sent by the client and evaluated by the server in FTP. The standard FTP syntax requires that a command and argument be sent with a space in-between (represented by <SP>) followed by the carriage return and line feed characters (represented by <CRLF>).

USER <SP><username><CRLF>

This command makes up one part needed for the login procedure. The argument associated with the command is the username which identifies the current user of the client which is connecting to the server. Once sent, the server will compare this to its own list of registered users to see if the user has access to use the FTP server.

PASS <SP><password><CRLF>

This command makes up the other part required for the login procedure. The argument that follows the command provides the password for the previously sent username. The client sends this to the server and the server compares the provided password (with respect to the previously received username) with its own records and grants or rejects access.

PASV <CRLF>

This command contains no argument and indicates to the server that the client wants to use a passive data connection. The server receives this command and generates an IP address and appropriate port number for the connection. A tuple is sent back to the client which has both the address and the port. The server then listens on this port awaiting a connection. The client receives the reply and connects to the address and port contained in the tuple, forming the data connection.

PORT <SP><address-port><CRLF>

This command is used when the client wants to make an active data connection to the server. The argument of this command is a tuple that contains the client address and a port number for the server to connect to. The client then listens on this port number, awaiting the connection from the server. The server processes the command and then connects to the specified address and port number, creating the data connection.

LIST <CRLF>

This command is sent by the client to obtain the files in the server's current working directory. An existing data connection must be open for the list data to be sent back to the client. This command solicits 2 replies from the server on the control connection, before and after the data is sent.

PWD <CRLF>

This command is sent by the client to instruct the server to reply back with its current working directory.

The server obtains the path of its current directory and sends this information back concatenated with the reply code.

CWD <SP><pathname><CRLF>

This command is sent by the client to tell the server to change the working directory to the pathname provided as the argument. The server sends a reply and completes the action if the provided pathname is valid otherwise it sends only a reply code.

TYPE <SP><type-code><CRLF>

This command is sent by the client to inform the server of what type the uploaded/downloaded file will be. The argument provided is the type-code and can either be "A" or "I" for "ASCII" or "Image" (known as Binary) respectively. ASCII type files are usually text files or some programming source code files while Binary files are Adobe PDFs, videos, images and music.

SYST <CRLF>

This command is sent by the client to ask the server for the operating system the server is currently hosted on. The reply code is concatenated with the operating system name.

RETR <SP><pathname><CRLF>

This command is used by the client to indicate to the server the file that is to be retrieved. The argument is the pathname for the file that is to be downloaded. A data connection is needed to transmit the data to the client. This command also has 2 replies, one to check the connection and file type and the other to confirm once the file has been downloaded.

STOR <SP><filename><CRLF>

This command is used by the client to indicate to the server the file that is to be stored. The argument provides the filename that is to be uploaded. A data connection is needed to transmit the data to the server. This command also has 2 replies, one to check the connection and file type and the other to confirm once the file has been uploaded.

NOOP <CRLF>

This command is sent by the client to the server to ensure that the control connection is still active and that the client has not disconnected from the server.

DELE <SP><pathname><CRLF>

This command is sent by the client to specify to the server that a file must be deleted. The server checks for the existence of the file and deletes it. If the file does not exist, a different reply code is returned as opposed to if the operation executed successfully.

MKD <SP><pathname><CRLF>

This command is sent by the client to request the server to create a directory with the name that is passed as the argument.

RMD <SP><pathname><CRLF>

This command is sent by the client to request the server to delete the directory/folder specified in the argument.

CDUP <CRLF>

This command is sent by the client to request the server change from the current working directory to its parent directory.

QUIT <CRLF>

This command is sent when the client intends to end the current FTP session with the server. The control connection is closed by the server after the reply is sent.

3.2 Reply Codes

Table 1 below illustrates which reply codes were implemented in the server as responses to commands.

Table 1 : Reply Codes

| Code | Reply Message |
|------|--|
| 150 | File status okay; about to open data connection |
| 200 | Ascii/Binary mode enabled NOOP OK Changed directory successfully |
| 215 | Operating system name |
| 221 | Goodbye |
| 225 | Entering active mode |
| 226 | Closing data connection. Requested transfer action successful |
| 227 | Entering passive mode (h1,h2,h3,h4,p1,p2) |
| 230 | Login successful |
| 250 | Requested file action okay, completed |
| 257 | <pathname> is the current working directory Folder has been successfully created The requested folder has been deleted |
| 331 | Please specify password |
| 425 | Cannot open data connection |
| 501 | Syntax error in parameters or arguments |
| 502 | Command not implemented |
| 530 | Login incorrect |
| 550 | Requested action not taken. File/Directory unavailable |

Reply codes from FTP servers mean a multitude of different things. The codes have 3 digits with the first digit signifying a good, bad or incomplete command [1]. Although these are general, it is used to give a user an overall idea as to the result of the server. The second digit is used to specify syntax errors, requests for information, connection status, authentication and file system status [1]. The third digit is used to further specify the exact response from the server. These type of dynamic replies were not implemented in the server - instead, the most common were hard coded as responses to the commands. A more detailed description can be found in table 2 in Appendix A.

4. STRUCTURE OF CODE

The command codes that were implemented in both FTP client and server were presented in section 3.1 and the implemented reply codes that are returned by the server are presented in section 3.2. These two features cover the core aspect of the FTP client and server. The code structure of both programs is detailed below. Appendix B outlines the division of work between the two students.

4.1 FTP Client

The client has 3 sections of interest: the Graphical User Interface (GUI) and the components it makes use of, the socket layer and how it is used to establish the TCP connection for control commands or data transmission and the logic of how the client works.

4.1.1 Graphical User Interface - The GUI was created through the use of QT designer, a tool for building graphical user interfaces. A single window exists whereby the components are placed. The tool removes the need to create and define the orientation, size and position of each component through code which would have convoluted the source code and made it harder to follow. The GUI is then loaded up as a separate file and it is linked to the Python back-end programming file, which places on-click events on the buttons present. It is made up of 9 labels to identify features on the GUI, a TreeView component to display the current directory and sub-directories of the computer the client is run from, 2 PlainTextEdits to display the files of the server when LIST is called and to show the commands sent and replies received, 7 LineEdits to capture user input, 2 radio buttons to set the data connection to passive or active and 4 buttons to allow the user to interact with the server. Once a button is clicked, the respective function is called to communicate with the server.

4.1.2 Socket Layer - The Python socket library was used to establish the TCP connections for control and data. The code first defines the type of socket using 2 parameters: AF_INET and SOCK_STREAM. The latter parameter is used to define the socket of type TCP and the former is to associate the socket with Internet Protocol v4 addresses, one of the most used online standards, allowing the socket to connect to FTP servers on the internet. From here, the socket connects to a server address and its port (both are defined by the user). FTP can either work in passive or active mode, which is defined by how the data connection is established. In passive mode the socket object of the client only ever connects to the address and port of the server, while in active mode the client sends an address (usually the IP address of the computer running the client) and a random port. The socket object of the client then listens to this port and awaits an incoming connection, accepting one if it arrives.

4.1.3 Client Logic - An object oriented approach was considered and implemented for the client in the form of one class called FTP_Client. The object oriented design was desirable as it could group entire client commands as activities under one function. For example, to login, the user name, password, IP address and port of the server are required and fetching all this data from the GUI is done after a single button is clicked. Once clicked, the commands USER, PASS and LIST are then sent with the relevant parameters and the login procedure is completed. Instead of coding multiple functions for the sending of commands, activities are coded. As the activities of the client are button specific, it does not have to be continually looped but rather waits upon an interrupt signal from the push of a button on the GUI which has better control, unlike waiting on a loop.

4.2 FTP Server

The server has 3 sections of interest: the explanation of the socket programming used to establish the TCP connections by the server, the servers multi-threading ability and the logic of how the server works.

4.2.1 Socket Programming - The socket library for Python was used to implement the necessary TCP connections that provide the control connection and the data connection. The server sets up a socket at its local IP address and uses port 21, as this is the standard in the FTP protocol. A client can connect through the host address and port to set up the control channel. The data channel is set up differently for active and passive mode. The server attempts to make a connection to the client's socket through its host address and port number in active mode. Upon successful connection the data channel is established and ready to transmit data. In passive mode, the server provides the client with its address and an available port, which the client uses to establish the connection.

4.2.2 Multi-threading - Multi-threading the FTP server allows for multiple clients to connect to the server concurrently. This was implemented by inheriting the threading.Thread class. This allows for multiple threads to handle multiple users. The threads are closed once the session has ended and the client disconnects from the server. A new thread is created for each new user that establishes a connection with the server. The server is able to handle multiple clients and keep track of each of their working directories independently. Furthermore, the server prints to received commands and sent replies to the console with the responsible client.

4.2.3 Server Logic - The main() function in the code listens for incoming connections from clients. An FTPServer object is created for every connection that

is made by a client and is run in its own thread. The FTPServer class is structured such that it executes the run() function after creation and initialisation of the object. The run() function first sends a welcome message to the client on the control channel. An infinite while loop is then executed, which receives the commands from the client and calls the appropriate functions that relate to the client's command. The server logic first checks that the function is in the list of available functions and sends the client a "502 Command not implemented" reply if the function has not been implemented. If the function has been implemented, a function gets called to handle the client request. This is done through the Python function getattr(self, command) which allows the function to be called according to the string command sent by the client. If there is an argument associated with the command, the appropriate function is called and the parameter is passed to the function.

5. RESULTS

The FTP client can successfully log onto a server provided the correct username, password, server IP address and port are entered into the login screen. The client can enlist the files and folders in the server and can move back and forward between folders as well as create and delete folders on the server. The client downloads files from a server and stores them all locally in a folder called "Server.Downloads". This folder shares the same location as the client file. The client uploads files directly into whatever folder the server has opened. Different file formats such as images, video and music files can be uploaded and downloaded. The client can also switch between passive and active data connections at the user's discretion. The client was tested with existing FTP servers such as the Wits hosted server "ELEN4017.ug.eie.wits.ac.za" and another FTP server called "ftp.uconn.edu". The client worked with both of these servers.

The FTP server successfully implemented multithreading, which allowed multiple clients to connect to the server concurrently. The server was tested with both an existing FTP GUI client called FileZilla and the coded FTP client. The server was able to provide user login, listing of files, ability to connect in both active and passive mode, upload and download of multiple file formats, ability to change, create and delete directories, check if connection is still available, provide system information and quit the FTP session. The server was also successful in handling the current working directories of multiple users concurrently. The implemented server and client were able to communicate and transfer data successfully when run on two separate PCs on the same network in active mode. Furthermore, Wireshark was able to successfully recognise and sniff the FTP data packets on the localhost. Wireshark screenshots in Appendix C illus-

trate the successful communication between the client and server by showing the commands sent by the client and responses sent by the server.

6. CRITICAL ANALYSIS

Both programs adhere to the standards of RFC959 with the minimum implementation outlined in section 5.1 achieved with the exception of the MODE and STRU commands [1]. The default MODE is stream and the default STRU is file, which the server and client currently comply with. There are 3 different MODE types (stream, block and compressed) and 3 different STRU types (file, record and page) in FTP. Block, compressed, record and page are less common today as many servers do not accept these modes and structures any longer. For this reason, they were not implemented in the client or the server. The implemented programs encounter some runtime errors, particularly in the case of obscure user input. The server does implement some negative reply codes to mitigate some of these errors however, for example, if the user enters the CRLF characters in the GUI text fields, the program will crash. Full error prevention was not implemented due to time constraints. The basics of the client and server can still be achieved with relative ease. Both server and client choose ports randomly to use when setting up data connections. There is a possibility of these ports being unavailable and will cause the program to encounter an error. The probability of this occurring is low and should not detract from the user experience.

The TYPE command contains two further types: E and L for EBCDIC text and local format respectively. These have been phased out and are no longer used as ASCII and Binary/Image are enough to handle all current formats. As a result they are not implemented in the programs. The server and client were able to communicate with each other when running on two separate computers on the same network. However the data transfer only functioned in the active mode, with passive mode unable to establish the data connection.

7. CONCLUSION

This report detailed the implementation and features of a File Transfer Protocol (FTP) client and server. These programs adhere to the minimum implementation detailed in the RFC959 document, with the exception of the full capability of the MODE and STRU commands. Some additional commands to allow for folder creation, deletion and navigation were implemented to enhance the functionality of both client and server. The implemented programs do not contain full error handling and could cause the program to crash in certain edge case scenarios. The FTP client and server were able to communicate with each other over a wireless network when run on two separate computers, albeit exclusively in active mode.

REFERENCES

- [1] J. Postel, J. Reynolds. "*File Transfer Protocol (FTP)*", RFC 959, 1985.
- [2] J. F. Kurose, K. W. Ross. "*Computer Networking: A top down approach*", Boston: Pearson, 2007.
- [3] D. Parker. 2005. "*Understanding the FTP Protocol.*" Available: <http://techgenix.com/understanding-ftp-protocol/>. Accessed 2 April 2018.
- [4] SlackSite.com. "*Active FTP vs. Passive FTP, a Definitive Explanation.*" Available: <http://slacksite.com/other/ftp.html>. Accessed 2 April 2018.
- [5] V. John. 2014. "*Managed File Transfer and Network Solutions.*" Available: <http://www.jscape.com/blog/bid/80512/Active-v-s-Passive-FTP-Simplified>. Accessed 2 April 2018.

Appendix

A Detailed Reply code information

Table 2 : Detailed descriptions of implemented reply codes

| Code | Reply Message | Command | Description |
|------|---|----------------------|--|
| 150 | File status okay; about to open data connection | LIST RETR STOR | This reply code is sent to indicate to the user that a transfer of data is about to take place along the data connection. |
| 230 | Login successful | PASS | This reply indicates a successful user login to the client. |
| 227 | Entering passive mode (h1,h2,h3,h4,p1,p2) | PASV | Sent upon successful establishment of a data connection through passive mode, this reply is sent to the client. |
| 225 | Entering active mode | PORT | Sent upon successful establishment of a data connection through active mode, this reply is sent to the client. |
| 226 | Closing data connection. Requested transfer action successful | LIST RETR STOR | The reply is sent once any file data or list data is successfully transferred along the data connection. |
| 257 | <pathname>is the current working directory Folder has been successfully created The requested folder has been deleted | PWD MKD | Upon successful retrieval of the working directory, or creation of a directory this reply is sent to the client. |
| 250 | Requested file action okay, completed | RMD DELE CWD | The server will send this reply code when successful. |
| 200 | Ascii mode enabled Binary mode enabled NOOP OK Changed directory successfully | TYPE NOOP CDUP | This reply is sent upon successful completion of the following commands. |
| 215 | <Operating system name> | SYST | The server responds with the OS that it is currently running. |
| 221 | Goodbye | QUIT | This command closes the control connection. |
| 331 | Please specify password | USER | If the username is accepted, the server sends this reply to the client to request the password associated with this account. |
| 425 | Cannot open data connection | PASV PORT | If the data channel cannot be established, this reply is sent to the client. |
| 530 | Login incorrect | USER PASS | If the client supplied username and password is not recognized this reply is sent to the user. |
| 502 | Command not implemented | - | This is the reply that is sent back to the user if a command has not been implemented on the server side. |
| 550 | Requested action not taken. File/Directory unavailable | CWD MKD RMD | If the server is unable to complete the command, the following reply is sent to the client. |
| 501 | Syntax error in parameters or arguments | TYPE | If the client provides an invalid type the server will send this reply to the client. |

B Division of Tasks and Individual Contribution


Table 3 : Project contribution of group members

| Member | Task Allocation & Contribution |
|------------------|--|
| Kavilan Nair | FTP research |
| | FTP server design |
| | FTP server code implementation |
| | Testing of server against partner coded client |
| | Testing of server against existing client |
| | Report write up |
| Iordan Tchaparov | FTP research |
| | FTP client design |
| | FTP client code implementation |
| | Testing of client against partner coded server |
| | Testing of client against existing server |
| | Report write up |

Signatures of students in agreement with above information



Kavilan Nair



Iordan Tchaparov

C Wireshark Screenshots

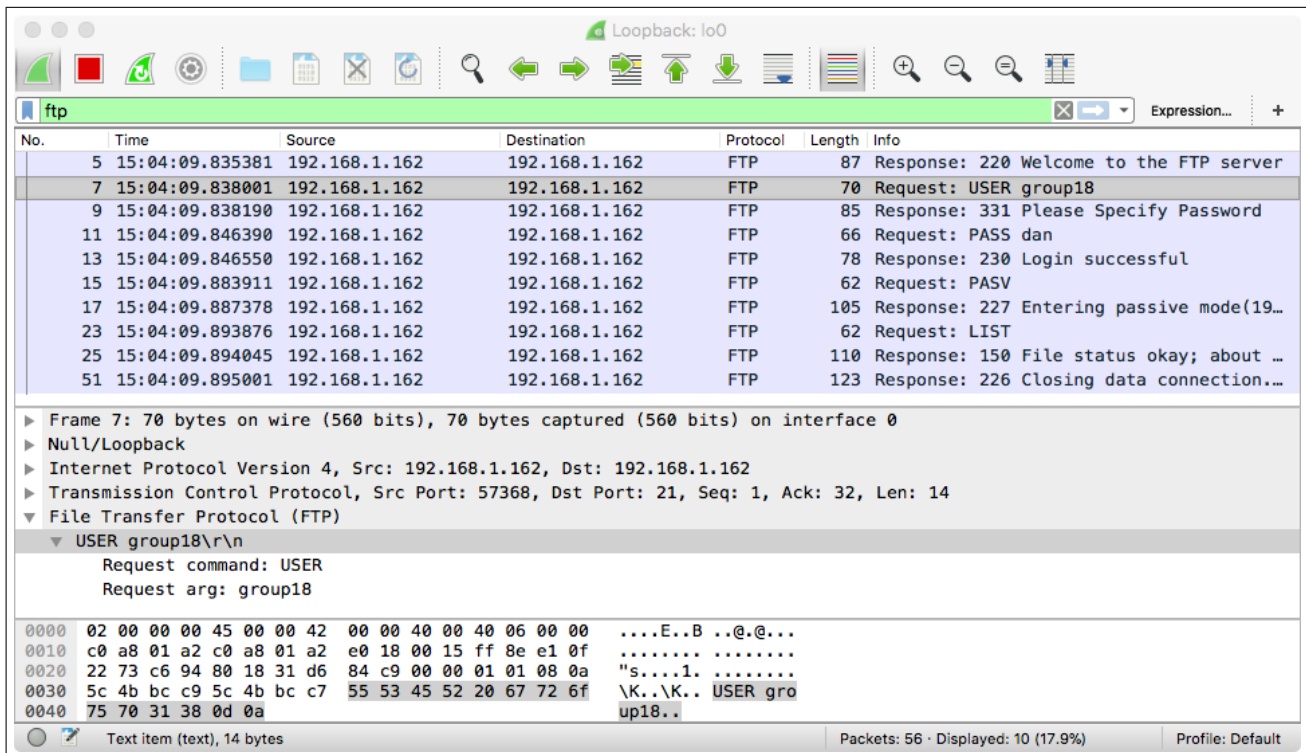


Figure 3 : Wireshark screenshot of login, and retrieval of list in passive mode

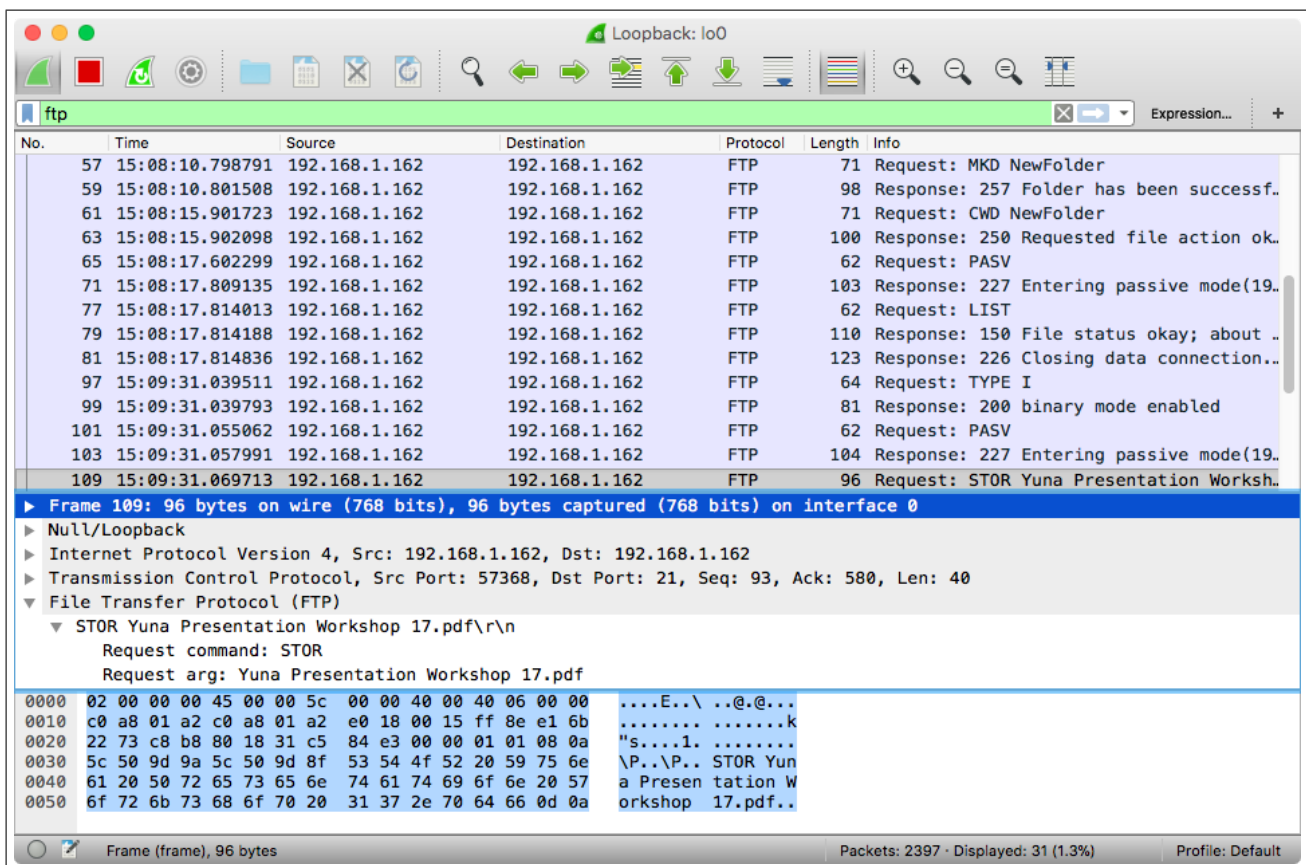


Figure 4 : Wireshark screenshot of creation of new folder, changing working directory to newly created directory, and upload of PDF document in passive mode

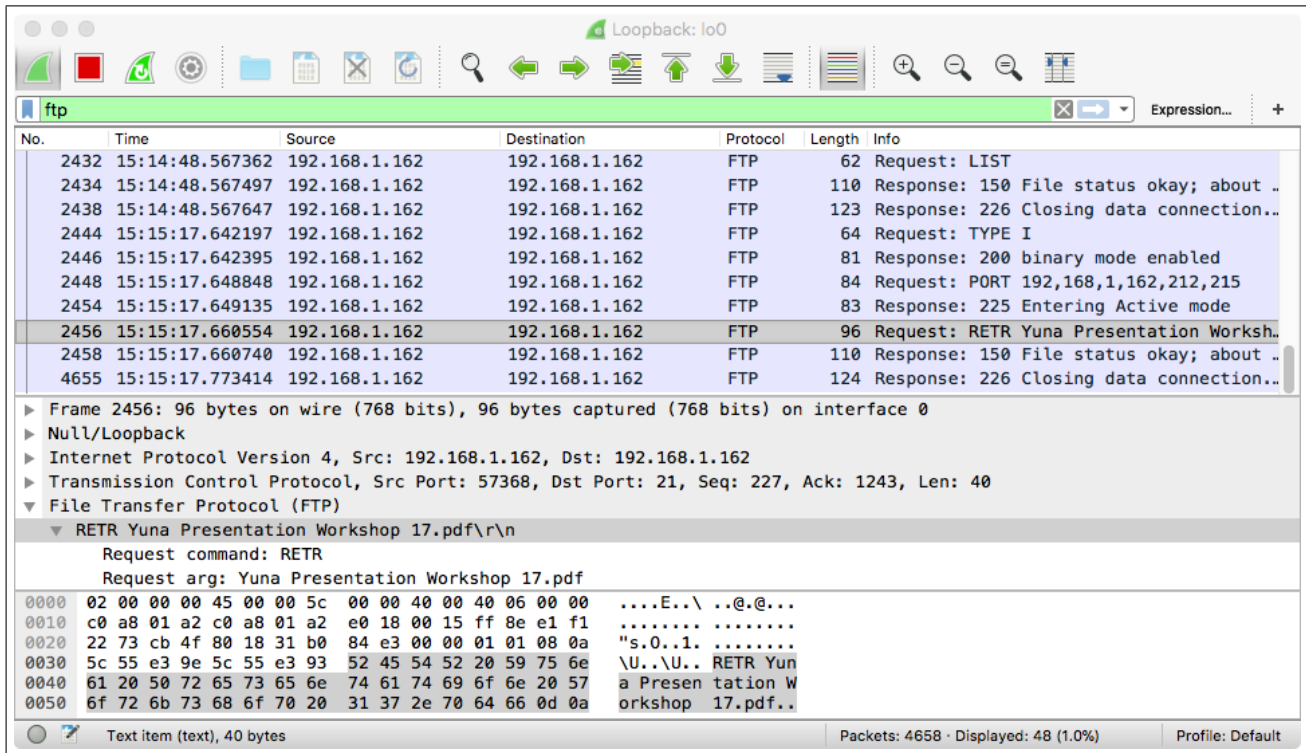


Figure 5 : Wireshark screenshot of deletion of file, change to parent directory and deletion of folder

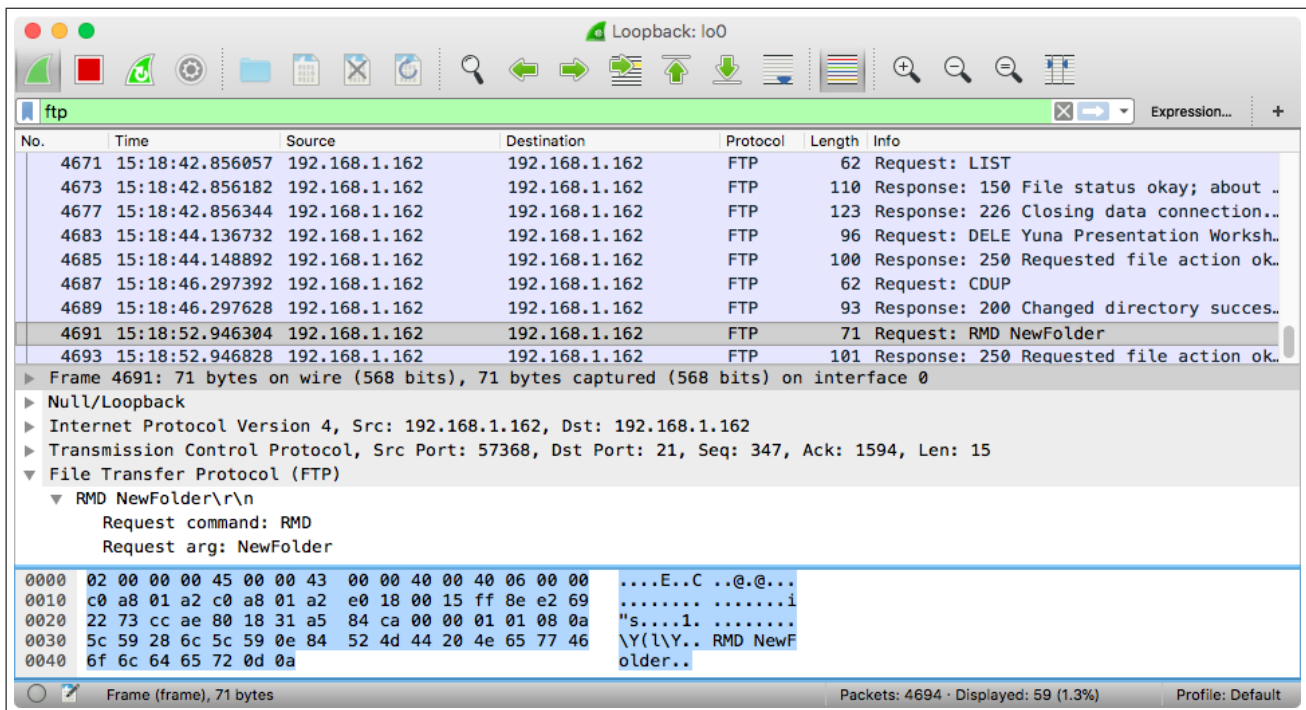


Figure 6 : Wireshark screenshot of download in active mode