ELEN4017 - DESIGN AND IMPLEMENTATION OF A FILE TRANSFER APPLICATION

Sasha Berkowitz (818737) & Lara Timm (704157)

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

Abstract: This report details the design and implementation of a client and server based file transfer application. Using the Python programming language a server was developed as well as a graphical client interface. The application's compliance to the RFC 959 file transfer specification is validated through the use of the Wireshark application, used to track system requests and responses. The implementation is deemed successful as more than the minimum required features are present and the client and server communicate and function well together, as well as the client to separate servers. Features such as login security and additional File Transfer Protocol functionality could be implemented in future versions.

Key words: File Transfer Protocol, python, RFC 959, socket, TCP

1. INTRODUCTION

File Transfer Protocol (FTP) is a protocol used to transfer files between two hosts over a TCP or IP network [1]. In its operation FTP makes use of two TCP connections, a control connection and a data connection. The control connection, used to send control information, is opened and remains open throughout the duration of the user session [2]. The data connection is non-persistent; a new data connection is established for each new file transfer [2]. The main objectives of FTP include the promotion of file sharing, to encourage the use of remote computers, to shield users from variations in file storage systems among hosts and to ensure reliable and efficient data transfer [3].

2. SYSTEM DESCRIPTION

An FTP client and server pair has been designed and implemented as per RFC 959 specifications [3]. By implementing the system in such a way, both the server and client are compatible with standardised servers and clients. The system is designed and tested on the Windows OS.

2.1 FTP Server

The FTP server is hosted locally, and can be accessed by an FTP client connecting from either the hosts computer, or from a computer running on the same network.

The server maintains a user repository for each registered client. To gain access to their remote repository, the user must be successfully authenticated using their unique username and password. Clients trying to connect without a registered username and password are not permitted to access the server file system.

To allow for simultaneous client connections, the server program is multithreaded. Each new connection is handled by a separate thread, and up to five simultaneous connections are allowed.

To communicate file, system and transfer status to the client, the server makes use of RFC 959 specified reply codes. These codes enable the client to detect errors and react accordingly. Further discussion of the FTP command and replies can be found in Section 3.

Unimplemented features: In accordance with the RFC 959 minimum requirements, functionality for *record* and *page* file structures, as well as for *block* and *compressed* transfer modes were not implemented. The project group deemed theses features unnecessary as file transfer is still possible in their absence.

2.2 FTP Client

The FTP client is composed of two working parts, a Graphical User Interface (GUI) and a logical FTP client. The user interacts with the GUI which instructs the FTP client to interact with the FTP server. In this way the interface is separated from the logic layer, following the separation of concerns principle.

2.2.1 Client Logic

The FTP client is hosted locally and can connect to a server hosted either on the client's computer or over a LAN connection. To connect to the server, the client must know the IP address and port on which the server is listening for a TCP connection.

The responsibility of the FTP client is to interact with the server in accordance with FTP standards. The client translates raw data received from the GUI and formats it to comply with the RFC 959 specification. In this way, the client can not only interact with the designed server, but also with a range of standard FTP servers.

It is the responsibility of the client to handle errors that are not relevant to the server. Such errors include trying to upload a file that does not exist in the clients file system and handling errors to do with incorrect formatting of client commands.

2.2.2 Client Interface

The FTP client is handled by a simple GUI containing entry fields for the client's credentials, windows displaying files hosted in local and remote directories, a terminal response window and a custom command entry field, as shown in Figure B1 in the Appendix.

Once logged in, the user can navigate through both their remote and local file systems, as well as delete and transfer files between them. The server's response is displayed in a terminal-like window.

The client is able to disconnect from the server through the GUI. Alternatively, closing the developed application terminates the client/server connection.

In addition to the functionality provided by the GUI, the client is able to enter custom FTP commands through the provided entry field.

Unimplemented features: In accordance with the server implementation, the client has no implementation for file structures and transfer modes other than *file* structure and *stream* transfer mode.

Functionality to create new files, rename files either locally or remotely and transfer files between directories on either system is not implemented. Additionally, directories cannot be uploaded or downloaded.

3. FTP COMMAND/REPLY OVERVIEW

The format of FTP reply messages is designed to ensure that requests and actions are well synchronized, and that the client is informed about the status of the server at all times [3].

FTP commands are sent via the persistent TCP control connection, established on port 21 of the server [2]. FTP dictates that a standard command and reply format must be followed for all communications on this connection [3]. The command and reply formats are demonstrated below, where <SP> indicates a space and <CRLF> indicates the end of line sequence, '\r\n'.

Command:	<CMD><SP><argument><CRLF>
Reply:	<code><description><CRLF>

There are 5 categories of FTP replies, characterised by the first digit of the three digit reply code [3]. In the developed file transfer application, at least one reply was implemented from each category. A description of the categories as well as list of the implemented commands and their associated server responses are detailed in Appendix A.

4. DETAILED FEATURE IMPLEMENTATION

Both the client and the server are implemented using Python 2.7. Essential Python modules utilised in both the FTP client and server, are the `socket` and `os` modules [4, 5]. The `socket` module is responsible for the creation of all TCP sockets used to make connections and handle all communications between the client and server. The `os` module enables the file transfer application to interface with the file system of both the server and client.

4.1 Server

The server program is multi-threaded, making use of Python's `threading` module, to allow for simultaneous client connections. When the server code is executed, a `serverThread` object is created which listens for and accepts new client connections. When a connection is made, a `clientThread` object is created and assigned to a new thread. The thread is passed the client's connection socket and the address bound to that socket as argument [5].

Within each `clientThread` object, the server continuously accepts commands (using `recv()`), performs the desired action and returns a reply (using `send()`). Based on the reply, the client is able to determine the success/failure of the command as well determine changes in transfer parameters and the status of the server.

4.1.1 Access Control Commands

Authentication: Once connected, a client must be authenticated. The `USER` and `PASS` commands are expected with plaintext arguments. For a successful authentication, the username and password must match those contained in the file *userdata.txt*. In the case that a user repository does not already exist, one is created upon successful authentication.

Navigation: To navigate the server file system, the `CWD` and `CDUP` commands are implemented. `CWD` takes a directory name or path as argument. `CDUP` is a specialised `CWD` command where the destination is one directory up in the directory tree. The server implementation of `CWD` makes use of `CDUP` if the arguement is the name of the working directory.

To exit the control connection, the `QUIT` command is used. The reply received by the client causes the client to shut down the connection socket and free the port.

4.1.2 Transfer Parameter Commands

All parameters controlling data transfer have default values, the following commands need only be called to change the defaults [3].

Data Connection: A data connection can be opened in passive or active mode. The former is default, and involves a request for the server to listen on an available data port for a data connection [3]. `PASV` initiates a server response containing the internet address and port to which the client must connect to transfer data. The active mode command, `PORT`, has as arguements the internet address and port to which the server must connect to transfer data [3]. In both cases, the port data is transmitted in two 8-bit fields. To recreate the 16-bit port address, the following formula is used: $port = 256 * byteUpper + byteLower$.

Transfer Mode: As per the minimum requirements specified by RFC 959, only the default transfer structure and transfer mode are implemented. `STRU` allows the transfer structure to be set to file [`F`] and `MODE` allows the transfer mode to be set to stream [`S`]. To cover a large range of file formats, both ASCII (text files) and Binary (media) representation types are implemented. The client determines the argument of the `TYPE` command as either ASCII [`A`] or binary [`I`] depending on the extension if the file to be transferred.

4.1.3 Service Commands

File Transfer: Using the `RETR` and `STOR` commands, the user can download file from and upload files to the server's remote file system.

The `RETR` implementation ensures the file exists before opening it in the appropriate read mode specified by the `TYPE` command. A data connection is opened by `open_dataSocket()` and the requested file is read and subsequently transmitted on the data connection in 1024 byte chunks. When the transmission is complete, the data connection is closed by `close_dataSocket()`.

A `STOR` command causes a file to be generated and opened, in the correct write mode, in the server file system. As with a `RETR` request, a data connection is opened to recieve the requested data. While data is being received, it is written to file. The data connection is closed when that transfer is completed.

File System Status: The `PWD` and `LIST` commands allow the client to know the status of the server's file system at any point in time. `PWD` requests that the server return the full path of the client's working directory as a parameter in the reply message. A `LIST` request prompts the server to send a file listing of the current working directory. To generate the file statistics, the `stat` module and the `createItemString()` function are used. For each entry in the directory list, the function is called and the returned string is appended to a string containing the information for the full directory. This complete string is sent over a TCP data connection, in ASCII mode, to the client.

File System Action: The `MKD`, `RMD` and `DELE` commands enable the client to make and delete a directory as well as delete a file on the server file system. All three functions take a file name, directory name or path as argument. If a directory is deleted, the entire contents of the directory is removed.

The `NOOP` command fulfils no purpose other than to solicit an OK response from the server.

The `AUTH` command was implemented after system testing with the standard FTP client, FileZilla. `AUTH` requests expect authentication details to be encrypted before transmission. In the implementation, `AUTH` simply responds with a message requesting log in with `USER` and `PASS`.

4.2 Client

The client side of the application is run with a simple GUI, developed using Python's `TkInter` module. The main function of the GUI is to view both local and remote file systems and working directories simultaneously. On the local side, this is done by using the `os` module's `getcwd()` and `listdir()` functions. On the remote side, `PWD` and `LIST` commands are sent to the server. The results are then displayed using TkInter `ListBox` widget and are selected and navigated through using click and double-click functions.

Connection To and Disconnection From The Server: To connect to the server, the client provides their username and password, the server address and the server port. Upon press of the 'CONNECT' button, the login process is initiated using the `USER` and `PASS` commands. Upon successful log-in, the user is able to view and interact with their server-hosted file directory.

The client is disconnected from the server upon click of the 'DISCONNECT' button, using the `QUIT` command. If the client closes the application window without disconnecting, the `QUIT` command is run automatically.

File Transfer: The user is able to upload a file from their local system or download one from their remote repository on press of the 'UPL' or 'DWNL' buttons respectively. A passive connection is requested using the `PASV` command and the file type is confirmed by passing the file name to the `TYPE` command.

File & Folder Creation & Deletion: Files cannot be created locally or remotely. Directories can be created remotely by pressing the 'NEW DIR' button. A pop-up window is displayed where the user can enter a desired directory name, which is passed to the `MKDR` command. The directory is created and shown in the remote file system.

The user can delete local and remote files using the

'DELETE' button. Locally, the `os` module is used to determine if the selection is a file or directory. If it is a directory, a popup informs the user that directories cannot be deleted (refer to Figure B2). If it is a file, the `os` module is used to delete it. Remotely, the selection is once again checked to be a file or directory before running either the `RMD` or `DELE` commands.

Custom Command Line: Included in the GUI is a 'custom command' entry field, to allow users to enter custom FTP commands. In this way, users are not limited by the functionality that the GUI allows. In addition, the terminal window provided prints out responses received by the server from both the custom commands and processes run through the GUI.

5. DIVISION OF WORK

The application's server code, as well as the client's `STOR`, `RETR`, `PORT`, `PASV` and `LIST` commands were developed by project partner Lara Timm. The client's interface, its integration and the remainder of the client code was developed by Sasha Berkowitz. As such, sections of this report pertaining to the above mentioned were written by their respective programmers. In addition, the introductory, FTP command & reply, results and critical analysis sections were written by Lara and the abstract and conclusion by Sasha.

6. RESULTS

All functionally on the client and server side is must be tested. This includes testing the FTP client and server with one another as will as with standard FTP clients and servers.

Wireshark is used to test the developed file transfer application. Appendix C contains screenshots of the described interactions.

6.1 Implemented FTP client and server interaction

Interactions between the implemented FTP client and server are seamless. All implemented functionality works as expected.

The client connects to the server and logs in. The client receives the directory listing and is able to navigate the file system. New directories can be created and deleted and files can be deleted. The data connection works in both passive and active mode (`PORT`) and both binary and ASCII files can be uploaded and downloaded. For screenshots depicting this interaction refer to Figures C1 and C2.

6.2 FTP Client interaction with standard FTP server

Similar to that of the FTP client/FTP server interaction, interfacing with the standard server proves to be successful for all implemented functionality.

Figure C3 depicts client authentication, directory listing and navigation, the initiation of a passive data connection, binary file download and file deletion.

Figure C4 depicts the creation and deletion of directories, ASCII file upload, printing and navigating up a directory, the initiation of an active data connection, the request of an OK response and closing the control connection.

6.3 Standard FTP Client interaction with FTP server

Compared to the interaction with the standard FTP server, the chosen standard FTP client is far less compatible with the implemented server. Basic FTP commands are successful. FileZilla is able to authenticate the user, open data connections and upload and download files without difficulty. Commands beyond the minimum implementation scope result in confusing results.

Unlike the implemented FTP client, FileZilla requests that the client provide a TLS or SSL authentication for secure data transfer. FileZilla does not always make use of the `CDUP` command to go up one directory. Rather a `CWD` command with the path of the working directory as argument is used. Upon issuing a request for file download, FileZilla may request the size and modification time of the file using `SIZE` and `MDTM` requests. Although the server does not have implementation for these commands, files transfer is still successful.

FileZilla appears to store a history of the previous directory listing, so performing a change directory up request does not cause FileZilla to send a change directory command or request a new directory listing. As a result the working directory of the server is not updated and errors arise. All the problems associated with directory traversal may be due to FileZilla expecting a directory list in a different form than the one which the server provides. The command `LIST -a` may expect a different format than the one provided, yet is not further investigated.

7. CRITICAL ANALYSIS

7.1 Successes

The system can be considered a success as is it well implemented and fully functional. This is proven by the results obtained and presented in Section 6.. The system fulfils all requirements of a file transfer application. These include:

- Both the client and server meet the minimum requirements of FTP specified by RFC 959 [3].
- The client utilises a simple UI which allows the

user to perform all required tasks.
- The server is multi-threaded as to allow simultaneous client connections and file transfers.
- The server maintains repositories for all registered users.
- Both the client and server are able to interact with standard FTP clients and servers and perform all tasks required for file transfer.
- The file transfer application is able to transfer files of varying file types.
- The file transfer application allows client and server communications when hosted on the same computer as well as hosted on different computers on the same network.

Rather than making use of high-level FTP libraries, the implementation makes use of low-level sockets. Providing functionality beyond the scope of the minimum requirements is considered a success as it adds to the usability of the system. Additionally, the server implementation of more than five reply codes enables a greater understanding of the server status and actions taken.

Another success is the accuracy of the GUI's representation of the back-end file system. With each operation the view is refreshed in order to display the current list of files hosted in both local and remote repositories. As a result the client cannot attempt to perform operations on files which do not exist and are able to get visual feedback on new file and folder creation.

7.2 Limitations

Only the 'stream' data mode has been implemented. Unlike the 'compressed' and 'block' modes it is unreliable when determining the time at which a connection should be closed. Another limitation is that transfer of files and folders with duplicated names overwrite those pre-existing, possibly causing the loss of important data on both local and remote systems.

7.3 Future Improvements

The developed application does not integrate well with the chosen standard FTP client. In future a different client should be considered, or the reasons they differ should be investigated in order to adapt methods and have a more seamless integration.

Security, such as hash encoding passwords or implementing SSL, could be provided for user login.

Additionally, more FTP functionality could be implemented, such as features omitted as described in sections 2.1 and 2.2.2.

8. CODE STRUCTURE AND REQUIREMENTS

Code Structure: The server code is contained within a single file, *FTPserver.py*, and defines the two threading classes used to handle simultaneous client connections. All FTP functionality is implemented through methods defined in the `clientThread` class. The client implementation is divided into logical and interface components. The interface makes use of the `clientLogic` class to initiate and handle all interactions with the server and is contained within the *interface.py* file.

Requirements: The application has been developed to be run using Python 2.7. In order to run the GUI, the `TkInter` module needs to be installed. In order to run the application, first the server has to be run using the following command:

```
python FTPserver.py
```

Following, the client is run using the following command.

```
python interface.py
```

9. CONCLUSION

The development process of a File Transfer Protocol application, built in Python 2.7, has been described, as well as details on its implementation. The development was considered a success as it meets more than the minimum requirements laid out by the RFC 959 specification. In addition, correct protocols followed were confirmed through the use of Wireshark, and a GUI was implemented successfully displaying both local and remote file systems. In future security could be added to the login process and additional FTP functionality could be implemented.

REFERENCES

[1] "FTP for Beginners." URL `https://www.wired.com/2010/02/ftp_for_beginners/`. Last Accessed: 18/03/2018.

[2] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 2012.

[3] J. Postel and J. K. Reynolds. "RFC 959: File Transfer Protocol.", Oct. 1985. URL `ftp://ftp.internic.net/rfc/rfc959.txt`.

[4] "10.1. os.path Common pathname manipulations." URL `https://docs.python.org/2.7/library/os.path.html`. Last Accessed: 23/03/2018.

[5] "17.2. socket Low-level networking interface." URL `https://docs.python.org/2.7/library/socket.html`. Last Accessed: 23/03/2018.

Appendix

A Implemented FTP Commands/Replies

The five categories of FTP replies and their explanations are presented below:

1** Positive Preliminary reply	Requested action initiated, expect reply before sending new command.
2** Positive Completion reply	Requested action completed, new command can be sent.
3** Positive Intermediate reply	Command received, server waiting for further information.
4** Transient Negative Completion reply	Command not accepted, action did not take place. Error is temporary and action may be requested again.
5** Permanent Negative Completion reply	Command not accepted, action did not take place. User discouraged from repeating same request.

A description of the All implemented FTP commands as well as the expected replies are detailed within table 1 below.

Table 1: Implemented FTP commands and the server reply they invoke

Command	**Command Description**	**Server Replies**
USER	The username input to the GUI is sent to the server for authentication	331 User name okay, need password. 332 Need account for login.
PASS	The password input to the GUI is sent to the server for authentication	230 User logged in, proceed. 332 Need account for login. 530 Not logged in. Password invalid.
CWD	Changes the working directory on the server. Argument is the name of the directory to change to.	250 Requested action okay. Working directory changed. 550 Requested action not taken. Directory does not exist.
CDUP	Go up one directory on the server. User cannot go further up than their base directory.	250 Requested action okay. Working directory changed. 550 Requested action not taken. Permission denied.
QUIT	Closes the control connection between the client and server.	221 Service closing control connection.
PORT	The client specifies a port for the data connection. Arguments are the IP address and port on which the client is listening for a connection.	200 Port command successful.
PASV	Requests that the server listens on an available port for a client data connection. The clients connects to the port specified in the server response.	227 Entering passive mode (*IP address*, *Port*)
TYPE	Specifies the type of the file which is to be uploaded to or downloaded from the server. ASCII and Binary types are implemented.	200 Switching to Binary mode. 200 Switching to ASCII mode. 504 Command not implemented for that parameter.
STRU	Specifies the structure of the file which is to be uploaded to or downloaded from the server. File structure is implemented.	200 Switching to File structure mode. 504 Command not implemented for that parameter.
MODE	Specifies the transfer mode of the file which is to be uploaded to or downloaded from the server. Stream transfer mode is implemented.	200 Switching to Stream transfer mode. 504 Command not implemented for that parameter.

Command	**Command Description**	**Server Replies**
RETR	A copy of a file existing on the server is sent to the client over a data connection. The command argument is the name of the file to be retrieved.	150 Opening data connection. 226 Closing data connection. File action successful. 550 Requested action not taken. File unavailable. 451 Requested action aborted: local error in processing. 425 Use `PORT` or `PASV` first.
STOR	A copy of a file existing on the client's computer is sent to the server over a data connection. The command argument is the name of the file to be stored.	150 File status okay. Opening data connection. 226 Closing data connection. File action successful. 550 Requested action not taken. File transfer unsuccessful. 425 Use `PORT` or `PASV` first.
MKD	Makes a new directory in the server's working directory. The argument specifies the name of the new directory.	257 "*currentDirectory**newDirectory*" created. 550 Requested action not taken. Directory already exists.
RMD	Deletes a directory, and all its contents, in the server's working directory. The argument specifies the name of the directory to be deleted.	250 Requested file action okay, directory deleted. 550 Requested action not taken. To delete file use `DELE` 550 Requested action not taken. Directory does not exist. 550 Requested action not taken. Permission denied.
NOOP	Prompts the server to send an Ok response	200 Command okay.
AUTH	Implemented for compatibility with the standard FTP client, FileZilla	530 Please log in with `USER` and `PASS`.

B Client Interface

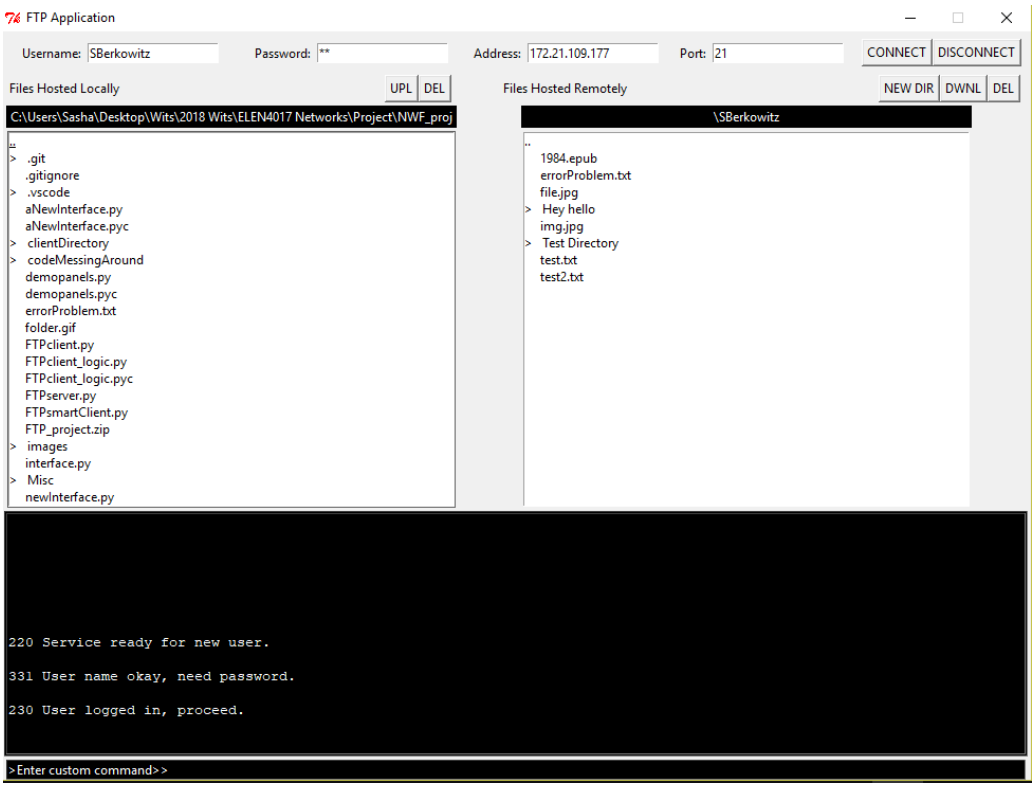

Figure B1: Screenshot of the client GUI once a user has logged in

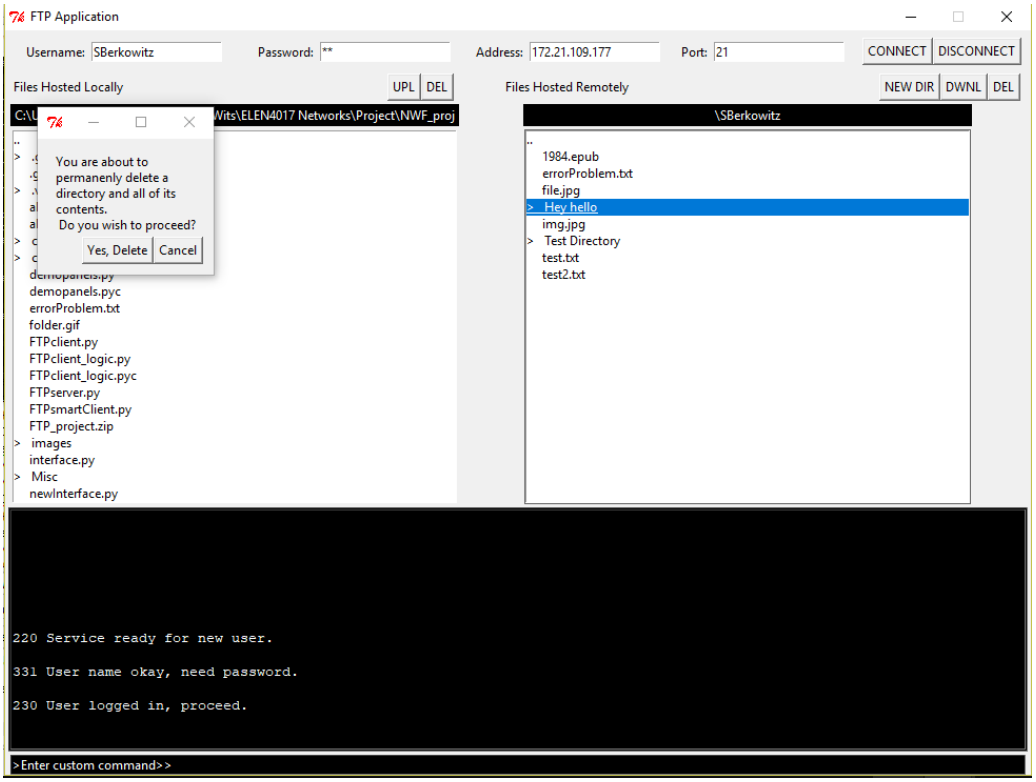

Figure B2: Screenshot of the client GUI after an attempt to delete a directory

C Testing Results

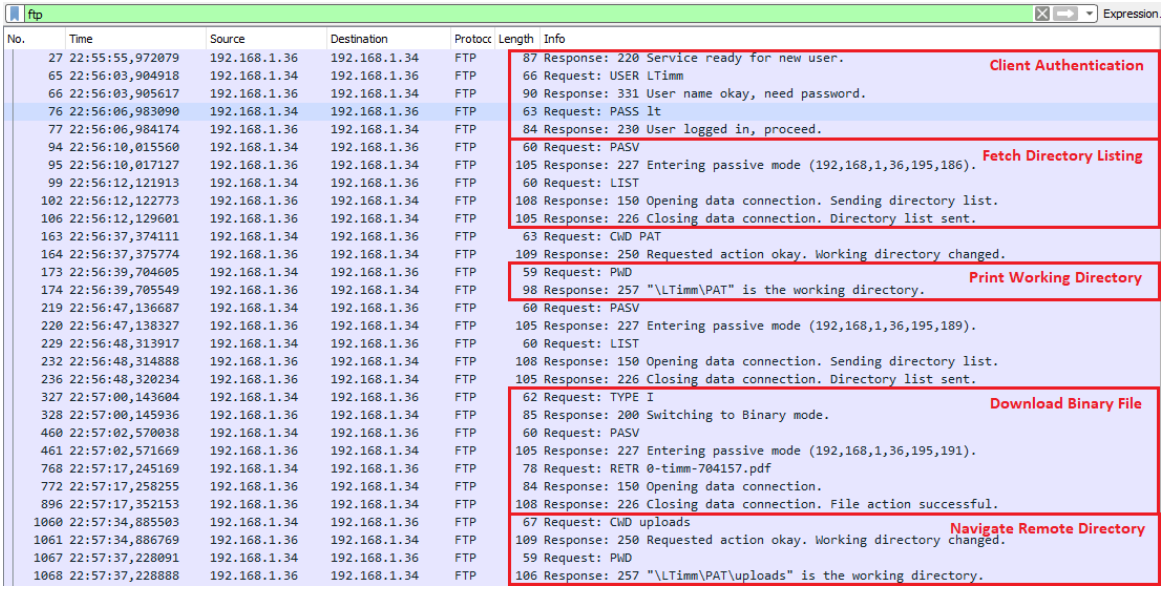

Figure C1: Wireshark screenshot 1 of functionality testing with implemented FTP server

Figure C2: Wireshark screenshot 2 of functionality testing with implemented FTP server

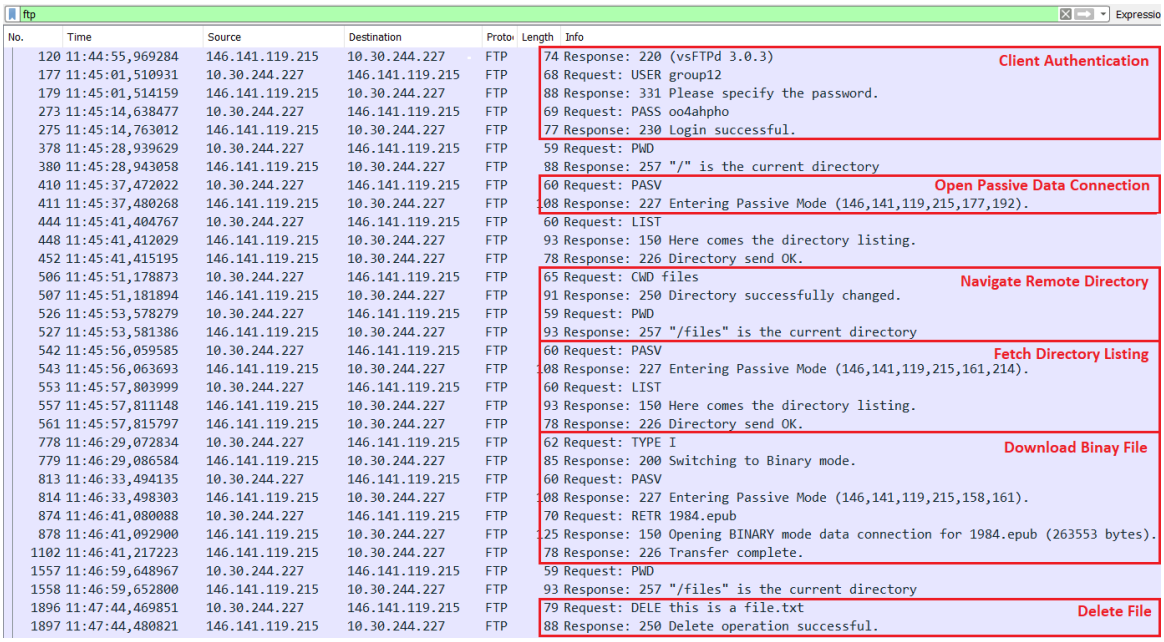

Figure C3: Wireshark screenshot 1 of functionality testing with standard FTP server

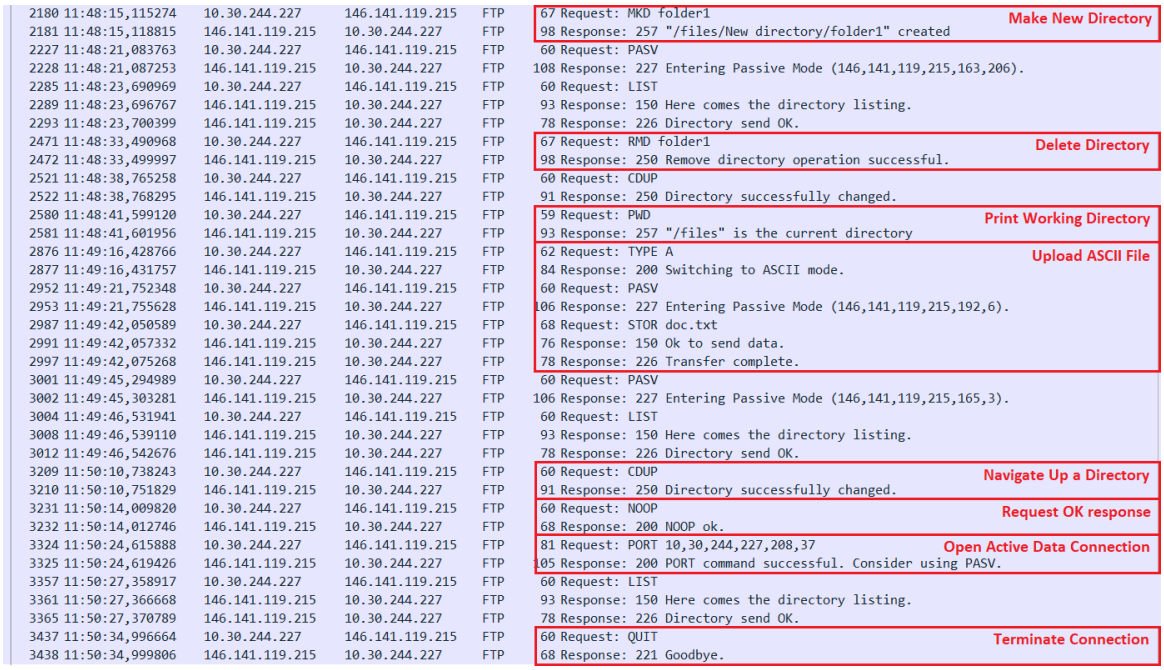

Figure C4: Wireshark screenshot 2 of functionality testing with standard FTP server