A Minimal FTP Implementation in Python

Junaid Dawood (1094837), Xongile Nghatsane (1110680)

***Abstract*—This report discusses a minimal implementation of the file transfer protocol (FTP) in Python including both an FTP server and client. This implementation is achieved without the use of any high-level FTP libraries and instead manually implements typical FTP commands and responses using low-level socket modules. The implemented server achieves a minimal FTP server implementation and some additional functionality such that it is able to work with common FTP clients such as FileZilla. The implemented FTP client employs a simple GUI in order to facilitate common FTP activities. The client is also able to detect potential encoding type mismatches and automatically prevents this through communications with the server, for increased ease of use.**

I. Introduction

The file transfer protocol (FTP) is an application layer protocol which defines a set of rules for achieving file transfer over TCP/IP [1]. Despite its age, the protocol is still commonly used, and has received extensions so as to improve on some of its initial shortcomings in areas such security and the ambiguity of certain response formats.

This report discusses a minimal implementation of an FTP client and server in Python. This implementation is constructed from only low-level socket components: all protocol messages, responses, encoding, etc. are achieved manually in this implementation. A critical analysis of the implemented FTP client and server is presented within this report as well as a critique of the protocol itself and the various extensions which have been made over the years.

II. Background

FTP implements a typical client-server model. This refers to a working model in which the client initiates communication with the server which is continually listening for connections [2].

FTP incorporates a two-layer model: 1) a control layer over which coordination messages are sent and 2) a data layer for the transferring of requested data such as files and directory structures. Typical coordination messages consist of client authentication, retrieving the name of the current directory, and setting up data connections. These messages consist of either commands or responses; both are formatted as telnet strings terminated by a carriage-return line-finish (CRLF). The general format for both commands and responses is: $< command/response >< args >< CR >< LF >$

The minimal implementation required of an FTP server is the ability to correctly interpret and respond to the following commands [1]:

- USER - Supply user name to server for login.
- QUIT - End current session.
- PORT - Set up active data connection.
- TYPE - Define the type of data being sent.
- MODE - Change transfer mechanism.
- STRU - Change file structure.
- RETR - Retrieve file from server.
- STOR - Upload file to server.
- NOOP - No operation - 'keep-alive' command.

A. Types of Session

RFC 959 specifies that an FTP server can either allow for anonymous sessions, normal sessions, or both. Anonymous FTP, as specified in RFC 1635 [3], is used for scenarios in which actual authorisation and credential management is unnecessary and wasteful, such as public access to non-sensitive files. Within anonymous FTP, users typically log in using the user name 'anonymous' and their emails as passwords. Normal FTP, on the other hand, requires a registered user name and password in order to log in and download files.

B. Types of Data Connection

As stated earlier, the data connection is used to transfer files and other information such as directory listings upon the receipt of an appropriate control message such as RETR. The RFC 959 standard specifies two ways types of data connection: passive and active.

Passive connections are the default type of connection: this refers to the server creating a socket and listening for a connection from the client in order for data transfer to begin. Active data connections work in the reverse manner: the client is the one to create a socket and listen for the servers connection prior to transfer beginning. Neither of these connection types is concerned with whether an upload (STOR) or download (RETR) is occurring; they simply specify which party must listen for an incoming connection and which party must attempt to connect.

C. Types of Transmission

The RFC 959 standard specifies three transmission types for data transfer over any type of data connection. The types of transmission include stream, block, and compressed. This is altered through the use of the MODE command with appropriate arguments.

The default mode is stream mode in which data is continuously streamed and the end of a transmission is indicated with the closing of the data connection.

The other two modes: block and compressed, are not commonly used and are not implemented by the majority of FTP clients. In block mode, data is sent in a series of blocks with header fields indicating supplementary information and the length of the data in the block. Block mode is typically unused since stream mode is simpler to implement and can almost always be used instead.

Compressed mode compresses input data by identifying repeated bytes and using a method similar to LZ encoding to replace these repeated bytes with markers [4]. Compressed mode is typically unused use since compression could be done at a file level before transmission; notwithstanding the compression that would occur at lower network layers in any case.

D. Types of Transferable Data

FTP specifies three file structures within the RFC 959 standard: file, record, and page. According to RFC 959, files are simply a continuous sequence of bytes with no internal structure, records are made up of sequential records, and pages are made up of indexed independent pages. The default structure is file; the desired structure can be modified using the STRU command. Of the three file structures, the overwhelming majority of typical FTP servers and clients support only the file structure.

E. Types of Encoding

FTP allows for the encoding of transmitted data in various formats. This is decided by the FTP client through the use of the TYPE command. The arguments for this command include: 'A - ASCII Text', 'I - Image (binary)', 'L - local format', and 'E - EBCDIC text'.

Typically, however, only binary and ASCII encoding are used: ASCII for text files and binary for all other files. Due to some text using different encoding e.g. UTF-8, ASCII encoding is not always applicable for text files and thus it may be safer to use binary encoding for all files.

III. Success Criteria

To be considered a sufficient implementation the server must implement the functionality defined within the minimal implementation section of RFC 959. Naturally, the implemented client must be able to exercise this minimal server implementation. It must, therefore, be able to send commands to the server and correctly interpret responses so as to achieve typical FTP client-server functionality.

Lastly, the implemented server should not differ from the specifications presented within RFC 959, such that it can be used with common FTP clients. That is, the FTP standard must not be modified within the server implementation such that it is only able to work with the created client.

IV. Implemented Functionality

The created FTP server implements both the minimal functionality described in RFC 959 and some additional functionality which is generally expected by typical FTP clients, such as FileZilla [5]. Moreover, the server maintains separate storage locations for each user, allowing an administrator to view and edit all of these locations. A brief catalogue of the implemented commands and associated responses is shown in the table below.

TABLE I: Implemented FTP commands and responses.

Command	Responses
USER	331
PASS	200, 430, 530
QUIT	221
PORT	501, 200
PASV	227
NOOP	200
TYPE (I and A)	200, 501, 504
STOR	250, 425, 426, 501, 550
RETR	250, 425, 501, 550
SYST	200
PWD	257
LIST	250, 226, 425, 530
CWD	200, 530, 550
STRU (F)	200, 501, 504
MODE (S)	200, 501, 504
MKD	257, 530, 550
SYST	200
Invalid/Not Implemented	502

The server itself exists as a command line application which requires no input after initialisation. After the server is initialised, it may be connected to by any number of clients including an implemented GUI client and typical GUI FTP clients (FileZilla, etc.).

The implemented GUI client allows for a user to exercise the functionality offered by the commands listed earlier, through the use of buttons and input boxes.

Together, the server and client are able to achieve typical FTP functionality: uploading and downloading of files, navigating directories, and printing server directory contents. These are discussed in greater detail along with other aspects of functionality within the various subsequent subsections. Screenshots of the implemented server and client 'in-use' can be found in Appendix A.

A. Creation of Connection

The implemented server can connect to multiple clients, due to a multithreaded implementation. The server sends a 220 reply code to a new client upon connection and requires that clients log in using USER and PASS after successfully connecting.

B. Login and Authentication

User login and authentication are achieved by the implemented FTP server. This involves the ability to respond to the USER and PASS commands. Credentials are simply stored in a server-side text file. Credentials supplied by clients are checked against username-password pairs within this file in order to either confirm or deny clients access to other commands.

Response codes 331, 530, 200, and 430 are implemented on the server. Code 331 is sent once a username has been received, informing a client that a password must still be sent. The 530 code is sent when a client attempts a command other than PASS once a USER command has already been received. The 200 reply code is used to indicate that the credentials supplied by the client are correct. Lastly, the 430 response code is sent if authorisation has failed: incorrect information was sent in either the USER or subsequent PASS command.

C. Creation/Requesting of Data Connections

Both passive and active data connections are allowed for by the implemented server i.e. it is able to correctly handle PASV and PORT commands.

For the active approach, the implemented client is able to obtain the user's desired IP address and port required for the PORT command and sends this to the server. The server then creates a data socket after parsing the information sent by the client.

The parsing consists of splitting the comma delimited list of input parameters so as to obtain an IP address and port number. The server creates a data socket and responds with a 200 reply code if the client sent valid arguments, else a 501 reply code is sent.

RFC 959 specifies the following format for the PORT command's arguments: (ip1,ip2,ip3,ip4,p1,p2). The *ip* fields simply translate to an IPv4 address and the port number is calculated from p1 and p2 by $portNo = p1 * 256 + p2$. For example: $192, 168, 1, 1, 250, 5 \iff 192.168.1.1 : 64008$.

The implementation of the passive approach is similar, with the exception that the server sends the arguments (ip1,ip2,ip3,ip4,p1,p2) in its response to the PASV command, since it is the client which will initiate the connection. A 227 reply code is sent if the server accepts the client's request.

A data connection is only used once per transmission operation. Once a data connection is used for a RETR, STOR, etc. command, it is closed and a new connection must be requested before further transfer requests are serviced.

D. Creating, Changing and Printing Directories

Allowing for the changing of directories is a common feature of FTP servers, despite not being specified in the minimal implementation of RFC 959. The server is able to respond to CWD commands from clients for this purpose. Related to CWD are the MKD and

PWD commands which create new directories and print the current working directory respectively.

This is done by storing a 'current working directory' string on a per-client basis, performing all file and directory related operations e.g. STOR, RETR, LIST, etc. relative to the stored string. This approach is chosen over the use of the *os.chdir* method as this would affect the working directory used for all connected clients. Allowing for this to occur would cause errors with reading common files such as the credentials text file.

The server responds with a 200 reply code if the directory requested by the client exists, else a 500 error code is sent.

Generating a response to a PWD command simply makes use of the stored string mentioned earlier; this is sent within a response message with the 257 response code.

The MKD command is implemented using the *os.mkdir* command, allowing for both relative and absolute paths as its input. The command is only executed if the proposed directory is possible to create and if the directory does not already exist. If the proposed directory cannot be created, the server responds with a 550 reply code, else a 257 reply code is sent. For both MKD and CWD, if a client, besides administrators, specifies a location which is not a part of their repository a 530 error code is sent.

E. Uploading and Downloading Files

With the uploading and downloading of files typically being the end goal of the use of FTP; this functionality is ultimately key to the successful implementation of the FTP client and server.

The implemented server is able to respond to the STOR and RETR commands and allows for the use of the TYPE command for switching between binary and ASCII encoding for transfer. The server responds with 200 if binary or ASCII is requested, otherwise a 501 (invalid type) or 504 (not implemented) reply is sent.

The server response to the STOR and RETR commands takes into account the TYPE specified by clients in order to encode sent data to the correct format. Type mismatch errors are prevented on the client side through the automated re-sending of the TYPE command, with correct arguments, when a mismatch is detected.

If a type mismatch occurs during an attempted upload or download, due to the use of manual commands on a different client, the server responds with a 550 reply code. Otherwise, the server will typically respond with a 250 reply code.

In addition, when a non-existent file is requested using a RETR command, a 550 reply code is sent. Similarly, when a RETR command's parameters specify a directory and not a file a 550 reply code is sent. STOR commands can also elicit a 426 error code if a write buffer could not be opened for the specified file.

For both RETR and STOR, if a blank argument is sent for the required filename/path, a 501 error code is sent. In addition, if a transfer is requested prior to PASV or PORT being sent, then a 425 error code is sent to indicate no data connection has been created.

F. Retrieving Directory Listing

Similarly to the CWD command, the LIST command is required for common FTP functionality despite not being a part of the RFC 959 minimal implementation. The LIST command can be sent with or without a directory path as an argument. If no path is sent, the returned directory listing will default to the current working directory of the session.

A list of files and directories is obtained for this string using the *os.listdir* function. For relative paths, this function is called relative to the stored working directory string mentioned earlier.

The *bin/ls* format [6] is used to format the list of files and directories that is sent to the client. This format is chosen due to its popularity and compatibility with many FTP client applications.

Either the 226 or 250 reply code is sent when the LIST command is sent to the server, depending on whether an active or passive data connection is used. If the directory requested does not exist, a 501 error code is sent. In a similar way to RETR and STOR, a 425 error code is sent when requesting a listing without first requesting a data connection. Like MKD and CWD, if a client tries to access an area outside of their repository, a 530 error code is sent.

G. Unimplemented Functionality

With reference to the minimal implementation described earlier, the implemented server and client lack some aspects of functionality. Specifically, the TYPE, STRU, and MODE commands are not fully implemented on the server, and thus there was no reason to implement them fully on the client.

For TYPE, only ASCII and binary encoding are supported, for STRU only file, and for MODE only stream. This is thought to be justified by the fact that almost all common FTP clients are unable to make use of more than the implemented transmission mode, file structure, and encoding types. It is generally advised that MODE commands other than *MODE S* be rejected; similarly *STRU F* need only be implemented [7].

The applicability of binary encoding to all file types is typically the reason for other types remaining unused. ASCII encoding is implemented despite this due to its commonality within typical FTP clients. All other encoding types are almost entirely unused and exist only to support legacy hardware.

V. Error Handling

The implemented client and server achieve simple error handling. The server does this through type checking and validation of arguments, sending error response codes if invalid requests are made. These error responses have been described in the previous section.

The client attempts error prevention instead of error handling i.e. the UI attempts to forbid invalid commands and invalid execution order through the enabling and disabling buttons etc.

A. Server-side Error Handling

Server-side error handling involves type checking, and general validation of parameters sent with commands.

1) Type Mismatches

Despite the implemented client having measures in place to prevent such issues, the implemented server implements simple type checking to cater for clients which allow for manual input. This refers to scenarios in which clients attempt to download (or upload) files which cannot be encoded using the specified format e.g. requesting a JPEG file after sending a *TYPE A* command. The server simply does a concurrency check between the selected TYPE and the requested file, returning an error code and aborting the transfer if an incompatibility is detected.

2) Bad Arguments and Unknown Commands

Bad arguments refer to arguments which do not comply with RFC 959 or are just generally invalid, such as attempting to download a file which does not exist. In both scenarios, the request is rejected with an error code.

Commands that cannot be handled by the implemented server are simply rejected with a 502 response message.

3) Client-side Error Handling

Client-side error handling consists of the prevention of error conditions through input validation and restricting command usage.

4) Prevention of Incorrect Ordering of Commands

The implemented GUI client prevents the user from executing commands in an incorrect order, such as attempting to send commands before first connecting to a server. This is done by controlling the order in which commands can be sent by selectively disabling and enabling UI components. This involves switching button 'clickability', allowing for clicks only when prerequisites have been met e.g. only allow for login once connected.

5) Prevention of Type Mismatches

The implemented client identifies the attempted transmission of files based on their extensions and the current type of encoding, as per the use of the TYPE command. The client remedies such scenarios through an automated re-sending of the TYPE command if a compatibility error is detected. For example, if the user attempts to download an image whilst the encoding type is ASCII, the client will automatically instruct the server to change the encoding type to binary, thereafter sending the RETR command to download the image.

VI. Testing and Results

Testing is conducted both through the developed client and through the use of FileZilla as an FTP client. The developed server was confirmed to work correctly with the FileZilla client and the developed GUI: allowing for uploading, downloading, listing of directories, etc., as well as the handling of common errors. In addition, the GUI client was tested and verified to work with the remote server *speedtest.tele2.net*. This is confirmed by the screenshots presented within Appendix A. Additionally, Wireshark capture screenshots of requests, responses, and file transfers are included in Appendix B.

VII. Division of Work

An attempt was made to keep work sharing as even as possible across all tasks, including non-development ones. The table below gives a high-level description of the division of work between group members.

TABLE II: Work division between group members.

Task	**Completed by**
Researching FTP and related protocols.	Both
Implementation of PORT, CWD, MKD, LIST, PWD, PASS, SYST, RETR(active), STOR(active), NOOP, QUIT, and USER commands and responses.	Junaid Dawood
Implementation of GUI client.	Junaid Dawood
Implementation of PASV, RETR(passive), STOR(passive), TYPE, STRU, MODE commands and responses.	Xongile Nghatsane
Server and client type checking for correct transmission encoding.	Xongile Nghatsane
Testing and Wireshark capture.	Junaid Dawood
Report writing and editing.	Junaid Dawood

VIII. Code Structure and Implementation

There are three important classes within the current implementation: the FTP server, FTP client, and GUI classes. The FTP client class exposes functions for facilitating connections and sending commands to an FTP server. These functions are named according to the FTP command they implement. Where necessary, these functions take in argument to be included in the commands e.g. file names for RETR.

The GUI class makes use of the Tkinter package for implementing GUI elements. The GUI class uses the FTP client class by composition, calling its functions with user input obtained through Tkinter pop-up windows.

Lastly, the FTPServer class exists in much the same way as the client class, exposing functions that relate directly to FTP commands. However, these functions are automatically called by the server upon receipt of commands from a client. Client commands enter as strings at a central point within the class. These command strings are subsequently parsed and the class's functions are called in accordance with commands extracted from the strings.

The diagram below offers a high-level description of all the components within the system and their interaction.

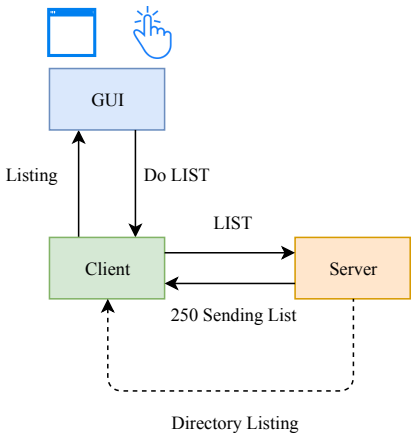

Fig. 1: High-level diagram of system component interaction.

A. Making use of Code

The implemented client and server can simply be started within two terminal windows; with the client making use of a GUI once started. Once the server is live, the user can then connect, log in and perform other actions using the buttons and pop-ups present in the GUI.

The *ttkthemes* package is required by the GUI client for its styling and must be installed, however, Tkinter itself is almost always included with default Python installations. Additional instructions can be found in the supplied *readme* file.

IX. Discussion

This section serves as a critical analysis of the developed FTP client and server, as well as the FTP protocol itself.

A. Shortcomings of Developed Client and Server

The main shortcoming of the developed GUI client is the lack of automated tools present in clients such as FileZilla; such as 'click-to-download' functionality. Although, to some extent, the implemented client is not meant to hide the underlying FTP commands for the purpose of this exercise.

The ability for a user to decide whether to overwrite or rename an uploaded or downloaded file is lacking within the implementation presented. The current implementation will simply overwrite files without prompting the user.

The implemented server is thought to be largely sufficient in its functionality. As discussed earlier, missing functionality related to legacy commands is not thought to significantly affect the quality of the server implementation. Of course, the server's ability to respond to a greater number of commands, especially modern additions to FTP, would be an improvement.

B. Shortcomings of FTP

This section discusses the shortcomings of the FTP protocol, with reference to literature on the subject and observations made during the development of the FTP client and server. This critique is made with specific reference to the RFC 959 standard. In addition, the concerns addressed by subsequent expansions of the FTP standard via RFC memos are also discussed.

1) No Standard Formatting for Some Commands

Whilst attempting to implement a response to the LIST command on the FTP server it was found that the RFC 959 standard does not specify a standard structure for this response. This has lead to the emergence of several standards all of which vary slightly in formatting, ordering of information, and identification markers for files and directories.

RFC 959 states the following with reference to the LIST response format "Since the information on a file may vary widely from system to system, this information may be hard to use automatically in a program, but may be quite useful to a human user." [1]. The problem with this view is that the format is, in fact, often used automatically within FTP clients. Sending a LIST response in a format unknown to FileZilla basically renders the client inoperable in terms of its GUI based file downloading technique. As a result, the use of user-defined LIST response formats has been requested of FTP clients, such as FileZilla [8].

The MLSD command, as presented in the RFC 3659 extension of the FTP protocol [9], exists specifically to remedy the issues encountered with the automated usage of the LIST command's response within FTP clients. This extension proposes a standardised format for the response to the MLSD command.

2) Security Concerns

Due to its age, the RFC 959 standard does not specify any means for the encryption of protocol messages and responses. Indeed, the standard does specify the USER and PASS commands, but it does not require that user names and passwords be encrypted by the client before being sent to the server.

There exist many workarounds for the purpose of achieving file transfer in a secure manner. A common method is to disregard FTP and make use of secure protocols to transfer files such as HTTPS or SCP. That said, there exist modern additions to the FTP protocol which enable the secure transmission of information. Specifically, the RFC 2288 and RFC 4217 protocol extensions describe methods for achieving secure transmission [10], [11]. The RFC 2288 extension introduced new commands for providing strong authentication, integrity, and confidentiality in the control and data channels, whereas RFC 4217 details the use of FTP over TLS.

3) IPv6 Compatability

As per the description of the format of the PORT command and PASV response, it is clear that the RFC 959 standard was designed with only IPv4 in mind. This is problematic due to the exhaustion of IPv4 addresses due to the rapid expansion of the internet [12]. Fortunately, the RFC 2428 extension of FTP proposes the EPSV and EPRT commands specifically for this purpose: allowing for an IPv6 compatible alternative to the PASV and PORT commands respectively[13].

4) Networking Complexities

The default FTP data port is not often used by typical FTP client and server applications. Instead, these clients generally send a PASV or PORT command prior to every data transfer request. As a result, FTP will use a variety of different ports per client session, which are often randomised within a range. Due to this randomisation, firewall configuration can prove complex on the client side. One solution is to simply forward all ports within the range of ports used by the FTP server [14].

The second networking complexity occurs due to network address translation (NAT). If the server and client do not exist on the same local network, then there will likely be stages of NAT within their connection. The server will generally have a static external IP, so the issue typically arises on the client side. Generally, a client is not aware of its external IP, only its internal one, which is provided through NAT. Hence the use of active data connections using the PORT command is made difficult since the correct IP address cannot be sent to the server. A common solution is to abandon active connections and to always use PASV to generate data connections. A superior alternative would be to use more sophisticated NAT software which has the ability to alter outgoing PORT commands in order to provide the correct IP address [14].

X. Recommendations for Future Work

It is recommended that the implemented FTP server be upgraded in its functionality so as to comply with the several proposed extensions of the RFC 959 specification. The most important additions are thought to be compatibility with IPv6 and implementation of the standardised LIST command i.e. MLSD.

The FTP client could be improved in terms of its usability, minimising reliance on text input and instead using click based functionality e.g. FileZilla's 'click-to-download' functionality.

XI. Conclusion

A discussion of the FTP protocol has been presented, along with its identified shortcomings and the current workarounds and solutions. This discussion is informed by a low-level implementation of an FTP server and client in Python. The developed server is able to respond to most common FTP commands, barring obsolete functionality and legacy support. The developed client employs a simple GUI interface and is capable of typical FTP client functionality. The developed client and server were tested using Wireshark and were verified to work correctly and respond to erroneous input appropriately. Future work should focus on improving the implementation so as to comply with the proposed extensions to the RFC 959 specification.

References

[1] J. Postel and J. Reynolds, "Rfc 959: File transfer protocol," Tech. Rep., 1985.

[2] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 2012, ISBN: 0132856204.

[3] P. Deutsch, A. Emtage, and A. Marine, "Rfc 1635: How to use anonymous ftp," 1994.

[4] *Lempel_ziv_notes.pdf*, http://www-math.mit.edu/~shor/PAM/lempel_ziv_notes.pdf, (Accessed on 19/04/2019),

[5] *Filezilla - the free ftp solution*, https://filezilla-project.org/, (Accessed on 03/04/2019),

[6] *The bin/ls format.* https://cr.yp.to/ftp/list/binls.html, (Accessed on 08/04/2019),

[7] *Cr.yp.to/ftp/type.html*, http://cr.yp.to/ftp/type.html, (Accessed on 10/04/2019),

[8] *#9078 (formatting list output) - filezilla*, https://trac.filezilla-project.org/ticket/9078, (Accessed on 06/04/2019),

[9] P. Hethmon, "Rfc 3659: Extensions to ftp," Tech. Rep., 2007.

[10] S. J. Lunt, "Rfc 2288: Ftp security extensions," 1997.

[11] P. Ford-Hutchinson, "Rfc 4217: Securing ftp with tls," Tech. Rep., 2005.

[12] *Free pool of ipv4 address space depleted - the number resource organization*, https://www.nro.net/ipv4-free-pool-depleted, (Accessed on 09/04/2019),

[13] M. Allman, S. Ostermann, and C. Metz, "Rfc 2428: Ftp extensions for ipv6 and nats," 1998.

[14] *The file transfer protocol (ftp) and your firewall / network address translation (nat) router / load balancing router*, https://www.ncftp.com/ncftpd/doc/misc/ftp_and_firewalls.html\#problems, (Accessed on 09/04/2019),

APPENDIX A

This appendix contains screenshots of the created GUI FTP client and the command line FTP server. Included screenshots also verify the interaction between the implemented server and the FileZilla FTP client.

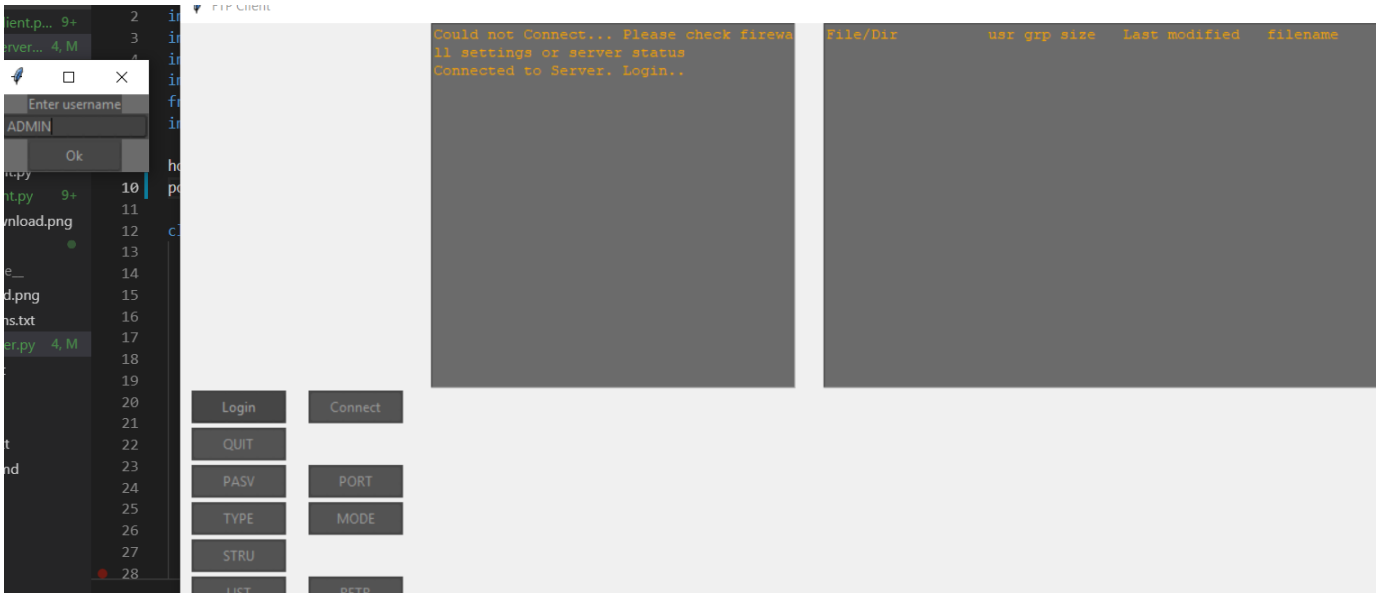

Fig. A.1: Initial login using GUI client.

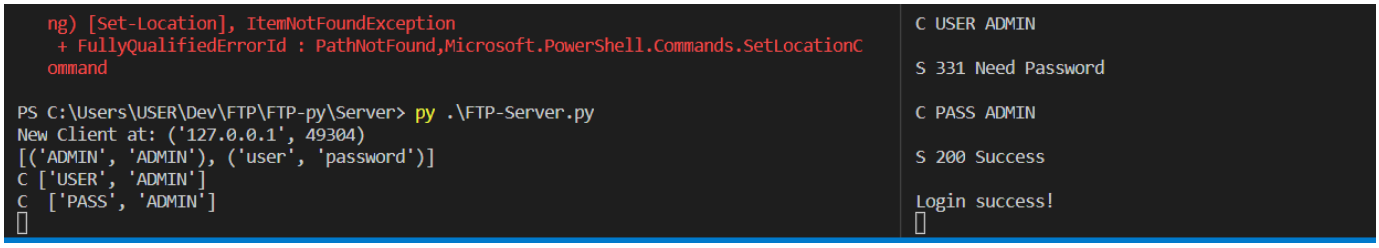

Fig. A.2: Side-by-side server-client login interaction command line printout.

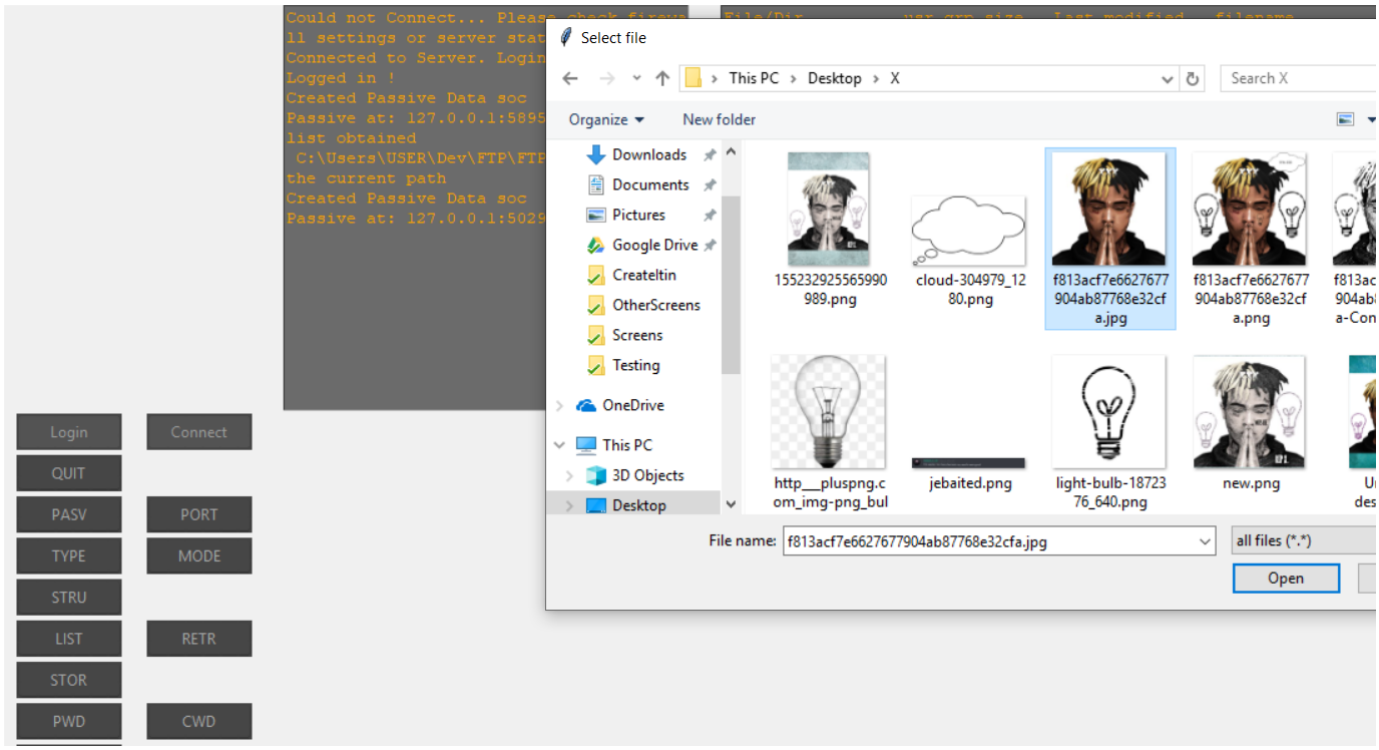

Fig. A.3: Uploading file using file dialog.

```
drwxr-xr-x 1 def def 0   Apr     16      15:14    __pycache__

Received ['PWD', '']
Received ['PASV', '']
Initiating passive data port
127,0,0,1,196,119
Received ['TYPE', 'I']
Received ['STOR', 'f813acf7e6627677904ab87768e32cfa.jpg']
```

```
.jpg
C TYPE I

S 200 Type Altered

C STOR f813acf7e6627677904ab87768e32cfa.jpg

S 250 Accepted
```

Fig. A.4: Side-by-side server-client STOR interaction command line printout.

```
File/Dir          usr grp size    Last modified    filename
-rw-r--r-- 1 def def 3364    Apr    10    12:45    download.png
-rw-r--r-- 1 def def 28    Apr    16    05:20    extensions.txt
-rw-r--r-- 1 def def 20904    Apr    20    06:33    FTP-Server.py
-rw-r--r-- 1 def def 13    Apr    10    12:45    logins.txt
drwxr-xr-x 1 def def 0    Apr    16    15:14    __pycache__
```

Fig. A.5: Directory listing in GUI client.

Remote site: C:\Users\USER\Dev\FTP\FTP-py

- FTP-py
  - .git
  - .vscode
  - Client

Filename	Filesize	Filetype	Last modified	P...	Owner/G...
..					
README.md	47	MD File	2019/03/29 10:58:00	-...	def def
minimal.txt	559	Text Doc...	2019/03/29 11:12:00	-...	def def
LICENSE	1 086	File	2019/03/29 10:58:00	-...	def def
.gitignore	1 307	Text Doc...	2019/03/29 10:58:00	-...	def def
Server		File folder	2019/04/20 08:36:00	d...	def def
Client		File folder	2019/04/17 14:04:00	d...	def def
.vscode		File folder	2019/04/01 21:35:00	d...	def def
.git		File folder	2019/04/20 08:33:00	d...	def def

Fig. A.6: Directory listing in FileZilla, demonstrating correct use of bin/ls format by server.

```
Status:    Logged in
Status:    Starting download of C:\Users\USER\Dev\FTP\FTP-py\minimal.txt
Status:    File transfer successful, transferred 542 bytes in 1 second
Status:    Starting upload of C:\Users\USER\Desktop\DocScan.pdf
Status:    File transfer successful, transferred 8 904 681 bytes in 1 second
Status:    Retrieving directory listing of "C:\Users\USER\Dev"...
Status:    Directory listing of "C:\Users\USER\Dev" successful
```

Fig. A.7: Various commands working in FileZilla and server interaction.

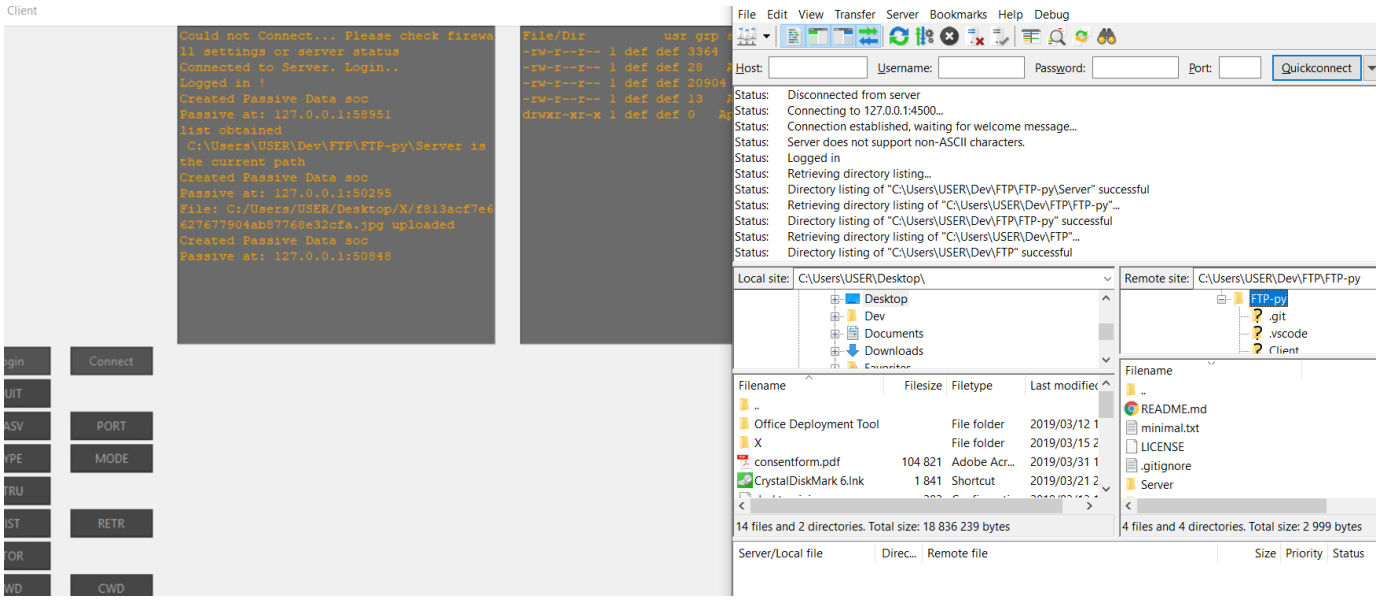

Fig. A.8: Simultaneous use of FileZilla and GUI client.

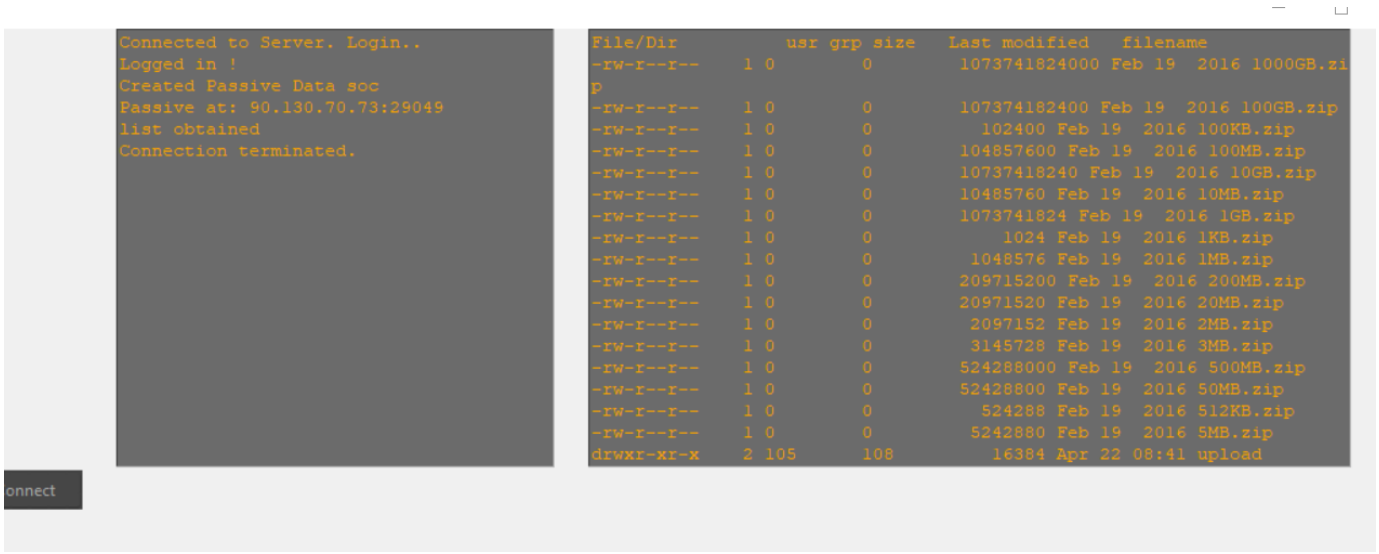

Fig. A.9: Use of client with remote server (speedtest.tele2.net).

APPENDIX B

This appendix consist of Wireshark screenshots captured during a local network FTP session. Included screenshots consists of initial connection, login, data connection creation (passive and active), downloading a file, uploading a file, retrieving a directory listing, changing directories, and making directories. Responses to bad arguments and invalid commands are also included.

Connection and Login

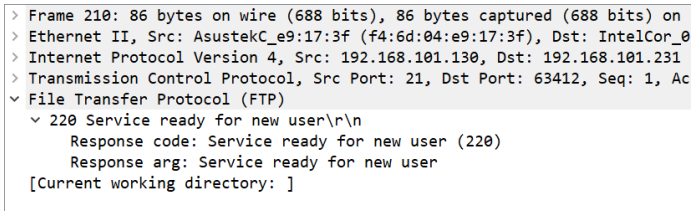

Fig. B.1: Initial connection greeting.

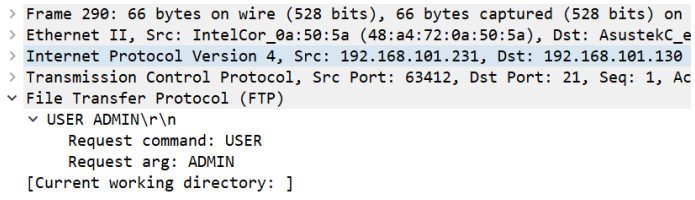

Fig. B.2: Sending USER command.

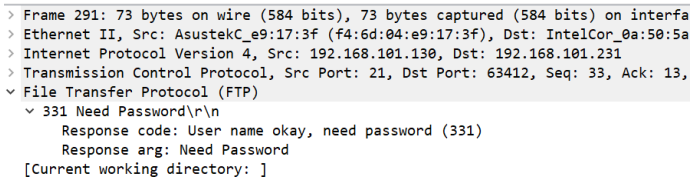

Fig. B.3: Server replies 'password needed' to USER.

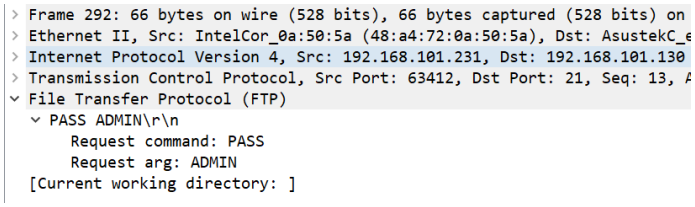

Fig. B.4: Sending PASS command.

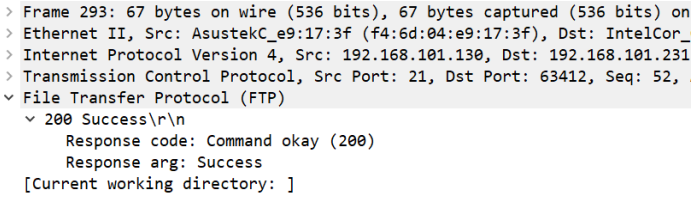

Fig. B.5: Correct username and password response.

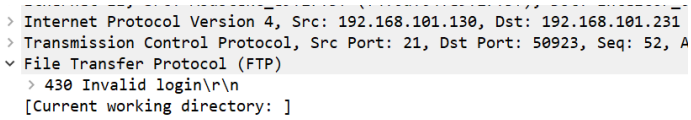

Fig. B.6: Bad username and password combination response.

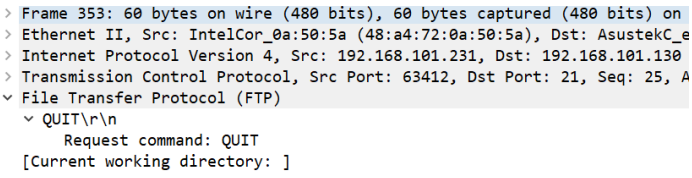

Fig. B.7: Sending QUIT command.

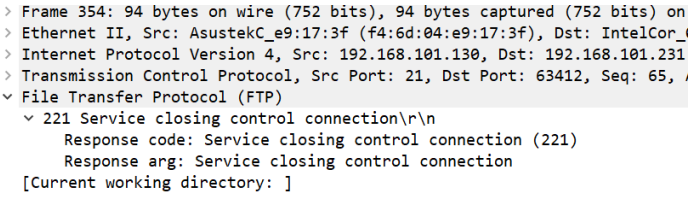

Fig. B.8: Server response to QUIT command.

PWD

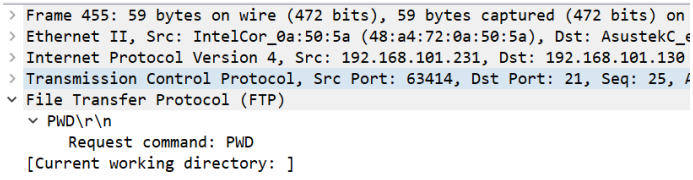

Fig. B.9: Sending PWD command.

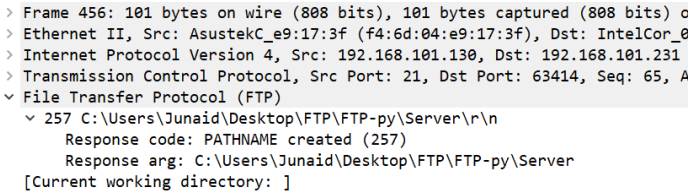

Fig. B.10: Server PWD response.

CWD

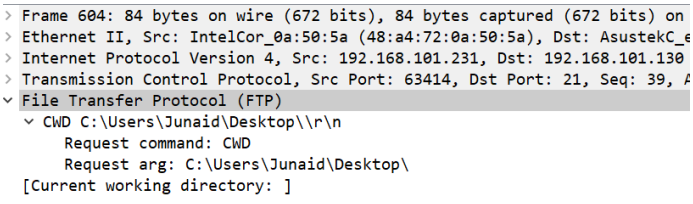

Fig. B.11: Sending CWD command.

```
> Frame 605: 105 bytes on wire (840 bits), 105 bytes captured (840 bits) o
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_0
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63414, Seq: 174,
v File Transfer Protocol (FTP)
  v 200 directory changed to+C:\Users\Junaid\Desktop\\r\n
      Response code: Command okay (200)
      Response arg: directory changed to+C:\Users\Junaid\Desktop\
  [Current working directory: ]
```

Fig. B.12: Server CWD response for existing directory.

```
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_0
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50923, Seq: 270,
v File Transfer Protocol (FTP)
  > 550 Requested action not taken\r\n
  [Current working directory: ]
```

Fig. B.13: Server CWD response for nonexistent directory.

MKD

```
> Frame 487: 63 bytes on wire (504 bits), 63 bytes captured (504 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 30, A
v File Transfer Protocol (FTP)
  v MKD dad\r\n
      Request command: MKD
      Request arg: dad
  [Current working directory: ]
```

Fig. B.14: Sending MKD command.

```
> Frame 488: 116 bytes on wire (928 bits), 116 bytes captured (928 bits) on i
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_0a:5
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63414, Seq: 112, Ack
v File Transfer Protocol (FTP)
  v 257 Dir C:\Users\Junaid\Desktop\FTP\FTP-py\Server\dadcreated\r\n
      Response code: PATHNAME created (257)
      Response arg: Dir C:\Users\Junaid\Desktop\FTP\FTP-py\Server\dadcreated
  [Current working directory: ]
```

Fig. B.15: Server MKD response for valid argument.

```
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCo
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.2
> Transmission Control Protocol, Src Port: 21, Dst Port: 50923, Seq: 20
v File Transfer Protocol (FTP)
  > 550 MKD failed \r\n
  [Current working directory: ]
```

Fig. B.16: Server MKD response for invalid argument.

PORT

```
> Frame 752: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 101,
v File Transfer Protocol (FTP)
  v PORT 192,168,101,231,178,97\r\n
      Request command: PORT
      Request arg: 192,168,101,231,178,97
      Active IP address: 192.168.101.231
      Active port: 45665
  [Current working directory: ]
```

Fig. B.17: Sending PORT command (with auto-formatted arguments).

```
> Frame 753: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63414, Seq: 302,
v File Transfer Protocol (FTP)
  v 200 Creating data socket\r\n
      Response code: Command okay (200)
      Response arg: Creating data socket
  [Current working directory: ]
```

Fig. B.18: Server PORT response for valid arguments.

```
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63702, Dst Port: 21, Seq: 133,
v File Transfer Protocol (FTP)
  > PORT 192,133,4,5,6,7,4\r\n
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 331,
v File Transfer Protocol (FTP)
  > 501 Syntax error in parameters or arguments\r\n
```

Fig. B.19: Server PORT response for invalid arguments.

PASV

```
> Frame 73: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on ir
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 50628, Dst Port: 21, Seq: 25, Ac
v File Transfer Protocol (FTP)
  v PASV\r\n
      Request command: PASV
  [Current working directory: ]
```

Fig. B.20: Sending of PASV command.

```
> Frame 74: 103 bytes on wire (824 bits), 103 bytes captured (824 bits) or
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_6
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50628, Seq: 65, A
v File Transfer Protocol (FTP)
  v 227  Entering Passive Mode 192,168,101,130,5,17\r\n
      Response code: Entering Passive Mode (227)
      Response arg:  Entering Passive Mode 192,168,101,130,5,17
      Passive IP address: 192.168.101.130
      Passive port: 1297
```

Fig. B.21: Server response for PASV.

RETR

```
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 50628, Dst Port: 21, Seq: 109, A
v File Transfer Protocol (FTP)
  > RETR download.png\r\n
  [Current working directory: ]
```

Fig. B.22: Sending RETR command.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50628, Seq: 277,
v File Transfer Protocol (FTP)
  > 250 Accepted\r\n
  [Current working directory: ]
```

Fig. B.23: Server RETR response for valid file.

```
> Frame 563: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50628, Seq: 717,
v File Transfer Protocol (FTP)
  v 550 File does not exist.\r\n
      Response code: Requested action not taken: File unavailable (550)
      Response arg: File does not exist.
  [Current working directory: ]
```

Fig. B.24: Server RETR response for nonexistent file.

```
48 a4 72 0a 50 5a f4 6d  04 e9 17 3f 08 00 45 00   H·r·PZ·m ···?··E·
04 28 45 23 40 00 80 06  64 f2 c0 a8 65 82 c0 a8   ·(E#@··· d···e···
65 e7 d1 9c b2 59 84 b5  66 db 58 2a 0c 02 50 18   e····Y·· f·X*··P·
01 00 fb 83 00 00 89 50  4e 47 0d 0a 1a 0a 00 00   ·······P NG······
00 0d 49 48 44 52 00 00  01 2b 00 00 00 a8 08 03   ··IHDR·· ·+······
00 00 00 8f 29 38 44 00  00 01 11 50 4c 54 45 73   ····)8D· ···PLTEs
af b9 20 0f 08 1d 01 00  74 b5 c1 3f 49 4b 1b 00   ·· ····· t··?IK··
00 ff ff ff 18 00 00 1e  00 00 17 00 00 1f 0b 00   ···· ···· ···· ····
00 00 00 75 b3 bd 73 b0  ba 20 00 00 21 0f 09 12   ···u··s· · ··!···
00 00 1e 05 00 75 af b5  6c a6 ad 1e 10 08 70 b1   ·····u·· l·····p·
b7 2a 25 21 21 0d 0b 1e  0f 05 74 ae bb 23 0e 0b   ·*%!!··· ··t··#··
23 00 00 70 a7 b2 73 b3  bb 1f 07 00 20 10 0c 58   #··p··s· ···· ···X
79 7d 40 52 53 70 a0 a4  4b 68 68 1d 12 09 50 6e   y}@RSp·· Khh···Pn
```

Fig. B.25: Partial RETR file contents sent on data connection.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50943, Seq: 239,
v File Transfer Protocol (FTP)
  > 550 Incompatible type encoding.\r\n
  [Current working directory: ]
```

Fig. B.26: Encoding conflict with requested file and selected TYPE.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50943, Seq: 239,
v File Transfer Protocol (FTP)
  > 550 Incompatible type encoding.\r\n
  [Current working directory: ]
```

Fig. B.27: Server response to RETR when incompatible file extension detected.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 292,
v File Transfer Protocol (FTP)
  > 425 Data connection was never created\r\n
  [Current working directory: ]
```

Fig. B.28: Server response when RETR requested but no data connection present.

STOR

```
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63779, Dst Port: 21, Seq: 103,
v File Transfer Protocol (FTP)
  > STOR plagarism_dec.pdf\r\n
  [Current working directory: ]
  [Command response frames: 48]
  [Command response bytes: 68622]
  [Command response first frame: 1993]
  [Command response last frame: 2051]
```

Fig. B.29: Sending of STOR command.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63779, Seq: 313,
v File Transfer Protocol (FTP)
  > 250 Accepted\r\n
  [Current working directory: ]
```

Fig. B.30: Server response to valid STOR command.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 202,
v File Transfer Protocol (FTP)
  > 550 Incompatible type encoding.\r\n
  [Current working directory: ]
```

Fig. B.31: Server to STOR with incompatible file and selected TYPE encoding.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63530, Seq: 211,
v File Transfer Protocol (FTP)
  > 501 No filename given\r\n
  [Current working directory: ]
```

Fig. B.32: Server to STOR with blank file name.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 253,
v File Transfer Protocol (FTP)
  > 425 Data connection was never created\r\n
  [Current working directory: ]
```

Fig. B.33: Server response when STOR requested but no data connection present.

STRU

```
> Frame 658: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 69, A
v File Transfer Protocol (FTP)
  v STRU F\r\n
      Request command: STRU
      Request arg: F
  [Current working directory: ]
```

Fig. B.34: Sending of STRU (F) command.

```
> Frame 659: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63414, Seq: 225,
v File Transfer Protocol (FTP)
  v 200 Structure Altered\r\n
      Response code: Command okay (200)
      Response arg: Structure Altered
  [Current working directory: ]
```

Fig. B.35: Server response to STRU (F) command.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63530, Seq: 65,
v File Transfer Protocol (FTP)
  > 504 Command not implemented for that parameter\r\n
  [Current working directory: ]
```

Fig. B.36: Server response to STRU valid but not implemented arguments.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63530, Seq: 113,
v File Transfer Protocol (FTP)
  > 501 Invalid structure given\r\n
  [Current working directory: ]
```

Fig. B.37: Server response to STRU (Invalid).

TYPE

```
> Frame 665: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 77, A
v File Transfer Protocol (FTP)
  v TYPE I\r\n
      Request command: TYPE
      Request arg: I
  [Current working directory: ]
```

Fig. B.38: Sending of TYPE (I) command.

```
> Frame 706: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 85, A
v File Transfer Protocol (FTP)
  v TYPE A\r\n
      Request command: TYPE
      Request arg: A
  [Current working directory: ]
```

Fig. B.39: Sending of TYPE (A) command.

```
> Frame 666: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_6
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63414, Seq: 248,
v File Transfer Protocol (FTP)
  v 200 Type Altered\r\n
      Response code: Command okay (200)
      Response arg: Type Altered
  [Current working directory: ]
```

Fig. B.40: Server response to TYPE (A) or TYPE (I).

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63530, Seq: 234, Ack:
v File Transfer Protocol (FTP)
  > 504 Command not implemented for that parameter\r\n
  [Current working directory: ]
```

Fig. B.41: Server response to TYPE with valid but not implemented arguments.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50923, Seq: 334,
v File Transfer Protocol (FTP)
  > 501 Invalid Type\r\n
  [Current working directory: ]
```

Fig. B.42: Server response to TYPE (Invalid).

MODE

```
> Frame 719: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 93, A
v File Transfer Protocol (FTP)
  v MODE S\r\n
      Request command: MODE
      Request arg: S
  [Current working directory: ]
```

Fig. B.43: Sending of MODE (S) command.

```
> Frame 720: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_6
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63414, Seq: 284,
v File Transfer Protocol (FTP)
  v 200 Mode Altered\r\n
      Response code: Command okay (200)
      Response arg: Mode Altered
  [Current working directory: ]
```

Fig. B.44: Server response to MODE (S) command.

```
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_6
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50923, Seq: 418,
v File Transfer Protocol (FTP)
  > 504 Command not implemented for that parameter\r\n
  [Current working directory: ]
```

Fig. B.45: Server response to MODE with valid but not implemented arguments.

```
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_6
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63615, Seq: 65, A
v File Transfer Protocol (FTP)
  > 501 Invalid mode given\r\n
  [Current working directory: ]
```

Fig. B.46: Server response to MODE (invalid).

NOOP

```
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63702, Dst Port: 21, Seq: 42, A
v File Transfer Protocol (FTP)
  > NOOP\r\n
  [Current working directory: ]
```

Fig. B.47: Sending of NOOP command.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 158,
v File Transfer Protocol (FTP)
  > 200 NOOP Done\r\n
  [Current working directory: ]
```

Fig. B.48: Server response to NOOP command.

LIST

```
> Frame 763: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 130,
v File Transfer Protocol (FTP)
  v LIST \r\n
      Request command: LIST
  [Current working directory: ]
  [Command response frames: 0]
  [Command response bytes: 0]
```

Fig. B.49: Sending LIST command with blank arguments.

```
> Frame 763: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 63414, Dst Port: 21, Seq: 130,
v File Transfer Protocol (FTP)
  v LIST \r\n
      Request command: LIST
  [Current working directory: ]
  [Command response frames: 0]
  [Command response bytes: 0]
```

Fig. B.50: Server response to LIST with blank arguments.

```
01 00 9e b4 00 00 2d 72  77 2d 72 2d 2d 72 2d 2d   ·····-r w-r--r--
20 31 20 64 65 66 20 64  65 66 20 33 33 36 34 20    1 def d ef 3364
09 20 41 70 72 20 09 20  31 36 20 09 20 30 35 3a   · Apr ·  16 · 05:
34 36 20 09 20 64 6f 77  6e 6c 6f 61 64 2e 70 6e   46 · dow nload.pn
67 20 0d 0a 2d 72 77 2d  72 2d 2d 72 2d 2d 20 31   g ··-rw- r--r-- 1
20 64 65 66 20 64 65 66  20 32 38 20 09 20 41 70    def def  28 · Ap
72 20 09 20 31 36 20 09  20 30 35 3a 34 36 20 09   r · 16 ·  05:46 ·
20 65 78 74 65 6e 73 69  6f 6e 73 2e 74 78 74 20    extensi ons.txt
0d 0a 2d 72 77 2d 72 2d  2d 72 2d 2d 20 31 20 64   ··-rw-r- -r-- 1 d
65 66 20 64 65 66 20 31  39 34 30 30 20 09 20 41   ef def 1 9400 · A
70 72 20 09 20 31 36 20  09 20 30 36 3a 31 30 20   pr · 16  · 06:10
09 20 46 54 50 2d 53 65  72 76 65 72 2e 70 79 20   · FTP-Se rver.py
```

Fig. B.51: Partial directory listing being sent on data connection.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63615, Seq: 252, A
v File Transfer Protocol (FTP)
    > 501 The directory does not exist \r\n
  [Current working directory: ]
```

Fig. B.52: Server response for nonexistent directory.

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 65, A
v File Transfer Protocol (FTP)
    > 425 No data connection created\r\n
  [Current working directory: ]
```

Fig. B.53: Server response for nonexistent data connection.

SYST

```
> Ethernet II, Src: IntelCor_0a:50:5a (48:a4:72:0a:50:5a), Dst: AsustekC_e
> Internet Protocol Version 4, Src: 192.168.101.231, Dst: 192.168.101.130
> Transmission Control Protocol, Src Port: 50943, Dst Port: 21, Seq: 25, A
v File Transfer Protocol (FTP)
    > SYST\r\n
  [Current working directory: ]
```

Fig. B.54: Sending of SYST command.

```
> Ethernet II, Src: AsustekC_e9:17:3f (f4:6d:04:e9:17:3f), Dst: IntelCor_6
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 50943, Seq: 65, A
v File Transfer Protocol (FTP)
    > 200 Windows\r\n
  [Current working directory: ]
```

Fig. B.55: Server response to SYST.

Unknown

```
> Internet Protocol Version 4, Src: 192.168.101.130, Dst: 192.168.101.231
> Transmission Control Protocol, Src Port: 21, Dst Port: 63702, Seq: 129,
v File Transfer Protocol (FTP)
    > 502 Command not implemented\r\n
  [Current working directory: ]
```

Fig. B.56: Server response to unknown command.

Remote Server

90.130.70.73	192.168.101.231	FTP	74	Response: 220 (vsFTPd 3.0.3)
192.168.101.231	90.130.70.73	FTP	70	Request: USER anonymous
90.130.70.73	192.168.101.231	FTP	88	Response: 331 Please specify the password.
192.168.101.231	90.130.70.73	FTP	65	Request: PASS pass
90.130.70.73	192.168.101.231	FTP	77	Response: 230 Login successful.
192.168.101.231	90.130.70.73	FTP	59	Request: PWD
90.130.70.73	192.168.101.231	FTP	88	Response: 257 "/" is the current directory
192.168.101.231	90.130.70.73	FTP	60	Request: PASV
90.130.70.73	192.168.101.231	FTP	103	Response: 227 Entering Passive Mode (90,13
192.168.101.231	90.130.70.73	FTP	62	Request: TYPE I
90.130.70.73	192.168.101.231	FTP	85	Response: 200 Switching to Binary mode.
192.168.101.231	90.130.70.73	FTP	62	Request: TYPE A

Fig. B.57: Use of client with remote server (speedtest.tele2.net).