

IMPLEMENTATION OF FILE TRANSFER PROTOCOL USING PYTHON SOCKET PROGRAMMING

Uyanda Mphunga
Student no: 1168101
Ashraf Omar
Student no: 710435

Abstract—An FTP program was implemented in python 3 using socket programming and TCP/IP connections.

I. INTRODUCTION

A File Transfer Protocol (FTP) was designed, implemented and tested by the authors using socket programming in python 3. The purpose of this project is to create a custom FTP protocol in order to fundamentally understand how protocols operate. This document assumes that the reader is familiar with computer networking and the terminology used. This document includes sections on the Design and Implementation, Code Functionality, Testing and Results, SWAT Analysis, Division of Tasks and Future Improvements.

II. BACKGROUND

FTP is an RFC959 [1] standardised file sharing (or file transferring) protocol that uses separate control and data connections between a client and server [2]. Protocols define the rules and format of messages and message transmissions [3]. The transmitted messages in FTP include control messages, which include the commands from the client to the server and response codes from the server to the client, and data messages which is used for file transmission between client and server. [4]. The protocol is a fifth layer protocol of the IP stack, the application layer [5], as seen in Fig 1, and is fundamentally a protocol with the intended application of sharing files between hosts (computer programs, etc).

FTP is built on TCP/IP, and is therefore a reliable connection protocol. This project uses socket programming in order to implement FTP. Sockets can be viewed as the *door* connecting the application and transport layers, as seen in Fig 2. The socket programs which are used to make the protocol use TCP and IP which is beyond the scope of this project and will not be discussed in detail since they belong to the transport and internet layers respectively [3]. FTP uses a set of commands and protocols in order order to achieve desired outcome, for example, a client has to send a password along with a specific command to the server in order for the input password to be valid. Upon receiving a said command, the server replies with a code number, accompanied by text, that indicates the failure or success of the implementation of said command.

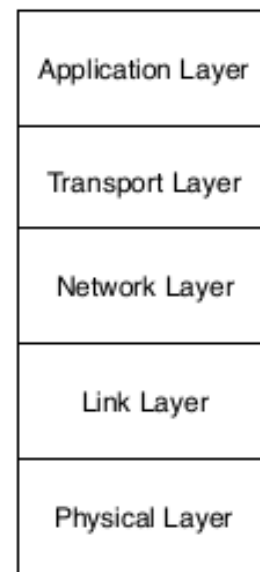


Fig. 1: IP Stack Layers

For this project, multi-threading was used. Multi-threading allows for the same code to run in different instances, effectively allowing the same code to run independently in different processes of the computer. For this project, multi-threading is used on the server side in order to allow different clients that connect to the server to run independently on different processes of the server. The detailing of threading falls out of the scope of this project, there is however literature that discuss threading such as reference [6].

As mentioned above, FTP is built on TCP/IP. TCP is a transport layer protocol whose fundamental task is the transmission of data between hosts [3], [7]. TCP transfers data in segments known as “packets”. The advantages of using TCP for data transfer are the following [8]:

- Data is transmitted in an ordered fashion according to the sequence number and rearranged at the destination.
- Any packets that are lost (i.e. not acknowledged) are retransmitted
- Data transfer is relatively error free due to the use of checksums
- Reliable delivery is maintained through flow control:
 - The receiver is made aware of the sliding window size of the sender.

- A window size of zero indicates that the receiving host's buffer is full and the transfer stops and the data in the buffer is processed.
- Modern TCP implementations can change the flow of data to avoid network congestion

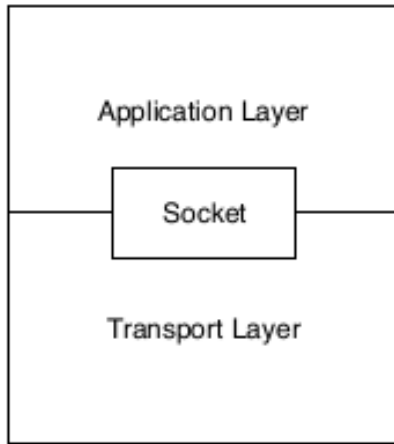


Fig. 2: IP Stack Layers

Wireshark, a packet analyser, was also used to capture and monitor the TCP packets when operating the FTP program. This allowed the authors to verify and validate the program by witnessing control messages, reply codes and file data being transferred through the TCP connections.

In order to store passwords and user names, a database needs to be used on the server side. The author opted to use MySQL database because of its security, reliability and scalability [9]. The database has a single table which is used to store the user name, user password, and the root directory for the said user wherein the user can perform file transfers on the server side. Due to MySQL using SQL, queries can be easily performed. An example of a useful query would be to find out if the user name, which is used in logging in when using FTP, exists (see Fig 3). If the user name exists, then the server proceeds to request for the user password, etc.

```
select count(*) from user
where name = some_password
```

Fig. 3: Querying whether user name exists

A. Project Requirements

The project requirements of the FTP program is the following:

- Implementation of FTP client code basic features
- Implementation of FTP server code basic features
- Using multi-threading
- Ability to deal with different types of files
- Use different computers for file transferring
- Using Wireshark to view TCP packets

III. DESIGN AND IMPLEMENTATION

The design and implementation of the authors' FTP program is in three parts. The parts are the client code, server code and MySQL database table. This section discusses the three mentioned parts and their implementation.

A. Database

Databases are useful for storing information. The authors opted to use MySQL database because of the simplicity of use and the author's experience with it. A table called **user** is created in the database (that is, MySQL) using the syntax shown in Fig 4. The table stores the user name, password of the user and the root name. The root name serves as a root directory for a user. Each root name therefore necessarily has to be unique for each user. By having unique root directory names, it is ensured that multiple users can access and use the FTP server, via the FTP client program, without affecting the data of other users. This results in ensuring that the FTP program can service multiple users as well. Due to the fact that the aforementioned table is created in using SQL (MySQL is a SQL database), all queries performed use SQL.

```
create table user(
user_name char(50),
password char(50),
root char(50),
primary key(50)
);
```

Fig. 4: SQL code for creating a table for storing credentials

A visual example of this table is shown in Table I. The SQL queries that are performed in the authors' FTP program are performed on this table. Storing the user credentials in a database instead of another storage format was chosen because of the following reasons. Firstly, by storing the data in a database ensures that the data is safely stored regardless of whether the server code is active or not. The use of the database also allows one to easily query data. Based on the authors' experience it was concluded that querying text files, csv files, etc. results in very complex code. Due to the database already using a language (that is, SQL), the amount of code necessary to perform a query is significantly reduced. Integrating the database with python is also simple. One of the major factors for using a database is that the data can be accessed across the entire server program, whether in functions, modules and/or classes. The benefit of such accessibility is if/when working with multiple python script files, the data can be accessed without having to make provision for passing data between the scripts that may need it. This in turn, results in the server code being easily scalable in terms of having multiple users.

```
select count (*)
where user_name = someUserName
```

Fig. 5: SQL code for checking if user exists

When the user enters the credentials stored in the database, queries have to be performed on the database, in order to ensure that the correct user is logging in, and to dictate which root directory (which is based on the root name column of Table I) a user can use when operating the FTP program. The basic queries that need to be performed on

the table (or database) are ones for user name verification, password verification and root name access.

```
select count (*)
where user_name = someUserName
and password = somePassword
```

Fig. 6: SQL code for checking if the password matches the user name

When a user enters their *unique* user name, a query that determines whether the user name exists in the database is executed. The SQL code example for this query is given in Fig 5. Due to the fact that each user name will be unique, when the query runs it returns either a one or a zero. When the result is a one, it means that the same user name exists in this query. When the result is zero, it means that the user name does not exist, that is, the login account does not exist.

```
select root from user
where user_name = someUserName
```

Fig. 7: SQL code for assigning root directory name for said user

Fig 7 is SQL code that returns a root folder name for a specified user, the returned user name is used to determine the root directory of said user, as stated above. Fig 6 shows the generic SQL code that returns the number of user names where the password entered on the client side matches the one stored in the databases' user table. Due to the fact that the user names of the table always unique, if the number returned by the query is the number one, then it means the password for said user name is correct. If the returned value is zero, then it means that the password is incorrect. It is important to note that the code given in Fig 6 is only ever run if the entered user name queried in Fig 5 exists. Due to this, the entered user name is always correct, and only the password authenticity is tested.

TABLE I: Table design of SQL Database for user credentials

user-name	password	root
user1	pass1	root1
user2	pass2	root2
user3	pass3	root3
user4	pass4	root4

B. Server

The server uses the database and queries it for user credentials and for determining root folders of each individual user, as stated above. The use of the database is embedded in the server's functions, modules, etc. This section focuses on an abstracted level of how the server interacts with the client code as demonstrated in Algorithm 1.

This section abstractly explains how the server is designed, and uses Algorithm 1's pseudo-code to explain it. Based on Algorithm 1's pseudo-code, it is seen that the server

first connects to the database in order to enable it to make queries to the database. Once database connections are established, the server then waits for client programs to make connections to it. Once a client programs connects to the server, a thread for servicing said client program is established. From this point on, how the server operates is essentially by taking in input commands from the client and processing them, and then performing the necessary task based on the command received and then sending replies back to the client. This design pattern is followed by the server whether it be when logging in, entering passwords or downloading a file.

Algorithm 1 Server Algorithm

```
1: procedure PROCESSING CLIENT COMMANDS
   connectDatabase()
   startSocketConnection()
2: while True: do
   createThread()
   command ← receiveCommand()
   commandType ← extractCommandType (command)
   commandValue ← extractCommandValue (command)
3: if loggedIn() == false then getCorrectLoginDetails (commandType, commandValue)
4: else processCommand(commandType, commandValue)
```

C. Client

Algorithm 2 shows the high level pseudo-code of how clients interact with the server. The client program first makes a connection to the server through the `PORT` command. Then after that the user enters commands to the client. The client then sends the commands to the server and the server responds to the client based on the input it receives. The client then proceeds to process these commands.

Algorithm 2 Client Algorithm

```
1: procedure PROCESSING SERVER COMMANDS IP ← PORT input connectToServer(IP)
2: while command != 'QUIT': do
   command ← getUserInputCommand()
   sendCommandToServer(command)
   response ← getServerResponse()
   responseCode ← extractResponseCode(response)
   processResponse (response, responseCode)
```

D. Commands and Reply Codes

Table II is a summary of the commands implemented by the authors' FTP program. It should be noted that the port command is specifically implemented in lower case alphabets.

When transmitting files between the client and server, the said files are read and written as binary files by the client and server programs. In python, any file type can be transmitted as binary, whether it be a text file, video, or an

image. Due to this fact, the authors did not implement the TYPE and MODE commands due to them being redundant in terms of file transfer. Multiple reply code have been implemented, for each command mentioned on Table III, there is a reply code implemented for it. In the event that a code entered by the user does not exist, the server has also implemented a reply code for it as well. The way the reply codes are used is that whenever a command is received, a reply code is sent back to the client for processing, depending on the said reply code, the server may then proceed to process data. An example of this would be when downloading a file from the server using the RETR, if the said file exists in the user's account, the server first sends the reply code confirming that it exists to the client and then proceeds to transfer the said file to the client.

TABLE II: Implemented FTP commands

Command type	<command value >	full command
USER	<User name>	USER <User name>
PASS	<password>	PASS <password>
NOOP	-	NOOP
RETR	<file name>	RETR <file name>
STOR	<file name>	STOR <file name>
STRU	-	STRU
port	<ip and port>	<ip and port>
QUIT	-	QUIT

The functionality of the commands, reply codes and formatting according to FTP standards can be found on reference [1].

TABLE III: Implemented RFC 959 Reply Codes and accompanying messages

Reply Code	Message Response
125	File status okay; about to open data connection.
150	Data connection already open; transfer starting
200	OK
225	Querying file structure
230	User logged in
331	User name accepted, please enter your password
450	File not found
502	Command not implemented
503	Please use correct command
530	Enter user name

IV. CODE FUNCTIONALITY

The client and server programs run on python 3 and use an installed MySQL database. This section gives a brief overview of how the program is used. The FTP is made to function only in active mode, this was done because according to the authors' views, allowing a user to connect to a port of their choosing can be problematic (in terms of security) if said ports are already in use by other users. Instead, by allowing the server to automatically dictate the port number to be used, connection error probability is mitigated. The user connects to the server by typing in port h1,h2,h3,h4,p1,p2, where h1-h4 is the IP address of the server (displayed in the server window) and p1 and p2 are the control and data ports respectively. After a client connects to the server, the server then requires the user name, then the password. Once these two have been accepted by the server, the user is then free to enter any other commands at the user's discretion. When the user types 'QUIT' or simply disconnect from the server,

the server acknowledges said disconnection and continues to run indefinitely while listening for other connections.

V. TESTING AND RESULTS

Wireshark was used to monitor the packets sent and received between two computers, one running the client and the other running the server. The TCP SYNACK messages were clearly visible and clear-text of command codes, reply codes (see Figure 8), user names and passwords (see Figure 9), and even binary file data were viewed without difficulty as shown in Figure 10. Refer to Appendix A for full, uncropped images of the Wireshark capture.

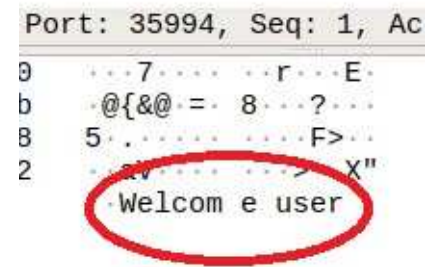


Fig. 8: Wireshark screen capture showing the clear-text of a received message

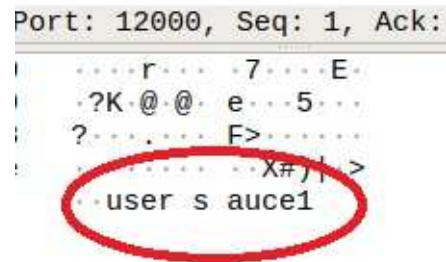


Fig. 9: Wireshark screen capture showing the clear-text of a username sent to the server

We can draw a conclusion from this that FTP is not secure and can easily be breached. This is because FTP pre-dates SSL and TLS. As demonstrated, FTP is vulnerable to "Packet Capture". Some of the other vulnerabilities of FTP are outlined in Reference [10] include:

- Using an open port to steal information
- FTP bounce attack, which is essentially a man in the middle attack which involves using the PORT command to access open ports while transfer is occurring
- Brute force attacks to try and guess IP and port numbers, or username and passwords
- Spoofing attacks in which IP data is falsified

```

ort: 13000, Seq: 1, Ack:
T.....@..X%
9.
ALICE' S ADVENT
URES IN WONDERLA
ND...

Lewis Ca rroll...

THE MILL ENNIUM F
ULCRUM E DITION 3
.0.....

CHAPTER
I.....
Down
the Rabb it-Hole
..... A lice was
beginni ng to ge
t very t ired of

```

Fig. 10: Wireshark screen capture showing the clear-text of a file being transferred through the data connection

VI. SWAT ANALYSIS

This section provides a SWAT analysis to the FTP program based on what the authors implemented.

A. Strengths

Most of the basic FTP commands were successfully implemented. The program is also easily scalable due to its design.

B. Weaknesses

Some of the basic features of the program for FTP were not implemented. The program does not make use of a GUI and therefore, cannot interact with FileZilla. It also does not have a `HELP` command, so it is not very user friendly. Another weakness is that the transferred data of FTP is neither secure nor encrypted and therefore vulnerable to being 'hijacked' by outside sources.

C. Advantages

The code is simple to use and the program provides detailed error codes to inform the user what their error is.

D. Trade-offs

Some of the basic commands of FTP were not implemented because they were seen as redundant, as such, the authors chose to implement less commands and limited functionality in order to have a running code that has less chances of an error.

VII. DIVISION OF TASKS

Due to the project requiring that a division of tasks be given, the authors opted code one python script each. Uyanda Mphunga implemented the server script and Ashraf Omar implemented the client side.

VIII. FUTURE IMPROVEMENTS

For future improvements, a GUI needs to be implemented, and perhaps also allow for the neglected commands to be implemented. It would be better if a `HELP` command were implemented to instruct the user on how to use the program.

IX. CONCLUSION

A file transfer protocol program was successfully written using python 3 socket programming. The FTP program used TCP connections to send control messages, receive reply codes and send and receive files through a data connection. The FTP program uses separate data and control connections as outlined in the RFC959 standard. The FTP program met all of the requirements of the project except one, which was to be able to interact with FileZilla. Wireshark was used to monitor the TCP packets being sent/received by the program. It was concluded that FTP is not designed to be secure since messages were able to be seen in plain text in the TCP packets.


REFERENCES

- [1] J. Postel and J. Reynolds, *Rfc 959: File transfer protocol (ftp)*, InterNet Network Working Group Std., 1985.
- [2] B. A. Forouzan, *TCP/IP protocol suite*. McGraw-Hill, Inc., 2002.
- [3] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, M. Goldstein, K. Alaura, and A. Agarwal, Eds. Pearson, 2017.
- [4] C. Bin, "Formalized description and analysis of ftp on petri net," Jiangxi Vocational College of Mechanical & Electrical Technology, Tech. Rep., 2015.
- [5] U. N, A. Khera, L. Suri, C. Gupta, and S. T, "Bandwidth analysis of file transfer protocol," , Tech. Rep., 2018.
- [6] K. A. Lambert and M. Osbourne, *Fundamentals of Python: First Programs*, M. Lee and B. Shailer, Eds. CENGAGE Learning, 2012.
- [7] D. E. Comer, *Computer Networks and Internets*, M. Goldstein, Ed. Pearson, 2015.
- [8] D. Comer, *Internetworking with TCP/IP: Principles, protocols, and architecture*, ser. Internetworking with TCP/IP. Pearson Prentice Hall, 2006. [Online]. Available: <https://books.google.co.za/books?id=jonyuTASbWAC>
- [9] T. Branson. (2016, Nov.) 8 Major Advantages of Using MySQL. Website. Quinstreet Inc. Last Accessed: 2019/07/05. [Online]. Available: <https://www.datamation.com/storage/8-major-advantages-of-using-mysql.html>
- [10] M. Allman and S. Ostermann, *FTP Security Considerations*, Online, Ohio University Std. 2557, May 1999. [Online]. Available: <https://tools.ietf.org/html/rfc2577>

APPENDIX

demo.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help



tcp

No.	Time	Source	Destination	Protocol	Length	Info
59	15:23:36.193113943	10.201.63.14	10.203.53.16	TCP	66	54219 → 14000 [ACK] Seq=1 Ack=2172929933 Win=66560 Len=0
60	15:23:36.194075995	10.201.63.14	10.203.53.16	TCP	75	54219 → 14000 [PSH, ACK] Seq=1 Ack=2172929933 Win=66560 Len=0
61	15:23:36.194132666	10.203.53.16	10.201.63.14	TCP	66	14000 → 54219 [ACK] Seq=2172929933 Ack=10 Win=29056 Len=0
62	15:23:36.194221756	10.201.63.14	10.203.53.16	TCP	66	54219 → 14000 [FIN, ACK] Seq=10 Ack=2172929933 Win=66560 Len=0
63	15:23:36.194929011	10.203.53.16	10.201.63.14	TCP	66	14000 → 54219 [FIN, ACK] Seq=2172929933 Ack=11 Win=29056 Len=0
64	15:23:36.197409364	10.201.63.14	10.203.53.16	TCP	66	54219 → 14000 [ACK] Seq=11 Ack=2172929934 Win=66560 Len=0
66	15:24:08.804201569	10.203.53.16	10.201.63.14	TCP	70	35994 → 12000 [PSH, ACK] Seq=27 Ack=118 Win=29312 Len=0
67	15:24:08.812271810	10.201.63.14	10.203.53.16	TCP	93	12000 → 35994 [PSH, ACK] Seq=118 Ack=31 Win=66560 Len=0
68	15:24:08.812325745	10.203.53.16	10.201.63.14	TCP	66	35994 → 12000 [ACK] Seq=31 Ack=145 Win=29312 Len=0 TSv=0
69	15:24:08.812475898	10.201.63.14	10.203.53.16	TCP	74	54222 → 14000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=0
70	15:24:08.812509392	10.203.53.16	10.201.63.14	TCP	54	14000 → 54222 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
71	15:24:09.407726288	10.201.63.14	10.203.53.16	TCP	74	[TCP Retransmission] 54222 → 14000 [SYN] Seq=0 Win=64240 Len=0
72	15:24:09.407799060	10.203.53.16	10.201.63.14	TCP	74	[TCP Port numbers reused] 14000 → 54222 [SYN, ACK] Seq=1 Ack=1036645531 Win=66560 Len=0
73	15:24:09.410461595	10.201.63.14	10.203.53.16	TCP	66	54222 → 14000 [ACK] Seq=1 Ack=1036645531 Win=66560 Len=0
74	15:24:09.411023100	10.201.63.14	10.203.53.16	TCP	72	54222 → 14000 [PSH, ACK] Seq=1 Ack=1036645531 Win=66560 Len=0

▶ Ethernet II, Src: IntelCor_37:02:b7 (00:db:df:37:02:b7), Dst: Cisco_f6:72:ff (00:a6:ca:f6:72:ff)

▼ Internet Protocol Version 4, Src: 10.203.53.16, Dst: 10.201.63.14

0100 = Version: 4

.... 0101 = Header Length: 20 bytes (5)

▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)

Total Length: 63

Identification: 0x4bed (19437)

▶ Flags: 0x4000, Don't fragment

Time to live: 64

Protocol: TCP (6)

Header checksum: 0x651a [validation disabled]

[Header checksum status: Unverified]

Source: 10.203.53.16

Destination: 10.201.63.14

▶ Transmission Control Protocol, Src Port: 35994, Dst Port: 12000, Seq: 1, Ack: 13, Len: 11

Data (11 bytes)

Fig. 11: Wireshark screen capture showing the IP addresses of the two hosts and TCP packets transferred

Wireshark - Packet 23 - demo.pcapng							
▶ Frame 23: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface 0 ▶ Ethernet II, Src: IntelCor_37:02:b7 (00:db:df:37:02:b7), Dst: Cisco_f6:72:ff (00:a6:ca:f6:72:ff) ▼ Internet Protocol Version 4, Src: 10.203.53.16, Dst: 10.201.63.14 0100 = Version: 4 0101 = Header Length: 20 bytes (5) ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT) Total Length: 63 Identification: 0x4bed (19437) ▶ Flags: 0x4000, Don't fragment Time to live: 64 Protocol: TCP (6) Header checksum: 0x651a [validation disabled] [Header checksum status: Unverified] Source: 10.203.53.16 Destination: 10.201.63.14 ▶ Transmission Control Protocol, Src Port: 35994, Dst Port: 12000, Seq: 1, Ack: 13, Len: 11 0000 00 a6 ca f6 72 ff 00 db df 37 02 b7 08 00 45 00 ...r...7...E- 0010 00 3f 4b ed 40 00 40 06 65 1a 0a cb 35 10 0a c9 .?K.@@e...5.. 0020 3f 0e 8c 9a 2e e0 8a ac 46 3e 08 cb 9d ca 80 18 ?...F>..... 0030 00 e5 ad 7f 00 00 01 01 08 0a 58 23 29 7c 00 3e X#} -> 0040 98 87 75 73 65 72 20 73 61 75 63 65 31 ..user s aucel							

Fig. 12: Wireshark screen capture showing the clear-text of a username sent to the server

