ELEN4017A - The Implementation and Analysis a File Transfer Protocol Application

Ntladi Mohajane. (1599953)

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

Abstract: This report discusses the implementation of a File Transfer Protocol (FTP) application that is built using Python. The aplication uses the TCP/IP protocol to establish connections between clients and servers and implements all the basic functionality outlined in Section 5.1 of the RFC 959 standard.

Key words:

1. INTRODUCTION

The world is becoming increasingly more connected in the age of the internet. Therefore, the need for standardized protocols and communication models that allow for devices to effectively communicate with one another is also important. These protocols and standards are defined within the 7 layers of the OSI Model, and the File Transfer Protocol (FTP) falls within the application layer of this model. FTP is the standard by which computers share data through the internet.

This report presents a file transfer protocol application that is implemented using the standards outlined in RFC 959. A detailed description of the implementation is given in the report. In addition, an analysis of the implementation is also given with includes results obtained from the WireShark packet sniffer.

2. BACKGROUND

The File Transfer Protocol was developed in 1971 by Abhay Bhushan. It was developed so that information could be transferred between servers and clients in a fast and reliable manner. Although the standard defining the protocol has been revised over the years, it is still the de facto standard for handling client/server file transfers over the internet to this day.

FTP facilitates communication between clients and servers through the Transmission Control Protocol/Internet Protocol (TCP/IP). The TCP/IP protocol is connection-orientated and allows for clients and servers to exchange information through a reliable byte stream channel. Packets sent through this protocol always arrive in the order in which they are sent, and the protocol has the advantage of guaranteed packet delivery due to error checking that occurs between transmissions.

RFC 959 is a document that outlines the standard of how FTP is implemented and used. It states that FTP consists of two, parallel TCP/IP connections between the client and the server called the control and data connections. The control connection is based on the Telnet protocol and is used by clients and servers to send commands and responses through what RFC 959 defines as a protocol interpreter (PI).

The data connection handles the actual transfer of files between the server and the client. It can operate in either active or passive passive mode and sends data in the form of either binary or ASCII plain text. Accessing files and transferring data between the client and the server is handled by what RFC 959 defines as a data transfer process (DTP). Figure 1 illustrates the FTP model as described in the RFC 959 document.

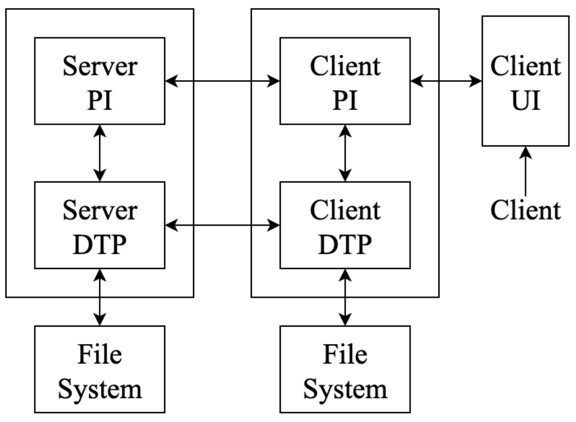


Figure 1: FTP Client-Server Model

The FTP process begins with the server opening a TCP/IP socket on port 21 for the client to connect to. Port 21 is the default port for establishing a control connection between the client and the server. However, other ports can be used as long as the client and server agree to it in advance. Figure 2 shows the process of how a connection is established between the client and the server, as well as how information is exchanged between them.

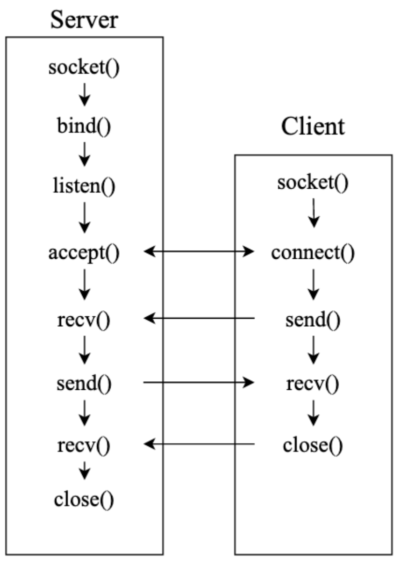


Figure 2: FTP Client-Server Communication

Once the connection has been established, the client is able to send commands to the server, some of which are sent with additional parameters, and the server is obligated to respond with reply codes and a message. The format of these is as follows:

- Client: `<Command Parameter Telnet Char>`
- Server: `<Code Message Telnet Char>`

Files can then be transferred between the client and server after the client has been successfully authenticated. The files are transferred on port 20 by default if the data connection is in active mode. However, this is rarely used and modern FTP implementations prefer to transfer data in passive mode using dynamic ports. In addition, clients can also change file transfer parameters such as the transfer mode and transfer type.

Using FTP as a transfer protocol is beneficial because it is widely used and implementing it according to the standard laid out in RFC 959 almost guarantees interoperability with different servers and clients. However, implementing plain FTP is insecure because commands and replies are sent over in plain text and are susceptible to attacks from anyone monitoring the network.

3. IMPLEMENTATION

3.1 Implemented Features

The FTP application described in this report consists of a client and server implemented using the Python 3 programming language. The code is split into 2 components (Server and Client) with 5 classes that are meant to model the ServerPI, ServerDTP, ClientPI, ClientDTP and ClientUI components illustrated in Figure 1. This is done to separate concerns and make the code more maintainable overall.

The server is multi-threaded and can process commands from multiple clients at once. In addition, the implemented server and client are able to connect to other clients and servers that conform to the basic FTP specification.

Furthermore, all the minimum commands that are outlined in section 5.1 of RFC 959 can be processed by the application. The application can also process additional commands related to creating, reading, updating and deleting files and directories as described in the specification. The implemented list of commands are as follows:

- *Minimum Commands*: USER, QUIT, PORT, TYPE, MODE, STRU, RETR, STOR, NOOP.
- *Additional Commands*: PASS, PASV, SYST, PWD, CWD, CDUP, MKD, RMD, DELE, LIST.

A description of all the implemented commands as well as their respective reply codes are outlined in Table 1 Appendix A.

RFC 959 also specifies the default parameters of some of the minimum commands. The application assumes that these parameters are set to their default values. The default parameters of the commands are as follows:

- TYPE A – ASCII Non-Print
- MODE S – Stream
- STRU F – File

The above commands can also take other arguments that change the properties of a file transfer. For example STRU can accept P as a parameter in order to display files in a page structure. The implemented application is not designed to account for any parameters that are not default. This simplifies the implementation overall thus resulting in less non-essential components being added to the code unnecessarily. However, an exception is made for the TYPE command. The application accepts the I parameter for the TYPE command, which allows the application to transfer different types of files encoded in binary instead of just ASCII plain text.

3.2 Data Connections

In order to use the the RETR, STOR and LIST commands, the application necessitates that a data connection be opened before hand between the client and the server. The application allows for this connection be opened either actively or passively by using the PORT and PASV commands respectively.

An active data connection occurs when the client opens a socket for the server to connect to in order to establish a data connection. The client sends their IP address as well as information about the available port to the server as a parameter when they send the PORT command. By default, the client should open port 21 however, the implemented application generates a relatively high random port between 5120 and 7935 so as not to trigger the firewall. The format of the PORT parameter is as follows:

- a1,a2,a3,a4,b1,b2

where a1-a4 are the highest to lowest order 8-bit components of the host IP address. In addition, b1 and b2 represent the high and low order 8-bit components of the open data port.

A passive data connection occurs when the server is the one to open the data socket for the client to connect to, in order to facilitate data transfer. The client sends a PASV command with no argument and the server ideally responds with its host information formatted in the same way as the PORT parameter.

RFC 959 requires that an FTP application support active data connections, however passive data connections have been implemented because they reduce the likelihood of the client's firewall preventing a data transfer in the name of security.

3.3 Users and Directories

RFC 959 states that a client can log into an FTP server in anonymous mode. i.e they can access specific contents of the server without an account. The application does not implement anonymous mode and requires all clients to log in with a unique username and a password by using the USER and PASS commands respectively.

In addition, the application does not implement any way of creating a new user. Instead, all users are predefined once the application executes. Furthermore, every user has a unique repository that only they have access to. Moreover, once the user is logged in, they can use the PWD, CWD, CDUP, MKD, RMD, and DELE commands to navigate the files and directories within their repositories

By not implementing anonymous mode or the creation of new accounts, the way in which the application interacts with the native file system is simplified. Moreover, requiring all users to login using passwords makes the application more secure. Furthermore, implementing extra commands for navigation makes the application more user friendly.

3.4 FTP Server Structure

When a client's PI sends a command to the server's PI, the server automatically splits the command from any parameters that might accompany it. The server then uses Python's *getattr()* method to turn the command into a function call. This establishes clear naming conventions for all the functions in the server's PI because they are named after the commands they are meant to process. Moreover, there is a clear distinction between methods that handle the PI and DTP because they are separated into different classes. Furthermore, the DTP is the only component that has direct access to system files.

The server also has the ability to interface with the standard FileZilla client hence why the SYST command has been implemented. This command is hard coded to inform FileZilla that the application is running on using off of the UNIX operating system and that it must format its commands (Particularly path names) accordingly. Furthermore, the server also has the ability to connect to multiple clients at once using multi-threading.

3.5 FTP Client Structure

The client is split into three sections and the user only directly interacts with the user interface defined in ClientUI.py. The user interface is a simple command line interface that abstracts away the process of directly formatting and encoding the commands that are sent to the server. In addition, the function calls that send commands to the server have descriptive names, thus making them more readable. Furthermore, much like the server, the client's PI and DTP also have clear separation.

When the user starts the client, it gives them the option of entering a host-name and host port to connect to. However, the client is configured to enter the implemented server's information by default if the user does not enter anything. Once connection to a server has been established, The user is prompted to enter their username and password. If the user is connected to the implemented server and enters blank details, they will be logged in using the credentials of the user 'Ntladi'.

Upon successful login, the directory list is then automatically retrieved and displayed through a passive data connection in order to show the user the contents of their repository. The user also has the ability to call up an instruction manual for the user interface at anytime by entering *help* into the console.

4. RESULTS

The application was tested on a machine running on the MacOS operating system using Python 3.8.3 and all functions of the application performed successfully. In the context of this project, success is defined by the application's ability to send and receive files between the client and the server without issue. Furthermore, the application was also successful in that the server is able to interface with external clients such as FileZilla without issue. Moreover, the client can also interface with external servers. Appendix B contains screenshots of the application being run through the Wireshark packet sniffer.

5. ANALYSIS AND CRITIQUE

Although the application performed as desired, there are still improvements that can be made. Firstly, the applications uses the plain-FTP standard and the WireShark results show the commands and messages are sent unencrypted and can be intercepted by malicious agents. Therefore, it is advised that future implementations make use of the secure-FTP standard for increased security. Moreover, the client occasionally misprints the list retrieved from the implemented server and there is currently reason to explain why. Furthermore, the external client does not suffer from this issue, thus indicating that the problem lies with the implemented client that must be fixed in any future implementations.

6. CONCLUSION

In conclusion, the implementation and analysis of a file transfer protocol application has been presented. It implements all the basic commands defined in the RFC 959 specification as well as others to increase the quality of the application overall. Despite issues concerning the application's security, it performed as expected and is deemed successful.

REFERENCES

[1] M. Rouse. (2014, Aug.) Osi reference model (open systems interconnection. [Online]. Available: http://searchnetworking.techtarget.com/definition/OSI

[2] D. R. Winkelman. (2013, Feb.) Chapter2:protocols. [Online]. Available: https://fcit.usf.edu/network/chap2/chap2.htm

[3] V. Beal. (2017, Dec.) The 7 layers of the osi model. [Online]. Available: https://www.webopedia.com/quickref/OSI Layers.asp

[4] J. T. Micah Cowan. (2017, Sep.) Ftp vs http. [Online]. Available: https://daniel.haxx.se/docs/ftp-vs-http.html

[5] J. R. J. Postel. (1985, Oct.) File transfer protocol(ftp). [Online]. Available: https://www.ietf.org/rfc/rfc959.txt

Appendix A

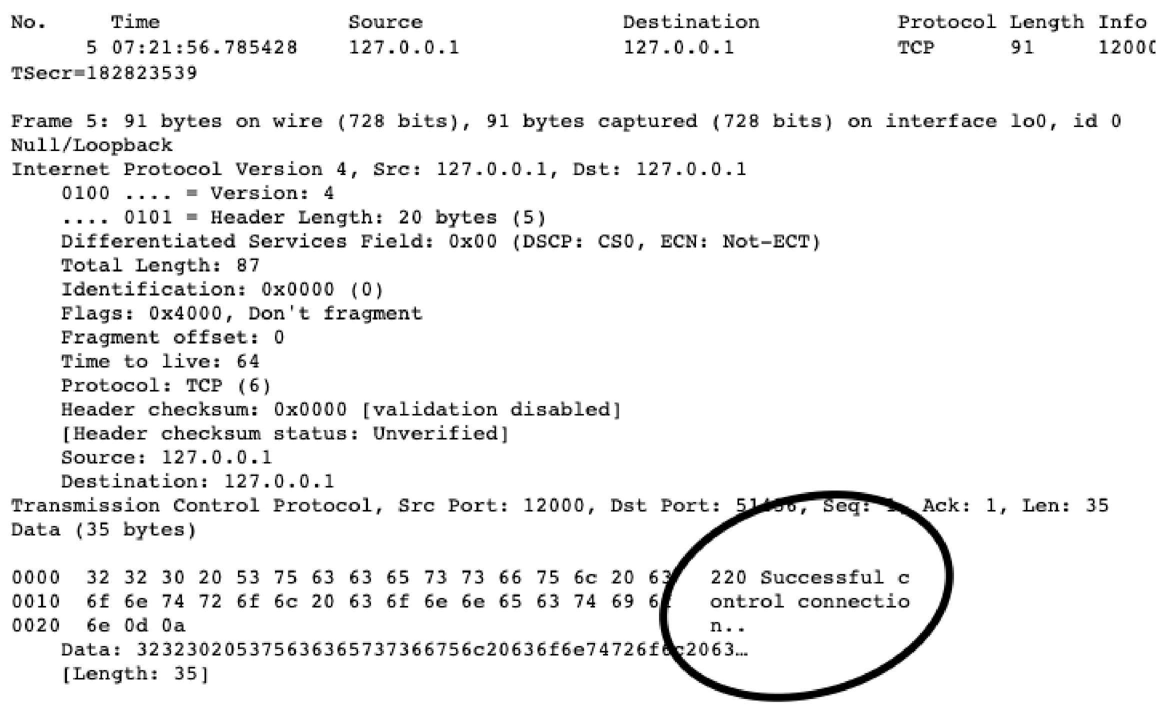
This Appendix contains a table of all the implemented commands as well as corresponding reply codes

Table 1: Table describing commands as well the corresponding reply codes

Command	Description	Success Reply Code	Error Reply Code
USER	Used to send the username to the server	331	332
PASS	Used to send the password to the server	230	530, 501
PASV	Used to establish a passive data connection	227	425
PORT	Used to establish an active data connection	225	425
SYST	Return the server's operating system	215	500
RETR	Download a file from the server	125, 226	426, 450
STOR	Upload a file to the server	125, 226	426
QUIT	Terminate the connection to the server	221	500
NOOP	Check if the server's control connection is still active	200	500
TYPE	Change the file open mode Default is ASCII	200	501
STRU	Change the directory structure, the default is File	200	501, 504
MODE	Change the transmission mode, the default is Stream	200	501, 504
PWD	Get the present working directory	200	500
CWD	Change the current working directory	250	501
CDUP	Change to the parent directory	200	500
MKD	Create a new directory	257	501
RMD	Delete a directory	250	501
DELE	Delete a file	250	501
LIST	List the contents of a directory	125, 226	426, 450

Appendix B

This Appendix contains screenshots several packets obtained using the WireShark Packet Sniffer. The Highlighted sections show the commands and replies being sent between the client and server.

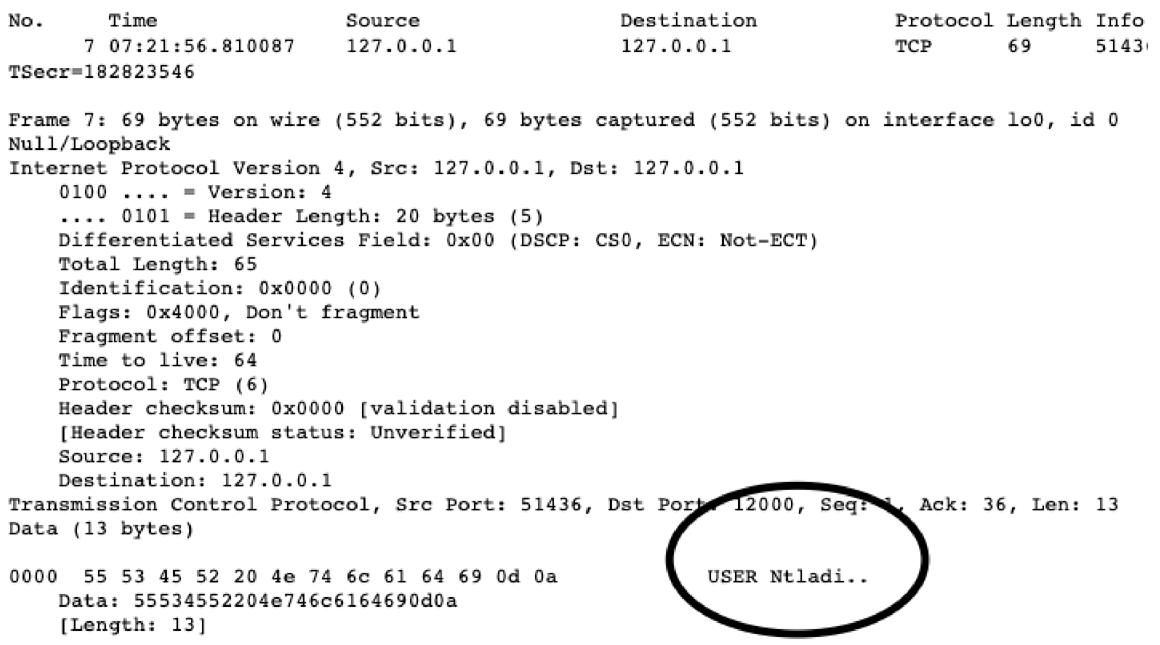


```
No.     Time                  Source                Destination           Protocol Length Info
      5 07:21:56.785428       127.0.0.1             127.0.0.1             TCP      91     12000
TSecr=182823539

Frame 5: 91 bytes on wire (728 bits), 91 bytes captured (728 bits) on interface lo0, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 87
    Identification: 0x0000 (0)
    Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source: 127.0.0.1
    Destination: 127.0.0.1
Transmission Control Protocol, Src Port: 12000, Dst Port: 51436, Seq: 1, Ack: 1, Len: 35
Data (35 bytes)

0000  32 32 30 20 53 75 63 63 65 73 73 66 75 6c 20 63   220 Successful c
0010  6f 6e 74 72 6f 6c 20 63 6f 6e 6e 65 63 74 69 6f   ontrol connectio
0020  6e 0d 0a                                          n..
    Data: 32323020537563636573736675 6c20636f6e74726f6c2063…
    [Length: 35]
```

Figure 1B: Welcome Message

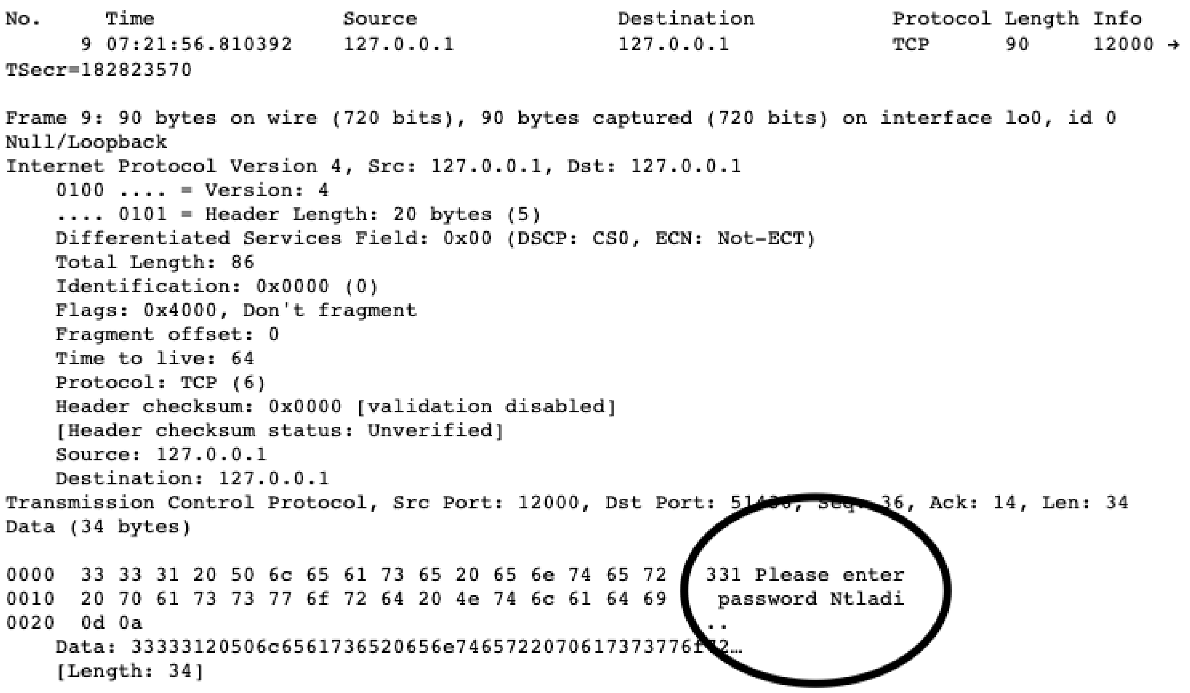


```
No.     Time                  Source                Destination           Protocol Length Info
      7 07:21:56.810087       127.0.0.1             127.0.0.1             TCP      69     5143
TSecr=182823546

Frame 7: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface lo0, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 65
    Identification: 0x0000 (0)
    Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source: 127.0.0.1
    Destination: 127.0.0.1
Transmission Control Protocol, Src Port: 51436, Dst Port: 12000, Seq: 1, Ack: 36, Len: 13
Data (13 bytes)

0000  55 53 45 52 20 4e 74 6c 61 64 69 0d 0a            USER Ntladi..
    Data: 55534552204e746c6164690d0a
    [Length: 13]
```

Figure 2B: Login Command

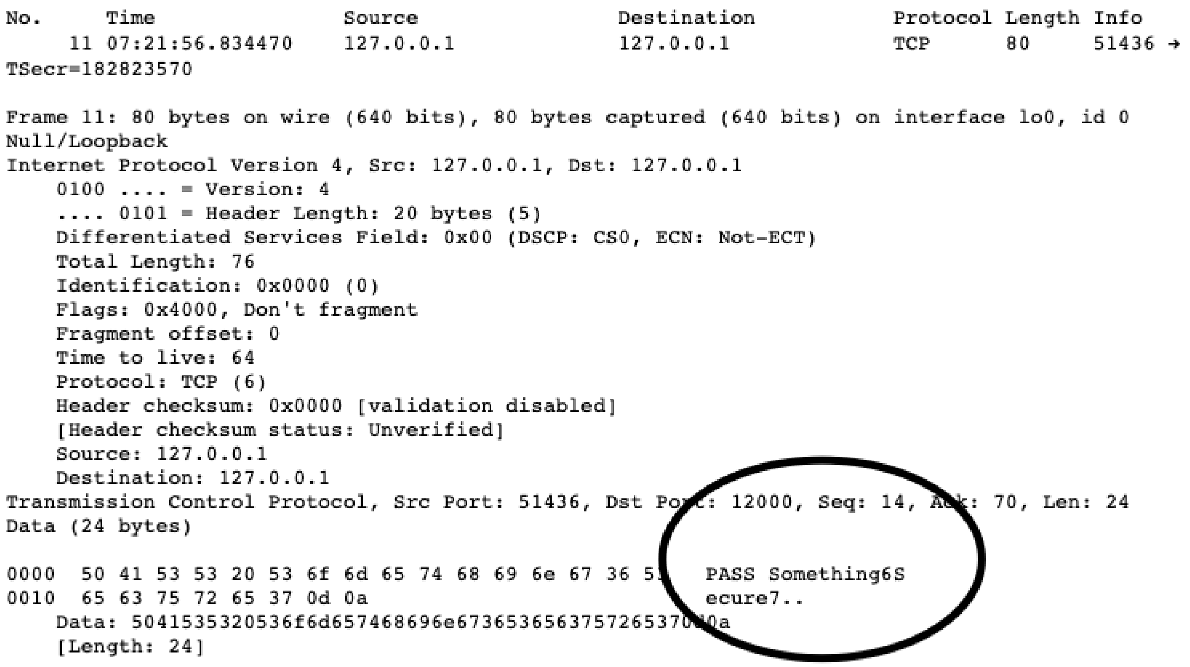


```
No.     Time               Source                Destination           Protocol Length Info
      9 07:21:56.810392    127.0.0.1             127.0.0.1             TCP      90     12000 →
TSecr=182823570

Frame 9: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface lo0, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 86
    Identification: 0x0000 (0)
    Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source: 127.0.0.1
    Destination: 127.0.0.1
Transmission Control Protocol, Src Port: 12000, Dst Port: 51436, Seq: 36, Ack: 14, Len: 34
Data (34 bytes)

0000  33 33 31 20 50 6c 65 61 73 65 20 65 6e 74 65 72   331 Please enter
0010  20 70 61 73 73 77 6f 72 64 20 4e 74 6c 61 64 69    password Ntladi
0020  0d 0a                                              ..
    Data: 33333120506c6561736520656e7465722070617373776f72…
    [Length: 34]
```

Figure 3C: Password Alert



```
No.     Time               Source                Destination           Protocol Length Info
     11 07:21:56.834470    127.0.0.1             127.0.0.1             TCP      80     51436 →
TSecr=182823570

Frame 11: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface lo0, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 76
    Identification: 0x0000 (0)
    Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source: 127.0.0.1
    Destination: 127.0.0.1
Transmission Control Protocol, Src Port: 51436, Dst Port: 12000, Seq: 14, Ack: 70, Len: 24
Data (24 bytes)

0000  50 41 53 53 20 53 6f 6d 65 74 68 69 6e 67 36 53   PASS Something6S
0010  65 63 75 72 65 37 0d 0a                           ecure7..
    Data: 50415353205366d657468696e673653656375726537 0d0a
    [Length: 24]
```

Figure 4D: Password Command