Advanced Network Programming

Reliable File Transfer Protocol

George Kozakjian

Table of Contents

[**1.** **Introduction** 3](#_Toc508139958)

[**2.** **Overview of FTP** 3](#_Toc508139959)

[**3.** **The FTP Protocol architecture & Model** 4](#_Toc508139960)

[**3.1.** **FTP protocol architecture** 4](#_Toc508139961)

[**3.2.** **FTP protocol model** 4](#_Toc508139962)

[**4.** **Implementation & code** 5](#_Toc508139963)

[**4.1.** **Use of Sockets** 5](#_Toc508139964)

[**4.2.** **Data representation** 5](#_Toc508139965)

[**4.3.** **Transfer logic, offsets, seeking and re-transmission** 6](#_Toc508139966)

[**4.4.** **Checksum validation** 6](#_Toc508139967)

[**4.5.** **Lightweight “Hello” handshake** 6](#_Toc508139968)

[**4.6.** **FTP client and server PDUs** 7](#_Toc508139969)

[**4.7.** **Dynamic timeout optimization (DTO)** 7](#_Toc508139970)

[**4.8.** **Verbose output for diagnostics** 7](#_Toc508139971)

[**4.9.** **Security** 8](#_Toc508139972)

[**5.** **Results** 9](#_Toc508139973)

[**6.** **Limitations** 11](#_Toc508139974)

[**6.1.** **Bidirectional file transfer** 11](#_Toc508139975)

[**6.2.** **Security** 11](#_Toc508139976)

[**7.** **Improving the protocol** 11](#_Toc508139977)

# **Introduction**

Communication networks were first created to connect people in remote in time and place. The main goal of the first networks was to share and be able to send/receive documents and code between academic people, which in turn will facilitate and reduce the time it take to send some document or receive one instantaneously, rather than waiting for hours or maybe days to send some file or info through the traditional transport mediums.

As these networks grown, more scalable and structured communication protocols were needed, with always having the idea of making easier for humans to share their information and resources with each other. The emergence of the OSI model played a pivotal role in revolutionizing networking, and the way protocols are created and maintained.

The structured and hierarchical design of the OSI model led to the inception of various research groups that took responsibility of developing protocols that “belong” naturally to some layer of the OSI model. IEEE and IETF are two examples of these groups. IEEE today develops L1 and L2 protocols mostly. And IETF usually develops protocols that belong to L3 to L7 of the OSI model.

Berkeley was one of the universities and research groups that had huge contribution to the development of the high level communication protocols. With the creation of the SOCKET interface and APIs, they made it easier to create application layer protocols like the HTTP, FTP, etc.

The file transfer protocol was originally released back in 1971 (RFC 114). It had the objectives of promoting file sharing, using remote computers to fetch or send data, give users the ability of seamlessly using remote storage and files as it was locally attached, and the reliable transfer of files.

RFTP mainly is an application layer protocol that provides users the ability of transferring file to/from a remote computer over a communication network reliability and efficiently.

# **Overview of FTP**

FTP is and end-to-end application protocol that runs over TCP or UDP, and facilitates the transfer of files between two nodes (client and server, or server and server) it is a standard built on the client-server architecture model, where the server listens to client connections on some specific, well-known port (port 21), and the client tries to establish a connection to the remote server, specifying the remote port as 21.

The original FTP uses two different connections for the data and control packets. The control connection is used to exchange command and replies between the client and the server, and the data connection is used to transfer the actual data in form of file, part of file, number of files, etc.

FTP has two modes for establishing a data connection between a client and a server. If the client is behind a firewall or NAT device, and unable to accept incoming TCP or UDP connection, *passive mode* is used, where client indicate to server over control channel that it will be running data channel in passive mode, and server will send IP/Port of its data channel, to enable the client establish a data connection to the server. The second mode is active mode, where client can accept incoming connection from server on some random port M.

Moreover, unlike HTTP, FTP is known to be a statefull protocol. It since it has a stateful control connection that keep track of the file being transferred, the directory to read from, the byte count, the mode, the file identity, the session identity among other flags.

# **The FTP Protocol architecture & Model**

## **FTP protocol architecture**

Since FTP is based on the client-server architecture model, there has to be two different software entities, running on different nodes that are connected with an underlying communication network. The two nodes can act in: (1) client-server model, or (2) server-server model. In the client-server model, the client node uses services provided by the server node; the client gets, puts, deletes files to/from the server, and the server tries to keep up and execute client requests. In the server-server model, both nodes execute the client and server part of the FTP software.

## **FTP protocol model**

In this implementation, the client-server model will be used. The implementation will try to follow the FTP model defined in *IETF RFC 959.* In this model, there are two processes running on two different machines or nodes: (1) *Client-FTP process*, and (2) *Server-FTP process*. These processes are composed of two different pieces of software that act together to provide the FTP client or server service: (1) protocol interpreter *PI,* and (2)data transfer process *DTP.*

Each component of a node (client or server) have different responsibilities *[Figure 1]:*

* Server:
  + Server-PI: listens on port 21 for connection from Client-PI, resulting in establishment of the control channel between the nodes. next, it receives the FTP commands from client and sends replies, and controls the DTP process (sending of data)
  + Server-DTP: transfers the data on command from the PI. Usually will be in passive state listening, waiting client to establish the data channel through control channel passive command (indicate to server it will be using passive technique to establish the data channel)
* Client:
  + Client-PI: initiate control channel choosing random local port to the server-FTP process, and starts sending FTP commands. Controls the DTP in receiving data.
  + Client-DTP: receives or transfer data on command of PI, and checks for data integrity.

In this implementation, the protocol interpreter (control channel) and DTP (data channel) will be using UDP. Also, the control and data channels will be consolidated into a single UDP channel, carried over *rftp\_pdu.*

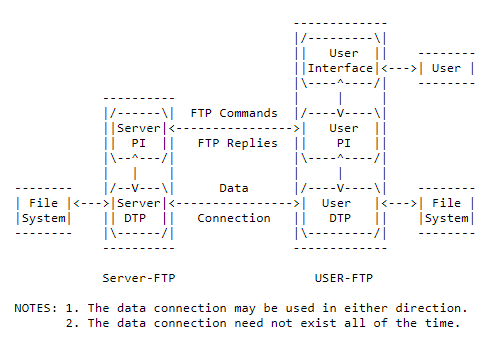


Figure Model for FTP Use [from RFC 959]

# **Implementation & code**

## **Use of Sockets**

This implementation tries to mimic the real FTP protocol by using python UDP sockets between a client and a server. Use of socket is helpful in simulating and testing real-life client and server that are trying to communicate end-to-end over a packet-switched shared network infrastructure. A socket is a high level API that is used to create transport layer end-to-end communication channel, which can be utilized to create application layer protocols and services. It is comprised of an IP address and a port number, which uniquely identifies it on the internet or intranet. This implementation uses sockets on the sender and receiver sides.

The implementation starts with the client creating its own UDP socket by using bind on a randomly generated port number outside the well-known UDP port range. Next, the client sends a “Hello” request to the server, allowing the server to obtain the client’s remote IP address and Port number, the server then tries to reply to the “hello” request, by using the port and ip of the received UDP datagram from the client.

## **Data representation**

In FTP, there are many data representations that can be used when transferring over the network:

* ASCII mode: mainly used for text. The idea here is to convert text into 8-bit ASCII representation before transmission, and can’t be used for files that are not text based.
* Image mode or binary mode: the client scan some given files, and sends it to the receiving node byte by byte or multiple bytes at a time. The receiver stores the byte stream, resulting in the transmission of exact file from source to destination. This mode is used to transfer all file types, and is the recommended mode to be used in FTP.
* Block mode: the FTP protocol breaks the data into several blocks, each containing a header and data fields. Next it passes this block to the TCP or UDP socket.

In this implementation, the *block mode* will be used for representing and transferring data from the client to the server. The blocks of data will be stored in *rftp\_pdu* , serialized and transmitted from the server to the client.

## **Transfer logic, offsets, seeking and re-transmission**

In this implementation, the client requests each block of data by sending the offset to the server, which in turn returns the requested blocks by seeking the fie some offset from the beginning of the requested file. Upon successful receipt and checksum validation, the client increments the offset to next block, and sends request to the server. If the checksum is falsely validated, the client just decrements the offset, allowing it to go back, and request the previous block.

This implementation provides the ability to put the most of the work on the client side, and remove processing overhead on the server side.

The python file library is used to seek, read and write to and from a file.

## **Checksum validation**

Since this implementation of FTP uses UDP and not TCP, error correction should be accounted for and implemented efficiently in the application protocol. Error control usually consists of two separate, but integrated functionalities:

* Error detection: concerned with detecting errors in transmission. To validate that the payload has been well received or not, some kind of checksum need to be used. In this implementation the MD5 algorithm is used to generate checksums on server and client side on the payload, and is validated and compared to detect transmission errors. So the server calculates an MD5 hash on the data, and adds It to the *server\_rftp\_pdu*. Upon datagram receipt, the client also calculates MD5 hash of data, and compares with transmssted checksum. If both match, then well received.
* Error correction: is the idea of recovering from transmission errors, and can be done in two ways which are ARQ or FCC. In this implementation, ARQ or repeat request will be used with timeout and checksums. If client calculate wrong checksum, re-transmission requested. If client times out, re-transmission requested.

The python *hashlib* library is used to generate and compare MD5 hashes.

## **Lightweight “Hello” handshake**

Since we are using UDP, no handshake is present in the underlying transport channel. Hence, this implementation adds a lightweight “Hello” handshake between client and a server, to make sure that server is online and able to transmit requested file.

The handshake starts by the client binding its UDP socket, and sending a hello message to the server. The server in turn accepts the hello message, and sends back a hello message to the clients UDP bound socket.

Upon completion of the handshake phase, the two parties start to transmit.

The handshake is very helpful, because it protects against faults and ensures both parties are ready to complete the file transfer.

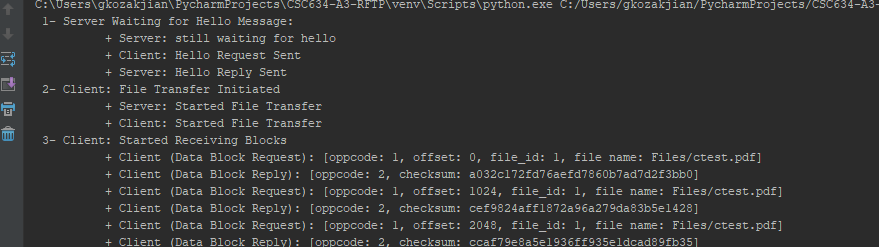


Figure : Hello messages exchanged

## **FTP client and server PDUs**

Since this implementation tries to imitate the real FTP protocol, it adds and uses the idea of PDUs (protocol data units) which include protocol headers and payload/data. The client and server have different PDUs, as they have different headers, so OOP inheritance has been used to create a base PDU that includes common attributes, and server/client PDUs inherit from it and add their own attributes.

The PDUs are transmitted from client to server, and vise versa. The headers and payload are loaded before each UDP transmission and serialized to be transmitted. This implementation uses *pickle* to serialize/deserialize the PDUs that need to be transmitted between the client and the server.

## **Dynamic timeout optimization (DTO)**

In this implementation, when the packet times out, we increase the timeout-value to next one. This delay will stay, even if the channel goes back to normal, and the latency of the channel decreases. Hence, this implementation adds an automatic channel optimization feature.

The idea is to keep track of the datagrams that are being received in time, and if the number of datagrams are more than some defined threshold value, then dynamic optimization will revert the latency/timeout back to a previous lesser value.

The dynamic optimization is off by default, and starts when first timeout is occurs. Upon optimizing the channel, the DTO goes back to being off, and starts again when a timeout occur.

## **Verbose output for diagnostics**

This implementation provides the ability to specify a verbose flag when starting the client and server, which triggers the verbose feature, allowing to see a detailed output of transmitted data and commands between client and server.

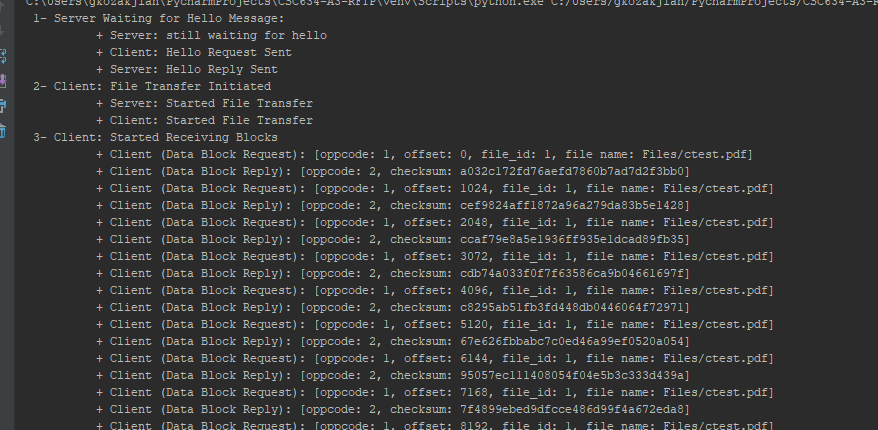


Figure : Verbose Output

## **Security**

FTP support authentication by using predefined username/password at the server side. However, since the protocol does not have built in encryption, the username/passwords are sent in clear text, and any intruder can capture and exploit them.

The protocol was not originally designed with security in mind, hence the data transfer between the client and server is not encrypted. This presents a weakness that should be addressed.

Many solutions and amendment to the protocol has been proposed and added: (1) use secure FTP over SSL instead of normal FTP, (2) transfer file over different secure protocol like SSH, (3) use a secure established tunnel to do FTP over like VPN or SSH tunnel.

In this implementation the third approach will be tested, which is using a secure established L3 IPSEC VPN tunnel to a destination site which includes the server. Below in the figure we can see a running RDP session to a remote computer over IPSEC VPN to a remote site over a WAN of speed 16 Mbps Up/Down. Since the link is low latency (100 – 200 milliseconds) and timeout-value is one second, we can see a reliable file transfer of size 20 Mbps over this link. In the figure, the computer on the left is in the remote site, acting as the server with IP Address 192.168.0.108, and computer on the right is acting as the client fetching a file from the server.

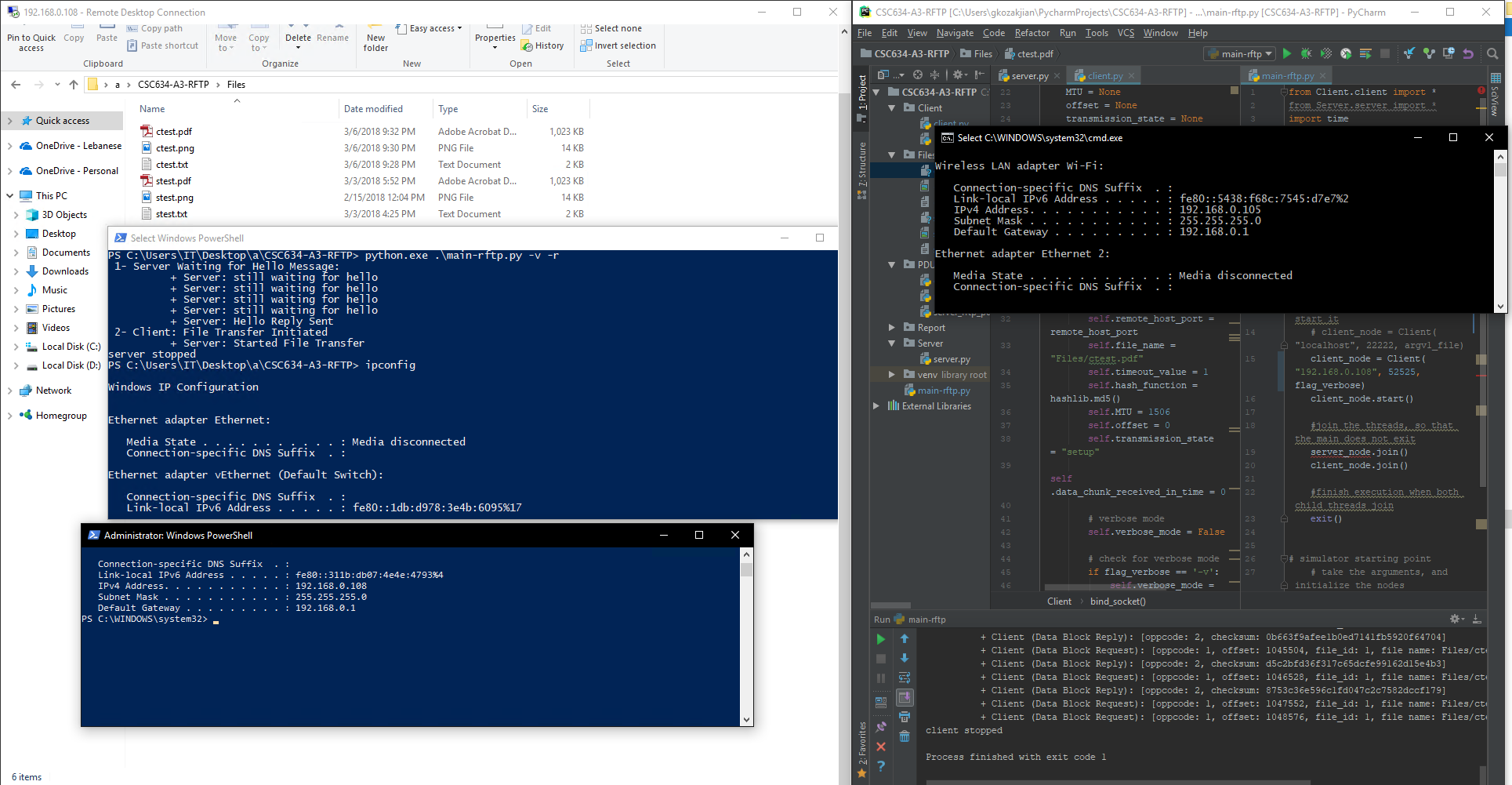


Figure : Test over VPN to remtoe server

# **Results**

The implementation transfers any type of file very efficiently and reliably. It has been tested so far on text files, pdf, jpeg , video , and audio files. This implementation is able to transfer type of file, because the server is reading and sending the requested file byte by byte, hence the receiver will get exact same file of 0s and 1s. So this will work on any type of file.

In the picture below, we can see transfer of an image file. Since we are using blocks of size 1024 bytes, we can see that file transfer was quick and efficient.

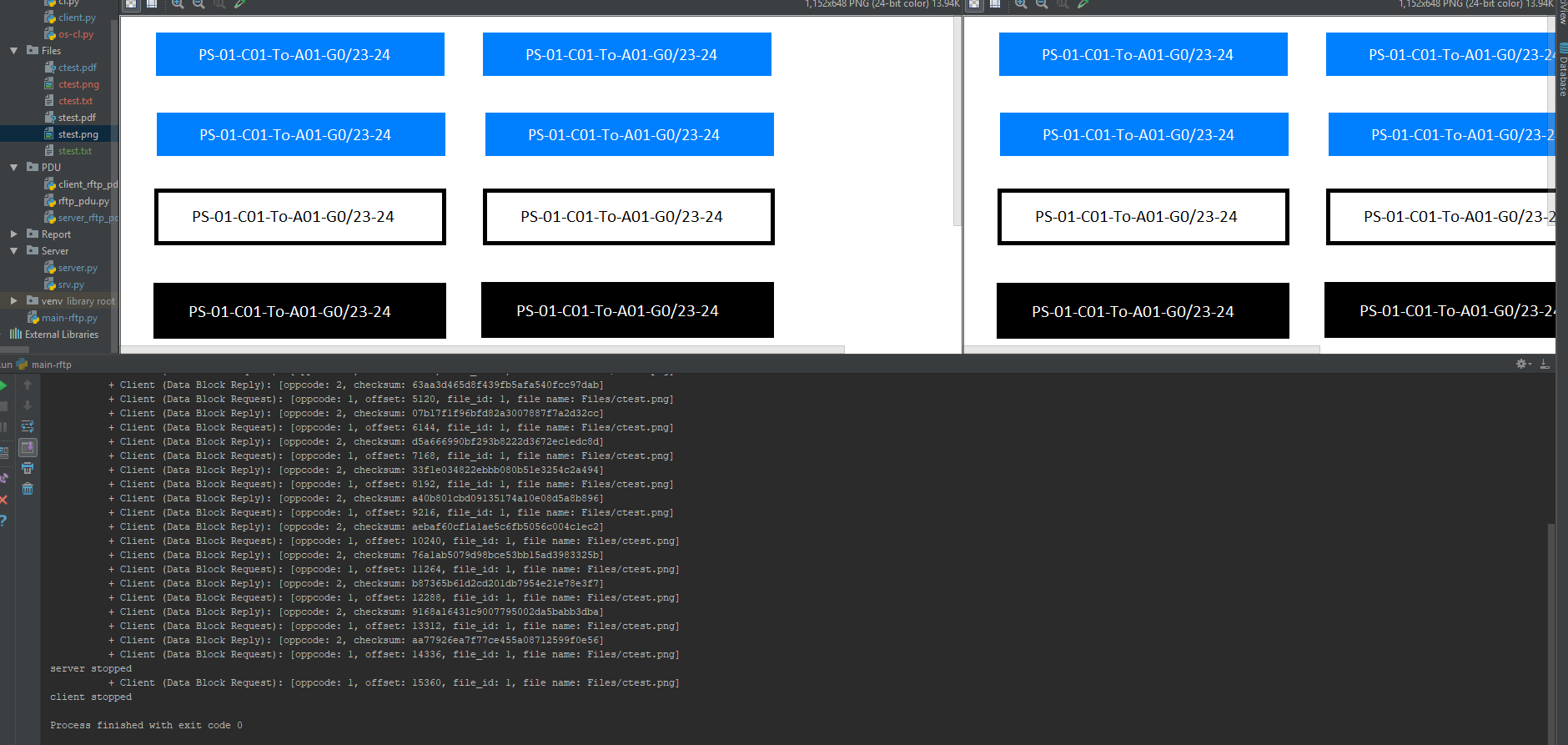


Figure : RFTP image file transfer

The implementation has also been tested to transfer PDF files. In the picture below, we can see transfer of a PDF file of size 1 MB. From the offsets we can see that this was a longer transmission.

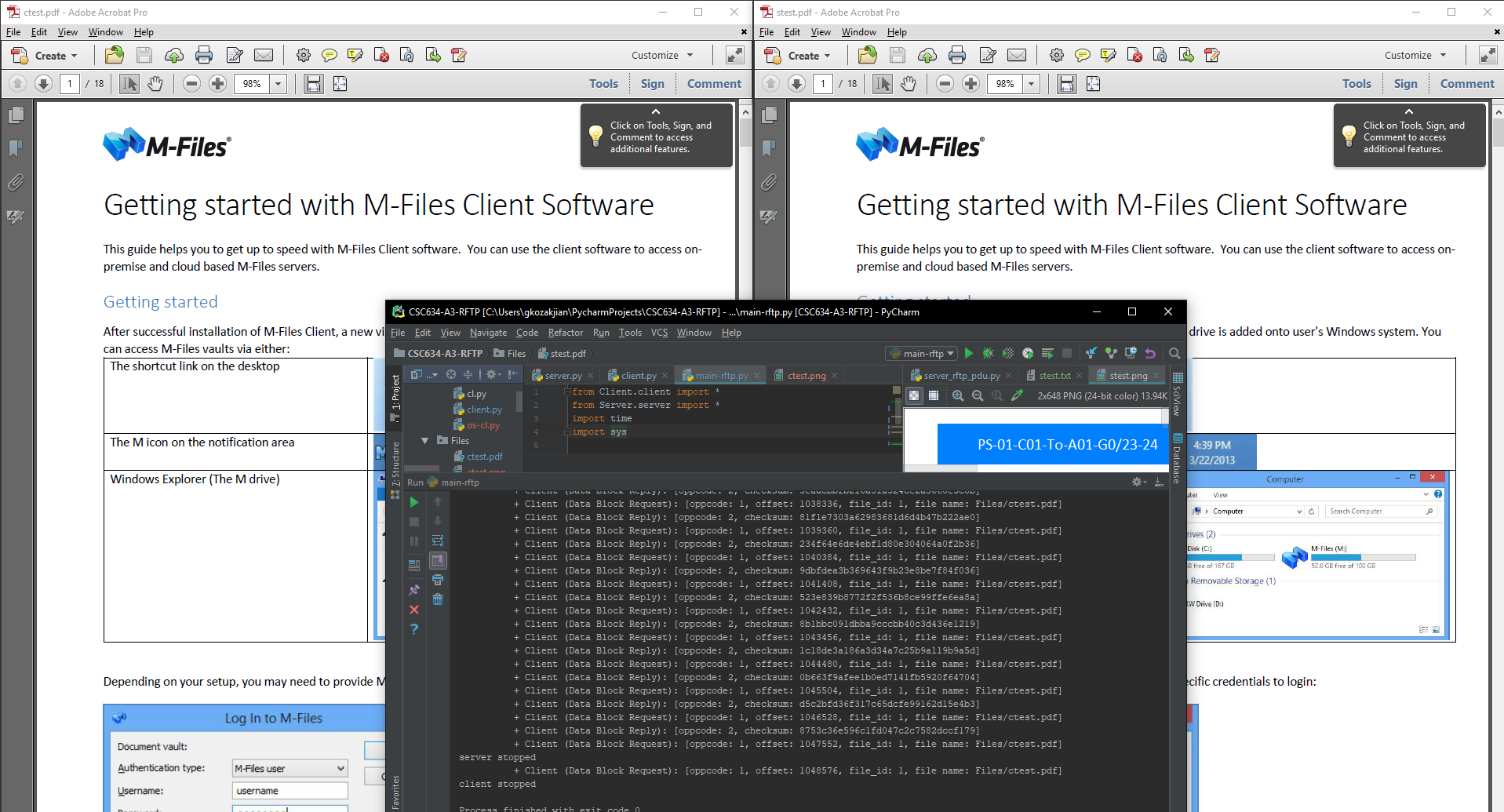


Figure : RFTP PDF file transfer

In this third test case, we have an error rate of 10 % of total transmissions. So every 9th transmission will be timeout, and client will repeat the request. We can see in the output that the client timeout is being optimized and increased to 4 rather than 10 at end of each cycle, since the dynamic timeout optimization algorithm is running and reducing timeout to 1 every 4 cycles of good transmissions which will occur, which is keeping the transmission running. We can see transmission was successful and verify file integrity .

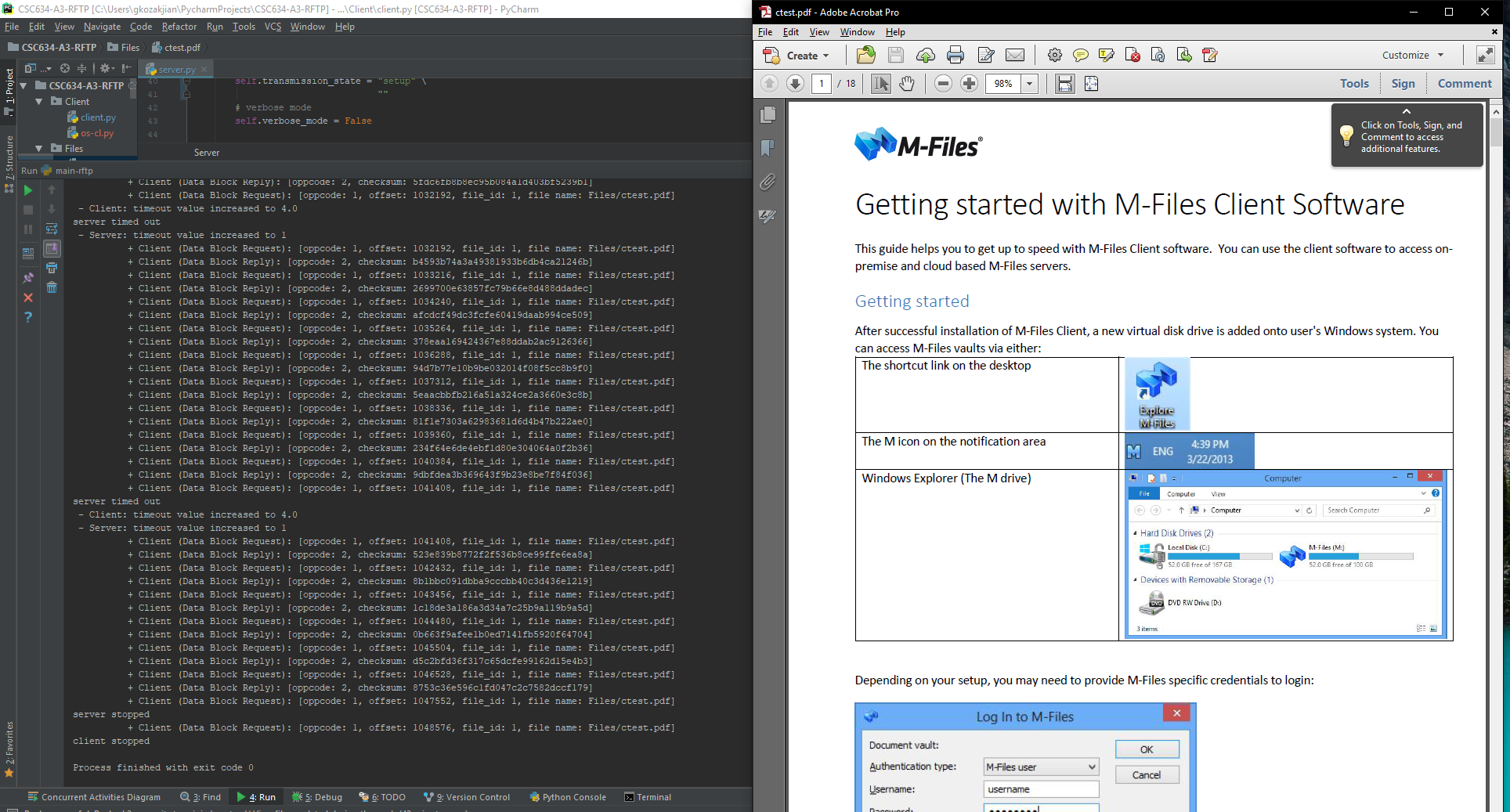


Figure : Error rate of 10%

In this final example, we can see that the error rate is more than 30%, so the client timeout value will be as high as 10, and it will stop, causing the transmission to stop as well.

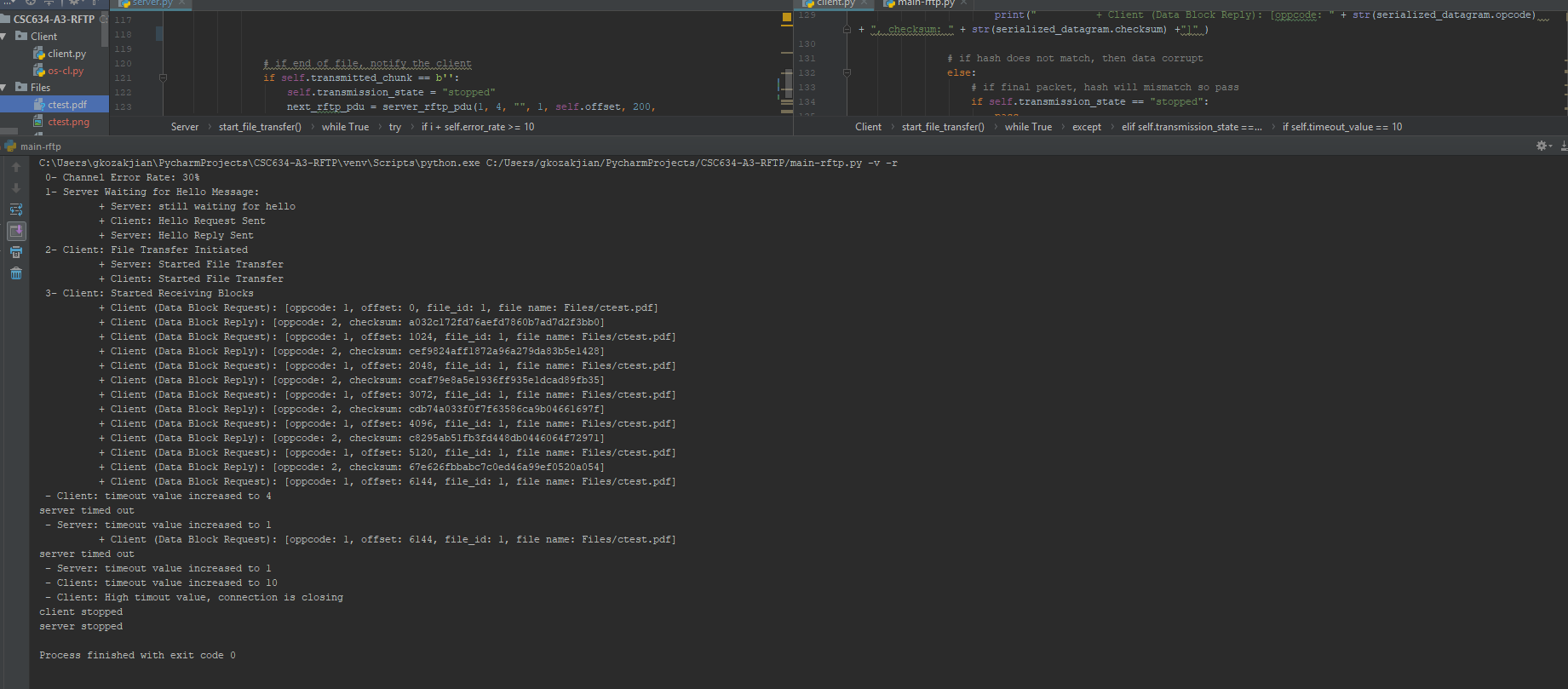


Figure : Error rate of 30%

# **Limitations**

## **Bidirectional file transfer**

This implementation of FTP is half-duplex, since only one transmitter and one receiver can exist at a time or transfer and receive a file at a time. To enable full-duplex file transfer, two separate sessions are needed with threading.

Since UDP Sockets are unidirectional in nature, to enable full-duplex file transfer, two sessions are needed to be established, one in each direction. Also, different sender and receiver threads are needed to enable full-duplex non-blocking file transfer

## **Security**

This implementation of FTP does not address security in any form. Hence, the transmitted data is in clear text, and can be sniffed and re-constructed by some intruder.

To address the security, encryption of data can be used with SSL sockets. to address authentication, password can be used or certificates by both parties (client and server)

# **Improving the protocol**

This implementation can be improved by extensive testing in real-life environments. Areas of improvement might also include parallel transmission of data across multiple sessions or connections using different network interface cards.