**Names: Nathaniel Hector, Saiuri Moodley & Zuleigha Patel**

**Student Numbers: HCTNAT003, MDLSAI008 & PTLZUL001**

**Course: CSC3002F**

**Date of Submission: 06 March 2023**

**Networks and Operating Systems**

**Semester One 2023**

**Assignment One: Socket Programming Project**

Table of Contents

[Introduction 2](#_Toc128940911)

[Server Features and Functionality 2](#_Toc128940912)

[Uploading Files 2](#_Toc128940913)

[Downloading Files 2](#_Toc128940914)

[File Validation 2](#_Toc128940915)

[Available Files Query 3](#_Toc128940916)

[Multiple Client Handling 3](#_Toc128940917)

[Client Session Ending 3](#_Toc128940918)

[Protocol Design & Specification 3](#_Toc128940919)

[Message Formats and Structure: 3](#_Toc128940920)

[Sequence Diagrams: 4](#_Toc128940921)

[Client Features and Functionality 5](#_Toc128940922)

[Client 1 (HCTNAT003 - Nathaniel) 5](#_Toc128940923)

[Client 2: (Saiuri) 6](#_Toc128940924)

[Client 3: (Zuleigha) 6](#_Toc128940925)

[Visual Models 7](#_Toc128940926)

[References: 9](#_Toc128940927)

# Introduction

The problem statement involved designing and implementing a basic Python client-server file sharing application using Transmission Control Protocol (TCP) sockets. The first specification surrounded the transfer of files between clients and the shared server. This feature allows a client to upload files to the shared server and download files from the shared server. To allow for the above, it is also necessary that the shared server receive files from clients and send files to clients. The second specification addressed the confidentiality of, and access to, certain files stored on the shared server. This feature allows for a client who uploaded a file to control the ability of other clients to download the file. To allow for the above, the client is required to provide a key during the upload process if the file is to be protected. It is also necessary that all other clients provide this same key during the download process. And the third specification was to include a file validation mechanism within the application. This feature allows for a sender to compute some data from the file to be sent to the receiver and include this data in the message to be sent to the receiver. The receiver then verifies the file received has not been altered during the transfer process by computing its own version of the data from the file received and cross-checking this against the sender's computation.

# Server Features and Functionality

## 

## Uploading Files

Using the POST command, a connected client may upload a file to the server (This file **must** be in the same directory as the client code). The file may be uploaded in one of 2 states: an open state or a protected state. If being uploaded in the protected state, the client restricts access to the file with a key (password). The protected file may only be downloaded by other clients if this key is entered along with the request. Open files may be downloaded by any client. Examples of the POST command may be seen in the “**Visual Models”** section.

## Downloading Files

Using the GET command, a connected client may download a file from the server. Every file stored on the server may be in one of 2 states: open or protected. Any client may download an open file but to download a protected file the client must add the correct key used to protect the file along with their request. Examples of the GET command may be seen in the “**Visual Models”** section.

## File Validation

(Python SHA256 Hashing Algorithm: Explained, 2021)

File validation is used when a client uploads to and downloads from the server. The sender sends a file along with a hash of the file to the receiver using a hexadecimal format of the SHA256 hashing algorithm. Once the receiver gets the entire file, it computes a hash of the file it received using the same algorithm. The receiver then checks whether the hash it computed corresponds with the hash it received from the sender. This allows the receiver to confirm whether the file was sent without being corrupted or not.

## Available Files Query

Using the QUERY command, the client can receive a list of available files on the server. All clients may view available open files. Clients may only view available protected files if they add a key when making the QUERY request. Examples of the QUERY command may be seen in the “**Visual Models”** section.

## Multiple Client Handling

The server listens for multiple clients but only accepts commands from one at a time. The rest of the clients wait in a queue. Once the message of the client being helped has been completed, the server disconnects that client. The disconnected client may attempt to reconnect in which case they will wait at the back of the queue until they are helped. This allows clients to wait for shorter periods before their command is executed. If a single client was helped until they disconnected, other clients may be forced to wait for an indefinite period.

## Client Session Ending

When the server is waiting for a client’s command, but the client wants to leave the server, the client may send an EXIT command to indicate that they are disconnecting from the server. This allows the server to attend to the next client in the queue.

# Protocol Design & Specification

This protocol was designed to be stateless. A single command is executed per connection to the server and the server retains no information about the client or what they did. The sending of messages were assumed to be reliable in most cases, verification only takes place when sending files or strings that are long as they may require many messages to be sent before they are completely received.

Users may specify the privacy of the files which they upload. If they upload a file in an open state anyone may access it. But if they upload the file in a protected state, only those with the special key used when uploading that file may access it. Users may tell others what they key is if they want them to have access to the file as well

## Message Formats and Structure:

**DATA TRANSFER MESSAGES**

**GET <privacy> <filename> <key, if applicable>**

**POST <privacy> <filename> <key, if applicable>**

* Privacy == “p” indicates the file being requested/uploaded is protected. In this case a correct key **must** be entered to access/upload the file
* Privacy == “o” indicates the file being requested/uploaded is open. In this case, no key should be entered
* Filename is the name of the file including the extension. File **must** be in the same directory as the client code when uploading.

**QUERY <privacy> <key, if applicable>**

* Privacy == “o” indicates a query for all available open files. No key should be entered
* Privacy == “p” indicates query for available files stored being protected by a specific key. The correct key **must** be entered.
* Privacy == “a” indicates query for available files stored being protected by a specific key as well as all available open files. The correct key **must** be entered.

**COMMAND MESSAGES**

**EXIT**

* No parameters

**CONTROL MESSAGES**

**OK|<command, applicable>**

* Sent to confirm that the previous message was received
* Sent with a command parameter to inform the client what command type was validated

**BAD|<error explained>**

* Sent when an error occurs along with an explanation of what the error is

## Sequence Diagrams:

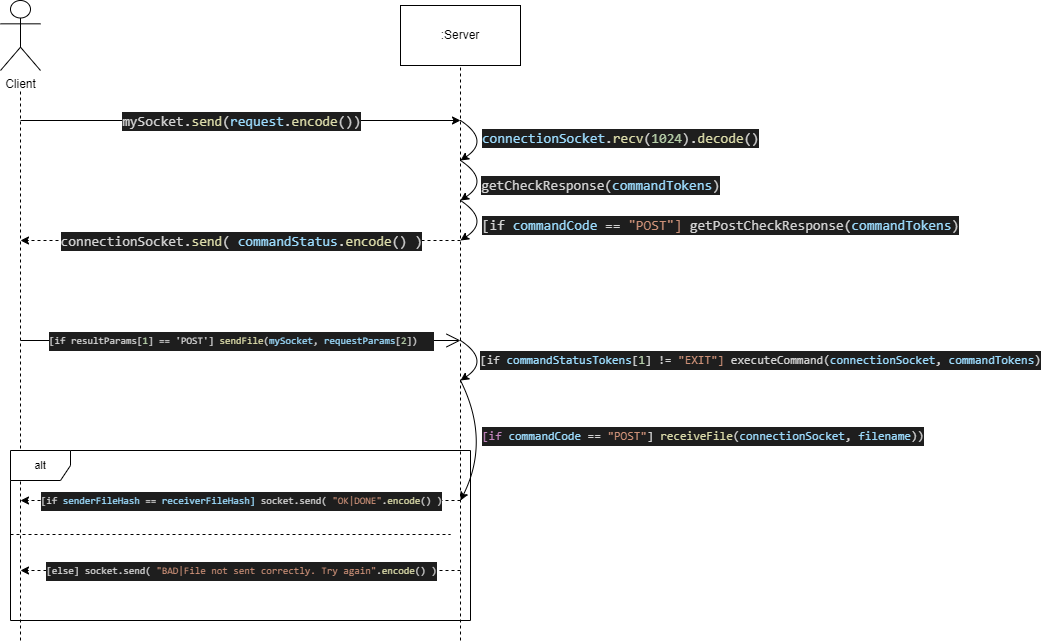


Figure 1- Sequence Diagram for Uploading Files

Diagram

Description automatically generated with medium confidence

Figure 2- Sequence Diagram for Downloading files

# Client Features and Functionality

### Client 1 (HCTNAT003 - Nathaniel)

**Error Handling**

* Prints error message if program is run without command line arguments.
* Prints error message if client loses connection to the server. Gives the client a choice to attempt to reconnect with the server or to exit the program.
* Prints error message if the client attempts to send a file that does not exist in their directory. Client may continue entering commands afterward.

**Query output parsing**

* Parses the QUERY string received from the server and prints it in a more understandable form.

**Available command instructions**

* Available commands and their parameters are kept at the top of the terminal screen. Terminal is cleared and available commands are reprinted after client’s command has been processed.

**POSTMANY**

* Special instruction allowing the client to upload several files by parsing the client's input into several POST commands. If used to upload protected files they will all be protected by the same key.

**GETMANY**

* Special instruction allowing the client to download several open files by parsing the client’s input into several GET commands.

### Client 2: (Saiuri)

* If command line arguments are not entered, the program prints an error message informing the user that an IP address and port number for the server are required.
* The program prints a greeting message with a list of requests that can be sent to the server, informing the user of the expected structure of these messages.
* After a user’s request is completed, the program reprints the list of requests that can be made for the user’s convenience.
* User output takes the form of slightly longer messages. The user is provided with more information about what the program is doing behind the scenes.
* User input (excluding requests) usually takes the form of an entire word or an obvious abbreviation rather than a character (e.g., “QUIT” instead of “q”).
* The program parses and formats the results of a QUERY request before printing. This is done using the helper methods: open, protected and all.
* The open(files) and protected(files) methods accept a string as a parameter, remove the leading privacy flag, parse the remainder of the string into individual filenames and print the filenames.
* The all(files) method also accepts a string as a parameter. However, it separates the string into two halves so that either the open(files) or protected(files) methods can be called for parsing and printing.

### Client 3: (Zuleigha)

* If the command line input is not entered correctly, the server sends an error message to the client informing the client that it didn’t use it correctly
* This client displays similar functionality to the other clients except that it has added a special method for the Welcome message and calls it under the main method:

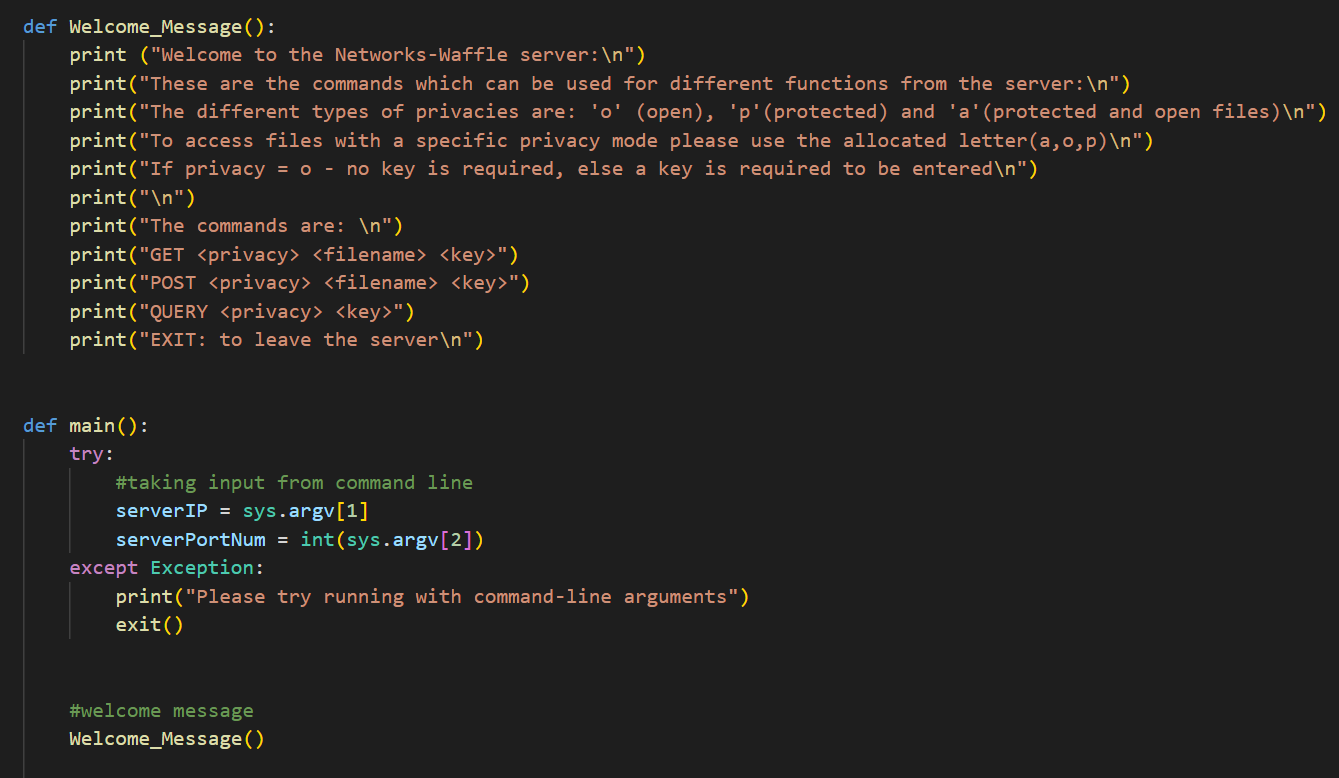


Figure 3 - Welcome message - Zuleigha

* As can be seen above, it is called straight after the try-catch method for the command – line-arguments.
* The client code has nested if statements which are used for command execution of the GET, POST and QUERY commands.
* Another feature being added is a special way of printing out the QUERY output as follows. It is formatted to print the privacy mode then a tab and the file name:

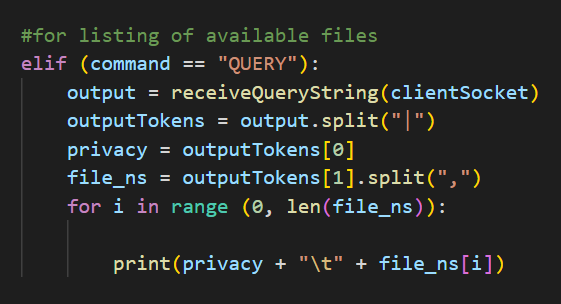


Figure 4 – Query Output – Zuleigha

# Visual Models

For this demonstration of the client-server interaction, a bare client which can only send commands and print the error messages and query output received from the server will be used. The console output of both the client and server will be shown. The server starts out as having no files stored.

For each type of command, the client will first send one invalid version of the command and proceed to send a valid version of all variations of the command. The server’s console output will be shown on the left terminal and the client’s input and output will be shown on the right terminal.

POST commands: In *figure 5 below*, the client successfully uploads 2 files to the server. One open file named “file.txt” and one protected file named “song.mp3” which has the key “123456”



Figure 5 - POST command

QUERY commands: In *figure below*, the client queries what files are available on the server. The available files are shown below the Message which the client has sent (for QUERY o, the applicable files are “o|file.txt|” and similar for the other messages)

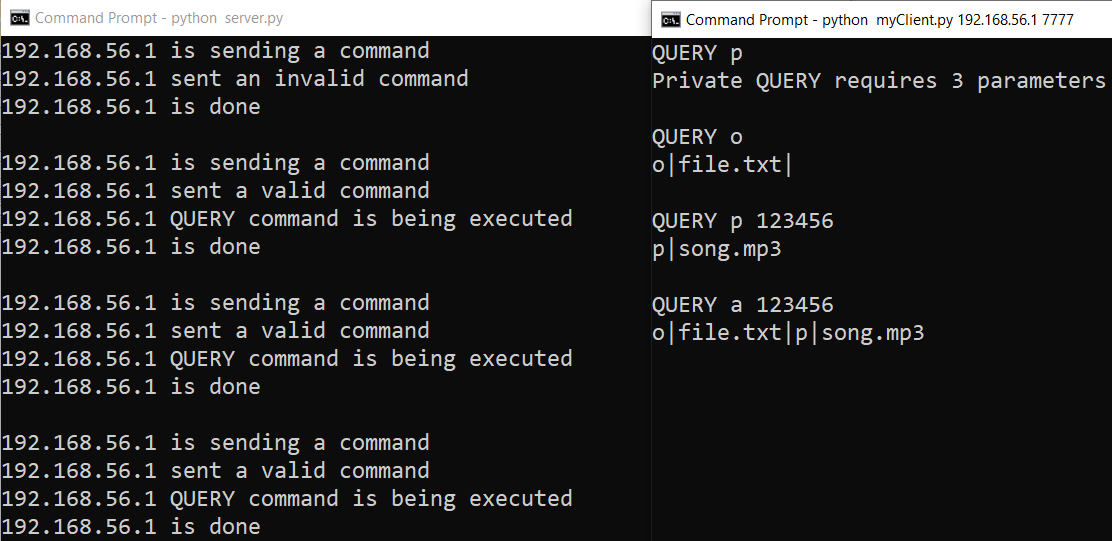


Figure 6- Query command

GET commands: In *figure below*, the client requests to download the files that are available on the server.

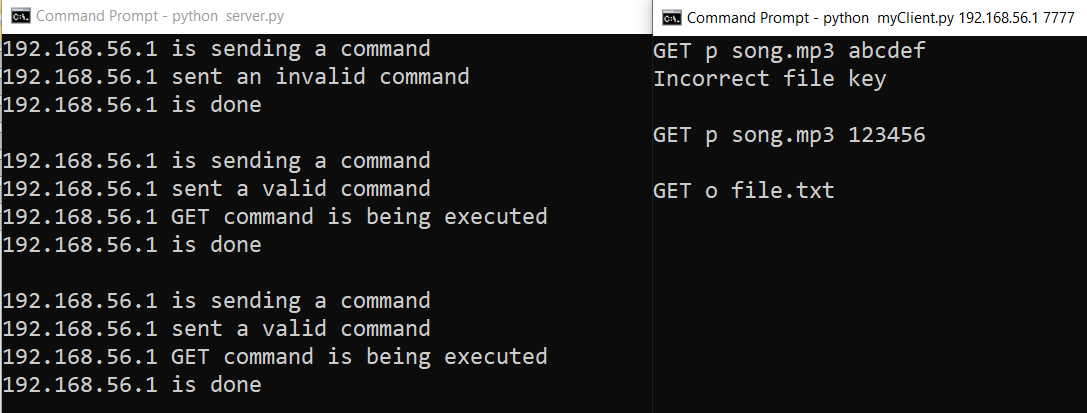


Figure 7- GET command.

If a file sent is corrupted, the receiver of the file informs the sender via a negative Status Code. It is up to the client implementation to inform the user that the file was not sent/received correctly. *Figure* shows a scenario where the server receives a corrupted file from a client.

If a client and the server lose connection, the server aborts that entire exchange and waits for another message from another client. The client may attempt to reconnect, but the previous session will not be restored. *Figure* shows server console output when a client loses connection.

# References:

Python SHA256 Hashing Algorithm: Explained. (2021, November 03). Retrieved from Datagy: https://datagy.io/python-sha256/