Documentation of FTP

Shervin Emrani

1 Introduction

This document provides detailed documentation for the implementation of a client-side FTP protocol using Python. The code enables a user to upload, download, and parallelize files between the client and the server. It also allows the user to fetch information about files available on the server, all while providing basic encryption for data transmission.

2 Code Overview

The client-side script performs the following tasks:

- Establishes a connection with an FTP server.
- Allows users to authenticate using a password.
- Provides functionality for file uploads, downloads, parallelization, and fetching file lists.
- Encrypts and decrypts data for basic security using a simple reversing algorithm.

3 Functions and Components

3.1 Main Program Structure

The client socket is created in the main block. The host address is set to `127.0.0.1` (localhost) and the port to `8080`. The client connects to the server and presents a list of available commands to the user.

```
host = '127.0.0.1'
port = 8080

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
```

The program then enters a loop where the user can input commands. However, authentication is required before any command can be executed. The default value for the authentication variable is `False`, and only becomes `True` after the correct password is provided.

3.2 Command Handling

Users can enter several types of commands:

- `upload <filename>` - Uploads a file from the client to the server.
- `download <filename>` - Downloads a file from the server to the client.
- `para <filename>` - Parallelizes a file between the client and server.
- `fetch` - Fetches a list of available files from the server.

3.2.1 Upload Command

The `upload` command follows the pattern:

```
upload <filename>
```

If the file exists on the client-side, the file is opened in binary mode, encrypted, and then sent to the server. The process ensures that the client sends the file contents after sending the command.

```
if listOfCommands[0] == "upload" and len(listOfCommands) == 2:
    requestedFile = listOfCommands[1]
    try:
        with open("files/" + requestedFile, 'rb') as file:
            sock.send(bytes(encrypt(msg), 'ascii'))
            file_data = file.read()
        sock.sendall(encrypt(file_data))
    except FileNotFoundError:
        print(Fore.RED, "*no such file exists*", Fore.RESET)
```

3.2.2 Download Command

The `download` command follows the pattern:

```
download <filename>
```

The client sends a download request to the server and waits for a confirmation. If the file exists, it is downloaded and saved locally.

```
if listOfCommands[0] == "download" and len(listOfCommands) == 2:
    sock.send(bytes(encrypt(msg), 'ascii'))
    response = decrypt(sock.recv(1024)).decode('ascii')
    if response == "downloadConfirmed":
        downloadedFileData = decrypt(sock.recv(1024))
        with open("files/" + listOfCommands[1], 'wb') as file:
            file.write(downloadedFileData)
        print(Fore.LIGHTGREEN_EX, "the file has been downloaded", Fore.
            RESET)
```

3.2.3 Parallelization Command

The `para` command allows parallelization of a file with the server. It follows the pattern:

```
para <filename>
```

The client checks whether it has the file before attempting to parallelize it with the server. The server's response determines whether the parallelization is successful.

```
if listOfCommands[0] == "para" and len(listOfCommands) == 2:
    haveFile = False
    try:
        with open("files/" + listOfCommands[1], 'rb') as file:
            pass
        haveFile = True
    except FileNotFoundError:
        haveFile = False
        print(Fore.RED, "could not find the file in clientSide to
            parallelize", Fore.RESET)
    if haveFile:
        sock.send(bytes(encrypt("para " + listOfCommands[1]), 'ascii'))
        paraResponse = decrypt(sock.recv(1024)).decode()
        if paraResponse == "SameFileHere":
            try:
                with open("files/" + listOfCommands[1], 'rb') as file:
                    file_data = file.read()
                    sock.sendall(encrypt(file_data))
                    paraOrNot = decrypt(sock.recv(1024)).decode()
                    if paraOrNot == "successfullyParallelized":
                        print(Fore.LIGHTGREEN_EX, "the file has been
                            parallelized", Fore.RESET)
```

3.2.4 Fetch Command

The `fetch` command retrieves information about the files stored on the server. It does not require any arguments:

```
fetch
```

```
elif listOfCommands[0] == "fetch" and len(listOfCommands) == 1:
    print("----------------------------------------")
    print("fetching files' info from the server ...")
    sock.send(bytes(encrypt("fetch"), 'ascii'))
    fetchResponse = decrypt(sock.recv(1024)).decode('ascii')
    for fileNumber in range(int(fetchResponse)):
        fetchedFile = decrypt(sock.recv(1024)).decode('ascii')
        print(fetchedFile)
    print("----------------------------------------")
```

3.3 Authentication

Before any file-related operations can be executed, the client must authenticate with the server. The authentication process sends an encrypted password to the server and waits for a response.

```
else:
    sock.send(bytes(encrypt(msg), 'ascii'))
    response = decrypt(sock.recv(1024)).decode('ascii')
    if response == "incorrect":
        print(Fore.RED, "not a valid password", Fore.RESET)
    elif response == "correct":
        auth = True
        print(Fore.LIGHTGREEN_EX, "you are connected to the server",
            Fore.RESET)
```

4 Conclusion

This FTP client-side code implements a basic system for secure file transfers between a client and server. It supports commands for file upload, download, parallelization, and fetching file lists. Authentication is required to ensure that only authorized users can interact with the server. Data transmission is protected by a simple encryption mechanism that reverses the string contents of the data being sent.