FTP Server-Documentation

Ferdowsi University of Mashhad

Spring 2022-2023

Team Members:

S.Melika Ranaei

Amin Pasandideh

Mohammad Afshari

Table of contents

- **Client Side**
  - Overview
  - File Structure
  - Dependencies
  - Usage
  - Client Configuration
  - Class Structure
    - __init__(self)
    - connect_to_server(self)
    - send_command(self, command)
    - get_help(self)
    - list_files(self)
    - upload_file(self, file_path)
    - download_file(self, file_name)
    - delete_file(self, file_name)

- **Database Creation File**

Server File Documentation

Overview

The server file is responsible for setting up and running an FTP (File Transfer Protocol) server. It allows clients to connect, authenticate, and perform various operations such as uploading, downloading, listing, and deleting files on the server.

File Structure

The server file consists of a single Python file named server.py.

Dependencies

The server file requires the following dependencies:

- os module
- socket module
- threading module
- sqlite3 module
- datetime module

Usage

To start the server, execute the server file using Python:

```
python server.py
```

Server Configuration

The server can be configured by modifying the following variables in the Server class:

- SERVER_HOST: The IP address or hostname on which the server should listen. By default, it uses socket.gethostbyname(socket.gethostname()) to automatically retrieve the host IP address.
- SERVER_PORT: The port number on which the server should listen. The default value is 9090.
- BUFFER_SIZE: The buffer size used for sending and receiving data. The default value is 4096.
- SERVER_FILES_DIR: The directory path where the server stores the uploaded files. By default, it is set to /home/afsharino/Desktop/server_files/.

Class Structure

The server file defines a single class called Server. This class encapsulates the server functionality and provides methods for handling client connections and executing client commands.

Class Methods

- __init__(self): The class constructor initializes the server configuration variables.
- start_server(self)

  Description: This method is responsible for starting the server by binding it to a specific host and port, listening for incoming connections, and creating a new thread for each client connection.

  Functionality:

  - Creates a server socket and binds it to the specified host and port.
  - Starts listening for incoming connections.
  - Accepts client connections and creates a new thread to handle each client.

- handle_client(self, client_socket, client_address):

  Description: This method handles communication with a connected client. It receives commands from the client, performs the requested operations, and sends back the appropriate responses.

  Functionality:

  - Authenticates the client by verifying their credentials.
  - Sends a welcome message to the client upon successful authentication.
  - Enters a loop to receive and process client commands until the client logs out.
  - Executes the requested commands such as listing files, uploading files, downloading files, deleting files, and providing help.

    - Sends appropriate responses to the client based on the command execution results.

- get_help(self, client_socket):

  Description: This method sends a list of available commands and their functionality to the client.

  Functionality:

    - Constructs a help message with the list of commands and their descriptions.
    - Send the help message to the client.

- list_files(self, client_socket):

  Description: This method retrieves the list of files in the server's file directory and sends it to the client.

  Functionality:

    - Retrieves the list of files in the server's file directory.
    - Formats the file information (name, creation date, modification date, and size) into a response message.
    - Sends the formatted response message to the client.

- upload_file(self, client_socket, args):

  Description: This method handles the uploading of a file from the client to the server.

Functionality:

- Receives the file name and size from the client.
- Checks if the file already exists on the server and appends "-copy" to the name if necessary.
- Receives the file data in chunks from the client and writes it to the server's file directory.
- Sends a success message to the client upon successful upload.

- download_file(self, client_socket, args):

  Description: This method handles the downloading of a file from the server to the client.

  Functionality:

  - Receives the file name from the client.
  - Checks if the file exists on the server.
  - Sends the client a message indicating the acceptance of the download and the file size.
  - Waits for the client's confirmation to start the file transfer.
  - Sends the file data to the client in chunks.

- delete_file(self, client_socket, args):

  Description: This method handles the deletion of a file from the server.

  Functionality:

    - Retrieves the list of files in the server's file directory.
    - Checks if the specified file exists in the directory.
    - Deletes the file from the server if it exists and sends a success message to the client.

These methods provide the core functionality of the server, allowing it to handle client connections, authenticate clients, process commands, and perform file operations.

- authenticate_user(self, client_socket): This method handles the user authentication process. It prompts the client for a username and password, verifies the credentials against an SQLite database, and allows or denies access to the server.
- add_user_to_database(self, username, password): This method adds a new user to the SQLite database.
- delete_user_from_database(self, username): This method deletes a user from the SQLite database.
- human_readable(cls, num, suffix="B"): This is a class method that converts a file size in bytes to a human-readable format.
- find_longest_name(cls, names): This is a class method that finds the longest name in a list of names.

That concludes the documentation for the server file. Now, let's move on to the client file documentation.

Client File Documentation

Overview

The client file provides a command-line interface for interacting with the FTP server. It allows users to connect to the server, authenticate, and perform various operations such as uploading, downloading, listing, and deleting files.

File Structure

The client file consists of a single Python file named client.py.

Dependencies

The client file requires the following dependencies:

- socket module
- os module

Usage

To start the client, execute the client file using Python:

```
python client.py
```

Client Configuration

The client can be configured by modifying the following variables in the Client class:

- SERVER_HOST: The IP address or hostname of the FTP server.
- SERVER_PORT: The port number of the FTP server. The default value is 9090.
- BUFFER_SIZE: The buffer size used for sending and receiving data. The default value is 4096.

Class Structure

The client file defines a single class called Client. This class encapsulates the client functionality and provides methods for connecting to the server and executing client commands.

Class Methods

- __init__(self): The class constructor initializes the client configuration variables.

  connect_to_server(self):

  Description: This method establishes a connection to the server by creating a client socket and connecting it to the server's host and port.

  Functionality:

  - Creates a client socket using the `socket.socket()` function.

    - Calls the `socket.connect()` method to connect the client socket to the server's host and port.
    - Prints a success message upon successful connection.
    - Catches any connection errors that may occur and prints an error message.

- send_command(self, command): This method sends a command to the server.

- get_help(self): This method retrieves the help message from the server and displays it to the client.

- list_files(self):

  Description: This method requests a list of files available on the server and displays the list to the client.

  Functionality:

    - Sends a "LIST" command to the server to request the list of files.
    - Receives the server's response, which contains the list of files.
    - Extracts and displays the file list to the client.

- upload_file(self, file_path):

  Description: This method uploads a file from the client to the server.

  Parameters:

- `file_path` (str): The path of the file to be uploaded.

Functionality:

- Opens the specified file in binary mode and reads its contents.
- Sends an "UPLOAD" command to the server along with the file name and file size.
- Sends the file data to the server in chunks.
- Receives the server's response to confirm the successful upload.

- download_file(self, file_name): Description: This method downloads a file from the server to the client.

  Parameters:

  - `file_name` (str): The name of the file to be downloaded.

  Functionality:

  - Sends a "DOWNLOAD" command to the server with the file name.
  - Receives the server's response to confirm the availability of the file.
  - If the file is available, it receives the file data from the server in chunks and writes it to a local file.

- delete_file(self, file_name):

  Description: This method deletes a file from the server.

  Parameters:

- `file_name` (str): The name of the file to be deleted.

Functionality:

- Sends a "DELETE" command to the server with the file name.
- Receives the server's response to confirm the success or failure of the deletion.

Database Documentation File

The `database.py` script is responsible for creating and initializing an SQLite database to store user information. It defines a function `create_database()` that establishes a connection to the database, creates a table for user information, and commits the changes.

Function Signature:

```
def create_database():
```

Functionality:

1. Establishes a connection to the SQLite database (or creates a new one if it doesn't exist).
2. Creates a cursor object to execute SQL queries.
3. Creates a table named "users" to store user information if it doesn't already exist. The table has three columns:
   - `id`: An auto-incrementing integer primary key.
   - `username`: A text column to store the username. The usernames must be unique.
   - `password`: A text column to store the password.
4. Commits the changes made to the database.

5. Closes the database connection.

Usage:

The script can be executed directly to create the database by calling the `create_database()` function. If the `database.db` file does not exist, it will be created. If the file already exists, the script will connect to the existing database and create the "users" table only if it doesn't exist.

Conclusion: FTP Server-Client Project

The FTP (File Transfer Protocol) Server-Client project is a network application that allows clients to connect to a server and perform various file management operations such as uploading, downloading, listing, and deleting files. The project consists of two main components: the server-side code and the client-side code.

The server-side code is implemented in the `server.py` file. It creates a server that listens for client connections and handles client requests. The server provides functionalities such as authentication, listing, uploading, downloading, and deleting. It uses sockets for communication and SQLite database for user authentication.

The client-side code is implemented in the `client.py` file. It provides a command-line interface for users to interact with the server. The client can connect to the server, authenticate with a username and password, and execute commands such as listing files, uploading files, downloading files, and deleting files. It uses sockets to communicate with the server.

The `database.py` script is responsible for creating and initializing an SQLite database that stores user information. It creates a table called "users" with columns for the user ID,

username, and password. The database is used for user authentication during server-client communication.

Overall, the project demonstrates a basic implementation of an FTP server-client system using socket programming and SQLite database. It provides essential file management functionalities and allows multiple clients to connect and interact with the server simultaneously. This project can serve as a foundation for building more advanced and secure file transfer systems or as a learning resource for understanding network programming concepts.