TAN TAO UNIVERSITY

FROM KNOWLEDGE TO THE STARS

School of Information Technology

Course CS440 - Computer Networks

REPORT

Socket Programming Project
"A Multi-user Chat Application"

Group 2 Members:

1. Nguyen Chanh Long - 2002043
2. Luong Trieu Vy - 2102164
3. Phan Le Kim Tinh - 2102058

June 16th, 2024

Table of contents

1. Introduction

The "Multi-user Chat Application" project is designed to provide a practical introduction to socket programming in Python. This project aims to develop a real-time chat application where multiple users can connect to a central server and communicate with each other. The project covers essential aspects of socket programming, including client-server architecture, message broadcasting, and handling multiple clients concurrently using multithreading.

2. Project Overview

The purpose of this project is to implement a socket-based communication system that allows users to sign up, log in, and communicate via a client-server architecture. The system is designed to handle multiple clients, support private and group chat modes, and ensure secure communication through the use of JWT (JSON Web Token) for authentication.

2.1. Features

User Authentication: allows users to establish their unique presence within the platform. Through the Sign Up functionality, users can create a fresh account by inputting necessary information like a username, password, and account name. Once an account is set up, users can use the Login feature, entering their credentials to obtain an access token, granting them entry to their account and the system's offerings.

Secure Communication: this feature guarantees the safety of information exchange between the client and server. The use of JWT tokens is a crucial part of this, ensuring that all communication is secure, maintaining user trust, and protecting sensitive data.

Chat Functionality: to foster interaction between users, the system provides both Private Chat and Group Chat features. In the Private Chat, users can send personal messages to specific users, ensuring a private and intimate conversation. On the other hand, the Group Chat allows users to send messages to all connected clients, promoting community interaction and engagement.

Database Integration: for storing user and authentication data, the system integrates with MongoDB. This ensures efficient and safe storage of both user information and authentication tokens in separate collections, enabling quick retrieval and secure handling of user data while maintaining system organization and integrity.

2.2. Overview of the system

The socket project is built on a client-server architecture, where the server is responsible for managing incoming connections and facilitating communication between clients. The clients, on the other hand, connect to the server, authenticate themselves using JWT tokens, and can then send and receive messages. This architecture allows for real-time communication between multiple clients, making it ideal for applications that require instant data exchange, such as chat applications.

Project Flow:

1. The user interacts with the application on their device, filling out a registration form with their username, password, and account name, and then hits "Send".
2. This triggers a POST request to the /signup endpoint on the server-side, with the user's details sent as the request body.
3. The server receives this request and checks if a user with the same username already exists in the MongoDB collection. If the user does not exist, the server stores the new user's details in the MongoDB collection.
4. The user fills out a login form with their username and password, and then hits "Send".
5. This triggers a POST request to the /login endpoint on the server-side, with the user's credentials sent as the request body.
6. The server receives this request and verifies the credentials against the stored user data in the MongoDB collection. If the credentials are valid, the server generates a JWT token and sends it back to the user.
7. The client connects to the server using socket programming. The client authenticates itself using the JWT token received during the login process. Once authenticated, the client can send and receive messages. Messages can be either private (sent to a specific user) or public (broadcasted to all connected users).

3. Implementation Details

3.1. Languages: The project is primarily implemented in Python.

3.2. Frameworks: The main framework used for building the chat application is Socket, with optional use of Flask for any web-based components.

3.3. Libraries: Key libraries used include:

1. socket: For establishing connections and communication between the server and clients.
2. threading: For handling multiple clients concurrently.
3. subprocess: For executing system-level commands within the application.
4. pymongo (MongoClient): For interacting with MongoDB for data storage.

3.4. Environments: The project can be run in any environment that supports Python, such as local machines or cloud-based environments like AWS or Azure.

3.5. Design Choices

3.5.1. Server-Side Design: The server-side architecture is designed to manage multiple client connections simultaneously and ensure real-time communication between clients.

Socket Initialization: The server initializes a server socket using Python's socket library. The server uses a class-based design encapsulated in the SocketServer class found in

InitializeSocketServer.py. It initializes a server socket using the TCP/IP protocol and binds it to a specified host and port. The socket is set to non-blocking mode to handle multiple connections efficiently.
Connection Handling: The server listens for incoming connections with *"self.server_socket.listen(5)"* , allowing up to 5 simultaneous connections in the queue. When a client connects, the server accepts the connection using *"self.server_socket.accept()"*, which returns a new socket object for the client and the address bound to the socket.
Client Management: The server keeps track of connected clients and their nicknames. When a new client attempts to connect, the server accepts the connection and stores the client's socket and address information. This is managed using a dictionary where the keys are client sockets and the values are the respective client addresses. Connected clients are managed using two lists: *"self.clients"* for the client sockets and *"self.nicknames"* for client nicknames. These lists ensure that messages can be routed to the correct clients.
Message Broadcasting: To broadcast messages from one client to all others, the server reads messages from each client and sends the message to every other connected client. This ensures that all clients receive the same message in real-time. Messages are broadcast to all connected clients using the *"send_to_other_clients"* method. This method iterates over all client sockets in the *"self.clients"* list and sends the message to each one.
Concurrent Handling: Each client connection is managed in a separate thread, which listens for incoming messages and processes them accordingly. To handle multiple clients concurrently, the server uses multithreading. Each client connection is managed in a separate thread, created with *"threading.Thread(target=self.handle_client, args=(client,))"* . This allows the server to manage multiple clients simultaneously without blocking.

3.5.2. Client-Side Design:

Socket Initialization: the setup involves initializing a client socket and connecting it to the server's IP address and port. The client socket is configured to be blocking, ensuring synchronous communication with the server. The client is implemented in the SocketClient class within InitializeSocketClient.py. It initializes a client socket and connects to the server using *"self.client_socket.connect((settings.SERVER_HOST, settings.SERVER_PORT))"*.
User Input Handling: The client handles user input by reading messages from the user and sending them to the server. This is done in a loop to allow continuous message input. The input is taken from the console and sent to the server via the socket.
Message Receiving: Clients receive and display messages from the server. To receive messages from the server, the client runs a separate thread that listens for incoming messages and displays them to the user. Incoming messages are received in a separate thread using the *"receive_messages"* method. This method continuously listens for messages from the server and displays them to the user, ensuring that the client can receive messages asynchronously while sending others.

3.6. Project Structure

3.6.1. **Handlers:** are classes or functions responsible for handling specific tasks in a chat application. This handler interacts with the database to retrieve or store user information and authentication tokens, and "UserHandle " for the purpose of user information management and "AuthHandle " user authentication management.

UserHandle: is responsible for managing user-related operations. It interacts with the database to perform various tasks such as retrieving, creating, updating, and deleting user information. This class uses the *"user_connected"* instance from *"api.utils.db_conn"* to establish a connection to the user-related collections in the database.

- Method:
  + get_user_by_username(self, username): This method fetches user information based on the provided username. It queries the database to find and return the user details.
  + create_user(self, user_data): This method inserts a new user into the database. It takes a dictionary user_data containing the username, password, and name_account, and uses the database connection to store this information.
- The UserHandle class is typically instantiated during operations that involve user data management, ensuring that user-related interactions with the database are handled efficiently and securely.

AuthHandle: is designed to manage authentication-related operations. It focuses on the retrieval and storage of authentication tokens, ensuring that user sessions are handled securely. This class uses the *"auth_connected"* instance from *"api.utils.db_conn"* to connect to the authentication-related collections in the database.

- Method:
  + get_auth_token(self, username): This method retrieves the authentication token associated with a given username. It queries the database to find and return the token.
  + create_auth_token(self, auth_data): This method inserts a new authentication token into the database. It takes a dictionary auth_data containing the username and the token, and stores this information in the database.
  + delete_auth_token(self, username): This method deletes the authentication token associated with a given username from the database.
- The AuthHandle class is utilized during user login and logout processes, ensuring that authentication tokens are correctly

managed and providing a secure mechanism for user sessions within the chat application.

3.6.2. **Models:** The models directory in the project defines the data structures used throughout the application. These structures are essential for managing user and authentication data, ensuring consistency and facilitating easy conversion to formats suitable for database storage.

serProperty: manages user information such as username, password, and nameaccount, and provides a to_dict() method to convert objects into a dictionary for storage in MongoDB.
AuthProperty: manages authentication information with the authToken attribute and provides a to_dict() method to convert objects to a dictionary for storage in MongoDB.

3.6.3. **Routes:** Flask manages the application's routes, determining how to handle HTTP requests from the client side.

API Design

1. **Signup an account**
    a. **Description:** The signup endpoint allows new users to register by providing their username, password, and name account. The server validates the input, checks for existing users, and if valid, stores the new user's information in the database.
    b. **Command:** POST
    c. **Domain:** http://localhost:5000/signup
    d. **Body:**

```
{
  "username": "exampletest2@gmail.com",
  "password": "secretpassword",
  "name_account": "user2"
}
```

    e. **Response:**
        i. **Successful responses:** 200

```
{
  "_id": "666da60e0669eafd17284386",
  "username": "exampletest2@gmail.com",
  "password": "secretpassword",
  "name_account": "user2"
}
```

        ii. **Error:** 400

```
{ "message": "Missing username, password, or nameaccount" }
{ "message": "User already exists" }
```

2. **Login an account**

a. **Description:** The login endpoint allows users to login by providing their username and password. The server validates the input, checks for existing users, and if valid, sends accessToken key for login to the client programming.
b. **Command:** POST
c. **Domain:** http://localhost:5000/login
d. **Body:**

```
{
  "username": "exampletest2@gmail.com",
  "password": "secretpassword"
}
```

e. **Response:**

i. **Successful responses:** 200

```
{
  "Name Account": "user2",
  "User Name": "exampletest2@gmail.com",
  "User Password": "secretpassword",
  "Access Token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."
}
```

ii. **Error:**

400:
```
{ "message": "Missing username or password" }
```
401:
```
{ "message": "User account does not exist!!!" }
```

3.6.4. **Services:** are classes or functions that undertake specific tasks in the application, such as connecting and communicating with servers and clients.

SocketClient: The purpose is to represent a client in a chat application.
Detail:

1. __init__(self): Initiates a socket connection to the server when the client is created.
2. auth_token(access_token): Verifies the authentication token sent from the client to ensure the user is logged in.
3. get_messages(): Receive and display messages from the server.
4. send_message(): Send message from client to server.
5. start(): Starts the authentication process and message transfer between the client and server.

SocketServer: Purpose is to represent the server in the chat application.
Detail:

1. __init__(self): Initializes a socket server to listen for connections from clients.
2. handle_client(client_socket): Handles messages from clients and sends messages back to other clients.

3. send_to_other_clients(message): Send messages to other clients.
4. send_private_message(sender_socket, target_nickname, message): Send private message from one client to another client.
5. start(): Start listening and processing connections from clients.

SocketClient and SocketServer are both classes that represent clients and servers in chat applications. SocketClient authenticates and sends/receives messages from the server. SocketServer listens for connections from clients and processes messages from different clients.

3.6.5. Utils

The utils file in your project is used to organize and import other components in the project such as routes, services, and models. This helps manage and organize the code in an organized and easier to understand manner. Contains classes *"UserConnection"* and *"AuthConnection"* to establish database connections for user and authentication collections, respectively.

https://drive.google.com/file/d/17oWxa_1ZEC3QMwAdapGt9I1fNDaVcJgY/view?usp=sharing

4. Challenges

4.1. Concurrent Connections

Managing multiple concurrent client connections in the server design necessitated the careful orchestration of threads to circumvent race conditions and ensure thread safety. This was accomplished using Python's threading module, which facilitated the management of each client connection in its own thread. However, this approach introduced several complexities. Race conditions, which occur when multiple threads simultaneously access shared resources and attempt to modify them, could lead to unpredictable behavior. Thread safety was another critical concern, as it was essential to ensure that the data structures used to store client information and messages were safe from concurrent modifications. Moreover, managing a large number of threads efficiently required careful resource management to prevent system resource exhaustion. Despite these complexities, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously. Therefore, the use of Python's threading module enabled managing a large number of threads efficiently required careful handling of system resources to avoid exhaustion.

4.2. User Authentication

Implementing user authentication involved securely storing and verifying user credentials. This required integrating a database and managing authentication tokens. Specific challenges included:

Security: Storing passwords securely and protecting user data from unauthorized access is paramount. We used hashing algorithms to securely store passwords and implemented measures to prevent common security vulnerabilities.

Database Integration: Integrating a database for user authentication required careful schema design and efficient query handling. We ensured that the database operations were optimized for performance and reliability. MongoDB was used for storing user information and authentication tokens.

Token management: Authentication token management to manage request sessions and securely verify and revoke tokens. We implemented token-based authentication using JSON Web Tokens (JWT), which included issuing tokens during login, verifying tokens for authenticated requests, and revoking tokens when necessary.

4.3. Private Messaging

Allowing users to send private messages required a method to identify and route messages to the correct recipient without broadcasting them to all clients. Routing: Implementing an efficient routing mechanism to deliver private messages to the intended recipient without exposing the message to others. We used unique identifiers for each client to manage private message delivery. Error handling: Managing potential errors in sending private messages, such as recipients being offline or disconnected.

5. Future Improvements

01. **Advanced User Authentication:** Enhancing user authentication mechanisms will provide a more secure and standardized way to manage user access.
    a. **OAuth Integration**: Integrate OAuth to provide a secure and standardized authentication mechanism. This will allow users to sign in with their existing accounts from other platforms such as Google, Facebook, or GitHub, simplifying the authentication process and improving security.
    b. **Multi-Factor Authentication (MFA)**: Implement MFA to provide an additional layer of security by requiring users to verify their identity using multiple methods, such as a password and a verification code sent to their mobile device. This significantly reduces the risk of unauthorized access.
02. **Graphical User Interface (GUI):** Developing a graphical user interface (GUI) will enhance the user experience and make the application more user-friendly. A well-designed GUI will make the application more accessible to users who may not be comfortable with command-line interfaces.
03. **Persistent Storage:** Using a database to store chat history and user information persistently will improve the functionality and reliability of the application. Integrate a database to store chat logs, user profiles, and other relevant data. This will ensure that chat history is preserved across sessions and users can retrieve past conversations.

6. Conclusion

The Multi-user Chat Application project provided a comprehensive introduction to socket programming in Python. It involved designing and implementing a client-server architecture, handling concurrent connections, and implementing additional features such as user authentication and private messaging. Through this project, we gained valuable insights into the challenges of network programming and learned effective strategies for managing multiple clients and ensuring robust communication.