

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



COMPUTER NETWORKS (CO3094)

Assignment

Chat Application

Advisor: Nguyễn Đức Thái
Students: Vương Hồng Linh - 2152728.
Lý Minh Quang - 2153722.
Nguyễn Quang Minh Tiến - 2150029.

HO CHI MINH CITY, AUGUST 2023



Contents

1	Introduction	2
1.1	Client-Server Model: an overview	2
1.2	Background knowledge	2
2	Requirement Analysis	2
2.1	Functional Requirements	2
2.1.1	User Interface and Registration	2
2.1.2	Contacts and Real-time Messaging	3
2.1.3	File transfer	3
2.2	Non-Functional Requirements	3
2.2.1	Performance	3
2.2.2	Scalability	3
2.2.3	Compatibility	3
2.3	Constraints	3
2.3.1	Technology Stack	3
2.3.2	Development Timeframe	3
2.4	Conclusion	4
3	Design	4
3.1	Introduction	4
3.2	Server-side Design	4
3.3	Client-side Design	4
4	Development	4
4.1	Server-side Design	4
4.1.1	Initialization and Setup	4
4.1.2	User Management	5
4.1.3	Message Broadcasting	5
4.1.4	Client Handler Threads	5
4.1.5	Server Main Loop	5
4.2	Client-side Design	5
4.2.1	Initialization and Setup	5
4.2.2	User Interface	5
4.2.3	Message Sending and Receiving	5
4.2.4	Threading for User Input and Message Reception	5
4.3	Overall Workflow	6
4.4	Considerations	6
5	Testing	6
5.1	User Registration Test	6
5.2	Send and Receive Public Messages	7
5.3	Private Chat	7
5.4	Check online User	8
5.5	Quit the Chat	8
6	Maintenance	9



1 Introduction

In this report, we demonstrate our elaboration of the Software Development Life Cycle by completing our class' assignment about developing a Chat Application with multiple clients hosted by a server. Within the scope of this class, we only develop a Chat App that works on a LAN connection.

1.1 Client-Server Model: an overview

The client-server model, or architecture, divides duties between servers and clients, who may be in the same system or interact through a computer network or the Internet. The client requests a server service from another software. The server runs applications that share resources and assign tasks to clients.

The client-server interaction uses a request–response message pattern and a shared communications protocol that specifies rules, vocabulary, and conversation patterns. TCP is used for client-server communication.

1.2 Background knowledge

Here are some technical terms that we use throughout our report:

- **Server:** A server is a computer or system that provides resources, data, services, or programs to other computers (clients) through a network.
- **Client:** A client is a device that makes the network request to the server. Typical clients include desktop PCs, laptops, tablets, smartphones, etc. The client sends a request to the server, which then responds to the client.
- **TCP:** Clients communicate with servers by protocols(follow rules). Clients and servers create a connection by using the 3-Way handshake technique. Because of the connection so this protocol is reliable but it can be slow.
- **Socket:** A fundamental concept that provides an interface for processes (applications) running on different devices to communicate with each other over a network. It serves as an endpoint for sending and receiving data in a network. A socket essentially acts as a combination of an IP address and a port number, allowing data to be routed to the correct process on a remote device. The socket is made up of two parts: IP Address and Port.

2 Requirement Analysis

The purpose of this document is to outline the requirements for the development of a chat application that operates on a client-server model using Python. The chat application will facilitate real-time communication between users through text-based messages.

2.1 Functional Requirements

2.1.1 User Interface and Registration

Users must provide a unique nickname for each session.

The client application should behave as a console application that is launched from the command

line. Users should be able to send and receive text messages in real-time. The application should maintain a message history, allowing users to view past conversations during a live session. The ability to open messages from previous sessions is optional.

2.1.2 Contacts and Real-time Messaging

Users must be able to get a list of current online users.

Users should be able to send text messages to their contacts in real-time. The application should support both one-on-one and group messaging.

It is optional to encrypt the messages exchanged between clients and the server to ensure data privacy.

2.1.3 File transfer

Clients are able to transfer files to others in both ways of communication (private or broadcast) and the receiver can accept or reject the incoming file.

This feature awards extra credit but unfortunately will not be covered in this assignment as we struggle with File Transfer Protocol (FTP)

2.2 Non-Functional Requirements

2.2.1 Performance

The application should have low latency to ensure real-time communication. The system should be able to handle a large number of concurrent users without significant performance degradation. The chat application should have high availability and minimal downtime. Messages should be reliably delivered without loss.

2.2.2 Scalability

The application architecture should be designed to scale horizontally to accommodate increased user loads.

2.2.3 Compatibility

The client application should be compatible with major operating systems (Windows, macOS, Linux). The server application should be deployable on common server platforms.

2.3 Constraints

2.3.1 Technology Stack

The application will be developed using Python for both the client and server components. Socket or other suitable real-time communication protocols will be used for instant messaging.

2.3.2 Development Timeframe

The development process should be completed within a specified timeframe. Regular milestones and progress updates should be provided to the lecturer.

2.4 Conclusion

The goals, features, and limitations of a client-server-based Python chat program are laid out in this requirement analysis. The goal of the software is to facilitate real-time communication among its users by meeting these criteria and making the necessary features easily accessible.

3 Design

3.1 Introduction

Overview: This phase focuses on the general structure of the two main parts of the system: `client.py` and `server.py`, which simulate how client-server model works.

3.2 Server-side Design

- Socket server initialization on the given host and port.
- Keep track of users and their associated connection sockets in a dictionary.
- Create a separate thread to handle communication for each client connection.
- Send all incoming client messages to the rest of the linked clients.
- Take out customers who have disconnected from the list.
- Maintain a persistent connection listener and thread creation mechanism.

3.3 Client-side Design

- Use a client socket to communicate with the server.
- Inquire about a user's login credentials.
- Begin a thread dedicated to receiving incoming messages.
- To communicate with the server, start a user input thread.
- Handle socket errors and disconnections

4 Development

4.1 Server-side Design

4.1.1 Initialization and Setup

Import necessary libraries (socket, threading).

Define the host and port for the server to listen on. Particularly for this assignment, we use port **50445** for the address.

Create a server socket and bind it to the address, consisting of a host and port.

Setting the standard format for the message as **utf-8**.

Listen for incoming connections.

4.1.2 User Management

Maintain a dictionary to store connected clients' information. The *clients* dictionary will have *address* as key and tuple of (*connection*, *nickname*) as value. Using *address*, which is unique across clients, is helpful for identification.

Handle new client connections in a separate thread using threading. Thread and use lock *blue_lock* to protect the clients' dictionary from simultaneous manipulations.

4.1.3 Message Broadcasting

Implement a function to broadcast messages (`broadcast_message(msg, client_address)`) to all connected clients.

Use a loop to continuously receive messages from clients and broadcast them.

4.1.4 Client Handler Threads

Define a function to handle individual client communication (`handle_client(connection, address)`).

Receive messages from a client and classify the messages' type among: terminate, get online users, send private messages, and broadcast messages.

Handle disconnections and remove disconnected clients from the list.

4.1.5 Server Main Loop

Continuously listen for incoming connections and spawn a new client handler thread for each connection (`start()`).

4.2 Client-side Design

4.2.1 Initialization and Setup

Import necessary libraries (`socket`, `threading`).

Define the host and port for the server to listen on. Particularly for this assignment, we use port **50445** for the address.

Create a client socket and connect to the server. (with `client.connect((SERVER, 50445))`)

4.2.2 User Interface

Implement a simple command line for the user and allow users to enter their username.

Provide input prompt `>>>` for getting the user's input.

4.2.3 Message Sending and Receiving

Define functions to send and receive messages from the server. Use a thread to listen for incoming messages from the server and display them to the user.

Implement input handling to send messages to the server.

4.2.4 Threading for User Input and Message Reception

Use threading. Thread to run the message reception loop in a separate thread.

Implement a function to handle user input and send messages to the server.

4.3 Overall Workflow

The server starts and listens for incoming connections. Clients connect to the server using their usernames (with a function `enter_username()`). Each connected client runs a message reception thread and a user input thread.

Clients send and receive messages through the server. Server broadcasts received messages to all connected clients.

Clients can send private messages to other clients with the syntax:

```
!private <client_name> <message>
```

Clients can check who is currently online with the syntax:

```
!online
```

Clients can exit the client program with the syntax:

```
!bye
```

4.4 Considerations

Ensure proper synchronization mechanisms (locks) to handle thread safety.


Implement logging to track server and client activities for debugging. Consider implementing message acknowledgments to ensure reliable message delivery.

This design structure provides a foundation for developing a basic client-server chat application using Python's socket and threading libraries. However, keep in mind that a production-level application would require additional features, security measures, and error handling to ensure a robust and secure communication platform.

5 Testing

5.1 User Registration Test

- Launch the server.



```
PROBLEMS  OUTPUT  TERMINAL  SQL CONSOLE  DEBUG CONSOLE
PS C:\Users\Admin> cd C:\Users\Admin\Desktop
PS C:\Users\Admin\Desktop> python server.py
[STARTING] server is starting on port 50445
[LISTENING] Server is listening on 192.168.43.61
█
```

Figure 1: the server recorded 2 connections

- Then, We'll test the username registration through the client program when a client enters a username that does not contain spaces.
- Check that the server has received and recorded the correct username

```
PS C:\Users\Admin\Desktop> python client.py
Enter username: quang
>>hi
>>
```

Figure 2: Client enter username and message

```
[STARTING] server is starting on port 50445
[LISTENING] Server is listening on 192.168.43.61
[NEW CONNECTION] ('192.168.43.61', 64705) connected.

[ACTIVE CONNECTIONS] 1
[('192.168.43.61', 64705), username = quang ] said: hi

```

Figure 3: The server notifies the username and the message correctly

```
G] server is starting on port 50445
[LISTENING] Server is listening on 192.168.43.61
[NEW CONNECTION] ('192.168.43.61', 64848) connected.

[ACTIVE CONNECTIONS] 1
[NEW CONNECTION] ('192.168.43.61', 64854) connected.

[ACTIVE CONNECTIONS] 2

```

Figure 4: the server recorded 2 connections

5.2 Send and Receive Public Messages

- Launch the server and at least two clients on the LAN
- A client enters a message and sends it.
- Check if the server received the message from the client and responded correctly.

5.3 Private Chat

- Launch the server and at least two clients on the LAN, making sure there are at least two users (but indeed we need 3 users to check that can it works as we need).
- A client enters a private message with the syntax !private [username], [message], and sends it.
- Check that the server sent a private message to the target user (username) and that the user received the correct message



```
PROBLEMS OUTPUT TERMINAL SQL CONSOLE DEBUG CONSOLE

PS C:\Users\Admin> cd C:\Users\Admin\Desktop
PS C:\Users\Admin\Desktop> python server.py
[STARTING] server is starting on port 50445
[LISTENING] Server is listening on 192.168.43.61
[NEW CONNECTION] ('192.168.43.61', 65157) connected.

[ACTIVE CONNECTIONS] 1
[NEW CONNECTION] ('192.168.43.61', 65159) connected.

[ACTIVE CONNECTIONS] 2
[NEW CONNECTION] ('192.168.43.61', 65162) connected.

[ACTIVE CONNECTIONS] 3
[('192.168.43.61', 65157), username = quang ] said: !private
quang2 hello quang2
[]

PS C:\Users\Admin> cd C:\Users\Admin\Desktop
PS C:\Users\Admin\Desktop> python client.py
Enter username: quang
>>[SERVER]: [NEW CONNECTION] User quang3 connected.
[SERVER]: [NEW CONNECTION] User quang2 connected.
!private quang2 hello quang2
>>[]

PS C:\Users\Admin> cd C:\Users\Admin\Desktop
PS C:\Users\Admin\Desktop> python client.py
Enter username: quang2
>>[SERVER]: [PRIVATE MESSAGE from quang]: hello quang2
[]

PS C:\Users\Admin> cd C:\Users\Admin\Desktop
PS C:\Users\Admin\Desktop> python client.py
Enter username: quang3
[SERVER]: [NEW CONNECTION] User quang2 connected.
>>[]
```

Figure 5: the chat room with 3 users

- As we can see in Fig 5 when the user uses the private chat, only the user who has permission the read can see it.

5.4 Check online User

- Launch the server
- A client enter command "!online"
- User will get the information of online user

```
PS C:\Users\Admin\Desktop> python client2.py
Enter username: quang
>>!online
>>[SERVER]: Online users (1): quang
[]
```

Figure 6: Information of online user

5.5 Quit the Chat

- Launch the server and a client
- A client enters !bye to exit the application.
- Check that the server received an exit message from the client and has handled the exit properly.

```
Enter username: quang
>>[SERVER]: [NEW CONNECTION] User quang3 connected.
[SERVER]: [NEW CONNECTION] User quang2 connected.
!private quang2 hello quang2
>>!bye
PS C:\Users\Admin\Desktop> |
```

Figure 7: Client chat !bye to quit the chat

```
[ACTIVE CONNECTIONS] 3
[('192.168.43.61', 65157), username = quang ] said: !private quang2 hello quang2
[('192.168.43.61', 65157), username = quang ] said: !bye
[DISCONNECTED] ('192.168.43.61', 65157) disconnected
|
```

Figure 8: The server received an exit message from the client and handled the exit properly.

6 Maintenance

Here are a few steps we can take to maintain our client-server programs for future uses:

- Regular Backup: Backup server-side codebase and configuration files regularly. Server user data and message history should be backed up to avoid data loss.
- Monitor software updates for Python, socket library, and threading library. Update the app for new features and security fixes.
- To maintain the server, monitor hardware health, storage space, and CPU utilization regularly. Check server logs for faults and odd activities.
- Security: Regularly examine and update authentication systems for robust user access. Keep up with server and client security best practices. Server-client data communication should be encrypted.
- Database Maintenance: Optimize searches, clear up obsolete data, and backups during regular maintenance operations for user information or message history databases.
- Performance Monitoring: Analyze server performance and scalability as the user base rises. Find bottlenecks and resource limits using monitoring tools.
- Error Handling and Bug Fixes: Address exceptions and crashes by reviewing error logs often. Fix user-reported and tested issues and unexpected behavior.
- Regularly test the application to discover and resolve any functional or performance problems. Automate testing to avoid regressions with new versions.



- User Support: Offer a place for reporting difficulties and asking inquiries. Address user comments and issues quickly.
- Documentation: Release and update application documentation, including setup instructions, use recommendations, and troubleshooting processes.
- Disaster Recovery Planning: Prepare for server outages and data breaches. Data recovery systems may restore user data after unforeseen incidents.
- Scaling: Plan for a higher load by scaling the application if the user base grows dramatically.
- Version Control: Use version control tools like Git to monitor changes and collaborate on code updates.
- Communication: Maintain open communication lines with the development team, users, and lecturers.
- Regular Review: Regularly assess the application's architecture and design to discover opportunities for enhancement or optimization.

This maintenance routine ensures your client-server chat application's operation, security, and stability. Regular maintenance prevents bugs, improves user experience, and keeps your program current with technology.

References

- [1] Our chat application - [Client-server-model-chat](#)
- [2] [Simple Chat room using Python](#)
- [3] [Python Socket Programming](#)
- [4] [Server chatroom sockets tutorial using Python3](#)