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


## COMPUTER NETWORK (CO3093)

### Assignment

# LAN Chatting Application

Students: Trần Nguyễn Gia Phát - 2153681
Phan Lê Tiến Thuận - 2153013
Lâm Gia Trúc - 2052764

HO CHI MINH CITY, AUGUST 2023

***Abstract**: This report presents a detailed overview of the development of a LAN-based chatting platform using the Python programming language. The development process adheres to the Software Development Life Cycle (SDLC), covering requirements analysis, design, implementation, testing, deployment, and maintenance. The report highlights each phase's significance and its contribution to the successful creation of the software application.*

# 1 Introduction

Effective communication is of the utmost importance in modern work and education environments. To address this need in local area networks (LANs), this report examines the development of a LAN-based chat platform using the Python programming language. The application is designed as a client-server model and allows seamless communication in a limited network environment.

This report describes the entire software development process following the Software Development Life Cycle (SDLC). The discussion includes the architecture, key features, implementation details, maintenance, and future chat platform enhancements. The ultimate goal is to fully understand the development path of the LAN chat platform and its importance in improving communication within LANs.

# 2 Requirements Analysis

The initial phase involves a thorough analysis of the project requirements, taking into account the client-server architecture and graphical user interface (GUI). The application is structured as follows:

- **Server Configuration:** The server component is assigned a hostname and port number, allowing it to listen for incoming client connections.
- **User Registration:** Clients are required to register upon connecting to the server. This involves creating a unique user profile for each client, which includes a username and potentially other user-specific information.
- **Messaging Functionality:** Users connected to the same LAN can engage in both group and private messaging. Group chats enable communication among multiple users, fostering collaborative discussions. Private messaging ensures discreet one-on-one conversations.
- **File Sharing:** Users connected to the same LAN can exchange files seamlessly through the application. This feature enhances collaboration by allowing users to share documents and media.

By identifying and understanding these requirements, the application's functionality and user experience can be effectively shaped in subsequent development phases.

# 3 System Design

Due to the limited development time and the relatively compact scope of the project, it consists of two main modules: the server and the client. The architecture revolves around these two components, facilitating communication in a local network environment. The server module is assigned a unique name and port number at startup. The client's task is to select the desired server by specifying the server name and the corresponding port. The customer then registers a valid username to access the chat room. The server module has functions to create socket-based connections for sending messages and files. Using threads, it can run multiple processes simultaneously. The server module performs critical tasks such as

new client registration and authentication. Use lists as data structures to manage message history and customer lists.

However, it is worth noting that data storage is limited; data is not saved while the server is offline. Front-end and back-end codes are integrated into the client module to facilitate user login and interaction. The graphical user interface (GUI) is divided into three main sections: the message and file input fields, the online user display area, and the message presentation area. Client-server interaction is central to this design, ensuring seamless communication and user engagement. The following sections discuss the technical implementation details of these modules and reveal subtleties that contribute to the functional coherence of the application.

# 4 Implementation

The implementation is based on Python programming language. It uses the PySide6 library for GUI development, threading for concurrent processes, socket handling for seamless connections, and UUID for efficient file sharing. The following subsections provide a detailed breakdown of each aspect of the implementation process.

## 4.1 Graphic User Interface

The PySide6 library augments the implementation with a rich set of tools for GUI development, ensuring an interactive and user-friendly experience.

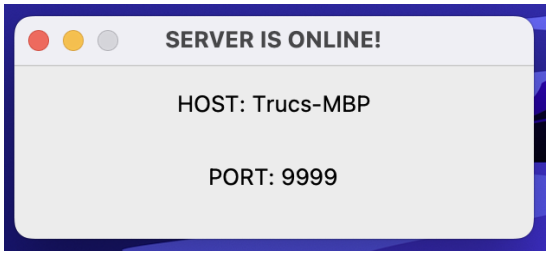


Figure 1: Server GUI

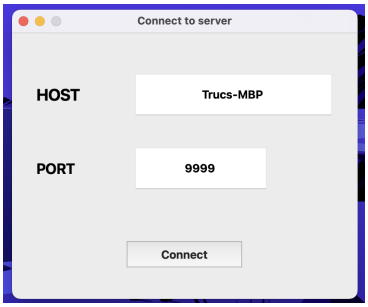


Figure 2: Client GUI

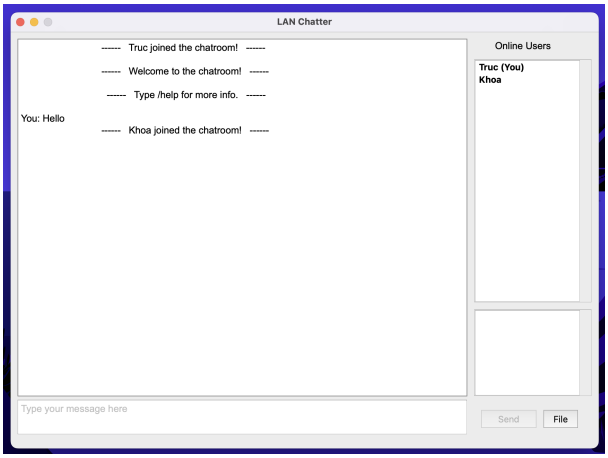


Figure 3: Chatting platform GUI

## 4.2 Threading and Socket Handling

To enable simultaneous execution of multiple tasks, threading is employed. This concurrent processing approach ensures that various operations, such as sending and receiving messages, can occur concurrently without causing delays or interruptions in the user interface. Socket handling plays a pivotal role in establishing connections between clients and the server, forming the basis for real-time communication.

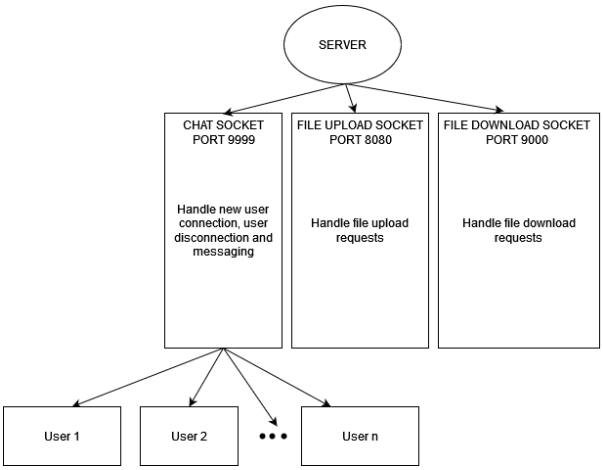


Figure 4: Threading in server

At the beginning, the server has three threads for different purposes: Handling user connections, user file uploading and user file downloading.

Whenever a new connection is established, the server creates a new thread corresponding to the user to handle behaviours such as sending messages (public or private) and exiting chat room.

## 4.3 File Sharing with UUID

The incorporation of Universally Unique Identifiers (UUIDs) in file sharing streamlines the process. UUIDs generate unique identifiers for files, preventing naming conflicts and simplifying the tracking of shared files. This approach enhances the reliability and efficiency of the file sharing functionality.

## 4.4 Source code

The complete source code of the project has been uploaded to GitHub. The repository can be accessed at: https://github.com/Zaphat/Tiem_Net_Hoang_Gia

# 5 Detailed System Design & Analysis

## 5.1 New Connection

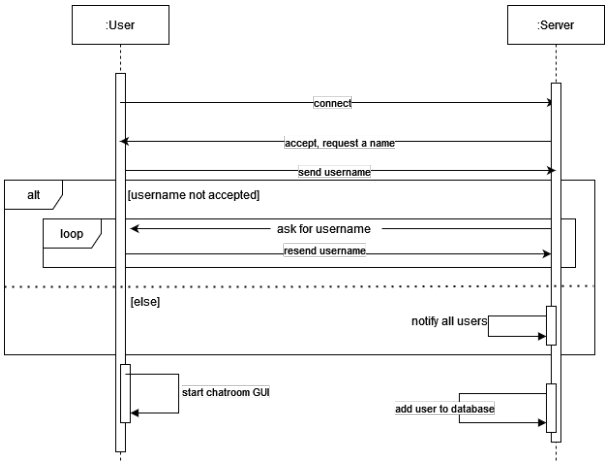


Figure 5: Sequence diagram of a new connection

On opening the client application, user will enter the host and port information to connect to the server. If successfully connected, the server will send a request for displaying nickname. This username must be unique, or else the user cannot log into the chat room. Next, the server notifies existing users to update their user list and stores the newcomer to the database. In the client side, the chat room GUI can now be opened.

## 5.2 Special Commands

```
help_pattern = re.compile(r'^\s*/help\s*$')
quit_pattern = re.compile(r'^\s*/quit\s*$')
clear_pattern = re.compile(r'^\s*/clear\s*$')
private_pattern = re.compile(r'^\s*/private.*$')
null_pattern = re.compile(r'^[\s\n]*$')

def _send_message_(self, message: str) -> None:
    # ... (content of the function)
```

Snippet 1: Handling special commands

Every message sent by user must go through local checking before being sent to the server. The function starts by checking if the message matches the /help pattern. If it does, the function displays a list of available commands in the chatroom and clears the text input field.

- If the message matches the /quit pattern, the function displays a message box indicating that the user has left the chatroom, then closes the application, closes the chat_socket, and exits the program.
- If the message matches the /clear pattern, the function clears both the chat history displayed in the UI and the text input field.
- If the message matches the /private pattern, it further checks if the message format is valid for sending a private message. If the format is valid, the function extracts the receiver's username and message content. It then sends the private message to the server using the send_to_server function and updates the UI to display the sent message.

- If the message is empty or contains only whitespace characters, the function simply clears the text input field.

If the message does not match any of the predefined patterns mentioned above, the function treats it as a general message and sends it to the server using the send_to_server function. It then updates the UI to display the sent message.

## 5.3 File Storage

```
    def upload_file(self) -> None:
        upload_socket = socket.socket(
            socket.AF_INET, socket.SOCK_STREAM)
        try:
            upload_socket.connect((server_host, 8080))
            # open dialog to choose file
            # save absolute path of the file
            file_path = os.path.abspath(
                QFileDialog.getOpenFileName(self, "Open File", "", "All Files (*.*)")
    [0])
            # extract file name from file path
            file_name = ntpath.basename(file_path)
            # check if file name is valid
            if not file_name:
                return
            # send metadata to server
            send_to_server(upload_socket,
                           f"/upload ({user_name.decode('utf-8')}) ({file_name})")
            # receive signal from server to start sending file content
            signal = upload_socket.recv(1024).decode('utf-8')
            if signal == "READY":
                with open(file_path, 'rb') as file:
                    file_data = file.read()
                    upload_socket.sendall(file_data)
        except:
            self.ui.textBrowser.append(
                "                              ------   Cannot connect to the server!
    ------                            \n")
        finally:
            upload_socket.close()
```

Snippet 2: Upload file (Client)

```
def on_file_upload(client_socket) -> None:
    try:
        metadata = client_socket.recv(2048)
        # /upload (filename) (sender)
        structure = re.compile(r'^(/upload)\s\((.{2,16})\)\s\((.+)\).*$')
        _, sender, filename = structure.match(
            metadata.decode('utf-8')).groups()

        client_socket.send('READY'.encode('utf-8'))
        # open directory for file storing
        if not os.path.exists(LOCATION):
            os.makedirs(LOCATION)
        # generate a unique token for the file
        TOKEN = uuid.uuid4().hex
        while TOKEN in FILES:
            TOKEN = uuid.uuid4().hex
```

```
        # store file information
        FILES[TOKEN] = (
            filename.encode('utf-8'), f'{LOCATION}/{TOKEN}.{filename.split(".")[-1]}',
    sender.encode('utf-8'))
        # store file in the directory
        with open(f'{FILES[TOKEN][1]}', 'wb') as file:
            while True:
                data = client_socket.recv(1024)
                if not data:
                    break
                file.write(data)
        # notify users that the file has been uploaded
        broadcast(
            f'{sender} has uploaded a file', "SERVER")
        # update the file list for all clients
        for client_socket, _ in CLIENTS.values():
            send_to_client(client_socket, '\x00UPDATE_FILE ' +
                           f"({filename}) ({TOKEN})")
    except:
        return
```

Snippet 3: Store File (Server)

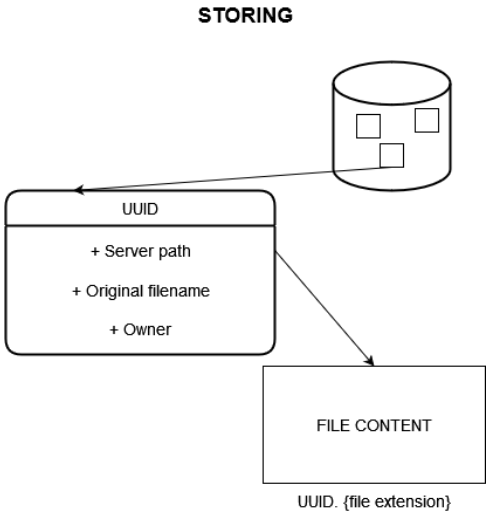


Figure 6: File Storage Schema

### Client-side `upload_file` Function

- The client initiates a connection to the server using a socket.
- It opens a dialog to allow the user to choose a file to upload.
- The chosen file's absolute path is extracted, and its base filename is extracted using the `os.path.abspath` and `ntpath.basename` functions respectively.
- The validity of the file name is checked. If it's not valid, the function returns.
- Metadata about the upload is constructed in the format: `/upload (username) (filename)`.
- The metadata is sent to the server using the `send_to_server` function.
- The client waits to receive a signal from the server to start sending the file content.
- If the signal is "READY", the client opens the file, reads its content in binary mode, and sends the data to the server.

### Server-side `on_file_upload` Function

- The server receives the metadata from the client indicating the sender's username and the filename.
- A regular expression pattern is used to parse the metadata and extract the sender and filename.
- The server sends a "READY" signal to the client to indicate that it's ready to receive the file content.
- The server checks if a directory exists for storing files. If not, it creates the directory.
- A unique token (`TOKEN`) is generated using the `uuid.uuid4().hex` function.
- If the generated token already exists in the `FILES` dictionary (used to track files), a new token is generated until a unique one is found.
- File information, including filename, path, and sender, is stored in the `FILES` dictionary.
- The server starts receiving file data in chunks (1024 bytes) from the client using a loop.
- The received data is written to the file in the server's storage directory.
- After the entire file is received, the server broadcasts a message to all connected clients to notify them that the sender has uploaded a file.
- The server sends an update command to all clients to update their file lists.

## 5.4 File Retrieving

```
    def download_file(self, token) -> None:
        download_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            download_socket.connect((server_host, 9000))
            download_socket.send(token.encode('utf-8'))
            file_name = download_socket.recv(1024).decode('utf-8')
            save_path = QFileDialog.getSaveFileName(
                self, "Save File", file_name, f"")[0]
            if not ntpath.basename(save_path):
```

```
                return
            with open(save_path, 'wb') as file:
                while True:
                    data = download_socket.recv(1024)
                    if not data:
                        break
                    file.write(data)
                file.close()
                self.ui.textBrowser.append(
                    "                        ------   File has been saved to your machine
    -----                        \n")
            except:
                self.ui.textBrowser.append(
                    "                    ------   ERROR: Failed to download attachment
    ------                    \n")
            finally:
                download_socket.close()
```

Snippet 4: Download File (Client)

```
def on_file_download(client_socket) -> None:
    try:
        TOKEN = client_socket.recv(1024).decode('utf-8')
        if TOKEN not in FILES:
            return
        client_socket.send(FILES[TOKEN][0])

        # send file content to client
        with open(f'{FILES[TOKEN][1]}', 'rb') as file:
            while True:
                data = file.read(1024)
                if not data:
                    break
                client_socket.send(data)
        client_socket.close()
    except:
        return
```

Snippet 5: Download File (Server)

## Client Side `download_file` Function

- The `download_file` function is defined on the client side. It takes a `token` parameter, which identifies the file to be downloaded.
- A socket, `download_socket`, is created using the AF_INET and SOCK_STREAM socket types for communication with the server.
- The client attempts to establish a connection with the server using the server's host address and port 9000.
- The `token` is sent to the server using `download_socket`'s `send` method, encoded as UTF-8.
- The client receives the file name from the server using `recv`. This name is used for selecting a save location via a file dialog.
- A save location is selected using a file dialog based on the received file name. If canceled or empty, the function returns.

- A file is created with the received name at the selected save path. Data received in 1024-byte chunks is written to the file.
- After data transfer, the file is closed, and a success message is displayed in the UI's text browser.

### Server Side `on_file_download` Function

- The `on_file_download` function is defined on the server side. It takes a `client_socket` parameter.
- The server receives a `TOKEN` from the client using `recv`. This token identifies the requested file in the `FILES` dictionary.
- If `TOKEN` is not found in `FILES`, the function returns.
- The server sends the file content associated with `TOKEN` to the client using `send`.
- The server opens the requested file in binary read mode and reads it in 1024-byte chunks.
- Each chunk is sent to the client via `client_socket`'s `send` method.
- After sending all file data, the client socket is closed.

## 5.5 Important Functions

| Function Name | Parameters | Description |
|---|---|---|
| private_message | client_socket, nickname, message | Handles private messages by parsing the message structure and sending it to the intended recipient's client. |
| send_to_client | client_socket, message | Sends messages to clients, padding to 1024 bytes if necessary, or breaking larger messages into parts for transmission. |
| broadcast | message, nickname, sender=None | Broadcasts messages to all clients or sends notifications to clients about events like user joins or leaves. |
| handle | client_socket, nickname | Handles client messages, processing private and public messages, and removing clients on error or disconnect. |
| update_client_list | new_client, storing_nickname | Updates the client list for all clients upon a new client's connection, ensuring everyone has the most up-to-date list. |
| on_connect | client_socket, address | Handles the initial connection of a client, requesting and storing their nickname, and starting threads for handling the client and updating the client list. |
| accept_chat | None | Accepts incoming client connections, creates threads for handling each connection, and notifying clients about new user joins. |
| on_file_upload | client_socket | Handles file uploads, storing file metadata and content, and broadcasting notifications about uploaded files to all clients. |
| accept_file_upload | None | Accepts incoming file upload connections, creating threads to handle each upload and notify clients about new file uploads. |
| on_file_download | client_socket | Handles file downloads by sending file content to the requesting client based on the provided file token. |
| accept_file_download | None | Accepts incoming file download connections, creating threads to handle each download request and provide requested files to clients. |

Table 1: Summary of functions (Server)

| Function Name | Parameters | Description |
|---|---|---|
| `receive` | None | Continuously receives and processes messages from the chat server. It handles various patterns in messages, such as updates, removals, new files, and null messages. |
| `send_to_server` | `client_socket` (socket), `message` (str) | Sends messages to the chat server. It manages messages that are longer than 1024 characters by splitting them into smaller chunks and sending them sequentially. |
| `_send_message_` | `self`, `message` (str) | Handles different commands and message types. It processes commands like "/help", "/private", "/quit", and "/clear", and sends regular messages to the server. |
| `send_message` | None | Invokes `_send_message_` with the message content from the UI's plain text input. Handles potential connection errors and displays appropriate messages. |
| `upload_file` | None | Establishes a connection to the server and sends a selected file's metadata, then transmits the file's content in chunks. Handles potential connection errors and displays appropriate messages. |
| `download_file` | `token` (str) | Connects to the server to request and download a specific file using its token. Saves the received file to the user's machine. Handles potential connection errors and displays appropriate messages. |
| `update_user_list` | `update` (str, optional) | Updates the list of online users displayed in the UI. It adds a new user to the list or refreshes the entire list if `update` is not provided. |
| `update_file_list` | `file_name` (str), `token` (str) | Updates the list of available files in the UI. Adds a new file entry with its name and token (for downloading). |
| `validate_nickname` | None | Validates whether the provided nickname is within the allowed pattern (2-16 characters, no special characters at the beginning or end, etc.). Returns `True` if the nickname is valid, `False` otherwise. |
| `enter_room` | None | Initiates the process of entering the chat room with the chosen nickname. Sends the nickname to the server and receives a response. Displays messages and starts the chat room interface if successful. |

Table 2: Summary of functions (Client)

# 6 Testing

During the testing phase of the project, a practical approach is taken to evaluate and showcase the functionality of the application. Although formal unit tests are not conducted, the application undergoes comprehensive demonstrations in different scenarios to ensure its performance and reliability.

## 6.1 Demonstration within the Same LAN - WiFi

The first testing form involves running the application within the same Local Area Network (LAN) environment using WiFi connectivity. This demonstration aims to validate the application's ability to establish connections and facilitate seamless communication among devices connected to the same LAN over a wireless network.

## 6.2 Demonstration within the Same LAN - Ethernet

Similar to the WiFi scenario, this form of testing explores the application's behavior within the same LAN environment. However, it utilizes Ethernet connectivity instead of WiFi. This test affirms the application's versatility in adapting to different LAN connectivity types.

## 6.3 Testing Results

The demonstrations consistently confirm that the application is strong and functional in all tested situations. It effectively connects users, enables messaging, and allows for file sharing without any major problems. The successful testing results show that the LAN-based chatting platform is reliable under different network conditions and device setups.

# 7 Deployment

The official deployment is currently pending release. However, you can access the code repository on GitHub at https://github.com/Zaphat/Tiem_Net_Hoang_Gia. Feel free to clone the project from the repository for local use. This approach allows you to explore and interact with the code base while waiting for the official deployment.

Detailed instructions for deploying the application can be found in the README.md file associated with the project's Git repository. The README.md file offers clear guidance on accessing and using the program, ensuring a straightforward and user-friendly deployment process.

# 8 Maintenance

Currently, the software is in an ongoing phase of testing, debugging, and refinement. The development group is actively engaged in fine-tuning the application based on user feedback and identified issues.

The maintenance phase entails the following key activities:

- Testing and Debugging: Rigorous testing is underway to identify any potential bugs, glitches, or performance issues. Detected problems are promptly addressed through debugging efforts, ensuring a smooth and error-free user experience.

- User Feedback Incorporation: Feedback collected from users who have tested the application plays a pivotal role. User input guides improvements, adjustments, and enhancements to align the application more closely with user expectations and requirements.
- Iterative Improvement: The maintenance phase follows an iterative approach, where frequent updates and refinements are made to address issues and add new features. This iterative process ensures that the application remains relevant, reliable, and efficient.
- Documentation Updates: As the application evolves, documentation such as user manuals, guides, and README files are updated to reflect the latest changes and improvements. This ensures that users have accurate and up-to-date resources at their disposal.
- Future Enhancements: User feedback and ongoing testing may uncover opportunities for future enhancements. These could include additional features, improved usability, or further optimization.

# 9 Future Enhancements

Looking ahead, there are opportunities to enhance its capabilities and usability. The following areas have potential for improvement:

- GUI Refinement: Enhancing the graphical user interface (GUI) can elevate the user experience. Incorporating modern design principles, user-friendly layouts, and visual elements can make the application more engaging and intuitive.
- Architectural Refactoring: Implementing the Model-View-Controller (MVC) pattern with object-oriented programming (OOP) principles can enhance the application's maintainability and scalability.
- WAN Chatting Platform: Expanding the application's scope beyond Local Area Networks (LANs) to support Wide Area Networks (WANs) would enable communication across geographically dispersed locations. This would require addressing challenges related to latency, security, and potential firewall restrictions.
- Enhanced Security: Implementing advanced security features such as encrypted messaging and user authentication mechanisms can bolster the application's privacy and protection against unauthorized access.
- Notification System: Incorporating a notification system that alerts users about new messages and updates, even when the application is in the background, can enhance real-time communication awareness.

The platform's future enhancements demonstrate its potential for growth and evolution. By embracing these possibilities, the LAN chatting platform can become a versatile and sophisticated communication tool that meets a wider range of networking needs.

# 10 Conclusion

Our application, with its user-friendly interface and critical features like messaging and file sharing, offers a seamless communication experience. Looking ahead, we see possibilities for improvement, such as refining the user interface, expanding to wide area networks, and enhancing security. These potential enhancements demonstrate the platform's adaptability and growth potential.