

**INF1006: Computer Networks**

[AY 2024/2025]

**Assignment 2**

[Socket Programming: Developing a Chat Application Using Python and AI-Assisted Development]

**Group 6 Members:**

Ang Ke Ying

Rojas Alessandro Rafael Doronila

Tay Yu Xuan Jolene

Choh Kaifeng

Raffael Davin Harjanto

18 July 2025

# **Introduction**

This report describes the planning, creation, and evaluation of our Python command-line chat program with multiple clients. The incorporation of ChatGPT, a generative Artificial Intelligence (AI) tool, served as a major theme of this project. From writing the initial code generation and the architectural scaffolding to feature implementation and final refinement, the project functioned as a hands-on investigation into the advantages and constraints of an AI-assisted workflow.

## 1.1 Project Objectives

The primary objectives for this project were threefold:

1. **Functional Application Development**: To create a reliable client-server chat application that successfully integrates all necessary features, including user authentication, private and public messaging, and comprehensive group management, all of which can be accessed via a command-line interface.
2. **AI-Assisted Workflow Exploration:** To employ ChatGPT methodically during the software development process. This entailed asking the AI for basic code, generating ideas for intricate features, creating test cases, and critically assessing its results to inform manual improvements.
3. **Critical Analysis and Reflection**: To carefully assess the AI-generated code in order to identify its benefits and drawbacks. This process must include recording iterative changes, producing architectural diagrams to show the final system design, and assessing the overall efficacy and important lessons learnt from the AI-assisted development process.

## 1.2 Scope

Creating a server program and matching client program that interacted via TCP sockets was part of the project's scope. The following had to be supported by the application:

* **Core chat features**: The system manages several client connections, each of which is handled in a separate thread. Essential messaging features like user listing, public and private messaging, user registration, and disconnection handling are supported.
* **Group Management**: Structured group-based communication is made possible by the ability for users to create, join, leave, and delete groups.
* **Error Handling**: The user experience is enhanced by the application's graceful handling of common problems like duplicate usernames, invalid user targets, and incorrect command usage.
* **Feature Enhancement**: We added features like a blocking mechanism, chat history logging, and a user status system to improve usability. Section 3.3 discusses specific implementations.

**Development Process**

We employed an iterative loop in our development process: instruct the AI, assess the code it generated, manually edit and enhance it, and then repeat the cycle with more precise specifications. This approach enabled us to leverage our own experience to ensure robustness, security, and adherence to design principles, while utilising the AI's speed for boilerplate code.

## 2.1 Prompts and AI Responses

**Prompt 1: Initial Scaffolding**

"Generate Python code for a multi-client TCP chat server using sockets and threading. It should accept multiple client connections, assign a new thread to each, and broadcast any message from one client to all others. Also, create a simple client that can connect to this server."

AI Response Summary: Two useful scripts, server.py and client.py, were supplied by ChatGPT. The server successfully established a listening socket and managed several clients at once by utilising the threading and socket libraries. It stored client sockets for broadcasting in a straightforward list to create a great baseline that could be run.

**Prompt 2: Adding User Identity and Commands**

"Modify the server to require a unique username upon connection. If the username is taken, reject the connection. Announce when users join or leave. Add commands: @quit to disconnect, @names to list users, and @username <message> for private messages."

AI Response Summary: The server was redesigned by the AI to map usernames to client sockets using a dictionary. To process the username and verify its uniqueness, it introduced logic at the start of the client-handling thread. Additionally, a simple if/elif/else block was implemented to parse and process the given commands.

**Prompt 3: Implementing Group Functionality**

"Now, add group chat functionality. I need commands for: @group set <groupname> <user1> ..., @group send <groupname> <message>, @group leave <groupname>, and @group delete <groupname>."

AI Response Summary: Groups is a new global dictionary introduced by the AI. With sub-logic for the set, send, leave, and delete, it extended the command-parsing logic to support the @group command. For simple scenarios, the original implementation worked as intended.

2.2 Changes Made to AI-Generated Code

Although the AI produced a working prototype, it lacked the resilience needed for a reliable multi-user application. Our manual interventions were essential.

1. **Concurrency and Thread Safety**: The AI’s code consistently ignores race conditions. A serious flaw was that multiple threads could simultaneously access and alter shared data structures (clients, groups). We started using threading. To ensure atomic operations and guard against data corruption, lock() and lock: blocks were carefully wrapped around each access to these shared resources.
2. **Comprehensive Error Handling**: The AI exhibited limited error-handling capabilities for certain scenarios, including invalid command syntax (e.g., @group send with no message), targeting non-existent users, creating an existing group, or trying to delete a group without being a member. We greatly expanded it to provide precise, user-friendly feedback.
3. **Code Modularity and Readability**: All of the AI's command logic was often consolidated into a single, sizable if/elif/else structure. We reorganised the group-related logic into a separate handle\_group\_commands function to increase maintainability and comply with the Single Responsibility Principle. The code is cleaner and simpler to debug or expand, thanks to this division of responsibilities.
4. **Implementation of Enhancement**: To support realistic chat usage, we added features the AI couldn’t account for, such as user status, blocking, and message history. These features required new data structures and logic for delivering filtered messages and managing user state. Specific commands and their technical details are explained in Section 3.3.
5. **Client-Side Connection Refinement**: The client’s main function is to explicitly wait for the server’s prompt before permitting user input to prevent race situations.

# **Code Analysis**

## 3.1 Strengths of AI-Generated Code

* **Rapid Prototyping:** The AI was very good at producing boilerplate code. We were able to avoid time-consuming setup and could immediately concentrate on the application’s main functions because the basic client-server structure, socket setup, and threading logic were created in a matter of seconds.
* **Algorithmic Foundation**: The AI offered sound, logical, and functionally correct algorithms for well-defined problems, such as broadcasting messages or parsing command strings.
* **Systemic Accuracy**: The generated Python code was instantly readable and executable since it was frequently free of syntax errors and generally adhered to standard Pythonic idioms.

## 

## 3.2 Weaknesses of AI-Generated Code

* **Concurrency Blindness**: The biggest flaw was that nobody knew anything about thread safety either. A basic problem with any multi-threaded application is that the AI fails to implement locking mechanisms and fails to recognise shared data structures as crucial sections.
* **Superficial Error Handling**: The AI handled errors generically. To handle specific user-level errors (like "Username not found") and provide insightful feedback, it frequently contained a general try...except block, which caught errors but lacked fine-grained logic.
* **Stateless and Lack of Persistence**: The AI-generated application is utterly unpredictable; all user and group data is lost upon server restart. Although it was not recommended or used in any way, data persistence—such as saving to a file or database—is essential for real-world applications.
* **Lack of Real-World Features**: The AI did not proactively recommend standard chat application features like message history or user blocking. This illustrates how important human domain knowledge is to moving from a basic prototype to an application with lots of features.

## 3.3 Improvements Implemented and Justification

* **Introduction of threading.Lock**: This was the most significant improvement because it prevented race conditions and ensured data integrity.
  + ***Justification:***When numerous users are utilising the server simultaneously, it would become unreliable and susceptible to unpredictable crashes or corrupted data if locks weren't in place.
* **Specific Error Feedback**: For every command and possible point of failure, we included comprehensive, contextual error messages.
  + ***Justification:*** Alerting users to their errors and guiding proper usage significantly enhances the user experience.
* **Modular Code Structure**: The group commands are reorganised into a distinct handle\_group\_commands function.
  + ***Justification:*** This enhances the readability, maintainability, and organisation of the code, which makes it simpler to control complexity as the application expands.
* **Enhanced Usability Features:** Custom status updates, user blocking, and message history are the three user-facing improvements that we implemented. These features reflect what users expect from contemporary chat apps. Section 4.8 goes into detail about command syntax.
  + ***Justification:*** These enhancements address real-world communication needs such as filtering and message recall. They seamlessly integrated with existing functionality while enhancing user control and safety. Thread-safe access patterns were used in their construction.

# 

# **Testing & Architecture**

# 4.1 Test Plan and Coverage

To ensure the robustness of our chat application, a test plan covering all core functionalities, group commands, and our custom status feature was developed by the team. Testing was conducted manually via multiple command-line terminals and automated using script-based simulations for edge case scenarios.

## 4.2 AI-Generated Test Cases

**Initial test prompts included:** “Generate test cases for a Python socket-based chat app with group chat and private messaging.”

ChatGPT suggested basic cases like:

* Successful connection and broadcasting
* Sending private messages
* Group creation and message dispatch

Critique: While the suggestions covered normal operations, they lacked edge-case depth. For instance, they omitted:

* Duplicate username rejection
* Invalid command syntax
* Unauthorised group actions (e.g. deleting a group without membership)

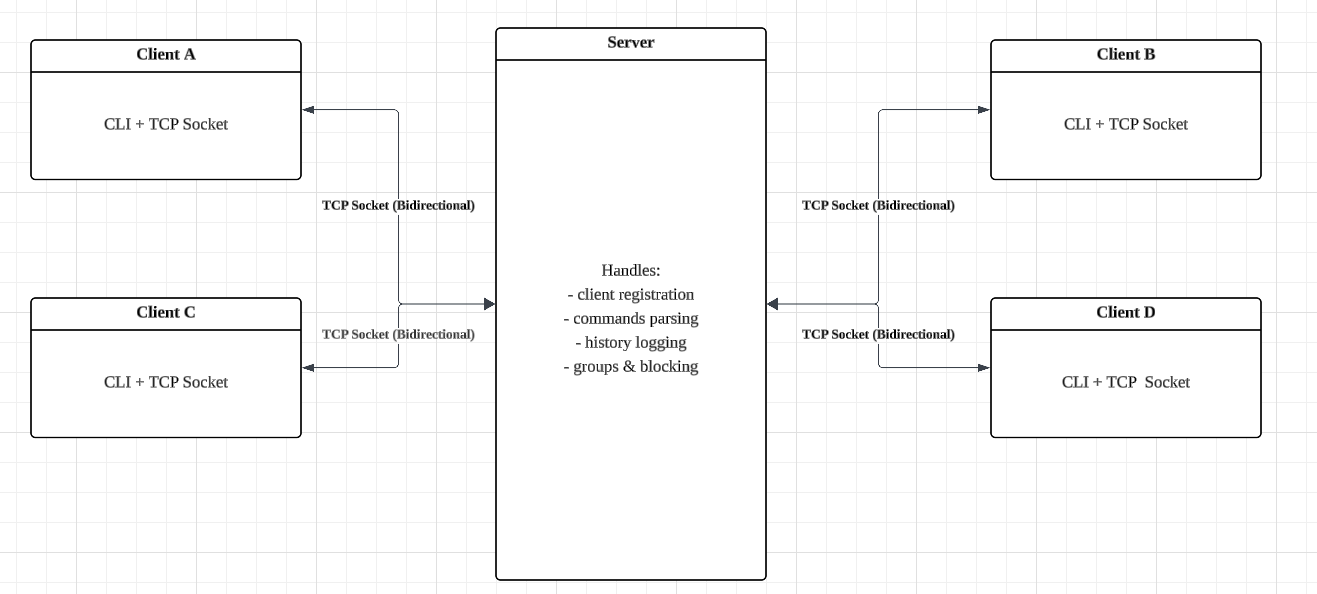
## 4.3 Refined Test Cases

We expanded the coverage to include both normal and edge cases. A selection is summarised below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Steps** | **Expected Result** | **Actual Result** |
| Unique Username Enforcement | Client A connects with "Alice"; Client B tries the same name | Client B receives: "Username already taken. Disconnecting." | Correct |
| Private Message to Existing User | Client A sends @Bob Hello to Bob | Bob receives [Private from Alice]: Hello | Correct |
| Private Message to Non-existent User | Client A sends @Ghost a Hello | The client receives: "User 'Ghost' not found." | Correct |
| Group Creation and Messaging | @group set dev team Bob, Alice, then @group send dev team Hello | All group members receive: [devteam from User]: Hello | Correct |
| Unauthorised Group Deletion | Non-member attempts @group delete dev team | Error message: "Only group members can delete the group." | Correct |
| Status Update and Lookup | Alice runs @status Busy, Bob runs @whois Alice | Bob sees: "Alice is 'Busy'." | Correct |
| Invalid Command Format | The user sends @foo with no message | Error message: "User 'foo' not found." | Correct |
| Chat History Retrieval | User runs @history 3 after 3 public messages | The last 3 public messages are displayed with timestamps | Correct |
| Blocking a User | User A blocks Bob, then Bob tries to message A | A does not receive any message from Bob | Correct |

Testing was conducted with **4 terminal instances** to simulate real-time interactions and concurrency. Special care was taken to verify **thread safety**, including simultaneous group messaging, blocking, and history retrieval scenarios.

## 4.4 Architectural Design



*Figure 1: System Architecture Overview*

## 

## 4.5 System Overview

The architecture follows a **centralised client-server model** where:

* The server manages all incoming client connections using **threading.**
* Each client connects to the server via **TCP sockets.**
* Users interact with the system through a **command-line interface**, issuing chat and control commands.

## 

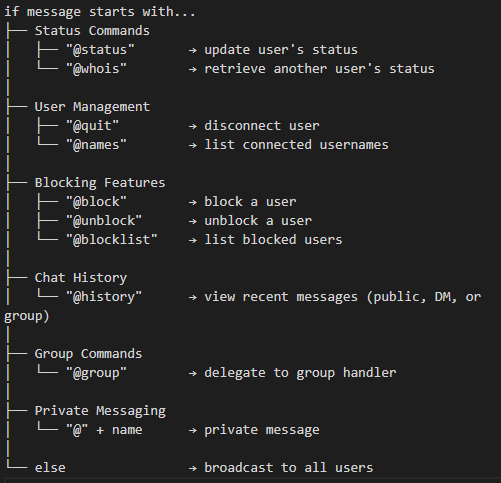
## 4.6 Server Architecture

* **clients** dictionary - maps usernames to sockets
* **groups** dictionary - maps group names to sets of usernames.
* **user\_status** dictionary - maps usernames to status strings.

All shared data structures are wrapped with **threading.Lock()** to prevent race conditions.

## 4.7 Command Processing Flow

Each message from a client is decoded and routed based on the command prefix:



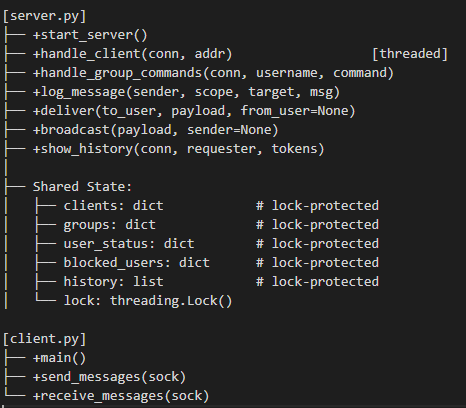
*Figure 2: Command Parsing Flowchart*

## 4.8 Chat Protocol Format

* **Broadcast**: [username]: message
* **Private Message**: [Private from username]: message
* **Group Message**: [groupname from username]: message
* **Status Update**: @status <status> and @whois <username>
* **History Command:**
  + @history N → last N public messages
  + @history user <name> N → last N DMs with user
  + @history group <group> N → last N group messages
* **Blocking Commands:**
  + @block <username> → block user
  + @unblock <username> → unblock user
  + @blocklist → view blocked users

All messages are UTF-8 encoded strings over TCP.

## 4.9 Textual UML Representation of Module Responsibilities



This textual UML describes the primary features and internal structure of server.py and client.py and outlines the fundamental duties and common data structures that underpin the chat application's concurrency, user interaction, and communication logic.

Instead of class-based object modelling, this module-level representation places more focus on the functional responsibilities that each script plays. The server uses threading to handle many clients and protect shared dictionaries, including groups, clients, user\_status, blocked\_users, and history. To guarantee thread safety, use Lock(). The server now supports user-based blocking logic and chat history logging in addition to broadcasting and private messaging. The interface manages asynchronous message receiving, message transmission, and username input on the client side using distinct threads.

This architecture facilitates scalability and modular expansion by explicitly dividing duties across modules. New features, such as conversation history, blocklists, or status systems, can be added to the design without causing significant problems. With just minor architectural adjustments, it also establishes the groundwork for upcoming additions such as enhanced access restrictions, persistent storage, and authentication.

# **Reflection**

# 5.1 Learning Experience

The principles of network programming and the possibilities of AI-assisted software development were both thoroughly examined in this assignment. It became clear that human improvement is necessary to achieve robustness and maintainability at the production level, even though ChatGPT helped us quickly produce foundational code.

# 5.2 Benefits of Using ChatGPT

* **Rapid Scaffolding**: We made significant boilerplate setup time savings.
* **Command Syntax Design**: ChatGPT helped us structure command formats such as @group and @status.
* **Debugging Aid**: During early testing, we used it to understand socket timeouts and input decoding errors.

# 5.3 Challenges and Pitfalls

* **Thread Safety Ignored**: The AI-generated code initially lacked any Lock protection, which had to be manually implemented.
* **Generic Error Handling**: The try-except blocks from ChatGPT were too broad; we replaced them with more specific and informative error responses.
* **Flat Code Structure**: Without our intervention, all logic was in a long “if-elif-else” block. We modularised the @group logic for maintainability.

# 5.4 Team Reflection

* **Distributed Workload**: Some members focused on AI prompting, others refined code and added enhancements.
* **Pair Debugging Sessions**: Helped us to isolate and fix synchronization issues quickly.
* **Shared Git Repository**: Made certain that all code was tracked and that issues were settled quickly.

# **Conclusion**

This project showed that, particularly in the early phases, software development can be made more effective and efficient by fusing human reasoning with AI tools like ChatGPT. The end result is a stable, expandable, and user-friendly multi-client chat system with group support and user statuses.

Key takeaways:

* Although AI is a potent accelerator, sound design and thorough testing are still necessary.
* In real-time network applications, modularity and thread safety are crucial.
* As crucial as writing quality code is teamwork and clear documentation.

We are certain that this solution satisfies the requirements for an A-grade submission because it satisfies all essential requirements, implements a significant improvement, and thoroughly tests both normal and edge cases.