SDI 3203: Network Fundamentals

Term Project

**Instructor:** Dr. Chenggang Wang

**Term Project Due Date**: April 29, 2025 (Tuesday), 11:59 PM

**Format**: Please submit a zip file, which should include your source code, a pdf file that summarize the execution of your code.

**Total Points**: 20% of your final grade

**Note**: This is a team project, each team has two students.

## Project Description

In this project, you will expected to choose a topic from below and implement the software/algorithm by yourself. This is a ***team project***; each student expected to team up with another student (2 students in a team) to do the programming and write the summarization.

The topic is: Client-server chat system

**Format**: you are expected to submit the source code and an experiment report that reflects the key steps of the execution of your code. Your code should follow the google programming guidelines [1], clear and detailed comments of the code are appreciated. For the experiment report, you need to follow the guidelines of Indiana University [2], you should also include the description of how to execute your code.

**When submit your project files, you should put all your code and file in a folder, and name the folder as the following naming conventions**:

* **class\_name-your\_name-team\_number-student\_number-date**

## Project Details

## ****1. Overview****

This project implements a simple push-style client-server chat application for a local network. The system allows multiple clients to connect to a single server, register unique usernames, and exchange messages in real-time. The server immediately pushes messages to recipients without requiring periodic polling.

### **Key Features**

* **User Registration**: Enforces a unique username for each client.
* **Real-Time Message Delivery**: Messages are “pushed” from the server to the recipient as soon as they are sent.
* **Online User List**: Clients can request a list of all currently active users.
* **Graceful Exit**: Clients can disconnect cleanly, notifying the server so their username is freed and no longer displayed.

## ****2. Architecture and Components****

### **2.1 Server**

* **Port Listening**
  + Listens on a designated port (e.g., 8080) for incoming connections.
* **Threaded Connections**
  + Spawns a new thread for each client connection, allowing concurrent handling of multiple clients.
* **User Management**
  + Maintains a dictionary (or map) of active username → socket.
  + Ensures usernames are unique. If a requested username is taken, the server rejects it.
* **Real-Time Push**
  + On receiving a message destined for <recipient>, the server immediately calls recipient\_socket.send(...).
* **Commands Handling**
  + **Registration**: Processes a registration command (e.g., server:register <username>).
  + **List Users**: Responds to a request for who is online (e.g., server:who).
  + **Exit**: Handles user exit, removing the user’s entry from the dictionary and closing the socket.

### **2.2 Client**

* **Connection**
  + Connects to the server using the server’s IP/hostname and port.
* **Username Prompt**
  + Requests a username from the user (e.g., Alice) upon startup.
  + Sends a registration command to the server to claim that username.
* **Two-Thread Model**
  + **Main Thread**: Waits for console input (input()) to parse commands and send data to the server.
  + **Listener Thread**: Continuously listens on the socket for incoming messages and prints them immediately to the user’s console.
* **Supported Commands**
  + **<recipient>:<message>** – Sends <message> to <recipient>.
  + **server:who** – Requests a list of online users from the server.
  + **server:exit** – Gracefully disconnects from the server, closing the socket and freeing the username.

## ****3. Detailed Requirements****

### **3.1 User Registration**

* **Client**
  + Prompt user for a username at launch.
  + Send a registration request to the server (e.g., server:register <username>).
* **Server**
  + Validates uniqueness of the username.
  + If accepted, records <username, socket> in its dictionary.
  + If the username is already taken, instructs the client to pick a different name.

### **3.2 Real-Time (Push) Message Delivery**

* **Server**
  + Upon receiving <recipient>:<message> from a client, immediately locates <recipient> in the dictionary.
  + If <recipient> is online, calls recipient\_socket.send(...) to push the message.
  + If <recipient> is offline, informs the sender that delivery failed (or queues the message, if desired).
* **Client**
  + The listener thread prints messages as soon as they arrive, without any polling logic.

### **3.3 Online User List**

* **Client**
  + Sends server:who to request the current list of active users.
  + The listener thread displays the received list.
* **Server**
  + Replies with a formatted list of currently registered usernames.

### **3.4 Graceful Exit**

* **Client**
  + User types server:exit to disconnect.
  + Notifies the server of the exit request, then closes the socket.
* **Server**
  + Removes the user from the internal dictionary.
  + Alerts other users if necessary (optional).

## ****4. Implementation Details****

1. **Imports**
   * Use Python’s built-in libraries: socket, threading, argparse (optional for command-line arguments).
2. **Server Module** (e.g., server.py)
   * **Parse Arguments**: --port <port\_number> for specifying the listening port.
   * **Start Server Socket**: server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) then bind() and listen().
   * **Connection Loop**: In a loop, accept() new client connections and spawn a thread to handle each client.
   * **Client Handler**: The thread responsible for:
     + Parsing registration commands (server:register).
     + Storing/validating usernames.
     + Handling message forwarding.
     + Responding to server:who and server:exit.
   * **Data Structures**:
     + users = {username: client\_socket, ...} for active users.
3. **Client Module** (e.g., client.py)
   * **Parse Arguments**: --host <server\_ip> and --port <port\_number> if desired.
   * **Establish Connection**: Create a socket and connect() to the server.
   * **User Registration**: Prompt for username and send server:register <username>.
   * **Listener Thread**: Continuously reads from the socket and prints any incoming messages to the console.
   * **Main Loop**: Reads user input. Depending on the format (<recipient>:<message>, server:who, server:exit), sends the appropriate command to the server.

## ****5. Usage Instructions****

1. **Start the Server**

python server.py --port 8080

* + The server will begin listening on localhost:8080.

1. **Start a Client**

python client.py --host server\_ip --port 8080

* + When prompted, enter a username, e.g., Alice.
  + If the username is accepted, you can begin sending messages.

1. **Sending a Message**
   * In the client console, type <recipient>:<message>, e.g.:

Bob:Hello Bob!

* + Bob’s client will immediately display Alice: Hello Bob!.

1. **Checking Who’s Online**
   * Enter server:who in the client console.
   * The server responds with a list of currently online users.
2. **Exiting**
   * Type server:exit.
   * The client notifies the server that you are disconnecting, then closes.

## ****6. Possible Extensions (Optional)****

* **Offline Messaging / Message Queues**
  + Store messages for users who are offline and deliver them once they reconnect.
* **Encryption**
  + Integrate TLS/SSL or other encryption libraries to secure messages in transit.
* **Private Group Chat or Chat Rooms**
  + Extend the server logic to handle group chat sessions with multiple recipients.
* **User Authentication**
  + Require a password for username registration to enhance security.
* **Graphical User Interface (GUI)**
  + Replace or supplement the command-line interface with a simple GUI using libraries like tkinter.

## Submission Requirements

* **Source Code:**  
  Submit all source files along with any scripts needed to set up the virtual environment.
* **Documentation:**  
  Include a README file with instructions on how to run your project and a brief report describing your design and implementation decisions.
* **Testing Evidence:**  
  Provide evidence (screenshots or video recording) showing that your chat system works as expected in the virtual environment.

## Evaluation

**Performance Levels**:

* **Excellent (90-100% of points)**
  + Fully meets requirements, shows strong understanding, robust and well-tested implementation, code is very clean and well-documented.
* **Good (80-89%)**
  + Meets most requirements with minor issues or omissions, code is mostly clean, only a few bugs or documentation gaps.
* **Needs Improvement (70-79%)**
  + Partially meets requirements, some notable bugs or missing features, code is somewhat disorganized or under-documented.
* **Incomplete (<70%)**
  + Fails to meet a significant number of requirements, has major errors or is non-functional in key areas, documentation is insufficient.

A screenshot of a computer

Description automatically generated

## Reference

[1]. <https://google.github.io/styleguide/>

[2]. <https://guides.libraries.indiana.edu/c.php?g=992698&p=7182653>