**Project: Chat Server System**

**Course:** Operating Systems

**Course Code:** CSC 302  
 **Branch & Section:** BSc CSC

**Semester & Year:** 5th Semester, 3rd Year

# **INTRODUCTION**

In the modern era of digital communication, real-time chat applications have become integral to both personal and professional interactions. These applications facilitate seamless communication between multiple users, enabling the exchange of messages, media, and other data in a fast, efficient, and user-friendly manner. To achieve this, chat systems employ various technologies, with **Socket Programming** playing a pivotal role in their architecture.

This project, titled **"Chat Server System,"** demonstrates the design and implementation of a basic yet functional chat application using **socket programming** in Python. The system enables multiple clients to connect to a server, send and receive messages, and communicate with one another. It includes advanced features such as **private messaging**, **broadcasting messages**, and handling multiple clients concurrently using **multi-threading**.

The project is a practical exploration of key concepts in networking, **inter-process communication (IPC)**, and software development. By leveraging **TCP/IP protocols**, the chat application ensures reliable message delivery, making it an ideal model for understanding the fundamentals of real-time communication systems.

# **OBJECTIVES**

The primary objectives of this project are:

1. **To build a client-server architecture** that allows multiple clients to connect and communicate with a central server.
2. **To facilitate real-time communication** by implementing features such as:
   * Broadcasting messages to all connected clients.
   * Sending private messages to specific clients.
3. **To explore multi-threading techniques** for handling multiple clients simultaneously without blocking other processes.
4. **To understand and implement inter-process communication (IPC)** using sockets as the core mechanism for data exchange between the server and clients.
5. **To monitor performance** by tracking server memory usage and execution time for key operations.
6. **To provide a user-friendly interface** where clients can interact seamlessly with the server and other clients.

# **DESCRIPTION**

1. **Server Functionality**:
   1. Listens for incoming connections from clients.
   2. Maintains lists of connected clients and their nicknames.
   3. Broadcasts messages to all clients or sends private messages based on the input command.
2. **Client Functionality**:
   1. Connects to the server using the provided IP address and port.
   2. Sends a nickname to the server for identification.
   3. Listens for incoming messages from the server.
   4. Allows users to input and send messages to the server.
3. **Multithreading**:
   1. The server uses threads to handle multiple client connections simultaneously.
   2. Each client runs separate threads for sending and receiving messages.
4. **Communication Flow**:
   1. The client sends its nickname upon connection.
   2. Clients can send regular messages (broadcast) or private messages (command /msg <nickname> <message>).
   3. The server processes and routes the messages appropriately.
5. **Socket Programming**:

Sockets provide an endpoint for sending and receiving data across a network. They are the backbone of this chat application, enabling communication between the server and clients.

The TCP protocol is used for reliable data transmission, ensuring that messages are delivered in order and without loss.

1. **Performance Evaluation:**  
    a. Memory usage monitored every 5 seconds by the server using psutil and reported in megabytes (MB).

b. Measures the time taken to send a message to clients.

# **FEATURES OF THE CHAT SYSTEM**

1. **Multi-Client Support**:

The system allows multiple clients to connect to the server and communicate in real time.

1. **Message Broadcasting**:

Messages one client sends are broadcast to all other clients connected to the server.

1. **Private Messaging**:

Clients can send direct, private messages to specific users by specifying their nicknames.

1. **Error Handling**:

The system includes mechanisms to handle invalid nicknames, disconnected clients, and other potential errors.

1. **Server Monitoring:**

The server monitors its performance, including memory usage, in real-time using the psutil library, ensuring efficient resource management even under heavy loads.

1. **User-Friendly Interface**:

Clients can easily interact with the system using a simple command-line interface, with prompts for actions such as setting nicknames and sending private messages.

# **TECHNICAL FOUNDATIONS**

**TCP/IP:**

The Transmission Control Protocol/Internet Protocol (TCP/IP) is a foundational suite of communication protocols used to connect devices on the internet and local networks. TCP ensures reliable data transmission by managing packet delivery, verifying data integrity, and handling retransmissions if packets are lost. IP, on the other hand, handles addressing and routing, ensuring data reaches the correct destination. This project uses TCP for its dependable communication, guaranteeing ordered and complete message delivery, a critical requirement for chat applications.

**IPC (Inter-Process Communication):**

Inter-Process Communication (IPC) allows processes to exchange data and coordinate actions. In the context of this project, IPC facilitates efficient communication between the server and clients, enabling real-time message exchange. By using sockets—a form of IPC—this application establishes connections between independent processes (clients and the server) across the network. IPC ensures synchronization and a seamless data flow, which is essential for the chat functionality.

**Multi-Threading:**

Multi-threading enables a program to perform multiple tasks concurrently by dividing work into separate threads that share the same memory space. In this chat application, the server employs multi-threading to handle multiple clients simultaneously. Each client connection is managed in a separate thread, ensuring smooth interaction even as the number of connected clients grows. Similarly, clients use multi-threading to separate the tasks of sending and receiving messages, ensuring responsiveness and uninterrupted communication.

These three concepts—TCP/IP, IPC, and multi-threading—are integral to the design and functioning of this chat application. Together, they create a robust, reliable, and efficient communication framework suitable for real-time applications.

# **CODE IMPLEMENTATION**

**Server-Side Code**

import socket

import threading

import time

import psutil

from datetime import datetime

# Server configuration

HOST = '127.0.0.1'

PORT = 12345

# Lists to manage clients and their nicknames

clients = []

nicknames = []

# Function to broadcast messages to all clients

def broadcast(message, sender\_socket=None):

start\_time = time.time()

for client in clients:

if client != sender\_socket:

try:

client.send(message.encode('utf-8'))

except:

if client in clients:

index = clients.index(client)

nickname = nicknames.pop(index)

clients.remove(client)

broadcast(f"{nickname} has left the chat.")

print(f"Broadcast Execution Time: {time.time() - start\_time:.4f} seconds")

# Function to handle private messaging for multiple recipients

def send\_private\_message(sender\_socket, recipient\_nicknames, message):

start\_time = time.time()

sender\_nickname = nicknames[clients.index(sender\_socket)]

failed\_recipients = []

for recipient\_nickname in recipient\_nicknames:

recipient\_nickname = recipient\_nickname.strip() # Remove extra spaces

if recipient\_nickname in nicknames:

recipient\_index = nicknames.index(recipient\_nickname)

recipient\_socket = clients[recipient\_index]

private\_message = f"Private message from {sender\_nickname}: {message}"

recipient\_socket.send(private\_message.encode('utf-8'))

else:

failed\_recipients.append(recipient\_nickname)

if failed\_recipients:

failed\_message = f"Failed to deliver message to: {', '.join(failed\_recipients)}"

sender\_socket.send(failed\_message.encode('utf-8'))

print(f"Private Message Execution Time: {time.time() - start\_time:.4f} seconds")

# Function to handle a client

def handle\_client(client\_socket):

try:

while True:

client\_socket.send("Enter your nickname:".encode('utf-8'))

nickname = client\_socket.recv(1024).decode('utf-8').strip()

if nickname not in nicknames and nickname.lower() != "you":

nicknames.append(nickname)

clients.append(client\_socket)

client\_socket.send("You have joined the chat!".encode('utf-8'))

client\_socket.send(

"To send a private message, type `/msg <nickname1,nickname2> <message>`.".encode('utf-8')

)

broadcast(f"{nickname} has joined the chat!", sender\_socket=client\_socket)

print(f"{nickname} joined the chat.")

break

else:

client\_socket.send("ERROR: Nickname already taken or invalid. Please try again.".encode('utf-8'))

while True:

message = client\_socket.recv(1024).decode('utf-8')

if message.startswith('/msg'):

try:

\_, recipient\_part, \*private\_message = message.split(' ')

recipient\_nicknames = recipient\_part.split(',')

private\_message = ' '.join(private\_message)

send\_private\_message(client\_socket, recipient\_nicknames, private\_message)

except ValueError:

client\_socket.send("ERROR: Invalid /msg format. Use `/msg <nickname1,nickname2> <message>`.".encode('utf-8'))

else:

sender\_nickname = nicknames[clients.index(client\_socket)]

broadcast(f"{sender\_nickname}: {message}", sender\_socket=client\_socket)

except Exception as e:

if client\_socket in clients:

index = clients.index(client\_socket)

nickname = nicknames.pop(index)

clients.remove(client\_socket)

broadcast(f"{nickname} has left the chat.")

print(f"{nickname} disconnected.")

client\_socket.close()

# Function to monitor server performance

def monitor\_performance():

while True:

print(f"Memory Usage: {psutil.Process().memory\_info().rss / 1024 \*\* 2:.2f} MB")

time.sleep(5)

# Start the server

def start\_server():

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind((HOST, PORT))

server\_socket.listen()

print(f"Server running on {HOST}:{PORT}")

# Start the performance monitoring thread

threading.Thread(target=monitor\_performance, daemon=True).start()

while True:

client\_socket, client\_address = server\_socket.accept()

print(f"New connection from {client\_address}")

thread = threading.Thread(target=handle\_client, args=(client\_socket,))

thread.start()

if \_\_name\_\_ == "\_\_main\_\_":

start\_server()

**Client-Side Code**

import socket

import threading

from datetime import datetime

# Client configuration

HOST = '127.0.0.1'

PORT = 12345

# Function to handle receiving messages

def receive\_messages(client\_socket):

while True:

try:

message = client\_socket.recv(1024).decode('utf-8')

print(f"[{datetime.now().strftime('%H:%M:%S')}] {message}")

except:

print("Disconnected from the server.")

client\_socket.close()

break

# Function to handle sending messages

def send\_messages(client\_socket):

while True:

message = input()

timestamp = datetime.now()

client\_socket.send(message.encode('utf-8'))

print(f"Message sent at {timestamp.strftime('%H:%M:%S')}")

# Start the client

def start\_client():

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client\_socket.connect((HOST, PORT))

# Start threads for sending and receiving messages

receive\_thread = threading.Thread(target=receive\_messages, args=(client\_socket,))

receive\_thread.start()

send\_thread = threading.Thread(target=send\_messages, args=(client\_socket,))

send\_thread.start()

if \_\_name\_\_ == "\_\_main\_\_":

start\_client()

# **CODE EXPLANATION**

### **Server Code Explanation**

1. **Connection Management**
   1. The server listens on a specified IP address (HOST = '127.0.0.1') and port (PORT = 12345) for incoming client connections using socket.listen().
   2. Once a connection is established, the server accepts the connection with server\_socket.accept() and starts a new thread (threading.Thread) to handle each client independently.
2. **Client Handling**
   1. Each client is assigned a nickname when they connect, which is validated to ensure it’s not already taken or invalid (e.g., not "you").
   2. The server maintains a list of connected clients and their corresponding nicknames (clients and nicknames lists).
   3. After a client joins, a welcome message is sent, and the server broadcasts a message to other clients announcing the new connection.
3. **Message Broadcasting**
   1. The broadcast() function sends a message to all clients except the sender.
   2. It iterates over the clients list and sends the message to each client using client.send(message.encode('utf-8')). This method is used for broadcasting regular messages from any client to all other clients.
4. **Private Messaging**
   1. The server supports private messaging, where a client can send messages to specific recipients using the /msg command.
   2. The server receives the message, checks if the specified nicknames exist in the nicknames list, and forwards the message to the corresponding client sockets.
   3. If any recipients are invalid, the server informs the sender with a failure message.
5. **Error Handling**
   1. If a client disconnects or sends an invalid message, the server removes them from the clients and nicknames lists, broadcasts that they have left, and closes the client connection.
6. **Performance Monitoring**
   1. The server uses the psutil library to monitor memory usage, printing the memory consumed by the server process at regular intervals (every 5 seconds).
   2. This is handled by a separate thread that continuously prints memory usage information.

### **Client Code Explanation**

1. **Connection Setup**
   1. The client establishes a socket connection to the server using client\_socket.connect((HOST, PORT)).
   2. After the connection is established, the client is prompted to provide a nickname, which is sent to the server.
   3. The server verifies the nickname, and if valid, the client is accepted into the chat.
2. **Message Handling**
   1. The receive\_messages() function runs in a separate thread and continuously listens for incoming messages from the server.
   2. When a message is received, it is printed to the console with a timestamp.
   3. The send\_messages() function handles user input. The client allows the user to send regular messages or private messages.
   4. When a message is entered, it is sent to the server.
   5. If the message is intended for private recipients, the client uses the /msg command format, specifying nicknames and the message.
   6. Otherwise, regular messages are broadcast to all clients.
3. **Command Support**
   1. The client supports sending private messages using the /msg <nickname1,nickname2> <message> format.
   2. This command is parsed in the client, and the message is sent to the server with the specified nicknames.
   3. The server then forwards the message to the corresponding clients.
   4. Regular messages sent without the /msg command are treated as broadcast messages and are sent to all connected clients.
   5. The server then broadcasts these messages to all other clients.
4. **Multithreading**
   1. The client uses two separate threads:
      1. One for receiving messages from the server (receive\_messages()).
      2. Another for sending messages to the server (send\_messages()).
   2. This allows the client to simultaneously listen for incoming messages and send messages without blocking either task.

**Libraries Used:**

1. **socket**
2. Provides low-level network interface for establishing client-server connections.
3. Used for creating sockets, connecting to the server, and sending/receiving data.
4. **Threading**

Enables multithreading, allowing the server to handle multiple clients concurrently and the client to handle sending and receiving messages simultaneously.

1. **psutil** (server-side only)

Used to monitor system resources like memory usage, helping evaluate server performance.

1. **Time**

Used to introduce delays (e.g., for periodic memory usage monitoring or debugging).

1. **Datetime**

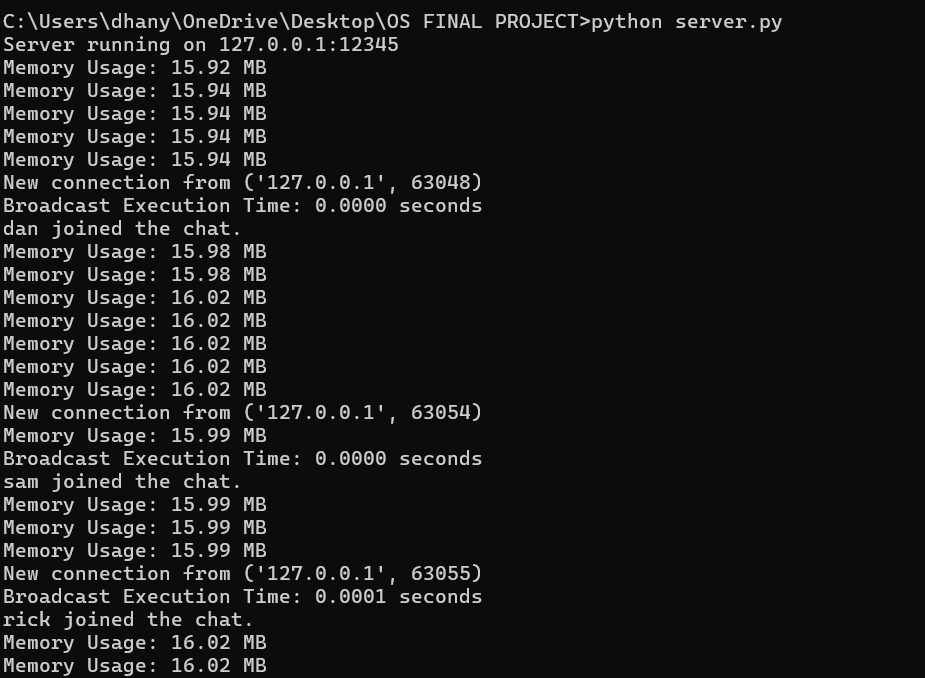
Provides timestamps for messages, improving the user interface by showing when a message was sent or received.

1. **Sys**

Facilitates clean program exits in case of errors or disconnections.

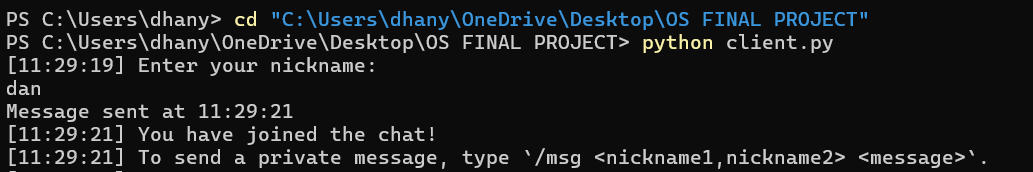
# **Output**

**Server Side:**

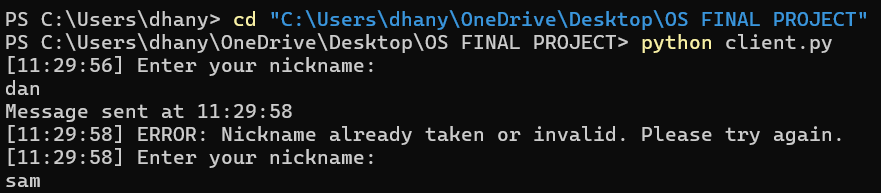


**Client Side:**

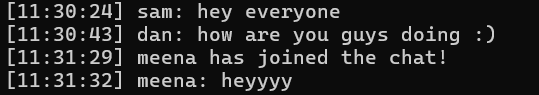
1. **Join the chat**



1. **Enter taken username**

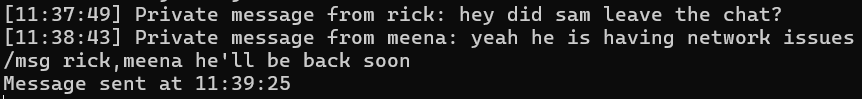


1. **Chat between multiple users**



1. **User left chat**



**5. Private messaging between 2 or more users** 

# **FUTURE GOALS**

1. **Security Enhancements:**
2. **Authentication:**  
   Add a login system where users authenticate with a username and password before joining the chat.
3. **IP Blocking:**  
   Introduce mechanisms to block malicious or unauthorized IP addresses attempting to connect to the server.
4. **Scalability:**
5. **Multi-Server Setup:**  
   Develop support for distributing the load across multiple servers to handle a large number of clients.
6. **Database Integration:**  
   Use a database (e.g., MySQL, PostgreSQL) to store user information, chat histories, and server logs for persistence and scalability.
7. **Advanced Features**
8. **Group Chats:**  
   Enable users to create or join specific chat groups or channels.
9. **Chat History:**  
   Implement a feature to store and retrieve chat histories for clients.
10. **Custom Commands:**  
    Add more commands, such as /ban, /mute, or /whois, to enhance moderation and interaction.
11. **Bot Integration:**  
    Introduce bots for automated responses, moderation, or fun activities like quizzes.

# **CHALLENGES FACED**

1. **Managing Multiple Connections:**

One of the biggest challenges was ensuring the server could handle multiple clients connecting and chatting at the same time. There were some issues initially with threads not working correctly, which caused the server to freeze or crash unexpectedly.

1. **Implementing Private Messaging:**

Setting up the private messaging system was tricky, especially figuring out how to parse the /msg command and make sure the messages reached the right person. There were also edge cases, like if someone entered a nickname that wasn’t online, which took extra effort to handle.

1. **Monitoring Performance:**

When we added memory usage monitoring, we realized the server wasn’t as efficient as we thought. We had to tweak how messages were processed to make sure the server didn’t slow down as more clients connected.

1. **Handling Errors:**

Making the chat system robust enough to handle unexpected situations, like clients disconnecting suddenly or sending invalid commands, was challenging. At first, these issues would cause errors that disrupted the chat, so we had to add more checks and testing to prevent that.

# **CONCLUSION**

In this project, a real-time multi-client chat system was successfully implemented using socket programming and multithreading. The server was designed to handle multiple client connections simultaneously, providing functionality for both message broadcasting and private messaging. The system ensures that clients can send and receive messages seamlessly, with added error handling mechanisms for disconnecting clients and managing invalid actions.

Through the use of multithreading, both the server and clients were able to perform tasks concurrently, such as listening for and sending messages without blocking the execution of other operations. The user-friendly command-line interface allows clients to interact with the system intuitively, supporting functionalities like private messaging and broadcasting. Additionally, the server monitored memory usage to ensure optimal performance.

Overall, this project demonstrated the application of socket programming and multithreading in creating a robust communication system, capable of handling multiple clients efficiently. The implementation successfully met the project objectives, providing a functional chat system with effective client-server interaction, message routing, and error handling. Future improvements could include adding encryption for secure communication and extending the system to support more advanced features like file transfer and user authentication.