PROJECT REPORT

ON

A DISTRIBUTED REAL-TIME MESSAGING SYSTEM

USING CLIENT-SERVER ARCHITECHTURE

SUBMITTED BY:

NAME: DOLLY KUMARI
RPS CLOUD USERNAME: 24NAG1279_U18

WIPRO NGA PROGRAM
C++ || LSP BATCH

PROJECT OVERVIEW

The project involves creating a distributed real-time messaging system using C++ and POSIX sockets, designed to facilitate instant communication between multiple users using Client Server Architechture. The system operates on a client-server model where the server manages multiple client connections and ensures real-time message distribution. Each client connects to the server, sends messages, and receives messages from other users, with message handling and communication taking place through multithreading to support concurrent user interactions.

The implementation demonstrates fundamental concepts in network programming and multithreading within a Linux environment. Key features include real-time messaging, concurrent client handling, and basic error management. Future improvements are planned to enhance scalability, security, and user experience by integrating advanced features like encryption, user authentication, and a graphical interface, ultimately aiming to create a more robust and user-friendly messaging system.

MOTIVATION

The motivation behind this project stemmed from the growing need for effective and scalable communication solutions in today's digital world. With the increasing reliance on real-time messaging for personal and professional interactions, there was a clear opportunity to explore and implement fundamental network programming concepts. Developing a distributed real-time messaging system provided a practical challenge to address issues such as concurrent user management, efficient message handling, and maintaining robust communication channels. This project not only aimed to deepen understanding of client-server architectures and multithreading but also to create a functional platform that demonstrates these principles in action, showcasing the practical application of theoretical knowledge in a real-world scenario.

FUNCTIONS USED

Client-Side Functions

1. `socket()`
   - **Description**: Creates a new socket for communication using the specified address family (AF_INET), socket type (SOCK_STREAM for TCP), and protocol (0 for default).
   - **Usage**: `client_socket = socket(AF_INET, SOCK_STREAM, 0);`
2. `connect()`
   - **Description**: Establishes a connection to the server using the specified socket and server address.
   - **Usage**: `connect(client_socket, (struct sockaddr*)&server_addr, sizeof(server_addr));`
3. `send()`
   - **Description**: Sends data through the socket to the connected server or client.
   - **Usage**: `send(client_socket, message.c_str(), message.size(), 0);`
4. `recv()`
   - **Description**: Receives data from the socket. Blocks until data is available or an error occurs.
   - **Usage**: `bytes_received = recv(client_socket, buffer, 256, 0);`
5. `close()`
   - **Description**: Closes the socket, releasing the associated resources.
   - **Usage**: `close(client_socket);`
6. `std::getline()`
   - **Description**: Reads a line of input from the standard input stream (`std::cin`) into a string.
   - **Usage**: `std::getline(std::cin, message);`
7. `std::thread()`
   - **Description**: Creates a new thread to run a specified function concurrently.
   - **Usage**: `std::thread(receive_messages, client_socket).detach();`

Server-Side Functions

1. `socket()`
   - **Description**: Creates a new socket for communication, similar to the client-side.
   - **Usage**: `server_socket = socket(AF_INET, SOCK_STREAM, 0);`
2. `bind()`

- **Description**: Binds the socket to a local address and port, allowing it to listen for incoming connections.
- **Usage**: `bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr));`

3. `listen()`
    - **Description**: Prepares the socket to accept incoming connections, specifying a maximum number of pending connections.
    - **Usage**: `listen(server_socket, 5);`
4. `accept()`
    - **Description**: Accepts an incoming connection request and creates a new socket for communication with the client.
    - **Usage**: `client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &client_len);`
5. `recv()`
    - **Description**: Receives data from the client socket, similar to the client-side.
6. `send()`
    - **Description**: Sends data to the connected client, similar to the client-side.
7. `close()`
    - **Description**: Closes the server socket or client socket, similar to the client-side.

These functions are fundamental for setting up and managing network communication in our chat application, handling both client and server operations effectively.

SOURCE CODE

//chat_server.cpp

```
#include <iostream>
#include <thread>
#include <mutex>
#include <map>
#include <vector>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

#define PORT 8080

std::mutex mtx;
std::map<int, std::string> clients;
std::map<int, int> client_sockets;

void handle_client(int client_socket) {
    char buffer[256];
    int bytes_received;

    // Receive client name
    bytes_received = recv(client_socket, buffer, 256, 0);
    std::string client_name(buffer, bytes_received);
    std::cout << client_name << " connected" << std::endl;

    // Add client to map
    mtx.lock();
    clients[client_socket] = client_name;
    mtx.unlock();

    while (true) {
        bytes_received = recv(client_socket, buffer, 256, 0);
        if (bytes_received <= 0) {
            break;
        }

        std::string message(buffer, bytes_received);
        std::cout << client_name << ": " << message << std::endl;

        // Relay message to all other clients
        for (auto& client : clients) {
            if (client.first != client_socket) {
                send(client.first, (client_name + ": " + message).c_str(), (client_name + ": " +
message).size(), 0);
            }
        }
    }
```

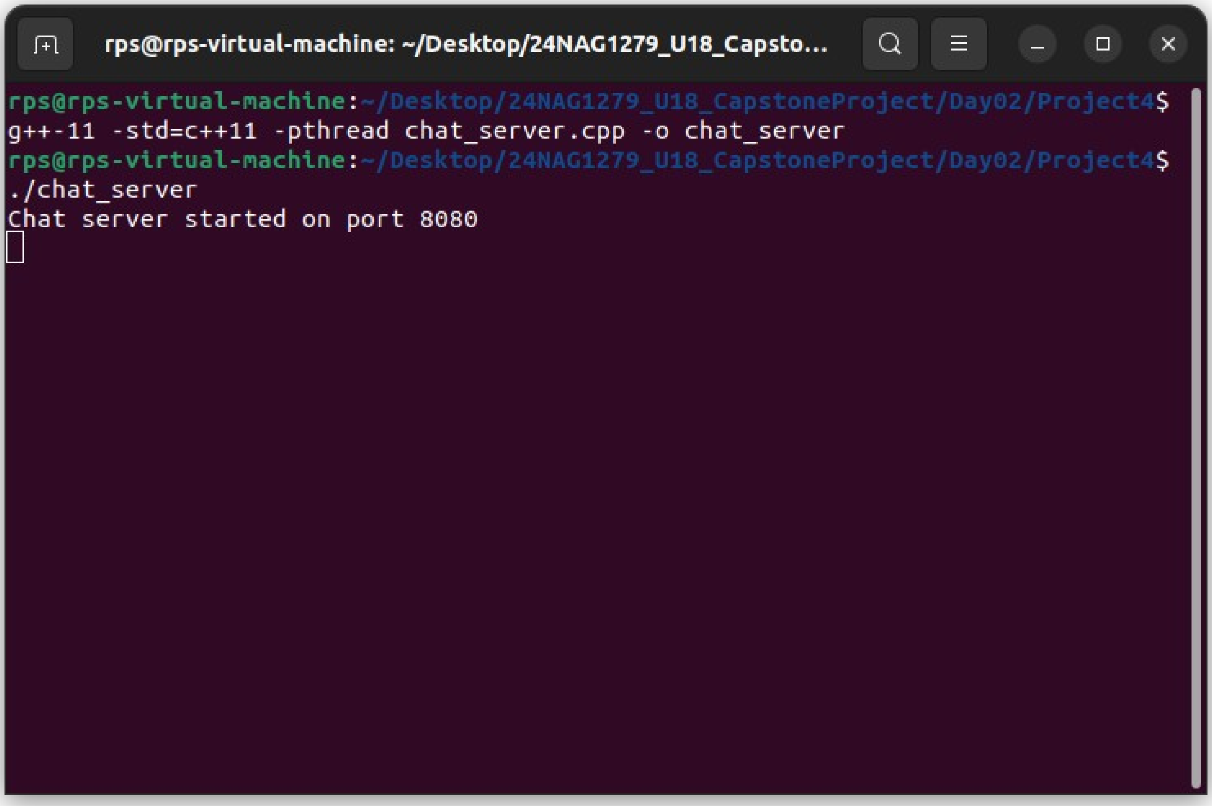
```
    // Remove client from map
    mtx.lock();
    clients.erase(client_socket);
    mtx.unlock();

    close(client_socket);
}

int main() {
    int server_socket, client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);

    // Create server socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        std::cerr << "Error creating server socket" << std::endl;
        return 1;
    }

    // Set server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // Bind server socket
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        std::cerr << "Error binding server socket" << std::endl;
        return 1;
    }

    // Listen for incoming connections
    if (listen(server_socket, 3) < 0) {
        std::cerr << "Error listening for incoming connections" << std::endl;
        return 1;
    }

    std::cout << "Chat server started on port " << PORT << std::endl;

    while (true) {
        // Accept incoming connection
        client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &client_len);
        if (client_socket < 0) {
            std::cerr << "Error accepting incoming connection" << std::endl;
            continue;
        }
        // Handle client in separate thread
        std::thread(handle_client, client_socket).detach();
    }
    return 0;
}
```

//chat_client.cpp

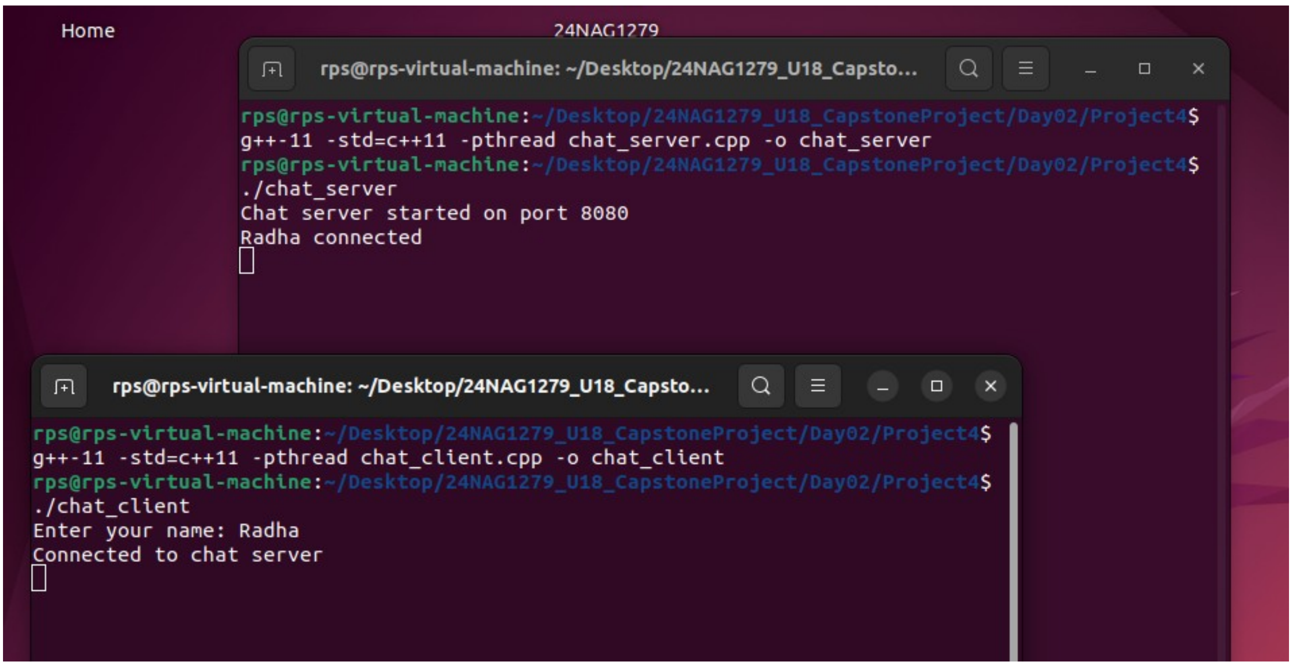
```
#include <iostream>
#include <thread>
#include <mutex>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080

std::mutex mtx;
std::string username;

void receive_messages(int client_socket) {
    char buffer[256];
    int bytes_received;

    while (true) {
        bytes_received = recv(client_socket, buffer, 256, 0);
        if (bytes_received <= 0) {
            break;
        }

        std::string message(buffer, bytes_received);
        std::cout << message << std::endl;
    }
}

int main() {
    int client_socket;
    struct sockaddr_in server_addr;

    // Prompt for username
    std::cout << "Enter your name: ";
    std::cin >> username;

    // Create client socket
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket < 0) {
        std::cerr << "Error creating client socket" << std::endl;
        return 1;
```

```
    }

    // Set server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr);

    // Connect to server
    if (connect(client_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) <
0) {
        std::cerr << "Error connecting to server" << std::endl;
        return 1;
    }

    // Send username to server
    send(client_socket, username.c_str(), username.size(), 0);

    std::cout << "Connected to chat server" << std::endl;

    // Start receiving messages in separate thread
    std::thread(receive_messages, client_socket).detach();

    while (true) {
        std::string message;
        std::getline(std::cin, message);

        // Send message to server
        send(client_socket, message.c_str(), message.size(), 0);
    }

    close(client_socket);
    return 0;
}
```

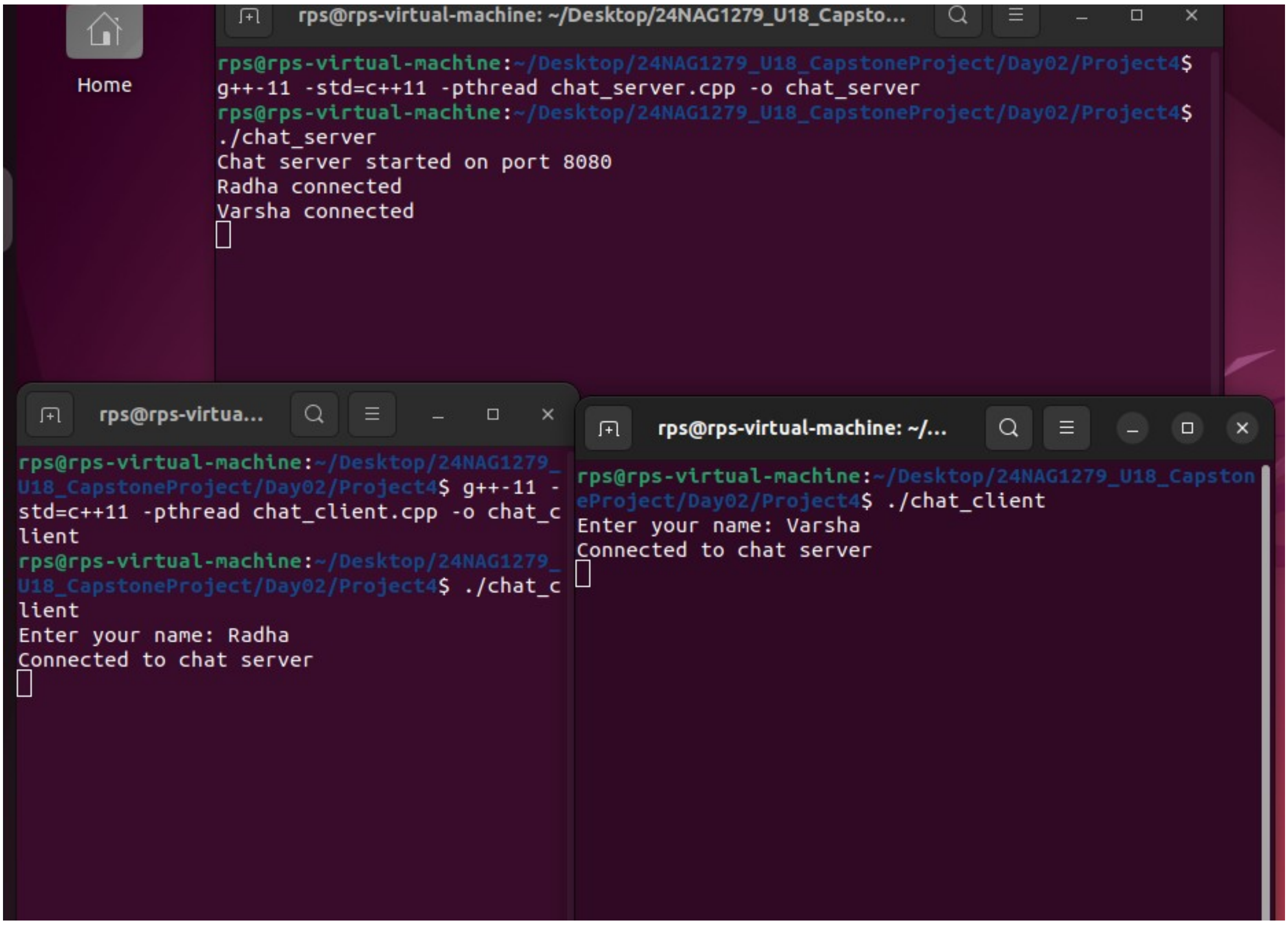
OUTPUT:



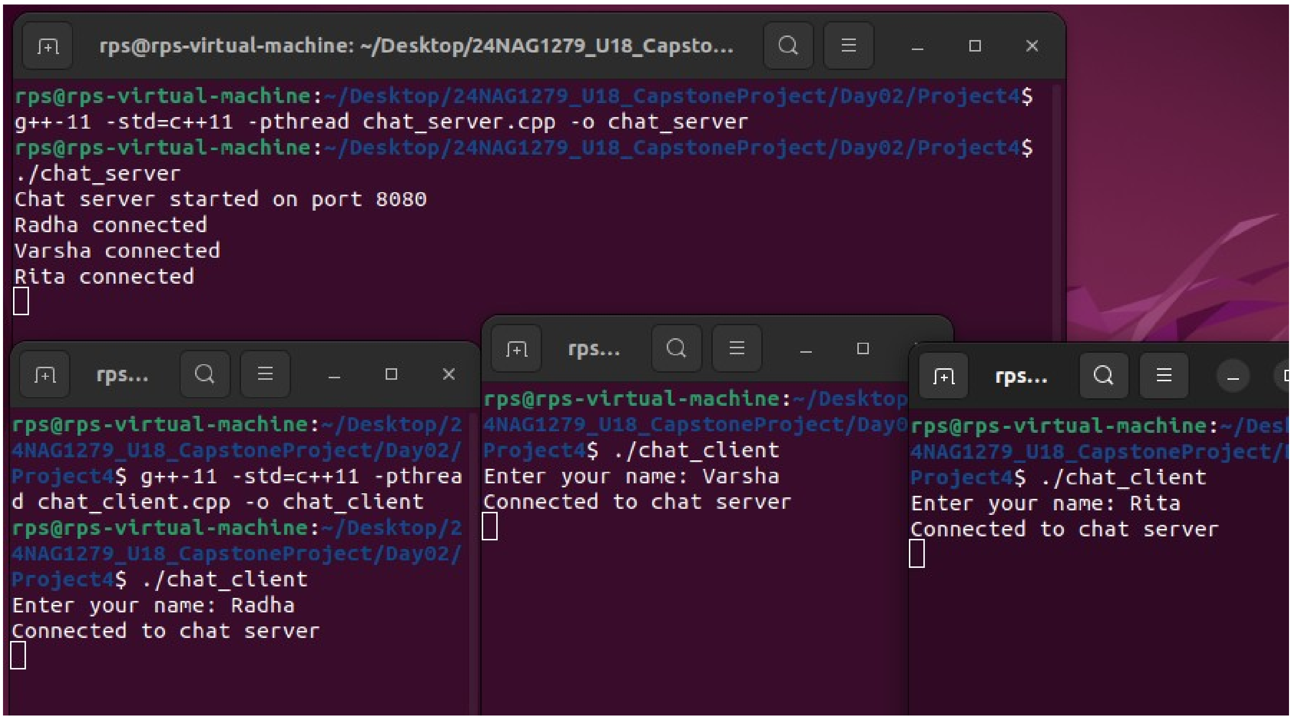
Server started



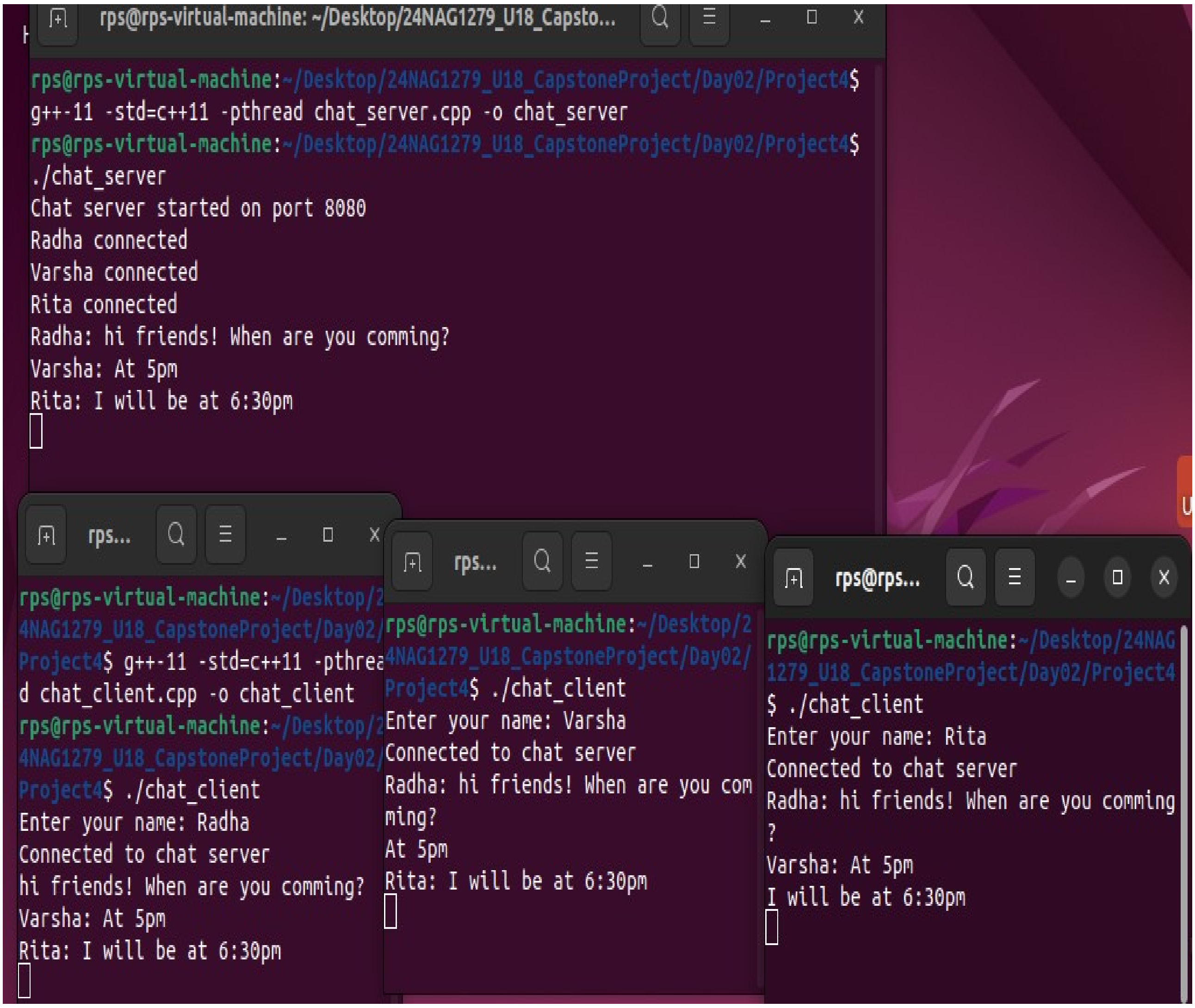
Client1 connected



Client2 connected



Client3 connected



Final output

CONCLUSION

In conclusion, this project successfully developed a distributed real-time messaging system that effectively utilizes C++ and POSIX sockets to enable instant communication between multiple users. By implementing a client-server architecture with multithreading support, the system handles concurrent user interactions and message exchanges efficiently. Despite facing challenges in managing multiple connections and ensuring robust error handling, the project achieved its goal of demonstrating core networking and threading concepts. The system's foundation allows for future enhancements, such as improved scalability, security features, and a more advanced user interface. Overall, the project provided valuable insights into real-time system development and offered a solid platform for further advancements and refinements.