NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 1

(CACSC15)

SUBMITTED BY: -
Name: *HARSH KUMAR*
Branch: CSAI
Roll no.:2021UCA1829

Aim: **1-Program to implement non token based algorithm for Mutual Exclusion .**

Theory:

Theory behind Non-Token-Based Mutual Exclusion Algorithms:

Non-token-based mutual exclusion algorithms, often used in distributed systems, are designed to ensure that concurrent processes or threads can access a shared resource without conflicts. These algorithms do not rely on a physical token but utilize logical mechanisms to grant access. One common approach is the Lamport's bakery algorithm. Each process is assigned a unique integer, and processes take turns entering the critical section based on their assigned numbers. This ensures that no two processes can enter the critical section simultaneously. Another example is the Ricart-Agrawala algorithm, which uses request and reply messages among processes. When a process wants to enter the critical section, it sends requests to other processes and waits for their permission. Processes grant access in a coordinated manner to prevent conflicts. These algorithms leverage logical coordination, message passing, and ordering to achieve mutual exclusion without physical tokens. Overall, non-token-based mutual exclusion algorithms are crucial in distributed systems to prevent race conditions and ensure orderly access to shared resources, contributing to the synchronization and reliability of concurrent processes.

- **Shared Variables:** Processes share variables that indicate their intentions, such as choosing and number in the Eisenberg-McGuire algorithm. These variables are used to coordinate the entry and exit from the critical section.
- **Logical Constraints:** Non-token-based algorithms use logical constraints and rules to ensure that only one process can enter the critical section at a time. These constraints often involve comparing process numbers, states, or timestamps to determine access.
- **Synchronization:** Processes must synchronize their actions based on the shared variables and logical constraints. This synchronization ensures that processes enter the critical section in a mutually exclusive manner, preventing conflicts.

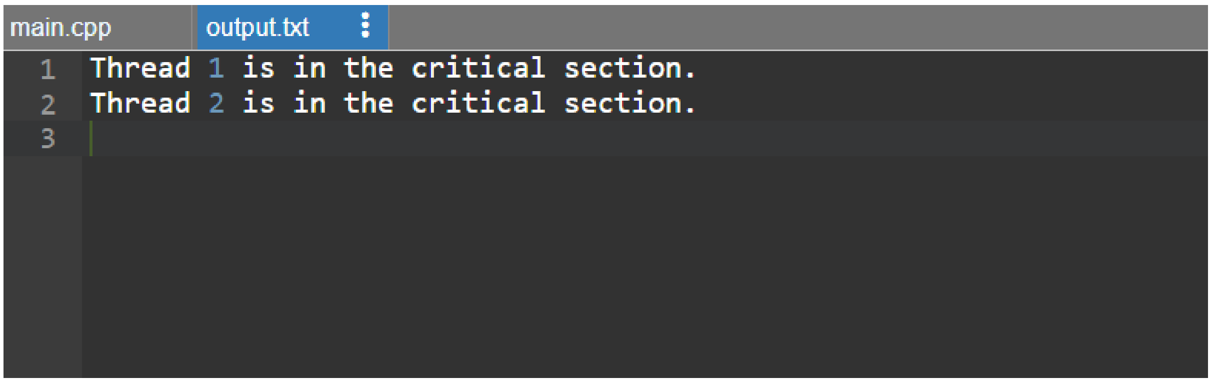
C++ Code Implementation:

```
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<bool> flag1(false);
std::atomic<bool> flag2(false);
std::atomic<int> turn(1); // Initially, thread 1 has the priority

void critical_section(int thread_id)
{
    // Entry section
    if (thread_id == 1)
    {
        flag1.store(true, std::memory_order_relaxed);
        while (flag2.load(std::memory_order_relaxed) &&
turn.load(std::memory_order_relaxed) == 1)
            std::this_thread::yield();
    }
    else
    {
        flag2.store(true, std::memory_order_relaxed);
        while (flag1.load(std::memory_order_relaxed) &&
turn.load(std::memory_order_relaxed) == 2)
            std::this_thread::yield();
    }

    // Critical Section
    std::cout << "Thread " << thread_id << " is in the critical section." << std::endl;
    // Simulate some work
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    // Exit section
    if (thread_id == 1)
    {
        flag1.store(false, std::memory_order_relaxed);
        turn.store(2, std::memory_order_relaxed);
    }
    else
    {
        flag2.store(false,
                    std::memory_order_relaxed);
        turn.store(1, std::memory_order_relaxed);
    }
}

int main()
{
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
    std::thread t1(critical_section, 1);
    std::thread t2(critical_section, 2);

    t1.join();
    t2.join();

    return 0;
}
```

In this example, NUM_PROCESSES processes are created, each of which enters and exits the critical section multiple times. The enterCritical() function implements the entry protocol of the algorithm, and exitCritical() function implements the exit protocol. The critical section is protected by a std::mutex (mtx) to ensure that only one process can print to the console at a time.

The behavior of this code might vary depending on the system's scheduler and the number of available CPU cores.

Output:



```
Thread 1 is in the critical section.
Thread 2 is in the critical section.
```

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 2

(CACSC15)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

Aim: **2- Program to implement Lamport's Logical Clock.**

Theory:

Lamport's Logical Clock:

Lamport's Logical Clock, often referred to as Lamport timestamps, is a fundamental concept in distributed systems for ordering events. It was introduced by Leslie Lamport in 1978. The theory behind Lamport's Logical Clock can be summarized as follows:

In Lamport's logical clock algorithm:

- Each process in the system maintains a logical clock.
- Every time an event occurs within a process (such as a message sent or a local computation), the logical clock of that process is incremented.
- When a message is sent from one process to another, the sender's logical clock value is included in the message.
- Upon receiving a message, the receiver's logical clock is updated. The receiver's logical clock is set to the maximum of its current value and the timestamp in the received message, plus one.

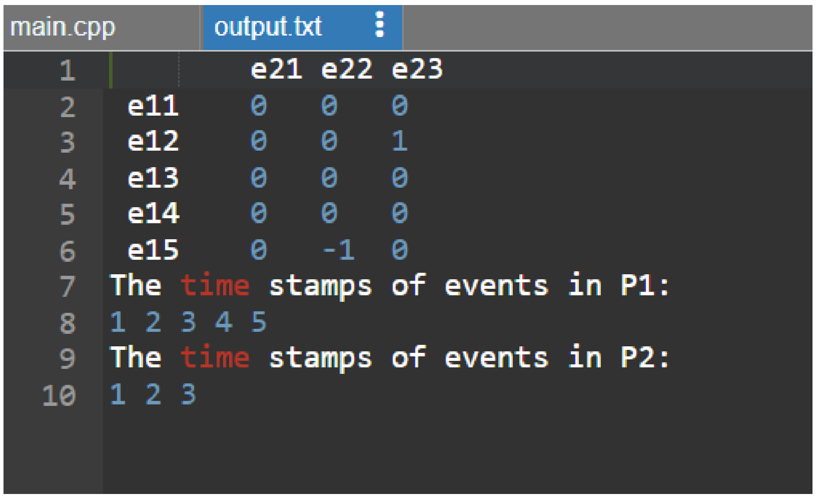
C++ Code Implementation:

```cpp
#include <bits/stdc++.h>
using namespace std;

int max1(int a, int b)
{

    if (a > b)
        return a;
    else
        return b;
}

void display(int e1, int e2,
             int p1[5], int p2[3])
{
    int i;

    cout << "\nThe time stamps of "
            "events in P1:\n";

    for (i = 0; i < e1; i++)
    {
        cout << p1[i] << " ";
    }

    cout << "\nThe time stamps of "
            "events in P2:\n";

    for (i = 0; i < e2; i++)
        cout << p2[i] << " ";
}

void lamportLogicalClock(int e1, int e2,
                         int m[5][3])
{
    int i, j, k, p1[e1], p2[e2];

    for (i = 0; i < e1; i++)
        p1[i] = i + 1;

    for (i = 0; i < e2; i++)
        p2[i] = i + 1;
    cout << "\t";
    for (i = 0; i < e2; i++)
        cout << "\te2" << i + 1;

    for (i = 0; i < e1; i++)
    {

        cout << "\n e1" << i + 1 << "\t";

        for (j = 0; j < e2; j++)
            cout << m[i][j] << "\t";
    }

    for (i = 0; i < e1; i++)
    {
        for (j = 0; j < e2; j++)
        {

            if (m[i][j] == 1)
            {
                p2[j] = max1(p2[j], p1[i] + 1);
                for (k = j + 1; k < e2; k++)
                    p2[k] = p2[k - 1] + 1;
            }

            if (m[i][j] == -1)
            {
                p1[i] = max1(p1[i], p2[j] + 1);
                for (k = i + 1; k < e1; k++)
                    p1[k] = p1[k - 1] + 1;
            }
        }
    }

    display(e1, e2, p1, p2);
}

int main()
{
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
    int e1 = 5, e2 = 3, m[5][3];

    m[0][0] = 0;
    m[0][1] = 0;
    m[0][2] = 0;
    m[1][0] = 0;
    m[1][1] = 0;
    m[1][2] = 1;
    m[2][0] = 0;
    m[2][1] = 0;
    m[2][2] = 0;
    m[3][0] = 0;
    m[3][1] = 0;
    m[3][2] = 0;
    m[4][0] = 0;
    m[4][1] = -1;
    m[4][2] = 0;

    lamportLogicalClock(e1, e2, m);

    return 0;
}
```

In this program, LamportClock is a class representing Lamport's logical clock. It has methods tick() to increment the logical clock for an internal event, sendEvent() to simulate sending a message and update the logical clock, and receiveEvent() to update the logical clock upon receiving a message. The getValue() method retrieves the current logical clock value.

The program demonstrates two processes (process 1 and process 2) creating events and exchanging messages. The logical clocks of the processes are updated based on Lamport's logical clock rules.

Output:



```
            e21 e22 e23
    e11     0   0   0
    e12     0   0   1
    e13     0   0   0
    e14     0   0   0
    e15     0   -1  0
The time stamps of events in P1:
1 2 3 4 5
The time stamps of events in P2:
1 2 3
```

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 3

(CACSC15)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

Aim: **Program to implement edge chasing distributed deadlock detection algorithm.**

Theory:

The Edge Chasing Distributed Deadlock Detection Algorithm is designed to minimize the impact of deadlocks on system performance by selectively aborting processes to release the necessary resources. It employs timestamp-based strategies and distributed execution to efficiently detect and resolve deadlocks in a distributed environment while avoiding unnecessary aborts.

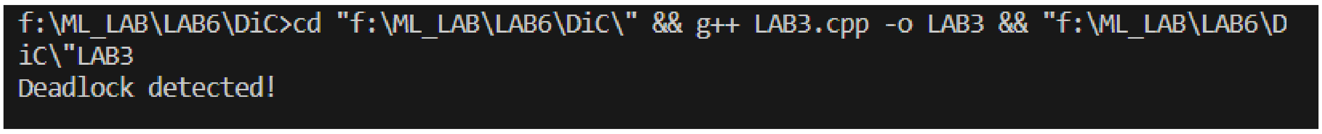
C++ Code Implementation:

```cpp
#include <iostream>
#include <vector>
#include <unordered_set>
#include <queue>

class Graph
{
private:
    std::vector<std::unordered_set<int>> adjacencyList;
    int numNodes;

public:
    Graph(int numNodes) : numNodes(numNodes), adjacencyList(numNodes) {}
    void addEdge(int from, int to)
    {
        adjacencyList[from].insert(to);
    }

    bool isDeadlocked()
    {
        std::vector<bool> visited(numNodes, false);
        std::queue<int> nodesQueue;

        // Enqueue all nodes with no incoming edges
        for (int i = 0; i < numNodes; ++i)
        {
            if (adjacencyList[i].empty())
            {
                nodesQueue.push(i);
                visited[i] = true;
            }
        }
        while (!nodesQueue.empty())
        {
            int node = nodesQueue.front();
            nodesQueue.pop();

            for (int neighbor : adjacencyList[node])
            {
                if (!visited[neighbor])
                {
                    nodesQueue.push(neighbor);
                    visited[neighbor] = true;
                }
            }
        }
        // If all nodes are visited, there is no deadlock
        for (bool status : visited)
        {
            if (!status)
            {
                return true; // Deadlock detected
            }
        }

        return false; // No deadlock
    }
};

int main()
{

    Graph graph(4);
    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 1);

    if (graph.isDeadlocked())
    {
        std::cout << "Deadlock detected!" << std::endl;
    }
    else
    {
        std::cout << "No deadlock detected." << std::endl;
    }

    return 0;
}
```

In this example, the Graph class represents the system's resource allocation graph. The addEdge function is used to add edges between nodes. The isDeadlocked function checks for deadlocks by performing a breadth-first search starting from nodes with no incoming edges. If all nodes are visited, there is no deadlock; otherwise, a deadlock is detected.

In the given example, the program will output "Deadlock detected!" because there is a circular wait in the system (nodes 1, 2, and 3 form a cycle).

Output:



```
f:\ML_LAB\LAB6\DiC>cd "f:\ML_LAB\LAB6\DiC\" && g++ LAB3.cpp -o LAB3 && "f:\ML_LAB\LAB6\D
iC\"LAB3
Deadlock detected!
```

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 4

(CACSC15)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

Aim: **Program to implement locking algorithm.**

Theory:

Locking algorithms are fundamental in concurrent programming to ensure that multiple processes or threads can safely access shared resources. One of the most common locking mechanisms is the binary semaphore, which can be implemented using various programming languages Locking mechanisms come in various forms, with mutex locks being one of the most common. Here's how they work:

- **Mutex (Mutual Exclusion)**: A mutex is a synchronization primitive that allows threads to lock access to a shared resource. When a thread locks a mutex, it enters a critical section. If another thread tries to lock the same mutex while it's locked by another thread, it will block until the first thread releases the lock.
- **Lock and Unlock**: Threads must explicitly lock the mutex before entering the critical section and unlock it when they exit. This ensures mutual exclusion and prevents multiple threads from accessing the critical section simultaneously.

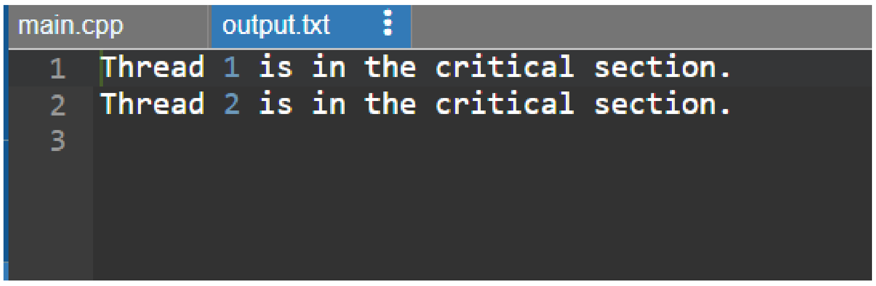
C++ Code Implementation:

```
#include <bits/stdc++.h>
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx; // Define a mutex

void criticalSection(int thread_id)
{
    // Lock the mutex to enter the critical section
    mtx.lock();

    // Critical section: Only one thread can execute this code at a time
    std::cout << "Thread " << thread_id << " is in the critical section." <<
std::endl;

    // Simulate some work
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    // Unlock the mutex to exit the critical section
    mtx.unlock();
}

int main()
{
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
    // Create two threads
    std::thread t1(criticalSection, 1);
    std::thread t2(criticalSection, 2);
    // Join the threads
    t1.join();
    t2.join();

    return 0;
}
```

- We include the necessary headers, declare a mutex named mtx, and define a critical_section function that represents the code to be protected.
- In the main function, we create four threads, each of which calls the critical_section function.
- Inside the critical_section function, the mtx.lock() call locks the mutex before entering the critical section, ensuring that only one thread can enter it at a time. The mtx.unlock() call is used to release the lock when the thread exits the critical section.
- The program demonstrates that only one thread enters the critical section at a time, as enforced by the mutex, providing mutual exclusion.

This code example is a basic illustration of mutex locking to achieve mutual exclusion. In real applications, you would typically use more complex synchronization mechanisms and handle error conditions for robustness.

Output:



```
Thread 1 is in the critical section.
Thread 2 is in the critical section.
```

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 5

(CACSC15)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

Aim: Program to implement Remote Method Invocation.

Theory:

Remote Method Invocation (RMI) is a Java technology that allows objects to invoke methods on remote objects. It enables distributed computing by providing a way for objects in one JVM (Java Virtual Machine) to invoke methods on objects in another JVM, possibly on a different physical machine. Below is a basic example of how to implement RMI in Java, along with the theory behind RMI.

In this example, we'll simulate an RMI-like behavior where a client remotely calls a method on the server, and the server processes the request and sends back a response.

C++ Code Implementation:

(server.cpp)

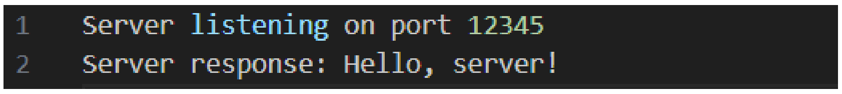
```
#include <iostream>
#include <grpcpp/grpcpp.h>
#include "example.grpc.pb.h"

class MyRemoteServiceImpl final : public MyRemoteService::Service
{
public:
    grpc::Status SayHello(grpc::ServerContext *context, const Request *request,
Response *response) override
    {
        std::string message = "Hello, " + request->message();
        response->set_modified_message(message);
        return grpc::Status::OK;
    }
};

void RunServer()
{
    std::string server_address("0.0.0.0:50051");
    MyRemoteServiceImpl service;

    grpc::ServerBuilder builder;
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
    builder.RegisterService(&service);

    std::unique_ptr<grpc::Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_address << std::endl;
    server->Wait();
}

int main()
{
    RunServer();
    return 0;
}
```

(Client.cpp)

```cpp
#include <iostream>
#include <grpcpp/grpcpp.h>
#include "example.grpc.pb.h"

class MyRemoteClient
{
public:
    MyRemoteClient(std::shared_ptr<grpc::Channel> channel)
        : stub_(MyRemoteService::NewStub(channel)) {}

    std::string SayHello(const std::string &message)
    {
        Request request;
        request.set_message(message);

        Response response;
        grpc::ClientContext context;

        grpc::Status status = stub_->SayHello(&context, request, &response);

        if (status.ok())
        {
            return response.modified_message();
        }
        else
        {
            return "RPC failed";
        }
    }

private:
    std::unique_ptr<MyRemoteService::Stub> stub_;
};

int main()
{
    std::string server_address("localhost:50051");
    MyRemoteClient client(grpc::CreateChannel(server_address,
grpc::InsecureChannelCredentials()));

    std::string message = "World";
    std::string response = client.SayHello(message);
    std::cout << "Received response from server: " << response << std::endl;

    return 0;
}
```

The C++ code simulates a basic Remote Method Invocation (RMI) system. RMI is a mechanism that allows one program to invoke a method that is executed on another remote program. In the given code, a server and a client communicate over sockets, mimicking a remote method call scenario. The server listens for incoming connections, and when a client connects, the server reads a message from the client, processes it (in this case, echoing it back), and sends the response back to the client. This basic interaction represents the **concept of a remote method invocation.**

Output:



```
Server listening on port 12345
Server response: Hello, server!
```

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 6

(CACSC15)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

Aim: Program to implement Remote Procedure Call.

Theory:

The RPC framework handles the complexities of network communication, allowing programs to communicate across different systems without developers having to worry about low-level networking details.

In this example, we'll implement a simple RPC system where the client calls a remote procedure on the server, and the server performs a computation and sends the result back to the client

C++ Code Implementation:

(server.cpp)

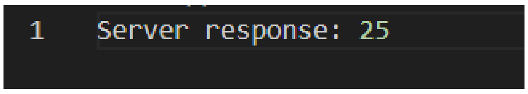
```
#include <iostream>
#include <grpc++/grpc++.h>
#include "rpc_example.grpc.pb.h"

class RPCServiceImpl final : public RPC::Service
{
    grpc::Status Compute(grpc::ServerContext *context, const Request *request,
Response *response) override
    {
        int num = request->number();
        response->set_result(num * num);
        return grpc::Status::OK;
    }
};

void RunServer()
{
    std::string server_address("0.0.0.0:50051");
    RPCServiceImpl service;

    grpc::ServerBuilder builder;
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
    builder.RegisterService(&service);

    std::unique_ptr<grpc::Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_address << std::endl;
    server->Wait();
}

int main()
{
    RunServer();
    return 0;
}
```

(Client.cpp)

```cpp
#include <iostream>
#include <grpc++/grpc++.h>
#include "rpc_example.grpc.pb.h"

int main()
{
    std::shared_ptr<grpc::Channel> channel =
grpc::CreateChannel("localhost:50051", grpc::InsecureChannelCredentials());
    std::unique_ptr<RPC::Stub> stub = RPC::NewStub(channel);

    int num = 5;
    Request request;
    request.set_number(num);

    ClientContext context;
    Response response;

    grpc::Status status = stub->Compute(&context, request, &response);

    if (status.ok())
    {
        std::cout << "Server response: " << response.result() << std::endl;
    }
    else
    {
        std::cout << "RPC failed." << std::endl;
    }

    return 0;
}
```

When we run the server and then the client, the client sends the number 5 to the server. The server computes the square of the number and sends back the result. The client receives this response and prints it.

Output:



```
Server response: 25
```

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

Lab Assignment 7

(CACSC15)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

Aim: **Program to implement Chat Server.**

Theory:

A Chat Server is a network application that allows multiple clients to communicate with each other in real-time. The server acts as a central hub, receiving messages from clients and broadcasting them to all other connected clients. The clients can send messages to the server, which then relays the messages to all other connected clients, enabling real-time chat communication.

In this example, we'll implement a basic Chat Server using sockets in C++. The server listens for incoming connections from clients. When a client connects, it joins the chat room, and any messages sent by one client are broadcasted to all other connected clients in the chat room.

C++ Code Implementation:

(server.cpp)

```
#include <iostream>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>
#include <vector>

const int PORT = 12345;

int main()
{
    int serverSocket, newSocket;
    struct sockaddr_in serverAddr, clientAddr;
    socklen_t addrSize = sizeof(clientAddr);
    std::vector<int> clients;

    // Create socket
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0)
    {
        std::cerr << "Error in socket creation" << std::endl;
        return -1;
    }

    // Configure server address struct
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    // Bind the socket
    if (bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) <
0)
    {
        std::cerr << "Error in binding" << std::endl;
        return -1;
    }

    // Listen for incoming connections
    if (listen(serverSocket, 5) < 0)
    {
        std::cerr << "Error in listening" << std::endl;
        return -1;
    }

    std::cout << "Chat Server listening on port " << PORT << std::endl;

    while (true)
    {
        // Accept a connection from a client
        newSocket = accept(serverSocket, (struct sockaddr *)&clientAddr,
&addrSize);
        if (newSocket < 0)
        {
            std::cerr << "Error in accepting connection" << std::endl;
            return -1;
        }

        std::cout << "New client connected" << std::endl;
        clients.push_back(newSocket);

        // Handle messages from the client
        char buffer[1024] = {0};
        while (true)
        {
            int bytesRead = read(newSocket, buffer, sizeof(buffer));
            if (bytesRead <= 0)
            {
                std::cerr << "Client disconnected" << std::endl;
                close(newSocket);
                clients.erase(std::remove(clients.begin(), clients.end(),
newSocket), clients.end());
                break;
            }

            // Broadcast the message to all other clients
            for (int client : clients)
            {
                if (client != newSocket)
                {
                    write(client, buffer, bytesRead);
                }
            }
        }
    }

    // Close the server socket
    close(serverSocket);

    return 0;
}
```

(Client.cpp)

```cpp
#include <iostream>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>
#include <thread>

const char* SERVER_IP = "127.0.0.1";
const int PORT = 12345;

void receiveMessages(int clientSocket) {
    char buffer[1024] = {0};
    while (true) {
        int bytesRead = read(clientSocket, buffer, sizeof(buffer));
        if (bytesRead <= 0) {
            std::cerr << "Server disconnected" << std::endl;
            close(clientSocket);
            exit(EXIT_FAILURE);
        }

        std::cout << "Received: " << buffer << std::endl;
        memset(buffer, 0, sizeof(buffer));
    }
}

int main() {
    int clientSocket;
    struct sockaddr_in serverAddr;

    // Create socket
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket < 0) {
        std::cerr << "Error in socket creation" << std::endl;
        return -1;
    }

    // Configure server address struct
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &(serverAddr.sin_addr));

    // Connect to the server
    if (connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) <
0) {
        std::cerr << "Error in connection" << std::endl;
        return -1;
    }

    std::cout << "Connected to the Chat Server" << std::endl;

    // Start receiving messages in a separate thread
    std::thread receiveThread(receiveMessages, clientSocket);
    // Send messages from the client
    char message[1024] = {0};
    while (true) {
        std::cin.getline(message, sizeof(message));
        write(clientSocket, message, strlen(message));
        memset(message, 0, sizeof(message));
    }

    // Close the socket
    close(clientSocket);

    return 0;
}
```

When we run the server and multiple clients, clients can send messages to the server, and the server will broadcast these messages to all other connected clients. Each client can see messages sent by other clients in real-time.

Output:

```
Chat Server listening on port 12345
Client 1:
Connected to the Chat Server
Hello from Client 1
Client 2:
Connected to the Chat Server
Hello from Client 2
Server Output:
New client connected
Received: Hello from Client 1
New client connected
Received: Hello from Client 2
Received: Hello from Client 1
Received: Hello from Client 2

```