**TITLE:**

Write a program to implement Echo server using socket programming

**AIM:**

To implement Client-Server architecture as echo server using

1. Socket programming
2. Multi-threading

**OBJECTIVE:**

To understand the concept of socket programming, multi-threading, and echo servers.

**THEORY:**

Most inter process communication uses the client server model. One of the two processes, the client, typically to make a request for information. The system calls for establishing a connection for the client and server are as follows:

Client side:

1. Socket ()

2. Connect ()

3. Read ()

4. Write ()

Server Side:

1. Socket ()
2. Bind ()
3. Listen ()
4. Accept ()
5. Read ()
6. Write ()

**SOCKET TYPES:**

When a socket is created, the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of same type and in the same domain. There are two widely used domains:

1. UNIX domain
2. internet domain

There are two widely used socket types.

1. Stream sockets
2. Datagram sockets

Stream sockets treat communication as continuous stream of characters, while datagram sockets have to read the entire message at once. Each uses its own communication protocol. Stream sockets use TCP, which is reliable, stream oriented protocol and datagram sockets are UDP, which is unreliable and message oriented.

**SYSTEM CALL FORMATS:**

1. int Socket (int family, int type, int protocol)

Type: f Socket type

SOCK\_DGRAM------- UDP

SOCK\_STREAM ----- TCP

Protocol: Type = 0-------------- TCP

Type = 1-------------- UDP

The above call returns socket descriptor.

2. int Bind (int SOCK\_FD,struct sockaddr \*myadddr, int addrlen)

use:

Binding socket with the address.

3. Listen (int SOCK\_FD, int backlog)

Where,

Backlog = number of requests that can be queued up to the server .

4. Accept (int SOCK\_FD, struct sockaddr \*peer, int \*addrlen)

Where,

int \*adddrlen = length of the structure

5. Connect (int SOCK\_FD, struct sock addr \*servaddr, socklen\_t addrlen)

RANGE OF PORTS:

1. Wellknown ports: 0-1023

2. Registered: 1024 – 59151

(Controlled by IANA)

3. Dynamic (Ephemeral): 49152-65535

Reserved port in UNIX is any port < 1024

**MULTITHREADED ECHO SERVER:**

Multiple clients request for service to the same server. Server creates threads to serve the clients. Threads are light weight processes. It provides concurrency. On the server side, process keeps listening to the requests made by clients. As soon as the connection has to be made with clients, server creates the threads.

**CODE:**

* + - 1. **Client**

#include <stdio.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

*int* main()

{

*int* sockfd;

*struct* sockaddr\_in server\_address;

*char* name[80], message[200];

    printf("Enter your name: ");

    fgets(name, sizeof(name), stdin);

    // Remove the newline character

    name[strcspn(name, "\n")] = '\0';

    sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

    // Error handling for socket creation

    if (sockfd == -1)

    {

        perror("Socket creation failed");

        exit(EXIT\_FAILURE);

    }

    server\_address.sin\_family = AF\_INET;

    // change the inet\_addr to server's ip and change the port to server's port

    server\_address.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    server\_address.sin\_port = htons(9129);

    // error handling for server connection

    if (connect(sockfd, (*struct* sockaddr \*)&server\_address, sizeof(server\_address)) == -1)

    {

        perror("Connection failed");

        close(sockfd);

        exit(EXIT\_FAILURE);

    }

    // if connected, sends client name to server

    write(sockfd, name, strlen(name));

    while (1)

    {

        // keeps asking the client for input and sends it to server

        printf("Enter your message (type 'bye' to exit): ");

        fgets(message, sizeof(message), stdin);

        message[strcspn(message, "\n")] = '\0';

        write(sockfd, message, strlen(message));

        printf("%s Sending: %s\n", name, message);

        // if the sent message was bye, terminate the client

        if (strcmp(message, "bye") == 0)

        {

            printf("Chat terminated.\n");

            break;

        }

    }

    close(sockfd);

    exit(EXIT\_SUCCESS);

}

* + - 1. **Server**

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <pthread.h>

#include <stdio.h>

// max client threads allowd

#define MAX\_CLIENTS 5

// global struct for client id and client name

*struct* client\_info

{

*int* sockfd;

*char* name[80];

};

*void* \*handle\_client(*void* \**arg*)

{

    // client handling thread

    // stays alive till a client is connected

    // ends when client is terminated

*struct* client\_info \*client = (*struct* client\_info \*)*arg*;

*char* message[200];

    // int flag=0;

    // read client name once a client connects

*ssize\_t* name\_len = read(client->sockfd, client->name, sizeof(client->name));

    // error handling for name

    if (name\_len <= 0)

    {

        // perror("Read name error");

        close(client->sockfd);

        free(client);

        pthread\_exit(NULL);

    }

    client->name[name\_len] = '\0';

    printf("%s has connected\n", client->name);

    while (1)

    {

        // keep reading and printing all client messages

*ssize\_t* message\_len = read(client->sockfd, message, sizeof(message));

        if (message\_len <= 0)

        {

            // perror("Read message error");

            break;

        }

        message[message\_len] = '\0';

        // if the read message was bye, print that client has left and stop listening

        if (strcmp(message, "bye") == 0)

        { // Remove '\n' from "bye"

            printf("%s has left the chat.\n", client->name);

            break;

        }

        printf("%s says: %s\n", client->name, message);

    }

    // close client connection and end the thread

    close(client->sockfd);

    free(client);

    pthread\_exit(NULL);

}

*int* main()

{

*int* server\_sockfd;

*struct* sockaddr\_in server\_address;

*pthread\_t* threads[MAX\_CLIENTS];

    server\_sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

    // error handling for socket creation

    if (server\_sockfd == -1)

    {

        perror("Socket creation failed");

        exit(EXIT\_FAILURE);

    }

    server\_address.sin\_family = AF\_INET;

    // change the inet\_addr to desired ip and change the port to desired port

    server\_address.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    server\_address.sin\_port = htons(9129);

    // error handling for binding ip address

    if (bind(server\_sockfd, (*struct* sockaddr \*)&server\_address, sizeof(server\_address)) == -1)

    {

        perror("Bind failed");

        close(server\_sockfd);

        exit(EXIT\_FAILURE);

    }

    // server has started and is waiting for clients

    printf("Server waiting for connections...\n");

    listen(server\_sockfd, MAX\_CLIENTS);

    while (1)

    {

        // keeps listening for clients

        // spawns a thread when new client connects

        // closes thread when client leaves

*struct* sockaddr\_in client\_address;

*socklen\_t* client\_len = sizeof(client\_address);

*int* client\_sockfd = accept(server\_sockfd, (*struct* sockaddr \*)&client\_address, &client\_len);

        // error handling for client connection to server

        if (client\_sockfd == -1)

        {

            perror("Accept failed");

            continue;

        }

        // new client has connected

*struct* client\_info \*client = (*struct* client\_info \*)malloc(sizeof(*struct* client\_info));

        // error handling for new client

        if (client == NULL)

        {

            perror("Memory allocation failed");

            close(client\_sockfd);

            continue;

        }

        // set new client id and spawn thread

        client->sockfd = client\_sockfd;

        pthread\_create(&threads[MAX\_CLIENTS], NULL, handle\_client, (*void* \*)client);

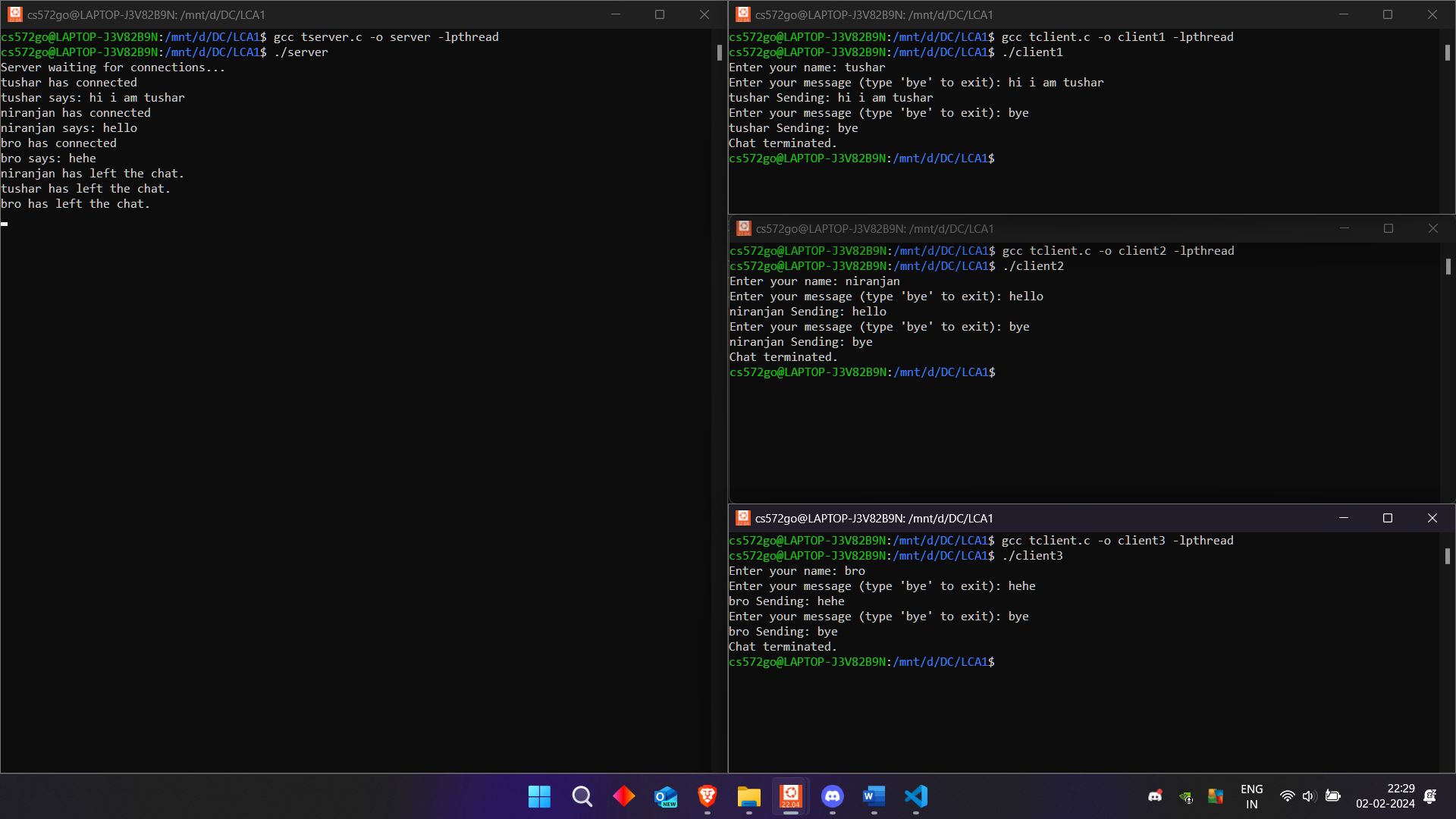
    }

    close(server\_sockfd);

    exit(EXIT\_SUCCESS);

}

**OUTPUT:**



**CONCLUSION:**

Implemented a Chat server using sockets in C.

**PLATFORM:**

Windows:

VSCode

Windows Subsystem for Linux (Ubuntu 22.04 LTS)

**LANGUAGE:**

C language.

FAQs

1. Give the differences between UDP and TCP protocols

Answer:

|  |  |  |
| --- | --- | --- |
| **Aspect** | **UDP** | **TCP** |
| **Connection** | Connectionless | Connection-oriented |
| **Reliability** | Unreliable | Reliable |
| **Order of Delivery** | No guarantee | In-order delivery assured |
| **Header Overhead** | Low | High |
| **Flow Control** | No flow control | Flow control mechanisms |
| **Error Checking** | Limited error checking | Extensive error checking |

1. How does the accept system call work in socket programming

Answer:

In socket programming, the accept system call is used by a server to handle incoming client connections. When the server is in a listening state, accept waits for a client to establish a connection. Once a connection request arrives, accept creates a new socket specifically for communication with that client.

The new socket is unique to the connected client, allowing the server to interact with multiple clients simultaneously. The original listening socket remains open and continues to accept new connections, so the server can serve multiple clients simultaneously.

1. What is the advantage of using threads in socket programming

Answer:

Advantages of using threads in socket programming are as follows-

1. Concurrency: Threads allow multiple clients to be served simultaneously without blocking the main application.
2. Responsiveness: It can quickly create a new thread to handle each incoming connection, ensuring that clients receive timely responses.
3. Resource Efficiency: Threads share the same process memory space, reducing overhead compared to creating separate processes for each client.
4. Scalability: Additional threads can be created as needed to accommodate increasing client connections, providing scalability and flexibility.