**IPC: Sockets**

**Subject - Unix Operating System**

**Name – Hemant Sharma**

**PRN – 22610001 Class – TYIT**

**Assignment No – 9(f)**

**Title-** Implement chatting using TCP/UDP socket (between two or more users.)

**Objectives:**

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To knows the iterative and concurrent server concept.

**Theory:**

A very basic one-way Client and Server setup where a client connects, sends messages to server and the server shows them using socket connection. Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers

CLIENT-SIDE PROGRAMMING:

Establish a Socket Connection

* To connect to other machine, we need a socket connection.
* A socket connection means the two machines have information about each other’s network location (IP Address) and TCP port. The java.net.Socket class represents a Socket.
* To open a socket: Socket socket = new Socket (“127.0.0.1”, 5000)

• First argument – IP address of Server. (127.0.0.1 is the IP address of localhost, where code will run on single stand-alone machine).

• Second argument – TCP Port. (Just a number representing which

application to run on a server. For example, HTTP runs on port 80.

Port number can be from 0 to 65535) To communicate over a socket

connection, streams are used to both input and output the data. Closing

the connection. The socket connection is closed explicitly once the

message to server is sent.

SERVER-SIDE PROGRAMMING:

Establish a Socket Connection

To write a server application two sockets are needed.

* A ServerSocket which waits for the client requests (when a client makes a new Socket())
* A plain old Socket socket to use for communication with the client getOutputStream() method is used to send the output through the socket. Close the Connection After finishing, it is important to close the connection by closing the socket as well as input/output streams

**Program:**

**Server:**#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <pthread.h>

#define PORT 12345

#define MAX\_CLIENTS 10

#define MAX\_MSG\_SIZE 1024

int client\_sockets[MAX\_CLIENTS];

pthread\_mutex\_t client\_sockets\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void \*client\_handler(void \*arg) {

    int client\_socket = \*(int \*)arg;

    char message[MAX\_MSG\_SIZE];

    int recv\_len;

    struct sockaddr\_in client\_addr;

    socklen\_t addr\_len = sizeof(client\_addr);

    // Get client details (IP and port)

    getpeername(client\_socket, (struct sockaddr \*)&client\_addr, &addr\_len);

    char client\_ip[INET\_ADDRSTRLEN];

    inet\_ntop(AF\_INET, &client\_addr.sin\_addr, client\_ip, sizeof(client\_ip));

    int client\_port = ntohs(client\_addr.sin\_port);

    printf("Client %s:%d connected\n", client\_ip, client\_port);

    while (1) {

        // Receive message from client

        recv\_len = recv(client\_socket, message, sizeof(message), 0);

        if (recv\_len <= 0) {

            printf("Client %s:%d disconnected\n", client\_ip, client\_port);

            break;

        }

        message[recv\_len] = '\0';

        // Print message with client identifier

        printf("Received message from %s:%d: %s\n", client\_ip, client\_port, message);

        // Broadcast message to all clients except the sender

        pthread\_mutex\_lock(&client\_sockets\_mutex);

        for (int i = 0; i < MAX\_CLIENTS; i++) {

            if (client\_sockets[i] != 0 && client\_sockets[i] != client\_socket) {

                send(client\_sockets[i], message, strlen(message), 0);

                printf("Sent message to client %s:%d: %s\n", client\_ip, client\_port, message);

            }

        }

        pthread\_mutex\_unlock(&client\_sockets\_mutex);

    }

    // Close client socket

    close(client\_socket);

    return NULL;

}

int main() {

    int server\_socket, client\_socket;

    struct sockaddr\_in server\_addr, client\_addr;

    socklen\_t client\_len = sizeof(client\_addr);

    pthread\_t thread\_id;

    // Initialize client sockets to 0

    memset(client\_sockets, 0, sizeof(client\_sockets));

    // Create TCP socket

    if ((server\_socket = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {

        perror("Socket creation failed");

        exit(1);

    }

    memset(&server\_addr, 0, sizeof(server\_addr));

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

    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

    server\_addr.sin\_port = htons(PORT);

    // Bind socket

    if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) == -1) {

        perror("Bind failed");

        close(server\_socket);

        exit(1);

    }

    // Listen for incoming connections

    if (listen(server\_socket, 5) == -1) {

        perror("Listen failed");

        close(server\_socket);

        exit(1);

    }

    printf("Chat Server is waiting for clients to connect...\n");

    while (1) {

        client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_len);

        if (client\_socket == -1) {

            perror("Accept failed");

            continue;

        }

        // Add client socket to list

        pthread\_mutex\_lock(&client\_sockets\_mutex);

        for (int i = 0; i < MAX\_CLIENTS; i++) {

            if (client\_sockets[i] == 0) {

                client\_sockets[i] = client\_socket;

                break;

            }

        }

        pthread\_mutex\_unlock(&client\_sockets\_mutex);

        // Create a new thread for the client

        if (pthread\_create(&thread\_id, NULL, client\_handler, (void \*)&client\_socket) != 0) {

            perror("Thread creation failed");

            close(client\_socket);

        }

    }

    close(server\_socket);

    return 0;

}

**Client:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define SERVER\_IP "127.0.0.1"

#define SERVER\_PORT 12345

#define MAX\_MSG\_SIZE 1024

void \*receive\_messages(void \*arg) {

    int client\_socket = \*(int \*)arg;

    char message[MAX\_MSG\_SIZE];

    while (1) {

        int recv\_len = recv(client\_socket, message, sizeof(message), 0);

        if (recv\_len <= 0) {

            printf("Connection closed by server\n");

            break;

        }

        message[recv\_len] = '\0';

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

    }

    return NULL;

}

int main() {

    int client\_socket;

    struct sockaddr\_in server\_addr;

    pthread\_t recv\_thread;

    // Create TCP socket

    if ((client\_socket = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {

        perror("Socket creation failed");

        exit(1);

    }

    memset(&server\_addr, 0, sizeof(server\_addr));

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

    server\_addr.sin\_port = htons(SERVER\_PORT);

    if (inet\_pton(AF\_INET, SERVER\_IP, &server\_addr.sin\_addr) <= 0) {

        perror("Invalid address");

        exit(1);

    }

    // Connect to the server

    if (connect(client\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) == -1) {

        perror("Connection failed");

        exit(1);

    }

    // Start a thread to receive messages from the server

    pthread\_create(&recv\_thread, NULL, receive\_messages, (void \*)&client\_socket);

    // Send messages to the server

    char message[MAX\_MSG\_SIZE];

    while (1) {

        printf("Enter message: ");

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

        // Remove the trailing newline character

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

        if (strcmp(message, "exit") == 0) {

            break;

        }

        send(client\_socket, message, strlen(message), 0);

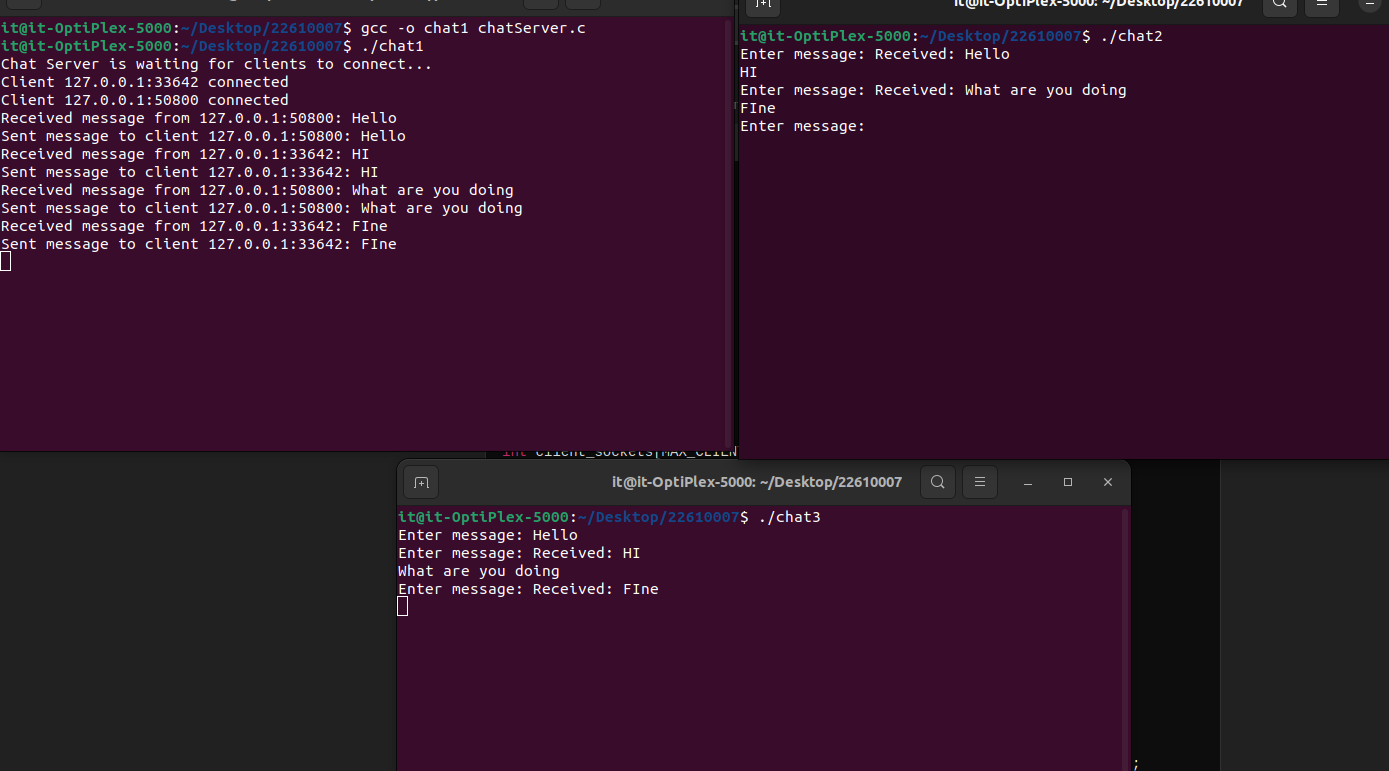
    }

    close(client\_socket);

    return 0;

}

**Output:**

****

**Conclusion:**

Implementing a chat system with TCP or UDP allows users to exchange messages in real-time. TCP ensures reliable, ordered communication, making it suitable for multi-user chat applications, while UDP provides a faster, connectionless alternative, though with less reliability. TCP chat servers often use threads to manage multiple clients concurrently, while UDP servers handle each message individually. In both cases, clients can send and receive messages, and the server facilitates communication, broadcasting messages to all participants.