**IPC: Sockets**

**Subject - Unix Operating System**

**Name – Hemant Sharma**

**PRN – 22610001 Class – TYIT**

**Assignment No – 9(e)**

**Title-** Implement echo server using TCP/UDP in iterative/concurrent logic.

**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>

#define PORT 12345

#define MAX\_MSG\_SIZE 1024

int main() {

    int server\_socket, client\_socket;

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

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

    char buffer[MAX\_MSG\_SIZE];

    int recv\_len;

    // Create a 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 the 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("TCP Echo Server (Iterative) is waiting for client connections on port %d...\n", PORT);

    while (1) {

        // Accept a client connection

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

        if (client\_socket == -1) {

            perror("Accept failed");

            continue;

        }

        printf("Client connected\n");

        while (1) {

            // Receive message from the client

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

            if (recv\_len == -1) {

                perror("Receive failed");

                break;

            }

            // Check if the client has closed the connection

            if (recv\_len == 0) {

                printf("Client disconnected\n");

                break;

            }

            // Null-terminate the received message

            buffer[recv\_len] = '\0';

            // Echo the message back to the client

            send(client\_socket, buffer, recv\_len, 0);

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

        }

        // Close the client connection

        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

int main() {

    int client\_socket;

    struct sockaddr\_in server\_addr;

    char message[MAX\_MSG\_SIZE], server\_response[MAX\_MSG\_SIZE];

    // Create a 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);

    }

    // Two-way communication loop

    while (1) {

        printf("Enter message to send to server (type 'exit' to quit): ");

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

        // Remove trailing newline character from the message

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

        // Exit the loop if the user types "exit"

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

            break;

        }

        // Send message to the server

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

        // Receive echoed message from the server

        int recv\_len = recv(client\_socket, server\_response, sizeof(server\_response) - 1, 0);

        server\_response[recv\_len] = '\0'; // Null-terminate the string

        printf("Received from server: %s\n", server\_response);

    }

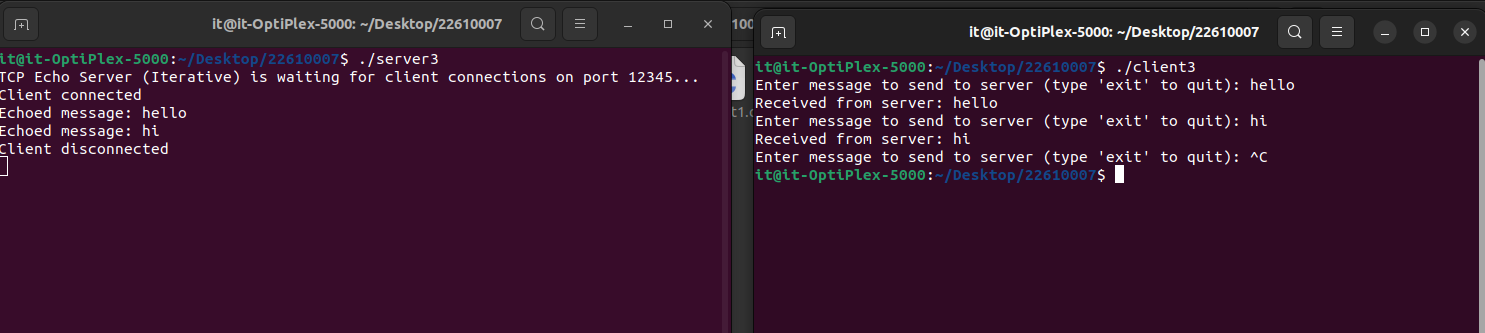
    // Close the socket

    close(client\_socket);

    return 0;

}

**Output:**

****

**Conclusion:**In an Echo Server, TCP can operate iteratively by handling one client at a time, which is simple but slower for many clients. With concurrent logic, it uses threads to handle multiple clients simultaneously, improving scalability. UDP, being connectionless, handles packets independently; iteratively, it processes each packet one by one, while concurrently, it can use threads to handle multiple packets at once, enhancing performance for high-demand scenarios. In essence, iterative logic is simpler but less efficient for many clients, whereas concurrent logic improves server performance by processing multiple clients or packets simultaneously.