EC3022D Computer Networks Winter 2023-24

Socket Programming Assignment

SUBMITTED BY

ABHINAV RAJ V

B210759EC

EC01

INTRODUCTION

The C socket program establishes a communication protocol between a client and a server, facilitating message exchange over a network. In this implementation, when the client sends a line to the server, the server echoes it back promptly. This bidirectional echoing requires both the client and server to alternate between receiving and sending data using the recv() and send() functions.

The code has both a server-side implementation of a modified simplex-talk protocol in a client-server communication model. The server is designed to listen for incoming connections on a specified port (5432 in this case), accept a new connection from a client, receive a line of text from the client, and echo the same line back to the client. The client is designed to connect to a server on a specified port (5432 in this case), send a line of text to the server, and receive the same line back from the server.

CODE

Server

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define SERVER_PORT 5432
#define MAX_PENDING 5
#define MAX_LINE 256

int main()
{
    struct sockaddr_in sin; // Server address structure
    char buf[MAX_LINE]; // Buffer for storing incoming messages
    int len; // Length of received message
    int s, new_s; // Socket descriptors

    // Initialize server address structure
    memset((char *)&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(SERVER_PORT);

    // Create a new socket
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("simplex-talk: socket");
        exit(1);
    }

    // Bind the socket to the server address and port
    if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
        perror("simplex-talk: bind");
        exit(1);
    }

    // Listen for incoming connections
    listen(s, MAX_PENDING);

    printf("Server started. Listening for connections...\n");

    // Main server loop
    while(1) {
        // Accept a new connection
```

```
        if ((new_s = accept(s, (struct sockaddr *)&sin, &len)) < 0) {
            perror("simplex-talk: accept");
            exit(1);
        }
        printf("Connection accepted from client.\n");

        // Receive messages from the client
        while (len = recv(new_s, buf, sizeof(buf), 0)) {
            buf[len] = '\0';
            printf("Received message from client: %s\n", buf);

            // Echo the received message back to the client
            send(new_s, buf, len, 0);
            printf("Sent message to client: %s\n", buf);
        }

        // Close the connection
        printf("Connection with client closed.\n");
        close(new_s);
    }
}
```

Client

```c
#include <stdio.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define SERVER_PORT 5432
#define MAX_LINE 256

int main(int argc, char * argv[])
{
    FILE *fp;
    struct hostent *hp; // Server info
    struct sockaddr_in sin; // Server address
    char *host; // Host name
    char buf[MAX_LINE]; // Buffer for messages
    int s; // Socket descriptor
    int len; // Message length

    // Check command line arguments
    if (argc==2) {
        host = argv[1];
    }
    else {
        fprintf(stderr, "usage: simplex-talk host\n");
        exit(1);
    }

    // Get server info
    hp = gethostbyname(host);
    if (!hp) {
        fprintf(stderr, "simplex-talk: unknown host: %s\n", host);
        exit(1);
    }

    // Set up server address
    memset((char *)&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    memcpy((char *)&sin.sin_addr, hp->h_addr, hp->h_length);
    sin.sin_port = htons(SERVER_PORT);

    // Create socket
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
```

```
        perror("simplex-talk: socket");
        exit(1);
    }

    // Connect to server
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("simplex-talk: connect");
        close(s);
        exit(1);
    }
    printf("Connected to server.\n");

    // Main client loop
    while (fgets(buf, sizeof(buf), stdin)) {
        buf[MAX_LINE-1] = '\0';
        len = strlen(buf) + 1;
        // Send message to server
        send(s, buf, len, 0);
        printf("Sent to server: %s\n", buf);

        // Receive response from server
        char response[MAX_LINE];
        int response_len = recv(s, response, sizeof(response) - 1, 0);
        if (response_len >= 0) {
            response[response_len] = '\0'; // Null-terminate the response
            printf("Received from server: %s\n", response);
        }
    }
    close(s); // Close the connection
    printf("Disconnected from server.\n");
}
```

Explanation

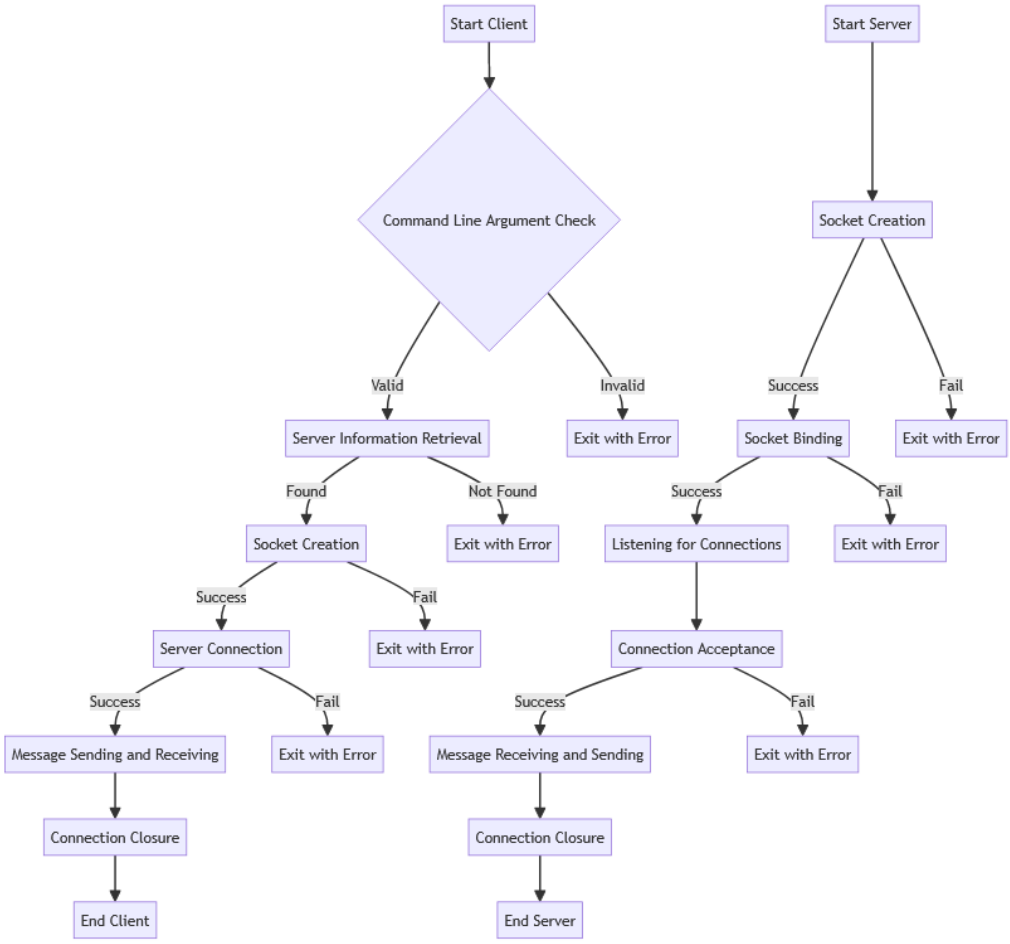
Server Code:

1. **Initialization**:
    - This section includes necessary header files for socket programming and defines constants for server configuration.
    - `SERVER_PORT` specifies the port on which the server will listen for connections.
    - `MAX_PENDING` defines the maximum number of pending connections that can be queued up before the server starts refusing new connections.
    - `MAX_LINE` defines the maximum length of messages that can be sent or received.
2. **Socket Creation**:
    - The `socket()` function creates a new socket descriptor for the server to communicate over.
    - It specifies the communication domain (`PF_INET` for IPv4), socket type (`SOCK_STREAM` for TCP), and protocol (0 for default protocol for the chosen domain and type).
3. **Binding**:
    - The `bind()` function associates the socket with a specific IP address and port number.
    - It takes the socket descriptor, server address structure (`sin`), and size of the address structure as arguments.
4. **Listening for Connections**:
    - The `listen()` function prepares the socket to accept incoming connections.
    - It takes the socket descriptor and the maximum number of pending connections as arguments.
5. **Accepting Connections**:
    - Inside the infinite loop, the server waits for incoming connections using the `accept()` function.
    - When a client connects, `accept()` returns a new socket descriptor (`new_s`) specific to that client.
6. **Receiving and Sending Data**:
    - Within the inner loop, the server receives data from the client using the `recv()` function.
    - It then echoes the received message back to the client using the `send()` function.
    - The server loops until the `recv()` function returns 0, indicating that the client has closed the connection.
7. **Closing Connection**:
    - After handling communication with a client, the server closes the connection using the `close()` function.

Client Code:

1. **Initialization**:
    - This section includes necessary header files, similar to the server, for socket programming and defines constants for client configuration.
2. **Command Line Argument Checking**:
    - The client checks if the hostname of the server is provided as a command-line argument.
    - If no hostname is provided or more than one argument is provided, it prints a usage message indicating the correct usage and exits the program with an error code.
3. **Server Information Retrieval**:
    - The client retrieves server information (such as IP address) based on the provided hostname using the `gethostbyname()` function.
    - If the hostname is not valid or cannot be resolved to an IP address, an error message is printed, and the program exits.
4. **Socket Creation**:
    - The client, like the server, creates a socket descriptor using the `socket()` function to establish communication with the server.
    - It specifies the communication domain (`PF_INET` for IPv4), socket type (`SOCK_STREAM` for TCP), and protocol (0 for default).
5. **Connection to Server**:
    - The client connects to the server using the `connect()` function.
    - This function establishes a connection to the server, specifying the socket descriptor, server address structure (`sin`), and size of the address structure as arguments.
    - If the connection fails (for example, due to the server being unreachable), an error message is printed, and the program exits.
6. **Main Client Loop**:
    - Inside the loop, the client repeatedly sends messages to the server and waits for responses.
    - It reads user input using `fgets()` to get the message to send to the server.
    - The message is sent to the server using the `send()` function.
    - After sending the message, the client waits to receive a response from the server using the `recv()` function.
    - The loop continues until the user decides to terminate the client.
7. **Closing Connection**:
    - After communication is done or the user decides to terminate the client, the connection to the server is closed using the `close()` function.

Block Diagram



TESTING AND RESULTS

The client and server programs were developed and written in Visual Studio Code. The code was compiled and executed directly from the integrated terminal within the Visual Studio Code environment.

Client Tests

Test 1: Connection to Server

Objective: To test if the client can successfully connect to a server.

Procedure: The client program was run from the integrated terminal in Visual Studio Code with the hostname of a running server as a command line argument.

Expected Result: The client should connect to the server and print "Connected to server."

Result

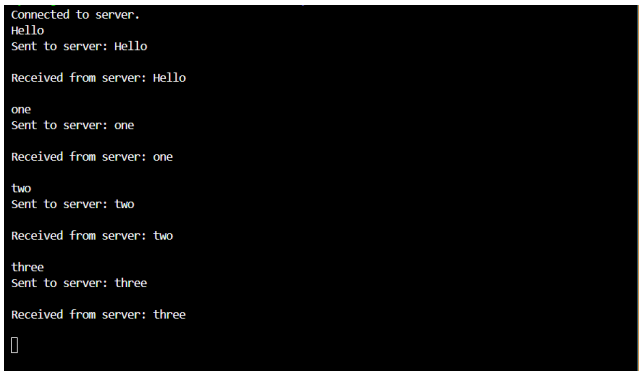
Test 2: Message Sending and Receiving

Objective: To test if the client can send messages to the server and receive responses.

Procedure: After connecting to the server, a message was typed into the client program and enter was pressed.

Expected Result: The client should send the message to the server, print "Sent to server: Hello", receive a response from the server, and print "Received from server: Hello".

Result



Server Tests

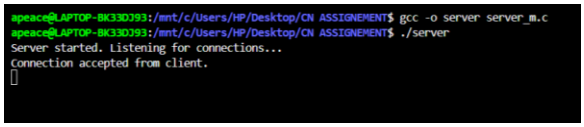
Test 1: Accepting Connections

Objective: To test if the server can accept connections from clients.

Procedure: The server program was run from the integrated terminal in Visual Studio Code. Then, the client program was run with the hostname of the server as a command line argument.

Expected Result: The server should accept the connection from the client and print "Connection accepted from client."

Result:



Conclusion: The server can accept connections from clients.

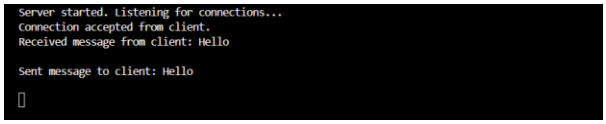
Test 2: Message Receiving and Sending

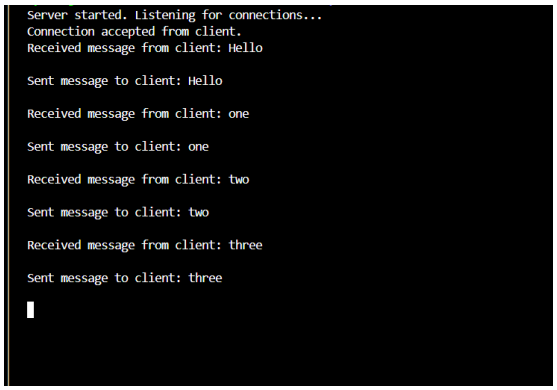
Objective: To test if the server can receive messages from the client and send responses.

Procedure: After the client connected to the server, a message was typed into the client program and enter was pressed.

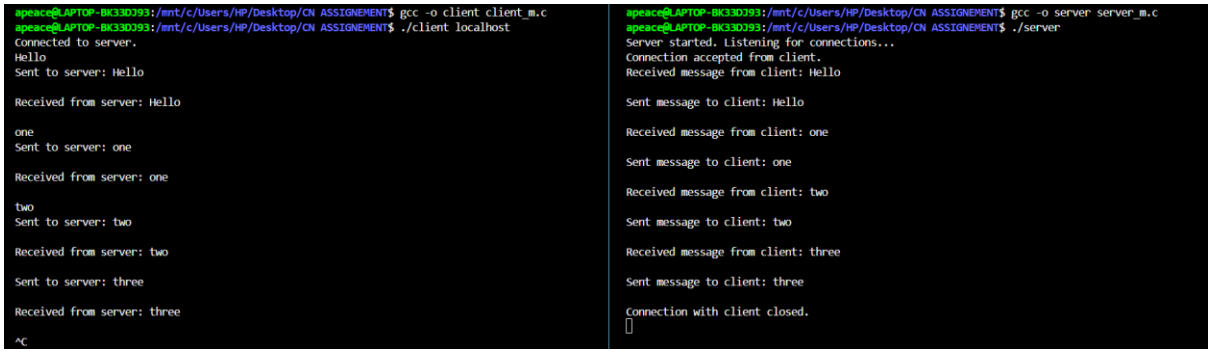
Expected Result: The server should receive the message from the client, print "Received from client: Hello" send a response to the client, and print "Sent message to client: Hello."

Result:





OUTPUT



```
apeace@LAPTOP-BK33DJ93:/mnt/c/Users/HP/Desktop/CN ASSIGNEMENT$ gcc -o client client_m.c
apeace@LAPTOP-BK33DJ93:/mnt/c/Users/HP/Desktop/CN ASSIGNEMENT$ ./client localhost
Connected to server.
Hello
Sent to server: Hello

Received from server: Hello

one
Sent to server: one

Received from server: one

two
Sent to server: two

Received from server: two

Sent to server: three

Received from server: three

^C
```

```
apeace@LAPTOP-BK33DJ93:/mnt/c/Users/HP/Desktop/CN ASSIGNEMENT$ gcc -o server server_m.c
apeace@LAPTOP-BK33DJ93:/mnt/c/Users/HP/Desktop/CN ASSIGNEMENT$ ./server
Server started. Listening for connections...
Connection accepted from client.
Received message from client: Hello

Sent message to client: Hello

Received message from client: one

Sent message to client: one

Received message from client: two

Sent message to client: two

Received message from client: three

Sent message to client: three

Connection with client closed.
```