

WIPRO CAPSTONE PROJECT

ON

TCP CALCULATOR WITH CHAT CLIENT MESSAGING

**C++ PROGRAMMING || LINUX SYSTEM
PROGRAMMING**

SUBMITTED BY

DEEKSHA R SAPATE [RESUME NUMBER :22992538]

INTRODUCTION

The client-server model is a fundamental concept in network programming, enabling multiple clients to communicate with a central server to perform various tasks. This model is widely used in applications ranging from web services to multiplayer games, where the server acts as a central hub that processes requests and sends appropriate responses to clients. In this context, the client module plays a crucial role in establishing and maintaining this communication.

The client module is responsible for connecting to the server, managing user interactions, and facilitating the exchange of data between the client and the server. It initiates the communication by creating a socket, which serves as the endpoint for data transmission. Once the socket is created, the client attempts to connect to the server using the server's IP address and port number. Upon successful connection, the client can send requests to the server, such as chat messages or arithmetic operations for the server's calculator service.

The client module is designed to handle multiple functionalities, including a chat system that allows the client to send and receive messages in real-time. Additionally, it provides a calculator feature, where the client inputs numbers and operations, and the server performs the calculation and returns the result. Throughout this process, the client module ensures that errors are handled gracefully, providing feedback to the user in case of issues such as connection failures or invalid inputs. In essence, the client module serves as the user's gateway to the server's services, enabling efficient and reliable communication in a networked environment.

Objectives

1. **Implement a Client-Server Communication Model:**
 - a. Establish a robust client-server architecture using TCP sockets that facilitates bi-directional communication. The project aims to provide a practical demonstration of how clients and servers interact in real-time.
2. **Develop a Real-Time Chat System:**
 - a. Create a functional chat system where clients can send and receive messages in real-time. This feature will help users communicate effectively, simulating a basic messaging service over a network.
3. **Implement a Remote Calculator Service:**
 - a. Integrate a calculator feature within the server that performs various arithmetic operations (addition, subtraction, multiplication, division, percentage, and square root) based on client requests. The server will process the data sent by the client and return the computed results.

4. **Enhance User Interaction:**
 - a. Ensure that the client module provides a seamless user experience by handling user inputs, displaying server responses clearly, and managing errors efficiently. The objective is to make the application user-friendly and intuitive.
5. **Ensure Robust Error Handling:**
 - a. Implement comprehensive error-handling mechanisms in both the client and server modules to manage common issues such as connection failures, invalid inputs, and server errors. The aim is to create a reliable and stable application.
6. **Log Server-Side Calculations:**
 - a. Introduce a logging mechanism on the server to record all calculations performed. This objective ensures that all operations are documented for future reference or auditing purposes.
7. **Provide Practical Experience with Network Programming:**
 - a. Offer a hands-on learning opportunity for understanding the fundamentals of socket programming, data transmission over networks, and client-server dynamics in a real-world context.

Software and Hardware Requirements

Software Requirements

1. **Operating System:**
 - Linux-based OS
2. **Development Tools:**
 - **Compiler:** GCC (GNU Compiler Collection) for C or C++.
 - **Editor/IDE:** Any text editor or IDE for coding, such as VS Code, Vim, Emacs, or Eclipse.
3. **Libraries:**
 - **Standard Libraries:** Standard C library (libc), which is generally included with GCC.
 - **Networking Libraries:** Standard networking libraries that come with the OS (e.g., `arpa/inet.h`, `sys/socket.h`).
4. **Networking:**
 - **TCP/IP Stack:** The application relies on the TCP/IP stack, which is included in the OS.
5. **Utilities:**
 - **File System Access:** Ability to read from and write to files for logging calculations and chat messages.

Hardware Requirements

1. **Processor:**
 - A processor with a minimum of 1 GHz, preferably multi-core (e.g., Intel Core i3 or better).

2. **Memory (RAM):**

- At least 512 MB RAM; 1 GB or more recommended for better performance.

3. **Storage:**

- **Server:** At least 10 MB of free disk space for the application and logs. More space may be required based on log file size.
- **Client:** Minimal storage required, mainly for the application and its configuration files.

4. **Network:**

- A network interface capable of handling TCP connections (e.g., Ethernet or Wi-Fi).

Features

1. Chat Functionality

The chat functionality allows the client and server to exchange text messages. The communication continues until the client sends the command exit to terminate the chat session. All chat conversations are logged on the server in a file named chat.log.

2. Calculator Functionality

The calculator functionality supports the following operations:

1. **Addition:** Adds two numbers.
2. **Subtraction:** Subtracts the second number from the first.
3. **Multiplication:** Multiplies two numbers.
4. **Division:** Divides the first number by the second (with error handling for division by zero).
5. **Percentage:** Calculates the percentage of one number relative to another.
6. **Square Root:** Calculates the square root of a single number (with error handling for negative inputs).

All calculations are logged on the server in a file named calculations.log.

Project Overview

Dependencies used

Both the client and server applications in this project rely on several standard C libraries, each of which serves a specific purpose in handling networking, input/output, and mathematical operations. Below is an explanation of the role each library plays:

1. **arpa/inet.h:** This header file is used for definitions related to the Internet operations, specifically for converting values between host and network byte order. It provides functions such as `inet_addr`, `inet_ntoa`, and `htonl`, which are used to handle IP addresses in the application.
2. **netdb.h:** This header file defines the network database operations. It provides functions and structures used for translating hostnames into IP addresses and vice versa. It also defines structures like `hostent` and functions like `gethostbyname` that are essential in network programming.
3. **stdio.h:** The standard input/output library is used for handling basic I/O operations in C. Functions such as `printf`, `scanf`, `fgets`, and `fprintf` are used throughout the client and server programs to read from and write to the console or files.
4. **stdlib.h:** The standard library provides utility functions for memory allocation (`malloc`, `calloc`), process control (`exit`, `system`), conversions (`atoi`, `atof`), and other utility functions. In this project, it's primarily used for memory management and exiting the program when errors occur.
5. **string.h:** This header file provides functions for handling strings in C, such as `strlen`, `strcpy`, `strncpy`, `strcmp`, `strcat`, `strcspn`, and `bzero`. These functions are used for manipulating and managing strings, which are essential for processing user input and communication between the client and server.
6. **strings.h:** Similar to `string.h`, this header provides additional string handling functions, specifically those that are part of the BSD operating system. For example, `bzero` is used to zero out a block of memory, and `strcasecmp` is used to compare strings in a case-insensitive manner.
7. **sys/socket.h:** This header is crucial for socket programming, providing definitions for the structures and functions needed to create, bind, listen, connect, send, and receive data over network sockets. Functions like `socket`, `bind`, `listen`, `accept`, `connect`, and `send` are defined here.
8. **unistd.h:** This header defines miscellaneous symbolic constants and types, as well as various functions related to system calls. It includes functions like `read`, `write`, `close`, and `fork`, which are used for file and socket operations in the project. It also provides access to the POSIX operating system API, making it essential for process control.
9. **math.h:** The mathematical library provides a set of functions to perform mathematical operations, such as `sqrt`, `pow`, and `log`. In this project, it is used primarily for performing square root calculations and other arithmetic operations required by the calculator functionality.

Server Module (server.c)

The `server.c` file is the server-side component of a client-server application. It handles multiple responsibilities such as creating and managing sockets, accepting client connections, processing client requests (like chatting and calculations), and logging operations. Below is an explanation of the various key components of the server module:

1. Socket Creation: `socket()`

- Function: `socket(AF_INET, SOCK_STREAM, 0);`
 1. This function creates a new socket, which is an endpoint for communication.
 2. `AF_INET` specifies the address family (IPv4).
 3. `SOCK_STREAM` indicates that the socket is for a stream-based protocol (TCP).
 4. The socket created is used for establishing a connection between the server and client.
 5. If the socket creation fails, the program prints an error message and exits.

2. Binding: `bind()`

- Function: `bind(sockfd, (SA*)&servaddr, sizeof(servaddr));`
 1. The `bind` function associates the socket with a specific IP address and port number.
 2. `servaddr` is a structure that contains the IP address (`INADDR_ANY` allows the server to accept connections on any of its network interfaces) and the port number (`PORT` is set to 8080).
 3. Binding is crucial because it tells the operating system which port the server will be listening to for incoming connections.
 4. Failure to bind results in an error message, and the server exits.

3. Listening: `listen()`

- Function: `listen(sockfd, 5);`
 1. The `listen` function makes the server ready to accept incoming connection requests.
 2. The second argument (5) specifies the maximum number of pending connections that can be queued.
 3. After this step, the server goes into a passive mode, waiting for clients to connect.
 4. If listening fails, the server exits after printing an error message.

4. Accepting Connections: `accept()`

- Function: `connfd = accept(sockfd, (SA*)&cli, &len);`
 1. `accept` waits for an incoming connection request from a client.
 2. When a client connects, `accept` creates a new socket (`connfd`) for the connection and returns it.
 3. This new socket is used to communicate with the client, allowing the server to handle multiple clients simultaneously by creating a new socket for each client.
 4. If accepting a connection fails, the server exits with an error message.

5. Chat Handling: chat()

- Function: chat(connfd);
 1. The chat function handles the chat functionality between the server and the client.
 2. It continuously reads messages from the client and sends responses back.
 3. The communication continues until the client sends the message "exit," signaling the end of the chat session.
 4. This function demonstrates basic message exchange using sockets.

6. Calculator Handling: perform_calculation()

- Function: perform_calculation(connfd);
 1. The perform_calculation function provides the calculator service to the client.
 2. It presents the client with a menu of arithmetic operations, such as addition, subtraction, multiplication, division, percentage calculation, and square root.
 3. The server reads the client's choices and operands, performs the corresponding calculation, and sends the result back to the client.
 4. This function demonstrates how to handle specific tasks based on client requests.

7. Logging: log_calculation()

- Function: log_calculation(const char *operation, float num1, float num2, float result);
 1. This function logs each calculation performed by the server to a file named calculations.log.
 2. It records the operation type, operands, and result, which helps in maintaining a record of all calculations handled by the server.
 3. This logging is crucial for auditing and debugging purposes.
 4. The function ensures the log file is properly opened and closed, and it handles errors in file operations gracefully.

8. Error Handling:

1. Throughout the server module, error handling is implemented to ensure the program does not crash unexpectedly.
2. For instance, if socket creation, binding, listening, or accepting fails, the server prints an error message and exits cleanly.
3. Additionally, if file operations (like opening the log file) fail, appropriate error messages are shown, and the operation is either retried or skipped based on the context.
4. This robust error handling ensures that the server can handle various failure scenarios gracefully.

Client Module (client.c)

The client.c file is the client-side component of a client-server application. It is responsible for establishing a connection with the server, enabling chat and calculator functionalities, handling user input, and managing errors during communication. Below is a detailed explanation of the key components of the client module:

1. Socket Creation: socket()

- Function: `sockfd = socket(AF_INET, SOCK_STREAM, 0);`
 1. The socket function creates a new socket that will be used by the client to communicate with the server.
 2. `AF_INET` specifies the address family for IPv4.
 3. `SOCK_STREAM` indicates that the socket is for TCP (a stream-oriented protocol).
 4. The function returns a file descriptor (`sockfd`) that is used to refer to this socket in subsequent operations.
 5. If socket creation fails, the program will print an error message and exit.

2. Connecting to Server: connect()

- Function: `connect(sockfd, (SA*)&servaddr, sizeof(servaddr));`
 1. The connect function is used to establish a connection with the server.
 2. `servaddr` is a structure that contains the server's IP address and port number (which must match the server's configuration).
 3. Once the connection is successful, the client can start sending and receiving data through this socket.
 4. If the connection fails, the client prints an error message and exits.

3. Chat Functionality: chat()

- Function: `chat(sockfd);`
 1. The chat function allows the client to send and receive messages to and from the server.
 2. It operates in a loop, continuously taking input from the user and sending it to the server while also receiving messages from the server.
 3. The loop continues until the user types "exit," at which point the chat session ends.
 4. This function demonstrates how the client handles basic text-based communication with the server.

4. Calculator Functionality: perform_calculation()

- Function: `perform_calculation(sockfd);`

1. The `perform_calculation` function allows the client to interact with the server's calculator service.
2. The client selects an arithmetic operation (like addition, subtraction, etc.), enters the required numbers, and sends this data to the server.
3. The server performs the calculation and sends the result back to the client, which is then displayed to the user.
4. This function shows how specific services can be requested from the server and how the results are processed and displayed.

5. User Input Handling:

1. The client module is responsible for capturing and validating user input for both chat and calculator operations.
2. User input is taken using standard input functions (like `scanf` and `fgets`).
3. The input is then formatted or converted as needed before being sent to the server.
4. This ensures that the server receives well-formed data, reducing the likelihood of errors during communication.

6. Error Handling:

1. The client module includes error-handling mechanisms to manage potential issues during communication with the server.
2. For instance, if there is a failure in socket creation, connection, or communication, the client prints an appropriate error message and exits gracefully.
3. This helps in preventing unexpected crashes and ensures that the client can handle various failure scenarios effectively.

CODE

Server.c

```
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <unistd.h>
#include <math.h>

#define MAX 256 // Increased buffer size
#define PORT 8080
#define SA struct sockaddr

void log_calculation(const char *operation, float num1, float num2, float result) {
    FILE *file = fopen("calculations.log", "a");
    if (file == NULL) {
        perror("Unable to open file");
        return;
    }
    fprintf(file, "Operation: %s\n", operation);
    fprintf(file, "Operand 1: %.2f\n", num1);
    fprintf(file, "Operand 2: %.2f\n", num2);
    fprintf(file, "Result: %.2f\n", result);
    fprintf(file, "-----\n");
    fclose(file);
}

void log_chat(const char *from, const char *message) {
    FILE *file = fopen("chat.log", "a");
    if (file == NULL) {
        perror("Unable to open file");
        return;
    }
    fprintf(file, "%s: %s\n", from, message);
    fclose(file);
}

void perform_calculation(int connfd) {
    char buff[MAX];
    float num1, num2, result;
    int choice;

    snprintf(buff, sizeof(buff), "Choose operation:\n"
                                   "1. Addition\n"
                                   "2. Subtraction\n")
```

```

        "3. Multiplication\n"
        "4. Division\n"
        "5. Percentage\n"
        "6. Square root\n");
write(connfd, buff, strlen(buff));

bzero(buff, MAX);
read(connfd, buff, sizeof(buff));
sscanf(buff, "%d", &choice);

switch (choice) {
    case 1:
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num1);
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num2);
        result = num1 + num2;
        snprintf(buff, sizeof(buff), "Result: %.2f", result);
        log_calculation("Addition", num1, num2, result);
        break;
    case 2:
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num1);
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num2);
        result = num1 - num2;
        snprintf(buff, sizeof(buff), "Result: %.2f", result);
        log_calculation("Subtraction", num1, num2, result);
        break;
    case 3:
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num1);
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num2);
        result = num1 * num2;
        snprintf(buff, sizeof(buff), "Result: %.2f", result);
        log_calculation("Multiplication", num1, num2, result);
        break;
    case 4:
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num1);
        bzero(buff, MAX);

```

```

        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num2);
        if (num2 != 0)
            result = num1 / num2;
        else {
            snprintf(buff, sizeof(buff), "Error: Division by zero");
            write(connfd, buff, strlen(buff));
            return;
        }
        snprintf(buff, sizeof(buff), "Result: %.2f", result);
        log_calculation("Division", num1, num2, result);
        break;
    case 5:
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num1);
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num2);
        result = (num1 * num2) / 100;
        snprintf(buff, sizeof(buff), "Result: %.2f", result);
        log_calculation("Percentage", num1, num2, result);
        break;
    case 6:
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        sscanf(buff, "%f", &num1);
        if (num1 >= 0)
            result = sqrt(num1);
        else {
            snprintf(buff, sizeof(buff), "Error: Negative input for square root");
            write(connfd, buff, strlen(buff));
            return;
        }
        snprintf(buff, sizeof(buff), "Result: %.2f", result);
        log_calculation("Square root", num1, 0, result);
        break;
    default:
        snprintf(buff, sizeof(buff), "Invalid option");
        break;
}
write(connfd, buff, strlen(buff));
}

void chat(int connfd) {
    char buff[MAX];

    while (1) {
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));

```

```

        if (strcmp("exit", buff, 4) == 0) {
            printf("Client exited chat.\n");
            log_chat("Client", "Exited chat");
            break;
        }
        printf("From client: %s\n", buff);
        log_chat("Client", buff);

        printf("To client: ");
        bzero(buff, MAX);
        fgets(buff, MAX, stdin);
        buff[strcspn(buff, "\n")] = 0; // Remove newline character
        write(connfd, buff, strlen(buff));
        log_chat("Server", buff);
    }
}

int main() {
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("Socket creation failed...\n");
        exit(0);
    } else
        printf("Socket successfully created..\n");

    bzero(&servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    if (bind(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("Socket bind failed...\n");
        exit(0);
    } else
        printf("Socket successfully binded..\n");

    if (listen(sockfd, 5) != 0) {
        printf("Listen failed...\n");
        exit(0);
    } else
        printf("Server listening..\n");

    len = sizeof(cli);

```

```

connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
    printf("Server accept failed...\n");
    exit(0);
} else
    printf("Server accepted the client...\n");

while (1) {
    char buff[MAX];
    bzero(buff, MAX);
    read(connfd, buff, sizeof(buff));
    int choice;
    sscanf(buff, "%d", &choice);

    switch (choice) {
        case 1:
            chat(connfd);
            break;
        case 2:
            perform_calculation(connfd);
            break;
        case 3:
            close(connfd);
            close(sockfd);
            exit(0);
        default:
            snprintf(buff, sizeof(buff), "Invalid option");
            write(connfd, buff, strlen(buff));
            break;
    }
}

return 0;
}

```

Client.c

```
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <unistd.h>

#define MAX 256
#define PORT 8080
#define SA struct sockaddr

void perform_calculation(int sockfd) {
    char buff[MAX];
    float num1, num2;
    int choice;
    bzero(buff, MAX);
    read(sockfd, buff, sizeof(buff));
    printf("%s", buff);

    bzero(buff, MAX);
    fgets(buff, MAX, stdin);
    buff[strcspn(buff, "\n")] = 0; // Remove newline character
    write(sockfd, buff, strlen(buff));

    if (buff[0] >= '1' && buff[0] <= '5') { // For choices 1 to 5
        printf("Enter two numbers:\n");
        bzero(buff, MAX);
        fgets(buff, MAX, stdin);
        buff[strcspn(buff, "\n")] = 0; // Remove newline character
        write(sockfd, buff, strlen(buff));
        bzero(buff, MAX);
        fgets(buff, MAX, stdin);
        buff[strcspn(buff, "\n")] = 0; // Remove newline character
        write(sockfd, buff, strlen(buff));
    } else if (buff[0] == '6') { // For square root
        printf("Enter a number:\n");
        bzero(buff, MAX);
        fgets(buff, MAX, stdin);
        buff[strcspn(buff, "\n")] = 0; // Remove newline character
        write(sockfd, buff, strlen(buff));
    } else {
        printf("Invalid option\n");
        return;
    }
}
```

```

    // Read and display the result
    bzero(buff, MAX);
    read(sockfd, buff, sizeof(buff));
    printf("From Server: %s", buff);
}

void chat(int sockfd) {
    char buff[MAX];

    while (1) {
        bzero(buff, MAX);
        printf("Enter message: ");
        fgets(buff, MAX, stdin);
        buff[strcspn(buff, "\n")] = 0; // Remove newline character
        write(sockfd, buff, strlen(buff));
        if (strncmp("exit", buff, 4) == 0) {
            printf("Exiting chat...\n");
            break;
        }
        bzero(buff, MAX);
        read(sockfd, buff, sizeof(buff));
        printf("From Server: %s", buff);
    }
}

int main() {
    int sockfd;
    struct sockaddr_in servaddr;
    int choice;
    char buff[MAX];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("Socket creation failed...\n");
        exit(0);
    } else
        printf("Socket successfully created..\n");

    bzero(&servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("Connection with the server failed...\n");
        exit(0);
    } else

```



```

    printf("Connected to the server..\n");

while (1) {
    printf("Choose option:\n");
    printf("1. Chat\n");
    printf("2. Calculator\n");
    printf("3. Quit\n");
    printf("Enter choice: ");
    fgets(buff, MAX, stdin);
    buff[strcspn(buff, "\n")] = 0; // Remove newline character
    write(sockfd, buff, strlen(buff));
    sscanf(buff, "%d", &choice);

    switch (choice) {
        case 1:
            chat(sockfd);
            break;
        case 2:
            perform_calculation(sockfd);
            break;
        case 3:
            close(sockfd);
            exit(0);
        default:
            printf("Invalid option\n");
            break;
    }
}

close(sockfd);
return 0;
}

```

OUTPUT

CHAT MESSAGES

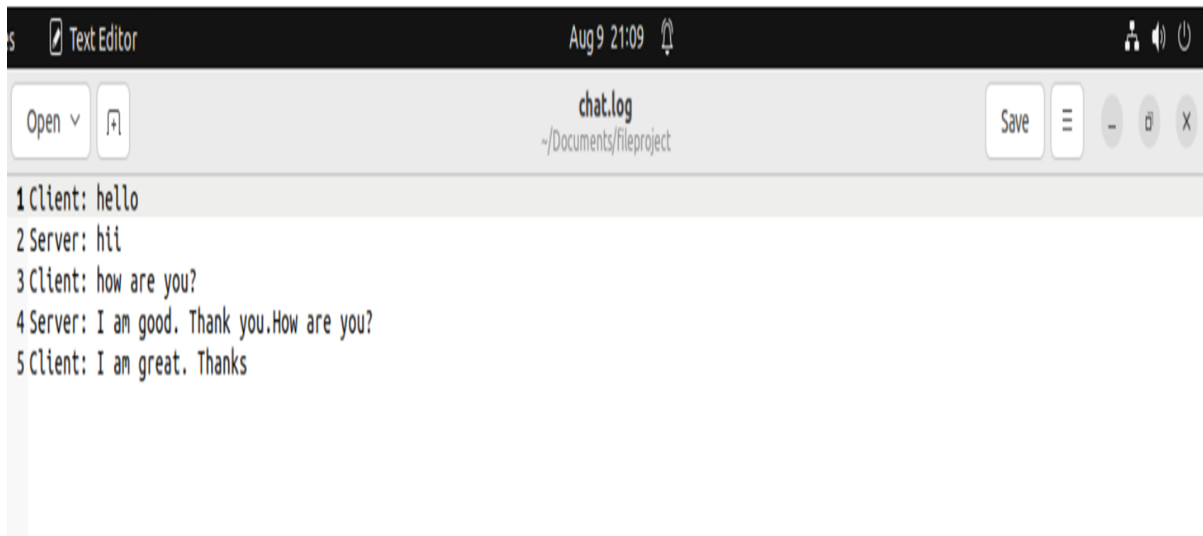
SERVER

```
rps@rps-virtual-machine: ~/Documents/fileproject
rps@rps-virtual-machine:~/Documents/fileproject$ ./server
Socket successfully created..
Socket successfully binded..
Server listening..
Server accepted the client...
From client: hello
To client: hii
From client: how are you?
To client: I am good. Thank you.How are you?
From client: I am great. Thanks
To client: 
```

CLIENT

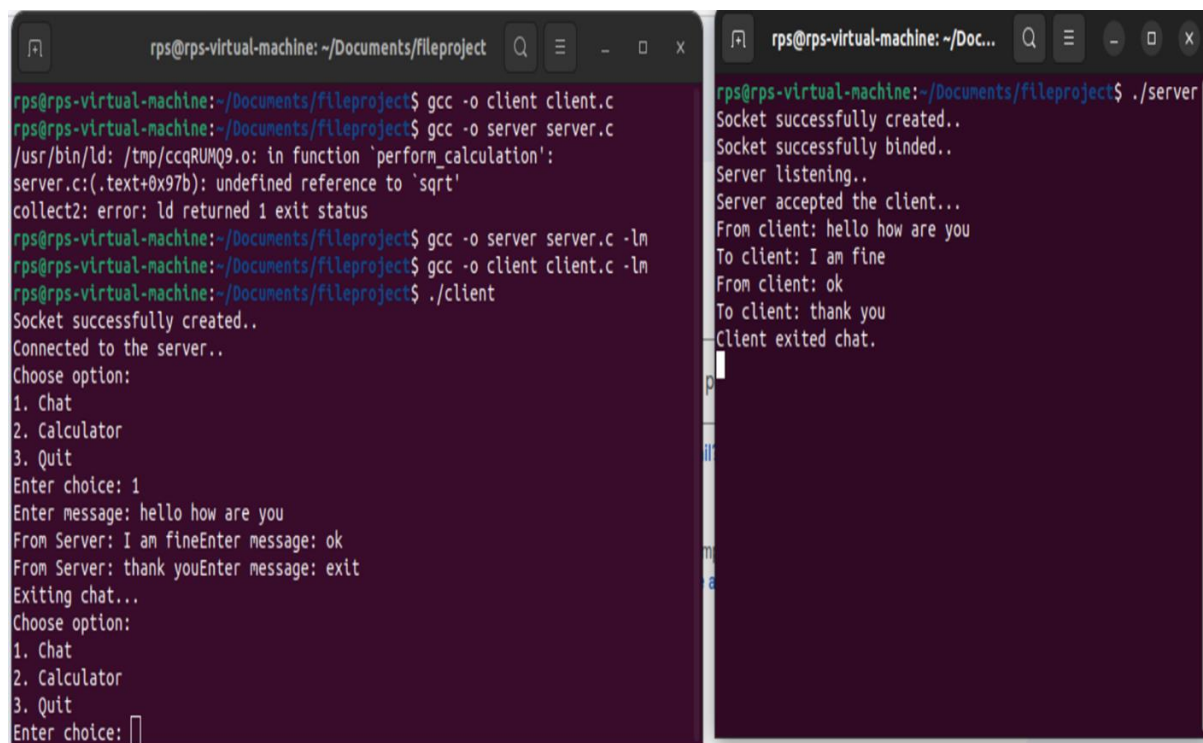
```
rps@rps-virtual-machine: ~/Documents/fileproject
rps@rps-virtual-machine:~/Documents/fileproject$ ./client
Socket successfully created..
Connected to the server..
Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice: 1
Enter message: hello
From Server: hiiEnter message: how are you?
From Server: I am good. Thank you.How are you?Enter message: I am great. Thanks
```

Client – Server messages are stored in a document called chat.log



A screenshot of a text editor window titled 'chat.log' with the path '~/.Documents/fileproject'. The window contains a log of five messages between a client and a server:

```
1 Client: hello
2 Server: hii
3 Client: how are you?
4 Server: I am good. Thank you.How are you?
5 Client: I am great. Thanks
```



Two terminal screenshots showing the compilation and execution of a client-server chat application.

Left Terminal (Client):

```
rps@rps-virtual-machine: ~/Documents/fileproject
rps@rps-virtual-machine:~/Documents/fileproject$ gcc -o client client.c
rps@rps-virtual-machine:~/Documents/fileproject$ gcc -o server server.c
/usr/bin/ld: /tmp/ccqRUMQ9.o: in function 'perform_calculation':
server.c:(.text+0x97b): undefined reference to 'sqrt'
collect2: error: ld returned 1 exit status
rps@rps-virtual-machine:~/Documents/fileproject$ gcc -o server server.c -lm
rps@rps-virtual-machine:~/Documents/fileproject$ gcc -o client client.c -lm
rps@rps-virtual-machine:~/Documents/fileproject$ ./client
Socket successfully created..
Connected to the server..
Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice: 1
Enter message: hello how are you
From Server: I am fineEnter message: ok
From Server: thank youEnter message: exit
Exiting chat...
Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice: 
```

Right Terminal (Server):

```
rps@rps-virtual-machine: ~/Documents/fileproject$ ./server
Socket successfully created..
Socket successfully binded..
Server listening..
Server accepted the client...
From client: hello how are you
To client: I am fine
From client: ok
To client: thank you
Client exited chat.
```

CALCULATOR OPERATIONS

SERVER

```
rps@rps-virtual-machine: ~/Documents/fileproject

rps@rps-virtual-machine:~/Documents/fileproject$ ./server
Socket successfully created..
Socket successfully binded..
Server listening..
Server accepted the client...
```

CLIENT

```
rps@rps-virtual-machine: ~/Documents/fileproject

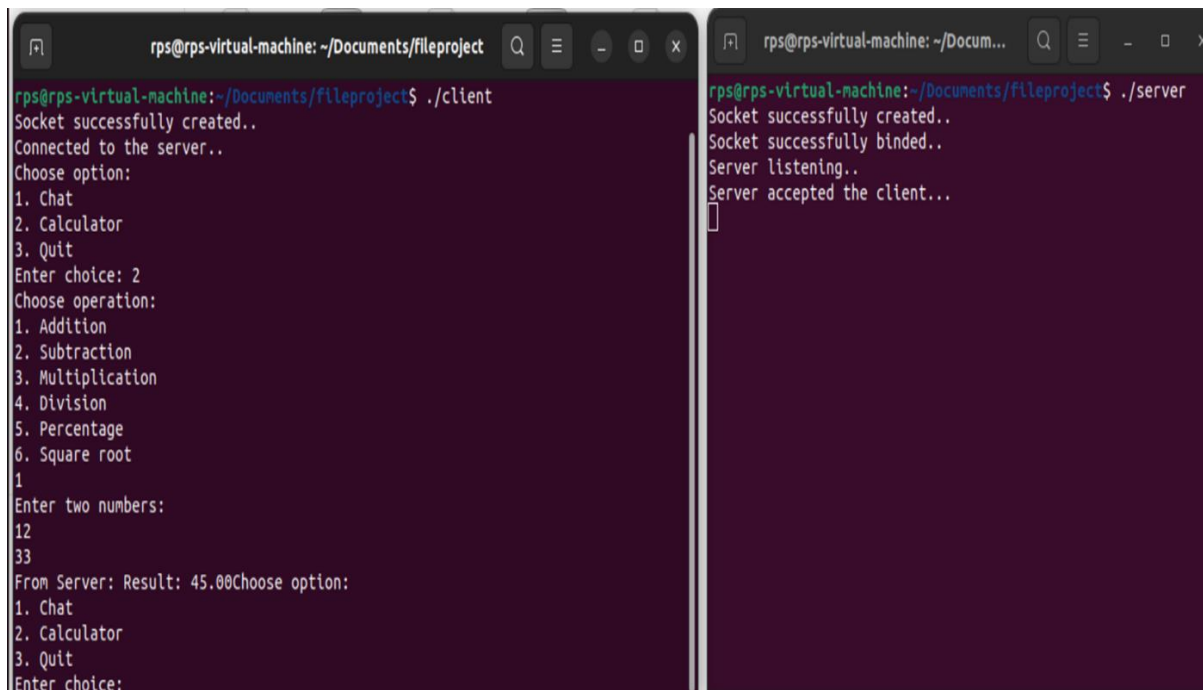
rps@rps-virtual-machine:~/Documents/fileproject$ ./client
connected to the server..
Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice: 2
Choose operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Percentage
6. Square root
1
Enter two numbers:
12
56
From Server: Result: 68.00Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice: 2
Choose operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Percentage
6. Square root
4
Enter two numbers:
12
78
From Server: Result: 0.15Choose option:
1. Chat
2. Calculator
3. Quit
```

Client calculator operations are stored in a document called **calculations.log**



A screenshot of a text editor window titled "calculations.log" with the path "~/Documents/fileproject". The window contains a log of calculator operations, numbered 1 through 15. The operations are grouped into three sections: Addition (lines 1-5), Division (lines 6-10), and Multiplication (lines 11-15). Each section starts with a header line (e.g., "1 Operation: Addition"), followed by operand lines (e.g., "2 Operand 1: 12.00", "3 Operand 2: 56.00"), and a result line (e.g., "4 Result: 68.00"). The log is separated by dashed lines.

```
1 Operation: Addition
2 Operand 1: 12.00
3 Operand 2: 56.00
4 Result: 68.00
5 .....
6 Operation: Division
7 Operand 1: 12.00
8 Operand 2: 78.00
9 Result: 0.15
10 .....
11 Operation: Multiplication
12 Operand 1: 45.00
13 Operand 2: 2.00
14 Result: 90.00
15 .....
```



A screenshot of two terminal windows side-by-side, both running on an "rps@rps-virtual-machine" with the directory "~/Documents/fileproject". The left terminal shows the client-side execution of the program, where the user selects the "Calculator" option and performs a multiplication of 45 by 2. The right terminal shows the server-side execution, which successfully creates a socket, binds it, listens for connections, and accepts the client.

```
rps@rps-virtual-machine: ~/Documents/fileproject
rps@rps-virtual-machine:~/Documents/fileproject$ ./client
Socket successfully created..
Connected to the server..
Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice: 2
Choose operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Percentage
6. Square root
1
Enter two numbers:
12
33
From Server: Result: 45.00Choose option:
1. Chat
2. Calculator
3. Quit
Enter choice:
```

```
rps@rps-virtual-machine: ~/Documents/fileproject$ ./server
Socket successfully created..
Socket successfully binded..
Server listening..
Server accepted the client...
```

Future Implementations

1. **Enhanced User Interface:** Develop a graphical user interface (GUI) for a more intuitive user experience.
2. **Extended Calculator Features:** Add more complex mathematical functions such as exponentiation, logarithms, and trigonometric functions.
3. **Encryption and Security:** Implement encryption for secure communication between client and server.
4. **Multi-Client Support:** Extend the server to handle multiple clients simultaneously.
5. **Persistent Data Storage:** Implement a database to store chat history and calculation results persistently.
6. **Web-based Client:** Develop a web-based client using technologies like HTML, CSS, and JavaScript for broader accessibility.

Conclusion

The development of this client-server application has successfully demonstrated the integration of basic network programming and user interaction through a TCP socket-based architecture. The project comprises two main components: a server that handles client requests for both chat and calculation functionalities, and a client that interacts with the server to perform these operations. The project provides a demonstration of a client-server application, with a focus on communication and arithmetic operations. The chat details as well as the calculator operations are stored in separate files. It serves as a foundation for building more complex and feature-rich networked applications.