

LINUX MULTI-THREADED CLIENT-SERVER
USING SHARED MEMORY PROJECT
DOCUMENT

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	2

Table of contents

1. Overview	3
2. Introduction	4
3. Project Scope	5
4.1 Functional Requirements	5
4.2. Non-Functional Requirements	5
5. System Design	6
6. Code comments and Explanation:	7
Compiling Application:	16
Running Application:	16
Testing the Applications	16
7.The Synchronization Mechanism	17
8. Test Cases and Results	18
Test Case 1: Single Client Communication	18
Test Case 2: Multiple Client Communication	19
9. Conclusion	20

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	3

1. OVERVIEW

This project is designed to provide hands-on experience in developing a multi-threaded client-server application using shared memory in a Linux environment. The main objectives are to understand multi-threading, inter-process communication (IPC) using threads (pthreads), and shared memory. The project involves creating a server capable of handling multiple client connections simultaneously and ensuring proper synchronization and data consistency.

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	4

2. Introduction

This project demonstrates the implementation of a client-server communication system using inter-process communication (IPC) mechanisms such as shared memory and semaphores in the C programming language. The core objective is to enable multiple clients to interact with a server simultaneously, with shared memory used to store and update the exchange of messages between them.

In this system, the server is designed to handle multiple clients concurrently by spawning a new thread for each client connection. Each client can send messages to the server, which processes the message, updates the shared memory, and responds back to the client. The shared memory segment is protected by a semaphore to ensure that only one process can access it at a time, preventing race conditions and ensuring data consistency.

On the client side, users can send messages to the server and receive responses in real-time. Clients also have the ability to view the current state of the shared memory, thus seeing the most recent communication updates.

This project highlights the practical application of these concepts in a concurrent programming environment.

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	5

3. Project Scope

The project includes the following key components:

- **Multi-Threaded Server Application:** Develop a server that can handle multiple client connections concurrently using pthreads.
- **Client Application:** Implement a client that connects to the server and uses shared memory to send and receive messages.
- **Shared Memory:** To define a shared memory segment accessible by both the client and the server.
- **Synchronization:** Implement synchronization mechanisms (semaphores) to manage concurrent access to shared memory, ensuring data consistency and preventing race conditions.
- **Communication Protocol:** Define and implement a simple protocol for client-server interactions, including message structures for data exchange.
- **Error Handling:** Incorporate proper error handling in both client and server applications, providing meaningful error messages and addressing edge cases.

4. Requirements

4.1 Functional Requirements

1. **Multi-Threaded Server:**
 - Implement a server application that can handle multiple client connections simultaneously.
 - Use threads (pthreads) to manage multiple client connections.
2. **Client Application:**
 - Implement a client application that connects to the server.
 - Use shared memory to send and receive messages to and from the server.
3. **Shared Memory:**
 - Use shared memory for IPC.
 - Define a shared memory segment that clients and the server can access.
4. **Synchronization:**
 - Implement synchronization mechanisms such as semaphores to manage concurrent access to shared memory.
 - Ensure data consistency and avoid race conditions.
5. **Communication Protocol:**
 - Define a simple communication protocol for client-server interactions.
 - Implement message structures for sending and receiving data.
6. **Error Handling:**
 - Implement proper error handling in both client and server applications.
 - Provide meaningful error messages and handle edge cases gracefully.

4.2. Non-Functional Requirements

1. Performance:

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	6

- The server should handle multiple clients concurrently without significant delays.
- Shared memory access should be synchronized efficiently to minimize wait times.

2. Reliability:

- The server should handle client disconnections gracefully.
- The logging mechanism should ensure that all messages are recorded accurately.

3. Usability:

- The client interface should be simple and user-friendly, allowing easy message input and display.

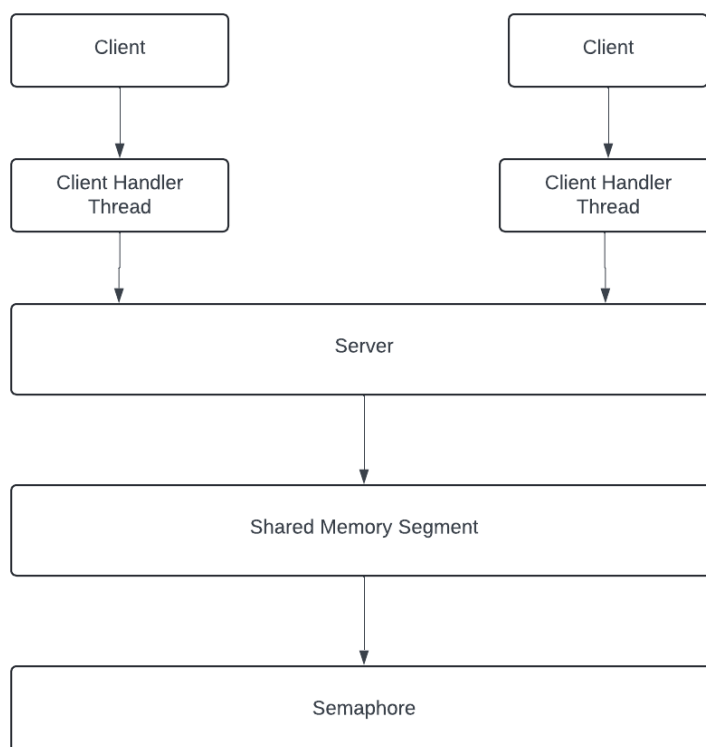
4. Scalability:

- The server design should accommodate an increase in the number of clients with minimal performance degradation.

5. Security:

- Access to shared memory should be controlled to prevent unauthorized access or data corruption.

5. System Design



Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	7

6. Code comments and Explanation:

6.1. Server code:(server.c)

```
//server
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <semaphore.h>

#define PORT 9090
#define MAX_CLIENTS 100
#define SHM_KEY 4
#define SHM_SIZE 1024

sem_t *semaphore;

void *client_handler(void *socket_desc) {
    int client_sock = *(int*)socket_desc;
    free(socket_desc);

    struct sockaddr_in client;
    socklen_t client_len = sizeof(client);
    getpeername(client_sock, (struct sockaddr*)&client, &client_len);
    char client_ip[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &(client.sin_addr), client_ip, INET_ADDRSTRLEN);
    printf("Client connected: \n");
```

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	8

```

// Access shared memory
int shm_id = shmget(SHM_KEY, SHM_SIZE, 0666);
if (shm_id == -1) {
    perror("Failed to open shared memory");
    return NULL;
}

char *shared_mem = (char*)shmat(shm_id, NULL, 0);
if (shared_mem == (char*)-1) {
    perror("Failed to attach shared memory");
    return NULL;
}

// Handle client communication
char buffer[256];
while (1) {
    memset(buffer, 0, 256);
    int read_size = recv(client_sock, buffer, 256, 0);
    if (read_size <= 0) {
        break;
    }printf("Thread is created\n");

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

    sem_wait(semaphore);
    strcpy(shared_mem, buffer); // Write to shared memory
    sem_post(semaphore);

    // Echo back to client
    send(client_sock, buffer, strlen(buffer), 0);
}

printf("Client disconnected: \n");

close(client_sock);
return NULL;
}

int main() {

```


Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	9

```

int server_fd, client_sock, *new_sock;

struct sockaddr_in server, client;

socklen_t client_len = sizeof(client);

pthread_t client_threads[MAX_CLIENTS];

int client_count = 0;


// Initialize semaphore
semaphore = sem_open("/sem_example", O_CREAT, 0666, 1);
if (semaphore == SEM_FAILED) {
    perror("Failed to initialize semaphore");
    return 1;
}


// Create socket
server_fd = socket(AF_INET, SOCK_STREAM, 0);
if (server_fd == -1) {
    perror("Could not create socket");
    return 1;
}

printf("Socket created\n");


// Set SO_REUSEADDR option
int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) {
    perror("setsockopt failed");
    close(server_fd);
    return 1;
}


// Prepare the sockaddr_in structure
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(PORT);


// Bind
if (bind(server_fd, (struct sockaddr *)&server, sizeof(server)) < 0) {
    perror("Bind failed");
    close(server_fd);
    return 1;
}

```

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	10

```

printf("Bind successful\n");

// Listen
if (listen(server_fd, 3) < 0) {
    perror("Listen failed");
    close(server_fd);
    return 1;
}
printf("Server listening on port %d...\n", PORT);

// Create shared memory
int shm_id = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
if (shm_id == -1) {
    perror("Failed to create shared memory");
    return 1;
}

// printf("Shared Memory Key: %#x\n", SHM_KEY); // Print shared memory key in hex
printf("Shared Memory ID: %d\n", shm_id); // Print shared memory ID

char *shared_mem = (char*)shmat(shm_id, NULL, 0);
if (shared_mem == (char*)-1) {
    perror("Failed to attach shared memory");
    return 1;
}

// Accept incoming connections and handle them in separate threads
while ((client_sock = accept(server_fd, (struct sockaddr *)&client, &client_len))) {
    printf("Connection accepted\n");

    // Create a new thread for this client
    new_sock = malloc(sizeof(int));
    *new_sock = client_sock;

    if (pthread_create(&client_threads[client_count], NULL, client_handler, (void*)
new_sock) < 0) {
        perror("Could not create thread");
        return 1;
    }
}

```

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	11

```

        client_count++;

        if (client_count >= MAX_CLIENTS) {
            printf("Maximum clients reached. No longer accepting connections.\n");
            break;
        }
    }

    if (client_sock < 0) {
        perror("Accept failed");
        return 1;
    }

    // Join all client threads before exiting
    for (int i = 0; i < client_count; i++) {
        pthread_join(client_threads[i], NULL);
    }

    // Cleanup
    shmctl(shm_id, IPC_RMID, NULL);
    sem_close(semaphore);
    sem_unlink("/sem_example");

    return 0;
}

```

6.2. Client Code:(client.c)

```

//CLIENT CODE

//client
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_IP "127.0.0.1" // Change this to the server IP address you want to connect
to
#define PORT 9090

```

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	12

```

int main() {
    int sock;

    struct sockaddr_in server;
    char message[256];

    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Socket creation failed");
        return 1;
    }

    // Configure server address
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(SERVER_IP);
    server.sin_port = htons(PORT);

    // Connect to server
    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connection failed");
        return 1;
    }
    printf("Connected to server\n");

    // Send messages to server
    while (1) {
        printf("Enter message (Q to quit): ");
        fgets(message, sizeof(message), stdin);

        // Send message to server
        if (send(sock, message, strlen(message), 0) < 0) {
            perror("Send failed");
            return 1;
        }

        // Quit on 'Q' or 'q'
        if (message[0] == 'Q' || message[0] == 'q') {
            break;
        }
    }
}

```

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	13

```

// Clear the buffer
memset(message, 0, sizeof(message));

// Receive server response
if (recv(sock, message, sizeof(message), 0) < 0) {
    perror("Receive failed");
    return 1;
}

printf("Server response: %s\n", message);
}

// Close socket
close(sock);
printf("Disconnected from server\n");

return 0;
}

```

Server Code Explanation:

Headers and Macros:

1. Includes standard libraries for input/output, memory management, socket programming, threading, and synchronization.
2. Defines constants for the server port, shared memory key, and shared memory size.

Global Variables:

- 1 Declares a pointer to a semaphore for synchronization.

Client Handling Thread (handle_client):

1. Handles client communication in a separate thread.
2. Retrieves client IP address and prints it when a client connects.
3. Accesses shared memory and attaches it.
4. Listens for messages from the client, writes them to shared memory, and echoes the message back to the client.
5. Uses semaphore to synchronize access to shared memory.
6. Closes the client socket when the client disconnects.

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	14

Main Function:

1 IServer Initialization:

Creates a server socket using `socket()`.

Sets the `SO_REUSEADDR` socket option to reuse the address.

Configures the server address structure (`sockaddr_in`) with the `AF_INET` family,

Binds the socket to the specified address and port using `bind()`.

Listens for incoming connections using `listen()`.

2. Shared Memory Setup:

Creates a shared memory segment using `shmget()` with the defined key and size.

Attaches the shared memory segment to the server process's address space using `shmat()`.

3 Client Connection Handling:

Enters a loop to accept incoming client connections using `accept()`.

When a client connects, prints "Connection accepted".

Allocates memory for the new client socket descriptor and creates a new thread to handle the client using `pthread_create()`.

Increments the client count and checks if the maximum number of clients has been reached, stopping further acceptance of connections if the limit is reached.

4 Client Threads Cleanup:

Joins all client threads using `pthread_join()` before exiting the main function to ensure all threads complete their execution.

5 Resource Cleanup:

Removes the shared memory segment using `shmctl()`.

Closes and unlinks the semaphore using `sem_close()` and `sem_unlink()` respectively.

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	15

Client Code Explanation

Headers and Macros:

Includes standard libraries for input/output, memory management, socket programming, and synchronization.

Defines constants for the server port, shared memory key, and shared memory size.

Main Function:

Creates a socket using `socket()`.

Configures the server address structure (`sockaddr_in`) with the `AF_INET` family, the server IP address, and the specified port number.

Connects to the server using `connect()`.

If the connection is successful, it prints "Connected to server".

Enters a loop to continuously prompt the user for input messages.

Reads user input using `fgets()`.

Sends the input message to the server using `send()`.

If the user inputs 'Q' or 'q', the client exits the loop and prepares to disconnect.

Clears the message buffer using `memset()`.

Receives the server's response using `recv()` and prints it.

7. User manual

This user manual provides instructions for compiling, running, and testing the server and client applications. These applications demonstrate the use of shared memory, semaphores, and threading in a networked client-server setup.

Prerequisites

- A Linux-based operating system (or any OS with shared memory and semaphore support).
- GCC compiler.
- Basic knowledge of using the terminal/command prompt.

File Description

- **server.c:** Source code for the server application.
- **client.c:** Source code for the client application.

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	16

Compiling Application:

- Open a terminal window.
- Navigate to the directory containing the source code files.
- Compile the server application:
 - `gcc -o server server.c`

This command compiles server.c and creates an executable named server.

- Compile the client application:
 - `gcc -o client client.c`

This command compiles client.c and creates an executable named client.

Running Application:

1. Running the Server

- Start the server application:
 - `./server`

The server will start and listen for incoming client connections on port 9999. It will also start a logging thread that writes shared memory updates to shared_memory_log.txt.

2. Running the Client

- Open another terminal window.
- Start the client application:
 - `./client`

The client will attempt to connect to the server running on 127.0.0.1 (localhost) on port 9999. Once connected, the client can send messages to the server.

Testing the Applications

Single Client Test

1. Start the server:

- `./server`

2. Run a single client instance:

- `./client`

3. Send a message from the client to the server.
4. Verify that the server receives the message and sends a response.
5. Check the shared memory contents from the client.

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	17

Multiple Client Test

1. Start the server:

- `./server`

2. Use the provided script to run multiple client instances:

- `./run_clients.sh`

```
#!/bin/bash
```

```
# Define the number of clients to run
num_clients=50
```

```
# Loop to run clients
```

```
for ((i=1; i<=$num_clients; i++))
do
```

```
    # Command to run your client, replace this with your actual client command
```

```
    # Example: ./client_program &
```

```
    ./client_p&
```

```
    # Optional: Add sleep to stagger the start of each client
```

```
    # sleep 1
```

```
done
```

```
# Optional: Wait for all clients to finish
```

```
wait
```

3. Verify that the server handles all client connections concurrently.
4. Check the shared memory contents from each client.

8.The Synchronization Mechanism

The server and client applications make use of several synchronization mechanisms to ensure correct and safe access to shared resources, particularly shared memory. The primary synchronization mechanisms used are semaphores and mutexes. Below is a detailed description of each mechanism and its role in the application.

Semaphores

Semaphores are used extensively in the server and client applications to manage concurrent access to shared resources. There are two main semaphores used:

- Shared Memory Semaphore (semaphore)
- Client Count Semaphore (client_semaphore)

Shared Memory Semaphore (semaphore)

- Purpose: To synchronize access to the shared memory segment.
- Type: semaphore.
- Initialization:

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	18

```
semaphore = sem_open("/sem_example", O_CREAT, 0666, 1);
```

The semaphore is initialized with a value of 1, indicating that updates to the client count are allowed.

➤ Usage:

Acquiring the Semaphore (Wait/Lock):

```
sem_wait(&client_semaphore);
```

When the server needs to increment the “client_count”, it calls “sem_wait(&client_semaphore)”. This ensures that only one thread can update the count at a time.

Releasing the Semaphore (Post/Unlock):

```
sem_post(&client_semaphore);
```

After updating the “client_count”, the thread calls “sem_post(&client_semaphore)”, allowing other threads to access the client count.

Scenarios of Usage:

Server: When a new client connects, the server increments the “client_coun”t.

Thread Synchronization

In addition to semaphores, the application relies on thread synchronization techniques to manage multiple threads:

Client Handling Threads: Each client connection is handled by a separate thread created using “pthread_create”. This allows the server to handle multiple clients concurrently.

9. Test Cases and Results

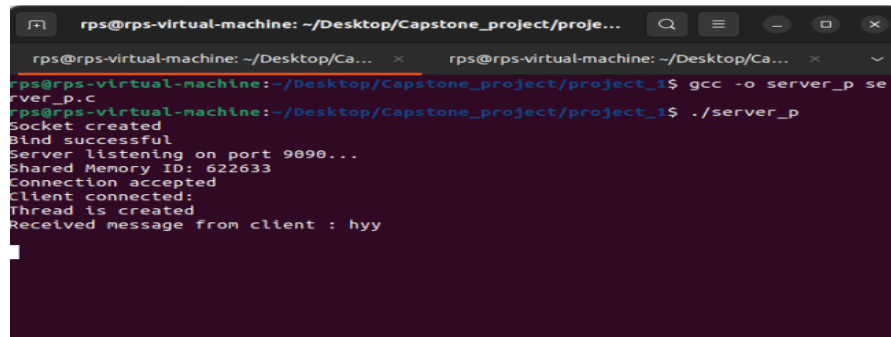
Test Case 1: Single Client Communication

- **Test Steps:**
 1. Start the server.
 2. Run a single instance of the client application.

```
rps@rps-virtual-machine: ~/Desktop/Capstone_project/proje...
rps@rps-virtual-machine: ~/Desktop/Ca... x rps@rps-virtual-machine: ~/Desktop/Ca... x
rps@rps-virtual-machine:~/Desktop/Capstone_project/project_1$ gcc -o server_p se
rver_p.c
rps@rps-virtual-machine:~/Desktop/Capstone_project/project_1$ ./server_p
Socket created
bind successful
Server listening on port 9090...
Shared Memory ID: 622633
connection accepted
Client connected:
Thread is created
Received message from client : hyy
```

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	19

3. Send a message from the client to the server.
4. Verify that the server receives the message and sends a response.
5. Check the shared memory contents from the client.



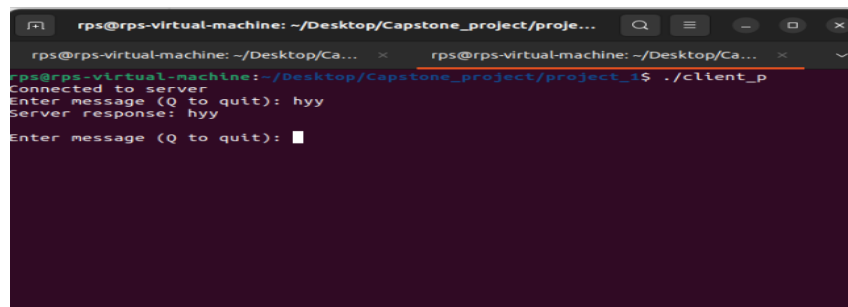
```

rps@rps-virtual-machine: ~/Desktop/Capstone_project/proje...
rps@rps-virtual-machine: ~/Desktop/Ca... x rps@rps-virtual-machine: ~/Desktop/Ca... x
rps@rps-virtual-machine:~/Desktop/Capstone_project/project_1$ gcc -o server_p se
rver_p.c
rps@rps-virtual-machine:~/Desktop/Capstone_project/project_1$ ./server_p
Socket created
Bind successful
Server listening on port 9090...
Shared Memory ID: 622633
Connection accepted
Client connected:
Thread is created
Received message from client : hyy

```

- **Expected Result:**

- The server successfully receives the message and sends a response.
- The shared memory contains the exchanged messages.



```

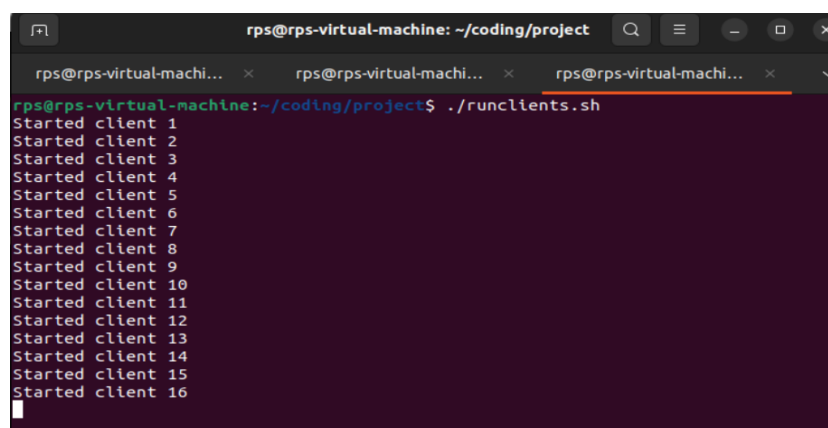
rps@rps-virtual-machine: ~/Desktop/Capstone_project/proje...
rps@rps-virtual-machine: ~/Desktop/Ca... x rps@rps-virtual-machine: ~/Desktop/Ca... x
rps@rps-virtual-machine:~/Desktop/Capstone_project/project_1$ ./client_p
Connected to server
Enter message (Q to quit): hyy
Server response: hyy
Enter message (Q to quit): 

```

Test Case 2: Multiple Client Communication

- **Test Steps:**

1. Start the server.
2. Run multiple instances of the client application concurrently using the provided script.



```

rps@rps-virtual-machine: ~/coding/project
rps@rps-virtual-machi... x rps@rps-virtual-machi... x rps@rps-virtual-machi... x
rps@rps-virtual-machine:~/coding/project$ ./runclients.sh
Started client 1
Started client 2
Started client 3
Started client 4
Started client 5
Started client 6
Started client 7
Started client 8
Started client 9
Started client 10
Started client 11
Started client 12
Started client 13
Started client 14
Started client 15
Started client 16

```

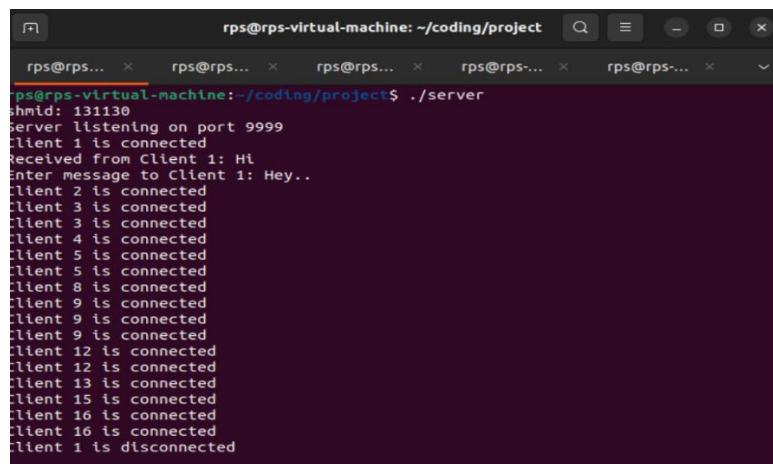
Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	20

3. Each client sends messages to the server.
4. Verify that the server handles all client connections concurrently.

5. Check the shared memory contents from each client.

- **Expected Result:**

- The server handles multiple client connections concurrently without errors.



```

rps@rps-virtual-machine: ~/coding/project
rps@rps-virtual-machine:~/coding/project$ ./server
shmid: 131130
server listening on port 9999
Client 1 is connected
Received from Client 1: Hi
Enter message to Client 1: Hey..
Client 2 is connected
Client 3 is connected
Client 3 is connected
Client 4 is connected
Client 5 is connected
Client 5 is connected
Client 8 is connected
Client 9 is connected
Client 9 is connected
Client 9 is connected
Client 12 is connected
Client 12 is connected
Client 13 is connected
Client 15 is connected
Client 16 is connected
Client 16 is connected
Client 1 is disconnected

```

10. Conclusion

In conclusion, this project demonstrates the implementation of a client-server system using shared memory and semaphores for inter-process communication and

Doc Id	Language	Version	Author	Date	Page No.
DOC/1	English	1.0	Ayushi	June 4	21

synchronization. The server efficiently handles multiple client connections . The client interacts with the server, sends messages, and displays responses along with shared memory contents. The use of shared memory and semaphores ensures reliable and synchronized communication between processes