

A
Project Report on
LINUX MULTI-THREADED CLIENT-SERVER
USING SHARED MEMORY

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	2

Contents

1. Abstract	3
2. Introduction.....	4
3. Project Scope.....	5
4. Requirements	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
8.The Synchronization Mechanism	17
9. Test Cases and Results	18
Test Case 1: Single Client Communication.....	18
Test Case 2: Multiple Client Communication	19
10. Conclusion	20

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	3

1. Abstract

This project involves developing a client-server application that uses shared memory and semaphores for inter-process communication. The server handles multiple clients concurrently, logging their interactions to a shared memory segment. Clients can send messages to the server and receive responses, with both the client and server able to view the current state of the shared memory. This setup demonstrates inter-process communication mechanisms, particularly shared memory and semaphores to manage access and synchronization between multiple processes.

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	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.

A unique feature of the server is its logging capability: the server logs the contents of the shared memory to a file whenever an update occurs, providing a persistent record of all communications. This is accomplished using a dedicated logging thread that monitors the shared memory for changes.

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.
ReqDOC/1	english	2.0	Saranya	May 2	5

3. Project Scope

The scope of the project includes:

- Implementing a server application that listens for incoming connections and handles client messages.
- Developing a client application that connects to the server, sends messages, and displays responses.
- Utilizing shared memory for storing messages exchanged between the server and clients.
- Using semaphores for synchronizing access to shared memory to avoid race conditions

4. Requirements

4.1. Functional Requirements:

1. Server Application:

- **Initialize Shared Memory:** Set up a shared memory segment for storing messages.
- **Initialize Semaphores:** Set up semaphores to synchronize access to shared memory.
- **Accept Connections:** Listen for and accept incoming client connections.
- **Handle Clients:** Create a new thread for each client to handle communication.
- **Update Shared Memory:** Write client messages and server responses to shared memory.
- **Log Messages:** Continuously log the contents of shared memory to a file.
- **Close Connections:** Cleanly shut down client connections and free resources.

2. Client Application:

- **Connect to Server:** Establish a connection with the server.
- **Send Messages:** Send user-input messages to the server.
- **Receive Responses:** Receive and display responses from the server.
- **Display Shared Memory:** Show the current contents of shared memory after each interaction.
- **Exit:** Allow the client to exit gracefully.

4.2. Non-Functional Requirements

1. Performance:

- 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.

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	6

- 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.

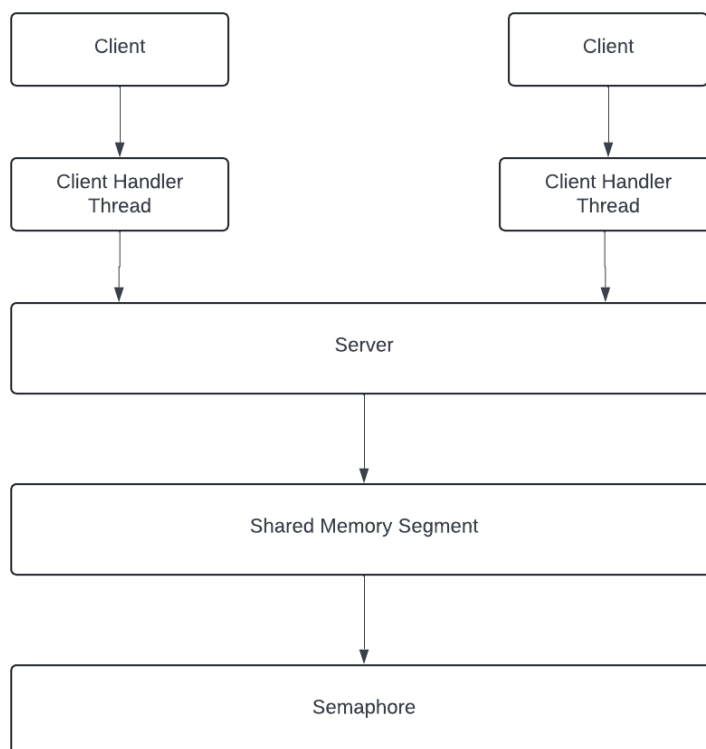
4. Scalability:

- 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.
ReqDOC/1	english	2.0	Saranya	May 2	7

6. Code comments and Explanation:

6.1. Server code:(server.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <time.h>

#define PORT 9999      // Port number for server to listen on
#define SHM_KEY 1234   // Key for shared memory
#define SHM_SIZE 1024  // Size of shared memory

// Shared resource
char *shared_data;
sem_t *semaphore;
sem_t client_semaphore;
int shared_data_updated = 0; // Flag to indicate shared memory update
int client_count = 0;       // Count of connected clients

// Thread function for logging shared memory updates to a file
void *log_shared_memory(void *arg) {
    FILE *log_file = fopen("shared_memory_log.txt", "a");
    if (!log_file) {
        perror("fopen");
        return NULL;
    }
}

```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	8

```

while (1) {
    // Check if shared memory has been updated
    sem_wait(semaphore);
    if (shared_data_updated) {
        char data[SHM_SIZE];
        strncpy(data, shared_data, SHM_SIZE);
        shared_data_updated = 0;
        sem_post(semaphore);
        // Get current time for logging
        time_t now = time(NULL);
        struct tm *t = localtime(&now);

        // Log the data with timestamp
        fprintf(log_file, "[%04d-%02d-%02d %02d:%02d:%02d] %s\n",

                t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec, data);
        fflush(log_file);
    } else {
        sem_post(semaphore);
        sleep(1); // Sleep for a while before checking again
    }
}

fclose(log_file);
return NULL;
}

// Thread function for handling individual clients
void *handle_client(void *arg) {
    int client_socket = *(int *)arg;
    free(arg);

    // Update client count safely
    sem_wait(&client_semaphore);
    int client_number = ++client_count;
    sem_post(&client_semaphore);

```


Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	9

```

char buffer[256];
char response[256];
while (1) {
    memset(buffer, 0, 256);
    int n = read(client_socket, buffer, 255);
    if (n <= 0) {
        break;
    }

    // Update shared memory with client message
    sem_wait(semaphore);
    snprintf(shared_data, SHM_SIZE, "Client %d: %s", client_number, buffer);
    shared_data_updated = 1;
    sem_post(semaphore);

    printf("Received from Client %d: %s", client_number, buffer);

    // Server prompts for a response
    printf("Enter message to Client %d: ", client_number);
    memset(response, 0, 256);
    fgets(response, 255, stdin);

    // Update shared memory with server response
    sem_wait(semaphore);
    snprintf(shared_data, SHM_SIZE, "Server to Client %d: %s", client_number,
response);
    shared_data_updated = 1;
    sem_post(semaphore);

    n = write(client_socket, response, strlen(response));
    if (n < 0) {
        perror("ERROR writing to socket");
    }
}
printf("Client %d is disconnected\n", client_number);
close(client_socket);
return NULL;
}

```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	10

```

int main() {
    int server_socket, client_socket, *new_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len;
    pthread_t thread_id, log_thread_id;
    // Initialize shared memory
    int shm_id = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shm_id == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    printf("shmid: %d\n", shm_id);
    shared_data = (char *)shmat(shm_id, NULL, 0);
    if (shared_data == (char *)-1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    // Initialize semaphores
    semaphore = sem_open("/my_semaphore", O_CREAT, 0666, 1);
    if (semaphore == SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }
    sem_init(&client_semaphore, 0, 1);
    // Create socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("ERROR opening socket");
        exit(EXIT_FAILURE);
    }
    // Prepare server address structure
    memset((char *)&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)) < 0) {
        perror("ERROR on binding");
    }
}

```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	11

```

        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    listen(server_socket, 5);
    printf("Server listening on port %d\n", PORT);

    // Start logging thread
    if (pthread_create(&log_thread_id, NULL, log_shared_memory, NULL) < 0) {
        perror("could not create logging thread");
        exit(EXIT_FAILURE);
    }

    // Accept incoming connections
    while (1) {
        client_len = sizeof(client_addr);
        client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_len);
        if (client_socket < 0) {
            perror("ERROR on accept");
            exit(EXIT_FAILURE);
        }
        printf("Client %d is connected\n", client_count + 1);
        new_sock = malloc(sizeof(int));
        *new_sock = client_socket;
        if (pthread_create(&thread_id, NULL, handle_client, (void *)new_sock) < 0) {
            perror("could not create thread");
            exit(EXIT_FAILURE);
        }
    }

    close(server_socket);
    shmctl(shm_id, IPC_RMID, NULL);
    sem_close(semaphore);
    sem_unlink("/my_semaphore");
    sem_destroy(&client_semaphore);
    return 0;
}

```

6.2. Client Code:(client.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	12

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
#include <fcntl.h>
```

```
#include <sys/shm.h>
```

```
#include <arpa/inet.h>
```

```
#include <semaphore.h>
```

```
#define PORT 9999    // Port number for server connection
```

```
#define SHM_KEY 1234 // Key for shared memory
```

```
#define SHM_SIZE 1024 // Size of shared memory
```

```
int main() {
```

```
    int sockfd, n;
```

```
    struct sockaddr_in server_addr;
```

```
    char buffer[256];
```

```
    // Initialize shared memory
```

```
    int shm_id = shmget(SHM_KEY, SHM_SIZE, 0666);
```

```
    if (shm_id == -1) {
```

```
        perror("shmget");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    char *shared_data = (char *)shmat(shm_id, NULL, 0);
```

```
    if (shared_data == (char *)-1) {
```

```
        perror("shmat");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    // Initialize semaphore
```

```
    sem_t *semaphore = sem_open("/my_semaphore", 0);
```

```
    if (semaphore == SEM_FAILED) {
```

```
        perror("sem_open");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    // Create socket
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if (sockfd < 0) {
```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	13

```

    perror("ERROR opening socket");
    exit(EXIT_FAILURE);
}

// Prepare server address structure
memset((char *)&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
if (inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr) <= 0) {
    perror("ERROR invalid address");
    exit(EXIT_FAILURE);
}

// Connect to server
if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("ERROR connecting");
    exit(EXIT_FAILURE);
}

while (1) {
    printf("Enter message to server: ");
    memset(buffer, 0, 256);
    fgets(buffer, 255, stdin);
    if (strncmp(buffer, "exit", 4) == 0) {
        break;
    }

    n = write(sockfd, buffer, strlen(buffer));
    if (n < 0) {
        perror("ERROR writing to socket");
    }

    memset(buffer, 0, 256);
    n = read(sockfd, buffer, 255);
    if (n < 0) {
        perror("ERROR reading from socket");
    }

    printf("Received from server: %s", buffer);

    // Print shared memory contents after receiving the response

```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	14

```

sem_wait(semaphore);

printf("Shared memory contains: %s\n", shared_data);

sem_post(semaphore);
}

close(sockfd);
return 0;
}

```

Server Code Explanation:

Headers and Macros:

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

Global Variables:

- shared_data: Pointer to the shared memory segment.
- semaphore: Semaphore to control access to shared memory.
- client_semaphore: Semaphore to control client count updates.
- shared_data_updated: Flag to indicate if shared memory has been updated.
- client_count: Tracks the number of connected clients.

Logging Thread (log_shared_memory):

- Opens a file to log shared memory updates.
- Continuously checks if the shared memory has been updated.
- Logs the updated shared memory content with a timestamp.
- Uses semaphores to ensure exclusive access to shared memory.

Client Handling Thread (handle_client):

- Handles communication with a connected client.
- Updates the shared memory with messages from the client.
- Prompts the server user for a response and sends it to the client.
- Uses semaphores to ensure exclusive access to shared memory and to update the client count.

Main Function:

- Initializes shared memory and semaphores.
- Creates a socket, binds it to the specified port, and listens for incoming connections.

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	15

- Starts the logging thread.
- Accepts incoming client connections and creates a new thread to handle each client.

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:

- Initializes shared memory and semaphore.
- Creates a socket and connects it to the server.
- Enters a loop to send messages to the server and receive responses.
- After each response from the server, prints the current content of the shared memory.
- Closes the socket and terminates when the user types "exit".

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 POSIX 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.
ReqDOC/1	english	2.0	Saranya	May 2	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.

Multiple Client Test

1. Start the server:

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	17

- ./server

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

- ./run_clients.sh

```
#!/bin/bash

# Number of clients to run
NUM_CLIENTS=16

# Command to run the client executable
CLIENT_CMD="./client"

# Function to run a client
run_client() {
    $CLIENT_CMD &
}

# Run the specified number of clients
for ((i=1; i<=NUM_CLIENTS; i++))
do
    run_client
    echo "Started client $i"
done

# Wait for all background processes to finish
```

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: POSIX named semaphore.
- Initialization:

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

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	18

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:

1. **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.
2. **Logging Thread:** A separate thread is created to continuously log updates to the shared memory. This thread checks if the shared memory has been updated and logs the content if necessary.

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.

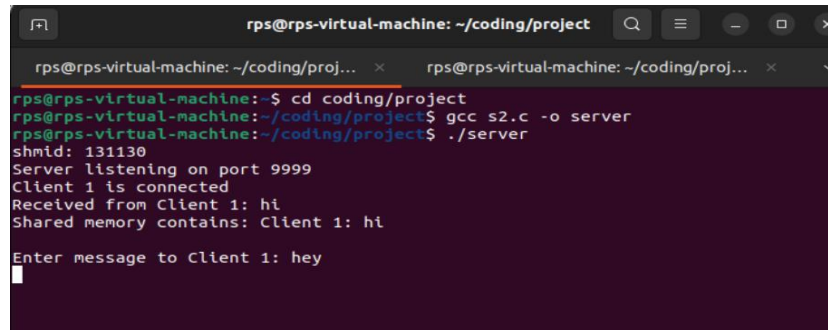
```

rps@rps-virtual-machine: ~/coding/project
rps@rps-virtual-machine: ~/coding/proj... x  rps@rps-virtual-machine: ~/coding/proj... x
rps@rps-virtual-machine:~/coding/project$ gcc s2.c -o server
rps@rps-virtual-machine:~/coding/project$ ./server
shmid: 131130
server listening on port 9999
client 1 is connected

```

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	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.



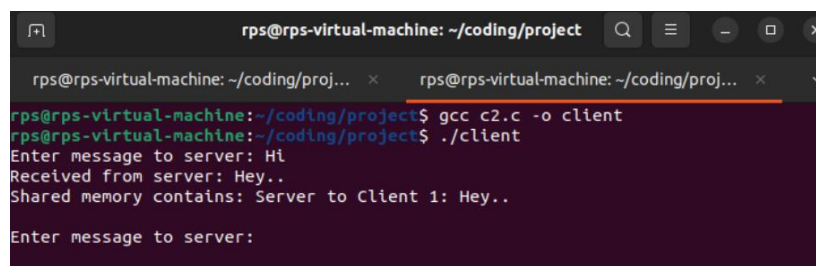
```

rps@rps-virtual-machine: ~/coding/project
rps@rps-virtual-machine: ~/coding/project$ cd coding/project
rps@rps-virtual-machine: ~/coding/project$ gcc s2.c -o server
rps@rps-virtual-machine: ~/coding/project$ ./server
shmid: 131130
Server listening on port 9999
Client 1 is connected
Received from Client 1: hi
Shared memory contains: Client 1: hi
Enter message to Client 1: hey

```

- **Expected Result:**

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



```

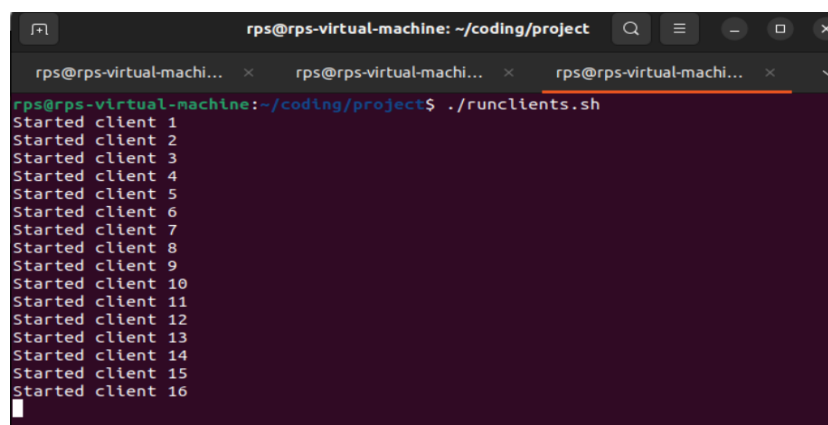
rps@rps-virtual-machine: ~/coding/project
rps@rps-virtual-machine: ~/coding/project$ gcc c2.c -o client
rps@rps-virtual-machine: ~/coding/project$ ./client
Enter message to server: Hi
Received from server: Hey..
Shared memory contains: Server to Client 1: Hey..
Enter message to server:

```

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-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

```

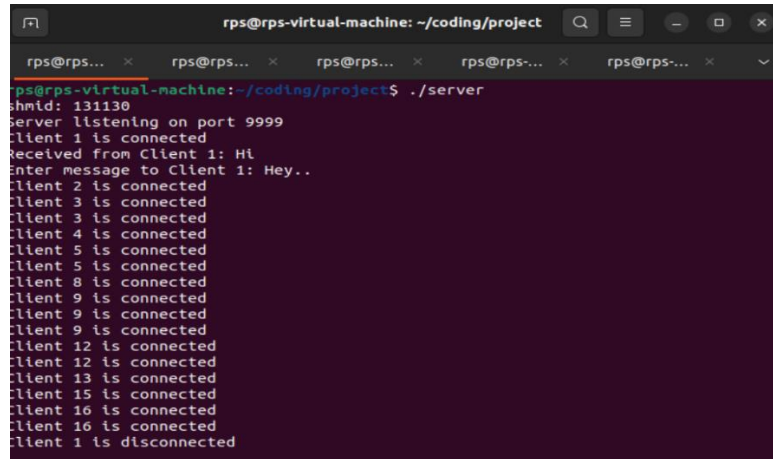
3. Each client sends messages to the server.
4. Verify that the server handles all client connections concurrently.

Doc Id	Language	Version	Author	Date	Page No.
ReqDOC/1	english	2.0	Saranya	May 2	20

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... x rps@rps... x rps@rps... x rps@rps... x rps@rps... x
rps@rps-virtual-machine:~/coding/project$ ./server
shmid: 131130
server listening on port 9999
Client 1 is connected
Received from Client 1: Hl
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 synchronization. The server efficiently handles multiple client connections and logs communication activities. 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.