Server Client Programming using Socket and TCP/IP

A PROJECT REPORT

Submitted by

SIDDHARTH
2100290140134

Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Ms. Shalika
Assistant Professor

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206
(JUNE 2023)

CERTIFICATE

Certified that **Siddharth 2100290140134** has/have carried out the project work having "**Server Client Programming Using Socket And TCP/IP"** for Master of Computer Applications from Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Technical University, Lucknow under my supervision. The project report embodies original work, and studies are carried out by the student himself / herself and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Date:

Siddharth - 2100290140134

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date:

Ms. Shalika
Assistant Professor
Department of Computer Applications
KIET Group of Institutions, Ghaziabad

Signature of Internal Examiner **Signature of External Examiner**

Dr. Arun Tripathi
Head, Department of Computer Applications
KIET Group of Institutions, Ghaziabad

ABSTRACT

The project aims to develop a client-server communication system using sockets and TCP/IP protocols. The system allows multiple clients to connect to a server and exchange data in real-time. The project involves designing and implementing a server application that listens for incoming client connections and manages the communication between clients. The clients can connect to the server using their IP address and port number, and the server can authenticate and authorize clients before allowing them to send or receive data. The system employs socket programming to establish reliable connections and TCP/IP protocol for error-free and ordered data transfer. The project also involves testing the system's performance and scalability under different conditions to ensure its effectiveness and efficiency in handling large-scale communication. The resulting system provides a robust and flexible communication platform that can be used in various domains, including gaming, messaging, and data streaming applications.

C and C++ Programming languages can be used to implement a client-server communication system using sockets and TCP/IP protocols.C and C++ are low-level languages that offer direct access to the operating system's networking capabilities. They are well-suited for developing high-performance, low-latency network applications.

ACKNOWLEDGEMENTS

Success in life is never attained single handedly. My deepest gratitude goes to my thesis supervisor, **Ms. Shalika** for her guidance, help and encouragement throughout my research work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to **Dr. Arun Tripathi**, Professor and Head, Department of Computer Applications, for his insightful comments and administrative help at various occasions.

Fortunately, I have many understanding friends, who have helped me a lot in many critical conditions.

Finally, my sincere thanks go to all my faculty members & also my family members and all those who have directly and indirectly provided me moral support and other kinds of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

Siddharth

TABLE OF CONTENTS

LIST OF CHAPTERS

LIST OF FIGURES

Chapter 1

Introduction

The Client-Server Communication using Sockets and TCP/IP protocol is a commonly used technique in computer networking to facilitate communication between two different programs running on different machines. This project aims to develop a client-server application using C programming language, which will allow two separate programs to exchange data over a network using sockets and the TCP/IP protocol.

The client-server model is a widely used distributed application architecture that enables two or more computers to communicate over a network. In this model, one computer acts as a server that provides services, and other computers, known as clients, request these services. The communication between the client and the server is done using sockets, which are endpoints for communication between two processes on a network.

This project will use C programming language, a popular choice for system-level programming due to its low-level control and efficient memory management. The TCP/IP protocol is used for data transmission over a network, providing reliable, ordered, and error-checked delivery of data packets.

The project will involve designing and implementing a client-server application that allows the client to send requests to the server and receive responses. The client will establish a connection to the server using sockets, send data to the server, and wait for a response. The server will listen for incoming connections, receive requests from the client, process them, and send a response back.

Overall, this project will provide hands-on experience in developing a client-server application using sockets and the TCP/IP protocol in C programming language.

1.1 Project Description

The project aims to develop a robust client-server communication system utilizing Inter-Process Communication (IPC) techniques. IPC enables the exchange of information and data between processes running on the same or different machines, facilitating seamless communication and collaboration. The client-server model provides a scalable and efficient architecture where clients request services from a central server, which processes and responds to these requests. By implementing IPC mechanisms such as pipes, sockets, or shared memory, the client and server can establish reliable communication channels. This project will focus on designing and implementing a secure and efficient IPC-based communication system, ensuring data integrity, authentication, and confidentiality. The solution will enable seamless interaction between clients and the server, allowing efficient resource sharing and enabling the development of robust distributed applications.

1.2 Project Scope

The project scope entails designing and implementing a client-server communication system. The objective is to establish a reliable and efficient communication channel between clients and servers, allowing for seamless data exchange and collaboration. The system will support multiple clients connecting to a central server, enabling them to transmit requests and receive responses in a secure and synchronized manner. The scope encompasses the development of robust protocols and mechanisms to handle various types of data, such as text, multimedia, and real-time updates. Additionally, the project includes implementing security measures, such as authentication and encryption, to ensure data privacy and integrity during transmission. The final deliverables will consist of a fully functional client-server communication system that meets the specified requirements and facilitates smooth and reliable communication between the clients and servers.

1.3 Hardware & Software Requirement:

1.3.1 Hardware Interfaces Requirement

- Processor: i3 11th Gen
- RAM: 8 GB
- Operating System: Linux
- System Type: 64-bit operating system
- Hard disk: 256 GB

1.3.2 Software Interfaces Requirement

- Linux terminal
- gcc compiler

Chapter 2

Feasibility Study

2.1 Operational Study

This operational study for client-server communication using IPC socket involves a detailed examination of the underlying processes, mechanisms, and protocols that facilitate the seamless exchange of data between a client and a server. IPC, or Inter-Process Communication, provides a means for processes running on the same or different machines to communicate with each other.

The client-server model is a common architecture where a client process requests services or resources from a server process, which then responds accordingly. In this context, IPC sockets serve as the communication channel that enables the interaction between the client and server. These sockets allow for bidirectional data flow, allowing both the client and server to send and receive information.

To ensure efficient communication, it is crucial to understand the operational aspects of IPC sockets. This includes the establishment of connections, message passing, error handling, and synchronization mechanisms. The study delves into the different types of IPC sockets, such as stream sockets (TCP) and datagram sockets (UDP), and their respective advantages and trade-offs.

Moreover, the study explores the implementation details of client-server communication using IPC sockets, including the steps involved in establishing a connection, sending and receiving data, and terminating the connection gracefully. It analyzes the various socket functions and system calls involved in the process, such as socket(), bind(), listen(), accept(), connect(), send(), and receive().

In addition to the technical aspects, the operational study also considers factors like performance, scalability, and security. It evaluates the impact of network latency, bandwidth limitations, and the number of concurrent clients on the overall system performance. It also addresses security concerns, such as data encryption and authentication, to ensure the confidentiality and integrity of the communication.

By conducting a comprehensive operational study, clients and servers can optimize their communication processes, identify potential bottlenecks, and implement strategies to enhance the overall performance and reliability of their systems. This study serves as a foundation for designing and implementing robust client-server architectures that leverage IPC sockets effectively.

2.2 Technical Feasibility

Technical feasibility for client-server communication using IPC sockets is high due to several reasons. IPC sockets, also known as inter-process communication sockets, provide a reliable and efficient means of communication between processes running on the same machine.

Firstly, IPC sockets are supported by most modern operating systems, making them a widely accepted and standardized method for inter-process communication. This ensures compatibility and ease of implementation across different platforms.

Secondly, IPC sockets offer high performance and low latency. They utilize shared memory or local loopback network interfaces, which are typically faster than traditional network sockets. This allows for efficient data transfer and real-time communication between the client and server processes.

Furthermore, IPC sockets provide a robust and secure communication channel. They offer various mechanisms for authentication and encryption, ensuring the confidentiality and integrity of data transmitted between the client and server. Additionally, operating systems often provide access control mechanisms to restrict access to IPC sockets, enhancing the overall security of the communication.

Moreover, IPC sockets support different communication protocols, such as TCP/IP or UDP, allowing developers to choose the most suitable protocol based on their specific requirements. This flexibility enables the implementation of various types of client-server applications, ranging from simple request-response models to more complex event-driven architectures.

In conclusion, client-server communication using IPC sockets is technically feasible and offers numerous advantages, including compatibility, high performance, security, and flexibility. By leveraging IPC sockets, developers can establish efficient and reliable communication channels between client and server processes, facilitating the exchange of data and enabling the development of robust applications.

2.3 Economical Feasibility

Economical feasibility of a system examines whether the finance is available for implementing the new system and whether the money spent is recoverable.

The cost involved is in designing and developing a good investment for the organization.

Thus, hardware requirements used for the proposed system are very standard. Moreover, by making use of the proposed system to carry out the work speedily will increase and also saves the valuable time of an organization.

In the proposed system the finance is highly required for the installation of the software which can also be recovered by implementing a better system.

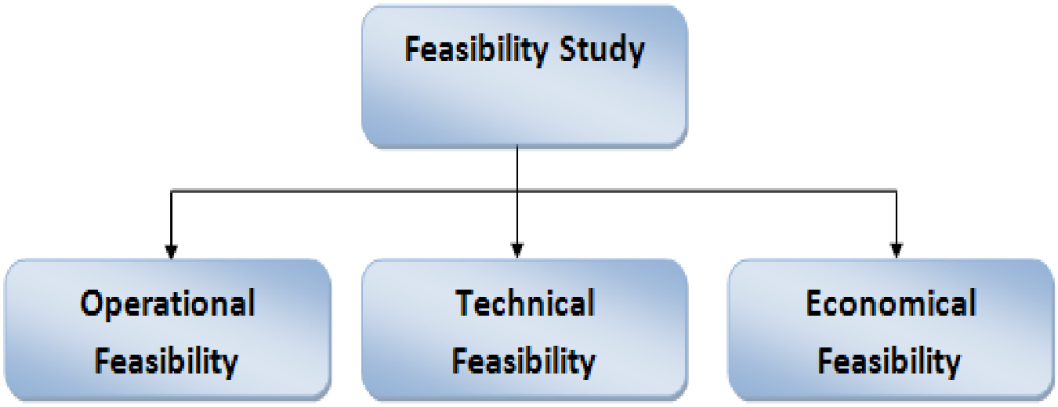

Fig: - 2.1

Chapter – 3

CODING

3.1 SERVER MODULE

3.1.1 createInfra.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"
#include "declarations.h"
void* createInfra(void *arg)
{
	int ret;
	Infra *infra;
#ifdef DEBUG
	printf("%s: Begin.\n",__func__);
#endif
	infra = (Infra*)malloc(sizeof(Infra));
	if(!infra)
	{
		perror("malloc");
		(*fptr[0])((void*)"failure"); // exitProgram()
	}

	infra->pipe = (int*)malloc(sizeof(int)*2);
	if(!infra->pipe)
	{
		perror("malloc");
		free(infra);
		(*fptr[0] )((void*)"failure"); // exitProgram()
	}
```

```
infra->fifoName = (char*)malloc(sizeof(char)*20);
if(!infra->fifoName)
{
        perror("malloc");
        free(infra->pipe);
        free(infra);
        (*fptr[0] )((void*)"failure"); // exitProgram()
}
memset(infra->fifoName,'\0',20);

infra->smKey = infra->mqKey = 0;

ret = pipe(infra->pipe);
if(ret == -1)
{
        perror("pipe");
        free(infra->pipe);
        free(infra->fifoName);
        free(infra);
        (*fptr[0])((void*)"failure"); // exitProgram()
}

ret = mkfifo(FIFONAME, 666);
if(ret == -1)
{
        perror("mkfifo");
        free(infra->pipe);
        free(infra->fifoName);
        free(infra);
        (*fptr[0])((void*)"failure"); // exitProgram()
}
```

```
infra->smKey = shmget((key_t)KEY_SHM, sizeof(Result), IPC_CREAT|0666);
if(infra->smKey == -1)
{
	perror("shmget");
	free(infra->pipe);
	free(infra->fifoName);
	free(infra);
	(*fptr[0])((void*)"failure"); // exitProgram()
}

infra->smptr = shmat(infra->smKey, NULL, 0);
if(!infra->smptr)
{
	perror("shmat");
	free(infra->pipe);
	free(infra->fifoName);
	free(infra);
	(*fptr[0])((void*)"failure"); // exitProgram()
}

infra->mqKey = msgget((key_t)KEY_MQ, IPC_CREAT|0666);
if(infra->mqKey == -1)
{
	perror("msgget");
	free(infra->pipe);
	free(infra->fifoName);
	free(infra);
	(*fptr[0])((void*)"failure"); // exitProgram()
}

ret = sem_init(&infra->tsem, 0, 1);
if(ret == -1)
```

```
{
        perror("sem_init");
        free(infra->pipe);
        free(infra->fifoName);
        free(infra);
        (*fptr[0])((void*)"failure"); // exitProgram()
}


infra->smKey1 = shmget((key_t)KEY_SHM1, sizeof(sem_t), IPC_CREAT|0666);
if(infra->smKey1 == -1)
{
        perror("shmget");
        free(infra->pipe);
        free(infra->fifoName);
        free(infra);
        (*fptr[0])((void*)"failure"); // exitProgram()
}


infra->smptr1 = shmat(infra->smKey1, NULL, 0);
if(!infra->smptr1)
{
        perror("shmat");
        free(infra->pipe);
        free(infra->fifoName);
        free(infra);
        (*fptr[0])((void*)"failure"); // exitProgram()
}

ret = sem_init((sem_t*)&infra->smptr1, 1, 1);
if(ret == -1)
```

```
        {
                perror("sem_init");
                free(infra->pipe);
                free(infra->fifoName);
                free(infra);
                (*fptr[0])((void*)"failure"); // exitProgram()
        }

#ifdef DEBUG
        printf("%s: End.\n",__func__) ;
#endif
        return (void*)infra;
}
```

3.1.2 declarations.h

```
#define NOF 4

int init();

extern void* exitProgram(void*);
extern void* createInfra(void*);
extern void* processRequest(void*);
extern void* readResult(void*);

extern void* (*fptr[NOF])(void*);
```

3.1.3 exitProgram.c

```
#include "../Common/headers.h"
#include "declarations.h"

void* exitProgram(void* arg)
{
#ifdef PRINT
        printf("%s: Begin.\n", __func__);
        printf("%s: arg:%s\n", __func__, (char*)arg);
#endif

        if(strncmp((char*)arg, "success", 7) == 0 )
                exit(EXIT_SUCCESS);

        if(strncmp((char*)arg, "failure", 7) == 0 )
                exit(EXIT_FAILURE);

#ifdef PRINT
        printf("%s: End.\n", __func__);
#endif
        return 0;
}
```

3.1.4 init.c

```
#include "../Common/headers.h"
#include "declarations.h"

void* exitProgram(void*);
void* createInfra(void*);
void* processRequest(void*);
```

```
void* readResult(void*);

void* (*fptr[NOF])(void*);

int init()
{
#ifdef DEBUG
        printf("%s: Begin.\n",__func__);
#endif

        fptr[0]=exitProgram;
        fptr[1]=createInfra;
        fptr[2]=processRequest;
        fptr[3]=readResult;

#ifdef DEBUG
        printf("%s: End.\n",__func__) ;
#endif
        return 0;
}
```

3.1.5 Makefile

```
#! /bin/bash

DF=-DDEBUG

server: server.o init.o exitProgram.o createInfra.o processRequest.o readResult.o
        gcc -o server server.o init.o exitProgram.o createInfra.o processRequest.o
readResult.o
```

```
readResult.o: readResult.c ../Common/headers.h ../Common/dataStruct.h declarations.h
	gcc -c ${DF} readResult.c

processRequest.o: processRequest.c ../Common/headers.h ../Common/dataStruct.h
declarations.h
	gcc -c ${DF} processRequest.c

createInfra.o: createInfra.c ../Common/headers.h ../Common/dataStruct.h declarations.h
	gcc -c ${DF} createInfra.c

exitProgram.o: exitProgram.c ../Common/headers.h declarations.h
	gcc -c ${DF} exitProgram.c
init.o: init.c ../Common/headers.h declarations.h
	gcc -c ${DF} init.c
server.o: server.c ../Common/headers.h ../Common/dataStruct.h declarations.h
	gcc -c ${DF} server.c

clean:
	-rm *.o
```

3.1.6 processRequest.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"
#include "declarations.h"

void* processRequest(void *arg)
{
	char prfd[4];
	int fd,ret,fret;
	Infra *infra;
	Request r;
```

```
        Result *res;

#ifdef DEBUG
        printf("%s: Begin.\n",__func__);
#endif

        infra= (Infra*)arg;
        fd = open(FIFONAME, O_RDONLY);
        if(fd == -1)
        {
                perror("open");
                (*fptr[0])((void*)"failure"); // exitProgram()
        }
#ifdef DEBUG
        printf("%s: Request FIFO Opened. fd:%d\n",__func__, fd) ;
#endif

        ret = read(fd, &r, sizeof(Request));
        if(ret == -1)
        {
                perror("read");
                (*fptr[0])((void*)"failure"); // exitProgram()
        }
#ifdef DEBUG
        printf("%s: Read Request %d Bytes.\n",__func__, ret) ;
#endif



        sem_post(&infra->tsem);

        fret = fork();
        switch(fret)
```

```
{
        case -1:
                perror("fork");
                (*fptr[0])((void*)"failure"); // exitProgram()
                break;
        case 0:
                sprintf(prfd,"%d", *(infra->pipe)+0);
                switch(r.oper)
                {
                        case '+':
                                execl("./adder", "adder", prfd, NULL);
                                break;
                        case '-':
                                execl("./subtr", "subtr", prfd, NULL);
                                break;
                        case '*':
                                execl("./multi", "multi", prfd, NULL);
                                break;
                }
                printf("%s: ERROR: execl() Failed.\n", __func__, ret);
        default:
                if( sem_wait((sem_t*)&infra->smptr1) == -1)
                {
                        printf("%s:%d: Error:sem_wait() failed.\n",__func__,
__LINE__);
                        (*fptr[0])((void*)"failure");
                }
                ret = write(*(infra->pipe)+1, &r, sizeof(Request));
                if(ret == -1)
                {
                        perror("write");
                        (*fptr[0])((void*)"failure"); // exitProgram()
```

```
                        }

#ifdef DEBUG
                        printf("%s:%s: Wrote Request %d Bytes.\n",__FILE__, __func__,
ret);
#endif

                        res = (Result*)infra->smptr;
                        sleep(1);
#ifdef DEBUG
                        printf("%s:%s: Result for Client: %d is %.2f \n",__FILE__,
__func__, res->cpid, res->result);
#endif
                        ret = msgsnd(infra->mqKey, (void*)res, sizeof(float), 0);
                        if(ret == -1)
                        {
                                perror("msgsnd");
                                (*fptr[0])((void*)"failure"); // exitProgram()
                        }



        }

#ifdef DEBUG
        printf("%s: End.\n",__func__) ;
#endif
        return 0;
}
```

3.1.7 readResult.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"
#include "declarations.h"

void* readResult(void *arg)
{
	int ret;
	Infra *infra;

#ifdef DEBUG
	printf("%s: Begin.\n",__func__);
#endif

	infra= (Infra*)arg;

	ret = msgsnd(infra->mqKey, infra->smptr, sizeof(float), 0);
	if(ret == -1)
	{
		perror("msgsnd");
		pthread_exit("failure");
	}

#ifdef DEBUG
	printf("%s: read Result %d Bytes.\n",__func__, ret) ;
#endif


#ifdef DEBUG
	printf("%s: End.\n",__func__) ;
#endif
```

```
	return (void*)0;
}
```

3.1.8 server.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"
#include "declarations.h"
int main()
{
	int ret;
	pthread_t thid;
	Infra *infra;
#ifdef DEBUG
	printf("%s: Begin.\n",__func__);
#endif

	init();
	infra = (*fptr[1])(0);  //  createInfra();

	ret = pthread_create(&thid, NULL, fptr[3], (void*)infra);
	if( ret == -1)
	{
		perror("pthread_create");
		(*fptr[0])((void*)"failure");   // exitProgram()
	}

	while(1)
	{
		sem_wait(&infra->tsem);
		ret = pthread_create(&thid, NULL, fptr[2], (void*)infra);
```

```
                if( ret == -1)
                {
                        perror("pthread_create");
                        (*fptr[0])((void*)"failure");   // exitProgram()

                }
        }

#ifdef DEBUG
        printf("%s: End.\n",__func__) ;
#endif
        return 0;
}
```

3.2 COMMON MODULE

3.2.1 dataStruct.h

```
#ifndef FIFONAME
#define FIFONAME "./reqFIFO"
#endif

#ifndef KEY_SHM
#define KEY_SHM 2222
#endif

#ifndef KEY_MQ
#define KEY_MQ 3333
#endif

#ifndef KEY_SHM1
#define KEY_SHM1 4444
#endif

typedef struct
{
        int *pipe;
        char *fifoName;
        int mqKey;
        int smKey;
        void *smptr;
        sem_t tsem;
        int smKey1;
        void *smptr1;
        sem_t tsem1;
}Infra;
```

```
typedef struct
{
        pid_t cpid;
        int opr1;
        int opr2;
        char oper;
}Request;

typedef struct
{
        pid_t cpid;
        float result;
}Result;
```

3.2.1 headers.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <pthread.h>
#include <fcntl.h>
```

3.3 CLIENTS MODULE

3.3.1 rc1.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"

int main()
{
	int fd,ret, mqid;
	Request r;
	Result res;

#ifdef DEBUG
	printf("%s: Begin.\n",__func__);
#endif

		fd = open(FIFONAME, O_WRONLY);
		if( fd == -1)
		{
			perror("open");
			exit(EXIT_FAILURE);
		}
#ifdef DEBUG
	printf("%s: fd:%d\n",__func__, fd);
#endif

	r.cpid = getpid();
	r.opr1 = 80;
	r.opr2 = 20;
	r.oper = '+';
	ret = write(fd, &r, sizeof(Request));
	if(ret == -1)
	{
		perror("write");
		exit(EXIT_FAILURE);
	}

#ifdef DEBUG
	printf("%s: Wrote request %d Bytes.\n",__func__, ret) ;
#endif

	mqid = msgget(KEY_MQ, IPC_CREAT|0666);
	if(mqid == -1)
	{
		perror("msgget");
		exit(EXIT_FAILURE);
	}

	ret = msgrcv(mqid, &res, sizeof(float), getpid(), 0);
```

```
	if(ret == -1)
	{
		perror("msgrcv");
		exit(EXIT_FAILURE);
	}
	printf("%s: Result:%f\n", __func__, res.result);

#ifdef DEBUG
	printf("%s: End.\n",__func__) ;
#endif
	return 0;
}
```

3.3.2 rc2.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"

int main()
{
	int fd,ret, mqid;
	Request r;
	Result res;

#ifdef DEBUG
	printf("%s: Begin.\n",__func__);
#endif

		fd = open(FIFONAME, O_WRONLY);
		if( fd == -1)
		{
			perror("open");
			exit(EXIT_FAILURE);
		}
#ifdef DEBUG
	printf("%s: fd:%d\n",__func__, fd);
#endif

	r.cpid = getpid();
	r.opr1 = 80;
	r.opr2 = 20;
	r.oper = '+';
	ret = write(fd, &r, sizeof(Request));
	if(ret == -1)
	{
		perror("write");
		exit(EXIT_FAILURE);
	}

#ifdef DEBUG
```

```
        printf("%s: Wrote request %d Bytes.\n",__func__, ret) ;
#endif

        mqid = msgget(KEY_MQ, IPC_CREAT|0666);
        if(mqid == -1)
    {
        perror("msgget");
        exit(EXIT_FAILURE);
    }

    ret = msgrcv(mqid, &res, sizeof(float), getpid(), 0);
    if(ret == -1)
    {
        perror("msgrcv");
        exit(EXIT_FAILURE);
    }
    printf("%s: Result:%f\n", __func__, res.result);

#ifdef DEBUG
        printf("%s: End.\n",__func__) ;
#endif
        return 0;
}
```

3.3.3 rc3.c

```
#include "../Common/headers.h"
#include "../Common/dataStruct.h"

int main()
{
        int fd,ret, mqid;
        Request r;
        Result res;

#ifdef DEBUG
        printf("%s: Begin.\n",__func__);
#endif

                fd = open(FIFONAME, O_WRONLY);
                if( fd == -1)
                {
                        perror("open");
                        exit(EXIT_FAILURE);
                }
#ifdef DEBUG
        printf("%s: fd:%d\n",__func__, fd);
#endif
```

```
	r.cpid = getpid();
	r.opr1 = 80;
	r.opr2 = 20;
	r.oper = '+';
	ret = write(fd, &r, sizeof(Request));
	if(ret == -1)
	{
		perror("write");
		exit(EXIT_FAILURE);
	}

#ifdef DEBUG
	printf("%s: Wrote request %d Bytes.\n",__func__, ret) ;
#endif

	mqid = msgget(KEY_MQ, IPC_CREAT|0666);
	if(mqid == -1)
	{
		perror("msgget");
		exit(EXIT_FAILURE);
	}

	ret = msgrcv(mqid, &res, sizeof(float), getpid(), 0);
	if(ret == -1)
	{
		perror("msgrcv");
		exit(EXIT_FAILURE);
	}
	printf("%s: Result:%f\n", __func__, res.result);

#ifdef DEBUG
	printf("%s: End.\n",__func__) ;
#endif
	return 0;
}
```

3.4 VENDORS MODULE

3.4.1 ADDER MODULE

3.4.1.1 declarations.h

```
void* vThreadAdder(void*);
```

3.4.1.2 headers.h

```
#include "../../Common/headers.h"
#include "../../Common/dataStruct.h"
```

3.4.1.3 main.c

```
#include "headers.h"
#include "declarations.h"

int main(int argc, char *argv[])
{
	int pfd, ret, smid;
	pthread_t thid;
	void *smptr;

#ifdef DEBUG
	printf("%s:%s: Begin.\n", __FILE__, __func__);
#endif

	if(argc != 2)
	{
		printf("ERROR: Pipe req_ad fd not Received.\n");
		return -1;
	}
	pfd = atoi(argv[1]);

#ifdef DEBUG
```

```
        printf("%s:%s: Pipe fd:%d\n", __FILE__, __func__, pfd);
#endif



        if(pthread_create(&thid, 0, vThreadAdder, (void*)&pfd) == -1)
        {
                perror("pthread_create");
                exit(EXIT_FAILURE);
        }



#ifdef DEBUG
        printf("%s:%s: End.\n", __FILE__, __func__) ;
#endif
        return 0;
}
```

3.4.1.4 Makefile

```
#! /bin/bash

adder: main.o vThreadAdder.o
        gcc -o adder main.o vThreadAdder.o

vThreadAdder.o: vThreadAdder.c headers.h declarations.h
        gcc -c vThreadAdder.c -DDEBUG
main.o: main.c headers.h declarations.h
        gcc -c main.c -DDEBUG
```

3.4.1.5 vThreadAdder.c

```
#include "headers.h"
#include "declarations.h"
void* vThreadAdder(void *arg)
{
	int pfd, ret, smid;
	pthread_t thid;
	void *smptr1;
	int smKey1;
	Request req;

#ifdef DEBUG
	printf("%s:%s: Begin.\n", __FILE__, __func__);
#endif
	pfd = *(int*)arg;

	smKey1 = shmget((key_t)KEY_SHM1, sizeof(sem_t), IPC_CREAT|0666);
	if( smKey1 == -1)
	{
		perror("shmget");
		exit(EXIT_FAILURE);
	}
	smptr1 = shmat(smKey1, NULL, 0);
	if(!smptr1)
	{
		perror("shmat");
		exit(EXIT_FAILURE);
	}

	ret = read(pfd, &req, sizeof(Request));
	if(ret == -1)
```

```
	{
		perror("read");
		exit(EXIT_FAILURE);
	}
#ifdef DEBUG
	printf("%s:%s: Request Read %d Bytes.\n", __FILE__, __func__, ret);
#endif
	if(sem_post((sem_t*)smptr1) == -1)
	{
		perror("sem_post");
		exit(EXIT_FAILURE);
	}
/*
	smid = shmget(KEY_SHM, sizeof(Result), IPC_CREAT|0666);
	if(smid == -1)
	{
		perror("shmget");
		exit(EXIT_FAILURE);
	}
	smptr = shmat(smid, NULL, 0);
	res = (Result*)smptr;
	res->result = (float)r.opr1 + r.opr2;
	res->cpid = r.cpid;
*/

#ifdef DEBUG
	printf("%s:%s: End.\n", __FILE__, __func__);
#endif
	return 0;
}
```

3.4.2 SUBTR MODULE

3.4.2.1 declarations.h

```
void* vThreadSubtr(void*);
```

3.4.2.2 headers.h

```
#include "../../Common/headers.h"
#include "../../Common/dataStruct.h"
```

3.4.2.3 main.c

```
#include "headers.h"
#include "declarations.h"

int main(int argc, char *argv[])
{
	int pfd, ret ,smid;
	pthread_t thid;
	Request r;
	Result *res;
	void *smptr;

#ifdef DEBUG
	printf("%s:%s: Begin.\n", __FILE__, __func__);
#endif

	if(argc != 2)
	{
		printf("ERROR: Pipe req_ad fd not Received.\n");
		return -1;
	}
	pfd = atoi(argv[1]);
```

```
#ifdef DEBUG
        printf("%s:%s: Pipe fd:%d\n", __FILE__, __func__, pfd);
#endif
        ret = read(pfd, &r, sizeof(Request));
        if(ret == -1)
        {
                perror("read");
                exit(EXIT_FAILURE);
        }
#ifdef DEBUG
        printf("%s:%s: Read Request %d Bytes.\n", __FILE__, __func__, ret);
#endif
        if(r.oper != '-')
        {
                printf("%s:%s: Invalid Request.\n", __FILE__, __func__);
                return 0;
        }

        if(pthread_create(&thid, 0, vThreadSubtr, (void*)&r) == -1)
        {
                perror("pthread_create");
                exit(EXIT_FAILURE);
        }

        smid = shmget(KEY_SHM, sizeof(Result), IPC_CREAT|0666);
        if(smid == -1)
        {
                perror("shmget");
                exit(EXIT_FAILURE);
        }
        smptr = shmat(smid, NULL, 0);
        res = (Result*)smptr;
```

```
        res->result = (float)r.opr1 - r.opr2;
        res->cpid = r.cpid;


#ifdef DEBUG
        printf("%s:%s: End.\n", __FILE__, __func__) ;
#endif
        return 0;
}
```

3.4.2.4 Makefile

```
#! /bin/bash

subtr: main.o vThreadSubtr.o
        gcc -o subtr main.o vThreadSubtr.o

vThreadSubtr.o: vThreadSubtr.c headers.h declarations.h
        gcc -c vThreadSubtr.c -DDEBUG
main.o: main.c headers.h declarations.h
        gcc -c main.c -DDEBUG
```

3.4.2.5 vThreadSubtr.c

```
#include "headers.h"
#include "declarations.h"

void* vThreadSubtr(void *arg)
{
        int ret, smid;
        pthread_t thid;
```

```
	Request r;
	Result *res;
	void *smptr1;
	int smKey1;

#ifdef DEBUG
	printf("%s:%s: Begin.\n", __FILE__, __func__);
#endif

	smKey1 = shmget((key_t)KEY_SHM1, sizeof(sem_t), IPC_CREAT|0666);
	if( smKey1 == -1)
	{
		perror("shmget");
		exit(EXIT_FAILURE);
	}


	smptr1 = shmat(smKey1, NULL, 0);
	if(!smptr1)
	{
		perror("shmat");
		exit(EXIT_FAILURE);
	}

	if(sem_post((sem_t*)smptr1) == -1)
	{
		perror("sem_post");
		exit(EXIT_FAILURE);
	}
/*
	smid = shmget(KEY_SHM, sizeof(Result), IPC_CREAT|0666);
	if(smid == -1)
```

```
        {
                perror("shmget");
                exit(EXIT_FAILURE);
        }
        smptr = shmat(smid, NULL, 0);
        res = (Result*)smptr;

        res->result = (float)r.opr1 + r.opr2;
        res->cpid = r.cpid;
*/

#ifdef DEBUG
        printf("%s:%s: End.\n", __FILE__, __func__) ;
#endif
        return 0;
}
```

3.4.3 MULTI MODULE

3.4.3.1 declarations.h

```
void* vThreadMulti(void*);
```

3.4.3.2 headers.h

```
#include "../../Common/headers.h"
#include "../../Common/dataStruct.h"
```

3.4.3.3 main.c

```
#include "headers.h"
#include "declarations.h"

int main(int argc, char *argv[])
{
	int pfd, ret, smid;
	pthread_t thid;
	Request r;
	Result *res;
	void *smptr;

#ifdef DEBUG
	printf("%s:%s: Begin.\n", __FILE__, __func__);
#endif

	if(argc != 2)
	{
		printf("ERROR: Pipe req_ad fd not Received.\n");
		return -1;
	}
	pfd = atoi(argv[1]);
```

```
#ifdef DEBUG
	printf("%s:%s: Pipe fd:%d\n", __FILE__, __func__, pfd);
#endif
	ret = read(pfd, &r, sizeof(Request));
	if(ret == -1)
	{
		perror("read");
		exit(EXIT_FAILURE);
	}
#ifdef DEBUG
	printf("%s:%s: Read Request %d Bytes.\n", __FILE__, __func__, ret);
#endif
	if(r.oper != '*')
	{
		printf("%s:%s: Invalid Request.\n", __FILE__, __func__);
		return 0;
	}

	if(pthread_create(&thid, 0, vThreadMulti, (void*)&r) == -1)
	{
		perror("pthread_create");
		exit(EXIT_FAILURE);
	}

	smid = shmget(KEY_SHM, sizeof(Result), IPC_CREAT|0666);
	if(smid == -1)
	{
		perror("shmget");
		exit(EXIT_FAILURE);
	}
	smptr = shmat(smid, NULL, 0);
	res = (Result*)smptr;
```

```
        res->result = (float)r.opr1 * r.opr2;
        res->cpid = r.cpid;


#ifdef DEBUG
        printf("%s:%s: End.\n", __FILE__, __func__) ;
#endif
        return 0;
}
```

3.4.3.4 Makefile

```
#! /bin/bash

multi: main.o vThreadMulti.o
        gcc -o adder main.o vThreadMulti.o

vThreadMulti.o: vThreadMulti.c headers.h declarations.h
        gcc -c vThreadMulti.c -DDEBUG
main.o: main.c headers.h declarations.h
        gcc -c main.c -DDEBUG
```

3.4.3.5 vThreadMulti.c

```
#include "headers.h"
#include "declarations.h"

void* vThreadMulti(void *arg)
{
        int pfd, ret, smid;
        pthread_t thid;
```

```
	Request r;
	Result *res;
	void *smptr1;
	int smKey1;

#ifdef DEBUG
	printf("%s:%s: Begin.\n", __FILE__, __func__);
#endif

	smKey1 = shmget((key_t)KEY_SHM1, sizeof(sem_t), IPC_CREAT|0666);
	if( smKey1 == -1)
	{
		perror("shmget");
		exit(EXIT_FAILURE);
	}


	smptr1 = shmat(smKey1, NULL, 0);
	if(!smptr1)
	{
		perror("shmat");
		exit(EXIT_FAILURE);
	}

	if(sem_post((sem_t*)smptr1) == -1)
	{
		perror("sem_post");
		exit(EXIT_FAILURE);
	}
/*
	smid = shmget(KEY_SHM, sizeof(Result), IPC_CREAT|0666);
	if(smid == -1)
```

```
        {
                perror("shmget");
                exit(EXIT_FAILURE);
        }
        smptr = shmat(smid, NULL, 0);
        res = (Result*)smptr;

        res->result = (float)r.opr1 + r.opr2;
        res->cpid = r.cpid;
*/

#ifdef DEBUG
        printf("%s:%s: End.\n", __FILE__, __func__) ;
#endif
        return 0;
}
```

CHAPTER-4

Testing

4.1 Test Case – 1

Black-Box Testing

Black Box Testing, also known as Behavioural Testing, is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.

This can be following way:

- Input interfacing
- Processing
- Output interfacing

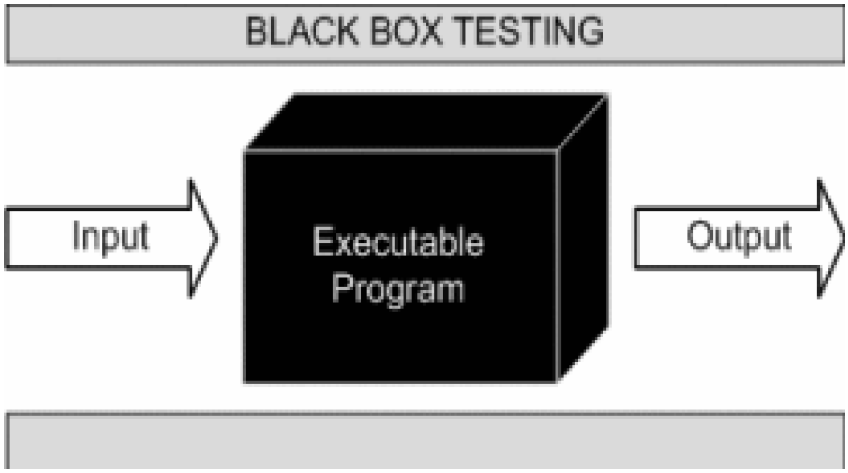

Fig: - 4.1

This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors.

4.2 Test Case – 2

White-Box Testing

White Box Testing, also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential.

White box testing is testing beyond the user interface and into the nitty-gritty of a system.

This method is named so because the software program, in the eyes of the tester, is like a white/ transparent box; inside which one clearly sees.

CHAPTER-5

FUTURE SCOPE

The future scope of a project focusing on client-server communication using Inter-Process Communication (IPC) and socket technologies is promising and offers several exciting possibilities. As technology continues to advance, there will be a growing demand for efficient and secure communication between clients and servers in various domains. Enhancements can be made to the project to incorporate emerging communication protocols, such as WebSocket or gRPC, to facilitate real-time and bi-directional communication. Additionally, integrating advanced security mechanisms like Transport Layer Security (TLS) or implementing encryption algorithms will strengthen the project's security framework. Furthermore, exploring cloud-based solutions, such as serverless architectures or containerization technologies like Docker, can provide scalability and flexibility to the system. With the ever-increasing need for fast and reliable communication in a connected world, this project has a bright future and can be adapted to meet the evolving demands of client-server communication.

CHAPTER-6

Conclusion

In conclusion, the project on client-server communication using IPC (Inter-Process Communication) and sockets provides a comprehensive understanding of how these technologies facilitate efficient communication between client and server systems. IPC mechanisms such as shared memory, message queues, and pipes allow processes to exchange data within the same system, while sockets enable communication across different systems over a network. Throughout the project, we explored the benefits and drawbacks of using IPC and sockets for client-server communication. IPC offers fast and direct communication between processes, making it suitable for situations where low latency is crucial. It allows for efficient data sharing and synchronization between client and server, enhancing overall system performance. On the other hand, IPC requires careful coordination and synchronization to avoid race conditions and data corruption. Sockets, on the other hand, provide a flexible and scalable solution for client-server communication over networks. They enable communication between systems, making it possible to establish connections over the internet. Sockets offer reliability, error handling, and support for various protocols, making them widely used in network programming. However, socket communication may introduce higher latency due to network overhead. Overall, the project demonstrated that the choice between IPC and sockets depends on the specific requirements of the client-server communication scenario. If low latency and direct process interaction within the same system are essential, IPC mechanisms can be employed. Conversely, when network communication and scalability across multiple systems are necessary, sockets are the preferred option. By understanding the concepts, implementation, and trade-offs associated with IPC and sockets, this project provides a solid foundation for designing and developing efficient client-server communication systems, catering to different application requirements and constraints.

CHAPTER 7

BIBLIOGRAPHY

7.1 ONLINE LEARNING SITES

- https://www.tutorialspoint.com
- https://stackoverflow.com
- https://www.emblogic.com

REFERENCES

1. Xue, M. and Zhu, C. , “The Socket Programming and Software Design for Communication Based on Client/Server”,” 2009 Pacific-Asia Conference on Circuits, Communications and Systems, pp. 774-777,2009
2. J. McManis, S. Hashmi, M Ahsan, and J. Haider, “Developing Intelligent Software Interface for Wireless Monitoring of Vehicle Speed and Management of Associated Data”, IET Wireless Sensor Systems,2016
3. IDG Communications (2017). Sockets Programming in Java World from IDG. [Online]. Available at:https://www.javaworld.com/article/2077322/core-java/core-java-sockets-programming-in-java-a-tutorial.html. [Accessed 17-July-2017]
4. T. Duong-Ba, T. Nguyen, 4B. Rose, D. Tan “Distributed Client Server Assignment for Computing, Volume 2, Issue 4, pp. 422 – 435,December 201