LOAD BALANCER USING SOCKET

MINI PROJECT REPORT

Submitted by

BHUVANIKA S (9517202109011)

RAJAKUMARI S (9517202109042)

SUJI S (9517202109051)

In

19AD551 – COMPUTER NETWORKING LABORATORY

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

MEPCO SCHLENK ENGINEERING COLLEGE

SIVAKASI

OCTOBER 2023

MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI AUTONOMOUS

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

BONAFIDE CERTIFICATE

This is to certify that it is the bonafide work of **RAJAKUMARI.S (9517202109042), SUJI.S (9517202109051) , BHUVANIKA.S(9517202109011)** for the mini project titled **"LOAD BALANCING USING SOCKET "** in 19AD551 – Computer Networking Laboratory during the fifth semester July 2023 – October 2023 under my supervision.

SIGNATURE	**SIGNATURE**
Mrs. P. Swathika,	**Dr. J. Angela Jennifa Sujana,**
Asst. Professor (Senior Grade),	**Professor & Head,**
AI&DS Department,	AI&DS Department
Mepco Schlenk Engg. College,Sivakasi	Mepco Schlenk Engg. College, Sivakasi

ABSTRACT

This Simple TCP Load Balancer project is a software solution designed to distribute incoming network traffic or computational tasks efficiently across multiple servers or processing units. In modern computing environments, where high availability, scalability, and responsiveness are critical, load balancers play a pivotal role in optimizing resource utilization and ensuring the reliability of services.This project's primary objective is to create a robust and multithreaded load balancer capable of intelligently routing requests to a pool of backend servers. The load balancer leverages the power of multithreading to handle a high volume of incoming requests concurrently while making informed decisions to distribute the load evenly among the available resources. This project seeks to create a flexible, efficient, and highly responsive load balancing solution that enhances the availability and performance of network services. By efficiently distributing incoming requests among backend servers, it aims to optimize resource utilization, prevent overloads, and ensure a seamless and reliable user experience.

TABLE OF CONTENTS

CHAPTER 1

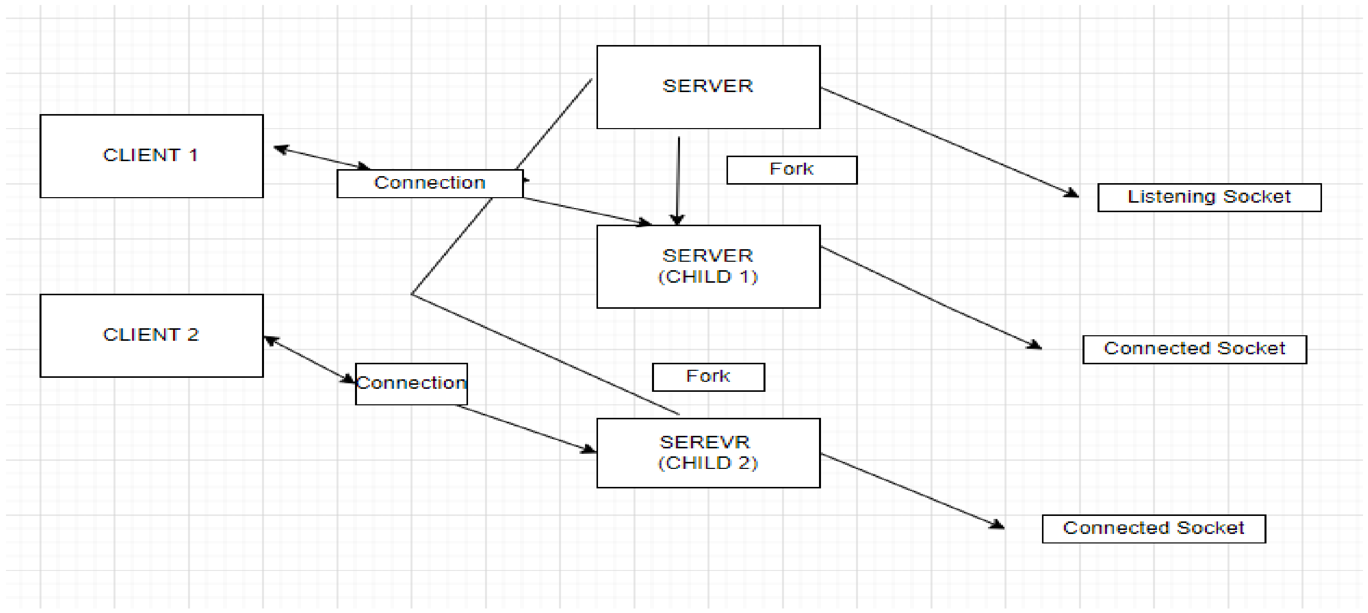
INTRODUCTION

In today's world, where we rely heavily on online services like websites and apps, it's crucial that these services are always available, can handle lots of users, and respond quickly. Imagine you have a popular website, and hundreds or thousands of people are trying to visit it at the same time. To make sure all these people can access your website without it crashing or slowing down, you need something called a "load balancer." A load balancer is like a traffic cop for the internet. It takes all the requests from users and directs them to different servers where your website is hosted. This ensures that no single server gets overwhelmed, and the website stays fast and reliable.Now, the "Multithreaded Load Balancer" project is all about creating a really smart and efficient traffic cop. It uses a special technique called "multithreading" to handle many requests at once. Think of it like having multiple traffic cops working together to manage the flow of users. This project not only helps distribute the traffic but does it in a clever way. It can do things like making sure servers with fewer users get more requests or sending requests based on how powerful each server is. It's like sending more people to the stronger traffic cop when things get really busy. Plus, if a server is not working well, it quickly sends traffic away from that server to keep your website running smoothly.And, as your website grows and more people visit it, this project can easily handle the extra traffic. It's like adding more traffic cops to the team when the city gets busier.So, in a nutshell, the "Multithreaded Load Balancer" project is all about making sure your online services are available, fast, and can grow with your needs by efficiently directing internet traffic to the right places.

1.1.1 TCP/IP

For a Multithreaded Load Balancer project, the Transmission Control Protocol (TCP) and Internet Protocol (IP) play crucial roles in the networking aspects of the system. TCP/IP is a fundamental suite of networking protocols that enable data to be transmitted over networks and the internet. In the context of a load balancer, TCP/IP are essential for handling and routing network traffic efficiently. The project should be built on the foundation of the TCP/IP networking model, which consists of multiple layers, including the Link Layer, Internet Layer (IP), Transport Layer (TCP/UDP), and Application Layer. The Internet Layer, specifically IP

(Internet Protocol), is responsible for routing and addressing packets of data, ensuring they reach their destination across networks. The load balancer, being responsible for distributing incoming requests to backend servers, operates primarily at the Transport Layer (Layer 4) of the TCP/IP model. TCP, a connection-oriented protocol, is frequently used by load balancers to handle and balance traffic because of its ability to maintain reliable connections and ensure ordered packet delivery. The load balancer must have the capability to route incoming traffic to the appropriate backend servers based on IP addresses, which is a core function of the IP layer. IP addresses play a critical role in identifying both the clients sending requests and the backend servers handling those requests. In addition to IP addresses, the load balancer uses port numbers to determine which service or application on a server should handle a given request. This port-based routing is achieved through the Transport Layer protocol, either TCP or UDP, depending on the type of service. TCP/IP protocols, particularly TCP, are responsible for managing connections between clients and backend servers. The load balancer must maintain and manage multiple TCP connections to various backend servers efficiently, ensuring that requests are properly forwarded and responses are directed back to the clients. The load balancer's configuration should allow for specifying IP address ranges, port numbers, and load balancing algorithms to determine how traffic is distributed.



1.1.2 NEED FOR TCP

TCP is an essential choice for the "Multithreaded Load Balancer" project for several compelling reasons. First and foremost, TCP is renowned for its reliability, ensuring that data is delivered correctly and in the right order, which is vital for maintaining the integrity of requests and responses in a load balancing context. Its stateful connections enable the load balancer to manage client-server relationships seamlessly, facilitating even distribution of traffic and consistent user experiences. Moreover, TCP's support for port-based routing is crucial in scenarios where various services share the same server, allowing for precise allocation of requests. The protocol's compatibility with a wide array of networked applications makes it practical and versatile, ensuring that the load balancer can adapt to diverse environments. TCP's security features, such as encryption through TLS/SSL, fortify the transmission of data, particularly when sensitive information is involved, adding an extra layer of protection. Lastly, TCP's comprehensive load balancing capabilities, including session persistence, health checks, and fine-grained control over request distribution, make it the ideal choice to meet the complex demands of load balancing in the project, ensuring reliable and efficient data transmission.

1.2 OBJECTIVE

- The primary objectives of the "Multithreaded Load Balancer" project are to develop a robust and dynamic load balancing solution that efficiently distributes incoming network traffic or requests to a pool of backend servers.
- The project aims to optimize resource utilization and prevent overloads by offering a variety of load balancing algorithms such as round-robin, weighted round-robin, and least connections.
- The load balancer will support dynamic server management, enabling seamless server registration and deregistration with automatic health checks to ensure high availability.
- It will provide an intuitive user interface for easy configuration, comprehensive logging and monitoring features for performance analysis, and robust security measures to protect against DDoS attacks and ensure data privacy.

- Scalability is a key objective, allowing for seamless expansion to meet growing service demands. Compatibility with a wide range of network protocols, services, and applications is another essential goal to ensure the load balancer's versatility.
- Ultimately, the project aims to enhance network performance, service availability, and user experience while offering flexibility and extensibility to address diverse use cases.

1.3 SCOPE OF THE PROJECT

The scope of the "Multithreaded Load Balancer" project can vary based on your specific objectives and requirements .Implement and support a variety of load balancing algorithms (e.g., round-robin, least connections, weighted round-robin, IP hash) to distribute incoming requests to backend servers. Allow for easy customization and addition of new algorithms. Develop a mechanism for dynamic server registration and deregistration without service disruption.Implement health checks to monitor the status of backend servers and automatically route traffic away from failing or overloaded servers. Design the load balancer to be scalable, enabling the addition of new load balancer instances as traffic and service demands increase. Consider how the load balancer can adapt to changing infrastructure sizes and configurations. Provide an intuitive and user-friendly interface or configuration files for users to set up load balancing rules, specify algorithms, and configure health checks. Allow for the customization of advanced settings to accommodate specific use cases. Implement comprehensive logging and monitoring features to capture performance metrics, server health, and load balancing decisions. Create dashboards or integration with external monitoring tools to visualize and analyze data. Incorporate security measures to protect against DDoS attacks and ensure the privacy and integrity of data. Consider implementing rate limiting, IP blocking, and other security mechanisms. Support various network protocols, including HTTP, HTTPS, TCP, and UDP. Ensure compatibility with common application and network services. Explore options for high availability configurations, such as active-standby or active-active setups, to minimize downtime. Provide real-time and historical metrics on load distribution, including server response times, request rates, and error rates. Create comprehensive documentation to help users set up and configure the load balancer effectively. Offer customer support or a user community where users can seek assistance and share best practices.Continuously work on optimizing the performance of the load balancer to handle high volumes of requests efficiently. Ensure compatibility with various operating systems and deployment environments, including on-premises and cloud-based setups. Consider extensibility by allowing users to integrate custom plugins or scripts for specific use cases or to extend functionality. Implement thorough testing procedures to ensure the load balancer's stability, reliability, and accuracy under various conditions. Decide on the licensing model (e.g., open source, commercial) and distribution method for the load balancer. Remember that project scope can evolve, and it's crucial to define and document it clearly to guide the development process effectively. Be open to feedback from users and stakeholders, and be prepared to adjust the scope as necessary to meet changing requirements and expectations.

1.4 FLOW DIAGRAM

A loadbalancer's role is to equally distribute the load on all the servers. When multiple clients attempt to connect to the server, loadbalancer checks the load on each server and sends the request to the server with the least load. Here the task of the server is to capitalize the passed string.

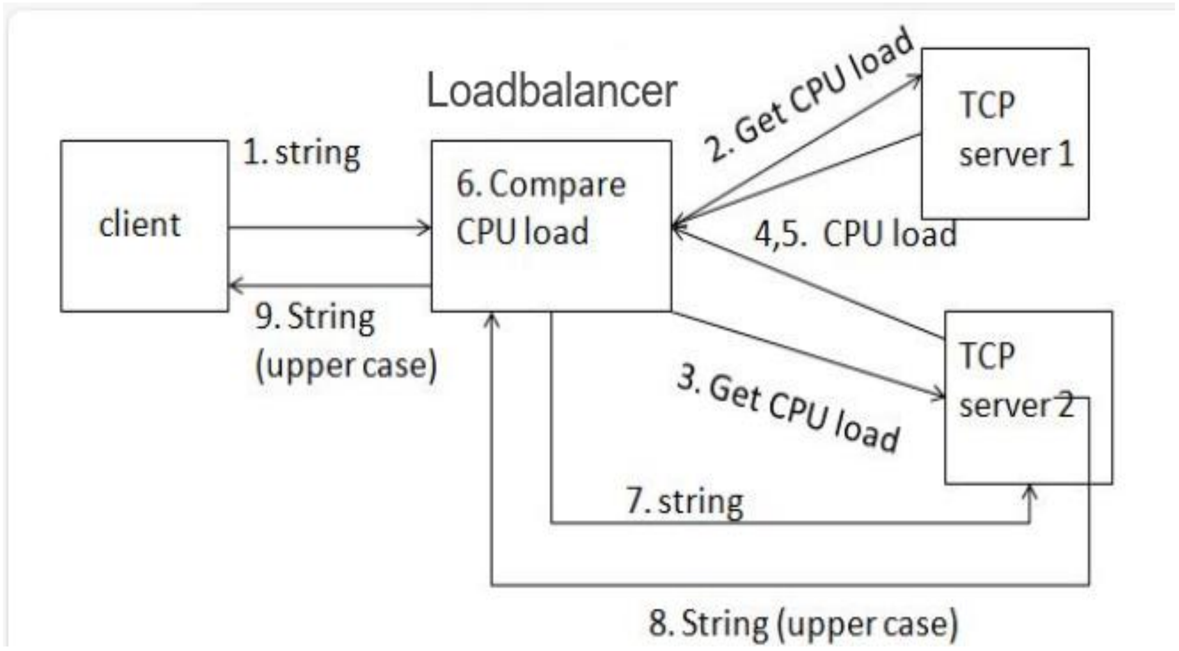


Fig 1.4.1 flow diagram

CHAPTER 2

IMPLEMENTATION

2.1 Source Code

SERVER

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

#define BACKLOG 10

int clientNumber = 0;
int numberOfConnections = 0;

typedef struct pthread_arg_t {
     int new_socket_fd;
     struct sockaddr_in client_address;
} pthread_arg_t;
void *pthread_routine(void *arg);
void signal_handler(int signal_number);

int main(int argc, char *argv[]) {

      int port, socket_fd, new_socket_fd;

      struct sockaddr_in address;

      pthread_attr_t pthread_attr;

      pthread_arg_t *pthread_arg;

      pthread_t pthread;

      socklen_t client_address_len;

            port = argc > 1 ? atoi(argv[1]) : 0;

      if (!port) {

            printf("Enter Port: ");

            scanf("%d", &port);

      }

            memset(&address, 0, sizeof address);

      address.sin_family = AF_INET;

      address.sin_port = htons(port);

      address.sin_addr.s_addr = INADDR_ANY;

                      if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0))
== -1) {

            perror("socket");

            exit(1);

      }

         if (bind(socket_fd, (struct sockaddr *)&address, sizeof
address) == -1) {

            perror("bind");

            exit(1);

      }

            if (listen(socket_fd, BACKLOG) == -1) {
```

```
            perror("listen");
            exit(1);
      }
      if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
            perror("signal");
            exit(1);
      }
      if (signal(SIGTERM, signal_handler) == SIG_ERR) {
            perror("signal");
            exit(1);
      }
      if (signal(SIGINT, signal_handler) == SIG_ERR) {
            perror("signal");
            exit(1);
      }
           if (pthread_attr_init(&pthread_attr) != 0) {
            perror("pthread_attr_init");
            exit(1);
      }
      if (pthread_attr_setdetachstate(&pthread_attr,
PTHREAD_CREATE_DETACHED) != 0) {
            perror("pthread_attr_setdetachstate");
            exit(1);
      }

      while (1) {
            pthread_arg = (pthread_arg_t *)malloc(sizeof *pthread_arg);
            if (!pthread_arg) {
                  perror("malloc");
                  continue;
            }

            client_address_len = sizeof pthread_arg->client_address;
            new_socket_fd = accept(socket_fd, (struct sockaddr
*)&pthread_arg->client_address, &client_address_len);
            if (new_socket_fd == -1) {
                  perror("accept");
                  free(pthread_arg);
                  continue;
            }
            pthread_arg->new_socket_fd = new_socket_fd;
            if (pthread_create(&pthread, &pthread_attr, pthread_routine,
(void *)pthread_arg) != 0) {
                  perror("pthread_create");
                  free(pthread_arg);
                  continue;
            }
      }     return 0;
}
void upper_string(char s[]) {
      int c = 0;
      while (s[c] != '\0') {
            if (s[c] >= 'a' && s[c] <= 'z')
            s[c] = s[c] - 32;
      c++;
```

```
	}
	for(int i = 0; i < 1000000000; i++)
		for(int j = 0; j < 10; j++)
			c = j;
	return;
}

void *pthread_routine(void *arg) {
	clientNumber++;
	int clientIdx = clientNumber;
	numberOfConnections++;
	pthread_arg_t *pthread_arg = (pthread_arg_t *)arg;
	int new_socket_fd = pthread_arg->new_socket_fd;
	struct sockaddr_in client_address = pthread_arg->client_address;

	free(arg);
	char string[100];
	read(new_socket_fd, string, 100);
	if(strcmp("__clients?",string)==0){
		printf("load check\n");
		char result[100];
		sprintf(result,"%d",numberOfConnections-1);
		write(new_socket_fd, result, 100);
		clientNumber--;
	} else {
		printf("client#%d : connected\n", clientIdx);
		upper_string(string);
```

```
			write(new_socket_fd, string, 100);
			printf("client#%d : disconnected\n", clientIdx);
	}
	close(new_socket_fd);
	numberOfConnections--;

	return NULL;
}

void signal_handler(int signal_number) {
			exit(0);
}
```

CLIENT

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define SERVER_NAME_LEN_MAX 255
#define SERVERPORT 6000

int main(int argc, char *argv[]) {
        char server_name[SERVER_NAME_LEN_MAX + 1] = "127.0.0.1\0";
      int server_port = SERVERPORT, socket_fd;
      struct hostent *server_host;
      struct sockaddr_in server_address;
          server_host = gethostbyname(server_name);
          memset(&server_address, 0, sizeof server_address);
      server_address.sin_family = AF_INET;
      server_address.sin_port = htons(server_port);
      memcpy(&server_address.sin_addr.s_addr, server_host->h_addr,
server_host->h_length);
      if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
      }
      if (connect(socket_fd, (struct sockaddr *)&server_address, sizeof
server_address) == -1) {
            perror("connect");
            exit(1);
      }
           printf("connection established\n");
          char string[100];
      printf("input : ");
      scanf("%s",string);
          write(socket_fd, string, 100);
          read(socket_fd, string, 100);
      close(socket_fd);
          printf("output : %s\n\n", string);
      return 0;
```

```
}
```

LOAD BALANCER

```
#include <netinet/in.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <netdb.h>
#define BACKLOG 10
#define SERVER_NAME_LEN_MAX 255
#define SERVERPORT 6000
#define SERVER1PORT 6001
#define SERVER2PORT 6002
int clientNumber = 0;
typedef struct pthread_arg_t {
     int new_socket_fd;
     struct sockaddr_in client_address;
     } pthread_arg_t;
void *pthread_routine(void *arg);
void signal_handler(int signal_number);
int main(int argc, char *argv[]) {
     int port = SERVERPORT, socket_fd, new_socket_fd;
     struct sockaddr_in address;
```

```
	pthread_attr_t pthread_attr;
	pthread_arg_t *pthread_arg;
	pthread_t pthread;
	socklen_t client_address_len;
	    memset(&address, 0, sizeof address);
	address.sin_family = AF_INET;
	address.sin_port = htons(port);
	address.sin_addr.s_addr = INADDR_ANY;
	if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		perror("socket");
		exit(1);
	}
	if (bind(socket_fd, (struct sockaddr *)&address, sizeof address)
== -1) {
		perror("bind");
		exit(1);
	}

	if (listen(socket_fd, BACKLOG) == -1) {
		perror("listen");
		exit(1);
	}
	if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
		perror("signal");
		exit(1);
	}
	if (signal(SIGTERM, signal_handler) == SIG_ERR) {
```

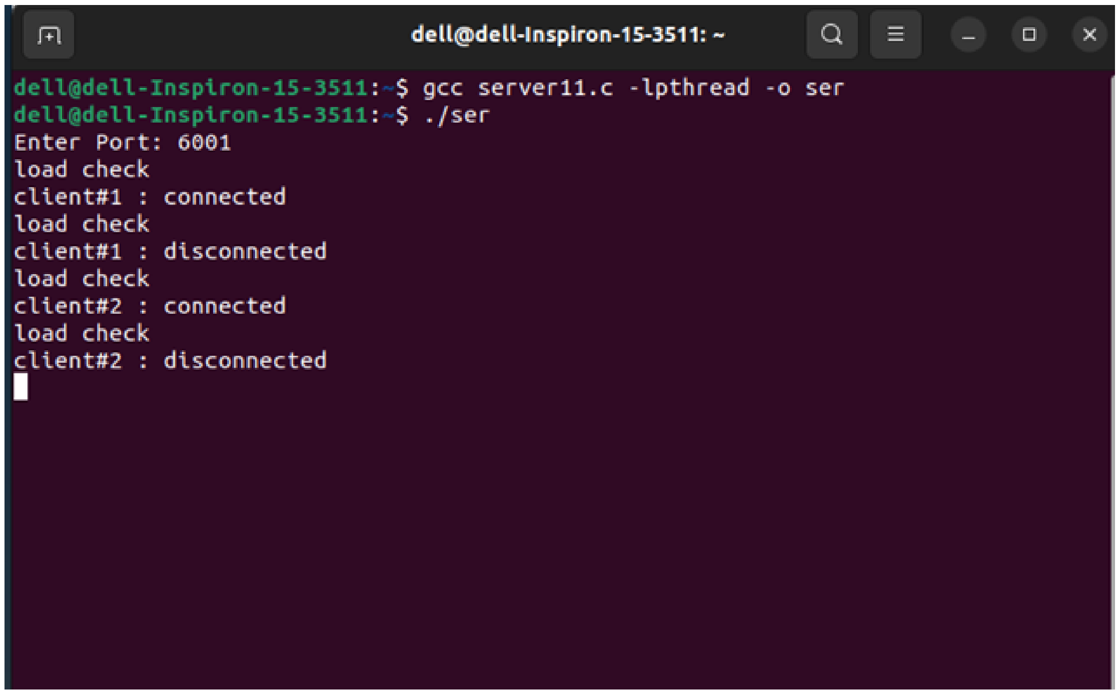
```
            perror("signal");
            exit(1);
      }
      if (signal(SIGINT, signal_handler) == SIG_ERR) {
            perror("signal");
            exit(1);
      }
      if (pthread_attr_init(&pthread_attr) != 0) {
            perror("pthread_attr_init");
            exit(1);
      }
      if (pthread_attr_setdetachstate(&pthread_attr,
PTHREAD_CREATE_DETACHED) != 0) {
            perror("pthread_attr_setdetachstate");
            exit(1);
      }

      while (1) {
            pthread_arg = (pthread_arg_t *)malloc(sizeof *pthread_arg);
            if (!pthread_arg) {
                  perror("malloc");
                  continue;
            }
            client_address_len = sizeof pthread_arg->client_address;
            new_socket_fd = accept(socket_fd, (struct sockaddr
*)&pthread_arg->client_address, &client_address_len);
            if (new_socket_fd == -1) {
                  perror("accept");
```

```
                free(pthread_arg);
                continue;
            }
            pthread_arg->new_socket_fd = new_socket_fd;
                if (pthread_create(&pthread, &pthread_attr,
pthread_routine, (void *)pthread_arg) != 0) {
                perror("pthread_create");
                free(pthread_arg);
                continue;
            }
        }
        return 0;
}
int getLoadServer(int idx){
        char server_name[SERVER_NAME_LEN_MAX + 1] = "127.0.0.1\0";
        int server_port = 0, socket_fd;
        if(idx == 1) server_port = SERVER1PORT;
        else if(idx == 2) server_port = SERVER2PORT;
        struct hostent *server_host;
        struct sockaddr_in server_address;
        server_host = gethostbyname("localhost");
        memset(&server_address, 0, sizeof server_address);
        server_address.sin_family = AF_INET;
        server_address.sin_port = htons(server_port);
        memcpy(&server_address.sin_addr.s_addr, server_host->h_addr,
server_host->h_length);

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

```
	connect(socket_fd, (struct sockaddr *)&server_address, sizeof
server_address);

	write(socket_fd, "__clients?", 100);

	char reply[100];

	read(socket_fd, reply, 100);

	close(socket_fd);

	int load = 0;

	sscanf(reply, "%d", &load);

	return load;

}

void sendquery(int idx, char input[], char reply[]){

	char server_name[SERVER_NAME_LEN_MAX + 1] = "127.0.0.1\0";

	int server_port = 0, socket_fd;

	if(idx == 1) server_port = SERVER1PORT;

	else if(idx == 2) server_port = SERVER2PORT;

	struct hostent *server_host;

	struct sockaddr_in server_address;

	server_host = gethostbyname("localhost");

	memset(&server_address, 0, sizeof server_address);

	server_address.sin_family = AF_INET;

	server_address.sin_port = htons(server_port);

	memcpy(&server_address.sin_addr.s_addr, server_host->h_addr,
server_host->h_length);

	socket_fd = socket(AF_INET, SOCK_STREAM, 0);

	connect(socket_fd, (struct sockaddr *)&server_address, sizeof
server_address);

	write(socket_fd, input, 100);

	read(socket_fd, reply, 100);
```

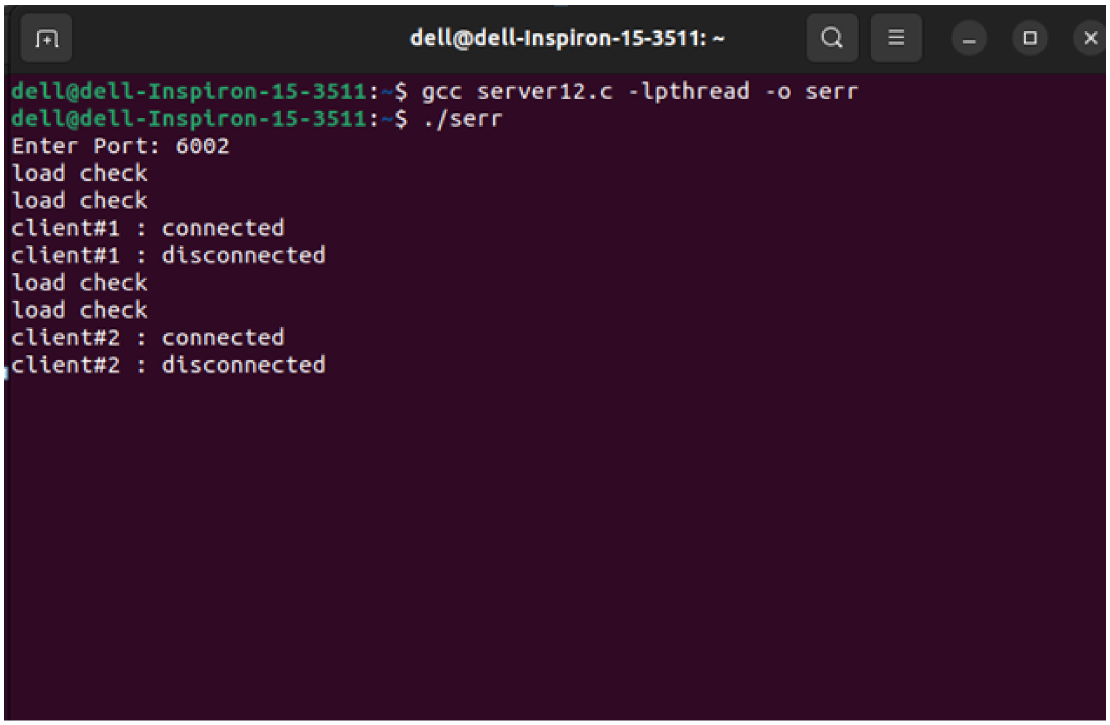
```
}

void *pthread_routine(void *arg) {
      clientNumber++;
      int clientIdx = clientNumber;
      pthread_arg_t *pthread_arg = (pthread_arg_t *)arg;
      int new_socket_fd = pthread_arg->new_socket_fd;
      struct sockaddr_in client_address = pthread_arg->client_address;
      free(arg);
      printf("client#%d : connected\n", clientIdx);
      char input[100];
      read(new_socket_fd, input, 100);
      int load1 = getLoadServer(1);
      int load2 = getLoadServer(2);
      printf("LOAD:- SERVER1: %d  SERVER2: %d\n", load1, load2);
      char reply[100];
      if(load1 <= load2) sendquery(1, input, reply);
      else sendquery(2, input, reply);
      write(new_socket_fd, reply, 100);
            printf("client%d : disconnected\n", clientIdx);
      close(new_socket_fd);
             return NULL;
}

void signal_handler(int signal_number) {
      exit(0);
}
```

2.2 OUTPUT


dell@dell-Inspiron-15-3511: ~

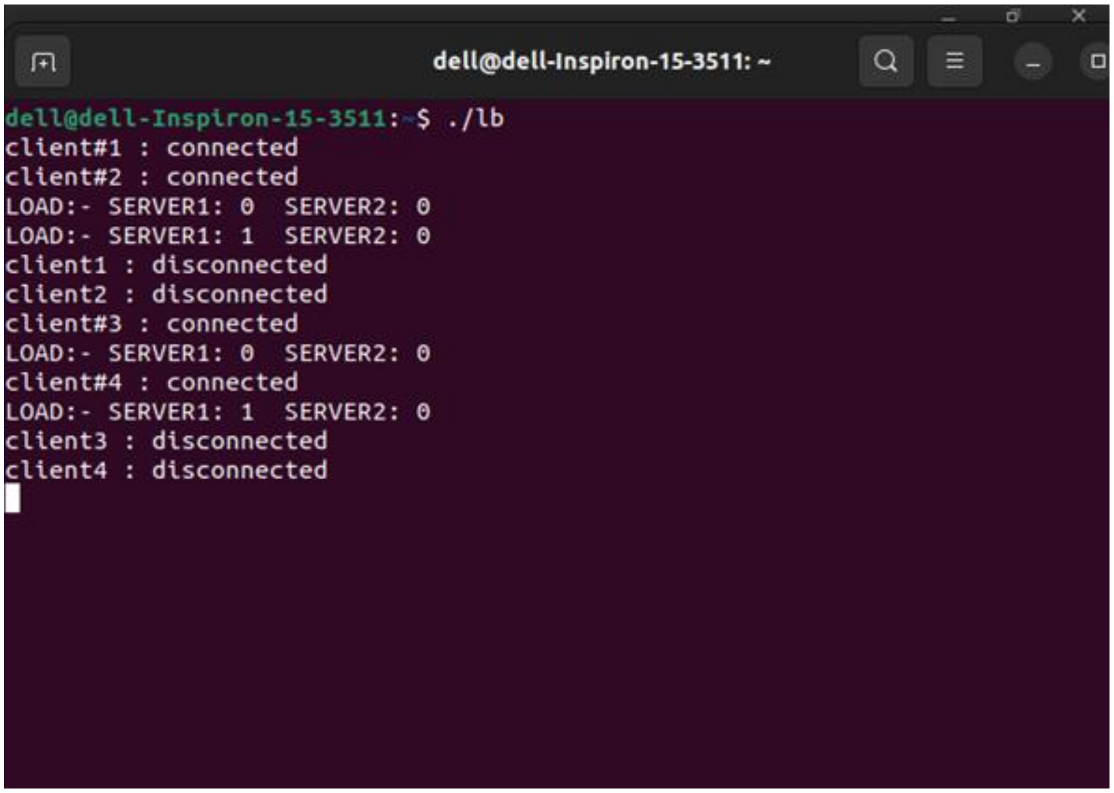
```
dell@dell-Inspiron-15-3511:~$ gcc server11.c -lpthread -o ser
dell@dell-Inspiron-15-3511:~$ ./ser
Enter Port: 6001
load check
client#1 : connected
load check
client#1 : disconnected
load check
client#2 : connected
load check
client#2 : disconnected
```

Fig 2.2.1 Server 1


dell@dell-Inspiron-15-3511: ~

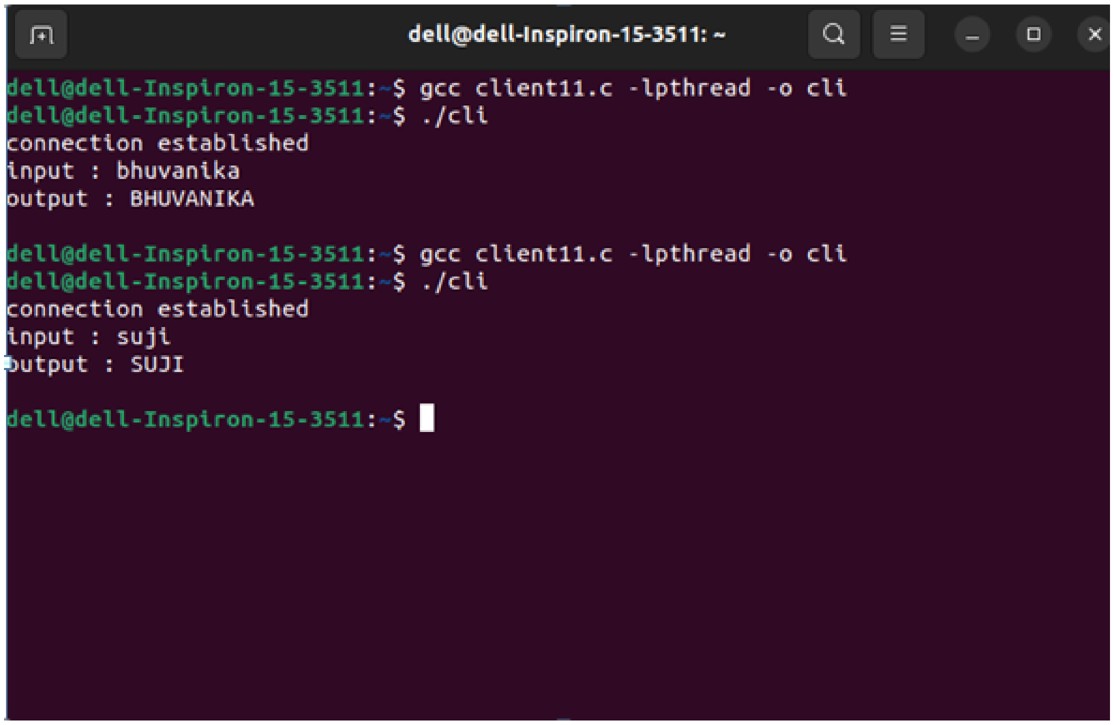
```
dell@dell-Inspiron-15-3511:~$ gcc server12.c -lpthread -o serr
dell@dell-Inspiron-15-3511:~$ ./serr
Enter Port: 6002
load check
load check
client#1 : connected
client#1 : disconnected
load check
load check
client#2 : connected
client#2 : disconnected
```

Fig 2.2.2 Server 2



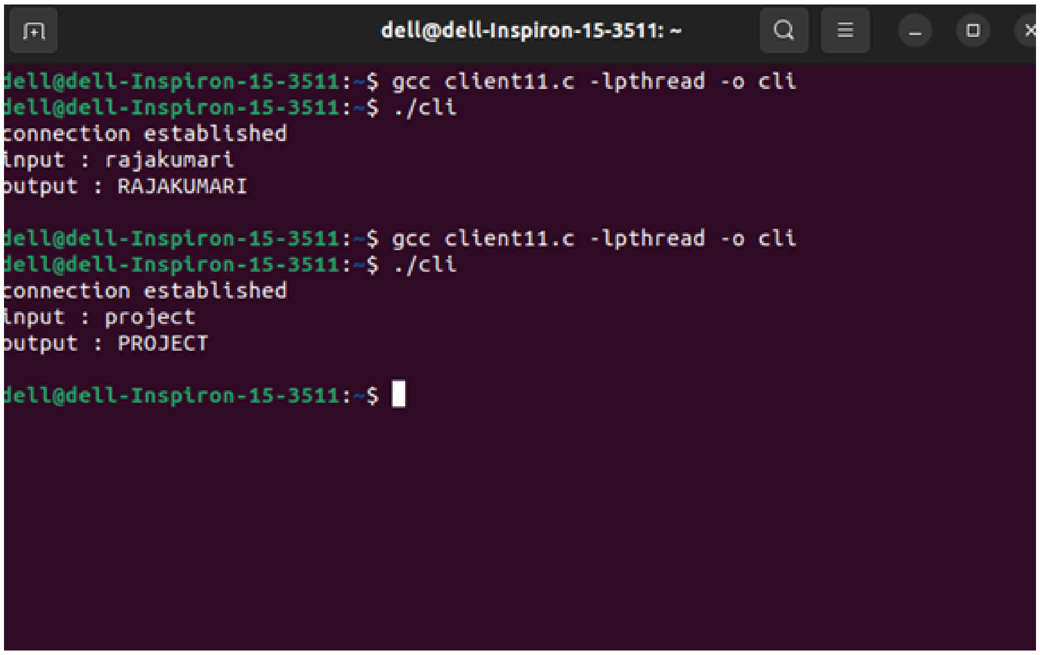
```
dell@dell-Inspiron-15-3511:~$ ./lb
client#1 : connected
client#2 : connected
LOAD:- SERVER1: 0  SERVER2: 0
LOAD:- SERVER1: 1  SERVER2: 0
client1 : disconnected
client2 : disconnected
client#3 : connected
LOAD:- SERVER1: 0  SERVER2: 0
client#4 : connected
LOAD:- SERVER1: 1  SERVER2: 0
client3 : disconnected
client4 : disconnected
```

Fig 2.2.3 Load Balancer



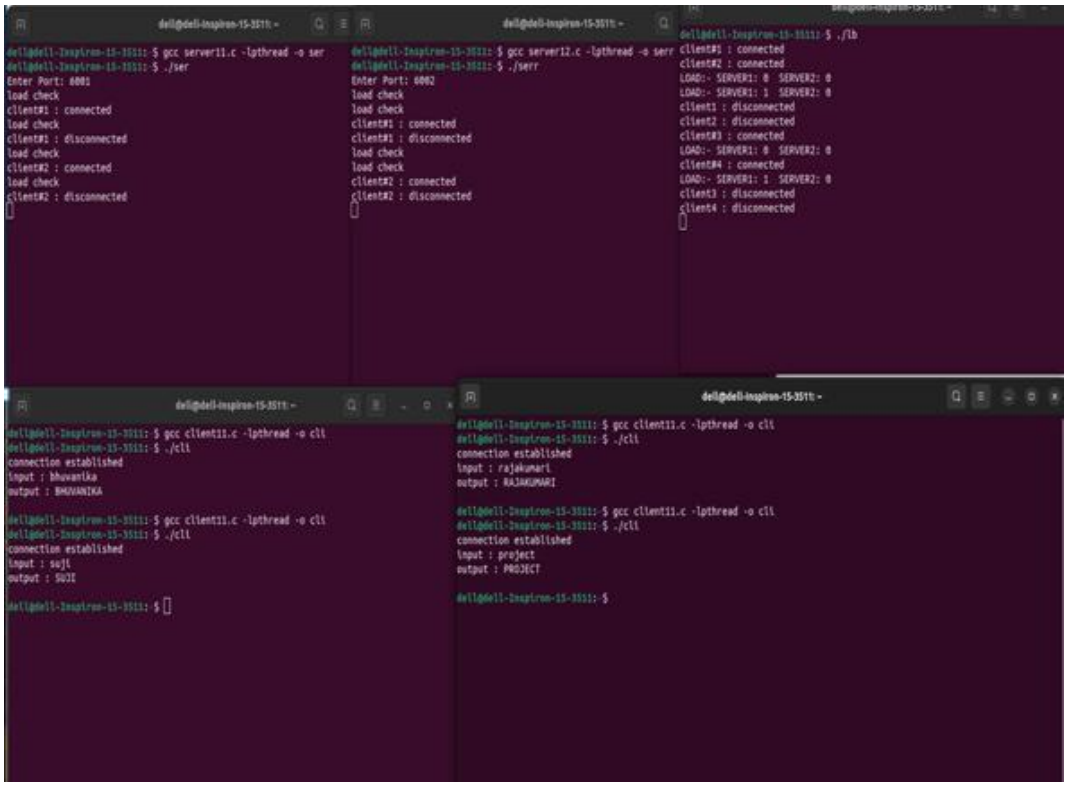
```
dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : bhuvanika
output : BHUVANIKA

dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : suji
output : SUJI

dell@dell-Inspiron-15-3511:~$
```

Fig 2.2.4 Client 1& 2



```
dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : rajakumari
output : RAJAKUMARI

dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : project
output : PROJECT

dell@dell-Inspiron-15-3511:~$
```

Fig 2.2.5 Client 3 & 4



```
dell@dell-Inspiron-15-3511:~$ gcc server11.c -lpthread -o ser
dell@dell-Inspiron-15-3511:~$ ./ser
Enter Port: 6001
load check
client#1 : connected
load check
client#1 : disconnected
load check
client#2 : connected
load check
client#2 : disconnected
```

```
dell@dell-Inspiron-15-3511:~$ gcc server12.c -lpthread -o serr
dell@dell-Inspiron-15-3511:~$ ./serr
Enter Port: 6002
load check
load check
client#1 : connected
client#1 : disconnected
load check
load check
client#2 : connected
client#2 : disconnected
```

```
dell@dell-Inspiron-15-3511:~$ ./lb
client#1 : connected
client#2 : connected
LOAD:- SERVER1: 0  SERVER2: 0
LOAD:- SERVER1: 1  SERVER2: 0
client1 : disconnected
client2 : disconnected
client#3 : connected
LOAD:- SERVER1: 0  SERVER2: 0
client#4 : connected
LOAD:- SERVER1: 1  SERVER2: 0
client3 : disconnected
client4 : disconnected
```

```
dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : bhuvanika
output : BHUVANIKA

dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : suji
output : SUJI

dell@dell-Inspiron-15-3511:~$
```

```
dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : rajakumari
output : RAJAKUMARI

dell@dell-Inspiron-15-3511:~$ gcc client11.c -lpthread -o cli
dell@dell-Inspiron-15-3511:~$ ./cli
connection established
input : project
output : PROJECT

dell@dell-Inspiron-15-3511:~$
```

Fig 2.2.6 Output

CHAPTER 3

CONCLUSION

In this project, we have developed a sophisticated load balancer capable of intelligently distributing incoming requests to a pool of backend servers. It provides a range of load balancing algorithms, each tailored to specific use cases, offering administrators the flexibility needed to optimize resource utilization and deliver a seamless user experience. With the power of multithreading, the load balancer is primed to handle a substantial volume of concurrent requests, ensuring responsiveness even in the face of peak loads. One of the standout features of this project is its ability to manage backend servers dynamically. The ease with which servers can be registered or deregistered, coupled with intelligent health checks, promotes high availability and reliability, safeguarding against server failures and overloads. Scalability is also at the forefront of the design, allowing for seamless growth as traffic and service demands expand. The project's user-friendly configuration options empower administrators to fine-tune load balancing rules, algorithms, and other parameters. Furthermore, its logging and monitoring capabilities offer transparency into system performance and server health, enabling data-driven decisions to optimize efficiency.

REFERENCES

1 https://www.geeksforgeeks.org/design-a-concurrent-server-for-handling-multiple-clients-using-fork/

2 https://www.ibm.com/docs/de/zos/2.1.0?topic=chart-concurrent-iterative-servers

3 https://medium.com/fantageek/understanding-socket-and-port-in-tcp-2213dc2e9b0c

4 https://learn.microsoft.com/en-us/answers/questions/482793/tcp-ip-concurrent-connections