A Course Based Project Report

On

MULTI THREADED PROXY SERVER

Submitted in partial fulfillment of requirement

for the completion of the

Operating Systems Laboratory

II B.Tech Computer Science and Engineering

of

VNR VJIET

By

M.RISHI	**– 23071A05H2**
M.L.KAVYA SAHITHI	**– 23071A05H7**
M.MYTHRI	**– 23071A05H8**
N.APOORVA SAI KARTHIKEY	**– 23071A05H9**
N.HASINI	**– 23071A05J0**

2024-2025

VALLURIPALLI NAGESWARA RAO
VIGNANA JYOTHI INSTITUTE OF ENGINEERING &TECHNOLOGY
(AUTONOMOUS INSTITUTE)
NAAC ACCREDITED WITH 'A++' GRADE
NBA Accreditation for B. Tech Programs
Vignana Jyothi Nagar, Bachupally, Nizampet (S.O), Hyderabad 500090
Phone no: 040-23042758/59/60, Fax: 040-23042761
Email:postbox@vnrvjiet.ac.in Website: www.vnrvjiet.ac.in

A Project Report On

MULTI THREADED PROXY SERVER

Submitted in partial fulfillment of requirement
for the completion of the
Operating Systems Laboratory

II B.Tech Computer Science and Engineering

of

VNRVJIET

2024-2025

Under the Guidance

of

DR. D N VASUNDHARA
Assistant Professor
Department of CSE

VNR VIGNANA JYOTHI INSTITUTE OF ENGINEERING & TECHNOLOGY

(AUTONOMUS INSTITUTE)

NAAC ACCREDITED WITH 'A++' GRADE

CERTIFICATE

This is to certify that the project entitled "MULTI THREADED PROXY SERVER" submitted in partial fulfillment for the course of Operating Systems laboratory being offered for the award of Batch (CSE-C) by VNRVJIET is a result of the bonafide work carried out by **23071A05H2**, **23071A05H7, 23071A05H8, 23071A05H9** and **23071A05J0** during the year **2024-2025**.

This has not been submitted for any other certificate or course.

.

Signature of Faculty **Signature of Head of the Department**

ACKNOWLEDGEMENT

An endeavor over a long period can be successful only with the advice and support of many well-wishers. We take this opportunity to express our gratitude and appreciation to all of them.

We wish to express our profound gratitude to our honorable **Principal Dr. C.D.Naidu** and **HOD Dr.V.Baby**, **CSE Department**, **VNR Vignana Jyothi Institute of Engineering and Technology** for their constant and dedicated support towards our career moulding and development.

With great pleasure we express our gratitude to the internal guide **DR. D.N VASUNDHARA, Assistant Professor, CSE department** for his timely help, constant guidance, cooperation, support and encouragement throughout this project as it has urged us to explore many new things.

Finally, we wish to express my deep sense of gratitude and sincere thanks to our parents, friends and all our well-wishers who have technically and non-technically contributed to the successful completion of this course-based project.

DECLARATION

We hereby declare that this Project Report titled "**MULTI THREADED PROXY SERVER**" submitted by us of Computer Science & Engineering in **VNR Vignana Jyothi Institute of Engineering and Technology**, is a bonafide work undertaken by us and it is not submitted for any other certificate /Course or published any time before.

Name & Signature of the Students:

M.RISHI	**– 23071A05H2**
M.L.KAVYA SAHITHI	**– 23071A05H7**
M.MYTHRI	**– 23071A05H8**
N.APOORVA SAI KARTHIKEY	**– 23071A05H9**
N.HASINI	**– 23071A05J0**

Date:

TABLE OF CONTENTS

ABSTRACT

This Operating Systems project implements a multithreaded proxy server that facilitates communication between a client and a remote web server. Acting as an intermediary, the proxy server listens for incoming HTTP requests, processes them, and forwards them to their intended destination. The project aims to reinforce OS concepts such as socket programming, thread creation, synchronization, and resource sharing, while also introducing performance considerations like caching and concurrent execution.

Two main variants of the proxy server are developed—one that uses a caching mechanism and one that does not. In the non-caching version, the proxy acts as a basic data conduit, transmitting HTTP requests and responses without storing any content. This version emphasizes fundamental socket operations and thread management. The caching version, however, introduces a content storage layer where frequently accessed web pages are saved locally, enabling faster retrieval and reducing redundant external requests. This component exemplifies how OS principles like memory management and file I/O can optimize real-world applications.

Concurrency is handled using POSIX threads (pthreads), where each client connection is managed in a separate thread. This allows the proxy to handle multiple clients in parallel without blocking on individual requests. Thread synchronization techniques, such as mutexes, are applied to ensure safe access to shared resources like the cache. This setup not only demonstrates practical use of multithreading but also mimics the behavior of scalable server architectures in production systems.
Parsing logic, defined in proxy_parse.c and proxy_parse.h, enables the server to dissect HTTP requests and manage headers efficiently. It ensures compatibility with various HTTP clients by correctly interpreting URL formats and headers. Together, the modular architecture, robust parsing, and multithreaded design showcase a comprehensive understanding of operating systems theory, offering a hands-on example of how OS-level abstractions power real-world networking tools.

This project explores the design and implementation of a multithreaded proxy server using core concepts from operating systems. A proxy server acts as an intermediary between clients and destination servers, handling requests and managing responses. The goal is to build a system that supports concurrent client connections using multithreading and optimizes performance through optional caching. By leveraging POSIX threads and socket programming, the project emphasizes concurrency, synchronization, and resource sharing. It also introduces practical use of parsing and memory management techniques. This setup provides a strong foundation for understanding how operating systems support networked applications.

INTRODUCTION

The role of operating systems in managing computer hardware and software resources is fundamental to the development of robust and efficient software systems. One of the key responsibilities of an operating system is managing concurrent processes, which is especially important in networked environments where multiple clients may simultaneously request access to shared services. This project aims to implement a multithreaded proxy server—an intermediary server that receives requests from clients, forwards them to destination servers, and relays the responses back to the clients. Through this project, core operating system concepts such as multithreading, socket programming, synchronization, and memory management are practically explored.

A proxy server plays a vital role in networking by acting as a gateway between users and the internet. It can provide various benefits, including improved performance, access control, and anonymity. This project enhances a traditional proxy server by enabling it to handle multiple client requests concurrently using POSIX threads (pthreads). With multithreading, the server can service multiple requests in parallel, significantly increasing efficiency and reducing client wait time. Each incoming connection is handled in its own thread, which prevents the server from becoming unresponsive under heavy load and reflects the architecture of scalable web services used in production environments.

The project comprises two variants of the proxy server: one that functions without any caching and one that incorporates a simple caching mechanism. The non-caching server is responsible for receiving HTTP requests, parsing them, and forwarding them to the appropriate web server. Upon receiving a response, it relays the data back to the client. This version serves as a foundation for understanding basic thread creation and socket communication. In contrast, the caching version introduces an additional layer of complexity by locally storing previously accessed web content. This cache reduces redundant requests to external servers, saves bandwidth, and speeds up response times for repeated client requests. Implementing caching also touches on memory allocation, file I/O, and data consistency, further deepening the practical application of OS principles.

To support these functionalities, the system uses a modular design with components for HTTP request parsing, thread management, client-server communication, and cache handling. The parsing module ensures that HTTP requests are correctly interpreted, extracting the hostname, path, and headers necessary to generate valid requests to destination servers. This modularity not only enhances code clarity but also supports maintainability and extensibility. Synchronization mechanisms like mutexes are employed to manage access to shared resources, such as the cache, thereby preventing race conditions and ensuring thread safety. These techniques are essential for building reliable multithreaded applications that mirror the behavior of modern-day web services.

In essence, this project bridges theoretical knowledge and practical implementation, offering a comprehensive experience in systems programming. By developing a multithreaded proxy server with and without caching, students gain hands-on exposure to the inner workings of operating systems, particularly in the context of concurrent network applications. The challenges addressed and solutions implemented in this project reflect real-world considerations in server design, making it a valuable exercise in both educational and applied computing environments.

METHODOLOGY

The development of the multithreaded proxy server was carried out in a structured, step-by-step approach to ensure clarity, modularity, and functionality. The methodology includes the design, implementation, and testing of both the caching and non-caching versions of the proxy server. The process can be broken down into the following key steps:

1. **Requirement Analysis and Design Planning:** The project began with identifying the core functionalities needed for a proxy server. These included accepting client connections, parsing HTTP requests, forwarding them to destination servers, and relaying the responses back to clients. Additional requirements included support for concurrent client handling via multithreading and implementing a caching mechanism for performance enhancement. A modular design was chosen to separate key components such as networking, parsing, threading, and caching.

2. **Socket Programming Setup:** The next step was to implement basic socket programming to establish communication between the client and the proxy server. The server creates a listening socket on a specified port and waits for incoming client connections. Upon receiving a connection, the server accepts it and processes the client's HTTP request. This step involved handling low-level TCP socket functions such as socket(), bind(), listen(), accept(), and connect().

3. **Thread Management for Concurrency**: To allow multiple clients to interact with the proxy server simultaneously, POSIX threads (pthreads) were integrated. Each new client connection is handled in a separate thread using pthread_create(). The thread processes the client's request independently, ensuring concurrent processing. Thread synchronization was considered for shared resources like the cache to prevent race conditions, and mutexes were used where appropriate.

4. **HTTP Request Parsing and Forwarding**: A dedicated module was implemented to parse the client's HTTP requests. This module extracts essential information such as the method, host, port, and file path. This parsed data is then used to construct a new request that is forwarded to the appropriate destination server. The response from the remote server is then read and relayed back to the client.

5. **Implementation of Caching Mechanism**: For the caching version of the server, a simple file-based caching system was created. When a request is received, the server first checks if the requested content exists in the cache. If it does, the content is served directly from the cache. Otherwise, the request is forwarded to the external server, and the response is saved to the cache before being sent to the client. This part involved file I/O and hash-based indexing for efficient lookup.

6. **Testing and Optimization**: Finally, the server was tested with multiple client connections to evaluate performance, correctness, and stability. Both versions of the server were benchmarked under different loads to observe the effectiveness of multithreading and caching. Code was refactored for readability and maintainability, ensuring a clean and modular structure.

Through this step-by-step methodology, the project successfully demonstrates the practical implementation of core operating system concepts in a real-world networked application. Each phase—from socket setup to multithreading and caching—was carefully designed to simulate how modern proxy servers operate in practice. By managing concurrency, ensuring thread safety, and optimizing performance through caching, the system reflects both the complexity and necessity of efficient resource handling in OS-based applications. This structured approach not only ensured the functional integrity of the system but also reinforced a deep understanding of how low-level OS mechanisms contribute to high-level application behavior.

OBJECTIVES

The primary objective of the multithreaded proxy server implemented in this project is to efficiently manage HTTP traffic between clients and destination servers, with a focus on concurrency, performance optimization, and modular system design. This system aims to achieve the following specific objectives:

1. **Concurrent Client Request Handling**:
    - To implement a multithreaded architecture that enables the proxy server to handle multiple client requests simultaneously.
    - To utilize POSIX threads (pthreads) for creating and managing individual threads for each client connection.

2. **Efficient HTTP Request Processing:**
    - To receive, parse, and forward HTTP requests from clients to target servers accurately.
    - To reconstruct and transmit HTTP responses back to the requesting clients with minimal delay.

3. **Caching Mechanism for Performance Optimization**:
    - To implement a caching system that stores frequently accessed content locally to reduce redundant requests.
    - To serve repeated requests faster by checking the cache before forwarding requests to the external server.

4. **Modular System Design and Maintainability**:
    - To design the proxy server using modular components (e.g., parsing, threading, caching, communication) for clarity and scalability.
    - To ensure that the code is easy to maintain, debug, and extend for future improvements or features.
5. **Robust HTTP Parsing and URL Resolution**:
    - To extract essential components from HTTP requests such as host, path, and headers.
    - To ensure compatibility with various request formats through accurate parsing and validation techniques.

6. **Thread Synchronization and Safe Resource Access**:
    - To use synchronization tools like mutexes to manage shared resources (e.g., cache) in a thread-safe manner.
    - To avoid race conditions and ensure reliable concurrent execution.

By achieving these objectives, the project delivers a practical, real-world proxy server model that reflects essential operating system concepts. It contributes to a deeper understanding of network programming, concurrency control, and efficient resource management in system-level development.

FLOW OF EXECUTION

Flow Program Of Multi Threaded Proxy Server

1. Start Program

|

V

2.Setup Server

- Accept Client Connections

- Listen for incoming client connections

|

V

3.Accept Client Connections

- Spawn a new thread per client

|

V

4. Thread Handler

- Read and parse client reques

- Check Cache:

- **If cached → send data.**
- **Else → fetch from server**

|

V

5.Update Cache

- Save new response (if space)

- Use LRU policy for eviction

|

V

6. Respond to Client

- Send the data (cached or fetched)

|

V

7.End Program

IMPLEMENTATION OF PROGRAM

Code:

proxy_server_with_cache.c

```c
#include "proxy_parse.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <sys/wait.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define MAX_BYTES 4096    //max allowed size of request/response
#define MAX_CLIENTS 400     //max number of client requests served at a time
#define MAX_SIZE 200*(1<<20)     //size of the cache
#define MAX_ELEMENT_SIZE 10*(1<<20)     //max size of an element in cache

typedef struct cache_element cache_element;

struct cache_element{
    char* data;         //data stores response
    int len;          //length of data i.e., sizeof(data)...
    char* url;        //url stores the request
    time_t lru_time_track;    //lru_time_track stores the latest time the element is  accesed
    cache_element* next;    //pointer to next element
};

cache_element* find(char* url);
int add_cache_element(char* data,int size,char* url);
void remove_cache_element();

int port_number = 8080;             // Default Port
int proxy_socketId;                 // socket descriptor of proxy server
pthread_t tid[MAX_CLIENTS];         //array to store the thread ids of clients
```

proxy_server_with_cache.c

```c
int add_cache_element(char* data,int size,char* url){
    // Adds element to the cache
    // sem_wait(&cache_lock);
    int temp_lock_val = pthread_mutex_lock(&lock);
    printf("Add Cache Lock Acquired %d\n", temp_lock_val);
    int element_size=size+1+strlen(url)+sizeof(cache_element); // Size of the new element which will be added to the cache
    if(element_size>MAX_ELEMENT_SIZE){
        //sem_post(&cache_lock);
        // If element size is greater than MAX_ELEMENT_SIZE we don't add the element to the cache
        temp_lock_val = pthread_mutex_unlock(&lock);
        printf("Add Cache Lock Unlocked %d\n", temp_lock_val);
        // free(data);
        // printf("--\n");
        // free(url);
        return 0;
    }
    else
    {   while(cache_size+element_size>MAX_SIZE){
            // We keep removing elements from cache until we get enough space to add the element
            remove_cache_element();
        }
        cache_element* element = (cache_element*) malloc(sizeof(cache_element)); // Allocating memory for the new cache element
        element->data= (char*)malloc(size+1); // Allocating memory for the response to be stored in the cache element
        strcpy(element->data,data);
        element -> url = (char*)malloc(1+( strlen( url )*sizeof(char)  )); // Allocating memory for the request to be stored in the cache element (as a key)
        strcpy( element -> url, url );
        element->lru_time_track=time(NULL);    // Updating the time_track
        element->next=head;
        element->len=size;
        head=element;
        cache_size+=element_size;
        temp_lock_val = pthread_mutex_unlock(&lock);
        printf("Add Cache Lock Unlocked %d\n", temp_lock_val);
        //sem_post(&cache_lock);
        // free(data);
        // printf("--\n");
        // free(url);
        return 1;
    }
```

proxy_server_with_cache.c

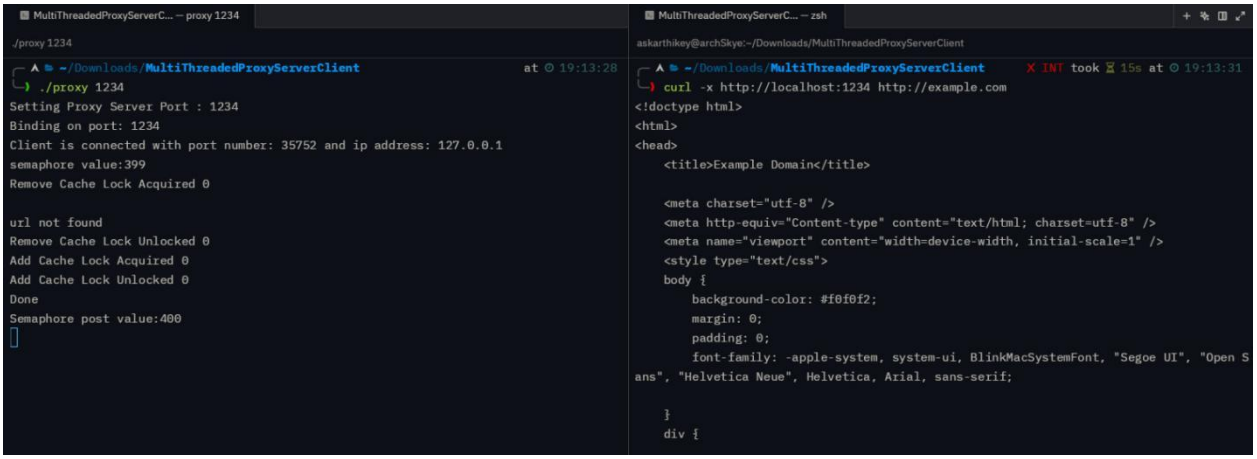
```c
int main(int argc, char * argv[]) {

    int client_socketId, client_len; // client_socketId == to store the client socket id
    struct sockaddr_in server_addr, client_addr; // Address of client and server to be assigned

    sem_init(&seamaphore,0,MAX_CLIENTS); // Initializing seamaphore and lock
    pthread_mutex_init(&lock,NULL); // Initializing lock for cache


    if(argc == 2)        //checking whether two arguments are received or not
    {
        port_number = atoi(argv[1]);
    }
    else
    {
        printf("Too few arguments\n");
        exit(1);
    }

    printf("Setting Proxy Server Port : %d\n",port_number);

    //creating the proxy socket
    proxy_socketId = socket(AF_INET, SOCK_STREAM, 0);

    if( proxy_socketId < 0)
    {
        perror("Failed to create socket.\n");
        exit(1);
    }

    int reuse =1;
    if (setsockopt(proxy_socketId, SOL_SOCKET, SO_REUSEADDR, (const char*)&reuse, sizeof(reuse)) < 0)
        perror("setsockopt(SO_REUSEADDR) failed\n");

    bzero((char*)&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port_number); // Assigning port to the Proxy
    server_addr.sin_addr.s_addr = INADDR_ANY; // Any available adress assigned
```

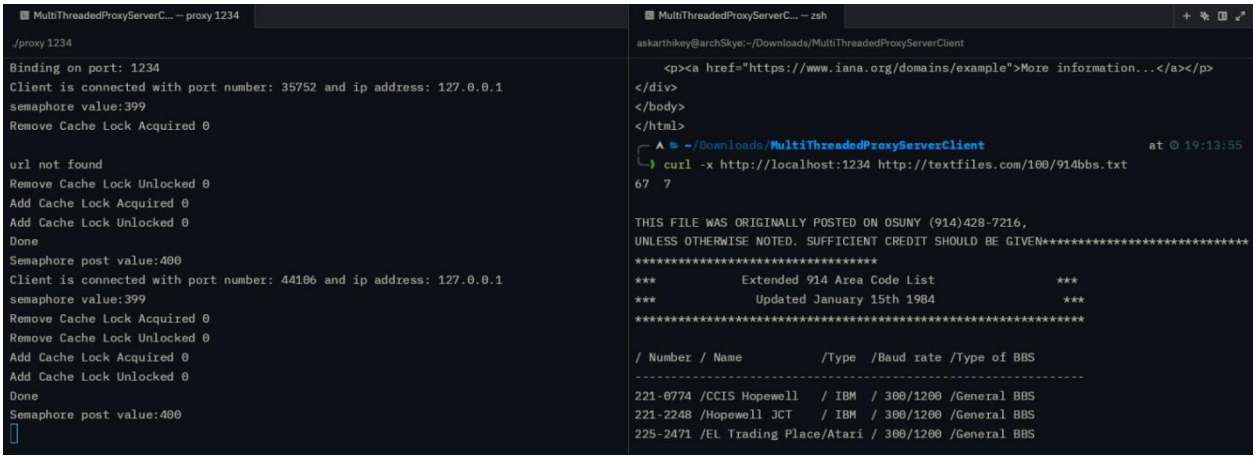
proxy_server_with_cache.c

```c
int handle_request(int clientSocket, ParsedRequest *request, char *tempReq)
{
    char *buf = (char*)malloc(sizeof(char)*MAX_BYTES);
    strcpy(buf, "GET ");
    strcat(buf, request->path);
    strcat(buf, " ");
    strcat(buf, request->version);
    strcat(buf, "\r\n");

    size_t len = strlen(buf);

    if (ParsedHeader_set(request, "Connection", "close") < 0){
        printf("set header key not work\n");
    }

    if(ParsedHeader_get(request, "Host") == NULL)
    {
        if(ParsedHeader_set(request, "Host", request->host) < 0){
            printf("Set \"Host\" header key not working\n");
        }
    }

    if (ParsedRequest_unparse_headers(request, buf + len, (size_t)MAX_BYTES - len) < 0) {
        printf("unparse failed\n");
        //return -1;               // If this happens Still try to send request without header
    }

    int server_port = 80;              // Default Remote Server Port
    if(request->port != NULL)
        server_port = atoi(request->port);

    int remoteSocketID = connectRemoteServer(request->host, server_port);

    if(remoteSocketID < 0)
        return -1;

    int bytes_send = send(remoteSocketID, buf, strlen(buf), 0);

    bzero(buf, MAX_BYTES);
```

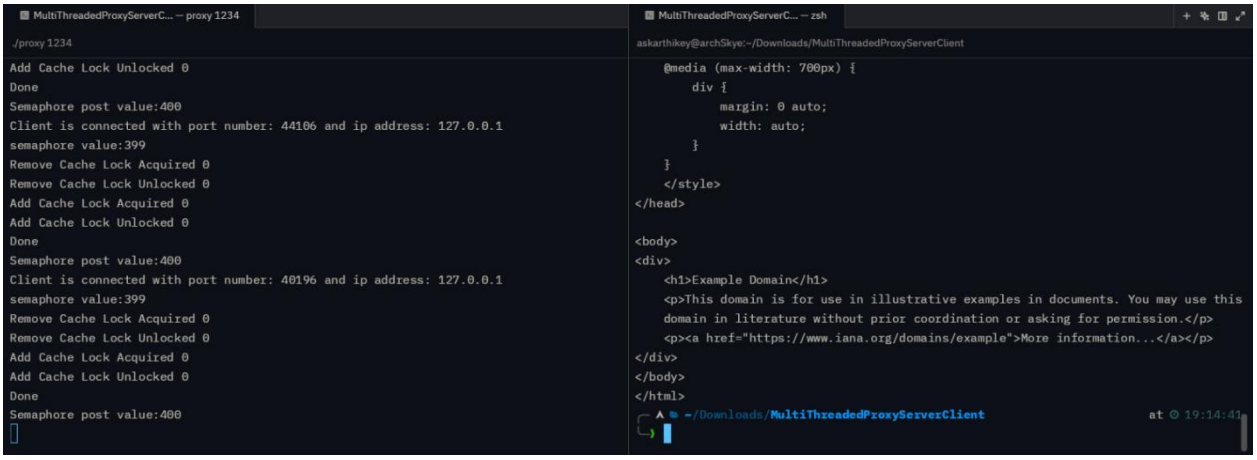
Output:

```
MultiThreadedProxyServerC... — proxy 1234
./proxy 1234
~/Downloads/MultiThreadedProxyServerClient                    at 19:07:25
❯ make all
make: Nothing to be done for 'all'.
~/Downloads/MultiThreadedProxyServerClient                    at 19:11:55
❯ ./proxy 1234
Setting Proxy Server Port : 1234
Binding on port: 1234
Client is connected with port number: 43306 and ip address: 127.0.0.1
semaphore value:399
Remove Cache Lock Acquired 0

url not found
Remove Cache Lock Unlocked 0
Add Cache Lock Acquired 0
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
```


MultiThreadedProxyServerC... — proxy 1234
./proxy 1234
~/Downloads/MultiThreadedProxyServerClient at 19:13:28
./proxy 1234
Setting Proxy Server Port : 1234
Binding on port: 1234
Client is connected with port number: 35752 and ip address: 127.0.0.1
semaphore value:399
Remove Cache Lock Acquired 0
url not found
Remove Cache Lock Unlocked 0
Add Cache Lock Acquired 0
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
MultiThreadedProxyServerC... — zsh
askarthikey@archSkye:~/Downloads/MultiThreadedProxyServerClient
~/Downloads/MultiThreadedProxyServerClient X INT took 15s at 19:13:31
curl -x http://localhost:1234 http://example.com
<!doctype html>
<html>
<head>
<title>Example Domain</title>
<meta charset="utf-8" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<style type="text/css">
body {
background-color: #f0f0f2;
margin: 0;
padding: 0;
font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open S
ans", "Helvetica Neue", Helvetica, Arial, sans-serif;
}
div {


MultiThreadedProxyServerC... — proxy 1234
./proxy 1234
Binding on port: 1234
Client is connected with port number: 35752 and ip address: 127.0.0.1
semaphore value:399
Remove Cache Lock Acquired 0
url not found
Remove Cache Lock Unlocked 0
Add Cache Lock Acquired 0
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
Client is connected with port number: 44106 and ip address: 127.0.0.1
semaphore value:399
Remove Cache Lock Acquired 0
Remove Cache Lock Unlocked 0
Add Cache Lock Acquired 0
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
MultiThreadedProxyServerC... — zsh
askarthikey@archSkye:~/Downloads/MultiThreadedProxyServerClient
More information...

</body>
</html>
~/Downloads/MultiThreadedProxyServerClient at 19:13:55
curl -x http://localhost:1234 http://textfiles.com/100/914bbs.txt
67 7
THIS FILE WAS ORIGINALLY POSTED ON OSUNY (914)428-7216,
UNLESS OTHERWISE NOTED. SUFFICIENT CREDIT SHOULD BE GIVEN**********
****
*** Extended 914 Area Code List ***
*** Updated January 15th 1984 ***
/ Number / Name /Type /Baud rate /Type of BBS
221-0774 /CCIS Hopewell / IBM / 300/1200 /General BBS
221-2248 /Hopewell JCT / IBM / 300/1200 /General BBS
225-2471 /EL Trading Place/Atari / 300/1200 /General BBS


MultiThreadedProxyServerC... — proxy 1234
./proxy 1234
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
Client is connected with port number: 44106 and ip address: 127.0.0.1
semaphore value:399
Remove Cache Lock Acquired 0
Remove Cache Lock Unlocked 0
Add Cache Lock Acquired 0
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
Client is connected with port number: 40196 and ip address: 127.0.0.1
semaphore value:399
Remove Cache Lock Acquired 0
Remove Cache Lock Unlocked 0
Add Cache Lock Acquired 0
Add Cache Lock Unlocked 0
Done
Semaphore post value:400
MultiThreadedProxyServerC... — zsh
askarthikey@archSkye:~/Downloads/MultiThreadedProxyServerClient
@media (max-width: 700px) {
div {
margin: 0 auto;
width: auto;
}
}
</style>
</head>
<body>

<h1>Example Domain</h1>
This domain is for use in illustrative examples in documents. You may use this
domain in literature without prior coordination or asking for permission.
More information...

</body>
</html>
~/Downloads/MultiThreadedProxyServerClient at 19:14:41

CONCLUSION

This project demonstrates the design and implementation of a multithreaded proxy server with and without caching mechanisms, serving as a practical application of core operating systems concepts such as multithreading, synchronization, socket programming, and memory management. The primary goal of the project was to act as an intermediary between clients and web servers, efficiently managing multiple client requests while offering enhanced performance through intelligent caching strategies.

The proxy server is capable of handling multiple simultaneous client connections using POSIX threads, ensuring concurrency and responsiveness. Each incoming client request is handled in its own thread, allowing the server to operate in parallel and efficiently manage numerous users without significant delay. This use of multithreading showcases how real-world servers maintain scalability and stability under varying network loads.

A significant enhancement in the project is the integration of a caching system, which stores frequently accessed web pages in memory to minimize redundant network requests. This not only reduces the latency experienced by clients but also lowers the overall network bandwidth usage. The cache is maintained using a Least Recently Used (LRU) policy to keep the memory usage optimal, evicting the least accessed content when space is required. Synchronization mechanisms like mutexes or semaphores are used to ensure thread-safe access to shared resources such as the cache, avoiding race conditions and maintaining data consistency.

Through this project, key operating system principles such as inter-process communication, concurrency control, and resource sharing are explored in a hands-on manner. The design also highlights the importance of performance optimization in network-based applications, particularly in scenarios where response time and throughput are critical.
Furthermore, the implementation without caching serves as a baseline to compare the performance benefits introduced by the caching mechanism. By analyzing request handling times and server load, the project makes a compelling case for intelligent caching in real-world proxy servers.

In summary, this proxy server project not only reinforces theoretical knowledge in systems programming but also delivers a functional tool that can be extended for more advanced use cases, such as support for HTTPS, access logging, or adaptive caching policies. It is a valuable learning exercise in building robust, efficient, and scalable network applications with a strong foundation in operating systems.

FUTURE SCOPE

Looking ahead, the multithreaded proxy server developed in this project holds substantial potential for advancement and real-world deployment in high-performance networking environments. As digital connectivity scales rapidly and web traffic continues to grow, efficient and intelligent proxy systems are increasingly vital. Here are several futuristic directions in which this project can evolve:

1. **HTTPS and SSL Support:** Currently focused on HTTP, the proxy server can be extended to handle secure HTTPS traffic. By implementing SSL/TLS tunneling and certificate handling, the system can securely proxy encrypted communications, making it suitable for modern web applications and enterprise use.

2. **Improved Caching Techniques:** Beyond the current LRU policy, more sophisticated caching strategies—like LFU (Least Frequently Used), time-based eviction, or adaptive cache size management—could be implemented to further optimize performance and hit rates.

3. **Load Balancing Capabilities:** Features such as access control, IP blocking, rate limiting, and encrypted cache storage could be added to make the proxy server more secure against misuse and cyber threats.

4. **Security Enhancements:** Features such as access control, IP blocking, rate limiting, and encrypted cache storage could be added to make the proxy server more secure against misuse and cyber threats.

5. **Integration with Cloud Services:** Deploying the proxy server in cloud environments (e.g., AWS, Azure) can offer high availability and scalability on-demand. Containerization using Docker and orchestration via Kubernetes would further enhance portability and ease of deployment.

6. **Security and Data Privacy**: Implementing features such as rate limiting, IP blacklisting, and encrypted caching will strengthen the server's security posture, protecting both users and backend servers from potential attacks.

In conclusion, with these enhancements, the proxy server can evolve into a more powerful, secure, and scalable tool suitable for educational, enterprise, or research use.

REFERENCES

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
  – Reference for operating system fundamentals, threads, synchronization, and concurrency control.
- Stevens, W. R., & Rago, S. A. (2013). *Advanced Programming in the UNIX Environment* (3rd ed.). Addison-Wesley.
  – Covers detailed implementation of socket programming and multithreaded network applications.
- Comer, D. E. (2018). *Computer Networks and Internets* (6th ed.). Pearson.
  – Provides background on client-server models and network protocols relevant to proxy servers.
- Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.
  – Useful for understanding concurrency, multithreading, process communication, and synchronization.
- Mozilla Developer Network (MDN). (n.d.). *HTTP Proxy*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling
  – Practical reference on how HTTP proxy servers work and their role in web communication.