

A SYNOPSIS ON

---

---

# MULTI-THREAD WEB CHAT

---

---

Submitted in partial fulfilment of the requirement for the award of the degree of

**BACHELOR OF TECHNOLOGY**

**In**

**Computer Science & Engineering**

**Submitted by:**

Sagar singh                      **University Roll No. - 2261500**

Ayushman Rawat                **University Roll No. - 2261138**

Bishal Singh Mehra            **University Roll No. - 2261147**

Ashish Singh bisht              **University Roll No. - 2261120**

*Under the Guidance of*

*Supervisor Name – Mr. Ansh Dhingra*

*Assistant Professor*

**Project Team ID: 73**



Department of Computer Science & Engineering

**Graphic Era Hill University, Bhimtal, Uttarakhand**

## CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the Synopsis entitled “**MULTI-THREAD WEB SERVER**” in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science & Engineering of the Graphic Era Hill University, Bhimtal campus and shall be carried out by the undersigned under the supervision of **Mr. Ansh Dhingra** Assistant Professor , Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Sagar singh	2261500	signature
Ayushman Rawat	2261138	signature
Bishal Singh Mehra	2261147	signature
Ashish Singh bisht	2261120	signature

The above mentioned students shall be working under the supervision of the undersigned on the “**MULTI-THREAD WEB CHAT**”

Signature  
Supervisor

Signature  
Head of the Department

### Internal Evaluation (By DPRC Committee)

**Status of the Synopsis:** Accepted / Rejected

**Any Comments:**

**Name of the Committee Members:**

**Signature with Date**

- 1.
- 2.

## Table of Contents

<b>Chapter No.</b>	<b>Description</b>	<b>Page No.</b>
Chapter 1	Introduction and Problem Statement	4 to 5
Chapter 2	Background/ Literature Survey	6 to 7
Chapter 3	Objectives	8 to 9
Chapter 4	Hardware and Software Requirements	10 to 11
Chapter 5	Possible Approach/ Algorithms	12 to 14
	References	

# Chapter 1

## 1. Introduction

A web server is responsible for handling client requests and serving responses over a network. Traditional single-threaded web servers can become inefficient when handling multiple simultaneous requests. This project focuses on implementing a **multi-threaded web server using Python** to improve concurrency and responsiveness. By utilizing **Python's threading library and the Sockets API**, the server can handle multiple client connections concurrently, ensuring faster and more efficient request processing.

Web servers are essential components of modern networking, enabling communication between clients and servers over the internet. A traditional **single-threaded server** processes one request at a time, which can lead to significant delays when handling multiple users simultaneously. To address this issue, this project implements a **multi-threaded web server using Python**, leveraging **POSIX Threads (pthreads) and Python's threading module** to efficiently manage multiple client connections.

The core of this web server relies on **the Sockets API**, which facilitates client-server communication over a network. By integrating **multi-threading and process synchronization** concepts, the server can manage multiple incoming requests without blocking execution. This approach improves response times and ensures better resource utilization.

The project will focus on:

- **Threading and Concurrency:** Using **Python's threading module** to create a concurrent server that can handle multiple clients simultaneously.
- **Networking with Sockets:** Implementing client-server communication using the **Sockets API**, enabling efficient data transfer.
- **Process Synchronization:** Ensuring thread safety and preventing race conditions while handling multiple requests.

This multi-threaded approach makes the web server **scalable, efficient, and capable of handling multiple concurrent users**, providing a strong foundation for real-world web applications.

## 2. Problem Statement

In today's digital world, web servers play a critical role in handling client requests and serving data over the internet. However, **traditional singlethreaded web servers** suffer from significant performance limitations when dealing with multiple simultaneous client requests. Since a single-threaded server processes only **one request at a time**, additional requests must wait in a queue, leading to **increased response times** and **poor scalability**.

To address this issue, the project focuses on developing a **multi-threaded web server using Python**, leveraging **POSIX Threads (pthreads) / Python threading** along with **the Sockets API for networking**. By implementing a **concurrent processing model**, the server can handle multiple client connections simultaneously, improving **throughput, responsiveness, and resource utilization**.

This project aims to solve the following key problems:

1. **Inefficiency of Single-Threaded Servers** – Traditional web servers struggle under high traffic loads, leading to slow response times.
2. **Concurrency Management** – Implementing multi-threading in Python to efficiently handle multiple client requests.
3. **Process Synchronization Challenges** – Ensuring proper thread synchronization to avoid race conditions and data inconsistencies.
4. **Scalability** – Creating a scalable architecture that allows the server to handle multiple requests dynamically without performance degradation.

By implementing **multi-threading, networking, and process synchronization concepts**, this project will provide a **more efficient and robust web server** capable of handling high concurrent loads, making it a practical solution for real-world applications.

## Chapter 2 Background/ Literature Survey

Web servers are fundamental components of the internet, responsible for processing and responding to client requests. Traditionally, **single-threaded web servers** were widely used, handling one request at a time in a sequential manner. However, with the **increasing demand for high-performance web applications**, this approach became inefficient, leading to slow response times and scalability issues.

To address these challenges, researchers and developers have explored **multithreading techniques** in web server architecture. Multi-threading allows a server to handle multiple client requests **concurrently**, improving efficiency and reducing latency. Several key technologies and concepts have contributed to the evolution of multi-threaded web servers:

1. **Threading in Web Servers** ◦ Early web servers, such as **Apache HTTP Server**, initially relied on a process-based model, spawning a new process for each client request. However, this approach consumed significant system resources. ◦ Later, **thread-based models** were introduced, utilizing lightweight threads to handle requests, reducing memory overhead and improving performance.
  - The introduction of **POSIX Threads (pthreads)** provided efficient thread management and synchronization mechanisms, allowing servers to scale effectively.
2. **Networking with Sockets API** ◦ **Sockets** have been a core technology in client-server communication, enabling data transmission over networks. ◦ **Traditional blocking sockets** required servers to wait for I/O operations, limiting concurrency. However, with advancements such as **non-blocking and asynchronous I/O**, web servers became more responsive.
  - Python's **socket library** provides built-in support for handling network connections, making it a suitable choice for implementing a multi-threaded web server.
3. **Process Synchronization challenges**
  - With multi-threading, ensuring **safe access to shared resources** is crucial to prevent race conditions and inconsistencies.
  - Techniques such as **mutex locks, semaphores, and condition variables** (available in pthreads and Python's threading module)

are widely used to synchronize threads. ○ Studies on **thread-pooling techniques** have shown improved server performance by pre-creating threads instead of spawning them dynamically.

4. **Modern Multi-Threaded Web Servers** ○ Contemporary web servers, such as **NGINX and Apache**, use optimized multi-threading and asynchronous processing to handle large-scale web traffic efficiently.
  - The adoption of **event-driven architectures** (e.g., using **Python's asyncio** or **Node.js**) further enhances scalability by reducing thread management overhead.

## Relevance to This Project

This project builds upon the advancements in **multi-threaded web servers** by utilizing:

- **Python's threading module and POSIX Threads (pthreads)** to handle multiple requests concurrently.
- **Sockets API** for efficient client-server communication.
- **Process synchronization mechanisms** to ensure thread safety and performance optimization.

By integrating these technologies, this project aims to develop a **scalable and efficient multi-threaded web server**, addressing the limitations of singlethreaded models and contributing to the growing field of concurrent computing.

## Chapter 3 Objectives

The primary objective of this project is to design and develop a **multi-threaded web server using Python** that efficiently handles multiple client requests concurrently. The project aims to overcome the limitations of traditional singlethreaded web servers by leveraging **multi-threading, networking, and process synchronization** techniques. The key objectives include:

1. **Develop a Multi-Threaded Web Server** ◦ Implement a **multi-threaded architecture** using **Python's threading module** and **POSIX Threads (pthreads)** to enable concurrent request handling.
2. **Efficient Client-Server Communication** ◦ Utilize the **Sockets API** to establish reliable and efficient communication between the server and multiple clients.
3. **Improve Performance and Scalability** ◦ Ensure that the web server can handle **multiple concurrent connections** without significant performance degradation.
  - Optimize resource utilization to support **scalability** for real-world applications.
4. **Implement Process Synchronization Mechanisms** ◦ Use synchronization techniques such as **mutex locks, semaphores, and thread-safe data structures** to manage shared resources safely.
5. **Ensure Reliability and Fault Tolerance** ◦ Handle **client disconnections, timeouts, and exceptions** gracefully to prevent server crashes.
6. **Compare Performance with Single-Threaded Servers** ◦ Analyze and compare the performance of the multi-threaded server against a **single-threaded implementation**, focusing on response time and request handling capacity.
7. **Enhance Security Measures** ◦ Implement basic **security features** such as request validation, error handling, and controlled resource access to prevent malicious activities.

By achieving these objectives, the project will result in a **robust, efficient, and scalable web server** capable of handling multiple concurrent client requests effectively.



## Chapter 4

# Hardware and Software Requirements

### 4.1 Hardware Requirements

Sl. No	Name of the Hardware	Specification
1.	Processor	Intel Core i3 (or equivalent) and above
2.	RAM	4 GB (8 GB recommended)
3.	Storage	20 GB free disk space
4.	Network Adapter	Ethernet/Wi-Fi for network communication
5.	Operating System	Windows 10/11, Linux (Ubuntu, Fedora), macOS

## 4.2 Software Requirements

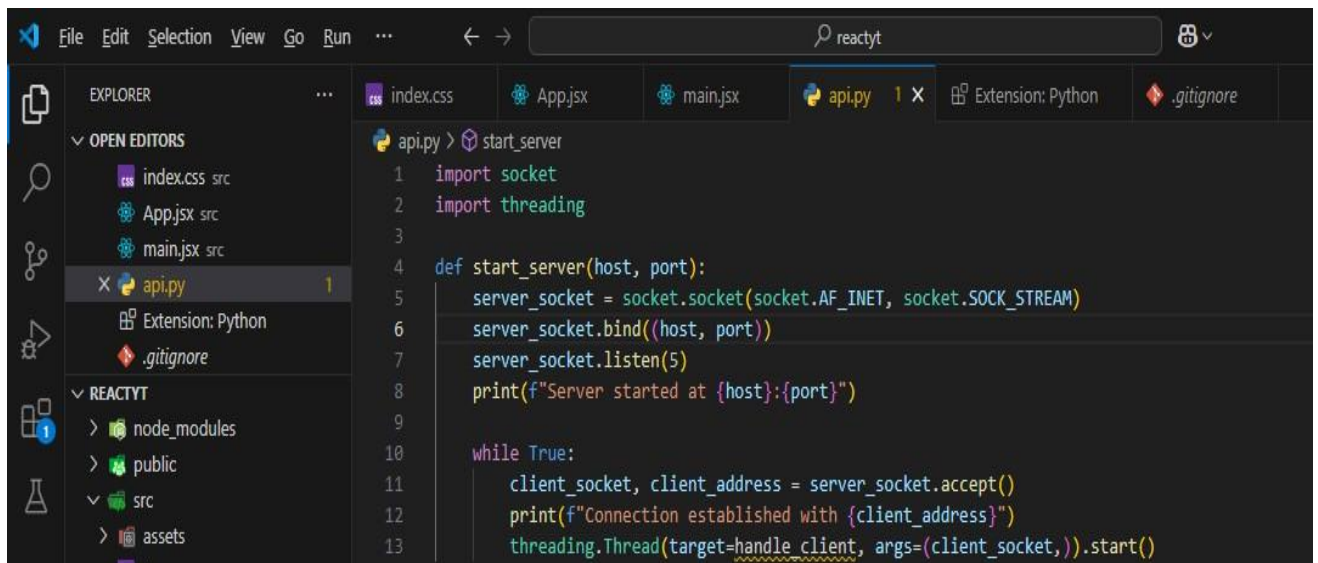
Sl. No	Name of the Software	Specification
1.	Programming Language	Python (version 3.x)
2.	Threading Library	POSIX Threads (pthreads) / Python threading module
3.	Networking	Sockets API for client-server communication
4.	IDE/Text Editor	PyCharm, VS Code, Sublime Text, or any preferred editor
5.	Operating System	Windows, Linux (Ubuntu recommended), or macOS
6.	Python Libraries	<code>socket, threading, sys, os</code>

## Chapter 5

### Possible Approach/ Algorithms

- **Initialize the Server**
  - Create a **server socket** using the **Sockets API** to listen for incoming client connections.
  - Bind the server to a specific **IP address and port** and start listening for client requests.
- **Accept Client Connections**
  - Accept incoming client requests using the **accept()** method.
  - Each request is assigned to a new **thread** for concurrent processing.
- **Handle Client Requests Concurrently**
  - Use **Python's threading module** or **POSIX Threads (pthreads)** to create multiple threads.
  - Each thread processes a client request independently, preventing blocking.
- **Process and Respond to Requests** □ Read client data from the socket.
  - Perform necessary processing (e.g., handling HTTP requests if implementing an HTTP server).
  - Send an appropriate response back to the client.
- **Ensure Synchronization & Resource Management**
  - Use **mutex locks or semaphores** to synchronize shared resources. □ Implement **thread-safe data structures** if required.
- **Handle Errors and Termination**
  - Implement **exception handling** for unexpected client disconnections.

## Algorithm 1: Server Initialization and Listening



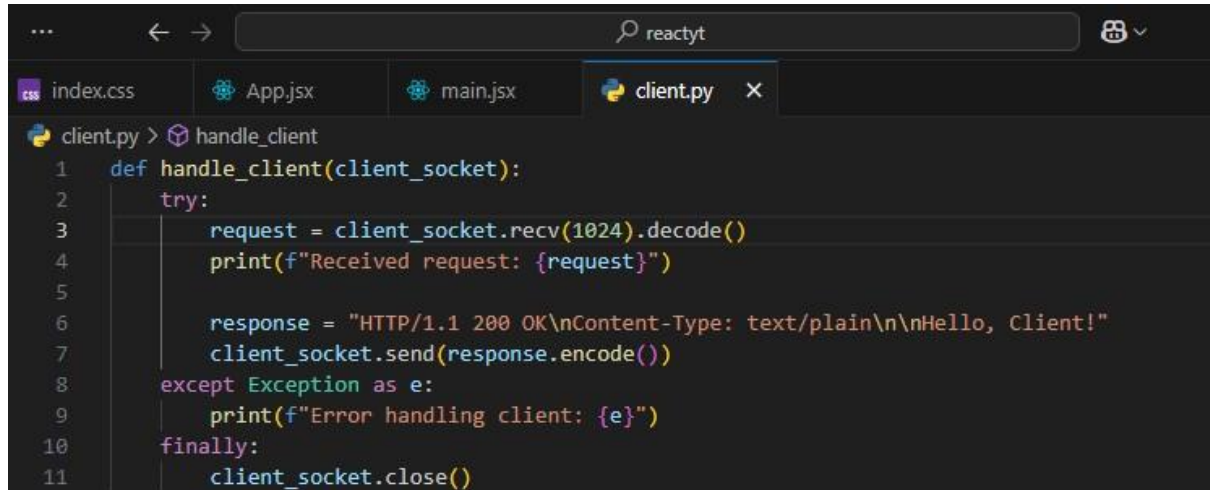
The screenshot shows a code editor with a dark theme. The Explorer panel on the left shows a project structure with folders like 'node\_modules', 'public', 'src', and 'assets'. The 'api.py' file is open in the editor, showing the following code:

```
api.py > start_server
1  import socket
2  import threading
3
4  def start_server(host, port):
5      server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6      server_socket.bind((host, port))
7      server_socket.listen(5)
8      print(f"Server started at {host}:{port}")
9
10     while True:
11         client_socket, client_address = server_socket.accept()
12         print(f"Connection established with {client_address}")
13         threading.Thread(target=handle_client, args=(client_socket,)).start()
```

This code **implements a multi-threaded web server** using **Python's sockets and threading**. It:

- **Creates a server socket** to listen for incoming client connections.
- **Accepts multiple clients** and assigns each to a separate thread for concurrent handling.
- **Ensures scalability** by allowing multiple requests to be processed simultaneously.

## Algorithm 2: Handling client Requests (Multi-Threading Implementation)



The screenshot shows a code editor with a dark theme. The top bar includes a search icon and the text 'reactyt'. Below the top bar, there are tabs for 'index.css', 'App.jsx', 'main.jsx', and 'client.py'. The 'client.py' tab is active, showing the following Python code:

```
client.py > handle_client
1 def handle_client(client_socket):
2     try:
3         request = client_socket.recv(1024).decode()
4         print(f"Received request: {request}")
5
6         response = "HTTP/1.1 200 OK\nContent-Type: text/plain\n\nHello, Client!"
7         client_socket.send(response.encode())
8     except Exception as e:
9         print(f"Error handling client: {e}")
10    finally:
11        client_socket.close()
```

This function **handles individual client requests** in the multi-threaded web server. It:

- **Receives data** from the client via the socket.
- **Processes the request** and sends an HTTP response (Hello, Client!).
- **Handles errors** to prevent crashes.
- **Closes the connection** after processing.