A SYNOPSIS ON

Multithreaded Social Media Web Application

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

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted by:

Himanshu Joshi	2261267
Hem Chandra Joshi	2261263
Devyanshu Suyal	2261181
Mayank Singh Negi	2261362

Under the Guidance of

Mr. Prince Kumar

Assistant Professor

Project Team ID: 84

Department of Computer Science & Engineering

Graphic Era Hill University, Bhimtal, Uttarakhand

March-2025

Graphic Era Hill University
BHIMTAL CAMPUS

CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the Synopsis entitled **"Multithreaded Social Media Web Application"** 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 **Prince Kumar , Assistant Professor ,**Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Himanshu Joshi	**2261267**
Hem Chandra Joshi	**2261263**
Devyanshu Suyal	**2261181**
Mayank Singh Negi	**2261362**

The above mentioned students shall be working under the supervision of the undersigned on the **"Multithreaded Social Media Web Application"**

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

Chapter 1

Introduction and Problem Statement

1. Introduction

In the modern era of digital communication, social media platforms play a pivotal role in connecting people, sharing information, and fostering online communities. Users expect these platforms to be highly responsive, scalable, and capable of handling real-time interactions such as chatting, media sharing, feed browsing, and notifications.

Traditional single-threaded server architectures struggle with scalability, often leading to delays and performance bottlenecks when handling multiple concurrent requests. These limitations make it challenging to develop real-time social media applications that require seamless, high-speed interactions. The key challenges include handling multiple requests efficiently, avoiding race conditions, ensuring real-time data synchronization, and maintaining optimal server performance.

To address these challenges, this project introduces **SocialSync**, a **Multi-Threaded Social Media Web Application** that leverages multithreading, socket programming, and real-time communication to enhance concurrency, responsiveness, and overall system efficiency. The system is designed to handle multiple client requests in parallel, providing seamless interactions for users. The integration of OS concepts such as process management, thread synchronization, and resource allocation is crucial in achieving the desired performance.

2. Problem Statement

Social media applications operate in highly concurrent environments where users perform multiple actions simultaneously, including messaging, browsing feeds, uploading media, and interacting with posts. However, building an efficient and scalable social media platform presents several challenges:

1. **Efficiently handling multiple client requests** without blocking or slowing down the system.
2. **Providing real-time communication and notifications** with minimal latency.
3. **Avoiding server slowdowns** during peak usage hours.
4. **Preventing race conditions** when multiple users update shared data (likes, comments, follows, etc.).
5. **Ensuring thread synchronization** to maintain data consistency across the platform.
6. **Maintaining server responsiveness** while executing I/O-intensive tasks such as file uploads and database queries.

This project aims to develop a multi-threaded web server and a scalable social media web application that enables:

- **Seamless concurrent user interactions** (browsing, messaging, uploading media).
- **Real-time chat and notification systems** using efficient threading mechanisms.
- **Optimized data handling and synchronization** for a smooth user experience.
- **Scalable performance** suitable for real-world deployment.

Chapter 2

Background and Literature Survey

1. Evolution of Social Media Systems

Social media platforms have evolved significantly from static web forums to dynamic, high-performance systems capable of handling millions of users simultaneously. Early platforms followed a **client-server model** with limited interaction capabilities, focusing on simple user communication.

- **Single-threaded models:** Initially, many web applications used single-threaded servers, processing one request at a time. This led to slow performance and poor scalability under heavy loads.
- **Multi-threaded architectures:** With increasing users and demand for real-time interaction, platforms like Facebook, Twitter, and WhatsApp adopted multithreading to enhance concurrent request handling.
- **Asynchronous and event-driven architectures:** Modern platforms integrate event-driven programming and non-blocking I/O (e.g., Node.js, WebSockets) to achieve real-time responsiveness and reduce latency.

2. Importance of Multithreading in Social Media Applications

Multithreading is crucial in social media applications due to their highly interactive nature. Without multithreading, platforms would struggle to handle simultaneous user interactions efficiently. Key benefits include:

- **Concurrency Management:** Multiple user requests can be processed simultaneously, ensuring smooth performance.
- **Server Efficiency:** The workload is distributed across multiple threads, preventing system overload.
- **Improved User Experience:** Faster response times, real-time updates, and seamless navigation enhance engagement.
- **Consistency and Data Integrity:** Thread synchronization techniques, such as mutexes and semaphores, prevent race conditions and ensure data consistency.

3. Related Work and Research

Numerous research studies and real-world implementations highlight the benefits of multithreading in web applications. Some key findings include:

- **Real-time Chat Systems:** Research on WebSockets and asynchronous communication has shown a significant reduction in latency for real-time messaging.

- **Thread-Pool Optimization:** Studies suggest that dynamic thread-pool allocation based on workload analysis improves server efficiency.
- **Distributed Systems and Load Balancing:** Many large-scale social media platforms use distributed multithreaded architectures to balance traffic and prevent server overload.

4. Relevant Technologies and Techniques

To develop an efficient multithreaded social media platform, the following technologies and techniques will be employed:

1. **Socket Programming** – Enables real-time communication through WebSockets and TCP persistent connections.
2. **Thread Synchronization** – Mutexes, semaphores, and condition variables help manage concurrent access to shared data.
3. **Load Balancing** – Distributes network traffic across multiple servers to optimize performance and prevent bottlenecks.
4. **Non-Blocking I/O** – Reduces waiting time for I/O operations, enhancing responsiveness and system efficiency.
5. **Database Optimization** – Techniques such as indexing, caching, and sharding improve data retrieval speeds in high-traffic environments.

This project will incorporate these modern concurrency models to build a scalable and efficient social media web application that delivers real-time interactions with optimal performance.

Chapter 3

Objectives

The primary goal of this project is to develop a **multi-threaded social media web application** that ensures high responsiveness, scalability, and real-time interactions for users. To achieve this, the following detailed objectives have been outlined:

---

1. Develop a Scalable, Multi-Threaded Web Server

Why is this important?

A traditional single-threaded server processes one request at a time, which results in significant delays when handling multiple simultaneous users. In contrast, a **multi-threaded server** allows multiple requests to be handled concurrently, improving response time and scalability.

Technical Aspects:

- Implement a **thread pool** to manage multiple connections efficiently.
- Use **multi-threading and process management** to distribute tasks effectively.
- Optimize server response time to handle user requests, media uploads, and notifications simultaneously.
- Prevent **server bottlenecks** by balancing load distribution across multiple threads.

Expected Impact:

- The system will support thousands of concurrent users without noticeable delays.
- Users will experience **fast page loading, quick media uploads, and seamless interactions** even during peak traffic.
- Server resources will be utilized **efficiently**, preventing overload and crashes.

2. Implement Real-Time Features (Live Chat & Notifications)

Why is this important?

In modern social media applications, **real-time communication** is crucial for user engagement. Features like instant messaging, notifications, and status updates must happen **without delays** to maintain a smooth user experience.

Technical Aspects:

- Use **WebSockets or TCP-based socket programming** to enable real-time interactions.
- Implement **event-driven architecture** where changes (new messages, likes, comments) are pushed instantly to connected clients.

- Optimize **message queuing and delivery** for reliable real-time chat.
- Ensure **low-latency communication** for instant feedback.

Expected Impact:

- Users will receive **instant notifications** for likes, comments, and messages.
- The chat system will work **seamlessly**, eliminating waiting times.
- The platform will function efficiently **even with thousands of real-time users**.

3. Optimize Performance for Media Uploads & Feed Rendering

Why is this important?

Social media applications rely heavily on **image and video sharing**. Slow media processing can **negatively impact user engagement**. Optimizing performance for uploads and feed rendering ensures smooth content consumption.

Technical Aspects:

- Implement **asynchronous I/O operations** for non-blocking media uploads.
- Use **CDN (Content Delivery Network) caching** to reduce media load times.
- Optimize database queries to ensure **fast content retrieval**.
- Load feeds using **lazy loading and pagination** to prevent excessive data fetching.

Expected Impact:

- **Instant media uploads** without affecting overall system performance.
- Smooth browsing experience with **fast-loading feeds and images**.
- Optimized **storage and retrieval** of large media files.

4. Apply Synchronization Mechanisms (Mutexes, Semaphores) to Prevent Race Conditions

Why is this important?

Race conditions occur when multiple threads **access and modify shared data simultaneously**, leading to inconsistent or incorrect results. Without proper synchronization, issues such as **duplicate likes, missing notifications, and incorrect post updates** may occur.

Technical Aspects:

- Implement **mutexes (mutual exclusion)** to control access to shared resources (e.g., post likes, comments, chat messages).

- Use **semaphores** to manage concurrent execution when multiple users interact with the same post or chat group.
- Ensure **atomic transactions** in the database to maintain data integrity.
- Prevent **data inconsistency** in scenarios where multiple users modify the same information simultaneously.

Expected Impact:

- No duplicate actions (e.g., multiple like counts for a single click).
- Database transactions remain **consistent and error-free**.
- Users experience **accurate and up-to-date information** in real-time.

5. Design an Interactive, User-Friendly Front-End

Why is this important?

A good user experience (UX) is crucial for engagement and retention. Users should be able to **easily navigate the platform**, interact with others, and receive real-time feedback **without confusion or frustration**.

Technical Aspects:

- Implement a **responsive UI** using modern frameworks like React.js.
- Use **efficient state management** for dynamic updates (e.g., Redux, Context API).
- Integrate **WebSockets for live updates** without requiring full page reloads.
- Optimize front-end rendering for seamless performance.

Expected Impact:

- Users will have a **smooth, responsive experience** with instant updates.
- The interface will be **mobile-friendly and intuitive**.
- The platform will function efficiently **even with complex real-time interactions**.

Chapter 4

Hardware Requirements

Component	Minimum Requirement	Recommended Requirement
Processor	Dual-Core 2.0 GHz	Quad-Core or higher
RAM	4 GB	8 GB or higher
Storage	10 GB free disk space	20 GB SSD or higher
Network	Stable Internet Connection	Broadband / LAN
Display	Standard HD Monitor	Full HD Monitor
Optional	GPU (if image processing is included)	GPU (NVIDIA GTX series or higher)

Software Requirements

Software	Description
Operating System	Windows / Linux / macOS (Linux Preferred)
Programming Languages	Python / C++ / Java (Any one for Backend)
Thread Library	Python threading module / Java concurrency utilities
Database	PostgreSQL / MySQL / MongoDB
Web Framework	Flask / FastAPI (Python) or Custom Socket Programming
Front-End	HTML, CSS, JavaScript, ReactJS
Real-Time Communication	Socket.IO or WebSocket
Version Control	Git
IDE	VS Code / PyCharm / Eclipse / IntelliJ
Other Tools	Postman (for API testing), Browser (Chrome / Firefox)

Chapter 5

Possible Approach/ Algorithms

1. Multi-Threaded Server Algorithm

Objective:

To efficiently handle multiple user requests (e.g., login, chat, media uploads) concurrently without blocking other operations.

Approach:

The system will implement a **multi-threaded server** where each client request is handled by a separate thread. A **thread pool** mechanism will be used to prevent excessive resource consumption.

Algorithm: Multi-Threaded Request Handling

1. **Initialize the Server:**
   - Start the main thread to listen for incoming connections on a predefined port.
   - Load necessary configurations (e.g., database, logging, security policies).
2. **Accept Client Requests:**
   - When a new client connects, assign a new thread from the thread pool to handle the request.
   - Store active connections in a connection pool.
3. **Process the Request:**
   - If the request is for authentication, check user credentials.
   - If the request is for chat, forward messages to the appropriate user.
   - If it is a media upload, handle it asynchronously.
4. **Terminate Thread After Processing:**
   - Release resources and return the thread to the thread pool.
   - Close the connection if the client disconnects.

Key Technologies:

- **Python/Java Threads**
- **Thread Pooling to manage multiple client requests efficiently**
- **Database connection pooling to prevent bottlenecks**

2. Real-Time Chat & Notification Algorithm

Objective:

To enable real-time interactions between users through an optimized chat and notification system.

Approach:

The system will use **WebSockets** for full-duplex communication between the server and clients, allowing messages and notifications to be sent and received instantly.

Algorithm: Real-Time Messaging

1. **Establish a WebSocket Connection:**
   - When a user logs in, create a WebSocket connection.
   - Store active connections in a dictionary {user_id: WebSocket_connection}.
2. **Handle Incoming Messages:**
   - Each message is received through a dedicated thread.
   - Parse the message format ({sender_id, receiver_id, message_text, timestamp}).
   - Store the message in a database for persistence.
3. **Forward the Message:**
   - If the recipient is online, deliver the message instantly via WebSocket.
   - If the recipient is offline, store the message and send a notification upon login.
4. **Handle Notifications:**
   - Trigger an event when a new message, like, or comment is received.
   - Send real-time notifications via WebSocket.
   - Store notifications in a database for users to retrieve later.

Key Technologies:

- **WebSockets or Socket.io for real-time bidirectional communication**
- **Redis or Kafka for efficient message queuing and event handling**
- **Database indexing to optimize chat history retrieva**

3. Feed Loading & Media Upload Algorithm

Objective:

To optimize how posts, images, and videos are uploaded and displayed to users, ensuring fast response times.

Approach:

- **Lazy Loading** for fetching only necessary content.
- **Asynchronous Uploads** to avoid blocking the user interface.

Algorithm: Optimized Feed Loading

1. **Paginate Posts:**
    - Load only N posts at a time using SQL LIMIT OFFSET or cursor-based pagination.
    - Fetch additional posts as the user scrolls (infinite scrolling).
2. **Cache Frequently Accessed Content:**
    - Use **Redis or Memcached** to store recently viewed posts.
    - Expire cache entries periodically to maintain fresh content.
3. **Use Asynchronous Media Uploads:**
    - When a user uploads a media file:
        - The file is stored in **temporary storage**.
        - A **background worker thread** processes and compresses the file.
        - After processing, the file is moved to a **CDN (Content Delivery Network)** for fast access.
4. **Optimize Database Queries:**
    - Use **indexes** on frequently accessed fields (e.g., post timestamps, user likes).
    - Store media metadata separately from post content to speed up retrieval.

Key Technologies:

- **Lazy Loading & Pagination (React.js, Next.js)**
- **Redis/Memcached for caching frequently accessed posts**
- **CDN (Cloudflare, AWS S3) for media storage**
- **Asynchronous Task Queue (Celery, RabbitMQ)**

References

1. Mozilla Developer Network (MDN) - WebSockets
2. Node.js Documentation - Multithreading and Worker Threads
3. MongoDB Documentation - NoSQL Database Design for Chat Applications
4. Express.js Official Documentation
5. Next.js Official Documentation
6. Socket.io Documentation - Real-time Communication
7. JWT Authentication - Best Practices and Implementation