OS PROGECT REPORT

WEB SERVER MODEL

Group Members

Bilal Jawaid

Maaz

Zeeshan

Hamza

Contents

Introduction:

This project involves the development of a **Multi-Threaded Lightweight HTTP Server** designed to handle HTTP GET requests, serve static files, and log server activity. The server uses multithreading to handle multiple client requests simultaneously. Advanced features such as caching, request queuing, and server metrics tracking were implemented to optimize performance and reliability.

Objectives:

The primary objectives of this project were:

1. Build a lightweight HTTP server capable of handling multiple concurrent client requests using threads.
2. Implement features like caching, metrics logging, and request queuing.
3. Test the server's performance under varying loads using a custom stress-testing tool.
4. Learn and apply key concepts from operating systems such as threads, mutexes, and socket programming.

System Design:

Features:

1. **Static File Serving**:
    a. The server serves files from a designated static directory.
    b. Handles valid file requests and returns HTTP 200 (OK) responses with the file content.
    c. Returns HTTP 404 (Not Found) for missing files and HTTP 405 (Method Not Allowed) for unsupported methods.
2. **Multithreading**:
    a. Uses a thread pool to manage worker threads for handling client requests.
    b. Avoids blocking and ensures efficient handling of multiple concurrent requests.
3. **Caching**:

    a. Implements an in-memory cache to store frequently accessed files.
    b. Reduces file system reads and improves response times.
4. **Request Queuing**:
    a. Queues incoming client requests using a thread-safe queue.
    b. Prevents overloading of worker threads.
5. **Metrics Tracking**:
    a. Tracks successful, failed, and rejected requests.
    b. Logs metrics to a file for further analysis.
6. **Thread-Safe Logging**:
    a. Logs server activity, including request handling and errors, in a thread-safe manner.

Architecture:

The server follows a modular architecture:

1. **Main Server Loop**:
    a. Accepts incoming connections.
    b. Queues requests for worker threads.
2. **Worker Threads**:
    a. Fetch requests from the queue.
    b. Handle client requests and send appropriate responses.
3. **Cache Module**:
    a. Stores file content in memory for frequently accessed files.
    b. Reduces file I/O and enhances performance.
4. **Metrics and Logging**:
    a. Records server activity and performance metrics.
    b. Ensures thread-safe updates to logs and metrics files.

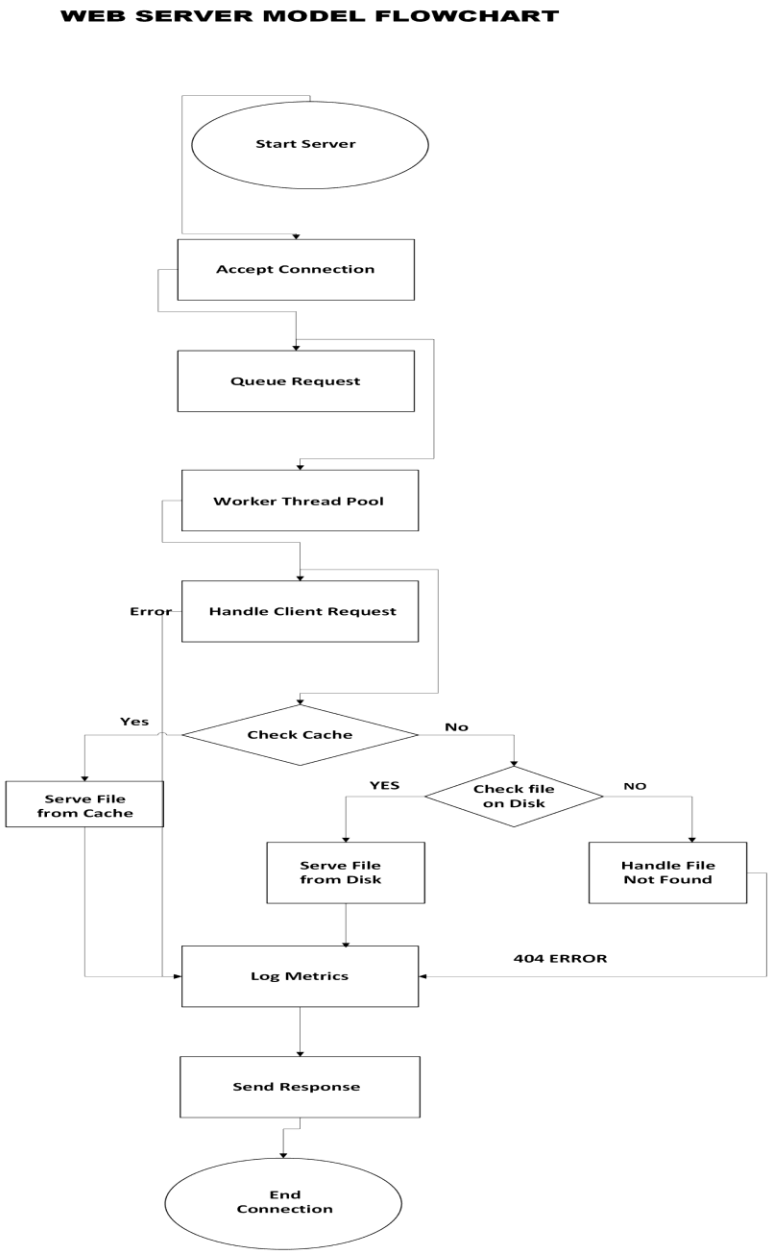
Implementations:

Technologies Used

The server follows a modular architecture:

1. **Main Server Loop**:
    a. Accepts incoming connections.
    b. Queues requests for worker threads.
2. **Worker Threads**:
    a. Fetch requests from the queue.
    b. Handle client requests and send appropriate responses.
3. **Cache Module**:
    a. Stores file content in memory for frequently accessed files.
    b. Reduces file I/O and enhances performance.
4. **Metrics and Logging**:
    a. Records server activity and performance metrics.
    b. Ensures thread-safe updates to logs and metrics files.

Flowchart:


WEB SERVER MODEL FLOWCHART
Start Server
Accept Connection
Queue Request
Worker Thread Pool
Handle Client Request
Error
Check Cache
Yes
No
Serve File from Cache
Check file on Disk
YES
NO
Serve File from Disk
Handle File Not Found
404 ERROR
Log Metrics
Send Response
End Connection

Testing:

Test Scenarios

1. **Normal Load**:
    a. 100 requests with no simulated failures.
    b. Observed metrics: Success rate and average response time.
2. **High Load**:
    a. 500 requests with a small thread pool.
    b. Evaluated request queuing and rejection rates.
3. **Failure Simulation**:
    a. Tested timeout and connection errors to ensure proper handling of failures.

Observations:

- **Multithreading**: The thread pool efficiently handled concurrent requests, minimizing latency.
- **Caching**: Reduced average response times significantly for frequently accessed files.
- **Request Queueing**: Successfully prevented server crashes during high loads by rejecting excessive requests.
- **Error Handling**: Properly logged and categorized failed and rejected requests.

Conclusion:

The project successfully demonstrated:

- Implementation of a robust multi-threaded HTTP server.
- Effective use of caching and thread-safe mechanisms.
- Real-world testing of server performance under varying conditions.