Indian Institute of Technology Bombay

CS 744 Autumn 2022 Semester, Project 3

Report On

HTTP Server and Client Implementation and Load Test Analysis

Submitted by: Shamik Kumar De(22m0822)

1 Introduction

In today's digital age, web servers are the backbone of online services, providing the infrastructure for everything from simple web pages to complex applications. The efficiency and reliability of these servers are critical to ensuring a smooth user experience. This report outlines the development, testing, and optimization of a multithreaded HTTP server-client application, focusing on enhancing performance under high-load conditions. The project also includes the creation of a custom load generator to simulate real-world scenarios, allowing for a detailed analysis of the server's response to varying levels of traffic.

2 Project Objectives

The primary objectives of this project were to:

1. **Develop a Multithreaded HTTP Server**: The server needed to handle multiple concurrent client connections, processing requests efficiently and ensuring that responses were sent in a timely manner.
2. **Implement a Client Application**: The client application was to be designed to interact with the server by sending HTTP requests and receiving the server's responses.
3. **Create a Load Generator**: A critical component of the project was the development of a load generator that could simulate a large number of concurrent users, each sending HTTP requests to the server. This tool was essential for stress-testing the server and evaluating its performance.
4. **Analyze Server Performance**: The performance of the server under different load conditions was to be thoroughly analyzed. Key metrics such as throughput and round-trip time (RTT) were measured to assess the server's efficiency.
5. **Optimize Server Throughput**: Based on the analysis, bottlenecks were to be identified and addressed to optimize the server's throughput and minimize response times.

3 HTTP Server Implementation

The HTTP server was implemented using C++ and the pthread library. The key elements of the server's implementation included:

- **Socket Creation and Binding**: The server initializes by creating a socket using the `socket()` function and binding it to a specific port using the `bind()` function. This sets up the server to listen for incoming connections.

- **Listening for Connections**: The server uses the `listen()` function to start listening for incoming connections from clients. Once a connection request is received, it is handled by a dedicated thread.
- **Multithreading for Concurrency**: The server's ability to handle multiple clients simultaneously is achieved through multithreading. Each new client connection is managed by a separate thread, created using the `pthread_create()` function. This allows the server to process requests concurrently, ensuring that each client is serviced independently of others.
- **Request Parsing and Response Handling**: Upon receiving a client's HTTP GET request, the server parses the request, processes it, and retrieves the requested resource. The server then sends the appropriate HTTP response back to the client.
- **Optimizations with Thread Pooling**: To avoid the overhead associated with creating and destroying threads, the server utilizes a thread pool. The size of this pool is optimized based on the hardware's capabilities, balancing the number of active threads with the available processing resources to minimize context switching and maximize throughput.

4 Client Implementation

The client application was developed to interact seamlessly with the HTTP server. The key features of the client include:

- **Establishing Connections**: The client establishes a connection to the server using the `socket()` and `connect()` functions. This connection persists for the duration of the request-response interaction.
- **Sending HTTP Requests**: The client sends an HTTP GET request formatted according to the HTTP/1.1 protocol. The request includes the resource path and the necessary headers.
- **Receiving and Displaying Responses**: After sending the request, the client waits for the server's response, which is then read and displayed. The connection is closed once the response is fully received, completing the interaction.

5 Load Generator Implementation

The load generator is a crucial tool designed to test the server under various simulated user loads. The load generator was implemented in C and features several key components:

- **User Simulation**: Each simulated user is represented by a thread, created using the `pthread_create()` function. These threads continuously send HTTP requests to the server, mimicking real-world usage patterns.
- **Metrics Collection**: Each user thread tracks its performance metrics, including the total number of requests sent and the cumulative round-trip time (RTT) for these requests. This data is essential for evaluating the server's performance.
- **Time Control and Synchronization**: The load generator includes a global control mechanism to manage the start and end of the test. This is handled using a `pthread_mutex_t` mutex for synchronization, ensuring that all threads start and stop their activities simultaneously.
- **Logging and Output**: Key events and metrics are logged to a file, providing a detailed record of the test. This includes the creation of threads, the start and end of the test, and the performance metrics collected.
- **Metrics Calculation**:
  - **Throughput Calculation**: The load generator calculates the server's throughput by dividing the total number of requests processed by the server by the test duration. This provides a measure of how many requests the server can handle per second under the given load.
  - **Average RTT Calculation**: The average round-trip time is computed by dividing the total round-trip time by the number of requests sent. This metric indicates the average time taken for a request to be processed and a response received.

6 Testing and Performance Evaluation

The server was subjected to rigorous load testing using the custom load generator, which was designed to simulate a variety of real-world scenarios. The following test parameters were used:

- **Number of Concurrent Users**: Tests were conducted with varying numbers of concurrent users, ranging from 10 to 100. This allowed us to evaluate how the server handled increasing loads.
- **Think Time**: The think time (delay between requests from the same user) was varied between 0.5 and 5 seconds. This parameter was crucial in understanding how the server performed under different interaction rates.
- **Test Duration**: Each test was conducted over a fixed duration of 60 seconds, ensuring consistency in the results.

6.1 Performance Metrics Collected

- **Throughput**: The server's throughput, measured in requests per second, was calculated by dividing the total number of requests processed during the test by the test duration.
- **Average Round-Trip Time (RTT)**: The average RTT was calculated by dividing the total round-trip time by the total number of requests. This metric provided insights into the server's responsiveness.

6.2 Results

- **Throughput vs. Concurrent Users**: The server's throughput increased with the number of concurrent users up to a certain point, beyond which it plateaued. This plateau indicated that the server had reached its maximum processing capacity.
- **RTT vs. Concurrent Users**: As the number of concurrent users increased, the average RTT also increased. This was due to the higher contention for server resources, leading to delays in request processing.
- **Impact of Think Time**: Longer think times generally resulted in lower server loads, leading to improved response times and slightly better throughput. This was because longer think times allowed the server to process requests with less contention and reduced resource competition.

7 Bottleneck Identification

The analysis of the load test results revealed several bottlenecks affecting server performance:

- **Thread Contention**: As the number of concurrent users increased, thread contention became a significant bottleneck. The overhead of managing a large number of threads, particularly context switching, led to degraded performance.
- **Socket Management Overhead**: The repeated creation and closure of sockets for each request added unnecessary overhead, limiting the server's ability to handle a large volume of requests efficiently. This overhead was particularly noticeable under high load conditions.
- **Limited Thread Pool Size**: The server's thread pool size was initially set without full consideration of the available hardware resources, leading to suboptimal performance. In some cases, the thread pool was too large, causing excessive context switching, while in others it was too small, leading to underutilization of available processing power.

8 Optimization Strategies

To address the identified bottlenecks and improve server performance, several optimization strategies were implemented:

- **Persistent Connections**: One of the most significant improvements was the introduction of persistent connections, allowing the server to reuse sockets for multiple requests. This change reduced the overhead associated with socket creation and closure, leading to a noticeable increase in throughput.
- **Optimized Thread Pool Size**: The size of the thread pool was adjusted based on the server's hardware capabilities. By fine-tuning the number of active threads, the server was able to balance resource utilization with the overhead of thread management, reducing context switching and improving performance.
- **Connection Handling**: The server was modified to handle connections more efficiently by reducing unnecessary operations during request processing. This included optimizing the way requests were read and responses were written to the network, minimizing delays.
- **Load Balancing**: Consideration was given to implementing load balancing strategies, such as distributing incoming requests across multiple server instances. Although this was not fully implemented within the scope of the project, it represents a viable strategy for further scaling the server to handle even higher loads.

9 Conclusion

The project successfully demonstrated the development and optimization of a multi-threaded HTTP server capable of efficiently handling concurrent requests. The comprehensive testing and performance analysis revealed several critical insights into the server's behavior under varying load conditions, guiding the implementation of key optimizations.

Through the introduction of persistent connections and careful tuning of the thread pool size, the server's throughput was significantly improved, and its average response time was reduced. The load generator proved to be an invaluable tool for simulating real-world scenarios, allowing for a detailed evaluation of the server's performance.

The findings from this project underscore the importance of resource management and optimization in the design of high-performance servers. The techniques and strategies developed here can be applied to a wide range of server applications, ensuring robust and scalable performance in production environments.

This report concludes with the acknowledgment that further optimizations, such as advanced load balancing and distributed server architectures, could enhance the server's scalability and resilience, paving the way for future work in this area.