MULTITHREADED VS EVENTDRIVEN WEBSERVER

Venkata Sai Indra Murala
MSECE
University Of Massachusetts, Amherst
vmurala@umass.edu

Shruti Parab
MSECE
University of Massachusetts, Amherst
shrutiajitpa@umass.edu

Siddhartha Roy Kurisetti
MSECE
University Of Massachusetts, Amherst
skurisetti@umass.edu

Abstract

This project aims to design, implement, and evaluate two distinct web server architectures — multithreaded and event-driven — and compare their performance characteristics under varying workloads. Through this comparison, we seek to gain a deeper understanding of the scalability and efficiency trade-offs that come with each model.

Introduction

Effective and scalable architectures of web servers are becoming a priority as more people are in need of responsive and high-throughput internet services. Web servers perform a key function in the provision of HTTP requests and for the delivery of static and dynamic content to users. The efficiency of the web server is closely related to such indicators of system performance as response time, scalability, and reliability.

There are two principal architectural models that the modern web servers use: the multithreaded solution or event-driven (asynchronous) solution. They both have their pros and cons making them better or worse according to the characteristics of the workload. Multithreaded servers, in turn, create a separate thread which handles an incoming connection thus providing for processing several requests at the same time. This model is quite easy to understand and works well with the traditional synchronous request-response protocols, but adds some overhead imposed by thread creation, context switching, and synchronization overhead.

On the other hand, event-driven servers work under a single-threaded event loop that handles various connections asynchronously. This architecture makes over-head very low and has better scaling in scenarios with high concurrency and I/O-bound workloads. Event-driven models such as Nginx are effective in real-time processing, within a cloud computing arrangement and the microservices, but call for more complicated asynchronous programming approaches.

In this project, there is both a multithreaded and an event-driven web server that is used to explore the relative performances. We measure throughput, latency, CPU and memory usage under different workload to get closer to the genuine trade-off between the two models. The aim is to gain insights that inform the developer and system architects on how to make informed decisions on concurrency model selection depending on performance needs.

Related Work

In our project, we implemented a web server based on the multithreaded approach. We hypothesized that this design would deliver higher throughput and lower response times. This assumption is based on the belief that multithreading makes better use of CPU and memory resources, allowing requests to be processed faster and more efficiently. Additionally, the complexity of asynchronous programming required by event-driven models can make development and debugging more difficult, potentially reducing the robustness of the server.

Pariag et al. [1] compared several high-performance server architectures, including event-driven servers like µserver, thread-per-connection servers using the Capriccio library, and hybrid models like WatPipe. Their study found that event-driven and hybrid architectures generally performed better under realistic workloads that included user think-times and varying file access patterns. Despite improvements in threading libraries, thread management overheads such as context switching and memory contention—continued to limit scalability in thread-based designs.

Another study by Prakash et al. [2] focused on performance in virtualized environments. They compared Apache, a multi-process/thread-per-connection server, and Nginx, an event-driven server. Nginx outperformed Apache under high loads, especially on virtual private servers (VPS), where efficient resource utilization is critical. However, under moderate loads, Apache was more efficient in terms of resource usage.

Finally, Albinali et al. [3] studied synchronization methods in multithreaded servers, comparing mutex locks, semaphores, spinlocks, and advanced protocols like Web-R2sync. Their results showed that semaphores had the lowest synchronization time, Web-R2sync reduced failure rates due to inconsistencies, and mutexes performed well in terms of scalability.

Matt Welsh et al. [4] introduced the Staged Event-Driven Architecture (SEDA), a novel design for building highly concurrent server applications. SEDA breaks the application into multiple stages connected by queues, allowing each stage to have its own thread management and scheduling policies. This modular approach helps improve code clarity and resource management. During periods of high load, SEDA prioritizes stages using fewer resources, which helps avoid scalability bottlenecks. By using a small number of threads efficiently distributed across different stages, SEDA reduces overhead and supports better performance under high concurrency.

In contrast, Von Behren et al. [4] argued that thread-based systems can offer simpler programming models with competitive performance. They introduced Capriccio, a user-level threading library capable of handling up to 100,000 threads efficiently. Capriccio uses linked stack management to reduce memory waste and provides resource-aware scheduling, which improves scalability without requiring complex asynchronous programming.

In conclusion, while event-driven architectures minimize overhead and scale effectively under high concurrency, multithreaded designs remain valuable for CPU-bound workloads and scenarios where synchronization is optimized. The choice of architecture should carefully consider workload types, resource availability, and deployment environments to achieve the best performance and scalability.

Implementation

This project aimed at designing and testing two kinds of web server architectures, i.e. one that were multithreaded and one based on event driven I/O, created from scratch using C++. These are servers which can support basic HTTP GET requests and offer static files stored in a predefined document root directory. All components were used with the help of low-level concepts in system programming, namely socket programming, and process/thread management.

i. Multithread Webserver

The multithreaded server implemented in this code is designed to handle multiple client requests concurrently by creating a dedicated thread for each incoming connection. The server starts by parsing optional command-line arguments to determine the document root directory and the port on which it should listen. By default, it serves files from the current directory and listens on port 8080. A TCP socket is created and configured with the SO_REUSEADDR option to allow the immediate reuse of the port after the server shuts down. The server binds to the specified port and begins listening for incoming connections.

For each new client connection, the server accepts the connection and spawns a new detached thread to handle it. This allows the main thread to remain free to accept additional client connections without waiting for ongoing requests to complete. The handle_client function is responsible for processing each client's HTTP request. It first reads the incoming data and performs basic HTTP request parsing, handling only GET requests. If the request method is unsupported or the requested file does not exist, the server responds with appropriate HTTP error codes such as 404 Not Found, 403 Forbidden, or 405 Method Not Allowed. If the requested file is valid, the server reads the file content and sends it back to the client along with the correct HTTP headers, including content type and length.

The server uses a simple MIME type detection mechanism based on file extensions to set the correct Content-Type in the response header. Connection logging and error handling are incorporated to track active client connections and provide helpful debug information. An atomic counter keeps track of the number of active connections, and a mutex ensures thread-safe logging. Once the response is fully sent, the client connection is closed, and the active connection count is updated. While this multithreaded approach allows concurrent handling of multiple clients, it may not scale efficiently under high loads due to the overhead of frequent thread creation and context switching.

ii. Event-Driven Webserver

This project implements an event-driven HTTP server that uses non-blocking sockets and the epoll API to efficiently handle multiple client connections within a single thread. The server is designed to serve static files from a specified directory and supports the HTTP Keep-Alive feature, which allows clients to reuse existing connections for multiple requests. This reduces the overhead of repeatedly establishing and closing TCP connections, improving overall performance. During startup, the server creates a TCP socket, enables the SO_REUSEADDR option to allow quick port reuse, and sets the socket to non-blocking mode. It then binds to the specified port and listens for incoming client connections. The epoll interface is used to monitor events on multiple sockets efficiently, and the edge-triggered mode (EPOLLET) ensures that event notifications are minimized, making the server more scalable under heavy traffic.

The main functionality of the server runs inside an event loop using epoll_wait. When a new client connects, the server accepts the connection, makes the client socket non-blocking, and registers it with epoll to wait for incoming data. As data arrives, the server reads it into a buffer and performs basic HTTP request parsing to extract the request method, URL path, and HTTP version. It also checks whether the client has requested a persistent connection using the Keep-Alive header. Based on the requested file, the server prepares the appropriate HTTP response, setting the correct MIME type and response headers. It supports common HTTP status codes such as 200 OK for successful responses, 403 Forbidden when access to the resource is denied, and 404 Not Found when the requested file does not exist.

When the server is ready to send data back to the client, it uses the EPOLLOUT event to write the response. If the entire response is successfully sent and the client requested a Keep-Alive connection, the server keeps the connection open for further requests. Otherwise, it closes the connection to free up resources. This event-driven design helps the server efficiently handle many simultaneous connections without using multiple threads, keeping resource usage low while maintaining high responsiveness.

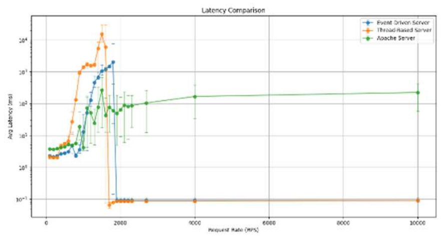
RESULTS

A range of controlled experiments was carried out with Vegeta to check how the performance of both web servers differed. A comparison was made between custom servers and the Apache server, since the latter is an optimized and time-tested reference. To ensure that no server had an unfair advantage, each was placed on a single CPU core.

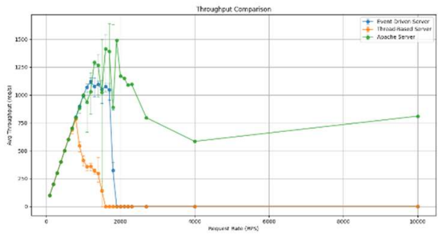
The process included serving a 1 MB HTML file to the server under several rates of requests. Every 100 RPS, the load was increased from 100 to 2500 until the end of the test. A configuration was tested five times and each run lasted around four minutes. The average (showed by the dots) is calculated across five runs and the error bars display the difference between the smallest and largest results.

For up to 1500 RPS, latency remained low and steady on the event-driven server and it handled over 1000 RPS in throughput. Even so, by 1500 RPS, Node.js slowed down suddenly because the single-threaded event loop couldn't keep up with the bigger workload. The thread-based server ran into issues as early as 1000 RPS, since latency began to rise and throughput fell sharply after 1200-1300 RPS and this led to its inability to process requests at all because of extra thread handling and switching contexts.

Latency_graph:



Throughput_graph:



The load was too great at 2500 RPS and both custom servers failed to connect any clients, leading to an increase in latency. At this point, both architectures began to struggle with carrying heavy loads. To learn more about Apache's capabilities, the test directly set the RPS to 4000, then increased it gradually to 10,000. While Apache handled peak loads, some decrease in performance could be observed because the system was completely filled with requests.

Such results confirm that custom servers are not well-suited to workloads with many connections. Up to a modest number of requests, the event-driven approach performs better than Nginx and Apache, but at very high scales, Apache is unmatched. Being hybrid, optimizing schedule and sharing resources in its own pool enable Apache to react well to different amounts of traffic.

LIMITATIONS & FUTURE SCOPE

An important limitation of our web server project is the lack of performance testing with video content. Since video files are typically large and require real-time streaming, they place significant demands on server resources, including bandwidth and processing power. Evaluating the server's ability to handle video streaming would provide valuable insights into its scalability and performance under more resource-intensive conditions.

Additionally, we did not assess the server's performance with dynamic web pages. Serving dynamic content involves server-side computations, database interactions, and generating content based on user inputs, all of which add complexity to request handling. Analyzing the server's performance in delivering dynamic content would offer a deeper understanding of its ability to manage complex operations efficiently.

To address these gaps, future research should focus on performance testing specifically for video streaming and dynamic content delivery. This would involve creating experiments that mimic real-world scenarios and measuring key performance indicators such as response time, throughput, and resource usage. Exploring these areas will provide a more complete picture of the server's strengths and limitations, helping to identify opportunities for further improvements and making the server more suitable for a wider range of web applications.

REFERENCES

[1] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla, "Comparing the Performance of Web Server Architectures," *Proceedings of the EuroSys Conference*, pp. 1–14, Mar. 2007.

[2] P. Prakash, B. R. Mohan, and S. Kamath, "Performance Analysis of Process Driven and Event Driven Web Servers," *2015 IEEE 9th International Conference on Intelligent Systems and Control (ISCO)*, Coimbatore, India, pp. 1–6.

[3] H. Albinali, M. Alharbi, R. Alharbi, and M. Aljabri, "Synchronization Techniques for Multi-threaded Web Server: A Comparative Study," *14th IEEE International Conference on Computational Intelligence and Communication Networks (CICN)*, 2022.

[4] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *Proc. 18th ACM Symp. Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001, pp. 230–243.

[5] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proc. 19th ACM Symp. Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, Oct. 2003, pp. 268–281.