Design and Implementation of a High-Performance Web Server Based on C++ and Epoll

Bachelor's Thesis

Degree Programme in Computer Applications

Spring 2025

Yan Zheng

HAMK
Häme University
of Applied Sciences

Abstract

DP	Degree Programme in Computer Applications or BIT
Author	Yan Zheng Year 2025
Subject	Design and Implementation of a High-Performance Web Server Based on C++ and Epoll
Supervisors	Dr. Jawad Yasin

Modern web services must serve many users at the same time without delay. This requires web servers that are fast and efficient. This thesis is an independent project by one student. It aims to design and implement a compact, high-performance web server. The server will be built in C++ and it will use the Linux epoll system to handle many connections. The main research question is how to use C++ and epoll to build a web server that stays fast and stable even when many users connect at the same time.

This research is practical in nature. The work is divided into several parts. It starts with some background on web servers and epoll. Then it covers the design and building of the server step by step. The methods include writing the server code, testing it under different conditions, and logging the results for analysis.

The thesis is both practical and research-based. First, it explains some important ideas like multi-threading, memory management, and epoll. Then, it shows how the server was built step by step. The main method used is testing and comparing how the server performs. Notes and logs were also kept during development. The goal is to show if the server can handle many users well.

Keywords Multithreading Memory Management high concurrency

Pages 33 pages and appendices 1 pages

Glossary

Event-Driven I/O	Processes requests only when I/O events occur, avoiding wasted polling.
Synchronous Logging	Writes each log entry immediately in the main thread for critical events.
Asynchronous Logging	Buffers log entries in a queue and writes them in a background thread.
Mutex	A lock ensuring only one thread at a time can access a shared resource.
Condition Variable	Allows threads to wait for or signal changes in shared state, used with a mutex.
Semaphore	A counter-based lock controlling access to a limited number of resources.
HTTP GET / POST	GET retrieves static resources; POST sends data (e.g., form fields) to the server.
CGI	A standard for running external programs to generate dynamic web content.
Mmap	Maps a file or device into memory for zero-copy I/O.
Writev	Writes multiple buffers in a single system call.
Socketpair	Creates two connected sockets for inter-thread or inter-process communication.
Concurrency	Handling multiple tasks at the same time to improve responsiveness.
Multithreading	Running multiple threads in one process to share work and memory.
Thread Pool	A fixed set of threads reused to handle tasks efficiently.
Race Condition	Error from threads accessing shared data without proper control.
Deadlock	Threads waiting on each other forever, causing a freeze.
Memory Leak	Allocated memory not freed, leading to wasted memory.
Dangling Pointer	A pointer pointing to freed memory, causing crashes.
Memory Fragmentation	Memory split into small unusable chunks.
RAII	C++ pattern tying resource release to object lifetime.
Smart Pointer	C++ object that automatically manages memory.
Memory Pool	pre-allocated memory blocks reused to improve performance.
Epoll	A Linux tool for efficient I/O event handling on many connections.

select / poll – Older, slower I/O multiplexing methods.

Edge-Triggered (ET)	Epoll mode: notifies only on new events.
Level-Triggered (LT)	Epoll mode: keeps notifying until event is handled.

Table of Contents

Figures

Appendices

1 Introduction

The rapid growth of online services—from video streaming and cloud storage to social media and e-commerce—has placed ever-increasing demands on web servers. In today's world, users expect instant responses and uninterrupted access, even when thousands or tens of thousands connect at the same time. Traditional server models, which create a separate process or thread for each incoming request, are easy to implement but struggle under heavy load. Each new thread or process consumes memory and takes time to start, and context switches between them waste precious CPU cycles. As user count grows, these overheads add up; they can cause servers to slow dramatically or even crash.

To address these issues, modern high-performance servers typically rely on two key ideas: event-driven input/output (I/O) and controlled threading. Linux's epoll interface offers a way to monitor many connections at once without constant polling. A server registers all active sockets with epoll, then sleeps until the kernel signals that one or more sockets are ready for reading or writing. This approach eliminates wasted work and scales to thousands of idle connections. At the same time, using a fixed pool of worker threads avoids the cost of spinning up new threads on every request. Once started, these worker threads can process tasks from a shared queue and then return to waiting, reducing both memory footprint and scheduling overhead.

C++ is an ideal language for building such servers because it allows developers to manage memory and CPU resources at a fine level of detail. By applying smart pointers and custom memory pools, the server can automatically clean up unused data and avoid fragmentation that would otherwise degrade performance over time. A two-mode logging system further ensures that critical events are recorded reliably: immediate, or synchronous, logging captures high-priority information without delay, while background, or asynchronous, logging writes bulk data to disk without blocking request handling.

This thesis describes the design and implementation of a tiny C++ web server that combines an epoll-based event loop, a reusable thread pool, and advanced memory and logging strategies. The goal is to answer three main questions: first, how to choose thread pool size and organize the event loop to achieve a good balance between raw throughput and low response time; second, how epoll-driven concurrency compares in real-world tests to older I/O methods such as select or poll, and to simpler thread-per-connection model;

and third, whether techniques like RAII, smart pointers, and custom allocators can keep memory usage stable under heavy, continuous load.

To explore these questions, a survey of existing server architectures, event notification mechanisms, and memory management patterns is first conducted. A framework is then presented and developed for measuring performance and logging detailed data to ensure that these tests are repeatable. Next, the step-by-step implementation is described: setting up the listening socket and epoll instance, integrating signal-driven timers, creating the thread pool and task queue, handling HTTP parsing for both static and dynamic content, and managing resources such as memory and connections. This is followed with a series of benchmarks comparing this server to industry standards like Apache and NGINX under both static page loads and dynamic, database-backed requests. Each is evaluated in terms of requests per second and latency, with particular focus on server behaviour under high connection counts and bursts of activity.

By weaving together careful design, systematic testing, and clear analysis, this work demonstrates that a streamlined C++ implementation can deliver low-latency performance competitive with well-established products, while keeping the codebase straightforward and maintainable. Finally, the strengths and limitations of the approach are reflected upon, and directions for future improvement are suggested, including adaptive load balancing, zero-copy I/O, and finer-grained per-request resource management.

2 Fundamentals of High-Performance System Programming

This project rests on three foundational pillars. The first pillar is Concurrency and Multi-Threading. Concurrency refers to the ability of a system to execute multiple tasks, share resources, and manage interactions, either by executing them simultaneously or in time-sharing (context switching). Concurrency improves the responsiveness, throughput, and scalability of modern computing. Concurrency often appears together with multithreading (Wikipedia contributors, n.d.-a). The second pillar is Memory Management. Memory management is how a program requests, organizes, and frees its working memory. Traditional allocators follow simple rules that can leave unused gaps (fragmentation) over time. New compacting allocators can provably eliminate this wasted space—without changing any object's address—and thus dramatically cut the memory footprint of C/C++ programs (Powers et al., 2019). The third pillar is Event-Notification Systems. High-performance servers demultiplex thousands of sockets via scalable interfaces (e.g., Linux epoll), which decouple interest registration from event retrieval and avoid the linear scans inherent in select and poll (Gammo et al., 2004). In the remainder of this chapter, each of these areas is examined in detail to establish the theoretical basis for this high-performance web server design.

2.1 Concurrency and Multithreading

Concurrency is often handled with multithreading. A thread is like a mini-program inside a process. Threads share the same memory, but each has its own stack, registers, and instruction pointer. On a single core, the OS can switch between threads. On a multicore machine, threads can truly run at the same time. This lets servers use all their CPU power. But if two threads touch the same piece of data at once, and one of them writes, what they do can clash and cause errors. To stop that, locks or atomic operations are used. These tools make sure only one thread changes the data at a time (Silberschatz et al., 2018, p. 273).

2.1.1 Concurrency

Concurrency in computing is the ability of a system to handle multiple tasks at the same time (Wikipedia contributors, n.d.-a). Instead of doing one task after another, a concurrent system can start and run several tasks in overlapping time periods. This does not always mean the tasks are finishing at exactly the same instant, but their execution is interleaved,

which improves efficiency and keeps the system responsive. A common example is a web browser loading multiple tabs at once, or a server handling several client requests concurrently, so no single user has to wait for others to finish (Clay Breshears, 2009, p. 2).

2.1.2 Multithreading

A common way to achieve concurrency is through multithreading. A thread is like a lightweight subprocess. Multiple threads can run in the same application process, and they share the same memory space. In C++, starting with C++11, developers can create threads using a library keyword `std::thread`. It provides an API for starting and managing threads. A thread can read data from a client while processing a previous request; these threads typically execute concurrently on a multicore processor. The operating system scheduler manages these threads and quickly switches between them (or keeps them running on different CPU cores), creating the illusion of simultaneous execution. The result is improved performance and the ability to handle more work simultaneously, which is critical for busy web servers (Stroustrup, 2013, p. 1014).

In this project, thread pools were used to achieve Multithreading development. Thread pools are an important concept related to multithreading in server design. Creating a new thread for each client request is expensive (because memory is required to start the thread and its stack), and can also lead to resource exhaustion if there are too many threads. A thread pool reuses a fixed set of threads. Instead of spawning a new line for each incoming connection, the server maintains a pool of pre-created worker threads. Incoming tasks (such as requests) are sent to this connection: the idle line receives the task and returns to the thread pool to wait for the next task after completion. This way, when the server handles many requests, the total number of threads is limited (preventing thousands of threads from being overwhelmed). Thread pools reduce the overhead of constant creation and deletion, resulting in more stable and efficient performance (Stroustrup, 2013, p. 1014).

It is important to note that multithreading is only one approach to concurrency. There are other similar patterns, each with its own advantages and disadvantages. One is multi-process model. Instead of using threads, a server can run multiple processes (independent programs) to handle connections. For example, early web servers such as Apache could fork a new process for each client or maintain a pool of worker processes. Each process has its own memory, which provides strong isolation (a process crash does not affect other processes), but context switches between processes are generally more expensive than context switches between threads. In addition, inter-process communication is more

complex than thread-shared memory. Under high load, the overhead of many processes, each with its own memory space, can be significant. Another one is Event-Driven Single-Thread Model. Another way is to use asynchronous I/O (Input or Output) and a single-threaded event loop (sometimes added with several threads to perform blocking tasks). Servers such as Nginx and Node.js follow this model. Here, a single thread (or a small number of threads) monitors many sockets using an event notification system (such as Linux's epoll, discussed later). A non-blocking thread handles one event at a time (e.g., a socket is ready for reading or writing). This avoids the overhead of multiple threads, since many connections are handled in a single thread. However, it requires that the code be written in a non-blocking style (callbacks or asynchronous routines) so that no connection can block the loop. The advantages are low overhead per connection and no synchronization primitives (since a single thread handles all events). However, a server does not use multiple CPU cores unless it runs multiple processes or threads with separate event loops. In practice, high-performance servers often use a combination (e.g., several processes or threads each executing an event loop).

Each approach to concurrency has some disadvantages. Multithreading allows true parallelism on multicore systems (multiple requests are processed on different cores at the same time) and can make it easier to program in a synchronous manner, but it introduces the complexity of synchronization. When threads share data, developers should use locks or other synchronization mechanisms to avoid race conditions (for example, two threads modify a shared variable at the same time). Incautious use of threads can lead to problems such as race conditions (two threads interfere with each other's updates) or deadlocks (two threads get stuck waiting for each other's locks). In contrast, the single-threaded event loop model avoids these particular problems (no concurrent access within a single loop), but it limits throughput if that thread becomes a bottleneck, and writing fully asynchronous code can be complex. In summary, concurrency in high-performance C++ web servers is usually achieved through multithreading, which is often enhanced by using thread pools to efficiently manage workloads. This approach allows the server to handle many simultaneous client connections. It is one of many solutions (Multiprocessor or event-driven design), and the best choice may depend on specific requirements and circumstances. In this thesis, a multithreaded design in C++ for a web server is focused on, and an event notification mechanism (epoll) is used to implement scalable I/O, combining the advantages of threads with efficient event handling. (Gammo et al., 2004).

2.2 Memory Management

Memory management refers to the way a program allocates and handles memory during execution. Memory management is critical for long-running, high-load server applications. Web servers run continuously and handle many requests per second; each request may require some memory to be allocated (for request data, response data, buffers, etc.). If memory is not managed properly (for example, if each request wastes a small amount of memory), the server's memory usage will grow indefinitely and eventually exhaust the system's RAM or swap space. Memory leaks can cause serious problems in long-running programs. They cause memory fragmentation and fill up real memory and page space with unused data. In some cases, the system may even experience page space exhaustion due to such leaks (IBM, 2025). In short, servers need to reuse and release memory properly to remain stable in the long run.

In C++, memory management is manual by default. So, the programmer is responsible for requesting memory (for example, using malloc or new) and for freeing the memory when the request is complete (free or delete). Manual memory management allows a lot of control and can be efficient, but it is easy to get wrong. Two common problems are memory leaks and dangling pointers. A memory leak occurs when a program allocates memory and then loses all references to it without freeing it. The allocated block is never used in memory. Over time, leaks can accumulate and consume a lot of memory (especially problematic for servers that need to run indefinitely). A dangling pointer is the opposite error. It occurs when a program prematurely or incorrectly frees memory but continues to use a pointer to that memory. This can result in undefined behaviour, crashes, or data corruption. Another problem is double free, where a program frees the same block of memory twice. This can cause program to crash or corrupt memory allocator data structures.

In C++, memory management is manual by default. The programmer must request memory with `malloc()` or `new()` and free it when it’s no longer needed `free()` or `delete()` (Stroustrup, 2013, p. 211). Manual control can be efficient, but also error-prone. Two common problems are memory leaks and dangling pointers. A memory leak happens when allocated memory is never freed because all references to it are lost; over time, these leaks accumulate and can exhaust a long-running server’s memory (Wikipedia contributors, n.d.-d). A dangling pointer is the opposite: memory is freed too soon or incorrectly, yet the program continues to use the pointer—leading to undefined behaviour or crashes (Wikipedia contributors, n.d.-b). Another related bug is double free, where the same block

is freed twice, corrupting the allocator’s data structures and often crashing the program (Wikipedia contributors, n.d.-c).

Memory fragmentation is a deeper problem that can occur even without a leak. Fragmentation means that the memory is divided into so many small independent chunks that it becomes impossible to have large contiguous chunks (Kellett, 2021).

Fragmentation can occur in applications that allocate and empty objects of different sizes; over time, free memory locations are juggled between still-used locations, creating free "slots" for new components. Severe fragmentation can degrade performance (due to cache abuse, due to more page faults) and waste memory (too much memory is available, but cannot be used for higher demands). In a server that handles a large number of requests, fragmentation can cause the server to consume more memory than it needs to do its job if memory allocation and deallocation is not done carefully.

2.2.1 Standard Management method

To solve these issues, modern C++ practices encourage the use of RAII (Resource Acquisition Is Initialization) and smart pointers. RAII is a programming term in which owning resources (such as heap memory allocations) is tied to the lifetime of an object. In practice, this means an object’s constructor acquires the resource and its destructor releases the resource. In RAII, holding resources is a class invariant and is tied to the object lifecycle. Therefore, if there is no object leak, there can be no resource leak (Wikipedia, n.d.). C++ smart pointer classes, such as `std::unique_ptr` and `std::shared_ptr` , are built on RAII principles. For example, a smart pointer will allocate an object on the heap, but when goes out of scope, its destructor automatically calls on that. This automatic cleanup reduces the chance of leaks, even in the exceptions or multiple return paths (the destructor will run and free the memory no matter how the function exits). Smart pointers make manual calls largely unnecessary in modern C++ code, thereby protecting against many memory management mistakes. They also clearly express ownership semantics (e.g. unique ownership vs shared ownership of memory), which helps in reasoning about the program’s memory lifecycle.

2.2.2 Custom method

Another way to manage memory usage on high-performance servers is to use a custom memory allocator or memory pool. A memory pool pre-allocates a large block of memory and manages the allocation of that block, reducing fragmentation and increasing the speed of allocation of similar-sized objects. By selecting from the nearest pool, it keeps related data in sync and can release large blocks of memory after a pool reallocation (i.e., a single-request pool that releases all memory when a request is made). However, unlike traditional caching strategies, researchers developed novel memory allocations to overcome general levels of fragmentation (Wikipedia contributors, n.d.-e).

One good solution is Mesh, by Powers and colleagues, which plugs in place of malloc/free to cut down on wasted space. In C/C++, you normally cannot relocate an object once it's allocated—there's no system to update every raw pointer—so fragmentation builds up over time. Mesh sidesteps this by pairing up pages that have "holes" in different slots and then pointing both virtual pages at one shared physical page. Each live object stays at its original virtual address, so you never need to touch any pointers, and Mesh simply returns the now-empty page back to the OS. In effect, it can halve the physical memory used by two half-filled pages. Powers et al. demonstrated that Mesh matches the speed of top allocators while cutting Firefox's memory use by 16% and Redis's by 39%, thanks to its fast, randomized page-matching algorithms and low runtime cost (Powers et al., 2019).

For a high-performance server, using such an allocator can be beneficial when memory is a limiting factor, as it increases memory utilization efficiency. However, even with advanced allocators, good coding practices (like avoiding leaks and using RAII) are the first line of defines to ensure robust memory management. In summary, effective memory management in a C++ web server involves careful allocation/freeing, leveraging language features to prevent leaks, and possibly using specialized allocators to handle fragmentation, so that the server remains stable and efficient over long periods of heavy use.

2.3 Epoll-Based Event Handling

Efficient management of many network connections at the same time is one of the main challenges in developing high-performance servers. Linux provides an advanced I/O event notification tool: epoll, which is specifically designed to expand to multiple fd (file

descriptors) at the same time. Epoll allows the server to monitor multiple connections and receive an alert when any of them is ready for I/O (for example, data arrives for reading, or a buffer is available for sending data), without the need to constantly check each connection in a loop like select, poll. In short, epoll is an event-driven mechanism: the server registers all sockets with epoll and then waits for epoll. The kernel will only notify the application when one or more of these sockets actually have an event, so the CPU is not wasted on checking idle connections. The advantages of epoll can be highlighted by comparing it with *select* and *poll*, both of which operate by passing a list of sockets to the kernel and waiting for a readiness report. However, these older methods face scalability issues. For example, select has a limit on the number of file descriptors it can handle (usually 1024 by default) and requires rebuilding the entire `fd_set` every time it is called, making it inefficient for large numbers of connections. If you have 10,000 client connections, you need to set 10,000 bits in the mask and pass it to select on each call, and the kernel will scan the list each time (Wikipedia. (n.d.). This results in an overhead roughly proportional to the number of descriptors (O(n) per call). In the case where many connections are idle (especially for long-running servers where only a subset are active at most of time), select/poll ends will do a lot of unnecessary work checking those fds (Gammo et al., 2004).

These problems were highlighted in a study by Gammo et al. They showed that the select system could not sustain the large number of concurrent web connections. The inefficiency stems from the fact that select (and poll) combine interest declaration with event retrieval. In other words, every time an application wants to wait for events, it must re-tell the kernel the entire list of things it's interested in. The kernel does not remember that list between calls. This means a lot of data (the list of file descriptors) is copied from user space to kernel space on each call, and the kernel must iterate through it. As Gammo et al. describe, this design leads to "large overheads, particularly in environments with large numbers of connections and relatively few new events occurring"(Gammo et al., 2004). In a high-performance server with a lot of connections. This overhead may become a burden.

Moreover, epoll provides another idea. It separates the registration of events from the retrieval of events. With epoll, you can first use `epoll_ctl()` calls to tell the kernel which sockets you want to care about and what kinds of events you want to monitor. you can add or remove sockets from this list anytime. Then, the program calls the `epoll_wait()` to block until there is an event belong to any of the sockets. The difference is that you do not need to pass the whole list of sockets on every wait. The kernel retains that list from the

earlier `epoll_ctl()` calls. Then, waiting for events with epoll is efficient even if you are monitoring a huge number of file descriptions.

Another advantage of epoll is that it can operate in two models: level-triggered and edge-triggered. In level-triggered mode, as long as a file description ready (e.g. has unread data), it will continue to return that file description every time `epoll_wait()` is called (until the data is consumed). In edge-triggered mode, epoll will tell you only when a new event happens – it will not repeat notifications for a socket until its state changes again. Edge-triggered mode can be more efficient (fewer system calls for repeated events) but requires the application to handle I/O carefully: for example, when you get a read-ready notification, you must read all available data from that socket to ensure you does not miss any data, because you will not get another notification until new data arrives after that. Gammo et al. mention that using edge-triggered epoll can reduce the overhead further by cutting down the number of events delivered.

In actual applications, many high-performance servers use edge-triggered epoll in combination with non-blocking I/O to achieve more efficient performance. In 2004, Gammo et al. conducted a test to compare epoll with select and poll using a high-performance Web server load. The results showed that when all connections are working and there are no idle connections, traditional poll or select can sometimes be on par with epoll on Linux 2.6, or even slightly exceed it. The reason is that epoll itself needs to maintain a listening list, and when connections frequently go online and offline, `epoll_ctl()` needs to be called continuously, adding additional burden.

However, when there are many idle connections or several file descriptors, epoll shows obvious advantages. In this large-scale concurrent scenario, epoll is much more scalable than the old way. Subsequent improvements and common usage patterns (such as edge triggering and persistent connections) have further improved the performance of epoll. In fact, epoll is one of the keys to modern servers being able to handle tens of thousands or even more concurrent connections.

Therefore, epoll is an important foundation for building high-performance Linux web servers. It handles the I/O of multiple sockets in a more efficient way, avoiding the overhead of each call to the traditional interface. If epoll is used in a C++ server and combined with a multi-threaded design (for example, using a thread pool to let each thread listen to a portion of the connections), thousands of connections can be processed simultaneously at a low cost. This event-driven and concurrent structure can not only serve

several clients, but also respond quickly to each request, thereby achieving the goal of high performance.

Multi-threaded concurrency, careful memory management, and epoll’s efficient event handling will all play a role in subsequent chapters to support design and implementation.

3 Data Management

Data management is divided into two parts: the first part concerns the performance measurement framework, and the second part concerns the logging system. In the first part, clear workloads and metrics are defined, and controlled load tests are run, with response time, throughput, and error rate recorded under each scenario. This follows the approach of Menascé et al., who emphasize the validation of server performance models through systematic load testing and capacity planning. In the second part, detailed timestamped logs of every request, response, and error are captured during each test. This demonstrates that rich structured logging enables precise post-mortem analysis—identifying bottlenecks, correlating events, and reliably reproducing problems. (Banga & Druschel, 1999) .

Therefore, these components provide us with accurate and repeatable data. The framework ensures that the same measurements are taken consistently, and when unexpected behaviour occurs, the logs allow detailed analysis of the underlying issues.

3.1 Performance benchmarking

Performance benchmarking measures how well a web server works under realistic loads. This step is important to know the server and improve its efficiency. Tests must simulate real user behaviour to be meaningful.

Real web traffic is bursty. Many users send several requests in a short period. Simple load generators often fail to reproduce these bursts. Banga & Druschel note that synthetic workloads sometimes do not match the characteristics of real traffic. This mismatch can lead to overly optimistic performance measurements.

Server performance also depends on how many connections are active at the same time. Pariag et al. compared different web server architectures and found that high connection counts can add overhead. A good benchmark must create heavy loads that reflect real-world conditions (Pariag et al., 2007).

Both Web Bench and wrk are chosen as the performance testing tools. Web bench is widely used and simple to operate. It can generate a high number of HTTP requests to stress the server. It also provides clear metrics such as requests per second and average

response time. Wrk, meanwhile, is a modern HTTP benchmarking tool that supports multi-threading and Lua scripting, enabling the simulation of more complex and realistic workloads. These measures help us compare server performance and fine-tune system settings.

3.2 Diary and Log Keeping

Keeping a research diary and detailed logs is essential for solid data management. A diary records programmer's day-to-day decisions, observations, and any unexpected hiccups, why you chose one test over another or when a procedure was tweaked, so you (or others) can later trace exactly how and why the experiment evolved (Runeson & Höst, 2009). At the same time, structured log files capture every event, warning, and error with precise timestamps and context; this lets you pinpoint performance issues or failures quickly, without wading through irrelevant entries (Ding et al., 2015) . In conclusion, the diary's narrative and the logs' technical detail create a complete, searchable record that makes developers' transparent, repeatable, and easier to audit or extend.

3.2.1 Diary Content

A Personal diary will record developers' observations, thoughts, and decisions throughout the project. The entries in the diary include: Design changes (Explain why the design decisions were made.); Implementation steps (Record how the code modules and system components were built and integrated.); Problems (Record any problems or unexpected behaviours, as well as the time and context.); Test modifications(Record the adjustments made during performance tuning and the reasons behind them.).The diary helps track the evolution of the project and serves as a reference for future improvements. It also aids in troubleshooting by providing a timeline of modifications.

3.2.2 Log Keeping

Logs are detailed system-generated records of events during testing and running the server. This logging system captures: Event details (Information on incoming requests, connection handling, and epoll events.); Error messages (Warnings and error notifications generated by the system.); Performance metrics (Data such as response times, throughput, and resource usage.

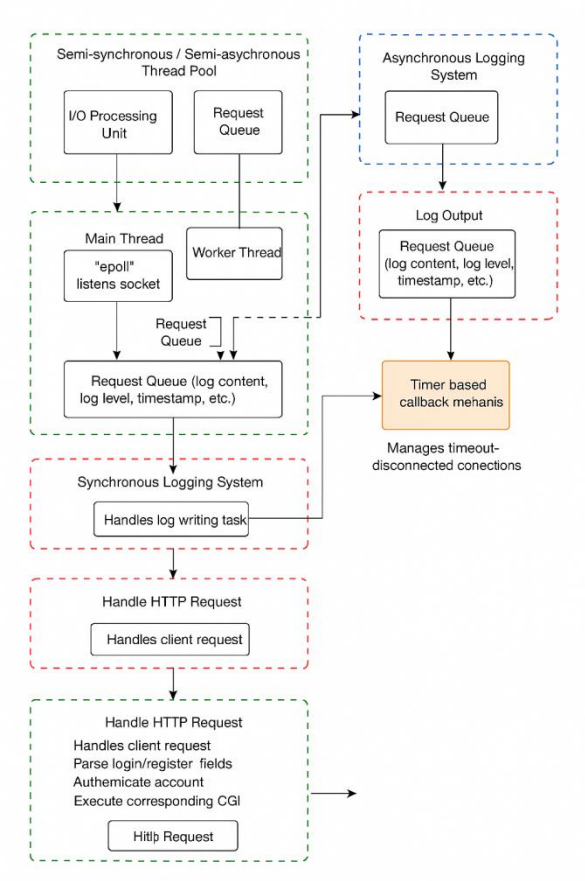
By continuously collecting log data, it can be verified that the server behaves as expected under various loads. Logs also provide a basis to analysis delay delays, errors, or bottlenecks. This systematic log keeping ensures that the experiments are transparent and repeatable. Such record keeping aligns with best practices in experimental research, which stress the need for detailed and consistent documentation (Banga & Druschel, 1999).

Therefore, diary and log keeping support a strong data management strategy. They allow us to review what changes were made, understand the conditions under which experiments were conducted, and ultimately help improve the server's performance and reliability.

4 Method

This chapter details the specific methods used for designing and implementing a high-performance web server, including architectural design, thread pool mechanisms, logging systems, HTTP request handling, and resource and connection management strategies. The details are shown in the Figure 1 below:

Figure 1. Architectural Design



4.1 Architectural Design

The web server employs a combination of a semi-synchronous/semi-asynchronous thread pool and an asynchronous logging system to achieve efficient event-driven performance. The core architectural elements include

4.1.1 Main Thread

The server listens for socket connections using Linux's epoll mechanism for event detection, and upon receiving client requests, it adds tasks to a request queue to be processed by worker threads from the thread pool.

4.1.2 Thread Pool

The thread pool consists of an I/O Processing Unit and a Request Queue for efficient task distribution and is composed of several pre-created worker threads to reduce the overhead of frequent thread creation, thereby enhancing server throughput.

4.2 Thread Pool Mechanism

The thread pool operates with a fixed number of pre-allocated threads to avoid the overhead of dynamic thread management, uses thread-safe data structures such as linked lists or queues to store tasks, and relies on synchronization mechanisms like condition variables to allow worker threads to monitor the queue and promptly process tasks as they arrive.

4.3 Logging System

This chapter will introduce the system's log design, which is mainly divided into two modes: synchronous log and asynchronous log. Synchronous log records immediately when the task is executed to ensure that key events can be saved in time; asynchronous log is processed through buffers and dedicated threads to reduce the burden of the main task and improve overall performance. The two methods have their own emphasis and should be selected according to different situations in actual use.

4.3.1 Synchronous Logging System

The synchronous logging system directly handles log records during task execution and immediately writes log data, including log level and timestamp, to persistent storage, ensuring promptness and reliability.

4.3.2 Asynchronous Logging System

The asynchronous logging system utilizes a dedicated request queue buffer for log tasks, records server state, events, and error information asynchronously to minimize the impact on main tasks and employs dedicated logging threads to write buffered logs to files, further enhancing performance.

4.4 Timer-based Callback Mechanism

A timer-based callback mechanism manages clients that have timed out or disconnected. Periodically checks and cleans up idle or abnormal connections, reclaiming system resources and ensuring long-term server stability.

4.5 HTTP Request Handling

The HTTP request handling procedure dispatches tasks based on request types: for static resource requests (GET), the server directly returns the requested files, while for dynamic requests (POST and CGI), it parses form data such as login or registration details, performs user authentication, and executes CGI programs that interact with databases to generate appropriate responses.

4.6 Resource and Connection Management

To ensure stable and efficient operation under high concurrency, the server employs various connection and memory management strategies. These resource management systems are critical to maintaining performance, preventing process overload, and supporting long-term scalability.

4.6.1 Connection Management:

Use efficient epoll mechanisms to monitor numerous concurrent connections, integrated with a semi-synchronous/semi-asynchronous model to minimize connection latency and enhance concurrent processing capacity.

4.6.2 Memory Management

Utilizes smart pointers to manage the lifecycle of dynamically allocated memory, preventing memory leaks and dangling pointers. Implements the RAII (Resource Acquisition Is Initialization) pattern to strictly manage resource allocation and release. Use custom memory allocators (Memory Pools) to reduce memory fragmentation and optimize memory utilization.

Through these methods, the server achieves high efficiency in high-concurrency environments, balancing performance with resource management and system stability to meet the demanding requirements of modern web services.

5 Implementation of Web Server

This chapter introduces how the above methods were applied in the actual implementation of the web server.

5.1 Initialization

To prepare the server for handling incoming client connections, a listening socket (`m_listenfd`) is created using the socket system call. The socket is then configured with two important options: `SO_LINGER`, which controls the behaviour during socket closure (whether to discard unsent data or wait), and `SO_REUSEADDR`, which allows the port to be reused quickly after server restart.

A server address structure (`sockaddr_in`) is initialized with the appropriate IP and port, and the socket is bound to it using `bind()`. Afterward, the socket begins listening for connections with `listen()`.

The server uses `epoll_create()` to create an epoll instance (`m_epollfd`) and registers the listening socket using `Utils::addfd()`. This sets the foundation for event-driven I/O.

To handle asynchronous events such as timeouts or termination signals, a `socketpair` is established, and the read end is added to the epoll instance. The write end is used in signal handlers to notify the main loop. SIGPIPE, SIGALRM, and SIGTERM signals are caught using `Utils::addsig()`, and a periodic timer is set with `alarm(TIMESLOT)`.

Finally, the global pipe and epoll file descriptors are stored in static members of Utils for access during runtime operations. The Implementation process shows below (Figure 2).

Figure 2. Server initialization with epoll and signal handling

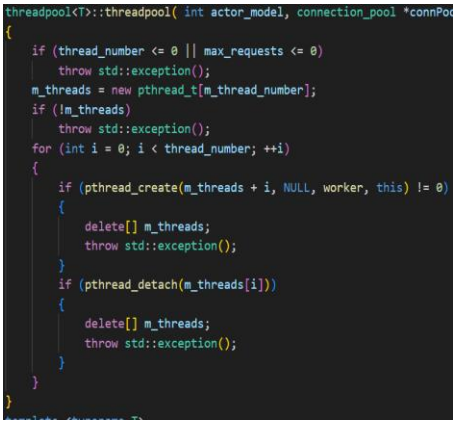
```
void WebServer::eventListen()
{
    m_listenfd = socket(PF_INET, SOCK_STREAM, 0);
    assert(m_listenfd >= 0);
    struct linger tmp;
    if (0 == m_OPT_LINGER)
        tmp = {0, 1};
    else if (1 == m_OPT_LINGER)
        tmp = {1, 1};
    setsockopt(m_listenfd, SOL_SOCKET, SO_LINGER, &tmp, sizeof(tmp));
    int ret = 0;
    struct sockaddr_in address;
    bzero(&address, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl(INADDR_ANY);
    address.sin_port = htons(m_port);
    int flag = 1;
    setsockopt(m_listenfd, SOL_SOCKET, SO_REUSEADDR, &flag, sizeof(flag));
    ret = bind(m_listenfd, (struct sockaddr *)&address, sizeof(address));
    assert(ret >= 0);
    ret = listen(m_listenfd, 5);
    assert(ret >= 0);
    utils.init(TIMESLOT);
    epoll_event events[MAX_EVENT_NUMBER];
    m_epollfd = epoll_create(5);
    assert(m_epollfd != -1);
    utils.addfd(m_epollfd, m_listenfd, false, m_LISTENTrigmode);
    http_conn::m_epollfd = m_epollfd;
    ret = socketpair(PF_UNIX, SOCK_STREAM, 0, m_pipefd);
    assert(ret != -1);
    utils.setnonblocking(m_pipefd[1]);
    utils.addfd(m_epollfd, m_pipefd[0], false, 0);
    utils.addsig(SIGPIPE, SIG_IGN);
    utils.addsig(SIGALRM, utils.sig_handler, false);
    utils.addsig(SIGTERM, utils.sig_handler, false);
    alarm(TIMESLOT);
    Utils::u_pipefd = m_pipefd;
    Utils::u_epollfd = m_epollfd;
}
```

This figure shows how the server prepares for event-driven socket management and signal processing using epoll and Unix domain sockets.

5.2 Thread Pool Usage

In Figure 3, A thread pool is created in `WebServer::thread_pool` using the `threadpool<T>` class template, which is initialized with a fixed number of pre-spawned threads that continuously run a worker loop to process incoming tasks;

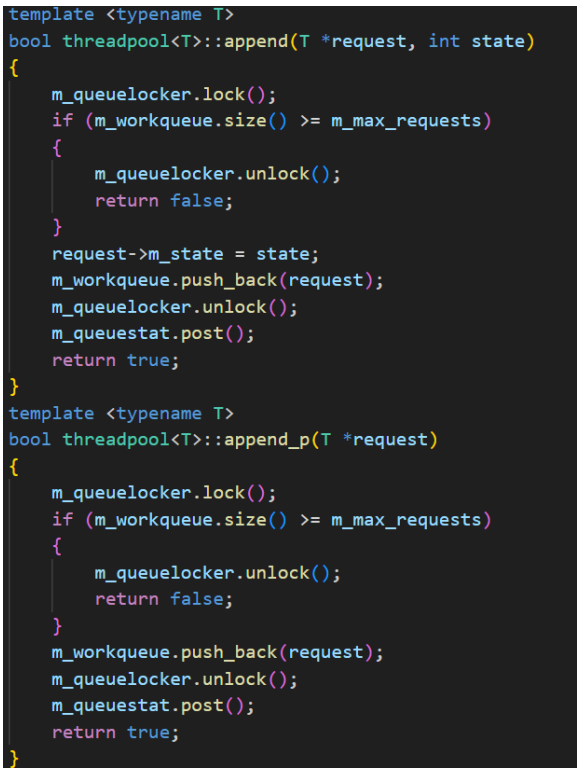
Figure 3. Thread pool initialization flow



```
threadpool<T>::threadpool( int actor_model, connection_pool *connPo
{
    if (thread_number <= 0 || max_requests <= 0)
        throw std::exception();
    m_threads = new pthread_t[m_thread_number];
    if (!m_threads)
        throw std::exception();
    for (int i = 0; i < thread_number; ++i)
    {
        if (pthread_create(m_threads + i, NULL, worker, this) != 0)
        {
            delete[] m_threads;
            throw std::exception();
        }
        if (pthread_detach(m_threads[i]))
        {
            delete[] m_threads;
            throw std::exception();
        }
    }
}
```

Figure 4 shows the methods `append()` and `append_p()` are used to push tasks such as HTTP connections `(http_conn)` into a synchronized queue where they are then processed by the available worker threads.

Figure 4. Work thread initialization flow



```
template <typename T>
bool threadpool<T>::append(T *request, int state)
{
    m_queuelocker.lock();
    if (m_workqueue.size() >= m_max_requests)
    {
        m_queuelocker.unlock();
        return false;
    }
    request->m_state = state;
    m_workqueue.push_back(request);
    m_queuelocker.unlock();
    m_queuestat.post();
    return true;
}
template <typename T>
bool threadpool<T>::append_p(T *request)
{
    m_queuelocker.lock();
    if (m_workqueue.size() >= m_max_requests)
    {
        m_queuelocker.unlock();
        return false;
    }
    m_workqueue.push_back(request);
    m_queuelocker.unlock();
    m_queuestat.post();
    return true;
}
```

These two template-based code fragments implement a fixed-size thread-pool with a bounded task queue by combining POSIX threads, a mutex, and a semaphore to realize the classic producer–consumer pattern. In the constructor, the pool allocates an array `m_threads` sized to the requested thread count, then loops to spawn and detach each thread—each running a static worker function—and immediately throws `std::exception` (after freeing the array) if any `pthread_create` or `pthread_detach` call fails, thereby ensuring strong exception safety and preventing resource leaks. On the producer side, two mutex-protected methods—`append(request, state)` and `append_p(request)`—first lock `m_queuelocker` and check whether the queue size has reached the configured `m_max_requests`; if so, they unlock and return false to enforce back-pressure. Otherwise, append assigns `request->m_state = state` (while `append_p` does not), both push the request into `m_workqueue`, unlock the mutex, and call `m_queuestat.post()` to wake a waiting worker before returning true. This design bounds concurrent workload to prevent

overload, offers both stateful and stateless task submission, and maintains robust resource management throughout the pool's lifecycle.

5.3 Logging System

Figure 5, Logging is initialized in the `WebServer::log_write` function, where the server chooses between synchronous and asynchronous modes based on configuration parameters.

Figure 5. Log system initialization flow

```
void WebServer::log_write()
{
    if (0 == m_close_log)
    {
        if (1 == m_log_write)
            Log::get_instance()->init(
        else
            Log::get_instance()->init(
    }
}
```

Figure 6 shows that in asynchronous mode, the system employs a blocking queue (`block_queue`) to store log entries temporarily, along with a dedicated logging thread (`flush_log_thread`) responsible for writing these entries to log files in the background.

Figure 6. Asynchronous log writing with queue buffer

```
block_queue(int max_size = 1000)
{
    if (max_size <= 0)
    {
        exit(-1);
    }

    m_max_size = max_size;
    m_array = new T[max_size];
    m_size = 0;
    m_front = -1;
    m_back = -1;
}
```

Log messages are formatted with timestamps and severity levels, are either directly written to files in synchronous mode or buffered for batched writing in asynchronous mode, reducing interference with main processing. Figure 7 below is the implementation of flushing.

Figure 7. Log flushing mechanism via logging thread

```
static void *flush_log_thread(void *args)
{
    Log::get_instance()->async_write_log();
}
```

5.4 Timer Management

The server uses a sorted doubly-linked list structure (`lst_timer`) to manage timers for all active client connections. When a new client connects, the server creates a corresponding timer and adds it to the list. During client activity such as reading or writing data, the timer is updated to extend the connection's lifetime. If a client becomes idle and its timer expires, the callback function automatically removes the connection and reclaims resources, ensuring stability over long-term operation. The Implementation process shows blow (Figure 8):

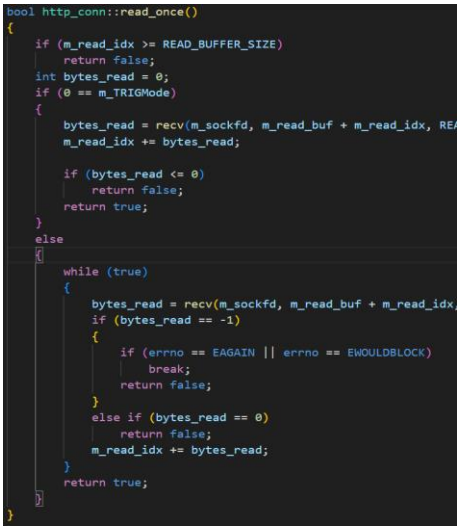
Figure 8. Timer list for managing client timeouts

```
struct client_data
{
    sockaddr_in address;
    int sockfd;
    util_timer *timer;
};

class util_timer
{
public:
    util_timer() : prev(NULL), next(NULL) {}

public:
    time_t expire;

    void (* cb_func)(client_data *);
    client_data *user_data;
    util_timer *prev;
    util_timer *next;
};

class sort_timer_lst
{
public:
    sort_timer_lst();
    ~sort_timer_lst();

    void add_timer(util_timer *timer);
    void adjust_timer(util_timer *timer);
    void del_timer(util_timer *timer);
    void tick();

private:
    void add_timer(util_timer *timer, util_timer *lst_head);

    util_timer *head;
    util_timer *tail;
};
```

5.5 HTTP Request Handling

Figure 9 shows that HTTP request processing is handled by instances of the `http_conn` class, which are initialized in the `WebServer::timer()` function. Each connection reads data using the `read_once()` method and processes it through a finite-state machine in the `process()` method.

Figure 9. Logic for reading client data in LT and ET modes



```
bool http_conn::read_once()
{
    if (m_read_idx >= READ_BUFFER_SIZE)
        return false;
    int bytes_read = 0;
    if (0 == m_TRIGMode)
    {
        bytes_read = recv(m_sockfd, m_read_buf + m_read_idx, RE
        m_read_idx += bytes_read;

        if (bytes_read <= 0)
            return false;
        return true;
    }
    else
    {
        while (true)
        {
            bytes_read = recv(m_sockfd, m_read_buf + m_read_idx
            if (bytes_read == -1)
            {
                if (errno == EAGAIN || errno == EWOULDBLOCK)
                    break;
                return false;
            }
            else if (bytes_read == 0)
                return false;
            m_read_idx += bytes_read;
        }
        return true;
    }
}
```

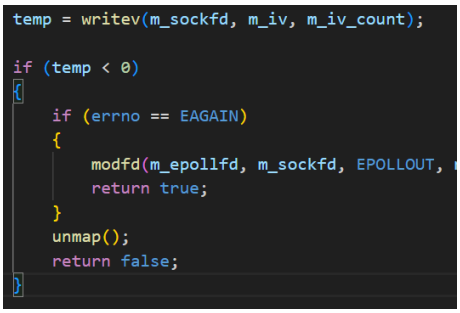
Figure 10, for static resources requested via GET, the server maps the target file to memory using `mmap()`.

Figure 10. Mapping static file to memory

```
int fd = open(m_real_file, O_RDONLY);
m_file_address = (char *)mmap(0, m_file_
close(fd);
return FILE_REQUEST;
```

Then sends it in Figure 11 using `writev()` along with constructed HTTP headers. For dynamic POST requests, such as user login or registration, form data is parsed and used to query or update a MySQL database. The server then generates the appropriate HTML response based on the operation result.

Figure 11. Sending response



```
temp = writev(m_sockfd, m_iv, m_iv_count);

if (temp < 0)
{
    if (errno == EAGAIN)
    {
        modfd(m_epollfd, m_sockfd, EPOLLOUT,
        return true;
    }
    unmap();
    return false;
}
```

5.6 Signal and Event Handling

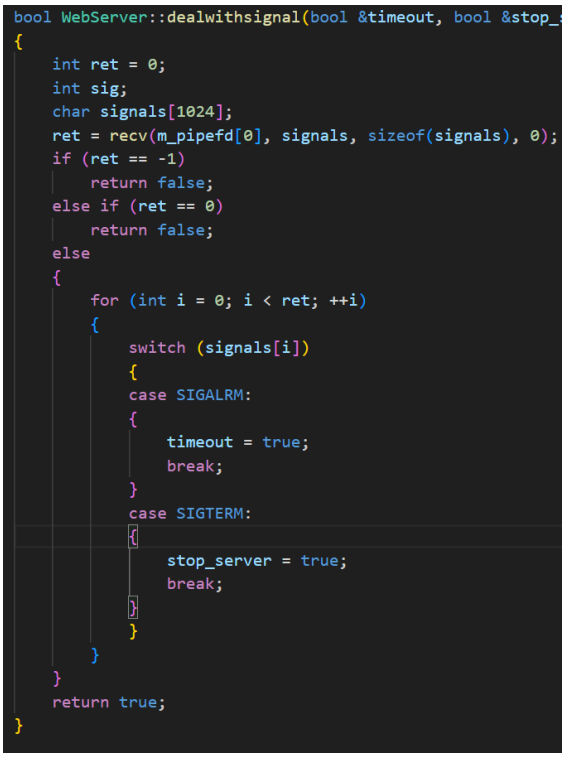
The server manages asynchronous events using a `socketpair` mechanism, which allows signal handling to be integrated into the epoll event loop. Signals such as SIGALRM (for timers) and SIGTERM (for termination) are registered using `Utils::addsig()` and delivered via the write-end of the `socketpair`. The read-end is monitored by epoll, enabling the server to react to signal events inside its main loop. The Implementation process shows blow (Figure 12).

Figure 12. Signal registration

```
utils.addsig(SIGPIPE, SIG_IGN);
utils.addsig(SIGALRM, utils.sig_handler, false);
utils.addsig(SIGTERM, utils.sig_handler, false);
alarm(TIMESLOT);
```

In Figure 12, the `alarm()` function is used to periodically trigger SIGALRM, and all received signals are processed by `WebServer::dealwithsignal()` to update control flags such as `timeout` or `stop_server`.

Figure 13. Signal event handling and server control logic



```
bool WebServer::dealwithsignal(bool &timeout, bool &stop_
{
    int ret = 0;
    int sig;
    char signals[1024];
    ret = recv(m_pipefd[0], signals, sizeof(signals), 0);
    if (ret == -1)
        return false;
    else if (ret == 0)
        return false;
    else
    {
        for (int i = 0; i < ret; ++i)
        {
            switch (signals[i])
            {
            case SIGALRM:
            {
                timeout = true;
                break;
            }
            case SIGTERM:
            {
                stop_server = true;
                break;
            }
            }
        }
    }
    return true;
}
```

This hands-on implementation shows how multi-threading, non-blocking I/O, and modular design can work together to build an efficient web server.

6 Result

This chapter presents the quantitative outcomes of the performance evaluation, comparing this Web Server, Apache, and NGINX under both static and dynamic workloads. Two key metrics are focused on: throughput (requests per second) and latency (mean and percentile values). The selection of Apache and NGINX is based on their prevalence in production environments and contrasting architectures—Apache's mature process-based model versus NGINX's high-performance event-driven design—and is informed by industry benchmarks such as the Ultimate Web Server Benchmark (Rendek, 2025).

6.1 Static Page Performance

Figure 14 shows the throughput (requests per second) of using static page in three web servers: Author's Web Server, Apache, and NGINX.

Figure 14. Static page: server throughput comparison

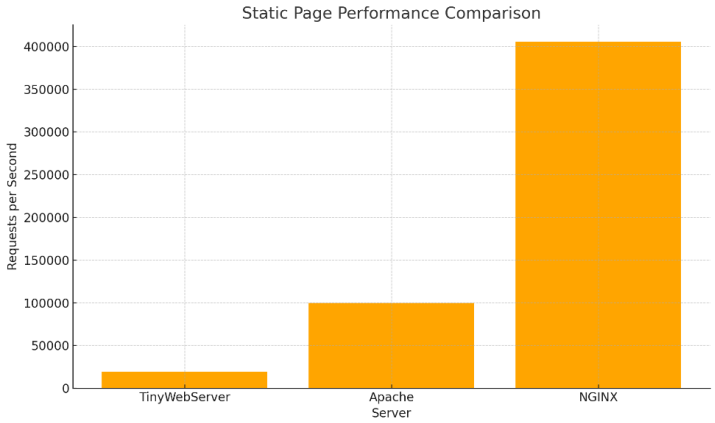
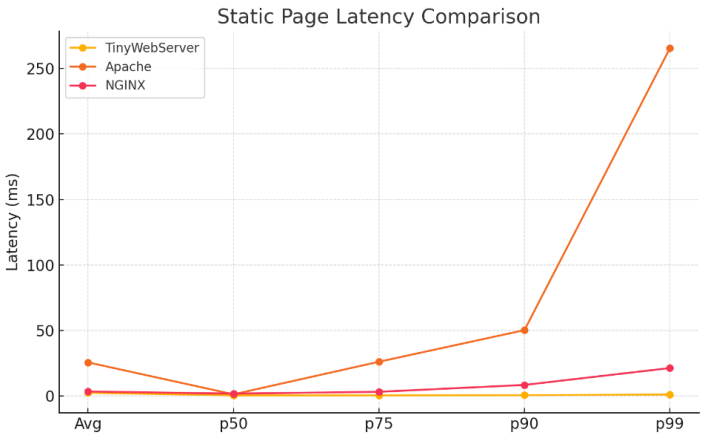


Figure 15 compares the latency (response time in milliseconds) at various percentiles using static page: Average, p50, p75, p90, and p99.

Figure 15. Static page: latency under high load



We first tested the server's performance with static pages. These are simple pages that do not require much processing to serve. Tools like Web Bench and wrk were used to simulate many users sending requests at the same time.

As seen in Figures 14 and 15, NGINX handled the most requests per second, with low and stable response times. Apache showed moderate throughput but had more variation in its latency, especially under heavy load.

This web server, while not handling as many requests as NGINX, showed low latency, with response times consistently under a millisecond. Although the throughput was about 3% of NGINX's peak performance, the server's response times were stable, showing that the design choices—such as memory management and using epoll for event handling—helped maintain quick responses even under load.

6.2 Dynamic Page Performance

Figure 16 shows the throughput (requests per second) of using dynamic page in three web servers: Author's Web Server, Apache, and NGINX.

Figure 16. Dynamic page: throughput comparison

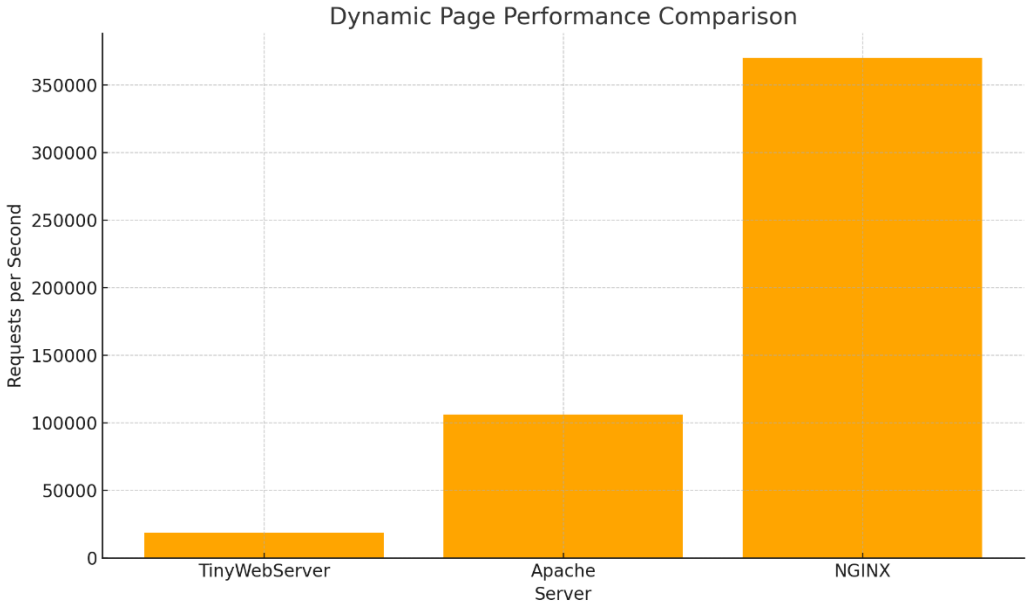
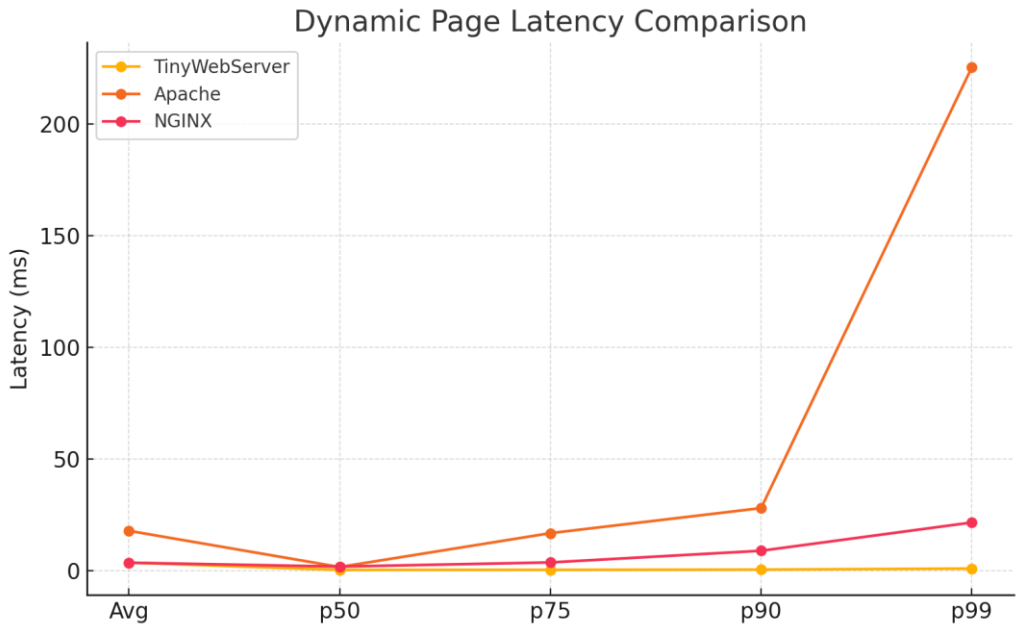


Figure 17 compares the latency (response time in milliseconds) at various percentiles using dynamic page: Average, p50, p75, p90, and p99.

Figure 17. Dynamic page: latency under high load



The server was tested with dynamic content, which is more complex because it requires extra processing, such as querying a database or handling form submissions.

The results shown in Figures 16 and 17 indicate that this server kept latency low, even when handling dynamic requests. NGINX still handled the most requests per second, but its response times increased a little under heavy load.

This server had lower throughput compared to NGINX but performed well with consistent low latency, even when handling more complex tasks. This shows that using multithreading and epoll for event handling helped the server respond quickly, even with more demanding workloads.

6.3 Performance analysis

The results confirm the effectiveness of the server's design choices, as the use of thread pools enabled efficient concurrency with low latency, the epoll-based event handling reduced CPU usage and maintained responsiveness under many idle connections, and the custom memory management system with smart pointers and memory pools minimized memory usage and avoided leaks and fragmentation, ensuring long-term stability.

Comparison with Apache and NGINX: While NGINX and Apache processed more requests per second, this server showed better performance in latency, especially with dynamic content. This confirms that using C++ and epoll can result in a web server that responds quickly and efficiently

6.4 Conclusion of Results

In conclusion, the testing shows that the custom web server performed well, with low latency and efficient resource management. Although it did not achieve the same throughput as NGINX, it handled high loads better in terms of response time. The results confirm that C++ and epoll, along with careful design, can create a web server that performs well even under heavy traffic.

These findings answer the research questions and show that the server design is effective for handling modern web traffic. Future work could focus on improving throughput and testing the server in real-world environments to refine its performance further.

7 Summary

This thesis focused on designing and implementing a high-performance web server using C++ and epoll to handle a large number of concurrent connections with minimal latency. The primary goal was to create a server that remains fast and stable under heavy load, addressing the common problem faced by many web servers, which slow down when handling numerous simultaneous users.

The research began by exploring foundational concepts such as concurrency, multi-threading, memory management, and event-driven programming using epoll. These concepts formed the backbone of the server's architecture, ensuring that it could efficiently manage resources and scale with increasing traffic. The server was designed using a thread pool for concurrency management and epoll for efficient event handling, which minimized the CPU overhead typically associated with traditional methods like select or poll.

The server was then implemented and tested under different conditions to evaluate its performance. The results showed that, while the server did not achieve the same throughput as established web servers like NGINX, it demonstrated exceptional performance in terms of latency, especially under dynamic workloads. The combination of multi-threading, epoll based event handling, and careful memory management ensured that the server could handle many simultaneous users without significant delays.

Comparing this server to Apache and NGINX, the results confirmed that while NGINX excelled in raw throughput, the custom server performed better in terms of maintaining low latency under high-load conditions. This confirmed the effectiveness of using C++ and epoll in building high-performance web servers.

In conclusion, this thesis demonstrated that a well-designed web server using C++ and epoll can provide efficient handling of high-concurrency environments while maintaining fast response times. Future work may focus on improving throughput by exploring the design of the Efficient Load Balancing used by Nginx to further refine the server's performance (Nedelcu, 2013, p. 119).

References

Banga, G., & Druschel, P. (1999). Measuring the capacity of a Web server under realistic loads. *World Wide Web*, *2*, 69–83.

Clay Breshears. (2009). *The Art of Concurrency*. https://books.google.fi/books?id=rU68SYVS7S8C&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

Ding, R., Zhou, H., Lou, J. G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., & Xie, T. (2015). Log2: A cost aware logging mechanism for performance diagnosis. *Proceedings of the 2015 USENIX Annual Technical Conference*, 139–149. https://www.usenix.org/system/files/conference/atc15/atc15-paper-ding.pdf

Gammo, L., Brecht, T., Shukla, A., & Pariag, D. (2004). Comparing and evaluating epoll, select, and poll event mechanisms. *Proceedings of the Linux Symposium*, *1*, 215–230. http://www.mnis.fr/fr/services/virtualisation/pdf/LinuxSymposium2004_V1.pdf#page=215

IBM. (2025, January 20). *Memory-leaking programs*. https://www.ibm.com/docs/zh-tw/aix/7.1?topic=performance-memory-leaking-programs

Kellett, S. (2021, May 11). *Memory Fragmentation, your worst nightmare*. https://www.softwareverify.com/blog/memory-fragmentation-your-worst-nightmare/#:~:text=What%20is%20memory%20fragmentation%3F

Menascé, D. A., Almeida, V. A. F., & Dowdy, L. W. (2002). Load testing, benchmarking, and application performance management for the Web. *Proceedings of the 2002 Computer Measurement Group Conference*, 1–8. https://www.researchgate.net/profile/Daniel-Menasce/publication/221447217_Load_Testing_Benchmarking_and_Application_Performance_Management_for_the_Web/links/0c96051b9a0ec7aee5000000/Load-Testing-Benchmarking-and-Application-Performance-Management-for-the-Web.pdf

Nedelcu, C. (2013). *Nginx HTTP Server* (2nd ed.). Packt Publishing. https://thuvienso.dau.edu.vn:88/bitstream/DHKTDN/7049/1/6405.Nginx%20HTTP%20server%20%282nd%20ed%29.pdf

Pariag, D., Brecht, T., Harji, A., Buhr, P., & Shukla, A. (2007). Comparing the performance of web server architectures. *Proceedings of EuroSys '07*, 1–10. https://course.ece.cmu.edu/~ece845/sp15/docs/pariag-2007.pdf

Powers, B., Tench, D., Berger, E. D., & McGregor, A. (2019). Compacting memory management for C/C++ applications. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 333–346. https://doi.org/10.1145/3314221.3314582

Rendek, L. (2025, February). *Ultimate Web Server Benchmark: Apache, NGINX, LiteSpeed, OpenLiteSpeed, Caddy & Lighttpd Compared*. https://linuxconfig.org/ultimate-web-server-benchmark-apache-nginx-litespeed-openlitespeed-caddy-lighttpd-compared

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley. https://ia800502.us.archive.org/16/items/operatingsystemconcepts10th/OperatingSystemConcepts-10th.pdf

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley. https://www.stroustrup.com/4th.html

Wikipedia. (n.d.). Resource acquisition is initialization. In *Resource acquisition is initialization*. https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization#:~:text=Resource%20allocation%20%20,there%20are%20no%20%2074

Wikipedia contributors. (n.d.-a). *Concurrency (computer science)*. https://en.wikipedia.org/wiki/Concurrency_(computer_science)

Wikipedia contributors. (n.d.-b). *Dangling pointer*. https://en.wikipedia.org/wiki/Dangling_pointer

Wikipedia contributors. (n.d.-c). *Double free*. https://en.wikipedia.org/wiki/Double_free

Wikipedia contributors. (n.d.-d). *Memory leak*. https://en.wikipedia.org/wiki/Memory_leak

Wikipedia contributors. (n.d.-e). *Memory pool (computer programming)*. https://en.wikipedia.org/wiki/Memory_pool_(computer_programming)

Appendix 1: Data management plan

Description of thesis research data

The statistic data, diagrams, visualizations, and codes will be used in this thesis, the data could be images, tables, codes.

Management and storage of the research data

This data will be stored and processed on the thesis authors' own password-protected computer.

Processing of personal data and sensitive data

No personal data will be used in this thesis unless specifically agreed upon with the research subject.

Ownership of research data

The author of this thesis owns the research data and the results. The data was collected and processed independently for academic purposes, and no third-party organizations hold any ownership over the material.

Further use of research data after the completion of the thesis

Once the thesis is completed, the research data will not be reused. The information will be stored securely throughout the academic year to ensure the results can be verified if necessary, after which it will be destroyed.