Design and Performance Evaluation of a Minimal Non-Blocking HTTP/1.1 Server

Arthur Herbette

April 11, 2026

Abstract

This report presents the design and performance evaluation of a minimal HTTP/1.1 server implemented in C. The server relies on non-blocking I/O, an epoll-based event loop, and a multithreaded worker architecture using `SO_REUSEPORT` to handle concurrent connections efficiently. We analyse architectural choices, implementation challenges, and evaluate performance under synthetic workloads using `wrk`. The final implementation achieves throughput within run-to-run variance of `nginx` on static file serving.

1 Introduction

The goal of this project was to gain a deeper understanding of HTTP and network programming. You can find the GitHub repository at this link. I started this project as a simple blocking, single-threaded, loop-based server. After understanding those fundamentals I implemented header parsing which enabled persistent (keep-alive) connections. The goal after that introduction was to have a very efficient and scalable server. Those reasons motivated the transition to a non-blocking server architecture, allowing the server to handle multiple connections concurrently using asynchronous I/O, and eventually to a fully multithreaded design.

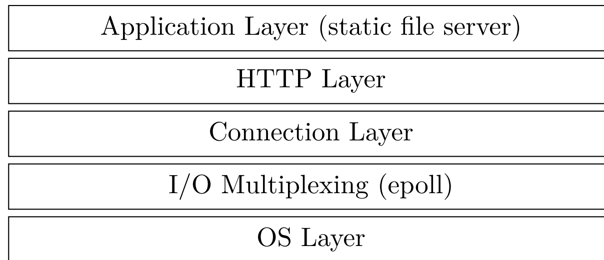
2 Architecture

The server follows an event-driven architecture. It started as a single-threaded reactor pattern and evolved into a multi-threaded design where each worker thread owns its own epoll instance and its own listen socket via `SO_REUSEPORT`. The architecture is divided into 7 layers:

- **core**: coordinates the server and orchestrates interactions between components
- **epoll**: contains structs to use epoll in the server loop
- **HTTP**: parses requests, turns plain text into `http_request_t`
- **net**: handles connection setup, TCP listen and accept
- **os**: provides low-level read/write operations and filesystem access
- **static**: handles all transactions which use static files (i.e., `index.html`)
- **util**: contains all utilities of the server; at the moment it contains only `buffer_t` which serves to store and consume all data from the connection

The system is divided into layers so that each component exposes a clear abstraction and minimises coupling with the others.

To better understand this server we can first explain what each layer does in the stack:



Incoming data flows upward through the stack (OS → HTTP), while responses are generated at the application layer and propagated downward to the socket.

OS Layer The OS layer's responsibilities are: sockets, file descriptors, read/write/accept, and filesystem access.

Event Layer The event layer (epoll) turns blocking OS events into asynchronous events. The question it tries to answer is: *which fd is ready?*

Connection Layer The connection layer abstracts over raw sockets; its responsibilities are: input/output buffers, write offsets, connection state, and the keep-alive lifecycle.

HTTP Layer It only works with plain text. The goal is to: parse requests, detect headers, handle keep-alive, and build responses. It transforms raw bytes into a structured request.

Application Layer (static) This is the highest-level logic. At the current moment, as it serves only GET, it only serves files, but could later become: API endpoints, dynamic content, routing.

2.1 Multithreaded Worker Model

The final architecture uses N worker threads (currently 8), each running its own epoll loop. Rather than having a single main thread accept connections and distribute them to workers via a queue, which introduces cross-thread synchronisation on every connection; each worker binds its own copy of the listen socket using `SO_REUSEPORT`. The kernel itself distributes incoming connections across workers with no mutex, no `eventfd` wakeup, and no shared queue. Each worker is entirely self-contained:

- its own `epoll` instance
- its own listen socket (`SO_REUSEPORT`)
- its own connection pool (`conn_pool_t`)

This design eliminates the cross-thread dispatch overhead that was previously responsible for roughly 20 % of latency under load. Compared to the earlier dispatch model, the hot path per connection drops from: `lock` → `memcpy` → `eventfd write` → `wakeup` → `lock` → `drain` → `epoll_ctl` to simply: `epoll fires` → `accept4` → `epoll_ctl`.

3 Implementation Challenges

The primary challenges were related to incremental I/O and state management. Unlike blocking implementations, requests may arrive in partial segments, requiring persistent buffering and repeated parsing attempts. Handling keep-alive connections introduced additional complexity, since multiple requests can coexist within the same input buffer.

A subtler challenge was the interaction between edge-triggered epoll (`EPOLLET`) and partial reads. With level-triggered epoll the kernel re-notifies as long as data is available, which is forgiving of incomplete

reads. With EPOLLET the notification fires exactly once on the state transition; if the entire available data is not drained in that single dispatch, the connection silently stalls. This required careful loop structures in handle_read to always drain until EAGAIN.

Another challenge specific to the multithreaded design was ensuring correct use of epoll_ctl. The EPOLL_CTL_MOD operation requires the file descriptor to already be registered via EPOLL_CTL_ADD; calling MOD on an unregistered fd returns ENOENT silently, leaving the connection permanently stalled. Systematic error checking on all epoll_ctl calls was necessary to surface these issues during development.

4 Experimental Setup

Benchmarks were performed using wrk:

```
wrk -t8 -c400 -d30s http://127.0.0.1:8080/<file>
```

The setup is a ThinkPad T16 with an Intel Core Ultra 7 155U (14 cores) @ 4.80 GHz with 32 GiB of RAM running Arch Linux. All benchmarks compare directly against nginx running on the same machine on port 8081, serving identical files.

5 Discussion

5.1 Effect of Keep-Alive

The transition to a non-blocking architecture significantly increased throughput, especially when combined with persistent connections. Without keep-alive, one request corresponds to exactly one TCP connection. This means that for every request the server must: accept(), TCP handshake, read the request, write the response, close socket. With keep-alive, one connection can carry many requests. The connection stays open, so we avoid repeating the accept and close process, which reduces syscall count and kernel allocations.

Measured impact of keep-alive (single-threaded baseline, ab -n 10000 -c 100):

Mode	Req/s	Notes
Without keep-alive	51,105	full TCP handshake per request
With keep-alive	152,935	connections reused across requests

Adding keep-alive does not especially reduce the latency of each individual request but adds significant throughput and scalability by removing the TCP setup cost.

5.2 Optimisation Journey

The server was profiled iteratively using perf record and perf report after each change. The following optimisations were applied in order of impact:

1. **SO_REUSEPORT per-worker sockets**: eliminated the centralised accept-and-dispatch model with eventfd wakeups. Each worker binds its own listen socket; the kernel distributes connections with no cross-thread synchronisation.

2. **Keep-alive typo fix**: a typo in the HTTP version comparison (HTTTP/1.1 instead of HTTP/1.1) caused every connection to be treated as HTTP/1.0 and closed after each response. Fixing this alone approximately doubled request throughput by enabling connection reuse.

3. **Parser cleanup**: replaced strncpy-based header copying with bounded memcpy using lengths already computed during scanning. Eliminated in-place buffer mutation (writing \0 into the buffer and restoring) which was cache-unfriendly and fragile.

4. **Eliminated dup() per request**: the original code called dup() on the cached file descriptor so that sendfile could have its own file offset. Since sendfile accepts an explicit offset pointer, the dup() is unnecessary; removing it eliminated a syscall and a close() on every request.

5. **TCP_CORK gating**: TCP_CORK was being set and unset on every request regardless of file size, adding two setsockopt syscalls per response. Gating behind a file-size threshold (only cork when file > 16 KiB) removed this overhead for the common case of small files.

5.3 Limitations

Performance behaviour can be interpreted through three regimes: connection-bound, CPU-bound, and network-bound execution. Enabling persistent connections moved the server from a connection-bound regime to a CPU-bound regime, significantly increasing throughput. The multithreaded design then distributed that CPU work across cores, pushing throughput further.

The remaining bottlenecks identified by perf are largely irreducible within the epoll architecture: HTTP parsing (∼11 %), epoll_ctl (∼10 %), read/write syscalls (∼10 % each), and sendfile (∼7 %). These represent the fundamental cost of serving a file over a socket.

6 Benchmark Results

Final benchmark — 100 KiB file After all optimisations, the server sustained 248,280 requests per second with a mean latency of 1.33 ms and a transfer rate of 23.70 GB/s. Under the same conditions nginx achieved 251,822 requests per second, placing the server at approximately 98.6 % of nginx throughput — a gap within run-to-run variance.

Server	Req/s	Avg latency	Transfer rate
This server	248,280	1.33 ms	23.70 GB/s
nginx	251,822	4.01 ms	24.07 GB/s

Table 1: Static file throughput, 100 KiB file, 8 threads, 400 concurrent connections, 30 s run.

Profiling summary Successive perf reports showed the hotspot distribution converging toward irreducible costs. The final profile attributes approximately 11 % of cycles to HTTP request parsing, 10 % each to epoll_ctl and read/write syscalls, and 7 % to sendfile. No single user-space function dominates; the remaining overhead is split across buffer_ensure_writable, http_response_write_file, and kernel time. This distribution indicates that the implementation has reached the practical ceiling of the epoll-based architecture for this workload.

Connection stability During long keep-alive stress tests, sporadic ECONNRESET errors were observed, consistent with wrk tearing down connections mid-flight at the end of a test window. These are suppressed in the final implementation by ignoring ECONNRESET and EPIPE on the read and write paths respectively. No errors were observed under normal single-connection curl workloads once the keep-alive parsing bug was corrected.

7 Future Work

io_uring The next architectural step would be to replace the epoll event loop with io_uring. Rather than one syscall per I/O operation, io_uring allows submitting a ring of operations and collecting completions in bulk via a single io_uring_enter call. This would reduce the cost of epoll_ctl, read, write, and sendfile — currently responsible for roughly 35 % of cycles — at the cost of a significant rewrite of the event loop and connection state machine.

Precomputed response headers Currently `http_response_write_file` builds the response headers at request time via several `buffer_append` calls. Since the files are cached at startup with known sizes and content types, the full response header string could be precomputed once per cached file and stored in `static_file_entry_t`, reducing the per-request header path to a single `memcpy`.

HTTPS Moving from HTTP to HTTPS requires an SSL/TLS certificate. OpenSSL is the standard way of doing it. Adding encryption changes the architecture by introducing a new TLS layer between the HTTP and connection layers. TLS will introduce CPU cost due to encryption and handshake overhead, which makes performance tracking more difficult. This feature is planned after the above optimisations.

8 Conclusion

As we have seen, non-blocking I/O combined with keep-alive and a multithreaded `SO_REUSEPORT` design allowed the server to reach throughput within run-to-run variance of `nginx` on static file serving.

This project really taught me how networking works at a very low level. I also really enjoyed going from a simple kind of 'single-cycle' server, into 'multi-cycle', into a 'superscalar' processor. And as a natural extension, the architecture would become a 'multi-core' server, which is exactly what it did.