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

Arthur Herbette

February 14, 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 and an epoll-based event loop to handle concurrent connections efficiently. We analyze architectural choices and evaluate performance under synthetic workloads.

1 Introduction

The goal of this project was to gain a deeper understanding of HTTP and network programming. You can find the github repositories at this link Therefore, I started this project as a simple blocking, one threaded, loop-based server. After understanding those fundamental I implemented headers parsing which enabled persistent (keep-alive) connections. The goal after that introduction is to have a very efficient server which is scalable. Those reasons motivated the transition to a non-blocking server architecture. Allowing the server to handle multiple connections concurrently using asynchronous I/O.

2 Architecture

The server follows an event-driven architecture based on a single-threaded reactor pattern. The architecture of this server is divided into 7 parts:

- **core**: coordinates the server and orchestrates interactions between components
- **epoll**: contains `struct` to be able to use `epoll` in our server loop
- **HTTP**: parse request, turns plain text into `http_request_t`
- **net**: handle connection with the socket, tcp connection
- **os**: provides low-level read/write operations and filesystem access
- **static**: handle all transaction which use static file (i.e., `index.html`)
- **util**: contains all utilities of the server, at the moment it contains only `buffer_t` which serves to store, consume all data from the connection

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

To have a better understanding about this server we can first explain what each layer is doing in our stack:

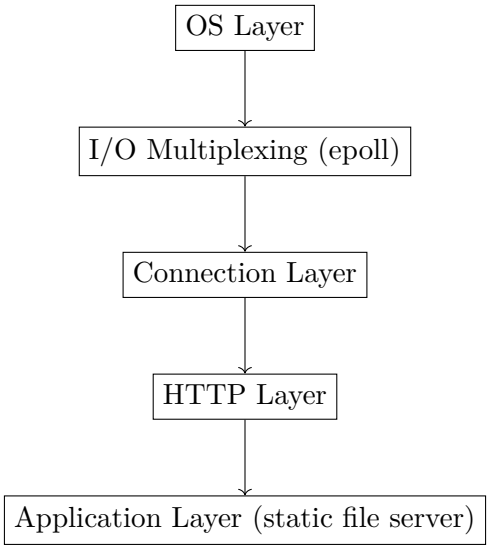

Figure 1: Layer stack

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

Each layer has its own responsibilities and purpose.

OS Layer The OS layer's responsibilities are:

- sockets
- file descriptors
- read/write/accept
- filesystem access

Event Layer The Event layer (epoll)'s purpose is to turn the 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
- keep-alive lifecycle

HTTP Layer As said before it only work with plain text, the goal is to:

- Parse request
- detect headers
- handle keep-alive
- build response

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 serve file but could later become:

- API endpoints
- dynamic content
- routing

2.1 State Machine

One of the issues during the developement of this server was that: as a connection does not complete in one step, when recieving an event for a connection, we needed to know the state of the connection to be able to proceed on the handling. To resolve this issue, I added the following state machine:

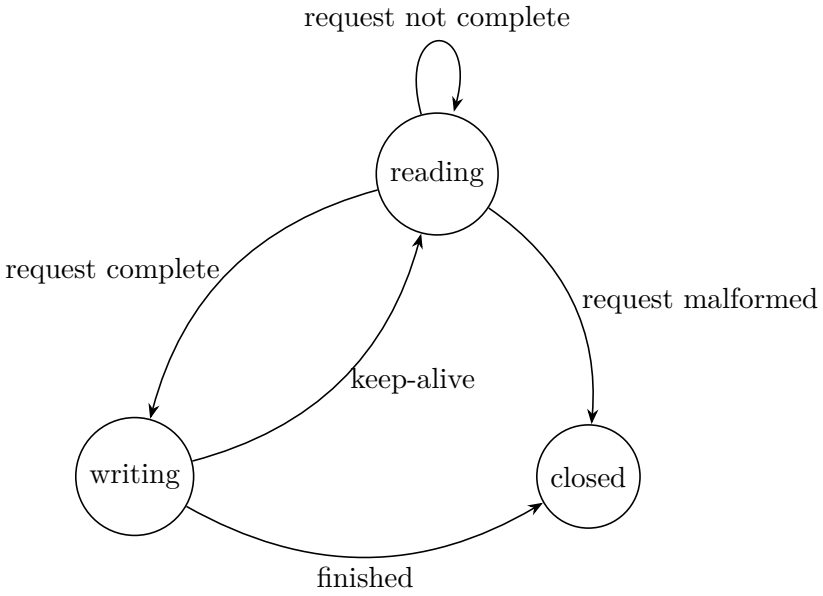

Figure 2: state machine of `connection_t`

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.

4 Experimental Setup

Benchmarks were performed using ApacheBench:

```
ab -n 10000 -c 100 -k http://localhost:8080/index.html
```

I also tried using `style.css` and an .ico file. The setup I am currenlty using is a ThinkPad t16 with a Intel(R) Core(TM) Ultra 7 155U (14) @ 4.80 GHz with 32 GiB of RAM (PS. I use arch btw).

File	Size [bytes]	Concurrency	Requests/sec	mean [ms]	Transfer rate[Kbytes/sec]
index.html	308	100	140,821	0.710	54595.74
style.css	1092	100	123,344	0.811	142255.26
favicon.ico	318	100	119104.32	0.084	49084.01

Table 1: Benchmark results

Remark Note that the 'mean' column here, represents the latency of each request, if we take the throughput meaning the mean, across all concurrent requests, the results are 0.007, 0.008, 0.008 ms respectively.

5 Discussion

The transition to a non-blocking architecture significantly increased throughput, especially when combined with persistent connections. However, this improvement came at the cost of increased implementation complexity, requiring explicit connection states and careful buffer management. Another improvement came when allowing connection to be keep-alive. Without having keep-alive, one request corresponds to only one TCP connection. This means that for every request the server must: accept(), TCP handshake, read the request, write response, close socket. Now with keep-alive we have that one connection can have many request. This implies that the connection stays open, we don't have to repeat the accept and close process which allows to have less syscalls and fewer kernel allocations.

If we take the number, I retried now to compare both with and without keep-alive, (I used the same benchmark as before `ab -n 10000 -c 100 -k http://localhost:8080/index.html`) without keep-alive, I get around 51105 request per second and as for the benchmark with

keep-alive, we get around 152935 requests per second.

Adding the keep-alive doesn't especially reduce the latency of each request but add more throughput to our server. This allows us to also have more scalability and be more stable (the TCP setup cost is removed).

5.1 Limitations

Performance behavior 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.

6 Future Work

As seen in the previous section the current bottleneck of this server is the CPU, there is a couple of way which may resolve this bottleneck:

Buffer At the current moment a buffer is used to store data. This means that the data flow go from `read` → `buffer_append`, we parse the buffer and then `buffer_consume` → `memmove`. This implies that we use `memmove()` a lot which becomes expensive at high request rate. A way of solving this issue would be that use a sliding window. Instead of moving the data around. Why not just move an index around? We have our buffer as before but now data is put into the buffer at a certain index which then would be updated everytime data is read into our buffer. Then parsing data and consuming data becomes fairly easy as it comes down to only updating the index.

Send files The `sendfile` command allows us to copy data from a file descriptor to another, as it works directly in the kernel, this syscall is faster than a `read` + `write`. At the moment I only tested on relatively small file and as I have lots a RAM it am not sure if it would greatly improve performance in our benchmark used above. **But** it will but when dealing with bigger file it will be a must to use this function.

Multi-threading The model at the moment is the following:

- One thread
- One epoll loop
- Many connections

A danger with multi-threading is the have mutliple thread handling the **same** connection. To solve this issue we have a main thread which accept connections → distribute sockets and then each worker threads has its own `epoll` and connection.
The goal is to have a queue for incoming HTTP requests and a pool of thread for satisfying them. The server take the first request from the queue and assignes a free thread from the pool. What would be running is basically n current server running independently.

HTTPS support Moving from HTTP to HTTPS requires an SSL/TLS certificate. If I understood right what is needed, OpenSSl is the standard way of doing it. By adding this new encryption the architecture of the server will actually change by adding a new TLS layer between the HTTP and the connection layer.
However TLS will introduce CPU cost (due to the encryption) which will make tracking the efficiency of our server more difficult. At the moment I think that I will implement this feature after the features mentionned above.

7 Conclusion

As we have seen before, non-blocking + keep-alive allowed our server too have to be really efficient and to handle a big concurrency.

This project really taught me how does network work on a really 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 extensions of the architecture would be 'multi-core' server.