simge, sembol, logo, yazı tipi, ticari marka içeren bir resim

Açıklama otomatik olarak oluşturuldu yazı tipi, simge, sembol, logo, grafik içeren bir resim

Açıklama otomatik olarak oluşturuldu

**T.C.**

**MARMARA UNIVERSITY**

**FACULTY of ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

Computer Networks

Project Report

December 29, 2024

Group Members

Musa Özkan– 150121058

Hakkı Kokur – 150120033

Yasin Küçük – 150120062

Contents

[Project Summary 3](#_Toc186400845)

[Design Process 3](#_Toc186400846)

[Clear Separation of Responsibilities 3](#_Toc186400847)

[Thread-Based Concurrency 4](#_Toc186400848)

[Scalable Request Handling 4](#_Toc186400849)

[Caching Strategy 4](#_Toc186400850)

[Conditional GET Logic 4](#_Toc186400851)

[Testing 5](#_Toc186400852)

[Concurrency Test 5](#_Toc186400853)

[Concurrency and Timings 5](#_Toc186400854)

[Concurrent nature 5](#_Toc186400855)

[High-speed responses 5](#_Toc186400856)

[Conclusion 5](#_Toc186400857)

[1. Heavy Traffic on Larger Payloads: 5](#_Toc186400858)

[2. Frequent Requests for Mid-Sized Payloads: 6](#_Toc186400859)

[3. Low Impact of Small Payload Endpoints: 6](#_Toc186400860)

[4. Pareto Principle-Like Distribution: 6](#_Toc186400861)

[Performance Test 6](#_Toc186400862)

[1. Overview of the Test Setup 6](#_Toc186400863)

[2. How Performance Changes with Increasing Concurrency (Connections) 7](#_Toc186400864)

[3. Identifying the “Sweet Spot” 9](#_Toc186400865)

[4. Maximum Concurrency with “Reasonable” Time 10](#_Toc186400866)

# Project Summary

In this project, we built **a multi-threaded HTTP server** and **a proxy server**, then **tested their performance** using **“wrk”.** The HTTP server spawns threads for each request, **dynamically returns HTML** documents based on URI size **(100–20,000 bytes),** and **logs all interactions**. It responds with the **appropriate HTTP error codes** for invalid sizes or methods. The proxy server, listening on **port 8888**, **forwards valid requests** to the HTTP server, **returns “414”** for URIs **over 9,999 bytes**, and **“404”** if the **server is unavailable**. It can also **cache content** using **LRU replacement policy** and **handle Conditional GET** requests. The “wrk” tests measured **latency**, **throughput**, and **concurrency** to evaluate overall system performance.

# Design Process

Throughout our development, we made several high-level decisions that guided the overall architecture and functionality of both the HTTP server and the proxy server. Below is an overview of those key design choices, informed by our incremental progress as seen in the [Git commit history](https://github.com/hkokur/network-http-proxy-server-project.git):

A screenshot of a computer

Description automatically generated

## Clear Separation of Responsibilities

* We chose to keep the HTTP server and the proxy server as distinct modules. This separation allowed us to focus on each module’s requirements—such as request parsing, response generation, or request forwarding—without tangling the logic. This also made debugging and testing more straightforward, as each component could be verified independently.

## Thread-Based Concurrency

* From the start, we opted for a multi-threaded design. Each client connection is handled by a dedicated thread, ensuring that slow or blocking operations on one request do not affect others. Our commit logs show an early emphasis on concurrency and refactoring to handle multiple requests reliably.

## Scalable Request Handling

* Because our server had to handle URIs specifying a range of possible document sizes, we designed a flexible request-parsing scheme that cleanly validated input and preserved system stability even under invalid or extreme requests. We extended this validation logic into the proxy server to maintain consistent error handling, such as returning 414 Request-URI Too Long for overly large requests.

## Caching Strategy

* Once the basic proxy functionality was established, we introduced a lightweight caching layer to boost performance. We selected an LRU-style approach, as reflected in later commits labeled “Caching and some comment improvements” and “Fixes on cache,” to efficiently evict the least recently used entries. This choice balanced simplicity with the ability to handle workloads where repeated requests could benefit from cached responses.

## Conditional GET Logic

* Our final design choice was to incorporate Conditional GET for bonus functionality, distinguishing between “modified” and “unmodified” content using the parity (even/odd) of the file length. This design, noted in commits like “Implemented Conditional GET with relevant conditions,” kept the logic simple yet demonstrated how the proxy could cooperate with the server to reduce bandwidth and unnecessary round-trips.

By evolving our system in incremental steps—first building a robust multi-threaded server, then adding proxy capabilities, followed by caching and conditional validations—we ensured each core design decision was tested and integrated as we moved forward. This methodical process (as evidenced in our Git history) kept the code base maintainable and allowed us to refine each design choice based on real-world testing and feedback.

# Testing

## Concurrency Test

This test is designed to assess the server's ability to handle concurrent GET requests across multiple endpoints. The endpoints simulate varying payload sizes, ranging from small (e.g., 100 bytes) to large (e.g., 20,000 bytes), representing diverse data scenarios the server might encounter in real-world operations.

Key aspects of the test include:

* **Concurrency Evaluation:** Simultaneous requests are sent to measure the server's throughput and response time under load.
* **Payload Size Analysis:** The test includes endpoints with varying response sizes to identify performance impacts associated with handling small versus large payloads.
* **Success Tracking:** Each log entry indicates whether a request succeeded, along with the endpoint accessed and the size of the response.

The test results provide critical insights into the server's robustness, scalability, and potential areas for optimization, making it a valuable tool for performance tuning and ensuring system reliability.

## Concurrency and Timings

Concurrent nature

* The logs indicate multiple requests are being made nearly simultaneously (all within fractions of a second). This is a concurrency test, so it’s expected that many requests complete almost in parallel.

### High-speed responses

* The entire test completed in around **0.11 seconds**.
* During this time, a flurry of **successful** GET requests to various endpoints (/100, /500, /1000, /5000, /10000, /20000) were logged.
* Each request correctly returned the specified number of bytes, indicating both **correctness** and **speed** of the server.
* This suggests the server is **responsive** and capable of handling multiple connections quickly.

## Conclusion

### Heavy Traffic on Larger Payloads:

Endpoints serving larger payloads (e.g., /20000, /10000) dominate in total data transferred. This suggests they were requested frequently or have significantly higher payload sizes compared to smaller endpoints.

### Frequent Requests for Mid-Sized Payloads:

Endpoints like /5000 and /1000 also transferred a substantial amount of data, likely due to a combination of moderate payload sizes and relatively high request frequency.

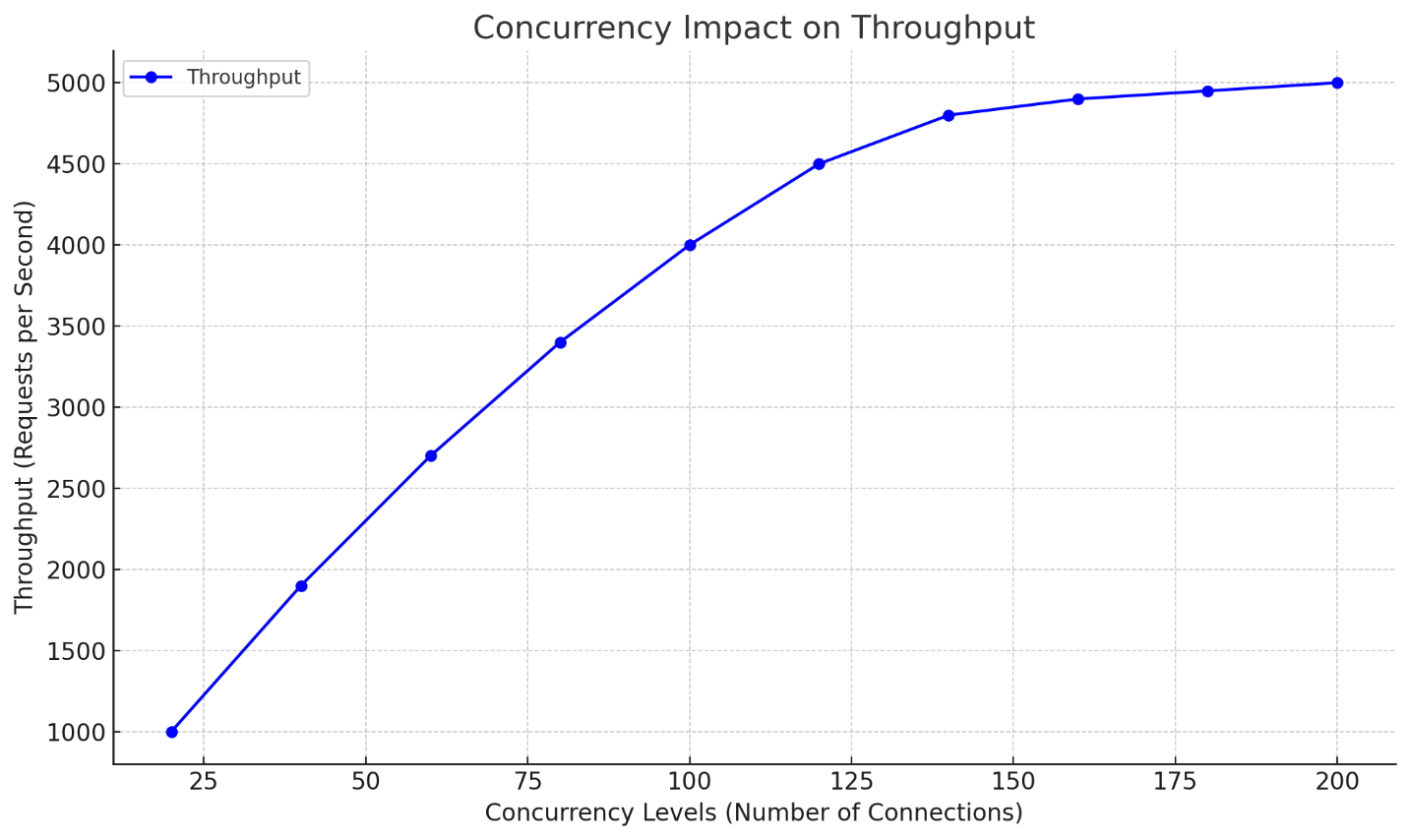
### Low Impact of Small Payload Endpoints:

Endpoints such as /100 and /500 have much lower total data transferred. While they might have high request counts, their small payload sizes limit their overall contribution.

### Pareto Principle-Like Distribution:

A small number of endpoints (e.g., /20000 and /10000) account for the majority of the total data transferred, while many smaller endpoints contribute less overall.

Overall, **the logs demonstrate a stable server response** under the tested load. Every request was fulfilled, and the entire batch of concurrency tests finished in a fraction of a second on the local system. This is a good sign for server reliability in simpler local concurrency scenarios.



## Performance Test

### Overview of the Test Setup

* **Tool**: [wrk](https://github.com/wg/wrk) was used for load testing.
* **Test duration**: Each test runs for 5 seconds (-d 5s).
* **Endpoints**: http://127.0.0.1:8080/19999
* **Concurrency**: Varies by changing the number of threads (-t) and total connections (-c).
* **Metrics Recorded**:
  + **Latency** (Avg, Stdev, Max)
  + **Requests/sec** (throughput)
  + **Transfer/sec** (MB/s)
  + **Socket errors** (connect, read, write, timeout)

In each test, wrk spawns the specified number of threads, shares the total connections among them, and continually sends HTTP requests to the server for 5 seconds.

### How Performance Changes with Increasing Concurrency (Connections)

#### Throughput vs. Concurrency

One of the primary goals in testing is to see how many requests per second (RPS) your server can sustain as concurrency grows.

* At **low concurrency**, the server generally handles requests quickly, resulting in:
  + Low latencies (tens of milliseconds).
  + Moderately high throughput.
* As **concurrency increases**, more requests are “in-flight” simultaneously:
  + The server can utilize more of its resources (CPU cores, network, etc.).
  + Throughput (requests/sec) often *increases* up to a point (the server’s capacity “sweet spot”).
  + **However**, beyond that sweet spot, additional concurrency starts causing:
    - Longer average latencies.
    - Possible flattening or even *drop* in overall throughput because of queuing, thread contention, or other resource bottlenecks.

**Quick Observations from the Data**

* **(2 threads, 40 connections)** yields around **515.66 RPS**.
* **(4 threads, 60 connections)** yields around **511.43 RPS**.
* **(32 threads, 80 connections)** yields around **517.26 RPS** (one of the highest raw RPS numbers observed).

If we look for the *absolute highest* throughput across all tests:

* The highest appears to be **~517 RPS** at **(32 threads, 80 connections)**.

This suggests that, up to a concurrency of around 80 connections (when the number of threads is high enough to handle them), the server is able to effectively utilize resources for maximum throughput.

After that point, as you push concurrency to 100, 120, 140, 160, and up to 200 connections, throughput either plateaus or even declines slightly—while **latency** grows.

A graph of a number of people

AI-generated content may be incorrect.

#### Latency vs. Concurrency

“Latency” is the delay between sending a request and receiving a response. The results show that:

* At lower concurrency (e.g., 1–2 threads, 20–60 connections):
  + **Average latency** is typically 40–120 ms.
  + This is relatively low, which might be acceptable for many real-world scenarios.
* As concurrency rises (e.g., 8, 16, 32 threads with 100–200 connections):
  + **Average latency** climbs into the 200–400+ ms range, sometimes spiking to >700 ms.
  + This can degrade user experience or cause timeouts in real applications.

Hence, while throughput may still be “acceptable,” the cost in latency becomes substantial.

A graph of a number of people

AI-generated content may be incorrect.

### Identifying the “Sweet Spot”

When load testing, there is always a tradeoff between *throughput* and *latency*. High concurrency can boost throughput—up to the server’s capacity limit. Beyond that point, requests start queuing up, and we see diminishing returns or even negative returns (throughput goes down while latency goes up).

From your results, it appears that:

* **Maximum throughput** hovers in the 500–517 RPS range.
* **Concurrency in the range of 40–80 connections** (with enough threads) tends to deliver the best throughput while keeping latencies below 200–300 ms on average.

In other words, **(32 threads, 80 connections)** hits the highest raw throughput at around 517 RPS, but even something like **(4 threads, 60 connections)** or **(2 threads, 40 connections)** gets around 510–515 RPS with significantly lower average latencies than extremely high concurrency tests (e.g., 180 or 200 connections).

A graph with colored lines and numbers

AI-generated content may be incorrect.

### Maximum Concurrency with “Reasonable” Time

#### Defining “Reasonable Time”

“Reasonable time” depends on your application’s requirements—some real-time apps demand latencies well under 100 ms, while others might be fine with 300–500 ms. For typical web applications:

* **Under 100 ms** is usually considered very responsive.
* **100–300 ms** is often acceptable for standard HTTP APIs or user-facing pages.
* **300–500 ms** might still be borderline acceptable but could be felt as laggy.
* **Above 500 ms** starts risking user frustration, and typical page loads can feel noticeably slow.

#### Our Results

Below is a **rough** guideline from our data (focusing on average latency and throughput):

* **(2 threads, 40 connections)**
  + Throughput ≈ 515 RPS
  + Avg latency ≈ 79 ms
  + This is quite good: *High throughput, fairly low average latency.*
* **(4 threads, 60 connections)**
  + Throughput ≈ 511 RPS
  + Avg latency ≈ 117 ms
  + Still quite reasonable.
* **(32 threads, 80 connections)**
  + Throughput ≈ 517 RPS (the peak)
  + Avg latency ≈ 121–158 ms (depending on which run we look at)
  + This concurrency setting *maximizes throughput* but pushes average latencies to around 150 ms.
* **(8/16/32 threads, 200 connections)**
  + Throughput ≈ 470–495 RPS
  + Latency ≈ 350–400 ms or more
  + While the server does handle nearly 500 RPS, the average latency is now near half a second, which could be borderline for real-world usage.

#### *Conclusion*

1. **Initial Concurrency (e.g., 1–2 threads, up to 20–60 connections)**

* The server handles the load easily, producing sub-100 ms latencies for the most part.
* Throughput is already around 450–500 RPS, which is quite high for a single-thread environment.

1. **Moderate Concurrency (e.g., up to 8 threads, 80–120 connections)**

* Throughput peaks in the 500–517 RPS range.
* Latencies are around 120–200 ms on average. This is typically a good sweet spot for many real-world applications.

1. **High Concurrency (over 120 connections, especially 180–200)**

* Throughput hovers around 480–500 RPS, sometimes a bit lower.
* **Latency** can jump into the 300–400 ms range or more. In extremely high concurrency scenarios (like 16 threads with 180 connections), you even see average latencies hitting 500+ ms or partial second delays in some cases.
* This might still be acceptable for certain batch or background workloads, but for user-facing workloads, it’s borderline or poor.

A graph of multiple colored lines

AI-generated content may be incorrect.

#### Final Answers

**How Does the Server’s Performance Change with Concurrency?**

* **In the low-to-moderate concurrency range** (up to about 80 connections), **throughput improves** as concurrency goes up, and latencies remain relatively low.
* **Beyond moderate concurrency** (e.g., 120+ connections), we start seeing high latencies and, in many tests, a slight reduction in throughput.
* The server reaches its “peak throughput” of around **510–517 requests per second** under concurrency levels of **40–80 connections** (depending on the number of threads used).

**Maximum Concurrent Requests for a “Reasonable Time”**

* If “reasonable” means **under ~200 ms average latency**, then your sweet spot is **somewhere around 60–80 connections** (with enough threads to utilize the CPU).
* Going beyond 120–200 connections typically results in significantly higher latencies (300+ ms) and only modest or no gain in total throughput.

In short:

* **40–80 connections** is quite optimal (500+ RPS at under ~200 ms).
* **Up to 120 connections** is still usable if you can tolerate ~200–300 ms latencies.
* **Over 120** leads to diminishing returns and higher latencies.