CS 553 : PROJECT REPORT

Adaptive Reverse-Proxy Load Balancer

Shrey Patel (sp2675) Abhishek Jani (aj1121) Mustafa Adil (ma2398)

Source Code:
https://github.com/abhishekjani123/DIS---Adaptive-Reverse-Proxy-Load-Balancer

Contents

List of Tables

List of Figures

1 Resources Used

This project utilized a variety of tools, libraries, and technologies to achieve its objectives. The key resources are listed below:

- **Python 3.9+:** The primary programming language used for developing the proxy server, backend servers, and dashboard logic. https://www.python.org/
- **aiohttp:** An asynchronous HTTP client/server framework for Python, crucial for building the non-blocking proxy and backend servers. https://docs.aiohttp.org/
- **Flask:** A lightweight WSGI web application framework in Python, used to create the web server for the real-time dashboard. https://flask.palletsprojects.com/
- **pandas:** A powerful Python data analysis and manipulation library, employed by the dashboard for processing log data. https://pandas.pydata.org/
- **wrk / wrk2:** A modern HTTP benchmarking tool used to generate load and measure the performance of the proxy server setup. https://github.com/wg/wrk (Original wrk)
- **Docker:** A platform for developing, shipping, and running applications in containers, used here to isolate and manage the backend server instances. https://www.docker.com/
- **Plotly.js:** A high-level, declarative JavaScript charting library used to create interactive graphs for latency trends in the real-time dashboard. https://plotly.com/javascript/
- **Chart.js:** A simple yet flexible JavaScript charting library used for displaying request distribution bar charts in the real-time dashboard. https://www.chartjs.org/
- **Python 'asyncio' Library:** The standard Python library for writing concurrent code using the async/await syntax, fundamental for asynchronous operations like simulating network delays. https://docs.python.org/3/library/asyncio.html
- **Python 'itertools' Library:** A standard Python module that implements a number of iterator building blocks, used for features like cycling through latency patterns or backend servers. https://docs.python.org/3/library/itertools.html
- **LaTeX:** A high-quality typesetting system used for the preparation and formatting of this project report. https://www.latex-project.org/
- **ChatGPT (OpenAI):** Utilized for assistance with report formatting, code debugging, and clarifying commands throughout the project development. Used for understanding reverse proxies and different approaches for implementing them. https://chat.openai.com/
- **Grammarly:** An online writing assistance tool used to improve sentence structure, grammar, and clarity in the project report. https://www.grammarly.com/
- **YouTube:** Various tutorials and informational videos were consulted for assistance with Docker setup, concepts, and troubleshooting. https://www.youtube.com/

2 Abstract

This project main idea is HTTP reverse proxy load balancer, which we built using Python. Its job is to take all the requests coming from users on the internet and spread them out smartly to different computer servers at the back end. We tried out two main ways of setting up how the proxy talks to these backend servers: one way where it makes a new connection for every single request (we call this 'non-persistent'), and another way where it keeps connections open and reuses them for many requests ('persistent'). We built both of these setups to work with three different methods for deciding where to send each request: a simple 'Round-Robin' method that just cycles through the servers, and two 'smarter' methods that learn from how fast servers are responding, called 'Adaptive Simple Moving Average' (SMA) and 'Adaptive Exponential Weighted Moving Average' (EWMA). To really see how well these smarter methods work, we made our backend servers pretend to have different speeds and delays. We used a tool called 'wrk' to send lots of requests to our system, like it was under a real workload, and we carefully measured and compared how well everything performed. Our findings show that keeping connections open (the 'persistent' way) really helps cut down on extra work and speeds things up. Also, the better 'adaptive' methods are very good at avoiding situations where users get stuck with really long waits, especially when some servers are slow. This project gives us a good, flexible setup that we can use to test out new ideas for these kinds of proxy systems.

3 Introduction

Distributed systems nowadays depend on load balancers to effectively distribute client requests among numerous instances of service in an attempt to achieve scalability, reliability, and low latencies. Two key design decisions that affect proxy performance are the strategy used in selecting a backend and recycling or not recycling TCP connections. Connection persistence (HTTP/1.1 keep-alive) may be used to help reduce handshake overhead, whereas adaptive routing algorithms utilize observed latency readings to stay away from bad-performing servers.

This project deploys a reverse proxy in Python with aiohttp and compares persistent and non-persistent connection policies for Round-Robin (RR), Adaptive Simple Moving Average (SMA), and Adaptive Exponential Weighted Moving Average (EWMA) routing policies. We emulate three backend servers with heterogeneous latency profiles: bursty fast-and-slow spikes, uniform medium, and uniform slow. With the wrk benchmarking tool under controlled load, we compare throughput (requests per second), mean latency, and tail latency (p95).

4 Our Goal

We aim to:

1. Contrast the performance difference between per-request connection establishment and connection re-use.
2. Assess the effectiveness of adaptive routing (SMA, EWMA) at reducing tail latency.
3. Offer a stand-alone experimental setup with extensive telemetry for follow-on research.

The traffic flows as follows:

```
Client (wrk) --> Proxy (9090) --> Docker (Backend A on Port 8081)
(many users) |
             +------------------> Docker (Backend B on Port 8082)
             |
             +------------------> Docker (Backend C on Port 8083)
```

In this setup, the proxy server communicates with the backend servers running inside their respective Docker containers. Docker helps isolate the backend server processes and manage their resources.

5 Key Features being Tested

- **Persistence Toggle:** The proxy server can operate in two modes for connecting to backend servers:
  - **Non-Persistent:** For each incoming client request, the proxy opens a new connection to a backend server and closes it after the response is received. This involves a "greeting process" (TCP handshake) for every single request.
  - **Persistent:** The proxy opens connections to backend servers and keeps them open, reusing them for multiple requests. This avoids the repeated "greeting process."
- **Routing Mode Switch:** The proxy can use one of three algorithms to decide which backend server gets the next request:
  - Round-Robin (RR)
  - Simple Moving Average (SMA)
  - Exponential Moving Average (EWMA)

By testing combinations of these features, we can see how connection management and routing logic affect overall system performance.

6 Directory / File Structure

The project files are organized as follows:

```
project_root/
|
|-- backend_server.py              # Backend server code
|-- proxy_server_non_persistent.py # Proxy with non-persistent connections
|-- persistent_proxy_server.py     # Proxy with persistent connections
|-- dashboard.py                   # Flask server for the live dashboard
|-- dashboard.html                 # HTML for the live dashboard
|-- Dockerfile                     # For containerizing the backend server
|
|-- logs/
|   |-- *.csv                      # Per-request logs from proxy (one per run)
|   |-- *.txt                      # Console captures, wrk outputs
```

This structure separates the server code, logging outputs, and any analysis tools. The results presented in this report are derived from the `*.csv` and `*.txt` files in the `logs/` directory.

7 Environment & Prerequisites

To run this project and reproduce the results, the following software and versions were used (or are recommended):

- **Python:** Version 3.9+ (specifically, Python 3.9-slim was used for Docker)
- **aiohttp:** Version 3.9+ (for asynchronous web server and client)
- **pandas:** Version 2.x (for data analysis from CSV logs)
- **matplotlib:** Version 3.x (for generating graphs, used by `analysis_notebook.ipynb`)
- **Flask:** For `dashboard.py` (version not specified, but recent versions should work)
- **wrk:** Version 4.1+ (HTTP benchmarking tool)
- **Docker:** (For running backend servers in containers) A recent version.

It's important to ensure that no other applications are using network ports 8081, 8082, 8083 (for backends), 9090 (for the proxy), or 5002 (default for the dashboard).

8 Backend Servers (Ports 8081-8083)

The backend servers are the main component of our system. They are designed to simulate application servers that process requests and send back responses. To make our load balancing tests meaningful, these servers have intentionally different performance characteristics.

8.1 Latency Profiles

Each backend server follows a predefined sequence of response times (latencies), measured in milliseconds (ms). This sequence repeats, creating a predictable yet varied performance profile.

- **Server A (Port 8081):** This server has a highly variable latency. Its pattern is: 10ms, 15ms, 20ms (fast), then 400ms, 400ms, 400ms (slow spikes), then 10ms, 15ms, 20ms (fast again). This 9-step cycle is repeated 5 times, making a full cycle of 45 responses.
  - *Latency sequence (one cycle of 9):* [10, 15, 20, 400, 400, 400, 10, 15, 20] ms
  - *Average Latency for Server A:* $\frac{(10+15+20)\times 2+(400\times 3)}{9} = \frac{90+1200}{9} = \frac{1290}{9} \approx 143.33$ ms
- **Server B (Port 8082):** This server provides a consistent, medium latency. Its pattern is 200ms for 9 responses, repeated 5 times.
  - *Latency sequence (one cycle of 9):* [200, 200, 200, 200, 200, 200, 200, 200, 200] ms
  - *Average Latency for Server B:* 200 ms
- **Server C (Port 8083):** This server is consistently the slowest. Its pattern is 250ms for 9 responses, repeated 5 times.
  - *Latency sequence (one cycle of 9):* [250, 250, 250, 250, 250, 250, 250, 250, 250] ms
  - *Average Latency for Server C:* 250 ms

These distinct profiles allow us to clearly see how different load balancing algorithms adapt to server performance changes.

8.2 Code Walk-through (`backend_server.py`)

The `backend_server.py` script is an asynchronous HTTP server built using 'aiohttp'.

- `LATENCY_CONFIG`**:** A Python dictionary stores the latency sequences for each server ID ('A', 'B', 'C').
- `itertools.cycle`**:** For each server, an infinite iterator (`latency_cycle`) is created from its latency list. This makes the server repeat its latency pattern endlessly.
- `async def handle(request)`**:** This function is called for every incoming request. It retrieves the next latency value from the server's cycle, simulates a delay using `asyncio.sleep(latency_ms / 1000.0)`, and then returns a simple text response indicating the server ID and the simulated latency.
- **Environment Variables:** The `SERVER_ID` (e.g., 'A', 'B', 'C') and `PORT` (e.g., 8081) are read from environment variables at startup. This allows running multiple instances of the same script, each configured as a different backend server. If `SERVER_ID` is not found, it defaults to 'A'.

8.3 Run Command

To start the three backend servers, you would typically open three separate terminal windows and run:

Terminal 1 (Server A):

```
docker run --rm -p 8081:8080 -e PORT=8080 -e SERVER_ID=A backend-server:
    latest
```

Terminal 2 (Server B):

```
docker run --rm -p 8082:8080 -e PORT=8080 -e SERVER_ID=B backend-server:
    latest
```

Terminal 3 (Server C):

```
docker run --rm -p 8083:8080 -e PORT=8080 -e SERVER_ID=C backend-server:
    latest
```

8.4 Dockerization (`Dockerfile`)

A `Dockerfile` is provided to containerize the backend server application, making it easy to deploy and manage, especially if running multiple instances.

```
FROM python:3.9-slim
WORKDIR /app
RUN pip install --no-cache-dir aiohttp
COPY backend_server.py .
EXPOSE 8080
ENV SERVER_ID=Default
ENV PORT=8080
CMD ["python3", "backend_server.py"]
```

Listing 1: Dockerfile for Backend Server

9 Proxy Variants

The proxy server is the core of our load balancing experiment. It sits between the client (‘wrk‘) and the backend servers. We implemented two main versions of the proxy, differing in how they manage connections to the backend servers.

9.1 Non-Persistent Proxy (`proxy_server_non_persistent.py`)

This version of the proxy establishes a new connection to a backend server for every single client request it handles.

- **Connection Strategy:** Inside its request handling function (`handle_proxy`), it creates a new `aiohttp.ClientSession`. To ensure the connection is not reused and is closed promptly, the session’s `TCPConnector` is configured with `force_close=True`. Additionally, it adds a ‘Connection: close‘ header to the request sent to the backend server. This tells the backend server to also close its side of theconnection after sending the response.

- **Impact:** This approach adds overhead because each request involves a TCP handshake (the "greeting process" to establish a connection) and a teardown process. We expect this to result in lower overall throughput and potentially higher latencies compared to using persistent connections.

- **Console Log Confirmation:** During tests with this proxy, console logs include lines like:

```
[Debug] backend session closed? True
[Debug] resp.force_close? True

```

  This confirms that the proxy is indeed closing connections after each use.

9.2 Persistent Proxy (`persistent_proxy_server.py`)

This version of the proxy aims to be more efficient by reusing connections to backend servers.

- **Connection Strategy:** At startup, a single, global `aiohttp.ClientSession` is created. This session uses a `TCPConnector` configured with `limit=0`, which means there's no limit to the number of pooled (reusable) sockets. When handling a client request, the proxy uses this shared session to send requests to the backend. It does *not* set the 'Connection: close' header, allowing the connection to remain open for future requests.

- **Impact:** By reusing connections, this proxy avoids the repetitive overhead of TCP handshakes and teardowns for each request. We anticipate this will lead to higher throughput and lower latencies.

- **Console Log Confirmation:** Logs from this proxy show:

```
[Debug] backend session closed? False
[Debug] resp.force_close? <bound method StreamResponse.force_close of <Response OK not
    prepared>>
[Debug] transport.is_closing()? False (or None initially)

```

  This indicates that connections are kept alive and reused.

9.3 Common Features

Both proxy variants share the following:

- **Routing Selector:** A function `choose_backend()` is responsible for selecting which backend server to route a request to, based on the chosen routing algorithm (RR, SMA, or EWMA).

- **Logging:** Every request processed by the proxy (whether successful or an error) results in a line being appended to a CSV file (e.g., `proxy_log.csv` or more specific names like `round_robin_persistent.csv`). This log includes a timestamp, the chosen backend URL, the latency experienced for that request (from the proxy's perspective), the HTTP status code, and the routing mode used.

10 Routing Algorithms in Depth

The core functionality of our load balancer lie in its routing algorithm – the logic it uses to decide which backend server should handle an incoming request. We tested three common algorithms.

10.1 Round-Robin (RR)

This is the simplest load balancing algorithm.

- **Logic:** It maintains a list of available backend servers and cycles through them in order. The first request goes to Server A, the second to Server B, the third to Server C, the fourth back to Server A, and so on.
- **Implementation:** Uses `itertools.cycle(BACKEND_SERVERS)` to get the `next(backend_cycle)`.
- **Pros:** Very simple to implement, ensures an even distribution of requests if all servers are equally capable. In our case, it guarantees each backend receives approximately 33.3% of the traffic over time.
- **Cons:** It doesn't consider the current load or response time of the servers. If one server becomes slow or overloaded, Round-Robin will continue sending requests to it, potentially degrading performance for users routed to that server.

10.2 Simple Moving Average (SMA)

This algorithm tries to be smarter by sending requests to the server that has been responding fastest recently.

- **Logic:** For each backend server, the proxy keeps track of the latencies of its last few responses (in this project, the last $N = 3$ responses). It calculates the simple average of these latencies. The proxy then chooses the backend server with the smallest (lowest) SMA.
- **Formula:** For a window of N latencies $(L_1, L_2, ..., L_N)$:

$$\text{SMA} = \frac{L_1 + L_2 + \cdots + L_N}{N}$$

- **Cold Start:** If a backend server hasn't received any requests yet (its list of recent latencies is empty), its SMA would be undefined or infinite. To handle this "cold start," the algorithm will probe such a server to get an initial latency sample. This is visible in console logs, e.g.:

```
[SMA] probing unmeasured: http://localhost:8081

```

- **Pros:** Adapts to changing server performance. If a server starts to slow down, its SMA will increase, and it will receive fewer requests.

- **Cons:** Can be a bit slow to react to sudden spikes if the window size (N) is large. If N is too small, it might react too aggressively to temporary fluctuations.

10.3 Exponential Moving Average (EWMA)

This is another adaptive algorithm, similar to SMA, but it gives more weight to more recent latencies.

- **Logic:** EWMA also calculates an average latency for each server, but instead of a simple average over a fixed window, it uses a smoothing factor, α (alpha), to incorporate new latency measurements. A smaller α results in smoother changes and relies more on past history, while a larger α makes the EWMA react more quickly to recent changes. In this project, $\alpha = 0.2$.
- **Formula:**

$$\text{ewma}_{\text{new}} = (\alpha \times \text{current\_latency}) + ((1 - \alpha) \times \text{ewma}_{\text{previous}})$$

- **Warm-up/Probing:** Similar to SMA, if a backend server has no EWMA calculated yet (e.g., `ewma == None` or `float('inf')` in the code), it's considered unsampled, and the proxy will probe it to get an initial latency value. The console logs show this:

```
[EWMA] probing unmeasured: http://localhost:8083

```

  And also show the calculated EWMAs during operation:

```
[EWMA] EWMAs: 8081=405.1ms, 8082=211.8ms, 8083=282.8ms
[EWMA] chosen: http://localhost:8082

```

- **Pros:** Generally considered more responsive to recent changes than SMA, depending on the choice of α.
- **Cons:** The choice of α is crucial and might need tuning based on the specific application and server behavior.

11 Logging & Data Collection

To understand how our proxy server and its different strategies perform, we need to collect data. This project uses several methods for logging.

11.1 CSV Request Logs

For every request handled by the proxy server, a detailed entry is recorded in a Comma Separated Values (CSV) file. A new CSV file is typically used for each test run (e.g., `round_robin_persistent.csv`, `adaptive_sma_non_persistent.csv`). The columns in the CSV log are:

- `timestamp`: The date and time (with millisecond precision) when the proxy finished processing the request.
- `backend_url`: The URL of the backend server that the request was routed to (e.g., `http://localhost:8081`).
- `latency_ms`: The time taken, in milliseconds, for the proxy to get a response from the backend server. This is measured from when the proxy sends the request to when it receives the full response.
- `status_code`: The HTTP status code returned by the backend server (e.g., 200 for success).
- `routing_mode`: The routing algorithm active when this request was processed (e.g., `round-robin`, `adaptive_sma`, `adaptive_ewma`).

This CSV data is the primary source for our quantitative performance analysis. An example of the log format is shown in Figure 1.

11.2 Console Output

The proxy server also prints information to the console (terminal) as it runs. This is useful for real-time monitoring and debugging. Examples from the console logs:

```
[RR] chosen: http://localhost:8082
[SMA] chosen: http://localhost:8081
[EWMA] chosen: http://localhost:8082
```

For adaptive algorithms, the calculated average latencies:

```
[SMA] SMAs: 8081=12.3ms, 8082=200.7ms, 8083=250.0ms
[EWMA] EWMAs: 8081=405.1ms, 8082=211.8ms, 8083=282.8ms
```

Debug messages confirming connection persistence behavior: Persistent proxy:

```
[Debug] backend session closed? False
```

Non-Persistent proxy:

```
[Debug] backend session closed? True
[Debug] resp.force_close? True
```

These console logs (e.g., `round_robin_persistent_console.txt`) were saved for each test run.

11.3 `wrk` Benchmark Output

The ‘wrk‘ tool, which we use to generate load, also produces a summary report at the end of each test. This report provides an overview of the test from the client's perspective. A screenshot of a typical `wrk` output is presented in Figure 2. Key information from ‘wrk‘ output (e.g., `round_robin_persistent_wrk.txt`):

- **Requests/sec (Throughput):** The average number of requests processed by the system per second.

proxy_log.csv ×

proxy_log.csv

```
timestamp,backend_url,latency_ms,status_code,routing_mode
2025-05-09T20:04:04.679379,http://localhost:8081,42,200,adaptive_sma
2025-05-09T20:04:04.680189,http://localhost:8081,51,200,adaptive_sma
2025-05-09T20:04:04.680439,http://localhost:8081,49,200,adaptive_sma
2025-05-09T20:04:04.683334,http://localhost:8081,46,200,adaptive_sma
2025-05-09T20:04:04.683711,http://localhost:8081,44,200,adaptive_sma
2025-05-09T20:04:04.683972,http://localhost:8081,44,200,adaptive_sma
2025-05-09T20:04:04.684188,http://localhost:8081,45,200,adaptive_sma
2025-05-09T20:04:04.684310,http://localhost:8081,45,200,adaptive_sma
2025-05-09T20:04:04.684430,http://localhost:8081,41,200,adaptive_sma
2025-05-09T20:04:04.684687,http://localhost:8081,41,200,adaptive_sma
2025-05-09T20:04:04.684882,http://localhost:8081,39,200,adaptive_sma
2025-05-09T20:04:04.685021,http://localhost:8081,45,200,adaptive_sma
2025-05-09T20:04:04.685135,http://localhost:8081,44,200,adaptive_sma
2025-05-09T20:04:04.685243,http://localhost:8081,41,200,adaptive_sma
2025-05-09T20:04:04.685346,http://localhost:8081,41,200,adaptive_sma
2025-05-09T20:04:04.685456,http://localhost:8081,44,200,adaptive_sma
2025-05-09T20:04:04.685560,http://localhost:8081,43,200,adaptive_sma
2025-05-09T20:04:04.685660,http://localhost:8081,44,200,adaptive_sma
2025-05-09T20:04:04.685962,http://localhost:8081,41,200,adaptive_sma
2025-05-09T20:04:04.686059,http://localhost:8081,45,200,adaptive_sma
2025-05-09T20:04:04.686148,http://localhost:8081,40,200,adaptive_sma
2025-05-09T20:04:04.686456,http://localhost:8081,55,200,adaptive_sma
2025-05-09T20:04:04.686559,http://localhost:8081,40,200,adaptive_sma
2025-05-09T20:04:04.686658,http://localhost:8081,41,200,adaptive_sma
2025-05-09T20:04:04.686753,http://localhost:8081,40,200,adaptive_sma
2025-05-09T20:04:04.732861,http://localhost:8081,96,200,adaptive_sma
2025-05-09T20:04:04.733202,http://localhost:8081,93,200,adaptive_sma
2025-05-09T20:04:04.733319,http://localhost:8081,93,200,adaptive_sma
2025-05-09T20:04:04.733405,http://localhost:8081,93,200,adaptive_sma
2025-05-09T20:04:04.733489,http://localhost:8081,94,200,adaptive_sma
2025-05-09T20:04:04.733568,http://localhost:8081,92,200,adaptive_sma
2025-05-09T20:04:04.733649,http://localhost:8081,91,200,adaptive_sma
2025-05-09T20:04:04.733733,http://localhost:8081,94,200,adaptive_sma
2025-05-09T20:04:04.733808,http://localhost:8081,92,200,adaptive_sma
2025-05-09T20:04:04.733883,http://localhost:8081,94,200,adaptive_sma
2025-05-09T20:04:04.733965,http://localhost:8081,89,200,adaptive_sma
2025-05-09T20:04:04.734040,http://localhost:8081,90,200,adaptive_sma
```

Figure 1: Screenshot of an example CSV request log file (`proxy_log.csv`).

- **Latency (Avg, Stdev, Max):** Average, standard deviation, and maximum response times experienced by the client.
- **Total Requests:** Total number of requests completed during the test duration.

An example snippet from a ‘wrk‘ output:

```
Running 2m test @ http://localhost:9090
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   202.00ms  112.98ms 470.45ms   67.16%
    Req/Sec   124.27     34.15   272.00     63.19%
  59392 requests in 2.00m, 10.52MB read
Requests/sec:    494.77
Transfer/sec:     89.76KB
```

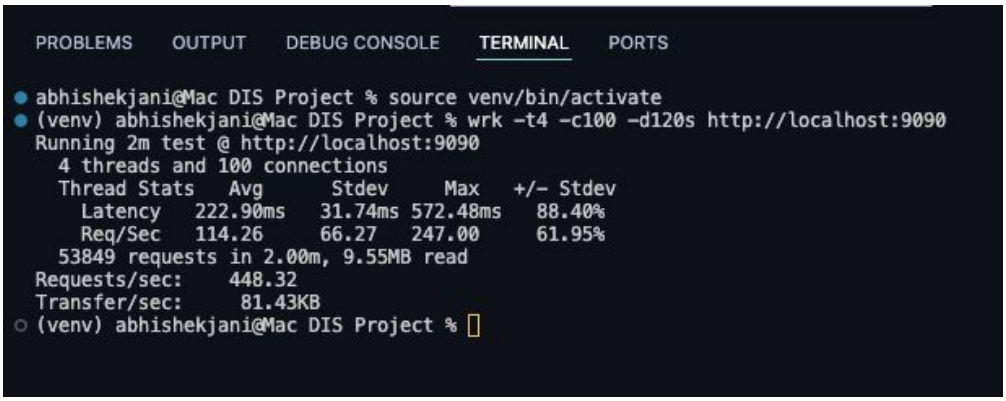


Figure 2: Screenshot of an example `wrk` benchmark output in the terminal.

11.4 Real-time Dashboard (`dashboard.py`, `dashboard.html`)

As an extension, a real-time dashboard was developed using Flask (Python web framework) and JavaScript (Plotly.js, Chart.js).

- `dashboard.py`: A Flask application that serves the HTML page and provides a `/data` API endpoint. This endpoint reads the live `proxy_log.csv` file, processes recent entries using ‘pandas‘ to calculate metrics like current average latency, request distribution per backend, and prepares data for charts.
- `dashboard.html`: An HTML page that fetches data from the `/data` endpoint periodically and updates several visualizations. Screenshots of the dashboard are shown in Figures 3 through 6.
  - A Plotly.js line chart showing the latency trend for each backend server.
  - A Chart.js bar chart showing the distribution of requests to each backend.
  - A table displaying the current SMA/EWMA values for each backend.
  - Current routing mode and overall average latency.

This dashboard allows for live observation of the proxy’s behavior and the effectiveness of the routing algorithms as a test is running.

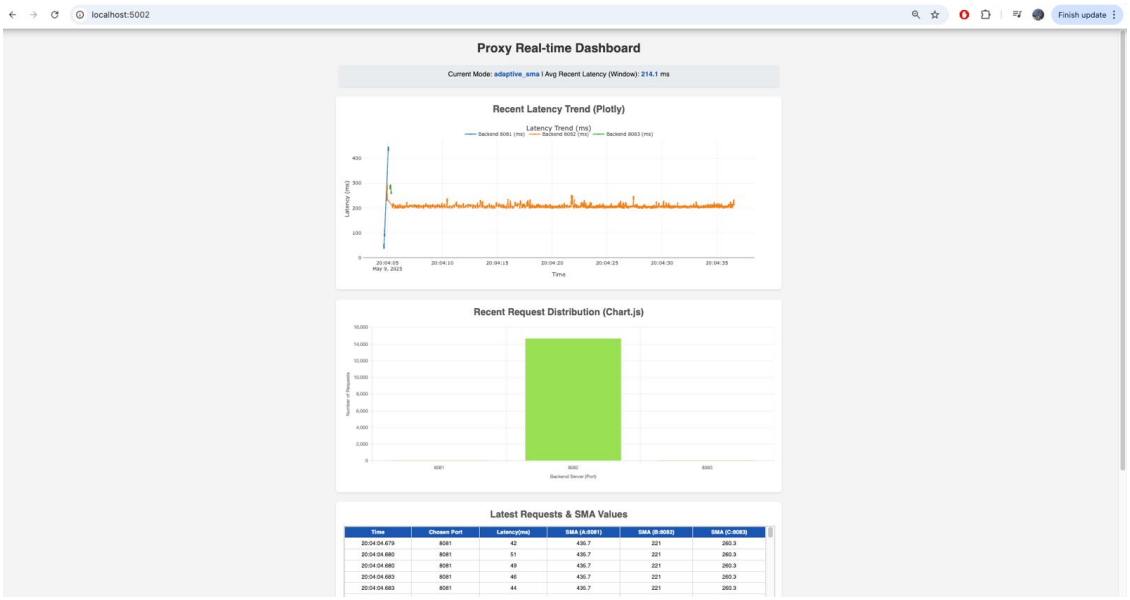


Figure 3: Main view of the Proxy Real-time Dashboard displaying latency trends, request distribution, and latest requests.

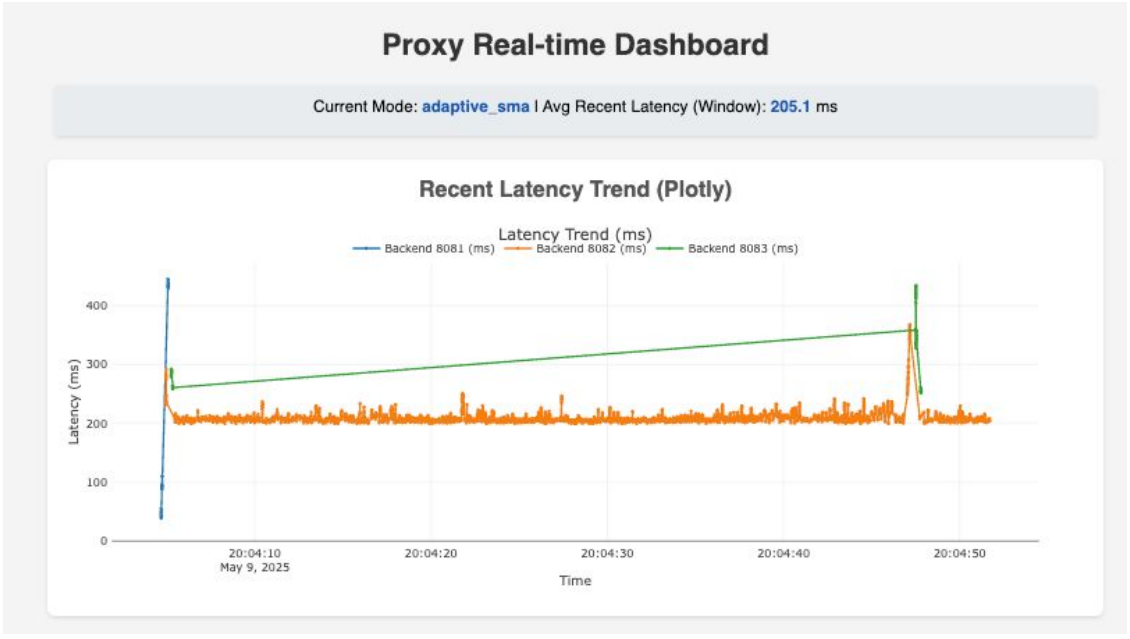
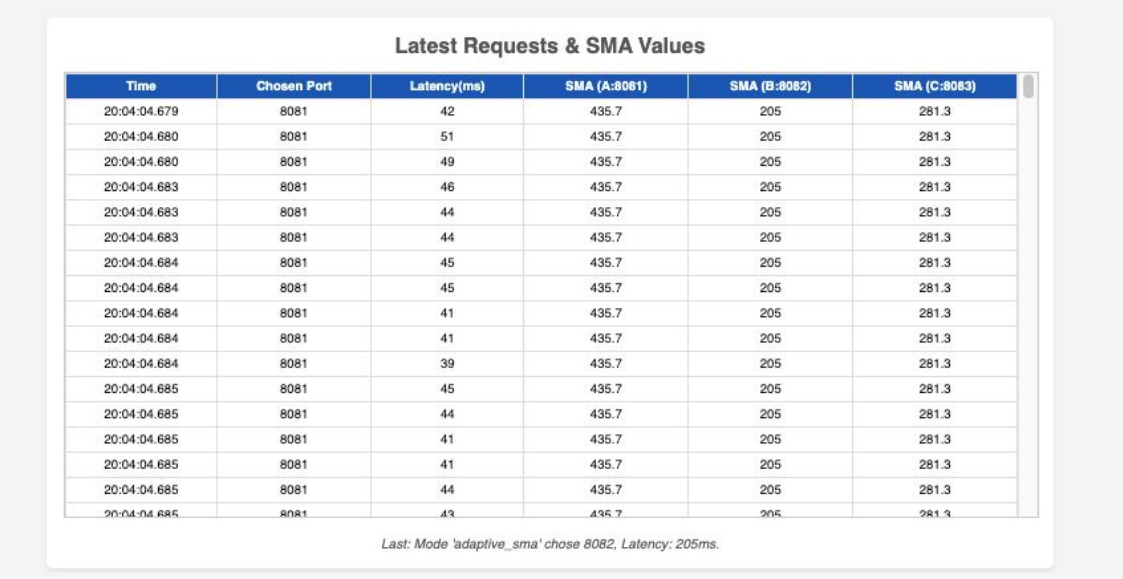


Figure 4: Dashboard: Close-up of the Recent Latency Trend (Plotly) for backend servers.

Figure 5: Dashboard: Close-up of the Recent Request Distribution (Chart.js) among backend servers.



Latest Requests & SMA Values

Time	Chosen Port	Latency(ms)	SMA (A:8081)	SMA (B:8082)	SMA (C:8083)
20:04:04.679	8081	42	435.7	205	281.3
20:04:04.680	8081	51	435.7	205	281.3
20:04:04.680	8081	49	435.7	205	281.3
20:04:04.683	8081	46	435.7	205	281.3
20:04:04.683	8081	44	435.7	205	281.3
20:04:04.683	8081	44	435.7	205	281.3
20:04:04.684	8081	45	435.7	205	281.3
20:04:04.684	8081	45	435.7	205	281.3
20:04:04.684	8081	41	435.7	205	281.3
20:04:04.684	8081	41	435.7	205	281.3
20:04:04.684	8081	39	435.7	205	281.3
20:04:04.685	8081	45	435.7	205	281.3
20:04:04.685	8081	44	435.7	205	281.3
20:04:04.685	8081	41	435.7	205	281.3
20:04:04.685	8081	41	435.7	205	281.3
20:04:04.685	8081	44	435.7	205	281.3
20:04:04.685	8081	43	435.7	205	281.3

Last: Mode 'adaptive_sma' chose 8082, Latency: 205ms.

Figure 6: Dashboard: Table displaying Latest Requests and calculated SMA values for each backend server.

12 Benchmark Methodology

To test the performance of our proxy server with different configurations, we used the ‘wrk‘ HTTP benchmarking tool. A consistent methodology was applied for all test runs to ensure fair comparisons.

12.1 `wrk` Command

The following ‘wrk‘ command template was used for all benchmarks:

```
wrk -t4 -c100 -d120s http://localhost:9090 > logs/<config_mode>_wrk.txt
```

Let's break down the parameters:

- `-t4` **(Threads):** ‘wrk‘ uses 4 threads to generate load. This means ‘wrk‘ can handle making requests and receiving responses in parallel, simulating multiple users more effectively.
- `-c100` **(Connections):** ‘wrk‘ maintains 100 concurrent HTTP connections to the proxy server. These are long-lived connections from ‘wrk‘'s perspective; ‘wrk‘ will try to send requests over these connections as fast as possible.
- `-d120s` **(Duration):** Each benchmark test runs for 120 seconds (2 minutes).
- `http://localhost:9090`**:** This is the URL of our proxy server.
- `> logs/<config_mode>_wrk.txt`**:** This part redirects the output summary from ‘wrk‘ into a text file specific to the test configuration (e.g., `round_robin_persistent_wrk.txt`).

12.2 Why these parameters?

- **120 seconds duration:** This duration was chosen for a few reasons:
  - It gives adaptive algorithms like EWMA and SMA enough time to "warm up" and for their averages to stabilize based on the observed backend performance. Short tests might not accurately reflect their steady-state behavior.
  - It allows the system to process a large number of requests (typically around 50,000 to 60,000 in our tests). A larger sample size leads to more statistically stable and reliable average metrics.
- **100 concurrent connections with 4 threads:** These settings provide a reasonably high load on the proxy server, allowing us to stress the system and reveal performance differences between configurations. The number of connections is high enough to ensure that the proxy is the bottleneck, not the client.

13 Step-by-Step Execution Guide

To run the experiments and gather data, the following steps were followed for each configuration (a combination of proxy type and routing algorithm):

1. **Start Backend Servers:** Launch the three backend servers, each in a separate terminal, as described in Section 8.

```
docker run -d -e SERVER_ID=A PORT=8081 python3 backend_server.py

```

Listing 2: Starting Backend Server A

(Similarly for Server B on port 8082 and Server C on port 8083).

2. **Clear Previous Log (Optional but Recommended):** Delete or rename the existing `proxy_log.csv` (or the specific CSV for the run if named differently by the proxy script) to ensure a clean log file for the current test. The provided proxy scripts initialize the log file with headers if it doesn't exist or is empty.

3. **Launch Proxy Server:** Start the desired proxy server variant (persistent or non-persistent) with the chosen routing algorithm. For example, to run the persistent proxy with EWMA routing, saving console output:

```
python3 persistent_proxy_server.py adaptive_ewma > logs/
    ewma_persistent_console.txt

```

Listing 3: Starting Persistent Proxy with EWMA

Replace `persistent_proxy_server.py` with `proxy_server_non_persistent.py` for non-persistent tests, and `adaptive_ewma` with `round-robin` or `adaptive_sma` as needed.

4. **Run `wrk` Benchmark:** In another terminal, execute the 'wrk' command (Section 12), ensuring the output is saved:

```
wrk -t4 -c100 -d120s http://localhost:9090 > logs/ewma_persistent_wrk
    .txt

```

Listing 4: Running wrk Benchmark

Match the log filename (`ewma_persistent_wrk.txt`) to the proxy configuration being tested.

5. **Monitor:** Observe the console output of the proxy server to see routing decisions in real-time. If using the dashboard (Section 11.4), open it in a web browser.

6. **Terminate Proxy & Collect Data:** Once 'wrk' finishes (after 120 seconds), stop the proxy server (usually with Ctrl+C in its terminal). The `proxy_log.csv` (or its specific variant like `ewma_persistent.csv` if the proxy names it that way) will contain the detailed request data for this run. The console output and 'wrk' output files will also be in the `logs/` directory.

This process was repeated for all 6 combinations:

- Persistent + Round-Robin
- Persistent + SMA
- Persistent + EWMA
- Non-Persistent + Round-Robin
- Non-Persistent + SMA
- Non-Persistent + EWMA

14 Data Analysis Pipeline

After running the benchmarks, the collected data (primarily the CSV logs and `wrk` output files) was analyzed to extract key performance metrics.

14.1 Metrics and Formulas

The following primary metrics were calculated from the `proxy_log.csv` files for each test run. These provide a detailed view of the proxy's performance from its own perspective.

- **Total Requests Processed:** The total number of entries in the CSV log for a given run.

$$\text{Total Requests} = \text{Count of rows in CSV}$$

- **Requests per Second (Throughput):** The average number of requests processed by the proxy per second over the 120-second test duration.

$$\text{Requests/sec} = \frac{\text{Total Requests}}{\text{Test Duration (120s)}}$$

- **Mean Latency:** The average latency of all requests routed by the proxy, based on the `latency_ms` column in the CSV.

$$\text{Mean Latency (ms)} = \frac{\sum \texttt{latency\_ms}}{\text{Total Requests}}$$

  This is calculated using `df.latency_ms.mean()` in pandas.

- **p95 Latency (95th Percentile Latency):** The latency value below which 95% of the requests fall. This is a crucial indicator of tail latency – the experience of the slowest 5% of requests. A lower p95 latency means more consistent performance.

$$\text{p95 Latency (ms)} = \text{95th percentile of } \texttt{latency\_ms} \text{ values}$$

  This is calculated using `df.latency_ms.quantile(0.95)` in pandas.

- **Backend Request Distribution:** The percentage of total requests routed to each of the three backend servers (A, B, C). This shows how the load was distributed by each algorithm.

$$\text{Distribution for Server X (\%)} = \frac{\text{Requests to Server X}}{\text{Total Requests}} \times 100$$

  Calculated using `df.backend_url.value_counts(normalize=True)` in pandas.

The `wrk` output files provided a client-side view of throughput and average latency, which served as a good cross-check for the proxy's own logs.

14.2 Tools

- **Pandas (Python library):** Used extensively to load the CSV log files into DataFrames, perform calculations for the metrics described above, and aggregate data.
- **Python:** Used for scripting the analysis with Pandas.
- **Matplotlib (Python library):** The original project plan included using Matplotlib for generating bar charts and other visualizations from the processed data. For this report, 'pgfplots' in LaTeX is used to create visualizations based on the data derived from the Pandas analysis.

The analysis involved processing each of the 6 CSV log files (one for each combination of persistence and routing mode) to generate the metrics, which are then presented in the Results section.

15 Results & Interpretation

This section presents the performance results from the benchmark tests. We analyze data for both persistent and non-persistent proxy configurations, each tested with Round-Robin (RR), Simple Moving Average (SMA), and Exponential Moving Average (EWMA) routing algorithms. All tests were run for 120 seconds.

15.1 Persistent Proxy Performance

In this mode, the proxy maintained open connections to the backend servers, reusing them for multiple requests.

15.1.1 Persistent Proxy - Round-Robin (RR)

- **Requests/sec:** 494.93
- **Mean Latency:** 201.57 ms
- **p95 Latency:** 416.0 ms

- **Backend Distribution:**
  - Server A (Port 8081): 33.33%
  - Server B (Port 8082): 33.33%
  - Server C (Port 8083): 33.33%

Interpretation: Round-Robin distributed load perfectly evenly. The mean latency is close to the mathematical average of the three backends (≈ 197.78 ms). The p95 latency is high, pulled up by Server A's 400ms spikes and Server C's 250ms responses, which RR cannot avoid.

15.1.2 Persistent Proxy - Adaptive SMA

- **Requests/sec:** 467.11
- **Mean Latency:** 213.61 ms
- **p95 Latency:** 238.0 ms
- **Backend Distribution:**
  - Server A (Port 8081): 22.83%
  - Server B (Port 8082): 75.48%
  - Server C (Port 8083): 1.69%

Interpretation: SMA significantly shifted traffic towards Server B (200ms constant) and away from Server C (slowest) and reduced load on Server A compared to RR. This dramatically improved p95 latency. The mean latency is slightly higher than RR, likely because Server B (200ms) handled the bulk of requests, whereas RR benefited more from Server A's very fast phases (10-20ms) for a third of its traffic. Throughput is slightly lower than RR.

15.1.3 Persistent Proxy - Adaptive EWMA

- **Requests/sec:** 468.70
- **Mean Latency:** 213.03 ms
- **p95 Latency:** 239.0 ms
- **Backend Distribution:**
  - Server A (Port 8081): 22.60%
  - Server B (Port 8082): 76.06%
  - Server C (Port 8083): 1.34%

Interpretation: EWMA's performance is very similar to SMA in persistent mode. It also heavily favored Server B, avoided Server C, and significantly reduced p95 latency compared to RR. Mean latency and throughput are comparable to SMA. Console logs confirmed EWMA choosing faster servers based on its calculated averages (e.g., `[EWMA] EWMAs: 8081=405.1ms, 8082=211.8ms, 8083=282.8ms -> chosen: http://localhost:8082`).

15.2 Non-Persistent Proxy Performance

In this mode, the proxy created a new connection to a backend server for every request.

15.2.1 Non-Persistent Proxy - Round-Robin (RR)

- **Requests/sec:** 490.61
- **Mean Latency:** 203.28 ms
- **p95 Latency:** 429.0 ms
- **Backend Distribution:**
  - Server A (Port 8081): 33.34%
  - Server B (Port 8082): 33.33%
  - Server C (Port 8083): 33.33%

Interpretation: Similar to persistent RR, load is evenly distributed. Mean and p95 latencies are slightly higher than persistent RR due to the overhead of new connection establishment for each request (TCP handshake). Throughput is slightly lower.

15.2.2 Non-Persistent Proxy - Adaptive SMA

- **Requests/sec:** 437.29
- **Mean Latency:** 228.08 ms
- **p95 Latency:** 416.0 ms
- **Backend Distribution:**
  - Server A (Port 8081): 34.62%
  - Server B (Port 8082): 63.19%
  - Server C (Port 8083): 2.19%

Interpretation: SMA still attempts to be smart, favoring Server B and avoiding C. However, the p95 latency is much higher than persistent SMA and only marginally better than non-persistent RR. The benefits of adaptive routing are significantly diminished by the high base cost of new connections. Mean latency is higher, and throughput is lower compared to persistent SMA and non-persistent RR.

15.2.3 Non-Persistent Proxy - Adaptive EWMA

- **Requests/sec:** 442.53
- **Mean Latency:** 225.30 ms
- **p95 Latency:** 416.0 ms
- **Backend Distribution:**
  - Server A (Port 8081): 34.32%
  - Server B (Port 8082): 63.92%
  - Server C (Port 8083): 1.76%

Interpretation: EWMA's performance mirrors SMA in non-persistent mode. It shows similar backend distribution, slightly higher throughput than SMA-NP, but suffers from high p95 and mean latencies, reinforcing that non-persistent connections are detrimental to achieving low and consistent latency, even with smart routing.

15.3 Comparative Analysis & Visualizations

15.3.1 Summary Tables

Table 1: Performance Summary: Persistent Proxy

Algorithm	Req/sec	Mean Latency (ms)	p95 Latency (ms)
Round-Robin	494.93	201.57	416.0
Adaptive SMA	467.11	213.61	238.0
Adaptive EWMA	468.70	213.03	239.0

Table 2: Performance Summary: Non-Persistent Proxy

Algorithm	Req/sec	Mean Latency (ms)	p95 Latency (ms)
Round-Robin	490.61	203.28	429.0
Adaptive SMA	437.29	228.08	416.0
Adaptive EWMA	442.53	225.30	416.0

Table 3: Backend Request Distribution for Adaptive Algorithms (%)

Proxy Type	Algorithm	Server A (8081)	Server B (8082)	Server C (8083)
Persistent	Adaptive SMA	22.83	75.48	1.69
Persistent	Adaptive EWMA	22.60	76.06	1.34
Non-Persistent	Adaptive SMA	34.62	63.19	2.19
Non-Persistent	Adaptive EWMA	34.32	63.92	1.76

15.3.2 Performance Graphs

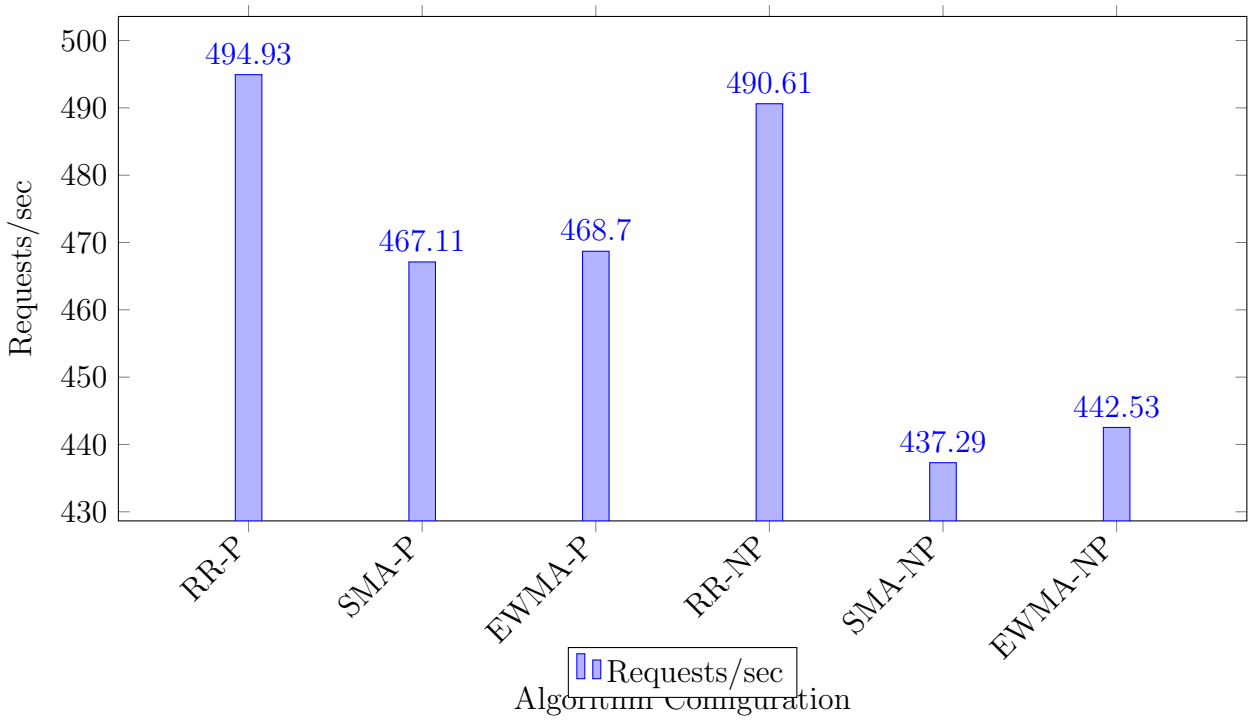


Figure 7: Requests per Second Comparison (P=Persistent, NP=Non-Persistent)

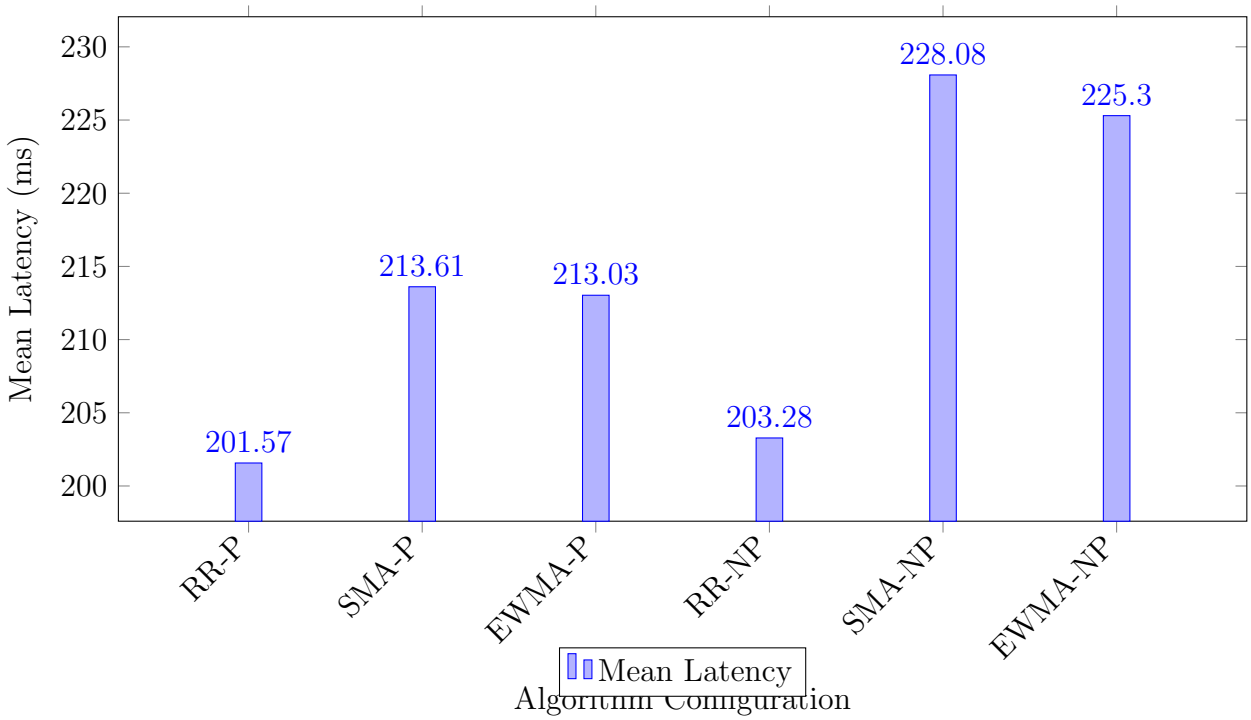


Figure 8: Mean Latency Comparison (P=Persistent, NP=Non-Persistent)

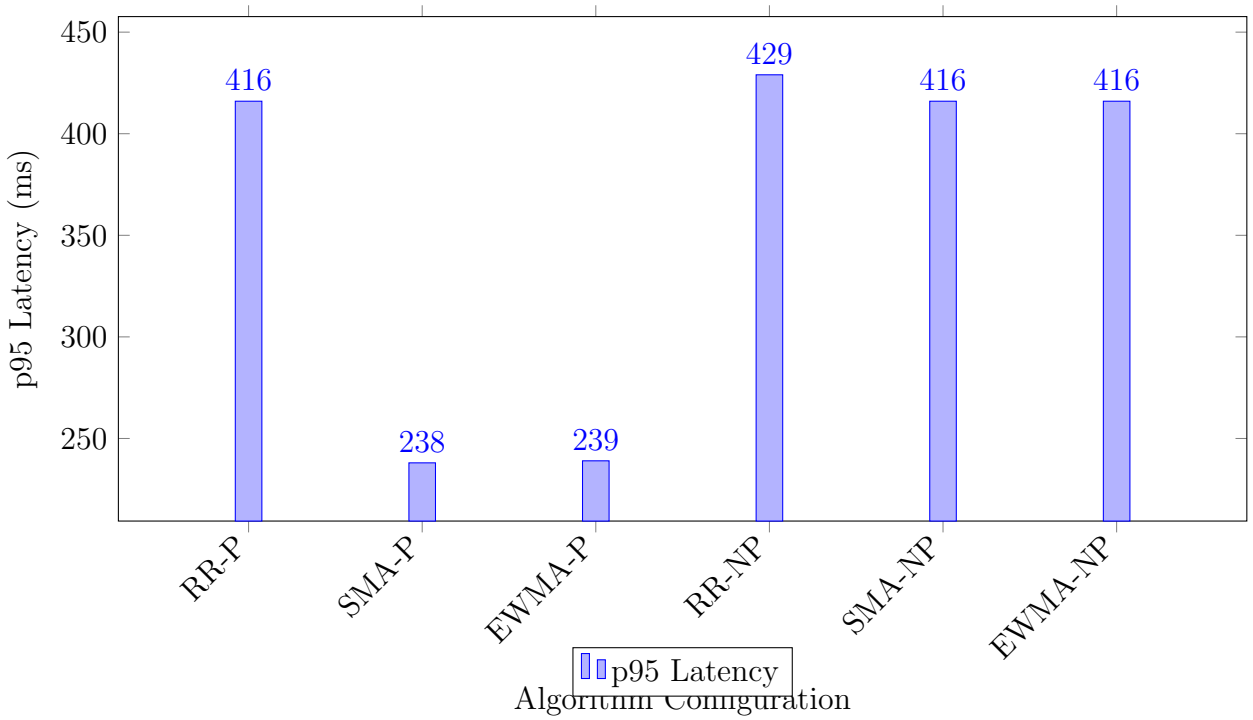


Figure 9: p95 Latency Comparison (P=Persistent, NP=Non-Persistent)

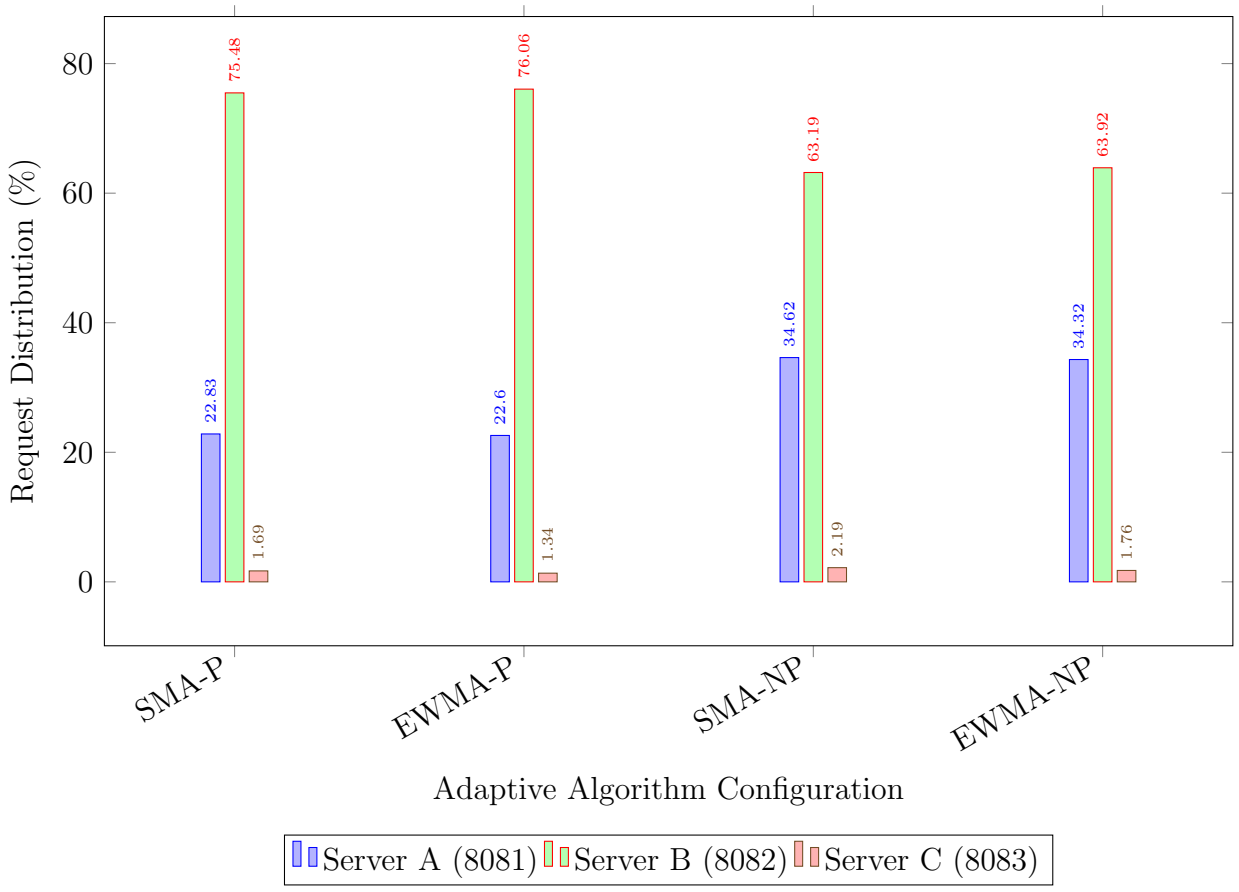


Figure 10: Backend Request Distribution for Adaptive Algorithms (P=Persistent, NP=Non-Persistent)

15.4 Key Takeaways & Interpretation from Results

1. **Persistence is Crucial for Performance:**
    - **Throughput:** Persistent connections consistently yielded slightly higher requests/second (see Figure 7). For adaptive algorithms, the gain was more noticeable (e.g., SMA: 467 req/s persistent vs. 437 req/s non-persistent). This is because eliminating the TCP handshake for each request saves time and resources.
    - **Mean Latency:** Persistent connections resulted in lower mean latencies (Figure 8). The improvement was around 1-2ms for Round-Robin but more significant for adaptive algorithms (12-14ms reduction). This directly translates to a faster experience for the average user.
    - **p95 Latency:** This is where persistence showed its most dramatic impact, especially for adaptive algorithms (Figure 9). For SMA/EWMA, p95 latency dropped from 416ms (non-persistent) to around 238-239ms (persistent) – a reduction of nearly 180ms! This means persistent connections make the service much more reliably fast for almost all users when smart routing is used. Non-persistent connection overhead seems to add too much noise and baseline delay for adaptive strategies to significantly improve tail latency.
2. **Adaptive Algorithms (SMA/EWMA) Drastically Reduce Tail Latency (especially with Persistence):**
    - Compared to Round-Robin, both SMA and EWMA significantly reduced the p95 latency when used with persistent connections (RR-P: 416ms vs. SMA-P: 238ms, EWMA-P: 239ms). This means fewer users experience very slow responses.
    - They achieved this by intelligently routing traffic away from the slow Server C and being cautious with Server A's high-latency phases, primarily favoring the consistent Server B (see Figure 10 and Table 3).
    - In non-persistent mode, the p95 improvement by adaptive algorithms over RR was minimal (429ms for RR-NP vs. 416ms for SMA/EWMA-NP).
3. **Round-Robin's Trade-off:**
    - Round-Robin consistently achieved the highest raw throughput (Requests/sec) in both persistent and non-persistent modes, albeit by a small margin. This is likely due to its simplicity, involving less computation per request than SMA or EWMA.
    - However, this higher throughput came at the cost of significantly worse p95 latency, making the user experience less consistent.
4. **SMA vs. EWMA Performance:**
    - With the current parameters ($\alpha = 0.2$ for EWMA, $N = 3$ for SMA), both adaptive algorithms performed almost identically across all metrics (throughput, mean latency, p95 latency, and backend distribution). This aligns with the initial project document's expectation.

5. **Mean Latency Anomaly with Adaptive Algorithms (Persistent):**

   - Interestingly, in persistent mode, the mean latency for SMA (213.61ms) and EWMA (213.03ms) was slightly higher than for Round-Robin (201.57ms).
   - This is because adaptive algorithms heavily routed traffic to Server B (200ms latency). While this strategy greatly improved p95 by avoiding Server A's 400ms spikes and Server C's 250ms responses, it meant fewer requests benefited from Server A's very fast 10-20ms phases. Round-Robin, by sending a third of its traffic to Server A unconditionally, benefited from these fast phases for its mean calculation, despite also being penalized by the slow phases for its p95. This highlights that optimizing for mean latency and p95 latency can sometimes involve different strategies. For most user-facing systems, a predictable (low p95) experience is often preferred over a slightly lower mean that includes very slow outliers.

In conclusion, the experiments clearly demonstrate that for achieving both high throughput and, more importantly, low and consistent response times (good p95 latency), a combination of persistent connections and an adaptive routing algorithm (SMA or EWMA) is the most effective strategy among those tested.

16 Common Problems & Fixes

During the development and execution of a project like this, we faced several challenges . Here are some potential solutions:

- **Problem: CSV latency values are -1ms or unexpectedly low/high.**
  - *Likely Cause:* The error handling block in the proxy might be writing a default error value (like -1) for timed-out or failed backend requests. Unexpectedly low values could indicate issues with time measurement. High values could be genuine or indicate severe backend/network issues.
  - *Fix:* Confirm all backend servers are running and accessible from the proxy. Check for errors in the proxy's console output. Increase timeout values in the proxy if backend servers are genuinely slow but operational. Double-check latency calculation logic (start/end time).
- **Problem: Adaptive algorithm (EWMA/SMA) seems stuck on one backend or doesn't adapt.**
  - *Likely Cause:* The warm-up or probing logic for unmeasured backends might be missing or flawed. The update logic for EWMA/SMA values after a request might not be working correctly. Latency values might not be correctly reported or recorded.

- *Fix:* Review the sections of code responsible for initializing and updating EWMA/SMA values. Check the console logs for probing messages and the calculated averages to see if they make sense. Ensure latency values are positive and correctly captured.

- **Problem: "Address already in use" error when starting servers.**
  - *Likely Cause:* Another process is already using one of the required ports (8081-8083 for backends, 9090 for proxy, or 5002 for dashboard). A previous instance of one of the servers might not have shut down cleanly.
  - *Fix:* Use system utilities to find and terminate the process occupying the port. On Linux/macOS: `lsof -i :<port_number>` or `netstat -tulnp | grep <port_number>`. On Windows: `netstat -ano | findstr <port_number>` then use Task Manager to kill the process by PID.

- **Problem: `wrk` shows very few requests or many errors (socket errors, timeouts).**
  - *Likely Cause:* The proxy server might be crashing or not handling requests correctly. Backend servers might not be running or accessible. System resource limits (like open file descriptors) might be hit if connections are not being closed properly in non-persistent mode over a very long test (less likely with `aiohttp`'s proper session handling).
  - *Fix:* Check proxy console output for tracebacks or error messages. Verify backend servers are running. Test accessing the proxy URL with a simple tool like 'curl' or a web browser. For persistent proxy, ensure sessions are cleaned up on shutdown.

- **Problem: Inconsistent results between test runs.**
  - *Likely Cause:* Other processes on the machine consuming CPU/network resources. Not clearing previous log files (if the script appends and averages over old data). Short test durations that don't allow the system to reach a steady state.
  - *Fix:* Minimize other system activity during benchmarks. Ensure a clean state for logs before each run. Use longer test durations (like the 120s chosen).