Adaptive Reverse-Proxy Load Balancer

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

1. Technical Goal

This project proposes the development and performance study of an HTTP reverse proxy load balancer, built using Python. The core idea is to create a system, similar in function to NGINX, that intelligently distributes incoming web traffic to three custom-built backend servers. We will explore and compare two primary connection strategies: non-persistent (new connection per request) and persistent (reusing connections). Traffic routing will be managed by two distinct policies: the basic Round-Robin (RR) method and an adaptive Simple Moving Average (SMA) algorithm, which aims to direct traffic based on real-time server performance. To test these approaches, backend servers will simulate varied response times. The system's performance under controlled load (generated by 'wrk') will be benchmarked, focusing on metrics like throughput and latency. Backend servers will be containerized using Docker, and a Flask/Chart.js dashboard will provide live monitoring. This project aims to demonstrate the impact of connection management and adaptive routing on load balancer efficiency and provide practical insights into how different load balancing strategies affect system efficiency and responsiveness.

2. Prior Work

- **NGINX & Envoy** provide industry-grade reverse proxies with advanced load-balancing (least-connections, health checks) and connection pooling.
- **Feedback-Driven Load Balancing** research explores reactive adjustments based on latency/throughput metrics.
- **Adaptive ML-Based Routing** applies predictive models to route traffic in datacenters.

Gap: Existing work focuses on large-scale deployments, complex heuristics, or black-box systems. There is less guidance on lightweight, extensible educational frameworks that expose per-request telemetry and allow hands-on tuning of simple adaptive algorithms in Python.

3. Project Contribution and Novelty

While production-grade load balancers like NGINX are highly optimized and feature-rich, this project's contribution lies in the hands-on implementation and direct comparative analysis of specific, clearly defined connection strategies and adaptive routing algorithms within a controlled, understandable Python environment. By building the components from the ground up (proxy, distinct backend server behaviors, live dashboard), the project provides a transparent platform for:

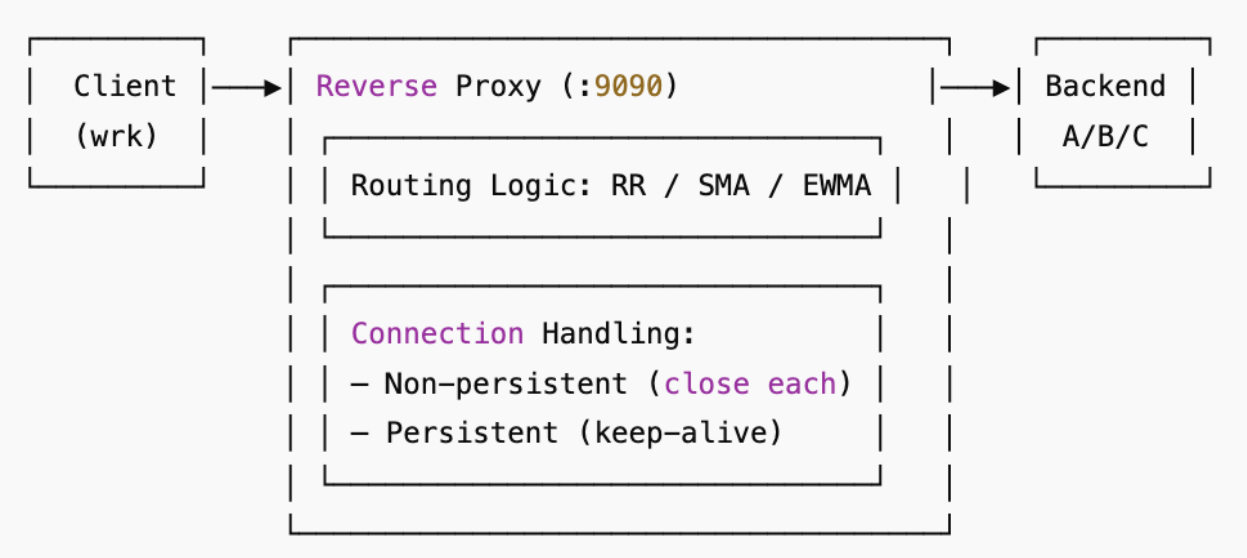
i. Quantitatively measuring the precise overhead of non-persistent vs. persistent connections in a Python/aiohttp context.

ii. Directly observing and evaluating the adaptive behavior of the SMA algorithm against varied, emulated backend latencies, including its ”cold start” behavior.
iii. Offering an integrated, real-time visualization dashboard (Flask/Chart.js) tied directly to the custom proxy’s internal state and logs, which serves as a learning and demonstration tool not typically part of off-the-shelf load balancer usage for educational purposes.

The focus is on a pedagogical exploration and empirical performance comparison of these core mechanisms, rather than attempting to replicate the full feature set or raw performance of established solutions. The novelty is in the integrated approach of building, testing, and visualizing these specific aspects for educational insight.

4. Technical Approach

4.1 High-Level Diagram



4.2 Key Components & Libraries

- **aiohttp**: Asynchronous HTTP server & client.
- **asyncio**: Event loop concurrency.
- **collections.deque**, **statistics**: For SMA/EWMA state.
- **wrk**: Load-generation (4 threads, 100 connections, 120 s).
- **Docker**: Used for Containerizing

4.3 Implementation Outline

1. **Backend Simulation**

- Three instances of backend_server.py, each with distinct LATENCY_CONFIG.

2. **Proxy Variants**
   - **Non-Persistent**: Per-request TCPConnector(force_close=True), Connection: close.
   - **Persistent**: Shared ClientSession with unlimited pool (limit=0), HTTP keep-alive.
3. **Routing Algorithms**
   - **RR**: next(itertools.cycle(backends)).
   - **SMA**: mean(last 3 latencies); treat ∞ until 3 samples.
   - **EWMA**: ewma_t = α·latency + (1−α)·ewma_{t−1}; initial warm-up probes.
4. **Logging**
   - Append to proxy_log.csv

5. Evaluation Plan

The project will be evaluated based on the following empirical metrics:

- *Requests per Second (Throughput):* Total requests processed by the system divided by the test duration.
- *Mean Latency*: Average response time for requests as measured by the proxy.
- *p95 Latency* (95th Percentile Latency): The latency value at which 95% of requests are completed, indicating tail-end performance.
- *Backend Request Distribution*: Percentage of requests routed to each backend server, to assess the behavior of adaptive algorithms.

Evaluation Methods: Evaluation will be performed by running ‘wrk‘ benchmarks against each defined configuration (proxy connection type + routing algorithm). The collected CSV logs will be processed to calculate and compare the defined metrics.

6. Foreseen Risks and Mitigation Strategies

The project faces several potential risks that could impact its progress or outcomes. High variability in measurements and environmental differences may threaten result consistency, which can be mitigated through repeated runs, extended durations, and use of a dedicated benchmarking machine. Resource exhaustion under high concurrency poses reliability concerns, addressed by bounding the connection pool and monitoring usage. Bugs or deadlocks in asynchronous logic using aiohttp demand thorough testing, incremental implementation, and detailed logging. Finally, the fixed academic timeline introduces time constraints, which can be mitigated by following a structured schedule, breaking down tasks, and addressing bugs early through regular progress checks.