Design and Architecture of the Go Load Balancer

Based on repository `alexovidiupopa/load-balancer`

November 23, 2025

1 Introduction

This document provides a technical overview of the load-balancer implementation in Go (from alexovidiupopa/load-balancer). We explain the core concepts and design decisions: load balancing, rate limiting, circuit breaking, and observability via Prometheus.

The load balancer listens on a frontend port (e.g. 8080) and forwards requests to a set of backend servers. Key features are:

- Request timeouts for resilience.
- Rate limiting (token bucket) to control load.
- Circuit breaker per backend to avoid overloading failing servers.
- Prometheus instrumentation for observability.

2 Load Balancing

2.1 What is Load Balancing

Load balancing is the technique of distributing incoming network or application traffic across multiple servers (backends) to:

- Improve availability (if one backend fails, others can serve).
- Scale throughput (parallelize serving).
- Reduce latency and avoid overloading a single node.

In the Go implementation, the `LoadBalancer` struct encapsulates the list of backends, health logic (via circuit breaker), and the forwarding behavior.

2.2 Load Balancing Strategies

While the sample code uses a simple round-robin-like mechanism (or sequential selection), in production systems multiple strategies exist:

- **Round Robin**: Each request is forwarded to the next server in a cyclic order.
- **Least Connections**: Send the request to the server with the fewest active connections.
- **Weighted Round Robin / Least Connections**: Assign weights to servers (e.g. more powerful machines get more traffic).
- **Consistent Hashing**: Compute a hash of some request attribute (e.g. client IP or session ID) to map to a backend, providing sticky affinity and minimizing disruption on backend changes.
- **Dynamic / Adaptive Algorithms**: Use real-time metrics (e.g. CPU, latency, error rate) to adjust routing decisions, possibly via feedback or machine learning.

3 Rate Limiting

3.1 Why Rate Limiting Matters

Rate limiting protects both the load balancer and backend servers from overload situations, preventing sudden surges of requests ("burst storms") from degrading system performance. It helps enforce fair use, avoid denial of service, and maintain predictability.

3.2 Token Bucket Algorithm

In the repository, rate limiting is implemented with a **token bucket** algorithm (in `rate_limiter.go`). The core idea:

- A "bucket" accumulates tokens at a fixed rate (e.g. r tokens per second), up to a maximum capacity C.
- Each incoming request consumes one (or more) tokens.
- If the bucket has no tokens, the request is rejected (or delayed) until tokens are available.

This allows occasional bursts (if tokens have built up) but enforces a long-term average rate.

3.3 Integration into the Load Balancer

The load balancer wraps request handling via the `RateLimiter`, checking before forwarding a request. If rate limiting rejects or delays a request, the balancer can respond with an HTTP error or back-off, protecting backend servers from being overwhelmed.

4 Circuit Breaker

4.1 Purpose of Circuit Breaking

A circuit breaker is a fault tolerance pattern designed to prevent repeated attempts to call a failing or unresponsive service. Rather than continuously routing requests to an unhealthy backend, the circuit breaker:

- Monitors failures (e.g. HTTP errors, timeouts).
- Opens the circuit when failure rate or count exceeds a threshold.
- While open, prevents further requests to the failing backend.
- After a cooldown period, transitions to *half-open* to test whether the backend has recovered.
- Closes the circuit when successful requests resume, or re-opens if failures persist.

4.2 Implementation in the Go Load Balancer

In `circuit_breaker.go`, each backend has a `CircuitBreaker` struct with states:

- **Closed**: normal operation.
- **Open**: backend is considered unhealthy; no forwarding.
- **Half-Open**: a limited number of test requests are allowed to probe recovery.

Transitions are governed by configurable thresholds (failure counts, timeout durations, recovery window). This prevents the load balancer from continuously sending requests to failing backends, improving overall system stability.

5 Putting It All Together: Request Flow

Here is a high-level summary of how the system handles a request:

1. Client sends HTTP request to load balancer.
2. Load balancer's rate limiter checks token availability:
   - If token available → proceed.
   - Else → reject or throttle the request.
3. Load balancer selects a backend using its balancing strategy.
4. Circuit breaker for the chosen backend is consulted:
   - If *open* → skip this backend, select another or return error.
   - If *half-open* → allow a limited test request.
   - If *closed* → forward the request.
5. Forward the request to the backend, using configured HTTP timeout parameters.
6. Receive response (or timeout/failure), update circuit breaker state (increment failure count if needed).
7. Record metrics (request count, latency, errors) via Prometheus instrumentation.
8. Return the response (or error) to the client.

6 Conclusion

This Go load-balancer is a compact but powerful demonstration of several fundamental resilience and scalability patterns:

- **Load balancing** distributes traffic to scale and improve availability.

- **Rate limiting** guards against overloads.
- **Circuit breaking** protects against failing backends.
- **Prometheus monitoring** provides visibility and alerting, enabling you to operate the system reliably.

These patterns are widely used in production systems. By combining them, this project builds a robust foundation that can be extended and hardened into a production-grade proxy or edge component.