Performance Analysis Report: Go Load Balancer

Sikander

June 18, 2025

Abstract

This report analyzes the performance of a Go-based load balancer designed to distribute requests across multiple FastAPI backends. An isolated performance test, bypassing backends, achieved 29,062 requests per second (req/s) with an average latency of 15.61ms. However, when proxying to backends, throughput dropped significantly to 500 req/s with one backend and 360 req/s with three. This report identifies bottlenecks, resource limitations, and logging overhead as primary causes of the performance degradation, providing detailed recommendations to optimize the system.

1 Introduction

The Go-based load balancer is implemented to distribute HTTP requests across multiple FastAPI backends using a round-robin algorithm, as specified in the configuration file (config.json). Initial tests showed a single FastAPI server handling 900 req/s. Introducing the load balancer reduced throughput to 500 req/s with one backend and 360 req/s with three backends, contrary to expectations of improved performance. An isolated test, bypassing backends, was conducted to measure the load balancer's standalone capacity, yielding 29,062 req/s. This report examines these results, identifies bottlenecks, and proposes solutions.

2 Test Configuration and Results

2.1 Isolated Test Setup

To isolate the load balancer's performance, the Balance function in balancer/balancer.go was modified to return a direct 200 OK response, bypassing backend proxying:

```go
func (lb *LoadBalancer) Balance(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
}
```

The test was conducted using wrk with the following parameters:

- Threads: 4
- Connections: 500
- Duration: 10 seconds
- Target: http://localhost:8080

2.2 Results

The wrk output is summarized in Table 1:

Key observations:

Table 1: Load Balancer Isolated Test Results

Metric	Value
Total Requests	293,274
Duration	10.09 seconds
Requests per Second	29,062.24
Average Latency	15.61ms
Latency StdDev	16.10ms
Maximum Latency	338.97ms
Data Transferred	20.98MB (2.08MB/s)
Socket Errors	Connect: 0, Read: 0, Write: 243, Timeout: 0

- Throughput: The load balancer handled 29,062 req/s, indicating strong standalone performance.
- Latency: Average latency was 15.61ms, but a high standard deviation (16.10ms) and maximum latency (338.97ms) suggest variability.
- Errors: 243 write errors indicate potential overload with 500 connections.

3 Performance Analysis

3.1 Comparison with Backend Tests

Previous tests showed:

- Single FastAPI Server: 900 req/s
- Load Balancer + 1 Backend: 500 req/s
- Load Balancer + 3 Backends: 360 req/s

The isolated test's 29,062 req/s confirms the load balancer's capacity is not the limiting factor. The significant drop when proxying suggests bottlenecks in the proxying process or backend interactions.

3.2 Identified Bottlenecks

3.2.1 Synchronous Logging

The load balancer code includes log.Printf statements in the request path, such as in the Director function of the reverse proxy:

```
log.Printf("Proxying request: %s %s Headers: %+v", req.Method,
    req.URL.String(), req.Header)
```

Synchronous logging is I/O-bound and can significantly slow down request processing under high load, contributing to the drop from 900 req/s to 500 req/s.

3.2.2 Proxying Overhead

The httputil.ReverseProxy implementation incurs overhead from:

- Establishing connections to backends.
- Handling request and response transformations.

- Health checks every 100 seconds, which may not detect backend issues promptly.

This overhead likely exacerbates the performance drop when adding more backends.

3.2.3 Resource Contention

Running the load balancer and three backends on a single machine leads to competition for CPU, memory, and network resources. The isolated test's high throughput (29,062 req/s) pushed system resources, as evidenced by 243 write errors and high maximum latency (338.97ms).

3.3 Resource Limitations

The test was conducted on a single machine (DESKTOP-2N24STS), with unspecified hardware. Key limitations include:

- CPU: With 4 threads and 500 connections, each thread handled 125 connections. High CPU usage could cause latency spikes.
- Memory: Frequent logging and connection management may strain memory.
- Network: Although on localhost, high connection rates could saturate internal network buffers, contributing to write errors.

4 Recommendations

4.1 Optimize Logging

Remove or minimize synchronous logging in the request path. Modify the Director function to log only errors or use an asynchronous logging library like zap. Example:

```go
proxy.Director = func(req *http.Request) {
    req.URL.Scheme = u.Scheme
    req.URL.Host = u.Host
    req.Host = u.Host
    req.URL.Path = path.Clean(req.URL.Path)
    if req.URL.Path == "" || req.URL.Path == "." {
        req.URL.Path = "/"
    }
    // Log only errors asynchronously
}
```

4.2 Enhance Proxy Efficiency

- Enable Keep-Alive: Ensure the http.Transport reuses connections:

```go
    proxy.Transport = &http.Transport{
        DialContext:           (&net.Dialer{Timeout: 5 * time.Second}).DialContext,
        ResponseHeaderTimeout: 30 * time.Second,
        MaxIdleConns:          100,
        IdleConnTimeout:       90 * time.Second,
    }
```

- Faster Health Checks: Reduce the health check interval (e.g., to 10 seconds) in health/healthcheck.go.
- Load Balancing Algorithm: Consider switching to least_conn if backends have varying performance.

4.3 Mitigate Resource Contention

- Monitor Resources: Use htop or top during tests to identify CPU or memory bottlenecks.
- Increase Threads: Adjust wrk to use more threads (e.g., 8) or tune GOMAXPROCS in Go.
- Distribute Components: Run backends on separate machines to reduce contention.

4.4 Further Testing

- Backend Isolation: Test each FastAPI backend individually to confirm their capacity (e.g., 900 req/s).
- Reduced Concurrency: Rerun the isolated test with fewer connections (e.g., 200) to check if write errors disappear.
- Alternative Load Balancers: Compare performance with Nginx or HAProxy to validate the Go implementation.

5 Conclusion

The isolated test demonstrated that the Go load balancer can handle 29,062 req/s with an average latency of 15.61ms, confirming its high standalone capacity. However, synchronous logging, proxying overhead, and resource contention caused significant performance degradation when proxying to backends (500 req/s with one backend, 360 req/s with three). By optimizing logging, enhancing proxy efficiency, and mitigating resource contention, the system can likely achieve or exceed the original 900 req/s. Implementing these recommendations and conducting further tests will ensure a robust and scalable load balancing solution.