My Final Project Evaluation

Concurrent Load-Balancing Reverse Proxy with Health Monitoring

Author: Tarik Ouabrk | Language: Go | Date: March 2026

Overall Score: 97 / 100 — Grade: A

1. Score Summary

Criteria	Max	Score	Notes
Proxy Functionality	30	**29**	Excellent — forwarding, 503 handling, failover all correct.
Concurrency & Safety	25	**24**	RWMutex + atomic used correctly. Minor: SetBackendStatus acquires pool write-lock then calls b.SetAlive() which acquires backend's own mux — nested locks, but no deadlock risk here.
Background Health Check	15	**15**	Periodic ticker, UP/DOWN log transitions, 2s timeout per check — fully correct.
Load Balancing Logic	10	**10**	Both Round-Robin (atomic counter) and Least-Connections implemented and tested.
Context & Timeouts	10	**9**	Per-attempt context timeout is correct. Client cancellation is propagated via r.Context(). Minor: client disconnect mid-stream after headers are written is not explicitly handled.
Code Organization & Docs	10	**10**	Clean packages, comprehensive README, bilingual documentation, full test suites.
TOTAL	**100**	**97**	**Outstanding work — production-quality implementation.**

2. What You Did Exceptionally Well

2.1 Architecture & Package Design

The project is split into tightly-scoped packages (pool, proxy, health, admin), each with a single clear responsibility. This separation makes the codebase easy to navigate, test, and extend — a design quality rarely seen in a course final project.

2.2 Interface-Driven Design

Defining LoadBalancer as an interface and having ServerPool implement it is exactly the right approach. It decouples the health checker, admin API, and proxy handler from the concrete pool implementation, making unit testing trivial (you can inject any mock that satisfies the interface).

2.3 Thread Safety

Your use of sync.RWMutex on the ServerPool (read-lock for reads, write-lock for mutations) combined with sync/atomic for the round-robin counter and CurrentConns counter is textbook-correct. The race-

condition test (TestConcurrentAccess_NoRace) with 50 concurrent readers and 10 concurrent writers demonstrates genuine understanding, not just mechanical compliance.

2.4 Health Checker

The health checker (health/checker.go) is clean and correct: a time.Ticker drives a goroutine, each check uses a context with a 2-second timeout, and state transitions are logged precisely (UP→DOWN, DOWN→UP only, not on every tick). Using CheckBackend as a public, side-effect-free function makes it independently testable — a nice touch.

2.5 Proxy Handler with Retry & Failover

The attemptBackend helper correctly scopes each attempt's context.WithTimeout to a single function call, so defer cancel() fires once per attempt rather than at the end of the entire request. This prevents goroutine/timer leaks on multi-attempt requests — a subtle but important correctness detail that most developers miss.

2.6 Test Coverage

The test suite is comprehensive across all three packages:

- pool: round-robin cycling, dead-backend skipping, all-dead returns nil, empty pool, least-connections selection, SetBackendStatus, RemoveBackend, and a race-detector test.
- health: healthy/unhealthy/unreachable CheckBackend cases, plus integration tests for Start() that verify UP→DOWN and DOWN→UP transitions within a real time window.
- proxy: forwarding, empty pool 503, all-dead 503, failover to second backend, connection counter reset, and backend timeout producing 503.

Running go test -race ./... on this suite would pass cleanly — that's a high bar.

2.7 Graceful Shutdown

main.go correctly listens for SIGINT/SIGTERM and calls server.Shutdown(ctx) with a 10-second drain window. In-flight requests are allowed to complete before the process exits. This is production-grade behavior.

2.8 Admin API

The Admin API correctly validates input (missing host, duplicate detection, non-existent removal), uses appropriate HTTP status codes (201 Created, 204 No Content, 404, 405, 409 Conflict), and adds new backends in an unverified state so that the health checker — not the API call — is the source of truth for liveness.

3. Minor Issues & Suggestions

3.1 Nested Lock Pattern in SetBackendStatus (Low Risk)

SetBackendStatus acquires the pool's write-lock and then calls b.SetAlive(), which acquires the backend's own RWMutex. This nested locking is currently safe because no code path holds both locks in the opposite order. However, as the codebase grows, this pattern can lead to subtle deadlocks. A safer approach is to inline the alive field update without calling the helper inside the pool lock, or to document the lock ordering explicitly.

3.2 Round-Robin Counter Overflow (Theoretical)

The atomic uint64 counter Current will overflow after 2^64 increments. In practice this is harmless (it wraps back to 0), but the modulo arithmetic still works correctly. This is a known, accepted pattern in Go load balancers — no action needed, just worth being aware of.

3.3 Response Buffering in the Proxy

The proxy uses httptest.ResponseRecorder to buffer the entire backend response before writing it to the real client. This means large responses (e.g., file downloads) are fully buffered in memory before the client receives a single byte. For a production proxy you would stream the response directly. For a course project this is a perfectly acceptable trade-off and enables the clean failover logic.

3.4 Config Validation

The config loader validates the strategy field but not port ranges (e.g., port 0 or port > 65535) or the health_check_frequency being 0 before the default is applied. Adding a dedicated validateConfig() function would make startup errors clearer.

3.5 Admin API Authentication

The admin API on port 8081 has no authentication. In a real deployment, a malicious actor on the same network could add or remove backends. Consider adding a simple bearer token check or binding the admin server to 127.0.0.1 only.

3.6 Backend Struct JSON Tags

The Backend struct in pool/server_pool.go does not have JSON tags, while the project spec shows them. The admin API serializes BackendStatus (a separate struct with JSON tags) rather than Backend directly, so this has no runtime impact — but aligning the struct with the spec is good practice.

4. Notable Design Decisions Worth Highlighting

- Startup health validation: Checking all backends before starting the proxy and warning when zero are healthy is a production-quality initialization pattern.
- GetBackends() returns a snapshot copy (append([]*Backend(nil), ...)) rather than a reference to the internal slice. This prevents external code from mutating the pool's slice while only holding a read lock.
- The transportWrapper records transport-level failures without mutating any shared state, keeping each request's error path fully isolated.
- README quality: The bilingual documentation, strategy comparison table, and concrete PowerShell/bash test commands for both strategies go well beyond what is typically expected in a course submission.

5. Suggested Next Steps (Optional Enhancements)

- Sticky Sessions: As suggested in the spec, implement IP-hash or cookie-based session affinity.
- Weighted Backends: Add a Weight field to Backend and adjust the least-connections selector to prefer backends with higher weight at equal connection counts.
- Metrics endpoint: Expose a /metrics endpoint (Prometheus-compatible) reporting request counts, latency percentiles, and error rates per backend.
- TLS/HTTPS: Allow the proxy to serve over SSL using crypto/tls with automatic certificate management via golang.org/x/crypto/acme/autocert.
- Response streaming: Replace httptest.ResponseRecorder with a direct pipe to improve performance for large responses.