Introduction to Go - Final Project

Concurrent Load-Balancing Reverse Proxy with Health Monitoring

---

Introduction

Welcome to your final project! This project is designed to consolidate your understanding of Go networking capabilities, concurrency primitives, and structural design. You will build a **Concurrent Load-Balancing Reverse Proxy**. This system will sit in front of multiple backend servers, distributing incoming traffic based on specific strategies while monitoring the health of those servers in the background.

This project will challenge you to:

- Implement complex interfaces and structural composition.
- Utilize the **net/http** package.
- Manage shared state across concurrent requests using synchronization primitives.
- Implement a **background health checker** using goroutines and tickers.
- Handle graceful shutdowns and request timeouts using the context package.

Project Overview

You will create a reverse proxy server that acts as a gateway for multiple backend services. The proxy will receive incoming HTTP requests and forward them to a pool of healthy backend servers.

The system will consist of:

1. **The Proxy Core:** Intercepts requests and forwards them using a Round-Robin or Least-Connections strategy.
2. **Health Checker:** A background service that periodically pings backend servers to ensure they are online.
3. **Admin API:** A separate set of endpoints to dynamically add, remove, or check the status of backend servers.

Project Objectives

By completing this project, you will:

- Use **composition** to build a flexible proxy handler.
- Implement **Round-Robin load balancing** logic.
- Use **sync.RWMutex** (and maybe also **sync/atomic**) to ensure thread-safe operations on the server pool.
- Leverage **httputil.ReverseProxy** for core forwarding logic.
- Implement a **periodic background job** to automate health monitoring.
- Master **Context propagation** to ensure backend requests are canceled if the client disconnects.

Project Requirements

1. Design Data Models with Nested Structs

Define the models for your backends and the proxy configuration.

Backend Struct

```
type Backend struct {
    URL          *url.URL      `json:"url"`
    Alive        bool          `json:"alive"`
    CurrentConns int64         `json:"current_connections"`
    mux          sync.RWMutex
}
```

ServerPool Struct

```
type ServerPool struct {
    Backends []*Backend `json:"backends"`
    Current  uint64     `json:"current"` // Used for Round-Robin
}
```

ProxyConfig Struct

```go
type ProxyConfig struct {
    Port            int           `json:"port"`
    Strategy        string        `json:"strategy"` // e.g., "round-robin" or "least-conn"
    HealthCheckFreq time.Duration `json:"health_check_frequency"`
}
```

2. Define Interfaces for Load Balancing

Create interfaces to abstract how the next server is selected.

LoadBalancer Interface

```go
type LoadBalancer interface {
    GetNextValidPeer() *Backend
    AddBackend(backend *Backend)
    SetBackendStatus(uri *url.URL, alive bool)
}
```

3. Implement Thread-Safe Server Pool

Your ServerPool must handle concurrent requests.

- Implement **GetNextValidPeer()** using an atomic counter for Round-Robin.
- Ensure that only "Alive" backends are returned.
- If no backends are available, the proxy should return an appropriate HTTP error (e.g., 503 Service Unavailable).

4. The Proxy Handler

Use the **net/http** package to serve the proxy.

- Incoming requests must be passed to the LoadBalancer.
- Use **httputil.NewSingleHostReverseProxy** or customize your own **httputil.ReverseProxy**.
- **Context:** Ensure the request context is passed through so that slow backend processing can be canceled.

5. Implement Periodic Health Checking

Create a background goroutine that runs at a configurable interval (e.g., every 30 seconds).

- It should iterate through all backends in the ServerPool.
- Perform a "ping" (a simple GET request or TCP dial).
- Update the Alive status of the backend.
- Log changes in status (e.g., "Backend https://api1.host.com is DOWN").

6. Admin API Endpoints

Expose a management interface (perhaps on a different port, e.g., :8081).

- GET /status: Return a JSON list of all backends and their current health/load.
- POST /backends: Dynamically add a new backend URL to the pool.
- DELETE /backends: Remove a backend from the pool.

Implementation Steps

1. **Project Setup:**
   - Initialize the module: go mod init <your_project_name>.
   - Create a config.json to store initial backend URLs.
2. **Logic Flow:**
   - **Main Goroutine:** Starts the HTTP Proxy server and the Admin API.
   - **Health Check Goroutine:** Runs a time.Ticker loop to verify backend availability.
   - **Request Handling:** When a request arrives, the ServerPool selects a backend, increments its connection count, forwards the request, and decrements the count upon completion.
3. **Error Handling:**
   - Implement a custom ErrorHandler for the proxy to catch connection-refused errors from backends and mark them as dead immediately.

Grading Criteria

Criteria	Points
Proxy Functionality: Correctly forwards traffic and returns responses.	30

Concurrency & Safety: Proper use of Mutexes/Atomic to prevent race conditions.	25
Background Job: Health checker correctly pings and updates backend states.	15
Load Balancing Logic: Successful implementation of Round-Robin or Least-Connections.	10
Context & Timeouts: Handling of client cancellations and backend timeouts.	10
Code Organization & Docs: Clean package structure and README instructions.	10
Total	**100**

Example Usage

Starting the Proxy:

```
go run main.go --config=config.json
```

Configuring a Backend via Admin API:

```
curl -X POST http://localhost:8081/backends \
     -H "Content-Type: application/json" \
     -d '{"url": "http://localhost:8082"}'
```

Checking System Status:

```
// GET /status
{
  "total_backends": 3,
  "active_backends": 2,
  "backends": [
    {
      "url": "http://app-server-1:8080",
      "alive": true,
      "current_connections": 12
    },
    {
      "url": "http://app-server-2:8080",
      "alive": false,
      "current_connections": 0
    }
  ]
}
```

Optional Enhancements

- **Sticky Sessions:** Implement logic to ensure a specific client (by IP or Cookie) always hits the same backend.
- **Weighted Load Balancing:** Assign weights to backends based on their hardware capacity ($Weight_i$).
- **HTTPS/TLS Support:** Allow the proxy to serve traffic over SSL.

Good luck, and happy coding!