University of Science and Technology of Hanoi

Distributed systems

LOAD BALANCER TO HTTP SERVER

Group 16

Le Minh Hoang (BI12-167)

Nguyen The Hoang (BI12-171)

Nguyen Minh Hoang (BI12-172)

Le Bui Huy Hoang (BI12-174)

Le Tuan Huy (BI12-195)

Nguyen Dac Minh Khoi (BI12-218)

April 2024

Contents

Chapter 1

Scenario

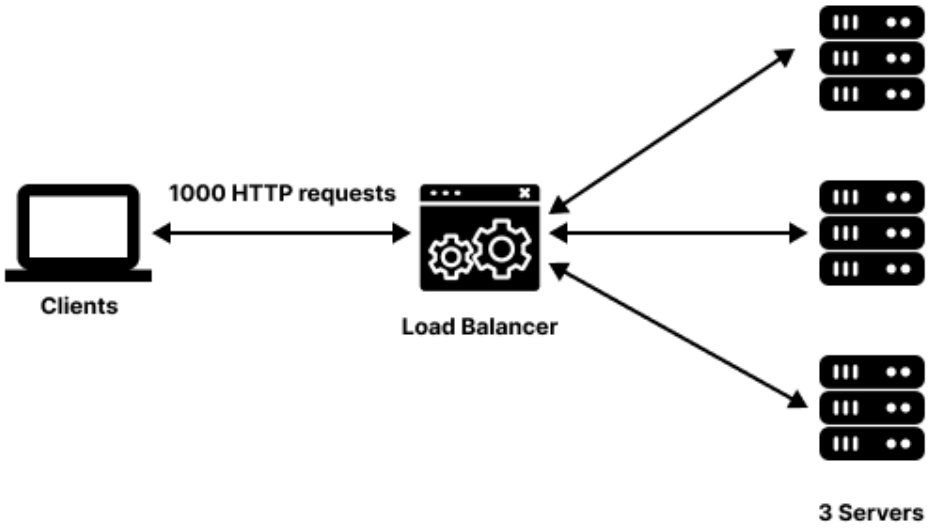

Figure 1.1: Scenario Design

We have a load balancer distributing 1000 requests from a client to three servers(containers):

- Server 1 (localhost:8081)
- Server 2 (localhost:8082)
- Server 3 (localhost:8083)

Each server has a different weight assigned to it, indicating its capacity to handle requests. The load balancer logs each request, recording the client IP address and the server it was directed to.

Chapter 2

Design

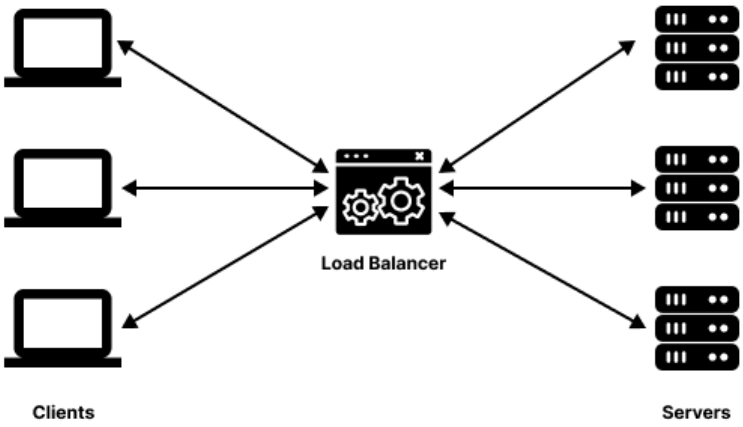

Figure 2.1: Load Balancing Architecture

2.1 Configuration Parser:

- The load balancer supports reading server configurations from a YAML file named `config.yaml`.
- If the YAML file is not found or has an error, the load balancer uses a default set of servers defined in the code.
- The configuration file specifies an array of server configurations, each containing an address, port, and weight.

2.2 Server Representation:

Each backend server is represented by a `Server` struct, which contains the server's URL and a weight value. The severs slice holds all the configured backend servers.

2.3 Round-Robin with Weighted Distribution:

- The load balancer uses a round-robin algorithm with weighted distribution to select the next backend server for each incoming request with following fomula:

$$r_{next} = (r_{current} + 1) \bmod n$$

- The `getTotalWeight` function calculates the total weight of all servers.
- The `getNextServerIndex` function determines the index of the next server to use based on the current server index and the weights of the servers.
- The `currentServer` variable keeps track of the index of the last server used, and it is atomically incremented for each request.

2.4 Health Checking:

Before forwarding a request to a backend server, the load balancer checks the server's health by sending a GET request to the `/health` endpoint.

- If the server is available (responds with a 200 OK status code), the load balancer creates a reverse proxy for that server and forwards the incoming request.
- If the server is not available, the load balancer tries the next server in the list.
- If none of the servers are available, the load balancer logs a message indicating that all servers are down.

2.5 Request Handling:

- The load balancer listens on port 8080 for incoming requests.
- For each incoming request, the `handleRequestAndRedirect` function is called.
- This function retrieves a reverse proxy for the next available server using the `getProxy` function.
- The reverse proxy is then used to forward the request to the selected backend server. A timeout of 10 seconds is set for the request to the backend server.

2.6 Reverse Proxy:

- The load balancer uses the `httputil.ReverseProxy` from the Go standard library to forward requests to the selected backend server.
- The `ReverseProxy` automatically handles tasks like URL rewriting, forwarding headers, and sending the response back to the client.

2.7 Application flow

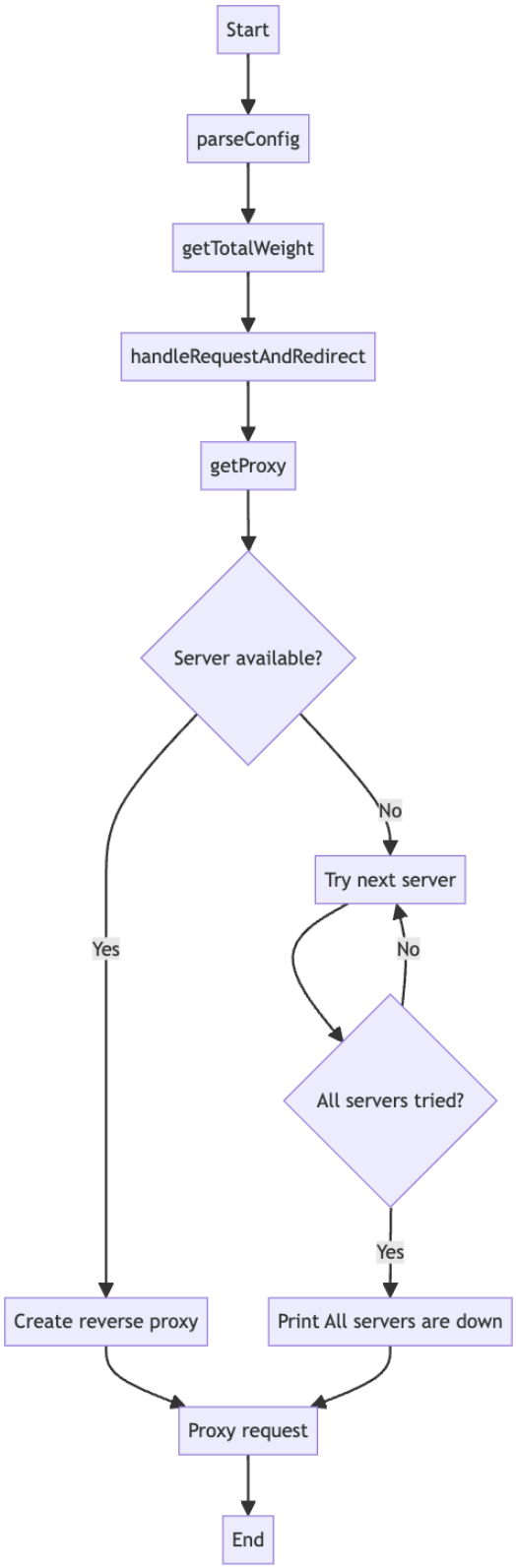

Figure 2.2: Application flow diagram

Chapter 3

Guide to build and run code

3.1 Pre-installation

3.1.1 Install Go

- https://go.dev/doc/install

3.1.2 Install Go packages

```
go mod tidy
```

3.1.3 Config server list

Edit `config.yaml` file to add server list

```
servers:
  - address: localhost
    port: 8081
    weight: 1
  - address: localhost
    port: 8082
    weight: 2
  - address: localhost
    port: 8083
    weight: 3
```

3.1.4 For testing

Install Docker and Docker Compose

- https://docs.docker.com/get-docker/

3.1.5 For analyzing

Install python (3.10+)

3.2 Usage Testing

3.2.1 Build main.go

```
go build main.go
```

3.2.2 Run main.go

```
./main
```

3.2.3 Server Setup

1. Setting up the running scirpt for server:

   - Setting up a Handler class for request:

```
        class RequestHandler(BaseHTTPRequestHandler):
        def do_GET(self):
            client_ip = self.client_address[0]
            server_ip = socket.gethostbyname(socket.gethostname())
            path = self.path
            if path == '/health':
                self.send_response(200)
                self.send_header('Content-type', 'text/plain')
                self.end_headers()
                self.wfile.write(b"OK")
                return
            else:
                response = f"Port: {os.environ.get('SERVER_PORT', 8080)}\
    nServer
                IP: {server_ip}\nClient IP: {client_ip}\nPath: {path}\n"
                self.send_response(200)
                self.send_header('Content-type', 'text/plain')
                self.end_headers()
                self.wfile.write(response.encode())

                with open('data.log', 'a') as f:
                    f.write(f"{server_ip},{client_ip},{path}\n")

```

   - This class defines a custom request handler by subclassing BaseHTTPRequestHandler. It overrides the do_GET method to handle GET requests.
     - It extracts the client's IP address using self.client_address.
     - Retrieves the server's IP address using socket.gethostbyname(socket.gethostname()).
     - Gets the requested path from self.path.
     - If the path is '/health', it responds with "OK".
     - Otherwise, it constructs a response containing server and client information and sends it back to the client.
     - It also logs server and client information to a file named 'data.log'.

2. Build Docker image for containers:

   - dockerfile:

```
        FROM python:3.9-slim
        WORKDIR /app
        COPY requirements.txt .

        # install requirements
        RUN pip install --no-cache-dir -r requirements.txt

        # Copy all running scripts
        COPY . .

        # load environment SERVER_PORT to arg[1]
        CMD ["python", "server.py", "$SERVER_PORT"]

```

   - build command:

```
        docker build -t {{image_name}} {{path_to_dockerfile}}

```

3.2.4 Run docker cluster

```
cd docker && docker-compose up
```

3.2.5 Test with 1000 requests

- We use docker-compose to create containers from docker-compose.yaml file to act as servers for the load balancer to distribute traffic
- The load balancer will be access through http://localhost:8080

```
    # Define the URL to send requests to
    url = "http://localhost:8080"

```

- Set the number of request to be send to the load balancer:

```
    # Define the number of requests to send
    num_requests = 1000

```

- For each request we got a response from the containers through the load balancer

```
    for i in range(num_requests):
    try:
        # Send GET request
        response = requests.get(url, headers=headers)

        # Print status code and content of response
        print(f"Request {i+1}: Status Code - {response.status_code}, Response -
    {response.text}")

```

```
    except requests.RequestException as e:
        # Print error if request fails
        print(f"Request {i+1}: Failed - {e}")

```

- To run the script:

```
cd test_scripts && python call.py && cd ../
```

3.2.6 Make test log for analyzing

Please delete docker/data.log before running the test for the correct result

```
docker/data.log
```

Send 1000 requests with 3 servers up

```
python test_scripts/call.py && cp docker/data.log docker/3servers.log && > docker/
    data.log
```

Now, you can shut down 1 server in docker and send 1000 requests again with 2 servers up

```
python test_scripts/call.py && cp docker/data.log docker/2servers.log && > docker/
    data.log
```

Now, shut down 1 more server in docker and send 1000 requests again with 1 server up

```
python test_scripts/call.py && cp docker/data.log docker/1server.log && > docker/data
    .log
```

Chapter 4

Analyzing

4.1 Distributing result

```
cd test_scripts && python analyze.py
```

Example output:

```
Analyze for all 3 servers up:
CLIENT 0: 172.20.0.1
172.20.0.3: 334 requests
172.20.0.2: 500 requests
172.20.0.4: 166 requests
====================

Analyze for all 2 servers up:
CLIENT 0: 172.20.0.1
172.20.0.2: 499 requests
172.20.0.3: 501 requests
====================

Analyze for all 1 servers up:
CLIENT 0: 172.20.0.1
172.20.0.3: 1000 requests
====================
```

This result indicates that HTTP load balancer has the ability the of distributing the server based on the weight of servers. And in the case of 1 or 2 server downs, the traffic can be distributed to the remaining servers.

4.2 Performance result

We ran the 5 parallel 1000 servers requests in python to capture the response time of the server, and we got the following graph. The graph indicates that the load balancer has the capability to handles around 800 rps with a single client. This result is vary based on that the client is only able to send 800 rps or the limit of python script.

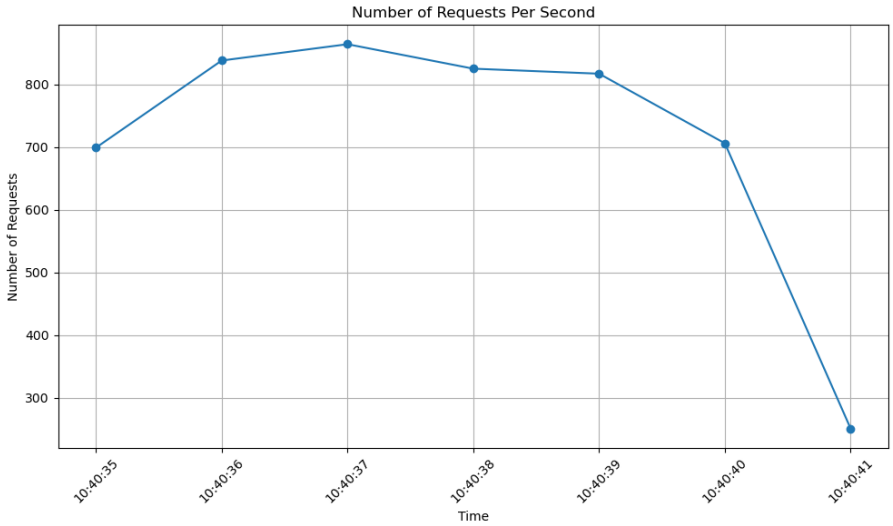

Figure 4.1: Performance graph

Chapter 5

Conclusion

- The load balancer distributes incoming HTTP requests to multiple servers based on the round-robin algorithm.
- The health check feature is implemented to check the status of the servers.
- The server list is configurable in the config.yaml file.
- The load balancer is tested with 1000 requests and analyzed for different scenarios.
- The result shows that the load balancer distributes the requests based on the weight of the servers.
- Performance testing showed the load balancer could handle around 800 requests per second from a single client under the testing conditions.
- Overall, this solution effectively distributes HTTP traffic based on weighted capacity and health status, with potential for adding additional features in the future.
- For further references see github.com/hzhoanglee/simple-http-lb-usth

Chapter 6

Reference

1. https://go.dev/src/net/http/httputil/reverseproxy.go

2. https://avinetworks.com/glossary/round-robin-load-balancing/