Load Balancer

Sprawozdanie z projektu

Programowanie Równoległe i Rozproszone

Jakub Łaba oraz **Maciej Michalski**

Politechnika Warszawska
Wydział Elektryczny
Czerwiec 2023

Spis treści

1 Wykorzystane technologie

Projekt został wykonany w języku **Rust** z wykorzystaniem biblioteki **Tokio**.
Do testów i prezentacji działania aplikacji zostały wykorzystane instancje **Mockservera**, zarządzane za pomocą **Dockera** oraz **docker-compose**.
Kod źródłowy projektu jest w całości dostępny publicznie na **GitHubie**.

2 Założenia

Podstawowymi założeniami projektu była implementacja kilku różnych algorytmów rozdzielania ruchu sieciowego po obsługiwanych serwerach, oraz możliwość konfiguracji za pomocą plików konfiguracyjnych. Oba te założenia udało się spełnić – aplikację można konfigurować za pomocą pliku JSON, o następującej strukturze:

```
{
    "address": "localhost:80",
    "strategy": "RoundRobin",
    "receiver_addresses": [
        "localhost:81",
        "localhost:82",
        // ...
    ]
}
```

Gdzie:

- `address` – Adres, na którym będzie działał load balancer. Parametr opcjonalny, w przypadku jego braku w pliku, domyślna wartość to `localhost:8080`
- `strategy` – Strategia rozdzielania ruchu sieciowego, z której będzie korzystał load balancer. Dostępne opcje to:
  - `Random` – Ruch będzie przekierowywany do obsługiwanych serwerów w sposób losowy
  - `RoundRobin` – Ruch będzie przekierowywany do obsługiwanych serwerów algorytem Round Robin (karuzelowym)
  - `IpHash` – Ruch będzie przekierowywany do obsługiwanych serwerów za pomocą wartości hasza adresu, z którego przychodzi zapytanie.

  Parametr opcjonalny, w przypadku jego braku w pliku, domyślna wartość to `Random`.
- `receiver_addresses` – Lista adresów obsługiwanych serwerów. Parametr wymagany.

3 Możliwości rozbudowy

Pomimo osiągnięcia podstawowych celów projektu, nadal istnieje mnóstwo możliwości rozbudowy funkcjonalności, na przykład:

- Monitorowanie zdrowia serwerów – prawdziwe produkcyjne load balancery zazwyczaj idą w parze z serwisem monitorującym zdrowie serwerów – taki serwis okresowo odpytuje serwery, aby sprawdzić czy działają poprawnie, i raportuje wyniki.
- Dodanie większej ilości strategii rozdziału ruchu sieciowego do serwerów, np. ważone wersje zaimplementowanych algorytmów, rozdział na podstawie ilości obecnie obsługiwanych przez poszczególne serwery zapytań, itd.
- Możliwość zmiany konfiguracji load balancera bez restartu całej aplikacji – prawdziwe produkcyjne load balancery pełnią dość krytyczną rolę w zarządzaniu działaniem aplikacji, w środowisku produkcyjnym niezbyt można pozwolić sobie na restart całej usługi w celu zmiany konfiguracji.
- Obsługa certifikatów SSL - w faktycznym środowisku produkcyjnym nie można by było pozwolić sobie na komunikację `http` zamiast `https`, tak jak ma to miejsce w tym projekcie.

4 Testy

4.1 Jednostkowe

Za pomocą testów jednostkowych zostały przetestowane moduły:

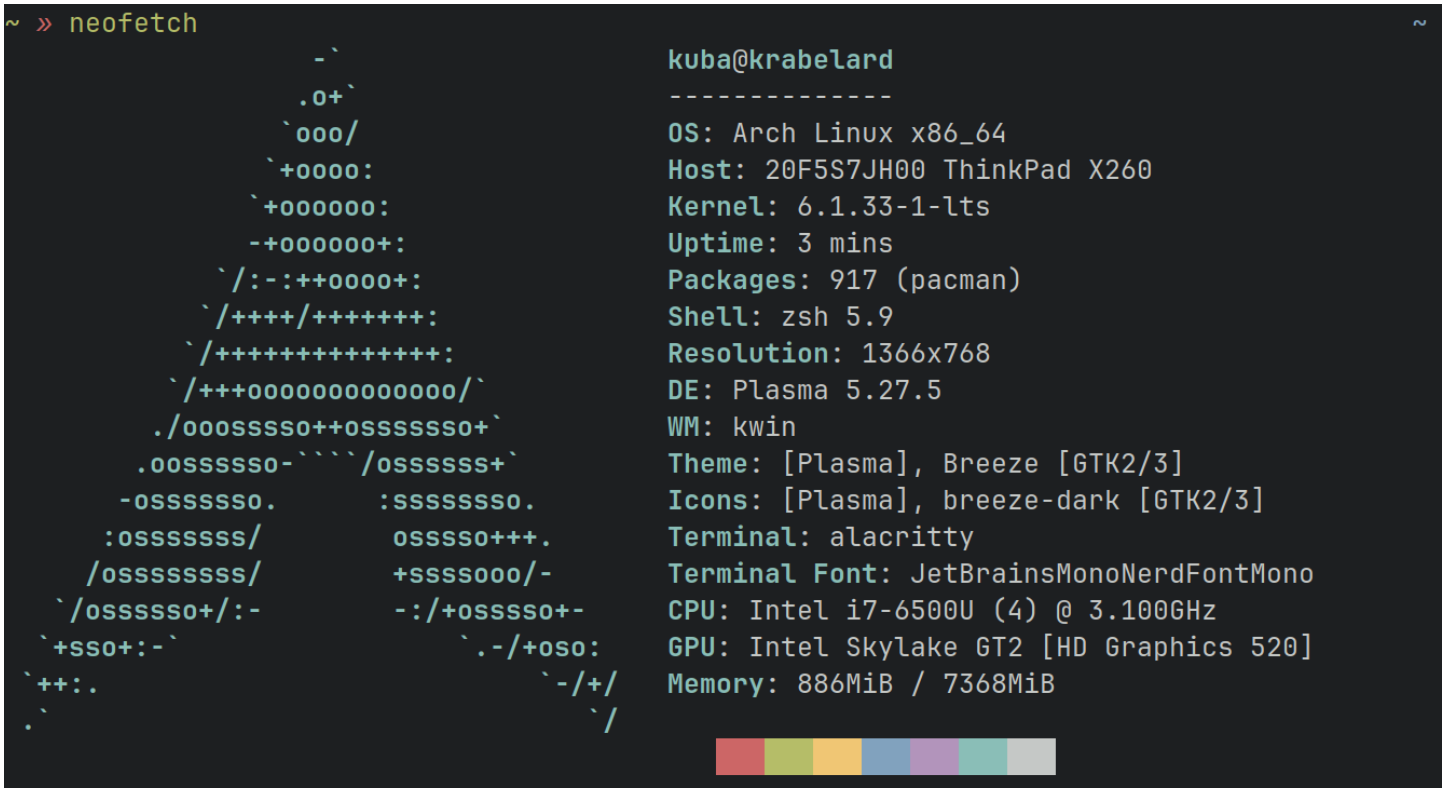
- `config` – moduł odpowiedzialny za parsowanie plików konfiguracyjnych, w testach została zweryfikowana poprawność wczytywania danych oraz parametrów domyślnych w przypadku ich braku
- `load_balancing` – moduł odpowiedzialny za różne strategie rozdzielania ruchu sieciowego, w testach została zweryfikowana poprawność działania każdego z nich:
  - `Random` – w przypadku tej strategii nie było zbyt wiele do testowania, zostało zweryfikowane czy losowane adresy faktycznie istnieją na liście obsługiwanych serwerów
  - `IpHash` – w przypadku tej strategii zostało zweryfikowane, czy wyznaczane adresy istnieją na liście, oraz czy obliczanie hashów jest deterministyczne (zawsze taki sam hash dla takiego samego adresu, z którego przychodzi zapytanie)
  - `RoundRobin` – w przypadku tej strategii zostało zweryfikowane czy wyznaczane adresy istnieją na liście, oraz czy są wyznaczane w odpowiedniej kolejności

4.2 Benchmarkowe

Do testów benchmarkowych zostało wykorzystane narzędzie **ApacheBench** oraz wspomniane we wstępie środowisko dockerowe z wykorzystaniem mockserverów.
Testowana wersja programu została skompilowana w zoptymalizowanej wersji (`cargo build -release`).

4.2.1 Platforma

OS	Arch Linux x86_64
Kernel	6.1.33-1-lts
CPU	Intel Core i7-6500U (2 rdzenie, 4 wątki, 3.1GHz
RAM	8GB



4.2.2 Parametry

Liczba instancji mockservera	10
Liczba wątków tokio	12
Liczba zapytań łącznie	100000
Liczba zapytań na raz	1000

4.2.3 Wyniki

- Random

```
Concurrency Level:      1000
Time taken for tests:   202.222 seconds
Complete requests:      100000
Failed requests:        0
Total transferred:      8400000 bytes
HTML transferred:       2600000 bytes
Requests per second:    494.51 [#/sec] (mean)
Time per request:       2022.216 [ms] (mean)
Time per request:       2.022 [ms] (mean, across all concurrent requests)
Transfer rate:          40.57 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.4      0      33
Processing:    14 2012 125.9   2022    2345
Waiting:        2 2012 125.9   2022    2345
Total:         36 2013 123.9   2022    2345

Percentage of the requests served within a certain time (ms)
  50%   2022
  66%   2038
  75%   2051
  80%   2058
  90%   2077
  95%   2098
  98%   2125
  99%   2194
 100%   2345 (longest request)
```

- RoundRobin

```
Concurrency Level:      1000
Time taken for tests:   221.028 seconds
Complete requests:      100000
Failed requests:        0
Total transferred:      8400000 bytes
HTML transferred:       2600000 bytes
Requests per second:    452.43 [#/sec] (mean)
Time per request:       2210.279 [ms] (mean)
Time per request:       2.210 [ms] (mean, across all concurrent requests)
Transfer rate:          37.11 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.4      0      32
Processing:    14 2200 213.8   2179    2811
Waiting:        1 2200 213.8   2179    2811
Total:         34 2200 212.5   2179    2811

Percentage of the requests served within a certain time (ms)
  50%   2179
  66%   2254
  75%   2297
  80%   2334
  90%   2455
  95%   2546
  98%   2680
  99%   2718
 100%   2811 (longest request)
```

- IpHash

```
Concurrency Level:      1000
Time taken for tests:   296.590 seconds
Complete requests:      100000
Failed requests:        0
Total transferred:      8400000 bytes
HTML transferred:       2600000 bytes
Requests per second:    337.17 [#/sec] (mean)
Time per request:       2965.895 [ms] (mean)
Time per request:       2.966 [ms] (mean, across all concurrent requests)
Transfer rate:          27.66 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.9      0      22
Processing:   196 2949 749.9   2687    7312
Waiting:      195 2949 749.8   2687    7312
Total:        218 2949 750.4   2687    7328

Percentage of the requests served within a certain time (ms)
  50%   2687
  66%   2964
  75%   3332
  80%   3546
  90%   4130
  95%   4506
  98%   4821
  99%   4977
 100%   7328 (longest request)
```

5 Wnioski

Zarówno pod względem stabilności, jak i średniego czasu obsługi zapytania, najlepiej wypadła strategia `Random`. Może być to podyktowane tym, że testy były uruchamiane na laptopie z serii ThinkPad, które są na ogół wyposażone w wydajne hardware'owe generatory liczb losowych. Przez *"stabilność"* rozumiemy tutaj odchylenie standardowe wyników – w wynikach widać, że różnica między 50 a 100 centylem jest niewielka (~0.3s). Zaskakująca jest znaczna przewaga `RoundRobin` nad `IpHash`, ze względu na to, że algorytm korzystający z zamka okazał się wydajniejszy od tego bez potrzeby synchronizacji danych (hashe mogą być liczone niezależnie, natomiast licznik w `RoundRobin` musi być synchronizowany). Algorytm `IpHash` okazał sie bardzo niestabilny - różnica między 99 a 100 centylem jest tutaj większa niż między 50 a 100 w pozostałych algorytmach. Na mniejszej ilości danych można by uznać to za przypadkowy wynik zaniżający średnią, ale należy przypomnieć, że dla przyjętych parametrów testów, na każdy centyl składa składa się 1000 pomiarów, a więc wynik można uznać za miarodajny.