Sprawozdanie – programowanie równoległe i rozproszone

Projekt końcowy – Load-balancer jako inteligentna usługa sieciowa

Adrian Nowosielski, Cezary Skorupski

1. Zarys teoretyczny problemu

Celem tego projektu końcowego było stworzenie inteligentnej usługi sieciowej, która pracowałby jako równoważnik obciążenia pomiędzy serwisami. W dzisiejszych systemach mamy dużo problemów z przydzieleniem odpowiednich zadań do serwerów, które wykonują dane zadania. Chcemy przede wszystkim, aby serwery wykonywały jak najszybciej dane zadania i nie były bardzo przeciążone. Odpowiedzią na te problemy jest technika nazywana Load-Balancingiem – technika wykorzystuje algorytmy heurystyczne, które równoważą obciążenie wykonywanych zadań pomiędzy serwerami w aplikacji rozproszonej. Load-Balancer jest jednym z podstawowych narzędzi do wykonania pełnego mapowania zadania równoległego zgodnie z metodyką PCAM do tworzenia algorytmów równoległych. W celu stworzenia Load-Balancera, zespół musiał zapoznać się z podstawowymi pojęciami, które opisują działanie równoważenia obciążenia. Przykładowe pojęcia, które były przydatne zespołowi podczas tworzenia projektu prezentują się następująco:

Podstawowe pojęcia

- **Serwer**: Komputer lub system komputerowy, który udostępnia zasoby, dane lub usługi klientom, na przykład poprzez Internet.
- **Klient**: Urządzenie lub aplikacja, która korzysta z zasobów lub usług dostarczanych przez serwer.
- **Sesja**: Pojedyncze połączenie lub interakcja między klientem a serwerem.
- **Zasoby sieciowe**: Infrastruktura, na której opierają się usługi sieciowe, obejmująca serwery, bazy danych, aplikacje, itd.

Następnie, zespół zadał sobie pytanie, dlaczego równoważnie obciążenia w systemach rozproszonych jest ważne i dlaczego load-balancer jest techniką, która jest lepsza niż inne podejścia do równoważnia obciążenia:

Dlaczego równoważenie obciążenia jest potrzebne?

- **Wysoka dostępność:** Load-balancer umożliwia dystrybucję ruchu w taki sposób, aby w przypadku awarii jednego z serwerów inne mogły przejąć jego obciążenie, co minimalizuje ryzyko przestojów.
- **Skalowalność:** Umożliwia dynamiczne dodawanie lub usuwanie serwerów w odpowiedzi na zmieniające się obciążenie, co pozwala na elastyczne zarządzanie zasobami.
- **Optymalizacja wydajności:** Poprzez efektywną dystrybucję ruchu, load-balancer minimalizuje czas odpowiedzi i zapewnia równomierne obciążenie serwerów, co maksymalizuje ich wydajność.

Kolejnym aspektem ważnym przy tworzeniu systemów do równoważnia obciążenia jest fakt, czy dane load-balancer jest sprzętowy czy programowy:

Rodzaje load-balancerów

- **Sprzętowe**: Fizyczne urządzenia, które są umieszczane w infrastrukturze sieciowej. Zapewniają wysoką wydajność, ale są kosztowne i mniej elastyczne.
- **Programowe**: Oprogramowanie instalowane na serwerach, które realizuje funkcje load-balancing. Są tańsze i bardziej elastyczne, ale mogą obciążać zasoby serwera.
- **Chmurowe**: Usługi load-balancing oferowane przez dostawców chmur obliczeniowych, takie jak Amazon Web Services (AWS) Elastic Load Balancing. Łatwe do skonfigurowania i skalowalne w zależności od potrzeb.

Kolejnym aspektem implementacji systemu równoważenia obciążenia jest wybór odpowiednich algorytmów do równoważenia obciążenia pomiędzy serwerami. W tym przypadku mamy następujące algorytmy równoważenia obciążenia:

Algorytmy równoważenia obciążenia

- **Round Robin**: Najprostszy algorytm, który przekazuje każde kolejne żądanie do następnego serwera w kolejce.
- **Least Connections**: Kieruje nowe żądania do serwera z najmniejszą liczbą aktywnych połączeń.
- **IP Hash**: Używa hash kodu adresu IP klienta do wyboru serwera, co może być przydatne do utrzymania sesji użytkownika na tym samym serwerze.
- **Weighted Round Robin**: Każdy serwer ma przypisaną wagę, a żądania są kierowane zgodnie z proporcjami tych wag.
- **Dynamiczne algorytmy**: Algorytmy wykorzystujące uczenie maszynowe i sztuczną inteligencję do przewidywania obciążenia i optymalizacji dystrybucji ruchu.

Warto dodać, że do równoważnia obciążenia wykorzystywane są również elementy uczenia maszynowego:

Uczenie maszynowe w load-balancing

Predykcja obciążenia: Wykorzystanie historycznych danych o ruchu sieciowym do przewidywania przyszłych obciążeń.

Optymalizacja zasobów: Automatyczne skalowanie zasobów (dodawanie lub usuwanie serwerów) w oparciu o przewidywane potrzeby.

Adaptacyjność: System może adaptować się do zmieniających się warunków w czasie rzeczywistym, ucząc się na podstawie bieżących danych.

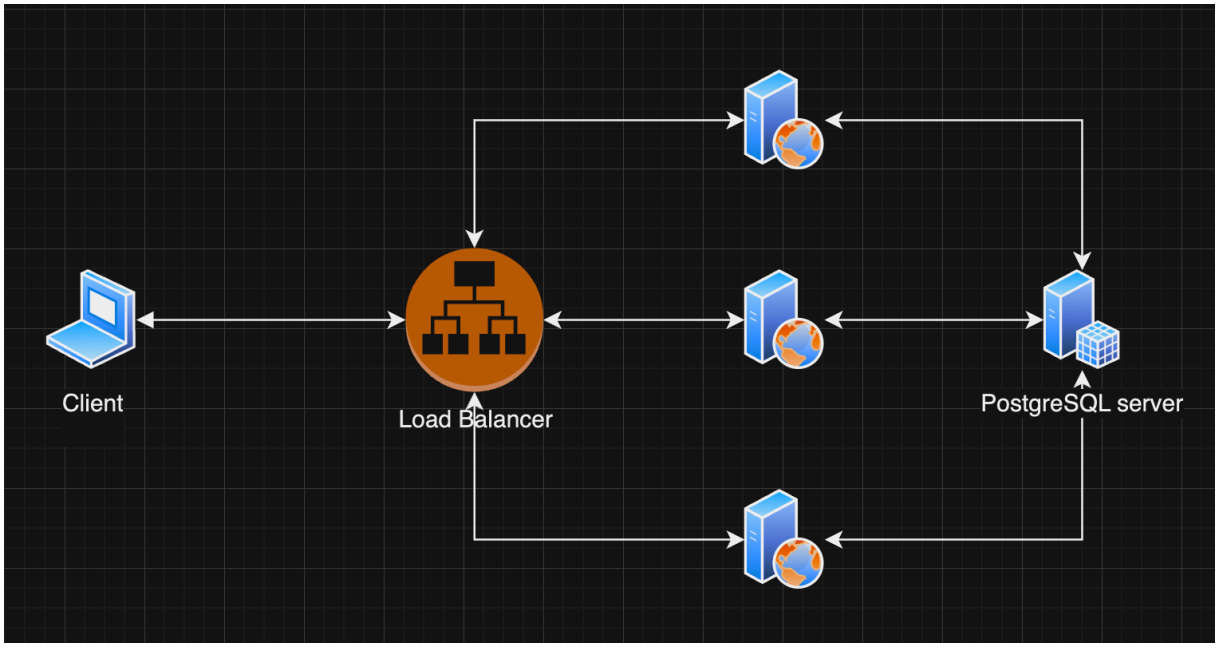
2. Zarys sposobu implementacji problemu

W celu implementacji problemu równoważenia obciążenia w systemie informatycznym wybraliśmy język Java. W języku Java musieliśmy napisać zasadniczo trzy oddzielne projekty, w którym jednym z nich jest serwer, jednym klient oraz jednym serwer do równoważenia obciążenia. W celu implementacji problemu równoważnia obciążenia wybraliśmy dwa algorytmy: algorytm round-robin oraz algorytm najmniejszej liczby połączeń. Następnie musieliśmy zaimplementować sposób komunikacji pomiędzy na początku klientem i load-balancerem, później load-balancerem i workerem.

Sposób w jaki zaimplementowaliśmy komunikację został oparty na mechanizmie gniazdek. Mechanizm ten pozwolił nam przekazywać informacje pomiędzy klientem w sposób połączeniowy. Na początku tworzyliśmy odpowiednie gniazdko na serwerze, potem łączyliśmy się z tym gniazdkiem przy kliencie oraz przekazywaliśmy odpowiednie dane w problemie.

Ostatnim etapem implementacji problemu była implementacja sposobu wykonywanych zadań przez workera. Zespół podjął decyzję o zaimplementowaniu mechanizmu, który będzie komunikować się z systemem bazy danych oraz ściągać określone rekordy z bazy danych, które chciał worker na podstawie identyfikatora. Worker w projekcie odpowiada za wysyłanie zapytania do bazy danych PostgreSQL i następnie odsyła zapytanie do mechanizmu load-balancera. Następnie load-balancer odsyła odpowiedź do klienta.

Sposób w jaki możemy zaprezentować komunikację pomiędzy serwerami oraz pojedynczymi workerami możemy zobrazować następującym diagramem:



3. Sposób implementacji problemu

3.1: Sposób implementacji klienta

Zasadniczo naszą pracę nad projektem zaczęliśmy od implementacji klienta, który będzie komunikować się z load-balancerem. Klient musiał wysłać odpowiednie dane do mechanizmu równoważenia obciążenia oraz ostatecznie uzyskać dane obliczone przez workera oraz wypisać daną na wyjście. Sposób implementacji klienta w języku Java prezentuje się następująco:

```java
@SneakyThrows  Adrian Nowosielski
public static void main(String... args) {
    logger.info(
        "Started client program with thread name=[{}]",
        Thread.currentThread().getName()
    );

    while (!Thread.interrupted()) {
        Socket loadBalancerSocket = new Socket(
            ClientConstants.LOAD_BALANCING_SERVER_URL,
            ClientConstants.LOAD_BALANCING_SERVER_PORT
        );

        logger.info("Sent the load-balancer request for data");
        Thread clientRequestSender = new ClientRequestSender(loadBalancerSocket);
        clientRequestSender.start();

        Thread.sleep(
            ClientConstants.TIME_OF_WAITING_BETWEEN_TASKS_IN_MILLISECONDS
        );
    }
}
```

```java
@Override  Adrian Nowosielski
@SneakyThrows
public void run() {
    int packageId = new Random()
        .nextInt(ClientConstants.NUMBER_OF_PACKAGES_DATA_IN_DATABASE) +
    1;

    SocketHelpers.writeStringToSocket(
        String.valueOf(packageId),
        loadBalancingSocket
    );

    String responseValueFromLoadBalancer = SocketHelpers.extractStringFromSocket(
        loadBalancingSocket
    );
    logger.info(
        "Client got the response from load-balancer with value=[{}]",
        responseValueFromLoadBalancer
    );
}
```

Oczywiście do implementacji problemu potrzebowaliśmy również mechanizmu Thread, który pozwolił wykonywać nam obliczenia na różnych wątkach w systemie. Dodaliśmy również czas pomiędzy żądaniami do load-balancera, aby lepiej zauważyć mechanizm równoważenia obciążenia.

3.2: Sposób implementacji workera

Kolejnym elementem implementacji systemu była implementacja systemu workera. Tak jak opisywaliśmy wcześniej – worker ma na celu połączenie się z systemem bazy danych oraz następnie wykonać zapytanie do bazy danych i przesłać odpowiedź do load-balancera, który następnie odeśle odpowiedź do klienta wysyłającego żądanie. Początkowym problemem było stworzenie bazy danych do otrzymywania żądań oraz przetrzymywania danych. W celu implementacji wykorzystaliśmy mechanizm docker, który stworzył lekki system bazodanowy z przykładowymi danymi na naszym komputerze. Sposób implementacji obrazu docker oraz dane prezentują się następująco:

```
FROM postgres:latest

# Copying the data into the container and setting password and user
ENV POSTGRES_USER=postgres
ENV POSTGRES_PASSWORD=root
COPY init.sql /docker-entrypoint-initdb.d/
```

```
CREATE TABLE packages (
    id SERIAL PRIMARY KEY,
    city VARCHAR(255),
    owner_name VARCHAR(255)
);

-- Insert data into the packages table
INSERT INTO packages (city, owner_name) VALUES ( city 'New York', owner_name 'John Doe');
INSERT INTO packages (city, owner_name) VALUES ( city 'Los Angeles', owner_name 'Jane Smith');
INSERT INTO packages (city, owner_name) VALUES ( city 'Chicago', owner_name 'Mike Johnson');
INSERT INTO packages (city, owner_name) VALUES ( city 'Warsaw', owner_name 'Adrian Nowosielski');
INSERT INTO packages (city, owner_name) VALUES ( city 'Warsaw', owner_name 'Cezary Skorupski');
INSERT INTO packages (city, owner_name) VALUES ( city 'Warsaw', owner_name 'Dawid Dobosz');
INSERT INTO packages (city, owner_name) VALUES ( city 'Warsaw', owner_name 'James Sullivan');
```

Kolejnym krokiem w implementacji problemu była implementacja komunikacji workera z klientami oraz bazą danych w Javie. Do tego użyliśmy odpowiedniej biblioteki do nawiązywania połączeń z silnikiem SQL oraz oczywiście gniazdek do przesyłania danych pomiędzy elementami systemu. Metoda main workera oraz wątek, który obsługuje nawiązanie połączenia prezentuje się następująco:

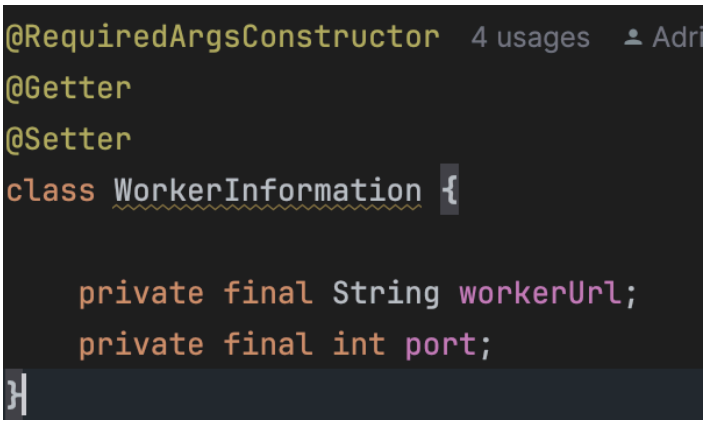
```java
@SneakyThrows  Adrian Nowosielski
public static void main(String... args) {
    Class.forName(DATABASE_PACKAGE_JAR);
    Connection databaseConnection = DriverManager.getConnection(
        DATABASE_URL,
        DATABASE_USER,
        DATABASE_PASSWORD
    );
    ServerSocket workerSocket = new ServerSocket(Integer.parseInt(args[0]));
    logger.info(
        "Started worker program with thread name=[{}]",
        Thread.currentThread().getName()
    );

    while (!Thread.interrupted()) {
        Socket loadBalancerSocket = workerSocket.accept();

        Thread workerTask = new Thread(
            new WorkerTask(loadBalancerSocket, databaseConnection)
        );

        workerTask.start();
    }
}
```

```java
Statement sqlStatement = databaseConnection.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
ResultSet queryResultSet = sqlStatement.executeQuery(query);
queryResultSet.first();

List<String> packageTableColumnNames = List.of("id", "city", "owner_name");
JsonObject responseJsonObject = new JsonObject();

for (String columnName : packageTableColumnNames) {
    responseJsonObject.addProperty(
        columnName,
        queryResultSet.getString(columnName)
    );
}

logger.info(
    "Sending response object with value=[{}] to the load-balancer",
    responseJsonObject
);

SocketHelpers.writeStringToSocket(
    responseJsonObject.toString(),
    loadBalancingSocket
);
```

Wykorzystaliśmy tutaj biblioteki również, które przesyłały dane w elegancki sposób (w notacji JSON). W tym przypadku musieliśmy skorzystać z biblioteki Gson do tworzenia elementów Json z zwykłych łańcuchów.

3.3: Sposób implementacji load-balancera

Ostatnim etapem implementacji systemu było odpowiednie zaadaptowanie się do workera oraz klienta i ostateczne zaimplementowanie mechanizmu obciążenia pomiędzy elementami. W tym celu zaimplementowaliśmy mechanizm do równoważenia obciążenia w systemie przy pomocy algorytmu RoundRobin oraz Least-Connections. W celu implementacji load-balancera na początku musieliśmy zarejestrować w load-balancerze wszystkie workery, który są w systemie. Zrobiliśmy to w najprostszy sposób, po prostu wczytując z pliku port oraz określony adres do workera. W tym przypadku implementacja czytania odpowiednich workerów oraz zapisanie go w mechanizmie load-balancera prezentuje się następująco:




```
private static List<WorkerInformation> LoadWorkerInformationFromFile() 1 usage
    throws IOException {
    InputStream inputStream = Objects.requireNonNull(
        ClassLoader.getSystemResourceAsStream(FILE_NAME)
    );
    BufferedReader bufferedReader = new BufferedReader(
        new InputStreamReader(inputStream)
    );

    String readLine;
    List<WorkerInformation> workerInformation = new ArrayList<>();

    final int hostUrlArrayFilePosition = 0;
    final int portNumberArrayFilePosition = 1;

    while ((readLine = bufferedReader.readLine()) != null) {
        String[] splitLineWithWorkerArguments = readLine.split( regex: ",");
        workerInformation.add(
            new WorkerInformation(
                splitLineWithWorkerArguments[hostUrlArrayFilePosition],
                Integer.parseInt(
                    splitLineWithWorkerArguments[portNumberArrayFilePosition]
                )
            )
        );
    }
```

Kolejnym elementem w projekcie było zaimplementowanie algorytmów do równoważenia obciążenia. Po wczytaniu elementu oraz otrzymaniu żądania musimy wybrać serwer, do którego przekazywane będą żądania. W naszym przypadku zaimplementowaliśmy algorytmy Round-Robin oraz Least-Connections. Implementacja algorytmów wygląda następująco:

```
switch (loadBalancingAlgorithm) {
    case ROUND_ROBIN -> {
        currentWorkerIndex =
            (currentWorkerIndex + 1) % workerLoads.getWorkerLoads().size();

        LOGGER.log(
            Level.INFO,
            msg: "In round robin algorithm, worker with index=[{0}] has been chosen",
            currentWorkerIndex
        );
    }
    case LEAST_CONNECTIONS -> {
        currentWorkerIndex = workerLoads.getMinLoadServer();
        int chosenWorkerWorkLoad = workerLoads.getSpecificWorkerLoadBasedOnIndex(
            currentWorkerIndex
        );
        LOGGER.log(
            Level.INFO,
            String.format(
                "In least connections algorithm, worker with index=[%d] and workLoad=[%d] has been chosen",
                currentWorkerIndex,
                chosenWorkerWorkLoad
            )
        );

        workerLoads.incrementParticularWorkerLoad(currentWorkerIndex);
    }
```

```
public static WorkerLoads fromWorkerClusterSize(int clusterSize) {  1 usage  Adrian Nowosielski
    WorkerLoads workerLoads = new WorkerLoads();
    final int initialWorkerLoadCount = 0;

    IntStream
        .of( ...values: 0, clusterSize)
        .forEach(element ->
            workerLoads
                .getWorkerLoads()
                .add(new AtomicInteger(initialWorkerLoadCount))
        );

    return workerLoads;
}

public int getSpecificWorkerLoadBasedOnIndex(int workerIndex) {  1 usage  Adrian Nowosielski
    return this.getWorkerLoads().get(workerIndex).get();
}

public synchronized int getMinLoadServer() {  1 usage  Adrian Nowosielski
    AtomicInteger minimumLoadNumber = workerLoads.get(0);
    workerLoads.forEach(workerLoadNumber ->
        minimumLoadNumber.set(
            Math.min(workerLoadNumber.get(), minimumLoadNumber.get())
        )
    );

    return minimumLoadNumber.get();
```

W przypadku algorytmu Round_robin to jedynie co robimy, to dajemy zadanie kolejnemu dostępnemu serwerowi, który jest następny w liście. Jest to stosunkowo prosty algorytm – następnym algorytmem jest algorytm minimalizacji liczby połączeń. Algorytm analizuje aktualną liczbę połączeń do workera oraz wybiera ten, który ma najmniej aktualnych połączeń. Do tego było trzeba stworzyć specjalną klasę WorkerLoads, która przechowywała aktualne przeciążenie elementów oraz odpowiednio inkrementować i dekrementować elementy. Load-balancer musi również przetwarzać określone żądania – kod, który to realizuje prezentuje się następująco:

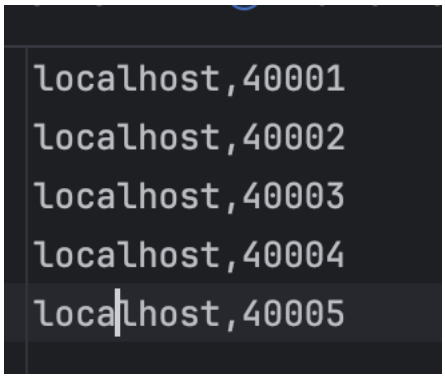
```
);
SocketHelpers.writeStringToSocket(
    packageIdToFetchFromClientSocket,
    workerSocket
);
logger.info(
    "Load balancer sent the request to worker with index=[{}] on port=[{}]",
    currentServer,
    workerSocket.getPort()
);

String responseJsonObjectFromWorker = SocketHelpers.extractStringFromSocket(
    workerSocket
);
SocketHelpers.writeStringToSocket(
    responseJsonObjectFromWorker,
    clientSocket
);
logger.info(
    "Worker with index=[{}] sent the response to loadbalancer",
    currentServer
);

workerSocket.close();
clientSocket.close();

workerLoads.decrementParticularWorkerLoad(currentServer);
```

Warto zauważyć, że gdy przetworzymy całe żądanie, to wtedy dekrementujemy liczbę wywołań żądań aktualną na danym serwerze. Jest to niezbędny element działania algorytmu minimalnych połączeń.

Zasadniczo to wszystkie elementy działania load-balancera w projekcie – po implementacji tych konkretnych rzeczy, load-balancer był gotowy na przechwytywanie żądań oraz przekazywanie żądań do odpowiednich workerów. Pozwoliło to na równoważenie obciążenia

4. Uruchomienie projektu

Kolejnym elementem w tym sprawozdaniu jest pokazanie działania load-balancera na przykładzie. W tym celu musieliśmy uruchomić wiele workerów na maszynie, żeby zobaczyć działanie load-balancera. W tym celu musieliśmy stowrzyć skrypt, który uruchomi wszystkie instancje na raz. Do tego użyliśmy pliku, w którym mieliśmy podane numery portów, na których działały workerzy oraz skrypty w gradle, które uruchamiały aplikacje w Javie. Skrypt, plik z danymi oraz zadania w gradle prezentują się następująco:




```
tasks.register('runWorkers') {
    doLast {
        def argumentsFile : File = file('src/main/resources/workers_list.txt')

        def javaExecTasks = []
        def counter : Integer = 0

        argumentsFile.eachLine { line ->
            println("Running worker program with following arguments: $line")
            def workerArguments : String[] = line.split(",")

            def javaExecTask : TaskProvider<JavaExec> = tasks.register("runWorker${counter}", JavaExec) {
                mainClass = 'ee.pw.microservice.worker.WorkerMain'
                classpath = sourceSets.main.runtimeClasspath
                args(workerArguments[1])
            }

            javaExecTasks.add(javaExecTask)
            counter++
        }

        javaExecTasks.each { task ->
            task.get().exec()
        }
    }
}
```

```
#!/bin/bash

# Running docker container on which the data will be stored
docker build -t postgres-custom .
docker run --name postgres-container -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=root -p 5432:5432 -d
sleep 6

# Then running workers on which the work on db will be done
#./gradlew runWorkers
sleep 3

# Then running load-balancer to serve the clients
#./gradlew runLoadBalancer
sleep 3

# And finally running the client to send the request
#./gradlew runClient
sleep 3
```

Na początku uruchamiamy oczywiście bazę danych na odpowiednim porcie, aby workerzy mogli się do niego podłączyć. W kolejnym kroku uruchamiamy działanie load-balancera i w kolejnym kroku uruchamiamy klienta, który wysyła żądania. Po uruchomieniu wszystkich elementów skryptu mamy następujące wyniki w konsoli:

Otrzymane wyniki żądań od load-balancera:

```
[main] INFO ee.pw.microservice.client.ClientMain - Sent the load-balancer request for data
[Thread-0] INFO ee.pw.microservice.client.ClientRequestSender - Client got the response from load-balancer
 with value=[{"id":"3","city":"Chicago","owner_name":"Mike Johnson"}]
[main] INFO ee.pw.microservice.client.ClientMain - Sent the load-balancer request for data
[Thread-1] INFO ee.pw.microservice.client.ClientRequestSender - Client got the response from load-balancer
 with value=[{"id":"3","city":"Chicago","owner_name":"Mike Johnson"}]
[main] INFO ee.pw.microservice.client.ClientMain - Sent the load-balancer request for data
[Thread-2] INFO ee.pw.microservice.client.ClientRequestSender - Client got the response from load-balancer
 with value=[{"id":"4","city":"Warsaw","owner_name":"Adrian Nowosielski"}]
[main] INFO ee.pw.microservice.client.ClientMain - Sent the load-balancer request for data
[Thread-3] INFO ee.pw.microservice.client.ClientRequestSender - Client got the response from load-balancer
 with value=[{"id":"2","city":"Los Angeles","owner_name":"Jane Smith"}]
[main] INFO ee.pw.microservice.client.ClientMain - Sent the load-balancer request for data
[Thread-4] INFO ee.pw.microservice.client.ClientRequestSender - Client got the response from load-balancer
 with value=[{"id":"2","city":"Los Angeles","owner_name":"Jane Smith"}]
```

Load-balancer – w działaniu możemy znakomicie zobaczyć, jak działa algorytm round-robin. Wybiera z dwóch dostępnych serwerów jeden i przesuwa index następnego wybranego serwera o jeden.

```
[Thread-18] INFO ee.pw.microservice.load_balancer.LoadBalancerMain - Worker with index=[1] sent the
 response to loadbalancer
Jun 06, 2024 11:00:18 PM ee.pw.microservice.load_balancer.LoadBalancerHandler
 getCurrentWorkerIndexBasedOnLoadBalancingAlgorithm
INFO: In round robin algorithm, worker with index=[0] has been chosen
[Thread-19] INFO ee.pw.microservice.load_balancer.LoadBalancerMain - Load balancer sent the request to
 worker with index=[0] on port=[40001]
[Thread-19] INFO ee.pw.microservice.load_balancer.LoadBalancerMain - Worker with index=[0] sent the
 response to loadbalancer
Jun 06, 2024 11:00:20 PM ee.pw.microservice.load_balancer.LoadBalancerHandler
 getCurrentWorkerIndexBasedOnLoadBalancingAlgorithm
INFO: In round robin algorithm, worker with index=[1] has been chosen
[Thread-20] INFO ee.pw.microservice.load_balancer.LoadBalancerMain - Load balancer sent the request to
 worker with index=[1] on port=[40002]
[Thread-20] INFO ee.pw.microservice.load_balancer.LoadBalancerMain - Worker with index=[1] sent the
 response to loadbalancer
```

Worker – w przypadku workera, to jedynie co robi to wysyła zapytanie do bazy danych i następnie odsyła dane do load-balancera. Możemy zobaczyć przykład w następujący sposób:

```
 "city":"Chicago","owner_name":"Mike Johnson"}] to the load-balancer
[Thread-3] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"2",
 "city":"Los Angeles","owner_name":"Jane Smith"}] to the load-balancer
[Thread-5] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"2",
 "city":"Los Angeles","owner_name":"Jane Smith"}] to the load-balancer
[Thread-7] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"1",
 "city":"New York","owner_name":"John Doe"}] to the load-balancer
[Thread-9] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"4",
 "city":"Warsaw","owner_name":"Adrian Nowosielski"}] to the load-balancer
[Thread-11] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"4",
 "city":"Warsaw","owner_name":"Adrian Nowosielski"}] to the load-balancer
[Thread-13] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"3",
 "city":"Chicago","owner_name":"Mike Johnson"}] to the load-balancer
[Thread-15] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"2",
 "city":"Los Angeles","owner_name":"Jane Smith"}] to the load-balancer
[Thread-17] INFO ee.pw.microservice.worker.WorkerTask - Sending response object with value=[{"id":"3",
```

5. Podsumowanie

Load-Balancer jest to znakomita usługa sieciowa, która w dobry sposób realizuje nam etap Mapowania w metodyce PCAM. W przypadku load-balancerów ważny jest dobry wybór algorytmu równoważącego obciążenia. W naszym przypadku wybraliśmy algorytm mapowania cyklicznego oraz najmniejszych połączeń. W przypadku algorytmu o najmniejszych połączeniach, ma to według nas sens, żeby implementować to na większą skalę. W przypadku algorytmu mapowania cyklicznego nie jesteśmy pewni do dobrego zastosowania tego algorytmu – jest to na pewno jakiś pomysł, ale wydaje nam się, że w większej ilości przypadków nie byłby efektywnym sposobem na równoważnie obciążenia. W projekcie użyliśmy również języka Java – język Java w łatwy sposób poprzez pakiet java.net pozwolił nam korzystać z gniazdek oraz przekazywania danych pomiędzy procesami (programami Javy). Sposób pisania tego projektu był bardzo przyjemny, dzięki interfejsom, jaki dostarczał język Java. Podsumowując, napisanie projektu load-balancer było bardzo ciekawym doświadczeniem oraz bardzo uczącym pod względem pisania równoległych programów.