

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

Сергій СТИРЕНКО

(підпис)

“ ” _____ 2022 р.

**Дипломний проєкт
на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного
забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”**

на тему: Спосіб балансування даних

Виконав : студент 4 курсу, групи ІІІ-83
(шифр групи)

Іванченко В’ячеслав Віталійович

(прізвище, ім’я, по батькові)

(підпис)

Керівник асистент Волокита І. М.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Консультант (нормоконтроль) професор, д.т.н., Сімоненко В. П.

(назва розділу)

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

(підпис)

Рецензент _____

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2022 р.

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ

Завідувач кафедри

Сергій СТИРЕНКО

(підпис)

“ ____ ” _____ 2022 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студента

Іванченка В’ячеслава Віталійовича

1. Тема проєкту Спосіб балансування даних
керівник проєкту Волокита Іван Миколайович, асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2022 року №1139-с
2. Термін здачі студентом закінченого проєкту 11 червня 2022 р.
3. Вихідні дані до проєкту технічна документація, теоретичні дані.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Розділ 1. Опис предметної області
Розділ 2. Алгоритми програми та використані технології
Розділ 3. Проектування системи.
Розділ 4. Тестування розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В. П.		

7. Дата видачі завдання «30» серпня 2021 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	<i>Затвердження теми проєкту</i>	<i>10.12.2021-15.12.2021</i>	
2.	<i>Вивчення та аналіз завдання</i>	<i>15.12.2021-15.03.2022</i>	
3.	<i>Розробка архітектури та загальної структури системи</i>	<i>15.03.2022-25.03.2022</i>	
4.	<i>Розробка структур окремих підсистем</i>	<i>25.03.2022-5.04.2022</i>	
5.	<i>Програмна реалізація системи</i>	<i>5.04.2022-15.04.2022</i>	
6.	<i>Оформлення пояснювальної записки</i>	<i>21.04.2022-20.05.2022</i>	
7.	<i>Захист програмного продукту</i>		
8.	<i>Передзахист</i>		
9.	<i>Захист</i>		

Студент-дипломник _____ В'ячеслав ІВАНЧЕНКО
(підпис)

Керівник проєкту _____ Іван ВОЛОКИТА
(підпис)

АНОТАЦІЯ

У даній роботі було детально розглянуто один із видів балансування даних, а саме балансування навантаження. Було описано принцип роботи балансеру, його допоміжний функціонал, а також дев'ять існуючих алгоритмів балансування. Програма була реалізована на мові програмування Javascript, а саме на платформі Node.js. Також було проведено тестування балансеру, де, залежно від користувацьких налаштувань, було продемонстровано різні варіанти його роботи.

Ключові слова: балансування навантаження, балансер, пул серверів, трафік, Node.js.

ANNOTATION

In this project, one of the types of data balancing was considered in detail, namely load balancing. The principle of operation of the balancer, its auxiliary functionality, as well as nine existing balancing algorithms were described. The program was implemented in the Javascript programming language, namely the Node.js platform. The balancer was also tested, where, depending on the user settings, different variants of its operation were demonstrated.

Keywords: load balancing, balancer, server pool, traffic, Node.js.

**ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ**

на тему: «Спосіб балансування даних»

Київ – 2022

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ	2
2 ПІДСТАВИ ДЛЯ РОЗРОБКИ.....	2
3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ.....	2
4 ДЖЕРЕЛА РОЗРОБКИ.....	2
5 ТЕХНІЧНІ ВИМОГИ	3
5.1 Вимоги до розробленого продукту.....	3
5.2 Вимоги до програмного забезпечення	3
5.3 Вимоги до апаратної частини	3
6 ЕТАПИ РОЗРОБКИ	3

					ІАЛЦ.467200.002 ТЗ								
		№ докум.	Підпис	Дата									
Розробив		Іванченко В. В.			Спосіб балансування даних Технічне завдання				Літ.	Аркуш	Аркушів		
Перевірив		Волокита І. М.									1	3	
									НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ПІ-83				
Н. Контр.		Сімоненко В. П.											
Затвердив													

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку прикладного балансувальника навантаження, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є розробка програмного забезпечення на основі масштабування, а також забезпечення надійності доставки та безпеки даних.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка прикладного балансувальника навантаження, що дозволить ефективно вирішувати задачу підвищення працездатності системи, забезпечення надійності доставки та безпеки даних.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

					ІАЛЦ.467200.002 ТЗ	Арк.
						2
Зм.	Арк.	№ докум.	Підпис	Дата		

5 ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Балансувати різні види мережевого трафіку.
- Надати можливість користувачам обирати алгоритм балансування.
- Надати можливість користувачам налаштовувати параметри безпеки.
- За відповідних налаштувань забезпечувати надійність доставки даних.
- Забезпечувати хорошу пропускну здатність та обробку помилок.

5.2 Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- WebStorm IDE.

5.3 Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core i3-5005U.
- ROM не менше ніж 4 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2021-15.12.2021
Вивчення та аналіз завдання	15.12.2021-15.03.2022
Розробка архітектури та загальної структури системи	15.03.2022-25.03.2022
Розробка структур окремих частин системи	25.03.2022-5.04.2022
Програмна реалізація системи	5.04.2022-15.04.2022
Виправлення помилок	15.04.2022-20.04.2022
Оформлення пояснювальної записки	21.04.2022-25.05.2022
Передзахист	
Захист	

					ІАЛЦ.467200.002 ТЗ	Арк.
						3
Зм.	Арк.	№ докум.	Підпис	Дата		

**ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ**

на тему: «Спосіб балансування даних»

Київ – 2022

ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1. ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ.....	9
1.1 Балансування навантаження	9
1.1.1 Поняття балансування навантаження.....	9
1.1.2 Статичні та динамічні алгоритми балансування.....	9
1.1.3 Апаратні та програмні балансувальники навантаження	13
1.1.4 Роль балансувальника навантаження	14
1.1.5 Історія балансування навантаження	15
1.1.6 Балансування навантаження та SSL	15
1.1.7 Балансування навантаження та безпека	16
1.1.8 Переваги балансування навантаження	17
1.1.9 Балансувальники навантаження L4, L7 і GSLB.....	17
1.2 Існуючі балансувальники навантаження.....	19
1.2.1 HAProxy	19
1.2.2 Elastic Load Balancing	21
1.2.3 F5 і NGINX.....	23
1.2.4 Loadbalancer.org	27
1.2.5 Citrix	28
1.2.6 Порівняльна таблиця балансувальників навантаження.....	31
ВИСНОВКИ ДО РОЗДІЛУ 1.....	33

РОЗДІЛ 2. АЛГОРИТМИ ПРОГРАМИ ТА ВИКОРИСТАНІ ТЕХНОЛОГІЇ ...	34
2.1 Алгоритми балансування	34
2.1.1 Round Robin.....	34
2.1.2 Weighted Round Robin	35
2.1.3 Least Connection	37
2.1.4 Weighted Least Connection	39
2.1.5 Resource Based (Adaptive).....	41
2.1.6 Weighted Response Time	42
2.1.7 Source IP hash or Sticky Session	44
2.1.8 URL Hash.....	46
2.1.9 Random	47
2.1.10 Порівняльна таблиця алгоритмів	48
2.2 Використані технології	50
2.2.1 JavaScript	50
2.2.2 Node.js	53
2.2.3 HTTP/HTTPS	54
2.2.4 WebSocket/WebSocket Secure	56
ВИСНОВОК ДО РОЗДІЛУ 2.....	60
РОЗДІЛ 3. ПРОЕКТУВАННЯ СИСТЕМИ	61
3.1. Структура програми	61
3.1.1 Базова концепція балансувальника навантаження	61
3.1.2 Черга	62
3.1.3 Контролер.....	65

3.1.4 Безпека	66
3.1.5 Кешування.....	69
3.1.6 Логування.....	71
3.1.7 Точка входу програми	72
3.2 Черга.....	74
3.2.1 ServerPool	74
3.2.2 Round Robin.....	78
3.2.3 Weighted Round Robin	80
3.2.3 Least Connection	81
3.2.4 Weighted Least Connection	83
3.2.5 Weighted Response Time	83
3.2.6 IP Hash та Url Hash	84
3.2.7 Random	86
3.2.8 Resource based(Adaptive)	87
3.3 Контролер	89
3.1.2 BaseContoller	89
3.1.2 HttpController	89
3.1.3 WsController	93
ВИСНОВКИ ДО РОЗДІЛУ 3.....	99
РОЗДІЛ 4. ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ.....	100
4.1 Компоненти.....	100
4.1.1 Контролер.....	100
4.1.2 Перевірка стану серверів.....	105

4.1.3 Безпека	109
4.1.4 Кешування.....	112
4.1.5 Логування.....	114
4.2 Алгоритми балансування	114
4.2.1 Round Robin.....	114
4.2.2 Weighted Round Robin	115
4.2.3 Least Connection	116
4.2.4 Weighted Least Connection	117
4.2.5 Weighted Response Time	118
4.2.6 Resource based(Adaptive)	119
4.2.7 IP Hash та UrlHash	120
4.2.7 Random	121
ВИСНОВКИ ДО РОЗДІЛУ 4.....	123
ВИСНОВКИ	124
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	125

ВСТУП

Сучасне програмне забезпечення складається з безлічі окремих компонентів, робота яких постійно потребує в надходженні нової інформації. Саме тому для того щоб система функціонувала як єдине ціле виникає потреба у балансуванні даних між її частинами.

Це можна спостерігати в мережі Інтернет, нейронних мережах, базах даних, хмарних середовищах, при роботі Web застосунків та багато іншого. Кожна з наведених систем має величезну кількість компонентів, кожен з яких виконує певну частину роботи.

Одним із видів балансування даних є балансування навантаження, при якому проміжне програмне забезпечення розподіляє трафік, що прямує до кінцевих точок. Ця система має широкий попит в сучасному програмуванні, адже кожен сервіс потребує у підвищенні продуктивності та зменшенні частоти виходу програми з ладу. Балансувальник дозволяє розгорнути кілька дублікатів програми, стаючи посередником між ними та віддаленими точками, які звертаються до системи.

Також варто звернути увагу на те, що існує певна кількість алгоритмів балансування навантаження. Під час її роботи, в залежності від внутрішньої інструкції, проходить аналіз інформації про стан кінцевих точок, який дає змогу виявити найкращий варіант для вирішення наступної задачі.

При цьому варто пам'ятати, що вибір алгоритму балансування залежить від потреб системи. Кожен з них має свої переваги та недоліки та може мати обмеженні області застосування.

В наступних розділах більш детально розглянуто роботу балансувальника.

					ІАЛЦ.467200.003 ПЗ	Арк.
						8
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1

ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Балансування навантаження

1.1.1 Поняття балансування навантаження

Балансування навантаження — це апаратний пристрій або програмна платформа, яка запобігає перевантаженню окремих серверів. Вони покращують доступність додатків і швидкість реагування, а також запобігають перевантаженню сервера.

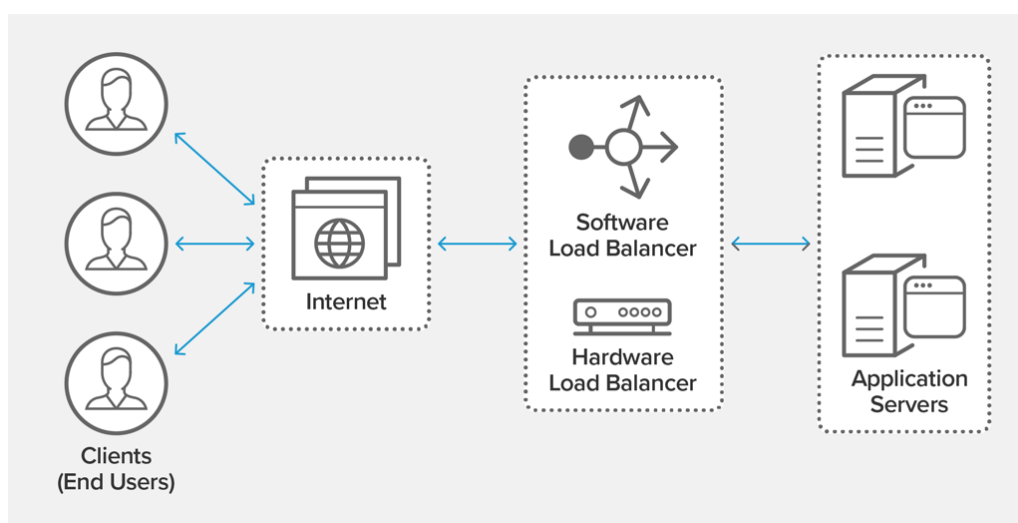


Рис. 1.1 Принцип роботи прикладного сервісу з використанням балансування навантаження

Кожен балансувальник навантаження знаходиться між клієнтськими пристроями і серверами, отримуючи, а потім розподіляючи вхідні запити на будь-який доступний сервер, здатний їх виконати. Сучасні широкі сервіси не можуть працювати без балансування навантаження.

1.1.2 Статичні та динамічні алгоритми балансування

Балансувальники навантаження використовують алгоритми для розподілу мережевого трафіку між серверами. Алгоритм — це логіка або набір попередньо визначених правил, які використовуються для сортування, а

алгоритми балансування навантаження можуть бути статичними або динамічними[1].

Методи статичного балансування навантаження розподіляють трафік без коригування поточного стану системи чи серверів. Деякі статичні алгоритми надсилають однакові обсяги трафіку або в заданому порядку, або у випадковому порядку на кожен сервер у групі. Алгоритми динамічного балансування навантаження враховують поточний стан системи та кожного сервера та розподіл базового трафіку за цими факторами.

Алгоритм статичного балансування навантаження не враховує стан системи, оскільки він розподіляє завдання. Натомість розподіл формується на основі припущень і відомостей про загальну систему, зроблених до початку сортування. Сюди входять такі відомості, як кількість процесорів, швидкість зв'язку та потужність, а також такі припущення, як потреби в ресурсах, час відгуку та час надходження вхідних завдань.

Алгоритми статичного балансування навантаження в розподілених системах мінімізують конкретні функції продуктивності, пов'язуючи відомий набір завдань з доступними процесорами. Ці типи стратегій балансування навантаження зазвичай зосереджуються навколо маршрутизатора, який оптимізує функцію продуктивності та розподіляє навантаження. Перевагою статичного балансування навантаження в розподілених системах є простота використання та швидке й просте розгортання, хоча є деякі ситуації, які не найкращим чином обслуговуються таким алгоритмом.

Динамічні алгоритми враховують поточне навантаження кожного вузла або обчислювального блоку в системі, досягаючи швидшої обробки за рахунок динамічного переміщення завдань від перевантажених вузлів до вузлів, які недостатньо завантажені. Динамічні алгоритми набагато складніше розробляти, але особливо коли час виконання для різних завдань сильно різниться, вони можуть дати чудові результати. Крім того, оскільки немає необхідності

					ІАЛЦ.467200.003 ПЗ	Арк.
						10
Зм.	Арк.	№ докум.	Підпис	Дата		

виділяти певні вузли для розподілу роботи, архітектура динамічного балансування навантаження часто є більш модульною.

Унікальне призначення завдань включає завдання, які однозначно призначаються процесору на основі його стану в певний момент. Динамічне призначення відноситься до постійного перерозподілу завдань на основі стану системи та її розвитку. Очевидно, що будь-який алгоритм балансування навантаження може фактично сповільнити загальний процес, якщо він вимагає надмірного обчислення для прийняття рішень.

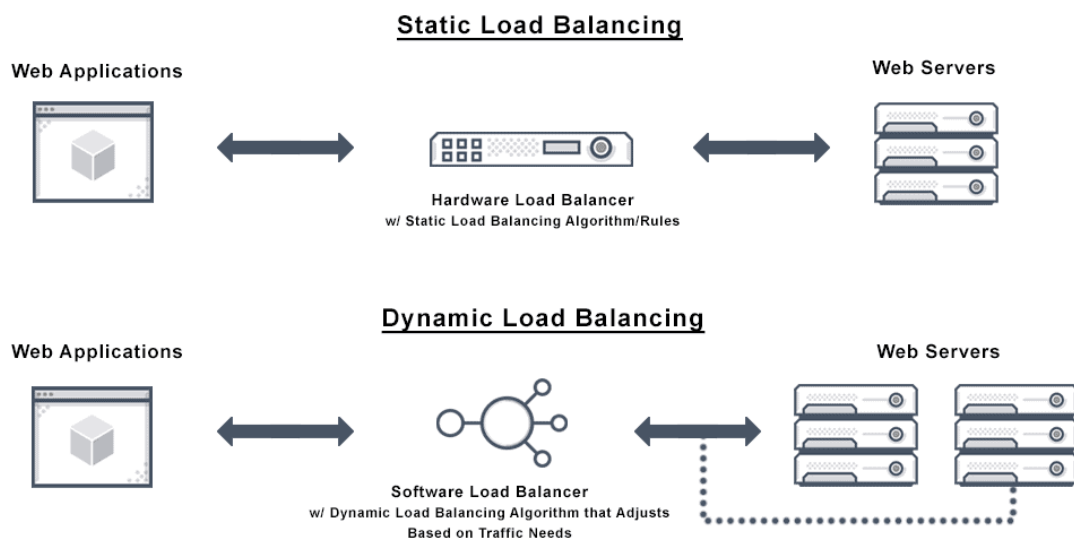


Рис. 1.2 Статичне та динамічне балансування[3]

І методи динамічного, і статичного балансування навантаження також визначаються іншими факторами:

- **Характер завдань.** Характер завдань має великий вплив на ефективність алгоритмів балансування навантаження, тому максимальний доступ до інформації про завдання під час прийняття рішень алгоритмом збільшує потенціал оптимізації.
- **Розмір завдання.** Точне значення часу виконання завдання зустрічається вкрай рідко, але дозволить оптимально розподілити навантаження. Існує кілька способів оцінити різний час виконання. Якщо завдання однакового розміру, середній час виконання може використовуватися успішно.

Однак, коли час виконання дуже нерегулярний, потрібні більш складні методи. Наприклад, можна додавати метадані до завдань, а потім робити висновки для майбутніх завдань на основі статистичних даних на основі попереднього часу виконання для подібних метаданих.

- Залежності. Завдання можуть залежати одне від одного, і деякі не можуть почати, поки інші не будуть завершені. Проілюструйте такі взаємозалежності за допомогою орієнтованого ациклічного графіка та мінімізуйте загальний час виконання, оптимізуючи порядок завдань. Деякі алгоритми можуть використовувати метаввристичні методи для обчислення оптимального розподілу завдань.
- Розмежування завдань. Це відноситься до здатності задач розбиватися на підзадачі під час виконання, оскільки ця специфіка важлива при розробці алгоритмів балансування навантаження.

Апаратна архітектура серед паралельних блоків [2]:

- Неоднорідність. Блоки різної обчислювальної потужності часто містять паралельні обчислювальні інфраструктури, і розподіл навантаження має враховувати цю зміну. Наприклад, блоки з меншою обчислювальною потужністю повинні отримувати запити, які вимагають менше обчислень або менше запитів однорідного чи невідомого розміру, ніж більші блоки.
- Пам'ять. Паралельні блоки часто поділяють на категорії спільної та розподіленої пам'яті. Спільні блоки пам'яті відповідають моделі PRAM, усі спільно використовують, читають і записують паралельно в одній загальній пам'яті. Розподілені блоки пам'яті відповідають моделі розподіленої пам'яті, кожен з яких обмінюється інформацією за допомогою повідомлень і має свою власну пам'ять. У будь-якого типу є переваги, але лише деякі системи підпадають під одну чи іншу категорію. Загалом, алгоритми балансування навантаження повинні бути адаптовані

					ІАЛЦ.467200.003 ПЗ	Арк.
						12
Зм.	Арк.	№ докум.	Підпис	Дата		

спеціально до паралельної архітектури, щоб уникнути зниження ефективності паралельного вирішення проблем.

- Ієрархія. Два основних типи алгоритмів балансування навантаження — контролер-агент і розподілене керування. У моделі контролер-агент блок управління призначає завдання агентам, які виконують завдання та інформують контролера про прогрес. У разі динамічних алгоритмів контролер може призначати або перепризначати завдання. Коли контроль розподіляється між вузлами, вузли розподіляють відповідальність за призначення завдань, а алгоритм балансування навантаження.

1.1.3 Апаратні та програмні балансувальники навантаження

Апаратні балансувальники навантаження[3]:

- Зазвичай це високопродуктивні пристрої, здатні безпечно обробляти кілька гігабіт трафіку з різних типів програм.
- Ці пристрої також можуть містити вбудовані можливості віртуалізації, які консоліднують численні віртуальні екземпляри балансування навантаження на тому самому обладнанні.
- Забезпечують більш гнучкі архітектури з багатьма орендарями та повну ізоляцію орендарів, серед інших переваг.

Програмні балансувальники навантаження:

- Можуть повністю замінити обладнання для балансування навантаження, забезпечуючи аналогічну функціональність і чудову гнучкість.
- Можуть працювати на звичайних гіпервізорах, у контейнерах або як процеси Linux з мінімальними накладними витратами на голих серверах.
- Вони легко налаштовуються залежно від випадків використання та технічних вимог, про які йдеться.
- Може заощадити місце та зменшити витрати на обладнання.

					ІАЛЦ.467200.003 ПЗ	Арк.
						13
Зм.	Арк.	№ докум.	Підпис	Дата		

Балансувальники навантаження визначають справність серверних ресурсів і не надсилають трафік на сервери, які не можуть виконувати запити. Незалежно від того, чи це апаратне чи програмне забезпечення, чи який(и) алгоритм(и) він використовує, балансувальник навантаження розподіляє трафік на різні веб-сервери в пулі ресурсів, щоб гарантувати, що жоден сервер не стане перевантаженим і, згодом, ненадійним. Це ефективно мінімізує час відповіді сервера та максимізує пропускну здатність.

1.1.4 Роль балансувальника навантаження

Роль балансувальника навантаження має на меті систематично направляти запити в потрібні місця в будь-який момент, запобігаючи тим самим дорогим вузьким місцям і непередбачуваним інцидентам. Балансувальники навантаження повинні в кінцевому підсумку забезпечувати продуктивність і безпеку, необхідні для підтримки складних ІТ-середовища, а також складних робочих процесів, що відбуваються в них[4].

Балансування навантаження є найбільш масштабованою методологією для обробки безлічі запитів із сучасних робочих процесів із багатьма програмами та різними пристроями. У тандемі з платформами, які забезпечують безперешкодний доступ до численних програм і настільних комп'ютерів у сучасних цифрових робочих просторах, балансування навантаження підтримує більш послідовний і надійний досвід роботи кінцевого користувача для співробітників.

Балансувальники навантаження керують потоком інформації між сервером і кінцевим пристроєм (ПК, ноутбук, планшет або смартфон). Сервер може бути локальним, у центрі обробки даних або у загальнодоступній хмарі. Сервер також може бути фізичним або віртуалізованим. Балансувальник навантаження допомагає серверам ефективно переміщувати дані, оптимізує використання ресурсів доставки додатків і запобігає перевантаженню сервера. Балансувальники навантаження проводять постійні перевірки працездатності

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		14

серверів, щоб переконатися, що вони можуть обробляти запити. При необхідності балансувальник навантаження видаляє непрацездатні сервери з пулу, доки вони не будуть відновлені. Деякі балансувальники навантаження навіть запускають створення нових віртуалізованих серверів додатків, щоб впоратися з підвищеним попитом.

1.1.5 Історія балансування навантаження

Балансування навантаження почалося в 1990-х роках як апаратні пристрої, що розподіляють трафік по мережі. Розробники хотіли покращити доступність додатків, що працюють на серверах. Згодом балансування навантаження взяло на себе більше обов'язків із появою контролерів доставки додатків (ADC). Вони забезпечують безпеку разом із безперешкодним доступом до програм у час пік[5].

ADC поділяються на три категорії: апаратні пристрої, віртуальні пристрої (по суті, програмне забезпечення, витягнуте із застарілого апаратного забезпечення) і балансувальники навантаження на основі програмного забезпечення. Оскільки обчислення переміщуються в хмару, програмні ADC виконують подібні завдання, що й апаратні. Вони також мають додаткову функціональність і гнучкість. Вони дозволяють організації швидко та безпечно розширювати свої прикладні послуги на основі попиту в хмарі. Сучасні ADC дозволяють організаціям консолідувати мережеві послуги. Ці послуги включають розвантаження SSL/TLS, кешування, стиснення, виявлення вторгнень і брандмауери веб-додатків. Це створює ще коротші терміни доставки та більшу масштабованість.

1.1.6 Балансування навантаження та SSL

Рівень безпечних сокетів (SSL) — це стандартна технологія безпеки для встановлення зашифрованого зв'язку між веб-сервером і браузером[6]. Трафік SSL часто розшифровується на балансірі навантаження. Коли балансувальник

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		15

навантаження розшифровує трафік перед передачею запиту, це називається завершенням SSL. Балансувальник навантаження позбавляє веб-сервери від необхідності витрачати додаткові цикли ЦП, які необхідні для дешифрування. Це покращує продуктивність програми.

Однак припинення SSL супроводжується проблемою безпеки. Трафік між балансувальниками навантаження та веб-серверами більше не шифрується. Це може піддати програму можливим нападам. Однак ризик зменшується, якщо балансувальник навантаження знаходиться в одному центрі обробки даних, що й веб-сервери.

Іншим рішенням є SSL pass-through. Балансувальник навантаження просто передає зашифрований запит на веб-сервер. Потім веб-сервер виконує розшифровку. Це використовує більше потужності ЦП на веб-сервері. Але сервіси, які потребують додаткової безпеки, можуть вважати, що додаткові витрати цього вартують[7].

1.1.7 Балансування навантаження та безпека

Балансування навантаження відіграє важливу роль у безпеці, оскільки обчислення постійно переміщуються в хмару. Функція розвантаження балансувальника навантаження захищає організацію від розподілених атак відмови в обслуговуванні (DDoS). Це робиться шляхом перенесення трафіку атаки з корпоративного сервера на постачальника загальнодоступної хмари. DDoS-атаки становлять значну частину кіберзлочинності, оскільки їх кількість і розмір продовжують зростати. Апаратний захист, такий як брандмауер периметра, може бути дорогим і вимагати значного обслуговування. Програмні балансери навантаження з хмарним розвантаженням забезпечують ефективний та економічно ефективний захист[8].

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		16

1.1.8 Переваги балансування навантаження

- Зменшений час простою
- Масштабований
- Надмірність
- Гнучкість
- Ефективність

Балансування навантаження може робити більше, ніж просто діяти як мережевий трафік. Балансувальники програмного навантаження забезпечують такі переваги, як прогнозна аналітика, яка визначає вузькі місця трафіку до того, як вони виникнуть. В результаті балансувальник програмного навантаження дає додатку корисні відомості. Вони є ключовими для автоматизації та можуть допомогти приймати бізнес-рішення.

1.1.9 Балансувальники навантаження L4, L7 і GSLB

Повсякденна поведінка сервісу в цифровому робочому просторі може бути дуже різним. Його продуктивність визначається усім: від заходів безпеки в їхніх облікових записах до різної продуктивності багатьох програм, які вони використовують — проблема, яка може погіршитися через погане реагування при неналежному балансуванні навантаження.

Іншими словами, цифрові робочі простори в значній мірі керуються додатками. Оскільки одночасний попит на додатки програмного забезпечення продовжують зростати, надійна доставка їх кінцевим користувачам може стати проблемою, якщо не буде належного балансування навантаження. Сервіси, яким уже важко орієнтуватися в кількох системах, інтерфейсах та вимогах безпеки, будуть нести додатковий тягар уповільнення продуктивності та перебоїв.

Щоб забезпечити більшу узгодженість і не відставати від постійно зростаючого попиту користувачів, ресурси сервера повинні бути легко

доступними та збалансованими навантаженням на рівнях 4 та/або 7 моделі взаємозв'язку відкритих систем (OSI)[9]:

- Балансувальники навантаження рівня 4 (L4) працюють на транспортному рівні. Це означає, що вони можуть приймати рішення щодо маршрутизації на основі портів TCP або UDP, які використовують пакети разом із IP-адресами джерела та призначення. Балансувальники навантаження L4 виконують трансляцію мережевих адрес (NAT), але не перевіряють фактичний вміст кожного пакету.
- Балансувальники навантаження рівня 7 (L7) діють на рівні програми, найвищому в моделі OSI. Вони можуть оцінювати ширший діапазон даних, включаючи заголовки HTTP та ідентифікатори сеансів SSL, коли вирішують, як розподілити запити по фермі серверів.

Балансування навантаження є більш інтенсивним на рівні L7, ніж L4, але воно також може бути ефективнішим на L7 через додатковий контекст у розумінні та обробці клієнтських запитів до серверів. На додаток до базового балансування навантаження L4 і L7, існує глобальне балансування навантаження сервера (GSLB), що може розширити можливості будь-якого типу в кількох центрах обробки даних, щоб великі обсяги трафіку можна було ефективно розподіляти без погіршення якості обслуговування для кінцевого користувача[10].

Оскільки програми все частіше розміщуються в хмарних центрах обробки даних, розташованих у різних регіонах, GSLB дає можливість ІТ-організаціям доставляти додатки з більшою надійністю та меншою затримкою на будь-який пристрій або місце. Це забезпечує більш узгоджений досвід для кінцевих користувачів, коли вони користуються кількома додатками та службами в цифровому робочому просторі.

					ІАЛЦ.467200.003 ПЗ	Арк.
						18
Зм.	Арк.	№ докум.	Підпис	Дата		

1.2 Існуючі балансувальники навантаження

1.2.1 HAProxy

HAProxy — це високопродуктивний балансувальник навантаження з відкритим вихідним кодом і зворотний проксі-сервер для додатків TCP і HTTP. Користувачі можуть використовувати HAProxy для підвищення продуктивності веб-сайтів і програм, розподіляючи свої робочі навантаження. Покращення продуктивності включають мінімізований час відповіді та збільшення пропускної здатності. HAProxy використовується в сервісах із високим трафіком, таких як GitHub і Twitter.



Рис. 1.3 Логотип HAProxy[11]

Балансувальники навантаження можна використовувати для розподілу робочих навантажень між комп'ютерами, мережами, дисковими накопичувачами або центральними процесорами. HAProxy, що входить до багатьох дистрибутивів Linux, є одним із провідних стандартів балансування програмного навантаження. Хоча HAProxy є відкритим вихідним кодом, комерційний варіант доступний через HAProxy Technologies, який називається HAProxy Enterprise. HAProxy Enterprise включає корпоративний набір доповнень, експертну підтримку та професійні послуги[11].

Особливості:

- Функції HAProxy включають такі послуги, як:
- Балансування навантаження рівнів 4 і 7 (TCP і HTTP відповідно)

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		19

- Підтримка протоколів HTTP, HTTP/2, gRPC, FastCGI
- Припинення SSL/TLS
- Динамічне зберігання сертифікатів SSL
- Перемикання вмісту та перевірка
- Прозоре проксі
- Детальна реєстрація
- CLI для керування сервером
- HTTP аутентифікація
- Багатопотоковість
- Переписування URL-адрес
- Розширена перевірка здоров'я
- Rate limiting

Адміністративні інструменти, включаючи API середовища виконання, DNS, API площини даних і шаблони серверів

Інструменти повільного запуску та зупинки для детального контролю над трафіком та доступом користувачів

Функція перемикання вмісту дозволить користувачам вибирати пули серверів за запитом. Крім того, налаштування прозорого проксі-сервера дозволить користувачам підключатися до сервера безпосередньо за допомогою IP-адреси клієнта. Перевірка вмісту дозволить користувачам блокувати будь-які неочікувані протоколи. Користувачі можуть використовувати його детальний журнал для перегляду даних запитів у реальному часі. CLI дозволить користувачам вносити зміни, наприклад вмикати або вимикати сервер. HAProxy також масштабується для підключення до тисяч серверів[12].

Безпека

Користувачі вважають HAProxy безпечним, оскільки протягом багатьох років він мав дуже мало вразливостей. Він містить функції, які обмежують поверхню атаки у разі проблем із безпекою, наприклад, ізоляції себе за

					ІАЛЦ.467200.003 ПЗ	Арк.
						20
Зм.	Арк.	№ докум.	Підпис	Дата		

допомогою chroot, перехід до непривілейованого користувача/групи під час запуску та уникнення доступу до диска після ініціалізації. HAProxy також можна використовувати в областях, щоб зробити інші системи більш безпечними. Наприклад, HAProxy може перевіряти трафік і контролювати поведінку клієнта за серією запитів, а потім блокувати клієнта, якщо вони здаються шкідливими. Користувачі можуть налаштувати списки контролю доступу, які визначають політику, коли дозволяти доступ на основі метаданих, знайдених у запиті. Він також дозволяє обмежувати швидкість і вносити IP-адреси в чорний/білий список.

Плюси та мінуси HAProxy

Плюси

- Гнучкість за допомогою інструментів для балансування навантаження, моніторингу, безпеки та перезапису
- Основа Linux економить на експлуатаційних витратах
- Легко налаштувати та впровадити у виробниче середовище

Мінуси

- Документація може бути складною та важкою для аналізу
- Як рішення на базі Linux, має простий інтерфейс користувача та меншу внутрішню підтримку
- Підтримка лише текстових файлів для керування конфігурацією

1.2.2 Elastic Load Balancing

Amazon Web Services (AWS) залишається лідером у хмарній інфраструктурі та послугах, завдяки чому рішення Elastic Load Balancing (ELB) стає простим вибором для існуючих клієнтів. Рішення AWS включають класичний балансер для екземплярів додатків і три додаткові для шлюзів (IP), мереж (TCP, UDP, TLS) і програм (HTTP/S, gRPC).

					ІАЛЦ.467200.003 ПЗ	Арк.
						21
Зм.	Арк.	№ докум.	Підпис	Дата		

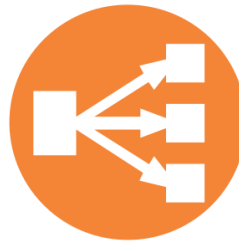


Рис. 1.4 Логотип Elastic Load Balancing[12]

Для потреб клієнта в трафіку рівня 7 Amazon має Application Load Balancer, фіксовані відповіді, режим пом'якшення десинхронізації та маршрутизацію на основі заголовків HTTP. Ви можете вибрати тип балансира навантаження, який найкраще відповідає вашим потребам[7].

Elastic Load Balancing автоматично розподіляє ваш вхідний трафік між кількома цілями, такими як екземпляри EC2, контейнери та IP-адреси, в одній або кількох зонах доступності. Він відстежує стан своїх зареєстрованих цілей і направляє трафік лише до здорових цілей. Elastic Load Balancing масштабує ваш балансувальник навантаження, оскільки ваш вхідний трафік змінюється з часом. Він може автоматично масштабуватися до переважної більшості робочих навантажень.

Особливості

- Реєстрація та моніторинг із метриками запитів, помилок, затримок тощо
- Закріплені сеанси, через які направляються запити між цільовими клієнтами та серверами
- Контролер Kubernetes, який пропонує прямий доступ до pod і підтримку приватних кластерів EKS
- Елементи керування конфігурацією для зняття підключень, міжзонного LB та дозволів доступу
- Можливості безпеки, такі як шифрування внутрішнього сервера та ідентифікація імені сервера

Плюси та мінуси AWS

Плюси

					ІАЛЦ.467200.003 ПЗ	Арк.
						22
Зм.	Арк.	№ докум.	Підпис	Дата		

- Простота інтеграції та використання для адміністраторів із мінімалістичним дизайном
- Гнучкість у виборі підбраного рішення на основі потреб клієнта
- Високодоступний і надійний з параметрами автоматичного масштабування трафіку

Мінуси

- Немає локальної опції для AWS ELB
- Відсутність розвантаження або реконфігурації SSL для тайм-аутів неактивного підключення
- Classic LB пропонує базові можливості, що мають затримку

1.2.3 F5 і NGINX

F5 вже давно відомий своїм апаратним забезпеченням, програмним забезпеченням і віртуальними приладами BIG-IP. Після придбання компанією продукту NGINX у 2019 році лідерська позиція F5 на ринку балансування навантаження не викликає сумнівів. Продукти балансування навантаження обох постачальників залишаються доступними окремо з F5 BIG-IP Local Traffic Manager (LTM) і NGINX Plus[14].



Рис. 1.5 Логотип F5[13]

BIG-IP LTM пропонує можливості керування трафіком додатків, вхід в контейнери, налаштовану автоматизацію та масштабовану інфраструктуру, необхідну для корпоративних IT-серед. Крім того, NGINX Plus був більш популярним із двох рішень, об'єднавши шлюз API, кешування, WAF та веб-сервер у надійне рішення для балансування навантаження.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		23

NGINX



Рис. 1.6 Логотип NGINX[13]

Вимовляється як «engine-ex», є веб-сервером з відкритим кодом, який з моменту свого початкового успіху як веб-сервер тепер також використовується як зворотний проксі-сервер та балансувальник навантаження.

Відомі компанії, які використовують Nginx, включають Autodesk, Atlassian, Intuit, T-Mobile, GitLab, DuckDuckGo, Microsoft, IBM, Google, Adobe, Salesforce, VMWare, Xerox, LinkedIn, Cisco, Facebook, Target, Citrix Systems, Twitter, Apple, Intel та багато інших[15].

Спочатку Nginx був створений Ігорем Сисоєвим, з його першим публічним випуском у жовтні 2004 року. Спочатку Ігор задумав програмне забезпечення як відповідь на проблему C10k, яка є проблемою щодо продуктивності обробки 10 000 одночасних підключень.

NGINX як веб-сервер

Метою NGINX було створення найшвидшого веб-сервера, і підтримка цієї досконалості все ще залишається центральною метою проекту. NGINX постійно перевершує Apache та інші сервери в тестах для вимірювання продуктивності веб-сервера. Однак після вихідного випуску NGINX веб-сайти розширилися від простих HTML-сторінок до динамічного багатогранного вмісту. NGINX виріс разом з ним і тепер підтримує всі компоненти сучасної мережі, включаючи WebSocket, HTTP/2, gRPC і потокове передавання кількох відеоформатів (HDS, HLS, RTMP та інші).

NGINX за межами веб-сервісу

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		24

Хоча NGINX став відомим як найшвидший веб-сервер, масштабована базова архітектура виявилася ідеальною для багатьох веб-завдань, крім обслуговування вмісту. Оскільки він може обробляти великий обсяг з'єднань, NGINX зазвичай використовується як зворотний проксі-сервер і балансувальник навантаження для керування вхідним трафіком і розподілу його на повільніші сервери вище по потоку – від застарілих серверів баз даних до мікросервісів.

NGINX також часто розміщується між клієнтами та другим веб-сервером, щоб служити термінатором SSL/TLS або веб-прискорювачем. Виступаючи як посередник, NGINX ефективно справляється із завданнями, які можуть уповільнити роботу вашого веб-сервера, наприклад, узгодження SSL/TLS або стиснення та кешування вмісту для підвищення продуктивності. Динамічні сайти, створені з використанням будь-чого, від Node.js до PHP, зазвичай розгортають NGINX як кеш вмісту та зворотний проксі, щоб зменшити навантаження на сервери додатків і максимально ефективно використовувати базове обладнання[14].

F5 Функції диспетчера локального трафіку BIG-IP (LTM).

- Оптимізація доставки за допомогою кешування RAM і симетричного адаптивного стиснення
- Видимість адміністратора за допомогою журналів, показників продуктивності та аналітики
- Активна кластеризація програм і масштабованість на вимогу
- Моніторинг працездатності, керування станом і балансування навантаження для трафіку програм
- Можливості програмованої інфраструктури з керуванням SOAP і REST API

Функції NGINX Plus

					ІАЛЦ.467200.003 ПЗ	Арк.
						25
Зм.	Арк.	№ докум.	Підпис	Дата		

- Підтримка хмарних (AWS, Azure, VMware) і контейнерних середовищ (Docker, K8s)
- Підтримка балансування навантаження для HTTP, TCP та UDP
- Варіанти автентифікації включають HTTP, NTLM, JWT, OpenID Connect і SSO
- WebSockets
- Підтримка сценаріїв і програмування для JS, Lua, Ansible, Chef і Puppet
- Режими високої доступності, синхронізація конфігурації та збереження фіксованого сеансу
- Зворотний проксі з кешуванням
- IPv6
- Підтримка FastCGI з кешуванням
- Обробка статичних файлів, файлів індексу та автоматичного індексування
- TLS/SSL з SNI

Плюси та мінуси F5 та NGINX

Плюси

- Дуже швидкий порівняно з іншими балансирами навантаження
- Невелике використання пам'яті та стабільний
- Похвала за високу продуктивність відносно вартості

Мінуси

- Відсутність форумів підтримки спільноти та документації
- Конфігурація та налаштування можуть бути складними для менш досвідчених адміністраторів
- Обмежена документація щодо характеристик і параметрів продукту

					ІАЛЦ.467200.003 ПЗ	Арк.
						26
Зм.	Арк.	№ докум.	Підпис	Дата		

1.2.4 Loadbalancer.org

Однотименний Loadbalancer.org пропонує клієнтам широкий вибір корпоративних варіантів між апаратними, віртуальними та хмарними рішеннями АЦП. Маючи на вибір шість моделей, компанія надає апаратне обладнання з однією стійкою (1U) для необмеженої кількості серверів і прогресивних рівнів максимальної пропускної здатності, ключів SSL TPS, одночасних з'єднань рівня 7 та максимальної кількості з'єднань.



Рис. 1.7 Логотип Loadbalancer.org[14]

Хмарні та віртуальні ADC рішення Loadbalancer.org мають гнучку структуру ціноутворення, як-от платіжна, безстрокова, підписка та ліцензія на сайт. Балансувальники навантаження британського постачальника підтримують три великі постачальники хмарної інфраструктури: AWS, Azure та Google Cloud[16].

Функції Loadbalancer.org

- Віртуальне балансування навантаження з необмеженою масштабованістю, пропускною здатністю та транзакціями SSL
- Керування конфігурацією та автоматизація для маршрутизації вмісту
- Функції безпеки, включаючи інтегрований WAF, віртуальне виправлення та зворотний проксі
- Високопродуктивна пряма маршрутизація та балансування навантаження сервера для будь-якого протоколу TCP/UDP

					ІАЛЦ.467200.003 ПЗ	Арк.
						27
Зм.	Арк.	№ докум.	Підпис	Дата		

- Підтримка прискорення та розвантаження SSL, а також автоматизованого ланцюжка сертифікатів SSL
- Моніторинг здоров'я
- Попередньо визначені протоколи
- Перевірка надмірності
- Зворотний проксі
- Планувальники
- Стиснення даних
- Автоматична конфігурація
- Кешування вмісту

Плюси та мінуси

Плюси

- Висока продуктивність і надійність з мінімальними або відсутністю простоїв
- Простота впровадження та доступність і якість технічної підтримки
- Багатофункціональний і гнучкий для продуктивності балансування навантаження

Мінуси

- Забезпечення якості та документація може бути корисною для покращення
- Ціни вищі порівняно з іншими галузевими виборами
- Використання енергії на пристроях та вплив на продуктивність

1.2.5 Citrix

У 2005 році корпоративний IT-постачальник Citrix вийшов на ринок балансування навантаження, придбавши компанію NetScaler, що займається прискоренням мережевого трафіку. Citrix ADC можна розгорнути разом із

					ІАЛЦ.467200.003 ПЗ	Арк.
						28
Зм.	Арк.	№ докум.	Підпис	Дата		

монолітними програмами та програмами на основі мікросервісів як уніфіковану базу коду на платформах гібридного середовища.



Рис. 1.8 Логотип Loadbalancer.org[15]

Citrix допомагає клієнтам оптимізувати доставку додатків за допомогою функцій для стиснення вмісту, зображень, інтерфейсу та TCP, а також інтегрованих функцій кешування. За допомогою Citrix Application Delivery Management (ADM) адміністратори можуть централізовано керувати політиками та звітами щодо здоров'я додатків, аналітики безпеки та моніторингу базової активності на основі ML[17].

Функція балансування навантаження розподіляє клієнтські запити між кількома серверами для оптимізації використання ресурсів. У реальному світі з обмеженою кількістю серверів, які надають обслуговування великій кількості клієнтів, сервер може стати перевантаженим і погіршити продуктивність ферми серверів. Пристрій Citrix ADC використовує критерії балансування навантаження, щоб запобігти вузьким місцям, пересилаючи кожен запит клієнта на сервер, який найкраще підходить для обробки запиту, коли він надходить.

Щоб налаштувати балансування навантаження, ви визначаєте віртуальний сервер для проксі кількох серверів у фермі серверів і балансуєте навантаження між ними.

Коли клієнт ініціює з'єднання з сервером, віртуальний сервер розриває клієнтське з'єднання та ініціює нове з'єднання з вибраним сервером або повторно використовує існуюче з'єднання з сервером для балансування навантаження. Функція балансування навантаження забезпечує керування трафіком від рівня 4 (TCP і UDP) до рівня 7 (FTP, HTTP і HTTPS).

Пристрій Citrix ADC використовує ряд алгоритмів, які називаються методами балансування навантаження, щоб визначити, як розподілити навантаження між серверами. Методом балансування навантаження за замовчуванням є метод найменших підключень.

Сутності функціонують наступним чином[18]:

- Віртуальний сервер. Представлена IP-адресою, портом і протоколом. IP-адреса віртуального сервера (VIP) зазвичай є загальнодоступною IP-адресою. Клієнт надсилає запити на підключення на цю IP-адресу. Віртуальний сервер являє собою банк серверів.
- Обслуговування. Логічне представлення сервера або програми, що працює на сервері. Визначає IP-адресу сервера, порт і протокол. Служби прив'язані до віртуальних серверів.
- Об'єкт сервера. Суб'єкт, який представлений IP-адресою. Об'єкт сервера створюється під час створення служби. За ім'я об'єкта сервера приймається IP-адреса сервісу. Ви також можете створити об'єкт сервера, а потім створити служби за допомогою об'єкта сервера.
- Монітор. Об'єкт, який відстежує працездатність послуг. Пристрій періодично перевіряє сервери за допомогою монітора, прив'язаного до кожної служби. Якщо сервер не відповідає протягом зазначеного часу очікування відповіді, а зазначена кількість запитів виходить з ладу, служба позначається ВНИЗ. Потім пристрій виконує балансування навантаження серед інших служб.

Особливості ADC Citrix

- Інструменти фронт-енд оптимізації для макета вмісту, оптимізації JS та розподілу домену
- Протоколи динамічної маршрутизації, захист від перенапруг і GSLP для доступності додатків
- Діюча аналітика та конструктор візуальної політики через Citrix ADM

					ІАЛЦ.467200.003 ПЗ	Арк.
						30
Зм.	Арк.	№ докум.	Підпис	Дата		

- Захист від DoS для L4-L7 і L7 рівнів
- Такі функції шлюзу, як аналіз кінцевих точок, проксі-сервер RDP без стану та підтримка PCoIP

Плюси та мінуси Citrix

Плюси

- Висока доступність і простота керування конфігурацією
- Можливість і гнучкість оновлювати пристрої балансування навантаження
- Інтеграція з віртуальними програмами та настільними комп'ютерами Citrix

Мінуси

- Надмірна залежність від підтримки спільноти для вирішення проблем
- Крута крива навчання і складний інтерфейс користувача
- Керування патчами мікропрограми може бути складним і вимагає додаткової допомоги

1.2.6 Порівняльна таблиця балансувальників навантаження

Таблиця 1.1 – Порівняння існуючих балансувальників навантаження

	GS LB	Integrat ed WAF	DDoS Prote ction	Analyti cs	Physic al	Cloud	Virtua l	Contain er
AWS	-	-	-	-	-	+	+	+
Citrix	+	+	+	+	+	+	+	+
F5 NGINX	-	+	+	+	+	+	+	+
HAProxy	-	+	+	-	-	-	-	-
Load balancer	+	+	+	-	+	+	+	-

Як ми можемо бачити з таблиці, майже всі представленні балансувальники мають в собі опції як програмного, так і апаратного балансування, окрім AWS ELB та Nginx. Тобто важливо звернути на це увагу, адже обраний балансувальник може не відповідати вашим потребам. Також Nginx, на відміну від інших балансувальників, не дає можливості керування трафіком у хмарі чи контейнері. Всі балансувальники мають хороший рівень безпеки (DDoS protection та Firewall), окрім AWS ELB. При цьому він має кращу пропускну здатність, адже не витрачає цикли ЦП на захист від кібератак. Можна сказати, що Nginx є чудовим рішенням для невеликого проекту, на відміну від інших описаних балансувальників. На мою думку, найкращим варіантом по ціні та якості є Nginx, з яким я мав змогу особисто працювати. В залежності від особливостей вашої системи ви можете налаштувати всі етапи балансування.

					ІАЛЦ.467200.003 ПЗ	Арк.
						32
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ ДО РОЗДІЛУ 1.

В межах даного розділу було розглянуто предметну область балансування навантаження. Перш за все це базове поняття балансувальника, його види, функції та призначення. Можна сказати, що переважна більшість сервісів, для підтримки працездатності, потребує у балансуванні навантаження. Балансувальник виконує роль адміністратора, який закриває вузькі місця сервісу та відповідає за його безпеку.

Також в цьому розділі описані найбільш популярні існуючі балансувальники. Завдяки порівняльній таблиці було зроблено висновок про те, де краще використовувати вибрані сервіси, в залежності від вимог та об'єму вашої системи.

Окрім цього, ми звернули на увагу на інтернет протоколи HTTP/s та WS/s, адже саме вони встановлюють зв'язки(односторонній та двосторонній), які використовують майже всі існуючі системи. Розуміння роботи цих протоколів потрібне нам для того, щоб реалізувати балансувальник прикладного типу(7 рівень OSI).

На останок були проаналізовані існуючі балансувальники навантаження. При цьому було з'ясовано, що кожне з рішень має свої переваги та недоліки в залежності від сфери використання. Таким чином, актуальною є задача розробки балансувальника навантаження, який є гнучким до потреб користувача.

					ІАЛЦ.467200.003 ПЗ	Арк.
						33
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2

АЛГОРИТМИ ПРОГРАМИ ТА ВИКОРИСТАНІ ТЕХНОЛОГІЇ

2.1 Алгоритми балансування

2.1.1 Round Robin

Алгоритм перебирає по черзі всі сервери, що є у пулі балансування. При цьому вузол може отримати запит, навіть якщо він завантажений.

Простий у реалізації та концептуалізації, round robin є найбільш поширеним алгоритмом балансування навантаження. За допомогою цього методу запити клієнтів циклічно маршрутизуються на доступні сервери. Повне розподілення навантаження сервера найкраще працює, коли сервери мають приблизно однакові обчислювальні можливості та ємність зберігання[19].

Коротше кажучи, кругова система балансування мережевого навантаження обертає запити на з'єднання між веб-серверами в порядку отримання запитів. Для спрощеного прикладу припустимо, що підприємство має пул із трьох серверів: Сервер А, Сервер В і Сервер С.

- Перший запит надсилається на сервер А.
- Другий запит надсилається на сервер В.
- Третій запит надсилається на сервер С.

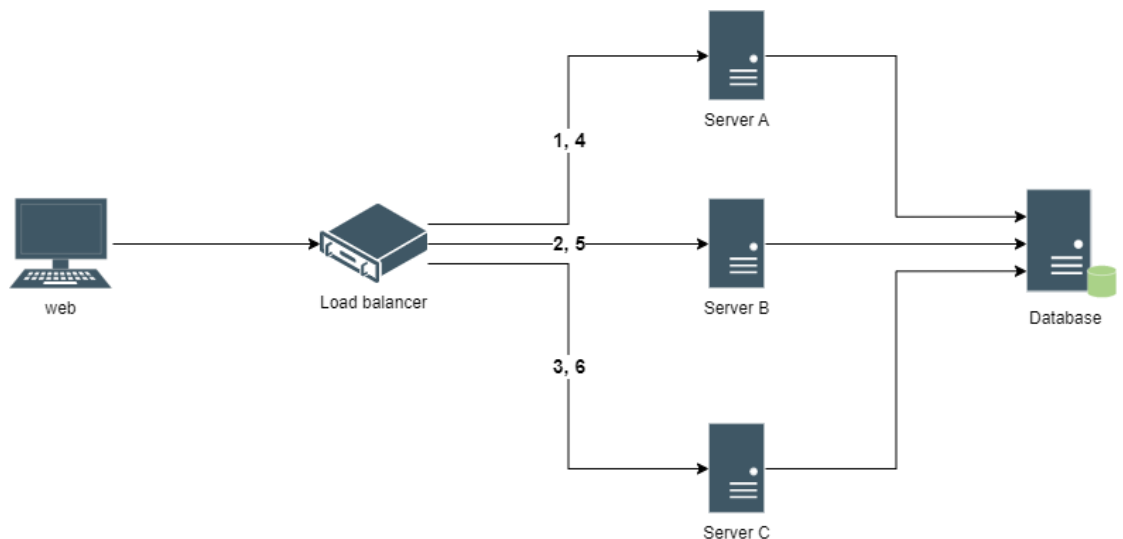


Рис. 2.1 Принцип роботи Round Robin

Переваги:

- Простий у реалізації і високо практичний(підтримує будь який тип з'єднання)
- Процес планування є мало затратним
- Навантаження на сервер розподіляється рівномірно для обробки великого трафіку, тобто кожен сервер отримує однакову частку трафіку

Недоліки:

- алгоритм передбачає, що сервери достатньо подібні, щоб обробляти еквівалентні навантаження. Якщо певні сервери мають більше ЦП, оперативної пам'яті чи інших специфікацій, алгоритм не має можливості розподілити більше запитів на ці сервери. В результаті сервери з меншою ємністю можуть перевантажуватися та швидше виходити з ладу, поки ємність на інших серверах не працює.

2.1.2 Weighted Round Robin

Удосконалена версія алгоритму Round Robin. Суть удосконалень полягає в наступному: кожному серверу присвоюється ваговий коефіцієнт відповідно

до його продуктивності та потужності. Це допомагає розподіляти навантаження гнучкіше: Сервери з більшою вагою отримують більшу частку запитів клієнтів[19].

Для спрощеного прикладу припустимо, що підприємство має пул із трьох серверів:

- Сервер А може обробляти в середньому 15 запитів на секунду
- Сервер В може обробляти в середньому 10 запитів на секунду
- Сервер С може обробляти в середньому 5 запитів на секунду

Далі припустимо, що балансувальник навантаження отримує 6 запитів.

- 3 запити надіслано на сервер А
- 2 запити надіслано на сервер В
- 1 запит надіслано на сервер С.

Таким чином, зважений цикловий алгоритм розподіляє навантаження відповідно до потужності кожного сервера.

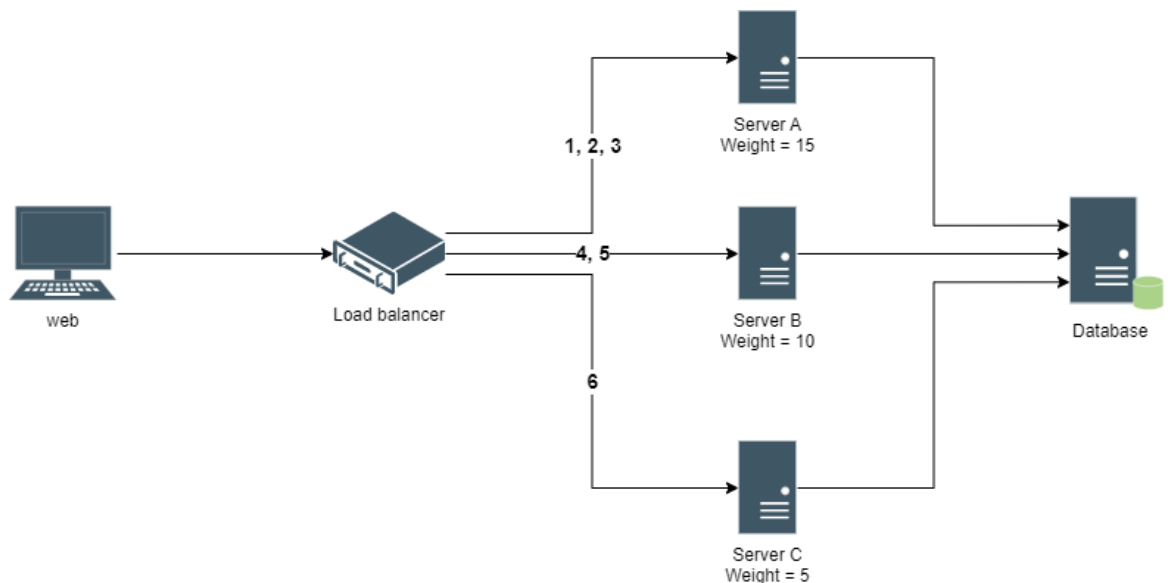


Рис. 2.2 Принцип роботи Weighted Round Robin

Переваги:

- Простий у реалізації і високо практичний(підтримує будь який тип з'єднання)

- Процес планування є мало затратним
- Навантаження на сервер розподіляється відповідно до його ваги. Більш потужний сервер обробляє більше запитів.
- Це алгоритм планування без стану, який не записує статус усіх підключень(всі ваги є статичними).

Недоліки:

- Є відносно простим, але не підходить для сценаріїв, коли час обслуговування запиту значно змінюється, або кожен запит займає різну кількість часу. У цих сценаріях циклове планування може призвести до дисбалансу розподілу навантаження між серверами.
- Негнучкий до критичних ситуацій(наприклад змінення потужності серверу).

2.1.3 Least Connection

Алгоритм динамічного балансування навантаження, при якому клієнтські запити розподіляються на сервер додатків з найменшою кількістю активних з'єднань на момент отримання клієнтського запиту. У випадках, коли сервери додатків мають подібні специфікації, сервер додатків може бути перевантажений через більш тривалі з'єднання; цей алгоритм враховує активне навантаження підключення[19].

Припустимо, що у нас є шість запитів, підключених до серверів за балансувальником навантаження.

Сервер А керує запитамі 1, 3, 5, другий сервер В керує запитамі 2, 4, 6, а третій сервер С керує запитамі 7, 8, 9. Але через певний проміжок часу запити 1, 2, 3 виконуються.

Тепер сервер А керує запитом 5, сервер В керує запитамі 4, 6, а сервер С керує такою ж кількістю запитів, тобто 7, 8, 9.

Якщо надходить новий запит (скажімо, 10), його слід переслати на сервер А балансувальником навантаження, оскільки він має найменшу кількість активних з'єднань.

Тепер сервер А керує двома запитами 5, 10, сервер В також керує двома запитами 4, 6, а сервер С керує такою ж кількістю запитів 7, 8, 9.

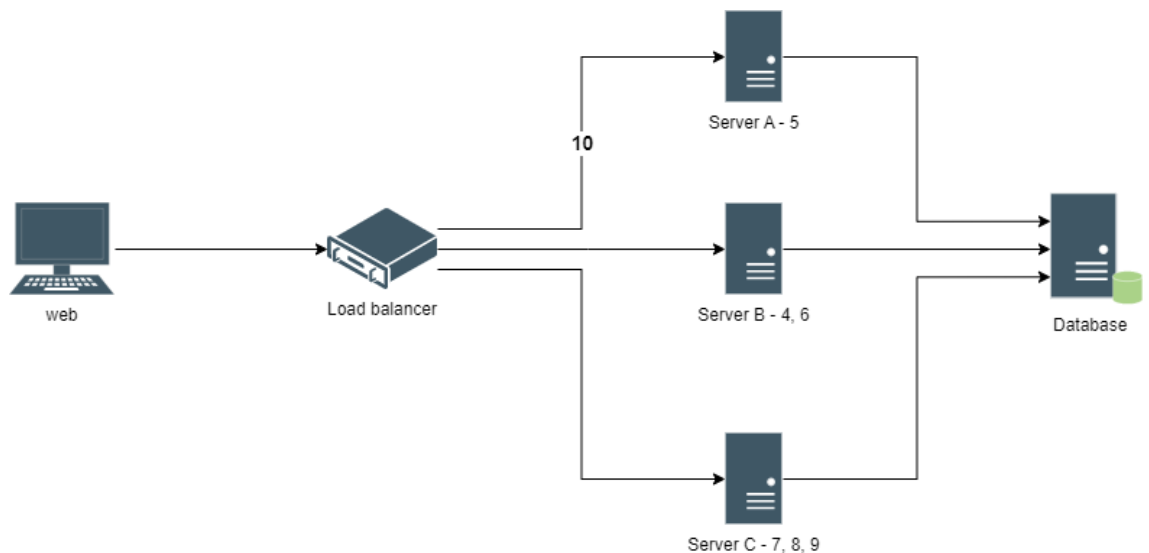


Рис. 2.3 Принцип роботи Least Connection

Переваги:

- Техніка балансування навантаження найменших з'єднань мінімізує ймовірність перевантаження сервера, оскільки вона надсилає запити лише на сервери з меншою кількістю активних з'єднань. Це на відміну від методів кругового або зваженого циклічного балансування навантаження, які надсилають запити на всі сервери по черзі.
- цей алгоритм чудово підходить для запитів, які потребують тривалої обробки, наприклад FTP чи WebSockets(окрім TCP).
- Як і приведені вище алгоритми є відносно мало затратним.

Недоліки:

- Алгоритм не є достатньо практичним. Тобто використання даного алгоритму є залежним від того скільки запитів одночасно може приймати

сервер(адмін). Якщо у нас кластерна система серверів, то даний алгоритм підходить для будь якого типу з'єднань. Якщо наш сервер-адмін в момент часу може обробляти лише один запит, то алгоритм підходить лише для довготривалих з'єднань(для http запитів він буде працювати як звичайний циклічний алгоритм при більших витратах).

- Погано працює з TCP з'єднаннями. TCP TIME_WAIT зазвичай становить 2 хвилини, між якими сервер отримує запитів на зв'язок.
- Не враховує ємність сервера при визначенні кількості поточних з'єднань, тобто) Сервер А має 10 з'єднань, а Сервер В має 20. Наступний запит буде надіслано до Сервера А, оскільки він має менше, але сервер А може мати 12 з'єднань, у той час як сервер В має 50, що робить сервер А більш імовірним перевантаженням.

2.1.4 Weighted Least Connection

Алгоритм, що базується на алгоритмі least connection, щоб врахувати різні характеристики сервера додатків. Адміністратор призначає вагу кожному серверу додатків на основі критеріїв за своїм вибором, щоб продемонструвати здатність серверів додатків обробляти трафік. Сервер-адмін створює критерії балансування навантаження на основі активних підключень і ваги серверів додатків[19].

Припустимо, що існує n вузлів, кожен вузол має вагу W_i ($i=1, \dots, n$) активні з'єднання C_i ($i=1, \dots, n$).

Тоді наступне мережеве з'єднання буде спрямовано на вузол де $\min(\frac{C_i}{W_i})$.

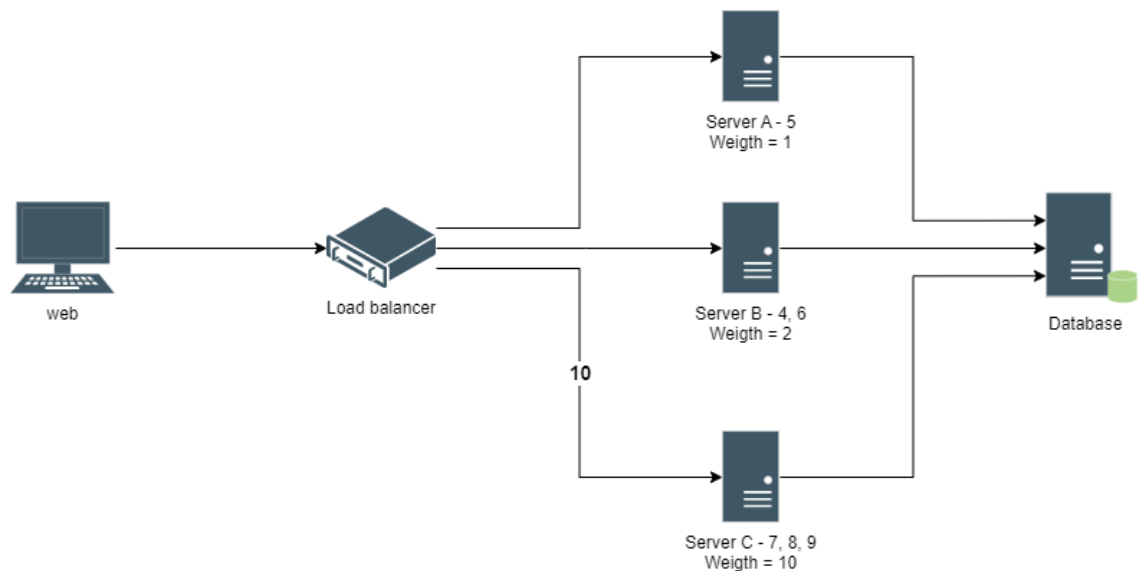


Рис. 2.4 Принцип роботи Weighted Least Connection

Переваги:

- Цей алгоритм чудово підходить для запитів, які потребують тривалої обробки, наприклад FTP, TCP чи WebSockets.
- Оскільки враховуються ваги серверів, то мінімізуються їхні перенавантаження. Більш потужні сервери будуть тримати більше підключень.

Недоліки:

- Алгоритм не є достатньо практичним. Тобто використання даного алгоритму є залежним від того скільки запитів одночасно може приймати сервер(адмін). Якщо у нас кластерна система серверів, то даний алгоритм підходить для будь якого типу з'єднань. Якщо наш сервер-адмін в момент часу може обробляти лише один запит, то алгоритм підходить лише для довготривалих з'єднань(для http запитів він буде працювати як звичайний циклічний алгоритм при більших витратах).

2.1.5 Resource Based (Adaptive)

Адаптивний метод, який регулярно перевіряє сервер, незалежно від його ваги, щоб планувати трафік у режимі реального часу. Для кожного серверу з нашого пулу зберігається інформація про нього (об'єм трафіку що пройшов через нього, час віддачі, кількість запитів та багато іншого). Інформація оновлюється після кожного запиту. Саме на основі даних про сервер під час кожного нового запиту розраховується його навантаження, що дозволяє обрати найкращий варіант для даної задачі.

Для розрахунку навантаження чудово підходить метод, що базується на основі рухомого середнього значення.

The Exponentially Weighted Moving Average (EWMA) - це кількісний або статистичний показник, який використовується для моделювання або опису часового ряду. EWMA розроблено таким чином, що давнішим спостереженням надається менша вага. Вагові коефіцієнти знижуються експоненціально в міру старіння точки даних – звідси й назва експоненціально зважена.

Єдине рішення, яке має прийняти користувач EWMA, це параметр альфа. Параметр визначає, наскільки важливим є поточне спостереження для обчислення EWMA. Чим вище значення альфа, тим точніше EWMA відстежує оригінальний часовий ряд.

Формула для розрахунку EWMA значення[20]:

$$EWMA(t) = \alpha * r(t) + (1 - \alpha) * EWMA(t - 1)$$

Переваги:

- У порівнянні зі звичайним алгоритмом обчислюваного середнього значення, EWMA не потребує збереження всіх минулих значень, що значно зменшує обчислення, а також ресурси зберігання.
- Традиційні обчислювальні алгоритми усереднення нечутливі до споживання часу мережі, в той час як EWMA може регулювати β за

допомогою частих запитів, щоб швидко відстежувати діри мережі або краще відображати загальне середнє значення.

- Коли запити стають більш частими, це означає, що навантаження на мережу вузла зростає, і ми хочемо відстежувати час, який вузол витрачає на обробку запитів (що відображає навантаження вузла), відповідно зменшуємо β . Чим менше β , тим ближче значення EWMA до поточного споживання часу, і тоді ми можемо швидко відстежувати діри мережі. Коли запити рідше, ми коригуємо значення β відносно більше. Таким чином, обчислене значення EWMA ближче до середнього значення.

Недоліки:

- Алгоритм є порівняно енергозатратним при визначенні наступного вузлу для обробки запиту.

2.1.6 Weighted Response Time

Алгоритм балансування навантаження, коли час відповіді серверів додатків визначає, який сервер додатків отримує наступний запит. Час відповіді сервера додатків на перевірку працездатності використовується для обчислення ваг сервера програм. Сервер додатків, який відповідає найшвидше, отримує наступний запит[21].

Припустимо, що в нас є 3 сервери А, В, С. Після перших запитів на перевірку працездатності ми визначаємо часи відповідей $t_a = 1$, $t_b = 2$, $t_c = 3$. Далі всі наступні клієнтські запити будуть іти на сервер А.

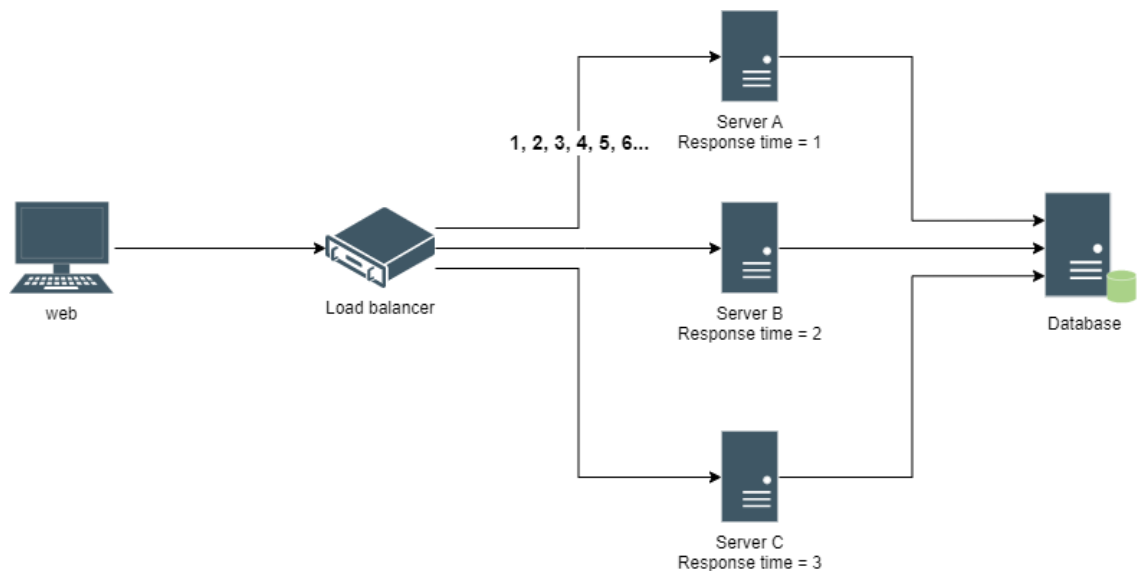


Рис. 2.5 Принцип роботи Weighted Response Time(1 фаза)

Однак сервер А перенавантажився і час відповіді при перевірці працездатності вийшов 4. Після цього всі наступні запити будуть йти на сервер В. І так далі.

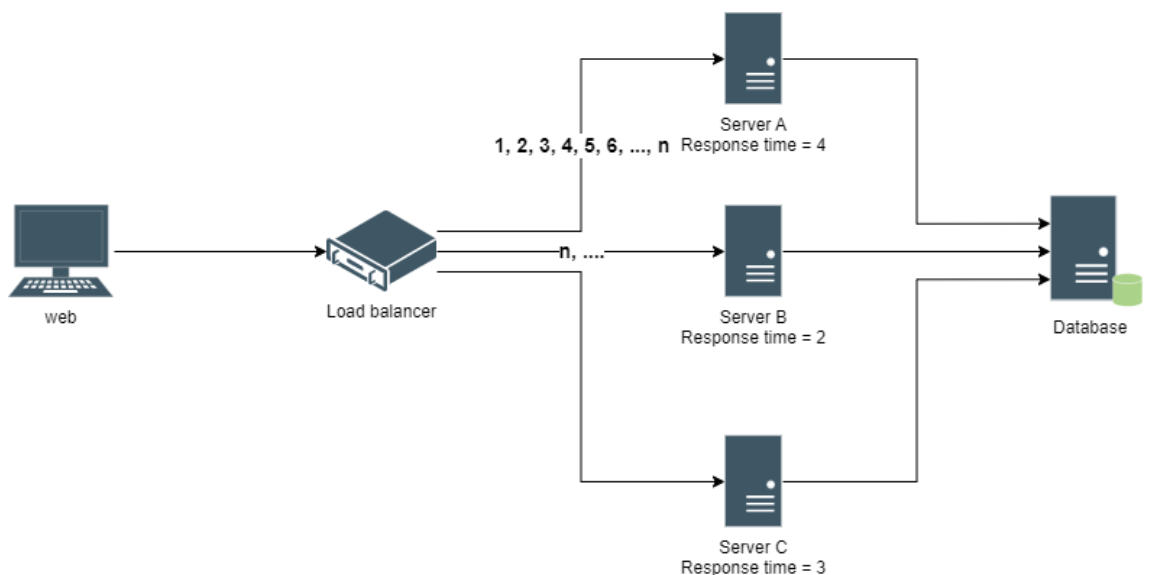


Рис. 2.6 Принцип роботи Weighted Response Time(2 фаза)

Переваги:

- Алгоритм дає змогу клієнтам отримувати відповідь якомога швидше. Всі запити будуть надсилатися до найшвидшого серверу(до певного моменту).

- Вага кожного серверу з часом оновлюється(не є статичною), що дає змогу рівномірно розподіляти запити(в залежності від завантаженості) і мінімізувати перенавантаження(як тільки сервер почне відповідати довго, запит піде на інший сервер).

Недоліки:

- Є достатньо енергозатратним(потребує постійного оновлення ваг для серверів з даного пулу).

2.1.7 Source IP hash or Sticky Session

Алгоритм балансування хеш-навантаження вихідного IP-адреси, який поєднує IP-адреси джерела та призначення клієнта та сервера для створення унікального хеш-ключа. Ключ використовується для виділення клієнта на певний сервер. Оскільки ключ можна відновити, якщо сеанс зламаний, запит клієнта спрямовується на той самий сервер, який він використовував раніше. Це корисно, якщо важливо, щоб клієнт підключився до сеансу, який все ще активний після відключення[22].

Припустимо, що в нас є 2 клієнта X і Y та 3 сервери в пулі A, B, C. Перший клієнт X підправляє запит і за ним закріплюється сервер A. Запис про вузол для даного клієнта буде виглядати приблизно так:

{ip_client_x: server backend1.example.com}

Так само відбувається прив'язка клієнта Y до серверу B. З часом сервер A падає і клієнт X прив'язується до серверу C, який є працює в штатному режимі. Після того як сервер A знову починає працювати, клієнт X знову починає відправляти запити саме на нього(змінюється прив'язка).

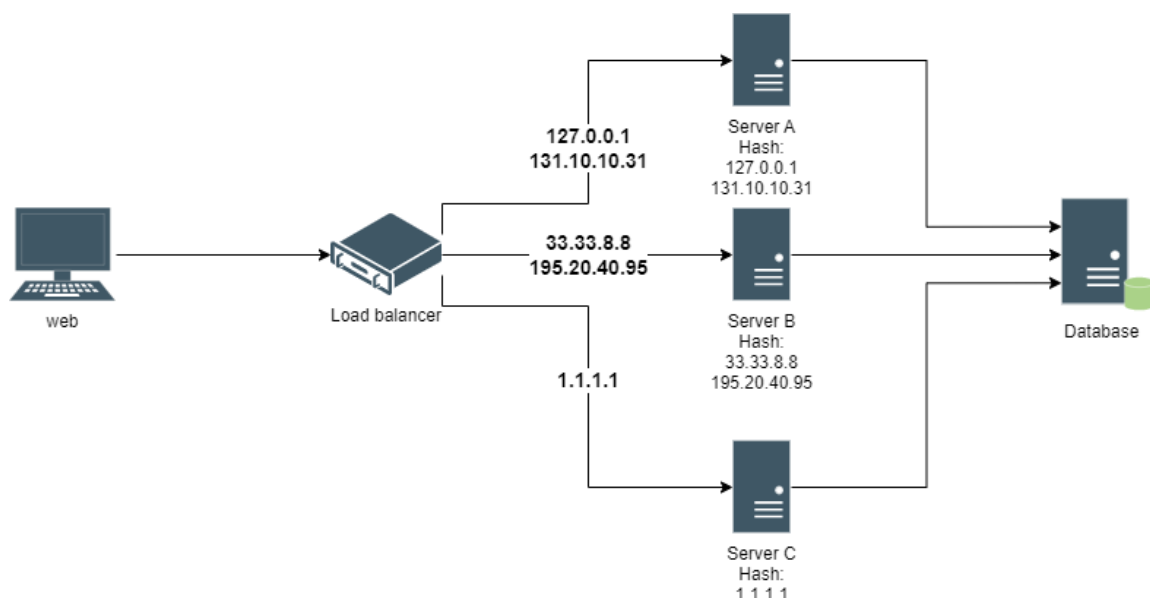


Рис. 2.7 Принцип роботи IP hash

Переваги:

- Мінімізований обмін даними. При використанні фіксованих сеансів серверам у вашій мережі не потрібно обмінюватися даними сеансу, що є дорогим процесом, якщо він виконується в масштабі.
- Використання кешу оперативної пам'яті – фіксовані сеанси дозволяють більш ефективно використовувати кеш оперативної пам'яті вашої програми, що призводить до кращого реагування.

Недоліки:

- Можуть виникнути проблеми з прив'язкою сесії, якщо клієнт використовує динамічний IP. У ситуації, коли велике число запитів проходить через один проксі-сервер, балансування врят чи можна назвати ефективним та справедливим. Описані проблеми, однак, можна вирішити, використовувати cookies.
- Збій вузла призведе до перерозподілу хеш-ключів, через що впаде швидкість.

2.1.8 URL Hash

Алгоритм балансування навантаження для рівномірного розподілу записів між кількома сайтами та надсилання всіх прочитань на сайт, який володіє об'єктом.

Коли віртуальний сервер балансування навантаження налаштований на використання методу хешування URL-адреси, хеш-значення генерується з URL-адреси HTTP, присутньої в запиті, а потім пересилається до вибраної служби. Усі майбутні з'єднання з використанням ідентичної URL-адреси будуть переправлені в ту саму службу. Це корисно в ситуаціях, коли кінцеві користувачі мають кешовані дані, до яких їм потрібно отримати доступ, наприклад, кошик для покупок сайту. Якщо їх направляють до служби без цих кешованих даних, вони можуть бути дуже розчаровані.

За замовчуванням балансувальник навантаження використовує перші 80 байтів URL-адреси для хеш-значення або всю URL-адресу, якщо вона менше 80 байт. Ви можете вказати будь-яку довжину хешування від 1 до 4096. Більша довжина хеша корисна для більш рівномірного балансування навантаження, якщо ви використовуєте довші URL-адреси, де відрізняється лише невелика кількість символів[22].

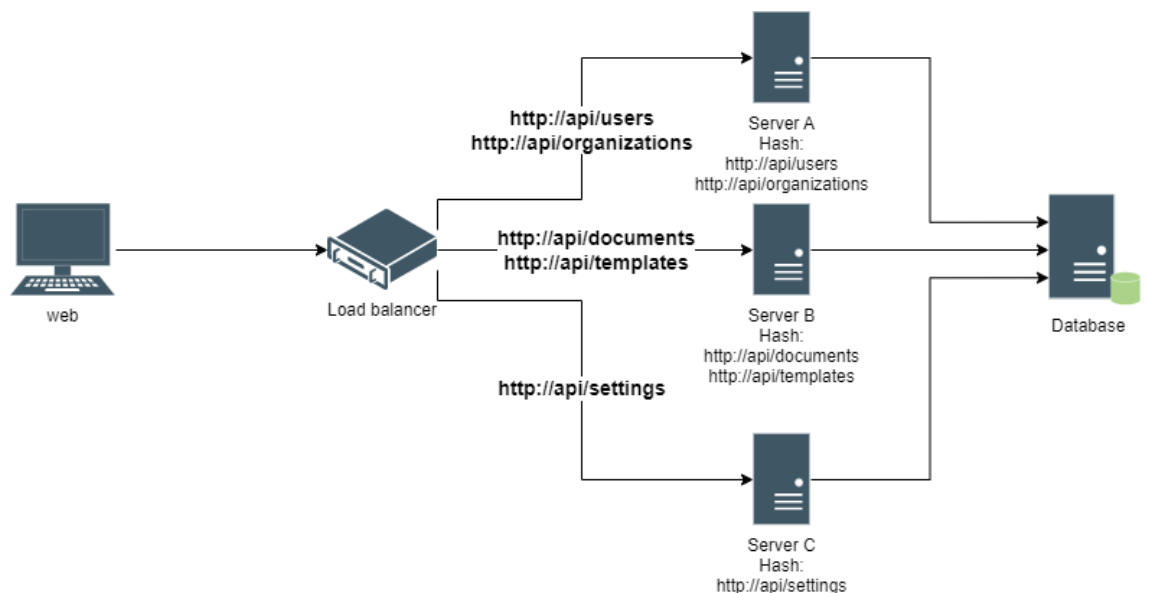


Рис. 2.8 Принцип роботи URL Hash

Переваги:

- один і той же ключ повинен потрапляти на один і той же вузол, тому його можна використовувати в сценаріях кешу із записом і читанням.

Недоліки:

- Збій вузла призведе до перерозподілу хеш-ключів, через що впаде швидкість.

2.1.9 Random

Як впливає з його назви, цей алгоритм визначає клієнти та сервери випадковим чином, тобто за допомогою базового генератора випадкових чисел. У випадках, коли балансувальник навантаження отримує велику кількість запитів, випадковий алгоритм зможе розподілити запити рівномірно між вузлами. Таким чином, як і Round Robin, алгоритм достатній для серверів, що мають однакові потужності[23].

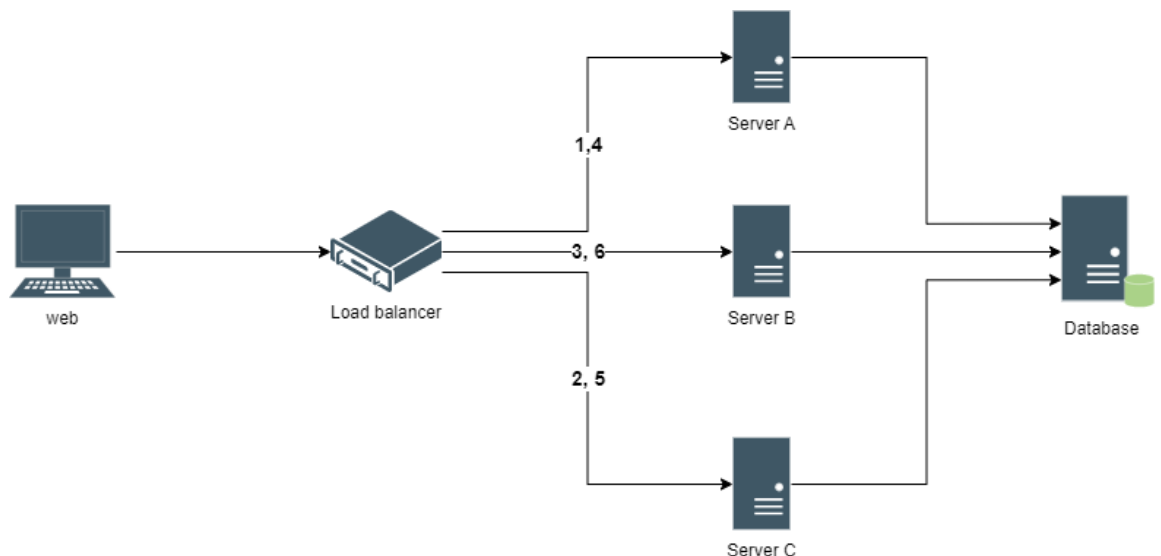


Рис. 2.9 Принцип роботи Random

Переваги:

- Загалом в більшості випадках, навантаження розподіляється досить рівномірно між всіма серверами в пулі.

Недоліки:

- Алгоритм передбачає, що сервери достатньо подібні, щоб обробляти еквівалентні навантаження. Якщо певні сервери мають більше ЦП, оперативної пам'яті чи інших специфікацій, алгоритм не має можливості розподілити більше запитів на ці сервери. В результаті сервери з меншою ємністю можуть перевантажуватися та швидше виходити з ладу, поки ємність на інших серверах не працює.
- Можливі випадки, коли деяку кількість разів алгоритм буде обирати сервер, що є неактивним, що призведе до втрати продуктивності. Навіть якщо ми маємо 2 сервери(1 з них не працює), то таким чином можливо 10000 разів потрапити на непрацюючий сервер.

2.1.10 Порівняльна таблиця алгоритмів

Таблиця 2.1 – Порівняльна таблиця описаних алгоритмів балансування

Load balanc ing algorit hm	Nat ure	Resp onse Time	Fault toler ance	Reso urce utiliz ation	Perfor mance	Throu ghput	Comp lexity	Over head	Exec ution time
Round Robin	static	medium	low	low	fast	high	low	low	slow
Weighted Round Robin	static	medium	medium	low	fast	high	low	low	slow
Least Connections	dynamic	slow	low	medium	medium	medium	medium	medium	medium

Кінець таблиці 2.1

Load balanc ing algorit hm	Nat ure	Resp onse Time	Fault toler ance	Reso urce utiliz ation	Perfor mance	Throu ghput	Comp lexity	Over head	Exec ution time
Weigh ted Least Conne ctions	dyna mic	slow	medi um	mediu m	mediu m	mediu m	mediu m	medi um	medi um
Resour ce Based (Adapt ive)	dyna mic	fast	high	high	slow	low	high	high	slow
Weigh ted Respo nse Time	dyna mic	fast	high	high	slow	low	high	high	slow
IP Hash	stati c	medi um	low	mediu m	mediu m	mediu m	mediu m	medi um	medi um
URL Hash	stati c	medi um	low	mediu m	mediu m	mediu m	mediu m	medi um	medi um
Rando m	stati c	medi um	low	low	fast	high	low	low	slow

Як ми можемо бачити з таблиці більш ретельний вибір наступного серверу для запиту потребує більших витрат. Саме тому не існує ідеального алгоритму балансування, а його підбір повинен проходити згідно з відповідними потребами.

2.2 Використані технології

2.2.1 JavaScript

Основні характеристики

1. JavaScript – це інтерпретована мова, тому вихідний код скриптів не потребує в компіляції. З метою підвищення ефективності він може компілюватися в байт-код Java, однак, це відбувається повністю прозоро і не вимагає ніяких додаткових дій.

2. JavaScript, в основному, застосовується як об'єктно-орієнтована мова програмування, однак сам по собі він є мультипарадигменним і підтримує об'єктно-орієнтований, імперативний і функціональний стилі.

3. Для даних застосовується слабкий (динамічний) контроль типів. При зверненні до даних вони автоматично приводяться до потрібного типу, якщо це можливо.

4. Як і Java, інтерпретатори цієї мови автоматично керують пам'яттю і очищають невикористовувані блоки, тобто програмі не потрібно слідкувати за виділенням і очищенням пам'яті, як, наприклад, на C++.

Переваги[24]:

- Швидкість - JavaScript, як правило, дуже швидко, оскільки він часто працює негайно в браузері клієнта. Поки це не потребує зовнішніх ресурсів, JavaScript не сповільнюється дзвінками на сервер. Крім того, основні браузери підтримують компіляцію JIT (якраз у часі) для JavaScript, тобто не потрібно складати код перед тим, як його запустити.

					ІАЛЦ.467200.003 ПЗ	Арк.
						50
Зм.	Арк.	№ докум.	Підпис	Дата		

- Простота - синтаксис JavaScript був натхненний Java's і його відносно легко навчитися порівняно з іншими популярними мовами, такими як C++.
- Популярність - JavaScript є скрізь в Інтернеті, а з появою Node.js все частіше використовується на бекенді. Існує незліченна кількість ресурсів для вивчення JavaScript. І Stackoverflow, і Github показують все більшу кількість проектів, які використовують JavaScript, і тяга, яку вона отримала в останні роки, лише очікується, що збільшиться.
- Сумісність - на відміну від PHP або інших мов сценаріїв, JavaScript можна вставити в будь-яку веб-сторінку. JavaScript можна використовувати в багатьох різних типах додатків через підтримку в інших мовах, таких як PEARL та PHP.
- Завантаження сервера - JavaScript на стороні клієнта, тому він зменшує попит на серверах в цілому, і простим програмам може взагалі не знадобитися сервер.
- Багаті інтерфейси - JavaScript можна використовувати для створення таких функцій, як перетягування та компоненти, такі як повзунки, всі вони значно покращують інтерфейс користувача та досвід сайту.
- Розширена функціональність - розробники можуть розширити функціональність веб-сторінок, написавши фрагменти JavaScript для сторонніх додатків, таких як Greasemonkey.
- Універсальність - Існує багато способів використання JavaScript через сервери Node.js. Якщо ви мали б завантажувати Node.js з Express, використовуйте базу даних документів, як MongoDB, і використовуйте JavaScript на фронтді для клієнтів, можна розробити цілий додаток JavaScript спереду на спину, використовуючи лише JavaScript.
- Оновлення - Оскільки поява EcmaScript 5 (специфікація сценаріїв, на яку покладається JavaScript), Ecma International присвячена оновленню

JavaScript щорічно. Поки що ми отримали підтримку браузера для ES6 у 2017 році та сподіваємось, що ES7 підтримується в майбутньому.

Недоліки:

- Безпека на стороні клієнта. Оскільки код JavaScript виконується на стороні клієнта, помилки та нагляди іноді можуть бути використані для зловмисних цілей. Через це деякі люди вирішують повністю відключити JavaScript.
- Підтримка браузера. Хоча сценарії на стороні сервера завжди виробляють однаковий вихід, різні браузери іноді по-різному інтерпретують код JavaScript. У наші дні відмінності мінімальні, і вам не доведеться турбуватися про це до тих пір, поки ви перевіряєте свій сценарій у всіх основних браузерах.

Майбутнє JavaScript

JavaScript пройшов довгий шлях з моменту його початкової реалізації: на його написання знадобилося всього 10 днів. Стандарт JavaScript, запропонований вперше як ECMAScript 1 у 1997 році, станом на кінець 2018 року перебуває у своїй 9-й ітерації (ES 2018). Відмінності між специфікаціями, описаними в ECMAScript 1 і ES 2018, величезні: вони, здається, описують різні мови. За останні роки JavaScript зазнав значних змін. Не всі погоджувалися з кожною зміною, але разом вони зробили JavaScript більш надійною, безпечною та виразною мовою.

Спільнота JavaScript сьогодні є, мабуть, найактивнішою спільнотою у програмуванні. Іноді здається, що щотижня виходять нові інструменти, фреймворки та бібліотеки. Існують усілякі препроцесори та транспілятори, починаючи від програм, які перекладають сучасні програми JavaScript, щоб їх могли запускати старіші двигуни JavaScript, до компіляції абсолютно нових мов за допомогою JavaScript. Сам стандарт JavaScript є документом, який постійно розвивається, з швидкими удосконаленнями. Механізми JavaScript вносять

					ІАЛЦ.467200.003 ПЗ	Арк.
						52
Зм.	Арк.	№ докум.	Підпис	Дата		

зміни майже так само швидко. Нові операційні системи в стадії розробки (наприклад, Fuchsia від Google) додають підтримку для розробки програм на JavaScript. Усе це означає, що у JavaScript захоплює майбутнє.

2.2.2 Node.js

Node.js — це середовище виконання JavaScript з відкритим вихідним кодом і між платформним. Це популярний інструмент практично для будь-яких проектів.

Node.js запускає двигун JavaScript V8, ядро Google Chrome, поза браузером. Це дозволяє розробляти серверну частину на JavaScript.

Програма Node.js виконується в одному процесі, не створюючи новий потік для кожного запиту. Node.js містить у своїй стандартній бібліотеці набір асинхронних примітивів введення-виводу, які запобігають блокуванню коду JavaScript, і загалом бібліотеки в Node.js написані з використанням неблокуючих парадигм, що робить поведінку блокування скоріше винятком, ніж нормою.

Коли Node.js виконує операцію введення-виведення, наприклад, читання з мережі, доступ до бази даних або файлової системи, замість того, щоб блокувати потік і витрачати цикли ЦП на очікування, Node.js відновить операції, коли відповідь повернеться.

Це дозволяє Node.js обробляти тисячі одночасних підключень з одним сервером, не вводячи тягар керування паралельністю потоків, що може бути значним джерелом помилок.

Node.js має унікальну перевагу, оскільки мільйони розробників інтерфейсу, які пишуть JavaScript для браузера, тепер можуть писати код на стороні сервера на додаток до коду на стороні клієнта без необхідності вивчати зовсім іншу мову.

У Node.js нові стандарти ECMAScript можна використовувати без проблем, оскільки вам не потрібно чекати, поки всі ваші користувачі оновлять

					ІАЛЦ.467200.003 ПЗ	Арк.
						53
Зм.	Арк.	№ докум.	Підпис	Дата		

свої браузери – ви самі вирішуєте, яку версію ECMAScript використовувати, змінюючи версію Node.js, і ви також можете ввімкнути певні експериментальні функції, запустивши Node.js з прапорцями[25].

2.2.3 HTTP/HTTPs

Протокол передачі гіпертексту (HTTP) — це протокол прикладного рівня для розподілених, спільних гіпермедійних інформаційних систем. Це основа передачі даних для всесвітньої павутини (тобто Інтернету) з 1990 року. HTTP — це загальний протокол без стану, який можна використовувати для інших цілей, а також із використанням розширень методів запиту, кодів помилок та заголовків.

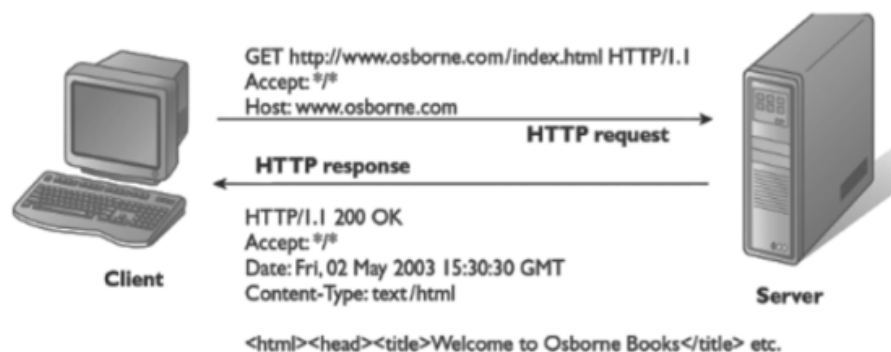


Рис. 2.10 Схема роботи HTTP протоколу

Цей посібник базується на специфікації RFC-2616, яка визначає протокол, який називається HTTP/1.1. HTTP/1.1 – це версія оригінального HTTP (HTTP/1.0). Основна відмінність між HTTP/1.0 і HTTP/1.1 полягає в тому, що HTTP/1.0 використовує нове з’єднання для кожного обміну запитами/відповідями, де як з’єднання HTTP/1.1 може використовуватися для одного або кількох обмінів запитами/відповідями. HTTPS, в свою чергу, є безпечним протоколом передачі гіпертексту. Це протокол, за яким зашифровані дані HTTP передаються через безпечне з’єднання. Використовуючи безпечне з’єднання, таке як захист транспортного рівня або рівень безпечних сокетів,

підтримується конфіденційність і цілісність даних, а також перевіряється автентифікація веб-сайтів.

Будучи протоколом без стану, HTTP працює поверх TCP, і це протокол на основі з'єднання, наприклад, SCTP і TCP. Після передачі клієнтами запитів HTTP на сервер між сервером і клієнтом швидко встановлюється TCP-з'єднання. Аналогічно, з'єднання TCP розривається після отримання відповіді[26].

Важливо знати, що кожен запит HTTP розблоковує ізольовані TCP-з'єднання з сервером. Розберемося в цьому на прикладі, при передачі 10 запитів клієнтом на сервер; буде встановлено ще 10 ізольованих TCP-з'єднань. Відповідно, він закривається після придбання резервного.

Деталі повідомлення HTTP закодовані в ASCII, де кожне з повідомлень із запитом HTTP контролює різні версії протоколу HTTP, наприклад:

- Методи HTTP (GET/POST тощо)
- Інформація про хост
- Версія протоколу HTTP (HTTP/1.1, HTTP/2)
- HTTP-заголовки (тип вмісту, довжина вмісту)

Разом з тілом, що складається з реального повідомлення, надсилається на сервер. Крім того, заголовки HTTP відрізнялися від 200 байт до 2 КБ. Однак загальний розмір заголовка HTTP становить 700-800 байт, коли провідний веб-додаток використовує більше файлів cookie та інших досвідчених інструментів на стороні клієнта, які можуть безперешкодно поширювати різні специфікації сховища агента та зменшувати корисне навантаження заголовка HTTP.

REST

Архітектурний стиль, REST (Representational State Transfer) — це, безумовно, найбільш стандартизований спосіб структурування веб-API для запитів[27]. REST — це суто архітектурний стиль, заснований на кількох принципах. API, що дотримуються принципів REST, називаються RESTful API.

					ІАЛЦ.467200.003 ПЗ	Арк.
						55
Зм.	Арк.	№ докум.	Підпис	Дата		

API REST використовують модель запиту/відповіді, де кожне повідомлення від сервера є відповіддю на повідомлення від клієнта. Загалом, RESTful API використовує HTTP як транспортний протокол. У таких випадках пошук повинен використовувати запити GET. Запити PUT, POST та DELETE слід використовувати для мутації, створення та видалення відповідно (уникайте використання запитів GET для оновлення інформації).

Переваги HTTP

- Затримка: HTTP може зменшити швидкість затримки підключення через відсутність процедури рукоштовування при виконанні запиту.
- Гнучкість: HTTP може завантажувати розширення та демонструвати життєво важливі дані, коли програма потребує додаткових компетенцій.

2.2.4 WebSocket/WebSocket Secure

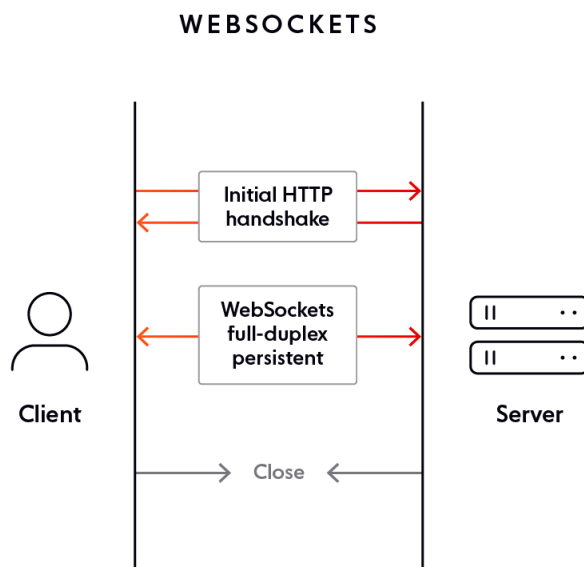


Рис. 2.11 Схема роботи WS протоколу

WebSockets дозволяють як серверу, так і клієнту надсилати повідомлення в будь-який час без будь-якого зв'язку з попереднім запитом. Однією з помітних

переваг використання WebSockets є те, що майже кожен браузер підтримує WebSockets.

WebSocket вирішує кілька проблем із HTTP[28]:

- Двонаправлений протокол – будь-який клієнт/сервер може надіслати повідомлення іншій стороні (у HTTP запит завжди ініціюється клієнтом, а відповідь обробляється сервером, що робить HTTP однонаправленим протоколом)
- Full-duplex communication – клієнт і сервер можуть спілкуватися один з одним незалежно одночасно.
- Єдине з'єднання TCP – після оновлення HTTP-з'єднання на початку клієнт і сервер спілкуються через те саме TCP-з'єднання (постійне з'єднання) протягом усього життєвого циклу підключення WebSocket.

Як встановлюються з'єднання WebSocket?

Процес починається з рукостискання WebSocket, яке передбачає використання нової схеми WS або WSS. Щоб швидко зрозуміти, ви можете вважати їх еквівалентними HTTP та HTTPS відповідно.

Використовуючи цю схему, очікується, що сервери та клієнти будуть дотримуватися стандартного протоколу підключення WebSocket. Встановлення з'єднання WebSocket починається з оновлення HTTP-запиту, який містить кілька заголовків, наприклад Connection: Upgrade, Upgrade: WebSocket, Sec-WebSocket- Key тощо.

Ось як це з'єднання встановлюється:

- Запит
- Підключення: заголовок оновлення позначає рукостискання WebSocket, а Sec-WebSocket-Key має випадкове значення, закодоване Base64. Це значення генерується довільно під час кожного рукостискання WebSocket. Крім перерахованого вище, заголовок ключа також є

					ІАЛЦ.467200.003 ПЗ	Арк.
						57
Зм.	Арк.	№ докум.	Підпис	Дата		

частиною цього запиту. Перераховані вище заголовки, об'єднані, утворюють запит HTTP GET. У ньому будуть аналогічні дані:

GET ws://websocketexample.com:8181/ HTTP/1.1

Host: localhost: 8181

Connection: Upgrade

Pragma: no-cache

Cache-Control: no-cache

Upgrade: websocket

Sec-WebSocket-Version: 13

Sec-WebSocket-Key: b6gjhT32u488lpuRwKaOWs==

Відповідь

- Заголовок відповіді Sec-WebSocket-Accept містить цінність, подану в заголовку запиту Sec-WebSocket-Key. Це пов'язано зі специфікацією конкретного протоколу і широко використовується для запобігання оманливій інформації. Іншими словами, це покращує безпеку API та запобігає створенню помилок у розробці додатків на погано налаштованих серверах.
- Після успішного виконання попередньо надісланого запиту буде отримано відповідь, подібну до зазначеної нижче текстової послідовності[29]:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: rG8wsswmHTJ85lJgAE3M5RTmcCE=

Переваги використання WebSocket

- Безпека: WebSockets використовує найкращі методи шифрування, що означає, що з'єднання краще захищене та захищене від кібератак.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		58

- Гнучкість: завдяки гнучкості протоколу WebSocket він може ефективно підтримувати всі плагіни та підпротоколи. Таким чином, це вказує, що WebSocket може підтримувати додаткову функціональність.
- Двонаправлені з'єднання: як згадувалося раніше, WebSockets використовують двонаправлені з'єднання, що означає, що після завершення початкового рукостискання та встановлення захищеного з'єднання клієнт і сервер можуть вільно спілкуватися один з одним одночасно.

					ІАЛЦ.467200.003 ПЗ	Арк.
						59
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВОК ДО РОЗДІЛУ 2.

В даному розділі було детально проаналізовано деякі існуючі алгоритми балансування(статичні та динамічні), а також описано використані технології під розробки балансеру.

Для кожного алгоритму було розглянуто приклад його роботи, а також переваги та недоліки. В будь-якому разі я з'ясував, що обрати ідеальне рішення майже неможливо. Підбір найбільш продуктивного серверу потребує більше роботи ЦП. При цьому якщо не проводити детальний аналіз, швидкість відгуку буде меншою. Саме тому, при виборі алгоритму роботи балансеру, перш за все варто враховувати потреби, налаштування та залежності системи, до якої він застосовується.

При виборі мови програмування було надано перевагу Javascript, а саме Node.js. Оскільки дана технологія є однопоточною, то це позбавляє від необхідності блокувати паралельний трафік. За одиницю часу програма буде обробляти один вхідний запит. Таким чином балансувальник буде мати чергу, де кожна задача буде чекати закінчення іншої.

Також було розглянуто протоколи HTTP та WS, які є найбільш поширеними при розробці сучасного програмного забезпечення. Майже кожен Web застосунок має потребу в даних видах трафіку, адже існують задачі, які потребують у їх комбінуванні.

Варто зазначити, що з цим набором технологій можливо розробити балансувальник, програма якого не буде мати жодної сторонньої бібліотеки, а лише ті, що є вбудованими в платформу Node.js.

Виходячи з цього можна зробити висновок, що розроблений балансер має бути гнучким до обраного алгоритму вибору наступної точки, а також до вхідного трафіку. При цьому програма має бути зрозумілою та придатною до майбутніх оновлень.

					ІАЛЦ.467200.003 ПЗ	Арк.
						60
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 3

ПРОЕКТУВАННЯ СИСТЕМИ

3.1. Структура програми

3.1.1 Базова концепція балансувальника навантаження

Перш за все, варто почати з того, що реалізований балансувальник є прикладним(7 рівень OSI). Основним його завданням є переправлення(на основі зібраної інформації) вхідних запитів до кінцевих точок – серверів. В основі балансера є HTTP/HTTPs сервер, який обробляє запити, що надходять від клієнтів. Він вміє працювати з різними типами підключень(односторонні – HTTP/HTTPs та двосторонні – WS/WSs). Робота з цими протоколами є обов’язковою, адже сучасне програмне забезпечення зазвичай має потребу у використанні як звичайних API запитів, так і сокетів. Також, важливим моментом є те, що для правильного функціонування балансувальника потрібно узгоджувати ті частини його роботи, які виконуються паралельно, адже при цьому можуть виникати випадки, коли для вибору кінцевої точки будуть присутні неактуальні дані про стан системи. В багатьох мовах програмування для цього використовують м’ютекси, що блокують паралельні ділянки коду. Оскільки для реалізації програми було обрано платформу Node.js, то це стає непотрібним, так як Javascript однопоточна мова програмування. Тобто реалізований балансувальник буде обробляти один запит за одиницю часу, а інші будуть чекати своєї черги.

Варто звернути увагу на те, що балансер має достатньо широкий функціонал, який можна змінювати у файлі конфігурації. При цьому для реалізації програми не було використано жодної сторонньої бібліотеки(лише влаштовані в Node.js).

Основні опції балансувальника:

					ІАЛЦ.467200.003 ПЗ	Арк.
						61
Зм.	Арк.	№ докум.	Підпис	Дата		

- Контроль вхідного потоку даних. При цьому існує можливість роботи з зашифрованими даними. Загалом здійснюється обробка запитів на основі таких протоколів – HTTP/HTTPS та WS та WSS.
- Робота з пулом серверів(кінцевих точок)
- Налаштування безпеки(SSL та захист від DDos атак)
- Кешування
- Логування

Приклад основного файлу конфігурації:

balancerConfig.js

```
'use strict';

const fs = require('fs');

module.exports = {
  // порт на якому працює сервер-балансувальник
  port: 3000,
  // тип серверу - HTTP|HTTPS
  secure: true,
  // захист від DDos атак
  doss: true,
  // ssl параметри
  ssl: {
    key: fs.readFileSync(`${__dirname}/certs/server-key.pem`),
    cert: fs.readFileSync(`${__dirname}/certs/server-crt.pem`),
    ca: [fs.readFileSync(`${__dirname}/certs/client-crt.pem`)],
    requestCert: true,
    rejectUnauthorized: true,
  },
};
```

В основі роботи програми є дві основні частини, які в залежності від обраних налаштувань, можуть значно доповнюватись:

- Черга
- Контролер

3.1.2 Черга.

Черга – клас, завданням якого є робота з пулом серверів. По суті, кожна черга – це певний алгоритм вибору наступної кінцевої точки для вирішення

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		62

завдання. Він повністю незалежний від роботи контролера і виконує наступні завдання:

- Видача наступного серверу для вирішення завдання
- Перевірка стану серверів

В нашій програмі завжди будуть дві черги: одна працює з пулом HTTP серверів, інша з WS. Також існують певні обмеження в залежності від того з яким протоколом йде робота, адже не всі алгоритми підходять для HTTP запитів, аналогічно і для WS. Для HTTP неможлива коректна робота з Least Connection та Weighted Least Connection, адже в цьому випадку з'єднання буде розриватись одразу після отримання відповіді, а так як ми маємо всього один потік, то фактично всі запити будуть йти на один і той же сервер. WS протокол, в свою чергу, не може працювати з адаптивним алгоритмом, адже при цьому рахується час відповіді кожної задачі. Для сокетів цей час неможливо підрахувати, так як проходить всього один запит для встановлення зв'язку, а після цього сервер та клієнт працюють між собою майже без втручання балансера.

Також, черги мають різні опції, які ви можете змінювати у файлі конфігурації:

- Алгоритм балансування(Round Robin, Weighted Round Robin, Least Connection, Weighted Least Connection, Resource Based (Adaptive), Weighted Response Time, Sticky Session, URL Hash, Random)
- Список серверів(кінцевих точок програми)
- Налаштування перевірки стану серверів(пасивна/активна, інтервал, тощо).

Завдяки цьому, існує можливість підбирати налаштування в залежності від ваших потреб та завдань. Але потрібно пам'ятати, що при покращенні роботи певних частин балансера, можна втратити продуктивність інших компонентів. Наприклад, якщо увімкнути активну перевірку стану серверів, то будемо мати

					ІАЛЦ.467200.003 ПЗ	Арк.
						63
Зм.	Арк.	№ докум.	Підпис	Дата		

актуальну інформацію про їхній стан. Це покращить відсоток вдалих запитів, але при цьому ми програємо в часі.

Клас *Queue*

```
'use strict';

const config = require('../config');
const { typeOptionMapping } = require('../infrastructure/Utils');

class Queue {
  constructor(type) {
    this.Option = require(`../${this.resolveAlgorithm(type)}`);
    this.type = type;
  }

  resolveAlgorithm(type) {
    const { algorithm } = config[type];

    if (algorithm && typeOptionMapping[type].includes(algorithm))
      return algorithm;
    return 'RoundRobin';
  }

  init() {
    const { Option, type } = this;
    const queue = new Option(type);
    queue.init();

    return queue;
  }
}

module.exports = Queue;
```

Даний клас є обгорткою для реалізованих алгоритмів, який на основі налаштувань імпортує потрібний клас для черги.

Нижче представлено об'єкт, що містить в собі доступні алгоритми балансування для протоколів HTTP та WS.

```
const typeOptionMapping = {
  http: [
    'RoundRobin',
    'WeightedRoundRobin',
    'IpHash',
    'UrlHash',
    'Random',
    'WeightedResponseTime',
    'Adaptive',
  ],
  ws: [
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						64
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    'RoundRobin',
    'WeightedRoundRobin',
    'IpHash',
    'UrlHash',
    'Random',
    'LeastConnection',
    'WeightedLeastConnection',
    'WeightedResponseTime',
  ],
};

```

Якщо користувач не вказав опцію для вибірки серверу, або вона не відповідає для даного протоколу, то встановлюється стандартний алгоритм балансування – Round Robin.

3.1.3 Контролер

Контролер – клас, який призначений для встановлення комунікації з кінцевими точками доступу. Так як і черги, в програмі завжди працюють два контролери:

- Http – для встановлення односпрямованого зв'язку
- Ws – для встановлення двоспрямованого зв'язку

Кожен з контролерів має можливість працювати з зашифрованим трафіком(HTTPS/WSs). Цю опцію також можна увімкнути у файлі конфігурації. При цьому кожен з них має власну чергу, яка не має відмінностей в алгоритмі своєї роботи.

Приклад файлу конфігурації контролера та його черги:

controller.js

```

module.exports = {
  // масив серверів
  servers: [
    { host: '127.0.0.1', port: 8000 },
    { host: '127.0.0.1', port: 8001 },
    { host: '127.0.0.1', port: 8002 },
  ],
  // алгоритм вибору наступної точки
  algorithm: 'WeightedRoundRobin',
  // перевірка стану серверів
  healthCheck: {
    // тип перевірки(активний чи пасивний)
  }
};

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						65
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    activeCheck: true,
    // маршрут
    path: '/health',
    // інтервал (при пасивній перевірці)
    time: 60, // seconds
  },
  // безпека
  secure: {
    // передача зашифрованого трафіку
    up: false,
    // ssl параметри для кінцевих точок
    ssl: {
      key: fs.readFileSync(`${__dirname}/certs/server-key.pem`),
      cert: fs.readFileSync(`${__dirname}/certs/server-crt.pem`),
      ca: fs.readFileSync(`${__dirname}/certs/client-crt.pem`),
      rejectUnauthorized: true,
    },
  },
};

```

3.1.4 Безпека

Розроблений балансер має певні компоненти захисту, які знову ж таки можна увімкнути у налаштуваннях. Серед них:

- SSL
- захист від DDos атак

SSL

При увімкненні протоколу безпечних сокетів є можливість передачі в параметри ключів, сертифікатів, паролів та багато іншого. Таким чином кожен користувач балансувальника може налаштувати безпеку в залежності від своїх потреб. Потрібно пам'ятати те, що при передачі зашифрованого трафіку, використовуються додаткові цикли ЦП(на дешифрування). Загалом програма дозволяє шифрувати трафік на кожному етапі доставки даних:

- Від клієнта до балансера
- Від балансера до кінцевих точок

Таким чином, при запуску балансувальника проходить перевірка параметрів, які вказує користувач. Якщо він встановив `secure: false`, то відбувається ініціалізація HTTP серверу, який встановлює з'єднання з будь-яким клієнтом. Якщо навпаки, то починає працювати HTTPS сервер, який в

					ІАЛЦ.467200.003 ПЗ	Арк.
						66
Зм.	Арк.	№ докум.	Підпис	Дата		

залежності від SSL опцій буде проводити автентифікацію віддаленого ресурсу, що прагне пройти балансир і зчитати дані з серверів.

Для створення серверу використовуються влаштовані в Node.js бібліотеки http та https. При цьому https самостійно проводить перевірку сертифікатів, нам лише потрібно передати до функції createServer параметри SSL:

```
const protocol = secure ? https : http;  
const options = secure ? { ...ssl } : {};  
const server = protocol.createServer(options, this.http.handler);
```

DDos protection

Основною функцією захисту від DDos атак є закриття з'єднання з клієнтом, який виконує занадто велику кількість запитів на певний проміжок часу. Таким чином, перш за все потрібно встановити функцію-слухач на подію connection(момент коли віддалений ресурс намагається під'єднатися до нашого балансеру).

При створенні серверу функція createServer повертає об'єкт, на який можна навішувати лісєнерів на певні події(через функцію on('event', listener)).

В даному випадку, на момент встановлення з'єднання, відбудеться виклик функції класу DDoS – handler.

```
if (doss) { // перевірка чи потрібно вмикати захист  
  // ініціалізація класу, параметри - інтервал між запитами в мс та к-  
  сть запитів  
  this.ddos = new DDoS(3000, 100);  
  // навішення функції на подію  
  server.on('connection', this.ddos.handler);  
}
```

Щоразу, коли клієнт(ip) намагається встановити з'єднання з балансером, функція handler записує даний момент часу до об'єкту ipTable(ключем буде ip).

```
this.ipTable[ip] = {  
  count : 1,  
  time  
};
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						67
Зм.	Арк.	№ докум.	Підпис	Дата		

Якщо новий запит відбудеться скоріше, ніж пройде 3 секунди, то кількість здійснених підключень для даного ір збільшиться на одиницю. Але коли інтервал між запитами є більшим, ніж заданий час, то кількість запитів стане рівною одиниці.

```
if(time - this.ipTable[ip].time > this.time) {
    this.ipTable[ip] = {
        count : 1,
        time
    };
    return;
}
this.ipTable[ip].count++;
this.ipTable[ip].time = time;
```

На момент, під час якого кількість з'єднань віддаленого ресурсу до балансера дійде до ста і більше, балансувальник не дозволить йому встановити підключення, допоки не пройде заданий інтервал часу.

```
if(this.ipTable[ip].count > this.requestCount) {
    socket.end('HTTP/1.1 429 Too Many Requests');
    socket.destroy();
    logger.warning(`Destroy connection ip: ${ip}`);
}
```

Клас *DDoS*

```
'use strict';

const logger = require('../Logger');

class DDoS {
    constructor(time, requestCount) {
        this.ipTable = {};
        this.time = time;
        this.requestCount = requestCount;
    }

    handler = (socket) => {
        try {
            const ip = socket.address().address;
            const time = Date.now();
            if(ip in this.ipTable) {
                if(time - this.ipTable[ip].time > this.time) {
                    this.ipTable[ip] = {
                        count : 1,
                        time
                    };
                    return;
                }
            }
            this.ipTable[ip].count++;
        }
    }
}
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						68
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        this.ipTable[ip].time = time;
        if(this.ipTable[ip].count > this.requestCount) {
            socket.end('HTTP/1.1 429 Too Many Requests');
            socket.destroy();
            logger.warning(`Destroy connection ip: ${ip}`);
        }
        return;
    }
    this.ipTable[ip] = {
        count : 1,
        time
    };
} catch (e) {
    logger.error(e);
}
}
}

module.exports = DDoS;

```

Захист від DDos атак є корисною опцією при використанні балансера навантаження. Таким чином є можливість захистити власний пул серверів, при цьому балансувальник стає стіною перед основними прикладними додатками.

3.1.5 Кешування

Розроблений балансувальник включає в себе такі алгоритми вибору наступного серверу, які потребують зберігання певних даних(IP Hash та Url Hash). Через це виникає потреба у кешуванні. Система має власний режим кешування, без використання сторонніх бібліотек. Більшість існуючих балансирів використовують для цього Redis, але це при цьому мають більші витрати ЦП. Оскільки програма повинна мати якомога кращу пропускну здатність, було прийнято рішення зберігати дані в пам'яті, а саме в об'єкті Map. При перезапуску серверу кеш очищується. Такий варіант є правильним, адже на цей момент сервери повинні мати однакові умови для виконання подальших етапів балансування.

Клас *CacheWorker*

```

'use strict';

const logger = require('../Logger');

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						69
Зм.	Арк.	№ докум.	Підпис	Дата		

```

class CacheWorker {
  constructor() {
    //ініціалізація об'єкту Map(всі ключі є унікальними)
    this.cache = new Map();
  }

  //взяти дані, які зберігаються з вхідним ключем
  get(key) {
    try {
      return this.cache.get(key);
    } catch (e) {
      logger.error(e);
    }
  }

  //перевірка чи існують дані з вхідним ключем
  has(key) {
    try {
      return this.cache.has(key);
    } catch (e) {
      logger.error(e);
    }
  }

  //встановлення даних за певним ключем. Якщо переданий параметр
  ttl(час зберігання), то дані видаляться через даний інтервал.
  set(key, value, ttl = null) {
    try {
      this.cache.set(key, value);

      if (ttl) {
        setTimeout(() => {
          this.cache.delete(key);
        }, ttl * 10);
      }
    } catch (e) {
      logger.error(e);
    }
  }
}

module.exports = CacheWorker;

```

Розроблений клас кешування є доволі простим і потребує мінімум ресурсів. Також кожний метод класу загорнуто в конструкцію try {...} catch(e) {...}, для того щоб вдало перехоплювати можливі помилки. Будь-яка помилка буде відображатися в консолі за допомогою написаного логера.

					ІАЛЦ.467200.003 ПЗ	Арк.
						70
Зм.	Арк.	№ докум.	Підпис	Дата		

3.1.6 Логування

Для транслювання різних подій, попереджень та помилок в консолі було розроблено клас `Logger`, що має 3 публічних методи:

- `Info`
- `Warning`
- `Error`

З наведених назв стає зрозуміло, що дані методи відображають рівень важливості повідомлення. Приклад виводу тексту в консоль(всі рівні):

```
INFO | Success | timestamp: Tue May 31 2022 23:47:05 GMT+0300 (GMT+03:00)
WARNING | Warning | timestamp: Tue May 31 2022 23:47:05 GMT+0300 (GMT+03:00)
ERROR | Error | timestamp: Tue May 31 2022 23:47:05 GMT+0300 (GMT+03:00)
```

Рис. 3.1 Вивід логера в консоль

Тобто спочатку ми бачимо тип повідомлення(з відповідним кольором), потім саме повідомлення, а наприкінці дата й час.

Клас *Logger*

```
'use strict';

class Logger {
  constructor() {
    // ініціалізація кодів для кольорів
    this.levels = {
      error: '\x1b[31m',
      info: '\x1b[32m',
      warning: '\x1b[33m',
    };
  }

  // приватний метод для виводу в консоль
  _log(message, level) {
    const time = new Date();
    const formatMessage = `${
      this.levels[level]
    }${level.toUpperCase()} | \x1b[0m ${message} | timestamp: ${time}`;

    // для виводу тексту використовуємо влаштовану в Node.js функцію
    console.log(formatMessage);
  }

  // всередині методів відбувається виклик функції _log з вказанням
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						71
Зм.	Арк.	№ докум.	Підпис	Дата		

```

відповідного рівня
    info(message) {
        this._log(message, 'info');
    }

    warning(message) {
        this._log(message, 'warning');
    }

    error(message) {
        this._log(message, 'error');
    }
}

module.exports = new Logger();

```

Приклад виклику функцій логера:

```

logger.info(`Success`);
logger.warning(`Warning`);
logger.error('Error');

```

3.1.7 Точка входу програми

Точкою входу є головний файл програми – *index.js*. Він містить клас *LoadBalancer*. Саме в ньому відбувається підняття серверу, який пізніше буде обробляти вхідні запити та передавати їх до кінцевих точок. Також у ньому збираються всі окремі компоненти програми і починають працювати як одне ціле.

Клас *LoadBalancer*

```

'use strict';

const http = require('http');
const https = require('https');
const { secure, port, ssl, doss } = require('./config/balancer');
const { HttpController, WsController } = require('./controllers');
const Queue = require('./queue');
const DDoS = require('./infrastructure/DDoS');
const logger = require('./infrastructure/Logger');

class LoadBalancer {
    constructor() {
        this.ddos = null;
        this.wsQueue = new Queue('ws').init();
        this.httpQueue = new Queue('http').init();
        this.ws = new WsController(this.wsQueue);
        this.http = new HttpController(this.httpQueue);
    }
}

```

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		72

```

init = () => {
  const protocol = secure ? https : http;
  const options = secure ? { ...ssl } : {};
  const server = protocol.createServer(options, this.http.handler);
  server.on('upgrade', this.ws.handler);

  if (doss) {
    this.ddos = new DDoS(3000, 100);
    server.on('connection', this.ddos.handler);
  }

  server.listen(port, () => {
    logger.info(`Start load balancer listen on ${port}`);
  });
}

const balancer = new LoadBalancer();
balancer.init();

```

В конструкторі класу проходить ініціалізація двох контролерів(HTTP та WS) та відповідних черг. Захист від DDos на цьому етапі ще не встановлюється.

```

constructor() {
  this.ddos = null;
  this.wsQueue = new Queue('ws').init();
  this.httpQueue = new Queue('http').init();
  this.ws = new WsController(this.wsQueue);
  this.http = new HttpController(this.httpQueue);
}

```

В методі init відбувається запуск балансера навантаження. Саме в цьому місці ми визначаємо тип працюючого серверу, навішуємо обробники для вхідних підключень та забезпечуємо захист від DDos атак.

```

init = () => {
  const protocol = secure ? https : http;
  const options = secure ? { ...ssl } : {};
  const server = protocol.createServer(options, this.http.handler);
  server.on('upgrade', this.ws.handler);

  if (doss) {
    this.ddos = new DDoS(3000, 100);
    server.on('connection', this.ddos.handler);
  }

  server.listen(port, () => {
    logger.info(`Start load balancer listen on ${port}`);
  });
}

```

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		73

І на останок відбувається створення екземпляру класу та викликаємо метод початку роботи програми

```
const balancer = new LoadBalancer();  
balancer.init();
```

В даному випадку, можна також спостерігати застосування інкапсуляції, коли певні незалежні компоненти збираються в одному місці програми. Дана парадигма полегшує адаптацію нового коду, а також дозволяє контролювати весь процес в точці входу.

3.2 Черга

3.2.1 ServerPool

Даний клас відповідає за контроль над набором вхідних серверів. Серед його завдань зберігання поточної інформації про стан кінцевих точок, перевірка стану інстансів, а також багато допоміжних функцій. Кожен з реалізованих алгоритмів балансування наслідуює даний клас, адже всі вони працюють з інформацією, що для них збирає *ServerPool*.

В конструкторі класі відбувається ініціалізація потрібних нам властивостей.

```
constructor(type) { //HTTP чи WS  
  this.up = false; //перевірка стану  
  this.active = false; //тип перевірка стану  
  this.path = '/'; //маршрут для перевірки стану  
  this.protocol = null; //протокол для запиту(в залежності від  
  налаштувань безпеки)  
  this.realProtocol = null; //реальний протокол(HTTP, WS ...) - лише  
  для логування  
  this.isTimeDependence = false; //показує чи потрібно рахувати час  
  відповіді  
  this.servers = []; //масив серверів  
  this.ssl = {}; //параметри безпеки  
  this.time = 5 * 60; //інтервал перевірки стану  
  this.setParams(type); //виклик функції для встановлення параметрів на  
  основі користувацьких налаштувань  
}
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						74
Зм.	Арк.	№ докум.	Підпис	Дата		

В методі `setParams` відбувається оновлення властивостей пулу на основі налаштувань користувача у файлі конфігурації. Якщо певні дані не вказані, залишаються вхідні параметри. Також, встановлюється параметр `isTimeDependence`, що показує чи потрібно заміряти час відповіді при перевірці стану серверу(в даному балансері потрібно лише для алгоритму `Weighted Response Time`).

```
setParams = (type) => {
  const params = config[type];
  if (params) {
    const {healthCheck, secure} = params;
    if (healthCheck) {
      this.up = healthCheck.up;
      this.active = healthCheck.activeCheck;
      this.path = healthCheck.path;
      this.time = healthCheck.time || this.time;
    }

    if (secure) {
      this.protocol = secure.up ? https : http;
      this.realProtocol = secure.up ? `${type}s` : type;
      this.ssl = secure.up ? secure.ssl : {};
    }

    this.isTimeDependence = params.algorithm ===
    'WeightedResponseTime';
    this.servers = params.servers;
  }
}
```

Запит, який визначає стан серверу, відправляється в методі `healthCheck`. Спочатку ми формуємо опції, які сформовані на основі вхідних параметрів, а потім відправляємо запит за маршрутом що вказаний користувачем у налаштуваннях. На основі статус коду, що прийшов у відповіді ми можемо зрозуміти чи сервер здатний приймати запити. Цим значенням може бути або 200, або 426 код(якщо кінцевий сервер працює на основі WS протоколу). Якщо прийшли дані значення, то ми повертаємо `true` – сервер активний, в іншому випадку `false`. Сам метод повертає об'єкт `Promise`, який використовується для відкладених та асинхронних операцій.

```
healthCheck = (server) => {
  return new Promise((resolve, reject) => {
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						75
Зм.	Арк.	№ докум.	Підпис	Дата		

```

const options = {
  method: 'GET',
  hostname: server.host,
  port: server.port,
  path: this.path,
  ...this.ssl
};

const request = this.protocol.request(options, (res) => {
  if (res.statusCode === 200 || res.statusCode === 426) {
    resolve(true);
  }

  resolve(false);
},
);

request.on('error', (err) => {
  resolve(false);
});
request.end();
});
}

```

Асинхронний метод `updateInstance` викликає функцію `healthCheck` для певного серверу. Також вимірюється час виконання перевірки стану. При цьому відбувається логування для інформування щодо роботи вхідних серверів. Також, якщо потрібно, заміряємо час виконання запиту.

```

updateInstance = async (server) => {
  try {
    let isHealth;
    let responseTime;

    if (this.isTimeDependence) {
      const startTime = new Date().getTime();
      isHealth = await this.healthCheck(server);
      const endTime = new Date().getTime();
      responseTime = endTime - startTime;
    } else {
      isHealth = await this.healthCheck(server);
    }

    logger.info(
      `Health check:
      ${this.realProtocol}://${server.host}:${server.port} ${
        isHealth ? 'OK' : 'NOT FOUND'
      }`,
    );

    return { ...server, responseTime, active: isHealth };
  }
}

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						76
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    } catch (e) {
      logger.error(e);
    }
  }
}

```

Нижче представлено метод `passiveCheck`, який оновлює інформацію щодо стану всіх серверів пулу. В середині, за допомогою функції `map` ми викликаємо `updateInstance` для кожного елементу масиву. Це все загорнуто у функцію `Promise.all()`, яка виконається тоді, коли завершиться виконання вкладеного циклу.

```

passiveCheck = async () => {
  try {
    this.servers = await
    Promise.all(this.servers.map(this.updateInstance));
  } catch (e) {
    logger.error(e);
  }
}

```

Завдання функції `activeCheck` полягає у тому, щоб викликати функцію `passiveCheck`, якщо у нас увімкнений активний тип перевірки стану серверів. Цей метод знадобиться нам пізніше, ми будемо його викликати під час кожного нового запиту.

```

activeCheck = async () => {
  try {
    if (this.active) await this.passiveCheck();
  } catch (e) {
    logger.error(e);
  }
}

```

І на останок метод `init`. Саме цей метод фактично починає роботу пулу серверів. Якщо ввімкнено пасивний тип перевірки стану, то відбувається створення таймеру – `setInterval`, який буде викликати функцію `passiveCheck` через певний інтервал часу.

```

init = async () => {
  try {
    if (this.up && !this.active) {
      await this.passiveCheck();
      setInterval(this.passiveCheck, this.time * 1000);
    }
  }
}

```

```

    } catch (e) {
        logger.error(e);
    }
}

```

3.2.2 Round Robin

Клас RoundRobin, як і усі інші класи алгоритмів наслідуює описаний вище клас ServerPool.

```
class RoundRobin extends ServerPool
```

Саме від нього кожен з алгоритмів отримує інформацію про стан серверів в даний момент часу.

В конструкторі класу відбувається оголошення значення індексу серверу, що виконає наступний запит, а також значення глибини вибірки(знадобиться нам пізніше, при перевірці активності кінцевої точки). Функція super робить виклик батьківського конструктора. Туди ми передаємо параметр type(тип черги, в залежності від контролера).

```

constructor(type) {
    super(type);

    this.currentIndex = 0;
    this.depth = 0;
}

```

Нижче описано метод setNextInstance, який збільшує значення індексу наступного серверу на одиницю. Якщо відбувся прохід по всім значенням масиву, то встановлюємо індекс нуль(перший елемент).

```

setNextInstance() {
    this.currentIndex =
        this.currentIndex === this.servers.length - 1 ? 0 :
        this.currentIndex + 1;
}

```

Метод next по значенню currentIndex витягує з масиву сервер для обробки запиту, а також викликає функцію setNextInstance.

					ІАЛЦ.467200.003 ПЗ	Арк.
						78
Зм.	Арк.	№ докум.	Підпис	Дата		

```

next() {
  const currentInstance = this.servers[this.currentIndex];
  this.setNextInstance();

  return currentInstance;
}

```

Функція `pick` присутня в кожному алгоритмі балансеру, головне її завдання надати контролеру наступну кінцеву точку для обробки запиту. Якщо немає активних серверів, повертаємо `null`.

Всередині методу на початку відбувається перевірка на те, чи є ще сервери, які не було перевірено на їхню активність. Якщо глибина проходу рівна довжині масиву серверів, то це означає що активних серверів немає.

Коли не спрацьовує перший `if`, методом `next` беремо наступний інстанс. Він проходить перевірку на активність і у разі успіху надходить до контролеру (значення глибини вибірки встановлюється рівним 0). Якщо сервер неактивний, глибина вибірки збільшується на одиницю, і функція `pick` повертає себе саму. Тобто вся описана вище логіка відбувається знову.

```

pick() {
  try {
    if (this.depth === this.servers.length) {
      return null;
    }

    const instance = this.next();
    if (instance.active) {
      this.depth = 0;
      return instance;
    }

    this.depth += 1;
    return this.pick();
  } catch (e) {
    logger.error(e);
  }
}

```

Також кожний алгоритм має методи `up` і `down`, які викликаються в середині контролерів під час певних подій. Завданням цих функцій є оновлення інформації про певний сервер. Round Robin є статичним та не робить зміни в

стані інстансу. Саме тому для цього алгоритму в цих методах немає жодної логіки.

```
up() {}  
down() {}
```

3.2.3 Weighted Round Robin

Принцип роботи Weighted Round Robin є майже повністю ідентичним як і Round Robin. Реалізація алгоритму все таки має певні відмінності.

Як бачимо з'явилося нове значення requestCount, яке рахує скільки запитів виконав сервер, коли черга дійшла до нього. Також всередині відбувається виклик методу initRequestPerServer, для того щоб ініціалізувати кількість запитів на кінцеву точку.

```
constructor(type) {  
  super(type);  
  this.currentIndex = 0;  
  this.requestCount = 0;  
  this.depth = 0;  
  this.initRequestPerServer();  
}
```

Функція initRequestPerServer ініціалізує для кожного серверу з пулу параметр requestWeight, який означає скільки запитів він повинен обробити під своєї черги. Для обчислення цього статичного значення потрібно поділити вагу серверу(яку треба вказати в налаштуваннях) на довжину пулу, потім округлюємо це значення до цілого вгору.

```
initRequestPerServer() {  
  this.servers = this.servers.map((server) => {  
    server.requestWeight = Math.ceil(server.weight /  
    this.servers.length);  
    return server;  
  });  
}
```

Даний алгоритм, має ідентичний з Round Robin метод setNextInstance. Попри це функції next та pick мають трохи відмінну логіку.

					ІАЛЦ.467200.003 ПЗ	Арк.
						80
Зм.	Арк.	№ докум.	Підпис	Дата		

В середині next знову ж таки витягується наступний сервер з масиву, але потім йде перевірка на те скільки задач він виконав. Якщо кількість призначених йому запитів є виконаною, то беремо наступний інстанс, який буде мати ту ж саму логіку. Але якщо сервер все ще має можливість виконати певну кількість задач, то обираємо саме його та збільшуємо лічильник запитів на одиницю.

```
next() {
  const currentInstance = this.servers[this.currentIndex];

  if (currentInstance.requestWeight === this.requestCount + 1) {
    this._setNextInstance();
    this.requestCount = 0;
  } else {
    this.requestCount += 1;
  }

  return currentInstance;
}
```

В середині методу pick лише додається одна перевірка, для того щоб у разі несправності кінцевої точки, одразу переключити значення індексу вперед(взяти наступний сервер).

```
if (instance.active) {
  this.depth = 0;
  return instance;
}

if (this.requestCount !== 0) {
  this._setNextInstance();
}
```

Методи up та down не несуть ніякої логіки, тому є аналогічними.

3.2.3 Least Connection

Для реалізації даного алгоритму, найперше нам потрібно створити для кожного об'єкту серверу поле connections, яке показує кількість з'єднань для даного серверу(саме тому LC призначений для сокетів, які тримають довготривалий зв'язок з клієнтами).

В конструкторі класу викликається метод `initInstances`, який встановлює для кожного серверу початкову кількість активних підключень – нуль.

```
constructor(type) {  
  super(type);  
  this.initInstances();  
}  
  
initInstances() {  
  this.servers = this.servers.map((server) => ({ ...server,  
    connections: 0 }));  
}
```

Метод `pick` на початку роботи викликає функцію, яку наслідують від `ServerPool` – `getAvailableServers`. На цьому етапі отримуємо всі активні сервери в даний момент часу. Якщо кількість цих інстансів дорівнює нулю, то повертаємо `null`. При наявності здорових кінцевих точок, для них (масиву) викликається влаштована в Javascript функція `reduce`. Аргументами є параметри акумулятор (попереднє значення масиву) та значення на даному етапі ітерації. Тобто циклом відбувається прохід по списку, а на для кожної пари сусідніх серверів ми повертаємо той, де кількість підключень є найменшою (потім це значення йде в акумулятор). Таким чином ми отримуємо кінцеву точку, що відповідає умовам алгоритму.

```
pick() {  
  try {  
    const activeServers = this.getAvailableServers();  
  
    if (activeServers.length === 0) {  
      return null;  
    }  
  
    return activeServers.reduce((acc, val) =>  
      acc.connections > val.connections ? val : acc,  
    );  
  } catch (e) {  
    logger.error(e);  
  }  
}
```

На момент, коли кінцева точка встановить з'єднання, в контролері відбудеться виклик методу `up`, який збільшить кількість підключень на

					ІАЛЦ.467200.003 ПЗ	Арк.
						82
Зм.	Арк.	№ докум.	Підпис	Дата		

одиницю. Індекс для даного серверу знаходиться за допомогою функції `getServerIndex`(наслідувана від класу `ServerPool`).

```
up(instance) {  
    const index = this.getServerIndex(instance);  
  
    instance.connections += 1;  
    this.servers[index] = instance;  
}
```

Якщо сервер розриває з'єднання, то кількість з'єднань серверу зменшиться на одиницю.

```
down(instance) {  
    const index = this.getServerIndex(instance);  
  
    instance.connections -= 1;  
    this.servers[index] = instance;  
}
```

3.2.4 Weighted Least Connection

Даний алгоритм фактично немає різниці з `Least Connection`, окрім того що для отримання наступного серверу також використовується його вага(разом з кількістю підключень), яку прописує користувач в налаштуваннях.

Єдиною відмінністю в реалізації є те, що при перевірці сусідів із списку активних точок, відбувається обчислення коефіцієнту на основі ділення кількості підключень серверу на його вагу.

```
return activeServers.reduce((acc, val) =>  
    acc.connections / acc.weight > val.connections / val.weight ? val :  
    acc,  
);
```

3.2.5 Weighted Response Time

Даний алгоритм залежний лише від одного параметру серверу – час відповіді на запит, який обчислюється під час перевірки здоров'я. Це все значно полегшує його реалізацію, адже окрім вибірки точки з найменшим часом відповіді та перевірки наявності активних серверів, не потрібно робити ніяких інших кроків.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		83

```

pick() {
  try {
    const activeServers = this.getAvailableServers();

    if (activeServers.length === 0) {
      return null;
    }

    return activeServers.reduce((acc, val) =>
      acc.responseTime > val.responseTime ? val : acc,
    );
  } catch (e) {
    logger.error(e);
  }
}

```

3.2.6 IP Hash та Url Hash

IP Hash та Url Hash – це алгоритми, які потребують у кешуванні даних. Для кожної ip чи url адреси(з хешом) потрібно зберігати список серверів, які обробляти запити від даного клієнта.

Реалізація даних алгоритмів має деякі відмінності від описаних вище методів пошуку наступного активного серверу. Вони не наслідують клас ServerPool, замість цього присутня їхня композиція(конструкція схожа з наслідуванням, але при цьому ініціалізація батьківського класу проходить в конструкторі) з Round Robin, адже розподіл кінцевих точок по клієнтам відбувається циклічно. Також, якщо б дані класи наслідували ServerPool, то його створення відбулося б двічі і при цьому можливо отримати різні значення в пулах серверів.

Окрім Round Robin, в конструкторі відбувається ініціалізація класу CacheWorker, представлення секретного ключу для створення хешу, а також прив'язка функцій Round Robin до даного об'єктного контексту, для того щоб пізніше викликати їх як методи реалізованих класів.

```

constructor(type) {
  this.secret = 'ipHash';
  this.queue = new RoundRobin(type);
  this.cache = new CacheWorker();
  this.init = this.queue.init.bind(this.queue);
  this.activeCheck = this.queue.activeCheck.bind(this.queue);
}

```

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		84

Як і в інших алгоритмах, функція `next` повертає наступний сервер для виконання запиту. Перш за все відбувається циклічний прохід по серверам, які прив'язані до даної адреси. Як тільки ми отримуємо активний інстанс, то ми його повертаємо до відповідного контролеру. Якщо кінцеві точки неактивні, то `RoundRobin` дістає новий сервер та за умови, що він не рівний `null` (немає активних) додаємо його в список серверів для даної адреси (зберігає в кеші).

```
next(servers, hash) {
  for (let i = 0; i < servers.length; i++) {
    const index = this.queue.getServerIndex(servers[i]);
    const instance = this.queue.servers[index];
    if (instance.active) return instance;
  }

  const nextInstance = this.queue.pick();
  if (nextInstance) {
    servers.push(nextInstance);
    this.cache.set(hash, servers);
  }

  return nextInstance;
}
```

Представлений метод спочатку знаходить хеш адреси, який є ключем для зберігання даних. Після цього проходить перевірка чи є в нашому кеші значення за даним ключем. Якщо так, то ми дістаємо пул серверів, що буде обробляти запит та закінчуємо виконання викликом функції `next`. Якщо за хеш адресою немає ніяких записів в `CacheWorker`, то циклічно отримуємо наступну точку та віддаємо її контролеру (якщо значення не `null`, записуємо значення в кеш).

```
pick({ ip }) {
  try {
    const hash = utils.hash(ip, this.secret);
    const isHash = this.cache.has(hash);

    if (isHash) {
      const servers = this.cache.get(hash);
      return this.next(servers, hash);
    }

    const instance = this.queue.pick();
    if (instance) this.cache.set(hash, [instance]);
  }
}
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						85
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    return instance;
  } catch (e) {
    logger.error(e);
  }
}

```

Єдина відмінність Url Hash це те, що хешуються різні параметри, а також те, що при використанні url адреси, від неї беруться лише перші 80 байтів(значення може бути доволі громістким).

```

prepareUrl(url) {
  const parsedUrl = Url.parse(url).pathname;
  return parsedUrl.substr(0, 80);
}

pick({ url }) {
  try {
    const preparedUrl = this.prepareUrl(url);
    const hash = utils.hash(preparedUrl, this.secret);
    const isHash = this.cache.has(hash);

    if (isHash) {
      const servers = this.cache.get(hash);
      return this.next(servers, hash);
    }

    const instance = this.queue.pick();
    if (instance) this.cache.set(hash, [instance]);

    return instance;
  } catch (e) {
    logger.error(e);
  }
}

```

Функція хешування реалізована з використанням влаштованої в Node.js бібліотеки crypto, з використанням алгоритму sha256.

```

'use strict';

const crypto = require('crypto');

const hash = (value, secret) =>
  crypto.createHmac('sha256', secret).update(value).digest('hex');

```

3.2.7 Random

З назви алгоритму стає зрозуміло, що нам потрібно щоразу(на кожний запит) обирати випадковий сервер з пулу.

					ІАЛЦ.467200.003 ПЗ	Арк.
						86
Зм.	Арк.	№ докум.	Підпис	Дата		

Для цього було реалізовано метод `getRandomIndex`, який повертає індекс для масиву серверів.

```
getRandomIndex() {  
    const availableServers = this.getAvailableServers();  
    const limit = availableServers.length - 1;  
  
    return Math.floor(Math.random() * limit);  
}
```

Функція `next` викликає `getRandomIndex`(значення передано до змінної `index`) та повертає кінцеву точку за знайденим індексом.

```
next() {  
    const randomIndex = this.getRandomIndex();  
  
    return this.servers[randomIndex];  
}
```

`Pick` перевіряє наявність активних серверів і якщо такі є, то передає наступний випадковий інстанс до контролеру.

```
pick() {  
    try {  
        const availableServers = this.getAvailableServers();  
        if (availableServers.length === 0) {  
            return null;  
        }  
  
        return this.next();  
    } catch (e) {  
        logger.error(e);  
    }  
}
```

3.2.8 Resource based(Adaptive)

Єдиний алгоритм реального часу в реалізованому балансувальнику. Після кожного запиту, на основі часу відповіді, потрібно оновлювати значення EWMA:

$$EWMA(t) = \alpha * r(t) + (1 - \alpha) * EWMA(t - 1)$$

Таким чином, знаходження наступної кінцевої точки потребує даних від контролера, який щоразу заміряє час відповіді.

					ІАЛЦ.467200.003 ПЗ	Арк.
						87
Зм.	Арк.	№ докум.	Підпис	Дата		

В конструкторі класу оголошується статичне значення `alpha`(в нашому випадку 0.15), а також викликається метод `initInstances`, який записує початкові значення `ewma` для кожного серверу з пулу – 0. Тобто в будь якому разі перші запити використають весь набір кінцевих точок(якщо вони активні).

```
constructor(type) {  
  super(type);  
  
  this.alpha = 0.15;  
  this.initInstances();  
}
```

метод `initInstances`

```
initInstances() {  
  this.servers = this.servers.map((server) => ({ ...server, ewma: 0  
}));  
}
```

Функція `pick` повертає або `null`(немає активних серверів), або точку що має найменше значення `ewma`.

```
pick() {  
  const activeServers = this.getAvailableServers();  
  
  if (activeServers.length === 0) return null;  
  return activeServers.reduce((acc, val) => (acc.ewma > val.ewma ? val  
: acc));  
}
```

Оновлення `ewma` для відповідного серверу на основі часу відповіді.

```
up(instance, data = {}) {  
  const { responseTime } = data;  
  const index = this.getServerIndex(instance);  
  this.servers[index].ewma =  
    this.alpha * responseTime + (1 - this.alpha) * instance.ewma;  
}
```

Варто зазначити те, що реалізація даного алгоритму в Node.js вийшла доволі простою, але через постійний підрахунок часу – енергозатратною.

3.3 Контролер

3.1.2 BaseController

Даний клас, є батьківським для `HttpController` та `WsController`. По суті він не несе в собі ніякої логіки і є абстрактним. Основним його завданням є описати методи, які повинні бути реалізованими в класах, що його наслідують. Якщо вони не представлені то ці функції з `BaseController` будуть повертати помилку. Дана конструкція є прикладом реалізації поліморфізму в Javascript.

Реалізація класу *BaseController*

```
'use strict';

class BaseController {
  constructor(queue) {
    const proto = Object.getPrototypeOf(this);
    if (proto.constructor === BaseController) {
      throw new Error('Abstract class');
    }

    this.queue = queue;
  }

  handler() {
    throw new Error('Not implement');
  }
}

module.exports = BaseController;
```

3.1.2 HttpController

При ініціалізації програми до функції `createServer` було передано колбек функцію класу `HttpController` – `handler`. З цього моменту вона стає обробником всіх `http` та `https` запитів, що надходять до серверу балансеру. Вона має вхідні параметри `request` та `response`, які одразу записано до властивостей даного класу. Також на основі цих даних формуються значення `ip` та `url` адрес. Використовуючи ці параметри(які в залежності від алгоритму можуть не знадобитись), спрацьовує метод черги `pick`, результат якого стає значенням серверу, що обробить запит. Наприкінці, за допомогою конструкції `async ... await` викликається метод класу `exchange`, який несе в собі основну логіку по

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		89

балансуванні даних. При цьому щоразу спрацьовує метод черги `activeCheck`, який буде оновлювати інформацію про стан серверів, якщо в налаштуваннях вказаний активний тип перевірки. Також відбувається вибір протоколу, що буде призначений для запитів, в залежності від параметру `secure.up`.

```
handler = async (request, response) => {
  try {
    await this.queue.activeCheck();
    this.request = request;
    this.response = response;

    const url = request.url;
    const ip = request.socket.localAddress;
    const { up } = config.http.secure;

    this.server = await this.queue.pick({ url, ip });
    this.protocol = up ? https : http;

    await this.exchange();
  } catch (e) {
    await this.reply('Load balancing error', 500);
    logger.error(e);
  }
};
```

В даному методі на початку проходить перевірка чи є сервер обраний чергою для запиту об'єктом. Якщо ні, то це означає що в даний момент часу немає активних серверів в пулі. При цьому відправляємо клієнту відповідь *Error: No available servers* з статус кодом 404. При успішному виборі кінцевої точки проходить налаштування параметрів запиту до неї, а також формуємо `body` на основі запиту від клієнту. Після цього викликаємо метод `requestToServer`. Також, якщо в черзі функціонує адаптивний алгоритм, відбувається підрахунок часу відповіді, що потім передається до функції черги `up`(для оновлення значення `ewma`).

```
exchange = async () => {
  try {
    if (!this.server) {
      await this.reply('Error: No available servers', 404);
      return;
    }

    const options = await this.prepareOption();
    const body = await this.finishTransaction(this.request);
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						90
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    if (config.http.algorithm && config.http.algorithm === 'Adaptive')
    {
        const startTime = new Date().getTime();
        this.requestToServer(options, body);
        const endTime = new Date().getTime();
        this.queue.up(this.server, {responseTime: endTime - startTime});
    } else {
        this.requestToServer(options, body);
    }
    } catch (e) {
        return Promise.reject(e);
    }
}

```

Асинхронна функція requestToServer виконує клієнтський запит до обраного серверу, при цьому навішуємо на запит обробку помилок, а передаємо до нього тіло клієнтської задачі. Обробка відповіді проходить в колбек функції this.protocol.request, яка формує тіло відповіді та повертає його клієнту, разом із статусом кодом та відповідними хедерами.

```

requestToServer = async (options, body) => {
    return new Promise((resolve, reject) => {
        try {
            const request = this.protocol.request(options, async (response)
=> {
                try {
                    const data = await this.finishTransaction(response);
                    await this.reply(data, response.statusCode,
response.headers);
                } catch (e) {
                    reject(e);
                }
            });

            this.errorTransaction(request);
            if (body) request.write(body);
            request.end();
        } catch (e) {
            reject(e);
        }
    });
}

```

Функція finishTransaction виконує роль формування зчитування body(дані що потрібно відправити до відділеного ресурсу), що надходить в запиті або відповіді. Спочатку фрагменти потоку даних записуються в масив chunks. На

					ІАЛЦ.467200.003 ПЗ	Арк.
						91
Зм.	Арк.	№ докум.	Підпис	Дата		

момент, коли потік закінчується, буфер формує реальний об'єкт даних, що відправляється в запиті чи повертається у відповіді.

```
finishTransaction = async (transaction) => {
  return new Promise((resolve, reject) => {
    const chunks = [];
    transaction.on('data', (chunk) => {
      try {
        chunks.push(chunk);
      } catch (e) {
        reject(e);
      }
    });

    transaction.on('end', () => {
      try {
        const body = Buffer.concat(chunks);
        resolve(body);
      } catch (e) {
        reject(e);
      }
    });
  });
}
```

Представлений нижче момент надсилає відповідь клієнту. Параметрами є тіло відповіді, статус код та хедери.

```
reply = async (body, statusCode, headers = {}) => {
  try {
    this.response.writeHead(statusCode, headers);
    this.response.end(body);
  } catch (e) {
    return Promise.reject(e);
  }
}
```

В методі errorTransaction навішується обробник помилки при відправці запиту на сервер. При цьому надсилається відповідь клієнту, де тіло – це помилка, а статус код – 500.

```
errorTransaction = (transaction) => {
  transaction.on('error', async (err) => {
    await this.reply(err, 500);
  });
}
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						92
Зм.	Арк.	№ докум.	Підпис	Дата		

Налаштування параметрів запиту до кінцевої точки проходить у функції `prepareOption`. Також якщо користувач вмикає параметр `secure` для пулу серверів, то до запиту додаються їхні параметри безпеки(ssl).

```
prepareOption = async () => {
  try {
    let otherParams = {};
    const { ssl, up } = config.http.secure;
    if (up) otherParams = ssl;

    return {
      hostname: this.server.host,
      port: this.server.port,
      path: this.request.url,
      method: this.request.method,
      headers: this.request.headers,
      ...otherParams
    };
  } catch (e) {
    return Promise.reject(e);
  }
}
```

Як бачимо, кожний метод класу обгорнуто в конструкцію `try ... catch`(навіть колбек функції). Це зроблено для того, щоб у разі будь-якої помилки, її викинуло з методу. Таким чином кінцева точка, яка буде проводити обробку ситуації – функція `handler`, яка поверне відповідне повідомлення до клієнту, а також виведе це в логи для інформування.

3.1.3 WsController

Даний контролер(клас) призначений для обробки двохспрямованого зв'язку, який встановлюється за допомогою протоколу `WebSocket`. Даний протокол має в своїй основі `http`, саме тому `WS` запит може обробляти як звичайний `HTTP`, але із заголовком `UPGRADE`. В `Node.js` є можливість встановити метод, який буде викликатися щоразу, коли отримає запит із даним заголовком. Ця ініціалізація відбулася в класі `LoadBalancer`:

```
server.on('upgrade', this.ws.handler);
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						93
Зм.	Арк.	№ докум.	Підпис	Дата		

Метод handler спочатку отримує параметри – запит, сокет та перший пакет вхідного потоку. Далі проходить виклик функції activeCheck та оновлення властивостей класу на основі вхідних параметрів. Логіка для вибору наступного серверу та протоколу для запиту до нього аналогічна з HttpController. Після цього відбувається виклик exchange, де відбувається основний процес балансування сокетів.

```
handler = async (request, socket, head) => {
  try {
    await this.queue.activeCheck();
    this.request = request;
    this.socket = socket;
    this.head = head;

    const url = request.url;
    const ip = request.socket.localAddress;
    const { up } = config.ws.secure;

    this.server = await this.queue.pick({ url, ip });
    this.protocol = up ? https : http;

    await this.exchange();
  } catch (e) {
    this.closeSocket();
    logger.error(e);
  }
}
```

Функція exchange перш за все проводить перевірку на те, чи існує активний сервер на даний момент часу. Якщо немає, то закриваємо сокет з'єднання, яке приєдналося до балансеру. В іншому випадку, проходить ініціалізація довготривалого зв'язку між клієнтом та сервером. Також перший пакет вхідного потоку записується в початок усього потоку сокету. Потім проходить підбір опцій та власне сам запит до кінцевої точки. Після цього встановлюються слухачі на різні події для даного зв'язку та завершується його виконання.

```
exchange = async () => {
  try {
    if (!this.server) {
      this.closeSocket();
      return;
    }
  }
}
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						94
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    await this.setupSocket();
    await this.unshiftHead(this.socket, this.head);

    const options = await this.prepareOption();
    const connection = this.protocol.request(options);

    this.socketListener(connection);
    this.responseListener(connection);
    this.errorListener(connection, this.socket);
    this.upgradeListener(connection);

    return connection.end();
  } catch (e) {
    return Promise.reject(e);
  }
}

```

Якщо кінцева точка працює на протоколі WS та повертає відповідний потік, викликаємо метод черги up. На цей момент відбудеться певне оновлення стану серверу(наприклад для LeastConnection кількість з'єднань збільшиться на одиницю).

```

socketListener = (connection) => {
  return new Promise((resolve, reject) => {
    connection.on('socket', () => {
      try {
        this.queue.up(this.server);
      } catch (e) {
        reject(e);
      }
    });
  });
}

```

При помилці даного запиту закривається клієнтське з'єднання та викликається callback, яка за стандартом є пустою функцією.

```

errorListener = (connection, socket, callback = () => {}) => {
  return new Promise((resolve, reject) => {
    connection.on('error', () => {
      try {
        socket.end();
        callback();
      } catch (e) {
        reject(e);
      }
    });
  });
}

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						95
Зм.	Арк.	№ докум.	Підпис	Дата		

Під час події response перевіряються заголовки відповіді і якщо вона не містить upgrade, то формуємо заголовки(на основі відповіді) та записуємо відповідний потік до клієнтського потоку(response.pipe(socket)). Таким чином у нас не відбудеться рукостискання між віддаленими точками. Якщо upgrade заголовок присутній, то цьому моменті нічого не відбудеться.

```
responseListener = (connection) => {
  return new Promise((resolve, reject) => {
    connection.on('response', async (response) => {
      try {
        if (!response.upgrade) {
          const headLine = `HTTP/${response.httpVersion}
${response.statusCode} ${response.statusMessage}`;
          const headers = response.headers;
          const header = await this.createHeader(headLine,
headers);

          this.socket.write(header);
          response.pipe(this.socket);
        }
      } catch (e) {
        reject(e);
      }
    });
  });
}
```

Подія upgrade означає що сервер повернув відповідь з однойменним заголовком. Це означає, що кінцева точка має можливість обробляти WS трафік. Параметрами колбек функції будуть: відповідь та сокет від серверу та перший пакет вхідного потоку. При цьому відбувається встановлення функцій слухачів помилок для обох сторін(під час цього спрацює функція down для черги), запис першого пакету назад в основний потік, а також налаштування сокету, що прийшов від бекенду на довготривалий зв'язок. Потім до клієнтського сокету записуються заголовки, першим рядком яких буде – “HTTP/1.1 101 Switching Protocol”, що вказує на те що рукостискання відбулося(встановлення двохстороннього зв'язку). Наприкінці відбувається створення так званого “тунелю” між клієнтом та сервером. Серверний потік записується до клієнтського, і навпаки.

					ІАЛЦ.467200.003 ПЗ	Арк.
						96
Зм.	Арк.	№ докум.	Підпис	Дата		

```

upgradeListener = (connection) => {
    return new Promise((resolve, reject) => {
        connection.on('upgrade', async (connectionResponse,
connectionSocket, connectionHead) => {
            try {
                const serverDown = () => this.queue.down(this.server);
                this.errorListener(connectionSocket, this.socket,
serverDown);
                this.errorListener(this.socket, connectionSocket,
serverDown);
                this.setupSocket(connectionSocket);
                await this.unshiftHead(connectionSocket,
connectionHead);

                const headLine = 'HTTP/1.1 101 Switching Protocols';
                const headers = connectionResponse.headers;
                const header = await this.createHeader(headLine,
headers);
                this.socket.write(header);

                connectionSocket.pipe(this.socket).pipe(connectionSocket);
            } catch (e) {
                reject(e);
            }
        });
    });
};

```

Встановлення довготривалого сокету: відключення параметру простою та включення функції перевірки активності.

```

setupSocket = async () => {
    try {
        this.socket.setTimeout(0);
        this.socket.setNoDelay(true);
        this.socket.setKeepAlive(true, 0);
    } catch (e) {
        return Promise.reject(e);
    }
};

```

Закриття з'єднання з сокетом, якщо він існує:

```

closeSocket = () => {
    if (this.socket) this.socket.end();
}

```

Запис першого пакету до основного потоку:

```

unshiftHead = async (socket, head) => {
    try {
        if (head && head.length) socket.unshift(head);
    }
}

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						97
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    } catch (e) {
      return Promise.reject(e);
    }
  }
}

```

Генерування заголовку на основі першого рядку(параметр headline) та об'єкту headers.

```

createHeader = async (headLine, headers) => {
  try {
    const values = Object.keys(headers).map(key => (`${key}:
    ${headers[key]}`));
    values.unshift(headLine);
    return `${values.join('\r\n')}\r\n\r\n`;
  } catch (e) {
    return Promise.reject(e);
  }
};

```

Метод prepareOption проводить налаштування параметрів для запиту до віддаленого ресурсу. Аналогічно як і в HttpController, при включенні параметру безпеки, її параметри додаються до запиту. В цьому випадку означає, що кінцевий сервер працює на протоколі WSS(WebSocket Secure).

```

prepareOption = async () => {
  try {
    let otherParams = {};
    const { ssl, up } = config.ws.secure;
    if (up) otherParams = ssl;

    return {
      port: this.server.port,
      method: this.request.method,
      headers: this.request.headers,
      path: this.request.url,
      ...otherParams
    };
  } catch (e) {
    return Promise.reject(e);
  }
}

```

Знову ж таки, усі без винятку функції класу, разом із колбеками обгорнуті в try ... catch для відловлювання помилки та прокидання її до методу handler, який при цьому закрий з'єднання з клієнтом та виведе відповідні логи в консоль.

					ІАЛЦ.467200.003 ПЗ	Арк.
						98
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ ДО РОЗДІЛУ 3.

В даному розділі було представлено реалізацію прикладного балансувальника навантаження на платформі Node.js.

Кінцева версія сервісу має достатньо широкий функціонал, що робить систему досить гнучною для вирішення різних типів задач.

Розроблений балансер дає змогу для обробки та перенаправлення http та ws трафіку до пулу кінцевих точок. При цьому користувач має змогу налаштовувати параметри безпеки системи, такі як захист від DDoS атак та рівень безпечних сокетів(SSL). Окрім цього в балансер було вбудовано систему кешування та логування.

Також було представлено реалізацію дев'ятьох вбудованих алгоритмів балансування, які можливо легко обрати у файлі конфігурації. В основі вибору наступної точки знаходиться система перевірка активності вхідних серверів. Було реалізовано кілька режимів її роботи – активна та пасивна.

Під час проектування системи основною задачею було розробити балансер, який зможе виконувати задачі для різних типів систем та налаштувань.

При написанні коду було використано сучасні патерни програмування, а остання версія системи є доволі гнучкою і придатною до майбутніх змін. При цьому не було використано жодної сторонньої бібліотеки, а лише ті, що є вбудованими в Node.js.

					ІАЛЦ.467200.003 ПЗ	Арк.
						99
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 4

ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Компоненти

4.1.1 Контролер

Для демонстрації роботи балансеру було реалізовано прості Http та Ws сервери, що в даному випадку будуть кінцевими точками для клієнтського запиту.

Http

Нижче представлено API тестового Http серверу, який буде мати 3 інстанси на портах 8000, 8001, 8002.

```
'use strict';

const controller = require('../controllers');
const service = require('../application');

const routes = [
  {
    method: 'GET',
    url: '/teams',
    handler: controller(service.getTeams),
  },
  {
    method: 'POST',
    url: '/teams',
    handler: controller(service.createTeam),
  },
  {
    method: 'DELETE',
    url: '/teams/:id',
    handler: controller(service.deleteTeam),
  },
  {
    method: 'PUT',
    url: '/teams/:id',
    handler: controller(service.updateTeam),
  },
];

module.exports = routes;
```

Налаштування для http контролеру:

					ІАЛЦ.467200.003 ПЗ	Арк.
						100
Зм.	Арк.	№ докум.	Підпис	Дата		

```
'use strict';

module.exports = {
  servers: [
    { host: '127.0.0.1', port: 8000 },
    { host: '127.0.0.1', port: 8001 },
    { host: '127.0.0.1', port: 8002 },
  ],
  algorithm: 'RoundRobin',
  healthCheck: {
    activeCheck: false,
    path: '/teams',
    time: 60,
  },
  secure: {
    up: false,
  },
};
```

Початок роботи балансеру. Можна побачити, що одразу відбулася перевірка активності серверів, адже згідно з налаштуваннями, працює пасивний режим.

```
PS C:\Users\slava\Documents\load_balancer\src> node .\index.js
INFO | Start load balancer listen on 3000 | timestamp: Sat Jun 04 2022 17:52:32 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sat Jun 04 2022 17:52:32 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sat Jun 04 2022 17:52:32 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sat Jun 04 2022 17:52:32 GMT+0300 (GMT+03:00)
```

Рис. 4.1 Перевірка стану серверів

Виконання чотирьох основних запитів – GET, POST, PUT та DELETE. Порт – 3000(балансер).

```
'use strict';

const request = require('request');

request(
  'http://localhost:3000/teams',
  { json: true, method: 'GET', body: null },
  (err, res, body) => {
    if (err) {
      console.log(err);
      return;
    }
    console.log('==GET==');
    console.log(`Status code: ${res.statusCode}`);
    console.log(JSON.stringify(body));
  },
);
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						101
Зм.	Арк.	№ докум.	Підпис	Дата		

```
request(
  'http://localhost:3000/teams',
  { json: true, method: 'POST', body: { name: 'Barcelona' } },
  (err, res, body) => {
    if (err) {
      console.log(err);
      return;
    }
    console.log('==POST==');
    console.log(`Status code: ${res.statusCode}`);
    console.log(JSON.stringify(body));
  },
);

request(
  'http://localhost:3000/teams/43',
  { json: true, method: 'PUT', body: { name: 'Real Madrid' } },
  (err, res, body) => {
    if (err) {
      console.log(err);
      return;
    }
    console.log('==PUT==');
    console.log(`Status code: ${res.statusCode}`);
  },
);

request(
  'http://localhost:3000/teams/43',
  { json: true, method: 'DELETE', body: null },
  (err, res, body) => {
    if (err) {
      console.log(err);
      return;
    }
    console.log('==DELETE==');
    console.log(`Status code: ${res.statusCode}`);
  },
);
```

Відповідь
GET

```
==GET==
Status code: 200
[{"id":4,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":5,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":6,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":7,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":8,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":9,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":10,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":11,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":12,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":13,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":14,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":15,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":16,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":17,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":18,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":19,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":20,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":21,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":22,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":23,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":24,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":25,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":26,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":27,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":28,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":29,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":30,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":31,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":32,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":33,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":34,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":35,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":36,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":37,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":38,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":39,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":40,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}]
```

Рис. 4.2 Відповідь GET

POST

```
==POST==  
Status code: 200  
{ "id": 43, "name": "Barcelona", "goalScored": 0, "goalConceded": 0, "points": 0 }
```

Рис. 4.3 Відповідь POST

PUT

```
==PUT==  
Status code: 200
```

Рис. 4.3 Відповідь PUT

DELETE

```
==DELETE==  
Status code: 200
```

Рис. 4.4 Відповідь DELETE

Логи балансеру:

```
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sat Jun 04 2022 18:34:19 GMT+0300 (GMT+03:00)  
INFO | Send request POST /teams to 127.0.0.1:8002 | timestamp: Sat Jun 04 2022 18:34:47 GMT+0300 (GMT+03:00)  
INFO | Send request PUT /teams/43 to 127.0.0.1:8000 | timestamp: Sat Jun 04 2022 18:35:04 GMT+0300 (GMT+03:00)  
INFO | Send request DELETE /teams/43 to 127.0.0.1:8001 | timestamp: Sat Jun 04 2022 18:35:23 GMT+0300 (GMT+03:00)
```

Рис. 4.5 Перенаправлені запити

WS

WebSocket сервер, який ініціалізовано на портах 8003, 8004, 8005.

```
const wss = new WebSocketServer.Server({ port: i });  
  
wss.on('connection', (ws) => {  
  console.log('Starting new connection');  
  // sending message  
  ws.on('message', (data) => {  
    ws.send(`${data} by tester`);  
  });  
});
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						103
Зм.	Арк.	№ докум.	Підпис	Дата		

```

});
ws.onerror = () => {
  console.log('Some Error occurred');
};
});

```

Налаштування контролеру для обробки сокетів:

```

'use strict';

module.exports = {
  servers: [
    { host: '127.0.0.1', port: 8003 },
    { host: '127.0.0.1', port: 8004 },
    { host: '127.0.0.1', port: 8005 },
  ],
  algorithm: 'RoundRobin',
  healthCheck: {
    activeCheck: false,
    path: '/',
    time: 60,
  },
  secure: {
    up: false,
  },
};

```

Створення клієнту, що буде з'єднуватись з віддаленим ресурсом. При цьому між ними приводиться обмін даними(після рукошестикання). Для тесту скрипт було запущено кілька разів.

```

const WebSocket = require('ws');

const socket = new WebSocket('ws://localhost:3000');

socket.onopen = (e) => {
  socket.send('Load balancer test');
};

socket.onmessage = (event) => {
  console.log(event.data);
};

socket.onerror = (error) => {
  console.log(error);
};

```

WS серверів після встановлення з'єднання:

					ІАЛЦ.467200.003 ПЗ	Арк.
						104
Зм.	Арк.	№ докум.	Підпис	Дата		

```
PS C:\Users\slava\Documents\test> node .\ws.js
Starting new connection
Starting new connection
Starting new connection
```

Рис. 4.6 WS сервер

Клієнт після обміну даними:

```
PS C:\Users\slava\Documents\test> node .\test.js
Load balancer test by tester
PS C:\Users\slava\Documents\test> node .\test.js
Load balancer test by tester
PS C:\Users\slava\Documents\test> node .\test.js
Load balancer test by tester
```

Рис. 4.7 WS клієнт

Логи балансеру:

```
INFO | Start load balancer listen on 3000 | timestamp: Sun Jun 05 2022 00:53:45 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 00:53:45 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 00:53:45 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 00:53:45 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8003 | timestamp: Sun Jun 05 2022 00:53:48 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8004 | timestamp: Sun Jun 05 2022 00:53:57 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8005 | timestamp: Sun Jun 05 2022 00:53:59 GMT+0300 (GMT+03:00)
```

Рис. 4.9 Утворення з'єднання

Відповідно до тестів, можна побачити, що реалізований балансер вміє обробляти різні види трафіку(HTTP та WS).

4.1.2 Перевірка стану серверів

Пасивний режим

Http

					ІАЛЦ.467200.003 ПЗ	Арк.
						105
Зм.	Арк.	№ докум.	Підпис	Дата		

```
healthCheck: {
  activeCheck: false,
  path: '/healthCheck',
  time: 10, // seconds
},
```

Ws

```
healthCheck: {
  activeCheck: false,
  path: '/',
  time: 10,
},
```

На початку тестування всі доступні сервери працюють в штатному режимі.

```
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 02:07:07 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:07:07 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:07:07 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:07:07 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 02:07:07 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:07:07 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 02:07:17 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:07:17 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:07:17 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:07:17 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:07:17 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 02:07:17 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 02:07:27 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:07:27 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:07:27 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:07:27 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:07:27 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 02:07:27 GMT+0300 (GMT+03:00)
```

Рис. 4.10 Перевірка стану серверів при пасивному режимі(перша ітерація)

Як бачимо, перевірка активності для http та ws проходить кожні 10 секунд(згідно з налаштувань).

Після цього виникає збій, через який зупиняється http сервер на 8002 порту та ws сервер на 8003.

```

INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:14:57 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:14:57 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:14:59 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:14:59 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:15:07 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:15:07 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:15:07 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:15:07 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:15:09 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:15:09 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:15:17 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:15:17 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:15:17 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:15:17 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:15:19 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:15:19 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:15:27 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:15:27 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:15:27 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:15:27 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:15:29 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:15:29 GMT+0300 (GMT+03:00)

```

Рис. 4.11 Перевірка стану серверів при пасивному режимі(друга ітерація)

Тепер в логуванні показано, що дані сервери не пройшли перевірку активності.

Активний режим

Параметр конфігурації для кожної черги:

```

healthCheck: {
  activeCheck: true,
},

```

Після перезапуску балансеру можна побачити, що перевірка стану кінцевих точок не проходить.

```

PS C:\Users\slava\Documents\load_balancer\src> node .\index.js
INFO | Start load balancer listen on 3000 | timestamp: Sun Jun 05 2022 02:18:39 GMT+0300 (GMT+03:00)

```

Рис. 4.12 Початок роботи балансеру

Далі відбувається відправка кількох запитів до балансувальника.

					ІАЛЦ.467200.003 ПЗ	Арк.
						107
Зм.	Арк.	№ докум.	Підпис	Дата		

```
PS C:\Users\slava\Documents\load_balancer\src> node .\index.js
INFO | Start load balancer listen on 3000 | timestamp: Sun Jun 05 2022 02:18:39 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:20:32 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:20:32 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:20:34 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 02:20:34 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 02:20:48 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 02:20:48 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:20:50 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 02:20:50 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:20:56 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:20:56 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:20:58 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8004 | timestamp: Sun Jun 05 2022 02:20:58 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 02:21:01 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 02:21:01 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:21:03 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8005 | timestamp: Sun Jun 05 2022 02:21:03 GMT+0300 (GMT+03:00)
```

Рис. 4.13 Перевірка стану серверів при активному режимі

Тепер перевірка активності проходить перед кожною обробкою вхідного трафіка. Даний режим сповільнює роботу системи, але якщо задача потребує гарантованої доставки даних, то дане рішення є більш надійним.

Спробуємо відключити всі доступні сервери та надіслати http та ws запити.

```
INFO | Health check: http://127.0.0.1:8001 NOT FOUND | timestamp: Sun Jun 05 2022 02:32:20 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 NOT FOUND | timestamp: Sun Jun 05 2022 02:32:20 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 NOT FOUND | timestamp: Sun Jun 05 2022 02:32:20 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 02:32:43 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 NOT FOUND | timestamp: Sun Jun 05 2022 02:32:43 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 NOT FOUND | timestamp: Sun Jun 05 2022 02:32:43 GMT+0300 (GMT+03:00)
```

Рис. 4.14 Відсутність активних серверів

Відповідь

Http

```
==GET==
Status code: 404
"Error: No available servers"
```

Рис. 4.15 Відповідь від Http контролеру

					ІАЛЦ.467200.003 ПЗ	Арк.
						108
Зм.	Арк.	№ докум.	Підпис	Дата		

Ws

```
[Symbol(kError)]: Error: socket hang up
  at connResetException (internal/errors.js:570:14)
  at Socket.socketOnEnd (_http_client.js:440:23)
  at Socket.emit (events.js:215:7)
  at endReadableNT (_stream_readable.js:1183:12)
  at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'ECONNRESET'
},
[Symbol(kMessage)]: 'socket hang up'
}
```

Рис. 4.16 Відповідь від Ws контролеру

Для http запиту приходить відповідь “Error: No available servers” із статус кодом 404. Для ws закривається сокет з’єднання.

4.1.3 Безпека

SSL

Для початку увімкнемо параметр рівня безпечних сокетів для серверу балансувальника, а також встановимо для нього SSL сертифікати:

```
secure: true,
ssl: {
  key: fs.readFileSync(`${__dirname}/certs/server-key.pem`),
  cert: fs.readFileSync(`${__dirname}/certs/server-crt.pem`),
  ca: [fs.readFileSync(`${__dirname}/certs/client-crt.pem`)],
  requestCert: true,
  rejectUnauthorized: true,
},
```

Після відправки звичайного Http запиту отримуємо відповідь про те, що з’єднання розірвано.

```
Error: socket hang up
    at connResetException (internal/errors.js:570:14)
    at Socket.socketOnEnd (_http_client.js:440:23)
    at Socket.emit (events.js:215:7)
    at endReadableNT (_stream_readable.js:1183:12)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'ECONNRESET'
}
```

Рис. 4.17 Розірвання з'єднання після невдалої автентифікації до HTTPS серверу

Тепер відправимо запит із відповідними сертифікатами, які використовуються для автентифікації з'єднання.

```
const https = require('https');
const fs = require('fs');

const options = {
  hostname: '127.0.0.1',
  port: 3000,
  path: '/teams',
  method: 'GET',

  rejectUnauthorized: true,
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-crt.pem'),
  ca: fs.readFileSync('server-crt.pem'),
};

const req = https.request(options, (res) => {
  res.on('data', (d) => process.stdout.write(d));
});

req.on('error', (e) => console.error(e));

req.end();
```

Тепер в логах можна побачити відповідь:

					ІАЛЦ.467200.003 ПЗ	Арк.
						110
Зм.	Арк.	№ докум.	Підпис	Дата		

```
[{"id":4,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":5,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":6,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":7,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":8,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":9,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":10,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":11,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":12,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":13,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":14,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":15,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":16,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":17,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":18,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":19,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":20,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":21,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":22,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":23,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":24,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":25,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":26,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":27,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":28,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":29,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":30,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":31,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":32,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":33,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":34,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":35,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":36,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":37,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":38,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}, {"id":39,"name":"Barcelona","goalScored":0,"goalConceded":0,"points":0}, {"id":40,"name":"Real Madrid","goalScored":0,"goalConceded":0,"points":0}]
```

Рис. 4.18 Отримання відповіді через зашифрований трафік

Захист від DDos атак

Увімкнення параметру захисту:

```
doss: true
```

Скрипт атаки:

```
const req = require('request');

setInterval(() => {
  for (let i = 1000; i--; ) {
    req(
      'http://localhost:3000/teams',
      { json: true, method: 'GET', body: null },
      (err, res, body) => {
        if (err) {
          console.log(err);
          return;
        }
        console.log(JSON.stringify(body));
      }
    );
  }
}, 1000);
```

Після його запуску, в певний момент, у відповідь почне приходити помилка ECONNRESET.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		111

```

Error: socket hang up
    at connResetException (internal/errors.js:570:14)
    at Socket.socketOnEnd (_http_client.js:440:23)
    at Socket.emit (events.js:215:7)
    at endReadableNT (_stream_readable.js:1183:12)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'ECONNRESET'
}
Error: socket hang up
    at connResetException (internal/errors.js:570:14)
    at Socket.socketOnEnd (_http_client.js:440:23)
    at Socket.emit (events.js:215:7)
    at endReadableNT (_stream_readable.js:1183:12)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'ECONNRESET'
}
Error: socket hang up
    at connResetException (internal/errors.js:570:14)
    at Socket.socketOnEnd (_http_client.js:440:23)
    at Socket.emit (events.js:215:7)
    at endReadableNT (_stream_readable.js:1183:12)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'ECONNRESET'
}

```

Рис. 4.19 Розірвання з'єднання після виявлення DDoS атаки

Після того, як кількість підозрілих підключень до балансера доходить до певної межі, він розриває з'єднання з клієнтом.

4.1.4 Кешування

Для тестування кешу було написано даний скрипт:

```

const cache = new CacheWorker();

cache.set('key1', 'test1');
cache.set('key2', 'test2', 2);

const hasValue1 = cache.has('key1');
const hasValue2 = cache.has('key2');

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						112
Зм.	Арк.	№ докум.	Підпис	Дата		

```

logger.info(`Cache has value with key1: ${hasValue1}`);
logger.info(`Cache has value with key2: ${hasValue2}`);

const value1 = cache.get('key1');
const value2 = cache.get('key2');

logger.info(`Value with key1: ${value1}`);
logger.info(`Value with key2: ${value2}`);

setTimeout(() => {
  logger.info('4 seconds have passed');

  const hasValue11 = cache.has('key1');
  const hasValue21 = cache.has('key2');

  logger.info(`Cache has value with key1: ${hasValue11}`);
  logger.info(`Cache has value with key2: ${hasValue21}`);

  const value11 = cache.get('key1');
  const value22 = cache.get('key2');

  logger.info(`Value with key1: ${value11}`);
  logger.info(`Value with key2: ${value22}`);
}, 4000);

```

Після його запуску до кешу записується 2 значення, одне з яких має час зберігання 2 секунди. Далі виконуються спроби дістати значення, а також перевірити його наявність.

```

PS C:\Users\slava\Documents\load_balancer\src\infrastructure\Cache> node .\index.js
INFO | Cache has value with key1: true | timestamp: Sun Jun 05 2022 13:02:41 GMT+0300 (GMT+03:00)
INFO | Cache has value with key2: true | timestamp: Sun Jun 05 2022 13:02:41 GMT+0300 (GMT+03:00)
INFO | Value with key1: test1 | timestamp: Sun Jun 05 2022 13:02:41 GMT+0300 (GMT+03:00)
INFO | Value with key2: test2 | timestamp: Sun Jun 05 2022 13:02:41 GMT+0300 (GMT+03:00)
INFO | 4 seconds have passed | timestamp: Sun Jun 05 2022 13:02:45 GMT+0300 (GMT+03:00)
INFO | Cache has value with key1: true | timestamp: Sun Jun 05 2022 13:02:45 GMT+0300 (GMT+03:00)
INFO | Cache has value with key2: false | timestamp: Sun Jun 05 2022 13:02:45 GMT+0300 (GMT+03:00)
INFO | Value with key1: test1 | timestamp: Sun Jun 05 2022 13:02:45 GMT+0300 (GMT+03:00)
INFO | Value with key2: undefined | timestamp: Sun Jun 05 2022 13:02:45 GMT+0300 (GMT+03:00)

```

Рис. 4.20 Тестування CacheWorker(вивід в консоль)

Як бачимо значення з ключем “key1” після 4 секунд залишається в системі, адже час його зберігання необмежений. В свою чергу, запис з “key2” зникає з пам’яті, так як час зберігання було вказано 2 секунди.

4.1.5 Логування

Прикладів використання логера вже було багато, але все таки варто ще раз продемонструвати його роботу.

Тестовий скрипт:

```
const logger = new Logger();

logger.info('Success');
logger.warning('Warning');
logger.error('Error');
```

Вивід в консоль:

```
PS C:\Users\slava\Documents\load_balancer\src\infrastructure\Logger> node .\index.js
INFO | Success | timestamp: Sun Jun 05 2022 13:46:04 GMT+0300 (GMT+03:00)
WARNING | Warning | timestamp: Sun Jun 05 2022 13:46:04 GMT+0300 (GMT+03:00)
ERROR | Error | timestamp: Sun Jun 05 2022 13:46:04 GMT+0300 (GMT+03:00)
```

Рис. 4.21 Робота логеру

Для кожного випадку вказується рівень повідомлення, яке має відповідний колір. Також можна побачити час виникнення певної події.

4.2 Алгоритми балансування

4.2.1 Round Robin

Пул серверів:

```
servers: [
  { host: '127.0.0.1', port: 8000 },
  { host: '127.0.0.1', port: 8001 },
  { host: '127.0.0.1', port: 8002 },
  { host: '127.0.0.1', port: 8003 },
]
```

Серед них сервер на 8003 порту є неактивним. Далі проходить відправка певної кількості запитів на отримання даних.

```

INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 14:26:46 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 14:26:47 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 14:26:47 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 14:26:49 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:26:49 GMT+0300 (GMT+03:00)
INFO | Send request GET /favicon.ico to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 14:26:49 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8002 | timestamp: Sun Jun 05 2022 14:26:50 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:26:51 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 14:26:51 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8002 | timestamp: Sun Jun 05 2022 14:26:52 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:26:52 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 14:26:53 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8002 | timestamp: Sun Jun 05 2022 14:26:53 GMT+0300 (GMT+03:00)

```

Рис. 4.22 Результати роботи Round Robin

З консолі можна побачити, що вхідний трафік циклічно перенаправляється по активним бекендам.

4.2.2 Weighted Round Robin

Пул серверів з відповідними вагами:

```

servers: [
  { host: '127.0.0.1', port: 8000, weight: 15 },
  { host: '127.0.0.1', port: 8001, weight: 10 },
  { host: '127.0.0.1', port: 8002, weight: 12 },
  { host: '127.0.0.1', port: 8003, weight: 20 },
],

```

Для тесту було додати додаткові логи, щоб показати скільки запитів було розподілено для кожного інстансу:

```

INFO | { host: 127.0.0.1, port: 8000, requestCount: 4 } | timestamp: Sun Jun 05 2022 14:37:07 GMT+0300 (GMT+03:00)
INFO | { host: 127.0.0.1, port: 8001, requestCount: 3 } | timestamp: Sun Jun 05 2022 14:37:07 GMT+0300 (GMT+03:00)
INFO | { host: 127.0.0.1, port: 8002, requestCount: 3 } | timestamp: Sun Jun 05 2022 14:37:07 GMT+0300 (GMT+03:00)
INFO | { host: 127.0.0.1, port: 8003, requestCount: 5 } | timestamp: Sun Jun 05 2022 14:37:07 GMT+0300 (GMT+03:00)

```

Рис. 4.23 Кількість запитів для кожного серверу

Значення requestCount показує скільки запитів буде обробляти сервер, коли до нього дійде черга.

					ІАЛЦ.467200.003 ПЗ	Арк.
						115
Зм.	Арк.	№ докум.	Підпис	Дата		

```

INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 14:45:45 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 14:45:45 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 14:45:45 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 14:45:47 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:45:49 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:45:49 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:45:49 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:45:50 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 14:45:50 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 14:45:50 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 14:45:51 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8002 | timestamp: Sun Jun 05 2022 14:45:51 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8002 | timestamp: Sun Jun 05 2022 14:45:51 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8002 | timestamp: Sun Jun 05 2022 14:45:52 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:45:52 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 14:45:52 GMT+0300 (GMT+03:00)

```

Рис. 4.24 Результати роботи Weighted Round Robin

На кожному етапі, коли черга доходить до певного активного серверу, він обробляє розраховану кількість запитів.

4.2.3 Least Connection

Пул серверів:

```

servers: [
  { host: '127.0.0.1', port: 8003 },
  { host: '127.0.0.1', port: 8004 },
  { host: '127.0.0.1', port: 8005 },
  { host: '127.0.0.1', port: 8006 },
],

```

Перед кожним з'єднанням з віддаленою точкою в консоль виводиться список усіх кінцевих точок(для тестування):

```

this.queue.servers.forEach(server => {
  logger.info(JSON.stringify(server));
});

```

```

INFO | {"host":"127.0.0.1","port":8003,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:10:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8004,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:10:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8005,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:10:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8006,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:10:25 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8003 | timestamp: Sun Jun 05 2022 15:10:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"connections":2,"active":true} | timestamp: Sun Jun 05 2022 15:10:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8004,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:10:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8005,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:10:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8006,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:10:26 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8004 | timestamp: Sun Jun 05 2022 15:10:26 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 15:10:29 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 15:10:29 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 15:10:29 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8006 NOT FOUND | timestamp: Sun Jun 05 2022 15:10:31 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"connections":2,"active":true} | timestamp: Sun Jun 05 2022 15:10:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8004,"connections":2,"active":true} | timestamp: Sun Jun 05 2022 15:10:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8005,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:10:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8006,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:10:34 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8005 | timestamp: Sun Jun 05 2022 15:10:34 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 15:10:39 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 15:10:39 GMT+0300 (GMT+03:00)

```

Рис. 4.25 Результати роботи Least Connection

З відповідних логів бачимо, що кожне наступне з'єднання встановлюється з сервером, що має найменшу кількість підключень та є активним.

4.2.4 Weighted Least Connection

Пул серверів:

```

servers: [
  { host: '127.0.0.1', port: 8003, weight: 15 },
  { host: '127.0.0.1', port: 8004, weight: 10 },
  { host: '127.0.0.1', port: 8005, weight: 12 },
  { host: '127.0.0.1', port: 8006, weight: 20 },
],

```

					ІАЛЦ.467200.003 ПЗ	Арк.
						117
Зм.	Арк.	№ докум.	Підпис	Дата		

```

INFO | {"host":"127.0.0.1","port":8006,"weight":20,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:18:08 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8005 | timestamp: Sun Jun 05 2022 15:18:08 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 15:18:08 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 15:18:08 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8005 OK | timestamp: Sun Jun 05 2022 15:18:08 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":15,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:18:09 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8004,"weight":10,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:18:09 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8005,"weight":12,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:18:09 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8006,"weight":20,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:18:09 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8003 | timestamp: Sun Jun 05 2022 15:18:09 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":15,"connections":2,"active":true} | timestamp: Sun Jun 05 2022 15:18:10 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8004,"weight":10,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:18:10 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8005,"weight":12,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:18:10 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8006,"weight":20,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:18:10 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8004 | timestamp: Sun Jun 05 2022 15:18:10 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8006 NOT FOUND | timestamp: Sun Jun 05 2022 15:18:10 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":15,"connections":2,"active":true} | timestamp: Sun Jun 05 2022 15:18:11 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8004,"weight":10,"connections":2,"active":true} | timestamp: Sun Jun 05 2022 15:18:11 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8005,"weight":12,"connections":1,"active":true} | timestamp: Sun Jun 05 2022 15:18:11 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8006,"weight":20,"connections":0,"active":false} | timestamp: Sun Jun 05 2022 15:18:11 GMT+0300 (GMT+03:00)
INFO | Creating tunnel connection to 127.0.0.1:8005 | timestamp: Sun Jun 05 2022 15:18:11 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8003 OK | timestamp: Sun Jun 05 2022 15:18:18 GMT+0300 (GMT+03:00)
INFO | Health check: ws://127.0.0.1:8004 OK | timestamp: Sun Jun 05 2022 15:18:18 GMT+0300 (GMT+03:00)

```

Рис. 4.26 Результати роботи Weighted Least Connection

Принцип роботи схожий з Least Connection, але наступною точкою обробки запиту буде та точка, де значення кількості з'єднань поділене на вагу серверу буде найменшою.

4.2.5 Weighted Response Time

Пул серверів:

```

servers: [
  { host: '127.0.0.1', port: 8000 },
  { host: '127.0.0.1', port: 8001 },
  { host: '127.0.0.1', port: 8002 },
  { host: '127.0.0.1', port: 8003 },
]

```

```

INFO | {"host":"127.0.0.1","port":8000,"weight":15,"active":true,"responseTime":16} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"weight":10,"active":true,"responseTime":16} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"weight":12,"active":true,"responseTime":17} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":20,"active":false,"responseTime":2066} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8000,"weight":15,"active":true,"responseTime":16} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"weight":10,"active":true,"responseTime":16} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"weight":12,"active":true,"responseTime":17} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":20,"active":false,"responseTime":2066} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8000,"weight":15,"active":true,"responseTime":16} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"weight":10,"active":true,"responseTime":16} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"weight":12,"active":true,"responseTime":17} | timestamp: Sun Jun 05 2022 16:06:06 GMT+0300 (GMT+03:00)

```

Рис. 4.27 Результати роботи Weighted Response Time (перша ітерація)

Всі запити йдуть на сервер, де час відповіді(знаходиться при перевірці активності) є найменшою.

					Арк.
					118
Зм.	Арк.	№ докум.	Підпис	Дата	

ІАЛЦ.467200.003 ПЗ

```

INFO | {"host":"127.0.0.1","port":8000,"weight":15,"active":true,"responseTime":8} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"weight":10,"active":true,"responseTime":7} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"weight":12,"active":true,"responseTime":8} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":20,"active":false,"responseTime":2046} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8000,"weight":15,"active":true,"responseTime":8} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"weight":10,"active":true,"responseTime":7} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"weight":12,"active":true,"responseTime":8} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":20,"active":false,"responseTime":2046} | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:06:34 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8000,"weight":15,"active":true,"responseTime":8} | timestamp: Sun Jun 05 2022 16:06:41 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"weight":10,"active":true,"responseTime":7} | timestamp: Sun Jun 05 2022 16:06:41 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"weight":12,"active":true,"responseTime":8} | timestamp: Sun Jun 05 2022 16:06:41 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"weight":20,"active":false,"responseTime":2046} | timestamp: Sun Jun 05 2022 16:06:41 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:06:41 GMT+0300 (GMT+03:00)

```

Рис. 4.28 Результати роботи Weighted Response Time(друга ітерація)

Після наступної перевірки значення часу для кожної точки оновилося. З цього моменту інший сервер має найменший час відповіді. Тепер всі запити йдуть до нього.

4.2.6 Resource based(Adaptive)

Пул серверів:

```

servers: [
  { host: '127.0.0.1', port: 8000 },
  { host: '127.0.0.1', port: 8001 },
  { host: '127.0.0.1', port: 8002 },
  { host: '127.0.0.1', port: 8003 },
],

```

Для тесту додаємо логування часу відповіді для кожного запиту:

```
logger.info(`Response time: ${responseTime} ms`);
```

```

INFO | {"host":"127.0.0.1","port":8000,"ewma":0.4908779655216775,"active":true} | timestamp: Sun Jun 05 2022 16:21:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"ewma":0.2723938612842773,"active":true} | timestamp: Sun Jun 05 2022 16:21:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"ewma":0.5417936613932958,"active":true} | timestamp: Sun Jun 05 2022 16:21:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"ewma":0,"active":false} | timestamp: Sun Jun 05 2022 16:21:25 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:21:25 GMT+0300 (GMT+03:00)
INFO | Response time: 1 ms | timestamp: Sun Jun 05 2022 16:21:25 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8000,"ewma":0.4908779655216775,"active":true} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"ewma":0.3815347820916357,"active":true} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"ewma":0.5417936613932958,"active":true} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"ewma":0,"active":false} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | Response time: 0 ms | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8000,"ewma":0.4908779655216775,"active":true} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8001,"ewma":0.324304564778903,"active":true} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8002,"ewma":0.5417936613932958,"active":true} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | {"host":"127.0.0.1","port":8003,"ewma":0,"active":false} | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:21:26 GMT+0300 (GMT+03:00)

```

Рис. 4.29 Результати роботи Adaptive

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		119

Після кожного запиту йде оновлення рухомого середнього для кожної кінцевої точки на основі попереднього значення та часу відповіді. Як бачимо запити йдуть на сервер, де обчислене ewma є найменшим.

4.2.7 IP Hash та UrlHash

Пул серверів:

```
servers: [  
  { host: '127.0.0.1', port: 8000 },  
  { host: '127.0.0.1', port: 8001 },  
  { host: '127.0.0.1', port: 8002 },  
  { host: '127.0.0.1', port: 8003 },  
],
```

Під час надходження певного запиту відбувається прив'язка ip клієнта або url до певного серверу. Дані зберігаються в кеші(клас CacheWorker), а ключем є хеш адреси. Для тесту було додано логування хешу:

```
logger.info(`Hash: ${hash}, present in the cache: ${isHash}`);
```

Спочатку жоден сервер не прив'язаний до певного запиту. Після того, як за відповідною адресою дані потрапляють в кеш, всі задачі буде виконувати відповідна кінцева точка:

```
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: false | timestamp:  
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:05 GMT+0300 (GMT+03:00)  
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:  
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:06 GMT+0300 (GMT+03:00)  
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:  
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:07 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 16:42:13 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 16:42:13 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 16:42:13 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 16:42:15 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 16:42:23 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 16:42:23 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8000 NOT FOUND | timestamp: Sun Jun 05 2022 16:42:25 GMT+0300 (GMT+03:00)  
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 16:42:25 GMT+0300 (GMT+03:00)  
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:  
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:42:28 GMT+0300 (GMT+03:00)  
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:  
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:42:29 GMT+0300 (GMT+03:00)  
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:  
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:42:29 GMT+0300 (GMT+03:00)
```

Рис. 4.30 Результати роботи IpHash(перша ітерація)

					ІАЛЦ.467200.003 ПЗ	Арк.
						120
Зм.	Арк.	№ докум.	Підпис	Дата		

Як бачимо, в певний момент сервер, що прив'язаний до ip чи url стає неактивним. З цього моменту інший бекенд стає обробником задачі.

Далі неактивний сервер перезавантажили:

```
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:42:29 GMT+0300 (GMT+03:00)
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:42:29 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 16:42:33 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 16:42:33 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 NOT FOUND | timestamp: Sun Jun 05 2022 16:42:35 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 16:42:35 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 16:42:43 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 16:42:43 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 16:42:43 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 16:42:45 GMT+0300 (GMT+03:00)
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:48 GMT+0300 (GMT+03:00)
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:49 GMT+0300 (GMT+03:00)
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:49 GMT+0300 (GMT+03:00)
INFO | Hash: 69cebfd0e74f73858f2e5c00ab1c2d04f92376017501e016822ded53eda8e58f, present in the cache: true | timestamp:
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:42:49 GMT+0300 (GMT+03:00)
```

Рис. 4.31 Результати роботи IpHash(друга ітерація)

З цього моменту усі вхідні(прив'язані певним чином запити) знову надходять на даний сервер.

4.2.7 Random

Пул серверів:

```
servers: [
  { host: '127.0.0.1', port: 8000 },
  { host: '127.0.0.1', port: 8001 },
  { host: '127.0.0.1', port: 8002 },
  { host: '127.0.0.1', port: 8003 },
],
```

					ІАЛЦ.467200.003 ПЗ	Арк.
						121
Зм.	Арк.	№ докум.	Підпис	Дата		

```

INFO | Start load balancer listen on 3000 | timestamp: Sun Jun 05 2022 16:50:09 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8000 OK | timestamp: Sun Jun 05 2022 16:50:09 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8001 OK | timestamp: Sun Jun 05 2022 16:50:09 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8002 OK | timestamp: Sun Jun 05 2022 16:50:09 GMT+0300 (GMT+03:00)
INFO | Health check: http://127.0.0.1:8003 NOT FOUND | timestamp: Sun Jun 05 2022 16:50:11 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:50:13 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:50:13 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:50:14 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:50:14 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:50:15 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:50:15 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:50:15 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8000 | timestamp: Sun Jun 05 2022 16:50:16 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:50:16 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:50:17 GMT+0300 (GMT+03:00)
INFO | Send request GET /teams to 127.0.0.1:8001 | timestamp: Sun Jun 05 2022 16:50:17 GMT+0300 (GMT+03:00)

```

Рис. 4.32 Результати роботи Random(друга ітерація)

З логування можна побачити, що кожен наступний сервер обирається випадковим чином. Єдиною умовою залишається його працездатність.

ВИСНОВКИ ДО РОЗДІЛУ 4.

В даному розділі було проведено тестування компонентів програми, а також доступних алгоритмів балансування.

Серед функціоналу балансеру перш за все було перевірено його здатність обробляти вхідний http та ws трафік. Були відправлені різні запити на різні типи мережових з'єднань, які балансувальник успішно перенаправив до кінцевого пулу серверів.

Також було проведено тестування параметрів безпеки системи, які можливо налаштувати у файлі конфігурації. Спочатку було зімітовано хакерську DDoS атаку. Після певної кількості підозрілих запитів балансер почав розривати з'єднання з підозрілим клієнтом. Після цього пройшла перевірка балансеру на здатність роботи з SSL(зашифрованим трафіком). Продемонстровано відправку двох клієнських запитів, один з яких відправив до серверу відповідні сертифікати. При цьому балансер успішно обробив даний запит і відкинув той запит, що не зміг пройти етап автентифікації, адже не мав відповідних SSL параметрів.

За допомогою тестових скриптів було перевірено роботу вбудованих в систему кешування та логування. Також відбулося тестування зданості балансеру до перевірки активності кінцевих точок. Як активний, так і пасивний режими відпрацювали належним чином і щоразу, від залежності від їхньої логіки, оновлювали інформацію про стан пулу серверів.

На останок було протестовано роботу черги кінцевих точок. Тестування показали, що усі реалізовані алгоритми балансування працюють належним чином.

Загалом тестування показали, що реалізований балансер є досить гнучким до потреб користувача та здатним виконувати різні типи задач.

					ІАЛЦ.467200.003 ПЗ	Арк.
						123
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ

У ході даної роботи було реалізовано балансувальник навантаження, який здатний обробляти різні види мережевого трафіку(односпрямований та двоспрямований). При цьому програма включає в себе певний функціонал, який в залежності від потреб користувача може бути корисним, а саме безпека(SSL та DDoS), кешування, логування та перевірка стану кінцевих точок(включає активний та пасивний режим)

Описані вище параметри можна налаштовувати у відповідному файлі конфігурації.

У першому розділі роботи було описано предметну область розробленої системи, що включає в себе поняття балансування навантаження, його історію, переваги та недоліки, види, допоміжний функціонал, а також існуючі рішення.

Другий розділ включає в себе аналіз дев'ятьох алгоритмів балансування, які входять у функціонал реалізованого балансеру, а саме Round Robin, Weighted Round Robin, Least Connection, Weighted Least Connection, Weighted Response Time, Adaptive, Ip Hash, Url Hash та Random. Окрім цього в даному розділі було представлено список використаних технологій під час розробки системи.

Третій розділ був присвячений деталям розробки програми, де описано реалізації балансування трафіку, допоміжних компонентів, а також алгоритмів балансування.

В четвертому розділі було показано приклади використання балансеру при різних користувацьких налаштуваннях, а також роботу влаштованих в систему алгоритмів.

Слід також зазначити, що під час розробки застосунку були враховані всі зазначені на початку вимоги, яким він мав відповідати.

					ІАЛЦ.467200.003 ПЗ	Арк.
						124
Зм.	Арк.	№ докум.	Підпис	Дата		

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sandeep Sharma, Sarabjit Singh, and Meenakshi Sharma – Performance Analysis of Load Balancing Algorithms – World Academy of Science, Engineering and Technology 14 2008.
2. Tang Zhong – LOAD BALANCING IN PARALLEL ARCHITECTURE – The University of Tennessee, Knoxville, May 2005.
3. Hardware vs. software load balancer [Електронний ресурс] – Режим доступу до ресурсу: –
<https://www.techtarget.com/searchitoperations/answer/Hardware-vs-software-load-balancer-Which-is-better-for-an-enterprise> – Дата доступу: 03.03.2022.
4. R. Gandhi et al – Duet: Cloud scale load balancing with hardware and software, Proc. ACM Conf. SIGCOMM (SIGCOMM) – pp. 27-38, 2014.
5. Ken Salchow – Load Balancing 101: The Evolution to Application Delivery Controllers, F5 Networks Inc. – 401 Elliott Avenue West, Seattle, WA 98119, 2012.
6. David Wagner, University of California, Berkeley Bruce Schneier, Counterpane Systems – Analysis of the SSL 3.0 Protocol – Oakland, California, November 1996.
7. S. Goldwasser and S. Micali. – Probabilistic Encryption, Journal of Computer and System Sciences – Vol. 28, 1984, pp. 270–299.
8. DNS, LOAD BALANCING AND DDOS ATTACKS [Електронний ресурс] – Режим доступу до ресурсу: –
<https://kemptechnologies.com/blog/load-balancing-and-ddos-attacks/#:~:text=Load%20balancers%20also%20add%20resiliency,exhaust%20resources%20and%20saturate%20links.> – Дата доступу: 18.03.2022.
9. Pasumpon Pandian, Xavier Fernando, Syed Mohammed Shamsul Islam – Computer Networks, Big Data and IoT – Madurai, India, December 2020.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		125

10. A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar et al. – The quic transport protocol: Design and internet-scale deployment, Proceedings of the Conference of the ACM Special Interest Group on Data Communication – pp. 183-196, 2017.
11. José Emmanuel Cruz de la Cruz, Ing. Christian Augusto Romero Goyzueta – Design of a high availability system with HAProxy and domain name service for web services – [2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing](#), Peru.
12. HAProxy [Електронний ресурс] – Режим доступу до ресурсу: – <https://www.ukhost4u.com/what-is-haproxy-load-balancer-and-how-does-it-work/> – Дата доступу: 22.03.2022.
13. Elastic Load Balancing [Електронний ресурс] – Режим доступу до ресурсу: – <https://aws.amazon.com/ru/elasticloadbalancing/> – Дата доступу: 22.03.2022.
14. F5 NGINX Ingress Controller and App Protect in a cloud-native, container world [Електронний ресурс] – Режим доступу до ресурсу: – <https://amazic.com/nginx-ingress-controller/> – Дата доступу: 23.03.2022.
15. NGINX [Електронний ресурс] – Режим доступу до ресурсу: – <https://www.hostinger.com/tutorials/what-is-nginx> – Дата доступу: 23.03.2022.
16. Load Balancer Enterprise ADC [Електронний ресурс] – Режим доступу до ресурсу: – <https://www.getapp.com/security-software/a/loadbalancer-org/> – Дата доступу: 23.03.2022.
17. Citrix [Електронний ресурс] – Режим доступу до ресурсу: – <https://www.citrix.com/solutions/digital-workspace/> – Дата доступу: 25.03.2022.
18. Load balancing with Citrix ADC appliance [Електронний ресурс] – Режим доступу до ресурсу: – <https://docs.citrix.com/en->

[us/storefront/1912-ltsr/integrate-with-citrix-gateway-and-citrix-adc/load-balancing-with-citrix-gateway.html](https://storefront/1912-ltsr/integrate-with-citrix-gateway-and-citrix-adc/load-balancing-with-citrix-gateway.html) – Дата доступу: 25.03.2022.

19. Payal Beniwal, Atul Garg – A comparative study of static and dynamic Load Balancing Algorithms – Maharishi Markandeshwar University, Mullana Ambala, India / Haryana, December
20. Derek L. Eager, Edward D. Lazowska, John Zahorjan – Adaptive load sharing in homogeneous distributed systems – IEEE Transactions on Software Engineering, v.12 n.5 – May 1986.
21. Mondal, Brototi, Dasgupta, Kousik, Dutta, Paramartha – Load balancing in cloud computing using stochastic hill climbing: A soft computing approach. Procedia Technol. – pp. 783–789, 2012.
22. R. Jain – A Comparison of Hashing Schemes for Address Lookup in Computer Networks", IEEE Transactions on Communications – Oct. 1992.
23. Roberto Beraldi, Claudia Canali, Riccardo Lancellotti, Gabriele Proietti Mattia – A Random Walk based Load Balancing Algorithm for Fog Computing – Paris, France, 20-23 April 2020.
24. Douglas Crockford – JavaScript: The Good Parts – O'Reilly Media, Inc., USA – May 2008.
25. M. Cantelon, M. Harter, TJ Holowaychuk, N. Rajlich – Node.js in Action – 2014
26. R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, August – Hypertext Transfer Protocol - HTTP – 2009.
27. R. T. Fielding – REST API must be hypertext driven – 28 October, 2008,
28. Andrew Lombardi – WebSocket: Lightweight Client-Server Communications – USA, 2015.
29. Fette, I. and Melnikov – A. The websocket protocol – 2011

ДОДАТОК 1

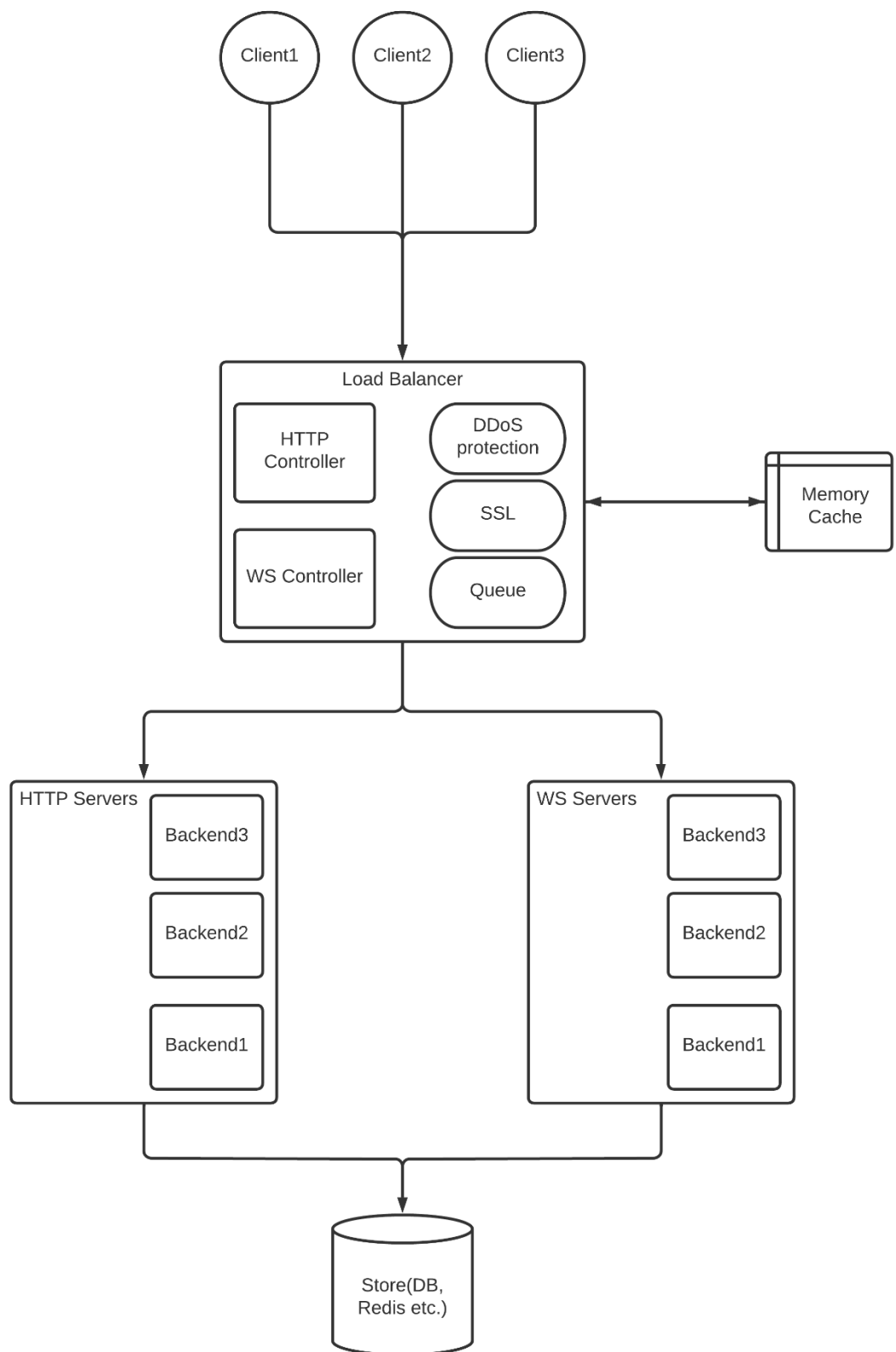
Спосіб балансування даних

Архітектура системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив	Іванченко В. В.				Спосіб балансування даних Архітектура системи		
Перевірив	Волокита І. М.						
Н. Контр.	Сімоненко В. П.						
Затвердив							
					Літ.	Аркуш	Аркушів
						1	1
					НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІІ-83		

ДОДАТОК 2

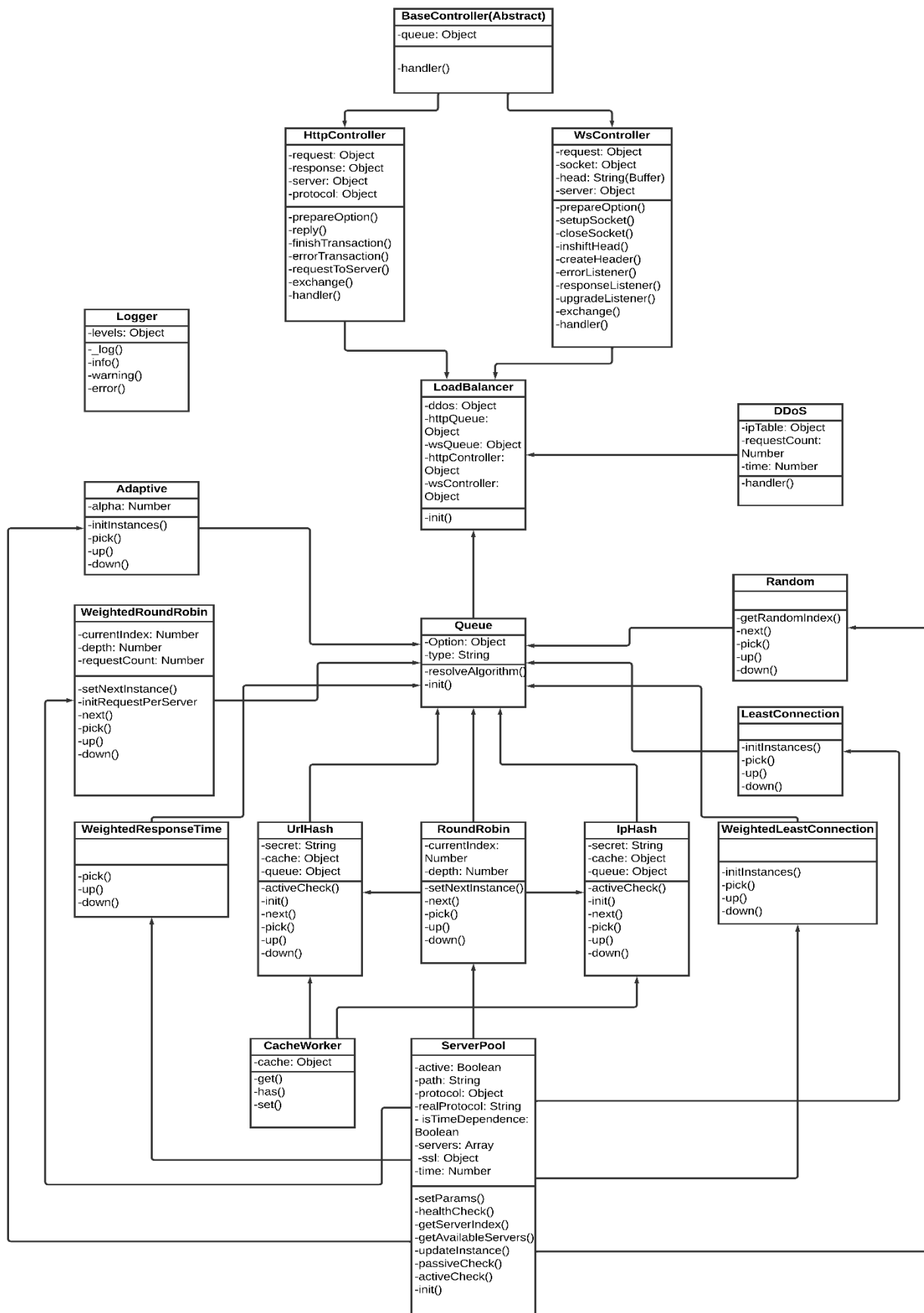
Спосіб балансування даних

Діаграма класів

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



ІАЛЦ.467200.005 Д2					Спосіб балансування даних Діаграма класів			Літ.	Аркуш	Аркушів
		№ докум.	Підпис	Дата					1	1
Розробив	Іванченко В. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІІ-83					
Перевірив	Волокита І. М.									
Н. Контр.	Сімоненко В. П.									
Затвердив										

ДОДАТОК 3

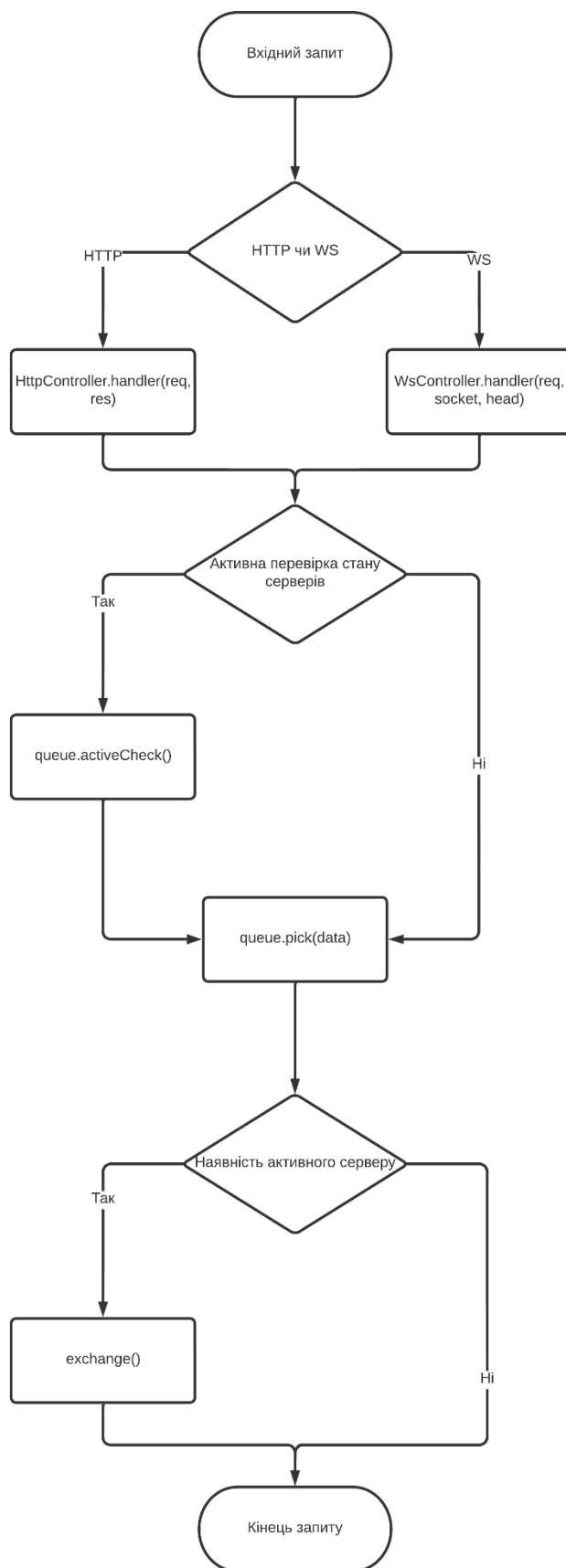
Спосіб балансування даних

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 ДЗ								
		№ докум.	Підпис	Дата									
Розробив		Іванченко В. В.			Спосіб балансування даних Алгоритм дій програм- ного забезпечення				Літ.	Аркуш	Аркушів		
Перевірив		Волокита І. М.									1	1	
									НТУУ КПІ ім. Ігоря				
Н. Контр.		Сімоненко В. П.							Сікорського, ФІОТ, ПІ-83				
Затвердив													

ДОДАТОК 4

Спосіб балансування даних

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 30

Київ 2022 р

```

'use strict';

const http = require('http');
const https = require('https');
const { secure, port, ssl, doss } = require('./config/balancer');
const { HttpController, WsController } = require('./controllers');
const Queue = require('./queue');
const DDoS = require('./infrastructure/DDoS');
const logger = require('./infrastructure/Logger');

class LoadBalancer {
  constructor() {
    this.ddos = null;
    this.wsQueue = new Queue('ws').init();
    this.httpQueue = new Queue('http').init();
    this.ws = new WsController(this.wsQueue);
    this.http = new HttpController(this.httpQueue);
  }

  init = () => {
    const protocol = secure ? https : http;
    const options = secure ? { ...ssl } : {};
    const server = protocol.createServer(options, this.http.handler);
    server.on('upgrade', this.ws.handler);

    if (doss) {
      this.ddos = new DDoS(3000, 100);
      server.on('connection', this.ddos.handler);
    }

    server.listen(port, () => {
      logger.info(`Start load balancer listen on ${port}`);
    });
  }

  const balancer = new LoadBalancer();
  balancer.init(

```

```

'use strict';

const http = require('http');
const https = require('https');
const config = require('../../config');
const logger = require('../../infrastructure/Logger');
const BaseController = require('../BaseController');

class HttpWorker extends BaseController {
  constructor(queue) {
    super(queue);
    this.request = null;
    this.response = null;
    this.server = null;
    this.protocol = null;
  }

  prepareOption = async () => {
    try {
      let otherParams = {};
      const { ssl, up } = config.http.secure;
      if (up) otherParams = ssl;

      return {
        hostname: this.server.host,
        port: this.server.port,
        path: this.request.url,
        method: this.request.method,
        headers: this.request.headers,
        ...otherParams
      };
    } catch (e) {
      return Promise.reject(e);
    }
  }

  reply = async (body, statusCode, headers = {}) => {
    try {
      this.response.writeHead(statusCode, headers);
      this.response.end(body);
    } catch (e) {

```

```

        return Promise.reject(e);
    }
}

finishTransaction = async (transaction) => {
    return new Promise((resolve, reject) => {
        const chunks = [];
        transaction.on('data', (chunk) => {
            try {
                chunks.push(chunk);
            } catch (e) {
                reject(e);
            }
        });

        transaction.on('end', () => {
            try {
                const body = Buffer.concat(chunks);
                resolve(body);
            } catch (e) {
                reject(e);
            }
        });
    });
}

errorTransaction = (transaction) => {
    transaction.on('error', async (err) => {
        await this.reply(err, 500);
    });
}

requestToServer = async (options, body) => {
    return new Promise((resolve, reject) => {
        try {
            const request = this.protocol.request(options, async (response) => {
                try {
                    const data = await this.finishTransaction(response);
                    await this.reply(data, response.statusCode, response.headers);
                } catch (e) {
                    reject(e);
                }
            });
        } catch (e) {
            reject(e);
        }
    });
}

```

```

        logger.info(`Send request ${options.method} ${options.path} to
${options.hostname}:${options.port}`);
        this.errorTransaction(request);
        if (body) request.write(body);
        request.end();
    } catch (e) {
        reject(e);
    }
});
}

exchange = async () => {
    try {
        if (!this.server) {
            await this.reply('Error: No available servers', 404);
            return;
        }

        const options = await this.prepareOption();
        const body = await this.finishTransaction(this.request);

        if (config.http.algorithm && config.http.algorithm === 'Adaptive') {
            const startTime = new Date().getTime();
            this.requestToServer(options, body);
            const endTime = new Date().getTime();

            this.queue.up(this.server, {responseTime: endTime - startTime});
        } else {
            this.requestToServer(options, body);
        }
    } catch (e) {
        return Promise.reject(e);
    }
}

handler = async (request, response) => {
    try {
        await this.queue.activeCheck();
        this.request = request;
        this.response = response;

        const url = request.url;
        const ip = request.socket.localAddress;
        const { up } = config.http.secure;

        this.server = await this.queue.pick({ url, ip });
        this.protocol = up ? https : http;

        await this.exchange();
    }
}

```

```

    } catch (e) {
      await this.reply('Load balancing error', 500);
      logger.error(e);
    }
  };
}

```

```

module.exports = HttpWorker;

```

```

'use strict';

```

```

const http = require('http');
const https = require('https');
const BaseController = require('../BaseController');
const logger = require('../../infrastructure/Logger');
const config = require('../../config');

```

```

class WsController extends BaseController {
  constructor(queue) {
    super(queue);
    this.request = null;
    this.socket = null;
    this.head = null;
    this.server = null;
  }

```

```

  setupSocket = async () => {
    try {
      this.socket.setTimeout(0);
      this.socket.setNoDelay(true);
      this.socket.setKeepAlive(true, 0);
    } catch (e) {
      return Promise.reject(e);
    }
  };

```

```

  closeSocket = () => {
    if (this.socket) this.socket.end();
  }

```

```

  prepareOption = async () => {
    try {
      let otherParams = {};
      const { ssl, up } = config.ws.secure;

```

```

    if (up && ssl) otherParams = ssl;

    return {
      port: this.server.port,
      method: this.request.method,
      headers: this.request.headers,
      path: this.request.url,
      ...otherParams
    };
  } catch (e) {
    return Promise.reject(e);
  }
}

unshiftHead = async (socket, head) => {
  try {
    if (head && head.length) socket.unshift(head);
  } catch (e) {
    return Promise.reject(e);
  }
}

createHeader = async (headLine, headers) => {
  try {
    const values = Object.keys(headers).map(key => (`${key}:
    ${headers[key]}`));
    values.unshift(headLine);
    return `${values.join('\r\n')}\r\n\r\n`;
  } catch (e) {
    return Promise.reject(e);
  }
};

socketListener = (connection) => {
  return new Promise((resolve, reject) => {
    connection.on('socket', () => {
      try {
        this.queue.up(this.server);
      } catch (e) {
        reject(e);
      }
    });
  });
}

```

```
}
```

```
errorListener = (connection, socket, callback = () => {}) => {  
  return new Promise((resolve, reject) => {  
    connection.on('error', () => {  
      try {  
        socket.end();  
        callback();  
      } catch (e) {  
        reject(e);  
      }  
    });  
  })  
}
```

```
responseListener = (connection) => {  
  return new Promise((resolve, reject) => {  
    connection.on('response', async (response) => {  
      try {  
        if (!response.upgrade) {  
          const headLine = `HTTP/${response.httpVersion}  
${response.statusCode} ${response.statusMessage}`;  
          const headers = response.headers;  
          const header = await this.createHeader(headLine,  
headers);  
  
          this.socket.write(header);  
          response.pipe(this.socket);  
        }  
      } catch (e) {  
        reject(e);  
      }  
    });  
  });  
}
```

```
upgradeListener = (connection) => {  
  return new Promise((resolve, reject) => {  
    connection.on('upgrade', async (connectionResponse,  
connectionSocket, connectionHead) => {  
      try {  
        const serverDown = () => this.queue.down(this.server);  
        this.errorListener(connectionSocket, this.socket,  
serverDown);  
      }  
    });  
  });  
}
```

```

        this.errorListener(this.socket, connectionSocket,
serverDown);

        this.setupSocket(connectionSocket);
        await this.unshiftHead(connectionSocket, connectionHead);

        const headLine = 'HTTP/1.1 101 Switching Protocols';
        const headers = connectionResponse.headers;
        const header = await this.createHeader(headLine, headers);
        this.socket.write(header);
        connectionSocket.pipe(this.socket).pipe(connectionSocket);
        logger.info(`Creating tunnel connection to
${this.server.host}:${this.server.port}`);
    } catch (e) {
        reject(e);
    }
  });
});
}

```

```

exchange = async () => {
  try {
    if (!this.server) {
      this.closeSocket();
      return;
    }

    await this.setupSocket();
    await this.unshiftHead(this.socket, this.head);
    const options = await this.prepareOption();
    const connection = this.protocol.request(options);

    this.socketListener(connection);
    this.responseListener(connection);
    this.errorListener(connection, this.socket);
    this.upgradeListener(connection);

    return connection.end();
  } catch (e) {
    return Promise.reject(e);
  }
}

```

```

handler = async (request, socket, head) => {

```

```

    try {
        await this.queue.activeCheck();
        this.request = request;
        this.socket = socket;
        this.head = head;

        const url = request.url;
        const ip = request.socket.localAddress;
        const { up } = config.ws.secure;

        this.server = this.queue.pick({ url, ip });
        this.protocol = up ? https : http;

        await this.exchange();
    } catch (e) {
        this.closeSocket();
        logger.error(e);
    }
}

```

```

module.exports = WsController;

```

```

'use strict';

```

```

class BaseController {
    constructor(queue) {
        const proto = Object.getPrototypeOf(this);
        if (proto.constructor === BaseController) {
            throw new Error('Abstract class');
        }

        this.queue = queue;
    }

    handler() {
        throw new Error('Not implement');
    }
}

```

```

module.exports = BaseController;

```

```

'use strict';

const HttpController = require('./http');
const WsController = require('./ws');

module.exports = {
  HttpController,
  WsController,
};

'use strict';

const http = require('http');
const https = require('https');
const config = require('../../config');
const logger = require('../Logger');

class ServerPool {
  constructor(type) { //HTTP чи WS
    this.active = false; //тип перевірка стану
    this.path = '/'; //маршрут для перевірки стану
    this.protocol = null; //протокол для запиту(в залежності від налаштувань
    безпеки)
    this.realProtocol = null; //реальний протокол(HTTP, WS ...) - лише для
    логування
    this.isTimeDependence = false; //показує чи потрібно рахувати час відповіді
    this.servers = []; //масив серверів
    this.ssl = {}; //параметри безпеки
    this.time = 5 * 60; //інтервал перевірки стану
    this.setParams(type); //виклик функції для встановлення параметрів на основі
    користувацьких налаштувань
  }

  setParams = (type) => {
    const params = config[type];
    if (params) {
      const {healthCheck, secure} = params;
      if (healthCheck) {
        this.active = healthCheck.activeCheck;
        this.path = healthCheck.path;
        this.time = healthCheck.time || this.time;
      }
    }
  }
}

```

```

    if (secure) {
      this.protocol = secure.up ? https : http;
      this.realProtocol = secure.up ? `${type}s` : type;
      this.ssl = secure.up ? secure.ssl : {};
    }

    this.isTimeDependence = params.algorithm === 'WeightedResponseTime';
    this.servers = params.servers;
  }
}

healthCheck = (server) => {
  return new Promise((resolve, reject) => {
    const options = {
      method: 'GET',
      hostname: server.host,
      port: server.port,
      path: this.path,
      ...this.ssl
    };

    const request = this.protocol.request(options, (res) => {
      if (res.statusCode === 200 || res.statusCode === 426) {
        resolve(true);
      }

      resolve(false);
    },
  );

  request.on('error', (err) => {
    resolve(false);
  });
  request.end();
});
}

getServerIndex = (instance) => {
  return this.servers.findIndex(
    (server) => server.host === instance.host && server.port ===
instance.port,
  );
};

```

```

}

getAvailableServers = () => {
  return this.servers.filter((server) => server.active);
}

updateInstance = async (server) => {
  try {
    if (this.isTimeDependence) {
      const startTime = new Date().getTime();
      server.active = await this.healthCheck(server);
      const endTime = new Date().getTime();
      server.responseTime = endTime - startTime;
    } else {
      server.active = await this.healthCheck(server);
    }

    logger.info(
      `Health check: ${this.realProtocol}://${server.host}:${server.port} ${
        server.active ? 'OK' : 'NOT FOUND'
      }`,
    );

    return server;
  } catch (e) {
    logger.error(e);
  }
}

passiveCheck = async () => {
  try {
    this.servers = await Promise.all(this.servers.map(this.updateInstance));
  } catch (e) {
    logger.error(e);
  }
}

activeCheck = async () => {
  try {
    if (this.active) await this.passiveCheck();
  } catch (e) {
    logger.error(e);
  }
}

```

```

}

init = async () => {
  try {
    if (!this.active) {
      await this.passiveCheck();
      setInterval(this.passiveCheck, this.time * 1000);
    }
  } catch (e) {
    logger.error(e);
  }
}

module.exports = ServerPool;

'use strict';

const logger = require('../../infrastructure/Logger');
const ServerPool = require('../../infrastructure/ServerPool');

class Adaptive extends ServerPool {
  constructor(type) {
    super(type);

    this.alpha = 0.15;
    this.initInstances();
  }

  initInstances() {
    this.servers = this.servers.map((server) => ({ ...server, ewma: 0 }));
  }

  pick() {
    try {
      const activeServers = this.getAvailableServers();
      if (activeServers.length === 0) return null;

      return activeServers.reduce((acc, val) => (acc.ewma > val.ewma ? val :
acc));
    } catch (e) {
      logger.error(e);
    }
  }
}

```

```

    }
}

up(instance, data = {}) {
    const { responseTime } = data;
    const index = this.getServerIndex(instance);
    this.servers[index].ewma =
        this.alpha * responseTime + (1 - this.alpha) * instance.ewma;
}

down(instance) {}
}

module.exports = Adaptive;

'use strict';

const utils = require('../..//infrastructure/Utils');
const CacheWorker = require('../..//infrastructure/Cache');
const logger = require('../..//infrastructure/Logger');
const RoundRobin = require('../RoundRobin');

class IpHash {
    constructor(type) {
        this.secret = 'ipHash';
        this.queue = new RoundRobin(type);
        this.cache = new CacheWorker();
        this.init = this.queue.init.bind(this.queue);
        this.activeCheck = this.queue.activeCheck.bind(this.queue);
    }

    next(servers, hash) {
        for (let i = 0; i < servers.length; i++) {
            const index = this.queue.getServerIndex(servers[i]);
            const instance = this.queue.servers[index];
            if (instance.active) return instance;
        }

        const nextInstance = this.queue.pick();
        if (nextInstance) {
            servers.push(nextInstance);
            this.cache.set(hash, servers);
        }
    }
}

```

```

    }

    return nextInstance;
}

pick({ ip }) {
  try {
    const hash = utils.hash(ip, this.secret);
    const isHash = this.cache.has(hash);

    if (isHash) {
      const servers = this.cache.get(hash);
      return this.next(servers, hash);
    }

    const instance = this.queue.pick();
    if (instance) this.cache.set(hash, [instance]);

    return instance;
  } catch (e) {
    logger.error(e);
  }
}

'use strict';

const ServerPool = require('../../infrastructure/ServerPool');
const logger = require('../../infrastructure/Logger');
const utils = require('../../infrastructure/Utils');

class LeastConnection extends ServerPool {
  constructor(type) {
    super(type);
    this.initInstances();
  }

  initInstances() {
    this.servers = this.servers.map((server) => ({ ...server, connections: 0
})));
  }

```

```

pick() {
  try {
    const activeServers = this.getAvailableServers();

    if (activeServers.length === 0) {
      return null;
    }

    return activeServers.reduce((acc, val) =>
      acc.connections > val.connections ? val : acc,
    );
  } catch (e) {
    logger.error(e);
  }
}

up(instance) {
  const index = this.getServerIndex(instance);
  instance.connections += 1;
  this.servers[index] = instance;
}

down(instance) {
  const index = this.getServerIndex(instance);

  instance.connections -= 1;
  this.servers[index] = instance;
}
}

module.exports = LeastConnection;

'use strict';

const ServerPool = require('../../infrastructure/ServerPool');
const logger = require('../../infrastructure/Logger');

class Random extends ServerPool {
  constructor(type) {
    super(type);
  }
}

```

```

getRandomIndex() {
  const availableServers = this.getAvailableServers();
  const limit = availableServers.length - 1;

  return Math.floor(Math.random() * limit);
}

next() {
  const randomIndex = this.getRandomIndex();

  return this.servers[randomIndex];
}

pick() {
  try {
    const availableServers = this.getAvailableServers();
    if (availableServers === 0) {
      return null;
    }

    return this.next();
  } catch (e) {
    logger.error(e);
  }
}

up() {}

down() {}
}

module.exports = Random;

'use strict';

const ServerPool = require('../../infrastructure/ServerPool');
const logger = require('../../infrastructure/Logger');

class RoundRobin extends ServerPool {
  constructor(type) {
    super(type);
  }
}

```

```

    this.currentIndex = 0;
    this.depth = 0;
}

setNextInstance() {
    this.currentIndex =
        this.currentIndex === this.servers.length - 1 ? 0 : this.currentIndex + 1;
}

next() {
    const currentInstance = this.servers[this.currentIndex];
    this.setNextInstance();

    return currentInstance;
}

pick() {
    try {
        if (this.depth === this.servers.length) {
            return null;
        }

        const instance = this.next();
        if (instance.active) {
            this.depth = 0;
            return instance;
        }

        this.depth += 1;
        return this.pick();
    } catch (e) {
        logger.error(e);
    }
}

up() {}

down() {}
}

module.exports = RoundRobin;

```

```

'use strict';

const Url = require('url');
const utils = require('../../infrastructure/Utils');
const CacheWorker = require('../../infrastructure/Cache');
const RoundRobin = require('../RoundRobin');
const logger = require('../../infrastructure/Logger');

class UrlHash {
  constructor(type) {
    this.secret = 'urlHash';
    this.queue = new RoundRobin(type);
    this.cache = new CacheWorker();
    this.init = this.queue.init.bind(this.queue);
    this.activeCheck = this.queue.activeCheck.bind(this);
  }

  prepareUrl(url) {
    const parsedUrl = Url.parse(url).pathname;
    return parsedUrl.substr(0, 80);
  }

  next(servers, hash) {
    for (let i = 0; i < servers.length; i++) {
      const index = this.queue.getServerIndex(servers[i]);
      const instance = this.queue.servers[index];
      if (instance.active) return instance;
    }

    const nextInstance = this.queue.pick();
    if (nextInstance) {
      servers.push(nextInstance);
      this.cache.set(hash, servers);
    }

    return nextInstance;
  }

  pick({ url }) {
    try {
      const preparedUrl = this.prepareUrl(url);
      const hash = utils.hash(preparedUrl, this.secret);
      const isHash = this.cache.has(hash);
    }
  }
}

```

```

    if (isHash) {
        const servers = this.cache.get(hash);
        return this.next(servers, hash);
    }

    const instance = this.queue.pick();
    if (instance) this.cache.set(hash, [instance]);

    return instance;
} catch (e) {
    logger.error(e);
}
}

up() {}

down() {}
}

module.exports = UrlHash;

'use strict';

const ServerPool = require('../../infrastructure/ServerPool');
const logger = require('../../infrastructure/Logger');

class WeightedLeastConnection extends ServerPool {
    constructor(type) {
        super(type);

        this.currentIndex = 0;
        this.initInstances();
    }

    initInstances() {
        this.servers = this.servers.map((server) => ({ ...server, connections: 0
    }));
    }

    pick() {
        try {
            const activeServers = this.getAvailableServers();

```

```

    if (activeServers.length === 0) {
        return null;
    }
    return activeServers.reduce((acc, val) =>
        acc.connections / acc.weight > val.connections / val.weight ? val : acc,
    );
} catch (e) {
    logger.error(e);
}
}

up(instance) {
    const index = this.getServerIndex(instance);

    instance.connections += 1;
    this.servers[index] = instance;
}

down(instance) {
    const index = this.getServerIndex(instance);

    instance.connections -= 1;
    this.servers[index] = instance;
}
}

module.exports = WeightedLeastConnection;

'use strict';

const ServerPool = require('../..//infrastructure/ServerPool');
const logger = require('../..//infrastructure/Logger');

class WeightedResponseTime extends ServerPool {
    constructor(type) {
        super(type);
    }

    pick() {
        try {
            const activeServers = this.getAvailableServers();

```

```

    if (activeServers.length === 0) {
        return null;
    }

    return activeServers.reduce((acc, val) =>
        acc.responseTime > val.responseTime ? val : acc,
    );
} catch (e) {
    logger.error(e);
}
}

up() {}

down() {}
}

module.exports = WeightedResponseTime;

'use strict';

const ServerPool = require('../../infrastructure/ServerPool');
const logger = require('../../infrastructure/Logger');

class WeightedRoundRobin extends ServerPool {
    constructor(type) {
        super(type);
        this.currentIndex = 0;
        this.requestCount = 0;
        this.depth = 0;
        this.initRequestPerServer();
    }

    initRequestPerServer() {
        this.servers = this.servers.map((server) => {
            server.requestWeight = Math.ceil(server.weight / this.servers.length);
            return server;
        });

        this.servers.forEach((server) => {
            logger.info(

```

```

        `{ host: ${server.host}, port: ${server.port}, requestCount:
${server.requestWeight} }`,
        );
    });
}

setNextInstance() {
    this.currentIndex =
        this.currentIndex === this.servers.length - 1 ? 0 : this.currentIndex + 1;
}

next() {
    const currentInstance = this.servers[this.currentIndex];

    if (currentInstance.requestWeight === this.requestCount + 1) {
        this.setNextInstance();
        this.requestCount = 0;
    } else {
        this.requestCount += 1;
    }

    return currentInstance;
}

pick() {
    try {
        if (this.depth === this.servers.length) {
            return null;
        }

        const instance = this.next();

        if (instance.active) {
            this.depth = 0;
            return instance;
        }

        if (this.requestCount !== 0) {
            this.setNextInstance();
        }

        this.depth += 1;
        return this.pick();
    }

```

```

    } catch (e) {
        logger.error(e);
    }
}

up() {}

down() {}
}

module.exports = WeightedRoundRobin;

'use strict';

const config = require('../config');
const { typeOptionMapping } = require('../infrastructure/Utils');

class Queue {
    constructor(type) {
        this.Option = require(`../${this.resolveAlgorithm(type)}`);
        this.type = type;
    }

    resolveAlgorithm = (type) => {
        const { algorithm } = config[type];

        if (algorithm && typeOptionMapping[type].includes(algorithm)) return
algorithm;
        return 'RoundRobin';
    }

    init() {
        const { Option, type } = this;
        const queue = new Option(type);
        queue.init();

        return queue;
    }
}

module.exports = Queue;

```

```

'use strict';

const logger = require('../Logger');

class CacheWorker {
  constructor() {
    this.cache = new Map();
  }

  get(key) {
    try {
      return this.cache.get(key);
    } catch (e) {
      logger.error(e);
    }
  }

  has(key) {
    try {
      return this.cache.has(key);
    } catch (e) {
      logger.error(e);
    }
  }

  set(key, value, ttl = null) {
    try {
      this.cache.set(key, value);

      if (ttl) {
        setTimeout(() => {
          this.cache.delete(key);
        }, ttl * 10);
      }
    } catch (e) {
      logger.error(e);
    }
  }
}

module.exports = CacheWorker;

```

```

'use strict';

const logger = require('../Logger');

class DDoS {
  constructor(time, requestCount) {
    this.ipTable = {};
    this.time = time;
    this.requestCount = requestCount;
  }

  handler = (socket) => {
    try {
      const ip = socket.address().address;
      const time = Date.now();
      if(ip in this.ipTable) {
        if(time - this.ipTable[ip].time > this.time) {
          this.ipTable[ip] = {
            count : 1,
            time
          };
          return;
        }
        this.ipTable[ip].count++;
        this.ipTable[ip].time = time;
        if(this.ipTable[ip].count > this.requestCount) {
          socket.end('HTTP/1.1 429 Too Many Requests');
          socket.destroy();
          logger.warning(`Destroy connection ip: ${ip}`);
        }
        return;
      }
      this.ipTable[ip] = {
        count : 1,
        time
      };
    } catch (e) {
      logger.error(e);
    }
  }
}

module.exports = DDoS;

```

```

    'use strict';

class Logger {
  constructor() {
    this.levels = {
      error: '\x1b[31m',
      info: '\x1b[32m',
      warning: '\x1b[33m',
    };
  }

  _log(message, level) {
    const time = new Date();

    const formatMessage = `${
      this.levels[level]
    }${level.toUpperCase()} | \x1b[0m ${message} | timestamp: ${time}`;
    console.log(formatMessage);
  }

  info(message) {
    this._log(message, 'info');
  }

  warning(message) {
    this._log(message, 'warning');
  }

  error(message) {
    this._log(message, 'error');
  }
}

module.exports = new Logger();

const typeOptionMapping = {
http: [
  'RoundRobin',
  'WeightedRoundRobin',
  'IpHash',
  'UrlHash',
  'Random',

```

```

        'WeightedResponseTime',
        'Adaptive',
    ],
    ws: [
        'RoundRobin',
        'WeightedRoundRobin',
        'IpHash',
        'UrlHash',
        'Random',
        'LeastConnection',
        'WeightedLeastConnection',
        'WeightedResponseTime',
    ],
};

```

```

'use strict';

```

```

const fs = require('fs');

```

```

module.exports = {
  port: 3000,
  doss: true,
  secure: false,
  ssl: {
    key: fs.readFileSync(`${__dirname}/certs/server-key.pem`),
    cert: fs.readFileSync(`${__dirname}/certs/server-crt.pem`),
    ca: [fs.readFileSync(`${__dirname}/certs/client-crt.pem`)],
    requestCert: true,
    rejectUnauthorized: true,
  },
};

```

```

'use strict';

```

```

const fs = require('fs');

```

```

module.exports = {
  servers: [
    { host: '127.0.0.1', port: 8000 },
    { host: '127.0.0.1', port: 8001 },
    { host: '127.0.0.1', port: 8002 },
    { host: '127.0.0.1', port: 8003 },
  ],
};

```

```

],
algorithm: 'Random',
healthCheck: {
  activeCheck: false,
  path: '/teams',
  time: 10, // seconds
},
secure: {
  up: false,
  ssl: {
    rejectUnauthorized: false,
    key: fs.readFileSync(`${__dirname}/certs/client-key.pem`),
    cert: fs.readFileSync(`${__dirname}/certs/client-crt.pem`),
    ca: fs.readFileSync(`${__dirname}/certs/server-crt.pem`),
  },
},
};

```

```

'use strict';

```

```

module.exports = {
  servers: [
    { host: '127.0.0.1', port: 8003, weight: 15 },
    { host: '127.0.0.1', port: 8004, weight: 10 },
    { host: '127.0.0.1', port: 8005, weight: 12 },
    { host: '127.0.0.1', port: 8006, weight: 20 },
  ],
  algorithm: 'LeastConnection',
  healthCheck: {
    activeCheck: false,
    path: '/',
    time: 10,
  },
  secure: {
    up: false,
  },
};

```

```

module.exports = {
  balancer: require('./balancer'),
  http: require('./http'),

```

```
ws: require('./ws'),  
};
```