НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____20__ р.

Дипломний проект

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Система виявлення сервісів з розумним контролем навантаження»

Виконав:
студентка IV курсу, групи ІО-62
Кузьменко Олександра Вікторівна _____

Керівник:
Професор, доктор фізико-математичних наук
Гордієнко Юрій Григорович _____

Консультант з нормоконтролю:
Професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Рецензент:
Доцент, кандидат технічних наук
Писаренко Андрій Володимирович _____

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

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____20__ р.

ЗАВДАННЯ

на дипломний проект студентці

Кузьменко Олександрі Вікторівні

1. Тема проекту «Система виявлення сервісів з розумним контролем навантаження», керівник проекту Гордієнко Юрій Григорович, професор, доктор фізико-математичних наук, затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проекту 06 червня 2020 р.

3. Вихідні дані до проекту: технічне завдання, науково-технічна література

4. Зміст пояснювальної записки: порівняльний аналіз наявних програмних рішень, вибір засобів реалізації та опис отриманої системи

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) :

1. Функціональна схема – плакат
2. Принципова схема – плакат
3. Структурна схема – плакат

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Сімоненко В.П., професор		

7. Дата видачі завдання 01 вересня 2019 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Затвердження теми роботи	01.09.2019-22.12.2019	
2	Вивчення та аналіз завдання	23.12.2019-20.03.2020	
3	Розробка архітектури та загальної структури системи	20.03.2020-01.04.2020	
4	Розробка структур окремих підсистем	01.04.2020-10.04.2020	
5	Програмна реалізація системи	11.04.2020-20.04.2020	
6	Оформлення пояснювальної записки	01.05.2020-23.05.2020	
7	Передзахист	24.05.2020-26.05.2020	
8	Захист	15.06.2020-20.06.2020	

Студент Олександра КУЗЬМЕНКО

Керівник Юрій ГОРДІЄНКО

Анотація

Робота присвячена розробці програми виявлення сервісів з розумним контролем навантаження. Через зростання актуальності розподілених обчислень та кількості трафіку в мережі з’явилась гостра потреба в елементах, що можуть оптимізувати використання ресурсів, зменшити час відгуку, максимізувати пропускну здатність та запобігти перевантаженню будь-якого одного ресурсу та системи в цілому. Саме ці проблеми і вирішують сервіси для контролю навантаження.

Запропонована програма є сполучною ланкою серверної частини з клієнтом, контролює передачу трафіку, розподіляє навантаження по частинах back end та перевіряє дієздатність всіх окремих частин. Таким чином, серверна частина може стати більш гнучкою у налаштуванні, краще та простіше масштабуватись і мати більшу відмовостійкість.

Abstract

The work is devoted to development a service discovery program with smart load balancing. With the increasing relevance of distributed computing and the amount of network traffic, there is an urgent need for elements that can optimize resource utilization, reduce response time, maximize bandwidth, and prevent overloading of any one resource and system as a whole. It is these problems that solve the load balance services.

The proposed program is a linking part of the server side with the client, controls the transmission of traffic, distributes load by back ends and checks the performance of all individual parts. Suchwise, the server side can become more flexible in configuration, better and easier to scale, and more resilient.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЕКТУ

№ рядка	Формат	Позначення	Найменування	Кількість	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	ІАЛЦ.468300.001 ВП	Відомість дипломного проекту	1	
3	А4	ІАЛЦ.468300.002 ТЗ	Технічне завдання	3	
4	А4	ІАЛЦ.468300.003 ПЗ	Пояснювальна записка	57	
5	А4	ІАЛЦ.468300.004 Д1	Функціональна схема	1	
6	А4	ІАЛЦ.468300.005 Д2	Принципова схема	1	
7	А4	ІІАЛЦ.468300.006 Д3	Структурна схема	1	
8	А4	ІАЛЦ.468300.007 Д4	Текст програми	9	

				ІАЛЦ.468300.001 ВП	
	ПІБ	Підп.	Дата		
Розробн.	Кузьменко О.В.			Система виявлення сервісів з розумним контролем навантаження Відомість дипломного проекту	Лист 1 / Листів 1
Керівн.	Гордієнко Ю.Г.				
Консульт.					НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-62
Н/контр.	Сімоненко В.П.				
Зав.каф.	Стіренко С.Г.				

Технічне завдання
до дипломного проекту
на тему: «Система виявлення сервісів з розумним контролем навантаження»

Київ – 2020 року

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

В даному технічному завданні розглянуто розробку системи виявлення сервісів з розумним контролем навантаження.

Область застосування: оптимізація розподілу навантаження між декількома мережевими пристроями (наприклад, серверами).

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки служить завдання на виконання розробки системи виявлення сервісів з розумним контролем навантаження, затвердженою кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний Інститут».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка системи виявлення сервісів з розумним контролем навантаження.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами для розробки служать науково-технічна література з комп’ютерних технологій, публікації в періодичних виданнях, публікації в Інтернеті за даним питанням.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваного продукту

- Можливість підключення сервісів
- Оптимальний алгоритм балансування навантаження

5.2. Вимоги до програмного забезпечення

- Unix-подібні операційні системи на базі ядра Linux
- Мова програмування Golang
- Docker версії 19.03.8 і вище

5.3. Вимоги до апаратного забезпечення

- Комп’ютер на базі процесору Intel Pentium 3 і вище
- Оперативної пам’яті не менше 4 Гбайт
- Підключення до інтернету

Пояснювальна записка
до дипломного проекту
на тему: «Система виявлення сервісів з розумним контролем навантаження»

Київ – 2020 року

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

OSI	Абстрактна мережева модель для комунікацій і розробки мережевих протоколів
TCP	Протокол призначений для управління передачею даних у комп'ютерних мережах
client	Клієнт
backend	Програмна частина додатку
load balancer	Система балансування навантаження
proxy	Проміжний сервер, що з’єднує клієнта з програмною частиною
API	Прикладний програмний інтерфейс. Набір визначених методів для взаємодії різних компонентів
service mesh	Сервісна сітка. Спеціалізований інфраструктурний рівень для полегшення комунікацій між послугами між сервісами
DNS	Доменна система імен
JSON	Текстовий формат обміну даними між комп'ютерами
http	Протокол передачі даних, що використовується в комп'ютерних мережах
cookie	Інформація у вигляді текстових або бінарних даних, отриманих від веб-сайту на веб-сервері, яка зберігається у клієнта
RFC 7159	Поточна формальна специфікація JSON

ВСТУП

З кожним днем зростає потреба в більшій кількості обчислювальних потужностей, а також підвищуються вимоги до їх можливості витримувати навантаження, швидкодії та відмовостійкості на величезних обсягах даних[1].

Основним методом збільшення ресурсів є кластеризація: кілька серверів об'єднуються в кластер; навантаження між ними розподіляється за допомогою комплексу спеціальних методів, що називаються балансуванням.

Ефективність кластеризації безпосередньо залежить від того, як розподіляється (балансується) навантаження між елементами кластера. Тож на сьогодні ефективне балансування навантаження є одним із найважливіших питань[2][3].

Актуальність теми

Через зростання популярності використання кластерів та розподілених обчислювальних систем, виникає потреба в розробці методів та засобів, що дозволять прискорити та забезпечити високу відмовостійкість даних систем.

Мета і задачі дослідження

Метою роботи є розробка системи, що буде використовувати дані моніторінгу стану складної розподіленої обчислювальної системи в цілому та кожної обчислювальної одиниці окремо та відповідно до отриманих даних буде рівномірно розподіляти навантаження.

Для досягнення поставленої мети були поставлені наступні основні задачі:

- Провести аналіз наявних систем балансування навантаження;
- Розробити модель методу;
- Створити програмну реалізацію згідно з розробленою моделлю; Провести моделювання запропонованого методу;

- Дослідити та проаналізувати отримані характеристики під час моделювання.

Практичне значення

Запропонований метод полягає у ефективному та швидкому розподіленні навантаження серед елементів розподіленої системи, що дає можливість масштабування, підвищує відмовостійкість та дозволяє раціонально використовувати наявні ресурси.

РОЗДІЛ 1

ОГЛЯД СИСТЕМ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ

Розподілені системи — це набори обчислювальних вузлів, об’єднаних комунікаційною мережею. Кожний обчислювальний вузол має свої власні ресурси і працює під керуванням своєї операційної системи. Кластер серверів - це різновид розподілених систем: певна кількість серверів, об'єднаних в групу, що утворюють єдиний ресурс. Дане рішення дозволяє істотно збільшити надійність і продуктивність системи[4].

Головною задачею таких систем є виключення простою чи несправності системи в цілому[3]. Для забезпечення правильної та оптимальної взаємодії — використовуються певні алгоритми та методи балансування, що будуть надалі розглянуті у цій роботі.

В обчислювальній техніці, балансування навантаження - це метод розподілу завдань між декількома мережевими пристроями (наприклад, серверами) з метою оптимізації використання ресурсів, скорочення часу обслуговування запитів та підвищення пропускної здатності, горизонтального масштабування кластера: додавання та видалення пристроїв, а також забезпечення відмовостійкості (резервування)[5].

Системи балансування навантажень можуть бути представлені, як апаратне обладнання (наприклад, багатошаровий комутатор), так і як програмного забезпечення (наприклад, система доменних імен) .

На рис. 1.1 зображена спрощена структура функціювання системи балансування навантаження.

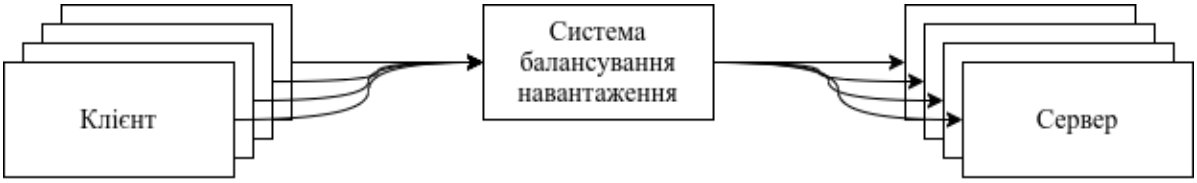


Рис. 1.1. Спрощена структура функціювання системи балансування навантаження

Система балансування навантаження є сполучною ланкою між клієнтами та серверною частиною. Її основними завданнями є:

- виявлення сервісів: автоматичне виявлення пристроїв та сервісів, що формують серверну частину;
- перевірка працездатності сервісів: перевірка доступності та функціональності;
- балансування завантаження: розподілення навантаження між працездатними сервісами.

Правильне використання балансування навантаження в розподіленій системі забезпечує ряд переваг:

- абстракція іменування: клієнту не потрібно знати інформацію про сервіси, всю роботу по з'єднанню виконує сама система балансування навантаження;
- відмовостійкість: завдяки перевірці працездатності сервісів та різних алгоритмічних методах, система балансування навантаження може ефективно обходити перевантажені сервера, або ті, що відмовили. Це означає, що системний адміністратор, як правило, може виправити помилки спокійно, аніж робити це в умовах надзвичайної ситуації;
- вартість та більша ефективність: система балансування навантаження може враховувати фізичне розташування елементів розподілених систем та максимально утримати трафік запитів у зонах, що збільшує

продуктивність (менша затримка) та зменшує загальну вартість системи (менше витрат на технічне забезпечення).

1.1. Різновиди систем балансування навантаження

Системи балансування навантаження, в залежності від рівня моделі OSI на якому вони працюють, поділяються на 2 типи[6]:

- L4 (траспортний рівень)

  L4, як випливає з назви, працює на 4 рівні моделі OSI. Коли клієнт робить запит, він створює TCP-з'єднання з системою балансування навантаження. Потім він використовує те саме TCP-з'єднання, що створив клієнт з ним, для з'єднання з одним із сервісів, як зображено на рис. 1.2.

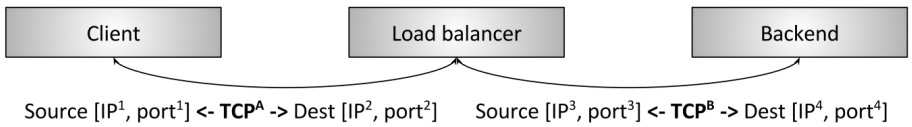


Рис. 1.2. Базове балансування навантаження TCP L4

- L7(прикладний рівень)

  L7, в свою чергу, працює на 7 рівні моделі OSI. Коли клієнт робить запит, він створює TCP-з'єднання з системою балансування навантаження. Потім система балансування створює нове TCP-з'єднання з одним із вищестоящих серверів, як показано на рис. 1.3. Таким чином, є 2 TCP-з'єднання порівняно з L4.

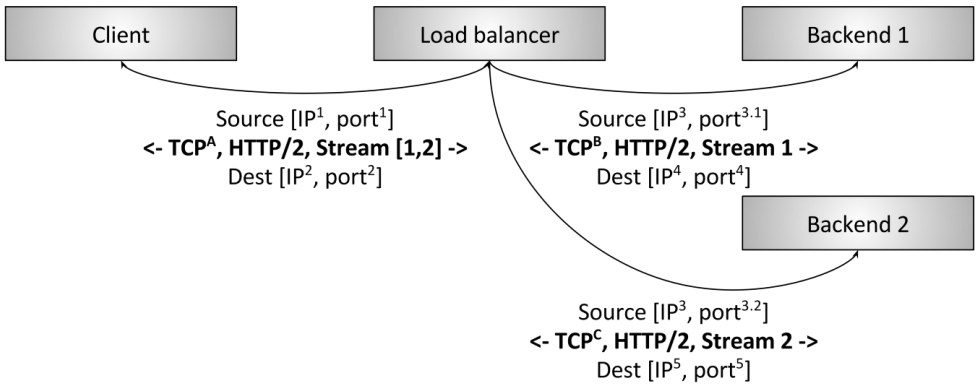


Рис. 1.3. Базове балансування навантаження HTTP/2 L7

1.2. Види топологій систем балансування навантаження

В залежності від того, як розгортаються системи балансування навантаження та як вони функціюють, можна виділити певні топології[7].

- Проміжний proxy

  Топологія з проміжним proxy, що показана на рис. 1.4, - один з найвідоміших способів балансування. До таких систем відносяться апаратні рішення Cisco, Juniper, F5; хмарні програмні рішення на кшталт Amazon ALB і NLB, Google Cloud Load Balancer; чисто програмні автономні рішення на кшталт HAProxy, NGINX і Envoy. Перевага схеми з проміжним proxy полягає в простоті. Користувачі підключаються до системи балансування навантаження через DNS і більше ні про що не турбуються. Недолік схеми в тому, що proxy навіть кластерізованний) - це єдина точка відмови, а також вузьке місце при масштабуванні. Крім того, проміжний proxy часто буває чорним ящиком, що ускладнює оперування та аналіз помилок та відмов в системі.

Рис. 1.4. Топологія системи балансування з проміжним proxy

- Крайовий proxy

  Дана топологія, що зображена на рис. 1.5 є варіантом топології з проміжним proxy, в якій система балансування навантаження є доступну через інтернет. В даному випадку система балансування зазвичай надає додаткові можливості «API-шлюзу» на кшталт TLS-переривань, обмеження швидкості, аутентифікації і просунутої маршрутизації трафіку. Переваги та недоліки такі ж, як у попередньої

топології. Зазвичай неможливо уникнути використання топології з крайовим proxy в великих, відкритих в інтернет розподілених системах.

Як правило, клієнтам потрібен доступ до системи по DNS з використанням різних мережевих бібліотек, які не контролюються власником сервісу (недоцільно використовувати прямо на клієнтах вбудовані клієнтські бібліотеки або топології з побічним proxy). Крім того, з міркувань безпеки краще мати єдиний шлюз, через який в систему надходить весь інтернет-трафік.

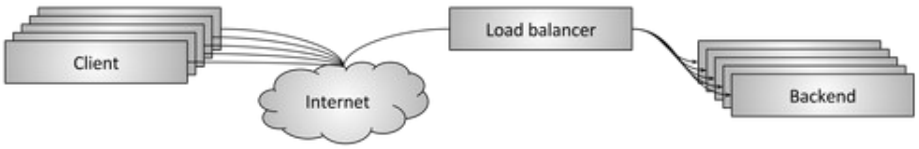


Рис. 1.5. Топологія системи балансування з крайовим proxy

- Вбудована клієнтська бібліотека

Щоб уникнути єдиної точки відмови або проблем з масштабуванням, властивих топологіям з проміжним proxy, в більш складних інфраструктурах застосовується вбудовування системи балансування навантаження за допомогою бібліотеки прямо в сервіси, як показано на рис. 1.6. Бібліотеки підтримують різні функції, найвідоміші і просунуті рішення - Finagle, Eureka / Ribbon / Hystrix і gRPC (заснований на внутрішній системі Google під назвою Stubby).

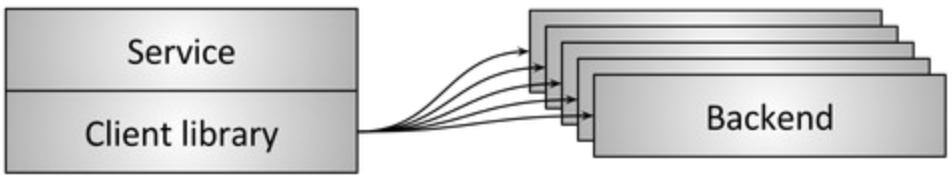


Рис. 1.6. Топологія системи балансування через вбудовану клієнтську бібліотеку

Головна перевага рішення на основі бібліотеки в тому, що функціональність системи навантаження повністю розподіляється по всіх клієнтах, а значить, єдиної точки відмови і труднощів масштабування немає. Основний недолік - бібліотека повинна бути реалізована на кожній мові, що використовується в елементах системи. В такому середовищі головна перешкода - вартість реалізації вкрай складною мережевої бібліотеки на багатьох мовах. А також, розгортання оновлення бібліотеки в архітектурі великого сервісу перетворюється на складний та проблематичний процес, що підвищує операційне когнітивне навантаження.

- Побічний proxy

Топологія з побічним proxy - варіант топології з вбудованою клієнтської бібліотекою, що зображено на рис. 1.7. В останні роки ця топологія була популяризована в якості service mesh. Завдяки цій топології можна отримати всі переваги варіанта з вбудованою бібліотекою без проблем з мовами програмування, але за рахунок невеликого збільшення затримки при переході до іншого процесу. Сьогодні найпопулярніші системи балансування навантаження з побічним proxy - це Envoy, NGINX, HAProxy, і Linkerd.

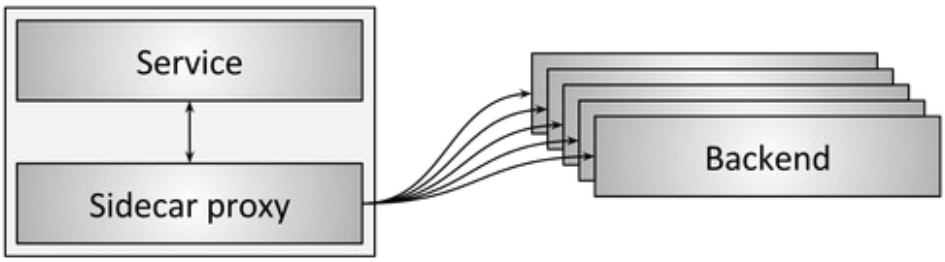


Рис. 1.7. Топологія системи балансування через побічний proxy

Переваги та недоліки різних топологій

- Топологія з проміжним proxy зазвичай найпростіша у використанні. Її недоліки: єдина точка відмови, труднощі масштабування і робота з чорним ящиком.

- Топологія з крайовим proxy аналогічна попередній, але за рахунок того, що вона працює як «API-шлюз», то є більш зручною у використанні.
- Топологія з вбудованою клієнтської бібліотекою має кращу продуктивність і масштабованість, але страждає від необхідності реалізувати бібліотеку на кожній мові і оновлювати бібліотеку по всіх сервісів.
- Топологія з побічним proxy працює не так добре, як попередня, але не має її обмежень.

1.3. Алгоритми і методи балансування

Існує багато різних алгоритмів і методів балансування навантаження. При реалізації алгоритмів балансування необхідно знайти баланс між затримками при розподіленні завдань і обробкою самих завдань, тобто потрібно виходити з специфіки та характеристик конкретного проекту, його цілей, типу завдання, топології мережі і вузлів, вартості розгортання рішення[8].

Основними цілями систем балансування навантаження є:

- **справедливість**: гарантоване надання необхідних системних ресурсів для кожного запиту та раціональне розподілення;
- **ефективність**: всі елементи розподіленої системи, що можуть виконувати обробку запитів, повинні бути зайняті максимально та рівномірно, тобто кожен з серверів має не простоювати чи не бути перезавантаженим;
- **скорочення часу виконання запиту:** забезпечення мінімальний часу між початком обробки запиту (або його постановкою в чергу на обробку) і його завершення;
- **скорочення часу відгуку:** мінімізування час відповіді на запит користувача;

- **передбачуваність**: очікуваний результат роботи системи навантаження виправдовує доцільність та ефективність використання для вирішення поставлених завдань;
- **рівномірне навантаження** системних ресурсів системи;
- **можливість масштабування**: змога алгоритму зберігати працездатність при збільшенні чи зменшенні кількості елементів системи та/чи навантаження.

За схемою роботи, системи балансування навантаження можна розділити на такі типи: статичні, динамічні та адаптивні[8][9].

Статичні алгоритми балансування не слідкують за станом розподіленої системи, рішення про вибір кожного наступного є сталим та не змінюється не залежно від того, в якому стані перебувають елементи розподіленої системи.

Прикладом такого типу алгоритму може слугувати Round Robin, або алгоритм кругового обслуговування. Система балансування навантаження ітеративно проходить по круговому циклу через весь список сервісів. Перша задача передається першому елементу. друга другому і так до останнього, потім знову спочатку. Однією із найбільш розповсюджених реалізацій цього алгоритму є робота DNS-серверу, що повторює IP адрес вузлів в необхідній пропорції розподілення навантаження.

Також до цього типу алгоритмів відносять наступні: Simulated Annealing or Genetic Algorithms (генетичний алгоритм) та Weighted Round Robin.

Динамічні алгоритми балансування навантаження, в свою чергу, враховують поточний стан системи: кількість активних підключень до кожного елементу та їх завантаженість, загальний об’єм ресурсів, пропускну здатність лінії тощо. Обробка такої кількості інформації збільшує навантаження на систему балансування та потребує більше ресурсів.

Гарним прикладом динамічної системи балансування може слугувати такий алгоритм, як Least Connections, що для розподілу навантаження враховує

кількість підключень, підтримуваних серверами в поточний момент часу та обирає сервер з найменшою кількістю активних підключень.

Вдосконалений варіант цього алгоритму називається Weighted Least Connections, що призначений для використання в кластерах, сервери якого мають різні технічні характеристики та різну обчислювальну потужність. Він враховує при розподілі навантаження не тільки кількість активних підключень, а й ваговий коефіцієнт серверів.

Також до цього типу можна віднести наступні алгоритми: Richard Karp для P2P, gLite (WMS), ARC (NorduGrid), Sticky Sessions.

Адаптивні ж алгоритми використовують поєднання двох вище наведених типів алгоритмів. Залежно від поточного стану та характеристик елементів розрахункової системи обирається найбільш слуший алгоритм. Наприклад, Derbal використовує масштабоване планування основане на ентропії розрахункової системи, а вже за допомогою ланцюгів Маркова сумує динамічні служби в мережі. Для прикладу є система з підтримкою поколінь планування (the introduction of the Generational Scheduling) з реплікацією завдань. Цей алгоритм адаптується до змін в продуктивності з реструктуризацією завдань[8].

У табл. 1.1. наведено порівняльну характеристику найбільш використовуваних методів балансування навантаження за основними критеріями[8][9].

Таблиця 1.1 – Порівняння методів балансування навантаження

	Через єдиний пристрій балансування	NLB-кластери	Без єдиного пристрою балансування	2-й рівень OSI	3-й рівень OSI	Проксі 4-й рівень OSI	Проксі 7-й рівень OSI	HTTP Redirect 7-й рівень OSI
Ефективність розподілу навантаження	+	+	-	+	єдин / без єдин.	+	+	+
Показники завантаження вузлів (прямі,непрямі)	прямі, непрямі	прямі, непрямі	-, або непрямі	непрямі	-, рідко непрямі	прямі, непрямі	непрямі	-
Врахування гео-положення вузлів	+	-/+, можливо, у деяких реалізаціях	-/+	-	-/+	-	-	-
Можливості маштабування	-	+	++	+	єдин / без єдин.	-	-	++
Контентно залежний аналіз, кешування	+	-	-	-	-	+	+	+, але без кешування
Забезпечення довгострокових сесій	+	+	-	+	єдин / без єдин.	+	+	-
Різні потужності обчислювальних вузлів	+	+/-	+	+	єдин / без єдин.	+	+	+
Різні платформи обчислювальних вузлів	+	+/-	+	+	+	+	+	+
Незалежність від типу мережі	+	-/+	+	-	+	-	-	-
Автоматичне визначення і видалення непрацюючих вузлів	+	-/+ з НА	-/+	-	-/+рідко	+	+	–
Незалежність від прикладного протоколу	+/-	+/-	+	+	+	+	-	-
Затрати ресурсів на балансування	+	+	+	+	+	+	- -	++

ВИСНОВКИ ДО РОЗДІЛУ 1

В ході виконання даного розділу моєї дипломної роботи, було розглянуто загальну структуру, топології та види систем балансування навантаження, а також наявні алгоритми балансування.

Системи балансування навантаження є одним із найважливіших елементів розподілених систем, що дозволяють розподіляти навантаження поміж елементів, можуть контролювати їх стан та стан системи в цілому, а також спрощують масштабування та зміни в системі.

Не існує ідеального та оптимального для будь-яких ситуацій алгоритму балансування: кожен має недоліки та переваги та підходить для певних груп задач.

Тому було прийнято рішення розробити власну систему балансування на основі нейронних мереж, що буде гнучкою та зможе адаптуватись під різні розподілені кластерні системи, аналізуючи кількість трафіку та завантаження системи.

РОЗДІЛ 2

АНАЛІЗ ЗАСОБІВ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Для створення гнучкої та адаптивної системи балансування навантаження, що зможе на підставі аналізу поточної ситуації приймати рішення та рівномірно розподіляти трафік, було обрано технології нейронних мереж.

2.1. Загальний опис нейронних мереж

Нейронні мережі - це набір алгоритмів, створених на основі моделі біологічної нейронної мережі людини, які розроблені для розпізнавання шаблонів. Вони інтерпретують сенсорні дані через своєрідне машинне сприйняття, маркування або групування вихідних даних. Шаблони, які вони розпізнають, є чисельними, містяться у векторах, у які повинні бути переведені всі дані реального світу, будь то зображення, звук, текст чи часовий ряд[10].

Нейронна мережа будується з шарів з вузлів, які ієрархічно з'єднані в мережу, де вихід з одного вузла є входом для інших. Кожен вузол приймає зважений вхід, активує активаційну функцію для суми входів, та генерує вихід, як схематично зображено на рис. 2.1.

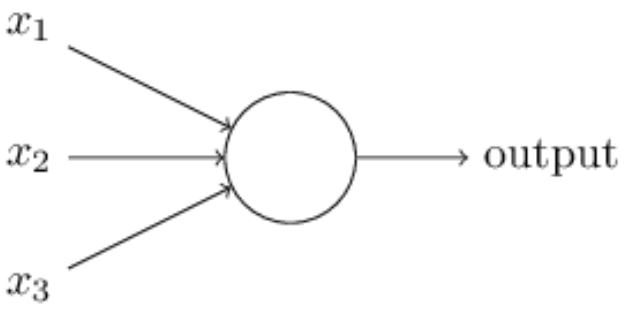


Рис. 2.1. Приклад вузла нейронної мережі

2.1.1. Штучний нейрон

Основним компонентом нейронної мережі є вузол, що називається штучним нейроном. Всі нейрони з'єднані між собою. Сигнали передаються по зваженим зв'язкам, з кожним з яких пов'язаний ваговий коефіцієнт (або вага) [11][12].

Множина вхідних сигналів x1, x2,…xn, поступає на штучний нейрон. Ці вхідні сигнали, в сукупності позначаються вектором X. Кожен сигнал множиться на відповідну вагу w1, w2,…, wn, що в сукупності позначаються вектором W, та поступає на суматор, позначений Σ. Цей блок, складає зважені входи алгебраїчно, створюючи вихід, які перетворюється активаційною функцією F, що вираховується за певною формулою, і дає вихідний нейронний сигнал[11].

2.1.2. Функції активації

Функція активації F(NET), де NET це вихід суматору, може бути

- пороговою бінарною функцією (1.1),

$$OUT = \begin{cases} 1, NET \geq T \\ 0, NET < T \end{cases}, \quad (1.1)$$

де T – деяка постійна порогова величина, або ж функція, що точніше моделює нелінійну передавальну характеристику нейрона;

- лінійною обмеженою функцією(1.2);

$$OUT = \begin{cases} 1, NET \geq T \\ NET, 0 \leq NET < T \\ 0, NET < 0 \end{cases}, \quad (1.2)$$

- функцією гіперболічного тангенса(1.3),

$$OUT = th(C \times NET) \ , \qquad (1.3)$$

де C > 0 – коефіцієнт ширини сигмоїди по осі абсцис (зазвичай, C=1);

- сигмоїдною (S-подібною) або логістичною функцією(1.4).

$$\mathrm{OUT} = \frac{1}{1 + e^{-C \cdot \mathrm{NET}}} \ , \qquad (1.4)$$

Найбільш популярною функцією, стала сигмоїдна, через її певні властивості. А саме:

- здатність надавати більшої ваги слабким сигналам більше, ніж великим, та підсилювати їх;
- монотонність і диференційованість на всій осі абсцис;
- простий вираз для похідної, що надає можливості застосовувати велику кількість різних алгоритмів.

2.1.3. Складена структура

У повній нейронній мережі, як було зазначено вище, знаходиться багато взаємозв'язаних між собою вузлів. Структури таких мереж можуть бути абсолютно різних форм, але логіка побудови завжди зберігається:

- вхідний шар, де мережа приймає зовнішні вхідні дані
- прихований шар (може бути декілька прихованих шарів)
- вихідний шар, де виводиться результат роботи

Приклад такої структури приведено на рис. 2.2.

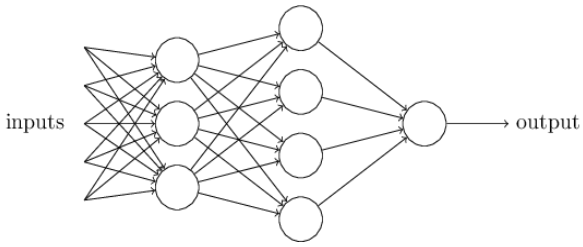


Рис. 2.2. Приклад структури нейронної мережі

На сьогодні, було створено неймовірну кількість структур та їх модифікацій для нейронних мереж. Кожна з яких спрямована на вирішення конкретного типу задач. Найбільш відомі типи таких структур показані на рис. 2.3.[12].

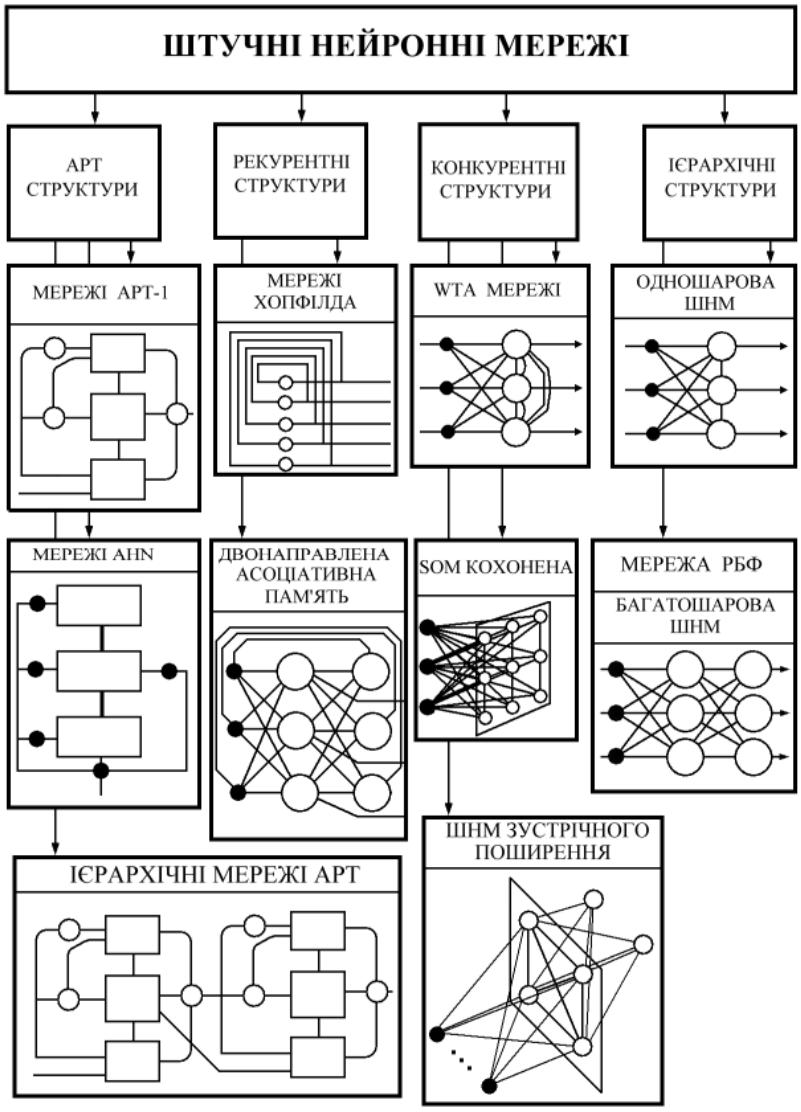


Рис. 2.3. Нейронні структури

2.2. Класифікація нейронних мереж та їх властивості

Штучні нейронні мережі надзвичайно різноманітні за конфігураціями[12].

Нейронні мережі розподіляються на типи залежно від структури зв’язків між нейронами мережі, або за використаними формальними нейронами.

Нейронні мережі можна розглядати як однонапрямний граф зі зваженими ребрами, де нейрони є вузлами. По архітектурі зв'язків НМ можуть бути згруповані в два класи:

- мережі прямого поширення, у яких графи не мають петель;
- рекурентні мережі, або мережі зі зворотними зв'язками.

За способом подачі інформації до входів нейронної мережі розрізняють:

- подачу сигналів на синапси вхідних нейронів;
- подачу сигналів до виходів вхідних нейронів;
- подачу сигналів у вигляді ваги синапсів вхідних нейронів;
- адитивну подачу на синапси вхідних нейронів.

За способом забору інформації з виходів нейронної мережі розрізняють:

- зняття з виходів вихідних нейронів;
- зняття з синапсів вихідних нейронів;
- зняття у вигляді значень ваги синапсів вихідних нейронів;
- адитивне зняття із синапсів вихідних нейронів.

За організацією навчання поділяють навчання нейронних мереж з вчителем (supervised neural networks), без вчителя (nonsupervised) та навчання з підкріпленням (reinforsment learning).

За способом навчання поділяють навчання на два типи:

- за входами: навчальний приклад складається лише з вектору вхідних сигналів;
- за виходами: навчальний приклад складається з вектору вхідних і відповідних до них вихідних сигналів.

За способом подання прикладів розрізняють:

- подання одиночних прикладів: навчання відбувається після подання кожного прикладу;
- подання “сторінки” прикладів: навчання відбувається після подання "сторінки" (множини) прикладів, на підставі аналізу одразу усіх їх.

2.3. Навчання нейронної мережі

Основною задачею навчання нейронної мережі є щоб для деякої множини входів X давати бажану множину виходів Y. Кожна така множина називається вектором. Навчання здійснюється шляхом послідовного отримання вхідних векторів з одночасним налагодженням ваги. В процесі навчання ваги нейронної мережі поступово набувають таких значень, щоб кожен вхідний вектор виробляв вихідний вектор. Основним критерієм оцінки ефективності навчання є цільова функція, що дає функціональний опис того, наскільки модельний вихід збігається з ідеальним для даної вхідної вибірки.

Розрізняють алгоритми навчання з вчителем, без вчителя та навчання з підкріпленням[11].

2.3.1. Навчання з вчителем

Навчання з вчителем припускає, що для кожного вхідного вектора X існує цільовий вектор Y, що є необхідним виходом. Разом вони називаються навчальною парою. Зазвичай, мережа навчається для певної кількості таких навчальних пар. Навчання складається з декільком етапів:

- зчитування вхідного вектор X;
- обчислення виходу мережі Y;
- порівняння Y з відповідним цільовим вектором Yt;
- подання різниці D ~ YT – Y за допомогою зворотного зв'язку в мережу для корегування ваги W відповідно до алгоритму, прагнучого мінімізувати помилку ε.

Дана процедура повторно виконується до тих пір, поки сумарна помилка для всієї навчальної множини не досягне заданого низького рівня.

2.3.2. Навчання без вчителя

Навчання без вчителя, розвинене Кохоненом і багатьма іншими, не потребує цільового вектора для виходів. Навчальна множина складається лише з вхідних векторів. Навчальний алгоритм налагоджує вагу мережі так, щоб виходили узгоджені вихідні вектори при досить близьких значеннях вхідних векторів. Процес навчання виділяє статистичні властивості навчальної множини і групує схожі вектори в класи.

Залежно від завдання модель систематизує дані по-різному, відповідно до цього виділяють наступні типи завдань навчання без учителях[11]:

- кластеризація

- пошук правил асоціації
- заповнення пропущених значень
- скорочення розмірності
- візуалізація даних

2.3.2.1. Кластеризація

Завдання кластеризації

Даний тип навчання без вчителя призначений для поділу безлічі об'єктів на підмножини, або кластери, з максимально однорідних та схожих між собою об’єктів, коли як підмножини сильно відрізняються за певними ознаками.

Вихідна інформація може надаватися у вигляді матриці відстаней.

Методи рішення

- Графові алгоритми кластеризації
- Статистичні алгоритми кластеризації
- Ієрархічна кластеризація або таксономія
- Нейронна мережа Кохонена
- Нейронна мережа зустрічного поширення
- Метод радіальних базисних функцій

Кластеризація може грати допоміжну роль при вирішенні задач класифікації і регресії (що відносяться до категорії навчання з учителем). В даному випадку, вибірка розподіляється на кластери, до кожного з яких, застосується який-небудь зовсім простий метод.

Всі алгоритми кластеризації спираються на гіпотезу компактності, згідно з якою в просторі ознак все близькі об'єкти повинні належати до одного кластеру, а все різні об'єкти - відповідно до різних кластерів, як зображено на рис. 2.4.

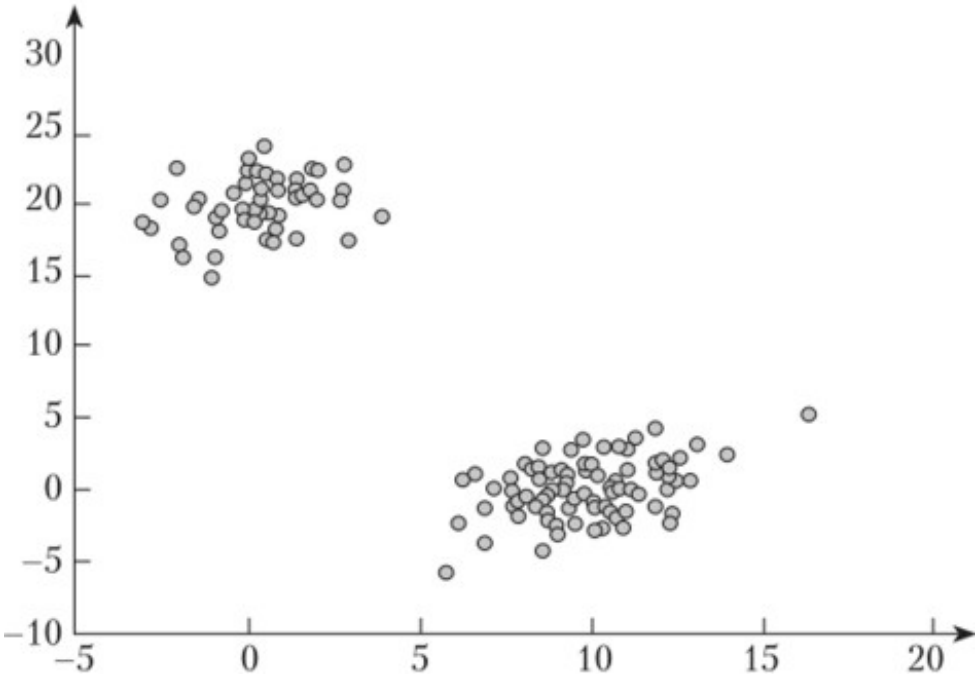


Рис. 2.4. Приклад двох очевидних кластерів

2.3.2.2. Пошук правил асоціації

Завдання пошуку правил асоціації

Даний тип навчання без вчителя призначений для знаходження таких ознак, як і значення цих ознак, які особливо часто (не випадково часто) зустрічаються в описах об'єктів. Вихідна інформація в свою чергу подається у вигляді опису ознак об’єктів.

Методи рішення

- Аналіз ринкових кошиків
- Завдання заповнення пропущених даних

Вихідна інформація може надаватися у вигляді описів ознак. Значення деяких ознак для деяких об'єктів можуть бути відсутні. Такі випадки часто виникають на практиці. Однак багато методів аналізу даних вимагають, щоб вхідні матриця описів ознак була заповнена повністю. Для заповнення відсутніх значень часто застосовують наступний підхід. Вважаючи цей показник цільовим, будують алгоритм, який прогнозує його значення в залежності від інших ознак. Пропущені значення заповнюють прогнозами. Ця операція проробляється з усіма ознаками, що мають пропущені значення. Дане завдання вирішується методами навчання з учителем:

- якщо ознака кількісний, застосовуються методи відновлення регресії;
- якщо ознака якісний (номінальний), застосовуються методи класифікації.

2.3.2.3. Скорочення розмірності

Завдання пошуку правил асоціації

Вихідна інформація може надаватися у вигляді описів певних ознак, причому число ознак може бути досить великим. Завдання полягає в тому, щоб представити ці дані в просторі меншої розмірності, по можливості, зменшити втрати інформації.

Методи рішення

- Метод головних компонент
- Метод незалежних компонент
- Багатовимірне шкалювання

2.3.2.4. Візуалізація даних

Завдання візуалізації даних

Деякі методи кластеризації та зниження розмірності будують уявлення вибірки в просторі розмірності два. Це дозволяє зображувати багатовимірні дані у вигляді плоских графіків і аналізувати їх візуально, що сприяє кращому розумінню даних і самої суті розв'язуваної задачі.

Методи рішення

- Дендрограмма
- Само організована карта Кохонена
- Пружні карти
- Карта подібності

У навчанні без учителя складно обчислити точність алгоритму, через відсутність «правильних відповідей» для порівняння. Але через те, що розмічені дані часто ненадійні або їх занадто дорого отримати, навчання з учителем може бути занадто витратним та не виправдовувати себе. У таких випадках, даний спосіб обрахунків може видати хороші результати, завдяки моделі свободу дій для пошуку залежностей.

2.3.3. Навчання з підкріпленням

Навчання з підкріпленням — це вид навчання, основною метою якого, є максимізація результатів дії, які повинні виконувати програмні агенти в певному середовищі.

Алгоритми RL можуть починатися виконуватися з 0. У процесі навчання агент, що приймає рішення, дізнається, що робити - як зображувати ситуації на дії, щоб максимально збільшити значення винагороди. Агенту не вказано

прямо, які дії потрібно вжити, але натомість він повинен виявити, яка дія приносить найбільшу винагороду за допомогою спроб та помилок[13].

Схема роботи навчання зображена на рис. 2.5.

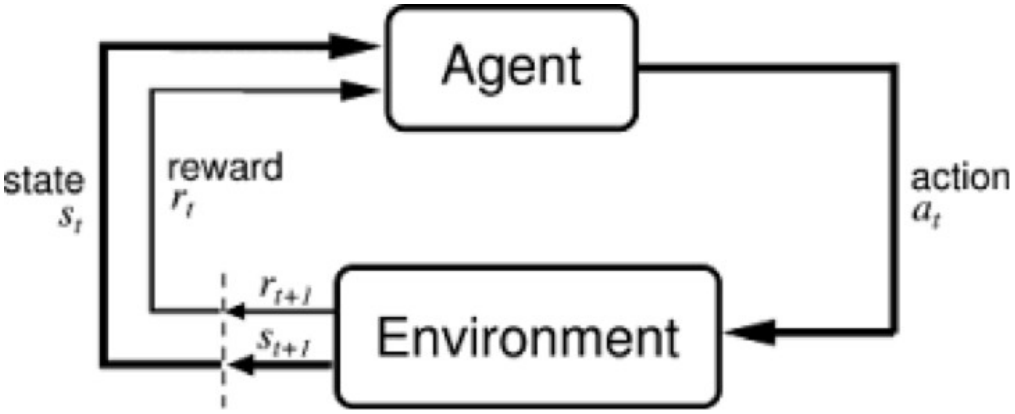


Рис. 2.5. Схематичне зображення роботи навчання з підкріпленням

2.3.3.1. Основні визначення

Агент (agent): той, що здійснює дії, тобто сам алгоритм пошуку найкращого рішення.

Дія (A): сукупність усіх можливих рухів, які може здійснити агент. Дія майже не пояснює себе, але слід зазначити, що агенти зазвичай вибирають із списку дискретних можливих дій.

Середовище: світ, в якому агент існує та приймає рішення. Середовище сприймає поточний стан та дії агента як вхідні дані та повертає як вихідну винагороду агента та його наступний стан.

Стан (S): конкретна і негайна ситуація, в якій опиняється агент; тобто конкретне місце та момент, миттєва конфігурація.

Нагорода (R): відгук, за допомогою якого вимірюється успіх чи невдача дій агента в даному стані. Вони ефективно оцінюють дію агента.

Політика (π): стратегія, яку агент використовує для визначення наступної дії на основі поточного стану. Вона поєднує дію, з найкращими винагородами, з відповідними станами.

Q-значення: оцінка вартості найкращої дії з урахуванням усіх майбутніх винагород. Саме вона підкріплює навчання в процесах прогнозування та контролю[13].

ВИСНОВКИ ДО РОЗДІЛУ 2

В ході виконання даного розділу дипломної роботи, було розглянуто загальну структуру, особливості та види навчання нейронних мереж.

Нейронні мережі — це чудовий засіб, для вирішення складних задач.

Оскільки, стан та характеристика розподіленої системи, для якої буде використовуватись системи балансування навантаження, завчасно невідомі, то для реалізації гнучкого та адаптивного алгоритм балансування, була обране навчання з підкріпленням. Такий алгоритм зможе підлаштуватись під будь-яку структуру розподіленої системи та дозволить розробити програму з гнучким інтерфейсом та можливістю розширення функціоналу.

РОЗДІЛ 3

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Вибір платформи

Більшість серверів розташована на Unix подібних системах, тому що сама оболонка даної системи зручна в використані. Представлена організація ОС дає змогу розділення користувачів на окремі процеси і надавання їм прав для використання системи, що робить їх безпечнішими. Тож Unix була обрано як операційна система для розробки, а саме дистрибутив Ubuntu.

Оскільки, для системи балансування навантаження необхідні певні залежності, то для створення зручного та простого встановлення продукту було обрано Docker, як інструмент ізоляції та контейнерізації програми.

Docker - це програмна платформа для швидкої розробки, тестування і розгортання додатків. Docker упаковує ПО в стандартизовані блоки, які називаються контейнерами, що включають в себе все необхідне для роботи програми: бібліотеки, системні інструменти, код і середу виконання. Завдяки Docker можна швидко розгортати і масштабувати додатки в будь-якому середовищі і зберігати впевненість в тому, що код буде працювати.

Docker забезпечує рішення та спрощення наступних аспектів:

- пришвидшення процесу розробки;
- зручна інкапсуляція програми;
- однакова поведінка на усіх серверах;
- простий і чіткий моніторинг;
- зручність масштабування.

3.2. Вибір мови програмування

Зараз існує безліч мов програмування, що покривають та задовольняють безліч потреб. Враховуючи запланований функціонал та високу потребу в стабільний та швидкій роботі, було обрано компільовану мову Golang, що містить вбудовані засоби для паралельних обчислень та віддаленого керування пакунками[14].

Використання Golang має ряд переваг над іншими мовами програмування, таких як[14]:

- Швидка компіляція та швидкість виконання
- Відсутність потреби в віртуальній машині
- Портативність
- Прості та зручні у використанні паралельні процеси
- Інтерфейси дозволяють забезпечити слабко пов'язані системи
- Автоматичне збирання сміття
- Безпека пам’яті
- Незалежна робота з помилками
- Великі вбудовані бібліотеки

3.3. Вибір інструментів

Для реалізації даної системи, було прийнято рішення про використання таких пакетів:

- net (http, url)
- context
- encoding/json
- sync
- log
- time
- math

Розглянемо кожен з пакетів окремо.

3.3.1. net

Пакет net забезпечує портативний інтерфейс для мережевого вводу-виводу, включаючи TCP / IP, UDP, роботу з доменною системою імен та з сокетами Unix.

Пакет HTTP Go використовує структуру під назвою "Клієнт" для управління внутрішніми комунікаціями через HTTP (S). Клієнти - це безпечні для кон'юнктури об'єкти, які містять конфігурацію, керують станом TCP, обробляють файли cookie тощо. При виконанні http запитів, використовується http.DefaultClient, змінна пакета, яка визначає конфігурацію за замовчуванням для клієнта.

Приклад використання:

```
server := http.Server{
        Addr: fmt.Sprintf(":%d", port),
        Handler: http.HandlerFunc(lb),
}
```

3.3.2. context

Пакет context надає інструменти, що дозволяють легко передавати значення в області видимості запиту, сигнали скасування і крайні терміни (deadlines) через API всім goroutine, які беруть участь в обробці запиту.

Завдяки даному пакету організовуються підключення до усіх сервісів, що обслуговуються системою балансування навантаження.

Приклад використання:

```
retries := GetRetryFromContext(request)
    if retries < 3 {
        //...
        ctx := context.WithValue(request.Context(), Retry, retries+1)
        proxy.ServeHTTP(writer, request.WithContext(ctx))
    }
```

3.3.3. encoding/json

Пакет encoding/json реалізує кодування та декодування JSON, як визначено в RFC 7159. Завдяки цьому, система балансування навантаження може отримувати дані про стан сервісів з окремої системи моніторингу.

Приклад використання:

```
type BackendLoad struct {
	MemoryUsage float64 `json:"memoryUsage"`
	CpuUsage    float64 `json:"cpuUsage"`
}
// ...
resp, err := http.Get(target)
var data BackendLoad
if err = json.NewDecoder(resp.Body).Decode(&data); err != nil
{
	continue
}
resp.Body.Close()
```

3.3.4. sync

Пакет sync забезпечує основні примітиви синхронізації, такі як блоки взаємного виключення. Зокрема, з цього пакету був використаний RWMutex, що засобом блокування взаємного виключення зчитування чи записування.

Приклад використання:

```
func (b *Backend) SetAlive(alive bool) {
	b.mux.Lock()
	b.Alive = alive
	b.mux.Unlock()
}
// IsAlive returns true when backend is alive
func (b *Backend) IsAlive() (alive bool) {
	b.mux.RLock()
	alive = b.Alive
	b.mux.RUnlock()
	return
}
```

3.3.5. log

Даний пакет дозволяє виводити системну інформацію про роботу сервера. За допомогою методів цього пакету виводяться дані про дії програми, що дозволяє краще аналізувати та бачити процеси, що відбуваються під час балансування.

Приклад використання:

```
if err != nil {
        log.Printf("Failed to get info from %s, error: [%s]\n", b.URL.Path, err)
}
log.Printf("Got data {%#v} from %s\n", data, b.URL)
```

3.3.6. time

Даний пакет забезпечує функціональність для вимірювання та відображення часу. Зокрема, були використані такі методи, як NewTicker, Second, Minute, Milliseconds, Now.

NewTicker був використаний для регулювання інтервалів для періодичної перевірки працездатності сервісів.

Приклад використання:

```
func healthCheck() {
        t := time.NewTicker(time.Minute * 2)
        // ...
}
```

Методи Second, Minute, Milliseconds, Now були використані для отримання поточного значення часу.

Приклад використання:

```
timeout := 2 * time.Second
time.Now().UnixNano()
```

3.3.7. math

Даний пакет забезпечує основні константи та математичні функції. Завдяки цьому цьому пакету був спрощений код, зокрема алгоритм балансування.

Приклад використання:

```
sd := 0.0
for i := 0; i < size; i++ {
        sd += math.Pow(actions[i].nn.mean-mean, 2)
}
sd = math.Sqrt(sd / float64(size))
```

3.4. Основні рішення з реалізації системи

Для легшої імплементації системи, було прийнято рішення створити docker контейнер, що буде містити скомпільовану програму та зможе бути використаний на будь-якій машині з Unix подібною операційною системою.

Створення контейнеру складається з декількох етапів:

- створення Dockerfile
- створення image
- запуск контейнеру

3.4.1. Створення Dockerfile

Dockerfile - це текстовий файл, який включає інструкції по створенню образу docker. Dockerfile визначає операційну систему, яка буде лежати в основі контейнера, а також мови, змінні середовища, розташування файлів, мережеві порти та інші необхідні компоненти і дії контейнера після його запуску. Кожна інструкція складає шар docker образу. Можливі інструкції та їх функції перераховані в табл. 3.1[15].

Таблиця 3.1 – Можливі інструкції в Dockerfile

Інструкція	ЇЇ функція
FROM	вказати батьківське зображення
WORKDIR	встановити робочий каталог для будь-яких команд, які слідують у Dockerfile
COPY	скопіювати файли чи директорії
ADD	як COPY, але також здатна обробляти віддалені URL-адреси та розпаковувати скомпресовані файли
RUN	виконувати будь-які команди в новому шарі та фіксувати результати
ENTRYPOINT	визначати команду, яка завжди буде виконуватися при запуску контейнера (за замовчуванням / bin / sh -c)
CMD	визначати аргументи передані до команди ENTRYPOINT. Якщо вона не встановлена , CMD буде командою, яка буде виконуватися при запуску контейнера
EXPOSE	визначити, які порти будуть доступні ззовні
ENV	визначити значення змінних середовища в контейнері

Приклад створеного Dockerfile:

```
FROM golang:1.13 AS builder
WORKDIR /app
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o lb .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
```

```
WORKDIR /root
COPY --from=builder /app/lb .
ENTRYPOINT [ "/root/lb" ]
```

3.4.2. Створення image

Після конфігурації Dockerfile використовується утиліта docker build для створення образу на його основі.

Docker образ, або docker image, - це файл, який містить специфікації програмних компонентів, які виконуються в контейнері. Як тільки образ створений, змінити його не можна (тільки додати новий шар).

3.4.3. Запуск контейнеру

Існуючі створені docker образи можна запускати за допомогою утіліти docker run.

Docker контейнер — це запущені образи. З якими є можливість виконувати безліч дій: можна запускати, спиняти, переміщувати і видаляти, дивитись логи та виконувати певні команди всередині самого контейнеру.

3.5. Огляд організації програми

Golang не підтримує ООП, але пропонує доволі зручну альтернативу у вигляді структур: типом даних, що може містити іменовані поля та методи.

Були створені наступні структури:

- ServicePool
- Backend
- BackendLoad
- Education

3.5.1. ServicePool

Дана структура представляє собою пул з сервісів для спрощення роботи зі списком сервісів.

ServicePool містить наступні поля:

- backends — список всіх підключених сервісів до системи балансування навантаження.

Методи даної структури:

- AddBackend — додавання нового виявленого сервісу до списку backends
- MarkBackendStatus — встановлення статусу вказаного сервісу
- HealthCheck — перевірка доступності виявлених сервісів
- GetBestPeer — виклик алгоритму балансування для отримання найбільш незавантаженого сервісу

3.5.2. Backend

Дана структура представляє собою зображення сервісу.

Backend містить наступні поля:

- URL — адреса сервісу
- Alive — стан сервісу
- mux — мютекс для контролю доступу до поля Alive
- ReverseProxy — proxy для ретрансляції запитів клієнтів
- load — поточне завантаження сервісу
- nn — структура Education для алгоритму нейронної мережі

Методи даної структури:

- SetAlive — встановлення значення поля Alive
- IsAlive — отримання значення поля Alive
- Update — оновлення даних в nn

3.5.3. BackendLoad

Дана структура використовується для отримання даних в форматі JSON та перетворення їх в зручну форму для подальшого використання.

BackendLoad містить наступні поля:

- MemoryUsage — відсоткова кількість використаної операційної пам’яті з усієї можливої
- CpuUsage — відсоткова кількість завантаження процесору

Дана структура не має методів. Використовується в методі GetBestPeer структури ServicePool для оновлення даних кожного сервісу.

3.5.4. Education

Дана структура містить необхідні дані для роботи алгоритму розумного балансування навантаження.

Education містить наступні поля:

- mean — середнє значення навантаження сервісу
- N — кількість разів, коли сервіс був обраний як найкращий

Дана структура не має методів. Використовується в методі GetBestPeer структури ServicePool для аналізу кожного сервісу.

3.6. Огляд алгоритму балансування

Як було зазначено у висновках до 2 розділу, було обрано навчання з підкріпленням. Для вибору найбільш оптимального алгоритму для початку треба проаналізувати проблематику завдання.

3.6.1. Проблематика завдання

Проблема “Multi-armed bandit” використовується в навчанні з підкріпленням для формалізації поняття прийняття рішень в умовах невизначеності. У проблемі з багатогранним бандитом агент обирає між k різними діями і отримує винагороду, виходячи з обраної дії.

Для вибору дії агентом, припускається, що кожна дія має окремий розподіл винагород і є принаймні одна дія, яка приносить максимальну чисельну винагороду. Таким чином, розподіл ймовірності винагород, що відповідають кожній дії, є різним і невідомий агенту. Отже, мета агента - визначити, яку дію вибрати, щоб отримати максимальну винагороду після заданого набору випробувань.

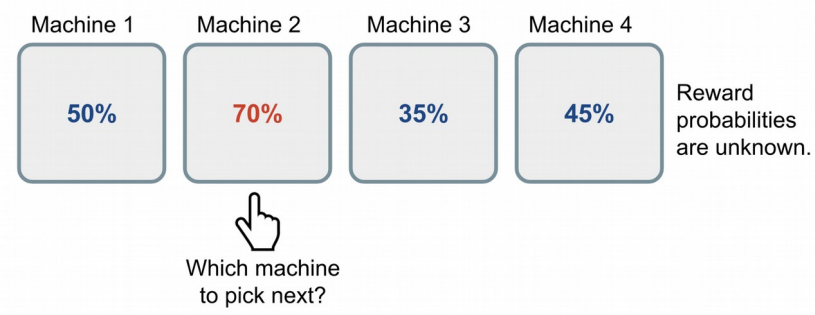


Рис. 3.1. Приклад проблеми “Multi-armed bandit”

3.6.2. Оцінка вартості дій

Для того, щоб агент вирішив, яка дія приносить максимальну винагороду, повинно бути визначено значення виконання кожної дії. Ми використовуємо поняття ймовірності для визначення цих значень за допомогою функції дія-значення.

Значення вибору дії визначається як очікувана винагорода, отримана при здійсненні цієї дії з набору всіх можливих дій. Оскільки значення вибору дії агенту невідомі, тож ми використовуємо метод "середня вибірка", щоб оцінити значення дії[16].

3.6.3. Математичне визначення проблеми

Елемент проблеми Бернуллі можна описати як кортеж ⟨A, R⟩[16], де:

- У нас є K машин з ймовірністю винагороди $\{\theta_1,\ldots,\theta_K\}$.
- На кожному кроці t ми робимо дію, а на одному ігровому автоматі і отримуємо винагороду r.
- A - це сукупність дій, кожна з яких стосується взаємодії з одним ігровим автоматом. Значення дії a - очікувана винагорода, $Q(a)=E[r|a]=\theta$. Якщо дія на етапі часу t знаходиться на і-й машині, то $Q(a_t)=\theta_i$.
- R - функція винагороди. У випадку з бандитом Бернуллі ми спостерігаємо винагороду r стохастично. На етапі часу t, $r_t=R(a_t)$ може повернути винагороду 1 з ймовірністю Q(at) або 0 в іншому випадку.

Мета - максимізувати сукупну винагороду $\sum Tt = r_t$.

Оптимальна ймовірність нагороди θ^* оптимальної дії a* (3.1):

$$\theta^* = Q(a^*) = \max_{a \in \mathcal{A}} Q(a) = \max_{1 \le i \le K} \theta_i \quad , \tag{3.1}$$

Функція втрати - це загальна кількість втрат, яке можемо мати, не вибираючи оптимальних дій до етапу часу T (3.2):

$$\mathcal{L}_T = \mathbb{E}\Big[\sum_{t=1}^{T} \big(\theta^* - Q(a_t)\big)\Big], \tag{3.2}$$

3.6.4. ε-жадібний алгоритм

Жадібний алгоритм обирає найкращі дії більшу частину часу, але час від часу проводить випадкові дослідження. Значення дії оцінюється відповідно до минулого досвіду шляхом усереднення винагород, пов'язаних із цільовою дією a, яку ми спостерігали до цього часу (до поточного кроку часу t) (3.3):

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau 1[a_\tau = a] \quad , \tag{3.3}$$

де 1 - функція бінарного індикатора, а Nt(a) - скільки разів дію a було обрано до цього моменту(3.4).

$$N_t(a) = \sum_{\tau=1}^{t} 1[a_\tau = a], \tag{3.4}$$

Відповідно до алгоритму, з невеликою ймовірністю ε ми робимо випадкову дію, але в іншому випадку (що має бути більшою частиною часу, ймовірністю 1-ε) ми вибираємо найкращу дію, про яку ми до цього часу дізналися (3.5).

$$\hat{a}_t^* = \arg\max_{a \in \mathcal{A}} \hat{Q}_t(a), \tag{3.5}$$

3.7. Реалізація алгоритму балансування

Для оцінки оптимальності кожного з доступних сервісів були обрані такі характеристики як завантаження процесора та операційної пам’яті.

Система балансування навантаження може отримувати дані як від окремих служб та засобів моніторингу, так і від самих сервісів.

Був використаний 2 варіант моніторингу для спрощення функціювання системи в цілому і через обмежені ресурси для тестування системи балансування навантаження. А саме, до кожного сервісу був доданий API, через який можна було отримати дані про поточний стан.

Реалізація зі сторони системи балансування навантаження:

```
memImportance := 0.5
cpuImportance := 0.7

var aliveBackends []*Backend

for _, b := range s.backends {
	if b.IsAlive() {
		resp, err := http.Get(target)
		if err != nil {
			continue
	}
	aliveBackends = append(aliveBackends, b)

	var data BackendLoad
	if err = json.NewDecoder(resp.Body).Decode(&data); err != nil
	{
		continue
	}
	resp.Body.Close()

	b.load = memImportance*data.MemoryUsage +
cpuImportance*data.CpuUsage
	}
}
```

Коефіцієнти memImportance та cpuImportance вказують важливість використання пам'яті та процесору відповідно.

Реалізація зі сторони сервісу, де змінні memStats та percents описують завантаження пам’яті та процесору:

```
func (h *handler) Stats(c echo.Context) error {
	memStats, err := mem.VirtualMemory()
	if err != nil {
		return err
	}
	percents, err := cpu.Percent(0, false)
	if err != nil {
		return err
	}


	return c.JSON(http.StatusOK, echo.Map{
		`memoryUsage`: memStats.UsedPercent,
		`cpuUsage`:   percents[0],
	})
}
```

Після отримання необхідних даних та їх оцінки запускається виконання ε-жадібного алгоритму, що обирає найменш завантажений сервер та повертає його клієнту.

```
spines := len(aliveBackends)

var actions []Backend
var history []float64
var best_ind int

j := 0
for i := 0; i < episodes; i++ {
	// epsilon greedy
	p := random.Float64()
	if p < epsilon {
		j = random.Intn(spines)
	} else {
		j = FindMin(aliveBackends)
	}
	n := rand.NormFloat64()
	x := aliveBackends[j].load + rand.Float64()*n
	aliveBackends[j].Update(x)

	actions = append(actions, *aliveBackends[j])
	reward := GetReward(spines, actions)
	if len(history) < 1 {
		best_ind = j
```

```
        } else {
            if reward > history[best_ind] {
                best_ind = j
                history = append(history, reward)
            }
        }
    }
    return aliveBackends[best_ind]
```

Значення episodes та epsilon, тобто кількість кроків та ε, є конфігурованими.

ВИСНОВКИ ДО РОЗДІЛУ 3

В ході виконання даного розділу дипломної роботи, було розглянуто технології для реалізації системи балансування навантаження, а також обрано мову програмування, на якій було розроблено програму, допоміжні пакети та засоби для спрощення й оптимізації роботи коду.

Був проведений аналіз проблематики розробки та обраний алгоритм для її вирішення. Був описаний його метод роботи та написана реалізація можливостями обраної мови програмування.

РОЗДІЛ 4

АНАЛІЗ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Для оцінки працездатності та рентабельності використання розробленої системи балансування навантаження були виконані порівняльні тести за допомогою утиліти hey. Вона дозволяє просто завантажувати сервер кількістю запитів та виводить статистику по отриманим результатам.

Для порівняння результатів та ілюстрації роботи розробленого алгоритму був використаний алгоритм Round Robin, або алгоритм кругового обслуговування.

Були виконані заміри при завантаженні 500, 1000, 2500, 5000, 10000, 50000 запитами (requests) та проаналізовані наступні показники:

- загальний час обробки усіх запитів (рис. 4.1, графік 1)
- найдовший час обробки одного запиту (рис. 4.1, графік 2)
- найкоротший час обробки одного запиту (рис. 4.1, графік 3)
- середній час обробки запитів (рис. 4.1, графік 4)
- середня різниця у використанні процесору та пам’яті між найменш та найбільш навантаженими сервісами (рис. 4.2)
- затримки між відправленням запиту та отримуванням відповіді від серверу (рис .4.3)

Також на основі отриманих результатів були побудовані гістограми для зображення розподілу кількості запитів у часі (рис. 4.4 — рис. 4.10).

Позначення на графіках:

- NN — результати для розробленого алгоритму
- RR — результати для алгоритму Round Robin

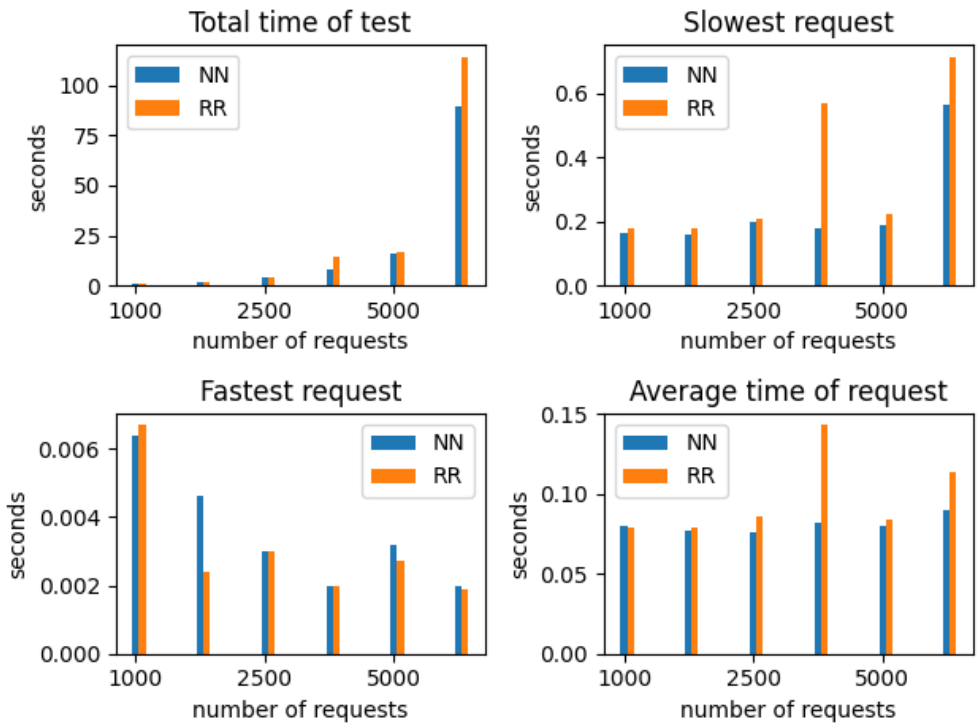


Рис. 4.1. Аналіз часу відповідей системи навантаження на запити:

number of requests — загальна кількість запитів при замірі;

seconds — час обробки в секундах;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

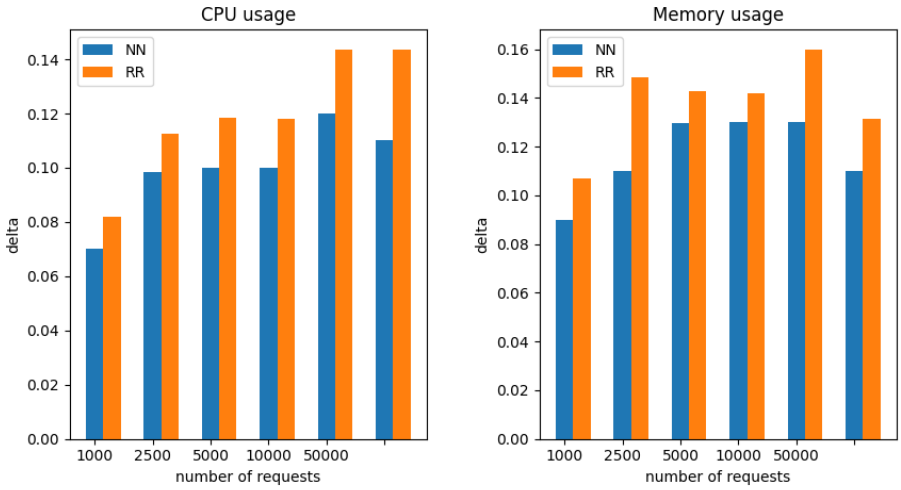


Рис. 4.2. Середня різниця у використанні процесору та пам’яті між найменш та найбільш навантаженими сервісами:

number of requests — загальна кількість запитів при замірі;

delta — відсоткова різниця між найбільш та наменш завантаженим сервісом;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

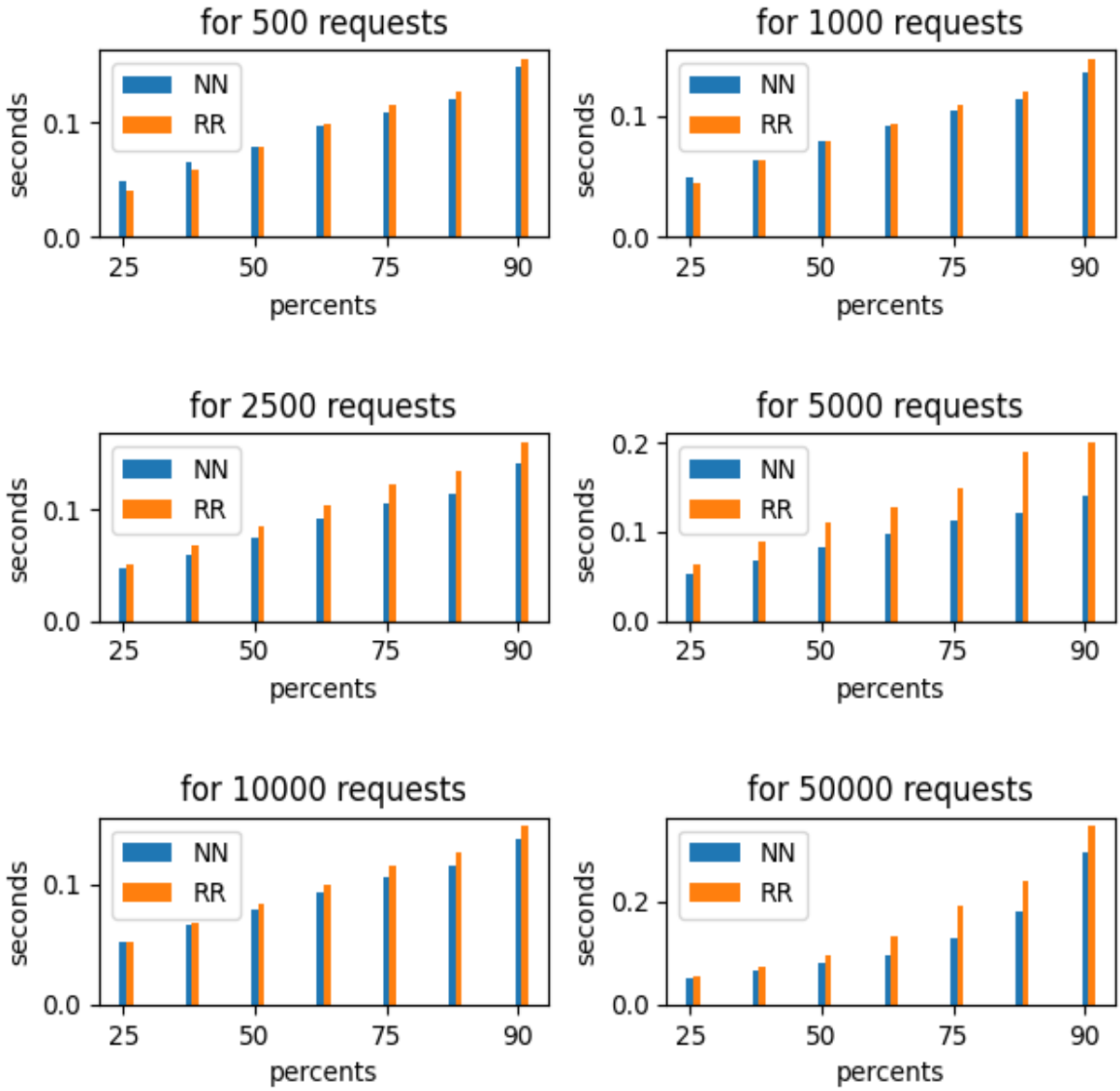


Рис. 4.3. Розподілення затримок між відправленням запиту та отримуванням відповіді від серверу:

for .. requests — загальна кількість запитів при замірі;

seconds — час затримки в секундах;

percents — відсоткова кількість запитів, що підпадають під вказаний час затримки;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

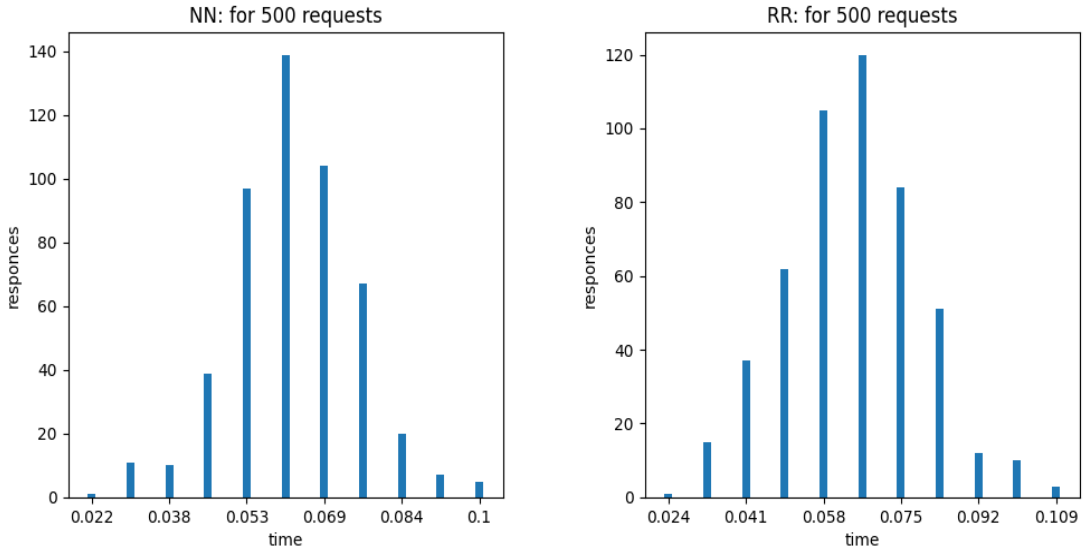


Рис. 4.4. Розподіл кількості запитів у часі для 500 запитів:

for .. requests — загальна кількість запитів при замірі;

responces — кількість відповідей;

time — час реагування серверу на запит та повернення відповіді;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

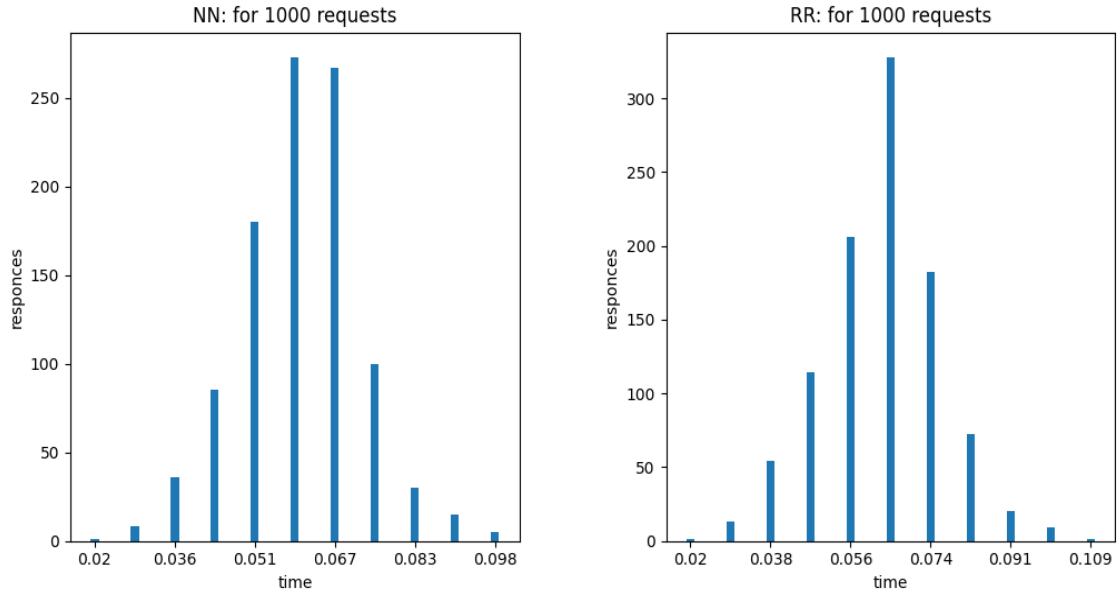


Рис. 4.5. Розподіл кількості запитів у часі для 1000 запитів:

for .. requests — загальна кількість запитів при замірі;

responces — кількість відповідей;

time — час реагування серверу на запит та повернення відповіді;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

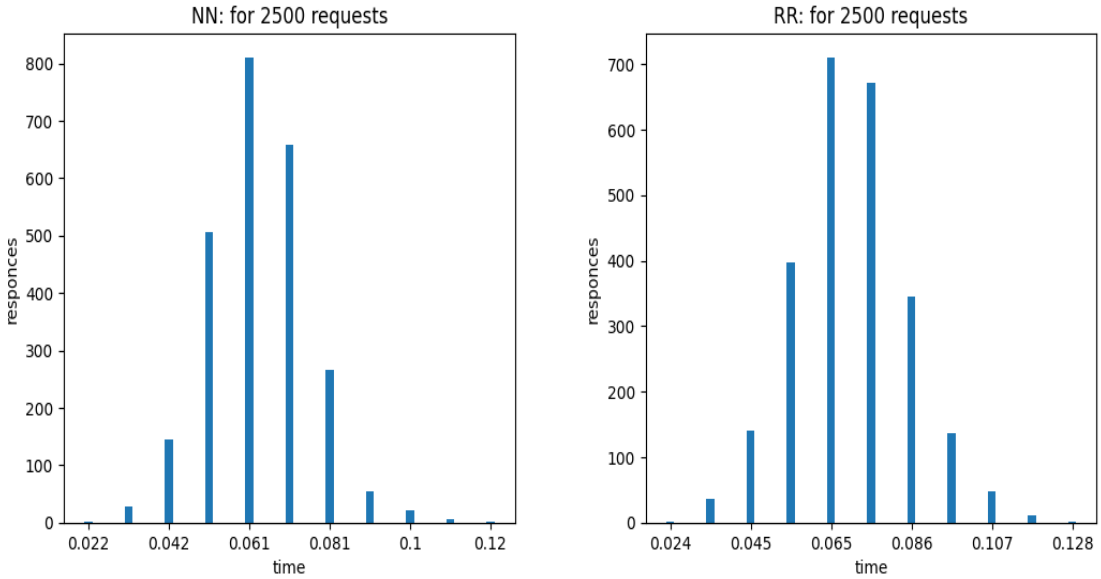


Рис. 4.6. Розподіл кількості запитів у часі для 2500 запитів:

for .. requests — загальна кількість запитів при замірі;

responces — кількість відповідей;

time — час реагування серверу на запит та повернення відповіді;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

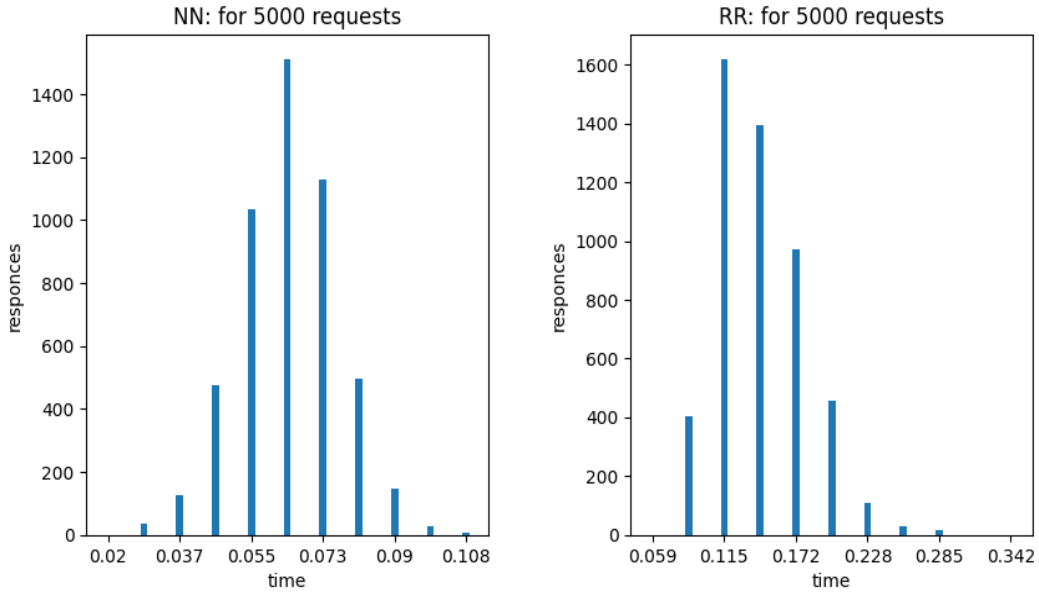


Рис. 4.7. Розподіл кількості запитів у часі для 5000 запитів:

for .. requests — загальна кількість запитів при замірі;

responces — кількість відповідей;

time — час реагування серверу на запит та повернення відповіді;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

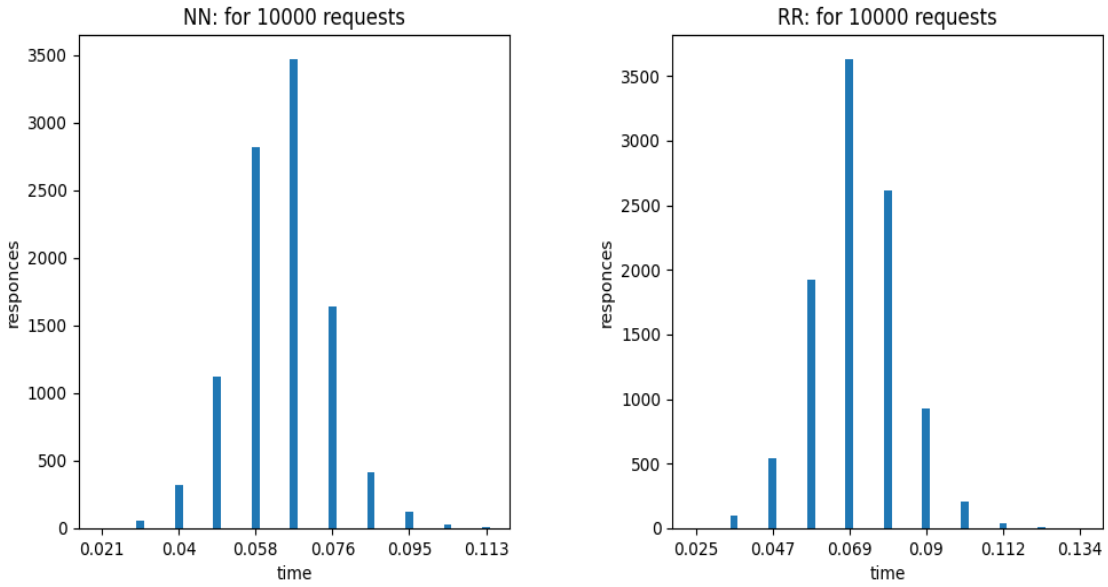


Рис. 4.8. Розподіл кількості запитів у часі для 10000 запитів:

for .. requests — загальна кількість запитів при замірі;

responces — кількість відповідей;

time — час реагування серверу на запит та повернення відповіді;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

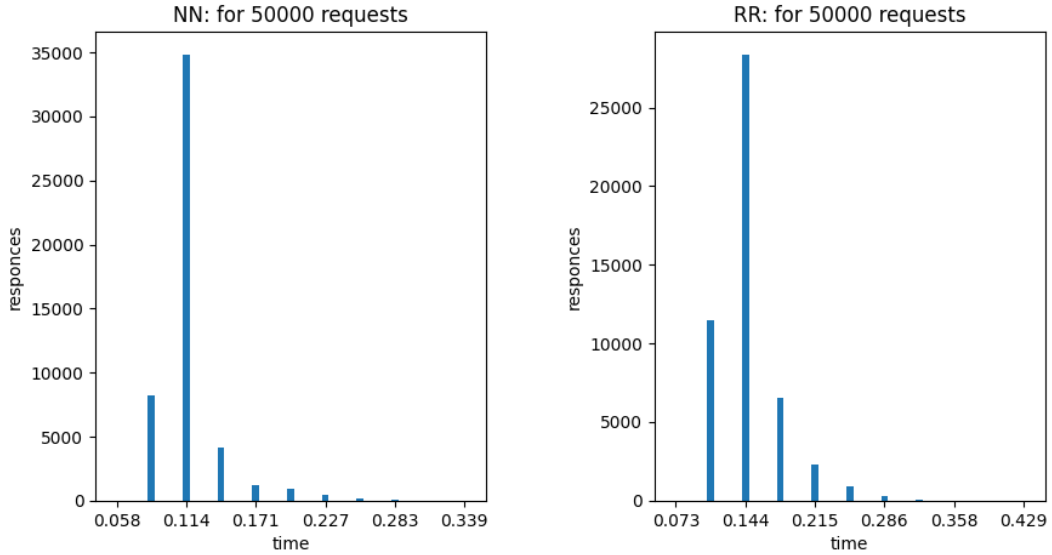


Рис. 4.9. Розподіл кількості запитів у часі для 50000 запитів:

for .. requests — загальна кількість запитів при замірі;

responces — кількість відповідей;

time — час реагування серверу на запит та повернення відповіді;

NN, RR — результати для розробленого алгоритму та Round Robin відповідно.

ВИСНОВКИ ДО РОЗДІЛУ 4

В ході виконання даного розділу дипломної роботи, було проаналізовано роботу реалізованої системи балансування навантаження і порівняно з широко поширеним алгоритмом, а саме алгоритмом кругового обслуговування.

Згідно з результатами, створена система балансування навантаження є більш швидкою у порівнянні із наявними. Виграш у часі відповіді становить ~2.5 рази для ~5000 запитів, що підтверджується кількісним аналізом статистичних розподілів основних параметрів. Виграш у рівномірності навантаження сервісів становить ~1.3 рази. Це загалом дозволяє більш оптимально розподіляти запити по доступним сервісам. Отже, створена система балансування навантаження є більш швидкою та оптимальніше розподіляє запити по доступним сервісам.

Тому, таку систему можна назвати кращим рішенням для вирішення задачі розподілу навантаження порівняно з наявними.

ВИСНОВКИ

Представлений дипломний проект присвячений розробці системі виявлення сервісів з розумним балансуванням навантаження.

В ході виконання роботи були розглянуто, те наскільки подібна система є актуальною відповідно до сучасних потреб. Були розглянуті наявні рішення, загальні характеристики та відомості подібних систем та проведений аналіз їх ефективності та рентабельності. Також був досліджений матеріал про нейронні мережі: їх види та способи створення.

Було надано опис предметної області, визначені основні структури та функції системи, алгоритм роботи та вимоги до продукту.

Основними потребами була швидкодія та можливість рівномірно розподіляти навантаження між сервісами. Саме тому система балансування була визначена як “розумна”, тобто така, що розуміє стан компонентів і системи в цілому та може адаптуватися під поточні умови. Виходячи з цих потреб було створено програмний додаток, що за рахунок використаних технологій гарно виконує поставлену задачу.

Основними інструментами для розробки стали мова програмування Golang яка дозволяє швидко виконувати обробку великої кількості запитів, та програмна платформа Docker, яка робить систему балансування зручною у використанні.

Отримані результати та їх аналіз показали, що дана система балансування є швидкою та у порівнянні з наявним алгоритмом дає виграш у часі відповіді та розподілу навантаження по сервісах, що збільшує ефективність роботи системи.

У більш загальному контексті існує нагальна потреба оцінки та розподілу навантаження серед компонентів складних систем на основі моніторінгу, аналізу та передбачення закономірностей завантаженості компонентів із

великою кількістю характеристик та вимог до цих характеристик. Отже запропонована "розумна" система може бути корисною для більш широкого кола практичних застосувань, як це продемонстровано в даній дипломній роботі на прикладі розподілення навантаження серед обчислювальних одиниць.

ПЕРЕЛІК ПОСИЛАНЬ

1. Rui Xu, Don Wunsch. Clustering — 1st edition — Wiley-IEEE Press, 2008, October 24 — 368 p.
2. HIGH AVAILABILITY FOR CLUSTER OF STATEFUL SERVICES BEHIND LOAD BALANCER IN PRIVATE/PUBLIC CLOUD / Kent Leung, Louis Zhijun Liu, Jeremy Felix, Andrew Ossipov, Mohamed Mahmoud —- IEEE —- 2019, June 19 — 7 p.
3. A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms / Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, Jameela Al-Jaroodi — IEEE — 2012, December 3 — 5p.
4. Б.Г. Айрапетян, РІЗНОВИДИ РОЗПОДІЛЕНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ. АРХІТЕКТУРНІ ОСОБЛИВОСТІ. ПЕРЕВАГИ ТА НЕДОЛІКИ // Системи обробки інформації, - 2013 № 6 — ст. 199 — 203.
5. Tony Bourke, Server Load Balancing — 1st Edition - O'Reilly Media — 2001, August 11 — 208 p.
6. L4 vs L7 Load Balancing [Електронний ресурс] — Режим доступу до ресурсу: https://levelup.gitconnected.com/l4-vs-l7-load-balancing-d2012e271f56
7. Introduction to modern network load balancing and proxying [Електронний ресурс] — Режим доступу до ресурсу: https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236
8. Алєксєєв М.О., Гусак О.В., КЛАСИФІКАЦІЯ АЛГОРИТМІВ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ ДЛЯ ВИКОРИСТАННЯ В РОЗПОДІЛЕНИХ ГЕТЕРОГЕННИХ МЕРЕЖАХ // Міжнародна

Науково-технічна Конференція "ПРОБЛЕМИ ТЕЛЕКОМУНІКАЦІЙ" — 2016, 7 квітня.

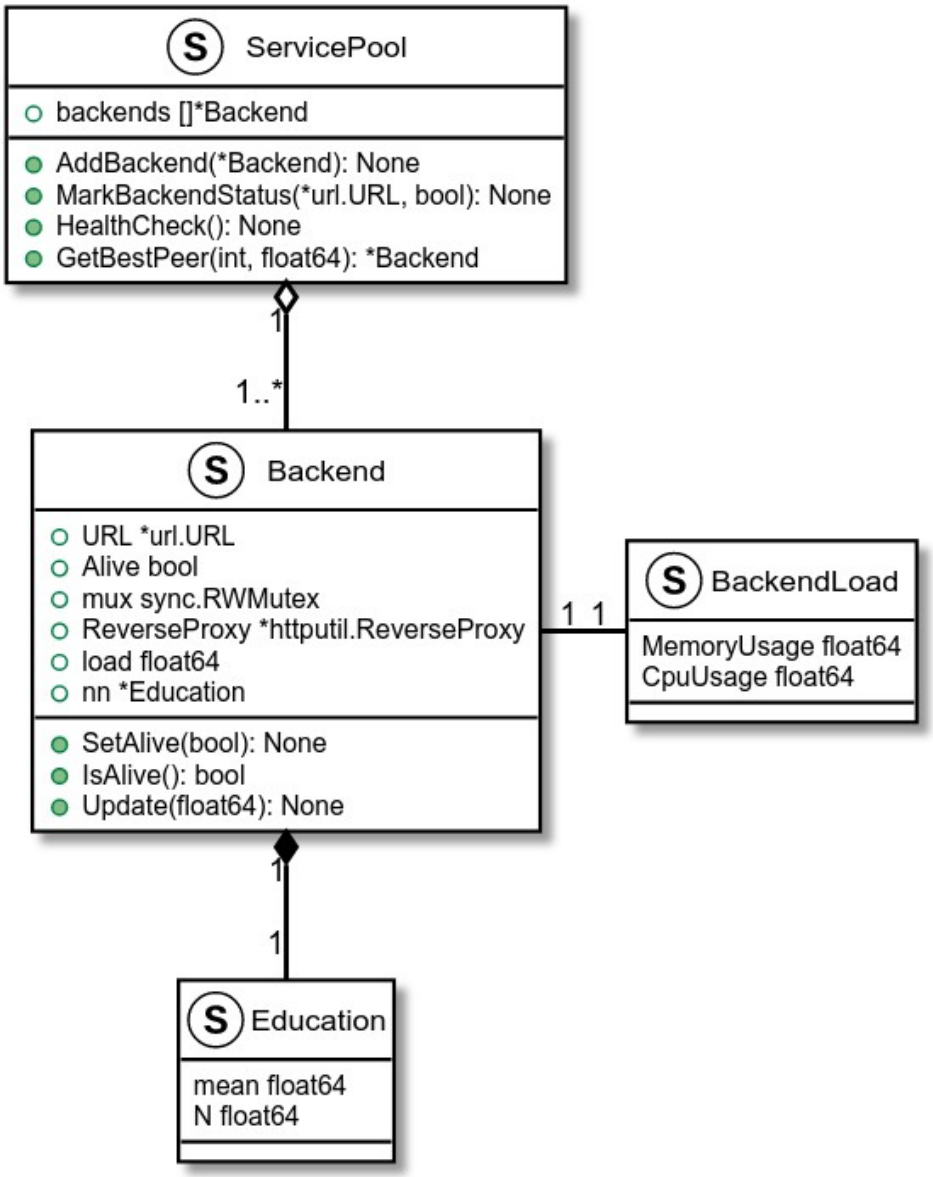
9. Mayanka Katyal, Atul Mishra, A Comparative Study of Load Balancing Algorithms in Cloud Computing Environment // International Journal of Distributed and Cloud Computing, Volume 1 Issue 2 December 2013 - 14p.
10. A Beginner's Guide to Neural Networks and Deep Learning [Електронний ресурс] — Режим доступу до ресурсу: https://pathmind.com/wiki/neural-network
11. Нейронні мережі [Електронний ресурс] — Режим доступу до ресурсу: https://wiki.tntu.edu.ua/%D0%9D%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D1%96_%D0%BC%D0%B5%D1%80%D0%B5%D0%B6%D1%96
12. М.А. Новотарський, Б.Б. Нестеренко, Штучні нейронні мережі: обчислення // Праці Інституту математики НАН України. – Т50. – Київ: Ін-т математики НАН України, 2004 – 408 с.
13. A Beginner's Guide to Deep Reinforcement Learning [Електронний ресурс] — Режим доступу до ресурсу: https://pathmind.com/wiki/deep-reinforcement-learning
14. Alan A. A. Donovan, Brian W. Kernighan, The Go Programming Language — Addison-Wesley Professional — 2015, November 16 — 400p.
15. Karl Matthias, Sean P. Kane, Docker: Up & Running — O'Reilly Media, Inc. — 2015, June 11 — 232p.
16. The Multi-Armed Bandit Problem and Its Solutions [Електронний ресурс] — Режим доступу до ресурсу: https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html

Додаток 1

до дипломного проекту

на тему: «Система виявлення сервісів з розумним контролем навантаження»

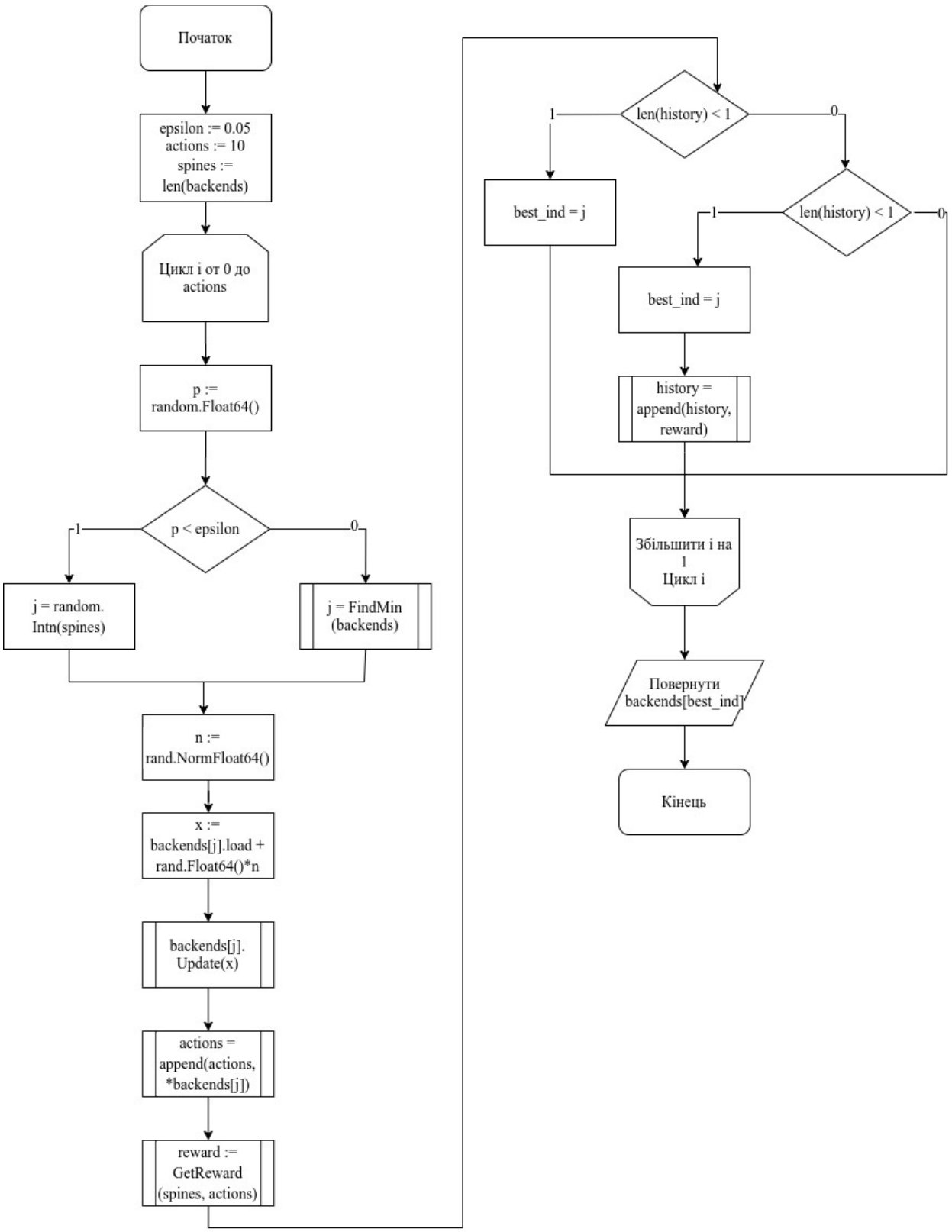
Київ – 2020 року



					ІАЛЦ.468300.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Кузьменко О.В.			Система виявлення сервісів з розумним контролем навантаження Функціональна схема	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Гордієнко Ю.Г.					
Н. контр.		Сімоненко В.П,.				НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-62	
Затверд.		Стіренко С.Г.					

Додаток 2
до дипломного проекту
на тему: «Система виявлення сервісів з розумним контролем навантаження»

Київ – 2020 року



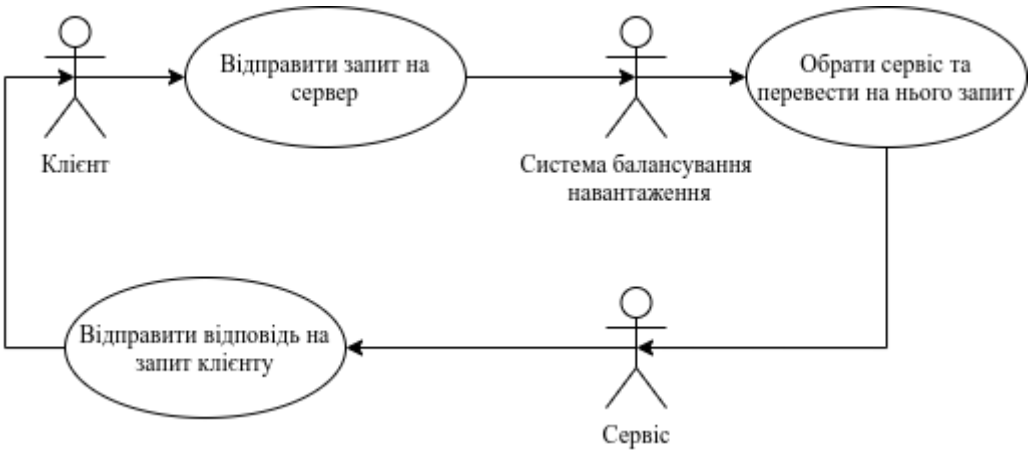
					ІАЛЦ.468300.005 Д2	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Кузьменко О.В.			Система виявлення сервісів з розумним контролем навантаження Принципова схем	Літ. / Аркуш 1 / Аркушів 1
Перевір.		Гордієнко Ю.Г.				
Н. контр.		Сімоненко В.П,.				НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-62
Затверд.		Стіренко С.Г.				

Додаток 3

до дипломного проекту

на тему: «Система виявлення сервісів з розумним контролем навантаження»

Київ – 2020 року



					IIАЛЦ.468300.006 ДЗ		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Кузьменко О.В.			Система виявлення сервісів з розумним контролем навантаження Структурна схема	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Гордієнко Ю.Г.					
Н. контр.		Сімоненко В.П,.				НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-62	
Затверд.		Стіренко С.Г.					

Додаток 4

до дипломного проекту

на тему: «Система виявлення сервісів з розумним контролем навантаження»

Київ – 2020 року

Текст програми

```
package main

import (
        "context"
        "flag"
        "fmt"
        "log"
        "net/http"
        "net/http/httputil"
        "net/url"
        "strings"
        "time"
)
var serverPool ServerPool
func main() {
        var serverList string
        var port int
        //TODO add config file parser
        flag.StringVar(&serverList, "backends", "", "Load balanced backends, use commas to separate")
        flag.IntVar(&port, "port", 3030, "Port to serve")
        flag.Parse()

        if len(serverList) == 0 {
                log.Fatal("Please provide one or more backends to load balance")
        }

        // parse servers
        tokens := strings.Split(serverList, ",")
        for _, tok := range tokens {
                serverUrl, err := url.Parse(tok)
                if err != nil {
                        log.Fatal(err)
                }

                proxy := httputil.NewSingleHostReverseProxy(serverUrl)
                proxy.ErrorHandler = func(writer http.ResponseWriter, request *http.Request, e error) {
```

```
                        log.Printf("[%s] %s\n", serverUrl.Host, e.Error())
                        retries := GetRetryFromContext(request)
                        if retries < 3 {
                                select {
                                case <-time.After(10 * time.Millisecond):
                                        ctx := context.WithValue(request.Context(), Retry, retries+1)
                                        proxy.ServeHTTP(writer, request.WithContext(ctx))
                                }
                                return
                        }
                        // after 3 retries, mark this backend as down
                        serverPool.MarkBackendStatus(serverUrl, false)
                        // if the same request routing for few attempts with different backends,
increase the count
                        attempts := GetAttemptsFromContext(request)
                        log.Printf("%s(%s) Attempting retry %d\n", request.RemoteAddr,
request.URL.Path, attempts)
                        ctx := context.WithValue(request.Context(), Attempts, attempts+1)
                        lb(writer, request.WithContext(ctx))
                }
                serverPool.AddBackend(&Backend{
                        URL:          serverUrl,
                        Alive:        true,
                        ReverseProxy: proxy,
                        load:           0.0,
                        nn:                     &Education{
                                mean: 0.0,
                                N:       0,
                        },
                })
                log.Printf("Configured server: %s\n", serverUrl)
        }
        // create http server
        server := http.Server{
                Addr:    fmt.Sprintf(":%d", port),
                Handler: http.HandlerFunc(lb),
        }
        // start health checking
        go healthCheck()
```

```
        log.Printf("Load Balancer started at :%d\n", port)
        if err := server.ListenAndServe(); err != nil {
                log.Fatal(err)
        }
}
package main
import (
        "encoding/json"
        "fmt"
        "log"
        "net/http"
        "net/url"
        "sync/atomic"
)
// ServerPool holds information about reachable backends
type ServerPool struct {
        backends []*Backend
}
// AddBackend to the server pool
func (s *ServerPool) AddBackend(backend *Backend) {
        s.backends = append(s.backends, backend)
}
// MarkBackendStatus changes a status of a backend
func (s *ServerPool) MarkBackendStatus(backendUrl *url.URL, alive bool) {
        for _, b := range s.backends {
                if b.URL.String() == backendUrl.String() {
                        b.SetAlive(alive)
                        break
                }
        }
}
// HealthCheck pings the backends and update the status
func (s *ServerPool) HealthCheck() {
        for _, b := range s.backends {
                status := "up"
                alive := isBackendAlive(b.URL)
                b.SetAlive(alive)
                if !alive {
```

```
                        status = "down"
                }
                log.Printf("%s [%s]\n", b.URL, status)
        }
}
package main
import "net/http"
const (
        Attempts int = iota
        Retry
)
// GetAttemptsFromContext returns the attempts for request
func GetAttemptsFromContext(r *http.Request) int {
        if attempts, ok := r.Context().Value(Attempts).(int); ok {
                return attempts
        }
        return 1
}
// GetRetryFromContext returns the attempts for request
func GetRetryFromContext(r *http.Request) int {
        if retry, ok := r.Context().Value(Retry).(int); ok {
                return retry
        }
        return 0
}
package main
import (
        "encoding/json"
        "fmt"
        "log"
        "math"
        "math/rand"
        "net/http"
        "time"
)
type BackendLoad struct {
        MemoryUsage float64 `json:"memoryUsage"`
        CpuUsage    float64 `json:"cpuUsage"`
}
```

```
type Education struct {
	mean float64
	N    float64
}
func FindMin(backends []*Backend) (ind int) {
	min := backends[0].nn.mean
	ind = 0
	for i, backend := range backends {
		if backend.nn.mean < min {
			min = backend.nn.mean
			ind = i
		}
	}
	return
}
func (b *Backend) Update(x float64) {
	b.nn.N += 1.0
	b.nn.mean = (1.0-1.0/b.nn.N)*b.nn.mean + 1.0/b.nn.N*x
}
func GetReward(spines int, actions []Backend) (reward float64) {
	size := len(actions)

	mean := 0.0
	for _, a := range actions {
		mean = mean + a.nn.mean
	}
	mean = mean / float64(size)

	sd := 0.0
	for i := 0; i < size; i++ {
		sd += math.Pow(actions[i].nn.mean-mean, 2)
	}
	sd = math.Sqrt(sd / float64(size))

	reward = (mean - sd) / mean
	return
}
```

```
func (s *ServerPool) GetBestPeer(episodes int, epsilon float64) *Backend {
	source := rand.NewSource(time.Now().UnixNano())
	random := rand.New(source)
	memImportance := 0.5
	cpuImportance := 0.7
	var aliveBackends []*Backend
	for _, b := range s.backends {
		if b.IsAlive() {
			target := fmt.Sprintf("%s/stats", b.URL)
			resp, err := http.Get(target)
			if err != nil {
				log.Printf("Failed to get info from %s, error: [%s]\n", b.URL.Path,
err)
				continue
			}
			aliveBackends = append(aliveBackends, b)
			var data BackendLoad
			if err = json.NewDecoder(resp.Body).Decode(&data); err != nil {
				continue
			}
			resp.Body.Close()
			log.Printf("Got data {%#v} from %s\n", data, b.URL)
			b.load = memImportance*data.MemoryUsage + cpuImportance*data.CpuUsage
		}
	}

	spines := len(aliveBackends)

	var actions []Backend
	var history []float64
	var best_ind int

	j := 0
	for i := 0; i < episodes; i++ {
		// epsilon greedy
		p := random.Float64()
		if p < epsilon {
			j = random.Intn(spines)
		} else {
```

```
                        j = FindMin(aliveBackends)
                        //j = np.argmax([a.mean for a in actions])
                }
                n := rand.NormFloat64()
                x := aliveBackends[j].load + rand.Float64()*n
                aliveBackends[j].Update(x)

                actions = append(actions, *aliveBackends[j])
                reward := GetReward(spines, actions)
                if len(history) < 1 {
                        best_ind = j
                } else {
                        if reward > history[best_ind] {
                                best_ind = j
                                history = append(history, reward)
                        }
                }
        }
        log.Printf("Have chosen %s\n", aliveBackends[best_ind].URL)
        return aliveBackends[best_ind]
}
package main

import (
        "net/http/httputil"
        "net/url"
        "sync"
)

// Backend holds the data about a server
type Backend struct {
        URL          *url.URL
        Alive        bool
        mux          sync.RWMutex
        ReverseProxy *httputil.ReverseProxy
        load         float64
        nn           *Education
}
```

```
// SetAlive for this backend
func (b *Backend) SetAlive(alive bool) {
        b.mux.Lock()
        b.Alive = alive
        b.mux.Unlock()
}

// IsAlive returns true when backend is alive
func (b *Backend) IsAlive() (alive bool) {
        b.mux.RLock()
        alive = b.Alive
        b.mux.RUnlock()
        return
}
package main
import (
        "log"
        "net"
        "net/url"
        "time"
)

// isAlive checks whether a backend is Alive by establishing a TCP connection
func isBackendAlive(u *url.URL) bool {
        timeout := 2 * time.Second
        conn, err := net.DialTimeout("tcp", u.Host, timeout)
        if err != nil {
                log.Println("Site unreachable, error: ", err)
                return false
        }
        _ = conn.Close()
        return true
}

// healthCheck runs a routine for check status of the backends every 2 mins
func healthCheck() {
        t := time.NewTicker(time.Minute * 2)
        for {
                select {
```

```
                case <-t.C:
                        log.Println("Starting health check...")
                        serverPool.HealthCheck()
                        log.Println("Health check completed")
                }
        }
}
package main
import (
        "log"
        "net/http"
)

// lb load balances the incoming request
func lb(w http.ResponseWriter, r *http.Request) {
        attempts := GetAttemptsFromContext(r)
        if attempts > 3 {
                log.Printf("%s(%s) Max attempts reached, terminating\n", r.RemoteAddr, r.URL.Path)
                http.Error(w, "Service not available", http.StatusServiceUnavailable)
                return
        }

        peer := serverPool.GetBestPeer(10, 0.05)
        if peer != nil {
                peer.ReverseProxy.ServeHTTP(w, r)
                return
        }
        http.Error(w, "Service not available", http.StatusServiceUnavailable)
}
```