Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій

УДК 004.75;004.042 *На правах рукопису*

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема: “ Розподілена система обробки потоків реального часу засобами ML із балансуванням навантаження ”

Спеціальність – 121 “Інженерія програмного забезпечення”

ПОЯСНЮВАЛЬНА ЗАПИСКА

БР.ІПЗ – 30.00.00.000

Студент
ІПЗ-43 ____________________ /Сергій ВАЩЕНКО/

Науковий керівник
асист. _________________ /Кирило КАДОМСЬКИЙ/

Консультант
з питань нормоконтролю
фахівець ________________ /Тамара ЧАПОВСЬКА/

Допускається до захисту
Завідувач кафедри
д.т.н., проф. _________________ /Олексій БИЧКОВ/

Київ – 2021

Рішенням Екзаменаційної комісії
випускна кваліфікаційна робота студента

____

захищена з оцінкою

____

Голова Екзаменаційної комісії

професор, доктор техн. наук Андрій БОНДАРЧУК

Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій
Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ:
Завідувач кафедри програмних систем і технологій
__________/Олексій БИЧКОВ/
(підпис) (розшифровка підпису)

„___” ________________2021р.

ЗАВДАННЯ
НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ
СТУДЕНТУ

Ващенку Сергію Олександровичу
(прізвище, ім’я, по-батькові)

1. Тема випускної кваліфікаційної бакалаврської роботи “ Розподілена система обробки потоків реального часу засобами ML із балансуванням навантаження ”______

керівник проекту (роботи) Кадомський Крило Костянтинович асистент_______

затверджені наказом вищого навчального закладу від „___” _____ 20__ р. № _____

2. Строк здачі студентом закінченої роботи _____________________________________

3. Вихідні дані до роботи Створена розподілена система обробки потоків реального часу засобами ML із балансуванням навантаження.

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)

1. Огляд предметної сфери та існуючих рішень. Мета роботи;
2. Архітектура системи та вибір основних компонентів;
3. Програмна реалізація систем;

5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень)

1. Діаграма прецедентів, що відображає взаємодію із загальною функціональністю платформи (Рис. 1.2.1, ст.16);
2. Схема роботи методу K-Means (Рис. 1.2.1, ст.19);
3. Діаграма класів головного сервера (Рис. 2.2.1.1, ст.31);
4. Діаграма класів обчислювального сервера (Рис. 2.2.1.2, ст.32);
5. Діаграма діяльності (Рис. 2.2.2.1, ст.33);
6. Діаграма розгортання (Рис. 2.2.3.1, ст.34);
7. Процес обробки запитів за допомогою DispatcherServlet (Рис. 2.3.3.1, ст.38);
8. Балансування (Рис. 2.5.1, ст.50);

9. Черга повідомлень (Рис. 2.6.1, ст.51);

10. Блок схема підключення клієнтом до пристрою (Рис. 2.6.1, ст.52);

11. Блок схема отримання даних від пристрою (Рис. 3.1.2.1, ст.54);

12. Блок схема видачі результатів клієнту (Рис. 3.1.3.1, ст.55);

13. Блок схема обробки даних з черги (Рис. 3.1.4.1, ст.56);

14.Головний інтерфейс (Рис. 3.3.1, ст.58);

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання ______________

Керівник _______ /Кирило КАДОМСЬКИЙ/
(підпис) *(розшифровка підпису)*

Завдання прийняв до виконання ______ /Сергій ВАЩЕНКО/
(підпис) *(розшифровка підпису)*

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Формування проблематики	28.12.2020	
2	Аналіз проблематики	15.01.2021	
3	Постановка задачі	18.02.2021	
4	Створення алгоритмів та план дій реалізації	20.02.2021	
5	Проектування архітектури системи	28.03.2021	
6	Розробка ПЗ для апробації архітектури	01.04.2021	
7	Тестування ПЗ та покращення архітектури	15.04.2021	
8	Підготовка документації	18.04.2021	

9	Попередній захист		
10	Нормоконтроль		
11	Перевірка на співпадіння		
12	Захист		

Студент – бакалавр ___________ /Сергій ВАЩЕНКО/
(підпис) *(розшифровка підпису)*

Керівник роботи ___________ /Кирило КАДОМСЬКИЙ/
(підпис) *(розшифровка підпису)*

АНОТАЦІЯ

Випускна кваліфікаційна бакалаврська робота: 69 сторінки, 14 рисунків, 1 додатків, 9 джерела.

Тема: Розподілена система обробки потоків реального часу засобами ML із балансуванням навантаження.

Об’єкт дослідження: Архітектура системи з розподіленою обробкою потоків реального часу.

Мета роботи: Дослідити та розробити систему розподіленої обробки потоків даних IoT моделями ML в реальному часі із балансуванням навантаження.

Предмет дослідження: Архітектура системи та технології розподілення обробки потоків даних в реальному часі із балансуванням навантаження.

Результати дослідження: Досліджено архітектуру та реалізовано систему розподіленої обробки потоків даних в реальному часі із балансуванням навантаження. Систему застосовано для розподілених розрахунків і використанням алгоритму K-Mean.

Висновок: В результаті роботи було досліджено та впроваджено систему розподіленої обробки потоків даних у режимі реального часу з балансуванням навантаження. Розроблено архітектуру пристосовану до високонавантажених розрахунків в реальному часі. Запропоновано метод розподілення алгоритму K-Mean. Розроблено систему та впроваджено у використання для розрахунків.

АННОТАЦИЯ

Выпускная квалификационная бакалаврская работа: 69 страницы, 14 рисунков, 1 дополнений, 9 источника.

Тема: Распределенная система обработки потоков реального времени средствами ML с балансировкой нагрузки.

Объект исследования: Архитектура системы с распределенной обработкой потоков реального времени.

Цель работы: исследовать и разработать систему распределенной обработки потоков данных IoT моделями ML в реальном времени с балансировкой нагрузки.

Предмет исследования: Архитектура системы и технологии распределения обработки потоков данных в реальном времени с балансировкой нагрузки.

Результаты исследования: Исследована архитектуру и реализована система распределенной обработки потоков данных в реальном времени с балансировкой нагрузки. Система применена для распределенных расчетов и использованием алгоритма K-Mean.

Вывод: В результате работы были исследованы и внедрена система распределенной обработки потоков данных в режиме реального времени с балансировкой нагрузки. Разработана архитектура приспособленную к высоконагруженных расчетов в реальном времени. Предложен метод распределения алгоритма K-Mean. Разработана система и внедрены в использование для расчетов.

ANNOTATION

Final qualifying bachelor's thesis: 69 pages, 14 figures., 1 addition, 9 sources.

Topic: Distributed system for real-time ML stream processing with load balancing.

Object of research: Real-time stream processing system architecture.

Purpose: Investigate and develop a system of distributed processing of IoT data streams by ML models in real time with load balancing.

Subject of research: Real-time system architecture and distribution technology for data flow processing with load balancing.

Research results: The architecture and the system of distributed processing of real-time data streams with load balancing are investigated. The system is used for distributed calculations and using the K-Mean algorithm.

Conclusion: As a result, the system of distributed processing of data streams in real time with load balancing was investigated and implemented. An architecture adapted to high-load real-time calculations has been developed. The method of distribution of K-Mean algorithm is offered. The system is developed and put into use for calculations.

ЗМІСТ

ВСТУП 10

РОЗДІЛ 1 13

ОГЛЯД ПРЕДМЕТНОЇ СФЕРИ ТА ІСНУЮЧИХ РІШЕНЬ. МЕТА РОБОТИ 13

1.1 Предметна область і актуальність роботи 13

1.2 Мета роботи. Функціональні та нефункціональні вимоги до системи 15

1.3 Алгоритми потокової обробки даних IoT 18

1.4 Підходи до побудови архітектури системи потокової обробки даних IoT 20

1.5 Існуючі рішення і шляхи їх покращення 23

РОЗДІЛ 2 28

АРХІТЕКТУРА СИСТЕМИ ТА ВИБІР ОСНОВНИХ КОМПОНЕНТІВ 28

2.1 Аналіз вимог до архітектури 28

2.2 Архітектура системи 31

2.2.1 Діаграма класів 31

2.2.2 Діаграма діяльності 32

2.2.3 Діаграма розгортання 34

2.3 Вибір засобів реалізації основних компонентів 35

2.3.1 Фреймворк Spring 35

2.3.2 Фреймворк Spring Boot 36

2.3.3 Фреймворк Spring Web MVC 37

2.3.4 Технологія Maven 38

2.3.5 База даних Redis 42

2.3.6 База даних MondoDB 42

2.4 Протоколи і формати обміну даними між компонентами системи 45

2.5 Масштабування системи 47

2.6 Балансування навантаження 51

РОЗДІЛ 3 53

ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ. 53

3.1 Алгоритм роботи кожного компонента системи 53

3.1.1 Підключення клієнтом до пристрою 53

3.1.2 Отримання даних від пристрою 54
3.1.3 Видача результатів клієнту 55
3.1.4 Обробка даних з черги 55
3.2 API 57
3.2.1 API симулятора пристрою 57
3.2.2 REST API головного сервера 57
3.3 Оцінювання роботи системи 57
3.4 Практичне застосування результатів 58
ВИСНОВКИ 60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 62
ДОДАТКИ 63

ВСТУП

Актуальність роботи

Інтернет речі - одна з найбільш перспективних сфер розвитку інформаційно-комунікаційних технологій. Кількість пристроїв, підключених до Інтернету, швидко зростає і вимагає сучасних підходів до створення серйозних серверних систем. Ці платформи IoT призначені для аналізу різних аспектів даних , необхідних для оптимізації різних процесів виробництва та інших.

Проблеми та завдання в області налаштування серверних систем для Інтернету речей можна розділити на загальні та конкретні проблеми, які притаманні багатьом іншим великим системам обробки даних і які виникають лише тут. Найпоширенішими проблемами є створення та використання ефективних та відмовостійких рішень масштабованих та розподілених систем даних. Проблемами, характерними для Інтернету речей, є надійне управління пристроями та моніторинг. Цей документ визначає вимоги до створення хмарних платформ для Інтернету речей, досліджує існуючі платформи та показує, як їх можна вдосконалити. Обговорюються та описуються високоархітектурні рішення для обробки даних та підходів до розробки програмного забезпечення, що дозволяють реалізувати платформу, яка відповідає цим функціональним та нефункціональним вимогам. Розглянуто можливості сторонніх рішень, за допомогою яких можна створити платформу. Вибір протоколів та форматів даних, щоб гарантувати найкращі параметри системи, є виправданим

Зв'язок роботи з науковими програмами, планами, темами

Розробка програмних рішень для управління великою кількістю пристроїв вимагає застосування новітніх підходів до хмарних обчислень, обробки та зберігання великих даних. Ці серверні рішення повинні масштабуватися і мати високу пропускну здатність.

Мета і задачі дослідження

Метою дипломної роботи є розробка серверних рішень, що забезпечують Інтернет речей. Зокрема, розглядається структура таких підсистем:

- Прийом та зберігання даних;
- Обробка даних. Виконує обробку потоків та аналітичні запити. Передача даних відбувається між сховищем даних на постійній умовній основі та базою даних (постійним сховищем даних).

Об’єктом дослідження э приклади систем розподіленої обробки потоків даних в реальному часі із балансуванням навантаженням

Предметом дослідження э архітектура систему розподіленої обробки потоків даних в реальному часі із балансуванням навантаженням.

Новизна одержаних результатів

Запропоновані та застосовані архітектурно-технічні рішення є дуже універсальними та дозволяють будувати широкий спектр систем обробки великих даних та інших платформ для бірюзового Інтернету, що спеціалізуються на вужчих областях.

Практичне значення одержаних результатів.

Запропонована та впроваджена така архітектура розподіленої сервісно-орієнтованої (мікросервіси) платформи, що пропонує високу продуктивність та майже необмежену масштабованість. Таким чином, ви можете збільшити потужність системи для обслуговування мереж пристроїв будь-якого розміру та можливості підсистеми аналізу даних. Ефективно впроваджуючи компоненти системи, ви можете оптимізувати вартість обладнання платформи. Результати виконаної дипломної роботи впроваджено на підприємстві ТОВ “МІТ”.

Особистий внесок студента.

Основний результат:

1. Сервісно-орієнтована архітектура розподіленої платформи (мікросервіс), яка пропонує високу продуктивність та майже необмежену масштабованість;
2. Ефективна реалізація компонентів системи може оптимізувати вартість обладнання платформи.

Публікації

Під час роботи над бакалаврською роботою було підготовлено тези для восьмої Східно-Європейської конференції на тему «Розподілена система обробки потоків реального часу із балансуванням навантаження», що надрукована у збірнику узагальнених матеріалів восьмої Східно-Європейської конференції в секції «Математичні та програмні технології Internet of Everything» (14.05.2021, Київ).

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ СФЕРИ ТА ІСНУЮЧИХ РІШЕНЬ. МЕТА РОБОТИ

1.1 Предметна область і актуальність роботи

Інтернет речей (IoT) складається з взаємодії між собою або із зовнішнім середовищем, що є концепцією комп'ютерної мережі фізичних об'єктів («речей»), які оснащені інтегрованими технологіями, що організовують такі мережі як явище в вважається розвитком . Курс реструктуризації. він може рухати економічні та соціальні процеси та усувати необхідність участі людини у діях та операціях.

Індустрія Інтернету речей протягом останніх років стабільно зростала, і її зростання буде лише прискорюватися. Дослідницька та консалтингова компанія Gartner передбачає у своєму звіті [1], що у 2016 році у світі буде 6,4 мільярда "речей", що на 30% більше, ніж у 2015 році (4,9 мільярда пристроїв). У грошовому вираженні звіт припускає, що галузь збільшиться в розмірі з 1,183 млрд доларів у 2015 році до 1,414 млрд доларів у 2016 році . До 2020 року буде підключено 20,7 мільярда підключених пристроїв та їх загальна вартість. це становитиме 3 трильйони доларів. близько половини - споживачі, інша половина - підприємства). Cisco, найбільший у світі виробник мережевих пристроїв [2], ще більш оптимістично оцінює розмір ринку IoT: 14,4 трлн. Доларів до 2022 року.

Інтернет речей проникає у всі сфери життя і забезпечує роботу багатьох систем: чисті споживчі датчики (наприклад, різні побутові датчики, портативні електронні пристрої, розумні будинки тощо), системи розумних міст, автономні машини тощо тощо. За допомогою "речей" додана вартість [2] збільшується за рахунок:

- Поліпшити взаємодію з клієнтами;
- Скоротити час між плануванням товару та його комерціалізацією (час виходу на ринок);
- Вдосконалення ланцюгів поставок та логістики;
- Збільшення продуктивності праці працівників;

- Більш ефективне використання активів (зменшення витрат).

Розвиток Інтернету речей просунувся у багатьох галузях техніки. Зростаючий попит на пристрої ВГД стимулює чітко на інновації в галузі електроніки, інтернет - індустрії заперечували електронні дошки, датчики, батареї і т.д

Специфічні характеристики портативних пристроїв ставлять певні обмеження щодо програмного забезпечення, яке ними управляє. Обмежена обчислювальна потужність компактних карток та вимоги до енергоспоживання (особливо для пристроїв, що живляться від акумуляторів) вимагають економії ресурсів та переносимості майже всієї обробки даних на сервері

Серверні платформи, які обробляють показання датчиків, також повинні враховувати, що дані неминуче містять помилки і що пристрої час від часу можуть виходити з ладу та виходити з ладу.

Розробка програмних рішень, що управляють великою кількістю пристроїв, вимагає застосування новітніх підходів у галузі хмарних обчислень, обробки та зберігання великих даних. Ці серверні рішення повинні масштабуватися, мати високу пропускну здатність, відмовостійкість і короткий час відгуку

Для ефективного використання даних у галузі Інтернету речей особливо важлива наявність різних інструментів для їх аналізу. В даний час більшість пристроїв носять суто інформаційний характер, тобто ті, які збирають і передають лише дані і, наприклад, не виконують жодної функції управління. Тому інтелектуальна обробка даних є центральною вимогою цих систем

Інформація, зібрана з пристроїв IoT, наприклад, може бути використана для оптимізації певних виробничих процесів, контролю якості продукції або моніторингу роботи інших систем. Це, як правило, проходить через формальний процес обробки, такі як: Б. агрегація, а потім розглянуто фахівцями. Розвиток засобів обробки даних, які використовують техніки машинного навчання, може зменшити або усунути необхідність залучати людей. За допомогою ефективних автоматичних механізмів

прийняття рішень можна буде збільшити частку пристроїв, які самостійно виконують функції управління певними процесами.

Слід також зазначити, що методи та засоби, що використовуються в серверних рішеннях для Інтернету речей, є загальноприйнятими, а також можуть використовуватися в інших областях, які збирають та обробляють великі обсяги даних. Це такі сфери: аналіз соціальних мереж, аналіз поведінки користувача веб-сайту, аналіз стану інфраструктури дата-центру, реферальні системи, націлювання оголошень, пошукова мережа та інші

1.2 Мета роботи. Функціональні та нефункціональні вимоги до системи

У цій роботі розглядається структура серверних рішень, що забезпечують Інтернет речей. Зокрема, розглядається структура таких підсистем:

- Прийом та зберігання даних. Він отримує його зі швидкістю прийому даних (тобто з мінімальними затримками) і зберігає в умовно постійному сховищі даних (черга повідомлень);
- Обробка даних. Виконує обробку потоків та аналітичні запити. Передача даних здійснюється між умовно постійним сховищем даних та базою даних (постійним сховищем даних). Вироджений випадок обробки даних, якщо він не потрібен: проста копія інформації з черги повідомлень до бази даних.

Показником успіху проекту є система, що працює, яка:

- вирішує основні проблеми з цими платформами (зберігання, аналіз даних тощо), тобто реалізує вищезазначені підсистеми з урахуванням загальновизнаних підходів до розробки систем розподіленої обробки великих даних [3];
- Розроблено з урахуванням найкращих практик та підходів програмної архітектури [3], що дозволяють розміщувати в системі функціональні та нефункціональні вимоги;

- має специфічні функції, яких немає в інших системах (наприклад, надійна доставка повідомлень між пристроями та адміністраторами та засоби контролю пристроїв);
- включає інструменти для розгортання хмарного хостингу, а також інструменти для управління групою серверів. З урахуванням цих цілей ми опишемо вищезазначене (use case) системи, що розробляється.

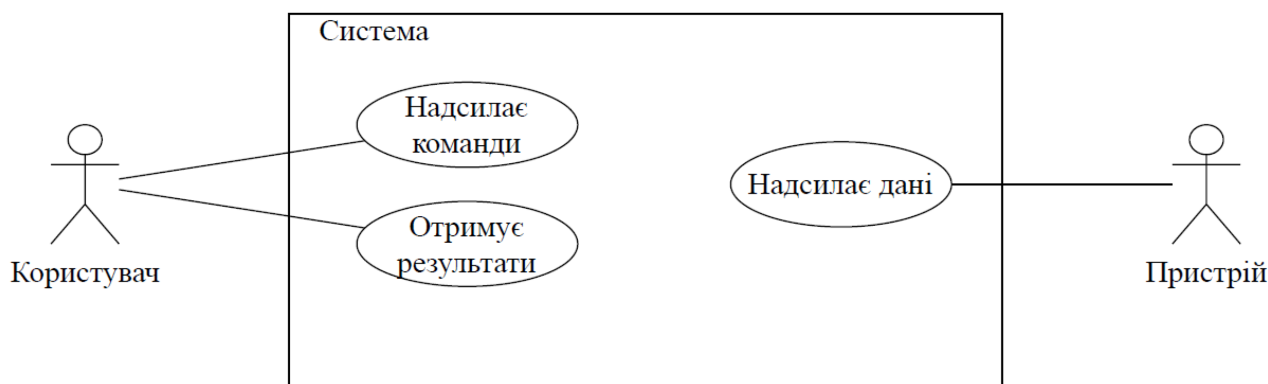

Рис. 1.2.1 Діаграма прецедентів, що відображає взаємодію із загальною функціональністю платформи

Наприклад, пристрій може мати підключення до Інтернету для дротових через Ethernet або бездротову точку доступу (Wi-Fi), що дозволяє на доступ до мережі . Якщо поблизу або в будівлі є кілька пристроїв, вони часто підключаються безпосередньо до шлюзу (кількість шлюзів набагато менше кількості датчиків), який, у свою чергу, передає дані на сервер через Інтернет. Це відбувається з кількох причин:

- економічні - наявність складних мережевих пристроїв збільшує вартість пристроїв;
- Споживана потужність - Виділені протоколи передачі даних, такі як Bluetooth Low Energy, дозволяють більш ефективно використовувати потужності, що особливо важливо, коли пристрій живиться від з окремим джерелом живлення;
- Програмне забезпечення: пристрої можуть передавати дані на шлюз в будь-якому довічним форматі, який легко спроектувати і зводить

до мінімуму раз, коли центральний процесор. Шлюз, у свою чергу, перетворює дані в загальний формат (наприклад, JSON або Protobuf) і передає їх на сервер. Мережу пристроїв на шлюзі також можна вважати захищеною і передавати відкриті дані, а також шифрувати та передавати в мережу за допомогою захищеного протоколу (наприклад, SSL).

Отже, як тільки дані надходять на сервер, вони попередньо обробляються та зберігаються. Ці дані доступні для подальшого аналізу.

Ця робота про створення серверної платформи. Конкретні проблеми розробки програмного забезпечення для пристроїв та шлюзів (firmware) виходять за рамки цього проекту: описані лише основні проблеми реалізації для клієнтської частини платформи. Також важливо сформулювати нефункціональні вимоги до платформи:

- Масштабованість Збільшення обчислювальних ресурсів повинно відповідно збільшити продуктивність системи (залежності повинні бути якомога ближчими до лінійності). Крім того, масштабованість повинні бути горизонтальними для потреб систем з Big Data і забезпечити хмара , щоб зустрітися , тобто підвищити продуктивність системи за рахунок кількості вузлів в мережі буде збільшити . Хмарний центр обробки даних. Горизонтальна шкала також набагато дешевша, ніж вертикальна;
- Відмовостійкість. Існує потреба забезпечити таку архітектуру системи, яка містить резервні компоненти (компоненти реплікації), а також може реагувати на помилки та перевантаження під час роботи без повного вимкнення служби. Діагностика толерантність є особливо важливою для тих частин системи, прийом і зберігання даних. Якщо сервер недоступний, пристрої просто не можуть довго зберігати свої дані через обмежену пам’ять, а потім передавати їх на сервер.

Продуктивність. Впровадження системи повинно бути максимально ефективним, щоб забезпечити пропускну здатність сотень тисяч запитів в секунду і більше для підсистеми прийому та обробки інформації. Запити на аналітику також повинні бути оброблені за відносно короткий час (не більше кількох хвилин). .

1.3 Алгоритми потокової обробки даних IoT

Кластеризація методом k-сере́дніх — популярний метод кластеризації, — впорядкування множини об'єктів в порівняно однорідні групи. Винайдений в 1950-х роках математиком ГуґоШтайнгаузом і майже одночасно Стюартом Ллойдом. Особливу популярність отримав після виходу роботи МакКвіна.

Мета методу — розділити n спостережень на k кластерів, так щоб кожне спостереження належало до кластера з найближчим до нього середнім значенням. Метод базується на мінімізації суми квадратів відстаней між кожним спостереженням та центром його кластера, тобто функції (1.3.1)

$$\sum_{t=1}^{T} d(x_i, m_j(x_i))^2 \tag{1.3.1}$$

де d — метрика, x_i —i-ий об'єкт даних, а $(m_j(x_i))$ — центр кластера, якому на j-ій ітерації приписаний елемент xi. Термін «k-середніх» був уперше вжитий Джеймсом МакКвіном (англ. James MacQueen) у 1967 році, хоча ідею методу вперше озвучив Гуґо Штайнгауз (англ. Hugo Steinhaus) у 1957 році[5]. Стандартний алгоритм був вперше запропонований Стюартом Лойдом (англ. Stuart Lloyd) у 1957 р.

Маємо масив спостережень (об'єктів), кожен з яких має певні значення по ряду ознак. Відповідно до цих значень об'єкт розташовується у багатовимірному просторі.

Дослідник визначає кількість кластерів, що необхідно утворити. Випадковим чином обирається k спостережень, які на цьому кроці вважаються 44 центрами кластерів. Кожне спостереження «приписується» до одного з n кластерів — того, відстань до якого найкоротша.

Розраховується новий центр кожного кластера як елемент, ознаки якого розраховуються як середнє арифметичне ознак об'єктів, що входять у цей кластер

Відбувається така кількість ітерацій (повторюються кроки 3-4), поки кластерні центри стануть стійкими (тобто при кожній ітерації в кожному кластері опинятимуться одні й ті самі об'єкти), дисперсія всередині кластера буде мінімізована, а між кластерами — максимізована.

Вибір кількості кластерів відбувається на основі дослідницької гіпотези. Якщо її немає, то рекомендують створити 2 кластери, далі 3,4,5, порівнюючи отримані результати (Рис. 1.3.1) [4]

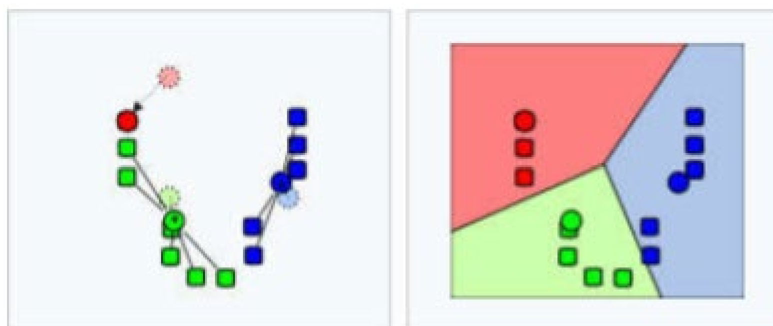

Рис. 1.3.1 Схема роботи методу

Принцип алгоритму полягає в пошуку таких центрів кластерів та наборів елементів кожного кластера при наявності деякої функції Ф(°), що виражає якість поточного розбиття множини на k кластерів, коли сумарне квадратичне відхилення елементів кластерів від центрів цих кластерів буде найменшим (1.3.2)

$$V = \sum_{i=1}^{k} \sum_{x_j \in S_i} (x_j - \mu_i)^2 \tag{1.3.2}$$

де k — число кластерів, S_i — отримані кластери, = 1,2, … ,k , μ_i — центри мас векторів є $x_j \in S_i$.

В початковий момент роботи алгоритму довільним чином обираються центри кластерів, далі для кожного елемента множини ітеративно обраховується відстань від центрів з приєднанням кожного елемента до кластера з найближчим центром. Для кожного з отриманих кластерів обчислюються нові значення центрів, намагаючись при цьому мінімізувати функцію Ф(°), після чого повторюється процедура перерозподілу елементів між кластерами.

Алгоритм методу «Кластеризація за схемою к-середніх»:

- вибрати k інформаційних точок як центри кластерів поки не завершиться процес зміни центрів кластерів;

- зіставити кожну інформаційну точку з кластером, відстань до центра якого мінімальна;
- переконатися, що в кожному кластері міститься хоча б одна точка. Для цього кожний порожній кластер потрібно доповнити довільною точкою, що розташована «далеко» від центра кластера;
- центр кожного кластера замінити середнім від елементів кластера;
- кінець.

Головні переваги методу k-середніх — його простота та швидкість виконання. Метод k-середніх більш зручний для кластеризації великої кількості спостережень, ніж метод ієрархічного кластерного аналізу (у якому дендограми стають перевантаженими і втрачають наочність)

Одним із недоліків простого методу є порушення умови зв'язності елементів одного кластера, тому розвиваються різні модифікації методу, а також його нечіткі аналоги (англ. fuzzy k-means methods), у яких на першій стадії алгоритму допускається приналежність одного елемента множини до декількох кластерів (із різним ступенем приналежності).

Незважаючи на очевидні переваги методу, він має суттєві недоліки:

- Результат класифікації сильно залежить від випадкових початкових позицій кластерних центрів;
- Алгоритм чутливий до викидів, які можуть викривлювати середнє;
- Кількість кластерів повинна бути заздалегідь визначена дослідником.

1.4 Підходи до побудови архітектури системи потокової обробки даних IoT

До цього часу послідовність обробки була технологічною " нішею ", яку використовували лише підгрупи малого бізнесу. Однак, завдяки швидкому зростанню SaaS, IoT та машинного навчання, компанії різних галузей зараз займаються постійною аналітикою. Важко знайти сучасну компанію, у якої немає

програми чи веб-сайту. Зі збільшенням трафіку до цих цифрових активів та зростаючим апетитом до вдосконаленої аналітики в режимі реального часу, потреба впровадження сучасної інфраструктури даних стає все більш поширеною.

У той час як архітектури традиційних партій масштаб менш може бути достатнім, обробки послідовностей пропонують ряд переваг, які інші платформи даних не можуть:

- Може обробляти нескінченні потоки подій; деякі дані, звичайно, структуровані таким чином. Традиційні засоби пакетної обробки вимагають зупинки потоку подій, збору пакетів даних та комбінування пакетів, щоб зробити загальні висновки. Хоча обробка потоків ускладнює комбінування та збір даних із декількох потоків, ви можете отримувати миттєву інформацію з великих обсягів потокових даних;
- Обробка в режимі реального часу або майже в реальному часі - більшість компаній використовують потокове передавання для аналізу даних у реальному часі. Хоча аналітика в режимі реального часу можлива і за допомогою потужних систем баз даних, дані часто спрощуються;
- Розпізнавання зразків у цих часових рядах - розпізнавання зразків з часом, таких як: В. Виявлення тенденцій у даних про відвідуваність веб-сайтів вимагає постійної обробки та аналізу даних. Пакетна обробка ускладнює це, оскільки розбиває дані на пакети, тобто деякі події діляться на два або більше пакетів;
- Легко масштабовані дані - Збільшення обсягу даних може порушити систему пакетного обслуговування та зажадати більше ресурсів або зміни архітектури. Сучасна інфраструктура обробки потоків є гіперскальованою і може обробляти гігабайти даних в секунду за допомогою одного потокового процесора. Таким чином, збільшеним обсягом даних можна легко керувати без зміни інфраструктури.

Більшість потокових стеків вже побудовані на черзі з відкритим кодом та власними рішеннями для конкретних проблем, таких як обробка потоків, зберігання,

інтеграція даних та аналітика в режимі реального часу. Незалежно від того, використовуєте ви сучасну платформу або традиційний інструмент, ваша потокова архітектура повинна включати ці чотири ключові елементи:

1. Брокер повідомлень / процесор потоку

Це елемент, який бере дані з джерела, так званого виробника, перетворює їх у стандартний формат повідомлення та передає безперервно. Тоді інші компоненти можуть прослуховувати та споживати повідомлення, надіслані брокером.

Перше покоління посередників повідомлень, таких як RabbitMQ та Apache ActiveMQ, базувалося на парадигмі Message Oriented Middleware (MOM). Згодом з'явилися гіперпродуктивні платформи обміну повідомленнями (їх часто називають потоковими процесорами), які більше підходили до парадигми потокового передавання. Два популярні інструменти потокового передавання - це потоки даних Apache Kafka та Amazon Kinesis.

2. Пакетні та реальні інструменти ETL

Потоки даних від одного або декількох посередників повідомлень повинні бути агреговані, перетворені та структуровані, перш ніж дані можна буде аналізувати за допомогою інструментів аналізу на основі SQL. Це робиться за допомогою інструмента ETL або платформи, яка отримує запити від користувачів, витягує події з черг повідомлень і застосовує запит для отримання результату, часто виконуючи додаткові агрегації та перетворюючи їх в агрегації. файлів. Результатом може бути виклик API, дія, візуалізація, попередження або, в деяких випадках, новий потік даних.

3. Аналіз даних / безсерверний механізм запитів

Після того, як потокові дані готові до споживання потоковим процесором, їх потрібно проаналізувати, щоб надати значення. Існує багато різних підходів до аналізу безперервних даних.

4. Постійне зберігання даних

З появою недорогих технологій зберігання даних більшість компаній зараз зберігають свої потокові дані про події. Ось кілька варіантів зберігання потокових даних, а також їх переваги та недоліки, які наведено в таблиці 1.4.1.

Таблиця 1.4.1

Порівняння варіантів зберігання

Варіант зберігання потокових даних	**Послуги**	**Недоліки**
У базі даних або сховищі даних, таких як PostgreSQL або Ama zon Redshift	Простий аналіз даних на основі SQL.	Важко масштабувати та управляти. Якщо це хмарне сховище, зберігання даних є дорогим.
Наприклад, у посереднику повідомлень через Kafka Persistent Storage	Розумне, неструктуроване зберігання даних - проблема даних у таблицях. Простота установки, без додаткових компонент ів.	Зберігання даних є проблемою, оскільки зберіга ння Kafka в 10 разів дорожче за зберігання в lake storage. Продуктивність Kaf ka краще читати (кешувати) останні дані (в кеші).
У lake storage, наприклад, Amazon	Розумно, не потрібно структурувати дані в таблиці. Низькі витрати на зберігання.	Висока затримка ускладнює аналіз у реальному часі. Важко провести аналіз на основі SQL.

1.5 Існуючі рішення і шляхи їх покращення

В даний час у цій галузі працює кілька платформ. Найвідоміший: Amazon Web Services IoT платформи , Google Cloud IoT Platform , ThingWorx , Oracle IoT , IBM IoT

Platform . Здебільшого існуючі рішення пропонують загальний набір функцій, включаючи зберігання даних та широкий спектр варіантів візуалізації та аналізу.

Здебільшого існуючі рішення пропонують загальний набір функцій, включаючи зберігання даних та широкий спектр варіантів візуалізації та аналізу.

Всі ці платформи комерційно закриті, що є величезним недоліком. Цей секрет ускладнює їх розширення, адаптацію та оптимізацію для конкретного сценарію використання. Компанії , які використовують платформи , які працюють в Відповідно до цієї моделі мають тенденцію залишатися в вигляді партії підвішування і не може змінитися , не витрачаючи час і гроші на іншій платформі.

Платформ з відкритим кодом майже немає. Однак можна розробити функціональну платформу, використовуючи компоненти та бібліотеки з відкритим кодом, оскільки в даний час доступно багато рішень з відкритим кодом, які можуть бути частиною системи, яка вже є стабільною та має розширені можливості. У цій роботі розглядаються, порівнюються та використовуються відповідні компоненти.

Існуючі платформи лише зрідка мають нестандартні рішення для очищення поганих даних. Однак загалом ці платформи можуть надавати базову статистику або інструменти машинного навчання, які підходять майже для будь-якого випадку використання. Однак слід зазначити, що очищення даних є важливою та складною частиною науковців з питань даних. Особливо перспективною областю є виявлення аномалій (виявлення відхилень). Виявлення аномалій в лінії в даний час одна області менш досліджувалися машинне навчання.

Крім того, універсальність розробленого рішення повинна пропонувати можливість спеціалізації платформи для певного сценарію. Наприклад, вам потрібно зберегти дані для всіх типів пристроїв. Однак, оскільки обробка цих даних відрізняється для різних видів використання пристроїв, рекомендується забезпечити можливість розширення платформи, визначивши певні ланцюжки операцій обробки даних для конкретних вимог.

Наприклад, розглянемо один з найбільш функціональних і популярних з існуючих платформ - на AWS IO платформи [6].

Основні компоненти системи:

- SDK для пристрою, що використовується для надсилання повідомлень;
- Вузол автентифікації та авторизації, який використовує сертифікати SigV4 або X.509;
- Пристрої шлюзу, що забезпечують зв'язок на основі протоколів pub / sub MQTT, WebSocket, HTTP 1.1;
- Зареєструйтесь, щоб зберегти інформацію про пристрій;
- "Тінь" пристроїв - механізм збереження останнього отриманого стану пристрою для подальшого його запиту через REST API;
- Механізм встановлення правил, за допомогою якого ви можете описати інтеграцію з іншими службами AW.

AWS IoT пропонує дуже складні засоби захисту, але він повністю прив'язаний до служб AWS, використовує незвичайний протокол MQTT і не пропонує API для визначення власних рядків даних.

Давайте подивимось на іншу платформу для Інтернету речей: платформу IBM IoT. Розробники розділили його функціональність на чотири основні блоки [7]:

- Підключення. Платформа пропонує різноманітні SDK для шлюзів та приладів, а також API для взаємодії з клієнтським програмним забезпеченням. Ці функції доступні для різних мов програмування, в тому числі в рішеннях НИЗЬКОЇ Оптимізовані для C- ради . Цей функціональний блок використовується для надійного підключення пристроїв до платформи даних. Найважливішими протоколами передачі є MQTT і HTTPS;
- Управління ризиками. Ця функціональна група відповідає за управління великими групами пристроїв, щоб користувачі могли переглядати інформацію про свої дії на панелях управління. Є також повідомлення про

незвичайні ситуації, які дозволяють вам менеджери , про ряд випадків, які швидко ізолюються ;

- Управління інформацією. Функціональний блок, за допомогою якого ви можете керувати зберіганням та архівуванням оперативної інформації та метаданих. Існують варіанти аналізу та обробки даних, звітів. Також є можливість вводити дані з різних джерел та платформ;
- Аналітика. Інтеграція з IBM Watson дозволяє користувачам виконувати складний аналіз даних із використанням існуючих підсистем Watson. Наприклад, блок аналізу природної мови, різні алгоритми машинного навчання, інструменти аналізу відео та зображень, аналіз текстових даних.

Тому головною перевагою платформи IBM IoT є широкий спектр функцій аналізу даних, реалізованих IBM Watson. Інші функції цілком характерні для таких платформ. Основним недоліком є закритість платформи і, отже, неможливість її розширення.

Ще однією функціональною платформою, яку можна переглянути, є Oracle IoT. Погляньте уважніше на їхні здібності, які можна розділити на три частини [8]:

1. Вхід. Цей пристрій має такі функції:
   - Віртуалізація пристроїв. Кожен пристрій представлений у вигляді набору ресурсів для абстрактного підключення пристрою та стандартизації інтеграції;
   - Надійне спілкування. Існує двостороння передача даних, яка забезпечує доставку повідомлень через ненадійні мережі та до / з пристроїв, які одночасно працюють і підключаються до мережі;
   - Гнучка топологія. Пристрої можуть підключатися до платформ за допомогою різних мережевих топологій, таких як: В. клієнтська бібліотека із програмним забезпеченням шлюзу або разом за допомогою REST API. Бібліотеки пристроїв, доступні в Java SE, Java ME, POSIX C. Підтримка не-IP протоколів, таких як шина, доступна для шлюзів.

2. Аналіз.:

- Потокова обробка. Аналіз потоку даних у реальному часі з агрегуванням, фільтрацією та кореляцією подій;
- Збагачення даних. Можливість збагатити необроблені потоки даних контекстною інформацією, такою як метадані пристрою, і створювати складені потоки даних;
- Пам'ять подій. Аналізовані потоки даних можна надсилати до сховищ та інтегрованих служб, які є частиною Oracle Cloud.

3. Інтеграція. Має такі функції:

- Об'єднання компаній. Дані IoT можуть передаватися в інше програмне забезпечення для управління процесами та підключатися до Oracle Cloud;
- REST API. Дозволяє інтегруватися з іншими системами, як у хмарі Oracle, так і поза нею;
- Команди пристрою та управління ними. Надає можливість надсилати повідомлення на пристрої, незалежно від того, підключені вони до мережі чи ні.

Як бачите, перевага Oracle IoT полягає в широкому діапазоні передачі даних та управління пристроями, а також у підтримці різних протоколів (особливо для шлюзів). Однак, щоб скористатися більшістю функцій, системні служби повинні використовувати Oracle Cloud.

РОЗДІЛ 2
АРХІТЕКТУРА СИСТЕМИ ТА ВИБІР ОСНОВНИХ КОМПОНЕНТІВ

2.1 Аналіз вимог до архітектури

Під час завантаження великої критичної системи потрібно вирішити дві важливі проблеми на стороні сервера:

- Відсутність блокування;
- Масштабування у міру збільшення та зменшення трафіку;
- Обробка помилок - помилка навколо: збої серверів, збої процесів, центри обробки даних.

Інструментом, яким користуються практично всі для вирішення цих завдань, є балансування навантаження. Ось як працює більшість з них.

У цій архітектурі балансування навантаження є єдиною точкою входу для клієнта або пристрою. Балансування навантаження веде реєстр на серверах, які можуть обробляти запит. Зазвичай цей реєстр оновлюється більш-менш вручну (або за допомогою сценаріїв). Якщо до пулу балансування навантаження слід додати новий сервер, на балансування навантаження надсилається виклик API. Подібним чином, коли сервер потрібно вивести з ладу (наприклад, для оновлення або зменшення масштабу), він повинен бути видалений з пулу вручну.

Щоразу, коли надходить запит клієнта, балансир навантаження вибирає сервер із пулу (як правило, випадковий або круговий) і передає йому запит. Відповідь сервера передається назад клієнту через балансування навантаження.

Ця система гарно масштабується, оскільки сам балансир навантаження повинен робити дуже мало, все, що він робить, - це запити проксі. Балансери навантаження, такі як HAProxy, можуть обробляти тони запитів (порядком десятків або сотень тисяч на секунду) і добре розподіляти їх на великій кількості серверів.

Залишається питання, скільки робочих серверів потрібно для обробки певного обсягу трафіку. Як дізнатися, чи потрібно масштабувати вгору чи вниз? Ви можете

поглянути на кілька речей, наприклад, час відгуку, завантаження процесора або використання пам'яті серверів. Однак це залишається складною проблемою.

Фактично на кожному великому сайті використовуються такі типи балансирів навантаження (будь то програмне чи апаратне забезпечення). Вони настільки поширені, що більшість хмарних провайдерів пропонують для них заздалегідь побудовані прилади, такі як ELB Amazon та Rackspace Cloud Load Balancers.

Хоча балансири навантаження чудово допомагають масштабувати сайт, збій з обробкою трохи складніший. Що робити, якщо один із серверів, зареєстрованих з балансиром навантаження, виходить з ладу? Запити, надіслані туди, просто ніколи не повернуть результат. Для вирішення цієї проблеми більшість балансирів навантаження періодично опитують усі сервери в пулі та викидають сервери, які неодноразово не відповідають на запити.

Однак викид невдалого сервера не відбувається миттєво. Зазвичай балансир навантаження повторює сервер кілька разів, щоб переконатися, що він справді не працює, перш ніж його викинути. Запити, які були проксі-сервером там, поки балансир навантаження не виявив несправності, швидше за все, просто не вдасться.

Отже, ми можемо визначити три проблеми / незручності з сучасним станом практики балансування навантаження:

- Управління реєстром з пулом серверів здійснюється вручну, ми повинні вручну (або за допомогою сценаріїв) додавати та видаляти сервери з пулу;
- Дізнатися, чи потрібно масштабувати систему вгору чи вниз, може бути складно;
- Виявлення помилок відбувається на основі опитування і вимагає часу, щоб виявити помилку.

Отже, підхід, заснований на старих добрих чергах повідомлень . У цій архітектурі модель дещо відрізняється. Потік такий:

- Клієнт надсилає запит до балансувача навантаження;

- Замість того, щоб направляти його на сервер безпосередньо, балансир навантаження розміщує запит у черзі повідомлень;
- Один або кілька серверів підписуються на цю чергу запитів, і один із них втягує повідомлення, що містить запит;
- Після обробки запиту та отримання відповіді сервер розміщує відповідь у черзі відповідей;
- Балансир навантаження підписується на чергу відповідей і пересилає відповідь назад клієнту.

З точки зору сервера це тягне модель. Коли сервер завантажується, він підписується на чергу запитів, тому немає ручного реєстру, який потрібно постійно оновлювати. Сервер сам вирішує, чи здатний він обробляти запит чи ні.

Система, що базується на черзі повідомлень, також прекрасно обробляє помилки. Якщо сервер аварійно завершує роботу під час обробки запиту, черга повідомлень виявляє втрачене з’єднання та передає повідомлення запиту іншому серверу: відсутні помилки.

Також простіше визначити, чи потрібно масштабувати вашу систему чи ні. Ви можете легко побачити, коли вашим серверам важко обробляти всі запити, коли черга повідомлень про запит зростає, а не залишається стабільною. Тривіально додати спостерігача черги повідомлень, який відповідно масштабує вашу систему: якщо черга переходить певний поріг, ви обертаєте додаткові сервери для обробки навантаження. Подібним чином, якщо черга повідомлень запиту постійно порожня, можливо, можна вбити один або кілька серверів.

Баланси навантаження на основі черг повідомлень приносять таблиці деякі інші переваги:

- Черги повідомлень зазвичай підтримують фільтрацію тем, що дозволяє гнучкі способи маршрутизації запитів: сервер сам може вирішити обробляти запити лише до певних шляхів або субдоменів, наприклад;

- Від'єднання реєстрації запитів та аналітики запитів (наприклад, часу відповіді) від звичайної обробки запитів стає набагато простішим за рахунок багатоадресного надсилання запитів та повідомлень відповідей до декількох черг (одна для регулярної обробки запитів, одна для реєстрації, одна для аналітики тощо).

2.2 Архітектура системи

2.2.1 Діаграма класів

- Connection клас для контролю підключень до пристроїв;
- DataRepository клас для взаємодії з місцем зберігання тимчасової інформації;
- Message клас описує вхідні дані;
- Data клас описує дані, які надсилаються в чергу;
- MyStompSessionHandler клас отримує повідомлення від пристроїв.

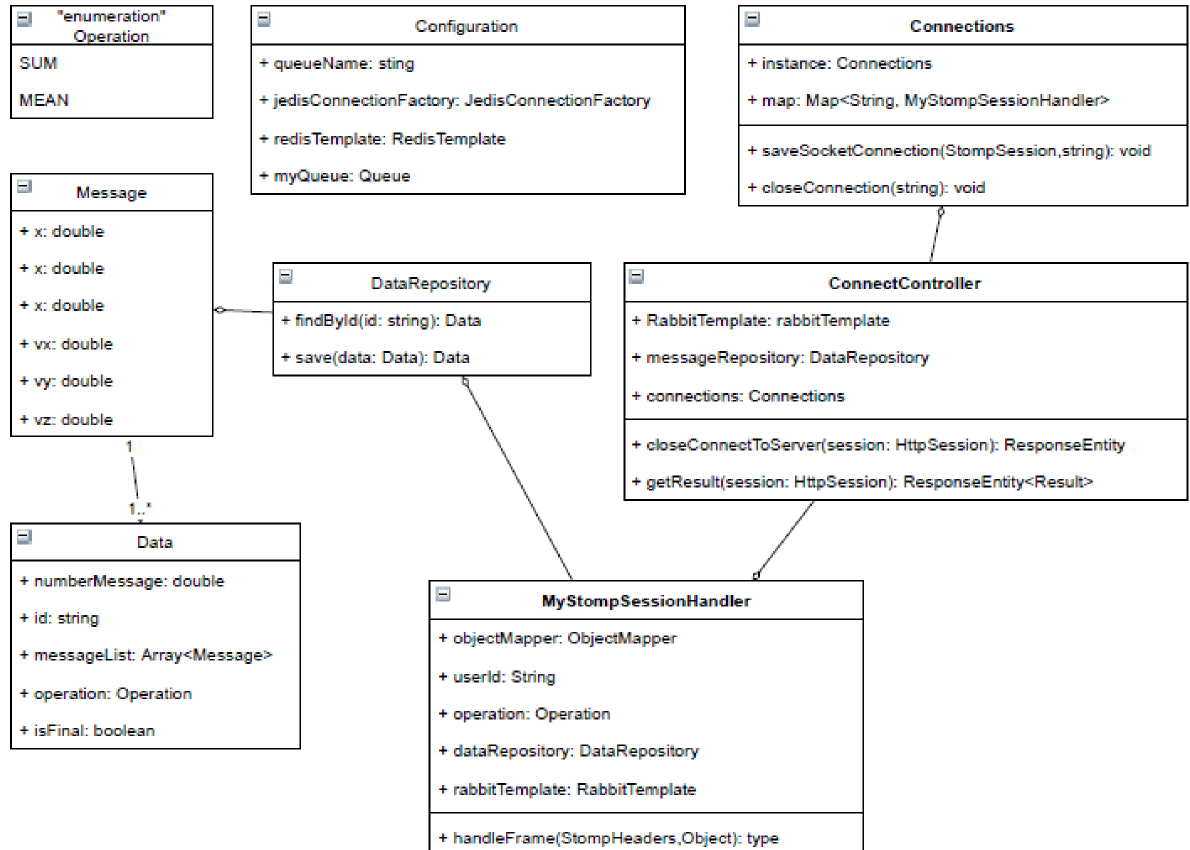

Рис. 2.2.1.1 Діаграма класів головного сервера

- Message клас описує вхідні дані;
- Data клас описує дані, які надсилаються в чергу;
- RabbitMQListener прослуховує та обробляє повідомлення з черги;
- Result клас описує результат роботи;
- SeqKMeans клас містить реалізацію алгоритму K-Mean;
- RabbitMQConfig клас описує конфігураціє черги;
- Configuration клас описує створення beans.

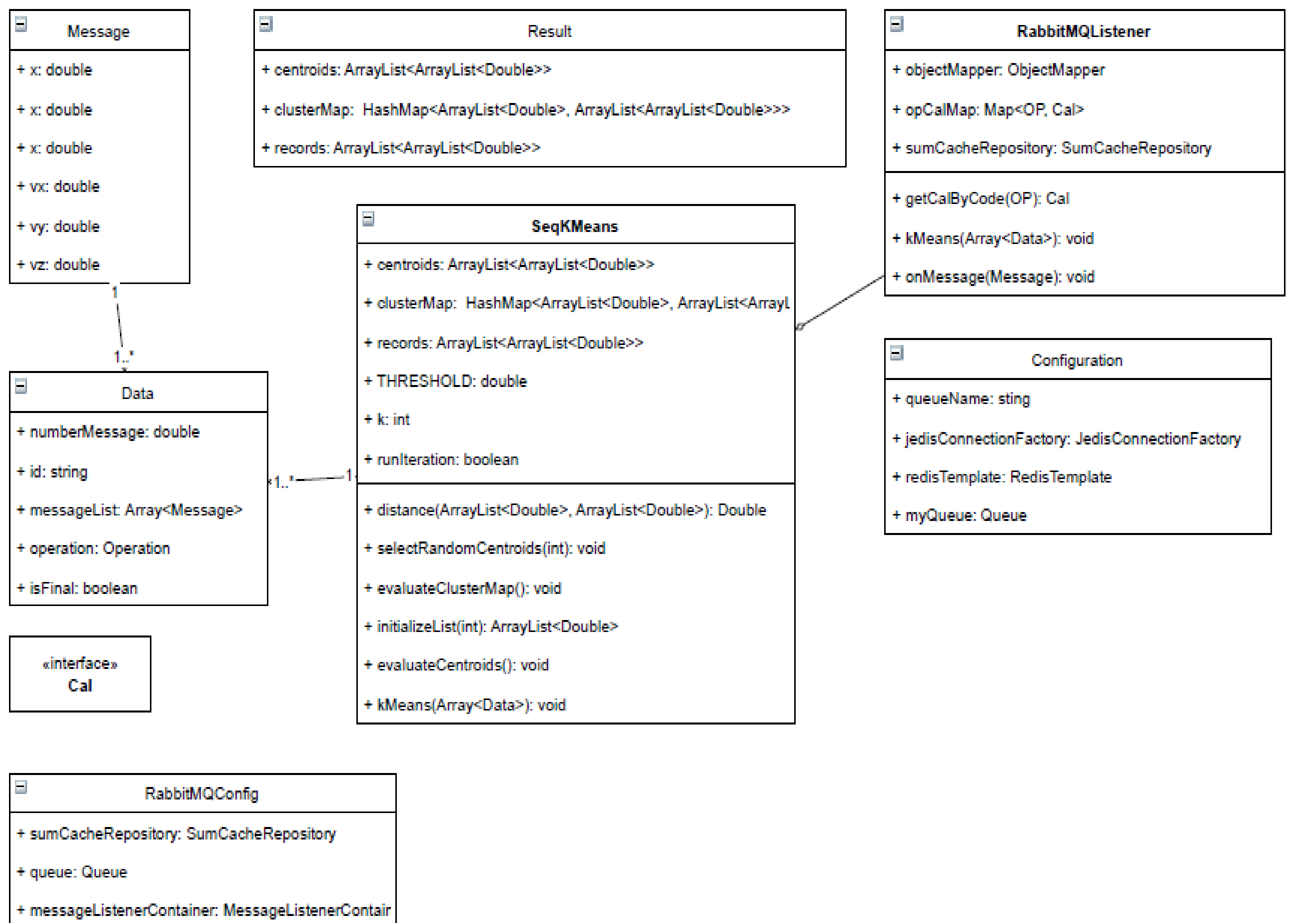

Рис. 2.2.1.2 Діаграма класів обчислювального сервера

2.2.2 Діаграма діяльності

- Користувач надсилає запит до головного сервера з повідомленням в якому вказано до якого пристрою підключися. Сервер підключається до пристрою та починає прослуховувати повідомлення, в разі помилки надсилає користувачу повідомлення про помилку підключення;

- Клієнт надсилає запит до головного сервера з повідомленням про закриття підключення до пристрою. Головний сервер перестає прослуховувати повідомлення пристрою;
- Клієнт надсилає запит до головного сервера з повідомленням отримання результатів. Сервер отримує результат з бази даних та передає клієнту.

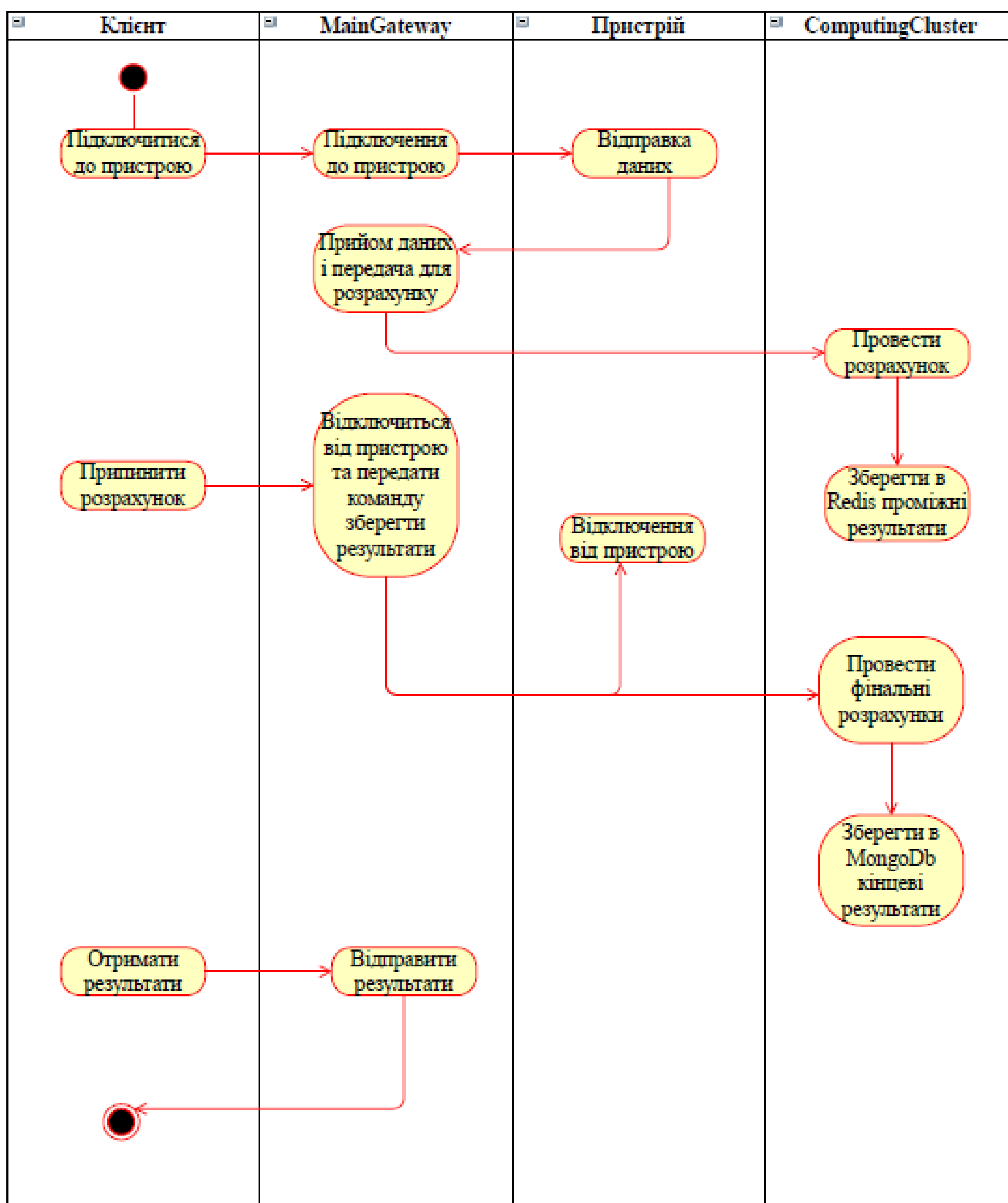

Рис. 2.2.2.1 Діаграма діяльності

2.2.3 Діаграма розгортання

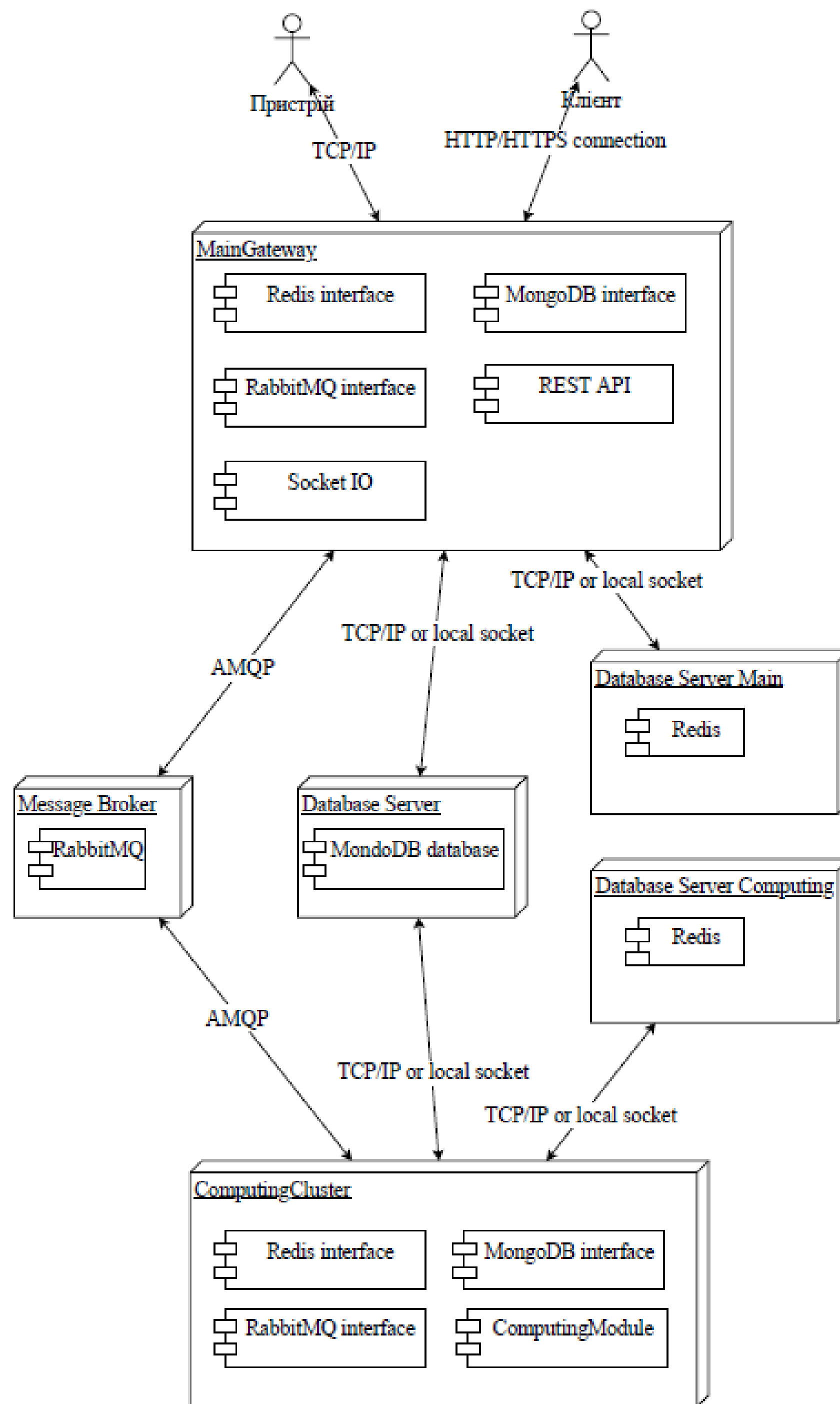

Рис. 2.2.3.1 Діаграма розгортання

- Клієнт – прямий користувач з системою;
- Пристрій – надсилає дані для обрахунків системі;
- MainGateway – головний сервер, що обслуговує користувачів та приймає повідомелння від пристроїв;

- MessageBroker – черга повідомлень в якій передаються дані для обрахунку
- DataBase – головне сховище результатів;
- DataBase Server Main – сховище для кєшування даних головного сервера;
- DataBase Server Computing - сховище для кєшування даних сервера який веде розрахунки;
- Computing cluster – сервера які отримують дані з черги повідомлень та проводять обрахунок даних.

2.3 Вибір засобів реалізації основних компонентів

2.3.1 Фреймворк Spring

Spring Framework - це прикладна програма з контейнером інверсії керування для платформи Java. Основні функції можуть використовуватися будь-якими програмами Java, але існують розширення для створення веб-додатків на платформі Java EE (Enterprise Edition) [9]. Хоча основні будівельні блоки не передбачають конкретної моделі програмування, вона стала популярною у спільноті Java як доповнення або навіть як заміна моделі Enterprise JavaBeans (EJB). Spring Framework містить кілька модулів, що пропонують широкий спектр функцій:

- Spring Core Container (це основний модуль Spring, який забезпечує контейнери BeanFactory та ApplicationContext);
- Аспектно-орієнтоване програмування;
- Угода про конфігурацію (в модулях Spring Roo пропонується швидке рішення для розробки програми компонентів на основі пружин);
- Доступ до даних: робота з реляційними системами управління базами даних на платформі Java за допомогою Java Database Connectivity (JDBC) та об’єктно-реляційних засобів відображення та з базами даних NoSQL;
- Контроль інверсії контейнера (конфігурація програмних компонентів та управління життєвим циклом об'єктів Java, що в основному здійснюється шляхом завантаження залежностей);

- Повідомлення: Налаштуйте ведення журналу об'єктів прослуховування повідомлень для прозорого розміщення повідомлень за допомогою звичайних повідомлень за допомогою служби повідомлень Java (JMS) та покращення доставки повідомлень за допомогою стандартного API JMS;
- Модель-View-Controller (HTTP та на сервері, створюючи гачки для розширення та побудови RESTful веб-додатків та веб-служб (передача репрезентативного стану);
- Структура RAS: налаштована процедура віддаленого відображення (RPC), сортування об'єктів Java у мережах, що підтримують виклик віддаленого методу Java (RMI), протоколи CORBA та HTTP, включаючи веб-службу SOAP;
- Управління транзакціями: поєднує декілька API управління транзакціями та координує обробку транзакцій для обробки Java;
- Розкриття інформації про конфігурацію та адміністрування об'єктів Java для локальної мережі локально або viveodpdalon або viveodpd end або viveodpd end або viveodiada Java Control;
- Тести: підтримка класу для написання та інтеграційних тестів та модульних тестів.

2.3.2 Фреймворк Spring Boot

Технологія Spring Boot є більш спеціалізованою. Spring Boot - приклад програми, яка використовує фреймворк Spring і може "просто запустити". Він поставляється заздалегідь налаштованим з найкращою конфігурацією та використанням платформи Spring та сторонніх бібліотек, щоб розпочати роботу з мінімальними зусиллями. Більшість програм Spring Boot вимагають дуже мало параметрів Spring. Подробиці:

- Ви можете створювати індивідуальні програми стрибків;
- Інтегруйте безпосередньо з Tomcat або Jetty (файли WAR не потрібні);
- Спрощена конфігурація Maven;

- Ви можете автоматично відрегулювати деталі пружини, якщо це необхідно;
- Абсолютно відсутні вимоги до генерації та конфігурації коду XML.

2.3.3 Фреймворк Spring Web MVC

Структура Spring Web MVC пропонує архітектуру моделі-view-controller (MVC) та готові компоненти, за допомогою яких можна розробляти гнучкі та вільно підключені веб-програми. Модель MVC відокремлює різні аспекти програм (вхідна логіка, бізнес-логіка та логіка користувальницького інтерфейсу) та забезпечує це безкоштовне з'єднання між цими елементами:

- Модель інкапсуляції використовує дані програми та, як правило, складається з POJO (звичайний клас Java);
- Viewer відповідає за візуалізацію моделей даних і зазвичай генерує вихідний HTML-код, який інтерпретує інтерфейс браузера Tare;
- Менеджер відповідає за обробку наборів даних користувачів та створення відповідних моделей та їх ідеальних ідей;
- Ось послідовність розділів, відповідальна за вхідний запит HTTP до DispatcherServlet;
- Після отримання запиту HTTP DispatcherServlet HandlerMapping рекомендує викликати відповідний контролер;
- Контролер приймає запит і дає відповіді на методичні методи обслуговування на основі використання OST GOST. Метод служби встановлює дані моделі на основі визначеної бізнес-логіки та повертає ім'я подання DispatcherServlet;
- DispatcherServlet допомагає ViewResolver вибрати вибраний перегляд запиту;
- Після завершення аналізу DispatcherServlet подає дані перегляду моделі, які створюються у браузері, як показано.

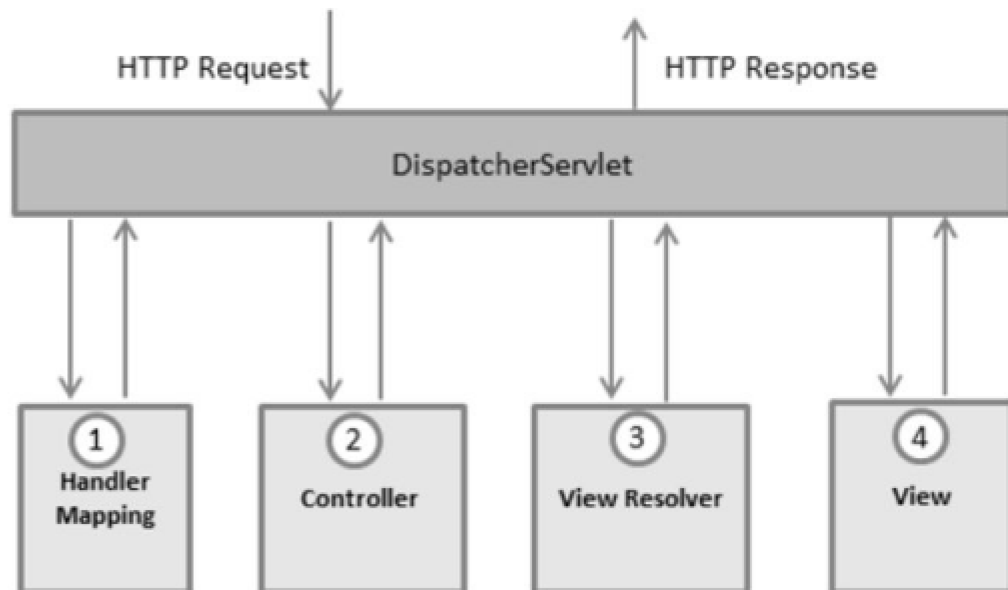

Рис. 2.3.3.1 Процес обробки запитів за допомогою DispatcherServlet

Структура Spring Web Model Controller (MVC) базується на DispatcherServlet, який обробляє всі запити та відповіді HTTP. Процес обробки запитів Spring Web MVC DispatcherServlet показаний на наступній схемі.

Усі вищезазначені компоненти, тобто Н. HandlerMapping, Controller і ViewResolver є частиною WebApplicationContext, розширення plainApplicationContext з деякими додатковими функціями, необхідними для веб-додатків.

2.3.4 Технологія Maven

Maven - це популярний інструмент для створення корпоративних проектів Java з відкритим кодом, призначених для більшої роботи над процесом збірки. Maven використовує декларативний підхід, який описує структуру та зміст проекту, а не підхід, заснований на завданнях, що використовується, наприклад, у Ant або традиційних файлах. Це допомагає забезпечити відповідність загальнокорпоративним стандартам розробки та зменшує час, необхідний для написання та збереження сценаріїв збірки.

Основою проекту Maven 2 є об'єктна модель проекту (коротше POM). Він надає детальний опис проекту, включаючи інформацію про управління версіями та конфігурацією, залежності, програмні та тестові ресурси, членів групи та структуру

тощо. POM доступний у вигляді XML-файлу (pom.xml), який зберігається в домашньому каталозі проекту.

Багато переваг Maven від стандартної практики, яку він використовує. Розробник, який вже працював над проектом Maven, негайно потребує структури та організації нового. Не потрібно витрачати час на організацію структур каталогів, умов та нестандартних сценаріїв проектування для кожного проекту. Хоча ви можете перевизначити кожне розташування каталогів для власних потреб, вам слід дотримуватися стандартної структури каталогів Maven з кількох причин:

- Це робить файл POM меншим та простішим;
- Це полегшує розуміння проекту та полегшує роботу інших розробників;
- Простота інтеграції плагінів.

Проект життєвого циклу лежить в основі Maven 2. Більшість розробників знайомі з такими поняттями фаз побудови, як побудова, тестування та розгортання. У Maven 1 реакція на плагіни позначається другим. Наприклад, для компіляції вихідного коду Java використовуйте модуль Java: $ maven java: compile. У Maven 2 ця концепція стандартизована на ряд добре відомих і чітко визначених фаз життєвого циклу. Отримавши доступ до нього для перегляду плагінів, розробник Maven 2 піклується про фазу життєвого циклу: компілюється $ mvn.

Деякі з найбільш корисних етапів життєвого циклу Maven 2:

- gener-source: генерує будь-який додатковий вихідний код, необхідний для програм, які зазвичай використовуються відповідними плагінами;
- Компіляція: компіляція вихідного коду проекту;
- Компіляція тестів: Компіляція тестових блоків проекту;
- Тест: запускає модульні тести (як правило, з JUnit) у каталозі src / test;
- Пакет: стискає скомпільований код у формат, що розповсюджується (JAR, WAR тощо);
- Інтеграційні тести: обробляє та, при необхідності, розгортає пакет в середовищі, в якому ви можете пройти інтеграційні тести;

- Встановити: Встановлює пакет у локальному сховищі для використання як залежність в інших проектах на вашому локальному комп’ютері;
- Розгортання: Виконайте перевірку в середовищі інтеграції або випуску, скопіюйте остаточний пакет у віддалене сховище, щоб поділитися з іншими розробниками та проектами.

Ці кроки ілюструють корисність рекомендованої найкращої практики Maven 2: Як тільки розробник ознайомиться з ключовими етапами життєвого циклу Maven 2, вони повинні почуватися невпевнено щодо етапу життєвого циклу проекту.

Фаза життєвого циклу створює плагіни, необхідні для виконання завдання. Виклик фази життєвого циклу автоматично створить усі попередні фази життєвого циклу. Оскільки фази життєвого циклу обмежені кількома користувачами, легкими для розуміння та добре організованими, називати їх новим життєвим циклом проекту Maven 2 дуже просто.

Однією з сильних сторін Maven 2 є його транзитивне управління залежністю. У Maven 1 ви повинні заявити кожен JAR, який прямо чи опосередковано вимагається програмою. Z Maven 2 не вимагає участі у відповідях на файли JAR для програмного модуля. Все, що вам потрібно зробити, - це надати інформацію про Maven, які бібліотеки йому потрібні, і Maven буде членом бібліотек, які потребують модулів.

У реальному корпоративному додатку вам може не знадобитися включати всі залежності в розгорнуту програму. Деякі файли JAR потрібні лише для модульного тестування, тоді як інші розгортаються під час роботи сервера додатків. Техніка, що називається визначенням залежності в Maven 2, дозволяє використовувати певні JAR лише тоді, коли вони вам дійсно потрібні, і виключати їх із шляху до класу, коли вони вам не потрібні.

Maven пропонує чотири зони залежності:

- компіляція: залежність від розміру компіляції доступна на всіх етапах. Це значення за замовчуванням;

- передбачено: вказана залежність використовується для компіляції програми, але не надається. Ви повинні використовувати це поле, якщо ви хочете, щоб JDK або сервер додатків обслуговував JAR. API сервлетів - хороший приклад;
- Час виконання: залежність від виконання не потрібна для компіляції, а лише для запуску, наприклад, драйверів Java Database Connectivity (JDBC);
- тест: залежність області тесту потрібна лише для складання та виконання тестів (наприклад, JUnit).

З мінімальними зусиллями ви можете створити професійний веб-сайт з якісним проектом. Розробка та підтримка веб-додатків стає набагато простішою, коли Maven інтегрований у процес створення сайтів шляхом постійної інтеграції або навіть автоматичної нічної збірки. Типовий сайт Maven може публікувати щодня:

- Загальна інформація про проект, наприклад, сховища джерел, відстеження помилок, члени команди тощо;
- Звіти про випробування та висвітлення випробувань;
- Автоматичний огляд коду за допомогою Checkstyle та PMD;
- Інформація про конфігурацію та версію;
- Залежності;
- Javado;
- Вихідний код в індексованому та багатоформатному HTML;
- Список членів команди.

Maven 2.0 - це потужний інструмент, який значно спрощує та стандартизує процес збірки. Maven просуває стандартну організацію проекту та рекомендації щодо дій та виконує більшу частину роботи. А стандартні плагіни, такі як конструктор сайтів, пропонують цінні інструменти проекту, які не потребують додаткових зусиль.

2.3.5 База даних Redis

Redis - це система управління базами даних NoSQL з відкритим кодом, яка працює зі структурами даних ключ-значення. Він використовується як для баз даних, так і для налаштування кеш-пам’ятки, посередників для повідомлень.

Він зосереджений на отриманні максимальної продуктивності від ядерних операцій (за оцінками приблизно 100 000 запитів SET та GET в секунду на сервері початкового рівня Linux). Інтерфейси доступу, написані на C, розроблені для більшості поширених мов програмування.

Redis зберігає всі дані у вигляді словника, в якому ключі пов'язані зі своїми значеннями. Однією з основних відмінностей між Redis та іншими сховищами даних є те, що значення цих ключів не обмежуються рядками. Підтримуються такі абстрактні типи даних: рядки, списки, набори, хеш-таблиці, упорядковані набори.

Тип даних значення визначає доступні йому операції (команди). Підтримуються операції високого рівня, такі як агрегування та різні набори, набори сортування.

2.3.6 База даних MondoDB

MongoDB є відкритою системою управління базами даних (СКБД), яка використовує документно-орієнтовану модель бази даних, що підтримує різні форми даних. Це одна з численних не реляційних технологій баз даних, які виникли в середині 2000-х років під банером NoSQL для використання у великих додатках та інших місцях обробки, що містять дані, які не вписуються в жорстку реляційну модель. Замість використання таблиць і рядків, як у реляційних базах даних, архітектура MongoDB складається з колекцій і документів.

Запис в MongoDB є документом, який є структурою даних, що складається з пар поля-значення. Документи MongoDB подібні до об'єктів JavaScript Object Notation, але використовують варіант Binary JSON (BSON), який вміщує більше типів даних. Поля в документах близькі до стовпців реляційної бази даних, а значення, які

вони містять, можуть бути різними типами даних, включаючи інші документи, масиви документів, згідно з керівництвом користувача MongoDB.

Документи, які також повинні включати первинний ключ як унікальний ідентифікатор, є основною одиницею даних у MongoDB. Колекції містять набори документів і функціонують як еквівалент реляційних таблиць бази даних. Колекції можуть містити будь-який тип даних, але обмеження полягає в тому, що дані в колекції не можуть бути розподілені між різними базами даних.

Оболонка mongo - це інтерактивний інтерфейс JavaScript для MongoDB, який дозволяє користувачам запитувати і оновлювати дані, а також проводити адміністративні операції. Оболонка є стандартним компонентом розподілів з відкритим вихідним кодом MongoDB. Після встановлення MongoDB користувачі підключають оболонку mongo до запущених MongoDB-екземплярів.

Формат зберігання та обміну даними BSON, що використовується в MongoDB, забезпечує двійкове представлення документів, подібних до JSON. Автоматична функція sharding - це ще одна ключова особливість, яка дозволяє розподілити дані в колекції MongoDB по декількох системах для горизонтальної масштабованості, оскільки обсяги даних і вимоги до пропускної здатності зростають.

СКБД NoSQL використовує єдину архітектуру майстра для узгодженості даних, з вторинними базами даних, які підтримують копії первинної бази даних. Операції автоматично копіюються до цих вторинних баз даних для автоматичного переключення. Найновіші технології MongoDB орієнтовані на хмарні та мобільні пристрої.

Як і інші бази даних NoSQL, MongoDB не вимагає попередньо визначених схем і зберігає будь-які типи даних. Це дає користувачам можливість створювати будь-яку кількість полів у документі, що полегшує масштабування баз даних MongoDB у порівнянні з реляційними базами даних.

Однією з переваг використання документів є те, що ці об'єкти відображаються у нативні типи даних у ряді мов програмування. Крім того, наявність вбудованих

документів зменшує потребу в об'єднанні баз даних, що може зменшити витрати. Основною функцією MongoDB є його горизонтальна масштабованість, що робить її корисною базою даних для компаній, що працюють з великими додатками даних. Крім того, sharding дозволяє базі даних поширювати дані по кластеру машин. Нові версії MongoDB також підтримують створення зон даних на основі клавіші shard. MongoDB підтримує ряд механізмів зберігання даних і надає інтерфейси API, які дозволяють третім сторонам розробляти власні механізми зберігання для MongoDB.

СКБД також має вбудовані можливості агрегації, які дозволяють користувачам запускати код MapReduce безпосередньо в базі даних, а не запускати MapReduce на Hadoop. MongoDB також має власну файлову систему GridFS, подібну до Hadoop Distributed File System (HDFS), в першу чергу для зберігання файлів, розмір яких перевищує обмеження розміру BSON на 16 МБ на документ. Ці подібності дозволяють використовувати MongoDB замість Hadoop, хоча програмне забезпечення бази даних інтегрується з Hadoop, Spark та іншими фреймворками обробки даних.

Хоча є багато переваг, у MongoDB є кілька недоліків. Завдяки політиці автоматичного заборони користувач встановлює один головний вузол у кластері MongoDB. Якщо ведучий виходить з ладу, ведений вузол автоматично стає новим. Цей перемикач забезпечує безперервність, але не миттєво - цей процес може зайняти до хвилини. Для порівняння, база даних Cassandra NoSQL підтримує кілька важливих вузлів. Головний вузол MongoDB також обмежує швидкість запису даних у базу даних. Дані реєстрації повинні бути записані на головний пристрій, а реєстрація нової інформації в базі даних обмежена можливостями цього головного вузла. Інша потенційна проблема полягає в тому, що MongoDB не забезпечує повну цілісність посилань через обмеження зовнішніх ключів, які можуть вплинути на цілісність даних. Крім того, стандартна автентифікація користувача не включена до баз даних MongoDB, що відповідає популярності розробників цієї технології.

MongoDB доступний у спільній та роздрібній версіях через MongoDB Inc. MongoDB Community Edition - це відкрита версія, тоді як MongoDB Enterprise Server

пропонує додаткові функції безпеки, механізми зберігання, функції адміністрування та автентифікації, а також моніторинг функцій за допомогою Ops Manager.

Графічний інтерфейс користувача (GUI) під назвою MongoDB Compass дозволяє користувачам працювати зі структурами документів, виконувати запити, індексувати дані тощо. MongoDB Connector для BI дозволяє користувачам підключати базу даних NoSQL до своїх інструментів бізнес-аналітики для візуалізації даних та створення звітів за допомогою запитів SQL.

Слідом за іншими постачальниками баз даних NoSQL, MongoDB Inc. представила хмарну базу даних як послугу під назвою MongoDB Atlas у 2016 році. Atlas працює на AWS, Microsoft Azure та Google Cloud Platform. MongoDB нещодавно випустив платформу для розробки додатків Stitch для MongoDB Atlas з наміром поширити її на локальні бази даних.

2.4 Протоколи і формати обміну даними між компонентами системи

Зв'язок з клієнтом здійснюється через зашифрований протокол HTTP / 2, який використовує криптографічний протокол TLS 1.2. Шифрування трафіку необхідно для запобігання перехоплення даних та інших атак. Стандартизований протокол HTTP / 2 2015 року дуже підходить для IoT, оскільки:

- Розроблено як оптимізацію HTTP / 1.1;
- Реалізовано стиснення заголовка HTTP (HPACK) за допомогою коду Хаффмана. Цей метод є дуже ефективним, але вимагає мало пам'яті, що дуже важливо для пристроїв з низьким енергоспоживанням;
- Такі методи, як конвеєр і мультиплексування, заохочують використовувати єдине з'єднання TCP та уникають повільних тристоронніх переговорів, що також вимагає додаткових ресурсів. У зашифрованому протоколі повторне використання з'єднання дозволяє уникнути ще більш тривалих переговорів про TLS;
- Забезпечує рівень протоколу PING, який можна використовувати для перевірки підключення пристрою;

- Підтримує сумісність високого рівня з HTTP / 1.1: методи, коди стану, URI, поля заголовка.

Зв'язок основного сервера з комп'ютерними серверами відповідно до розширеного протоколу черги повідомлень. AMQP - це протокол, який бере свій початок у фінансовому секторі. Він може використовувати різні транспортні протоколи, але пропонує надійний транспортний протокол, такий як TCP.

- AMQP забезпечує асинхронний зв'язок шляхом розміщення / підписки на повідомлення. Головною його перевагою є функція зберігання та передачі даних, яка гарантує надійність навіть після відмов мережі. Це забезпечує надійність завдяки таким гарантіям доставки повідомлень;
- Принаймні один раз: означає, що повідомлення буде надіслано один раз, незалежно від того, доставлене воно чи ні;
- Принаймні один раз: означає, що повідомлення буде доставлено рівно один раз, можливо, кілька разів;
- Рівно один раз: означає, що повідомлення буде доставлено лише один раз;
- Захист забезпечує TLS / SSL через протоколи TCP.

Недавні дослідження показали, що AMQP має низький рівень успіху доставки, коли пропускна здатність низька і зростає зі збільшенням пропускної здатності. Інше дослідження показує, що, порівнюючи AMQP із REST, згаданим вище, AMQP може надсилати більше повідомлень в секунду.

Крім того, середовище AMQP буде обробляти 300 мільйонів повідомлень на день з 2000 користувачами на п'яти континентах. Крім того, JPMorgan, американська компанія з банківських та фінансових послуг, використовує AMQP для відправки 1 млрд повідомлень на день.

Зв'язок основного сервера з пристроями IoT через протокол websocket. Websocket був розроблений в рамках проекту HTML 5 для розширення каналів зв'язку через TCP. Веб-сайт не є журналом запитів / відповідей або журналом публікацій / підписів. У Websocket клієнт ініціює рукостискання з сервером для створення сеансу.

Сама переговорність подібна до HTTP, тому веб-сервери можуть обробляти веб-сеанси та HTTP-з'єднання на одному порту. Однак після встановлення сеансу рукостискання не відповідає правилам HTTP. Фактично заголовки HTTP очищаються під час сеансу, а клієнти та сервери можуть обмінюватися повідомленнями через асинхронне повнодуплексне з'єднання. Сеанс може бути припинено, коли це більше не потрібно ні серверу, ні клієнту. Websocket розроблений, щоб зменшити клопоти з підключенням до Інтернету, одночасно забезпечуючи повний дуплексний зв'язок у режимі реального часу. Існує також підпротокол Websocket, який називається Протокол обміну повідомленнями веб-додатків (WAMP), який забезпечує системні повідомлення / передплати.

Websocket працює з надійним TCP і не використовує власні механізми надійності. За потреби сеанси можна захистити через веб-сайт за допомогою протоколів TLS / SSL.

Websocket мають лише 2 байти службових даних під час сеансу обміну повідомленнями. Як зазначалося у відповідних дослідженнях, запит HTTP (у REST) повторює інформацію заголовка у міру збільшення швидкості передачі даних, що збільшує затримку. За підрахунками, Websocket зменшує затримку втричі порівняно з половиною дуплексного запиту HTTP. Websocket не призначений для пристроїв з обмеженими ресурсами, оскільки попередні протоколи та його архітектура клієнт-сервер не підходять для додатків IoT. Однак він призначений для обміну в режимі реального часу, є безпечним, мінімізує експлуатаційні витрати та може забезпечити ефективні системи обміну повідомленнями з WAMP. Таким чином, він може конкурувати з будь-яким іншим протоколом, що працює через TCP

2.5 Масштабування системи

Наявність системи, яка може автоматично масштабуватись і відповідати належним чином на збільшене навантаження або вихід з ладу певних вузлів, є пріоритетом, але в деяких випадках це неактуально та вимагає великих ресурсів.

Розглядаючи необхідність та методи масштабування системи, щоб краще справлятися з навантаженням або відмовою, висуваються такі вимоги:

- Час відгуку / затримка;
- Доступність;
- Зберігання даних.

Важливою частиною побудови відмовостійкої системи, особливо коли функціональність поширюється на декілька мікросервісів, які можуть і не можуть працювати, є забезпечення безпечного зниження навантаження. Працюючи з одним монолітним додатком, нам не потрібно приймати багато рішень. Продуктивність системи залежить від роботи двійкового коду. Однак, використовуючи архітектуру мікропослуг, потрібно враховувати набагато складніші ситуації. Чим більше послуга залежить від інтеграції інших служб, тим більше належне функціонування послуги впливає на виконання завдань іншими службами. Використання технологій інтеграції, які дозволяють вимкнути сервер нижчого рівня в автономному режимі, може зменшити ймовірність планового та незапланованого простою на вищих сервісах.

У випадку недімпотентних операцій результат не змінюється після першого застосування, навіть якщо операція виконується кілька разів поспіль. Якщо операції ідемпотентні, ми можемо повторити виклик кілька разів без негативних наслідків. Це нам дуже знадобиться, якщо ми хочемо відтворювати повідомлення, коли немає впевненості в їх обробці. Це досить поширений спосіб усунення помилок. Масштабування системи здійснюється в основному з двох причин.

Розглянемо низку найпопулярніших технологій масштабування, які можна використовувати, і те, як їх можна застосувати до архітектури мікропослуг.

- Нарощування потенціалу. Деякі компанії можуть лише виграти від нарощування потенціалу. Більше шасі з більш швидким процесором та ефективнішою підсистемою введення-виведення часто може зменшити затримки та збільшити пропускну здатність, завдяки чому ви зможете виконати більше роботи за менший

час. Але такий тип масштабування, який часто називають масштабуванням по вертикалі, може бути занадто дорогим: іноді один великий сервер може коштувати набагато дорожче, ніж два менших сервери з меншою ємністю.

- Розподіл робочих навантажень. Одинарна мікрослужба на кожному хості, безумовно, краща, ніж модель, яка надає кілька мікропослуг на хості. Однак, щоб зменшити витрати на пристрій або спростити управління хостом, багато людей обирають існування декількох мікросервісів на одній фізичній машині. Оскільки мікросервіси працюють у незалежних процесах, які обмінюються даними по мережі, завдання їх подальшого переміщення до власних хостів для збільшення пропускної здатності та масштабованості не представляє особливої складності.

- Розподіл навантаження. Коли служба потребує стійкості до несправностей, вам потрібні способи вирішення критичних помилок. Для мікросервісу, що забезпечує синхронну кінцеву точку HTTP, найпростіший спосіб вирішити цю проблему (Рис. 2.5.1) - мати кілька хостів, що запускають екземпляри мікросервісу позаду балансувача навантаження. Споживачі мікросервісу не знають, присвоєно їм один екземпляр чи сотні таких екземплярів.

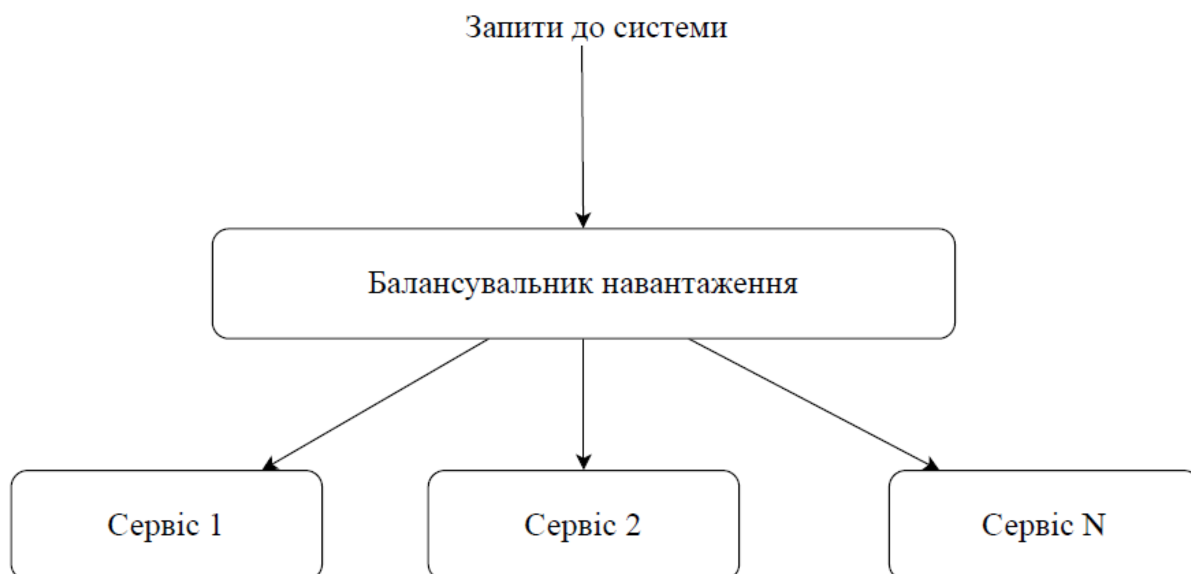

Рис. 2.5.1 Балансування

- Системи виконавців. Використання балансира - це не єдиний спосіб розподілити навантаження по декількох екземплярах служби та зменшити їх крихту. Крім того, залежно від типу оператора, це може бути система на основі литого складу. Ця модель також добре працює для публікацій, де зростає потреба в запуску додаткових екземплярів для відповідності вхідних завантажень. Поки черга функціонує сама по собі, вона залишається стабільною. Ця модель може використовувати масштабування для збільшення як пропускної здатності, так і стійкості до несправностей, так що вона ось-ось спрацює з ефектом невдалого (або відсутніх) виконавця. Робота триватиме довше, але нічого не буде витрачено даремно [8].

Масштабування основних даних також є можливою проблемою. Масштабування мікросервісів без збереження стану легко. Що робити, якщо ми зберігаємо дані в базі даних? Різні типи баз даних вимагають різних форм масштабування, і ви повинні знати, яка з них найкраща для вас.

Багато служб в першу чергу займаються зчитуванням даних. Масштаб для читання стає більшим, ніж великий для письма. Розташування даних тут може зіграти важливу роль.

У системі управління базами даних (СУБД), як MySQL або Postgres, дані можна копіювати з основного вузла в одній або декількох копіях. Служба може виконувати всі записи в один основний вузол, але вона може розподіляти зчитування по кількох репліках, визначених для зчитування даних.

Резервне копіювання з основної бази даних до репліки відбувається через деякий час після збереження. Це означає, що ця технологія читання може зберігати дані до завершення реплікації. Через деякий час, щоб прочитати стан готовності до роботи, фактичні дані вже доступні.

2.6 Балансування навантаження

У багатьох хмарних рішеннях використовуються завдання, що вимагають надання послуг. Коли служба піддається короткочасним піковим навантаженням у цьому середовищі, можуть виникнути проблеми з продуктивністю або надійністю.

Послуга може бути частиною того самого рішення, що і завдання, які її використовують. Або це може бути стороння служба, яка надає доступ до часто використовуваних ресурсів, таких як кеш-пам'ять або служба зберігання. Коли служба використовує кілька завдань, що виконуються одночасно, може бути важко передбачити кількість запитів, поданих до служби в будь-який момент часу.

Послуга може зазнати піків попиту, що призведе до перевантаження та неможливості своєчасно відповісти на запити. Заповнення служби великою кількістю одночасних запитів може також спричинити збій служби, якщо вона не може обробити конфлікт, викликаний цими запитами.

Рішення полягає в створенні черги між завданням і службою. Завдання та служба працюють асинхронно. Завдання надсилає повідомлення з даними, необхідними службі для черги. Черга діє як буфер і зберігає повідомлення, доки воно не буде видалено службою. Служба отримує повідомлення з черги та обробляє їх. Запити на цілий ряд завдань, які можна створювати з дуже різною швидкістю, можна надсилати до служби через ту саму чергу повідомлень. На цьому малюнку показано використання черги для збалансування навантаження на послугу.

Черга відокремлює завдання від служби, і служба може обробляти повідомлення з відповідною швидкістю, незалежно від кількості паралельних запитів завдань. Крім того, із завданням немає затримки, якщо служба не доступна під час надсилання повідомлення в чергу.

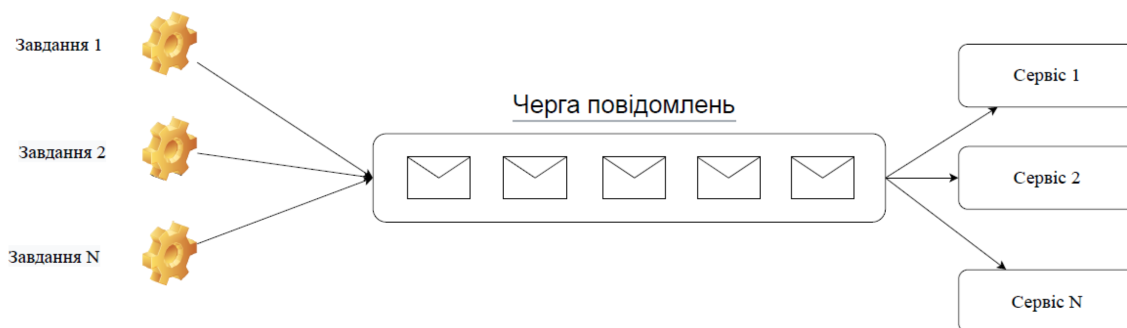

Рис. 2.6.1 Черга повідомлень

Такий підхід забезпечує наступні переваги:

- Це допомагає максимізувати час безвідмовної роботи, оскільки затримки служби не мають негайного та прямого впливу на програму, яка все ще може надсилати повідомлення в чергу, навіть якщо служба в даний час недоступна або не виконує повідомлення;
- Це дає вам максимальну масштабованість, оскільки ви можете змінювати як кількість черг, так і кількість служб за необхідності.

Це допомагає контролювати витрати, оскільки кількість послуг, що надаються, повинна бути достатньою для обробки середнього навантаження, а не пікового навантаження.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ.

3.1 Алгоритм роботи кожного компонента системи

3.1.1 Підключення клієнтом до пристрою

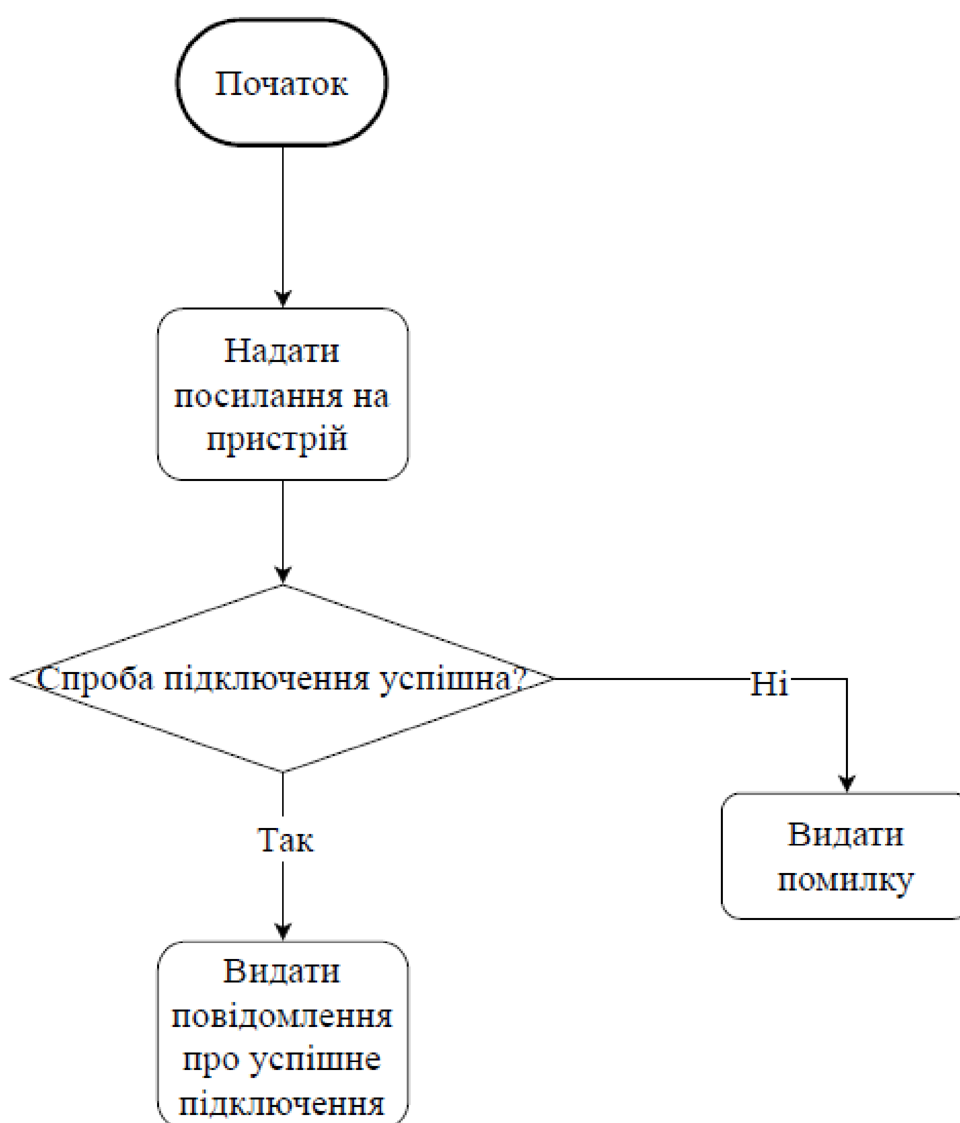

Рис. 3.1.1.1 Блок схема підключення клієнтом до пристрою

Блок схема описує процес підключення головного сервера на прослуховуван повідомлень від пристроїв. Клієнт надсилає запит на сервер і в разі успіху отримує код відповіді 200, а при помилці 503.

3.1.2 Отримання даних від пристрою

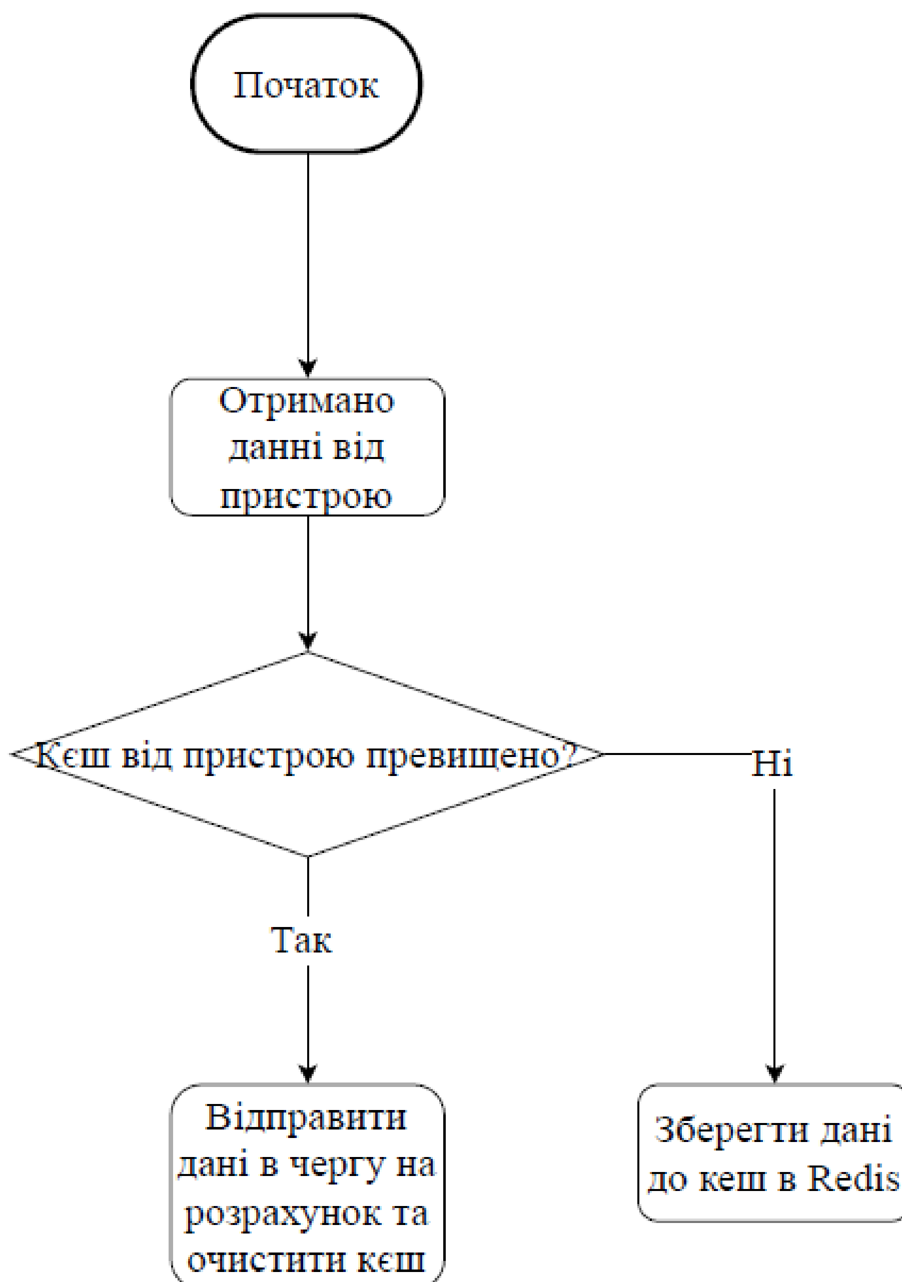

Рис. 3.1.2.1 Блок схема отримання даних від пристрою

Блок схема описує процес отримання даних від пристрою. Коли сервер отримує дані то проводиться перевірка не переповнений кєш для користувача в разу переповнення дані передаються в чергу повідомлень та кєш очиститься для отримання нових даних, якщо ліміт не перевищено то дані зберігаються в кєш.

3.1.3 Видача результатів клієнту

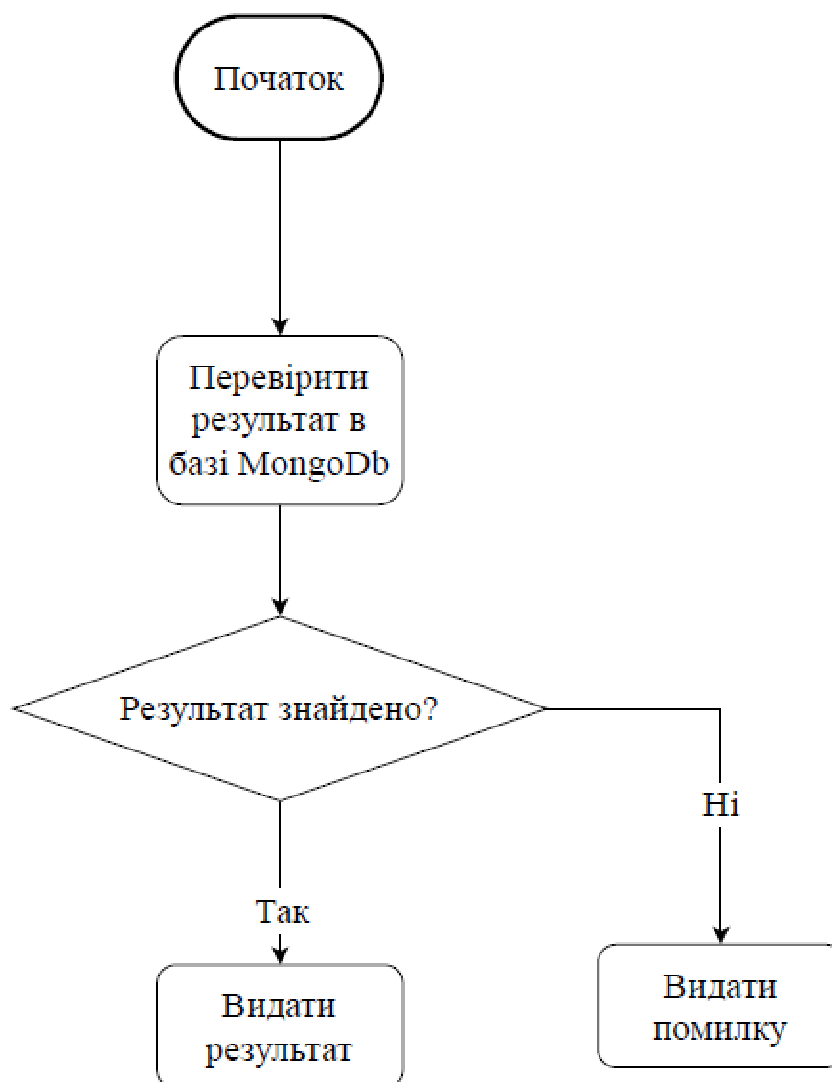

Рис. 3.1.3.1 Блок схема видачі результатів клієнту

Блок схема описує процес отримання результатів від серверу, якщо результати є то користувач отримає в форматі JSON, якщо результат відсутній помилку з кодом 404.

3.1.4 Обробка даних з черги

Блок схема описує процес отримання даних з черги повідомлення для обчислювання. Якщо в об’єкті даних присутній індикатор про завершення обрахунків

результат буде записаний в головну базу даних MondoDb, якщо індикатор відсутній то дані кєшуються в Redis.

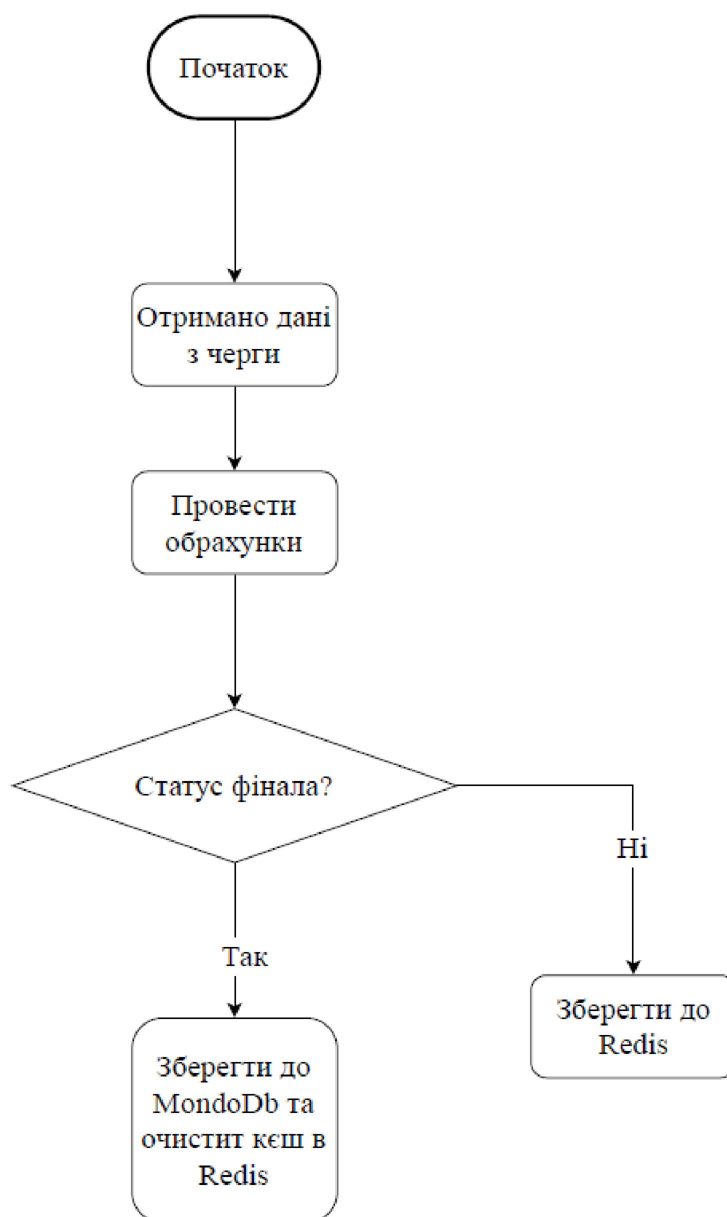

Рис. 3.1.4.1 Блок схема обробки даних з черги

3.2 API

3.2.1 API симулятора пристрою

Сервер має відкритий до підключення інтерфейс для підключення через протокол web-socket. Та кожні 600 мілісекунд надсилає порцію фейкових даних на топік.

3.2.2 REST API головного сервера

Головний сервера має REST-інтерфейс із такими кінцевими точками (endpoint), як описано в таблиці 3.3.1.1. Указано відносні адреси, кожна з яких має префікс /api.

Таблиця 3.2.2.1

Кінцеві точки REST API

Метод	**Адреса (шаблон адреси)**	**Опис**
POST	/connect	Використовується користувачами для задання команди головному серверу підключитися до пристрою.
PUT	/close	Використовується для задання команди головному серверу підключитися до пристрою.
GET	/result	Використовується для отримання результатів.

3.3 Оцінювання роботи системи

Таблиця 3.3.1

Кількість повідомлень в черзі

Кількість підключень до пристроїв	**Кількість повідомлень, що очікують в черзі в секунду**	**Кількість запущених серверів для обрахунку**
10	0-1	1
1000	10-15	1

10000	15-50	1
10000	5-7	2
100000	5-9	4

В результаті тестувань було виявлено, що система успішно горизонтально масштабується в залежності від кількості повідомлень до кількості запущених серверів, які в разі переповнення черги можливо підключати та втримувати навантаження.

Користувач має змогу наглядати за даними які надходять від пристрою, та відслідковувати проміжні результати Рис. 3.3.1.

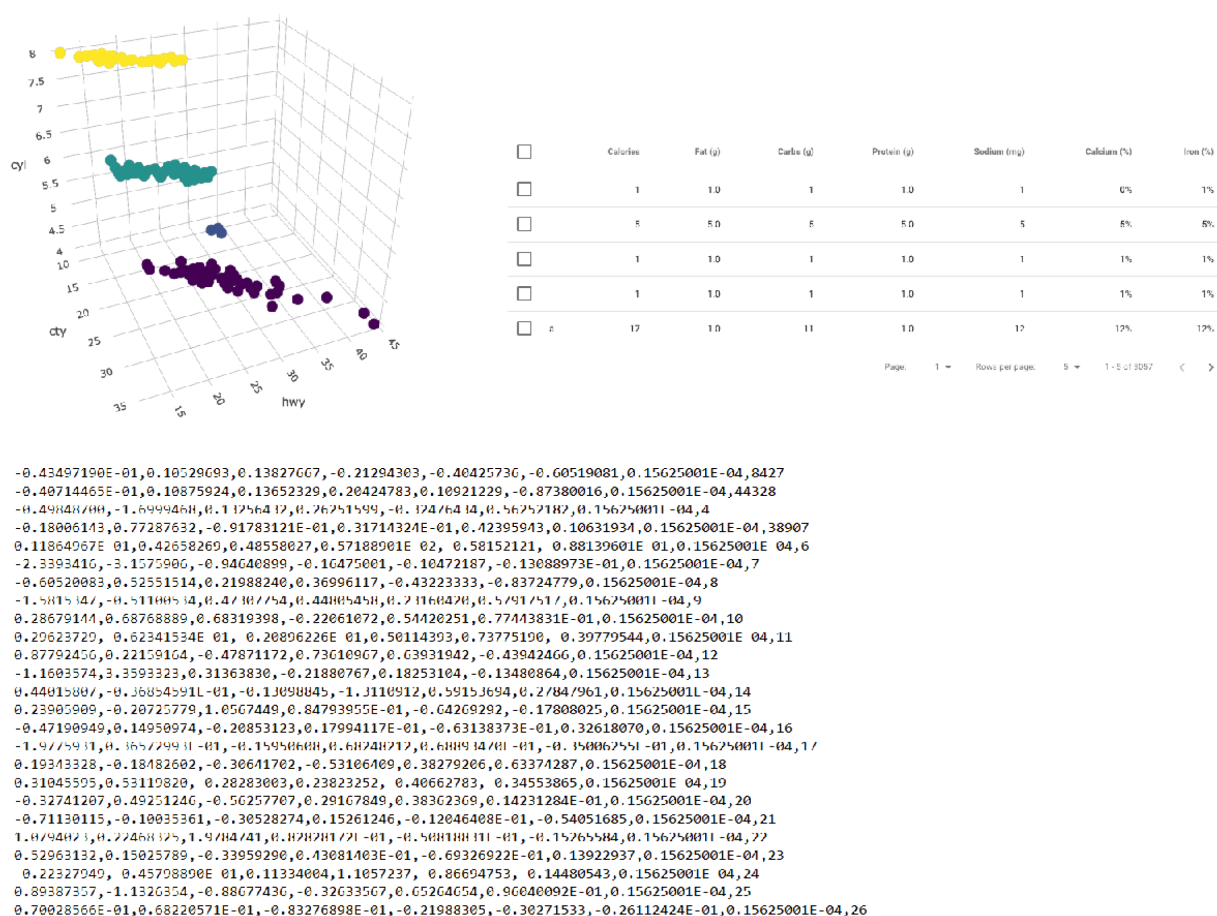

Рис. 3.3.1 Головний інтерфейс

3.4 Практичне застосування результатів

Можливості, напрямки та завдання, де це можна застосувати:

- Розробка розподілених додатків без кластерів з можливістю подальшої масштабованості;
- Доступ до стану програми без використання традиційних сховищ;
- Взаємодія з клієнтами в режимі реального часу, наприклад, для швидкого повідомлення про подію або розробки моделей прогнозування для прогнозного аналізу з мінімальними затримками;
- Багаторазове та безперервне агрегування, групування та згортання відфільтрованих результатів протягом певного періоду часу - наприклад, при розрахунку кількості біржових торгів акціями певної компанії за останні 10 хвилин;
- Обробляти платежі та фінансові операції в режимі реального часу, наприклад на фондових біржах, в банках та на страхових компаніях;
- Для відстеження та моніторингу автомобілів, вантажних автомобілів, автопарків та вантажів у режимі реального часу, наприклад у логістиці та автомобільній промисловості;
- Для постійного збору та аналізу даних датчиків від пристроїв IoT чи іншого обладнання, таких як заводи та вітряні парки;
- Збирати та негайно реагувати на взаємодію та замовлення клієнтів, наприклад, у роздрібній торгівлі, готельному та туристичному бізнесі та мобільних додатках;
- Для моніторингу пацієнтів, які перебувають у лікарні, та прогнозування змін у стані, щоб забезпечити своєчасне лікування в надзвичайних ситуаціях;
- Для підключення, зберігання та надання доступних даних, вироблених різними підрозділами компанії;
- Служити основою для платформ даних, керованих подіями архітектур та мікросервісів.

ВИСНОВКИ

Під час реалізації цього дипломного проекту було досліджено та встановлено Інтернет-платформу.

Зокрема, були визначені основні вимоги до таких систем; Вивчаються найважливіші методи вирішення проблем з обробкою великих для реалізації даних; Розглядаються існуючі платформи та показуються можливості для вдосконалення.

Розроблена платформа відповідає визначеним функціональним і нефункціональним вимогам і включає реалізацію таких підсистем: прийом, зберігання, обробка операційних даних; робота з адміністративними завданнями; Спостереження. Ви можете застосувати його до таких вибраних груп користувачів: пристроїв, адміністраторів, аналітиків та адміністраторів безпеки.

Така сервісно-орієнтована архітектура розподіленої платформи (мікросервіси) була запропонована та впроваджена, пропонуючи високу продуктивність та майже необмежену масштабованість. Ми можемо забезпечити дослідження потужності системи для мережевих пристроїв будь-якого розміру та для будь-якої підсистеми аналітичних даних. Ефективно впроваджуючи компонентні системи, ви можете оптимізувати вартість апаратних платформ. Платформа також гарантує високий рівень стабільності.

Детально описані різні аспекти рішень освітньої архітектури: від створення високорівневого підключення до функціональних модулів до функцій для ефективної реалізації вводу-виводу та застосування обраних схем авторизації до масштабної архітектури платформи. масштаб.

Система відповідає функціональним та нефункціональним вимогам, дозволяє динамічно необмежено горизонтально масштабуватися до лімітів бюджету підприємства.

На відміну від існуючих платформ, розроблене рішення працює добре, оскільки дозволяє легко адаптувати його до аналітичних даних, що містяться в різних пристроях та датчиках, що визначають ланцюжки пакетів даних. Протоколи та

формати даних були розроблені не лише для їхньої ефективності, а й для збільшення зручності та простоти розширень платформи.

Запропоновані та застосовані архітектурно-технічні рішення дуже універсальні, що дозволяє будувати різноманітні блоки з великими комп'ютерними системами та іншими Інтернет-платформами, що спеціалізуються на більш вузьких областях.

Систему було впроваджено в підприємство, та система успішно впоралась з поставленими цілями.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Доповідь компанії Gartner - Gartner. – [Електронний ресурс]. – Режим доступу: http://www.gartner.com/newsroom/id/3165317
2. Доповідь компанії Cisco – Cisco – [Електронний ресурс]. – Режим доступу: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoE_Economy.pdf
3. Nathan Marz. Big Data: Principles and best practices of scalable realtime data systems / Nathan Marz, James Warren. – Manning Publications, 2015. – 328 p
4. Сайт general - general [Електронний ресурс.] - Режим доступу http://www.generalpicturerecognition.co
5. Березький О.М. Синтез альтернативних рішень при структурному проектуванні систем автоматизованої мікроскопії / О.М. Березький, К.М. Березька, Ю.М. Батько, Г.М. Мельник // Науковий вісник Українського державного лісотехнічного університету. – 2009. – Т. 19 (5), № 23. – С. 258– 268.
6. Інформація про платформу AWS IoT. - AWS – [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/iot/how-it-works/.
7. Інформація про платформу IBM IoT Platform. - IBM – [Електронний ресурс]. – Режим доступу: http://www.ibm.com/internet-of-things/iot-platform.html
8. Інформація про платформу Oracle IoT. - Oracle – [Електронний ресурс]. – Режим доступу: https://cloud.oracle.com/iot.
9. Інформація про платформу Java. - Oracle – [Електронний ресурс]. –Режим доступу: https://www.oracle.com/java/technologies/java-ee-glance.html

Додаток А – Software Architecture Document (SAD)

Taras Shevchenko National University of Kyiv

Distributed system for real-time ML stream processing with load balancing

Software Architecture Document (SAD)

CONTENT OWNER: Vashchenko Serhii

DOCUMENT NUMBER:	**RELEASE/REVISION**:	**RELEASE/REVISION DATE:**
1	Version 1.0	March 2021

Table of Contents

1.1 Document Management and Configuration Control Information 65
1.2 Purpose and Scope 65
1.3 Viewpoint Definitions 65
1.3.1 Algorithm of multimedia application Viewpoint Definitions 65
1.3.1.1 Abstract 65
1.3.1.2 Stakeholders and Their Concerns Addressed 65
1.3.1.3 Elements, Relations, Properties, and Constraints 65
1.3.1.4 Language(s) to Model/Represent Conforming Views 65
2. Architecture Background 67
2.1. Problem Background 67
2.1.1. System Overview 67
2.1.2. Goals and Context 67
2.1.3. Significant Driving Requirements 67
2.2. Solution Background 67
2.2.1. Architectural Approaches 67
2.2.2. Analysis Results 68
2.2.3. Requirements Coverage 68
3. Referenced Materials 69
4. Directory 69
4.1 Glossary 69
4.2 Acronym List 69

1.1 Document Management and Configuration Control Information

- Revision Number: 1
- Revision Release Date: 22/05/2021
- Purpose of Revision: creating the final version of the documentation

1.2 Purpose and Scope

Purpose: to develop and implement a system for real-time ML stream processing with load balancing.

In Scope

The project will be based on creating an system for real-time ML stream processing.

- Load balancing.
- ML

1.3 Viewpoint Definitions

1.3.1 Algorithm of multimedia application Viewpoint Definitions

1.3.1.1 Abstract

Views that correspond to the point of view on the algorithm of the system. The procedure for determining actions, entry points, exit points, transitions.

1.3.1.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include

- system build engineers who use the elements to produce a running version of the system;
- testers need to know how the program should behave in different situations;
- configuration management specialists who are in charge of maintaining current and past versions of the elements;

1.3.1.3 Elements, Relations, Properties, and Constraints

Elements of the program algorithm are a sequence of precise instructions aimed at solving a specific problem. Every action in the algorithm is connected and logical. There are different types of interactions between them: linear algorithm, full branch algorithm, incomplete branch algorithm, cycle with precondition, cycle with postcondition, cycle with parameter.

Requirements and Limitations

- Attention. The algorithm consists of a series of individual commands, each of which is executed for a limited time. Only after completing one command does the executor start executing another.
- Uniqueness. Each command of the algorithm clearly defines the actions of the executor and does not allow them to be interpreted twice. The order of the operations is also clear.
- Official. Any executor who has a certain system of commands from the executor can, without knowing the nature of the task, execute the algorithm and get the same result.
- Dimensions. The algorithm must offer the ability to solve a group of typical problems and to change the input (output) data within certain permissible limits.
- Finiteness. The algorithm consists of a finite number of actions, the knees of which are executed for a limited period of time. The execution of the algorithm cannot end in an unsafe situation or not at all.
- Efficiency. Executing the algorithm with valid input produces the expected result.

1.3.1.4 Language(s) to Model/Represent Conforming Views

- plain text using indentation or outline form
- Formula-verbal - presentation of algorithms in educational and scientific activities using the language of mathematical formulas with verbal explanations in natural language.

- Graphic - representation of algorithms in the form of graphical schemes (block diagrams or block diagrams UML) to simplify the development and analysis of algorithms, facilitate the transition from writing algorithms to writing programs.
- Software - presentation of algorithms in a programming language for their further processing on a computer.

2. Architecture Background

2.1. Problem Background

2.1.1. System Overview

- Receiving and storing data. It receives it at the speed of data reception (ie with minimal delays) and stores it in a conditionally constant data warehouse (message queue).
- Data Processing. Performs thread processing and analytical queries. Data transfer is carried out between a conditionally permanent data warehouse and a database (permanent data warehouse). A degenerate case of data processing, if it is not required: a simple copy of information from the message queue to the database.

2.1.2. Goals and Context

- Scalability Increasing computing resources should increase system performance accordingly (dependencies should be as close to linearity as possible). In addition, scalability should be horizontal to the needs of systems with Big Data and provide a cloud to meet, ie increase system performance by increasing the number of nodes in the network will increase. Cloud data center. The horizontal scale is also much cheaper than the vertical one.
- Fault tolerance. There is a need to provide a system architecture that includes backup components (replication components) and can respond to errors and overloads during operation without shutting down the service completely. Diagnosis of tolerance is especially important for those parts of the system that receive and store data. If the server is unavailable, devices simply cannot store their data for a long time due to limited memory and then transfer it to the server.

2.1.3. Significant Driving Requirements

The Architecture Tradeoff Analysis Method (ATAM) is a method for evaluating software architectures relative to quality attribute goals. ATAM evaluations expose architectural risks that potentially inhibit the achievement of an organization's business goals.

In Bass they discuss about “Quality attributes”. These attributes are certain characteristics that the final software should possess to some extent (as given in the requirements). present six of the most common qualities; availability, modifiability, performance, security, testability, and usability.

If a horizontal scalability system is used as an example, these quality attributes can be sorted according to importance. One such arrangement may be (1) ease of connection, (2) the amount of data in the queue. Based on such an order, the developing organization can prioritize efforts and other resources to achieve the desired results.

2.2. Solution Background

2.2.1. Architectural Approaches

The software architecture of a computer program or system is the structure or structures of the system, the software elements, the properties visible from the outside of these elements and the relationships between them.

Examples:

1. The structure (s) of the system. This implies that a program consists mainly of several elements and that the order between them is important. It also implies that this order can be changed in response to different conditions.
2. Include the software items. This implies that the actual elements chosen to perform the task are part of the software architecture. This in turn implies that while the pieces can be interchangeable, choosing one over the other is an architectural choice.
3. The properties visible from the outside of these elements. At the same time as you add every last bit of code to the software architecture, you are also abstracting at a sufficient level and examining the observable effects of different elements of the software architecture. This allows

us to put restrictions on the different elements, both in terms of interfaces and in terms of relationships between the elements.

4. And the relationships between them. Finally, we come back to the relationships between the elements. This definition makes it clear that the shape of the final product is important and that the various roles that the elements play and the relationships between them are also part of the software architecture.

Responsible design. In the article "Object-Oriented Design: A Responsibility-Driven Approach", authors Wirfs-Brock and Wilkerson present a design method that focuses on the responsibilities of different entities in a system. You start by solving some of the problems with the data-driven design methodology, including the fact that it can break the encapsulation when the structure of an object becomes part of its definition. Then they describe the responsible approach. In responsibility-driven design, designers look at the relationships between different modules like a client-server relationship.

A server is an object that can perform a service (eg, provides data) and a client is an object that requests services (eg, requests data). By designing modules in this way, you are not binding the actual structure of the various modules or classes in the definition, but creating objects that have clearly defined services that they can perform. If we look at a game engine, any main module can be seen as a server. For example, the renderer provides the service to generate an image that can be viewed by the end user. The physics engine can calculate the interactions of objects and the sound engine can emit sound. All of these are great examples of object-oriented design because they provide an encapsulated service and can in theory be replaced by another module that runs the same services (for example, by adding a wrapper to the new module).

2.2.2. Analysis Results

Such a service-oriented distributed platform architecture (microservices) has been proposed and implemented, offering high performance and almost unlimited scalability. We can provide system capacity research for network devices of any size and for any analytical data subsystem. By effectively implementing component systems, you can optimize the cost of hardware platforms. The platform also guarantees a high level of stability.

Such a service-oriented distributed platform architecture (microservices) has been proposed and implemented, offering high performance and almost unlimited scalability. We can provide system capacity research for network devices of any size and for any analytical data subsystem. By effectively implementing component systems, you can optimize the cost of hardware platforms. The platform also guarantees a high level of stability.

2.2.3. Requirements Coverage

Development Requirements:

- IntelliJ IDEA Ultimate;
- Programming language JAVA;
- Spring;
- Hibernate;

System Requirements

- Windows 7, 8, 8.1, 10;
- 2 GB RAM or higher;

3. Referenced Materials

1	Bass, Clements, Kazman, Software Architecture in Practice, second edition, Addison Wesley Longman, 2003.
2	ANSI/IEEE-1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 21 September 2000.
3	R. Wirfs-Brock and B. Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. SIGPLAN Not., 2010.
4	Jeff Ward. What is a Game Engine?, April 2008. URL: http://gamecareerguide.com/features/529/what_is_a_game_.php.
5	Andrew Rollings and David Morris. Game Architecture and Design: A New Edition. New Riders Games, 2003.
6	Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003

4. Directory

4.1 Glossary

Term	Definition
Software architecture	The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
Spring	is an application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE (Enterprise Edition) platform. Although the framework does not impose any specific programming model, it has become popular in the Java community as an addition to the Enterprise JavaBeans (EJB) model. The Spring Framework is open source.
Hibernate	is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object–relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.
Software elements	components, connectors, and data--constrained in their relationships in order to achieve a desired set of architectural properties.

4.2 Acronym List

Term	Definition
ML	Machine learning
OS	Operating System
SAD	Software Architecture Document