Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

”Допущено до захисту”
Завідувач кафедри кібербезпеки
та програмного забезпечення
д.т.н., професор
_____ Олексій СМІРНОВ
“____” _____ 2024 р.

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА

за другим (магістерським) рівнем вищої освіти

на тему:

“ Дослідження та програмна реалізація системи безпеки на основі бездротових технологій ”

Виконав здобувач вищої освіти
ІІ курсу, групи КІ-23М_____
ОПП «Комп'ютерна інженерія»
спеціальності 123 «Комп'ютерна інженерія»
_____Д.В. Довгань
«____» _____ 2024р.

Керівник проекту
кандидат технічних наук, доцент
_____ О.А Кислун
«____» _____ 2024 р.
Рецензент _____
_____

м. Кропивницький

Центральноукраїнський національний технічний університет

Факультет *Механіко-технологічний*

Кафедра *Кібербезпеки та програмного забезпечення*

Освітній ступінь *магістр*

Галузь знань *12 "Інформаційні технології"*

Спеціальність *123 "Комп'ютерна інженерія"*

Освітньо-професійна (освітньо-наукова) програма *"Комп'ютерна інженерія"*

ЗАТВЕРДЖУЮ
Завідувач кафедри
д.т.н., проф.
_____Олексій СМІРНОВ
"____" _____2024 року

ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ ЗА ДРУГИМ (МАГІСТЕРСЬКИМ) РІВНЕМ ВИЩОЇ ОСВІТИ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Довганю Дмитру Валентиновичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження та програмна реалізація системи безпеки на основі бездротових технологій*

2. Керівник роботи *Кислун Олег Андрійович, канд. техн. наук, доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу № 19-13 від 07.08.24

3. Строк подання роботи до захисту *15.12.2024 р.*

4. Мета та завдання випускної кваліфікаційної роботи: *Метою роботи є дослідження та програмна реалізація розподіленої сервісно-орієнтованої платформи Інтеренету речей для забезпечення високої продуктивності та захист інформації*

5. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Призначення та область використання.
2.Перегляд аналогічних існуючих систем.
3. Опис і обґрунтування проектних рішень.
4. Етапи програмування системи.
5. Впровадження системи в промислову експлуатацію.
6. Наукова новизна
7. Маркетингове та економічне обґрунтування ІТ-проєкту
8. Заходи з охорони праці та техніки безпеки.
9. Висновки.

6. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Наукова новизна	*1 аркуш*
Структурна схема системи	*1 аркуш*
Функціональна схема системи	*1 аркуш*
Блок-схема алгоритмів роботи системи	*2 аркуші*
Діаграма взаємодії процесів	*1 аркуш*
Маркетингове та економічне обґрунтування ІТ-проєкту	*1 аркуш*

6. Консультанти по роботі, із зазначенням розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Маркетингове та економічне обґрунтування ІТ-проєкту	Доренська А.О	09.11.2024	17.11.2024
Охорона праці	Марченко К.М., к.т.н., доцент	03.11.2024	21.11.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів випускної кваліфікаційної роботи за другим (магістерським) рівнем вищої освіти	Строк виконання етапів випускної кваліфікаційної роботи за другим (магістерським) рівнем вищої освіти	Примітка
1.	Аналіз існуючих систем	12.10.2024	
2.	Постановка задачі, оформлення ТЗ	18.10.2024	
3.	Розробка моделі компонента	23.10.2024	
4.	Розробка структур даних	25.10.2024	
5.	Розробка алгоритмів зв’язку та відображення	32.10.2024	
6.	Програмування алгоритмів	11.11.2024	
7.	Маркетингове та економічне обґрунтування ІТ-проєкту	13.11.2024	
8.	Розрахунки з охорони праці та техніки безпеки	16.11.2024	
9.	Оформлення ПЗ	18.11.2024	
10.	Попередній захист роботи	04.12.2024	

Дата видачі завдання
«__»_____2024 р.

Підпис керівника

_____ _____
(прізвище та ініціали)

Завдання прийнято до виконання
«__»_____2024 р.

Підпис здобувача

_____ _____
(прізвище та ініціали)

АНОТАЦІЯ

Довгань Д.В. Дослідження та програмна реалізація системи безпеки на основі бездротових технологій. 123 Комп'ютерна інженерія. Центральноукраїнський національний технічний університет. Кропивницький. 2024.

В даній магістерській роботі розроблено програмне забезпечення, яке призначено та побудови серверних рішень які забезпечують роботу IoT з розробкою підсистеми безпеки передачі даних.

Метою розробки є дослідження та програмна реалізація для побудови системи серверних рішень, які забезпечують роботу Інтернету речей. Розробка хмарної платформи для Інтернету речей

Об'єктом дослідження є процес побудови рішень для робити IoT.

Предметом дослідження є серверні платформи для побудови рішень для роботи IoT.

Методи дослідження базуються на методах теорії кодування, методах математичної статистики, методах розробки програмного забезпечення.

Результат роботи – розробка хмарної платформи для Інтернету речей.

В процесі роботи над програмною моделлю виконано аналіз існуючих апаратних та програмних засобів. В повній мірі описані всі компоненти розробленого програмного забезпечення.

Розроблено зручний інтерфейс користувача. Наведені інструкції по роботі з програмними засобами.

Програма може використовуватися на ПЕОМ архітектури IBM PC з ОС Windows XP/Vista/7/8/10/11.

Програму розроблено в середовищі Java та Scala.

Ключові слова: комп'ютерна науки, IoT, серверні платформи,обробка даних.

ABSTRACT

Dovgan D.V. Research and software implementation of a security system based on wireless technologies. 123 Computer engineering. Central Ukrainian National Technical University. Kropyvnytskyi. 2024.

In this master's thesis, software is developed, which is designed and the construction of server solutions that ensure the operation of IoT with the development of security subsystems for data transmission.

The purpose of the development is research and software implementation for building a system of server solutions that ensure the operation of the Internet of Things.

Development of a cloud platform for the Internet of Things

The object of research is the process of building solutions for making IoT.

The subject of research is server platforms for building solutions for IoT operation.

Research methods are based on coding theory methods, mathematical statistics methods, and software development methods.

The result of the work is the development of a cloud platform for the Internet of Things.

In the process of working on the software model, an analysis of existing hardware and software was performed. All components of the developed software are fully described.

A convenient user interface has been developed. Instructions for working with software tools are provided.

The program can be used on PCs of IBM PC architecture with Windows XP/Vista/7/8/10/11.

The program was developed in the Java and Scala environment.

Keywords: computer science, IoT, server platforms, data processing.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, СИМВОЛІВ ТА СПЕЦІАЛЬНИХ ТЕРМІНІВ

AES (Advanced Encryption Standard)	симетричний шифр (стандарт).
DES (Data Encryption Standard)	симетричний шифр
PKI (Public Key Infrastructure)	інфраструктура відкритих ключів
IoT (Internet of Things)	концепція мережі, що складається із взаємозв'язаних фізичних пристроїв
SDL (Software Development Life Circle)	методика створення програмних продуктів
СКБД	система керування базами даних
HTTP	протокол передачі гіпертексту
ЕЦП	електронного цифрового підпису
ANSI/SPARC	американський національний інститут стандартів /стандарти і вимоги до планування комітету;- мова структурованих запитів
B2C (business-to-consumer)	("компанія-споживач)

ВСТУП

Актуальність теми. Інтернет речей (IoT) є одним із найдинамічніших і перспективних напрямків у розвитку інформаційно-комунікаційних технологій. Кількість пристроїв, підключених до мережі, постійно збільшується, що створює необхідність у сучасних підходах до проєктування високонавантажених серверних систем. IoT-платформи повинні забезпечувати аналіз даних із різних аспектів, що сприяє оптимізації виробничих і операційних процесів.

Задачі та проблеми, пов'язані з побудовою серверних систем для Інтернету речей, поділяються на дві категорії: загальні, які характерні для всіх систем роботи з великими даними, та специфічні, що виникають лише у сфері IoT. До загальних завдань належать розробка ефективних, відмовостійких, масштабованих і розподілених систем обробки даних. Специфічні ж аспекти включають забезпечення стабільного контролю та моніторингу підключених пристроїв.

У цьому дослідженні визначено вимоги до хмарних IoT-платформ, проведено аналіз існуючих рішень і запропоновано способи їх удосконалення. Розглянуто архітектурні підходи до обробки великих даних та методи розробки програмного забезпечення, які дозволяють створювати платформи, що відповідають як функціональним, так і нефункціональним вимогам. Окрім цього, проаналізовано можливості використання сторонніх рішень, вибрано оптимальний протокол та формат даних, що забезпечують максимальну ефективність роботи системи.

Основною метою є створення хмарної IoT-платформи з реалізацією ключових підсистем, таких як прийом, зберігання та обробка оперативних даних, забезпечення аутентифікації та безпеки, управління адміністративними даними й моніторинг. Окрему увагу приділено питанням інформаційної безпеки.

У сучасних умовах дистанційне управління пристроями та доступ до даних вимагають використання захищених схем шифрування й захисту, щоб мінімізувати ризики та унеможливити шкідливий вплив зловмисників.

Мета й завдання дослідження. Метою роботи є дослідження та розробка серверних рішень, що забезпечують функціонування систем Інтернету речей (IoT), з акцентом на створення підсистеми безпеки передачі даних.

Для досягнення поставленої мети сформульовано такі завдання:

– проведення огляду існуючих IoT-систем із підсистемами безпеки передачі даних;

– дослідження та розробка серверних рішень для підтримки роботи IoT;

– програмна реалізація системи управління з інтеграцією підсистеми безпеки передачі даних.

Об'єктом дослідження є процес створення серверного рішення для хмарної платформи IoT.

Предметом дослідження є методи реалізації серверних рішень для хмарних систем.

Методологія дослідження базується на використанні методів теорії кодування, захисту інформації, математичної статистики, а також методів розробки програмного забезпечення.

Наукова новизна отриманих результатів. У процесі виконання завдань, визначених цілями дослідження, були отримані такі результати:

– удосконалено серверне рішення для хмарної платформи Інтернету речей шляхом впровадження підсистеми безпеки передачі даних;

– здійснено огляд технологій зв’язку, що використовуються в системах IoT;

– розроблено вітчизняний продукт для управління хмарною платформою з інтегрованою підсистемою безпеки передачі даних, який має розширені можливості порівняно з існуючими аналогами.

Практична цінність отриманих результатів полягає в тому, що розроблені алгоритми забезпечують ефективне вирішення завдань із реалізації серверних рішень для управління хмарними платформами.

Достовірність наукових результатів підтверджується теоретичними викладеннями, даними комп'ютерного моделювання, коректними дослідженнями параметрів на функціонуючій обчислювальній мережі, а також відповідністю отриманих результатів аналогам, описаним у науковій літературі.

Таким чином, дослідження та програмна реалізація серверного рішення для хмарної платформи IoT з інтегрованою підсистемою безпеки передачі даних є актуальним завданням, яке вирішується в межах цієї магістерської роботи.

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

1.1. Призначення системи

У роботі досліджено створення платформи з використанням сучасних підходів до розробки розподілених систем, орієнтованих на прийом, зберігання та обробку великих обсягів даних. Для розробки платформи були використані актуальні бібліотеки та бази даних із відкритим кодом. Реалізація платформи проведена на кластері в хмарному хостингу.

Запропонована архітектура платформи забезпечує високу масштабованість і ефективність, а також дозволяє реалізовувати рішення для моніторингу мережі пристроїв та управління адміністративними даними. Ці рішення вирізняються розширеною функціональністю та підвищеним рівнем безпеки передачі даних порівняно з існуючими аналогами.

Метою роботи є створення серверного рішення для хмарної платформи Інтернету речей, яке відповідатиме підвищеним вимогам до безпеки передачі даних.

Система повинна задовольняти такі ключові вимоги:

- вирішувати основні задачі, характерні для таких платформ, включаючи зберігання та аналіз даних, шляхом реалізації підсистем, побудованих на загальноприйнятих підходах до розробки розподілених систем обробки великих даних;

- бути розробленою з урахуванням передового досвіду та сучасних підходів до архітектури програмного забезпечення, що забезпечують дотримання функціональних та нефункціональних вимог до системи;

- мати унікальну функціональність, яка відрізняє її від існуючих рішень, наприклад, надійна доставка повідомлень між пристроєм та адміністратором, а також засоби моніторингу пристроїв;

- включати інструменти для налаштування хмарного хостингу та управління групою серверів.

Розробка цієї платформи враховує актуальні виклики та тенденції у сфері Інтернету речей, а також забезпечує інноваційні рішення для підвищення ефективності та безпеки її використання.

1.2. Область застосування

У цій роботі досліджується розробка серверних рішень, які забезпечують функціонування систем Інтернету речей (IoT). Зокрема, розглядається побудова таких ключових підсистем:

- отримання та зберігання даних: система забезпечує прийом даних із мінімальними затримками та їх збереження в умовно постійному сховищі даних (наприклад, черзі повідомлень).

- обробка даних: здійснюється обробка потокових даних і виконання аналітичних запитів. Потокова обробка виконується між умовно постійним сховищем даних і базою даних (постійним сховищем). У випадках, коли обробка не потрібна, можливий простий перенос даних із черги повідомлень до бази даних.

- аутентифікація та безпека: система перевіряє автентичність відправників (пристроїв) і адміністраторів, а також забезпечує безпечну передачу даних між компонентами системи.

- передача адміністративних даних: передбачено обмін даними між пристроєм і адміністратором. Система передає дані про стан пристрою до адміністратора, а команди — від адміністратора до пристрою. Адміністративні дані зберігаються в проміжній підсистемі для забезпечення доставки навіть у разі тимчасової недоступності пристрою.

- моніторинг: здійснюється контроль стану пристроїв і аналіз надісланих ними даних відповідно до заданих правил.

Таким чином, дослідження та програмна реалізація систем управління серверними рішеннями IoT із інтегрованою підсистемою безпеки передачі даних є актуальним завданням, що вирішується в межах цієї магістерської роботи.

2 ПЕРЕГЛЯД АНАЛОГІЧНИХ ІСНУЮЧИХ СИСТЕМ

Серверні системи для Інтернету речей (IoT) відіграють важливу роль у забезпеченні обробки, безпеки, аналізу, збереження та управління даними, що генеруються численними пристроями та сенсорами, підключеними до Інтернету. Основні характеристики та властивості таких систем:

- масштабованість: Системи IoT повинні підтримувати динамічне масштабування, оскільки кількість підключених пристроїв постійно зростає. Вони мають обробляти великі обсяги даних і забезпечувати ефективне виконання обчислювальних завдань;

- обробка в реальному часі: Деякі IoT-додатки потребують обробки даних у реальному часі, наприклад, для виявлення аномалій або автоматичного надсилання сповіщень на основі даних із сенсорів;

- збереження даних: Надійне та довготривале збереження інформації з IoT-пристроїв необхідне для подальшого аналізу, створення звітів або архівування. Важливими є також швидкість доступу до даних і можливість пошуку;

- аналіз даних: Серверні системи мають підтримувати інструменти для аналітики даних, зокрема, машинне навчання та штучний інтелект, для виявлення закономірностей, прогнозування подій та прийняття рішень;

- моніторинг і управління: Ефективний моніторинг та управління IoT-пристроями включає віддалену конфігурацію, відправку команд та контроль стану пристроїв;

- безпека: захист від несанкціонованого доступу до даних і пристроїв є критично важливим. Системи мають забезпечувати шифрування, аутентифікацію користувачів та пристроїв, а також захист від кібератак.

- інтеграція: IoT-серверні системи повинні інтегруватися з іншими корпоративними системами, такими як CRM (системи управління взаємодією з клієнтами) чи ERP (системи управління ресурсами підприємства);

- підтримка різних протоколів: Через різноманіття IoT-пристроїв системи повинні працювати з різними комунікаційними протоколами (наприклад, MQTT, CoAP, HTTP);

- хмарні обчислення: Використання хмарних ресурсів дозволяє системам IoT забезпечувати гнучкість, масштабованість і економічність у зберіганні та обробці даних.

При виборі серверної системи для IoT важливо враховувати вимоги конкретного проєкту щодо масштабу, продуктивності, безпеки, інтеграції та функціональності. Це дозволяє вибрати рішення, яке найкраще відповідає потребам користувача.

2.1. Огляд існуючих систем

Інтернет речей (IoT) - це концепція мережі фізичних об'єктів («речей»), оснащених вбудованими технологіями, які забезпечують їх взаємодію між собою або із зовнішнім середовищем. Ця ідея передбачає організацію таких мереж як механізм, здатний трансформувати економічні та соціальні процеси, усуваючи потребу в участі людини в рутинних діях та операціях.

Індустрія Інтернету речей демонструє стабільний розвиток, і темпи її зростання лише прискорюються. Згідно з прогнозом дослідницько-консалтингової компанії Gartner, до 2025 року у світі буде функціонувати близько 6,4 мільярда IoT-пристроїв, що на 30% більше порівняно з 2015 роком, коли цей показник складав 4,9 мільярда. У грошовому вираженні галузь зросла з 1183 мільярдів доларів у 2018 році до очікуваних 1414 мільярдів доларів у 2023 році. Прогнозується, що до 2025 року кількість підключених пристроїв досягне 20,7 мільярда, а загальні витрати на них становитимуть близько 3 трильйонів доларів. При цьому приблизно половину з них складуть споживчі пристрої, а решта - пристрої, що використовуються підприємствами.

Компанія Cisco, найбільший у світі виробник мережевого обладнання, прогнозує ще більш значне зростання ринку IoT - до 14,4 трильйона доларів до

2026 року. Інтернет речей поступово інтегрується у всі сфери життя, забезпечуючи функціонування різноманітних систем.

Використання IoT охоплює як споживчі, так і промислові напрями:

- споживчі рішення: різноманітні домашні датчики, носима електроніка, системи «розумного будинку»;

- промислові рішення: управління та моніторинг виробничих процесів, впровадження «розумних» енергетичних систем, розвиток концепцій «розумних» міст, а також автономні транспортні засоби;

Таким чином, IoT не лише трансформує повсякденне життя, але й відіграє важливу роль у підвищенні ефективності та автоматизації ключових галузей промисловості.

Завдяки «речам» у рамках Інтернету речей (IoT) додана вартість значно збільшується через ряд переваг:

- поліпшення клієнтського досвіду: Інтерактивні системи та персоналізація послуг забезпечують більш точну відповідність потребам користувачів;

- скорочення часу до виходу на ринок (time-to-market): Швидше введення товарів у продаж завдяки автоматизації процесів і покращенню обміну інформацією;

- вдосконалення ланцюгів поставок і логістики: Завдяки IoT здійснюється реальний моніторинг товарів і процесів, що дозволяє зменшити витрати і підвищити ефективність;

- підвищення продуктивності праці працівників: За допомогою автоматизації та моніторингу знижується час на виконання рутинних завдань;

- більш ефективне використання коштів і зниження витрат: Завдяки оптимізації ресурсів, зменшенню помилок і витрат на утримання активів;

Розвиток IoT стимулює прогрес в багатьох галузях техніки, зокрема, сприяючи інноваціям у електронній промисловості, де активно розробляються нові компоненти — електронні плати, датчики, батареї тощо.

Специфіка портативних пристроїв накладає обмеження на програмне забезпечення, яке ними керує: обмежені обчислювальні потужності, вимоги до енергоспоживання, особливо для пристроїв, що працюють на батареях, вимагають економії ресурсів. Тому обробка даних часто здійснюється на стороні сервера.

Також серверні платформи, що обробляють дані з датчиків, повинні враховувати, що дані можуть містити помилки, а пристрої можуть виходити з ладу або повністю припиняти свою роботу, що вимагає надійних алгоритмів для забезпечення стійкості системи.

Безпека стає критично важливою в умовах постійного зростання кількості підключених пристроїв і обсягу даних, що ними генеруються та передаються. У світі, де пристрої збирають і передають комерційні, промислові чи особисті дані, існує високий ризик їх несанкціонованого доступу або використання. Ці дані можуть стосуватися як приватних осіб, так і компаній, тому захист їхньої конфіденційності, цілісності та доступності є основним завданням.

Для того, щоб забезпечити ефективне управління великою кількістю пристроїв та обробку великих даних (Big Data), необхідно використовувати новітні технології, зокрема в сфері хмарних обчислень, що дозволяють створювати масштабовані та відмовостійкі серверні рішення. Для таких систем важливими є такі властивості:

- горизонтальне масштабування: можливість нарощувати потужності за рахунок додавання нових серверів або обчислювальних ресурсів без порушення роботи системи;

- висока пропускна здатність: здатність обробляти великий потік даних без значних затримок;

- відмовостійкість: система повинна продовжувати працювати, навіть якщо окремі її частини виходять з ладу;

- короткий час відповіді: важливість швидкої обробки запитів і відгуків на події.

Наявність інструментів для аналізу даних є важливою для ефективного використання даних, які збираються з пристроїв IoT. Більшість таких пристроїв зараз виконують лише функцію збору та передачі даних, а не керування процесами. Однак обробка та інтелектуальний аналіз даних дозволяють значно покращити управління процесами.

Інтелектуальна обробка даних, за допомогою методів машинного навчання та штучного інтелекту, дозволяє автоматизувати прийняття рішень і значно зменшити потребу в людському втручанні. Це особливо корисно для оптимізації виробничих процесів, контролю якості продукції та моніторингу роботи інших систем.

Також варто зазначити, що методи обробки великих даних, застосовані в системах IoT, є універсальними і можуть бути використані в інших сферах, де збираються великі обсяги даних, зокрема:

- аналіз соціальних мереж: для виявлення трендів та прогнозування подій;
- аналіз поведінки користувачів веб-сайтів: для покращення користувацького досвіду та персоналізації контенту;
- аналіз стану інфраструктури ЦОД: для забезпечення надійності та безпеки інфраструктури;
- системи рекомендацій: для надання персоналізованих рекомендацій;
- націлювання реклами: для збільшення ефективності рекламних кампаній;
- веб-пошук: для покращення точності результатів пошуку на основі аналізу користувацьких запитів.

Ці технології виявляються ключовими для розвитку численних галузей і можуть допомогти автоматизувати безліч процесів, знижуючи потребу в людському втручанні та підвищуючи ефективність.

Платформи Інтернету речей (IoT) є основою для створення, моніторингу та управління підключеними пристроями, забезпечуючи зберігання, обробку та аналіз даних. Найбільші комерційні платформи, такі як Amazon Web Services IoT Platform, Google Cloud IoT Platform, ThingWorks, Oracle IoT та IBM IoT Platform,

пропонують широкий набір інструментів для інтеграції, зберігання та аналізу даних з пристроїв. Однак є суттєві обмеження в їх використанні, зокрема:

Закритий код. Ці платформи є комерційними і не надають доступу до вихідного коду, що ускладнює їх адаптацію та оптимізацію під конкретні потреби користувачів.

Прив'язка до постачальника: Використання таких платформ може створювати проблему прив'язки до одного постачальника, що обмежує гнучкість переходу на інші рішення без значних витрат часу та ресурсів.

Малий вибір рішень з відкритим кодом

Платформи з відкритим кодом в цій сфері практично не існують, хоча можливості для їх розробки є, оскільки з'являються стабільні та багатофункціональні бібліотеки з відкритими ліцензіями, які можуть стати основою для побудови таких рішень.

Незважаючи на ці обмеження, комерційні платформи IoT мають багато загальних функцій, таких як:

- зберігання даних: ефективне управління великими обсягами даних;

- аналіз даних: базові інструменти статистики та машинного навчання для обробки отриманих даних;

- візуалізація: можливості для відображення даних у вигляді графіків, таблиць або інших зручних форматів.

Однак, важливим напрямком є очищення даних, яке часто недостатньо добре реалізовано в існуючих рішеннях. Виявлення аномалій, яке є частиною цього процесу, є однією з найменш досліджених областей машинного навчання. Онлайн-виявлення аномалій, тобто здатність виявляти відхилення в даних в реальному часі, є важливим аспектом, який може значно покращити ефективність роботи платформи.

Ще одним важливим функціоналом є моніторинг пристроїв та передача адміністративних даних. Цей аспект включає передачу команд від адміністратора до пристроїв і зворотну передачу інформації про стан пристроїв. Існуючі

платформи іноді обмежені в реалізації таких можливостей, що робить їх менш ефективними для управління великими мережами пристроїв.

Рішення для розширення можливостей платформ:

- використання відкритих рішень та компонентів для інтеграції різних частин платформи та створення адаптованих рішень під конкретні потреби;

- розробка гнучких інструментів очищення та аналізу даних, зокрема для виявлення аномалій та покращення точності обробки;

- створення спеціалізованих інтерфейсів для моніторингу та управління групами пристроїв з урахуванням специфіки IoT;

Це дозволить отримати більш адаптовані, відкриті та гнучкі рішення для специфічних вимог IoT-проектів, а також забезпечить кращу інтеграцію з іншими системами та платформами.

Розроблене рішення, орієнтоване на універсальність, є важливим кроком для подолання обмежень існуючих платформ IoT. Завдяки використанню загальноприйнятих веб-протоколів (наприклад, HTTP, WebSocket, MQTT), замість специфічних для кожного виробника протоколів, система здатна забезпечити сумісність з широким спектром пристроїв і зменшити витрати на розробку програмного забезпечення. Ось кілька основних аспектів цієї універсальності:

Підтримка загальноприйнятих веб-протоколів:

- використання стандартних веб-протоколів дозволяє легко інтегрувати різноманітні пристрої без необхідності розробки спеціалізованого SDK для кожного пристрою;

- це значно знижує витрати на розробку клієнтських додатків для пристроїв, оскільки існує безліч бібліотек і фреймворків для роботи з такими протоколами.

Модульність і спеціалізація:

- система має бути гнучкою і розширюваною, з можливістю налаштування під конкретний сценарій використання. Наприклад, платформа повинна

підтримувати різні стратегії обробки даних для пристроїв різних типів і для різних сфер застосування;

- замість жорстко заданих ланцюгів обробки даних, рішення повинно надавати можливість налаштування обробки залежно від типу пристрою чи вимог конкретного бізнес-сценарію.

Це може включати:

- фільтрацію або агрегацію даних;

- обробку в реальному часі для виявлення аномалій;

- застосування специфічних алгоритмів для конкретних типів даних.

Розширення можливостей:

- оскільки різні сфери використання пристроїв вимагають різних підходів до обробки даних, система повинна надавати розширювану інтерфейсну можливість для додавання нових операцій обробки;

- це дозволяє платформі адаптуватися під різні галузі без необхідності кардинальних змін у архітектурі, просто додаючи нові модулі або конфігурації для специфічних вимог.

Універсальність у масштабуванні:

- платформа повинна бути здатна підтримувати масштабування не лише в контексті кількості пристроїв, але й в контексті різних типів даних, які можуть потребувати різних підходів до зберігання, обробки та аналізу;

Це включає:

- підтримку різних типів баз даних (реляційні, NoSQL, часова база даних) в залежності від характеру даних;

- використання гнучких інструментів аналізу та машинного навчання, що дозволяє адаптувати алгоритми до конкретних потреб.

Збереження даних для всіх типів пристроїв:

- платформа повинна забезпечити зберігання даних від усіх типів пристроїв, але обробка цих даних може відрізнятися в залежності від типу пристрою чи конкретної сфери використання. Це дозволяє системі залишатися

універсальною, одночасно забезпечуючи можливість спеціалізації обробки даних під конкретні потреби.

Вигоди для користувачів:

- зниження вартості розробки: Завдяки використанню стандартних веб-протоколів і підтримці широкого спектра пристроїв;

- гнучкість у налаштуванні: Платформа легко налаштовується для специфічних сценаріїв, що дозволяє підприємствам адаптувати її під свої вимоги без значних витрат на переписування коду;

- масштабованість: легке розширення платформи для обробки нових типів даних і підтримка великої кількості пристроїв;

- це дозволяє розробленому рішенню стати універсальним і потужним інструментом для управління IoT-екосистемами, з можливістю адаптації під будь-який бізнес-сценарій або технічну задачу.

Amazon Web Services (AWS IoT) пропонує широкий набір послуг для ефективного підключення, управління та аналізу даних від IoT-пристроїв. Ось огляд основних компонентів:

AWS IoT Core:

- це центральна служба для підключення IoT-пристроїв до хмари. Вона підтримує різноманітні протоколи, такі як MQTT, HTTP, і WebSocket. Це дозволяє пристроям відправляти дані в хмару та отримувати команди з хмари;

- функціональність: Пристрої можуть бути зареєстровані в IoT Core, де вони можуть безперешкодно взаємодіяти з серверами для передачі даних, управління та виконання різних завдань.

AWS IoT Device Management:

- ця служба призначена для управління та віддаленого налаштування IoT-пристроїв. Вона дозволяє відстежувати стан пристроїв, встановлювати правила і виконувати дії на основі змін в їх стані;

- функціональність: з її допомогою можна автоматизувати налаштування пристроїв, оновлювати їх програмне забезпечення та здійснювати моніторинг.

AWS IoT Analytics:

- ця служба дозволяє аналізувати дані, зібрані з пристроїв IoT, через створення пайплайнів обробки даних;

- функціональність: Ви можете виконувати попередню обробку, аналізувати дані, створювати звіти, а також візуалізувати дані для отримання цінних бізнес-інсайтів.

AWS IoT Events:

- інструмент для виявлення аномалій і подій в даних IoT;

- функціональність: Ви можете налаштувати правила для автоматичного спостереження за даними і реагувати на події в реальному часі, наприклад, генерувати сповіщення або активувати певні дії у разі виявлення аномалій.

AWS IoT Greengrass:

- це служба, яка дозволяє виконувати обчислення та обробку даних на "краю" (edge), безпосередньо поблизу IoT-пристроїв. Вона зменшує навантаження на хмарні ресурси і дозволяє працювати з даними локально;

- функціональність: Ви можете розгортати локальні застосунки та виконувати їх на пристроях Greengrass, що дозволяє прискорити обробку даних і знизити залежність від хмари.

AWS IoT Device Defender:

- пропонує засоби для забезпечення безпеки IoT-пристроїв;

- функціональність: Виявляє і запобігає загрозам безпеки, а також перевіряє налаштування безпеки пристроїв. Це допомагає забезпечити цілісність і конфіденційність даних.

AWS IoT SiteWise:

- цей інструмент спеціалізується на зборі, обробці і аналізі даних з промислових пристроїв і обладнання;

- функціональність: Ви можете створювати візуалізації даних, проводити аналіз і використовувати ці дані для оптимізації виробничих процесів та управління активами в промисловості.

Вигоди AWS IoT:

- широкий вибір протоколів і служб для різних типів пристроїв і сценаріїв використання;

- інтеграція з іншими сервісами AWS, що дозволяє будувати комплексні рішення на основі хмарної інфраструктури;

- масштабованість та гнучкість, що дозволяє використовувати ці сервіси для малих та великих IoT-рішень, від розумних будинків до промислових додатків;

- безпека та управління завдяки таким інструментам як AWS IoT Device Defender, що дозволяє забезпечити надійний захист.

AWS IoT є потужним інструментом для створення та управління IoT-екосистемами, оскільки надає все необхідне для підключення, обробки, зберігання та аналізу даних, а також для забезпечення безпеки пристроїв.

AWS IoT пропонує інтеграцію з іншими потужними сервісами AWS, що дозволяє створювати комплексні та масштабовані рішення для Інтернету речей. Ось кілька ключових аспектів цієї інтеграції:

Хмарне зберігання та інтеграція:

- Amazon S3: Для зберігання великих обсягів даних, зібраних з IoT-пристроїв. AWS IoT може автоматично зберігати дані в S3 для подальшого використання, обробки або архівування;

- Amazon Kinesis: Для обробки потокових даних в реальному часі. використовувується Kinesis для передачі та аналізу даних, які поступають від IoT-пристроїв, що дозволяє працювати з великими обсягами даних у режимі реального часу;

- AWS Lambda: Для виконання безсерверних обчислень у відповідь на події. За допомогою Lambda можна автоматизувати процеси, такі як обробка даних або виконання команд для пристроїв у реальному часі, без необхідності керувати серверною інфраструктурою.

Ці сервіси можуть бути з'єднані для створення потужних рішень, які збирають, зберігають, обробляють і аналізують дані IoT, автоматизуючи процеси та знижуючи витрати.

Безпека:

AWS IoT пропонує декілька ключових механізмів для забезпечення безпеки даних та пристроїв:

- аутентифікація пристроїв: AWS IoT підтримує механізми безпечної аутентифікації пристроїв, що гарантує, що тільки авторизовані пристрої можуть підключатися до платформи та взаємодіяти з даними;

- керування доступом: налаштовування права доступу до ресурсів IoT, визначаючи, хто і як може взаємодіяти з пристроями та даними в системі;

- шифрування: AWS IoT підтримує шифрування даних як при передачі, так і під час зберігання. Це забезпечує конфіденційність і захист від несанкціонованого доступу до чутливих даних.

Загальний огляд

AWS IoT - це не тільки зручна та потужна платформа для розробки проектів IoT, але й розширювана система, яка може підтримувати як малий бізнес, так і великі підприємства. Завдяки інтеграції з іншими сервісами AWS, таке рішення може бути адаптоване для різноманітних потреб - від простих пристроїв до складних промислових додатків.

Рішення AWS IoT допомагають:

- спростити розробку IoT-проектів, завдяки підтримці стандартних протоколів і можливості використання готових сервісів для зберігання та обробки даних;

- забезпечити високу безпеку з використанням передових методів шифрування та аутентифікації;

- аналізувати дані для отримання цінних інсайтів, що дозволяють оптимізувати бізнес-процеси, підвищити ефективність і зменшити витрати.

Таким чином, AWS IoT дозволяє створювати інтелектуальні, масштабовані й безпечні IoT-рішення, що ідеально підходять для різних галузей і сценаріїв використання.

Microsoft Azure IoT - це комплексне рішення для створення, впровадження та масштабування проектів Інтернету речей, яке надає набір інструментів для підключення пристроїв, збору, обробки та аналізу даних, а також забезпечення безпеки.

Компоненти та можливості Azure IoT

Azure IoT Hub:

- це центральна платформа для підключення, моніторингу та управління IoT-пристроями;

- підтримка протоколів: IoT Hub підтримує різні протоколи для зв'язку з пристроями, зокрема MQTT, HTTP, та AMQP;

- функціонал: Забезпечує реєстрацію пристроїв, керування ними, моніторинг їхнього стану та передачу даних до хмари для подальшої обробки та зберігання.

Azure IoT Central:

- це високорівнева платформа, яка дозволяє швидко розробляти IoT-рішення без необхідності глибоких знань в програмуванні.

- готові рішення: Azure IoT Central надає інтерфейси для створення додатків, що включають вбудовані інструменти для моніторингу та аналізу даних, що робить її ідеальною для бізнесів, які хочуть швидко розпочати впровадження IoT.

Azure Stream Analytics:

- це інструмент для аналізу потокових даних, що надходять від IoT-пристроїв;

- обробка в реальному часі: налаштування правил, фільтри та агрегацію даних, а також автоматично передавати оброблену інформацію в інші Azure-сервіси для подальшого використання чи зберігання.

Azure IoT Edge:

- Azure IoT Edge дозволяє виконувати обчислення та обробку даних на краю мережі (edge), безпосередньо на пристроях або поблизу них;

- локальне виконання: Це зменшує навантаження на центральні хмарні сервіси і дозволяє працювати з даними на місці, що зменшує затримки та потребу в великій пропускній здатності для передавання даних.

Azure Time Series Insights:

- спеціалізований інструмент для аналізу та візуалізації даних з часовими рядами, що збираються від IoT-пристроїв;

- інтерфейси для візуалізації: Дозволяє користувачам створювати звіти та аналізи для виявлення трендів і аномалій у даних, що можуть бути корисні для прогностичної аналітики та оптимізації бізнес-процесів.

Azure IoT Solutions:

- Microsoft пропонує готові рішення для різних галузей, включаючи медицину, виробництво, транспорт та інші.

- ці рішення включають готові сценарії, візуалізації, шаблони та компоненти, які значно полегшують розробку та впровадження проектів IoT.

Azure IoT Security

Безпека є важливою частиною Azure IoT. Вона включає в себе різні механізми для забезпечення захисту даних і пристроїв:

- аутентифікація пристроїв для підтвердження їхнього достовірного підключення;

- шифрування даних для забезпечення конфіденційності переданої інформації;

- керування доступом для налаштування прав доступу до пристроїв та даних.

Azure IoT — це потужна і масштабована платформа, що охоплює всі аспекти IoT-проектів: від підключення пристроїв до їхнього управління, зберігання даних і аналізу, а також забезпечення високої безпеки. Azure

пропонує як розробникам, так і бізнесам широкі можливості для створення інтелектуальних і ефективних IoT-рішень в різних сферах.

Інтеграція з іншими сервісами Azure

Azure IoT легко поєднується з різними сервісами Azure, такими як Azure Machine Learning, Azure Functions, Azure Logic Apps та іншими.

Підтримка широкого спектру пристроїв і платформ

Платформа Azure IoT забезпечує роботу з різноманітними пристроями та операційними системами, включаючи мікроконтролери, Linux, Windows, а також інтеграцію з обліковими записами в інших хмарних сервісах.

Глобальне масштабування

Azure IoT пропонує гнучкі можливості масштабування — від кількох пристроїв до мільйонів, забезпечуючи стабільну та доступну інфраструктуру для реалізації IoT-проектів.

Azure IoT — це багатофункціональна та надійна платформа, яка надає широкий спектр інструментів і можливостей для створення успішних IoT-рішень.

Google Cloud IoT

Google Cloud пропонує інструменти для підключення, обробки та аналізу даних IoT. До них входять Cloud IoT Core для управління пристроями та Cloud Pub/Sub для передачі даних.

Google Cloud IoT - це комплекс сервісів і засобів, розроблених для створення й впровадження IoT-рішень на платформі Google Cloud. Нижче наведено ключові компоненти.

Cloud IoT Core - це основний сервіс Google для підключення та управління IoT-пристроями. Він підтримує протоколи MQTT і HTTP, дозволяючи легко підключати пристрої, управляти ними, а також створювати правила для передачі даних у інші сервіси Google Cloud.

Cloud IoT Device Manager

Cloud IoT Device Manager дозволяє ефективно керувати IoT-пристроями та їх конфігурацією. Ви можете створювати, управляти та відстежувати статус усіх підключених пристроїв.

Cloud Pub/Sub

Cloud Pub/Sub забезпечує передачу даних з IoT-пристроїв до різних сервісів Google Cloud для подальшої обробки й аналізу. Ця служба дозволяє налаштовувати правила маршрутизації для обробки даних і створення звітів.
Cloud Dataflow.

Cloud Dataflow — це інструмент для обробки потокових даних, зокрема з IoT-пристроїв, у режимі реального часу. Він дозволяє виконувати аналіз, обробку та агрегування даних швидко та ефективно.

Cloud Bigtable - розподілена база даних, оптимізована для зберігання та аналізу великих обсягів IoT-даних. Вона забезпечує високу продуктивність і швидке виконання запитів до даних.

Cloud Machine Learning Engine пропонує засоби для створення та впровадження моделей машинного навчання, що дозволяє аналізувати дані IoT, виявляти закономірності та прогнозувати події.

Інтеграція з іншими сервісами Google Cloud

Google Cloud IoT легко поєднується з іншими інструментами Google Cloud, такими як Google Cloud Storage, Google Cloud Functions, BigQuery тощо, забезпечуючи комплексний підхід до обробки IoT-даних.

Безпека

Google пропонує потужні засоби для захисту IoT-пристроїв і даних, включаючи механізми аутентифікації, шифрування та контроль доступу.

Глобальні можливості масштабування

Google Cloud IoT забезпечує масштабовану інфраструктуру, яка дозволяє ефективно розширювати ваш проект від кількох до мільйонів IoT-пристроїв.

Google Cloud IoT - це потужна і гнучка платформа з багатим набором інструментів, розроблених для успішної реалізації IoT-рішень.

IBM Watson IoT

IBM Watson IoT пропонує інструменти для створення, управління та аналізу IoT-рішень. Платформа забезпечує інтеграцію підключених пристроїв, обробку даних та створення аналітичних моделей.

Watson IoT Platform

Центральний компонент платформи, який дозволяє підключати, реєструвати та керувати IoT-пристроями. Ви можете відстежувати стан пристроїв, надсилати команди та створювати правила для обробки даних.

Watson IoT Platform Analytics

Цей сервіс забезпечує аналіз даних з IoT-пристроїв, використовуючи технології машинного навчання та аналіз часових рядів. Можливості включають прогнозування, виявлення аномалій і оптимізацію роботи пристроїв.

Watson IoT Platform Operations Console

Зручний веб-інтерфейс для управління IoT-пристроями та їхніми даними. Платформа дозволяє створювати візуалізації, налаштовувати сповіщення та встановлювати правила для автоматизації процесів.

IBM Watson IoT - це комплексне рішення для збору, обробки й управління даними IoT, яке допомагає оптимізувати роботу вашого обладнання та підвищити ефективність проектів.

Watson IoT Platform Blockchain

IBM Watson IoT може інтегруватися з технологією блокчейн для забезпечення безпеки та надійності даних з IoT-пристроїв.

Це особливо корисно в галузях, де важлива доказова база даних.

Watson IoT Platform Edge Analytics

Watson IoT Platform Edge Analytics забезпечує можливість обробки даних на краю (edge) безпосередньо біля IoT-пристроїв. Ця функція дозволяє

створювати локальні застосунки для аналізу даних, що зменшує залежність від передачі інформації в хмару.

Watson IoT Platform Device Lifecycle Management

Цей сервіс пропонує інструменти для управління повним життєвим циклом IoT-пристроїв. Ви можете реєструвати пристрої, налаштовувати їх, а також ефективно керувати оновленнями програмного забезпечення.

Watson IoT Continuous Engineering

IBM пропонує інструменти для інженерії продукту, які ідеально підходять для розробки IoT-пристроїв. Ці рішення сприяють ефективності розробки та впровадження нових функцій.

Інтеграція з іншими сервісами IBM

Watson IoT легко інтегрується з іншими продуктами IBM, такими як Watson AI для аналітики та штучного інтелекту, IBM Cloud для хмарних сервісів і IBM Maximo для управління активами.

Cisco IoT Cloud Connect

Cisco IoT Cloud Connect пропонує комплексні рішення для підключення та управління IoT-пристроями. Це включає мережеве обладнання, яке забезпечує стабільність підключення, інструменти для аналізу даних і рішення для забезпечення безпеки IoT-мереж.

Cisco IoT - це платформа, яка поєднує потужну мережеву інфраструктуру з інструментами для управління IoT-проектами та їхньою аналітикою.

Платформа IoT від MathWorks

ThingSpeak - це безкоштовна платформа для IoT, створена MathWorks (розробник MATLAB). Вона забезпечує збір, збереження, візуалізацію та аналіз даних з IoT-пристроїв, а також створення сповіщень і інтеграцію з іншими сервісами.

Збір даних IoT

ThingSpeak дозволяє підключати IoT-пристрої через різні протоколи, такі як MQTT, HTTP або ThingHTTP, для передачі даних на платформу. Вона

підтримує інтеграцію з мікроконтролерами, полегшуючи збирання даних з ваших пристроїв.

Збереження та візуалізація даних

ThingSpeak зберігає отримані дані та пропонує гнучкі інструменти для їх візуалізації. Ви можете створювати графіки, таблиці й настроювані панелі для зручного перегляду інформації в реальному часі.

Аналіз даних

ThingSpeak інтегрується з MATLAB, надаючи інструменти для аналізу та обробки даних. Ви можете створювати аналітичні скрипти для виявлення закономірностей, прогнозування та застосування інших методів обробки даних.

Сповіщення та інтеграція

Платформа підтримує налаштування сповіщень на основі виконання певних умов. Ви також можете інтегрувати ThingSpeak з іншими веб-сервісами та платформами для передачі чи обробки даних у зовнішніх додатках.

Відкритий доступ до даних

ThingSpeak дозволяє керувати правами доступу до інформації. Ви можете визначати, хто має можливість читати або записувати дані, а також розміщувати їх на публічних каналах для загального доступу.

Безкоштовний план та публічні канали

ThingSpeak пропонує безкоштовний план, який підходить для початкового використання або навчальних цілей. Публічні канали надають змогу ділитися даними з іншими користувачами.

ThingSpeak - це ефективне рішення для малих і середніх IoT-проектів, яке поєднує простоту використання, потужність MATLAB і гнучкість інтеграції.

Легка інтеграція з MATLAB і Simulink

ThingSpeak має тісну інтеграцію з MATLAB і Simulink, що робить його ідеальним інструментом для користувачів, які вже працюють із продуктами MathWorks. Ця взаємодія спрощує аналіз, візуалізацію даних і моделювання IoT-систем.

ThingSpeak - це зручна платформа для створення простих IoT-проектів, збору та візуалізації даних, а також для навчальних і демонстраційних завдань.

PlatformIO

PlatformIO - це багатофункціональне інтегроване середовище розробки (IDE) та платформа для роботи з вбудованими системами та IoT-пристроями. Воно підтримує різноманітні мікроконтролери, платформи й мови програмування, забезпечуючи гнучкість для розробників IoT-проектів.

Підтримка різних мікроконтролерів та платформ

PlatformIO працює з широким спектром мікроконтролерів і платформ, включаючи:

- Arduino;
- ESP8266ESP32;
- Raspberry Pi;
- STM32;
- AVR;

Ця гнучкість дозволяє інтегрувати різні мікроконтролери в межах одного проекту.

Компіляція та завантаження коду

PlatformIO забезпечує інструменти для:

- компіляції коду для різних мікроконтролерів;
- завантаження прошивки на пристрої через USB або мережу;
- зідлагодження для швидкого виявлення та виправлення помилок у коді.

Підтримка мов програмування

PlatformIO підтримує популярні мови програмування, серед яких:

- C/C++ — основні мови для розробки мікроконтролерів;
- Python — для швидкої розробки й автоматизації;
- Rust — для високопродуктивних рішень.

PlatformIO — це потужний інструмент для розробників IoT та вбудованих систем, який поєднує широкий функціонал, підтримку сучасних технологій і зручність у роботі.

Вибір мови, яка найкраще відповідає потребам та навичкам.

Бібліотеки та ресурси

PlatformIO пропонує доступ до великої кількості бібліотек і ресурсів для розробки вбудованих систем. Завдяки вбудованому менеджеру пакетів, встановлення й оновлення бібліотек стає простим і швидким процесом.

Відлагодження коду

За допомогою інтеграції з відлагоджувачем GDB, PlatformIO дозволяє:

- встановлювати точки зупинки;
- аналізувати значення змінних;
- виконувати покрокове відлагодження коду в режимі реального часу.

Це спрощує пошук і виправлення помилок під час розробки.

Інтегровані засоби роботи з версіями

PlatformIO підтримує інтеграцію із системами контролю версій, такими як Git. Це дозволяє:

- відслідковувати історію змін у проекті;
- працювати спільно над проектами з іншими розробниками;
- забезпечувати безпеку даних і організованість проекту.

Підтримка платформ IoT і хмарних сервісів

PlatformIO забезпечує розробку для платформ Інтернету речей (IoT) з можливістю інтеграції з хмарними сервісами, такими як:

- Amazon Web Services (AWS);
- Google Cloud;
- Microsoft Azure.

Це відкриває додаткові можливості для створення масштабованих і надійних IoT-рішень.

Кросплатформеність та відкритий код

PlatformIO доступний для основних операційних систем:

- Windows;

- macOS;

- Linux;

Як проект з відкритим вихідним кодом, він дозволяє спільноті розробників вносити поліпшення та додавати нові функції.

PlatformIO: Ідеальний інструмент для IoT-розробників

PlatformIO спрощує процеси розробки, тестування й впровадження коду для різних мікроконтролерів і платформ.

Вибір платформи

Обираючи платформу для IoT-проекту, враховуйте:

- технічні вимоги проекту;

- ваші знання та досвід;

- доступні ресурси для розробки та підтримки.

Перед прийняттям остаточного рішення проведіть аналіз і тестування, щоб обрати оптимальну платформу для реалізації вашого проекту IoT.

2.2 Обґрунтування вибору методів розробки

Для розробки та реалізації платформи Інтернету речей були обрані та обґрунтовані наступні технології, інструменти та мови програмування:

Архітектурні рішення:

- розроблено розподілену мікросервісну архітектуру, яка забезпечує високу продуктивність та масштабованість;

- архітектура дозволяє ефективно обслуговувати мережі будь-якого розміру, підвищувати продуктивність системи та вдосконалювати підсистему аналізу даних;

- компоненти системи впроваджено з урахуванням оптимізації витрат на апаратну інфраструктуру.

Використані сучасні технології та відкриті рішення:

- застосовано системи, протоколи, формати даних, бібліотеки та бази даних з відкритим кодом, що спрощує розширення та забезпечує економічну ефективність;

- детально описано вибір і реалізацію архітектурних рішень: поділ на функціональні модулі, введення/виведення даних, автентифікацію та їх вплив на масштабованість платформи.

Гнучкість та розширюваність:

- платформа адаптована для обробки даних від різних пристроїв і сенсорів, з можливістю налаштування ланцюжків обробки;

- протоколи й формати даних обрано з урахуванням ефективності, зручності використання та простоти масштабування.

Інструменти та мови програмування:

- практична реалізація виконана на **Java (версія 8.1)** та **Scala**.

Використані бібліотеки та технології:

- Undertow: легковаговий веб-сервер на Java з неблокувальним введенням/виведенням. Використаний для реалізації HTTP (REST API) та WebSocket серверів.

Kafka clients: клієнтські інтерфейси для взаємодії з кластером Apache Kafka.

Apache Spark:

Spark-Cassandra Connector: інтеграція з Apache Cassandra для розподіленої обробки даних;

MongoDB Async Driver: асинхронний драйвер для MongoDB з використанням Java NIO.

JSON Web Tokens: для створення та перевірки веб-токенів.

Apache Commons Codec: використовується для кодування (наприклад, Base64).

SLF4J: API для ведення журналів.

Logback: реалізація SLF4J для логування.

JUnit: бібліотека для модульного тестування.

Цей набір технологій забезпечує високу продуктивність, гнучкість та ефективність платформи, що дозволяє реалізувати проекти різного масштабу з мінімальними витратами.

2.3 Розгорнута постановка завдання

Згідно з технічним завданням для магістерської роботи, потрібно розробити програмне забезпечення для серверної системи управління Інтернетом речей (IoT). Під час виконання роботи необхідно здійснити наступні етапи:

а) провести аналіз існуючих аналогічних систем для виявлення їх позитивних та негативних характеристик, а також врахувати результати цього аналізу у подальшому розробленні;

б) обрати та обґрунтувати методику побудови системи для контролю роботи технологічного обладнання на виробництві в автоматизованому режимі, розробити функціональну та структурну схеми системи;

в) розробити програмне забезпечення системи, що дозволить реалізувати поставлену задачу, а також побудувати блок-схеми алгоритмів програми та підпрограм;

г) організувати інтерфейс користувача для формування та виведення повідомлень на екран ЕОМ про некоректні дії користувача та нестандартні ситуації в роботі технологічного обладнання;

д) розробити рекомендації щодо організаційних і методичних заходів, що забезпечать успішне впровадження системи в промислову експлуатацію та її подальшу ефективну роботу;

е) провести розрахунки для визначення економічної ефективності розробленої системи;

ж) розробити заходи щодо охорони праці під час впровадження та експлуатації системи, а також заходи з цивільного захисту;

з) сформулювати висновки щодо виконаного обсягу робіт і отриманих результатів.

3 ОПИС І ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ

3.1 Опис функціонування системи

Розглянемо основні компоненти системи, потоки даних між ними, а також її входи та виходи. На рисунку 3.1 представлені деякі зовнішні системи та бібліотеки, такі як бази даних, які використовуються в певних компонентах системи. Детальніше обґрунтування вибору цих компонентів буде наведено в наступних розділах.

На вході системи (блок "Джерела даних") знаходяться різні пристрої, наприклад, датчики температури, тиску, якості повітря та інші. Ці пристрої передають дані на набір вхідних серверів (блок "Вхідні сервери"). Крім основних даних, пристрої також передають інформацію для автентифікації та свої ідентифікаційні дані (наприклад, унікальний ідентифікатор пристрою, який дозволяє визначити тип пристрою).

Внутрішні сервери здійснюють перевірку автентичності інформації. Якщо вона є некоректною (наприклад, відправник невідомий), дані відхиляються. В залежності від інформації про пристрій, дані завантажуються на зовнішні сервери: визначається тип повідомлення та направляється в певну чергу повідомлень за допомогою технології "Apache Kafka" (блок "Apache Kafka").

Внутрішні сервери також перевіряють цілісність отриманих даних, проводячи їх валідацію, перед тим як передати до черги повідомлень.

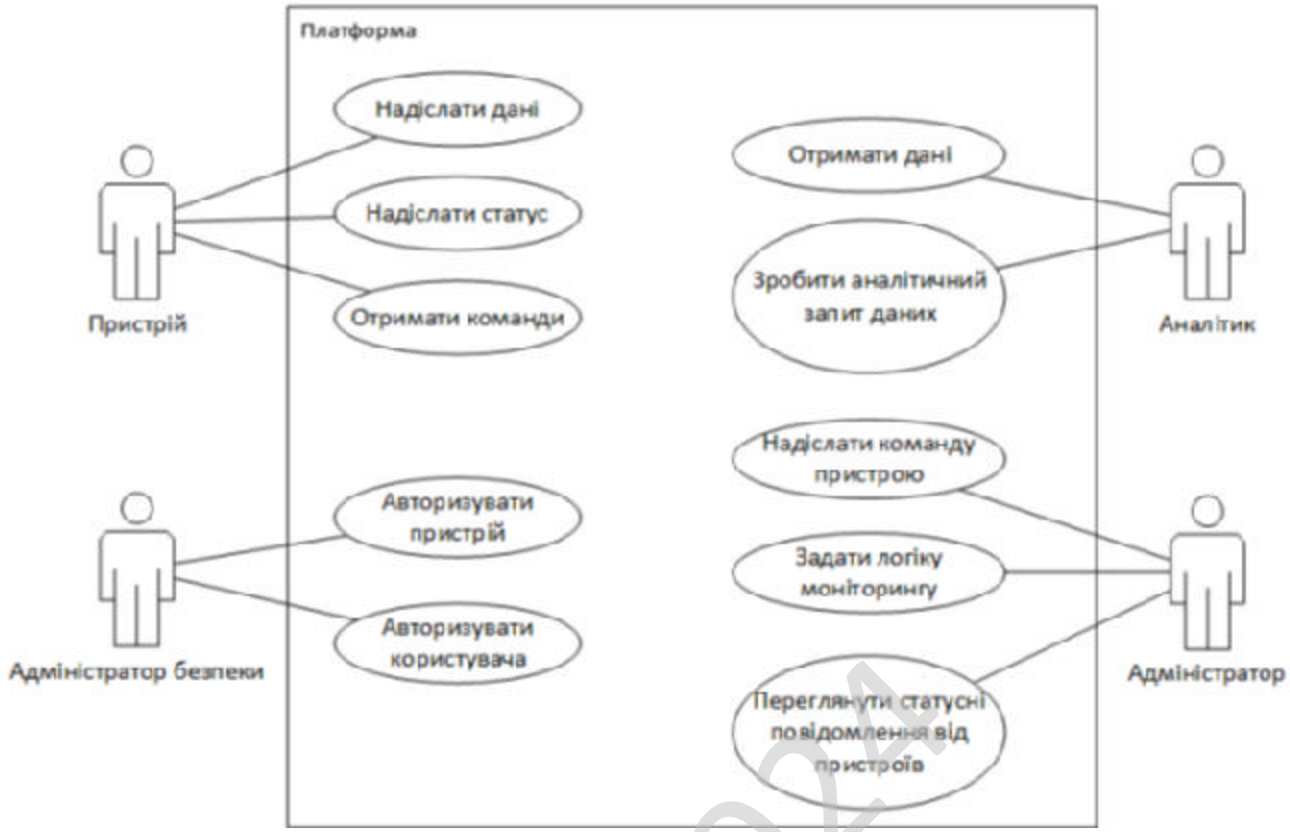

Рисунок 3.1 - Взаємодія із загальною функціональністю платформи

3.2 Розробка структурної схеми

Структурна схема системи є сукупністю об'єктів і їх частин, а також взаємозв'язків між ними. Основне призначення структурної схеми - наочне відображення складових частин розроблюваної системи, її основних блоків, вузлів і взаємодій між ними.

У системі можна виділити чотири основні групи користувачів. Перша група - це пристрої або датчики, які надсилають дані до системи для обробки та зберігання. Вони також відправляють повідомлення про свій статус, які потім отримують адміністратори, та приймають команди від них.

Адміністратори займаються налаштуванням логіки моніторингу та вводять дані для автентифікації пристроїв і користувачів. Вони мають змогу визначати права доступу для інших користувачів системи, вказуючи, які пристрої вони можуть контролювати чи отримувати з них дані. Аналітики, в

свою чергу, можуть запитувати готові дані, які вже зберігаються та попередньо обробляються в системі, або формувати аналітичні запити для виконання обчислень на основі існуючих даних.

Структурна схема системи IoT зображена на рисунку 3.2 .

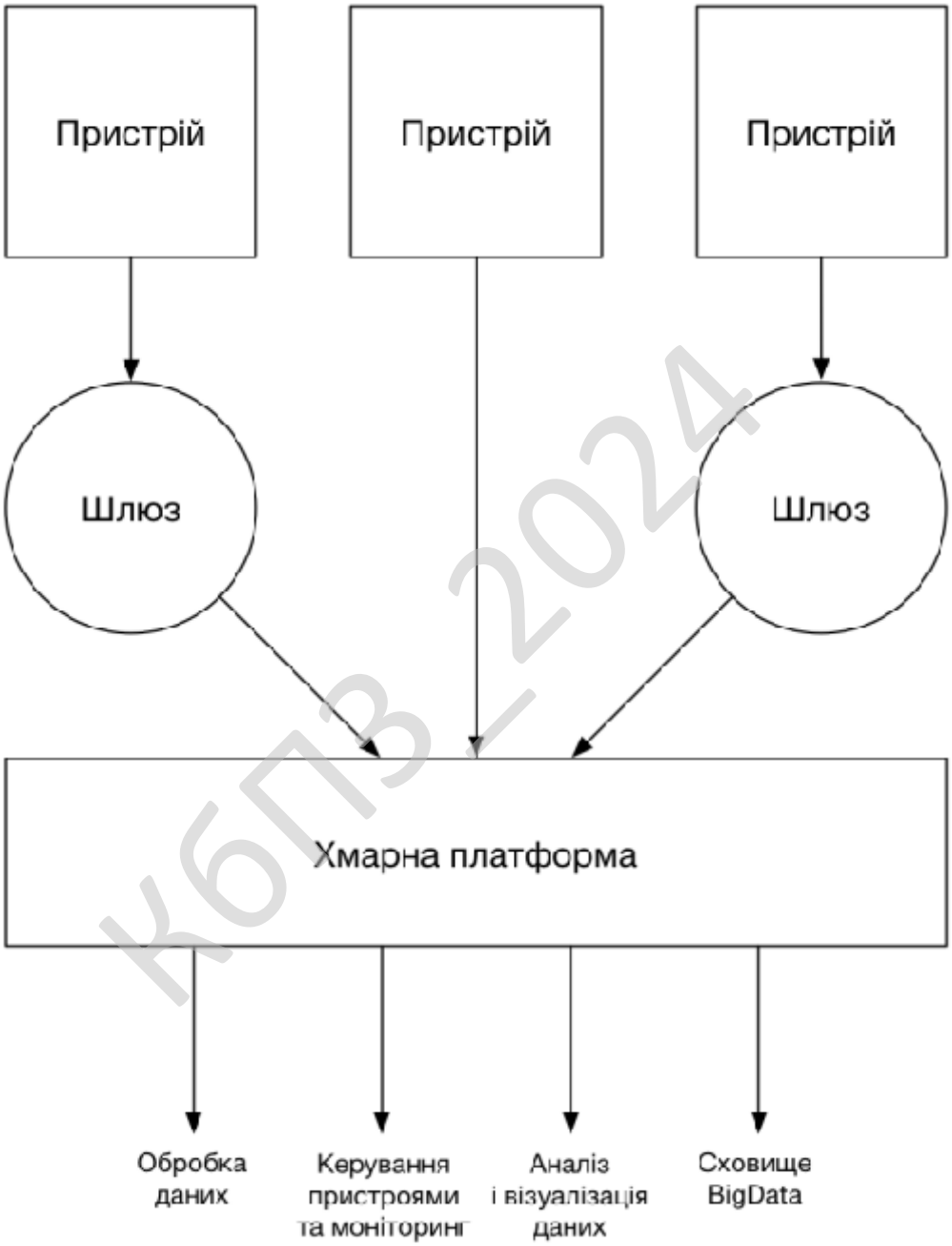

Рисунок 3.2 – Структурна схема систем IoT

3.3 Розробка функціональної схеми

Згідно з блок-схемою, є кілька пристроїв на різних платформах, що передають дані на сервер. Деякі пристрої оснащені вбудованими картами, які дозволяють безпосередньо передавати дані на сервер.

Наприклад, пристрій може бути підключений до Інтернету через Ethernet-кабель або бездротовий доступ через Wi-Fi. У випадку, коли кілька пристроїв розташовані на одній території або в межах однієї будівлі, вони можуть бути підключені до шлюзу. Кількість шлюзів значно менша за кількість датчиків, і вони забезпечують передачу даних на сервер через Інтернет.

Це рішення має кілька переваг і недоліків:

- економічний аспект: складне мережеве обладнання збільшує вартість пристроїв.

- енергоспоживання: спеціалізовані протоколи, як-от Bluetooth Low Energy, сприяють більш ефективному використанню енергії, що важливо для пристроїв, які працюють від автономного живлення.

Програмне забезпечення. Пристрої можуть передавати дані на шлюз у двійковому форматі, що полегшує розробку та зменшує навантаження на процесор. Шлюз перетворює ці дані в стандартний формат (наприклад, JSON або Protobuf) і передає їх на сервер. При цьому мережа між пристроєм і шлюзом може бути безпечною, а сам шлюз здійснює шифрування та передає дані через захищений протокол, наприклад, SSL.

Після того, як дані потрапляють на сервер, вони проходять попередню обробку та зберігаються, а потім доступні для подальшого аналізу.

Ця робота зосереджена на розробці серверної платформи. Питання розробки програмного забезпечення для пристроїв і шлюзів (прошивок) виходять за рамки цього проекту. У ньому будуть розглянуті лише основні аспекти реалізації клієнтської частини платформи.

Також важливо визначити нефункціональні вимоги до платформи:

- **адаптивність**: збільшення обчислювальних ресурсів повинно лінійно збільшувати пропускну здатність системи. Крім того, для підтримки систем великих даних та хмарних додатків система повинна мати горизонтальне масштабування, що дозволяє збільшувати продуктивність через додавання нових вузлів до хмарного центру обробки даних. Горизонтальне масштабування є економічно вигіднішим, ніж вертикальне;

- **відмовостійкість**: система повинна мати резервні компоненти (реплікацію) і бути здатною реагувати на збої та перевантаження без повної зупинки служби. Це особливо важливо для частин системи, що забезпечують прийом і зберігання даних. Якщо сервер недоступний, пристрої не можуть зберігати свої дані протягом тривалого часу і потім передавати їх на сервер через обмежений обсяг пам'яті;

- **продуктивність**: система повинна мати високу ефективність для забезпечення пропускної здатності на рівні сотень тисяч запитів на секунду та більше для підсистеми прийому та обробки даних. Аналітичні запити також мають оброблятися швидко, не більше ніж за кілька хвилин.

Функціональна схема розробленої системи зображена на рисунку 3.3.

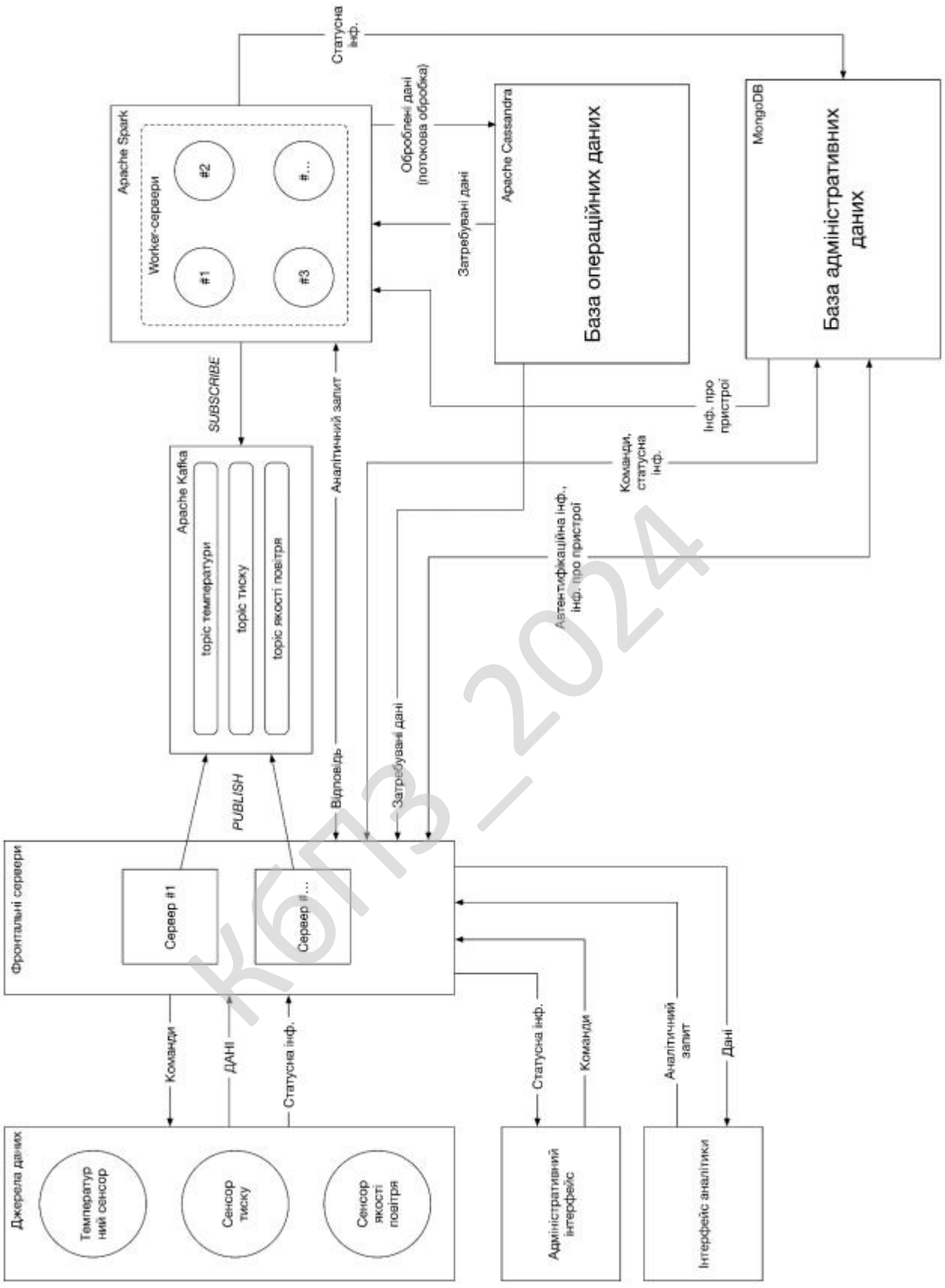

Рисунок 3.3 - Функціональна схема системи

3.4 Розробка діаграми процесів

Діаграма процесів розробленої системи, представлена на рисунку 3.4, показує, як відбувається взаємодія в межах системи. Використовується модель проектування, яка графічно зображає «потоки» даних в інформаційній системі.

Діаграма взаємодії процесів слугує для візуалізації процесів обробки даних (структурне проектування). Для розробника звичним є створення діаграми взаємодії процесів на рівні контексту, що дозволяє показати взаємодію основних компонентів системи. Ця діаграма, на наступних етапах, уточнюється через деталізацію процесів та потоків даних, що дозволяє точніше зобразити функціонування розроблюваної системи.

Використовуючи сучасні методи проектування, була створена система управління розумним будинком, що включає підсистему забезпечення безпеки передачі управляючих сигналів від смартфона до мікроконтролера. Ця підсистема забезпечує захищену передачу даних при віддаленому управлінні системою через Інтернет.

Таким чином, після розгляду опису системи, її структурної та функціональної схем, а також діаграми взаємодії процесів, ми перейдемо до опису блок-схем основної програми та підпрограм, які використовуються для реалізації цієї системи.

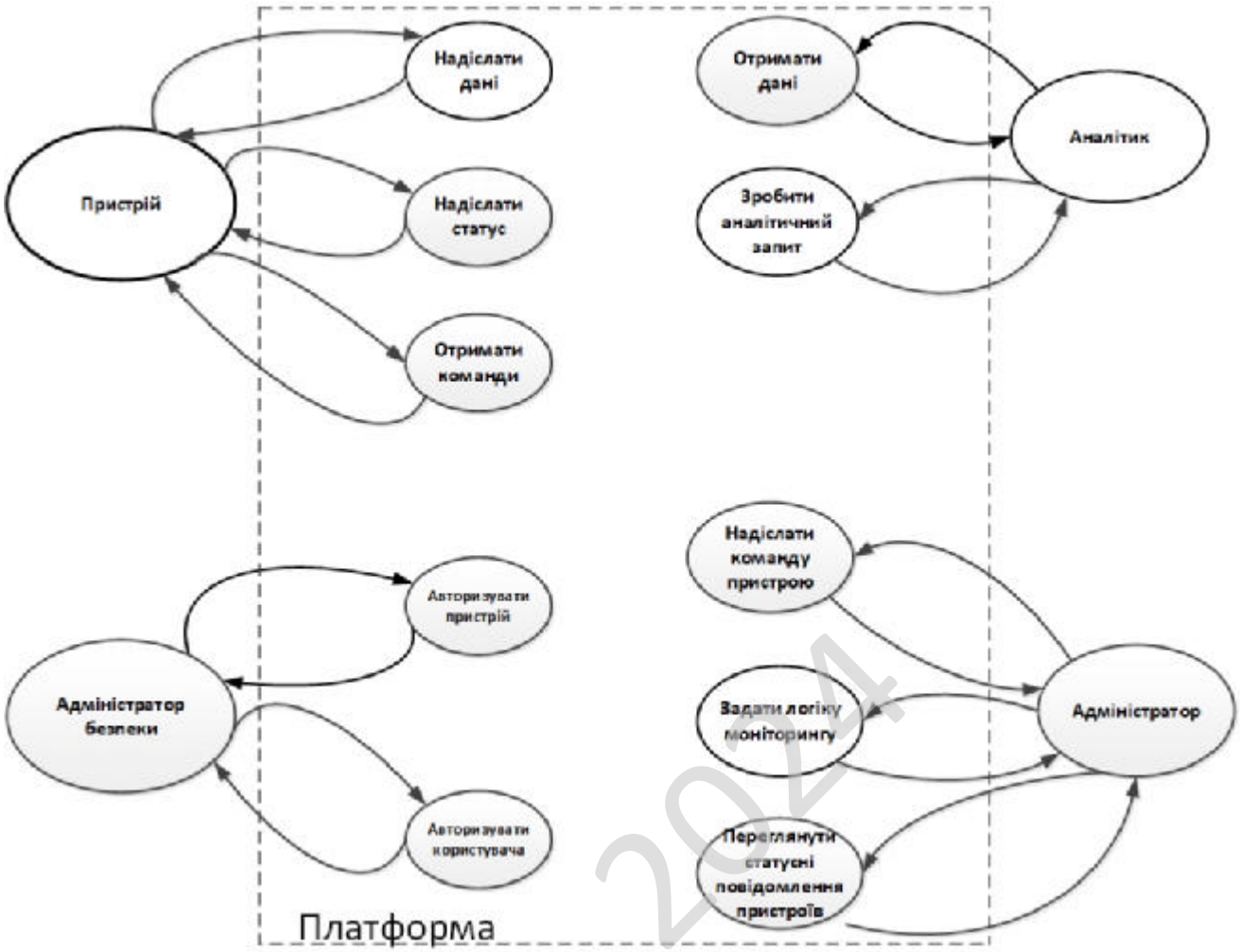

Рисунок 3.4 – Діаграма процесів системи

4 РЕАЛІЗАЦІЯ РОБОТИ. РОЗРАХУНКИ ТА ЕКСПЕРИМЕНТАЛЬНІ ДАНІ, ЩО ПІДТВЕРДЖУЮТЬ ВІРНІСТЬ ПРОЕКТНИХ І ПРОГРАМНИХ РІШЕНЬ

Опишемо основні компоненти системи, потоки даних між ними, а також її входи та виходи.

На вході системи (блок «Джерела даних») ми маємо різні пристрої, такі як датчик температури, датчик тиску, датчик якості повітря або будь-які інші. Вони надсилають дані на набір передніх серверів (блок «Вхідні сервери»). Разом із даними пристрої передають інформацію автентифікації та інформацію про себе (наприклад, ідентифікатор пристрою, за яким можна визначити тип пристрою).

Внутрішні сервери виконують перевірку інформації автентифікації. Якщо воно неправильне (тобто відправник невідомий), дані відхиляються. Відповідно до інформації пристрою відбувається завантаження даних на зовнішні сервери: визначається тип повідомлення та надсилається у відповідну тему в черзі повідомлень (блок «Apache Kafka»). Внутрішні сервери також перевіряють цілісність даних (валідують).

Черга повідомлень реалізує шаблон дизайну публікації-підписки. Інтерфейсні сервери публікують дані в Apache Kafka. На стороні підписки є система обробки даних – Apache Spark. Здійснює первинну обробку даних, які згодом зберігаються в оперативній базі даних. Обробка даних на цьому етапі може бути чисто потоковою або мікропакетною (мікропакетною). Мікропакетна обробка може емулювати потокову обробку, таким чином дозволяючи застосовувати інструменти та логіку пакетної обробки. Також мікропакетна обробка природно відповідає логіці агрегації та віконної обробки, коли, наприклад, необхідно зібрати певну кількість показань за хвилину, знайти їх середнє значення та зберегти в базі даних. Ви також можете застосувати фільтрацію (або виявлення аномалії) на рівні мікропакета.

База даних, яка містить оперативні дані, забезпечує постійне зберігання оброблених даних. З точки зору продуктивності, це сховище має бути оптимізоване для запису, оскільки операцій читання, очевидно, значно менше. Однак черга повідомлень служить буфером у випадку, якщо операційна база даних не в змозі (наприклад, під час стрибка навантаження) зберігати дані зі швидкістю надходження. Слід зазначити, що ця база даних не вимагає таких значних гарантій, як ті, що надаються традиційними транзакційними базами даних. Наразі буде описана послідовність перенесення інформації з пристрою на постійне сховище. Візуально потоки даних у цьому випадку використання показані на схемі (рисунку 4.1).

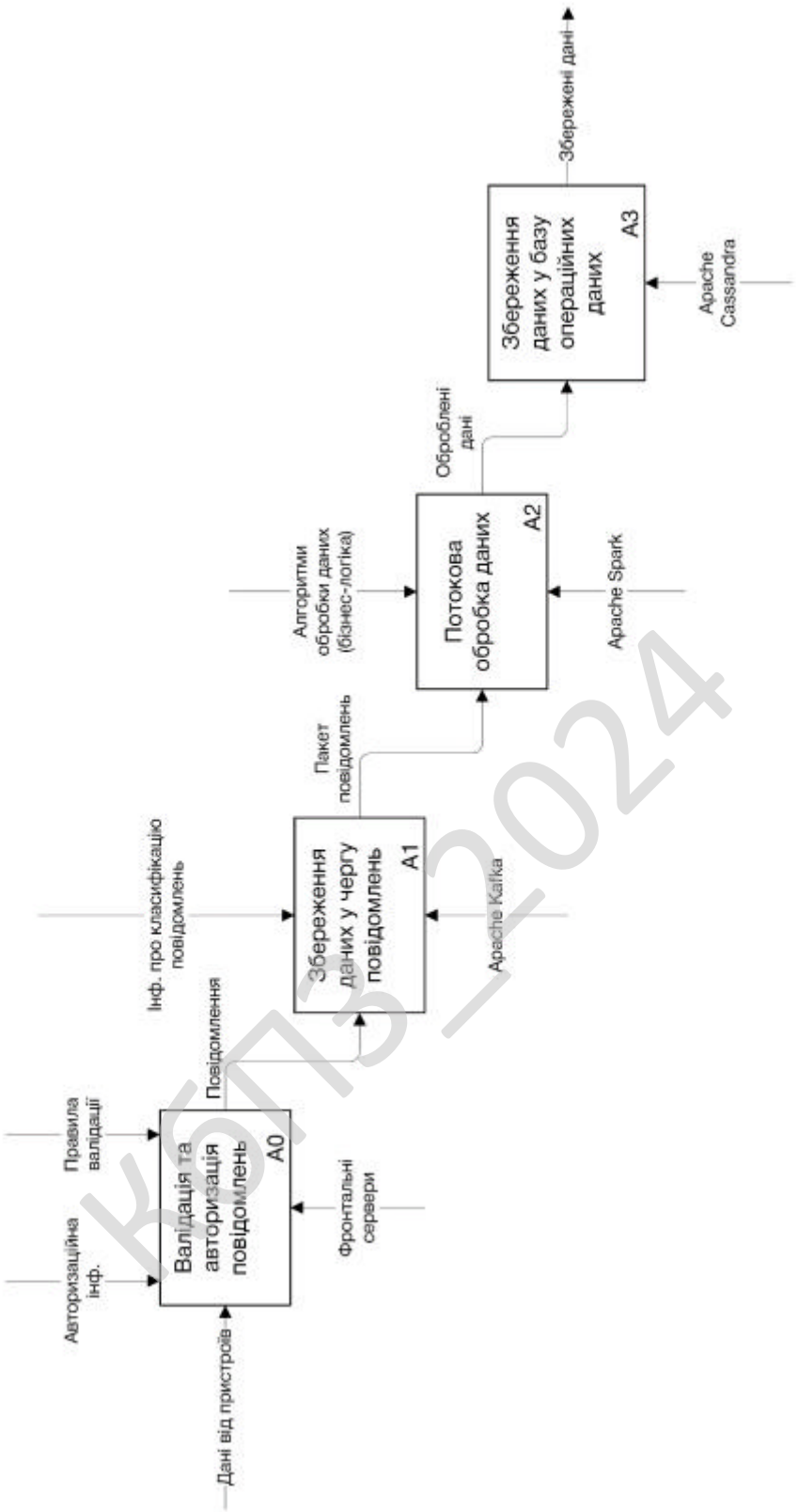

Рисунок 4.1 – Діаграма потоків і процесів при отриманні даних від сенсорів

Опишемо передачу інформації від пристрою до адміністратора. Такою інформацією може бути певна інформація про стан, наприклад, стан акумуляторів пристрою або певні сповіщення про проблеми.

Інформація про статус надсилається з пристрою на фронт-енд-сервери, які, як і при отриманні оперативних даних, підтверджують і аутентифікують одержувача запиту.

Після цього дані надсилаються до адміністративної бази даних. У цьому випадку зазвичай потрібні значні гарантії постійного зберігання та узгодженості даних. Таким чином, після того, як зовнішній сервер надсилає дані до БД, він очікує підтвердження отримання та збереження, перш ніж надсилати відповідне підтвердження на пристрій, який може реалізувати певну логіку повторної спроби, якщо повідомлення є важливим. Після збереження цієї інформації адміністратор надсилає запит на зовнішні сервери, які перевіряють його права доступу до цих даних і роблять запит до бази даних відповідно до предикату запиту (наприклад, повідомлення про стан пристрою із зазначеним ідентифікатором).

Іншим джерелом інформації про стан є механізми моніторингу. Адміністратори у вигляді правил встановлюють певні умови, які повинні бути дотримані під час очікуваної роботи пристрою, і визначають порядок їх перевірки. Наприклад, адміністратор може визначити правило, згідно з яким певний пристрій має надсилати принаймні певну кількість повідомлень за одиницю часу. Якщо ця умова не виконується, тобто активується певний тригер, платформа надсилає повідомлення про статус, щоб повідомити адміністратора про можливу проблему. Подібні умови можна перевірити в різних частинах системи. Внутрішні сервери можуть відстежувати стан з’єднання з пристроями. Підсистема обробки даних може періодично (за розкладом) опитувати сховище оперативних даних для виявлення потенційно некоректних даних, спричинених збоєм пристрою. Також опишемо передачу команд від адміністратора на пристрій. Команда адміністратора надсилається на зовнішній сервер, який

надійно зберігає її в адміністративній базі даних. Пристрої періодично опитують зовнішні сервери щодо нових команд. Потоки даних показано на рисунку 4.2. Така архітектура необхідна для надійної доставки даних на пристрої, оскільки вони можуть бути тимчасово недоступні (вимкнені або не мають зв'язку з сервером). Загалом можна було б зменшити затримку передачі даних і заощадити ресурси, надсилаючи дані безпосередньо (якщо це можливо). Наприклад, якщо зовнішній сервер має відкрите підключення до пристрою (наприклад, через WebSocket), то при отриманні команди від адміністратора її можна відправити негайно. Але це зменшить гарантії доставки. Якщо команда записана в базі даних, пристрій може отримати її, виконати і лише після цього відправити підтвердження на фронтенд-сервер, який позначить команду як прийняту або видалить її. Якщо команду надіслано безпосередньо на пристрій, він може підтвердити отримання, але якщо під час виконання станеться помилка, пристрій не зможе знову запитати команду з зовнішнього сервера, оскільки вона не буде збережена. Якщо ви не підтвердите отримання команди перед її виконанням, то адміністратору доведеться довго чекати завершення свого запиту, що погіршить його сприйняття інтерфейсу в порівнянні з асинхронною схемою, коли він отримує підтвердження відразу після того, як його команда буде збережена в базі даних. Тепер опишемо дії, які відбуваються при отриманні аналітичних запитів. Інтерфейсні сервери автентифікують запит і визначають його тип. Запит може бути задоволений доступними даними або може вимагати додаткових обчислень. Існуючі дані - це ті, що пройшли первинну обробку та зберігаються в оперативній базі даних. Для цього типу запиту достатньо прочитати відповідні дані та надіслати їх. Якщо надійшов аналітичний запит, то необхідно сформувати план його виконання, зчитати дані з однієї або обох баз даних, а потім обробити їх. Якщо потрібна лише додаткова обробка наявних даних, то на вхід системи для обробки подається лише інформація з оперативної бази даних. Адміністративну базу даних можна використовувати, якщо запит вимагає читання певної інформації про пристрій. Наприклад, аналітику можуть

знадобитися зведені дані від датчиків у певній області. У цьому випадку необхідно спочатку здійснити пошук в адміністративній базі ідентифікаторів пристроїв на досліджуваній території, а потім за отриманими ідентифікаторами зчитувати оперативні дані з відповідних датчиків.

З огляду на вищезазначене, потоки даних у сценарії виконання аналітичного запиту виглядають, як показано на діаграмі.

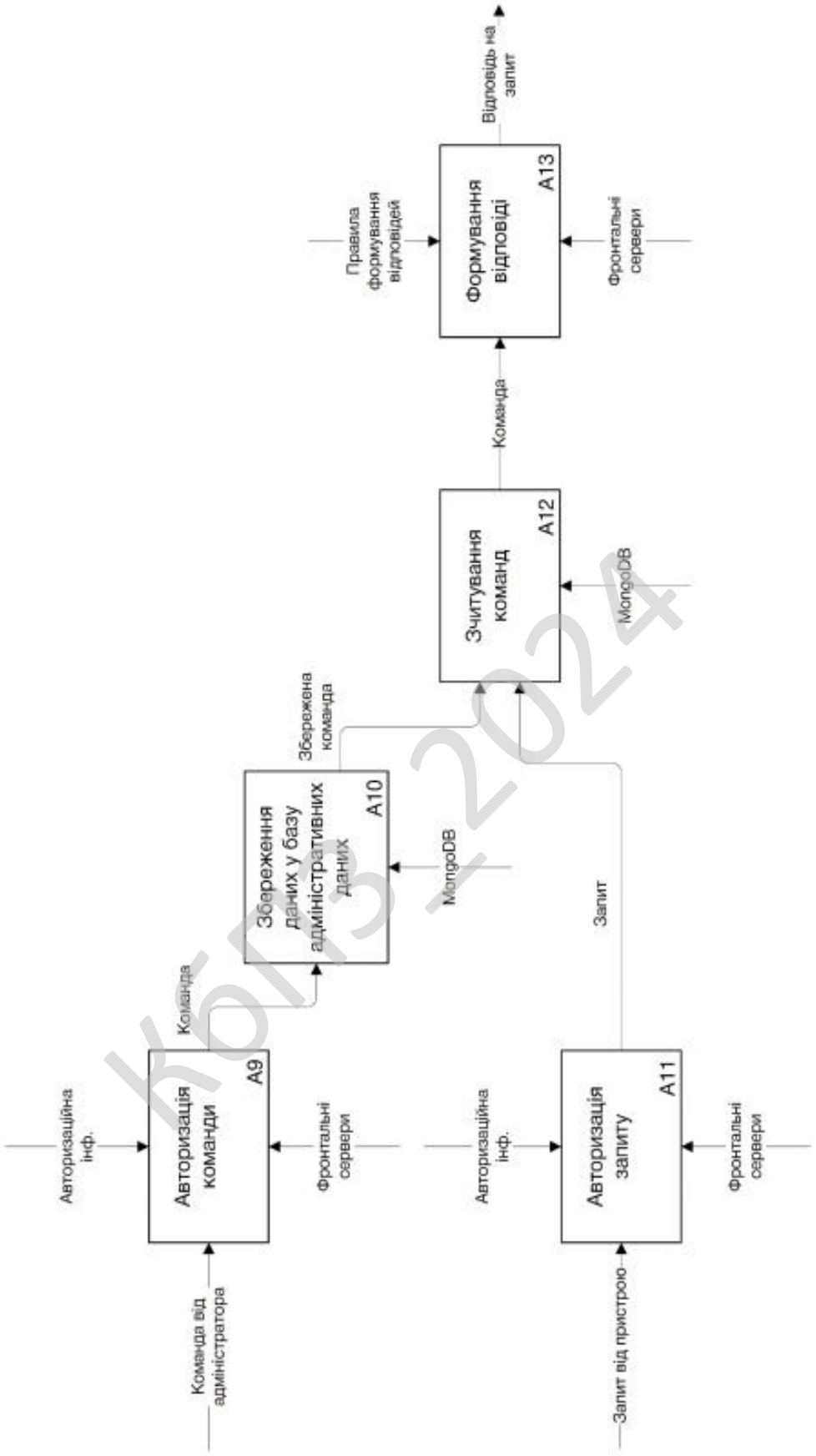

Рисунок 4.2 – Діаграма потоків даних і процесів при передачі команд від адміністратора до пристрою

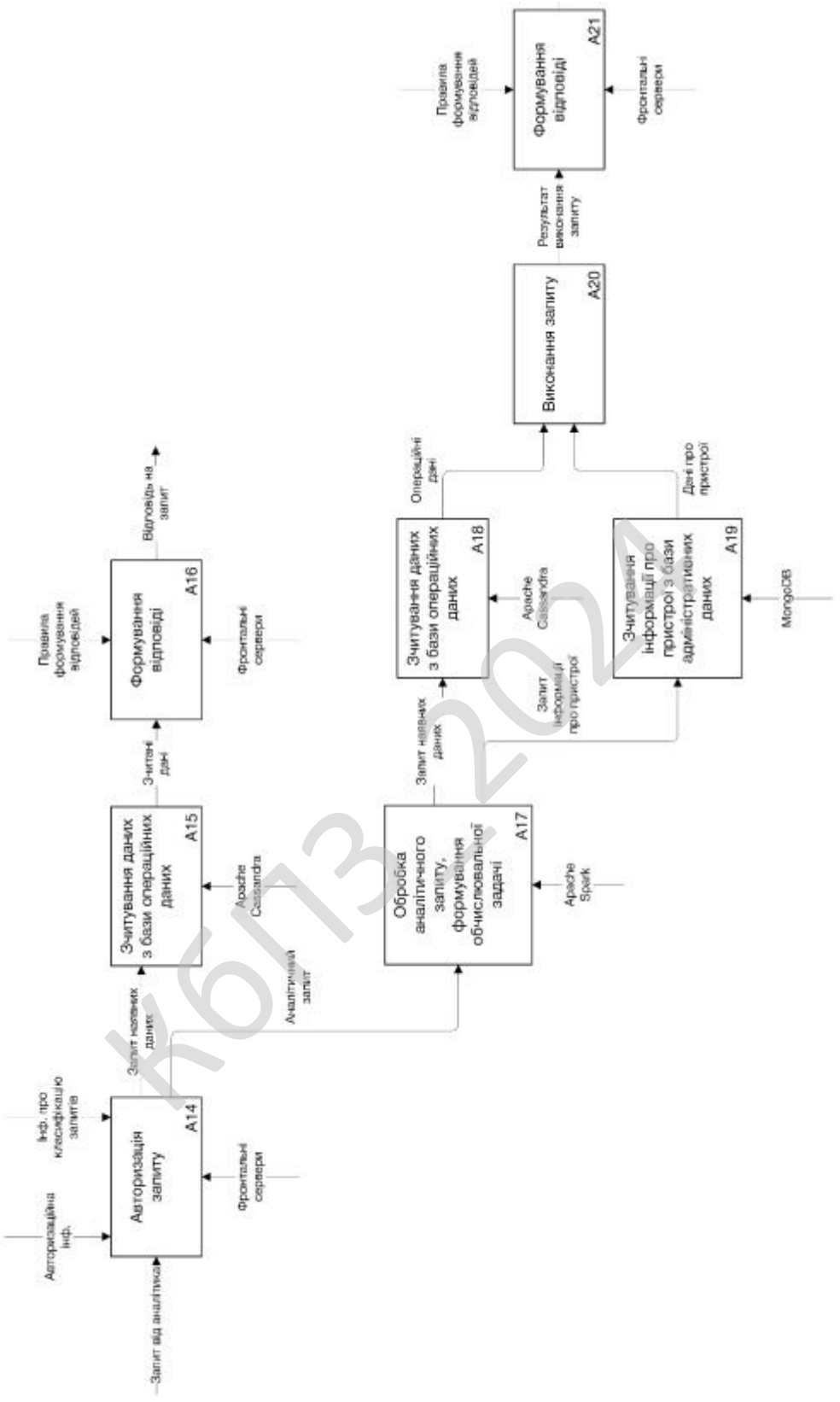

Рисунок 4.3 – Діаграма потоків даних і процесів при обробці запитів від аналітика

Протоколи та формати обміну даними між компонентами системи Опишемо протоколи передачі даних між компонентами системи. Зв’язок між пристроями (або шлюзами) і платформою відбувається через зашифрований протокол HTTP/2, який використовує криптографічний протокол TLS 1.2. Шифрування трафіку потрібне, щоб уникнути перехоплення даних та інших атак. Протокол HTTP/2, стандартизований [17] у 2015 році, дуже підходить для IoT, оскільки:

- розроблено як оптимізацію HTTP/1.1;

- реалізує стиснення заголовка HTTP (HPACK) за допомогою коду Хаффмана. Цей спосіб високоефективний, але вимагає мало пам'яті, що дуже важливо для малопотужних пристроїв;

- за допомогою таких прийомів, як конвеєрність і мультиплексування, стимулює використання єдиного TCP-з'єднання, що дозволяє уникнути повільних тристоронніх рукостискань, які, крім того, вимагають додаткових ресурсів. У зашифрованому протоколі повторне використання з’єднання дозволяє уникнути ще більш тривалого діалогу TLS (узгодження);

- забезпечує виконання операції PING на рівні протоколу, що дозволяє з її допомогою перевіряти правильність з'єднання з пристроєм;

- підтримує високорівневу сумісність з HTTP/1.1: методи, коди стану, URI, поля заголовків. Багато платформ IoT використовують MKTT, який набагато ефективніший, ніж HTTP/1.1. Проте з появою HTTP/2 HTTP/2 і MKTT стали майже рівними за продуктивністю.

Основною перевагою використання HTTP є його поширеність і загальне визнання в Інтернеті, що означає наявність функціональних і стабільних серверних і клієнтських бібліотек.

HTTP/2 реалізує техніку серверного проштовхування, яка дозволяє серверу ініціювати надсилання даних клієнту замість традиційної моделі запит-відповідь. Однак більш природним і ефективним для повного дуплексного зв'язку є використання протоколу WebSocket [18]. Пристрої, якщо вони можуть

встановити з’єднання з сервером, підтримують відкритий канал, через який вони можуть отримувати адміністративні команди від сервера за допомогою протоколу WebSocket. Використовується захищена версія протоколу WebSocket - WebSocket Secure (VSS). JSON використовується як основний формат даних, у якому зовнішні сервери отримують інформацію. Він, як і HTTP, дуже поширений; читається людиною (тобто не двійковий); простий у створенні та аналізі. Недоліком порівняно з двійковими форматами є розмір повідомлення, але його можна подолати за допомогою стандартних методів стиснення, таких як gzip. Загалом зовнішні сервери реалізують REST API, який використовується пристроями, адміністраторами та аналітиками. Як уже згадувалося, зв’язок між пристроєм і точкою входу в хмарну інфраструктуру платформи відбувається за допомогою зашифрованого з’єднання. Розшифровка (термінація SSL) відбувається на рівні балансувальника навантаження, а зовнішні сервери отримують уже відкриті дані. Це можна зробити, оскільки з’єднання всередині центру обробки даних можна вважати безпечним: трафік або ізольований від інших користувачів хостингу за допомогою засобів віртуалізації, або інфраструктура платформи може бути розташована в повністю фізично ізольованому сегменті мережі.

Крім того, він дозволяє виконувати дешифрування за допомогою спеціально адаптованих більш продуктивних рішень, витрачаючи при цьому ресурси фронтенд-серверів тільки на базову логіку обробки запитів. Передача даних між чергою повідомлень, підсистемою обробки даних, оперативною базою даних, адміністративною базою даних і зовнішнім сервером відбувається в певних двійкових форматах через TCP, реалізованих відповідними бібліотеками. Наприклад, підсистема обробки даних може обмінюватися серіалізованими об’єктами між своїми робочими серверами під час обчислень. Наочно показані формати даних і протоколи їх передачі між різними компонентами системи рисунку 4.4.

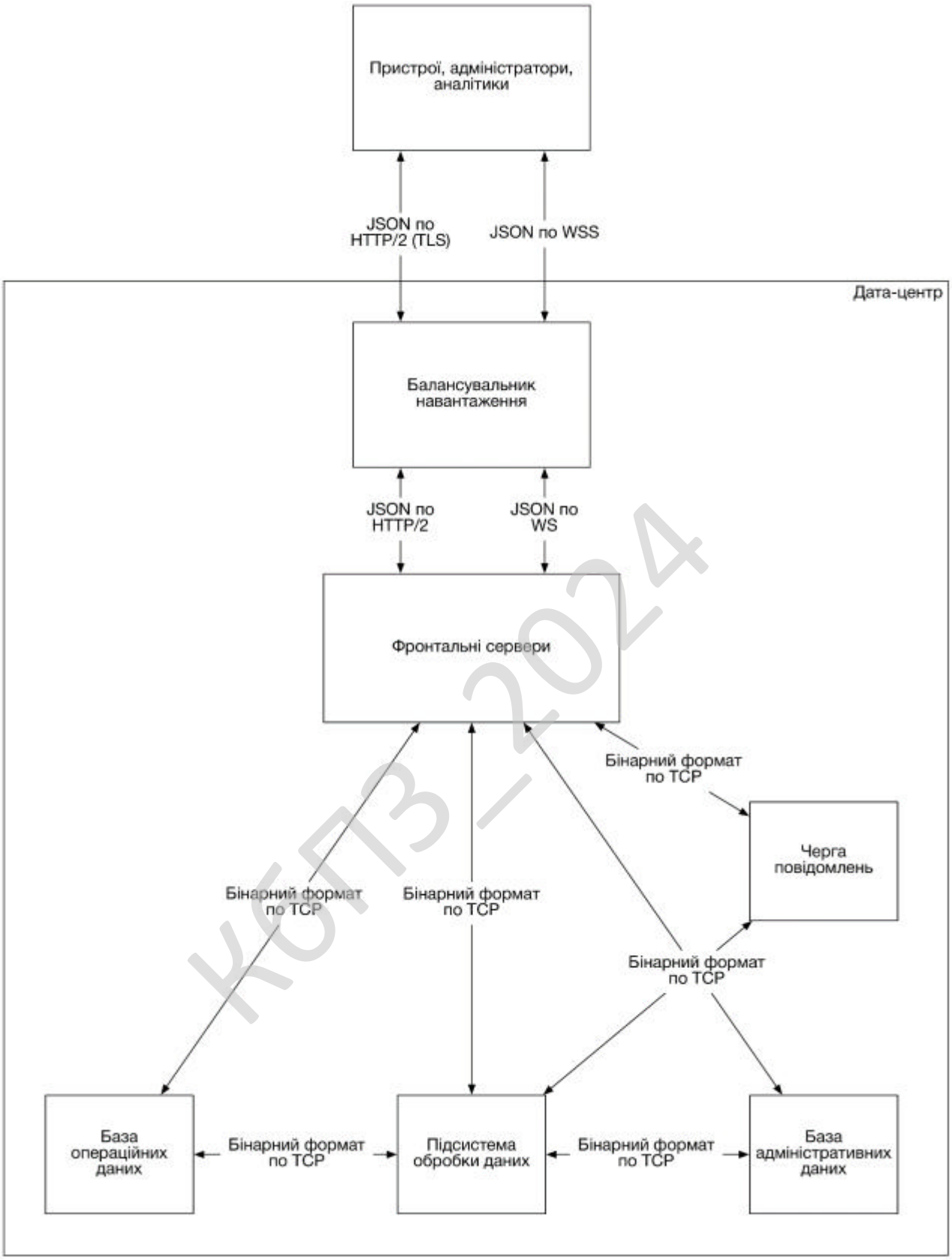

Рисунок 4.4 – Запропоновані формати і протоколи їх передачі між компонентами системи

Мікросервісна архітектура

Стиль архітектури мікросервісу - це підхід до розробки повної програми як набору невеликих служб, кожна з яких працює у своєму власному процесі та підключається до інших за допомогою легких механізмів, таких як RPC або HTTP. Сервіси будуються під конкретне завдання і можуть розгортатися самостійно за допомогою автоматизованих систем. Існує мінімальне централізоване управління такими послугами. Традиційні серверні системи зазвичай будуються як моноліти – логічно окремі виконувані файли. І такий підхід є природним: уся логіка обробки запитів виконується в одному процесі, що дозволяє використовувати наявні інструменти мови програмування, щоб розділити вашу програму на класи, функції та простори імен. Моноліт можна масштабувати горизонтально, запустивши багато екземплярів із балансувальника навантаження. Монолітне програмне забезпечення досить успішне, але воно має свої недоліки. Цикли зміни моноліту взаємопов'язані, тобто зміна невеликої частини системи вимагає перезбирання (компіляції) і розгортання. З часом стає важко підтримувати хорошу модульну структуру, зберігаючи зміни, що стосуються модуля, лише всередині цього модуля. Ви повинні масштабувати весь моноліт замість окремих частин, які вимагають більше ресурсів. Мікросервіси можна розгортати та масштабувати незалежно один від одного. Вони також забезпечують чіткі межі між модулями, навіть дозволяючи реалізовувати окремі підсистеми різними мовами програмування. Це розмежування допомагає керувати складністю кодової бази, оскільки кожен модуль матиме загальнодоступний API, який міститиме лише необхідні йому функції, а все інше буде інкапсульованим і не матиме відношення до розробки інших служб, які від нього залежать. Основним недоліком архітектури мікросервісів є підвищена складність обробки помилок. На відміну від моноліту, будь-який виклик служби може вийти з ладу. Тому необхідно розробити механізми моніторингу стану сервісів, перевірки різних метрик їх функціонування, а також автоматизувати відновлення мікросервісів у разі його збою. Позитивним є те, що збої в

архітектурі мікросервісів здебільшого поодинокі. Якщо для виконання запиту потрібно викликати багато служб, а деякі з них недоступні, можна дати менш повну відповідь (витончена деградація). Труднощі також можуть бути викликані забезпеченням послідовності розгортання залежних мікросервісів, а також необхідністю управління транзакціями, які взаємодіють з кількома підсистемами. Як бачимо, запропонована архітектура багато в чому увібрала ідеї мікросервісної архітектури. Компоненти чітко розділені та мають власний спеціальний набір функцій. Інтерфейсні сервери забезпечують єдину точку входу в систему. Коли запит отримано, відповідно до практики архітектури мікросервісу, зовнішні сервери здійснюють необхідну кількість викликів API служби, а потім об'єднують отримані результати для формування відповіді. Як описано, внутрішній зв'язок між серверами відбувається у двійковому форматі через TCP з міркувань ефективності, але з точки зору коду кожна служба надає загальнодоступний API, який інкапсулює створення повідомлень для відповідної служби. Окрім розподілу обов'язків, основною перевагою є можливість горизонтального масштабування кожної послуги окремо. Наприклад, якщо значно прискорити надходження оперативних даних без збільшення потоку адміністративних даних, відповідну СУБД можна масштабувати окремо.

Підсистема прийому даних. Організація, вимоги та обґрунтування вибору свого рішення

Оскільки дані надходять із високою швидкістю та можуть виникати стрибки трафіку, вам потрібен буфер між базою даних і зовнішнім сервером, щоб збалансувати навантаження. Для цього в архітектурі платформи передбачено використання черги повідомлень. Черги повідомлень визначають асинхронний протокол зв'язку. Це означає, що відправник і одержувач повідомлення не можуть спілкуватися з чергою повідомлень одночасно. Повідомлення в черзі зберігаються, доки їх не отримає одержувач. Важливою особливістю черги повідомлень є забезпечення стабільності та надійності, які реалізуються за допомогою різних стратегій зберігання даних. Для підвищення надійності

доставки повідомлень є можливість зберігати їх на диску до того, як їх отримає одержувач. Навіть якщо програма-одержувач або черга повідомлень виходить з ладу через збій, повідомлення будуть безпечними та доступними для одержувачів, щойно буде створено резервну копію системи. Використовуючи чергу повідомлень, ви можете підтримувати набагато більший обсяг повідомлень. Загалом, впровадження архітектури черги повідомлень є успішною стратегією для організації асинхронної обробки великих даних. Сформулюємо вимоги до черги повідомлень, які необхідно враховувати при виборі між доступними рішеннями:

1) Висока продуктивність. Необхідно забезпечити високу швидкість запису повідомлень у чергу.

2) Горизонтальна масштабованість. Можливість збільшення пропускної здатності системи за рахунок збільшення кількості серверів.

3) Можливість збереження повідомлень на диск. Необхідно зменшити кількість втрачених даних, коли сервер черги повідомлень вимкнено. Це особливо необхідно при зчитуванні даних з черги великими пакетами з великими інтервалами - в цьому випадку при відсутності постійного сховища велика кількість даних може бути втрачено.

4) Тиражування. Для певного збільшення обсягу зберігання даних він також гарантує доступність черги повідомлень для одержувачів у разі вимкнення частини сервера.

5) Можливість налаштування рівня гарантії надійності зберігання повідомлень. Тобто, на скільки серверів реплікується та як часто повідомлення зберігаються на диску. Необхідно налаштувати систему для більш надійного збереження повідомлень з пристроїв, які мають тривалий період між операціями надсилання даних. У разі великого навантаження слід мати можливість послабити такі гарантії, навпаки.

6) Можливість розподілу повідомлень на групи за темами. Таким чином, можна дати вказівку одержувачам повідомлень, які реалізують логіку, специфічну

для певного типу пристрою, отримувати повідомлення з певної черги, в якій розміщені відповідні дані. Apache Kafka найкраще задовольняє описані потреби. Обґрунтуємо цей вибір.

Apache Kafka – черга повідомлень для прийому даних

Apache Kafka — це брокер повідомлень з відкритим кодом, розроблений Apache Software Foundation і написаний на Scala. Kafka — це розподілений, розділений (розділений), реплікований журнал записів (журнал фіксації). Він пропонує функції обміну повідомленнями, але має унікальну архітектуру [20]. Спочатку коротко опишемо термінологію:

- стрічка повідомлень розділена на категорії, які називаються темами;

- процеси, які публікують повідомлення в Kafka, називаються виробниками;

- процеси, які підписуються на теми та обробляють потік опублікованих повідомлень, називаються споживачами.

- kafka працює як кластер, що охоплює один або більше серверів, кожен з яких називається брокером. Kafka копіює журнали для кожного розділу теми на налаштовану кількість серверів (цей параметр можна встановити для кожної теми окремо). Це дозволяє автоматично переключатися на репліку в разі збою сервера в кластері, щоб повідомлення залишалися доступними навіть за наявності збоїв. Kafka широко використовує файлову систему для зберігання та кешування повідомлень. Хоча диски зазвичай вважаються повільними, правильно спроектована дискова структура може працювати з тією ж швидкістю, що й мережа.

Лінійне читання та запис є найбільш передбачуваними моделями використання, тому операційна система їх ефективно оптимізує. Сучасні операційні системи забезпечують методи випереджального читання (читання сторінки кешу) і запису позаду (запис сторінки кешу), які попередньо завантажують дані у великі блоки та групують малі логічні операції запису у великі фізичні операції. Щоб компенсувати розрив у продуктивності між

оперативною пам’яттю та дисковою пам’яттю, сучасні операційні системи дедалі агресивніше використовують основну пам’ять для кешування диска. Майже всю вільну пам'ять можна використовувати для кешування диска з мінімальними витратами на звільнення. Усі операції читання та запису проходять через цей єдиний кеш. Таку функцію неможливо легко вимкнути без використання прямого вводу-виводу, тому, якщо процес підтримує власний кеш, дані, швидше за все, дублюються в кеші сторінок ОС, тобто одна й та сама інформація зберігається двічі. Крім того, оскільки Kafka працює на JVM, тоді:

- витрати на зберігання об'єктів дуже високі, що часто подвоює розмір даних (або більше);

- Збирання сміття JVM значно сповільнюється, коли розмір даних, що зберігаються в купі, збільшується.

Враховуючи наведені вище фактори, використання файлової системи з кешуванням сторінок є більш ефективним, ніж підтримка кешу чи іншої структури в пам’яті, оскільки вона: принаймні подвоює доступний кеш шляхом автоматичного доступу до всієї вільної пам’яті, і, швидше за все, вдвічі більше завдяки зберіганню компактна структура байтів замість окремих об'єктів. У результаті розмір кешу може досягати 28-30 ГБ на машині з 32 ГБ пам’яті без накладних витрат на збирання сміття. Крім того, цей кеш зберігатиме дані навіть після перезапуску процесу, тоді як внутрішній кеш потрібно перебудувати (що може зайняти до 10 хвилин для 10 ГБ кешу), інакше продуктивність буде дуже низькою, якщо почати з порожнім. .

Це також спрощує реалізацію, оскільки підтримка узгодженості між кеш-пам’яттю та файловою системою обробляється ОС, яка робить це більш ефективно та коректно порівняно з реалізаціями в процесі.

Якщо модель використання диска показує часте лінійне читання, тоді читання наперед заповнює кеш необхідними даними для кожної операції читання. Натомість, щоб зберегти кеш якомога більшим і швидко скинути дані на диск, коли місце закінчиться, робиться навпаки. Усі дані негайно записуються у

файлову систему, але це не обов'язково спричиняє операції фізичного запису. Це означає, що дані переміщуються на сторінки кешу ядра. Пакетна обробка - гарантія ефективності.

Щоб забезпечити це, постачальник даних (виробник) намагається накопичувати дані в пам'яті та надсилати більші пакети за одну операцію. Пакетну обробку можна налаштувати так, щоб накопичувати не більше певної фіксованої кількості повідомлень або очікувати не більше певного ліміту затримки (скажімо, 10 мс). Це дозволяє виконувати менше операцій введення-виведення, кожна з яких матиме більший обсяг. Настроювана буферизація дозволяє знайти компроміс між додатковою затримкою та вищою пропускною здатністю.

Споживач даних (споживач) працює, надсилаючи запити на вибірку брокерам, які керують розділами, з яких йому потрібні дані. З кожним запитом він визначає зміщення в журналі та отримує сегмент журналу, починаючи з указаної позиції. Таким чином, споживач даних має значний контроль над цією позицією і тому може перечитати повідомлення якщо потрібно.

Можна виділити декілька різних видів доправлення повідомлень:

- максимум один раз (не більше одного разу) – повідомлення можуть бути втрачені, але більше ніколи не доставлені;

- принаймні один раз – повідомлення ніколи не втрачаються, але можуть бути відправлені повторно;

- рівно один раз – повідомлення доставляється рівно один раз.

Ми опишемо семантику доставки повідомлень у Kafka. При публікації повідомлення існує поняття запису (внесення) повідомлення в журнал. Після написання повідомлення воно не буде втрачено, доки принаймні один посередник, який обслуговує розділ, до якого воно було написано, залишається в робочому стані. Якщо постачальник даних відчуває помилку мережі під час публікації повідомлення, він не може визначити, чи сталася помилка до чи після написання повідомлення. Ця семантика подібна до вставки рядка в таблицю в

базі даних із автоматично згенерованим ключем. Таким чином, Kafka за замовчуванням гарантує доставку «принаймні один раз» і дозволяє користувачеві реалізувати схему «принаймні один раз», вимикаючи повторні спроби на стороні постачальника та зміщення запису перед обробкою пакетів повідомлень. Семантика «точно один раз» вимагає співпраці з кінцевою системою зберігання даних.

Apache Kafka використовує Apache Zookeeper, розподілену службу конфігурації та синхронізації. Він використовується для координації вузлів у кластері, моніторингу статусу брокера, запису позицій під час читання повідомлень тощо.

Підсистема обробки даних

Архітектура Lambda - це архітектура обробки даних, розроблена для роботи з великими обсягами даних, одночасно використовуючи переваги пакетного та потокового підходів до обробки даних. Цей архітектурний підхід намагається збалансувати затримку, пропускну здатність і відмовостійкість за допомогою пакетної обробки, щоб забезпечити всебічне й точне представлення пакетів даних, одночасно використовуючи потокове передавання в реальному часі для створення робочих фрагментів даних. Ці дві частини даних можна об’єднати перед виведенням. Поширення лямбда-архітектур зумовлене збільшенням обсягів даних, потребами в аналітиці в реальному часі та зусиллями щодо зменшення затримок MapReduce. Архітектура Lambda працює з моделлю даних, яка має незмінне джерело (лише додавання). Ця модель призначена для захоплення та обробки подій із мітками часу, які додаються до наявних даних, а не перезаписуються. Стан системи визначається природним упорядкуванням даних у часі. Архітектура Lambda показана на схемі (рисунок 4.5).

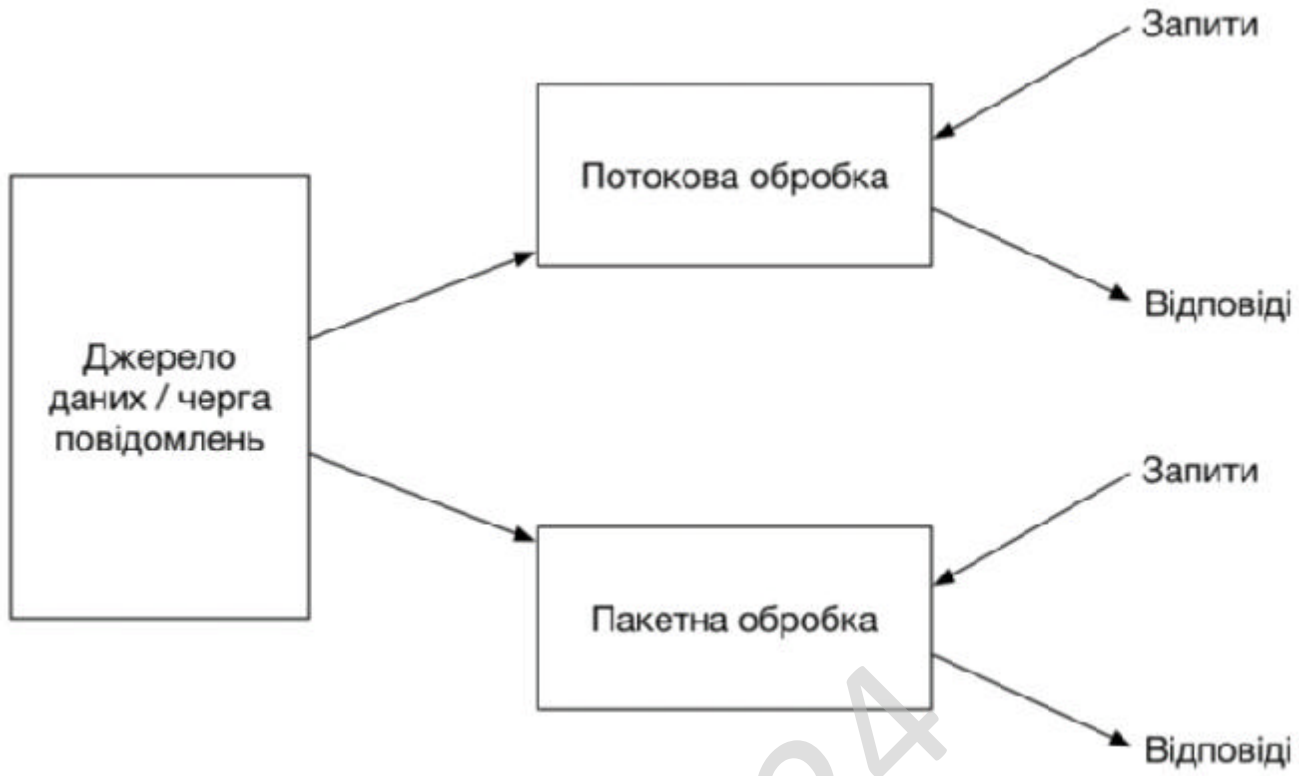

Рисунок 4.5 - Лямбда-архітектура

Також опишемо підхід MapReduce, який частково використовується для обробки даних платформою. Цей підхід призначений для обробки завдань, які є паралельними. Він складається з наступних кроків:

- Карта: кожен робочий сервер застосовує функцію карти до даних і записує результат у тимчасову пам’ять. Головний сервер забезпечує обробку лише однієї копії вхідних даних.

- Shuffle: робочі сервери розподіляють дані відповідно до вихідних ключів (виходи функції карти), щоб усі дані, що відповідають одному ключу, були на одному робочому сервері.

- Скорочення: робочі сервери обробляють кожну групу даних, відповідно до ключа, паралельно.

Розглянемо обчислення в парадигмі MapReduce на прикладі обробки значень температури, отриманих від декількох пристроїв.

Обробка буде складатися з усереднення значень температури для кожного пристрою (це робиться протягом певного періоду часу). На вході ми маємо

записи JSON, які містять ідентифікатор пристрою (всього їх 3) в полі «deviceId» і значення температури в полі «temp». Початкова фаза обробки (карта) формує пари ключ-значення: ключ – це ідентифікатор пристрою, а значення – температура. Вважайте, що MapReduce працює на трьох серверах. На етапі «тасування» дані розподіляються так, що перший робочий сервер містить дані з першого пристрою, другий — з другого і так далі. Після «тасування» на кожному сервері залишаються записи з однаковими значеннями ключів. Етап «зниження» полягає в обчисленні середнього арифметичного значень температури. Наприкінці розрахунку ми маємо три пари «ключ-значення», що містять, відповідно, ID пристрою та середнє значення температури, отримане з нього.

Apache Spark - система обробки даних

Apache Spark - це система обробки великих даних з відкритим кодом, створена для високої швидкості, простоти використання та складної аналітики [21]. Розроблено в 2009 році в AMLab (UC Berkeley), публічний випуск як проект Apache відбувся в 2010 році. Spark має значні переваги перед іншими технологіями Big Data і MapReduce, такими як Hadoop і Storm. Перш за все, Spark забезпечує комплексну та уніфіковану структуру для роботи з наборами даних різного характеру (текст, графік тощо) та джерела (пакет або потік).

Було доведено, що Spark [22] працює до 100 разів швидше, ніж Hadoop, під час виконання операцій у пам’яті та до 10 разів швидше, якщо використовується диск. Він має велику бібліотеку вбудованих даних і алгоритмів. Окрім операцій зіставлення та скорочення, він підтримує запити SQL, потокову передачу, машинне навчання та обробку даних графіків. Ці можливості можна використовувати окремо або об'єднати в єдиний ланцюжок обробки даних. Hadoop і Spark. Hadoop як технологія обробки великих даних популярна вже близько 10 років і довела свою ефективність. MapReduce є чудовим рішенням для однопрохідних обчислень, але не настільки ефективним для завдань, які потребують багатопрохідних обчислень і алгоритмів. Кожен крок у ланцюжку обробки даних має одну фазу Map і Reduce, і вирішення будь-якої проблеми має

бути описано цими операціями. Вихідні дані кожного кроку повинні бути збережені в розподілену файлову систему перед початком наступного кроку. Таким чином, цей підхід є досить повільним через реплікацію та зберігання на диску. Крім того, рішення на основі Hadoop зазвичай використовують складні для створення та керування кластери. Їм також потрібно інтегрувати кілька інструментів для різних випадків використання (наприклад, Mahout для машинного навчання та Storm для потокового передавання). Для складних обчислень кілька завдань MapReduce мають бути об’єднані та виконуватися послідовно. При цьому кожне завдання має тривалу затримку і не може початися до повного завершення попереднього. Spark дозволяє розробляти складні багатоетапні ланцюжки обробки даних за допомогою шаблону спрямованого ациклічного графіка (DAG). Він також підтримує обмін даними в пам’яті за допомогою різних графіків, що дозволяє виконувати кілька завдань над одними даними. Spark працює з існуючою інфраструктурою розподіленої файлової системи Hadoop (HDFS), розширюючи та додаючи до неї функціональність. Особливості Spark. Spark покращує MapReduce, забезпечуючи швидший і ефективніший етап Shuffle під час обробки даних. Продуктивність у декілька разів вища, ніж у інших технологій великих даних, завдяки зберіганню даних в оперативній пам’яті та обробці майже в реальному часі. Spark підтримує відкладене оцінювання запитів, що допомагає оптимізувати їх виконання. API високого рівня спрощує розробку та забезпечує узгоджену архітектурну модель для складних рішень. Проміжні результати зберігаються в пам’яті, а не на диску, що корисно, коли над певним набором даних виконується кілька операцій. Зазвичай система обробки працює як у пам'яті, так і на диску. Якщо дані не поміщаються в пам'ять, виконуються зовнішні операції. Spark можна використовувати для обробки наборів даних, розмір яких перевищує загальну пам’ять серверів у кластері. Spark намагається зберегти якомога більше даних у пам’яті. Деякі дані можуть бути на диску. Використання оперативної пам’яті є однією з головних переваг у продуктивності, але ви повинні відповідно

проаналізувати вимоги до обладнання, щоб використовувати її ефективно. Стійкий розподілений набір даних [23] (RDD) є основною концепцією Spark. RDD можна порівняти з таблицею в базі даних. Він може містити будь-які типи даних. Spark зберігає RDD на різних розділах. Ця абстракція дозволяє оптимізувати обробку шляхом зміни порядку обчислень. Вони також відмовостійкі, оскільки RDD знає, як відтворити або перерахувати набір даних. RDD є незмінними. Його можна змінити шляхом перетворення, але він повертає новий RDD, а вихідний RDD залишається незмінним. Підтримуються два типи операцій:

- перетворення. Повертає новий RDD. Виклик функції перетворення RDD повертає новий RDD, вони не запускаються, доки обчислення не розпочато явно (ліниві обчислення). Прикладами перетворень є map, filter, flatMap, groupBiKey, reduceBiKey, aggregateBiKey, pipe та coalesce;

- дії. Дія виконується негайно та повертає значення. Приклади: зменшити, зібрати, підрахувати, спочатку, взяти, countBiKey і foreach.

Окрім API Spark Core, існують додаткові бібліотеки, які є частиною екосистеми Spark і надають додаткові можливості в сферах аналітики та машинного навчання.

Ці компоненти включають:

- Spark Streaming. Його можна використовувати для обробки потокових даних у реальному часі. Він заснований на мікропакетному стилі обчислень і обробки. Використовується абстракція DStream, яка є масивом RDD. Основна перевага полягає в тому, що той самий код, який описує ланцюжок обробки даних, можна використовувати як для пакетної, так і для потокової обробки даних;

- Spark SQL. Він надає можливість відкривати набори даних Spark через API JDBC і запускати SQL-запити за допомогою традиційних інструментів BI (бізнес-аналітики) і візуалізацій. За допомогою цієї бібліотеки ви можете виконувати ETL (Extract, Transform, Load) дані з різних форматів (JSON, Parquet,

реляційна БД та інші), перетворювати їх і робити доступними для спеціальних запитів;

- Spark MLlib. Масштабована бібліотека машинного навчання, що містить традиційні алгоритми навчання та утиліти, включаючи класифікацію, регресію, кластеризацію, спільну фільтрацію, зменшення розмірності та примітиви оптимізації низького рівня.

Операційна підсистема зберігання даних. Організація, вимоги та обґрунтування вибору стороннього рішення.

Після первинної обробки дані записуються в оперативну базу даних, що забезпечує їх постійне зберігання. З цього сховища дані зчитуються як зовнішніми серверами для виконання запитів, так і підсистемою обробки даних для подальшого виконання обчислень, необхідних для задоволення аналітичного запиту. Отже, сформулюємо вимоги, яким має відповідати оперативна база даних:

- оптимізовано для запису. Зазвичай майже у всіх подібних сховищах оперативних даних у системах великих даних, як-от розглянута платформа, спостерігається головна особливість: кількість операцій запису значно перевищує кількість операцій читання. Цей шаблон використання відрізняється від того, для якого оптимізовані традиційні реляційні бази даних. Вони найкраще підходять для сценарію, коли дані лише час від часу записуються в базу даних, але запити на читання є частими. Наприклад, якщо є певна таблиця, яка містить новини на певному веб-сайті, очевидно, що записів набагато менше, ніж читань. Якщо сховище даних обслуговує систему IoT, воно повинно виконувати десятки тисяч запитів на запис у кожну секунду. Читання має здійснюватися лише за запитом адміністратора або аналітика. Також було б непогано мати високу продуктивність читання, але це набагато менш важливо, ніж швидкість запису;

- горизонтальна масштабованість. Збільшення кількості серверів має збільшити пропускну здатність для операцій запису, а також обсяг збережених даних;

- доступність. За допомогою реплікації має бути забезпечена доступність бази даних (для запису та читання) у разі відключення частини кластера;

- послаблені гарантії транзакцій. Оперативна база даних вимагає набагато слабших транзакційних гарантій, ніж традиційний набір ACID, гарантований реляційними даними. Наприклад, якщо ми говоримо про узгодженість, то більшість сценаріїв використання задовольнять граничну узгодженість;

- мінімальний набір функціональних можливостей для запитів. Оскільки підсистема обробки даних використовується для виконання запитів, оперативна база даних повинна забезпечувати лише мінімальний набір операцій, таких як запис, читання, видалення тощо.

Наприклад, якщо до запиту необхідно застосувати функцію агрегації (таку як сума, мінімум або максимум), то це зробить підсистема обробки даних, а операційній базі даних потрібно лише прочитати необхідні дані (відфільтровані за певним ключ, наприклад дані на певну дату).

Враховуючи вищезазначені вимоги, зрозуміло, що реляційна база даних навряд чи підійде для зберігання оперативних даних і що слід використовувати рішення NoSQL.

Реляційні бази даних записують дані набагато повільніше, ніж читають їх. Реплікація доступна в деяких базах даних SQL, але її функціональність часто обмежена та має певні характеристики через вимоги транзакцій ACID.

Горизонтальна масштабованість у реляційних базах даних (шардинг) дуже складна з тих же причин. Це призводить до того, що коли реляційній базі даних не вистачає дискового простору або вона не може обслуговувати необхідну кількість запитів за одиницю часу, можна лише збільшити потужність одного сервера (вертикальне масштабування), не витрачаючи багато часу та збільшуючи складність системи.

Крім того, необхідно зберігати великі обсяги даних, а продуктивність запитів до таблиць у реляційних базах даних значно знижується зі збільшенням розміру таблиць. Для зберігання оперативних даних ми виберемо Apache

Cassandra з баз даних NoSQL. Розповімо, як він влаштований і чому відповідає поставленим вимогам.

Apache Cassandra – база для зберігання операційних даних

Apache Cassandra - це система керування розподіленою базою даних із відкритим вихідним кодом, призначена для обробки великих обсягів даних, розподілених між великою кількістю неспеціалізованих серверів, і забезпечення високої доступності без єдиної точки відмови [24]. Cassandra надійна навіть для кластерів, частини яких розташовані в різних дата-центрах. База даних Cassandra написана на Java та включає повністю розподілену хеш-систему Dynamo, яка забезпечує майже лінійну масштабованість із збільшенням обсягу даних.

4.1 Розробка блок-схем та опис алгоритмів функціонування системи

Теорема CAP. Щоб зрозуміти Кассандру, варто спочатку згадати теорему CAP. Віна стверджує, що розподілена система не може надати більше ніж дві з наступних гарантій:

- узгодженість даних (всі вузли бачать однакові дані в будь-який час);

- доступність (гарантія отримання відповіді на кожен запит про його успішне або неправильне виконання);

- стійкість до поділу (незважаючи на поділ на ізольовані частини або втрату зв'язку з частиною вузлів, система не втрачає стабільності та здатності правильно реагувати на запити). Cassandra надає користувачам сильні гарантії доступності та відмовостійкості розділів, тобто, з точки зору теореми SAC, це AR-система. Послідовність налаштовується, що дозволяє знайти компроміс між узгодженістю та затримкою. Під час запису/читання даних ви можете встановити рівень узгодженості для кожного запиту. Модель даних BigTable / Log-structured.

У моделі даних BigTable первинний ключ і імена стовпців зіставляються з відповідними значеннями, утворюючи багатовимірне відображення. Кожен стіл має багато вимірів. Мітка часу є одним із таких розмірів, який дозволяє керувати

версіями таблиці, а також використовується для внутрішнього збирання сміття (видалених даних).

Таблиця 4.1 – Структура даних у Cassandra

Ключ рядка	Колонка 1	…	Колонка N
	Значення 1	…	Значення N
	Сімейство колонок		

Розглянемо приклад зберігання інформації для сценарію роботи з метеорологічними даними. У таблиці 4.2 ключ рядка є ідентифікатором пристрою («deviceId»), а значення температури і тиску записуються в стовпці «температура» і «тиск».

Таблиця 4.2 – Приклад зберігання даних

deviceId = 1	temperature	pressure
	24.3	738.4
deviceId = 2	temperature	pressure
	16.1	762.9

Варто відзначити, що в кожному рядку зберігаються не тільки значення, але і назви стовпців, що дозволяє структурі (схемі) бути динамічною. Родини колонок. Стовпці групуються в набори, які називаються сімействами стовпців, доступ до яких можна отримати за допомогою клавіші рядка. Створення сімейства стовпців є обов’язковим для зберігання даних. Зазвичай вони намагаються зберегти невелику кількість різних сімейств стовпців у просторі ключів, а сімейства змінюються лише час від часу. Однак кількість динаміків у родині необмежена. Ключовий простір. Ключовий простір — це група сімейств стовпців. Стратегії реплікації та контроль доступу реалізовані на рівні простору

ключів. У порівнянні з реляційними базами даних, ключовим простором є база даних (схема), а сімейством стовпців є таблиця. SSTable (Сортована таблиця рядків). SSTable - це формат зберігання файлів, який використовується Cassandra; це впорядкована незмінна структура рядків стовпців (пар "ім'я-значення"). Існує операція пошуку значення, пов’язаного з певним ключем, а також сортування пар ім’я стовпця – значення у вказаному діапазоні ключів. Кожна таблиця SSTable містить масив ключів рядка та набір пар ключ-значення стовпця. У файлі індексу є індекс і початкова позиція ключа рядка, які зберігаються окремо. Урізаний індекс завантажується в пам’ять, коли SSTable відкривається, щоб оптимізувати обсяг пам’яті, необхідний для індексу. Пошук рядків можна виконувати за допомогою пошуку на одному диску та послідовного сканування. Memtable. Memtable - це область пам’яті, до якої записуються дані під час операцій оновлення або видалення. Це тимчасове місце, з якого дані будуть записуватися на диск як SSTable, коли він заповнюється. Загалом операція оновлення або запису в Cassandra - це послідовний запис у журнал фіксації на диску та оновлення пам’яті; отже, запис відбувається так само швидко, як і запис у пам'ять. Виглядає так, як показано рисунку 4.6.

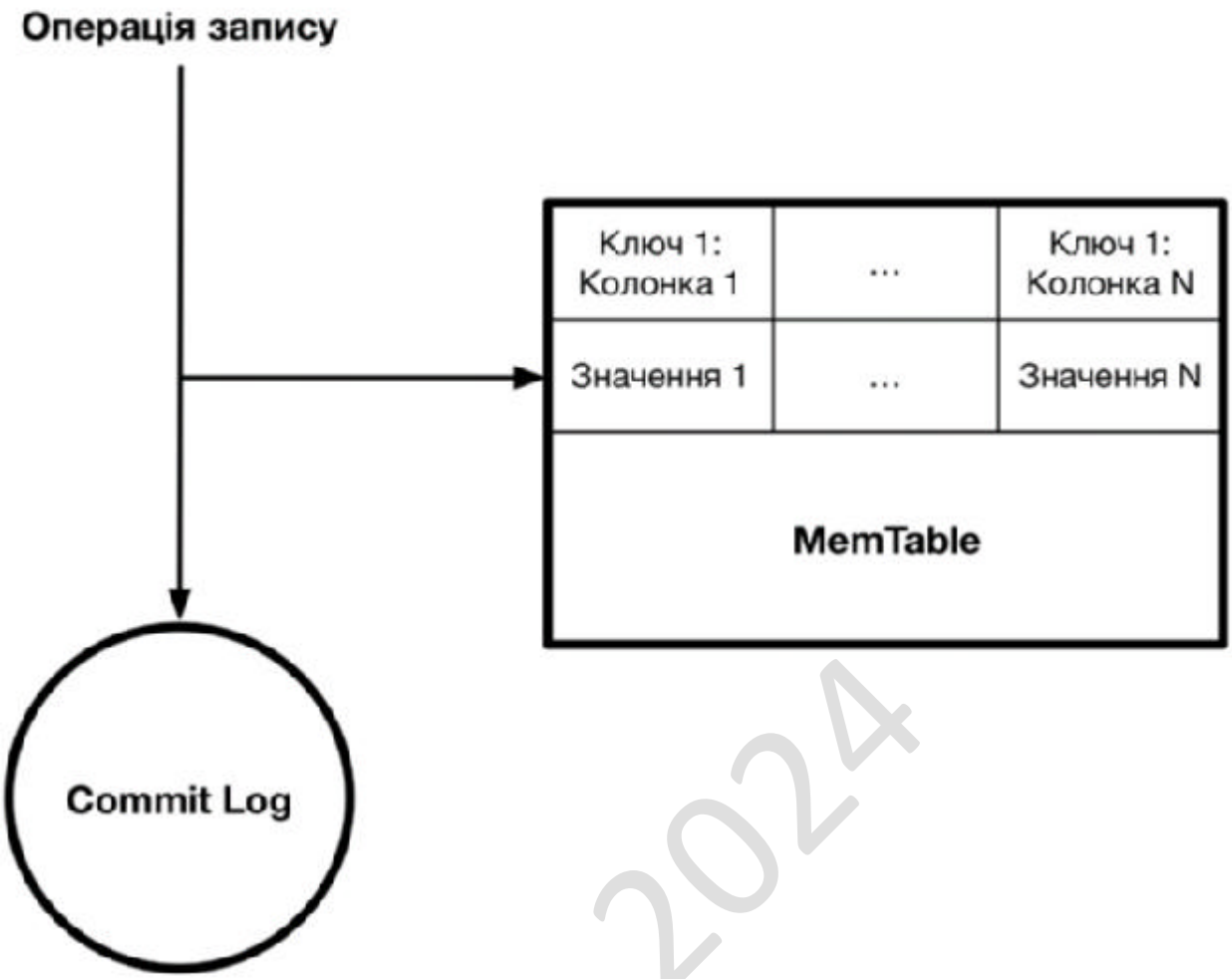

Рисунок 4.6 – Схема запису у Cassandra

Операції читання в Cassandra поєднують дані з різних SSTables і дані з Memtables. Запити на читання завжди повинні містити ключ рядка (первинний ключ), за винятком сканування діапазону. Коли надходить кілька запитів на оновлення певного стовпця, Cassandra використовує надані клієнтом позначки часу для вирішення конфліктів. Операції видалення працюють інакше: оскільки SSTables є незмінними, Cassandra записує так званий надгробний камінь, щоб уникнути випадкових записів. Надгробок — це спеціальне значення, яке записується замість миттєвого видалення даних і може бути видалено під час збирання сміття. Його можна надіслати вузлам у кластері, які не отримали перший запит на видалення. Ущільнення. Щоб обмежити кількість файлів, які беруть участь у читанні даних, і звільнити місце, зайняте непотрібними даними, Cassandra виконує стиснення. Таким чином, компактифікація поєднує кілька (це

число можна налаштувати) SSTables в одну велику SSTable. SSTables спочатку мають такий самий розмір, як Memtables. Таким чином, SSTtables експоненціально зростають у розмірі в міру старіння та надходження даних. Схема розбиття та реплікації подібна до описаної в статті Dynamo [25]. Протокол пліток. Cassandra — це однорангова система без єдиної точки відмови; інформація про топологію кластера передається за допомогою протоколу gossip. Протокол пліток схожий на плітки в соціальних комунікаціях (звідси і назва). Відповідно до цього протоколу вузол B передає іншим вузлам у кластері те, що він знає про стан вузла A. Ці вузли передають інформацію про A іншим, і через деякий час усі вузли дізнаються про стан A. Розподілена хеш-таблиця. Ключовою особливістю Cassandra є її здатність до поступового масштабування. Це включає в себе можливість динамічного розділення даних між набором вузлів у кластері. Cassandra поділяє дані на кластер за допомогою узгодженого хешування та рандомізує рядки в мережі відповідно до хешу ключа рядка. Коли вузол включається в кільце, йому призначається маркер, який вказує його розташування в кільці (рисунок. 4.7).

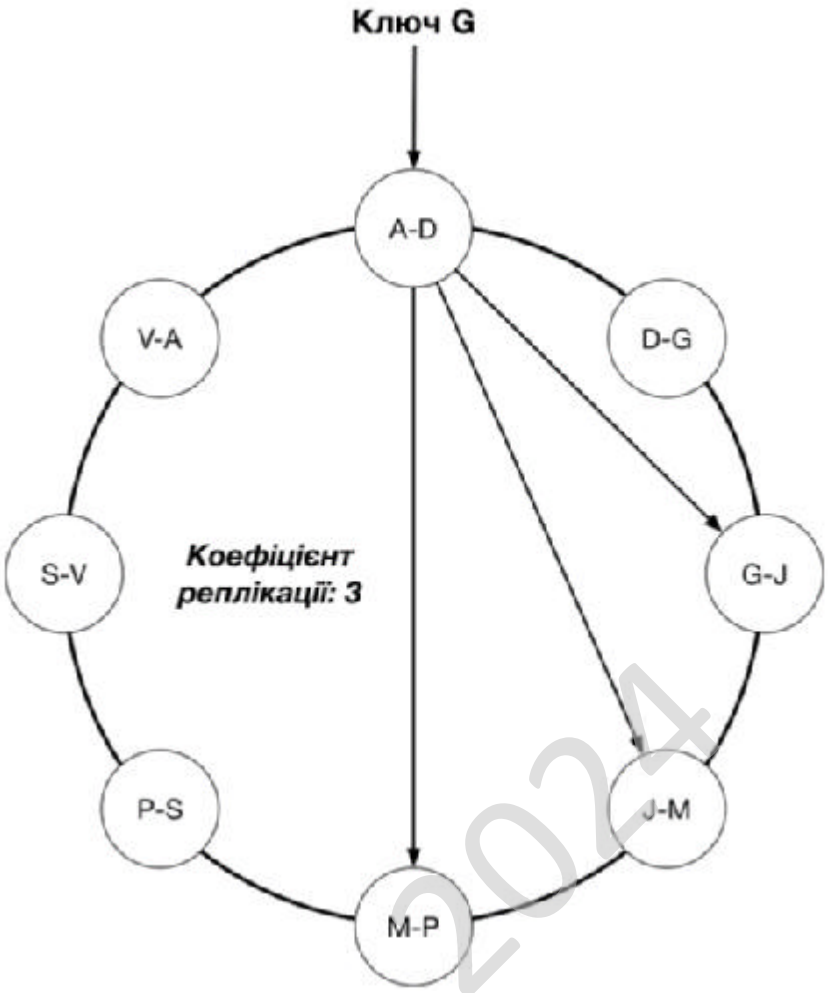

Рисунок 4.7 – Партиціонування даних при записі в Cassandra

Постійність підсумкового рахунку. Після достатнього періоду часу без змін будь-які оновлення мають поширюватися системою, синхронізуючи репліки. Cassandra підтримує як сильну узгодженість, так і остаточну узгодженість; це може контролювати клієнт, який виконує операцію. Cassandra підтримує різні рівні узгодженості під час запису та читання даних. Рівень узгодженості описує кількість реплік даних, до яких вузол-координатор повинен підключитися, перш ніж підтвердити клієнту завершення операції. Якщо V + R > K, де V — кількість реплік, які повинні підтверджувати записи, R — кількість вузлів, які повинні підтверджувати читання, K — коефіцієнт реплікації, тоді забезпечується надійна узгодженість. Доступні такі рівні консистенції:

- ONE: R/V принаймні для одного вузла;
- ALL: R/V для всіх вузлів для цього ключа розділу;

- КВОРУМ: R/V принаймні на ⌊K/2⌋+1 вузлах, де K — коефіцієнт реплікації.

Коли вузли не працюють на технічне обслуговування, Cassandra збереже інформацію про оновлення даних на цих вузлах для використання, щойно вузли знову стануть доступними. Крім того, для забезпечення узгодженості Cassandra використовує такі методи, як підказки щодо передачі даних, виправлення читання та виправлення антиентропії.

Програмна реалізація

Практична реалізація платформи виконана на Java (версія 8) та Scala (версія 2.11).

Програмний код основних компонентів наведено в додатку. Використовувалися такі бібліотеки:

- Undertow - це веб-сервер, написаний мовою Java, який дуже ефективний завдяки використанню неблокувального вводу-виводу. Це дозволяє описати логіку за допомогою окремих обробників, які обслуговують запити на певну адресу. За допомогою цієї бібліотеки реалізовані сервери переднього плану, тобто інтерфейси HTTP (REST API) і WebSocket.

Набір бібліотек та інтерфейсів для роботи з Apache Spark:

- конектор Spark-Cassandra – драйвер Apache Cassandra з інтерфейсом, інтегрованим з API Apache Spark;

- MongoDB Async Driver – драйвер MongoDB, реалізований за допомогою Java NIO для неблокуючих запитів до бази даних;

- бібліотека для створення та перевірки веб-токенів JSON;

- Apache Commons Codec – набір реалізацій кодеків, наприклад, Base64 для формування веб-токенів JSON;

- SLF4J (Java API) для запису різноманітних інформаційних та діагностичних повідомлень про хід програми;

- Logback – реалізація SLF4J.

Структура бази адміністративних даних

Адміністративна база даних, побудована на MongoDB, має дві колекції: користувачів і пристроїв.

Колекція Користувачі зберігають інформацію про свої права доступу. Хоча MongoDB підтримує динамічну схему, базовий набір полів у документах є незмінним.

Структура документів збірки

{ "login": рядок,

"пароль": рядок,

"ролі": рядок,

"info": Об'єкт

}

Наведена вище структура описана в ієрархічному форматі, подібному до JSON, у якому ключі є іменами полів у документі, а значення — типами даних у цих полях.

Поля «логін» і «пароль» мають тип string і призначені для аутентифікації.

Поле «роль» визначає роль користувача і, відповідно, його права доступу. Наприклад, користувач може бути адміністратором, аналітиком тощо.

У реалізації Java доцільніше зберігати роль користувача як enum, але MongoDB це не підтримує.

Можна було б зберігати enum у MongoDB як число, але рядковий тип більш інтуїтивно зрозумілий, і той факт, що він менш компактний, не викликає проблем через малу кількість записів у цій таблиці. У полі «info» можна (опціонально) зберігати додаткову інформацію про користувача у вигляді об'єкта з довільними полями.

Колекція пристроїв містить інформацію про пристрої, які використовуються для їх автентифікації та виконання різноманітних запитів.

Структура документів цієї збірки така:

{ "device_id": рядок,

"пароль": рядок,

"type": рядок,

"status_messages": рядок

}

Поля «device_id» і «password» використовуються для автентифікації. "type" – тип пристрою, enum. "location" містить координати пристрою у форматі GeoJSON.

Поля «status_messages» і «commands» містять повідомлення про стан пристрою та команди відповідно. "info" - довільний об'єкт з додатковою інформацією.

Опис статусних повідомлень виглядає так:

{ "створено": позначка часу,

"bi": рядок, "types": рядок,

"корисне навантаження": об'єкт,

"отримано": позначка часу }

«created» і «received» — це, відповідно, мітки часу створення запису та його отримання адміністратором. "bi" - відправник повідомлення (підсистема або трекер). "type" - тип повідомлення (наприклад, інформаційне або повідомлення про помилку). "корисне навантаження" - це довільний об'єкт з описом вмісту повідомлення.

Об'єкти, що описують команди, зазвичай мають таку структуру:

{ "створено": позначка часу,

"корисне навантаження": об'єкт,

"отримано": позначка часу,

"результат": об'єкт }

Тобто містять дату і час створення (внесення до бази даних) – «створено»; відмітка часу моменту прийому пристроєм – «отримано»; об'єкт з довільним набором полів, що описують команду - "корисне навантаження"; і об'єкт з

динамічною структурою, який описує результат виконання пристроєм команди - “result”.

Розробка блок схем та алгоритмів роботи

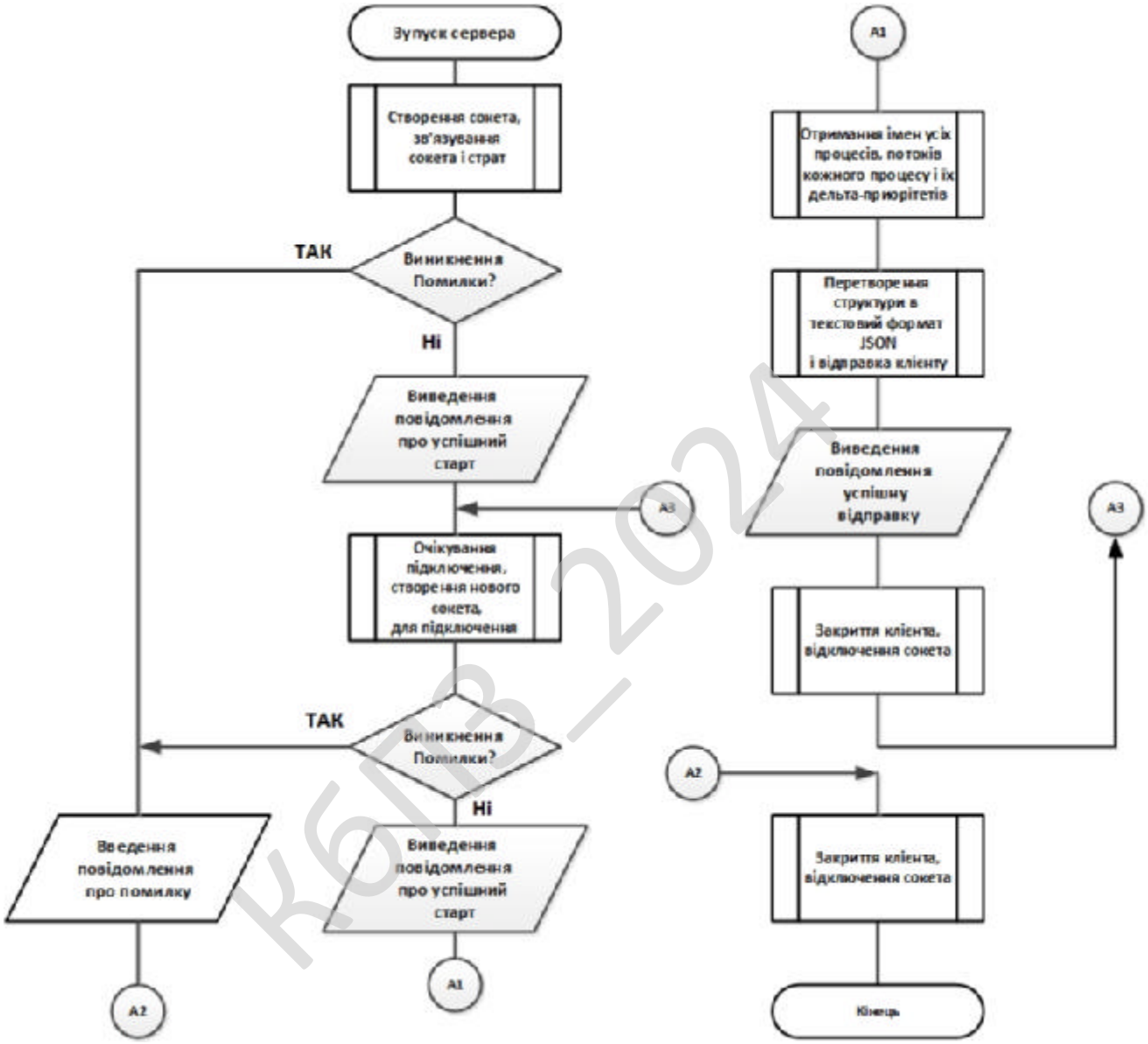

Рисунок 4.8 - Блок-схему алгоритму підключення клієнта

Розглянемо алгоритм роботи основної програми. Його блок-схема зображена на рисунку 4.9.

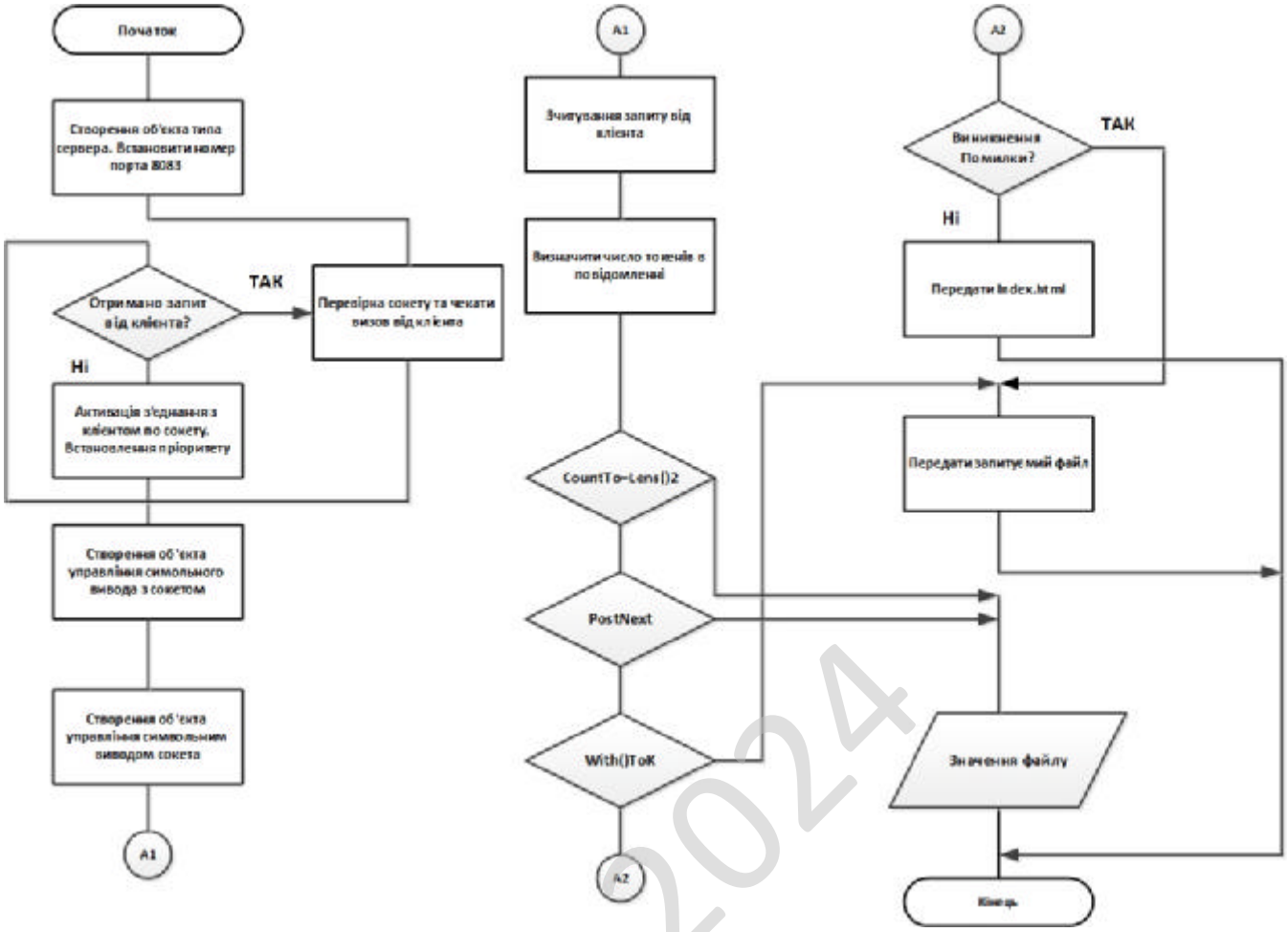

Рисунок 4.9 – Алгоритм роботи основної програми

4.2 Захист розробленого програмного забезпечення

Підсистема зберігання адміністративних даних. Організація, вимоги та обґрунтування вибору стороннього рішення для захисту інформації в ІБ

Платформа дозволяє передавати інформацію про стан від пристроїв до адміністраторів і команди від адміністраторів до пристроїв. Крім того, повідомлення про статус, призначені для адміністраторів, можуть генеруватися за допомогою тригерів відстеження. Як описано в розділі «Архітектура платформи», ці дані мають бути безпечно збережені перед передачею одержувачу.

Ця база даних також зберігає інформацію про автентифікацію/авторизацію (тобто права доступу користувача та пристрою) і дані пристрою (його тип, місцезнаходження та іншу інформацію).

Для безпеки паролі зберігаються в хешованій формі за допомогою алгоритму Bcrypt. Bcript - це функція хешування паролів, розроблена Нільсом Провосом і Девідом Мазьєром [26], яка базується на шифрі Blowfish.

На додаток до використання солі для захисту від атак веселкової таблиці, bscript є масштабованим: з часом кількість ітерацій можна збільшити, щоб сповільнити його обчислення, тому він залишається стійким до атак грубої сили, навіть коли кількість доступних обчислювальних ресурсів збільшується .

Опишемо вимоги до адміністративної бази даних.

Надійність зберігання. Якщо база даних підтверджує виконання запиту на запис, ми повинні бути впевнені, що дані збережені в постійному сховищі, тобто вони не будуть втрачені після збою сервера.

Сильна консистенція. Якщо адміністративна база даних є розподіленою, то після успішного запису певного значення операція читання повинна повернути те саме значення.

Тиражування. Мінімізуйте час простою бази даних у разі збою сервера, утримуючи головний сервер-репліку в режимі гарячого очікування та переключаючись на нього. Важливо для зберігання інформації про стан: пристрої можуть повторювати запити протягом деякого часу (якщо вони важливі), але їхня ємність пам’яті сильно обмежена.

Мова функціональних запитів. Сховище адміністративних даних містить інформацію про пристрої, які можна використовувати для виконання аналітичних запитів. Наприклад, вам може знадобитися відфільтрувати пристрої за певним критерієм. Підтримка геопросторових запитів також є обов’язковою. Оскільки платформа націлена на збір і аналіз даних з великої кількості просторово розподілених пристроїв, геопросторові запити необхідні для виконання аналітичних запитів, наприклад, пошуку пристроїв у певному радіусі

заданих координат. Повнотекстовий пошук (повнотекстовий пошук) також буде корисний при аналізі даних, що містяться в адміністративній базі даних. Наприклад, ви можете шукати повідомлення про статус, які містять певне слово.

Ефективні операції видалення. З описаних типів даних, що зберігаються в цій базі даних, найбільший обсяг мають повідомлення про стан і команди. Але більшість із них після отримання можна видалити. Тому база даних повинна підтримувати ефективні операції видалення, які є швидкими та звільняють місце на диску.

Динамічна структура даних. Зручно мати довільну структуру даних, щоб спростити адаптацію до мінливих форматів, які використовуються пристроями.

Реляційні бази даних загалом задовольняють вимогам (1)-(3). Вони традиційно забезпечують значні гарантії зберігання даних, а в нових версіях (наприклад, PostgreSQL 9.0+) мають підтримку режиму гарячого очікування.

Мова запитів SQL має широкі аналітичні можливості, але стандарт не описує геопросторові запити, тому вони доступні лише як розширення в деяких базах даних, а їх можливості дещо обмежені (наприклад, у MySQL [27]). Можливості повнотекстового пошуку також не дуже добре розвинені в реляційних базах даних, тому для цього потрібно використовувати окреме рішення.

Крім того, реляційні бази даних видаляють дані неефективно: такі операції дуже повільні, а звільнення місця на диску ще довше і складніше. Наприклад, звільнення місця в таблиці InnoDB (MiSKL) вимагає повторного створення таблиці (тобто створення нової таблиці, копіювання даних зі старої в нову та видалення старої) [27]. Більше ніж інші рішення, MongoDB відповідає нашим вимогам. Розглянемо, як це влаштовано і чому підходить.

MongoDB – база даних для зберігання адміністративних даних

MongoDB - це кросплатформна документоорієнтована база даних. Класифікований як NoSQL, MongoDB використовує динамічно структуровані

JSON-подібні документи (формат називається BSON) замість традиційних табличних структур реляційних баз даних, що робить інтеграцію даних швидшою. Розроблено MongoDB Inc. і випускається як безкоштовне програмне забезпечення з відкритим вихідним кодом під комбінацією GNU Affero General Public License та Apache License.

Станом на липень 2024 року MongoDB є четвертою за популярністю СУБД і найпопулярнішим документоорієнтованим сховищем. Давайте розглянемо основні функції MongoDB: Спеціальні запити. Підтримуються запити полів, діапазонів і регулярних виразів. Запити можуть читати окремі поля документа та включати визначені користувачем функції JavaScript, які виконуються безпосередньо сервером. Індексація. У документі можна індексувати будь-яке поле, включаючи масиви та вкладені документи (індекси в MongoDB концептуально подібні до індексів у реляційній СУБД). Доступні первинний і вторинний індекси. тиражування. Наявність підтримується наборами реплік. Набір реплік складається з двох або більше копій даних. Кожен такий пул може діяти як первинна або вторинна репліка. За замовчуванням основний виконує всі записи та читання, що означає, що забезпечується надійна узгодженість. Вторинні зберігають копію даних первинної репліки. Якщо основна репліка виходить з ладу, набір реплік автоматично вибирає, яка з вторинних реплік стане основною. Вторинні репліки можуть (необов'язково) виконувати операції читання, але дані будуть узгодженими в остаточному обліковому записі.

Балансування навантаження. MongoDB масштабується горизонтально за допомогою сегментування. Користувач вибирає ключ, який визначає, як дані розподіляються в колекції. Вони поділені на діапазони та розподілені по кількох серверах. Ключ шардингу також можна хешувати для рівномірного розподілу. Агрегація. Це дозволяє вам виконувати запити, подібні до SQL GROUP BY. Оператори агрегації можуть бути об'єднані разом як канали Unix. Також є пошуковий оператор, який об'єднує документи.

Швидкі операції видалення. MongoDB зберігає дані у подвійно зв'язаному списку, тому для видалення потрібно лише змінити два покажчики. Однак автоматичного стиснення немає, тому дисковий простір не звільняється автоматично, а реалізується спеціальною утилітою. Лімітовані колекції. Він підтримує колекції фіксованого розміру, які називаються обмеженими колекціями.

Вони підтримують порядок вставки, і коли заданий розмір досягається, вони поводяться як циклічна черга. Це може бути корисно для повідомлень про стан і команд, наприклад, щоб зберігати не більше ніж указану кількість записів. Використання таких колекцій може бути набагато ефективнішим, ніж регулярне видалення записів. Відсутність підтримки багатодокументних операцій. Транзакції ACID підтримуються лише на рівні документа. Але, як буде показано пізніше, структура документа, яка буде використовуватися, не вимагає багатодокументних операцій.

Динамічна структура. Оскільки значення в документах зберігаються разом з іменами полів, модель даних може змінюватися динамічно. Повнотекстовий пошук. Підтримується широкий спектр можливостей повнотекстового пошуку, для яких потрібен текстовий індекс.

Геопросторові запити. При наявності відповідного індексу можна виконувати різні геопросторові запити, наприклад, шукати точки на заданій відстані від заданої. При цьому обчислення ведуться з великою точністю, тому що відстань, наприклад, відраховується від точки на сфері, тобто враховується кривизна Землі.

Організація аутентифікації. Веб-маркер JSON

Автентифікація потрібна для запитів, що надходять від пристрою та інших користувачів системи (адміністраторів, аналітиків тощо). Необхідно забезпечити значну безпеку системи, яка, однак, дозволяє підтримувати високу пропускну здатність обробки запитів. Давайте виберемо JSON Web Token (JVT) - сучасний інструмент, який має багато переваг перед традиційними методами

аутентифікації. JVT - це відкритий стандарт [28], який визначає компактний і самодостатній спосіб безпечної передачі інформації між сторонами у формі об’єкта JSON. Цю інформацію можна перевірити, оскільки вона має цифровий підпис. JVT можна підписувати за допомогою секретного слова (алгоритм HMAC) або пари публічно-приватних ключів (RSA). Через невеликий розмір JVT можна передати в URL-адресі, у параметрі POST або в заголовку HTTP. Крім того, компактність збільшує швидкість передачі. Самодостатність означає, що токен містить всю необхідну інформацію про користувача, що дозволяє не робити запити до бази більше одного разу. JVT складається з трьох частин:

- назва. Він містить дві частини: тип маркера (JVT) і використовуваний алгоритм хешування (наприклад, HMAC SHA256 або RSA). Заголовок кодується за допомогою Base64Url і представляє першу частину JVT;

- вантажопідйомність. Містить інформацію про користувача та додаткові метадані. Є зарезервовані, публічні та приватні поля. Корисне навантаження також кодується Base64Url і представляє другу частину JVT;

- для створення підпису закодований заголовок і корисне навантаження, а також секретне слово беруться та підписуються за допомогою алгоритму, зазначеного в заголовку. Підпис використовується для того, щоб переконатися, що відправник JVT є тим, за кого себе видає, і що повідомлення не було змінено на шляху до сервера.

Під час використання JVT для автентифікації, коли користувач успішно входить із своїми обліковими даними, сервер повертає веб-токен JSON, який клієнт повинен зберігати локально. Цей підхід замінює традиційний, коли сеанс створюється на сервері, а його ідентифікатор повертається клієнту. Коли користувачеві потрібно отримати доступ до захищеної адреси або ресурсу, він надсилає JVT, наприклад, у HTTP-заголовку «Авторизація». JVT — це механізм аутентифікації без стану, тому що стан користувача не потрібно зберігати на сервері, токен містить всю необхідну інформацію. JVT має передаватися через безпечне з’єднання за допомогою, наприклад, HTTPS.

Це пов’язано з тим, що криптографічний алгоритм лише забезпечує перевірку того, що токен дійсно був згенерований сервером у тому вигляді, в якому він був отриманий.

Коли ви використовуєте відкритий протокол, третя сторона може легко побачити вміст повідомлення. За цією схемою пристрої надсилають свій ідентифікатор і пароль на спеціальну адресу для підключення до платформи. Сервер перевіряє в адміністративній базі даних, чи знає він про такий пристрій і чи повинен він з ним працювати. У разі позитивного рішення сервер надсилає на пристрій JVT, який записує додаткову інформацію про пристрій та його права доступу. Також доцільно включити тип пристрою в його корисне навантаження при формуванні JVT. Потім, щоб надіслати дані на сервер, клієнтське програмне забезпечення пристрою додає веб-токен JSON до HTTP-заголовка кожного запиту.

Коли він отримує запит, що містить маркер, сервер лише перевіряє його цифровий підпис за допомогою відповідного алгоритму. Маркер містить усю необхідну інформацію: ідентифікатор пристрою, який вказує, де були отримані певні дані, і тип пристрою, за допомогою якого зовнішній сервер може надсилати дані - надішліть повідомлення у відповідну тему в черзі повідомлень. Тому для обробки повідомлення не потрібно робити додаткові запити до бази даних; це значно збільшує пропускну здатність системи.

5 ВПРОВАДЖЕННЯ СИСТЕМИ В ПРОМИСЛОВУ ЕКСПЛУАТАЦІЮ

Схема розміщення системи

Щоб розгорнути платформу, ви можете вибрати один із хмарних хостів, наприклад Amazon Web Services або Google Cloud Platform. Але динамічне масштабування сервісів вимагає спеціальних рішень, які забезпечують роботу системи в цілому. Перш за все, коли потрібно розгорнути більше одного фронтенд-сервера, виникає необхідність розгорнути балансувальник навантаження, тобто інструмент для розподілу навантаження на кілька серверів.

Балансування навантаження

Для балансування навантаження ми будемо використовувати HAProki. HAProki - це серверне програмне забезпечення для забезпечення високої доступності та балансування навантаження для програм TCP і HTTP шляхом розподілу вхідних запитів між декількома серверами [29]. Програма написана мовою S.

HAProki є відкритим вихідним кодом і розповсюджується за загальною публічною ліцензією GNU (GNU GPL v2). Особливості HAProki:

- періодична перевірка доступності внутрішніх серверів, на які пересилаються запити користувачів. Дуже корисно, оскільки ця функція виключає сервер із пулу, до якого розповсюджуються запити, як тільки він перестає бути доступним з будь-якої причини;

- підтримка HTTPS (TLS) і HTTP/2. Як уже було сказано, захищений (TLS) варіант HTTP/2 є основним протоколом зв’язку клієнтського програмного забезпечення з зовнішніми серверами;

- підтримка IPv6, стиснення HTTP (deflate, gzip, libslz), постійне з'єднання HTTP;

- підтримка WebSocket. Що стосується продуктивності, то з тестів відомо, що HAProki на серверах, оснащених процесором Xeon E5 (2014 року випуску), здатний забезпечити потік даних до 40-60 Гбіт/с. Він розроблений якомога ефективнішим і використовує мінімальний час ЦП, тому продуктивність мережевої карти наразі є обмежуючим фактором, а не ЦП чи програмний балансувальник навантаження.

Коли ви досягаєте певного ліміту продуктивності HAProki, ви можете реалізувати ще більш ефективне рішення, яке масштабується майже необмежено – балансування на рівні DNS. HAProki надає можливість відновлювати сесії (закріплення сесії), що дозволяє надсилати запити конкретного клієнта на відповідний сервер, який обслуговує свою сесію в сесії. У балансуванні DNS такої функції немає, але платформі вона не потрібна: уся архітектура, включаючи автентифікацію, побудована так, щоб не зберігати стан між запитами.

Таким чином, ви можете реалізувати найпростіший алгоритм Round Robin DNS, який пересилає запити один за одним на сервери зі списку.

Пошук сервісів

Якщо конкретне клієнтське програмне забезпечення викликає певний сервер, йому, очевидно, потрібно знати його IP-адресу та порт для цього. У традиційних програмах, які працюють на невеликій кількості власних серверів компанії, мережеві розташування служб досить статичні. Їх можна прочитати з деякого файлу конфігурації, який час від часу оновлюється. У сучасних хмарних системах з мікросервісною архітектурою дізнатися IP і порт потрібного сервісу набагато складніше.

Мережеві адреси призначаються серверам динамічно. Крім того, набір послуг постійно змінюється через автомасштабування або збої. Тому необхідний більш складний механізм пошуку послуг. Відмінне рішення - Consul. Consul – розподілена система конфігурації та пошуку сервісу, яка має такі можливості [32]:

- пошук служб через DNS або HTTP. Деякі клієнти Consul надають (реєструють) послугу, а інші можуть подати запит і знайти її;

- Огляд здоров'я. Така інформація може використовуватися для моніторингу працездатності кластера або для перенаправлення трафіку з сервера, що має проблеми;

- сховище ключ/значення. Його можна використовувати для налаштування або налаштування служб;

- підтримка багатьох ЦОД. На кожному вузлі, який надає консульські послуги, працює агент. Він відповідає за тестування продуктивності. Агенти спілкуються з одним або кількома серверами Consul.

Дані зберігаються та тиражуються на серверах. З реплік вибирають головну. Consul може працювати з одним сервером, але щоб уникнути сценаріїв відмови сервера, які призводять до втрати даних, рекомендується використовувати від 3 до 5. Кожен центр обробки даних повинен мати окремий кластер Consul.

Будь-який компонент інфраструктури, якому потрібно знайти службу, може зробити запит до будь-якого сервера або агента. Агенти автоматично перенаправляють запити на сервери. Consul вимагає кворуму реплік для генерації відповідей на запит, щоб забезпечити надійну узгодженість. При наявності мережевого поділу Consul жертвує доступністю, тобто, з точки зору теореми CAP, це система CP.

Порівняно з іншими подібними рішеннями, головною перевагою Consul є те, як він виконує перевірку працездатності сервера. Більшість альтернативних систем надсилають мережеві запити службам, щоб визначити їхній статус. Такі перевірки надають набагато менше інформації, ніж консульські агенти, які працюють на кожному сервері, і мають можливість збирати набагато детальнішу інформацію про систему.

Традиційні реалізації перевірки працездатності також вимагають обсягу роботи, який лінійно залежить від кількості вузлів у кластері, тому погано масштабується.

Клієнти Consul використовують протокол gossip для передачі даних про стан сервера, що дозволяє масштабувати кластери будь-якого розміру, не зосереджуючи основне навантаження на певній групі серверів.

Розгортання і масштабування

Рішення щодо масштабування можна приймати за допомогою спеціальних інструментів хмарного хостингу, таких як Amazon Auto Scaling [30]. Amazon Auto Scaling може запускати нові сервери, коли перевищено певний поріг навантаження ЦП або використання оперативної пам'яті. Крім того, цей інструмент може зупинити частину сервера, якщо кластер споживає мало ресурсів. Після запуску сервера (початкового, в результаті масштабування або при відновленні після збою) необхідно встановити на цей сервер необхідне програмне забезпечення і запустити відповідну службу, тобто якийсь виконуваний код.

Існує два основних підходи до налаштування:

- Push розгортання. Після запуску сервера на нього передається необхідний виконуваний код, наприклад, за допомогою rsync (Unik);

- Витягнути розгортання. Сервер запускається з певною конфігурацією, яка визначає, звідки можна завантажити виконуваний код. Сервер запитує його з певного центрального сховища, збирає та запускає.

Push-розгортання більше підходить для невеликих кластерів фіксованого розміру. Динамічне масштабування вимагає підходу залучення, розміщення виконуваного коду служби в центральному репозиторії, щоб, коли сервер автоматично запускається, він сам розгортав службу. Щоб запустити службу, ви також повинні встановити необхідні залежності, такі як JRE для програм Java. Залежностей може бути багато, тому цей аспект розгортання також має бути

автоматизованим. Зручним і функціональним рішенням для розгортання автоматизації є Docker.

Docker - це утиліта з відкритим кодом, яка автоматизує розгортання додатків у програмних контейнерах, забезпечуючи додатковий рівень абстракції та автоматизації віртуалізації на рівні ОС у Linux [31].

Docker реалізує API високого рівня для легких контейнерів, які ізольовано запускають процеси. Створений для використання можливостей, наданих ядром Linux (інструменти контрольних груп і просторів імен), контейнер Docker, на відміну від віртуальної машини, не вимагає (і не містить) окремої операційної системи. Замість цього він використовує ізоляцію ресурсів (ЦП, пам’ять, введення/виведення, мережу тощо) і окремі простори імен для ізоляції програми в операційній системі. Це дозволяє створити розділ операційної системи для програм, який має власний простір ідентифікатора процесу, а також структуру файлової системи та мережеві інтерфейси. Різні контейнери мають одне ядро, але кожен з них може бути обмежений у використанні ресурсів.

Використання Docker для створення контейнерів і керування ними спрощує створення розподілених систем, дозволяючи програмам працювати автономно на окремих серверах або в кластері. Екосистема Docker, окрім основного модуля Docker Engine, включає такі компоненти:

- Docker Compose – дозволяє легко описати та запустити конфігурацію багатоконтейнерної програми з усіма її залежностями;

- Docker Swarm – інструмент управління кластером, який дозволяє логічно об'єднати кілька движків Docker в один;

- Docker Registries – утиліта для зберігання та розповсюдження образів Docker. Корисно для організації відкатів - сервери завантажуватимуть зображення з цього реєстру. Покажемо схему розміщення системи.

Consul надає послуги моніторингу та пошуку зовнішніх серверів, а Zookeeper потрібен для координації внутрішніх вузлів Kafka. Робота MongoDB

забезпечується одним головним сервером і одним або двома підлеглими серверами. Розташування Спарка і Кассандри також є важливим аспектом. Оскільки більшість даних для виконання обчислень береться з оперативної бази даних, рекомендується запускати Spark Worker і вузол Cassandra на одному фізичному сервері. Spark, а також драйвер для його підключення до Cassandra (Spark Cassandra Connector) мають широкі можливості для організації обчислень таким чином, щоб підвищити локалізацію даних, тобто максимально зменшити рух даних мережею.

6 НАУКОВА НОВИЗНА

У магістерській роботі розроблено програмне забезпечення, яке призначено для побудови системи платформи Інтернету речей.

Метою розробки є дослідження та програмна реалізація розподіленої сервісно-орієнтованої платформи Інтеренету речей яка забезпечить високу продуктивність і захист інформації.

Об'єктом дослідження є процес побудови платформи Інтернету речей.

Предметом дослідження є методи реалізації серверних систем платформ Інтернету речей.

Методи дослідження базуються на методах теорії кодування, методах математичної статистики,методах захисту інформації, методах розробки програмного забезпечення та методах побудови та обслуговування мережних пристроїв.

Наукова новизна отриманих результатів. У процесі рішення завдань, обумовлених цілями дослідження, отримані наступні результати:

– ефективна реалізація компонентів системи дає змогу оптимізувати витрати на апаратне забезпечення платформи;

– платформа також гарантує значний рівень відмовостійкості, що знижує ймовірність втрати операційних даних і забезпечує високий рівень її доступності.;

– досліджені та реалізовані аспекти безпеки надають високий рівень захищеності даних при передачі й убезпечують систему від різноманітних інформаційних атак.

7 МАРКЕТИНГОВЕ ТА ЕКОНОМІЧНЕ ОБГРУНТУВАННЯ ІТ-ПРОЄКТУ

7.1 Визначення цільової аудиторії кінцевого готового продукту

Результати дослідження та програмної реалізації системи безпеки на основі бездротових технологій можуть бути цікавими для груп і осіб, представлених на рисунку 7.1.

Підприємства: Великі та малі компанії можуть зацікавитися бездротовими системами безпеки для захисту своїх офісів, складів і виробничих потужностей.

Фінансові установи: Банки та інші фінансові організації, які мають підвищені вимоги до безпеки даних та фізичних об'єктів.

Торгові мережі: Роздрібні мережі можуть використовувати бездротові технології для моніторингу і захисту магазинів.

Правоохоронні органи: Поліція, служби безпеки та інші державні організації, які займаються охороною публічної безпеки.

Місцеві органи влади: Органи, відповідальні за охорону громадських місць, таких як парки, школи та спортивні об’єкти.

Сектори, що потребують безпеки: Промислові об’єкти, які потребують контролю за доступом і моніторингом, наприклад, заводи, нафтопереробні підприємства, склади небезпечних матеріалів.

Дослідницькі організації: Інститути, які займаються дослідженнями у сфері безпеки, технологій, телекомунікацій, можуть зацікавитися результатами для подальшого розвитку.

Користувачі домашніх систем безпеки: Фізичні особи, які бажають захистити свої домогосподарства, можуть зацікавитися рішеннями для безпеки на основі бездротових технологій.

Користувачі «розумного» дому: Споживачі, які інтегрують системи «розумного» дому, зацікавлені в рішеннях, які поєднують функції безпеки з автоматизацією.

Фірми, що надають послуги з інтеграції систем безпеки: Компанії, які спеціалізуються на інтеграції різних технологій для забезпечення безпеки.

Стартапи, що працюють у сфері ІТ та безпеки: Можуть бути зацікавлені в нових рішеннях для розробки власних продуктів або послуг.

Інвестори: Особи або фонди, які зацікавлені в інвестуванні в проекти, пов'язані з безпекою та технологіями.

Рисунок 7.1 – Цільова аудиторія

Таким чином, результати дослідження системи безпеки на основі бездротових технологій можуть мати значний інтерес для різноманітних секторів, що охоплюють як бізнес, так і приватні особи, організації та державні структури. Це відкриває можливості для комерціалізації розробок та їх впровадження у різні сфери.

7.2 Оцінка привабливості шляхом застосування методів експертних оцінок

Оцінка привабливості проєкту програмної реалізації системи безпеки на основі бездротових технологій може бути виконана шляхом застосування методів експертних оцінок. Ось приклад такого процесу:

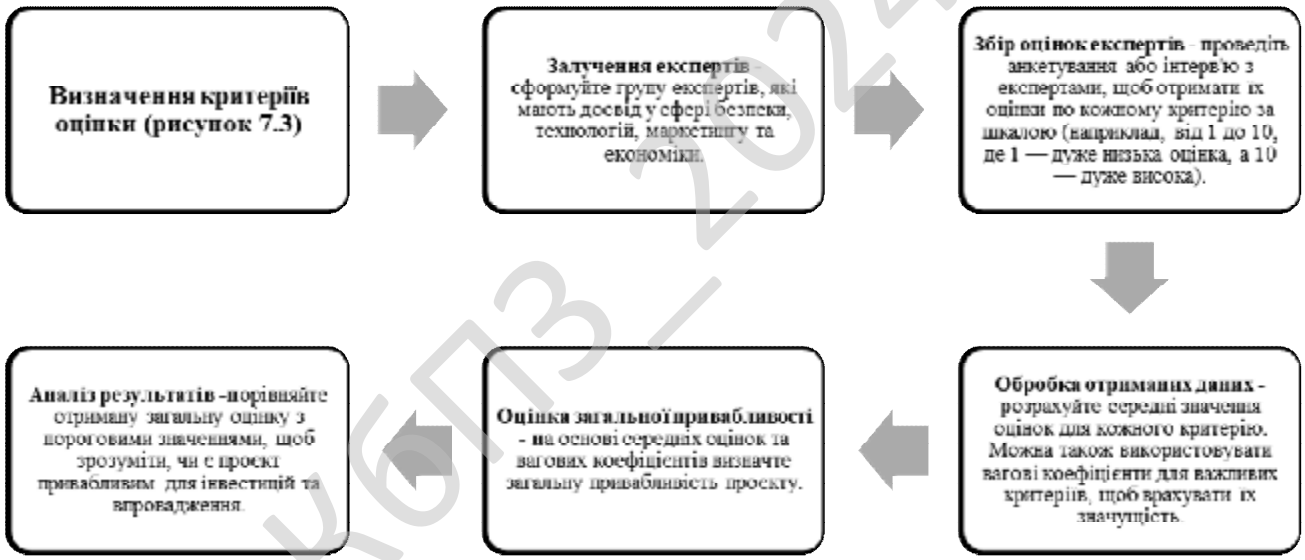

Рисунок 7.2 – Кроки для оцінки привабливості

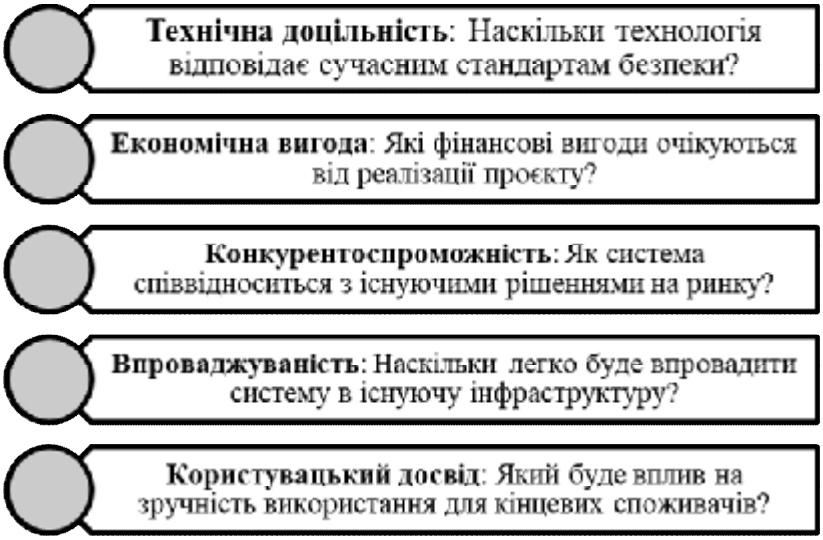

Рисунок 7.3 – Критерії експертної оцінки

Експерти оцінюють кожен критерій та враховують ваговий коефіцієнт. Результати зводимо до таблиці 7.1.

Таблиця 7.1 – Зведена таблиця

Критерій	Експерт 1	Експерт 2	Експерт 3	Середня оцінка	Вага	Вагова оцінка
Технічна доцільність	8	9	7	8	0.3	2.4
Економічна вигода	6	7	8	7	0.25	1.75
Конкурентоспроможність	7	8	6	7	0.2	1.4
Впроваджуваність	9	8	8	8.33	0.15	1.25
Користувацький досвід	8	9	8	8.33	0.1	0.83

На основі середніх оцінок та вагових коефіцієнтів визначте загальну привабливість проєкту. Загальна оцінка привабливості = 2.4 + 1.75 + 1.4 + 1.25 + 0.83 = 7.63

Результат оцінки вказує на загальну привабливість проєкту. Чим вища оцінка, тим більш привабливим є проєкт для інвестицій і реалізації. Цей підхід дозволяє систематично і об'єктивно оцінити потенціал системи безпеки на основі бездротових технологій.

7.3 Вибір методу оцінки вартості ПЗ

Для оцінки вартості програмної реалізації системи безпеки на основі бездротових технологій можна використовувати кілька методів. Часто використання комбінації цих методів дозволяє отримати більш об'єктивну та точну оцінку вартості. Наприклад, комбінуючи метод витрат і метод доходу, ви можете врахувати як витрати на розробку, так і потенційні доходи від системи. При оцінці вартості важливо враховувати ризики, пов'язані з технологіями, ринком та реалізацією проєкту.

Вибір найкращого методу залежить від конкретних умов проекту, доступних даних та цілей оцінки. Як варіант методу оцінки пропонуємо метод порівняльного аналізу та метод експертних оцінок.

Метод порівняльного аналізу оцінює вартість системи, порівнюючи її з аналогічними проектами на ринку: оцінка цін на схожі системи безпеки, які вже реалізовані, аналіз попиту на системи безпеки та бездротові технології, коригування оцінки з урахуванням особливостей вашої системи.

Метод експертних оцінок передбачає залучення експертів для оцінки вартості на основі їхнього досвіду і знань шляхом збору оцінок від групи експертів з різних галузей чи збору думок експертів для досягнення консенсусу.

7.4 Розрахунок економічної ефективності від впровадження реалізованого ПЗ як фактору його привабливості

Економічна ефективність від впровадження системи безпеки на основі бездротових технологій може бути продемонстрована через ряд показників, таких як зниження витрат на обслуговування, підвищення продуктивності, зменшення ризиків і втрат, а також повернення інвестицій (ROI).

Умовні вхідні дані та результати орієнтовного розрахунку від впровадження системи зводимо в таблицю 7.2.

Таблиця 7.2 – Основні показники впровадження проєкту

1. Витрати на впровадження системи безпеки: **Первинні витрати**: Придбання обладнання (датчики, камери, контролери) – 50,000 грн Розробка програмного забезпечення – 30,000 грн Установка та налаштування – 20,000грн **Загальні первинні витрати**: 100,000грн
2. Операційні витрати до впровадження: **Традиційна система безпеки** (наприклад, фізична охорона): Щомісячні витрати на охорону – 5,000грн Витрати на ремонт та обслуговування – 1,000грн **Річні витрати**: Щомісячні витрати × 12 = 5,000 × 12 = 60,000грн Річні витрати на обслуговування = 1,000 × 12 = 12,000грн **Загальні річні витрати**: 60,000 + 12,000 = 72,000грн
3. Операційні витрати після впровадження: **Бездротова система безпеки**: Щомісячні витрати на обслуговування (онлайн моніторинг, оновлення ПЗ) – 1,000грн Витрати на аварійні ремонти – 500грн **Річні витрати**: Щомісячні витрати × 12 = 1,000 × 12 = 12,000грн Річні витрати на обслуговування = 500 × 12 = 6,000грн **Загальні річні витрати**: 12,000 + 6,000 = 18,000грн
4. Економія: **Економія на річних витратах**: Економія=Старі річні витрати−Нові річні витрати=72,000−18,000=54,000 грн
5. Повернення інвестицій (ROI): **ROI** розраховується за формулою: ROI=(Економія−Первинні витратиПервинні витрати)×100% =−46% (В даному прикладі ROI негативний у перший рік, але в подальші роки він стане позитивним, оскільки економія продовжиться.)
6. Додаткові переваги: **Покращена безпека**: зменшення випадків крадіжок і вандалізму на 30%, що може призвести до додаткової економії. **Підвищена продуктивність**: можливість використовувати ресурси (персонал) більш ефективно, зменшуючи потребу в охоронних заходах.

Впровадження бездротової системи безпеки в довгостроковій перспективі може значно знизити витрати на обслуговування та підвищити рівень безпеки. При правильному управлінні та ефективній експлуатації система буде приносити економічну вигоду, що перевищить первинні інвестиції.

7.5 Пропозиція алгоритму просування проєкту розробки ПЗ

Алгоритм просування проєкту програмної реалізації системи безпеки на основі бездротових технологій, що включає етапи планування, реалізації та моніторингу схематично подано на рисунку 7.4.

Рисунок 7.4 – Алгоритм просування проєкту

Цей алгоритм охоплює всі ключові етапи просування проєкту програмної реалізації системи безпеки на основі бездротових технологій, від аналізу ринку до моніторингу результатів. Гнучкість і здатність адаптуватися до змін у ринковій ситуації також є важливими для успішного просування.

7.6 Оптимізація каналів збуту та шляхів реалізації ПЗ

Для оптимізації каналів збуту та шляхів реалізації проєкту програмної реалізації системи безпеки на основі бездротових технологій можна розглянути такі підходи:

– онлайн-продажі: розробка та оптимізація власного веб-сайту для продажу системи;

– інтеграція можливостей електронної комерції, таких як оплата онлайн, оформлення замовлень, підтримка користувачів через чат-боти;

– маркетплейси: використання платформ, таких як Amazon, Ebay або спеціалізовані сайти для продажу технологій безпеки. це дозволить розширити аудиторію;

– співпраця з інтеграторами систем безпеки: укладання угод з компаніями, що спеціалізуються на інтеграції технологій безпеки. вони можуть пропонувати вашу систему у своєму портфоліо;

– партнерство з дистриб'юторами: робота з дистриб'юторами, які мають налагоджені канали розповсюдження та можуть швидше виводити продукт на ринок;

– сегментація ринку: визначення ключових сегментів споживачів, наприклад, підприємства, приватні особи, установи, та створення індивідуальних пропозицій для кожного сегмента;

– персоналізація пропозицій: використання даних про споживачів для налаштування рекламних кампаній, що робить їх більш релевантними;

– знижки та спеціальні пропозиції: проведення акцій, знижок на першу покупку, безкоштовного тестового періоду або пакетних пропозицій для залучення нових клієнтів;

– реферальні програми: запровадження програм, які заохочують існуючих клієнтів рекомендувати систему своїм знайомим;

– вебінари та семінари: проведення навчальних заходів для потенційних клієнтів щодо використання системи безпеки та її переваг;

– технічна підтримка: запровадження цілодобової технічної підтримки для клієнтів, щоб допомогти вирішити проблеми з установкою та експлуатацією;

– моніторинг каналів збуту: аналіз ефективності різних каналів збуту, щоб виявити найбільш прибуткові та оптимізувати або закрити менш ефективні;

– зворотний зв'язок від клієнтів: збір відгуків від користувачів для покращення продукту та коригування маркетингових стратегій;

– постійне вдосконалення: регулярне оновлення програмного забезпечення з новими функціями на основі зворотного зв'язку від користувачів, що може підвищити його конкурентоспроможність на ринку;

– інтеграція з іншими технологіями: розширення функціоналу продукту шляхом інтеграції з іншими системами (наприклад, IoT-пристрої, мобільні додатки).

Оптимізація каналів збуту та шляхів реалізації проєкту програмної реалізації системи безпеки на основі бездротових технологій вимагатиме комплексного підходу, що включає аналіз ринку, активне просування, розвиток партнерських відносин і постійне вдосконалення продукту та обслуговування. Ці заходи допоможуть залучити нових клієнтів і підвищити рівень задоволеності існуючих.

7.7 Визначення ключових факторів успіху конкретного проєкту

Ключові фактори успіху проєкту програмної реалізації системи безпеки на основі бездротових технологій схематично подані на рисунку 7.5.

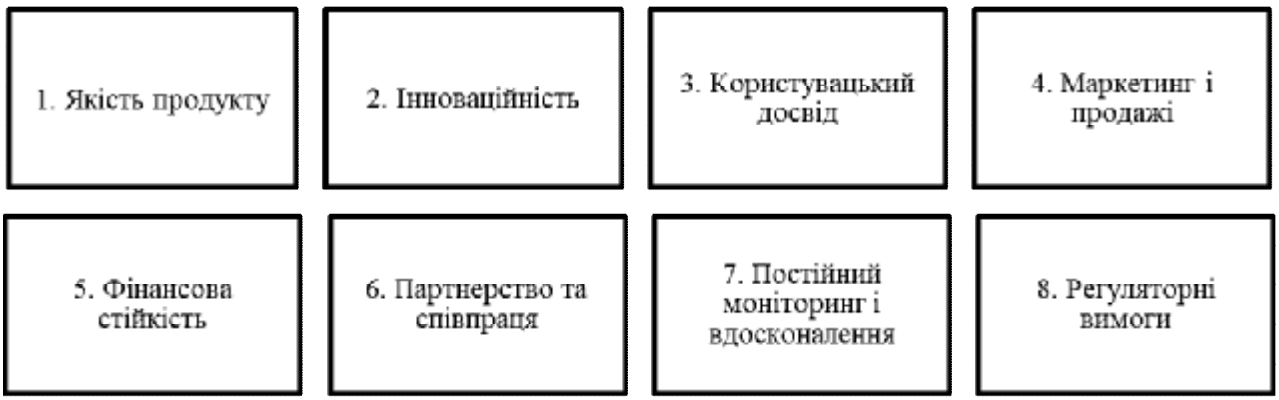

Рисунок 7.5 – Ключові фактори успіху проєкту

Успіх проєкту програмної реалізації системи безпеки на основі бездротових технологій залежить від комплексного підходу, що враховує якість продукту, інноваційність, користувацький досвід, ефективний маркетинг, фінансову стабільність, партнерство, постійний моніторинг та дотримання регуляторних вимог. Це дозволяє не лише успішно впровадити продукт, а й забезпечити його конкурентоспроможність на ринку.

8 ЗАХОДИ З ОХОРОНИ ПРАЦІ ТА ТЕХНІКИ БЕЗПЕКИ

Вступ

Протягом усієї історії людство приділяє прискіпливу увагу безпеці життя. Охорона праці є складовою частиною безпеки життя.

Законом України "Про охорону праці" регламентуються загальні положення державної політики в галузі охорони праці, а конкретизуються ці положення нормативно-правовими актами про охорону праці, зокрема Наказом Міністерства соціальної політики України 14.02.2018 № 207, який зареєстровано в Міністерстві юстиції України 25 квітня 2018 р. за №508/31960 «Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», НПАОП 0.00-1.28-10 «Правила охорони праці під час експлуатації електронно-обчислювальних машин», та ДСанПіН 3.3.2-007-98 «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин».

Програмісти у процесі роботи мають негативний вплив на органи зору, а також мають значну розумову напругою і нервово-емоційне навантаження. Руки (суглоби польців та м'язи рук) при роботі з клавіатурою мають теж істотне навантаженням. До шкідливих факторів, які впливають на робитників галузі інформаційних технологій (ІТ) спеціалісти відносять високочастотні електромагнітні коливання (випромінювання) роботи апаратної частини ЕОМ та виділення шкідливих газів.

Ці шкідливі фактори можуть привести до професійних захворювань.

При розгляді шкідливих чинників роботи програмістів та інших спеціалістів ІТ будемо керуватись наступними нормативно-правовими актами: «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин» ДСанПіН 3.3.2-007-98, та

«Правила охорони праці під час експлуатації електронно-обчислювальних машин» НПАОП 0.00-1.28-10,

Умови праці програміста включають наступні фактори:

- параметри повітряного середовища в приміщенні;
- вентиляція приміщення;
- освітлення приміщення;
- параметри повітряного середовища в приміщенні, тощо.

Щоб запропонувати заходи щодо зменшення впливу комп'ютера на організм програміста визначимо фактори, які можуть викликати професійне захворювання і впливають на працездатність програміста.

8.1 Шкідливі і небезпечні фактори при роботі з комп'ютером

Програміст працює з електронно-обчислювальною машиною (ЕОМ) та іншим обладнанням, яке є джерелом небезпеки ураження електричним струмом. Так як робота програміста характеризується істотним зоровим навантаженням, то вимагає належного освітлення. Так як програміст постійно перебуває в приміщенні, тому для комфортних умов праці в цьому приміщенні необхідно створити належний мікроклімат.

При роботі з використанням ЕОМ відзначають наступні небезпечні та шкідливі фактори:

- ризик виникнення надзвичайних ситуацій природного або штучного характеру на об'єкті або території.
- ризик виникнення пожежі;
- негативний вплив на органи зору людини;
- ризики ураження електричним струмом;
- недостатня, або надмірна освітленість робочого місця;
- монотонність праці;
- електромагнітні електромагнітні (у т.ч. високочастотні) випромінювання (коливання);

- несприятливі мікрокліматичні умови;

- нервово-емоційна напруженість праці;

- інтелектуальні навантаження;

- невідповідність ергономічних показників робочого місця діючим вимогам;

- шуми;

- статичні навантаження на кістково-м’язовий апарат;

8.2 Аналіз санітарно-гігієнічних умов праці на робочому місці програміста

Розглянемо умови праці у приміщенні, в якому працюють програмісти. Геометричні розміри приміщення наведено у таблиці 8.1.

Таблиця 8.1 - Розміри приміщення

Найменування	Значення, м
Ширина	2,4
Довжина	3
Висота	2,8

Таблиця 8.2 - Площа та обсяг приміщення, на одного працюючого

Геометрична характеристика	Одиниця виміру	Нормативне значення*	Фактичне значення
Площа, S	$м^2$	не менше 6.0	7,2
Обсяг, V	$м^3$	не менше 20.0	20,1

* Згідно ДСанПіН 3.3.2.007-98 (Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин).

У зазначеному приміщенні працює 1 людина. За даними, які наведено у табл. 8.1 та табл. 8.2, можна зробити висновок, що площа та об'єм приміщення у розрахунку на одно робоче місце програміста відповідають нормативним вимогам (Наказу Міністерства соціальної політики України № 207, від 14.02.2018 «Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», ДСанПіН 3.3.2-007-98 «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин» та НПАОП 0.00-1.28-10 «Правила охорони праці під час експлуатації електронно-обчислювальних машин»).

Температура повітря в приміщенні визначається впливом температури зовнішнього повітря і тепловою енергією, яка виділяється всередині приміщення. Джерелами виділення теплоти в даному приміщенні є електроустаткування, освітлювальні прилади, а також люди. У світлий час доби джерелом надлишкового тепла є сонячна радіація. Згідно Постанови № 42 від 01.12.1999 Головного державного санітарного лікаря України, робота, яка виконується в даному приміщенні, відноситься до категорії Іа. В цьому випадку людина витрачає енергії до 120 ккал у годину. Вологість повітря у приміщенні визначається впливом багатьох факторів, серед яких: вологість атмосферного повітря, виділення вологи людьми (при диханні та випарами з поверхні шкіри).

Мікроклімат повітряного середовища в приміщенні характеризується запиленістю та загазованістю повітря. Мікроклімат приміщення визначається діючим на організм людини поєднанням, вологості, температури, швидкості руху повітря та інтенсивності теплового випромінювання. Аналіз мікроклімату складається з визначення зазначених вище факторів і порівняння результатів із встановленими нормами.

Таблиця 8.3 - Оптимальні і фактичні значення параметрів мікроклімату

Пора року	Оптимальні для Іа			Фактичні		
	Температура, °С	Вологість, %	Швидкість повітря, м/с	Температуа, °С	Вологість %	Швидкість повітря, м/с
Холодна	22-24	40-60	0,1	22-23	41-59	0,1
Тепла	23-25	50-70	0,1	23,5-24	53-70	0,13

У таблиці 8.3 наведено оптимальні та фактичні значення параметрів мікроклімату як для категорії ваги робіт Іа, так і розглянутого приміщення. У приміщеннях, де встановлено ЕОМ, рекомендується застосування тільки оптимальних значень показників мікроклімату.

Проведений аналіз показує, що показники мікроклімату в приміщенні відповідають установленим нормам. Штучне опалення застосовується у холодний період року.

В літню пору застосовується кондиціонер.

Для боротьби з пилом робляться регулярні провітрювання та вологі прибирання приміщенні.

У приміщенні знаходяться наступні джерела шуму: принтер *HP LaserJet 1018*, електродвигуни вентиляторів ЕОМ.

Одним з найважливіших факторів, які впливають на ефективність трудової діяльності людини, та попереджають травматизм і професійні захворювання програмістів є освітлення на робочому місці.

Працю працівника, який постійно працює за комп’ютером, згідно ДБН В.2.5 – 28 – 2006 р. можна віднести до роботи з малою точністю (найменший розмір об'єкта розрізнення від 1 до 5 мм) V-го розряду зорової роботи, з великою контрастністю об'єкта розрізнення (символів на екрані дисплея), з темним тлом (під розряд зорової роботи В). Приміщення можна віднести до 1-ої групи приміщень, у яких проводиться розрізнення об'єктів зорової роботи при фіксованому напрямку лінії зору того, що працює на робочу поверхню. Для такого типу приміщень і розряду зорової роботи нормоване значення

коефіцієнта природної освітленості (КПО) робочої поверхні (при поєднаному, спільному освітленні), повинен становити не більше 1,5%, освітленість при штучному висвітленні повинна становити 300 лк. Крім того все поле зору повинне бути освітлено достатньо рівномірно - ця основна гігієнічна вимога. Так як яскраве світло на ділянці периферійного зору значно збільшує напруженість очей і, як наслідок, призводить до їх швидкої стомлюваності, ступінь освітлення приміщення і яскравість екрану комп'ютера повинні бути приблизно однаковими.

8.3 Розробка заходів з умов поліпшення охорони праці

Згідно аналізу умов праці в розглянутому приміщенні, ми одержали наступні результати:

- розмірі приміщення, у розрахунку на одному працюючого, відповідають нормативам;
- мікроклімат відповідає нормативному значенню;
- акустичні умови роботи не перевищують нормативних значеннь;

Таким чином можна припустити, що основною причиною можнивого зниження працездатності програміста є психофізіологічний фактор, тому основна пропозиція буде така: дотримання позитивної психологічної атмосфери в колективі та регламентованого режиму праці та відпочинку, організація робочого місця з урахуванням ергономічних вимог.

Рекомендовані заходи: регулярні періодичні наочні огляди персоналом шляхів для евакуації людей із приміщення, відповідно до плану евакуації (який повинен розташовуватись на видному місці у приміщенні), включення до колективного договору мінімально можливого вмісту аптечок з обов'язковою наявністю масок-клапанів, або іншого спорядження для штучного дихання. Регулярна періодична перевірка параметрів заземлення та занулення (вимірювання опору розтіканню електричного струму на землю).

Так як при ураженні електричним струмом у людини може статися фібриляція шлуночків серця, в організації бажано мати дефібрилятор і підготовлений персонал для роботи з ним.

8.4 Розрахункова частина

Початкові данні для розрахунку штучного захисного заземлення

Тип заземлення: робоче заземлення нульової точки трансформатора. Напруга — 220/380 В. Розташування заземлюючих електродів — по контуру.

Розрахунок проводиться за допустимим опором розтіканню струму заземлювача методом коефіцієнта використання заземлювачів.

Початкові дані для розрахунку захисного заземлення: тип верхнього шару грунта — чорнозем, нижнього шару грунта — глина. Умовна товщина верхнього шару грунта: *H=0,8 м*. Для захисного заземлення: застосовуються ветикальні електроди — пруток довжиною *L=2 м*. Відстань між вертикальними заземлювачами (електродами) *A*=2 *м*. Діаметр ветикальньного електрода (прутка) *D=60 мм*, Тип горизонтального заземлювача: металева полоса. Розміри перетину з'єднуючої полоси: *60 х 6 мм*. (*b=60 мм*.). Опір заземлювача, який нормується: $R_{ЗН}$ = *4 Ом*. Глибина закладення горизонтального контура заземлення *t= 0,6 м*.

Розрахунок захисного заземлення можна автоматизувати за допомогою програми, сирцевий код якої опублікован на стр.13-16 Метододичних вказівок до виконання розрахунків з викор. персон. ЕОМ IBM сумісного типу / Охорона праці. Ч. 1. Захисне заземлення / Кіровоград. ін-т с.-г. Машинобуд.; URL : http://dspace.kntu.kr.ua/jspui/handle/123456789/4358, або будь якої відповідної іншої.

Розрахунок

Відстань від центра вертикального заземлювача до поверхні землі:

T=t+L/2=0,6+2/2=1,6 м.

Еквівалентний питомий опір грунта:

$\rho_{екв} = \psi \rho_1 \rho_2 L / [\rho_1 \psi(L-H+t) + \rho_2 \psi(H-t)] = 93,75$ *Ом*

де $\psi = 1,2$ - табличне значення коефіцієнта сезонності для відповідної кліматичної зони у багатошаровому грунті (джерело: http://ftemk.mpei.ac.ru/bgd/_private/Grunt_/t_goda.htm);

$\rho_1 = 50$ *Ом*м.* - табличне значення питомого опору верхнього шару грунта (джерело: https://el-line.ru/udelnoe_soprotivlenie_grunta.shtml);

$\rho_2 = 100$ *Ом*м.* - табличне значення питомого опору нижнього шару грунта (джерело: https://zandz.com/ru/udelnoe_soprotivlenie_grunta).

Опір розтіканню електричного струму одного електрода вертикального заземлювача:

$R_O = [\rho_1/(2\pi*L)]*[ln(2L/D)+0,5ln(4T+L)/(4T-L)] = 40,6$ *Ом.*

Опір розтіканню електричного струму заземлювача, який нормується, $R=1$ якщо $\rho_{екв} < 100$ *(у нас* $\rho_{екв} = 93,75 < 100$*),* *та* $R = R_{ЗН} * \rho_{екв}/100$, *якщо* $\rho_{екв} > 100$.

Опір розтіканню електричного струму горизонтального заземлювача (полоси):

$$R_П = 0,366(\rho_{екв}\, \psi / L_П \cdot \eta_П) \lg(2 / L_П * L_П / bt) = 35,7 \text{ Ом.}$$

де $\eta_П = 0,3$ - табличне значення коефіцієнта використання горизонтального заземлювача, залежить від розташування (в ряд або по контуру) та співвідношення A/L;

довжина горизонтального заземлювача (полоси) при розташуванні заземлювачів по контуру:

$L_П = A*n = 2*16 = 32$ *м.;*

де n – ітераційна кількість вертикальних заземлювачів.

Загальний опір штучного заземлюючого пристрою розтіканню електричного струму на землю:

$$R_B = R_П R / (R_П - R) = 4,5 \text{ Ом.}$$

Враховуючи, що значення ρ_1 *та* ρ_2 по факту можуть бути меншими, розрахункове значення R_B цілком прийнятне.

Кількість вертикальних заземлювачів:

$$n = R_O \ / R_B * \eta_B = 16шт.$$

де $\eta_B = 0,6$ - табличне значення коефіцієнта використання вертикального заземлювача, залежить від розташування (в ряд або по контуру) та співвідношення A/L.

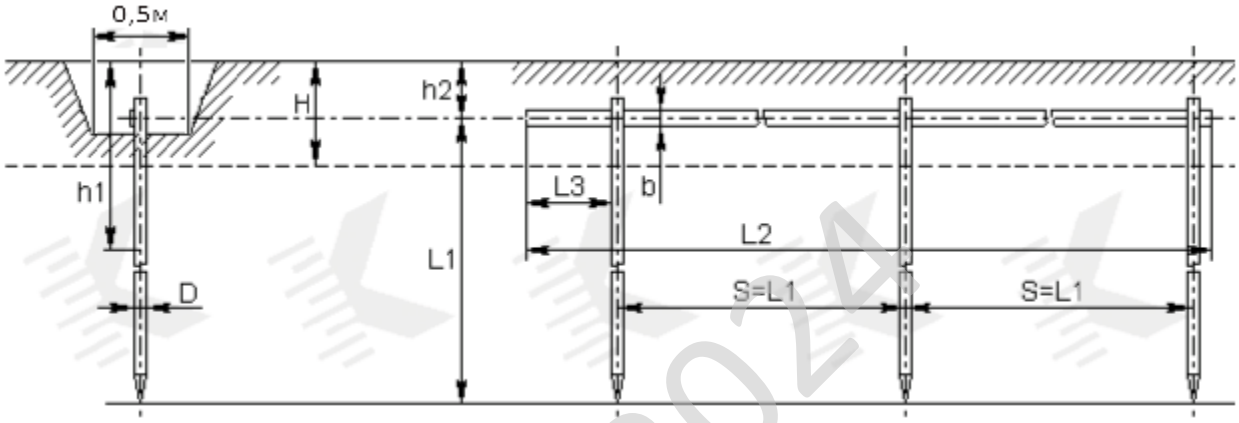

Рисунок 8.1 — Штучний заземлювач

Висновки до розділу

Дотримання всіх необхідних умов праці не лише сприяє збереженню здоров'я працівників, а також підвищує ефективність виробництва в цілому.

З цих міркувань було здійснено аналіз приміщення, призначеного для праці програмістів, проведено розгляд небезпечних та шкідливих факторів, що негативно впливають на програмістів під час роботи. Виконано розрахунок штучного освітлення, як одного з ключових факторів впливу на працездатність та здоров'я програміста. Розроблено заходи з охорони праці.

9 ОСНОВНІ ВИСНОВКИ

Програмне забезпечення, створене в результаті виконання магістерської роботи, призначено для дослідження та побудови платформ для Інтернету речей.

В межах України в недостатній мірі представлені вітчизняні розробки в цій області.

Зокрема, було виділено основні вимоги, що висуваються до подібних систем; вивчено основні підходи до вирішення задач обробки великих за обсягами даних; досліджено існуючі платформи та виокремлено шляхи їх покращення.

У магістерській роботі наведені теоретичне узагальнення й рішення наукового завдання дослідження методів .

Розроблена платформа задовольняє поставленим функціональним і нефункціональним вимогам та містить реалізацію таких підсистем: приймання, зберігання, обробки операційних даних; автентифікації та безпеки; роботи з адміністративними даними; моніторингу.

Це дозволяє застосування її такими виділеними групами користувачів: пристроями, адміністраторами, аналітиками і адміністраторами безпеки. Було запропоновано та реалізовано таку розподілену сервісно-орієнтовану (мікросервісну) архітектуру платформи, яка забезпечує високу продуктивність і майже нескінченну масштабованість. Це дозволяє нарощувати потужність системи для обслуговування мереж пристроїв будь-яких розмірів і для збільшення можливостей підсистеми аналітики даних.

Ефективна реалізація компонентів системи дає змогу оптимізувати витрати на апаратне забезпечення платформи.

Платформа також гарантує значний рівень відмовостійкості, що знижує ймовірність втрати операційних даних і забезпечує високий рівень її доступності.

Досліджені та реалізовані аспекти безпеки надають достатній рівень захищеності даних при передачі й убезпечують систему від різноманітних інформаційних атак.

У ході розробки платформи були з належним обґрунтуванням відібрані та застосовані сучасні системи, протоколи, формати даних, бібліотеки та бази даних.

Використанні сторонні рішення мають відкритий вихідний код, що спрощує розширення системи, має економічні та інші переваги.

Було детально описано різноманітні аспекти обраних архітектурних рішень: від обґрунтування високорівневого поділу на функціональні модулі до особливостей ефективної реалізації вводу/виводу та впливу обраної схеми автентифікації на масштабованість архітектури платформи.

На відміну від існуючих платформ, розроблене рішення є добре розширюваним у тому сенсі, що його можливо легко адаптувати під аналітику даних, які надходять від різноманітних пристроїв і сенсорів, задаючи ланцюжки обробки даних. Протоколи та формати даних було обрано не лише з огляду на їх ефективність, а ще й враховуючи зручність і простоту розширення платформи.

Іншими ключовими особливостями платформи є моніторинг і функціональність надійної передачі службової інформації між пристроями та адміністраторами. Ці можливості дозволяють ефективно керувати мережею пристроїв, а також гнучко налаштовувати інструменти попередження про можливі неполадки та оперативно на них реагувати.

Запропоновані та застосовані в роботі архітектурні та технічні рішення є достатньо універсальними, що дозволяє використовувати їх для побудови різноманітних систем обробки великих даних та інших платформ для Інтернету

речей, спеціалізованих для більш вузьких областей. Роботу платформи було показано за допомогою демонстраційного прикладу, в якому моделюється робота системи з мережею метеорологічних станцій.

Також до платформи було підключено мікрокомп'ютер Raspberry Pi 3 та продемонстровано обробку даних із його сенсора. Ці приклади охоплюють більшість можливостей системи та показують її застосування всіма групами користувачів.

Описано реалізацію на боці клієнта процесів реєстрації та автентифікації пристрою, а також організацію відправки даних до сервера. Розроблене програмне забезпечення має простий, дружній та зручний інтерфейс користувача, що забезпечує легкість у освоєнні роботи програмного продукту, зручність у використанні, і не потребує особливих спеціальних знань.

При створені програмного забезпечення було використано об'єктно-орієнтований підхід, що відповідає сучасним тенденціям у галузі розробки комерційних програмних систем.

Програма реалізована на мові високого рівня Java та Python. Дані мови програмування дозволяю найбільш ефективно обробляти дані. Це дозволило мінімізувати строк розробки програмного забезпечення, і, як слід, зменшити витрати на його розробку.

Запропоноване програмне забезпечення ділиться на загальне програмне забезпечення, що поставляється із засобами обчислювальної техніки й спеціальне програмне забезпечення, що спеціально розроблене для даної конкретної системи й включає програми, що реалізують її функції.

Даються необхідні рекомендації з установки розробленого програмного забезпечення.

В цілому створене програмне забезпечення підтверджує правильність використаних проектних рішень та повністю відповідає вимогам технічного завдання. Створене програмне забезпечення має потенційну можливість для подальшого вдосконалення і застосування у різних галузях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Доповідь компанії Gartner. – Режим доступу: http://www.gartner.com/newsroom/id/3165317. – Дата доступу: 22.09.2023.

2. Доповідь компанії Cisco. – Режим доступу: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoE_Economy.pdf. – Дата доступу: 23.05.2023.

3. Nathan Marz. Big Data: Principles and best practices of scalable realtime data systems / Nathan Marz, James Warren. – Manning Publications, 2015. – 328 p.

4. Arvind Sathi. Big Data Analytics: Disruptive Technologies for Changing the Game / Arvind Sathi. – Mc Press, 2012. – 96 p.

5. Vijay K. Garg. Elements of Distributed Computing / Vijay K. Garg. – Wiley-IEEE Press, 2002. – 448 p.

6. Gerard Tel. Introduction to Distributed Algorithms / Gerard Tel. – Cambridge University Press, 2000. – 612 p.

7. M. Tamer Özsu. Principles of Distributed Database Systems / M. Tamer Özsu, Patrick Valduriez. – Springer, 2011. – 846 p.

8. Refactoring: Improving the Design of Existing Code / [Martin Fowler, Kent Beck, John Brant et al.]. – Addison-Wesley Professional, 1999. – 464 p.

9. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship / Robert C. Martin. – Prentice Hall, 2008. – 464 p.

10. Design Patterns: Elements of Reusable Object-Oriented Software / [Erich Gamma, Richard Helm, Ralph Johnson et al.]. – Addison-Wesley Professional, 1994. – 395 p.

11. Martin Fowler. Patterns of Enterprise Application Architecture / Martin Fowler. – Addison-Wesley Professional, 2002. – 560 p.

12. Len Bass. Software Architecture in Practice / Len Bass, Paul Clements, Rick Kazman. – Addison-Wesley Professional, 2012. – 640 p.

13. Luke Hohmann. Beyond Software Architecture: Creating and Sustaining

Winning Solutions / Luke Hohmann. – Addison-Wesley Professional, 2003. –352 p.

14. Інформація про платформу AWS IoT. – Режим доступу: https://aws.amazon.com/iot/how-it-works/. – Дата доступу: 24.05.2016.

15. Інформація про платформу IBM IoT Platform. – Режим доступу: http://www.ibm.com/internet-of-things/iot-platform.html. – Дата доступу: 25.05.2016.

16. Інформація про платформу Oracle IoT. – Режим доступу: https://cloud.oracle.com/iot. – Дата доступу: 25.05.2016.

17. Специфікація HTTP/2 (Internet Engineering Task Force. Request for Comments 7540). – Режим доступу: https://tools.ietf.org/html/rfc7540. –Дата доступу: 26.05.2023.

18. Специфікація WebSocket (Internet Engineering Task Force. Request for Comments 6455). – Режим доступу: https://tools.ietf.org/html/rfc6455. –Дата доступу: 27.05.2023.

19. Sam Newman. Building Microservices: Designing Fine-Grained Systems / Sam Newman. – O'Reilly Media, 2015. – 280 p.

20. Офіційна документація Apache Kafka. – Режим доступу: http://kafka.apache.org/documentation.html. – Дата доступу: 29.05.2023.

21. Офіційна документація Apache Spark. – Режим доступу: http://spark.apache.org/docs/latest/. – Дата доступу: 31.05.2023.

22. Інформація про Sort Benchmark і його результати. – Режим доступу: http://sortbenchmark.org/. – Дата доступу: 31.05.2023.

23. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing / [Matei Zaharia, Mosharaf Chowdhury, Tathagata Das et al.]. – Режим доступу: https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf. – Дата доступу: 31.05.2023.

24. Офіційна документація Apache Cassandra. – Режим доступу: https://wiki.apache.org/cassandra/. – Дата доступу: 02.06.2023.

25. Dynamo: Amazon's Highly Available Key-value Store / [Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al.]. – Режим доступу:

http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf. – Дата доступу: 02.06.2023.

26. Niels Provos. A Future-Adaptable Password Scheme / Niels Provos, David Mazières // Proceedings of 1999 USENIX Annual Technical Conference. –1999. – P. 81-92.

27. Офіційна документація MySQL. – Режим доступу: http://dev.mysql.com/doc/refman/5.7/en/. – Дата доступу: 02.06.2023.

28 .Специфікація JSON Web Token (Internet Engineering Task Force. Request for Comments 7519). – Режим доступу: https://tools.ietf.org/html/rfc7519. – Дата доступу: 03.06.2023.

29. Офіційна документація HAProxy. – Режим доступу: http://www.haproxy.org/. – Дата доступу: 04.06.2023.

30. Офіційна документація Amazon Web Services. – Режим доступу: https://aws.amazon.com/documentation/. – Дата доступу: 04.06.2023.

31. Офіційна документація Docker. – Режим доступу: https://www.docker.com/. – Дата доступу: 04.06.2023.

32.Офіційна документація Consul. – Режим доступу: https://www.consul.io/. – Дата доступу: 04.06.2023

33. IoT Based Home Security System Using Raspberry Pi with Email and Voice Alert. Reena Rani, S. Lavanya,. B.Poojitha Published Computer Science, 2018 pp 21-26

34. Home Security System using IOT and AWS Cloud Services.Mahendra Mehra; Vedant Sahai; Pratik Chowdhury; Elvis Dsouza. International Conference on Advances in Computing, Communication and Control (ICAC3), 2019 pp 159-163

35. Anitha A, “Home security system using internet of things”, IOP Conf. Series: Materials Science and Engineering 263, 2017 pp 1-11

36. OneM2M Architecture Based Secure MQTT Binding in Mbed OS. Ahsan Muhammad, Bilal Afzal, Bilal Imran, Asim Tanwir, Ali Hammad Akbar, Ghalib A. Shah. Computer Science. IEEE European Symposium on Security and Privacy, 2019

pp 48-56.

37. Heapster: Compute Resource Usage Analysis and Monitoring of Container Clusters [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/kubernetes/heapster

38. Performance Testing Guidance for Web Applications by Microsoft Corporation Microsoft Corporation., 2007. – 221 с. – (Microsoft Corporation).

39. The httperf HTTP load generator [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/httperf/httperf.

40. Docker [Електронний ресурс] / Wikipedia – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Docker.

41. Стек MEAN [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/MEAN_(веброзробка)

42. Веб-технології для розробників [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/uk/docs/Web

43. jQuery [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/JQuery

44. Linux [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Linux

45. TypeScript [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/

46. SQL [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/SQL

47. Postman [Електронний ресурс] – Режим доступу до ресурсу: https://www.postman.com/

48. SQL [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/SQL

49. REST [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/REST

50. М. Кантелон , М. Хартер, Т. Головайчук, Н. Райлих // “Node.js в дії.

Додаток А
(обов'язковий)

Технічне завдання

Зміст

1 Найменування та область застосування

Це технічне завдання розповсюджується на розробку програмного забезпечення системи захисту в інформаційній системі.

2 Підстава для розробки

Підставою для розробки служить завдання на магістерську роботу, видане на кафедрі кібербезпеки та програмного забезпечення (нак. № 19-13 від 07.08.2024 року).

3 Мета та призначення розробки

Метою роботи є дослідження та програмна реалізація розподіленої сервісно-орієнтованої платформи Інтеренету речей яка забезпечить високу продуктивність і захист інформації.

4 Джерела розробки

Джерелом цієї магістерської роботи є відносна до теми література і існуючі аналоги.

5 Технічні вимоги

5.1 Склад продукції

Складниками розробки є:

– вибір і обґрунтування методів реалізації проекту;

– розробка програмної частин системи, а також розробка взаємодії системи з ОС та з користувачем;

– розробка програми, що реалізує спроектовані алгоритми роботи системи.

5.2 Показники призначення

Система повинна забезпечувати:

– Розробку додатку;

– систему підключення та тестування баз даних ;

– цілісність даних у процесі роботи та при зберіганні;

– простий, інтуїтивно зрозумілий інтерфейс.

5.3 Вимоги до функціональних характеристик

Розроблене програмне забезпечення не повинно мати обмежень на версію драйверів та операційної системи.

5.4 Вимоги до архітектури

Компонент, що розробляється повинен використовувати системні засоби та апаратні засоби, що на даному етапі розвитку обчислювальної техніки найбільше поширені.

5.5 Вимоги до надійності

Програмні модулі написані по всім правилам, які стосуються стандартних викликів процедур, функцій, методів і форм, визначених технічною документацією на середовище розробки.

5.6 Умови експлуатації

Робочі місця користувачів ПЗ повинні задовольняти наступним умовам експлуатації:

– температура повітря: 19-20 град. по Цельсію;

– відносна вологість повітря до 80%;

– атмосферний тиск 107 кПа.

5.7 Вимоги до складу та параметрів технічних засобів

Програмне забезпечення повинно бути реалізоване на ПЕОМ архітектури IBM PC, працювати в ОС Windows XP/Vista/7/8/10/11 і з сумісними з цією платформою пристроями і прикладним програмним забезпеченням.

5.8 Вимоги до інформаційної і програмної сумісності

Переносність програмного забезпечення повинна бути забезпечена за рахунок його реалізації стандартного інтерфейсу взаємодії з ОС, що працюють під управлінням ОС Windows XP/Vista/7/8/10/11.

5.8.1 Обладнання

Комп’ютер Intel® Celeron/8 Mb/1.2 Gb/SVGA 14” 1Mb або сумісні з ним.

5.8.2 Мова програмування

Середовище Java та Scala

Бази даних SQL та NOSQL

5.8.3 Вхідні дані

Опис алгоритму роботи запропонованої системи.

5.8.4 Вихідні дані

Робоча програма.

6 Вимоги до програмної документації

Програмна продукція повинна бути представлена у виді опису структури даних, схем та опису алгоритму, а також текстів вихідних модулів програмного забезпечення згідно ЄСПД .

7 Економічні вимоги

7.1 Маркетингове та економічне обґрунтування ІТ-проєкту

7.2 Виконати розрахунок витрат показників економічного ефекту з урахуванням цін на 3 вересня 2024 року.

8 Вимоги щодо охорони праці

В частині охорони праці та техніки безпеки в магістерській роботі повинні бути розглянуто заходи покращення умов праці та розрахунок захисного заземлення.

9 Перелік документів, що розробляються

– Наукова новизна – 1 аркуш.
– Структурна схема системи – 1 аркуш.
– Функціональна схема системи – 1 аркуш.
– Діаграма процесів – 1 аркуш.
– Блок-схема алгоритму роботи програм – 2 аркуша.
– Маркетингове та економічне обґрунтування ІТ-проєкту – 1 аркуш.
– Пояснювальна записка - 115 аркушів.

10 Етапи розробки

10.1 Збір і обробка інформації по темі випускної роботи. Постановка задачі на виконання кваліфікаційної роботи (складання ТЗ).

10.2 Проведення досліджень або експериментальних робіт для уточнення основних положень випускної магістерської роботи.

10.3 Розробка функціональних схем, блок схем алгоритмів роботи програмного забезпечення.

10.4 Побудова схем взаємодії даних.

10.5 Створення прототипу ПЗ.

10.6 Віднаходження ПЗ, аналіз отриманих результатів.

10.7 Оформлення пояснювальної записки і виконання робіт по графічній частині.

11 Порядок контролю та приймання

11.1 Подання випускної роботи на попередній захист 04.12.2024 р.

11.2 Подання магістерської роботи на захист 10.12.2024 р.

Додаток Б

(обов'язковий)

Міністерство освіти і науки України

Центральноукраїнський національний технічний університет

ЗАТВЕРДЖУЮ
Керівник випускної кваліфікаційної роботи
за другим (магістерським) рівнем вищої освіти
_____ Кислун О.А.

Дослідження та програмна реалізація системи безпеки на основі бездротових технологій

Лістинг програми

Код документу 12

Носій: CD/DVD-диск

Загальна кількість аркушів: 16

Літера: РП

Кропивницький – 2024 року

Файл основного файлу програми

MessageQueueSender.java

```
package com.iot.mq;
import java.util.concurrent.CompletionStage;
public interface MessageQueueSender {
CompletionStage<Void> send(String topic, String key, String data);
}
```

KafkaSender.java

```
package com.iot.mq;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;
import java.util.Properties;
import java.util.concurrent.CompletableFuture;
public class KafkaSender implements MessageQueueSender {
private final Producer<String, String> producer;
public KafkaSender() {
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
properties.put("acks", "1");
properties.put("key.serializer", StringSerializer.class.getName());
properties.put("value.serializer", StringSerializer.class.getName());
producer = new KafkaProducer<>(properties);
}
public CompletableFuture<Void> send(String topic, String key, String data) {
ProducerRecord<String, String> record = new ProducerRecord<>(topic,
key, data);
CompletableFuture<Void> future = new CompletableFuture<>();
producer.send(record, (metadata, exception) -> {
if (exception != null) {
future.completeExceptionally(exception);
} else {
future.complete(null);
}
```

```
});
return future;
}
}
```

AuthenticationService.java

```
package com.iot.auth;
import java.util.concurrent.CompletionStage;
public interface AuthenticationService {
CompletionStage<Boolean> authenticate(String login, String password);
}
```

MongoAuthenticationService.java

```
package com.iot.auth;
import com.mongodb.async.client.MongoClient;
import com.mongodb.async.client.MongoClients;
import com.mongodb.async.client.MongoCollection;
import com.mongodb.async.client.MongoDatabase;
import org.bson.Document;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import static com.mongodb.client.model.Filters.eq;
public class MongoAuthenticationService implements AuthenticationService {
public static final String DB_NAME = "iot";
public static final String COLLECTION_NAME = "device_info";
public static final String DEVICE_ID = "device_id";
public static final String PASSWORD = "password";
private final MongoCollection<Document> deviceInfoCollection;
public MongoAuthenticationService() {
MongoClient mongoClient = MongoClients.create();
MongoDatabase db = mongoClient.getDatabase(DB_NAME);
deviceInfoCollection = db.getCollection(COLLECTION_NAME);
}
@Override
public CompletionStage<Boolean> authenticate(String login, String password) {
CompletableFuture<Boolean> future = new CompletableFuture<>();
deviceInfoCollection.find(eq(DEVICE_ID, login)).first((result,
exception) -> {
```

```
if (exception != null) {
future.completeExceptionally(exception);
return;
}
if (result == null) {
future.complete(false);
return;
}
Object pwd = result.get(PASSWORD);
if (pwd instanceof String && pwd.equals(password))
future.complete(true);
115
else
future.complete(false);
});
return future;
}
}
```

JwtTokenService.java

```
package com.iot.token;
import com.auth0.jwt.JWTSigner;
import com.auth0.jwt.JWTVerifier;
import java.util.Map;
import java.util.regex.Pattern;
public class JwtTokenService {
private static final String SECRET = "feRtfVqoyDJWVVoP4X2m";
private static final Pattern AUTH_HEADER_PATTERN = Pattern.compile("^Bearer$",
Pattern.CASE_INSENSITIVE);
private final JWTVerifier verifier = new JWTVerifier(SECRET);
private final JWTSigner signer = new JWTSigner(SECRET);
public String createToken(Map<String, Object> claims) {
return signer.sign(claims);
}
public Map<String, Object> verifyToken(String authorizationHeader) throws
TokenParseException,
TokenVerificationException {
```

```
String token = extractToken(authorizationHeader);
try {
return verifier.verify(token);
} catch (Exception e) {
throw new TokenVerificationException(e);
}
}
private static String extractToken(String authorizationHeader) throws
TokenParseException {
if (authorizationHeader == null)
throw new TokenParseException("Authorization header value is
null");
String[] parts = authorizationHeader.split(" ");
if (parts.length != 2) {
throw new TokenParseException("Incorrect format: '" +
authorizationHeader +
"'. Format is Authorization: Bearer
[token]");
}
String scheme = parts[0];
String credentials = parts[1];
116
if (AUTH_HEADER_PATTERN.matcher(scheme).matches()) {
return credentials;
} else {
throw new TokenParseException("Incorrect scheme: " + scheme);
}
}
}
HttpUtils.java
package com.iot.http;
import io.undertow.server.HttpServerExchange;
import java.util.Deque;
import java.util.Map;
import static io.undertow.util.StatusCodes.BAD_REQUEST;
import static io.undertow.util.StatusCodes.INTERNAL_SERVER_ERROR;
```

```
public class HttpUtils {
public static Void sendRequestError(HttpServerExchange exchange) {
return sendError(exchange, BAD_REQUEST);
}
public static Void sendServerError(HttpServerExchange exchange) {
return sendError(exchange, INTERNAL_SERVER_ERROR);
}
public static Void sendError(HttpServerExchange exchange, int code) {
return sendError(exchange, "", code);
}
public static Void sendError(HttpServerExchange exchange, String body, int
code) {
exchange.setStatusCode(code);
exchange.getResponseSender().send(body);
return null;
}
public static String extractQueryParameter(String parameter, Map<String,
Deque<String>> queryParameters) {
Deque<String> values = queryParameters.get(parameter);
if (values == null || values.isEmpty())
return null;
return values.getFirst();
}
}
117
AuthenticationHttpHandler.java
package com.iot.http;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.auth.AuthenticationService;
import com.iot.token.JwtTokenService;
import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;
import io.undertow.util.Headers;
import io.undertow.util.Methods;
import io.undertow.util.StatusCodes;
import java.util.Deque;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.CompletionStage;
import static com.iot.http.HttpUtils.extractQueryParameter;
import static com.iot.http.HttpUtils.sendError;
import static com.iot.http.HttpUtils.sendServerError;
import static io.undertow.util.StatusCodes.UNAUTHORIZED;
public class AuthenticationHttpHandler implements HttpHandler {
public static final String DEVICE_ID = "deviceId";
public static final String LOGIN = "login";
public static final String PASSWORD = "password";
private final AuthenticationService authenticationService;
private final JwtTokenService tokenService;
private final ObjectMapper objectMapper;
public AuthenticationHttpHandler(AuthenticationService authenticationService,
JwtTokenService tokenService,
 ObjectMapper objectMapper) {
this.authenticationService = authenticationService;
this.tokenService = tokenService;
this.objectMapper = objectMapper;
}
@Override
public void handleRequest(HttpServerExchange exchange) throws Exception {
Map<String, Deque<String>> queryParameters =
exchange.getQueryParameters();
String login = extractQueryParameter(LOGIN, queryParameters);
String password = extractQueryParameter(PASSWORD, queryParameters);
if (login == null || password == null) {
sendError(exchange, "Login credentials are not present",
UNAUTHORIZED);
return;
}
exchange.dispatch();
CompletionStage<Boolean> authenticationFuture =
authenticationService.authenticate(login, password);
authenticationFuture
```

```
.thenAccept((authenticated) -> {
if (authenticated) {
createAndSendToken(login, exchange);
return;
118
}
sendError(exchange, "Authentication
credentials are invalid", UNAUTHORIZED);
})
.exceptionally(throwable ->
sendServerError(exchange));
}
private void createAndSendToken(String login, HttpServerExchange exchange) {
Map<String, Object> tokenClaims = new HashMap<>();
tokenClaims.put(DEVICE_ID, login);
String token = tokenService.createToken(tokenClaims);
String json = objectMapper.createObjectNode().put("token",
token).toString();
exchange.getResponseSender().send(json);
}
}
```

DataHttpHandler.java

```
package com.iot.http;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.mq.MessageQueueSender;
import com.iot.token.JwtTokenService;
import com.iot.token.TokenParseException;
import com.iot.token.TokenVerificationException;
import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;
import io.undertow.util.HeaderValues;
import io.undertow.util.Headers;
import io.undertow.util.Methods;
import io.undertow.util.StatusCodes;
import java.io.IOException;
```

```
import java.util.Map;
import static com.iot.http.AuthenticationHttpHandler.DEVICE_ID;
import static com.iot.http.HttpUtils.*;
import static com.iot.mq.KafkaTopics.DATA_TOPIC;
import static io.undertow.util.StatusCodes.UNAUTHORIZED;
public class DataHttpHandler implements HttpHandler {
public static final String DATA = "data";
private final MessageQueueSender sender;
private final JwtTokenService tokenService;
private final ObjectMapper objectMapper;
public DataHttpHandler(MessageQueueSender sender, JwtTokenService tokenService,
ObjectMapper objectMapper) {
this.sender = sender;
this.tokenService = tokenService;
this.objectMapper = objectMapper;
}
@Override
public void handleRequest(HttpServerExchange exchange) throws Exception {
exchange.getRequestReceiver().receiveFullBytes(this::processMessage,
(exch, e) -> sendServerError(exchange));
}
private void processMessage(HttpServerExchange exchange, byte[] message) {
String authorizationHeader = getAuthorizationHeader(exchange);
if (authorizationHeader == null) {
sendError(exchange, "No 'Authorization' header",
UNAUTHORIZED);
return;
}
Map<String, Object> tokenPayload;
try {
tokenPayload = tokenService.verifyToken(authorizationHeader);
} catch (TokenParseException e) {
sendError(exchange, "Token could not be parsed",
UNAUTHORIZED);
return;
} catch (TokenVerificationException e) {
```

```
sendError(exchange, "Token could not be verified",
UNAUTHORIZED);
return;
}
JsonNode json;
try {
json = objectMapper.readTree(message);
} catch (IOException e) {
sendRequestError(exchange);
return;
}
JsonNode data = json.get(DATA);
String deviceId = getDeviceId(tokenPayload);
if (deviceId == null || data == null) {
sendRequestError(exchange);
return;
}
exchange.dispatch(); // do not end exchange when this method returns,
because saving to kafka is async
sender.send(DATA_TOPIC, deviceId, data.toString())
.thenRun(() -> exchange.getResponseSender().close())
.exceptionally((throwable ->
sendServerError(exchange)));
}
private static String getAuthorizationHeader(HttpServerExchange exchange) {
HeaderValues authorizationHeaderValues =
exchange.getRequestHeaders().get(Headers.AUTHORIZATION);
return authorizationHeaderValues == null ? null :
authorizationHeaderValues.getFirst();
}
private static String getDeviceId(Map<String, Object> tokenPayload) {
if (tokenPayload == null)
return null;
Object o = tokenPayload.get(DEVICE_ID);
if (o instanceof String) {
String deviceId = (String) o;
```

```
return deviceId.isEmpty() ? null : deviceId;
}
return null;
}
}
```

HttpServer.java

```
package com.iot.http;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.auth.AuthenticationService;
import com.iot.auth.MongoAuthenticationService;
import com.iot.mq.KafkaSender;
import com.iot.token.JwtTokenService;
import com.iot.weather.http.ReportHttpHandler;
import io.undertow.Undertow;
import static io.undertow.Handlers.routing;
public class HttpServer {
public static void main(final String[] args) {
JwtTokenService tokenService = new JwtTokenService();
ObjectMapper objectMapper = new ObjectMapper();
AuthenticationService authenticationService = new
MongoAuthenticationService();
DataHttpHandler dataHttpHandler = new DataHttpHandler(new
KafkaSender(), tokenService, objectMapper);
AuthenticationHttpHandler authenticationHttpHandler =
new AuthenticationHttpHandler(authenticationService,
tokenService, objectMapper);
ReportHttpHandler reportHttpHandler = new ReportHttpHandler();
Undertow server = Undertow.builder()
.addHttpListener(8080, "localhost")
.setHandler(routing()
.get("/api/report/device/{id}",
reportHttpHandler)
.post("/api/data", dataHttpHandler)
.get("/api/auth/login/{login}/password/{password}", authenticationHttpHandler))
.build();
server.start();
```

```
}
}
DoubleStatsCounter.scala
package com.iot.util
import scala.math.sqrt
case class DoubleStats(count: Long, avg: Double, stdDev: Double)
case class DoubleStatsCounter(var n: Long = 0, var sum: Double = 0.0, var
sumOfSquares:
Double = 0.0) {
 def add(number: Double): DoubleStatsCounter = {
 n += 1
 sum += number
 sumOfSquares += number * number
 this
 }
 def merge(anotherCounter: DoubleStatsCounter): DoubleStatsCounter = {
 n += anotherCounter.n
 sum += anotherCounter.sum
 sumOfSquares += anotherCounter.sumOfSquares
 this
 }
 def avg = sum / n
 def stdDev = sqrt(sumOfSquares / n - avg * avg)
 def stats = DoubleStats(n, avg, stdDev)
}
object DoubleStatsCounter {
 def apply(number: Double): DoubleStatsCounter = new
DoubleStatsCounter().add(number)
}
WeatherDomain.scala
package com.iot.weather
import java.util.Date
import com.iot.util.{DoubleStats, DoubleStatsCounter}
object WeatherDomain {
 // db
 val (keyspace, table) = ("iot", "weather")
 // columns
```

```
 val (deviceId, timestamp, temperatureStats, pressureStats) = ("device_id",
"timestamp", "temperature_stats", "pressure_stats")
 val (count, avg, stdDev) = ("count", "avg", "std_dev")
 case class AggregateWeatherDataRecord(temperatureStats: DoubleStatsCounter,
 pressureStats: DoubleStatsCounter) {
 def merge(anotherRecord: AggregateWeatherDataRecord) = {
 temperatureStats.merge(anotherRecord.temperatureStats)
 pressureStats.merge(anotherRecord.pressureStats)
 this
}
 }
 case class WeatherDatabaseRecord(deviceId: String, timestamp: Date,
 temperatureStats: DoubleStats, pressureStats:
DoubleStats)
 case class WeatherDataRecord(temperature: Double, pressure: Double)
}
ReportHttpHandler.scala
package com.iot.weather.http
import com.datastax.spark.connector._
import com.iot.http.HttpUtils.{extractQueryParameter, sendServerError}
import com.iot.weather.WeatherDomain._
import io.undertow.server.{HttpHandler, HttpServerExchange}
import io.undertow.util.HttpString
import org.apache.spark.{SparkConf, SparkContext}
import org.json4s.NoTypeHints
import org.json4s.jackson.Serialization
import org.json4s.jackson.Serialization._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}
class ReportHttpHandler extends HttpHandler {
 val conf = new SparkConf()
 .setMaster("local[*]")
 .setAppName(getClass.getSimpleName)
 .set("spark.cassandra.connection.host", "127.0.0.1")
 val sc = new SparkContext(conf)
 implicit val formats = Serialization.formats(NoTypeHints)
```

```
 val corsHeader = new HttpString("Access-Control-Allow-Origin")
 override def handleRequest(exchange: HttpServerExchange): Unit = {
 var weatherTable = sc.cassandraTable[WeatherDatabaseRecord](keyspace, table)
 Option(extractQueryParameter("id", exchange.getQueryParameters))
 .foreach(id => weatherTable = weatherTable.where(s"$deviceId = ?", id))
 exchange.dispatch
 val rowsFuture = weatherTable.collectAsync()
 exchange.getResponseHeaders.add(corsHeader, "*")
 rowsFuture.onComplete {
 case Success(rows) => exchange.getResponseSender.send(write(rows))
 case Failure(e) => sendServerError(exchange)
 }
 }
}
```

WeatherStreamingProcessor.scala

```
package com.iot.weather.stream
import java.util.Date
import com.datastax.spark.connector.cql.CassandraConnector
import com.datastax.spark.connector.streaming._
import com.iot.mq.KafkaTopics.DATA_TOPIC
import com.iot.util.DoubleStatsCounter
import com.iot.weather.WeatherDomain._
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, SparkContext}
import org.json4s._
import org.json4s.jackson.JsonMethods._
object WeatherStreamingProcessor extends App {
 implicit val formats = DefaultFormats
 val batchDuration = Seconds(1)
 val conf = new SparkConf()
 .setMaster("local[*]")
 .setAppName(getClass.getSimpleName)
 .set("spark.cassandra.connection.host", "127.0.0.1")
 val sc = new SparkContext(conf)
```

```
 val ssc = new StreamingContext(sc, batchDuration)
 type Key = String
 val (zkQuorum, groupId) = ("localhost:2181", "weather-consumer")
 val stats = "stats" // type
 // json fields
 val (temperature, pressure) = ("temperature", "pressure")
 CassandraConnector(conf).withSessionDo { session =>
 session.execute(s"DROP KEYSPACE IF EXISTS $keyspace")
 session.execute(s"CREATE KEYSPACE IF NOT EXISTS $keyspace " +
 s"WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 1 }")
 session.execute(s"CREATE TYPE $keyspace.$stats ($count BIGINT, $avg DOUBLE,
$stdDev
DOUBLE)")
 session.execute(
 s"""CREATE TABLE IF NOT EXISTS $keyspace.$table
 ($deviceId TEXT, $timestamp TIMESTAMP,
 $temperatureStats FROZEN<stats>,
 $pressureStats FROZEN<stats>,
 PRIMARY KEY ($deviceId, $timestamp))""")
 session.execute(s"TRUNCATE $keyspace.$table")
 }
 val stream = KafkaUtils.createStream(ssc, zkQuorum, groupId, Map(DATA_TOPIC ->
1),
StorageLevel.MEMORY_ONLY)
 stream
 .map(parseKafkaRecord)
 .map(toAggregateRecord)
.reduceByKeyAndWindow((r1: AggregateWeatherDataRecord, r2:
AggregateWeatherDataRecord) => r1.merge(r2),
 batchDuration, batchDuration)
 .map(toDatabaseRecord)
 .saveToCassandra(keyspace, table)
 ssc.start()
 def parseKafkaRecord(t: (String, String)): (Key, WeatherDataRecord) = {
 val json = parse(t._2)
 (t._1, WeatherDataRecord(
 (json \ temperature).extractOrElse(0.0),
```

```
 (json \ pressure).extractOrElse(0.0)))
 }
 def toAggregateRecord(t: (Key, WeatherDataRecord)): (Key,
AggregateWeatherDataRecord)
=
 (t._1, AggregateWeatherDataRecord(DoubleStatsCounter(t._2.temperature),
DoubleStatsCounter(t._2.pressure)))
 def toDatabaseRecord(t: (Key, AggregateWeatherDataRecord)) =
 WeatherDatabaseRecord(t._1, new Date(), t._2.temperatureStats.stats,
t._2.pressureStats.stats)
```