Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«17» травня 2024 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Резильєнтна архітектура для кластеру розподілених сервісів із часовою самосинхронізацією»

Виконавець: студент ІІ курсу, групи КБм-21

_____ **Максим КОТОВ**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Сергій ТОЛЮПА	
Нормоконтроль	Яніна ШЕСТАК	

Київ 2024

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«17» листопада 2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача КБм-21 (група) Котова Максима Сергійовича (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи «Резильєнтна архітектура для кластеру розподілених сервісів із часовою самосинхронізацією»

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 5 від 15.11.2023 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Процес побудови розподілених резильєнтних систем, процес моделювання архітектур складних розподілених систем, та імплементація стійких додатків.
Предмет досліджень	Комбінації архітектурних компонентів, протоколів, та технологій для побудови розподілених систем, а також сервіси, що застосовуються в складних системах із високим навантаженням
Мета	Розробка варіацій архітектур побудови розподілених резильєнтних систем із застосуванням механізмів гарантування

	стійкості, включаючи методів на основі часової самосинхронізації.
Вихідні дані для проведення роботи	Архітектури розробки та взаємодії реплікованих систем, стек технологій функціонування вузлів кластеру, механізми синхронізації, алгоритми обміну інформацією у розподіленому середовищі

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна	полягає у вдосконаленні та створенні нових механізмів та методів забезпечення стійкості та масштабованості розподілених систем. Розроблено варіації логічної побудови розподілених систем, та порівняно їх ефективність. Вперше розроблено та описано метод синхронізації сервісів та досягнення узгодженості на основі принципів ідемпотентності та часової самосинхронізації.
Практична цінність	полягає розробці декількох варіантів архітектур побудови розподілених систем, список рекомендацій щодо побудови резильєнтних розподілених систем та програмна імплементації однієї з розроблених архітектур.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Робота виконана у повному обсязі відповідно до теми.

5. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	17.11.2023 – 25.11.2023
Аналіз літературних джерел	26.11.2023 – 14.01.2024
Дослідження структури реплікованого середовища.	15.01.2024 – 22.01.2024
Аналіз технологій та протоколів комунікації розподілених систем.	23.01.2024 – 29.01.2024
Визначення стеку технологій.	30.01.2024 – 06.02.2024
Розробка базової архітектури збору інформації та її надійної обробки.	07.02.2024 – 14.02.2024
Розробка архітектури з реплікацією на основі зовнішнього планувальника та зворотного запису.	15.02.2024 – 23.02.2024

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Розробка архітектури з реплікацією на основі синхронізації за допомогою зовнішніх сервісів.	24.02.2024 – 08.03.2024
Розробка архітектури з реплікацією на основі протоколу досягнення консенсусу.	09.03.2024 – 18.03.2024
Розробка архітектури з реплікацією на основі часової самосинхронізації.	19.03.2024 – 26.03.2024
Імплементація системи представленої при розробці архітектури з реплікацією на основі часової самосинхронізації.	27.03.2024 – 18.04.2024
Формування рекомендацій щодо побудови розподілених систем.	19.04.2024 – 25.04.2024
Оформлення пояснювальної записки згідно методичних рекомендацій	26.04.2024 – 12.05.2024
Подача пакету документів на розгляд ЕК	13.05.2024 – 18.05.2024

6. РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект Результати роботи мають значний вплив на зниження потенційних збитків від ймовірних внутрішніх та зовнішніх чинників, а також ефективне застосування ресурсів системи.

Соціальний ефект Покращення технологій гарантування стійкості та доступності розподілених систем.

7. ДОДАТКОВІ ВИМОГИ

Завдання видав _____ Сергій ТОЛЮПА
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Максим КОТОВ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 17.11.2023 р.
Термін подання кваліфікаційної роботи до ЕК 17.05.2024 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Основний текст займає 110 сторінок, включає в себе зміст, вступ, три розділи дипломної роботи, висновки та список джерел. Крім того, робота містить 4 додатки із загальною кількістю 18 сторінок. У пояснювальній записці міститься 15 рисунків. Використано 76 джерел.

Об’єкт дослідження - процес побудови розподілених резильєнтних систем.

Мета роботи - розробка архітектури побудови розподілених резильєнтних систем на основі часової самосинхронізації.

Предметом дослідження є комбінації архітектурних компонентів, протоколів, та технологій для побудови розподілених систем.

Методи дослідження: спостереження, порівняння, узагальнення, абстрагування, формалізація, аналіз і синтез.

Наукова новизна: вдосконалено та створено нові механізми та методи забезпечення стійкості та масштабованості розподілених систем. Розроблено варіації логічної побудови розподілених систем, та порівняно їх ефективність. Вперше розроблено та описано метод синхронізації сервісів та досягнення узгодженості на основі принципів ідемпотентності та часової самосинхронізації.

Актуальність теми: В епоху постіндустріальної інформатизації, використання цифрових технологій встає все більшою частиною не лише суспільного життя, а й його інфраструктурної основи. В сучасному світі все більше ключових сфер, що забезпечують існування суспільства та людства в цілому, такі як комерційна, правоохоронна, військова, медична, політична, та інші – стають залежними від цифрової інфраструктури. В зазначених сферах, кожна помилка може коштувати як значних ресурсних витрат, так і втрати людських життів. Саме тому надійність систем, що є основою таких критичних сфер – є надзвичайно важливою.

Ключові слова: резильєнтність, розподілені системи, реплікація сервісів, стійкість через надмірність, AMQP, RabbitMQ, часова самосинхронізація.

В ході виконання роботи виконано наступні завдання:

– проведено аналіз технологій, протоколів та методів комунікації у розподілених системах;

– побудовано вісім варіацій архітектури резильєнтної розподіленої системи збору та обробки інформації, порівняно їх ефективність та способи застосування;

– виконана програмна реалізація однієї із зазначених архітектур на платформі Node.js з використанням фреймворку Express.js, клієнтом AMQP протоколу, на базі брокера RabbitMQ;

– запропоновано список рекомендацій щодо побудови резильєнтних розподілених систем, враховуючи результати оцінки ефективності розроблених механізмів.

Практична цінність отриманих результатів: розробка декількох варіантів архітектур побудови розподілених систем, програмна імплементації однієї з розроблених архітектур та список рекомендацій щодо побудови резильєнтних розподілених систем.

Напрямки подальших досліджень: криптографія, блокчейн, розподілені системи, системи з високим навантаженням, відмовостійкі системи, розробка безпечних серверних частин з використанням Golang, вдосконалення методів захисту вебдодатків Node.js.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

AMQP	–	Advanced Message Queuing Protocol
MQTT	–	Message Queuing Telemetry Transport
TCP	–	Transmission Control Protocol
IP	–	Internet Protocol
UDP	–	User Datagram Protocol
HTTP	–	Hypertext Transfer Protocol
REST	–	Representational State Transfer
BGP	–	Border Gateway Protocol
OSPF	–	Open Shortest Path First
API	–	Application Programming Interface
RPC	–	Remote Procedure Call
gRPC	–	gRPC Remote Procedure Calls
ORM	–	Object-Relational Mapper
WS	–	WebSocket
SSL	–	Secure Sockets Layer
TLS	–	Transport Layer Security
NTP	–	Network Time Protocol
RTP	–	Real-time Transport Protocol
SQL	–	Structured Query Language
QoS	–	Quality of Service
IoT	–	Internet of Things
NPM	–	Node Package Manager

ЗМІСТ

ВСТУП

В епоху постіндустріальної інформатизації, використання цифрових технологій встає все більшою частиною не лише суспільного життя, а й його інфраструктурної основи. В сучасному світі все більше ключових сфер, що забезпечують існування суспільства та людства в цілому, такі як комерційна, правоохоронна, військова, медична, політична, та інші – стають залежними від цифрової інфраструктури. В зазначених сферах, кожна помилка може коштувати як значних ресурсних витрат, так і втрати людських життів. Саме тому надійність систем, що є основою таких критичних сфер – є надзвичайно важливою.

Зі збільшенням залежності від цифрової інфраструктури та зростанням її інтеграції, надійність, та можливість функціонувати при можливих непередбачуваних обставинах, таких як: природні катаклізми, війни, помилки у експлуатації або розробці – стає основною проблемою, яку повинні вирішувати інженери, архітектори та дослідники. У даній дипломній роботі надається глибокий аналіз побудови розподілених систем, їх основи, існуючі технології, методи, та практики.

Сучасні розподілені системи потребують надійних та ефективних механізмів зв’язку для синхронізації, обробки та координації обробки даних між різними вузлами. У цій дипломній роботі розглядаються приклади найбільш поширених комунікаційних протоколів такі як AMQP (Advanced Message Queuing Protocol) і MQTT (Message Queuing Telemetry Transport), що застосовуються для обміну інформацією в асинхронному режимі і порівнюється їх застосування та ефективність у різних сценаріях та вимогах щодо складності логіки маршрутизації, безпеки, процесорного навантаження та використання пам’яті. Дана робота містить детальний аналіз брокерів повідомлень RabbitMQ і Kafka, що реалізують зазначені протоколи та відіграють ключову роль в управлінні зв’язком між різними компонентами розподіленої системи.

Перший розділ роботи містить детальний огляд комунікаційних протоколів у розподілених системах. В першу чергу проаналізовано та розглянуто історичну

еволюції протоколів комунікації мережевої та міжмережевої взаємодії. Надано порівняльний аналіз існуючих асинхронних протоколів обміну інформацією на основі черг, порівняно їх ефективність та специфіку використання. Розглянуто комерційні та публічні реалізації таких протоколів у системах обміну повідомлень, або так званих «брокерах» повідомлень. Важливо частиною даного розділу є також огляд інфраструктурних рішень, що забезпечують ізоляцію розподіленого додатку від деталей реалізації та залежностей системи, на якій цей додаток виконує свої функції, а також їх можливостей щодо координації мережевої взаємодії та абстрагування складнощів мережевого налаштування та керування.

У наступному розділі розглядається архітектура стійких розподілених систем, розроблені та представлені різні моделі та підходи до побудови систем, здатних протистояти збоям і ефективно відновлюватися після них. Розглянуті та порівняні потенційні методи, їх недоліки та переваги, що надають можливість гарантувати послідовну обробку інформації у системі. Надано аналіз механізмів зворотного циклічного запису інформації, стійкості через надмірність, реплікації. Розглянуто існуючі алгоритми та протоколи обрання лідера у реплікованому середовищі та описано модифікацію одного з таких алгоритмів, застосованого у одній із запропонованих архітектур розподіленої системи. Особливе значення надано дослідженню методів стійкості через надмірність з використанням механізмів ідемпотентності на основі систем токенізації.

Практична реалізація описана у третій частині даної роботи та становить значну частину цього дослідження, зосереджена на методах самосинхронізації на основі часу як новому підході до підвищення стійкості системи. У процесі побудови розподіленої системи, значну увагу приділено вибору технологічних стеків, як інфраструктурних, так і програмних, а також обговорюється їхня роль у впровадженні стійкої розподіленої системи.

Мета роботи полягає в тому, щоб отримати глибоке розуміння труднощів пов’язаних із проектуванням, розробкою, впровадженням та підтримкою стійких розподілених систем, а також, знайти оптимальні рішення, що будуть задовольняти потреби системи та користувачів, за допомогою проведення досліджень, аналізу

існуючих технологічних рішень, розробки стійких системних архітектур розподілених додатків і практичної реалізації. Надання інформації та рекомендацій щодо створення систем, які є ефективними, масштабованими, безпечними та надійними, коли виникають різноманітні операційні проблеми, є однією з цілей цієї дипломної роботи.

Дане дослідження має на меті вплинути на те, як будуть проектуватися майбутні безпечні та ефективні розподілені системи, що мають впоратися зі зростаючою складністю сучасних обчислювальних потреб. Таким чином, ця робота є важливим ресурсом для тих, хто бере участь у розробці та впровадженні розподілених систем, надаючи покращене розуміння створення рішень, які є стійкими, безпечними та технологічно просунутими.

Практична цінність цього дослідження полягає у розробці декількох варіантів архітектур побудови розподілених систем, список рекомендацій щодо побудови резильєнтних розподілених систем та програмна імплементації однієї з розроблених архітектур. Практична цінність виходить за межі академічного середовища і є значимою у сферах зі зростаючою потребою в надійних розподілених системах, таких як торгівля цінними паперами та криптовалют, що характеризуються значними об’ємами переданої та оброблюваної інформації, а також інших зазначених раніше галузях, що вимагають високого рівня стійкості та надійності.

Об’єктом дослідження у даній роботі є процес побудови розподілених резильєнтних систем, процес моделювання архітектур складних розподілених систем, та імплементація стійких додатків.

Предметом дослідження є комбінації архітектурних компонентів, протоколів, та технологій для побудови розподілених систем, а також сервіси, що застосовуються в складних системах із високим навантаженням.

Методами дослідження, використаних під час виконання даної кваліфікаційної роботи є: спостереження, порівняння, узагальнення, абстрагування, формалізація, аналіз і синтез.

Науковою новизною є вдосконалення та створення нових механізмів та методів забезпечення стійкості та масштабованості розподілених систем. Розроблено варіації

логічної побудови розподілених систем, та порівняно їх ефективність. Вперше розроблено та описано метод синхронізації сервісів та досягнення узгодженості на основі принципів ідемпотентності та часової самосинхронізації.

Для досягнення мети роботи , було поставлено наступні завдання:

– провести аналіз технологій, протоколів та методів комунікації у розподілених системах;

– побудувати варіації архітектури резильєнтної системи, порівняти їх ефективність та способи застосування;

– виконати програмну реалізацію однієї із розроблених архітектур на платформі Node.js з використанням фреймворку Express.js, клієнтом AMQP протоколу, на базі брокера RabbitMQ;

– запропонувати список рекомендацій щодо побудови резильєнтних розподілених систем, враховуючи результати оцінки ефективності розроблених механізмів та архітектурних рішень.

Апробація результатів та публікації за темою кваліфікаційної роботи виконана в ході подання 11 тез доповідей та участі у міжнародних наукових конференціях «Проблеми кібербезпеки інформаційно-телекомунікаційних систем», «IT&I Information Technology and Interactions», «Захист інформації і безпека інформаційних систем», та «Прикладні системи та технології в інформаційному суспільстві». За темою кваліфікаційної роботи опубліковані 2 статті у фахових наукових міжнародних виданнях. Посилання на відповідні наукові праці та участь у конференціях описана у додатку «Г» цієї кваліфікаційної роботи.

РОЗДІЛ 1
ПРОТОКОЛИ, ТЕХНОЛОГІЇ ТА СПОСОБИ КОМУНІКАЦІЇ У РОЗПОДІЛЕНИХ СИСТЕМАХ

1.1 Огляд комунікаційних протоколів у розподілених системах

При побудові складних розподілених систем, початковими основними проблемами є встановлення єдиного, стандартизованого інтерфейсу комунікації між вузлами, що можуть бути під’єднаними до мережі із різних міст, країн, та навіть частин світу.

Для вирішення цієї проблеми були створені протоколи мережевої комунікації, котрі описують послідовність інструкцій та формат обміну інформації, створюючи стандартизований інтерфейс комунікації між віддаленими вузлами.

Для ефективної розробки дедалі складніших систем комунікації, при побудови протоколів мережевої взаємодії застосовано фундаментальний принцип комп’ютерних наук, а саме – абстрагування, котре дозволяє зосереджуватись на розробці складніших систем, на основі простіших, за допомогою наданого інтерфейсу, не звертаючи уваги на деталі реалізації нижчих рівнів взаємодії.

У контексті мережевої взаємодії, визначено 7 рівнів абстракції протоколів, відповідно до моделі OSI (Open Systems Interconnection), що включають [1]:

– рівень прикладних додатків, відповідальний за високорівневу логіку взаємодії, що може бути представленою механізмами обміну файлами, або віддаленого керування;

– представницький рівень, відповідальний за встановлення єдиного формату обміну інформацію та трансляцію між форматами, що можуть бути використані у різних системах у глобальній мережі;

– рівень сесії, відповідальний за відслідковування сеансу мережевої взаємодії, де сенсом називають послідовність обміну інформацією між двома учасниками мережі для виконання поставленого завдання;

– транспортний рівень, що відповідальний за доставку інформації від процесу, що створений та виконується на одному вузлі в мережі, до процесу, що знаходиться на іншому вузлі, а також, за реалізацію механізму гарантованої доставки інформації між ними;

– мережевий рівень відповідальний за адресацію та маршрутизацію пакетів у мережі та його основним завданням є доставка пакетів від одного вузла до іншого, проте він не несе відповідальність за встановлення з'єднання та обміну інформації між процесами;

– канальний рівень відповідальний за логіку обміну інформації безпосередньо у конкретній реалізації каналу обміну інформацією у локальній мережі. Канальний рівень не несе відповідальність за глобальну адресацію, проте відповідальний за адресацію та логічну частину комунікації у локальній мережі, що об'єднана єдиним каналом та технологією комунікації;

– завданням фізичного рівня взаємодії є встановлення стандарту функціонування та взаємодії з фізичними каналами комунікації та інтерфейсами пристроїв.

1.1.1 Історичний контекст та еволюція протоколів комунікації

Протоколи зв'язку були створені ще до початку епохи глобальних комп'ютерних мереж. Ранні етапи розробки протоколів були в основному результатом необхідності стандартизації передачі даних у середовищі, де домінували різноманітні та несумісні системи. У цей період виникли численні проблеми, зокрема відсутність універсальних правил передачі та прийому даних. У міру розширення мереж нездатність різних систем ефективно обмінюватись інформацією стала серйозною перешкодою, яка потребувала структурованого підходу до обміну даними. Перші протоколи були орієнтовані виключно на транспорт і їхня основна мета полягала в тому, щоб розробити базові методи підключення та передачі даних [2].

Прогрес у мережевих технологіях демонструє еволюцію комунікаційних протоколів від простої транспортної системи до складних послідовностей обробки

інформації та заздалегідь закладених архітектурних рішень, що вирішують проблеми надійності, безпеки та масштабованості. Ці зміни відбулись зокрема завдяки появі Інтернету, оскільки все більше протоколів були розроблені для швидкого розширення цифрових мереж, котрі повинні передавати сотні петабайт інформації по всьому світу. Перетворюючи протоколи з простих механізмів передачі даних на складні системи, здатні обробляти складний, великий обсяг зв'язку, кожна стадія призводила до покращення швидкості, надійності та безпеки.

Розробка та впровадження стандартизованих протоколів стеку TCP (Transmission Control Protocol)/IP (Internet Protocol), а пізніше і протоколу обміну гіпертекстом HTTP (Hypertext Transfer Protocol), стали важливим етапом розвитку цифрового зв'язку та поклали основу веб-інфраструктури сьогодення. Стандартизація зробила правила комунікації одноманітними та дозволила об'єднувати мережі зі всього світу в єдину глобальну всесвітню павутину. Стек протоколів TCP/IP був та залишається основою Інтернет-зв'язку, який дозволяє взаємодіяти між численними системами та мережами. HTTP створив основу Всесвітньої павутини, суттєво змінивши спосіб обміну та доступу до інформації та поклав початок неспинному розвитку все більш нових протоколів, що вирішують завдання передачі відформатованих даних, в тому числі веб-додатків, у режимі «на вимогу» а також реального часу [2-5].

З розширенням використання цифрової інфраструктури, все частіше інженерам доводилось вирішувати завдання внутрішнього масштабування системи та гарантувати не тільки доставку повідомлень, а й те, що великі системи зможуть справлятись зі скачками навантаження як в мережі так і на кінцевих станціях. Протоколи на основі стійких буферизованих черг AMQP та MQTT, були створені на вимогу вирішення проблем розширення систем, що працювали у розподіленому режимі, обмінювались даними і мали нерівномірний розподіл трафіку та навантаження. Протоколи на основі черг повідомлень надають ряд високорівневих функцій, таких як логічна маршрутизація, чергування повідомлень і надійна доставка, котрі необхідні для сучасних розподілених додатків. Такі протоколи надають можливість подолати обмеження попередніх систем, які вимагали негайної обробки

переданих даних, або виконання команд. Вони задовольнили зростаючий попит на рішення для комунікації, які є високопродуктивними, масштабованими та гнучкими в масштабних середовищах хмарних обчислень та Інтернету речей [4-7].

Вимоги до допустимого часу відгуку системи, або отримання інформації чи результатів обчислень швидко змінюються, що вимагає обміну інформацією в розподілених системах у режимі реального часу. У відповідь на цю потребу були розроблені та інтегровані протоколи зв’язку в реальному часі, такі як WebSocket. Такі протоколи дозволяють клієнтам і серверам мати безперервний двонаправлений зв’язок та стають архітектурною основою у випадках, коли швидка передача даних є критично важливою, таких як онлайн-ігри, фінансова торгівля та послуги потокового передавання. Упровадження протоколів передачі даних у режимі реального часу відкрило нову еру в розвитку протоколів зв’язку, оскільки вони значно підвищили швидкість реагування та інтерактивність сучасних програм.

1.1.2 Основні характеристики та цілі використання протоколів

Протоколи мережевої взаємодії визначаються набором основних характеристик, які задають спосіб обміну та обробки даних. Синтаксис протоколу визначає формат і структуру пакетів даних, що передаються між системами і є найважливішим із них оскільки дозволяє стандартизувати очікування формату переданої інформації в різних частинах системи. Не менш важливою є семантика, яка визначає значення та інтерпретацію кожної частини пакета, що дозволяє системі автоматизовано реагувати на передану інформацію. Крім того, надійні механізми обробки помилок є важливою частиною протоколів зв’язку, оскільки вони дозволяють ідентифікувати та виправляти втрати, які можуть виникнути під час передачі [5-7].

На основі фундаментальних принципів проектування та того, як вони використовуються, комунікаційні протоколи можна групувати в різні категорії. Надійність та швидкість передачі даних у логічні площині, без урахування спроможностей фізичного каналу, через мережу, залежить від транспортних

протоколів, таких як TCP і UDP (User Datagram Protocol). Для оптимізації шляхів передачі даних через складні мережі - існують протоколи маршрутизації, такі як BGP (Border Gateway Protocol) і OSPF (Open Shortest Path First), що працюють на рівні інтра та інтермережевої маршрутизації відповідно. Для сучасних розподілених додатків, протоколи обміну повідомленнями через черги AMQP і MQTT - є основою управління масштабованістю та високим навантаженням. Кожна з цих груп виконує цілі відповідного рівня міжмережевої взаємодії де вибір протоколу має значний вплив на загальну продуктивність, надійність і функціонування системи [7-10].

Придатність протоколів комунікації до застосування у цільовій системі залежить від конкретних випадків використання та рівня продуктивності. Протоколи націлені на мінімізацію використання фізичних ресурсів, такі як MQTT, є більш ефективними в додатках Internet of Things (IoT), де підключення пристроїв і низьке енергоспоживання є пріоритетними. Такі протоколи, як WebSocket або RTP (Real-time Transport Protocol), забезпечують швидкість і миттєву передачу даних для потреб у зв’язку в режимі реального часу, як-от у онлайн-іграх або фінансових торгових платформах. Показники продуктивності, такі як швидкість, надійність, масштабованість і безпека, є ключовими при порівнянні цих протоколів. Зрештою, вибір правильного протоколу залежить від ретельного поєднання цих елементів, адаптованого до конкретних потреб і обмежень системи [10-11].

AMQP високо цінується за свою універсальність і надійність у сферах логічної маршрутизації, надійності, безпеки, та черги повідомлень. AMQP сприяє ефективній відправці та прийому повідомлень у мікросервісах, де різні служби повинні взаємодіяти надійним і масштабованим способом. Він підтримує складну маршрутизацію на основі обмінників, черг та зв’язків між ними, а також гнучку обробку повідомлень, які необхідні в розподіленому середовищі, де додаткові сервіси розробляються, розгортаються та масштабуються незалежно [12].

MQTT, відомий своєю простотою та ефективністю, особливо підходить для сценаріїв з обмеженими ресурсами та пропускною здатністю. MQTT пропонує ефективний спосіб обробки переривчастих з’єднань або мереж із низькою пропускною здатністю в архітектурах мікросервісів, особливо тих, що включають

пристрої IoT або потребують оновлення в реальному часі. Його модель публікації-підписки ідеально підходить для мікросервісів, яким потрібно швидко поширювати інформацію, наприклад, у сенсорних мережах [13].

1.2 Порівняльний аналіз протоколів AMQP та MQTT

1.2.1 Основні особливості AMQP

AMQP — це протокол обміну повідомленнями прикладного рівня з відкритим стандартом, що розроблений для управління логікою обробки та маршрутизації трафіку до черги повідомлень при високих навантаженнях та складних вимогах до послідовності обробки інформації. Здатність AMQP надавати складні рішення маршрутизації та гарантувати доставку повідомлень навіть у разі перебоїв у мережі та збоїв сервера є його основною перевагою [14-18].

AMQP підтримує схему комунікації запит-відповідь, обробку подій і публікацію-підписку, а також багато інших шаблонів, що, з огляду на його гнучкість, дозволяє його використання у багатьох контекстах, включаючи фінансові послуги та хмарну інфраструктуру [14-18].

AMQP дозволяє обмінюватись між різними платформами та має імплементацію клієнтів на різних мовах, що є важливою умовою в різноманітних корпоративних середовищах, які в більшості своїй працюють на основі архітектури мікросервісів. Крім того, гнучкість, необхідна для задоволення різноманітних вимог додатків, від простих передач повідомлень до складних транзакційних систем, забезпечується підтримкою AMQP кількох рівнів якості обслуговування та його здатністю обробляти широкий спектр розмірів і типів повідомлень.

Даний протокол декомпозує логіку маршрутизації у декілька ключових компонентів з різними механізмами управління станів: брокер, черги, та зв’язки між ними. У такій системі черги масштабуються виключно вертикально, а зв’язки та обмінники – горизонтально, оскільки не мають стану, що вимагає синхронізації між різними вузлами системи.

1.2.2 Основні особливості MQTT

Розроблений для підключення віддалених пристроїв з мінімальною пропускною здатністю мережі, протокол публікації та підписки телеметричних повідомлень, відомий як MQTT зараз широко використовується в програмах Інтернету речей. Його концепція дизайну зосереджена на простоті впровадження, які є критично важливими для пристроїв з обмеженими можливостями обробки, а також для середовищ з низькою пропускною здатністю або ненадійними мережевими з'єднаннями [19-22].

Масштабованість цього протоколу є однією з його основних переваг у порівнянні з аналогічними протоколами на основі черг та буферів повідомлень, оскільки він може керувати тисячами одночасних з'єднань, не використовуючи надмірну кількість мережевих ресурсів. Такий ефект досягається завдяки бінарному формату обміну повідомленнями, що значно менш гнучким у порівнянні з AMQP, проте, також є значно більше компактним та не вимагає використання додаткових ресурсів пропускної здатності, обробки та перетворення форматів.

Архітектура MQTT спрощує розповсюдження повідомлень у схемах комунікації типу «один до багатьох» і відокремити виробників і споживачів даних за допомогою моделі публікації-підписки, що підвищує гнучкість і масштабованість у розподілених системах.

Підтримка кількох рівнів якості обслуговування QoS (Quality of Service), що забезпечує широкий спектр гарантій доставки повідомлень, є ключовою особливістю MQTT та включає доставку «щонайбільше один раз» для ситуацій, коли втрата повідомлення дозволена, доставку «принаймні один раз» для того, щоб гарантувати доставку повідомлень, але дозволити їх дуплікацію і доставку та «рівно один раз» для випадків використання, коли важливо, щоб кожне повідомлення отримано лише один раз і не було б жодної втрати.

Крім того, функції MQTT, такі як збережені повідомлення та «Last Will and Testament», підвищують його надійність та є ключовими для програм віддаленого моніторингу та контролю в екосистемах Інтернету речей [19-22].

1.2.3 Сценарії використання та відмінності у практичному застосуванні

Хоча AMQP і MQTT обидва є протоколами обміну повідомленнями на основі черг, вони мають чіткі практичні відмінності та розроблені для різних сценаріїв використання та як функціональних так і ресурсних вимог розроблюваних систем.

У програмах корпоративного рівня, де складність системи вимагає протоколу, який може обробляти різноманітні шаблони обміну повідомленнями та гарантувати надійну та безпечну доставку повідомлень, найчастіше використовується AMQP, через його широкий спектр функцій, додаткові складні способи маршрутизації та схеми обробки повідомлень. AMQP є найбільш популярним у сферах великих розподілених фінансових інфраструктур та великомасштабних, високонавантажених хмарних системах [14-18].

На противагу, простота та ефективність використання смуги пропускання та мінімальна надмірність використаних ресурсів є пріоритетними для MQTT. Він краще підходить для середовищ Інтернету речей, де пристрої з обмеженою обчислювальною потужністю потребують надійного обміну даними через потенційно нестабільні мережі. MQTT зазвичай використовується в сенсорних мережах, системах домашньої автоматизації та програмах, які потребують оновлення в режимі реального часу, таких як моніторинг навколишнього середовища та відстеження транспортних засобів [19-22].

Як було вказано у попередньому підрозділі, MQTT надає можливість рівнів QoS, кожний з яких є відповідним компромісом між ефективністю застосування ресурсів та надійністю обміну повідомленнями. AMQP, в свою чергу, надає механізм підтвердження отримання повідомлень, що є значно більш гнучким, проте потребує більшого використання мережевих ресурсів для функціонування системи.

Принципова відмінність у призначенні, використанні та способі побудови вищезазначених протоколів є різниця у гнучкості та ефективності застосування ресурсів системи. AMQP як протокол для побудови складних корпоративних каналів обміну інформацією із застосуванням брокерів та черг, в той час як основний фокус MQTT полягає у мінімізації використання ресурсів фізичної системи.

1.3 Брокери повідомлень в розподілених системах

Брокери повідомлень – це сервіси, що працюють як посередники при встановленні комунікаційних каналів між мікросервісами у розподіленій системі та допомагають різним частинам такої системи взаємодіяти та обмінюватися даними.

Брокери гарантують, що повідомлення направляються, виконуючи складні завдання логічної маршрутизації, ставляться в чергу та, врешті, отримуються сервісом, що підписаний на оновлення заданої черги повідомлень. Послуги брокерів відіграють ключову роль у побудові систем, де всі компоненти повинні бути слабко зв’язані на рівні архітектури, щоб гарантувати масштабованість та стійкість, а також працювати у асинхронному режимі.

Брокери повідомлень абстрагують складнощі управління деталями обміну повідомленнями та взаємодії з конкретними протоколами комунікації, таких як встановлення з’єднань та підтвердження доставки, що дозволяє різним частинам розподіленої системи, які можуть бути побудовані на різних технологіях, безперебійно спілкуватися одна з одною. Більшість комерційних та некомерційних імплементацій брокерів мають заздалегідь розроблені клієнтські частини на різних мовах програмуваннях, що дозволяє значно спростити інтеграцію різних середовищ та обмін даними між ними.

Черга повідомлень, підписка на теми, маршрутизація та гарантії доставки є основною зоною відповідальності брокерів повідомлень. Розсилка на «тему», адресу, або мережу - надає комунікаційну модель публікації та підписки, що створює умови для асинхронної обробки повідомлень та запитів.

Постановка в чергу повідомлень гарантує, що повідомлення зберігаються, доки вони не будуть оброблені. Імплементація черги в більшості брокерів мають додаткові можливості для гарантування стійкості, такі як збереження непідтверджених повідомлень на фізичному диску або журналювання подій для відтворення та реплікації історії повідомлень.

Комплексні можливості маршрутизації дозволяють брокерам спрямовувати повідомлення відповідно до певних шаблонів або правил, що дозволяє будувати як

класичні широкополосні розсилки так і обмежені, цілеспрямовані передачі інформації. Такі можливості корисні у випадку, коли система має слабку залежність між компонентами. Брокер може не лише надавати можливості публікувати повідомлення або події але й виконувати маршрутизацію на основі заданих заголовків, або «тем», що дозволяє абстрагувати конкретні реалізації сервісів та все ще гарантувати доставку і можливість вибору підписки на певні типи повідомлень.

Брокери також відповідальні за надійність та стійкість каналів обміну повідомленнями, та імплементують механізми «щонайбільше один раз» або «точно один раз». Механізм «щонайбільше один раз» повинен гарантувати доставку повідомлень не більше ніж один раз. Така модель є важливою у випадку, коли обробка повідомлення не є ідемпотентною і може призвести до неконсистентного стану системи. Механізм «точно один раз», в свою чергу, є більш жорстким та вимагає точну доставку повідомлення один раз. Такий механізм найчастіше використовується при побудові транзакційних оновлень станів системи.

1.3.1 Огляд ключових брокерів повідомлень: RabbitMQ, Kafka, та інші

RabbitMQ і Kafka є двома найвідомішими брокерами повідомлень, кожен з яких має свої особливі переваги, та, цілком ймовірно, можуть бути одночасно використані в різних частинах розподіленої системи.

RabbitMQ відомий своєю надійною чергою повідомлень, прив’язками до черги, складною маршрутизацією з різними типами обміну та найбільше підходить для сценаріїв, що вимагають складної логіки маршрутизації та гарантованої доставки повідомлень. Kafka, в свою чергу, вирішує проблеми великомасштабної обробки даних і потокових програм завдяки своїй високій пропускній здатності та масштабованості. Розподілена архітектура Kafka забезпечує надійну платформу для агрегації журналів і каналів даних у реальному часі, що дозволяє обробляти величезні обсяги даних із мінімальною затримкою [23-25].

ActiveMQ і Mosquitto є іншими відомими брокерами повідомлень, окрім RabbitMQ і Kafka. ActiveMQ відзначається підтримкою множини протоколів,

включаючи JMS, AMQP, MQTT, та STOMP, що значно спрощує його використання у випадку комплексної корпоративної інтеграції. Mosquitto є чудовим вибором для сценаріїв Інтернету речей, особливо там, де протокол MQTT необхідний для ефективного зв'язку між великою кількістю малопотужних пристроїв. Служба обміну повідомленнями на основі хмари Azure Service Bus добре інтегрується з іншими службами Azure і пропонує функції, такі як дедуплікація повідомлень і «мертві» повідомлення, що підходить для складних хмарних архітектур [26-28].

Вибір брокера повідомлень для розподіленої системи залежить від масштабованості, продуктивності, надійності та сумісності з екосистемою. Важливо щоб функції брокера відповідали потребам системи, якими можуть бути обробка великих потоків даних, надійна доставка повідомлень для корпоративних програм або полегшення процесорного навантаження та використання пам'яті для комунікації в реальному часі в мережах Інтернету речей. Легкість інтеграції з існуючою інфраструктурою, досвід команди з використанням тих чи інших протоколів або брокерів та підтримка бажаних шаблонів обміну повідомленнями також мають вирішальне значення, оскільки напряму впливають на вартість інтеграції брокерів та підтримку подальшої розробки та розширення системи.

1.3.2 Порівняльний аналіз RabbitMQ та Kafka

Незважаючи на те, що обидва працюють як брокери повідомлень, RabbitMQ і Kafka дуже відрізняються з точки зору дизайну та архітектури. RabbitMQ, заснований на AMQP, орієнтований на повідомлення з надійними функціями доставки та керуванням визначення кінцевих черг даних, що робить його популярним рішенням для складних сценаріїв маршрутизації та обробки повідомлень на рівні підприємства. RabbitMQ підтримує такі функції за допомогою комплексної архітектури обміну повідомленнями, що включають такі компоненти як: джерело повідомлень, обмінник, зв'язок, та черга. Обмінники бувають декількох типів та в залежності від нього, надають різні інтерфейси маршрутизації. «Прямий» обмінник дозволяє напряму надсилати повідомлення до вказаної серги. Обмінник на основі «тем» або «топіків» -

дозволяє направляти одне і те ж повідомлення в декілька черг одночасно, що були прив’язані до такого обмінники із зазначеним ключем маршрутизації. Обмінники типу «fanout» дозволяють надсилати повідомлення до абсолютно всіх підключених черг. Обмінник на основі заголовків надає найбільш можливості будувати найбільш складніші схеми маршрутизації з урахуванням значень самих заголовків. У такому обміннику кількість заголовків не є обмеженою і кожний зв’язок з чергою може надавати правила, що будуть визначати чи потрібно надсилати повідомлення із заданими заголовками до прив’язаної черги [29-32].

Kafka, з іншого боку, є розподіленою потоковою платформою, розробленою для високої пропускної здатності, тривалого зберігання повідомлень і обробки в реальному часі. Kafka не надає таких просунутих методів маршрутизації, які надає RabbitMQ, однак Kafka побудований на простих ідеї послідовного запису у файли, що робить обробку повідомлень неймовірно швидкою і дозволяє масштабувати сервіси, котрі потребують великої пропускної здатності на багато краще, ніж це може RabbitMQ. Додаткова складність у використанні Kafka у порівнянні з RabbitMQ полягає у тому, що кожний клієнт, котрий під’єднується до Kafka – відповідальний самостійно за відстеження послідовності повідомлень та ведення обліку вже оброблених повідомлень. Kafka більш використовуваний для агрегації журналів, обробки потоків і пошуку подій, оскільки його архітектура оптимізована для запису та читання великих потоків даних із мінімальною затримкою [33-36].

Хоча масштабування для більших робочих навантажень може вимагати більше низькорівневого керування та налаштування у зв’язку з відсутність балансування навантажень доступу до однієї сутності черги, перевага RabbitMQ полягає в його здатності обробляти велику кількість повідомлень зі складною логікою обробки. Його ефективність значною мірою залежить від конфігурації та складності шаблонів обміну повідомленнями, які використовуються [29-32].

Kafka, розроблений для масштабованості, краще обробляє великі кількості даних завдяки своїй розподіленій природі та механізму розподілу. Kafka краще підходить для додатків із великими даними, вимогами до пропускної здатності та сценаріїв, де дані постійно генеруються та споживаються у великих об’ємах [33-36].

Вибір між RabbitMQ і Kafka часто залежить від конкретних ситуацій і вимог системи як логічних так і технічних. RabbitMQ підходить для традиційних корпоративних додатків, які потребують складної маршрутизації, надійної доставки та різноманітних типів повідомлень. Фінансові операції, зв'язок між службами в архітектурах мікросервісів і керування чергою завдань є прикладами ситуацій, у яких він використовується найчастіше. Kafka розроблений для сценаріїв, які вимагають обробки та аналізу великих потоків даних у реальному часі, наприклад, спостереження за потоками пристроїв Інтернету речей, аналітика в реальному часі або відстеження журналів активності користувачів.

1.3.3 Переваги та недоліки різних брокерів повідомлень

У відповідь на різноманітні вимоги розподілених систем, різні брокери повідомлень пропонують різні переваги як було показано у оглядах систем з минулих розділів. Не існує єдиного брокера, який би вирішив всі проблеми сучасних розподілених систем, та був би одночасно просунутим у конфігурації і логіці обробки повідомлень та здатним обробляти гігантські об'єми інформації і працювати у середовищах з вимогами до надвеликих пропускних здатностей.

RabbitMQ високо оцінюється за свою надійність, складну та висококонфігуровану маршрутизацію повідомлень і широку підтримку протоколів, що робить його одним з найбільш популярних рішень для широкого спектру потреб у обміні повідомленнями.

Завдяки своїм ефективним можливостям обробки великих об'ємів інформації та високій пропускній здатності, що компенсують складність у використанні з точки зору клієнта, Kafka краще підходить для архітектур, керованих подіями та аналітики в реальному часі.

ActiveMQ є вдалим вибором для стандартних моделей корпоративної інтеграції, забезпечуючи гармонійне поєднання різноманітних функцій та зручності у використанні.

Mosquitto та інші брокери на основі MQTT адаптовані до сценаріїв Інтернету речей, оскільки здатні забезпечувати простий і ефективний зв’язок для великої кількості пристроїв, що не вимагають значних процесорних ресурсів або оперативної чи постійної пам’яті.

Зазначені брокери повідомлень мають деякі переваги, але, як було показано, вони також мають обмеження.

У випадку високих вимог до пропускної здатності, RabbitMQ може зіткнутися з проблемами горизонтального масштабування, що призведе до потреби побудови самостійного рішення для балансування навантаження у режимі кластеризації. RabbitMQ підтримує реплікацію, але кожна окрема черга у системі не може бути поширена через проблеми консистентності станів.

Репліки RabbitMQ працюють у асинхронному режимі наслідування, тобто завжди існує головний вузол, та послідовник, який приймає до себе повідомлення у в асинхронному режимі від головного вузла. Асинхронний спосіб наслідування є компромісом між надійністю системи та її швидкодією, однак, такий спосіб унеможливлює також використання послідовника як основної точки входу в систему. Кожне під’єднання до послідовника буде перенаправлене до головного вузла.

Проте варто зазначити, що така модель застосовується до кожної черги окремо, що дозволяє створити примітивні методи балансування навантаження, створюючи різні черги як «головні» на різних вузлах реплікованих вузлів RabbitMQ. Таким чином можна розподілити навантаження по чергам, але кожна окрема черга все ще буде обмежена у своїй пропускній здатності. Саме тому при конструюванні архітектури обміну повідомленнями з RabbitMQ – потрібно планувати її так, аби уникнути надсилання великої кількості повідомлень через єдину чергу, тобто розбивати великі черги на менші та використовувати можливості маршрутизації, що надає RabbitMQ, такі як зазначені вище маршрутизації за «темами» або «заголовками».

Kafka підходить для потокової передачі даних, але він може бути надмірним для простих потреб обміну повідомленнями та вимагає значних зусиль з налаштування та обслуговування, особливо щодо керування журналами даних. Kafka

не є найкращим рішенням для класичної мікросервісної архітектури, що вимагає складного логічного масштабування та ізоляції компонентів.

Незважаючи на те, що ActiveMQ є універсальним, йому може бракувати продуктивності RabbitMQ або Kafka у великомасштабних розгортаннях. Для сценаріїв, які вимагають керування транзакціями та розширеної маршрутизації повідомлень.

Брокери, що імплементують розглянутий раніше протокол MQTT, адаптовані до Інтернету речей, не є найкращим рішенням для складних корпоративних систем, оскільки зазвичай не мають просунутих систем обробки повідомлень, що привиде до ситуації, де таку логіку доведеться імплементувати на стороні проекту, а не інфраструктури.

Таким чином, під час вибору брокера повідомлень необхідно ретельно оцінити ці компроміси щодо конкретних потреб системи. Вибір конкретного брокера має ключове значення не тільки на вартість розробки системи а й її технічну спроможність. Неправильний вибір брокера повідомлень може призвести до значних бюджетних витрат у майбутньому.

1.4 Технології забезпечення доступності та масштабування

1.4.1 Балансування навантаження та кластеризація

Балансування навантаження та кластеризація є основними механізмами, що надають можливість пропорційно збільшувати максимально можливу пропускну здатність розподіленої системи, та мають вирішальне значення для збільшення доступності та масштабованості системи.

За минулі декілька десятиліть, інженери невпинно покращували окремі компоненти системи, такі як центральний процесор та основну пам’ять комп’ютера. Обчислювальні системи пройшли значний шлях розвитку від використання вакуумних труб до мікротранзисторів, дублюючі попередні потужності кожні декілька років відповідно до закону Мура. Проте у сучасних системах все більше

виникає проблем з продовженням покращення технологічної основи, у зв’язку зі складнощами, що виникають на атомічних масштабах та все більше чутливими до ефектів квантової механіки [37].

Саме через такі технологічні обмеження – все більше набирають значення механізми реплікації та кластеризації системи, тобто використання, синхронізація та управління декількох систем одночасно, для вирішення поставленого завдання.

Балансування навантаження у своїй основі означає ефективно розподіляти трафік мережі таких запитів між різними серверами або вузлами, щоб мінімізувати вплив надмірного попиту на єдиний вузол системи. Такий механізм не тільки оптимізує використання ресурсів, але й покращує доступність і час відгуку, оскільки у випадку неочікуваних помилок на одному вузлі – інші можуть прийняти навантаження на себе. Доволі великою популярністю нині користуються хмарні системи, котрі автоматично вміють додавати нові вузли до мережі, спираючись на значення та метрики, що надаються системою моніторингу навантажень [38-39].

Кластеризація означає об’єднання кількох серверів або вузлів, щоб вони могли працювати разом як одна система. Такий метод підвищує надійність системи та відмовостійкість, оскільки збій одного вузла не призводить до збою всієї системи. До того ж, кластеризація забезпечує ефективне масштабування розподілених систем і підтримку високої продуктивності за змінних умов навантаження [40-42].

Важливо зазначити, що хоч балансування та кластеризація мають схожі абстрактні призначення, вони відрізняються своєю побудовою та цілями. Балансування навантаження спрямоване на розподіл однотипних запитів між реплікованими системами, які абсолютно дублюють функції та сервіси що надаються. В свою чергу, кластеризація об’єднує декілька систем для вирішення однієї складної задачі, та вимагає побудови механізмів синхронізації між такими системи, в той час як сервіс, що знаходиться за балансувальником навантаження, зазвичай не має інформації про інших учасників та реплік системи і точно так само не має інформації про механізм, що виконує балансування навантаження [38-42].

Кластеризація доволі часто застосовується у таких задачах як тренування моделей штучного інтелекту, моделюванні всесвіту, складних розрахунках

біоінженерії або астрофізики. Балансування навантаження частіше застосовується у мікросервісних архітектурах сучасних вебдодатків, що дозволяє підтримувати все більшу кількість одночасних під'єднань користувачів.

Технології балансування навантаження та кластеризації є життєво важливими для розподілених систем, особливо зараз, коли швидкість реагування та безвідмовна робота є критично важливими для задоволення користувачів і безперервності бізнесу.

Упровадження цих технологій вимагає ретельного розгляду специфічних вимог програми та архітектури системи. Балансувальники навантаження можуть працювати в різних формах, таких як прикладне програмне забезпечення, апаратне забезпечення або навіть як сервіси третіх сторін, як у випадку з AWS, GCP, або Azure. Кожна з цих форм має свої переваги, наприклад, апаратний балансувальник навантаження типічно значно швидший за програмний, але в той же час вимагає більших інвестицій та моніторингу, в той же час, зовнішні сервіси дозволяють перекласти відповідальність за справність такого компоненту на третю сторону, але вимагають більших фінансових інвестицій.

Будь-яка стратегія кластеризації, будь то активний-активний або активний-пасивний, повинна відповідати цілям відновлення системи та потребам експлуатації. Застосування цих технологій гарантує безперебійну обробку великих обсягів трафіку та стійкість системи до збоїв апаратного чи програмного забезпечення, але також є неймовірно складною у розробці та підтримці. Помилка при побудові механізму синхронізації кластеру може коштувати значних збитків. Наприклад, якщо помилка при синхронізації призвела до неправильних обрахунків складних моделей, інвестовані ресурси на процесорний час будуть витрачені задарма.

1.4.2 Реплікація даних та забезпечення консистентності

Основним механізмом гарантування стійкості розподілених систем є реплікація даних. Такий механізм надає можливості не тільки гарантувати збереження даних у випадку фатальної помилки на одній із систем, але й також надає функції забезпечення доступності даних та оптимізації продуктивності. У випадку, коли

декілька систем мають асинхронну реплікацію інформації – операції читання інформації можуть бути розподілені між системами, в залежності від вимог до консистентності. У випадку синхронної реплікації, розподіленими можуть бути навіть операції запису.

Розподілені системи можуть гарантувати безперервний доступ до даних навіть у разі збою вузла або проблем з мережею завдяки тиражуванню даних на різних вузлах. Реплікація також може покращити роботу системи, дозволяючи доступ до даних із ближчих вузлів, що зменшує затримку. Підтримка узгодженості даних — забезпечення того, щоб усі копії даних залишалися синхронізованими в системі, незважаючи на оновлення чи зміни і є основною проблемою при реплікації даних.

Для вирішення підтримки узгодженості даних існують декілька основних поширених методів використання моделей узгодженості, таких як «сильна» або «причинно-наслідкова» узгодженість, що пропонують різні компроміси між узгодженістю та доступністю. «Сильне» або також так зване «синхронне» узгодження гарантує, що всі вузли переглядають однакові дані одночасно та мають однаковий стан, але це може призвести до більшого використання ресурсів і часу відповіді.

«Причинно-наслідкова», або «асинхронна» узгодженість гарантує, що всі вузли матимуть однакові дані в кінцевому підсумку. Така модель є більш ефективною, але може призвести до тимчасових розбіжностей у даних, що не є прийнятним для цілої низки систем, в першу чергу тих, що вимагають транзакційної обробки запитів. Щоб вирішити проблеми узгодженості, сучасні розподілені системи часто використовують такі методи, як контроль версій, алгоритми вирішення конфліктів і ведення журналу з попереднім записом.

1.4.3 Відмовостійкість і стратегії відновлення після збоїв

Стійкість розподіленої системи визначається як здатність надавати надійні послуги незважаючи на збої в системі. Підвищення стійкості системи залежить від впровадження стандартів дизайну та архітектури, які дозволяють виявляти збої, пом’якшувати їх наслідки і швидко відновлюватися.

Для швидкого та оперативного реагування на інциденти функціонування системи, однією з основних задач архітектора завжди є побудова надійної системи моніторингу. Існує безліч сучасних рішень, що дозволяють збирати так звані «метрики» з цільових платформ, на яких розгорнутий компонент розподіленої системи. Такі рішення можуть оперативно надавати інформацію про стан операційної системи та використовуваного апаратного забезпечення. До того ж, існують безліч стандартів та реалізацій клієнтів, що надають можливості надавати високорівневі метрики рівня системи, для швидкого реагування на інциденти в логіці роботи такої системи.

Одним з основних механізмів забезпечення надійності розподіленої системи є резервування критичних компонентів, щоб будь-який збій у системі не вплинув на всю систему. Впровадження механізмів відновлення після відмови є ключовим елементом стійкості системи, оскільки це дозволяє операціям автоматично перемикатися на резервну систему або вузол у разі збою. Автоматизовані системи оповіщення та процеси самовідновлення також значно підвищують стійкість системи. Такі механізми підвищують ймовірність того, що розподілені системи можуть справлятися з непередбаченими помилками та збоями, не впливаючи на загальну функціональність і досвід користувача [43-44].

У сучасних великих системах, що повинні надавати надійний сервіс мільйонам користувачів одночасно, архітектура передбачає реплікацію не тільки на рівні окремих компонентів, але й повністю незалежних екземплярів систем у різних регіонах. У випадку, коли один з регіонів зазнав стихійного лиха, або інших надзвичайних ситуації – навантаження може бути успішно переадресовано до іншого регіону [43-44].

Хмарні рішення для відновлення та географічний розподіл центрів обробки даних надають можливість гарантувати надійність у разі катастрофи в певному регіоні та автоматизувати процес реплікації системи, проте вимагають додаткових фінансових вкладень.

1.5 Сучасні підходи до розробки та впровадження розподілених систем

1.5.1 Мікросервісна архітектура

Щоб задовольнити вимоги масштабованості, гнучкості та швидкого розвитку, сучасні методи демонструють перехід від монолітних конструкцій до більш модульних і динамічних. Така стратегія дозволяє ефективно використовувати сервіси масштабування, реплікації та балансування навантаженнями, що можуть бути налаштовані як самостійно, так і використані на стороні технології хмарних провайдерів, які дозволяють абстрагувати складнощі пов’язані з підтримкою масштабованої інфраструктури [45-46].

Монолітна конструкція традиційно була основним способом побудови систем, оскільки надавала спрощену модель управління компонентами, доступу до ресурсів та синхронізацією процесів. Однак така модель має критичний недолік, оскільки несправність одного компоненту – неодмінно припинить функціонування всієї системи.

Саме через такі обмеження, у сучасних системах з високими вимогами до навантаження та пропускної спроможності, основним підходом до створення розподілених систем є архітектура мікросервісів. Цей метод передбачає поділ великої програми на менші, незалежно розгорнуті служби, кожна з яких має власний процес і взаємодіє через прості механізми, API (Application Programming Interface) на основі HTTP або gRPC (gRPC Remote Procedure Calls) [43-44].

Таких підхід надає можливість розробляти, розгортати та масштабувати незалежно кожний окремий компонент, що дозволяє створювати масштабовані та гнучкі розподілені системи без критичної залежності від єдиного компоненту. Оскільки кожен мікросервіс можна оновлювати або виправляти без впливу на всю програму, така архітектура сприяє більш стійкій конструкції системи, а також полегшує розробку у випадку, коли декілька віддалених команд працюють одночасно над різними компонентами системи.

1.5.2 Безсерверна архітектура

Сучасні хмарні рішення надають безліч сервісів, що значно спрощують процес розгортання розробленої системи, надаючи готові платформи та навіть системи менеджменту мережею вузлів. Проте, при використанні таких сервісів, клієнти все ще зобов'язані розуміти вимоги до середовища виконання цільового коду системи, а у випадку із системами управління мережею вузлів – розуміти інтерфейс та правила її використання.

Безсерверні середовища надають вищий рівень абстракції над деталями реалізації, управління та підтримки платформи виконання цільового коду програми. Сервіси безсерверного запуску надають можливість розробникам зосереджуватись на єдиній та основній задачі – написанні програмного коду [47-48].

У безсерверному середовищі, хмарний постачальник постійно контролює надсилання та розподіл запитів між функціями, або рутинами, що створюються інженерами на стороні клієнта, а також балансування навантаження між системами, що виконують програмний код таких функцій [47-48].

Використовуючи таку модель, розробники можуть зосередитися на розробці окремих функцій своєї програми, а не на управлінні сервером, що може значно пришвидшити розробку системи та знизити поріг необхідних знань та вмінь.

Безсерверні архітектури керуються подіями та масштабуються з робочим навантаженням автоматично. Розробники платять лише за обчислювальні ресурси, які використовуються під час роботи заданих рутин, або кінцевих точок. Цей метод не тільки робить розгортання простішим, але й допомагає оптимізувати витрати та ефективність роботи [47-48].

Масштабованість, економічність і скорочення часу виходу на ринок – основні переваги використання безсерверної архітектури. Незважаючи на це, необхідно враховувати потенційні складнощі, такі як зв'язок із постачальником, обмеження часу виконання функцій і проблеми з налагодженням і моніторингом.

Важливо зазначити, що технології безсерверного розгортання є обмеженими у своїх можливостях і не є придатними до побудови складних систем синхронізації, та

можуть перетворитись з переваги на недолік, у випадку, якщо у майбутньому у додатку з’явиться залежність або потреба у використанні спеціального інфраструктурного рішення, що не надається стороною провайдера.

1.5.3 Контейнеризація та оркестрація (Docker, Kubernetes)

Ранні розподілені системи вимагали запуску окремої фізичної машини під кожний окремий компонент системи, що вимагало значних ресурсних, фінансових, та адміністративних затрат. Створення перших систем віртуалізації розпочало епоху консолідації сервісів на одній потужній серверній частині. Віртуалізації сервісів досі використовується і надає рівень ізоляції операційних систем, що намагаються одночасно отримати доступ до фізичних ресурсів комп’ютерної системи.

Ізоляція – основна перевага технологій віртуалізації, проте, ізоляція операційних систем вимагає значних додаткових ресурсних витрат та надмірного використання обмежених ресурсів системи. Контейнеризація була створена, аби надати рівень ізоляції між сервісами, але уникнути управління декількома операційними системами одночасно.

Контейнеризація змінила розробку та розгортання програм у розподілених системах, зокрема завдяки таким платформам, як Docker. Docker дозволяє упаковувати програми та їхні залежності в компактні, автономні та ефективні програмні образи, що гарантує узгодженість у різних середовищах, оскільки контейнери включають в себе всі необхідні залежності, бібліотеки, ресурси, та налаштування цільового компоненту системи [49].

У міру зростання використання контейнерів зростає і потреба в інструментах для їх ефективного керування. Існують деякі більш спрощені версії управління контейнерами, що обмежуються єдиним вузлом, як-от «Docker Compose». Розподілені системи можуть бути запущені та керуватись на єдиній фізичній машині. Такий підхід значно спрощує управління контейнерами, проте значно зменшує надійність системи, оскільки несправність апаратної частини зупинить роботу всієї системи без можливості відновлення [50].

Саме для вирішення координації та управління контенерами на різних вузлах була розроблена технологія Kubernetes. Kubernetes — це платформа з відкритим кодом, призначена для автоматизації розгортання, масштабування та керування контейнерними програмами, а також надає механізми групування контейнерів, які складають розподілену систум, у логічні одиниці, що дозволяє спростити доступ до них та їх керування. Самовідновлення, автоматичне розгортання, відкат, виявлення служб і балансування навантаження, а також оркестрування сховища є основними функціями Kubernetes. Ця технологія дозволяє ефективно обробляти сотні, якщо не тисячі контейнерів, що є загальною вимогою для розподілених систем [51-52].

1.6 Постановка завдання

Дана дипломна робота зосереджена на дослідженні механізмів, що виконують функції гарантування стійкості, масштабування, надійності та розробці варіацій архітектур розподілених систем. Додатково, завдання полягає в тому, щоб глибоко вивчити протоколи зв’язку, класичні та новітні системні компоненти, їх архітектуру.

Відповідно до результатів дослідження, представленого у першому розділі роботи, розподілені системи широко використовуються в різних сферах, що створює унікальні проблеми, зокрема щодо забезпечення ефективного зв’язку та стійкості системи. Розробка та впровадження таких систем стало особливо важливим завданням у сучасному цифровому середовищі, де неефективність або збої в системі можуть мати довгострокові репутаційні або фінансові наслідки.

У центрі уваги першого розділу - аналіз і порівняння різних протоколів зв’язку, таких як AMQP і MQTT, а також розуміння їхніх функцій, переваг і недоліків у розподілених системах. Дослідження також проведене щодо брокерів повідомлень, таких як RabbitMQ і Kafka, щоб визначити, наскільки вони підходять для різних архітектур розподілених систем, запропонованих у ході практичної частини даної роботи. Особлива увага буде приділена сценаріям, які вимагають високої пропускної здатності, надійності та обробки даних у режимі реального часу.

Завданням даної дипломної роботи є отримання повного розуміння комунікаційних стратегій і архітектурних компонентів, які сприяють відмовостійкості розподілених систем, а також запропонувати набір найкращих практик і рекомендацій щодо розробки та впровадження таких систем, що включає визначення правильного поєднання протоколів і технологій, які відповідають функціональним вимогам і підвищують надійність і стійкість системи в цілому.

Висновки за розділом 1

У першому розділі даної роботи, було проведено дослідження історії розвитку розподілених систем, протоколів та технологій, що ставали їх основою, підкреслюючи перехід від простих механізмів обміну даними до розширених систем, здатних обробляти складні потоки інформації з високими вимогами до пропускної здатності та ймовірністю випадкового зростання навантаження в режимі реального часу. Така зміна підкреслює зростаючу складність розподілених систем і необхідність рішень, здатних ефективно управляти чергами запитів.

Розділ класифікує та аналізує важливі протоколи, такі як AMQP і MQTT, надаючи розуміння їхніх особливостей, переваг і застосування в різних сценаріях. Розглянуті проблеми масштабованості, та ймовірні рішення, що надаються зазначеними протоколами.

Було проведено порівняльний аналіз, щоб з’ясувати, як брокери повідомлень, такі як RabbitMQ і Kafka, виконують функції гарантування доставки повідомлень, а також балансування навантажень та зменшення впливу сплесків кількості надісланих запитів. В розділі також розглядаються останні тенденції в розробці розподілених систем, включаючи архітектуру мікросервісів, безсерверні обчислення та контейнеризацію за допомогою інструментів Docker і Kubernetes. Зазначені підходи ізоляції, та слабкої залежності компонентів - демонструють еволюцію галузі до більшої масштабованості, модульності та придатності для обслуговування систем, їх розгортання та підтримки.

РОЗДІЛ 2
РОЗРОБКА ВАРІАНТІВ АРХІТЕКТУРИ РЕЗИЛЬЄНТНОЇ РОЗПОДІЛЕНОЇ СИСТЕМИ

2.1 Розробка базової архітектури обробки даних

На рисунку 2.1 зображена архітектура базового сервісу обробки даних:

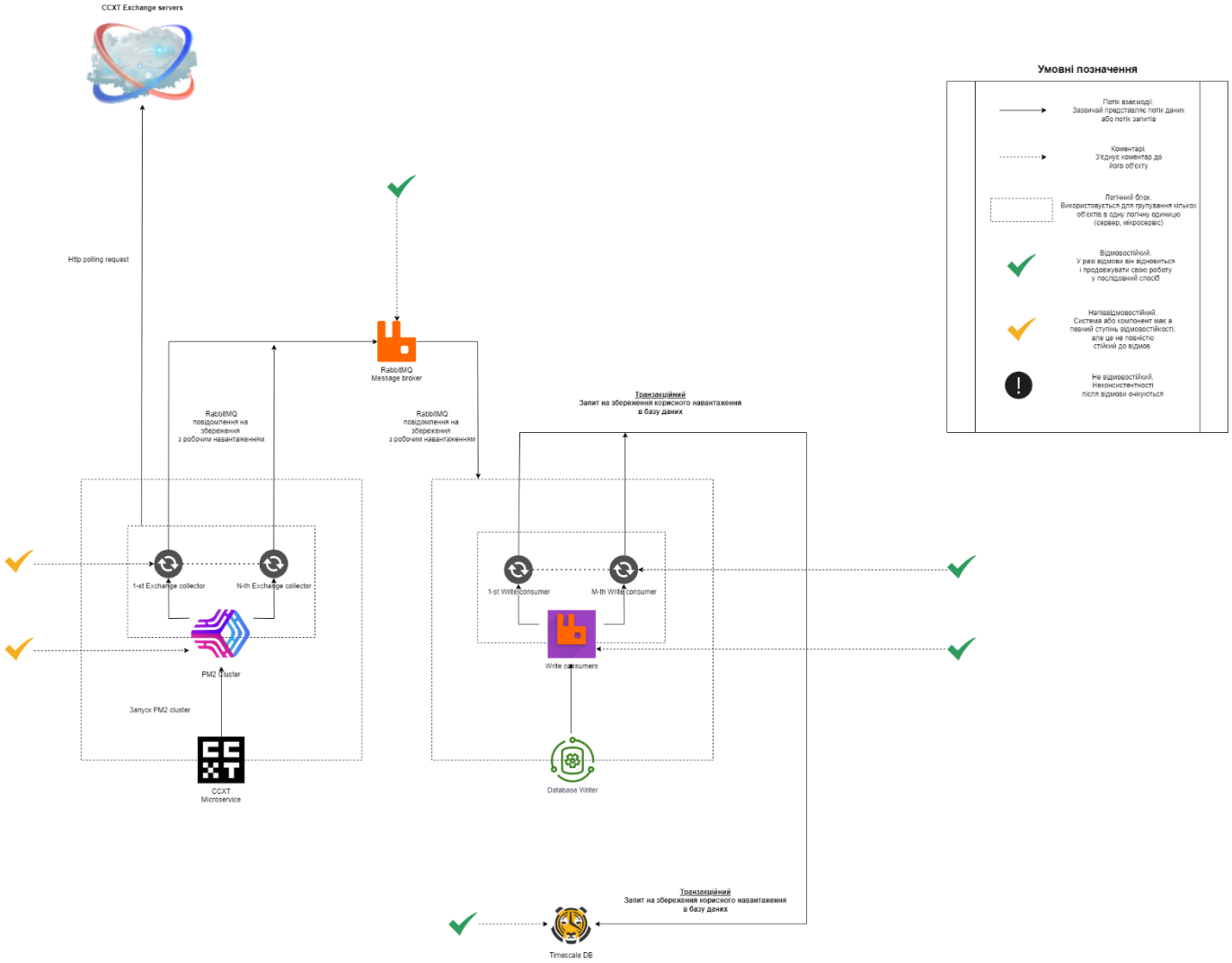

Рисунок 2.1 — Базова архітектура обробки даних

2.1.1 Огляд компонентів та потоку даних

В основі розробленої системи, та наступних архітектур – збір фінансової інформації з різноманітних джерел як традиційної торгівля цінними паперами, так і сучасних бірж криптовалют. В даному підрозділі розглянуто технологічну основу збору та збереження інформації, в той час як у наступних підрозділах буде представлено механізми та варіації архітектур, що будуть покращувати характеристики стійкості та надійності системи.

Вхідний функціонал системи починається із інфраструктури збору інформації. Така інфраструктура і буде частиною кластеризованого розподілу системи, в той час як інфраструктура доступу до інформації буде мати слабкий зв’язок між реплікаціями та буде використовувати механізми балансування навантаження, описані у попередньому розділі. Завдяки такій побудові, робота має на меті продемонструвати хід прийняття рішень щодо механізмів масштабування.

Кластер збору інформації буде складатись із ряду синхронізованих мікросервісів, що звуться «колекторами», або збірниками інформації. У даному підрозділі, на рисунку 2.1, колектори представлені як сервіси, що використовують стандартизований набір рутин, тобто драйверів, для доступу до безлічі існуючих систем, що надають фінансову інформацію, таку як агреговані ціни торгівлі, фактичні обміни цінними ресурсами та аналітичну інформацію.

Центральним компонентом комунікації було обрано RabbitMQ у зв’язку з необхідністю складної маршрутизації повідомлень відповідно до їх типу, а також спрощеного інтерфейсу управління гарантуванням доставки та балансування навантаження. Можливості RabbitMQ щодо збереження буферу повідомлень у черзі дозволяють впроваджувати стратегію повторної обробки інформації, отриманої від інфраструктури збору.

RabbitMQ покращує стійкість системи та відмовостійкість, відділяючи пошук даних від обробки, а також надає можливості до покращеної модульності системи, та її масштабування, завдяки розподілу різних типів повідомлень у внутрішній реплікації сервісів черг.

Архітектура RabbitMQ надає переваги за рахунок асинхронної обробки, балансування навантаження та здатності обробляти робочі навантаження з високою пропускною здатністю завдяки надійній системі черги повідомлень.

«Database Writer» є кінцевим сервісом стандартної інфраструктури збору інформації та відповідальний за запис такої інформації до бази даних та транслювання події запису в інші частини системи. Database Writer працює у транзакційному режимі та гарантує, що дані зберігаються в узгодженому стані в TimescaleDB.

TimescaleDB, в свою чергу, є реляційною базою даних часових рядів, та, відповідно, підтримує SQL (Structured Query Language). TimescaleDB була розроблена для швидкого запису та оптимізованого зберігання інформації, що представляє собою хронологічну послідовність [53-54].

У поєднанні з налаштованим механізмом повторних спроб, транзакційна природа Database Writer забезпечує успішне завершення операцій запису та обробку можливих помилок, що значно зменшує ймовірність неконсистентності збереженої інформації. Однак, таке рішення не достатнє для гарантування послідовності збору фінансової інформації, що буде центральною темою обговорення та основною проблемою, рішення для якої будуть представлені у даному розділі.

2.1.2 Опис сервісу збору даних

На рисунку 2.1, служба збору даних «колектор» - відповідальна за встановлення комунікації з великою кількістю джерел, котрими можуть виступати як конкретні біржі, так і вузли з мережі блокчейну.

Кожний сервіс третьої сторони, що має публічно доступний HTTP REST (Representational State Transfer) API, або інтерфейс RPC – вимагає виконувати запити із дотриманням вимог системи обмежень доступу, для уникнення випадків надмірного використання системних ресурсів зі сторони єдиного клієнту, а також уникнення атак типу DOS та DDOS. Механізмом, що імплементує такі політики, може бути як просте обмеження за IP-адресою та ключем доступу до API, або ж, механізм спливаючого пулу ресурсів. Суть реалізації першого механізму полягає у фіксованій

кількості запитів, що може виконувати клієнт за встановлену одиницю часу, що є значно простішим з точки зору навантаження та програмної реалізації, проте не враховує можливості тимчасового сплеску запитів від користувачів, які можуть статись, наприклад, при початковому завантажені сторінки.

В свою чергу, механізм на основі пулу запитів – припускає можливість сплеску навантажень, та обмежує кількість запитів не в дискретних спільних одиницях, а для кожного запиту окремо. В такій системі, право на виконання наступного запиту надається після закінчення періоду очікування після попереднього, та не поширюються на інші запити в системі.

Саме через такі обмеження та вимоги третіх сторін, однією з основних задач сервісів збору інформації – побудувати спосіб синхронізації та системи управління частотою та складністю надісланих запитів. Тому, центральним компонентом сервісу «колектора» є клас планувальника, що відповідальний за прорахування час виконання операцій із точністю до мілісекунд.

Система збору інформації має критичну залежність від коректності функціонування рівня планування частотою, оскільки у випадку неправильно заданої частоти – адресу, або ключ до API буде заблоковано і, як наслідок, система збору буде мати значну втрату інформації, котра доступна лиже у режимі збору в реальному часі, а також, доведеться купувати нову адресу, або ключ доступу. Проте, у випадку неповного використання частотних можливостей, дискретність отриманої інформації буде знижена, що, в свою чергу, знизить цінність впровадження системи та її рентабельність.

Побудова системи збору та агрегації фінансової інформації не має цінності, у випадку використання єдиної біржі як джерела, та потребує широкої інтеграції з багатьма існуючими платформами торгівлі. Оскільки поставлене завдання потребує збору інформації із великої кількості різноманітних систем третіх сторін, архітектура сервісу «колектора» повинна враховувати та створити рівень абстракції над джерелами інформації.

Стандартний драйвер оптимізує взаємодію між системою та зовнішніми серверами для забезпечення однорідного обміну даними між різними типами служб.

Ця одноманітність необхідна для підтримки постійного та ефективного процесу збору даних і дозволяє також значно спростити розробку рівня планування частот, синхронізації процесів та їх масштабування.

Різні біржі та платформи доволі часто працюють за різними схемами побудови транзакцій та обміну цінними ресурсами, що створює необхідність до ще одного рівня абстракції, що ізолює тип зібраної інформації від рівня планування та відповідальний за цільове використання рівня драйверів.

Для вирішення цього завдання, служба пропонує набір класів збирачів для обробки різноманітних типів даних, що призначені для отримання та форматування конкретних типів даних, та мають схему конкретної взаємодії із єдиним інтерфейсом драйверів. Наприклад, розроблений спеціальний клас «CandleSticksCollector», який відповідальний за збір інформації про рух цін з часом у фінансах. Подібним чином створені та існують окремі класи для отримання відповідних типів даних: «TickersCollector», «TradesCollector» та «OrderBooksCollector», кожен з яких виконує форматування даних для подальшої обробки. Така ізольована модульна структура класу дозволяє цілій системи спеціалізуватися на різних типах даних, що легко інтегруються без змін рівня управління, а також інкапсулює логіку для кожного типу, що робить систему масштабованою та простішою в обслуговуванні.

2.1.3 Опис сервісу обробки та зберігання даних

На рисунку 2.1, служба збереження даних «Database Writer» - відповідальна за отримання повідомлень від RabbitMQ, перевірки корисного навантаження та передачі його в базу даних. Використання окремого сервісу та інфраструктури для збереження отриманої інформації дозволяє розподіл рівнів відповідальності отримання та збереження інформації у різні сервіси, що дозволяє впровадити механізми підтримки надійності збереження та обміну інформації, що вже перейшла периметр відповідальності та контролю сторони системи обробки.

Служба виконує обробку кожного повідомлення у транзакційному режимі та перевіряє вхідні дані відповідно до схеми валідації, яка була визначена заздалегідь та

у випадку невідповідності – відхилить збереження небажаної інформації. Така перевірка допомагає запобігти зберіганню даних, які є неправильними або пошкодженими, що може скомпрометувати консистентність системи, та аналітичних висновків, зроблених на основі автоматизованих агрегацій.

При побудові системи, що має декілька розподілених компонентів, та де спосіб обробки інформації покладається на надсилання повідомлень від одного вузла до іншого – існує ймовірність дублювання повідомлень через помилки обробки або надсилання даних. Для вирішення таких проблем, сервіс запису інформації попередньо перевіряє, чи задане повідомлення за окремий часовий ряд – було вже записано у постійному сховищі.

Як тільки перевірка буде завершена, служба спробує записати дані в TimescaleDB і у випадку, якщо операція запису не вдалася, повідомлення не підтверджується. Така помилка може статися через порушення обмежень бази даних, проблему підключення або тимчасову помилку, котра може бути виправлена через певний час системними адміністраторами. Замість відкидання, повідомлення у такому випадку перенаправляється до обміннику «мертвими» листами, який налаштовано у RabbitMQ, що забезпечує збереження повідомлень та повторну спробу запису через налаштовану кількість часу. Механізм тайм-ауту в обміні «мертвими» листами дозволяє повторно обробляти повідомлення через певний проміжок часу, та явно вказувати кількість можливість спроб і необхідну кількість часу до наступної спроби.

2.1.4 Інтеграція RabbitMQ для забезпечення надійної комунікації

RabbitMQ полегшує відокремлення сервісів «колекторів» для збору інформації з віддалених джерел та сервісу запису інформації у базу даних «Database Writer». RabbitMQ надає буфер для зібраної інформації та повідомлень, який знижує вплив сплесків частоти надсилання даних або спалахів трафіку, що запобігає втраті даних і дозволяє системі підтримувати стабільну швидкість обробки незалежно від мінливості навантаження.

Цілком очікувано, що система, яка повинна збирати фінансову інформації та агрегувати дані з великої кількості джерел, повинна буде оброблювати великий потік інформації та мати відповідну пропускну здатність.

Як було розглянуто у попередньому розділі, Kafka краще справляється із великими об’ємами трафіку, проте рішення на користь RabbitMQ буле прийнято у зв’язку з наступними причинами:

– додаток потребує механізмів складної маршрутизації, оскільки система повинна підтримувати велику кількість типів даних, що збирається із систем третіх сторін, а також, система повинна підтримувати можливості розсилки та базову основу для побудови механізмів синхронізації на основі протоколу AMQP;

– механізм обміну мертвими листами RabbitMQ є центральним інструментом для ефективного управління збоями обробки повідомлень, що дозволяє системі, та її окремим компонентам повторювати операції без втрати повідомлень;

– використання RabbitMQ підвищує стійкість системи, дозволяючи їй протистояти ситуаціям, коли система залежна від нестабільності роботи мережі та збоям окремих компонентів завдяки механізмам черг повідомлень, асинхронній реплікації черг, а також збереженню повідомлень на диску;

– RabbitMQ підтримує тривалі черги, зберігаючи повідомлення до тих пір, поки вони не будуть успішно оброблені, навіть у разі збоїв або перезавантаження системи, що нівелює вплив випадкових інфраструктурних помилок;

– масштабованість системи досягається завдяки механізму реплікації, що вже імплементовані у брокері RabbitMQ, та дозволяють балансувати навантаження завдяки розподілу черг між вузлами та створення їх копій, для досягнення високої доступності системи.

У наступних розділах та при розробці цільової розподіленої системи, RabbitMQ буде використано не лише для надійного обміну та передачі отриманої інформації до сервісу збереження, але й для маршрутизації інформації про події та стани сервісів, а також створення систем підписок на асинхронні завдання.

2.2 Розробка архітектури з механізмом зворотного запису інформації

На рисунку 2.2 зображена архітектура сервісу з механізмом зворотного запису інформації:

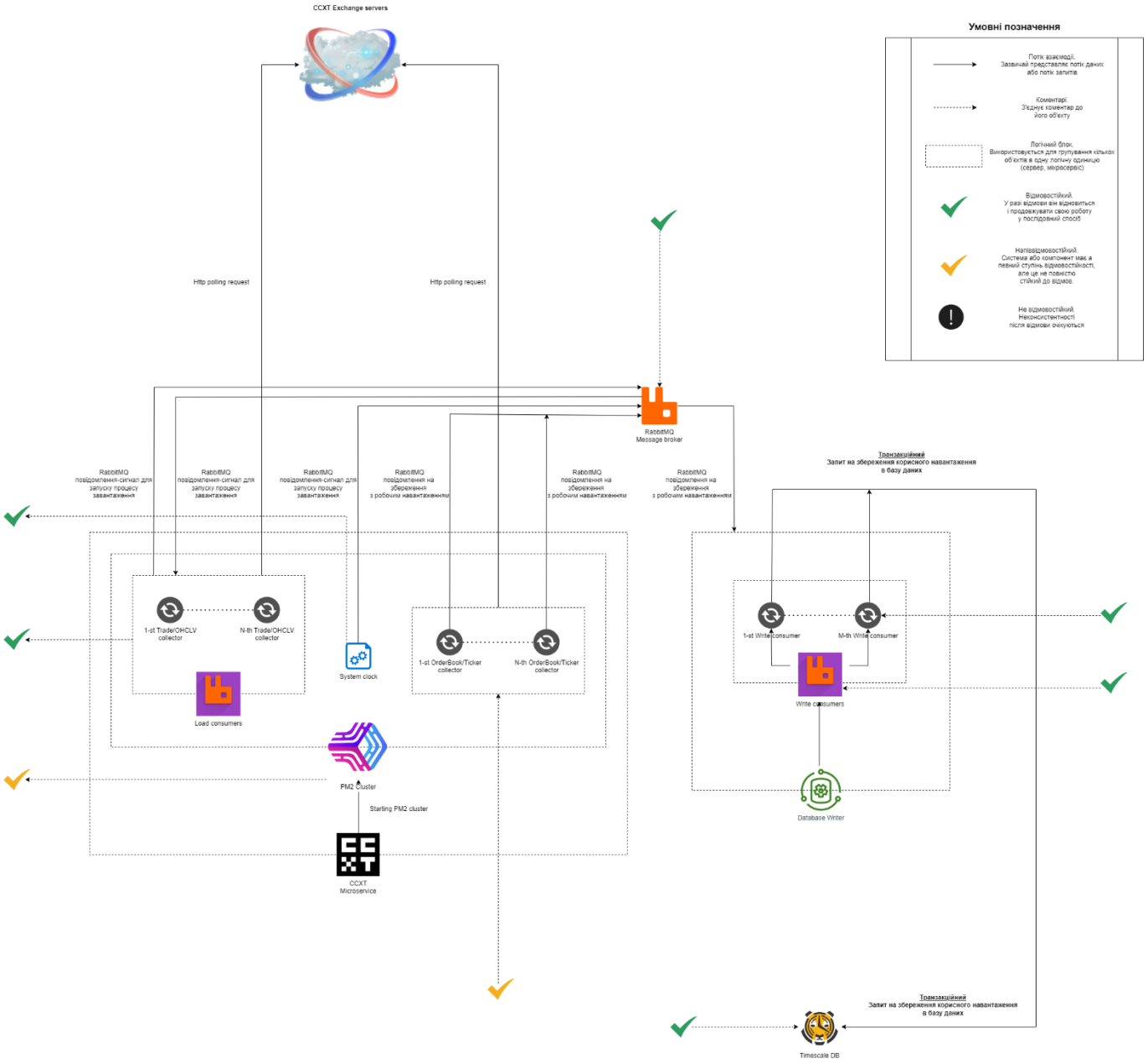

Рисунок 2.2 — Архітектура з механізмом зворотного запису інформації

2.2.1 Принципи та архітектура механізму відновлення даних

У розділі 2.1 було продемонстровано базові компоненти інфраструктури, що виконують функції збору інформації, планування частот збору, транспорт інформації, та її запис до постійного сховища. Було вирішено гарантування надійності передачі да процесів оброки інформації в зоні периметру контролю системи за допомогою механізмів черг та повторної обробки запитів. Однак, такі механізми не зменшують ризики, що знаходяться за зоною контролю системи.

Архітектура, зображена на рисунку 2.2, демонструє перший з розроблених підходів гарантування послідовності та консистентності стану системи, що широко використовується в різних сферах розподіленої обробки інформації.

Така архітектура передбачає можливості створення механізму для відновлення втрачених даних, який працює рекурсивно у зворотному хронологічному порядку. Суть механізму полягає в тому, аби повторно надсилати запити до зовнішніх сервісів на отримання інформації про стан минулого такту збору. Цей механізм доволі часто застосовується у системах індексації інформації щодо виконаних транзакцій, отриманих з одного з вузлів мережі блокчейну.

Мета механізму полягає в тому, щоб запускати його в певних ситуаціях, наприклад, у тих випадках, коли виявляються розбіжності в даних, їх пропуски у часовому ряді, або після перезапуску системи, щоб отримати історичні дані, які не були зібрані чи оброблені через збої або інші причини.

Для підтримки функціонування такого механізму, та гарантування виконання операції збору інформації, в архітектурі передбачено використання можливостей RabbitMQ для повторної обробки запиту, що збережений у черзі та впровадження нового компоненту «системного годинника», котрий відповідальній за генерацію запитів і виконання функції контролю частоти надсилання таких запитів.

Системний годинник повинен містити в собі схеми запитів, що будуть надіслані до рівня колекторів та інтерпретовані у запити до зовнішніх джерел інформації. Цей компонент повинен контролювати частоту генерацій повідомлень, проте рівень колекторів все ще повинен відповідати за планування операцій, що будуть надіслані

до серверів третіх сторін, що створює додаткову складність управління, та гарантування відповідності вимогам безпеки джерел інформації.

Database Writer зберігає те саме головне призначення запису інформації у базу даних та все ще придатний до використання за допомогою інтерфейсу, описаного у попередньому підрозділі. Проте у випадку, якщо ліміт кількості повторних спроб запису інформації у базу вичерпано, або надане повідомлення не пройшло перевірку формату та встановлену валідацію, Database Writer повинен сповістити компонент системного годинника для повторного збору інформації.

2.2.2 Оцінка ефективності та недоліки механізму відновлення

Механізм зворотного запису інформації дозволяє відновити консистентність стану системи у випадку збоїв, або інших неочікуваних подій у системі. Такий механізм у випадку постійного та необмеженого доступу до джерела інформації – простий у імплементації та підтримці та не потребує ні значних зусиль, ні часу на розробку, ні значних додаткових вкладень у фізичну інфраструктуру.

Проте, такий механізм має критичну залежність від інтерфейсу взаємодії та підтримки історичного збору інформації на стороні зовнішніх серверів, що робить його обмежено придатним до застосування при інтеграції великої кількості сторонніх сервісів, адже значна частина з них можуть зовсім не підтримувати зберігання та видачу історичної інформації, обмежувати типи інформації, або час її зберігання.

Беручи до прикладу рівень драйверів, описаний у підрозділі 2.1, що заснований на популярній бібліотеці CCXT, котра абстрагує деталі API централізованих бірж, отримання інформації OHLCV (Open, High, Low, Close, Volume) – підтримується більшістю обмінників, проте деякі мають обмеження збору історичної інформації не більше ніж 15 хвилин у минулому. В той же час, збір інформації про замовлення в історичному режимі не підтримуються біржами взагалі [55].

Додатковою проблемою є складність побудова системи контролю частоти надсилання записів у системах, де залежність існує від джерел інформації, що обмежують використання їх API.

2.3 Розробка архітектури на основі синхронізованої реплікації сервісів

На рисунку 2.3 зображена архітектура на основі синхронізованої реплікації сервісів:

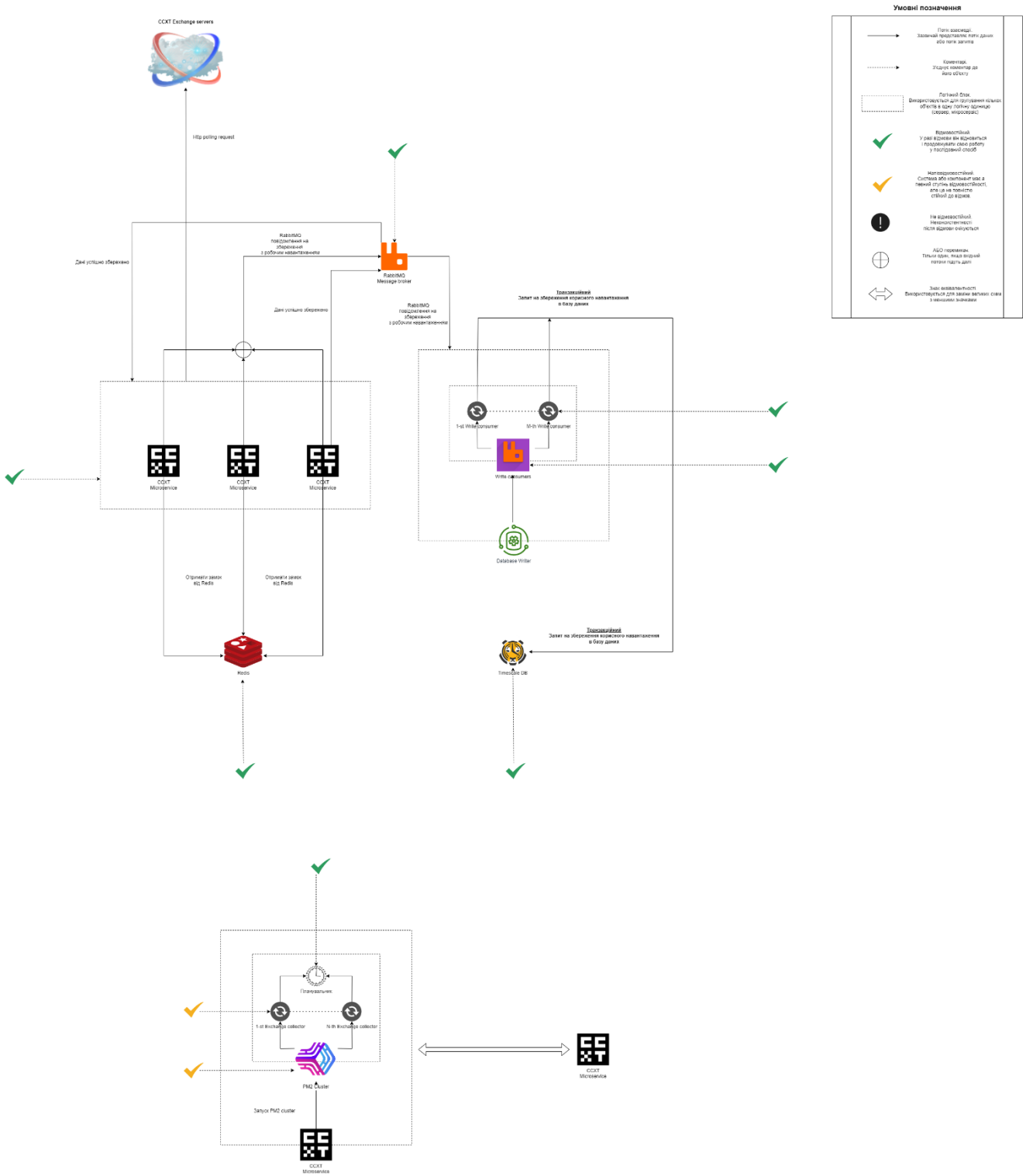

Рисунок 2.3 — Архітектура на основі синхронізованої реплікації сервісів

2.3.1 Стратегії синхронізації віддалених реплік

В архітектурі описаній в підрозділі 2.2, та відповідному рисунку було представлено потенційний механізм, що дозволяє зменшити вплив непередбачуваних інцидентів в мережі, або на стороні джерела інформації на послідовність інформації, що зберігається у контрольованій системі, проте, як було показано, його ефективність цілком і повністю залежить від підтримки ключових функцій збереження та надання історичної інформації за сторони джерела інформації.

Для побудови надійної системи, що повинна інтегрувати значну кількість сторонніх сервісів та можливих конфігурацій, що диктуються такими сервісами, зазначений механізм не є найкращим рішенням, та потребує іншої схеми підвищення надійності інфраструктури збору інформації, оскільки можливість відновлення інформації покладається повністю на механізми, що реалізовані третіми сторонами та у випадку відсутності яких – нівелюють ефективність розробленої архітектури. Враховуючи, що результатом інтеграції можуть бути тисячі під’єднаних джерел інформації, існує висока ймовірність того, що значна частина таких сервісів будуть мати відмінні або повністю відсутні механізми отримання історичної інформації, необхідної для відновлення консистентності системи.

Одним з фундаментальних принципів гарантування надійності будь яких систем, буть то системи дискового збереження інформації, або обробки потоку інформації чи надання інших сервісів, що вимагають високої доступності – є надійність через надмірність (Resilience Through Redundancy).

Застосування такого принципу в архітектурі цільової системи призвело до рішення побудови системи реплікації сервісів колекторів, та їх використання для досягнення ефекту «хоча б один». Якщо хоч один колектор зможе успішно отримати інформацію від зовнішнього джерела інформації – операція вважається успішною.

Імплементації такої реплікації відноситься до задач типу побудови кластерних систем, оскільки, як описано у першому розділу цієї дипломної роботи, декілька реплікованих сервісів повинні виконати єдину задачу, координуючи хід виконання операцій між собою та синхронізовуючи результати виконання.

До того ж, варто пам’ятати обмеження, що накладаються специфічними вимогами системи, такі як обмежений доступ до зовнішніх API. Кластеризована система, повинна вміти планувати виконання запитів таким чином, аби не порушувати політики зовнішних джерел, що може привести до блокування API ключів колекторів, або їх IP адрес. Це призводить до ситуації, де система повинна гарантувати, що виконання запитів зі сторони декількох реплікованих колекторів одночасно – но є можливою ситуацією.

У випадку архітектури, зображеної на рисунку 2.3, для виконання завдання синхронізації та координації системи використовується Redis. Redis надає простий механізм розподіленого «замка», що дозволяє уникати одночасного виконання операцій та входів у критичну секцію різних процесів у системі.

Механізм замків – є фундаментальним методом синхронізації процесів, що використовується не тільки у контексті розподілених систем, але й для синхронізації операції, що виконуються незалежно одна від одної. Яскравим прикладом слугує їх використання при синхронізації процесів в операційній системі, що намагаються отримати доступ до одного й того ж ресурсу одночасно [56].

Перед тим як розпочати запит до зовнішнього джерела, колектор повинен отримати замок від Redis, або чекати, якщо він зайнятий. Операції отримання інформації розділені у інтервальні проміжки, що диктуються політикою обмежень кількості запитів джерела інформації [57].

При першому запуску, кожен з колекторів буде намагатись отримати замок, проте отримати його зможе лише перший, що звернувся до Redis. Колектор не повинен віддавати замок і повинен оновлювати його «час життя», тобто кінцеву дату його дійсності.

Якщо колектор не оновив його кінцеву дату, що повинна займати 1/n часу від періоду збору інформації, де n – кількість реплік, операція вважається проваленою, тож інший колектор отримає замок у власне володіння. Цей цикл може продовжуватись стільки часу, скільки необхідно для функціонування системи.

2.3.2 Оцінка ефективності та обмеження синхронізованої реплікації

Механізм реплікації, що був описаний у даному підрозділі – значно простіший у імплементації, оскільки не потребує додаткових рівнів управління і може врахувати наявність інших систем завдяки статичній конфігурації. Динамічне виявлення реплікацій можливе, проте не є необхідним у такій системі у випадку, якщо немає вимог до самостійного динамічного масштабування системи.

Механізм реплікації нівелює вплив зовнішніх чинників, таких як завантаженість мережі, перебої у з'єднанні, або втрата фізичної платформи на консистентність зібраної інформації.

Таке рішення не має недоліку залежності від функціоналу та механізмів джерела інформації, про те створює складнощі у випадку, якщо колектор зібрав та надіслав інформацію до сервісу запису в базу даних, але не зміг оновити право власності замка, що призведе до дуплікації та неконсистентності даних.

До того ж, в інфраструктурі збору інформації додається ще одна критична залежність у вигляді сервісу Redis, у випадку несправності якого, колектори не зможуть синхронізувати та координувати власні операції.

Система все ще не зможе гарантувати консистентність, у випадку, якщо сервіс зламається на стороні джерела інформації, але, як було продемонстровано у попередньому підрозділі 2.2, відновлення після таких збоїв цілком і повністю залежить від механізмів, імплементованих на стороні стороннього сервісу.

Потенційне поєднання архітектури на рисунку 2.3, з механізмом зворотного хронологічного запису інформації зможе максимально зменшити проблеми, пов'язані з втратою послідовності записів в обох випадках інцидентів на стороні комунікаційної інфраструктури та джерела інформації. Проте, така система буде потребувати побудови протоколів синхронізації запитів зворотного запису та збору у режимі реального часу у випадках обмежень API запитів на стороні сервісу третьої сторони.

2.4 Розробка архітектури з модифікованим протоколом вибору лідера

На рисунку 2.4 зображена архітектура з модифікованим протоколом вибору лідера:

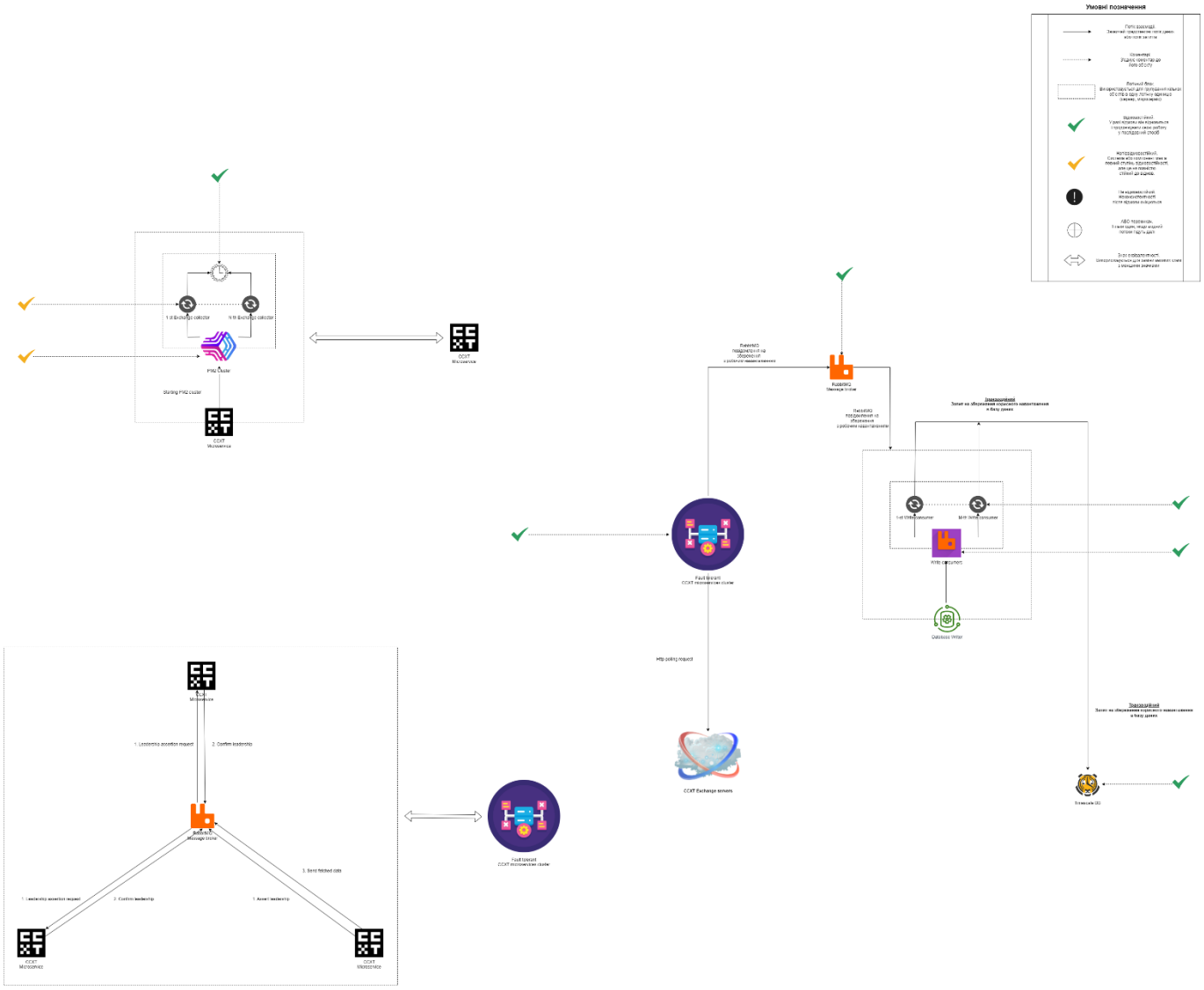

Рисунок 2.4 — Архітектура з модифікованим протоколом вибору лідера

2.4.1 Детальний опис модифікацій протоколу RAFT

Попередня ітерація архітектури, зображена на рисунку 2.3, мала критичну залежність від доступності додаткового сервісу, що надає можливість створювати замки для логічної синхронізації розподілених та віддалених процесів, та, до того ж,

допускала ймовірність неконсистентної поведінки системи у певних випадках стану перегонів.

Задача наступної розробленої архітектури – звести залежність від інфраструктури, необхідної для синхронізації, до вже існуючого набору компонентів у системі, а також побудувати логічну послідовність збору інформації та її зберігання, яка зменшує ймовірність впливу стану перегонів на функціонування системи.

Для вирішення зазначених проблем, одним з механізмом гарантування узгодженості є алгоритм вибору лідера. Існує цілий ряд розроблених алгоритмів, що описують послідовність кроків учасників мережі для вибору єдиного вузла, котрий зветься лідером, та виконує функції управління операціями кластеру. У даному випадку, всі учасники реплікованої мережі повинні надіслати запит на отримання інформації, проте тільки лідер репліки у кластері повинен надіслати результат збору інформації з віддаленого джерела, таким чином уникаючи дублювання інформації при збереженні а також досягаючи стійкості у випадках тимчасової недоступності до мережі на одному з вузлів.

Одним з таких алгоритмі є RAFT, функції якого були модифіковані для особливих потреб системи в контексті виконання операцій збору інформації від зовнішніх джерел та надсилання результатів до сервісу запису інформації до постійного сховища.

RAFT був розроблений для забезпечення постійної доступності та узгодженості в кластері, взаємодія якого починається з процесу вибору лідера для координації ресурсів. Спочатку всі сервери починають як послідовники в пасивному стані під наглядом лідера. Якщо послідовник не отримує своєчасне оновлення, яке вказує на потенційний збій лідера або розділу мережі, він переходить у стан кандидата та починає новий термін виборів. Після цього цей кандидат сам голосує, а також запитує голоси з інших серверів [58-59].

Для встановлення нового лідера потрібна більшість голосів від існуючих налаштованих вузлів всередині кластеру. Якщо кандидату не вдається отримати більшість за допомогою розділення голосів або якщо інший сервер претендує на лідерство протягом одного або більшого терміну, протокол вимагає повернення

внутрішнього стану такого кандидату до послідовника та підготовку до наступних виборів, щоб зберегти послідовність і надійність кластера.

Після остаточного вибору лідера, всі колектори почнуть збирати інформацію, не залежно від статусу, однак, у цій модифікованій системі, лише обраний керівник має право передавати дані до RabbitMQ, який працює як механізм черги для сервісу запису інформації до бази даних. Таке розрізнення гарантує, що навіть якщо усі вузли оновлені, лише одне надійне джерело — лідер — може запускати операції запису, що дозволяє уникнути дуплікації повідомлень, котрі передаються через брокер, а також уникнення конфліктів та дуплікації інформації записаної в базі даних.

Після того, як запит на збереження успішно надіслав керівник, він надсилає сповіщення послідовникам, що підтверджує завершення поточного циклу синхронізації. Розмежування ролей у протоколі зменшує потребу в інструментах синхронізації за межами протоколу, таких як Redis і усуває окремі точки збою, пов’язані з цими залежностями.

Основною перевагою модифікованого протоколу RAFT є його стійкість і здатність до управління відмовами на стороні учасників кластеру, оскільки, якщо головний вузол стикається з проблемою або не може зв’язатися протягом очікуваного часу очікування, система залишається без лідера, що призводить до проведення повторних виборів.

2.4.2 Оцінка ефективності застосування модифікованого протоколу

Механізм вибору лідера дозволяє автономно виконувати синхронізацію кластера та консенсусно встановлювати вузол, відповідальний за відправку інформації. Така централізація управління та надсилання даних до черги повідомлень дозволяє уникнути дуплікації трафіку, конфліктів інформації у базі даних та забезпечити їх послідовність.

Рішення із застосуванням протоколів досягнення консенсусу є масштабованим, оскільки не має ніяких логічних обмежень щодо кількості учасників кластеру, а також

високодоступним, оскільки у випадку несправності лідера – одразу розпочнеться повторний раунд виборі для призначення нового відповідального вузла.

У такого рішення відсутня залежність від додаткового інфраструктурного сервісу, окрім як RabbitMQ, котрий вже використовується як основа комунікації у системі. До того ж, RabbitMQ слугує гарантом доставки та обробки запитів на вибори лідера та обробку голосів.

Недоліками використання такого механізму є пряма залежність від швидкості мережі та навантеженості брокера обміну повідомленням, оскільки, у випадку, коли хоча б один з цих елементів призводить до затримок – процес вибору лідера може зайняти значну частину інтервалу збору інформації, та, врешті, в крайніх випадках втратити успішно завантажену інформацію, консенсус щодо відправки якої не було досягнуто за час інтервалу збору.

Для проведення фази синхронізації, що включає в себе вибір лідера кластеру – потрібні додаткові процесорні та мережеві ресурси, що, зазвичай, не є значною проблемою, проте, така фаза також призведе до збільшення часу, необхідного для досягнення консенсусу. Така система обмежує потенційну фізичну масштабованість у випадках, коли кількість вузлів-учасників сягає високих значень, оскільки для верифікації голосів – кандидати повинні опитати кожного учасника окремо.

Варто пам’ятати, що використання публічних джерел інформації з політиками безпеки, спрямованими на обмеження кількості запитів від одного клієнта, робить час ресурсом, який вимагає найкращого планування та використання. Коли головним завданням є максимальне скорочення часу, що виділяється на синхронізацію, така система не є найкращим варіантом.

Збільшення складності системи завжди призводить до підвищення ймовірності помилок, в тому числі помилок синхронізації. У випадку провалу вибору лідера та критичній помилці системи – збір інформації буде перервано або продовжено у неконсистентному вигляді. До того ж, підвищена складність вимагає більших затрат на розробку та більш досвідчених інженерів.

2.5 Розробка архітектури з ідемпотентною реплікацією

2.5.1 Розробка механізму ідемпотентності на основі планувальника

На рисунку 2.5 зображена архітектура з механізмом ідемпотентності на основі планувальника:

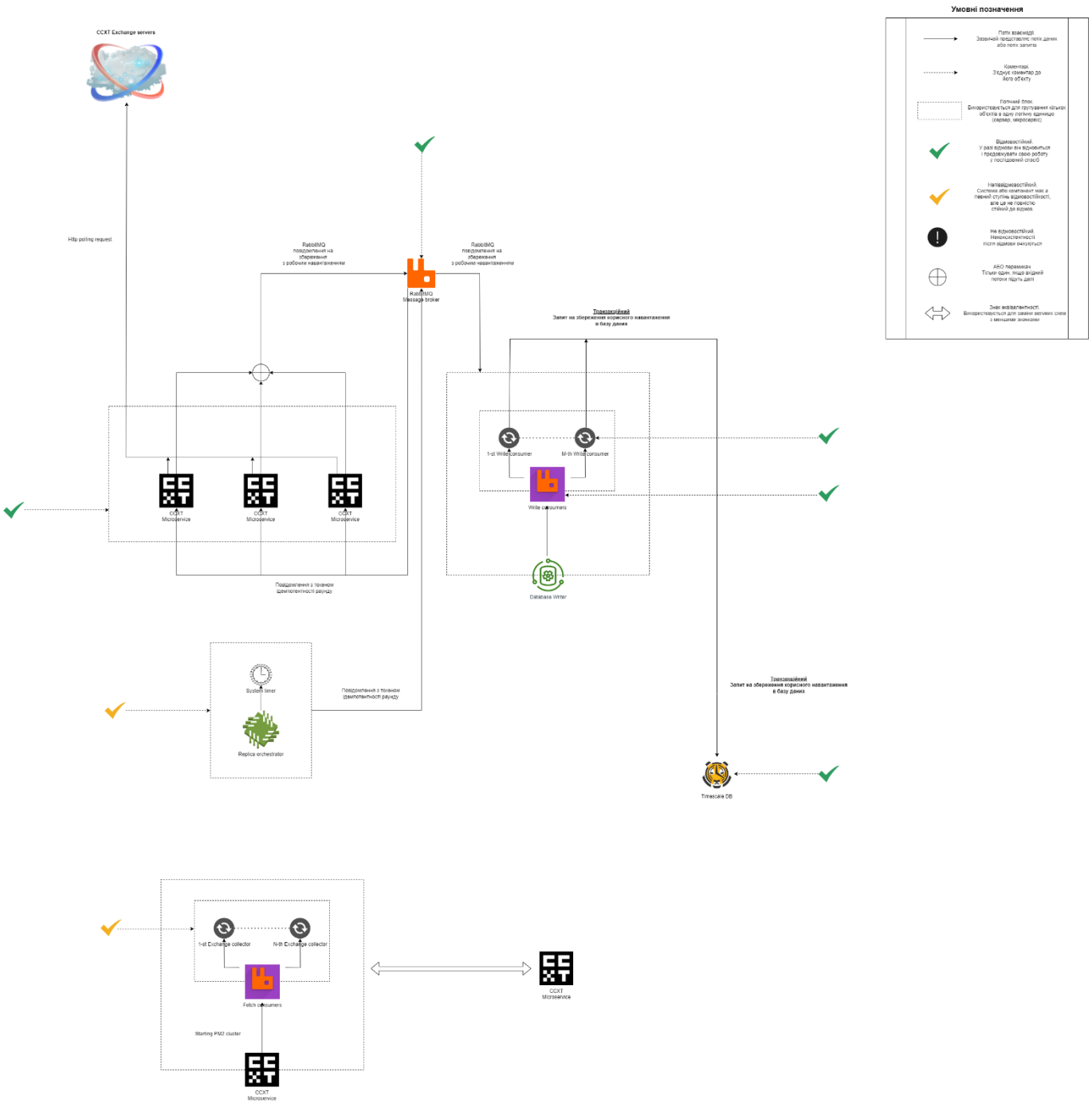

Рисунок 2.5 — Архітектура з механізмом ідемпотентності на основі планувальника

Як було зазначено при розгляді архітектури, зображеної на рисунку 2.4, час – це ресурс, що є доволі критичним фактором для застосування методів синхронізації, що вимагають складної системи досягнення консенсусу. Ресурс часу буде витрачено на механізм, що не застосовує його для виконання бізнес-функцій, та призведе до втрат. Завданням, у такому випадку, - є розробка механізмів, що вимагають мінімального впливу рівня контролю кластеру та його синхронізації.

Архітектура, зображена на рисунку 2.5 – має спільний компонент з архітектурою, зображеною на рисунку 2.2, а саме – компонент «оркестратора репліки», що повинен виконувати функції, котрі покладались на компонент «системного годинника», але також забезпечувати ідемпотентність операцій.

Ідемпотентність – одна з характеристик методу, що означає повторюваність результатів після повторних використань та відкидає можливість сторонніх ефектів у системі. Застосування ідемпотентної функції припускає повторне її застування, проте гарантує, що таке застосування не призведе до змін результатів [60-61].

Тож, чим саме такий принцип може зарадити у випадку розроблюваної системи? Застосування принципів ідемпотентності дозволяє взагалі не проводити етап синхронізації для надсилання результатів збору інформації з віддалених джерел. Оскільки кожна операція запису буде мати такий самий ефект як попередня – стан системи, та послідовність запису часових рядів у базі даних – повинні залишитись у консистентному вигляді.

Механізм ідемпотентності у такому випадку працює на основі токенів ідемпотентності і вимагає певних модифікації до сервісу Database Writer.

Оркестратор репліки, разом із командою початку завантаження інформації до колекторів – повинен прикріплювати токен ідемпотентності, що може бути згенерований випадковим чином, але мати достатній простір для уникнення колізій. Колектори, отримавши команду, повинні виконати збір інформації, але при надсиланні результатів до Database Writer – повинні прикріпити токен ідемпотентності, отриманий від команди.

Database Writer, при отриманні чергового корисного навантаження – повинен спочатку перевірити наявність токена ідемпотентності у швидкодоступному місці для

зберігання, такому як оперативна пам’ять, що керується сервісом Redis. Якщо токен було знайдено – запис не відбувається, якщо не знайдено – токен зберігається у Redis, а корисна інформації – у базі даних.

2.5.2 Застосування хешування для забезпечення ідемпотентності

На рисунку 2.6 зображена архітектура з хешуванням для забезпечення ідемпотентності:

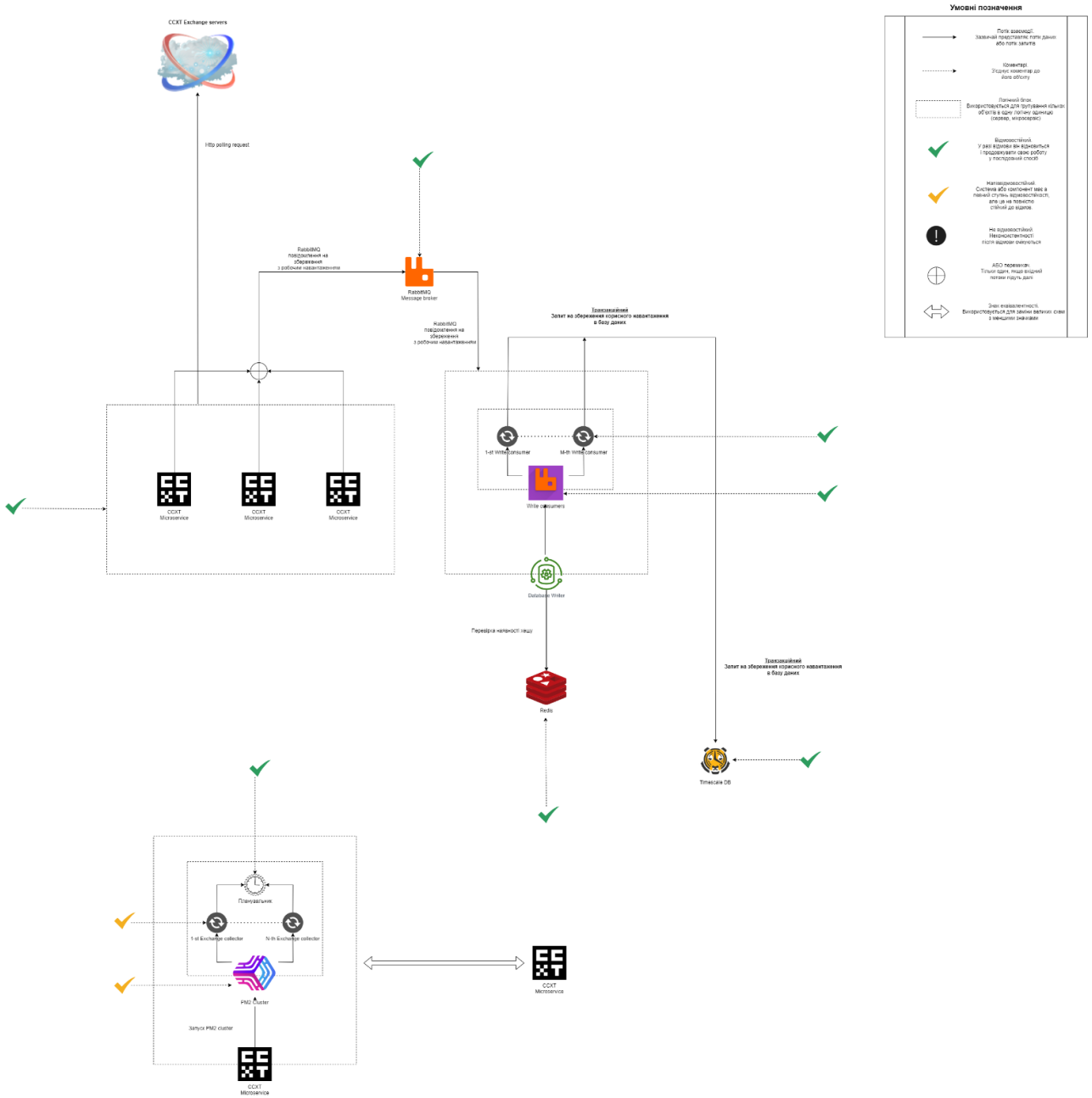

Рисунок 2.6 — Архітектура з хешуванням для забезпечення ідемпотентності

Як було зазначено при оцінці результатів та ефективності у підрозділі 2.2, використання компоненту, що керує синхронізацію – створює критичну залежність та вразливість системи у випадку втрати такого контролера. Якщо оркестратор колекторів перестане функціонувати – вся система перестане функціонувати. Така ситуація не є проблемою у випадку з RabbitMQ, оскільки цей сервіс підтримує реплікацію та має власні гарантії стійкості, на відміну від оркестратора.

Для вирішення такої проблеми, компонент оркестрування повинен мати власні механізми гарантування стійкості, реплікацію, та синхронізацію. Повинні існувати системи моніторингу стану оркестратора та оперативної зміни відповідальності. Для вирішення таких проблем може бути використана кластеризація оркестраторів із досягнення консенсусу завдяки протоколу, описаному у підрозділі 2.4.

Проте, як вже було розглянуто у розділі 2.4, побудова такої системи має свої обмеження, та не завжди є найкращим рішенням, особливо у випадку цінності ресурсу часу.

Саме тому наступним кроком – варто спробувати позбутись критичного окремого компоненту керування кластером, але все ще дотримуватись принципів ідемпотентності для отримання функціоналу, описаного у пункті 2.5.1.

На рисунку 2.6 – зображено архітектуру системи, що досягає стану ідемпотентності колекторів за допомогою алгоритмів хешування. Суть механізму ідемпотентності, у такому випадку, полягає у використанні спрощених та швидких односторонніх функцій перетворенні інформації фіксованої довжини, що буде, в результаті, використовуватись як токен ідемпотентності.

Відповідальність за формування токенів у такому випадку – повністю покладається на сервіси колекторів, та не потребує додаткового компоненту рівня контролю виконання операцій, що поєднує позитивні характеристики архітектури, розписаної у підрозділі 2.4 та пункті 2.5.1.

Роль Database Writer у такі системі залишається незмінною та все ще вимагає збереження токенів у базі даних, або сервісі Redis, для перевірки та підтвердження того, чи була вже записана інформація.

2.5.3 Імплементація ідемпотентності через часову самосинхронізацію

На рисунку 2.7 зображена архітектура з ідемпотентністю через часову самосинхронізацію:

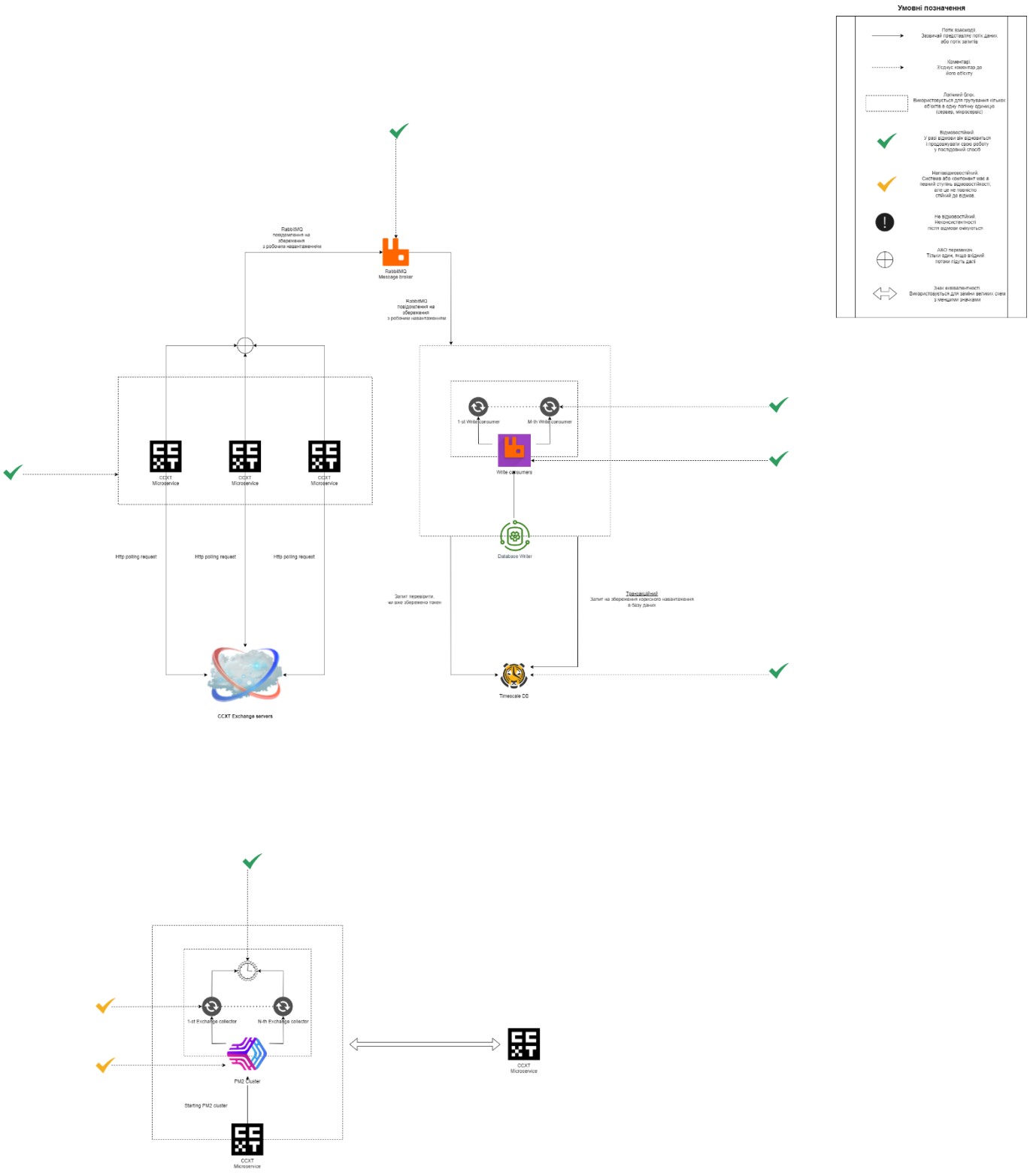

Рисунок 2.7 — Архітектура з ідемпотентністю через часову самосинхронізацію

Метод ідемпотентності на основі хешуванні, що описаний у пункті 2.5.2, є доволі ефективним способом побудови стійкої розподіленої системи, проте має ключовий недолік у вигляді залежності від волатильності результатів операцій.

У випадку зі збором інформації, що агрегується раз у заданому частотному інтервалі – такий метод чудово підходить, оскільки різниця у декілька мілісекунд не призведе до отримання різних результатів, а, як наслідок – до ти самих токенів ідемпотентності.

Проте, у випадку з високо-ліквідною інформацією, типу станів записів замовлень, або нещодавно виконаних обмінів, або стан значень у пулів ліквідності – різниця мілісекунд може призвести до різних результатів операції, а отже різних токенів ідемпотентності, що прямо нівелює концепт ідемпотентної системи за заданою, контрольованою частотою.

Зважаючи на такі системні вимоги, було розроблено архітектуру, що передбачає часову самосинхронізацію, зображену на рисунку 2.7, де токен ідемпотентності визначається межами часового інтервалу виконання операції.

Кожний колектор відповідальний за рівень контролю частоти надсилання запитів, синхронізація частот може бути виконана як в статичному так і динамічному режимі. Статичний режим передбачає надання інформації про розмір репліки та позицію конкретного колектора через змінні оточення, в той час як динамічний спосіб вимагає побудови алгоритму керування станами репліки.

Рішення динамічного встановлення репліки не має негативних наслідків описаних у підрозділі 2.4, оскільки відповідає за інший рівень абстракції. Протокол описаний в підрозділі 2.4 виконує механізм синхронізації для прийняття рішення про надсилання запиту в ході роботи інфраструктури збору інформації, проте протокол пошуку стану репліки – повинен лише дізнатись кількість учасників кластеру [62].

Застосування такої системи дозволяє повністю уникнути будь яких додаткових навантажень та потенційних критичних затримок, оскільки колектори не повинні ні вираховувати значення токену ідемпотентності, ні синхронізувати його через алгоритми досягнення консенсусу.

Роль Database Writer залишається в цілому незмінною, проте спосіб збереження інформації стає більш оптимізованим, оскільки інтервали збору є частиною схеми збереження корисної інформації та не потребують ні окремих таблиці, ні окремих сервісів для їх збереження.

Висновки за розділом 2

У даному розділі було побудовано ряд архітектур, що повинні гарантувати надійність та послідовність функціонування системи, забезпечуючи один з основних принципів тріади кібербезпеки, а саме «доступність».

Було сконструйовано систему обробки інформації, що надає прийнятний рівень надійності передачі та збереження інформації у межах периметру зони контролю системи. Описано фундаментальні процеси збору інформації, її передавання у внутрішній мережі, та процес її запису до постійного сховища. До того ж, оцінено значення побудови системи, компоненти якої мають слабкі зв’язки, та працюють на основі декількох рівнів абстракції: комунікації із зовнішнім джерелом, інтерфейсу комунікації, інтерфейсу контролю типу інформації, планування частот виконання операції та, врешті, синхронізації процесів.

Розроблено та описано архітектуру, що працює на основі механізмів самовиправлення, завдяки запису інформації у зворотному хронологічному порядку та проаналізовано недоліки такого способу в залежності від можливих сфер використання.

Розроблено та описано архітектору систем, що досягають стану синхронізації та консенсусу щодо виконання операції, за допомогою протоколів вибору лідера, зокрема протоколу RAFT та його модифікації для спеціальних потреба системи. Розглянуті причини та випадки, коли таке рішення не є найкращим через критичну залежність від часу прийняття рішень.

В результаті розроблено архітектуру, що поєднує покращений функціонал розглянутих систем, а також уникає недоліків, котрі були описані, за рахунок ідемпотентного механізму на основі часової самосинхронізації.

РОЗДІЛ 3
ПРОГРАМНА ІМПЛЕМЕНТАЦІЯ РОЗПОДІЛЕНОЇ РЕЗИЛЬЄНТНОЇ АРХІТЕКТУРИ НА ОСНОВІ ЧАСОВОЇ САМОСИНХРОНІЗАЦІЇ

3.1 Вибір стеку інфраструктурних технологій та його огляд

При проектуванні масштабованої та стійкої розподіленої системи, вибір сервісів, імплементованих третіми сторонами, що виконують фундаментальні поширені функції, такі як збереження або передача інформації у різних середовищах – має вирішальне значення та вплив на складність, модульність, та надійність всієї системи. Невдалий вибір інфраструктурних рішень на певному етапі розробки призведе до потреби пропрієтарної додаткової надбудови, або інтеграцію іншого сервісу, що в обох випадках є доволі важкими та повільними процесами.

Обрані сервіси повинні бути сумісними та підтримувати заплановані механізми логічного управління та гарантування надійності, що вимагає глибокого розуміння існуючих рішень та їх аналогів, представлених на ринку та у вигляді рішень з відкритим кодом.

Центральним сховищем даних, та сервісом, що дозволяє постійно масштабувати кількість збережених записів – обрано TimescaleDB, завдяки його можливостям до оптимізації операцій читання та запису для інформації часових рядків. Кожний із типів даних, що збирається інфраструктурою колекторів, відображає послідовні зміни ліквідності ринків у крипто та класичних середовищах, що є зразковим випадком використання, для якого був створений TimescaleDB.

Для підтримки сервісу кешування та операцій, що вимагають швидкого збереження та отримання інформації – обрано Redis, зважаючи на його розширені можливості щодо масштабування та підтримуваних функцій у порівнянні з конкурентами.

Реверс проксі – ключовий компонент, що дозволяє приховувати внутрішню мережеву інфраструктуру, надаючи зовнішнім системам лише прями доступ до API,

а також є компонентом, що найчастіше виконує функції балансування навантаження. Для виконання зазначених функцій – було прийнято рішення використати NGINX, що підтримує значний спектр функцій, включаючи валідацію запитів, їх модифікацію, балансування навантаження, аутентифікацію, та безліч інших функцій, що можуть бути підключені через систему плагінів.

У другому розділі даної дипломної роботи, в результати досліджень різних механізмів та способів зменшення ймовірності втрати інформації – було побудовано архітектуру, що покладається на час виконання, як спосіб синхронізації. Для надійного функціонування такої системи – необхідне застосування та налаштування NTP (Network Time Protocol) протоколу на цільових серверах, для забезпечення синхронізації системного часу на вузлах.

Врешті, для ізоляції розробленого рішення від інфраструктури розгортання системи – налаштовано процеси побудови образів Docker. Для керування контейнерами, побудованих на основі таких образів – використано Kubernetes у зв’язку з потребою у складних правилах афінності сервісів та їх комунікації між рядом віддалених вузлів.

3.1.1 Критерії вибору та огляд TimescaleDB

Перш за все варто зазначити, що TimescaleDB є публічним форком PostgreSQL з відкритим кодом, причиною створення якого була у потребі рішення, що спеціалізується на збереженні інформації, представленої у вигляді часової послідовності, тобто часових рядів. Попит на таку систему був та залишається особливо поширений серед систем зберігання, обробки та агрегації історичної фінансової інформації.

TimescaleDB дозволяє оптимізувати операції запису такої інформації завдяки поділу реляційних таблиць на шматки даних в залежності від часового інтервалу, що дозволяє значно спростити патерни доступу до інформації, завантаження індексів, запис та пошук місця розташування групи взаємопов’язаної та близької за часом інформації [53-54].

Масштабованість TimescaleDB досягається завдяки можливості розділення таблиць та часткового зберігання інформації на різних носіях, та навіть системах, що дозволяє горизонтально масштабувати систему [53-54].

Оптимізація досягається завдяки вже зазначенням можливостям гіпертаблиць щодо розподілу інформації часових рядів, а також, побудови індексів у вигляді бінарного дерева для швидкого пошуку необхідних рядків.

Для оптимізації використання постійної пам’яті, TimescaleDB застосовує збереження на основі стовпців , що дозволяє зменшити обсяг використаної пам’яті та покращує продуктивність введення/виведення, дозволяючи завантажувати більше даних у сховище одночасно [53-54].

Будучи реляційною базою даних, TimescaleDB повністю підтримує інтерфейс SQL, що дозволяє легкій перехід для інженерів, що працювали з іншими сховищами такого типу. До того ж, TimescaleDB розширює стандартний набір функцій PostgreSQL додатковими рутинами для роботи з часовими рядками.

Підсумовуючи, TimescaleDB, будучи розширенням PostgreSQL – сумісний із більшістю продуктів, що були розроблені для використання у екосистемі PostgreSQL. Будучи однією з найбільш поширених реляційних баз даних у світі, PostgreSQL має значну кількість розробок та плагінів, що спрощують управління сховищем та з’єднаннями до нього і є часто сумісними з TimescaleDB. До того ж, значна підтримка зі сторони спільноти дозволяє постійно покращувати існуючий функціонал TimescaleDB, оперативно виправляти вразливості та баги в системі.

3.1.2 Переваги та налаштування RabbitMQ

Брокер повідомлень RabbitMQ був обраний завдяки здатності до побудови складних схем маршрутизації, що дозволяють розділяти трафік джерел та споживачів за призначенням та тематикою.

RabbitMQ здатен підтримувати комунікацію одразу на декількох протоколах: AMQP, MQTT, та STOMP, що дозволяє його застосування у різних сферах, таких як

підтримка транзакцій, легкий обмін повідомленнями або програми на основі браузера відповідно [29-32].

Логічна побудова інфраструктури RabbitMQ складається з компонентів обмінників, черг, зв'язків між ними, споживачів та джерел інформації. Існує декілька базових типів обмінників [29-32]:

1. На основі прямої адресації, що дозволяє надсилати повідомлення напряму до черги за її адресою.

2. На основі розголошення «fanout» - дозволяє сповістити абсолютно всі черги, що мають зв'язки із вказаним обмінником.

3. На основі тематики «topics», що дозволяє маршрутизацію на основі відповідності заданої теми. Також існує можливість налаштування часткового співпадіння із тематикою. Правила співпадіння прописують при побудові зв'язків.

4. На основі заголовків, що надає найбільш гнучкий методо побудови маршрутизації, на основі значень заголовків, прикріплених до повідомлення. У такому випадку зв'язки з обмінником вказують на те, як інтерпретувати та які дії виконувати у випадку конкретних значень заголовків або їх відсутності.

Надійність передачі та обробки досягається за допомогою використання механізмів черг, схеми підтвердження повідомлень та стійких елементів логічної маршрутизації.

Механізм черг у RabbitMQ дозволяє зберегти наплив повідомлень у буфері, допомагаючи уникнути перенавантажень споживачів повідомлень або запитів. Черги, їх відстеження та позиція курсору - знаходяться під повним управлінням RabbitMQ, на відміну таких рішень як Kafka, де брокер відповідальний лише за запис, а клієнти за читання.

Схема підтвердження повідомлень може використовуватись у декількох режимах, котра може бути обрана клієнтом підключення. RabbitMQ завжди очікує підтвердження отримання повідомлення і не видалить інформацію з черги, допоки таке підтвердження не буде надіслане клієнтом.

Перший варіант підтвердження повідомлення передбачає надсилання «ACK» запиту до брокера одразу після повного отримання корисного навантаження. Такий

спосіб особливо корисний у випадку необхідності швидкої обробки потоку інформації, що не вимагає повторної обробки невдалих спроб споживача виконати операції щодо корисного навантаження.

Інший варіант підтвердження повідомлення передбачає надсилання «ACK» запиту до брокера тільки після повного та успішного опрацювання інформації отриманої від черги, де критерій успішності визначається правилами бізнес-логіки. Така схема дозволяє будувати механізми повторної обробки інформації. У випадку використання для сервісу Database Writer, така можливість дозволить уникнення неконсистентного стану систему через тимчасову відсутність зв’язку із базою даних, оскільки отримані повідомлення не будуть підтверджені, а отже залишаться в черзі та будуть повторно опрацьовані пізніше.

Стійкість елементів логічної маршрутизації брокера обміну повідомленнями можлива у декількох виглядах [29-32]:

1. При створенні логічного елементу, такого як обмінник, черга, або зв’язок між ними – можливо вказати його рівень стійкості. Такій рівень передбачає два стани: «стійкий» або «не стійкий», кожен з яких, відповідно вказує на необхідність збереження сутності після зникнення каналу зв’язку або перезавантаження системи.

2. При побудові черги, задання стану стійкості дозволить зберігати повідомлення не лише в структурі, що знаходиться в оперативній пам’яті, але й записувати її копію на місці постійного збереження інформації, що дозволяє відновлювати стан черги після перезавантажень системи, або інцидентів функціонування.

Масштабованість RabbitMQ досягається за рахунок можливості до реплікованої кластеризації та автоматичної синхронізації вузлів. Важливо зазначити, що масштабування навантаження в межах єдиної черги є слабким місцем RabbitMQ, з яким значно краще справляється Kafka, дозволяючи пропускну здатність на порядки вищу, ніж у RabbitMQ.

При створенні черги у кластері RabbitMQ – вона буде репліковано між учасники у асинхронному режимі, що дозволяє забезпечувати високу доступність у випадку несправності основного вузла, але не є рішенням, призначеним до масштабування чи

балансування навантаження, оскільки схема асинхронної реплікації дозволяє пришвидшено підтримувати стани реплік проте за рахунок втрати консистентності. При такій схемі реплікації, використання декількох вузлів одночасно призвело б до неконсистентного хаосу, в якому важко було б зрозуміти, котре повідомлення було вже оброблено [29-32].

Для масштабування RabbitMQ однак, рекомендовано створювати черги на різних вузлах, що дозволяє балансувати процесорне навантаження та використання пам’яті. У випадку необхідності підтримки великої кількості повідомлень, одна велика черга повинна бути логічно розділена на декілька менших. Обробка у таких випадках та групування логіки маршрутизації можливо досягти завдяки раніше зазначеними типами обмінників.

RabbitMQ, до того ж, має доволі зрілу спільноту та екосистеми. Існує безліч доступних плагінів, що розширюють його можливості, а також велика кількість імплементацій клієнтів на різних мова програмування, що значно спрощує процес інтеграції.

3.1.3 Роль Redis у кешуванні та управлінні станом

У заданій системі, Redis у першу чергу виконує ролі сервісу управління кешуванням, проте не обмежується такою функцією. Redis є значно більш розширеним сервісом у порівнянні з аналогом типу Memcached.

Вся інформація, котрою керує Redis зберігаються у оперативній пам’яті, що дозволяє виконувати швидкі операції пошуку, отримання та запису. Такий спосіб управління інформацією робить Redis ідеальним рішенням для застосування як сервіс синхронізації, кешування, управління сесіями, або навіть обміну повідомленнями.

На відміну від сервісів, що надають аналогічні функції, проте вміють працювати виключно як системи сховища типу «ключ-значення», де значенням може бути лише рядок, Redis здатний працювати з такими типами як: рядки, списки, набори, відсортовані набори із запитами діапазону, растрові зображення, гіперлоглоги та геопросторові індекси [63-65].

Рішення, такі як Memcached все ще дозволяють зберігати як значення всі зазначені типи даних, проте виключно у вигляді рядків. Завдання управління логікою співставлення типу та підтримки фундаментальних операцій, типу інкрементації для чисел – покладається на клієнти, що користуються таким сервісом.

Redis, однак, дозволяє не лише зберігати окремі типи даних, але й надає безліч функцій для їх маніпуляцій, що значно спрощують інтеграцію бізнес-логіки у сервіс збереження інформації, від якого вимагається в першу чергу швидкість операції.

Для стійкості, Redis підтримує конфігуруванні параметри збереження даних на диску, такі як знімки на певний момент часу (RDB) і файли лише для додавання (AOF). Такі функції забезпечують можливість відновлення інформації після неочікуваного помилкового завершення основного процесу або з'єднання [63-65].

Масштабування у Redis досягається за рахунок реплікації та кластеризації. Реплікація у Redis працює за асинхронною схемою, де головний вузол передає інформацію про свій стан до залежних вузлів як процес на фоні, що дозволяє досягти високої доступності, але не балансування навантаження, у зв'язку з такими ж причинами що й стосуються RabbitMQ [63-65].

Балансування навантаження у Redis, однак, можливо досягти за допомогою техніки кластеризації, що розділяє стан системи на частини, та розподіляє відповідальність між ними за існуючими вузлами у кластері.

Redis може також бути використаний як брокер, надаючи прості способи підписки та публікації подій, що можуть бути цілком достатніми для простих асинхронних систем, проте значно поступаються можливостям, гнучкості та функціям, що надаються RabbitMQ [63-65].

Надійність у Redis може бути досягнути за рахунок його здатності до виконання операцій як атомічних команд, що дозволяє створювати трансакційну логіку у системі швидкого маніпулювання та читання інформації.

Варто також зазначити, що Redis має неймовірно велику спільноту та багату екосистеми. Redis підтримує систему плагінів, що дозволяють розширювати його функціонал, додаючи можливості обробки додаткових типів даних, розширення існуючих функцій, або додавання цілком нових.

3.1.4 Використання Nginx як зворотного проксі

Nginx було обрано як рішення, що виконує функції реверс проксі, для підтримки як налаштувань безпеки, так і масштабованості. Різниця між класичними проксі-системами та реверс-проксі полягає у способі взаємодії та місці їх застосування.

Класичні проксі системи використовуються клієнтами для повного перенаправлення трафіку, що дозволяє приховати початкового джерела інформації та широко застосовується для анонімної взаємодії у глобальній мережі. В свою чергу, реверс проксі призначення для маскування системи яка є споживачем трафіку, а не його джерелом, що відповідає серверній частині комунікації у клієнт-серверній архітектури.

Основними перевагами використання реверс проксі є здатність до ізоляції комунікації між клієнтом та сервером, що дозволяє підмінити сервер у будь який момент такої комунікації. Такий механізм дозволяє досягати високої доступності, оскільки реверс-проксі часто застосовується для балансування навантажень та перенаправлення трафіку.

Nginx працює на основі асинхронних обробок запитів, та має внутрішню систему управління подіями. На відміну від інших рішень, що надають функції реверс проксі, такі як «Apache HTTP Server», котрі створюють новий потік або процес для обробки вхідного запиту, Nginx працює на основі «Event Loop», зберігаючи контекст виклику та лише очікуючи на результати відповіді від внутрішніх сервісів [66-67].

Такий спосіб обробки запитів не є найкращим рішенням при високому процесорному навантаженні на кінцевих точках, оскільки єдиний запит може заблокувати можливість інших клієнтів до підключення. Проте, така система ідеально підходить для проксі-сервісів та стандартних запитів на отримання інформації з бази даних, або інших сервісів збереження інформації, котрі виконують основну масу процесорного навантаження.

Nginx дозволяє масштабувати системи, що складаються із серверів, котрі не містять стану, надаючи можливості до балансування навантаження запитів

протоколів транспортного та прикладного рівнів, а також підтримує складні схеми переадресації, та пулів доступних вузлів, надаючи можливість переписати, або частково замінити вхідні запити за вказаними правилами.

Масштабування самого сервісу Nginx є простим та припускає як горизонтальні так і вертикальні механізми. Nginx не потребує синхронізації між репліками, оскільки не має стану, що дозволяє значно спростити горизонтальне масштабування.

Nginx також може бути застосований як інструмент для впровадження стандартизованих мір безпеки, таких як застосування механізмів термінації SSL (Secure Sockets Layer) / TLS (Transport Layer Security), встановлення безпечних заголовків, або навіть налаштування базової системи аутентифікації, та правил доступу до кінцевих точок [66-67].

Nginx має зрілу екосистему та широку спільноту, що дозволяє швидко та ефективно виявляти потенційні баги у системі та створювати все більше доповнень. Підтримка модульної системи завантажень розширень – дозволяє інтегрувати плагіни, котрі можуть розширювати функціонал щодо контролю вхідних запитів та їх маніпуляцій.

3.1.5 Налаштування Docker, Docker Compose, та Kubernetes

Docker є де-факто стандартом у розгортанні, масштабуванні та підтримці сучасних вебдодатків та розподілених систем, дозволяючи абстрагувати інфраструктурні особливості платформи, на які запущено цільову систему.

Контейнеризація дозволяє упакувати вихідний код програми у вигляду одиниці управління, котра зветься контейнером. Контейнери значно простіші для створення та запуску, на відміну від віртуальних машин, котрі вимагають запуску окремої операційної системи.

Docker побудований на основі клієнт-серверної архітектури, де dockerd демон постійно очікує запитів на власному REST API, CLI, або SDK. Технологічну основу цієї технології складають можливості Linux-систем, а саме chroot, namespaces, та cgroups для ізоляції різних процесів та їх ресурсів [49-50].

Контейнери Docker створюються на основі образів, структура яких описуються у Dockerfile. Кожний запис у Dockerfile – створю окремий рівень запису, що працює на основі OverlayFS [49-50].

Dockerfile для базового сервісу розроблюваної розподіленої системи має наступний вигляд, зображений на рисунку 3.1:

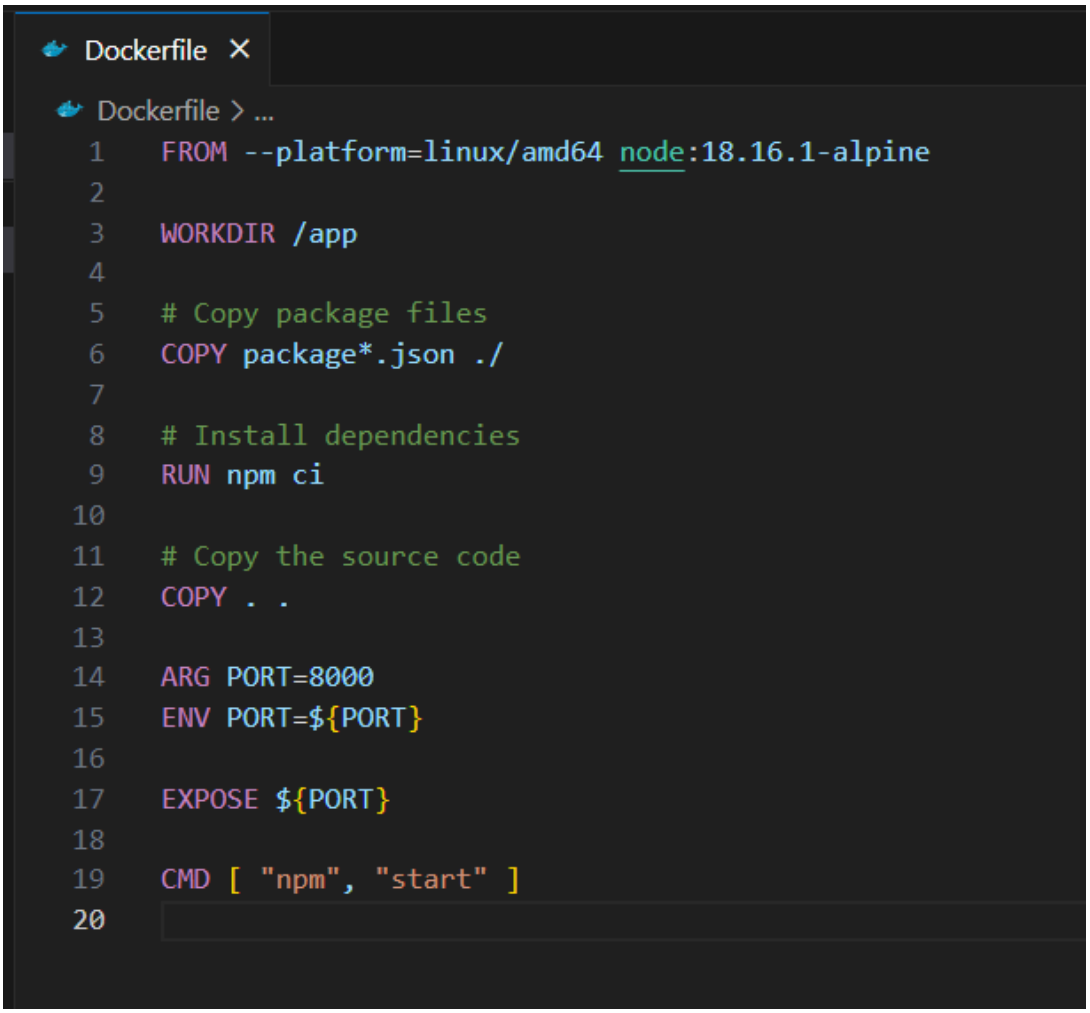

Рисунок 3.1 — Dockerfile сервісів на основі Node.js

Кожний рядок у коді, зображеному на рисунку 3.1 – окремій рівень OverlayFS, що дозволяє оптимізувати процес створення образу у випадку, якщо тільки останній рівень було змінено, в той час як інші залишились такі самі.

Зважаючи на вище зазначені причини, спочатку відбувається копіювання файлу «package*.json» та встановлення залежностей за допомогою NPM (Node Package Manager) і лише після цього – копіюється основний код програми до образу. В такі

побудові, при мінімальних змінах кодової бази – не буде потреби у повному перезавантаженні залежностей.

Створення контейнерів на основі образів наслідує такий самий принцип «Copy on Write», що робить їх спрощеними та зменшує використання місця на диску. Додаткове місце буде виділено лише у тому випадку, коли контейнеру необхідно змінити стан файлової системи, що ним контролюється.

Docker також надає можливості щодо збереження стану контейнерів за допомогою функціоналу «розділів». Розділи бувають декількох типів, проте їх суть завжди залишається єдиною – збереження інформації у випадку знищення контейнера.

Анонімний розділ повністю контролюється системою Docker, доступ до такого розділу не є керованим, а шлях його зберігання та наповнення повністю підконтрольне лише системі Docker.

Іменований розділ дозволяє об’єднання використання файлової системи між декількома контейнерами, що може бути особливо корисним при горизонтальному масштабуванні сервісів, що мають певний стан, а також, при синхронізації реплікованих сервісів, або міграції інформації.

Розділ встановленого зв’язку дозволяє надати доступ контейнеру до порції основної файлової системи, що дозволяє пряме управління наповненням такого розділу зі стороно хост-системи.

Для автоматизованого управління декількома контейнерами, їх мережевою взаємодією, відстеженню стану, а також одночасному запуску, оновлення або зупинки – використовується інструмент оркестрації Docker Compose.

Docker Compose дозволяє описати спосіб взаємодії контейнерів в одному або декількох фалах формату yaml, що дозволяє управління контейнерами без використання імперативного підходу, що вимагає послідовного виконання команд.

Проте, Docker Compose придатний до застосування лише на єдиній фізичній машині, для оркестрації та управління контейнерами, що створені та знаходяться на різних фізичних системах – потрібні інструменти такі як Docker Swarm, або Kubernetes.

Docker Swarm наслідує стандарт Docker Compose і є повністю сумісним з ним, що дозволяє значно простішу інтеграцію в системи, що раніше були побудовані на єдиному хості з використанням Docker Compose.

Kubernetes, в свою чергу, не є сумісним із Docker Compose, є набагато більш складнішою системою в управлінні, проте надає значно ширші функції щодо політик створення контейнерів, їх взаємодії та прав доступу до них.

Kubernetes є найкращим рішенням для управління великими кластерами контейнерів, завдяки його гнучкості, можливостям адміністрації, та надійності. Docker Swarm краще підходить для середніх за розміром кластерів, що не вимагають синхронізації великої кількості вузлів.

3.2 Вибір стеку програмних технологій та його огляд

Окрім вибору інфраструктурних рішень, що будуть допомагати досягати масштабованого збереження та передачі інформації, вибір програмної основи розроблюваних рішень – має також критичне значення щодо можливостей системи, її стійкості, розширюваності, підтримки та масштабованості.

Мовою програмування, що буде використана для розробки сервісів, обрано JavaScript, завдяки можливості його застосування як для розробки серверної частини, так і клієнтської сторони. До того ж, подійно-орієнтована модель програмування буде найкращим та найбільш ефективним способом для використання у контексті заданої системи, як буде показано далі.

Для побудови веб серверу, та системи REST API, було обрано Express.js у зв’язку з його швидкістю та простоті інтеграції та застосування. Express.js має найбільшу екосистему розширень серед бібліотек HTTP протоколу у середовищі Node.js.

Для взаємодії з базою даних TimescaleDB – обов’язкове застосування ORM (Object-Relational Mapper) системи, що по-перше, абстрагує управління з’єднанням з базою даних, а по-друге – несе відповідальність за перевірку та очищення параметрів переданих від клієнта для уникнення ін’єкцій SQL.

AMQP Connection Manager в даному проекті застосовано як бібліотеку, що спрощує інтерфейс управління сервісом RabbitMQ за допомогою протоколу AMQP. Окрім спрощеного інтерфейсу комунікації, AMQP Connection Manager також відповідальний за відновлення підключень до сервісу RabbitMQ у разі збоїв, що підвищує надійність системи.

Для інтеграції та підтримки з'єднання із сервісом Redis – використано клієнт «Redis io client». Зазначений клієнт надає ряд методів що напряму відображають нативні функції Redis, а також відповідальний за підтримку аутентифікованого з'єднання та його неперервності.

3.2.1 Застосування Node.js у мікросервісній архітектурі

Node.js – це платформа виконання коду JavaScript, що застосовує двигун V8 для інтерпретації програми. Створення такої платформи стало історичним рішенням, що дозволило розробникам використовувати ту саму мову програмування та модель, що орієнтована на події, не тільки на стороні клієнта, але й на стороні сервера.

Асинхронність Node.js досягається завдяки імплементації механізму «Event Loop», котрий обробляє виклики до I/O у неблокуючому режимі, що є значно швидшим та більш оптимальним у використанні для більшості веб-додатків та сервісів, котрі не виконують складних обчислень результатів [68-69].

Цільова розподілена система побудована на принципах, що не вимагають складних обчислень та є в своїй основі орієнтованою на події: отримання інформації, її відправка, її перевірка та збереження. Заважаючи на це, асинхронна модель обробки Node.js є чудовим рішенням, що дозволяє уникнути складнощів синхронізації процесів та потоків, а також є значно більш ефективною за рахунок відсутності потреби у побудові контекстів нових процесів для кожного окремого запиту, а також уникнення надмірно частих змін контекстів виконання.

Основна програма, побудована на основі Node.js – запускається в єдиному процесі, проте не обмежена використанням лише єдиного процесу або потоку. Екосистема Node.js надає можливості створювати, синхронізувати, та керувати

новими процесами, або потоками, ще не є найбільш поширеним способом його використання, але може бути корисним у випадку вимог до складних обчислень, котрі можна виконувати паралельно [68-69].

Основним інструментом управління залежностями Node.js є NPM, що надає CLI утиліти та публічний реєстр з тисячами розроблених бібліотек для різноманітних потреб: від абстракцій HTTP серверу, до повноцінних веб-фреймворків, що значно спрощують побудову складних та масштабованих систем, впроваджуючи патерни найкращих практик, таких як ін'єкція залежностей та доменно-орієнтована модель побудови системи.

Нативні модулі Node.js надають інтерфейси до файлових систем, найбільш поширених протоколів комунікації, а також широкого набору криптографічних функцій. До того ж, однією з переваг Node.js є його крос-платформеність, що дозволяє використовувати один і той самий код на різних фізичних машинах та операційних системах [68-69].

Екосистема Node.js постійно розвивається, надаючи все більше можливостей до побудови складних систем без необхідності повторного вирішення загальновідомих складнощів, таких як розробка драйверів до сервісів. Спільнота Node.js є однією з найбільших технічних спільнот світу, завдяки якій постійно підтримуються та вдосконалюються нові рішення, побудовані на платформі Node.js.

3.2.2 Особливості та налаштування Express.js

Express.js є простою бібліотекою, що абстрагує управління HTTP сервером, та з'єднаннями до нього. Express.js відрізняється від інших спрощеною системою побудови шляхів кінцевих точок, швидкістю, а також широкою екосистемою розширень, що значно спрощують як отримання інформації, так її валідацію та аутентифікацію.

Побудова API на основі Express.js включає декілька ключових логічних компонентів [70-71]:

– роутер виконує функцію групування окремих кінцевих точок, або інших роутерів нижчого рівня, що значно спрощую управління масштабуванням кінцевих точок, оскільки дозволяє їх об’єднувати у деревоподібну структуру;

– посередник виконує функції перехоплення повідомлення між клієнтом та кінцевою точкою, та надає можливістю контролю ходом виконання запиту. Посередники є основним способом розширення можливостей Express.js, оскільки можуть бути використані як для всіх запитів, так і для окремих, надаючи сервіси парсингу корисного навантаження, валідації, або аутентифікації;

– кінцеві точки є частинами, що відповідальні за виконання заданої бізнес логіки серверної частини. У контексті розробки розподіленої системи, описаної у цій роботі, кінцеві точки будуть відповідальні за встановлення комунікації з базою даних та отримання інформації, що в ній збережена.

Express.js має велику спільноту та обширну екосистему, надаючи безліч додаткових бібліотек, що імплементують готові функції аутентифікації, валідації, або парсингу вхідної інформації.

Проте важливо зауважити, що Express.js не є фреймворком, не надає формату та правил побудови проекту, що вимагає значного розуміння принципів чистої архітектури від інженера. Express.js є простою бібліотекою, але разом із пришвидшенням побудови систем – простіше зробити архітектурних помилок, що можуть привести до ускладнень при розробці та масштабуванні у майбутньому.

3.2.3 Використання Sequelize як ORM-інструмента

При побудові систем, що приймають параметри від користувача та на їх основі будують та надсилають запити до бази даних, ORM є необхідним інструментом, котрий спрощує інтерфейс отримання інформації. Керує пулом з’єднань, а також виконує валідацію та очищення вхідних параметрів, що захищає від SQL ін’єкцій.

Sequelize є одним з найбільш популярних та найстарших ORM рішень у екосистемі Node.js та абстрагує управління інформацією, що зберігається за допомогою моделей даних. Моделі Sequelize імплементують паттерн «Active Record»

для взаємодії з доменною моделлю, що дозволяє визначати логіку взаємодії з інформацію в смій моделі збереження.

Моделі Sequelize можуть бути згенеровані автоматизованим шляхом, надаючи лише схему до методу «define» основної моделі, або представлені як класи, котрі містять специфікацію сховища, таку як схема таблиці, її назва, та індекси. Моделі також зберігають логіку найвищого рівня за принципами чистої архітектури, імплементуючи методи доступу до інформації та її обробки [72-73].

Додаткові функції Sequelize, такі як «relation mixins», «scopes», та «hooks» - дозволяють значно спростити управління життєвим циклом обробки сутності, котра репрезентує рядок у даних у базі. Relation Mixins надають можливість автоматизованого додавання методів, котрі відповідальні за отримання або встановлення зв’язків між різними таблицями. Scopes надають можливість спростити складні запити, розділяючи умови на окремі частини, котрі можуть як фільтрувати інформацію, так і обмежувати кількість полів, що повертається. Hooks, в свою чергу, дозволяють перехоплювати інформацію на різних етапах її життєвого циклу та вносити корективи до її збереження [72-73].

Sequelize надає інтерфейс до створення та управління міграціями, котрі дозволяють постійне оновлення та підтримку системи, моделі даних якої постійно змінюються або доповнюються. Міграції описуються у послідовних файлах, кожний з яких повинен експортувати два основних методи «up» та «down», котрі надають можливість управління версіями стану схеми таблиць [72-73].

Sequelize відповідальний за встановлення аутентифікованого з’єднання з базою даних та управління пулами з’єднань. Детальна конфігурація дозволяє налаштовувати час життя з’єднання, що може бути особливо корисним, оскільки бази даних на основі PostgreSQL постійно кешують додаткову інформацію у контексті процесу з’єднання, що призводить до їх постійного збільшення споживання оперативної пам’яті.

Метод основної моделі «sanitize» дозволяє виявляти та коментувати SQL код, що передається як рядок, та є основним способом захисту від ін’єкцій. Sequelize також підтримує функції перевірки вхідних параметрів при запуску сконструйованих

запитів, котрі значно зменшую ймовірність виконання атаки «SQL Injection», що досі є однією із найбільш поширених способі зламу вебдодатків.

Велика спільнота та популярність даного рішення дозволяють постійну підтримку, виправлення критичних багів, вразливостей, та постійне покращення. Нині у розробці 7 версія Sequelize, котра надасть покращений інтерфейс для використання разом з TypeScript, що стає все більш популярний при побудові як клієнтських так і серверних рішень.

3.2.4 Інтеграція AMQP Connection Manager для комунікації з RabbitMQ

AMQP Connection Manager – клієнт, що абстрагує канал комунікації на основі раніше розглянутого протоколу AMQP, та дозволяє здійснювати управління обмінниками, зв’язками та чергами RabbitMQ.

Дана бібліотека спрощує модель комунікації, підтримуючи асинхронний спосіб взаємодії з RabbitMQ, та надаючи прості інтерфейси для під’єднання та управління споживачами інформації.

AMQP Connection Manager дозволяє керувати завантаженістю споживачів за допомогою механізму встановлення попереднього отримання повідомлень. Зазначений механізм дозволяє явно вказати ліміт одночасно оброблюваних повідомлень на стороні клієнта, що є особливо необхідним у випадках, коли повідомлення містять інформацію для оновлення стану системи. У такому випадку одночасна обробка декількох запитів на оновлення може призвести до неконсистентності та стану перегонів [74].

Клієнт також створює рівень абстракції управління з’єднанням та його відновлення, що значно підвищує надійність цільової системи, котра побудована та повністю опирається на можливість обміну повідомленнями через RabbitMQ. У випадку виникнення неочікуваного розриву з’єднання, команди, що повинні були бути надіслані до брокера повідомлень – будуть буферизовані, та після повторного підключення – надіслані та виконані [74].

3.2.5 Конфігурація та інтеграція Redis io client

Для стійкої взаємодії із сервісом Redis, в системі використано клієнт, що надається бібліотекою «Redis io client», котра, в свою чергу, абстрагує деталі встановлення аутентифікованого з'єднання, його підтримку, та надсилання послідовних команд.

Дана бібліотека здатна інтегруватись із «Redis Sentinel», для ефективної взаємодії та використання розподіленої реплікованої системи, що складається з декількох екземплярів сервісу Redis. Така інтеграція значно підвищує надійність системи, оскільки дозволяє автоматично перепідключити з'єднання до асинхронної репліки, у випадку критичної несправності основного вузла.

«Redis io» підтримує декілька схем оптимізації, такі як «pipelining», що дозволяє відправити декілька команд одночасно в єдиному пакеті, що значно зменшує використання пропускної здатності мережі. До того ж, бібліотека підтримує функцію «Lua Scripting», котра дозволяє виконувати складні скриптові операції напряму на одному із серверів Redis, що може значною мірою знизити навантаженні із основного сервера Node.js [74].

Із зростаючою популярністю TypeScript, дана бібліотека також надає прописану типізацію, що значно спрощує застосування функцій та відслідковування потоку інформації на етапі розробки. Більшість функцій мають схожий семантичний опис із прямою імплементацією на стороні Redis, та мають детальний опис типів даних, що очікуються як параметри, а також результатів виконання.

Для підвищення стійкості та надійності системи, «Redis io» також імплементує механізми черг запитів на стороні клієнта. У випадку неочікуваного розриву з'єднання, клієнт спробує його відновити, та у разі успіху виконає всі запити, що були збережені у черзі.

Дана бібліотека повністю реплікує інтерфейс взаємодії Redis, включно із можливістю застосуванні різних типів даних, транзакційних режимів, а також схеми використання Redis як спрощеного брокера повідомлень.

3.3 Імплементація сервісу збору та обробки інформації

На рисунку 3.2 зображена архітектура сервісу збору та обробки інформації:

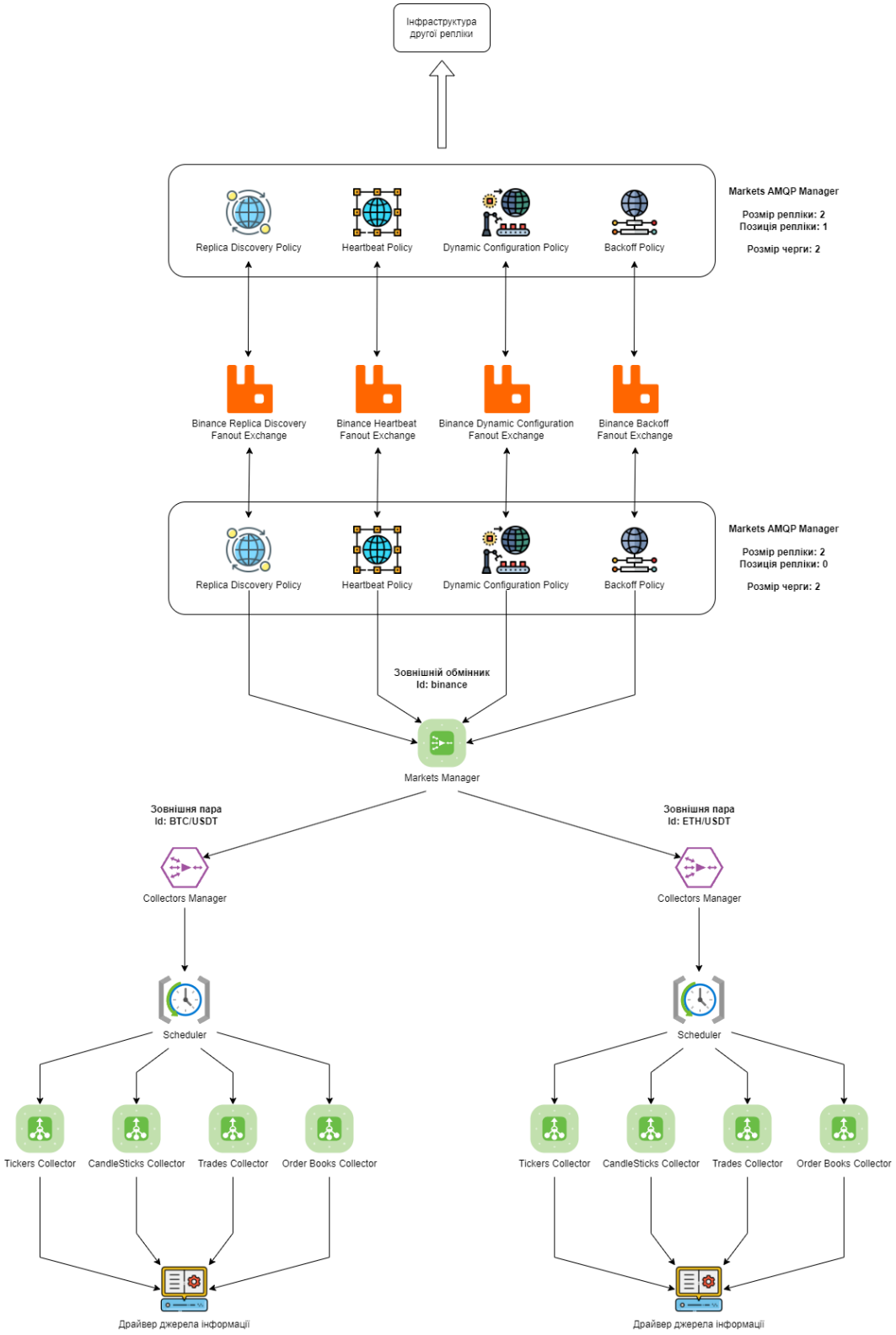

Рисунок 3.2 — Архітектура сервісу збору та обробки інформації

Рисунок 3.2 зображує архітектури сервісу збору інформації, що є частиною базової інфраструктури збору інформації, описаної у підрозділі 2.1. Завданням даного сервісу є встановлення каналу комунікації з віддаленими серверами та передача інформації до сервісу «Database Writer», за допомогою сервісу RabbitMQ.

Для виконання поставленого завдання, з урахуванням складнощів, що були описані у розділі 2, архітектуру сервісу розділено на декілька ключових рівнів абстракції: уніфікований драйвер взаємодії із зовнішніми сервісами, колектори, що відповідальні за конкретний тип інформації, його форматування та адресацію, рівень управління частотою надсилання запитів, рівень управління парами обміну, рівень управління обмінником, та рівень комунікації.

Основним завданням уніфікованого драйверу взаємодії із зовнішніми сервісами є абстрагування імплементації API на стороні різноманітних джерел інформації. Драйвер відповідальний за встановлення сесії обміну інформації, якщо така необхідна, та її аутентифікацію, відповідно до специфікації обмінника. Драйвер повинен підтримувати всі можливі варіанти аутентифікації, що можуть застосовуватись на стороні обмінників, такі як на основі паролю, токену доступу, або приватного ключа. Драйвер також відповідальний за перетворення перехоплених помилок в уніфікований стандарт на стороні контрольованої системи.

Рівень колекторів представлений набором класів, кожен з яких відповідальний за отримання, форматування та відправку до сервісу збереження відповідного типу даних. Базовий клас має два основних методи: «fetch» та «save», де «fetch» реалізує логіку взаємодії із рівнем драйверів, перехоплює помилки та повертає корисну інформацію, в той час як «save», отримавши інформацію від «fetch» - виконає її форматування, сконструює повідомлення для RabbitMQ, додаючи токен ідемпотентності, розглянутий та описаний у підрозділі 2.5, після чого надішле таке повідомлення до черги обробки.

Компонент «Scheduler» відповідальний за розрахунки інтервалів запуску контейнерів із застосуванням як нативних можливостей «Node.js», таких як доступні в оточені рутини «intervalStart» та «setTimeout», так і за допомогою застосування «cron» планувальника для часової синхронізації. Операційна послідовність

виконання планування є критичною у системі, оскільки, як було обумовлено у розділі 2, невиконання вимог політик безпеки зовнішніх сервісів щодо частоти надсилання запитів – може привести до повного або тимчасового блокування джерела таких запитів.

Планувальник частот відповідальний за запуск набору колекторів у задані моменти часу, проте не є відповідальним за ініціалізацію стану сервісу збору інформації цифрової репрезентації заданої пари товарів. Завданням рівня управління парами обміну є ініціалізація планувальника та колекторів, а також надання інтерфейсу для припинення операції, або повної зупинки збору інформації, що є абсолютно необхідним у випадку потреби оновлення стану, або тимчасового обмеження зі сторони політик безпеки сервісу, що надає інформацію.

Побудова рівня комунікації одразу над рівнем управління парами обміну призведе до надмірного та неоптимального використання трафіку мережі, оскільки кількість різних представлених у системі пар може сягати сотень тисяч. Щоб вирішити дану проблему, існує рівень управління обмінниками, котрий об’єднує, синхронізує та надає API до управління групами пар одночасно. Такий спосіб комунікації значно зменшує навантаження з мережі, та значно пришвидшує комунікацію, оскільки вона відбувається прямими імперативними викликами до рутин, а не через посередників, або очікування повідомлень.

Останнім рівнем абстракції є рівень комунікацій, що відповідальний за встановлення каналів з’єднань у реплікованому середовищі та обміну інформації у ньому. Такий рівень відповідальний за ініціалізацію менеджера рівня обмінника, а також ініціалізацію інтегрованих протоколів комунікації.

Зазначені протоколи повинні виконувати базові функції початкової синхронізації стану, що є основним завданням «Replica State Discovery Protocol», а також динамічного оновлення у випадку повідомлень про порушення політик безпеки віддалених сервісів (Backoff Policy), динамічного оновлення стану колекторів (Dynamic Configuration Policy), та протокол для отримання метрик репрезентації стану системи (Heartbeat Policy) [62].

3.4 Розробка сервісу запису інформації у базі даних

На рисунку 3.3 зображена архітектура сервісу запису інформації у базі даних:

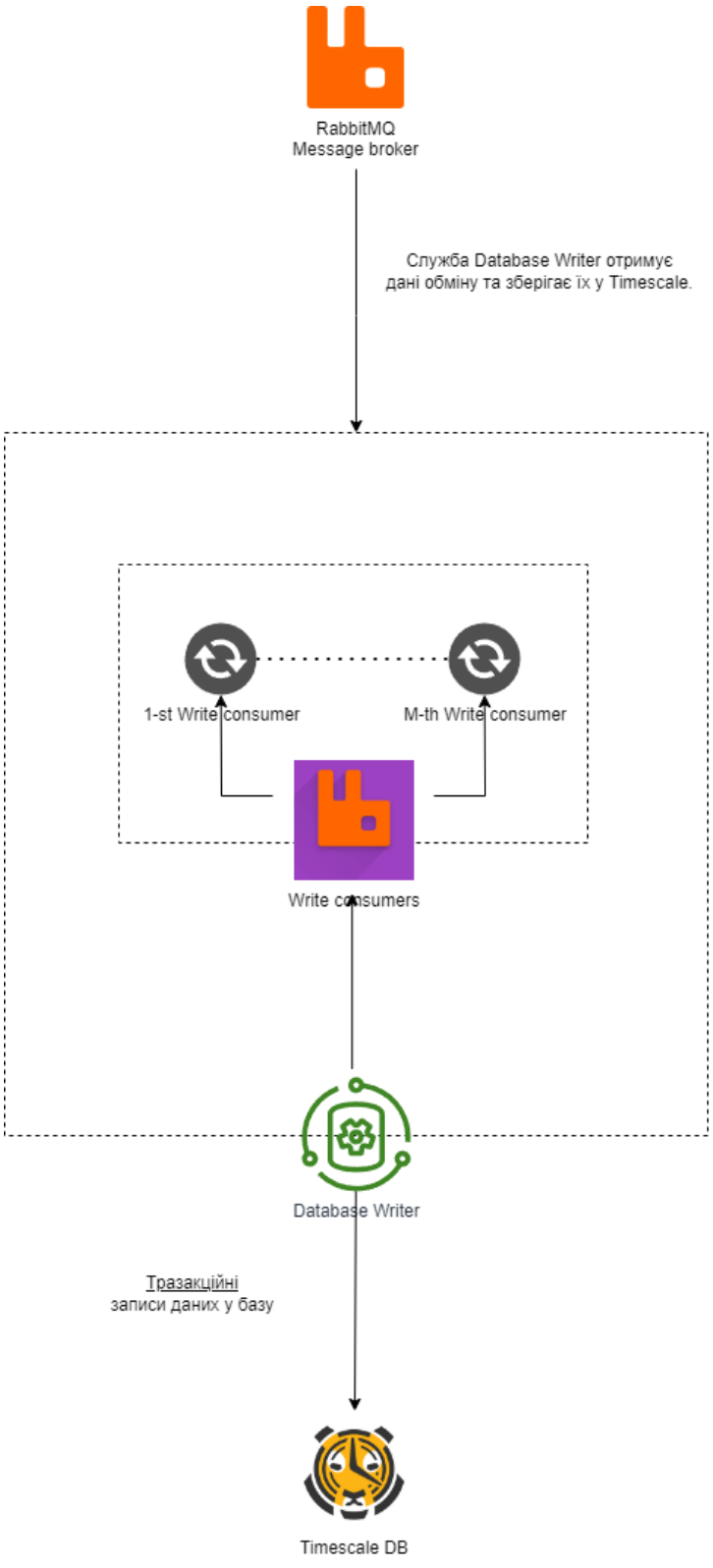

Рисунок 3.3 — Архітектура сервісу запису інформації у базі даних

Database Writer Service має просту структуру та є споживачем інформації, що надсилається через RabbitMQ. Основною задачею цього сервісу є верифікація токенів ідемпотентності, та запис інформації у базу даних TimescaleDB.

Архітектура сервісу складається з набору класів, кожний з яких відповідальний за споживання відповідного типу даних, та є кінцевим призначенням, що адресується відповідним колектором, описаним у підрозділі 3.3.

Спільна логіка роботи кожного сервісу передбачає перевірку наявності токену ідемпотентності у базі даних, що представлений як об’єкт із трьома полями «intervalStart», «intervalEnd», «marketId». У підрозділі 2.5 описано механізм ідемпотентності та важливість його застосування для досягнення консенсусу із мінімальним додатковим навантаженням та використанням часу у системі.

Масштабованість даного сервісу досягається за рахунок способу обробки запитів у режимі споживача. У такі конструкції, RabbitMQ виконує роль управління станом черги та гарантує, що в системі неможливий випадок одночасної обробки одного і того самого повідомлення двома споживачами одночасно. Завдяки цьому, сервіс підтримує як горизонтальне так і вертикальне масштабування, що дозволяє збільшувати кількість інтегрованих пар та типів даних без додаткових потреб у зміні коду Database Writer.

Database Writer також відповідальний за налаштування політики повторної обробки повідомлень у випадку тимчасової несправності бази даних, або мережевих проблем. Система повторної обробки інформації працює на основі механізму «dead letter exchange», що надається сервісом RabbitMQ. Для заданої черги обробки повідомлень, що містять інформацію про конкретний тип даних, зібраний з API обмінника – створюється відповідна йому черга «мертвих» повідомлень. Усі непідтверджені повідомлення у такому випадку будуть надіслані до такої черги, котра, в свою чергу, має обмеження на час зберігання повідомлень, та, як власну чергу «мертвих» повідомлень – призначає основну чергу запитів на збереження інформації. Таким чином після закінчення терміну зберігання повідомлення у черзі «мертвих» повідомлень – воно буде перенаправлене назад до основної черги, та врешті повторно опрацьовано сервісом Database Writer.

3.5 Створення основного HTTP серверу

На рисунку 3.4 зображена архітектура основного HTTP серверу:

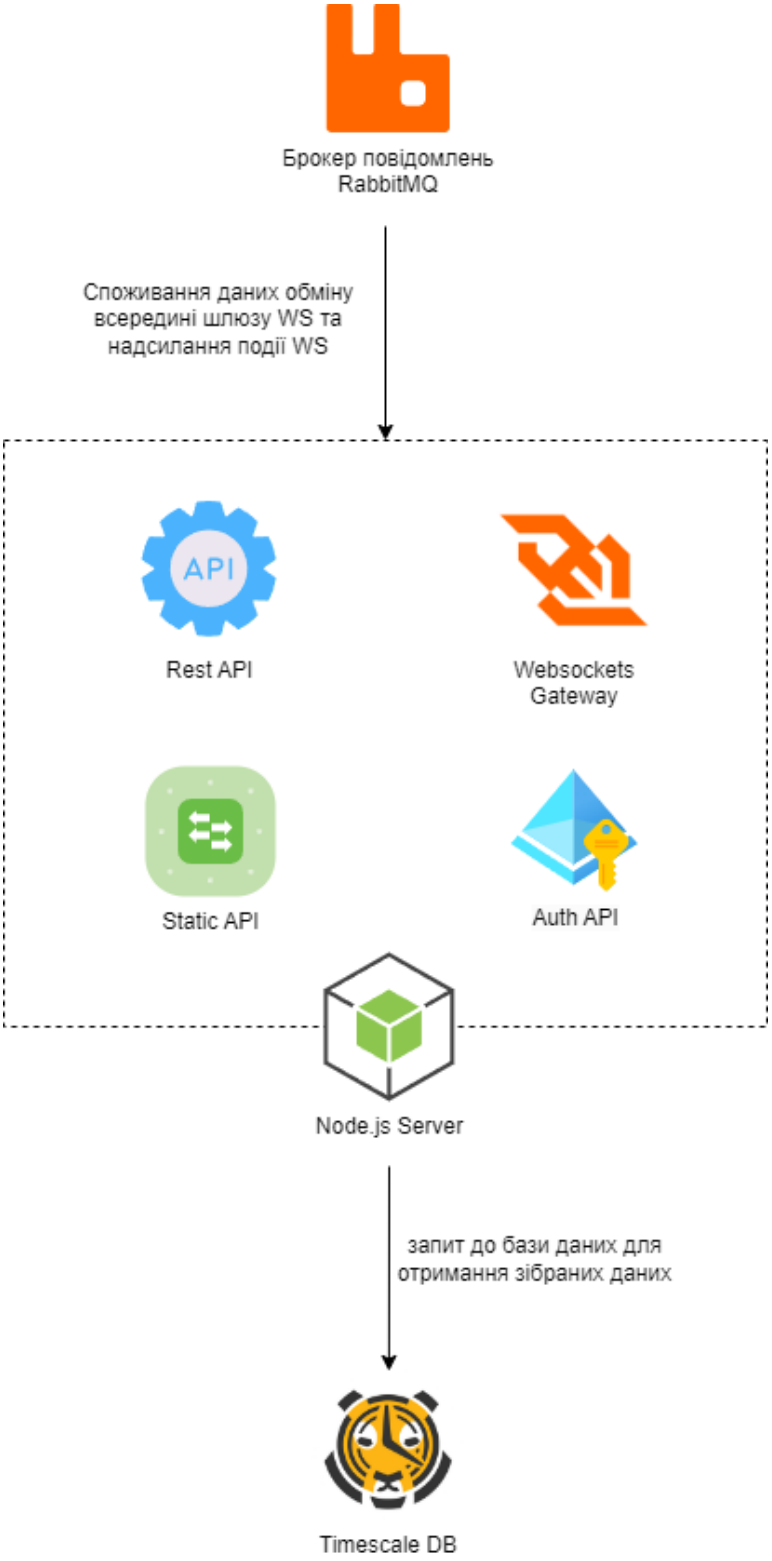

Рисунок 3.4 — Архітектура основного HTTP серверу

Для надання доступу користувачам до збереженої інформації – розроблено компонент основного Node.js HTTP серверу, що надає API читання та запису інформації. Сервер побудований із застосуванням раніше зазначених у підрозділі 3.2 бібліотек Express.js для спрощення створення та управління HTTP сервером, Sequelize, для взаємодії із базою даних, та Redis io, для взаємодії із сервісом кешування.

Створений сервер складається з чотирьох основних компонентів: «Rest API», «WebSocket Gateway», «Static API», та «Auth API». Разом, ці компоненти забезпечують аутентифікований доступ до інформації як в асинхронному, так і режимі реального часу, та надають можливість аналітикам оперативно формувати висновки, на основі отриманої інформації з бази даних, або подій вебсокетів.

Компонент «Rest API» імплементує та надає колекцію кінцевих точок, що безпосередньо відносяться до логіки отримання зібраної та агрегованої інформації із бази даних. Кожна кінцева точка має власну специфікацію права доступу, а також схему валідації вхідних параметрів та систему приведення інформації, що міститься у відповіді сервера до стандартизованого формату.

«WebSocket Gateway» надає можливість доступу до інформації, що збирається у режимі реального часу. Як тільки інформація про конкретний ти даних буде записаний у базу даних сервісом Database Writer, він також надішле повідомлення через брокер RabbitMQ до «WebSocket Gateway», після чого воно буде автоматично розповсюджене до користувачів, що підписані на оновлення заданого типу інформації та пари, представленої в обміннику.

Компонент «Auth API» відповідальний за управління аутентифікацією та правами доступу користувачів. Система аутентифікації заснована на кукі-сесіях, та вимагає також додаткових механізмів захисту від атак типу CSRF. Для запобіганням таких атак імплементовано систему токена синхронізації, що прикріплений до сесії, та надсилається користувачу разом із повідомленням про успішну авторизацію у системі. Для кожного наступного запиту користувач повинен надіслати токен синхронізації, аби підтвердити свою дію. У випадку відсутності токена, або його не співпадіння із тим, що був збережений у сесії, запит вважається нелегітимним.

На рисунку 3.5 зображено принципи функціонування паттерна захисту від CSRF на основі токена синхронізації:

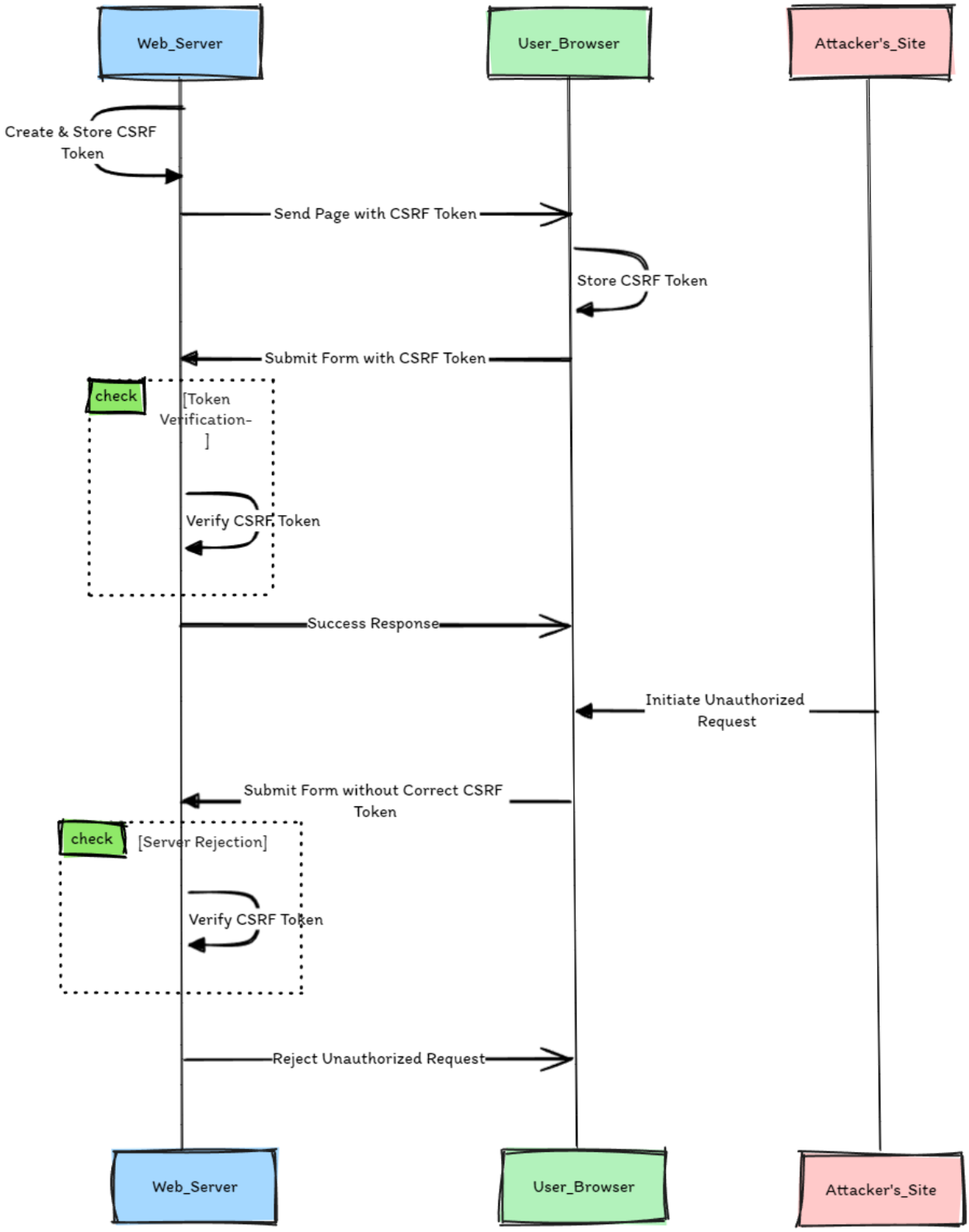

Рисунок 3.5 — Паттерн токену синхронізації

Компонент «Static API» відповідальний за проксюванні запитів, що стосуються отримання статичних файлів. Основним його завданням є перевірка прав доступу користувачів до конкретного заданого типу файлів, а також створення потоку передачі інформації від сховища до кінцевого користувача.

Управління правами доступу відбувається на основі системи рольового доступу до ресурсів. Кожний користувач має єдину роль, що має набір прав доступу до конкретних заданих кінцевих точок. Адміністратор може також створити токен доступу, що дозволяє стороннім сервісам використання кінцевих точок без проходження процесу аутентифікації на основі паролю.

3.6 Налаштування WebSocket інтерфейсу для отримання інформації у режимі реального часу

Rest API надає можливість отримувати інформацію за заданими параметрами виконуючи механізм запит-відповідь, де кожне оновлення інформації вимагає послідовного надсилання нового запиту. Для отримання інформації у режимі реального часу застосовується «WebSocket Gateway», підключення до якого надає можливість отримати оновлення як тільки таке оновлення відбудеться на стороні серверу.

Як було зазначено у підрозділі 3.5, «WebSocket Gateway» отримує повідомлення в асинхронному режимі від сервісу Database Writer, та розподіляє їх відповідно за типом, парою, та джерелом інформації, що дозволяє користувачам виконувати підписку лише на задані типи подій.

Технологічна основа побудови «WebSocket Gateway» полягає в застосування бібліотеки Socket.IO, котра надає спрощений інтерфейс взаємодії з протоколом WebSocket, а також надає власні додаткові функції, що допомагають надсилати та розподіляти трафік між клієнтами [76].

Перш за все, однією з таких функціоналів є абстракція кімнат, що дозволяє групувати декілька користувачів та надсилати однакову подію до всіх учасників створеної кімнати.

Іншою функцію контролю є можливість створення просторів імен, що дозволяють розділяти трафік між користувачами та кімнати на більш низькому рівні. Такий логічний розподіл зображено на рисунку 3.6:

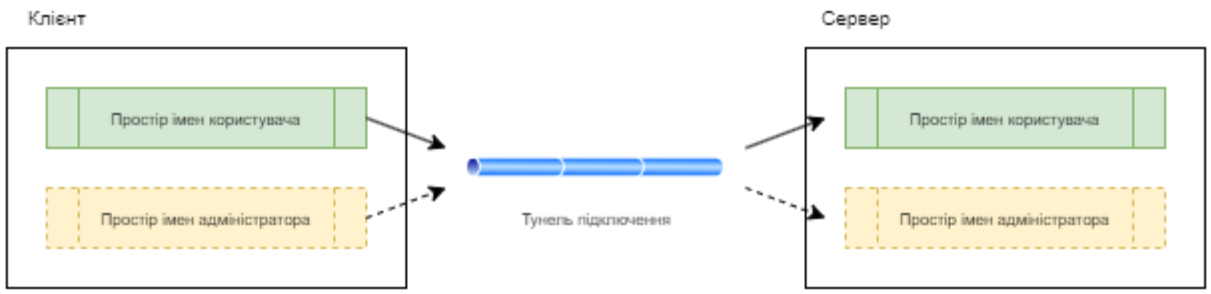

Рисунок 3.6 — Взаємодія із застосуванням Namespaces

Простір імен користувачів та адміністраторів логічно розділені на схемі рисунку 3.6, повідомлення між ними у такому випадку не будуть поширені, що є прямою вимогою системи розмежування прав доступу. Такий механізм корисний не тільки у випадку побудови системи розмежування доступу, але й при розділенні тематики, або зони відповідальності джерел подій.

Socket.IO імплементує та відповідальний за механізм відновлення підключень у випадку тимчасового перенавантаження мережі, або несправності сервера, що надає інтерфейс комунікації. У випадку повторних спроб під’єднання, Socket.IO також імплементує механізм очікування з експоненційним ростом, аби уникнути лишнього надсилання трафіку, та потенційних порушень політик безпеки.

Масштабованість системи, що надає інтерфейс вебсокетів досягається за рахунок адаптерів, основна задача котрих це ретрансляція повідомлень через

спільний сервіс, що підтримує функції підписки та оновлення, такий як Redis, та його здатність до функціонування як брокер повідомлень через механізми Pub/Sub [63-65].

Схема такого механізму зображена на рисунку 3.7:

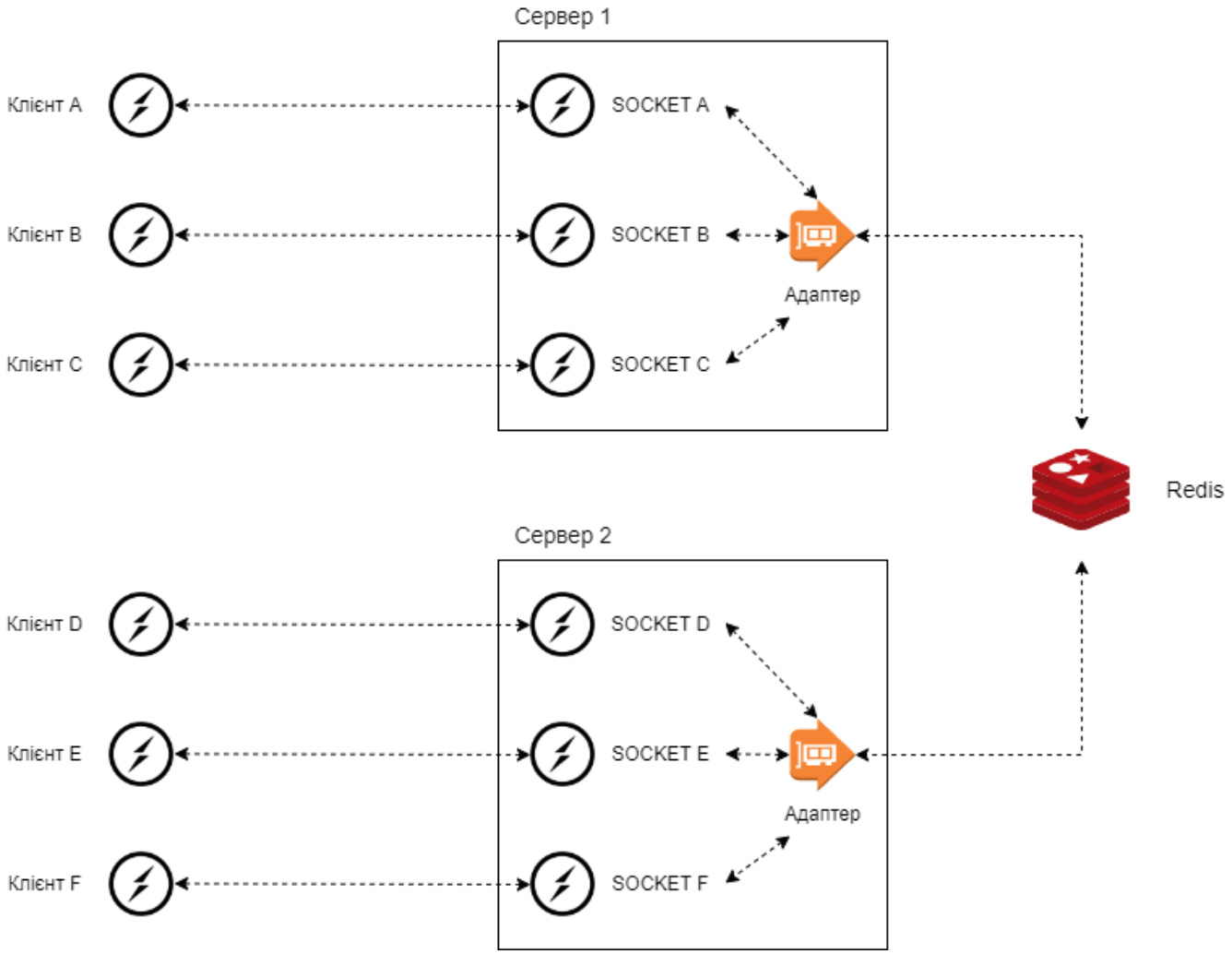

Рисунок 3.7 — Масштабування з’єднань WS

На рисунку 3.7 зображено вебсокети на стороні клієнті, котрі підключені до двох різних серверів 1 та 2 відповідно, та кожен з яких має вебсокети, що відповідають за управління з’єднанням на стороні сервера. При необхідності комунікації клієнта A з клієнтом F, повідомлення на сервері 1 буде ретрансльовано через адаптер на сервері 1 до адаптера на сервері 2 завдяки можливостям підписки, що надаються Redis. Адаптер на сервері 2 передасть отримане оновлення до вебсокета F на стороні серверу, котрий в свою чергу передасть повідомлення до вебсокета F на стороні клієнта.

Масштабованість трансляцій зібраної інформації вимагає іншого підходу та структури, оскільки джерелом комунікації у даному випадку є сервіс запису інформації у базу даних. Для побудови такої інфраструктури застосовується обмінник на основі заголовків, що дозволяє використання черг, котрі відповідальні за конкретну пару окремо. Такий розподіл відповідальності дозволяє балансувати навантаження у кластері сервісів RabbitMQ.

Схема механізму масштабування трансляцій зібраної інформації у режимі реального часу зображена на рисунку 3.8:

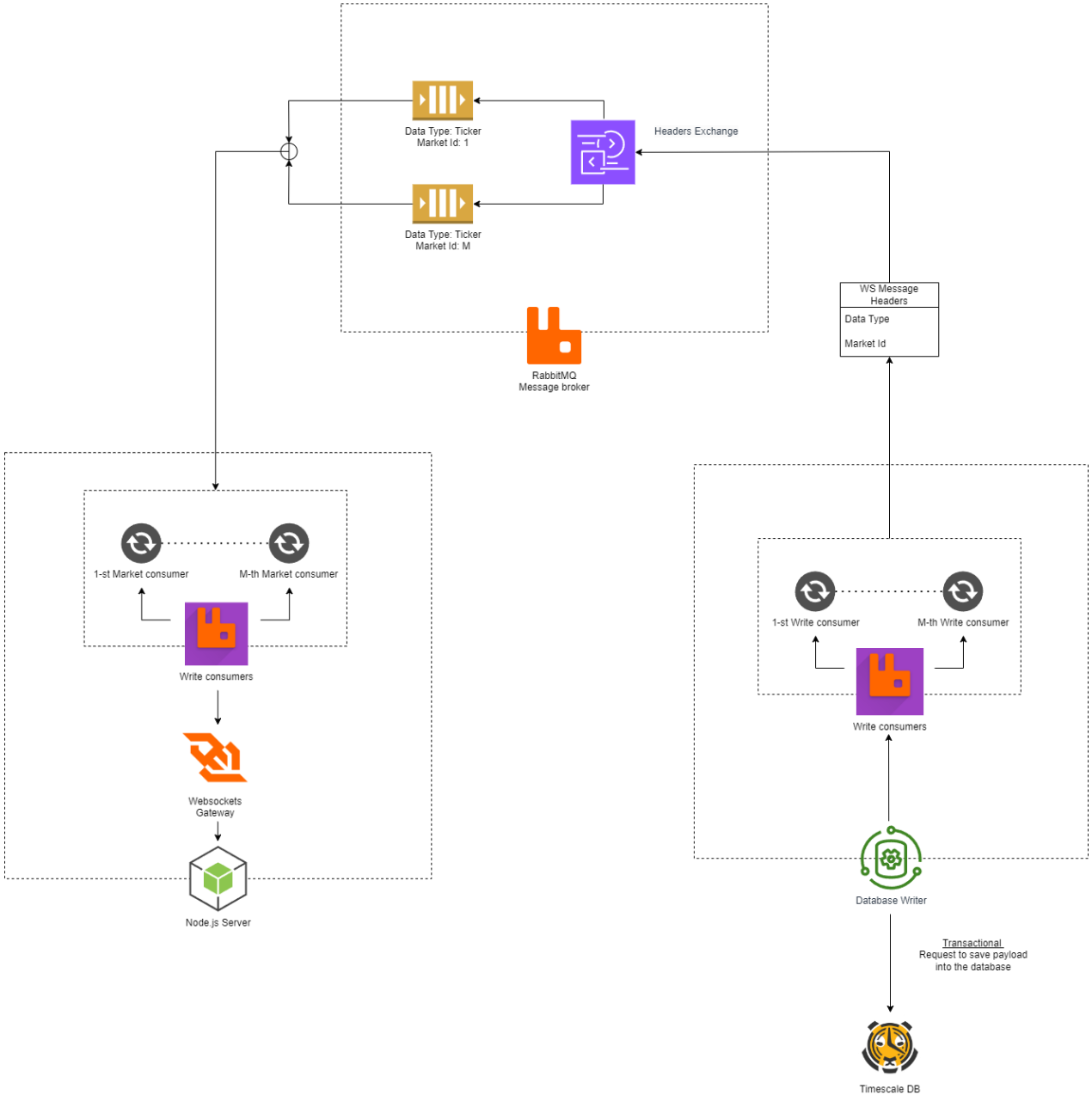

Рисунок 3.8 — Масштабування трансляцій WS

У такі структурі, при записі інформації у базу даних, сервіс Database Writer сконструює повідомлення, що має два заголовки: «Data Type», «Market Id», та надішле їх до обміннику заголовків. Обмінник, в свою чергу, виконає функцію маршрутизації до конкретної черги повідомлень. Споживачі на стороні сервісу «WebSocket Gateway» після цього будуть отримувати повідомлення з черги та перенаправляти їх лише до тих клієнтів, котрі знаходяться у відповідній кімнаті та були підключені до трансляції.

Однією з ключових можливостей Socket.IO є також застосування двох моделей комунікації: «HTTP polling» та «WebSocket», та автоматичне застосування одного чи іншого в залежності від можливості встановлення з’єднання. Спосіб «HTTP polling» працює на основі постійного повторюваного надсилання запитів до сервера про оновлення інформації, в той час як «WebSocket» дозволяють встановлювати повноцінний дуплексний зв'язок між клієнтом та сервером.

3.7 Розробка рекомендацій щодо побудови стійких розподілених систем

Розробивши декілька варіацій архітектур, оцінивши їх надійність та масштабованість, а також імплементувавши один із варіантів – було описано значну кількість складнощів, що виникають при побудові стійких розподілених систем. Зібравши рішення зазначених проблем та проаналізувавши їх вплив, було розроблено список рекомендацій, що направлені на впровадження кращих практик при розробці розподілених систем.

Першою рекомендацією є чітке визначення та розмежування рівнів абстракції у складній розподіленій системі. Абстрагування – фундаментальний принцип комп’ютерних наук, що дозволяє розробляти дедалі складніші системи, на основі вже існуючих. В заданому контексті, абстрагування дозволяє будувати складні системи управління та синхронізації без врахування різноманітних назькорівневих деталей, таких як канал комунікації чи середовище виконання. Абстрагування дозволяє також значно зменшити ймовірність виникнення критичних помилок при розробці системи, завдяки відсутності логічної надмірності в єдиному компоненті управління.

Другою рекомендацією є завчасне планування масштабування сервісів. Для того, аби система мала здатність постійно еволюціонувати, підтримувати все більше типів даних, джерел інформації, та одночасних підключень користувачів – завчасно повинна бути спланована та імплементована система горизонтального масштабування. Для досягнення цієї цілі, найпростішим способом є побудова сервісів, що не мають стану, оскільки горизонтальне масштабування не буде потребувати додаткової синхронізації. У випадку використання сервісів, котрі потребують наявність стану, як-от, наприклад, репліки всередині кластеру, що повинні автоматично розподіляти частини задач між собою – необхідна розробка системи синхронізації, що значно ускладнює побудову системи.

Третьою рекомендацією є орієнтація на слабкий зв'язок між представленими компонентами системи. Така система побудови дозволяє замінювати один компонент системи та змінювати його, без необхідності в оновлені іншого компоненту системи, що є важливим для систем, котрі постійно підтримуються та оновлюються. Слабкий зв’язок дозволяє розподілити відповідальність за розробку між декількома командами, кожна з яких відповідальна лише за імплементацію очікуваного API, та не має критичних вимог до внутрішньої структури системи зі сторони інших компонентів системи.

Четвертою рекомендацією є застосування сервісів, що мають значну спільноту та постійно вдосконалюються. Підтримка та впровадження розподілених систем є постійним процесом, що вимагає оперативного вирішення помилок у коді, та неочікуваних проблем, котрі можуть виникати в залежності як зовнішніх, так і внутрішніх чинників. Інтегруючи сервіси, що мають широку спільноту та постійно оновлюються, можливо значно знизити ризики пов’язані як з помилками на стороні інтегрованих сервісів, так і їх вразливостями. У випадку наявності широкої спільноти, значно вища ймовірність вчасного знаходження вразливостей та помилок, а також їх оперативне виправлення.

П’ятою рекомендацією є використання лише тих пакетів NPM, котрі мають значну спільноту та постійно оновлюються. Основні причини мають певну схожість із попереднім пунктом, проте застосовні на рівні побудови власних сервісів, а також

мають додаткові наслідки та причини. У випадку з інтеграцією пакетів NPM, доступ до створення нових бібліотек в реєстрі мають абсолютно всі охочі інженери та розробки, що може бути використано для ін’єкцій зловмисного коду та виконання неочікуваних сторонніх ефектів. Важливо завжди перевіряти джерело залежності, його спільноту та історію його застосування і уникати інтеграції пакетів, котрі не були перевірені спільнотою, або особисто інженером, котрий займається його інтеграцією у свою систему.

Шостою рекомендацією є вибір клієнтів, що мають вбудовані та протестовані механізми захисту і стійкості. Клієнт доступу до сервісу зазвичай використовується з єдиною причиною, аби уникнути надлишкових затрат зусиль, часу та ресурсів, на розробку власного клієнту, що буде мати такий самий функціонал. Проте важливо при виборі клієнта проаналізувати та переконатись його можливості щодо управління з’єднаннями та прав доступу до ресурсів. У випадку, якщо така перевірка доведе відсутність існуючих рішень, що імплементуються зазначені механізми – важливо створити їх власноруч, аби уникнути відмов сервісів виключно через тимчасові збої на платформі запуску або у мережі.

Сьомою рекомендацією є максимальне уникнення побудови систем безпеки самостійно з повного нуля. Існують безліч рішень, бібліотек, клієнтів, протоколів, та сервісів, що були розроблені та протестовані значною кількістю інженерів, дослідників, та користувачів. Такі рішення мають безліч вирішених проблем, виявити які неможливо без наявності широкої спільноти для тестування та перевірки, а тому наскільки б не здавалось, що нова система безпеки, розроблена без використання будь яких сторонніх сервісів, буде безпечною та надійною, у більшості випадків при тривалому застосуванні – зворотне може бути доведене та принести значних репутаційних та ресурсних збитків.

Восьмою рекомендацією є застосуванні рішень під час розробки, впровадження, та підтримки, що нівелюють вплив інфраструктури, її імплементації, та платформи на хід виконання бізнесової логіки системи. Для вирішення таких завдань існують декілька рішень, описані в підрозділах 1.5 та 3.5, основою котрих є орієнтованість на впровадження програмного коду, а не платформи. Такого ефекту

можна дістатись за допомогою раніше зазначених технологій віртуалізації, контейнеризації, та сервісів безсерверного виконання коду.

Дев’ятою та заключною рекомендацією є необхідність у побудові масштабованої та гнучкої системи моніторингу цільового розподіленого додатку та платформи, на якій такий додаток розгорнутий. Моніторинг дозволяє приймати зважені рішення при розробці та масштабуванні сервісів, а також виявляти критичні недоліки та вразливості системи.

Висновки за розділом 3

За результатами розробки архітектури у розділі 2, в розділі 3 було імплементовано розподілену систему, що застосовує метод часової самосинхронізації для досягнення ідемпотентності операцій.

Розглянуто та обґрунтовано застосування інфраструктурних сервісів, їх переваги та недоліки при масштабування. Розглянути механізми стійкості, відновлення інформації та робочого стану в інтегрованих сервісах, таких як TimescaleDB, Redis, RabbitMQ, Nginx, Docker.

Визначено та обґрунтовано застосування готових програмних рішень при розробці та імплементації власної кодової бази сервісів, відповідальних за збір, передачу, та обробку критично важливої інформації, що має високі вимоги до консистентності та послідовності. Розглянуто причини та переваги застосування асинхронної моделі обробки запитів, що є основним способом роботи сервісів на основі платформи Node.js. Оцінено можливості до масштабування та адаптивності до нестійких середовищ, готових клієнтів, таких як sequelize, redis io, та amqp connection manager.

До того ж, розглянути принципи застосування Docker для побудови образі сервісів, їх масштабування, інтеграції, впровадження, та постійної підтримки, а також, системи управління контейнерами, так як Docker Compose та Kubernetes.

Проаналізовано побудову сервісів збору інформації, так званих «колекторів», їх рівні абстракції та причини їх створення та ізоляції. Продемонстровано їх структуру

та рішення, що допомагають пришвидшувати розробку та підвищувати надійність інфраструктури збору інформації з великої кількості зовнішніх джерел.

Розглянуто побудову сервісу запису інформації у постійне сховище, його здатність до масштабованості та надійність. Надано детальний опис масштабованого механізму повторних спроб запису інформації, за допомогою можливостей брокера повідомлень RabbitMQ.

Описано основні компоненти головного API у вигляді HTTP серверу, що включає в себе аутентифікацію, кінцеві точки отримання інформації, комунікацію в режимі реального часу, та отримання статичних даних. До того ж, описано системи аутентифікації та авторизації доступу до інформації та сервісів системи.

Надано детальний огляд побудови масштабованих систем обміну інформації у режимі реального часу на основі бібліотеки Socket.io. Розглянуті можливості створення адаптерів та застосування функцій сторонніх сервісів для масштабування кімнат користувачів.

Врешті, в результаті розробки варіацій архітектур та імплементації системи розподіленого отримання та обробки інформації – створено список рекомендацій щодо побудови стійких розподілених систем.

ВИСНОВКИ

Основною ціллю даної дипломної роботи було дослідити існуючі, проаналізувати, та розробити нові протоколи, механізми та методи, що використовуються або можуть бути використані для побудови надійних розподілених систем. У сучасному суспільстві інформаційної ери, побудова стійких до непередбачуваних інцидентів систем має критичне значення, створюючи основу розвитку та функціонування життєво необхідних сфер діяльності людства.

Перший розділ даної дипломної роботи надає огляд історії розвитку обчислювальних систем, розвитку міжмережевої взаємодії та, зокрема взаємодії у високонавантажених розподілених системах.

Окрім цього, перший розділ надає детальний аналіз існуючих протоколів, що імплементують паттерн черги, для вирішення проблем раптових сплесків трафіку або запитів у мережі. Зокрема, розглянуто та порівняно протоколи AMQP та MQTT. Розглянуто їх функціональну основу та найбільш придатні сфери використання.

Розглянуто, проаналізовано та порівнянь способи та сфери використання брокерів обміну повідомлень, з особливим акцентом на RabbitMQ та Kafka. Порівняно можливості та швидкодію таких брокерів, їх здатності до масштабування, пропускну здатність, та додаткові функції нативної обробки повідомлень як-от маршрутизація за різними схемами, та логіка повторних спроб. Результати цього дослідження стали основою при прийнятті рішення щодо використання RabbitMQ як сервісу брокера, завдяки його здатності до встановлені складних схем маршрутизації, включи пряму адресацію, поширення повідомлень всім доступним чергам, адресацію на основі тематики та, в решті, адресацію на основі заголовків.

RabbitMQ має складнощі при потребі пропускної здатності мільйонів повідомлень та масштабуванні, проте існують методи, описані у першому розділі, що дозволяють RabbitMQ впоратись з таким навантаженням за допомогою реплікації та зменшення зони відповідальності єдиної черги.

Було проаналізовано використання різних методів та підходів щодо побудови архітектури системи, зокрема систем на основі мікросервісів та безсерверних рішень. Продемонстровано відмінність, сфера застосування, переваги та обмеження таких архітектур при побудові розподіленої системи, звертаючи особливу увагу на можливості масштабування та модульності.

До того ж, у першому розділі розглянуто історію створення та сучасні методи ізоляції сервісів, такі як віртуалізація та контейнеризації, їх вплив на складність побудови незалежних сервісів та вклад у гарантування надійності таких компонентів розподіленої системи. Розглянуто та проаналізовано застосування інструментів оркестрації та управління контейнерами, такі як Docker Compose та Kubernetes, їх можливості, призначення та обмеження.

Другий розділ зосереджений на розробці та оцінці стійкості різних варіантів архітектур системи розподіленого збору та обробки інформації. Побудовано план стійкого обміну та обробки інформації, що знаходиться у периметрі контролю розроблюваної системи, з особливою увагою та наголосом на важливості ізоляції рівнів абстракції системи та слабких зв’язків між різними сервісами.

Продемонстровано та проаналізовано ефективність механізму із зворотним хронологічним записом інформації для відновлення консистентності системи після інцидентів функціонування на різних етапах обробки, а також, наголошено на обмеженнях такого механізму та його залежність від джерела правдивої інформації.

Розроблено архітектуру, що застосовує модифікацію протоколу RAFT, для досягнення консенсусу між репліками у кластері щодо права надсилання інформації до сервісу запису у базу даних, а також, продемонстровано обмеження, до придатність використання такої системи у випадках, коли час – це критичний ресурс.

Досліджено поняття ідемпотентності та його застосовність до побудови сервісів у розподілений системи. На основі характеристики ідемпотентності – розроблено декілька варіацій архітектур, що досягають цільового функціоналу збереження інформації без потреби у підтримці складних протоколів синхронізації та вибору лідера.

В результати оцінки переваг та недоліків представлених механізмів – розроблено архітектуру на основі часової самосинхронізації, що дозволяє водночас уникнути складнощів побудови алгоритмів синхронізації та досягти позитивних ефектів таких як незалежність від підтримки сторонніх сервісів, а також складності розробки та підтримки такої системи.

Розглянуто використання протоколів виявлення стану репліку (Replica State Discovery Protocol), його вплив на підтримку масштабованості та динамічного розширення кількості учасників кластеру. Проаналізовано його роль у порівнянні з протоколами досягнення консенсусу, переваги, недоліки та сфери застосування.

У третьому розділі дипломної роботи було програмно імплементованої архітектуру на основі часової самосинхронізації, описану у другому розділі. За результатами практичної імплементації та попереднього аналізу застосування інфраструктурних одиниць системи – надано список рекомендацій щодо побудови надійних розподілених систем.

Досліджено та обґрунтовано процес прийняття рішень щодо інфраструктурних складових системи, таких як брокер повідомлень, база даних, сервіс управління кешу, застосування систем реверс-проксі, та керування контейнерами сервісів.

Представлено процес прийняття рішень щодо технологічної основи програмної імплементації сервісів. Розглянуто застосування платформи Node.js та її роль у побудові масштабованих мікросервісів. Надано опис та процес інтеграції бібліотеки, що абстрагує реалізації протоколу HTTP – Express.js. Обґрунтовано причини використання рішень ORM при створенні та управлінні пулами підключень до бази даних, а також, окреслено використання клієнтів інтегрованих сервісів, такі як «redis.io» та «amqp connection manager».

В ході практичної реалізації сервісів – описано архітектури компоненту колекторів, його рівні абстракції, їх зони відповідальності та способи взаємодії. Розглянуто сервіс запису інформації до бази даних, його функціонал та залежності. Описано створення та налаштування основного HTTP серверу, та сервісів стрімінгу інформації у режимі реального часу, на основі протоколу WebSockets.

Результати проведеного дослідження мають значний внесок у сферах комп’ютерних наук та, зокрема, кібербезпеки, дозволяючи досягти одного з основоположних принципів – «доступності».

Проведений аналіз протоколів та системних компонентів має на меті допомогти інженерам у сфері розробки розподілених систем у процесі прийнятті рішень. Розроблені варіації архітектур стійких систем, дозволять надати архітекторам оцінку придатності конкретних механізмів стійкості до використання у заданих бізнес-вимогах.

Результати імплементації описаної системи дозволяють розробникам програмно забезпечення отримати практичний досвід при побудові цільової розподіленої системи, ознайомитись із принципами побудови структурних частин такого рішення.

Наданий список рекомендацій на основі проведених аналітичних порівнянь, розроблених архітектур та імплементованої системи – дозволить покращити розуміння принципів побудови стійких розподілених систем, що мають здатність до масштабованості, є гнучкими у використанні та впроваджують найкращі практики відновлення після катастроф.

Підсумовуючи, завдання дипломної роботи було виконано у повному обсязі, а мета – досягнута. Аналітичні висновки та інновації у сфері кібербезпеки, отримані у ході виконання даної дипломної роботи – мають вирішальне значення, та можуть вплинути на безліч побудованих розподілених систем майбутнього.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is the OSI Model. [Електронний ресурс]. Режим доступу: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/.

2. Harrenstein, N. The history of the Internet and its protocols. [Електронний ресурс]. Режим доступу: https://medium.com/@nathanharrenstein/the-history-of-the-internet-and-its-protocols-c5327da5f8d5.

3. Segal, B. A short history of Internet protocols at CERN. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/245061001_A_short_history_of_Internet_protocols_at_CERN.

4. Greenemeier, L. Remembering the day the World Wide Web was born. [Електронний ресурс]. Режим доступу: https://www.scientificamerican.com/article/day-the-web-was-born/.

5. Hafner, K., & Lyon, M. Where wizards stay up late: The origins of the Internet. New York: Simon & Schuster, 1996. [Електронний ресурс]. Режим доступу: https://archive.org/details/wherewizardsstay00haf_vgj/page/n3/mode/2up.

6. Russell, A. L. Open standards and the digital age: History, ideology, and networks. New York: Cambridge University Press, 2014. [Електронний ресурс]. Режим доступу: https://books.google.com.ua/books?id=jqroAgAAQBAJ&pg=PA196&redir_esc=y#v=onepage&q&f=false.

7. Abbate, J. Inventing the Internet. Cambridge, MA: MIT Press, 2000. [Електронний ресурс]. Режим доступу: https://books.google.com.ua/books?id=E2BdY6WQo4AC&pg=PA123&redir_esc=y#v=onepage&q&f=false.

8. Bauer, D., Yuksel, M., Kalyanaraman, S., & Carothers, C. D. Understanding OSPF and BGP interactions using efficient experiment design. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/254957957_Understanding_OSPF_and_BGP_Interactions_Using_Efficient_Experiment_Design.

9. Isaacson, W. The innovators: How a group of hackers, geniuses, and geeks created the digital revolution. New York: Simon & Schuster, 2014. [Електронний ресурс]. Режим доступу: https://archive.org/details/innovatorshowgro0000isaa_p2p3.

10. Nguyen, Q. U., & Vu, H. N. A comparison of AMQP and MQTT protocols for Internet of Things. doi: 10.1109/NICS48868.2019.9023812. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/339759697_A_comparison_of_AMQP_and_MQTT_protocols_for_Internet_of_Things.

11. Gemirter, C. B., & Baydere, S. A comparative evaluation of AMQP, MQTT and HTTP protocols using real-time public smart city data. doi: 10.1109/UBMK52708.2021.9559032. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/355205575_A_Comparative_Evaluation_of_AMQP_MQTT_and_HTTP_Protocols_Using_Real-Time_Public_Smart_City_Data.

12. Kramer, J. Advanced message queuing protocol (AMQP). [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/234794767_Advanced_message_queuing_protocol_AMQP.

13. Shah, P. H. MQTT systems: A survey. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/379444424_MQTT_Systems_A_Survey.

14. O'Hara, J. Toward a commodity enterprise middleware. ACM Queue, 5(4), 2007. doi:10.1145/1255421.1255424. [Електронний ресурс]. Режим доступу: https://dl.acm.org/doi/10.1145/1255421.1255424.

15. Vinoski, S. Advanced message queuing protocol. IEEE Internet Computing, 10(6), 2006. doi:10.1109/MIC.2006.116. [Електронний ресурс]. Режим доступу: https://steve.vinoski.net/pdf/IEEE-Advanced_Message_Queuing_Protocol.pdf.

16. Hintjens, P. Background to the AMQ Project, Authors. iMatix Corporation, 2006, February 7. [Електронний ресурс]. Режим доступу: https://github.com/imatix/openamq/blob/master/website/doc_background.txt.

17. AMQP Working Group transitions to OASIS member section. 2012, April 16. [Електронний ресурс]. Режим доступу: https://web.archive.org/web/20120416223525/http://www.amqp.org/node/54.

18. OASIS forms AMQP Technical Committee to advance business messaging interoperability within middleware, mobile, and cloud environments. [Електронний ресурс]. Режим доступу: https://www.amqp.org/node/58.

19. Getting started with MQTT. HiveMQ, 2020, April 24. [Електронний ресурс]. Режим доступу: https://www.hivemq.com/article/how-to-get-started-with-mqtt/.

20. The HiveMQ Team. Introducing the MQTT Protocol - MQTT Essentials: Part 1. [Електронний ресурс]. Режим доступу: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/.

21. MQTT v3.1 and MQTT v3.1.1 differences. OASIS Message Queuing Telemetry Transport (MQTT) TC, 2015, February 12. [Електронний ресурс]. Режим доступу: https://www.oasis-open.org/committees/document.php?document_id=55095&wg_abbrev=mqtt.

22. MQTT V3.1 Protocol Specification. Eurotech, International Business Machines Corporation (IBM), 2010. [Електронний ресурс]. Режим доступу: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html.

23. What’s the difference between Kafka and RabbitMQ? Amazon Web Services. [Електронний ресурс]. Режим доступу: https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/.

24. Kafka vs RabbitMQ: Biggest differences and which should you learn? Simplilearn. [Електронний ресурс]. Режим доступу: https://www.simplilearn.com/kafka-vs-rabbitmq-article.

25. Understanding the differences between RabbitMQ and Kafka. VMware Tanzu. [Електронний ресурс]. Режим доступу: https://tanzu.vmware.com/content/blog/understanding-the-differences-between-rabbitmq-vs-kafka.

26. Flexible & powerful open source multi-protocol messaging. Apache ActiveMQ. [Електронний ресурс]. Режим доступу: https://activemq.apache.org/.

27. Eclipse Mosquitto an open source MQTT broker. [Електронний ресурс]. Режим доступу: https://mosquitto.org/.

28. Introduction to Azure Service Bus, an enterprise message broker. Microsoft. [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview.

29. RabbitMQ is the most widely deployed open source message broker. [Електронний ресурс]. Режим доступу: https://rabbitmq-website.pages.dev/.

30. Part 1: RabbitMQ for beginners - What is RabbitMQ? CloudAMQP. [Електронний ресурс]. Режим доступу: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html.

31. Understanding RabbitMQ queue & messaging simplified 101. Hevo Data. [Електронний ресурс]. Режим доступу: https://hevodata.com/learn/rabbitmq-queue/.

32. What is RabbitMQ? Wallarm. [Електронний ресурс]. Режим доступу: https://lab.wallarm.com/what/what-is-rabbitmq/.

33. What is Apache Kafka? Amazon Web Services. [Електронний ресурс]. Режим доступу: https://aws.amazon.com/what-is/apache-kafka/.

34. What is Kafka? Definition, working, architecture, and uses. Spiceworks. [Електронний ресурс]. Режим доступу: https://www.spiceworks.com/tech/data-management/articles/what-is-kafka/.

35. Apache Kafka use cases: When to use it? When not to? Upsolver. [Електронний ресурс]. Режим доступу: https://www.upsolver.com/blog/apache-kafka-use-cases-when-to-use-not.

36. Apache Kafka: The basics definition and uses. Aiven. [Електронний ресурс]. Режим доступу: https://aiven.io/developer/what-is-apache-kafka.

37. Larus, J. R. Evolution of computing. 2023. DOI:10.1007/978-3-031-45304-5_3. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/376715484_Evolution_of_Computing.

38. Zhao, W. Data and service replication. 2021. DOI:10.1002/9781119682127.ch4. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/352524573_Data_and_Service_Replication.

39. Mounine, H. S., Hioual, O., & Hioual, O. Dynamic load balancing upon the replication and deletion of cloud services. Journal of Intelligent & Fuzzy Systems, 44(2), 2022. DOI:10.3233/JIFS-221989. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/363629963_Dynamic_load_balancing_upon_the_replication_and_deletion_of_cloud_services.

40. Petrov, D. Clusterization method based on breadth-first search or BFS for a graph. 2022. DOI:10.31891/2307-5732-2022-307-2-87-91. [Електронний ресурс]. Режим

доступу: https://www.researchgate.net/publication/366405149_CLUSTERIZATION_METHOD_BASED_ON_BREADTH_FIRST_SEARCH_OR_BFS_FOR_A_GRAPH.

41. An overview of cluster computing. GeeksforGeeks. [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/an-overview-of-cluster-computing/.

42. Chehab, G. Computer clusters, types, uses, and applications. Baeldung. [Електронний ресурс]. Режим доступу: https://www.baeldung.com/cs/computer-clusters-types.

43. Systems and operations continuity: Disaster recovery. Georgetown University, University Information Services. 2012, August 3. [Електронний ресурс]. Режим доступу: https://emergencymanagement.georgetown.edu/.

44. Jorrigala, V. Business continuity and disaster recovery plan for information security. The Repository at St. Cloud State. [Електронний ресурс]. Режим доступу: https://repository.stcloudstate.edu/msia_etds/44.

45. Microservice architectures: More than the sum of their parts? IONOS Digitalguide. 2020, March 2. [Електронний ресурс]. Режим доступу: https://www.ionos.com/digitalguide/websites/web-development/microservice-architecture/.

46. Fowler, M. Microservices. [Електронний ресурс]. Режим доступу: https://martinfowler.com/articles/microservices.html.

47. Miller, R. AWS Lambda makes serverless applications a reality. TechCrunch, 2015, November 24. [Електронний ресурс]. Режим доступу: https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/.

48. MSV, J. PaaS vendors, watch out! Amazon is all set to disrupt the market. Forbes, 2015, July 16. [Електронний ресурс]. Режим доступу: https://www.forbes.com/sites/janakirammsv/2015/07/16/paas-vendors-watch-out-amazon-is-all-set-to-disrupt-the-market/?sh=3832b1dde88d.

49. Docker builds: Now lightning fast announcing Docker Build Cloud general availability. Docker. [Електронний ресурс]. Режим доступу: https://www.docker.com/.

50. What is container management and why is it important? TechTarget. [Електронний ресурс]. Режим доступу: https://www.techtarget.com/searchitoperations/definition/Docker.

51. What is Kubernetes? TechTarget. [Електронний ресурс]. Режим доступу: https://www.techtarget.com/searchitoperations/definition/Google-Kubernetes.

52. Kubernetes is software that automatically manages, scales, and maintains multi-container workloads in desired states. Mirantis. [Електронний ресурс]. Режим доступу: https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-is-kubernetes/.

53. Timescale logo products customer stories developers pricing contact us PostgreSQL ++ for time series and events. Timescale. [Електронний ресурс]. Режим доступу: https://www.timescale.com/.

54. An open-source database designed to make SQL scalable for time-series data. GitHub. [Електронний ресурс]. Режим доступу: https://github.com/timescale/timescaledb.

55. CCXT – Cryptocurrency exchange trading library. GitHub. [Електронний ресурс]. Режим доступу: https://github.com/ccxt/ccxt.

56. Wang, K. C. Process synchronization. In Design and Implementation of the MTX Operating System, 2015, June, pp. 175-214. DOI:10.1007/978-3-319-17575-1_6. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/300297849_Process_Synchronization.

57. Distributed locks with Redis. [Електронний ресурс]. Режим доступу: https://redis.io/docs/latest/develop/use/patterns/distributed-locks/.

58. Aggarwal, S. Raft and Paxos: Consensus algorithms for distributed systems. Medium. [Електронний ресурс]. Режим доступу: https://medium.com/@mani.saksham12/raft-and-paxos-consensus-algorithms-for-distributed-systems-138cd7c2d35a.

59. The Raft Consensus Algorithm. What is Raft? Raft Visualization. [Електронний ресурс]. Режим доступу: https://raft.github.io/.

60. Shukla, K. Build resilient systems with idempotent APIs. Medium, 2023, July 28. [Електронний ресурс]. Режим доступу: https://medium.com/@qlong/the-myths-of-idempotent-apis-in-practices-9025a94487f2.

61. Quanzheng, L. The myths of idempotent APIs in practices. 2023, December 5. [Електронний ресурс]. Режим доступу: https://dev.to/karishmashukla/building-resilient-systems-with-idempotent-apis-5e5p.

62. Kotov, M., Toliupa, S., & Nakonechnyi, V. REPLICA STATE DISCOVERY PROTOCOL BASED ON ADVANCED MESSAGE QUEUING PROTOCOL. Cybersecurity Education Science Technique, 3(23), 2024, March. DOI:10.28925/2663-4023.2024.23.156171. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/379626261_REPLICA_STATE_DISCOVERY_PROTOCOL_BASED_ON_ADVANCED_MESSAGE_QUEUING_PROTOCOL.

63. Introduction to Redis. [Електронний ресурс]. Режим доступу: https://redis.io/docs/about/.

64. Redis: What it is, what it does, and why you should care. [Електронний ресурс]. Режим доступу: https://backendless.com/redis-what-it-is-what-it-does-and-why-you-should-care/.

65. What is Redis? Benefits of Redis. Popular Redis Use Cases. Redis vs. Memcached. [Електронний ресурс]. Режим доступу: https://aws.amazon.com/redis/.

66. NGINX Plus software load balancer, web server, and cache. [Електронний ресурс]. Режим доступу: https://www.nginx.com/.

67. What is Nginx: Everything you need to know. [Електронний ресурс]. Режим доступу: https://www.papertrail.com/solution/guides/nginx/.

68. What is Node.js: A comprehensive guide. [Електронний ресурс]. Режим доступу: https://nodejs.org/en.

69. Node.js — Run JavaScript everywhere. [Електронний ресурс]. Режим доступу: https://www.freecodecamp.org/news/what-is-node-js/.

70. Express 4.18.1 Fast, unopinionated, minimalist web framework for Node.js. [Електронний ресурс]. Режим доступу: https://expressjs.com/.

71. Express/Node introduction. Web Frameworks. Introducing Express. Where did Node and Express come from? [Електронний ресурс]. Режим доступу: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

72. Modern TypeScript and Node.js ORM for Oracle, Postgres, MySQL, MariaDB, SQLite and SQL Server, and more. [Електронний ресурс]. Режим доступу: https://sequelize.org/.

73. Node.js database magic: Exploring the powers of Sequelize ORM. Medium. [Електронний ресурс]. Режим доступу: https://medium.com/@rishu__2701/node-js-database-magic-exploring-the-powers-of-sequelize-orm-a22a521d9d9d.

74. amqp-connection-manager. [Електронний ресурс]. Режим доступу: https://www.npmjs.com/package/amqp-connection-manager.

75. ioredis. [Електронний ресурс]. Режим доступу: https://www.npmjs.com/package/ioredis.

76. Socket.IO documentation and introduction. [Електронний ресурс]. Режим доступу: https://socket.io/docs/v4.

ДОДАТОК А

ЛІСТИНГ КОДУ ІНФРАСТРУКТУРНИХ КЛІЄНТІВ

Ініціалізація Sequelize:

```
function initModels(dbConfig, params) {
    const { database, username, password, dialect, host, port } = dbConfig;
    const retriesDefault = 4;

    Sequelize.useCLS(sequelizeTransactions);

    const sequelize = new Sequelize(database, username, password, {
        host,
        port,
        dialect,
        logging: false,
        dialectOptions: {
            connectTimeout: 10000,
            timezone: 'local',
        },
        pool: {
            min: 5,
            max: 50,
            idle: 10000, // The maximum time, in milliseconds, that a connection can be idle before being released.
            acquire: 120000, // ..., that pool will try to get connection before throwing error
        },
```

```
    retry: {
      // Set of flags that control when a query is automatically retried.
      match: [
        /SequelizeConnectionError/,
        /SequelizeConnectionRefusedError/,
        /SequelizeHostNotFoundError/,
        /SequelizeHostNotReachableError/,
        /SequelizeInvalidConnectionError/,
        /SequelizeConnectionTimedOutError/,
        /TimeoutError/,
        /SequelizeDatabaseError/,
      ],
      max:
        params && params.maxRetries
          ? +params.maxRetries
          : retriesDefault,
    },
});

const models = {
    // models dictionary    };

Object.values(models).forEach((model) => {
    model.init(sequelize);
});
Object.values(models).forEach((model) =>
    model.initRelationsAndHooks(sequelize),
);

BaseEntity.setSequelize(sequelize);
```

```
    return {
        ...models,
        sequelize,
    };
}
```

Ініціалізація Redis LRU:

```
class RedisLeastRecentlyUsedCache {
    constructor({
        ioRedisClient,
        namespace = 'defaultNamespace',
        maxCacheSize = 10000,
        timeToLiveSeconds = 3600,
    }) {
        this.redisClient = ioRedisClient;

        this.cacheNamespace = namespace;
        this.maximumCacheSize = maxCacheSize;
        this.timeToLive = timeToLiveSeconds;

        this.lruCacheKey = `${this.cacheNamespace}:LRUCache`;
    }

    _createNamespacedKey(key) {
        return `${this.cacheNamespace}:${key}`;
    }

    async insertIntoCache(key, value) {
```

```
        const namespacedKey = this._createNamespacedKey(key);
        const currentTimeStamp = Date.now();

        await this.redisClient.zadd(this.lruCacheKey, currentTimeStamp, namespacedKey);
        await this.redisClient.setex(namespacedKey, this.timeToLive, JSON.stringify(value));

        const currentCacheSize = await this.redisClient.zcard(this.lruCacheKey);

        if (currentCacheSize > this.maximumCacheSize) {
          const [oldestKey] = await this.redisClient.zrange(this.lruCacheKey, 0, 0);

          await this.redisClient.zrem(this.lruCacheKey, oldestKey);
          await this.redisClient.del(oldestKey);
        }
      }

      async retrieveFromCache(key) {
        const namespacedKey = this._createNamespacedKey(key);
        const cachedValue = await this.redisClient.get(namespacedKey);

        if (cachedValue) {
          const currentTimeStamp = Date.now();

          await this.redisClient.zadd(this.lruCacheKey, currentTimeStamp, namespacedKey);
          await this.redisClient.expire(namespacedKey, this.timeToLive);

          return JSON.parse(cachedValue);
```

```
        }

        return null;
    }

    async deleteFromCache(key) {
        const namespacedKey = this._createNamespacedKey(key);

        await this.redisClient.zrem(this.lruCacheKey, namespacedKey);
        await this.redisClient.del(namespacedKey);
    }

    async purgeCache() {
        const allCacheKeys = await this.redisClient.zrange(this.lruCacheKey, 0, -1);

        await this.redisClient.del(this.lruCacheKey);

        return Promise.all(allCacheKeys.map((key) => this.redisClient.del(key)));
    }
}
```

Ініціалізація AMQP Client:

```
class RabbitMQClient {
    constructor(configuration) {
        this.configuration = configuration;

        this.activeConnection = null;
        this.activeChannel = null;
    }
```

```
    async establishConnection() {
        this.activeConnection = await amqp.connect(this.configuration.urls);

        this.activeChannel = await this.activeConnection.createChannel();
    }

    async disconnect() {
        await this.activeChannel.close();
        await this.activeConnection.close();
    }

    sendToQueue(queueName, message) {
        const messageBuffer = Buffer.from(JSON.stringify(message));

        return this.activeChannel.sendToQueue(queueName, messageBuffer);
    }

    ensureChannelAvailable() {
        if (!this.activeChannel) {
            throw new Error('RabbitMQ channel is not established');
        }

        return this.activeChannel;
    }
}
```

Ініціалізація системи:

```
import initializeDatabaseModels from '#infrastructure/sequelize/
initializeDatabaseModels.js';
import systemConfig from '#configs/ systemConfig.cjs';
import applicationNamespace from '#infrastructure/namespaces/
applicationNamespace.js';
import createWinstonLogger from '#infrastructure/logger/ createWinstonLogger.js';
import MessageQueueClient from '#infrastructure/amqp/ MessageQueueClient.js';
import BaseApplicationUseCase from '#use-cases/ BaseApplicationUseCase.js';
import initializeRedisClient from '#infrastructure/ioredis/ initializeRedisClient.js';

class SystemInfrastructureProvider {
  systemConfig = systemConfig;
  appNamespace = applicationNamespace;
  databaseConnection = null;
  messageQueueClient = null;
  logger = null;

  constructor() {
    this.initializeServices();
    this.registerSignalHandlers();
  }

  async startInfrastructure() {
    await this.messageQueueClient.establishConnection();
  }

  async stopInfrastructure() {
    await this.messageQueueClient.disconnect();
    await this.databaseConnection.close();
    this.redisClient.disconnect();
```

```
            this.logger.info('[App] Shutdown complete');
            process.exit(0);
        }

        initializeServices() {
            this.logger = this.createLogger(this.systemConfig);
            this.databaseConnection = this.createDatabaseConnection(this.systemConfig);
            this.redisClient = this.createRedisConnection(this.systemConfig);
            this.messageQueueClient                                                    =
this.createMessageQueueConnection(this.systemConfig);
            this.initializeApplicationUseCases();
        }

        createLogger(config) {
            const logger = createWinstonLogger({
                ...config.logger,
                clsConfig: {
                    namespace: this.appNamespace,
                    contextKey: 'traceID',
                },
            });
            logger.info(`[App] Initialization Mode: ${config.app.mode}`);
            return logger;
        }

        createDatabaseConnection(config) {
            const databaseOptions = this.extractDatabaseOptions(config);
            const { sequelize } = initializeDatabaseModels(databaseOptions);
            return sequelize;
        }
```

```
createRedisConnection(config) {
   return initializeRedisClient(config.redis);
}

createMessageQueueConnection(config) {
   return new MessageQueueClient(config.rabbitmq);
}

initializeApplicationUseCases() {
   BaseApplicationUseCase.setInfrastructureProvider(this);
}

extractDatabaseOptions(config) {
   return config.db;
}

registerSignalHandlers() {
   process.on('SIGTERM', async () => {
      this.logger.info('[App] SIGTERM signal received');
      await this.stopInfrastructure();
   });

   process.on('SIGINT', async () => {
      this.logger.info('[App] SIGINT signal received');
      await this.stopInfrastructure();
   });

   process.on('unhandledRejection', (error) => {
      this.logger.emergency({
```

```
            type: 'UnhandledRejection',
            error: error,
          });
      });

      process.on('uncaughtException', (error) => {
          this.logger.emergency({
            type: 'UncaughtException',
            error: error,
          });
      });
    }
}

export default SystemInfrastructureProvider;
```

ДОДАТОК Б

ЛІСТИНГ КОДУ БАЗОВОГО КОЛЕКТОРА

```
import crypto from 'crypto';

class MarketDataCollector {
    constructor({
        logger,
        exchangeName,
        tradingSymbol,
        currencyPair,
        marketIdentifier,
        exchangeAPIConnector,
        messageQueueClient,
    }) {
        this.logManager = logger;
        this.exchangeName = exchangeName;
        this.tradingSymbol = tradingSymbol;
        this.currencyPair = currencyPair;
        this.marketIdentifier = marketIdentifier;
        this.exchangeAPIConnector = exchangeAPIConnector;
        this.messageQueueClient = messageQueueClient;

        this.DATA_QUEUE = 'default';
    }

    async collectData({ startInterval, endInterval }) {
        const collectorMetadata = {
```

```
            startInterval,
            endInterval,
            collectorId: crypto.randomBytes(4).toString('hex'),
        };

        this.logManager.debug({
            message: `Data collection for '${this.exchangeName} & ${this.tradingSymbol} & ${this.DATA_QUEUE}' initiated`,
            details: this.composeLogDetails(collectorMetadata),
        });

        try {
            const marketData = await this.retrieveMarketData(collectorMetadata);

            await this.storeCollectedData(marketData, collectorMetadata);
        } catch (error) {
            this.logManager.error({
                message: `Error in data collection for '${this.exchangeName} & ${this.tradingSymbol}'`,
                details: this.composeLogDetails(collectorMetadata),
                error,
            });

            throw error;
        }
    }

    async establishMessageQueueConnection() {
        this.messageQueueClient.getChannel().addSetup((channel) => {
            return channel.assertQueue(this.DATA_QUEUE, {
```

```
            durable: true,
        });
    });
}

dispatchToQueue(payload, { startInterval, endInterval, collectorId }) {
    return this.messageQueueClient.publish(this.DATA_QUEUE, {
        exchange: this.exchangeName,
        symbol: this.tradingSymbol,
        payload: {
            ...payload,
            startInterval,
            endInterval,
            marketId: this.marketIdentifier,
        },
        collectorId,
    });
}

composeLogDetails({ startInterval, endInterval, collectorId }) {
    return {
        startInterval: new Date(startInterval).toISOString(),
        endInterval: new Date(endInterval).toISOString(),
        symbol: this.tradingSymbol,
        exchange: this.exchangeName,
        marketId: this.marketIdentifier,
        collectorId,
    };
}
```

```
    /* istanbul ignore next */
    getCollectorName() {
        return this.constructor.name;
    }

    /* istanbul ignore next */
    async retrieveMarketData() {}

    /* istanbul ignore next */
    async storeCollectedData() {}
}

export default MarketDataCollector;
```

ДОДАТОК В

ЛІСТИНГ КОДУ СЕРВІСУ DATABASE WRITER

```
import { WS_PRICING_QUEUE } from '#constants/rabbimqQueues.js';
import MarketDataAMQPWorker from '../MarketDataAMQPWorker.js';

class MarketDataRecorder extends MarketDataAMQPWorker {
    async processMarketData({ exchange, symbol, payload, collectorTraceId }) {
        const [record, isCreated] = await this.MarketDataModel.findOrCreate({
            where: {
                intervalStart: payload.intervalStart,
                intervalEnd: payload.intervalEnd,
                marketId: payload.marketId,
            },
            defaults: {
                ...payload,
            },
        });

        if (isCreated) {
            this.amqpClient.sendToQueue(WS_PRICING_QUEUE, {
                exchange,
                symbol,
                payload,
                recordType: this.recordType,
                collectorTraceId,
            });
        }
```

```
            this.logger.debug({
                message: `${this.recordType} record for '${exchange} & ${symbol}' has been ${
                    isCreated ? 'created' : 'found'
                } successfully`,
                context: {
                    intervalStart: new Date(payload.intervalStart).toISOString(),
                    intervalEnd: new Date(payload.intervalEnd).toISOString(),
                    marketId: payload.marketId,
                    collectorTraceId,
                },
            });
        }
    }

    export default MarketDataRecorder;
```

ДОДАТОК Г

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Статті у наукових фахових виданнях України

1. Толюпа С., Наконечний В., Котов М., Солодовник В. Radio frequency signals encryption with AES in wireless data input devices // CEUR Workshop Proceedings. 2021. Вип. 2845. С. 96–105. (Scopus)

2. Котов М., Толюпа С., Наконечний В. REPLICA STATE DISCOVERY PROTOCOL BASED ON ADVANCED MESSAGE QUEUING PROTOCOL // Cybersecurity Education Science Technique. Березень 2024. Том 3, № 23. DOI: 10.28925/2663-4023.2024.23.156171. Ліцензія CC BY-NC-SA 4.0.

3. Котов М., Толюпа С., Наконечний В. METHODS OF BUILDING DURABLE UDP PORT MAPPINGS IN A NAT-BASED ENVIRONMENT // Cybersecurity Education Science Technique. Том 4, № 2. Ліцензія CC BY-NC-SA 4.0. (Статтю буде опубліковано у червні 2024 року)

Тези наукових доповідей

1. Котов М.С., Толюпа С. Intrusion Prevention System (IPS) // Тези доп. VII Міжнар. наук.-техн. конф. "Захист інформації і безпека інформаційних систем". Львів, Україна, 30-31.05.2019.

2. Толюпа С., Наконечний В., Дружинін В., Котов М. Model of the process of optimal planning of the modular structure of an information security system // Тези доп. IT&I 2019 Information Technology and Interactions.

3. Толюпа С., Наконечний В., Котов М., Солодовник В. RF Signals Encryption with AES in WDID96-105 // Тези доп. IT&I 2020 Information Technology and Interactions.

4. Фесенко А., Пономаренко Ю., Котов М., Руденко К. Data transfer technologies between password managers // Тези доп. Міжнар. наук.-практ. конф. "Прикладні системи та технології в інформаційному суспільстві".

5. Котов М., Толюпа С., Богуславська О. SYMMETRIC ALGORITHM OF BLOCK ENCRYPTION OR ADVANCED ENCRYPTION STANDARD // Тези доп. II Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2019.

6. Котов М.С., Толюпа С.В., Солодовник В.О. Asymmetric cryptographic algorithm or public key cryptographic algorithm RSA // Тези доп. III Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2020.

7. Блотницька Д., Толюпа С., Котов М. ANALYSIS OF NETWORK STEGANOGRAPHY METHODS // Тези доп. III Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2020.

8. Толюпа С., Наконечний В., Котов М. Analysis of reliability, security, and vulnerabilities of the Transport Layer Security protocol // Тези доп. IV Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2021.

9. Котов М., Толюпа С., Наконечний В. Prototype pollution vulnerability and its mitigation in Node.js environment // Тези доп. V Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2022.

10. Котов М., Толюпа С., Наконечний В. Node.js package management and security // Тези доп. VI Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2023.

11. Котов М., Толюпа С., Наконечний В. Resilience through Advanced Message Queuing Protocol and its security // Тези доп. VII Міжнар. наук.-практ. конф. "Проблеми кібербезпеки інформаційно-телекомунікаційних систем", 2024.