Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Гарантована доставка повідомлень у мікросервісній архітектурі

Текстова частина до магістерської роботи
за спеціальністю «Інженерія програмного забезпечення» 121

Керівник магістерської роботи
Доктор техн. наук, доцент
Глибовець А. М.

_____

(підпис)

“____” _____ 2023 р.

Виконав студент 2-го курсу магістратури
Смакула Р.В.
“____” _____ 2023 р.

Київ 2023

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики, к.ф.-м.н.

_____С. С. Гороховський

(підпис)

„____”_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на магістерську роботу

студенту 2 р. н. магістерської програми Інженерія програмного забезпечення

Смакулі Роману Васильовичу

Дослідити способи гарантування доставки повідомлень у мікросервісній архітектурі та реалізувати шаблон Transactional Outbox на мові програмування Kotlin, порівняти його швидкодію з аналогами.

Зміст текстової частини до магістерської роботи:

Зміст

Анотація

Вступ

1 Аналіз архітектурних шаблонів для гарантованої доставки повідомлення у чергу повідомлень.

2 Дослідження способів резервування черги повідомлень.

3 Аналіз CDC та інструментів для його реалізації.

4 Опис створеної бібліотеки, яка надає інструменти для реалізації шаблону Transactional Outbox, та порівняння її швидкодії з аналогами.

Висновки

Список літератури

Додатки

Дата видачі „___” _____ 2023 р.

Керівник

А.М. Глибовець, доктор технічних наук, доцент

_____

(підпис)

Завдання отримав

Р.В. Смакула _____

(підпис)

Календарний план виконання курсової роботи

Тема: Гарантована доставка повідомлень у мікросервісній архітектурі

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Розгляд та аналіз проблеми	Вересень-жовтень 2022р.	
2.	Пошук фреймворків для вирішення проблеми та порівняння їх між собою	Жовтень-грудень 2022р.	
3.	Розробка фреймворку для вирішення проблеми	Грудень-лютий 2022-2023р.	
4.	Замірювання ефективності фрейморку	Лютий-березень 2023р.	
5.	Написання текстової частини	Березень 2023р.	
6.	Перегляд праці науковим керівником	Квітень 2023р.	
7.	Підготовка до презентації роботи	Травень 2023р.	
8.	Презентація магістерської роботи	Червень 2023р.	

Студент Смакула Р.В _____

Керівник Глибовець А.М. _____

“_____” _____ 2023 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

2PC – Two-phase commit protocol

CDC – Change Data Capture

IoT – Internet of things

SQL – Structured Query Language

JDBC – Java Database Connectivity

JPA – Java Persistence API

API – Application Programming Interface

АНОТАЦІЯ

Дана робота має на меті дослідити методи, які дозволяють досягнути високого рівня гарантування доставки повідомлень у мікросервісній архітектурі. Для цього було розглянуто прикладну проблему, яка виникає в сфері розробки хмарних систем IoT.

В роботі розглянуті архітектурні шаблони для збереження інформації в сховищі сервісу та надсилання її в чергу повідомлень. Також розглянуто методи резервування черги повідомлень.

У практичній частині даного дослідження було розроблено бібліотеку, яка написана на мові програмування Kotlin та надає інструменти для реалізації шаблону Transactional Outbox. Було проведені заміри швидкодії даної бібліотеки та її аналогів.

ВСТУП

Мікросервісна архітектура має численні переваги, але з нею пов'язані й деякі виклики. Наприклад, з появою мікросервісів інформацію потрібно передавати через мережу. Такий канал комунікації є складним і у ньому часто виникають помилки. Також кожен сервіс має свій життєвий цикл, він може ставати недоступним у будь-який момент часу та може масштабуватися до більше, ніж одного екземпляру.

З появою мікросервісів з’являється проблема розподілених транзакцій. Адже часто виникає потреба зберегти інформацію на одному сервісі та надіслати її іншому. При цьому ситуація, при якій інформації збережеться лише на одному сервісі, неприпустима.

В даній роботів розглянута прикладна проблема, яка виникає в сфері розробки хмарних систем IoT. Способи її розв’язання можна застосувати до багатьох інших проблем в різних сферах.

Робота складається з чотирьох частин.

Перша частина містить інформацію про архітектурні шаблони, за допомогою яких можна зберегти інформацію в сховищі сервісу та надіслати її в чергу повідомлень. У цьому розділі розглянути такі шаблони: Transactional Outbox, 2PC, Listen to Yourself та Event sourcing.

У другій частині висвітлені методи резервування черги повідомлень. У ній розглянуто такі методи як використання Master/Slave та використання спільної файлової системи.

Третя частина включає дані про інструменти для реалізації CDC. У ній досліджується Debezium та Eventuate tram.

У четвертій частині описано практичну частину даної роботи. В ній розповідається про структуру створеної бібліотеки, а також про експеримент, який проводився для заміру швидкодії бібліотеки та її аналогів.

ПОРІВНЯННЯ ШАБЛОНІВ TRANSACTIONAL OUTBOX, 2PC, LISTEN TO YOURSELF ТА EVENT SOURCING

У цьому розділі описані шаблони, які вирішують проблему гарантованої доставки повідомлень у мікросервісній архітектурі. Вони відрізняються складністю реалізації, толерантністю до збоїв системи, швидкодією та іншим.

Для спрощення припустимо, що, якщо повідомлення потрапило в чергу повідомлень, то слухач черги гарантовано його отримає. В наступному розділі розглядається як досягнути цих гарантій.

1.1 Опис проблеми

Існує система IoT, в якій прилад надсилає сервісу А інформацію про події, які у нього виникають. Відповідальність сервісу А обмежується комунікацією з приладом та збереженням поточного стану приладу. Події обробляє та зберігає сервіс В.

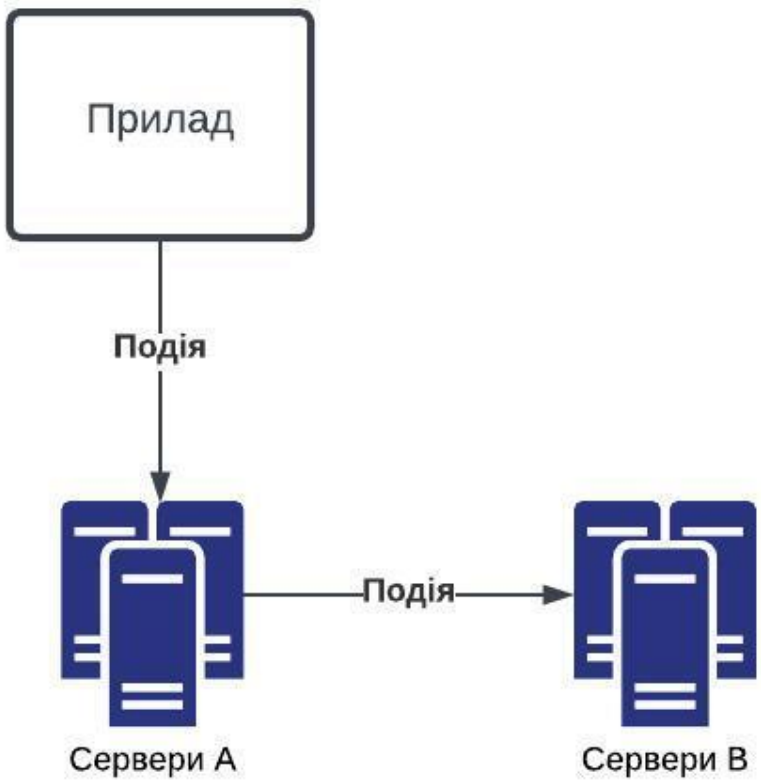

Рисунок 1

Потрібно створити таку систему, в якій події з приладу гарантовано збережуться сервісом В.

В чому виникає складність?

- сервіс А та В можуть втратити живлення в будь-який момент часу
- зв’язок між сервісами може зникати
- швидкість надсилання подій сервісом А інша від швидкості оброблення подій сервісом В
- може існувати декілька екземплярів сервісу А та В; дублювання подій неприпустиме

Оскільки сервіси мають різну швидкість роботи, комунікація між ними має бути асинхронна. Для такого виду комунікації підходить черга повідомлень. Тому кожен шаблон, який описаний у даному розділі, вирішує проблему гарантованої доставки події з сервісу А в чергу повідомлень. Сервіс В слухає чергу повідомлень та отримує подію звідти. Забезпечення гарантованості доставки повідомлення з черги сервісом В буде розглянуто у наступних розділах.

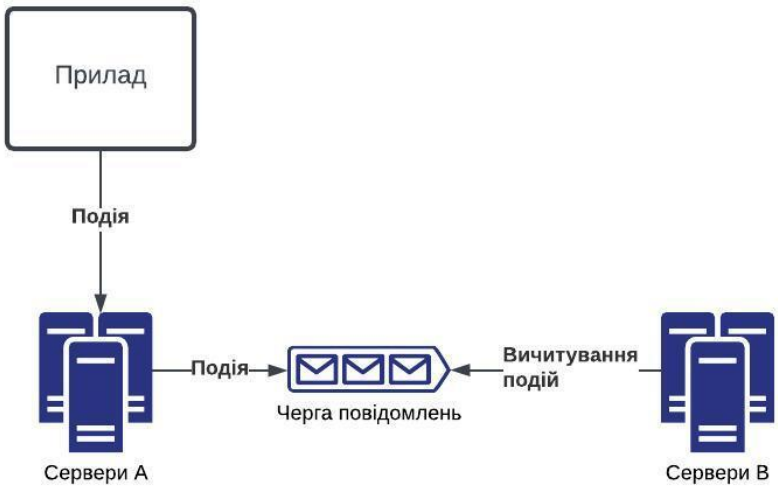

Рисунок 2

1.2 Проблема двох генералів

Проблема двох генералів є схожою до прикладної проблеми, яка вирішується в цій роботі. Вона звучить наступним чином: два генерали

взяли в облогу місто з різних сторін, вони розділені територією, на якій присутній їхній ворог. Генерали планують атакувати місто, проте вони не погодили час атаки. Щоб напад був успішних, їм потрібно його виконати одночасно. Для обміну повідомленнями генерали використовують посланців, проте, оскільки маршут лежить через ворожу територію, їх можуть вбити по дорозі.

Було доведено, що гарантовано досягти синхронізації часу нападу між двома генералами неможливо. Припустимо, що перший генерал надсилає повідомлення про час атаки. Якщо посланець дійде до другого генерала, то про це повинен дізнатися перший генерал, тому другий надсилає листа з підтвердженням отримання повідомлення. Якщо перший генерал отримає підтвердження, то йому потрібно повідомити другого генерала про це. Таким чином надсилання повідомлень про підтвердження отримання буде безкінечним.

Хоча не існує можливості гарантувати одночасну атаку двох генералів, існують алгоритми, які мають високу ймовірність досягнення цілі. Наприклад, за одним з алгоритмів потрібно вважати, що, якщо гінців не було протягом певного проміжку часу, то останній посланець дістався своєї цілі. Цей алгоритм використовує додатковий вимір - час. Щоб його можна було застосувати, потрібно визначити скільки часу потрібно гінцю для переходу ворожої території.

1.3 Transactional Outbox

1.3.1 Принцип роботи

Сервіс А, який приймає повідомлення від приладу, має залежність від бази даних. В базі присутня таблиця PANEL_CONFIGURATION, яка створена через потреби бізнес логіки, та PANEL_EVENT_OUTBOX - таблиця для реалізації шаблону Transactional Outbox. Сервіс А містить в собі компонент (також може бути сторонній сервіс), який слухає оновлення

таблиці PANEL_EVENT_OUTBOX та надсилає нові записи в чергу повідомлень.

Коли прилад надішле повідомлення, сервіс А створить записи в таблицях PANEL_CONFIGURATION та PANEL_EVENT_OUTBOX в рамках однієї транзакції. Таким чином уникається проблема успішного виконання бізнес логіки сервісу А та помилки надсилання події в чергу повідомлень та навпаки.

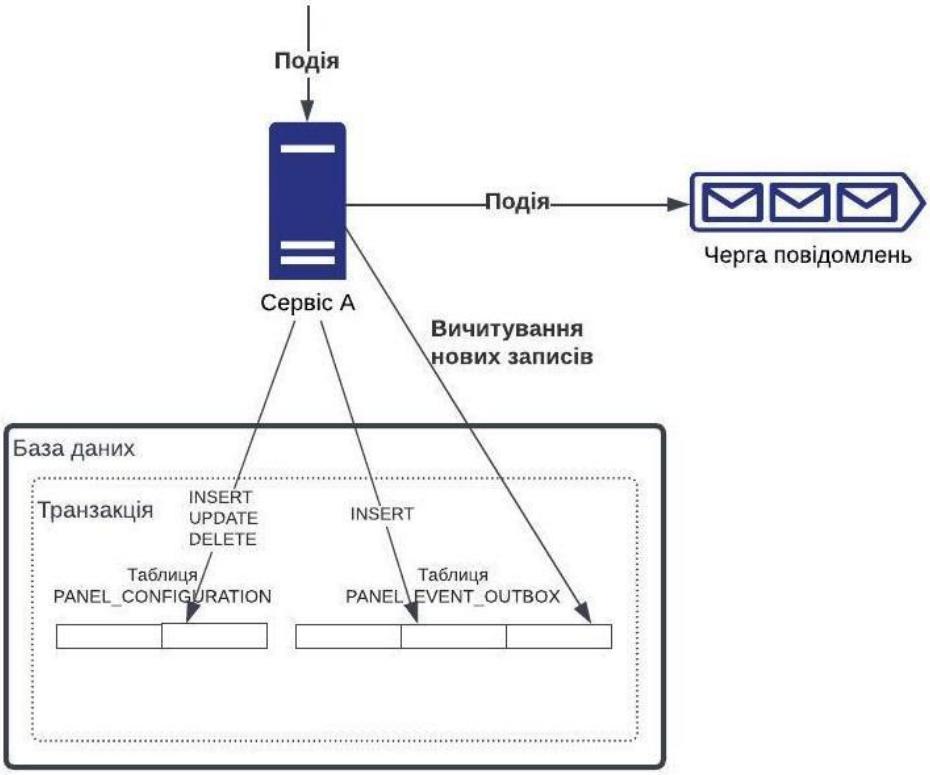

Рисунок 3

Існує два популярних методи слухання змін в таблиці PANEL_EVENT_OUTBOX: Polling publisher та Transaction log tailing.

1.3.2 Polling publisher

Отримання нових записів з таблиці PANEL_EVENT_OUTBOX реалізовується через періодичне вичитування таблиці. Перевагою є те, що цей підхід працює з усіма SQL базами даних та простий у реалізації. Недоліками є додаткова складність при забезпеченні правильного порядку

вичитування (потрібно мати додатковий стовпчик, який вказує на порядок запису). При великих навантаженнях такий підхід може працювати повільніше відносно Transaction log tailing, та при паралельному вичитуванні важко уникнути дублюючих оброблень записів.

1.3.3 Transaction log tailing

У цьому підході вичитується лог транзакцій бази даних, щоб дізнатися про нові записи у таблиці PANEL_EVENT_OUTBOX. У різних базах даних різне API для вичитування логу транзакції. Наприклад, у PostgreSQL лог транзакцій представлений у форматі Postgres WAL. Перевагами такого підходу є швидкість та невелике навантаження на процесор. Один з основних недоліків є необхідність специфічної реалізації для кожної бази даних. Для вирішення такої проблеми існують інструменти Debezium, Eventuate Tram та інші. Також варто зазначити, що не всі бази даних підтримують такий підхід.

1.3.4 Переваги

Транзакційність легко реалізовується на сервісі A. Якщо обрати Polling Publisher як метод слухання змін у таблиці, то такий підхід швидко реалізовується без використання сторонніх залежностей. У разі обрання методу Transaction Log Tailing можна використати доступні фреймворки, що полегшить та пришвидшить розробку.

1.3.5 Недоліки

Найбільшим недоліком такого підходу є надмірна залежність від бази даних. Якщо база даних стає недоступною, то блокується робота усього сервісу. Тому використання шаблону Transactional Outbox має бути обмеженим.

1.4 2PC

1.4.1 Принцип роботи

2PC один з найбільш поширених в індустрії програмного забезпечення протоколів для реалізації розподілених транзакцій. У нього

існує багато варіацій, проте в цьому розділі розглянута найпопулярніша версія.

2PC складається з двох частин та потребує координатора транзакції.

У випадку прикладної проблеми, яка розглянута у даній роботі, координатором може виступати сервіс A, учасниками транзакції є база даних, яка зберігає дані для бізнес логіки, та черга повідомлень.

На першому кроці сервіс A (координатор) розсилає запити на виконання транзакції до бази даних та черги повідомлень. Вони перевіряють чи можуть виконати транзакцію та надсилають відповідь координатору.

На другому кроці координатор перевіряє відповіді від учасників розподіленої транзакції. Якщо хоча б один з них проголосував за скасування транзакції, то її буде скасовано: координатор надішле їм команду ROLLBACK. Якщо база даних та черга повідомлень можуть виконати транзакцію, то сервіс A надсилає учасникам команду COMMIT. Вони отримують цю команду, завершують локальну транзакцію та надсилають підтвердження виконання команди.

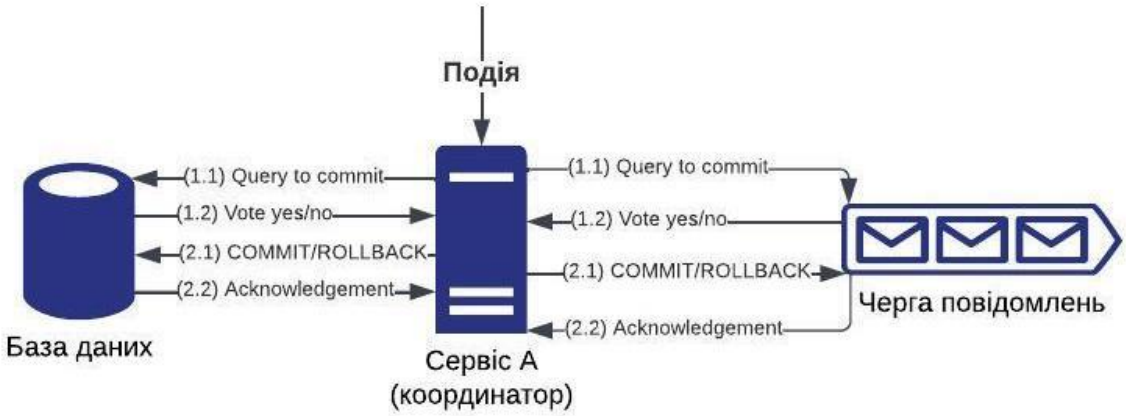

Рисунок 4

1.4.2 Переваги

2PC популярний протокол, його реалізовують багато хмарних компонентів, наприклад, PostgreSQL та ActiveMQ. Тому, щоб використати цей шаблон, потрібно затратити відносно небагато ресурсів. Two-phase

commit добре задокументований, в інтернеті наявно багато ресурсів про нього.

1.4.3 Недоліки

Попри свою популярність цей протокол має велику кількість недоліків. Основною його вадою є швидкість виконання. Причиною повільного виконання розподіленої транзакції виникає через необхідність комунікації між серверами. При масштабування тривалість 2PC збільшується. Також, якщо дві паралельні транзакції стосуються одної множини інформації, сервіс (база даних або інші) не буде повідомляти про готовність виконати другу за порядком отримання транзакцію поки не отримає команду COMMIT/ROLLBACK стосовно першої транзакції.

Другим основним недоліком є нетолерантність до недоступності координатора. Якщо на другому кроці, коли учасники транзакції проголосували за продовження її виконання, сервіс А стане недоступним, то учасники не зможуть ні скасувати транзакцію, ні виконати її, оскільки вони не знатимуть про готовність виконати транзакцію іншими учасниками. Для вирішення цієї проблеми був створений протокол 3PC, проте через повільну швидкодію та складність він не набув великої популярності.

Основною вадою 2PC в рамках прикладної проблеми, яка вирішується у даній роботі, є залежність від доступності черги повідомлень та бази даних. Прилад не отримає відповідь про успішне збереження події, якщо один з учасників розподіленої транзакції буде недоступним. Також очікування на відповідь приладом буде довге через тривалість виконання транзакції.

1.5 Listen to Yourself

1.5.1 Принцип роботи

Listen to Yourself передбачає зміни у підході до розробки мікросервісів. На відміну від дій в Transactional Outbox сервіс А при отриманні події від приладу не виконує бізнес логіку, а одразу пересилає її

в чергу повідомлень. Сервіс А, окрім записування в чергу, слухає її. Тому він вичитає подію, яку поклав у чергу повідомлень та виконає локальну бізнес логіку. Якщо ця подія порушує умови бізнес логіки сервісу А, то він надсилає повідомлення про скасування транзакції. Це повідомлення отримає інший слухач черги та скасує зміни, які він зробив після оброблення події.

Listen to Yourself має багато спільного з 2PC, проте за першим локальні транзакції виконуються паралельно, тоді як за 2PC координатор в першому кроці має опитати усіх учасників. Listen to Yourself передбачає, що кожен з учасників транзакції може її виконати, не знаючи при цьому про готовність виконати транзакцію іншими учасниками.

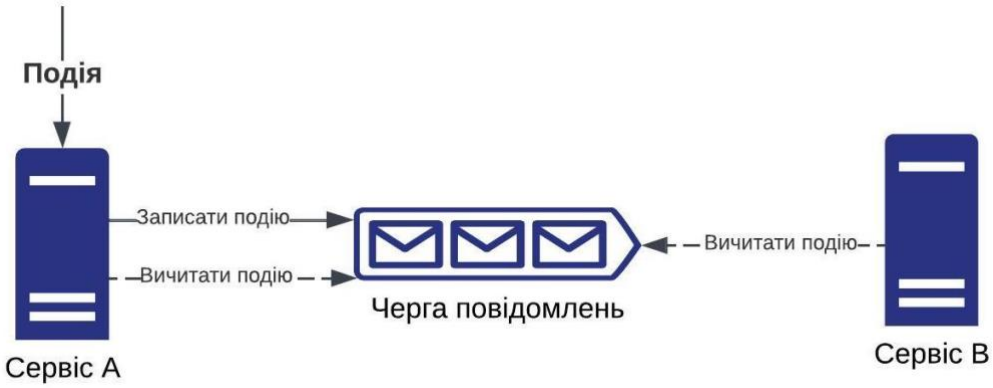

Рисунок 5

1.5.2 Переваги

Listen to Yourself працює швидко, оскільки кожному з учасників транзакції не потрібно знати про готовність виконати транзакцію іншими учасниками. Це дає змогу уникнути зайвої комунікації. Також цей шаблон уникає надмірної залежності від бази даних, яка присутня в Transactional Outbox.

У рамках прикладної проблеми сервіс А зможе швидко відповідати приладу, оскільки створення запису у черзі повідомлень займає відносно небагато часу.

1.5.3 Недоліки

Listen to yourself не підходить для випадків, коли паралельна обробка запиту не може мати місце. Адже існує ненульова ймовірність, що сервіс B швидше обробить подію, ніж сервіс A.

Хоча уникається залежність від бази даних, створюється інша від черги повідомлень. Тому доступність черги повідомлень має бути високою.

Реалізація цього шаблону передбачає зміну підходу до розробки мікросервісів, тому його адаптація може вимагати затрати багатьох ресурсів.

1.6 Event sourcing

1.6.1 Принцип роботи

Event sourcing – це шаблон, за яким інформація про стан зберігається у вигляді послідовних подій. Сервіс A після отримання події від приладу зберігає її у Event store, який її обробить та оновить інформацію про поточний стан приладу.

Event store дає можливість стороннім сервісам підписуватися на зміни в ньому. Таким чином він також відіграє роль черги повідомлень, передаючи інформацію з сервісу A на сервіс B.

Самостійно реалізувати Event store вимагає багато ресурсів, проте існує багато реалізацій, серед найпопулярніших Kafka, EventStoreDB.

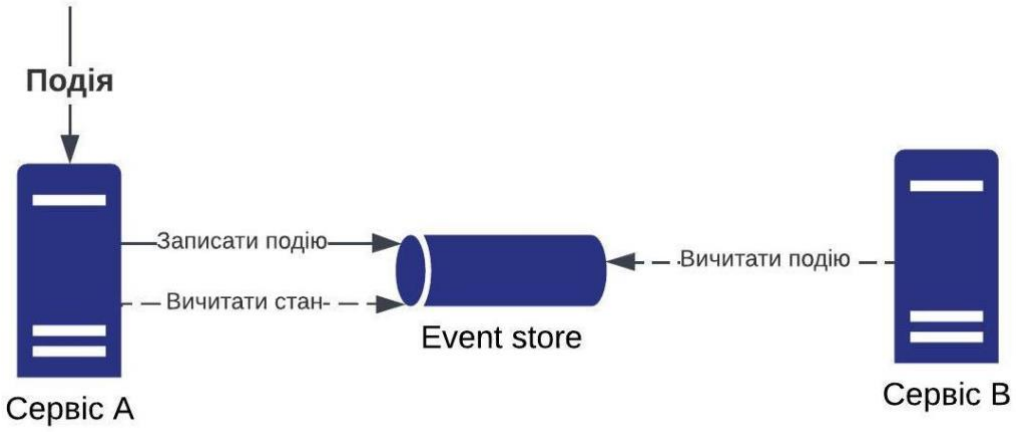

Рисунок 6

1.6.2 Переваги

Шаблон Event sourcing толерантний до недоступності усіх елементів системи. Поточний стан можна визначити після обробки усіх подій спочатку. Це дозволяє уникнути помилок в поточному стані, а за їх наявності легко відновитися до правильного стану.

Іншою перевагою даного підходу є швидкодія. Запис події та вичитування стану відбувається відносно швидко.

Також Event sourcing можна легко масштабувати: створення додаткових сервісів типу сервісу B не має бути складним завданням.

1.6.3 Недоліки

Попри багато переваг, які налічує Event sourcing, у нього є вагомий недолік. Його адаптація часто потребує істотних змін поточної архітектури системи. Він передбачає незнайомий для багатьох інженерів стиль програмування. Тому часто такий підхід важко реалізувати, і через надмірну складність реалізації архітектори системи відкидають його.

ГАРАНТОВАНА ДОСТАВКА ПОВІДОМЛЕННЯ ЧЕРЕЗ ЧЕРГУ ПОВІДОМЛЕНЬ

У цьому розділі розглядаються підходи до використання черги повідомлень, використовуючи які можна досягнути мінімальної ймовірності втрати повідомлень та можливості витримувати високі навантаження. Було досліджено чотири найбільш популярні реалізації черг - ActiveMQ Classic, Kafka, RabbitMQ та IBM MQ. Кожна з них має власний підхід до гарантування доставки повідомлення клієнту. Методи, які описані у цьому розділі можуть не бути наявними в всіх досліджених реалізаціях черги, а також можуть мати іншу назву.

Варто зазначити, які переваги має черга над іншими брокерами повідомлень. Важливою перевагою черги є те, що той, хто надсилає повідомлення, може працювати з іншою швидкістю, ніж той, хто отримує. Ще однією перевагою черги є те, що надсилач не має знати нічого про отримувача та навпаки. Це дозволяє масштабувати сервери та зменшити взаємозв'язки між компонентами системи.

Масштабування черги може вирішити дві проблеми. По-перше, це збільшить кількість проходження даних за одиницю часу через чергу. По-друге, певні реалізації кластеризації дадуть змогу забезпечити гарантоване збереження повідомлень навіть при недоступності одного з екземплярів черги.

При виборі стратегії масштабування черги можна обрати лише дві опції з трьох: швидкість обробки повідомлення, кількість повідомлень, які обробляються за одиницю часу (паралелізм), та гарантованість збереження повідомлення.

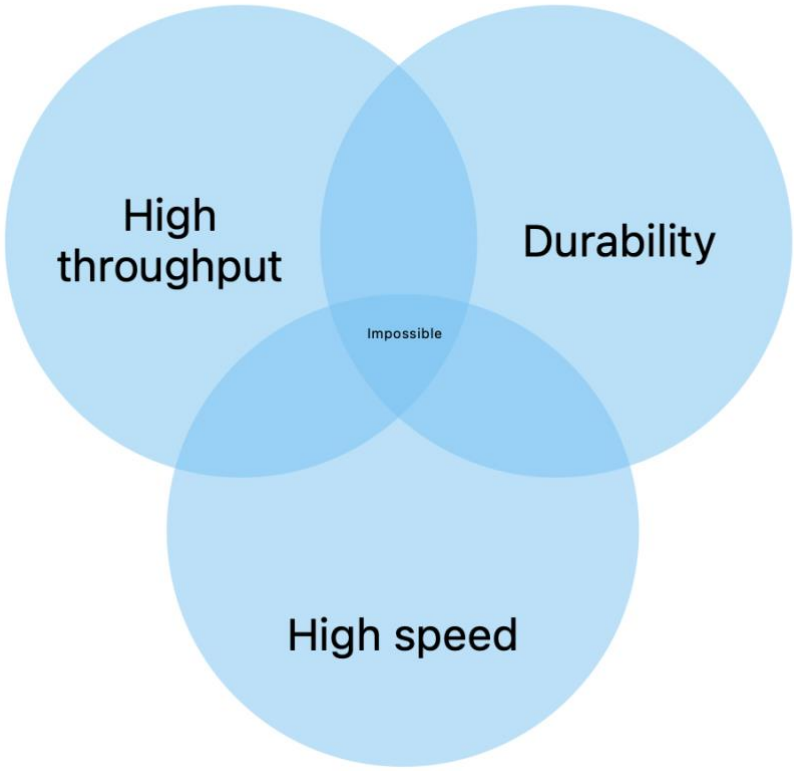

Рисунок 7

2.1. Кластеризація черги повідомлень

Кластер — це декілька незалежних обчислювальних машин, що використовуються спільно і працюють як одна система для вирішення тих чи інших задач, наприклад, для підвищення продуктивності, забезпечення надійності, спрощення адміністрування тощо. [1]

Кластеризація в різних реалізаціях черги повідомлень реалізована по різному, проте кожна з них має спільні елементи.

Головною метою кластеризації є збільшити рівень навантаження, який може витримати черга. Для цього навантаження рівномірно розподіляється по всіх учасниках кластеру. Розподіляти навантаження можна у двох місцях: на клієнті або на одному з учасників кластеру.

В кластері кожен учасник має мати змогу обмінюватися інформацією з будь-яким іншим учасником. Для цього кожен брокер має знати адреси інших брокерів кластеру. Також клієнту потрібно знати адреси брокерів. Ці адреси на кожному брокері та клієнті можна вказувати самостійно або можна використати механізм динамічного пошук учасників кластеру.

Динамічний пошук особливо корисний, коли кількість екземплярів у кластері велика та може змінюватися.

В кожній реалізації черги повідомлень, яка розглянута у цьому дослідженні, застосовується оптимізація трафіку між учасниками кластеру. Наприклад, в ActiveMQ Classic, якщо на брокері кластеру не зареєстрований слухач, якому адресується повідомлення, то цьому брокеру це повідомлення не буде переслано.

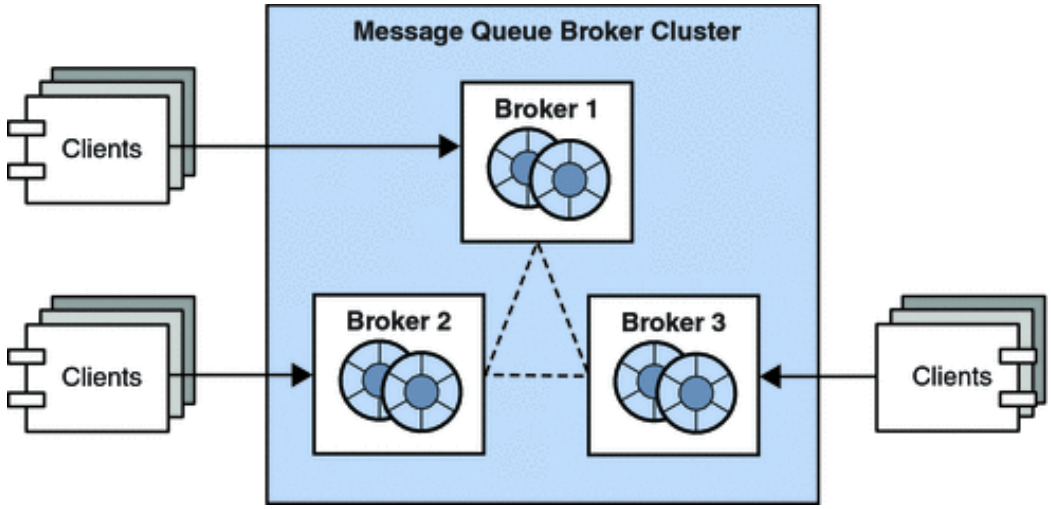

Рисунок 8 [2]

Кластеризація збільшує рівень навантаження, який може витримати черга, проте самостійно вона не забезпечить гарантованість доставки повідомлення. Може відбутися так, що брокер отримає повідомлення від клієнта, надішле йому відповідь про успішне виконання операції, проте стане недоступним перед тим як переслати це повідомлення клієнту або іншому учаснику кластеру. Це призведе до того, що клієнт не отримає повідомлення, якщо черга, на якому воно зберігається не відновиться.

2.2 Реплікація черги повідомлень

Щоб уникнути втрат повідомлень після того як екземпляр черги стає недоступним, використовують різні методи реплікації черги.

Серед найбільш популярних методів можна виокремити Master/Slave та використання спільної файлової системи.

Резервування екземпляру черги може гарантувати з високою ймовірністю те, що повідомлення не буде втрачено, проте це не збільшить рівень навантаження, який може витримати черга. Для збільшення кількості інформації, яка передається через брокера повідомлень потрібно розподіляти навантаження між декількома логічними чергами.

2.2.1 Використання Master/Slave

Метод Master/Slave - це модель асиметричної взаємодії чи комунікації де один прилад чи процес (“Master”) контролює один чи більше інших пристроїв чи процесів (“Slaves”) і служить їх комунікаційним концентратором. [3]

Реалізації Master/Slave відрізняється в різних реалізаціях черг.

Наприклад, в ActiveMQ всі клієнти під’єднуються до головного брокера та всі брокери мають спільну базу даних. Якщо головний брокер стає недоступним, то його роботу підхоплює другорядний брокер. Недоліком такого підходу є висока залежність від бази даних та нездатність розподіляти навантаження між учасниками кластеру.

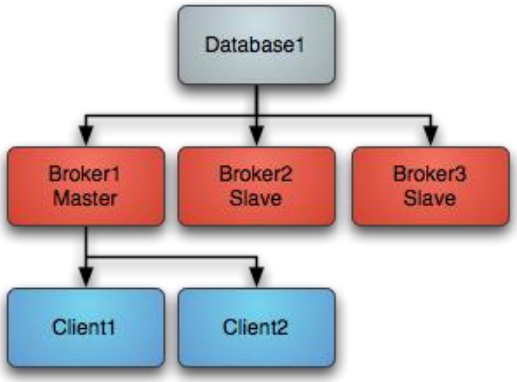

Рисунок 9 [4]

2.2.1.1 Quorom Queue

Kafka, RabbitMQ та IBM MQ використовують Quorum Queue як основний інструмент для реплікації черг.

Основна відміність Quorom queue від звичайного підходу реплікації є використання алгоритму Raft.

Raft — це консенсусний алгоритм, розроблений таким чином, щоб його було легко зрозуміти. Він еквівалентний Paxos за відмовостійкістю та продуктивністю. Різниця полягає в тому, що він розкладається на відносно незалежні підпроблеми, і він чітко стосується всіх основних частин, необхідних для практичних систем. [5]

Роботам Qurom Queue починається з голосування за головних брокерів кластеру. Коли клієнт надсилає повідомлення в чергу, воно зберігається на всіх головних чергах кворуму. Даний підхід стійкий до недоступності головного брокера кластеру. Коли це трапляється, кворум обирає нового головного брокера.

Варто зазначити, що Qurom Queue буде приймати повідомлення клієнтів лише тоді, коли всі головні брокери доступні. У разі недоступності усіх головних брокерів повідомлення будуть втрачені.

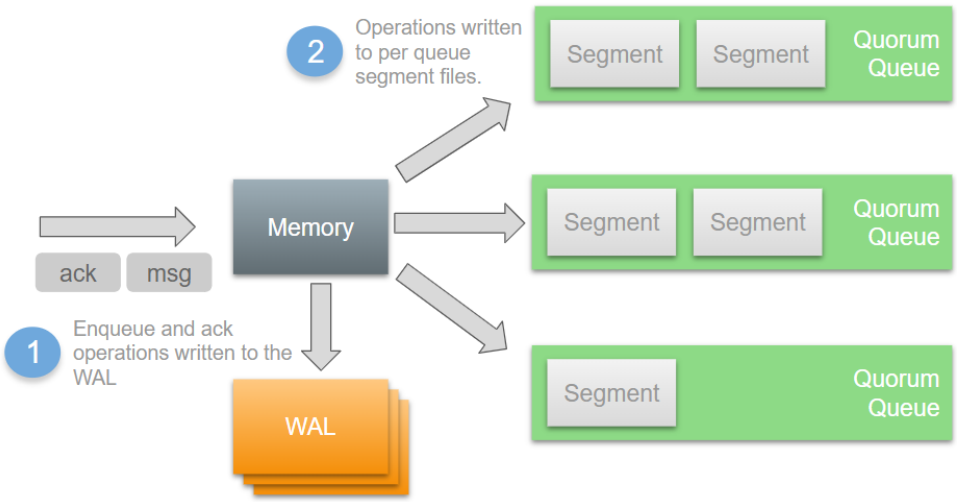

Рисунок 10 [6]

2.2.2 Використання спільної файлової системи

Замість спільної бази даних можна використовувати спільну файлову систему. У цьому випадку лише один брокер буде обслуговувати клієнтів в певний момент часу, блокуючи при цьому сховище для інших брокерів. Якщо основна черга стає недоступною, то вона перестає блокувати спільну файлову системи, і її функцію переймає інша черга.

Багато реалізацій черги повідомлень використовують журнальні сховища, які працюють швидше за інші типи сховищ. Тому спільна файлова система працює швидше за Master/Slave.

Даних підхід вимагає, щоб файлова система була швидка та завжди доступна, що не завжди можна легко досягнути. У цьому методі вся система надмірно залежна від сховища, і він не вирішує проблему високого навантаження.

2.3 Гарантованість доставки повідомлення через чергу повідомлень

Розглянувши методи кластеризації та реплікації черг, можна сказати, що існують підходи, з якими можна досягнути високої ймовірності передачі повідомлення через чергу. Кожен підхід має позитивні та негативні сторони. Спільна файлова система хоч і дає змогу швидко обробляти повідомлення, проте вимагає потужного обладнання. Quorom Queue може обробляти велику кількість повідомлень за одиницю часу та гарантує збереження повідомлення з високою ймовірністю, проте через це швидкість оброблення одного повідомлення не є високою.

CHANGE DATA CAPTURE

CDC - набір шаблонів програмування, які використовуються для визначення та відслідковування зміни даних, щоб мати змогу застосувати різні операції над цими змінами. [7]

У цьому розділі описується методи отримання змін даних та розглядаються інструменти, які реалізовують ці методи: Debezium, Eventuate tram.

CDC використовується у шаблоні Transactional Outbox, який було розглянуто у першому розділі. Цей шаблон легко застосувати у рамках прикладної проблеми, яка вирішується в даній роботі, він дає можливість швидко відповідати приладу. Тому було вирішено детальніше дослідити CDC - основну частину шаблону Transactional Outbox.

Здебільшого CDC використовують, щоб мігрувати велику кількість даних з одного цифрового сховища на інше. В мережі доступно багато інструментів для цього, наприклад, striim [8] та qlick [9]. Хоча вони надають інструменти CDC, проте їх складно застосувати для імплементації Tranactional Outbox. Debezium та Eventuate tram популярні засоби для вирішення такої задачі.

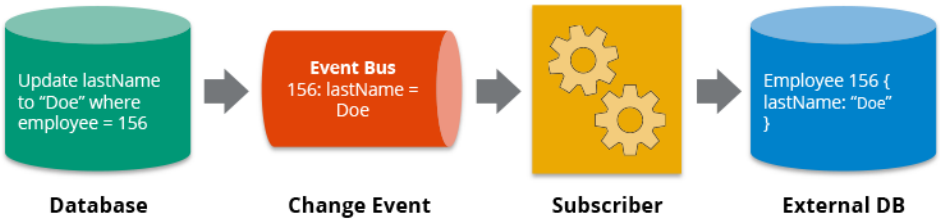

Рисунок 11 [10]

3.1 Використання допоміжних колонок

Щоб отримувати зміни, які відбулися в таблиці за певний проміжок, можна використати допоміжну колонку, в якій зберігати час оновлення запису. Ще потрібно створити допоміжну таблицю (може бути будь-яке інше сховище), в якій зберігати час оновлення останнього запису, який був оброблений. Таким чином, щоб отримати зміни, які були застосовані до

певної таблиці, потрібно відфільтрувати усі її записи, які мають час оновлення менший, ніж час оновлення останнього обробленого запису.

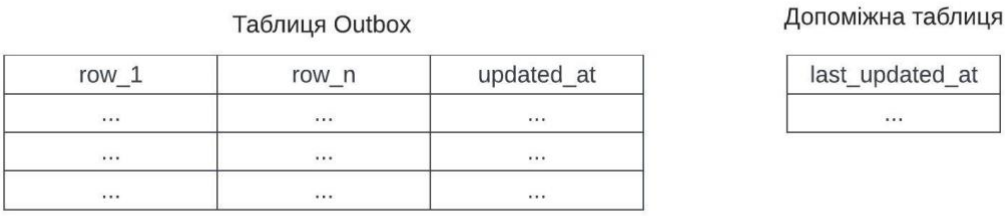

Рисунок 12

Переваги:

- Не потрібно використовувати жодних сторонніх інструментів
- Можна реалізувати зі всіма базами даних

Недоліки:

- Надмірне навантаження на базу даних
- Важко відслідкувати видалення записів

3.2 Пошук різниці між таблицями

Одним із варіантів отримання змін, які відбулися в таблиці, є порівняння її з минулою версією. Для цього кожного разу після виконання операції пошуку змін потрібно зберегти поточний стан таблиці в допоміжній таблиці.

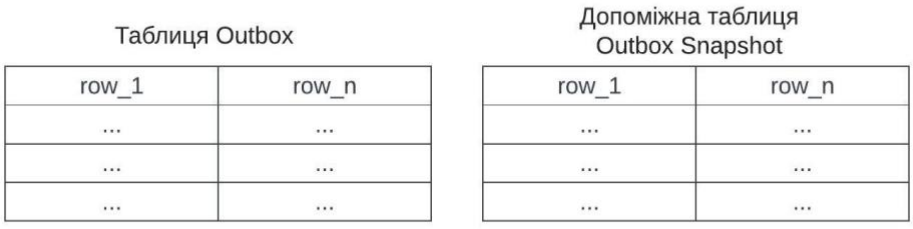

Рисунок 13

Переваги:

- Можна знайти зміни в таблиці, використовуючи лише SQL скрипт
- Легко відслідкувати усі види змін

Недоліки:

- Потрібно багато обчислювальних ресурсів
- Потрібно велике сховище

3.3 Використання тригерів

Популярні бази даних, серед них PostgreSQL, MySQL, надають інструмент тригерів. Цей інструмент повідомляє про зміни в таблиці одразу після їх застосування. Він повідомляє про зміни, викликаючи SQL процедуру, яку можна визначити самостійно.

Переваги:

- Не потрібно періодично вичитувати вміст таблиці

Недоліки:

- Хоча тригер викличе SQL процедуру, передати цю інформацію з неї на процес, на якому виконується основна бізнес логіка, не завжди легко

3.4 Використання логу транзакцій

Використання логу транзакцій - це найпопулярніший підхід серед тих, які описані в цій роботі. Бази даних зберігають лог транзакцій, який дозволяє їм відновитися у випадку тимчасової недоступності. У ньому зберігаються інформація про всі події, які відбулися з базою даних. Цей лог можна використати для отримання змін в таблиці за певний проміжок часу. Для цього потрібно зберігати час останньої вичитування логу, щоб знати який лог вказує на нові зміни.

Рисунок 14

Переваги:

- Невелике навантаження на базу даних
- Легко відслідкувати усі види змін

Недоліки:

- В кожної реалізації бази даних різна структура логу
- В різних мажорних версіях реалізації бази даних може бути різна структура логу

3.5 Debezium

Debezium — це проєкт із відкритим вихідним кодом, який забезпечує платформу потокового передавання даних із низькою затримкою для збору змінених даних (CDC). [11]

Debezium отримує зміни в таблиці, використовуючи лог транзакцій. Він вміє декодувати лог транзакцій більшості популярних баз даних. Користувач Debezium отримує дані в моделі, структура якої не залежить від того, яка база даних використовується.

Debezium зберігає інформацію про останнє вичитування логу транзакції в сховищі, і він гарантує, що клієнт отримає всі зміни, незалежно від того чи були перерви в роботі. Існує багато варіантів де зберігати інформацію про останнє вичитування логу. Найпоширенішим варіантом є збереження у файлі.

Існують дві опції розгортання Debezium: як зовнішній або як вбудований сервіс. Використання Debezium як сторонній сервіс надає перевагу в тому, що у такому разі гарантується відсутність дублікатів змін. Проте потребується більше ресурсів для підтримки такої архітектури.

3.6 Eventuate tram

Eventuate tram – це платформа, яка надає інструменти для вирішення проблем управління розподіленими даними в мікросервісній архітектурі. [12]

Eventuate tram, як і Debezium, імплементує CDC, проте він адаптований під роботу з Spring Boot/Java застосунком, які використовують JDBC/JPA.

Eventuate tram вміє декодувати логи транзакцій PostgreSQL та MySQL. Для інших баз даних він використовує допоміжні колонки, щоб отримати зміни в таблиці за певний проміжок часу.

Щоб використати Eventuate tram для отримання змін в таблиці, потрібно запустити сторонній сервіс, який називається Eventuate tram CDC.

Eventuate tram та Debezium відрізняються в основному тим, що перший створений для реалізації шаблонів Transactional Outbox, Saga, тоді як другий використовуються лише для CDC. Eventuate tram може бути легше використати в Spring Boot/Java застосунку, проте Debezium пропонує інтеграцію з більшою кількістю баз даних.

ОПИС ПРАКТИЧНОЇ ЧАСТИНИ

У практичній частині даної роботі було створено бібліотеку, яка реалізована на мові програмування Kotlin та надає інтсрументи для реалізації шаблону Transactional Outbox. Також був проведений експеримент для порівняння швидкодії шаблону Transactional Outbox реалізованого за допомогою власної бібліотеки, Debezium та Envetuate tram.

У цьому розділі описана структура створеної бібліотеки, а також структура проєкту для проведення експерименту і його результати.

4.1 Опис структури бібліотеки

Бібліотека складається з двох модулів:

- transactional-outbox
- transactional-outbox-spring-boot

4.1.1 Модуль transactional-outbox

У даному модулі наявна реалізація підходу CDC та парсинг змін бази даних PostgreSQL, які вичитуються з транзакційного логу також відомого як Postgres WAL.

Бізнес логіка міститься в пекеджі transactional.outbox, а інтеграція з PostgreSQL розташована в пекеджі transactional.outbox.postgres.

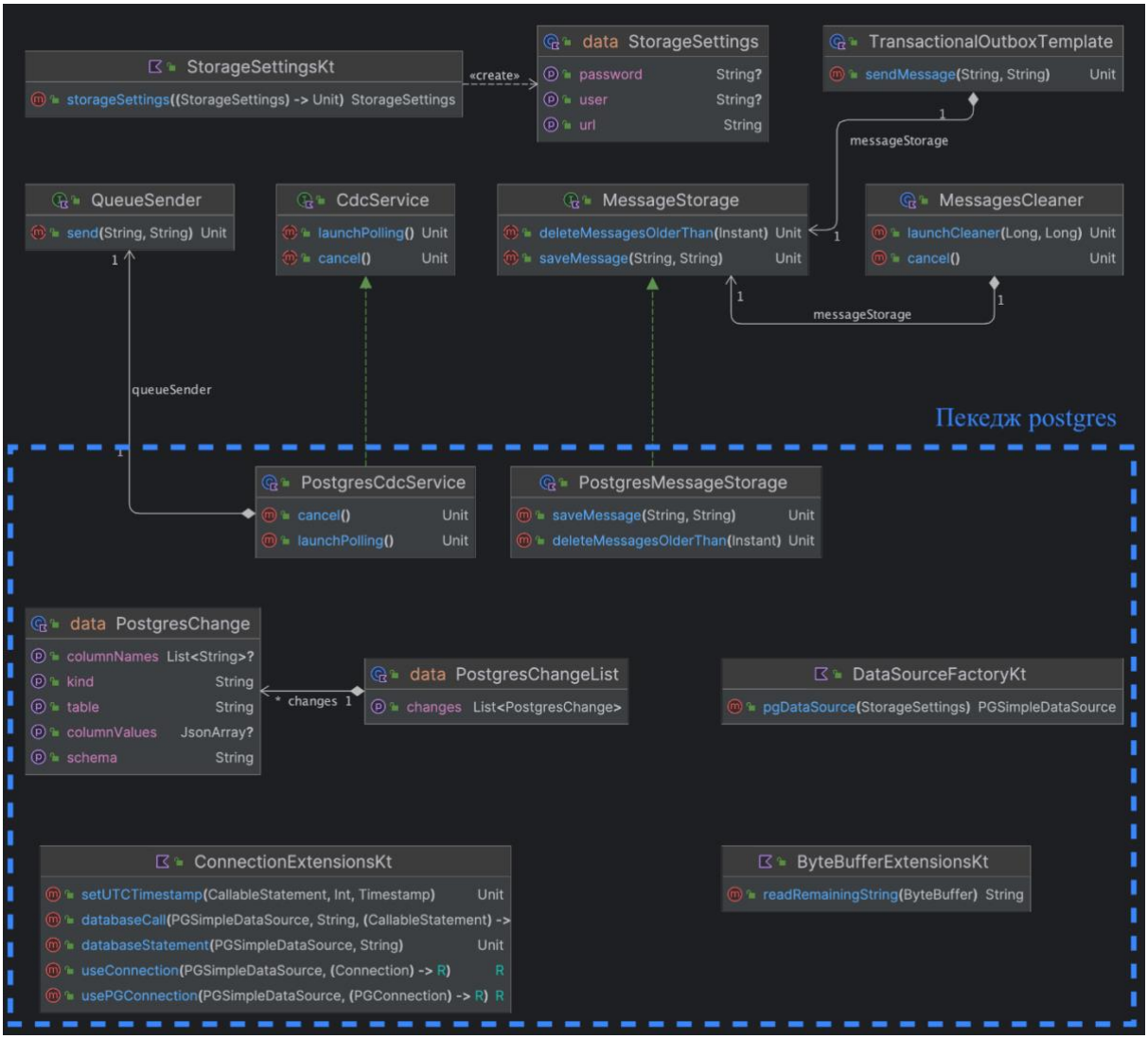

Рисунок 15

4.1.2 Модуль transactional-outbox-spring-boot

Даний модуль містить код, мета якого надати можливість легко використати модуль transactional-outbox в проєкті, який використовує фреймворк Spring Boot.

У модулі transactional-outbox-spring-boot міститься один клас з назвою TransactionalOutboxPostgresWalConfiguration. У цьому класі вказано які імплементації інтерфейсів потрібно використовувати, щоб налаштувати роботу бібліотеки з PostgreSQL.

Щоб використати модуль transactional-outbox-spring-boot, потрібно в Spring Boot проєкті в класі з анотацією @Configuration додати анотацію @Import(TransactionalOutboxPostgresWalConfiguration::class).

4.2 Опис структури проєкту для проведення експерименту

Для порівняння швидкодії роботи шаблону Transactional Outbox реалізованого за допомогою власної бібліотеки, Debezium та Eventuate tram було створено проєкт, який складається з трьох модулів:

- panel-emulation
- service-a
- service-b

4.3 Модуль panel-emulation

У даному модулі реалізований функціонал, який дає змогу емулювати роботу приладу IoT. Його відповідальність полягає у тому, щоб періодично надсилати згенеровані події на сервер.

Земульованому приладу можна задавати різні параметри, наприклад, кількість повідомлень, які потрібно надіслати, а також перерва у часі між повідомленнями.

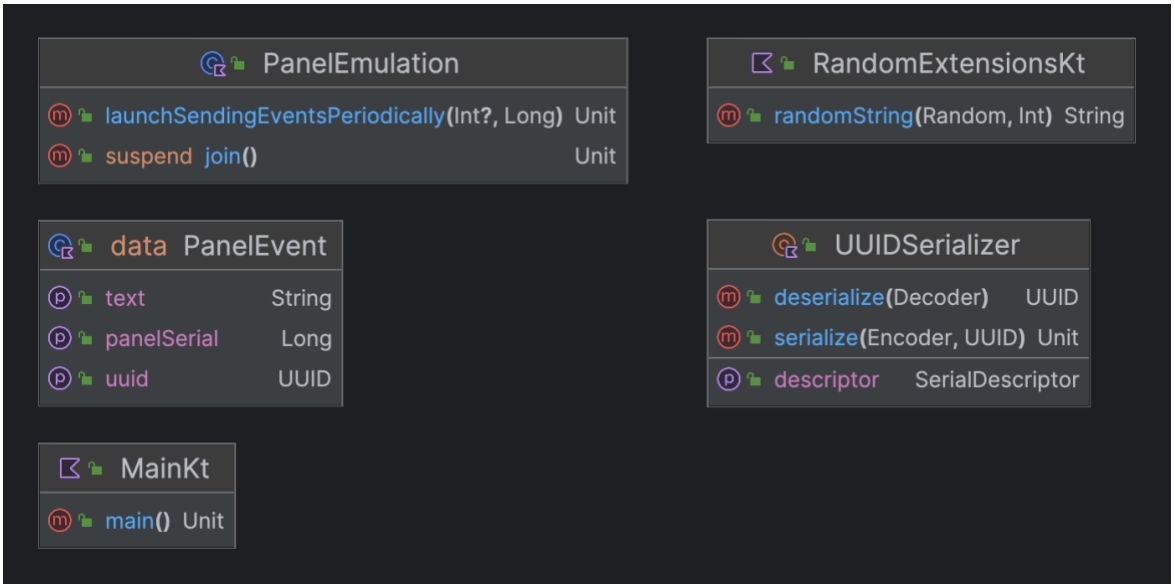

Рисунок 16

4.4 Модуль service-a

Модуль service-a описує сервер, який приймає повідомлення від приладу та з використання шаблону Transactional Outbox надсилає повідомлення в чергу RabbitMQ.

У цьому модулі міститься реалізація Transactional Outbox за допомогою власної бібліотеки, Debezium та Eventuate tram.

Також у модулі service-a наявні SQL міграції, які необхідні для імплементації шаблону Transactional Outbox.

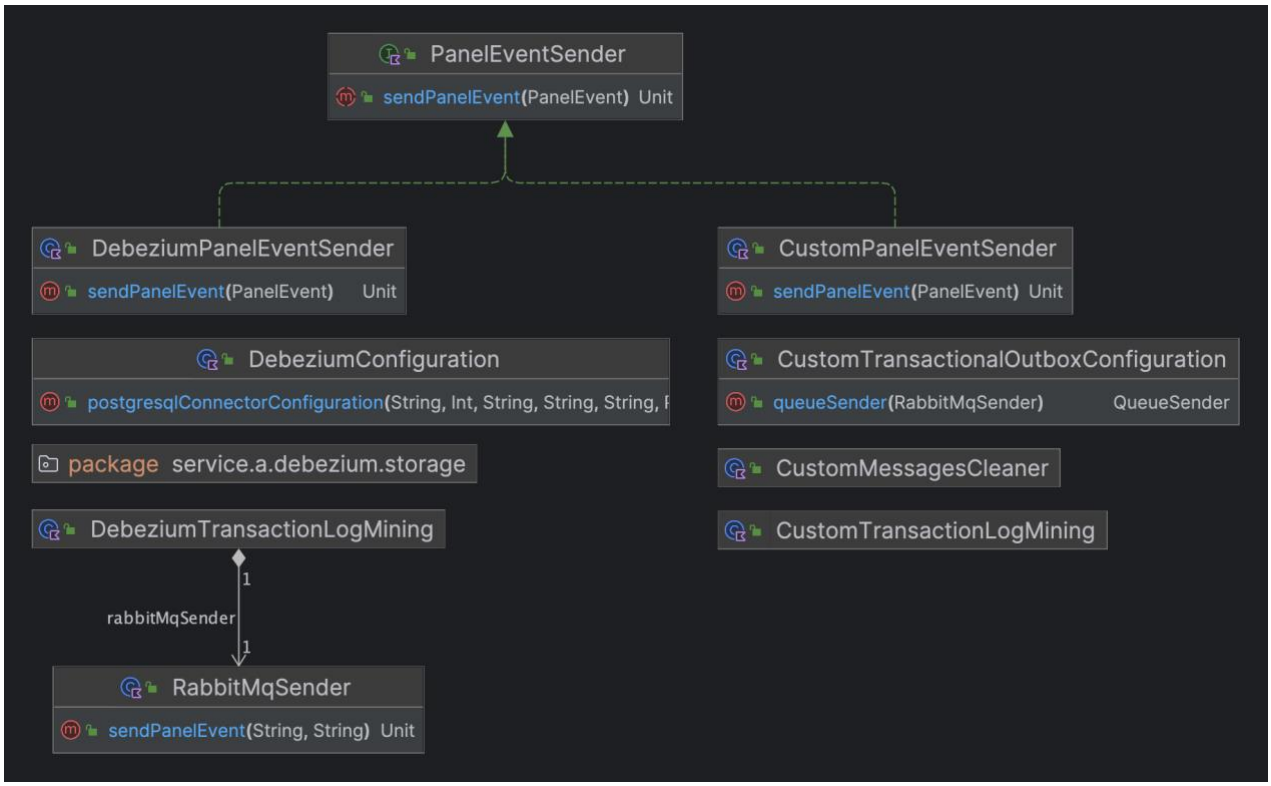

Рисунок 17

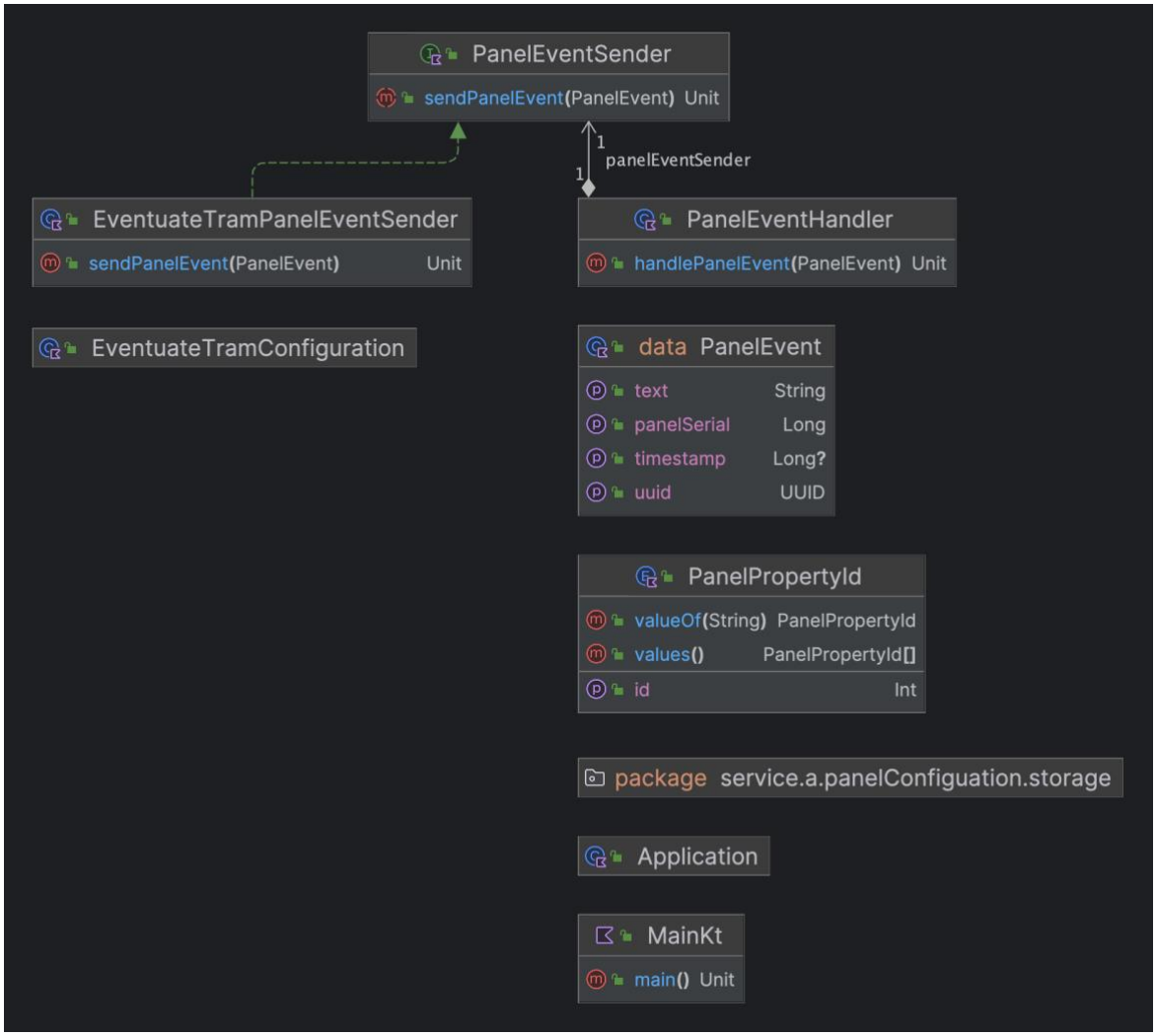

Рисунок 18

У пекеджі service.a.debezium.storage розташований код для збереження записів у таблицю panel_event_outbox.

У пекеджі sevice.a.panelConfiguration.storage наявний функціонал для збереження властивостей приладу на сервері.

4.5 Модуль service-b

Модуль service-b містить код, який реалізує сервер, що приймає повідомлення з черги повідомлень RabbitMQ. Також у цьому модулі виконаються підрахунки швидкодії реалізацій Transactional Outbox. Результати замірів виводяться в консоль.

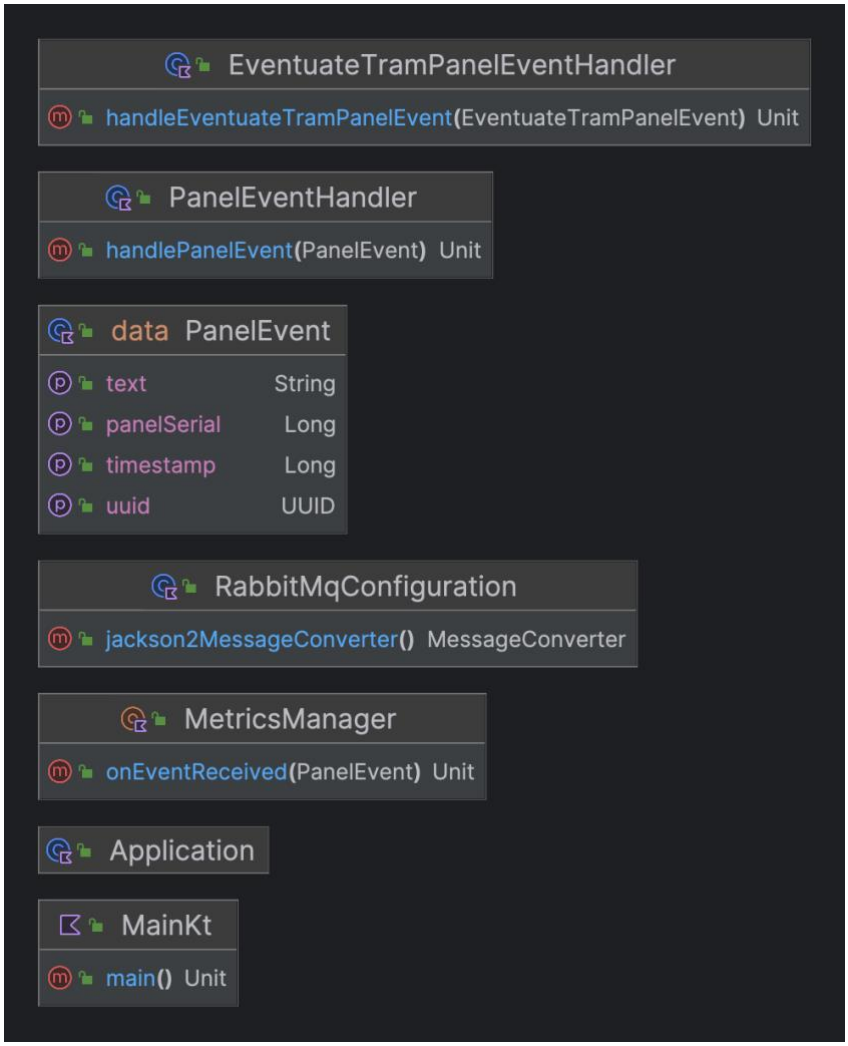

Рисунок 19

4.6 Використання черги повідомлень RabbitMQ

Щоб гарантувати високу ймовірність доставки повідомлення, було використано RabbitMQ. Чергу було зарезервовано за допомогою підходу quorum queue. Для резервування черги було розгорнуто кластер з трьох вузлів.

Для створення екземплярів черги RabbitMQ було використано інструмент docker-compose.

4.7 Результати експерименту

Під час проведення кожного з замірів було надіслано п’ятсот випадкових подій з приладу на сервер з перервою півсекунди між надсиланнями. В середньому кожен з замірів тривав більше п’яти хвилин.

Було отримано наступні результати середньої швидкості надсилання повідомлення з сервісу A на сервіс B:

- Власна бібліотека: 67.69 ms
- Debezium: 547.57 ms
- Eventuate Tram: 274.09 ms

З результатів експерименту можна побачити, що власна бібліотека працює швидше за інша два аналоги.

Існує багато варіантів пояснень такого результату. Найбільш ймовірним є те, що бібліотека, створена у рамках цього дослідження, має більшу частоту вичитування логу транзакцій і створює більше навантаження на процесор.

ВИСНОВОК

В рамках даної роботи було проаналізовано архітектурні шаблони Transactional Outbox, 2PC, Listen to Yourself, Event sourcing. Також було досліджено методи резервування черги повідомлень.

Результатом практичної частини є створення бібліотеки на мові програмування Kotlin, яка надає інструменти для реалізації шаблону Transactional Outbox. Її можна використовувати у застосунках, які написані на Kotlin або на мові, що компілюється в Java bytecode. Розроблену бібліотеку адаптовано під використання разом з фреймворком Spring Boot. Було оцінено швидкодію розробленої бібліотеки та її аналогів Debezium, Eventuate tram.

Після виконання даної роботи можна зробити висновок, що використання шаблонів Transaction Outbox, 2PC, Listen to Yourself, Event Sourcing з чергою, яка зарезервована за допомогою механізму Quorom Queue, не гарантує доставки повідомлень, проте зменшить ймовірність можливості втрати повідомлення.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Комп’ютерний кластер - https://uk.wikipedia.org/wiki/Комп%27ютерний_кластер
2. Кластер брокерів черги Oracle GlassFish - https://docs.oracle.com/cd/E19798-01/821-1798/aerdj/index.html
3. Master/Slave - https://uk.wikipedia.org/wiki/Master/slave_(технологія)
4. Опис Master/Slave в ActiveMQ - https://activemq.apache.org/jdbc-master-slave
5. Консенсусний алгоритм Raft - https://raft.github.io
6. RabbitMQ Quorum Queue - https://blog.rabbitmq.com/posts/2020/04/quorum-queues-and-why-disks-matter/
7. Change Data Capture -https://en.wikipedia.org/wiki/Change_data_capture
8. Реалізація CDC striim - https://www.striim.com/blog/change-data-capture-cdc-what-it-is-and-how-it-works/
9. Реалізація CDC qlick - https://www.qlik.com/us/change-data-capture/cdc-change-data-capture
10. Реалізація CDC hazelcast - https://hazelcast.com/glossary/change-data-capture/
11. Debezium - https://github.com/debezium/debezium
12. Eventuate tram - https://github.com/eventuate-tram/eventuate-tram-core
13. Ретроспектива розробників RabbitMQ Quorum Queue - Keynote: Quorum Queues: A Retrospective | Karl Nilsson | RabbitMQ Summit 2022
14. Побудова CDC - https://medium.com/everything-full-stack/building-a-cdc-lessons-learned-part-2-80a62d968164
15. Transactional Outbox - https://microservices.io/patterns/data/transactional-outbox.html
16. Transactional Outbox Polling Publisher - https://microservices.io/patterns/data/polling-publisher.html

17. Transactional Outbox Transaction Log Tailing - https://microservices.io/patterns/data/transaction-log-tailing.html
18. Міжпроцесорна комунікація - https://en.wikipedia.org/wiki/Inter-process_communication
19. Час відмовитися від 2PC - https://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html
20. Огляд 2PC - https://exactly-once.github.io/posts/notes-on-2pc/
21. Розподілені транзакції в Spring Boot - https://forketyfork.medium.com/distributed-xa-transactions-in-a-spring-boot-camel-application-230655b2ae89
22. Перестаньте надмірно використовувати Transactional Outbox - https://www.squer.at/en/blog/stop-overusing-the-outbox-pattern/
23. Порівняння шаблонів розподілених транзакцій - https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#how_to_choose_a_distributed_transactions_strategy
24. Як масштабувати шаблон Transactional Outbox - https://softwareengineering.stackexchange.com/questions/440390/how-to-scale-transactional-outbox-pattern-with-document-database
25. Як використовувати Debezium - https://www.up-2date.com/post/debezium
26. Transaction Log Tailing з використанням Debezium - https://medium.com/trendyol-tech/transaction-log-tailing-with-debezium-part-1-aeb968d72220
27. 2PC - https://en.wikipedia.org/wiki/Two-phase_commit_protocol
28. Event sourcing - https://www.eventstore.com/event-sourcing

ДОДАТОК А. ПРОГРАМНИЙ КОД ВИЧИТУВАННЯ ЛОГУ ТРАНЗАКЦІЙ

```
class PostgresCdcService(
    private val queueSender: QueueSender,
    private val dataSource: PGSimpleDataSource,
    private val logger: Logger? = null
) : CdcService {
    private val isPollingActive = AtomicBoolean(false)
    private val coroutineScope = CoroutineScope(Dispatchers.IO)
    private val json = Json { ignoreUnknownKeys = true }

    init {
        createReplicationSlotIfNotExist()
    }

    private fun createReplicationSlotIfNotExist() {
        val createReplicationSlotSqlScript =
javaClass.getResource("/db/createReplicationSlot.sql")?.readText() ?: throw
IllegalStateException("Can't read createReplicationSlot.sql")
        dataSource.databaseStatement(createReplicationSlotSqlScript)
    }

    override fun launchPolling() {
        if (isPollingActive.get())
            return
        coroutineScope.launch {
            isPollingActive.set(true)
            while (isPollingActive.get()) {
                try {
                    coroutineScope {
                        dataSource.usePGConnection {
                            getPostgresWalStream(this).use { walStream ->
                                readWalStream(walStream)
                            }
                        }
                    }
                } catch (e: Exception) {
                    logger?.error("Exception while polling: ${e.message}")
                }
            }
        }
    }

    private fun getPostgresWalStream(connection: PGConnection):
PGReplicationStream {
        return connection.replicationAPI
            .replicationStream()
            .logical()
            .withSlotName(ReplicationSlotName)
            .withSlotOption("write-in-chunks", "true")
            .withStatusInterval(50, TimeUnit.MILLISECONDS)
            .start()
    }

    private suspend fun readWalStream(walStream: PGReplicationStream) {
        val changesBuffer = StringBuilder()
        while (isPollingActive.get()) {
            val walEntry = walStream.readPending()
            if (walEntry == null) {
```

```
                logger?.debug("Received empty wal entry")
                confirmReceivingWalEntry(walStream)
                delay(50.milliseconds)
            } else {
                val walEntryText = walEntry.readRemainingString()
                logger?.debug("Received wal entry: $walEntryText")
                changesBuffer.append(walEntryText)
                if (isLastWalEntryOfMessage(walEntryText)) {
                    val changes = changesBuffer.toString().also {
changesBuffer.clear() }
                    processChanges(changes)
                }
                confirmReceivingWalEntry(walStream)
            }
        }
    }

    private fun confirmReceivingWalEntry(walStream: PGReplicationStream) {
        walStream.setAppliedLSN(walStream.lastReceiveLSN)
        walStream.setFlushedLSN(walStream.lastReceiveLSN)
        walStream.forceUpdateStatus()
    }

    private fun isLastWalEntryOfMessage(walEntryText: String): Boolean {
        return walEntryText == "]}"
    }

    private fun processChanges(changesTextValue: String) {
        logger?.debug("Processing changes: $changesTextValue")
        val changes =
json.decodeFromString<PostgresChangeList>(changesTextValue)
        changes.changes.forEach(::processChange)
    }

    private fun processChange(change: PostgresChange) {
        if (change.kind != InsertKind)
            return
        if (change.schema != SchemaName || change.table !=
TransactionalOutboxTableName)
            return
        val destinationColumnIndex =
change.columnNames?.indexOf(DestinationColumnName) ?: return
        if (destinationColumnIndex == -1)
            return
        val messageColumnIndex =
change.columnNames.indexOf(MessageColumnName)
        if (messageColumnIndex == -1)
            return
        val destination =
change.columnValues?.getOrNull(destinationColumnIndex)?.jsonPrimitive?.cont
ent ?: return
        val message =
change.columnValues.getOrNull(messageColumnIndex)?.jsonPrimitive?.content
?: return
        queueSender.send(destination = destination, message = message)
    }

    override fun cancel() {
        isPollingActive.set(false)
        coroutineScope.cancel()
    }

    companion object {
        private const val ReplicationSlotName = "transactional_outbox_slot"
```

```
        private const val SchemaName = "transactional_outbox"
        private const val TransactionalOutboxTableName = "messages"
        private const val DestinationColumnName = "destination"
        private const val MessageColumnName = "message"
        private const val InsertKind = "insert"
    }
}
```

ДОДАТОК В. ПРОГРАМНИЙ КОД ДЛЯ ІНТЕГРАЦІЇ СТВОРЕНОЇ БІБЛІОТЕКИ З ФРЕЙМВОРКОМ SPRING BOOT

```
@Configuration
class TransactionalOutboxPostgresWalConfiguration {
    @Bean
    fun transactionalOutboxTemplate(pgSimpleDataSource:
PGSimpleDataSource): TransactionalOutboxTemplate {
        return
TransactionalOutboxTemplate(PostgresMessageStorage(pgSimpleDataSource))
    }

    @Bean
    fun messagesCleaner(messageStorage: MessageStorage): MessagesCleaner {
        return MessagesCleaner(messageStorage,
LoggerFactory.getLogger("CustomTransactionalOutbox"))
    }

    @Bean
    fun cdcService(queueSender: QueueSender, pgSimpleDataSource:
PGSimpleDataSource): CdcService {
        return PostgresCdcService(queueSender, pgSimpleDataSource,
LoggerFactory.getLogger("CustomTransactionalOutbox"))
    }

    @Bean
    fun messageStorage(pgSimpleDataSource: PGSimpleDataSource):
MessageStorage {
        return PostgresMessageStorage(pgSimpleDataSource)
    }

    @Bean
    fun pgSimpleDataSource(
        @Value("\${spring.datasource.url}")
        databaseUrl: String,
        @Value("\${spring.datasource.username:''}")
        databaseUser: String,
        @Value("\${spring.datasource.password:''}")
        databasePassword: String
    ): PGSimpleDataSource {
        return pgDataSource(storageSettings {
            url = databaseUrl
            user = databaseUser
            password = databasePassword
        })
    }
}
```