Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Грушко Єгора Олександровича*
	(ПІБ)
академічної групи	*122М-21-2*
	(шифр)
спеціальності	*122 Комп'ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«122 Комп'ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження методів забезпечення консистентності даних та комунікації в мікросервісних системах*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*проф. Мещеряков Л.І.*			

Рецензент				

Нормоконтролер	*проф. Лактіонов І. С.*			

Дніпро
2022

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 20 22 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122М-21-2* *Грушко Єгора Олександровича*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження методів забезпечення консистентності даних та комунікаї в мікросервісній архітектурі*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 20.10.2022 р. № 2127 -л

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес збереження та обміну даними в мікросервісній архітектурі.

Предмет досліджень – програмні методи вирішення проблеми подвійного запису в мікросервісній архітектурі.

Мета НДР – створення горизонтально масштабованої та відмовостійкої архітектури мікросервісних систем, яка підтримує гарантії ACID та BASE для розподілених транзакцій.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження для вирішення проблеми подвійного запису в мікросервісній архітектурі, та стратегії вибору технологій реалізації.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає в розробці та застосуванні сучасних методів реалізації розподілених транзакцій в гетерогенних мікросервісних системах, які потребують гарантії ACID.

Практична цінність результатів полягає у тому, що отримані в ході дослідження результати можуть застосовуватися для створення високодоступної та горизонтально масштабованої мікросервісної архітектури, яка зберігає узгоджений стан.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень повинні бути подані у вигляді, який дозволяє побачити та оцінити безпосереднє використання розподілених транзакції в мікросервісній архітектурі. В результаті роботи повинен бути розроблений програмний комплекс для забезпечення високої доступності та горизонтального масштабування мікросервісних системи, та містить менеджер розподілених транзакцій.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	05.09.2022-24.09.2022
Збір, дослідження та систематизація інформації щодо методів організації даних та комунікації в гетерогенних мікросервісних системах	25.09.2022-11.10.2022
Розробка і тестування програмного комплексу для вирішення задачі забезпечення узгодженості даних та комунікації в мікросервісній архітектурі	12.10.2022-07.11.2022

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки позбавленню від ризиків, які виникають при роботі с неузгодженим станом мікросервісної системи. Підтримка горизонтального масштабування дозволить зменшити витрати на серверну інфраструктуру.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки можливості вивчення та використання розробленого програмного комплексу для створення надійних мікросервісних систем. Описані в роботі методи забезпечення узгодженого стану мають критичне значення для систем, в яких непередбачувана поведінка становить загрозу життю, безпеці, або дотримання прав людини.

Завдання видав _____ *Мещеряков Л.І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Грушко Є.О.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *05.09.2022 р.*

Термін подання кваліфікаційної роботи до ЕК *09.12.2022*

РЕФЕРАТ

Пояснювальна записка: 89 стор., 25 рис., 1 табл., 3 додатки, 51 джерело.

Об'єкт дослідження: процес збереження та обміну даними в мікросервісній архітектурі.

Предмет дослідження: програмні методи вирішення проблеми подвійного запису в мікросервісній архітектурі.

Мета магістерської роботи: створення горизонтально масштабованої та відмовостійкої архітектури мікросервісних систем, яка підтримує гарантії ACID та BASE для розподілених транзакцій.

Методи дослідження. Методи дослідження базуються на основних принципах теорії розподілених систем та теорії баз даних. Використано методи теоретичного моделювання, теоретичні основи проектування реляційних та нереляційних баз даних, теоретичні основи побудови сховищ даних, основи розподіленої обробки даних.

Новизна отриманих результатів визначається тим, що вперше розроблена оптимізація гетерогенних розподілених транзакцій, яка дозволяє ефективно поєднувати двофазні транзакції, однофазну транзакцію, та відправлення повідомлень, зберігаючи при цьому ACID та BASE гарантії.

Практична цінність результатів роботи полягає в створенні відмовостійкого та горизонтально масштабованого менеджера гетерогенних розподілених транзакцій для мікросервісної архітектури.

Область застосування. Запропонована мікросервісна архітектура може бути використана для створення відмовостійких та горизонтально масштабованих систем обробки транзакцій.

Список ключових слів: мікросервісна архітектура, розподілені транзакції, ACID, BASE, оптимізація двофазних транзакцій, відмовостійкість, горизонтальне масштабування, асинхронна комунікація

ABSTRACT

Explanatory note: 89 pages, 25 figures, 1 table, 3 applications, 51 source.

Object of research: the process of storing and exchanging data in the microservice architecture.

Subject of research: software methods for solving the dual-write problem in microservice architecture.

Purpose of Master's thesis: the creation of a horizontally scalable and fault-tolerant architecture of microservice systems that supports ACID and BASE guarantees for distributed transactions.

Research methods. Research methods are based on the basic principles of the theory of distributed systems and the theory of databases. The methods of theoretical modeling, the theoretical foundations of designing relational and non-relational databases, the theoretical foundations of building data storages, the foundations of distributed data processing are used.

Originality of research is determined by the fact that the optimization of heterogeneous distributed transactions is developed for the first time, which allows to efficiently combine two-phase transactions, one-phase transaction, and sending messages, while preserving ACID and BASE guarantees.

Practical value of the work results lies in the creation of a fault-tolerant and horizontally scalable manager of heterogeneous distributed transactions.

Scope of application. The proposed microservice architecture can be used to create fault-tolerant and horizontally scalable transaction processing systems.

Key words list: microservice architecture, distributed transactions, ACID, BASE, two-phase transactions optimization, fault tolerance, horizontal scaling, asynchronous communication

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CAP – узгодженість, доступність, стійкість до розподілення

ACID – атомарність, узгодженість, ізоляція, надійність

BASE – доступність, м’який стан, кінцева узгодженість

1PC – однофазна фіксація

2PC – двофазна фіксація

XA – стандарт X/Open XA (скорочення від «eXtended Architecture»)

CDC – запис змін даних

ЗМІСТ

ВСТУП

Актуальність дослідження. Підтримка узгодженості даних у мікросервісній архітектурі може бути серйозною проблемою. Теорема CAP стверджує [48], що в розподіленій системі неможливо одночасно підтримувати узгодженість, доступність, та стійкість до розподілення мережі (у випадках тимчасових збоїв комунікації).

Алгоритми консенсусу, такі як Raft, пом’якшують обмеження теореми CAP, дозволяючи досягти високої доступності, забезпечуючи при цьому узгодженість та стійкість до розподілення мережі [32]. Це значно покращує властивості системи, але абсолютна доступність все одно не гарантована.

Суть мікросервісної архітектури полягає в наявності мережевої комунікації між мікросервісами. Необхідність оновлення даних в кількох мікросервісах в межах однієї операції призводить до необхідності використання розподілених транзакцій між мікросервісами.

Таким чином, проблема полягає у тому, як реалізувати наскрізні бізнес-процеси, зберігаючи узгодженість між кількома мікросервісами. Ця проблема також пов’язана з питанням про те, як поширювати зміни між кількома мікросервісами, коли певні дані мають бути надлишковими.

Виходячи з теореми CAP, транзакції можуть надавати гарантії ACID, або надавати гарантії BASE. Фундаментальна відмінність між ACID та BASE полягає у тому, яку узгодженість вони забезпечують [1]. ACID гарантує сувору узгодженість, але не забезпечує доступність. В свою чергу BASE забезпечує доступність, але гарантує узгодженість лише кінцевого результату, дозволяючи системі тимчасово перебувати в неузгодженому стані.

ACID-транзакція між кількома мікросервісами вимагає або використання спільної бази даних в межах однієї транзакції, або використання двофазних транзакцій. На відміну від BASE, не всі бази даних підтримують ACID, і ще менше баз даних підтримують двофазні транзакції.

Таким чином, розподілені ACID-транзакції накладають суттєві обмеження на те, які СУБД можуть використовуватися мікросервісами.

Оскільки транзакції ACID та BASE мають різні переваги та недоліки, виникає питання, чи можливо використовувати їх разом в межах однієї мікросервісної системи, зберігаючи при цьому можливість мати різні СУБД для різних мікросервісів? Так, це можливо [16].

Більше того, це можливо навіть в межах однієї операції, якщо об’єднати транзакції в абстрактну ієрархічну графоподібну структур. Наприклад, операція може починатися з розподіленої ACID-транзакції, успішна фіксація якої буде починати розподілену BASE-транзакцію, яка в свою чергу складається з інших BASE та ACID-транзакцій. У випадку виникнення помилки на одному з етапів розподіленої транзакції, можливо ініціювати компенсаційну транзакцію, яка відкотить усі транзакції в зворотньому порядку.

Мета дослідження полягає у створення горизонтально масштабованої та відмовостійкої архітектури мікросервісних систем, яка підтримує гарантії ACID та BASE для розподілених транзакцій. Відповідно до мети у кваліфікаційній роботі необхідно вирішити наступні **завдання**:

1. Виконати огляд існуючих методів організації даних та комунікацій в мікросервісній архітектурі, описати їх переваги та недоліки.

2. Дослідити та порівняти методи вирішення проблеми подвійного запису в мікросервісній архітектурі.

3. Створити гнучку систему комунікації для мікросервісної архітектури, яка дозволить використовувати та поєднувати різні методи організації даних та вирішення проблеми подвійного запису.

Об'єкт дослідження: процес збереження та обміну даними в мікросервісній архітектурі.

Предмет дослідження: програмні методи вирішення проблеми подвійного запису в мікросервісній архітектурі.

Методи дослідження. Методи дослідження базуються на основних принципах теорії розподілених систем та теорії баз даних. Використано методи

теоретичного моделювання, теоретичні основи проектування реляційних та нереляційних баз даних, теоретичні основи побудови сховищ даних, основи розподіленої обробки даних.

Новизна отриманих результатів визначається тим, що вперше розроблена оптимізація гетерогенних розподілених транзакцій, яка дозволяє ефективно поєднувати двофазні транзакції, однофазну транзакцію, та відправлення повідомлень, зберігаючи при цьому ACID та BASE гарантії.

Практична цінність результатів роботи полягає в створенні відмовостійкого та горизонтально масштабованого менеджера гетерогенних розподілених транзакцій.

Особистий внесок автора: використання Apache Kafka для забезпечення відмовостійкої та швидкої мікросервісної комунікації, що дозволило створити універсальне середовище для виконання розподілених транзакцій; забезпечення семантики ACID для розподілених транзакцій, які охоплюють Apache Kafka, MongoDB та PostgreSQL, використовуючи при цьому двофазні транзакції лише в PostgreSQL.

Структура і обсяг роботи. Робота складається з вступу, трьох розділів та висновків. Містить 89 сторінок, в тому числі 58 сторінок тексту основної частини з 25 рисунками, списку використаних джерел з 51 найменуваннями на 7 сторінках, 3 додатки на 11 сторінках.

РОЗДІЛ 1
ОГЛЯД ЗАГАЛЬНИХ ПРИНЦИПІВ ОРГАНІЗАЦІЇ ДАНИХ ТА КОМУНІКАЦІЙ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

1.1. CAP-теорема та організація даних в мікросервісній архітектурі

Оглядаються наступні три властивості розподіленої системи, на які посилається теорема CAP:

– узгодженість (C) означає, що всі клієнти бачать однакові дані одночасно, незалежно від способу їхнього запиту;

– доступність (A) означає, що всі компоненти системи повертатимуть коректну відповідь, навіть якщо вони не працюють. Це особливо важливо, якщо користувачі системи мають низьку стійкість до збоїв;

– стійкість до розподілення (P) означає, що система працюватиме навіть під час збою мережі, що призводить до втрати або затримки повідомлень між компонентами. Це стосується систем, які інтегруються з великою кількістю розподілених незалежних компонентів.

На жаль, неможливо мати всі ці властивості одночасно. Коли справа доходить до розподілених систем, таких як ті, що створюються за мікросервісною архітектурою, для цього існує математичний доказ [11].

У своїй основі це говорить про те, що в розподіленій системі у є три речі, які можливо порівнювати одна з одною: узгодженість, доступність і стійкість до розподілення. Зокрема, теорема CAP говорить, що можливо зберігати лише дві властивості при відмові.

Наприклад, існує мікросервіс Склад, який розгорнутий в двох окремих центрах обробки даних. В кожному центрі обробки даних є своя окрема база даних, і ці дві бази даних спілкуються одна з одною, щоб спробувати синхронізувати бази даних між собою (рис. 1.1).

Читання та запис виконуються через вузол локальної бази даних, а реплікація використовується для синхронізації даних між вузлами.

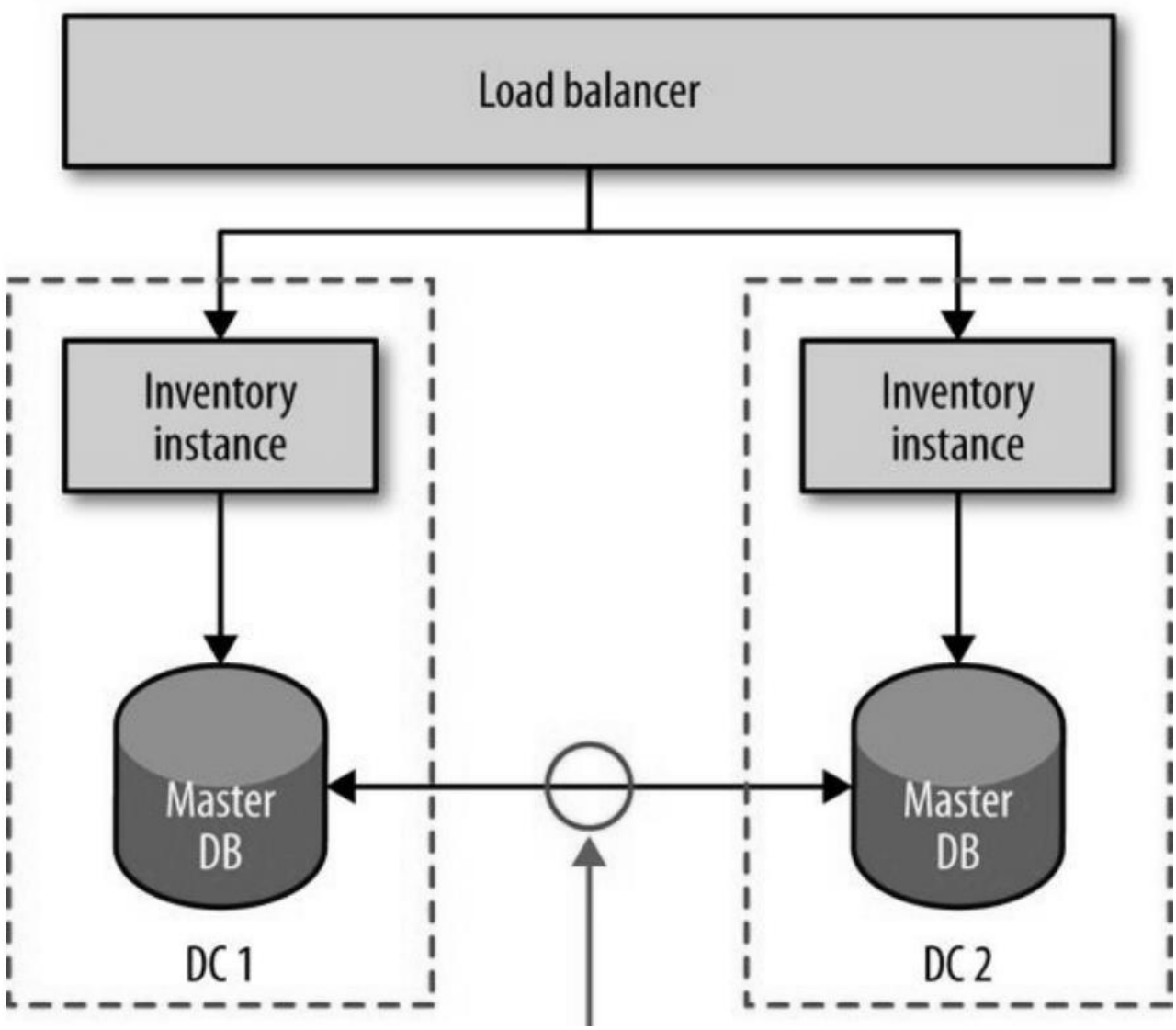

Рис. 1.1. Реплікація даними між двома вузлами бази даних

Важливо подумати про те, що відбувається, коли щось не вдається, наприклад мережеве з'єднання між двома центрами обробки даних, перестає працювати. Синхронізація на цьому етапі не вдається. Записи, зроблені до основної бази даних у DC1, не поширюватимуться на DC2 і навпаки. Більшість баз даних також підтримують техніку черги, щоб гарантувати можливість відновитися, але що відбувається тим часом?

Припускається, що мікросервіс Склад не вимикається поновністю. Якщо внести зміни до бази даних в DC1, база даних у DC2 не побачить їх. Це означає, що будь-які запити, зроблені до вузла мікросервісу в DC2, бачать потенційно застарілі дані. Іншими словами, система все ще доступна в тому сенсі, що обидва

вузли здатні обслуговувати запити, і підтримується роботу системи, незважаючи на розподілення, але втрачена послідовність.

Це часто називають AP-системою.

Під час цього розподілення, якщо продовжувати приймати записи, приймається той факт, що в якийсь момент у майбутньому їх потрібно буде повторно синхронізувати. Чим довше триває розподілення, тим складнішою може стати повторна синхронізація.

Реальність така, що навіть якщо між вузлами бази даних немає збою мережі, реплікація даних не відбувається миттєво. Системи, які готові відмовитися від узгодженості, щоб зберегти стійкість до розподілення і доступність, вважаються зрештою узгодженими, тобто очікується, що в якийсь момент у майбутньому всі вузли побачать оновлені дані, але це станеться не відразу, тому існує можливість того, що користувачі побачать старі дані.

Що станеться, якщо потрібно зберегти послідовність і для цього можливо відмовитися від чогось іншого? Що ж, щоб зберегти узгодженість, кожен вузол бази даних повинен знати, що копія даних, яку він має, така сама, як і в іншого вузла бази даних. При розподілення, якщо вузли бази даних не можуть спілкуватися один з одним, вони не можуть координуватися для забезпечення узгодженості. Неможливо гарантувати послідовність, тому єдиний вихід — відмовитися відповідати на запит. Іншими словами, пожертвувати доступністю. Тоді система послідовна та стійка до розподілення.

Таку систему також називають CP-системою.

У цьому режимі мікросервісу потрібно буде вимкнути частину функціональності, доки комунікацію не буде відновлено, а вузли бази даних можна буде повторно синхронізувати.

Узгодженість між кількома вузлами справді складна. Є не так багато речей (можливо, жодної) складніших у розподілених системах.

Наприклад, треба прочитати запис із вузла локальної бази даних. Як дізнатися, що він оновлений? Потрібно піти і запитати інший вузол. Але також треба попросити цей вузол бази даних не дозволяти його оновлювати, поки

читання завершується; іншими словами, потрібно ініціювати транзакційне читання на кількох вузлах бази даних, щоб забезпечити узгодженість. Але загалом транзакції не використовуються для читання, оскільки транзакції виконуються повільно. Вони потребують блокування. Зчитування може заблокувати всю систему. Усі узгоджені системи потребують певного рівня блокування, щоб виконувати свою роботу.

Розподілені системи повинні бути готовими до збою. Використовуючи транзакційне читання через набір послідовних вузлів, віддалений вузол блокує запис, поки ініціюється читання. Коли читання завершується, віддалений вузол має зняти блокування, але тепер з ним неможливо спілкуватися. Що тепер відбувається? Блокування справді важко встановити правильно навіть у системі з одним сервером, і значно важче правильно реалізувати в розподіленій системі.

Основна причина, чому блокування є складним, полягає в цій проблемі із забезпеченням узгодженості між кількома вузлами.

Чому б не розробити CA-систему? Ну, якщо можливо пожертвувати стійкістю до розподілення? Якщо система не має доступу до тих чи інших компонентів, вона не може працювати у мережі. Фактично, це має бути єдиний процес, що працює локально. CA-системи не існують у розподілених системах.

Що правильно, AP чи CP? Що ж, насправді не існує однозначної відповіді на це питання. Коли справа доходить до мікросервісів, теорема CAP здається нерозв'язною проблемою. Яку з цих трьох речей можна дозволити відкинути? Однак головне те, що вибору немає.

За всім цим CAP-теорема є дистиляцією дуже логічного набору міркувань. Використається кілька прикладів, які допомогають це зрозуміти.

Відомо, що AP-системи масштабуються легше та їх простіше створити, і відомо, що CP-система вимагатиме більше роботи через проблеми з підтримкою розподіленої узгодженості. Але потрібно розуміти вплив на бізнес-логіку.

Для мікросервісу Склад, якщо запис застарів на п'ять хвилин, це нормально? Якщо відповідь ствердна, тоді це може бути AP-система. Але як щодо балансу, який зберігається для клієнта в банку? Чи може він бути

застарілим? Не знаючи контексту, у якому використовується операція, неможливо знати, що робити правильно.

Також важливо те, що мікросервісній системі в цілому не обов'язково бути AP або CP. Наприклад, мікросервіс Каталог може бути AP, оскільки може деякий час містити записи, яких вже немає в мікросервісі Склад. Але мікросервіс Склад повинен бути CP, оскільки неправильно продавати клієнту те, чого немає в наявності, а потім вибачатися.

Але окремим мікросервісам також не обов'язково бути CP або AP. Наприклад, існує мікросервіс балансу балів, де зберігаються записи про те, скільки балів лояльності накопичили клієнти. Не проблема, якщо баланс, який показується для клієнта, застарілий, але коли мова йде про оновлення балансу, потрібно, щоб він був послідовний, щоб клієнти не використовували більше балів, ніж вони мають на балансі.

Це CP-система, або AP, чи і те, і інше? Дійсно, можливо зсунути компроміси навколо CAP до можливостей окремих мікросервісів.

Можливо знайти ще більш деталізований компроміс. Наприклад, розробити різні компроміси для різних бізнес-операцій. Отже, якщо потрібна сувора узгодженість, можна виконати читання, яке блокується, доки всі репліки не дадуть відповіді, підтверджуючи узгодженість значення, або поки не відповість певний кворум реплік [28], або навіть лише один вузол. Очевидно, чим вище рівень узгодженості між репліками, тим довше буде блокування.

Але якщо достатньо лише простого кворума вузлів, з'являється можливість бути менш вразливим до недоступності однієї репліки [31].

Тобто, можливо створити систему, де частина функціоналу має властивості CP, а інша частина має властивості AP [15].

Багато чого з того, що будується, є лише відображенням реального світу, де діють закони спеціальної теорії відносності, які розповсюджуються також і на мікросервісну систему, оскільки вона розподілена у просторі [43].

Забезпечити сувору узгодженість операцій можливо лише при використанні умовної єдиної точки у просторі для взаємодії з системою [34], і тому CAP-теорема точно відображає те, як працює реальний світ.

Мікросервіс Склад відображає реальні фізичні предмети. Наприклад, у ведеться підрахунок кількості альбомів. На початку дня було 100 копій альбомів. Один був проданий. Залишилося 99 альбомів.

Що трапиться, якщо під час надсилання замовлення хтось валить копію альбому на підлогу, на неї наступають і розбивають? Що тепер відбувається? Система говорить про 99, але на полиці є 98 копій.

Якщо натомість зробити AP-мікросервіс, і час від часу зв'язуватися з користувачем й повідомляти йому, що одного з товарів насправді немає в наявності? Таку систему було б набагато простіше побудувати та масштабувати.

Якими б узгодженими не були системи самі по собі, вони не можуть знати все, що відбувається, особливо коли ведуться записи про реальний світ.

Окрім складності побудови, CP-системи все одно не можуть вирішити всі проблеми. Саме тому AP-системи поводять себе правильно у багатьох ситуаціях. Вони здатні забезпечити причинно-наслідкову узгодженість операцій. Забезпечення такої узгодженості не потребує впровадження єдиної точки, яка контролює послідовність операції для всієї системи [25].

1.1.1. Шаблон «Спільна база даних» та теорема PACELC

Мікросервіси повинні спілкуватися один з одним [18]. Спілкування між ними завжди пов'язане з отриманням або оновленням даних, якими володіє інший сервіс. Що, якщо мікросервіс отримає прямий доступ до всіх даних, які йому потрібні? Це найпростіший спосіб для мікросервісів спілкуватися один з одним, і цей шаблон називається «Спільна база даних» (рис 1.2).

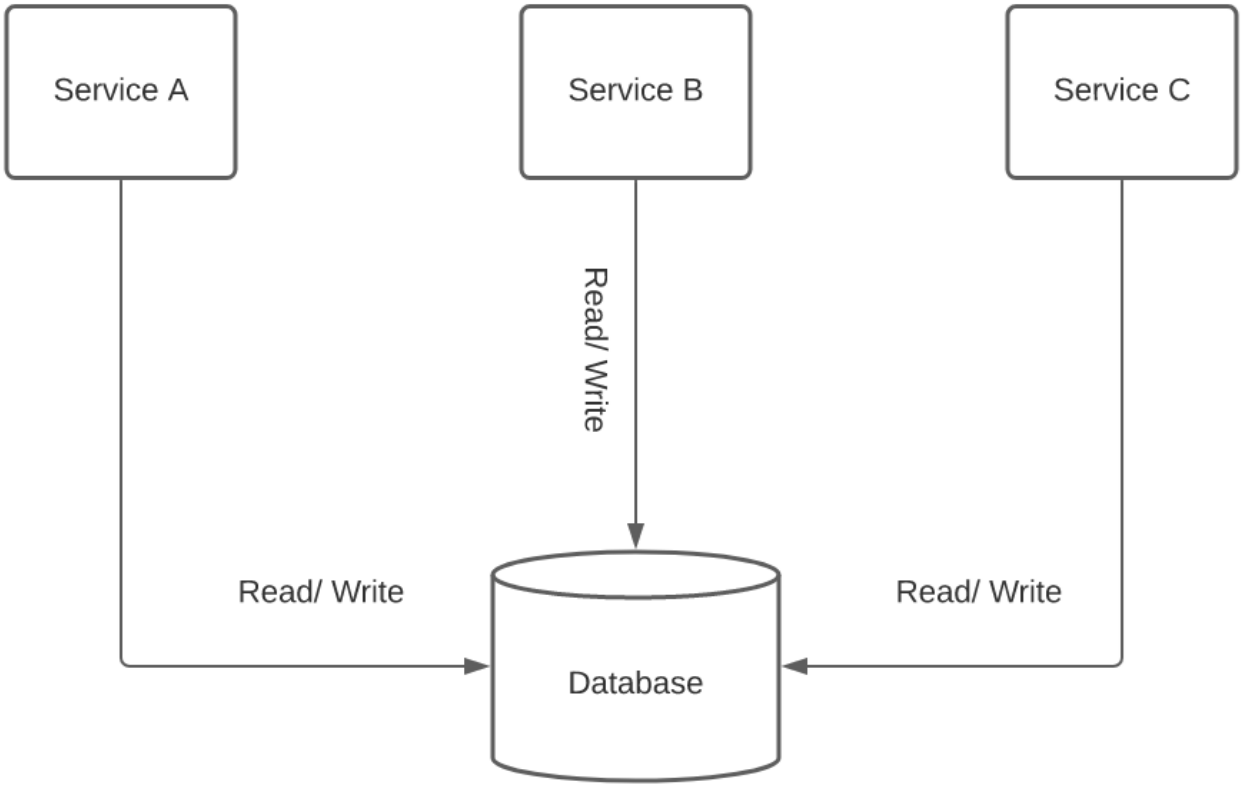

Рис. 1.2. Загальна схема мікросервісної архітектури, де використовується спільна база даних

Основна ідея тут полягає в тому, щоб будь-хто, кому потрібні дані з мікросервісу або хто хоче щось оновити, напряму спілкувався з його базою даних — посередники не потрібні.

Наприклад: створюючи багатокористувацьку систему для ведення блогів, припускається, що є мікросервіс Блог, який керує всією інформацією, пов’язаною з блогами, і є мікросервіс Аналітика, який піклується про всю аналітику, як-от «Подобається», «Поширення», «Перегляди» тощо.

Сервіс аналітики асинхронно оновлює інформацію, наприклад, про кількість переглядів, безпосередньо в базі даних блогу. Цього можна легко досягти шляхом спільного використання бази даних між мікросервісами.

При застосуванні шаблону «Спільна база даних» дуже важливо правильно обрати СУБД з оглядом на потреби всієї мікросервісної системи. Допомогти в цьому може теорема PACELC, що базується на теоремі CAP, та декларує, що в разі розподілення мережі (P) у розподільній комп’ютерній системі необхідно

вибирати між доступністю (A) і узгодженістю (C), але в будь-якому випадку, навіть якщо система працює нормально у відсутності розподілення (E), потрібно обирати між затримками (L) та узгодженістю (C).

Обидві теореми описують, які обмеження та компроміси мають розподілені бази даних щодо узгодженості, доступності та стійкості до мережевого розподілення. Проте PACELC йде далі та заявляє, що існує додатковий компроміс: між затримкою та узгодженістю, навіть за відсутності мережевого розподілення, таким чином забезпечуючи більш повне зображення потенційних компромісів узгодженості для розподілених систем [42].

Вимога високої доступності означає, що система повинна реплікувати дані до резервних баз даних. Як тільки розподілена система реплікує дані, виникає компроміс між узгодженістю та затримкою.

Використовуючи PACELC для визначення СУБД, можливо класифікувати їх відповідно до того, на які компроміси вони йдуть:

– З точки зору PACELC, реляційні системи керування базами даних і бази даних NoSQL, які реалізують ACID, призначені для забезпечення узгодженості, класифікуючи їх як PC/EC. Типові бізнес-додатки, як-от програми для роботи з персоналом і системи продажу квитків, імовірно, використовуватимуть цю модель, особливо якщо кілька користувачів використовують різні екземпляри компонентів. Гарним прикладом цього є база даних Google Bigtable;

– Такі СУБД як MongoDB, вписуються в модель PA/EC, яка найкраще підходить для таких речей, як системи електронної комерції, яким потрібна висока доступність навіть під час збоїв мережі чи компонентів системи;

– Системи реального часу, такі як системи IoT, вписуються в модель PC/EL, яку пропонують СУБД, такі як PNUTS. Це стосується будь-якої системи, де узгодженість реплікацій є критичною;

– СУБД на основі моделі PA/EL, такі як Dynamo та Cassandra, найкраще підходять для систем реального часу, які не потребують частих оновлень, оскільки узгодженість буде меншою проблемою.

Використання спільної бази даних між мікросервісами має свої переваги, зокрема це найпростіший спосіб інтеграції мікросервісів між собою, що само по собі скорочує час розробки системи, а також зменшує затримки в системі, оскільки в такій системі відсутній будь-який посередник між мікросервісами, окрім самої бази даних. Але цей підхід також має і власні недоліки.

Зовнішні сторони знають внутрішні деталі. Якщо мікросервіси мають спільну базу даних, зовнішня сторона (мікросервіс Аналітика) дізнається внутрішні деталі мікросервісу Блог; наприклад: практика видалення, схема тощо. Це призводить до дуже тісного зв’язку між мікросервісами; що потім обмежує ремонтопридатність і продуктивність системи. Наприклад, коли мікросервіс Блог змінює схему, мікросервіс Аналітика має бути проінформована про цю зміну.

Спільний доступ до бази даних означає спільний доступ до логіки. Щоб обчислити деяку інформацію, потрібно зробити запит до набору таблиць; і припускається, ця інформація потрібна для мікросервісів Блог, Аналітика та Рекомендації. Бізнес-логіка для обчислення інформації має бути відтворена в усіх 3 мікросервісах. Будь-яку зміну потрібно вносити в усі компоненти.

Існує ризик того, що один з мікросервісів може пошкодити або видалити деякі дані, оскільки база даних спільно використовується між мікросервісами.

Один мікросервіс, який виконує дорогі запити до бази даних, зловживає спільною базою даних, що впливає на продуктивність інших мікросервісів, які спільно використовують ту саму базу даних.

Шаблон «Спільна база даних» корисний, коли розробка повинна бути швидкою. Хоча це не найкраща практика, спільне використання бази даних значно скорочує зусилля щодо розробки.

Хоча більшість людей вважає, що це неправильний спосіб побудови мікросервісної архітектури, неправильно повинні відкидати його повністю.

1.1.2. Шаблон «Одна база даних на сервіс»

Використовуючи шаблон «Одна база даних на сервіс» (рис. 1.3), можливо обирати найбільш відповідні сховища даних (наприклад, реляційні або нереляційні) для окремого мікросервісу та його бізнес-вимог [24].

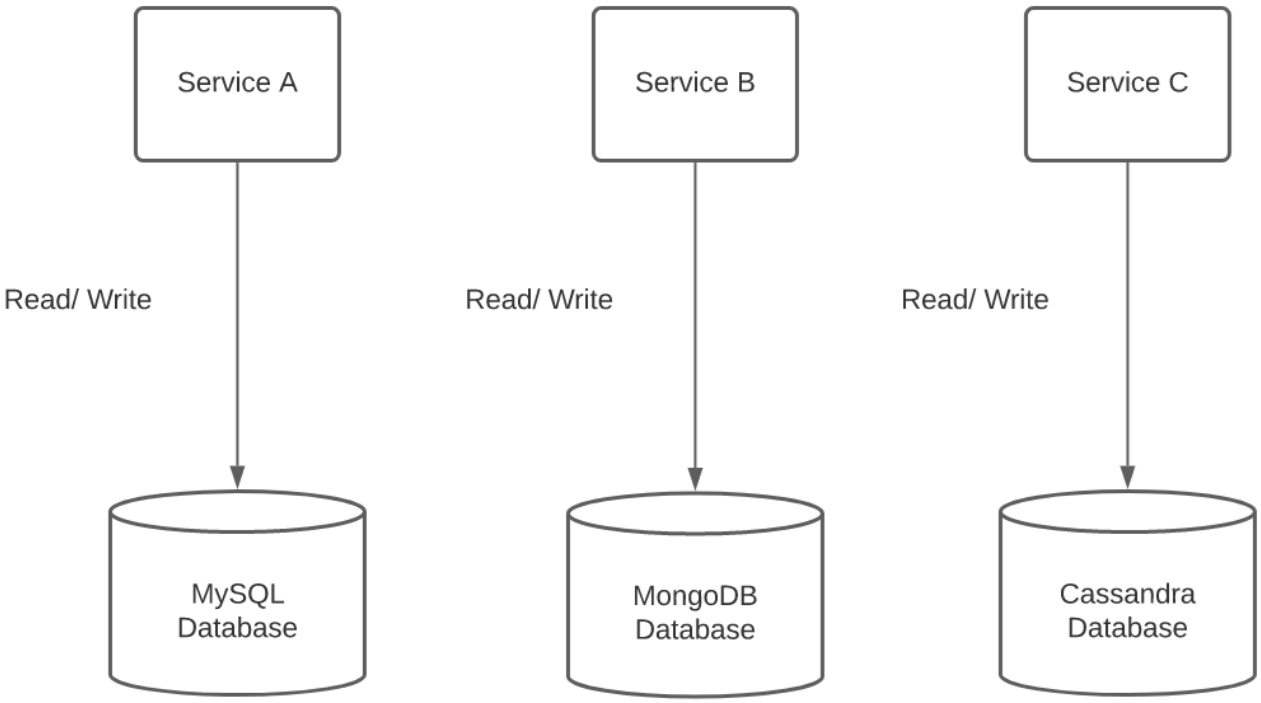

Рис. 1.3. Загальна схема мікросервісної архітектури, де застосовується шаблон «Одна база даних на сервіс»

Незважаючи на те, що цей підхід сприяє незалежності мікросервісів, слабкому зв’язку та ізоляції відмов в архітектурі, важливо враховувати компроміси: а саме рівень складності управління та вимоги до ресурсів, які може накласти цей тип структури [29].

По-перше, об’єднання великих даних через мережу займає багато часу та зусиль. Також це збільшує ризик великої надлишковості даних, що виснажує ресурси, особливо якщо не вдається правильно визначити обмежені контексти.

Наявність безлічі різноманітних баз даних мікросервісів також ускладнить послідовну підтримку синхронізованих станів системи [19].

Найпряміший спосіб уникнути підводних каменів шаблону — надати прямий доступ до бази даних в якості API іншим мікросервісам, так вони зможуть отримувати необхідні дані напряму.

Наприклад, використання бази даних напряму, робить процес об’єднання непересічних структур даних набагато простішим.

Поки всі необхідні таблиці знаходяться в одній базі даних, розподілені транзакції можуть безпечно виконуватися завдяки використанню атомарних гарантій та примітивів синхронізації.

Звичайно, подібне розповсюдження прямого доступу до бази даних має свої недоліки. Оскільки схеми бази даних розвиваються з часом, усі мікросервіси, які залежать від цієї бази даних, потребуватимуть оновлення якщо буде оновлена схема таблиці в базі даних.

Інша незручність полягає в тому, що не всі мікросервіси, які спільно використовують одну базу даних, обов’язково використовуватимуть цю базу даних однаково, а налаштування бази даних для потреб одного мікросервісу може перешкодити іншому виконувати свою роботу.

У великомасштабних програмних системах одна база даних може бути не в змозі забезпечити рівень обсягу даних і пропускну здатність, необхідні для кожного мікросервісу, що може змусити IT-спеціалістів несподівано розділити базу даних. База даних також постраждає при роботі з великими обсягами запису, а ущільнення таблиць сегментів може призвести до значного зниження продуктивності. Це ситуація, яка нагадує класичну проблему шумного сусіда, яка зазвичай пов’язана з усіма типами спільних ресурсів системи.

Виходячи з цього, в системах високої складності, як правило, існує посередник, API, що надає доступ до даних іншим мікросервісам за допомогою будь-яких синхронних та асинхронних методів комунікації, залишаючи базу даних приватною для окремого мікросервісу.

1.2. Комунікації в мікросервісній архітектурі

Слід обговорити ще одне з найважливіших рішень, які потрібно прийняти щодо того, як будувати комунікації між мікросервісами [17].

Мікросервісна система, як правило, складається з кількох обмежених контекстів [4], і в кожному з них знаходяться дані, які не мають бути передані ззовні, а також дані, які використовуються зовні в інших обмежених контекстах. Кожен обмежений контекст має явний інтерфейс, де вирішується, якими моделями ділитися з іншими контекстами.

Зміни, які впроваджуються в систему, часто стосуються змін бізнес-логіки. Змінюються функціональні можливості, які доступні клієнтам. Якщо система розбита на обмежені контексти, які представляють домен, зміни, які треба впровадити, швидше за все, будуть ізольовані в межах одного мікросервісу. Це зменшує кількість місць, які потрібні для внесення змін, і дозволяє швидко впроваджувати ці зміни [21].

Якщо потрібно отримати інформацію з обмеженого контексту або зробити запити до функціональності в обмеженому контексті, важливо подумати про комунікацію між мікросервісами.

Чи комунікація має бути синхронною чи асинхронною? Це принциповий вибір неминуче спрямовує нас до певних деталей реалізації.

При синхронному зв'язку виклик здійснюється на віддалений сервер, який блокується до тих пір, поки операція не буде завершена.

Завдяки асинхронному зв’язку клієнт не чекає завершення операції, і може навіть не хвилюватися, чи операція взагалі була завершена.

Синхронні комунікації простіше. Є можливість дізнатися, коли що завершено, успішно чи ні. Асинхронний зв’язок може бути дуже корисним для для виконання тривалих операцій, де залишати з’єднання відкритим протягом тривалого часу між клієнтом та сервером є непрактичним [3].

Ці два різні способи спілкування можуть створювати два різні стилі комунікації: запит/відповідь або на основі подій. За допомогою запиту/відповіді клієнт ініціює запит і чекає на відповідь.

Модель запиту/відповіді чітко підходить до синхронної комунікації, але може працювати і для асинхронного зв’язку. Можливо розпочати операцію та зареєструвати зворотний виклик, попросивши сервер повідомити клієнта, коли операцію буде завершено.

Комунікація на основі подій працює навпаки. Замість того, щоб клієнт ініціював запити, щоб щось було зроблено, клієнт натомість говорить, що щось сталося, і очікує від інших сторін, що вони знають, що робити.

Подієві системи за їхньої природи асинхронні. Кажучи інакше, розум розподілений рівномірно — тобто бізнес−логіка не зосереджена в головному мозку, а натомість більш рівномірно розподіляється до різних співавторів.

Комунікація на основі подій також сильно відокремлена [51]. Клієнт, який створює подію не має жодного способу дізнатися, хто або що відреагує на неї. Це також означає, що можливо додавати нових підписників на ці події без необхідності знати про це клієнту.

Отже, чи існують якісь інші чинники, які можуть підштовхнути до вибору одного стилю замість іншого?

Один важливий фактором, який слід враховувати, є те, як обробляються процеси, які охоплюють декілька мікросервісів, і чи можуть оброблятися довго?

1.2.1. Оркестрація та хореографія

При моделюванні все більш і більш складної логіки, доводиться мати справу з проблемою управління бізнес-процесами, які охоплюють межі декількох окремих мікросервісів. Наприклад (рис. 1.4):

1. Для клієнта створюється новий запис у банку балів лояльності;
2. Надсилається вітальне повідомлення поштовою службою;
3. Надсилається вітальний лист електронною поштою.

Коли справа доходить до реальної реалізації цього процесу, існує два стилі архітектури. Оркестрація покладається на центральний мозок, щоб керувати процесом. Це дуже схоже на диригента в оркестрі.

Хореографія покладається на повідомлення про події в системі, де кожен компонент знає про частину своєї роботи. Як танцюристи, які знаходять свій шлях реагуючи на оточуючих у балеті.

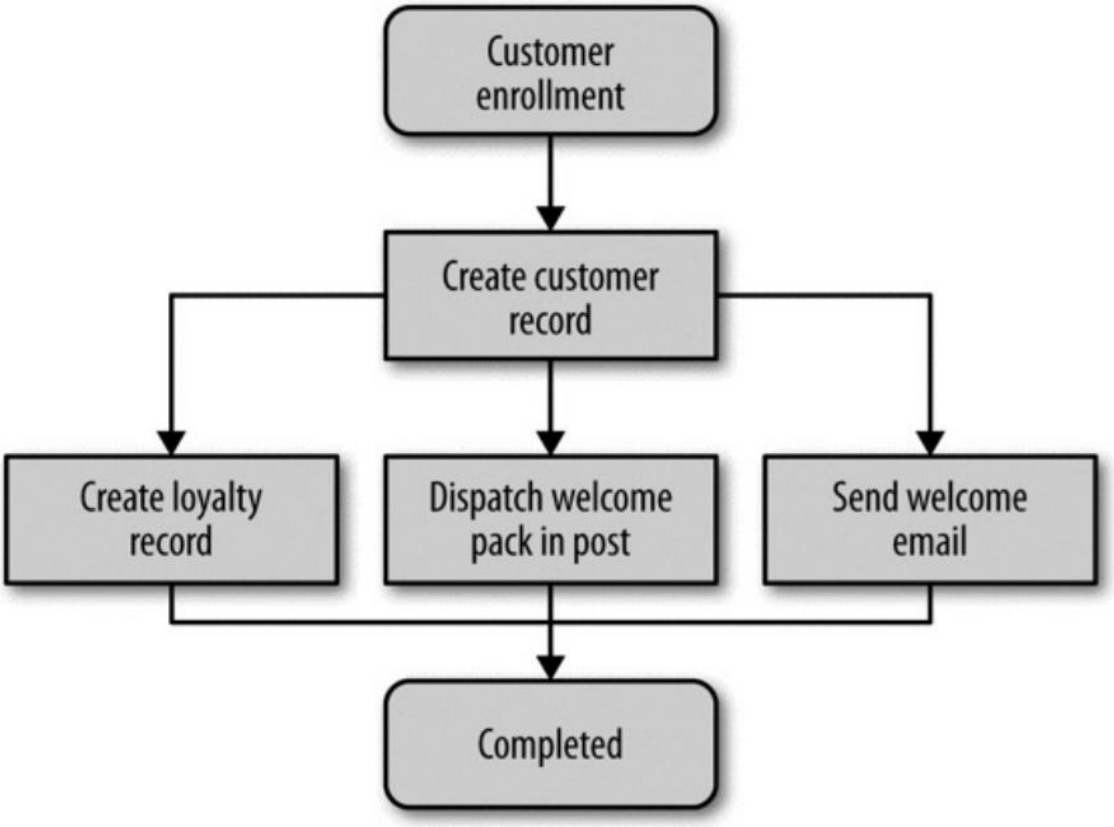

Рис. 1.4. Приклад бізнес-процесу «Створення клієнта»

Виникає питання, як виглядатиме оркестрація для цього процесу? Тут, мабуть, найпростішим було б зробити так, щоб мікросервіс обслуговування клієнтів виступав в якості центрального мозку.

Під час створення мікросервіс обслуговування клієнтів спілкується з мікросервісом балів лояльності, електронною поштою та поштовою службою через серію запитів/відповідей. Мікросервіс обслуговування клієнтів сам може відстежувати, де знаходиться клієнт у цьому процесі. Таким чином, можливо перевірити, чи обліковий запис клієнта налаштовано, або електронний лист надіслано, або повідомлення доставлено. Припускаючи, що використовується

синхронний метод комунікації через запит/відповідь, можливо навіть знати, чи спрацював кожен окремий етап.

Недоліком підходу оркестрації (рис. 1.5) є те, що мікросервіс обслуговування клієнтів може стати надто центральним керівним органом.

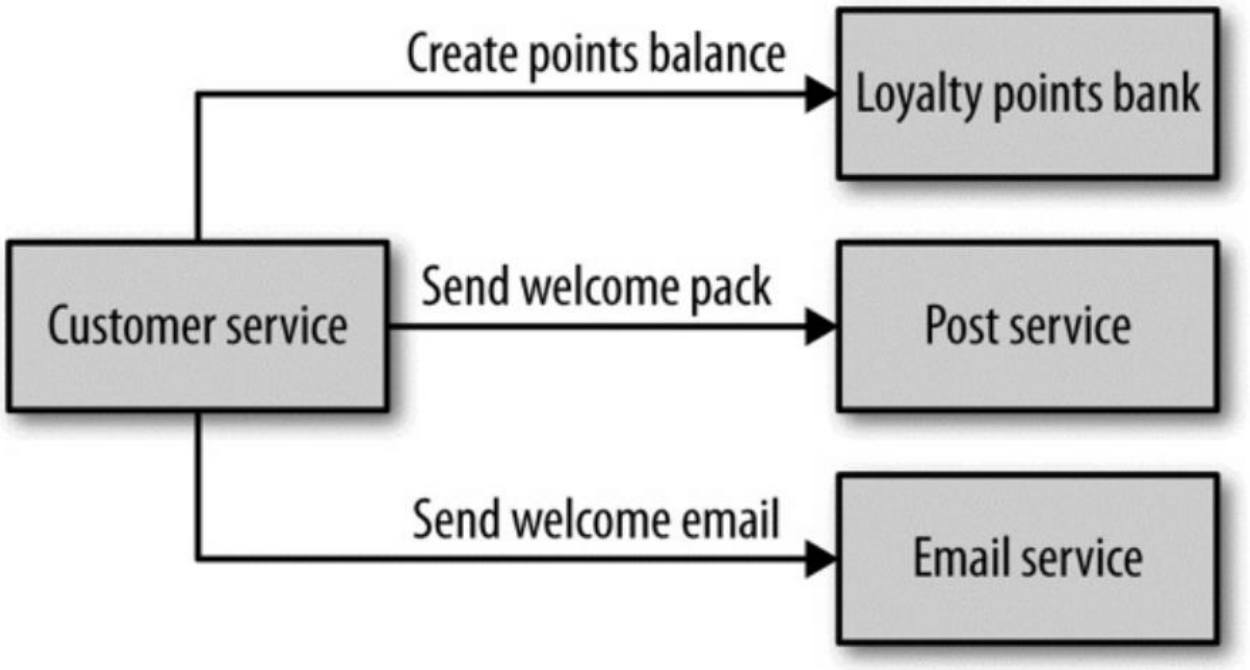

Рис. 1.5. Керування створенням клієнта за допомогою оркестрації

Цей підхід може призводити до невеликої кількості розумних «божественних» мікросервісів, які вказують іншим анемічним мікросервісам, що робити, і це призводить до нерівномірного розподілення логіки.

За допомогою хореографічного підходу (рис. 1.6) можливо натомість просто змусити мікросервіс клієнтів видавати подію асинхронним способом, кажучи, що Клієнт створено. Служба електронної пошти, поштова служба та мікросервіс балів лояльності просто підписуються на ці події та реагують відповідним чином. Цей підхід є значно більш роз’єднаним. Якщо якомусь іншому мікросервісу необхідно дізнатися про створення клієнта, просто потрібно підписатися на події та виконувати роботу, коли це необхідно.

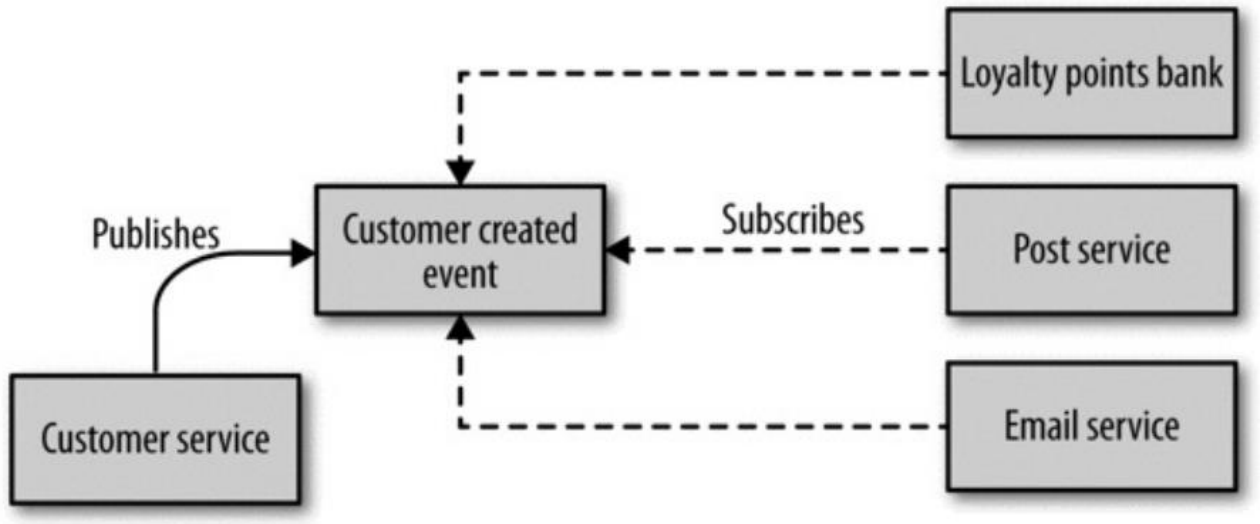

Рис. 1.6. Керування створенням клієнта за допомогою хореографії

Недоліком є те, що бізнес-процес, тепер лише неявно відображається в системі. Це означає, що потрібна додаткова робота, щоб мати можливості контролювати та відстежувати, чи відбуваються правильні речі. Наприклад, якщо в мікросервісу балів лояльності виникла помилка та з якоїсь причини не був чинним образом налаштований обліковий запис.

Один із підходів, полягає в тому, щоб побудувати систему моніторингу, яка явно відповідає вигляду бізнес-процесу, та відстежує, що робить кожен мікросервіс як незалежні об’єкти, дозволяючи бачити дивні винятки, відображені на більш чіткий потік процесу [12].

Загалом, системи, які більше прагнуть до хореографічного підходу, мають більш слабкий зв’язок, є більш гнучкими та піддаються змінам. Однак потрібно буде зробити додаткову роботу для моніторингу та відстеження процесів, що розподілені в декількох мікросервісах.

Більшість сильно зв’язаних реалізацій надзвичайно крихкі, та мають більш високу вартість змін. Варто надавати перевагу хореографічній системі, де кожен мікросервіс достатньо розумний, щоб зрозуміти свою роль.

Тут є досить багато факторів, які слід розкрити.

Синхронні виклики простіші, і можливо відразу дізнатися, чи все спрацювало. Якщо подобається семантика запиту/відповіді, або довотиться мати

справу з довготривалими процесами, можливо просто ініціювати асинхронні запити та чекати зворотних викликів.

З іншого боку, асинхронна комунікації через події допомагає використовувати хореографічний підхід, який може давати значно більше розподілення — це саме те, що потрібно, щоб гарантувати те, що окремі мікросервіси можна буде випускати незалежно.

Звичайно, також можливо поєднувати ці підходи. Деякі технології більш природно впишуться в той чи інший стиль.

1.2.2. Виклик віддалених процедур (RPC)

Основна ідея RPC полягає в тому, щоб приховати складність віддаленого виклику. Однак багато реалізацій RPC приховують занадто. Прагнення в деяких формах RPC зробити віддалені виклики методів схожими на локальні виклики методів приховує той факт, що ці дві речі дуже різні.

Можливо здійснювати велику кількість локальних викликів, не турбуючись про продуктивність. Проте з RPC вартість маршалінгу та демаршалінгу даних може бути значною, не кажучи вже про час, витрачений на надсилання речей через мережу. Це означає, що потрібно по-різному думати про дизайн API для віддалених і локальних інтерфейсів.

Проста спроба без зайвих роздумів перетворити локальний API на віддалений, швидше за все, призведе до проблем. У деяких із найгірших прикладів розробники можуть використовувати віддалені виклики, не підозрюючи про це, якщо абстракція надто непрозора.

Потрібно подумати також про саму мережу. Відомо, що першою з помилок розподілених обчислень є припущення щодо того, що мережа надійна. Мережі ненадійні [10]. Вони можуть і будуть виходити з ладу, навіть якщо клієнт і сервер в порядку. Мережі можуть виходити з ладу швидко, можуть повільно.

Збій може бути спричинений віддаленим сервером, який повертає помилку, або тим, що був зроблений неправильний виклик.

Чи можливо помітити різницю, і якщо так, чи можливо щось з цим зробити? А що робити, коли віддалений сервер просто повільно відповідає?

Незважаючи на недоліки, RPC складно назвати поганим. Деякі поширені реалізації можуть призвести до тих проблем, які були описані. Через труднощі, пов’язані з використанням RPC, звичайно, потрібно приділяти цій технології широку увагу. Багато операцій цілком добре вписуються в модель на основі RPC. Треба пам’ятати про деякі потенційні підводні камені, пов’язані з RPC, якщо вибирати саме цю модель.

Не треба абстрагувати віддалені виклики до точки, коли мережа повністю прихована. Треба переконатися, що клієнти не забувають про те, що буде здійснено мережевий виклик. Якщо клієнтські бібліотеки не структуровані правильно, вони можуть бути проблематичними.

Порівняно з комунікацією через спільну базу даних, RPC, безперечно, є покращенням, коли мова йде про запити і відповіді.

Але є інший варіант, який варто розглянути.

1.2.3. Асинхронна комунікація на основі подій

Для багатьох мікросервісних систем рішенням є використання архітектури, керованої подіями [36]. У цій архітектурі мікросервіс публікує подію, коли відбувається щось помітне, наприклад, коли він оновлює бізнес-сутність. Інші мікросервіси підписуються на ці події.

Коли мікросервіс отримує подію, він може оновлювати власні бізнес-сутності, що може призвести до публікації більшої кількості подій.

Можливо використовувати події для реалізації бізнес-транзакцій, які охоплюють кілька мікросервісів [5]. Транзакція складається з ряду кроків. Кожен крок складається з оновлення мікросервісом бізнес-сутності та публікації події, яка запускає наступний крок.

За умови, що кожен мікросервіс атомарно оновлює базу даних і публікує подію, а брокер повідомлень гарантує, що події доставляються принаймні один

раз, можливо реалізувати бізнес-транзакції, які охоплюють кілька мікросервісів. Важливо зазначити, що це не ACID-транзакції, які пропонують набагато більш суворі гарантії. Цю модель транзакцій називають BASE, і вона може запропонувати лише узгодженість кінцевого результату [2].

Можливо використовувати події для підтримки матеріалізованих представлень, які попередньо об’єднують дані, що належать кільком мікросервісам [40]. Мікросервіс, що підтримує представлення, підписується на відповідні події та оновлює представляння.

Наприклад, мікросервіс оновлення представлення замовлень клієнтів, що підтримує перегляд замовлень клієнтів, підписується на події, опубліковані мікросервісом обслуговування клієнтів і мікросервісом замовлень.

Коли мікросервіс оновлення матеріалізованого представлення замовлення клієнта отримує подію клієнта або замовлення, він оновлює сховище даних, що містить представлення. Можливо реалізувати перегляд замовлення клієнта за допомогою бази даних документів, такої як MongoDB, і зберігати один документ для кожного окремого клієнта.

Тут потрібно розглянути дві основні частини: спосіб, за допомогою якого мікросервіси видають події, і спосіб, за допомогою якого споживачі дізнаються, що ці події відбулися.

Традиційно брокери повідомлень намагаються вирішити обидві проблеми. Виробники використовують API, щоб опублікувати подію для брокера. Брокер обробляє підписки, дозволяючи споживачам отримувати інформацію про подію.

Брокери повідомлень можуть навіть керувати станом споживачів, допомагаючи відстежувати, які повідомлення вони бачили раніше [47].

Ці системи, як правило, розроблені таким чином, щоб бути масштабованими та стійкими [45]. Але це не безкоштовно, тому що може ускладнити процес розробки, оскільки це ще одна система, яку може знадобитися запустити для розробки та тестування. Для підтримки роботи цієї інфраструктури також можуть знадобитися додаткові машини та досвід. Але як

тільки це станеться, це може стати неймовірно ефективним способом реалізації слабозв’язаних архітектур, керованих подіями [8].

Інший підхід полягає в тому, щоб спробувати використовувати HTTP як спосіб поширення подій [13]. ATOM — це REST-сумісна специфікація, яка визначає семантику (серед іншого) для публікації каналів ресурсів. Існує багато клієнтських бібліотек, які дозволяють створювати та використовувати ці канали. Споживачі просто опитують канал, очікуючи на зміни.

З одного боку, той факт, що можливо повторно використовувати існуючу специфікацію ATOM і будь-які пов’язані бібліотеки, корисний, і відомо, що HTTP дуже добре справляється з масштабуванням [7]. Однак HTTP погано працює з низькою затримкою (де деякі брокери повідомлень перевершують), і все одно потрібно мати справу з тим фактом, що споживачі повинні відстежувати, які повідомлення вони вже бачили раніше.

Шаблон «Конкурентний споживач» описує метод, за допомогою якого можуть співіснувати кілька екземплярів працівників для конкуренції за повідомлення, що добре працює для збільшення кількості працівників для обробки списку незалежних завдань. Однак бажано уникнути випадку, коли два або більше працівників бачать одне й те саме повідомлення, оскільки тоді завдання буде виконуватися більше, ніж потрібно. За допомогою брокера повідомлень, стандартна черга впорається з цим. З ATOM потрібно власноруч керувати спільним станом для всіх працівників. Якщо є хороший, надійний брокер повідомлень, можна подумати про його використання для публікації та підписки на події. Але якщо його ще немає, можна подивитися на ATOM. Якщо з часом знадобиться більше функціоналу, який надає брокер повідомлень, завжди можливо змінити підхід.

З точки зору того, що фактично надсилається через ці асинхронні протоколи, застосовуються ті ж міркування, що й для синхронного зв’язку.

Архітектура, керована подіями, має кілька переваг і недоліків.

Це дає змогу реалізувати транзакції, які охоплюють кілька мікросервісів, і забезпечують узгодженість. Ще одна перевага полягає в тому, що це також дозволяє програмі підтримувати матеріалізовані представлення.

Один недолік полягає в тому, що модель програмування є більш складною, ніж при використанні ACID-транзакцій. Часто доводиться впроваджувати компенсаційні транзакції для відновлення після збоїв; наприклад, скасувати замовлення, якщо перевірка кредитоспроможності не пройде.

Крім того, компоненти системи повинні мати справу з тимчасово неузгодженими даними. Програма може бачити невідповідності, якщо вона читає з матеріалізованого представлення, яке ще не оновлено. Іншим недоліком є те, що потрібно виявляти та ігнорувати події, які вже були оброблені раніше.

1.3. Висновки до першого розділу

В першому розділі були розглянуті фундаментальні питання щодо організації даних та комунікацій, які слід вирішити в мікросервісній архітектури.

Мікросервісна система може використовувати спільну базу даних для всіх мікросервісів, але також може бути окрема база даних для кожного мікросервісу.

Комунікація між мікросервісами може бути синхронною та асинхронною, із використанням черги повідомлень, або без неї.

Були проаналізовані довгострокові наслідки та вплив кожного з описаних архітектурних рішень на здатність системи до масштабування.

Розподілені системи не мають альтернатив, якщо виникає потреба забезпечити горизонтальне масштабування. Але теорема CAP наголошує, що неможливо досягти всіх властивостей одночасно. Тому процес створення мікросервісної архітектури вимагає постійного пошуку компромісів.

Сама по собі концепція мікросервісної архітектури дозволяє системі бути гетерогенною, тобто можливо використовувати різні рішення для різних мікросервісів в межах однієї системи.

На перший план виходить детальний аналіз бізнес-логіки на предмет того, які функціональні можливості мають бути в системі для її коректної реалізації. Розробники мають бути свідомими щодо того, як ті чи інші процеси, працюють в реальному світі, тому що обмеження, які накладає теорема CAP, безпосередньо відображають обмеження реального світу.

Все це може створювати додаткове когнітивне навантаження на розробників. Але якщо все зробити правильно, стає зрозуміло, що описані обмеження насправді не є обмеженнями. Навпаки, відсутність обмежень в межах одного комп’ютерного вузла є лише абстракцією, яку можливо забезпечити в цьому окремому випадку, тому що можливість виникнення розподілу мережі відсутня як така.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ТА ОПТИМІЗАЦІЯ МЕТОДІВ ЗДІЙСНЕННЯ РОЗПОДІЛЕНИХ ТРАНЗАКЦІЙ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

2.1. Проблема подвійного запису та методі її вирішення

Єдиним показником того, що може виникнути проблема подвійного запису, є необхідність передбачуваного запису в декілька систем запису.

Ця вимога може бути неочевидною, і вона може проявлятися різними способами в процесі проектування розподілених систем. Наприклад:

– були вибрані найкращі інструменти для кожної окремої частини системи, і тепер потрібно оновити базу даних NoSQL, пошуковий індекс і кеш як частину однієї бізнес-транзакції;

– мікросервіс має оновити свою базу даних, а також надіслати сповіщення іншому мікросервісу про зміни;

– є бізнес-операції, які охоплюють межі кількох мікросервісів;

– можливо, доведеться реалізувати операції мікросервісу як ідемпотентні, оскільки клієнти повинні повторювати невдалі виклики.

У цьому розділі використовується один приклад сценарію, щоб оцінити різні підходи до обробки подвійних записів у розподілених транзакціях.

Сценарій — існує клієнтська програма, яка викликає мікросервіс під час операції зміни. Мікросервіс А має оновити свою базу даних, але він також має викликати мікросервіс В під час операції запису.

Невелике, але критичне уточнення пояснює, чому немає простих рішень цієї проблеми (рис. 2.1). Якщо мікросервіс А записує у свою базу даних, а потім надсилає сповіщення до черги для мікросервісу В (підхід «локально зафіксувати, а потім опублікувати»), все ще існує ймовірність, що програма не завжди працюватиме надійно [37].

Фактичний тип бази даних, протокол взаємодії між мікросервісами, не має значення для дослідження оскільки проблема залишається та ж сама [20].

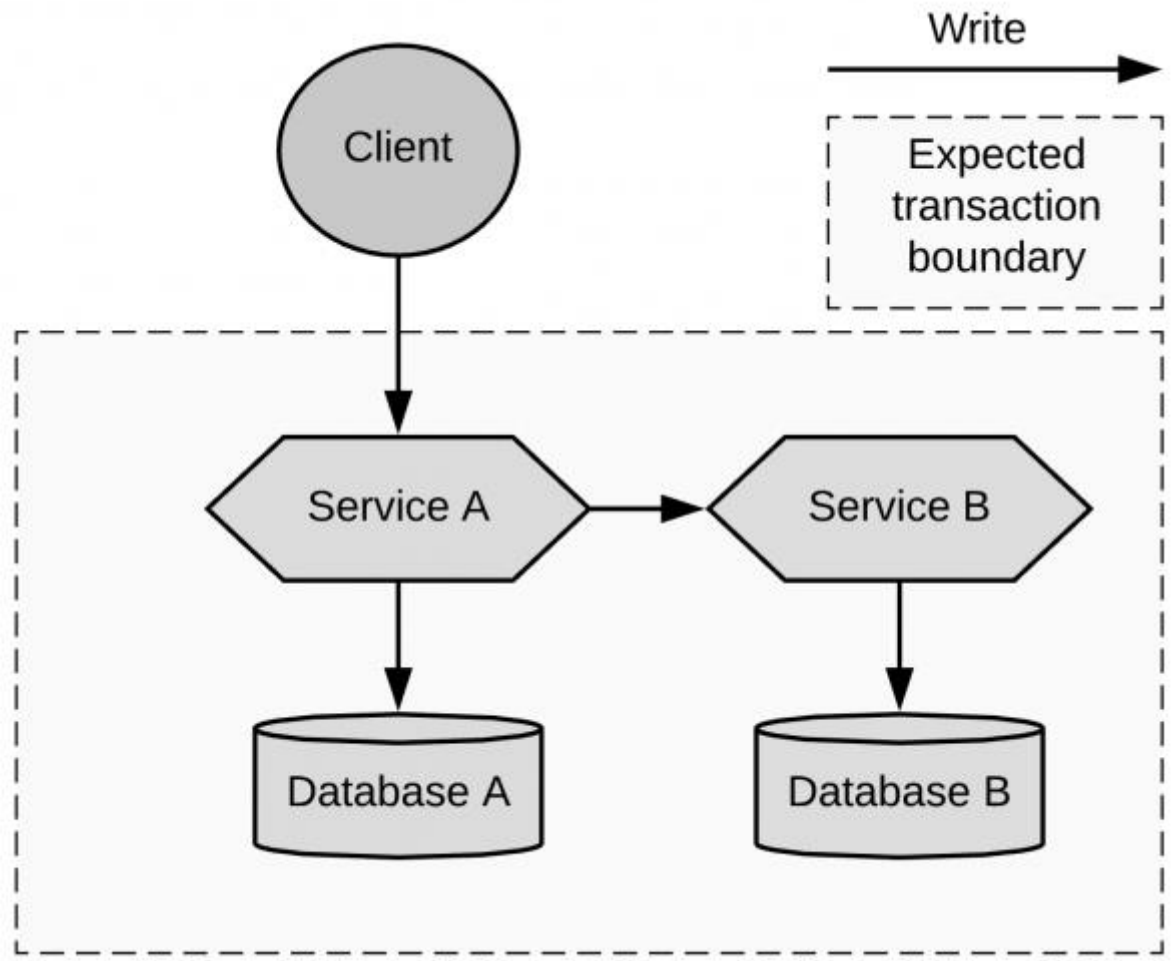

Рис. 2.1. Проблема подвійного запису

Якщо мікросервіс А записує у свою базу даних, а потім надсилає повідомлення до черги, існує невелика ймовірність збою програми після фіксації в базі даних і перед другою операцією, що залишить систему в неузгодженому стані. Якщо повідомлення надсилається до запису в базу даних (підхід «опублікувати, а потім локально зафіксувати»), існує ймовірність невдачі запису в базу даних або проблеми з часом, коли мікросервіс В отримує подію до того, як мікросервіс А зафіксує зміни до своєї бази даних.

У будь-якому випадку цей сценарій передбачає подвійний запис до бази даних і черги, що є основною проблемою, яка досліджується.

Далі розглядаються різні доступні на сьогодні підходи до реалізації цієї постійно актуальної проблеми.

2.1.1. Модульний моноліт

Розробка системи як модульного моноліту може здаватися поверненням назад в еволюції архітектури, але це може добре працювати на практиці. Це не шаблон мікросервісів, а виняток із правила мікросервісів, який можна обережно поєднувати з іншими мікросервісами.

Якщо сувора узгодженість запису є головною вимогою, важливішою навіть за здатність самостійно розгортати та масштабувати мікросервіси, тоді можна вибрати модульну монолітну архітектуру.

Наявність монолітної архітектури не означає, що система погано спроектована або погана. Це нічого не говорить про якість.

Як випливає з назви, це система, спроектована за модульним принципом, яка розгортається як єдина програма. Це спеціально розроблений і реалізований модульний моноліт, який відрізняється від випадково створеного моноліту, який з часом розростається. У цілеспрямованій модульній монолітній архітектурі кожен модуль відповідає принципам мікросервісів. Таким чином, кожен модуль інкапсулює весь доступ до своїх даних.

За допомогою цього підходу доведеться перетворити обидва мікросервіси (A і B) на бібліотечні модулі, які можна розгорнути в спільному виконавчому середовищі. Потім обидва мікросервіси налаштовуються на спільний доступ до одного екземпляра бази даних.

Оскільки мікросервіси написані та розгорнуті як бібліотеки в спільному виконавчому середовищі, вони можуть разом брати участь в транзакціях. Оскільки модулі спільно використовують екземпляр бази даних, можливо використовувати локальну транзакцію для фіксації або відкоту всіх змін одночасно. Існують також відмінності щодо методу розгортання, оскільки модулі повинні розгорталися як бібліотеки в рамках більшого розгортання та брали участь у існуючих транзакціях.

Навіть у монолітній архітектурі існують способи ізоляції коду та даних (рис. 2.2). Наприклад, можливо розділити модулі на окремі пакети, створити модулі та сховища вихідного коду, які можуть належати різним командам.

Можливо виконати часткову ізоляцію даних, згрупувавши таблиці за правилами іменування, схемами, бази даних або навіть за серверами баз даних.

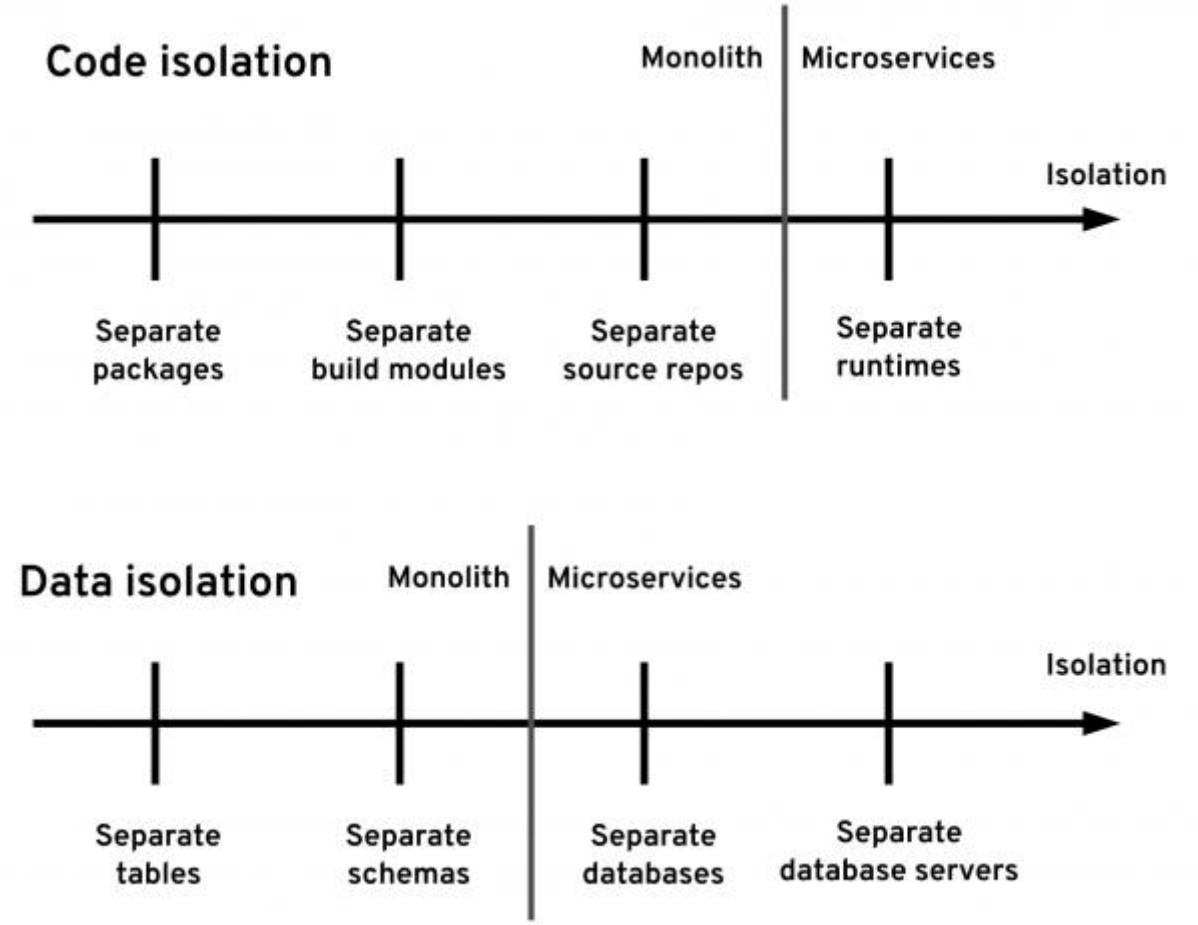

Рис. 2.2. Ізоляція коду та даних

Останньою частиною головоломки є використання середовища виконання та модуля-обгортки, здатного використовувати інші модулі та включати їх у контекст єдиної транзакції.

Усі ці обмеження роблять модулі більш зв’язаними, ніж типові мікросервіси, але перевага полягає в тому, що модуль-обгортка може розпочати транзакцію, викликати бібліотечні модулі для оновлення їхніх баз даних, а також

зафіксувати або відкотити транзакцію як одну операцію, не турбуючись про частковий збій або втрачену послідовність.

Для прикладу (рис. 2.3) мікросервіс A та мікросервіс B перетворені на бібліотеки та розгорнуті у спільному середовищі виконання.

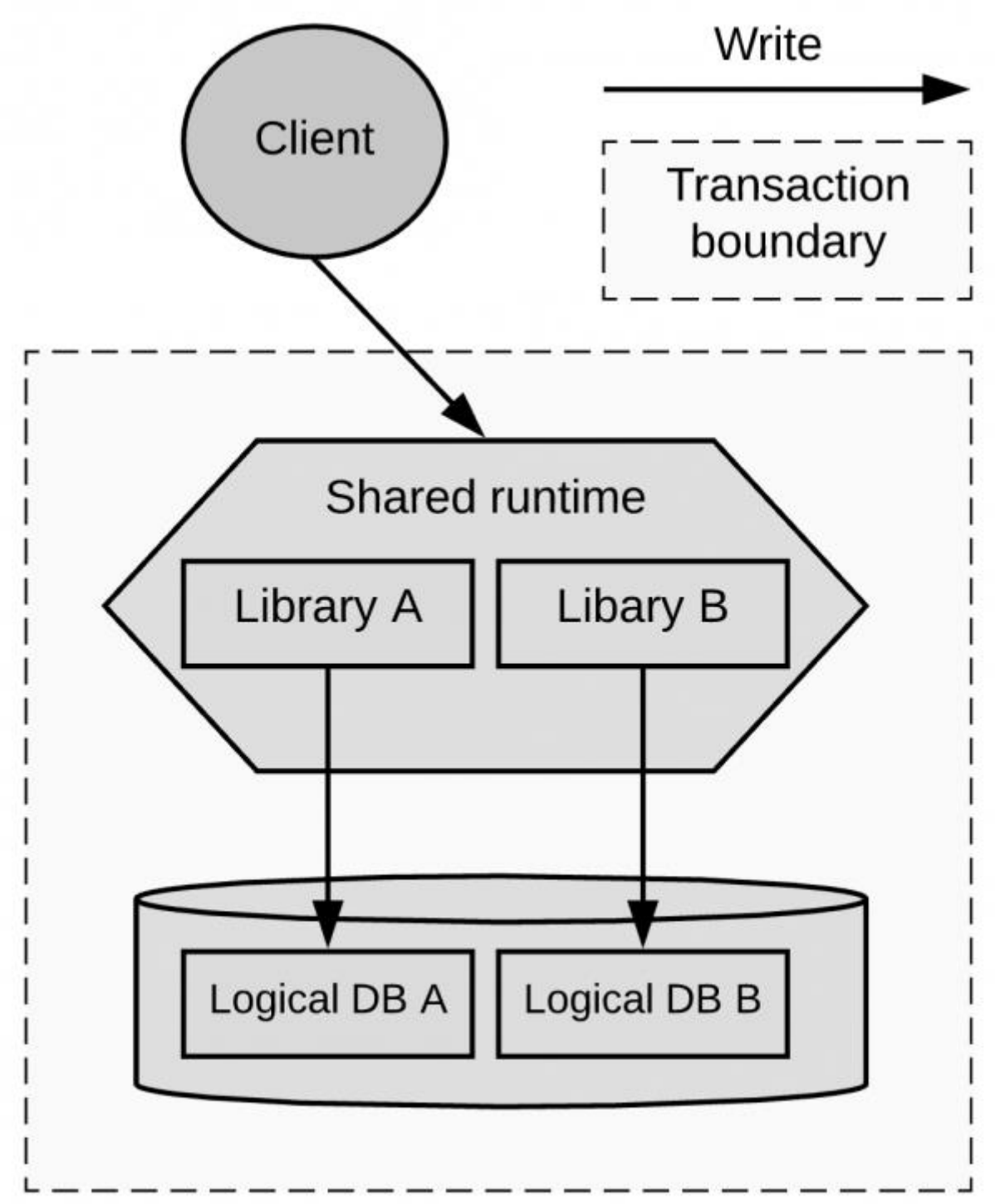

Рис. 2.3. Модульний моноліт

Ключова перевага модульного моноліту полягає в використанні простої семантики локальних транзакцій, що забезпечують узгодженість даних, читання записів, відкоти тощо. Втім, є і суттєві недоліки:

– спільне середовище виконання не дозволяє самостійно розгортати та масштабувати модулі, а також запобігає ізоляції відмов;

– логічне розділення таблиць в одній базі даних не є надійним. З часом він може перетворитися на спільний інтеграційний рівень;

– з'єднання модулів і обмін контекстом транзакцій вимагає координації на етапі розробки та збільшує зв'язок між компонентами системи.

2.1.2. Двофазні транзакції

Двофазні транзакції зазвичай є останнім засобом, який використовується в переважно в наступних випадках:

– коли записи в різні ресурси потребують суворої узгодженості;

– коли доводиться писати в гетерогенні сховища даних;

– коли потрібна гарантія одноразової обробки повідомлень, але не можна реорганізувати систему, щоб зробити її операції ідемпотентними;

– під час інтеграції зі сторонніми застарілими системами, які реалізують специфікацію двофазної транзакції.

У всіх цих ситуаціях, коли відсутність горизонтальної масштабованості не є проблемою, можливо розглядати двофазні транзакції як варіант.

Технічні вимоги для двофазної фіксації полягають у тому, що потрібен менеджер розподілених транзакцій, і надійний рівень зберігання для журналів транзакцій [41]. Також потрібні сховища даних, сумісні з XA, які здатні брати участь у розподілених транзакціях, такі як реляційні СУБД, брокери повідомлень та кеші. Менеджер транзакцій повинен бути доступним і завжди мати доступ до журналу, який зберігає стан транзакцій [22].

У прикладі (рис. 2.4) мікросервіс A використовує двофазні транзакції для внесення всіх змін до своєї бази даних та черги повідомлення, не залишаючи жодного шансу для дублікатів або втрачених повідомлень.

Подібним чином мікросервіс B може використовувати двофазні транзакції для споживання повідомлень і фіксації в базі даних B в одній транзакції без дублікатів. Або мікросервіс B може вибрати не використовувати двофазні транзакції, а використовувати локальні транзакції та реалізувати модель ідемпотентного споживача.

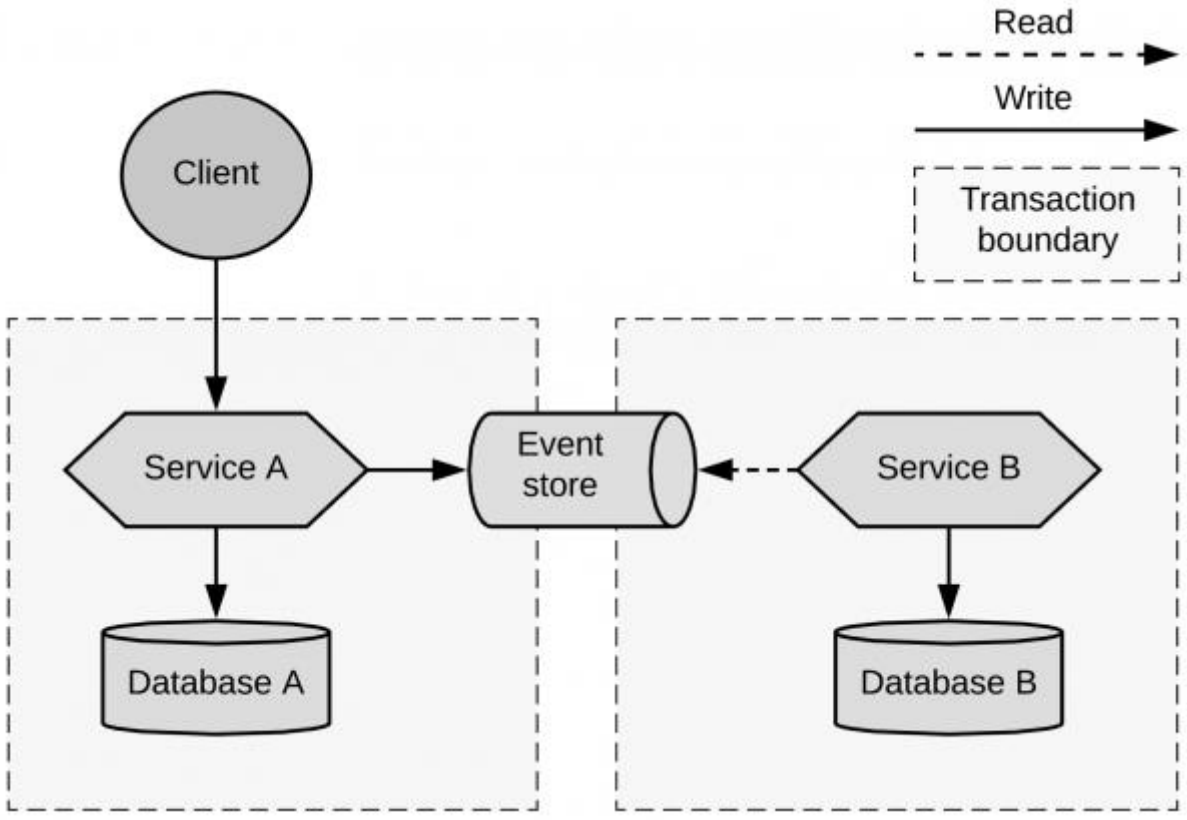

Рис. 2.4. Двофазні транзакції

Двофазні транзакції пропонують аналогічні гарантії локальних транзакцій у модульному монолітному підході, але за кількома винятками. Оскільки в атомарному оновленні бере участь два або більше окремих сховищ даних, вони можуть вийти з ладу і заблокувати транзакцію [33].

Але завдяки центральному координатору все ще легко відновити стан розподіленої системи порівняно з іншими підходами

.

2.1.3. Оркестрація транзакцій

З модульним монолітом використовуються локальні транзакції та завжди можливо знати стан системи. Завдяки розподіленим транзакціям на основі протоколу двофазної фіксації також гарантується узгоджений стан. Але що, якщо полегшити вимоги до узгодженості, знаючи загальний стан розподіленої системи та координуючи її з одного місця?

У цьому випадку можливо розглянути підхід оркестрації, коли один із мікросервісів діє як координатор загальної зміни розподіленого стану. Мікросервіс оркестратора несе відповідальність за виклик інших мікросервісів, доки вони не досягнуть бажаного стану, або вживає компенсаційні дії у разі невдачі. Оркестратор використовує свою локальну базу даних для відстеження змін стану, і він відповідає за відновлення після збоїв (рис. 2.5).

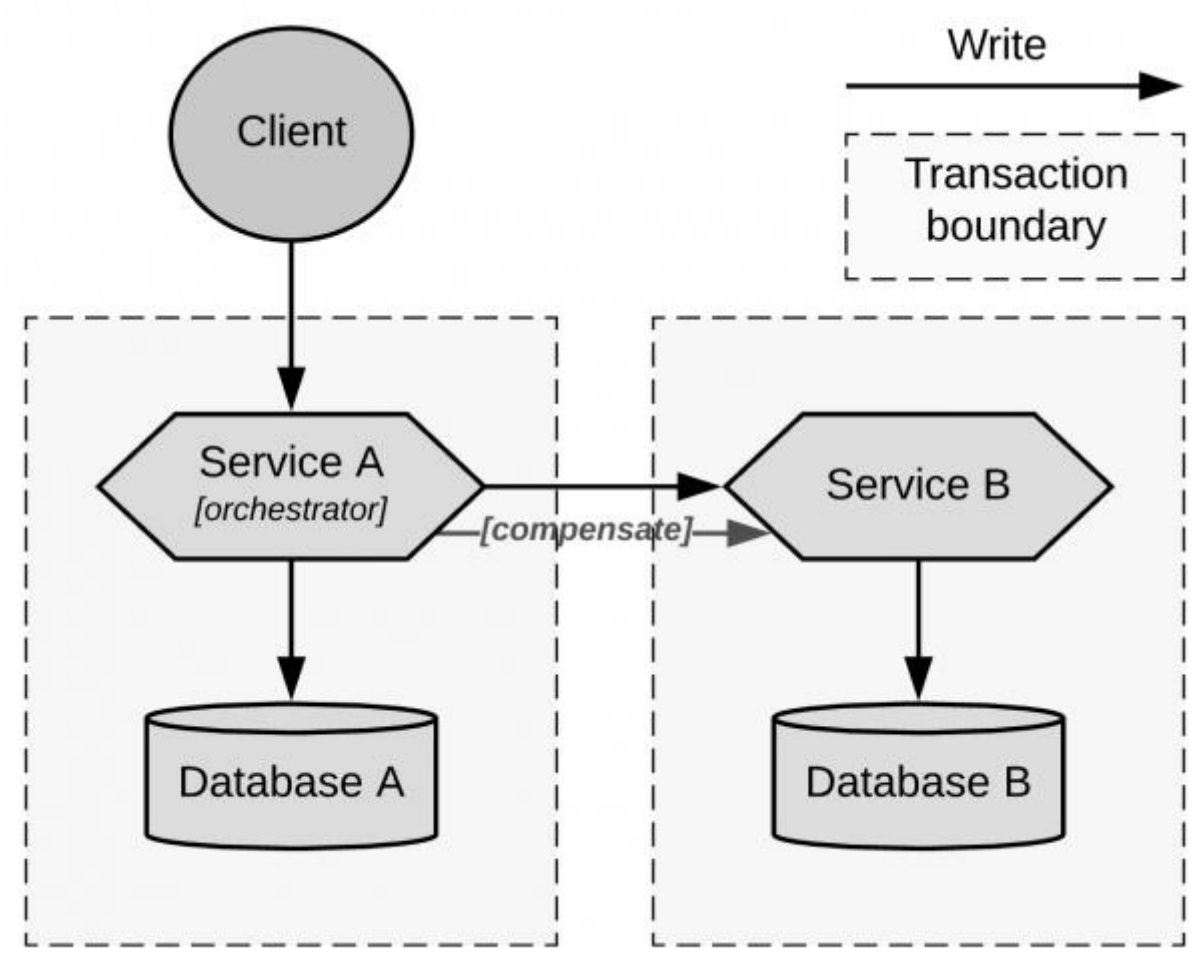

Рис. 2.5. Оркестрація транзакцій

У прикладі існує мікросервіс А, який діє як координатор стану, відповідальний за виклик мікросервісу В і відновлення після збоїв за допомогою операції компенсації, якщо це необхідно.

Важливою характеристикою цього підходу є те, що мікросервіс А та мікросервіс Б використовують локальні транзакцій, але мікросервіс А знає та відповідає за організацію загального потоку взаємодії.

Для реалізації комунікації можливо використовувати синхроні взаємодії, або чергу повідомлень, яка дозволяє використовувати двофазні транзакції.

Оркестрація транзакцій забезпечує узгодженість лише кінцевого результату, тому може включати повторні спроби та відкоти, щоб привести систему в узгоджений стан [23]. Хоча це дозволяє уникнути потреби в двофазних транзакціях, оркестрація вимагає від мікросервісів-учасників ідемпотентності операцій на випадок, якщо координатору доведеться повторити операцію. Великою перевагою цього підходу є можливість перевести гетерогенні мікросервіси, які можуть не підтримувати двофазні транзакції, в узгоджений стан, використовуючи лише локальні транзакції.

Координатору та мікросервісам-учасникам потрібні лише локальні транзакції, і завжди можна дізнатися стан системи, запитавши координатора, навіть якщо система перебуває в частково узгодженому стані. Використовуючи іншім методи, зробити це неможливо.

Переваги оркестрації транзакцій:

– координує стан між гетерогенними розподіленими компонентами;

– немає необхідності в XA-транзакціях;

– відомий розподілений стан на рівні координатора.

Недоліки оркестрації транзакцій:

– складна модель розподіленого програмування;

– може вимагати підтримки ідемпотентності та компенсаційних операцій від мікросервісів-учасників;

– лише кінцевий результат є узгодженим;

– можливі невиправні збої під час компенсації.

2.1.4. Хореографія транзакцій

Шаблон оркестрації транзакцій використовує централізований мікросервіс-координатор, який повідомляє всім учасникам, що їм робити.

Альтернативою є хореографія, яка є стилем координації мікросервісів, де учасники обмінюються подіями без централізованого контролю [14]. За допомогою цього шаблону кожен мікросервіс виконує локальну транзакцію та публікує події, які запускають локальні транзакції в інших мікросервісах.

Кожен компонент системи бере участь у прийнятті рішень щодо робочого процесу бізнес-операцій замість того, щоб покладатися на центральну точку контролю. Історично найпоширенішою реалізацією підходу хореографії транзакцій було використання асинхронного рівня обміну повідомленнями для взаємодії мікросервісів (рис. 2.6).

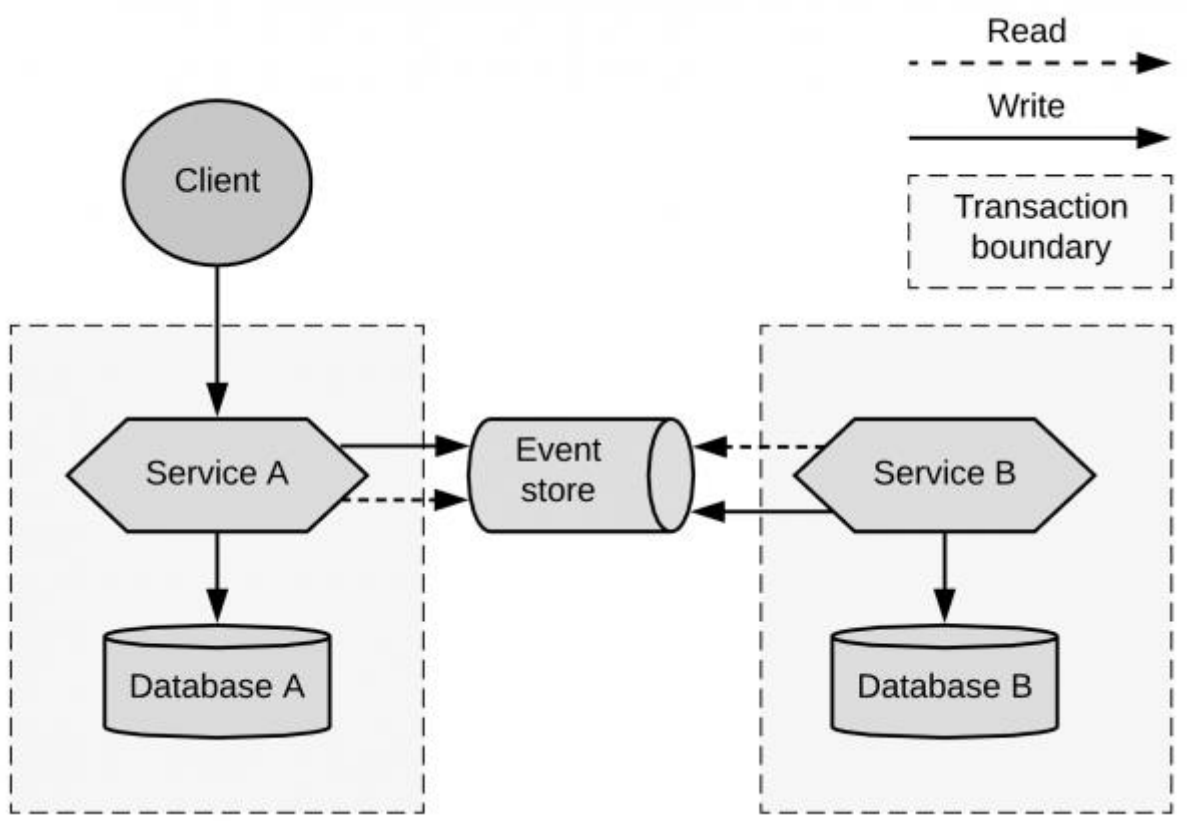

Рис. 2.6. Хореографія транзакцій

Щоб хореографія на основі повідомлень працювала, потрібно, щоб кожен мікросервіс-учасник виконував локальну транзакцію та ініціював виклик

наступного мікросервісна, публікуючи команду чи подію в інфраструктурі обміну повідомленнями. Подібним чином інші мікросервіси-учасники повинні споживати повідомлення та виконувати локальну транзакцію.

Це саме по собі є проблемою подвійного запису в межах проблеми подвійного запису вищого рівня. Коли розробляється рівень обміну повідомленнями з подвійним записом для реалізації хореографії транзакцій, можливо використати двофазну транзакцію, яка охоплює локальну базу даних і брокер повідомлень. Крім того, можливо також використати наступні методи:

– спочатку опублікувати повідомлення, а потім зафіксувати локальну транзакцію. Хоча цей варіант може здаватися непоганим, він має практичні проблеми. Наприклад, часто потрібно опублікувати ідентифікатор, який генерується під час фіксації транзакції, але він буде недоступний для публікації. Крім того, локальна транзакція може завершитися помилкою, але неможливо відкотити опубліковане повідомлення. У цьому підході відсутня семантика, яка дозволяє читати щойно зроблені записи, і тому це непрактичне рішення для більшості сценаріїв транзакцій;

– спочатку зафіксувати локальну транзакцію, а потім опублікувати повідомлення. Це має невелику ймовірність збою після здійснення транзакції та до публікації повідомлення. Але навіть у цьому випадку можливо зробити мікросервіси ідемпотентними та повторити операцію. Це означатиме повторне виконання транзакції, а потім публікацію повідомлення. Цей підхід може спрацювати, якщо зробити споживачів ідемпотентними. Загалом це також досить непоганий варіант реалізації.

Різноманітні способи реалізації хореографічної архітектури змушують кожен мікросервіс записувати лише в одне сховище даних за допомогою локальної транзакції, і нікуди більше. Потрібно розглянути, як це може працювати без подвійного запису.

Припускається, що мікросервіс А отримує запит і записує його в базу даних А і нікуди більше. Мікросервіс В періодично опитує мікросервіс А та

виявляє нові зміни. Коли зміни з’являються, мікросервіс B оновлює свою власну базу даних, а також індекс або часову мітку останньої зміни.

Критичним тут є той факт, що обидва мікросервіси записують лише у свою власну базу даних, здійснюючи локальну транзакцію.

Цей підхід (рис. 2.7), можна описати як хореографію, або можна описати його як пайплайн даних. Можливі варіанти реалізації тоді цікавіші.

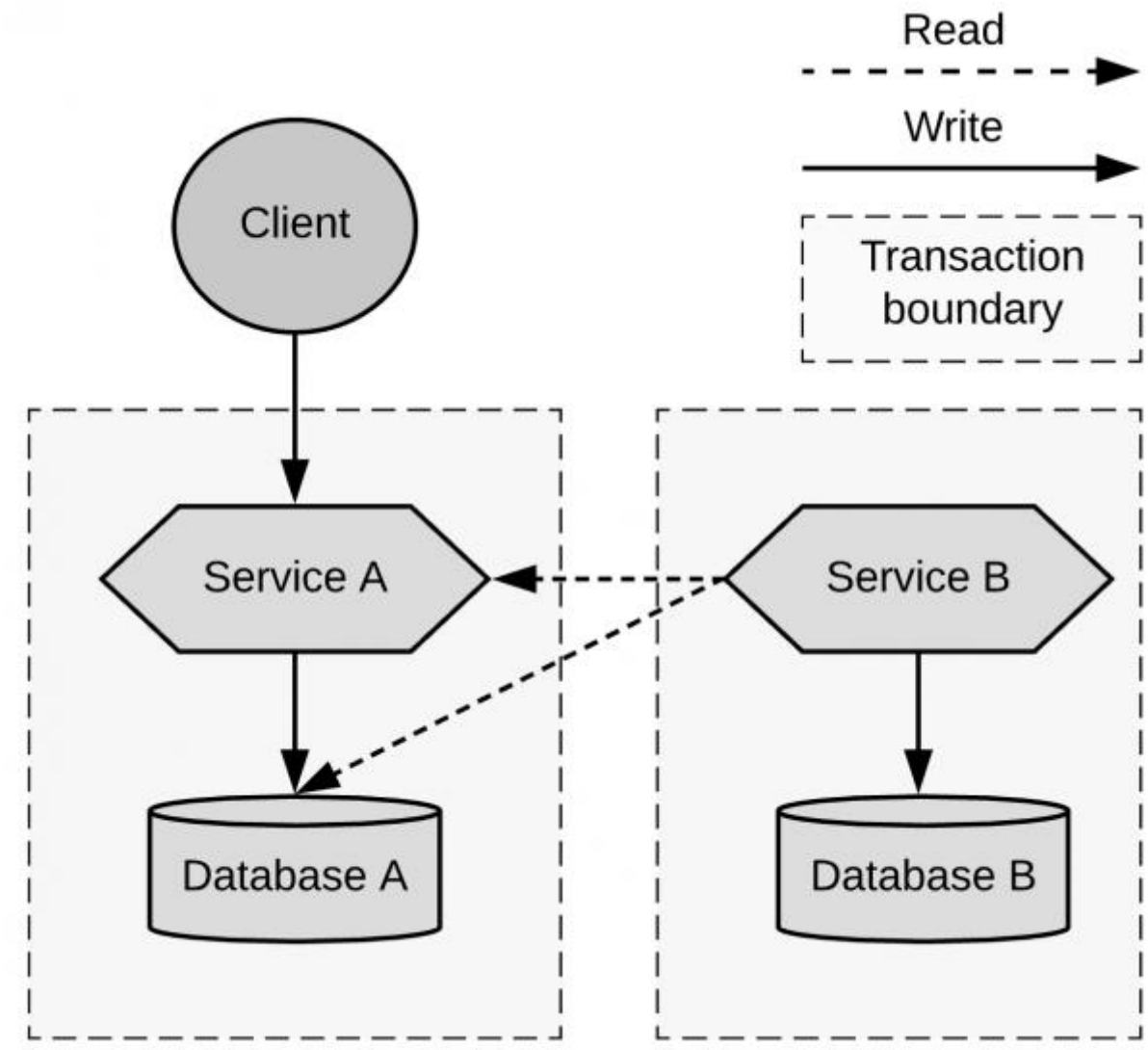

Рис. 2.7. Хореографія транзакцій без подвійного запису

Найпростіший сценарій полягає в тому, що мікросервіс B підключається до бази даних мікросервісу A і читає таблиці, що належать мікросервісу A.

Однак індустрія намагається уникати такого рівня зв’язку, і на це є поважна причина: будь-які зміни в реалізації мікросервісу A можуть порушити

роботу мікросервісу B. Можливо внести кілька поступових покращень у цей сценарій, наприклад, використовуючи шаблон «Outbox» та надавши мікросервісу A таблицю, яка діє як публічний інтерфейс. Ця таблиця може містити лише ті дані, які потрібні мікросервісу B, і її можна спроектувати так, щоб було легко запитувати та відстежувати зміни.

Якщо цього недостатньо, подальшим удосконаленням було б, щоб мікросервіс B запитував мікросервіс A про будь-які зміни через API, а не підключався безпосередньо до бази даних A.

По суті, усі ці варіації мають один спільний недолік: мікросервіс B має постійно опитувати мікросервіс A. Це може призвести до непотрібного безперервного навантаження на систему або непотрібної затримки в отриманні змін. Опитування мікросервісу на предмет змін – справа непроста, тож варто розглянути, що можливо зробити, щоб покращити цю архітектуру.

Один із способів покращити архітектуру хореографії та зробити її більш привабливою — це використати такий інструмент, як Debezium, який можливо використовувати для запису змін даних (CDC) за допомогою журналу транзакцій бази даних мікросервісу A.

Debezium може відстежувати журнал транзакцій бази даних, виконувати будь-яку необхідну фільтрацію та трансформацію та вносити відповідні зміни в Apache Kafka. Таким чином, мікросервіс B може слухати події, а не опитувати базу даних або API мікросервісу A.

Заміна опитування бази даних для потокових змін і введення черги повідомлень між мікросервісами робить розподілену систему більш надійною, масштабованою та відкриває можливість запровадження інших споживачів для нових випадків використання [6].

Використання Debezium пропонує елегантний спосіб реалізації оркестрації або хореографії з використанням подій (рис. 2.8).

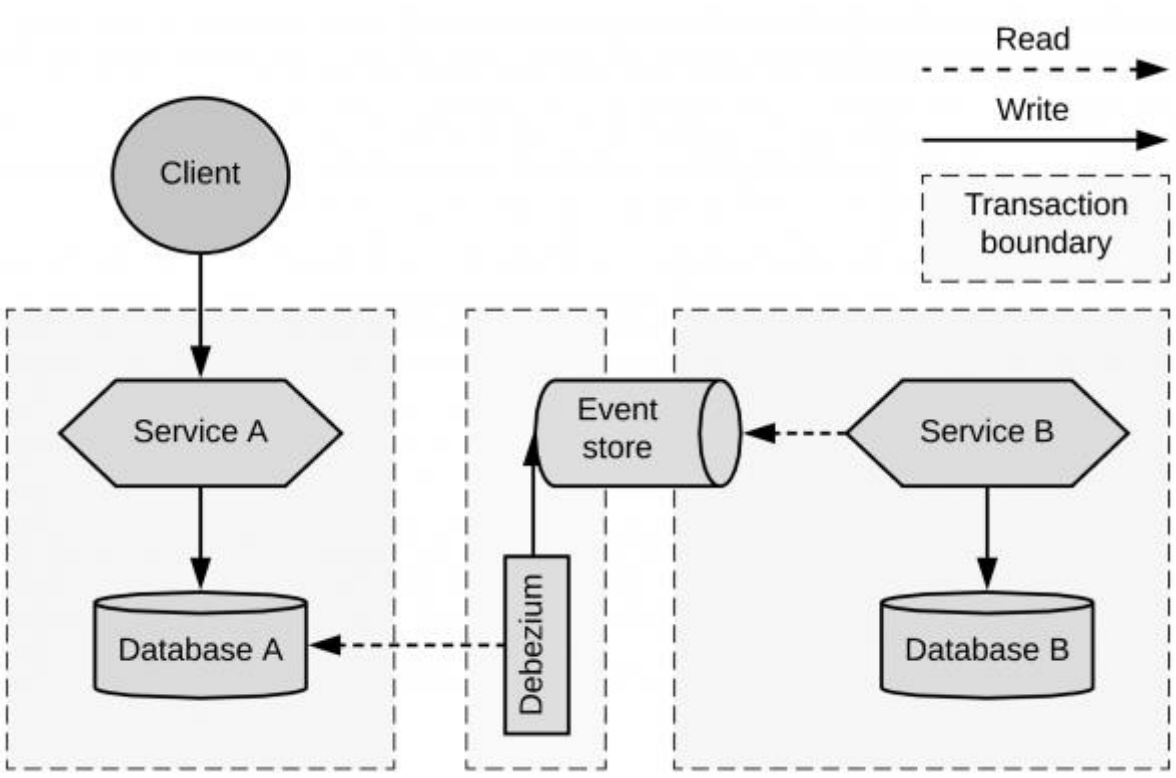

Рис. 2.8. Хореографія транзакцій без подвійного запису із застосуванням Debezium

Побічним ефектом цього підходу є те, що він вводить можливість отримання мікросервісом B дублікатів повідомлень. Це можна вирішити, реалізувавши мікросервіс ідемпотентним на рівні бізнес-логіки або за допомогою технічного дедуплікатора [27].

Event Sourcing є ще однією реалізацією підходу хореографії мікросервісів. За допомогою цього шаблону стан сутності зберігається як послідовність подій, що змінюють стан. Коли відбувається нове оновлення, замість оновлення стану сутності, нова подія додається до списку подій. Додавання нових подій до сховища подій є атомарною операцією, яка виконується в локальній транзакції [50].

Краса методу Event Sourcing (рис. 2.9), полягає у тому, що сховище подій також поводиться як черга повідомлень для інших мікросервісів, які споживають оновлення в реальному часі.

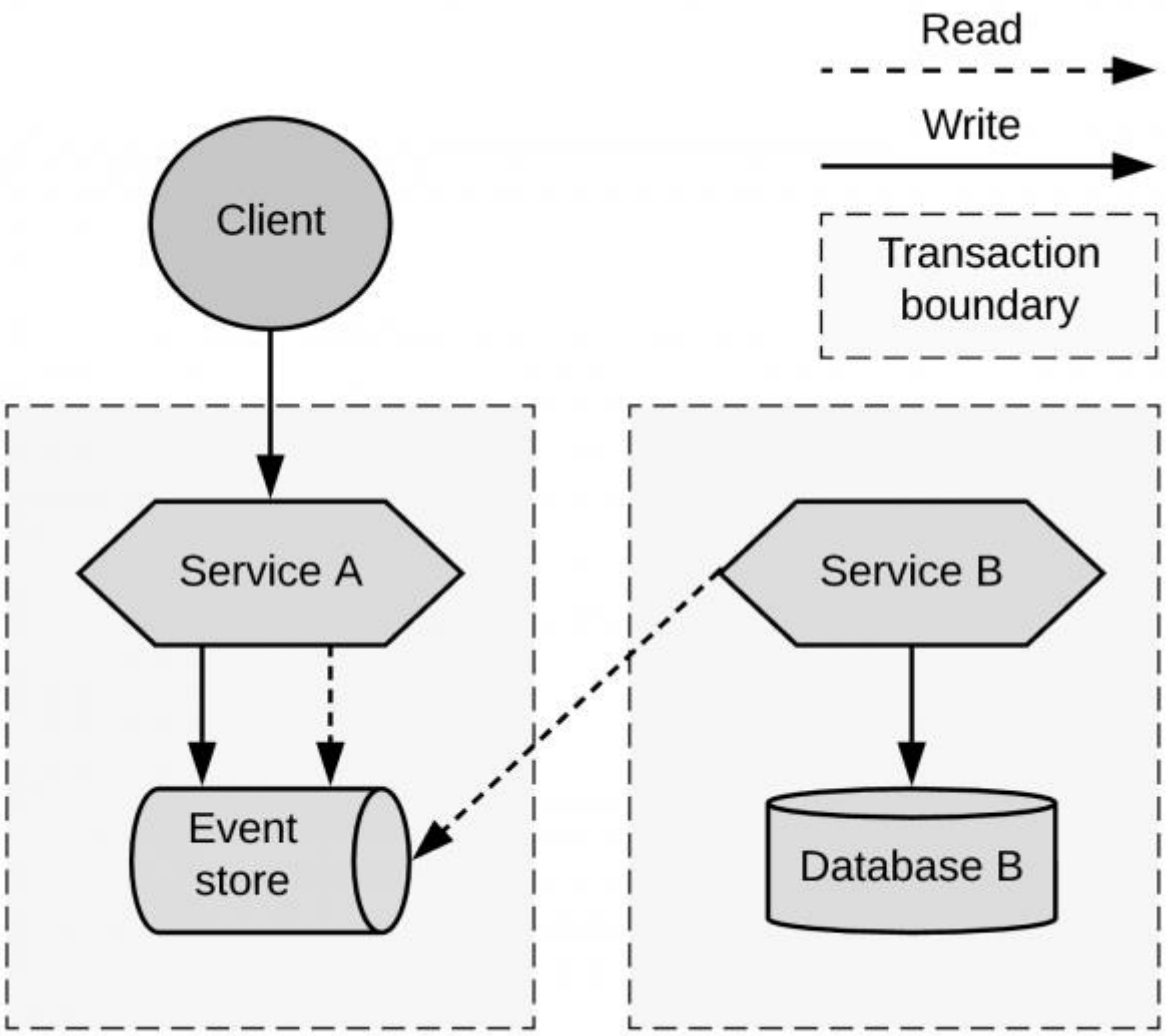

Рис. 2.9. Хореографія транзакцій без подвійного запису із застосуванням Event Sourcing

При використанні сховища подій клієнтські запити лише додаються в сховище подій. Мікросервіс А може реконструювати свій поточний стан, відтворюючи події. Сховище подій також дозволяє мікросервісу В підписатися на ті самі події оновлення. За допомогою цього механізму мікросервіс А використовує свій рівень зберігання також як рівень комунікації з іншими мікросервісами. Хоча цей механізм дуже акуратний і вирішує проблему надійної публікації подій щоразу, коли відбувається зміна стану, він вводить новий стиль програмування, незнайомий багатьом розробникам, і додаткову складність навколо реконструкції стану та ущільнення повідомлень, для чого потрібні спеціалізовані сховища даних.

Незалежно від механізму, який використовується для отримання змін даних, хореографія відокремлює записи, забезпечує незалежне масштабування мікросервісів та покращує загальну стійкість системи.

Недоліком цього підходу є те, що потік прийняття рішень децентралізований, тому важко виявити глобально розподілений стан. Виявлення стану клієнтського запиту вимагає запиту кількох сховищ даних, що може бути складним завданням із великою кількістю мікросервісів.

Переваги хореографії транзакцій:

– розділяє бізнес-логіку та комунікацію;

– відсутність центрального координатора транзакцій;

– покращені характеристики масштабованості та стійкості;

– взаємодія майже в реальному часі;

– Debezium та інше CDC дозволяють зменшити навантаження на систему.

Недоліки оркестрації транзакцій:

– логіка стану системи розкидана між усіма учасниками;

– лише кінцевий результат є узгодженим.

2.1.5. Паралельні пайплайни

При застосуванні хореографії немає центрального місця для запиту про стан системи, але є послідовність мікросервісів, яка поширює стан через розподілену систему. Хореографія створює послідовний пайплайн мікросервісів, тому є гарантія, що коли повідомлення досягає певного етапу загального процесу, воно пройшло всі попередні кроки.

Що, якщо можливо послабити це обмеження та обробити всі кроки незалежно? У цьому сценарії мікросервіс B може обробити запит незалежно від того, обробив його мікросервіс A чи ні (рис. 2.10).

За допомогою паралельних пайплайнів [30] створюється мікросервіс-маршрутизатор, який приймає запити та пересилає їх до мікросервісу A та мікросервісу B через брокера повідомлень в одній локальній транзакції.

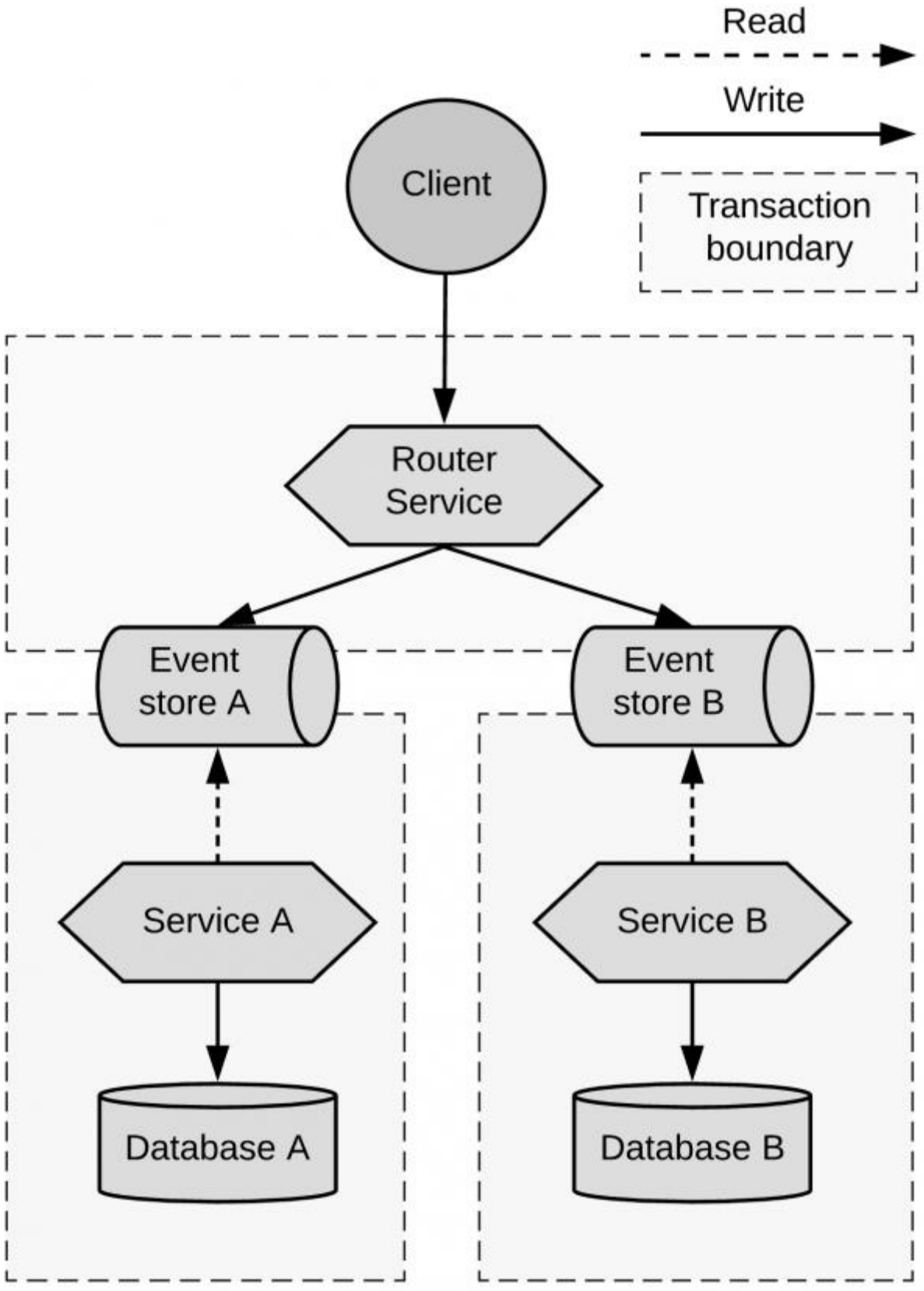

Рис. 2.10. Паралельний пайплайн

Обидва мікросервіси можуть обробляти запити незалежно та паралельно. Хоча цей шаблон дуже простий у реалізації, він застосовний лише до ситуацій, коли немає прив’язки між мікросервісами.

Існує легша альтернатива цьому підходу, відома як шаблон «Listen to yourself», де один із мікросервісів також діє як маршрутизатор. За допомогою цього альтернативного підходу, коли мікросервіс А отримує запит, він не буде записувати в свою базу даних, а натомість публікуватиме запит у системі обміну повідомленнями, яка перенаправить повідомлення мікросервісу В, а також самому мікросервісу А (рис. 2.11).

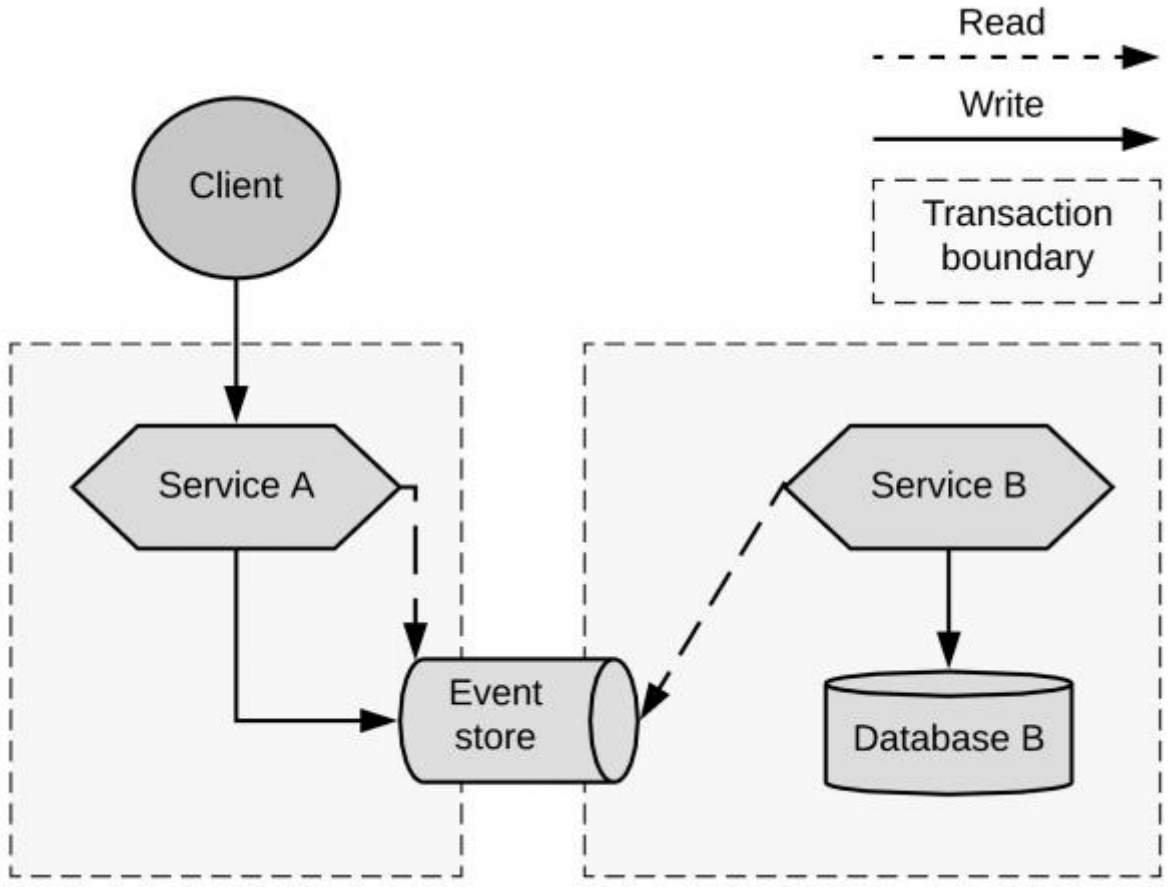

Рис. 2.11. Паралельний пайплайн із використанням шаблону «Listen to yourself»

Причиною відмови від запису в базу даних є уникнення подвійного запису. Коли повідомлення потрапляє в систему обміну повідомленнями, воно надходить до мікросервісу В, а також до мікросервісу А в абсолютно окремому контексті транзакції. Завдяки такому процесу обробки мікросервіси А та В можуть незалежно обробляти запит і записувати дані у свої бази даних.

Незалежно від реалізації, паралельні пайплайни надають простий та масштабований спосіб обробки запиту кількома мікросервісами одночасно, якщо виконання операції не потребує синхронізації між мікросервісами.

2.1.6. Як вибрати стратегію розподіленої транзакцій

Не існує правильного чи неправильного шаблону для обробки розподілених транзакцій в мікросервісній архітектурі. Кожен має свої плюси і мінуси. Кожен шаблон вирішує одні проблеми, у свою чергу породжуючи інші. У таблиці 2.1 наводиться короткий опис основних характеристик методів вирішення проблеми подвійного запису.

Таблиця 2.1

Стратегії розподілених транзакцій

	Модульний моноліт	Двофазні транзакції	Оркестрація	Хореографія	Паралельні пайплайни
Виконавче середовище	Єдине	Єдине/Розподілене	Розподілене	Розподілене	Розподілене
Сховища даних	Єдине	Гетерогенні (двофазні)	Гетерогенні	Гетерогенні	Гетерогенні
Координація	Центральна	Центральна	Центральна	Розподілена	Розподілена
Транзакційні властивості	ACID, блокуючі, синхронні	ACID, блокуючі, синхронні	ACD, неблокуючі, (а)синхронні	ACD, неблокуючі, асинхронні	ACD, неблокуючі, асинхронні
Реалізації	Локальні транзакції	XA	Saga, Outbox	Saga, Outbox, Event Sourcing	Listen to yourself

Можливо також організовувати та оцінювати описані стратегії на основі їхніх атрибутів узгодженості даних і масштабованості (рис. 2.12).

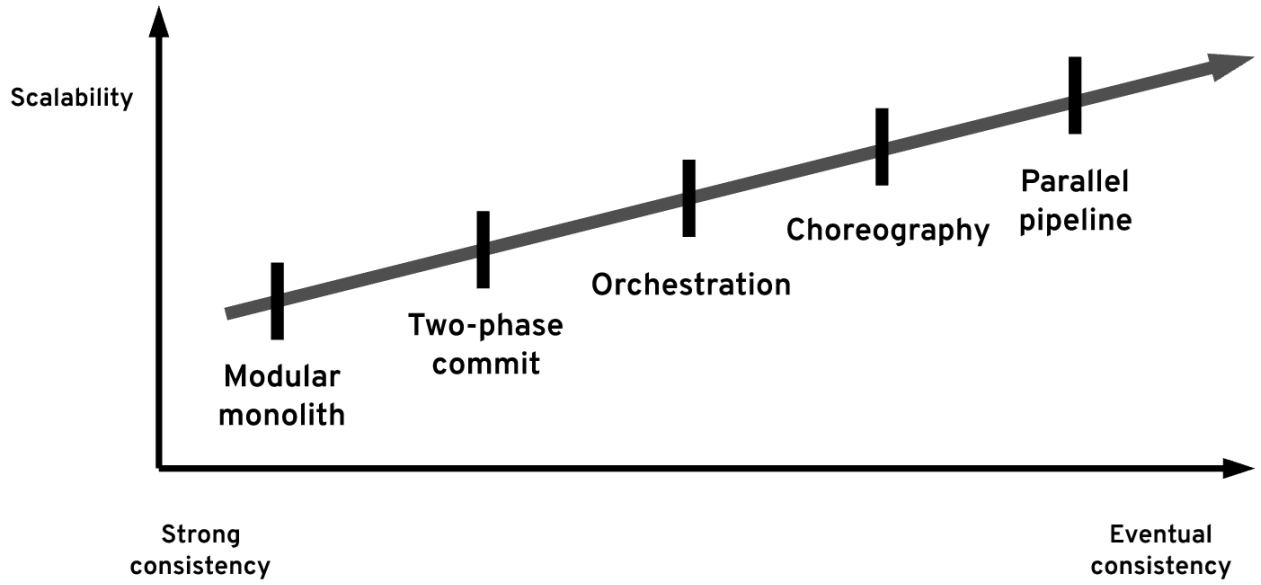

Рис. 2.12. Стратегії розподілених транзакцій за атрибутами узгодженості та масштабованості

Можливо відсортувати підходи, починаючи з найбільш масштабованих і високодоступних, закінчуючи найменш масштабованими і доступними:

– паралельні пайплайни та хореографія: Якщо транзакції не залежать одна від одної, тоді має бути сенс запустити їх у методі паралельних пайплайнів. Далі, якщо припустити, що між етапами обробки існує часовий зв’язок і певні операції та послуги мають відбуватися раніше інших, можливо розглянути підхід хореографії. Використовуючи хореографію, можна створити масштабовану, керовану подіями архітектуру, де повідомлення перетікають від мікросервісу до мікросервісу через децентралізований процес оркестрації. У цьому випадку реалізації із використанням Debezium та Apache Kafka є особливо цікавою;

– оркестрація та двофазні транзакції: Якщо хореографія не підходить, і потрібна центральна точка, яка відповідає за координацію та прийняття рішень, тоді можливо розглянули оркестрацію. Це популярна архітектура з доступними стандартними та спеціальними реалізаціями з відкритим кодом. У той час як стандартні реалізації можуть змушувати використовувати певну семантику транзакцій, своя реалізація оркестрації дозволяє розробити компроміс між бажаною узгодженістю даних і масштабованістю;

– модульний моноліт: Якщо дуже потрібна послідовність даних, потрібно бути готовим платити за це значними компромісами. У цьому випадку розподілені транзакції за допомогою двофазних транзакцій працюватимуть з певними сховищами даних, але їх важко надійно реалізувати в динамічних хмарних середовищах, розроблених для масштабованості та високої доступності. У цьому випадку можливо продовжити весь шлях до модульного моноліту, що використовує практики, отримані від мікросервісів. Цей підхід забезпечує найвищу узгодженість даних, але без масштабованості.

2.2. Оптимізація двофазних транзакцій в гетерогенних системах

Хоча двофазний протокол фіксації (2PC) частіше зустрічається в транзакціях, деякі ситуації не вимагають або не можуть вмістити обидві фази. У цих випадках можна використовувати однофазний протокол фіксації (1PC). Однією з ситуацій, коли це може статися, є коли сховище даних, яке не підтримує XA, має брати участь у транзакції.

У таких ситуаціях використовується оптимізація, відома як оптимізація фіксації останнього ресурсу [49]. Однофазна транзакція обробляється останньою на етапі підготовки транзакцій, і робиться її зафіксувати. Якщо фіксація вдається, журнал транзакцій записується, а транзакції, що залишилися, фіксуються із використанням двофазного протоколу. Якщо останній ресурс не вдається зафіксувати, транзакція відкочується.

Хоча цей протокол дозволяє більшості транзакцій виконуватися нормально, певні типи помилок можуть спричинити суперечливий результат.

Коли збій системи відбувається в певний момент часу під час двофазної обробки фіксації, це спричиняє неузгодженість даних. Зокрема, останній ресурс може бути зафіксовано, а решта ресурсів буде відкочено.

Розглядаючи збій оптимізації фіксації останнього ресурсу, припускається, що є розподілена транзакція, яка пише у сховище даних, а також надсилає повідомлення брокеру повідомлень, який не підтримує XA:

1. Виконується оновлення бази даних за допомогою команди INSERT INTO, таким чином відбувається залучення XA-ресурсу до транзакції.

2. Надсилається повідомлення брокеру повідомлень, таким чином ще один ресурс залучається до транзакції. Цей ресурс не підтримує XA.

3. Починається двофазна фіксація транзакції.

4. Виконується підготовка ресурсів до фіксації. Етап підготовки для ресурсу, який не підтримує XA, означає виклик фіксації, і виконується останнім. Цей виклик робить зміни даних видимими для зовнішнього світу. Журнал транзакції для інших ресурсів зберігається в надійних сховищах.

5. Відбувається збій до того, як менеджер транзакцій може зберегти інформацію про фіксацію в своєму журналі транзакцій.

6. Після перезапуску менеджер транзакцій немає поняття про існування будь-якої транзакції, тому підготовлені двофазні транзакції відкочуються під час відновлення розподіленої транзакції.

Можливо покращити стійкість цього алгоритму до помилок.

Для цього потрібно на етапі підготовки ресурсів до фіксації, замість фіксації того ресурсу, що не підтримує XA, відмічати цей ресурс як готовий до фіксації. А потім, на етапі фіксації, фіксувати цей ресурс першим.

Такий алгоритм досягає функціональних можливостей, подібних до XA, вимагаючи спеціальної таблиці, яка доступна для запису та читання менеджером транзакцій, та знаходиться у тій базі даних, що не підтримує XA.

Поведінка демонструється на прикладі:

1. Виконується оновлення бази даних за допомогою команди INSERT INTO, таким чином відбувається залучення XA-ресурсу до транзакції.

2. Надсилається повідомлення брокеру повідомлень, таким чином ще один ресурс залучається до транзакції. Цей ресурс не підтримує XA.

3. Починається двофазна фіксація транзакції.

4. Виконується підготовка ресурсів до фіксації. Етап підготовки для ресурсу, що не підтримує XA, означає збереження ідентифікатора цього ресурсу

до спеціальної таблиці. Цей виклик не робить зміни даних видимими для зовнішнього світу. Журнал транзакції зберігається в надійному сховищі.

5. На етапі фіксації той ресурс, який не підтримує XA, повинен бути зафіксований в першу чергу.

6. Двофазна обробка транзакції завершується.

Відмінність від попереднього прикладу полягає у тому, що той ресурс, який не підтримує XA, не впорядковується як останній під час обробки ресурсу, а впорядковується як перший. Підготовка ресурсу, який не підтримує XA, не означає виконання фіксації, але означає збереження інформації про те, що ресурс вважається підготовленим, у базі даних.

Як правило, всі операції з ресурсом, що не підтримує XA, виконуються в контексті одного і того ж з'єднання з базою даних.

Основний факт, який слід розуміти, полягає в тому, що розподілена транзакція вважається повністю підготовленою лише після фіксації того ресурсу, що не є XA. До цього часу транзакція вважається непідготовленою і буде оброблена з відкотом шляхом відновлення транзакції.

Важливо відзначити, що за коректну обробку збоїв під час транзакції, відповідає спеціальний модуль періодичного відновлення. Йому потрібно перевірити та зрештою обробити всі XA-ресурси, що належать до транзакції.

Модуль відновлення дізнається про факт фіксації ресурсу, що не підтримує XA, із запису, збереженого у сховищах даних.

Таким чином можливо об'єднати всі ресурси, що належать до тієї самої транзакції, зафіксувати їх, або відкотити.

2.3. Аномалія ізоляції розподілених транзакцій

Будь-які розподілені транзакції, навіть повністю двофазні, мають неочевидний рівень ізоляції, тому що зберігають властивості ізоляції лише в межах локальних транзакцій. Це може призводити до аномалій, які унеможливлюють зчитування глобального узгодженого стану системи [35].

Наприклад, якщо всі локальні транзакції, які приймають участь у розподілений транзакції, не забезпечують серіалізацію операцій, інша розподілена транзакція може зчитати стан системи в той момент, коли розподілена транзакція зафіксована лише частково. Таким чином друга розподілена транзакція буде працювати з неузгодженими даними.

Це тема для окремого дослідження, але в цілому можна сказати, що найбільш ефективним методом усунення цієї аномалії є використання всіма учасниками розподіленої транзакції синхронізованого годинника, такого як, наприклад, годинник Лемпорта [46], для визначення однакового часу початку та завершення для всіх локальних транзакцій. Це дозволить отримувати стан системи, узгоджений всіма учасниками транзакції.

Інший варіант полягає в використанні серіалізаційного рівня ізоляції для всіх локальних транзакцій, або в використанні блокувань. Втім, ці варіанти, як правило, набагато менш ефективні.

Варто також зазначити, що описана в цій роботі оптимізація гетерогенних розподілених транзакцій дозволяє використовувати будь-який рівень ізоляції, що гарантує причинно-наслідковий зв’язок, для транзакції, яка фіксується першою. Також можливо використовувати цю транзакцію в якості джерела часу початку та завершення всієї розподіленої транзакції.

2.4. Висновки до другого розділу

Проблема подвійного запису в мікросервісних системах має різні методи вирішення. В цьому розділі проаналізовані та порівняні основні методи здійснення розподілених транзакцій.

Також продемонстрований алгоритм, що дозволяє поєднувати декілька двофазних транзакцій з однією однофазною в межах розподіленої транзакції.

У сильно розподіленій системі з десятками мікросервісів не буде єдиного підходу, який би працював для всіх, але кілька з них комбінуються та застосовуються в різних контекстах.

Може бути кілька мікросервісів, розгорнутих у спільному середовищі виконання для виняткових вимог щодо узгодженості даних. Можна вибрати двофазну фіксацію для інтеграції із системою, яка підтримує ХА. Можна використати оркестрацію щоб організовувати складний бізнес-процес, а також використовувати хореографію та паралельну обробку для решти мікросервісів.

Зрештою, не має значення, яку стратегію використовувати. Важливо обдумано вибрати стратегію з правильних причин і реалізувати її.

РОЗДІЛ 3
ПРИКЛАД ОБРОБКИ РОЗПОДІЛЕНИХ ТРАНЗАКЦІЙ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

3.1. Функціональні можливості Apache Kafka, MongoDB та PostgreSQL

Перш ніж почати, варто зазначити, що жодна з наведених технологій не є обов'язковою для обробки розподілених транзакцій в мікросервісній архітектурі. Будь-яка технологія може бути замінена на альтернативну, якщо цієї альтернативи буде достатньо для втілення потреб системи. Крім того, сама концепція мікросервісної архітектури дозволяє використовувати різні технології в різних мікросервісах в межах однієї системи.

Втім, станом на поточний момент, запропоновані технології є достатньо функціональними та гнучкими для застосування фактично в усіх можливих сценаріях, що пов'язані з мікросервісною архітектурою. Що також важливо, всі ці технології є одними з найбільш використовуваних у своїх нішах, мають відкритий сирцевий код, та доступні за безкоштовними ліцензіями.

Apache Kafka — це розподілене сховище подій і платформа для обробки потоків даних. Це система з відкритим кодом, розроблена Apache Software Foundation, написана на Java та Scala. Проект має на меті забезпечити уніфіковану високопродуктивну платформу з низькою затримкою для обробки потоків даних у реальному часі. Apache Kafka може підключатися до зовнішніх систем (для імпорту/експорту даних) через Kafka Connect і надає бібліотеки Kafka Streams для створення програм, що оброблюють потоки.

MongoDB — документо-орієнтована система управління базами даних (СУБД) з відкритим вихідним кодом, яка не потребує опису схеми таблиць. MongoDB займає нішу між швидкими і масштабованими системами, що оперують даними у форматі ключ/значення, і реляційними СУБД, функціональними і зручними у формуванні запитів.

PostgreSQL — це безкоштовна система управління реляційними базами даних (РСУБД) із відкритим вихідним кодом, яка наголошує на розширюваності та сумісності з SQL. Спочатку вона була названа POSTGRES, посилаючись на її походження як спадкоємця бази даних Ingres, розробленої в Каліфорнійському університеті в Берклі.

.NET (вимовляється як «дотнет»; раніше .NET Core) — це безкоштовна платформа з відкритим кодом для виконання програмного забезпечення для операційних систем Windows, Linux і macOS. Це кросплатформний наступник .NET Framework. Проект в основному розроблено співробітниками Microsoft за допомогою .NET Foundation,

Apache Kafka, MongoDB та PostgreSQL мають суттєво різні функціональні можливості, використовуються для різних завдань, але при цьому підтримують дуже схожі моделі узгодженості, що робить їх інтеграцію простою та ефективною.

Apache Kafka та MongoDB здатні до теоретично необмеженого горизонтального масштабування, а PostgreSQL підтримує двофазні транзакції, що в свою чергу дозволяє виконувати розподілені транзакції між усіма трьома технологіями одночасно, використовуючи різні стратегії.

Використання .NET як універсального середовища виконання програмного коду дозволяє уникнути дублювання коду між різними мікросервісами, при цьому зберігається можливість використання різних мов програмування, хоча в цій роботі використовується лише мова C#. Також підтримується парадигма асинхронного програмування, що завдяки уникненню зайвих блокувань на рівні операційної системи дозволяє значно знизити споживання серверних ресурсів для всіх програм, в основі яких лежить інтенсивна мережева взаємодія, в тому числі мікросервісів. При необхідності також дозволяється напряму викликати машинний код, що робить можливою інтеграцію з програмами, які не використовують .NET.

3.2. Забезпечення балансування навантаження та високої доступності виконавчого середовища за допомогою Apache Kafka

Цікавою функцією Apache Kafka є можливість забезпечити балансування навантаження під час обробки повідомлень [38]. Програму можна налаштувати з кількома екземплярами споживачів, при цьому кожен споживач отримує частину (частку) повідомлень на тему. Ця функція дозволяє системі масштабуватись у міру збільшення обсягу даних, що записуються в тему. Це також забезпечує високу доступність: якщо один із екземплярів споживачів виходить з ладу, повідомлення можна перенаправити до інших споживачів [39].

У Apache Kafka тема — це категорія або канал, до якого записуються записи. Тема подібна (хоча й дещо інша) до концепції черг, що використовується в інших системах обміну повідомленнями.

Кожна тема може містити один або кілька розділів. Розділ — це впорядкована послідовність записів, які лише додаються. Щоразу, коли запис записується в тему, він направляється до розділу в цій темі (рис. 3.1)

.

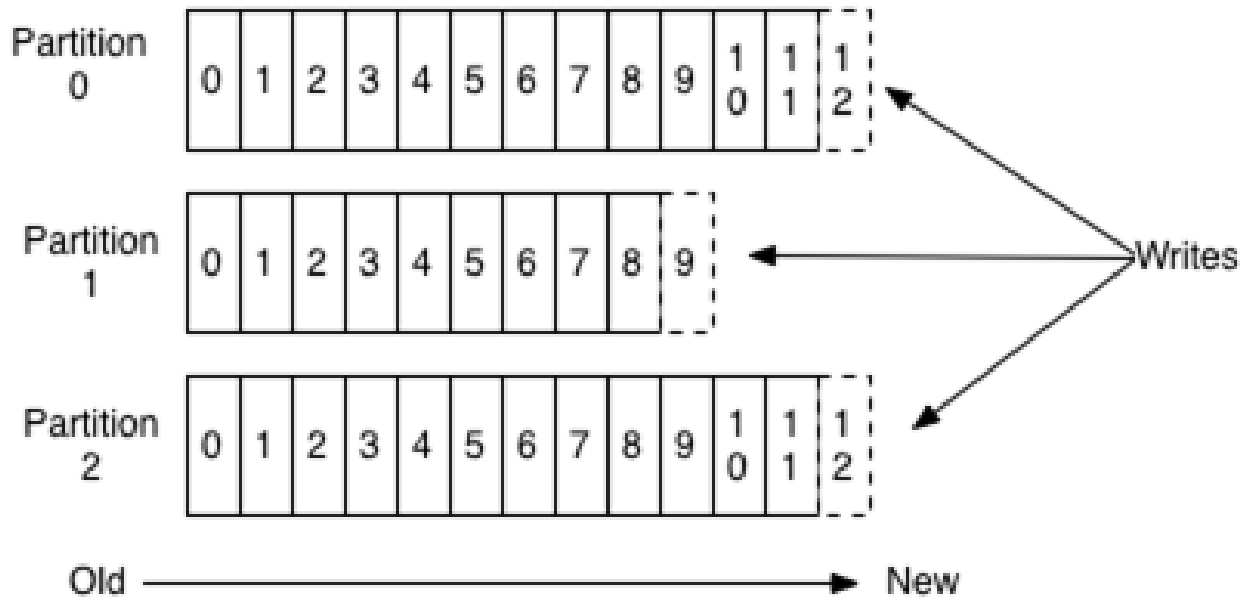

Рис. 3.1. Анатомія теми Apache Kafka

Щоразу, коли споживач підключається, він призначається до одного або кількох розділів. Повідомлення, отримані споживачем, надходитимуть лише з тих розділів, яким він призначений.

Балансування навантаження досягається шляхом призначення різних розділів різним споживачам. Наприклад, якщо тема містить 5 розділів, можливо мати 5 споживачів, які підключаються до кожного з них. Хоча цим призначенням, звичайно, можна керувати вручну, кращим підходом є використання груп Apache Kafka, щоб у разі збою споживача розділи, яким він був призначений, могли бути автоматично перерозподілені іншим споживачам.

Щоб забезпечити автоматичне призначення розділів споживачам, кожен екземпляр споживача може позначити себе як частину групи споживачів. Кілька екземплярів споживачів можуть бути частиною однієї групи, інакше кажучи, група може містити одного або більше споживачів.

Коли споживач підписується на тему, група, до якої він входить, цьому споживачеві автоматично призначаються розділи теми. Якщо кілька споживачів в одній групі підписуються на ту саму тему, група призначить різні розділи для кожного споживача (рис. 3.2).

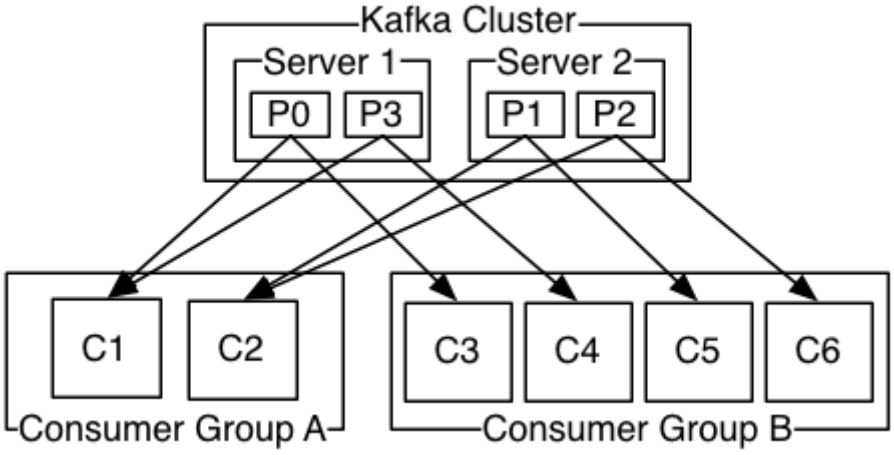

Рис. 3.2. Кластер Apache Kafka та дві групи споживачів

3.2. Гарантування одноразового виконання транзакції

Гарантія виконання принаймні один раз, яку надає Apache Kafka, означає, що точно буде отримано та оброблено кожне повідомлення, але повідомлення може бути отримано повторно в разі збою [26].

Є різні сценарії збою, які можуть призвести до неузгодженості.

Наприклад, програма надсилає повідомлення в Apache Kafka. Програма ніколи не отримує відповіді, тому знову надсилає повідомлення (рис. 3.3). У цьому випадку, можливо, перше повідомлення було успішно збережено, але підтвердження було втрачено, тому повідомлення додано двічі.

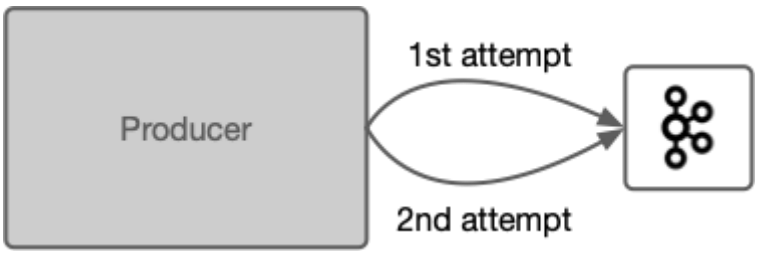

Рис. 3.3. Дуплікація при відмові

Або система обробляє великий файл, що містить події. Процес оброблює файл, надсилаючи повідомлення в Apache Kafka для кожної події. Після обробки половини файлу процес раптово падає та перезапускається. Процес знову починає обробляти файл із самого початку й позначає його як оброблений лише тоді, коли обробить весь файл. У цьому випадку події з першої половини файлу будуть надіслані в Apache Kafka двічі (рис. 3.4).

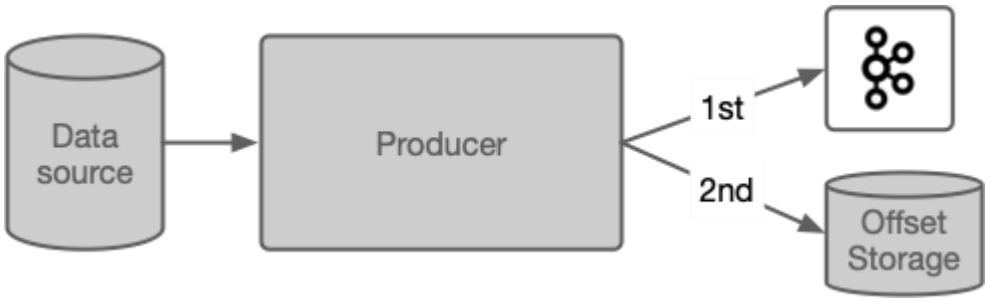

Рис. 3.4. Дуплікація через відсутність атомарності

Або споживач отримує повідомлення від Apache Kafka, перетворює їх і записує результати в базу даних. Споживач підтверджує обробку повідомлення лише після успішного запису результатів в базу даних (рис. 3.5). Якщо користувач зазнає помилки після запису у базу даних, але до підтвердження обробки повідомлення, він повторно обробить ті самі записи під час наступного запуску та збереже їх у базі даних ще раз.

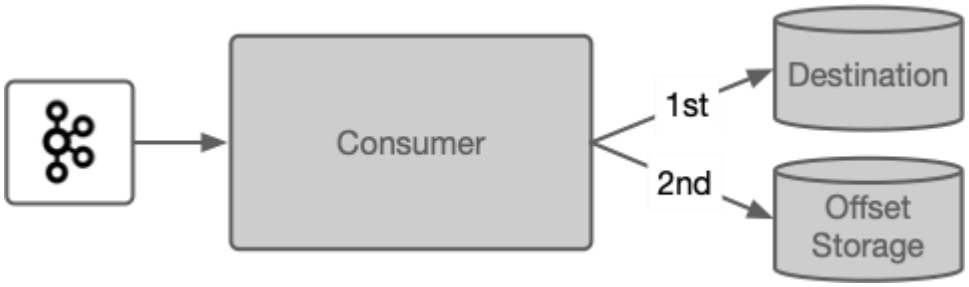

Рис. 3.5. Дуплікація даних при відмові сервера під час обробки повідомлення

Шаблон «Outbox» гарантує, що кожне повідомлення обробляється лише один раз за допомогою локальної транзакції бази даних, яка використовується для зберігання бізнес-даних. Алгоритм складається з двох фаз.

Алгоритм виконання першої фази «Outbox» (рис. 3.6):

1. Надходить вхідне повідомлення з Apache Kafka, але воно не підтверджується, щоб у разі невдачі повідомлення було доставлено знову.

2. Розпочинається транзакцію в базі даних.

3. Перевіряється таблиця Outbox в базі даних, щоб побачити, чи вхідне повідомлення вже оброблено. Це називається дедуплікацією. Якщо повідомлення вже оброблено, потрібно перейти до кроку 6. Якщо повідомлення ще не оброблено, потрібно перейти до кроку 4.

4. Виконується обробник повідомлень для вхідного повідомлення

5. Будь-які вихідні повідомлення не надсилаються негайно.

6. Будь-які вихідні повідомлення зберігаються в базі даних.

7. Транзакцію фіксується в базі даних. Це операція, яка забезпечує узгодженість між операціями обміну повідомленнями та базою даних.

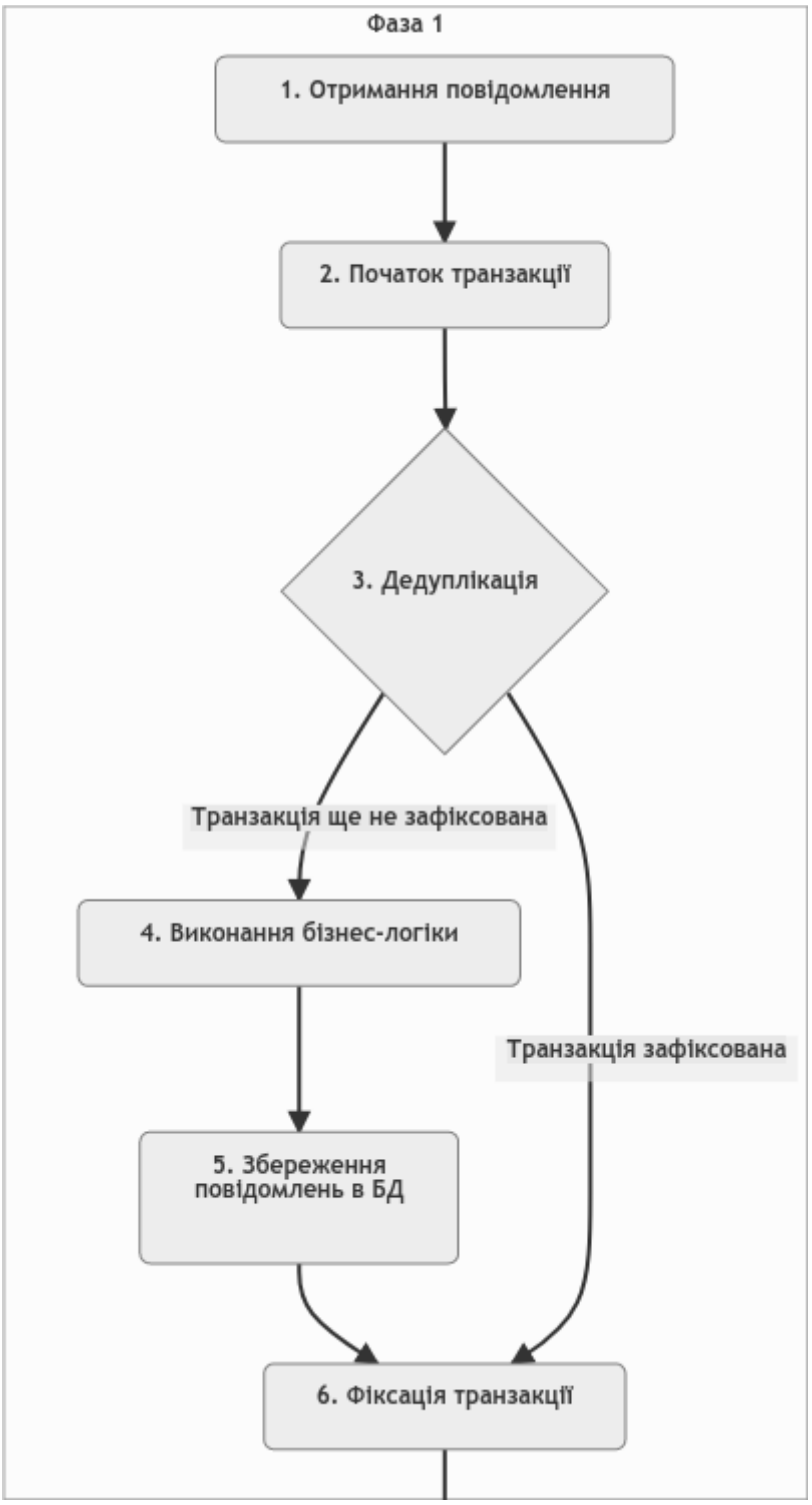

Рис. 3.6. Перша фаза «Outbox»

Алгоритм виконання другої фази «Outbox» (рис. 3.7):

1. Перевіряється, чи вихідні повідомлення вже надіслано. Якщо вихідні повідомлення вже надіслані, це означає, що вхідне повідомлення є дублікатом, тож потрібно перейти до кроку 4. Але якщо вихідні повідомлення ще не надіслано, потрібно перейти до кроку 2.

2. Відправляються вихідні повідомлення в чергу. Якщо на цьому етапі обробка не вдається, дублікати повідомлень можуть бути надіслані в чергу. Дублікати матимуть однаковий ідентифікатор повідомлення, тому їх буде дедубліковано за допомогою «Outbox» (фаза 1, крок 3) у кінцевій точці.

3. Оновлюється сховище «Outbox», фіксуючи факт публікації.

4. Вхідне повідомлення підтверджується (ACK).

Рис. 3.7. Друга фаза «Outbox»

В процесі виконання першої фази вихідні повідомлення не надсилаються. Вони серіалізуються та зберігаються в сховище «Outbox». Це відбувається в одній транзакції (кроки 2–6), яка також включає оновлення бізнес-даних, внесені обробниками повідомлень. Це гарантує, що зміни бізнес-даних не вносяться без запису вихідних повідомлень, і навпаки.

В процесі виконання другої фази вихідні повідомлення надсилаються до інфраструктури обміну повідомленнями, а сховище вихідних повідомлень «Outbox» оновлюється, щоб вказати, що вихідні повідомлення надіслано (кроки 1–3). Через можливі збої одне повідомлення може бути надіслано кілька разів. Наприклад, якщо на кроці 3 виникає збій (не вдалося оновити сховище вихідних повідомлень), повідомлення буде знову прочитано зі сховища вихідних повідомлень і надіслано знову. Поки приймальні кінцеві точки використовують «Outbox», ці дублікати дедуплікуються на 3 кроці.

Вихідні використовують лише ідентифікатор вхідного повідомлення як унікальний ключ для дедуплікації повідомлень. Якщо відправник не використовує «Outbox» під час надсилання повідомлень, він відповідає за те, щоб значення ідентифікатора повідомлення було узгодженим, коли це повідомлення надсилається кілька разів.

3.3. Розподілена ACID-транзакція в MongoDB та PostgreSQL

Транзакції ACID корисні для задоволення потреб розробників у складних сценаріях, які вимагають виконання «всього або нічого» під час роботи з узгодженими даними.

Коли програми працюють із величезними даними, потужності одного сервера з часом може стати недостатньо, оскільки клієнти перевантажують сервер одночасними операціями.

Шардінг — це механізм горизонтального масштабування шляхом розділення даних на частини, а потім розподілу їх між кількома серверами для підтримки великих наборів даних і підвищення продуктивності системи [44].

PostgreSQL реалізував шардінг поверх розділення даних на частини, дозволяючи розміщення будь-якого розділу розділеної таблиці на віддаленому сервері. Основою для цього є Foreign Data Wrapper (FDW), який протягом тривалого часу був частиною ядра PostgreSQL.

Двофазні транзакції PostgreSQL дозволяють базі даних зберігати деталі транзакції на диску без її фіксації. Це робиться шляхом виконання команди PREPARE TRANSACTION в кінці блоку транзакції. Для фіксації транзакції має бути виконана команда COMMIT PREPARED. Оскільки деталі транзакції зберігаються на диску за допомогою двофазних транзакцій, PostgreSQL може гарантувати можливість фіксації двофазної транзакцію пізніше, навіть якщо він виходить з ладу або не працює деякий час.

MongoDB вирішує питання керування великими колекціями безпосередньо через шардінг: у MongoDB немає поняття локального розділення колекцій. Насправді вся стратегія масштабування MongoDB базується на шардінгу, який займає центральне місце в архітектурі бази даних. Таким чином, процес шардінгу є максимально прозорим для програми: все, що повинен зробити адміністратор бази даних, це визначити ключ сегмента. MongoDB надає можливість виконувати розподілені транзакції, які можуть охоплювати весь кластер MongoDB, та дозволяє встановлювати потрібний рівень узгодженості для кожної окремої транзакції [9].

Двофазні транзакції ніколи не потрібні для збереження даних в одне сховище. Але неможливо досягти атомарності з одними лише однофазними транзакціями, якщо потрібно писати дані в декілька сховищ.

MongoDB використовує двофазні транзакції у своїй реалізації розподілених транзакцій. Але, на відміну від PostgreSQL, не надає користувачам подібний до XA інтерфейс. Таке обмеження пов’язане в першу чергу з тим, що архітектурно та концептуально MongoDB не розрахований на довготривалі транзакції, тому що це потребує додаткових песимістичних блокувань, тоді як двофазні транзакції теоретично можуть заблокувати дані навічно.

Завдяки семантиці Apache Kafka, можливо створювати операції, які оркеструють та поєднують в розподілені ACID-транзакції інші операції. Фактично, таким чином можливо отримати надійний, розподілений та відмовостійкий менеджер розподілених транзакцій.

Розподілені транзакції можна використовувати в межах кількох операцій, баз даних та шардів. Можливо обійти обмеження MongoDB на використання двофазних транзакцій, щоб використовувати MongoDB в межах двофазної розподіленої транзакції. Для цього потрібно використати описану оптимізацію розподілених транзакцій в гетерогенних системах, що дозволяє використовувати однофазну транзакцію для одного зі сховищ даних (MongoDB, в даному випадку).

Варто зазначити, що MongoDB використовує консенсусну реплікацію. А оскільки оптимізація, яка дозволяє використовувати транзакцію MongoDB в межах розподіленої транзакції вимагає збереження помітки про фіксацію однофазної транзакції безпосередньо в MongoDB, треба бути обережним із обраним рівнем узгодженості операцій. Потрібно дочекатися фіксації однофазної транзакції консенсусом, перш ніж переходити до фіксації двофазних транзакцій. Відповідно, при відновленні після відмови стан однофазної транзакції повинен також бути узгоджений консенсусом.

3.4. Висновки до третього розділу

В цьому розділі був описаний приклад мікросервісної архітектури на основі асинхронної комунікації через брокер повідомлень. Мікросервіси, які побудовані за цією архітектурою, мають можливість в межах однієї операції атомарно працювати як з реляційними, так і з нереляційними даними, а також взаємодіяти з іншими мікросервісами.

Можуть бути використані різні методи вирішення проблеми подвійного запису для різних операцій, в залежності від того, який рівень узгодженості потрібен для окремої операції. Сувора узгодженість із застосуванням двофазних транзакцій також доступна. Оптимізація двофазних транзакцій, а також

використання шаблону «Outbox», дозволяє зробити подібні операції більш ефективними завдяки позбавленню одразу від 2 двофазних транзакцій.

Використання Apache Kafka в якості брокера повідомлень дозволяє контролювати ступінь паралелізму кожної окремої операції, а також зробити мікросервіси високодоступними завдяки балансуванню навантаження. Використання механізму дедуплікації повідомлень захищає систему від аномалій, які можуть виникати при розподілу мережі.

Семантика підтвердження оброки повідомлень, яку надає Apache Kafka, також дозволила створити специфічну оптимізацію шаблону «Outbox», яка не впливає на ефективність операцій сама по собі, але дозволяє спростити інфраструктуру, позбавивши систему від необхідності підтримки фонових процесів, таких як Debezium, що використовуються в класичних реалізаціях.

Загалом отримана архітектура є доволі простою, але дуже функціональною, оскільки фактично підтримує всі можливі стратегії розподілених транзакцій їх оптимізованих формах.

ВИСНОВКИ

Були описані загальні принципи організації даних та комунікацій в мікросервісній архітектурі. Порівняно вплив організації даних та комунікацій на рівень ізоляції логіки та розподільності даних в контексті теореми CAP.

Описана проблема здійснення подвійного запису, коли операція в мікросервісній системі повинна виходити за межі окремого мікросервісу. Виконано дослідження та порівняння методів вирішення цієї проблеми.

В процесі дослідження проблеми здійснення подвійного запису була розроблена специфічна оптимізація методу двофазних транзакцій. Оптимізація дозволяє зберегти семантику ACID, позбувшись від необхідності використання двофазної транзакції для одного зі сховищ даних, а також атомарно надсилати повідомлення як результат розподіленої транзакції.

В якості прикладу була розроблена універсальна платформа для створення мікросервісних систем на основі Apache Kafka, MongoDB та PostgreSQL, яка дозволяє використовувати будь-який з методів організації даних та комунікацій. Була забезпечена надійність, висока доступність, та здатність до горизонтального масштабування системи.

Проблема забезпечення рівнів ізоляції для розподілених гетерогенних ACID-транзакцій залишається відкритою для подальшого дослідження.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. E. Brewer A certain freedom: thoughts on the CAP theorem / E. Brewer. - in Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, New York, NY, USA. - 2010. - с. 335.

2. S. Nishida і Y. Shinkawa A Comprehensive Evaluation Model for BASE Transaction Processing / S. Nishida і Y. Shinkawa. - presented at the 9th International Conference on Software Engineering and Applications. - 2022. - с. 393–400.

3. S. Kul і A. Sayar A Survey of Publish/Subscribe Middleware Systems for Microservice Communication / S. Kul і A. Sayar. - in 2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). - 2021. - с. 781–785.

4. A. Diepenbrock, F. Rademacher, і S. Sachweh An Ontology-based Approach for Domain-driven Design of Microservice Architectures / . Gesellschaft für Informatik, Bonn, 2017.

5. E. Levy, H. F. Korth, і A. Silberschatz An optimistic commit protocol for distributed transaction management / E. Levy, H. F. Korth, і A. Silberschatz. - in Proceedings of the 1991 ACM SIGMOD international conference on Management of data, New York, NY, USA. - 1991. - с. 88–97.

6. H. Chandra Analysis of Change Data Capture Method in Heterogeneous Data Sources to Support RTDW / H. Chandra. - in 2018 4th International Conference on Computer and Information Sciences (ICCOINS). - 2018. - с. 1–6.

7. A. Karakos, D. Patsas, A. Bornea, і S. Kontogiannis Balancing HTTP Traffic Using Dynamically Updated Weights, an Implementation Approach / A. Karakos, D. Patsas, A. Bornea, і S. Kontogiannis. - in Advances in Informatics, Berlin, Heidelberg. - 2005. - с. 858–868.

8. D. Monteiro, R. Gadelha, P. H. M. Maia, L. S. Rocha, і N. C. Mendonça Beethoven: An Event-Driven Lightweight Platform for Microservice Orchestration / D. Monteiro, R. Gadelha, P. H. M. Maia, L. S. Rocha, і N. C. Mendonça. - in Software Architecture, Cham. - 2018. - с. 191–199.

9. T. Panpaliya Benchmarking MongoDB multi-document transactions in a sharded cluster / , Master of Science, San Jose State University, San Jose, CA, USA, 2020.

10. Y. Chen і Z. He Bounds on the reliability of distributed systems with unreliable nodes & links / Y. Chen і Z. He. - IEEE Transactions on Reliability. - вип. 53, вип. 2, Чер 2004. - с. 205–215.

11. S. Gilbert і N. Lynch Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services / S. Gilbert і N. Lynch. - SIGACT News. - вип. 33, вип. 2, Чер 2002. - с. 51–59.

12. M. A. Fardbastani, F. Allahdadi, і M. Sharifi Business process monitoring via decentralized complex event processing / M. A. Fardbastani, F. Allahdadi, і M. Sharifi. - Enterprise Information Systems. - вип. 12, вип. 10, Лис 2018. - с. 1257–1284.

13. N. Naik Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP / N. Naik. - in 2017 IEEE International Systems Engineering Symposium (ISSE). - 2017. - с. 1–7.

14. C. K. Rudrabhatla Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture / C. K. Rudrabhatla. - International Journal of Advanced Computer Science and Applications (ijacsa). - вип. 9, вип. 8, 49/01 2018. - .

15. D. Abadi Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story / D. Abadi. - Computer. - вип. 45, вип. 2, Лют 2012. - с. 37–42.

16. J. Bauwens, F. Myter, і E. G. Boix Constraining the eventual in eventual consistency / J. Bauwens, F. Myter, і E. G. Boix. - in Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data, New York, NY, USA. - 2018. - с. 1–3.

17. V. Chanron і K. Lewis Convergence and Stability in Distributed Design of Large Systems / V. Chanron і K. Lewis. - presented at the ASME 2004 International

Design Engineering Technical Conferences and Computers and Information in Engineering Conference. - 2008. - c. 593–603.

18. D. Surya Sai Venkatesh i S. Agarwal Data Access Pattern Recommendations for Microservices Architecture / D. Surya Sai Venkatesh i S. Agarwal. - in 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). - 2022. - c. 241–243.

19. T. Zhou i Y. Wei Database replication technology having high consistency requirements / T. Zhou i Y. Wei. - in 2013 IEEE Third International Conference on Information Science and Technology (ICIST). - 2013. - c. 793–797.

20. G. Sharma i C. Busch Distributed transactional memory for general networks / G. Sharma i C. Busch. - Distrib. Comput. - вип. 27, вип. 5, Жов 2014. - c. 329–362.

21. S. Kapferer i O. Zimmermann Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling / S. Kapferer i O. Zimmermann. - presented at the 8th International Conference on Model-Driven Engineering and Software Development. - 2022. - c. 299–306.

22. Y. J. Singh, Y. S. Singh, A. Gaikwad, i S. C. Mehrotra Dynamic management of transactions in distributed real-time processing system / Y. J. Singh, Y. S. Singh, A. Gaikwad, i S. C. Mehrotra. - IJDMS. - вип. 2, вип. 2, Трав 2010. - c. 161–170.

23. R. Laursen Ensuring Eventual Consistency in a Microservices Architecture / R. Laursen. - 2021. - .

24. K. Munonye i P. Martinek Evaluation of Data Storage Patterns in Microservices Archicture / K. Munonye i P. Martinek. - in 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE). - 2020. - c. 373–380.

25. E. Sherratt i A. Prinz Eventual Consistency Formalized / E. Sherratt i A. Prinz. - in System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Cham. - 2019. - c. 249–265.

26. S. Bhola, R. Strom, S. Bagchi, Y. Zhao, i J. Auerbach Exactly-once delivery in a content-based publish-subscribe system / S. Bhola, R. Strom, S. Bagchi, Y. Zhao, i J. Auerbach. - in Proceedings International Conference on Dependable Systems and Networks. - 2002. - c. 7–16.

27. Y. Huang i H. Garcia-Molina Exactly-once semantics in a replicated messaging system / Y. Huang i H. Garcia-Molina. - in Proceedings 17th International Conference on Data Engineering. - 2001. - c. 3–12.

28. P. Cai, J. Guo, H. Zhou, W. Qian, i A. Zhou Fast Raft Replication for Transactional Database Systems over Unreliable Networks / P. Cai, J. Guo, H. Zhou, W. Qian, i A. Zhou. - in Database Systems for Advanced Applications, Cham. - 2019. - c. 461–465.

29. R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, i Y. Zhou From a Monolithic Big Data System to a Microservices Event-Driven Architecture / R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, i Y. Zhou. - in 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). - 2020. - c. 213–220.

30. M. A. Shah, J. M. Hellerstein, i E. Brewer Highly available, fault-tolerant, parallel dataflows / M. A. Shah, J. M. Hellerstein, i E. Brewer. - in Proceedings of the 2004 ACM SIGMOD international conference on Management of data, New York, NY, USA. - 2004. - c. 827–838.

31. D. Richter, M. Konrad, K. Utecht, i A. Polze Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice / D. Richter, M. Konrad, K. Utecht, i A. Polze. - in 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). - 2017. - c. 130–137.

32. C. Deyerl i T. Distler In Search of a Scalable Raft-based Replication Architecture / C. Deyerl i T. Distler. - in Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data, New York, NY, USA. - 2019. - c. 1–7.

33. M. Burgess Locality, Statefulness, and Causality in Distributed Information Systems (Concerning the Scale Dependence Of System Promises) / . arXiv, 20-Bep-2019.

34. S. Gilbert i W. Golab Making Sense of Relativistic Distributed Systems / S. Gilbert i W. Golab. - in Distributed Computing, Berlin, Heidelberg. - 2014. - c. 361–375.

35. L. Frank i R. U. Pedersen Managing consistency anomalies in distributed integrated databases with relaxed ACID properties / L. Frank i R. U. Pedersen. - in Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication, New York, NY, USA. - 2014. - c. 1–7.

36. S. Braun Mastering Eventual Consistency / S. Braun. - in 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). - 2022. - c. 237–239.

37. G. Chockler i A. Gotsman Multi-shot distributed transaction commit / G. Chockler i A. Gotsman. - Distrib. Comput. - вип. 34, вип. 4, Сер 2021. - c. 301–318.

38. T. P. Raptis i A. Passarella On Efficiently Partitioning a Topic in Apache Kafka / T. P. Raptis i A. Passarella. - in 2022 International Conference on Computer, Information and Telecommunication Systems (CITS). - 2022. - c. 1–8.

39. D. Dedousis, N. Zacheilas, i V. Kalogeraki On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems / D. Dedousis, N. Zacheilas, i V. Kalogeraki. - in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). - 2018. - c. 76–86.

40. R. Sawarkar i M. M. Baig Performance Tuning of Queries in Distributed System using Secure Materialized View Approach / R. Sawarkar i M. M. Baig. - in 2020 International Conference on Smart Innovations in Design, Environment, Management, Planning and Computing (ICSIDEMPC). - 2020. - c. 216–218.

41. D. H. Steves, C. Edmondson-Yurkanan, i M. Gouda Properties of secure transaction protocols / D. H. Steves, C. Edmondson-Yurkanan, i M. Gouda. - Computer Networks and ISDN Systems. - вип. 29, вип. 15, Лис 1997. - c. 1809–1821.

42. W. Golab Proving PACELC / W. Golab. - SIGACT News. - вип. 49, вип. 1, Бер 2018. - с. 73–81.

43. D. G. Messerschmitt Relativistic Timekeeping, Motion, and Gravity in Distributed Systems / D. G. Messerschmitt. - Proceedings of the IEEE. - вип. 105, вип. 8, Сер 2017. - с. 1511–1573.

44. B. M. Abdelhafiz і M. Elhadef Sharding Database for Fault Tolerance and Scalability of Data / B. M. Abdelhafiz і M. Elhadef. - in 2021 2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM). - 2021. - с. 17–24.

45. F. Jammes, B. Bony, P. Nappey, A. W. Colombo, J. Delsing, J. Eliasson, R. Kyusakov, S. Karnouskos, P. Stluka, і M. Till Technologies for SOA-based distributed large scale process monitoring and control systems / F. Jammes, B. Bony, P. Nappey, A. W. Colombo, J. Delsing, J. Eliasson, R. Kyusakov, S. Karnouskos, P. Stluka, і M. Till. - in IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society. - 2012. - с. 5799–5804.

46. L. Lamport Time, clocks, and the ordering of events in a distributed system / L. Lamport. - Commun. ACM. - вип. 21, вип. 7, Лип 1978. - с. 558–565.

47. D. Patel, F. Khasib, I. Sadooghi, і I. Raicu Towards In-Order and Exactly-Once Delivery Using Hierarchical Distributed Message Queues / D. Patel, F. Khasib, I. Sadooghi, і I. Raicu. - in 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. - 2014. - с. 883–892.

48. E. A. Brewer Towards robust distributed systems (abstract) / E. A. Brewer. - in Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC ’00, Portland, Oregon, United States. - 2000. - с. 7.

49. G. Samaras, K. Britton, A. Citron, і C. Mohan Two-phase commit optimizations and tradeoffs in the commercial environment / G. Samaras, K. Britton, A. Citron, і C. Mohan. - 1993. - с. 520–529.

50. Y. Zhong, W. Li, і J. Wang Using Event Sourcing and CQRS to Build a High Performance Point Trading System / Y. Zhong, W. Li, і J. Wang. - in Proceedings

of the 2019 5th International Conference on E-Business and Applications, New York, NY, USA. - 2019. - c. 16–19.

51. F. Nawab Weaker Consistency Models/Eventual Consistency / , in Encyclopedia of Big Data Technologies, S. Sakr i A. Zomaya, Ред. Cham: Springer International Publishing, 2018, c. 1–7.

ДОДАТОК А. ЛІСТІНГ ПРОГРАМИ

```
public class KafkaMessageReceiver<TKey, TValue> :
        Agent,
        IKafkaMessageReceiver<TKey, TValue>
        where TValue : class
    {
        readonly CancellationTokenSource _cancellationTokenSource;
        readonly ConsumerContext<TKey, TValue> _consumerContext;
        readonly ReceiveEndpointContext _context;
        readonly TaskCompletionSource<bool> _deliveryComplete;
        readonly IReceivePipeDispatcher _dispatcher;

        public KafkaMessageReceiver(ReceiveEndpointContext context, ConsumerContext<TKey, TValue>
consumerContext)
        {
            _context = context;
            _consumerContext = consumerContext;
            _cancellationTokenSource = CancellationTokenSource.CreateLinkedTokenSource(Stopping);

            _consumerContext.ErrorHandler += HandleKafkaError;

            _deliveryComplete = TaskUtil.GetTask<bool>();

            _dispatcher = context.CreateReceivePipeDispatcher();
            _dispatcher.ZeroActivity += HandleDeliveryComplete;

            Task.Run(Consume);
        }

        public long DeliveryCount => _dispatcher.DispatchCount;

        public int ConcurrentDeliveryCount => _dispatcher.MaxConcurrentDispatchCount;

        async Task Consume()
        {
            var executor = new ChannelExecutor(_consumerContext.ReceiveSettings.PrefetchCount,
_consumerContext.ReceiveSettings.ConcurrencyLimit);

            await _consumerContext.Subscribe().ConfigureAwait(false);

            SetReady();

            try
            {
                while (!_cancellationTokenSource.IsCancellationRequested)
                {
                    ConsumeResult<TKey, TValue> consumeResult = await
_consumerContext.Consume(_cancellationTokenSource.Token).ConfigureAwait(false);
                    await executor.Push(() => Handle(consumeResult),
Stopping).ConfigureAwait(false);
                }

                SetCompleted(Task.CompletedTask);
            }
            catch (OperationCanceledException exception) when (exception.CancellationToken ==
Stopping
                                                               || exception.CancellationToken ==
_cancellationTokenSource.Token)
            {
                SetCompleted(Task.CompletedTask);
            }
            catch (Exception exception)
            {
                LogContext.Error?.Log(exception, "Consume Loop faulted");

                SetCompleted(TaskUtil.Faulted<bool>(exception));
            }
            finally
            {
                await executor.DisposeAsync().ConfigureAwait(false);
            }
        }

        async Task Handle(ConsumeResult<TKey, TValue> result)
        {
```

```
            if (IsStopping)
                return;

            var context = new KafkaReceiveContext<TKey, TValue>(result, _context, _consumerContext,
_consumerContext.HeadersDeserializer);

            try
            {
                await _dispatcher.Dispatch(context, context).ConfigureAwait(false);
            }
            catch (Exception exception)
            {
                context.LogTransportFaulted(exception);
            }
            finally
            {
                context.Dispose();
            }
        }

        void HandleKafkaError(IConsumer<TKey, TValue> consumer, Error error)
        {
            EnabledLogger? logger = error.IsFatal ? LogContext.Error : LogContext.Warning;
            logger?.Log("Consumer error ({Code}): {Reason} on {Topic}", error.Code, error.Reason,
_consumerContext.ReceiveSettings.Topic);

            if (_cancellationTokenSource.IsCancellationRequested)
                return;

            var activeDispatchCount = _dispatcher.ActiveDispatchCount;
            if (activeDispatchCount == 0 || error.IsLocalError)
            {
                _cancellationTokenSource.Cancel();
                _deliveryComplete.TrySetResult(true);
                SetCompleted(TaskUtil.Faulted<bool>(new KafkaException(error)));
            }
        }

        async Task HandleDeliveryComplete()
        {
            if (IsStopping)
            {
                LogContext.Debug?.Log("Consumer shutdown completed: {InputAddress}",
_context.InputAddress);

                _deliveryComplete.TrySetResult(true);
            }
        }

        protected override async Task StopAgent(StopContext context)
        {
            await _consumerContext.Close().ConfigureAwait(false);

            _consumerContext.ErrorHandler -= HandleKafkaError;

            LogContext.Debug?.Log("Stopping consumer: {InputAddress}", _context.InputAddress);

            SetCompleted(ActiveAndActualAgentsCompleted(context));

            await Completed.ConfigureAwait(false);
        }

        async Task ActiveAndActualAgentsCompleted(StopContext context)
        {
            if (_dispatcher.ActiveDispatchCount > 0)
            {
                try
                {
                    await
_deliveryComplete.Task.OrCanceled(context.CancellationToken).ConfigureAwait(false);
                }
                catch (OperationCanceledException)
                {
                    LogContext.Warning?.Log("Stop canceled waiting for message consumers to
complete: {InputAddress}", _context.InputAddress);
                }
            }
        }
    }
```

```
public interface OutboxMessageContext :
    MessageContext
{
    long SequenceNumber { get; }

    new Guid MessageId { get; }

    string ContentType { get; }

    string Body { get; }

    IReadOnlyDictionary<string, object> Properties { get; }
}

public class OutboxSendEndpoint :
    ITransportSendEndpoint
{
    readonly ITransportSendEndpoint _endpoint;
    readonly OutboxSendContext _outboxContext;

    /// <summary>
    /// Creates an send endpoint on the outbox
    /// </summary>
    /// <param name="outboxContext">The outbox context for this consume operation</param>
    /// <param name="endpoint"></param>
    public OutboxSendEndpoint(OutboxSendContext outboxContext, ISendEndpoint endpoint)
    {
        _outboxContext = outboxContext;
        _endpoint = endpoint as ITransportSendEndpoint ?? throw new ArgumentException("Must be a
transport endpoint", nameof(endpoint));
    }

    public ISendEndpoint Endpoint => _endpoint;

    public ConnectHandle ConnectSendObserver(ISendObserver observer)
    {
        return new EmptyConnectHandle();
    }

    public Task<SendContext<T>> CreateSendContext<T>(T message, IPipe<SendContext<T>> pipe,
CancellationToken cancellationToken)
        where T : class
    {
        return _endpoint.CreateSendContext(message, new OutboxSendEndpointPipe<T>(pipe),
cancellationToken);
    }

    public async Task Send<T>(T message, CancellationToken cancellationToken)
        where T : class
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));

        SendContext<T> context = await _endpoint.CreateSendContext(message, new
OutboxSendEndpointPipe<T>(), cancellationToken).ConfigureAwait(false);

        await AddSend(context).ConfigureAwait(false);
    }

    public async Task Send<T>(T message, IPipe<SendContext<T>> pipe, CancellationToken
cancellationToken)
        where T : class
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));
        if (pipe == null)
            throw new ArgumentNullException(nameof(pipe));

        SendContext<T> context = await _endpoint.CreateSendContext(message, new
OutboxSendEndpointPipe<T>(pipe), cancellationToken).ConfigureAwait(false);

        await AddSend(context).ConfigureAwait(false);
    }

    public Task Send(object message, CancellationToken cancellationToken)
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));
```

```
        var messageType = message.GetType();

        return SendEndpointConverterCache.Send(this, message, messageType, cancellationToken);
    }

    public Task Send(object message, Type messageType, CancellationToken cancellationToken)
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));
        if (messageType == null)
            throw new ArgumentNullException(nameof(messageType));

        return SendEndpointConverterCache.Send(this, message, messageType, cancellationToken);
    }

    public async Task Send<T>(T message, IPipe<SendContext> pipe, CancellationToken
cancellationToken)
        where T : class
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));
        if (pipe == null)
            throw new ArgumentNullException(nameof(pipe));

        SendContext<T> context = await _endpoint.CreateSendContext(message, new
OutboxSendEndpointPipe<T>(pipe), cancellationToken).ConfigureAwait(false);

        await AddSend(context).ConfigureAwait(false);
    }

    public Task Send(object message, IPipe<SendContext> pipe, CancellationToken cancellationToken)
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));
        if (pipe == null)
            throw new ArgumentNullException(nameof(pipe));

        var messageType = message.GetType();

        return SendEndpointConverterCache.Send(this, message, messageType, pipe, cancellationToken);
    }

    public Task Send(object message, Type messageType, IPipe<SendContext> pipe, CancellationToken
cancellationToken)
    {
        if (message == null)
            throw new ArgumentNullException(nameof(message));
        if (messageType == null)
            throw new ArgumentNullException(nameof(messageType));
        if (pipe == null)
            throw new ArgumentNullException(nameof(pipe));

        return SendEndpointConverterCache.Send(this, message, messageType, pipe, cancellationToken);
    }

    public async Task Send<T>(object values, CancellationToken cancellationToken)
        where T : class
    {
        if (values == null)
            throw new ArgumentNullException(nameof(values));

        (var message, IPipe<SendContext<T>> sendPipe) =
            await MessageInitializerCache<T>.InitializeMessage(values, new
OutboxSendEndpointPipe<T>(), cancellationToken).ConfigureAwait(false);

        SendContext<T> context =
            await _endpoint.CreateSendContext(message, new OutboxSendEndpointPipe<T>(sendPipe),
cancellationToken).ConfigureAwait(false);

        await AddSend(context).ConfigureAwait(false);
    }

    public async Task Send<T>(object values, IPipe<SendContext<T>> pipe, CancellationToken
cancellationToken)
        where T : class
    {
        if (values == null)
            throw new ArgumentNullException(nameof(values));
```

```
        (var message, IPipe<SendContext<T>> sendPipe) =
            await MessageInitializerCache<T>.InitializeMessage(values, new
OutboxSendEndpointPipe<T>(pipe), cancellationToken).ConfigureAwait(false);

        SendContext<T> context =
            await _endpoint.CreateSendContext(message, new OutboxSendEndpointPipe<T>(sendPipe),
cancellationToken).ConfigureAwait(false);

        await AddSend(context).ConfigureAwait(false);
    }

    public async Task Send<T>(object values, IPipe<SendContext> pipe, CancellationToken
cancellationToken)
        where T : class
    {
        if (values == null)
            throw new ArgumentNullException(nameof(values));
        if (pipe == null)
            throw new ArgumentNullException(nameof(pipe));

        (var message, IPipe<SendContext<T>> sendPipe) =
            await MessageInitializerCache<T>.InitializeMessage(values, new
OutboxSendEndpointPipe<T>(pipe), cancellationToken).ConfigureAwait(false);

        SendContext<T> context =
            await _endpoint.CreateSendContext(message, new OutboxSendEndpointPipe<T>(sendPipe),
cancellationToken).ConfigureAwait(false);

        await AddSend(context).ConfigureAwait(false);
    }

    async Task AddSend<T>(SendContext<T> context)
        where T : class
    {
        StartedActivity? activity = LogContext.Current?.StartOutboxSendActivity(context);
        try
        {
            await _outboxContext.AddSend(context).ConfigureAwait(false);
            activity?.Update(context);
        }
        catch (Exception ex)
        {
            activity?.AddExceptionEvent(ex);
            throw;
        }
        finally
        {
            activity?.Stop();
        }
    }


    class OutboxSendEndpointPipe<T> :
        IPipe<SendContext<T>>
        where T : class
    {
        readonly IPipe<SendContext<T>> _pipe;
        readonly ISendContextPipe _sendContextPipe;

        public OutboxSendEndpointPipe()
        {
            _pipe = default;
            _sendContextPipe = default;
        }

        public OutboxSendEndpointPipe(IPipe<SendContext<T>> pipe)
        {
            _pipe = pipe;
            _sendContextPipe = pipe as ISendContextPipe;
        }

        public void Probe(ProbeContext context)
        {
            _pipe?.Probe(context);
        }

        public async Task Send(SendContext<T> context)
        {
            if (_sendContextPipe != null)
```

```
                await _sendContextPipe.Send(context).ConfigureAwait(false);

            if (_pipe.IsNotEmpty())
                await _pipe.Send(context).ConfigureAwait(false);

            context.ConversationId ??= NewId.NextGuid();
        }
    }
}


public class TransactionMongoDbContext :
    MongoDbContext
{
    static readonly ClientSessionOptions _sessionOptions = new ClientSessionOptions
    {
        DefaultTransactionOptions = new TransactionOptions(ReadConcern.Majority,
ReadPreference.Primary, WriteConcern.WMajority)
    };

    readonly IMongoClient _client;
    readonly object _lock = new object();
    readonly IServiceProvider _provider;

    Task<IClientSessionHandle>? _session;

    public TransactionMongoDbContext(IMongoClient client, IServiceProvider provider)
    {
        _client = client;
        _provider = provider;
    }

    public IClientSessionHandle? Session { get; private set; }

    public void Dispose()
    {
        Session?.Dispose();
    }

    public Task<IClientSessionHandle> StartSession(CancellationToken cancellationToken)
    {
        async Task<IClientSessionHandle> Start()
        {
            var handle = await _client.StartSessionAsync(_sessionOptions,
cancellationToken).ConfigureAwait(false);

            Session = handle;

            return handle;
        }

        lock (_lock)
        {
            if (_session != null)
                return _session;

            _session = Start();

            return _session;
        }
    }

    public async Task BeginTransaction(CancellationToken cancellationToken)
    {
        var session = await StartSession(cancellationToken).ConfigureAwait(false);

        if (!session.IsInTransaction)
            session.StartTransaction();
    }

    public Task CommitTransaction(CancellationToken cancellationToken)
    {
        if (Session == null)
            throw new InvalidOperationException("No session has been created");

        if (Session.IsInTransaction == false)
            throw new InvalidOperationException("The session is not in an active transaction");

        return Session.CommitTransactionAsync(cancellationToken);
```

```
    }

    public Task AbortTransaction(CancellationToken cancellationToken)
    {
        if (Session == null)
            throw new InvalidOperationException("No session has been created");

        if (Session.IsInTransaction == false)
            throw new InvalidOperationException("The session is not in an active transaction");

        return Session.AbortTransactionAsync(cancellationToken);
    }

    public MongoDbCollectionContext<T> GetCollection<T>()
    {
        var collection = _provider.GetRequiredService<IMongoCollection<T>>();

        return new TransactionMongoDbCollectionContext<T>(this, collection);
    }
}
```

Додаток Б.

ВІДГУК КЕРІВНИКА
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Мещерякова Леоніда Івановича, д.т.н., проф. каф. ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

на магістерську роботу

студента Грушко Єгора Олександровича
(прізвище, ім'я, по батькові)

курсу І групи 122м-21-2
спеціальності 122 Комп'ютерні науки
освітньої програми «122 Комп'ютерні науки»
на тему Дослідження методів забезпечення консистентності даних та комунікації в мікросервісній архітектурі

Актуальність теми Розглянута магістерська кваліфікаційна робота присвячена методам забезпечення узгодженого стану мікросервісної системи, яка має підтримувати горизонтальне масштабування. Узгоджений стан є критичним для коректної роботи розподілених алгоритмів, але існують різні методи забезпечення узгодженості, тож є важливим систематизувати та порівнти ці методи. На сьогодняшній день мікросервісна архітектура активно використовується в корпоративних системах завдяки можливості підтримувати теоретично необмежену кількість користувачів, таким чином магістерська робота відзначається актуальністю.

Мета досліджень Полягає в створенні горизонтально масштабованої, надійної та відмовостійкої архітектури мікросервісних систем.

Коротка характеристика розділів роботи Перший розділ роботи складається з аналітичного розбору існуючих методів загальної організації даних та комунікації в мікросервісних системах. Другий розділ містить огляд проблеми подвійного запису в гетерогенних мікросервісних системах, порівнює існуючі методи вирішення цієї проблеми. Третій розділ присвячено програмній реалізації мікросервісної системи, враховуючи досліджені проблеми.

Практичне значення роботи Отримані результати роботи є актуальними у проектуванні мікросервісних систем, які мають підтримувати узгоджений стан. Також надані результати дослідження є підставою для більш поглибленого вивчення проблеми ізоляції розподілених гетерогенних транзакцій.

Зауваження та недоліки В роботі недостатньо повно оглянуто реалізацію програмного інтерфейсу користувача, а також процес керування інфраструктурою мікросервісних систем.

Висновки та оцінка Магістром було проведено порівняльний аналіз існуючих методів забезпечення узгодженого стану мікросервісної системи, виявлено недоліки та реалізовано програмний комплекс для вирішення досліджених проблем. Під час виконання магістерської кваліфікаційної роботи студентк Грушко Є.О. постав кваліфікованим та впевненим спеціалістом, який знаходить оптимальні рішення у складних технічних питаннях. Вважаю, що магістерська кваліфікаційна робота заслуговує оцінку «_____», а Грушко Є.О. – присвоєння кваліфікації «магістра» по спеціальності комп'ютерних наук.

Науковий керівник Мещеряков Л.І., док. техн. наук, проф., проф. каф. ПЗКС

(прізвище, ім'я, по батькові, посада, місце роботи)

«___»_____20__ р. _____

(підпис)

ДОДАТОК В.

РЕЦЕНЗІЯ

на магістерську роботу

студента Грушко Єгора Олександровича

(прізвище, ім'я, по батькові)

курсу І групи _122м-21-2___

кафедри програмного забезпечення комп'ютерних систем

спеціальності 122 Комп'ютерні науки

освітньої програми «122 Комп'ютерні науки»

Тема роботи Дослідження методів забезпечення консистентності даних та комунікації в мікросервісній архітектурі

Стисла характеристика розділів роботи Перший розділ роботи складається з аналітичного розбору існуючих методів загальної організації даних та комунікації в мікросервісних системах. Другий розділ містить огляд проблеми подвійного запису в гетерогенних мікросервісних системах, порівнює існуючі методи вирішення цієї проблеми. Третій розділ присвячено програмній реалізації мікросервісної системи, враховуючи досліджені проблеми.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування В даній роботі студентом було надано декілька обґрунтованих пропозицій для вирішення поставлених завдань, кожна з яких була актуальною та підкріпленою науковими даними.

Практичне значення роботи Отримані результати роботи є актуальними у проектуванні мікросервісних систем, які мають підтримувати узгоджений стан. Також надані результати дослідження є підставою для більш поглибленого вивчення проблеми ізоляції розподілених гетерогенних транзакцій.

Якість оформлення роботи Надана на рецензію магістерська кваліфікаційна робота виконана у повному обсязі та у встановлений термін. Проведене дослідження добре проілюстрованою та логічно структурованою. В процесі

виконання роботи було подано основну суть проблеми та проаналізовані різні існуючі методи для її вирішення.

Недоліки в роботі розроблений програмний інтерфейс потребує більш детального опису, проте це не впливає на функціонал на надійність створеного програмного продукту.

Загальний висновок Магістерська кваліфікаційна робота виконана у

(підготовленість студента до самостійної роботи як спеціаліста)

відповідності з завданням із дотриманням всіх вимог.

Оцінка магістерської роботи Робота заслуговує на оцінку « », а студент Грушко Є.О. на присвоєння кваліфікації «магістра» з комп'ютерних наук.

Рецензент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»_____20__ р. _____

(підпис)

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Грушко.docx	Пояснювальна записка роботи. Документ Word.
Диплом_Грушко.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.zip	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Грушко.pptx	Презентація роботи