Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра інформатики

Кваліфікаційна робота

Освітній ступінь – бакалавр

на тему: **«ПРОБЛЕМА УЗГОДЖЕНОСТІ ДАНИХ У РОЗПОДІЛЕНИХ БАЗАХ ДАНИХ»**

Виконала студентка 4-го року навчання спеціальності
121 Інженерія програмного забезпечення
Золотар Анастасія Сергіївна

Керівник: Яремко С.А., ст. викладач

Рецензент _____

Кваліфікаційна робота захищена з оцінкою
_____

Секретар ЕК _____

«____» _____ 2024 р.

Київ – 2024

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

доцент, кандидат наук

_____ Гороховський С. С.

«____» _____ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

для кваліфікаційної роботи

студентці 4-го курсу факультету інформатики

Золотар Анастасії Сергіївні

Тема: “Проблема узгодженості даних у розподілених базах даних”

Зміст:

Вступ

1. Теоретичні відомості

2. Стратегії вирішення проблеми узгодженості даних

3. Реалізація розподіленого сховища даних

Висновок

Дата видачі «____» _____ 2024 р.

Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання роботи

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника. Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи.	жовтень	
2.	Вивчення джерел літератури, збір та узагальнення фактів, даних.	жовтень – грудень	
3.	Складання плану кваліфікаційної роботи та узгодження з науковим керівником.	листопад	
4.	Написання першого розділу роботи.	грудень	
5.	Написання другого розділу роботи.	січень	
6.	Проєктування та реалізація практичної частини роботи.	січень – квітень	
7.	Написання третього розділу роботи.	квітень	
8.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника.	квітень	
9.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику.	початок травня	
10.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності.	17 травня	
11.	Подання на зовнішню рецензію.	середина травня	
12.	Підготовка до захисту кваліфікаційної роботи на засіданні кафедри: написання доповіді та виготовлення ілюстративного матеріалу.	до __ травня	
13.	Попередній захист кваліфікаційної роботи на засіданні кафедри.	до __ травня	
14.	Подання кваліфікаційної роботи на кафедру з усіма супроводжувальними документами.	до __ травня	
15.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією.	згідно з розкладом роботи ЕК	

Графік узгоджено «__» _____ 20__ р.

Виконавець кваліфікаційної роботи: Золотар Анастасія Сергіївна

Науковий керівник: Яремко Соломія Андріївна

Зміст

Анотація

Робота присвячена дослідженню методів досягнення узгодженості даних в розподілених базах даних. Наведено опис та порівняння алгоритмів розподіленого консенсусу Paxos і Raft, а також зіставлено протоколи двофазної (2PC) та трифазної (3PC) фіксації. На основі власних реалізацій Raft і 2PC розроблено розподілене сховище “ключ-значення”.

Вступ

Стрімкий розвиток мережі Інтернет призвів до зростання потреби в постійно доступних і здатних до масштабування застосунках. Основними способами втілення цих двох властивостей є забезпечення відмовостійкості шляхом створення надлишковості компонентів і розподілення навантаження між декількома вузлами.

Розподілені системи складаються з множини процесів, комунікація між якими відбувається за допомогою мережі. Попри те, що метою таких систем є вирішення проблем відмовостійкості та паралелізму, вони також ставлять перед розробниками нові виклики, пов’язані з підтримкою узгодженості, частковими відмовами, ненадійними мережами та ін.

База даних є ключовим елементом будь-якої інформаційної системи, тому питання про забезпечення її надійності та ефективності постає особливо гостро. Більшість сучасних баз даних використовують декілька вузлів, об’єднаних у кластери, з метою підвищення продуктивності та доступності, а також збільшення ємності сховища. Для досягнення узгодженості між даними, розташованими на різних вузлах, використовуються розподілені алгоритми, які здатні працювати попри часткові відмови та ненадійність мереж.

Актуальність дослідження зумовлена великим попитом на масштабовані та доступні застосунки, здатні працювати з великими обсягами даних та гарантувати їх цілісність і достовірність.

Мета дослідження: здійснити порівняльний аналіз алгоритмів розподіленого консенсусу та протоколів атомарної фіксації; розробити розподілене сховище на основі обраних рішень.

Об’єкт дослідження: методи досягнення узгодженості у розподілених базах даних.

Предмет дослідження: алгоритми розподіленого консенсусу та протоколи атомарної фіксації.

Текстова частина кваліфікаційної роботи складається з трьох розділів.

У першому розділі представлено основні теоретичні відомості, пов’язані з розподіленими базами даних: наведено поняття горизонтальної масштабованості, реплікації та секціонування; описано різні моделі узгодженості даних; розглянуто особливості виконання розподілених транзакцій.

Другий розділ присвячений стратегіям досягнення узгодженості у розподілених базах даних, а саме алгоритмам розподіленого консенсусу Paxos і Raft, а також протоколам двофазної та трифазної фіксації.

Третій розділ містить детальний опис реалізації розподіленого сховища “ключ-значення” на основі алгоритму Raft і протоколу двофазної фіксації.

Розділ 1. Теоретичні відомості

1.1. Горизонтальне масштабування баз даних

Існує декілька причин, які зумовлюють необхідність використання розподілених систем: масштабованість, доступність і мінімізація затримки.

Горизонтальна масштабованість полягає у збільшенні пропускної здатності (або навантаження) системи завдяки залученню більшої кількості незалежних вузлів – комп'ютерів або віртуальних машин, з'єднаних за допомогою мережі.

Архітектура без використання спільних ресурсів передбачає, що всі вузли мають свої окремі процесори, пам'ять та диски.

Одним із підходів розподілення даних є секціонування: кожна машина зберігає лише підмножину записів, які в сукупності становлять одну велику базу даних. Основною метою секціонування є масштабованість, яка досягається завдяки розподіленню даних на різні диски і запитів на різні процесори.

Утім для підвищення доступності системи вузли можуть зберігати та підтримувати повні копії даних – репліки. За такого підходу програма продовжуватиме стабільну роботу попри збій або цілковите від'єднання одного або декількох вузлів. Секціонування і реплікацію можна одночасно застосовувати в одній системі, тобто зберігати декілька копій кожної з секцій.

Реплікація також використовується, щоб зберігати дані географічно близько до різних груп користувачів з метою мінімізації затримки й очікування мережевих пакетів.

Основною проблемою реплікації є необхідність підтримання узгодженості копій. Тобто під час оновлення однієї репліки, ми маємо гарантувати, що аналогічне оновлення відбудеться й на інших вузлах. Складність полягає в тому, що для цього потрібна чітка синхронізація між усіма копіями даних, що є досить нетривіальним завданням.

Отож, з одного боку, ми намагаємося масштабувати застосунок й зменшити час затримки для покращення продуктивності, а з іншого – забезпечення узгодженості між вузлами потребує значних витрат, які впливають на швидкодію системи.

Вирішення цієї дилеми полягає у компромісі між узгодженістю й ефективністю. Інакше кажучи, послаблюючи гарантії того, що кожне оновлення має виконуватися атомарно, ми в такий спосіб підвищуємо продуктивність. Ціна такого компромісу визначатиметься тим, наскільки в певний момент часу дані на різних вузлах будуть однаковими. Існує низка моделей узгодженості даних, які надають різні гарантії.

Однак під час проєктування розподіленого сховища даних вибір доводиться робити не лише між узгодженістю й продуктивністю. Згідно з відомою теоремою CAP [1], у будь-якій реалізації розподіленої системи не можна одночасно забезпечити узгодженість, доступність і стійкість до фрагментації. В одній системі можуть виконуватися лише дві з трьох властивостей. Утім варто зазначити, що теорема розглядає лише один варіант узгодженості (лінеаризовність) і один варіант збою – розділення мережі (network partition), через що є думка, що вона малопридатна для застосування на практиці [2, с. 337].

1.2. Моделі узгодженості

Моделі узгодженості можна розглядати як домовленість щодо того, чого варто очікувати користувачам під час виконання операцій запису або читання даних. Кожна модель описує те, наскільки сильно поведінка системи відрізняється від очікуваної поведінки, що дає змогу відрізняти “всі можливі варіанти послідовності виконання операцій” від “варіантів, допустимих відповідно до деякої моделі”, що значно спрощує розгляд питання про видимість зміни станів [3, с. 222].

Лінеаризовність (linearizability)

Лінеаризовність передбачає, що результати запису стають доступними для всіх операцій читання в один момент часу. Це найбільш сильна модель узгодженості,

яку можна імплементувати на практиці. При цьому лінеаризовність може допускати декілька варіантів впорядкування операцій [4].

Попри те, що операції виконуються паралельно і можуть накладатися один на одного, їхні результати стають доступними в такий спосіб, що вони видаються послідовними. Інакше кажучи, жодна подія не виконується миттєво, але для користувачів виглядає як атомарна операція.

Послідовна узгодженість (sequential consistency)

Модель послідовної узгодженості була вперше запропонована Леслі Лампортом в контексті спільної пам'яті для багатопроцесорних систем [5]. Згідно з цією моделлю, операції запису не обов'язково мають бути видними для всіх вузлів миттєво, однак записи, виконані різними процесами, всі вузли повинні бачити в одному порядку.

Варто звернути увагу, що в моделі послідовної узгодженості ніяк не згадуються ані поняття останньої операції, ані час взагалі.

Послідовна узгодженість може давати недетерміновані результати. Це пояснюється тим, що послідовність таких операцій між процесорами може відрізнятися під час різних запусків програми.

Причинна узгодженість (causal consistency)

В моделі причинної узгодженості розрізняються записи, які потенційно можуть мати причинно-наслідковий зв'язок, і незалежні операції. Сховище даних вважають причинно узгодженим, якщо записи, між якими є потенційним причинно-наслідковий зв'язок, розглядаються усіма процесами в одному порядку [6, с. 401]. Модель причинної узгодженості була визначена Філліпом Хатто [7].

Узгодженість у кінцевому підсумку (eventual consistency)

Узгодженість у кінцевому підсумку означає, що оновлення в системі виконуються асинхронно. Така модель узгодженості допускає, що стани реплік тимчасово будуть відрізнятися.

Узгодженості в кінцевому підсумку часто віддають перевагу в розподілених системах, оскільки вона забезпечує більшу доступність і зменшує затримку, адже оновлення можуть бути застосовані локально без очікування синхронізації з іншими репліками.

1.3. Розподілені транзакції

Транзакція – це набір операцій, які в сукупності складають єдину логічно неподільну одиницю роботи. Часто властивості транзакцій описують відомою абревіатурою ACID (atomicity, consistency, isolation, durability), введеною Андреа Рейтером і Тео Хардером 1983 року [8]. В контексті розподілених баз даних особливу увагу потрібно звернути на ізольованість та надійність.

- Атомарність (atomicity) — властивість, яка полягає в тому, що жодна транзакція не може бути виконаною частково. Або всі операції у межах однієї транзакції завершуються успішно, або жодна з них не має ефекту на систему.
- Узгодженість (consistency) в контексті ACID означає дотримання коректного стану бази даних до початку транзакції і після її завершення, що зазвичай досягається за допомогою встановлення обмежень цілісності (наприклад, первинних і зовнішніх ключів, а також додаткових перевірок, визначених користувачем). При цьому поняття коректного стану є специфічним для кожного застосунку.
- Властивість ізольованості (isolation) полягає в тому, що транзакції, які виконуються паралельно, не можуть впливати на результати один одної, тобто одночасно виконувати операції над одними й тими самими елементами даних.

- Надійність (durability) гарантує, що результати виконання транзакції будуть довговічними, тобто зберігатимуться й будуть доступними навіть попри збої чи відмови в роботі системи. У розподілених базах даних це часто означає реплікацію на решту вузлів, а на кожному окремому вузлі — здійснення запису на енергонезалежний носій.

Під час опрацювання паралельних транзакцій важливим є моделювання та представлення можливих розкладів виконання. Розклад називають серіалізованим (serializable), якщо він еквівалентний деякому послідовному порядку виконання цих транзакцій.

У розподілених системах доцільно розглядати два типи транзакцій — локальні та глобальні. Локальні транзакції звертаються до даних в межах однієї локальної бази даних, тоді як глобальні — у декількох різних вузлах.

Для того, щоб операції в межах однієї розподіленої транзакції виглядали атомарно, потрібно застосування протоколів атомарної фіксації (atomic commitment protocols), які гарантують, що транзакція не буде зафіксованою, якщо принаймні один учасник не зможе цього забезпечити. Протоколи двофазної та трифазної фіксації розглянуті в наступному розділі.

Багато розподілених сховищ відмовилися від розподілених транзакцій через складність реалізації, а також тому, що вони можуть створювати проблеми у ситуаціях, які вимагають високу доступність і продуктивність [2, с. 231]. Особливо це стосувалося NoSQL баз даних, які отримали значне поширення з кінця 2000-х років.

Розділ 2. Стратегії вирішення проблеми узгодженості даних

2.1. Протоколи фіксації транзакцій

Щоб забезпечити дотримання властивості атомарності, всі вузли, на яких виконуються операції розподіленої транзакції *T*, мають узгодити між собою кінцевий результат: транзакція *T* або успішно фіксується на кожному з вузлів, або перериває́ться. Для досягнення цього застосовуються протоколи двофазної (two-phase commit) і трифазної фіксації (three-phase commit).

Протокол двофазної фіксації

Протокол двофазної фіксації передбачає наявність координатора, який може бути реалізований в одному ж процесі з клієнтом, що робить запит на обробку транзакції, або ж як окремий сервіс [2, с. 366].

Виконання відбувається у два етапи: підготовка (prepare) та фіксація (commit). На першому етапі координатор розсилає на всі вузли, які беруть участь у транзакції, повідомлення, запитуючи, чи готовий кожен з них виконати свою частину транзакції. Під час другого етапу координатор ініціює фіксацію результатів.

Щоб гарантувати надійність (durability) виконання цієї транзакції, і координатор, і вузли повинні використовувати WAL-логування [9, с. 71].

Під час підготовчого етапу координатор логує команду "*prepare T*" і надсилає її до всіх учасників, які беруть участь у виконанні цієї транзакції. Отримавши повідомлення від координатора, вузол дбає про все необхідне для успішного виконання своєї частини роботи (наприклад, отримує блокування елементів даних).

Якщо вузол не може забезпечити коректне виконання транзакції зі свого боку, він логує команду "*abort T*" і надсилає її у відповідь координатору. В іншому разі вузол логує "*ready T*" й інформує координатора про готовність зафіксувати транзакцію.

Після отримання позитивних відповідей від всіх учасників без винятку, або хоча б одного повідомлення про скасування, координатор остаточно визначає долю

транзакції. Якщо всі вузли надіслали “*ready T*”, транзакція повинна бути зафіксованою, інакше – скасованою. Залежно від прийнятого рішення координатор записує в лог команду “*commit T*” або “*abort T*” і розсилає її на всіх учасників. Отримавши повідомлення від координатора, вузол відповідно фіксує або скасовує транзакцію.

Рисунок нижче ілюструє сценарій успішної фіксації транзакції відповідно до двофазного протоколу.

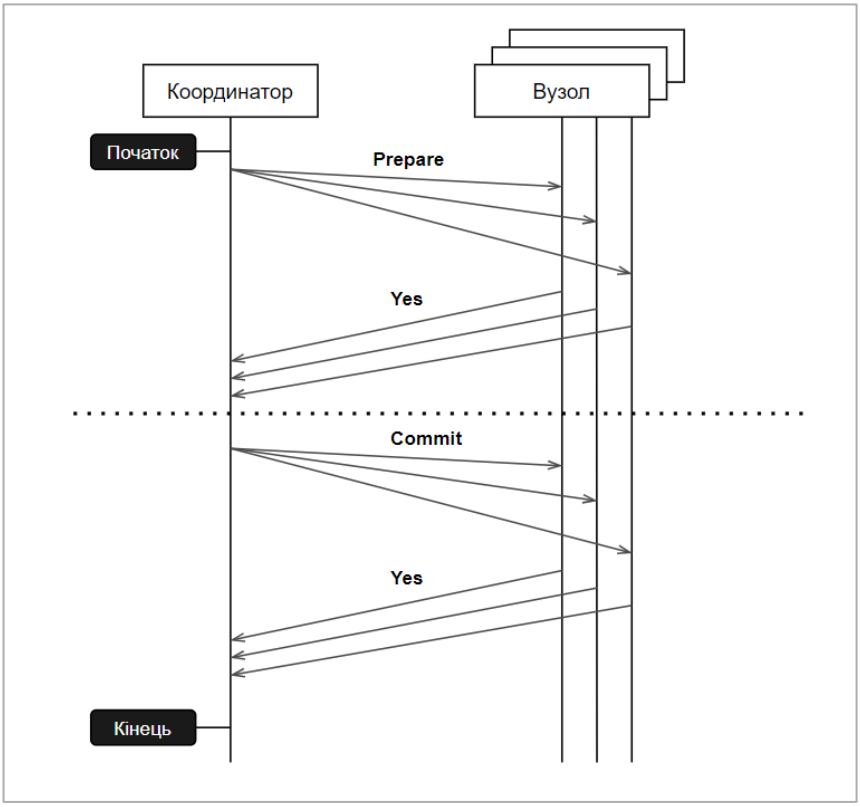

Рисунок 1. Схема успішного виконання двофазної фіксації.

Варто врахувати, що кожен окремий учасник може не відповідати внаслідок збою або відмови. Якщо це трапляється на першому етапі, тоді координатор очікує певний проміжок часу, після якого діє так, ніби отримав у відповідь “*abort T*”. Якщо ж вузол не відповідає після того як надіслав підтвердження “*ready T*”, координатор завершує виконання протоколу, не звертаючи уваги на відмову окремого учасника.

Вузол, який не надав відповіді унаслідок відмови, у майбутньому зможе відновитися шляхом сканування власного логу. У цьому випадку можливі три варіанти: останнім записом буде 1) “*abort T*”, тоді вузол здійснить відкат; 2) “*commit*

T”, тоді вузол зафіксує транзакцію; 3) “*ready T*” означитиме, що вузол не встиг отримати відповіді від координатора і тому має надіслати до нього запит, щоб дізнатися про остаточне рішення.

Значно серйознішою проблемою у двофазному протоколі фіксації є відмова координатора. Якщо окремий вузол не отримав фінальне повідомлення від координатора, він може спробувати зробити запити на інших учасників і скопіювати собі результат. Це не загрожує небезпекою, адже сама ідея протоколу полягає в тому, що рішення приймаються одноголосно і значення у всіх мають бути однакові. Однак у випадку, коли координатор відмовив ще до того як розіслати результати, відбувається блокування роботи системи, допоки координатор не буде відновлено. Тож варто подбати про механізми забезпечення відмовостійкості координатора за допомогою реалізації алгоритму розподіленого консенсусу.

Протокол трифазної фіксації

Протокол трифазної фіксації є розширенням, покликаним забезпечити стійкість системи до відмови координатора шляхом запровадження третього кроку у процес прийняття рішення. Однак трифазна транзакція передбачає виконання певних умов: обмеженість затримок в мережі та обмеженість часу відповіді вузлів. В іншому разі протокол не здатен гарантувати атомарність виконання операцій.

У трифазній транзакції наявний додатковий етап між підготовкою і фіксацією, під час якого координатор розсилає інформацію про стан вузлів, що дає змогу уникнути блокування роботи навіть за відмови координатора.

Під час першого етапу, як і у двофазній транзакції, координатор розподіляє повідомлення про транзакцію між вузлами і збирає голоси. Переривання транзакції на цьому кроці можливе за однієї з умов: відмови координатора, завершення часу очікування на відповідь, негативної відповідь хоча б від одного вузла.

Отримавши позитивну відповіді від всіх учасників, координатор розсилає команду “*prepare T*”. Транзакція переривається під час другої фази алгоритму, якщо відбувається відмова координатора або спливає час очікування на вузлі.

Під час третьої фази тоді, як всі учасники підтвердили свою готовність, транзакція фіксується навіть попри відмову координатора, адже на цьому етапі прийняте рішення відоме всім.

Найгіршим варіантом для трифазної транзакції є розділення мережі (network partition), за якого відбувається ізоляція окремих учасників. Ті, що мають зв'язок з координатором можуть пройти фазу підготовки й успішно продовжити фіксацію, решта не зможуть отримати повідомлення від координатора і перервуть транзакцію. Унаслідок стан системи стає неузгодженим.

Трифазний протокол не використовується часто на практиці, що можна пов'язати з необхідністю великих витрат на повідомлення і потенційну неузгодженість.

2.2. Алгоритми розподіленого консенсусу

Консенсус є фундаментальною проблемою в розподілених системах і полягає в досягненні згоди між різними вузлами щодо деякого запропонованого значення. Послідовність узгоджених консенсусом значень може формувати розподілений лоґ, який застосовується для побудови реплікованого автомату (replicated state machine) [10]. Система складається з множини таких автоматів, кожен з яких містить свою репліку лоґу і послідовно виконує записані команди. Якщо послідовність команд однакова на всіх вузлах, відповідно стани, у якому перебувають автомати після їх покрокового виконання, однакові, тобто узгоджені.

Алгоритми консенсусу мають задовольняти такі вимоги [11]:

- дійсність (validity) – узгодження відбувається лише для значень, запропонованих одним із процесів;
- узгодженість (agreement) – коректні процеси не можуть приймати різні значення;
- цілісність (integrity) – жоден процес не може приймати рішення більше одного разу;

- завершуваність (termination) – всі коректні процеси врешті приймають деяке значення.

Найбільш поширеними алгоритмами консенсусу на сьогодні є Paxos і Raft. Класичний Paxos, опис якого був опублікований Леслі Лампортом 1998 року в статті “The Part-Time Parliament” [12], передбачав досягнення консенсусу лише для одного значення. Коли мова йде про підтримку розподіленого логу, застосовують модифікацію початкового алгоритму (яка зі свого боку теж має різні варіанти реалізації), відому як Multi-Paxos, яка й розглядатиметься далі.

Попри те, що Paxos отримав широке застосування у різноманітних системах (зокрема, Google Spanner [13], Apache Cassandra [14], Amazon DynamoDB [15] та ін.), він завжди видавався надто важким для розуміння і, відповідно, для коректної реалізації. Цей факт підтверджується низкою спроб протягом років повторно його пояснити [16], [17], [18].

Складність в розумінні деталей, пов’язаних з імплементацією Paxos, стала основною мотивацією для розробки нового алгоритму розподіленого консенсусу Raft, вперше опублікованого 2013 року в статті Дієго Онгаро та Джона Остерхута “In Search of an Understandable Consensus Algorithm” [19]. Як і попередник, Raft швидко отримав визнання та застосування у низці систем, зокрема таких як CockroachDB [20], Hazelcast [21], RabbitMQ [22], Apache Kafka [23] та ін.

І Paxos, і Raft не передбачають можливість візантійських [24] помилок в системі, що робить можливим досягнення консенсусу з меншою кількістю одночасно активних вузлів. При цьому система розглядається як асинхронна: швидкість роботи вузлів може відрізнятися, а повідомлення – затримуватися на невизначений час.

Обидва алгоритми передбачають обрання деякого вузла лідером, на якого надходитимуть запити на узгодження певного значення (команди) від клієнта. Лідер повинен спершу записати це значення до свого логу, а потім розіслати його на решту учасників, щоб ті зробили те саме. Як тільки лідер отримує підтвердження від

більшості вузлів, що вони успішно реплікували нове значення, він застосовує його до свого автомату. Цей процес повторюється, доки лідер не зазнає відмови і на його місце стане інший вузол.

Далі окремо розглянуто деякі важливі відмінні деталі Multi-Paxos і Raft.

Multi-Paxos

Класичний варіант Paxos передбачає три ролі для вузлів:

- заявники (proposers) отримують запити від клієнтів і пропонують значення для голосування;
- виборці (acceptors) можуть отримувати пропозиції від різних заявників з різними значеннями і мають голосувати за прийняття або відхилення запропонованих значень; при цьому відмова одного виборця не становить загрози для роботи системи, допоки є більшість;
- учні (learners) зберігають результат прийнятих рішень.

Кожен вузол може виконувати будь-яку з перелічених ролей, тобто всі три працюють на одній машині як один процес.

Одним із недоліків класичного алгоритму Paxos є необхідність щоразу проводити перший етап (пропозиції). Заявник може розпочати другий етап (реплікацію) лише після отримання (або підтвердження) лідерства. Щоб дозволити заявнику перевикористати свою позицію й оминути етап пропозиції, використовуються алгоритм Multi-Paxos, який визначає деякого стабільного заявника-лідера.

У Multi-Paxos є сенс розглядати два стани: лідер (leader) і послідовник (follower). Алгоритм передбачає виконання у дві фази: підготовка (prepare) і прийняття (accept).

Наведений опис ґрунтується на статті [16]. У реальних системах реалізації часто значно відрізняються.

Для роботи кожен лідер має генерувати унікальний номер пропозиції *n*, який монотонно зростає з кожною пропозицією.

Коли деякий учасник стає лідером, він переходить у фазу підготовки, під час якої розсилає на всіх учасників повідомлення *prepare* зі своїм згенерованим унікальним номером *n*. Коли вузол-послідовник отримує повідомлення, можливі два варіанти.

- Якщо *n* менше за номер значення, який вузол пообіцяв прийняти до цього, тоді послідовник ігнорує це повідомлення.
- Якщо *n* більше за номер значення, який вузол пообіцяв прийняти до цього, тоді послідовник у відповідь обіцяє прийняти отримане значення й порівнює останній індекс свого логу з тим, який у лідера. Якщо в послідовника значення вище, це означає, що він містить значення, яких немає у лідера, тож у відповідь надсилає повідомлення *promise* зі списком відсутніх значень.

Коли лідер отримує *promise* від більшості послідовників, він оновлює свій лог, додаючи в кінець значення від послідовника з найвищим індексом. Після цього лідер додає до послідовності пропозиції, отримані від клієнтів.

Лідер надсилає повідомлення *accept* на всіх послідовників, які пообіцяли прийняти значення. Повідомлення містить номер лідера *n*, і список значень логу від індексу, надісланим у *promise*. Якщо *n* збігається з тим, який послідовник пообіцяв прийняти, значення додається до локального логу після останньої прийнятої команди у списку. Послідовник надсилає у відповідь повідомлення *accepted*. Цей процес синхронізує логи на всіх вузлах. Фаза підготовки завершується, коли більшість послідовників прийняли повідомлення accept, після чого починається фаза прийняття.

З цього часу лідер може розсилати пропозиції від клієнтів безпосередньо повідомленням accept. Послідовники відповідають accepted, якщо надісланий номер збігається з тим, який вони пообіцяли прийняти. Лідер відслідковує останній прийнятий індекс на кожному з учасників, оновлюючи його щоразу після отримання accepted від послідовника.

Якщо лідер отримує позитивні відповіді від більшості, він вважає, що значення було прийнято (більшість вузлів успішно додали його до свого логу). Після цього лідер надсилає повідомлення decide. Коли вузол зафіксував значення, команда може бути виконана.

Raft

Як і Multi-Paxos, Raft також забезпечує консенсус шляхом обрання лідера, який прийматиме пропозиції від клієнтів і реплікуватиме їх на решту вузлів. Головною відмінністю двох алгоритмів є те, як саме відбувається обрання лідера.

Raft передбачає, що кожен вузол може перебувати в одному з трьох станів: послідовник (follower), кандидат (candidate), лідер (leader). За коректної роботи в один момент часу в системі є лише один лідер, а решта вузлів є послідовниками. Raft поділяє час на терміни, які мають монотонну нумерацію. Один термін – це період роботи від обрання нового лідера до моменту, коли він припинить відповідати. Для інформування послідовник про наявність активного лідера використовується періодичний сигнал heartbeat, який надсилається з кожним повідомлення від лідера.

Всі вузли починають як послідовники. Не отримуючи певний час повідомлення від лідера, учасник збільшує свій термін, стає кандидатом і розсилає повідомлення RequestVote на інші вузли. Ті віддають свій голос за трьох умов:

- якщо термін кандидата не нижчий за їх власний,
- якщо вони ще не проголосували за іншого,
- якщо лог кандидата містить як мінімум всі значення, які записані локально.

Коли кандидат отримує голоси від більшості вузлів, він стає новим лідером і реплікує свій лог, розсилаючи повідомлення AppendEntries, яке або містить значення, яке потрібно реплікувати, або є порожнім і в цьому випадку працює лише як сигнал heartbeat.

У Raft вся комунікація відбувається лише з ініціативи лідера, тобто записи з логу реплікуються лише в один бік – від лідера до послідовника.

Коли лідер переконується, що більшість вузлів успішно реплікували значення, він вважає його зафіксованим (committed) і може застосувати операцію до автомату.

Зіставлення Multi-Paxos і Raft

Основною відмінністю між двома алгоритмами є процес обрання нового лідера.

Під час голосування у Paxos послідовники надсилають у відповідь свої записи. Коли лідер отримав відповіді від більшості учасників, він додає отримані записи з найбільшим номером до свого логу. Етап голосування у Raft відбувається простіше: лідером може стати лише той кандидат, лог якого є актуальним, тому на відміну від Paxos немає потреби пересилати значення, які не збігаються.

Іншою важливою деталлю є те, що Raft надає гарантію: якщо два логи містять одну й ту саму операцію, вона в обидвох матиме однаковий порядковий номер і термін, за якими її можна ідентифікувати. Paxos цього не гарантує, адже в ньому операція може бути перезаписана іншим лідером з вищим терміном. Записи додаються до логу лідера з його поточним номером і саме з ним він реплікує їх на інші вузли. У Raft реплікація записів відбувається без зміни значення номера.

Значення, які реплікує лідер у Paxos, або належать до поточного терміну, або вже є зафіксованими. У Raft лідер може розсилати ще не зафіксовані значення з попередніх термінів.

Під час виборів лідера в Raft є можливість виникнення ситуації, коли жоден кандидат не отримає більшості голосів і лідера не буде обрано. Це може трапитися, коли декілька послідовників одночасно перейдуть у стан кандидата і ніхто з них не отримає більшості голосів. В такому випадку починається ще один раунд виборів, унаслідок якого лідера врешті буде обрано.

Розділ 3. Реалізація розподіленого сховища даних

3.1. Специфікація розподіленого сховища

Більшість NoSQL систем початково жертвували гарантіями сильної узгодженості на користь швидкодії та доступності. Однак останнім часом можна спостерігати деяку тенденцію до впровадження механізмів розподіленого консенсусу для забезпечення узгодженості між репліками та відмовостійкості. Врешті гарантія сильної узгодженості в розподілених NoSQL базах даних є необхідною у випадках, коли коректність і достовірність даних є критично важливою.

Засновуючись на наведених вище міркуваннях, було вирішено реалізувати розподілене сховище “ключ-значення” (DKVS), яке надає гарантій сильної узгодженості (лінеаризовності) і підтримує виконання розподілених транзакцій.

Основними функціональними вимогами DKVS є

- збереження даних у вигляді “ключ-значення”;
- виконання операцій додавання, оновлення, читання та видалення даних;
- підтримка виконання транзакцій за допомогою блокування даних;
- забезпечення відмовостійкості за допомогою реплікації;
- забезпечення масштабованості за допомогою секціонування.

DKVS гарантує, що операції читання завжди повертають найактуальніше значення.

3.2. Інструменти розробки

Для реалізації сховища було обрано мову Go, основними характеристиками якої є швидкість виконання, простий синтаксис, автоматичне збирання сміття та статична типізація. У контексті реалізація розподілених алгоритмів перевагами Go також є вбудована підтримка паралельного програмування на основі механізму горутин і каналів, а також стандартні бібліотеки net/rpc для віддаленого виклику процедур і net/http.

Клієнтське застосування з інтерфейсом командного рядка створене за допомогою бібліотеки Cobra.

3.3. Реалізація розподіленого сховища

Розподілене сховище DKVS складається з основних трьох компонентів:

- Store Group відповідає за збереження деякої підмножини пар “ключ-значень”;
- Coordinator Group координує виконання транзакцій;
- Config Group керує конфігурацією сховища.

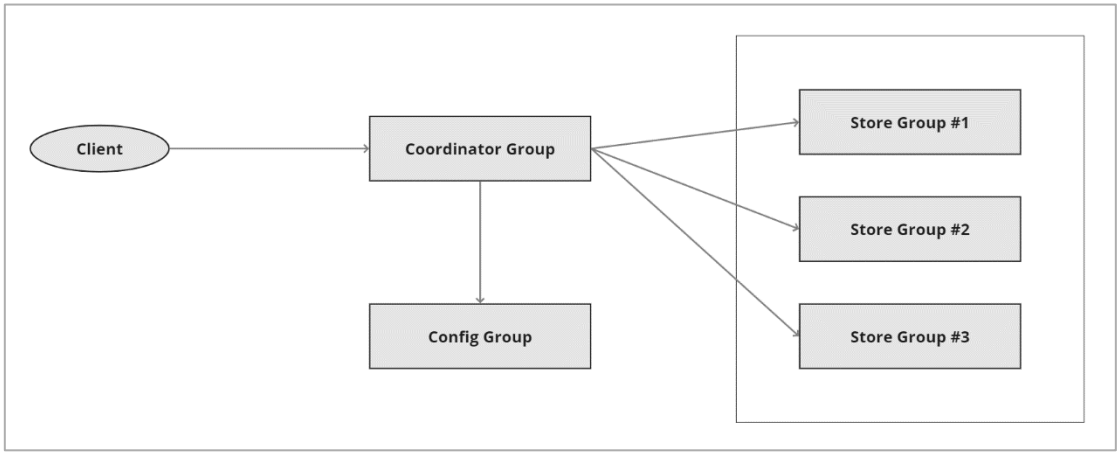

Рисунок 2. Загальна архітектура DKVS

Розглянемо кожен з компонентів детальніше.

Store Groups

Всі дані у сховищі розподіляються між S ($S \geq 9$) секціями відповідно до результату застосування хеш-функції до ключа під час вставки нових елементів даних. Кожну секцію обслуговує деяка група вузлів Store Group. При цьому розраховується, що загальна кількість секцій S в рази більша за сукупну кількість груп G. Тобто одна група обслуговує одразу декілька секцій з даними. У такий спосіб під час масштабування сховища (зміни G) відповідність ключів до сегментів залишається

сталою (шаблон, відомий як Fixed Partitions [9, с. 219]). Чим більше значення S, тим краще вдається досягнути рівномірного розподілу ключів між групами.

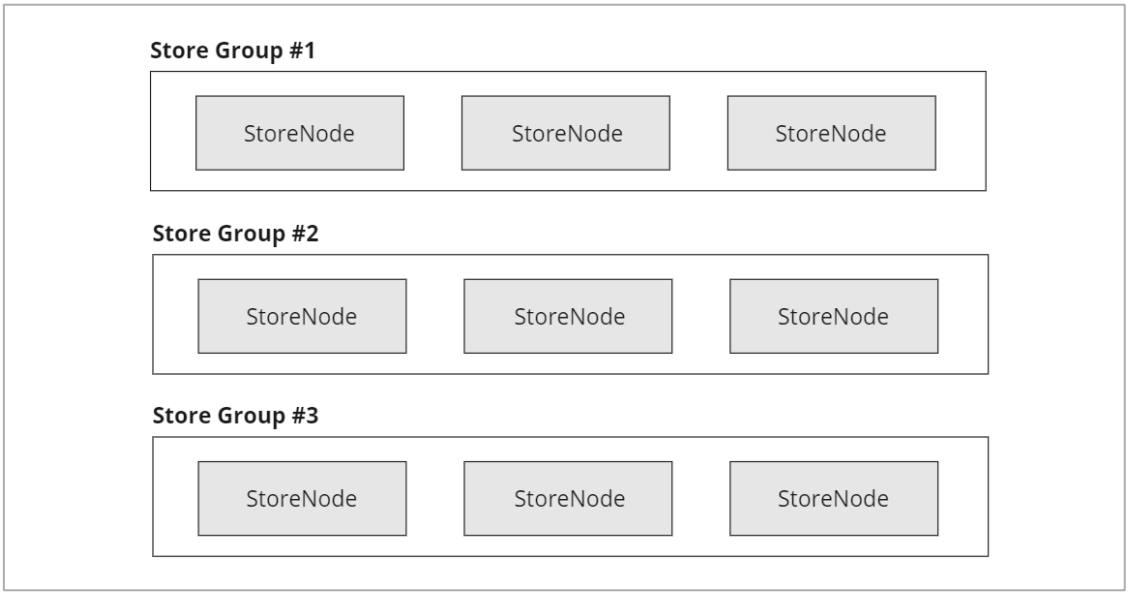

Рисунок 3. Архітектура Store Group

Кожна група складається з повних копій вузлів StoreNode, які обслуговують одну й ту саму множину секцій. Реплікація даних між вузлами однієї групи здійснюється за допомогою Raft. Щоб забезпечити стабільну роботу у випадку часткових відмов, група має містити 2n + 1 вузлів, тобто щонайменше три сервери.

```
type StoreNode struct {
    mu            sync.Mutex
    id            int
    raftPeer      *raft.Raft
    applyChanel   chan raft.ApplyMessage
    groupId       int
    ongoingTxns   map[int]*Transaction
    data          []*repository.Repository
    config        configservice.Config
    configClient  *configservice.Client
}
```

Кожен вузол StoreNode sn містить три важливих компоненти:

- sn.raftPeer – один екземпляр учасника Raft, який цілковито відповідає за комунікацію з іншими вузлами StoreNode в межах групи;

- sn.data – репозиторій-обгортка над хеш-таблицею з даними, яка працює як менеджер локальних транзакцій, тобто надає і знімає блокування (на основі двофазного протоколу блокування) окремих ключів на сервері, а також періодично здійснює запис даних на диск;
- sn.config – об'єкт, який зберігає поточну конфігурацію сховища, а саме які сегменти належать до яких груп; періодично sn надсилає запит до лідера Config Group для оновлення інформацію про поточну конфігурацію.

У сховищі є три типи операцій над даними:

- Get(key) – повертає значення, що відповідає ключу;
- Put(key, value) – додає новий ключ до сховища й асоціює значенням з ним (якщо такий ключ вже наявний, оновлює його значення);
- Delete(key) – видаляє ключ і асоційоване з ним значення.

```
type Transaction struct {
    Id          int64
    ClientId    int64
    RequestId   int64
    Status      TxnStatus
    Operations  []Operation
}
```

Будь-яка операція на вузлі відбувається в межах локальної транзакції, яку Coordinator Group надсилає на StoreNode як частину розподіленої транзакції. Транзакція може складатися щонайменше з однієї операції. Coordinator Group звертається до StoreNode, який в цей момент є лідером, надсилаючи rpc, для виклику однієї з основних функцій:

- Lock() – для блокування ключів, що беруть участь в поточній транзакції;
- Prepare() – для підтвердження того, що вузол отримав всі потрібні блокування і готовий виконати свою частину транзакції;
- Commit() – для виконання операцій транзакції і звільнення заблокованих ключів.

Coordinator Group

CoordinatorNode – це сервер, на який надходять запити від клієнтів у вигляді одної чи низки операцій з даними. Як вже було зазначено, будь-які маніпуляції з даними представляються у вигляді транзакцій. Для виконання запиту клієнта координатор імплементує двофазний протокол.

Coordinator Group, як і Store Group, реалізований у вигляді кластеру з 2n + 1 вузлів CoordinatorNode, реплікація між якими відбувається за допомогою Raft. Це дає змогу уникнути блокування роботи системи (про яке йде мова в описі двофазного протоколу).

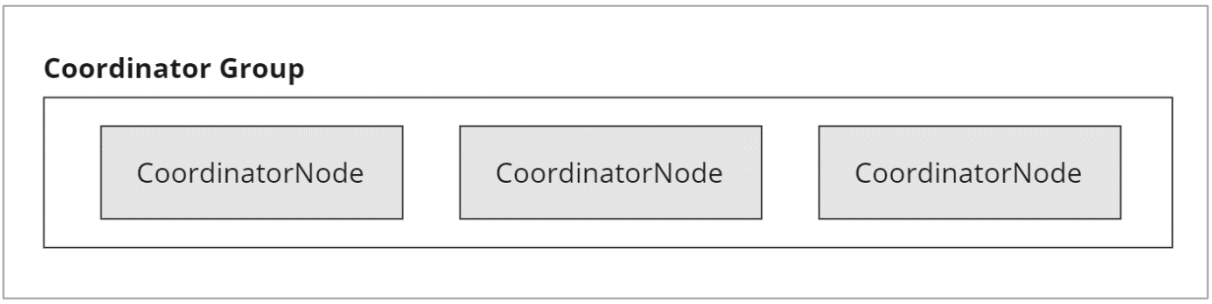

Рисунок 4. Архітектура Coordinator Group

Основними компонентами CoordinatorNode *cn* є

- cn.raftPeer – один екземпляр учасника Raft, який цілковито відповідає за комунікацію з іншими серверами в межах групи;
- cn.config – об’єкт, який зберігає поточну конфігурацію сховища;

Клієнти звертаються до координатора, надсилаючи rpc, для виклику функції Process(), передаючи список з n ($n \geq 1$) операціями на виконання.

Отримавши запит від клієнта, CoordinatorNode визначає, які Store Group беруть участь у розподіленій транзакції і розпочинає виконання двофазного протоколу. Під час першого етапу координатор розсилає запити Lock() для блокування ключів на всі групи, що беруть участь у транзакції. Якщо деяка група не може заблокувати принаймні один з ключів, вона інформує координатор про помилку, після чого координатор періодично повторює спроби отримати блокування. Після отримання

блокування всіх ключів, координатор надсилає повідомлення Prepare() на кожну задіяну групу для підтвердження її готовності виконати операції. Коли координатор отримує позитивні відповіді від всіх вузлів, він надсилає Commit() для фіксації результатів і завершення транзакції.

Config Group

Щоб забезпечити можливість масштабування сховища, реалізовано окремий сервіс, який відповідає за зміну налаштувань у системі. У процесі роботи адміністратор сховища може додавати або видаляти окремі Store Group у міру збільшення чи зменшення обсягів даних, які потрібно зберігати.

Config Group реалізована як відмовостійкий кластер вузлів ConfigNode на основі Raft. Клієнт звертається до ConfigNode, який є лідером, надсилаючи rpc, для виклику однієї з функцій:

- Join() – для приєднання нової групи вузлів;
- Leave() – для видалення наявної групи вузлів;
- Info() – для отримання інформації про поточну конфігурацію системи.

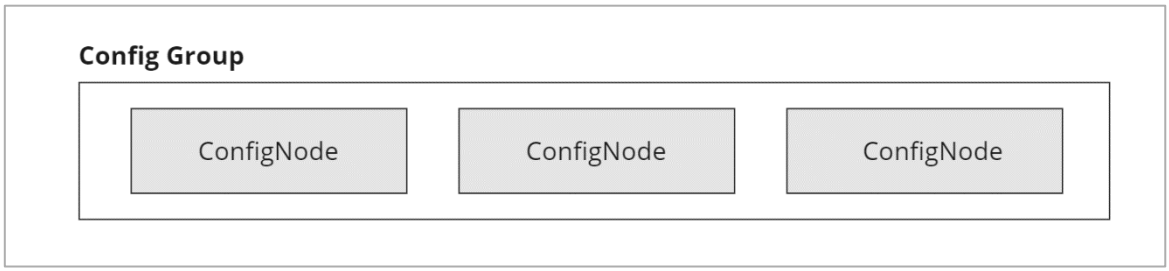

Рисунок 5. Архітектура Config Group

Після отримання запиту на зміну конфігурації, ConfigNode лише створює новий об’єкт Config і не відповідає за інформування про це Store Group. Натомість лідер кожної групи StoreNode періодично надсилає запити Info(), щоб дізнатися, чи не змінилася конфігурація. Якщо виявлено зміни, Store Group розпочинає процес міграції окремих сегментів.

Якщо група позбувається деякого сегменту, вона більше не приймає запити на виконання операцій з цими ключами і негайно пересилає дані на іншу групу.

```
type Config struct {
    Id             int
    ShardsToGroups []int
    GroupsToNodes  map[int][]string
}
```

Аналогічно, якщо група отримує новий сегмент, вона починає приймати запити на виконання операцій з новими даними лише після їх остаточної міграції.

Зміна конфігурації і міграція сегментів є витратною операцію, тому очікується, що потреба в її виконанні виникатиме нечасто.

3.4. Реалізація клієнта

Клієнт надає інтерфейс командного рядка для користувачів і здійснює комунікацію з координатором (Coordinator Group) на основі протоколу HTTP.

Користувач може використовувати такі команди:

- get <key> для отримання значення, асоційованого з ключем;
- put <key, value> додає ключ і значення або оновлює значення, асоційоване з ключем;
- delete <key, value> видаляє ключ та значення зі сховища;
- process <command list> надсилає список команд для атомарного виконання в межах однієї транзакції.

3.5. Можливі покращення

Розроблена система підтримує зміни конфігурації на рівні груп реплік, при цьому відсутня можливість динамічної зміни кількості вузлів у межах одного кластеру, що було б однозначно корисним для практичного застосування й гнучкості в масштабуванні.

Іншим можливим напрямом вдосконалення реалізованого розподіленого сховища є оптимізація схеми блокувань елементів даних шляхом запровадження механізмів

оптимістичного підходу, що потенційно може зменшити час виконання окремих операцій.

Також наявні варіанти для розширення функціональності клієнтського застосування й інтерфейсу користувача, зокрема додавання можливості виконання адміністративних операцій, таких як динамічне додавання та видалення груп реплік.

Висновок

Підсумком роботи є дослідження та порівняння алгоритмів розподіленого консенсусу Paxos і Raft, а також двофазного та трифазного протоколів атомарної фіксації. Унаслідок, для практичної реалізації було обрано Raft як більш новий підхід, що пропонує чітко визначену процедуру реплікації, і двофазний протокол як перевірене часом рішення для розподілених транзакцій.

На основі реалізованих алгоритмів було розроблено розподілене сховище “ключ-значення”, в основі роботи якого покладено три відмовостійких сервіси (Store Group, Coordinator Group, Config Group), представлених у вигляді груп вузлів, реплікація між якими відбувається за допомогою Raft. Горизонтальне масштабування сховище здійснюється одночасно за використання реплікації та секціонування. Розроблене рішення гарантує, що операції читання завжди повертають найактуальніше значення, тобто забезпечують лінеаризовність.

Можливим покращенням системи у майбутньому є додавання підтримки зміни кількості реплік на рівні одного кластера, що може бути корисним на практиці. А також оптимізація схеми блокування, розглянувши можливості застосування оптимістичного підходу.

Узагальнюючи результати роботи, можна стверджувати, що проблема узгодженості у розподілених базах даних є багатогранною і вимагає комплексного підходу до її вирішення. Вибір конкретного методу забезпечення узгодженості повинен базуватися на детальному аналізі вимог до системи, характеру даних, навантаження та очікуваного рівня доступності.

Проблема узгодженості даних є критично важливою для розвитку сучасних інформаційних систем і потребує постійного вдосконалення та адаптації до нових вимог.

Список джерел та літератури

1. Brewer E. Towards Robust Distributed Systems. Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. 2000.
2. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Inc., 2017. 611 p.
3. Petrov A. Database Internals: A Deep Dive into How Distributed Data Systems Work. O'Reilly Media, Inc., 2019. 370 p.
4. Herlihy M., M. Wing J. Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems,. 1990. Vol. 12, no. 3. P. 463–492.
5. Lamport L. How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Transactions on Computers. 1997. Vol. 46, no. 7. P. 779–782.
6. Van Steen M., Tanenbaum A. Distributed Systems. Maarten van Steen, 2023. 683 p.
7. Hutt P. W., Ahamad M., Slow memory: weakening consistency to enhance concurrency in distributed shared memories. Proceedings., 10th International Conference on Distributed Computing Systems, Paris, France, 1990, P. 302-309.
8. Haerder T., Reuter A. Principles of transaction-oriented database recovery. ACM Computing Surveys. 1983. Vol. 15, no. 4. P. 287–317.
9. Joshi U. Patterns of Distributed Systems. Addison-Wesley Professional, 2023. 464 p.
10. Schneider F. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys. 1990. Vol. 22, no. 4. P. 299–319.
11. Cachin C., Guerraoui R., Rodrigues L. Introduction to Reliable Distributed Programming. 2nd ed. Springer, 2011. 386 p.
12. Lamport L. The part-time parliament. ACM Transactions on Computer Systems. 1998. Vol. 16, no. 2. P. 133–169.
13. Corbett J., Dean J., Epstein M. Spanner: Google's Globally Distributed Database. ACM Transactions on Computer Systems. 2013. Vol. 31, no. 3. P. 1–22.
14. Apache Cassandra 4.1: Rock Solid, Cloud-Native, Strongly Consistent and Highly Scalable. [Електронний ресурс] – Режим доступу до ресурсу: https://cassandra.apache.org/_/blog/Cassandra-4.1-is-here.html.

15. Elhemali M., Gallagher N., Gordon N. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. Amazon Web Services. 2022.

16. Lamport L. Paxos Made Simple. ACM SIGACT News (Distributed Computing Column). 2001. Vol. 32, no. 4. P. 51–58.

17. Meling, H., and Jehl, L. Tutorial summary: Paxos explained from scratch. In Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (Berlin, Heidelberg, 2013), OPODIS 2013, Springer-Verlag, pp. 1–10.

18. Van Renesse R., Altinbuken D. Paxos Made Moderately Complex. ACM Computing Surveys. 2015. Vol. 47, no. 3. P. 1–36.

19. Ongaro D., Ousterhout J. In search of an understandable consensus algorithm. USENIX ATC'14: Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference. 2014. P. 305–320.

20. Replication Layer. CockroachDB Docs. Version v23.2.5 [Електронний ресурс] – Режим доступу до ресурсу: https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html

21. CP Subsystem. Hazelcast. Version 4.2 [Електронний ресурс] – Режим доступу до ресурсу: https://docs.hazelcast.com/imdg/4.2/cp-subsystem/cp-subsystem

22. Quorum Queues. RabbitMQ. Version: 3.13 [Електронний ресурс] – Режим доступу до ресурсу: https://www.rabbitmq.com/docs/quorum-queues

23. KRaft Overview. Apache Kafka. Version 7.6 [Електронний ресурс] – Режим доступу до ресурсу: https://docs.confluent.io/platform/current/kafka-metadata/kraft.html

24. Lamport L., Shostak R., Pease M. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems. 1982. Vol. 4, no. 3. P. 382–401.

25. Howard H., Mortier R. Paxos vs Raft: Have we reached consensus on distributed consensus?. PaPoC '20: Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data. 2020. P. 1–9.