СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет ___інформаційних технологій та електроніки_____

Кафедра _____інформаційних технологій та програмування_____

Пояснювальна записка

до магістерської дипломної роботи

_____магістр_____
(освітньо-кваліфікаційний рівень)

на тему: Розробка алгоритму синхронізації стану баз даних для No-SQL систем.

Виконав: студент 2 курсу, групи_ІСТ-24зм_
126 «Інформаційні системи та технології»___
(шифр і назва спеціальності)
_____Дьомін М.К._____
(прізвище та ініціали)
Керівник __проф. д.т.н. Захожай О. І._____
(прізвище та ініціали)
Рецензент проф., д. т. н. Меняйленко_О.С.__
(прізвище та ініціали)

Київ – 2025 року

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет ___інформаційних технологій та електроніки_____
Кафедра _____інформаційних технологій та програмування_____
Освітньо-кваліфікаційний рівень __магістр_____
Спеціальність 126 «Інформаційні системи та технології»_____
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТП
_____ д.т.н., проф. Захожай О.І.
(підпис)
« ___ » _____2025 р.

ЗАВДАННЯ

на магістерську дипломну роботу студенту
_____Дьоміну Максиму Костянтиновичу_____
(прізвище, ім'я, по батькові)

1.Тема роботи: Розробка алгоритму синхронізації стану баз даних для No-SQL систем.

керівник роботи __проф. Захожай О.І._____
(вчене звання, науковий ступінь, прізвище, ім'я, по батькові)
затверджені наказом університету від «_28_»_11_ 2025року №_241/17.03_
2. Строк подання студентом роботи: 18 грудня 2025 р._____
3. Вихідні дані до роботи: _Матеріали науково-дослідної практики, науково-методична література, дані інтернет-мережі _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1 Вступ
4.2 Аналітичний огляд питання (огляд публічних джерел інформації)
4.3 Основна частина, в якої висвітлити методи, які будуть використовуватися для реалізації проекту.
4.4 Практична частина – огляд технологій, які використовуються під час реалізації проекту.
4.4 Висновки
4.5 Перелік використаних джерел

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

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Огляд існуючих підходів до синхронізації між центральною та мобільними базами даних	д.т.н., проф. Захожай О.І.		
Проектування алгоритму синхронизації станів баз даних	д.т.н., проф. Захожай О.І.		
Особливості реалізації алгоритму синхронізації	д.т.н., проф. Захожай О.І.		

7. Дата видачі завдання __07.11. 2025р._____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Одержання завдання на виконання роботи	07.11.2025	виконано
2.	Укладання і погодження з керівником плану і етапів виконання роботи	10.11.2025	виконано
3.	Узагальнення даних літературних джерел	15.11.2025	виконано
4.	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху виконання завдання	18.11.2025	виконано
5.	Аналіз технічних засобів та існуючих алгоритмів	22.11.2025	виконано
6.	Реалізація практичної частини завдання	07.12.2025	виконано
7.	Укладання, оформлення та погодження пояснювальної записки з керівником	17.12.2025	виконано
8.	Надання пояснювальної записки на кафедру	18.12.2025	виконано
9.	Підготовка доповіді та презентації	21.12.2025	виконано

Студент _____ Дьомін М.К.
(підпис) (прізвище та ініціали)

Керівник роботи _____ _Захожай О.І._
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Загальний обсяг основного тексту складає 49 сторінок, включає 3 рисунки, список джерел на 40 найменувань та 17 сторінок додатків.

Дипломна робота присвячена дослідженню проблеми синхронізації стану даних у мобільних offline-first застосунках із багатопристроєвим доступом. У роботі проаналізовано фундаментальні принципи розподілених систем, моделі остаточної узгодженості та сучасні підходи до інкрементальної реплікації даних.

Актуальність теми зумовлена широким використанням мобільних застосунків в умовах нестабільного мережевого з'єднання, а також припиненням підтримки керованих сервісів синхронізації, що створює потребу у власних, незалежних від постачальників алгоритмах.

Об'єктом дослідження є процес синхронізації даних у мобільних offline-first системах.

Предметом дослідження є алгоритми інкрементальної синхронізації, механізми фіксації змін на рівні властивостей об'єктів та методи розв'язання конфліктів.

Метою роботи є розроблення алгоритму синхронізації, оптимізованого під реальні сценарії використання, з підтримкою per-user узгодженості та мінімізацією втрат даних.

Наукова новизна полягає у запропонованому алгоритмі синхронізації на рівні окремих властивостей із використанням журналу змін.

Практичне значення підтверджується реалізацією для iOS та Android із урахуванням обмежень RealmSwift і Realm Kotlin.

Ключові слова: синхронізація даних, offline-first, мобільні застосунки, розподілені системи, журнал змін.

ABSTRACT

The master thesis consists of an introduction, three chapters, general conclusions, a list of references, and appendices. The main text covers 49 pages, includes 3 figures and references to 40 sources, and 17 pages of appendices.

This thesis addresses the problem of data synchronization in mobile offline-first applications with multi-device access. The study analyzes fundamental principles of distributed systems, eventual consistency models, and modern approaches to incremental data replication.

The relevance of the topic is driven by the widespread use of mobile applications under unreliable network conditions and by the deprecation of managed synchronization services, which necessitates the development of custom, vendor-independent solutions.

The object of the research is the data synchronization process in mobile offline-first systems.

The subject of the research includes incremental synchronization algorithms, property-level change tracking mechanisms, and conflict resolution methods.

The aim of the research is to design a synchronization algorithm optimized for real-world usage scenarios, supporting per-user consistency while minimizing data loss during synchronization failures.

Scientific novelty lies in the proposed property-level synchronization algorithm based on a change log mechanism.

The practical significance is demonstrated through implementations for iOS and Android platforms, taking into account the limitations of RealmSwift and Realm Kotlin.

Research methods include analysis and synthesis of theoretical materials, comparative evaluation of existing solutions, and experimental validation through practical implementation.

Keywords: data synchronization, offline-first, mobile applications, distributed systems, change log.

ЗМІСТ

ВСТУП

У сучасному світі мобільні застосунки є однією з найпоширеніших форм цифрової взаємодії. Користувачі очікують не лише надійної роботи застосунків, але й того, що їхні персональні дані залишатимуться синхронізованими між кількома пристроями. Наприклад, студент, який практикує вивчення лексики на смартфоні, очікує побачити ідентичний прогрес, відображений на планшеті, без ручного експорту чи імпорту інформації. В основі цієї вимоги лежить проблема розподіленої синхронізації даних, яка передбачає узгодження змін з від'єднаних реплік у послідовний і узгоджений спільний стан.

Технічне підґрунтя цієї проблеми добре відоме в галузі розподілених систем. Теорема CAP встановлює, що розподілене сховище даних не може одночасно гарантувати сильну узгодженість, доступність і стійкість до розділення мережі. Мобільні системи, які змушені витримувати розділення через необхідність роботи в офлайн режимі, неминуче надають пріоритет доступності. Таким чином, синхронізація не є тривіальною додатковою функцією, а виступає суттєвим архітектурним компонентом, що потребує ретельного проєктування. MongoDB — документоорієнтована NoSQL база даних — пропонує гнучкість, масштабованість та підтримку екосистеми, що робить її поширеним вибором для серверного управління даними. Realm, у свою чергу, забезпечує швидку вбудовану базу даних, придатну для мобільних середовищ, пропонуючи об'єктно-орієнтовану модель, яка добре інтегрується з платформами iOS та Android.

До недавнього часу MongoDB Atlas Device Sync надав інтегроване рішення синхронізації між Realm на мобільних пристроях і MongoDB Atlas у хмарі. Однак підтримка цього рішення була нещодавно припинена, що змушує розробників шукати альтернативні підходи. Виклик полягає не лише у відтворенні функціональності Device Sync, а й у переосмисленні синхронізації з позицій узагальнених, незалежних від постачальника принципів. Це передбачає проєктування алгоритму, який забезпечує, що кожен пристрій завантажує та передає лише дані, які належать автентифікованому користувачеві, є стійким до роз'єднань

тобто підтримує підхід offline-first, за якого зміни зберігаються локально та пізніше об’єднуються, а також виявляє та розв’язує розбіжні оновлення таким чином, щоб зберігати семантичний намір.

Цілі роботи:

1. Оглянути сучасний стан досліджень у галузі синхронізації баз даних для мобільних і NoSQL-контекстів, з акцентом на підходи, що підтримують offline-first реплікацію.
2. Спроєктувати алгоритм синхронізації, який використовує MongoDB і локальне сховище Realm для реалізації безпечної, ефективної, орієнтованої на конкретного користувача конвергенції стану.
3. Проілюструвати алгоритм у контексті конкретного прикладу реалізації.

у роботі запропоновано детальний алгоритм синхронізації. Запропонований дизайн демонструє, як усталені принципи розподілених систем можуть бути поєднані для створення практичної, готової до промислового використання системи.

Структура магістерської роботи є такою: у розділі 1 наведено огляд наявних досліджень і споріднених систем. Розділ 2 присвячено проєктуванню алгоритму синхронізації. Розділ 3 присвячений особливостям практичної реалізації алгоритму. Також в роботі зроблені висновки по дослідженню, підсумовані результати та окреслені напрями подальших досліджень.

1 ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ ДО СИНХРОНІЗАЦІЇ МІЖ ЦЕНТРАЛЬНОЮ ТА МОБІЛЬНИМИ БАЗАМИ ДАНИХ

Розділ містить огляд базових теоретичних положень і практичних систем синхронізації стану в розподілених і мобільних NoSQL базах даних. Обговорення ґрунтується на науковій літературі з питань узгодженості, розв’язання конфліктів та антиентропії, після чого оцінюються промислові рішення, релевантні для архітектури, у якій MongoDB виступає як централізована база даних, а Realm — як локальне сховище на пристрої. Метою є виявлення шаблонів проєктування та компромісів, які безпосередньо застосовні до синхронізації на рівні окремого користувача в застосунках з підходом offline-first.

1.1 Теоретичні основи синхронизації баз даних

Теорема CAP стверджує, що розподілена система не може одночасно гарантувати сильну узгодженість, високу доступність і стійкість до розділення мережі. З цього випливає, що мобільні та периферійні (edge) середовища, які за своєю природою повинні витримувати мережеві розділення, не можуть підтримувати сувору узгодженість без жертви доступності. TODO

Умови нестабільного або переривчастого з’єднання є типовими для мобільних застосунків, тому системи, що намагаються забезпечити сильну узгодженість за будь-яких обставин, неминуче стикаються з блокуванням операцій, що може призводити навіть до повної недоступності сервісу. З цієї причини практичні рішення орієнтуються на моделі остаточної або причинної узгодженості, роблячи акцент на збіжність стану після відновлення комунікації між вузлами [1], [2].

Остаточна узгодженість гарантує, що за відсутності нових записів усі репліки з часом досягнуть однакового стану. Хоча ця модель не забезпечує миттєвої узгодженості, вона добре відповідає вимогам масштабованих розподілених систем і широко використовується у великих веб-платформах [2]. Для мобільних сценаріїв такий підхід є особливо привабливим, оскільки дозволяє локальне виконання операцій без постійного зв’язку з сервером і відкладає узгодження до моменту синхронізації.

Поняття причинності формалізує відношення «відбувається-раніше» між операціями та дозволяє відрізняти причинно пов’язані зміни від справді конкурентних. Класичні механізми, такі як логічні годинники Лампорта та векторні годинники, забезпечують засоби для відстеження порядку подій у розподіленій системі та для безпечного застосування оновлень [3]. Використання цих механізмів дає змогу системам виявляти конкурентні оновлення, які не мають чітко визначеного порядку, і відповідно застосовувати політики розв’язання конфліктів або відкладати остаточне рішення до етапу злиття.

Водночас векторні годинники та їхні розширення, зокрема обмежені векторні версії, можуть створювати значні накладні витрати у вигляді метаданих, особливо в системах з великою кількістю реплік або користувачів. З практичних міркувань багато промислових реалізацій віддають перевагу узагальненим або гібридним підходам, які зменшують обсяг службової інформації, зберігаючи при цьому достатній рівень точності для виявлення конфліктів і забезпечення коректного порядку застосування змін [14].

Антиентропійні механізми є ще одним фундаментальним елементом синхронізації. Вони спрямовані на поступове усунення розбіжностей між репліками шляхом порівняння станів і обміну стислими уявленнями даних [13]. Підходи на основі дерев Меркла та обміну дайджестами широко застосовуються у сховищах типу «ключ–значення» та документоорієнтованих базах даних, оскільки дозволяють ефективно визначати області розбіжностей без передачі повного набору даних [11, 12]. Такі методи добре масштабуються, але часто потребують додаткових раундів обміну та складної логіки порівняння.

Як доповнення до антиентропії, дедалі більшого поширення набуває підхід Change Data Capture (CDC). Замість періодичного порівняння повного стану система фіксує та поширює впорядкований потік змін, які відбулися у базі даних. Клієнти підписуються на ці події та застосовують лише дельти, що значно зменшує обсяг переданих даних і спрощує відновлення стану після тимчасової відсутності з’єднання. Change Streams у MongoDB є характерним прикладом такого механізму, забезпечуючи впорядковану доставку подій і можливість відновлення з певної

позиції у потоці [19, 20]. У поєднанні з локальним сховищем на пристрої цей підхід створює основу для ефективної, масштабованої та орієнтованої на користувача синхронізації в offline-first застосунках.

Водночас застосування Change Streams має важливі практичні обмеження в мобільному контексті. Механізм Change Streams передбачає пряме підключення клієнта до сервера MongoDB та підтримку довготривалого з'єднання для отримання потоку подій. Для мобільних пристроїв такий підхід зазвичай вважається небажаною практикою з міркувань безпеки, керування доступом і стабільності з'єднання. Надання мобільному застосунку прямого доступу до бази даних у хмарі ускладнює контроль прав доступу, збільшує поверхню атаки та створює залежність від надійного постійного мережевого каналу.

З огляду на це, у реальних системах події Change Streams зазвичай потребують проксування через проміжний серверний рівень, наприклад через REST або GraphQL API. Такий бекенд виступає як контрольована точка доступу: він автентифікує користувачів, фільтрує події відповідно до їхніх прав і транслює лише релевантні зміни на клієнтські пристрої. Це додає архітектурної складності та затримок, а також вимагає додаткової логіки для буферизації та повторної доставки подій у разі втрати з'єднання. У цьому сенсі, хоча Change Streams виглядають концептуально привабливим і потужним механізмом, їх безпосереднє використання в мобільних застосунках є нетривіальним і часто потребує комбінування з іншими підходами до синхронізації.

1.2 Стратегії розв'язання конфліктів

Стратегії розв'язання конфліктів у розподілених системах охоплюють широкий спектр підходів — від операційного перетворення (Operational Transformation, OT) до безконфліктних реплікованих типів даних (Conflict-Free Replicated Data Types, CRDTs) [4]. Підходи на основі CRDTs забезпечують сильну збіжність стану завдяки використанню алгебраїчних функцій злиття, які є асоціативними, комутативними та ідемпотентними. Це дозволяє системам досягати однакового результату незалежно від порядку застосування оновлень і без потреби у глобальній координації або централізованому впорядкуванні операцій [5]. Саме ця

властивість робить CRDTs особливо привабливими для офлайн-орієнтованих і мобільних сценаріїв, де вузли можуть тривалий час працювати ізольовано.

Подальшим розвитком цього підходу є дельта-орієнтовані CRDTs, які оптимізують використання мережевих ресурсів за рахунок поширення компактних дельт стану замість повних знімків або детальних журналів операцій [7]. Такий підхід суттєво зменшує обсяг передаваних даних, що є критично важливим для мобільних пристроїв із обмеженою пропускною здатністю або високою вартістю трафіку [8]. Дельта-CRDTs зберігають ті самі властивості збіжності, що й класичні CRDTs, але водночас краще пристосовані до асинхронних і ресурсно обмежених середовищ.

Сучасні дослідження розширюють застосування CRDTs на більш складні структури даних, зокрема масиви, послідовності та багаті документні моделі, які характерні для реальних прикладних систем [15]. Такі розширення дають змогу моделювати складні об’єкти користувацького інтерфейсу та контенту без втрати властивостей збіжності, що відкриває можливості для використання CRDTs у текстових редакторах, системах керування знаннями та мобільних застосунках з насиченими моделями даних.

У практичних мобільних продуктових системах, однак, повсюдне використання CRDTs не завжди є доцільним. Часто найбільш ефективним виявляється гібридний підхід, який поєднує кілька стратегій розв’язання конфліктів залежно від семантики конкретних даних [9]. Наприклад, CRDTs добре підходять для комутативних агрегатів, таких як лічильники або множини, тоді як для полів, що відображають уподобання або налаштування користувача, може бути достатньо стратегії «останній запис перемагає» (last-writer-wins, LWW). Для більш складних сутностей, семантика яких пов’язана з накопиченням доказів або рівнем упевненості, застосовуються предметно-орієнтовані політики злиття, що враховують змістовне значення оновлень [10].

Ці шаблони безпосередньо застосовні до прикладу синхронізації на рівні окремого користувача в застосунку для вивчення словникового запасу. Зокрема, кількість повторень або переглядів слова природно моделюється за допомогою PN-

лічильників, які коректно об’єднують інкрементальні та декрементальні оновлення з різних пристроїв. Колекції тегів або категорій можуть бути реалізовані як OR-множини, що дозволяє безпечно додавати й видаляти елементи у конкурентному середовищі [5], [8]. Таким чином, поєднання різних стратегій розв’язання конфліктів забезпечує як формальну коректність синхронізації, так і практичну відповідність доменній логіці мобільного навчального застосунку.

1.3 Синхронізація видалення документів

Синхронізація операцій видалення є однією з найбільш проблемних і схильних до помилок задач у розподілених системах. На відміну від створення або оновлення даних, видалення не залишає матеріального об’єкта, який можна безпосередньо передати іншим реплікам, що створює ризик втрати інформації про сам факт видалення. Тому більшість систем використовують так звані «надгробки» (tombstones) — явні маркери видалення, які зберігаються протягом визначеного вікна ущільнення (compaction window). Ці маркери гарантують, що відстаючі або тимчасово від’єднані репліки зможуть спостерігати факт видалення під час подальшої синхронізації та не відтворять об’єкти, які були видалені раніше.

Проблематика видалень тісно пов’язана з класичними аномаліями, такими як «воскресіння» об’єктів, коли застаріла репліка повторно поширює вже видалений стан. Підходи, натхненні архітектурою Dynamo, використовують антиентропійні процедури ремонту для поширення інформації про видалення між вузлами, що дозволяє поступово усунути розбіжності [6]. Аналогічно, різновиди CRDT-множин формалізують семантику видалення таким чином, щоб операції додавання та видалення коректно поєднувалися навіть за конкурентних оновлень, гарантуючи відсутність неконсистентних станів після злиття [5].

Зберігання надгробків, однак, створює додаткове навантаження на систему, оскільки такі записи не можуть зберігатися необмежено довго без негативного впливу на обсяг сховища та продуктивність. Тому практичні реалізації зазвичай поєднують використання надгробків із періодичним ущільненням даних, під час якого застарілі маркери видалення остаточно прибираються після того, як вважається, що всі активні репліки вже «побачили» відповідну подію видалення.

У проєктах, побудованих на основі MongoDB, поширеною практикою є застосування так званих м’яких видалень (soft deletes), коли замість фізичного видалення документа встановлюється спеціальний атрибут, наприклад deleted: true. Такий підхід дозволяє зберігати семантичну інформацію про видалення в базі даних і водночас забезпечує коректну роботу механізмів синхронізації. Зокрема, Change Streams продовжують генерувати події, що відображають зміни стану документів, включно з логічними видаленнями, що є критично важливим для клієнтів, які відновлюють стан після періоду офлайн-роботи [19].

Періодичне серверне ущільнення або очищення таких «м’яко видалених» документів дозволяє з часом вивільняти сховище, коли всі активні клієнти гарантовано просунулися за горизонт відповідних надгробків. У результаті поєднання soft delete, Change Streams та контрольованої політики очищення формує збалансований підхід, який одночасно забезпечує коректність семантики видалення, запобігає аномаліям воскресіння даних і підтримує прийнятні витрати на зберігання у довготривалій перспективі.

1.4 Синхронізація на рівні окремого користувача

Синхронізація на рівні окремого користувача вимагає точного та однозначного обмеження області даних, щоб кожен клієнт реплікував виключно інформацію, яка належить його власнику. У контексті розподілених і мобільних систем це питання є не лише архітектурним, але й безпековим, оскільки помилки у визначенні області синхронізації можуть призвести до витоків даних або несанкціонованого доступу. Тому сучасні системи поєднують кілька перевірених шаблонів, спрямованих на забезпечення як коректності синхронізації, так і дотримання принципу найменших привілеїв.

Одним із базових і найпоширеніших підходів є фільтрація на рівні рядків або документів за ключем, що ідентифікує власника даних, зазвичай ownerId. У цьому випадку всі об’єкти в базі даних явно пов’язані з конкретним користувачем, а запити на читання та синхронізацію автоматично обмежуються відповідним значенням ідентифікатора. Такий підхід є концептуально простим і добре масштабується, проте

вимагає суворого контролю на серверному боці, щоб жоден клієнт не міг обійти фільтрацію або підмінити ідентифікатор власника.

Більш декларативні механізми авторизації реалізуються у вигляді політик доступу, які вбудовані в рівень синхронізації або API. Прикладами таких підходів є канали у Sync Gateway, правила безпеки у Firestore або директиви авторизації у GraphQL-схемах, зокрема в AppSync [21, 23, 26]. У цих моделях правила доступу описуються окремо від прикладного коду, що підвищує прозорість і зменшує ризик помилок реалізації. Декларативні політики також полегшують аудит і формальну перевірку того, що клієнти мають доступ лише до дозволених ресурсів.

Крім контролю операцій читання, критично важливими є серверні перевірки під час запису або «push»-операцій. Навіть якщо клієнт коректно обмежений у доступі до даних під час синхронізації, сервер повинен примусово відхиляти будь-які спроби модифікації об’єктів, що належать іншому користувачеві. Такі перевірки зазвичай виконуються на основі автентифікаційного контексту запиту та зіставлення ідентифікатора користувача з полем власника в даних, що запобігає міжкористувацьким мутаціям навіть у разі компрометації клієнтської логіки.

Додатковим рівнем захисту виступає шифрування даних. Обов’язковим є шифрування під час передачі (encryption in transit), тоді як для персонально ідентифікованої інформації (PII) або чутливих атрибутів може застосовуватися шифрування на рівні окремих полів. Такий підхід зменшує наслідки потенційних витоків і забезпечує відповідність вимогам регуляторів щодо захисту персональних даних.

У контексті MongoDB механізми Change Streams дозволяють реалізувати серверно-кероване обмеження області синхронізації під час отримання змін. Зокрема, фільтри $match можуть застосовуватися до поля fullDocument.ownerId або до метаданих змін, що гарантує доставку клієнту лише тих подій, які стосуються його власних даних [19]. Водночас кінцеві точки для запису повинні виконувати валідацію токенів автентифікації, наприклад перевіряти значення sub у JWT, перш ніж приймати будь-які зміни від клієнта. Поєднання цих механізмів забезпечує

цілісну модель безпеки, у якій персональне обмеження області даних є невід'ємною частиною архітектури синхронізації.

1.5 Огляд існуючих рішень для синхронізації

MongoDB підтримує серверну модель захоплення змін даних (Change Data Capture, CDC) за допомогою механізму Change Streams, який надає впорядкований журнал подій вставки, оновлення, видалення та заміни на рівні колекції, бази даних або всього кластера [19]. Такий журнал відображає послідовність змін у даних у майже реальному часі та слугує фундаментальною примітивою для побудови інкрементальної синхронізації між сервером і клієнтами.

Підписки на Change Streams підтримують агрегаційний конвеєр, що дозволяє виконувати вибіркову фільтрацію та проєкцію подій. Це є критично важливим для сценаріїв синхронізації на рівні окремого користувача, де видимість даних має бути обмежена, наприклад, за ідентифікатором власника (ownerId). Завдяки цьому сервер може гарантувати, що кожен споживач отримує лише ті події, які відповідають його області доступу, без потреби у додатковій обробці або фільтрації на клієнтському боці [19].

Важливою властивістю Change Streams є використання токенів відновлення (resume tokens), які дозволяють споживачам продовжувати обробку потоку змін після тимчасових збоїв або розривів з'єднання без втрати подій. Ця характеристика має особливе значення для мобільних клієнтів синхронізації, які часто працюють в умовах нестабільного мережевого середовища. Можливість точного відновлення з попередньої позиції в журналі змін забезпечує надійність і коректність інкрементальної доставки даних.

У порівнянні з підходами, заснованими на дифах стану або періодичних повних знімках, Change Streams забезпечують природну інкрементальну доставку змін, мінімізують кількість мережевих раундів і усувають потребу у дорогих операціях сканування повного стану після того, як клієнт успішно виконав початкову ініціалізацію [19], [20]. Це робить їх ефективним механізмом для масштабованих систем із великою кількістю клієнтів.

MongoDB Atlas Device Sync — сервіс, який історично надавав керовану синхронізацію між локальною базою даних Realm і MongoDB Atlas — був офіційно визнаний застарілим у 2024 році, а завершення його життєвого циклу заплановане на 2025 рік. В офіційних рекомендаціях зазначається, що локальна база даних Realm залишається доступною та підтримуваною, однак керований хмарний механізм синхронізації буде остаточно вилучено після настання EOL, що змушує кінцевих користувачів і команди розробників мігрувати на альтернативні або власні рішення [16–18].

Ця зміна має суттєвий вплив на команди, які покладалися на повністю керовану модель синхронізації, особливо в системах із вимогами до чіткої сегрегації даних між користувачами. Втрата готового сервісу синхронізації означає необхідність самостійного проєктування механізмів узгодження стану, розв’язання конфліктів і забезпечення безпеки. Водночас припинення підтримки Atlas Device Sync стимулює повернення до більш універсальних, незалежних від постачальника архітектурних рішень і портативних алгоритмів [16-18].

Також зробимо огляд інших існуючих рішень синхронізації стану центральної та локальних користувацьких баз даних.

Couchbase Mobile надає зрілий наскрізний стек для offline-first застосунків, побудований навколо вбудованої бази даних Couchbase Lite та серверного компонента Sync Gateway. Ця платформа підтримує контроль доступу на основі каналів і механізми дельта-синхронізації, що робить її добре пристосованою для сценаріїв реплікації на рівні окремого користувача [22]. Вбудовані засоби обмеження області даних дозволяють чітко визначати, які набори документів доступні конкретному клієнту, а безперервна синхронізація спрощує підтримання узгодженого стану між пристроями. Водночас Couchbase Mobile не є сумісним із MongoDB на рівні сховища даних. Його впровадження зазвичай вимагає міграції моделі даних, операційних інструментів і семантики запитів. Для організацій, які вже глибоко інтегровані в екосистему MongoDB, Couchbase Mobile радше означає зміну платформи, ніж використання як заміни «без заміни» (drop-in replacement).

Firebase Cloud Firestore пропонує синхронізацію в реальному часі у поєднанні з розвиненими офлайн-SDK та гнучкими правилами безпеки. Платформа вирізняється високою продуктивністю розробки та широкою міжплатформною підтримкою, що робить її привабливою для швидкого створення клієнтських застосунків [24]. Обмеження доступу на рівні окремого користувача може бути реалізоване за допомогою правил безпеки та відповідної організації колекцій. Як і у випадку з Couchbase, Firestore не є сумісним із MongoDB, а міграція на цю платформу потребує суттєвого перероблення моделей даних і серверної інтеграції. У результаті Firestore є особливо привабливим для нових (greenfield) проєктів, у яких пріоритетом є керована синхронізація та інструменти екосистеми, а не портативність бази даних.

Amplify DataStore у поєднанні з AWS AppSync забезпечує офлайн-синхронізацію на основі GraphQL, включно з виявленням і розв'язанням конфліктів за допомогою стандартних стратегій, таких як last-writer-wins, або користувацьких резолверів на базі AWS Lambda [25, 26]. Такий підхід органічно інтегрується з Amazon Cognito для реалізації автентифікації та авторизації на рівні користувача. З операційної точки зору цей стек орієнтований передусім на бекенди DynamoDB або Aurora, а не на MongoDB. Відповідно, його впровадження зазвичай передбачає прийняття ширшої AWS-орієнтованої архітектури та пов'язаної з нею інфраструктури.

ElectricSQL реалізує часткову реплікацію та механізми обробки конфліктів поверх PostgreSQL, дозволяючи створювати offline-first застосунки з локальним станом і подальшою конвергенцією в хмарі [27]. Для команд, відкритих до використання PostgreSQL, ця платформа пропонує концептуально вивірену основу та набір інструментів для побудови синхронізації. Водночас ElectricSQL не орієнтований на сховище MongoDB, а перехід на нього вимагає прийняття реляційної моделі даних і використання власного runtime середовища ElectricSQL, що може бути суттєвою архітектурною зміною.

Replicache пропонує модель синхронізації, засновану на клієнтському журналі мутацій, серверно-керованих операціях pull та клієнтських операціях push. Платформа є агностичною до конкретної бази даних, і серверна реалізація може бути

відносно безпосередньо побудована поверх MongoDB [28–30]. У цьому випадку розробники відповідають за створення серверних кінцевих точок, які виконують фільтрацію за користувачем, валідацію мутацій і обчислення мінімальних дифів для відповідей на pull-запити. Модель Replicache добре узгоджується з вимогами персонифікованої синхронізації та оптимістичних інтерфейсів користувача, проте її основна екосистема орієнтована насамперед на веб- і TypeScript-застосунки.

RxDB є реактивною клієнтською базою даних із плагінами реплікації для CouchDB та кастомних GraphQL-бекендів. Реплікація з використанням MongoDB є можливою через проміжний сервісний рівень, який транслює зміни у протокол реплікації RxDB [31–33]. Екосистема RxDB є особливо сильною для веб- і гібридних мобільних застосунків. У контексті нативних мобільних застосунків із використанням Realm RxDB радше слугує еталонним прикладом проєктування власних протоколів синхронізації, ніж безпосереднім компонентом архітектури.

WatermelonDB оптимізована для високої продуктивності в середовищі React Native та використовує задокументований власний протокол синхронізації, який передбачає реалізацію серверних кінцевих точок для операцій pull і push [34, 35]. Хоча локально база даних побудована на SQLite, сам протокол синхронізації та відповідні архітектурні підходи безпосередньо застосовні до побудови шлюзу синхронізації на основі MongoDB з підтримкою обмеження області даних на рівні окремого користувача.

Ditto робить акцент на синхронізації «edge-to-edge» та peer-to-peer, використовуючи канали Bluetooth, Wi-Fi або локальні мережі, з можливістю підключення хмарних компонентів за потреби [36, 37]. Такий підхід є особливо привабливим для сценаріїв, у яких польові команди працюють у середовищах із тривалими або частими від’єднаннями від мережі. Водночас для бекендів, суворо орієнтованих на MongoDB, Ditto фактично вводить паралельний шар зберігання даних, а інтеграція Ditto з MongoDB вимагає розроблення спеціалізованих механізмів узгодження та трансляції даних.

PowerSync надає шар офлайн-синхронізації, який зазвичай використовується у поєднанні з Supabase та PostgreSQL, і містить опубліковані рекомендації для команд,

що мігрують від Atlas Device Sync [38]. Це рішення є привабливим у випадках, коли перехід до екосистеми PostgreSQL є прийнятним з архітектурної та організаційної точки зору. В іншому разі PowerSync радше слугує еталонною моделлю для проєктування власних рішень синхронізації, ніж готовою заміною для MongoDB-орієнтованих систем.

ObjectBox пропонує вбудовану базу даних із додатковим комерційним сервісом синхронізації та активно позиціонує себе як альтернативу Realm і Device Sync [39, 40]. Як і у випадку з іншими рішеннями, не сумісними з MongoDB, впровадження ObjectBox зазвичай означає зміну платформи та потребує суттєвих зусиль із міграції даних і прикладної логіки.

1.6 Доцільність розробки власних алгоритмів синхронізації

Незважаючи на наявність численних зрілих фреймворків і керованих сервісів синхронізації, існують вагомі технічні, архітектурні та організаційні підстави для проєктування й упровадження власного алгоритму синхронізації.

Суттєвим чинником залишається довгострокова життєздатність і ризик залежності від постачальника. Частина рішень для синхронізації, особливо тих, що розробляються невеликими компаніями або спільнотами з обмеженим рівнем поширення, мають підвищений ризик припинення активного розвитку, зниження якості підтримки або повного згортання проєкту. Водночас навіть продукти великих і відомих компаній не застраховані від депрекації, стратегічних змін або припинення підтримки внаслідок бізнес-рішень. Побудова власного алгоритму синхронізації зменшує ці ризики та забезпечує незалежність системи від зовнішніх рішень, життєвий цикл яких не контролюється організацією.

Вирішальне значення часто має архітектурна автономія та повний контроль над серверною логікою. Готові платформи синхронізації, як правило, нав’язують певні моделі даних, правила реплікації та обмеження на рівні архітектури. Хоча такі абстракції спрощують початкову інтеграцію, вони можуть істотно ускладнювати реалізацію нетипових бізнес-вимог, доменно-орієнтованих стратегій розв’язання конфліктів або глибоку інтеграцію з наявними серверними компонентами. Власне рішення дозволяє повністю контролювати процеси автентифікації, авторизації,

валідації, аудиту та спостережуваності, узгоджуючи синхронізацію з внутрішніми стандартами та регуляторними вимогами.

Надмірна складність часто не виправдана реальними сценаріями використання. Для значної частини сучасних мобільних застосунків характерним є послідовне використання пристроїв одним користувачем, за якого справжні конкурентні редагування трапляються рідко. У таких умовах застосування повнофункціональних механізмів синхронізації з підтримкою складних протоколів, багатих метаданих і загальних стратегій узгодження може призводити до зайвих витрат ресурсів і ускладнення системи. Власний алгоритм синхронізації може бути свідомо спроєктований з урахуванням домінуючих сценаріїв, забезпечуючи простіші та ефективніші механізми узгодження стану без втрати коректності.

Передбачуваність поведінки та зручність налагодження є важливими перевагами індивідуальних рішень. Керовані системи синхронізації часто функціонують як «чорні ящики»: хоча вони декларують певні гарантії, аналіз нетипових збоїв, аномалій або проблем із продуктивністю може бути складним. Реалізація власного алгоритму робить логіку синхронізації прозорою та підконтрольною, що спрощує тестування, налагодження й формальне міркування щодо коректності, особливо в системах, де цілісність даних безпосередньо впливає на довіру користувачів.

Економічні міркування та операційна прозорість також можуть схиляти до розроблення власного рішення. Керовані сервіси зазвичай передбачають постійні витрати, пов’язані з обсягом даних, кількістю активних підключень або використанням пропрієтарної інфраструктури. Зі зростанням системи такі витрати стають менш передбачуваними та складнішими для оптимізації. Власна реалізація, розгорнута на наявній серверній інфраструктурі, дозволяє більш гнучко керувати ресурсами та приймати усвідомлені компроміси між вартістю й продуктивністю.

Нарешті, стратегічна гнучкість і портативність суттєво зростають у разі володіння власною логікою синхронізації. Алгоритм, побудований на загальнодоступних і стандартизованих примітивах, може бути адаптований до нових вимог, перенесений між інфраструктурними провайдерами або розширений для

підтримки нових платформ без прив’язки до конкретної екосистеми. Така гнучкість є особливо цінною для довготривалих продуктів і дослідницьких проєктів, де еволюція вимог є неминучою.

Отже, хоча наявні рішення для синхронізації забезпечують значну практичну цінність, особливо на ранніх етапах розроблення, вибір на користь власного алгоритму часто є обґрунтованим з огляду на вимоги до довгострокової стабільності, контролю, простоти для типових сценаріїв використання, передбачуваності поведінки, економічної ефективності та стратегічної незалежності.

1.7 Висновки до першого розділу

У першому розділі було закладено теоретичну та практичну основу дослідження синхронізації даних у мобільних розподілених системах з орієнтацією на offline-first і per-user сценарії. Показано, що синхронізація є ключовим архітектурним компонентом, який безпосередньо впливає на коректність, надійність і масштабованість системи.

Аналіз фундаментальних принципів розподілених систем, зокрема теореми CAP, моделей остаточної та причинної узгодженості, підтвердив неминучість компромісів у мобільних середовищах і доцільність підходів, що забезпечують збіжність стану після відновлення з’єднання. Розглянуті механізми розв’язання конфліктів, включно з CRDT і гібридними стратегіями, продемонстрували необхідність поєднання формальних гарантій із доменно-орієнтованою логікою даних.

Окрему увагу приділено питанням персонального обмеження області даних і безпеки, які є критичними для per-user синхронізації. Порівняльний огляд наявних рішень показав, що попри їхню зрілість, вони часто передбачають суттєві архітектурні компроміси або залежність від конкретних постачальників, що стало особливо актуальним у контексті припинення підтримки Atlas Device Sync.

Узагальнюючи, перший розділ обґрунтовує необхідність розроблення власного, портативного алгоритму синхронізації, адаптованого до реальних сценаріїв мобільних застосунків. Отримані висновки слугують основою для

подальшого проєктування архітектури та опису запропонованого алгоритму в наступних розділах роботи.

2 ПРОЕКТУВАННЯ АЛГОРИТМУ СИНХРОНИЗАЦІЇ СТАНІВ БАЗ ДАНИХ

2.1 Основні сценарії використання баз даних, що потребують синхронізації

Проєктування власного алгоритму синхронізації доцільно починати з чіткого визначення базових сценаріїв використання, оскільки саме вони визначають необхідний рівень складності механізмів узгодження стану. Для більшості мобільних застосунків характерні обмежена кількість типових патернів взаємодії користувача з даними, і коректна ідентифікація цих патернів дозволяє уникнути надмірно ускладнених рішень. У межах цієї роботи розглядаються два ключові сценарії, які покривають переважну більшість практичних випадків.

Перший сценарій є найбільш поширеним і відповідає послідовному використанню пристроїв одним користувачем. У цьому випадку користувач взаємодіє з даними на одному пристрої за раз, наприклад спочатку на смартфоні, а пізніше — на планшеті або іншому мобільному пристрої. За умови коректної синхронізації після завершення сесії всі зміни встигають бути передані на сервер до початку роботи на іншому пристрої. У такій моделі практично відсутні справжні конкурентні оновлення, а конфлікти мають винятковий характер. Відповідно, для забезпечення коректності достатньо механізму узгодження стану на рівні «останній узгоджений знімок», без необхідності зберігати детальний журнал змін або застосовувати складні алгоритми злиття. Такий підхід дозволяє мінімізувати обсяг метаданих, спростити реалізацію та знизити накладні витрати як на клієнтському, так і на серверному боці.

Другий сценарій передбачає квазіпаралельне або одночасне використання кількох пристроїв одним користувачем. У реальних умовах такий сценарій трапляється значно рідше, проте він може виникати як наслідок збоїв синхронізації, зокрема через нестабільне або відсутнє інтернет-з’єднання. У цьому випадку користувач може виконувати зміни на кількох пристроях, не

отримуючи актуального стану з сервера, що з погляду системи виглядає як конкурентне редагування. Саме цей сценарій потребує наявності механізмів фіксації змін, виявлення конфліктів і подальшого коректного злиття станів після відновлення зв’язку.

Важливо підкреслити, що другий сценарій не обов’язково означає реальну одночасну активність користувача. Навіть за послідовного використання пристроїв тимчасові затримки синхронізації можуть призвести до ситуацій, які логічно еквівалентні паралельним оновленням. Тому алгоритм синхронізації має бути спроєктований таким чином, щоб оптимізуватися під перший, домінуючий сценарій, але водночас коректно обробляти другий як граничний випадок.

Таким чином, виділення цих двох сценаріїв створює основу для збалансованого проєктування алгоритму синхронізації. Воно дозволяє поєднати простоту й ефективність у типових умовах із достатньою стійкістю та коректністю в разі збоїв або нетипових режимів використання. Саме на цьому компромісі між оптимізацією для найпоширенішого випадку та підтримкою складніших ситуацій базується подальший дизайн запропонованого алгоритму.

2.2 Спрощений алгоритм синхронізації даних у розподілених мобільних системах

Спрощений алгоритм синхронізації даних у розподілених мобільних системах постає як компромісне рішення між повноцінною подієво-орієнтованою реплікацією та відсутністю механізмів узгодження змін взагалі. Його ключовою характеристикою є відсутність серверного журналу змін та орієнтація на єдиний часовий маркер — момент останнього оновлення об’єкта. Така модель дозволяє реалізувати синхронізацію з мінімальними обчислювальними витратами та без складних операцій відстеження конфліктів на рівні окремих властивостей. У даному підрозділі детально висвітлено архітектуру, механізми застосування змін, особливості оброблення «застарілих» пакетів, а також межі застосовності алгоритму.

При реалізації спрощеного алгоритму доцільно використання логіки «останній запис перемагає» на рівні цілого об’єкту (чи документу) бази даних. Це означає, що будь-яка зміна вважається атомарною: якщо об’єкт має позначку часу, новішу за дані, що надходять від клієнта, то весь пакет оновлень ігнорується, навіть якщо окремі його властивості залишаються актуальними. Така поведінка є спрощенням загальної моделі LWW, однак виправдана для застосунків, у яких користувачі не змінюють один і той самий об’єкт паралельно з різних пристроїв.

Клієнтський застосунок має гарантувати, що перед надсиланням змін виконується зчитування останнього локального стану, а всі модифіковані властивості фіксуються у вигляді компактного JSON-об’єкта. Крім того, клієнт зобов’язаний передавати власний lastSyncAt, який сервер використовує для формування дельт-відповідей. Якщо клієнт працює у стані офлайн упродовж тривалого часу, то обсяг отриманих дельт може бути значним, що слід враховувати при проєктуванні UI та локального кешу. Також має бути гарантовано, що кожний об’єкт має глобально унікальний ідентифікатор, що можна досягнути завдяки використанню GUID у якості первинного ключа.

Структура даних на сервері має наступні особливості. Колекція objects у MongoDB репрезентує поточний консолідований стан. Вона містить: унікальний _id, набір властивостей, а також поле lastChangedAt, що є єдиним критерієм для валідації змін. На відміну від розширених варіантів алгоритму, не зберігається історія попередніх станів, не ведеться журнал ключ-значення, не застосовується revision. Це значно спрощує модель даних, але унеможливлює виконання ретроспективних операцій та аналітики й ускладнює аудит.

Оброблення запиту на синхронізацію виглядає наступним чином. Після отримання HTTP-запиту функція синхронізації: (1) десеріалізує пакет; (2) для кожного об’єкта витягує поточний документ з бази; (3) порівнює часову позначку; (4) або застосовує оновлення, або відкидає його. Усі відкинуті зміни інтерпретуються як наслідок наявності «новішої» версії на сервері, і тому сервер повертає її клієнтові. Така поведінка гарантує узгодженість без конфліктних

гілок редагування. Оскільки повторне надсилання того самого батча не повинно змінювати стан системи, ідемпотентність досягається природним чином: кожен пакет завжди перевіряється на актуальність незалежно від того, чи бачив його сервер раніше. Таким чином, не потрібні requestId, clientToken або інші механізми запам’ятовування оброблених транзакцій.

Алгоритм залежить від коректності часової позначки, яку надсилає клієнт. У разі, якщо годинник пристрою сильно «відстає» або «випереджає» UTC, сервер може або відхилити коректну зміну, або прийняти застарілу як новішу. Це є системним ризиком моделі, який ускладнюється тим, що сервер не має змоги перевірити справжність timestamp без додаткових механізмів синхронізації годинника.

На відміну від повної заміни стану, дельта-механізм формує відповідь, що містить лише ті об’єкти, які змінювалися після моменту lastSyncAt клієнта. Таким чином, сервер повертає мінімальну кількість даних, забезпечуючи економію трафіку. Водночас клієнт самостійно оновлює локальну БД.

Алгоритм є цілком прийнятним для застосунків, де:

- об’єкти редагуються рідко;
- користувач взаємодіє з одним пристроєм;
- конфлікти вважаються малоймовірними, а у разі, якшо вони трапляються, то негативний ефект від них настільки мінімальний, що може бути проігнорованим;
- журнал змін не є обов’язковим для аудиту.

Проте для систем співредагування, високої частоти змін модель непридатна. Перевагою цього алгоритму є те, що завдяки однопрохідній перевірці дат зміни та відсутності журналу змін алгоритм досягає високої продуктивності й низької собівартості оброблення запитів. На рисунку 2.1. представлена діаграма послідовності, що ілюструє роботу алгоритму. Спрощений алгоритм розрішення конфліктів наведений на рисунку 2.2.

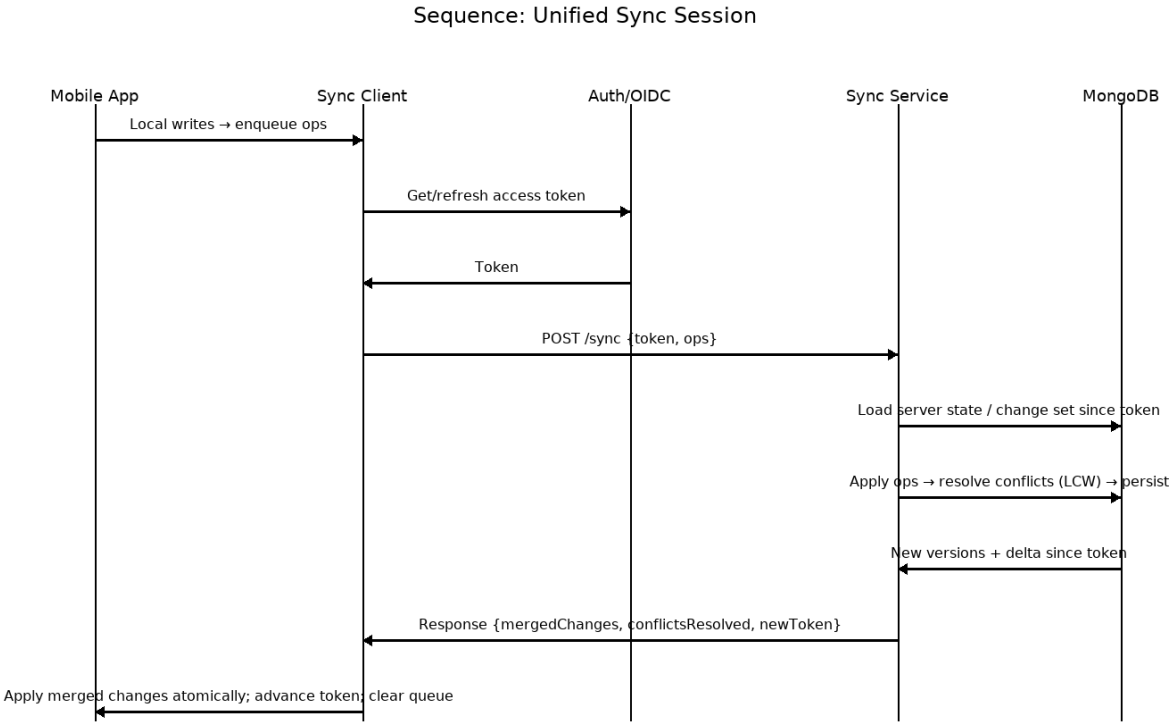

Рисунок 2.1 – UML діаграма послідовності для сеансу синхронізації

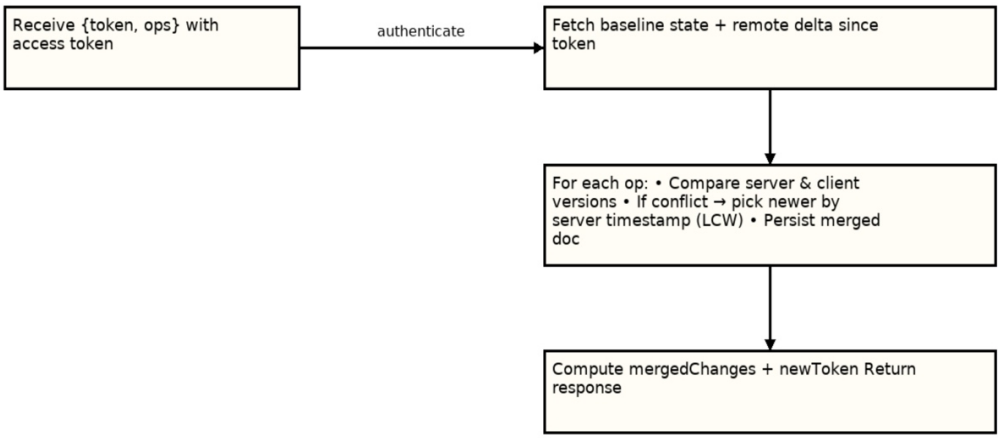

Рисунок 2.2 – Спрощений алгоритм розрішення конфліктів

2.3 Алгоритм синхронізації на основу журналу змін

Розширений алгоритм синхронізації, побудований на основі збереження історії змін окремих властивостей об’єкта, є логічним розвитком спрощеної моделі об’єктно-орієнтованого LWW-узгодження. Його доцільність зумовлена тим, що у більшості реальних сценаріїв користувач не змінює весь об’єкт одразу, а редагує лише частину полів, тоді як інші можуть паралельно змінюватися

іншими пристроями. Відмова від журналу змін робить неможливим коректне відновлення намірів користувача, а також призводить до втрати значень, які були відкинуті через незначні відмінності у часових позначках. Тому серверний рівень у даній моделі зберігає не лише консолідований стан об’єкта, але і останні змінені значення для кожного його поля, що дає змогу замінити грубе «перезаписати весь об’єкт» на вибіркове «оновити лише ті властивості, для яких зміна є найновішою».

Основою розширеного алгоритму є журнал змін. Журнал змін (changeLog) виконує роль незалежного реєстру найновіших версій окремих властивостей об’єкта. Його зміст не дублює повний історичний ланцюжок редагувань, а лише фіксує останню актуальну версію. Під час синхронізації сервер не порівнює об’єкти цілком, а звіряє зміну на рівні ключа (propertyKey) для конкретного objectId, що усуває зайві конфлікти. Це дозволяє зберегти консистентність стану навіть у випадку, коли два пристрої змінюють різні поля одного й того ж документа у різні моменти часу.

Колекція objects містить консолідований стан документа, тоді як колекція changeLog зберігає метадані для кожної властивості. Приклад запису журналу:

{ objectId, property, value, valueType, changedAt, revision }.

У типовому сценарії зберігається лише один запис на властивість, тобто обсяг журналу зростає лінійно від кількості змінюваних полів, а не від кількості синхронізацій. Для забезпечення ефективності потрібне індексування за (objectId, property) та окремий індекс за changedAt для вибірок дельт.

Огляд алгоритму синхронізації:

1. Клієнт надсилає лише змінені властивості з датами їх змін (так само, як і для спрощеного алгоритму).

2. Сервер для кожної властивості виконує пошук у журналі змін.

3. Якщо зміна новіша за дату останнього оновлення відповідного документу, то вона застосовується до відповідних об’єктів і записується до журналу.

4. Якщо зміна старіша — вона відкидається, лише у випадку, якщо у журналі змін є запис про зміну властивостей, який новіший за цю зміну. В іншому випадку вона все одно застосовується та записується до журналу змін.

5. При записі нової зміни до журналу всі попередні зміни, що стосуються відповідної властивості та об’єкту з журналу видаляються.

6. Після оброблення всього пакета сервер формує дельту, обмежену датою останньої синхронізації, яка була передана клієнтом.

Цей підхід дозволяє застосовувати зміни інкрементально, без перезапису неактуальних полів. Такий механізм забезпечує суттєве зменшення кількості логічних конфліктів у порівнянні з об’єктним LWW-підходом, оскільки конкуренція виникає лише на рівні конкретної властивості, а не всього документа. У результаті система здатна коректно об’єднувати незалежні зміни, виконані на різних пристроях, без втрати даних і без необхідності залучення складних глобальних механізмів упорядкування операцій. Це особливо важливо для мобільних сценаріїв, у яких затримки синхронізації та короткочасні від’єднання є нормою, а справжні семантичні конфлікти трапляються відносно рідко. Водночас запропонований алгоритм зберігає помірну складність реалізації. На відміну від повноцінних CRDT-структур або журналів операцій, журнал змін властивостей не потребує збереження повної історії редагувань і не накладає вимог до глобальної унікальності операцій. Це спрощує серверну логіку, зменшує обсяг метаданих і полегшує подальше обслуговування системи. Видалення попередніх записів для тієї самої властивості дозволяє контролювати зростання changeLog і гарантує, що журнал відображає лише актуальний стан узгодження.

Таким чином, розширений алгоритм займає проміжне положення між спрощеним LWW-підходом і більш складними реплікованими моделями. Він оптимізований під домінуючий сценарій послідовного використання пристроїв, але водночас коректно обробляє квазіпаралельні оновлення, спричинені збоєм синхронізації або тимчасовою відсутністю мережі. Це робить його практичним компромісом між простотою, ефективністю та коректністю для реальних

мобільних застосунків. UML діаграма послідовностей для зазначеного алгоритму наведена на рисунку 2.3.

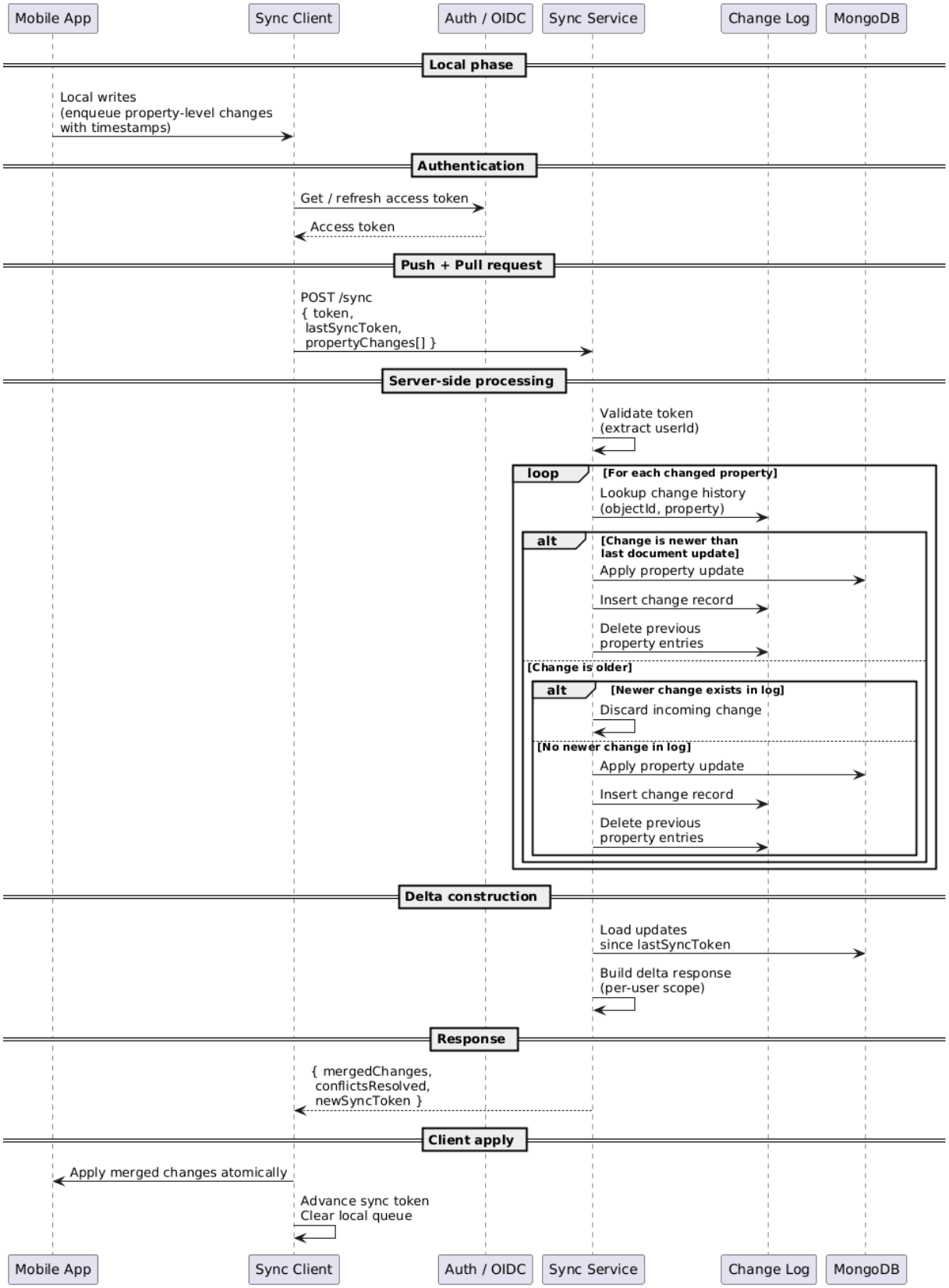

Рисунок 2.3 – UML діаграма послідовності для сеансу синхронізації з використанням журналу змін

2.4 Висновки до другого розділу

У другому розділі було розроблено концептуальний дизайн власного алгоритму синхронізації, орієнтованого на реальні сценарії використання мобільних застосунків і вимоги per-user узгодження стану. Вихідною точкою проєктування стало чітке розмежування двох базових сценаріїв: послідовного використання пристроїв, який є домінуючим у практиці, та квазіпаралельного редагування, що виникає внаслідок збоїв синхронізації або нестабільного мережевого з’єднання.

Запропонований розширений алгоритм ґрунтується на рівні окремих властивостей об’єкта та використовує журнал змін як механізм фіксації актуальних оновлень. Такий підхід дозволяє уникнути грубого об’єктного перезапису, зменшити кількість хибних конфліктів і зберегти наміри користувача навіть у разі конкурентних змін різних полів одного документа. Важливо, що журнал змін не зберігає повну історію операцій, а містить лише останні релевантні значення, що забезпечує контрольований обсяг метаданих і передбачувану складність системи.

Також у розділі було показано, що інкрементальна обробка змін і формування дельт на сервері дозволяють ефективно масштабувати синхронізацію та мінімізувати мережеві витрати. Алгоритм свідомо оптимізований під найпоширеніший сценарій послідовного використання, але водночас коректно обробляє граничні випадки, які логічно еквівалентні паралельному редагуванню.

3 ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ АЛГОРИТМУ СИНХРОНІЗАЦІЇ ПРИ ВИКОРИСТАННІ MONGO DB ТА REALM

3.1 Реалізація алгоритму для платформи iOS

У сучасних мобільних застосунках, орієнтованих на офлайн-роботу та багатопристроєвий доступ, особливе значення має питаннякоректної фіксації змін на стороні клієнта. Користувач може змінювати об’єкти даних у будь-якому стані мережевого з’єднання, і застосунок має гарантувати, що жодна зміна не буде втрачена, дубльована або перезаписана без контролю. На платформі iOS цю задачу можна розв’язати на основі RealmSwift —продуктивного локального сховища, яке на відміну від класичних SQLite-рішень, реалізує подієву модель зміни об’єктів. Саме ця властивість дозволяє побудувати автоматичний журнал змін, що фіксує оновлення властивостей без ручного втручання програміста. Такий підхід підвищує надійність, спрощує синхронізаційний алгоритм і зменшує кількість переданих даних, оскільки замість повного об’єкта передається лише мінімальна дельта-інформація.

RealmSwift надає два основні механізми, що роблять можливою автоматичну реєстрацію змін: Key-Value Observing (KVO) та внутрішні сповіщення про модифікації об’єктів. KVO спрацьовує на зміну значення властивості класу, що наслідує NSObject, а Realm розширює цю модель, дозволяючи відслідковувати зміни не тільки на рівні окремого поля, але й під час усієї транзакції. Отже, після завершення будь-якого запису в базу даних, розробник отримує структуровану інформацію про всі змінені властивості, що виключає необхідність самостійно порівнювати старий та новий стан. На практиці це означає, що код, відповідальний за журналювання, може бути повністю універсальним і не залежати від бізнес-логіки або кількості моделей.

Кожна зареєстрована зміна зберігається у вигляді окремого запису ModelUpdateEntry. На відміну від повного об’єкта, він містить мінімальний набір інформації: унікальний ідентифікатор, назву зміненої властивості, тип значення

та його серіалізовану форму. Це дозволяє передавати всі зміни у вигляді компактних JSON-масивів, що надзвичайно важливо для мобільних застосунків із нестабільним з'єднанням. Реалізація включає алгоритм insertOrUpdate, що гарантує, що для кожної властивості зберігається лише останнє відоме значення. Таким чином, якщо властивість змінюється кілька разів до синхронізації, до сервера буде передано лише фінальний стан, що зменшує навантаження на мережу та сервер.

Ключовим елементом реалізації є використання механізму observe(), який дозволяє отримати список змін властивостей у межах транзакції Realm. На відміну від KVO, який реагує на кожну зміну окремо, observe() повертає агрегований список, у якому вже здійснено диференціювання за полями. Це дозволяє уникнути зайвих викликів та зменшує витрати пам'яті. У ході реалізації було вирішено ігнорувати службові поля, а також властивості, визначені як «статичні» або такі, що не впливають на логіку синхронізації. Це робить систему розширюваною: нові властивості можна додати без оновлення механізму журналювання. Основні класи, що реалізують ведення журналу змін для iOS застосунку наведені у додатку 1.

Реалізація журналу змін на iOS демонструє, що автоматизація виявлення змін значно знижує складність розробки та дозволяє створити надійний офлайн-орієнтований застосунок без жорсткої прив'язки до UI-логіки. Завдяки поєднанню механізмів KVO та подієвої моделі RealmSwift, розробник отримує готовий механізм збору інкрементальних змін, який не потребує дублювання коду, не створює надмірного навантаження та повністю сумісний із серверним алгоритмом синхронізації, що використовує порівняння відміток часу по властивостях.

Наведена реалізація конкретизує описаний вище підхід автоматичного журналювання змін і демонструє, яким чином теоретична модель інкрементальної фіксації властивостей може бути безпосередньо реалізована на платформі iOS. Архітектурно рішення базується на введенні спільного базового класу BaseModel, від якого наслідуються всі доменні об'єкти, що беруть участь

у синхронізації. Такий підхід дозволяє інкапсулювати логіку спостереження за змінами та їх реєстрації в одному місці, усуваючи необхідність дублювання коду в кожній окремій моделі даних.

Ключовим елементом реалізації є використання механізму спостереження Realm через метод observe(), який надає агрегований список змін, здійснених у межах транзакції. На відміну від класичного KVO, що реагує на кожну зміну окремо, цей механізм дозволяє отримати вже структурований перелік змінених властивостей. У коді це відображено через обробку масиву changeList, кожен елемент якого приводиться до уніфікованого протоколу VocPropertyChangeProtocol. Завдяки цьому механізм журналювання є типовим і не залежить від конкретного типу об’єкта чи бізнес-контексту.

Функція processChanges реалізує логіку перетворення подієвих змін Realm у записи журналу змін. Для кожної підтримуваної категорії типів (Int, Bool, String, Date, списки та вкладені об’єкти) створюється екземпляр ModelUpdateEntry, який містить лише мінімально необхідну інформацію для синхронізації: ідентифікатор об’єкта, назву властивості, тип значення та його серіалізовану форму. Така нормалізація даних є принциповою, оскільки забезпечує незалежність журналу змін від конкретної структури доменних моделей і спрощує подальшу передачу змін на сервер.

Окремої уваги заслуговує механізм insertOrUpdate, який гарантує, що в локальному журналі зберігається не повна історія змін, а лише останній актуальний стан кожної властивості. Якщо властивість змінюється кілька разів між двома синхронізаціями, попередні значення перезаписуються, а до сервера зрештою передається лише фінальний результат. Це безпосередньо відповідає проєктним вимогам алгоритму синхронізації, описаного у попередньому розділі, та дозволяє суттєво зменшити як обсяг переданих даних, так і кількість конфліктів, що потребують оброблення на серверному боці.

Реалізація також передбачає механізми фільтрації змін, які не повинні впливати на синхронізацію. Метод keyPathsForIgnoreUpdateRegistration дозволяє виключати службові або похідні поля, а перевірка intProperties() дає змогу

коректно обробляти числові властивості, значення яких можуть мати спеціальну семантику. Завдяки цьому система залишається розширюваною: додавання нових властивостей або моделей не вимагає змін у базовому механізмі журналювання.

У підсумку представлена реалізація демонструє, що автоматичне виявлення та реєстрація змін на рівні властивостей є практично здійсненним і добре узгоджується з архітектурою RealmSwift. Поєднання подієвої моделі локального сховища з універсальним журналом змін дозволяє реалізувати надійний клієнтський компонент синхронізації, який не залежить від UI-логіки, не потребує ручного порівняння станів і повністю відповідає серверному алгоритму, що базується на порівнянні часових міток для окремих властивостей.

3.2 Реалізація алгоритму для платформи Android

На відміну від реалізації RealmSwift, що підтримує подієве сповіщення на рівні окремих властивостей, Realm Kotlin не надає механізму автоматичного перехоплення змін. Єдине доступне сповіщення стосується факту модифікації об’єкта цілком, що є непридатним для формування дельта-наборів. Тому фіксація змін виконується вручну, шляхом виклику спеціалізованої функції, якій передаються назви оновлених полів.

Kotlin реалізація так само базується на окремій сутності ModelUpdateEntry, що містить лише мінімальний набір метаданих: ідентифікатор об’єкта, назву властивості, тип та серіалізоване значення. Ведення журналу здійснюється за принципом insertOrUpdate, тобто для кожної властивості зберігається лише остання відома модифікація. Такий підхід мінімізує обсяг даних, що передається серверу, та відповідає стратегії «останній запис перемагає».

Функція registerChanges(...) викликається після завершення операції редагування моделі. Зчитування значень відбувається через відображення (reflection), що забезпечує універсальність та незалежність від структури конкретного класу. Кожна зміна трансформується у запис журналу, що дозволяє накопичувати модифікації до моменту синхронізації.

Після формування пакета змін виконується перевірка на предмет появи нових модифікацій у період підготовки передачі. За потреби пакет перегенеровується, що унеможливлює втрату даних. Це забезпечує властивість причинної узгодженості (causal consistency) між локальною та серверною моделями.

Фрагмент реалізації представлений у додатку 2. Наведений фрагмент реалізації для Android демонструє принципово іншу інженерну модель ведення журналу змін у порівнянні з iOS-варіантом. Ключовою причиною відмінностей є обмеження Realm Kotlin: на відміну від RealmSwift, він не надає подієвого механізму, який повертає перелік змінених властивостей у межах транзакції. У наявному API фактично доступним є лише факт модифікації об’єкта загалом, що не дозволяє автоматично отримати мінімальну дельту для синхронізації. Саме тому реалізація змушена переносити відповідальність за фіксацію змін з рівня базового класу на рівень конкретної моделі та її сеттерів.

У запропонованому дизайні доменний клас WordsSet виконує роль не лише структури даних, а й активного джерела подій синхронізації. Для цього в кожному сеттері властивостей (name, imageUrl, order, isDeleted тощо) викликається метод registerChange(...), який у підсумку делегує виконання до registerChanges(...). Таким чином, реєстрація модифікацій стає частиною контракту моделі: будь-яка зміна стану, виконана через публічні властивості, автоматично породжує запис у локальному журналі. Це дозволяє наблизити поведінку до iOS-варіанту, хоча й за рахунок більшої «ручної» інструментації коду.

Обмеження Realm Kotlin також пояснює, чому не можна винести цю логіку в універсальний BaseModel для всіх моделей застосунку. У RealmSwift базовий клас може централізовано перехоплювати зміни через observe() і обробляти їх незалежно від конкретної моделі. У Realm Kotlin таке узагальнення є практично недосяжним без істотних компромісів: система не надає універсального списку змінених полів, а отже базовий клас не може «дізнатися», які саме властивості були модифіковані. У результаті будь-яка спроба реалізувати автоматичний

облік змін у базовому класі зводилася б або до повного знімка об’єкта (що суперечить вимогам дельта-синхронізації), або до інвазивного ручного опису всіх полів у кожному класі, що фактично і реалізовано, але вже локально в доменних моделях.

Практична реалізація формування запису журналу змін представлена класом ModelUpdateEntry, який містить стандартний мінімальний набір атрибутів (id, name, type, value). Метод getValue() відображає зворотне перетворення серіалізованого значення в конкретний тип під час формування JSON-дельти або застосування змін. Використання типового поля type дозволяє уніфікувати зберігання різних типів значень у єдиній колекції, що спрощує запити до локального сховища і забезпечує незалежність протоколу синхронізації від конкретних класів.

Функція registerChanges(fields: Array<String>) реалізує механізм «зняття» поточних значень властивостей за їх назвами. Для універсальності використовується відображення (reflection): метод getProperty(name: String) знаходить член класу за його ім’ям і повертає поточне значення. Хоча reflection має вищу вартість виконання у порівнянні зі статично типізованим доступом, у даному випадку він виправданий тим, що дозволяє описувати перелік змінюваних полів декларативно (через масив імен) та не прив’язує механізм журналювання до конкретного набору властивостей на рівні коду синхронізації.

Важливою деталлю реалізації є те, що в registerChanges(...) значення читаються не з поточного інстансу об’єкта, а з «останньої» версії, отриманої з Realm (query<WordsSet>("id == $0", id).find()). Такий підхід має методологічне значення: він гарантує, що в журнал записується стан, який фактично зафіксований у локальному сховищі, а не проміжний стан об’єкта, який може бути змінений у пам’яті до завершення транзакції. Відповідно, журнал змін відображає саме персистентний стан, що узгоджується з серверною моделлю оброблення дельт і зменшує ризики некоректного порядку застосування оновлень.

Аналогічно до iOS-реалізації, на Android використовується політика insertOrUpdate, реалізована у приватному методі insertOrUpdate(entry: ModelUpdateEntry). Якщо запис для пари (objectId, propertyName) відсутній, він створюється; якщо присутній — оновлюється лише значення. Таким чином, журнал накопичує не історію, а останній ефективний стан кожної властивості. Це дозволяє уникати росту журналу при багаторазовому редагуванні одного поля до синхронізації та безпосередньо зменшує навантаження на мережу при формуванні пакета змін.

Окремий аспект реалізації стосується керування побічними ефектами під час застосування серверних змін на клієнті. Для цього введено прапорець autoRegisterChanges: перед виконанням масового оновлення з JSON (метод update(...)) він тимчасово вимикається, що запобігає повторній реєстрації змін, які насправді є результатом синхронізації, а не дій користувача. Після завершення оновлення прапорець повертається у вихідний стан. Такий механізм є критичним для уникнення «петель синхронізації», коли отримані з сервера значення одразу повторно потрапляють у журнал і знову відправляються на сервер.

Нарешті, метод json(updatesOnly: Boolean = false) демонструє, як журнал змін інтегрується у формування мережевого протоколу. У режимі updatesOnly=true об’єкт не серіалізується повністю; натомість виконується вибірка ModelUpdateEntry за id, після чого ключі переводяться у компактні скорочення через мапу nameMap. Це забезпечує мінімізацію розміру повідомлень і робить синхронізацію економною для мобільних мереж. Додатково забезпечується включення часу оновлення (u) як обов’язкового метаданого, що є необхідним для серверного порівняння та застосування LWW-логіки на рівні властивостей.

У підсумку, Android-реалізація підтверджує, що за відсутності в Realm Kotlin механізмів подієвого відстеження змін на рівні властивостей, система синхронізації повинна бути спроєктована з урахуванням ручної реєстрації оновлень. Саме тому універсальна реалізація на рівні базової моделі, аналогічна

RealmSwift, є непридатною або надмірно складною: необхідна інформація про змінені поля недоступна на рівні платформи. Запропонований підхід із викликами registerChange у сеттерах та контролем autoRegisterChanges дозволяє досягти еквівалентної функціональності формування дельт, зберігаючи сумісність із серверним алгоритмом синхронізації та забезпечуючи прийнятний рівень надійності у реальних умовах експлуатації.

3.3 Реалізації алгоритму синхронізації на стороні серверу

У контексті офлайн-орієнтованих мобільних систем серверний компонент синхронізації виконує роль остаточного арбітра узгодженості станів. На відміну від клієнтської частини, де фіксуються інкрементальні модифікації, сервер має забезпечити детерміноване застосування змін, відстеження причинно-часових взаємозв’язків, а також розв’язання конфліктів між кількома пристроями користувача. У даному розділі викладено принципи та практичні аспекти реалізації серверної частини на основі Azure Functions (ізольована модель виконання) та MongoDB як сховища даних. Метою є визначення архітектури, здатної приймати батчі змін від клієнтів iOS/Android, застосовувати політику «last-write-wins» на рівні властивостей об’єкта, підтримувати журнал змін для кожного документа та повертати клієнтам мінімальні дельти, необхідні для приведення локальної бази до узгодженого стану.

Серверна підсистема має задовольняти низку вимог, обумовлених характером офлайн-практик. По-перше, необхідна ідемпотентність обробки запитів: повторне надсилання того самого батча не повинно призводити до дублювання або неконсистентних станів. По-друге, потрібна підтримка часткових оновлень документів з точністю до властивості, що спрощує застосування політики «last-write-wins» та зменшує мережеве навантаження. По-третє, система повинна фіксувати момент останньої зміни як на рівні об’єкта загалом, так і на рівні окремих полів (через спеціалізований журнал змін). По-четверте, API має бути придатним для горизонтального масштабування: кожна функція повинна бути безстанною (stateless), а всі необхідні дані —

зберігатися у MongoDB з відповідними індексами. Нарешті, потрібні механізми захисту від «застарілих» або «несумісних» клієнтів, які можуть надсилати дані зі значними часовими зсувами або з відсутніми критичними полями.

Azure Functions доцільно використовувати у вигляді набору HTTP-тригерів для оброблення операцій «/sync», «/fetch-delta» та «/health». Кожна функція працює у відокремленому середовищі на .NET 8 (ізольований процес), що спрощує ін'єкцію залежностей, логування та тестування. Конфігурація підключення до MongoDB виконується через Azure Key Vault або керовані секрети, тоді як пул з'єднань MongoClient ініціалізується один раз під час старту хоста. Організація коду передбачає виділення шарів: (1) HTTP-контролер (function endpoint), (2) сервіс доменної логіки синхронізації, (3) репозиторій доступу до MongoDB. Така декомпозиція підвищує тестованість, оскільки бізнес-правила синхронізації перевіряються незалежно від транспортного шару та специфіки функцій.

Основними колекціями є: «objects» (поточний стан бізнес-об'єктів) та «changeLog» (журнал змін у розрізі полів). Документ у «objects» містить: унікальний ідентифікатор (id), актуальні значення властивостей, мітку часу останньої зміни (lastChangedAt), а також, за потреби, метадані (deviceIds, userId тощо). Колекція «changeLog» зберігає записи виду: objectId, property, value, valueType, changedAt та revision. Для ефективних вибірок необхідне комбіноване індексування принаймні за (objectId, property, changedAt desc), що забезпечує швидкий доступ до останніх значень по конкретних полях. Зауважимо, що «objects» репрезентує консолідований стан, тоді як «changeLog» — орієнтований на аудит і розв'язання конфліктів у часовому вимірі.

Клієнт надсилає на «/sync» батч описів об'єктів у форматі: { id, props: {k→v}, changedAt, clientId, lastSyncAt }. Сервер визначає ідемпотентність через поєднання (clientId, batchId або requestId), що дозволяє безпечно повторювати запит без побічних ефектів. Крім того, для кожної властивості об'єкта обчислюється ефективна «актуальність» за порівнянням changedAt із останнім відомим revision у «changeLog». Якщо воно новіше — зміна допускається до

застосування; інакше — відкидається (discard) як застаріла. Відкидання не є помилкою протоколу, а сигналізує клієнтові, що на сервері існує більш актуальна версія. У відповідь повертається набір дельт, які мають бути інтегровані на клієнті.

Політика «останній запис перемагає» реалізується шляхом порівняння часових позначок на рівні кожної властивості. Після валідації відносно «changeLog» сервер здійснює часткове оновлення документа в «objects». Успішне застосування зміни призводить до: (1) оновлення значення поля у «objects», (2) додавання/оновлення запису у «changeLog» із новим changedAt та збільшенням revision, (3) оновлення lastChangedAt об’єкта на поточний момент. Важливо, що в «changeLog» повинно залишатися лише останнє значення для кожної властивості, яке є домінуючим, тоді як застарілі записи для того самого поля можуть бути видалені або позначені як неактуальні, залежно від обраної політики зберігання.

Після оброблення батча сервер формує відповідь у вигляді списку об’єктів, чий стан новіший відносно lastSyncAt клієнта. Для кожного об’єкта повертається мінімально необхідний набір полів (props), що змінилися після вказаного моменту. Задля зменшення розміру відповіді може застосовуватися компресія (наприклад, GZip на рівні функцій), тоді як механізм пагінації обмежує кількість елементів у великих вибірках. Передбачено підтримку курсового пост-фільтрування (наприклад, за userId або logicalPartition), що дозволяє зменшити міжкористувацькі перетини при масових оновленнях.

Оскільки «changeLog» з часом зростає, необхідний механізм його періодичного «ущільнення». Компація може бути реалізована як планова Azure Function на таймері (TimerTrigger) з консервативними лімітами I/O. Основна ідея полягає у видаленні застарілих записів для кожної пари (objectId, property), залишаючи лише останню домінантну версію значення. За бажанням, замість твердого видалення може застосовуватись «soft delete» (позначення прапорцем), що спрощує аудит. Для великих колекцій доцільно обмежувати компакцію «горизонтальними» порціями (batch size), аби уникати пікових навантажень.

Оскільки серверна частина приймає дані з незахищених середовищ, застосовуються обов’язкові механізми аутентифікації та авторизації (наприклад, JWT). Для захисту від «несумісних» клієнтів перевіряється версія протоколу синхронізації та схема даних; у разі невідповідності повертаються семантичні коди помилок із пропозицією оновлення клієнта. Телеметрія за подіями «застосовано зміну», «відкинуто зміну», «віддано дельту» дозволяє ідентифікувати вузькі місця та некоректні часові моделі на пристроях.

Фрагмент реалізації серверної частини алгоритму синхронізації наведено у додатку 3.

ВИСНОВКИ

У цій роботі було розглянуто проблему синхронізації стану даних у мобільних offline-first застосунках із багатопристроєвим доступом та індивідуальною (per-user) моделлю даних. Аналіз сучасних підходів до синхронізації, теоретичних основ розподілених систем і наявних промислових рішень показав, що універсальні керовані платформи не завжди задовольняють вимоги до довгострокової стабільності, контрольованості й адаптивності архітектури. Особливої актуальності це набуло в контексті припинення підтримки Atlas Device Sync, що підтвердило необхідність незалежних і портативних алгоритмів синхронізації.

У роботі було обґрунтовано доцільність розроблення власного алгоритму синхронізації, оптимізованого під реальні сценарії використання мобільних застосунків. Ключовим архітектурним рішенням стало виділення двох базових сценаріїв: послідовного використання пристроїв та квазіпаралельних оновлень, що виникають унаслідок збоїв мережевого з’єднання. Такий підхід дозволив спроєктувати алгоритм, який зберігає простоту та ефективність у типовому випадку, водночас коректно обробляючи конкурентні зміни.

Основним внеском роботи є розширений алгоритм синхронізації на рівні окремих властивостей об’єкта, побудований на використанні журналу змін. Запропонована модель поєднує принципи LWW-узгодження з інкрементальною обробкою дельт, що дозволяє вирішувати конфлікти, уникнути перезапису неактуальних полів і зменшити обсяг переданих даних. Журнал змін зберігає лише останні релевантні значення властивостей, що забезпечує контрольований ріст метаданих і передбачувану складність алгоритму.

Практична реалізація алгоритму була продемонстрована для двох мобільних платформ — iOS та Android — із урахуванням обмежень конкретних технологій. Для iOS було показано, що подієва модель RealmSwift дозволяє автоматизувати фіксацію змін на рівні властивостей без ручного втручання в бізнес-логіку. Натомість для Android було доведено, що через обмеження Realm Kotlin необхідно застосовувати ручну реєстрацію змін у сеттерах моделей.

Попри відмінності реалізації, обидва підходи забезпечують однакову семантику дельта-синхронізації та повну сумісність із серверним алгоритмом.

Отримані результати підтверджують, що запропонований алгоритм є практичним компромісом між формальною коректністю та інженерною простотою. Він не потребує складних глобальних журналів операцій або спеціалізованих реплікованих структур даних, але водночас гарантує збереження намірів користувача й коректне злиття станів у разі збоїв синхронізації. Це робить алгоритм придатним для реальних мобільних продуктів із тривалим життєвим циклом.

До можливих напрямів подальших досліджень належать дослідження інтеграції алгоритму з потоковими механізмами серверної синхронізації та оптимізації формування дельт для великих наборів даних, також узагальнення запропонованого підходу для інших типів клієнтів і локальних сховищ, що дозволить оцінити його універсальність і адаптивність у ширшому контексті розподілених систем.

ПЕРЕЛІК ПОСИЛАНЬ

1. Gilbert S. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services / S. Gilbert, N. Lynch / ACM SIGACT News, 33(2), 2002. - p 51–59. DOI: 10.1145/564585.564601

2. Vogels W. Eventually Consistent / Communications of the ACM, 52(1), 2008. - p 40–44. DOI: 10.1145/1466443.1466448

3. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System / Communications of the ACM, 21(7), 1978. - p 558–565. DOI: 10.1145/359545.359563

4. Saito Y. Optimistic replication / ACM Computing Surveys, 37(1) / Y. Saito , M. Shapiro, 2005 p 42–81. URL: https://pages.cs.wisc.edu/~remzi/Classes/739/Fall2015/Papers/saito-optimistic-05.pdf

5. Conflict-Free Replicated Data Types / [Shapiro M., Preguiça N., Baquero C., Zawirski M.] / SSS, 2011. DOI: 10.1007/978-3-642-24550-3_29

6. DeCandia, G., et al. (2007). Dynamo: Amazon’s Highly Available Key-Value Store / SOSP ’07. DOI: 10.1145/1294261.1294281

7. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS / [Lloyd W., Freedman M., Kaminsky M., Andersen D.] / SOSP, 2011. - DOI: 10.1145/2043556.2043593

8. Almeida P. S.. Delta State Replicated Data Types / P. S. Almeida, A. Shoker, C. Baquero / Journal of Parallel and Distributed Computing № 111, 2018. - p 162–173. arXiv:1603.01529

9. Edwards, W. K., et al. Designing and implementing asynchronous collaborative applications with Bayou, 1997. DOI: 10.1145/263407.263530

10. Terry, D., et al. Managing Update Conflicts in Bayou / SOSP, 1995. - URL: https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p172-terry.pdf

11. Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors / Communications of the ACM, 13(7), 1970, p 422–426. DOI: 10.1145/362686.362692

12. Merkle, R. C. A Digital Signature Based on a Conventional Encryption Function / CRYPTO ’87, 1987. DOI: 10.1007/3-540-48184-2_32

13. Preguiça, N., et al. Dotted Version Vectors: Logical Clocks for Optimistic Replication, 2010. arXiv:1011.5808

14. Almeida, J. B., et al. Bounded Version Vectors / DAIS, 2004. DOI: 10.1007/978-3-540-30186-8_8

15. Baquero, C., et al. Array CRDTs Using Delta-Mutations / PACMPL, 2021. DOI: 10.1145/3447865.3457971

16. MongoDB Docs — Device Sync Deprecation (Sep 2024; EOL Sep 30, 2025). URL: https://www.mongodb.com/docs/atlas/app-services/sync/device-sync-deprecation/

17. MongoDB Community — Atlas Device Sync End of Life & Deprecation. URL: https://www.mongodb.com/community/forums/t/atlas-device-sync-end-of-life-and-deprecation/296687

18. MongoDB Community — Update to EOL and deprecation notice. URL: https://www.mongodb.com/community/forums/t/update-to-end-of-life-and-deprecation-notice/297168

19. MongoDB Manual — Change Streams. URL: https://www.mongodb.com/docs/manual/changeStreams/

20. MongoDB Resource — Introduction to Change Streams. URL: https://www.mongodb.com/resources/products/capabilities/an-introduction-to-change-streams

21. MongoDB Blog — Realm now part of Atlas platform. URL: https://www.mongodb.com/company/blog/product-release-announcements/realm-now-part-atlas-platform

22. Couchbase Docs — Sync Gateway. URL: https://docs.couchbase.com/sync-gateway/current/introduction.html

23. Couchbase Product — Sync Gateway. URL: https://www.couchbase.com/products/sync-gateway/

24. Firebase Docs — Firestore offline. URL: https://firebase.google.com/docs/firestore/manage-data/enable-offline

25. Firebase Docs — Firestore overview. URL: https://firebase.google.com/docs/firestore

26. AWS Blog — Amplify DataStore offline-first. URL: https://aws.amazon.com/blogs/mobile/building-offline-first-applications-with-aws-amplify-datastore-part-1/

27. AWS Docs — DataStore sync to cloud. URL: https://docs.amplify.aws/gen1/react/build-a-backend/more-features/datastore/sync-to-cloud/

28. ElectricSQL Docs — Intro. URL: https://electric-sql.com/docs/intro

29. Replicache Docs — How it works. URL: https://doc.replicache.dev/concepts/how-it-works

30. Replicache Docs — Server pull/push. URL: https://doc.replicache.dev/reference/server-pull

31. RxDB Docs — Overview & Replication. URL: https://rxdb.info/replication.html

32. RxDB — CouchDB Replication plugin. URL: https://rxdb.info/replication-couchdb.html

33. WatermelonDB — Sync implementation. URL: https://watermelondb.dev/docs/Implementation/SyncImpl

34. WatermelonDB — Sync Frontend. URL: https://watermelondb.dev/docs/Sync/Frontend

35. Ditto Docs — Sync concepts. URL: https://docs.ditto.live/key-concepts/syncing-data

36. Ditto — Edge SDK. URL: https://www.ditto.com/products/edge-sdk

37. PowerSync — Alternative to Atlas Device Sync. URL: https://www.powersync.com/blog/powersync-as-alternative-to-mongodb-atlas-device-sync

38. Supabase — MongoDB Realm/Device Sync alternatives. URL: https://supabase.com/blog/mongodb-realm-and-device-sync-alternatives

39. ObjectBox — Alternative to MongoDB Sync. URL: https://objectbox.io/alternative-to-mongodb-sync/

40. ObjectBox — Realm Device Sync deprecation. URL: https://objectbox.io/mongodb-realm-device-sync-deprecation/

ДОДАТОК 1. ОСНОВНІ КЛАСИ ДЛЯ РЕАЛІЗАЦІЇ ЖУРНАЛУ ЗМІН НА IOS ПЛАТФОРМІ

```
public class ModelUpdateEntry: Object {
    @Persisted public var id = ""
    @Persisted public var name = ""
    @Persisted public var type = ""
    @Persisted public var value = ""
    @Persisted var lastUpdate: Date = Date()

    public func getValue() -> Any {
        if type == "String" {
            return value
        } else if type == "Bool" {
            return (value == "true" || value == "1")
        } else if type == "Int" {
            return Int(value) ?? 0
        } else if type == "Date" {
            let milliseconds = Int((Double(value) ?? 0.0) * 1000)
            return milliseconds
        } else if type == "List" {
            let data = Data(value.utf8)
            do {
                let array = try JSONSerialization.jsonObject(with: data) as!
[String]
                return array
            } catch {
                print(error)
            }
        }
        return ""
    }
}
protocol VocPropertyChangeProtocol {
    var name: String { get }
    var newValue: Any? { get }
    var oldValue: Any? { get }
}

extension PropertyChange: VocPropertyChangeProtocol {
}
```

```
public struct VocPropertyChange: VocPropertyChangeProtocol {
    /**
     The name of the property which changed.
    */
    public let name: String

    /**
     Value of the property before the change occurred. This is not supplied if
     the change happened on the same thread as the notification and for `List`
     properties.

     For object properties this will give the object which was previously
     linked to, but that object will have its new values and not the values it
     had before the changes. This means that `previousValue` may be a deleted
     object, and you will need to check `isInvalidated` before accessing any
     of its properties.
    */
    public let oldValue: Any?

    /**
     The value of the property after the change occurred. This is not supplied
     for `List` properties and will always be nil.
    */
    public let newValue: Any?
}

public class BaseModel: Object {
    @Persisted(primaryKey: true) var id: ObjectId
    private var token: NotificationToken?
    public func getId() -> String {
        return id.stringValue
    }
    public override init() {
        super.init()
    }

    public func intProperties() -> Set<String> {
        return Set<String>()
    }
```

```
    override public func observeValue(forKeyPath keyPath: String?, of object:
Any?, change: [NSKeyValueChangeKey : Any]?, context: UnsafeMutableRawPointer?) {
        NSLog("Property updated: %@", keyPath ?? "")
    }

    func performForAllKeyPaths(_ action: (String) -> Void) {
        var count: UInt32 = 0
        guard let properties = class_copyPropertyList(object_getClass(self),
&count) else { return }
        defer { free(properties) }
        for i in 0 ..< Int(count) {
            let keyPath = String(cString: property_getName(properties[i]))
            action(keyPath)
        }
    }

    func registerChange(name: String, value: Any, oldValue: Any) {
        let realm = try! Realm()
        processChanges(change: VocPropertyChange(name: name, oldValue: oldValue,
newValue: value), realm: realm)
    }

    private func processChanges(change: VocPropertyChangeProtocol, realm:
Realm?) {
        guard let realm = realm else {
            return
        }
        if keyPathsForIgnoreUpdateRegistration().contains(change.name) {
            return
        }
        if let val = change.newValue as? Int,
intProperties().contains(change.name) {
            let updateEntry = ModelUpdateEntry()
            updateEntry.name = change.name
            updateEntry.value = "\(val)"
            updateEntry.type = "Int"
            updateEntry.id = getId()
            insertOrUpdate(entry: updateEntry, realm: realm)
        } else if let val = change.newValue as? Bool {
            let updateEntry = ModelUpdateEntry()
            updateEntry.name = change.name
```

```
            updateEntry.value = "\(val)"
            updateEntry.type = "Bool"
            updateEntry.id = getId()
            insertOrUpdate(entry: updateEntry, realm: realm)
        } else if let val = change.newValue as? String {
            let updateEntry = ModelUpdateEntry()
            updateEntry.name = change.name
            /*if change.name == "translation" {
                NSLog("Translation updated: %@", val)
            }*/
            updateEntry.value = "\(val)"
            updateEntry.type = "String"
            updateEntry.id = getId()
            insertOrUpdate(entry: updateEntry, realm: realm)
        } else if let val = change.newValue as? Date {
            let updateEntry = ModelUpdateEntry()
            updateEntry.name = change.name
            updateEntry.value = "\(val.timeIntervalSince1970)"
            updateEntry.type = "Date"
            updateEntry.id = getId()
            insertOrUpdate(entry: updateEntry, realm: realm)
        } else if let val = change.newValue as? LinkingObjects<WordsSet> {
            let updateEntry = ModelUpdateEntry()
            updateEntry.name = change.name
            updateEntry.value =
convertIntoJSONString(Array(val.map({$0.id.stringValue}))) ?? "[]"

            updateEntry.type = "List"
            updateEntry.id = getId()
            insertOrUpdate(entry: updateEntry, realm: realm)
        } else if let val = change.newValue as? Meaning {
            let oldVal = change.oldValue as? Meaning
            if (val.definition != oldVal?.definition) {
                let updateEntry = ModelUpdateEntry()
                updateEntry.name = "definition"
                updateEntry.value = val.definition
                updateEntry.type = "String"
                updateEntry.id = getId()
                insertOrUpdate(entry: updateEntry, realm: realm)
            }
            if (val.synonyms != oldVal?.synonyms)
```

```
            {
                let updateEntry = ModelUpdateEntry()
                updateEntry.name = "synonyms"
                updateEntry.value = convertIntoJSONString(Array(val.synonyms))
?? "[]"
                updateEntry.type = "List"
                updateEntry.id = getId()
                insertOrUpdate(entry: updateEntry, realm: realm)
            }
            if(val.examples != oldVal?.examples)
            {
                let updateEntry = ModelUpdateEntry()
                updateEntry.name = "examples"
                updateEntry.value = convertIntoJSONString(Array(val.examples))
?? "[]"
                updateEntry.type = "List"
                updateEntry.id = getId()
                insertOrUpdate(entry: updateEntry, realm: realm)
            }
        }
    }

    public func registerForUpdates() {
        if token != nil {
            return
        }
        performForAllKeyPaths { keyPath in
            addObserver(self, forKeyPath: keyPath, options: [], context: nil)
        }
        token = self.observe { [weak self] changes in

            let realm = try! Realm()
            if case .change(_, let changeList) = changes {
                for change in changeList {
                    self?.processChanges(change: change, realm: realm)
                }
            }
        }
    }

    public func keyPathsForIgnoreUpdateRegistration() -> Set<String> {
```

```
        return Set<String>()
    }

    func insertOrUpdate(entry: ModelUpdateEntry, realm: Realm) {
        let existingEntries = realm.objects(ModelUpdateEntry.self).where({$0.id
== entry.id && $0.name == entry.name})
        if let existingEntry = existingEntries.first {
            if realm.isInWriteTransaction {
                if !existingEntry.isInvalidated {
                    existingEntry.value = entry.value
                }
            } else {
                try! realm.write{
                    if !existingEntry.isInvalidated {
                        existingEntry.value = entry.value
                    }
                }
            }
        } else {
            if realm.isInWriteTransaction {
                if !entry.isInvalidated {
                    realm.add(entry)
                }
            } else {
                try! realm.write{
                    if !entry.isInvalidated {
                        realm.add(entry)
                    }
                }
            }
        }
    }

    func convertIntoJSONString(_ arrayObject: [String]) -> String? {
        do {
            let jsonData: Data = try JSONSerialization.data(withJSONObject:
arrayObject, options: [])
            if  let jsonString = NSString(data: jsonData, encoding:
String.Encoding.utf8.rawValue) {
                return jsonString as String
            }
```

```
        } catch let error as NSError {
            print("Array convertIntoJSON - \(error.description)")
        }
        return nil
    }

}
```

ДОДАТОК 2. ПРИКЛАД ДОМЕННИХ КЛАСІВ, ЩО РЕАЛІЗУЮТЬ ЖУРНАЛ ЗМІН НА ПЛАТФОРМІ ANDROID

```
class WordsSet(id:ObjectId = BsonObjectId()): RealmObject {
    constructor() : this(
        id = ObjectId()
    )
    var autoRegisterChanges = true
    @PrimaryKey
    var id: ObjectId
    init {
        this.id = id
    }
    public fun getId(): String {
        return id.toHexString()
    }

    public fun intProperties(): Set<String> {
        return HashSet<String>()
    }

    public fun getProperty(name: String): Any? {
        val prop = this::class.members.first { it.name == name } as
KProperty1<Any, *>
        if (prop != null) {
            return prop.get(this)
        }
        return null
    }

    fun registerChange(propertyName: String) {
        registerChanges(arrayOf(propertyName))
    }

    fun registerChanges(fields: Array<String>) {
        if (!autoRegisterChanges) {
            return
        }
        val objects = BaseModel.realm.query<WordsSet>("id == $0", id).find()
        val lastObject: WordsSet
```

```
if (objects.isEmpty()) {
    lastObject = this
} else {
    lastObject = objects.first()
}
fields.forEach {
    val value = lastObject.getProperty(it)
    if (value is Int) {
        val intVal = value as Int
        val entry = ModelUpdateEntry()
        entry.name = it
        entry.value = intVal.toString()
        entry.type = "Int"
        entry.id = getId()
        insertOrUpdate(entry)
    } else if (value is Boolean) {
        val boolVal = value as Boolean
        val entry = ModelUpdateEntry()
        entry.name = it
        entry.value = boolVal.toString()
        entry.type = "Bool"
        entry.id = getId()
        insertOrUpdate(entry)
    } else if (value is String) {
        val entry = ModelUpdateEntry()
        entry.name = it
        entry.value = value
        entry.type = "String"
        entry.id = getId()
        insertOrUpdate(entry)
    } else if (value is RealmInstant) {
        val timeVal = value as RealmInstant
        val entry = ModelUpdateEntry()
        entry.name = it
        entry.value = timeVal.epochSeconds.toString()
        entry.type = "Date"
        entry.id = getId()
        insertOrUpdate(entry)
    } else if (value is RealmList<*> && it == "sets") {
        val list = value as RealmList<*>
        val stringsList = list.map { it as? WordsSet }.mapNotNull {
```

```
it -> it?.id }.map { it -> it.asString().value.htmlEncode() }
                val str = stringsList.joinToString("'")
                val entry = ModelUpdateEntry()
                entry.name = it
                entry.value = str
                entry.type = "List"
                entry.id = getId()
                insertOrUpdate(entry)
            }
        }
    }

    private fun insertOrUpdate(entry: ModelUpdateEntry) {
        val oid = getId()
        val obj = BaseModel.realm.query(ModelUpdateEntry::class, "id == $0 &&
name == $1", oid, entry.name).first().find()
        if (obj == null) {
            BaseModel.realm.writeBlocking {
                copyToRealm(entry)
            }
        } else {
            BaseModel.realm.writeBlocking {
                findLatest(obj)?.value = entry.value
            }
        }
    }

    var name: String = ""
        set(value) {
            field = value
            registerChange("name")
        }
    @Index
    var lastUpdate: RealmInstant = RealmInstant.now()
    var lastManualTraining: RealmInstant = RealmInstant.now()
    var creationDate: RealmInstant = RealmInstant.now()
    var words: RealmList<Word> = realmListOf<Word>()
        set(value) {
            field = value
            registerChange("words")
        }
```

```
var subsets: RealmList<WordsSet> = realmListOf<WordsSet>()
    set(value) {
        field = value
        registerChange("subsets")
    }
var journey: LearningJourney? = null
var imageUrl: String = ""
    set(value) {
        field = value
        registerChange("imageUrl")
    }
var imageUrlSrc: String = ""
    set(value) {
        field = value
        registerChange("imageUrlSrc")
    }
var imageAuthor: String = ""
    set(value) {
        field = value
        registerChange("imageAuthor")
    }
@Index
var order = 0
    set(value) {
        field = value
        registerChange("order")
    }
@Index
var isDeleted = false
    set(value) {
        field = value
        registerChange("isDeleted")
    }
@Index
var isNew = true
var parent: WordsSet? = null
    set(value) {
        field = value
        registerChange("parent")
    }
var total = 0
```

```
        set(value) {
            field = value
            registerChange("total")
        }
    var learned = 0
        set(value) {
            field = value
            registerChange("learned")
        }
    var shared = false
        set(value) {
            field = value
            registerChange("shared")
        }
    var author: String? = null
        set(value) {
            field = value
            registerChange("author")
        }

    public fun updateStatistics(recursive: Boolean = false, background:
Boolean = false) {
        val realm = if (background) BaseModel.backgroundRealm!! else
BaseModel.realm
        var learnedWordsCount = 0
        var totalWordsCount = 0
        val sets: RealmResults<WordsSet> =
            realm.query<WordsSet>("parent.id == $0 && isDeleted == false",
id).find()
        for (subset in sets) {
            if (recursive) {
                subset.updateStatistics(recursive, background = background)
            }
        }
        val setsUpdated: RealmResults<WordsSet> =
            realm.query<WordsSet>("parent.id == $0 && isDeleted == false",
id).find()
        BaseModel.writeBlocking<MutableRealm>(background) {
            for (subset in setsUpdated) {
                learnedWordsCount += findLatest(subset)?.learned ?: 0
                totalWordsCount += findLatest(subset)?.total ?: 0
```

```
            }
        }
        BaseModel.writeBlocking<MutableRealm>(background) {
            learnedWordsCount += findLatest(this@WordsSet)?.words?.filter({
it.isLearned && !it.isDeleted})?.size ?: 0
            totalWordsCount += findLatest(this@WordsSet)?.words?.filter({
!it.isDeleted})?.size ?: 0
            val latest = findLatest(this@WordsSet)
            latest?.total = totalWordsCount
            latest?.learned = learnedWordsCount
        }
    }
    public fun json(updatesOnly: Boolean = false): JsonObject {
        val json = JsonObject()
        if (isNew || !updatesOnly) {
            json.addProperty("lid", id.toHexString())
            json.addProperty("jid", journey?.id?.toHexString() ?: "")
            json.addProperty("cd", creationDate.epochSeconds * 1000 +
creationDate.nanosecondsOfSecond / 1000000)
            json.addProperty("u", lastUpdate.epochSeconds * 1000 +
lastUpdate.nanosecondsOfSecond / 1000000)
            json.addProperty("mt", lastManualTraining.epochSeconds * 1000 +
lastManualTraining.nanosecondsOfSecond / 1000000)
            json.addProperty("d", isDeleted)
            json.addProperty("n", name)
            json.addProperty("i", imageUrl)
            json.addProperty("ius", imageUrlSrc)
            json.addProperty("ia", imageAuthor)
            json.addProperty("o", order)
            json.addProperty("ps", parent?.id?.toHexString())

        } else {
            val existingEntries: RealmResults<ModelUpdateEntry> =
                BaseModel.realm.query<ModelUpdateEntry>("id == $0",
getId()).find()
            var hasValue = false
            existingEntries.forEach({
                val key = nameMap.get(it.name)
                if (key != null) {
                    val boolValue = it.getValue() as? Boolean
                    if (boolValue != null) {
```

```
                        json.addProperty(key, boolValue)
                        hasValue = true
                    }
                    val numberValue = it.getValue() as? Int
                    if (numberValue != null) {
                        json.addProperty(key, numberValue)
                        hasValue = true
                    }
                    val longValue = it.getValue() as? Long
                    if (longValue != null) {
                        json.addProperty(key, longValue)
                        hasValue = true
                    }
                    val stringValue = it.getValue() as? String
                    if (stringValue != null) {
                        json.addProperty(key, stringValue)
                        hasValue = true
                    }
                    val listValue = it as? ArrayList<String>
                    if (listValue != null) {
                        val list = JsonArray()
                        for (str in listValue) {
                            list.add(str)
                        }
                        json.add(key, list)
                        hasValue = true
                    }
                }
            })
            if (hasValue) {
                json.addProperty("lid", id.toHexString())
                if (!json.has("u")) {
                    json.addProperty("u", lastUpdate.epochSeconds * 1000 +
lastUpdate.nanosecondsOfSecond / 1000000)
                }
            }
        }
        return json
    }
    companion object {
        val nameMap = hashMapOf(
```

```
            "id" to "lid",
            "name" to "n",
            "lastUpdate" to "u",
            ...
        )
        fun update(json: JsonObject, background: Boolean = false) {
            val id = if (json.get("lid").isJsonNull) null else
json.get("lid")?.asJsonPrimitive?.asString
            if (id?.startsWith("BsonObjectId") ?: false) {
                return
            }
            if (id == null) {
                return
            }
            var objId = BsonObjectId(id)
            if (objId == null) return
            val jid = if (json.get("jid").isJsonNull) null else
json.get("jid")?.asJsonPrimitive?.asString
            if (jid == null) {
                return
            }
            if (jid?.startsWith("BsonObjectId") ?: false) {
                return
            }
            var jId = BsonObjectId(jid)

            if (jId == null) {
                return
            }
            val realm = if (background) BaseModel.backgroundRealm!! else
BaseModel.realm

            BaseModel.writeBlocking<MutableRealm>(background = background) {
                val sets: RealmResults<WordsSet> =
                    realm.query<WordsSet>("id == $0", objId).find()
                val journeys: RealmResults<LearningJourney> =
                    realm.query<LearningJourney>("id == $0", jId).find()
                if (journeys.isEmpty()) return@writeBlocking
                val journey = journeys.first()
                var set: WordsSet
                val needLatest: Boolean
```

```
                if (sets.isEmpty()) {
                    set = copyToRealm(WordsSet(objId))
                    set.isNew = false
                    needLatest = true
                } else {
                    set = sets.first()
                    set = findLatest(set) ?: set
                    needLatest = true
                }
                set.autoRegisterChanges = false
                set.journey = if (needLatest) findLatest(journey) else
journey
                val name = json.get("n")
                if (name != null) {
                    set.name = name.asString
                }
                val update = json.get("u")
                if (update != null) {
                    set.lastUpdate = RealmInstant.from(update.asLong / 1000,
(update.asLong % 1000).toInt() * 1000)
                }
// similar code for other properties
                set.autoRegisterChanges = true
            }
        }
    }
}
```

ДОДАТОК 3. ФРАГМЕНТ РЕАЛІЗАЦІЇ СЕРВЕРНОЇ ЧАСТИНИ АЛГОРИТМУ СИНХРОНІЗАЦІЇ

```
[Function("Sync")]
public async Task<HttpResponseData> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = "sync")]
HttpRequestData req,
    FunctionContext ctx)
{
    var logger = ctx.GetLogger("Sync");
    var body = await new StreamReader(req.Body).ReadToEndAsync();
    var batch = JsonSerializer.Deserialize<SyncBatch>(body, new
JsonSerializerOptions { PropertyNameCaseInsensitive = true });
    var result = await _syncService.ApplyBatchAsync(batch);
    var res = req.CreateResponse(HttpStatusCode.OK);
    await res.WriteAsJsonAsync(result);
    return res;
}

public async Task<SyncResult> ApplyBatchAsync(SyncBatch batch)
{
    var updated = new List<DeltaObject>();
    foreach (var item in batch.Objects)
    {
        var objId = item.Id;
        foreach (var kv in item.Props)
        {
            var prop = kv.Key;
            var proposed = kv.Value.Value;
            var pType = kv.Value.Type;
            var changedAt = item.ChangedAt;

            var newest = await _changeLog.GetNewestAsync(objId, prop);
            if (newest == null || changedAt > newest.ChangedAt)
            {
                await _objects.PartialSetAsync(objId, prop, proposed, pType);
                await _changeLog.UpsertAsync(objId, prop, proposed, pType,
changedAt);
            }
        }
        // для клієнта з lastSyncAt < lastChangedAt повертаємо дельти
```

```
        var delta = await _objects.GetDeltaSinceAsync(objId,
batch.LastSyncAt);
        if (delta != null) updated.Add(delta);
    }
    return new SyncResult { Updated = updated };
}
public async Task PartialSetAsync(string id, string prop, BsonValue value,
string type)
{
    var filter = Builders<BsonDocument>.Filter.Eq("_id", id);
    var update = Builders<BsonDocument>.Update
        .Set(prop, value)
        .Set("lastChangedAt",
DateTimeOffset.UtcNow.ToUnixTimeMilliseconds());
    await _objects.UpdateOneAsync(filter, update, new UpdateOptions {
IsUpsert = true });
}

public async Task UpsertAsync(string id, string prop, BsonValue value, string
type, long changedAt)
{
    var filter = Builders<BsonDocument>.Filter.And(
        Builders<BsonDocument>.Filter.Eq("objectId", id),
        Builders<BsonDocument>.Filter.Eq("property", prop)
    );
    var update = Builders<BsonDocument>.Update
        .Set("value", value)
        .Set("valueType", type)
        .Set("changedAt", changedAt);
    await _changeLog.UpdateOneAsync(filter, update, new UpdateOptions {
IsUpsert = true });
}
```