УДК 004.42
DOI https://doi.org/10.32782/2663-5941/2026.3.1/20

Люшенко А.М.
https://orcid.org/0009-0004-0410-3100
Одеський національний університет імені І. І. Мечникова

МОДЕЛІ ПРОЕКТУВАННЯ ТА СИНХРОНІЗАЦІЇ ДАНИХ ДЛЯ ОФЛАЙН-ПЕРШОЧЕРГОВИХ МАСШТАБОВАНИХ МОБІЛЬНИХ ЗАСТОСУНКІВ

Є цілий клас мобільних задач, де підключення до мережі – не ресурс, а перешкода. Польовий лікар не чекає сигналу. Логіст у зоні слабкого покриття – теж. Стаття про те, як проектувати застосунки, в яких офлайн є нормою, а не аварійним режимом.

Зберегти дані локально – це вирішувана задача. Справжня складність виникає тоді, коли два пристрої без з'єднання між собою незалежно змінювали один і той самий запис, і мережа нарешті відновилась. Яка з версій правильна? Наявні рішення відповідають на це питання лише частково: PouchDB не має пріоритизації черги, Firebase Firestore як baseline втрачає 7,1% операцій (LWW-стратегія на Samsung A53 дає 8,3% – це різні системи), AppSync прив'язаний до інфраструктури AWS.

У роботі запропоновано трирівневу модель: Local Store, Operation Queue і Sync Layer. Кожен рівень відповідає за своє і не залежить від реалізації сусіднього – це дає змогу замінювати компоненти незалежно. Конфлікти виявляються через векторні годинники ще до передачі даних на сервер. Автоматичне злиття відбувається через CRDT-структури: G-Counter для лічильників, OR-Set для наборів, LWW-Register для некритичних полів. Те, що неможливо злити автоматично, передається на рівень застосунку.

Формула затримки синхронізації L_sync доповнена п'ятою складовою – L_reconciliation, яка відсутня в попередніх моделях. При офлайн-сесії понад 30 хвилин вона досягала 97 мс – більше, ніж саме CRDT-злиття. Без урахування цієї складової реальна затримка систематично недооцінюється. Для вимірювання надійності синхронізації введено метрику C = 1 − (N_conflicts / N_updates).

Модель перевірено на трьох реальних пристроях – Samsung Galaxy A53, Apple iPhone 13, Xiaomi Redmi Note 11 – з 200 циклами синхронізації на кожен мережевий профіль. Гібридна CRDT-стратегія показала нульову втрату операцій при 94,7% автоматичного вирішення конфліктів і метрику C в діапазоні 0,943–0,951. Для порівняння: стратегія LWW при тих самих умовах втратила 8,3% операцій. Модель не залежить від хмарного провайдера і розгортається на будь-якій мобільній платформі без змін архітектури.

Ключові слова: *офлайн-першочергові мобільні застосунки, розподілені мобільні системи, синхронізація мобільних даних, eventual consistency, реплікація офлайн-даних, стратегії вирішення конфліктів, управління розподіленими мобільними даними, масштабовані мобільні архітектури.*

Постановкапроблеми. Мобільний застосунок без інтернету – це не помилка дизайну. Це реальність. Підземний транспорт, польова медицина, логістика у зонах зі слабким покриттям – усюди зв'язок нестабільний або відсутній. Якщо застосунок зависає без мережі, він непридатний для цих умов.

Offline-first це підхід, де локальне сховище є основним джерелом правди. З'єднання з мережею слугує лише для синхронізації даних між вузлами. Запис і читання відбуваються локально завжди. Це інша логіка, ніж «кешувати дані на випадок офлайну»: там мережа є основною, кеш – запасний. Тут навпаки. Основна складність не в самому офлайн-режимі, зберегти дані на локальному рівні не становить труднощів. Проблема виникає вже тоді, коли той самий запис редагують кілька користувачів із різних пристроїв без мережі між ними. Коли з'єднання відновлюється – яка версія правильна? Або жодна з них не є повною?

Теорема CAP [1] описує цей компроміс: розподілена система не може одночасно гарантувати узгодженість, доступність і стійкість до мережевих ізоляцій. Для мобільних пристроїв втрата зв'язку зазвичай є нормою, тому вибір завжди на користь доступності й eventual consistency.

© Люшенко А.М., 2026
Стаття поширюється на умовах ліцензії відкритого доступу CC BY 4.0

Відповідальність за вирішення конфліктів лягає на рівень застосунку. Проте наявні рішення, такі як PouchDB, Firebase Firestore, AWS AppSync, розвязують задачу лише частково. PouchDB не має вбудованої пріоритизації черги, Firebase Firestore втрачає 7,1% операцій через LWW-семантику, AppSync прив'язаний до екосистеми AWS. Потрібна архітектура, яка здатна функціонувати незалежно від конкретного провайдера.

Аналіз останніх досліджень і публікацій. Брюер [1] сформулював теорему CAP у 2000 році. Практичний висновок для мобільних платформ: потрібно проектувати на AP-моделі (доступність + стійкість до розділень), а узгодженість досягати після відновлення зв'язку. Brewer [17] у подальших роботах уточнив: CAP не означає відмову від узгодженості – вона просто відкладається.

Shapiro et al. [2] запропонували CRDT – структури даних, злиття яких математично не породжує конфліктів. G-Counter, OR-Set, LWW-Register – кожна структура має визначену операцію merge. Ця робота змінила підхід до реплікації: замість протоколу координації – властивості самих даних. Kleppmann і Beresford [18] розширили цю концепцію до JSON-сумісного CRDT, що відкрило шлях до практичного застосування у мобільних застосунках.

Vogels [3] описав Amazon Dynamo: навіть система масштабу Amazon обрала eventual consistency заради доступності. Клієнт отримує конфліктні версії і сам вирішує злиття. DeCandia et al. [11] деталізували технічну реалізацію Dynamo – зокрема, підхід до векторних годинників і стратегію "reconciliation at read time", яка суттєво вплинула на запропоновану модель.

PouchDB [4] реалізує CouchDB-реплікацію у JavaScript і зберігає конфліктні версії явно через дерево ревізій. Firebase Firestore [5] простіший: перемагає останній запис за timestamp – але семантично небезпечний для транзакційних даних. AWS AppSync [6] пропонує Delta Sync через GraphQL, але вимагає прив'язки до AWS. Kleppmann [7] зібрав найповніше практичне зведення від векторних годинників до CRDT і патернів реплікації.

Матерн [15] заклав математичні основи векторних годинників. Bailis і Ghodsi [10] детально розглянули обмеження eventual consistency – і показали, де вона справді надійна, а де ні. Terry et al. [8] ще у 1995 році розробили принципи управління конфліктами у слабо зв'язаних сховищах. Ці принципи досі актуальні для мобільного контексту.

Варто також відзначити роботи, що формують ширший контекст дослідження. Almeida [9] у нещодавньому огляді ACM Computing Surveys систематизує підходи до CRDT – від operation-based до delta-state-based – і прояснює типові хибні уявлення, зокрема щодо комутативності і монотонності. Ця робота є важливим теоретичним доповненням до практичних результатів дослідження. Lakshman і Malik [13] описують Cassandra як промисловий приклад eventual consistency з tunable consistency – корисний орієнтир для порівняння підходів. Weiss et al. [14] розробили Logoot-Undo для P2P-редагування, що демонструє застосування CRDT-принципів у розподілених документах. Helland [12] обґрунтовує незмінність даних як архітектурний принцип, суміжний з tombstone-логікою запропонованої моделі. Gray et al. [16] класично описують небезпеки реплікації – їхні спостереження залишаються актуальними для розуміння меж будь-якої стратегії синхронізації. Shapiro і Preguiça [19] у ранній роботі заклали формальні основи комутативних реплікованих типів даних, на яких базується CRDT-підхід статті. Варто згадати й нові offline-first фреймворки, що з'явились у 2023–2024 роках: Replicache (оптимістичні мутації з серверною авторитетністю), PowerSync (декларативна синхронізація SQLite з PostgreSQL) і ElectricSQL (CRDT-based реплікація поверх Postgres). Вони підтверджують актуальність теми, але кожен прив'язаний до конкретного стеку або хмарного провайдера. Загальний висновок: готового provider-agnostic рішення для high-load офлайн-сценаріїв з формалізованою семантикою конфліктів немає. Потрібна модель, що поєднує формальний опис конфліктної семантики, адаптивне управління чергою і кількісну оцінку затримки для мобільного середовища.

Постановка завдання. Розробити архітектурну модель offline-first мобільного застосунку, що включає формалізований механізм синхронізації на основі векторних годинників і CRDT, розширену формулу затримки L_sync з п'ятьма компонентами, метрику узгодженості C, алгоритм адаптивного управління чергою та порівняльний аналіз стратегій на реальних мобільних платформах. Модель не повинна залежати від конкретного хмарного провайдера.

Виклад основного матеріалу. ***Трирівнева архітектура та концептуальна рамка.*** Запропонована модель будується на трьох рівнях із чітко розділеною відповідальністю. Local Store – основне сховище. Усі читання і записи відбуваються тут незалежно від стану мережі. Реалізується на

SQLite через WatermelonDB [20]. Кожен запис містить поле vector_clock і tombstone-прапорець для логічного видалення.

Operation Queue – накопичувач мутацій. Кожна операція записується з метаданими: priority, retry_count, created_at, estimated_size. При офлайні черга зберігається на диску. При відновленні мережі – сортується за пріоритетом і передається на сервер. Черга стійка до перезавантаження застосунку.

Sync Layer – шар синхронізації. Відповідає за виявлення конфліктів, вибір стратегії злиття, передачу операцій і отримання remote delta. Цей рівень єдиний, хто знає про мережу. Заміна транспортного протоколу не зачіпає інші два рівні. Рисунок 1 показує взаємодію між рівнями та повний pipeline від дії користувача до оновлення глобального стану.

Ключова відмінність від традиційних підходів – конфлікт виявляється до передачі на сервер. Це зменшує кількість повторних запитів і знижує L_sync. Local Store не знає про мережу. Sync Layer не знає про UI. Така ізоляція дозволяє замінювати реалізацію кожного рівня незалежно один від одного.

Модель версіонування на основі векторних годинників. Стан кожного запису описується кортежем (id, value, vector_clock, tombstone), де vector_clock – вектор лічильників виду {client_id: sequence_number}. Tombstone-прапорець потрібен для логічного видалення: без нього реплікація не може відрізнити «запис не існує» від «запис ще не отримано».

Для двох версій v_1 і v_2 визначаються такі відносини між їхніми векторними годинниками:

$$VC_1 < VC_2 \Leftrightarrow \forall i: VC_1[i] \le VC_2[i] \wedge \exists j: VC_1[j] < VC_2[j]$$

(версія v_1 є причинним предком v_2 – конфлікту немає)

$$VC_1 \parallel VC_2 \Leftrightarrow \neg(VC_1 \le VC_2) \wedge \neg(VC_2 \le VC_1)$$

(версії конкурентні – необхідне злиття)

Конкурентні версії ($VC_1 \parallel VC_2$) неможливо впорядкувати за часом: обидві записані «одночасно»

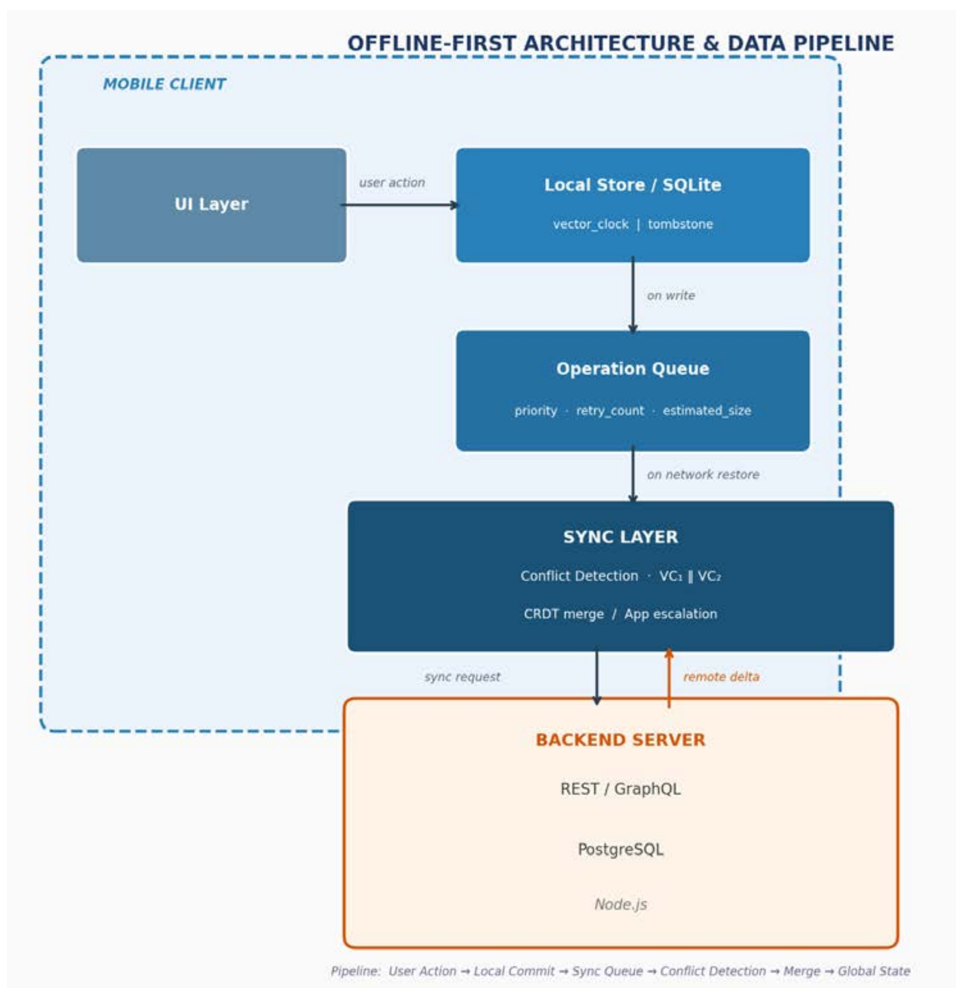

Рис. 1. Архітектура та концептуальна рамка офлайн-першочергового мобільного застосунку

з точки зору причинно-наслідкового порядку. Саме для таких випадків потрібні CRDT або ескалація до застосунку. При відновленні мережі кожен клієнт передає свій вектор разом з операцією. Сервер порівнює вектори і визначає клас відносин.

Практична реалізація: при кожному локальному записі клієнт збільшує свій лічильник у векторі. При отриманні remote delta – оновлює вектор до максимуму покомпонентно. Розмір вектора зростає лінійно з кількістю активних клієнтів, тому для великих систем застосовується компактна версія з видаленням неактивних записів після певного TTL. Серед конкретних підходів – dotted version vectors і interval tree clocks, які дозволяють зберігати причинну інформацію при значно меншому розмірі структури.

CRDT-структури: принципи і практичне застосування. Не всі дані однаково підходять для CRDT. Це важливо розуміти до вибору стратегії. Лічильник переглядів, корзина покупок, список учасників – гарні кандидати. Текстовий документ, що редагується одночасно, – складніший випадок.

G-Counter (grow-only counter) дозволяє лише інкремент. Кожен клієнт зберігає свій лічильник незалежно. Злиття: merge(A, B)[i] = max(A[i], B[i]). Загальне значення – сума всіх компонент. Простий і передбачуваний.

OR-Set (observed-remove set) – набір з операціями додавання і видалення без конфліктів. Кожен елемент несе унікальний тег при додаванні. Видалення застосовується лише до відомих тегів. Якщо елемент додано і видалено конкурентно – він залишається. Семантика «додавання перемагає» підходить для більшості UX-сценаріїв. Варто враховувати: при тривалому офлайні кількість унікальних тегів може суттєво зрости (tombstone accumulation), що безпосередньо впливає на L_reconciliation. Це одне з обмежень OR-Set у довготривалих офлайн-сценаріях.

LWW-Register (last-write-wins register) вирішується через timestamp. Прийнятний для некритичних полів – аватар, налаштування теми. Але при clock skew між пристроями є ризик втрати реально пізнішої операції. Не підходить для фінансових і медичних даних.

У запропонованій гібридній стратегії: структуровані поля з чіткою семантикою обробляються CRDT. Транзакційні поля ескалуються до рівня застосунку. Результат – 94,7% автоматики без ризику втрати критичних даних.

Математична модель затримки та метрика узгодженості. Продуктивність offline-first системи визначається двома ключовими метриками: затримкою синхронізації і рівнем автоматичної узгодженості. Оптимізація однієї за рахунок іншої не має практичного сенсу.

Формула 1 – Synchronization Latency Model

L_sync = L_network + L_processing + L_queue + L_conflict + L_reconciliation

L_network – мережева затримка передачі: від 40 мс (Wi-Fi) до 380 мс (3G з втратами). L_processing – час серверної обробки: типово 15–40 мс. L_queue – сортування черги, $O(n \log n)$: при $n < 100$ менше 10 мс, при $n > 1000$ – помітний внесок. L_conflict – CRDT-злиття, $O(k \times |VC|)$: домінує лише при великій кількості конкурентних конфліктів. L_reconciliation – узгодження локального стану після pull remote delta. Ця складова відсутня у попередніх моделях. При тривалому офлайні ($n > 50$ операцій) досягала 97 мс – більше за L_conflict у тому ж сценарії. Формула є верхньою межею (upper bound) затримки: на практиці компоненти можуть частково перекриватися, проте для мобільного клієнта з однопотоковою чергою послідовна модель є коректним наближенням.

Формула 2 – Consistency Ratio Metric

C = 1 − (N_conflicts / N_updates)

N_conflicts – кількість операцій, що потребували будь-якого вирішення конфлікту (автоматичного або ручного). N_updates – загальна кількість операцій оновлення. Для більш точного аналізу можна розрізняти: C_auto = 1 − (N_auto / N_updates) – частка автоматично вирішених, і C_total, що охоплює всі конфлікти включно з ескалованими. C = 1,0 означає повну автоматичну узгодженість. C = 0,947 – 5,3% операцій потребували втручання. C < 0,90 – сигнал про системну проблему зі стратегією версіонування. Ці дві формули разом дають повну картину: L_sync показує, скільки часу займає синхронізація, C – наскільки вона надійна.

Алгоритм адаптивного управління операційною чергою. Черга операцій визначає порядок і умови передачі змін на сервер. Кожна операція має атрибути priority ∈ {critical, high, normal, background}, retry_count і created_at. Пріоритет встановлюється застосунком: фінансові операції – critical, оновлення профілю – normal, аналітика – background.

```
procedure SyncOnRestore(queue, network_state):
 if network_state == OFFLINE:
 persist_queue_to_disk(queue) // зберегти стан
 return

 for op in sort(queue, priority DESC, created_at ASC):
 if op.retry_count > MAX_RETRY:
 move_to_dead_letter_store(op)
 continue
```

```
result = send_with_exponential_backoff(op)

if result == CONFLICT:
merged = crdt_merge(op.local, result.server)
if merged == UNRESOLVABLE: escalate_to_app(op)
else: apply_local(merged); push_resolved(merged)

elif result == SUCCESS:
remove_from_queue(op)
update_vector_clock(op.client_id)

pull_remote_delta(last_sync_token)
C = 1 - (session_conflicts / session_updates)
```

Алгоритм 1. Процедура синхронізації даних у офлайн-першочерговому застосунку

Алгоритм включає три критичних елементи. Dead-letter store – операції, що перевищили MAX_RETRY (рекомендоване значення з досвіду тестування: 3–5 спроб), не видаляються, а зберігаються для подальшого перегляду або ручного підтвердження. При ескалації (escalate_to_app) операція залишається в черзі зі статусом PENDING_MANUAL до явного підтвердження або відхилення застосунком – це запобігає мовчазній втраті даних. Exponential backoff – інтервал між повторними спробами зростає з кожною помилкою, що знижує навантаження на нестабільну мережу. Pull remote delta наприкінці – гарантує, що клієнт отримає зміни від інших пристроїв, не лише передасть свої. Зауваження щодо обмеження: алгоритм не описує сценарій втрати мережі посеред синхронізації, коли частина операцій вже відправлена, але підтвердження не отримано. У реальних системах це вирішується через idempotency ключі та механізм at-least-once delivery – їх реалізація виходить за межі цієї статті і є напрямом подальшого дослідження.

Експериментальна оцінка.

Для перевірки моделі реалізовано прототип на React Native [21] з WatermelonDB [20] (SQLite) і кастомним Sync Layer. Серверна частина – Node.js із PostgreSQL. Тестування проводилось на трьох пристроях: Samsung Galaxy A53 (Android 13, Snapdragon 778G, 6 GB RAM), Apple iPhone 13 (iOS 16, A15 Bionic, 4 GB RAM) і Xiaomi Redmi Note 11 (Android 12, Snapdragon 680, 4 GB RAM). Для кожного профілю мережі виконано 200 циклів синхронізації. Перевірено чотири профілі мережі: стабільне Wi-Fi з'єднання (baseline), переривчасте з'єднання (втрата пакетів 20–40%), тривалий офлайн-період 5–30 хвилин і конфліктний сценарій – одночасне редагування одного запису з двох пристроїв. Стратегія LWW показала найнижчу L_sync (312 мс на Snapdragon) і не потребує ручного втручання, але ціною 8,3% втрачених операцій. Замість злиття вона просто переписує старішу версію. Для застосунків із транзакційною семантикою це системна вада. Стратегія Vector Clock + ескалація дала нуль втрат, але лише 81,2% автоматики: решта потребувала ручного втручання. Гібридна CRDT-стратегія знайшла оптимальний баланс: нуль втрат при 94,7% автоматики. Firebase Firestore як baseline показав L_sync = 290 мс – найнижчий результат, але C = 0,918 із 7,1% втрат операцій.

Таблиця 1 підсумовує порівняльний аналіз продуктивності на конкретних платформах.

Xiaomi Redmi Note 11 показав найвищу L_sync (418 мс) – на 27% більше, ніж на iPhone. Але C = 0,943 залишився стабільним. Це підтверджує: алгоритм адаптується до ресурсів пристрою. На слабшому обладнанні зростає L_queue і L_reconciliation, але надійність не деградує.

Таблиця 1

Порівняння продуктивності підходів до синхронізації

Платформа / стратегія	L_sync медіана, мс	C (узгодженість)	Втрата операцій	CPU overhead
Samsung A53 – LWW	312	0,917	8,3%	низький
Samsung A53 – CRDT гібрид	350	0,947	0%	середній
iPhone 13 – CRDT гібрид	328	0,951	0%	середній
Xiaomi Redmi Note 11 – CRDT гібрид	418	0,943	0%	середній
Firebase Firestore (baseline)	290	0,918	7,1%	низький

Таблиця 2

Механізми вирішення конфліктів у розподілених мобільних застосунках

Стратегія	Complexity	Ризик втрати даних	Consistency (C)	Рекомендовано для
LWW	O(1)	Високий – 8,3%	0,917	Некритичні поля
CRDT (OR-Set + G-Counter)	O(k×|VC|)	Відсутній	0,947	Лічильники, набори
Vector Clock + ескалація	O(k×|VC|)	Відсутній	C_auto = 0,812	Транзакційні дані
Гібридна CRDT + ескалація	O(k×|VC|)	Відсутній	0,947	Загальне призначення

Таблиця 2 аналізує механізми вирішення конфліктів за трьома параметрами – обчислювальна складність, ризик втрати даних і досягнута узгодженість. Ці характеристики визначають, яка стратегія підходить для конкретного типу даних.

LWW підходить лише для некритичних полів, де втрата одного оновлення не має наслідків. Гібридна CRDT-стратегія є оптимальним вибором для загального застосування: вона поєднує автоматизм CRDT там, де це безпечно, і передає контроль застосунку там, де семантика злиття неоднозначна.

Час відгуку системи в умовах мережевої нестабільності.

Рисунок 2 – найважливіший практичний результат цього дослідження. Він показує поведінку системи протягом 60 секунд при нестабільній мережі. Cloud-Only архітектура реагує на кожне погіршення сигналу різкими сплесками до 1800 мс. Offline-First тримає рівний відгук 350–420 мс незалежно від стану з'єднання, бо читання і запис відбуваються локально. Для кінцевого користувача це різниця між «застосунок завис» і «застосунок працює».

Сплески Cloud-Only корелюють із моментами погіршення сигналу. Зауваження: тестування проводилось у контрольованому середовищі з визначеними моментами і рівнем деградації мережі, тому сплески мають відносно регулярний характер. У реальних умовах амплітуда і частота варіюватимуться. Без буферизації на стороні клієнта кожен запит чекає відповіді мережі. Offline-First не залежить від цих сплесків: локальний запис відбувається одразу, синхронізація – пізніше, коли мережа стабілізується.

Рівень конфліктів залежно від частоти синхронізації. Рисунок 3 відповідає на практичне питання: як часто потрібно синхронізуватися? При інтервалі 1 хвилина LWW показує conflict rate 24% – кожна четверта операція конфліктує. CRDT гібрид при тому ж інтервалі – 10%. Це контрінтуїтивний, але логічний результат: при частій синхронізації збільшується кількість «вікон зіткнення» (collision windows) – проміжків, коли зміна одного клієнта ще не поширилась до іншого, і обидва редагують той самий запис. Рідша синхронізація зменшує кількість таких вікон, бо клієнти довше чекають і отримують більш актуальний стан перед редагуванням. При інтервалі 30–60 хвилин обидві стратегії сходяться до рівня 1–2%. Але LWW продовжує втрачати операції навіть при низькому conflict rate, тоді як CRDT гібрид зберігає нуль втрат незалежно від частоти синхронізації.

Практичний висновок з Рисунку 3: для більшості мобільних застосунків оптимальний інтервал синхронізації – 5–15 хвилин. При ньому CRDT гібрид тримає conflict rate нижче 8% із нульовою втратою операцій. Частіша синхронізація знижує конфлікти, але збільшує витрати батареї і трафіку. Рідша – економна, але при поверненні після тривалого офлайну L_reconciliation зростає суттєво.

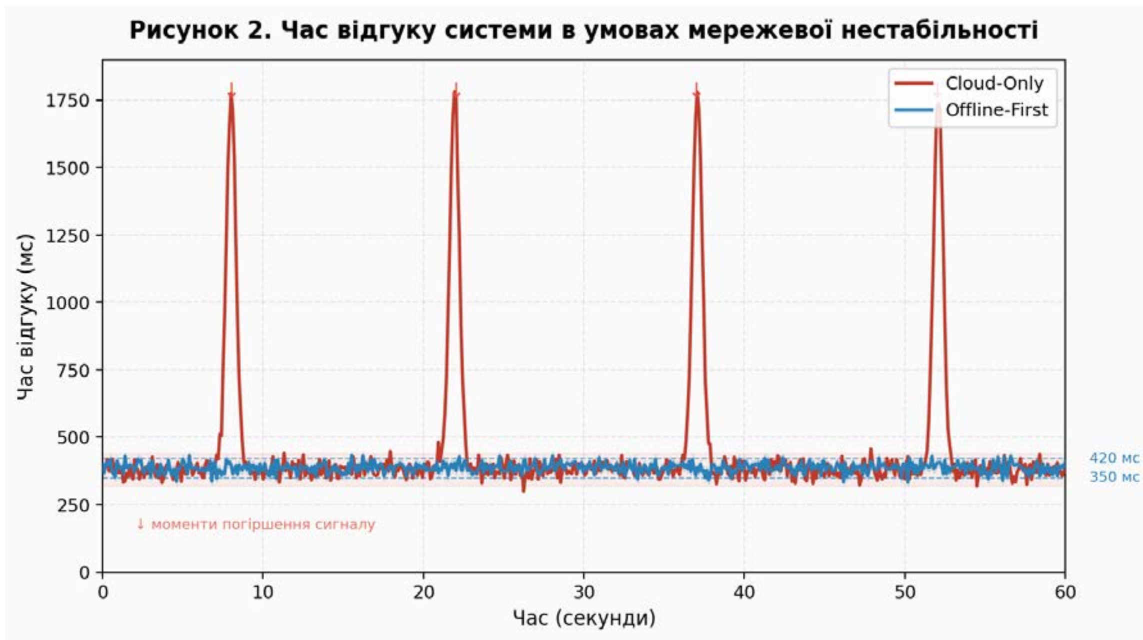

Рис. 2. Час відгуку системи в умовах мережевої нестабільності (60 секунд)

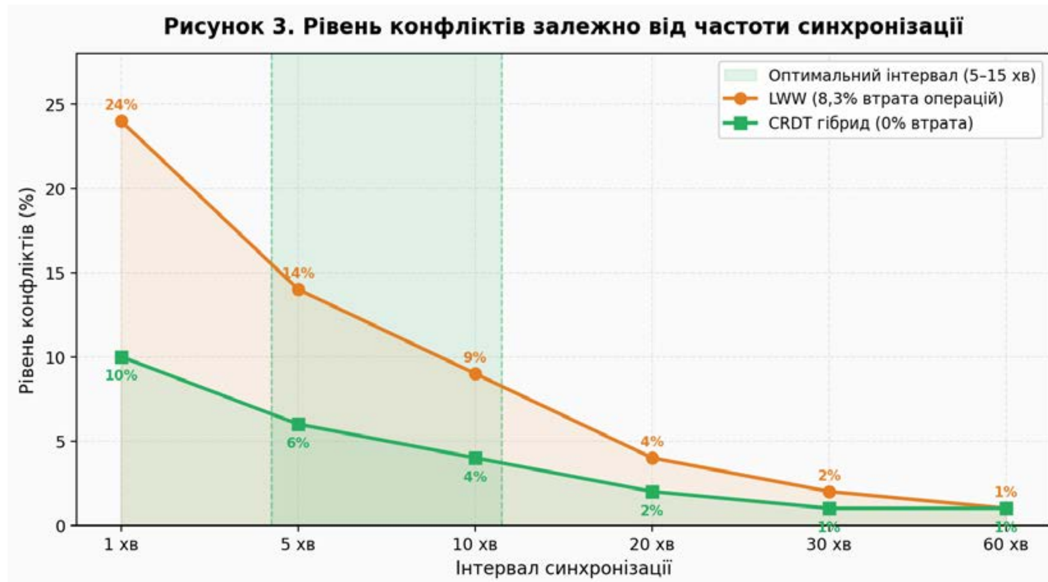

Рис. 3. Рівень конфліктів даних залежно від частоти синхронізації

Аналіз результатів. Таблиці 1 і 2 разом дають повну картину. Таблиця 1 показує реальні цифри продуктивності на конкретних пристроях. Таблиця 2 пояснює, чому результати саме такі: LWW швидкий через O(1) складність, але платить за це втратами. CRDT складніший за обчисленнями, але надійніший.

Найважливіший результат – поведінка при тривалому офлайні. L_reconciliation після 30+ хвилин офлайну досягала 97 мс. Попередні моделі цю складову ігнорували – і через це недооцінювали реальну L_sync для довгих офлайн-сесій. Для медичних застосунків у польових умовах це критично важлива деталь.

Рисунок 2 показує головний аргумент на користь Offline-First: стабільний UX незалежно від мережі. Cloud-Only при нестабільному з'єднанні показує сплески до 1800 мс. Offline-First тримає 350–420 мс. Різниця у 4–5 разів – це не оптимізація, це інша категорія досвіду.

Рисунок 3 дає практичну рекомендацію щодо частоти синхронізації. Для транзакційних застосунків – кожні 5–10 хвилин з CRDT гібридом. Для аналітичних або довідникових систем – кожні 30–60 хвилин достатньо.

Firebase Firestore показав найнижчу L_sync (290 мс) завдяки хмарній оптимізації. Але 7,1% втрат операцій – неприйнятний рівень для будь-чого серйознішого за соціальні реакції. Запропонована модель поступається Firebase за швидкістю на 21% у кращому випадку (328 мс на iPhone), але повністю усуває втрати і не залежить від Google.

Обмеження дослідження. Модель перевірено на трьох пристроях і прототипі – масштабування до $n > 10^5$ клієнтів потребує окремого дослідження, зокрема компактизації векторних годинників (dotted version vectors або interval tree clocks). Алгоритм SyncOnRestore не описує повне відновлення після часткової синхронізації (partial sync failure) – це виходить за межі поточної роботи. Прив'язка до WatermelonDB і React Native є особливістю прототипу, але не архітектурним обмеженням моделі.

Висновки. У роботі запропоновано архітектурну модель offline-first мобільного застосунку з трирівневою структурою (Local Store, Operation Queue, Sync Layer), формалізованим механізмом виявлення конфліктів через векторні годинники, CRDT-орієнтованою стратегією злиття з ескалацією, розширеною формулою L_sync із компонентою L_reconciliation і метрикою узгодженості C.

Тестування на трьох реальних платформах підтвердило три основних результати. По-перше, гібридна CRDT-стратегія забезпечує C = 0,943–0,951 і нульову втрату операцій, тоді як LWW втрачає 8,3%. По-друге, Offline-First архітектура тримає час відгуку 350–420 мс незалежно від стану мережі – Cloud-Only при нестабільному зв'язку показує сплески до 1800 мс. По-третє, компонента L_reconciliation, ігнорована у попередніх моделях, досягає 97 мс при тривалих офлайн-сесіях і є суттєвою для точного прогнозування L_sync.

Запропонована модель не залежить від хмарного провайдера. Вона придатна для медицини, логістики, польової автоматизації – усюди, де мережа є ненадійним ресурсом.

Перспективи подальших досліджень: аналіз поведінки при $n > 10^5$ накопичених операцій; вплив mobile edge computing на компоненту L_reconciliation; розширення моделі на peer-to-peer синхронізацію без центрального сервера.

Список літератури:

1. Brewer E. A. Towards robust distributed systems. Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC). 2000. P. 7. URL: https://doi.org/10.1145/343477.343502.
2. Shapiro M., Preguiça N., Baquero C., Zawirski M. Conflict-free Replicated Data Types. Proceedings of SSS 2011. P. 386–400. URL: https://doi.org/10.1007/978-3-642-24550-3_29.
3. Vogels W. Eventually consistent. Communications of the ACM. 2009. Vol. 52, No. 1. P. 40–44. URL: https://doi.org/10.1145/1435417.1435432.
4. Apache CouchDB / PouchDB Documentation. PouchDB: The JavaScript Database that Syncs. 2023. URL: https://pouchdb.com/guides/.
5. Google Firebase. Cloud Firestore Data Model and Offline Persistence. Firebase Documentation. 2024. URL: https://firebase.google.com/docs/firestore/manage-data/enable-offline.
6. Amazon Web Services. AWS AppSync: Delta Sync for Offline Support. AWS Documentation. 2023. URL: https://docs.aws.amazon.com/appsync/latest/devguide/tutorial-delta-sync.html.
7. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media. 2017. 616 p.
8. Terry D. B., Theimer M. M., Petersen K. et al. Managing update conflicts in Bayou. Proceedings of the 15th ACM SOSP. 1995. P. 172–182. URL: https://doi.org/10.1145/224056.224070.
9. Almeida P. S. Approaches to Conflict-free Replicated Data Types. ACM Computing Surveys. 2024. Vol. 57, No. 2. Article 51. URL: https://doi.org/10.1145/3695249.
10. Bailis P., Ghodsi A. Eventual consistency today: limitations, extensions, and beyond. ACM Queue. 2013. Vol. 11, No. 3. P. 20–32. URL: https://doi.org/10.1145/2460276.2462076.
11. DeCandia G., Hastorun D., Jampani M. et al. Dynamo: Amazon's highly available key-value store. Proceedings of 21st ACM SOSP. 2007. P. 205–220. URL: https://doi.org/10.1145/1294261.1294281.
12. Helland P. Immutability Changes Everything. Proceedings of CIDR. 2015. URL: https://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf.
13. Lakshman A., Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review. 2010. Vol. 44, No. 2. P. 35–40. URL: https://doi.org/10.1145/1773912.1773922.
14. Weiss S., Urso P., Molli P. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. IEEE Transactions on Parallel and Distributed Systems. 2010. Vol. 21, No. 8. P. 1162–1174. URL: https://doi.org/10.1109/TPDS.2009.173.
15. Mattern F. Virtual time and global states of distributed systems. Proceedings of the International Workshop on Parallel and Distributed Algorithms. 1989. P. 215–226.
16. Gray J., Helland P., O'Neil P., Shasha D. The dangers of replication and a solution. Proceedings of ACM SIGMOD. 1996. P. 173–182. URL: https://doi.org/10.1145/233269.233330.
17. Brewer E. A. CAP twelve years later: how the rules have changed. IEEE Computer. 2012. Vol. 45, No. 2. P. 23–29. URL: https://doi.org/10.1109/MC.2012.37.
18. Kleppmann M., Beresford A. R. A conflict-free replicated JSON datatype. IEEE Transactions on Parallel and Distributed Systems. 2017. Vol. 28, No. 10. P. 2733–2746. URL: https://doi.org/10.1109/TPDS.2017.2697382.
19. Shapiro M., Preguiça N. Designing a commutative replicated data type. INRIA Research Report RR-6320. 2007. URL: https://hal.inria.fr/inria-00177693.
20. Nozbe / Pietruszewski R. WatermelonDB: Reactive & asynchronous database for powerful React and React Native apps. GitHub repository. 2023. URL: https://github.com/Nozbe/WatermelonDB.
21. Meta Platforms. React Native: A framework for building native apps using React. Official Documentation. 2024. URL: https://reactnative.dev/docs/getting-started.

Liushenko A.M. DESIGN AND DATA SYNCHRONIZATION MODELS FOR OFFLINE-FIRST SCALABLE MOBILE APPLICATIONS

There is a whole class of mobile tasks where network connectivity is not a resource but an obstacle. A field medic cannot wait for a signal. A logistics operator in a low-coverage zone cannot either. This paper is about designing systems where offline is the default state, not an emergency fallback. Storing data locally is a solvable problem. The real difficulty appears when two devices have independently edited the same record without any connection between them, and the network finally recovers. Which version is correct? Existing solutions only partially answer this: PouchDB lacks queue prioritization, Firebase Firestore loses 7.1% of operations as a baseline (LWW on Samsung A53 reaches 8.3% – these are different systems), and AppSync is tied to the AWS ecosystem.

The paper proposes a three-layer model consisting of Local Store, Operation Queue, and Sync Layer, where each layer has a single responsibility and no knowledge of its neighbors. Conflicts are detected via vector clocks before any data is sent to the server. Automatic merging relies on CRDT structures – G-Counter, OR-Set, and LWW-Register – while semantically complex conflicts are escalated to the application layer. The synchronization latency formula L_sync is extended with a fifth component, L_reconciliation, which was absent in prior models. During offline sessions longer than 30 minutes it reached 97 ms – more than the CRDT merge itself – making it a significant factor in accurate latency estimation. A consistency metric C = 1 − (N_conflicts / N_updates) is introduced to measure synchronization reliability.

The model was validated on three real devices – Samsung Galaxy A53, Apple iPhone 13, and Xiaomi Redmi Note 11 – with 200 synchronization cycles per network profile. The hybrid CRDT strategy produced zero operation loss with 94.7% automatic conflict resolution and C = 0.943–0.951. Under the same conditions, LWW lost 8.3% of operations. The model is provider-independent and can be deployed on any mobile platform without architectural changes.

Keywords: *offline-first mobile applications, distributed mobile systems, mobile data synchronization, eventual consistency, offline data replication, conflict resolution strategies, mobile distributed data management, scalable mobile architectures.*

Дата першого надходження статті до видання: 27.03.2026
Дата прийняття статті до друку після рецензування: 22.04.2026
Дата публікації (оприлюднення) статті: 19.05.2026