МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу

Кафедра інженерії програмного забезпечення

Пояснювальна записка до дипломної роботи

магістра
(освітній ступінь)

на тему «Експериментальне дослідження структур безконфліктних реплікованих типів даних в колаборативних офлайн застосунках»

ХАІ.603.667п1.121.156341.20О

Виконав: студент 6 курсу групи № 667п1
Спеціальність 121 – Інженерія програмного забезпечення
(код та найменування)
Освітня програма Хмарні обчислення та Інтернет речей
(найменування)
Луценко А.А.
(прізвище й ініціали студента)
Керівник Мандрікова Л.В.
(прізвище та ініціали)
Рецензент Ільїна І.В.
(прізвище та ініціали)

Харків – 2020

Міністерство світи і науки України
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу
(повне найменування)
Кафедра інженерії програмного забезпечення
(повне найменування)
Рівень вищої освіти другий (магістерський)

Спеціальність 121 – інженерія програмного забезпечення
(код та найменування)
Освітня програма хмарні обчислення та Інтернет речей
(найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ _______І. Б. Туркін___________
(підпис) (ініціали та прізвище)
"_____" __________________ 2020 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Луценко Артуру Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема дипломного проекту *Експериментальне дослідження структур безконфліктних реплікованих типів даних в колаборативних офлайн застосунках*

керівник дипломного проекту *Мандрікова Людмила Василівна, к.т.н.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "__" ___2020 року № __

2. Термін подання студентом роботи ______________________________

3. Вихідні дані до роботи: *мобільний додаток для колаборативної роботи з текстовими документами на базі операційної системи iOS та сховища даних CloudKit*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1) критичний аналіз проблем розробки колаборативних офлайн застосунків;
2) аналіз проблем реплікації даних;
3) розроблення програмного додатку для створення та редагування текстових документів у колаборативному режимі для експериментальних досліджень;
4) аналіз отриманих даних експериментального дослідження

5. Перелік графічного матеріалу
РПЗ – стор. 99, рисунків – 39 шт., таблиць – 1 шт., презентація – 16 слайдів.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Мандрікова Л.В., доц. каф. 603		
2	Мандрікова Л.В., доц. каф. 603		
3	Мандрікова Л.В., доц. каф. 603		

8. Нормоконтроль _____ __В.А. Постернакова__ «____» _______2020 р.
(підпис) (ініціали та прізвище)

7. Дата видачі завдання ____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту	Строк виконання етапів проекту	Примітка
1	Отримання і затвердження теми диплому	03.09.2019	
2	Аналіз предметної області	04.09.2019	
3	Постановка задачі	20.11.2019	
4	Проведення теоретичних досліджень	22.11.2019	
5	Розробка прототипу ПЗ	02.09.2020	
6	Підготовка пояснювальної записки	22.10.2020	
7	Оформлення пояснювальної записки до дипломного проекту	10.11.2020	
8	Передзахист дипломного проекту	27.11.2020	
9	Захист дипломного проекту	16.12.2020	

Студент _____________ ____Луценко А.А. ____
(підпис) (прізвище та ініціали)

Керівник роботи _______________ ___Мандрікова Л.В._
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломного проєкту містить 99 стор., 39 рис., 1 додаток, 30 джерел.

Об'єкт дослідження – процес колаборативного опрацювання документів.

Предмет дослідження – моделі безконфліктних реплікованих типів даних.

Метою дослідження є підвищення ефективності колаборативних офлайн застосунків при роботі з текстовими документами шляхом використання моделей безконфліктних реплікованих типів даних для узгодженості даних.

Для досягнення поставленої мети необхідно розв'язати такі задачі: провести критичний аналіз проблем розробки колаборативних офлайн застосунків; провести аналіз проблем реплікації даних; розробити програмний додаток для створення та редагування текстових документів у колаборативному режимі для експериментальних досліджень; провести аналіз отриманих даних експериментального дослідження.

Методи досліджень. У роботі було використано методи розробки, що базуються на платформі iOS, мові програмування Swift, UIKit, сховища даних CloudKit, менеджеру пакетів Swift PM.

Наукова новизна. Удосконалено модель безконфліктних реплікованих типів даних, яка на відміну від існуючих використовує модель суворої узгодженості у кінцевому рахунку, що дасть змогу провести узгодженість даних.

Практична значимість отриманих результатів. В результаті роботи був розроблений мобільний додаток для колаборативної роботи з текстовими документами на базі операційної системи iOS та сховища даних CloudKit. В подальшому, реалізований програмний модуль потребує впровадження підтримки більш складних структур CRDT, що дозволить використовувати його в інших галузях окрім роботи з текстовими документами.

БЕЗКОНФЛІКТНІ РЕПЛІКОВАНІ ТИПИ ДАНИХ, КОЛАБОРАЦІЯ, МОБІЛЬНИЙ ДОДАТОК, РЕПЛІКАЦІЯ, CLOUDKIT, CRDT, IOS, SWIFT

ABSTRACT

Explanatory note to the master's thesis 99 pp., 39 fig., 1 app., 30 sources.

The object of study - the process of collaborative processing of documents.

The subject of research - models of conflict-free replicated data types.

The aim of the study is to increase the efficiency of collaborative offline applications when working with text documents by using models of conflict-free replicated data types for data consistency.

To achieve this goal it is necessary to solve the following tasks: to conduct a critical analysis of the problems of developing collaborative offline applications; analyze data replication problems; develop a software application for creating and editing text documents in collaborative mode for experimental research; to analyze the obtained data of the experimental study.

Research methods. Development methods based on the iOS platform, Swift programming language, UIKit, CloudKit data warehouse, Swift PM package manager were used in the work.

Scientific novelty. The model of conflict-free replicated data types has been improved, which, in contrast to the existing ones, uses a model of strict consistency in the end, which will allow for data consistency.

The practical significance of the obtained results. As a result, a mobile application was developed for collaborative work with text documents based on the iOS operating system and the CloudKit data warehouse. In the future, the implemented software module requires the introduction of support for more complex CRDT structures, which will allow its use in other areas besides working with text documents.

CONFLICT-FREE REPLACED DATA TYPES, COLLABORATION, MOBILE APP, REPLICATION, CLOUDKIT, CRDT, IOS, SWIFT

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8

ВСТУП........ 9

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ 12

1.1 Класифікація моделей роботи з даними 12

1.2 Аналіз проблем розробки колаборативних офлайн застосунків 14

1.3 Проблема реплікації даних та моделі узгодженості 16

1.3.1 Причинний зв’язок 19

1.4 Існуючі підходи до кінцевої узгодженості у офлайн застосунках 20

1.4.1 Operational Transformation 20

1.4.2 Диференціальна синхронізація 21

1.4.3 Conflict-Free Replicated Data Types 32

1.5 Постановка мети й завдань дослідження 33

1.6 Висновки по розділу 1 34

2 ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ СТРУКТУР БЕЗКОНФЛІКТНИХ РЕПЛІКОВАНИХ ТИПІВ ДАНИХ 35

2.1 Базові поняття 35

2.2 Моделі реплікації 47

2.2.1 Реплікація на основі передачі стану 47

2.2.2 Реплікація на основі передачі операцій 48

2.3 Класифікація CRDT 50

2.4 Структури CRDT 55

2.4.1 Лічильники 56

2.4.2 Регістри 58

2.4.3 Множини 61

2.5 Робота з текстом 64

2.3 Висновки по розділу 2 65

3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ СТРУКТУР БЕЗКОНФЛІКТНИХ РЕПЛІКОВАНИХ ТИПІВ ДАНИХ В КОЛАБОРАТИВНИХ ОФЛАЙН ЗАСТОСУНКАХ 66

3.1 Проектування програмної системи 66

3.2 Технології для реалізації програмного забезпечення 70

3.3 Опис програмної реалізації мобільного додатку 71

3.4 Опис реалізації CRDT структур для роботи з текстом 75

3.4.1 Опис вирішення проблеми збіжності та досягнення загального порядку . 75

3.4.2 Опис реалізації підходу RGA на базі суміжного масиву 77

3.4.3 Опис реалізації підходу RGA на базі причинних дерев 78

3.5 Порівняння реалізацій 82

3.5 Висновки по розділу 3 83

ВИСНОВКИ 84

ПЕРЕЛІК ПОСИЛАНЬ 86

ДОДАТОК А 89

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

OT - Operational Transformation.

CRDT - Conflict-Free Replicated Data Types.

ВСТУП

В даний момент досягли широкого розповсюдження різноманітні мобільні та веб-застосунки, а вимоги користувача до програмного забезпечення постійно зростають. Однією з сучасних вимог є стабільна робота програмного продукту незалежно від якості мережевого підключення та його доступність у режимі офлайн. Дана вимога є особливо актуальною для сучасних колаборативних застосунків для роботи з документами. Більшість з них рано чи пізно потребує синхронізації та режиму колаборативної роботи. Відповідно до цього з'являється необхідність вирішення багатьох проблем, що присутні у розподілених програмних системах.

Колаборативне програмне забезпечення створено для взаємодії користувачів між собою та володіння певним спільним ресурсом, доступним для модифікації [1]. Колаборативні застосунки у більшості випадків проектуються як мобільні або веб-застосунки. Відомими продуктами є Evernote, Trello, Notes та інші. Дуже поширеними є застосунки для спільної роботи з документами, наприклад Google Docs, Office 365.

Основна проблема та складність розробки колаборативних застосунків полягає у правильному виборі моделі роботи з даними. Неправильний вибір моделі може мати незворотній негативний ефект для успіху продукту.

Для колаборативної роботи з документами достатньо добре підходить офлайн модель [2]. Основними проблемами є розподілення даних та їх реплікація під час синхронізації змін, внесених кожним з клієнтів, а також наявність механізму вирішення або уникнення конфліктів.

Існує достатня кількість рішень, які є пропрієтарними, але часто їх використання вносить додаткові ризики для підтримки продукту, а також розширення його функціональності – в першу чергу тому, що розробник не має повного доступу до найбільш важливих компонентів системи, що пов'язані з реплікацією.

Ad-hoc рішення є ризиковими, сильно ускладнюють протокол комунікації та можуть бути помилковими. Виходячи з цього, є необхідність звернення до наукового підґрунтя. Для гарантії стабільності програмного продукту код має бути локально доведеним, тобто збіжність має бути математично підтвердженою [3]. З даною метою використовують незмінні об’єкти, слабко пов’язані модулі, ідемпотентні функції тощо.

Виділяють два основних інструменти для вирішення даних проблем:

– безконфліктні репліковані типи даних (CRDT);

– Operational Transformation (OT).

Operational Transformation має більш високі вимоги до серверу, більш складні та менш стабільні алгоритми, які пов’язані з мутацією вхідних операцій. Деякі дослідження довели, що його алгоритми іноді не сходяться, як було заявлено у реалізації.

Підхід CRDT майже повністю протилежний Operational Transformation – він полягає у розгляданні проблеми синхронізації з точки зору проектування самих структур даних, а не послідовності операцій над ними [4].

Останнім часом набувають популярності програмні рішення типу «serverless», де логіка роботи з даними частково або повністю реалізується на стороні клієнта [5]. CRDT може застосовуватись в подібних архітектурних рішеннях та навіть у повністю розподілених та децентралізованих системах.

Таким чином актуальною є тема аналізу моделей безконфліктних реплікованих типів даних у якості інструменту для проведення реплікації у колаборативних офлайн застосунках.

Об'єктом дослідження – процес колаборативного опрацювання документів.

Предмет дослідження – моделі безконфліктних реплікованих типів даних.

Метою дослідження є підвищення ефективності колаборативних офлайн застосунків при роботі з текстовими документами шляхом використання моделей безконфліктних реплікованих типів даних для узгодженості даних.

Для досягнення поставленої мети необхідно розв'язати такі задачі:

– провести критичний аналіз проблем розробки колаборативних офлайн застосунків;

– провести аналіз проблем реплікації даних;

– розробити програмний додаток для створення та редагування текстових документів у колаборативному режимі для експериментальних досліджень;

– провести аналіз отриманих даних експериментального дослідження.

Методи досліджень. У роботі було використано методи розробки, що базуються на платформі iOS, мові програмування Swift, UIKit, сховища даних CloudKit, менеджеру пакетів Swift PM.

Наукова новизна. Удосконалено модель безконфліктних реплікованих типів даних, яка на відміну від існуючих використовує модель суворої узгодженості у кінцевому рахунку, що дасть змогу провести узгодженість даних.

Практична значимість отриманих результатів. В результаті роботи був розроблений мобільний додаток для колаборативної роботи з текстовими документами на базі операційної системи iOS та сховища даних CloudKit. В подальшому, реалізований програмний модуль потребує впровадження підтримки більш складних структур CRDT, що дозволить використовувати його в інших галузях окрім роботи з текстовими документами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Класифікація моделей роботи з даними

В даний час існує достатньо велика кількість застосунків, що функціонують на різноманітних пристроях. Такими є мобільні та веб-застосунки. Роботу з багатьма з них практично неможливо уявити без підключення до мережі Інтернет.

Підключення до глобальної мережі надає можливість дистанційно взаємодіяти з пристроями інших користувачів або пристроями, що просто зберігають певні дані. За наявності підключення до мережі пристрій знаходиться у стані онлайн. Протилежним станом є офлайн – стан, коли комп’ютер не підключений до мережі або до будь-якого іншого пристрою, що додає певні обмеження на взаємодію користувача з програмним забезпеченням [6]. Виходячи з цього, в основу проектування та розробки програмного забезпечення мають бути закладені різні архітектурні принципи.

Існує певний недолік знань, пов’язаних з проектуванням онлайн та офлайн застосунків. Наслідки вибору неправильного підходу можуть бути дуже серйозними та мати великий вплив на успіх проекту. Дуже важливим є вибір правильної моделі роботи з даними. Виділяють 3 моделі [2]:

- онлайн модель;
- офлайн модель;
- змішана модель.

Кожна з моделей має свою ступінь толерантності до відсутності підключення до мережі.

В багатьох випадках, коли вимоги до роботи з даними недостатньо чітко сформульовані, де насправді необхідна офлайн модель, реалізовують онлайн модель з наявністю кешування. Внаслідок цього не підтверджуються попередні

очікування користувачів, клієнтів та партнерів, що негативно впливає на програмний продукт, який в подальшому повинен бути перепроектований з самого початку з урахуванням усіх вимог, або взагалі припинить своє існування на ринку.

Дуже важливо, щоб дані моделі роботи з даними були повністю зрозумілі перед тим, як починати роботу над проектом. Це стосується усіх типів застосунків, тому що ключовим фактором є дані та їх передача, а не особливості пристроїв чи програмних бібліотек.

Типовим прикладом онлайн моделі є веб-застосунки. Онлайн застосунки у більшості випадків базуються на HTML, а увесь програмний код та ресурси завантажуються з серверу кожного разу, коли користувач запитує відповідний URL. Оскільки користувач очікує, що програма відповідає на його дії та запити без затримки, підключення до мережі є абсолютно необхідним. Основним обмеженням даної моделі є пропускна здатність каналу зв’язку. Низька пропускна здатність не дозволяє передачу великих об’ємів даних, або дуже негативно впливає на досвід користувача. Враховуючи цю особливість, в багатьох випадках замість того, щоб завантажувати одні й ті самі дані щоразу, використовують кешування. Дана модель роботи з даними є найпростішою для реалізації та розробки MVP, але менш продуктивною у мобільних застосунках [7].

Офлайн модель є протилежністю онлайн моделі. Найбільш часто вона реалізується як нативний застосунок. Типовими прикладами офлайн застосунків є месенджери, почтові клієнти, тощо. У режимі відсутності підключення до мережі вони підтримують свій базовий функціонал, що дозволяє продовжити користуватися застосунком за умови нестабільного з’єднання з мережею або його повної відсутності.

Виділяють декілька вимог до офлайн моделі.

Першою вимогою є здатність до розподілення даних [2]. Потрібно мати можливість завантаження лише підмножини даних на пристрій користувача,

оскільки сервери можуть мати доступ до великих баз даних, які фізично неможливо завантажити на пристрій.

Другою та найбільш складною для реалізації вимогою є необхідність підтримки різноманітних сценаріїв реплікації даних. В багатьох сценаріях система повідомляє клієнту про список змін, що відбулися в масиві необхідних даних з моменту останнього запиту. Система може надати клієнту певний токен, що відповідає деякій контрольній точці [2]. В деяких системах виділяють також токени пропуску, що дозволяє клієнтам завершити реплікацію за декілька разів [2]. Дуже важливим аспектом є здатність до вирішення конфліктів даних між декількома репліками або наявність механізму їх уникнення.

Змішана модель певною мірою має властивості онлайн та офлайн моделі. Вона є простішою для реалізації ніж офлайн модель. Дана модель активно користується можливістю кешування даних. В офлайн режимі програмне забезпечення дозволяє користувачу отримати дані, що були актуальними на момент останнього підключення до мережі, але в цей же час в більшості випадків блокує функціонал, пов’язаний з модифікацією даних. Таким чином дана модель уникає необхідності реалізації складних механізмів реплікації, але надає гірший досвід користувача.

1.2 Аналіз проблем розробки колаборативних офлайн застосунків

Колаборативне програмне забезпечення – це прикладне програмне забезпечення, що розроблене для людей, що працюють над спільним завданням для досягнення своїх цілей [1].

Виділяють три основні способи взаємодії людей у якості користувачів: розмови, транзакції та співпраця. Розмовна взаємодія є обміном інформації між декількома учасниками, де відсутня центральна сутність, навколо якої відбувається взаємодія. У транзакційній взаємодії головним є зміна відносин між

учасниками такої взаємодії. В свою чергу, взаємодія у співпраці передбачає наявність деякого спільного центрального ресурсу.

Однією з основних цілей розробки колаборативного програмного забезпечення було перетворення способу обміну мультимедійними повідомленнями та документами для забезпечення більш ефективної колективної співпраці [8]. Узагальнюючи, подібне програмне забезпечення можна класифікувати на 2 типи:

– платформи спільного редагування контенту в режимі реального часу, що дозволяють багатьом користувачам одночасно редагувати єдиний спільний документ [8];

– платформи контролю версій, що дозволяють різним користувачам вносити паралельні зміни у документ, зберігаючи при цьому зміни зроблені окремим користувачем у вигляді окремих файлів, які є варіантами актуального робочого файлу [8].

Більшість колаборативного програмного забезпечення реалізовано у вигляді мобільних та веб-застосунків. Зокрема, розповсюдженими є Office 365, Google Docs, Trello, Evernote, Notes та інші. Можна зробити висновок, що ПЗ для роботи з документами є одним з найбільш поширених.

Основна проблема та складність розробки колаборативних застосунків для роботи з документами полягає у правильному виборі моделі роботи з даними. Великий відсоток даного програмного забезпечення використовує офлайн-модель роботи з даними, що в свою чергу передбачає необхідність вирішення усіх проблем та складностей, властивих даній моделі, пов’язаних з:

– розподіленням даних;

– реплікацією даних.

Проблема реплікації є основною в даному випадку. Подібні системи повинні дозволяти двом або більше користувачам одночасно працювати з документом: читати та редагувати його.

У випадку, коли використовується офлайн модель, користувачі не зіштовхуються з будь-яким блокуванням на доступ до документу та в певний час у майбутньому синхронізують свої зміни з актуальним на той момент станом документу, а отже – проводять реплікацію своїх даних.

Змішана модель гірше підходить для колаборативних застосунків через меншу толерантність до підключення до мережі.\

1.3 Проблема реплікації даних та моделі узгодженості

Проблема реплікації є однією з найбільш актуальних проблем у розподілених системах. Проектування подібних систем передбачає наявність певних компромісів.

Існує твердження, що для будь-якої розподіленої системи неможливо забезпечити виконання одночасно трьох наступних властивостей:

– доступність (усі вузли системи повинні мати однакові дані в будь-який довільний проміжок часу);

– узгодженість (клієнт повинен отримати коректну відповідь на кожен свій запит);

– толерантність до розподілу (незважаючи на розподілення на ізольовані вузли, або втрату в’язку з їх частиною, система має стабільно та коректно відповідати на запити).

Дане твердження описано у теоремі CAP, що представлена графічно на рисунку 1.1 [9]. CAP-теорема також відома, як теорема Брюера.

Як наслідок теореми, розподілені системи можна розподілити на три класи, в залежності від пари забезпечених властивостей (рисунок 1.1).

CA – система не підтримує стійкість до розподілення, але забезпечує узгодженість та доступність. Такими є системи, що підтримують вимоги ACID, наприклад реляційні СУБД [10].

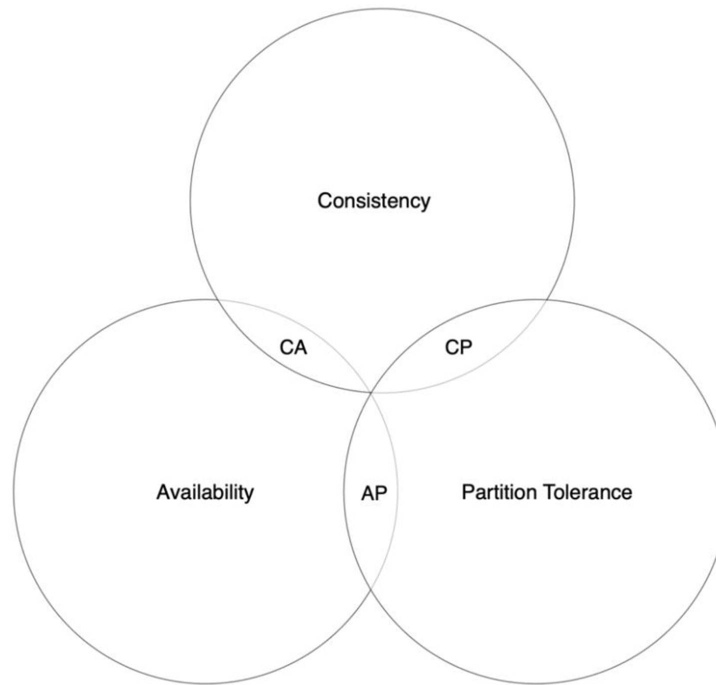

Рисунок 1.1 – Теорема CAP

CP – система не гарантує доступність, але гарантує повну цілісність даних на усіх вузлах та здатність до розподілення. Такими є фінансові системи, зокрема банківські мережі або банкомати, у яких узгодженість є найвищим пріоритетом.

AP – система не гарантує цілісності, але забезпечує високу доступність і працездатність при розподіленні. Такою системою є DNS, а також деякі NoSQL системи, що посилаються на теорему CAP.

Колаборативні офлайн застосунки повинні:

– гарантувати високу доступність;

– гарантувати постійну можливість оновлювати дані незалежно від доступності інших вузлів системи.

Виходячи з цього, в контексті розробки офлайн застосунків не можна відмовитись від гарантії доступності та здатності до розподілення. Отже необхідно повністю або частково вирішити проблему, пов’язану з узгодженістю даних обравши потрібну модель узгодженості даних.

Модель узгодженості – це конкретний підхід, що використовується у розподілених системах для забезпечення узгодженості даних [11]. Виділяють наступні основні моделі:

– сувора узгодженість;
– послідовна узгодженість;
– причинна узгодженість;
– слабка узгодженість;
– узгодженість у кінцевому рахунку;
– узгодженість по входи та виходу.

Отже є достатня кількість альтернатив суворій узгодженості, що дозволяє обирати різні моделі для кожної конкретної задачі.

Гарантію постійною доступності надає модель кінцевої узгодженості (EC), яка є видом слабкої моделі узгодженості [12]. Даний підхід також називається оптимістичною реплікацією та широко застосовується у розподілених системах, зокрема тих, що мають свій мобільний клієнт.

Модель кінцевої узгодженості передбачає, що реплікація проходить асинхронно. Оновлені дані будуть знаходитись в узгодженому стані через деякий час у кінцевому рахунку в майбутньому. Тобто дані оновлюються локально, розсилаючи іншим реплікам оновлення у той момент часу, коли це буде можливим. Даний алгоритм задовольняє вимогам офлайн моделі роботи з даними та може бути використаним у колаборативних офлайн застосунках. В даному випадку мобільний або веб-клієнт дозволяє користувачу виконувати маніпуляції з даними локально, після чого повідомляє API про внесені зміни при першій можливості. Відмінність моделі кінцевої узгодженості полягає у тому, що консенсус між репліками є необхідним не у реальному часі, порівнюючи з моделлю суворої узгодженості. Відсутність певного методу вирішення конфліктів між репліками під час реплікації залишається проблемою. Існує також підвид моделі кінцевої узгодженості – модель суворої кінцевої

узгодженості (SEC), у якій присутні усі властивості, що наявні EC, але додатково мається певний визначений алгоритм для вирішення конфліктів.

Отже, вирішується проблема необхідності консенсусу між репліками. Можна відмітити, що модель суворої кінцевої узгодженості частково вирішує проблеми теореми CAP та резолюції конфліктів реплікації, що дозволяє використати її у колаборативних офлайн застосунках.

1.3.1 Причинний зв’язок

Виділяють декілька важливих термінів для розуміння EC та SEC. Вважається, що система складається з пристроїв, які працюють паралельно, кожен з яких створює операції, що мутують дані та обмінюються інформацією з іншими пристроями. В даному контексті визначають поняття причинності – операція спричинена іншою операцією, якщо вона безпосередньо модифікує або просто включає результат роботи іншої операції.

Визначення причинності є критично важливим для відновлення часової шкали (або лінеаризації) операцій по всій мережі [13]. Операція, що спричиняє іншу операцію, має завжди бути упорядкованою раніше спричиненої операції [14]. Алгоритми припускають, що операція спричинила іншу, якщо пристрій вже знав про стару операцію до створення нової. Отже, кожна операція має своє певне причинне минуле.

Для визначення причинності використовують часові мітки. Найпростішою з них є часова мітка Лампорта, яка вимагає, щоб нова операція мала більш високу часову позначку, ніж будь-яка інша відома пристрою операція [15]. Незважаючи на існування схем узгодженості, що приймають операції у довільному порядку, більшість алгоритмів покладаються на слідування причинному порядку. Наприклад, вставка обов'язково надходить до видалення.

Існують також операції, що не мають причинно-наслідкового зв’язку. Такими є одночасні операції, що були створені одночасно з декількох пристроїв, не знаючи про існування одне одного. Основна складність використання моделі

ЕС зводиться до вирішення проблеми роботи саме з одночасними операціями. Подібні операції повинні бути комутативними. Є декілька варіантів вирішення конфлікту у подібній ситуації. Застосувавши більш складну часову мітку – вектор версій, можна визначити чи є декілька подій послідовними або одночасними [15]. Однак дане рішення є складним у застосуванні через те, що кожна часова мітка має включати в себе значення кожного пристрою системи.

Наступним поняттям є журнал подій. Зберігаючи операції, як дані можна використовувати їх для реконструкції об’єкта моделі. Журнал подій, пов’язаних причинним зв’язком має частковий порядок, тому що одночасні операції можуть знаходитись у різних позиціях на різних пристроях залежно під порядку їх отримання. В противному випадку, журнал подій має загальний порядок, якщо журнал повністю ідентичний на всіх пристроях.

Загальний порядок може бути забезпечений за допомогою сортування операцій за часовою міткою, а також деякий унікальним ідентифікатором. Але дана реалізація не може бути оптимальною через наявність квадратичної асимптотичної складності алгоритму [3].

1.4 Існуючі підходи до кінцевої узгодженості у офлайн застосунках

В даний момент існує декілька основних альтернатив для подолання проблем, пов’язаних з моделлю кінцевої узгодженості: Operational Transformation (OT) та Conflict-Free Replicated Data Types (CRDT).

Обидва підходи вирішують такі проблеми як поєднання одночасних змін та вирішення або уникнення конфліктів при реплікації у розподіленій системі.

1.4.1 Operational Transformation

Operational Transformation є лідером у галузі колаборативної роботи з документами. Даний підхід використовується у Google Docs, Wave, ShareJS.

Кожен клієнт має свою копію даних, які дозволено змінювати за допомогою певного набору операцій. При кожній мутації даних, операція пересилається усім іншим пристроям через центральний сервер. Operational Transformation припускає, що операції можуть бути застосовані лише поверх існуючого документа без можливості змін їхнього порядку. Виходячи з цього, єдиним способом роботи з одночасними операціями є їх перетворення на кожному пристрої в залежності від стеку відомих операцій.

Можна продемонструвати роботу OT на прикладі. Нехай пристрій «А» вставляє символ в індекс 3, а в одночасно з цим пристрій «Б» видаляє символ по індексу 2. В залежності від порядку отримання даних двох операцій, третій пристрій може зіштовхнутися з конфліктом. Якщо операція вставки буде отримана першою – все є коректним, але, коли операція видалення прийде раніше, операція вставки матиме некоректний індекс. Вирішенням є перетворення позиції для вставки, віднімаючи кількість видалених символів пристроєм «Б».

Подібні алгоритми стають експоненційно складнішими при наявності більшої кількості операцій, а додавання нової операції вимагає математичного доведення її збіжності з усіма можливими комбінаціями попередньо існуючих операцій. Також більшість існуючих алгоритмів OT мають математично доведені незбіжності [16].

1.4.2 Диференціальна синхронізація

Можна також виділити алгоритми диференціальної синхронізації, що використовувались у Google Docs до появи Operational Transformation [17]. Даний підхід передбачає знаходження контекстної різниць між локальними ревізіями документа для генерації потоку невеликих атомарних змін з подальшим його обміном між репліками [18]. У разі виникнення конфлікту, приймаюча репліка використовує нечітке виправлення для того, щоб найкращим чином застосувати нові зміни [19]. Після цього репліка виокремлює різницю між

утвореним локальним документом і відтвореним документом відправника та відправляє нові зміни назад, утворюючи таким чином додатковий цикл синхронізації [20].

Незважаючи на те, що на рівні абстракцій даний підхід виглядає найкращим, реалізація достатньо універсальних методів diff та patch також є достатньо складною, яку можна порівняти зі складністю ОТ. Недоліком даного підходу є те, що кінцевий результат злиття не обґрунтовується математично, а покладається лише на поведінку нечіткого виправлення. Серед іншого, підхід має певні проблеми у використанні офлайн моделі роботи з даними – є велика ймовірність того, що дані не зможуть злилися при наявності великої кількості розбіжностей між репліками на протязі великого проміжку часу, що спровокує втрату даних [3]. Також, дві репліки не можуть одночасно обмінюватись пакетами даних, через наявність додаткового циклу синхронізації, описаного раніше.

1.4.3 Conflict-Free Replicated Data Types

На відміну від Operational Transformation та диференціальної синхронізації, підхід CRDT полягає у розгляданні проблеми синхронізації з точки зору проектування самих структур даних, а не послідовності операцій над ними. Загалом, структура CRDT – це певний об’єкт, що може бути об’єднаний з будь-якими іншими об’єктами однакового типу в довільному порядку для отримання ідентичного з’єднаного об’єкта. Сама операція злиття об’єктів зазвичай є ідемпотентною, а концепт роботи є простішим, порівняно з ОТ та диференціальною синхронізацією. Транспортний рівень є повністю абстрагованим від механізму синхронізації, а отже протокол комунікації може бути легко змінений у майбутньому – наприклад HTTPS може бути змінений на Bluetooth або будь-який інший протокол передачі даних.

Слід відмітити, що для системи, що заснована на подіях (як ОТ), у більшості випадків потрібен активний сервер. На відміну від цього, система, що

базується на CRDT може бути реалізована з використанням простої бази даних, наприклад CloudKit або Firebase, та є ближчою до поняття справжньої розподіленої системи.

Отже, враховуючи переваги та недоліки представлених підходів, було вирішено використовувати безконфліктні репліковані типи даних для подальшого дослідження.

1.5 Постановка мети й завдань дослідження

У даній роботі необхідно дослідити моделі безконфліктних реплікованих типів даних у якості інструменту для вирішення конфліктів між репліками. Дослідити особливості їх використання у мобільних колаборативних офлайн застосунках для редагування текстових документів.

Розробити програмний продукт, мобільний застосунок для операційної системи iOS, що дозволить користувачу створювати текстові документи та вносити до них зміни у колаборативному режимі з декількох пристроїв одночасно. Моделі та операції для роботи з безконфліктними реплікованими типами даних реалізувати як окремий програмний модуль.

Система має задовольняти наступним вимогам:

– користувач повинен мати можливість редагування документів без затримки, навіть при відсутності підключення до мережі;

– синхронізація даних повинна виконуватись у фоновому режимі;

– злиття змін повинно бути автоматичним при спільній роботі над документом у режимі реального часу;

– користувач повинен мати можливість працювати над документом офлайн протягом невизначеного проміжку часу; система має бути продуктивної при синхронізації не лише декількох змін, а також при синхронізації тисяч локальних змін;

– користувач не повинен самостійно обирати правильну версію документу у разі виникнення конфлікту злиття даних;

– використання вторинних структур даних повинно буди зведено до мінімуму, а уся необхідна для синхронізації інформація повинна зберігатися у самому документі;

– додаткові метадані та кеш також повинні бути зведені до мінімуму;

– передача даних по мережі повинна бути зведена до мінімуму.

1.6 Висновки по розділу 1

В першому розділі дипломної роботи проведено оглаяд та аналіз проблем розроблення колаборативних офлайн застосунків та репликації даних. Був проведений аналіз існуючих моделей роботи з даними у залежності від толерантності до відсутності підключення до мережі. Вибір правильної моделі для кожного типу програмного забезпечення є дуже важливим. Для колаборативних застосунків найбільше підходить офлайн модель, тому що вона є найбільш толерантною до відсутності підключення до мережі, а доступність даних має найвищий пріоритет. Розглянуті існуючи підходи до кінцевої узгодженості у офлайн застосунках. Зроблено постановку мети й завдань дослідження.

2 ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ СТРУКТУР БЕЗКОНФЛІКТНИХ РЕПЛІКОВАНИХ ТИПІВ ДАНИХ

2.1 Базові поняття

Розподілена система складається з певної кількості процесів, поєднаних між собою асинхронною мережею. Мережа може бути розподілена та відновлена, а вузли можуть знаходитися у відключеному режимі незалежно один від одного деякий час.

Процес може зберігати атоми та об’єкти. Атом є базовим незмінним типом даних, який можна однозначно ідентифікувати за допомогою його змісту [4]. Атоми можна копіювати між процесами. Вони є еквівалентними, якщо еквівалентним є їх контент або зміст. Атомами є:

– integers;
– floats;
– strings;
– sets;
– arrays;
– tuples.

Атомами також є будь-які інші незмінні типи разом з множиною їх власних немутуючих операцій. Існує також певна конвенція щодо найменування типів даних – назви атомів починаються з малої літери.

Об’єкт – це змінний тип даних, що здатний до реплікації. Об’єкт має власну ідентичність та контент (payload), що включає в себе будь-яку кількість атомів або інших об’єктів, початковий стан та публічний інтерфейс, що складається з набору операцій [4].

Два або більше об’єктів, що мають однакову ідентичність, але розташованих у різних процесах називають репліками одне одного [4]. Наприклад, на рисунку 2.1 зображено логічний об’єкт разом з його репліками в процесах 1, 2 і 3 та поточний стан об’єкта у третій репліці [4].

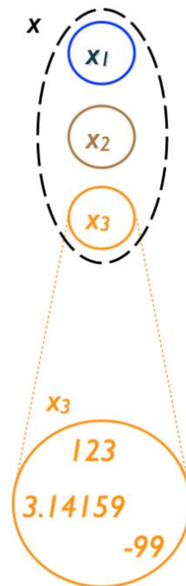

Рисунок 2.1 – Об’єкт та його репліки

Об’єкти повинні бути незалежними та не підтримують транзакції. Отже, зберігається фокус лише на одному об’єкті в певний момент часу.

Система складається з невизначених клієнтів, що читають та модифікують стан об’єкту за допомогою операцій з його публічного інтерфейсу. Клієнт обирає будь-яку необхідну в даний момент репліку, що називають реплікою-джерелом. Запит на операцію виконується локально в даній репліці.

Оновлення виконується у 2 фази:

– клієнт виконує операцію на репліці-джерелі та, за необхідністю, проводить певну початкову обробку;

– оновлення передається асинхронно всім реплікам.

2.2 Моделі реплікації

В літературі розділяють 2 підходи до проведення реплікації:

– заснований на передачі стану;

– заснований на передачі операцій.

2.2.1 Реплікація на основі передачі стану

Реплікацію, засновану на передачі стану називають пасивною реплікацією [21]. Відповідні об'єкти називаються пасивними. В даному підході оновлення відбувається повністю у репліці-джерелі, а потім поширюється шляхом передачі повного модифікованого стану між репліками, як представлено на рисунку 2.2 [4].

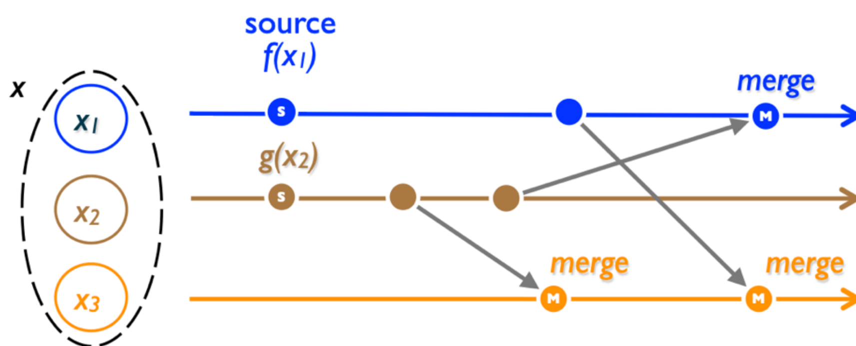

Рисунок 2.2 – Пасивна реплікація

Загальна специфікація пасивних об'єктів представлена на рисунку 2.3 [4]. Об'єкт має певний контент та початковий стан у кожній репліці. Об'єкт також має підтримувати операції «update» та «query», що виконуються атомарно (в межах однієї репліки).

```
1:  payload Payload type; instantiated at all replicas
2:      initial Initial value
3:  query Query (arguments) : returns
4:      pre Precondition
5:      let Evaluate synchronously, no side effects
6:  update Source-local operation (arguments) : returns
7:      pre Precondition
8:      let Evaluate at source, synchronously
9:      Side-effects at source to execute synchronously
10: compare (value1, value2) : boolean b
11:     Is value1 ≤ value2 in semilattice?
12: merge (value1, value2) : payload mergedValue
13:     LUB merge of value1 and value2, at any replica
```

Рисунок 2.3 – Специфікація пасивного об’єкту

Для безпеки, оновлення об’єкту відбувається тільки після валідації попередньо відомої опціональної передумови. Після локального оновлення система передає повну копію стану об’єкту між довільними парами реплік для поширення змін. Це дозволяє оновити стан репліки приймача з результатом операції злиття, викликаного з двома аргументами: локальним станом приймача та отриманим станом. Будь-яке оновлення повинно бути записано до історії кожної репліки для забезпечення життєдіяльності системи. Для цього припускається наявність певного процесу синхронізації, що передає стани між парами реплік у невизначений час та нескінченно часто [4]. При цьому необхідно відзначити, що зв’язки між репліками повинні утворювати пов’язаний граф.

2.2.2 Реплікація на основі передачі операцій

Реплікацію, засновану на передачі операцій називають активною реплікацією. Відповідні об’єкти називаються активними. На відміну від пасивної реплікації, система поширює операції та зміни замість цілого стану репліки, що представлено на рисунку 2.4 [4].

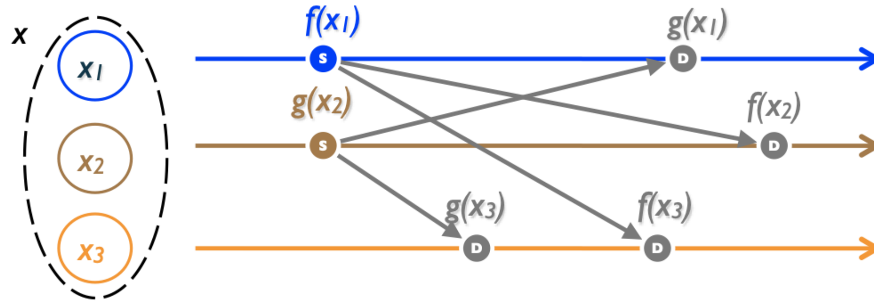

Рисунок 2.4 – Активна реплікація

Об’єкти, що використовуються у активній реплікації мають відмінності у реалізації операції оновлення. Специфікація активного об’єкту представлена на рисунку 2.5 [4].

```
 1: payload Payload type; instantiated at all replicas
 2:     initial Initial value
 3: query Source-local operation (arguments) : returns
 4:     pre Precondition
 5:     let Execute at source, synchronously, no side effects
 6: update Global update (arguments) : returns
 7:     atSource (arguments) : returns
 8:         pre Precondition at source
 9:         let 1st phase: synchronous, at source, no side effects
10:     downstream (arguments passed downstream)
11:         pre Precondition against downstream state
12:         2nd phase, asynchronous, side-effects to downstream state
```

Рисунок 2.5 – Специфікація активного об’єкту

Отже, виходячи з рисунку, процес оновлення розбивається на 2 фази.

Перша фаза, що позначена як «atSource» виконується повністю на репліці-джерелі. Вона також має опціональну передумову, виконується атомарно та отримує аргументи з виклику операції. На даній фазі заборонено робити будь-які побічні ефекти. Можливо лише синхронно обчислити локальний результат, повернути його до викликаючого блоку коду та підготувати аргументи для наступної другої фази [4].

Друга фаза має назву «downstream». Після перевірки передумови створюються операції з замиканнями для виконання на інших репліках. Дана фаза виконується одразу після першої фази синхронно на поточній репліці та асинхронно на інших репліках. Операція виконується атомарно на кожній з реплік, якщо виконується передумова.

2.3 Класифікація CRDT

Відповідно до розглянутих моделей реплікації, за моделями синхронізації CRDT також розподіляються на два класи [3]:

– засновані на передачі операцій (Commutative Replicated Data Type, CmRDT);

– засновані на передачі стану (Convergent Replicated Data Type, CvRDT).

Для проведення успішної реплікації за допомогою CvRDT необхідно виконання наступних умов:

– дані повинні формувати напіврешітку [22];

– функція злиття повинна створювати точну верхню грань (LUB);

– репліки повинні складати пов’язаний граф.

Функція злиття має бути комутативною та ідемпотентною, а також монотонно рости на заданій множині даних, що представлено на рисунку 2.6.

Це гарантує сходження реплік та дозволяє послабити вимоги до протоколу передачі даних, а втрата повідомлення з новим станом є допустимою. Властивість ідемпотентності дозволяє відправляти одне й те саме повідомлення декілька разів, а комутативність – відправляти їх в довільному порядку.

Історично, підхід CvRDT використовується в таких файлових системах як AFS, NFS, Coda, а також у сховищах даних – таких як Riak та Dynamo.

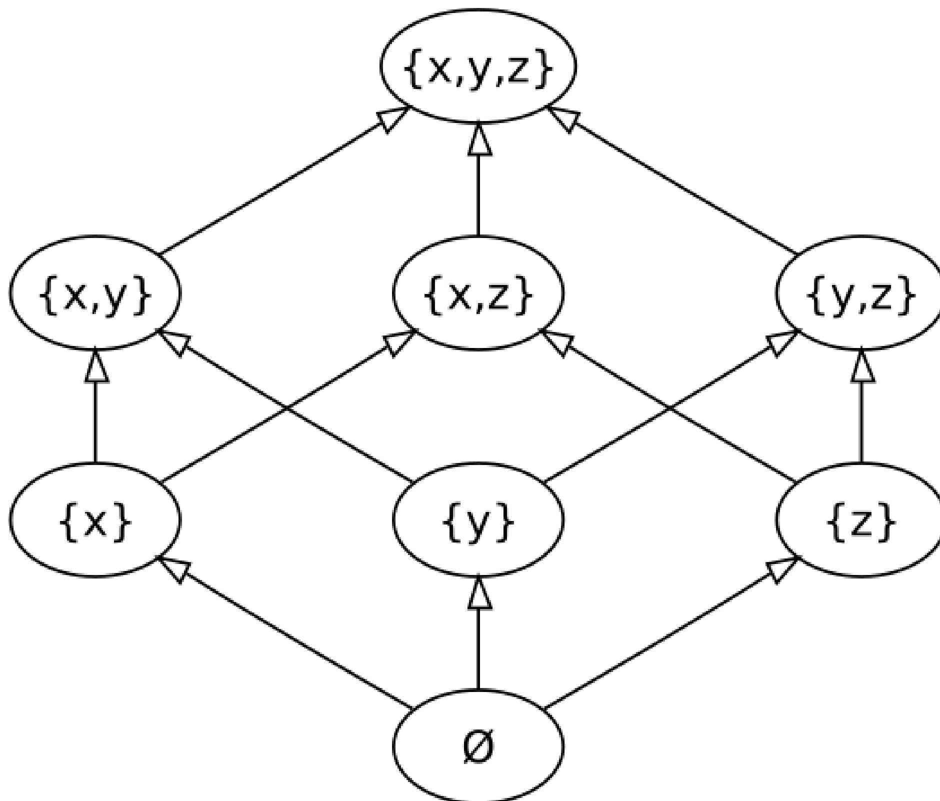

Рисунок 2.6 – Ілюстрація реплікації у CvRDT

CmRDT, на відміну від CvRDT, має високі вимоги до протоколу передачі даних. Репліки обмінюються операціями оновлення стану. На рисунку 2.7 представлено процес обміну операціями на прикладі операції «ADD». Репліка-джерело створює «effector» – замикання-операцію, що виконується спочатку локально на поточній репліці синхронно, а потім на усіх інших репліках асинхронно.

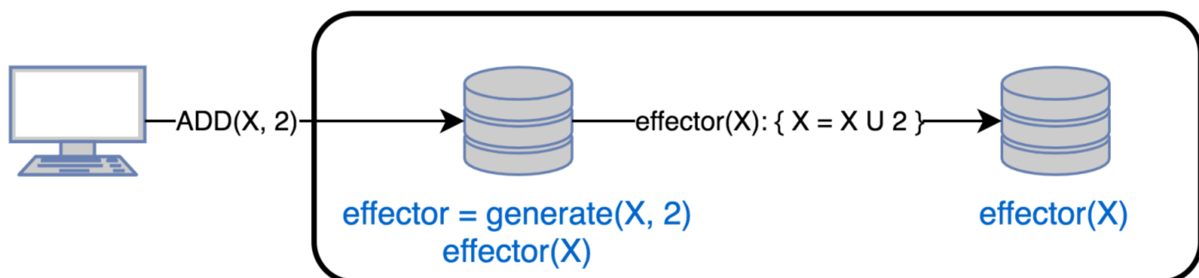

Рисунок 2.7 – Ілюстрація обміну операціями у CmRDT

Однією з проблем, що ускладнює реалізацію CmRDT є необхідність зберігати історію, але, навпаки, вони є набагато більш гнучкими, порівняно з CvRDT. CmRDT використовується в наступних корпоративних системах: Bayou, Rover, IceCube, Telex.

Механізми CvRDT є простішими для реалізації, тому що вся необхідна для реплікації інформація включена у стан об’єктів. Ідемпотентність та комутативність дозволяє знизити вимоги до якості каналу передачі даних та його пропускної здатності. Також дозволяється працювати з невідомим числом реплік.

Одним з недоліків CvRDT є неефективність при реплікації об’єктів великого розміру. Дана проблема вирішується за допомогою використання дельта-мутаторів, що оновлюють стан згідно до попереднього часу синхронізації, передаючи не повний стан, а лише його змінену частину, таким чином наближуючи механізм реплікації до CmRDT [23]. Даний підхід називають дельта-синхронізацією, яка об’єднує підходи активної та пасивної реплікації.

Оптимізацією CmRDT та дельта синхронізації є компресія операцій, що представлена на рисунку 2.8. Даний підхід можна використовувати у системах де допускається певна затримка.

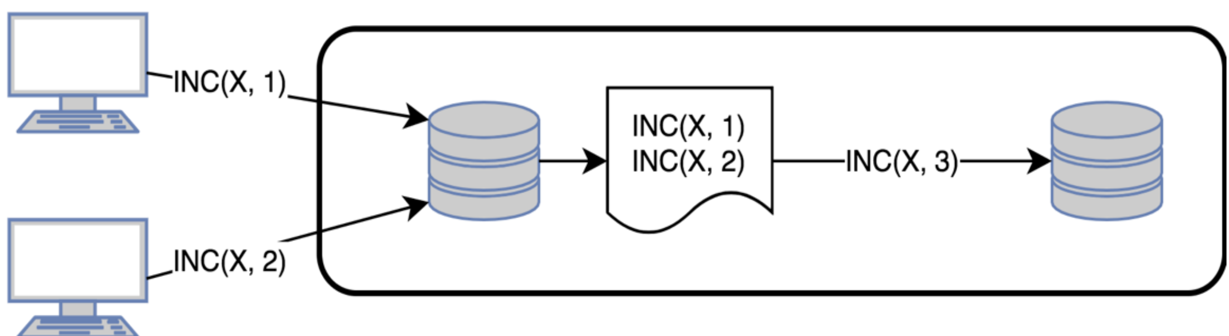

Рисунок 2.8 – Компресія операцій

Для систем, у яких оновлення повинні бути передані негайно без затримок, підхід пасивної синхронізації, що використовується у CvRDT, буде поганим вибором. Передача цілого стану коштує значно дорожче ніж тільки операція оновлення. Дельта-синхронізація підходить краще, але різниця з CvRDT у даному випадку буде незначною [24].

CvRDT та дельта синхронізація є достатньо ефективним вибором, коли необхідно провести синхронізацію репліки після відмови [24]. У випадку, коли CmRDT є безальтернативним варіантом, існують декілька варіантів вирішення даної проблеми:

– послідовно відправити усі пропущені операції з моменту відмови;

– зробити повну копію однієї з реплік та лише після цього відправити пропущені операції.

CmRDT вимагає, щоб оновлення були доставлені кожній з реплік не більше одного разу. Дану вимогу можна проігнорувати тільки при наявності ідемпотентної реалізації функції «effector», але на практиці дану властивість набагато важче реалізувати.

Було доведено, що будь-яку структуру CRDT можна представити у вигляді CmRDT та CvRDT [4]. Тобто CmRDT-структура може бути представлена у вигляді CvRDT-структури та навпаки, таким чином емулюючи одне одного.

Специфікація емуляції об’єкту CmRDT на базі CvRDT представлена на рисунку 2.9 [4].

1: **payload** State-based S
2: **initial** Initial payload
3: **update** State-based-update (operation f, args a) : state s
4: **atSource** (f, a) : s
5: **pre** $S.f$.precondition(a)
6: **let** $s = S.f(a)$
7: **downstream** (s)
8: $S := merge(S, s)$

Рисунок 2.9 – Емуляція CmRDT на базі CvRDT

Механізм емуляції, що представлений на рисунку є достатньо простим. Ігноруючи операцію запиту, об’єкт має єдину операцію оновлення, локально оновлює свій стан та передає операцію злиття як замикання для виконання на інших репліках. В даному випадку, операція злиття є тією самою операцією, що називається «effector».

Можна відзначити, що емульований об’єкт CmRDT є комутативним, тому що операція merge є комутативною у всіх об’єктів CvRDT. Відповідно в даному випадку операції можуть бути отримані у довільному порядку.

Механізм емуляції об’єкту CvRDT на базі CmRDT є більш складним. Загальна специфікація емульованого об’єкту представлена на рисунку 2.10 [4]. Виклик операції оновлення додає її до множини M повідомлень, які необхідно доставити. Операція злиття представляє собою операцію об’єднання двох наборів повідомлень. Щоб уникнути повторної доставки операції оновлення, доставлені повідомлення зберігаються у множині D.

1: **payload** Operation-based P, set M, set D
2:     **initial** Initial state of payload, $\varnothing, \varnothing$
3: **update** op-based-update (update f, args a) : returns
4:     **pre** $P.f$.atSource.pre(a)
5:     **let** returns $= P.f$.atSource(a)
6:     **let** $u = unique()$
7:     $M := M \cup \{(f, a, u)\}$
8:     deliver()
9: **update** deliver ()
10:     **for** $(f, a, u) \in (M \setminus D) : f$.downstream.pre$(a)$ **do**
11:         $P := P.f$.downstream(a)
12:         $D := D \cup \{(f, a, u)\}$
13: **compare** (R, R') : boolean b
14:     **let** $b = R.M \leq R'.M \vee R.D \leq R'.D$
15: **merge** (R, R') : payload R''
16:     **let** $R''.M = R.M \cup R'.M$
17:     R''.deliver()

Рисунок 2.10 – Емуляція CvRDT на базі CmRDT

Стани об’єкта емуляції утворюють монотонну напіврешітку. Виклик або доставка операції додає її до відповідного набору повідомлень, а отже, змінює стан у частковому порядку. Операція злиття визначається як об'єднання M множин, і, таким чином, є LUB-операцією, утворюючи точну верхню грань. При цьому множина M є тотожною причинно-наслідковій історії репліки, а неконкурентні оновлення відображаються в M у причинному порядку [4].

2.4 Структури CRDT

Структури сучасних безконфліктних реплікованих типів даних, як правило, є композицією раніше відомих базових типів, що описані у наукових роботах, реалізовані у деяких бібліотеках та можуть бути використані при побудові розподілених систем [4].

Можна виділити наступні існуючі базові типи [25]:

- G-Counter;
- RN-Counter;
- LWW-Register;
- MV-Register;
- G-Set;
- 2P-Set;
- PN-Set;
- LWW-Set;
- OR-Set.

Отже, існують прості типи (лічильники та регістри), колекції та інші типи з комплексними вимогами. Серед комплексних типів виділяють графи та послідовності.

2.4.1 Лічильники

Лічильник є реплікованим типом даних, що підтримує операції інкременту та декременту над цілим числом. Згідно семантики, величина повинна збігатися до загальної кількості операцій інкременту мінус кількості операцій декременту. Лічильник може бути корисним багатьом застосункам, наприклад для підрахунку кількості авторизованих у даний момент користувачів. Незважаючи на свою простоту, лічильник розкриває деякі проблеми дизайну структур CRDT.

Аналогічно багатьом іншим типам, лічильним може бути представлений як у вигляді CvRDT, так і у вигляді CmRDT.

Реалізація, що базується на передачі операцій є очевидною – операція-замикання просто додає або віднімає від числа одиницю локально на кожній з реплік. Операції додавання та віднімання є комутативними. Специфікація лічильнику CmRDT представлена на рисунку 2.11 [4].

Незважаючи на простоту реалізації CmRDT, реалізація лічильника, що є CvRDT не настільки очевидна. Специфікація об'єкту зростаючого (G-Counter) лічильника у CvRDT представлена на рисунку 2.12 [4].

```
1:  payload integer i
2:      initial 0
3:  query value () : integer j
4:      let j = i
5:  update increment ()
6:      downstream ()
7:          i := i + 1
8:  update decrement ()
9:      downstream ()
10:         i := i − 1
```

Рисунок 2.11 – Специфікація лічильнику у CmRDT

Рішенням є зберігання стану об'єкта у вигляді вектору з розміром, що дорівнює кількості реплік. Даний вектор називається вектором версій [24].

Кожна репліка асоціюється з відповідним індексом у векторі та збільшує значення лише в позиції зі своїм індексом.

1: **payload** integer$[n]$ P
2: **initial** $[0, 0, \ldots, 0]$
3: **update** $increment$ $()$
4: **let** $g = myID()$
5: $P[g] := P[g] + 1$
6: **query** $value$ $()$: integer v
7: **let** $v = \sum_i P[i]$
8: **compare** (X, Y) : boolean b
9: **let** $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
10: **merge** (X, Y) : payload Z
11: **let** $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$

Рисунок 2.12 – Специфікація лічильнику у CvRDT

Функція злиття бере максимальне значення у кожній позиції, а підсумковим значенням є сума усіх елементів вектору. Дана реалізація об’єкту має декілька важливих припущень:

– стан об’єкту (числа, що знаходяться у векторі) не переповнюється;

– кількість реплік є відомою.

Підтримка операції декременту реалізована у структурі RN-Counter, специфікація якого представлена на рисунку 2.13 [4].

Різниця між G-Counter та RN-Counter полягає у додатковому векторі версій. Одного вектору версій недостатньо, тому що операція декременту порушує монотонність напіврешітки, а оскільки функція злиття є операцією максимуму – декремент не матиме ефекту. Виходячи з цього, RN-Counter має по одному вектору версій для інкременту та декременту.

RN-лічильник може бути корисним, наприклад, для підрахунку кількості користувачів, які є онлайн у P2P-застосунку, такому як Telegram. Через асинхронність та затримку значення лічильника може тимчасово бути не

актуальним та відрізнятися від його справжнього значення, але буде точним у кінцевому рахунку.

1: **payload** integer$[n]$ P, integer$[n]$ N
2: **initial** $[0,0,\ldots,0]$, $[0,0,\ldots,0]$
3: **update** $increment$ ()
4: **let** $g = myID()$
5: $P[g] := P[g] + 1$
6: **update** $decrement$ ()
7: **let** $g = myID()$
8: $N[g] := N[g] + 1$
9: **query** $value$ () : integer v
10: **let** $v = \sum_i P[i] - \sum_i N[i]$
11: **compare** (X, Y) : boolean b
12: **let** $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \wedge \forall i \in [0, n-1] : X.N[i] \leq Y.N[i])$
13: **merge** (X, Y) : payload Z
14: **let** $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$
15: **let** $\forall i \in [0, n-1] : Z.N[i] = \max(X.N[i], Y.N[i])$

Рисунок 2.13 – Специфікація RN-Counter

Існують також інші види лічильників, що мають більш складну структуру, наприклад не негативний лічильник [4].

2.4.2 Регістри

Регістр є коміркою пам’яті, що зберігає атом або об’єкт. Таким чином він виконує роль контейнеру та обгортки. Регістр підтримує лише 2 операції:

- assign (запис);
- value (читання).

Основною проблемою є те, що операція запису не комутативна. Існують 2 найбільш поширені типи регістрів, що певним чином узгоджують запис даних:

- регістр LWW (last writer wins);
- регістр з декількома значеннями (multi-value register або MV-register).

LWW-регістр дозволяє упорядкувати оновлення шляхом присвоєння часової позначки до кожного оновлення. Дані часові позначки вважаються унікальними, повністю впорядкованими і такими, що відповідають причинному порядку. Якщо оновлення номер 1 сталося перед оновленням номер 2, то часова позначка другого оновлення повинна бути більшою ніж у першого. Загальна специфікація LWW-регістру у вигляді CmRDT представлена на рисунку 2.14 [4].

payload $X\ x$, $\mathrm{timestamp}\ t$
    initial $\perp, 0$
query $value\ ()\ :\ X\ w$
    let $w = x$
update $assign\ (X\ x')$
    atSource () t'
        let $t' = now()$
    downstream (x', t')
        $\mathbf{if}\ t < t'\ \mathbf{then}\ x, t := x', t'$

Рисунок 2.14 – Специфікація LWW-регістру для CmRDT

Виходячи з наведеного рисунку видно, що стан об’єкту перезаписується тільки у випадку, коли реплікою було отримано дані з більшою часовою позначкою. Для даного типу регістру реалізація CmRDT та CvRDT практично не відрізняється. Механізм реплікації з використанням подібного регістру, що є CvRDT представлений на рисунку 2.15 [4].

LWW-регістри застосовуються у файловій системі NFS, у якості стовбців СУБД Cassandra та багатьох інших видах програмного забезпечення.

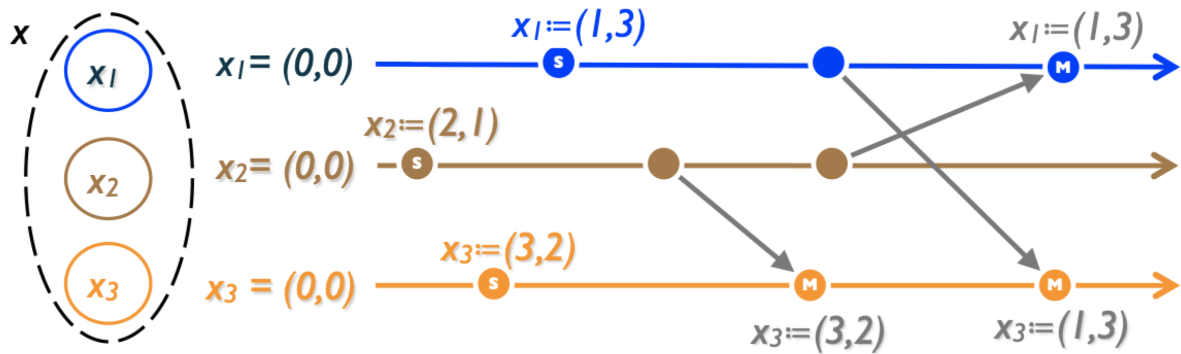

Рисунок 2.15 – Реплікація з використанням LWW-регістру та CvRDT

Як було попередньо відзначено, іншою альтернативою є MV-регістр. Він є схожим одночасно на G-лічильник та LWW-регістр через використання як часової позначки, так і вектору версій для кожного елементу набору значень регістру. Операція злиття реалізує об'єднання елементів з вхідних наборів, за виключенням тих елементів, чия часова мітка у відповідного елемента є меншою, порівняно з іншою реплікою. Механізм реплікації з використанням MV-регістру представлений на рисунку 2.16 [4].

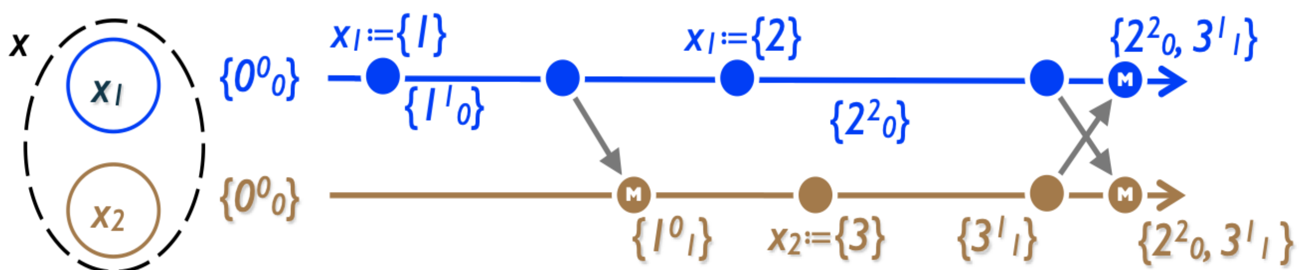

Рисунок 2.16 – Реплікація з використанням MV-регістру та CvRDT

Подібний тип регістрів використовується для зберігання товарів у кошику інтернет-магазину, наприклад Amazon. Його використання є дуже актуальним у

випадку, коли користувачу потрібно мати можливість редагувати кошик у режимі офлайн, а також більше ніж на одному пристрої.

2.4.3 Множини

Множини – це базовий тип для побудови інших контейнерів, відображень та графів. Функція злиття реалізується як операція об'єднання елементів двох множин. Операціями, що підтримуються є додавання та видалення елементу, які не є комутативними, а тому реалізувавши множину на базі операцій, репліки не будуть сходитися.

Найпростішим рішенням є заборона операції видалення – залишається тільки операція додавання, яка є комутативною. Відповідно подібна множина стає CvRDT, наприклад G-Set, специфікація якого представлена на рисунку 2.17 [4].

1: **payload** set A
2: **initial** $\varnothing$
3: **update** add (element e)
4: $A := A \cup \{e\}$
5: **query** $lookup$ (element e) : boolean b
6: **let** $b = (e \in A)$
7: **compare** (S, T) : boolean b
8: **let** $b = (S.A \subseteq T.A)$
9: **merge** (S, T) : payload U
10: **let** $U.A = S.A \cup T.A$

Рисунок 2.17 – Специфікація G-Set

Двофазова множина (2P-Set) має підтримку обох операцій: додавання та видалення. Специфікація у вигляді CvRDT представлена на рисунку 2.18 [4].

1: **payload** set A, set R
2: **initial** $\varnothing, \varnothing$
3: **query** $lookup$ (element e) : boolean b
4: **let** $b = (e \in A \wedge e \notin R)$
5: **update** add (element e)
6: $A := A \cup \{e\}$
7: **update** $remove$ (element e)
8: **pre** $lookup(e)$
9: $R := R \cup \{e\}$
10: **compare** (S, T) : boolean b
11: **let** $b = (S.A \subseteq T.A \vee S.R \subseteq T.R)$
12: **merge** (S, T) : payload U
13: **let** $U.A = S.A \cup T.A$
14: **let** $U.R = S.R \cup T.R$

Рисунок 2.18 – Специфікація 2P-Set

Особливістю даного типу є те, що елемент не може бути доданий назад після того, як був видалений. З метою уникнення аномалій видалення дозволяється тільки, якщо елемент вже був доданий до множини. Для множини видалених елементів використовується окремий G-Set, що називається «tombstone set» в даному випадку.

Наступною альтернативою є LWW-Set, у якому, аналогічно LWW-регістру, використовується генерація унікальних часових позначок для кожного елемента. Приклад даного підходу представлений на рисунку 2.19 [4].

LWW-Set використовує 2 G-Set, як і 2P-Set, та при перевірці наявності елементу в множині перевіряє, де часова позначка більше – в підмножині доданих чи видалених елементів.

PN-Set відрізняється відсутністю розбиття структури на множину доданих та множину видалених елементів. Замість цього з кожним елементом асоціюється лічильник, який інкрементується при додаванні та декрементується при відніманні відповідно, як представлено на рисунку 2.20 [4].

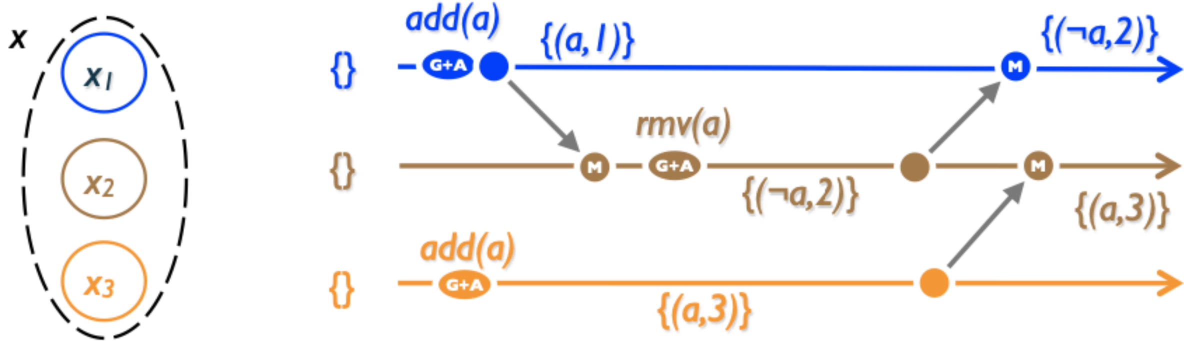

Рисунок 2.19 – Реплікація з використанням LWW-set

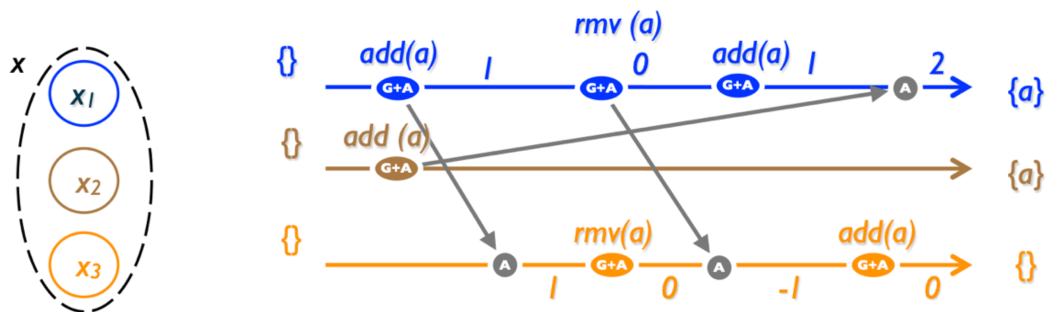

Рисунок 2.20 – Реплікація з використанням RN-Set

В представленому випадку, операція додавання не призводить до появи елементу у репліці, тому що його лічильник залишається рівним нулю.

OR-Set має іншу унікальну реалізацію – з кожним доданим елементом асоціюється унікальний тег (відносно елемента, а не множини). Операція видалення видаляє елемент та розсилає іншим реплікам на видалення усі пари елементів з пов’язаним тегом, як представлено на рисунку 2.21 [4].

В OR-Set операція додавання має пріоритет над операцією видалення. Якщо це не задовольняє вимогам, то можна скористатись RW-Set, який має пріоритет видалення над додаванням.

Існують також інші види множин, наприклад U-Set, але вони не застосовуються настільки часто як наведені у даному підрозділі.

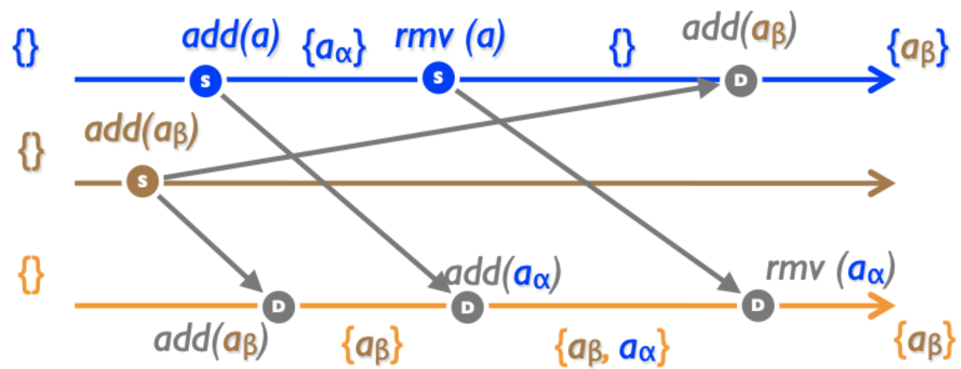

Рисунок 2.21 – Реплікація з використанням OR-Set

2.5 Робота з текстом

Простих структур CRDT недостатньо для роботи з довільними даними. Тому велика кількість досліджень присвячена пошуку нових способів реалізації послідовних CRDT, зокрема для роботи з текстом.

Існує велика кількість алгоритмів, але, як правило, використовують наступний підхід – кожній окремій букві надається свій унікальний ідентифікатор. Додавання кожної літери робиться за допомогою посилання на її сусіда замість того, щоб працювати з індексами. Після видалення, літери замінюються на пустий заповнювач, таким чином зберігаючи цілісність даних, що дозволяє декільком реплікам посилатися на символ або видаляти його одночасно без створення конфліктів злиття. Даний підхід передбачає, що розмір структури постійно зростає пропорційно кількості видалених елементів, але існують різні способи збірки сміття, що вирішують дану проблему [4].

Для роботи з текстом практично завжди використовують CvRDT, тому що використання CmRDT робить процес більш схожим на Operational Transformation з відповідними для цього наслідками та вимогами до системи.

Відомими послідовними CRDT є WOOT, Treedoc, Logoot, LSEQ та RGA, кожен з яких має свої недоліки.

WOOT є однією з перших реалізацій і надає кожному символу посилання на двох сусідів з обох боків. Як показали дослідження, даний підхід є менш ефективним, порівняно з новими реалізаціями [3]. Treedoc також має подібні проблеми, пов’язані з продуктивністю.

Структури Logoot на LSEQ уникають традиційного використання заповнювачів для видалених даних, розглядаючи кожен символ як унікальну точку впродовж щільної координатної площини. Але їх алгоритми мають доведені проблеми під час одночасного редагування тексту.

Найбільш актуальною реалізацією є RGA. Кожен символ посилається на свого крайнього лівого сусіда та має додаткову хеш-таблицю для покращення продуктивності пошуку символів. RGA містить також визначену операцію оновлення в додаток до звичайних операцій вставки та видалення.

2.3 Висновки по розділу 2

У другому розділі було проведено огляд та аналіз існуючих структур, починаючи від базових понять атому та об’єкту, лічильників, регістрів, їх особливостей, та закінчуючи множинами та графами, адже сучасні CRDT представляють собою композицію існуючих більш простих структур. Безконфліктні репліковані типи даних вирішують проблему узгодженості даних, використовуючи модель суворої узгодженості у кінцевому рахунку. Вони поділяються на 2 категорії: CvRDT (засновані на передачі стану) та CmRDT (засновані на передачі операцій). CmRDT є більш гнучким рішенням, але потребує реалізації більш складних алгоритмів ніж CvRDT. В свою чергу, CvRDT має спрощені вимоги до протоколу комунікації між репліками, що є великою перевагою.

3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ СТРУКТУР БЕЗКОНФЛІКТНИХ РЕПЛІКОВАНИХ ТИПІВ ДАНИХ В КОЛАБОРАТИВНИХ ОФЛАЙН ЗАСТОСУНКАХ

3.1 Проектування програмної системи

В рамках аналізу предметної галузі була поставлена задача реалізації мобільного застосунку з підтримкою колаборативної роботи з текстовим документом та програмного модуля, що включає структури даних CRDT для роботи з текстовими документами.

Першим етапом проектування програмної системи є розробка архітектури програмного забезпечення. Для представлення компонентів системи було застосовано мову UML, що є невід'ємною частиною процесу розробки програмного забезпечення. UML дозволяє використовувати графічні зображення з метою створення абстрактної моделі системи, яку називають UML-моделлю [26]. Практично усі CASE-засоби мають підтримку UML. Моделі, що розроблені в UML, дозволяють значно спростити процес кодування і дозволити зосередитися безпосередньо над реалізацією системи.

Як було зазначено раніше, більшість сучасних рішень для колаборативного редагування документів зводиться до використання моделі суворої кінцевої узгодженості. Даний підхід робить застосунок-клієнт достатньо складним, але це рішення є єдиним компромісом для реалізації поставленої задачі.

Використання інструменту типу «serverless» для централізованого зберігання даних задовольняє усім вимогам до програмної системи. Для операційної системи iOS таким інструментом є CloudKit. Виходячи з того, що алгоритми синхронізації реалізуються на клієнті, основною задачею сервера є лише зберігання даних.

Отже, система повинна складатися з товстого клієнту та серверу бази даних, що розсилає оновлення. Діаграма розгортання програмного продукту, яка ілюструє дане твердження представлена на рисунку 3.1.

iCloud має централізоване сховище даних та достатньо простий інструмент CloudKit для забезпечення обміну документами у застосунку.

Фреймворк дозволяє ефективно зберігати, завантажувати дані та слухати їх нові зміни.

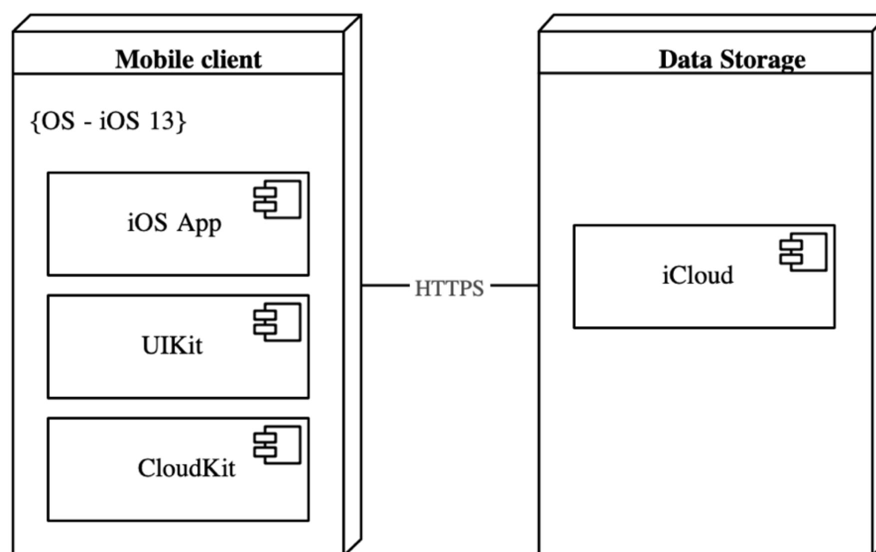

Рисунок 3.1 – Діаграма розгортання програмного продукту

Більшість роботи, пов'язаної з передачею даних, виконується у фоновому режимі [27]. А, виходячи з того, що Apple не дозволяє запускати сторонній код на своїх серверах, конфлікти злиття повинні вирішуватись локально.

Другим етапом проектування є реалізація протоколу взаємодії мобільного додатку зі сховищем даних. Протокол завантаження даних з iCloud представлений на рисунку 3.2 за допомогою діаграми послідовностей.

Першим кроком для отримання даних є авторизація користувача у iCloud. Перевіривши коректність статусу аккаунта, виконується операція створення зони у сховищі бази даних. Зона є окремим розділом приватного сховища даних

та може бути створена під час виконання програмного забезпечення. Отриманий об’єкт CKRecordZone використовується для створення підписки на події, що пов’язані зі змінами контенту бази даних.

Підписка на події представлена об’єктом CKSubscription, що повинен бути збереженій у контейнері сховища даних – CKDatabase.

Кожен файл є об’єктом CKRecord, а його дані представлені об’єктом CKAsset.

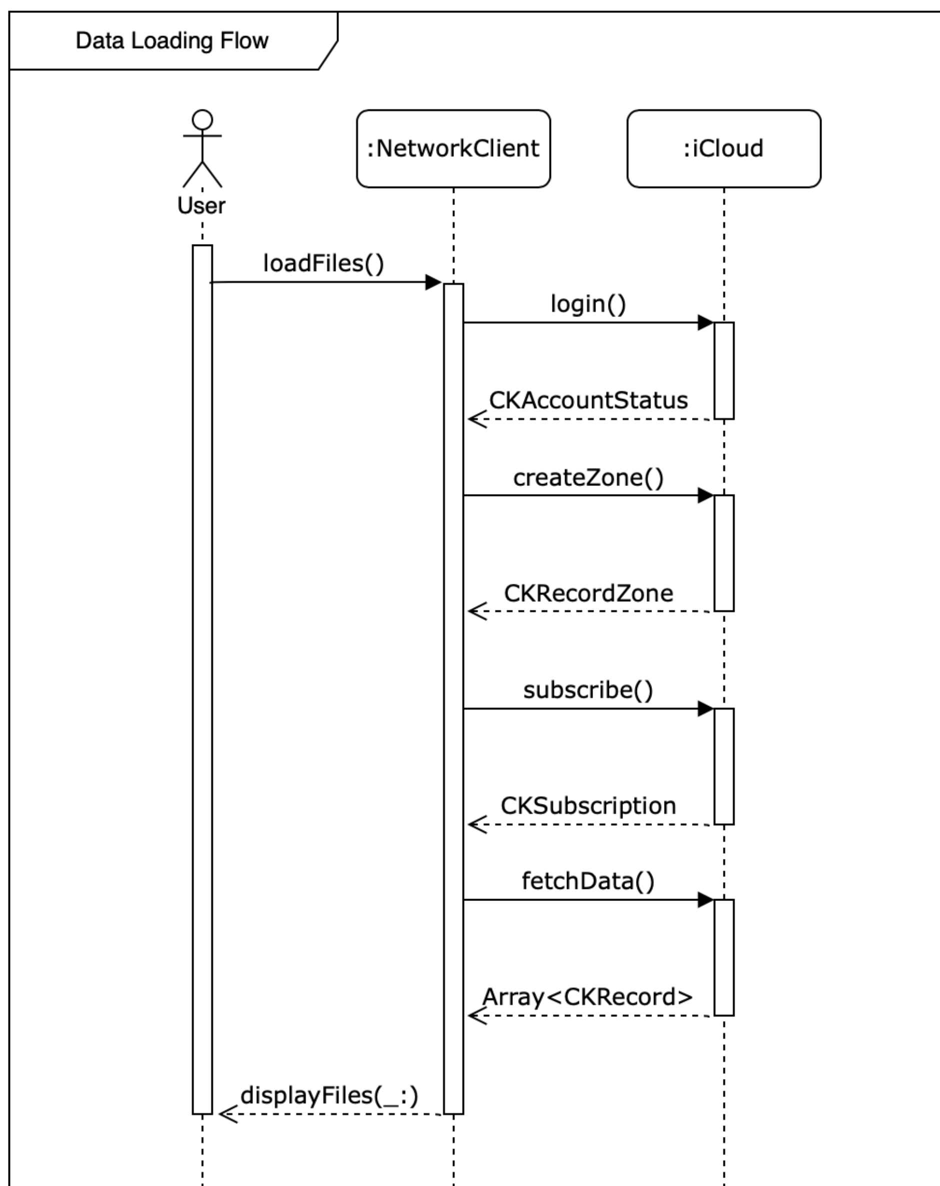

Рисунок 3.2 – Протокол завантаження даних з iCloud

Таким чином, після завантаження даних мобільний клієнт повинен лише прослуховувати нові події та виконувати операції злиття локальних даних з тими, що були отримані з мережі.

Третім етапом проектування є вибір структур безконфліктних реплікованих типів даних для реалізації колаборативного редагування текстових документів. Як було зазначено раніше, найбільш актуальною реалізацією послідовних CRDT для роботи з текстом є RGA. Було вирішено порівняти та реалізувати 2 підходи:

– підхід RGA на базі суміжного масиву;
– підхід RGA на базі причинних дерев.

Дані алгоритми та структури даних буде розглянуто в підрозділі 3.4.

Заключним етапом є проектування моделі зв’язку між інтерфейсом користувача та моделлю даних CRDT. Для редагування тексту необхідно використовувати компонент UITextView. Текстові дані, що відображаються для користувача повинні бути інкапсульовані у клас, що реалізує інтерфейс NSTextStorage. Зв’язок між компонентами представлений у діаграмі класів на рисунку 3.3.

CRDTTextStorage є реалізацією інтерфейсу NSTextStorage і таким чином виконує роль адаптера для інтерфейсів моделей CRDT та інтерфейсу користувача.

На даному етапі процес проектування системи можна вважати завершеним та переходити до деталей стеку використаних технологій та реалізації програмного продукту.

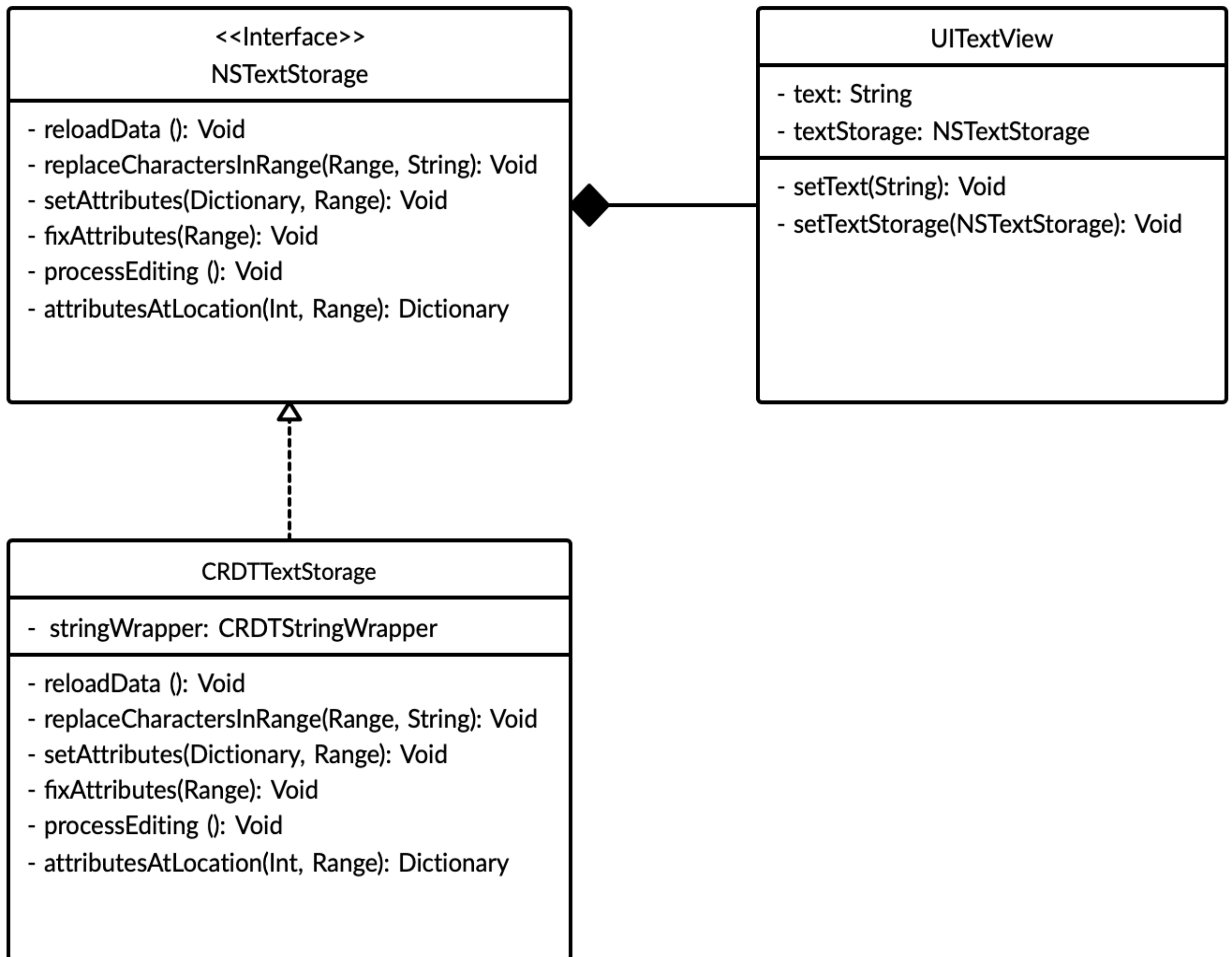

Рисунок 3.3 – Модель зв’язку між моделлю CRDT та інтерфейсом користувача

3.2 Технології для реалізації програмного забезпечення

Для реалізації програмного продукту було прийнято рішення використати мову програмування Swift та платформу iOS.

Swift – це мультипарадигмова мова програмування для платформ iOS, macOS, tvOS, watchOS та Linux [28]. Swift є швидким, як багато інших компільованих мов програмування та водночас дуже простим і виразним як скриптові мови. Підхід протокол-орієнтованого програмування дозволяє будувати чіткі абстракції в узагальненому виді згідно з вимогами SOLID, DRY,

KISS [29]. Механізм виведення типів та шаблони мають сучасний легкий синтаксис. Окремо можна виділити механізм роботи з пам'яттю – Swift не має збирача сміття, а використовує систему ARC для більш економного використання ресурсів [30].

Для розробки інтерфейсу користувача було обрано UIKit, який є безальтернативним інструментом для платформи iOS.

У якості менеджера пакетів було обрано Swift Package Manager. Однією з його переваг порівняно з іншими інструментами є те, що модулі, які адаптували Swift Package Manager підтримуються плагіном GitHub Actions, що в майбутньому може суттєво зменшити навантаження на власні CI сервери.

У якості сховища даних було обрано iCloud та фреймворк CloudKit для роботи з ним. CloudKit включений до стандартної бібліотеки компонентів iOS SDK, а тому є достатньо надійним та протестованим інструментом. Перевагою використання CloudKit є те, що процес синхронізації даних між клієнтом та сервером бази даних вже реалізований у SDK, а відповідно виникає необхідність сконцентруватись лише на вирішення конфліктів злиття даних.

3.3 Опис програмної реалізації мобільного додатку

В ході магістерської дипломної роботи було розроблено мобільний додаток для операційної системи iOS 13, який дозволяє користувачу працювати з текстовими документами у режимі колаборативного редагування.

Перед початком роботи було необхідно зареєструвати аккаунт розробника Apple та провести його конфігурацію. На рисунку 3.4 представлена конфігурація App ID мобільного додатка.

Рисунок 3.4 – Конфігурація App ID мобільного додатка

Стартовим екраном мобільного додатка є екран перегляду існуючих документів, який представлений на рисунку 3.5.

Авторизація відбувається у фоновому режимі. Екран авторизації не потрібен, тому що CloudKit завжди використовує системний аккаунт користувача, що вказаний у системних налаштуваннях операційної системи.

Натиснувши на файлову назву або на знак «+», користувач відкриває екран колаборативного редагування текстового документа, який представлений на рисунку 3.6. Інтерфейс даного екрану містить:

– текстовий редактор;

– кнопку відкриття доступу до документу для інших користувачів.

Синхронізація тексту також відбувається у фоновому режимі з часовим інтервалом у декілька секунд.

Також синхронізуються позиції курсорів, що додає більшої інтерактивності мобільному додатку.

Важливою частиною елементу текстового редактора є реалізація сховища даних для компоненту UITextView. Далі наведено фрагмент реалізації дочірнього класу NSTextStorage. Зв’язок між даними компонентами був представлений в діаграмі класів на рисунку 3.3.

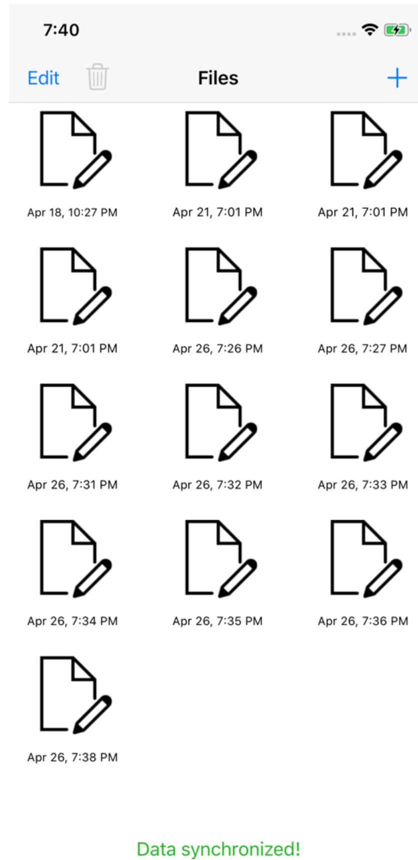

Рисунок 3.5 – Екран перегляду існуючих документів

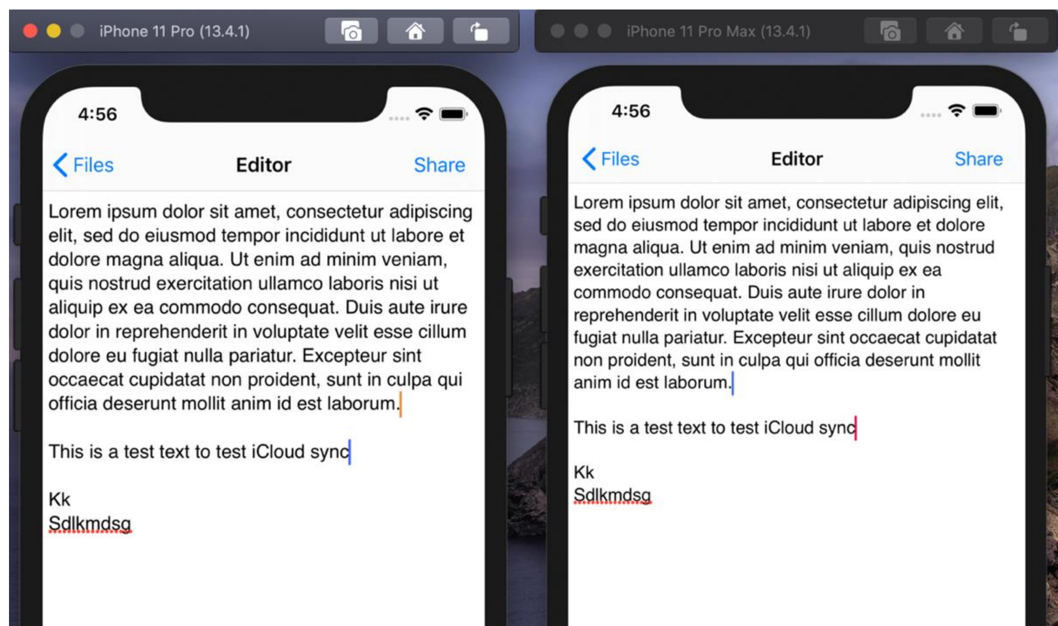

Рисунок 3.6 – Екран колаборативного редагування текстового документа

Особливістю CloudKit є розділення бази даних на публічну та приватну частини, що проілюстровано на рисунку 3.7.

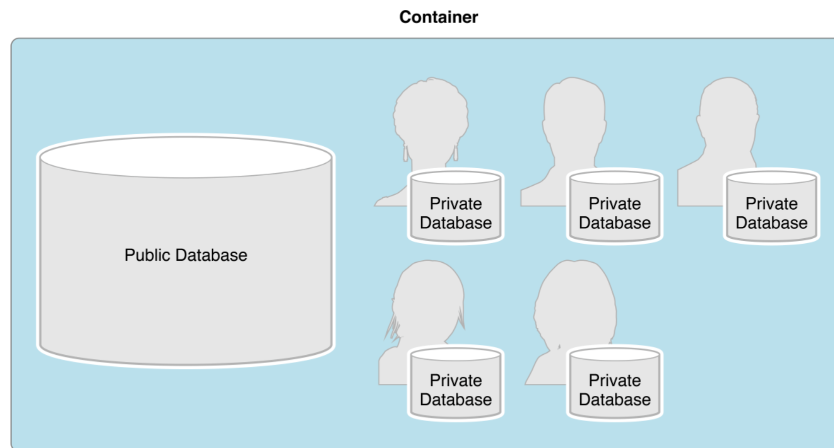

Рисунок 3.7 – Структура бази даних iCloud

Публічна база даних використовується за замовчуванням та є доступною для усіх користувачів додатку. Але для зберігання документів використовується приватна база даних, що є індивідуальною для кожного користувача. Користувач може відкрити доступ до певного документу для інших користувачів, натиснувши кнопку «Share». Інтерфейс відкриття доступу до документа представлений на рисунку 3.8.

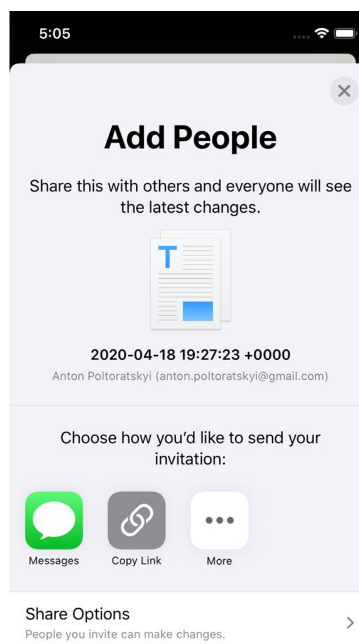

Рисунок 3.8 – Інтерфейс відкриття доступу до документа

3.4 Опис реалізації CRDT структур для роботи з текстом

В даному підрозділі буде розглянуто способи вирішення проблем причинного зв'язку, а також способи реалізації CRDT структури за допомогою послідовних масивів даних та причинних дерев з оглядом на алгоритми RGA.

3.4.1 Опис вирішення проблеми збіжності та досягнення загального порядку

Для початку, незважаючи на те, що для реалізації було обрано тип структур CvRDT, потрібно відзначити, що залишається необхідність знати про тип події, який пов'язаний із кожним символом у тексті: вставка або видалення.

Для вирішення даної проблеми було прийнято рішення вести журнал подій, за яким в подальшому текст може бути реконструйований. В даному підході присутня ідемпотентна, асоціативна та комутативна функція злиття, а дані представлені у якості упорядкованої колекції операцій. Отже, подібну структуру даних можна назвати комбінацією CvRDT та CmRDT.

За замовчуванням, операції впорядковані лише у частковому порядку в рамках одної репліки, що є проблемою при одночасному редагуванні, приклад якого представлений на рисунку 3.9.

Виходячи з рисунку, є 3 репліки: R1, R2, R3. Спочатку користувач репліки R1 пише текст «ASD», після чого зміни розсилаються на репліки R2, R3, а користувач на R1 продовжує робити зміни. Користувачі R2 та R3 одночасно мутують текст та відправляють результат до R1.

З метою досягнення загального порядку кожна операція асоціюється не тільки з часовою міткою Лампорта, а й з унікальним ідентифікатором репліки – UUID. Часова мітка використовується для сортування елементів тексту у причинному порядку, а UUID допомагає вирішити колізію між одночасними операціями, які мають однакові часові мітки.

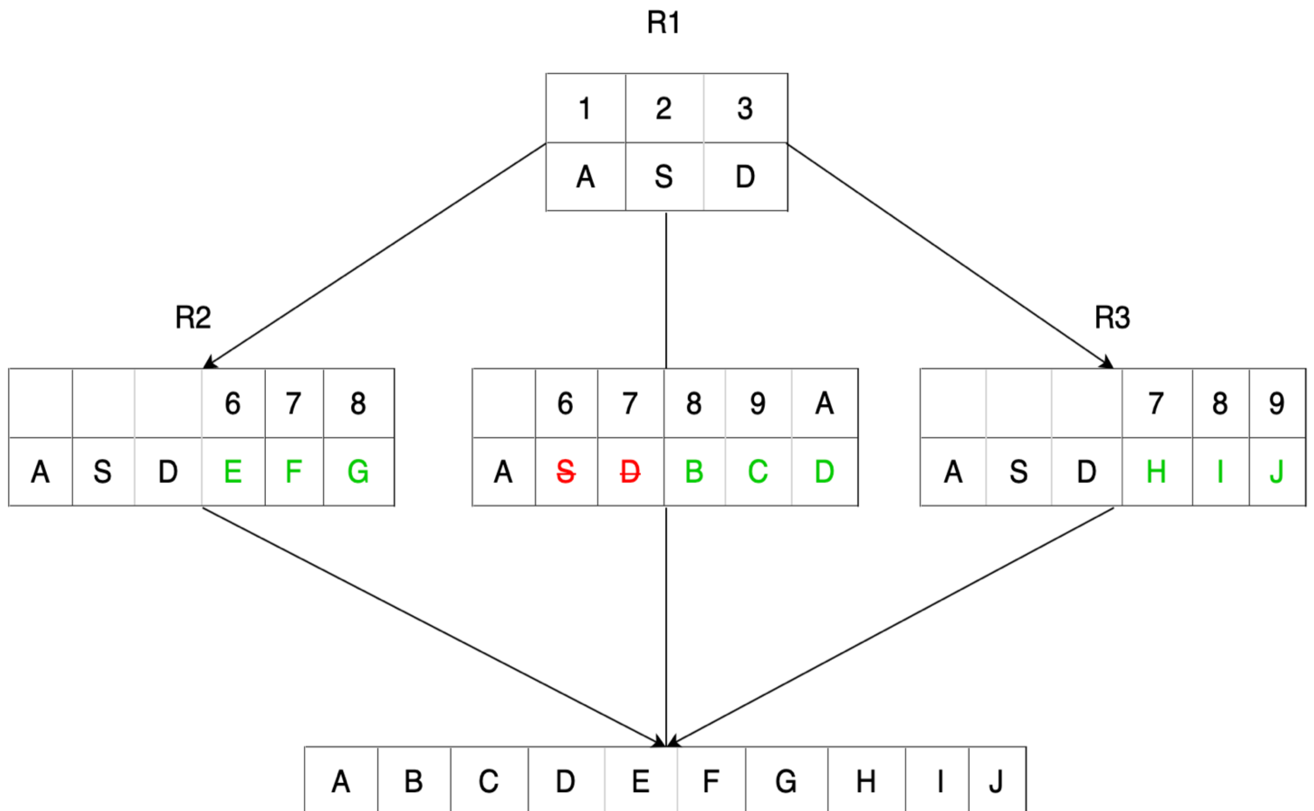

Рисунок 3.9 – Ілюстрація одночасного редагування

Відповідно, в очевидній реалізації операція включає в себе наступні дані:

– текстовий символ (опціонально);
– тип операції (вставка або видалення);
– індекс у тексті, за яким виконується операція;
– UUID репліки;
– часова мітка.

Функція злиття представляє собою просте сортування злиттям, відповідно приймаючи 2 відсортованих масиви операцій, що на виході зливаються в один єдиний. На рисунку 3.10 представлений результат злиття операцій, описаних вище на рисунку 3.9.

Рисунок 3.10 – Результат злиття

Отже, результат є збіжною CRDT-структурою, але має певні проблеми:

– алгоритм реконструкції тексту має квадратичну складність;

– результат реконструкції не є інтуїтивним для користувача.

Перша проблема очевидна та потребує покращення алгоритму.

Друга проблема пов’язана зі структурою операцій – використання індексів при одночасному редагуванні робить їх невалідними, та може привести до виходу за межі масиву.

3.4.2 Опис реалізації підходу RGA на базі суміжного масиву

Враховуючи те, що операції повинні бути незмінними, індекси не можуть бути будь-яким чином адаптовані перед застосуванням операції, як це робиться в Operational Transformation. Тому єдиним правильним рішенням для CRDT структур є відмова від використання індексів у метаданих операції.

Альтернативою є підхід, що реалізований у RGA, у якому кожен наступний символ посилається на попередній за його ідентифікатором замість індексу. Ідентифікатор є відомою комбінацією UUID репліки та часової мітки. Даний підхід наведений у формулі 3.1 на прикладі операції вставки [3].

$$insert\ A^{UUID+timestamp}\ after\ B^{UUID+timestamp}, \qquad (3.1)$$

де A, B – це символи тексту разом з асоційованим UUID та часовою міткою

Оскільки видалені символи зберігаються як заповнювачі, дані посилання на ідентифікатори будуть завжди коректними.

Отримана структура даних схожа на зв’язний список. Результат злиття представлено на рисунку 3.11, де чітко простежуються залежності елементів від крайнього лівого сусіда.

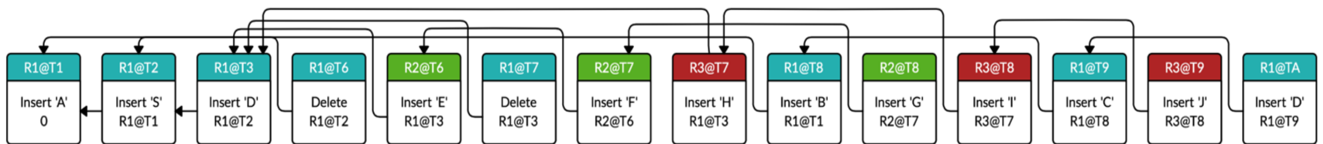

Рисунок 3.11 – Результат злиття операцій, пов'язаних ідентифікаторами

Проблема зміщення індексів вже не є актуальною, а тому результат реконструкції є інтуїтивним для користувача та становить «ABCDEFGHIJ», як і очікувалось. Але не вирішеною є проблема складності алгоритму реконструкції, яка в даній реалізації має також квадратичну складність. Операції вставки та видалення мають константну складність.

3.4.3 Опис реалізації підходу RGA на базі причинних дерев

Асимптотичну складність алгоритму реконструкції можна покращити, якщо робити вставку нової операції не в кінець масиву, а поруч з символом, на який йде посилання, що представлено на рисунку 3.12 [3].

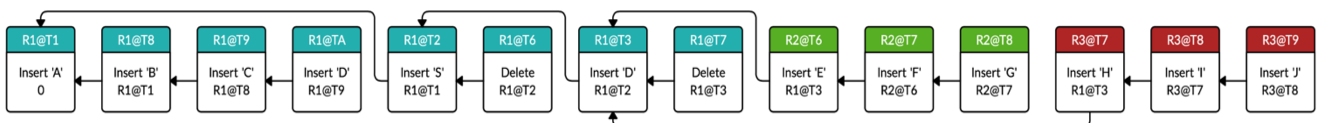

Рисунок 3.12 – Результат злиття після оптимізації зберігання даних

Виходячи з рисунку 3.12, можна зробити висновок, що, позиції символів у реконструйованому тексті є суворо детермінованими. Часова мітка не впливає на позицію символу при одночасному редагуванні, тому що кожен символ суворо пов’язаний зі своїм лівим сусідом.

Дана структура даних може бути представлена у вигляді дерева операцій, як показано на рисунку 3.13.

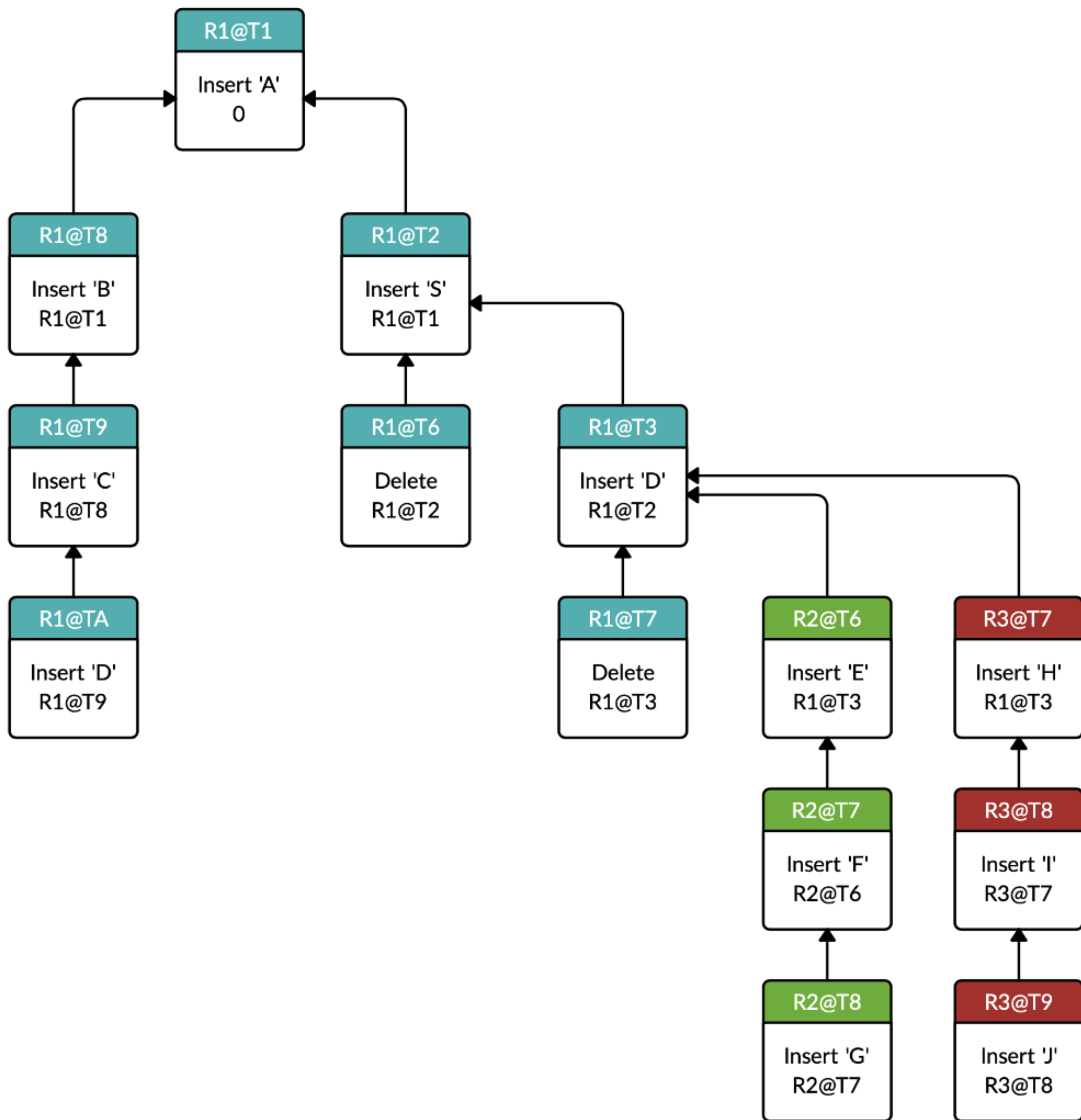

Рисунок 3.13 – CRDT структура у вигляді причинного дерева

Кожному одночасному редагуванню відповідають суміжні гілки, які разом зі своїми елементами відсортовані в порядку зростання UUID репліки та часової мітки. Подібні структури даних називаються причинними деревами [3].

Кожен елемент у причинному дереві є екземпляром узагальненого типу Operation, що представлений на рисунку 3.14.

```
struct Operation<Value: Codable>: Codable {
    struct Id: Codable, Hashable {
        let instance: UUID
        let timestamp: Int
    }
    let id: Id
    let cause: Id
    let value: Value
}

typealias StringOperation = Operation<StringValue>
```

Рисунок 3.14 – Реалізація структури Operation

Кожна операція має свій ідентифікатор, де «instance» – це UUID репліки, а «timestamp» є часовою міткою Лампорта. Поле «cause», в свою чергу, є ідентифікатором батьківського елемента у дереві.

Для роботи з текстом був визначений окремий тип StringValue, що представлений на рисунку 3.15. Екземпляри StringValue зберігаються у полі «value» структури Operation.

```
enum StringValue: Codable {
    case null
    case insert(char: UInt16)
    case delete

    init(from decoder: Decoder) throws {•••}

    func encode(to encoder: Encoder) throws {•••}
}
```

Рисунок 3.15 – Реалізація структури StringValue

Кожне причинне дерево починається з пустого значення «null» у корні.

Виходячи з того, що робота з послідовним блоком пам'яті є більш ефективною, ніж з класичною реалізацією дерев, де елементи пов'язані

покажчиками, є необхідність в реалізації свого типу сховища, що зберігає елементи у відсортованому порядку відповідно до обходу дерева у глибину [3]. Таким є структура Weave, яка є частиною структури CausalTree. Обидві структури даних реалізують протокол CvRDT, що представлений на рисунку 3.16.

```
public protocol CvRDT: Codable, Hashable {

    mutating func integrate(_ value: inout Self)

    func isSuperset(of value: inout Self) -> Bool

    func validate() throws -> Bool
}
```

Рисунок 3.16 – Протокол CvRDT

Для оптимізації операції пошуку операції до O(1), структура Weave має додатковий кеш – структуру Yarn, яка представляє собою хеш-таблицю з відповідними UUID репліки у якості ключа та масиву операцій, що були спричинені нею [3]. Для цього до ідентифікатора операції було додане додаткове поле «index». Приклад заповненого кешу представлений на рисунку 3.17.

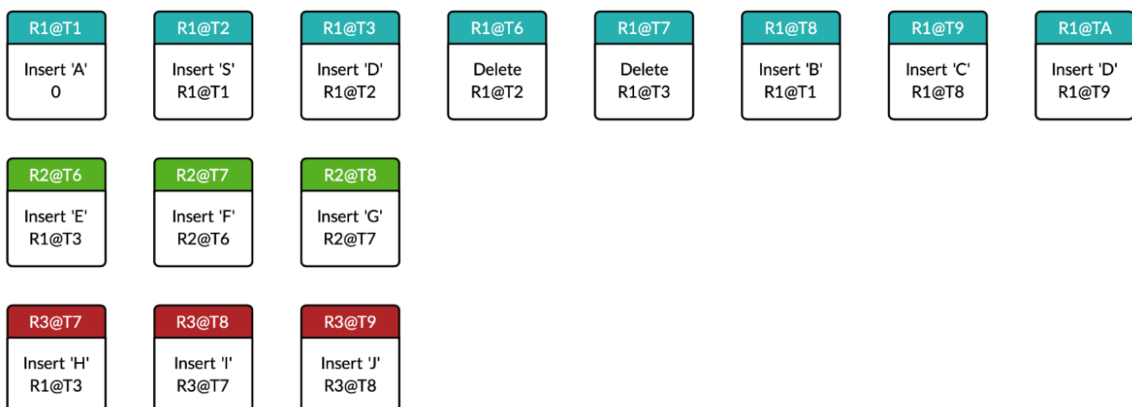

Рисунок 3.17 – Приклад заповненої структури Yarn

Таким чином роботу алгоритму реконструкції було покращено і він має лінійну складність. Дане покращення призвело до погіршення асимптотичної складності операцій вставки та видалення, які також тепер мають лінійну складність замість константної, але операція злиття є більш пріоритетною. Додаткова пам’ять зросла лінійно, що є припустимим в межах норми.

3.5 Порівняння реалізацій

Вище було розглянуто 2 реалізації послідовних CRDT для редагування тексту:

- реалізація RGA на базі суміжного масиву;
- реалізація RGA на базі причинних дерев.

Основними критеріями порівняння є алгоритмічна складність наступних операцій:

- злиття реплік;
- вставка символу;
- видалення символу;
- пошук символу.

Порівняння даних реалізацій на основі відповідних критеріїв оцінки представлено в таблиці 3.1. Продуктивність операції злиття є найважливішою.

Таблиця 3.1 – Порівняння ефективності реалізованих структур даних

Операція	RGA на базі суміжного масиву	RGA на базі причинних дерев
злиття	$O(N^2)$	$O(N)$
вставка	$O(1)$	$O(N)$
видалення	$O(1)$	$O(N)$
пошук	$O(N)$	$O(1)$

Реалізація на базі причинних дерев потребує вдвічі більше пам'яті для зберіганні додаткового кешу, але дане зростання є припустимим, тому що є лінійним.

Отже, виходячи з таблиці 3.1, якщо відомо, що операція злиття виконується рідко та розмір документу є невеликим (до 5 000 символів), рекомендованою є реалізація моделі RGA на базі суміжного масиву. В противному випадку – реалізація на базу причинних дерев.

3.5 Висновки по розділу 3

В третьому розділі був спроектований та реалізований програмний модуль, що включає моделі безконфліктних реплікованих типів даних для роботи з текстом. Даним програмний модуль був використаний у мобільному додатку для колаборативної роботи з текстовими документами на базі операційної системи iOS та сховища даних CloudKit. Для інтеграції програмного модулю в мобільний додаток були реалізовані класи обгортки для компонентів редагування тексту, які можуть бути використані повторно в іншому продукті при необхідності.

Для роботи з текстом виділяють структури WOOT, Treedoc, Logoot, LSEQ та RGA. Найбільш гнучкою та актуальною реалізацією є RGA, алгоритми та підходи якої були досліджені у науковій роботі. Даний підхід передбачає, що кожен символ тексту посилається на свого лівого сусіда та має додаткову хеш-таблицю для покращення продуктивності пошуку символів.

В ході роботи було порівняно 2 реалізації структури даних RGA: реалізація на базі суміжного масиву; реалізація на базі причинних дерев.

Найважливішою операцією є операція злиття. В реалізації на базі причинних дерев вона має лінійну асимптотичну складність алгоритму. Отже, було встановлено, що реалізація на базі причинних дерев є більш продуктивною, ніж реалізація на базі суміжного масиву, але потребує більшого обсягу пам'яті через наявність додаткового кешу.

ВИСНОВКИ

В ході дипломної роботи були досліджені моделі безконфліктних реплікованих типів даних у якості інструменту для вирішення конфліктів між репліками, а також особливості їх використання у мобільних колаборативних офлайн застосунках для редагування текстових документів.

Був проведений аналіз існуючих моделей роботи з даними у залежності від толерантності до відсутності підключення до мережі. Вибір правильної моделі для кожного типу програмного забезпечення є дуже важливим. Для колаборативних застосунків найбільше підходить офлайн модель, тому що вона є найбільш толерантною до відсутності підключення до мережі, а доступність даних має найвищий пріоритет.

Підхід безконфліктних реплікованих типів даних був порівняний з двома альтернативами: Operational Transformation та диференціальною синхронізацією, основним недоліком яких є неповністю доведена математична збіжність алгоритмів.

Безконфліктні репліковані типи даних вирішують проблему узгодженості даних, використовуючи модель суворої узгодженості у кінцевому рахунку. Вони поділяються на 2 категорії: CvRDT (засновані на передачі стану) та CmRDT (засновані на передачі операцій). CmRDT є більш гнучким рішенням, але потребує реалізації більш складних алгоритмів ніж CvRDT. В свою чергу, CvRDT має спрощені вимоги до протоколу комунікації між репліками, що є великою перевагою.

Сучасні CRDT представляють собою композицію існуючих більш простих структур. Тому було проведено огляд та аналіз існуючих структур, починаючи від базових понять атому та об’єкту, лічильників, регістрів, їх особливостей, та закінчуючи множинами та графами.

Для роботи з текстом виділяють структури WOOT, Treedoc, Logoot, LSEQ та RGA. Найбільш гнучкою та актуальною реалізацією є RGA, алгоритми та

підходи якої були досліджені у науковій роботі. Даний підхід передбачає, що кожен символ тексту посилається на свого лівого сусіда та має додаткову хеш-таблицю для покращення продуктивності пошуку символів.

В ході роботи було порівняно 2 реалізації структури даних RGA: реалізація на базі суміжного масиву; реалізація на базі причинних дерев.

Найважливішою операцією є операція злиття. В реалізації на базі причинних дерев вона має лінійну асимптотичну складність алгоритму. Отже, було встановлено, що реалізація на базі причинних дерев є більш продуктивною, ніж реалізація на базі суміжного масиву, але потребує більшого обсягу пам’яті через наявність додаткового кешу.

В ході роботи був спроектований та реалізований програмний модуль, що включає моделі безконфліктних реплікованих типів даних для роботи з текстом. Даним програмний модуль був використаний у мобільному додатку для колаборативної роботи з текстовими документами на базі операційної системи iOS та сховища даних CloudKit. Для інтеграції програмного модулю в мобільний додаток були реалізовані класи обгортки для компонентів редагування тексту, які можуть бути використані повторно в іншому продукті при необхідності.

В подальшому, реалізований програмний модуль потребує впровадження підтримки більш складних структур CRDT, що дозволить використовувати його в інших галузях окрім роботи з текстовими документами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Definition of a Collaborative Application. URL: http://www.cs.unc.edu/~dewan/290/s97/notes/intro/node2.html (дата звернення 26.04.2020).

2. Online vs Offline Application Design. URL: https://blogs.sap.com/2016/04/29/offline-vs-online-application-design/ (дата звернення 26.04.2020).

3. Data Laced with History: Causal Trees & Operational CRDTs. URL: http://archagon.net/blog/2018/03/24/data-laced-with-history/ (дата звернення 26.04.2020).

4. A comprehensive study of Convergent and Commutative Replicated Data Types. URL: https://hal.inria.fr/inria-00555588/document (дата звернення 26.04.2020).

5. Andriy Yerokhin, Valerii Semenets , Alina Nechyporenko, Andrii Babii, Oleksii Turuta. F-transform 3D Point Cloud Filtering Algorithm // Proc. of the 2th IEEE International Conference on Data Stream Mining & Processing. 21-25 August 2018, Lviv, Ukraine. - P.524-527. DOI: 10.1109/DSMP.2018.8478581.

6. Offline Definition. URL: https://techterms.com/definition/offline (дата звернення 26.04.2020).

7. What does Minimum Viable Product (MVP) mean. URL: https://www.techopedia.com/definition/27809/minimum-viable-product-mvp (дата звернення 26.04.2020).

8. Collaborative Software Defitition. URL: https://en.wikipedia.org/wiki/Collaborative_software (дата звернення 22.03.2020).

9. Теорема CAP. URL: https://uk.wikipedia.org/wiki/Теорема_CAP (дата звернення 26.04.2020).

10. ACID Definition. URL: https://uk.wikipedia.org/wiki/ACID (дата звернення 26.04.2020).

11. Consistency Model. URL. https://en.wikipedia.org/wiki/Consistency_model (дата звернення 26.04.2020).

12. Eventual Consistency Definition. URL: http://mlwiki.org/index.php/Eventual_Consistency (дата звернення 26.04.2020).

13. Andriy Yerokhin, Alina Nechyporenko, Andrii Babii, Oleksii Turuta, Ihor Mahdalina. Usage of Phase Space Diagram to Finding Significant Features of Rhinomanometric Signals // Computer Science & Information Technologies (CSIT'2016), 14-17 September 2016, Lviv, Ukraine. – P. 70 - 72. DOI: 10.1109/STC-CSIT.2016.7589871.

14. Andriy Yerokhin, Alina Nechyporenko, Andrii Babii, Oleksii Turuta. A new intelligence-based approach for rhinomanometric data processing // Proc. of 2016 IEEE 36th International Conference on Electronics and Nanotechnology, ELNANO 2016. – 19-21 April 2016. – P.198-201. DOI: 10.1109/ELNANO.2016.7493047

15. Why Logical Clocks Are Easy. URL: https://queue.acm.org/detail.cfm?id=2917756 (дата звернення 26.04.2020).

16. What Is The oOPT Puzzle. URL: https://www3.ntu.edu.sg/home/czsun/projects/otfaq/#_Toc321146192 (дата звернення 26.04.2020).

17. Differential Synchronization. URL: https://neil.fraser.name/writing/sync/ (дата звернення 20.09.2020).

18. Writing: Diff Strategies. URL: https://neil.fraser.name/writing/diff/ (дата звернення 26.09.2020).

19. Yerokhin, A.L., Babii, A.S., Nechyporenko, A.S., Turuta, O.P. / A Lars-Based Method of the Construction of a Fuzzy Regression Model for the Selection of Significant Features // Cybernetics and Systems Analysis. №4, 2016. - P. 167–173.

20. Fuzzy patch. URL: https://neil.fraser.name/writing/patch/ (дата звернення 19.10.2020).

21. Saito Y., Shapiro M. Optimistic replication. ACM Computing Surveys, 2005. – 81p.

22. Напіврешітка. URL: https://ru.wikipedia.org/wiki/Полурешётка (дата звернення 26.04.2020).

23. CRDT Primer Part II: Convergent CRDTs. URL: http://jtfmumm.com/blog/2015/11/24/crdt-primer-2-convergent-crdts/ (дата звернення: 28.10.2020).

24. CRDT: Conflict-free Replicated Data Types. URL: https://habr.com/ru/post/418897/ (дата звернення 26.09.2020).

25. Реплікація без конфліктів: CRDT в теорії та на практиці. URL: https://habr.com/ru/post/272987/ (дата звернення 26.10.2020).

26. Larman K. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Tutorial. Williams, 2006. – 736 p.

27. CloudKit Quick Start. URL: https://developer.apple.com/library/archive/documentation/DataManagement/Conceptual/CloudKitQuickStart/Introduction/Introduction.html#//apple_ref/doc/uid/TP40014987 (дата звернення 20.10.2020).

28. The Swift Programming Language. URL: https://docs.swift.org/swift-book/ (дата звернення 06.11.2020).

29. Мартін Р. Принципи OOD. SOLID. Навч. посіб. Вільямс, 2010. – 690 с.

30. Neuburg M. Programming iOS 13. Tutorial. O’Reilly, 2020. – 1204 p.

ДОДАТОК А

Лістинг фрагментів коду програми

```
final class FilesViewController: UIViewController {

    @IBOutlet private var collectionView: UICollectionView!
    @IBOutlet private var label: UILabel!
    @IBOutlet private var spinner: UIActivityIndicatorView!

    private lazy var deleteButtonItem =
UIBarButtonItem(barButtonSystemItem: .trash,
                                                        target: self,
                                                        action:
#selector(deleteObjects(_:)))

    private lazy var addButtonItem = UIBarButtonItem(barButtonSystemItem:
.add,
                                                     target: self,
                                                     action:
#selector(insertNewObject(_:)))

    private var detailViewController: DetailViewController?

    private var ids: [Network.FileID] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        navigationItem.leftBarButtonItems = [editButtonItem,
deleteButtonItem]
        navigationItem.rightBarButtonItems = [addButtonItem]
        observeFileUpdates()
        prepare()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        setEditing(false, animated: false)
    }

    override func viewDidDisappear(_ animated: Bool) {
        super.viewDidDisappear(animated)
        collectionView.indexPathsForSelectedItems?.forEach { indexPath in
            collectionView.deselectItem(at: indexPath, animated: false)
        }
    }

    private func observeFileUpdates() {
        NotificationCenter.default.addObserver(forName:
Network.FileChangedNotification, object: nil, queue: nil) {
            [weak self] notification in
            guard let self = self else { return }

            let changedIdsArray: [Network.FileID] =
notification.userInfo?[Network.FileChangedNotificationIDsKey] as?
[Network.FileID] ?? []
```

```
            let changedIds = Set(changedIdsArray)

            let newIds = DataManager.sharedInstance.network.ids()

            let oldIdsSet = Set(self.ids)
            let newIdsSet = Set(newIds)

            assert(oldIdsSet.count == self.ids.count && newIdsSet.count ==
newIds.count)

            let insertions = newIdsSet.subtracting(oldIdsSet)
            let deletions = oldIdsSet.subtracting(newIdsSet)
            let                         refreshes                         =
newIdsSet.intersection(oldIdsSet).intersection(changedIds)

            var insertionCommands: [IndexPath] = []
            var deletionCommands: [IndexPath] = []
            var refreshCommands: [IndexPath] = []

            var i = 0
            while i < self.ids.count {
                if refreshes.contains(self.ids[i]) {
                    refreshCommands.append(IndexPath(row: i, section: 0))
                }
                if deletions.contains(self.ids[i]) {
                    deletionCommands.append(IndexPath(row:  i,  section:
0))
                }
                i += 1
            }

            var j = 0
            while j < newIds.count {
                if insertions.contains(newIds[j]) {
                    insertionCommands.append(IndexPath(row:  j,  section:
0))
                }
                j += 1
            }

            self.ids = newIds

            self.collectionView.performBatchUpdates({
                self.collectionView.reloadItems(at: refreshCommands)
                self.collectionView.deleteItems(at: deletionCommands)
                self.collectionView.insertItems(at: insertionCommands)
            }, completion: nil)

            if     let     id     =     self.detailViewController?.id,
deletions.contains(id) {
                self.navigationController?.popViewController(animated:
true)
            }
        }
    }

    private func setInteractionEnabled(_ isEnabled: Bool) {
```

```
        navigationItem.leftBarButtonItems?.forEach   {   $0.isEnabled   =
isEnabled }
        navigationItem.rightBarButtonItems?.forEach   {   $0.isEnabled   =
isEnabled }
        collectionView.isUserInteractionEnabled = isEnabled
        collectionView.allowsSelection = isEnabled
    }

    @objc private func deleteObjects(_ sender: Any) {
        deleteSelectedItems()
    }

    @objc private func insertNewObject(_ sender: Any) {
        create()
    }

    // MARK: - Segues

    private var pendingMemoryId: Memory.InstanceID?

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if   segue.identifier   ==   "showDetail",   let   controller   =
segue.destination as? DetailViewController {
            if             let             indexPath             =
collectionView.indexPathsForSelectedItems?.first,   let   memoryId   =
pendingMemoryId {
                let             crdt             =
DataManager.sharedInstance.memory.getInstance(memoryId)
                controller.crdt = crdt
                controller.id = self.ids[indexPath.row]
                controller.navigationItem.leftItemsSupplementBackButton =
true
            }
            detailViewController = controller
            pendingMemoryId = nil
        }
    }

    override func setEditing(_ editing: Bool, animated: Bool) {
        super.setEditing(editing, animated: animated)
        deleteButtonItem.isEnabled = editing
    }
}

// MARK: - Actions

extension FilesViewController {

    private func prepare() {
        setInteractionEnabled(false)

        handleAction("Logging in...")

        DataManager.sharedInstance.network.login { error in
            if let error = error {
                self.handleError("Could not log in: \(error)")
            } else {
```

```
                self.handleSuccess("Data synchronized!")

                self.ids = DataManager.sharedInstance.network.ids()
                self.collectionView.reloadData()

                self.setInteractionEnabled(true)
            }
        }
    }

    private func create() {
        setInteractionEnabled(false)
        handleAction("Creating file...")

        let  id  =  DataManager.sharedInstance.memory.create(withString:
"Edit      me!      Created      on      \(Date().description)      by
\(DataManager.sharedInstance.id)", orWithData: nil)


DataManager.sharedInstance.memoryNetworkLayer.sendInstanceToNetwork(id,
createIfNeeded: true) { _, error in
            defer {
                // file was only opened to save
                DataManager.sharedInstance.memory.close(id)

DataManager.sharedInstance.memoryNetworkLayer.tempUnmap(memory: id)
            }

            if let error = error {
                self.handleError("Could not create file: \(error)")
                self.setInteractionEnabled(true)
            } else {
                self.handleSuccess("Created file, good to go!")

                // notification will have arrived at this point to alter
table
                self.setInteractionEnabled(true)
            }
        }
    }

    private func deleteSelectedItems() {
        collectionView.indexPathsForSelectedItems?.forEach { indexPath in
            delete(ids[indexPath.item])
        }
    }

    private func delete(_ id: Network.FileID) {
        setInteractionEnabled(false)

        handleAction("Deleting file...")

        DataManager.sharedInstance.memoryNetworkLayer.delete(id) { e in
            if let error = e {
                self.handleError("Could not delete file: \(error)")
                self.setInteractionEnabled(true)
            } else {
```

```
                self.handleFileDelete("Deleted file!")

                // notification will have arrived at this point to alter
table
                self.setInteractionEnabled(true)
            }
        }
    }
}

// MARK: - Loading Indicator

extension FilesViewController {

    private func handleError(_ message: String) {
        spinner.stopAnimating()
        spinner.isHidden = true

        label.textColor = .red
        label.text = message
    }

    private func handleSuccess(_ message: String) {
        spinner.stopAnimating()
        spinner.isHidden = true

        var hue: CGFloat = 0
        UIColor.green.getHue(&hue,  saturation:  nil,  brightness:  nil,
alpha: nil)
        label.textColor = UIColor(hue: hue, saturation: 0.8, brightness:
0.7, alpha: 1.0)
        label.text = message
    }

    private func handleAction(_ message: String) {
        spinner.startAnimating()
        spinner.isHidden = false

        var hue: CGFloat = 0
        UIColor.blue.getHue(&hue, saturation: nil, brightness: nil, alpha:
nil)
        label.textColor = UIColor(hue: hue, saturation: 0.9, brightness:
1.0, alpha: 1.0)
        label.text = message
    }

    private func handleFileDelete(_ message: String) {
        spinner.stopAnimating()
        spinner.isHidden = true

        var hue: CGFloat = 0
        UIColor.blue.getHue(&hue, saturation: nil, brightness: nil, alpha:
nil)
        label.textColor = UIColor(hue: hue, saturation: 0.9, brightness:
1.0, alpha: 1.0)
        label.text = message
    }
```

```
}

// MARK: - Collection View

extension         FilesViewController:         UICollectionViewDataSource,
UICollectionViewDelegateFlowLayout {

    func      collectionView(_      collectionView:      UICollectionView,
numberOfItemsInSection section: Int) -> Int {
        return ids.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {
        let                          cell                              =
collectionView.dequeueReusableCell(withReuseIdentifier:   "Cell",   for:
indexPath) as! FileCollectionViewCell

        let                        metadata                            =
DataManager.sharedInstance.network.metadata(ids[indexPath.item])!

        let formatter = DateFormatter()
        formatter.dateFormat = "yyyy-MM-dd HH:mm:ss Z"

        if let date = formatter.date(from: metadata.name) {
            let prettyFormatter = DateFormatter()
            prettyFormatter.dateFormat = "MMM d, h:mm a"

            cell.nameLabel.text = prettyFormatter.string(from: date)
        } else {
            cell.nameLabel.text = metadata.name
        }

        var hue: CGFloat = 0
        UIColor.green.getHue(&hue,  saturation:  nil,  brightness:  nil,
alpha: nil)
        let green = UIColor(hue: hue, saturation: 0.9, brightness: 0.8,
alpha: 1.0)

        if metadata.remoteShared {
            cell.nameLabel.textColor = green
        } else {
            cell.nameLabel.textColor = metadata.associatedShare != nil ?
.blue : .black
        }

        return cell
    }

    func collectionView(_ collectionView: UICollectionView,
                        layout                      collectionViewLayout:
UICollectionViewLayout,
                        sizeForItemAt indexPath: IndexPath) -> CGSize {
        let interItemSpacing: CGFloat = 16 * 2
        let      sectionInset      =      (collectionViewLayout      as!
UICollectionViewFlowLayout).sectionInset.left + (collectionViewLayout as!
UICollectionViewFlowLayout).sectionInset.right
```

```
        let contentWidth = collectionView.bounds.width - interItemSpacing
- sectionInset
        let itemSize = contentWidth / 3
        return CGSize(width: itemSize, height: itemSize)
    }

    func      collectionView(_      collectionView:      UICollectionView,
didSelectItemAt indexPath: IndexPath) {
        guard          !isEditing,          let          indexPath          =
collectionView.indexPathsForSelectedItems?.first else {
            return
        }

        let id = ids[indexPath.row]

        var memoryId: Memory.InstanceID?
        var errorValue: Error?

        let group = DispatchGroup()
        group.enter()

DataManager.sharedInstance.memoryNetworkLayer.sendNetworkToInstance(id,

createIfNeeded: true,

continuingAfterMergeConflict: false) {

identifier, error in
            if let error = error {
                errorValue = error
            } else {
                memoryId = identifier
            }
            group.leave()
        }

        group.notify(queue: .main) {
            if let error = errorValue {
                self.handleError("Error retrieving file: \(error)")
            } else {
                self.pendingMemoryId = memoryId
                self.performSegue(withIdentifier:  "showDetail",  sender:
nil)
            }
        }
    }
}

public final class CRDTTextStorage: NSTextStorage {

    public private(set) var stringWrapper: CausalTreeStringWrapper

    private var backedString: NSMutableAttributedString!

    private var defaultTextAttributes: [NSAttributedString.Key: Any] {
        let paragraphStyle = NSMutableParagraphStyle()
```

```
        paragraphStyle.lineSpacing = 2.5
        return [
            .font: UIFont.systemFont(ofSize: 16),
            .foregroundColor: UIColor.darkText,
            .paragraphStyle: paragraphStyle
        ]
    }

    public var revision: CausalTreeString.WeftT? {
        didSet {
            stringWrapper.revision = revision
            reloadData()
        }
    }

    private var isFixingAttributes = false

    // AB: a new container is sometimes created on paste — presumably to
hold the intermediary string — so we have
    // to do this slightly ugly hack; this CT is merely treated like an
ordinary string and does not merge with anything
    var _kludgeCRDT: CausalTreeString?

    // MARK: - Init

    public override convenience init() {
        let kludge = CausalTreeString(site: UUID.zero, clock: 0)
        self.init(withCRDT: kludge)
        self._kludgeCRDT = kludge
    }

    public required init(withCRDT crdt: CausalTreeString) {
        self.stringWrapper = CausalTreeStringWrapper()
        self.stringWrapper.initialize(crdt: crdt)

        super.init()

        self.backedString       =       NSMutableAttributedString(string:
self.stringWrapper as String, attributes: defaultTextAttributes)
    }

    public required init?(coder aDecoder: NSCoder) {
        fatalError()
    }

    public required init?(pasteboardPropertyList propertyList: Any, ofType
type: UIPasteboard.Type) {
        fatalError()
    }

    // MARK: - Reload

    public func reloadData() {
        self.beginEditing()
        let oldLength = backedString.length
        let newString = stringWrapper
```

```
        backedString.replaceCharacters(in: NSRange(location: 0, length:
oldLength), with: newString as String)
        let newLength = backedString.length
        edited(.editedCharacters, range: NSRange(location: 0, length:
oldLength), changeInLength: newLength - oldLength)
        endEditing()
    }

    // MARK: - Parent Methods

    public override var string: String {
        return backedString.string
    }

    public override func replaceCharacters(in range: NSRange, with text:
String) {
        stringWrapper.replaceCharacters(in: range, with: text)

        let oldCacheLength = backedString.length
        backedString.replaceCharacters(in: range, with: text)
        let newCacheLength = backedString.length

        edited(.editedCharacters, range: range, changeInLength:
newCacheLength - oldCacheLength)
    }

    public override func setAttributes(_ attributes:
[NSAttributedString.Key: Any]?, range: NSRange) {
        if isFixingAttributes {
            backedString.setAttributes(attributes, range: range)
            edited(.editedAttributes, range: range, changeInLength: 0)
        }
    }

    public override func fixAttributes(in range: NSRange) {
        isFixingAttributes = true
        super.fixAttributes(in: range)
        isFixingAttributes = false
    }

    public override func processEditing() {
        isFixingAttributes = true
        setAttributes(nil, range: editedRange)
        setAttributes(defaultTextAttributes, range: editedRange)
        isFixingAttributes = false
        super.processEditing()
    }

    public override func attributes(at location: Int, effectiveRange
range: NSRangePointer?) -> [NSAttributedString.Key: Any] {
        return backedString.attributes(at: location, effectiveRange:
range)
    }
}
public final class CRDTTextEditor: NSCopying, Codable {
    public typealias CRDTMapType = CRDTMap<CausalTreeString.SiteUUIDT,
AtomId, CausalTreeString.SiteUUIDT>
```

```
    public private(set) var tree: CausalTreeString
    public private(set) var cursorMap: CRDTMapType

    public init(site: CausalTreeString.SiteUUIDT) {
        self.tree = CausalTreeString(site: site, clock:
Clock(CACurrentMediaTime() * 1000))
        self.cursorMap = CRDTMap(withOwner: site)
    }

    public func updateCursor(to atomId: AtomId) {
        cursorMap.setValue(atomId)
    }

    public func transfer(to uuid: CausalTreeString.SiteUUIDT, clock:
Clock) -> ([SiteId: SiteId]) {
        let remap = tree.transferToNewOwner(withUUID: uuid, clock: clock)

        for pair in cursorMap.map {
            if let newSite = remap[pair.value.value.site] {
                cursorMap.setValue(AtomId(site: newSite, index:
pair.value.value.index), forKey: pair.key, updatingId: false)
            }
        }
        cursorMap.owner = uuid

        return remap
    }

    public func incrementTimestamp() {
        let _ = tree.weave.lamportTimestamp.increment()
        let _ = cursorMap.lamportTimestamp.increment()
    }

    public static func ==(lhs: CRDTTextEditor, rhs: CRDTTextEditor) ->
Bool {
        return lhs.tree == rhs.tree && lhs.cursorMap == rhs.cursorMap
    }

    public func hash(into hasher: inout Hasher) {
        hasher.combine(tree)
        hasher.combine(cursorMap)
    }

    public func copy(with zone: NSZone? = nil) -> Any {
        let returnCopy = CRDTTextEditor(site: tree.ownerUUID())

        returnCopy.tree = tree.copy() as! CausalTreeString
        returnCopy.cursorMap = cursorMap.copy() as! CRDTMap

        return returnCopy
    }
}

// MARK: - CvRDT

extension CRDTTextEditor: CvRDT {
```

```
    public func integrate(_ v: inout CRDTTextEditor) {
        let remaps = tree.integrateReturningSiteIdRemaps(&v.tree)

        for pair in cursorMap.map {
            if let newSite = remaps.localRemap[pair.value.value.site] {
                cursorMap.setValue(AtomId(site:      newSite,      index:
pair.value.value.index), forKey: pair.key, updatingId: false)
            }
        }
        for pair in v.cursorMap.map {
            if let newSite = remaps.remoteRemap[pair.value.value.site] {
                v.cursorMap.setValue(AtomId(site:     newSite,     index:
pair.value.value.index), forKey: pair.key, updatingId: false)
            }
        }
        cursorMap.integrate(&v.cursorMap)
    }

    public func isSuperset(of v: inout CRDTTextEditor) -> Bool {
        return  tree.isSuperset(of:  &v.tree)  &&  cursorMap.isSuperset(of:
&v.cursorMap)
    }

    public func validate() throws -> Bool {
        return try tree.validate() && cursorMap.validate()
    }
}
```