НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри

_____ Артем ВОЛОКИТА
(підпис)

“___” _____ 2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Модуль синхронізації текстових даних у розподілених системах

Виконав: студент 4 курсу, групи ІМ-21
(шифр групи)

Сірик Максим Олександрович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доц. кафедри ОТ ФІОТ, к.т.н., с.н.с. Долголенко О. М. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент кафедри ОТ Коренко Д. В. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доц. кафедри СПСКС ФПМ, к.т.н., доц. Щербина О. А. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)
Освітньо-професійна програма
«Інженерія програмного забезпечення комп'ютерних систем»
спеціальності 121 "Інженерія програмного забезпечення"

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри

_____ Артем ВОЛОКИТА
(підпис)

"___" _____ 2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Сірика Максима Олександровича

1. Тема проєкту *Модуль синхронізації текстових даних у розподілених системах*
   керівник проєкту Долголенко Олександр Миколайович, к.т.н., с.н.с.,
   (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
   затверджені наказом по університету від «11» травня 2026 р. №1139-с
2. Термін здачі студентом закінченого проєкту «4» червня 2026 р.
3. Вихідні дані до проєкту *технічна документація, теоретичні дані*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
   Розділ 1. Аналіз задачі та підходів до спільного редагування
   Розділ 2. Обґрунтування вибору технологій для розробки модуля
   Розділ 3. Проєктування та реалізація модуля синхронізації
   Розділ 4. Тестування, аналіз та обмеження розробленого модуля

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) компоненти застосунка (структурна схема), діаграма класів (функціональна схема), алгоритм роботи застосунку (принципова схема),

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Коренко Д. В.		

7. Дата видачі завдання «13» квітня 2026 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	02.02.2026-19.04.2026	
2.	*Вивчення та аналіз завдання*	20.04.2026-26.04.2026	
3.	*Розробка архітектури та загальної структури системи*	27.04.2026-30.04.2026	
4.	*Розробка структур окремих підсистем*	01.05.2026-03.05.2026	
5.	*Програмна реалізація системи*	04.05.2026-15.05.2026	
6.	*Оформлення пояснювальної записки*	16.05.2026-24.05.2026	
7.	*Захист програмного продукту*	05.06.2026	
8.	*Передзахист*	08.06.2026	
9.	*Захист*	15.06.2026	

Студент-дипломник _____ Максим СІРИК
(підпис)

Керівник проєкту _____ Олександр ДОЛГОЛЕНКО
(підпис)

АНОТАЦІЯ

У бакалаврському дипломному проєкті розроблено модуль синхронізації текстових даних у розподілених системах. Модуль призначений для спільного редагування тексту без обов'язкового центрального координатора та ґрунтується на підході конфліктно-вільних реплікованих типів даних (CRDT).

У роботі реалізовано текстову CvRDT-структуру. Реалізацію виконано мовою Zig з можливістю компіляції у WebAssembly. Для використання у вебсередовищі розроблено TypeScript-обгортку та демонстраційний застосунок. Коректність основних властивостей перевірено модульними, сценарними та рандомізованими fuzz-тестами.

Ключові слова: розподілені системи, синхронізація текстових даних, спільне редагування документів, конфліктно-вільні репліковані типи даних (CRDT), збіжність, автономна робота, peer-to-peer-синхронізація, local-first-застосунки, WebAssembly, Zig, TypeScript.

ANNOTATION

The bachelor's diploma project develops a module for synchronizing text data in distributed systems. The module is intended for collaborative text editing without requiring a central coordinator and is based on the Conflict-free Replicated Data Types (CRDT) approach.

The work implements a text state-based CRDT structure. The implementation is written in Zig and can be compiled to WebAssembly. A TypeScript wrapper and a demonstration application were developed for use in a web environment. The correctness of the main properties was verified using unit tests, scenario tests, and randomized fuzz tests.

Keywords: distributed systems, text data synchronization, collaborative document editing, conflict-free replicated data types (CRDT), convergence, offline operation, peer-to-peer synchronization, local-first applications, WebAssembly, Zig, TypeScript.

справки	Формат	Значення	Найменування	Кіл. листів	№ екземплярів	Додаток
			Документація загальна			
			Знову розроблена			
	А4	**ІАЛЦ.467100.002 ТЗ**	Модуль синхронізації текстових	4		
			даних у розподілених системах			
			Технічне завдання			
	А4	**ІАЛЦ.467100.003 ПЗ**	Модуль синхронізації текстових	65		
			даних у розподілених системах			
			Пояснювальна записка			
	А4	**ІАЛЦ.467100.004 Д1**	Модуль синхронізації текстових	1		
			даних у розподілених системах			
			Компоненти застосунка			
			(структурна схема)			
	А4	**ІАЛЦ.467100.005 Д2**	Модуль синхронізації текстових	1		
			даних у розподілених системах			
			Діаграма класів			
			(функціональна схема)			
	А4	**ІАЛЦ.467100.006 Д3**	Модуль синхронізації текстових	1		
			даних у розподілених системах			
			Алгоритм роботи застосунку			
			(принципова схема)			
	А4	**ІАЛЦ.467100.007 Д4**	Модуль синхронізації текстових	26		
			даних у розподілених системах			
			Текст програмного коду			

					ІАЛЦ.467100.001 ОА		
Зм.	Лист	№ докум.	Підпис	Дата			
Розроб.		Сірик М. О.			*Модуль синхронізації текстових даних у розподілених системах* *Опис альбому*	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Долголенко О. М.					
						НТУУ «КПІ» ФІОТ ІМ-21	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Модуль синхронізації текстових даних у розподілених системах*»

Київ – 2026 р.

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

Це технічне завдання поширюється на розробку модуля синхронізації текстових даних у розподілених системах.

Область використання: браузерні та локальні застосунки для спільного редагування текстових документів, системи local-first, прототипи peer-to-peer-редакторів.

2 ПРИЧИНИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського дипломного проєкту за темою «Модуль синхронізації текстових даних у розподілених системах» спеціальності 121 «Інженерія програмного забезпечення», затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення модуля синхронізації текстових даних, придатного для використання у браузерному середовищі.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література, документація, публікації та статті в мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до продукту, що розробляється

Реалізація модуля має бути придатною для створення систем, що відповідають таким вимогам:

- **Гарантована збіжність.** Якщо всі репліки отримають однаковий набір оновлень, вони мають зрештою перейти в однаковий стан.

- **Збереження наміру користувача.** Конкурентні редагування не повинні непомітно втрачатися. Об’єднаний результат має відображати те, що користувачі могли б обґрунтовано очікувати від застосування їхніх змін.
- **Висока інтерактивність.** Локальні зміни мають застосовуватися з дуже малою затримкою, бажано меншою за ~100 мс.
- **Редагування в реальному часі.** Віддалені оновлення мають поширюватися достатньо швидко, щоб спільне редагування сприймалося як природне.
- **Повністю розподілена робота.** Система не повинна вимагати додаткової інфраструктури, наприклад центрального сервера, для синхронізації або керування конкурентними змінами.
- **Динамічна участь.** Учасники можуть приєднуватися до системи або залишати її у будь-який момент.
- **Стійкість до мережевих збоїв та автономного редагування.** Система має продовжувати роботу за наявності затримок повідомлень, зміни їхнього порядку, тимчасового роз’єднання та роботи без мережі.
- **Масштабованість.** Система має залишатися практично придатною для великої кількості учасників (> 100).
- **Робота у веббраузері.**

5.2 Вимоги до програмного забезпечення

Для розробки, збирання та перевірки модуля необхідне таке програмне забезпечення:

- операційна система Linux;
- компілятор Zig версії не нижче `0.16.0`;

5.3 Вимоги до апаратної частини

- процесор архітектури `x86_64`;
- 16 ГБ оперативної пам’яті;
- 20 ГБ вільного місця на диску.

6 ЕТАПИ РОЗРОБКИ

Розробка виконується за такими етапами:

№	Назва етапу	Термін виконання
1	Затвердження теми проєкту	02.02.2026-19.04.2026
2	Вивчення та аналіз завдання	20.04.2026-26.04.2026
3	Розробка архітектури та загальної структури системи	27.04.2026-30.04.2026
4	Розробка структур окремих підсистем	01.05.2026-03.05.2026
5	Програмна реалізація системи	04.05.2026-15.05.2026
6	Оформлення пояснювальної записки	16.05.2026-24.05.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Модуль синхронізації текстових даних у розподілених системах*»

Київ – 2026 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API	Application Programming Interface – інтерфейс прикладного програмування
CRDT	Conflict-free Replicated Data Type – конфліктно-вільний реплікований тип даних
CvRDT	State-based Conflict-free Replicated Data Type – CRDT на основі стану
CmRDT	Operation-based Conflict-free Replicated Data Type – CRDT на основі операцій
OT	Operational Transformation – операційне перетворення
P2P	Peer-to-peer – однорангова взаємодія
RGA	Replicated Growable Array – реплікований розширюваний масив
RLE	Run-Length Encoding – кодування довжин повторів
UTF-8	Unicode Transformation Format 8-bit – 8-бітний формат перетворення Unicode
UUID	Universally Unique Identifier – універсальний унікальний ідентифікатор
WASM	WebAssembly – портативний бінарний формат інструкцій для виконання у вебсередовищі та поза ним
YATA	Yjs Algorithm for Text Arrangement – алгоритм упорядкування текстових елементів у Yjs
LEB128	Little Endian Base 128 – кодування цілих чисел змінної довжини

ВСТУП

Спільне редагування текстових документів стало однією з базових можливостей сучасних програмних систем. Для користувача така функціональність має виглядати просто, тому зміни мають застосовуватися локально без помітної затримки, віддалені правки з'являтися автоматично, а результат спільної роботи залишатися узгодженим навіть за одночасного редагування.

З технічного погляду ця задача є складною, оскільки документ існує у вигляді кількох копій на різних вузлах системи. Ці копії можуть тимчасово розходитися через затримки мережі, відсутність з'єднання, повторну доставку повідомлень або отримання оновлень в іншому порядку. У централізованих системах частину цих проблем можна розв'язувати за допомогою сервера, який встановлює єдиний порядок операцій. Однак така архітектура створює залежність від центрального вузла та гірше підходить для сценаріїв, у яких важливі автономна робота, локальне збереження даних і подальша синхронізація.

У роботі порівнюються три підходи до узгодження змін: Operational Transformation, тристороннє злиття та конфліктно-вільні репліковані типи даних. Для поставленої задачі найважливішими є локальне застосування змін, асинхронна синхронізація та збіжність реплік.

Актуальність роботи полягає в необхідності дослідити та реалізувати механізм синхронізації текстових даних, який не потребує обов'язкового центрального координатора, підтримує локальне редагування та забезпечує збіжність реплік після обміну оновленнями. Такий механізм може бути використаний як основа для редакторів, локально-орієнтованих застосунків, peer-to-peer-систем або вебзастосунків, що потребують спільної роботи з текстом.

Метою дипломного проєкту є розроблення модуля синхронізації текстових даних у розподілених системах, який забезпечує локальне редагування, обмін оновленнями між репліками та збіжність їхнього стану.

Для досягнення поставленої мети необхідно проаналізувати підходи до спільного редагування текстових документів та обґрунтувати вибір підходу для

розроблюваного модуля; реалізувати локальні операції редагування й інтеграцію віддалених оновлень; розробити механізм обміну станом між репліками з компактним кодуванням синхронізаційних повідомлень; надати JavaScript/ TypeScript-інтерфейс до модуля; перевірити коректність реалізації.

Об’єктом розробки є процес спільного редагування текстових даних у розподіленій системі. Предметом розробки є структура даних і алгоритми синхронізації, що забезпечують локальне редагування, обмін оновленнями та збіжність реплік.

Очікуваний технічний ефект від реалізації полягає у створенні компактного модуля, який задовольняє вимоги, зазначені в роботі. Практичне значення роботи полягає в можливості використання розроблених рішень як основи для подальшого створення систем спільного редагування.

РОЗДІЛ 1
АНАЛІЗ ЗАДАЧІ ТА ПІДХОДІВ ДО СПІЛЬНОГО РЕДАГУВАННЯ

1.1 Алгоритми спільного редагування

Алгоритми спільного редагування можна класифікувати за різними ознаками. З погляду керування конкурентними змінами їх зазвичай поділяють на дві категорії:

- **Песимістичні** оновлення вимагають отримання блокування від інших вузлів або центрального сервера перед застосуванням зміни.
- **Оптимістичні** оновлення відразу стають видимими користувачу в локальній копії, а конфлікти, спричинені конкурентними редагуваннями, розв’язуються пізніше. Тому оптимістичні підходи краще підходять для середовищ із великими затримками зв’язку [1, с. 112].

Песимістичні алгоритми є зрілими і широко використовуються на практиці. До поширених технік належать блокування, транзакції, одноосібна активна участь та виявлення залежностей [2, с. 401].

У сучасних дослідженнях спільного редагування частіше розглядають оптимістичні підходи. Дві найпомітніші родини таких підходів – операційне перетворення (Operational Transformation, OT) та конфліктно-вільні репліковані типи даних (Conflict-free Replicated Data Types, CRDT). Ще однією пов’язаною технікою оптимістичної синхронізації є тристороннє злиття, широко відоме за системами керування версіями, зокрема git [3].

1.1.1 Операційне перетворення (OT)

В OT зміни подаються як операції, наприклад *insert* та *delete*. Таке представлення є гнучкішим, ніж розгляд повних замін документа [4].

Простий приклад демонструє цю ідею. Нехай клієнт і сервер починають з однакового стану (`"Hola"`). Клієнт виконує набір змін *a*:

- вилучити `"o"` у позиції 2: `"Hola"` $\rightarrow$ `"Hla"`;
- вилучити `"a"` у позиції 3: `"Hla"` $\rightarrow$ `"Hl"`.

Водночас сервер отримує набір змін b:

- вставити `"e"` у позиції 2: `"Hola"` $\rightarrow$ `"Heola"`;
- вставити `"l"` у позиції 4: `"Heola"` $\rightarrow$ `"Heolla"`;
- вставити `"o"` у позиції 5: `"Heolla"` $\rightarrow$ `"Heolola"`.

У цьому прикладі позиції операцій задаються відносно локального стану сторони, яка їх створила. Якщо застосувати отримані віддалені операції з тими самими позиціями без перетворення, клієнт отримає `"Hello"`, а сервер – `"Hoola"`.

Формально ОТ визначає функцію перетворення [5]:

$$\text{xform}(a, b) = (a', b'), \text{ де } b' \circ \text{a} \equiv a' \circ \text{b} \tag{1.1}$$

Тобто для двох конкурентних операцій функція перетворення створює скориговані операції a' та b' так, щоб застосування a з подальшим b' дало той самий результат, що й застосування b з подальшим a' (рис. 1.1).

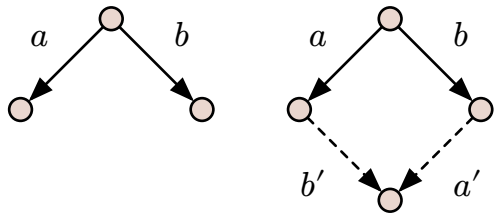


Рисунок 1.1 — Властивість ромба

Ця властивість ромба є основною ідеєю ОТ, але ситуація значно ускладнюється, коли стани розходяться більше ніж на один крок [4] (рис. 1.2). Формальні доведення коректності стають складними та схильними до помилок навіть для найпростішого випадку з двома операціями: видаленням і вставленням [6].

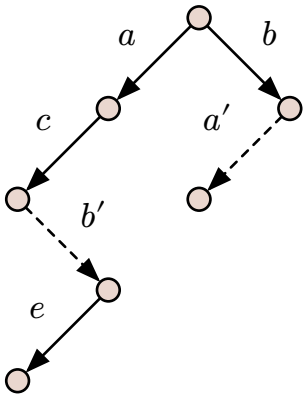


Рисунок 1.2 — Відхилення стану більш ніж на один крок в OT

OT часто використовується в клієнт-серверній архітектурі. І клієнт, і сервер можуть виконувати операційне перетворення, але сервер виступає центральною точкою координації, яка задає узгоджений порядок інтеграції для всіх реплік.

Джозеф Джентл, один з інженерів OT у Google Wave, також зазначав, що розподілена реалізація OT є значно складнішою, тоді як централізований варіант добре підходить для систем на зразок Google Docs [7].

Алгоритмічна складність не є єдиною причиною, чому OT гірше підходить для однорангової (peer-to-peer) взаємодії. У децентралізованих середовищах OT часто потребує додаткових метаданих для відстеження причинності та координації, щоб визначити, які операції є конкурентними і як їх потрібно перетворювати. Зі збільшенням кількості учасників підтримувати ці метадані та зберігати узгоджену поведінку перетворення стає складніше. Це робить OT менш привабливим для великих або динамічних однорангових груп [8, с. 259].

OT історично пов’язане з продуктами на зразок Google Docs [9]; у цьому продукті зараз кількість одночасних користувачів обмежена 100 [10].

1.1.2 Конфліктно-вільні репліковані типи даних (CRDT)

CRDT – це репліковані структури даних, спроєктовані так, щоб кілька реплік могли оновлюватися незалежно і все одно збігатися до однакового стану після синхронізації. Дві поширені родини CRDT – CRDT на основі стану (CvRDT) та CRDT на основі операцій (CmRDT). Для пояснення базових властивостей зручно розглянути CRDT на основі стану.

Ключова вимога полягає в тому, що множина станів має утворювати напівґратку об'єднання (рис. 1.3). Це означає, що для станів визначено частковий порядок $\leq$, а будь-які два стани x та y мають найменшу верхню межу ($x \sqcup y$).

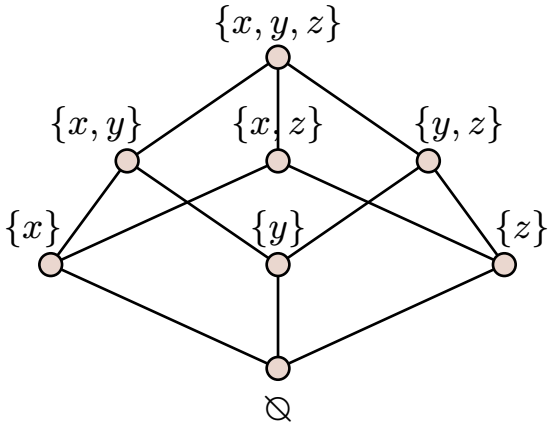


Рисунок 1.3 — Ґратка підмножин за відношенням включення

Будь-яке локальне оновлення має бути монотонним, тобто стан системи може лише зростати відносно заданого часткового порядку. Щоб гарантувати збіжність, операція злиття (*join*) має бути:

- комутативною: $x \sqcup y = y \sqcup x$;
- асоціативною: $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$;
- ідемпотентною: $x \sqcup x = x$ [11, с. 6].

Типовим прикладом є лічильник, що лише зростає (grow-only counter, G-Counter) [12]. Розглянемо розподілений сервіс, який зберігає кількість вподобань допису. Для масштабування системи запити користувачів можуть оброблятися різними серверами. Нехай дві репліки, a та b, починають з однакового видимого значення 0.

Якщо стан подати одним цілим числом, конкурентні збільшення можуть бути втрачені:

- запит «like» потрапляє на репліку a: $0 \to 1$;
- запит «like» потрапляє на репліку b: $0 \to 1$;
- після синхронізації злиття двох станів усе ще дає 1.

Отже, одного цілого числа недостатньо для представлення конкурентних збільшень.

Щоб розв'язати цю проблему, G-Counter зберігає вектор лічильників для окремих реплік, де кожна репліка збільшує лише власний компонент [11, с. 11].

- репліка a: $[0, 0] \to [1, 0]$;
- репліка b: $[0, 0] \to [0, 1]$;
- після синхронізації стани зливаються покомпонентно за допомогою операції $\max$: $[1, 0] \sqcup [0, 1] = [\max(1, 0), \max(0, 1)] = [1, 1]$.

Видиме значення лічильника отримують сумуванням усіх компонентів, що дає правильне підсумкове значення 2.

Видалення є складнішим, оскільки воно зазвичай зменшує стан, тобто є немонотонним оновленням, що порушує вимогу напівґратки. Для розв'язання цієї проблеми CRDT часто використовують tombstone-маркери, тобто маркери видалених елементів. Найпростішим прикладом є 2P-Set, поданий як дві множини, що лише зростають: $\mathbb{A}$ для доданих елементів і $\mathbb{R}$ для видалених елементів, тоді як видима множина визначається як $\mathbb{A} \setminus \mathbb{R}$. Після потрапляння елемента до $\mathbb{R}$ його вже не можна додати повторно. Саме тому tombstone-маркери зберігають достатньо історії для злиття [13].

Ці ідеї не є лише теоретичними. CRDT досліджувалися формально, зокрема за допомогою машинно перевірених доведень [14], і також використовуються у виробничих системах, таких як Redis [15], Zed [16], спільний редактор Typst [17] та інших. Вони також вплинули на новіші архітектури спільного редагування, наприклад архітектуру Figma, яка спирається на подібні ідеї, хоча не є чистим CRDT-рішенням [18].

1.1.3 Тристороннє злиття

Тристороннє злиття – ще одна техніка на основі оптимістичного керування конкурентністю. Найчастіше вона асоціюється із системами керування версіями. Цей підхід дозволяє незалежно змінювати локальні копії та узгоджувати результат значно пізніше. Процес злиття обчислюється за трьома вхідними даними: двома зміненими версіями та їхнім спільним предком. Якщо обидві сторони несумісно змінили один і той самий фрагмент, алгоритм повідомляє про конфлікт, який зазвичай потрібно розв'язувати вручну (рис. 1.4).

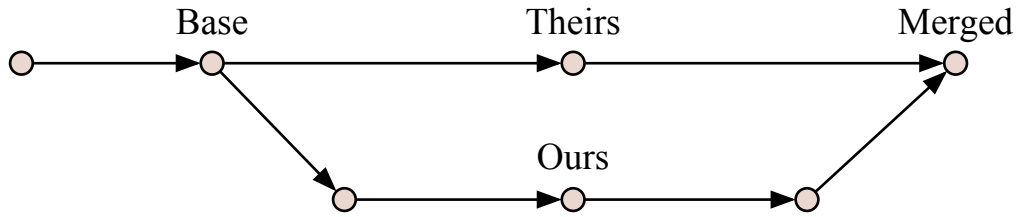


Рисунок 1.4 — Тристороннє злиття

Загалом цей підхід краще підходить для асинхронних сценаріїв із гілками, ніж для спільного редагування в реальному часі.

1.2 Порівняння підходів

Порівняємо ці підходи за різними критеріями у таблиці 1.1:

Таблиця 1.1 — Порівняння підходів до спільного редагування

Характеристика	Тристороннє злиття	OT	CRDT
Співпраця в реальному часі	Низька	Висока	Висока
Негайне застосування локальних змін	Так	Так	Так
Повністю розподілена робота	Висока	Низька. Теоретично можлива, але складна на практиці	Висока
Офлайн-редагування з відкладеною синхронізацією	Висока	Середня. Можливе, але узгодження змін ускладнюється	Висока
Динамічна участь реплік	Висока	Середня	Висока
Потреба в ручному розв’язанні конфліктів	Так	Ні	Ні
Математичні гарантії збіжності	Ні	Так	Так
Складність реалізації	Низька	Висока	Висока

Кінець таблиці 1.1

Характеристика	Тристороннє злиття	OT	CRDT
Накладні витрати метаданих синхронізації	Низькі	Середні або високі	Низькі або середні
Накладні витрати метаданих стану документа	Низькі	Низькі або середні	Високі

З огляду на вимоги цієї роботи (Розділ 5.1), найдоцільнішим підходом є CRDT. Тристороннє злиття корисне як концептуальна основа оптимістичного узгодження змін, однак воно краще підходить для асинхронних сценаріїв керування версіями. OT є сильним рішенням для централізованих редакторів реального часу, але гірше відповідає меті цієї роботи, оскільки передбачає наявність постійного координаційного сервера або значно ускладнюється в повністю розподіленому середовищі.

1.3 Огляд практичних реалізацій текстових CRDT

Текстові документи складніші за прості CRDT-лічильники або множини, оскільки текст є впорядкованою послідовністю елементів. Текстовий CRDT має зберігати порядок вставлених символів або фрагментів тексту, детерміновано розв’язувати конкурентні вставлення та підтримувати видалення без порушення збіжності. Тому практичні редактори спільного редагування використовують спеціалізовані CRDT для послідовностей.

У цьому розділі порівнюються кілька практичних реалізацій CRDT, які можна використовувати для спільного редагування тексту.

Yjs [19] – це реалізація CRDT на основі YATA, яка підтримує спільне редагування тексту, форматований текст та інші спільні типи даних, що можуть поєднуватися у складніші структури документа [20]. Yjs оптимізовано для практичних навантажень редагування. Зокрема, реалізація враховує

типові властивості введення тексту, за яких користувачі часто вставляють дані суцільними фрагментами, а локальні зміни зазвичай відбуваються поблизу попередньої позиції редагування. Також Yjs використовує компактне бінарне кодування повідомлень синхронізації, що допомагає зменшити використання пропускної здатності мережі.

Yjs реалізовано мовою JavaScript і спроєктовано для роботи у веббраузерах. Хоча сучасні рушії JavaScript оптимізують об’єкти зі стабільною формою (stable shape) [21], кожен об’єкт усе одно має службові метадані середовища виконання та збільшує накладні витрати керування пам’яттю. Тому компактне внутрішнє представлення є важливим для великих спільних документів. Також існує реалізація моделі Yjs/Yrs мовою Rust, яку можна компілювати у WebAssembly [22].

Automerge [23] – це JSON-подібний документний CRDT з підтримкою спільного редагування тексту. Його текстова модель використовує ідеї Peritext [24], тому підходить не лише для звичайного тексту, а й для форматованого тексту. На відміну від праць, присвячених окремим CRDT-алгоритмам, Automerge, схоже, не надає простої асимптотичної оцінки часової складності для всього публічного API. Його продуктивність переважно обговорюється через емпіричні вимірювання і залежить від конкретного робочого навантаження, структури документа та версії реалізації [25].

Бібліотека **json-joy** [26] – це набір алгоритмів для редагування в реальному часі, що містить реалізацію текстового CRDT на основі сімейства RGA. Його RGA-реалізація не зберігає кожен символ як окремий елемент зв’язаного списку, а використовує блокову RGA-структуру, орієнтовану на практичне редагування послідовностей. Тому її можна розглядати як приклад оптимізованого CRDT для послідовностей на основі RGA. Водночас збереження намірів користувача під час редагування форматованого тексту не є основним фокусом цієї реалізації; саме цю проблему розглядає Peritext, оскільки просте розширення CRDT для звичайного тексту або деревоподібних структур не завжди зберігає очікуваний результат форматування [27].

1.4 Порівняння реалізацій

Порівняємо ці реалізації за різними характеристиками у таблиці 1.2:

Таблиця 1.2 — Порівняння реалізацій CRDT для редагування тексту

Характеристика	Yjs	Automerge	json-joy RGA
Модель CRDT	CRDT на основі YATA з оптимізаціями реалізації	JSON-подібний CRDT з текстовою моделлю на основі Peritext	Блокова RGA / RGA-split CRDT для послідовностей
Звичайний текст	Так	Так	Так
Форматований текст	Так	Так	Частково; збереження намірів користувача не гарантується
Топологія синхронізації	Не залежить від мережевого транспорту; може використовувати клієнт-серверні або P2P-провайдери, зокрема WebRTC	Використовує протокол синхронізації Automerge поверх транспорту, вибраного застосунком	Залежить від застосунку або конкретної бібліотеки
Фокус продуктивності	Оптимізовано для вставок суцільних фрагментів тексту	Загальні local-first документи; продуктивність залежить від історії документа та робочого навантаження	Блокова RGA-структура для практичного редагування послідовностей

Продовження таблиці 1.2

Характеристика	Yjs	Automerge	json-joy RGA
Локальна вставка	$O(\log(H))$ [20, с. 45]	Переважно оцінюється емпірично	$O(N)$ [28, с. 106]
Локальне видалення	$O(\log(H))$ [20, с. 45]	Переважно оцінюється емпірично	$O(N)$ [28, с. 106]
Віддалена вставка	$O(H^2)$ [20, с. 45]	Переважно оцінюється емпірично	$O\left(1+\frac{c}{n}\right)$ [28, с. 106]
Віддалене видалення	$O(\log(H))$ [20, с. 45]	Переважно оцінюється емпірично	$O(1)$ [28, с. 106]
Переваги	Ефективна синхронізація тексту, компактні оновлення, зріла екосистема, підтримка браузерів, наявна Rust/WASM-реалізація	Універсальна local-first модель документа, підтримка форматованого тексту через ідеї Peritext, зручність для структурованих документів	Менша кількість функцій може спростити розуміння реалізації

Кінець таблиці 1.2

Характеристика	Yjs	Automerge	json-joy RGA
Обмеження	Складна внутрішня модель; деталі YATA/Yjs важче пояснити, ніж простіші академічні CRDT	Немає простої публічної асимптотичної оцінки складності для всього API; менше спеціалізованих оптимізацій для редагування тексту, ніж у Yjs	Менший фокус на збереженні намірів користувача у форматованому тексті; менша екосистема, ніж у Yjs або Automerge
Придатність для цієї роботи	Висока	Середньо-висока	Середня

У таблиці (див. таблицю 1.2) використано такі позначення:

- c – середня кількість операцій, конкурентних до заданої операції;
- n – розмір документа, тобто кількість невидалених символів;
- N – загальна кількість вставлених символів, включно з видаленими символами, що зберігаються як tombstone-маркери;
- H – кількість операцій, які вплинули на документ.

ВИСНОВОК ДО РОЗДІЛУ 1

У розділі розглянуто основні підходи до узгодження змін у системах спільного редагування. Песимістичні методи добре працюють там, де допустиме очікування блокування, але вони за визначенням гірше відповідають вимозі швидкої локальної реакції, тому для задачі інтерактивного редагування більш придатними є оптимістичні підходи. Операційне перетворення є практичним для централізованих редакторів, однак у повністю розподіленому середовищі потребує складної координації. Тристороннє злиття корисне для відкладеного узгодження версій і не призначене для одночасної роботи кількох активних користувачів над одним текстом.

Для поставленої задачі найбільш придатним є підхід CRDT, оскільки він дозволяє реплікам приймати локальні зміни незалежно, передавати їх пізніше та отримувати однаковий результат після обміну даними. Порівняння практичних реалізацій показало, що найближчим архітектурним орієнтиром для цієї роботи є Yjs та його модель текстового CRDT на основі YATA.

РОЗДІЛ 2
ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ МОДУЛЯ

2.1 Критерії вибору технологій

Для модуля синхронізації важливо, щоб ядро можна було запускати у браузері, інтегрувати з JavaScript/TypeScript-кодом і водночас зберігати контроль над внутрішнім представленням документа. Така вимога обмежує вибір технологій, оскільки код має або виконуватися безпосередньо у JavaScript-середовищі, або компілюватися у WebAssembly, який підтримується сучасними браузерами.

Окремо важливим є питання залежностей. Готові пакети прискорюють розробку, але розширюють ланцюг довіри до стороннього коду та його супровідників [29]. Навіть provenance-атестації npm підтверджують походження публікації, а не відсутність шкідливого коду в пакеті [30]. Це підтверджують інциденти компрометації популярних пакетів, зокрема атака на пакети TanStack [31]. Тому для ядра синхронізації було обрано підхід із мінімальною кількістю зовнішніх залежностей.

Основні критерії вибору технологій мають відповідати вимогам до реалізації. Найбільше на внутрішню структуру модуля впливає мова програмування, оскільки саме вона визначає модель пам’яті, доступні засоби інтеграції з браузером і спосіб організації залежностей.

2.2 Вибір мови програмування

Для реалізації розглядалися три основні варіанти: JavaScript/TypeScript, Rust і Zig. Вони відрізняються не лише синтаксисом, а й тим, як код інтегрується з браузером, які залежності зазвичай використовуються і наскільки явно програміст керує пам’яттю.

2.2.1 JavaScript та TypeScript

JavaScript є природною мовою для розроблення вебзастосунків, тому реалізація всього модуля цією мовою забезпечила б найпростішу інтеграцію з інтерфейсом користувача. TypeScript додає до JavaScript статичну перевірку типів і водночас зберігає JavaScript як модель виконання, оскільки після компіляції типи стираються, а поведінка програми залишається поведінкою JavaScript [32].

Для зовнішньої межі це перевага. TypeScript добре підходить для опису публічного API, типів параметрів і результатів. Саме тому у проєкті TypeScript використовується як інтерфейс над ядром.

Для самого CRDT-ядра цей варіант менш вдалий. Структура документа складається з елементів, індексів, діапазонів, байтових буферів і службових метаданих. У JavaScript таку модель можна реалізувати, але представлення об’єктів і оптимізації пам’яті залежать від рушія. Крім того, найшвидший шлях у JavaScript-екосистемі часто веде до використання готової бібліотеки, наприклад Yjs. Це було б практичним рішенням для продукту, але не відповідало б меті роботи.

2.2.2 Rust

Rust також є сильним кандидатом для системної частини. Його модель ownership дозволяє забезпечувати гарантії безпеки пам’яті без збирача сміття [33], а екосистема Rust має розвинену підтримку WebAssembly [34]. Крім того, існує реалізація моделі Yjs/Yrs мовою Rust, яку можна використовувати як практичний орієнтир [22].

Водночас у контексті цієї роботи Rust мав би інший компроміс. Значна частина складності могла б перейти в модель володіння та запозичення, особливо в коді з індексами, розділенням елементів, буферами та взаємними логічними посиланнями. Для навчальної реалізації важливо, щоб основна увага залишалася на структурі CRDT, а не на подоланні обмежень borrow checker.

2.2.3 Zig

Zig було обрано для реалізації ядра, оскільки ця мова надає низькорівневу модель програмування без прихованого середовища виконання та дозволяє явно керувати виділенням пам’яті через передавання allocator-об’єктів у структури, які володіють ресурсами [35]. Це добре відповідає внутрішній моделі проєкту, де елементи документа зберігаються в масивах, а текстові дані та атрибути розміщуються у спільних байтових буферах.

Другою важливою причиною є проста інтеграція з WebAssembly. Zig підтримує кроскомпіляцію та має вбудовану підтримку цілі WebAssembly, що дозволяє використовувати реалізоване ядро у вебсередовищі без перенесення основної логіки на JavaScript або TypeScript.

Водночас цей вибір має обмеження. Zig має меншу екосистему, ніж TypeScript або Rust, і на момент написання цієї роботи ще не досяг стабільної версії 1.0. Крім того, безпека пам’яті забезпечується не перевіркою запозичень borrow checker, як у Rust, а дисципліною реалізації з коректним використанням allocator-об’єктів, перевірками інваріантів, тестами та обмеженням складних патернів володіння.

Узагальнене порівняння мов і середовищ виконання наведено в таблиці 2.1.

Таблиця 2.1 — Порівняння мов для реалізації модуля

Критерій	JavaScript / TypeScript	Rust	Zig
Роль у проєкті	TypeScript-обгортка, публічні типи та інтеграція з вебінтерфейсом	Розглядався як системна альтернатива	Основна мова реалізації CRDT-ядра
Браузерне виконання	Безпосередньо	Через WebAssembly	Через WebAssembly

Кінець таблиці 2.1

Критерій	JavaScript / TypeScript	Rust	Zig
Модель пам’яті	Керується рушієм JavaScript	Ownership і borrow checker	Явні allocator-и та контрольовані структури даних
Готові CRDT-рішення	Yjs, Automerge та інші пакети	Yrs / y-crdt та WASM-екосистема	Власне ядро без використання готової CRDT-бібліотеки
Основний компроміс	Найзручніша інтеграція з вебсередовищем, але менший контроль над внутрішнім представленням	Сильні гарантії безпеки пам’яті, але складніша модель володіння для цієї реалізації	Більше відповідальності на рівні реалізації, але прозоре представлення даних і контроль пам’яті

2.3 Керування пам’яттю та вибір allocator-а

У Zig allocator є частиною API, а не прихованою деталлю середовища виконання. Це вплинуло на структуру модуля, оскільки екземпляр документа отримує allocator під час створення, зберігає його всередині та використовує для масивів елементів, байтового буфера, синхронізаційних повідомлень і тимчасових структур.

Для тестів використовується `std.testing.allocator`. Його цінність полягає не в швидкодії, а в тому, що він допомагає виявляти помилки володіння пам’яттю під час виконання тестів. Це добре поєднується з підходом, за якого кожен метод, що повертає виділений буфер, явно визначає відповідальність викликача за звільнення цієї пам’яті.

Для WebAssembly-збірки використовується `std.heap.wasm_allocator`. Через це буфери, з якими взаємодіє JavaScript-код, розміщуються в лінійній пам'яті WebAssembly. JavaScript-код копіює байти у wasm-пам'ять, викликає експортовану функцію, а після використання результату звільняє буфер через експортовану функцію `free`.

Такий підхід робить межу між JavaScript і WebAssembly більш ручною, але водночас передбачуваною, тому що немає прихованого перетворення об'єктів, не потрібно переносити внутрішні структури у JavaScript, а володіння пам'яттю видно із сигнатур функцій.

2.4 Стиль реалізації та інженерні правила

Під час розробки частково використовувалися ідеї Tiger Style – стилю програмування, сформульованого в проєкті TigerBeetle [36]. У цій роботі йдеться не про повне дотримання всіх правил Tiger Style, а про застосування окремих принципів, корисних для побудови передбачуваної структури даних.

У реалізації це проявляється в кількох рішеннях. Дескриптори елементів зберігаються як `u32`-індекси, текстові байти розміщуються у спільному буфері, а декодування повідомлень супроводжується перевірками формату, розміру та допустимості полів. Для перевірки внутрішньої структури документа передбачено `debug.checkIntegrity`, а fuzz-тести доповнюють перевірки інваріантів і допомагають знаходити помилки у складних сценаріях синхронізації.

Водночас ця реалізація не є safety-critical системою рівня TigerBeetle. Пам'ять не виділяється повністю на старті, документ може рости під час редагування, а динамічні масиви є нормальною частиною моделі. Тому Tiger Style використовується радше як набір інженерних орієнтирів.

2.5 Використання AI-інструментів у процесі розробки

Окремо варто зазначити роль AI-інструментів у процесі виконання роботи. Під час розробки використовувався Codex – coding agent, який може читати, редагувати й запускати код у середовищі розробки [37].

У межах цього проєкту Codex не є частиною розробленого модуля та не бере участі у синхронізації документів. Його роль була допоміжною й інструмент використовувався для навігації кодовою базою, планування змін, редагування окремих фрагментів і перевірки проміжних результатів. Альтернативні coding-agent інструменти детально не порівнювалися, оскільки вибір AI-інструмента не є предметом дослідження і не впливає на архітектуру розробленого CRDT-модуля.

ВИСНОВОК ДО РОЗДІЛУ 2

У цьому розділі було розглянуто можливі технологічні рішення для реалізації розроблюваного модуля. Під час вибору мови реалізації ядра порівнювалися JavaScript/TypeScript, Rust і Zig. JavaScript і TypeScript забезпечують просту інтеграцію з вебсередовищем, однак є менш придатними для реалізації низькорівневого CRDT-ядра з явним контролем структури даних і пам’яті. Rust має переваги щодо безпеки пам’яті та продуктивності, однак у межах цієї роботи міг би перенести значну частину складності в роботу з моделлю володіння та запозичення.

Після проведеного аналізу було прийнято рішення реалізувати власне ядро модуля мовою Zig з мінімальною кількістю залежностей. Такий вибір дозволяє поєднати явне керування пам’яттю, перевірку інваріантів, розширену систему тестування та можливість компіляції у WebAssembly. Для зовнішньої межі модуля використовується TypeScript, оскільки він добре підходить для опису публічного API та інтеграції з вебзастосунком.

Отже, обраний технологічний підхід дозволяє реалізувати власне ядро синхронізації текстових даних із явним контролем пам’яті та зручною інтеграцією з вебсередовищем.

РОЗДІЛ 3
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ МОДУЛЯ СИНХРОНІЗАЦІЇ

3.1 Загальний огляд реалізації

У цьому розділі описано реалізацію текстового CRDT. Вона натхненна Yjs та родиною CRDT для послідовностей YATA [20], але не є прямим перенесенням Yjs і не намагається бути сумісною з форматом оновлень Yjs. Натомість реалізація використовує ідеї, корисні для цього проєкту, і застосовує їх до меншої моделі документа.

Таке звуження обсягу є навмисним. Yjs – це зріла бібліотека узагальнених спільних структур даних з підтримкою багатьох типів документів і багатьох крайових випадків. Мета цієї реалізації вужча: побудувати компактне та зрозуміле ядро CRDT, достатнє для браузерного редактора спільного редагування. Тому реалізація зосереджується на механіці, потрібній для спільної роботи з текстом, а не на відтворенні всієї екосистеми Yjs.

Реалізація не потребує центрального сервера, який визначає остаточний порядок операцій. Сервер може використовуватися як транспортний ретранслятор, але сам стан CRDT спроєктований так, щоб репліки могли редагувати локально та обмінюватися оновленнями пізніше. Оновлення безпечно отримувати більше одного разу, а оновлення, що приходять раніше за свої залежності, тимчасово зберігаються і повторно застосовуються після пізніших оновлень.

Усередині видимий документ не зберігається як один змінюваний рядок. Він зберігається як двозв’язна послідовність CRDT-елементів. Частина елементів містить текст, частина – маркери форматування, а видалений вміст лишається як tombstone-маркери, щоб пізніші віддалені операції все ще могли посилатися на стабільні ідентифікатори.

3.2 Модель документа

Найважливіша ідея реалізації полягає в тому, що стан CRDT і видимий текст не є одним і тим самим. Стан CRDT зберігає історію редагування, потрібну для синхронізації, тоді як видимий текст виводиться з цього стану.

Основною одиницею цього стану є `Item`. Елемент може представляти фрагмент тексту, маркер форматування або видалений вміст. Текстові елементи додають символи до видимого документа. Маркери форматування не додають символів і лише змінюють активні атрибути для наступного тексту. Видалені елементи також не з'являються у видимому тексті, але залишаються у структурі, щоб інші репліки могли посилатися на їхні ідентифікатори.

Наприклад, після видалення середини слова внутрішня послідовність усе ще може містити видалений фрагмент: `[text "h"] -> [deleted text "ell"] -> [text "o"]`. Під час відтворення документа алгоритм проходить цю послідовність і виводить лише видимі елементи: `"ho"`.

На відміну від звичайного рядкового редактора, таке представлення створює додаткові накладні витрати зберігання, але водночас дає змогу виконувати злиття та версіонування на рівні структури даних [38].

Та сама послідовність також зберігає форматування тексту. Операція форматування представлена вставленням маркерів у послідовність. Під час відтворення реалізація відстежує поточні активні атрибути форматування та додає їх до наступних текстових елементів. Це зберігає звичайний текст і форматування в одній упорядкованій CRDT-структурі.

Отже, внутрішня послідовність є джерелом істини, а звичайний текст і форматований текст є похідними представленнями.

3.3 Ідентичність елементів

Кожен елемент потребує глобально унікального ідентифікатора. Один простий спосіб створювати унікальні ідентичності без центрального координатора – використовувати UUID. UUIDv7 є особливо привабливим, оскільки містить впорядковане за часом поле, похідне від Unix epoch timestamp, що

робить ідентифікатори приблизно сортованими за часом створення [39]. Однак повний UUID для кожного елемента додавав би 16 байтів метаданих ідентифікатора. Крім того, фізичний час не є надійним джерелом причинного порядку в розподілених системах, оскільки годинники можуть розходитися.

Логічні годинники є поширеним розв'язанням задачі впорядкування подій у розподілених системах [40]. Замість того, щоб запитувати фізичний годинник, коли відбулася подія, вони рахують події. Годинники Лампорта призначають кожній події логічну часову позначку і можуть бути доповнені ідентифікатором репліки, щоб отримати детермінований повний порядок ($\prec$) операцій [41, с. 560-562]. Цей порядок визначається так:

$$(a \prec b) \iff (C(a) < C(b) \vee (C(a) = C(b) \wedge r(a) < r(b))), \qquad (3.1)$$

де $C(a)$ – часова позначка Лампорта для операції a, а $r(a)$ – унікальний ідентифікатор репліки, що створила операцію a. Цей повний порядок узгоджений з відношенням happened-before ($\rightarrow$). Якщо операція a відбулася перед операцією b, то годинники Лампорта гарантують $C(a) < C(b)$, а отже a буде впорядкована перед b: $(a \rightarrow b) \Rightarrow C(a) < C(b) \Rightarrow (a \prec b)$.

У стандартному алгоритмі годинників Лампорта (рис. 3.1) кожна репліка збільшує локальний годинник перед створенням події.

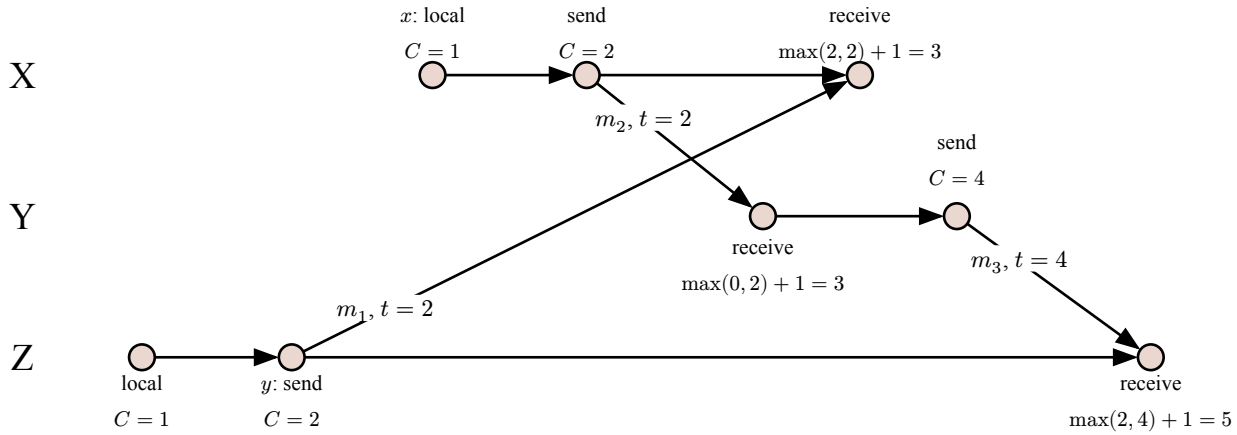


Рисунок 3.1 — Оновлення годинників Лампорта

Після отримання повідомлення отримувач встановлює свій годинник у максимум між поточним локальним значенням і часовою позначкою повідомлення, а потім збільшує його. Наприклад, репліка Z отримує m_3 з часовою

позначкою 4, тоді як її локальний годинник дорівнює 2, тому наступна часова позначка дорівнює $\max(2, 4) + 1 = 5$.

Та сама діаграма також показує, чому часові позначки Лампорта не можна використовувати як повну перевірку причинності. Події x та y мають впорядковані часові позначки, $C(x) = 1$ та $C(y) = 2$, але немає ланцюга локальних кроків або повідомлень від x до y. Тому $C(x) < C(y)$ не доводить, що x спричинила y.

Ця реалізація використовує ту саму інтуїцію, але не повний глобальний годинник Лампорта. Кожен елемент ідентифікується парою `Id` (рис. 3.2).

```
pub const ClientId = u64;
pub const Clock = u64;
pub const Id = struct { client: ClientId, clock: Clock, };
```

Рисунок 3.2 — Ідентифікатор елементів

`ClientId` ідентифікує репліку, яка створила елемент. `Clock` є монотонно зростальним лічильником у локальній історії цієї репліки. Це схоже на приклад G-Counter, де кожна репліка збільшує лише власну компоненту, а синхронізація об'єднує знання про всі репліки.

Один текстовий елемент може покривати більше ніж одне значення лічильника. Наприклад, якщо клієнт `1` вставляє `"hello"` у порожній документ, реалізація може зберегти це як один елемент з ідентифікатором `{1, 0}` і довжиною `5`. Цей елемент володіє діапазоном лічильників `{1, [0..4]}`. Наступний елемент, створений тим самим клієнтом, почнеться з лічильника `5`.

Реалізація припускає, що активні репліки використовують унікальні ідентифікатори клієнтів, і не надає генерації `ClientId`. У реальному застосунку ідентифікатори клієнтів можуть походити з облікового запису або з іншого механізму на рівні застосунку. Отже, CRDT-ядро не потребує центрального координатора для синхронізації, але коректність роботи все одно залежить від унікальності `ClientId`, яку забезпечує рівень застосунку. Якщо дві активні репліки використають однаковий `ClientId`, їхні діапазони лічильників можуть перетнутися, і CRDT більше не зможе коректно розрізняти їхні історії.

3.4 Внутрішнє представлення

Основною структурою даних, що зберігає документ, є `TextImpl`. Вона містить CRDT-послідовність, довжину видимого тексту, байтове сховище та індекс лічильників за клієнтами, потрібний для синхронізації.

Порядок документа представлений як двозв'язний список елементів. Кожен елемент зберігає локальні дескриптори `left` і `right`, а `TextImpl` зберігає дескриптори `start` і `end` для всієї послідовності. Цей зв'язний порядок використовується під час відтворення документа.

Елементи не зберігаються в пам'яті в порядку зв'язаного списку. Натомість вони зберігаються в масиві, а `ItemHandle` є індексом у цьому масиві. Це дозволяє зробити зв'язки компактними, оскільки такий локальний дескриптор має тип `u32`, а не є вказівником або стабільним ідентифікатором елемента.

Отже, реалізація використовує два різні способи посилатися на елемент:

- `ItemHandle` є локальним для однієї репліки і використовується всередині для зв'язків між елементами;
- `Id` є стабільним між репліками і використовується у повідомленнях синхронізації.

Це розрізнення потрібне, тому що дві репліки можуть зберігати той самий логічний елемент у різних позиціях масиву. Тому оновлення, які описують місце вставлення елемента, не можуть використовувати локальні дескриптори; вони мають посилатися на стабільні ідентифікатори сусідніх елементів.

Вміст елементів також зберігається компактно. Текст і рядки атрибутів розміщуються в байтовому буфері, яким володіє `TextImpl`. Елемент не володіє окремим рядком; натомість він зберігає діапазон у цьому буфері. Текстовий зріз зберігає і байтовий діапазон, і логічну довжину в скалярних значеннях Unicode. Це дає змогу зберігати текст у UTF-8, але надавати індекси в одиницях видимих символів.

Для синхронізації реалізація використовує `StructStore`. Він групує дескриптори елементів за клієнтом, який їх створив, і зберігає елементи кожного

клієнта впорядкованими за `Clock`. Для кожного клієнта діапазони лічильників мають бути неперервними. Завдяки цьому впорядкуванню сховище може знаходити елемент, що містить заданий ідентифікатор `{client, clock}`, за допомогою бінарного пошуку. `StructStore` використовується, щоб відповідати на кілька питань синхронізації:

- який лічильник має отримати наступний локальний елемент;
- чи вже відомий ідентифікатор елемента;
- який локальний дескриптор містить заданий ідентифікатор `{client, clock}`;
- який вектор стану потрібно надіслати іншій репліці;
- яких елементів бракує іншій репліці відносно її вектора стану.

Загалом внутрішнє представлення поєднує три погляди на той самий стан: зв'язний список визначає порядок документа, масив елементів забезпечує компактне локальне зберігання, а `StructStore` індексує елементи за клієнтом і лічильником для синхронізації.

3.5 Локальне редагування

Локальні операції редагування застосовуються на локальній репліці миттєво. Метадані CRDT створюються разом із локальною зміною, тому пізніше операцію можна закодувати та надіслати іншим реплікам.

3.5.1 Пошук позицій

Публічний API редагування використовує видимі текстові індекси. Ці індекси рахують скалярні значення Unicode у відтвореному документі, а не байти і не внутрішні елементи. Тому видалені елементи та маркери форматування потрібно ігнорувати під час перетворення публічного індексу у внутрішню позицію.

Наприклад, якщо внутрішня послідовність має вигляд `[text "h"] -> [deleted text "ell"] -> [text "o"]`, тоді видимий індекс `1` вказує на позицію між `"h"` і `"o"`, хоча між ними всередині існує видалений елемент.

Позиція представлена як проміжок між двома дескрипторами елементів: елементом зліва та елементом справа. Якщо індекс операції потрапляє всере-

дину текстового елемента, цей елемент спочатку розділяється. Наприклад, вставка за індексом 2 всередині `[text "hello"]` розділяє елемент на `[text "he"]` і `[text "llo"]`, створюючи між ними чистий проміжок для вставки.

Новий елемент після цього можна зв'язати з цим проміжком. Розділення не копіює байти рядка; воно лише змінює метадані зрізу наявного елемента і створює інший елемент, що вказує на той самий байтовий буфер. Розділення виконується на межах скалярних значень UTF-8, тому публічні індекси символів лишаються узгодженими з внутрішнім байтовим представленням.

3.5.2 Вставлення

Звичайне вставлення тексту виконує невелику послідовність кроків:

- перевіряє, що індекс належить видимому документу;
- перевіряє вставлені байти як UTF-8 і обчислює їхню довжину в скалярних значеннях Unicode;
- знаходить проміжок у зв'язному списку, що відповідає видимому індексу;
- додає вставлені байти до байтового буфера;
- створює новий текстовий елемент з наступним ідентифікатором `{client, clock}`;
- зберігає стабільні ідентифікатори сусідніх елементів як origin-ідентифікатори;
- додає елемент до `StructStore`;
- зв'язує елемент у порядку документа.

Важливий момент полягає в тому, що вставлення не переписує весь документ. Воно створює новий елемент і розміщує його в послідовності.

Розглянемо такі операції над порожнім документом:

- `insert(0, "Hello")`
- `insert(5, " world")`
- `insert(5, ",")`

У таблиці 3.1 елемент коми додається як дескриптор `h2`, тому в масиві елементів він стоїть після `h0` і `h1`. Однак видимий документ не читається у порядку масиву. Він читається за порядком зв'язного списку: `h0 -> h2 -> h1`, що відтворюється як `"Hello, world"`.

Таблиця 3.1 — Метадані елементів після вставлення коми

дескриптор	id	зріз вмісту	зв'язки
`h0`	`{1,0}`	`bytes[0..5] = "Hello"`	`left=null, right=h2`
`h1`	`{1,5}`	`bytes[5..11] = " world"`	`left=h2, right=null`
`h2`	`{1,11}`	`bytes[11..12] = ","`	`left=h0, right=h1`

Рисунок (рис. 3.3) показує той самий стан з іншого погляду. Зв'язаний список визначає порядок документа, тоді як кожен елемент посилається на зріз у спільному байтовому буфері. Байтовий буфер зберігає дані в порядку додавання, тому його фізичне розташування не обов'язково збігається з видимим порядком документа.

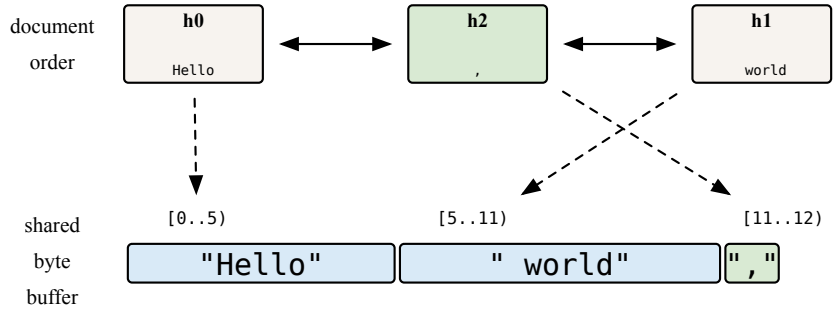


Рисунок 3.3 — Порядок документа і спільний байтовий буфер після вставлення коми

Елемент також зберігає ліве й праве походження. Вони не є тим самим, що поточні зв'язки `left` і `right`, оскільки ці зв'язки можуть змінитися пізніше, коли поруч вставляться інші елементи, тоді як походження залишаються стабільними. Рисунок (див. рис. 3.3) не зображує лівого і правого походження для спрощення. Детальніше ця інформація використовується під час інтеграції віддалених змін.

3.5.3 Видалення

Видалення також починається з видимих індексів. Реалізація перетворює вказаний видимий діапазон на внутрішні межі елементів, за потреби розділяє

текстові елементи на межах і після цього позначає відповідні видимі текстові елементи як видалені.

Наприклад, видалення `"ell"` з `"hello"` може утворити внутрішню послідовність `[text "h"] -> [deleted text "ell"] -> [text "o"]`.

Видима довжина зменшується, але видалений елемент залишається доступним за своїм стабільним діапазоном ідентифікаторів. Це необхідно, оскільки віддалене оновлення все ще може посилатися на елемент, який уже було видалено локально.

3.6 Інтеграція віддалених змін

Коли репліка отримує оновлення від іншої репліки, вона не застосовує операцію за видимим індексом. Видимі індекси є локальними та тимчасовими: поки репліки не мають оновлень одна одної, той самий видимий індекс може вказувати на різні внутрішні позиції. Тому віддалена інтеграція ґрунтується на стабільних ідентифікаторах елементів (ідентифікаторах походження, або origin-ідентифікаторах).

Віддалений елемент містить:

- власний ідентифікатор `{client, clock}` і довжину;
- стабільний ідентифікатор елемента, що був зліва під час створення;
- стабільний ідентифікатор елемента, що був справа під час створення;
- вміст, який може бути текстом або маркером форматування.

Лівий і правий ідентифікатори називаються origin-ідентифікаторами. Вони описують логічний проміжок, у який елемент було вставлено. Локальне вставлення записує ці origin-ідентифікатори, тому що локальні дескриптори мають сенс лише всередині однієї репліки, тоді як віддаленим реплікам потрібні стабільні ідентифікатори для відновлення того самого контексту вставлення.

Наприклад, якщо кома вставляється між `"Hello"` та `" world"`, віддалене оновлення не каже «вставити після дескриптора `h0`», бо `h0` існує лише на відправнику. Натомість оновлення повідомляє, що новий елемент було вставлено

після останнього ідентифікатора `"Hello"` і перед першим ідентифікатором `" world"`. Отримувач використовує власний `StructStore`, щоб знайти локальні дескриптори, які містять ці ідентифікатори, за потреби розділяє елементи на потрібних межах лічильника і зв'язує віддалений елемент в отриманий проміжок.

Перед інтеграцією віддаленого елемента реалізація перевіряє, чи елемент уже відомий і чи доступні його залежності, порівнюючи наступний очікуваний лічильник для цього клієнта. Якщо один з origin-ідентифікаторів ще невідомий, елемент поки не можна зв'язати з документом. Шар синхронізації зберігає такі оновлення як відкладені та повторює спробу пізніше.

3.6.1 Конкурентні вставлення

Найважливіший конфліктний випадок виникає, коли дві репліки конкурентно вставляють у той самий логічний проміжок. Наприклад, дві репліки можуть одночасно вставити текст на початку порожнього документа (рис. 3.4).

```
replica A inserts "X"
replica B inserts "Y"
```

Рисунок 3.4 — Приклад конкурентних вставлень

Обидва вставлення не мають ні лівого, ні правого origin-ідентифікатора, тому вони спрямовані в той самий проміжок. Якщо кожна репліка вставлятиме віддалені елементи лише в порядку отримання повідомлень, фінальний порядок документа може відрізнятися між репліками, що порушить збіжність.

Щоб уникнути цього, реалізація використовує детерміноване правило впорядкування, визначене в роботі про YATA [20]. Мета полягає не в тому, щоб визначити, який користувач редагував першим, оскільки конкурентні операції не мають надійного фізичного порядку. Мета – гарантувати, що кожна репліка вибере той самий порядок з того самого набору елементів.

У результаті, якщо всі репліки отримають ті самі конкурентні вставлення, вони зрештою збіжаться, навіть якщо оновлення прийдуть у різному порядку.

3.7 Синхронізація

Синхронізація відокремлена від транспортного шару. CRDT не залежить від того, чи байти оновлень передаються через WebRTC, WebSocket або локальне сховище. API синхронізації потребує лише двох повідомлень: вектора стану та оновлення.

3.7.1 Вектори стану

Вектор стану – це компактний підсумок того, що репліка вже знає. Для кожного клієнта він зберігає наступний лічильник, який ця репліка очікує від цього клієнта. Наприклад, якщо репліка має елементи клієнта `7`, що покривають лічильники `[0, 12]`, тоді її вектор стану містить наступний очікуваний елемент: `"client 7": 13`.

Це означає, що лічильники нижче `13` від клієнта `7` уже відомі, а лічильник `13` є наступною пропущеною позицією в історії цього клієнта. Таке компактне представлення коректне завдяки інваріанту `StructStore`, що діапазони лічильників для кожного клієнта зберігаються неперервними.

Вектори стану малі, тому що вони залежать від кількості клієнтів, а не від розміру документа. Реалізація будує цей підсумок безпосередньо з `StructStore`.

3.7.2 Оновлення

Оновлення кодується відносно цільового вектора стану. Якщо цільовий вектор стану не задано, реалізація кодує весь відомий стан документа. Якщо цільовий вектор стану задано, реалізація надсилає лише ті діапазони елементів, яких бракує цільовій репліці.

Для кожного клієнта кодувальник знаходить перший елемент, діапазон лічильників якого не повністю покритий цільовим вектором стану. Якщо цільова репліка вже має першу частину текстового елемента, оновлення може початися з середини цього елемента і надіслати лише решту суфікса. Це можливо, тому що кожен текстовий елемент володіє неперервним діапазоном лічильників і зберігає свою логічну довжину.

Кодування оновлень як JSON було б неефективним, оскільки дані спільного редагування містять багато повторюваних метаданих. Узагальнені бінарні формати, такі як Cap'n Proto [42], компактніші, але спеціальний бінарний формат може ще більше зменшити обсяг метаданих, використовуючи специфічні залежності структури CRDT-оновлень [43].

Як показано на рисунку (рис. 3.5), оновлення складається з невеликої оболонки, блоків змінених елементів і набору видалень.

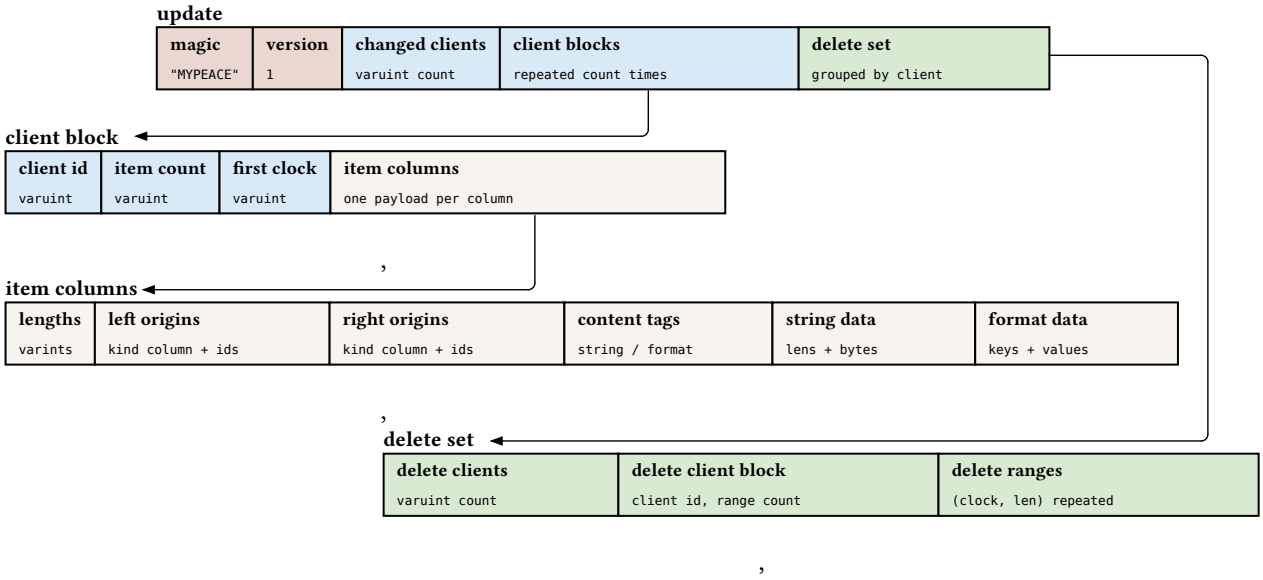


Рисунок 3.5 — Бінарна структура оновлення

Оболонка містить магічні байти та номер версії. Цілочисельні значення кодуються за допомогою беззнакового LEB128-подібного кодування цілих чисел змінної довжини, що відповідає тому самому загальному підходу, який використовується в бінарному форматі WebAssembly [44].

Змінені елементи групуються за клієнтом, а їхні поля записуються стовпцями, а не як незалежні рядкові записи. Таке стовпцеве представлення полегшує стиснення повторюваних метаданих.

Наприклад, нехай оновлення містить 42-символьний текст `The quick brown fox jumps overthe lazy dog`, представлений як 42 послідовні односимвольні елементи, створені клієнтом 7. Пряме поелементне кодування, яке зберігає і `ClientId`, і `Clock` як фіксовані 64-бітні значення для кожного елемента, використало б $42 \cdot \frac{64+64}{8} = 672$ байти лише для ідентифікаторів елементів.

Натомість реалізація записує один блок клієнта (рис. 3.6).

```
client id: 7
item count: 42
first clock: C
item columns: ...
```

Рисунок 3.6 — Схема блока клієнта в оновленні

Ідентифікатор клієнта зберігається один раз, і всі елементи всередині блока успадковують цього клієнта. Їхні лічильники відновлюються з першого лічильника та стовпця довжин елементів. Для односимвольних елементів стовпець довжин містить значення `1`, повторене `42` рази. Це можна подати за допомогою кодування довжин повторів (RLE) [45, с. 20-24] як один повтор, наприклад `(value: 1, run_len: 42)`.

Оскільки малі цілі числа компактні в беззнаковому LEB128-кодуванні, метадані, потрібні для відновлення ідентифікаторів, у цьому прикладі займають приблизно `5` байтів без урахування службового конверта та інших стовпців. Точна економія залежить від даних і накладних витрат стовпців, тому кодувальник використовує RLE лише тоді, коли воно менше за сирі стовпці.

Інформація про видалення кодується окремо як множина видалень (`delete set`). Множина видалень групує видалені діапазони лічильників за клієнтом, тому кілька сусідніх видалень можна представити як діапазони, а не як окремі ідентифікатори елементів.

Цей розділ не описує повну бінарну структуру. Важлива проєктна ідея полягає в тому, що синхронізація ґрунтується на стабільних ідентифікаторах елементів та векторах стану, і ці метадані можна компактно закодувати.

3.7.3 Доставка з порушенням порядку

Оновлення можуть прийти раніше за свої залежності. Наприклад, репліка може отримати видалення для текстового діапазону до того, як вона отримала вставлення, яке створило цей діапазон. Так само віддалений елемент може посилатися на лівий або правий origin-ідентифікатор, який локально ще невідомий.

Коли це відбувається, реалізація не відкидає оновлення одразу. Вона зберігає його в обмеженому буфері відкладених оновлень. Після успішної інтеграції будь-якого наступного оновлення відкладені оновлення перевіряються повторно. Щойно їхні залежності стають доступними, вони можуть бути застосовані звичайним способом.

Віддалені операції є ідемпотентними. Повторна інтеграція вже відомого елемента не має ефекту, а повторне застосування того самого видалення лише залишає елемент видаленим. Це означає, що транспортний рівень не зобов’язаний забезпечувати причинний порядок або доставку рівно один раз. Якщо всі репліки зрештою отримають той самий набір оновлень, стан CRDT усе одно збіжиться.

3.8 Форматований текст

Форматування представлене всередині тієї самої CRDT-послідовності, що й текст. Реалізація не підтримує окреме інтервальне дерево для стилів. Натомість вона вставляє у зв’язний список маркери форматування, які не враховуються у видимій довжині. Ці елементи мають ідентифікатори та origin-ідентифікатори, як і текстові елементи, тому можуть синхронізуватися та впорядковуватися тим самим механізмом CRDT, але не додають символів до видимого тексту.

Маркер форматування змінює активне значення одного ключа атрибута для тексту, який іде після нього. Значення може бути рядком, що встановлює атрибут, або `null`, що очищає його. Наприклад, застосування `bold=true` до `"world"` у `"Hello, world"` представлено розміщенням маркера перед форматованим діапазоном і маркера очищення після нього:

```
[Hello, ] [bold=true] [world] [bold=null]
```

У таблиці 3.2 подані метадані елементів для цього прикладу.

Таблиця 3.2 — Метадані елементів після форматування діапазону

дескриптор	тип	зріз вмісту	видимий
`h0`	string	`bytes[0..5] = "Hello"`	так
`h2`	string	`bytes[11..12] = ","`	так

Кінець таблиці 3.2

дескриптор	тип	зріз вмісту	видимий
h1	string	bytes[5..6] = " "	так
h4	format	key = bytes[12..16] = "bold", value = bytes[16..20] = "true"	ні
h3	string	bytes[6..11] = "world"	так
h5	format	key = bytes[20..24] = "bold", value=null	ні

Рисунок (рис. 3.7) показує той самий стан як упорядковану послідовність. Маркери форматування є звичайними CRDT-елементами з ідентифікаторами та байтовими зрізами, але вони пропускаються під час створення звичайного видимого тексту.

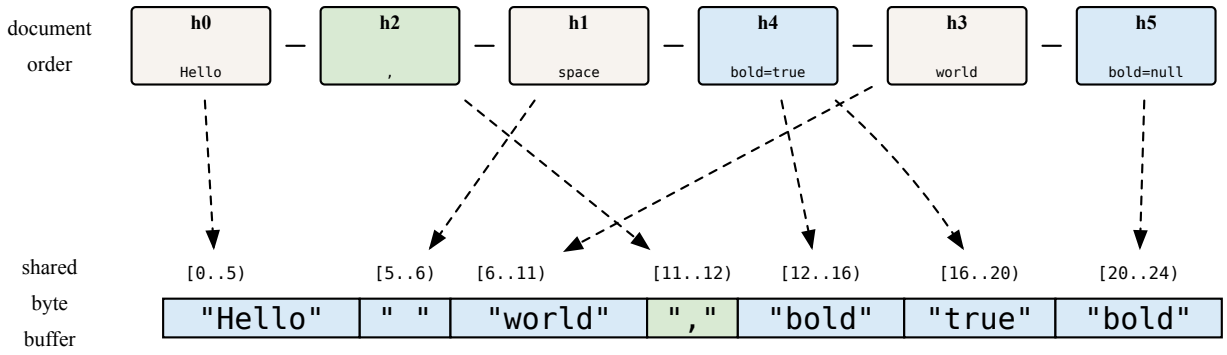


Рисунок 3.7 — Маркери форматування в послідовності елементів

Відтворення проходить послідовність зліва направо. Коли алгоритм зустрічає маркер форматування, він оновлює поточні активні атрибути. Коли він зустрічає невидалений текстовий елемент, він виводить текст з атрибутами, активними у цей момент. Той самий обхід може створювати звичайний текст, ігноруючи активні атрибути, або форматовані Delta-операції [46], додаючи активні атрибути до кожного виведеного текстового фрагмента.

Маркери відновлення потрібні лише під час форматування діапазону. Якщо після діапазону атрибут мав попереднє значення, реалізація відновлює це значення; інакше вона вставляє `null`-маркер для очищення атрибута. Це дає змогу природно обробляти перекривне форматування. Наприклад, застосуван-

ня синього форматування до середини вже червоного діапазону впливає лише на вкладений діапазон, а текст після нього повертається до червоного (рис. 3.8).

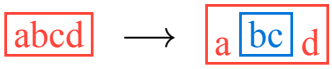


Рисунок 3.8 — Відновлення форматування після застосування вкладеного діапазону

Маркери форматування можуть стати надлишковими після пізніших редагувань. Наприклад, форматований діапазон може бути видалений, або два сусідні маркери можуть встановлювати те саме значення. Тому реалізація надає `compact()`, яка видаляє надлишкові маркери форматування і об'єднує сусідні рядкові елементи, коли це безпечно. Компактизація є явною операцією обслуговування, а не кроком після кожного редагування, тому що локальне редагування має лишатися швидким, навіть якщо внутрішнє представлення не одразу мінімальне.

3.9 Інтерфейс WebAssembly

Інтерфейс WebAssembly навмисно мінімальний і складається з простих експортованих функцій. JavaScript-код не звертається до Zig-структур напряму. Натомість він створює документ і отримує непрозорий дескриптор, який потім передається назад до експортованих функцій.

Текст і дані оновлень перетинають межу як пари вказівник-довжина, оскільки прості WebAssembly-експорти не можуть напряму отримувати JavaScript-рядки [47]. JavaScript спочатку копіює байти в пам'ять WebAssembly, а потім передає вказівник і довжину до експортованої функції. Вихідні дані повертаються так само, а обгортка надає функції виділення та звільнення пам'яті, щоб JavaScript міг керувати цими тимчасовими буферами.

Операції, які можуть завершитися помилкою, повертають компактні цілочислові коди помилок на межі WebAssembly. Усередині Zig-реалізація все ще використовує error union, але експортований API перетворює їх у представлення, яке просто обробляти з JavaScript.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі описано проєктування та реалізацію CRDT-модуля для синхронізації текстових даних. Основою реалізації є внутрішня послідовність `Item`, яка відокремлює видимий текст від CRDT-стану. Такий підхід дає змогу зберігати не лише актуальний вміст документа, а й метадані, необхідні для синхронізації: стабільні ідентифікатори, посилання на походження, tombstone-маркери та маркери форматування.

Для ідентифікації елементів використано пару `{client, clock}`. Вона дає змогу створювати стабільні ідентифікатори без центрального координатора, групувати історію за клієнтами та будувати вектори стану. Водночас локальне представлення документа використовує компактні `ItemHandle`-дескриптори, які є індексами в масиві. Це розділення дозволяє поєднати стабільні міжреплікові ідентифікатори з ефективним локальним зберіганням.

Локальні операції вставлення, видалення та форматування виконуються через перетворення видимих індексів у внутрішні межі елементів. Віддалені оновлення, навпаки, інтегруються не за видимими індексами, а за стабільними ідентифікаторами та origin-ідентифікаторами. Це дозволяє застосовувати оновлення на репліках, які тимчасово мають різний видимий стан, і зберігати збіжність після отримання однакового набору змін.

Механізм синхронізації побудовано на векторах стану та бінарному форматі оновлень. Вектор стану компактно описує, які діапазони лічильників уже відомі репліці, а оновлення передає лише відсутні елементи та множину видалень. Для зменшення обсягу повідомлень елементи групуються за клієнтами, поля записуються стовпцями, а повторювані значення можуть кодуватися за допомогою RLE та LEB128-подібних змінної довжини цілих чисел.

Також у розділі описано підтримку простого форматованого тексту через маркери атрибутів і мінімальний WebAssembly-інтерфейс. У результаті реалізація формує цілісне CRDT-ядро, яке підтримує локальне редагування,

інтеграцію віддалених змін, компактну синхронізацію та використання у веб-середовищі через TypeScript-обгортку.

РОЗДІЛ 4
ТЕСТУВАННЯ, АНАЛІЗ ТА ОБМЕЖЕННЯ РОЗРОБЛЕНОГО МОДУЛЯ

Уся реалізація, fuzz-тести, тести продуктивності та допоміжні інструменти розміщені в одному репозиторії на GitHub erotourtes/mirage. Цей розділ описує приклади використання, підхід до тестування, результати експериментального аналізу та обмеження поточної реалізації.

4.1 Приклади використання та інструкція користувача

Розроблений модуль Mirage може використовуватися як Zig-бібліотека або як WebAssembly-модуль через TypeScript-обгортку. Оскільки Zig добре інтегрується з C і може експортувати функції через C ABI, ядро не обмежене лише Zig- або JavaScript-середовищем. У цій роботі практично реалізовано саме WebAssembly-напрям. Для вебсценарію основною точкою входу є workspace-пакет `@mirage/wasm`, розміщений у директорії `js/packages/mirage`. Він приховує низькорівневу роботу з WebAssembly-пам'яттю, вказівниками та кодами помилок і надає типізований API.

Низькорівневий WebAssembly-інтерфейс описано у файлі `mirage.d.ts`. На цьому рівні рядки й оновлення передаються як пари «вказівник-довжина», а буфери, які повертає WebAssembly, потрібно звільняти вручну. TypeScript-обгортка виконує ці дії автоматично. Вона кодує рядки, копіює байти у WebAssembly-пам'ять, читає результат, перевіряє коди помилок і звільняє тимчасові буфери.

Основні типи публічного API такі:

- `loadMirage()` – завантажує `mirage.wasm` і повертає фабрику `Mirage`;
- `createDocument(clientId)` – створює документ для репліки з унікальним ідентифікатором клієнта;

- `insert(index, text)`, `delete(index, length)` та `format(index, length, attribute)` – виконують локальні зміни;
- `toString()` і `toDelta()` – повертають видиме представлення документа;
- `encodeStateVector()` – кодує вектор стану репліки;
- `encodeUpdate(targetStateVector)` – формує оновлення відносно стану іншої репліки;
- `applyUpdate(update)` – застосовує отримане оновлення;
- `destroy()` – звільняє документ у WebAssembly-модулі.

Мінімальний приклад синхронізації двох реплік через TypeScript-обгортку наведено на рисунку (рис. 4.1) (це приклад коду з `js/packages/example`).

```
import { loadMirage } from "@mirage/wasm";

const mirage = await loadMirage();

const a = mirage.createDocument(1);
const b = mirage.createDocument(2);

a.insert(0, "Hello");
a.insert(5, " world", { key: "bold", value: "true" });

const bState = b.encodeStateVector();
const update = a.encodeUpdate(bState);
b.applyUpdate(update);

console.log(b.toString()); // "Hello world"
console.log(b.toDelta());

a.destroy();
b.destroy();
```

Рисунок 4.1 — Синхронізація двох реплік через TypeScript-обгортку

Якщо `encodeUpdate()` викликати без аргументу, буде створено повне оновлення, яке описує відомий стан документа. Якщо передати вектор стану

іншої репліки, результат міститиме лише дані, яких цій репліці бракує. Саме ця схема використовується для синхронізації у демонстраційному редакторі.

Для збирання WebAssembly-артефакту використовується команда з директорії `mirage_zig`, наведена на рисунку (рис. 4.2).

```
$ zig build wasm -Doptimize=ReleaseFast
```

Рисунок 4.2 — Збирання WebAssembly-артефакту

Після цього в `mirage_zig/zig-out/wasm` з'являються `mirage.wasm` і `mirage.d.ts`. TypeScript-обгортка завантажує цей артефакт через `MIRAGE_WASM_URL`. Саме тому важливо зібрати WebAssembly-артефакт перед використанням TypeScript-обгортки.

Для перевірки TypeScript-частини можна виконати команди з директорії `js` (рис. 4.3).

```
$ pnpm install
$ pnpm --filter @mirage/wasm check
$ pnpm --filter @mirage/example start
```

Рисунок 4.3 — Перевірка TypeScript-частини

4.1.1 Демонстраційний редактор

Окремо було реалізовано прототип вебредактора в директорії `js/demo`. Це статична вебсторінка, яку можна відвідати онлайн за посиланням erotourtes.github.io/mirage/collaborative-editor і яка використовує пакет `@mirage/wasm` як локальну workspace-залежність і демонструє типовий сценарій local-first редактора з кількома репліками. Для прототипу було використано Svelte, TypeScript, Vite, Tailwind CSS та pnpm workspaces; ці технології не є предметом окремого дослідження в роботі.

Запуск демонстраційного редактора наведено на рисунку (рис. 4.4).

```
$ cd js
$ pnpm --filter demo dev
```

Рисунок 4.4 — Запуск демонстраційного редактора

Після запуску Vite відкриває локальний HTTP-сервер. Демонстраційний редактор доступний на сторінці `/collaborative-editor`. У ньому кожен клієнт має власний `MirageDocument`, а застосунок імітує обмін оновленнями між кількома репліками всередині браузера (рис. 4.5).

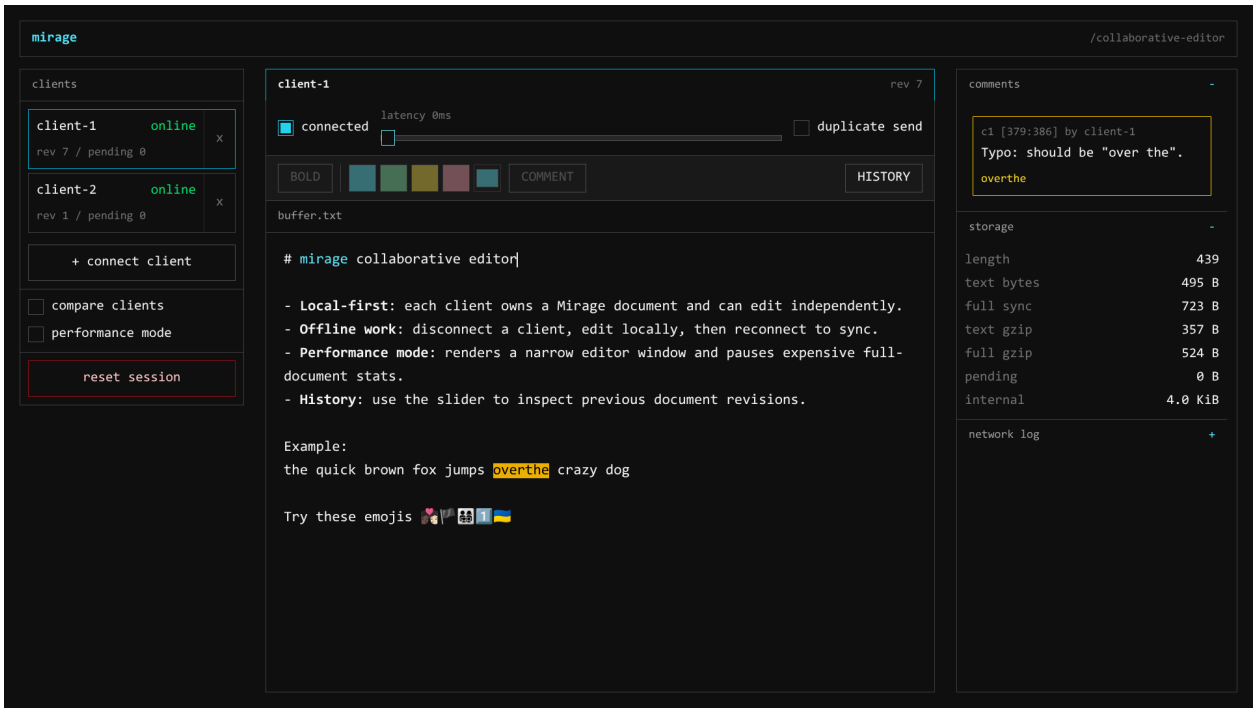


Рисунок 4.5 — Демонстраційний редактор

Після локальної зміни редактор формує інкрементальні повідомлення для інших реплік. Якщо репліка вимкнена або недоступна, оновлення зберігається як відкладене; після повторного підключення синхронізація виконується на рівні редактора.

Демонстраційний редактор перевіряє типові local-first сценарії, необхідні для роботи з текстом. Після отримання всіх оновлень репліки збігаються до однакового тексту й форматування. Він також показує, що маркери форматування можна використовувати не лише для зміни візуального представлення, наприклад застосування жирного шрифту або кольору, а й для збереження метаданих, прикріплених до певного фрагмента. У цьому випадку атрибут `comment` використовується для додавання коментаря до виділеного тексту. Під час роботи з редактором можна помітити, що форматування не завжди зберігає намір користувача, а інколи його застосування неможливе поверх інших атри-

бутів. Це підкреслює обмеження поточної моделі форматування і важливість розуміння її поведінки для подальшого вдосконалення.

Демонстраційний редактор також має перегляд історії документа. Для кожного клієнта можна відкрити модальне вікно `history` і за допомогою повзунка перейти до попередніх ревізій (рис. 4.6):

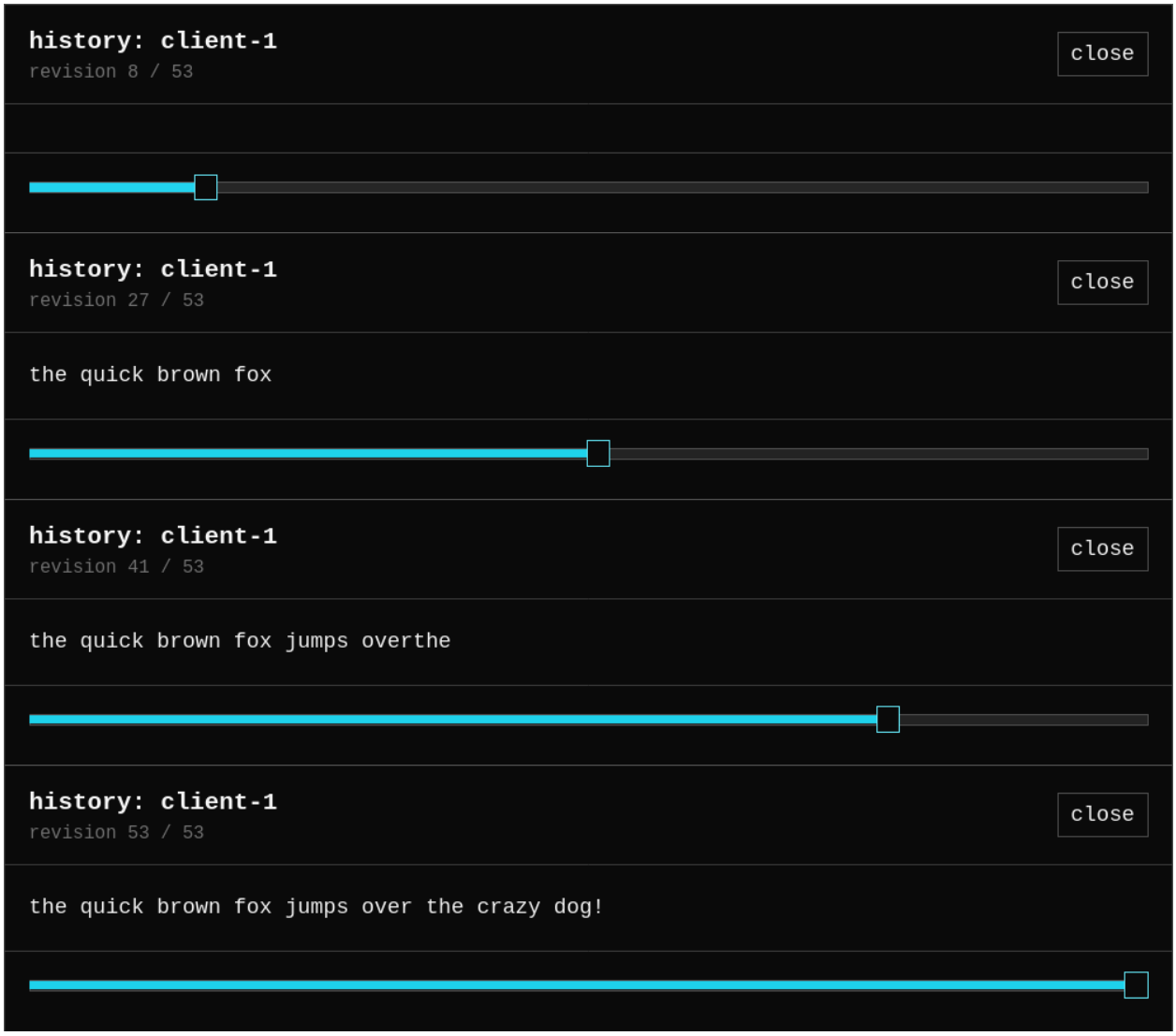


Рисунок 4.6 — Перегляд історії документа

Це не є повноцінним undo/redo, оскільки перегляд історії не створює нової операції та не змінює поточний стан документа. Він лише показує, що ядро зберігає достатньо інформації, щоб відтворити видимий текст і форматування на певній ревізії.

Окремою частиною інтерфейсу є панель статистики, яка показує поточний стан документа, зокрема довжину документа в логічних символах і розмір

видимого тексту в байтах UTF-8. Ці значення корисні для перевірки, тому що різні рівні системи використовують різні одиниці вимірювання тексту. Mirage зберігає текст як UTF-8 і рахує публічні індекси в скалярних значеннях Unicode. JavaScript-рядки у браузері працюють з UTF-16 code units, тому редактор під час роботи перетворює зміщення у скалярні індекси Mirage і навпаки.

Це особливо помітно на складних Unicode-послідовностях. Наприклад, емодзі «👩‍🔬» займає 11 байтів у UTF-8, 5 UTF-16 code units (10 байтів) у JavaScript (`"👩‍🔬".length`) і 3 скалярні значення Unicode в індексації Mirage, але для користувача зазвичай має вигляд одного графемного кластера [48]. Браузер під час навігації курсором часто поводиться саме на рівні графемних кластерів, тому стрілки можуть переносити курсор через усе емодзі як через один видимий символ. Натомість поточна логіка `backspace` і `delete` у редакторі видаляє одне скалярне значення Unicode, а не весь графемний кластер. Через це видалення складної послідовності може залишити частину емодзі, наприклад "🧑" або 🌾". Це не помилка CRDT-ядра, а обмеження демонстраційного редактора, і для повноцінної реалізації потрібна окрема сегментація за графемними кластерами. Ще одним прикладом є «👨‍👩‍👧‍👦», яке можна видалити тільки через 7 натискань `backspace`, тому що саме емодзі скомбіноване із символів «👨 👩 👧 👦» та трьох символів zero-width joiner.

Та сама панель відображає повну кількість байтів, необхідних для синхронізації у звичайному та стисненому (gzip) форматах. Це дає уявлення про розмір даних, які потрібно передавати між репліками або зберігати на диску, а також показує, як ефективно кодуються оновлення. Наприклад, після випадкового редагування повне кодування оновлення зайняло б 1.4 KiB, тоді як стиснене оновлення займає лише 681 B. Це демонструє, що значна частина інформації – це звичайні повторювані шаблони тексту, і метадані CRDT кодуються доволі компактно. Однак у цьому випадку внутрішня репрезентація займає 21.7 KiB (це приблизна кількість оперативної пам’яті, яку використовує документ), що підкреслює різницю між розміром даних для синхронізації і фактичним використанням пам’яті.

У демонстраційному редакторі також передбачено режим продуктивності. У звичайному режимі редактор будує повний `Delta` для всього документа, що зручно для малих і середніх текстів. У режимі продуктивності відображається лише вікно документа навколо поточного курсора через `toDeltaRange()`, а частина дорогих статистичних обчислень вимикається. Це показує, як застосунок може адаптувати роботу з CRDT-ядром для більших документів.

Для повноцінного текстового редактора цього недостатньо, і потрібен віртуальний рендеринг із власним механізмом позиціювання, вимірювання висоти рядків і синхронізації видимої області з позиціями в документі. Однак навіть спрощений режим продуктивності демонструє, що знання структури даних дозволяє зменшити обсяг обчислень під час рендерингу.

Подібний принцип використовується і в інших системах візуалізації великих просторів даних. Наприклад, простір UUID версії 4 містить приблизно $2^{122} \approx 5.3 \times 10^{36}$ можливих значень, тому його можна показувати лише через віртуалізоване представлення, а не через повне створення всіх елементів. Реалізацію цієї ідеї можна дослідити на вебсайті everyuuid.com.

Для зручності прототип зберігає стан сесії в `localStorage` і синхронізує відкриті вкладки через `BroadcastChannel`. Це слід відрізняти від повноцінного онлайн-редактора, тому що демонстраційний застосунок не містить сервера, авторизації, постійного мережевого каналу чи протоколу роботи між різними пристроями. Воно лише моделює кілька реплік у межах одного браузера або кількох вкладок. Такий рівень достатній для перевірки CRDT-ядра й демонструє, як закодований стан документа може використовуватися на рівні застосунку для відновлення сесії та локального обміну між вікнами браузера.

Отже, `js/demo` виконує роль не повноцінного продуктового редактора, а інтерактивної перевірки можливостей модуля.

4.2 Тестування та валідація

Ця реалізація не містить формального машинно перевіреного доведення коректності. Натомість вона перевіряється поведінковими тестами, які охоплюють основні властивості коректності, очікувані від моделі CRDT.

Найнижчим рівнем перевірки є модульні тести. Вони покривають допоміжну поведінку, яку використовують алгоритми вищого рівня.

Найважливішими є рандомізовані fuzz-тести. Вони створюють кілька реплік документа, застосовують випадкові операції, а потім перевіряють, що всі репліки зрештою відтворюють однаковий документ. Такі тести не доводять коректність для кожного можливого виконання, але ефективно виявляють помилки в інтеграції операцій.

Тести запускаються через систему збірки Zig (рис. 4.7).

```
$ cd mirage_zig
$ zig build test
```

Рисунок 4.7 — Запуск тестів Zig

В основі використовується вбудована система тестування Zig [49]. Тести описані безпосередньо у файлах за допомогою блоків `test "..."`, а перевірки виконуються через `std.testing`. Проте стандартний механізм було розширено у `build.zig`, оскільки проєкт має дві групи тестів. Частина тестів розміщена поруч із кодом у `src/lib`, а сценарні та fuzz-тести винесені в окрему директорію `tests` і підключають бібліотеку як високорівневий модуль `mirage_lib`.

Щоб запуск `zig build test` охоплював обидві групи, build-скрипт автоматично обходить потрібні директорії, генерує службові кореневі файли на зразок `all_lib_tests.zig` та `all_external_tests.zig`, імпортує знайдені файли і додає їх до спільного кроку `test`. Це прибирає потребу вручну підтримувати список тестових файлів і дозволяє зберігати зовнішні тести окремо від коду бібліотеки.

Окремо додано кроки для роботи з редактором і налагодженням. Команда `zig build zls-tests` створює окремі тестові артефакти для файлів з директорії

`tests`, щоб ZLS (Zig Language Server) міг коректно бачити їх як самостійні тестові модулі з імпортом `mirage_lib`. Для налагодження окремого файла використовується команда, наведена на рисунку (рис. 4.8).

```
$ zig build debug-test -Ddebug-test-file=tests/text_test.zig
```

Рисунок 4.8 — Збирання Debug-артефакту для окремого тестового файла

За потреби можна додати фільтр конкретного тесту (рис. 4.9).

```
$ zig build debug-test \
  -Ddebug-test-file=tests/text_test.zig \
  -Ddebug-test-filter="re-applying the same update is idempotent"
```

Рисунок 4.9 — Збирання Debug-артефакту з фільтром тесту

Цей крок збирає окремий Debug-артефакт `zig-out/bin/debug_test`, який можна запускати під налагоджувачем. Для файлів у вкладених піддиректоріях `src/lib` build-скрипт додатково генерує `src/lib/debug_test_root.zig`, який імпортує вибраний файл. Це обходить незручність прямого тестування вкладених Zig-файлів як окремих кореневих модулів і зберігає ті самі відносні імпорти, що й під час звичайної збірки бібліотеки.

Для зручності було створено `.vscode/launch.json`, який налаштовує конфігурацію запуску `debug_test` з вибраним файлом і фільтром. Це дає змогу швидко запускати окремі тести під налагоджувачем без необхідності вручну вводити команду збірки.

4.3 Експериментальний аналіз

Для емпіричної оцінки реалізацію було підключено до набору `crdt-benchmarks` [50]. У репозиторії проєкту для цього додано окрему директорію `benchmark`, яка містить Docker-конфігурацію, адаптер Mirage до інтерфейсу набору benchmark-тестів та скрипт для підключення реалізації як додаткового workspace-пакета. Запуск виконується командою з кореневої директорії проєкту (рис. 4.10).

```
$ docker compose -f benchmark/docker-compose.yml run --rm --build
crdt-benchmarks
```

Рисунок 4.10 — Запуск benchmark-тестів

Для синтетичних сценаріїв порівняння проводилося з параметром `N = 6000` для чотирьох реалізацій: Yjs, Ywasm, Automerge та Mirage. Окремо оцінювався сценарій B4 з реальним набором редагувань. Benchmark-набір вимірює час виконання операцій, середній або повний розмір оновлень, час кодування оновлення, розмір закодованого документа, час завантаження документа та орієнтовне використання пам'яті. Значення `0 ms` у результатах слід трактувати як час, менший за роздільну здатність конкретного вимірювання, а не як відсутність витрат.

Вибіркові результати для Mirage наведено у таблиці 4.1. Порівняльні спостереження зроблено на основі повної таблиці результатів benchmark-тестів.

Таблиця 4.1 — Вибіркові результати benchmark-тестів

Сценарій	Час	Оновлення	Розмір документа	Спостереження
B1.1: додавання `N` символів у кінець	49 ms	41 B	6053 B	найменший час серед порівнюваних реалізацій
B1.4: вставлення `N` символів у випадкові позиції	74 ms	45 B	28909 B	швидше за Yjs, Ywasm та Automerge
B1.7: випадкові вставлення та видалення рядків	25799 ms	806 B	67553 B	найгірший сценарій для поточної реалізації

Кінець таблиці 4.1

Сценарій	Час	Оновлення	Розмір документа	Спостереження
B2.2: конкурентне вставлення символів	130 ms	30733 B	61430 B	порівняний час і компактніший update, ніж у Yjs/Ywasm
B2.4: конкурентні вставлення та видалення	780 ms	177308 B	343170 B	повільніше за Yjs, але швидше за Ywasm та Automerge
B3.5: багато клієнтів, що конкурентно вставляють текст	37 ms	66556 B	51167 B	найменший час, але більші оновлення, ніж у Yjs/Ywasm
B4: реальний набір редагувань	5198 ms	-	744082 B	найменший час застосування, але великий розмір закодованого документа

У сценаріях локальних вставлень Mirage показує добрі результати. Послідовне додавання `N` символів займає 49 ms проти 458 ms у Yjs, 323 ms у Ywasm та 628 ms в Automerge.

Розмір оновлень для простих посимвольних сценаріїв у Mirage більший, ніж у Yjs/Ywasm, але значно менший, ніж в Automerge. Для додавання символів у кінець документа Mirage формує оновлення в середньому 41 B, тоді як Yjs і Ywasm – 27 B, а Automerge – 121 B. Основна причина цієї різниці полягає у форматі кодування. Кожне оновлення Mirage містить службовий конверт із

magic-рядком і версією формату, що особливо помітно для односимвольних змін. Натомість Yjs/Ywasm використовують формат Update V2, оптимізований для компактного кодування малих змін. Водночас така компактність ускладнює підтримку формату. Update V2 є окремою версією кодування, не сумісною з початковим форматом оновлень. Через це Yjs підтримує кілька функцій для кодування, застосування та конвертації оновлень різних версій. Це показує практичний компроміс між компактністю повідомлень і складністю підтримки формату синхронізації.

У конкурентних сценаріях результат змішаний. Mirage близький до Yjs/Ywasm для конкурентного вставлення великого рядка, але гірше працює з випадковими позиціями. Імовірна причина полягає в тому, що простий кеш останньої позиції добре допомагає під час послідовного редагування поруч із курсором, але випадковий доступ частіше потребує лінійного обходу.

Найслабше місце поточної реалізації – сценарії з довгою історією видалень. У тесті B1.7, де рядки вставляються і видаляються у випадкових позиціях, Mirage витрачає 25799 ms, тоді як Yjs, Ywasm та Automerge виконують цей тест за 384 ms, 229 ms і 919 ms відповідно. Середній розмір оновлення в цьому сценарії також зростає до 806 B. У цьому сценарії поєднуються дві проблеми. По-перше, простий кеш останнього курсора часто не дає виграшу. По-друге, видалення зберігаються як частина CRDT-стану, де tombstone-маркери залишаються доступними для майбутніх віддалених операцій. Під час видалення елемента реалізація додає його до множини видалень (`delete set`), яка підтримує діапазони об’єднаними та впорядкованими за позицією. Це дає змогу швидше кодувати оновлення, однак, як показує дослідження, таке рішення не є оптимальним. Тому `compact()`, який не є повним збиранням сміття, не допоможе в цій ситуації, і потрібно оптимізувати саме додавання елемента до множини видалень.

У тесті з багатьма клієнтами Mirage показує дуже добрий час виконання: 37 ms проти 73 ms у Yjs, 91 ms у Ywasm та 3260 ms в Automerge. Водночас розмір оновлення становить 66556 B, що більше за 48137 B у Yjs та 48119 B

у Ywasm, але значно менше за 298020 B в Automerge. Отже, поточна модель добре масштабується за часом інтеграції у сценарії великої кількості реплік, але має резерв для подальшого стискання метаданих.

Окремо важливим є тест B4 з реальним набором редагувань. Він містить 259778 операцій над LaTeX-джерелом наукової статті й краще наближає benchmark до типового користування редактором. Mirage відтворює цей набір за 5198 ms, що краще за Yjs із 6971 ms, Ywasm із 42634 ms та Automerge із 16452 ms. Водночас відновлення документа із закодованого стану залишається дорогим: `parseTime`, час створення нового документа із закодованого стану, становить 2745 ms, а розмір закодованого документа після B4 – 744082 B проти 159929 B у Yjs/Ywasm та 129116 B в Automerge.

Показники пам’яті у benchmark-наборі треба інтерпретувати обережно, оскільки порівнюються JavaScript-об’єкти, WebAssembly-пам’ять і різні стратегії allocator-об’єктів та збирача сміття.

Загалом експериментальні результати показують, що Mirage ефективний для локальних вставлень, конкурентного додавання тексту та застосування реалістичних послідовностей операцій. Основні проблемні напрями – велика історія видалень, розмір закодованого документа після тривалого редагування та повільне завантаження такого стану. Тому подальша оптимізація має бути зосереджена на збиранні tombstone-маркерів, компактнішому поданні видалень та покращенні індексування видимих позицій.

Повна таблиця результатів benchmark-тестів для всіх реалізацій доступна в репозиторії проєкту за посиланням erotourtes/mirage#benchmarks.

4.4 Обмеження реалізації

Реалізація навмисно вужча за промислові CRDT-бібліотеки, такі як Yjs або Automerge. Частина обмежень є наслідком свідомого звуження обсягу роботи, а частина була виявлена під час тестування реалізації. Основні обмеження наведено нижче:

- **Один текстовий документ.** Реалізація зосереджена на одному спільному текстовому значенні. Вона не надає загальних CRDT-мап, масивів, XML-подібних структур або піддокументів.
- **Керування скасуванням змін** (undo/redo) відсутнє. Класичне керування скасуванням, яке відновлює попередній стан документа, може не зберігати намір користувачів, тому вимагає додаткової логіки для відновлення наміру.
- **Стан присутності користувачів** відсутній. Реалізація не підтримує відстеження курсорів, виділення або іншої інформації про присутність користувачів, яка часто використовується в спільних редакторах.
- **Індексація за скалярними значеннями Unicode.** Публічні індекси визначені за скалярними значеннями Unicode у коректному UTF-8, а не за графемними кластерами.
- **Обмежена модель форматованого тексту.** Атрибути представлені лише рядковими значеннями або `null`. Цього достатньо для простих метаданих, таких як жирний шрифт, кольори та посилання, але це не є повною моделлю форматованого документа. Значення атрибутів форматування також дублюються у спільному буфері, а не інтернуються в окремому сховищі ключів і значень. Також у цій реалізації не імплементовано повне збереження наміру користувача, адже форматування на межі документа або в специфічних позиціях може поводитися неочікувано після подальших редагувань.
- **Зростання tombstone-маркерів.** Видалені елементи залишаються у структурі, тому що віддалені операції можуть усе ще посилатися на їхні ідентифікатори. Реалізація надає `compact()` для частини безпечного локального очищення, але не реалізує повне збирання сміття для CRDT.
- **Проста синхронізація видалень.** Оновлення наразі містять відому множину видалень. Це робить видалення ідемпотентним і простим для інтеграції, але може бути неефективним для документів з великою історією видалень.
- **Обмежений буфер відкладених оновлень.** Оновлення, що приходять з порушенням порядку, зберігаються в обмеженому буфері. Це запобігає нео-

бмеженому зростанню пам’яті, але повторна спроба є спрощеною і не індексується за точними відсутніми залежностями.

- **Примітивний кеш позиції.** Перетворення видимих індексів у внутрішні позиції використовує лише простий кеш останнього курсора. Це допомагає для близьких послідовних редагувань, але не є повноцінною індексною структурою, тому випадковий доступ у великих документах усе ще може вимагати лінійного обходу.
- **Явні перевірки розміру та формату.** Некоректний UTF-8, пошкоджені оновлення, непідтримувані версії, зайві байти та значення, що перевищують використані цілочислові діапазони, відхиляються. Тому дуже великі документи можуть вимагати поділу на частини або обробки на рівні застосунку.

ВИСНОВОК ДО РОЗДІЛУ 4

У цьому розділі було розглянуто використання, перевірку та експериментальну оцінку розробленого модуля. Коректність перевірялася модульними, сценарними та рандомізованими fuzz-тестами; цей підхід не є формальним доведенням, але практично перевіряє основні властивості текстової CRDT-структури.

Окрему увагу приділено організації тестування в екосистемі Zig. Для зручного запуску тестів, роботи з файлами з різних директорій, підтримки Zig Language Server і налагодження окремих сценаріїв було додано власні кроки в `build.zig`. Це показує, що екосистема Zig ще не має такого рівня зрілості та готових інструментальних практик, як більш поширені екосистеми, тому частину інфраструктури для тестування й налагодження довелося будувати на рівні проєкту.

Демонстраційний редактор підтверджує, що ядро може застосовуватися у вебсередовищі для local-first сценаріїв із кількома репліками. Він також підкреслює відмінність між CRDT-ядром та повноцінним продуктовим редактором і демонструє практичні межі поточної реалізації. Mirage має спрощену модель форматування, відсутність повної підтримки графемних кластерів, і потребу у віртуальному рендерингу для великих документів.

Експериментальний аналіз за допомогою `crdt-benchmarks` показав добрі результати для локальних вставлень, конкурентного додавання тексту та реалістичного набору редагувань. У цих сценаріях реалізація працювала на рівні Yjs, Ywasm та Automerge або швидше за них. Слабкі місця виникають у сценаріях з великою історією видалень, випадковим доступом до позицій, зростанням tombstone-маркерів і великим розміром закодованого документа після тривалого редагування.

Отже, подальший розвиток має бути зосереджений на ефективнішому індексуванні видимих позицій, компактнішому поданні видалень та покращенні завантаження великих станів.

ВИСНОВКИ

У бакалаврському дипломному проєкті було розроблено модуль синхронізації текстових даних у розподілених системах. Метою роботи було створення механізму, який дає змогу виконувати локальне редагування документа, обмінюватися оновленнями між репліками та досягати збіжності їхнього стану без обов'язкового використання центрального координатора.

У процесі роботи було проаналізовано основні підходи до синхронізації спільно редагованих документів: Operational Transformation, тристороннє злиття та конфліктно-вільні реплікові типи даних. На основі порівняння було обґрунтовано вибір CRDT-підходу, оскільки він краще відповідає вимогам локального редагування, асинхронного обміну оновленнями та роботи в умовах тимчасового розходження станів реплік.

У межах проєкту було спроєктовано внутрішню модель текстового документа на основі послідовності елементів зі стабільними ідентифікаторами. Реалізовано локальні операції вставлення, видалення та простого форматування, а також інтеграцію віддалених оновлень, конкурентних вставлень і віддалених видалень. Для синхронізації між репліками використано вектори стану, набори видалень і власний бінарний формат оновлень.

Реалізацію виконано мовою Zig, що дало змогу явно керувати пам'яттю, використовувати компактне внутрішнє представлення документа та компілювати ядро у WebAssembly. Для використання модуля у вебсередовищі було розроблено TypeScript-обгортку та демонстраційний застосунок, який демонструє основні сценарії локального редагування, синхронізації реплік і роботи з простим форматованим текстом.

Коректність основних властивостей реалізації було перевірено модульними, сценарними та рандомізованими fuzz-тестами. Тестування показало, що після обміну оновленнями репліки збігаються до однакового стану, навіть якщо операції виконуються локально та доставляються в різному порядку. Додатково

проведено експериментальний аналіз продуктивності, який показав працездатність обраної моделі та дозволив виявити сильні й слабкі сторони реалізації.

Розроблений модуль є навчально-дослідним прототипом, а не повною заміною промислових CRDT-бібліотек. Водночас він може бути використаний як основа для подальшого створення local-first редакторів або як експериментальне ядро для дослідження синхронізації текстових даних. Основними обмеженнями залишаються неповна підтримка графемних кластерів, накопичення tombstone-даних, спрощена модель форматування та потреба в ефективнішому індексуванні позицій для великих документів. Подальший розвиток роботи може бути спрямований на оптимізацію структури індексів, компактніше подання видалень, покращення формату збереження документа, підтримку складнішого rich-text форматування та інтеграцію з повноцінним текстовим редактором.

Отже, у межах роботи досягнуто поставленої мети та розроблено модуль синхронізації текстових даних на основі CRDT, який підтримує локальне редагування, обмін оновленнями, збіжність реплік і використання у вебсередовищі через WebAssembly.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Nichols D.A. et al. High-latency, low-bandwidth windowing in the Jupiter collaboration system. New York, NY, USA: Association for Computing Machinery, 1995. С. 111–12010 с.
2. Ellis C.A., Gibbs S.J. Concurrency control in groupware systems // SIGMOD Rec. New York, NY, USA: Association for Computing Machinery, 1989. Вип. 18, № 2. С. 399–4079 с.
3. 3.2 Git Branching - Basic Branching and Merging [Online]. URL: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging (дата звернення: 13.04.2026).
4. Daniel Spiewak. Understanding and Applying Operational Transformation [Online]. URL: https://codecommit.com/blog/java/understanding-and-applying-operational-transformation/ (дата звернення: 11.04.2026).
5. Nick Fitzgerald. Operational Transformation: An Introduction [Online]. URL: https://fitzgen.com/2011/03/26/operational-transformation-an-introduction.html (дата звернення: 11.04.2026).
6. Operational transformation [Online]. URL: https://en.wikipedia.org/wiki/Operational_transformation (дата звернення: 12.04.2026).
7. Collaborative Editing in ProseMirror [Online]. URL: https://news.ycombinator.com/item?id=10003918 (дата звернення: 12.04.2026).
8. Oster G. et al. Data consistency for P2P collaborative editing. New York, NY, USA: Association for Computing Machinery, 2006. С. 259–26810 с.
9. John Day-Richter. What's different about the new Google Docs: Making collaboration fast [Online]. 2010. URL: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html (дата звернення: 19.04.2026).
10. Share files from Google Drive [Online]. URL: https://support.google.com/docs/answer/2494822?sjid=11492262890051079160-EU (дата звернення: 12.04.2026).
11. Shapiro M. et al. Conflict-free Replicated Data Types. INRIA, 2011. 18 с.

12. John Mumm - A CRDT Primer: Defanging Order Theory.
13. Lars Hupel. An introduction to Conflict-Free Replicated Data Types; Part 5: Tombstones [Online]. URL: https://lars.hupel.info/topics/crdt/05-tombstones/#2p-sets-are-also-maps (дата звернення: 13.04.2026).
14. Gomes V.B.F. et al. Verifying strong eventual consistency in distributed systems: 109 // Proc. ACM Program. Lang. New York, NY, USA: Association for Computing Machinery, 2017. Вип. 1, № OOPSLA. 28 с.
15. Under the Hood: Redis CRDTs [Online]. URL: https://redis.io/resources/under-the-hood/ (дата звернення: 13.04.2026).
16. How CRDTs make multiplayer text editing part of Zed's DNA [Online]. URL: https://zed.dev/blog/crdts (дата звернення: 18.04.2026).
17. Who is using Yjs [Online]. URL: https://github.com/yjs/yjs/blob/6909511b5bf5c8ab572fc7d12a2268a4646501b0/README.md?plain=1#L126 (дата звернення: 13.04.2026).
18. How Figma's multiplayer technology works [Online]. URL: https://www.figma.com/blog/how-figmas-multiplayer-technology-works/ (дата звернення: 18.04.2026).
19. Yjs [Online]. URL: https://github.com/yjs/yjs (дата звернення: 04.05.2026).
20. Nicolaescu P. et al. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. New York, NY, USA: Association for Computing Machinery, 2016. С. 39–4911 с.
21. Fast Properties in V8 [Online]. URL: https://v8.dev/blog/fast-properties (дата звернення: 04.05.2026).
22. y-crdt [Online]. URL: https://github.com/y-crdt/y-crdt (дата звернення: 04.05.2026).
23. Automerge [Online]. URL: https://github.com/automerge/automerge (дата звернення: 04.05.2026).
24. Automerge 2.2: Rich Text [Online]. URL: https://automerge.org/blog/rich-text/ (дата звернення: 04.05.2026).

25. Modeling Data [Online]. URL: https://automerge.org/docs/cookbook/modeling-data/#performance (дата звернення: 05.05.2026).
26. Block-wise Replicated Growable Array (RGA) Tree Split [Online]. URL: https://github.com/streamich/json-joy/tree/master/packages/json-joy/src/json-crdt/nodes/rga (дата звернення: 04.05.2026).
27. Litt G. et al. Peritext: A CRDT for Collaborative Rich Text Editing // Proceedings of the ACM on Human-Computer Interaction. Association for Computing Machinery, 2022. Вип. 6, № CSCW2. 36 с.
28. Ahmed-Nacer M. et al. Evaluating CRDTs for real-time document editing // Proceedings of the 11th ACM Symposium on Document Engineering. Association for Computing Machinery, 2011. С. 103–112.
29. Zahan N. et al. What are Weak Links in the npm Supply Chain?. Association for Computing Machinery, 2022. С. 331–340.
30. Generating provenance statements [Online]. URL: https://docs.npmjs.com/generating-provenance-statements (дата звернення: 14.05.2026).
31. Postmortem: TanStack npm supply-chain compromise [Online]. URL: https://tanstack.com/blog/npm-supply-chain-compromise-postmortem (дата звернення: 16.05.2026).
32. TypeScript for the New Programmer [Online]. URL: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html (дата звернення: 14.05.2026).
33. Understanding Ownership [Online]. URL: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html (дата звернення: 14.05.2026).
34. WebAssembly [Online]. URL: https://www.rust-lang.org/what/wasm/ (дата звернення: 14.05.2026).
35. Documentation - The Zig Programming Language [Online]. URL: https://ziglang.org/documentation/master/ (дата звернення: 14.05.2026).
36. Tiger Style [Online]. URL: https://github.com/tigerbeetle/tigerbeetle/blob/main/docs/TIGER_STYLE.md (дата звернення: 14.05.2026).

37. Codex web [Online]. URL: https://developers.openai.com/codex/cloud (дата звернення: 14.05.2026).
38. Automerge: a new foundation for collaboration software.
39. UUIDv7: The Time-Sortable Identifier for Modern Databases [Online]. URL: https://uuid7.com/ (дата звернення: 09.05.2026).
40. Distributed Systems 4.1: Logical time.
41. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM. Association for Computing Machinery, 1978. Вип. 21, № 7. С. 558–565.
42. Encoding Spec [Online]. URL: https://capnproto.org/encoding.html (дата звернення: 10.05.2026).
43. CRDTs: The Hard Parts.
44. Binary Format - Values [Online]. URL: https://webassembly.github.io/spec/core/binary/values.html (дата звернення: 06.05.2026).
45. Salomon D. Data Compression: The Complete Reference. Springer, 2004.
46. Delta [Online]. URL: https://quilljs.com/docs/delta/ (дата звернення: 09.05.2026).
47. Understanding WebAssembly text format [Online]. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Understanding_the_text_format#webassembly_memory (дата звернення: 10.05.2026).
48. Mitchell Hashimoto. Grapheme Clusters and Terminal Emulators [Online]. 2023. URL: https://mitchellh.com/writing/grapheme-clusters-in-terminals (дата звернення: 10.05.2026).
49. Zig Test [Online]. URL: https://ziglang.org/documentation/master/#Zig-Test (дата звернення: 12.05.2026).
50. crdt-benchmarks [Online]. URL: https://github.com/dmonad/crdt-benchmarks/tree/main (дата звернення: 16.05.2026).

ДОДАТОК 1

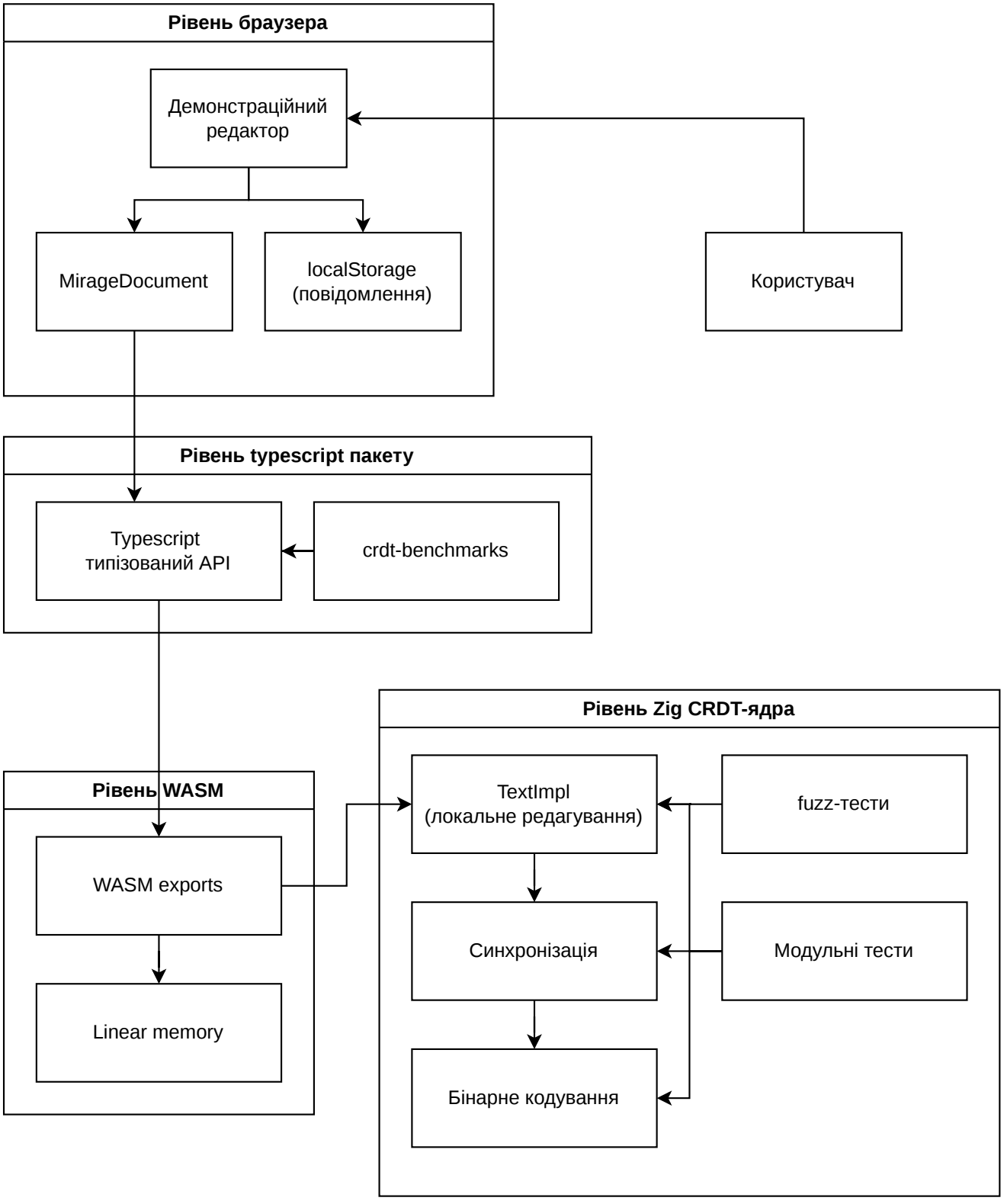
Модуль синхронізації текстових даних у розподілених системах

Компоненти застосунка
(структурна схема)

ІАЛЦ.467100.004 Д1

Аркушів 1

Київ – 2026 р.



					ІАЛЦ.467100.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Сірик М. О.			**Модуль синхронізації текстових даних у розподілених системах**	Літера: — / Аркуш: 1 / Аркушів: 1
Перевірив		Долголенко О. М.				
Н. Контр.		Коренко Д. В.			Компоненти застосунка (структурна схема)	НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Затвердив		Волокита А. М.				

ДОДАТОК 2

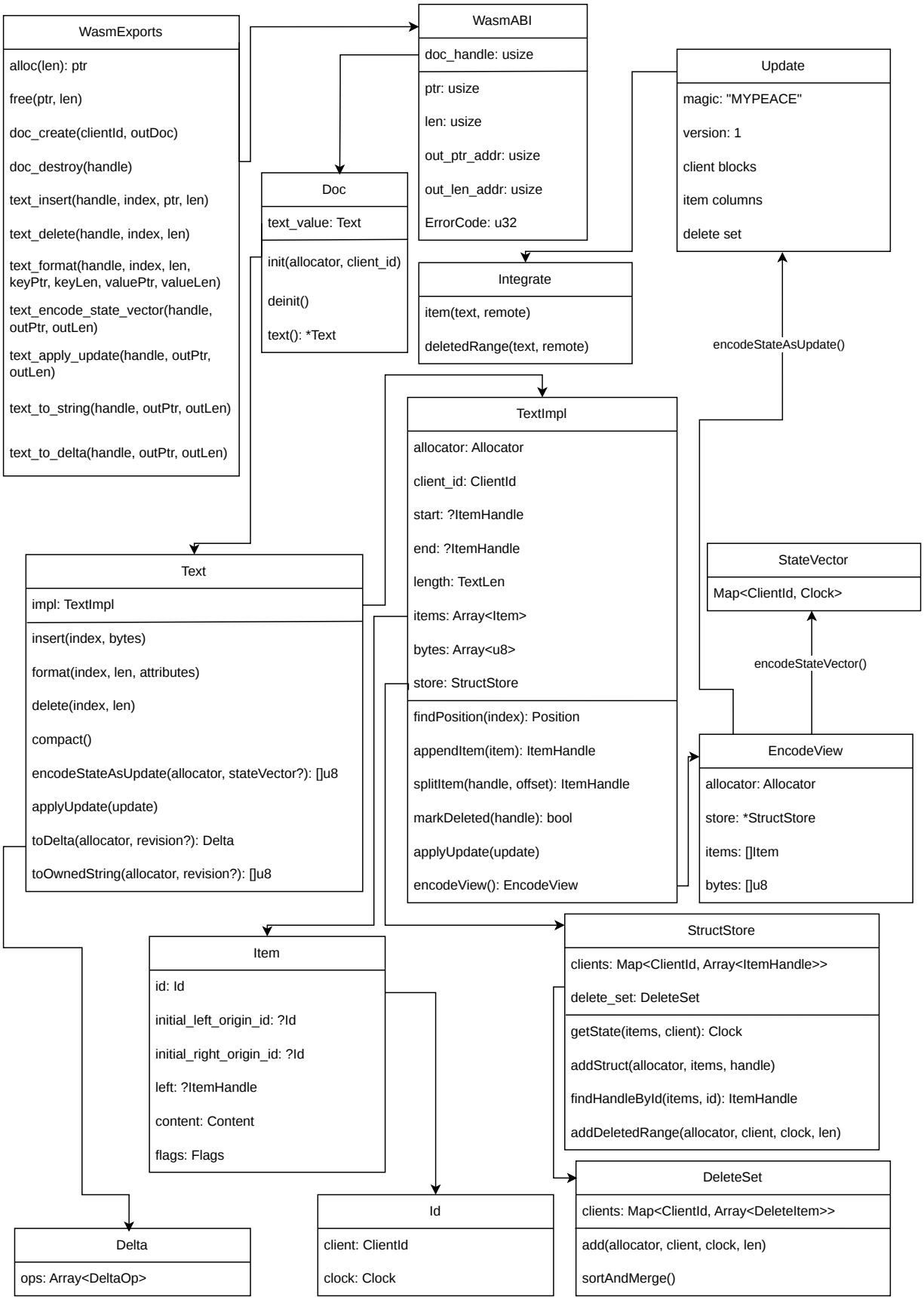
Модуль синхронізації текстових даних у розподілених системах

Діаграма класів
(функціональна схема)

ІАЛЦ.467100.005 Д2

Аркушів 1

Київ – 2026 р.



					ІАЛЦ.467100.005 Д2
		№ докум.	Підпис	Дата	
Розробив		Сірик М. О.			**Модуль синхронізації текстових даних у розподілених системах**
Перевірив		Долголенко О. М.			
Н. Контр.		Коренко Д. В.			Діаграма класів (функціональна схема)
Затвердив		Волокита А. М.			

Літера	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21

ДОДАТОК З

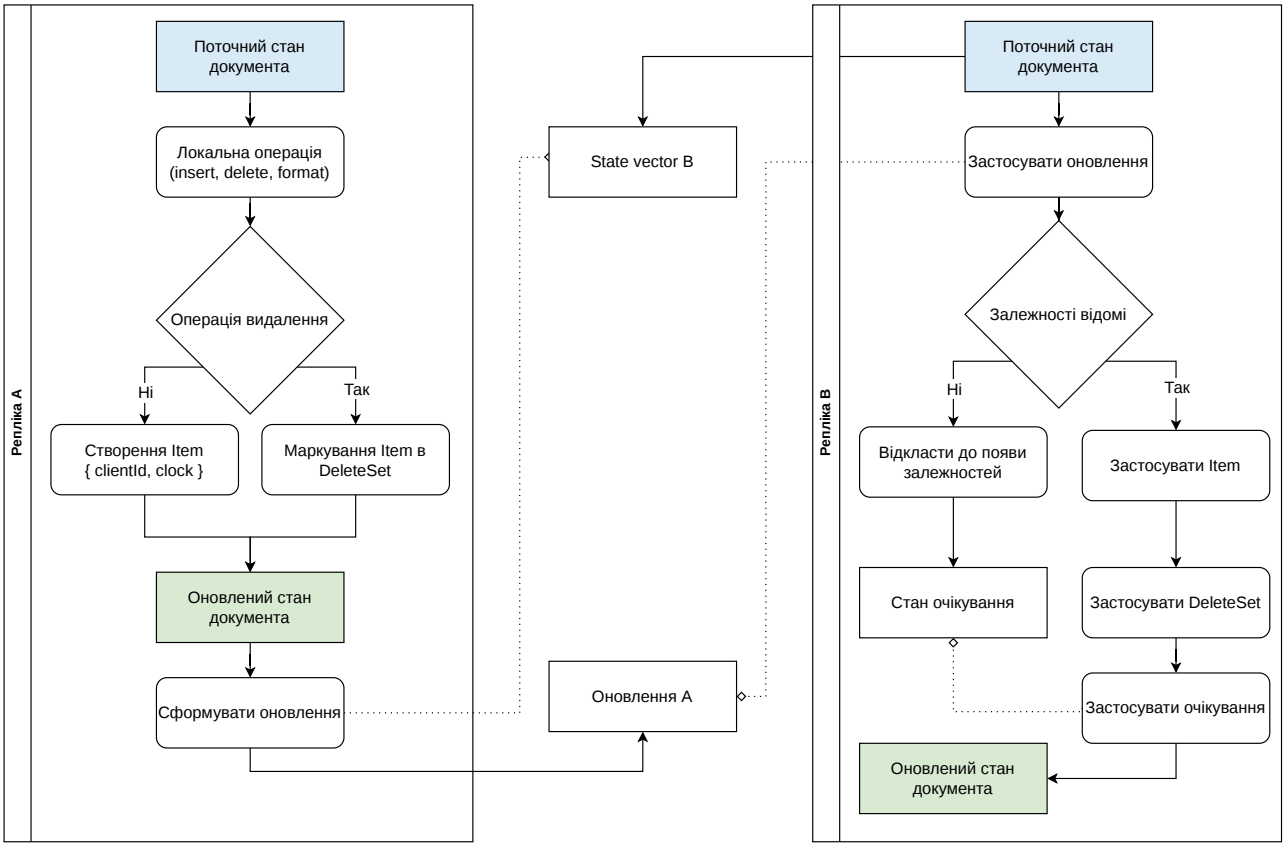
Модуль синхронізації текстових даних у розподілених системах

Алгоритм роботи застосунку
(принципова схема)

ІАЛЦ.467100.006 Д3

Аркушів 1

Київ – 2026 р.



					ІАЛЦ.467100.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Сірик М. О.			**Модуль синхронізації текстових даних у розподілених системах**	Літера	Аркуш 1 / Аркушів 1
Перевірив		Долголенко О. М.					
Н. Контр.		Коренко Д. В.			Алгоритм роботи застосунку (принципова схема)	НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Затвердив		Волокита А. М.					

ДОДАТОК 4

Модуль синхронізації текстових даних у розподілених системах

Текст програмного коду

ІАЛЦ.467100.007 Д4

Аркушів 26

Київ – 2026 р.

```zig
const std = @import("std");
const id = @import("../id.zig");
const item_mod = @import("../item.zig");
const store_mod = @import("../store.zig");
const attrs = @import("../attrs.zig");
const utf = @import("../utf.zig");
const formatting = @import("../formatting.zig");
const delta_mod = @import("../delta.zig");
const sync = @import("../sync.zig");
const integrate = @import("../integrate.zig");

pub const TextError = error{
    IndexOutOfBounds,
    InvalidHandle,
    InvalidUtf8,
    ScalarIndexOutOfBounds,
    TextTooLarge,
    ItemTooLarge,
    UnsupportedContent,
    MissingDependency,
    InvalidUpdate,
    UnsupportedUpdateVersion,
    VarIntOverflow,
    TrailingBytes,
    PendingUpdatesTooLarge,
} || std.mem.Allocator.Error || store_mod.StoreError;

const Position = struct {
    index: id.TextIndex,
    left: ?item_mod.ItemHandle,
    right: ?item_mod.ItemHandle,
};

const max_pending_update_bytes: usize = 16 * 1024 * 1024;

pub const TextImpl = struct {
    allocator: std.mem.Allocator,
    client_id: id.ClientId,

    /// The first item in the linked list
    /// If it's `null`, the list is empty
    start: ?item_mod.ItemHandle = null,

    /// The last item in the linked list
```

```
    /// If it's `null`, the list is empty
    end: ?item_mod.ItemHandle = null,

    /// The visible text length (non-deleted, countable items)
    length: id.TextLen = 0,

    /// All items in the document.
    /// The elements are not necessarily in the document order.
    /// The document order is defined by the `left`/`right` fields of the items.
    ///
    /// `ItemHandle` is just the index in this array
    items: std.ArrayList(item_mod.Item) = .empty,

    /// A shared byte buffer holding the actual string data
    /// (both text and formatting attributes)
    ///
    /// `TextItem` doesn't own separate strings,
    /// but just slices of this buffer
    bytes: std.ArrayList(u8) = .empty,

    /// TODO:
    /// Client clock order, contrary to the `Item.left`/`right` which defines
    /// the document order.
    ///
    /// It answers the questions:
    /// - what clock should the next local item use
    /// - do we already have item {client_id, clock}
    /// - which item contains this clock
    /// - is there a clock gap in the client's history
    store: store_mod.StructStore = .{},

    /// Updates that arrived before their dependencies.
    /// They can't be applied yet
    /// TODO: store the missing dependency per pending update and retry only
    /// updates waiting on clients whose state advanced.
    pending_updates: std.ArrayList([]u8) = .empty,
    pending_update_bytes: usize = 0,
    position_cursor: ?Position = null,
    /// Latest completed document-local history number
    /// Each mutation operation bumps it
    current_revision: id.Revision = 0,
    /// Temporary field, to let multiple internal items share the same revision
    /// E.g formatting items
    active_revision: ?id.Revision = null,
```

```
pub fn init(allocator: std.mem.Allocator, client_id: id.ClientId) TextImpl {
    return .{
        .allocator = allocator,
        .client_id = client_id,
    };
}

pub fn deinit(self: *TextImpl) void {
    for (self.pending_updates.items) |pending| {
        self.allocator.free(pending);
    }
    self.pending_updates.deinit(self.allocator);
    self.store.deinit(self.allocator);
    self.bytes.deinit(self.allocator);
    self.items.deinit(self.allocator);
    self.* = undefined;
}

pub fn len(self: *const TextImpl) id.TextLen {
    return self.length;
}

pub fn currentRevision(self: *const TextImpl) id.Revision {
    return self.current_revision;
}

pub fn historyLen(self: *const TextImpl) id.Revision {
    return self.current_revision;
}

pub fn internalByteLen(self: *const TextImpl) usize {
    var total: usize = 0;
    total += self.items.items.len * @sizeOf(item_mod.Item);
    total += self.bytes.items.len;
    total += self.store.byteLen();
    total += self.pending_updates.items.len * @sizeOf([]u8);
    total += self.pending_update_bytes;
    return total;
}

pub fn visibleByteLen(self: *const TextImpl, revision: ?id.Revision) usize {
    var total: usize = 0;
    var cursor = self.start;
```

```
        while (cursor) |handle| {
            const current = self.items.items[handle];
            cursor = current.right;
            if (!self.isItemVisibleAt(handle, revision)) continue;
            switch (current.content) {
                .string => |slice| total += self.sliceBytes(slice).len,
                .format => {},
            }
        }
        return total;
    }

    /// Inserts into the visible character index
     pub fn insert(self: *TextImpl, index: id.TextIndex, bytes: []const u8) TextError!
void {
        if (index > self.length) return error.IndexOutOfBounds;
        const logical_len = try utf.countUnicodeLen(bytes);
        if (logical_len == 0) return;

        defer self.finishRevision();
        const pos = try self.findPosition(index);
        _ = try self.insertStringAt(pos, bytes);
    }

    /// Inserts string with formatting
    ///
    /// E.g inserting `C` with `bold: true` at index 1
    ///   0  1
    /// [A] [B]
    /// [A] [bold:true] [C] [bold:null] [B]
    pub fn insertWithAttrs(
        self: *TextImpl,
        index: id.TextIndex,
        bytes: []const u8,
        attributes: []const attrs.Attribute,
    ) TextError!void {
        if (index > self.length) return error.IndexOutOfBounds;
        const logical_len = try utf.countUnicodeLen(bytes);
        if (logical_len == 0) return;

        defer self.finishRevision();
        var pos = try self.findPosition(index);
        for (attributes) |attribute| {
            const marker = try self.insertFormatAt(pos, attribute);
```

```
                        pos = .{ .index = pos.index, .left = marker, .right =
self.items.items[marker].right };
        }
        const string_handle = try self.insertStringAt(pos, bytes);
            pos = .{ .index = pos.index + logical_len, .left = string_handle, .right
= self.items.items[string_handle].right };
        for (attributes) |attribute| {
            const marker = try self.insertFormatAt(pos, .{
                .key = attribute.key,
                .value = .null,
            });
                        pos = .{ .index = pos.index, .left = marker, .right =
self.items.items[marker].right };
        }
    }

    pub fn format(
        self: *TextImpl,
        index: id.TextIndex,
        format_len: id.TextLen,
        attributes: []const attrs.Attribute,
    ) TextError!void {
        if (index > self.length) return error.IndexOutOfBounds;
        if (format_len > self.length - index) return error.IndexOutOfBounds;
        if (format_len == 0 or attributes.len == 0) return;

        defer self.finishRevision();
        const restore_attributes = try formatting.findRestoreAttr(
            self,
            self.allocator,
            index + format_len,
            attributes,
        );
        defer formatting.freeOwnedAttributes(self.allocator, restore_attributes);

        var start_pos = try self.findPosition(index);
        for (attributes) |attribute| {
            const marker = try self.insertFormatAt(start_pos, attribute);
                  start_pos = .{ .index = start_pos.index, .left = marker, .right =
self.items.items[marker].right };
        }
        var end_pos = try self.findPosition(index + format_len);
        for (restore_attributes) |owned_attribute| {
            const marker = try self.insertFormatAt(end_pos, owned_attribute.attribute);
```

```
                    end_pos = .{ .index = end_pos.index, .left = marker, .right =
self.items.items[marker].right };
        }
    }

    /// Inserts a string into a `gap` defined by the position.
    /// .{ .left = 0, .right = 1 } means that the string
    /// will be inserted between items with handles 0 and 1.
      fn insertStringAt(self: *TextImpl, pos: Position, bytes: []const u8) TextError!
item_mod.ItemHandle {
        const logical_len = try utf.countUnicodeLen(bytes);
        if (logical_len == 0) return error.IndexOutOfBounds;

        const bytes_len = try intCast(u32, bytes.len);
        const bytes_start = try self.appendBytes(bytes);
        const handle = try self.appendItem(.{
            .id = .{
                .client = self.client_id,
                .clock = self.store.getState(self.items.items, self.client_id),
            },
         .initial_left_origin_id = if (pos.left) |left| self.items.items[left].getLastId()
else null,
            .initial_right_origin_id = if (pos.right) |right| self.items.items[right].id
else null,
            .left = pos.left,
            .right = pos.right,
            .content = .{ .string = .{
                .bytes_start = bytes_start,
                .bytes_len = bytes_len,
                .logical_len = logical_len,
            } },
            .flags = .{
                .countable = true,
                .deleted = false,
            },
            .inserted_revision = self.mutationRevision(),
        });

        try self.store.addStruct(self.allocator, self.items.items, handle);
        self.linkInserted(handle, pos.left, pos.right);
        self.length += logical_len;
        self.position_cursor = .{
            .index = pos.index + logical_len,
            .left = handle,
```

```
            .right = self.items.items[handle].right,
        };
        return handle;
    }

    fn insertFormatAt(self: *TextImpl, pos: Position, attribute: attrs.Attribute)
TextError!item_mod.ItemHandle {
        const content = try self.appendAttribute(attribute);
        const handle = try self.appendItem(.{
            .id = .{
                .client = self.client_id,
                .clock = self.store.getState(self.items.items, self.client_id),
            },
            .initial_left_origin_id = if (pos.left) |left| self.items.items[left].getLastId()
else null,
            .initial_right_origin_id = if (pos.right) |right| self.items.items[right].id
else null,
            .left = pos.left,
            .right = pos.right,
            .content = .{ .format = content },
            .flags = .{
                .countable = false,
                .deleted = false,
            },
            .inserted_revision = self.mutationRevision(),
        });

        try self.store.addStruct(self.allocator, self.items.items, handle);
        self.linkInserted(handle, pos.left, pos.right);
        self.position_cursor = .{
            .index = pos.index,
            .left = handle,
            .right = self.items.items[handle].right,
        };
        return handle;
    }

    pub fn delete(self: *TextImpl, index: id.TextIndex, delete_len: id.TextLen) TextError!
void {
        if (index > self.length) return error.IndexOutOfBounds;
        if (delete_len > self.length - index) return error.IndexOutOfBounds;
        if (delete_len == 0) return;

        defer self.finishRevision();
```

```
        var pos = try self.findPosition(index);
        var remaining = delete_len;
        while (remaining > 0) {
            const handle = pos.right orelse return error.IndexOutOfBounds;
            const current = self.items.items[handle];
            const is_visible = !current.flags.deleted and current.flags.countable;
            if (!is_visible) {
                pos = .{ .index = index, .left = handle, .right = current.right };
                continue;
            }
            const current_len = current.getClockLen();
            const should_split = current_len > remaining;
            if (should_split) {
                _ = try self.splitItem(handle, remaining);
            }

            const delete_handle = handle;
            const deleted_len = self.items.items[delete_handle].getClockLen();
            _ = try self.markDeleted(delete_handle);
            remaining -= deleted_len;
            pos = .{
                .index = index,
                .left = delete_handle,
                .right = self.items.items[delete_handle].right,
            };
        }
        self.position_cursor = pos;
    }

    pub fn compact(self: *TextImpl) TextError!void {
        defer self.finishRevision();
        try self.cleanupFormatting();
        try self.mergeAdjacentStringItems();
        self.invalidatePositionCursor();
    }

       pub  fn  encodeStateVector(self:  *const  TextImpl,  allocator:  std.mem.Allocator)
TextError![]u8 {
        return try sync.encodeStateVector(self.encodeView(), allocator);
    }

    pub fn encodeStateAsUpdate(
        self: *const TextImpl,
        allocator: std.mem.Allocator,
```

```
        encoded_target_state_vector: ?[]const u8,
    ) TextError![]u8 {
                    return   try   sync.encodeStateAsUpdate(self.encodeView(),   allocator,
encoded_target_state_vector);
    }

    pub fn applyUpdate(self: *TextImpl, update: []const u8) TextError!void {
        self.applyUpdateOnce(update) catch |err| switch (err) {
            error.MissingDependency => {
                if (update.len > max_pending_update_bytes - self.pending_update_bytes) {
                    return error.PendingUpdatesTooLarge;
                }
                const copy = try self.allocator.dupe(u8, update);
                errdefer self.allocator.free(copy);
                try self.pending_updates.append(self.allocator, copy);
                self.pending_update_bytes += copy.len;
                return;
            },
            else => return err,
        };
        try self.retryPendingUpdates();
    }

    pub fn toOwnedString(
        self: *const TextImpl,
        allocator: std.mem.Allocator,
        revision: ?id.Revision,
    ) TextError![]u8 {
        var out: std.ArrayList(u8) = .empty;
        errdefer out.deinit(allocator);

        var cursor = self.start;
        while (cursor) |handle| {
            const current = self.items.items[handle];
            defer cursor = current.right;
            if (!self.isItemVisibleAt(handle, revision))
                continue;

            switch (current.content) {
               .string => |slice| try out.appendSlice(allocator, self.sliceBytes(slice)),
                .format => {},
            }
        }
        return try out.toOwnedSlice(allocator);
```

```
}

pub fn toDelta(
    self: *const TextImpl,
    allocator: std.mem.Allocator,
    revision: ?id.Revision,
) TextError!attrs.Delta {
    return try delta_mod.toDelta(self, allocator, revision);
}

pub fn toDeltaRange(
    self: *const TextImpl,
    allocator: std.mem.Allocator,
    start: id.TextIndex,
    end: id.TextIndex,
    revision: ?id.Revision,
    include_leading_attrs: bool,
) TextError!attrs.Delta {
    return try delta_mod.toDeltaRange(self, allocator, start, end, .{
        .revision = revision,
        .include_leading_attrs = include_leading_attrs,
    });
}

fn encodeView(self: *const TextImpl) sync.EncodeView {
    return .{
        .allocator = self.allocator,
        .store = &self.store,
        .items = self.items.items,
        .bytes = self.bytes.items,
    };
}

/// Converts a visible character index into a position.
/// Splits the item if the index is in the middle of it.
///
/// Position returns a `gap`
///  0   1   2
/// [A] [B] [C] -index: 1> { .left = 0, .right = 1 }
///
/// Note if the character has formatting
///              0   1  2
/// [{bold: true}] [A] [{bold: true}] -index: 1> { .left = 2, .right = null }
fn findPosition(self: *TextImpl, index: id.TextIndex) TextError!Position {
```

```
        if (index > self.length) return error.IndexOutOfBounds;

        const Direction = enum {
            forward_from_start,
            backward_from_end,
            forward_from_cursor,
            backward_from_cursor,
        };

        var direction: Direction = .forward_from_start;
        var best_distance = index;

        const distance_from_end = self.length - index;
        if (distance_from_end < best_distance) {
            direction = .backward_from_end;
            best_distance = distance_from_end;
        }

        if (self.position_cursor) |cached| {
            if (index >= cached.index) {
                const distance = index - cached.index;
                if (distance <= best_distance) {
                    direction = .forward_from_cursor;
                    best_distance = distance;
                }
            } else {
                const distance = cached.index - index;
                if (distance <= best_distance) {
                    direction = .backward_from_cursor;
                    best_distance = distance;
                }
            }
        }

        return switch (direction) {
            .forward_from_start => try self.findPositionForward(index, null, self.start,
best_distance),
             .backward_from_end => try self.findPositionBackward(index, self.end, null,
best_distance),
            .forward_from_cursor => blk: {
                const cached = self.position_cursor.?;
               break :blk try self.findPositionForward(index, cached.left, cached.right,
best_distance);
            },
```

```
            .backward_from_cursor => blk: {
                const cached = self.position_cursor.?;
               break :blk try self.findPositionBackward(index, cached.left, cached.right,
best_distance);
            },
        };
    }

    fn findPositionForward(
        self: *TextImpl,
        index: id.TextIndex,
        initial_left: ?item_mod.ItemHandle,
        initial_cursor: ?item_mod.ItemHandle,
        initial_remaining: id.TextLen,
    ) TextError!Position {
        var remaining = initial_remaining;
        var left = initial_left;
        var cursor = initial_cursor;
        while (cursor) |handle| {
            const current = self.items.items[handle];
            defer {
                  // Defer is ok, since return value is evaluated before; and they are
scalar fields
                left = handle;
                cursor = current.right;
            }

            const is_visible = !current.flags.deleted and current.flags.countable;
            if (!is_visible) {
                continue;
            }

            const is_found = remaining == 0;
            if (is_found) {
                const pos = Position{ .index = index, .left = left, .right = handle };
                self.position_cursor = pos;
                return pos;
            }

            const current_len: id.TextLen = current.getClockLen();
            const is_within_current = remaining < current_len;
            if (is_within_current) {
                const right = try self.splitItem(handle, remaining);
                const pos = Position{ .index = index, .left = handle, .right = right };
```

```
                self.position_cursor = pos;
                return pos;
            }

            remaining -= current_len;
        }

        if (remaining != 0) return error.IndexOutOfBounds;
        const pos = Position{ .index = index, .left = left, .right = null };
        self.position_cursor = pos;
        return pos;
    }

    fn findPositionBackward(
        self: *TextImpl,
        index: id.TextIndex,
        initial_cursor: ?item_mod.ItemHandle,
        initial_right: ?item_mod.ItemHandle,
        initial_remaining: id.TextLen,
    ) TextError!Position {
        var remaining = initial_remaining;
        var right = initial_right;
        var cursor = initial_cursor;
        if (remaining == 0) {
            const pos = Position{ .index = index, .left = cursor, .right = right };
            self.position_cursor = pos;
            return pos;
        }

        while (cursor) |handle| {
            const current = self.items.items[handle];
            const is_visible = !current.flags.deleted and current.flags.countable;
            if (!is_visible) {
                right = handle;
                cursor = current.left;
                continue;
            }

            const current_len: id.TextLen = current.getClockLen();
            if (remaining < current_len) {
               const right_handle = try self.splitItem(handle, current_len - remaining);
             const pos = Position{ .index = index, .left = handle, .right = right_handle };
                self.position_cursor = pos;
                return pos;
```

```
            }
            if (remaining == current_len) {
                    const pos = Position{ .index = index, .left = current.left, .right
= handle };
                self.position_cursor = pos;
                return pos;
            }

            remaining -= current_len;
            right = handle;
            cursor = current.left;
        }

        if (remaining != 0) return error.IndexOutOfBounds;
        const pos = Position{ .index = index, .left = null, .right = right };
        self.position_cursor = pos;
        return pos;
    }

    pub fn invalidatePositionCursor(self: *TextImpl) void {
        self.position_cursor = null;
    }

    pub fn mutationRevision(self: *TextImpl) id.Revision {
        if (self.active_revision) |revision| return revision;
        self.current_revision += 1;
        self.active_revision = self.current_revision;
        return self.current_revision;
    }

    fn finishRevision(self: *TextImpl) void {
        self.active_revision = null;
    }

    pub fn isItemAliveAt(self: *const TextImpl, handle: item_mod.ItemHandle, revision: ?
id.Revision) bool {
        const current = self.items.items[handle];
        const target_revision = revision orelse return !current.flags.deleted;
        return current.inserted_revision <= target_revision and
                    (current.deleted_revision == null or current.deleted_revision.? >
target_revision);
    }

   pub fn isItemVisibleAt(self: *const TextImpl, handle: item_mod.ItemHandle, revision: ?
```

```
id.Revision) bool {
        const current = self.items.items[handle];
        return current.flags.countable and self.isItemAliveAt(handle, revision);
    }

    pub fn markDeleted(self: *TextImpl, handle: item_mod.ItemHandle) TextError!bool {
        if (self.items.items[handle].flags.deleted) return false;

        const current = self.items.items[handle];
        self.items.items[handle].flags.deleted = true;
        self.items.items[handle].deleted_revision = self.mutationRevision();
        try self.store.addDeletedRange(
            self.allocator,
            current.id.client,
            current.id.clock,
            current.getClockLen(),
        );

        if (current.flags.countable) {
            self.length -= current.getClockLen();
            self.invalidatePositionCursor();
        }
        return true;
    }

    /// Splits the item at the given offset (in visible characters).
    /// FormatItems can't be split, since they don't have a visible length.
    ///
    /// Offset is the length of the left part after the split.
    /// [a, b, c] - offset: 1> [a] [b, c]
    ///
    /// If the offset is out of bounds returns the error.
    /// Returns the handle of the right part after the split.
      fn splitItem(self: *TextImpl, handle: item_mod.ItemHandle, offset: id.TextLen)
TextError!item_mod.ItemHandle {
        const left_len = self.items.items[handle].getClockLen();
        const is_offset_out_of_bounds = offset == 0 or offset >= left_len;
        if (is_offset_out_of_bounds) return error.IndexOutOfBounds;

        const left_snapshot = self.items.items[handle];
        const slice = switch (left_snapshot.content) {
            .string => |string_slice| string_slice,
            .format => return error.UnsupportedContent,
        };
```

```
        const full_bytes = self.sliceBytes(slice);
       const byte_offset = try intCast(u32, try utf.getByteOffsetForCharIndex(full_bytes,
offset));
        const right_bytes_len = slice.bytes_len - byte_offset;
        const right_len = left_snapshot.getClockLen() - offset;

        self.items.items[handle].content = .{ .string = .{
            .bytes_start = slice.bytes_start,
            .bytes_len = byte_offset,
            .logical_len = offset,
        } };

        const right_handle = try self.appendItem(.{
            .id = .{
                .client = left_snapshot.id.client,
                .clock = left_snapshot.id.clock + offset,
            },
            .initial_left_origin_id = self.items.items[handle].getLastId(),
            .initial_right_origin_id = left_snapshot.initial_right_origin_id,
            .left = handle,
            .right = left_snapshot.right,
            .content = .{ .string = .{
                .bytes_start = slice.bytes_start + byte_offset,
                .bytes_len = right_bytes_len,
                .logical_len = right_len,
            } },
            .flags = left_snapshot.flags,
            .inserted_revision = left_snapshot.inserted_revision,
            .deleted_revision = left_snapshot.deleted_revision,
        });

        self.items.items[handle].right = right_handle;
        if (left_snapshot.right) |old_right| {
            self.items.items[old_right].left = right_handle;
        } else {
            self.end = right_handle;
        }
             try  self.store.insertStructAfter(self.allocator,  self.items.items,  handle,
right_handle);
        self.invalidatePositionCursor();
        return right_handle;
    }

    // Ensures there is item gap before `target`.
```

```
    /// Returns the right handle of the gap.
    ///  01234
    /// `Hello` - target: 2> `He` [gap] `llo`
    pub fn getItemCleanStart(self: *TextImpl, target: id.Id) TextError!item_mod.ItemHandle
{
        const handle = try self.store.findHandleById(self.items.items, target);
        const current = self.items.items[handle];
        if (current.id.clock < target.clock) {
            return try self.splitItem(handle, target.clock - current.id.clock);
        }
        return handle;
    }

    /// Ensures there is item gap after `target`
    /// Returns the left handle of the gap
    ///  01234
    /// `Hello` - target: 2> `Hel` [gap] `lo`
    pub fn getItemCleanEnd(self: *TextImpl, target: id.Id) TextError!item_mod.ItemHandle {
        const handle = try self.store.findHandleById(self.items.items, target);
        const current = self.items.items[handle];
        const offset = target.clock - current.id.clock + 1;
        if (offset < current.getClockLen()) {
            _ = try self.splitItem(handle, offset);
        }
        return handle;
    }

    fn cleanupFormatting(self: *TextImpl) TextError!void {
        try formatting.cleanup(self);
    }

    fn mergeAdjacentStringItems(self: *TextImpl) TextError!void {
        var cursor = self.start;
        while (cursor) |left_handle| {
            const right_handle = self.items.items[left_handle].right orelse break;
            if (canMergeStringItems(self, left_handle, right_handle)) {
                try self.mergeStringItems(left_handle, right_handle);
                continue;
            }
            cursor = right_handle;
        }
    }

      fn canMergeStringItems(self: *const TextImpl, left_handle: item_mod.ItemHandle,
```

```
right_handle: item_mod.ItemHandle) bool {
        const left = self.items.items[left_handle];
        const right = self.items.items[right_handle];
        if (left.id.client != right.id.client) return false;
        if (left.id.clock + left.getClockLen() != right.id.clock) return false;
            if (left.flags.countable != right.flags.countable or left.flags.deleted !=
right.flags.deleted) return false;
      if (!id.checkIfIdEql(right.initial_left_origin_id, left.getLastId())) return false;
       if (!id.checkIfIdEql(left.initial_right_origin_id, right.initial_right_origin_id))
return false;

        const left_slice = switch (left.content) {
            .string => |slice| slice,
            .format => return false,
        };
        const right_slice = switch (right.content) {
            .string => |slice| slice,
            .format => return false,
        };
        return left_slice.bytes_start + left_slice.bytes_len == right_slice.bytes_start;
    }

    // `canMergeStringItems` should be checked before calling this function
    fn mergeStringItems(self: *TextImpl, left_handle: item_mod.ItemHandle, right_handle:
item_mod.ItemHandle) TextError!void {
        const right = self.items.items[right_handle];
        const right_slice = right.content.string;

        self.items.items[left_handle].content.string.bytes_len += right_slice.bytes_len;
                            self.items.items[left_handle].content.string.logical_len    +=
right_slice.logical_len;
                                self.items.items[left_handle].initial_right_origin_id    =
right.initial_right_origin_id;
        self.items.items[left_handle].right = right.right;

        if (right.right) |next_handle| {
            self.items.items[next_handle].left = left_handle;
        } else {
            self.end = left_handle;
        }

        // TODO: this marks item as a dead slot
        self.items.items[right_handle].flags.deleted = true;
        self.items.items[right_handle].deleted_revision = self.mutationRevision();
```

```
        self.items.items[right_handle].left = null;
        self.items.items[right_handle].right = null;
        try self.store.removeStruct(self.items.items, right_handle);
    }

    fn retryPendingUpdates(self: *TextImpl) TextError!void {
        var index: usize = 0;
        while (index < self.pending_updates.items.len) {
            const pending = self.pending_updates.items[index];
            self.applyUpdateOnce(pending) catch |err| switch (err) {
                error.MissingDependency => {
                    index += 1;
                    continue;
                },
                else => return err,
            };
            const removed = self.pending_updates.orderedRemove(index);
            self.pending_update_bytes -= removed.len;
            self.allocator.free(removed);
            index = 0;
        }
    }

    fn applyUpdateOnce(self: *TextImpl, update: []const u8) TextError!void {
        try sync.validateUpdateBytes(update);
        defer self.finishRevision();

        const callbacks = struct {
            const Context = struct {
                text: *TextImpl,
            };

              fn item(context: Context, decoded: sync.DecodedItem) sync.ReadUpdateError!
void {
                const content: integrate.RemoteContent = switch (decoded.content) {
                    .string => |string| .{ .string = string },
                    .format => |format_content| .{ .format = .{
                        .key = format_content.key,
                        .value = format_content.value,
                    } },
                };
                try integrate.item(context.text, .{
                    .id = decoded.id,
                    .len = decoded.len,
```

```
                    .initial_left_origin_id = decoded.initial_left_origin_id,
                    .initial_right_origin_id = decoded.initial_right_origin_id,
                    .content = content,
                });
            }

                fn deletedRange(context: Context, decoded: sync.DecodedDeleteRange)
sync.ReadUpdateError!void {
                try integrate.deletedRange(context.text, .{
                    .client = decoded.client,
                    .clock = decoded.clock,
                    .len = decoded.len,
                });
            }
        };

        try sync.readUpdate(callbacks.Context, update, .{
            .context = .{ .text = self },
            .item = callbacks.item,
            .deletedRange = callbacks.deletedRange,
        });
    }

    pub fn linkInserted(
        self: *TextImpl,
        handle: item_mod.ItemHandle,
        left: ?item_mod.ItemHandle,
        right: ?item_mod.ItemHandle,
    ) void {
        if (left) |left_handle| {
            self.items.items[left_handle].right = handle;
        } else {
            self.start = handle;
        }
        if (right) |right_handle| {
            self.items.items[right_handle].left = handle;
        } else {
            self.end = handle;
        }
        self.items.items[handle].left = left;
        self.items.items[handle].right = right;
        self.invalidatePositionCursor();
    }
```

```
    /// Returns the `new_item` handle
          pub   fn   appendItem(self:   *TextImpl,   new_item:   item_mod.Item)   TextError!
item_mod.ItemHandle {
        const handle = try intCast(item_mod.ItemHandle, self.items.items.len);
        try self.items.append(self.allocator, new_item);
        return handle;
    }

    /// Returns the start index of the appended bytes in the shared byte buffer.
    pub fn appendBytes(self: *TextImpl, source: []const u8) TextError!u32 {
        const start = try intCast(u32, self.bytes.items.len);
        try self.bytes.appendSlice(self.allocator, source);
        return start;
    }

      pub  fn  appendAttribute(self:  *TextImpl,  attribute:  attrs.Attribute)  TextError!
item_mod.AttributeSlice {
        const key_start = try self.appendBytes(attribute.key);
        const key_len = try intCast(u32, attribute.key.len);
        return switch (attribute.value) {
            .null => .{
                .key_start = key_start,
                .key_len = key_len,
                .value_is_null = true,
            },
            .string => |value| .{
                .key_start = key_start,
                .key_len = key_len,
                .value_start = try self.appendBytes(value),
                .value_len = try intCast(u32, value.len),
                .value_is_null = false,
            },
        };
    }

    pub fn sliceBytes(self: *const TextImpl, slice: item_mod.TextSlice) []const u8 {
        const start: usize = slice.bytes_start;
        return self.bytes.items[start..][0..slice.bytes_len];
    }

   pub fn attributeKeyBytes(self: *const TextImpl, slice: item_mod.AttributeSlice) []const
u8 {
        const start: usize = slice.key_start;
        return self.bytes.items[start..][0..slice.key_len];
```

```
    }

     pub fn attributeValueBytes(self: *const TextImpl, slice: item_mod.AttributeSlice)
[]const u8 {
        const start: usize = slice.value_start;
        return self.bytes.items[start..][0..slice.value_len];
    }
};

fn intCast(comptime T: type, value: anytype) error{TextTooLarge}!T {
    return std.math.cast(T, value) orelse error.TextTooLarge;
}

//============================================================================
// findPosition
//============================================================================
test "findPosition returns gap before item at exact boundary" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "A");
    try text.insert(1, "B");
    try text.insert(2, "C");

    const position = try text.findPosition(1);

    try std.testing.expectEqual(0, position.left);
    try std.testing.expectEqual(1, position.right);
}

test "findPosition splits item when index is inside visible text item" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "ABC");

    const position = try text.findPosition(1);

    try std.testing.expectEqual(0, position.left);
    try std.testing.expectEqual(1, position.right);

    try std.testing.expectEqual(1, text.items.items[0].getClockLen());
```

```
    try std.testing.expectEqual(2, text.items.items[1].getClockLen());
}

test "findPosition returns end gap at document length" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "A");
    try text.insert(1, "B");

    const position = try text.findPosition(text.len());

    try std.testing.expectEqual(1, position.left);
    try std.testing.expectEqual(null, position.right);
}

test "findPosition tracks invisible items in linked-list gap" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    const bold = attrs.Attribute{
        .key = "bold",
        .value = .{ .string = "true" },
    };

    try text.insertWithAttrs(0, "A", &.{bold});

    const position = try text.findPosition(1);

    try std.testing.expectEqual(2, position.left);
    try std.testing.expectEqual(null, position.right);
}

test "findPosition can scan backward from the cursor and split item" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "ABCD");

    const position = try text.findPosition(2);
```

```
    try std.testing.expectEqual(0, position.left);
    try std.testing.expectEqual(1, position.right);
    try std.testing.expectEqual(2, text.items.items[0].getClockLen());
    try std.testing.expectEqual(2, text.items.items[1].getClockLen());
    try std.testing.expectEqual(1, text.end);
}

test "findPosition can use the document end as a backward anchor" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "A");
    try text.insert(1, "B");
    try text.insert(2, "C");
    text.position_cursor = null;

    const position = try text.findPosition(2);

    try std.testing.expectEqual(1, position.left);
    try std.testing.expectEqual(2, position.right);
    try std.testing.expectEqual(2, text.end);
}

//==========================================================================
// splitItem
//==========================================================================

test "splitItem splits string item at given offset" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "ABC");

    const right_handle = try text.splitItem(0, 1);

    try std.testing.expectEqual(1, text.items.items[0].getClockLen());
    try std.testing.expectEqual(2, text.items.items[right_handle].getClockLen());
                                        try          std.testing.expectEqualStrings("A",
text.sliceBytes(text.items.items[0].content.string));
                                       try          std.testing.expectEqualStrings("BC",
text.sliceBytes(text.items.items[right_handle].content.string));
    try std.testing.expectEqual(0, text.items.items[1].left);
```

```
}

test "splitItem splits string item at the end" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "ABC");

    const right_handle = text.splitItem(0, 3);
    try std.testing.expectError(error.IndexOutOfBounds, right_handle);
}

//==========================================================================
// getItemCleanStart/End
//==========================================================================

test "getItemCleanStart splits item if target clock is in the middle of it" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "Hello");

    const handle = try text.getItemCleanStart(.{ .client = 1, .clock = 2 });

    try std.testing.expectEqual(2, text.items.items[handle].id.clock);
    try std.testing.expectEqual(3, text.items.items[handle].getClockLen());
                                        try          std.testing.expectEqualStrings("llo",
text.sliceBytes(text.items.items[handle].content.string));
}

test "getItemCleanEnd splits item if target clock is in the middle of it" {
    const allocator = std.testing.allocator;
    var text = TextImpl.init(allocator, 1);
    defer text.deinit();

    try text.insert(0, "Hello");

    const handle = try text.getItemCleanEnd(.{ .client = 1, .clock = 2 });

    try std.testing.expectEqual(0, text.items.items[handle].id.clock);
    try std.testing.expectEqual(3, text.items.items[handle].getClockLen());
                                        try          std.testing.expectEqualStrings("Hel",
```

```
    text.sliceBytes(text.items.items[handle].content.string));
    }
```