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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

Дипломний проєкт

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

на тему: Спосіб захищеного розподіленого зберігання даних

Виконав (-ла): студент (-ка) 4 курсу, групи ІП-83
(шифр групи)

Розгон Олександр Олександрович
(прізвище, ім’я, по батькові) (підпис)

Керівник доц., к.т.н., доц. Волокита А.М. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) професор, д.т.н. Сімоненко В. П. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(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 – 2022 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
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Розгона Олександра Олександровича

1. Тема проєкту *Спосіб захищеного розподіленого зберігання даних*
керівник проєкту Волокита Артем Миколайович, доцент, к.т.н., доц.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету

2. Термін здачі студентом закінченого проєкту *11 червня 2022 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Розділ 1. Огляд і порівняльний аналіз систем зберігання даних.
Розділ 2. Огляд використаних технологій та інструментів.
Розділ 3. Розробка та проектування власної систем.
Розділ 4. Тестування та аналіз розробленої системи
5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень) структурна схема системи, функціональна схема, алгоритм пошуку даних в розподіленій системі.

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В. П.		

7. Дата видачі завдання «10» грудня 2021 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2021-15.12.2021*	
2.	*Вивчення та аналіз завдання*	*15.12.2021-15.03.2022*	
3.	*Розробка архітектури та загальної структури системи*	*15.04.2022-25.04.2022*	
4.	*Розробка структур окремих підсистем*	*25.04.2022-5.05.2022*	
5.	*Програмна реалізація системи*	*5.05.2022-15.05.2022*	
6.	*Оформлення пояснювальної записки*	*15.05.2022-20.05.2022*	
7.	*Захист програмного продукту*		
8.	*Передзахист*		
9.	*Захист*		

Студент-дипломник _____ Олександр РОЗГОН
(підпис)

Керівник проєкту _____ Артем ВОЛОКИТА
(підпис)

АНОТАЦІЯ

У даній роботі було розглянуто системи захищеного розподіленого зберігання даних, проаналізовано наявні рішення та їх принципи роботи. На основі їх переваг та недоліків спроектовано каркас системи, котра дозволяє покривати слабкі місця наявних систем, такі як: наявність незалежного пошуку по розподіленій мережі без використання централізованих індексаторів та допоміжних платформ, надання користувачу права самому обирати довірені вузли та налагоджувати з ними взаємодію, захищений доступ до даних на основі пар приватних та публічних ключів та адрес користувачів. В роботі спроектовано та реалізовано основний функціонал системи, котрий дозволяє вирішити перечисленні вище проблеми та розвивати інфраструктуру проекту інтегруючи в неї різноманітні правила взаємодії на рівні вузла та користувача.

Програмний продукт був розроблений на мові програмування JavaScript та платформі Node.JS, використовуючи систему керування реляційними базами даних MySQL.

Ключові слова: децентралізація, розподілена система, захищене зберігання даних, однорангова мережа, JavaScript, Node.JS.

ANNOTATION

In this paper, the systems of secure distributed data storage were considered, the existing solutions and their principles of operation were analyzed. Based on their advantages and disadvantages, the framework of the system is designed to cover the weaknesses of existing systems, such as: independent search on a distributed network without the use of centralized indexers and auxiliary platforms, giving the user the right to choose and configure trusted nodes. interaction, secure access to data based on pairs of private and public keys and user addresses. The main functionality of the system is designed and implemented, which allows to solve the above problems and develop the project infrastructure by integrating various rules of interaction at the node and user level.

The software product was developed in the JavaScript programming language and the Node.JS platform using the MySQL relational database management system.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Спосіб захищеного	4		
			розподіленого зберігання			
			даних			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Спосіб захищеного	87		
			розподіленого зберігання			
			даних			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Спосіб захищеного	1		
			розподіленого зберігання			
			даних			
			Функціональна схема			
	А4	***ІАЛЦ.4672008.005 Д2***	Спосіб захищеного	1		
			розподіленого зберігання			
			даних			
			Структурна схема системи			
	А4	***ІАЛЦ.467200.006 Д3***	Спосіб захищеного	1		
			розподіленого зберігання			
			даних			
			Алгоритм пошуку даних в			
			розподіленій системі			
	А4	***ІАЛЦ.467200.007 Д4***	Спосіб захищеного	51		
			розподіленого зберігання			
			даних			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА	
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***		
Розроб		Розгон О.О.			*Спосіб захищеного розподіленого зберігання даних* Опис альбому	Літ. / Аркуш: 1 / Аркушів: 1
Перев		Волокита А.М.				
						КПІ ім. Ігоря Сікорського ФІОТ ІП-83

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Спосіб захищеного розподіленого зберігання даних»

Київ – 2022

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 2

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ 2

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 2

4 ДЖЕРЕЛА РОЗРОБКИ 3

5 ТЕХНІЧНІ ВИМОГИ 3

Вимоги до розробленого продукту 3

Вимоги до програмного забезпечення 4

Вимоги до апаратної частини 4

6 ЕТАПИ РОЗРОБКИ 4

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи захищеного зберігання даних на основі децентралізованої однорангової мережі, а також на подальшу підтримку та вдосконалення розробленої системи інтегруючи в неї правила взаємодії користувача та віддаленого вузла та оптимізації роботи самої мережі.

Областю застосування цієї системи є розширення можливостей глобальної децентралізації всієї мережі та конкретних рішень, зменшення впливу централізованих сервісів зберігання даних, розробка децентралізованих сховищ даних з вбудованим пошуком по мережі без додаткових централізованих індексаторів та сторонніх сервісів, розробка розподілених файлових систем з захищеним доступом до даних.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи розподіленого захищеного зберігання даних із використанням однорангової мережі та швидкого пошуку по ній, системи авторизації на основі пар приватних та публічних ключів, що дозволить досягти справжньої децентралізації в мережі та незалежності від додаткових централізованих інструментів та сервісів індексаторів.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література та власний досвід роботи з різними децентралізованими мережами.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Наявність рівноправної однорангової мережі та правил поведінки та взаємодії в ній.
- Надання користувачу права самому приймати рішення щодо взаємодії з конкретними вузлами.
- Надати можливість пошуку даних по мережі без застосування сторонній централізованих рішень та інструментів, таких як індексатори.
- Надати можливість безпечно зберігати дані та встановлювати рівні доступу до них.
- Надати можливість користувачам авторизуватися на основі пар закритих та відкритих ключів задля конфіденційності та захищеності власних секретів в розподіленій мережі.
- Система повинна бути легко підтримувана та дозволяти інтегрувати в неї нові правила поведінки в мережі.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Docker

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- не менше ніж 1ТБ для виробничих вузлів та 32ГБ для легких клієнтів постійної пам’яті.
- RAM не менше ніж 16 ГБ для виробничих вузлів та 2Гб для легких клієнтів.
- Для виробничих вузлів мережа повинна бути не менше 100Мб/с.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2021-15.12.2021
Вивчення та аналіз завдання	15.12.2021-15.03.2022
Розробка архітектури та загальної структури системи	15.04.2022-25.04.2022
Розробка структур окремих частин системи	25.04.2022-5.05.2022
Програмна реалізація системи	5.05.2022-15.05.2022
Виправлення помилок	15.05.2022-20.05.2022
Оформлення пояснювальної записки	25.04.2022

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Спосіб захищеного розподіленого зберігання даних*»

Київ – 2022

ЗМІСТ

ВСТУП 4

РОЗДІЛ 1. ОГЛЯД І ПОРІВНЯЛЬНИЙ АНАЛІЗ СИСТЕМ ЗБЕРІГАННЯ ДАНИХ..... 6

1.1 Типи систем. Централізовані, розподілені, децентралізовані системи їх характеристика та порівняння 6

1.2 Розподілена та децентралізована файлові системи 10

1.3 Приклади існуючих децентралізованих файлових систем 13

ВИСНОВОК ДО РОЗДІЛУ 1 27

РОЗДІЛ 2. ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ 28

2.1 JavaScript та декларування типів за допомогою TypeScript(d.ts) 28

2.2 Платформа Node.Js 31

2.3 Система управління реляційними базами даних MySQL 32

2.4 Транспорт та мережа 35

2.5 Безпека та конфіденційність в розподіленій мережі 37

2.6 Інструменти розробки та розгортання 38

2.7 IPFS протокол 41

ВИСНОВОК ДО РОЗДІЛУ 2 47

РОЗДІЛ 3. РОЗРОБКА ТА ПРОЕКТУВАННЯ СИСТЕМИ 48

3.1 Вимоги до розроблюваної децентралізованої файлової системи. 48

3.2 Проектування архітектури системи 49

3.2.1 Рівень доступу до даних 49

3.2.2 Внутрішня мережа. Взаємодія між вершинами та транспорт..... 53

3.2.3 Зовнішня мережа. Транспорт для інтерфейсу взаємодії користувача..... 55

3.2.4 Узагальнення розробленої інфраструктури вузла..... 56

3.3 Розробка дизайну системи 59

3.3.1 Проектування сховища..... 59

3.3.2 Авторизація та верифікація користувачів 62
3.3.3 Пошук даних в мережі 66
3.3.4 Проектування модулів системи та їх взаємодія 68
ВИСНОВОК ДО РОЗДІЛУ 3 73
РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ 74
ВИСНОВОК ДО РОЗДІЛУ 4 81
ВИСНОВКИ 82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 84

ВСТУП

Найбільшим та найвпливовішим напрямком двадцятих років двадцять першого століття стала децентралізація. Люди прагнуть позбутися залежності від централізованих органів управляння, провайдерів сервісів та централізованих систем. Напрямок розподілених систем стрімко розвивався і раніше, але з появою блокчейну та можливістю врегулювання відносин між суб'єктами за допомогою розумних контрактів, котрі здатні виконувати роль незалежного регулятора, з'явилось таке поняття, як Web3. Це новий етап еволюції глобальної мережі, де кожен виступає рівноправним її учасником і не залежить від будь яких провайдерів сервісів та їх централізованих систем з їх правилами гри.

На даному етапі розвитку децентралізованих систем, ми вже можемо створювати децентралізовану банківську систему, заключати угоди, де немає централізованого регулятора. З'явились децентралізовані сховища даних такі, як BitTorrent та IPFS. Але майже всі розподілені та децентралізовані системи перебувають в залежності від централізованих. По справжньому зручними у використанні вони стають лише у симбіозі з централізованими системами, котрі досліджують їх мережу та індексують її, а отже дозволяють переглядати її стан, історію та виконувати пошук потрібних нам даних.

На даному етапі, децентралізовані банки перебувають у залежності централізованих бірж, обмінників та індексаторів. Децентралізовані файлові сховища перебувають у залежності від централізованих пошукових систем та індексаторів їх вмісту. Наступним кроком еволюції даних систем буде позбавлення від цієї залежності. Наприклад, вже деякі блокчейн системи підтримують різні операції для обміну та торгів і це справді революційний напрям.

Незважаючи на це, ще не існує жодної розподіленої файлової системи з можливістю пошуку по власній мережі даних, що дозволить стати їй по справжньому незалежною від централізованої мережі та її сервісів. Також,

немає жодної розподіленої файлової системи, котра б дозволяла зберігати власні дані ізольовано від інших користувачів та закриваючи доступ до них.

Ця дипломна робота присвячена вирішенню цього питання. Ми розробимо безпечну децентралізовану файлову систему, котра б мала можливість рефлексивного пошуку по власній мережі та різні рівні захисту файлів користувача на основі системи ролей та прав користувачів, що матимуть змогу авторизуватися в мережі за допомогою пари публічного та приватного ключа. Буде спроектовано та розроблено протокол взаємодії вузлів та клієнтів в мережі та реалізовано базовий функціонал.

Система буде гнучкою, що дасть їй можливість розвиватися в різних напрямках та ставати по справжньому незалежною, що покриває всі потреби користувачів або ж однієї із складових більш складних систем.

РОЗДІЛ 1.
ОГЛЯД І ПОРІВНЯЛЬНИЙ АНАЛІЗ СИСТЕМ ЗБЕРІГАННЯ ДАНИХ.

1.1 Типи систем. Централізовані, розподілені, децентралізовані системи їх характеристика та порівняння

Основні типи організації систем:

- Централізовані
- Розподілені
- Децентралізовані

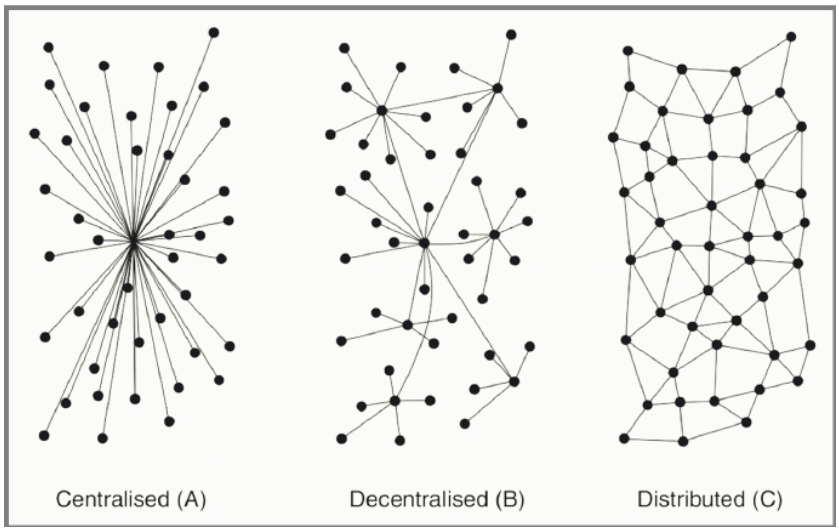


Рисунок 1.1 – Типи організації систем [1].

Централізовані системи – це системи, які використовують архітектуру клієнт/сервер, де один або кілька клієнтських вузлів безпосередньо підключені до центрального сервера. Це найбільш часто використовуваний тип системи в багатьох організаціях, де клієнт надсилає запит на сервер компанії та отримує відповідь. [1, 2]

Характеристики централізованої системи:

- **Наявність глобального годинника:** оскільки вся система складається з центрального вузла (сервер/головний) і багатьох клієнтських вузлів (комп'ютер/підпорядкований), усі клієнтські вузли синхронізуються із глобальним годинником (годинником центрального вузла).

- **Один єдиний центральний блок:** один центральний блок, який обслуговує/координує всі інші вузли в системі.
- **Залежний збій компонентів:** збій центрального вузла призводить до відмови всієї системи. Це має сенс, оскільки, коли сервер не працює, жоден інший об’єкт не може надсилати/отримувати відповіді/запити.

Масштабування:

- можливе лише вертикальне масштабування на центральному сервері. Горизонтальне масштабування буде суперечити єдиній центральної одиниці, характерній для цієї системи єдиної центральної сутності.[1]

Переваги централізованої системи -

- Легко фізично захищати. Серверні та клієнтські вузли легко захищати й обслуговувати завдяки їхньому місцезнаходження
- Гладкий та елегантний особистий досвід – клієнт має спеціальну систему, яку він використовує (наприклад, персональний комп’ютер), а компанія має подібну систему, яку можна модифікувати відповідно до власних потреб.
- Виділені ресурси (пам'ять, ядра процесора тощо)
- Більш рентабельні для малих систем до певної межі – оскільки центральні системи потребують менше коштів для налаштування, вони мають перевагу, коли потрібно будувати невеликі системи
- Можливі швидкі оновлення – лише один комп’ютер для оновлення.
- Легке від'єднання вузла від системи.

Недоліки централізованої системи -

- Дуже залежить від підключення до мережі – система може вийти з ладу, якщо вузли втратять зв’язок, оскільки є лише один центральний вузол.
- Немає витонченої деградації системи – різкий вихід з ладу всієї системи

- Менша можливість резервного копіювання даних. Якщо вузол сервера виходить з ладу і немає резервної копії, ви відразу втрачаєте дані
- Складне обслуговування сервера. Існує лише один серверний вузол, і через доступність неефективно та непрофесійно вилучати сервер для обслуговування. Таким чином, оновлення потрібно виконувати на льоту (гарячі оновлення), що важко, і система може зламатися.

Розподілена система – це обчислювальне середовище, в якому різні компоненти розподілені на кількох комп'ютерах у мережі. Ці пристрої розподіляють свою роботу, координуючи свої зусилля, щоб виконати роботу ефективніше або надійніше, ніж якби за завдання відповідав один пристрій.[1, 2]

Розподілені системи є важливим розвитком для IT та інформатики, оскільки зростаюча кількість пов'язаних робіт є настільки масовими та складними, що один комп'ютер не зможе впоратися з ними самостійно. Але розподілені обчислення також пропонують додаткові переваги в порівнянні з традиційними обчислювальними середовищами. Розподілені системи зменшують ризики, пов'язані з наявністю єдиної точки відмови, підвищуючи надійність і відмовостійкість. Сучасні розподілені системи, як правило, розраховані на масштабування майже в реальному часі; Крім того, ви можете на льоту залучати додаткові обчислювальні ресурси, підвищуючи продуктивність і ще більше скорочуючи час до завершення.

Характеристики розподіленої системи:

- **Паралелізм компонентів:** вузли застосовують протоколи консенсусу, щоб узгодити ті самі значення/транзакції/команди/журнали.
- **Відсутність глобального годинника:** всі вузли підтримують свої власні годинники.
- **Незалежний збій компонентів:** у розподіленій системі вузли виходять з ладу незалежно, не маючи суттєвого впливу на всю систему. Якщо

один вузол виходить з ладу, вся система без збою вузла продовжує працювати.

Масштабування:

- можливе горизонтальне та вертикальне масштабування.

Переваги розподіленої системи -

- Низька затримка, ніж у централізованої системи. Розподілені системи мають низьку затримку через високе географічне поширення, що призводить до меншого часу для отримання відповіді.

Недоліки розподіленої системи -

- Важко досягти консенсусу
- Звичайний спосіб реєстрації подій за абсолютним часом їх виникнення тут неможливий

Децентралізація – це процес, за допомогою якого діяльність системи або організації, зокрема діяльність щодо прийняття рішень або планування розподіляється або делегується від центрального авторитетного місця до самоорганізованих однорангових структур.[2]

Децентралізована система – взаємопов'язана інформаційна система, де жоден суб'єкт не є єдиним авторитетним центральним компонентом, котрий може мати можливість прямого впливу на всю систему. У контексті обчислювальної техніки та інформаційних технологій децентралізовані системи зазвичай мають форму мережевих комп'ютерів.[1, 3]

Для підприємств і організацій переваги децентралізованої системи включають стійкість до відмов і резервування за ціною додаткової складності управління. У будь-якій організації інформаційних технологій децентралізація повинна бути збалансована з централізацією при розробці мережевої системи, щоб оптимізувати баланс витрат і вигод.

Децентралізовані системи тісно пов'язані з ідеєю **самоорганізації** - процес за якого локальні взаємні компоненти системи встановлюють порядок і

координацію для досягнення глобальних цілей без центрального командного впливу.

Децентралізована система відрізняється від розподіленої системи. Децентралізована система, як правило, має кілька авторитетних вузлів, кожен з яких обслуговує підмножину загальної кількості кінцевих користувачів. У розподіленій системі, однак, немає кінцевих користувачів, оскільки кожен вузол мережі спілкується з кожним іншим, щоб вести себе як єдине ціле

Характеристики децентралізованої системи –

- **Відсутність глобального годинника:** кожен вузол незалежний один від одного і, отже, має різні годинники, які вони запускають і слідують.
- **Кілька центральних блоків (комп'ютери/вузли/сервери):** більше одного центрального блоку, який може прослуховувати з'єднання з інших вузлів
- **Залежний збій компонентів:** одна збій центрального вузла призводить до відмови частини системи; не вся система

Масштабування

- можливе вертикальне масштабування. Кожен вузол може додавати ресурси (апаратне забезпечення, програмне забезпечення) до себе, щоб підвищити продуктивність, що призводить до підвищення продуктивності всієї системи.

1.2 Розподілена та децентралізована файлові системи

Розподілена файлова система — це фізично розподілена реалізація класичної моделі розподілу традиційної файлової системи, що дозволяє користувачам безперешкодно маніпулювати, організовувати та обмінюватися даними, незалежно від їх фактичного розташування в мережі.[4]

Розподілена файлова система знаходиться на різних машинах або сайтах і пропонує єдине логічне представлення даних, розкиданих між машинами/сайтами, як локальними, так і віддаленими.[5]

Які переваги розподіленої файлової системи?

- Багатокористувацький, локальний/віддалений обмін файлами
- Уніфікований і послідовний перегляд спільних папок/ресурсів
- Прозорість доступу до даних і незалежність від місця розташування
- Доступність даних навіть у разі збою сервера або диска
- Покращена доступність файлів, час доступу та ефективність мережі
- Ефективне балансування навантаження та прозорість розташування файлів
- Підвищена масштабованість і сумісність
- Підвищена безпека та адміністративна гнучкість

За допомогою розподіленої файлової системи клієнти можуть отримати доступ до даних, що знаходяться на дисках сервера, віддалено, як якщо б дані були на локальних дисках відповідних клієнтів.

Але чому б не зберігати дані на локальних дисках клієнтів, що скорочують пропускну здатність мережі через операції отримання даних? Навіщо тримати його на сервері централізовано?

Збереження даних на одному з кількох серверів дозволяє їм залишатися послідовними. Уявіть, що клієнти зберігають відповідні дані та регулярно оновлюють їх локально. Це зробить дані значною мірою неузгодженими в мережі. Кожен клієнт матиме власну версію даних, чого ми, звичайно, не хочемо.

Якщо дані узгоджені, ними можна легко поділитися між клієнтами без будь-якої метушні. Крім того, його можна легко створити, якщо він знаходиться на кількох серверах, а не на значно більшій кількості клієнтів. Крім того, централізоване зберігання даних на сервері робить їх більш безпечними.

У типовій розподіленій файловій системі клієнт-сервер і клієнт, і сервер містять файлову систему. Файлова система на обох кінцях розподіленої файлової системи дозволяє клієнту читати та оновлювати дані на сервері, як на локальный диск клієнта.

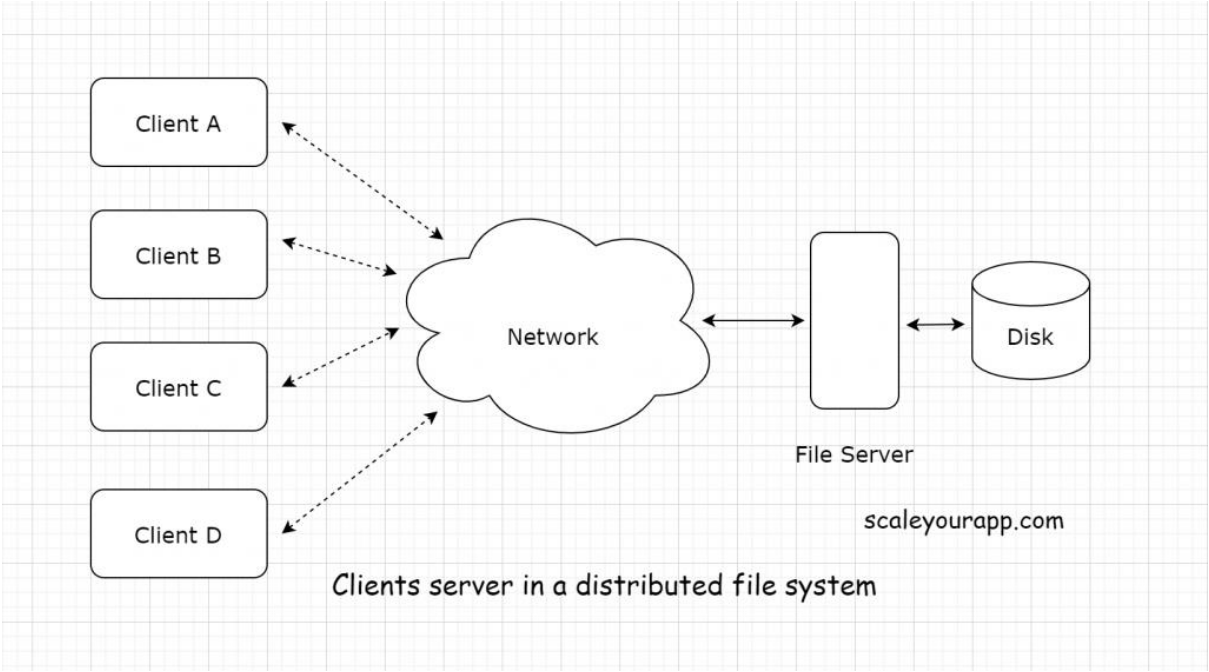


Рисунок 1.2 – Клієнт-серверна архітектура у розподіленій системі [5].

Децентралізована файлова система або «Міжпланетна файлова система» - це система, де дані розподіляються на різні блоки та зберігаються всередині різних вузлів однорангової мережі. Зазвичай система забезпечує унікальний простій імен та ідентифікаторів файлів. У минулому було кілька успішних децентралізованих систем обміну файлами, таких як BitTorrent, Napstep …

Однак ці програми не були розроблені як інфраструктура, на якій слід розбудовувати. Перевага використання такого сховища полягає в тому, щоб служити тій же цілі, що й децентралізована мережа (тобто, безпека, конфіденційність, відсутність єдиної точки збою, рентабельність).

Наступною сходинкою таких систем стали системи на основі блокчейну, котрі мають методи стимуляції вузлів. Найпопулярнішою технологією наразі стала IPFS(*InterPlanetary File System*).

Наразі IPFS є однією з найбільш обговорюваних технологій в екосистемі DApps, оскільки вона може замінити HTTP. Це система зберігання з адресою вмісту – для кожного збереженого об’єкта (наприклад, файли, зображення, відео) створюється унікальний криптографічний хеш, і той самий хеш потрібен для отримання цих об’єктів. Подібно до URI в Інтернеті, це розподілена

файлова система, яка прагне з'єднати всі обчислювальні пристрої з однією системою файлів. Він натхненний попередніми успішними одноранговими системами, включаючи DHT, BitTorrent, Git і SFS.

IPFS також є одним із життєво важливих компонентів децентралізованої архітектури Skeps. IPFS разом з асиметричним шифруванням допомагає Skeps безпечно обмінюватися файлами між вузлами.

Цифрова валюта Filecoin створена для стимулювання зберігання даних у мережі IPFS. Протокол Filecoin забезпечує зберігання та пошук даних через мережу незалежних постачальників даних. Ці постачальники сховища не покладаються на єдиного координатора, де клієнти платять за зберігання та отримання даних, майнери сховища заробляють токени, пропонуючи сховище, а майнери пошуку заробляють токени, обслуговуючи дані.

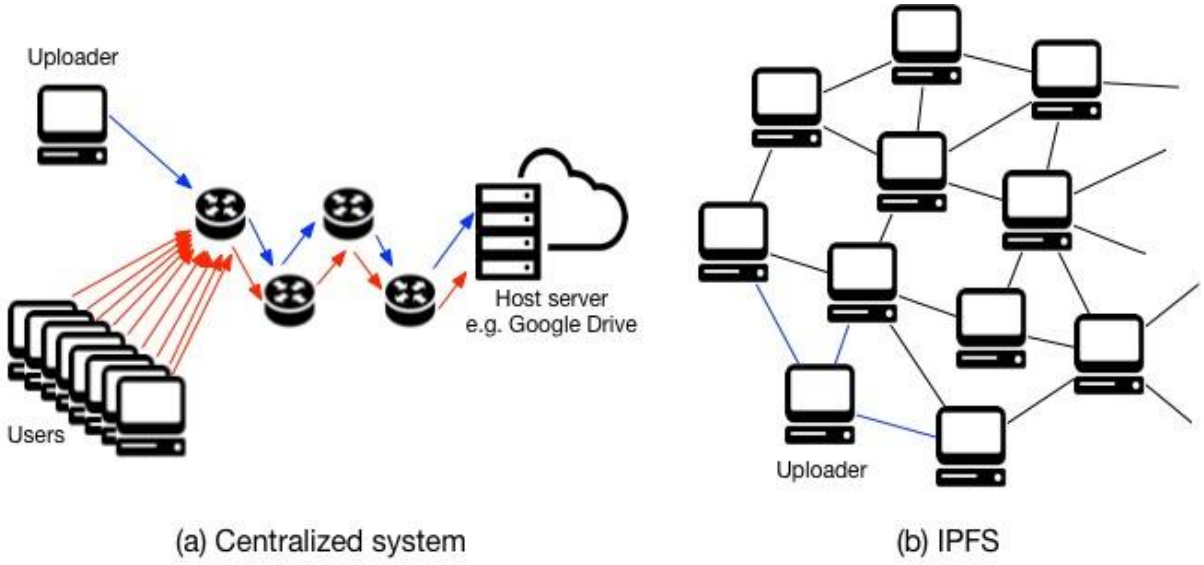


Рисунок 1.3 – Відмінність між централізованою системою та IPFS системою [6].

Також, існують і інші не менш цікаві та популярні технології, деякі з них будуть описані у наступних розділах.

1.3 Приклади існуючих децентралізованих файлових систем

Існує багато готових систем розподіленого зберігання даних. Деякі найкращі рішення розглянемо, задля визначення їх сильних та слабких сторін. Даний

аналіз будемо використовувати задля визначення проблематики, котру необхідно вирішити, проектуючи та розроблюючи власну систему.

1.3.1 BitTorrent

BitTorrent — це одноранговий протокол, що означає, що комп'ютери в BitTorrent «зграї» (група комп'ютерів, які завантажують і завантажують один і той же торрент) передають дані між собою без необхідності використання центрального сервера.[9]

Традиційно комп'ютер приєднується до зграї BitTorrent, завантажуючи файл .torrent у клієнт BitTorrent. Клієнт BitTorrent зв'язується з «трекером», зазначеним у файлі .torrent. Трекер – це спеціальний сервер, який відстежує підключені комп'ютери. Трекер ділиться своїми IP-адресами з іншими клієнтами BitTorrent у зграї, дозволяючи їм підключатися один до одного.

Після підключення клієнт BitTorrent завантажує фрагменти файлів у торренті невеликими шматочками, завантажуючи всі дані, які може отримати. Як тільки клієнт BitTorrent отримає певні дані, він може почати завантажувати ці дані іншим клієнтам BitTorrent у зграї. Таким чином, кожен, хто завантажує торрент, також завантажує той самий торрент. Це прискорює швидкість завантаження кожного. Якщо 10 000 людей завантажують один і той же файл, це не створює великого навантаження на центральний сервер. Замість цього кожен завантажувач надає пропускну здатність завантаження іншим завантажувачам, забезпечуючи швидкість торрента.

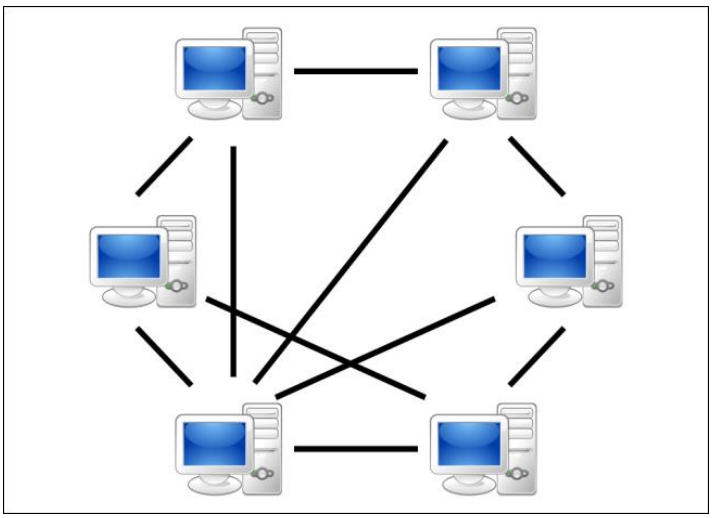


Рисунок 1.4 – будова однорангової мережі[10]

Після підключення клієнт BitTorrent завантажує фрагменти файлів у торренті невеликими шматочками, завантажуючи всі дані, які може отримати. Як тільки клієнт BitTorrent отримає певні дані, він може почати завантажувати ці дані іншим клієнтам BitTorrent у зграї. Таким чином, кожен, хто завантажує торрент, також завантажує той самий торрент. Це прискорює швидкість завантаження кожного. Якщо 10 000 людей завантажують один і той же файл, це не створює великого навантаження на центральний сервер. Замість цього кожен завантажувач надає пропускну здатність завантаження іншим завантажувачам, забезпечуючи швидкість торрента.

Важливо, що клієнти BitTorrent ніколи не завантажують файли з самого трекера. Трекер бере участь у торренті лише шляхом відстеження клієнтів BitTorrent, підключених до зграї, а не фактично завантажуючи чи завантажуючи дані.

Останнім часом децентралізована торрент-система «без відстеження» дозволяє клієнтам BitTorrent спілкуватися один з одним без необхідності використання будь-яких центральних серверів. Клієнти BitTorrent використовують для цього технологію розподіленої хеш-таблиці (DHT), причому кожен клієнт BitTorrent функціонує як вузол DHT. Коли ви додаєте торрент за допомогою «магнітного посилання», вузол DHT зв’язується з

сусідніми вузлами, а ці інші вузли зв’язуються з іншими вузлами, доки вони не знайдуть інформацію про торрент.

BitTorrent File System (BTFS) - це файлова система, що включає подвійну природу зберігання та передачі. Користувачі можуть забезпечити певну стабільність зберігання, закріпивши свої файли на своїх вузлах BTFS. Коли інші вузли мережі запитують файл, цей файл дублюється через запитувачі. BTFS також забезпечує ефективну та безпечну передачу файлів між вузлами, розташованими в різних регіонах. Це робить BTFS ідеальним вибором для передачі файлів.[9]

Хоча продуктивність передачі файлів за допомогою BTFS порівнянна з іншими централізованими службами, BTFS має додаткову гарантію безпечної передачі P2P. Централізовані сховища файлів потенційно можуть бути скомпрометовані, що призведе до розкриття, викрадення або пошкодження конфіденційних файлів.

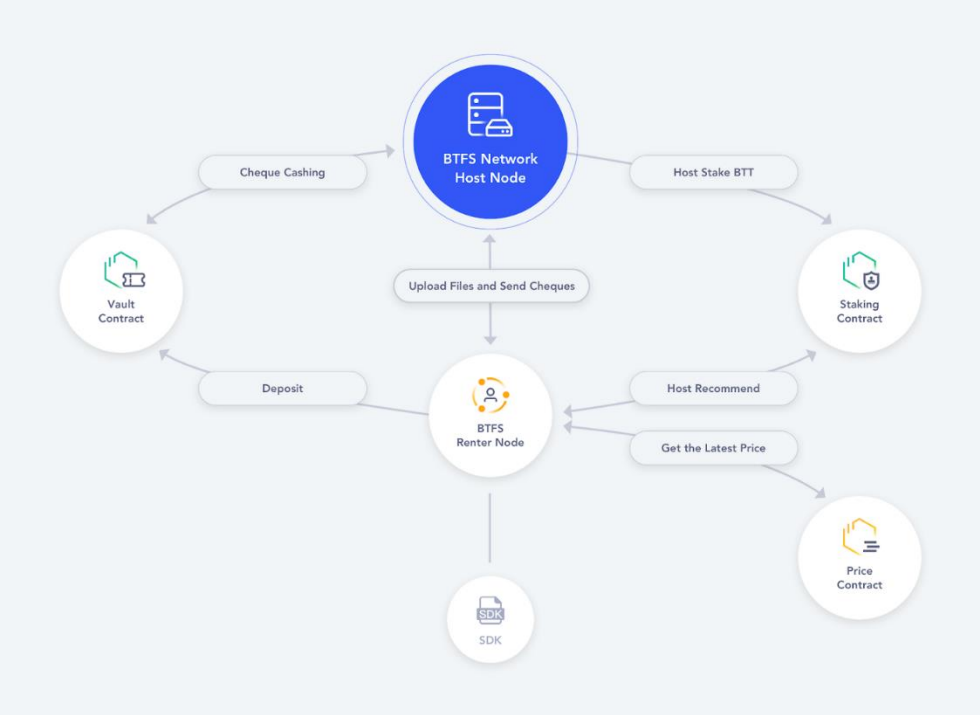


Рисунок 1.5 – будова інфраструктури BTFS технології [11]

BTFS є форком IPFS. Однак BTFS містить ряд унікальних функцій. Наприклад, BTFS підтримує як видалення файлів, так і

шифрування/дешифрування файлів. Це дозволяє користувачам легко видаляти незаконні або захищені авторським правом матеріали на своїх вузлах, а також безпечно передавати та зберігати файли. BTFS також унікально інтегрований з блокчейном TRON, оскільки користувачі можуть ініціювати вузли через гаманці, які підтримують TRON.

Користувачі можуть завантажувати файли в мережу BTFS через демона BTFS. Коли незакріплені файли зберігаються в кеші, вони зберігаються до тих пір, поки або вузол не припинить роботу, або збережена кількість не перевищить ліміт збору сміття. Файли дублюються, коли їх запитує інший вузол BTFS.

Зацікавлені користувачі можуть підключитися до BTFS, встановивши вузол. Майбутні плани, включаючи упаковку BTFS з клієнтом BT, щоб користувачі BT Client мали можливість встановити та запустити демон BTFS. Після запуску демон BTFS підключається до мережі через вузли завантаження, розгорнуті на екземплярах по всьому світу.[12]

1.3.2 Pinata (IPFS Pinning service)

Pinata — це перший мультимедійний центр для творців, розробників і художників NFT у децентралізованому світі Web 3. Бачення Pinata — сприяти тому, щоб кожен автор відчував свій простір в Інтернеті. Найбільша служба закріплення на IPFS, Pinata дає користувачам можливість завантажувати, керувати та ділитися своїм вмістом будь-де, де завгодно і з ким вони хочуть. [13]

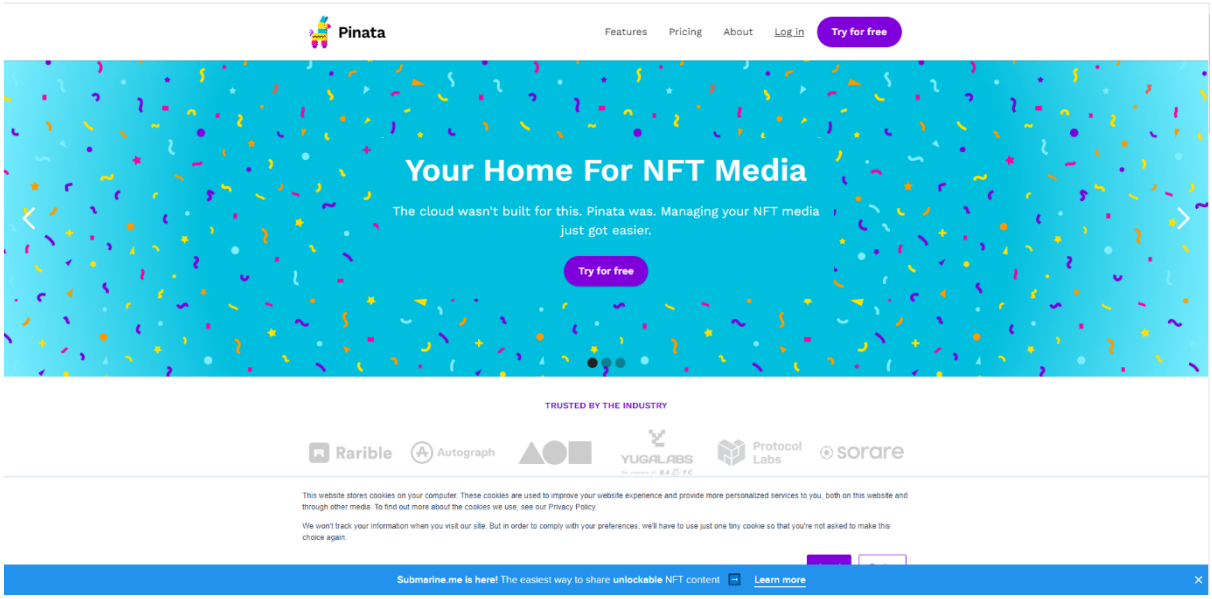


Рисунок 1.6 – Головна сторінка веб сайту Pinata. [14]

З 2018 року Pinata стала загальною назвою, на яку користувачі покладалися для потреб міжпланетної файлової системи (IPFS), одночасно обслуговуючи зростаючу базу користувачів NFT. Потужний, але простий у використанні продукт Pinata дозволяє користувачам завантажувати, обмінюватися файлами та керувати ними з неперевершеною швидкістю по всій мережі IPFS. Однією з проблем, на яку звернула їхня увагу їхня база користувачів, було те, що оскільки IPFS є загальнодоступною мережею, їй не вистачає конфіденційності та безпеки, які потрібні багатьом користувачам. Завдяки дослідженням користувачів і відгукам клієнтів команда Pinata зробила висновок, що в просторі NFT є багато користувачів, які потребують більшого захисту свого вмісту, не втрачаючи можливості ділитися та відображати свою роботу будь-де.

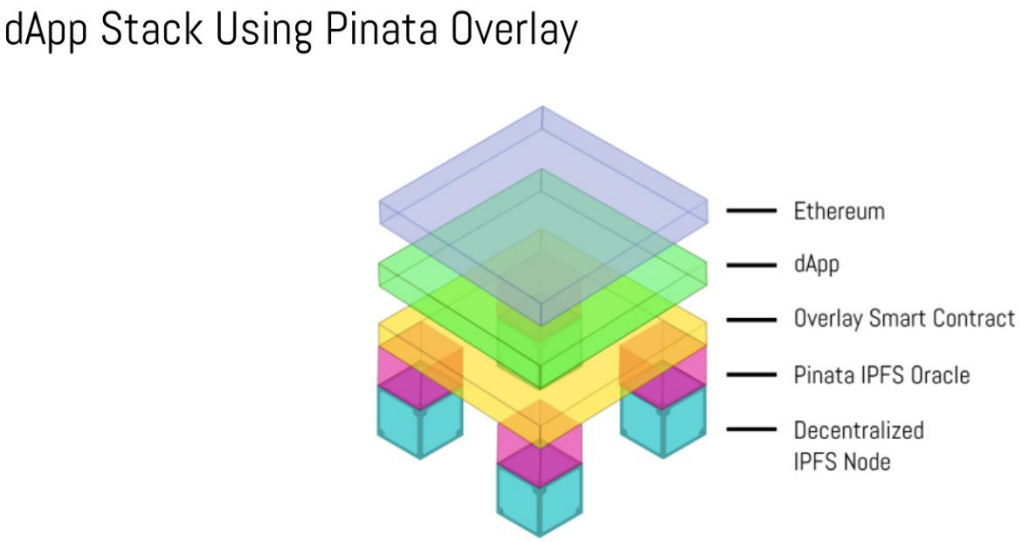


Рис.1.7. Приклад будови Web3 застосунку з використанням IPFS [15]

Додатковий рівень захисту для NFT

Таким чином, з'явилася найновіша функція Pinata, Submarining. Submarining дозволить користувачам у професійному плані Pinata вмикати та вимикати функцію «Підводного човна», коли вони завантажують файл на платформу. Якщо користувач не хоче, щоб файл був доступний у загальнодоступній мережі IPFS, він може просто увімкнути «Підводний човен», який приховає інформацію з мережі. Хоча функція все ще знаходиться в бета-версії, компанія продовжує розширювати свої пропозиції для потреб користувачів. За допомогою Submarining користувачі тепер можуть встановлювати такі властивості, як, хто може переглядати файли NFT, які перебувають у підводному стані, як довго вони можуть переглядати їх тощо.

1.3.3 Arweave

Arweave — це новий тип сховища, який підтримує дані стійкими та постійними ресурсами (токенами, що зберігаються в рамках протоколу, які отримують вигоду від інфляції та зниження вартості зберігання протягом тривалих періодів часу). Це дозволяє користувачам і розробникам зберігати

дані вічно. Arweave діє як жорсткий диск, який належить колективній власності, і дозволяє своїм користувачам зберігати цінну інформацію, програми та історію на невизначений термін.[16]

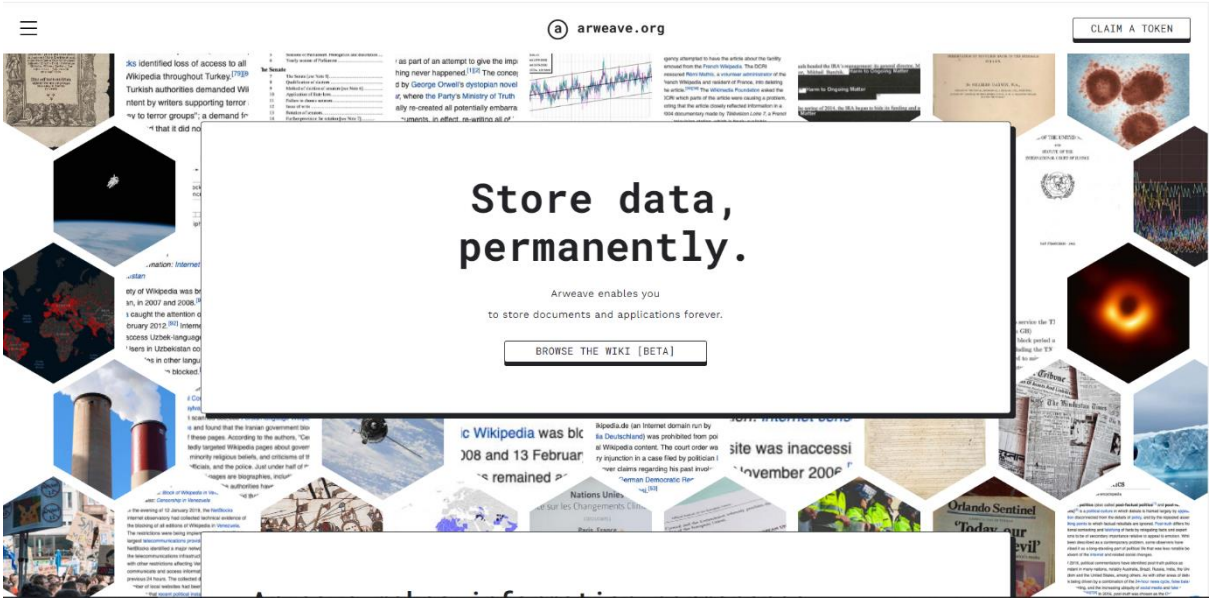


Рисунок 1.8 – Головна сторінка проекту “Arweave” [17]

Протокол Arweave поєднує схожий на торрент рій зацікавлених майнерів із величезним колективним місцем на жорсткому диску з тими особами та організаціями, яким потрібно постійно зберігати дані або розміщувати вміст. Це досягається в децентралізованій мережі, і всі дані, що зберігаються, підкріплюються винагородами за видобуток блоків і стійким фондом, який гарантує їх безперервну доступність.

Протокол поєднує людей, які мають вільний простір на жорсткому диску, з тими особами та організаціями, яким потрібно постійно зберігати дані або розміщувати вміст. Це схоже на те, як Uber з’єднує водіїв з людьми, яким потрібен транспорт. Це досягається в децентралізованій мережі, як-от біткойн, і всі дані, що зберігаються, підтримуються стійкими ресурсами, які забезпечують їх постійну доступність.

Спільнота Arweave створила повний набір інструментів і послуг, які дозволяють запускати **повністю децентралізовані** програми, які працюють на цій постійній мережі. Ці інструменти пропонують все: від хостингу

інтерфейсу користувача до запису бази даних і запитів , а також смарт-контракти для програмування фінансових винагород і заохочення. Цей набір інструментів можна компонувати та взаємозамінити, що дозволяє знайти ідеальне поєднання технологій, яке відповідає потребам вашого децентралізованого веб-додатка.

Мережа Arweave працює на основі токена AR, який можна знайти на більшості великих бірж.

Permaweb — це постійна і децентралізована мережа, побудована на основі Arweave. Permaweb складається з набору модульних і взаємозамінних протоколів.[18]

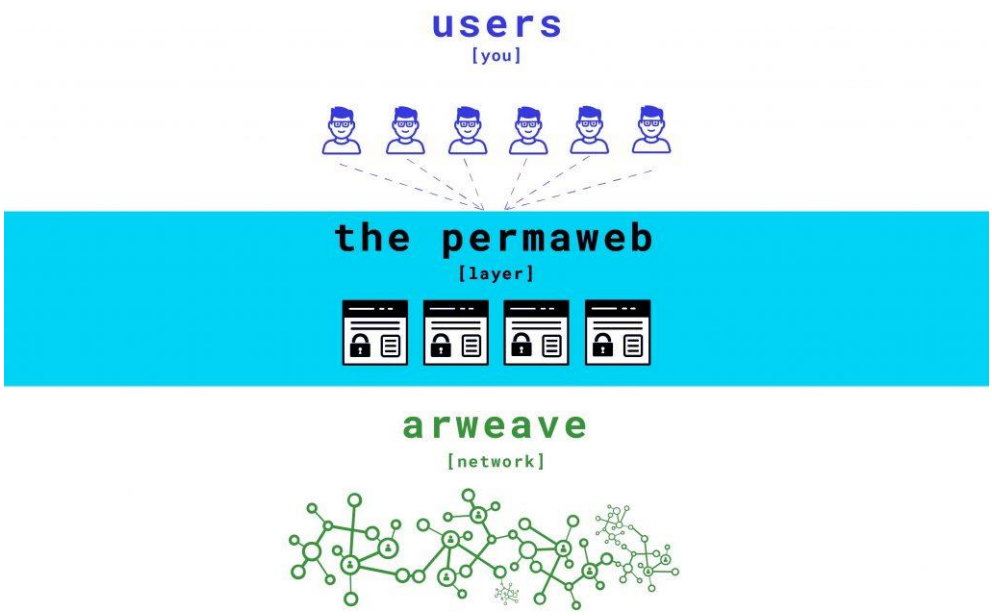


Рисунок 1.9 – Взаємодія між протоколом Arweave та мережую Permaweb [19]

Початкове бачення традиційної мережі полягало в тому, щоб дати людству можливість створити доступну колекцію всього, що нам відомо. Хоча цей проект був у значній мірі успішним, він страждає від одного критичного недоліку: Інтернет не має пам’яті. Коли документи видаляються з одного сервера в мережі, вони втрачаються з бази знань. Крім

того, програми, які колись розміщувалися в цій системі, більше не доступні, коли їх автори та супроводжувачі видаляють або змінюють їх.

Permaweb на Arweave — це наступна еволюція Інтернету, що дозволяє розробникам і користувачам ділитися стійкою базою знань, яка постійно розповсюджується у величезній кількості по всьому світу.

Базовим протоколом Permaweb є Arweave. Ця система базується на двох фундаментальних компонентах, масштабованій структурі, схожій на блокчейн, яка використовує нову систему майнінгу та стабільному ресурсі для забезпечення доступності та постійності даних із довільними тегами. На додаток до протокола Arweave є децентралізована мережа серверів шлюзу, яка робить дані легко доступними для веб-браузерів без необхідності модифікації або спеціального програмного забезпечення.

Основний механізм

Протокол BitTorrent пропонує, щоб вузли брали участь у грі, яка називається «оптимістичне око за око». У цій грі вузли обмінюються даними взаємно з іншими вузлами, які обмінюються даними з ними. Крім того, вузли іноді обмінюються даними випадковим чином. Ця гра веде до рівноваги Неша, в якій вузли обмінюються даними вільно, на максимальній потужності мережі.

Мережа Arweave покращує це поняття рівноправної оцінки та оптимістичного розподілу пропускної здатності, створюючи більш узагальнену форму «карми». У версії цієї гри від Arweave вузли відстежують «послуги », виконані для них іншими вузлами. Основною «послугою», яка була здійснена в мережі є вузли, які надають доступ до набора даних Arweave, передаючи дані з мережі іншим за запитом. Однак оператори вузлів можуть змінювати свій код, щоб відстежувати будь-які переваги (це можуть бути переваги, які вони отримують безпосередньо, або переваги для мережі в цілому). Учасники мережі використовують

інформацію про ці переваги для обчислення рейтингу однолітків. Потім учасник використовує ці рейтинги, щоб визначити, як вони будуть ділитися своїми ресурсами з іншими учасниками мережі, віддаючи перевагу тим, хто має вищі рейтинги, наприклад, ділиться з ними більшою пропускною здатністю. Потім, приблизно так само, як і з BitTorrent, вузли час від часу, «оптимістично» взаємодіяти один з одним, не беручи до уваги рейтинги однолітків. Це призводить до рівноваги Неша, коли всі вузли стимулюються вільно ділитися ресурсами та виконувати просоціальну поведінку.

Дизайн адаптивного механізму

Примітно, що система карми Arweave не застосовується на рівні блокчейну протоколу. Тобто не існує універсального консенсусу між вузлами щодо стану репутаційної гри або переходів станів у грі під час її роботи. Кожному вузлу залишається підтримувати власні рейтинги інших однорангових вузлів, без зобов'язань повідомляти, як і чому було визначено ці рейтинги, що означає, що кожен вузол може віддавати перевагу будь-якій поведінці інших вузлів, які вони вибирають. Це призводить до високомасштабованої ігрової структури, де накладні витрати на комунікацію щодо консенсусу практично не існують, але все ще досягають цілей механізму.[19]

Виходячи з цього, фактично немає консенсусу щодо того, що всі вузли навіть грають в одну і ту ж гру (винагороджуючи однакові «послуги» — розподіляючи ресурси однаково). Це дозволяє механізмам у грі бути дуже гнучкими та з часом адаптуватися до нових ситуацій у середовищі мережі — винагороджуючи та караючи нові форми поведінки, коли вони з'являються.

Ця «пристосованість» механізму надихається явищами, які з'явилися в екосистемі BitTorrent протягом 2007–2009 років. Було опубліковано початковий план Брема Коена оптимістичного механізму «синиця за око» у BitTorrent. У цьому документі детально описується дизайн BitTyrant,

спеціально створеного клієнта BitTorrent, який зловживав недоліками в початковій формулюванні оптимістичного принципу «око за око», щоб дозволити користувачам викачувати більше даних, ніж вони передають іншим учасникам мережі. На цьому етапі в мережі BitTorrent були розіграні дві активні ігри — оптимістична «синиця за око» та BitTyrant. Після цього спільнота Bittorrent почала створювати модифікованих оптимістичних агентів, щоб зірвати діяльність BitTyrant із безкоштовного завантаження, що призвело до того, що три різні гри запускалися одночасно в багатьох різних групах BitTorrent. Згодом користувачі BitTorrent, як правило, перейшли до клієнтів, стійких до BitTyrant, і метагра була адаптована, щоб знову заохочувати просоціальну поведінку.

Подібні приклади проектування адаптивних механізмів вже з'явилися в екосистемі Arweave. Наприклад, у березні 2021 року частина майнерів внесла зміни до своїх вузлів, щоб боротися з іншою частиною майнерів, використовуючи програмне забезпечення для об’єднання, яке вони сприймали як хижацьке.

Майнінг в мережі Arweave складається з кількох етапів:

1. Під час генерації блоку для вузлів для зчитування випадкових даних вибирається підпростір пошуку по всій мережі (10% мережі).
2. Метадані блоку-кандидата та випадкове число (nonce) хешуються разом за допомогою RandomX.
3. Вихід цього хешу — дуже велике ціле число — зменшується до діапазону, заданого розміром простору пошуку. Це число додається до основи простору пошуку (обчислюється на кроці 1) і використовується як «пробний байт» — точка в наборі даних, до якої майнер повинен підтвердити, що вони можуть отримати доступ, щоб продовжити процес видобутку.

4. Тепер майнер намагається прочитати фрагмент, що містить байт виклику, з диска в пам'ять. Якщо майнер не зберігає копію даного фрагмента, він перезапускає процес майнінгу з другого етапу.
5. Потім майнер зчитує або відтворює, відповідно до необхідності, доказ включення даного байта у відповідне місце в наборі даних (доказ Меркла).
6. Метадані блоку, доказ Меркла, фрагмент і одноразовий номер потім хешуються разом за допомогою RandomX.
7. Вихід цього хешу інтерпретується як число, а потім перевіряється відповідно до поточного рівня складності в мережі. Усі, крім однієї спроби на цикл виробництва блоку в мережі, зазнають невдачі в цій перевірці, і ці майнери почнуть знову з другого етапу.
8. Коли вузол створив доказ для дійсного блоку-кандидата, підтвердження SPoRA та заголовок блоку поширюються на решту мережі.

Proof of Access Consensus

Дизайн Areweave також означає, що спосіб перевірки точності транзакцій відрізняється від більшості криптовалют.

Там, де біткойн просить комп’ютери у своїй мережі змагатися за вирішення математичної головоломки — процес, який називається підтвердженням роботи — Arweave використовує інший механізм, який називається «доказом доступу».

Коротше кажучи, Arweave вимагає від кожного комп’ютера, який бере участь у мережі, перевіряти, чи новий пакет транзакцій також містить випадково вибраний маркер із попередньої групи.

Якщо цей маркер присутній, нові транзакції можна додати до мережі. Комп’ютер, який додає новий пакет, отримує винагороду у вигляді криптовалюти AR.

Доказ доступу допомагає впевнитися, що комп’ютери в мережі Arweave можуть підтвердити, що всі нові транзакції точні і що старі транзакції не були підроблені.

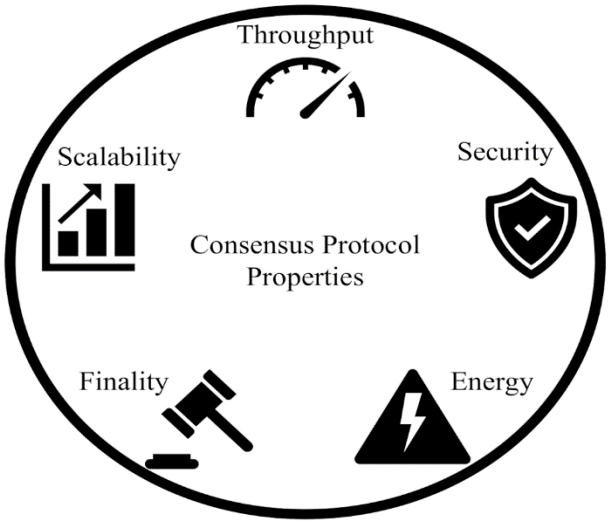


Рисунок 1.11 – Основні вимоги до протоколу консенсусу [20]

Модерація вмісту

Іншою особливістю мережі Arweave є можливість для будь-кого, хто використовує програмне забезпечення, вибирати тип даних, які він бажає зберігати. Цей процес відомий як модерація вмісту в Arweave.

Простіше кажучи, комп’ютери в мережі можуть вирішувати, які типи вмісту вони хочуть розміщувати.

Можливо, вони хочуть розміщувати лише аудіофайли, а не зображення. Коли новий вміст буде завантажено в мережу, Arweave запитає кожен комп’ютер, чи прийме він дані.

Проте є стимули для більш інтенсивного зберігання даних, оскільки користувачі, які це зроблять, будуть винагороджені більшим відсотком комісії за транзакції.

ВИСНОВОК ДО РОЗДІЛУ 1

В першому розділі розглянуті типи систем, їх аналіз та характеристики.

Розглянуті типи систем збереження даних та їх особливості. Досліджено сучасні повноційні системи збереження даних, зроблено висновки щодо них та виділено їх сильні сторони та проблематику.

З дослідження можна зробити висновой, що на даному етапі розвитку технологій децентралізовані системи збереження даних лише починають розвиватися та набувають популярності. У прагені світу бути рівноправними та вільними учасниками мережі інтернет, набуло популярність використання децентралізованих систем та зародивсь новий напрям Web3, котрий прийшов на заміну відомої технології Web2 – централізованих веб систем з інтерактивним інтерфейсом.

Так, як цей напрям існує менше 5 років, він є недосконалим та йому лише належить розвиватися та вдосконалюватися.

Таким чином, актуальною є задача розробки системи захищеного розподіленого зберігання даних, адже навіть у проаналізованих вище системах є значні недоліки, хоча вони і є лідируючими по популярності та функціоналу. Наприклад, в жодній із систем не реалізовано приватного сховища даних та захищеного доступу до них на основі пар ключів.

В деяких з них є проблема того, що користувач не має змоги сам вирішувати, котрим вузлам він довіряє та не має змоги обрати власноруч вузли для завантаження власних файлів. Така система, як Pinata все ж не має справжньої децентралізації та поклажається на деякі централізовані рішення для її контролю та розвитку.

Саме тому, було вирішено розробити систему, котра поєднує деякі виправдані практики існуючих систем та виршує ті проблеми, котрі в них були виявлені під час їх аналізу.

РОЗДІЛ 2
ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ

2.1 JavaScript та декларування типів за допомогою TypeScript(d.ts)

Для реалізації програмного застосунку було вирішено використовувати мову програмування JavaScript в поєднанні з можливостями декларування та опису типів в TypeScript.

Декларування типів TypeScript використовувалося для спрощення розуміння кодовою бази та документації коду. Аналогом є технологія JSdoc, котра також дозволяє описувати типи, як коментарі, в динамічній мові JavaScript.

Перевагою декларування типів Typescript, замість використання самого Typescript, або ж JSdoc – є те, що ми можемо так само зручно використовувати всі можливості JavaScript без будь-яких обмежень та розділяти логіку застосунку від документації, декларування або опису типів.

Як відомо, не зважаючи, на те, що прийнято писати модульний та абстрактний код, котрий не має зачеплень між компонентами та модулями та взаємодіє виключно на контрактах або інтерфейсах, існує проблема зачеплення типів між модулями.

Використовуючи типи окремо на рівні опису компонентів, ми можемо уникати зачеплень коду на типах і робити наші модулі абстрагованими від інших компонентів або ж типів.

Тепер детальніше про мову JavaScript та TypeScript.

JavaScript – інтерпретована та JIT-компільована, мультипарадигменна, динамічна, слаботипізована, скриптова мова програмування високого рівня, котра є реалізацією стандарта ECMAScript (ECMA-262). Підтримує об’єктно-орієнтований(прототипно-орієнтований), імперативний і функціональний стилі та як показує практика має можливості реалізації підтримки декларативного стилю програмування.[22]

Виконується переважно в спеціальних середовищах виконання, котрі називаються двигунами. Найпопулярнішими двигунами JavaScript є V8 та SpiderMonkey.

SpiderMonkey – найперший двигун JavaScript, котрий створений Бренданом Айхом в Netscape.

V8 – найновіший, найсучасніший та найпопулярніший двигун виконання JavaScript, котрий розроблений компанією Google та використовується в більшості браузерів та платформі Node.Js.

Особливості мови:

- Легка інтерпретована мова (JavaScript не є компільованою мовою, тому він не перетворюється на байт-код заздалегідь. Однак він дотримується парадигми під назвою Just-In-Time (JIT) Compilation. Це означає, що він перетворюється на байт-код, коли він збирається запуститися. Це дозволяє JS бути легким. Навіть менш потужні пристрої можуть запускати JavaScript)
- Незалежний від платформи (JavaScript не залежить від платформи; він може працювати на будь-якому комп'ютері, незалежно від використовуваної операційної системи. JavaScript вбудований у Netscape 2.0 і новіших версій. Оскільки інтерпретатор JavaScript є частиною Netscape, він працює на платформах, які підтримують Netscape, а це означає, що фрагмент коду Javascript дасть однаковий вихід з однаковими налаштуваннями на всіх платформах.)
- Асинхронність (JavaScript підтримує асинхронну поведінку, котра реалізовується за допомогою EventLoop. Є дуже багато інструментів та підходів для роботи з асинхронністю. Наприклад, callbacks, promises, async-await, events, streams, rxjs, …)
- З'явилась можливість запуску в декількох потоках(Можливість запустити декілька потоків виконання коду за допомогою Worker Threads, котрі ізольовані власним контекстом в пам'яті, але мають

вбудовану можливість роботи зі спільною областю пам’яті та каналів спілкування подібними до pub\sub підходу).

- Прототипно-орієнтована (використовує прототипну модель успадкування замість загальновідомого успадкування класів. Це означає, що замість того, щоб створювати класи та виводити з них об’єкти, ми визначаємо прототип Objects і використовуємо цей прототип об’єкта для створення інших об’єктів того ж типу. Ця поведінка подібна до метода фабрики Object, тому такий шаблон проектування зазвичай можна побачити в коді JavaScript.)
- Мультипарадигменість(підтримує більшість парадигм програмування, включаючи навіть реактивне та метапрограмування)

TypeScript - це наднабір мови JavaScript, який має єдиний компілятор з відкритим кодом і розробляється в основному одним постачальником: Microsoft. Мета TypeScript полягає в тому, щоб допомогти виявити помилки на ранній стадії за допомогою системи типів і зробити розробку JavaScript більш ефективною. [23]

TypeScript досягає своїх цілей трьома способами:

- **Підтримка сучасних функцій** JavaScript. Мова JavaScript (а не середовище виконання) стандартизована за допомогою стандартів ECMAScript. Не всі браузери та середовища виконання JavaScript підтримують усі функції всіх стандартів ECMAScript. TypeScript дозволяє використовувати багато найновіших функцій ECMAScript і перекладає їх на старіші цільові елементи ECMAScript за вашим вибором. Це означає, що ви можете безпечно використовувати нові функції, такі як модулі, лямбда-функції, класи, оператор поширення та деструктуризацію, залишаючись зворотною сумісністю зі старими браузерами та середовищами виконання JavaScript.
- **Розширена система** типів. Підтримка типів не є частиною стандарту ECMAScript і, швидше за все, ніколи не буде пов’язана з інтерпретованою

природою замість компільованого JavaScript. Система типів TypeScript неймовірно багата і включає в себе: інтерфейси, перерахування, гібридні типи, генерики, типи об'єднання/перетину, модифікатори доступу та багато іншого. Система типів Typescript є на одному рівні з більшістю інших типізованих мов і в деяких випадках, можливо, є більш потужною.

- **Підтримка інструментів розробника** – компілятор TypeScript може працювати як фоновий процес, щоб підтримувати як поетапну компіляцію, так і інтеграцію IDE, щоб ви могли легше орієнтуватися, виявляти проблеми, перевіряти можливості та реорганізувати свою кодову базу.

2.2 Платформа Node.Js

Node.Js - це серверна платформа, побудована на платформі JavaScript Engine (V8 Engine) Google Chrome. Node.js був розроблений Райаном Далом у 2009 році. Це кросплатформне середовище виконання з відкритим вихідним кодом для розробки серверних і мережевих додатків. Програми Node.js написані на JavaScript і можуть запускатися в середовищі виконання Node.js в OS X, Microsoft Windows і Linux.[24]

Node.js також надає багату бібліотеку різних модулів JavaScript, що значною мірою спрощує розробку веб-додатків за допомогою Node.js.

Особливості:

- **Асинхронний і керований подіями** — усі API бібліотеки Node.js асинхронні, тобто не блокують. По суті, це означає, що сервер на основі Node.js ніколи не чекає, поки API поверне дані. Сервер переходить до наступного API після його виклику, а механізм сповіщень про події Node.js допомагає серверу отримати відповідь від попереднього виклику API.
- **Дуже швидкий**. Бібліотека Node.js, побудована на двигуні JavaScript V8 від Google Chrome, дуже швидко виконує код.

- **Однопотоковий, але високомасштабований**— Node.js використовує однопотокову модель з циклом подій. Механізм подій допомагає серверу реагувати неблокуючим способом і робить сервер дуже масштабованим на відміну від традиційних серверів, які створюють обмежені потоки для обробки запитів. Node.js використовує одну потокову програму, і та сама програма може надавати обслуговування набагато більшої кількості запитів, ніж традиційні сервери, такі як Apache HTTP Server.
- **Без буферизації** — програми Node.js ніколи не буферизують будь-які дані. Ці програми просто виводять дані фрагментами.

Має дуже багате середовище та інструментарій, як вбудований, так і сторонні бібліотеки.

За допомогою даної платформи, мова JavaScript стала мовою загального призначення, котра знайшла своє застосування майже у всіх сферах.

Хоча JavaScript не може похвалитись високою продуктивністю в складних обчислювальних операціях, Node.js має можливість підключення та використання сторонніх модулів, котрі написані на більш продуктивних мовах та були попередньо скомпільовані у WASM, або як аддон до платформи.

2.3 Система управління реляційними базами даних MySQL

MySQL - система керування базами даних SQL з відкритим вихідним кодом, розроблена, поширюється та підтримується Oracle Corporation.[25]

Внутрішня будова та портативність:

- Написано на C і C++.
- Перевірено за допомогою широкого спектру різних компіляторів.
- Працює на багатьох різних платформах.
- Для переносимості налаштовано за допомогою CMake.
- Використовує багаторівневий дизайн сервера з незалежними модулями.
- Розроблено для повної багатопотокової роботи з використанням потоків ядра, щоб легко використовувати декілька ЦП, якщо вони доступні.
- Забезпечує транзакційні та нетранзакційні механізми зберігання даних.

- Використовує дуже швидкі дискові таблиці B-дерева (MyISAM) зі стисненням індексів.
- Створено для відносно легкого додавання інших механізмів зберігання даних. Це корисно, якщо ви хочете надати інтерфейс SQL для внутрішньої бази даних.
- Використовує дуже швидку систему розподілу пам'яті на основі потоків.
- Виконує дуже швидкі об’єднання за допомогою оптимізованого з’єднання вкладеного циклу.
- Реалізує хеш-таблиці в пам’яті, які використовуються як тимчасові таблиці.
- Реалізує функції SQL, використовуючи високооптимізовану бібліотеку класів, яка має бути максимально швидкою. Зазвичай після ініціалізації запиту виділення пам’яті взагалі не відбувається.
- Надає сервер як окрему програму для використання в мережевому середовищі клієнт/сервер.

Типи даних

Багато типів даних: цілі числа зі знаком/без знаку довжиною 1, 2, 3, 4 і 8 байтів: FLOAT, DOUBLE, CHAR, VARCHAR, BINARY, VARBINARY, TEXT, BLOB, DATE, TIME, DATETIME, TIMESTAMP, YEAR, SET, ENUM, і просторові типи OpenGIS.

Підтримувані функції:

- Повна підтримка операторів і функцій у SELECTсписку та WHEREпункті запитів.
- Повна підтримка SQL GROUP BYі ORDER BYположень. Підтримка групових функцій (COUNT(), AVG(), STD(), SUM(), MAX(), MIN(), і GROUP_CONCAT()).
- Підтримка стандартного синтаксису SQL і ODBC і з ним LEFT OUTER JOIN. RIGHT OUTER JOIN

- Підтримка псевдонімів у таблицях і стовпцях, як того вимагає стандартний SQL.
- Підтримка DELETE, INSERT, REPLACE, і UPDATEдля повернення кількості рядків, які були змінені (уражені), або для повернення кількості відповідних рядків замість цього, установивши прапор під час підключення до сервера.
- Підтримка специфічних для MySQL SHOW операторів, які отримують інформацію про бази даних, системи зберігання даних, таблиці та індекси. Підтримка INFORMATION_SCHEMAбази даних, реалізована за стандартом SQL.
- Оператор EXPLAIN, який показує, як оптимізатор вирішує запит.
- Незалежність імен функцій від назв таблиць або стовпців. Наприклад, ABSдійсна назва стовпця. Єдине обмеження полягає в тому, що для виклику функції не допускаються пробіли між іменем функції.
- Ви можете посилатися на таблиці з різних баз даних в одному операторі.[25]

Безпека

- Система привілеїв і паролів, яка є дуже гнучкою та безпечною, а також забезпечує перевірку на хості.
- Захист паролем шляхом шифрування всього трафіку паролів при підключенні до сервера.

Масштабованість та обмеження:

- Підтримка великих баз даних. Ми використовуємо MySQL Server з базами даних, які містять 50 мільйонів записів. Ми також знаємо користувачів, які використовують MySQL Server з 200 000 таблиць і близько 5 000 000 000 рядків.
- Підтримка до 64 індексів на таблицю. Кожен індекс може складатися з 1 до 16 стовпців або частин стовпців. Максимальна ширина індексу

для InnoDB таблиць становить 767 байт або 3072 байти. Максимальна ширина індексу для MyISAM таблиць становить 1000 байт. Індекс може використовувати префікс стовпця для типів CHAR, VARCHAR, BLOB, або стовпців TEXT.[25]

2.4 Транспорт та мережа

Застосунок буде використовувати 2 типи інтерфейсу взаємодії:

- P2P (Взаємодія між учасниками мережі(нодами))
- client-server (Взаємодія користувача, клієнта з нодою задля доступу до мережі та її функцій. Нода буде точкою входу в мережу, як це зараз відбувається майже в усіх децентралізованих мережах).

Для реалізації даного підходу буде використовуватись 2 протоколи прикладного рівня: HTTP(S) та WebSocket.

HTTP(S) - це протокол прикладного рівня для передачі гіпермедійних документів, таких як HTML. Він був розроблений для зв'язку між веб-браузерами та веб-серверами, але його також можна використовувати для інших цілей. HTTP дотримується класичної моделі клієнт-сервер, коли клієнт відкриває з'єднання, щоб зробити запит, а потім чекає, поки не отримає відповідь. HTTP є протоколом без стана, що означає, що сервер не зберігає жодних даних (стан) між двома запитами.[26]

Працює поверх TCP, який є протоколом, орієнтованим на з'єднання, він гарантує доставку пакетів даних за допомогою методів тристороннього рукостискання та повторно передає втрачені пакети.

HTTP може працювати поверх будь-якого надійного протоколу, орієнтованого на підключення, такого як TCP, SCTP. Коли клієнт надсилає запит HTTP на сервер, TCP-з'єднання відкривається між клієнтом і сервером, і після отримання відповіді TCP-з'єднання припиняється, кожен HTTP-запит відкриває окреме TCP-з'єднання до сервера, наприклад, якщо клієнт надсилає Після 10 запитів до сервера буде відкрито 10 окремих TCP-з'єднань. і закритися після отримання відповіді/резервного рішення.

Інформація про повідомлення HTTP, закодована в ASCII, кожне повідомлення запиту HTTP складається з версії протоколу HTTP (HTTP/1.1, HTTP/2), методів HTTP (GET/POST тощо), заголовків HTTP (тип вмісту, довжини вмісту), інформації про хост тощо і тіло, яке містить фактичне повідомлення, яке передається на сервер. Розмір заголовків HTTP варіюється від 200 байт до 2 КБ, звичайний розмір заголовка HTTP становить 700-800 байт. Коли веб-додаток використовує більше файлів cookie та інших інструментів на стороні клієнта, які розширюють можливості зберігання агента, це зменшує корисне навантаження заголовка HTTP.[26]

WebSocket – двонаправлений, повнодуплексний протокол, який використовується в сценарії спілкування клієнт-сервер. Це постійне з'єднання між клієнтом і сервером. WebSockets забезпечують двонаправлений повнодуплексний канал зв'язку, який працює через HTTP через єдине з'єднання TCP/IP. По суті, протокол WebSocket полегшує передачу повідомлень між клієнтом і сервером. У цій статті наведено введення в протокол WebSocket, включаючи проблему, яку вирішують WebSockets, а також огляд того, як WebSockets описуються на рівні протоколу.[27]

WebSockets починається як стандартний запит і відповідь HTTP. У ланцюжку відповідей на запит клієнт просить відкрити з’єднання WebSocket, а сервер відповідає (якщо може). Якщо це початкове рукостискання пройшло успішно, клієнт і сервер погодилися використовувати існуюче з’єднання TCP/IP, яке було встановлено для запиту HTTP, як з’єднання WebSocket. Дані тепер можуть передаватися через це з’єднання за допомогою базового протоколу повідомлень у фреймі. Як тільки обидві сторони визнають, що з’єднання WebSocket має бути закрито, з’єднання TCP розривається.

Ідея WebSockets виникла через обмеження технології на основі HTTP. За допомогою HTTP клієнт запитує ресурс, а сервер відповідає запитаними даними. HTTP є суворо односпрямованим протоколом — будь-які дані, надіслані від сервера до клієнта, повинні бути спочатку запитані

клієнтом. Довге опитування традиційно служило обхідним шляхом для цього обмеження. При тривалому опитуванні клієнт робить HTTP-запит із тривалим періодом очікування, і сервер використовує цей тривалий час очікування, щоб передати дані клієнту. Довгий опитування працює, але має недолік — ресурси на сервері прив’язані на всьому протязі довгого опитування, навіть якщо дані для надсилання недоступні.

WebSockets, з іншого боку, дозволяють надсилати дані на основі повідомлень, подібно до UDP, але з надійністю TCP. WebSocket використовує HTTP як початковий транспортний механізм, але підтримує з'єднання TCP після отримання відповіді HTTP, щоб його можна було використовувати для надсилання повідомлень між клієнтом і сервером. WebSockets дозволяють нам створювати програми в режимі реального часу без використання тривалого опитування.[27]

2.5 Безпека та конфіденційність в розподіленій мережі

Майже кожна система повинна вміти ідентифікувати користувача за певними даними та виходячи з того, хто до неї звертається виконувати чи не виконувати певні запити та процедури. Багато систем потребують саме авторизації користувачів задля того, щоб оцінити права та можливості того, хто до неї звертається та відповідно до того робити певні висновки та захищати особисті дані своїх клієнтів. У централізованих системах це зазвичай робиться за допомогою секретного паролю, котрий у захешованому вигляді зберігається в базі даних системи. Для авторизації користувач повинен передати даний пароль і відповідно авторизуватися у разі його дійсності. В централізованих системах ми завжди довіряємо провайдеру системи, адже саме він відповідає за наші дані.

Але в децентралізованих системах все інакше. Кожного разу ми можемо звертатись до різних вузлів і не маємо гарантії, кому передаємо свій секретний пароль.

Відповідно не знаємо, що далі може з ним відбуватись і чи добрі наміри у власника вузла щодо нас. Тому у децентралізованих системах задля авторизації завжди використовують пари публічного та приватного ключів. І передаючи на віддалений вузол свій публічний ключ ми можемо бути впевненими, що наш секретний ключ у надійному місці і ми його нікуди не передаємо, а публічний ключ нічого не вартий в плані власної безпеки та конфіденційності.

Існує такий напрям, як асиметрична криптографія, котра дозволяє нам авторизувати користувача, не отримуючи від нього його секретів.

Асиметрична криптографія масштабована для використання у дуже великих і постійно розширюваних середовищах, де часто обмінюються даними між різними комунікаційними партнерами.[28]

В проектованій системі і будемо використовувати дані методи задля забезпечення безбеки користувача та захищенності його особистих даних.

2.6 Інструменти розробки та розгортання

VsCode - безкоштовний текстовий редактор з відкритим вихідним кодом від Microsoft. VS Code доступний для Windows, Linux і macOS. Хоча редактор є відносно легким, він містить деякі потужні функції, які зробили VS Code одним із найпопулярніших інструментів середовища розробки останнім часом.

VS Code підтримує широкий спектр мов програмування від Java, C++ і Python до CSS, Go і Dockerfile. Крім того, VS Code дозволяє вам додавати і навіть створювати нові розширення, включаючи лінтери коду, налагоджувачі та підтримку хмарної та веб-розробки.[29]

DataGrip - це потужна IDE для роботи з базами даних і SQL. Він розроблений, щоб допомогти вам писати та зберігати запити; робота із збереженими процедурами; перегляд, імпорт та експорт даних; і виконувати потрібну роботу з базами даних в одному інструменті. Він підтримує PostgreSQL, MySQL, SQL Server, Oracle, MongoDB та багато інших баз даних і джерел даних DDL. DataGrip також допомагає вам швидше кодувати

за допомогою розумного кодування, аналізу «на льоту» та швидких виправлень.[30]

Docker - це відкрита платформа для розробки, доставки та запуску програм. Docker дає змогу відокремити ваші програми від інфраструктури, щоб ви могли швидко постачати програмне забезпечення. За допомогою Docker ви можете керувати своєю інфраструктурою так само, як і своїми програмами. Використовуючи переваги методологій Docker для швидкої доставки, тестування та розгортання коду, ви можете значно зменшити затримку між написанням коду та його запуском у виробництві.

Docker надає можливість пакувати та запускати програму в слабо ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють запускати багато контейнерів одночасно на даному хості. Контейнери легкі та містять все необхідне для запуска програми, тому вам не потрібно покладатися на те, що в даний момент встановлено на хості. Ви можете легко ділитися контейнерами під час роботи, і будьте впевнені, що всі, з ким ви ділитеся, отримають той самий контейнер, який працює так само.[31]

Docker надає інструменти та платформу для керування життєвим циклом ваших контейнерів:

- Розробляйте свою програму та її допоміжні компоненти за допомогою контейнерів.
- Контейнер стає блоком для розповсюдження та тестування вашої програми.
- Коли ви будете готові, розгорніть свою програму у виробничому середовищі, як контейнер або організовану службу. Це працює однаково, незалежно від того, чи є ваше виробниче середовище локальним центром обробки даних, хмарним провайдером або гібридом обох.

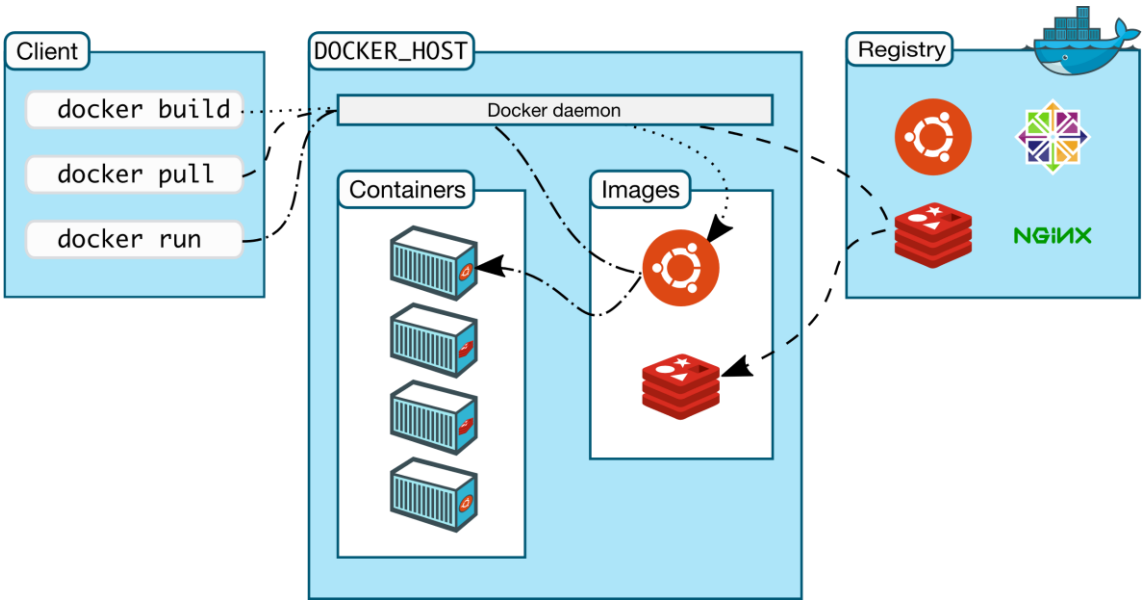


Рисунок 2.1 – Взаємодія компонентів системи контейнеризації Docker [32]

Docker використовує архітектуру клієнт-сервер. *Клієнт* Docker спілкується з *демоном* Docker, який виконує важку роботу зі створення, запуску та розповсюдження ваших контейнерів Docker. Клієнт і демон Docker *можуть* працювати в одній системі, або ви можете підключити клієнта Docker до віддаленого демона Docker. Клієнт Docker і демон спілкуються за допомогою REST API, через сокети UNIX або мережевий інтерфейс. Іншим клієнтом Docker є Docker Compose, який дозволяє працювати з додатками, що складаються з набору контейнерів.

Об’єкти Docker

Коли ви використовуєте Docker, ви створюєте та використовуєте зображення, контейнери, мережі, томи, плагіни та інші об’єкти. У цьому розділі наведено короткий огляд деяких з цих об’єктів.

Зображення — це шаблон лише *для* читання з інструкціями щодо створення контейнера Docker. Часто зображення *засноване на* іншому зображенні з деякими додатковими налаштуваннями. Наприклад, ви можете створити образ, який базується на ubuntu зображенні, але встановлює веб-

сервер Apache і вашу програму, а також деталі конфігурації, необхідні для запуска вашої програми.

Ви можете створювати власні зображення або використовувати лише ті, які створені іншими та опубліковані в реєстрі. Щоб створити власний образ, ви створюєте *Dockerfile* з простим синтаксисом для визначення кроків, необхідних для створення образу та його запуску. Кожна інструкція у файлі Docker створює шар у зображенні. Коли ви змінюєте файл Docker і перебудовуєте образ, перебудовуються лише ті шари, які були змінені. Це частина того, що робить зображення такими легкими, невеликими та швидкими в порівнянні з іншими технологіями віртуалізації.

Контейнер — це екземпляр зображення, який можна виконувати. Ви можете створювати, запускати, зупиняти, переміщувати або видаляти контейнер за допомогою Docker API або CLI. Ви можете підключити контейнер до однієї або кількох мереж, приєднати до нього сховище або навіть створити нове зображення на основі його поточного стану.

За замовчуванням контейнер відносно добре ізольований від інших контейнерів і його хост-машини. Ви можете контролювати, наскільки ізольовані мережа контейнера, сховище чи інші базові підсистеми від інших контейнерів або від хост-машини.

Контейнер визначається його зображенням, а також будь-якими параметрами конфігурації, які ви надаєте йому під час створення або запуску. Коли контейнер видаляють, усі зміни його стану, які не зберігаються в постійному сховищі, зникають.

2.7 IPFS протокол

IPFS — це однорангова (p2p) мережа зберігання даних. Контент доступний через однорангових партнерів, розташованих у будь-якій точці світу, які можуть передавати інформацію, зберігати її або робити те й інше. IPFS знає, як знайти те, що ви просите, використовуючи адресу вмісту, а не його місцезнаходження.[7]

Існує три фундаментальних принципи розуміння IPFS:

1. Унікальна ідентифікація за допомогою адресації вмісту
2. Посилання вмісту за допомогою орієнтованих ациклічних графіків (DAG)
3. Виявлення вмісту за допомогою розподілених хеш-таблиць (DHT)

Ці три принципи ґрунтуються один на одному, щоб створити екосистему IPFS. Почнемо з *адресації вмісту* та унікальної ідентифікації вмісту.

Адресування вмісту

IPFS використовує *адресацію вмісту* , щоб ідентифікувати вміст за тим, що в ньому, а не за місцем його розташування. Ви вже постійно шукаєте предмет за змістом. Наприклад, коли ви шукаєте книгу в бібліотеці, ви часто просите її за назвою; це адресація вмісту, тому що ви запитуєте, *що* це таке. Якщо ви використовуєте адресацію місцеположення, щоб знайти цю книгу, ви б запитали її за *місцем* її знаходження: «Я хочу книгу, яка знаходиться на другому поверсі, перша стопка, третя полиця знизу, чотири книги зліва». Якби хтось переніс цю книгу, вам не пощастило!

Ця проблема існує в Інтернеті та на вашому комп’ютері! Зараз вміст можна знайти за місцем розташування.

IPFS та багато інших розподілених систем використовують переваги структури даних, що називається орієнтованими ациклічними графамиабо DAG. Зокрема, вони використовують *DAG Merkle*, де кожен вузол має унікальний ідентифікатор, який є хешем вмісту вузла. Це посилається на концепцію *CID*, яку ми розглянули в попередньому розділі.

Іншими словами: ідентифікація об’єкта даних (наприклад, вузла Merkle DAG) за значенням його хешу *є адресацією вмісту*.

IPFS використовує Merkle DAG, оптимізовану для представлення каталогів і файлів, але ви можете структурувати Merkle DAG багатьма різними способами. Наприклад, Git використовує Merkle DAG, який містить багато версій вашого репо.

Щоб створити представлення Merkle DAG вашого вмісту, IPFS часто спочатку розбиває його на *блоки*. Розбиття його на блоки означає, що різні частини файлу можуть надходити з різних джерел і швидко перевірятися. (Якщо ви коли-небудь користувалися BitTorrent, ви, можливо, помічали, що коли ви завантажуєте файл, BitTorrent може отримати його з кількох однорангових пристроїв одночасно).

Ще одна корисна функція Merkle DAG і розбиття вмісту на блоки полягає в тому, що якщо у вас є два подібних файли, вони можуть спільно використовувати частини Merkle DAG, тобто частини різних DAG Merkle можуть посилатися на ту саму підмножину даних. Наприклад, якщо ви оновлюєте веб-сайт, лише оновлені файли отримують нові адреси вмісту. Ваша стара і нова версія можуть посилатися на ті самі блоки для всього іншого. Це може зробити передачу версій великих наборів даних (таких як дослідження геномії або дані про погоду) більш ефективним, оскільки вам потрібно передавати лише нові або змінені частини, замість того, щоб щоразу створювати абсолютно нові файли.

Отже, підсумовуючи, IPFS дозволяє надавати CID вмісту та пов’язувати цей вміст разом у Merkle DAG. Тепер перейдемо до останнього: як ви знаходите та переміщуєте вміст.

Розподілені хеш-таблиці(DHT)

Щоб визначити, які однорангові вузли розміщують вміст, який ви шукаєте, IPFS використовує розподілену хеш-таблицю або DHT.

Хеш-таблиця — це база даних ключів до значень. Розподілена хеш-таблиця — це та, де таблиця розділена на всіх однорангових у розподіленій мережі.[8]

Після того, як ви дізнаєтеся, де знаходиться ваш вміст (або, точніше, які однорангові пристрої зберігають кожен із блоків, які складають вміст, який ви шукаєте), ви знову використовуєте DHT, щоб знайти поточне розташування цих однорангових вузлів (*маршрутизація*).

Ви знайшли свій вміст і знайшли поточне розташування цього вмісту. Тепер вам потрібно підключитися до цього вмісту та отримати його (*обмін*).

Короткий підсумок

Проект IPFS спрямований на розвиток інфраструктури Інтернету та Інтернету, використовуючи багато речей, які ми навчилися в успішних системах, таких як Git, BitTorrent , Kademlia , Bitcoin та багато-багато інших.

IPFS (InterPlanetary File System) — це протокол розповсюдження гіпермедіа, який адресується вмістом та ідентифікаторами. Він дає змогу створювати повністю розподілені додатки, і при цьому прагне зробити Інтернет швидшим, безпечнішим і відкритим.

IPFS - це розподілена файлова система, яка прагне з'єднати всі комп'ютерні пристрої з однією системою файлів. У певному сенсі це схоже на початкові цілі Інтернету, але IPFS насправді більше схожий на один рій BitTorrent, що обмінюється об’єктами Git. Докладніше про його походження можна прочитати в статті IPFS – адресована, версійна, файлова система P2P.

IPFS стає новою великою підсистемою Інтернету. Якщо побудувати правильно, він міг би доповнити або замінити HTTP. Він міг би доповнити або замінити ще більше. Давайте по пунктам розберемося, як.

IPFS - це протокол:

- Визначає файлову систему, адресовану вмістом
- Координує доставку контенту
- Поєднує Kademlia + BitTorrent + Git
- IPFS - це файлова система
- Має каталоги та файли
- Є файловою системою для монтування

IPFS - це веб:

- Можна використовувати для перегляда документів, як у звичайному Інтернеті
- Файли доступні через HTTP за адресоюhttps://ipfs.io/<path>
- Браузери та розширення можуть навчитися використовувати схеми ipfs://URL або dweb:/ipfs/URI безпосередньо
- Вміст із хеш-адресою гарантує автентичність

IPFS є модульним:

- Рівень підключення через будь-який мережевий протокол
- Шар маршрутизації
- Використовує рівень маршрутизації DHT
- Використовує службу імен на основі шляху
- Використовує блочний обмін, натхненний BitTorrent

IPFS використовує криптовалюту:

- Адресація криптографічного хеш-контенту
- Дедуплікація на рівні блоку
- Цілісність файлів плюс управління версіями
- Шифрування на рівні файлової системи плюс підтримка підпису

IPFS - це p2p:

- По всьому світу однорангова передача файлів
- Повністю децентралізована архітектура
- Немає центральної точки відмови

IPFS - це CDN:

- Додайте файл до файлової системи локально, і тепер він стане доступним у всьому світі
- Зручний для кешування (найменування хеш вмісту)
- Розподіл пропускної здатності на основі BitTorrent

IPFS має службу імен:

- IPNS, система імен, натхненна SFS

- Глобальний простір імен на основі PKI
- Він служить для побудови ланцюгів довіри
- Він сумісний з іншими НС
- Може зіставити DNS, .onion, .bit тощо з IPNS

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі були розглянуті інструменти та технології, котрі будуть використовуватись для розробки захищеної розподіленої системи збереження даних.

Детально було розглянуто платформу Node.JS та мову програмування JavaScript.

Провівши аналіз даних технологій було вирішено використовувати саме їх, адже їх переваги асинхронної обробки операцій та швидкого неблокуючого входу\виходу ідеально підходять для вирішення проблем даної системи.

В розділі також широко представлено та описано систему керування реляційними базами даних MySQL, котру планується використовувати, як одне з основних сховищ даних. Описані основні операції та переваги даного рішення.

Було порушено проблему та описано мережеві протоколи HTTP та WebSocket, за допомогою яких будуть будуватись зв'язки в розподіленій мережі та виконуватись взаємодія, як між вузлами, так і між вузлом та користувачем. Наведено переваги та технічні аспекти даних протоколів для більш детального розуміння роботи мережі.

Коротко розглянуто технології за допомогою, яких планується розробляти основний клієнтський інтерфейс для взаємодії з віддаленими вузлами, а саме мову розмітки HTML, мову табличних стилів CSS та бібліотеку React.js.

Детально розглянуто технологію контейнеризації Docker та її будову. Описано основні її частини та компоненти, їх взаємодію. Виокремлено проблеми, котрі вона вирішує.

На етапі розробки даного розділу ми проаналізували безліч рішень та технологій та змогли вибрати саме ті, котрі найкраще та найручніше вирішують поставлену нами проблему та допоможуть швидко та якісно розробити проектовану розподілену захищену систему збереження даних.

РОЗДІЛ 3.
РОЗРОБКА ТА ПРОЕКТУВАННЯ СИСТЕМИ

3.1 Вимоги до розроблюваної децентралізованої файлової системи.

Майже кожна децентралізована система повинна обробляти великі потоки входів та виходів, мати можливість паралельно або ж асинхронно(не блокуючись) обробляти запити та завантажувати дані. В нашій системі основне навантаження буде саме на операції входу\виходу та сховище.

Саме тому і було вирішено використовувати платформу Node.JS для реалізації системи.

Необхідно мати можливість запускати вузли в будь-якому середовищі та на будь-якій операційній системі. По перше це досягається, також, за допомогою кросплатформеної технології Node.JS та додатково з використанням технології контейнеризації Docker.

Із основних функціональних вимог, в першу чергу це:

- швидкий асинхронний пошук даних по всій мережі
- гарантування уникнення дублювання даних
- гарантування швидкого доступу до даних
- можливість завантаження приватних файлів та можливість приховувати свої файли під час пошуку іншими користувачами
- авторизація власників даних за допомогою асиметричних підписів
- можливість прикріплення метаданих до файлів, що може дозволити реалізувати деревовидну структуру. Самі файли зберігаються на вузлі в один рівень, прозоро, але за допомогою метаданих можна вказувати до котрого каталогу належать дані. Також, метадані до файлу дозволяють вирішити проблему, котра зараз існує при створенні NFT, коли необхідно завантажувати 2 сутності файлі у децентралізоване сховище: файл з медіа та окремо файл з метаданими. У нашій системі це можна зробити одним запитом в одній сутності.

Перспективи в наступних версіях:

- монетизація сховища за допомогою сторонніх смарт-контрактів та створення власного токена(наприклад ERC-20) задля стимулювання та оплати вузлам за надане сховище.

- реалізація смарт контракту, котрий буде гарантувати доброчесність вузлів та вести їх рейтинг, або ж блокувати. Також, це має бути система, котра за допомогою POA(Proof Of Access) дозволяти вузлам зберігати нові данні та отримувати за це оплату.

- пошук файлів за метаданими

- зручний клієнтський інтерфейс, котрий дозволить підключитись до будь-якого вузла в мережі та виконувати взаємодію з мережею через нього.

Основні вимоги до системи для запуску власного вузла:

- хоча б 8Гб оперативної пам’яті

- хоча б 200МБ мережі

- хоча б 1ТБ диску(залежить, скільки файлів ви готові примати)

3.2 Проектування архітектури системи

Система має складну та різноманітну інфраструктуру, котра складається в основнуму з сховищ данних, внутнірньої однорангової мережі та зовнішньої мережі. Тому проектування системи вимагає значних архітектурних зусиль задля вдалого поєднання різних інфраструктурних компонентів та їх взаємодії.

3.2.1 Рівень доступу до даних

Наша система повинна мати змогу зберігати, індексувати, виконувати пошук по мережі файлів та даних різних розмірів, тому першою проблемою, котру потрібно вирішити – як організувати зберігання даних та вибрати інструменти для роботи з ними та над ними.

Тобто проблема – організувати рівень доступу до даних.

Відразу потрібно уточнити, що це інфраструктурна складова системи і вона повинна бути винесена у відповідний рівень інфраструктури, бути написаною максимально абстрактно та нічого не знати про домен та бізнес логіку самої системи.

Проаналізувавши майбутній домен та вимоги, було вирішено використовувати 3 сховища, кожне з котрих буде вирішувати свою проблему.

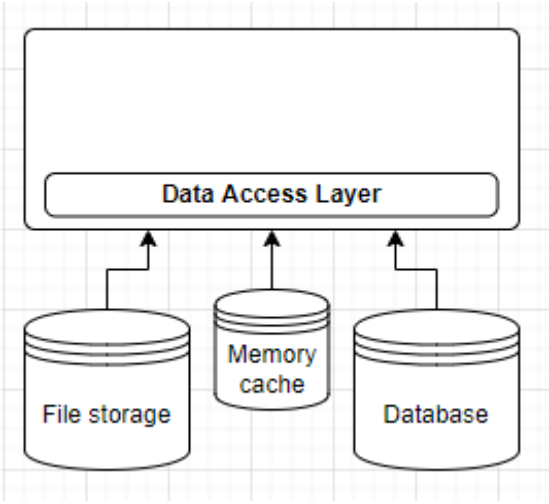


Рисунок 3.1 – Ілюстрація спроектованого рівня доступу до даних

File Storage (Local file system)

Перше та основне сховище – це власне локальна файлова система, задача котрої буде зберігати власне файли різного розміру. Для швидкого пошуку, індексування та запобігання дублюванню, буде використовуватись спеціальний формат іменування.

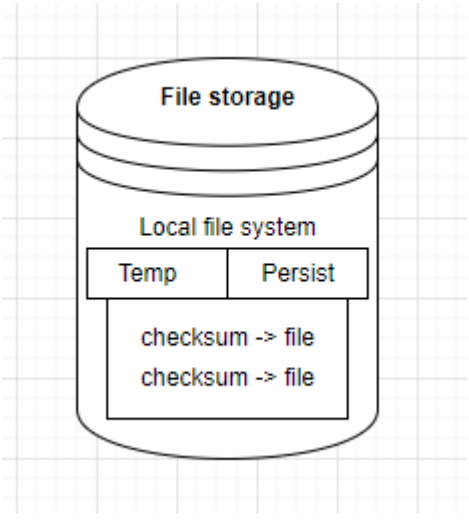


Рисунок 3.2 – Структура файлового сховища

Ім’я файлу буде рівноцінним його хеш сумі, котрі буде передаватись з клієнтської частини, перевірятись нодою.

Даним рішенням вирішено одночасно дві проблеми – індексування файлів у сховищі та гарантування їх цілісності при передачі та записі.

Сховище буде розділено на дві частини – розділ, котрий буде зберігати тимчасові та службові файли та час від часу очищатися та власне розділ, котрий буде зберігати постійні клієнтські файли та гарантувати постійний доступ до них.

Database (MySQL)

Інше сховище – база даних. Як було вказано раніше - обрано MySQL базу даних, як основне сховище даних.

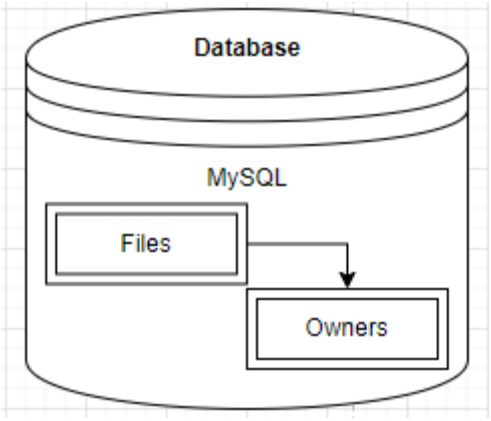


Рисунок 3.3 – Структура постійного сховища бази даних для індексування та пошуку файлів та їх авторів

В ній будуть зберігатись метадані про файли та права доступу до них для різних користувачів. База даних фактично буде книгою для довідок, котра повинна вказувати, які файли містяться у локальній файловій системі, їх опис та права на них.

Також, вона буде вирішувати один із основних проблем – пошук файлів локально. Адже набагато швидше та простіше перевірити наявність файлу за його метаданими, власником або хеш сумою у структурованих реляційних таблицях, ніж все це тримати та шукати у файловому сховищі.

В перспективі на даний рівень доступу до даних може бути покладено ще вирішення ще дуже багатьох проблем. Таких, як індексування операцій, зберігання історії мережі та інші.

Memory cache

Та поки остання, найпростіше, але не менш важливе – швидке сховище для кешу та тимчасових, службових даних.

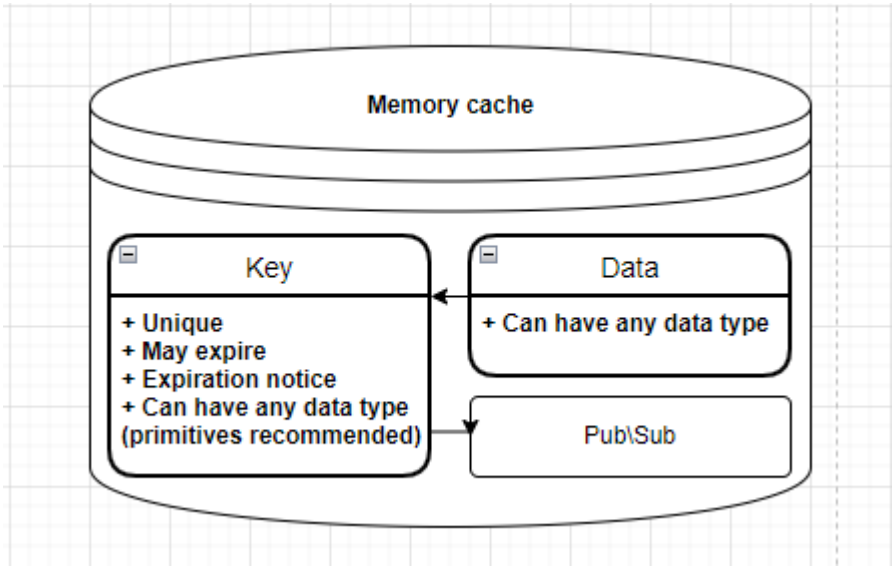


Рисунок 3.4 – Структура кешу в пам’яті.

В якості такого сховище було вибрано – оперативну пам’ять. Існують такі рішення, як Redis, задача котрого зберігати дані, котрі не надто важливі, але повинні мати можливість швидкого доступу.

В даній системі немає необхідності використовувати Redis, адже головна його перевага – можливість взаємодіяти з ним з різних інстансів одночасно, тобто масштабувати пам’ять та робити її спільною для всіх.

Потреби в даній функціональності немає, адже ми будуємо децентралізовану систему, де кожна вершина повинна мати власний контекст та ізольовані від інших дані.

Саме тому було вирішено побудувати більш просту систему зберігання таких даних на базі оперативної пам’яті. Рішення повинно підтримувати зберігання даних у форматі key – value, мати змогу налаштовувати час життя кожного ключа та систему оповіщення щодо закінчення терміну життя ключа. Тобто, крім прямого зберігання даних, як і Redis, наше рішення буте мати вбудовану підтримку сповіщень на базі принципу pub/sub.

І на відміну, від Redis, наше сховище може зберігати не лише серіалізовані дані, а й посилання на області пам’яті, адже воно використовує спільну пам’ять з застосунком.

3.2.2 Внутрішня мережа. Взаємодія між вершинами та транспорт

Для побудови внутрішньої мережі нам необхідно визначитись з транспортом та тим, як наші вузли будуть взаємодіяти.

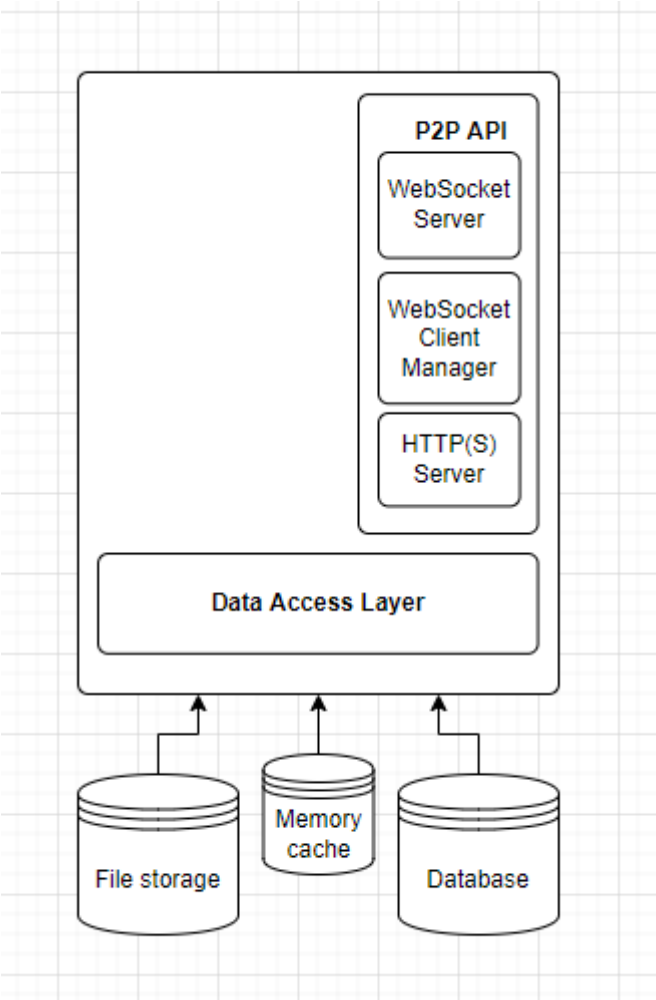


Рисунок 3.5 – Рівень доступу до внутрішньої однорангової мережі.

Для визначення з транспортом необхідно визначити вимоги та протоколи взаємодії. Перш за все на внутрішня мережа повинна мати змогу швидко розповсюджувати запити, інформацію та події між вузлами та мати отримувати відповіді від віддалених вузлів, до котрих немає прямого підключення.

Головна задача, котру повинна вирішувати мережа – це дослідження стану віддалених вузлів, всієї мережі та пошук контенту за метаданими(спершу за хеш сумою).

Для постійного зв’язку між вузлами було вирішено використовувати протокол WebSocket, котрий забезпечує двосторонній зв’язок.

Було вирішено розділити вхідні та вихідні з’єднання з іншими вершинами, задля більш оптимального пошуку по мережі та одному і методів запобіганню циклічних запитів(інші методи боротьби з циклічними запитами будуть розглянуті далі).

Тобто ми маємо 2 сутності, котрі майже однаково можуть працювати з запитами, обробляти їх та надавати відповіді, але розділяють з’єднання за критерієм вхід\вихід.

Ці сутності - WebSocket Server та WebSocket Client.

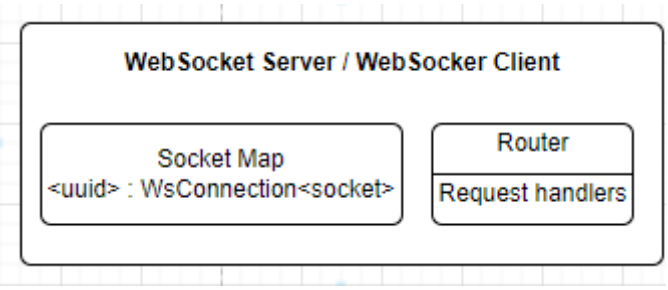


Рис.3.6. Компонент вебсокет серверу та клієнту.

Під час запуску застосунку автоматично встановлюється з’єднання до віддалених вузлів, котрі необхідно вказати в конфігурації.

Через встановлені з’єднання ми можемо відправляти запити та події в мережу, котрі будуть розповсюджуватись по всіх вершинах.

Під час підключення кожного нового вузла, йому надається унікальний ідентифікатор в межах вузла.

Також, ми можемо обмежувати кількість вхідних з’єднань до нашого вузла та обмежувати адреси з котрих можливо встановити з’єднання.

Проблему розповсюдження по мережі даних\подій вирішено. Залишилось вирішити, як отримувати відповіді від віддалених вузлів з котрими немає прямого з’єднання.

Для цього можемо використовувати технологію WebHooks. Наприклад, нам потрібно розповсюдити по мережі запит на пошук файлу.

Кожен вузол, котрий отримує запит на пошук повинен перевірити власне сховище та відправити даний запит до інших вузлів, до котрих приєднаний.

Якщо вузол має шуканий файл, йому потрібно мати спосіб сповістити про це найпершого вузла, котрий цей запит у мережі створив.

Для цього вузол, котрий породжує запит у мережу, повинен вказати в запиті, на котру адресу відправити позитивну відповідь та унікальний ідентифікатор запиту.

Для цього ми використовуємо додатковий Http Server, котрий повинен буде отримувати та обробляти WebHook відповіді та накладати їх на раніше створені запити. Як будуть працювати дані запити більш детально буде вказано в наступних розділах.

3.2.3 Зовнішня мережа. Транспорт для інтерфейсу взаємодії користувача

Перш за все необхідно зазначити, що можливі будь-які форми взаємодії з вузлом. Починаючи від CLI(Command line interface) закінчуючи веб, мобільними та десктоп застосунками. Вузол повинен надавати універсальне API для інтеграції та роботи з внутрішньою мережею.

Для простих запитів в межах вузла було вирішено використовувати, як транспорт, HTTP сервер, котрий буде мати змогу отримати запит та відправити відповідь. Він не матиме стану та буде розроблений за методологією REST.

Але єдиним HTTP сервером без збереження стану неможливо обійтись в межах нашої задачі, адже в більш складних запитах, як пошук в мережі або дослідження мережі, дані для відповіді приходять асинхронно частинами і потрібно мати змогу переправити їх на сторону клієнта.

Задля цього було вирішено використовувати згаданий раніше універсальний компонент, котрий знайшов своє місце і в даній задачі – WebSocket Server.

Клієнтська частина зможе дослідити мережу, обрати вузол для роботи з мережею, підключитись до нього та працювати з усією мережею через його інтерфейс.

Підключившись до вузла, буде можливо надсилати запити з синхронною відповіддю через HTTP з'єднання, так і більш складні асинхронні запити, відповіді на котрі будуть приходити частинами в продовж певного часу.

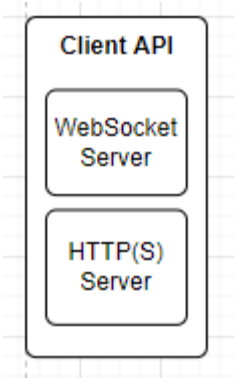


Рисунок 3.7 – Рівень користувацького інтерфейсу взаємодії з вузлом та мережею

3.2.4 Узагальнення розробленої інфраструктури вузла

Підсумовуючи проектування інфраструктури розроблюваного застосунку та об'єднавши всі вище описані рішення, можемо продемонструвати схематично напрацювання.

Можливо, в перспективі рівень інфраструктури буде розширюватись. Наприклад, додати підтримку розумних контрактів в інших децентралізованих мережах, задля підтримки оплати за зберігання приватних файлів, гарантування доступності файлів, ведення обліку доброчесності вузлів(вузол, котрий не гарантує доступності збережених файлів або втрачав файли повинен бути або виключений із мережі та добавлений в чорний список, або ж мати певні обмеження в майбутньому).

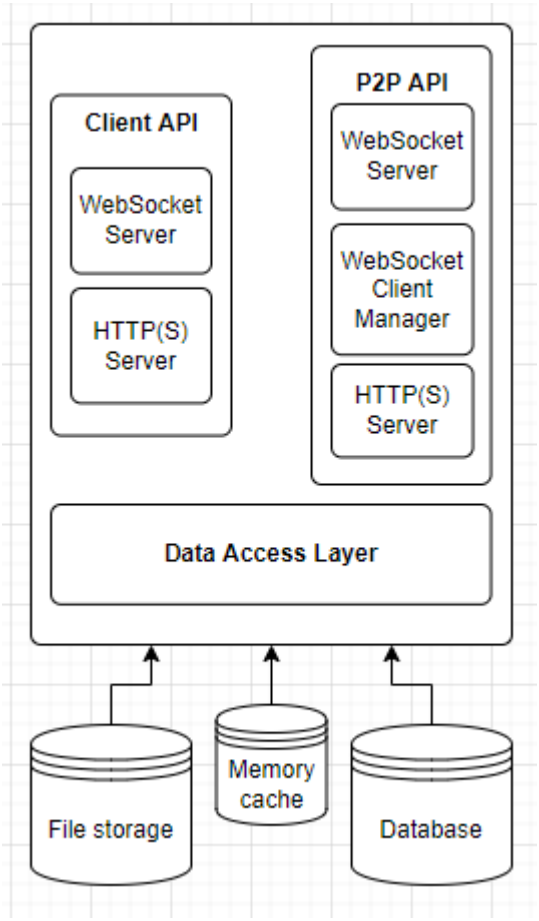


Рисунок 3.8 – Основні компоненти інфраструкторного рівня вузла.

Зовнішні інтерефейси вузла.

Можливо, в перспективі рівень інфраструктури буде розширюватись. Наприклад, додати підтримку розумних контрактів в інших децентралізованих мережах, задля підтримки оплати за зберігання приватних файлів, гарантування доступності файлів, ведення обліку доброчесності вузлів(вузол, котрий не гарантує доступності збережених файлів або втрачав файли повинен бути або виключений із мережі та добавлений в чорний список, або ж мати певні обмеження в майбутньому).

Загальна схема взаємодії з інфраструктурою застосунку.

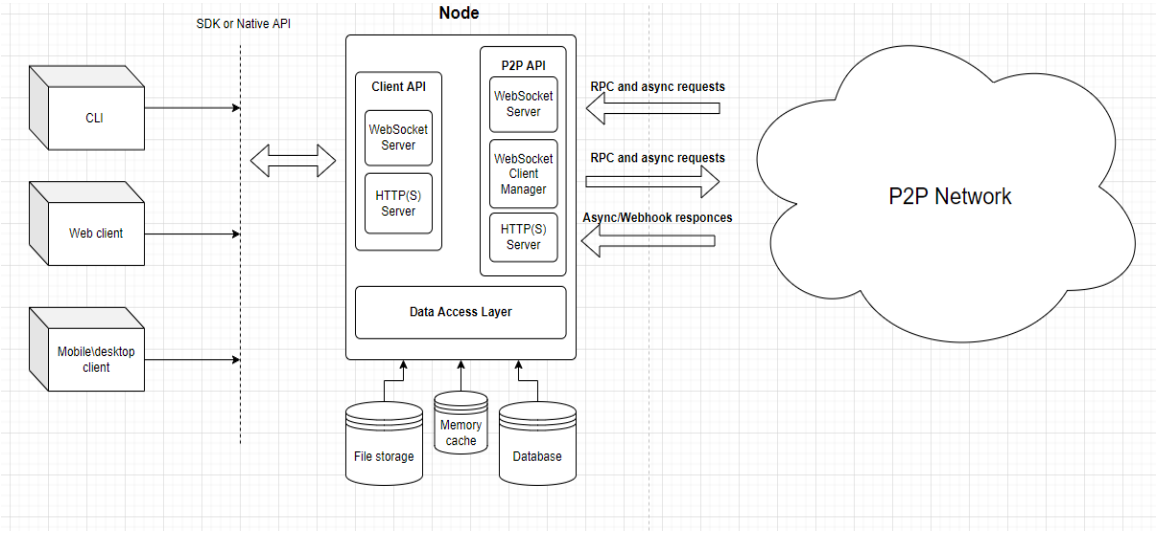


Рисунок 3.9 – взаємодія вузла із користувацькими застосунками, одноранговою мережею та рівнем доступу до даних.

Як видно зі схеми, клієнтська частина може взаємодіяти з мережею лише через запущений вузол. Вузли можуть бути, як повними, що приймають та зберігають данні та файли, надаючи власні ресурси, так і легкі вузли, котрі лише взаємодіють з мережею. Це повинно стимулювати розвиток мережі.

3.3 Розробка дизайну системи

Система повинна вирішувати три основні пробеми: як зберігати дані, як їх захищати та привласнювати відповідним користувачам та як виконувати пошук по розподіленій мережі. Саме тому далі будуть детально описані відповідні рішення поставлених проблем.

3.3.1 Проектування сховища

Як було вказано раніше, сховище складатиметься з 3 компонентів: файлового, бази даних, та локального кешу. В ролі постійного сховища, котре повинно надійно зберігати та індексувати дані буде саме файлова система та база даних.

Система повинна запобігати дублювання даних. Це можна вирішити за допомогою хеш суми взятої з даних, котрі необхідно зберігати.

Також, система може обмежувати доступ до файлів на рівні власника.

Для зручного пошуку файла в системі та запобіганню дублювання всі файли будуть мати ім’я, котре міститиме його хеш суму. І якщо даний файл існує, то необхідно лише додати нового власника у базі даних, без повторного збереження файлу.

Також, хеш сума буде гарантувати цілісність даних, що відправляє користувач. На етапі відправки клієнтська частина також братиме хеш суму та окремо передаватиме її в запиті.

Перейдемо більш детальніше до проектування сутностей бази даних.

В першу чергу нам необхідно спроектувати модель файлу.

Файл повинен зберігати ім’я, хеш для зручного пошуку та його індексування, розмір для швидкого розрахунку використаного обсягу пам’яті та можливості накладати ліміти можливості вузла.

Також, до файла повинні прикріплюватись його автори, або власники, якщо це приватні данні. Ну і звичайно час створення, задля зручної відладки, моніторингу та можливості зберігати данні певний час, котрий може вказати користувач.

З вимог описаних вище, можна спроектувати наступну сутність файлу.

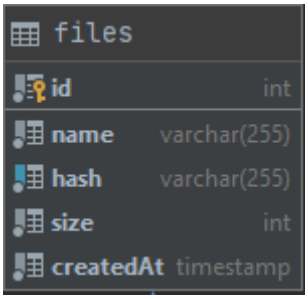


Рис.3.10. Сутність файлу в базі даних з описом полів та їх типів.

Тепер необхідно визначити, які саме данні необхідно прикріплювати разом з сутністю автора файлу.

Перш за все, це адресу користувача, котра має спеціальний формат, унікальна та за допомогою неї у буде відбуватися авторизація в подальшому.

Також, потрібно вказати на котрий термін користувач хоче зберегти свої данні. Для зручності це буде вказуватись в годинах цілим числом.

Необхідно визначити тип доступу до даного файлу. Він може бути, як відкритим, так і приватним.

Ми будемо зберігати метадані в форматі JSON для того, щоб користувач міг прикріпити опис або ж якісь свої власні та службові дані. В майбутньому по цьому опису можливо буде проводити пошук в мережі.

В результаті отримаємо наступну сутність.

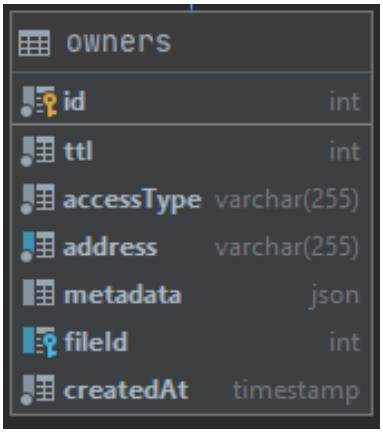


Рисунок 3.11 – Сутність автору файлу в базі даних з описом полів та їх типів.

Можемо помітити, що саме сутність власника містить посилання(зовнішній ключ) на самий файл, що означає, що сутності пов’язані зв’язком багато до одного.

Тобто кожен файл може мати багато авторів та власників та запобігатиме дублюванню. За гарними практиками до кожна сутність має ідентифікатор – головний ключ в таблиці.

Для запобіганню дублювання файлів та авторів відповідні сутності мають унікальні індекси. На сутності файлу встановлено індекс на поле з хешем. На сутності з власником встановлено індекс на адресу власника та зовнішній ідентифікатор файлу, що робить неможливим завантаження однакових файлів декілька разів на один вузол.

В результаті у нас вийшла не складна, але досить ефективна модель, котра вирішує всі поставлені задачі та вимоги.

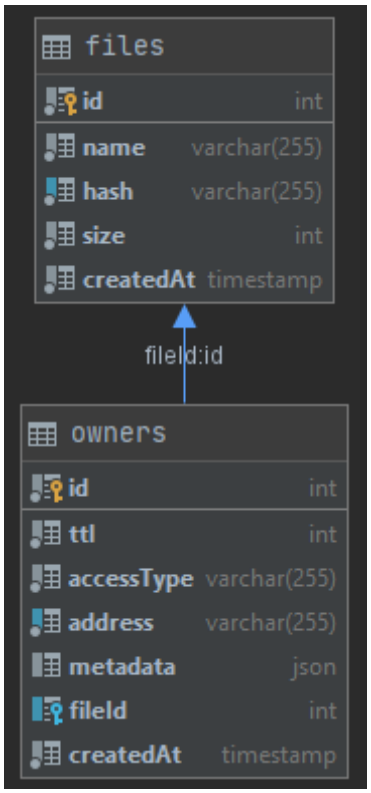


Рисунок 3.12 – Демонстрація зв’язків сутностей в реляційній моделі.

Файлова структура буде зберігатись у двох каталогах.

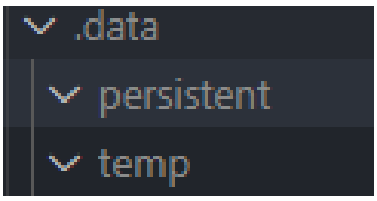


Рисунок 3.13 – Структура файлового сховища.

Перший – постійний та другий тимчасовий. З назви можна зрозуміти, що в постійному будуть зберігатись дані, котрі користувачі зберігають на вузлах, а в тимчасовому службові та нові непідтверджені файли. Тимчасове сховище буде час від часу очищатись, адже не може містити критичних даних.

3.3.2 Авторизація та верифікація користувачів

У децентралізованих системах неможливо використовувати звичні методи авторизації за допомогою паролю, адже ми відправляємо свої секретні дані на невідомий нам вузол і не можемо бути впевненими, що він є надійним та не має на меті використати наші секрети.

Отже, постає проблема, як ми можемо розпізнати користувача і як користувач може довести, що це саме він не зберігаючи та не отримуючи його секрети в системі та не використовуючи попередньої реєстрації.

Проблему можна вирішити за допомогою асиметричного шифрування та підписів.

Процес цифрового підпису працює так само, як і шифрування. Шифрування буває двох типів: **асиметричне та симетричне шифрування**. Процес асиметричного шифрування працює шляхом створення пари ключів із відкритим і закритим ключем. Закритий ключ зберігається в секреті від усіх, крім творця ключа, тоді як відкритий ключ доступний кожному. Дані шифруються за допомогою приватного ключа, а при необхідності розшифровуються відкритим ключем. Симетричне шифрування використовує лише один ключ як для шифрування, так і для **дешифрування**. Оскільки асиметричне шифрування є більш безпечним, ніж симетричне шифрування, воно, як правило, використовується частіше. Під час надсилання даних

одержувачу правильним методом шифрування є шифрування даних відкритим ключем одержувача, оскільки це означає, що лише власник пари ключів може розшифрувати ці дані.

Цифрове підписання працює навпаки. Дані підписуються шляхом **хешування** повідомлення за допомогою алгоритму хешування та приватного ключа відправника. Це створює хеш-дайджест, який можна відтворити лише за допомогою одного з ключів у парі ключів, створеної відправником. Потім одержувач отримує повідомлення, хеш-дайджест і відкритий ключ, якщо він його ще не мав. Потім одержувач використовує відкритий ключ відправника для хешування отриманого повідомлення. Якщо отриманий хеш-дайджест збігається з хеш-дайджестом, надісланим разом із повідомленням, значить, особу відправника підтверджено. Це також підтверджує, що дані не були змінені під час доставки. Однак сам по собі підпис не гарантує, що дані не були перехоплені та не прочитані.

Щоб захистити дані від компрометації та одночасно автентифікувати відправника, шифрування та цифрове підписання використовуються разом. Вони також використовуються в тандемі для виконання стандартів відповідності для компаній. Такі стандарти, як **Федеральні стандарти обробки інформації (FIPS)** або **Загальний регламент захисту даних (GDPR)**, вимагають від компаній якомога надійніше захищати дані разом із аутентифікацією даних, отриманих від інших осіб. Шифрування та цифрове підписування забезпечують досягнення цих стандартів, а також те, що користувачі можуть бути в безпеці, знаючи, що дані, які надсилаються їм і від них, не будуть порушені.

Конфіденційні або конфіденційні дані завжди повинні бути зашифровані та підписані для їхньої власної безпеки. Спільне використання шифрування та підпису гарантує досягнення основних цілей криптографії, конфіденційності, цілісності, автентичності та недопущення відмови. Конфіденційність і цілісність досягаються, коли дані шифруються асиметрично, оскільки тільки

призначений одержувач може розшифрувати повідомлення. Невідмовність і автентичність відбуваються завдяки цифровому підпису. Невідмовність означає, що, використовуючи техніку цифрового підпису, відправник будь-якої інформації не може в майбутньому сказати, що він не надсилав дані, оскільки використання їх закритого ключа підтверджує, що він надіслав дані.

Ми використовуватимемо пару ключів типу ed25519, що використовує еліптичні криві Едвардса та на даний час є одним із найпопулярніших форматів пар ключів, що на відміну від RSA є невеликими за розміром та зручними у використанні.

Як було вказано раніше, що пара містить приватний та публічний ключ. Але до цих даних, для зручності ми також генеруватимемо ще одне поняття – адресу користувача, що буде братися з публічного ключа.

Для наочності зобразимо схему, як ключі пов'язані між собою.

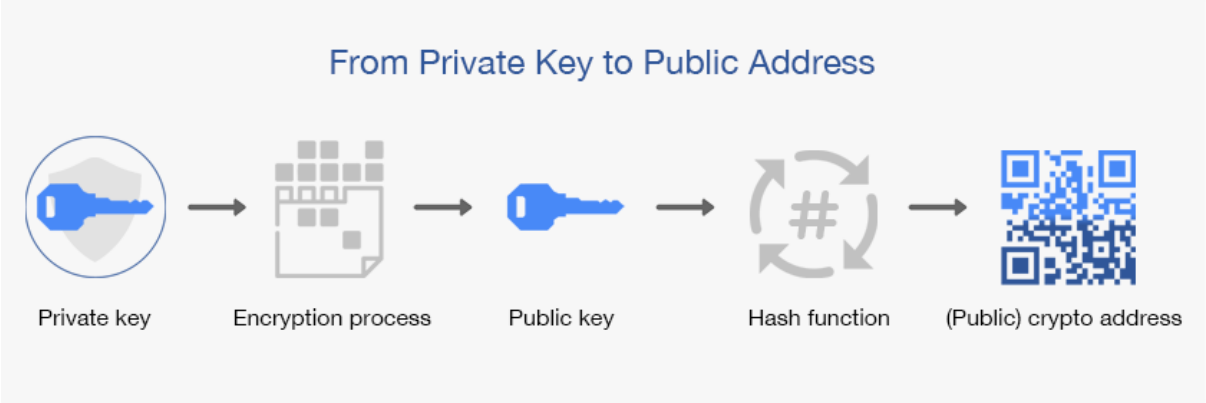


Рисунок 3.14 – Операції перетворення та взаємодія пари ключів між собою [31]

Спробую описати весь процес верифікації користувача на прикладі запиту приватного файлу з вузла.

Перш за все користувач виконує пошук по мережі та отримує посилання за котрим може завантажити власний файл. Як було вказано раніше, ми визначаємо власників файлу саме за адресою користувача.

Клієнтська частина повинна відіслати запит на вузол з проханням згенерувати тимчасовий nonce(випадкові дані середнього розміру) та прикріпити до запиту свою адресу, котру хоче верифікувати.

Ми отримуємо адресу клієнта, генеруємо nonce, тимчасово зберігаємо в кеш отриманий nonce по індексу адреси та відправляємо ці дані у відповіді.

Далі частина клієнту повинна підписати отримані дані свої приватним ключем. Сформувати запит на отримання файлу та прикріпити до нього підписані дані, отримані раніше та свій публічний ключ.

На стороні серверу з публічного ключа ми можемо отримати адресу, за адресою дістати згенеровані раніше дані, якщо вони ще існують та верифікувати підпис. Якщо підпис дійсний, то даний клієнт дійсно той, за кого він себе видає і ми можемо віддати йому необхідні данні.

Перевага даного методу полягає в тому, що на сервер фактично не надходять секрети користувача та неможливо використати один і той же підпис для декількох запитів, адже кожен раз необхідно підписати новий випадковий nonce. Це суттєво збільшує рівень безпеки, як користувача, так і всієї системи.

Подібні методи використовуються у блокчейн системах.

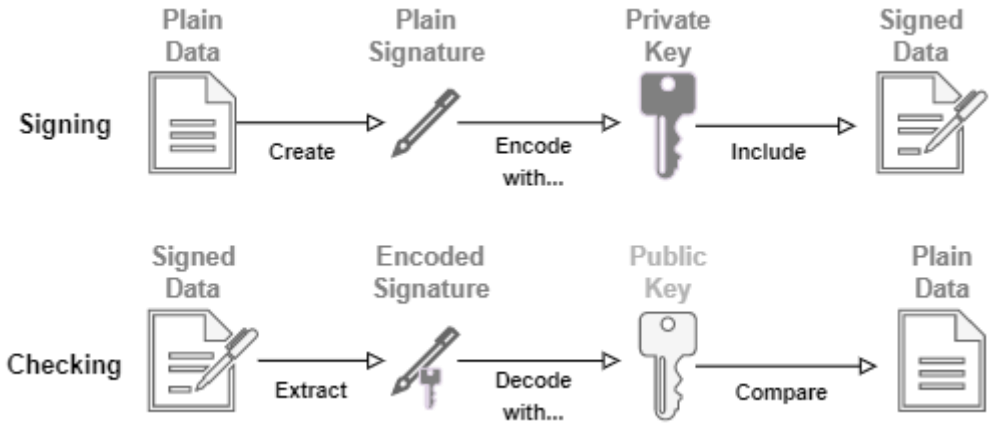


Рисунок 3.15 – Підписання та порівняння асиметричних цифрових підписів [32]

3.3.3 Пошук даних в мережі

На даному етапі еволюції системи планується виконувати пошук по хеш сумі, але в подальшому можливо шукати по метаданих, котрі вказав автор при завантаженні даних.

Як було вказано раніше, для постійного зв'язку між вузлами будемо використовувати технологію WebSocket. Кожен вузол міститиме два набора сокетів: клієнти(вузли, котрі підключились до вузла) та сокети до інших вузлів, котрі ініціював сам вузол.

Перша з проблем і напевно найбільша, котра постає перед нами - циклічність графу мережі. Як відомо, будь-яка децентралізована мереже може мати цикли і це природньо.

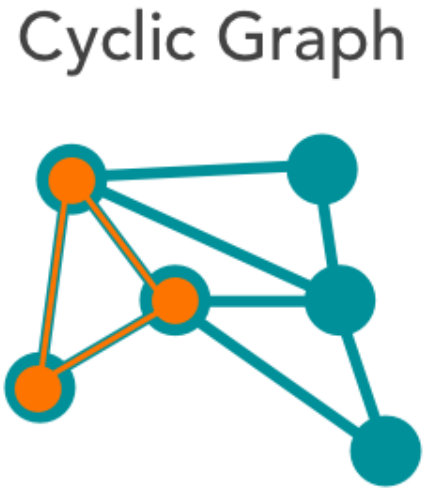


Рисунок 3.16 – Циклічний ненапрямлений граф [33]

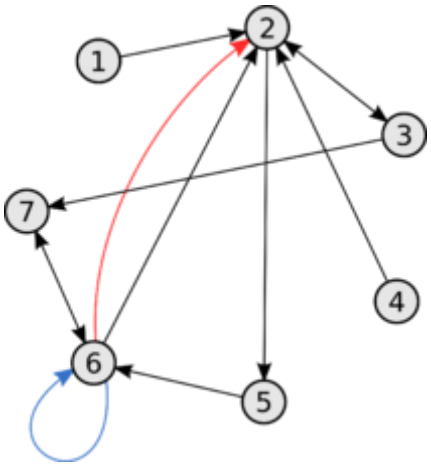


Рисунок 3.17 – Циклічний напрямлений граф [34]

Це означає, якщо ми будемо відправляти запит на пошук файлу, то він може нескінченно мандрувати мережею, що є проблемою.

Для того, щоб це вирішити було розроблено спеціальний метод, котрий гарантує шляху запиту в мережі та ліміт в часі поширення.

На початку необхідно зазначити, що ми отримуватимемо запити лише від сокетів клієнтів та надсилатимемо їх лише на сокети, котрі самі ініціювали. Це рішення означає, що граф нашої мережі буде напрямленим та циклічним.

Тобто, нам потрібно отримати запит, перевірити власний вузол на можливість його виконання та поширити далі.

Як було вказано раніше, для відправки відповіді на вузол, котрий згенерував запит, до запиту прикріпляється посилання за яким відправити успішну відповідь та використовується метод WebHook, котрий має асинхронну природу.

Для запобігання циклічних запитів використовуватимемо наступні прийоми:

- Кожен запит повинен мати час життя, котрий вказується в тілі запиту і за його вичерпанням вузол не повинен далі поширювати його(вузли мають синхронізований годинник з незначним відхиленням);
- Кожен запит повинен мати унікальний ідентифікатор, котрий також буде надсилатись у разі успішної відповіді для співставлення з запитом. Но найголовніше, що він буде використовуватись іншими вузлами для розпізнавання циклів. Вузол, отримавши запит, буде обробляти його, поширювати, якщо він ще дійсний та зберігати у тимчасовому кеші ідентифікатор запиту. При повторному надходженні, вузол повинен перевірити власний кеш на наявність даного запиту за ідентифікатором та у разі його існування – проігнорувати.

За допомогою вище вказаних прийомів ми можемо гарантувати відсутність циклічних запитів та контролювати час поширення.

Після вичерпання часу запиту, вузол, котрий його ініціював може переставати слухати відповіді по даному запиту.

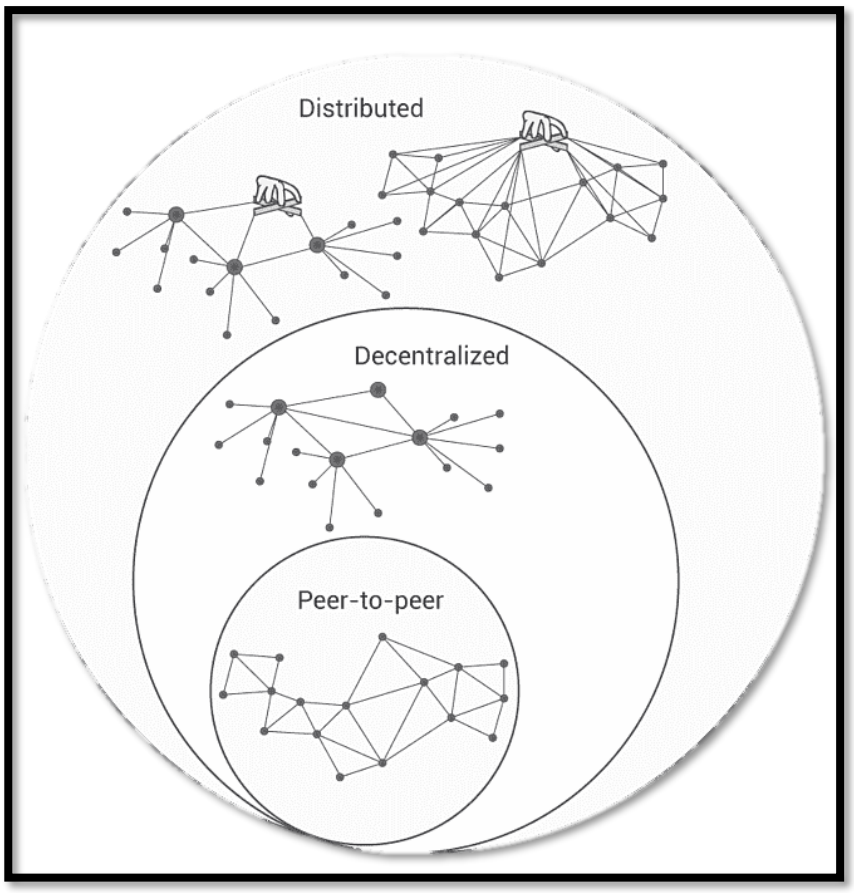


Рисунок 3.18 – Різновиди розподілених мереж та їх підтипи [35]

3.3.4 Проектування модулів системи та їх взаємодія

В центрі всієї системи лежить домена модель з сервісами.

В моделі знаходиться дві сутності, котрі описувались раніше: файлу та автора(власника) файлу.

В сервісах на даному етапі проектування знаходиться чотири основних елемента, котрі повинні покривати всі основні функції системи, а саме:

- Завантаження та вивантаження файлів (File service);
- Взаємодія з внутрішньою одноранговою мережею на прикладному рівні (Network service);
- Пошук елементів в мережі та дослідження мережі(Search service);

- Авторизація та верифікація користувачів та авторів файлів(Verification Service);

Ці сервіси в свою чергу використовують різноманітну інфраструктуру для взаємодії з різними сторонніми елементами системи.

Наприклад, файлова система підключена до застосунку за допомогою адаптера, котрий реалізовує спеціальний інтерфейс, котрий необхідний основним сервісам.

База даних підключена за допомогою шаблона Unit Of Work, котрий поєднує шаблони репозиторіїв сутностей бази та з’єднання до бази.

Даний шаблон дозволяє легко та зручно взаємодіяти з базою даних та виконувати звичайні запити та транзакції.

Хоча внутрішній кеш не використовує сторонніх сервісів, він також винесений на рівень інфраструктури та має пряме підключення та підключення через внутрішній pubsub, задля відстеження терміну дії ключів, аналогічно, як в сховищі Redis.

Окремим незалежним модулем винесено сервіси пов’язана з криптографією, котрі повинні уміти шифрувати, хешувати, генерувати пари ключів, підписувати та верифікувати повідомлення.

Всі залежності між сервісами та сутностями вирішуються за допомогою ін’єкції залежностей через конструктор.

Всі сервіси та відповідні елементи системи збираються в контейнер, котрий представляє всю логіку та можливості застосунку.

Даний контейнер доступний на рівні транспортних контролерів, котрі в свою чергу реєструються на раніше ініціалізованих маршрутизаторах.

Після ініціалізації контейнера, відбувається реєстрація рівня контролерів та запуск самого контейнера.

Запуск контейнера означає запуск та прослуховування всіх серверів, ініціалізацію з’єднань однорангової мережі з іншими вузлами та створення пулу з’єднань з базою даних.

Після цього кроку, всі наші сервіси готові до використання.

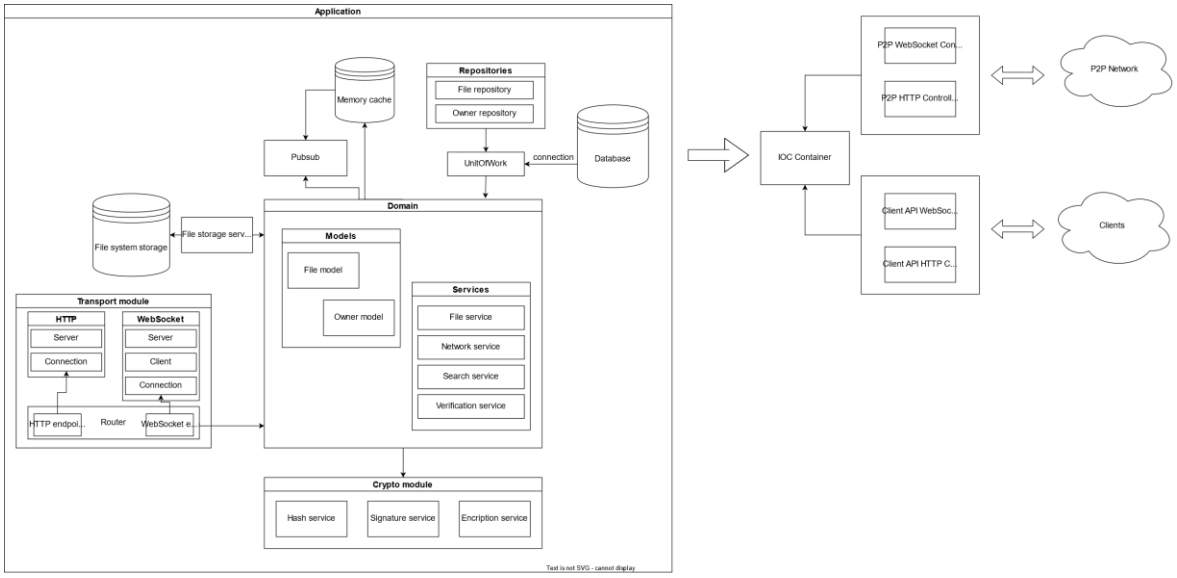


Рисунок 3.19 – Основні компоненти системи та їх взаємодія

Ініціалізація контейнера має наступний вигляд. Відбувається створення всіх сервісів, котрі мають вигляд ациклічного напрямленого графа.

```
const container = {
  //configs
  NetworkConfig,
  //system
  pubsub,
  memoryCache,
  //storage
  db,
  fileStorage,
  //crypto
  signatureService,
  //network
  networkRouter,
  networkHttpServer,
  networkWebsocketServer,
  networkWebsocketClientManager,
  //access
  verificationService,
  //api
  apiRouter,
  apiHttpServer,
  apiWebsocketServer,
  //services
  fileService,
  networkService,
  searchService,

  start: () =>
    Promise.all([
      networkHttpServer.start(),
      networkWebsocketServer.start(),
      networkService.start(),
      apiHttpServer.start(),
      apiWebsocketServer.start(),
    ]),

  stop: () =>
    Promise.allSettled([
      networkHttpServer.stop(),
      networkWebsocketServer.stop(),
      networkWebsocketClientManager.stop(),
      apiHttpServer.stop(),
      apiWebsocketServer.stop(),
    ]),
};
```

Рисунок 3.20 – Вміст та ініціалізація контейнера

Даний контейнер є готовим зібраним застосунком, в котрий необхідно лише зареєструвати сторонні контролери запитів, тобто API та запустити.

Приклад реєстрації контролерів та запуску контейнера:

```
'use strict';

const {useHttpNodeApi, useWebSocketNodeApi} = require('./src/api/node');
const {useHttpClientApi, useWebSocketClientApi} = require('./src/api/client');
const container = require('./container');

(async () => {
  //register node api
  useHttpNodeApi(container);
  useWebSocketNodeApi(container);
  //register client api
  useHttpClientApi(container);
  useWebSocketClientApi(container);
  //start application
  await container.start();
  //graceful shutdown appilication
  const shutdown = async () => {
    await container.stop();
    process.exit(0);
  };
  process.on('SIGINT', shutdown);
  process.on('SIGTERM', shutdown);
})();
```

Рисунок 3.21 – Реєстрація контролерів та запуск вузла

Після цього наш застосунок починає свою роботу та очікує на надходження запитів від мереж та на команду закінчення роботи для того, щоб мати змогу зупинитися без збоїв та дати час на завершення всіх критичних процесів.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було розглянуто архітектуру та дизайн розроблюваної захищеної розподіленої системи зберігання даних. Проаналізувавши вимоги до системи було спроектовано та описано архітектуру, прийнято відповідні рішення щодо проектування сховища, мережі та структури даних. Розроблено та продемонстровано відповідні алгоритми, котрі чітко вирішують поставлені задача, такі як: пошук даних в розподіленій системі, верифікація користувачів та гарантування унікальності збережених даних на вузлі. Також, було описано бачення наступних поколінь системи та її розвиток. Спроектовано компоненти мережі та маршрутизатори.

Наведено схеми будови системи з детальним описом компонентів та взаємодії між ними. Аргументовано використання певних підходів та шаблонів, вказано, які проблеми вони вирішують та наведено їх переваги. Продемонстровано збірку системи в єдиний компонент та описано, як відбувається ініціалізація та запуск вузла.

На даному етапі вже розроблено каркас системи та практично доведено зручність використання даної архітектури та підходів. Під час розширення функціоналу не виникає будь-яких проблем, адже всі рішення зручно розподілені по модулях та рівнях, що дозволяє взаємодіяти з будь-якими інтегрованими компонентами на високому рівні. Завдяки використанню підходу ін’єкції залежностей система легко розширюється та масштабується.

РОЗДІЛ 4.
ТЕСТУВАННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

Задля зручного використання системи було розроблено також простий та інтуїтивний SDK для платформи Node.JS, котрий поєднує в собі всі базові можливості розробленої системи. Через взаємодію з ним ми будемо проводити тестування та аналіз всієї нашої системи.

На базі нашого SDK можна дуже легко розробити будь-які клієнтські застосунки, що будуть взаємодіяти з нашою мережею на платформі Node.JS та виконавши деякі маніпуляції з модулями та поліфілами перенести це рішення на середовище виконання браузера.

Для виконання тестування необхідно запустити наш вузол, можна навіть декілька та поєднати їх в спільну мережу.

Після налаштування та запуску вузла в консолі повинно з’явитись повідомлення, що вузол почав прослуховувати виклики від клієнтів та від однорангової мережі по протоколах HTTP та WebSocket

```
Start server http://0.0.0.0:4001
Start WebSocket server ws://0.0.0.0:4000
Start server http://0.0.0.0:5001
Start WebSocket server ws://0.0.0.0:5000
```

Рисунок 4.1 – Запуск вузла

Далі ми можемо робити запити на нього.

Наприклад, зробимо запит на збереження нового файлу в мережі.

```
const uploadFileTest = async () => {
  console.log('Upload file test starting');
  const path = testFilePath();
  try {
    await generateLargeFile(path, fileSeed);
    const checksum = await getChecksum(path);
    const readStream = fs.createReadStream(path);
    const result = await SDK.httpApiClient.uploadFile({
      name: 'TEST_FILE_NAME',
      ttl: 99999,
      checksum: checksum,
      accessType: SDK.FileAccessType.OPEN,
      file: readStream,
      secret: account.privateKey,
    });
    const {name, hash} = result;
    console.dir(result);
    assert.deepEqual(
      {name, hash},
      {name: 'TEST_FILE_NAME', hash: checksum},
      'Upload file error: response does not match request',
    );
    return true;
  } catch (err) {
    console.log(err);
    throw new Error('Upload file test error');
  } finally {
    await rm(path);
    console.log('Upload file test completed');
  }
};
```

Рисунок 4.2 – генерація та запис файлу на вузол

Виконавши даний тест ми отримаємо наступний вивід

```
Upload file test starting
{
  id: 63,
  name: 'TEST_FILE_NAME',
  hash: 'wVDbEE_JslzfpcTlRoeL7Q',
  size: 293,
  createdAt: '2022-06-02T15:07:38.000Z'
}
Upload file test completed
```

Рисунок 4.3 – результат завантаження файлу на вузол

При повторному завантажені даного файлу у відповідь отримуємо такий же результат, що свідчить про те, що вузол слідкує за тим, щоб не створювались дублюючі дані.

Виконавши даний запит до нашої бази даних, можна переконатись, що файл був дійсно створений та до нього був прикріплений користувач.

select * from files left join owners on files.id = owners.fileId;

Отримаємо наступне



files.id	name	hash	size	files.createdAt	owners.id	ttl	accessType	address	metadata	fileId	owners.createdAt
1	63	TEST_FILE_NAME	wVDbEE_JslzfpcTlRoeL7Q	293	2022-06-02 18:07:38	58	99999	OPEN	af1b1499ee6be45b5cb383cd487444ce	{}	63	2022-06-02 18:07:38

Рисунок 4.4 – результат виконання запиту в базу даних

Також, можемо пересвідчитись, що на диску був також створений файл

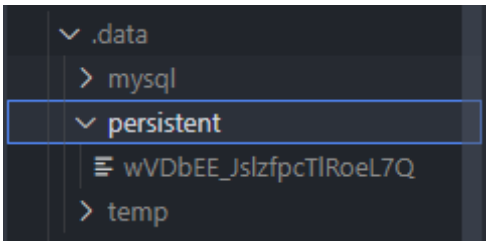


Рисунок 4.5 – Скриншот файлового сховища вузла

Далі реалізуємо тестування пошуку файлу за унікальним хешем

```
const searchFileByHashTest = async () => {
  console.log('Search file by hash test starting');
  try {
    const files = [];
    const onFileFound = (payload) => files.push(payload);
    wsApiClient.connect(wsNode);
    wsApiClient.subscribe(WS_EVENTS.FILE_FOUND, onFileFound);
    const finish = new Promise((res, rej) => {
      function onSearchFinish() {
        wsApiClient.unsubscribe(WS_EVENTS.FILE_FOUND, onFileFound);
        wsApiClient.unsubscribe(WS_EVENTS.FILE_SEARCH_FINISHED, onSearchFinish);
        wsApiClient.unsubscribe(WS_EVENTS.ERROR, onError);
        res(files);
      }
      function onError(_, err) {
        wsApiClient.unsubscribe(WS_EVENTS.FILE_FOUND, onFileFound);
        wsApiClient.unsubscribe(WS_EVENTS.FILE_SEARCH_FINISHED, onSearchFinish);
        wsApiClient.unsubscribe(WS_EVENTS.ERROR, onError);
        console.dir(err);
        rej(err);
      }
      wsApiClient.subscribe(WS_EVENTS.FILE_SEARCH_FINISHED, onSearchFinish);
      wsApiClient.subscribe(WS_EVENTS.ERROR, onError);
    });
    await wsApiClient.searchFileByHashRequest({
      hash: existChecksum,
      secret: account.privateKey,
      timeout: 1000,
    });
    const result = await finish;
    console.log(`Found files: ${result.length}`);
    const matched = !result.some(({file}) => file.hash !== existChecksum);
    assert.equal(matched, true, 'Found file does not matched with requested');
  } catch (err) {
    console.log(err);
    throw new Error('Search file by hash test error');
  } finally {
    console.log('Search file by hash test completed');
    wsApiClient.disconnect();
  }
};
```

Рисунок 4.6 – пошук файлу в мережі за унікальним ідентифікатором

Виконавши даний код, отримаємо наступний результат

```
Search file by hash test starting
Found files: 1
Search file by hash test completed
```

Рисунок 4.7 – результат пошуку створеного файлу за унікальним ідентифікатором

Спробуємо знайти наш файл за частиною його імені, задля цього напишемо наступний тест.

```
const searchFilesByNameTest = async () => {
  console.log('Search file by name test starting');
  const NAME = 'test_file';
  try {
    const files = [];
    wsApiClient.connect(wsNode);
    const onFileFound = (payload) => files.push(payload);
    wsApiClient.subscribe(WS_EVENTS.FILE_FOUND, onFileFound);
    wsApiClient.subscribe(WS_EVENTS.FILE_FOUND, onFileFound);
    const finish = new Promise((res, rej) => {
      function onSearchFinish() {
        wsApiClient.unsubscribe(WS_EVENTS.FILE_FOUND, onFileFound);
        wsApiClient.unsubscribe(WS_EVENTS.FILE_SEARCH_FINISHED, onSearchFinish);
        wsApiClient.unsubscribe(WS_EVENTS.ERROR, onError);
        res(files);
      }
      function onError(_, err) {
        wsApiClient.unsubscribe(WS_EVENTS.FILE_FOUND, onFileFound);
        wsApiClient.unsubscribe(WS_EVENTS.FILE_SEARCH_FINISHED, onSearchFinish);
        wsApiClient.unsubscribe(WS_EVENTS.ERROR, onError);
        rej(err);
      }
      wsApiClient.subscribe(WS_EVENTS.FILE_SEARCH_FINISHED, onSearchFinish);
      wsApiClient.subscribe(WS_EVENTS.ERROR, onError);
    });
    await wsApiClient.searchFilesByNameRequest({
      name: NAME,
      secret: account.privateKey,
      timeout: 1000,
    });
    const result = await finish;
    console.log(`Found files: ${result.length}`);
    const matched = !result.some(({file}) => !file.name.toLowerCase().includes(NAME.toLowerCase()));
    assert.equal(matched, true, 'Found file does not matched with requested');
  } catch (err) {
    console.log(err);
    throw new Error('Search file by name test error');
  } finally {
    console.log('Search file by name test completed');
    wsApiClient.disconnect();
  }
};
```

Рисунок 4.8 – Пошук файлу за частиною імені

Виконавши даний тест, отримаємо аналогічний попередньому пошуку результат.

Спробуємо завантажити наш файл з віддаленого вузла

```
const downloadFileTest = async () => {
  console.log('Download file test starting');
  try {
    const readFileStream = await SDK.httpApiClient.getFile({
      checksum: existChecksum,
      secret: account.privateKey,
    });
    const chunks = [];
    for await (const chunk of readFileStream) {
      chunks.push(chunk);
    }
    const data = Buffer.concat(chunks).toString('utf-8');
    const matched = data.includes(fileSeed);

    assert.equal(matched, true, 'Downloaded file not matched with requested file');
    return;
  } catch (err) {
    console.log(err);
    throw new Error('Download file test error');
  } finally {
    console.log('Download file test completed');
  }
};
```

Рисунок 4.9 – завантаження файлу з віддаленого вузла

Та останній тест по видаленню файлу з віддаленого вузла

```
const deleteFileTest = async () => {
  console.log('Delete file test starting');
  try {
    const success = await SDK.httpApiClient.deleteFile({
      checksum: existChecksum,
      secret: account.privateKey,
    });
    assert.equal(typeof success, 'boolean', 'Unexpected response');
    return;
  } catch (err) {
    console.log(err);
    throw new Error('Delete file test error');
  } finally {
    console.log('Delete file test completed');
  }
};
```

Рисунок 4.10 – Видалення файлу з віддаленого вузла

Було протестовано основний розроблений функціонал. Переконатися в успішності тестів можна з наведених вище рисунків.

Необхідно звернути увагу на те, що майже в кожному виклику до SDK ми передаємо свій секретний ключ за допомогою підписується випадково згенерований вузлом nonce та вузол за допомогою даного підпису проводе авторизації та визначає права доступу до даних.

Детальніше дані процедури будуть описані у схемах авторизації та у вихідному коді проекту.

Для прикладу тестування проводилось з правильними даними. Всі інші тести по критичним місцям було проведено окремо, як у автоматичному, так і в ручному режимі.

На даному етапі система вміє успішно авторизувати користувачів, зберігати та контролювати їх файли, видаляти їх за запитом власника та проводити пошук даних за унікальним ідентифікатором, котрий будується на основі хеш суми та за частиною імені збережених даних.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі було детально розглянуто основні можливості розробленої системи та протестовано її функціонал. Розглянуто роботу спеціально розробленого SDK для зручної взаємодії з мережею та можливості розробки та інтеграції з даною системою інших застосунків.

Протестувавши систему не було виявлено жодних недоліків у її роботі на даному етапі. Було звернено увагу на те, що кожен запит супроводжується спеціальною процедурою авторизації користувача.

Продемонстровано можливість пошуку файлів за різними критеріями у розподіленій мережі, що є досить новим напрямом у даному типі систем.

Даний розділ можна розцінювати, як практичні рекомендації та приклади роботи з нашою системою, часткова документації по інтеграції з нею.

ВИСНОВКИ

Під час реалізації даної дипломної роботи було оцінено наявні розподілені системи збереження даних, проведено їх аналіз та виявлено слабкі місця систем.

Одним із найбільш проблемних місць виявилось, що системи не здатні буде по справжньому зручними у використанні та виконувати свої базові функції без допоміжних централізованих систем пошуку та індексації вмісту їх мереж.

Також, проблемою стало те, що користувач, завантаживши свої файли та дані до системи не міг бути впевненим, що до них не отримає доступ будь хто інший.

У першому розділі ми проводили експерименти та досліджували наявні системи, намагалися проаналізувати та виявити їх сильні та слабкі сторони, задля того, щоб вирішити їх у нашій майбутній системі та наслідувати найкращі існуючі практики.

У другому розділі були розглянуті технології на базі яких ми збирались будувати наше рішення. Аргументували свій вибір по кожній технології, виходячи з її технічних характеристик та сильних сторін. Також, було порушено тему безпеки в децентралізованих системах та описано відмінності централізованих та децентралізованих систем в питаннях ідентифікації, аутентифікації та авторизації користувачів, що з ними взаємодіють.

У третьому розділі було вже спроектовано архітектура нашої системи, розроблено відповідні дизайнерські рішення та розпочато їх реалізацію. Було зображено схеми інфраструктури вузла та всієї архітектури децентралізованої мережі, спроектовано сховище даних та розроблено алгоритми пошуку по мережі, котрі дозволяють виконувати ефективний пошук уникаючи циклічних запитів та алгоритм аутентифікації користувачів на основі пар секретного та публічного ключа.

У четвертому розділі проводились дослідження та тестування бази реалізованої системи. Була розвернута тестова мережа з декількох вузлів задля проведення тестів та налагодження. Розроблено SDK для більш простої взаємодії з мережею та на базі SDK розроблено тести, що покривають успішні випадки використання базових функцій системи. Продемонстровано ілюстрації, котрі підтверджують успішність проведення тестів та зроблено відповідні висновки. На етапі тестування було виявлено деталі та проблеми, котрі дозволило суттєво покращити розроблену систему.

В результаті система безвідмовно пройшла декілька ітерацій тестування та готова до використання та подальшого свого розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Comparison – Centralized, Decentralized and Distributed Systems [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/ (дата звернення 03.05.2022).
2. Centralized vs Decentralized vs Distributed Systems [Електронний ресурс] – Режим доступу до ресурсу: https://berty.tech/blog/decentralized-distributed-centralized (дата звернення 03.05.2022).
3. Decentralized systems engineering [Електронний ресурс] – Режим доступу до ресурсу: https://www.researchgate.net/figure/Decentralized-systems_fig2_318385532 (дата звернення 03.05.2022).
4. Distributed File System (DFS) [Електронний ресурс] – Режим доступу до ресурсу: https://www.techopedia.com/definition/1825/distributed-file-system-dfs (дата звернення 05.05.2022).
5. Network file system – A deep dive into a distributed file system protocol [Електронний ресурс] – Режим доступу до ресурсу: https://www.scaleyourapp.com/network-file-system-a-deep-dive-into-a-distributed-file-system-protocol/ (дата звернення 05.05.2022).
6. BIMServerless: Serverless Architecture, Function as a Service (FaaS) and Cryptography for BIM/IFC [Електронний ресурс] – Режим доступу до ресурсу: https://www.researchgate.net/project/BIMServerless-Serverless-Architecture-Function-as-a-Service-FaaS-and-Cryptography-for-BIM-IFC (дата звернення 07.05.2022).
7. IPFS powers the Distributed Web [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/ipfs/ipfs (дата звернення 06.05.2022).
8. Distributed Hash Tables (DHTs) [Електронний ресурс] – Режим доступу до ресурсу: https://docs.ipfs.io/concepts/dht/ (дата звернення 08.05.2022).

9. BitTorrent (BTT) White Paper [Електронний ресурс] – Режим доступу до ресурсу: https://www.bittorrent.com/btt/btt-docs/BitTorrent_(BTT)_White_Paper_v0.8.7_Feb_2019.pdf (дата звернення 08.05.2022).
10. Blockchain [Електронний ресурс] – Режим доступу до ресурсу: https://www.nucleodoconhecimento.com.br/wp-content/uploads/2019/03/figura-4-rede-peer-to-peer.png (дата звернення 08.05.2022).
11. BTFS Architecture Diagram [Електронний ресурс] – Режим доступу до ресурсу: https://files.readme.io/af75286-.png (дата звернення 12.05.2022).
12. BTFS2.0 Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://docs.btfs.io/docs (дата звернення 12.05.2022).
13. Pinata documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.pinata.cloud/ (дата звернення 13.05.2022).
14. Pinata cloud [Електронний ресурс] – Режим доступу до ресурсу: https://www.pinata.cloud/ (дата звернення 13.05.2022).
15. The Centralized Secret [Електронний ресурс] – Режим доступу до ресурсу: https://miro.medium.com/max/700/1*Sjeb3CJvzQwmKzIFN2S8Zw.png (дата звернення 14.05.2022).
16. Arweave web documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.arweave.org/info/ (дата звернення 15.05.2022).
17. Arweave: "Store data, permanently" [Електронний ресурс] – Режим доступу до ресурсу: https://www.arweave.org/ (дата звернення 15.05.2022).
18. Welcome to the Permaweb [Електронний ресурс] – Режим доступу до ресурсу: https://arweave.medium.com/welcome-to-the-permaweb-ce0e6c73ddfb (дата звернення 16.05.2022).
19. Network and Application Architecture [Електронний ресурс] – Режим доступу до ресурсу:

https://mirror.xyz/msfew.eth/FB2ls0jlkCV2ROOenM3ecC2zFua393fg0rYpFDikxtg (дата звернення 16.05.2022).

20. Arweave token analytics [Електронний ресурс] – Режим доступу до ресурсу: https://coinmarketcap.com/currencies/arweave/ (дата звернення 17.05.2022).
21. Blockchain Consensus Mechanisms [Електронний ресурс] – Режим доступу до ресурсу: https://miro.medium.com/fit/c/224/224/1*hxarAQqr0J-86yXdNbRukw.png (дата звернення 17.05.2022).
22. JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/ru/docs/Web/JavaScript (дата звернення 19.05.2022).
23. TypeScript is JavaScript with syntax for types [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/ (дата звернення 19.05.2022).
24. Node.js - Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm (дата звернення 19.05.2022).
25. MySQL developers documentation [Електронний ресурс] – Режим доступу до ресурсу: https://dev.mysql.com/doc/ (дата звернення 19.05.2022).
26. Hypertext Transfer Protocol (HTTP) [Електронний ресурс] – Режим доступу до ресурсу: https://www.extrahop.com/resources/protocols/http/ (дата звернення 20.05.2022).
27. The WebSocket Protocol [Електронний ресурс] – Режим доступу до ресурсу: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення 20.05.2022).
28. Asymmetric cryptography [Електронний ресурс] – Режим доступу до ресурсу: https://www.sciencedirect.com/topics/computer-science/asymmetric-cryptography (дата звернення 02.06.2022).

29. Visual Studio Code overview [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/docs/editor/whyvscode (дата звернення 21.05.2022).
30. DataGrip [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/ru-ru/datagrip/ (дата звернення 21.05.2022).
31. Docker tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://dker.ru/docs/docker-engine/docker-overview/ (дата звернення 21.05.2022).
32. Docker architecture [Електронний ресурс] – Режим доступу до ресурсу: https://dker.ru/static/images/docs/engine/article-img/architecture.svg (дата звернення 21.05.2022).

ДОДАТОК 1

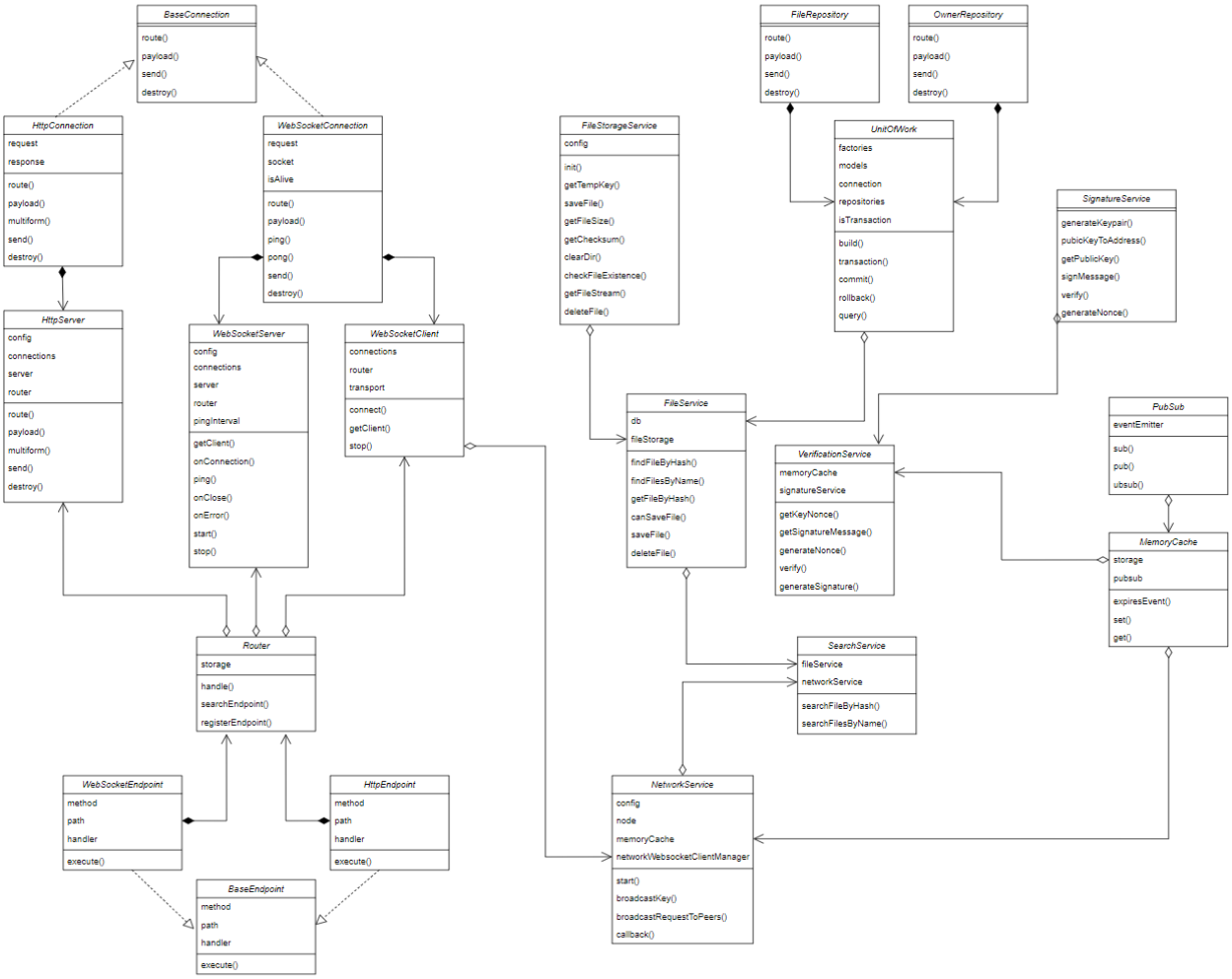
Спосіб захищеного розподіленого зберігання даних

Функціональна схема

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Розгон О. О.			**Спосіб захищеного розподіленого зберігання даних** Функціональна схема	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.				
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83
Н. Контр.		Сімоненко В. П.				
Затвердив						

ДОДАТОК 2

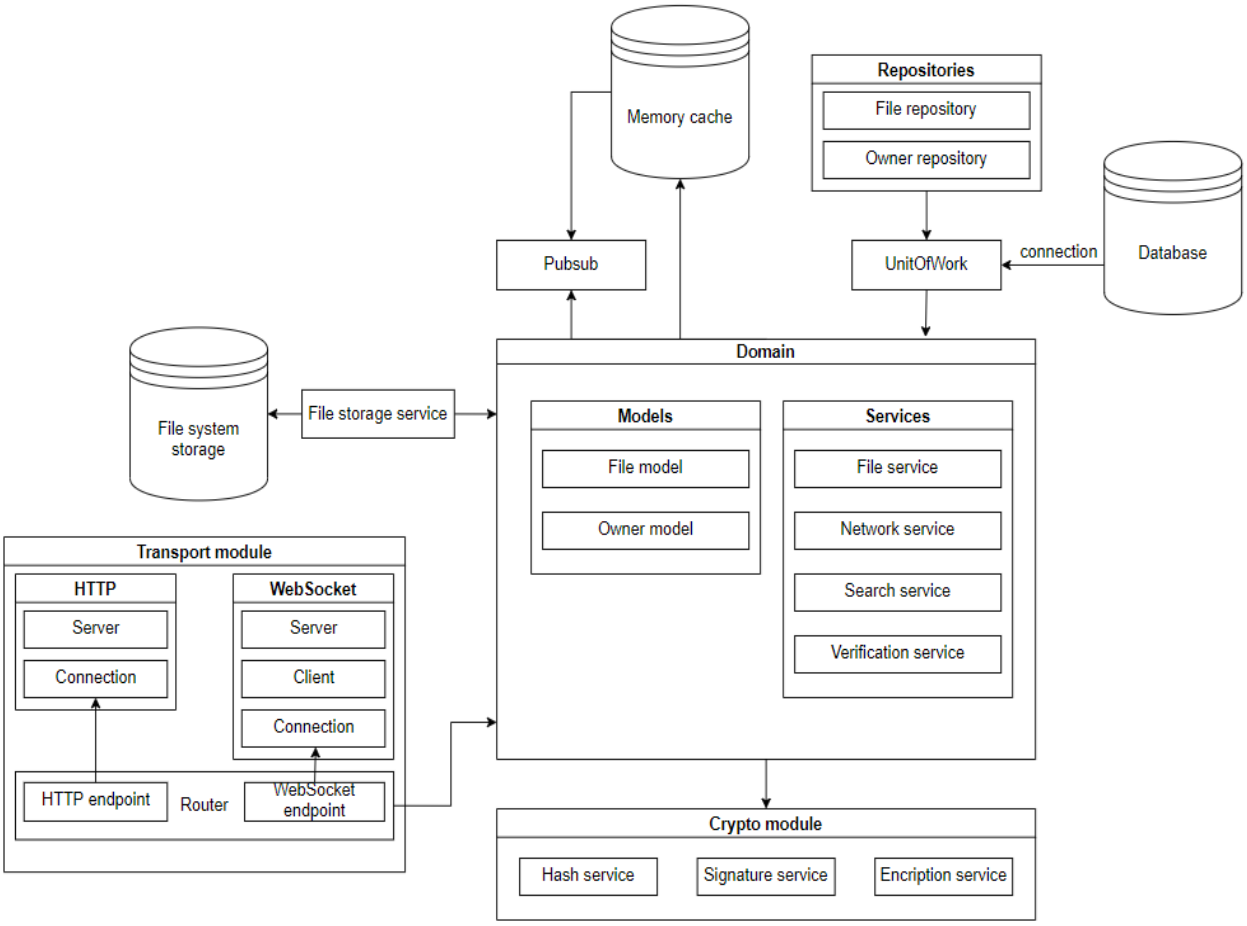
Спосіб захищеного розподіленого зберігання даних

Структурна схема системи

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Розгон О.О.			**Спосіб захищеного розподіленого зберігання даних** Структурна схема	Літ. / Аркуш: 1 / Аркушів: 1
Перевірив		Волокита А. М.				
Н. Контр.		Сімоненко В. П.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83
Затвердив						

ДОДАТОК 3

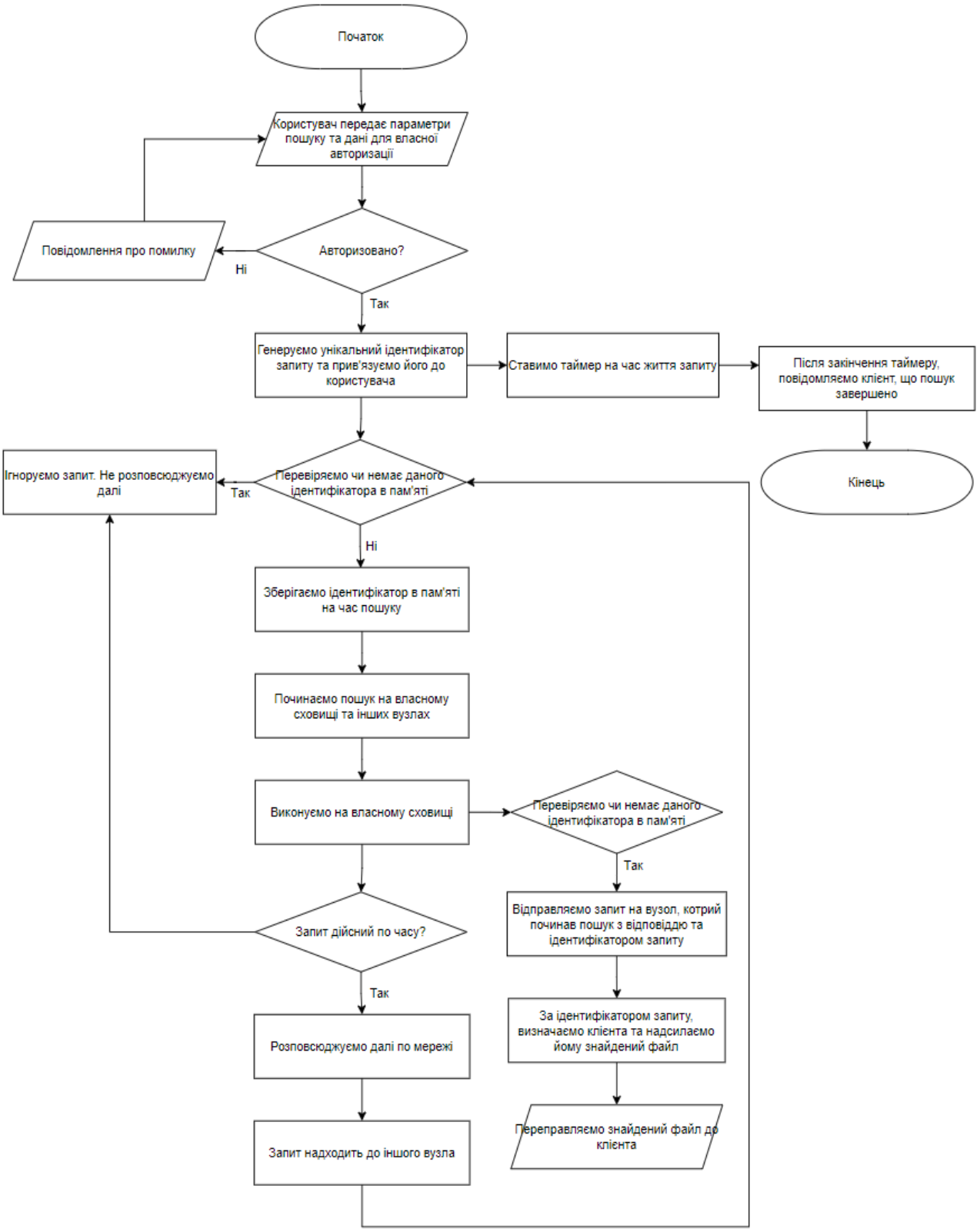
Спосіб захищеного розподіленого зберігання даних

Алгоритм пошуку даних в розподіленій системі

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Розгон О. О.			**Спосіб захищеного Розподіленого зберігання даних** Алгоритм пошуку даних в розподіленій системі	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Н. Контр.		Сімоненко В. П.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83	
Затвердив							

ДОДАТОК 4

Спосіб захищеного розподіленого зберігання даних

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 51

Київ 2022 р

```
'use strict';

const {v4: uuid} = require('uuid');
const fs = require('fs');

const {TransportType} = require('../interfaces/constants');
const {BaseServer, BaseServerError} = require('../interfaces/base.server');
const {HttpConnection} = require('./http.connection');

class HttpServerError extends BaseServerError {}

//router: {handle({connection: BaseConnection}): Promise<any>}
class HttpServer extends BaseServer {
  constructor({config, router}) {
    super({type: TransportType.HTTP});
    const isValid = HttpServer.validateConfig(config);
    if (!isValid) throw new HttpServerError('Invalid config');
    this.config = config;
    const {ssl} = config;
    const protocol = ssl ? require('https') : require('http');
    const params = ssl
      ? {
          cert: fs.readFileSync(ssl.cert),
          key: fs.readFileSync(ssl.key),
        }
      : {};
    this.connections = new Map();
    const listener = this.#listener.bind(this);
    const server = protocol.createServer(params, listener);
    this.server = server;
    this.router = router;
  }

  get clients() {
    const {connections} = this;
    return new Map(connections);
  }

  static validateConfig(config) {
    return true;
  }

  async #listener(req, res) {
    const {router, connections, transport} = this;
    const clientId = uuid();
    const connection = new HttpConnection({req, res});
    connections.set(clientId, connection);
    res.on('close', () => {
```

```
      connections.delete(clientId);
    });
    router.handle({
      connection,
      metadata: {
        transport,
        client: clientId,
      },
    });
  }

  start() {
    const {
      config: {port, address, ssl},
      server,
    } = this;
    server.listen(port, address, () => {
      console.log(`Start server ${ssl ? 'https' : 'http'}://${address}:${port}`);
    });
    return this;
  }

  async stop() {
    const {connections, server, config} = this;
    const {address, port, ssl} = config;
    for (const [clientId, connection] of connections.entries()) {
      await connection.destroy();
      connections.delete(clientId);
    }
    server.close(() => {
      console.log(`Closed server ${ssl ? 'https' : 'http'}://${address}:${port}`);
    });
    return this;
  }
}

module.exports = {
  HttpServer,
  HttpServerError,
};


'use strict';

const formidable = require('formidable');
const stream = require('stream');
const fs = require('fs');

const {BaseConnection, BaseConnectionError} = require('../interfaces/base.connection');

class HttpConnectionError extends BaseConnectionError {}
```

```
class HttpConnection extends BaseConnection {
  constructor({req, res}) {
    super();
    this.request = req;
    this.response = res;
  }

  route() {
    const {request} = this;
    const {url, method} = request;
    return {path: url, method};
  }

  async payload() {
    const {request} = this;
    const chunks = [];
    for await (const chunk of request) {
      chunks.push(chunk);
    }
    const buffer = Buffer.concat(chunks).toString();
    return buffer;
  }

  async multiform() {
    const {request} = this;
    const form = new formidable.IncomingForm({
      maxFiles: 1,
    });
    const clear = async (files) =>
      Promise.all(files.map((filepath) => fs.promises.rm(filepath))).then(() => files);
    return new Promise((res, rej) => {
      form.parse(request, (err, fields, files = {}) => {
        if (err) return rej(err);
        const filepaths = Object.keys(files)
          .map((file) => files?.[file]?.filepath)
          .filter((f) => !!f);
        res({
          files: files,
          data: fields,
          clear: clear.bind(null, filepaths),
        });
      });
    });
  }

  async send(data) {
    const {response} = this;
    response.writeHead(200);
    if (data instanceof stream.Readable) {
      return data.pipe(response);
    }
    if (typeof data === 'string') {
```

```
      return response.end(data);
    }
  }

  async destroy() {
    const {response} = this;
    return response.destroy();
  }
}
module.exports = {HttpConnection, HttpConnectionError};


'use strict';

const WS = require('ws');
const fs = require('fs');
const {v4: uuid} = require('uuid');

const {TransportType} = require('../interfaces/constants');
const {BaseServer, BaseServerError} = require('../interfaces/base.server');
const {WebSocketConnection} = require('./websocket.connection');

class WebSocketServerError extends BaseServerError {}

class WebSocketServer extends BaseServer {
  constructor({config, router}) {
    super({type: TransportType.WebSocket});
    const isValid = WebSocketServer.validateConfig(config);
    if (!isValid) throw new HttpServerError('Invalid config');
    this.config = config;
    this.router = router;
    const {ssl} = config;
    const protocol = ssl ? require('https') : require('http');
    const params = ssl
      ? {
          cert: fs.readFileSync(ssl.cert),
          key: fs.readFileSync(ssl.key),
        }
      : {};
    this.connections = new Map();
    const http = protocol.createServer(params);
    this.server = http;
    const ws = new WS.Server({
      server: http,
      clientTracking: false,
    });
    ws.on('connection', this.#onConnection.bind(this))
      .on('close', this.#onClose.bind(this))
      .on('error', this.#onError.bind(this));
    this.pingInterval = this.#ping();
  }
```

```
get clients() {
  const {connections} = this;
  return new Map(connections);
}

getClient(clientId) {
  const {connections} = this;
  const client = connections.get(clientId);
  return client;
}

async #onConnection(socket, request) {
  const {connections, router, transport} = this;
  const connection = new WebSocketConnection({socket, request});
  const clientId = uuid();
  console.log(`WS: connected: ${clientId}`);
  connections.set(clientId, connection);
  socket.on('close', (_) => {
    console.log(`WS: disconnected: ${clientId}`);
    connections.delete(clientId);
  });
  socket.on('pong', () => (connection.isAlive = true));
  socket.on('message', async (message) => {
    if (!router) return;
    const payload = JSON.parse(message);
    router.handle({connection, payload, metadata: {transport, client: clientId}});
  });
}

#ping() {
  const {connections} = this;
  const ping = () => {
    connections.forEach(async (connection) => {
      if (connection.isAlive === false) await connection.destroy();
      connection.isAlive = false;
      await connection.ping();
    });
  };
  const interval = setInterval(ping, 30000);
  return interval;
}

async #onClose() {
  const {connections, pingInterval} = this;
  clearInterval(pingInterval);
  for (const [clientId, connection] of connections.entries()) {
    await connection.destroy();
    connections.delete(clientId);
  }
  return;
}
```

```
  async #onError(err) {
    console.log(err);
    return;
  }

  static validateConfig(config) {
    return true;
  }

  async start() {
    const {config, server} = this;
    const {ssl, port, address} = config;
    server.listen(port, address, () => {
      console.log(`Start WebSocket server ${ssl ? 'wss' : 'ws'}://${address}:${port}`);
    });
    return this;
  }

  async stop() {
    const {server, config} = this;
    const {ssl, port, address} = config;
    server.close(() => {
      console.log(`WebSocket server ${ssl ? 'wss' : 'ws'}://${address}:${port} closed`);
    });
    return this;
  }
}

module.exports = {
  WebSocketServer,
  WebSocketServerError,
};



'use strict';

const {BaseConnection, BaseConnectionError} = require('../interfaces/base.connection');

class WebSocketConnectionError extends BaseConnectionError {}

class WebSocketConnection extends BaseConnection {
  constructor({socket, request}) {
    super();
    this.socket = socket;
    this.request = request;
    this.isAlive = true;
  }

  route() {}

  async payload() {}
```

```
  async send(data) {
    const {socket} = this;
    return socket.send(data);
  }

  async destroy() {
    const {socket} = this;
    return socket.terminate();
  }

  async ping() {
    const {socket} = this;
    return socket.ping();
  }

  async pong() {
    const {socket} = this;
    return socket.pong();
  }
}

module.exports = {
  WebSocketConnection,
  WebSocketConnectionError,
};



'use strict';

const WS = require('ws');
const {v4: uuid} = require('uuid');

const {TransportType} = require('../interfaces/constants');
const {WebSocketConnection} = require('./websocket.connection');

class WebSocketClientError extends Error {}

class WebSocketClient {
  constructor({router}) {
    this.router = router;
    this.transport = TransportType.WebSocket;
    this.connections = new Map();
  }

  async connect({url}) {
    const {connections, router, transport} = this;
    const socket = new WS(url);
    const connection = new WebSocketConnection({socket, request: {}});
    const serverId = uuid();
    connections.set(serverId, connection);
```

```
    //heartbeat
    let pingTimeout;
    const heartbeat = async () => {
      await connection.pong();
      clearTimeout(pingTimeout);
      pingTimeout = setTimeout(async () => {
        await connection.destroy();
      }, 31000);
    };
    socket.on('open', heartbeat);
    socket.on('ping', heartbeat);
    //
    socket.on('close', async (_) => {
      clearTimeout(pingTimeout);
      connections.delete(serverId);
    });
    socket.on('message', async (message) => {
      if (!router) return;
      const payload = JSON.parse(message);
      router.handle({connection, payload, metadata: {transport, client: serverId}});
    });
    return connection;
  }

  get clients() {
    const {connections} = this;
    return new Map(connections);
  }

  getClient(serverId) {
    const {connections} = this;
    const client = connections.get(serverId);
    return client;
  }

  async stop() {
    const {connections} = this;
    for (const [serverId, connection] of connections.entries()) {
      await connection.destroy();
      connections.delete(serverId);
    }
    return;
  }
}

module.exports = {
  WebSocketClient,
  WebSocketClientError,
};
```

```
'use strict';

const {HttpConnection} = require('../http/http.connection');
const {BaseEndpoint} = require('../interfaces/base.endpoint');
const {BaseRouterError, BaseRouter} = require('../interfaces/base.router');
const {TransportType, EndpointMethods} = require('../interfaces/constants');
const {WebSocketConnection} = require('../websocket/websocket.connection');

const DEFAULT_PATH = 'DEFAULD_PATH';

const TransportHandler = {
  [TransportType.HTTP]: (connection, payload) => {
    if (!(connection instanceof HttpConnection)) throw new RouterError('Invalid connection');
    const {method, path} = connection.route();
    return {method, path};
  },
  [TransportType.WebSocket]: (connection, payload) => {
    if (!(connection instanceof WebSocketConnection)) throw new RouterError('Invalid connection');
    const {event} = payload;
    return {
      method: EndpointMethods.WS,
      path: event,
    };
  },
};

class RouterError extends BaseRouterError {}

class Router extends BaseRouter {
  constructor() {
    super();
    this.storage = {};
  }

  async handle({connection, payload, metadata}) {
    const {transport} = metadata;
    const {method, path} = TransportHandler[transport](connection, payload);
    const endpoint = this.#searchEndpoint(method, path);
    if (!endpoint) return;
    const context = {payload, metadata};
    endpoint.execute({connection, context});
  }

  #searchEndpoint(method, path) {
    const {storage} = this;
    const methodStorage = storage[method];
    const endpoint = methodStorage?.get(path);
    //TODO: fix default endpoint
    const defaultEndpoint = {execute: () => {}};
    return endpoint ?? methodStorage?.get(DEFAULT_PATH) ?? defaultEndpoint;
  }
```

```
  registerEndpoint(endpoint) {
    const {storage} = this;
    if (!(endpoint instanceof BaseEndpoint)) throw new RouterError('Invalid endpoint');
    const {path, method} = endpoint;
    const methodStorage = storage[method] ?? new Map();
    methodStorage.set(path, endpoint);
    storage[method] = methodStorage;
    return this;
  }
}

module.exports = {
  Router,
  RouterError,
  DEFAULT_PATH,
};


'use strict';

const {BaseEndpointError, BaseEndpoint} = require('../interfaces/base.endpoint');
const {EndpointMethods} = require('../interfaces/constants');

class WebSocketEndpointError extends BaseEndpointError {}

class WebSocketEndpoint extends BaseEndpoint {
  constructor({path, handler}) {
    super({method: EndpointMethods.WS, path, handler});
  }
}

module.exports = {
  WebSocketEndpoint,
  WebSocketEndpointError,
};



'use strict';

const {BaseEndpointError, BaseEndpoint} = require('../interfaces/base.endpoint');
const {EndpointMethods} = require('../interfaces/constants');

const SUPPORTED_METHODS = [EndpointMethods.GET, EndpointMethods.POST];

class HttpEndpointError extends BaseEndpointError {}

class HttpEndpoint extends BaseEndpoint {
  constructor({method, path, handler}) {
    if (!SUPPORTED_METHODS.includes(method)) throw new HttpEndpointError('Method not
supported');
    super({method, path, handler});
```

```
  }
}

module.exports = {
  HttpEndpoint,
  HttpEndpointError,
};


'use strict';

const mysql = require('mysql2');
const migration = require('mysql-migrations');
const path = require('path');

const {DatabaseConfig} = require('../../../../config');

const connection = mysql.createPool(DatabaseConfig);

const connectionPromise = connection.promise();

migration.init(connection, path.join(__dirname, '../../../../../migrations'), function () {
  console.log('finished running migrations');
});

module.exports = {
  connection: connectionPromise,
};


'use strict';

class UnitOfWorkError extends Error {}

class UnitOfWork {
  #isTransaction = false;
  #models;
  constructor({models, connection, transaction = false}) {
    this.factories = models;
    this.connection = connection;
    const builtModels = this.#build(models, connection);
    this.#models = builtModels;
    this.#isTransaction = transaction;
  }

  get repositories() {
    const {connection} = this;
    if (!connection) throw new UnitOfWorkError('Connection released');
    return {...this.#models};
  }

  #build(models, connection) {
```

```
  const builtModels = Object.keys(models).reduce((builtModels, name) => {
    const modelFactory = models[name];
    const model = modelFactory(connection);
    return Object.assign(builtModels, {[name]: model});
  }, {});
  return builtModels;
}

async transaction() {
  const {connection, factories} = this;
  if (!connection) throw new UnitOfWorkError('Connection released');
  if (this.#isTransaction) throw new UnitOfWorkError('Transaction has already been created');
  const transaction = await connection.getConnection();
  await transaction.beginTransaction();
  return new UnitOfWork({
    models: factories,
    connection: transaction,
    transaction: true,
  });
}

async commit() {
  const {connection} = this;
  if (!connection) throw new UnitOfWorkError('Connection released');
  if (!this.#isTransaction) throw new UnitOfWorkError('Transaction not created');
  await connection.commit();
  return this;
}

async rollback() {
  const {connection} = this;
  if (!connection) throw new UnitOfWorkError('Connection released');
  if (!this.#isTransaction) throw new UnitOfWorkError('Transaction not created');
  await connection.rollback();
  return this;
}

release() {
  const {connection} = this;
  if (!connection) throw new UnitOfWorkError('Connection released');
  if (!this.#isTransaction) throw new UnitOfWorkError('Transaction not created');
  this.connection = null;
  this.factories = null;
  connection.release();
  return null;
}

query(query, ...args) {
  const {connection} = this;
  if (!connection) throw new UnitOfWorkError('Connection released');
  return connection.query(query, args);
}
```

```
}

module.exports = {
  UnitOfWork,
  UnitOfWorkError,
};



'use strict';

const {v4: uuid} = require('uuid');
const fs = require('fs');
const crypto = require('crypto');
const path = require('path');

class FileStorageError extends Error {}

class FileStorage {
  constructor({config}) {
    this.config = config;
    this.#init();
  }

  #init() {
    const {tempStorage, persistentStorage} = this;
    try {
      const dirs = [tempStorage, persistentStorage];
      dirs.map((dir) => {
        if (!fs.existsSync(dir)) {
          fs.mkdirSync(dir, {recursive: true});
        }
      });
      return true;
    } catch (err) {
      console.log(err);
      throw new FileStorageError('Init file storage error');
    }
  }

  static getTempKey() {
    return uuid();
  }

  get tempStorage() {
    const {config} = this;
    return path.join(config.storage, '/temp');
  }

  get persistentStorage() {
    const {config} = this;
    return path.join(config.storage, '/persistent');
```

```
}

async saveFile({checksum, readFileStream}) {
  const {tempStorage, persistentStorage} = this;
  try {
    const fileTempKey = FileStorage.getTempKey();
    const tempPath = path.join(tempStorage, `/${fileTempKey}`);
    const writeStream = fs.createWriteStream(tempPath);
    await new Promise((res, rej) => {
      writeStream.on('finish', res).on('error', rej);
      readFileStream.pipe(writeStream);
    });
    const fileChecksum = await this.getChecksum({filepath: tempPath});
    if (fileChecksum !== checksum) {
      await fs.promises.rm(tempPath);
      throw new FileStorageError('Checksum not equal');
    }
    const persistentPath = path.join(persistentStorage, `/${fileChecksum}`);
    await fs.promises.rename(tempPath, persistentPath);
    const fileSize = await this.getFileSize({filepath: persistentPath});
    return {
      size: fileSize,
      path: persistentPath,
      checksum: fileChecksum,
    };
  } catch (err) {
    console.log(err);
    throw new FileStorageError('Save file error');
  }
}

async getFileSize({filepath}) {
  try {
    const stats = await fs.promises.stat(filepath);
    const sizeBytes = stats.size;
    const sizeMb = sizeBytes / Math.pow(1024, 1); //KB
    return sizeMb;
  } catch (err) {
    console.log(err);
    throw new FileStorageError();
  }
}

async getChecksum({filepath}) {
  try {
    const hasher = crypto.createHash('md5');
    hasher.setEncoding('base64url');
    const hash = await new Promise((res, rej) => {
      const fileStream = fs.createReadStream(filepath);
      fileStream
        .on('end', () => {
          hasher.end();
```

```
          const hash = hasher.read();
          res(hash);
        })
        .on('error', (err) => rej(err));
      fileStream.pipe(hasher);
    });
    return hash;
  } catch (err) {
    if (err instanceof FileStorageError) throw err;
    console.log(err);
    throw new FileStorageError('Get checksum error');
  }
}

async clearDir({dirpath}) {
  try {
    await fs.promises.rm(dirpath, {recursive: true, force: true});
    await fs.promises.mkdir(dirpath, {recursive: true});
    return true;
  } catch (err) {
    console.log(err);
    throw new FileStorageError(`Clear dir ${dirpath} error`);
  }
}

async checkFileExistence({filepath}) {
  try {
    const exist = await fs.promises.access(filepath);
    return true;
  } catch (err) {
    return false;
  }
}

async getFileStream({hash}) {
  const {persistentStorage} = this;
  try {
    const filepath = path.join(persistentStorage, `/${hash}`);
    const exist = await this.checkFileExistence({filepath});
    if (!exist) return null;
    const stream = fs.createReadStream(filepath);
    return stream;
  } catch (err) {
    console.log(err);
    throw new FileStorageError('Get file stream error');
  }
}

async deleteFile({hash}) {
  const {persistentStorage} = this;
  try {
    const filepath = path.join(persistentStorage, `/${hash}`);
```

```
      const exist = await this.checkFileExistence({filepath});
      if (!exist) return;
      await fs.promises.rm(filepath);
      return;
    } catch (err) {
      console.log(err);
      throw new FileStorageError('Delete file error');
    }
  }
}

module.exports = {
  FileStorage,
  FileStorageError,
};


'use strict';

const crypto = require('crypto');

class EncryptionService {
  constructor() {}

  encrypt(str, key) {
    const iv = crypto.randomBytes(12);
    const cipher = crypto.createCipheriv('aes-256-gcm', Buffer.from(key, 'base64'), iv);
    const enc = cipher.update(str, 'utf-8');
    const enc2 = cipher.final();
    return Buffer.concat([enc, enc2, iv, cipher.getAuthTag()]).toString('base64');
  }

  decrypt(data, key) {
    const enc = Buffer.from(data, 'base64');
    const iv = enc.slice(enc.length - 28, enc.length - 16);
    const tag = enc.slice(enc.length - 16);
    const payload = enc.slice(0, enc.length - 28);
    const decipher = crypto.createDecipheriv('aes-256-gcm', Buffer.from(key, 'base64'), iv);
    decipher.setAuthTag(tag);
    const message = decipher.update(payload, undefined, 'utf-8') + decipher.final('utf-8');
    return message;
  }
}

module.exports = {
  EncryptionService,
};


'use strict';

const bcrypt = require('bcrypt');
```

```
class HashService {
  constructor() {}

  async hash(data, rounds = 10) {
    const salt = await bcrypt.genSalt(rounds);
    const hash = await bcrypt.hash(data, salt);
    return hash;
  }

  async compare(data, hash) {
    const compared = await bcrypt.compare(data, hash);
    return compared;
  }
}

module.exports = {
  HashService,
};


'use strict';

const crypto = require('crypto');
const ed25519 = require('@noble/ed25519');
const {uint8toHex, hexToUint8} = require('./helpers.js');

class SignatureService {
  constructor() {}

  async generateKeyPair() {
    const privateKey = ed25519.utils.randomPrivateKey();
    const publicKey = await ed25519.getPublicKey(privateKey);
    return {
      privateKey: uint8toHex(privateKey),
      publicKey: uint8toHex(publicKey),
    };
  }

  publicKeyToAddress(publicKey) {
    const address = crypto.createHash('md5').update(publicKey).digest('hex');
    return address;
  }

  async getPublicKey(privateKey) {
    const publicKey = await ed25519.getPublicKey(privateKey);
    return uint8toHex(publicKey);
  }

  async signMessage(message, privateKey) {
    const hexMessage = Buffer.from(message).toString('hex');
    const signature = await ed25519.sign(hexMessage, privateKey);
```

```
    return uint8toHex(signature);
  }

  async verify(message, signature, publicKey) {
    const hexMessage = Buffer.from(message).toString('hex');
    const verified = await ed25519.verify(signature, hexMessage, publicKey);
    return verified;
  }

  generateNonce() {
    const nonce = crypto.randomBytes(16).toString('hex');
    return nonce;
  }
}

module.exports = {
  SignatureService,
};


'use strict';

const {FileModel} = require('../models/file.model');
const {FileAccessType, OwnerModel} = require('../models/owner.model');
const {FindFilesByNameDTO} = require('./dto/file.dto');
const {FindFileByHashDTO, SaveFileDTO} = require('./dto/file.dto');

class FileServiceError extends Error {}

class FileService {
  constructor({db, fileStorage}) {
    this.db = db;
    this.fileStorage = fileStorage;
  }

  async findFileByHash(findFileByHashDTO) {
    const {
      db: {unitOfWork},
    } = this;
    if (!(findFileByHashDTO instanceof FindFileByHashDTO))
      throw new FileServiceError('Invalid find file dto');
    const {hash, address} = findFileByHashDTO;
    const {FileRepository} = unitOfWork.repositories;
    try {
      const fileModel = await FileRepository.findOne({hash}, false);
      if (!fileModel) return null;
      const {owners, ...file} = fileModel;
      const accesses = owners.filter((owner) => {
        const {accessType, address: ownerAddress} = owner;
        const access = accessType !== FileAccessType.HIDDEN || address === ownerAddress;
        return access;
      });
```

```
    if (!accesses.length) return null;
    return FileModel.fromRaw({...file, owners: accesses});
  } catch (err) {
    console.log(err);
    throw new FileServiceError('Fild file by hash error');
  }
}

async findFilesByName(findFilesByNameDTO) {
  const {
    db: {unitOfWork},
  } = this;
  if (!(findFilesByNameDTO instanceof FindFilesByNameDTO))
    throw new FileServiceError('Invalid find file dto');
  const {name, address} = findFilesByNameDTO;
  const {FileRepository} = unitOfWork.repositories;
  try {
    const fileModels = await FileRepository.find({name}, false);
    const accessesFileModels = fileModels
      .map((fileModel) => {
        const {owners, ...file} = fileModel;
        const accesses = owners.filter((owner) => {
          const {accessType, address: ownerAddress} = owner;
          const access = accessType !== FileAccessType.HIDDEN || address === ownerAddress;
          return access;
        });
        if (!accesses.length) return false;
        return FileModel.fromRaw({...file, owners: accesses});
      })
      .filter((fileModel) => !!fileModel);
    return accessesFileModels;
  } catch (err) {
    console.log(err);
    throw new FileServiceError('Find file by name error');
  }
}

async getFileByHash(findFileByHashDTO) {
  const {fileStorage} = this;
  if (!(findFileByHashDTO instanceof FindFileByHashDTO))
    throw new FileServiceError('Invalid find file dto');
  const {hash, address} = findFileByHashDTO;
  try {
    const fileModel = await this.findFileByHash(findFileByHashDTO);
    if (!fileModel) return null;
    const {owners, ...file} = fileModel;
    const accesses = owners.filter((owner) => {
      const {accessType, address: ownerAddress} = owner;
      const access = accessType !== FileAccessType.CLOSED || ownerAddress === address;
      return access;
    });
    if (!accesses.length) return null;
```

```
    const fileStream = await fileStorage.getFileStream({hash});
    return fileStream;
  } catch (err) {
    console.log(err);
    throw new FileServiceError('Get file by hash error');
  }
}

async canSaveFile() {
  try {
    //TODO
    return true;
  } catch (err) {
    console.log(err);
    throw new FileServiceError('Can save file error');
  }
}

async saveFile(saveFileDTO) {
  const {
    db: {unitOfWork},
    fileStorage,
  } = this;
  if (!(saveFileDTO instanceof SaveFileDTO)) throw new FileServiceError('Invalid save file dto');
  const canSave = await this.canSaveFile(saveFileDTO);
  if (!canSave) throw new FileServiceError('Can not save file');
  const {readFileStream, checksum, address, metadata, accessType, ttl, name} = saveFileDTO;
  const transaction = await unitOfWork.transaction();
  try {
    const {FileRepository} = transaction.repositories;
    const ownerModel = OwnerModel.fromRaw({
      accessType,
      address,
      metadata,
      ttl,
    });
    const fileModel = FileModel.fromRaw({
      name,
      hash: checksum,
      size: 0,
      owners: [ownerModel],
    });
    const {size} = await fileStorage.saveFile({checksum, readFileStream});
    const newFileModel = await FileRepository.create({...fileModel, size});
    await transaction.commit();
    return newFileModel;
  } catch (err) {
    await transaction.rollback();
    console.log(err);
    throw new FileServiceError('Save file error');
  } finally {
    transaction.release();
```

```
    }
  }

  async deleteFile(findFileByHashDTO) {
    const {
      fileStorage,
      db: {unitOfWork},
    } = this;
    if (!(findFileByHashDTO instanceof FindFileByHashDTO))
      throw new FileServiceError('Invalid find file dto');
    const {hash, address} = findFileByHashDTO;
    const transaction = await unitOfWork.transaction();
    try {
      const {OwnerRepository, FileRepository} = transaction.repositories;
      const {owners} = await FileRepository.findOne({hash}, false);
      const accesses = owners.filter((owner) => {
        const {address: ownerAddress} = owner;
        const access = address === ownerAddress;
        return access;
      });
      await Promise.all(accesses.map((access) => OwnerRepository.delete(access)));

      if (owners.length > accesses.length) {
        await transaction.commit();
        return;
      }
      await FileRepository.delete({hash});
      await fileStorage.deleteFile({hash});
      await transaction.commit();
      return;
    } catch (err) {
      await transaction.rollback();
      console.log(err);
      throw new FileServiceError('Delete file error');
    } finally {
      transaction.release();
    }
  }
}

module.exports = {
  FileService,
  FileServiceError,
};



'use strict';

const {default: axios} = require('axios');
const {RequestDTO} = require('./dto/network.dto');
const {NodeInfoDTO} = require('./dto/node-info.dto');
```

```
class NetworkServiceError extends Error {}

class NetworkService {
  constructor({config, node, networkWebsocketClientManager, memoryCache}) {
    this.config = config;
    this.node = node;
    this.memoryCache = memoryCache;
    this.networkWebsocketClientManager = networkWebsocketClientManager;
  }

  async start() {
    const {config, networkWebsocketClientManager} = this;
    const {
      ws: {peers},
    } = config;
    await Promise.all(peers.map((peer) => networkWebsocketClientManager.connect({url: peer})));
    return this;
  }

  static broadcastKey(requestId) {
    return `broadcast_to_peers:${requestId}`;
  }

  async broadcastRequestToPeers(path, requestDTO, payload) {
    const {networkWebsocketClientManager, memoryCache} = this;
    if (!(requestDTO instanceof RequestDTO)) throw new NetworkServiceError('Invalid request');
    const {requestId, expirationTime} = requestDTO;
    try {
      const currentTime = Date.now();
      const timeDiff = expirationTime - currentTime;
      if (timeDiff <= 0) return;
      const key = NetworkService.broadcastKey(requestDTO);
      const isCyclicalReq = await memoryCache.get(key);
      if (!!isCyclicalReq) return;
      await memoryCache.set(key, requestId, timeDiff / 1000);
      const peers = networkWebsocketClientManager?.clients ?? new Map();
      const broadcastData = JSON.stringify({
        event: path,
        data: {
          request: requestDTO,
          payload: payload,
        },
      });
      await Promise.allSettled([...peers.values()].map((peer) => peer.send(broadcastData)));
      return;
    } catch (err) {
      console.log(err);
      throw new NetworkServiceError('Broadcast to peers error');
    }
  }
```

```
  async callback(requestDTO, payload) {
    const {node} = this;
    try {
      const nodeInfo = NodeInfoDTO.fromRaw({...node});
      const {requestId, callbackUrl} = requestDTO;
      const {data} = await axios.post(callbackUrl, {payload, requestId, nodeInfo});
      return data;
    } catch (err) {
      console.log(err);
      throw new NetworkServiceError('Send callback error');
    }
  }
}

module.exports = {
  NetworkService,
  NetworkServiceError,
};



'use strict';

const {SearchFilesByNameDTO} = require('./dto/search.dto');
const {SearchFileByHashDTO} = require('./dto/search.dto');

class SearchServiceError extends Error {}

class SearchService {
  constructor({fileService, networkService}) {
    this.fileService = fileService;
    this.networkService = networkService;
  }

  async searchFileByHash(searchFileByHashDTO) {
    const {fileService, networkService} = this;
    if (!(searchFileByHashDTO instanceof SearchFileByHashDTO))
      throw new SearchServiceError('Invalid searh file dto');
    const {request, payload} = searchFileByHashDTO;
    try {
      const fileModel = await fileService.findFileByHash(payload);
      if (fileModel)
        await networkService.callback(request, fileModel).catch((err) => console.log(err));
      await networkService.broadcastRequestToPeers('FIND_FILE_BY_HASH', request, payload);
      return fileModel;
    } catch (err) {
      console.log(err);
      throw new SearchServiceError('Search file error');
    }
  }

  async searchFilesByName(searchFilesByNameDTO) {
```

```
    const {fileService, networkService} = this;
    if (!(searchFilesByNameDTO instanceof SearchFilesByNameDTO))
      throw new SearchServiceError('Invalid searh file dto');
    const {request, payload} = searchFilesByNameDTO;
    try {
      const fileModels = await fileService.findFilesByName(payload);
      await Promise.allSettled(
        fileModels.map((fileModel) =>
          networkService.callback(request, fileModel).catch((err) => console.log(err)),
        ),
      );
      await networkService.broadcastRequestToPeers('FIND_FILES_BY_NAME', request, payload);
      return fileModels;
    } catch (err) {
      console.log(err);
      throw new SearchServiceError('Search file error');
    }
  }
}

module.exports = {
  SearchService,
  SearchServiceError,
};



'use strict';

class VerificationServiceError extends Error {}

class VerificationService {
  constructor({memoryCache, signatureService}) {
    this.memoryCache = memoryCache;
    this.signatureService = signatureService;
  }

  static getKeyNonce(address) {
    return `temp_nonce:${address}`;
  }

  static getSignatureMessage(message, nonce) {
    return `${message}:${nonce}`;
  }

  async generateNonce({address}) {
    const {memoryCache, signatureService} = this;
    try {
      const nonce = signatureService.generateNonce();
      const key = VerificationService.getKeyNonce(address);
      await memoryCache.set(key, nonce, 60); //1 min expiry
      return nonce;
```

```
    } catch (err) {
      console.log(err);
      throw new VerificationServiceError('Generate nonce error');
    }
  }

  async verify(signatureDTO) {
    const {memoryCache, signatureService} = this;
    try {
      const {message, nonce, signature, publicKey} = signatureDTO;
      const address = signatureService.publicKeyToAddress(publicKey);
      const key = VerificationService.getKeyNonce(address);
      const cacheNonce = await memoryCache.get(key);
      if (cacheNonce !== nonce) throw new VerificationServiceError('Invalid nonce');
      const signatureMessage = VerificationService.getSignatureMessage(message, nonce);
      const verified = signatureService.verify(signatureMessage, signature, publicKey);
      return verified ? address : null;
    } catch (err) {
      if (err instanceof VerificationServiceError) throw err;
      console.log(err);
      throw new VerificationServiceError('Verify message error');
    }
  }

  async generateSignature({message, nonce, privateKey}) {
    const {signatureService} = this;
    try {
      const signatureMessage = VerificationService.getSignatureMessage(message, nonce);
      const signature = signatureService.signMessage(signatureMessage, privateKey);
      return signature;
    } catch (err) {
      console.log(err);
      throw new VerificationServiceError('Generate signature error');
    }
  }
}

module.exports = {
  VerificationService,
  VerificationServiceError,
};



'use strict';

const {notEmptyValue} = require('../helpers/common');
const {OwnerModel, OwnerRepository} = require('./owner.model');

class FileModelError extends Error {}

class FileModel {
```

```
  constructor(raw) {
    const {id, name, hash, size, createdAt, owners} = raw;
    this.id = id; //number(int)
    this.name = name; //string
    this.hash = hash; //string
    this.size = size; //number(int)
    this.owners = owners; //OwnerModel[]
    this.createdAt = createdAt; //string
  }

  static fromRaw(raw) {
    FileModel.validate(raw);
    const model = new FileModel(raw);
    return model;
  }

  static validate(raw) {
    const {name, hash, size, owners} = raw;
    if (!notEmptyValue(name)) throw new FileModelError('Invalid file name');
    if (!notEmptyValue(hash)) throw new FileModelError('Invalid file hash');
    if (!Number.isInteger(size)) throw new FileModelError('Invalid file size');
    if (!!owners?.length && !owners.some((owner) => owner instanceof OwnerModel))
      throw new FileModelError('Invalid file owners');
    return true;
  }
}

const FileRepository = (connection) => {
  const ownerRepository = OwnerRepository(connection);
  return {
    findOne: async ({hash}, lazy) => {
      const getFileQuery = 'SELECT * FROM `files` WHERE `hash` = ?;';
      const getFileParams = [hash];
      const fileRaw = await connection
        .query(getFileQuery, getFileParams)
        .then(([[fileRaw]]) => fileRaw);
      if (!fileRaw) return null;
      const owners = lazy ? null : await ownerRepository.find({fileId: fileRaw.id});
      const fileModel = FileModel.fromRaw({...fileRaw, owners});
      return fileModel;
    },

    create: async function (fileModel) {
      const {owners = [], ...file} = fileModel;
      const {name, hash, size} = file;
      const createFileQuery =
        'INSERT INTO `files` (`name`, `hash`, `size`) VALUE (?, ?, ?)\n' +
        'ON DUPLICATE KEY UPDATE `id` = `id`;';
      const createFileParams = [name, hash, size];
      await connection.query(createFileQuery, createFileParams);
      const newFileModel = await this.findOne({hash}, true);
      await Promise.all(
```

```
      owners.map((owner) => ownerRepository.create({...owner, fileId: newFileModel.id})),
    );
    return newFileModel;
  },

  totalFileSize: async () => {
    const getTotalSizeQuery = 'SELECT SUM(`size`) as `totalSize` FROM `files`;';
    const getTotalSizeParams = [];
    const totalSize = await connection
      .query(getTotalSizeQuery, getTotalSizeParams)
      .then(([[result]]) => result?.totalSize ?? 0);
    return totalSize;
  },

  delete: async function ({hash}) {
    const fileModel = await this.findOne({hash}, false);
    if (!fileModel) throw new FileModelError('Can not delete file');
    const deleteFileQuery = 'DELETE FROM `files` WHERE `hash` = ?;';
    const deleteFileParams = [hash];
    await connection.query(deleteFileQuery, deleteFileParams);
    return fileModel;
  },

  find: async ({name}, lazy) => {
    const ownerRepository = OwnerRepository(connection);
    const searchFilesQuery = "SELECT * FROM `files` WHERE `name` LIKE CONCAT('%', ?,
'%')";
    const parameters = [name];
    const fileModels = await connection.query(searchFilesQuery, parameters).then(([fileRaws]) =>
      Promise.all(
        fileRaws.map(async (fileRaw) => {
          const owners = lazy ? null : await ownerRepository.find({fileId: fileRaw.id});
          const fileModel = FileModel.fromRaw({...fileRaw, owners});
          return fileModel;
        }),
      ),
    );
    return fileModels;
  },
 };
};

module.exports = {
 FileModel,
 FileRepository,
};



'use strict';

const {notEmptyValue} = require('../helpers/common');
```

```
const FileAccessType = {
  OPEN: 'OPEN',
  CLOSED: 'CLOSED',
  HIDDEN: 'HIDDEN',
};

class OwnerModelError extends Error {}

class OwnerModel {
  constructor(raw) {
    const {id, ttl, accessType, address, metadata, fileId, createdAt} = raw;
    this.id = id;
    this.accessType = accessType; //FileAccessType(string)
    this.address = address; //string
    this.metadata = metadata; //json -> object
    this.ttl = ttl; //number(int) days
    this.fileId = fileId; //number(int)
    this.createdAt = createdAt;
  }

  static fromRaw(raw) {
    OwnerModel.validate(raw);
    const model = new OwnerModel(raw);
    return model;
  }

  static validate(raw) {
    const {ttl, accessType, address, metadata} = raw;
    if (!Number.isInteger(ttl) || ttl <= 0) throw new OwnerModelError('Invalid file ttl');
    if (!Object.values(FileAccessType).includes(accessType))
      throw new OwnerModelError('Invalid file access type');
    if (!notEmptyValue(address)) throw new OwnerModelError('Invalid file address');
    if (!!metadata && typeof metadata !== 'object')
      throw new OwnerModelError('Invalid file metadata');
  }
}

const OwnerRepository = (connection) => ({
  create: async (ownerModel) => {
    const {ttl, accessType, address, metadata = {}, fileId} = ownerModel;
    const createOwnerQuery =
      'INSERT INTO `owners` (`ttl`, `accessType`, `address`, `metadata`, `fileId`) VALUE
(?, ?, ?, ?, ?)\n' +
      'ON DUPLICATE KEY UPDATE `id` = `id`;';
    const createOwnerParams = [ttl, accessType, address, JSON.stringify(metadata), fileId];
    await connection.query(createOwnerQuery, createOwnerParams);
  },

  find: async ({fileId}) => {
    const getOwnersQuery = 'SELECT * FROM `owners` WHERE `fileId` = ?;';
    const getOwnersParams = [fileId];
```

```
    const ownerRaws = await connection
      .query(getOwnersQuery, getOwnersParams)
      .then(([ownerRaws]) => ownerRaws);
    const owners = ownerRaws.map((ownerRaw) => OwnerModel.fromRaw(ownerRaw));
    return owners;
  },

  delete: async ({id}) => {
    const deleteOwnerQuery = 'DELETE FROM `owners` WHERE `id` = ?;';
    const deleteOwnerParams = [id];
    await connection.query(deleteOwnerQuery, deleteOwnerParams);
  },
});

module.exports = {
  OwnerModel,
  OwnerModelError,
  OwnerRepository,
  FileAccessType,
};


'use strict';

class MemoryCacheError extends Error {}

class MemoryCache {
  constructor({pubsub = null}) {
    this.storage = new Map();
    this.pubsub = pubsub;
  }

  static expiresEvent = (key) => `@key_expires:${key}`;

  set(key, value, expiry = false) {
    const {storage, pubsub} = this;
    if (!!expiry) {
      const timeout = parseInt(expiry) * 1000; //sec
      if (Number.isNaN(timeout)) throw new MemoryCacheError('Invalid expiry');
      setTimeout(async () => {
        storage.delete(key);
        if (!pubsub) return;
        const event = MemoryCache.expiresEvent(key);
        pubsub.pub(event, value);
      }, timeout);
    }
    storage.set(key, value);
    return true;
  }

  get(key) {
    const {storage} = this;
```

```
    const value = storage.get(key);
    return value;
  }
}

module.exports = {
  MemoryCache,
  MemoryCacheError,
};


'use strict';

const {EventEmitter} = require('events');

class PubSubError extends Error {}

class PubSub {
  constructor() {
    this.ee = new EventEmitter();
  }

  async sub(key, listener) {
    const {ee} = this;
    ee.on(key, listener);
    return true;
  }

  async pub(key, value) {
    const {ee} = this;
    ee.emit(key, value);
    return true;
  }

  async unsub(key, listener) {
    const {ee} = this;
    ee.off(key, listener);
    return true;
  }
}

module.exports = {
  PubSub,
  PubSubError,
};


'use strict';

const {SignatureDTO} = require('../../../services/dto/signature.dto');

const verifivationGuard = async (container, signature, address) => {
```

```
  const {verificationService} = container;
  if (!(signature instanceof SignatureDTO))
    throw new Error('Signature not instanceof SignatureDTO');
  try {
    const verifiedAddress = await verificationService.verify(signature);
    if (!verifiedAddress) throw new Error('Not verified');
    if (address !== verifiedAddress) throw new Error('Address not verified');
    return address;
  } catch (err) {
    console.log(err);
    throw new Error('Verify guard error');
  }
};

module.exports = {
  verifivationGuard,
};


'use strict';

const {EndpointMethods} = require('../../../infrastructure/transport/interfaces/constants');
const {HttpEndpoint} = require('../../../infrastructure/transport/router/http.endpoint');
const {FindFileByHashDTO} = require('../../../services/dto/file.dto');
const {SignatureDTO} = require('../../../services/dto/signature.dto');
const {verifivationGuard} = require('../guards/verification.guard');

const deleteFileController = async (container, {connection, context}) => {
  const {fileService} = container;
  try {
    const payload = JSON.parse(await connection.payload());
    const {data, signature} = payload;
    const findFileByHashDTO = FindFileByHashDTO.fromRaw(data);
    //verification
    const message = findFileByHashDTO.toMessage();
    const signatureDTO = SignatureDTO.fromRaw({...signature, message});
    await verifivationGuard(container, signatureDTO, findFileByHashDTO.address);
    //
    await fileService.deleteFile(findFileByHashDTO);
    await connection.send(
      JSON.stringify({
        success: true,
      }),
    );
    return;
  } catch (err) {
    console.log(err);
    await connection.send(
      JSON.stringify({
        success: false,
        message: err?.message ?? '',
      }),
```

```
    );
  }
};

module.exports = (container) =>
  new HttpEndpoint({
    method: EndpointMethods.POST,
    path: '/client/delete-file',
    handler: deleteFileController.bind(null, container),
  });


'use strict';

const {EndpointMethods} = require('../../../infrastructure/transport/interfaces/constants');
const {HttpEndpoint} = require('../../../infrastructure/transport/router/http.endpoint');
const {FindFileByHashDTO} = require('../../../services/dto/file.dto');
const {SignatureDTO} = require('../../../services/dto/signature.dto');
const {verifivationGuard} = require('../guards/verification.guard');

const getFileController = async (container, {connection, context}) => {
  const {fileService} = container;
  try {
    const payload = JSON.parse(await connection.payload());
    const {data, signature} = payload;
    const getFileByHashDTO = FindFileByHashDTO.fromRaw(data);
    //verification
    const message = getFileByHashDTO.toMessage();
    const signatureDTO = SignatureDTO.fromRaw({...signature, message});
    await verifivationGuard(container, signatureDTO, getFileByHashDTO.address);
    //
    const fileStream = await fileService.getFileByHash(getFileByHashDTO);
    if (!fileStream) throw new Error('File not found or not available');
    await connection.send(fileStream);
    return;
  } catch (err) {
    console.log(err);
    await connection.send(
      JSON.stringify({
        success: false,
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new HttpEndpoint({
    method: EndpointMethods.POST,
    path: '/client/get-file',
    handler: getFileController.bind(null, container),
  });
```

```
'use strict';

const {EndpointMethods} = require('../../../infrastructure/transport/interfaces/constants');
const {HttpEndpoint} = require('../../../infrastructure/transport/router/http.endpoint');

const getTempNonce = async (container, {connection, context}) => {
  const {verificationService, signatureService} = container;
  try {
    const payload = JSON.parse(await connection.payload());
    const {
      data: {publicKey},
    } = payload;
    const address = signatureService.publicKeyToAddress(publicKey);
    const nonce = await verificationService.generateNonce({address});
    await connection.send(
      JSON.stringify({
        success: true,
        result: {nonce},
      }),
    );
    return;
  } catch (err) {
    console.log(err);
    await connection.send(
      JSON.stringify({
        success: false,
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new HttpEndpoint({
    method: EndpointMethods.POST,
    path: '/client/get-temp-nonce',
    handler: getTempNonce.bind(null, container),
  });


'use strict';

const fs = require('fs');
const {EndpointMethods} = require('../../../infrastructure/transport/interfaces/constants');
const {HttpEndpoint} = require('../../../infrastructure/transport/router/http.endpoint');
const {SaveFileDTO} = require('../../../services/dto/file.dto');
const {SignatureDTO} = require('../../../services/dto/signature.dto');
const {verifivationGuard} = require('../guards/verification.guard');

const uploadFileController = async (container, {connection, context}) => {
```

```
  const {fileService} = container;
  const {files, data: body, clear} = await connection.multiform();
  try {
    const {
      file: {filepath},
    } = files;
    const payload = JSON.parse(body.payload);
    const {data, signature} = payload;
    const readFileStream = fs.createReadStream(filepath, 'utf-8');
    const saveFileDTO = SaveFileDTO.fromRaw({
      ...data,
      readFileStream,
    });
    //verification
    const message = saveFileDTO.toMessage();
    const signatureDTO = SignatureDTO.fromRaw({...signature, message});
    await verifivationGuard(container, signatureDTO, saveFileDTO.address);
    //
    const fileModel = await fileService.saveFile(saveFileDTO);
    await connection.send(
      JSON.stringify({
        success: true,
        result: fileModel,
      }),
    );
    return;
  } catch (err) {
    console.log(err);
    await connection.send(
      JSON.stringify({
        success: false,
        message: err?.message ?? '',
      }),
    );
  } finally {
    const filepaths = await clear();
    console.log(`Cleaned temp files: ${JSON.stringify(filepaths)}`);
  }
};

module.exports = (container) =>
  new HttpEndpoint({
    method: EndpointMethods.POST,
    path: '/client/upload-file',
    handler: uploadFileController.bind(null, container),
  });


'use strict';

const getHttpControllers = (container) => ({
  getFileController: require('./get-file.controller')(container),
```

```
  uploadFileController: require('./upload-file.controller')(container),
  getTempNonce: require('./get-nonce.controller')(container),
  deleteFileController: require('./delete-file.controller')(container),
});

module.exports = getHttpControllers;


'use strict';

const {v4: uuid} = require('uuid');
const {WebSocketEndpoint} = require('../../../infrastructure/transport/router/websocket.endpoint');
const {FindFileByHashDTO} = require('../../../services/dto/file.dto');
const {RequestDTO} = require('../../../services/dto/network.dto');
const {SearchFileByHashDTO} = require('../../../services/dto/search.dto');
const {SignatureDTO} = require('../../../services/dto/signature.dto');
const {MemoryCache} = require('../../../system/memory-cache');
const {verifivationGuard} = require('../guards/verification.guard');

const FIND_FILE_TIME = 5 * 60 * 1000; //default 5 min

const findFileController = async (container, {connection, context}) => {
  const {NetworkConfig, searchService, memoryCache, pubsub} = container;
  try {
    const {
      payload: {payload},
      metadata: {client},
    } = context;
    const {signature, data, timeout} = JSON.parse(payload);
    //create dto and validation
    const findFileByHashDTO = FindFileByHashDTO.fromRaw(data);
    //verifivation
    const message = findFileByHashDTO.toMessage();
    const signatureDTO = SignatureDTO.fromRaw({...signature, message});
    await verifivationGuard(container, signatureDTO, findFileByHashDTO.address);
    //
    const requestDTO = RequestDTO.fromRaw({
      requestId: uuid(),
      expirationTime: Date.now() + (timeout ?? FIND_FILE_TIME),
      callbackUrl: `${NetworkConfig.http.callback}/node/file-found`,
    });
    const searchFileByHashDTO = SearchFileByHashDTO.fromRaw({
      request: requestDTO,
      payload: findFileByHashDTO,
    });
    //requestId - client set to cache
    await memoryCache.set(requestDTO.requestId, client, (timeout ?? FIND_FILE_TIME) / 1000);
    const expiresEvent = MemoryCache.expiresEvent(requestDTO.requestId);
    //on search finished
    await pubsub.sub(expiresEvent, async function onExpires() {
      await connection.send(
        JSON.stringify({
```

```
          event: 'FILE_SEARCH_FINISHED',
          payload: {
            hash: findFileByHashDTO.hash,
            requestId: requestDTO.requestId,
          },
        }),
      );
      await pubsub.unsub(expiresEvent, onExpires);
    });
    //start search
    await searchService.searchFileByHash(searchFileByHashDTO);
    //send search data
    await connection.send(
      JSON.stringify({
        event: 'FILE_SEARCH_STARTED',
        payload: {
          hash: findFileByHashDTO.hash,
          requestId: requestDTO.requestId,
          expirationTime: requestDTO.expirationTime,
        },
      }),
    );
    return;
  } catch (err) {
    console.log(err);
    connection.send(
      JSON.stringify({
        event: 'ERROR',
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new WebSocketEndpoint({
    path: 'FIND_FILE_BY_HASH',
    handler: findFileController.bind(null, container),
  });


'use strict';

const {v4: uuid} = require('uuid');
const {WebSocketEndpoint} = require('../../../infrastructure/transport/router/websocket.endpoint');
const {FindFilesByNameDTO} = require('../../../services/dto/file.dto');
const {RequestDTO} = require('../../../services/dto/network.dto');
const {SearchFilesByNameDTO} = require('../../../services/dto/search.dto');
const {SignatureDTO} = require('../../../services/dto/signature.dto');
const {MemoryCache} = require('../../../system/memory-cache');
const {verifivationGuard} = require('../guards/verification.guard');
```

```
const FIND_FILE_TIME = 5 * 60 * 1000; //default 5 min

const findFilesController = async (container, {connection, context}) => {
  const {NetworkConfig, searchService, memoryCache, pubsub} = container;
  try {
    const {
      payload: {payload},
      metadata: {client},
    } = context;
    const {signature, data, timeout} = JSON.parse(payload);
    //create dto and validation
    const findFilesByNameDTO = FindFilesByNameDTO.fromRaw(data);
    //verifivation
    const message = findFilesByNameDTO.toMessage();
    const signatureDTO = SignatureDTO.fromRaw({...signature, message});
    await verifivationGuard(container, signatureDTO, findFilesByNameDTO.address);
    //
    const requestDTO = RequestDTO.fromRaw({
      requestId: uuid(),
      expirationTime: Date.now() + (timeout ?? FIND_FILE_TIME),
      callbackUrl: `${NetworkConfig.http.callback}/node/file-found`,
    });
    const searchFilesByNameDTO = SearchFilesByNameDTO.fromRaw({
      request: requestDTO,
      payload: findFilesByNameDTO,
    });
    //requestId - client set to cache
    await memoryCache.set(requestDTO.requestId, client, (timeout ?? FIND_FILE_TIME) / 1000);
    const expiresEvent = MemoryCache.expiresEvent(requestDTO.requestId);
    //on search finished
    await pubsub.sub(expiresEvent, async function onExpires() {
      await connection.send(
        JSON.stringify({
          event: 'FILE_SEARCH_FINISHED',
          payload: {
            name: findFilesByNameDTO.name,
            requestId: requestDTO.requestId,
          },
        }),
      );
      await pubsub.unsub(expiresEvent, onExpires);
    });
    //start search
    await searchService.searchFilesByName(searchFilesByNameDTO);
    //send search data
    await connection.send(
      JSON.stringify({
        event: 'FILE_SEARCH_STARTED',
        payload: {
          name: findFilesByNameDTO.name,
          requestId: requestDTO.requestId,
          expirationTime: requestDTO.expirationTime,
```

```
        },
      }),
    );
    return;
  } catch (err) {
    console.log(err);
    connection.send(
      JSON.stringify({
        event: 'ERROR',
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new WebSocketEndpoint({
    path: 'FIND_FILES_BY_NAME',
    handler: findFilesController.bind(null, container),
  });


'use strict';

const getWebSocketControllers = (container) => ({
  findFileByHashController: require('./find-file-by-hash.controller')(container),
  findFilesByNameController: require('./find-files-by-name.controller')(container),
});

module.exports = getWebSocketControllers;


'use strict';

const getWebSocketControllers = require('./ws');
const getHttpControllers = require('./http');

const useWebSocketClientApi = (container) => {
  const {findFileByHashController, findFilesByNameController} =
getWebSocketControllers(container);
  const {apiRouter} = container;
  apiRouter.registerEndpoint(findFileByHashController);
  apiRouter.registerEndpoint(findFilesByNameController);
};

const useHttpClientApi = (container) => {
  const {getFileController, uploadFileController, getTempNonce, deleteFileController} =
    getHttpControllers(container);
  const {apiRouter} = container;
  apiRouter.registerEndpoint(getTempNonce);
  apiRouter.registerEndpoint(getFileController);
  apiRouter.registerEndpoint(uploadFileController);
```

```
  apiRouter.registerEndpoint(deleteFileController);
};

module.exports = {
  useHttpClientApi,
  useWebSocketClientApi,
};


'use strict';

const {EndpointMethods} = require('../../../infrastructure/transport/interfaces/constants');
const {HttpEndpoint} = require('../../../infrastructure/transport/router/http.endpoint');
const {SearchFileResponseDTO} = require('../../../services/dto/search.dto');

const foundFileCallbackController = async (container, {connection, context}) => {
  const {memoryCache, apiWebsocketServer} = container;
  try {
    const payload = JSON.parse(await connection.payload());
    const {requestId, payload: file, nodeInfo} = SearchFileResponseDTO.fromRaw(payload);
    const clientId = await memoryCache.get(requestId);
    if (clientId) {
      const socket = apiWebsocketServer.getClient(clientId);
      if (socket) {
        await socket.send(
          JSON.stringify({
            event: 'FILE_FOUND',
            payload: {
              file,
              node: nodeInfo,
            },
          }),
        );
      }
    }
    await connection.send(JSON.stringify({success: true}));
    return;
  } catch (err) {
    console.log(err);
    await connection.send(
      JSON.stringify({
        success: false,
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new HttpEndpoint({
    method: EndpointMethods.POST,
    path: '/node/file-found',
```

```
    handler: foundFileCallbackController.bind(null, container),
  });


'use strict';

const getHttpControllers = (container) => ({
  foundFileCallbackController: require('./found-file.controller')(container),
});

module.exports = getHttpControllers;



const {WebSocketEndpoint} = require('../../../infrastructure/transport/router/websocket.endpoint');
const {FindFileByHashDTO} = require('../../../services/dto/file.dto');
const {RequestDTO} = require('../../../services/dto/network.dto');
const {SearchFileByHashDTO} = require('../../../services/dto/search.dto');

const findFileByHashController = async (container, {connection, context}) => {
  const {searchService} = container;
  try {
    const {
      payload: {data},
    } = context;
    const {request, payload} = data;
    const findFileByHashDTO = FindFileByHashDTO.fromRaw(payload);
    const requestDto = RequestDTO.fromRaw(request);
    const searchFileByHashDTO = SearchFileByHashDTO.fromRaw({
      request: requestDto,
      payload: findFileByHashDTO,
    });
    await searchService.searchFileByHash(searchFileByHashDTO);
    return;
  } catch (err) {
    console.log(err);
    connection.send(
      JSON.stringify({
        event: 'ERROR',
        request: context.payload.data.request,
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new WebSocketEndpoint({
    path: 'FIND_FILE_BY_NAME',
    handler: findFileByHashController.bind(null, container),
  });
```

```
const {WebSocketEndpoint} = require('../../../infrastructure/transport/router/websocket.endpoint');
const {FindFilesByNameDTO} = require('../../../services/dto/file.dto');
const {RequestDTO} = require('../../../services/dto/network.dto');
const {SearchFilesByNameDTO} = require('../../../services/dto/search.dto');

const findFilesByNameController = async (container, {connection, context}) => {
  const {searchService} = container;
  try {
    const {
      payload: {data},
    } = context;
    const {request, payload} = data;
    const findFilesByNameDTO = FindFilesByNameDTO.fromRaw(payload);
    const requestDto = RequestDTO.fromRaw(request);
    const searchFilesByHashDTO = SearchFilesByNameDTO.fromRaw({
      request: requestDto,
      payload: findFilesByNameDTO,
    });
    await searchService.searchFilesByName(searchFilesByHashDTO);
    return;
  } catch (err) {
    console.log(err);
    connection.send(
      JSON.stringify({
        event: 'ERROR',
        request: context.payload.data.request,
        message: err?.message ?? '',
      }),
    );
  }
};

module.exports = (container) =>
  new WebSocketEndpoint({
    path: 'FIND_FILES_BY_NAME',
    handler: findFilesByNameController.bind(null, container),
  });


'use strict';

const getWebSocketController = (container) => ({
  findFileByHashController: require('./find-file-hash.controller')(container),
  findFilesByNameController: require('./find-files-name.controller')(container),
});

module.exports = getWebSocketController;


'use strict';
```

```
const getHttpControllers = require('./http');
const getWebSocketController = require('./ws');

const useHttpNodeApi = (container) => {
  const {foundFileCallbackController} = getHttpControllers(container);
  const {networkRouter} = container;
  networkRouter.registerEndpoint(foundFileCallbackController);
};

const useWebSocketNodeApi = (container) => {
  const {findFileByHashController, findFilesByNameController} =
getWebSocketController(container);
  const {networkRouter} = container;
  networkRouter.registerEndpoint(findFileByHashController);
  networkRouter.registerEndpoint(findFilesByNameController);
};

module.exports = {
  useHttpNodeApi,
  useWebSocketNodeApi,
};
```

```
'use strict';

//configs
const {NetworkConfig, FileStorageConfig, ApiConfig, NodeInfo} = require('./src/config');

//crypto
const {EncryptionService} = require('./src/infrastructure/crypto/encryption.service');
const {HashService} = require('./src/infrastructure/crypto/hash.service');
const {SignatureService} = require('./src/infrastructure/crypto/signature.service');

const signatureService = new SignatureService();
const hashService = new HashService();
const encryptionService = new EncryptionService();

//transport
const {WebSocketClient} = require('./src/infrastructure/transport/websocket/websocket.client');
const {WebSocketServer} = require('./src/infrastructure/transport/websocket/websocket.server');
const {HttpServer} = require('./src/infrastructure/transport/http/http.server');
const {Router} = require('./src/infrastructure/transport/router/router');

//system
const {MemoryCache} = require('./src/system/memory-cache');
const {PubSub} = require('./src/system/pub-sub');
const pubsub = new PubSub();
const memoryCache = new MemoryCache({pubsub});

//storage
const {FileStorage} = require('./src/infrastructure/storage/file-system');
```

```
const fileStorage = new FileStorage({config: FileStorageConfig});

const {
  connection: dbConnection,
} = require('./src/infrastructure/storage/database/connections/main.connection');
const {UnitOfWork} = require('./src/infrastructure/storage/database/unit-of-work');
const models = require('./src/models');
const unitOfWork = new UnitOfWork({models, connection: dbConnection});
const db = {unitOfWork};

//network
const networkRouter = new Router();
const networkWebsocketServer = new WebSocketServer({
  router: networkRouter,
  config: NetworkConfig.ws,
});
const networkHttpServer = new HttpServer({router: networkRouter, config: NetworkConfig.http});
const networkWebsocketClientManager = new WebSocketClient({router: networkRouter});

//access
const {VerificationService} = require('./src/services/verification.service');
const verificationService = new VerificationService({memoryCache, signatureService});

//api
const apiRouter = new Router();
const apiWebsocketServer = new WebSocketServer({
  router: apiRouter,
  config: ApiConfig.ws,
});
const apiHttpServer = new HttpServer({router: apiRouter, config: ApiConfig.http});

//services
const {NetworkService} = require('./src/services/network.service');
const networkService = new NetworkService({
  config: NetworkConfig,
  node: NodeInfo,
  networkWebsocketClientManager,
  memoryCache,
});
const {FileService} = require('./src/services/file.service');
const fileService = new FileService({db, fileStorage});
const {SearchService} = require('./src/services/search.service');
const searchService = new SearchService({fileService, networkService});

const container = {
  //configs
  NetworkConfig,
  //system
  pubsub,
  memoryCache,
  //storage
  db,
```

```
  fileStorage,
  //crypto
  signatureService,
  //network
  networkRouter,
  networkHttpServer,
  networkWebsocketServer,
  networkWebsocketClientManager,
  //access
  verificationService,
  //api
  apiRouter,
  apiHttpServer,
  apiWebsocketServer,
  //services
  fileService,
  networkService,
  searchService,

  start: () =>
    Promise.all([
      networkHttpServer.start(),
      networkWebsocketServer.start(),
      networkService.start(),
      apiHttpServer.start(),
      apiWebsocketServer.start(),
    ]),

  stop: () =>
    Promise.allSettled([
      networkHttpServer.stop(),
      networkWebsocketServer.stop(),
      networkWebsocketClientManager.stop(),
      apiHttpServer.stop(),
      apiWebsocketServer.stop(),
    ]),
};

module.exports = container;



'use strict';

const {useHttpNodeApi, useWebSocketNodeApi} = require('./src/api/node');
const {useHttpClientApi, useWebSocketClientApi} = require('./src/api/client');
const container = require('./container');

(async () => {
  //register node api
  useHttpNodeApi(container);
  useWebSocketNodeApi(container);
```

```
  //register client api
  useHttpClientApi(container);
  useWebSocketClientApi(container);
  //start application
  await container.start();
  //graceful shutdown appilication
  const shutdown = async () => {
    await container.stop();
    process.exit(0);
  };
  process.on('SIGINT', shutdown);
  process.on('SIGTERM', shutdown);
})();



'use strict';

const FormData = require('form-data');
const fetch = require('node-fetch');

class HttpApiClientError extends Error {}

const DEFAULT_ENDPOINT = 'http://localhost:5001';

class HttpApiClient {
  constructor({signatureService}) {
    this.signatureService = signatureService;
    this.endpoint = DEFAULT_ENDPOINT;
  }

  setEndpoint(endpoint) {
    this.endpoint = endpoint;
    return this;
  }

  async uploadFile({name, ttl, checksum, metadata = {}, accessType, file, secret}) {
    const {signatureService, endpoint} = this;
    try {
      const publicKey = await signatureService.getPublicKey(secret);
      const address = signatureService.publicKeyToAddress(publicKey);
      const nonce = await this.getNonce({publicKey});
      const message = `${name}:${checksum}:${nonce}`;
      const signature = await signatureService.signMessage(message, secret);
      const formData = new FormData();
      formData.append('file', file);
      formData.append(
        'payload',
        JSON.stringify({
          signature: {signature, publicKey, nonce},
          data: {
            ttl,
```

```
          name,
          checksum,
          metadata,
          accessType,
          address,
        },
      }),
    );
    const result = await fetch(`${endpoint}/client/upload-file`, {
      method: 'POST',
      body: formData,
    });
    const {success, result: fileResult} = await result.json();
    if (!success) throw new HttpApiClientError('Unsuccess result: upload file');
    return fileResult;
  } catch (err) {
    if (err instanceof HttpApiClientError) throw err;
    console.log(err);
    throw new HttpApiClientError('Upload file error');
  }
}

async getNonce({publicKey}) {
  const {endpoint} = this;
  try {
    const data = {publicKey};
    const result = await fetch(`${endpoint}/client/get-temp-nonce`, {
      method: 'POST',
      body: JSON.stringify({data}),
    });
    const response = await result.json();
    if (!response?.success) throw new HttpApiClientError('Unsuccess result: get nonce');
    const {
      result: {nonce},
    } = response;
    return nonce;
  } catch (err) {
    if (err instanceof HttpApiClientError) throw err;
    console.log(err);
    throw new HttpApiClientError('Get nonce error');
  }
}

async getFile({checksum, secret}) {
  const {endpoint, signatureService} = this;
  try {
    const publicKey = await signatureService.getPublicKey(secret);
    const address = signatureService.publicKeyToAddress(publicKey);
    const nonce = await this.getNonce({publicKey});
    const message = `${checksum}:${nonce}`;
    const signature = await signatureService.signMessage(message, secret);
    const payload = {
```

```
      data: {
        hash: checksum,
        address: address,
      },
      signature: {
        nonce,
        publicKey,
        signature,
      },
    };
    const result = await fetch(`${endpoint}/client/get-file`, {
      method: 'POST',
      body: JSON.stringify(payload),
    });
    return result.body;
  } catch (err) {
    if (err instanceof HttpApiClientError) throw err;
    console.log(err);
    throw new HttpApiClientError('Get file error');
  }
}

async deleteFile({checksum, secret}) {
  const {endpoint, signatureService} = this;
  try {
    const publicKey = await signatureService.getPublicKey(secret);
    const address = signatureService.publicKeyToAddress(publicKey);
    const nonce = await this.getNonce({publicKey});
    const message = `${checksum}:${nonce}`;
    const signature = await signatureService.signMessage(message, secret);
    const payload = {
      data: {
        hash: checksum,
        address: address,
      },
      signature: {
        nonce,
        publicKey,
        signature,
      },
    };
    const result = await fetch(`${endpoint}/client/delete-file`, {
      method: 'POST',
      body: JSON.stringify(payload),
    });
    const {success} = await result.json();
    return success;
  } catch (err) {
    console.log(err);
    throw new HttpApiClientError('Delete file error');
  }
}
```

```
}

module.exports = {
  HttpApiClient,
  HttpApiClientError,
};


'use strict';

const WS = require('ws');

const WS_EVENTS = {
  FILE_SEARCH_STARTED: 'FILE_SEARCH_STARTED',
  FILE_SEARCH_FINISHED: 'FILE_SEARCH_FINISHED',
  FILE_FOUND: 'FILE_FOUND',
  FIND_FILE_BY_HASH: 'FIND_FILE_BY_HASH',
  FIND_FILES_BY_NAME: 'FIND_FILES_BY_NAME',
  ERROR: 'ERROR',
};

class WsApiClientError extends Error {}

class WsApiClient {
  constructor({signatureService, httpApiClient}) {
    this.signatureService = signatureService;
    this.httpApiClient = httpApiClient;
    this.node = null;
    this.socket = null;
    this.connected = false;
    this.router = new Map();
  }

  subscribe(event, listener) {
    const {router} = this;
    const listeners = router.get(event) ?? new Set();
    listeners.add(listener);
    router.set(event, listeners);
    return this;
  }

  unsubscribe(event, listener) {
    const {router} = this;
    const listeners = router.get(event) ?? new Set();
    const filtered = [...listeners.values()].filter((l) => l !== listener);
    router.set(event, new Set(filtered));
    return this;
  }

  async #route(event, payload, message) {
    const {router} = this;
    const listeners = router.get(event) ?? new Set();
```

```
  const handled = [...listeners.values()].map((listener) => listener(payload, message, this));
  await Promise.allSettled(handled);
  return;
 }

 connect(node) {
  try {
   const socket = new WS(node);
   let pingTimeout;
   const heartbeat = () => {
    socket.pong();
    clearTimeout(pingTimeout);
    pingTimeout = setTimeout(async () => {
     await socket.destroy();
    }, 31000);
   };
   socket.on('open', heartbeat);
   socket.on('ping', heartbeat);
   //
   socket.on('close', async (_) => {
    clearTimeout(pingTimeout);
    this.disconnect();
   });
   //
   socket.on('message', async (message) => {
    const data = JSON.parse(message);
    const {event, payload, message: mes} = data;
    await this.#route(event, payload, mes);
   });
   //
   this.socket = socket;
   this.node = node;
   this.connected = true;
   //
  } catch (err) {
   console.log(err);
   throw new WsApiClientError('Connect ws error');
  }
 }

 disconnect() {
  try {
   this.node = null;
   this.socket = null;
   this.connected = false;
   return this;
  } catch (err) {
   console.log(err);
   throw new WsApiClientError('Disconnect ws error');
  }
 }
```

```
async send(event, payload) {
  const {connected, socket} = this;
  if (!connected) throw new WsApiClientError('Ws socket not connected');
  try {
    const request = JSON.stringify({event, payload});
    return socket.send(request);
  } catch (err) {
    console.log(err);
    throw new WsApiClientError('Send request error');
  }
}

async searchFileByHashRequest({hash, secret, timeout}) {
  const {signatureService, httpApiClient} = this;
  try {
    const publicKey = await signatureService.getPublicKey(secret);
    const address = signatureService.publicKeyToAddress(publicKey);
    const nonce = await httpApiClient.getNonce({publicKey});
    const message = `${hash}:${nonce}`;
    const signature = await signatureService.signMessage(message, secret);
    const payload = JSON.stringify({
      signature: {signature, publicKey, nonce},
      data: {hash, address},
      timeout: timeout,
    });
    await this.send(WS_EVENTS.FIND_FILE_BY_HASH, payload);
    return;
  } catch (err) {
    console.log(err);
    throw new WsApiClientError('Search file by hash request error');
  }
}

async searchFilesByNameRequest({name, secret, timeout}) {
  const {signatureService, httpApiClient} = this;
  try {
    const publicKey = await signatureService.getPublicKey(secret);
    const address = signatureService.publicKeyToAddress(publicKey);
    const nonce = await httpApiClient.getNonce({publicKey});
    const message = `${name}:${nonce}`;
    const signature = await signatureService.signMessage(message, secret);
    const payload = JSON.stringify({
      signature: {signature, publicKey, nonce},
      data: {name, address},
      timeout: timeout,
    });
    await this.send(WS_EVENTS.FIND_FILES_BY_NAME, payload);
    return;
  } catch (err) {
    console.log(err);
    throw new WsApiClientError('Search file by hash request error');
  }
```

```
  }
}

module.exports = {
  WsApiClient,
  WsApiClientError,
  WS_EVENTS,
};
```