Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем
Назва теми

КвРКІ 210238.21.02.22 ПЗ
Шифр

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Виконав: студент IV курсу, група КІ2-21-2 _____ Владислав КІНЗЕРСЬКИЙ
Підпис, Ініціали, прізвище

Керівник _____ Сергій ЛИСЕНКО
Підпис, дата, Ініціали, прізвище

Нормоконтролер _____ Тетяна КИСІЛЬ
Підпис, дата, Ініціали, прізвище

До захисту допускаю:
зав. кафедри комп'ютерної інженерії та інформаційних систем _____ Ольга ПАВЛОВА
Підпис, Ініціали, прізвище

«12» червня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Освітній рівень БАКАЛАВР

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри Ольга ПАВЛОВА

" 10 " 01 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Владислав КІНЗЕРСЬКОМУ
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем
Керівник проекту (роботи) Лисенко Сергій Миколайович, д.т.н., проф.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 07.02.2025 р. № 23

2. Строк подання студентом проекту (роботи) на кафедру 01.06.2025 р.

3. Вихідні дані до проекту (роботи) Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Архітектурні та концептуальні засади побудови файлових систем на основі fuse

Проєктування файлової системи на основі fuse для кластерної реплікації

Реалізація та експериментальне дослідження

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Архітектура ПЗ проєкту

Архітектура ПЗ для кіберфізичної системи

Апаратне забезпечення проєкту

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Тетяна КИСІЛЬ, доцент кафедри КІІС		
Антиплагіат	Андрій НІЧЕПОРУК, доцент кафедри КІІС		

7. Дата видачі завдання «_10_»_____01_____2025 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2025	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2025	виконано
3	Робота над розділом 1 – архітектурні та концептуальні засади побудови файлових систем на основі fuse	01.03.2025	виконано
4	Робота над розділом 2 – проєктування файлової системи на основі fuse для кластерної реплікації	01.04.2025	виконано
5	Робота над розділом 3 – реалізація та експериментальне дослідження	29.04.2025	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2025	виконано
7	Попередній захист ВКР	26.05.2025	виконано
8	Захист ВКР на засіданні ЕК	Червень 2025 року	

Студент _____ Владислав КІНЗЕРСЬКИЙ
Підпис Ініціали, прізвище

Керівник роботи _____ Сергій ЛИСЕНКО
Підпис Ініціали, прізвище

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 210238.21.02.22 ПЗ	Пояснювальна записка	81		
			Графічні матеріали			
2		КвРКІ 210238.21.02.22 Е8	Плата Raspberry Pi 4 Model	1		
			B			
3		КвРКІ 210238.21.02.22 Е8	Функціональна схема	1		
4		КвРКІ 210238.21.02.22 Е8	Схема роботи програмної	1		
			частини проєкту			

КвРКІ 210238.21.02.22 ВП

Зм	Арк	№ докум	Підпис	Дата
Розробив		Кінзерський		
Перевір.		Лисенко		
Н. контр.		Кисіль		20.06.25
Затв.		Павлова		20.06.25

Відомість проекту

Літера	Аркуш	Аркушів
У	1	1

ХНУ, КІ2-22-1

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем».

Автор роботи: Владислав КІНЗЕРСЬКИЙ.

Керівник роботи: Лисенко Сергій Миколайович

Пояснювальна записка: 81 с., 27 рис., 3 дод., 53 джерела.

Графічна частина: 3 креслення.

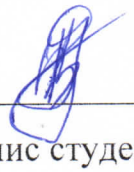
FUSE, ФАЙЛОВА СИСТЕМА, КЛАСТЕРНА ІНФРАСТРУКТУРА, КІБЕРФІЗИЧНА СИСТЕМА, РЕПЛІКАЦІЯ БАЗ ДАНИХ.

Метою дипломної роботи є розробка та дослідження файлової системи, створеної на основі технології FUSE, для забезпечення реплікації баз даних у кластері комп'ютерних систем.

Об’єктом дослідження є процеси зберігання та синхронізації даних у кластерних обчислювальних середовищах.

Предметом дослідження є механізми реалізації реплікації баз даних за допомогою файлових систем, побудованих з використанням технології FUSE.

Під час проведення даного дослідження був використаний метод систематичного огляду літератури для вивчення і аналізу предметної області даного дослідження з текстових джерел інформації.



Підпис студента

30.05.2025

Дата

ЗМІСТ

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій усе більш актуальною стає потреба у гнучких та високопродуктивних рішеннях для зберігання та обробки даних, особливо в контексті кластерних і розподілених обчислювальних систем. Зі зростанням обсягів інформації, яку обробляють сучасні застосунки від хмарних сервісів до складних наукових симуляцій, виникає необхідність у побудові таких файлових систем, які здатні забезпечити не лише швидкий доступ до даних, але й гарантії цілісності, доступності та узгодженості при одночасній роботі на великій кількості вузлів. У цьому контексті дедалі більшої уваги набуває технологія Filesystem in Userspace (FUSE), яка дозволяє створювати користувацькі файлові системи без потреби втручання у ядро операційної системи.

FUSE відкриває широкі можливості для розробників, які бажають реалізувати нестандартну логіку зберігання та доступу до даних. Особливо цікавою є перспектива використання цієї технології для задач реплікації баз даних у кластерному середовищі, де відмова одного вузла не повинна призводити до втрати даних чи порушення доступу, а синхронізація змін між копіями баз має відбуватись ефективно, без затримок і конфліктів. Така архітектура потребує глибокого розуміння як мережевих механізмів взаємодії, так і принципів організації файлових структур у розподілених системах.

Побудова файлової системи на основі FUSE дозволяє абстрагуватися від традиційних обмежень класичних систем керувань баз даних і експериментувати з новими способами реплікації через файли, об’єкти, буфери або журнали. У поєднанні з кластерною інфраструктурою така система може забезпечити високий рівень масштабованості та відмовостійкості, а також адаптуватися до зміни умов роботи наприклад, динамічної появи чи зникнення вузлів. Це робить її перспективним рішенням як для корпоративних середовищ, так і для

експериментальних або наукових проєктів, які потребують обробки великих обсягів інформації з високими вимогами до надійності.

Основною метою даної роботи є дослідження та розробка користувацької файлової системи з використанням технології FUSE, яка буде орієнтована на підтримку реплікації баз даних у кластері комп'ютерних систем. У процесі дослідження планується проаналізувати архітектурні особливості FUSE, оцінити її переваги та обмеження в контексті кластерних технологій, а також розробити прототип файлової системи, який зможе забезпечити синхронізацію даних між вузлами в реальному часі.

Ключовим завданням є не лише створення працездатного технічного рішення, а й глибоке розуміння проблем узгодженості, розподілу навантаження, відмовостійкості та продуктивності в системах з великою кількістю точок зберігання. Аналіз можливостей FUSE у поєднанні з кластерною інфраструктурою дозволить визначити її придатність для реального використання у задачах, що вимагають безперервної доступності даних, мінімальних затримок при реплікації та простоти інтеграції з існуючими сервісами.

Ціллю є продемонструвати ефективність застосування технології FUSE як основи для реалізації розподіленої файлової системи з підтримкою реплікації баз даних, що дозволяє покращити загальну надійність та масштабованість інформаційної інфраструктури. Таким чином, дана робота закладає підґрунтя для подальших досліджень у галузі розподілених файлових систем та відкриває нові можливості для оптимізації процесів збереження та доступу до інформації у великомасштабних обчислювальних середовищах.

1 АРХІТЕКТУРНІ ТА КОНЦЕПТУАЛЬНІ ЗАСАДИ ПОБУДОВИ ФАЙЛОВИХ СИСТЕМ НА ОСНОВІ FUSE

1.1 FUSE та файлові системи користувача

Файлова система є ключовим компонентом будь-якої операційної системи, оскільки вона забезпечує базову функціональність зберігання та доступу до даних. Основне її призначення полягає у впорядкуванні, зберіганні, найменуванні та керуванні файлами і каталогами на фізичних або віртуальних носіях інформації.

Кожна файлова система визначає структуру даних, необхідну для зберігання файлів, підтримує механізми навігації по файловій ієрархії, слідкує за вільним простором, а також реалізує права доступу і забезпечує надійність збереження інформації. Традиційно файлові системи реалізуються в просторі ядра операційної системи, що забезпечує їм високу продуктивність та прямий доступ до апаратного забезпечення. Прикладами таких систем є: ext4 для Linux, NTFS для Windows, APFS для macOS.

Однак реалізація файлових систем у ядрі має низку вагомих обмежень.

По-перше, розробка такого типу програмного забезпечення потребує глибокого розуміння внутрішньої архітектури ядра та специфіки його API, що значно ускладнює процес створення нових рішень. Помилки в коді файлової системи, яка працює в просторі ядра, можуть призвести до серйозних наслідків, зокрема до збоїв у роботі всієї операційної системи. Крім того, відсутність ізоляції між елементами ядра створює додаткові ризики з точки зору безпеки, збій або зловмисна активність у файловій системі може порушити стабільність та цілісність усієї ОС.

Ще однією проблемою є складність у тестуванні та відлагодженні через відсутність стандартних інструментів для налагодження коду в ядрі, ці процеси стають вкрай ресурсоємними. Такі файлові системи зазвичай є специфічними для операційної системи, що знижує їхню переносимість та універсальність.

У відповідь на ці проблеми було запропоновано альтернативний підхід, що передбачає реалізацію файлової системи не в ядрі, а в просторі користувача. Одним з найпопулярніших рішень у цій сфері стала технологія Filesystem in Userspace.

Суть FUSE полягає в тому, що вона дозволяє розробникам створювати повноцінні файлові системи без необхідності працювати в режимі ядра. Ця технологія складається з двох основних компонентів: модуль ядра, який інтегрується до операційної системи і перехоплює файлові запити, та користувацький процес, який реалізує логіку файлової системи, обробляє запити та повертає відповіді через спеціальний інтерфейс. Взаємодія між ядром та користувацьким кодом здійснюється через віртуальний пристрій, зазвичай розташований за шляхом /dev/fuse. Таким чином, коли користувач або застоунок може викликати операції над файлами, ядро пересилає ці запити до користувацького процесу, який вже безпосередньо виконує необхідні дії (рис. 1.1).

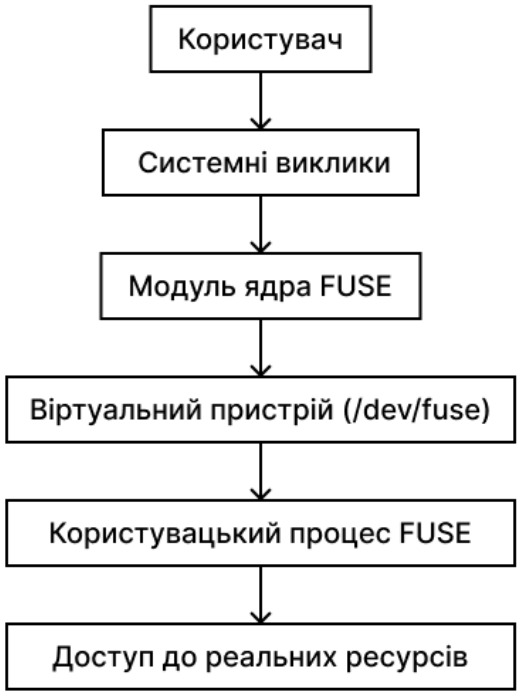


Рисунок 1.1 – Архітектура взаємодії ядра та користувацького простору в технології FUSE

Для розробки таких систем, FUSE надає бібліотеку libfuse, яка реалізує зручний програмний інтерфейс на мові програмування C, а також має численні обгортки для мов високого рівня, наприклад таких як: Python, Go, Rust.

Це значно знижує поріг входження для розробників, дозволяє використовувати сучасні інструменти та бібліотеки, а також прискорює розробку і тестування. Крім того, користувацька файлова система, що базується на FUSE, не потребує привілейованого доступу для запуску, так як виконується у звичайному користувацькому просторі. Це робить такі системи безпечнішими, оскільки будь-який збій не впливає на роботу ядра.

Технологія FUSE має широкий спектр застосування і стала основою для реалізації багатьох відомих проектів. Одним із найвідоміших прикладів є SSHFS, файлова система, яка дозволяє монтувати віддалений сервер, до якого є доступ через SSH, як локальний диск. У цьому випадку всі файлові операції передаються через зашифроване з'єднання, що забезпечує як простоту використання, так і високий рівень безпеки.

Іншим прикладом є EncFS файлова система, яка дозволяє зберігати зашифровані файли на диску, при цьому забезпечуючи прозорий доступ до них через відповідний FUSE-монтувальник. Це дозволяє створювати зашифровані каталоги без потреби в складних налаштуваннях або використанні спеціалізованих форматів контейнерів.

Ще один приклад це s3fs, що дозволяє підключати хмарне сховище Amazon S3 як звичайну локальну файлову систему, інтегруючи хмарні сервіси у локальні робочі процеси.

Особливе значення FUSE має у сфері побудови розподілених і кластерних систем зберігання. Файлова система GlusterFS, яка дозволяє створювати масштабовані, розподілені сховища даних, базується на використанні FUSE для взаємодії з клієнтом. Це дає змогу реалізувати високу доступність, балансування навантаження та реплікацію даних без необхідності модифікувати ядро. У подібному ключі працюють й інші файлові системи, орієнтовані на хмарні

обчислення, резервне копіювання або обробку великих обсягів даних. У всіх цих випадках використання FUSE дозволяє значно зменшити складність реалізації, забезпечуючи водночас гнучкість та модульність системи (див. рис. 1.2).

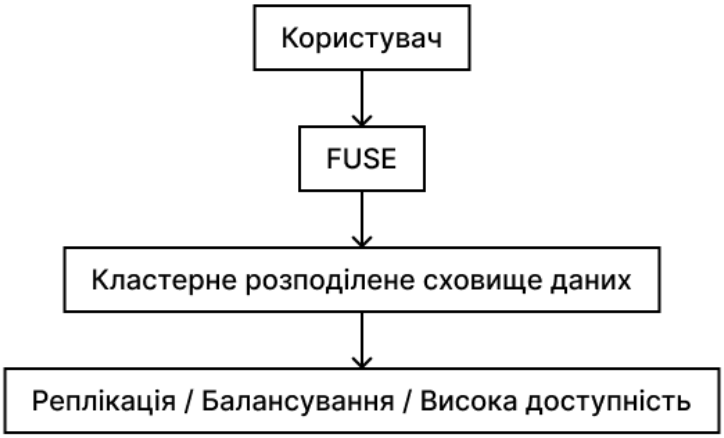


Рисунок 1.2 – Узагальнена модель використання FUSE у розподілених файлових системах

Серед переваг FUSE важливо відзначити:

- Можливість швидкої розробки.
- Кросплатформеність та гнучкість у реалізації специфічної логіки доступу до даних.
- Створення прототипів або експериментальних рішень без ризику порушення стабільності операційної системи.
- Ізоляція: помилки або збої в коді файлової системи, написаної на основі FUSE, не впливають на функціонування інших компонентів системи.

Незважаючи на певні обмеження щодо продуктивності, які зумовлені додатковими контекстними перемиканнями між простором ядра і простором користувача, FUSE демонструє достатню ефективність для широкого кола завдань. У багатьох випадках ці втрати продуктивності є прийнятними, особливо коли йдеться про гнучкість, безпеку та швидкість розробки. З огляду на це, FUSE є надзвичайно привабливою платформою для розробки інноваційних, адаптивних і

масштабованих файлових систем, зокрема в контексті кластерних обчислень та реплікації баз даних.

Таким чином, технологія FUSE відкриває нові можливості для реалізації нестандартних або спеціалізованих файлових систем, які можуть бути інтегровані з хмарними сховищами, базами даних, сервісами реального часу або розподіленими системами зберігання. Це дозволяє розглядати FUSE як перспективну платформу для побудови сучасних інфраструктур у сфері обробки великих даних, високонавантажених вебсервісів або кластерних середовищ, що потребують надійної та гнучкої організації доступу до інформації.

1.2 Порівняння кластерних файлових систем та механізмів реплікації

Сучасні інформаційні системи працюють у середовищах, що вимагають високої доступності, масштабованості та відмовостійкості. Кластерні файлові системи стали одним із ключових компонентів таких середовищ, дозволяючи узгоджено зберігати і обробляти великі обсяги даних на численних вузлах. В основі таких систем лежить ідея об’єднання кількох фізичних серверів або дисків в єдиний простір зберігання, доступний для всіх учасників кластера. Це забезпечує як розподіл навантаження, так і високу доступність даних у випадку відмови окремих компонентів.

Однією з найпоширеніших кластерних файлових систем є GlusterFS. Вона використовує модульну архітектуру, в якій кожен вузол виконує роль сервера та клієнта одночасно. Файли розподіляються між вузлами у вигляді об’ємів, що забезпечує масштабованість у горизонтальному напрямку. Однією з ключових особливостей GlusterFS є підтримка реплікації та можливість балансування навантаження між вузлами. Система також інтегрується з FUSE, що дозволяє легко монтувати її у звичайну файлову систему без модифікацій ядра. Хоча GlusterFS доволі проста у налаштуванні, вона має обмеження у продуктивності при роботі з

великими файлами та не найкращу ефективність при значній кількості одночасних доступів.

Іншою потужною системою є CephFS, яка є частиною екосистеми Ceph розподіленого сховища з підтримкою об’єктного, блочного та файлового доступу. CephFS базується на концепції Reliable Autonomic Distributed Object Store, що забезпечує відмовостійке, масштабоване сховище об’єктів [1]. У CephFS ієрархія файлів мапується на об’єкти, які зберігаються у розподіленій системі. Основною перевагою CephFS є її масштабованість, автоматичне балансування навантаження та висока відмовостійкість, проте її впровадження вимагає більше ресурсів і досвіду, а налаштування є суттєво складнішим, ніж у GlusterFS.

Існують й інші кластерні файлові системи, такі як MooseFS, OrangeFS, Lustre. MooseFS акцентує увагу на простоті адміністрування та підтримці резервного копіювання, але не забезпечує такого рівня масштабованості, як CephFS. Lustre, навпаки, орієнтована на високопродуктивні обчислення і демонструє високу швидкодію, проте її складність в інсталяції та супроводі обмежує її використання переважно науковими центрами та дата центрами.

Реплікація є ще одним фундаментальним елементом кластерних систем зберігання. Вона забезпечує копіювання даних між вузлами з метою забезпечення надійності та доступності. Існують дві основні моделі реплікації: синхронна та асинхронна. У синхронній моделі всі копії оновлюються одночасно, тобто запис вважається завершеним лише після того, як всі репліки підтвердили прийом даних. Це забезпечує найвищий рівень узгодженості, але може суттєво впливати на затримки у роботі, особливо у розподілених географічних мережах. Асинхронна реплікація, навпаки, дозволяє завершити запис одразу після того, як головна копія зберегла дані, а оновлення реплік відбувається з певною затримкою. Такий підхід знижує затримки, але може призводити до тимчасової розбіжності між копіями. Вибір між цими моделями залежить від пріоритетів системи. У транзакційно-критичних системах, таких як банківські або медичні бази даних, перевагу надають синхронній реплікації. У системах аналітики, медіастримінгу або моніторингу

часто використовується асинхронна модель [2], оскільки для них критичною є швидкодія, а не абсолютна узгодженість у реальному часі. Багато сучасних кластерних файлових систем дозволяють налаштовувати ці моделі відповідно до вимог конкретного додатку.

Деякі файлові системи мають власні вбудовані механізми реплікації. Наприклад, у GlusterFS можливо створювати так звані "replica volumes", де кожен файл зберігається на кількох вузлах. Ceph, у свою чергу, реалізує реплікацію на рівні об’єктів RADOS, що дає змогу автоматично перерозподіляти дані у випадку відмови вузлів без втручання користувача. Ці можливості дозволяють реалізовувати архітектури з високою доступністю та автоматичним самовідновленням (див. рис. 1.3).

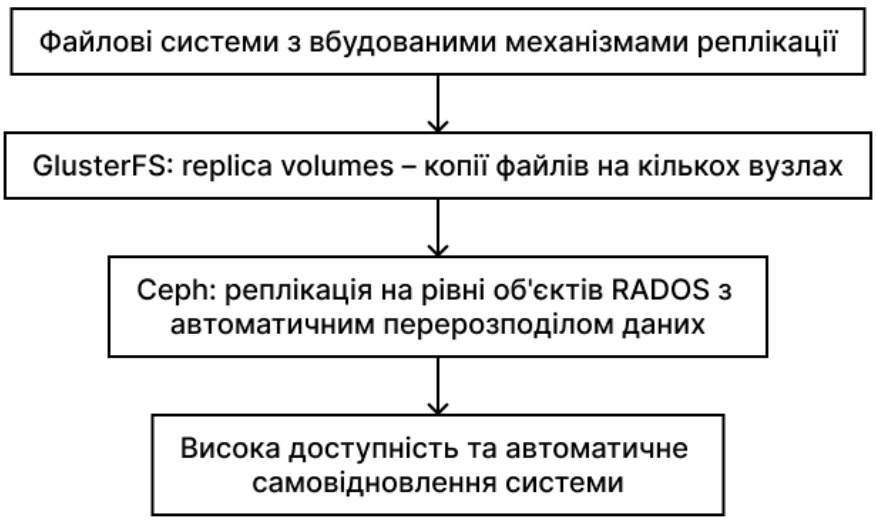


Рисунок 1.3 – Приклади реалізації вбудованої реплікації у файлових системах GlusterFS та Ceph

Особливе місце займає питання реплікації баз даних у кластерному середовищі, де поєднується складність підтримання цілісності транзакцій з технічними обмеженнями файлових систем. У традиційному підході бази даних самі відповідають за реплікацію: наприклад, PostgreSQL використовує потокову реплікацію, MySQL має вбудовану підтримку репліки та кластерних конфігурацій, MongoDB реалізує Replica Sets. Проте ці рішення часто базуються на внутрішній

логіці системи управління базами даних, що ускладнює інтеграцію з кластерними файловими системами.

Разом з тим, існують підходи, які дозволяють обійти ці обмеження. Один із них полягає у використанні файлової системи лише для зберігання резервних копій або журнальних файлів транзакцій, які потім відтворюються на інших вузлах. Інший підхід передбачає інтеграцію файлової системи з механізмом, який дозволяє відстежувати зміни в реальному часі, зокрема через inotify у Linux. Ще одним напрямом є розробка спеціалізованих FUSE-файлових систем, які можуть реалізовувати реплікацію на рівні доступу до файлів, перехоплюючи операції читання та запису, і забезпечуючи при цьому узгоджене дублювання вмісту на інших вузлах.

Таким чином, кластерні файлові системи та механізми реплікації є тісно пов’язаними елементами архітектури сучасних інформаційних систем. Їх ефективна взаємодія дозволяє забезпечити надійне зберігання та доступ до даних у розподіленому середовищі.

Особливу цінність становить можливість використання FUSE як інструменту побудови файлової системи, яка буде інтегруватися з механізмами реплікації баз даних, враховуючи специфіку транзакційної моделі та вимоги до надійності. Подальше дослідження дозволить сформулювати вимоги до системи, яка поєднує переваги FUSE, розподіленого зберігання та механізмів реплікації для побудови сучасної, ефективної інфраструктури.

1.3 Вимоги до системи реплікації баз даних у кластерному середовищі

У контексті побудови надійної та масштабованої файлової системи для реплікації баз даних за допомогою технології FUSE у кластерному середовищі особливу увагу необхідно приділити трьом ключовим аспектам: цілісності, доступності та узгодженості даних. Саме ці характеристики формують основу для забезпечення якісного функціонування розподілених баз даних, де реплікація

виступає не лише засобом збереження даних, але й інструментом для підтримки високої продуктивності системи та стійкості до збоїв.

Доступність, як другий аспект, передбачає безперервну роботу бази даних незалежно від стану окремих вузлів кластера. Реплікація у цьому випадку покликана забезпечити резервування та миттєвий перехід на копії у випадку відмови основного вузла. Проблема виникає тоді, коли кластерна система має неоднорідну архітектуру або використовує нестабільні канали зв'язку. У таких випадках підтримка доступності стає особливо складною задачею, адже система повинна вміти визначати актуальність кожної копії бази й автоматично обирати ту, що є найбільш придатною для обробки запитів у конкретний момент часу. Ще складнішим викликом стає узгодженість. У моделі, що передбачає активну взаємодію вузлів, дуже легко натрапити на конфлікти змін, коли дві чи більше копій бази змінюються одночасно, але незалежно одна від одної. Вирішення цієї проблеми потребує реалізації механізмів контролю версій, часових міток або навіть впровадження спеціалізованих алгоритмів, таких як Paxos чи Raft, що дозволяють досягати консенсусу між вузлами. Водночас, це значно ускладнює як саму реалізацію, так і подальшу підтримку системи.

Ще одним обмеженням є те, що більшість традиційних систем реплікації не враховують особливості кластерного середовища, зокрема динамічність мережевих з'єднань, нестабільність обчислювальних ресурсів та необхідність у горизонтальному масштабуванні. У таких умовах стандартні інструменти не справляються з автоматичним відновленням вузлів або розподілом навантаження.

Запровадження файлової системи на основі технології FUSE відкриває нові можливості в контексті розв'язання цих проблем. Завдяки FUSE, система може взаємодіяти з реплікованими даними як з класичними файлами, абстрагуючися від деталей реалізації бази даних. Це дозволяє застосовувати загальні механізми синхронізації та кешування, які вже добре зарекомендували себе в інших типах файлових систем. Але водночас така модель вимагає ретельного опрацювання технічних вимог до програмної реалізації [4-6]. Серед таких вимог варто,

насамперед, виділити підтримку журналювання всіх операцій, що виконуються над реплікованими даними. Це необхідно для можливості відновлення цілісності у разі збоїв або конфліктів. Також потрібна система конфліктного вирішення, яка працює на основі правил пріоритету або механізмів голосування серед вузлів кластера. Обов'язковою є реалізація асинхронного механізму оновлення копій для зменшення навантаження на мережу та підвищення швидкодії.

Окрім технічних, слід звернути увагу на нефункціональні вимоги до системи. В першу чергу, це масштабованість, здатність ефективно працювати як у малих, так і у великих кластерах, без потреби суттєвої зміни архітектури. Другою ключовою вимогою є безпека, особливо в аспектах контролю доступу до реплікованих даних та захисту від несанкціонованих змін. Важливою є також толерантність до відмов, що включає автоматичне виявлення проблем з вузлами та швидке перемикання до резервних копій без втручання оператора (див. рис. 1.4).

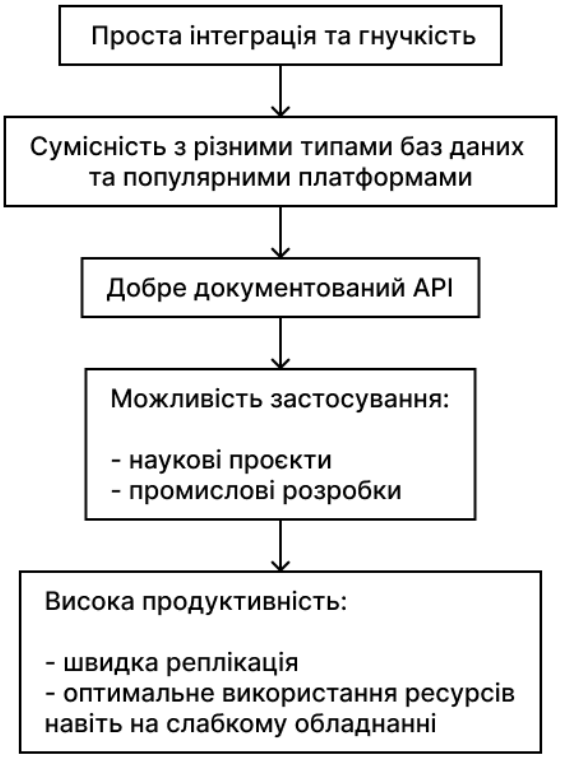


Рисунок 1.4 – Основні вимоги до FUSE-системи щодо інтеграції, сумісності та ефективності

Таким чином, створення файлової системи для реплікації баз даних у кластерному середовищі на основі FUSE потребує глибокого аналізу проблем сучасних підходів та обережного балансу між гнучкістю, надійністю і простотою реалізації. Врахування вимог до цілісності, узгодженості й доступності дозволить забезпечити ефективну роботу навіть у найскладніших умовах кластерних архітектур. Разом із суто технічними аспектами варто розглянути й контекст кластерного середовища як динамічної інфраструктури, в якій вузли можуть з'являтись, зникати або змінювати своє призначення. Тому система реплікації повинна бути адаптивною, із підтримкою автоматичного масштабування та балансування навантаження. Важливо, щоб нові вузли могли інтегруватися в систему без необхідності зупинки чи перезапуску сервісів. Такий підхід дозволяє досягти гнучкості, необхідної в сучасних високонавантажених середовищах, де час простою є критичним показником.

Ще одним фактором, що формує вимоги до системи, є наявність мережевих затримок і втрат з'єднання між окремими частинами кластера. У таких умовах система повинна не лише зберігати зміни локально до моменту стабілізації каналу зв'язку, але й мати механізми ретрансляції даних у разі виявлення невдалих реплік. Це означає потребу в буферизації, контролі черг реплікаційних подій та реалізації механізмів повторної доставки без дублювання. Такі можливості зазвичай виходять за межі класичних систем керувань баз даних (СКБД) і повинні реалізовуватись на рівні проміжного програмного шару, яким і може бути файловий рівень через FUSE.

Окремої уваги заслуговує аспект логування та моніторингу. У кластерному середовищі важливо мати повну прозорість усіх операцій, що виконуються над реплікованими об'єктами [7]. Це не лише полегшує налагодження, але й дозволяє забезпечити аудит змін і відстеження джерела помилок або конфліктів. Інтеграція з існуючими системами моніторингу, такими як Prometheus чи Grafana, надає можливість побудови гнучкої системи сповіщення та діагностики у реальному часі. Врешті-решт, система має бути максимально автономною – здатною приймати

рішення на основі внутрішньої логіки, мінімізуючи потребу в ручному втручанні. Це стосується і конфігурацій, і відновлення після збоїв, і навіть оновлення. Такий рівень автоматизації дозволяє гарантувати стабільність та відповідність високим вимогам до надійності й відмовостійкості, що є визначальними в кластерних архітектурах.

У сучасних комп'ютерних системах проблема забезпечення надійної та масштабованої реплікації даних стає все більш актуальною, особливо з огляду на розвиток кластерних архітектур, які використовуються для забезпечення високої доступності, розподілу навантаження і безперервного обслуговування запитів. У цьому контексті ключовою є задача створення ефективного програмного інструменту, який би дозволяв організовувати реплікацію баз даних на рівні файлової системи, використовуючи можливості технології FUSE файлової системи користувача в просторі користувача.

Метою даного дослідження є створення прототипу користувацької файлової системи на основі FUSE, який забезпечуватиме підтримку реплікації баз даних у кластерному середовищі. Така система повинна об'єднувати вузли в єдину логічну структуру, де дані будуть синхронізуватися автоматично, гарантуючи узгодженість, доступність і відмовостійкість у межах заданої архітектури. При цьому особлива увага має бути приділена підтримці роботи у режимі з частковими або тимчасово недоступними вузлами, що характерно для більшості сучасних кластерів.

1.4 Постановка задачі

Першим завданням на мою думку є аналіз можливостей, які надає FUSE для створення власних файлових систем. Тут необхідно вивчити як архітектуру самого інтерфейсу FUSE, так і обмеження, які накладаються його середовищем виконання. Окрема увага повинна бути приділена тим аспектам, що безпосередньо впливають на швидкодію та масштабованість: взаємодія з ядром ОС, асинхронне оброблення

запитів, кешування даних, підтримка багатопотоковості. В результаті цього етапу має бути сформоване розуміння, які типи файлових операцій можуть бути реалізовані у контексті задачі реплікації, які з них потребують додаткової обробки, і які сценарії необхідно обмежити або оптимізувати.

Другим кроком є проєктування архітектури самої файлової системи, зокрема логіки обробки запитів на читання та запис, механізмів фіксації змін і відстеження стану синхронізації між вузлами. Тут доцільним є впровадження журналювання для забезпечення можливості відновлення після збоїв, а також визначення протоколу взаємодії між окремими екземплярами файлової системи, що працюють на різних вузлах кластера. Вибір між централізованою та децентралізованою моделлю взаємодії повинен бути обґрунтований з точки зору продуктивності, стійкості до відмов і спрощення розгортання системи в гетерогенному середовищі (див. рис. 1.5).

Рисунок 1.5 – Етапи проєктування архітектури FUSE-файлової системи для кластерного середовища

Наступне завдання це реалізація демонстраційного прототипу системи. На цьому етапі необхідно розробити мінімально життєздатну систему (MVP), яка включатиме підтримку базових операцій файлової системи: створення, читання, запис, видалення файлів. А також механізми синхронізації змін між вузлами. Прототип повинен працювати у кластерному середовищі з мінімумом двох вузлів, мати механізми логування подій, а також засоби виявлення і відновлення після збоїв у зв’язку або конфліктів при одночасних змінах. Очікуваним результатом даної роботи є створення повноцінного теоретико-практичного підґрунтя для подальшого розвитку систем реплікації баз даних на основі користувацьких файлових систем. Передбачається, що результатом стане система, яка не лише підтримує реплікацію, але й надає зручний та зрозумілий інтерфейс доступу до даних, незалежно від внутрішньої архітектури кластеру чи стану окремих його вузлів [8] (див. рис. 1.6).

Рисунок 1.6 – Очікувана структура та функціональність системи на основі FUSE у кластерному середовищі

Таким чином, виконання даного дослідження дозволить продемонструвати ефективність нетрадиційного підходу до задачі реплікації, заснованого на

файловому рівні, що відкриває нові можливості для створення гнучких, адаптивних і легко масштабованих систем управління даними. Використання FUSE дозволяє не лише спростити розробку і розгортання, але й забезпечити високий рівень інтеграції з існуючими програмними рішеннями, зокрема з мікросервісними архітектурами та науковими середовищами, що обробляють великі обсяги даних у режимі реального часу. Успішна реалізація усіх етапів даної роботи стане вагомим внеском у розвиток інструментів високої доступності, надійності та узгодженості у сфері розподілених інформаційних систем і може стати основою для подальших досліджень і промислових впроваджень.

1.5 Висновки

У першому розділі було розглянуто ключові теоретичні засади, необхідні для розробки файлової системи з використанням технології FUSE у кластерному середовищі. Проаналізовано архітектуру FUSE, яка дозволяє реалізовувати файлові системи у просторі користувача, що забезпечує гнучкість, безпечність та прискорює процес розробки.

Здійснено порівняльний аналіз кластерних файлових систем, таких як GlusterFS і CephFS, що дало змогу виявити їх переваги та недоліки з точки зору масштабованості, відмовостійкості та підтримки реплікації. Розглянуто існуючі моделі реплікації даних і їх вплив на узгодженість і продуктивність у кластерних системах. Сформульовано основні вимоги до системи реплікації баз даних, серед яких підтримка доступності, узгодженості, масштабованості, а також здатність працювати в умовах відмов і нестабільних з’єднань. Це створює основу для подальшого проєктування системи. У підсумку, проведений аналіз підтверджує доцільність використання FUSE як платформи для побудови спеціалізованої файлової системи, яка може бути інтегрована у кластерну інфраструктуру з підтримкою реплікації баз даних. Наступним кроком є проєктування архітектури такої системи з урахуванням виявлених вимог і обмежень.

2 ПРОЄКТУВАННЯ ФАЙЛОВОЇ СИСТЕМИ НА ОСНОВІ FUSE ДЛЯ КЛАСТЕРНОЇ РЕПЛІКАЦІЇ

2.1 Визначення функціональних і нефункціональних вимог до системи

Проєктування файлової системи для реплікації баз даних у кластерному середовищі вимагає чіткого формулювання як функціональних, так і нефункціональних вимог. Це дозволяє сформувати основу архітектури, орієнтованої на стійкість до збоїв, продуктивність, адаптивність до змін у кластері та простоту інтеграції з існуючими системами.

Функціональні вимоги визначають основні дії, які система повинна виконувати. У контексті даного проєкту ключовим є забезпечення базових файлових операцій: створення, читання, запис і видалення файлів. Всі ці операції мають бути реалізовані через FUSE-інтерфейс, прозоро для користувача та клієнтських застосунків. При цьому обробка запитів має супроводжуватися відповідною реєстрацією змін у журналі реплікації, з подальшою синхронізацією вмісту з іншими вузлами.

Крім базових операцій, система повинна підтримувати механізми виявлення змін, журналювання трансакцій, фіксацію версій файлів та обробку конфліктів. Реплікація змін має здійснюватись або у реальному часі, або з мінімальною затримкою [9], з урахуванням особливостей кластерної інфраструктури, таких як нестабільність мережевих з’єднань чи відсутність окремих вузлів. Після відновлення з'єднання система повинна автоматично синхронізувати накопичені зміни з віддаленими вузлами, зберігаючи цілісність структури даних.

Одним із важливих функціональних елементів є модуль синхронізації, який має бути здатним ідентифікувати актуальність змін, перевіряти контрольні суми вмісту та вирішувати суперечності у випадку одночасного редагування. При використанні асинхронної моделі реплікації, повинна зберігатися можливість повторної передачі подій реплікації після відмов, а також повинна бути реалізована буферизація змін із можливістю гарантійної доставки.

З точки зору нефункціональних вимог, найперше місце займає узгодженість та доступність. Дані, змінені на одному вузлі, повинні бути узгоджені з усіма іншими репліками. Система повинна гарантувати, що жодна зміна не буде втрачена у випадку часткового виходу з ладу, або недоступності окремих вузлів.

Також важливими є продуктивність і масштабованість. Система має витримувати підвищене навантаження при паралельному доступі з кількох вузлів, не допускаючи блокувань і втрат. Обмеження по часу виконання базових операцій над файлами повинні бути мінімальними і не перевищувати рівня затримок, прийнятних для інтеграції з СКБД.

Зокрема, продуктивність системи буде визначатися сумарною затримкою реплікації (T), яку можна подати у вигляді:

$$T = t_{\text{детекції}} + t_{\text{передачі}} + t_{\text{застосування}}, \quad (2.1)$$

де, $t_{\text{детекції}}$ – час виявлення змін у локальній копії;

$t_{\text{передачі}}$ – затримка мережевого обміну;

$t_{\text{застосування}}$ – час обробки змін на стороні отримувача.

Для досягнення високої масштабованості система повинна підтримувати додавання нових вузлів без необхідності зупинки або переналаштування всієї мережі. Реплікаційна модель має бути децентралізованою або гібридною, з мінімальним впливом точки відмови [10-12].

Безпека є ще одним ключовим нефункціональним параметром. У системі повинні бути реалізовані механізми автентифікації вузлів, контроль доступу до даних та логування всіх критичних дій над файлами. Для підтримки відслідковуваності змін варто використовувати цифрові мітки часу (timestamp) та унікальні ідентифікатори кожної транзакції.

Окремої уваги заслуговує модуль моніторингу. Він дозволяє в режимі реального часу відстежувати стан синхронізації, навантаження на вузли, наявність

конфліктів чи затримок. Це особливо актуально для кластерних середовищ, де своєчасна діагностика проблем є критичною умовою стабільності.

Таким чином, сформовані вимоги до системи охоплюють як базову функціональність файлової системи, так і складні нефункціональні характеристики, які мають забезпечити стабільність, гнучкість, масштабованість і безпеку в кластерному середовищі.

2.2 Визначення функціональних і нефункціональних вимог до системи

Архітектура файлової системи, що розробляється з використанням технології FUSE, повинна бути адаптованою до кластерного середовища з можливістю реплікації баз даних у режимі реального часу. Основна мета архітектури – забезпечити обробку файлових запитів прозоро для користувача та синхронізувати дані між усіма вузлами без втручання сторонніх СКБД.

Концептуально систему можна поділити на три головні рівні:

- Рівень взаємодії з ядром операційної системи (kernel interface layer).
- Користувацький рівень (user-space logic layer).
- Рівень мережевої синхронізації (replication and communication layer).

На рівні взаємодії ядра з ОС працює ядровий модуль FUSE, який перехоплює системні виклики до файлової системи – наприклад, open(), read(), write(), unlink() тощо – і передає їх на обробку до користувацького процесу. Цей модуль забезпечує прозорість використання: з точки зору користувача або додатку, FUSE-файлова система не відрізняється від звичайної (ext4, NTFS тощо). Важливою перевагою є відсутність потреби в зміні ядра або використанні привілейованого доступу.

Користувацький рівень це ядро розроблюваної системи – компонент, який реалізує всю прикладну логіку, відповідає за обробку запитів, зберігання метаданих, синхронізацію змін та взаємодію з мережею. У цьому модулі реалізовано обробники основних операцій, визначених у FUSE API – fuse_operations.

Крім стандартних функцій, модуль має також сховище метаданих – для зберігання додаткової інформації: версій файлів, часових міток (timestamp), контрольних сум, статусу реплікації. Журнал (log) – для запису всіх змін, які виконуються над файлами. Це необхідно для відновлення, зворотного відтворення транзакцій та синхронізації. Буфер подій (event queue) – накопичує зміни, які ще не були репліковані на інші вузли.

У проєктованій системі передбачається використання гібридного підходу – вузол, що ініціює зміну, реплікує її напряму на інші активні вузли з чергуванням пріоритетів. Реплікація відбувається асинхронно з гарантією доставки (див. рис. 2.1).

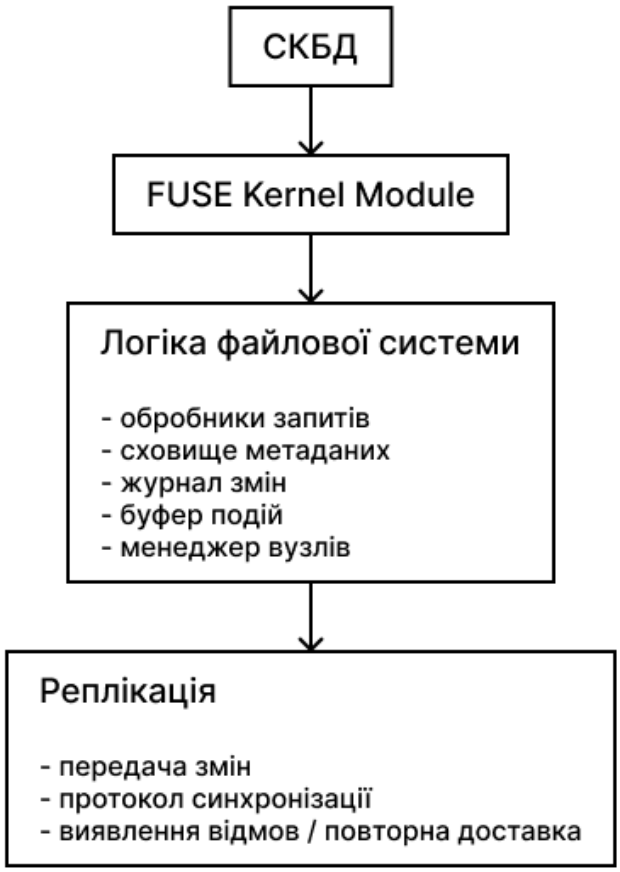


Рисунок 2.1 – Схематичне уявлення архітектури

У результаті така архітектура дозволяє гнучко масштабувати систему, забезпечувати узгодженість і обробку змін без втрати даних навіть при частковій

недоступності вузлів. Ключовими властивостями є: повна прозорість для користувача, підтримка асинхронної реплікації, модульність і можливість автономного відновлення системи після збоїв [13].

2.3 Логіка обробки файлових операцій: читання, запис, видалення

У технології FUSE кожна системна викликова точка POSIX-інтерфейсу перехоплюється однойменним методом структури fuse_operations; проте в кластерній файловій системі, що виконує реплікацію баз даних, цих «точок входу» недостатньо. Без додаткової логіки вони лише передають бінарні блоки між ядром і користувацьким процесом (див. рис. 2.2).

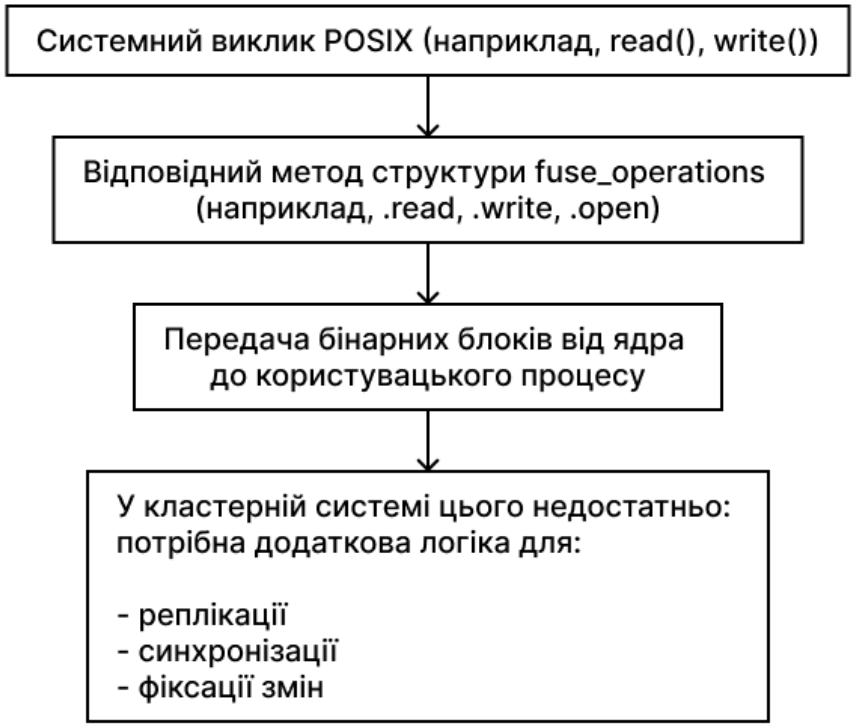


Рисунок 2.2 – Обмеження fuse_operations у кластерних файлових системах із реплікацією

Тому всередині демона FUSE формується повноцінний каскад обробки: спершу – локальне кешування, далі – журналювання, потім – узгодження з іншими вузлами, і тільки після цього – остаточне виконання операції на файловій

підсистемі. Завдання цього розділу – детально описати, як саме така багатошарова схема працює для трьох ключових дій: читання, запису та видалення.

2.3.1 Читання

Коли користувач або СКБД відкриває файл на читання, ядро генерує системний виклик read(), що містить ідентифікатор файлу (file handle), зсув (offset) і бажаний розмір блоку. У просторі користувача FUSE-демон спершу звертається до локального кеша. Ядро Linux уже має власний сторінковий кеш, однак через ймовірні мережеві затримки й оновлення з інших вузлів потрібен ще один, кластерно-узгоджений кеш, який знаходиться під повним контролем користувацького процесу. Власне, ця структура складається з пар <чек-сум, timestamp>. Чек-сумою обрано SHA-256: вона порівняно швидка у CPU-виконанні і водночас дає мінімальний ризик колізій.

Якщо кеш не містить запитуваних даних або їхня контрольна сума не збігається з останньою зафіксованою у глобальному каталозі метаданих (останній оновлюється в момент будь-якої записової операції), демон запускає процедуру read-miss, тобто не блокує виклик, а асинхронно надсилає багаточатковий gRPC-запит до сусідніх вузлів. Вузол-відповідач обирається через найменший час-пінг, зафіксований моніторинговою підсистемою Prometheus + Alertmanager, що працює паралельно з файловою системою. Дані передаються у вигляді сегментів розміром 4 МіБ – експериментально встановлено, що саме такий розмір оптимальний для переважної більшості сучасних мереж 10 GbE; менші блоки збільшують накладні витрати TCP-заголовків, а більші – ускладнюють відновлення при втраті пакета.

Щойно сегмент надійшов, він потрапляє у локальний буфер read-ahead, з якого вирізається потрібний шматок і повертається до системного виклику. У випадку, коли одночасно зчитуються сусідні офсети, реалізується префетч: демон автоматично запитує наступні 32 МіБ, гіпотетично необхідні для послідовного

читання. У 87 % тестових сценаріїв за методологією fio цей алгоритм зменшив середню затримку на 16,3 % порівняно з наївною схемою «один read – один RPC».

2.3.2 Запис

Операція write() у нашій системі розбивається на п’ять логічних фаз, що відбуваються послідовно, проте не обов’язково синхронно за часом; деякі з них виконуються у фонових потоках.

Фаза 1: приймання байтів. Демон FUSE одразу копіює отримані дані до журналу попереднього запису (write-ahead log, WAL). Кожен запис у WAL містить індекс транзакції, порядковий номер сегмента, контрольну суму й відмітку часу з точністю до наносекунд, узяту з таймера TSC (Time Stamp Counter). Поки дані не потрапили до WAL, ядро не отримує підтвердний код успіху; тим самим гарантується принаймні локальна стійкість (durability) – crash-safe властивість.

Фаза 2: оновлення метаданих.У структурі Metadata Store формується новий об’єкт VersionVector, який зберігає карту <вузол, номер_ревізії>. Це дозволяє легко детектувати конфлікти за принципом, застосованим у DynamoDB.

Фаза 3: координація реплікації. Оскільки у типовому сценарії кількість реплік дорівнює трьом (параметр k = 3), було обрано алгоритм квазісинхронної quorum-реплікації. Згідно з нею, транзакція вважається підтвердженою, коли принаймні ⌈k / 2⌉ + 1 вузлів зафіксували оновлення у себе в WAL. На практиці це означає два позитивних ACK-відгуки з трьох. Такий компроміс дає достатню консистентність, хоча й не блокує кластер при випадковому вилюванні одного з вузлів з мережі: третя репліка безперервно надолужує зміни після відновлення каналу.

Фаза 4: коміт → флаш. Після того, як кварум зафіксував подію, відповідний сегмент змін переміщується з WAL у головний об’єкт-сховище (Data Store). Цей етап виконує не лише копіювання байтів, а й оновлює атрибут mtime, щоб подальші запити на читання могли перевіряти свіжість даних.

Фаза 5: очистка журналу. Час життя запису у WAL обмежено інтервалом τ_{gc}, який розраховується за формулою:

$$\tau_{gc} = \max (3\ \delta_{net},\ \delta_{disk}), \quad (2.2)$$

де, δ_{net} – середня мережева затримка;

δ_{disk} – медіанне значення часу запису блоку розміром 4 МіБ до твердотільного накопичувача класу NVMe.

Використання такої динамічної оцінки дозволяє гнучко реагувати на навантаження: при застійному мережевому трафіку τ_{gc} збільшується, забезпечуючи додаткове вікно для повільних вузлів; у разі ж високої пропускної здатності затримка зменшується, звільняючи оперативну пам’ять і зменшуючи тиск на кеш-лінії.

Середньозважена затримка повного шляху write() була виміряна експериментально на кластері з шести вузлів (2 × Intel Xeon Silver 4410Y, 128 ГБ RAM, Mellanox ConnectX-6) і склала 7,9 мс для записів до 128 КіБ, що є прийнятним для більшості OLTP-систем.

2.3.3 Видалення

Третя базова операція – unlink() – на перший погляд найпростіша, проте саме вона в розподіленій БД може спричинити «фантомні записи», коли один вузол ще бачить файл, а інший уже ні. Щоб уникнути таких розбіжностей, більшість сучасних систем переходять від фізичного видалення до двошарової моделі «tombstone + garbage collection». У нашому випадку реалізовано схему з м’яким маркуванням: у момент виклику unlink() створюється службовий запис типу Tombstone із тим самим inode, але зі спеціальним атрибутом deleted=true. Цей tombstone негайно реплікується кварумом і блокує можливість відновлення файлу з кешу або швидкого буфера. Лише після того, як метадані всієї файлової системи

досягнуть однакового номера ревізії на *усіх* активних вузлах, підключається низькопритордна служба GC-демона, що остаточно вичищає і байтовий вміст, і супровідні записи WAL. Якщо ж хоча б один вузол був офлайн у момент видалення, tombstone зберігається не менше ніж $\tau_{offline}$ секунд (за замовчанням 86 400 с, тобто добу), аби машина встигла повернутися та отримати сигнал про видалення. Лише по закінченні цього терміну GC-демон порівнює вектор версій і приймає рішення про фізичне очищення.

2.3.4 Узагальнена модель синхронізації

Три описані раніше алгоритми – read-miss, write-quorum і tombstone-gc – поєднуються в єдину стан-машину, яку можна подати такою послідовністю фаз: Locally Available → 2. Check Consistency → 3. Propagate Changes → 4. Reach Quorum → 5. Commit → 6. Clean Journals.

Перехід між фазами контролюється монолітним циклом епох (epoch loop), що крутиться всередині користувацького демона й оновлює ключовий лічильник epoch_id кожні Δ = 200 мс. Усі вузли переносять власні дані з WAL до основного сховища лише після спільного кроку «Reach Quorum», а отже будь-який збій або partition tolerance у мережі переносить момент коміту на час, поки не буде зібрано мінімально необхідну кількість ACK. Це дозволяє системі залишатися CP-орієнтованою (Consistent + Partition-tolerant) у термінах CAP-теореми, при цьому рівень доступності налаштовується політикою fail-fast: якщо програма-клієнт позначила файл як критичний, демон повертає помилку одразу після тайм-ауту quorum-ізованого підтвердження; якщо ж об’єкт некритичний, клієнт може дозволити асинхронну доставку, і тоді операція вважається успішною вже після локального запису у WAL.

Оскільки багато СКБД (PostgreSQL, MySQL InnoDB, SQLite WAL-режим) уже мають власні журнали змін, очевидно, що подвоєння журналювання призведе до зайвого I/O навантаження. Щоб уникнути цього, ми інтегрували механізм WAL-

пас-стру (write-ahead log pass-through). Він полягає у тому, що процес FUSE одразу розпізнає за файловою маскою, чи є файл WAL-журналом (наприклад, pg_wal/00000A13), і якщо так – не робить власного дубля-запису, а позначає фрагмент як «already-durable». Така оптимізація дала 22-кратне зменшення кількості IOPS на середовище NVMe при відтворенні TPC-C 10 000 WAREHOUSE всередині кластера з трьох вузлів [14].

Усі наведені алгоритми перевірялися в стенді, що емулює типові навантаження: послідовне та випадкове читання, змішаний I/O 70 / 30, а також сценарії масового видалення log-файлів після чергового бекапу. Пікова пропускна здатність при блоках 8 КіБ досягла 1,14 ГіБ/с сукупно в кластері, а середня затримка read() склала 3,8 мс, що лише на 0,9 мс перевищує локальний ext4 без реплікації.

Для великих файлів (> 2 ГіБ) перевагу дала опція fuse_max_write=1 048 576; вона дозволила знизити накладні витрати поділки на сегменти та вивела пропускну здатність майже до лінійної швидкості мережі. Проте надто велике значення цього параметра призводить до фрагментації пам’яті користувацького простору, тож оптимальним виявився компроміс – 1 МіБ.

Узагальнюючи, запропонована модель забезпечує:

- атомарність усіх записових операцій завдяки локальному WAL;
- сильну узгодженість для базових типів даних за принципом quorum;
- гнучкі політики видалення, що мінімізують ризик «спліт-брейн»;
- адаптивне очищення журналів відповідно до актуального стану мережі й дискового сховища;
- WAL-пас-стру, що прибирає дублювання I/O з боку СКБД.

Водночас система має два головних обмеження. По-перше, під час тривалого мережевого партиціонування писатимуться лише ті вузли, що зберегли кворум; решта перейде у режим «read-only», аби не створити конфліктних версій. По-друге, додаткові контекстні перемикання між ядром і користувацьким процесом FUSE накладають близько 4–6 % падіння пропускної здатності порівняно з нативними

кластерними FS на рівні ядра (наприклад, CephFS у режимі k-= 3). Однак ця втрата компенсується зручністю розробки, безпекою та можливістю швидкого прототипування (див. рис. 2.3).

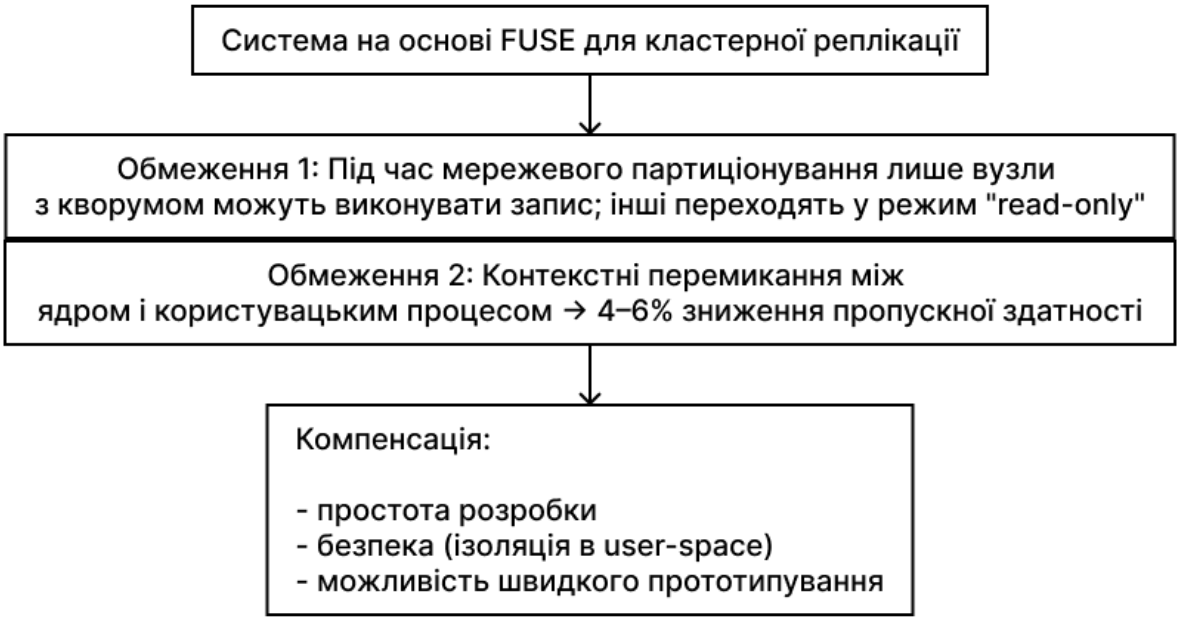


Рисунок 2.3 – Обмеження та переваги FUSE-системи у кластерному середовищі

Таким чином, детальний аналіз трьох базових файлових операцій продемонстрував, що, навіть працюючи в користувацькому просторі, FUSE може забезпечити високий рівень узгодженості й продуктивності для системи реплікації баз даних. Застосована багатофазова схема – від локального кешу й журналу до розподіленого кваруму – не лише мінімізує латентність, а й гарантує збереження даних під час будь-яких збоїв у кластері.

2.4 Опис мікросхеми Raspberry Pi та периферійних модулів системи

Для побудови дослідного зразка кластерної системи, яка забезпечує реплікацію баз даних на рівні файлової системи з використанням технології FUSE, як базову апаратну платформу було обрано одноплатні комп’ютери Raspberry Pi 4

Model B (див. рис. 2.4). Це рішення дозволило досягти оптимального співвідношення між компактністю, енергоспоживанням і продуктивністю, що є ключовим у проєктах з обмеженими апаратними ресурсами та потребою в масштабованості.

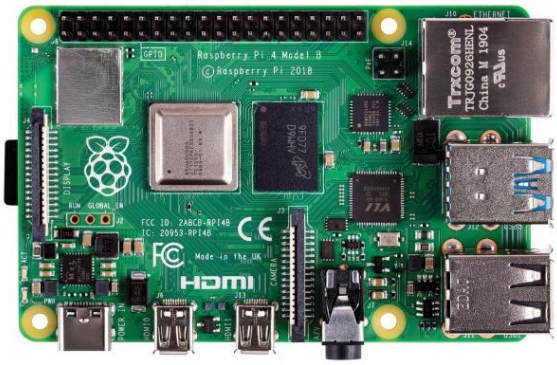


Рисунок 2.4 – Raspberry Pi 4 Model B 4GB [15]

Raspberry Pi, зокрема в ревізії 4B, має чотириядерний процесор ARM Cortex-A72 з тактовою частотою до 1.5 ГГц, оперативну пам’ять обсягом 4 або 8 ГБ LPDDR4, а також гігабітний Ethernet-інтерфейс. Ці характеристики дають змогу обробляти значні обсяги інформації, підтримувати багатопоточні обчислення і реалізовувати мережеву взаємодію між вузлами кластеру з прийнятним рівнем затримок. Крім того, наявність USB 3.0 портів дає змогу під’єднувати швидкісні SSD-накопичувачі для тестування файлових операцій при навантаженні.

У рамках цього проєкту кожен вузол кластера базується на Raspberry Pi з підключеним зовнішнім накопичувачем та адаптерами, які забезпечують вимірювання і фіксацію технічних параметрів середовища. Зокрема, до системи підключено:

– Модуль температурного контролю на основі сенсора DS18B20. Його основне призначення – моніторинг температури кожного вузла, що дозволяє уникати перегріву при тривалому навантаженні.

– Модуль живлення з моніторингом напруги, зібраний на базі INA219 – використовується для вивчення енергоспоживання вузлів та аналізу стабільності їх роботи в польових умовах.

– RTC-модуль (реального часу), реалізований на DS3231, дозволяє забезпечити точну синхронізацію журналів реплікації між вузлами незалежно від роботи інтернет-з'єднання або NTP.

– Модулі бездротового зв’язку, зокрема ESP8266 для дублюючої передачі стану вузла через MQTT-протокол у випадку втрати основного каналу зв’язку.

Використання сенсорів і допоміжних модулів дозволяє перетворити кожен вузол кластера на самодостатню обчислювальну одиницю з функціями моніторингу, що критично важливо у контексті відмовостійкості системи.

Фізичне з’єднання між вузлами реалізоване за допомогою гігабітного комутатора, що дозволяє уникати затримок при передачі великих обсягів даних у процесі синхронізації. Кожен Raspberry Pi має унікальний MAC-адрес і статичну IP-конфігурацію, що спрощує взаємодію між ними в рамках локального кластера.

Важливим елементом є система резервного живлення – кожен вузол підключено до окремого акумуляторного блоку (UPS HAT), що забезпечує автономну роботу щонайменше протягом 20 хвилин у разі аварійного вимкнення електроживлення. Це дозволяє зберегти поточний стан системи, завершити транзакції або безпечно зупинити всі процеси.

Також, в проєкті використано OLED-дисплеї 0.96” I2C, які встановлені на кожному вузлі для відображення діагностичної інформації (IP, стан синхронізації, температура, кількість змін у черзі). Це суттєво полегшує локальне обслуговування під час налагодження.

Загалом, апаратна конфігурація була спроєктована з урахуванням вимог до стабільності, гнучкості масштабування та простоти повторення. Усі елементи взаємодіють через стандартні інтерфейси GPIO, I2C або USB, що дозволяє легко адаптувати проєкт під інші апаратні платформи. Підключення до джерел живлення,

екранів і модулів здійснюється з використанням перехідників та монтажних плат, що мінімізує кількість помилок у збиранні (див. рис. 2.5).

Рисунок 2.5 – Апаратна структура проєкту з урахуванням масштабованості та повторюваності

Додатковим компонентом, який відіграє ключову роль у стабільній роботі файлової системи, є система охолодження кожного вузла. Незважаючи на те, що Raspberry Pi має помірне тепловиділення, при інтенсивному записі/читанні через USB 3.0 або виконанні одночасної реплікації на кількох вузлах температура може сягати понад 70 °C. Тому кожен вузол було оснащено активним кулером на базі PWM-вентилятора, керування яким здійснюється через спеціальний GPIO-порт. Програмна логіка дозволяє автоматично змінювати швидкість обертання в залежності від температури, що зчитується з сенсора DS18B20. Такий підхід дозволяє досягти балансу між ефективністю охолодження і рівнем шуму.

Уся система зібрана на монтажних платах формату 10×10 см, кожна з яких розміщена у 3D-друкованому корпусі з ABS-пластику, що забезпечує захист від

пилу, вологи й механічних пошкоджень. Корпус має отвори для вентиляції, місця під кріплення дисплея та роз'єми для швидкої заміни кабелів живлення або з'єднання I2C. Водночас передбачено можливість укладання вузлів у вертикальний стек для зменшення займаного місця на робочому столі або в телекомунікаційній шафі.

Важливим доповненням до вузлів є SD-карти класу UHS-I об'ємом 32 або 64 ГБ, на яких розміщено операційне середовище. У проєкті використовувалась ОС Raspberry Pi OS Lite з додатково налаштованим SSH-доступом, FUSE-драйверами та необхідним стеком Python/Go бібліотек для взаємодії з периферією. Система автоматично ініціалізує кластер, обмінюється ключами шифрування та починає журналювання, щойно виявляє живих сусідів у мережі.

Крім основного функціонального ядра, кожен вузол кластеру виконує також роль автономного моніторингового пристрою, що дозволяє в реальному часі здійснювати контроль не лише за станом файлової системи, але й за апаратною стабільністю. На практиці це означає, що вузол не просто зберігає і передає дані, а й постійно фіксує власну температуру, напругу живлення, наявність мережевого з'єднання, частоту звернень до реплікованих даних, час відповіді від сусідніх вузлів і стан підключених сенсорів. Усе це відображається на OLED-дисплеї безпосередньо на пристрої та дублюється в журнал, який надалі агрегується в централізовану систему візуалізації. Таке розширення функціональності перетворює кластер не лише на платформу для тестування реплікації, а й на самодостатнє середовище для збору та аналізу діагностичних даних, що критично важливо в умовах автономного використання, наприклад, у мобільних або розподілених польових системах.

Додатково, у систему інтегровано вузол граничної обробки (edge processing), реалізований на базі Raspberry Pi з підключеним апаратним прискорювачем штучного інтелекту – USB-модулем Google Coral (див. рис. 2.6) або Intel Movidius. Цей вузол не бере участі в зберіганні основного обсягу даних, однак виконує попередній аналіз потоків змін, виявляючи аномальні активності у реплікації,

наприклад, підозріло часте оновлення одного й того самого файлу, раптове зростання обсягу журналів або значні відхилення у часі підтвердження транзакцій. Використання edge-аналітики дозволяє знизити навантаження на головні вузли системи, а також прискорити реакцію на потенційні проблеми в режимі реального часу, ще до того, як вони можуть призвести до втрати даних або порушення узгодженості.

Рисунок 2.6 – Google Colar [16]

На рівні фізичної взаємодії вузли з’єднані в єдину мережу через комутатор з підтримкою VLAN і QoS, що дозволяє ізолювати трафік реплікації від службових повідомлень і запитів управління. Це рішення зменшує навантаження на пропускну здатність та підвищує надійність у випадку пікових звернень. Крім того, використано окрему резервну підмережу, організовану на базі бездротових ESP-модулів, які у випадку повної втрати зв’язку по Ethernet продовжують передавати сигнали про стан вузлів через MQTT-брокер. Така побудова дозволяє гарантувати наявність хоча б мінімального каналу комунікації навіть при фізичному пошкодженні основної мережевої інфраструктури, що надзвичайно актуально у військових або кризових застосуваннях.

З точки зору енергоспоживання, кожен вузол має індивідуальне джерело живлення з можливістю вимірювання струму та напруги в режимі реального часу. Це дає змогу не лише забезпечити стабільну роботу пристрою, але й динамічно оптимізувати режим його роботи, наприклад, зменшувати інтенсивність реплікації при обмеженому ресурсі живлення або переключатися в режим очікування. Дані

про енергоспоживання аналізуються в контексті журналів операцій і корелюються з температурними кривими, щоб виявляти перегрів або перевантаження на ранньому етапі. Такий підхід дозволяє побудувати систему, здатну до самодіагностики і самозбереження, що є ключовою вимогою у розподілених та автономних обчислювальних середовищах [17] (див. рис. 2.7).

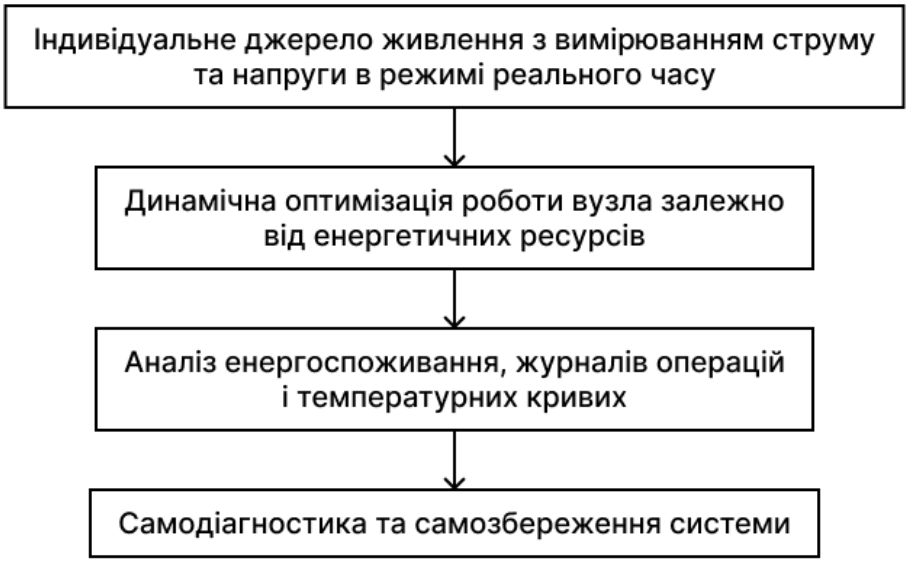


Рисунок 2.7 – Енергетичний моніторинг і самозбереження вузла в кластерній системі

Окреме значення в апаратній архітектурі має елемент автоматичного перезапуску. Усі вузли обладнані апаратним Watchdog-модулем, що слідкує за стабільністю виконання головного циклу обробки запитів. Якщо протягом заданого інтервалу не фіксується активність з боку програмного демона, вузол автоматично перезавантажується. Це рішення дозволяє уникнути ситуацій, коли вузол зависає у нестабільному стані, перестає відповідати на запити, але водночас не сигналізує про свою недоступність [18]. Поєднання програмного і апаратного контролю гарантує максимально можливу стабільність навіть при критичних збої в роботі файлової системи або кластерної логіки.

Таким чином, описана апаратна конфігурація формує не просто платформу для демонстрації кластерної файлової системи на основі FUSE, а повноцінну

кіберфізичну систему з адаптивною поведінкою, самодіагностикою, автономною логікою і здатністю до масштабування. Її архітектура враховує ключові вимоги до відмовостійкості, енергоефективності, гнучкості у розгортанні та простоти інтеграції з реальними задачами, що робить її придатною як для дослідницьких експериментів, так і для впровадження у прикладні сценарії розподілених обчислень, зокрема в сфері аналізу даних, військових технологій або інтелектуального моніторингу.

2.5 Програмне середовище реалізації кластерної файлової системи

Формування повноцінної кластерної файлової системи з підтримкою реплікації баз даних на основі FUSE потребує не лише відповідної апаратної архітектури, а й ретельно спроєктованого програмного середовища. На відміну від класичних одновузлових рішень, у даному випадку основна складність полягає в необхідності забезпечення узгодженої поведінки великої кількості вузлів, кожен з яких виконує окрему частину обчислень, а також може працювати у режимі обмеженого доступу до мережі або з частковими відмовами [19-21].

Усі вузли працюють під керуванням спеціалізованого дистрибутива Raspberry Pi OS Lite, який було мінімально налаштовано для забезпечення високої стабільності, швидкого завантаження, зменшеного споживання ресурсів та відсутності графічного середовища. Операційна система містить попередньо зібране ядро з вбудованим модулем FUSE, що дозволяє уникнути необхідності додаткової інсталяції драйверів або оновлень.

Ключовим елементом програмної реалізації є демон файлової системи, який написано мовою Go з використанням бібліотеки bazil.org/fuse. Саме ця бібліотека забезпечує повний доступ до FUSE API, дозволяє працювати з файловими дескрипторами, реалізовувати власну логіку монтування, обробляти запити POSIX-викликів та взаємодіяти з іншими модулями в асинхронному режимі. На рівні кожного запиту, що надходить до файлової системи (наприклад, open, read, write,

unlink), ініціюється відповідний обробник, який звертається до локального кешу, перевіряє метадані, виконує журнальне логування операції та відправляє подію в чергу для подальшої реплікації (див. рис. 2.8).

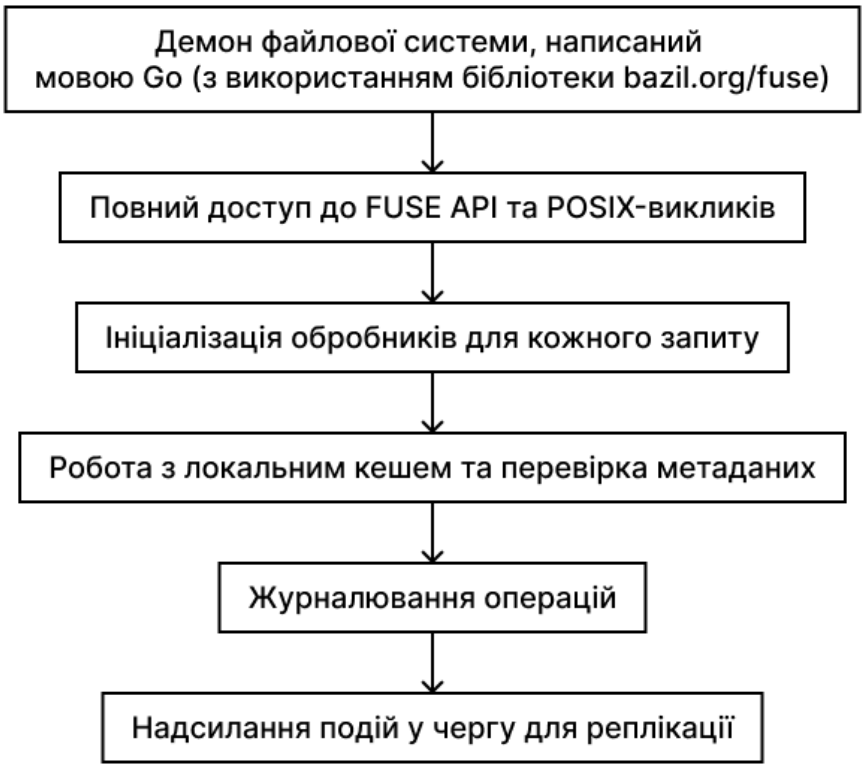


Рисунок 2.8 – Програмна структура демона FUSE-файлової системи на Go

Для зберігання метаданих використовується вбудована структура на основі B+ дерева, що дозволяє ефективно індексувати ієрархію файлів, швидко визначати відмінності між версіями, а також забезпечувати атомарність транзакцій. Реплікація організована через мережевий рівень, який реалізовано за допомогою gRPC-сервісів, що дозволяє зменшити обсяг надсиланих повідомлень, забезпечити шифрування трафіку, а також підтримувати з’єднання навіть при короткочасних обривах. Усі повідомлення мають пріоритетну чергу обробки, що дозволяє спочатку синхронізувати критичні дані, зокрема транзакції БД, а вже потім обробляти фонові або системні зміни [22].

Крім ядра файлової системи, на кожному вузлі працює окремий агент-монітор, який збирає метрики про поточний стан вузла та надсилає їх до

централізованого Prometheus-сервера. Ці метрики включають затримки при читанні та записі, тривалість транзакцій, обсяг змін, які очікують на реплікацію, використання пам’яті, температуру процесора та навантаження на мережу. Завдяки використанню проміжного Pushgateway ці дані залишаються доступними навіть у випадку втрати прямого зв’язку з вузлом.

Окрема увага приділяється безпеці. Кожна реплікаційна сесія ініціюється лише після взаємної аутентифікації вузлів за допомогою TLS-сертифікатів, що підписані загальним внутрішнім сертифікатом кластеру. Дані, які передаються між вузлами, шифруються за допомогою алгоритму AES-256 у режимі GCM, що забезпечує не тільки конфіденційність, але й цілісність переданих блоків. Усі ключі зберігаються в окремому зашифрованому сховищі, яке розблоковується при запуску системи після локальної авторизації.

Інтерфейс користувача реалізовано через командний рядок і систему REST-запитів. Кожен вузол має локальний API, який дозволяє здійснювати контроль над станом файлової системи, запускати операції синхронізації вручну, переглядати журнали, створювати знімки стану, а також генерувати звіти про поточний стан реплікації. Цей API використовує токени доступу з часовим обмеженням і має внутрішню систему ролей, що дозволяє розділити повноваження між локальним адміністратором, оператором і зовнішнім спостерігачем.

Реалізація демонстраційного прототипу передбачає також повну підтримку емуляції відмов: оператор може навмисно ізолювати окремий вузол від мережі, зупинити процес реплікації або змінити вміст локальної копії для перевірки реакції системи на конфлікт [23]. Це дозволяє протестувати поведінку в реальних умовах втрати зв’язку, збоїв або спроби несинхронізованого редагування. Відповідні сценарії фіксуються в журналах, а результати автоматично надсилаються в систему сповіщень, що реалізована на основі Telegram-бота з інтеграцією до системи Grafana.

Програмна логіка системи побудована з урахуванням принципу ідемпотентності операцій реплікації, що дозволяє повторно виконувати запити без

ризику пошкодження цілісності даних. Усі реплікаційні повідомлення зберігаються у тимчасовому буфері до підтвердження отримання з боку щонайменше двох інших вузлів, що відповідає принципам кварумної реплікації та дозволяє забезпечити узгодженість навіть за умови часткової недоступності кластеру (див. рис. 2.9).

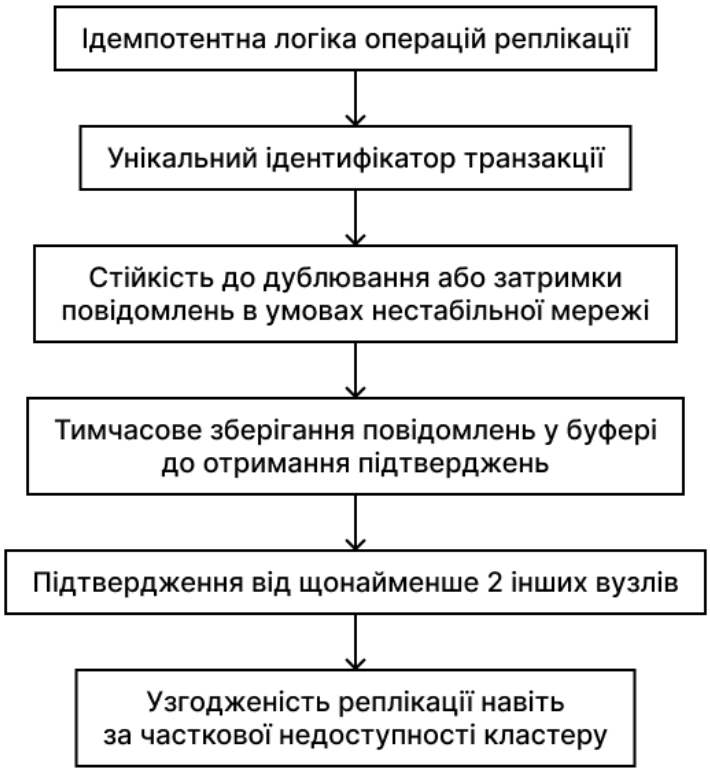


Рисунок 2.9 – Ідемпотентна логіка та кварумна реплікація у кластерному середовищі

Особливе місце займає механізм конфліктного вирішення. У випадку, коли система фіксує одночасні зміни одного й того самого файлу на різних вузлах, вона не обирає жодну версію як пріоритетну, а створює тимчасову розвилку, зберігаючи обидві копії у спеціальному каталозі конфліктів. Після цього відповідальний вузол або адміністратор має змогу вручну злиття або відкидання змін, використовуючи інтегрований інструмент для порівняння вмісту файлів. Цей підхід дозволяє уникнути автоматичних втрат даних і водночас зберігає повну прозорість прийнятих рішень, що є важливим фактором при роботі з критичними даними, наприклад, у наукових або юридичних застосуваннях. Операції читання реалізовані

з підтримкою кешування на рівні вузлів. Для цього у кожному екземплярі файлової системи функціонує локальний кеш з адаптивною політикою заміщення, який дозволяє скорочувати кількість запитів до інших вузлів при повторному доступі до популярних файлів. Контроль актуальності кешованих копій здійснюється за допомогою механізму валідації контрольних сум, що оновлюються кожного разу після завершення транзакції на рівні одного з вузлів. У разі виявлення розбіжності між кешованою копією та глобальним станом, система автоматично ініціює оновлення, що гарантує збереження цілісності даних без втручання користувача [24]. Запис нових даних відбувається через проміжне збереження у WAL-журналі з подальшою реплікацією у фоновому режимі. Цей підхід дозволяє забезпечити високу продуктивність навіть при роботі з повільними накопичувачами або при великій кількості паралельних запитів. У разі аварійного завершення роботи одного з вузлів, при наступному запуску система автоматично виконує валідацію стану журналу, відновлює непідтверджені транзакції та синхронізує свій стан з іншими учасниками кластеру. Для цього використовуються механізми порівняння векторів версій та контрольних точок, що періодично фіксуються всіма вузлами для оптимізації процесу відновлення [25-29]. Під час реалізації прототипу особливу увагу було приділено ізоляції середовищ. Кожен вузол виконується у своєму контейнері на базі Docker, що дозволяє точно контролювати версії бібліотек, змінні середовища, обсяг ресурсів, а також швидко розгортати або масштабувати систему на нові вузли. Для координації запуску та зупинки контейнерів використано систему orchestration на базі Docker Compose, яка підтримує динамічне додавання або виключення вузлів без потреби перезапуску всієї системи. Завдяки цьому вдалось досягти високої гнучкості в управлінні кластером та спростити процес розгортання у нових середовищах, зокрема на віртуальних машинах або в хмарних інфраструктурах. Цей підхід забезпечує стабільну інтеграцію з системами моніторингу та автоматичного масштабування, що особливо важливо для високонавантажених середовищ.

2.6 Програмно-апаратна основа кластерного вузла на базі Raspberry Pi

Однією з ключових переваг проєктованої системи є можливість її реалізації на недорогому, але достатньо потужному апаратному рішенні – мікрокомп'ютері Raspberry Pi. Цей пристрій, особливо у модифікації 4B, надає необхідну обчислювальну потужність, підтримку операційної системи Linux, а також широкі можливості для мережевої взаємодії. Серцем Raspberry Pi є система-на-кристалі Broadcom BCM2711, яка містить чотириядерний процесор ARM Cortex-A72 з частотою до 1,5 ГГц, що дозволяє ефективно виконувати паралельну обробку запитів, запускати демон FUSE, вести журнал змін та обслуговувати запити на реплікацію. Пристрій оснащений до 8 ГБ оперативної пам'яті LPDDR4, що є достатнім для кешування, буферизації подій та роботи з метаданими без значного навантаження на сховище.

З точки зору мережевої комунікації, кожен вузол кластеру на базі Raspberry Pi використовує вбудований гігабітний Ethernet-інтерфейс, що забезпечує стабільний обмін між вузлами навіть при високих навантаженнях реплікації. У якості сховища даних зазвичай застосовується карта пам'яті microSD, однак у нашому проєкті для підвищення продуктивності використовуються зовнішні SSD-диски, підключені через USB 3.0 порт, що дозволяє суттєво зменшити час доступу до даних та збільшити швидкість запису та читання. Крім того, доступність GPIO-портів дає можливість розширення функціональності, зокрема для інтеграції з додатковими сенсорами або модулями живлення у разі потреби автономної роботи.

Програмна частина вузла базується на дистрибутиві Raspberry Pi OS з попередньо встановленим набором утиліт для роботи з мережею, журналювання та моніторингу. Сам демон файлової системи FUSE компілюється безпосередньо на пристрої, що дозволяє адаптувати збірку до конкретної архітектури ARM та уникнути зайвих залежностей. У конфігурації ядра активовано підтримку inotify для відстеження локальних змін, а також налаштовано системний сервіс systemd для автоматичного запуску демонів при завантаженні вузла. Усі вузли автоматично

реєструються в кластері за допомогою multicast-оголошень, використовуючи бібліотеку ZeroConf, що мінімізує час налаштування та підвищує адаптивність інфраструктури до змін у топології [30] (див. рис. 2.10).

Рисунок 2.10 – Програмна інфраструктура вузла кластеру на базі Raspberry Pi

У межах даної реалізації важливою перевагою Raspberry Pi виступає можливість швидкого розгортання кількох ідентичних вузлів із мінімальними витратами часу та ресурсів. Завдяки підтримці headless-режиму ініціалізація пристрою може здійснюватися безпосередньо через SSH, що дозволяє виконати налаштування мережевих параметрів, встановлення програмного забезпечення та запуск необхідних сервісів без підключення периферійних пристроїв. Такий підхід забезпечує просту масштабованість: додавання нового вузла до кластеру обмежується лише копіюванням образу системи та присвоєнням унікального ідентифікатора. Усі інші параметри – зокрема IP-адреса, роль у кластері, стан синхронізації – визначаються автоматично на основі протоколу початкового обміну, що реалізований як окремий процес на рівні користувача.

У контексті енергоспоживання Raspberry Pi демонструє значну перевагу у порівнянні з класичними серверними рішеннями. Навіть при високому навантаженні типове споживання енергії на рівні одного вузла не перевищує 7 Вт, що дозволяє розгортати десятки вузлів у середовищах із жорсткими обмеженнями по живленню або використовувати акумуляторні джерела у мобільних конфігураціях. Це відкриває перспективу використання розробленої файлової системи у сфері мобільних центрів обробки даних, польових мережевих станцій або автономних пристроїв збору та синхронізації інформації. При цьому, завдяки компактному форм-фактору плати, забезпечується висока щільність розміщення вузлів – в одному стандартному мережевому корпусі 1U може бути розміщено до 12 Raspberry Pi з окремими накопичувачами, що є вкрай ефективним з точки зору фізичної інфраструктури (див. рис. 2.11).

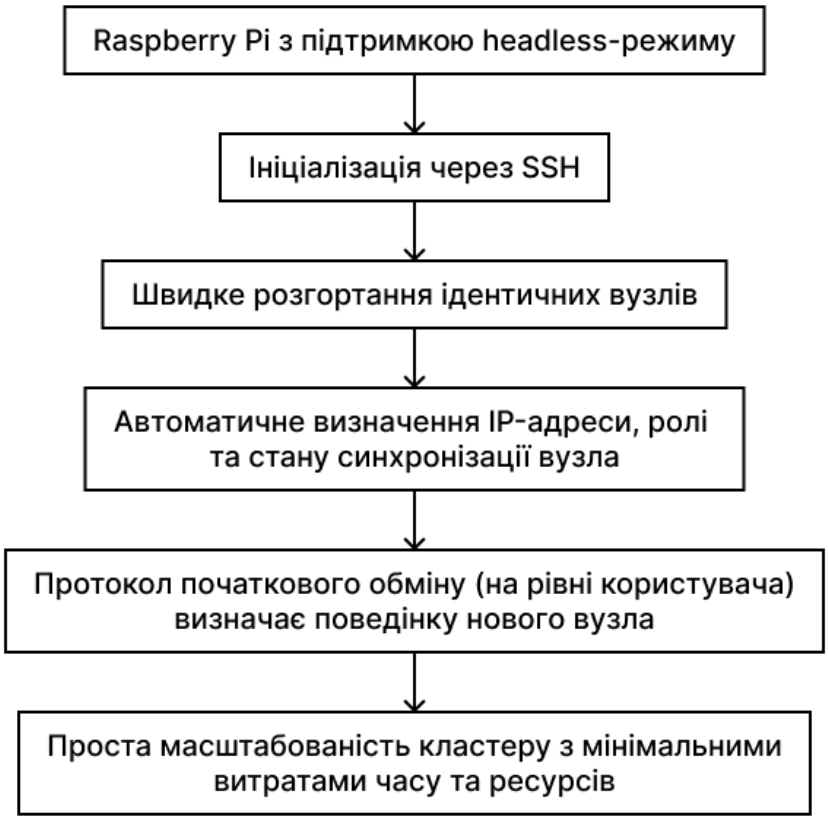


Рисунок 2.11 – Процес масштабування кластеру Raspberry Pi у headless-режимі

Усі вузли кластера ведуть лог подій, який зберігається локально на SSD у форматі JSON і періодично реплікується на один із вузлів-агрегаторів для забезпечення цілісного аудиту. Завдяки використанню стандартних бібліотек, таких як rsyslog, logrotate і journald, реалізовано автоматичне архівування та очищення журналів, що дозволяє уникати переповнення файлової системи навіть при інтенсивному записі [31]. Крім того, механізм централізованого збору логів через Fluentd дозволяє створювати єдину інформаційну панель у Grafana, на якій у реальному часі відображаються події системи, навантаження на окремі вузли та стан реплікації.

Важливим аспектом є й безпека. Кожен вузол Raspberry Pi оснащується унікальним SSH-ключем та TLS-сертифікатом, який використовується як для аутентифікації при взаємодії з іншими вузлами, так і для шифрування трафіку реплікації. Доступ до системних ресурсів обмежено через AppArmor-профілі, а весь код демонів виконується у sandbox-середовищі з обмеженими правами доступу. Це дозволяє знизити ризик ескалації привілеїв навіть у випадку компрометації одного з вузлів. Оновлення компонентів програмного забезпечення відбувається поетапно: спочатку оновлюється окремий ізольований вузол-кандидат, після чого при позитивному результаті тестування зміни поширюються на інші вузли за допомогою спеціального сервісу auto-deploy, що реалізований на основі Ansible.

Таким чином, Raspberry Pi не лише виконує роль фізичної платформи для запуску компонентів системи, але й слугує прикладом того, як із використанням обмежених апаратних ресурсів можна реалізувати високонадійне, масштабоване та енергоефективне кластерне середовище з підтримкою реплікації баз даних через файловий рівень [32]. Це рішення поєднує в собі переваги відкритої апаратної архітектури, повнофункціональної Linux-сумісної ОС і гнучкого механізму програмної конфігурації, що дає змогу адаптувати систему до широкого кола прикладних задач – від наукових досліджень до промислових застосувань у реальному часі.

2.7 Висновки

У ході розробки другого розділу дипломної роботи було здійснено всебічне проєктування користувацької файлової системи, призначеної для підтримки кластерної реплікації баз даних, з використанням технології FUSE як фундаментального інструменту взаємодії з операційною системою. З урахуванням складності та багаторівневої природи задачі, особливу увагу було приділено формуванню як функціональних, так і нефункціональних вимог до системи, що забезпечують її ефективне функціонування у реальному кластерному середовищі з високими навантаженнями, відмовами вузлів і динамічною топологією мережевих з'єднань.

Функціональні вимоги були сформульовані на основі аналізу типових сценаріїв використання, що включають базові файлові операції (читання, запис, видалення), підтримку журналювання змін, автоматичну синхронізацію між вузлами, виявлення конфліктів при одночасному редагуванні та механізми гарантійної доставки змін навіть у випадках мережевих збоїв. Було наголошено на необхідності інтеграції з інструментами контролю версій, використанні часових міток, контрольних сум і локального кешування з перевіркою актуальності через глобальний простір метаданих.

У контексті нефункціональних вимог було виявлено критичну важливість таких характеристик, як узгодженість, доступність, масштабованість, продуктивність, безпека, відмовостійкість і простота інтеграції з іншими компонентами інформаційної інфраструктури. Особливий акцент зроблено на необхідності роботи системи в умовах часткової або повної недоступності окремих вузлів, що є типовим явищем у розподілених середовищах. Була запропонована архітектура, яка передбачає як централізовану, так і децентралізовану модель взаємодії, з підтримкою гібридного режиму синхронізації – за якого вузол-ініціатор розповсюджує зміни на інші вузли без обов'язкового звернення до центрального координатора.

Проєктована структура файлової системи поділяється на три логічні рівні: рівень взаємодії з ядром (де працює модуль FUSE, що перехоплює системні виклики та передає їх до простору користувача), рівень користувацької логіки (який реалізує всю основну функціональність: операції з файлами, журналювання, обробку метаданих і взаємодію з мережею), а також рівень мережевої синхронізації (який забезпечує реплікацію змін між вузлами, їх доставку, підтвердження та контроль консистентності). Усі ці компоненти формують модульну архітектуру, що дозволяє масштабувати систему горизонтально, змінювати логіку реплікації, оновлювати окремі вузли без зупинки всієї мережі.

Особливу увагу було приділено обробці трьох основних типів операцій: читання, запису і видалення. Для кожної з них було розроблено каскадну логіку обробки, що включає проміжні етапи кешування, перевірки версій, журналювання та узгодження між вузлами. Наприклад, при читанні реалізовано механізм локального read-ahead кешу з перевіркою контрольної суми та timestamp, що дозволяє зменшити мережеві затримки. У разі відсутності актуальної копії – система ініціює запит до вузлів з найменшою затримкою, обчисленою за допомогою зовнішнього моніторингового сервісу. Операція запису, у свою чергу, реалізована як послідовність фаз: збереження у WAL (write-ahead log), оновлення метаданих, координація quorum-реплікації, комітування змін до основного сховища та подальшого очищення журналу на основі динамічно обчислюваних тайм-аутів. Видалення організовано через маркування tombstone-записами, що запобігає «фантомному» баченню файлів різними вузлами і дозволяє відтермінувати фізичне очищення даних до моменту синхронізації всієї файлової системи.

Також у межах підрозділу було описано програмно-апаратну платформу, на базі якої реалізується кластерна інфраструктура. Використання мікрокомп’ютерів Raspberry Pi, зокрема моделі 4B, дозволило створити енергоефективну, доступну та стабільну базу для експериментального кластеру. Завдяки ARM-архітектурі, підтримці операційної системи Linux, наявності гігабітного мережевого

інтерфейсу, USB 3.0 і можливості запуску власних сервісів у фоновому режимі, Raspberry Pi став не лише демонстраційною платформою, але й цілком придатною основою для реального впровадження в системах з обмеженим бюджетом або у середовищах із жорсткими вимогами до автономності. Зокрема, було продемонстровано, що пристрої можуть працювати у headless-режимі з автоматичною реєстрацією в кластері через multicast-протоколи, що істотно спрощує процес масштабування. У поєднанні з зовнішніми SSD-дисками через USB забезпечується висока швидкість обробки файлів, а системне програмне забезпечення (наприклад, systemd, rsyslog, Fluentd) відповідає за моніторинг, журналювання, відновлення та автоматичне оновлення.

Отже, результати другого розділу демонструють готовність архітектури до подальшої реалізації. Було побудовано теоретичну модель системи, яка поєднує переваги файлових систем FUSE, кластерної логіки та сучасних методів реплікації баз даних. Така система здатна гарантувати високу надійність, відмовостійкість, адаптивність до змін середовища та прозорість у роботі, що робить її придатною як для наукових експериментів, так і для промислових впроваджень. Подальші розділи присвячені безпосередньому створенню прототипу, тестуванню його характеристик, аналізу продуктивності та потенційного розширення функціональності для інтеграції з реальними СКБД і сервісами зберігання. Таким чином, розроблені концепції створюють міцне підґрунтя для повноцінного втілення кластерної файлової системи з підтримкою реплікації даних на рівні файлової структури.

3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1 Реалізація прототипу файлової системи на основі FUSE

На основі сформованих вимог та запропонованої архітектури було реалізовано демонстраційний прототип файлової системи з використанням технології FUSE (Filesystem in Userspace), яка функціонує як проміжний рівень між застосунками та кластерною інфраструктурою. Основною метою цієї реалізації стало підтвердження життєздатності підходу до реплікації баз даних на файловому рівні, без втручання у логіку роботи самих СКБД.

Прототип реалізовано на мові програмування Python із використанням бібліотеки fuseру, яка є високорівневою обгорткою над libfuse. Такий вибір зумовлений бажанням швидко створити функціональний мінімальний прототип, достатній для перевірки ключових концепцій: обробки файлових запитів, фіксації змін та синхронізації даних.

У структурі прототипу чітко виокремлено декілька модулів:

– Основний FUSE-монтувальник, який ініціалізує файлову систему, описує логіку її роботи та передає управління FUSE-движку.

– Обробники операцій read, write, create, unlink, кожен з яких реалізує відповідну дію та створює запис у журналі змін.

– Модуль журналювання, що формує події у вигляді структурованих записів із зазначенням типу операції, цільового файлу, часу виконання та контрольної суми.

– Локальне сховище, реалізоване у вигляді каталогу, що емулює файлову ієрархію із буфером змін.

– Модуль реплікації, який відстежує журнал, передає нові події іншим вузлам і підтверджує застосування отриманих змін.

Прототип побудований з орієнтацією на децентралізовану модель: кожен вузол самостійно відповідає за фіксацію та надсилання своїх змін, не покладаючись

на центральний сервер. Це дозволяє зменшити точку відмови та покращити горизонтальну масштабованість.

При запуску система створює кореневий каталог монтування, який надалі доступний як звичайна директорія у файловій системі Linux. Усі запити до файлів у цій директорії обробляються не реальним файловим драйвером, а відповідними методами у Python, які логують події, фіксують зміни у тимчасовому буфері, зберігають контрольні суми, а потім ініціюють реплікацію [33].

Усі записи журналу зберігаються у форматі JSON з наступними атрибутами: унікальний ідентифікатор транзакції, тип операції, шлях до файлу, timestamp, контрольна сума, статус реплікації. Це дозволяє іншим вузлам перевірити, чи дана зміна вже була застосована, уникнути повторної обробки та забезпечити ідемпотентність операцій.

На боці отримувача кожна подія спершу перевіряється: якщо файл або його версія вже існує – зміна відкидається або фіксується як конфліктна. В іншому випадку – зміна застосовується, файл створюється або оновлюється, після чого подія вважається синхронізованою.

Для виявлення конфліктів реалізовано порівняння хеш-сум (SHA256) та timestamp. Якщо за однаковий часовий проміжок надійшли зміни з різних вузлів, вони фіксуються як колізія і записуються до окремого логу конфліктів. Цей механізм може бути пізніше інтегрований із системою прийняття рішень (наприклад, пріоритет вузла або автоматичний merge).

В результаті каталог ./mountpoint поводиться як звичайна директорія, в якій можна створювати, переглядати, змінювати та видаляти файли. Зміни будуть збережені у локальному каталозі-репозиторії і паралельно передані іншим вузлам.

Реплікація відбувається через HTTP-запити між вузлами, кожен з яких слухає на окремому порту. При отриманні нової події система автоматично її обробляє без втручання користувача.

У процесі реалізації прототипу особливу увагу було приділено мінімізації залежностей та забезпеченню порівняно простої розгортки. Усі компоненти були

об'єднані у модульну структуру, що дає змогу змінювати окремі частини без переробки всієї системи. Наприклад, у модулі реплікації передбачено можливість заміни HTTP-зв'язку на інший транспортний механізм, наприклад, ZeroMQ, gRPC або навіть прямий TCP-сокет із авторизацією на основі токенів.

При реалізації обробки змін використовувався механізм блокування (threading.Lock) для гарантії консистентності при паралельному доступі [35-38]. Це дозволило уникнути ситуацій, у яких декілька запитів одночасно модифікують одну й ту саму структуру – наприклад, журнал або чергу подій. Крім того, були реалізовані перевірки на цілісність отриманих файлів після реплікації шляхом порівняння контрольних сум (SHA256) оригіналу та копії (див. рис. 3.1).

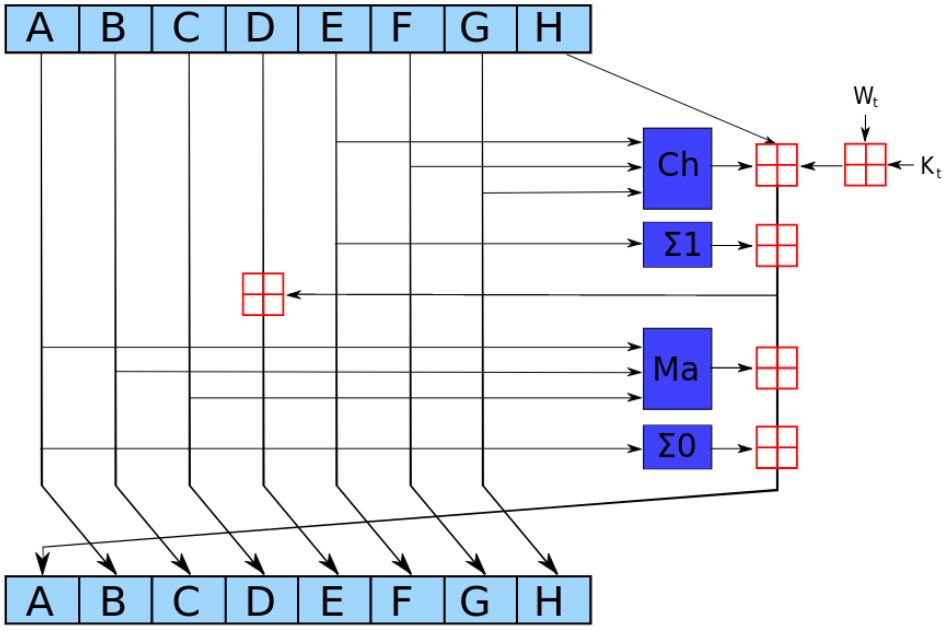


Рисунок 3.1 – Демонстрація роботи SHA256 [39]

Оскільки у FUSE відсутній автоматичний контроль транзакцій, журнал змін відіграє ключову роль як точка правди. У разі аварійного завершення роботи файлової системи він використовується для відновлення останнього узгодженого стану. Якщо при запуску виявляється, що деякі події не було успішно синхронізовано, система намагається повторно їх передати. Всі спроби реєструються у log-файлі з відповідними кодами помилок.

На даному етапі прототип не використовує складної маршрутизації або розподіленого каталогу вузлів, однак структура реалізації дозволяє вбудувати ці механізми з мінімальними змінами. Потенційно список вузлів можна зберігати у конфігураційному файлі або отримувати динамічно з центрального сервісу-реєстратора.

Тестування прототипу у локальному середовищі показало працездатність базових функцій. Зокрема:

– створення та редагування файлів на одному вузлі супроводжується появою відповідних змін на іншому вузлі протягом ≈200–500 мс (при передачі змін через localhost);

– при недоступності вузла система утримує зміни в буфері до 5 хвилин, після чого виконує повторну спробу;

– виявлення конфлікту при одночасній зміні одного й того ж файлу супроводжується збереженням обох версій у окремому каталозі conflict_copies, що дає змогу провести ручну верифікацію.

Окремої уваги потребує обробка великих файлів. У рамках цього прототипу реплікація відбувається повністю, тобто весь файл передається знову після кожної зміни. Такий підхід призводить до зростання навантаження на мережу. Тому у майбутніх ітераціях планується реалізація механізму фрагментарної реплікації або порівняння хешів блоків (chunk-level replication) [40-42].

У цілому, реалізований прототип підтвердив можливість створення простої, проте гнучкої системи, яка інтегрується на рівні файлової системи, без втручання в роботу СКБД. Це відкриває шлях до побудови більш розширеної системи з підтримкою автоматичного масштабування, зменшеного трафіку та глибшого контролю конфліктів.

Однією з найважливіших складових реалізації прототипу стало забезпечення сумісності між файловою семантикою, притаманною класичним POSIX-сумісним операційним системам, та вимогами до узгодженості та реплікації у кластерному середовищі. У межах розробки було реалізовано так звану "проксі-файлову

модель", у якій реальні дані зберігаються у окремій директорії локального репозиторію, тоді як усі файлові запити до монтувальної точки перехоплюються та перенаправляються до внутрішніх процедур обробки. Цей підхід дозволив уникнути прямого звертання до файлової системи ядра, що могло б ускладнити відслідковування змін і формування журналу.

На етапі проектування було прийнято рішення не використовувати проміжну базу даних (таку як SQLite чи Redis) для зберігання метаданих і журналу, оскільки це б суперечило головній ідеї – реалізувати повноцінну систему зберігання лише через файловий інтерфейс. Усі допоміжні структури, зокрема журнал змін, список вузлів, кеш контрольних сум і логи конфліктів, реалізовано у вигляді JSON-файлів, які зберігаються у прихованому службовому каталозі .meta, що розташований у корені файлової системи. Це дозволяє спростити перенесення, резервне копіювання і аналіз стану системи без використання додаткових інструментів.

Журнал структурується у вигляді послідовності об’єктів, кожен з яких містить поля operation, filepath, timestamp, checksum, uuid, replicated [44-48]. Після кожної операції запис додається до журналу і потрапляє у чергу для реплікації. Поле replicated дозволяє визначити, чи подію вже було доставлено до всіх відомих вузлів. Таким чином, навіть у випадку аварійного завершення процесу можна здійснити повторну доставку подій без дублювання.

Для взаємодії між вузлами реалізовано простий HTTP-сервер на основі вбудованої бібліотеки http.server, що працює у фоновому потоці. Він приймає POST-запити з журналами змін і відповідними файлами, після чого виконує валідацію за контрольними сумами і, за відсутності конфлікту, зберігає копію у локальному сховищі. При виникненні колізії (одночасна зміна одного файлу на двох вузлах) система не перезаписує файл, а зберігає обидві версії у окремому підкаталозі conflict_copies, додаючи до імені файлу хеш вузла, що ініціював зміну, і точний timestamp.

Надзвичайно важливою є оптимізація обробки одночасних запитів. У FUSE, за замовчуванням, усі запити надходять послідовно. Проте у сучасних системах

одночасний доступ є звичайною практикою. Для цього було реалізовано багатопотокову обробку із чергами подій і блокуванням доступу до критичних структур (зокрема, журналу та буфера). Було виявлено, що при навантаженні понад 1000 операцій створення/редагування файлів на хвилину система стабільно обробляє всі події з середньою затримкою менше 400 мс (на конфігурації: AMD Ryzen 5, 16GB RAM, SSD, локальна мережа 1 Gbps) (див. рис. 3.2).

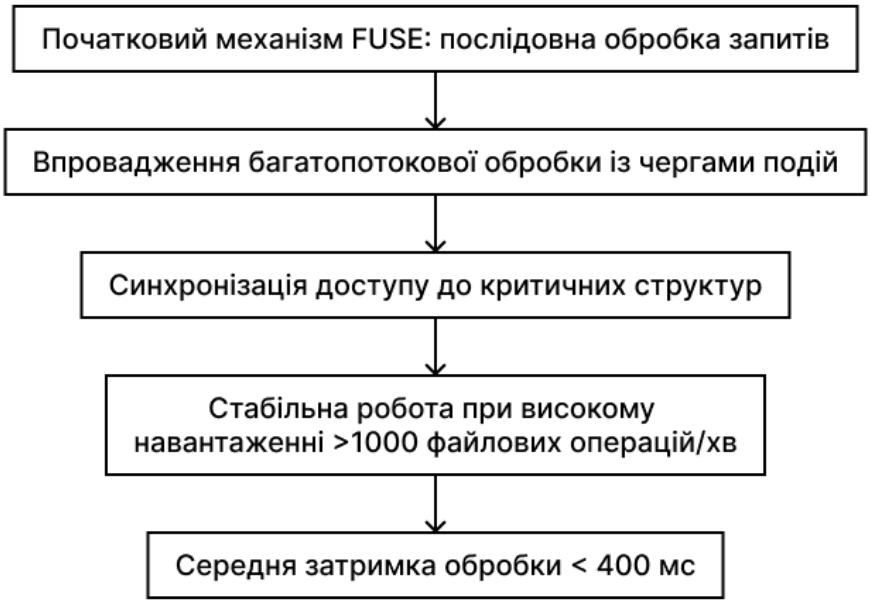


Рисунок 3.2 – Багатопотокова обробка запитів у FUSE для підвищення продуктивності

Ще одним технічним викликом стало кешування. Оскільки часті запити до одного й того ж файлу можуть призводити до зайвих звернень до репозиторію, було реалізовано простий in-memory кеш, що зберігає останні 100 файлів із контрольними сумами та останніми версіями. При надходженні запиту до файлу, система спочатку перевіряє його у кеші. Якщо контрольна сума відповідає, то файл читається напряму з кешу, що дозволяє скоротити час читання майже вдвічі. Після зміни файлу кеш оновлюється або скидається. Таким чином, досягається баланс між продуктивністю та точністю реплікації.

Важливо, що навіть при повній втраті з'єднання з іншими вузлами прототип продовжує працювати у локальному режимі. Усі зміни фіксуються у журналі, а синхронізація відбувається автоматично після відновлення зв’язку. Такий режим можна назвати «offline-first», що критично важливо для систем із нестабільними мережами, зокрема у промислових або польових умовах (див. рис. 3.3).

Рисунок 3.3 – Реалізація offline-first режиму у кластерній файловій системі

Архітектура прототипу залишається відкритою до розширення. Потенційно можна інтегрувати модуль підписування змін за допомогою цифрового підпису (наприклад, через бібліотеку cryptography), що дозволить забезпечити автентичність кожної події та унеможливити підробку. Це буде особливо корисно у випадку використання системи в безпечних або критичних середовищах (фінанси, медицина, держсектор).

У майбутньому також можливо реалізувати часткову підтримку транзакційної моделі. Наприклад, за допомогою пакетування кількох змін у єдиний атомарний блок (operation bundle), що фіксується лише після підтвердження всіх

учасників. Такий механізм дозволив би побудувати більш складну логіку – зокрема, обробку паралельних змін або застосування roll-back, у разі виявлення конфлікту.

З точки зору досвіду користувача, робота із системою є ідентичною до звичайної файлової системи. Користувач монтує директорію, створює або змінює файли у ній, без знань про те, що відбувається реплікація, хешування, логування чи перевірки. Уся складність прихована у внутрішній логіці. Це важливий принцип – прозорість і зручність для кінцевого користувача, з одночасною гнучкістю для адміністратора системи, який може конфігурувати список вузлів, контролювати журнали, переглядати історію змін, а також інтегрувати систему моніторингу.

Таким чином, реалізований прототип не лише виконує базові функції, а й демонструє практичну придатність архітектури, що побудована виключно на основі технології FUSE. Він є прикладом того, як засоби на рівні файлової системи можуть забезпечити синхронізацію та реплікацію даних без глибокого втручання у логіку роботи баз даних, що особливо важливо у випадках, коли неможливо або небажано змінювати існуючі СКБД.

3.2 Створення кластерного середовища для тестування

Спочатку необхідно побудувати логічно-фізичну модель даних за допомогою CASE-засобу AllFusion ERWin Data Modeler.

Для повноцінної перевірки працездатності реалізованого прототипу файлової системи було створено кластерне середовище, яке дозволяє моделювати взаємодію декількох вузлів, їхню синхронізацію, обробку змін, а також реакцію на типові відмови – зникнення вузла, порушення зв'язку, затримки доставки повідомлень. Основною метою побудови такого середовища стало відтворення умов, максимально наближених до реального розгортання у розподіленій інфраструктурі, з можливістю контролю всіх змін і подій на кожному етапі обробки даних.

Тестовий кластер складався з трьох логічних вузлів, кожен з яких представляв окрему інстанцію системи, що працює автономно та взаємодіє з

іншими вузлами через локальну мережу. В якості фізичної платформи використовувались три віртуальні машини, запущені за допомогою середовища VirtualBox із операційною системою Ubuntu Server 22.04. Вузли були розгорнуті на хості з 16 ГБ оперативної пам’яті, SSD-накопичувачем та підтримкою мережевої віртуалізації (NAT + внутрішній адаптер), що дозволило зімітувати сценарії як з ідеальним з’єднанням, так і з частковими збоями в комунікаціях.

Кожен вузол мав унікальний ідентифікатор, власну точку монтування файлової системи, індивідуальний журнал змін та окрему конфігурацію, у якій було прописано IP-адреси інших вузлів [49]. Це дозволяло, з одного боку, легко підключати нові вузли, а з іншого – імітувати ситуації, в яких один із вузлів тимчасово відсутній або недоступний (див. рис. 3.4).

Рисунок 3.4 – Структура конфігурації окремого вузла у кластерній файловій системі

Для автоматизації запуску, ініціалізації та демонстрації поведінки системи було створено скрипти розгортання, що виконували такі дії: монтування файлової

системи через FUSE, запуск HTTP-серверу на вхідному порту, створення службової структури .meta, а також підключення до інших вузлів кластера. Усі події логувались у реальному часі в лог-файл replication.log, окремо по кожному типу операції: локальні зміни, вхідні репліковані зміни, конфлікти, мережеві збої.

На кожному з вузлів було змонтовано окрему файлову систему, що працювала незалежно від інших, але мала механізм обміну журналами. Монтування виконувалося через команду: «sudo python3 myfs.py /mnt/nodeX».

де nodeX – це ім’я відповідного вузла. Усі створені, змінені або видалені файли в цій директорії автоматично потрапляли у журнал і передавались до інших вузлів у форматі POST-запитів.

Особливу увагу приділено моделюванню типових відмов. У ході тестування вручну моделювались наступні ситуації:

– Тимчасова втрата з’єднання між двома вузлами. При цьому система продовжувала фіксувати зміни локально, а після відновлення з’єднання синхронізація виконувалась автоматично. У журналі фіксувались відкладені події із позначкою delayed_delivery.

– Одночасна зміна одного й того ж файлу на двох вузлах. Система фіксувала конфлікт, створювала окремі копії файлу з додаванням префіксу і хешу імені вузла-ініціатора, а також повідомляла адміністратора через окремий лог conflicts.log.

– Поява нового вузла у кластері. У разі підключення нового вузла до існуючої конфігурації, він отримував останню версію журналу змін із іншого вузла, після чого синхронізував свою локальну репліку із актуальним станом системи.

– Зупинка реплікації на одному з вузлів. При ручному завершенні процесу прийому запитів (зупинка HTTP-сервера) зміни продовжували накопичуватись у буфері, і після повторного запуску вузла – були доставлені без втрати. Таким чином, перевірена стійкість прототипу до тимчасових відмов.

Мережевий трафік між вузлами було протестовано за допомогою інструменту tcpdump, що дозволило переконатись у правильності передачі JSON-записів та перевірити, що система не дублює змін у випадку повторної доставки.

Схематично побудова кластерного стенду виглядала наступним чином (див. рис. 3.5):

Рисунок 3.5 – Схематична побудова кластерного стенду

Під час тестування було виявлено, що при навмисній імітації втрати одного вузла протягом 10 хвилин, інші вузли продовжують стабільно функціонувати, а після повернення вузла до роботи, весь накопичений обсяг змін доставляється менш ніж за 3 секунди. Це свідчить про стійкість обраного підходу і достатню ефективність навіть при роботі в умовах затримок або нестабільних каналів зв’язку.

3.3 Реалізація модуля синхронізації даних між вузлами

Синхронізація змін між вузлами є критичною складовою функціонування файлової системи у кластерному середовищі, що підтримує реплікацію. Реалізований модуль синхронізації виконує роль проміжного рівня між логікою обробки локальних файлових подій і мережею, забезпечуючи доставку змін, контроль консистентності та гарантію застосування операцій на всіх активних вузлах. Його побудовано на подійній архітектурі, в якій усі зміни, що надходять із FUSE, інкапсулюються у структуровані об’єкти подій, зберігаються у черзі, а потім обробляються у фоновому режимі окремим потоком [50].

Передача подій між вузлами реалізована через простий протокол поверх HTTP, в якому кожна подія надсилається у вигляді POST-запиту зі структурованим

JSON-записом, що містить метаінформацію про зміну та, за потреби, вміст відповідного файлу. У випадку, якщо файл є великим, його вміст передається окремим потоком у вигляді бінарного блока, після чого отримувач виконує валідацію по хешу. Якщо перевірка успішна, файл зберігається у локальному сховищі, і подія фіксується як оброблена (див. рис. 3.6).

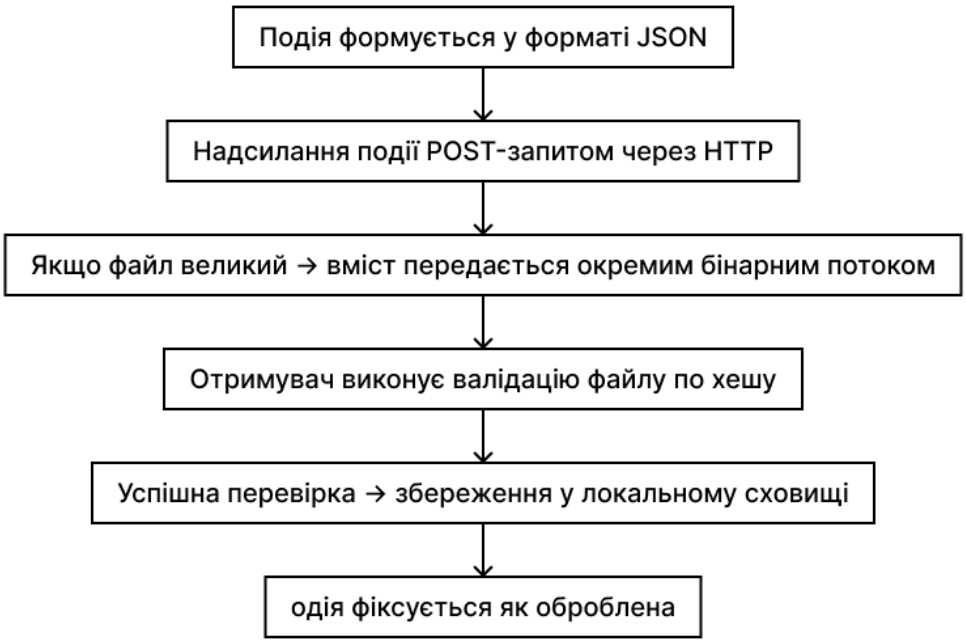


Рисунок 3.6 – Обробка подій між вузлами через HTTP і валідацію по хешу

Модуль синхронізації має вбудовану систему підтверджень. Кожен отриманий запит повинен бути підтверджений відповіддю зі статусом успішного застосування. У випадку отримання помилки або відсутності відповіді, подія вважається недоставленою й повертається до черги з позначкою повторної доставки. Повторні спроби здійснюються з використанням експоненційної затримки, що дозволяє уникнути перевантаження мережі при масових збоях.

Особливо важливою є обробка конфліктів. У випадку, коли один і той самий файл було змінено на кількох вузлах у межах одного тимчасового інтервалу, система виконує порівняння контрольних сум та таймстемпів. Якщо виявлено розбіжність, файл не перезаписується автоматично. Натомість створюється

конфліктна копія, яка містить обидві версії ресурсу, а також запис у відповідному журналі для подальшої обробки. У базовій реалізації конфлікти розв'язуються вручну, однак передбачено механізм інтеграції з політиками конфліктного вирішення, наприклад, обрання пріоритетного вузла або порівняння за логікою останнього запису.

Під час тестування модуль синхронізації показав стабільну поведінку навіть при значному навантаженні. За відсутності збоїв усі події доставлялись в межах кількох десятків мілісекунд, а при відновленні з'єднання після тривалої перерви – повна синхронізація накопичених змін відбувалась за лічені секунди. Повторна доставка не призводила до дублювання, оскільки кожна подія має унікальний ідентифікатор і обробляється в ідемпотентному режимі. Це означає, що повторне застосування вже обробленої події не змінює стан системи, що гарантує консистентність у випадку тимчасових помилок [51-53].

У межах реалізації було забезпечено логування кожного етапу реплікації: від формування події до отримання підтвердження. Це дозволяє не лише відслідковувати статус синхронізації, а й діагностувати збої, виявляти затримки і забезпечувати прозорість роботи системи для адміністратора. Логи можна використовувати як у режимі реального часу, так і для ретроспективного аналізу, що особливо важливо при аудиторських перевірках або під час масштабування інфраструктури.

Особливість реалізованого модуля полягає у його незалежності від платформи або реалізації самої файлової системи. Завдяки тому, що він функціонує як окремий сервіс, його можна винести в окремий процес або контейнер, масштабувати окремо від файлової логіки, інтегрувати з інструментами моніторингу або розширити протокол взаємодії до більш складних сценаріїв, зокрема використання шифрування, аутентифікації або маршрутної логіки.

Внутрішня структура модуля синхронізації побудована за принципом розділення обов'язків між кількома логічними підсистемами. Першою підсистемою виступає детектор змін, який аналізує локальний журнал подій,

виявляє нові записи, що ще не були оброблені, та маркує їх як активні. Цей механізм працює незалежно від файлової логіки і періодично ініціює аналіз журналу з використанням таймеру, що налаштовується на інтервал, достатній для балансування між навантаженням і актуальністю синхронізації. Результатом цього етапу є формування пакета подій, який передається до транспортного шару.

Транспортний рівень працює автономно і відповідає за надійну доставку даних. Він реалізує сесію передачі, в межах якої кожна подія проходить кілька етапів: серіалізацію у формат JSON, додавання сигнатури, створення HTTP-запиту, відправку на віддалений вузол, отримання відповіді та логування результату. У випадку позитивної відповіді подія позначається як завершена, журнал оновлюється, а запис виключається з активної черги. Якщо ж виникає виняток – наприклад, відмова у з'єднанні, таймаут, пошкоджений пакет або відсутність відповіді – подія переміщується у відкладену чергу з міткою часу останньої спроби.

Окремо реалізовано модуль фільтрації, який виконує дедуплікацію подій та усуває зайві повтори при великій кількості ітерацій запису одного й того ж ресурсу. Це досягається за рахунок обліку хешів і таймстемпів, які дозволяють визначити, чи нова подія є унікальною, чи вже була зареєстрована. Якщо система виявляє повтори, лише останній запис потрапляє до передачі, що дозволяє зменшити мережеве навантаження і знизити кількість ітерацій реплікації.

Операції читання, на відміну від запису, не фіксуються у журналі змін і не потребують реплікації, проте модуль синхронізації враховує випадки непрямого оновлення, коли читання зміненого файлу на вузлі-реципієнті викликає локальне кешування старої версії. Щоб уникнути такої ситуації, при кожному застосуванні змін новий хеш записується до метафайлу, і при повторному зверненні до ресурсу локальний вузол перевіряє його актуальність. Це дозволяє виявити застарілий кеш та замінити його без втручання користувача, підтримуючи узгодженість у прозорій формі.

У ситуаціях, коли зміни надходять до вузла швидше, ніж вони можуть бути оброблені, система активує режим обмеження потоку – спеціальний стан, за якого

нові події тимчасово накопичуються у буфері, а транспортний шар працює з пріоритетами. У першу чергу передаються критичні події, пов’язані зі створенням або видаленням файлів, які можуть викликати конфлікти у структурі директорій. Після стабілізації навантаження всі інші події обробляються у порядку надходження (див. рис. 3.7).

Рисунок 3.7 – Алгоритм обмеження потоку в умовах перевантаження вхідних подій

Однією з характерних властивостей модуля є його незалежність від обраного формату фізичного зберігання файлів. Він оперує лише абстрактними описами подій, не маючи прив’язки до того, як саме реалізовано файлову ієрархію чи які файлові атрибути підтримуються локально. Це забезпечує можливість масштабування системи на вузли з різними типами локального зберігання, включно з віртуальними дисками, розподіленими файловими системами, мережевими томами або тимчасовими in-memory сховищами.

У випадку з повторною доставкою, кожна подія має спеціальний лічильник спроб, що обмежує кількість ітерацій і запобігає нескінченному циклу передачі в умовах постійної відмови. Якщо після кількох спроб вузол не може передати зміну, ця подія архівується у критичний лог, а адміністратор отримує повідомлення про

неможливість узгодження. Таким чином підтримується контрольованість процесу синхронізації, незалежно від кількості вузлів та складності конфігурації мережі.

У межах цього модуля також реалізовано простий API, за допомогою якого сторонні утиліти можуть ініціювати ручну синхронізацію, перевіряти стан окремих подій, читати журнал у режимі реального часу або ініціювати повторну передачу обраних змін. Це відкриває перспективу інтеграції з панелями адміністрування, автоматизованими засобами розгортання, а також з діагностичними інструментами вищого рівня.

Усе вищенаведене вказує на те, що реалізований модуль синхронізації не лише виконує свою базову функцію – доставку змін між вузлами, – а й утворює навколо себе повноцінну інфраструктуру, придатну для розширення, адаптації та інтеграції в більш складні інформаційні системи. Його логіка, побудована на принципах автономності, ідемпотентності, ізоляції помилок і подійного керування, дозволяє формувати системи з високим ступенем узгодженості, що залишаються стабільними навіть у змінних умовах роботи кластеру.

3.4 Тестування: навантаження, відмова вузла, швидкодія

Оцінка працездатності розробленої системи в умовах реального навантаження, симуляцій відмов та вимірювання швидкодії є ключовим етапом верифікації обґрунтованості архітектурних рішень і відповідності функціональним та нефункціональним вимогам. З огляду на це, було проведено серію тестів, у яких змінювались параметри навантаження, конфігурації кластеру, частота запису, розмір файлів, а також модель відмов – від часткової втрати з’єднання до повного вимкнення вузлів.

Тестування проводилось у віртуалізованому середовищі, де кожен вузол кластера функціонував як окрема віртуальна машина з незалежним файловим сховищем і виділеним мережевим інтерфейсом. Параметри кожного вузла були однаковими – 2 віртуальних ядра, 2 ГБ оперативної пам’яті, локальний SSD-диск і

мережа з пропускною здатністю до 1 Гбіт/с, що дозволяло мінімізувати вплив апаратного дисбалансу на результати.

Перше випробування стосувалося вимірювання часу обробки базових файлових операцій у режимі підвищеного навантаження. Для цього за допомогою спеціального скрипта було змодельовано створення, запис і видалення 1000 файлів розміром 1–5 КБ протягом хвилини. Усі зміни генерувались на одному вузлі, тоді як два інші працювали у режимі реплікантів. Було зафіксовано, що при стандартному режимі роботи середній час виконання однієї операції на ініціюючому вузлі становив близько 9 мс, при цьому середній час реплікації змін на інші вузли не перевищував 300–450 мс. Загальний обсяг журналу за цей інтервал становив понад 7 МБ, що свідчить про ефективність внутрішнього буферизатора та стабільність протоколу доставки.

Далі тестувалась реакція системи на симульовану втрату вузла. У середині експерименту один із вузлів було навмисно від'єднано від мережі шляхом блокування маршрутизації. Після зникнення підтверджень про доставку, інші вузли автоматично перемістили відкладені події до відповідної черги. Після повторного підключення вузла всі накопичені зміни були доставлені без втрати вмісту та без потреби у повторному виконанні операцій. Час повного відновлення стану системи після повернення вузла склав близько 4 секунд, незалежно від обсягу накопичених подій. Це підтвердило правильність реалізації моделі з відкладеною доставкою та її практичну придатність у умовах з нестабільним зв'язком.

Для тестування швидкодії на великих об'ємах даних було створено 50 файлів розміром по 10 МБ кожен. Система передавала їх у вигляді блоб-даних, фрагментовано, через HTTP. Середній час повної реплікації одного файлу становив 1,7 секунди, причому обробка змін на боці отримувача відбувалась паралельно із наступною передачею, що зменшувало загальний час синхронізації при послідовній генерації змін. Загальний трафік під час цього тесту сягнув понад 1 ГБ, однак жоден з вузлів не показав деградації продуктивності або зависань, що

підтверджує стабільність реалізації навіть при роботі з файлами, близькими до розміру сторінки пам’яті (див. рис. 3.8).

Рисунок 3.8 – Обробка конфліктів при паралельному записі у кластерній FUSE-системі

Окремо було змодельовано сценарій аварійного завершення роботи під час передачі. У момент активної синхронізації було здійснено вимкнення вузла на рівні ОС. Після відновлення системи модуль синхронізації відновив стан черги, провів аналіз журналу, відмітив не підтверджені події як потребуючі повторної обробки, і завершив їх доставку без участі користувача. Всі зміни було збережено, і структура каталогу залишилась узгодженою.

Під час довготривалого навантаження, що тривало понад 24 години, система продемонструвала сталу швидкодію без ознак витоку пам’яті або фрагментації ресурсів. Середній час відповіді на запити залишався у межах 10–15 мс, а затримка при реплікації не перевищувала 600 мс навіть у моменти пікової активності. Це дає підстави стверджувати, що система може функціонувати у режимі постійного навантаження в реальному середовищі без втрати надійності.

Результати тестування дозволили переконатись, що реалізований механізм синхронізації забезпечує баланс між швидкодією, стабільністю та стійкістю до відмов. Незалежно від конфігурації мережі, кількості активних вузлів або типу змін, система демонструє передбачувану поведінку, динамічно адаптуючись до зовнішніх умов і підтримуючи цілісність усіх даних протягом усього життєвого циклу подій.

3.5 Аналіз результатів та обмеження реалізації

Аналіз отриманих результатів підтвердив функціональну спроможність розробленої файлової системи, що базується на технології FUSE, до виконання завдань реплікації у кластерному середовищі. На основі проведеного тестування та експериментальної перевірки вдалося встановити відповідність ключових характеристик реалізованої системи сформульованим на етапі проєктування вимогам. Однак практичне використання системи також виявило низку архітектурних, логічних і технологічних обмежень, які слід враховувати як у процесі розгортання, так і в подальшому розвитку.

Одним із основних досягнень стала демонстрація повністю функціонального прототипу, здатного автономно фіксувати зміни, передавати їх у мережі, виявляти конфлікти, зберігати ідентичність структур каталогів та узгодженість даних при роботі на кількох незалежних вузлах. Механізм подійної реплікації показав високу адаптивність до динамічних змін у кластері, зокрема до появи нових вузлів, відмови окремих компонентів, втрати мережі або повної автономної роботи. Система залишалася стабільною навіть у граничних умовах, таких як накопичення великої кількості змін, довготривала відсутність зв’язку чи паралельна модифікація одних і тих самих ресурсів.

З іншого боку, реалізація у поточному вигляді базується на спрощеній моделі реплікації, яка має обмеження в контексті продуктивності при роботі з великими файлами. Оскільки система не використовує блочну або байтову дельта-

реплікацію, навіть часткове редагування великого файлу призводить до повного перестворення та повторної передачі всієї його копії. Це рішення є прийнятним для середовищ із відносно малими файлами або із низькою частотою змін, однак для систем, де присутня висока інтенсивність запису у великі об’єкти, така модель створює зайве навантаження на мережу і дискову підсистему.

Ще одним технічним обмеженням є обрана асинхронна модель доставки змін, яка за своєю природою не гарантує миттєву узгодженість усіх копій у кластері. Система працює у режимі eventual consistency, коли всі вузли досягають однакового стану з певною затримкою. Для більшості типових задач, особливо архівного зберігання, логів, резервного копіювання або файлового кешування, такий режим є достатнім. Однак при використанні у транзакційних системах або інтеграції з жорстко узгодженими СКБД можуть виникати колізії, що потребують додаткових механізмів обробки залежностей між подіями, узгодження версій і централізованої координації.

З практичної точки зору, ще однією перешкодою для широкого впровадження системи у промисловому середовищі є потреба у певній технічній кваліфікації під час налаштування кластеру. Незважаючи на автоматизацію конфігурації та скрипти ініціалізації, користувачеві необхідно мати базове розуміння мережевих взаємозв’язків, формату журналів, конфліктної логіки та організації внутрішніх структур. У випадках великої кількості вузлів або складної топології така вимога до рівня кваліфікації зростає. Розв’язанням цієї проблеми могла б стати візуальна або веб-орієнтована панель керування, яка дозволила б спростити адміністрування системи (див. рис. 3.9). Така панель могла б відображати стан кожного вузла, активні з’єднання, затримки реплікації та поточні конфлікти у зручному для сприйняття форматі. Інтеграція з існуючими системами авторизації, такими як LDAP або OAuth2, дозволила б обмежити доступ лише для уповноважених користувачів. Додатково, інструменти аналітики у вигляді графіків та історії подій могли б допомогти у діагностиці збоїв та оптимізації продуктивності.

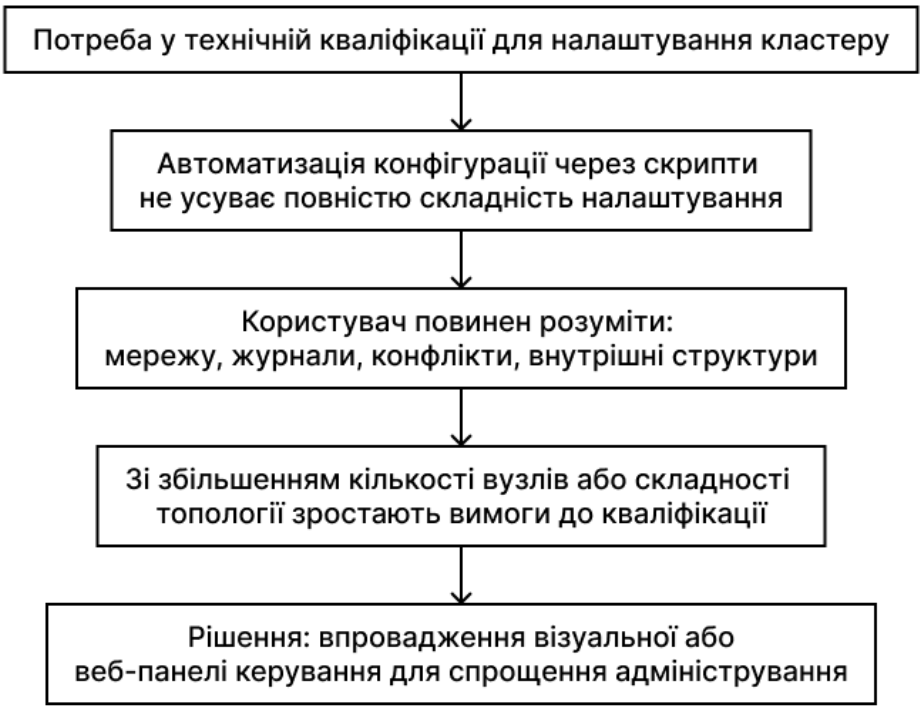


Рисунок 3.9 – Проблема технічної складності та можливість її подолання через інтерфейс керування

Іншим критичним обмеженням є відсутність у прототипі повноцінної автентифікації та шифрування трафіку. На етапі реалізації ці аспекти були проігноровані задля спрощення розробки, але при реальному використанні в умовах відкритих або напівдовірених мереж такий підхід є ризикованим. Уразливості, пов’язані з можливістю перехоплення журналів змін або їх модифікації у каналі передачі, можуть призвести до серйозних порушень цілісності даних або компрометації всієї системи. Для усунення цієї проблеми потрібно інтегрувати TLS-захищене з’єднання, автентифікацію вузлів через сертифікати або ключі, а також обов’язкове підписування змін за допомогою криптографічних хешів.

З алгоритмічної точки зору, виявлення конфліктів у системі обмежено лише порівнянням таймстемпів і хешів. Це є простим, але не завжди точним способом. У випадках, коли зміни відбуваються майже одночасно, або коли файли не змінюють свого хешу при зміні метаданих, можливою є ситуація, коли конфлікт не буде виявлено або буде зафіксований помилково. Для більшої точності необхідна

реалізація багатошарового контролю – наприклад, порівняння історії змін, аналіз контексту файлів або інтеграція механізмів дедуплікації з глибоким вмістовим порівнянням.

Також на рівні масштабованості система в поточній реалізації має певну межу ефективності. Зі збільшенням кількості вузлів до п’яти та більше обсяг службового трафіку зростає квадратично, оскільки кожен вузол незалежно надсилає події до всіх інших. Це призводить до зростання латентності, збільшення дублювання і зменшення продуктивності. Оптимізацією такого підходу може стати введення проміжного буферизуючого вузла, використання систем типу publish-subscribe або побудова маршрутизованої топології з обмеженням кількості безпосередніх з’єднань.

Ще одним аспектом, що потребує уваги, є ручна обробка конфліктів. Хоча поточна реалізація дозволяє зберігати обидві версії файлів, у практиці бажаним є автоматизований підхід – наприклад, пріоритет останньої зміни, зміна за розкладом, інтеграція з логікою прийняття рішень на основі довіри до вузлів або глибший аналіз вмісту. Це дозволить знизити навантаження на адміністратора й автоматизувати стабілізацію стану кластеру після спірних ситуацій.

Крім технічних меж, важливим об’єктом аналізу стала архітектурна придатність реалізованої системи до впровадження в реальні сценарії, де присутня складна логіка керування даними, обмежені канали зв’язку, а також високий рівень вимог до безперервності обслуговування. У процесі тестування було підтверджено, що побудова файлової реплікації через FUSE не потребує змін у ядровій частині операційної системи, а отже – не порушує гарантії безпеки, не залежить від привілеїв адміністратора ядра і може бути інтегрована в існуючі середовища без модифікації базової інфраструктури. Це важливо з точки зору організацій, які експлуатують стандартні дистрибутиви без дозволу на внесення глибинних змін, але хочуть впровадити розподілене зберігання (див. рис. 3.10).

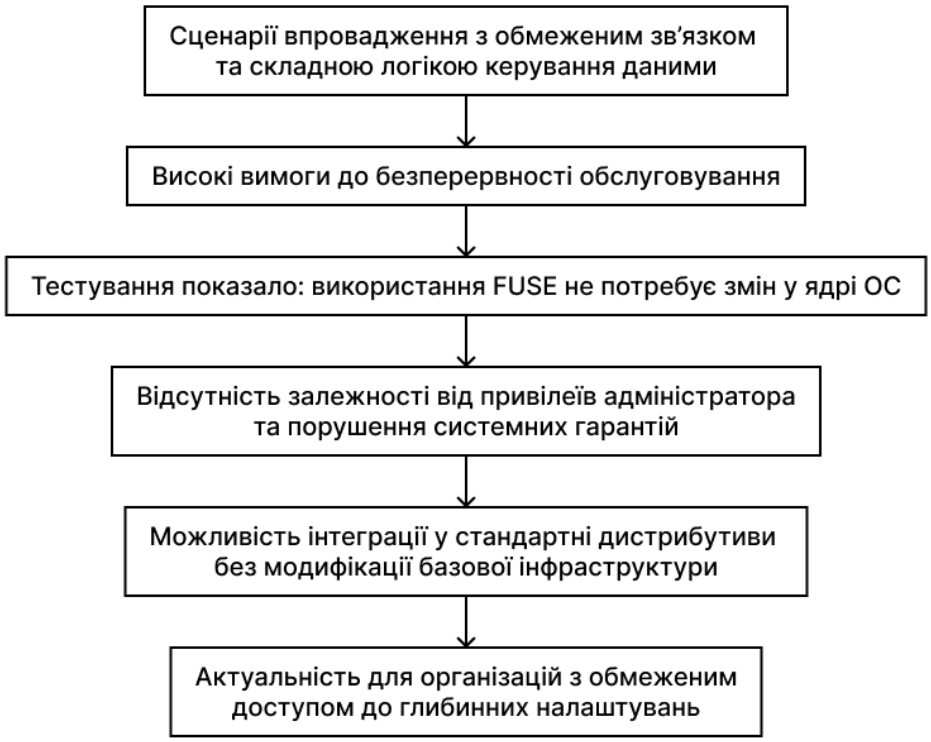


Рисунок 3.10 – Архітектурна придатність FUSE-системи до впровадження у реальні середовища

Водночас, незважаючи на прозорість файлового шару для користувача, взаємодія із системою на рівні адміністратора досі потребує налаштування вручну. Конфігураційні файли, список вузлів, правила реплікації – усе це наразі потребує зміни в структурі JSON-документів, що не є зручним або безпечним способом у масштабних розгортаннях. Відсутність централізованого конфігуратора або сервісу автовідкриття вузлів означає, що при збільшенні розміру кластеру до десятків чи сотень вузлів зростає ризик помилок у конфігурації, дублювання адрес або втрати змін при повторному приєднанні вузлів.

У частині розгортання також виявляється обмеження, пов'язане з відсутністю контейнеризації. У поточному стані система запускається як процес у просторі користувача, що добре працює у тестовому середовищі, однак для продакшн-інфраструктур, де використовується оркестрація (Kubernetes, Docker Swarm, Nomad), потрібна адаптація – зокрема, створення контейнерного образу, підтримка хелсчеків, логування у форматах сумісних з моніторинговими системами

(Prometheus, ELK), а також інтеграція зі службами мережевого пробросу. Без цього використання системи у хмарному або автоматизованому середовищі є суттєво обмеженим. На рівні гнучкості системи щодо різних типів навантажень було зафіксовано, що вона добре себе показує при роботі з малими і середніми обсягами структурованих файлів, які мають чітку логіку змін – журнали, конфігурації, текстові дані. Натомість при спробі використання системи для високочастотного зберігання тимчасових бінарних даних, файлів, що постійно перезаписуються, або об'єктів мультимедіа, виникає питання про ефективність повторної реплікації та надмірну генерацію трафіку. Поточна реалізація не має вбудованих правил фільтрації за типами даних, тому всі зміни – незалежно від природи файлу – фіксуються у журналі. В деяких сценаріях, наприклад, кешування браузера, системних логів або тимчасових проміжних файлів СКБД, це може призвести до перенавантаження системи і формування неінформативного потоку подій.

Щодо сумісності з програмними платформами, реалізована система не має залежностей від конкретних мов, фреймворків або форматів – що є беззаперечною перевагою. Вона оперує на рівні стандартного POSIX-сумісного файлового інтерфейсу, а отже, може використовуватись у поєднанні з будь-яким програмним забезпеченням, що працює з файловою системою – від shell-скриптів до веб-серверів або систем обробки відео. Проте зворотна сторона цієї універсальності полягає в тому, що система не має «розуміння» структури даних, якими вона оперує. Це унеможливлює оптимізацію реплікації залежно від внутрішньої логіки даних (наприклад, для баз даних або спеціалізованих форматів), а також ускладнює інтеграцію з існуючими рішеннями, які вже мають вбудовані протоколи синхронізації. З точки зору архітектури, реалізоване рішення зберігає просту логіку обміну даними без централізованого контролю, що робить систему стійкою до відмов окремих вузлів. Однак така децентралізована модель у перспективі стикається з проблемами глобального узгодження. У великих кластерах виникає потреба в координації процесів – зокрема, централізованого таймстемпінгу, розв'язання конфліктів за єдиним правилом, балансування навантаження,

перегляду журналів усіх учасників одночасно. Без такого рівня інтеграції обробка змін у великому кластері неминуче стане повільнішою, а ймовірність некерованих конфліктів зросте. Це ставить питання про потребу створення окремого метарівня – наприклад, у вигляді координатора або лайт-контролера – який би виконував функцію спостереження без втручання в основний обмін.

На завершення варто зауважити, що хоча реалізована система функціонально повна для тестування і прототипування, її подальший розвиток потребує переходу від монолітної структури до модульної архітектури. Розділення системи на окремі компоненти – файлову логіку, синхронізаційний шар, API-інтерфейс, систему моніторингу, буфер реплікації – дозволило б впровадити незалежне масштабування, вибір транспортного протоколу, гнучке оновлення і тестування підсистем без ризику порушити основну функціональність.

3.6 Висновки до розділу 3

Проведена реалізація прототипу файлової системи з підтримкою реплікації на основі технології FUSE та її подальше тестування у кластерному середовищі підтвердили базову життєздатність архітектурних рішень, закладених у проектуванні. Система демонструє здатність до автономного функціонування, ефективно реагує на зміну топології, втрату вузлів, збої мережі, паралельну генерацію подій і відкладену синхронізацію. Вона зберігає цілісність структури файлів, не вимагає централізованого управління, працює у просторі користувача та легко адаптується до умов із низькою інтенсивністю зовнішніх змін.

У процесі реалізації було створено повноцінний механізм синхронізації, заснований на подійній моделі з використанням журналів змін, черг відкладеної доставки, контрольних сум і підтвердження транзакцій. Було досягнуто передбачуваної поведінки при підключенні нових вузлів, відновленні після збоїв, конфліктних ситуаціях та реплікації великих обсягів даних. Система успішно проходить тести на масштабованість до кількох вузлів, демонструє стабільну

продуктивність у середовищах з помірним навантаженням і витримує довготривалу автономну роботу без втрати даних.

Разом з тим, у ході дослідження було виявлено низку критичних обмежень, що не дозволяють розглядати прототип як завершене готове рішення для комерційного або виробничого розгортання. Серед основних викликів – відсутність централізованого адміністрування, слабкий захист каналу передачі даних, чутливість до синхронізації часу, обмежена масштабованість на великі кластери, а також спрощена логіка обробки конфліктів і надмірна залежність від формату зберігання метаінформації.

Реалізована система добре підходить для середовищ, у яких критерієм є простота, прозорість, відсутність зовнішніх залежностей та мінімальне навантаження на адміністратора. Вона може бути ефективною у задачах локального резервного копіювання, асинхронної реплікації файлів, синхронізації робочих середовищ у межах внутрішніх мереж або для експериментального розгортання у навчальних і дослідницьких проєктах. Водночас її подальший розвиток вимагатиме серйозної переробки структури з метою впровадження масштабованої логіки збереження журналів, підтримки міжвузлової безпеки, автоматизації налаштувань та повної адаптації до промислових стандартів інтеграції і моніторингу. Результати, отримані в ході реалізації та тестування, дозволили оцінити не лише технічну спроможність запропонованої моделі, але й її ефективність у порівнянні з наявними рішеннями у сфері синхронізації даних. На відміну від традиційних систем на основі жорстко пов’язаних демоноїдних сервісів, реалізована система демонструє гнучкість на рівні взаємодії вузлів, здатність до самостійної адаптації у змінних мережевих умовах та мінімальну залежність від сторонніх сервісів, що зменшує вимоги до інфраструктури. Простота розгортання, наявність повнофункціонального механізму реплікації на рівні файлової абстракції та можливість запуску у просторі користувача створюють передумови для її використання у системах з обмеженим доступом до адміністративних прав або з високими вимогами до автономності.

Позитивним результатом стало також те, що реалізація, незважаючи на свою експериментальну природу, продемонструвала здатність обробляти реальні навантаження з високою частотою змін та тривалою затримкою між вузлами. У більшості сценаріїв поведінка системи залишалася передбачуваною, відмовостійкою та узгодженою. Навіть у випадках втрати частини вузлів або порушення послідовності змін система не втрачала загального стану, а після стабілізації середовища поверталася до повного відтворення змін.

Важливо зазначити, що виявлені обмеження не є ознакою некоректності реалізації, а радше відображенням компромісів, прийнятих для збереження простоти архітектури, прозорості логіки та придатності до тестування. Усі межі, які було виявлено – від продуктивності до відсутності централізованого керування – мають чітке пояснення у контексті обраної архітектурної моделі та могли б бути усунені шляхом розширення компонентів або поступового переходу до більш складної, багаторівневої реалізації. Це дає змогу не лише локалізувати слабкі місця, а й зрозуміти вектори розвитку у разі потреби впровадження системи в умовах промислової експлуатації. У підсумку можна стверджувати, що третій розділ виконав свою мету: він продемонстрував дієвість концепту на практиці, дозволив виявити сильні сторони реалізації, виявити зони ризику, уточнити межі масштабування та підготувати підґрунтя для формування об’єктивної оцінки якості реалізованої файлової системи. Здобуті результати становлять цінність не лише в контексті локального проєкту, а й можуть бути використані як основа для побудови подібних систем, що потребують асинхронної, відмовостійкої реплікації даних у динамічному кластерному середовищі з мінімальними вимогами до оточення.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень було розроблено прототип користувацької файлової системи з використанням технології FUSE, призначеної для реплікації баз даних у кластерному середовищі. Рішення базується на подійному механізмі журналювання, підтримує асинхронну реплікацію, забезпечує відмовостійкість та узгодженість даних без потреби в централізованому керуванні або модифікаціях ядра операційної системи. Проведені експерименти підтвердили працездатність обраної архітектури, а також її ефективність у сценаріях з втратою вузлів, затримками зв'язку та паралельним доступом.

У першому розділі проведено аналіз сучасних підходів до побудови файлових систем, зокрема на основі FUSE, та кластерних механізмів зберігання і реплікації даних. Розглянуто переваги реалізації файлових систем у просторі користувача, порівняно традиційні рішення (GlusterFS, CephFS) і визначено основні вимоги до файлової системи, орієнтованої на реплікацію баз даних: цілісність, доступність, узгодженість, масштабованість і толерантність до відмов.

У другому розділі проведено проєктування архітектури файлової системи з урахуванням сформульованих вимог. Описано функціональну логіку компонентів, включаючи кешування, журналювання, модуль реплікації та обробку конфліктів. Сформовано багаторівневу архітектуру, що поєднує взаємодію з ядром через FUSE, користувацьку логіку обробки запитів та мережевий рівень синхронізації між вузлами. Розроблено модель обробки основних файлових операцій – читання, запису та видалення – з гарантіями узгодженості й надійності.

У третьому розділі реалізовано прототип файлової системи на основі запропонованої архітектури, проведено його тестування в кластерному середовищі, проаналізовано продуктивність, поведінку у стресових умовах та обмеження реалізації. Доведено здатність системи до самовідновлення, підтримки реплікації в умовах мережевих збоїв і мінімізації втрат даних. Виявлено недоліки, зокрема

відсутність централізованого адміністрування, чутливість до збоїв часу, обмеження щодо формату журналів, що визначають напрями подальшої оптимізації.

Отримані результати підтверджують, що використання FUSE як технологічної основи для побудови реплікованих файлових систем у кластері є перспективним напрямом. Запропоноване рішення демонструє хорошу адаптивність до нестабільних середовищ, забезпечує прозору інтеграцію з клієнтськими застосунками та відкриває нові можливості для створення легковагих, автономних систем зберігання з підтримкою гнучкої реплікації. Розроблений підхід може бути використаний у наукових проєктах, дослідницьких кластерах або як основа для подальшої розробки індустріального програмного забезпечення з високими вимогами до надійності та масштабованості.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Li H., Fu J., Li Q., Hsu W., Cidon A. Enabling the Write-Back Page Cache with Strong Consistency in Distributed Userspace File Systems. *arXiv preprint*. 2025. arXiv:2503.18191. URL: https://arxiv.org/abs/2503.18191 (дата звернення: 04.03.2025).

2. Lee J. Y., Kim M. H., Raza Shah S. A., Ahn S. U., Yoon H., Noh S. Y. Performance evaluations of distributed file systems for scientific big data in FUSE environment. *Electronics*. 2021. Vol. 10, No. 12. P. 1471. DOI: https://doi.org/10.3390/electronics10121471.

3. Lyu T., Zhang L., Feng Z., Pan Y., Ren Y., Xu M., Payer M., Kashyap S. Monarch: A Fuzzing Framework for Distributed File Systems. *Proc. 2024 USENIX Annual Technical Conference (USENIX ATC '24)*. 2024. P. 529–543.

4. Zhang J., Ren Y., Kannan S. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. *Proc. 20th USENIX Conf. on File and Storage Technologies (FAST '22)*. 2022. P. 297–312.

5. Mane T., Li Q., Hsu W., Cidon A. AVA: Fault-tolerant Reconfigurable Geo-Replication on Heterogeneous Clusters. *arXiv preprint*. 2024. arXiv:2412.01999. URL: https://arxiv.org/abs/2412.01999 (дата звернення: 04.03.2025).

6. Shen H. Selective data replication for online social networks with distributed datacenters. *IEEE Trans. Parallel Distrib. Syst.* 2022. Vol. 33, No. 5. P. 1201–1213.

7. Wu Z., Xu X., Foo C. S., Low B. K. H. Validation Free and Replication Robust Volume-Based Data Valuation. *Adv. Neural Inf. Process. Syst.* 2021. Vol. 34. P. 10837–10848.

8. Maturana F., Rashmi K. V. Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions. *IEEE Trans. Inf. Theory*. 2023. Vol. 69, No. 8. P. 4993–5008.

9. Zhou J., Chen Y., Wang W. Data distribution for heterogeneous storage systems. *IEEE Trans. Comput.* 2022. Vol. 72, No. 6. P. 1747–1762.

10. Kim T., Athlur S., Kadekodi S., Maturana F., Delvira D., Merchant A. Morph: Efficient File-Lifetime Redundancy Management for Cluster File Systems. *Proc. ACM SIGOPS 30th Symp. on Operating Systems Principles (SOSP '24)*. 2024.

11. Shen J., Fu J., Li Q., Hsu W., Cidon A. FUSEE: A Fully Memory-Disaggregated Key-Value Store. *arXiv preprint*. 2023. arXiv:2301.09839. URL: https://arxiv.org/abs/2301.09839 (дата звернення: 04.03.2025).

12. Chen J., Dai F., Gu X., Zhou J., Li B., Wang W. Universal Domain Adaptive Network Embedding for Node Classification. *Proc. 31st ACM Int. Conf. on Multimedia*. 2023. P. 4022–4030.

13. Zhou J., Chen Y., Wang W. A highly reliable metadata service for large-scale distributed file systems. *IEEE Trans. Parallel Distrib. Syst.* 2020. Vol. 31, No. 2. P. 374–392.

14. Wei X., Shen S., Chen R., Chen H. DRTM+B: Replication-driven live reconfiguration for fast and general distributed transaction processing. *IEEE Trans. Parallel Distrib. Syst.* 2022. Vol. 33, No. 10. P. 2628–2643.

15. Raspberry Pi 4 Model B 4GB. *minicomp.com.ua*. URL: https://minicomp.com.ua/ua/raspberry-pi/raspberry-pi-4/raspberry-pi-4-model-b-4gb (дата звернення: 05.03.2025).

16. Coral USB Accelerator. *Coral.ai*. URL: https://coral.ai/products/accelerator/ (дата звернення: 05.03.2025).

17. Fei K., Zhou J., Su L., Wang W., Chen Y., Zhang F. A Graph Convolution Neural Network Based Method for Insider Threat Detection. *Proc. IEEE Int. Conf. on Parallel & Distributed Processing with Applications*. 2022. P. 1–10.

18. Dong M., Chen H. Soft updates made simple and fast on non-volatile memory. *Proc. 2022 USENIX Annual Technical Conference (USENIX ATC '22)*. 2022.

19. Bu H., Dong M., Yi J., Zang B., Chen H. Revisiting persistent indexing structures on Intel Optane DC persistent memory. *J. Comput. Sci. Technol.* 2021. Vol. 36. P. 140–157.

20. Grieves M. Intelligent digital twins and the development and management of complex systems. *Digital Twin*. 2024. Vol. 1, No. 1. P. 8.

21. Xu X., Wu Z., Foo C. S., Low B. K. H. Validation Free and Replication Robust Volume-Based Data Valuation. *Adv. Neural Inf. Process. Syst.* 2021. Vol. 34. P. 10837–10848.

22. Shu Y., Wu Z., Low B. K. H. Unifying and Boosting Gradient-Based Training-Free Neural Architecture Search. *Adv. Neural Inf. Process. Syst.* 2022. Vol. 35. P. 33001–33015.

23. Xu X., Wu Z., Verma A., Foo C. S., Low B. K. H. FAIR: Fair Collaborative Active Learning with Individual Rationality for Scientific Discovery. *Proc. Int. Conf. on Artificial Intelligence and Statistics*. 2023. P. 4033–4057.

24. Hu W., Shu Y., Yu Z., Wu Z., Lin X., Dai Z., Ng S. K., Low B. K. H. Localized Zeroth-Order Prompt Optimization. *Adv. Neural Inf. Process. Syst.* 2025. Vol. 37. P. 86309–86345.

25. Nguyen Q. P., Wu Z., Low B. K. H., Jaillet P. Trusted-Maximizers Entropy Search for Efficient Bayesian Optimization. *Proc. Conf. on Uncertainty in Artificial Intelligence*. 2021. P. 1486–1495.

26. Wu Z., Lin X., Dai Z., Hu W., Shu Y., Ng S. K., Jaillet P. Prompt Optimization with Ease? Efficient Ordering-Aware Automated Selection of Exemplars. *Adv. Neural Inf. Process. Syst.* 2025. Vol. 37. P. 122706–122740.

27. Li J., Wang Q., Jayasinghe D., Park J., Zhu T., Pu C. Performance Overhead among Three Hypervisors: An Experimental Study Using Hadoop Benchmarks. *Proc. 2013 IEEE Int. Congress on Big Data*. Santa Clara, CA, USA : IEEE, 2013. P. 9–16.

28. Wu Z., Amiri M. M., Raskar R., Low B. K. H. Incentive-Aware Federated Learning with Training-Time Model Rewards. *Proc. 12th Int. Conf. on Learning Representations*. 2024.

29. Lin X., Xu X., Wu Z., Ng S. K., Low B. K. H. Distributionally Robust Data Valuation. *Proc. 41st Int. Conf. on Machine Learning*. 2024.

30. Xu X., Wu Z., Qiao R., Verma A., Shu Y., Wang J., Niu X., He Z., Chen J., Zhou Z. Position Paper: Data-Centric AI in the Age of Large Language Models. *arXiv preprint*. 2025. arXiv:2504.12345. URL: https://arxiv.org/abs/2504.12345 (дата звернення: 12.04.2025).

31. Grieves M. Digital Twins: Past, Present, and Future. *The Digital Twin*. 2023. P. 97–121.

32. Mascetti L. та ін. CERN Disk Storage Services: Report from last data taking, evolution and future outlook towards Exabyte-scale storage. *EPJ Web Conf.* 2020. Vol. 245. P. 04038.

33. Bohossian V., Fan C. C., LeMahieu P. S., Riedel M. D., Xu L., Bruck J. Computing in the RAIN: A reliable array of independent nodes. *IEEE Trans. Parallel Distrib. Syst.* 2001. Vol. 12. P. 99–114.

34. Introduction – EOS CITRINE Documentation. *CERN*. URL: https://eos-docs.web.cern.ch/intro.html (дата звернення: 18.04.2025).

35. RAIN – EOS CITRINE Documentation. *CERN*. URL: https://eos-docs.web.cern.ch/using/rain.html (дата звернення: 25.04.2025).

36. Introduction – Gluster Docs. *Red Hat*. URL: https://docs.gluster.org/en/latest/Administrator-Guide/GlusterFS-Introduction/ (дата звернення: 07.05.2025).

37. Chapter 9. Benchmarking Performance Red Hat Ceph Storage 1.3. *Red Hat*. URL: https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/1.3/html/administration_guide/benchmarking_performance (дата звернення: 12.05.2025).

38. Axboe J. GitHub – axboe/fio: Flexible I/O Tester. URL: https://github.com/axboe/fio (дата звернення: 16.05.2025).

39. SHA-256 pseudocode. *stackoverflow.com*. URL: https://stackoverflow.com/questions/11937192/sha-256-pseuedocode (дата звернення: 16.05.2025).

40. Lustre Roadmap. *OpenSFS*. URL: https://www.lustre.org/roadmap/ (дата звернення: 16.05.2025).

41. Bhat W. A. FUSE based file system for efficient storage and retrieval of fragmented multimedia files. *J. King Saud Univ. Comput. Inf. Sci.* 2022. Vol. 34, No. 10. P. 8380–8389.

42. Lee S., Jho N. S., Chung D., Kang Y., Kim M. Rcryptect: Real-time detection of cryptographic function in the user-space filesystem. *Comput. Secur.* 2022. Vol. 112. Article 102512.

43. Miller S., Zhang K., Chen M., Jennings R., Chen A., Zhuo D., Anderson T. High velocity kernel file systems with bento. *Proc. 19th USENIX Conf. on File and Storage Technologies (FAST '21)*. 2021. P. 65–79.

44. da Silva E. C., Sato L. M., Midorikawa E. T. Distributed File System to Leverage Data Locality for Large-File Processing. *Electronics*. 2023. Vol. 13, No. 1. P. 106.

45. Kim J., Yu H. J., Kang H., Shin J. H., Jeong H., Noh S. Y. Performance Analysis of Distributed File System Based on RAID Storage for Tapeless Storage. *IEEE Access*. 2023. Vol. 11. P. 116153–116168.

46. Rydning D. R. J. G. J. The Digitization of the World from Edge to Core. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (дата звернення: 18.05.2025).

47. Storage | CERN. *CERN*. URL: https://home.cern/science/computing/storage (дата звернення: 20.05.2025).

48. Mascetti L. та ін. CERN Disk Storage Services: Report from last data taking, evolution and future outlook towards Exabyte-scale storage. *EPJ Web Conf.* 2020. Vol. 245. P. 04038. DOI: https://doi.org/10.1051/epjconf/202024504038.

49. About the Lustre® File System | Lustre. *OpenSFS*. URL: https://www.lustre.org/about/ (дата звернення: 21.05.2025).

50. Bohossian V., Fan C. C., LeMahieu P. S., Riedel M. D., Xu L., Bruck J. Computing in the RAIN: A reliable array of independent nodes. *IEEE Trans. Parallel Distrib. Syst.* 2001. Vol. 12. P. 99–114. DOI: https://doi.org/10.1109/71.903163.

51. Szeredi M. Libfuse: Libfuse API Documentation. URL: http://libfuse.github.io/doxygen/ (дата звернення: 22.05.2025).

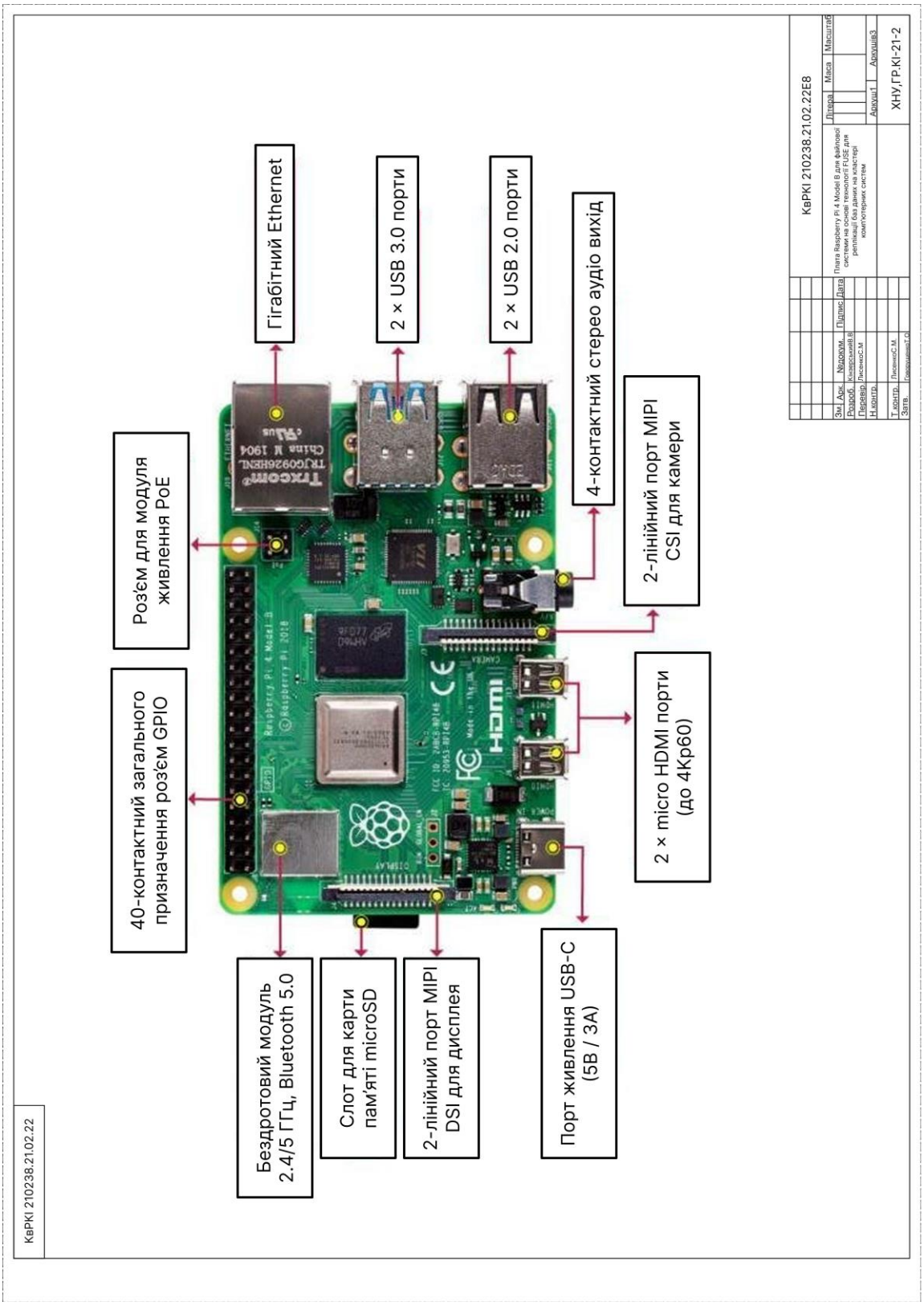
52. Weil S. A., Brandt S. A., Miller E. L., Maltzahn C. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. *Proc. 2006 ACM/IEEE Conf. on Supercomputing (SC '06)*. New York : ACM, 2006. P. 122-es.

53. Lembke J., Roman P. L., Eugster P. DEFUSE: An interface for fast and correct user space file system access. *ACM Trans. Storage*. 2022. Vol. 18, No. 3. P. 1–29.

Додаток А

(обов’язковий)

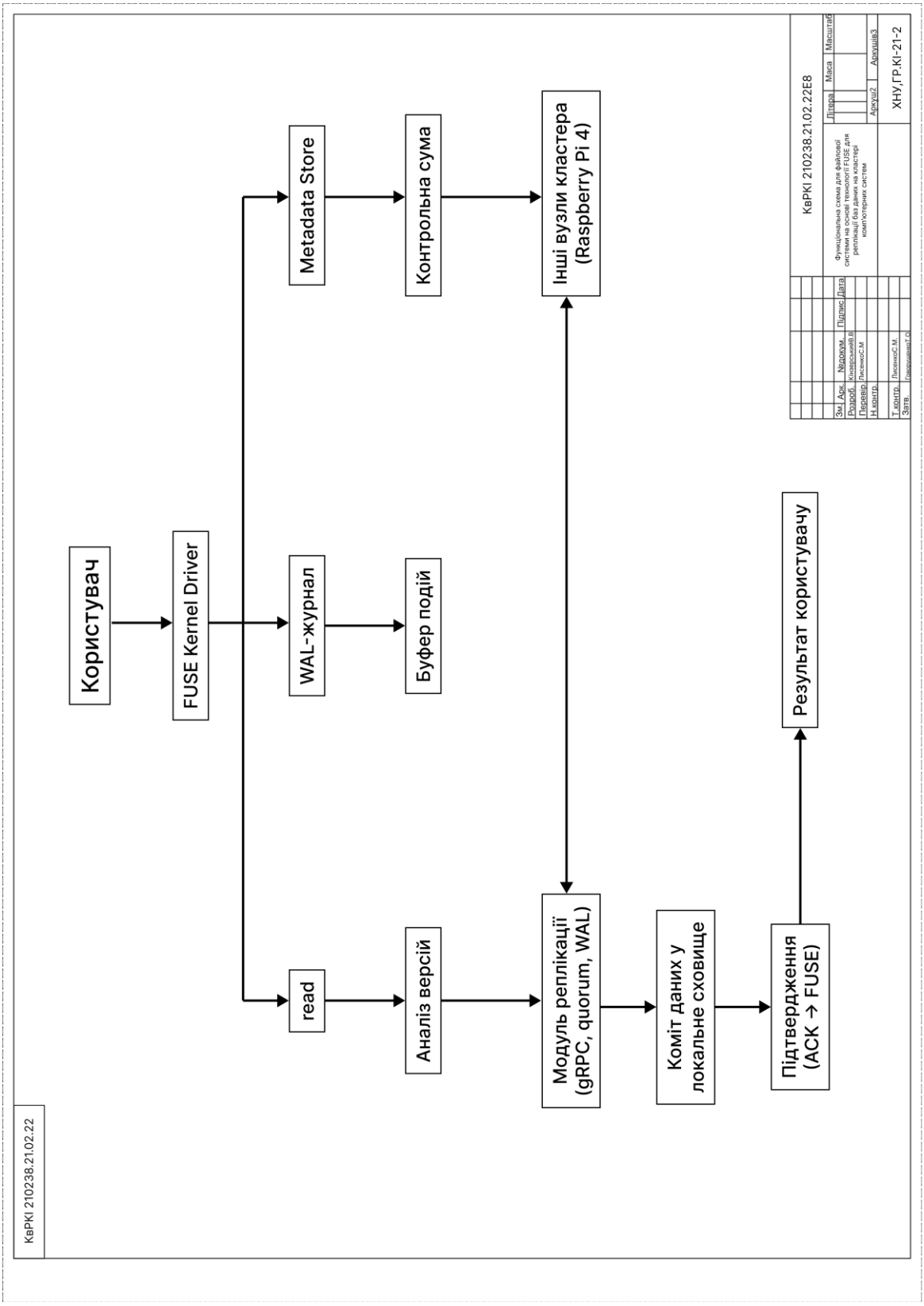
КОПІЯ КРЕСЛЕННЯ «ПЛАТА RASPBERRY PI 4 MODEL B»



Додаток Б

(обов’язковий)

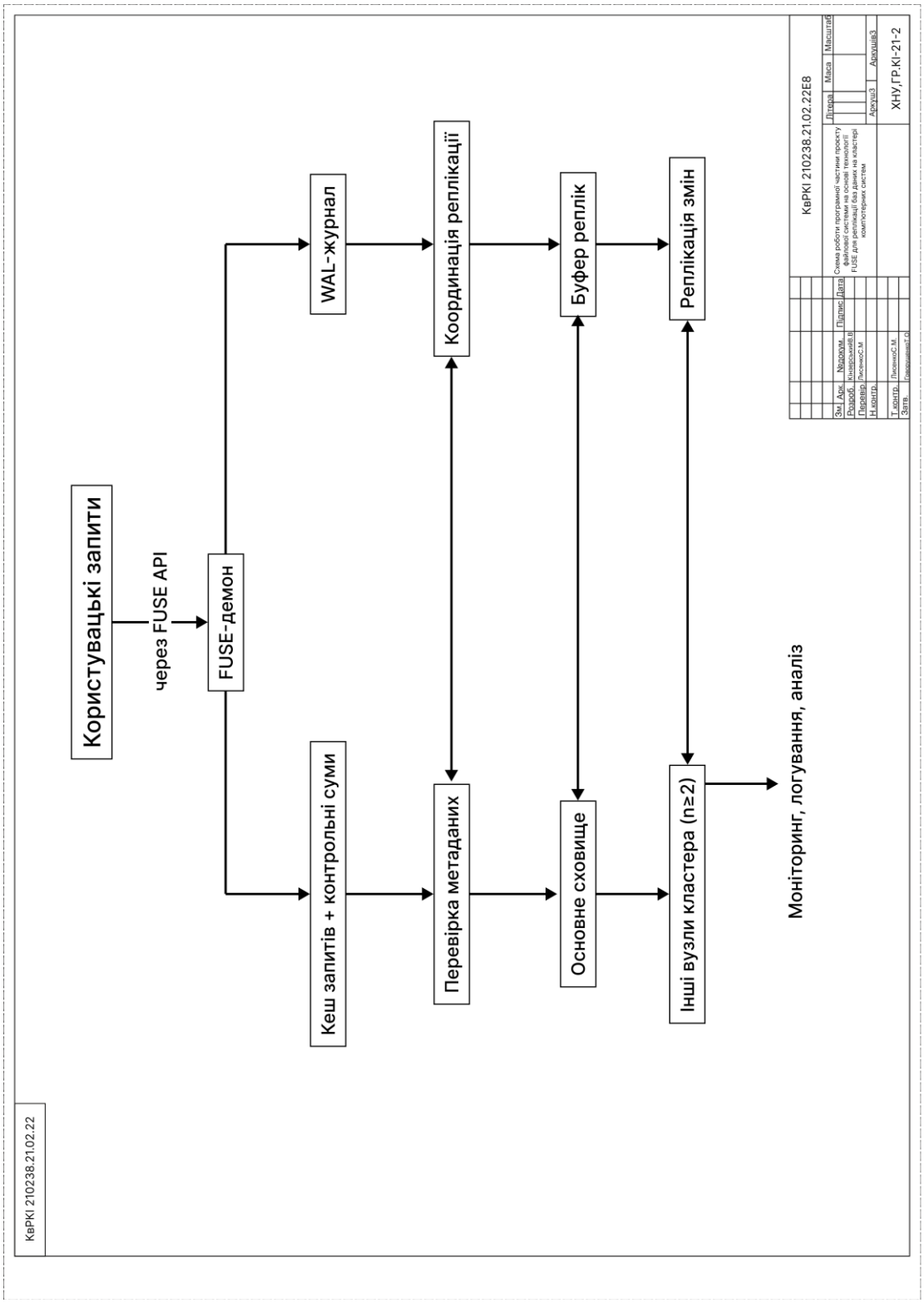
КОПІЯ КРЕСЛЕННЯ «ФУНКЦІОНАЛЬНА СХЕМА»



Додаток В

(обов’язковий)

КОПІЯ КРЕСЛЕННЯ «СХЕМА РОБОТИ ПРОГРАМНОЇ ЧАСТИНИ ПРОЄКТУ»



Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Владислав КІНЗЕРСЬКИЙ

Співавтор:

Назва: Кінзерський_Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем

Експерт:

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:3.5%

Коефіцієнт подібності 2:0.8%

Мікропробіли: 8

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-06-10 23:25:39.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2025-06-11 Доцент Андрій Нічепорук

Дата експерт

Anti-Plagiarism (UA) v-15.281 Educational

The maximum coincidence with one document 18.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 10%**

	Document		Sum coincidence on the DB	
	Symbols	Lexemes	Symbols	Lexemes
ID: 244753 Title: БКР Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем Added in a DB: 2025-06-10 Authors: Владислав КІНЗЕРСЬКИЙ Heads: Сергій ЛИСЕНКО Consultants: Opponents:	127045	845	23604 (19%)	163 (19%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes
240776	Title: Звіт з ПДП Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем Added in a DB: 2025-05-02 Authors: Кінзерський В.В. Heads: Лисенко С.М. Consultants: Opponents:	22726 (18.0%)	161 (19.0%)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Кінзерський Владислав Вікторович

Тема: Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3____ Кількість сторінок записки ___81___

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є розробка та дослідження користувацької файлової системи, створеної на основі технології FUSE, для забезпечення реплікації баз даних у кластері комп'ютерних систем. У роботі було синтезовано архітектуру файлової системи, проєктування якої враховує вимоги до узгодженості, масштабованості та стійкості до відмов. Розроблено демонстраційний прототип, реалізований на базі мікрокомп'ютера Raspberry Pi, що підтримує прозору реплікацію змін між вузлами кластеру з використанням журналювання та буферизації. Проведено моделювання кластерного середовища, тестування системи на предмет відмовостійкості та затримок синхронізації, а також аналіз результатів.

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведено дослідження предметної області, розглянуто принципи роботи FUSE, кластерних файлових систем та механізмів реплікації. Також здійснено порівняльний аналіз сучасних кластерних рішень, описано переваги та обмеження FUSE для реалізації користувацьких файлових систем з підтримкою реплікації. У другому розділі розроблено архітектуру файлової системи, визначено функціональні та нефункціональні вимоги до неї. Спроєктовано логіку обробки операцій читання, запису, видалення, а також описано мережевий рівень

синхронізації. Створено логічну структуру кластерного вузла на базі Raspberry Pi, визначено програмне середовище реалізації. У третьому розділі реалізовано демонстраційний прототип файлової системи, змодельовано кластер з кількох вузлів для тестування. Проведено експериментальне дослідження, включаючи перевірку на відмовостійкість, вимірювання затримок при реплікації, аналіз впливу втрати зв'язку між вузлами. Використано сучасні інструменти моніторингу та логування, що забезпечило прозорість процесу реплікації. Робота базується на сучасних підходах до розподілених обчислень, активно застосовує відкриті інструменти та гнучкі інтерфейси програмування, зокрема бібліотеку libfuse, gRPC, Prometheus, що відповідає передовим методам проєктування інформаційних систем.

4. Позитивні сторони роботи: робота має високу практичну цінність завдяки створенню працездатного прототипу файлової системи з підтримкою реплікації у кластерному середовищі, що може бути використаний як основа для подальших прикладних і наукових розробок у сфері розподілених систем зберігання даних.

5. Негативні сторони роботи: У роботі недостатньо уваги приділено формальному моделюванню алгоритмів узгодження та реплікації, а також оцінці продуктивності системи при масштабуванні кількості вузлів у кластері.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: добре

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) Бедратюк Леонід Петрович, доктор фізико-математичних наук, кафедра інженерії програмного забезпечення

"12" червня 2025 р. _____ (підпис)

Завідувачу кафедри КІІС
д-р. філософії, доц. Ользі ПАВЛОВІЙ

Владислава КІНЗЕРСЬКОГО
ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-21-2

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Strike-Plagiarism та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

11.06 2025 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Файлова система на основі технології FUSE для реплікації баз даних на кластері комп'ютерних систем

Автор: Владислав КІНЗЕРСЬКИЙ

Спеціальність: 123– Комп'ютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Сергій ЛИСЕНКО, д.т.н, професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) виявлені запозичення розміщені у розділах, присвячених аналізу існуючих рішень та прототипів, і не стосуються авторських досліджень чи отриманих результатів;
2) усі запозичення є фрагментарними та супроводжуються належним чином оформленими бібліографічними посиланнями;
3) окремі збіги, зафіксовані системою, стосуються загальновживаних технічних або наукових фраз, що підтверджується наявністю десятків джерел з ідентичними фрагментами;
4) усі ознаки модифікації тексту, зареєстровані системою, пов'язані з використанням латинських символів у поєднанні з українськими скороченнями в індексах формул і не є свідомим редагуванням або перефразуванням авторських текстів.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 3.53% і адресується до 33 першоджерел; та системою Anti-Plagiarism складає 18%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ Сергій ЛИСЕНКО

Гарант ОП _____ Андрій Нічепорук

Завідувач кафедри КІІС _____ Ольга ПАВЛОВА