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

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ І.А. Дичка
«___» _____ 2019 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 6.050103 «Програмна інженерія»

на тему: «Розподілена файлова система.
Підсистема користувацького інтерфейсу»

Виконав:
студент IV курсу, групи КП-51,
Лобов Віталій Михайлович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент,
Вунтесмері Ю.В. _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н.,
Онай М.В. _____

Рецензент:
Доцент кафедри ММСА, к.т.н., доцент,
Дідковська М.В. _____

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

Київ – 2019 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки – 6.050103 «Програмна інженерія»

«ЗАТВЕРДЖУЮ»
Науковий керівник кафедри
_____ І.А. Дичка
«___» _____ 2018 р.

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЕКТ СТУДЕНТУ

Лобову Віталію Михайловичу

1. Тема проекту РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ, керівник проекту Вунтесмері Юрій Володимирович, к.т.н., доцент, затверджені наказом по університету від «22» травня 2019 року № 1331-С.

2. Термін подання студентом проекту: «11» червня 2019 р.

3. Вихідні дані для дипломного проектування: див. Технічне завдання.

4. Перелік задач, які потрібно вирішити:
- провести аналіз існуючих аналогів та аналіз предметної області;
- обрати модуль для написання файлових систем;
- сформувати вимоги до користувацького інтерфейсу;
- виконати програмну реалізацію клієнтського модулю відповідно до вимог технічного завдання;
- провести тестування роботи програми.

5. Перелік обов'язкового ілюстративного матеріалу:
- алгоритм роботи бібліотеки FUSE (креслення);
- протокол передачі даних (креслення);
- схема роботи FUSE (плакат);
- структура клієнтського модулю (плакат).

6. Консультанти:

Питання	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент		

7. Дата видачі завдання: «31» жовтня 2018 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів	Примітка
1.	Вивчення літератури за тематикою проекту	08.12.2018	
2.	Розробка та узгодження технічного завдання	13.12.2018	
3.	Підготовка матеріалів першого розділу дипломного проекту	25.12.2018	
4.	Розроблення структури файлової системи	22.01.2019	
5.	Підготовка матеріалів другого розділу дипломного проекту	01.03.2019	
6.	Програмна реалізація користувацького інтерфейсу	10.03.2019	
7.	Тестування клієнтського модулю	20.04.2019	
8.	Підготовка матеріалів третього розділу дипломного проекту	06.05.2019	
9.	Підготовка матеріалів четвертого розділу дипломного проекту	13.05.2019	
10.	Підготовка графічної частини дипломного проекту	23.05.2019	
11.	Оформлення документації дипломного проекту	27.05.2019	

Студент _____ Лобов В.М.
(підпис)

Керівник проекту _____ Вунтесмері Ю.В.
(підпис)

АНОТАЦІЯ

Даний дипломний проект присвячений розробленню розподіленої файлової системи.

У роботі описана актуальність та проблематика даної системи, виконаний порівняльний аналіз існуючих програмних рішень, обґрунтовано вибір технологій, а також описана архітектура та особливості реалізації даної розподіленої файлової системи, зокрема підсистеми користувацького інтерфейсу.

Розроблена розподілена файлова система надає користувачам можливість отримати доступ до файлів з декількох хостів через комп'ютерну мережу. Особливістю даної системи є те, що дані з файлу розбиваються та зберігаються на різних серверах. Таким чином, дана файлова система гарантує, що тільки власник може отримати доступ до всього файлу, а окремі частини, які зберігаються на серверах, не представляють ніякої цінності без усіх даних.

Дана файлова система представляє собою три окремі модулі, що можуть знаходитись на різних комп'ютерах та взаємодіють один з одним. Саме тому особливу увагу було приділено швидкості передачі даних та коректності роботи усієї системи.

У даному дипломному проекті розроблено клієнтський модуль для взаємодії з серверною частиною файлової системи та протокол передачі даних.

ABSTRACT

This diploma project is devoted to the development of a distributed file system.

The paper describes the relevance and the problems of the given system, compares the existing software solutions, rationales the choice of technologies, and also describes the architecture and features of the implementation of this distributed file system, in particular the subsystem of the user interface.

The developed distributed file system provides users with the ability to access files from multiple hosts over a computer network. The feature of this system is that the data from the file is divided into parts and stored on different servers. Thus, this file system ensures that only the owner can access the entire file, and individual parts stored on the servers do not represent any value without all data.

This file system represents three separate modules that can be located on different computers and interact with each other. That is why special attention was paid to the speed of data transmission and the correctness of the whole system.

At this project the client module for interacting with the server part of the file system and the data transfer protocol was developed.

ДП.045200-01-90 Розподілена файлова система. Підсистема користувацького інтерфейсу. Відомість проекту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проекту		
ДП.045200-02-91	Розподілена файлова	4	
	система. Підсистема		
	користувацького		
	інтерфейсу. Технічне		
	завдання		
ДП.045200-03-81	Розподілена файлова	57	
	система. Підсистема		
	користувацького		
	інтерфейсу.		
	Пояснювальна записка		
ДП.045200-04-51	Розподілена файлова	4	
	система. Підсистема		
	користувацького		
	інтерфейсу. Програма та		
	методика тестування		
ДП.045200-05-34	Розподілена файлова	4	
	система. Підсистема		
	користувацького		
	інтерфейсу. Керівництво		
	користувача		
ДП.045200-06-99	Розподілена файлова	1	
	система. Підсистема		
	користувацького		
	інтерфейсу. Алгоритм		
	роботи бібліотеки FUSE.		
	Схема алгоритму		

Позначення	Найменування	Кіл-ть	Примітка
ДП.045200-07-99	Розподілена файлова	1	
	система. Підсистема		
	користувацького		
	інтерфейсу. Протокол		
	передачі даних. Діаграма		
	діяльності		
ДП.045200-08-98	Розподілена файлова	1	
	система. Підсистема		
	користувацького		
	інтерфейсу. Компакт-диск		

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ І.А. Дичка

«___» _____ 2018 р.

РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

Технічне завдання

ДП.045200-02-91

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.М. Лобов

2018

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Розподілена файлова система. Підсистема користувацького інтерфейсу.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є завдання на дипломне проектування, затверджене кафедрою програмного забезпечення комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім. Ігоря Сікорського).

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для використання в якості файлової системи з метою надання клієнту можливості зберігати файли великого розміру та зчитувати їх, не турбуючись про розміри файлу, швидкість обробки та можливість стороннього доступу.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Клієнтська частина повинна бути виконана на мові програмування C, з використанням інтерфейсу для створення файлових систем – FUSE, та забезпечувати наступні функціональні можливості:

1) перегляд вмісту директорій;
2) створення файлу;
3) відкриття файлу;
4) запис даних в файл;
5) зчитування даних з файлу;
6) закриття файлу.

5. ВИМОГИ ДО ПРОЕКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання проекту повинна бути розроблена наступна документація:

1) пояснювальна записка;

2) програма та методика тестування;

3) керівництво користувача;

4) креслення:

- «Алгоритм роботи бібліотеки FUSE. Схема алгоритму»;
- «Протокол передачі даних. Діаграма діяльності».

6. ЕТАПИ ПРОЕКТУВАННЯ

Вивчення літератури за тематикою роботи 08.12.2018

Розроблення та узгодження технічного завдання 13.12.2018

Розроблення структури файлової системи 22.01.2019

Програмна реалізація користувацького інтерфейсу 10.03.2019

Тестування клієнтського модулю 20.04.2019

Підготовка текстової частини дипломного проекту 01.05.2019

Підготовка графічної частини дипломного проекту 23.05.2019

Оформлення документації дипломного проекту 27.05.2019

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до «Програми та методики тестування».

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ І.А. Дичка

«___» _____ 2019 р.

РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

Пояснювальна записка

ДП.045200-03-81

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.М. Лобов

2019

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

ФС – файлова система.

ОС – операційна ситсема.

OSD – пристрої зберігання об’єктів у файловій системі Ceph.

MDS – сервер для зберігання метаданих у файловій системі Ceph.

POSIX – набір стандартів, які описують інтерфейси між операційною системою та застосунками.

ACL – список прав доступу до об’єкта, який визначає, хто або що може отримувати доступ до нього.

Unix – комп’ютерна операційна система.

FUSE – Filesystem in Userspace, модуль для Unix-подібної операційної системи, який дозволяє користувачам без спеціальних прав та без модифікації ядра створювати власні файлові системи.

NFS – протокол мережевого доступу до файлових систем.

Linux – загальна назва UNIX-подібних операційних систем на основі однойменного ядра.

Ext3 – журнальована файлова система, що використовується в операційних системах на ядрі Linux.

Ext4 – журнальована файлова система, яка використовується в операційних системах з ядром Linux, заснована на файловій системі ext3.

XFS – високопродуктивна журнальована файлова система для власної операційної системи IRIX.

Btrfs – нова файлова система для Linux, створена з метою реалізації додаткових функцій, які б покращили відмовостійкість, спростили адміністрування і ремонтні роботи.

Infiniband – високошвидкісна комутована послідовна шина, що застосовується як для внутрішніх (внутрішньо-системних), так і для міжсистемних з’єднань.

RDMA – апаратне рішення для забезпечення прямого доступу до оперативної пам’яті іншого комп’ютера.

TCP – протокол керування передачею, призначений для управління передачею даних у комп'ютерних мережах, працює на транспортному рівні моделі OSI.

UDP – протокол датаграм користувача, один з найпростіших протоколів транспортного рівня моделі OSI, котрий виконує обмін повідомленнями без підтвердження та гарантії доставки.

IP – протокол мережевого рівня для передавання датаграм між мережами.

Lustre – розподілена файлова система масового паралелізму, використовувана зазвичай для великомасштабних кластерних обчислень.

ARM – 32-бітна архітектура процесорів зі скороченим набором команд.

SMB/CIFS – протокол прикладного рівня (в моделі OSI), зазвичай використовується для надання розділеного доступу до файлів, принтерів, послідовних портів передачі даних, та іншої взаємодії між вузлами в комп’ютерній мережі.

QEMU – вільна програма з відкритим кодом для емуляції апаратного забезпечення різних платформ.

Samba – вільна реалізація мережевого протоколу SMB/CIFS.

WebDAV – це набір розширень та доповнень до протоколу HTTP (Hypertext Transfer Protocol), які дозволяють користувачам спільно редагувати та керувати файлами на веб-серверах.

OpenStack – це комплекс проектів вільного програмного забезпечення для створення обчислювальних хмар і хмарних сховищ, як публічних, так і приватних.

Master-Slave – один зі способів реплікації баз даних.

Apache Hadoop – вільна програмна платформа і каркас для організації розподіленого зберігання і обробки наборів великих даних.

MapReduce – це програмна модель та програмний каркас, що її реалізує, розроблені компанією Google для проведення розподіленої

паралельної обробки великих масивів даних з використанням кластерів звичайних недорогих комп'ютерів.

Java – об'єктно-орієнтована мова програмування.

API – Application Programming Interface (прикладний програмний інтерфейс);

C99 – стандарт мови програмування C.

FreeBSD – Unix-подібна операційна система.

OpenBSD – Unix-подібна операційна система.

NetBSD – вільна, захищена, Unix-подібна операційна система.

OpenSolaris – Unix-подібна операційна система.

Minix 3 – проект, метою якого є створення маленької, простої та високонадійної операційної системи.

Android – операційна система і платформа для мобільних телефонів та планшетних комп'ютерів, створена компанією Google на базі ядра Linux.

MacOS – перша комерційна графічна операційна система, створена компанією Apple Computer.

VFS – віртуальна файлова система.

Сокет – назва програмного інтерфейсу для забезпечення обміну даними між процесами.

БД – база даних.

SQL – декларативна мова програмування для взаємодії користувача з базами даних, що застосовується для формування запитів, оновлення і керування реляційними БД, створення схеми бази даних та її модифікації, системи контролю за доступом до бази даних.

Порт – кінцева точка зв'язку між комп'ютерами.

Кластер – це декілька незалежних обчислювальних машин, що використовуються спільно і працюють як одна система для вирішення тих чи інших задач.

Вузол – складова одиниця кластера (сервер).

Middleware – проміжний сервіс керування мережевими з'єднаннями.

ID – унікальний ідентифікатор.

WEB – інтернет простір.

Багатопоточність – властивість операційної системи або застосунку, яка полягає в тому, що процес, породжений в операційній системі, може складатися з кількох потоків, що виконуються паралельно, або навіть одночасно на багатопроцесорних системах.

Асинхронність – це процес обробки, що дозволяє продовжити обробку інших завдань, не чекаючи завершення попереднього завдання.

М’ютекс – є одним із засобів синхронізації роботи потоків або процесів.

TTR – чисельний показник, який визначає кількість повторів операції.

ВСТУП

У наш час, коли інформація займає важливе місце в житті кожної людини, виникає проблема зберігання цієї інформації. Коли дані накопичуються, їх приходиться видаляти або збільшувати розмір накопичуючого пристрою, що супроводжується деякими грошовими витратами.

Особливо проблема зі зберіганням великої кількості інформації неймовірно поширена в компаніях або державних підприємствах. Кількість інформації, яку вони зберігають, щоденно збільшується, а засоби для зберігання модернізуються значно повільніше. Більшість таких компаній зберігають свої дані на персональних комп'ютерах. Деякі з них використовують віддалені сервера, де і зберігають потрібну інформацію. Але це вимагає великих грошових витрат та певної кваліфікації персоналу, так як звичайний співробітник не зможе підключитися до сервера та зчитати потрібну йому інформацію. Для цього необхідно виділити час та ресурси для його навчання, що є ще однією проблемою при роботі з великою кількістю інформації.

Найпростішим способом, яким можна вирішити ці проблеми, є збільшення кількості доступної пам'яті на локальній машині. Але з часом, коли інформації стає ще більше, необхідно підтримувати стабільну роботу цих пристроїв для зберігання даних. Саме тому треба шукати інші зручні варіанти, які зможуть вирішити усі вищезазначені проблеми.

Проаналізувавши сучасні технології і потреби, які виникають у IT-індустрії, можна зробити висновок, що відповідним рішенням для вирішення проблем, які виникають при зберігання інформації великого обсягу, є перенесення даних з фізичних носіїв в мережеву хмару – скупчення великої кількості невеликих пристроїв. Але взаємодія з ними вимагає певних знань для використання звичайним користувачем. Саме тому використовують розподілені файлові системи, адже вони максимально спрощують роботу з даними. Інтерфейс взаємодії таких систем є дуже

простий та звичний кожному користувачу комп’ютера, адже перегляд інформації здійснюється через термінал або звичайний провідник (файловий менеджер).

Підсумовуючи вище сказане, можна зробити висновок, що тема, якій присвячений даний дипломний проект, а саме – розробка розподіленої файлової системи, є актуальною та затребуваною в наш час.

1. АНАЛІЗ ПРОБЛЕМИ І ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Опис проблеми

Робота з файлами може супроводжуватися деякими проблемами, особливо якщо ці файли мають великий розмір. В такому випадку ми обов’язково стикаємося з наступними проблемами:

1. Збереження великих файлів на локальному носії. Для цього необхідна значна кількість місця на комп’ютері. І в результаті, чим більший файл, тим більше місця він займає, що погано впливає на кількість вільного місця та на роботу комп’ютера.
2. Ресурсозатратність при переміщенні файлів, особливо через мережу інтернет. В результаті чого, на це витрачається багато часу, а також ресурсів комп’ютера, що впливає на обробку інших його задач в даний момент часу.
3. Масштабування сховища даних. З часом кількість інформації стає настільки багато, що одного пристрою для її зберігання не вистачає, особливо це стосується великих фірм або серверів. В такому випадку необхідно розширювати кількість вільного місця, шляхом збільшення кількості накопичуючих пристроїв, що призводить до зростання накладних витрат.

Для вирішення вищезазначених проблем використовують розподілені файлові системи [1].

Розподілена файлова система – це будь-яка файлова система, що надає доступ до даних, які зберігаються на декількох серверах через комп’ютерну мережу. Файли в таких системах, доступні по мережі, для програм і користувачів так само, як і файли на локальному диску [2].

В таких системах не потрібно турбуватися про кількість вільного місця на комп’ютері, так як інформація зберігається на віддалених пристроях, що легко масштабуються. А також такі файлові системи оптимізовані для передачі великого обсягу даних, що пришвидшує роботу з великими файлами [3].

1.2. Огляд існуючих рішень

1.2.1. CephFS

Ceph – це відкрита програмна розробка еластичного та легко масштабованого петабайтного сховища даних. В основі якого лежить об’єднання дискових просторів декількох десятків серверів в об’єктне сховище, що дозволяє реалізувати гнучку багаторазову псевдовипадкову надмірність даних [4].

CephFS – це файлова система, сумісна з стандартами POSIX, яка використовує кластер Ceph Storage для зберігання своїх даних [5]. Для запуску цієї файлової системи необхідно мати запущений Ceph Storage з принаймні одним сервером метаданих Ceph (MDS).

Файлова система Ceph пропонує такі функції та вдосконалення [6]:

1. Масштабованість. Файлова система Ceph має високу масштабованість, оскільки клієнти безпосередньо читають і записують на усі вузли OSD.
2. Спільна файлова система. CephFS – це спільна файлова система, завдяки чому кілька клієнтів можуть працювати на одній файловій системі одночасно [7].
3. Висока доступність. Файлова система Ceph забезпечує кластер серверів метаданих Ceph (MDS). Один з яких є активний, а інші перебувають у режимі очікування. Якщо активний MDS несподівано завершується, один в режимі очікування MDS стає активним. Як результат, монтування клієнта продовжує працювати навіть при збою сервера. Така поведінка робить файлову систему Ceph високо доступною.
4. Макети файлів і каталогів. Файлова система Ceph дозволяє користувачам конфігурувати макети файлів і каталогів для використання декількох пулів.
5. Списки контролю доступу POSIX (ACL). Файлова система Ceph підтримує списки контролю доступу POSIX (ACL). ACL увімкнені

за замовчуванням за допомогою файлових систем Ceph, встановлених як клієнти ядра.

6. Квоти клієнта. Клієнт FUSE файлової системи Ceph підтримує встановлення квот на будь-який каталог системи. Квота може обмежувати кількість байтів або кількість файлів, що зберігаються під цією точкою в ієрархії каталогів.

До обмежень файлової системи відносяться:

1. Підтримка списків керування доступом (ACL) у клієнтах FUSE. Щоб використовувати функцію ACL з файловою системою Ceph, встановленою як клієнт FUSE, необхідно попередньо ввімкнути її та перезапустити Ceph сервіс.
2. Знімки. Створення знімків не включено за промовчанням, оскільки ця функція все ще є експериментальною і може призвести до несподіваного завершення MDS або вузлів клієнта.
3. Кілька активних MDS. За замовчуванням підтримуються лише конфігурації з одним активним MDS. Наявність більш активної MDS може призвести до помилки файлової системи Ceph.
4. Кілька файлових систем Ceph. Створення декількох файлових систем Ceph в одному кластері ще не повністю підтримується і може призвести до несподіваного завершення MDS або вузлів клієнта.

Файлова система Ceph прагне дотримуватися семантики POSIX, де це можливо. Наприклад, на відміну від багатьох інших поширених мережевих файлових систем, таких як NFS [8], CephFS підтримує сильну когерентність кешу для всіх клієнтів [9]. Мета полягає в тому, щоб процеси, що використовують файлову систему, на різних хостах, поводилися так само, як і при знаходженні на одному і тому ж хості.

1.2.2. GlusterFS

GlusterFS – це розподілена, паралельна, лінійно масштабована файлова система з можливістю захисту від збоїв. За допомогою неї можна

об’єднати безліч сховищ даних, розміщених на різних серверах (горизонтальне масштабування) в одну мережеву файлову систему. Так само можливе об’єднання кількох сховищ одного сервера (вертикальне масштабування). А захист від збоїв досягається за допомогою різних політик дублювання даних [10].

GlusterFS використовує механізми FUSE і працює по верх будь-яких POSIX файлових систем, наприклад Ext3, Ext4, XFS, Btrfs. В якості транспорту може використовуватися Infiniband RDMA і TCP/IP.

На відміну від інших розподілених файлових систем, таких як Lustre і Ceph, для роботи GlusterFS не потрібно окремий сервер для зберігання метаданих. В даному випадку вони зберігаються разом з даними в розширених атрибутах файлів. Завдяки відсутності прив’язки до централізованого сервера мета-даних файлова система забезпечує практично необмежену масштабованість. Обсяг сховища може вимірюватися петабайт [11].

GlusterFS надає наступні можливості для використання:

1. Можна використовувати будь-яке обладнання (підійде навіть ARM).
2. Працює через стандартні протоколи NFS, SMB/CIFS або рідний клієнт;
3. Автоматичне виявлення відмови окремого сховища (Brick Failure Detection).
4. Можливість стиснення даних при передачі по мережі.
5. Підтримується шифрування даних дискових розділів на стороні сервера з використанням ключів, доступних тільки клієнту. При цьому шифрується тільки вміст файлів, імена та метадані залишаються незашифровані. Шифрування застосовується при використанні NFS.
6. Оптимізована для використання в якості розподіленого сховища образів віртуальних машин.

7. Інтеграція з QEMU і Samba дозволяє організувати прямий доступ до даних, що зберігаються в GlusterFS, без монтування розділу.
8. Може використовуватися в якості первинного сховища в OpenStack.
9. Механізм zerofill дозволяє заповнювати нулями нові образи віртуальних машин.
10. Підтримується асинхронна гео-реплікація по моделі master-slave.

GlusterFS агрегує різні сервери зберігання по мережевому з'єднанню в одну велику паралельну мережеву файлову систему. Заснований на конструкційному просторі користувача, він забезпечує виняткову продуктивність для різноманітних робочих навантажень і є основним будівельним блоком Red Hat Gluster Storage.

До серверів GlusterFS, сумісних з POSIX, які використовують формат файлової системи XFS для зберігання даних на дисках, можна отримати доступ за допомогою стандартних протоколів доступу, включаючи мережеву файлову систему (NFS) і серверний блок повідомлень (SMB) (також відомий як CIFS).

Використання сховища об'єктів Red Hat Gluster накладає свої обмеження [12]:

1. Ім'я об'єкта. Для підтримки сумісності з доступом до файлової мережі, ім'я об'єктів не повинні мати префіксів або суфіксів з символом «/», а також суміжних декількох символів «/».
2. Управління обліковим записом. Сховище об'єктів не підтримує імена облікових записів (тобто імена файлів сховищ Red Hat Gluster), які мають підкреслення.

Отже, GlusterFS є масштабованою мережевою файловою системою, яка підходить для виконання завдань, що потребують великих обсягів даних, таких як хмарне сховище та потокова передача медіаданих.

1.2.3. HDFS

Apache Hadoop являє собою набір програмних утиліт, які полегшують роботу в мережі з багатьма комп'ютерами, для вирішення проблем, пов'язаних з великими обсягами даних і обчисленнями. Hadoop забезпечує програмну основу для розподіленого зберігання та обробки великих даних за допомогою моделі програмування MapReduce [13,14].

Apache Hadoop складається з наступних модулів [15]:

1. Hadoop Common – містить бібліотеки та утиліти, необхідні для інших модулів Hadoop;
2. Hadoop Distributed File System (HDFS) – розподілена файлова система, яка зберігає дані на товарних машинах, забезпечуючи дуже високу сукупну пропускну здатність по всьому кластеру;
3. Hadoop YARN – платформа, що відповідає за управління обчислювальними ресурсами в кластерах та їх використання для планування користувацьких додатків;
4. Hadoop MapReduce – реалізація моделі програмування MapReduce для високошвидкісної обробки великих обсягів даних на великих паралельних кластерах обчислювальних вузлів.

Розподілена файлова система Hadoop – це основна система зберігання даних, яка використовується додатками Hadoop [16,17]. HDFS багаторазово копіює блоки даних і розподіляє ці копії по обчислювальним вузлам кластера, тим самим забезпечуючи високу надійність і швидкість обчислень [18,19]:

- дані розподіляються по декількох машин під час завантаження;
- HDFS оптимізована більше для виконання потокових зчитувань файлів, ніж для нерегулярних, довільних зчитувань;
- файли в системі HDFS пишуться одноразово і внесення ніяких довільних записів в файли не допускається;
- додатки можуть зчитувати і писати файли розподіленої файлової системи безпосередньо через програмний інтерфейс Java.

У розподіленій файловій системі Hadoop застосовуються наступні концепції [20]:

1. HDFS не повинна мати обмежень на розмір даних.
2. Вихід з ладу вузла є нормальним явищем, так як, навіть на надійному фізичному пристрої, один з тисячі вузлів може перестати працювати.
3. Вузли в кластері не повинні потребувати додаткового адміністрування.

Клієнтами HDFS можуть бути будь-які додатки або користувачі, які взаємодіють з розподіленою файловою системою через спеціальний API. Вони працюють з HDFS так само як зі звичайною файловою системою – ієрархія каталогів з вкладеними в них підкаталогами і файлами [21,22].

Як і в звичайних файлових системах клієнтам, за наявності необхідних прав доступу, дозволені операції: створення, видалення, перейменування, переміщення файлів та папок [23].

1.3. Розширена постановка задачі

В ході виконання даного дипломного проекту передбачається розроблення програмних засобів, для реалізації розподіленої файлової системи.

Для цього необхідно розробити підсистему користувацького інтерфейсу та підсистему управління даними.

Для реалізації підсистеми користувацького інтерфейсу мають бути вирішені наступні задачі:

1. Написати клієнтський модуль, шо дозволить запустити код файлової системи в просторі користувача, для перевизначення системних викликів ядра.
2. Описати логіку передачі даних, а саме інтерфейс взаємодії між серверами та протокол обміну даними.

Клієнтський модуль має реалізувати наступні операції файлової системи:

1) перегляд вмісту директорій;

2) створення нового файлу;

3) відкриття файлу;

4) запис даних у файл;

5) зчитування даних з файлу;

6) закриття файлу.

Крім того, при розробці даних програмних засобів мають бути враховані наступні вимоги:

1. Для написання клієнтського модулю використати мову програмування C.
2. Застосувати високорівневий інтерфейс FUSE для написання файлових систем, без редагування коду ядра.
3. Надати можливість монтувати та розмонтувати файлову систему.

2. ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Вибір мови програмування

Мова програмування С – є універсальною, процедурною, імперативною мовою програмування, що підтримує структурне програмування. Вона була розроблена з метою написання операційної системи Unix [24].

Хоча мова С була призначена для написання системного програмного забезпечення, вона також часто використовується і для прикладного програмного забезпечення.

С є імперативною, процедурною мовою, яка була розроблена для компіляції з використанням відносно простого компілятора, щоб забезпечити низький рівень доступу до пам’яті, для використання мовних конструкцій, які ефективно відображаються на машинних інструкціях [25].

Незважаючи на низький рівень можливостей, мова С була розроблена для заохочення крос-платформного програмування. Програму, написану на мові С, можна скомпілювати для широкого кола комп’ютерних платформ і операційних систем з незначними змінами в вихідному коді.

Однією з найважливіших функцій мови програмування С є забезпечення засобів керування пам’яттю та об’єктів, що зберігаються в цій пам’яті. С забезпечує три різні способи виділення пам’яті для об’єктів [26]:

1) виділення статичної пам’яті: простір для об’єкта надається в двійковому режимі під час компіляції; ці об’єкти мають час життя до тих пір, поки бінарний файл, який їх містить, завантажується в пам’ять;
2) автоматичне виділення пам’яті: тимчасові об’єкти можуть зберігатися в стеку, і цей простір автоматично звільняється і повторно використовується після завершення блоку, в якому вони оголошені;
3) динамічне виділення пам’яті: блоки пам’яті довільного розміру можуть бути призначені під час виконання з використанням

бібліотечних функцій, таких як «malloc», з області пам'яті, що називається купою. Ці блоки зберігаються до тих пір, поки не звільняються для повторного використання, викликаючи бібліотечні функції «realloc» або «free».

Ці три підходи в різних ситуаціях мають різні компроміси. Наприклад, виділення статичної пам’яті має невеликі накладні витрати, автоматичне виділення пам’яті може мати трохи більше накладних витрат, а виділення динамічної пам’яті може потенційно мати великі накладні витрати як для розподілу так і для звільнення [27].

Постійний характер статичних об’єктів корисний для підтримки інформації про стан через виклики функцій, автоматичне виділення легко використовувати, але простір стека, як правило, набагато більш обмежений і перехідний, ніж статична пам’ять або купа, а динамічне виділення пам’яті дозволяє зручно розподіляти об’єкти, розмір яких стає відомим тільки під час виконання. Але все-таки більшість програм мовою С використовують всі три способи виділення пам’яті [28].

Там, де це можливо, автоматичне або статичне виділення, як правило, є найпростішим, оскільки управлінням займається компілятор, звільняючи програміста від потенційно схильної до помилок рутинної роботи з виділенням та вивільнення пам’яті вручну. Однак, багато структур даних можуть змінюватися в розмірі під час виконання, і оскільки статичні виділення (і автоматичні виділення до С99) повинні мати фіксований розмір під час компіляції, існує багато ситуацій, в яких необхідне динамічне виділення пам’яті [29,30].

Інша проблема полягає в тому, що виділення пам’яті в купі має бути синхронізовано з його фактичним використанням у будь-якій програмі, щоб її можна було повторно використати якомога більше. Наприклад, якщо виклик free() єдиного покажчика на виділення пам’яті виходить за рамки або має перезаписане його значення, то ця пам’ять не може бути відновлена для подальшого повторного використання і по суті втрачена для програми, це

явище, відоме як витік пам’яті. І навпаки, пам’ять можна звільнити, але продовжувати посилатися, що призводить до непередбачуваних результатів. Як правило, симптоми з’являтимуться у частині програми, яка далеко від фактичної помилки, що ускладнює відстеження проблеми [31].

Але не дивлячись на важкість написання програм на мові С, вона, імовірно, є найпопулярнішою мовою за кількістю написаного нею програмного забезпечення [32].

2.2. Вибір інтерфейсу для створення файлової системи

Filesystem in Userspace (FUSE) – це програмний інтерфейс для Unix і Unix-подібних операційних систем, що дозволяє непривілейованим користувачам створювати власні файлові системи без редагування коду ядра. Це досягається за допомогою запуску коду файлової системи в просторі користувача, тоді як модуль FUSE надає лише «міст» до фактичних інтерфейсів ядра [33].

FUSE доступний для Linux, FreeBSD, OpenBSD, NetBSD, OpenSolaris, Minix 3, Android і macOS.

Проект FUSE складається з двох компонентів: модуля fuse kernel (зберігається в звичайних сховищах ядра) і бібліотеки простору користувача libfuse [34]. Ця бібліотека забезпечує еталонну реалізацію для зв’язку з модулем ядра FUSE.

Для реалізації нової файлової системи libfuse треба записати програму-обробник, пов’язану з наданою бібліотекою. Основна мета цієї програми полягає в тому, щоб вказати, як файлова система реагує на запит read/write/stat. Програма також використовується для монтування нової файлової системи. Під час монтування файлової системи обробник реєструється ядром. Якщо користувач тепер видає запит на read/write/stat для цієї нещодавно встановленої файлової системи, ядро пересилає ці запити обробникові, а потім відправляє відповідь до користувача.

Бібліотека libfuse пропонує два API: «високий рівень», синхронний API і «низький» асинхронний API [35]. В обох випадках вхідні запити з ядра передаються в головну програму за допомогою зворотних викликів. При використанні API високого рівня, зворотні виклики можуть працювати з іменами файлів і шляхами замість inodes, і обробка запиту завершується, коли функція зворотного виклику повертається. При використанні низькорівневого API, зворотні виклики повинні працювати з inodes, а відповіді повинні бути відправлені явно, використовуючи окремий набір функцій API.

В даному дипломному проекті використовується API високого рівня.

FUSE особливо корисний для написання віртуальних файлових систем . На відміну від традиційних файлових систем, які, по суті, працюють з даними про масову пам’ять, віртуальні файлові системи фактично не зберігають самі дані. Вони діють як перегляд або переклад існуючої файлової системи або пристрою зберігання даних.

В принципі, будь-який ресурс, доступний для реалізації FUSE, може бути експортований як файлова система.

Очевидно, що реалізація файлової системи у просторі користувача є великою перевагою, так як через це є можливість використовувати будь-яку з доступних бібліотек, на відміну від простору ядра, яке потребує глибокого його розуміння.

3. АРХІТЕКТУРА ФАЙЛОВОЇ СИСТЕМИ

Структура даної розподіленої файлової системи складається з двох підсистем: підсистеми користувацького інтерфейсу та підсистеми управління даними (див. Додаток 1).

Підсистема користувацького інтерфейсу поділяється на дві частини.

Перша частина – це модуль адміністрування файлової системи. Цей модуль призначений для управління кластером. Доступ до нього мають тільки розробники файлової системи.

Друга частина – це клієнтський модуль, він є загальнодоступний і використовується як головний інтерфейс для взаємодії з файловою системою.

Підсистеми управління даними, яка відповідає за логіку роботи розподіленої файлової системи, теж поділяється на дві частини.

Перша частина – це проміжний сервіс керування мережевими з’єднаннями. Він використовується для комунікації клієнтського модулю та модулю адміністрування з кластером файлової системи.

Друга частина – це сам кластер. Він є головною частиною файлової системи. Його можна поділити на дві складові. Це вузли кластера – невеликі сервера, на який зберігаються дані з файлів, та кластер-менеджер – сервер, який відповідає за логіку роботи з файлами. Кожен вузол має свою невеличку базу даних, доступ до якої має лише він. А кластер-менеджер взаємодіє з головною великою базою даних, на якій зберігається інформація про файли в розподіленій файловій системі.

3.1. Клієнтський модуль

Реалізація файлової системи є складним завданням, що вимагає написання деякої її частини на рівні ядра. Але в даному дипломному проекті, ми не пишемо реальну файлову систему, а лише обгортку поверх вже існуючої. Для цього ми використовуємо FUSE – модуль для Unix-подібної операційної системи, який дозволяє користувачам без спеціальних

прав та без модифікації ядра створювати власні файлові системи. Це стає можливим завдяки тому, що драйвер файлової системи працює в просторі користувача, а модуль FUSE забезпечує «міст» до поточних інтерфейсів ядра, що дозволяє нам перевизначити системні виклики ядра. Тобто, замість викликів open, write, read, close та інших, викликаються обробники, які ми реалізуємо в клієнтському модулі. Саме там, ми можемо відправити запит до нашого сервера, а відповідь зберегти у структурах, які необхідні операційній системі для роботи з файлами та папками.

3.1.1. Модуль FUSE

Найважливіше, що потрібно знати при роботі з FUSE – це те, як він працює. Для цього розглянемо схему роботи FUSE (див. Додаток 1).

На схемі можна побачити, що запит «ls –l /tmp/fuse» на перегляд вмісту директорії з назвою «fuse» з простору користувача перенаправляється ядром через віртуальну файлову систему (VFS) до самого модулю FUSE. FUSE потім виконує зареєстровану програму-обробник «./hello» і передає їй запит «ls –l /tmp/fuse» у потрібному форматі. А після виконання цієї операції відповідь обробника повертається назад до FUSE, який потім перенаправляє її до програми користувача, що ініціалізувала запит.

Програма-обробник – це і є та частина клієнтського модулю, яку необхідно реалізувати для створення інтерфейсу для взаємодії з розподіленою файловою системою.

Але не менш важливо розуміти як працює сама бібліотека FUSE (libfuse). Алгоритм її роботи (див. Додаток 1) можна описати наступним чином [36]:

1. Коли програма-обробник викликає функцію fuse_main(), ця функція аналізує аргументи, передані цій програмі, а потім викликає fuse_mount().

2. Функція fuse_mount() створює сокетну пару і продовжує паралельний процес, в якому запускає fusermount, передаючи один кінець сокета в змінну середовища FUSE_COMMFD_ENV.
3. Далі fusermount переконується, що модуль FUSE завантажений, а потім відкриває файл /dev/fuse і відправляє дескриптор файлу через сокет назад до fuse_mount().
4. Функція fuse_mount() повертає дескриптор файлу /dev/fuse в fuse_main().
5. Після чого, fuse_main() викликає функцію fuse_new(), яка створює структуру fuse datastructure, яка зберігає і кешує образ даних файлової системи.
6. І в кінці, fuse_main() викликає або fuse_loop() функцію, або fuse_loop_mt(), яка починає перехоплювати файлові системні виклики з /dev/fuse і, реагуючи на них, викликає відповідні функції в просторі користувача, які збережені в структурі fuse_operations (рис. 1) перед викликом fuse_main(). А результати цих функцій записуються назад у файл /dev/fuse, звідки вони можуть бути передані назад до системних викликів.

```
static struct fuse_operations fuse_example_operations = {
  .getattr = getattr_callback,
  .open = open_callback,
  .read = read_callback,
  .readdir = readdir_callback,
};

int main(int argc, char *argv[]) {
  return fuse_main(argc, argv, &fuse_example_operations, NULL);
}
```

Рис. 1. Приклад запуску програми-обробника та списку зворотних викликів

Структура «fuse_operations» описує операції, які можливо перевизначити для реалізації своєї файлової системи. Більшість із них має

працювати дуже подібно до відомих операцій файлової системи UNIX. Головною відмінністю є те, що замість того, щоб повернути помилку в «errno», операція повинна безпосередньо повернути негативне значення помилки «-errno».

Структура «fuse_operations» представляє собою головний API модулю FUSE. До цієї структури входять наступні операції [37]:

1. int* getattr (const char*, struct stat*, struct fuse_file_info*)
2. int* readlink (const char*, char*, size_t)
3. int* mknod (const char*, mode_t, dev_t)
4. int* mkdir (const char*, mode_t)
5. int* unlink (const char*)
6. int* rmdir (const char*)
7. int* symlink (const char*, const char*)
8. int* rename (const char*, const char*, unsigned int flags)
9. int* link (const char*, const char*)
10. int* chmod (const char*, mode_t, struct fuse_file_info*)
11. int* chown (const char*, uid_t, gid_t, struct fuse_file_info*)
12. int* truncate (const char*, off_t, struct fuse_file_info*)
13. int* open (const char*, struct fuse_file_info*)
14. int* read (const char*, char*, size_t, off_t, struct fuse_file_info*)
15. int* write (const char*, const char*, size_t, off_t, struct fuse_file_info*)
16. int* statfs (const char*, struct statvfs*)
17. int* flush (const char*, struct fuse_file_info*)
18. int* release (const char*, struct fuse_file_info*)
19. int* fsync (const char*, int, struct fuse_file_info*)
20. int* setxattr (const char*, const char*, const char*, size_t, int)
21. int* getxattr (const char*, const char*, char*, size_t)
22. int* listxattr (const char*, char*, size_t)
23. int* removexattr (const char*, const char*)
24. int* opendir (const char*, struct fuse_file_info*)

25. int* readdir (const char*, void*, fuse_fill_dir_t, off_t, struct fuse_file_info*, enum fuse_readdir_flags)
26. int* releasedir (const char*, struct fuse_file_info*)
27. int* fsyncdir (const char*, int, struct fuse_file_info*)
28. void** init (struct fuse_conn_info*, struct fuse_config*)
29. void* destroy (void*)
30. int* access (const char*, int)
31. int* create (const char*, mode_t, struct fuse_file_info*)
32. int* lock (const char*, struct fuse_file_info*, int cmd, struct flock*)
33. int* utimens (const char*, const struct timespec tv[2], struct fuse_file_info*)
34. int* bmap (const char*, size_t blocksize, uint64_t* idx)
35. int* ioctl (const char*, unsigned int cmd, void* arg, struct fuse_file_info*, unsigned int flags, void* data)
36. int* poll (const char*, struct fuse_file_info*, struct fuse_pollhandle* ph, unsigned* reventsp)
37. int* write_buf (const char*, struct fuse_bufvec* buf, off_t off, struct fuse_file_info*)
38. int* read_buf (const char*, struct fuse_bufvec** bufp, size_t size, off_t off, struct fuse_file_info*)
39. int* flock (const char*, struct fuse_file_info*, int op)
40. int* fallocate (const char*, int, off_t, off_t, struct fuse_file_info*)
41. ssize_t* copy_file_range (const char* path_in, struct fuse_file_info* fi_in, off_t offset_in, const char* path_out, struct fuse_file_info* fi_out, off_t offset_out, size_t size, int flags)

Всі ці операції не є абсолютно необхідними для реалізації своєї файлової системи, але деякі з них потрібні для коректної роботи. Це залежить від поставленої задачі, яку повинна виконувати файлова система. Так, наприклад, операції open, flush, release, fsync, opendir, releasedir, fsyncdir, access, create, truncate, lock, init і destroy є спеціальними методами,

без яких все ще може бути реалізована повнофункціональна файлова система.

Ще однією важливою структурою є fuse_file_info – це структура, яка описує інформацію про відкритий файл або папку. Вона заповнюється у реалізованих методах open, opendir або create.

До даної структури входять наступні поля, які необхідні для коректної роботи програми-обробника [38]:

1. flags (int)

   Відкриті прапори. Доступні в функціях open() та release().

2. writepage (unsigned int)

   У разі операції запису вказує, чи була вона викликана затримкою запису з кешу сторінки. Якщо так, то поля pid, uid і gid контексту не будуть дійсними, і значення fh може не збігатися зі значенням fh, яке було б надіслано з відповідними індивідуальними запитами запису, якщо кешування запису було вимкнено.

3. direct_io (unsigned int)

   Встановлюється функцією open(), щоб використовувати прямий ввід/вивід для цього файлу.

4. keep_cache (unsigned int)

   Встановлюється функцією open(). Сигналізує ядру, що всі поточні кешовані файлові дані (тобто дані, які файлова система надала при останньому відкритті файлу), не повинні бути анульовані. Не має ефекту при встановленні в інших контекстах (зокрема, він нічого не робить, коли його встановлює функція opendir()).

5. flush (unsigned int)

   Вказує на функцію flush(). Встановлюється в функції flush(). А також може бути встановлено в режимі lock високого рівня і операції release низького рівня.

6. nonseekable (unsigned int)

Встановлюється функцією open(), вказує що файл не можна шукати.

7. cache_readdir (unsigned int)

   Встановлюється функцією opendir(). Сигналізує ядру про необхідність дозволити кешування записів, повернутих readdir(). Не має ефекту при встановленні в інших контекстах (зокрема, він нічого не робить, коли його встановлює функція open()).

8. padding (unsigned int)

   Зарезервовано для майбутнього використання.

9. fh (uint64_t)

   Файловий ідентифікатор, який може буди заповнений в функціях create(), open() та opendir().

10. lock_owner (uint64_t)

    Блокує ідентифікатор власника. Доступний в операціях блокування та flush().

11. poll_events (uint32_t)

    Запитувані «poll» події. Доступний в функції poll(). Встановлюється лише на ядрах, які його підтримують. Якщо не підтримується, це поле встановлюється рівним нулю.

3.1.2. Обмін даними

Взаємодія між клієнтською частиною і проміжним сервером відбувається за допомогою протоколу керування передачею даних у комп’ютерних мережах (TCP). Він забезпечує надійне пересилання даних від клієнта-відправника до сервера-отримувача.

В даній розподіленій файловій системі цей протокол використовується для відправки команд на сервер та для отримання відповіді на ці команди.

А для передачі даних файлу використовується протокол датаграм користувача (UDP). Він не встановлює з’єднання з сервером-отримувачем,

тому і не гарантує доставку даних, але, саме через це, працює швидше ніж TCP і може ефективно відправляти великий обсяг даних.

Дані файлу по UDP передаються у вигляді пакетів. Пакет містить самі дані та допоміжну інформацію, яка необхідна для роботи з цими даними.

Щоб не втрачати пакети при передачі по комп'ютерній мережі, перевіряється їх цілісність. Якщо пакет був втрачений або дані пошкоджені, то він пересилається знову. Це гарантує доставку пакету серверу-отримувачу.

3.2. Бази даних файлової системи

Для збереження інформації у системі використовується дві SQL бази даних, одна з яких – це PostgreSQL. Ця база вибрана через те, що ми маємо визначену структуру полів, яка не змінюється від вхідних даних, а також тому що вона дуже швидка та надійна. PostgreSQL необхідна для збереження інформації про файли та вузли. Кластер-менеджер може додавати, змінювати, редагувати та видаляти дані з цієї бази даних. Також до неї є доступ з проміжного сервера, але тільки для читання, що дає можливість зосередити усю логіку файлової системи в кластер-менеджері.

Другою з використаних баз даних є SQLite, вона проста у використанні та займає дуже мало місця. Ця база даних зосереджена на вузлі кластера. Тобто кожен вузол має свою SQLite базу даних, де зберігається інформація про пакети, збережених на цьому вузлі, а також загальна інформація про вузол. І саме через те, що кількість цих даних не така велика, ми можемо використати цю базу даних.

Центральною базою даних даної розподіленої файлової системи є база даних, в якій зберігається інформація про файли та підключені вузли. Ця база складається з трьох таблиць: File, Package та Node.

File – є головною таблицею, де розміщуються дані про файл, а саме:

- id – унікальний ідентифікатор файлу в таблиці;

- pathname – повний шлях цього файлу, що включає в себе назви всіх батьківських каталогів, розділених слешем, або, інакше кажучи, повний шлях до батьківського каталогу та назва файлу;
- type – тип файлу, що представляє собою одне з перелічених значень у вигляді символів, запозичених у UNIX подібних операційних системах. Наприклад, звичайний файл – «f», або каталог – «d»;
- data – залежно від типу, може містити різну інформацію. Якщо типом є директорія, то в цьому полі зберігається список ідентифікаторів дочірніх файлів. А якщо звичайний файл, то список ідентифікаторів пакетів даних з таблиці Package.
- size – розмір файлу у байтах;
- order_num – кількість пакетів в файлі. А для папки це значення завжди є нулем.
- status – статус файлу, який необхідний для відстеження етапу роботи над файлом, наприклад, при запису даних в файл, по цьому значенню можна перевірити чи всі дані записались.

В табл. 1 наведені поля таблиці File та їх характеристики.

Таблиця 1

Характеристики полів таблиці File

Назва поля	Тип	Обов’язковість	Унікальність
id	Integer	Так	Так
pathname	String	Так	Так
type	Enum	Так	Ні
data	Array	Так	Ні
size	Integer	Так	Ні
order_num	Integer	Так	Ні
status	Enum	Так	Ні

Package – це таблиця, яка описує інформацію про місцезнаходження даних файлу на вузлах, а саме:

- pack_id – унікальний ідентифікатор даних в таблиці, він також вказує, де шукати дані файлу на конкретному вузлі;
- file_id – ідентифікатор файлу в таблиці File, що вказує, до якого файлу відноситься даний пакет;
- node_id – ідентифікатор вузла в таблиці Node, що вказує, на якому вузлі знаходиться даний пакет;
- next_package_id – ідентифікатор, що вказує на пакет в цій таблиці, який є наступним після даного пакету.
- status – статус пакету, який потрібен для відстеження етапу роботи над пакетом, наприклад, при збереженні пакету.

В табл. 2 наведені поля таблиці Package та їх характеристики.

Таблиця 2

Характеристики полів таблиці Package

Назва поля	Тип	Обов’язковість	Унікальність
pack_id	Integer	Так	Так
file_id	Integer	Так	Ні
node_id	Integer	Так	Ні
next_package_id	Integer	Ні	Так
status	Enum	Так	Ні

Node – це таблиця, де зберігається інформація про підключені до кластера вузли, а саме:

- node_id – унікальний ідентифікатор вузла в таблиці;
- ip – ip-адреса вузла в локальній або глобальній мережі, яка необхідна для взаємодії з вузлом;
- tcp_port – TCP порт вузла;

- udp_port – UDP порт вузла;

В табл. 3 наведені поля таблиці Node та їх характеристики.

Таблиця 3

Характеристики полів таблиці Node

Назва поля	Тип	Обов’язковість	Унікальність
node_id	Integer	Так	Так
ip	String	Так	Ні
tcp_port	Integer	Так	Так
udp_port	Integer	Так	Так

Packages – це таблиця, де розміщується інформація про пакети, збережені на даному вузлі, а саме:

- id – ідентифікатор пакету в даній таблиці;
- package_id – ідентифікатор пакету в таблиці Package.
- block_offset – зміщення відносно початку сховища, яке вимірюється в кількості пакетів;
- real_size – кількість актуальних даних, що не перевищує максимального розміру пакета.

В табл. 4 наведені поля таблиці Packages та їх характеристики.

Таблиця 4

Характеристики полів таблиці Packages

Назва поля	Тип	Обов’язковість	Унікальність
id	Integer	Так	Так
package_id	Integer	Так	Так
block_offset	Integer	Так	Так
real_size	Integer	Так	Ні

3.3. Модуль взаємодії

Модуль взаємодії є центральним елементом всієї архітектури описуваної файлової системи, він відповідає за отримання запитів і команд користувача та їх обробку. Також даний модуль відповідає за систему контролю доступності об’єктів для конкретного користувача. Модуль складається з декількох окремих частин:

1. Підмодуль адміністрування кластера – створений для керування структурою кластера. Він надає адміністратору API для зміни кластера чи його частин.
2. Підмодуль взаємодії з файловою системою – створений для взаємодії користувача з файловою системою. Приймає та оброблює запити користувача. Слугує проксі-сервером при завантаженні даних користувачем (при записі і читанні).
3. Механізм передачі даних та команд. Описує загальний спосіб комунікації між користувачем, системою та елементами системи.

Модуль взаємодії має доступ до бази даних файлової системи, але може виконувати тільки операції читання, зміна бази даних чи окремих записів йому недоступна. Це необхідно для підвищення безпеки системи.

Модуль взаємодії надає користувачу API для роботи з кластером та файловою системою та визначає спосіб та структуру взаємодії між ними. Саме в цьому елементі системи описуються механізми передачі даних файлів та механізми реагування на команди користувача, ці механізми є спільними для всієї системи і всюди працюють однаково.

3.3.1. Підмодуль адміністрування кластера

Надає можливість адміністратору змінювати структуру кластера, змінювати параметри балансування навантаження, відключати та додавати вузли. Являє собою WEB-сервер, який надає адміністратору можливість перегляду інформації про кластер (без перегляду даних на кластері). Структурно складається безпосередньо з WEB-сервера, та, дещо обмеженого, механізму передачі даних, необхідного для зв’язку з кластер-

менеджером. Доступ до цієї функціональності має виключно адміністратор кластера, звичайні користувачі такої можливості позбавлені.

Підмодуль адміністрування, для збереження інформації про кластер, використовує, спільно з кластер-менеджером, відповідну базу даних, але як і у випадку з підмодулем взаємодії з файловою системою може тільки читати з бази даних, а всі зміни змушений проводити через кластер-менеджер. Для відслідковування ролі користувача використовуються спеціальні криптографічні ключі, створені на основі пароля адміністратора з подальшим його шифрування. Такі ключі є унікальними для кожного адміністратора і зберігаються в відповідній таблиці користувацької бази даних.

3.3.2. Підмодуль взаємодії з файловою системою

Даний підмодуль є головним елементом взаємодії клієнта з файловою системою. Саме ця частина програми відповідає за обробку команд клієнта. Підмодуль взаємодії з файловою системою умовно складається з механізму передачі даних і безпосередньо самого модуля (який є Worker-ом для Transmitter). Даний підмодуль – єдине місце системи, яке взаємодіє з усіма учасниками системи, саме через нього користувач керує своїми даними, адміністратор – кластером. Ця частина є відповідальною за прийом даних (UDP даних) від користувача і їх подальший запис, причому в даному процесі підмодуль є проксі-сервером через який проходять дані, також саме тут зберігаються всі обробники для користувацьких викликів.

Кожен з обробників є незалежним один від одного потоком з власною чергою команд. Основною особливістю даних обробників є те, що вони не постійно виконують певні дії, а «засинають» при простої впродовж певного часу, а після додавання до їх черги нового завдання «прокидаються» і продовжують працювати. Така можливість досягається шляхом розширення стандартного класу Thread в мові програмування Python, додаванням до нього таймеру простою, та можливості закінчувати виконання потоку за командою. Данна можливість є неймовірно корисною, адже дозволяє

максимально пришвидшити та розпалалелити роботу програми, при цьому мінімізуючи непотрібні ресурсні затрати. Також плюсом такої системи обробників є те, що вона неймовірно легко розширюється, для цього просто необхідно зареєструвати новий обробник (така можливість передбачена в підмодулі).

При надходженні даних від клієнта підмодуль визначає, який це тип даних (UDP/TCP) і переправляє вже до конкретних обробників, які оброблюють команди. Так, наприклад, при надходженні TCP команди дана команда перенаправляється в спеціальний TCPHandler в якому визначається тип команди (співпадає з функціями FUSE, наприклад: open, write, flush, тощо), далі команди передаються в конкретні обробники, які вже займаються безпосередньою обробкою. В ході опрацювання команди, при необхідності, обробники можуть відправляти запити в ClusterManager (Менеджер кластера), наприклад, при записі нових даних, відправляється запит на «балансування» цих даних по вузлу (визначення оптимального вузла зберігання для конкретного пакету).

Підмодуль взаємодії з файловою системою має прямий доступ до бази даних файлової системи, але йому доступне тільки читання решта функцій йому недоступні. Це зроблено для того, щоб максимально розвантажити цей підмодуль, адже він є вузьким місцем системи і найбільше впливає на її продуктивність), а також унеможливити проблеми з одночасним записом в базу даних з різних серверів.

3.3.3. Механізм передачі даних та команд

Механізм передачі даних та команд (Transmitter) – асинхронний модуль створений для передачі даних та команд між усіма учасниками файлової системи. Складається з декількох логічних частин:

1. Receiver – отримує дані по протоколам TCP і UDP, у випадку UDP підтверджує прийом пакету (детальніше нижче), та передає отримані дані в Interaction.

2. Interaction (черга обробки) – черга запитів, яка доступна одночасно всім іншим учасникам передачі даних, потрібна для передачі даних між частинами. Також гарантує порядок надходження команд до обробника. Умовно поділяється на дві підчастини:
    a) Receiver Interaction – є спільною частиною для Receiver і Worker, використовується, для передачі команд і запитів між ними;
    b) Sender Interaction – є спільною частиною для Worker і Sender, слугує для передачі відповідей між ними.
3. Worker – обробник запитів, дістає запит з черги, оброблює та формує на нього відповідь, далі сформована відповідь потрапляє в Sender Interaction.
4. Sender – дістає відповіді з Sender Interaction і передає дані відповідним адресатам.

Кожна з вищеназваних частин, крім Interaction, що слугує засобом передачі інформації між потоками, є незалежною один від одної і працюють паралельно, завдяки механізму багатопоточності. Так як вказаний вище механізм є асинхроний, він не гарантує дотримання порядку відповіді відповідно до порядку прийому команд, проте дозволяє забезпечити пріоритетність команд (TCP запити) над даними (UDP запити), для цього використовується абстрактна структура – черга за пріоритетами. Також дана система гарантує, що TCP команди надійдуть у виконання згідно з їх отриманням на сервері.

Receiver

Receiver – частина Transmitter-а, яка відповідає за постійний та безперебійний прийом даних (у вигляді TCP запитів чи UDP пакетів). Поділяється на дві частини відповідно до типу приймаючих даних, тобто перша частина приймає тільки TCP запити, а інша тільки UDP. Кожна з цих частин є окремим потоком і не залежать один від одного (крім моменту запису даних в чергу обробки). Підчастини Receiver подані у вигляди

неблокуючих сокетів відповідного протоколу, які слугують серверами, що здатні приймати вхідні повідомлення.

Для коректної роботи UDP частини стандартний протокол обміну даних був загорнутий в написаний додатковий шар, який розбиває об’єм даних на блоки фіксованого розміру, додає до них хедера (порядковий номер, фактичний розмір блоку, контрольну сума, тощо) та відправляє отримувачу; основною зміною є впровадження контрольної суми, яка використовується для перевірки правильності отримування даних.

Після прийому даних, вони кладуться в чергу виконання. У випадку, якщо дані надійшли по UDP, перевіряється контрольна сума і в разі її невідповідності відправляється запит на повторне отримання даних, якщо в проміжку певного часу не приходить відповідь – пакет вважається таким, що загубився і запит відправляється ще раз, так відбувається певну, визначену кількість разів або доки пакет не прийде коректним.

Прийом даних по протоколу TCP відбувається в два етапи:

1. Зчитується розмір вхідного повідомлення: тобто при кожному отримані інформації на сокет, спочатку зчитуються 4 байти, які описують розмір вхідного повідомлення.
2. Зчитується безпосередньо саме повідомлення заданого розміру.

Така процедура необхідна через те, що TCP сокет всього один і є неблокуючим, тобто одночасно на нього можуть бути записані декілька команд різного розміру, яку необхідно розділити.

Черга обробки є спільною як для TCP частини, так і для UDP частини. Оскільки TCP та UDP частини є незалежними один від одного потоками, для їх взаємодії з чергою використовується м’ютекс.

Interaction

Interaction – частина взаємодії, саме через цей шар відбувається передача даними між елементами Transmitter. Ця частина представлена у вигляді черги, в залежності від ситуації вона може бути, або звичайною, або з пріоритетами. Для можливості використання, одночасно з декількох

різних потоків, а також для внесення даних в чергу, або видалення їх з черги використовується м’ютекс, який блокується при виклику відповідної функції одним з потоків і не дає можливості іншим потокам як-небудь впливати на зміст черги. Після завершення операції м’ютекс розблоковується і наступний потік може виконувати дії над чергою.

В залежності від місця перебування цього шару (між Receiver та Worker, чи між Worker та Sender), використовуються різні типи черга. Так, в першому випадку використовується черга з пріоритетом, яка є спільна, як для TCP Receiver, так і для UDP Receiver. Необхідність використання черги з пріоритетами обумовлена тим, що команди, що надходять по TCP, є важливішими ніж дані, що надходять по UDP, і вимагають максимально швидкої обробки. Це дозволяє забезпечити працездатність системи навіть при записі великих об’ємів даних. В випадку Worker та Sender, частина взаємодії представлена двома незалежними один від одного чергами, окремо для TCP Sender і для UDP Sender. Також в другому випадку особливістю є те, що UDP частина написана на мові C, що пояснюється необхідністю взаємодії з пакетами сформованими в цій мові.

Worker

Worker (обробник) – обробник запитів, є модульно-залежним (тобто в залежності від місця використання, може бути різний). Основний сенс цієї частини – це обробка запитів та, у разі необхідності, формування відповідей на них.

Структурно обробник складається з декількох частин:

1. InteractionListener – частина, яка прослуховує Interaction чергу та дістає з неї дані, по мірі їх надходження. Далі дані передаються в основний обробник.
2. Worker – основний обробник, саме тут визначається тип пакету (UDP/TCP) і приймається рішення стосовно того як його оброблювати. За замовчуванням кожний з обробників конкретного

типу пакетів є функцією, але в разі необхідності, може бути замінений специфічним обробником.

3. AnswerCreator – конструктор відповіді. Відповідає за формування відповіді користувачу з подальшим поміщенням відповіді в Interaction чергу.

Всі вищеназвані частини, за замовчуванням є синхронними і визначеними, але за необхідністю можуть бути змінені чи розширенні, для цього використовується механізм обробників-колбеків, який дозволяє користувачу передавати необхідні йому обробники.

Sender

Sender – відправник. Частина, структурно схожа з Receiver-ом, але виконує протилежні функції.

Поділяється на дві частини у відповідності до двох вищезгаданих протоколів. Кожна з двох частин має свою окрему чергу відповідей, з якої дістає сформовані відповіді і відправляє їх адресату. Адресат визначається на етапі опрацювання Worker-ом і разом з відповіддю кладеться в відповідну чергу.

TCP Sender підключається до отриманого адресу і відправляє на нього дані, після цього з’єднання закривається і на обробку береться нові дані.

UDP Sender відправляє дані на отриману адресу та чекає підтвердження отримання даних, якщо через певний, наперед визначений час, підтвердження не надходить пакет вважається втраченим і відправляється ще раз. Якщо клієнт присилає запит на повторну відправку даних (якщо контрольні суми не співпадають), відбувається повторне надсилання даних.

3.4. Модуль збереження даних

3.4.1. Менеджер кластера

Керуючий елемент кластера. Відповідає за додавання/видалення нових вузлів до кластера (їх облік в кластері), аналіз їх статусу,

балансування навантаження на кластер (розподіл даних між вузлами кластера). Єдиний елемент системи, який має прямий доступ, до бази даних системи і може здійснювати її редагування.

Структурно кластер-менеджер складається з декількох окремих, незалежних одна від одної частин:

1. Health monitor – відповідальний за підтвердження активного статусу вузла, кожен визначений користувачем період часу (за замовчуванням це 5 секунд) відправляє на конкретний вузол кластера запит, якщо впродовж двох хвилин відповідь не приходить, то вузол кластера вважається вимкнутим.
2. Data mapper – слугує для зручного підключення бази даних до об’єктної моделі програми. Він використовується для взаємодії з базою даних, транслює виклики методів класу в запити в базу даних (в даному випадку в базу даних файлової системи).
3. Main Worker – сам кластер-менеджер, він оброблює запити від Middleware, Cluster, та модулю взаємодії з кластером, формує відповіді. Саме в цій частині відбуваються такі процеси як баланс навантаження на кластері, реєстрація отриманих пакетів і зміна їх статусу після запису. В цій частині відбувається зміна бази даних файлової системи, саме тут додаток зберігає метадані про файлову систему і її частини. При цьому доступу до самої інформації Cluster Manager не має, він лише знає, в якому місці кластера знаходиться пакет, проте не може надіслати запит на зчитку тих даних. Такий підхід гарантує максимальну безпеку даних.

3.4.2. Вузли кластера

Нода або вузол кластера – основна складова частина кластера, саме на ній система зберігає блоки даних. Структурно вузол складається з декількох частин:

- Transmitter – використовується для прийому команд і даних.

- Storage – модуль, який займається безпосереднім зберіганням та реплікацією даних. Визначає який блок пам’яті куди записати і веде облік вільних та зайнятих блоків пам’яті, зберігає метадані та змінює їхні відповідники в базі даних.
- Data Mapper – слугує для зручного поєднання об’єктів в базі даних і в Storage.
- Database – база даних вузла. Використовується для зберігання метаданих вузла.

Основною функцією вузла є зберігання та реплікація даних. Зберігання відбувається шляхом прийому пакету даних по UDP Transmitter з подальшим записом на жорсткий диск і сигналізування кластер-менеджеру, про запис конкретного пакету. Видалення даних відбувається в зворотному порядку. При зчитці даних сигналізування до кластер-менеджера не відбувається.

Для реплікації даних, вузол обирає задану кількість своїх «сусідів» (за замовчуванням їх два), вибір проводиться з урахуванням завантаженості «сусіда» по кластера. Далі вузол пересилає прийнятий пакет даних на обрані «сусідні» вузли. Такий підхід дозволяє значно підвищити відмовостійкість системи, а також розгрузити інші елементи системи, поклавши роботу по реплікації даних безпосередньо на сам кластер.

4. СОБЛИВОСТІ РЕАЛІЗАЦІЇ ПРОГРАМНИХ ЗАСОБІВ

Для реалізації роботи клієнтського модулю необхідно було використати модуль для створення файлових систем в просторі користувача, тобто в адресному просторі віртуальної пам’яті операційної системи, що відводиться для користувацьких програм.

Для такої задачі було розглянуто чотири рішення: FUSE, NFS [39], Samba [40], WebDAV [41].

З цих чотирьох рішень було вибрано FUSE. Цей модуль дозволяє реалізовувати власну файлову систему як повний автономний модуль для простору користувача без необхідності вносити зміни в ядро і встановлювати файлову систему як модуль ядра.

4.1. Підсистема користувацького інтерфейсу

Клієнтський модуль ділиться на дві частини, точніше кажучи, на два потоки. Перший потік – це основний процес модулю FUSE. У ньому реєструються та виконуються обробники операцій файлової системи. А також він створює другий потік, який займається відправкою пакетів даних на проміжний сервер взаємодії (див. Додаток 1).

При ініціалізації програми-обробника створюються два сервери: один TCP сервер та один UDP. TCP сервер необхідний для прийому відповідей від сервера взаємодії на відправлені команди. UDP сервер, відповідно, – для прийому пакетів з даними файлу.

TCP клієнт створюється кожен раз при запиті на сервер взаємодії. Таким чином він може бути ініціатором подій, для яких непотрібна відповідь. Такий варіант використовується в обробнику операції запису даних у файл.

Програма-обробник також містить хеш-таблицю, яка необхідна для коректного порядку даних при записі у файл. Так як файлова система не гарантує, що пакети, передані по UDP, оброблюються послідовно один за одним.

Також клієнтський модуль має свій журнал для збереження інформації про події, які відбуваються у ході роботи програми-обробника.

4.1.1. API передачі даних

Структура TCP запитів та відповідей в даній розподіленій файловій системі складається з двох частин. У першій частині зберігається розмір даних другої частини. Це необхідно для того, щоб сервер-отримувач знав скільки байтів даних необхідно зчитати з сокета. Цей розмір займає рівно чотири байти. У другій частині розміщується інформація про те, що необхідно зробити або про результат цієї роботи. Ця інформація має свою особливу структуру. Параметри, які в ній передаються, розділені спеціальним символом – амперсандом «&».

Інформація, яка передається в TCP запиті (рис. 2) починається з назви команди, яку необхідно виконати, та аргументів, які їй необхідні. Дуже часто останнім параметром передається порт TCP сервера клієнтського модуля. Це необхідно для того, щоб сервер взаємодії знав, якому користувачу відправити відповідь. При цьому IP-адресу він бере з інформації про TCP запит.

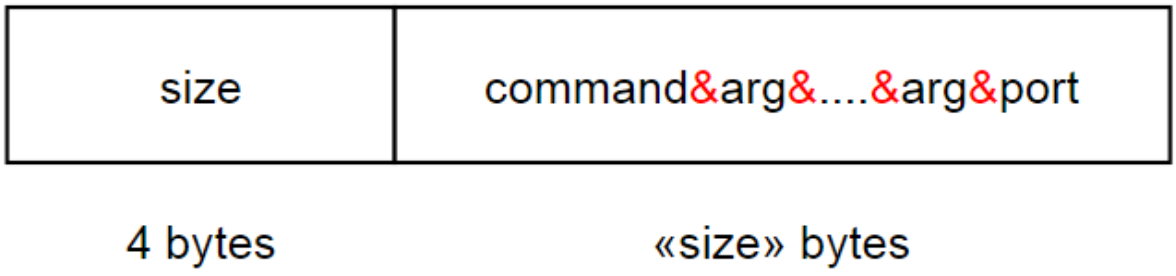


Рис. 2. Структура TCP запиту

А інформація, яка передається в TCP відповіді (рис. 3) складається зі статусу виконання та результатів. Статус в даній розподіленій файловій системі може набувати двох цілочисельних значень: 0 або 1. Статус 1 відповідає за коректне відпрацювання, а 0 повертається при помилках. При такому форматі є можливість додавати нові значення помилок, змінюючи

лише число, яке відповідає за статус. Це може бути корисним при перевірці статусу, якщо необхідно буде по-різному відреагувати на код помилки.

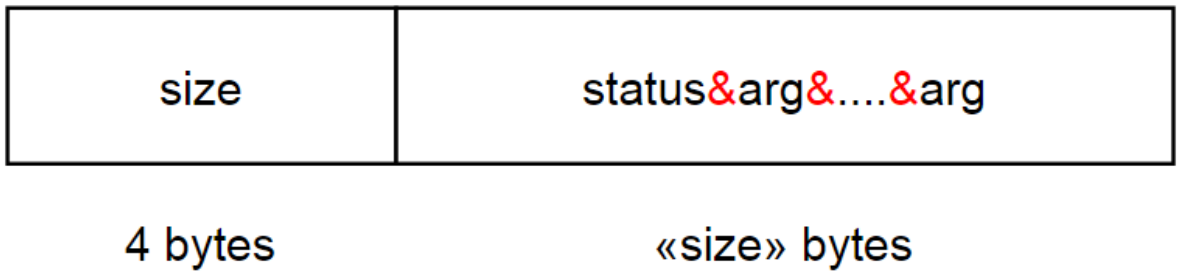


Рис. 3. Структура TCP відповіді

Залежно від статусу в результатах відповіді розміщуються відповідна інформація. У випадку коректного статусу, повертаються параметри, які необхідні клієнту для продовження роботи. А якщо сталася помилка, то результатом є текстове повідомлення, у якому вказується детальна інформація про помилку. При цьому на стороні клієнта ця помилка зберігається в журналі повідомлень. Також при цьому є можливість при поверненні помилки передати не одне, а декілька повідомлень через амперсанд. В результаті чого в журналі додасться два окремих записи з текстами помилок.

Передача даних по UDP має свою визначену структуру (рис. 4), яка не змінюється залежно від того чи дані відправляються на сервер чи приймаються. Пакет, який при цьому передається, складається з заголовків та даних в певній послідовності.

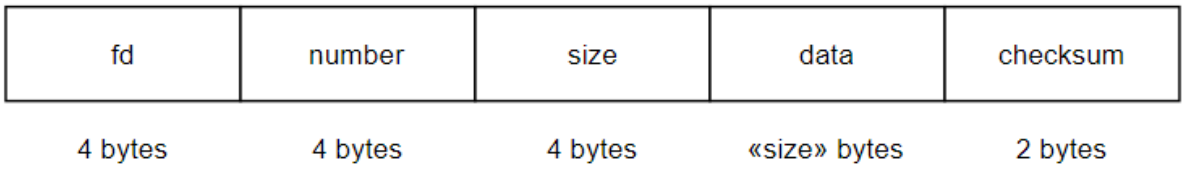


Рис. 4. Структура пакету даних

Спочатку передається файловий дескриптор, необхідний для того, щоб сервер знав до якого файлу відносяться ці дані. Так як файловий дескриптор часто використовуватися в коді програм, то він розміщується першим, для зручності доступу до нього.

Після ідентифікатора файлу розташовується номер цього пакету в послідовності відправлених пакетів. Він необхідний для того, щоб при зчитуванні даних відновити їх правильний порядок.

Наступним йде розмір самих даних, які складають основну частину пакету. В даній розподіленій файловій системі було обрано максимальний розмір цих даних, який становить 512 байт. Але дані можуть займати і менше місця ніж це значення. У такому випадку порожні байти заповнені нулями. Саме тому актуальний розмір цих даних зберігається в попередньому блоці.

Останнім, але не менш важливим параметром в пакеті є контрольна сума цього пакету. Саме завдяки їй можна перевірити цілісність даних. При цьому для її обчислення використовуються тільки попередні блоки.

Для отримання контрольної суми використовується алгоритм циклічного надлишкового коду CRC-16-IBM з основою поліному 0x8005.

4.1.2. Реалізовані операції

Розробка клієнтського модулю розпочинається з перевизначення операцій, які надає модуль FUSE. При цьому, ми описуємо лише ті функції, які необхідні конкретно для нашої розподіленої файлової системи.

Для написання даного клієнтського модулю, були обрані наступні операції:

- getattr – для отримання атрибутів файлу;
- mkdir – для створення папки;
- readdir – для відображення вмісту папки;
- open – для відкриття файлу;
- create – для створення і відкриття файлу;
- write – для запису інформації в файл;

- read – для зчитування інформації з файлу;
- flush/release – для закриття файлу;

Даних операцій достатньо для того, щоб реалізувати розподілену файлову систему.

Інтерфейс, який необхідний для роботи з цими операціями наведений в табл. 5. Параметр «path» – це шлях до файлу або папки, «port» – це TCP порт, на який буде приходити відповідь, «type» – це тип файлу (в даній розподіленій файловій системі існують два типи файлів: звичайний файл, який позначається літерою «f» та директорія, відповідно, «d»), «mode» – це параметр, який вказує на права доступу до файлу або папки (в даній реалізації він має значення 0777, що свідчить про повний доступ до файлу), «size» – це розмір файлу або папки в байтах, «descriptor» або файловий дескриптор, який ідентифікує файл в системі.

Таблиця 5

API передачі даних по TCP

Операція	Запит	Відповідь
getattr	getattr&path&port	1&type&mode&size
mkdir	mkdir&path&port	1
readdir	readdir&path&port	1&folder&…&file
open	open&path&port	1&descriptor
create	create&path&port	1&descriptor
flush/release	close&path&port	1
write	write&fd&size&offset	
read	read&fd&size&offset&tcp&udp	1&size

Операції запису та зчитування даних файлу поєднує у собі використання двох протоколів, TCP та UDP.

Операція запису спочатку відправляє TCP запит з командою «write», дескриптором файлу, у який записуються дані, розміром записуваних даних та відступом від початку файлу. При чому розмір та відступ одразу переводяться з байтів в кількість пакетів. Таким чином, якщо розмір файлу 2048 байти, то на сервер відправляється значення розміру, що дорівнює 4, при умові, що максимальний розмір даних в пакеті 512 байт, та, відповідно, якщо відступ нуль передається просто 0.

Після цього запиту одразу, без TCP відповіді від сервера взаємодії, відправляються і самі дані по UDP. Даний процес описується як протокол передачі даних (див. Додаток 1).

Спочатку буфер, переданий в функцію «write», при записі в файл, розбивається на блоки рівної довжини по 512 байт. З кожного блоку формується пакет. Потім пакети додаються в чергу на відправлення. Після додавання всіх пакетів, функція «write» вважається виконаною. Далі ці пакети забираються з черги окремим потоком (в даній реалізації можна виділити більше одного потоку, але достатньо і одного), який займається лише відправкою пакетів на сервер.

Черга, у яку додають пакети, зберігає інформацію про адресу одержувача, а саме ip-адрес і UDP порт сервера, на який необхідно відправити дані, та сам пакет з даними. Це необхідно для сумісності зі схемою передачі даних в підсистемі управління даними, а також наявність адреси дозволить розширити функціональні можливості системи у майбутньому. В даній реалізації клієнтського модулю ця адреса є незмінною адресою сервера взаємодії.

Відправка пакетів є дуже важливою частиною операції запису. Тому у разі помилки при роботі, операція повторюється TTR разів (в даній реалізацій TTR = 16). Якщо помилка викликана самою функцією запису даних у сокет, то ця функція теж повторюється TTR разів. Якщо запис у сокет пройшов успішно, то відправник чекає відповіді від сервера зі статусом прийому даних. Якщо дані прийшли на сервер взаємодії і

перевірка контрольної суми пакету підтвердила цілісність даних, то клієнту повертається статус 1, в іншому ж випадку – 0, і операція повторюється знову. Коли кількість повторів закінчується і пакет не відправлений, то він повертається в кінець черги. Таким чином не затримуючи потік, відправник продовжує оброблювати наступні пакети в черзі, а даний пакет повертається до відправника через деякий час.

Обмеженням операції запису є те, що дані можна лише дозаписувати у файл. Реалізація редагування запланована на майбутнє подальше вдосконалення системи.

Операція читання з файлу спочатку відправляє TCP запит з командою «read», дескриптором файлу, з якого треба зчитати дані, розміром даних (за замовчуванням передається 4096 байт), відступом від початку файлу та TCP і UDP портами, на які буде передана відповідь. Аналогічно, до операції запису, розмір та відступ одразу переводяться з байтів в кількість пакетів.

У відповідь на TCP запит користувачу на TCP сокет повертається кількість пакетів, яка доступна для зчитування, але не більше тієї кількості, що запросив користувач.

Після чого починається зчитування даних з UDP сокета стільки разів скільки є доступних пакетів для зчитування. Далі перевіряється контрольна сума пакету та у разі невідповідності серверу повертається статус 0, а кількість зчитування збільшується на один. Коли всі пакети прийшли, починається їх розміщення у буфер відповіді у тому порядку, який зазначений у другому блоці пакету, що відповідає за його номер у послідовності.

4.2. Рекомендації щодо подальшого вдосконалення

Головним вдосконаленням клієнтського модулю є реалізація усіх операцій, що надає FUSE. Таким чином збільшиться варіативність можливостей при роботі з файловою системою, що дасть більшу гнучкість для її вдосконалення.

Одне з важливих рішень є надання доступу до даних декільком користувачам. Це також дозволить додати велику кількість нових можливостей.

Спільний доступ може бути як до усієї файлової системи, так і до окремних папок чи файлів. У такому випадку користувачі файлової системи поділяються на ролі, вони можуть бути або власниками, або гостями.

Власник своєї файлової системи матиме можливість створювати та налаштовувати доступ групам користувачів і, відповідно, права доступу до окремих файлів та папок для цієї групи, наприклад, тільки для читання або запису.

Для цього необхідно надати користувачу можливість зареєструватися та авторизуватися в даній розподіленій файловій системі. Це дозволить ідентифікувати користувача та створити список доступу до даних для власника файлової системи.

Ще одним варіантом є надання доступу до файлів та папок за посиланням. При цьому можна вказати термін дії посилання для підвищення надійності системи або додати перевірку на конкретного користувача, щоб тільки він мав право доступу до даних.

Також важливо дозволити створювати декілька розподілених файлових систем на одному комп’ютері. Це полегшить роботу зі своїми даними та доступу до даних файлової системи іншого користувача. При цьому не буде необхідно відключати одну файлову систему, щоб перейти на іншу.

Необхідно також подбати про безпеку даних, так як вона відіграє важливу роль у даній системі, особливо, беручи до уваги, те, що інформація пересилається через мережу інтернет. Тому для захисту даних можна використати криптографічний прокол, який забезпечить конфіденційність при передачі даних між клієнтом та сервером.

Також, можна дати користувачу можливість шифрувати дані файлу перед відправкою їх на сервер, а потім при зчитуванні розшифровувати. Це

не дозволить стороннім особам отримати оригінальний вміст файлу. В такому варіанті тільки користувач, який має ключ шифрування, може отримати інформацію з файлу. Цей ключ не зберігається на серверах розподіленої файлової системи, а тільки у самого клієнта.

Також, не менш важливим є оптимізація роботи клієнтського модулю, а саме, зменшення часу на відправку та прийом даних. Аналіз роботи поточної версії програми дасть можливість покращити її функціональну частину, що призведе до пришвидшення її роботи.

ВИСНОВКИ

Метою даного дипломного проекту було створення розподіленої файлової системи, яка вирішує проблему зберігання великої кількості інформації.

Розроблену систему можна поділити на чотири частини, що можуть знаходитись на різних комп'ютерах та взаємодіяти один з одним через комп'ютерну мережу.

Перша частина – це клієнт файлової системи, який перевизначає стандартні операції ядра та взаємодіє з розподіленою файловою системою по протоколам TCP та UDP.

Друга частина – це проміжний сервер взаємодії між клієнтом та кластером. Цей модуль представляє собою проксі-сервер, який слугує для розподілу навантаження та функціоналу кластер-менеджера.

Третій модуль – це сам кластер-менеджер, він відповідає за зберігання інформації на вузлах та взаємодіє з базою даних.

Четвертий модуль – це група серверів або вузли, на яких зберігаються дані файлів.

Для реалізації клієнтської частини була обрана мова програмування C, вона дає можливість доступу до апаратних ресурсів комп'ютера. А також програми, написані на цій мові, працюють значно швидше ніж на мовах програмування високого рівня.

Для написання клієнтського модулю, переглянувши декілька існуючих програмних рішень, було обрано використати модуль FUSE, який дозволяє користувачам без спеціальних прав та без модифікації ядра створювати власні файлові системи.

Як результат, реалізована клієнтська частина дозволяє створювати файли та папки, переглядати вміст директорій, а також записувати та зчитувати дані з файлів.

Можливими напрямками подальшої роботи щодо вдосконалення розробки є збільшення функціональних можливостей файлової системи, для

цього необхідно реалізувати усі операції, що надає інтерфейс FUSE. А також надати можливість спільного доступу до даних декільком користувачам. Не менш важливо приділити увагу надійності передачі даних та швидкодії роботи клієнтського модулю.

Розробку виконано згідно з вимогами Технічного завдання, тестування виконано у відповідності до затвердженої програми та методики тестування.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Сучасні розподілені файлові системи для Linux [Електронний ресурс]. – Режим доступу: http://easy-code.com.ua/2012/08/suchasni-rozpodileni-fajlovi-sistemi-dlya-linux-osnovni-vidomosti-linux-operacijni-sistemi-statti/. – Дата доступу: грудень 2018. – Назва з екрана.
2. Розподілена файлова система: опис, особливості, переваги [Електронний ресурс]. – Режим доступу: http://hi-news.pp.ua/internet/15207-rozpodlena-faylova-sistema-opis-osoblivost-perevagi.html. – Дата доступу: грудень 2018. – Назва з екрана.
3. Проблемы сетевых файловых систем [Електронний ресурс]. – Режим доступу: https://www.osp.ru/os/1999/03/179733/. – Дата доступу: грудень 2018. – Назва з екрана.
4. Ceph [Електронний ресурс]. – Режим доступу: https://ceph.com/. – Дата доступу: січень 2019. – Назва з екрана.
5. What is the Ceph File System (CephFS)? [Електронний ресурс]. – Режим доступу: https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/2/html/ceph_file_system_guide_technology_preview/what_is_the_ceph_file_system_cephfs. – Дата доступу: січень 2019. – Назва з екрана.
6. Архитектура и компоненты Ceph [Електронний ресурс]. – Режим доступу: http://onreader.mdl.ru/LearningCeph/content/Ch03.html. – Дата доступу: січень 2019. – Назва з екрана.
7. Ceph Filesystem [Електронний ресурс]. – Режим доступу: http://docs.ceph.com/docs/mimic/cephfs/. – Дата доступу: січень 2019. – Назва з екрана.
8. NFS - What is it and why should we care? [Електронний ресурс]. – Режим доступу: https://mapr.com/blog/nfs-what-it-and-why-should-we-care/. – Дата доступу: січень 2019. – Назва з екрана.

9. How to integrate Ceph with OpenStack [Електронний ресурс]. – Режим доступу: https://superuser.openstack.org/articles/ceph-as-storage-for-openstack/. – Дата доступу: січень 2019. – Назва з екрана.
10. Gluster is a free and open source software scalable network filesystem [Електронний ресурс]. – Режим доступу: https://www.gluster.org/. – Дата доступу: січень 2019. – Назва з екрана.
11. GlusterFS Documentation [Електронний ресурс]. – Режим доступу: https://docs.gluster.org/en/latest/. – Дата доступу: січень 2019. – Назва з екрана.
12. GlusterFS. Limitations [Електронний ресурс]. – Режим доступу: https://access.redhat.com/documentation/en-us/red_hat_gluster_storage/3.1/html/administration_guide/limitations4. – Дата доступу: січень 2019. – Назва з екрана.
13. Что такое Hadoop? [Електронний ресурс]. – Режим доступу: http://www.taskdata.com/index.php?id=26&Itemid=5&option=com_content&view=article&lang=ru. – Дата доступу: січень 2019. – Назва з екрана.
14. Apache Hadoop в Amazon EMR [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/ru/emr/features/hadoop/. – Дата доступу: січень 2019. – Назва з екрана.
15. HDFS Architecture [Електронний ресурс]. – Режим доступу: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. – Дата доступу: січень 2019. – Назва з екрана.
16. A Brief Summary of Apache Hadoop: A Solution of Big Data Problem and Hint comes from Google [Електронний ресурс]. – Режим доступу: https://towardsdatascience.com/a-brief-summary-of-apache-hadoop-a-solution-of-big-data-problem-and-hint-comes-from-google-95fd63b83623. – Дата доступу: січень 2019. – Назва з екрана.

17. HDFS Users Guide [Електронний ресурс]. – Режим доступу: https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html. – Дата доступу: січень 2019. – Назва з екрана.
18. Уайт, Т. Hadoop. Подробное руководство [Текст]. – 2-е. – СПб.: Питер, 2013. – 672 с. – 1000 экз. – ISBN 978-5-496-00662-0.
19. Лэм, Ч. Hadoop в действии [Текст]. – ДМК Пресс, 2012. – 424 с. – 500 экз. – ISBN 978-5-97060-156-3, 978-5-94074-785-7.
20. Hadoop, a Free Software Program, Finds Uses Beyond Search [Електронний ресурс]. – Режим доступу: https://www.nytimes.com/2009/03/17/technology/business-computing/17cloud.html. – Дата доступу: січень 2019. – Назва з екрана.
21. Cloudera floats commercial Hadoop distro [Електронний ресурс]. – Режим доступу: https://www.theregister.co.uk/2009/03/16/cloudera_hadoop_launch/. – Дата доступу: січень 2019. – Назва з екрана.
22. How Yahoo Spawned Hadoop, the Future of Big Data [Електронний ресурс]. – Режим доступу: https://www.wired.com/2011/10/how-yahoo-spawned-hadoop/. – Дата доступу: січень 2019. – Назва з екрана.
23. Apache Hadoop. The Scalability Update [Електронний ресурс]. – Режим доступу: https://www.usenix.org/system/files/login/articles/105470-Shvachko.pdf. – Дата доступу: січень 2019. – Назва з екрана.
24. Linux. системное программирование. 2-е изд [Електронний ресурс]. – Режим доступу: https://itsecforu.ru/wp-content/uploads/2018/01/lav_r_linux_sistemnoe_programmirovanie.pdf. – Дата доступу: січень 2019. – Назва з екрана.
25. Керниган, Б. Язык программирования Си [Текст] / Д. Ритчи. – 2-е изд. – М.: Вильямс, 2007. – С. 304. – ISBN 0-13-110362-8.
26. Гукин, Д. Язык программирования Си для «чайников» [Текст]. – М.: Диалектика, 2006. – С. 352. – ISBN 0-7645-7068-4.

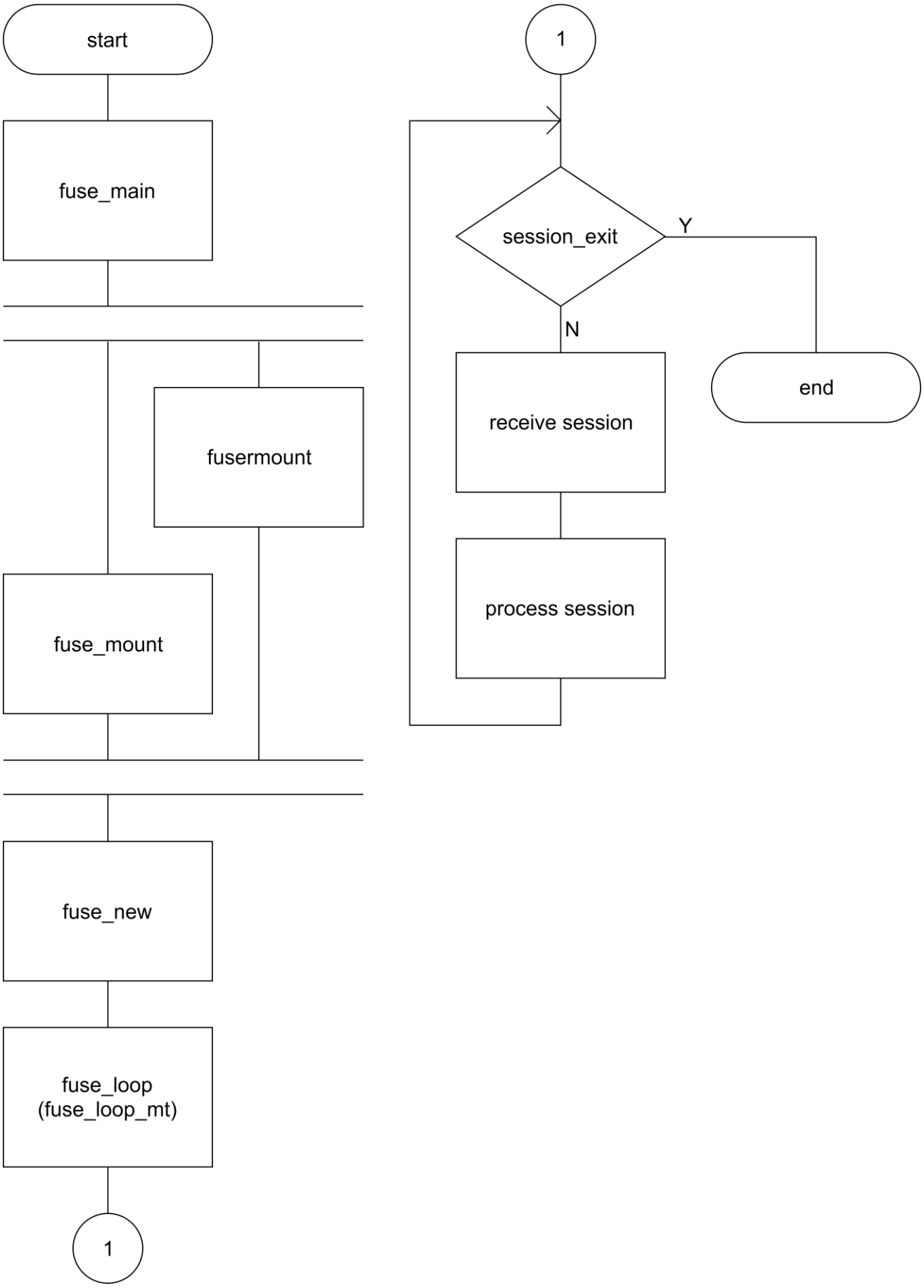
27. Подбельский, В.В. Курс программирования на языке Си: учебник [Текст] / С.С. Фомин. – М.: ДМК Пресс, 2012. – 318 с. – ISBN 978-5-94074-449-8.
28. Прата, С. Язык программирования С. Лекции и упражнения [Текст]. – М.: Вильямс, 2006. – С. 960. – ISBN 5-8459-0986-4.
29. Прата, С. Язык программирования С (С11). Лекции и упражнения, 6-е издание [Текст]. – М.: Вильямс, 2015. – 928 с. – ISBN 978-5-8459-1950-2.
30. Столяров, А.В. Язык Си и начальное обучение программированию [Текст] / Сборник статей молодых учёных факультета ВМК МГУ. – Издательский отдел факультета ВМК МГУ, 2010. – № 7. – С. 78-90.
31. Шилдт, Г. С: полное руководство, классическое издание [Текст]. – М.: Вильямс, 2010. – С. 704. – ISBN 978-5-8459-1709-6.
32. Фьюэр, А. Языки программирования Ада, Си, Паскаль [Текст] / Н. Джехани. – М.: Радио и Связь, 1989. – 368 с. – 50 000 экз. – ISBN 5-256-00309-7.
33. Разработка собственной файловой системы с помощью FUSE [Електронний ресурс]. – Режим доступу: https://www.ibm.com/developerworks/ru/library/l-fuse/index.html. – Дата доступу: січень 2019. – Назва з екрана.
34. Libfuse API documentation [Електронний ресурс]. – Режим доступу: http://libfuse.github.io/doxygen/. – Дата доступу: травень 2019. – Назва з екрана.
35. Libfuse [Електронний ресурс]. – Режим доступу: https://github.com/libfuse/libfuse. – Дата доступу: травень 2019. – Назва з екрана.
36. How Fuse Works [Електронний ресурс]. – Режим доступу: https://github.com/osxfuse/fuse/blob/master/doc/how-fuse-works. – Дата доступу: травень 2019. – Назва з екрана.

37. Libfuse: fuse_operations Struct Reference [Електронний ресурс]. – Режим доступу: https://libfuse.github.io/doxygen/struct fuse__operations.html. – Дата доступу: травень 2019. – Назва з екрана.
38. Libfuse: fuse_file_info Struct Reference [Електронний ресурс]. – Режим доступу: https://libfuse.github.io/doxygen/structfuse__file__info.html. – Дата доступу: травень 2019. – Назва з екрана.
39. Introduction to the Network File System (NFS) on Linux [Електронний ресурс]. – Режим доступу: http://www.linuceum.com/Server/ srvNFSIntro.php. – Дата доступу: травень 2019. – Назва з екрана.
40. Introducing Samba [Електронний ресурс]. – Режим доступу: https://www.linuxjournal.com/article/2716. – Дата доступу: травень 2019. – Назва з екрана.
41. Introduction to WebDAV [Електронний ресурс]. – Режим доступу: https://espace.cern.ch/winservices-help/DFS/WebDAV/Pages/WebDAV Introduction.aspx. – Дата доступу: травень 2019. – Назва з екрана.

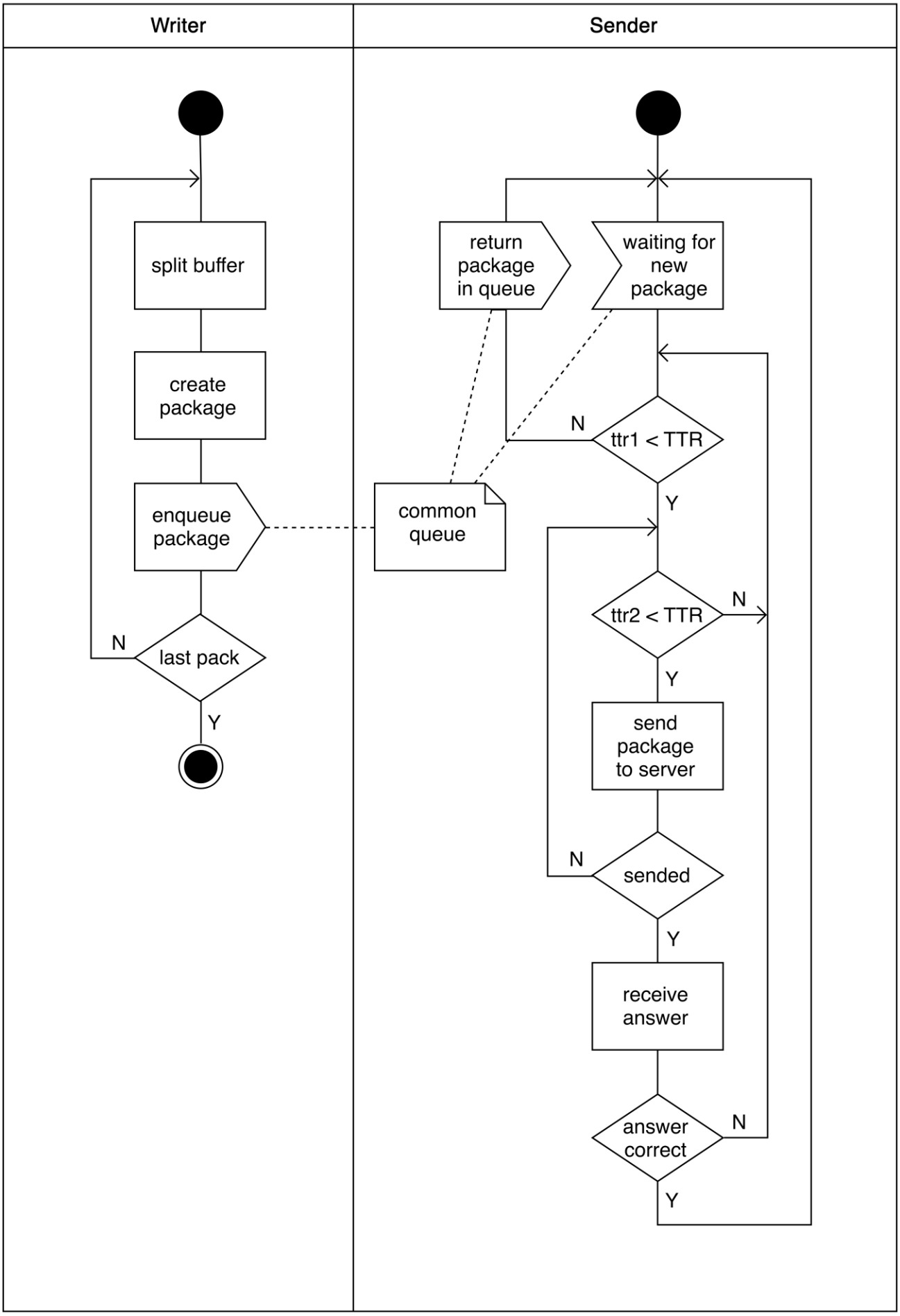
ДОДАТКИ

Додаток 1

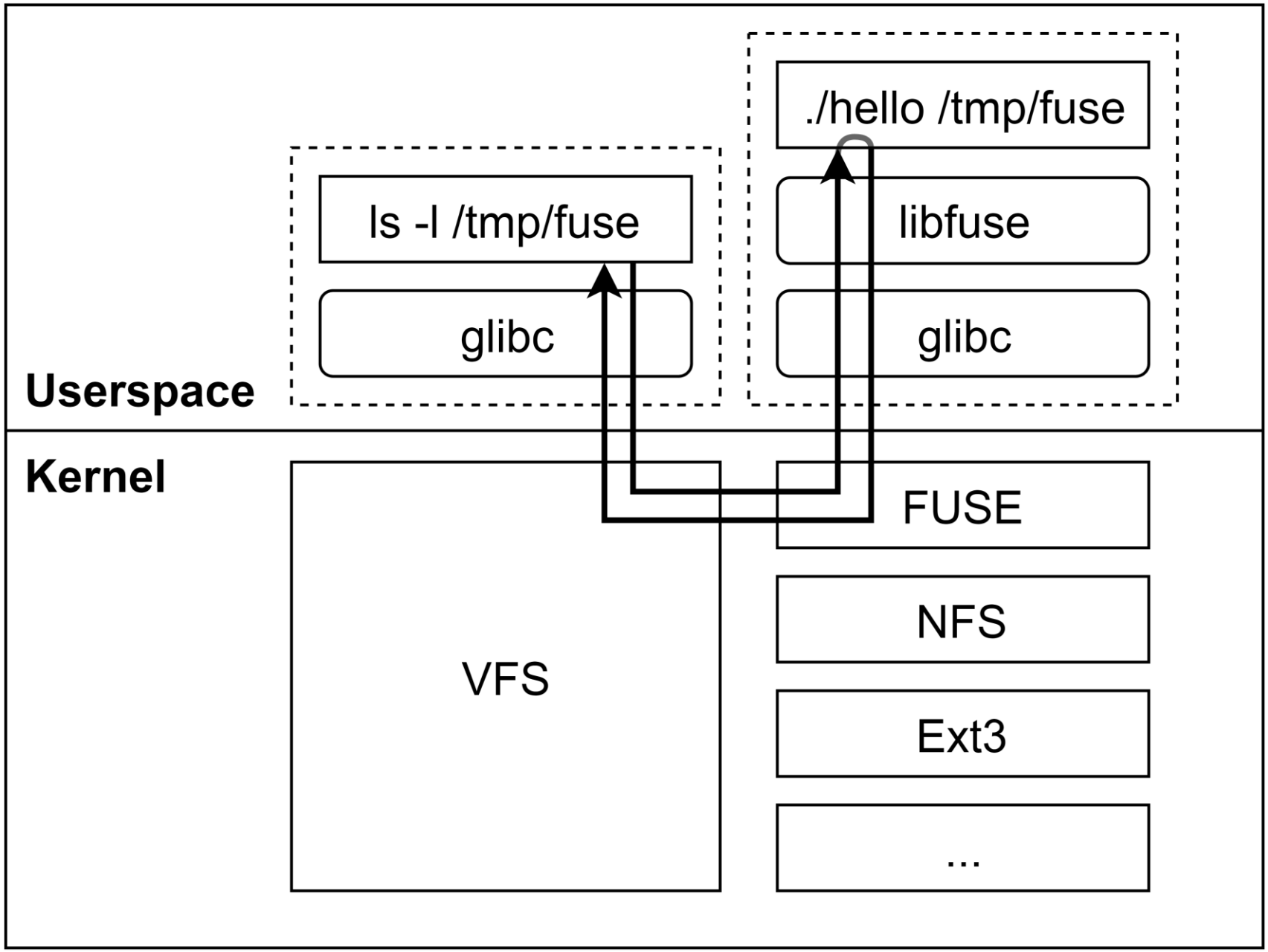
Копії графічних матеріалів

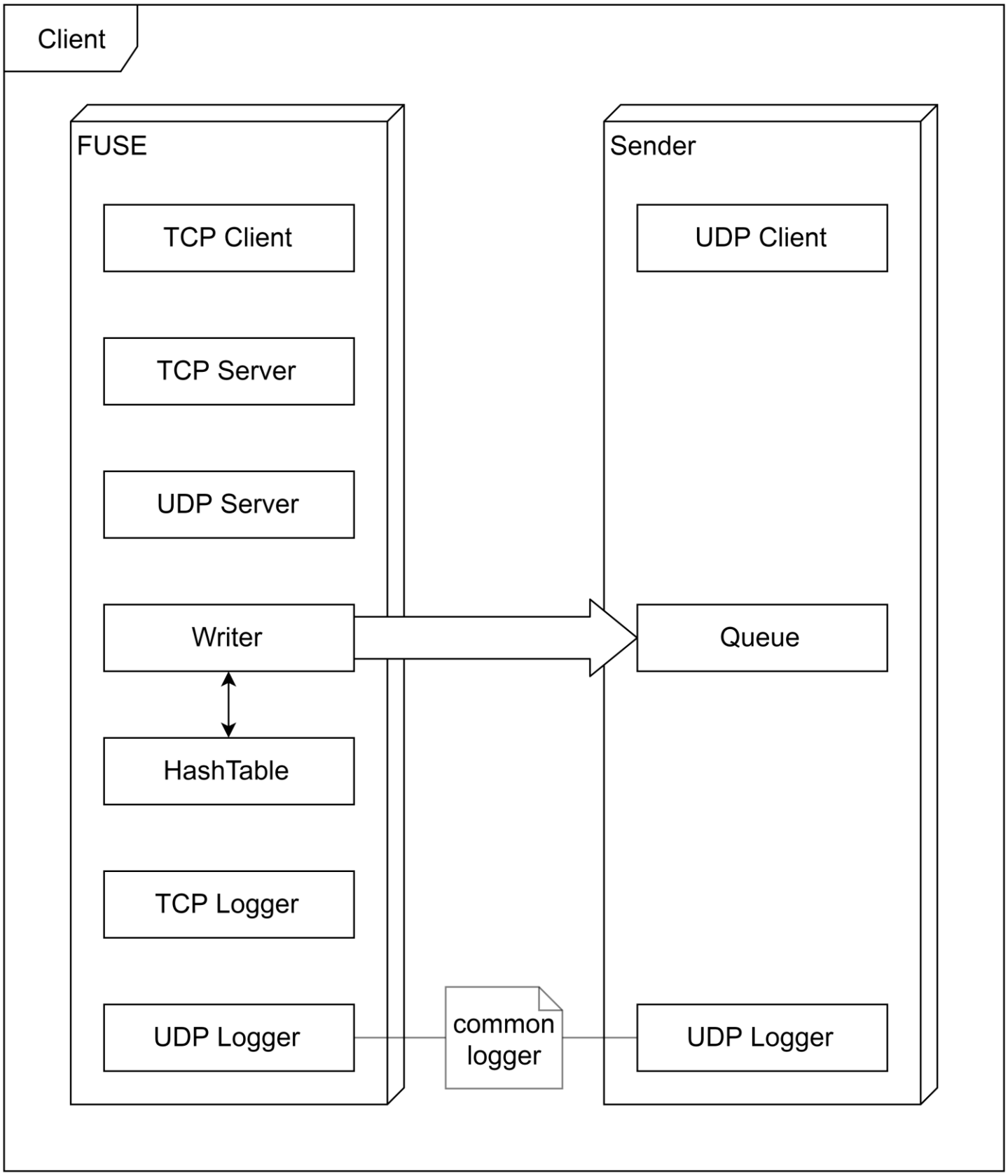


ДП.045200-06-99
Розподілена файлова система.
Підсистема користувацького інтерфейсу.
Алгоритм роботи бібліотеки FUSE. Схема алгоритму



ДП.045200-07-99
Розподілена файлова система.
Підсистема користувацького інтерфейсу.
Протокол передачі даних. Діаграма діяльності


./hello /tmp/fuse
ls -l /tmp/fuse
libfuse
glibc
glibc
Userspace
Kernel
FUSE
VFS
NFS
Ext3
...



Лобов Віталій Михайлович, група КП-51

Додаток 2

Лістинги програм

Програма-обробник. Реалізація операцій, які надає інтерфейс FUSE

```
#define FUSE_USE_VERSION 26

#include <fuse.h>
#include <string.h>
#include <errno.h>
#include <zconf.h>

#include "utils.h"

#define success 1
#define failure 0

#define tcp_port "7777"

#define DATA_SIZE 512

logger_t* logger;
socket_t* client;
socket_t* server;
sender_t* sender;

socket_t* udp_server;
logger_t* udp_logger;

hashtable_t* hashtable;

address_t* middleware_address;

char* new(size_t size) {
    return calloc(size, sizeof(char));
}

int ceil(float num) {
    int inum = (int)num;
    if (num == (float)inum) {
        return inum;
    }
    return inum + 1;
}

void repack(void* package, uint32_t fd, uint32_t number, uint32_t size,
void* data) {
    size_t offset = 0;
    memcpy(package + offset, &fd, sizeof(fd));

    offset += sizeof(fd);
    memcpy(package + offset, &number, sizeof(number));

    offset += sizeof(number);
    memcpy(package + offset, &size, sizeof(size));

    offset += sizeof(size);
    memcpy(package + offset, data, size);

    offset = DATA_SIZE + 14 - sizeof(uint16_t);
    uint16_t checksum = hash_function(package, offset);
    memcpy(package + offset, &checksum, sizeof(checksum));
}

void transmit(uint32_t fd, void* data, uint32_t size) {
    void* package = malloc(DATA_SIZE + 14);
```

```
    hashtable_node_t* node = hashtable_get(hashtable, &fd, sizeof(fd));

    uint32_t length = 0;
    uint32_t offset = 0;
    uint32_t number = (uint32_t) (node != NULL ? node->value : 0);

    do {
        if (size > DATA_SIZE) {
            length = DATA_SIZE;
        } else {
            length = size;
        }

        memset(package, 0, DATA_SIZE + 14);
        repack(package, fd, number, length, data + offset);

        sender_enqueue(sender, "127.0.0.1", 4235, package, DATA_SIZE + 14);

        offset += length;
        size -= length;
        number++;

    } while (size > 0);

    hashtable_set(hashtable, &fd, sizeof(fd), number);

    free(package);
}

void client_send(char* request) {
    size_t size = strlen(request) + 1 + strlen(tcp_port);
    char* buffer = new(4 + size + 1); //+1
    memcpy(buffer, &size, 4);

    sprintf(buffer + 4, "%s&%s", request, tcp_port);

    if (logger == NULL) {
        logger_printf(20, "fs: logger == NULL");
    } else {
        logger_printf(20, "fs: logger != NULL");
    }

    client = socket_init_tcp(logger);
    socket_connect(client, middleware_address);
    socket_write(client, buffer, 4 + size);
    socket_free(client);
    free(buffer);
}

int check_response(char* response) {
    char* status = strtok(response, "&");

    if (status != NULL) {
        if (strcmp(status, "1") == 0) {
            return success;
        }

        char* message = strtok(NULL, "&");

        while (message != NULL) {
            logger_print(logger, message, WARNING);
            message = strtok(NULL, "&");
        }
    }
```

```
    return failure;
}

int server_recv(char* response, size_t size) {
    char* buffer = new(4 + size + 2 + 1); //+1

    socket_t* conn = socket_accept(server);
    socket_read(conn, buffer, 4 + size + 2 + 1); //+1

    int status = check_response(buffer + 4);

    logger_printf(100, "fs: %d - %d", strlen(response), strlen(buffer));

    strcpy(response, buffer + 6);

    logger_printf(4096, "fs: res: %s; buff: %s", response, buffer);

    socket_free(conn);
    free(buffer);

    return status;
}

// operations //

static int fs_getattr(const char* path, struct stat* stbuf) {
    logger_printf(20, "fs: 1");
    memset(stbuf, 0, sizeof(struct stat));
    logger_printf(20, "fs: 2");
    if (strcmp(path, "/") == 0) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
        return 0;
    }

    if (strcmp(path, "/.Trash") == 0) {
        return -ENOENT;
    }

    if (strcmp(path, "/.Trash-1000") == 0) {
        return -ENOENT;
    }

    if (strcmp(path, "/.hidden") == 0) {
        return -ENOENT;
    }

    char* command = "getattr";
    size_t request_size = strlen(command) + 1 + strlen(path);
    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%s", command, path);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    char* response = new(1024 + 1); //+1
    int status = server_recv(response, 1024); //+1

    logger_printf(2048, "fs: %s (response): %s", command, response);

    if (status == failure) {
        free(response);
        return -ENOENT;
```

```
    }

    char* type = strtok(response, "&");
    uint mode = (uint) strtoull(strtok(NULL, "&"), NULL, 0);
    uint64_t size = strtoull(strtok(NULL, "&"), NULL, 0);

    if (strcmp(type, "f") == 0) {
        stbuf->st_mode = S_IFREG | mode;
        stbuf->st_nlink = 1;
        stbuf->st_size = size;

        free(response);
        return 0;
    }

    if (strcmp(type, "d") == 0) {
        stbuf->st_mode = S_IFDIR | mode;
        stbuf->st_nlink = 2;

        free(response);
        return 0;
    }

    free(response);
    return -ENOENT;
}

static int fs_readdir(const char* path, void* buffer, fuse_fill_dir_t
filler, off_t offset, struct fuse_file_info* fi) {
    char* command = "readdir";

    size_t request_size = strlen(command) + 1 + strlen(path);

    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%s", command, path);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    char* response = new(2550 + 9 + 1); //+1 // 10 files of 255 characters

    int status = server_recv(response, 2550 + 9 + 1); //+1

    logger_printf(2750, "fs: %s (response): %s", command, response);

    if (status == failure) {
        free(response);
        return -ENOTDIR;
    }

    filler(buffer, ".", NULL, 0);
    filler(buffer, "..", NULL, 0);

    char* filename = strtok(response, "&");

    while (filename != NULL) {
        filler(buffer, filename, NULL, 0);
        filename = strtok(NULL, "&");
    }

    free(response);

    return 0;
}
```

```
static int fs_open(const char* path, struct fuse_file_info* fi) {
    char* command = "open";

    size_t request_size = strlen(command) + 1 + strlen(path);

    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%s", command, path);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    char* response = new(100 + 1); //+1

    int status = server_recv(response, 100 + 1); //+1

    logger_printf(2048, "fs: %s (response): %s", command, response);

    if (status == failure) {
        free(response);
        return -ENOENT;
    }

    char* descriptor = strtok(response, "&");
    char* number = strtok(NULL, "&");

    int32_t fd = (int32_t) atoi(descriptor);

    hashtable_set(hashtable, &fd, sizeof(fd), atoi(number));

    fi->fh = (uint64_t) fd;

    free(response);

    return 0;
}

static int fs_mkdir(const char* path, mode_t mode) {
    char* command = "mkdir";

    size_t request_size = strlen(command) + 1 + strlen(path);

    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%s", command, path);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    char* response = new(1024 + 1); //+1

    int status = server_recv(response, 1024 + 1); //+1

    logger_printf(2048, "fs: %s (response): %s", command, response);

    if (status == failure) {
        free(response);
        return -ENOTDIR;
    }

    free(response);

    return 0;
}
```

```
static int fs_create(const char* path, mode_t mode, struct fuse_file_info*
fi) {
    char* command = "create";

    size_t request_size = strlen(command) + 1 + strlen(path);

    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%s", command, path);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    char* response = new(100 + 1); //+1

    int status = server_recv(response, 100 + 1); //+1

    logger_printf(2048, "fs: %s (response): %s", command, response);

    if (status == failure) {
        free(response);
        return -ENOENT;
    }

    fi->fh = (uint64_t) atoi(response);

    free(response);

    return 0;
}

static int fs_release(const char* path, struct fuse_file_info* fi) {
    return 0;
}

static int fs_write(const char* path, const char* buffer, size_t size,
off_t offset, struct fuse_file_info* fi) {
    int fd = (int) fi->fh;

    if (fd == -1)
        return -ENOENT;

    float s = size;
    int count = ceil(s / DATA_SIZE);

    char* command = "write";

    size_t request_size = strlen(command) + 1 + 20 + 1 + 20;

    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%d&%d", command, fd, count);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    transmit((uint32_t) fd, buffer, size);

    return (int) size;
}

static int fs_read(const char* path, char* buffer, size_t size, off_t
offset, struct fuse_file_info* fi) {

    int fd = (int) fi->fh;
```

```
    if (fd == -1)
        return -ENOENT;

    char* command = "read";

    size_t request_size = strlen(command) + 1 + 20 + 1 + sizeof(size) + 1 +
sizeof(offset) + 4;

    char* request = new(request_size + 1); //+1
    sprintf(request, "%s&%d&%zu&%li&8888", command, fd, size, offset);
    client_send(request);
    logger_printf(2048, "fs: %s (request): %s", command, request);
    free(request);

    char* response = new(size + 1); //+1

    int status = server_recv(response, size + 1); //+1

    logger_printf(2048, "fs: %s (response): %s", command, response);

    if (status == failure) {
        free(response);
        return -ENOENT;
    }

    int count = atoi(response);

    free(response);

    int total = 0;


    void* array[count];
    int sizes[count];

    for (int i = 0; i < count; i++) {

        char* result = new(DATA_SIZE + 14 + 1);

        address_t addr;
        memset(&addr, 0, sizeof(addr));
        socklen_t addr_len = 0;

        socket_recvfrom(udp_server, result, DATA_SIZE + 14 + 1, &addr,
&addr_len);


        int number;
        memcpy(&number, result + 4, 4);
        int _size;
        memcpy(&_size, result + 8, 4);

        total += _size;

        sizes[number] = _size;
        array[number] = malloc(_size+1);
        memcpy(array[number], result+12, _size);

        free(result);

    }

    int offset_b = 0;
```

```
    for (int i = count - 1; i >= 0; i--) {
        memcpy(buffer + offset_b, array[i], sizes[i]);
        offset_b += sizes[i];
    }

    for (int i = 0; i < count; i++) {
        free(array[i]);
    }

    return total;
}

static int fs_truncate(const char* path, off_t size) {
    logger_printf(2048, "fs: fs_truncate %s; %d", path, size);
    return 0;
}

static int fs_getxattr(const char* path, const char* name, char* value,
size_t size) {
    logger_printf(2048, "fs: fs_getxattr %s; %s; %s; %d", path, name,
value, size);
    return 0;
}

static struct fuse_operations fuse_example_operations = {
        .getattr = fs_getattr,
        .open = fs_open,
        .readdir = fs_readdir,
        .release = fs_release,
        .create = fs_create,
        .mkdir = fs_mkdir,
        .write = fs_write,
        .read = fs_read,
        .getxattr = fs_getxattr,
        .truncate = fs_truncate,
};

int main(int argc, char* argv[]) {

    logger = logger_create(NULL);

    hashtable = hashtable_init(150, 2);

    middleware_address = socket_addr_init(AF_INET, "0.0.0.0", 4234);

    server = socket_init_tcp(logger);
    socket_bind(server, socket_addr_init(AF_INET, "0.0.0.0", 7777));
    socket_listen(server, 10);

    udp_logger = logger_create("diplom2.log");

    udp_server = socket_init_udp(udp_logger);
    socket_bind(udp_server, socket_addr_init(AF_INET, "0.0.0.0", 8888));

    sender = sender_init(1, udp_logger);

    sender_start(sender);

    return fuse_main(argc, argv, &fuse_example_operations, NULL);
}
```

Logger. Для запису повідомлень та відстеження помилок

```
logger_t* logger_create(char* pathname) {
    logger_t* logger = malloc(sizeof(logger_t));

    if (pathname == NULL) {
        logger->fd = creat(DEFAULT_LOG, 0777);
    } else {
        logger->fd = creat(pathname, 0777);
    }

    return logger;
}

void logger_print(logger_t* logger, char* text, log_type type) {
    char* error = strerror(errno);

    char* log_text = NULL;
    size_t log_text_size = 0;
    char* log_type_text = NULL;
    time_t lt = time(NULL);
    char* time_text = asctime(localtime(&lt));

    switch (type) {
        case ERROR:
            log_type_text = "[ERROR] ";
            log_text_size = strlen(log_type_text) + strlen(time_text) +
strlen(text) + strlen(error) + 4 + 1; //+1
            log_text = calloc(log_text_size, sizeof(char));
            sprintf(log_text, "%s%s\t%s: %s\n", log_type_text, time_text,
text, error);
            break;
        case WARNING:
            log_type_text = "[WARNING] ";
            log_text_size = strlen(log_type_text) + strlen(time_text) +
strlen(text) + 2 + 1; //+1
            log_text = calloc(log_text_size, sizeof(char));
            sprintf(log_text, "%s%s\t%s\n", log_type_text, time_text,
text);
            break;
        case INFO:
            log_type_text = "[INFO] ";
            log_text_size = strlen(log_type_text) + strlen(time_text) +
strlen(text) + 2 + 1; //+1
            log_text = calloc(log_text_size, sizeof(char));
            sprintf(log_text, "%s%s\t%s\n", log_type_text, time_text,
text);
            break;
    }

    if (log_text != NULL) {
        write(logger->fd, log_text, log_text_size - 1);
    }

    free(log_text);
}

void logger_printf(size_t size, const char* format, ...) {
    char* buffer = new(size);

    va_list args;
    va_start(args, format);
    vsprintf(buffer, format, args);
    va_end(args);
```

```
    logger_print(logger, buffer, INFO);

    free(buffer);
}

void logger_free(logger_t* logger) {
    close(logger->fd);
    free(logger);
}
```

Додаток З

Копія презентації

Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Розподілена файлова система. Підсистема користувацького інтерфейсу

Студент: Лобов Віталій
Керівник: к.т.н., доцент Вунтесмері Ю.В.

Київ 2019

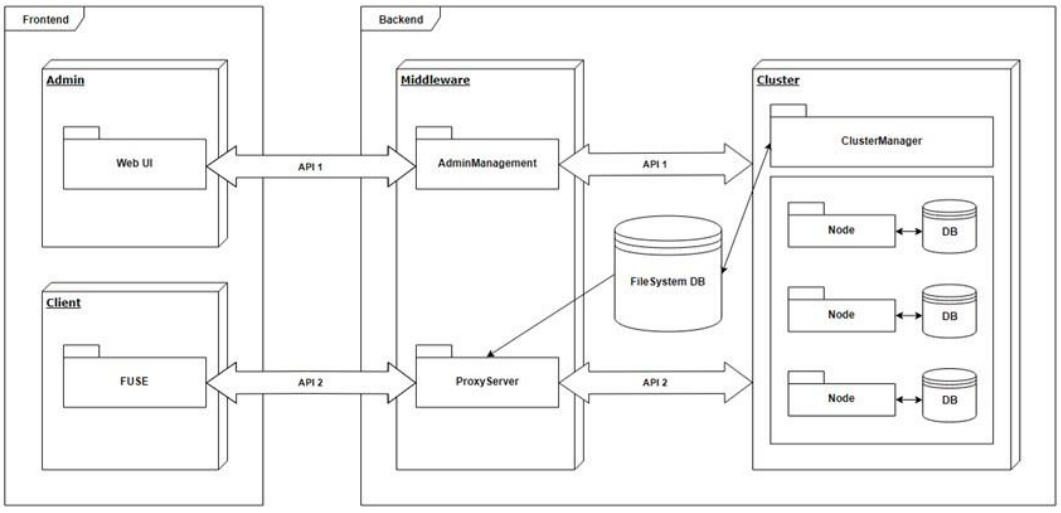
Постановка задачі

- Розроблення клієнтського модулю для розподіленої файлової системи
  - Перегляд вмісту директорій
  - Створення та відкриття файлів
  - Запис у файл
  - Читання з файлу
  - Закриття файлу

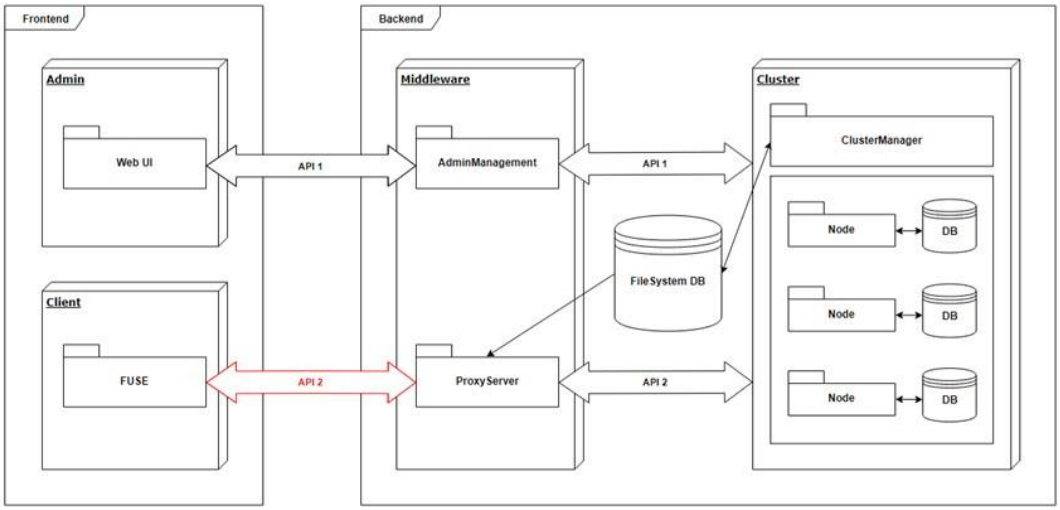
Аналіз існуючих рішень

- Ceph
  - недоліки: операція запису не є атомарною
- GlusterFS
  - недоліки: імена файлів не повинні містити підкреслення
- HDFS
  - недоліки: виконання завдання займає декілька секунд

Структура розподіленої файлової системи



Структура розподіленої файлової системи



API передачі даних по TCP

Запит

size	command&arg&....&arg&port
4 bytes	«size» bytes

Відповідь

size	status&arg&....&arg
4 bytes	«size» bytes

API передачі даних по TCP

Отримання атрибутів:

<u>запит</u>:
getattr&path&port

<u>відповідь</u>:
1&type&mode&size

Папка: 1&d&777&0
Файл: 1&f&777&10

API передачі даних по TCP

Створення файлу:

<u>запит</u>:
create&path&port

<u>відповідь</u>:
1&descriptor

API передачі даних по TCP

Відкриття файлу:

запит:
open&path&port

відповідь:
1&descriptor

API передачі даних по TCP

Створення папки:

запит:
mkdir&path&port

відповідь:
1

API передачі даних по TCP

Перегляд вмісту папки:

запит:
readdir&path&port

відповідь:
1&folder&…&file

API передачі даних по TCP

При помилках:

відповідь:
0&message

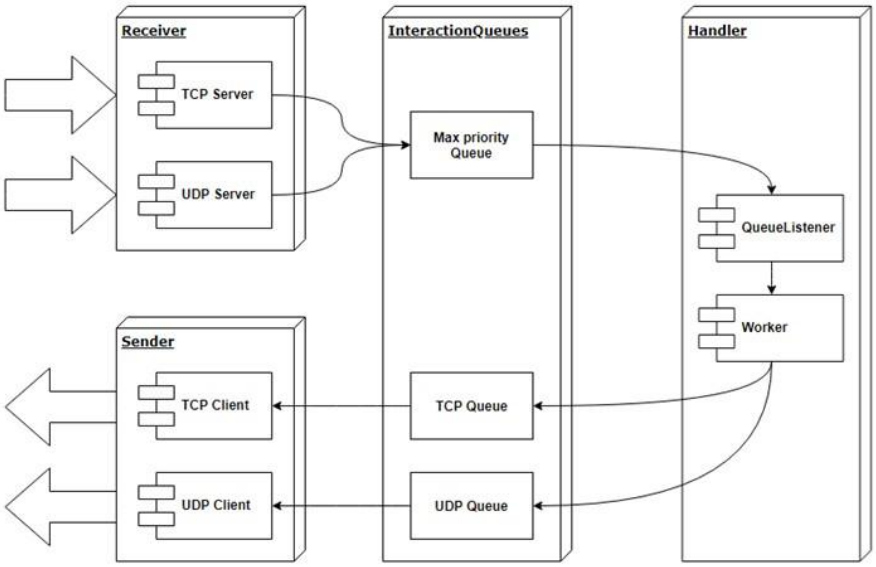
API передачі даних

Запис у файл?

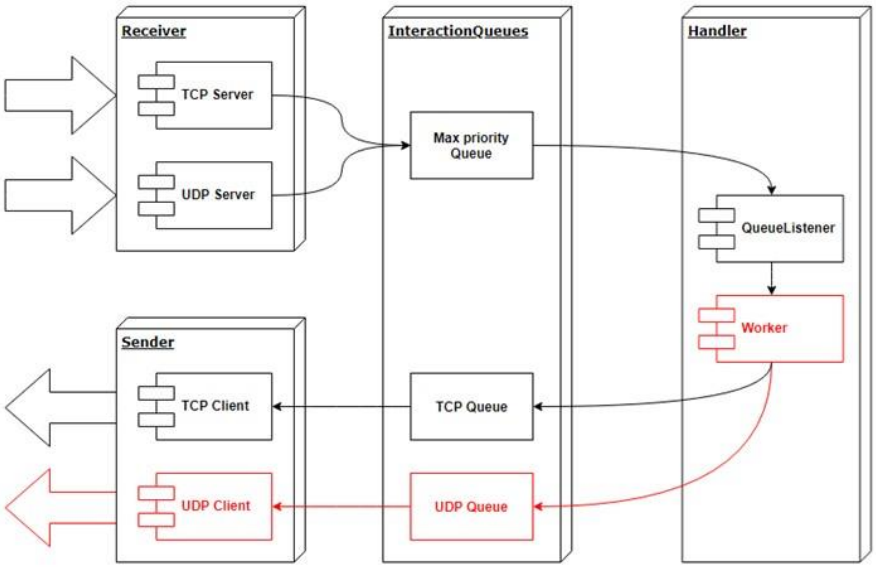
Читання з файлу?

TCP + UDP

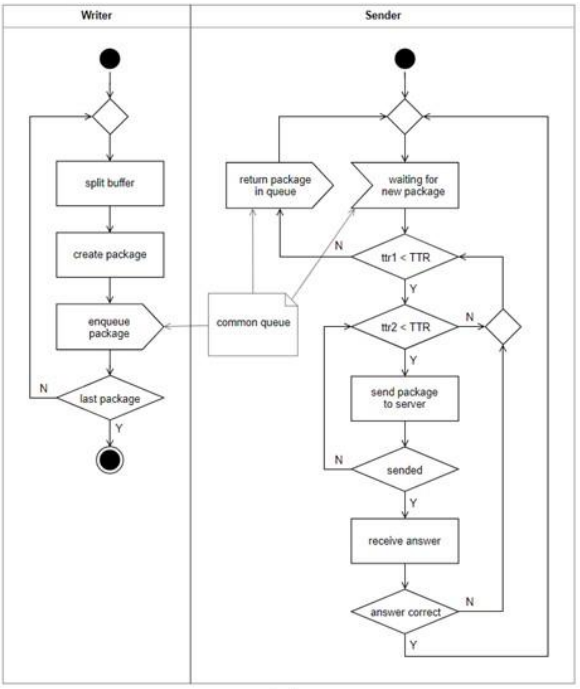
Структура передачі даних



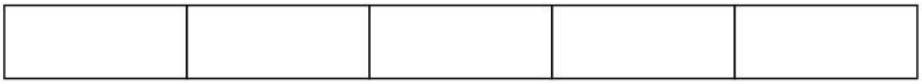
Структура передачі даних



Протокол передачі даних



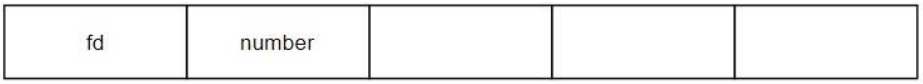
Структура пакету з даними. Вміст пакету



Структура пакету з даними. Файловий дескриптор

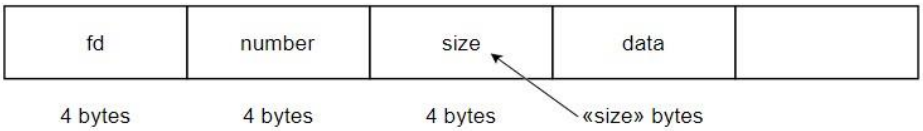
4 bytes

Структура пакету з даними. Номер пакету

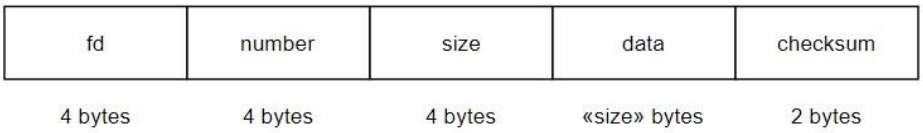


Структура пакету з даними. Розмір даних

Структура пакету з даними. Дані файлу



Структура пакету з даним. Контрольна сума



API передачі даних

Запис у файл:

TCP запит:
write&fd&size&offset

UDP запит:
package

API передачі даних

Читання з файлу:

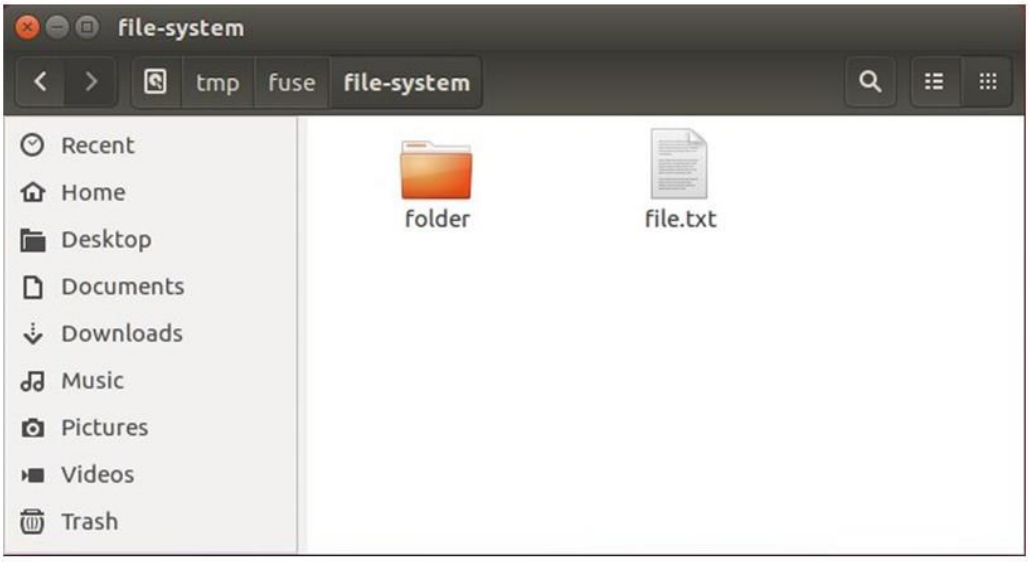
TCP запит:
read&fd&size&offset&tcp&udp

TCP відповідь:
1&size

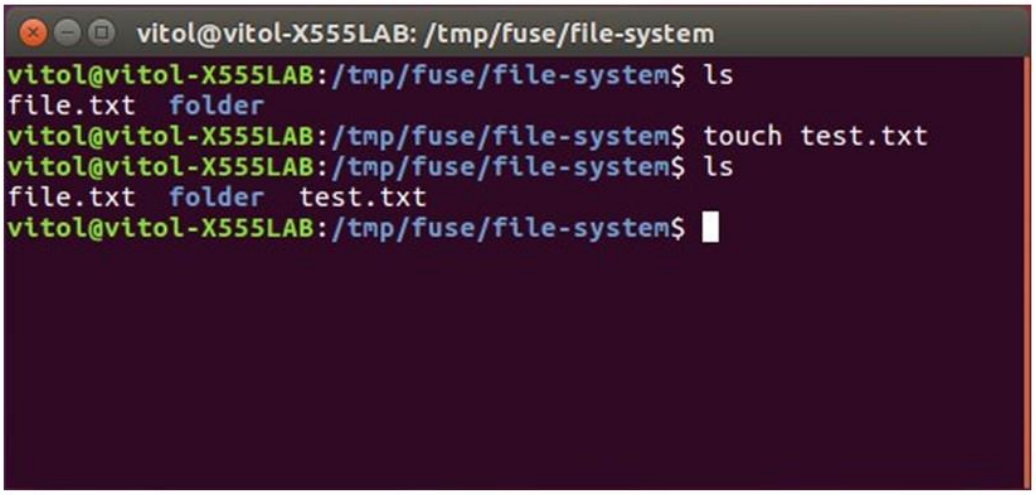
UDP відповідь:
package

Отримані результати

Взаємодія з файловою системою



Взаємодія з файловою системою



Можливі шляхи подальшого вдосконалення

- Реалізація усіх операцій файлової системи
- Надання спільного доступу до даних
- Надійність даних
- Швидкодія

Висновки

- Розроблено клієнтський модуль для розподіленої файлової системи, який дозволяє зберігати дані та отримувати доступ до них через комп’ютерну мережу

Дякую за увагу!

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ І.А. Дичка

«___» _____ 2018 р.

РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

Програма та методика тестування

ДП.045200-04-51

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.М. Лобов

2018

ЗМІСТ

1. ОБ’ЄКТ ВИПРОБУВАНЬ

Об’єктом випробувань є підсистема користувацького інтерфейсу розподіленої файлової системи, створена за допомогою модулю FUSE, на мові програмування C.

2. МЕТА ТЕСТУВАННЯ

Метою тестування є перевірка роботи системи користувацького інтерфейсу, а саме:

1) функціональна працездатність;
2) коректне створення пакету даних;
3) коректна робота протоколу передачі даних;
4) відповідність вимогам Технічного завдання.

3. МЕТОДИ ТЕСТУВАННЯ

Тестування виконується методом Black Box Testing. Перевіряється безпосередньо програмний продукт на відповідність функціональним вимогам.

Використовуються наступні методи:

1) функціональне тестування, зокрема на рівні Critical path test (базове тестування);
2) тестування продуктивності програмного забезпечення, зокрема Stability testing (тестування стабільності) та Load testing (навантажувальне тестування).

4. ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Тестування клієнтського модулю виконується засобом динамічного ручного тестування.

Працездатність перевіряється шляхом перевірки роботи усіх методів користувацького інтерфейсу:

1) тестування перегляду вмісту директорії;

2) тестування створення файлу;

3) тестування відкриття файлу;

4) тестування запису інформації в файл;

5) тестування зчитування інформації з файлу;

6) тестування закриття файлу.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ І.А. Дичка

«___» _____ 2019 р.

РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

Керівництво користувача

ДП.045200-05-34

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.М. Лобов

2019

ЗМІСТ

1. Опис структури користувацького інтерфейсу

Клієнтський модуль, призначений для роботи з розподіленою файловою системою, використовує FUSE – модуль для Unix-подібних операційних систем, який дозволяє користувачу без спеціальних прав та без модифікацій ядра створювати власні файлові системи. Це стає можливим завдяки тому, що драйвер файлової системи працює в просторі користувача, а модуль FUSE дає можливість перевизначити системні виклики ядра. Таким чином, замість стандартних операцій, виконуються ті операції, які ми реалізували.

2. Монтування файлової системи

Основна мета модулю FUSE полягає в тому, щоб вказати, як файлова система реагує на запити від користувача. Але він також використовується для монтування нової файлової системи. Під час монтування файлової системи програма-обробник (клієнтський модуль) реєструється ядром. Після чого, якщо користувач видає запит, наприклад, на читання або запис, для цієї нещодавно встановленої файлової системи, ядро пересилає ці запити обробникові, а потім відправляє відповідь назад користувачу.

Щоб змонтувати файлову систему, треба запустити скомпільований клієнтський модуль з параметром, який представляє собою шлях до директорії, у яку буде змонтована файлова система (попередньо створивши цю папку, якщо її не існує).

Також при монтуванні можуть бути використані наступні параметри:

1) default_permissions – для виконання власної перевірки прав дозволу;
2) allow_other – надає доступ до файлів усім користувачам, у тому числі і root-користувачу;
3) флаг «-f» – не запускає файлову систему у фоновому режимі, що дозволяє виводити програмні повідомлення в термінал користувача.

3. Опис можливостей файлової системи

Файлова система надає можливість користувачу зберігати та зчитувати дані файлу. Для цього клієнтський модуль перевизначає наступні операції:

1) операція перегляду вмісту директорії;

2) операція створення файлу;

3) операція відкриття файлу;

4) операція запису інформації в файл;

5) операція зчитування інформації з файлу;

6) операція закриття файлу.

Для виклику цих операцій, після монтування файлової системи, необхідно перейти в папку, у яку була змонтована ця файлова система, наприклад, з терміналу або файлового менеджеру, та виконати операції створення, відкриття, запису, перегляду файлу, які надаються цим терміналом або файловий менеджером.

4. Розмонтування файлової системи

Після завершення роботи з файловою системою її необхідно розмонтувати. Для цього використовується утиліта модулю FUSE «fusermount» з параметром «-u» або утиліта командного рядка в UNIX-подібних системах «umount» зі шляхом до директорії, у яку була змонтована файлова система.