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 управління даними»

Виконав:
студент IV курсу, групи КП-51,
Захарченко Віталій Ігорович _____

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, 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

Захарченку Віталію Ігоровичу

1. Тема проекту РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА УПРАВЛІННЯ ДАНИМИ, 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;
- описати схему передачі даних;
- спроектувати бази даних файлової системи;
- виконати програмну реалізацію серверної частини відповідно до вимог технічного завдання;
- провести тестування роботи програми.

5. Перелік обов'язкового ілюстративного матеріалу:
- схема роботи операції запису (креслення);
- схема роботи операції читання (креслення);
- структура файлової системи (плакат);
- схема передачі даних (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 системи управління даними	10.03.2019	
7.	Тестування серверної частини файлової системи	30.04.2019	
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 _____ Захарченко В.І.
(підпис)

Керівник проекту _____ Вунтесмері Ю.В.
(підпис)

АНОТАЦІЯ

В даному дипломному проекті описана розробка серверної частини розподіленої файлової системи.

У роботі описана проблематика даної системи, проаналізовано існуючі програмні рішення, обґрунтовано вибір технологій розробки та тестування системи. Також в дипломному проекті описано архітектуру та особливості реалізації серверної частини розподіленої файлової системи. Крім того доведена актуальність розроблюваної системи.

Розроблена частина розподіленої файлової системи надає функціонал для отримання та обробки команд, що надходять від користувача через комп'ютерну мережу, крім того система, за необхідністю, може формувати відповідь на конкретні запити користувача. Створено механізм розбиття даних на блоки однаково розміру з їх подальшим збереженням на різних, незалежних один від одного віддалених серверах. Даний механізм гарантує, що доступ до даних матиме тільки відповідний користувач, адже окремі блоки не несуть ніякої цінності без усіх даних.

Серверна частина розподіленої файлової системи представлена у вигляді трьох незалежних один від одного частин і серверної бази даних. Кожна з частин може знаходитись на різних фізичних пристроях і взаємодіяти з іншими через мережу інтернет. Через описані вище особливості багато уваги було приділено створенню механізмів швидкої передачі даних та коректності роботи усієї системи.

ABSTRACT

This diploma project describes the development of the server part of the distributed file system.

In this work the problems of this system are described, existing software solutions are analyzed, the choice of technologies of system development and testing is substantiated. Also, the diploma project describes the architecture and features of the implementation of the server part of the distributed file system. In addition, the relevance of the developed system is proved.

The developed part of the distributed file system provides a functional for receiving and processing commands coming from the user through the computer network, in addition, the system can, if necessary, form response to specific user queries. The mechanism of data splitting into blocks of the same size is created with their subsequent preservation on different, independent remote servers. This mechanism ensures that only the appropriate user will have access to the data, since individual blocks do not carry any value without all data.

The server part of the distributed file system is represented as three independent parts and a server database. Each of the parts can be on different physical devices and interact with others through the Internet. Due to the features described above, much attention has been devoted to the creation of fast data transfer mechanisms and the correct operation of the entire system.

ДП.045200-01-90 Розподілена файлова система. Підсистема управління даними. Відомість проекту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проекту		
ДП.045200-02-91	Розподілена файлова	4	
	система. Підсистема		
	управління даними.		
	Технічне завдання		
ДП.045200-03-81	Розподілена файлова	55	
	система. Підсистема		
	управління даними.		
	Пояснювальна записка		
ДП.045200-04-51	Розподілена файлова	4	
	система. Підсистема		
	управління даними.		
	Програма та методика		
	тестування		
ДП.045200-05-34	Розподілена файлова	4	
	система. Підсистема		
	управління даними.		
	Керівництво користувача		
ДП.045200-06-99	Розподілена файлова	1	
	система. Підсистема		
	управління даними.		
	Схема роботи операції		
	запису. Діаграма		
	послідовностей		

Позначення	Найменування	Кіл-ть	Примітка
ДП.045200-07-99	Розподілена файлова	1	
	система. Підсистема		
	управління даними.		
	Схема роботи операції		
	читання. Діаграма		
	послідовностей		
ДП.045200-08-98	Розподілена файлова	1	
	система. Підсистема		
	управління даними.		
	Компакт-диск		

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ І.А. Дичка

«___» _____ 2018 р.

РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА УПРАВЛІННЯ ДАНИМИ

Технічне завдання

ДП.045200-02-91

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.І. Захарченко

2018

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Розподілена файлова система. Підсистема управління даними.

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 надання клієнту функціональних можливостей для роботи з розподіленою файловою системою, а саме для модифікації структури файлової системи та фізичного збереження файлів на віддалених серверах зберігання даних.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Серверна частина повинна бути виконана на мові програмування Python, з використанням двох баз даних PostgreSQL та SQLite, 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:

- «Схема роботи операції запису. Діаграма послідовностей»;
- «Схема роботи операції читання. Діаграма послідовностей».

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 системи управління даними 10.03.2019

Тестування серверної частини файлової системи 30.04.2019

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 УПРАВЛІННЯ ДАНИМИ

Пояснювальна записка

ДП.045200-03-81

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.І. Захарченко

2019

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

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.

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.

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

NFS – протокол мережевого доступу до файлових систем.

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.

API – Application Programming Interface (прикладний програмний інтерфейс);

IP – протокол мережевого рівня для передавання датаграм між мережами.

ВСТУП

В дану історичну епоху людство стало свідком неймовірно швидкого і колосального росту кількості інформації (в середньому, загальна кількість інформації подвоюється щороку), це продукує проблеми зберігання та обробки цієї інформації. В даній роботі ми зосередимось на першій проблемі.

Проблема зберігання великої кількості інформації, особливо в маленьких компаніях або державних підприємствах чи структурах, є неймовірно поширеною, адже кількість використовуваної інформації щоденно збільшується при цьому засоби її зберігання модернізуються значно повільніше. На сьогоднішній день в Україні більшість компаній зберігають свої дані на персональних комп'ютерах, або на серверах (здебільшого ці сервери знаходяться у внутрішній мережі цієї компанії), також використовується варіант збереження даних на віддалених серверах, але він здебільшого поширений у великих комерційних підприємствах, адже вимагає великих затрат коштів, та певної кваліфікації персоналу. Якщо при зберіганні даних (в нашому випадку данні являють собою набір файлів різного розміру) на персональних комп'ютерах не виникає проблем з доступом до них (всі ми вміємо відкривати потрібні папки і вибирати потрібні файли), то при роботі з серверним збереженням даних, виникають проблеми доступу до цих файлів. Головна проблема полягає в тому, що без додаткових знань, або програм звичайний співробітник не зможе підключитися до сервера і дістати звідти потрібний йому звіт чи документ, також, на більш низькому рівні, проблемою є сам спосіб збереження даних і розширення доступної пам'яті для збереження (масштабування системи). Якщо проблема зручності доступу до даних є проблемою безпосередньо персоналу компанії, то проблема масштабування серверів та сховищ даних є серйозною технічною проблемою, неймовірно поширеною в наш час. На неї ми і звернемо свою увагу.

В наш час, більшість невеликих підприємств для збільшення загально доступної кількості пам’яті використовують найбільш простий спосіб, вони просто розширюють її шляхом додавання нової пам’яті або за допомогою придбання нових носіїв з більшою кількістю пам’яті. Цей шлях є доволі ефективний в разі помірного зростання кількості робочої інформації, або невеликої кількості загальної інформації на підприємстві. Однак в разі роботи з великими даними, або при збиранні та зберіганні статистики виникають проблеми з подальшим розширенням одного серверу чи комп’ютера, (це стає надто дорого), тому рішення потрібно шукати в іншому місці.

Проаналізувавши сучасні технологічні можливості і потреби сучасного IT-світу, найбільш очевидним і логічним варіантом є перенесення даних з фізичних носіїв інформації (жорстких дисків, серверів у внутрішній мережі) в мережеву хмару – скупчення великої кількості невеликих пристроїв, з’єднаних за допомогою мережі інтернет в одне ціле. Зазвичай такі скупчення з’єднаних між собою пристроїв (далі кластер), доволі незручні для використання звичайними користувачами, адже вимагають певних знань в цій області, саме тому використовують розподілені файлові системи, адже вони дають змогу максимально спростити взаємодію з даними, які знаходяться в кластерах. В сучасному світі існує величезна кількість файлових систем, які дозволяють користувачу створювати певні мережеві файли чи документи, однак, більшість цих систем надають доступ тільки до існуючих файлів на локальному комп’ютері чи сервері, що робить неможливим, або неймовірно важким, їх використання на кластерах, в яких файли зберігаються на окремих пристроях кластеру.

Більшість існуючих мережевих файлових систем не надають можливості обліку файлів, якщо файли розміщенні не на конкретній частині, а розділені на блоки і розподілені по різних складових кластеру (останнє необхідно для як найефективнішого використання пам’яті). Аналоги, які дозволяють вирішити вище названі задачі, такі як DFS, HDFS,

тощо, здебільшого є або обмежені певною системою, так DFS – розробка Microsoft і тому в ній є обмеження на систему в якій вона встановлена, або не реалізують всієї повноти функціоналу, яка необхідна для комфортної роботи з розподіленою файловою системою.

Підсумовуючи вище сказане, можна зробити висновок, що тема, якій присвячений даний проект, а саме – розробка розподіленої файлової системи і конкретно розробка підсистеми керування даними, яку можна буде використовувати на кластерах для зберігання інформації, незалежно від операційної системи пристрою (можливе обмеження на не комп’ютерні ОС), є актуальною і затребуваною в наш час.

1. АНАЛІЗ ПРОБЛЕМИ І ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

Робота з даними (файлами) великого розміру завжди пов’язана з безліччю проблем. Основними проблемами, з якими зустрічається кожен розробник або звичайний користувач, при використанні даних великого обсягу є:

- велика кількість пам’яті, що необхідна для збереження такої кількості інформації. Як наслідок, неможливість зберігання всієї інформації на одному пристрої чи на обмежені кількості пристроїв;
- ресурсозатратність при передачі даних, особливо мережею інтернет. В наслідок чого, доводиться годинами чекати завантаження великого файлу, особливо це помітно, якщо не користуватися програмами «паралельного» завантаження, наприклад торрентом;
- більш специфічна, але від цього не менш помітна проблема – проблема масштабування сховища даних. Вона пов’язана з тим, що в наш час інформації і даних настільки багато, що кожен користувач чи підприємство, час від часу зіштовхується з необхідністю розширення доступної фізичної пам’яті і чим далі відбувається розширення, тим більше витрат на це необхідно.

Для вирішення вищезазначених проблем використовують розподілені файлові системи.

Розподілена файлова система (мережева файлова система) – будь-яка файлова система, що дозволяє отримати доступ до файлів з декількох хостів через комп'ютерну мережу. Файли в таких системах, доступні по мережі, для програм і користувачів так само, як і файли на локальному диску.

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.

1.1. Огляд існуючих рішень

1.1.1. Ceph

Ceph – сховище об'єктів, що зберігає дані на розподіленому кластері та забезпечує інтерфейси різних рівнів детальності.

Особливості:

- розподіленим система;
- висока масштабованість системи;
- вільно розповсюджуваний проєкт.

Для реплікації та відмовостійкості даних в системі Ceph використовується звичайне обладнання, що не вимагає спеціальної підтримки. Це забезпечує мінімізацію часу адміністрування системи, крім того значно підвищуючи самовідновлюваність та самокерованість системи.

Ceph складається з 4 основних демон-серверів:

- монітори кластеру (ceph-mon) – відслідковують активні та зламані ноди кластеру;
- сервери метаданих (ceph-mds) – зберігають метадані inode та каталогів;
- пристрої зберігання об'єктів (ceph-osd) – зберігають вміст файлів;
- шлюзи передачі репрезентативного стану (RESTful) (ceph-rgw).

Користувачу доступна взаємодія з кожним елементом системи.

Для досягнення максимальної пропускної здатності Ceph розбиває файли користувача по багатьох нодах.

В Ceph вбудована підтримка розподіленого сховища даних. Для цього використовуються програмні бібліотеки, що забезпечують клієнтським програмам прямий доступ до автоматичного розподіленого сховища зберігання даних (RADOS).

Недоліком CephFS (файлова система) є відсутність стандартних засобів відновлення файлової системи, тому користувацька документація Ceph не рекомендує зберігати критичні дані через відсутність можливості аварійного відновлення та інструментів.

1.1.2. GlusterFS

GlusterFS – розподілена файлова система, що дозволяє організувати роботу сховища, розгорнутого поверх стандартних файлових систем POSIX. GlusterFS надає засоби автоматичного відновлення після збоїв і забезпечує практично необмежену масштабованість, завдяки відсутності прив'язки до централізованого серверу мета-даних.

Система складається з двох частин: серверної та клієнтської. На кожному сервері працює демон glusterfsd, який надає клієнтам доступ до локального сховища даних (у вигляді тому). Клієнтський процес glusterfs з'єднується з серверами за допомогою TCP і об'єднує всі доступні серверні томи в один. Отриманий том монтується на клієнтському хості.

Основна частина функціональності GlusterFS реалізована у вигляді трансляторів, налаштування яких дозволяє тонко налаштовувати поведінку всієї системи.

Транслятори використовуються для:

- синхронної реплікації між серверами;
- чергування порцій даних між серверами;
- розподілу файлів між серверами;
- балансування навантаження;
- перезавантаженню після аварійної відмови вузла;
- випереджаючого читання і запізнілого запису для збільшення швидкодії;
- дискових квот.

Реалізація серверів GlusterFS є неймовірно простою: вона надає в користування клієнтові лише своє сховище даних, невпливаючи на самі механізми зберігання даних. Завдяки описаній архітектурі кластери системи можуть розширюватись до кількох петабайт, використовуючи при цьому, звичайні комп’ютери середньої продуктивності. Особливістю GlusterFS є те, що для коректної роботи системі не потрібні окремі сервери метаданих. Це

значно підвищує масштабованість і надійність системи. Метадані зберігаються разом з даними (у розширених атрибутах файлів).

1.1.3. Hadoop

Apache Hadoop – вільна програмна платформа для організації розподіленого зберігання і обробки наборів великих даних з використанням моделі програмування MapReduce. Всі модулі в Hadoop спроектовані з оглядом на те, що апаратне забезпечення буде часто виходити з ладу, тому відмовостійкість та самовідновлення я наріжними каменями всієї системи.

Ядро системи Apache Hadoop складається з розподіленої файлової системи Hadoop Distributed Filesystem, та системи обчислень. Hadoop ділить файли на блоки , а потім розподіляє їх між всіма доступними вузлами кластера зберігання. Потім код обробки передається безпосередньо на вузли де й відбувається, власне, обробки даних. Описаний принцип обробки використовує локальність даних, тобто вузли взаємодіють лише з локально збереженими даними, не маючи жодної інформацію про решту даних. Такий підхід дозволяє значно пришвидшити продуктивність обробки даних.

Hadoop складається з наступних модулів:

- Hadoop Common – містить бібліотеки та утиліти, що необхідні для коректної роботи інших модулів Hadoop;
- Hadoop Distributed File System (HDFS) – файлова система (розподілена), в якій відбувається збереження користувацьких даних;
- Hadoop YARN – модуль що відповідає за керування обчислювальними ресурсами в кластерах;
- Hadoop MapReduce – реалізація моделі програмування MapReduce.

Розподілена файлова система Hadoop є розподілено, масштабованою, портативною файловою системою, написаною на мові Java.

Hadoop ділиться на HDFS і MapReduce. HDFS використовується для зберігання даних, а MapReduce – для їх обробки. HDFS має п’ять сервісів:

1. NameNode.

2. Secondary NameNode.

3. Job tracker.

4. Data Node.

5. Task Tracker.

NameNode (вузол імен): Головний вузол HDFS містить подробиці про кількість блоків, їх фізичне розташування на інших вузлах, інформацію про те, де зберігаються реплікації тощо. Має пряме з'єднання з клієнтом. HDFS може мати тільки один вузол цього типу.

Data Node (вузол даних, або робочий вузол): зберігає дані у вигляді блоків. Кожен Data Node надсилає повідомлення Heartbeat на NameNode що 3 секунди. Таким чином, коли NameNode не отримує heartbeatзапит з вузла даних протягом 2 хвилин, він сприймає цей вузол даних як мертвий і запускає процес реплікації блоку на іншому робочому вузлі.

Secondary NameNode: використовується лише для того, щоб відслідковувати контрольні точки метаданих файлової системи, які знаходяться у name node. Це допоміжний вузол для вузла імен.

Job Tracker: Використовується у процесі обробки даних. Job Tracker отримує від клієнта запити на Map Reduce.

Task Tracker (трекер завдань): це підлеглий вузол Job Tracker, виконує задачі, які йому надає останній.

Кластер Hadoop складається з одного вузла імен і кластеру зберігання даних. Кожен вузол даних обслуговує блоки даних через мережу інтернет за допомогою блокового протоколу. Файлова система використовує сокети TCP для зв'язку.

Файлова система HDFS включає в себе так званий вторинний NameNode, який регулярно з'єднується з первинним NameNode і будує знімки основної інформації каталогу NameNode, які система потім зберігає в локальних або віддалених каталогах. Ці контрольовані зображення можуть бути використані для перезапуску невдалого первинного NameNode без необхідності повторного відтворення всього журналу дій файлової системи,

з подальшим редагуванням журналу для створення найновішої структури каталогів. Оскільки NameNode є єдиною точкою для зберігання і управління метаданими, він може стати вузьким місцем при підтримці величезної кількості файлів, особливо великої кількості невеликих файлів. HDFS Federation, нове доповнення, має на меті вирішити цю проблему певною мірою, дозволяючи множині просторів імен обслуговуватися окремими NameNode-ми. Крім того, в HDFS є деякі проблеми, такі як проблеми з малими файлами, проблеми масштабованості, Single Point Failure (SPoF) і вузькі місця у великих запитах на метадані. HDFS був розроблений для в основному незмінних файлів і не підходить для систем, що вимагають одночасних операцій запису. Також проблемою виступає те, що дизайн системи вводить обмеження на портативність, що призводить до вузьких місць у роботі системи.

1.2. Розширена постановка задачі

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

Для цього мають бути вирішені наступні задачі:

- описати архітектуру розподіленої файлової системи. А саме логіку передачі даних, інтерфейс взаємодії між серверами та протокол обміну даними;
- спроектувати бази даних розподіленої файлової системи. Які включають основну базу даних, де зберігається інформація про місцезнаходження файлів, базу даних активних нод кластера та базу даних для зберігання інформації про розташування даних з файлу на ноді;
- розробити модуль збереження даних, у яких входить менеджер кластера та його ноди. А також взаємодію між ними та базами даних;

- розробити модуль комунікації. А саме проксі-сервер, який необхідний для взаємодії між клієнтом та кластером;
- написати клієнтський модуль. Що дозволить запустити код файлової системи в просторі користувача, що перевизначить системні виклики ядра для взаємодії з проксі-сервером.

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

- реалізувати підтримку для одного користувача файлової системи;
- надати можливість відкривати файл, записувати та зчитувати дані з файлу, переглядати вміст папок файлової системи;
- відслідковувати активні та зламані ноди кластера.

2. ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Вибір мов програмування

Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Python є мовою, що підтримує велику кількість парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована. Як і більшість інтерпретованих мов Python є кросплатформеною і не вимагає додаткових зусиль для перенесення з однієї платформи на іншу. Мова підтримує модулі та пакети модулів, що значно підвищує повторне використання коду і його модульність, що є корисним при розробці великих додатків. Також стандартний Python має неймовірно велику базу стандартних пакетів, які підходять для абсолютно різних галузей праці, що робить його зручним, при використанні в складних, багато цільових системах. Ще одним значним плюсом мови є те, що код є інтуїтивно зрозумілим і читабельним, що дозволяє швидко розроблювати великі додатки і розуміти сенс вже написаного коду.

Серед основних переваг Python можна назвати такі:

- чистий синтаксис;
- портабельність програм;
- стандартний дистрибутив має велику кількість корисних модулів;
- можливість використання Python в діалоговому режимі;
- стандартний дистрибутив має просте, але разом із тим досить потужне середовище розробки, яке зветься IDLE і яке написане на мові Python;
- зручний для розв'язання математичних проблем;
- відкритий код.

Python має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис Python, динамічна обробка типів, а також те, що це

інтерпретована мова, роблять її ідеальною для написання скриптів та швидкої розробки прикладних програм у багатьох галузях на більшості платформ.

Python дозволяє використовувати код скомпільований на мові C та C++, це значно розширює використання мови, крім того дозволяє переносити важкі обрахунки на більш швидку мову C. Python також зручна як мова розширення для прикладних програм, що потребують подальшого налагодження.

2.2. Вибір баз даних

В підсистемі керування даними бази даних використовуються для зберігання метаданих файлової системи і ноди. Так як ці дані мають наперед визначені, незмінну структуру, то, для спрощення і підвищення продуктивності системи, були обрані реляційні бази даних (SQL database). Такі бази містять в собі чітку структуру даних (таблиць, пакетів, тригерів, тощо) і дозволяють швидко і, що є головним, легко взаємодіяти з цими даними. Також реляційні бази даних є простішими у використанні ніж їхні нереляційні аналоги.

2.2.1. База даних файлової системи

Як основну базу даних (базу даних файлової системи), було обрано PostgreSQL.

PostgreSQL – написана на C об'єктно-реляційна система керування базами даних (СКБД). Так як СКБД є об'єктно-реляційною то в неї можна завантажувати користувацькі типи даних, що є корисним в нашому випадку, наприклад, для визначення типу файлу. Також великою перевагою PostgreSQL є розширений перелік вбудованих типів даних, деякі з яких не підтримуються стандартними реляційними системами керування базами даних. Так дуже зручним, для нас, типом даних є INET, який дозволяє зберігати ip адреси в форматах: IPv4 та IPv6. Крім того PostgreSQL надає можливість взаємодії з складними користувацькими структурами, які

можна легко завантажувати в і вивантажувати з бази даних. Крім того в даній СКБД доступні багатовимірні масиви.

Ще однією перевагою PostgreSQL є те, що її обмеження по розміру даних є неймовірно великими, що дозволяє не хвилюватись за це. До прикладу, в PostgreSQL взагалі немає обмеження по кількості рядків в таблицях, або індексах в них, тоді як інші СКБД такі обмеження мають.

PostgreSQL є клієнт-серверною СКБД, що є неймовірно важливим, для нас, адже різні частини нашої системи можуть знаходитись на різних фізичних пристроях і в різних місцях (буде описано нижче). Крім того така структура дозволить взаємодіяти з базою даних користувачам, які розташовані в різних місцях.

Також дана СКБД має відкритий код, а отже розповсюджується безкоштовно і не потребує додаткових ліцензій для її використання. Крім того це передбачає постійне вдосконалення та покращення вже існуючих систем СКБД, що є гарним підґрунтям для вдосконалення вже нашої системи керування даними. Ще одним плюсом є те, що до PostgreSQL є неймовірна кількість програм обгорток, написаних на різних мовах програмування, що дозволяє використовувати її у зв’язці з будь-якою мовою програмування.

І наостанок, в PostgreSQL містить доволі простий механізм реплікації даних, що значно підвищує комфорт її використання.

2.2.2. База даних вузла кластера (ноди)

Особливістю бази даних на ноді є те, що вона є невеликою за розмірами та не потребує клієнт-серверного з’єднання, адже використовується тільки одним вузлом і зберігає інформацію тільки про один вузол. Саме тому як базу даних для ноди було обрано SQLite.

SQLite – неймовірно популярна (використовується, майже, на усіх смартфонах), полегшена реляційна СКБД. Реалізована на мові С у вигляді бібліотеки. Сирцевий код SQLite розповсюджується як суспільне надбання, а отже може бути використаний у будь-якому проекті, що є неймовірно

важливо. Також, SQLite не використовує парадигму клієнт-сервер, а для передавання команд використовується API вбудоване вихідну бібліотеку. Це дозволяє значно пришвидшити взаємодію розроблюваної програми з базою даних, адже зникає затримка мережі і значно скорочується час відгуку бази даних на запит.

SQLite зберігає всі дані, налаштування, структури таблиць, взаємозв'язки між ними, тригери, тощо в одному файлі, який знаходиться на комп'ютері, на якому виконується користувацький застосунок. Однією з головних ідей SQLite є те, що вона дозволяє тільки один одночасний запис в базу даних, при цьому не обмежуючи читання з неї. Така схема ідеально підходить нам в рамках заданого проекту, адже одночасно до бази даних буде звертатись тільки один вузол кластеру.

Так як дана СКБД розповсюджується у вигляді бібліотеки то існує неймовірна кількість бібліотек обгорток, для роботи в різних операційних системах і на різних мовах програмування, це дозволяє не хвилюватись про платформні особливості комп'ютера, це дозволяє легко перенести розроблювальну систему з однієї платформи на іншу, не застосовуючи ніяких додаткових маніпуляцій.

3. АРХІТЕКТУРА ФАЙЛОВОЇ СИСТЕМИ

Структура даної розподіленої файлової системи складається з чотирьох частин: клієнтський модуль, сервер керування з'єднанням, кластер менеджер та сам кластер, на якому зберігаються дані з файлів. Також до цієї структури входять дві основні бази даних, які необхідні для роботи з розподіленою файловою системою. В даній частині будуть розглянуті всі складові системи без прив'язки до конкретної теми дипломної роботи, це необхідно для розуміння функціонування всієї системи і не може бути розділене на дві менші частини.

3.1. Клієнтський модуль

Файлова система – це частина програмного забезпечення, яка відповідає за роботу з даними, представлених у вигляді файлів і каталогів.

Для реалізації клієнтського модулю, використано інтерфейс FUSE – необхідний для створення файлових систем в просторі користувача. Таким чином відпадає необхідність переписувати код ядра, а все що необхідно – це лише перевизначити окремі операції, необхідні для даної файлової системи, які надає інтерфейс FUSE.

Проект FUSE складається з двох компонентів: модуля fuse kernel (зберігається в звичайних сховищах ядра) і бібліотеки простору користувачів libfuse. Бібліотека libfuse забезпечує еталонну реалізацію для зв'язку з модулем ядра FUSE.

Найважливіше, що потрібно знати при роботі з FUSE – це його API. Перш за все, для створення файлової системи необхідно оголосити змінну структури типу fuse_operations, де описуються зворотні виклики, і передати її в функцію fuse_main для початку роботи програми (рис. 1).

Після опису всіх необхідних функцій, залишається лише змонтувати розроблену файлову систему, запустивши клієнтський модуль з параметрами -f та директорією, у яку монтувати.

```
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 клієнтського модулю та списку зворотних викликів

3.2. Бази даних файлової системи

Як було сказано в одному з попередніх розділів для збереження метаданих в підсистемі використовується дві бази даних: база даних файлової системи (PostgreSQL) та база даних вузла кластера (SQLite).

3.2.1. База даних файлової системи

База даних файлової системи являє собою основну базу даних всього продукту. В цій базі зберігається вся інформація про стан файлової системи, а саме:

- інформація про файли і теки, що містяться в даній файловій системі, їх розмір, шлях, вміст, тощо (таблиця File);
- інформація про пакети (таблиця Package). Тут зберігається інформаціє місце фізичного знаходження пакету даних, його статус, порядок в файлі, тощо;
- інформація про ноди (таблиця Node). Використовується як метадані кластера. Зберігає інформацію про підключені (відомі) вузли кластера (ноди), їх ір та порти. Також тут зберігається ідентифікатор ноди в системі.

Данна база даних використовується в усіх операціях системи і є доступна усім елементам підсистеми керування даними, крім вузла кластера. Проте редагування бази доступно тільки менеджеру кластеру і відбувається виключно в ньому (в частині, що відповідає за файлову

систему), решта ж учасників процесу роботи підсистеми мають лише право на читання з бази даних. Такі обмеження накладені для інкапсуляції і максимального розподілення призначень елементів підсистеми.

Треба зазначити, що поняття файл в даній базі даних і в системі в цілому є широким і розповсюджується як на файли в звичному розумінні (регулярні файли) так і на теки.

Розглянемо структуру таблиць бази даних файлової системи. Перша таблиця це File, вона містить в собі інформацію, про структуру файлової системи і складається з наступних полів:

- file_id – ідентифікатор файлу в системі. Виступає первинним ключом запису (primary key) в таблиці. Має тип SERIAL, тобто є цілим без знаковим числом, яке автоматично інкрементується на одиницю при додавання нового запису в таблицю. Є унікальним для кожного запису в таблиці. Присвоюється автоматично, незмінний;
- pathname – повний шлях файлу в системі. Має формат STR, тобто є символьним рядком. Використовується, для ідентифікації файлу при його відкриті, редагуванні, тощо. Є унікальним в рамках таблиці;
- type – тип файлу. Має формат FILE_TYPES, що є користувацьким типом даних і являє з себе перелік (enum) двох можливих символьних значень: f – для регулярного файлу, d – для теки;
- data – вміст файлу. Являє з себе масив цілих чисел кожне з яких це ідентифікатор іншого файлу, або пакету. Це поле різниця для різних типів файлів, так якщо файл є регулярним, то в цьому полі зберігається ідентифікатор першого пакету в файлі. Якщо ж файл є папкою, то в даному полі зберігаються ідентифікатори всіх файлів;
- size – розмір файлу. Є цілим без знаковим числом. Також залежить від типу файлу, якщо це тека, то зберігається кількість файлів в ній, якщо це регулярний файл, то в даному полі зберігається інформація про розмір заданого файлу;

- order_num – зберігає кількість пакетів в файлі. Є без знаковим цілим числом, залежить від типу файлу. Для регулярних файлів, вказує кількість пакетів в файлі і визначає номер наступного, для теки завжди 0;
- status – визначає стан файлу. Має тип BOOL, тобто може приймати значень true, false (1 і 0). Використовується, для перевірки готовності файлу для зчитки і закриття.

Наступна за використання м та значенням таблиця – Package. Вона використовується для відслідковування фізичного розташування даних і їх взаємозв’язку між собою. Складається з наступних полів:

- pack_id – ідентифікатор пакету в системі. Є унікальним для кожного пакету. Виступає первинним ключом запису (primary key) в таблиці. Має тип SERIAL, тобто є цілим без знаковим числом, яке автоматично інкрементується на одиницю при додавання нового запису в таблицю. Присвоюється автоматично, незмінний;
- next_pack_id – ідентифікатор наступного файлу. Має тип даних INTEGER, тобто є цілим числом. Використовується для відслідковування зв’язків між пакетами даних. Слугує для встановлення зв’язку один до одного поміж пакетами;
- node_id – ідентифікатор вузла кластеру, на якому знаходиться цей пакет. Ціле число. Слугує для встановлення зв’язку багато до одного між пакетом та нодою (вузлом);
- file_id – ідентифікатор файлу, до якого цей пакет належить. Використовується для швидкого знаходження відповідного файлу. Слугує для встановлення зв’язку багато до одного між пакетом та нодою (вузлом);
- status – описує стан пакету. Має тип BOOL, тобто може приймати значень true, false (1 і 0). Використовується для визначення стану пакету, якщо стан true, то пакет записаний, інакше пакет знаходиться в стані запису.

Варто зазначити, що пакети в підсистемі керування даними зберігаються як односторонній зв’язний список, це забезпечує легке редагування файлу, адже для зміни послідовності пакетів достатньо модифікувати їх зв’язки.

Остання таблиця, що лишилась – Node. Потрібна для обліку кластера. Складається з наступних полів:

- node_id – ідентифікатор вузла кластера в системі. Є унікальним для кожного пакету. Виступає первинним ключом запису (primary key) в таблиці. Має тип SERIAL, тобто є цілим без знаковим числом, яке автоматично інкрементується на одиницю при додавання нового запису в таблицю;
- ip – адреса ноди в мережі інтернет. Має тип даних INET;
- tcp_port – порт для з’єднання за допомогою протоколу TCP. Ціле число. Унікальне в рамках однієї таблиці;
- udp_port – порт для з’єднання за допомогою протоколу UDP. Ціле число. Унікальне в рамках однієї таблиці.

На рис. 2 зображена uml діаграма бази даних файлової системи.

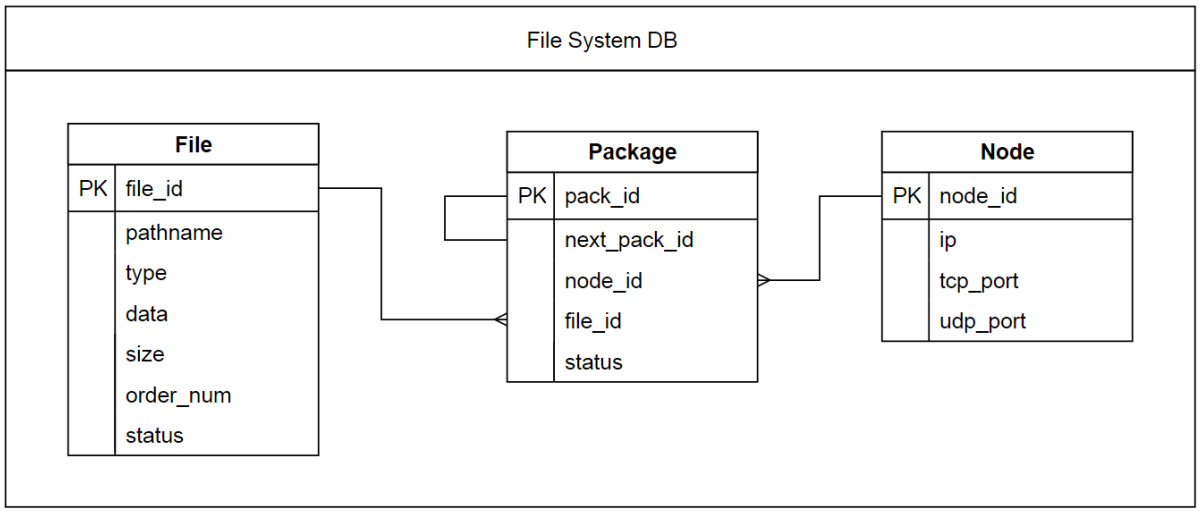


Рис. 2. Схема бази даних файлової системи.

3.2.2. База даних ноди

База даних ноди (SQLite) використовується для зберігання метаданих ноди та ведення обліку вільного простору в ноді. зауважимо, що пам'ять для зберігання у вузлі кластера побита на блоку такого самого розміру, що пакети даних, тобто ведення обліку вільного місця зводиться до введення обліку вільних блоків.

База даних ноди складається всього з однієї таблиці, котра називається Packages. Дана таблиця складається з наступних полів:

- id – ідентифікатор пакету в ноді. Є первинним ключем таблиці. Автоматично інкрементується на 1 при кожному новому записі в таблицю;
- block_offset – позиція пакету відносно початку файлу зберігання. Є цілим додатнім числом. Унікальне для кожного записаного пакету на заданій ноді. Якщо пакет зберігання вільний поле дорівнює 0;
- real_size – реальний розмір пакету даних. За замовчуванням дорівнює нулю, при заповнені відповідного пакету значення змінюється;
- inblock_offset – внутрішній відступ в середині блоку. Ціле додатне число. За замовчуванням 0.

На рис. 3 зображено uml діаграму бази даних вузла кластеру.

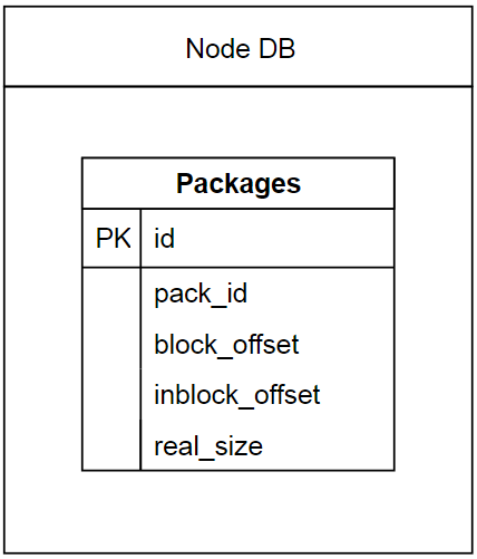


Рис. 3. Схема бази даних ноди

3.3. Сервер керування з’єднанням

Сервер керування з’єднанням або Middleware є основним, керуючим елементом всієї підсистеми керування даними. Він створений, для прийому команд користувача, їх подальшої обробки, та формування відповіді на згадані вище команди. Також даний модуль відповідає за керування доступності файлів для конкретного користувача, та реєстрацію нових користувачів в системі. Він є вхідною точкою для підсистеми користувацького інтерфейсу через , яку останнє взаємодіє з усією системою.

Структурно сервер керування можна розбити на дві умовні частини:

- модуль прийому/передачі даних. Відповідає за коректний прийом та відправку даних. Працює на двох протоколах UDP та TCP, для кожного з яких виділено по потоку на прийом та передачу даних. Після приймання даних, кладе їх в чергу обробки;
- модуль обробки даних. Оброблює команди користувача та формує відповідь на них, передає відповіді до модулю прийому/передачі. Є найбільшою і найважливішою частиною сервера. Взаємодіє з базою даних.

Доступ до бази даних файлової системи має лише модуль обробки даних. Більш детально кожна з частин буде розглянута нижче.

Також важливою частиною сервера є, так званий, Interaction модуль, фактично це пристосована для роботи з декількома потоками одночасно черга, яка в своїй реалізації використовує мютекси для блокування одночасного доступу. Саме така черга використовується для передачі даних і команд в рамках одного модулю і між модулями сервера керування зв’язком.

Варто зазначити, що описана вище структура є спільною для кожного елемента підсистеми керування даними. Різниця полягає тільки у функціонуванні модуля обробки даних.

На рис. 4 наведена структурна схема сервера керування зв’язком (Middleware). Кожна з її частин буде розглянута нижче.

3.3.1. Модуль прийому/передачі даних

Модуль прийому/передачі даних або Transmitter слугує пов’язувальним елементом для різних частин підсистеми керування даними. Саме через нього відбувається як комунікація підсистеми і користувача, так і комунікація між елементами в самій підсистемі. Він є спільним для усіх елементів системи єдина різниця між ними, це черга в, яку модуль кладе, чи з якої він забирає дані, вона може бути модифіковано залежно від частини підсистеми.

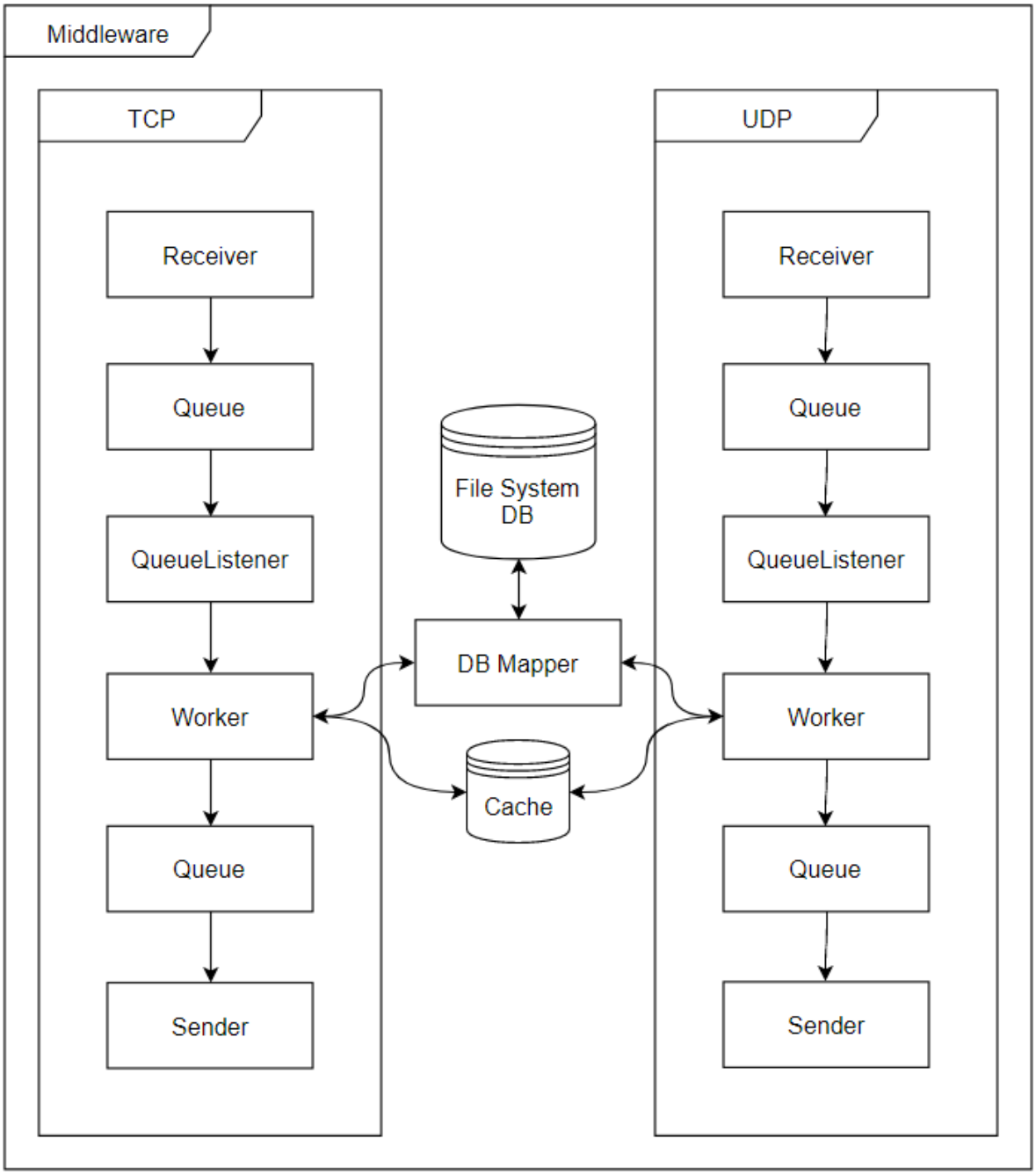


Рис. 4. Структурна схема Middleware

Структурно модуль прийому/передачі складається з двох незалежних частин кожна з яких розбита на два окремі підчастини для взаємодії з TCP і UDP відповідно. Ці частини, це:

- Receiver – підмодуль отримання даних. Поділяється на дві окремі частини для двох згаданих вище протоколів. Приймає запити клієнта, або іншого елемента системи і у разі коректного прийому

кладе запити на опрацювання в чергу опрацювання (вона є спільною для різних протоколів). У випадку UDP під час прийому пакету даних перевіряється відповідність чексум і випадку їх невідповідності надсилається запит на повторне надсилання даних. Кожна з частин підмодуля запускається в одному окремому потоці;

- Sender – підмодуль передачі даних. Отримує відповідь з черги відповідей (на відміну від черги обробки задана черга поділяється на дві по одній на кожен з підтримуваних протоколів) та відправляє отримані дані адресату. У випадку з UDP може повторно відправляти дані у випадку відсутності відповіді від адресата, або відповідного запиту від нього.

Для кожної з вищезгаданих частин необхідно у файлі конфігурації прописати інтернет адресу та порт прийому/передачі даних.

Така система прийому даних, дозволяє максимально розсинхронізувати підсистему керування даних, адже прийом, відправка та обробка даних стають незалежними один від одного частинами.

Розглянемо ближче кожну з описаних підчастин.

Receiver

Receiver – частина Transmitter-a, 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). Підчастини Receiver подані у вигляди неблокуючих сокетів відповідного протоколу, які слугують серверами, що здатні приймати вхідні повідомлення.

Для коректної роботи UDP частини Receiver стандартний протокол обміну даних, був загорнутий в написаний додатковий шар, який розбиває об’єм даних на блоки фіксованого розміру, додає до них заголовкові дані (порядковий номер, фактичний розмір блоку, чек-сума, тощо) та відправляє

отримувачу; основною зміною є впровадження чек-суми, яка використовується для перевірки правильності отримування даних.

Після прийому даних, отримані дані кладуться в чергу виконання. У випадку, якщо дані надійшли по UDP перевіряється чек-сума даних 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 користувачем, кількість разів, або доки пакет не прийде коректним. Описана процедура повторюється визначену користувачем кількість разів, так званий TTR. У разі, якщо пакет не може бути отриманий навіть по завершенню повного циклу відправки, відбувається зупинення процесу передачі з відповідним коментарем для користувача.

Прийом даних по протоколу TCP відбувається в два етапи:

- зчитується розмір вхідного повідомлення: тобто при кожному отримані інформації на сокет, спочатку зчитуються 4 байти (розмір int на 64х системах), які описують розмір вхідного повідомлення;
- 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.

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 модулем обробником 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 новий пакет.

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 чек-суми не співпадають), відбувається повторне надсилання даних.

На відміну від Receiver, кожна з частин Sender-а може бути представлена у вигляді більше ніж одного потоку, що надає можливість пришвидшення відправки відповідей у разі необхідності.

3.3.2. Модуль обробки даних

Модуль обробки даних, або Worker, слугує ядром будь-якою з існуючих частин підсистеми керування даними. Саме в цьому модулі відбувається обробка даних, у випадку сервера керування зв’язком, обробка команд користувача.

Структурно модуль поділяється на декілька незалежних частин:

- Protocol Dispatcher – використовується для сортування даних згідно протоколу їх отримання. Залежно від того TCP це UDP відправляє в різні обробники. Також постійно перевіряє чергу обробки на наявність нових даних і в пазі їх присутності бере їх на опрацювання. Необхідний у зв’язку з тим, що черга обробки одна на два протоколи, а дані отримані різними шляхами треба оброблювати по різному;
- TCP Handler – безпосередньо обробник команд користувача. Отримує від Protocol Dispatcher дані і розподіляє їх між обробниками конкретної команди. Розподіляється на конкретні обробники для кожної з команд. Більш детально структура буде описана нижче;
- UDP Handler – UDP аналог TCP Handler-а. Не містить жодних ідейних відмінностей від останнього. Різниця полягає в тому, що оброблює виключно UDP команди.

Кожна з описаних вище частин є окремим потоком. Це дозволяє максимально пришвидшити процес обробки запитів і даних, крім того значно зменшує кількість «вузьких» місць системи, адже немає жодного місця, де система очікує виконання попередньої команди (при цьому нічого не роблячи).

Також в склад модуля обробки даних можна умовно включити такі дві частини, як: Cache I DataMapper. Перша з них, кеш, використовується для зберігання конкретної інформації в операційній пам’яті і сприяє пришвидшенню роботи системи і зменшенню кількості необхідних запитів в систему. Друга використовується для взаємозв’язку з базою даних. Кожна з цих частин може як належати конкретному модулю обробки (як в Middleware), так і бути спільною для декількох модулів обробки (такий приклад буде наведений нижче). Кеш являє собою словник пристосований для використання одночасно декількома різними потоками, має в своєму складі мютекс і реалізовує весь необхідний словники функціонал.

DataMapper містить в собі функції для під’єднання до бази даних, відправки, прийому, та обробки отриманих даних. В ньому немає механізму роботи з декількома потоками, але такі механізми реалізовані в самих базах даних, тому в даному випадку вони не є необхідними.

На рис. 4.1 на прикладі сервера контролю зв’язку користувача наведена структурна схема модулю обробки даних.

Розглянемо детальніше структуру Handler-ів, без прив’язки до протоколу, адже структура є спільною, а відрізняються лише самі конкретні обробники. Отже кожен обробник складається з:

- методу сортування команд чи даних;
- словнику з конкретних обробників. Ключ в такому словнику є назвою команди, а значенням безпосередньо сам конкретний обробник.

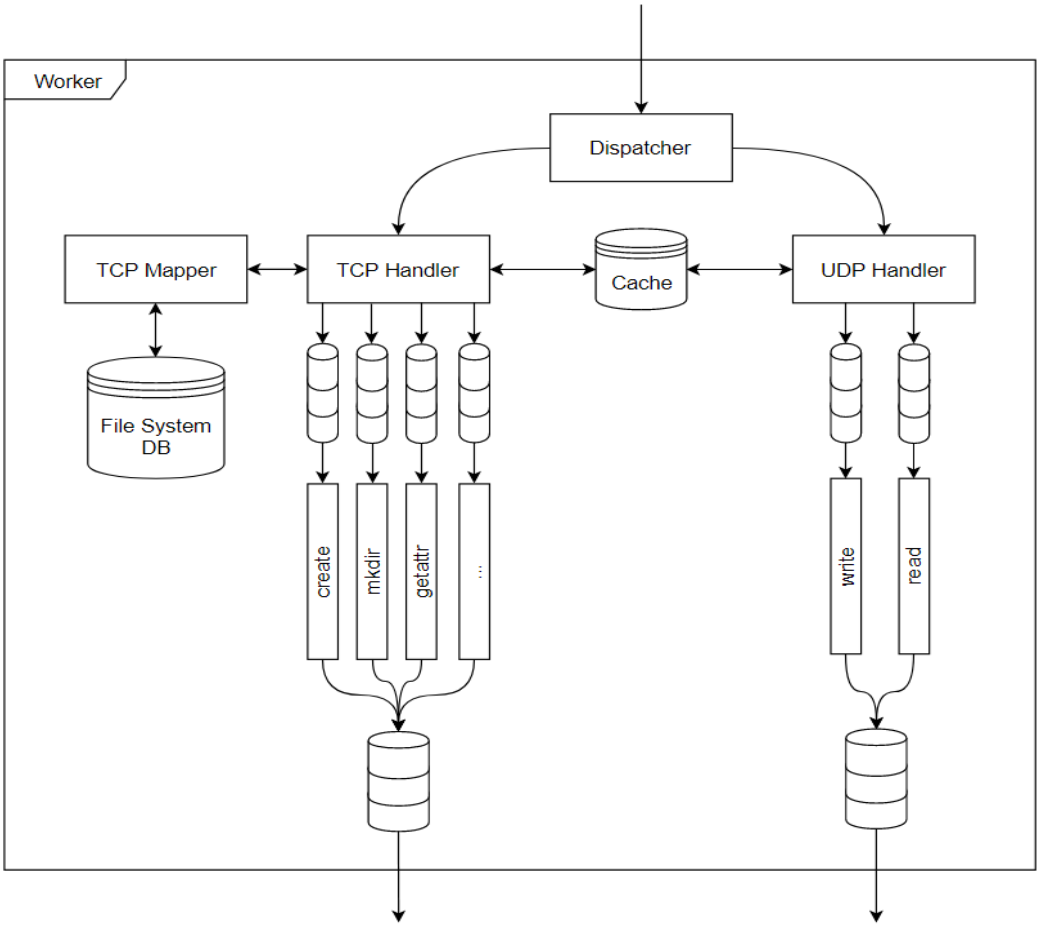


Рис. 4.1. Структурна схема модуля обробки сервера керування зв’язком

Обробники команд (конкретні обробники) являють собою окремий потік, з власною, невеликою, чергою завдань (черга аналогічна до вищеописаної) та безпосередньо метода обробника. Особливістю таких обробників є те, що потік, в якому вони виконуються я не постійним, а здатен засинати після певного часу простою (черга виконання пуста), і знов прокидатися, після додавання нової команди. Це дозволяє максимально розпаралелити обробки, при цьому не надто навантажувати процесор обробкою великої кількості потоків.

Крім вище названих частин кожен з Handler-ів має відповідну чергу відповідей (для TCP чи UDP відповідно до типу Handler-а), в яку кладе відповідь сформовано після обробки команди чи пакету даних. Також кожен з конкретних обробників має можливість доступу та редагування кеша і доступ до Datamapper-а. В окремих випадках, в разі необхідності, наприклад

при неможливості виконання конкретної команди в даний момент часу конкретний обробник, може повертати команду чи пакет даних назад в чергу обробки. Такий підхід використовується при обробці пакетів даних, оскільки балансування пакетів відбувається в іншій частині підсистеми керування даними, а необхідність прийому і обробки пакетів даних залишається.

3.4. Кластер менеджер

Кластер менеджер (ClusterManager) є місцем де відбуваються всі зміни кластера та файлової системи. Він умовно складається з двох незалежних частин, які для взаємодії між собою використовують кеш та модуль передачі/обробки даних. Вищезгаданими частинами є:

- File System Manager – частина, яка відповідає за керування файловою системою, саме тут відбувається, створення нових фалів, балансування навантаження на кластер, тощо. Саме в цій частині відбувається балансування навантаження між нодами кластера. Балансування відбувається на основі інформації про вільний простір на ноді і кількості пакетів, які чекають на відправку на конкретну ноду. Якщо дані не можуть бути записані на відбалансовану ноду, з сервера керування звязками приходить відповідний запит і відбувається повторне налаштування ноди, до повторного балансування на Cluster Controller; відправляється запит на виключення ноди з списку живих;
- Cluster Controller – частина, що відповідає за роботу з кластером. Тут відбувається перевірка статусу ноди, реєстрація нової ноди. Також ця частина є відповідає за взаємозв’язок з конкретними вузлами кластеру. Прямим призначенням цієї частини є перевірка життєдіяльності кожного з вузлів. Дана ціль досягається завдяки системі health monitor, яка відслідковує відповідні запити з кожного з вузлів кластера і у випадку, якщо якийсь із вузлів протягом певного

часу не передав відповідний запит, виключає вузол з списку активних, після чого вузол перестає бути доступною на запис чи читання. Також вузол може бути вимкнений у разі, якщо він не зміг прийняти дані, в такому випадку з File System Manager приходить відповідний запит.

Структура кластер менеджера представлена на рис. 5.

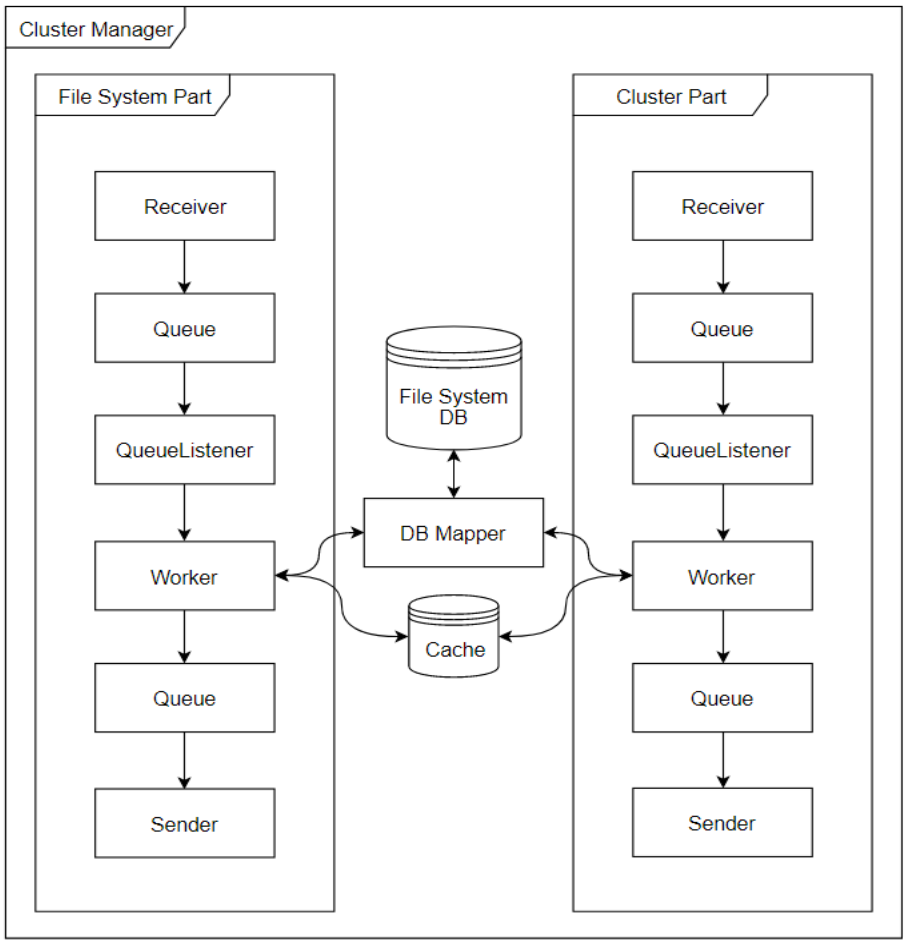


Рис. 5. Структура кластер менеджера

Як видно з малюнку кластер менеджер складається з незалежних частин, які можуть взаємодіяти через кеш непрямим шляхом, і через модуль прийому/передачі прямим. Під непрямим шляхом мається на увазі, що один елемент модулю може класти певні дані в кеш, а інший їх звідти діставати (детальніше цей процес описаний в 4 розділі).

Структурно кожен з елементів системи ніяк не відрізняється від писаного вище сервера керування зв’язком. Основна відмінність полягає у тому, що кластер менеджер це єдине місце в усій підсистемі керування даними з якого можна модифікувати базу даних файлової системи, решта елементів підсистеми, або взагалі не мають до неї доступу (до прикладу вузол кластера), або мають тільки права читання. Також особливість даної частини є те, що замість стандартного одного Worker-а даний підмодуль має одразу два незалежних, кожен з яких має власний підмодуль прийому/передачі даних. Необхідність об’єднання згаданих вище частин є їх дуже близька функціональна взаємодія та логічне призначення, адже коректна робота жодної з них неможлива без іншої.

Ще однією важливою особливістю кластер менеджера є те, що йому не доступний протокол UDP, тобто всі відповідні функції і обробники відключені. Це пояснюється тим, що немає необхідності передавати пакети даних на кластер менеджер, а отже немає необхідності у використанні вищезгаданого протоколу.

Детально зупинятись на розгляді кожної з частин не будемо, адже структурно вони подібні сервер керування зв’язком.

3.5. Вузол кластера

Вузол кластера, або нода – основна одиниця зберігання даних. Саме тут фізично зберігаються всі пакети даних передані користувачем. Структурно подібна до сервера керування зв’язком з декількома особливостями. Перша, з них, це те, що у зв’язку з малою функціональною різноманітністю окремі обробники для UDP та TCP протоколів були об’єднані в один великий обробник, при цьому механізм прийому/передачі даних так і лишився бінарним. Другою особливістю є те, що Worker даної системи заточений не на обробку команд, а на зберігання даних на диску і їх зчитку звідти.

Для зберігання даних вузол кластер використовує попередньо створений файл заданого розміру (задається в налаштуваннях). Даний файл б’ється на блоки відповідного розміру (розмір співпадає з розміром пакету передачі даних). Також модуль обробки в даній частині підсистеми займається обліком вільних блоків в файлі. Ще однією особливістю даної частини є те, що вона, незалежно від отриманих команд і даних, кожен наперед визначений проміжок часу (час визначається в конфігураційних файлах) відправляє сигнал на Cluster Controller, про те що вузол кластера є активним.

Останнім важливим зауваження стосовно вузла кластера є те, що саме тут відбувається реплікація даних в кластері. Вузол пересилає ці дані двом своїм випадково обраним «сусідам» після чого повідомляє про це Cluster Controller. Структура вузла подібна до структури сервера керування зв’язком, тому окремий малюнок не наводитиметься, заначу, що єдиною структурною відмінністю є те, що замість кеша (він не використовується) використовується файл.

4. ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ПРОГРАМНИХ ЗАСОБІВ

В цій частині описано особливості програмної реалізації всіх частин підсистеми керування даними і описані можливості подальшого вдосконалення системи.

4.1. Підсистема управління даними

Розглянемо особливості реалізації підсистеми керування даними на прикладі кожної окремої частини даної підсистеми.

4.1.1. Сервер керування зв'язком

Спершу розглянемо обмеження накладені на сервер у зв'язку з недостатньою кількістю часу та засобів програмної розробки. Основним таким обмеженням є зменшення кількості користувацьких функцій, які підтримує система. Так розроблена система підтримує наступні функції: open, create, mkdir, getattr, readdir, flush, read, write. Відповідно до кожної з цих функцій був створений конкретний обробник в TCPHandler. Також обмеженням можна назвати неможливість використання символа & в назві шляху, адже цей символ використовується для конкатенації команди і її аргументів при передачі по TCP. Крім того загальним обмеженням системи можна вважати неможливість одночасного відкриття більш ніж 128 файлах, більш детальніше про це буде описано нижче. На цьому відомі обмеження закінчуються.

Перейдемо безпосередньо до особливостей реалізації.

Першою такою особливістю, вона є спільною для всіх частин підсистеми, є спосіб передачі команд, вони передаються по протоколу TCP, але не просто як текст, команди конкатенуються з їхніми атрибутами і передаються як єдине ціле, відповідь приходить у такому самому форматі. Конкатенація відбувається за допомогою символа &, який використовується для розділення частин запиту.

Також особливим є структура передачі пакету даних по UDP, ця особливість є спільною як для сервера керування зв’язком так і для вузла кластера, пакет даних визначеного розміру (за замовчуванням це 512 байтів) обгортається додатковими даними до яких належать (далі по їх порядку слідування в пакеті): файловий дескриптор файлу, до якого ці данні належать (4 байти), порядковий номер пакету в файлі (4 байти), розмір блоку даних (4 байти), далі самі дані (розмір завжди 512 байт, але актуальні дані лише в перших n байтах, які були визначені попереднім полем) і останньою йде хеш-сума пакету. Як стає очевидно з вищесказаного розмір хедерів пакету даних завжди складає рівно 14 байт. Дані передаються блоком по 512 байт, незалежно від фактичного вмісту для того, щоб з сокета можна було зчитувати сталу кількість байт. При отриманні пакету всі дані, включно з хедерами (окрім останніх 2 байт вони обнуляються) відправляються в функцію обрахунку хеш-суми, потім отримана сума порівнюється з тією, що прийшла в пакеті і робиться висновок про успішність чи неуспішність передачі пакету даних. В разі неуспішного прийому даних Receiver відправляє запит на повторне отримання даних.

Особливістю прийому даних по TCP є те, що дані приймаються в два підходи, спочатку зчитуються перші 4 байти, які описують розмір відправленої команди, а потім зчитується сама команда заданого розміру. Це зумовлене тим, що сокет для прийому даних один і всі команди в ньому мають різний розмір.

Як можна було зрозуміти з описаного вище Worker (модуль обробник) даної частини реалізовує наступні функції:

- open;
- create;
- mkdir;
- getattr;
- readdir;
- flush;

- read;
- write;
- load;
- cache_add.

Всі крім останніх двох функцій потрібні для обробки команд користувача, останні дві допоміжні функції, які необхідні для правильної комунікації між елементами підсистеми.

Також в модулі обробнику реалізовані функції read і write для обробника UDP, вони використовуються для зчитування і запису пакетів даних відповідно. Кожна з описаних вище функцій є засинаючим потоком, який переводиться у сонний режим у випадку якщо немає ніяких даних до обробки протягом певного часу.

Детальніше розглянемо кожну з наведених вище функцій і опишемо алгоритм їх роботи.

Першою, за використанням і значенням йде функція open. Як і решта функцій-обробників, вона винесена в окремий потік і має свою власну чергу команд. Основною задачею функції є генерування файлового дескриптора для конкретного файла, переданого користувачем. Для генерації використовується функція, яка була спеціально розроблена для цих цілей, вона забезпечує унікальність дескриптора. Особливістю дескриптора є те, що він знаходиться в рамках від 0 до 127, це зумовлене тим, що дескриптор, для зменшення розміру даних, що передаються, повертається користувачу у вигляді ASCII символу. На вхід функція open приймає повний шлях до файла, що необхідно відкрити, так званий pathname та порт , на вихід, як вже було сказано вище, подається згенерований файловий дескриптор. В ході виконання самої функції, остання звертається до бази даних, для перевірки існування файлу, якщо файлу немає користувачу відправляється відповідна помилка. В разі існування файлу, відбувається генерування дескриптора, для цього файлу, даний дескриптор вноситься в кеш, для зручності співставлення дескриптора і pathname. Згенерований дескриптор кладеться

в чергу відправку, обгортаючись в кортеж (тип даних в мові Python, який являє з себе незмінний список) разом з адресом отримувача. Адреса отримувача генерується з двох параметрів: інтернет адреси і порт прийому даних. Останній передається разом з pathname, як вхідний параметр. Адреса відповідає адресі з якої був отриманий запит і дістається автоматично, така система формування адреси відповіді використовується для всіх запитів клієнта і для деяких функцій в середині підсистеми.

Наступною функцією, є метод create. Метод використовується для створення нових регулярних файлів в файловій системі. На вхід метод приймає параметри ідентичні до вхідних параметрів методу open, а саме pathname і порт на який мусить бути відправлена відповідь. Вихідними атрибутами є статус (успішно чи ні відпрацював запит). В ході роботи метод перевіряє чи існує вже файл з зазначеним pathname, якщо існує, то користувачу повертається відповідна помилка, інакше на кластер менеджер відправляється запит на створення нового файлу.

Далі розглянемо функцію mkdir є аналогом функції create, але викликається для тек. Має спільні з функцією create вхідні та вихідні параметри. Алгоритм роботи не відрізняється від алгоритму попередньої функції.

Функція getattr. Використовується для отримання інформації про файл. На вхід функції подаються pathname і порт отримувача. Виходом функції є сконкатенований рядок з атрибутів відповідного файлу. До атрибутів фалу відносяться: розмір та тип. Для отримання інформації про конкретний файл функціє звертається до бази даних і проводить пошук файлу по pathname. В разі, якщо пошук не дав результатів (файл не існує) користувачу повертається відповідна помилка, інакше отримані дані конкатенуються і відправляються отримувачу.

Наступною функцією, яку ми розглянемо є функція readdir. Функція використовується для отримання інформації про вміст теки. Як і для попередніх функцій вхідними параметрами є: pathname та порт отримання

відповіді. Функція виконує пошук в базі даних по отриманому pathname і відповідному типу файлу, далі з отриманого результату (якщо він є) дістається інформація про вміст директорії (поле data), після чого робиться ще один запит для отримання pathname відповідних файлів. Результат останнього запиту конкатенуються і передається користувачу. В разі відсутності вхідного pathname і базі даних, користувачу повертається відповідна помилка.

Функція flush використовується для закриття файлу, фактично це означає, що дескриптор отриманого файлу буде видалений з кеша підсистеми. Також дана функція гарантує те, що всі операції над файлом, що підлягає закриттю будуть завершені. Це досягається шляхом підняття пріоритета для пакетів відповідного файлу, що гарантує їх якнайшвидше опрацювання. Як було сказано вище, на вхід функція приймає дескриптор файла, який підлягає закриттю і порт відповіді. Як результат відправляється статус закриття файлу (успішно чи ні).

Наступною йде функція read. Це одна з основних функцій всієї підсистеми, яка в ході свого відпрацювання зачіпає всі частини підсистеми. Функція використовується для читання відповідної кількості даних з відповідного файлу. Як вхідні параметри функція приймає: дескриптор файлу, який треба зчитати, кількість даних (в пакетах), що підлягаються зчитанню, відступ від початку файла (місце з якого треба проводити читання), порт для прийому інформації по TCP протоколу, порт для прийому інформації по UDP протоколу. Після отримання даної команди, відбувається перевірка файлового дескриптора на існування, якщо його не існує, то відповідна помилка відправляється користувачу. Якщо перевірка закінчується успішно, то формується запит на кластер менеджер. В запит входять: дескриптор, pathname, кількість пакетів для зчитки, відступ від початку файлу. Далі запит відправляється безпосередньо на кластер менеджер. Під час відпрацювання функції порти користувача вносяться в кеш частини, для подальшого використання в інших функціях.

Функція write використовується для запису даних в існуючий файл. Варто зазначити, що є певні обмеження на запис даних, так дані можуть тільки дописуватись в файл, оскільки механізмів для видалення блоків в систему не закладено. На вхід дана функція приймає: файловий дескриптор, та кількість пакетів для запису. Після отримання даних, формується запит на балансування, який відправляється на кластер менеджер. В запит входять: кількість пакетів для балансування, pathname файла. Варто зазначити, що дана функція нічого не повертає користувачу. Також функція може бути застосована тільки до регулярних файлів.

Наступною реалізованою функцією є load. Вона використовується для відправлення відповідним вузлам запиту на отримання відповідних пакетів, крім того функція відправляє користувачу інформацію про кількість пакетів, які йому будуть відправлені. Ця функція є елементом алгоритму зчитування даних. На вхід вона приймає: кількість пакетів, список, кожен елемент якого є кортежем з трьох елементів (ідентифікатором вузла кластера, ідентифікатором пакету даних, порядковим номером пакету даних в файлі). Після отримання даних,, спочатку відбувається відправлення користувачу інформації про кількість пакетів, що будуть надіслані, потім відбувається розсилання запитів на отримання пакетів.

Останньою серед реалізованих функцій в цій частині, є функціє cache_add, вона використовується для занесення в кеш даних, отриманих через мережу. Вона створена для того, щоб надати можливість іншим частинам підсистеми впливати на вміст кешу сервера керування зв’язком. Функція не відправляє жодних відповідей. На вхід приймає два списки, перший з яких є списком ключів, за якими дані будуть занесені в кеш, другий є списком значень для занесення. Після отримання даних відбувається парсинг даних з подальшим їх занесенням в кеш.

Варто відзначити, що термін вхідні параметри подається не зовсім в звичайному сенсі. Як було описано вище на вхід кожної функції подаються два параметри payload та address. Перший параметр відповідає за атрибути

передані в цю функцію, власне вміст цих атрибутів і є вхідними параметрами описаними вище, другий параметр описує адресу клієнту з якої прийшов запит. Під відправкою слід розуміти поміщення сформованого запиту в чергу відправки даних. Описана в особливість є спільною для всіх частин підсистеми.

Окремим блоком функцій є блок функцій, що взаємодіють з UDP протоколом. Таких функцій всього дві.

Першою з таких функцій є функція write. Вона приймає пакети даних від користувача, перепаковує їх та відправляє на зберігання на відповідний вузол кластера. Інформація про вузол зберігання дістається з кешу програми. В разі якщо для конкретного пакету ще не визначений вузол, пакет даних кладеться в кінець черги опрацювання.

Другою, і останньою, функцією з заданого блоку є функція read. Вона виконує схожі до попередньої функції дії, але в зворотньому порядку. Тобто спочатку отримує дані від вузла кластера, потім перепаковує пакет і відправляє користувачу. Інформація про користувача знаходиться в кеші.

4.2. Кластер Менеджер

Крім спільних для всієї підсистеми особливостей в кластер менеджері наявні особливості реалізації самого кластер менеджера. Так це єдина частина підсистеми, яка містить два незалежних модулі обробника, при тому, що кеш і DataMapper у них спільний. Це пов’язано з необхідністю розмежування функціональних можливостей з роботи з файловою системою, і можливостей взаємодії з кластером. Необхідність спільного використання кеша зумовлена, тим, що список активних нод відслідковується в реальному часі і їх облік ведеться саме в кеші. А так як список активних нод необхідний як File System Manager так і Cluster Controller, то виникає необхідність в спільній пам’яті, якою і слугує кеш. Спільне використання мапера бази даних повязане з тим, що обидві вищезгадані частини звертаються до однієї і тієї самої бази даних і

використання спільного мапера значно спрощує взаємозв’язок між кластер менеджером і базою даних.

В File System Manager реалізовані наступні функції:

- read;
- write;
- create;
- mkdir;
- status.

Всі функції крім останньою слугують для реалізації функцій описаних в попередній підглаві. Остання використовується для зміни стану файла.

В Cluster Controller реалізовано наступні функції:

- init;
- alive;
- status.

Перші дві функції необхідні для взаємодії з кластером. Остання використовується для зміни стани пакету і як вихідна точка зміни стану файлу. Розглянемо функції детальніше.

Функція read застосовується для формування списку пакетів для читання. В сформований список входять: ідентифікатор пакету, порядковий номер пакету в фалі, дескриптор файлу, до якого пакет належить, ідентифікатор вузла, на якому зберігається пакет даних. На вхід функціє приймає: дескриптор файлу, для читання, pathname, кількість пакетів для читання, відступ від початку файла. Після генерації вищезгаданого списку відбувається його сереалізація і відправка даних на сервер керування зв’язком.

Функція write використовується для резервації місця на кластері, крім того саме тут відбувається балансування навантаження на вузли кластера. На вхід функція отримує наступні параметри: pathname, кількість пакетів для запису. Ґрунтуючись на даних про кількість вільного місця на конкретному вузлі і кількість пакетів, що очікують запису на конкретний

вузол, відбувається відбір вузла зберігання для кожного з пакету. Після цього відповідна кількість пакетів створюється в базі даних. Далі на сервер керування зв’язком відправляється запит з інформацією про новостворені пакети.

Функції create та mkdir використовуються для створення регулярних файлів та тек відповідно. Працюють однаково єдиною відмінністю є те, що функція create після відпрацювання відправляє додатковий запит на сервер керування зв’язком. На вхід функції приймають pathname, після чого відбувається створення нового запису в базі даних і, у випадку create, запит на сервер керування зв’язкам.

Функція статус використовується для зміни статусу файлу, відповідно на вхід приймає pathname та новий статус. Після чого відбувається запит в базу даних на зміну статусу файла.

Наступними йдуть функції Cluster Controller. Першою з них є функція init. Використовується для реєстрацію вузла кластера в системі, відповідно вхідними параметрами функції є адреса вузла в мережі інтернет і два порти, по одному для протоколів TCP та UDP. Після отримання даних відбувається створення запису в базі даних. Також у вхідних даних передається інформація про кількість вільного місця вузла, ця інформація поміщується в кеш. У відповідь вузлу кластера повертається його ідентифікатор в системі.

Функція alive використовується для підтвердження активності вузла кластера. На вхід приймається ідентифікатор вузла, який відправив запит. Скидає таймер відключення вузла, який використовується для переведення останнього в неактивний режим після певного часу простоювання.

І останньою є функція status, що використовується для зміни статусу відповідного пакету. На вхід приймає ідентифікатор пакету даних і новий статус, після чого відбувається зміна статусу в базі даних. У разі, якщо всі пакети конкретного файлу переходять в активний стан, автоматично генерує запит до кластер менеджера з проханням змінити стан всього файла.

4.3. Вузол кластера

Для вузла як і для всієї системи в цілому існує ряд спільних особливостей, описаних вище. Окрім вже згаданих особливосте й також є певний перелік особливостей унікальних конкретно для цієї частини підсистеми. До таких особливостей можна віднести:

- відсутність кеша. Це єдина частина підсистеми управління даними, яка не використовує інструмент кеша, адже він тут просто не потрібний. Замість нього використовується файл, який одночасно є і місцем збереження даних;
- єдина частина підсистеми в якій обробники різних протоколів об’єднані в один загальний обробник. Поясненням цієї особливості є те, що вузол кластера реалізує надто малу кількість функцій для розбиття їх на обробники за протоколом;
- використовує власну, незалежну базу даних. Як було описано вище вузол кластера має свою, унікальну для кожного вузла, базу даних. Це аргументується тим, що вузол зберігає лише інформацію про місце збереження конкретного пакету даних в спеціальному файлі і не має доступу до загальної бази даних системи, Такий підхід дозволяє дещо підвищити рівень безпеки системи, адже жоден вузол нічого не знає про блоки памяті, які в ньому збережені.

Дана частина підсистеми реалізовує всього дві функції, а саме:

- write;
- read.

Розглянемо кожну з цих функцій більш детально. Функція write використовується для запису пакету даних в файл зберігання. Це функція UDP протоколу. На вхід передається безпосередньо сам пакет даних. Після отримання пакету він зберігається в одному з вільних блоків в файлі. Після зберігання пакету даних, на кластер менеджер відправляється запит на зміну статусу збереженого пакету.

Останньою функцією в цій частині є функція read. Використовується для зчитки і подальшої відправки даних на сервер керування зв’язком. На відміну від попередньої функції використовує обидва протоколу, але активується протоколом TCP. Як вхідний параметр використовується ідентифікатор пакету, для передачі. Після отримання параметрів відбувається запит в базу даних вузла для отримання зсувів конкретного пакету даних, після цього пакет зчитується з файла і передається на сервер керування зв’язком по протоколу UDP.

Крім вищеописаних функцій на вузлі кластера існує функція, яка є вічно зациклена сама на себе і використовується для відправки is_alive запита на кластер менеджер. Функція називається alive вона активізується після запуску вузла. В ході роботи вона відправляє запит на кластер менеджер, після чого створює таймер (об’єкт який виконує переданий колбек після певного, визначеного користувачем часу) в який поміщає себе і задає час повторного запиту. Функція необхідна для сигналізації кластер менеджеру про активний статус вузла.

4.4. Рекомендації щодо подальшого вдосконалення

Розіб’ємо рекомендації по вдосконаленню на умовні пункти, що відповідають частинам підсистеми і розглянемо кожну з них окремо.

4.4.1. Сервер керування зв’язком

Подальші вдосконалення даної частини в основному пов’язані з розширенням функціональної бази в частині обробки запитів клієнта, тобто збільшення доступних для обробки функцій. Також подальше вдосконалення може заключатись в тому, щоб змінити схему передачі даних з та на ноду, наприклад перейти до механізмів TCP та UDP patching, що значно розширить можливості внесення ноди в кластер, адже в такому разі ноди зможуть знаходить за фреймворком, або взагалі не мати публічного ip. Дане вдосконалення вимагає перебудови механізму взаємодії між сервером

керуванням зв’язку і вузлами кластеру, але може бути корисних у випадку розміщення но в різних частинах світу.

Ще одним вдосконаленням, не так сервера керування як структури в цілому, я додавання до підтримуваних протоколів протоколу TLS, що дозволить значно підвищити захист даних користувача. Дане покращення вимагає введення системи приватного і публічного ключів, але б значно підвищило захищеність даних від зовнішніх чинників.

Ще одним цікавим вдосконаленням є надання можливості додавання додаткових серверів обробників під сервер керування зв’язком, адже він, фактично, є вузьким місцем системи і від його продуктивності залежить загальна продуктивність системи.

4.4.2. Кластер Менеджер

До основних покращень Кластер Менеджера, крім очевидного розширення функціональної частини обробки запитів клієнта, можна додати створення механізму керування ролями користувачів. Тобто створити декілька різних за можливостями ролей, наприклад адміністратору надати можливість переглядати та редагувати кластер онлайн. Дана вдосконалення вимагає створення спеціалізованого інтерфейсу для адміністрування, але змогло б значно спростити керування кластером. Крім того введення додаткових ролей значно розширює загальну кількість функціональності всієї системи.

Ще одним важливим поліпшенням є вдосконалення системи балансування навантаження на кластер, щоб при балансуванні навантаження використовувалась інформація не тільки про вільний простір і кількість пакетів в очікувані на відправку, а наприклад затримка мережі чи продуктивність самої ноди. Такий підхід значно б покращив балансування та підвищив продуктивність всієї системи.

Також цікавим виглядає введення механізмів TCP та UDP patching для роботи з нодами. Це б дозволило відмовитись від необхідності зберігання інформації про ноди, а просто ввести їх облік в реальному часі, також така

система дозволила б використовувати ноди, яким не доступний з різних причин публічна адреса в мережі інтернет. Але варто зауважити, що така система вимагає підвищених заходів безпеки, адже відслідкувати кінцевий сервер зберігання стає неймовірно важко.

4.4.3. Вузли кластеру

Основне поліпшення вузлів кластеру полягає у зміні середовища зберігання даних з файлу на операційну систему самого фізичного пристрою на якому встановлена нода, це дозволить значно розширити і дещо спростити механізми взаємодії з даними.

Також цікавим вдосконаленням є зміна механізму реплікації даних з випадкового вибору двох «сусідів» на механізм підрахунку оптимальних нод для реплікації. Проте це вдосконалення неможливе в рамках TCP та UDP patching, адже тоді система нічого не знатиме про «сусідні» ноди, а це означає, що механізм реплікації мусить бути винесений з ноди в кластер менеджер чи сервер керування зв’язком.

Також ідея з TCP та UDP patching є актуальної і для цієї частини. Проте для її використання прийдеться повністю змінити механізми взаємодії ноди з рештою системи.

Крім того можна додати шифрування даних, що зберігаються, це дозволить не хвилюватись про те, що дані будуть вкрадені.

Крім того оптимістично виглядає додавання можливості дефрагментування пакетів на ноді, що дозволить значно підвищити максимальний об’єм зберігання даних.

ВИСНОВКИ

Метою даного дипломного проекту було створення підсистеми керування даними файлової системи.

Розроблена підсистема складається з наступних складових: сервер керування зв’язком, кластер менеджер, вузол кластера. Також були розроблені дві бази даних: база даних файлової системи і база даних ноди.

Аналізуючи наявні рішення було вирішено, що оптимальним стеком засобів реалізації підсистеми буде мова програмування Python з відповідними бібліотеками для роботи з PostgreSQL та SQLite. А для розробки необхідних баз даних будуть використані згадані вище СКБД. Крім того обрано протоколи UDP та TCP для передачі даних

Сервер керування зв’язком забезпечує:

- прийом команд та пакетів даних від користувача;
- обробку вищезгаданих команд і формування відповіді на них з подальшим відправленням клієнту;
- передачу пакетів даних на ноду для їх подальшого використання та зворотній процес.

Кластер менеджер забезпечує:

- облік кластеру. Реєстрацію нових вузлів кластера;
- балансування навантаження на кластер. Зміни у файловій системі, створення нових фалів.

Вузол кластера забезпечує:

- зберігання даних користувача;
- реплікацію вищезгаданих даних.

Можливими напрямками подальшої роботи щодо вдосконалення розробки описані у відповідному пункті пояснювальної записки.

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. 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.
6. GlusterFS Documentation [Електронний ресурс]. – Режим доступу: https://docs.gluster.org/en/latest/. – Дата доступу: січень 2019. – Назва з екрана.
7. Уайт, Т. Hadoop. Подробное руководство [Текст]. – 2-е. – СПб.: Питер, 2013. – 672 с. – 1000 экз. – ISBN 978-5-496-00662-0.
8. Лэм, Ч. Hadoop в действии [Текст]. – ДМК Пресс, 2012. – 424 с. – 500 экз. – ISBN 978-5-97060-156-3, 978-5-94074-785-7.
9. 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.

10. 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.
11. 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.
12. 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.
13. 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.
14. Python is a programming language that lets you work quickly and integrate systems more effectively [Електронний ресурс]. – Режим доступу: https://www.python.org/. – Дата доступу: січень 2019. – Назва з екрана.
15. Самоучитель Python [Електронний ресурс]. – Режим доступу: https://pythonworld.ru/samouchitel-python. – Дата доступу: січень 2019. – Назва з екрана.
16. Саммерфилд, М. Python на практике [Текст] : пер. з англ. – М.: ДМК Пресс, 2014. – 338 с. – ISBN 978-5-97060-095-5.
17. Лутц, М. Программирование на Python [Текст] : пер. з англ. – 4-е изд. – СПб.: Символ-Плюс, 2011. – Т. I. – 992 с. – ISBN 978-5-93286-210-0.
18. Лутц, М. Программирование на Python [Текст] : пер. з англ. – 4-е изд. – СПб.: Символ-Плюс, 2011. – Т. II. – ISBN 978-5-93286-211-7.
19. Лутц, М. Изучаем Python, 4-е издание [Текст] : пер. з англ. – СПб.: Символ-Плюс, 2010. – 1280 с – ISBN 978-5-93286-159-2.
20. Бизли, Д. Python. Подробный справочник, 4-е издание [Текст] : пер. з англ. – СПб.: Символ-Плюс, 2010. – 864 с – ISBN 978-5-93286-157-8.

21. Саммерфилд, М. Программирование на Python 3. Подробное руководство [Текст] : пер. з англ. – СПб.: Символ-Плюс, 2009. – 608 с – ISBN 978-5-93286-161-5.
22. Гифт, Н. Python в системном администрировании UNIX и Linux [Текст] / Дж. Джеремі : пер. з англ. – СПб.: Символ-Плюс, 2009. – 512 с – ISBN 978-5-93286-149-3.
23. Бизли, Д. Язык программирования Python. Справочник [Текст]. – К.: ДиаСофт, 2000. – 336 с. – ISBN 966-7393-54-2, ISBN 0-7357-0901-7.
24. Сузи, Р.А. Python. Наиболее полное руководство [Текст]. – СПб.: БХВ-Петербург, 2002. – 768 с. – ISBN 5-94157-097-X.
25. Сузи, Р.А. Язык программирования Python: Учебное пособие [Текст]. – М.: ИНТУИТ, БИНОМ. Лаборатория знаний, 2006. – 328 с. – ISBN 5-9556-0058-2, ISBN 5-94774-442-2.
26. Хахаев, И.А. Практикум по алгоритмизации и программированию на Python. Учебник [Текст]. – М.: Альт Линукс, 2010. – 126 с. – (Библиотека ALT Linux). – ISBN 978-5-905167-02-7.
27. Фёдоров, Д.Ю. Основы программирования на примере языка Python. Учебное пособие [Текст]. – СПб.: Юрайт, 2018. – 167 с. – ISBN 978-5-534-04479-9.
28. 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.
29. Керниган, Б. Язык программирования Си [Текст] / Д. Ритчи. – 2-е изд. – М.: Вильямс, 2007. – С. 304. – ISBN 0-13-110362-8.
30. Гукин, Д. Язык программирования Си для «чайников» [Текст]. – М.: Диалектика, 2006. – С. 352. – ISBN 0-7645-7068-4.
31. 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.

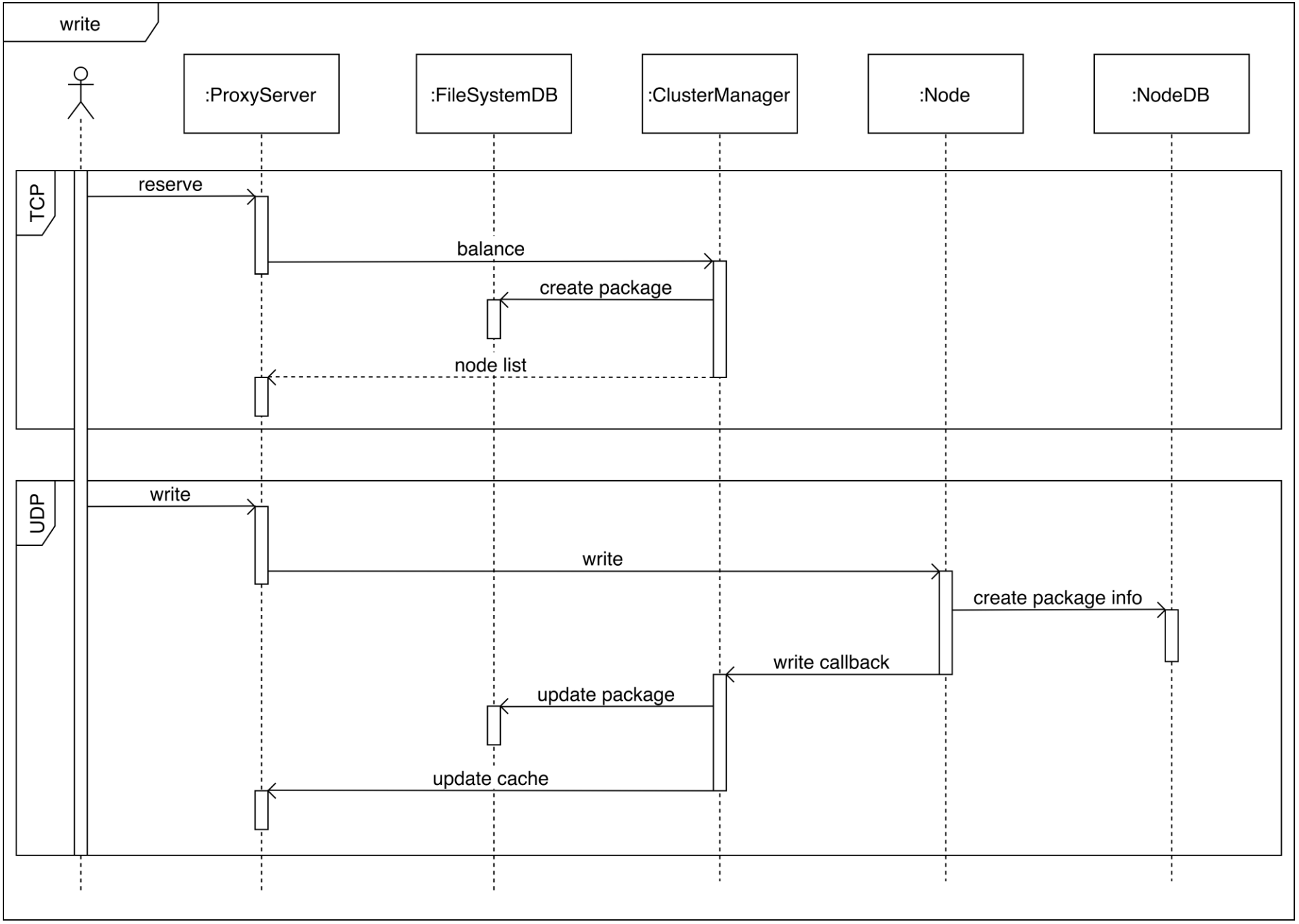
32. Прата, С. Язык программирования С. Лекции и упражнения [Текст]. – М.: Вильямс, 2006. – С. 960. – ISBN 5-8459-0986-4.
33. Прата, С. Язык программирования С (С11). Лекции и упражнения, 6-е издание [Текст]. – М.: Вильямс, 2015. – 928 с. – ISBN 978-5-8459-1950-2.
34. Столяров, А.В. Язык Си и начальное обучение программированию [Текст] / Сборник статей молодых учёных факультета ВМК МГУ. – Издательский отдел факультета ВМК МГУ, 2010. – № 7. – С. 78-90.
35. Шилдт, Г. С: полное руководство, классическое издание [Текст]. – М.: Вильямс, 2010. – С. 704. – ISBN 978-5-8459-1709-6.
36. 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.
37. Cython C-Extensions for Python [Електронний ресурс]. – Режим доступу: https://cython.org/. – Дата доступу: квітень 2019. – Назва з екрана.
38. Welcome to Cython’s Documentation [Електронний ресурс]. – Режим доступу: https://cython.readthedocs.io/en/latest/. – Дата доступу: квітень 2019. – Назва з екрана.
39. PostgreSQL: The World's Most Advanced Open Source Relational Database [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/. – Дата доступу: квітень 2019. – Назва з екрана.
40. Работа с PostgreSQL в Python [Електронний ресурс]. – Режим доступу: https://khashtamov.com/ru/postgresql-python-psycopg2/. – Дата доступу: квітень 2019. – Назва з екрана.
41. Python-PostgreSQL Database Adapter – psycopg2 [Електронний ресурс]. – Режим доступу: https://pypi.org/project/psycopg2/. – Дата доступу: квітень 2019. – Назва з екрана.

42. What Is SQLite? [Електронний ресурс]. – Режим доступу: https://www.sqlite.org/index.html. – Дата доступу: квітень 2019. – Назва з екрана.
43. Самоучитель Sqlite [Електронний ресурс]. – Режим доступу: https://sites.google.com/site/javatokens/sqlite. – Дата доступу: квітень 2019. – Назва з екрана.
44. Модуль sqlite – Работаем с базой данных [Електронний ресурс]. – Режим доступу: https://python-scripts.com/sqlite. – Дата доступу: квітень 2019. – Назва з екрана.

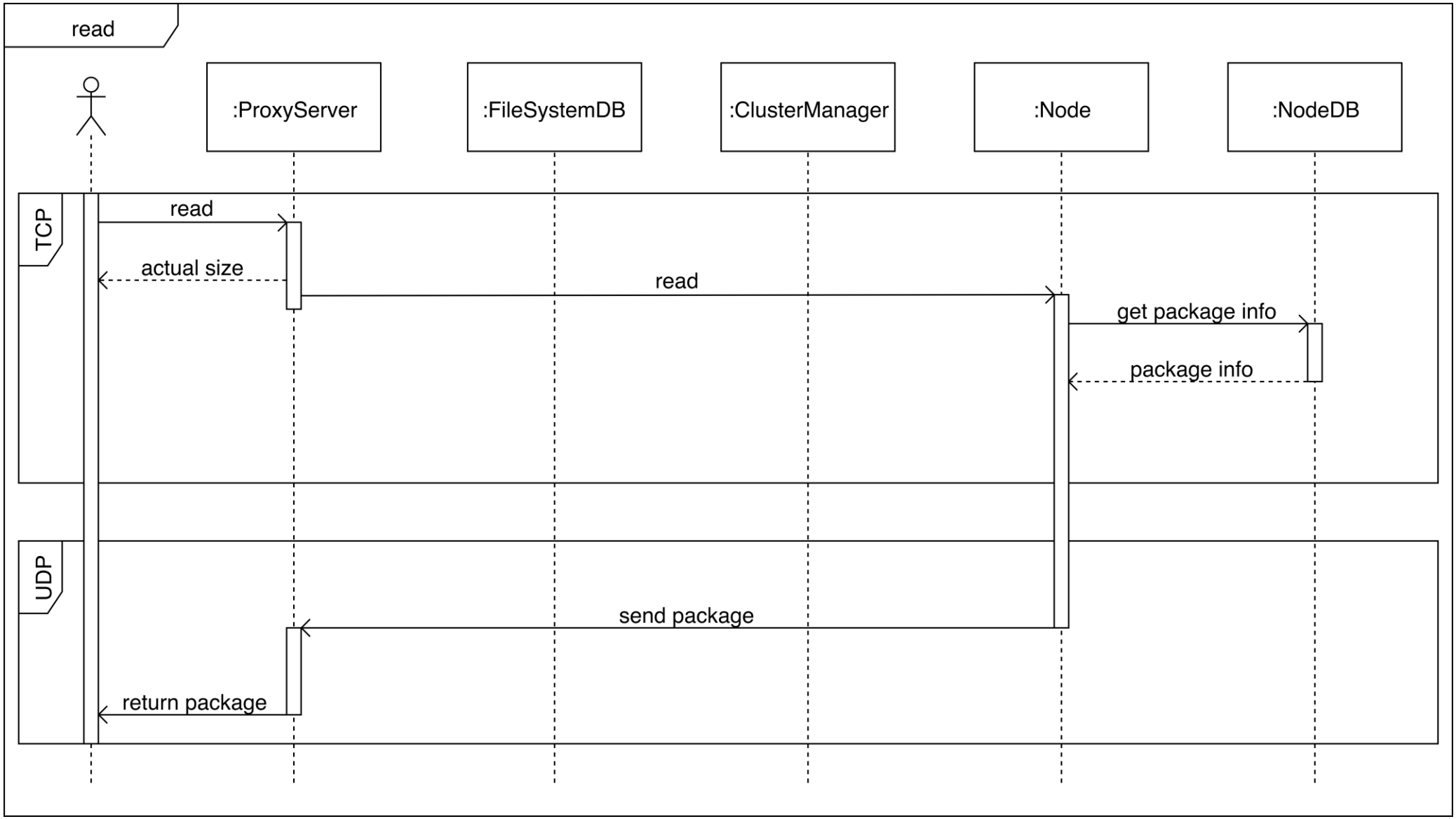
ДОДАТКИ

Додаток 1

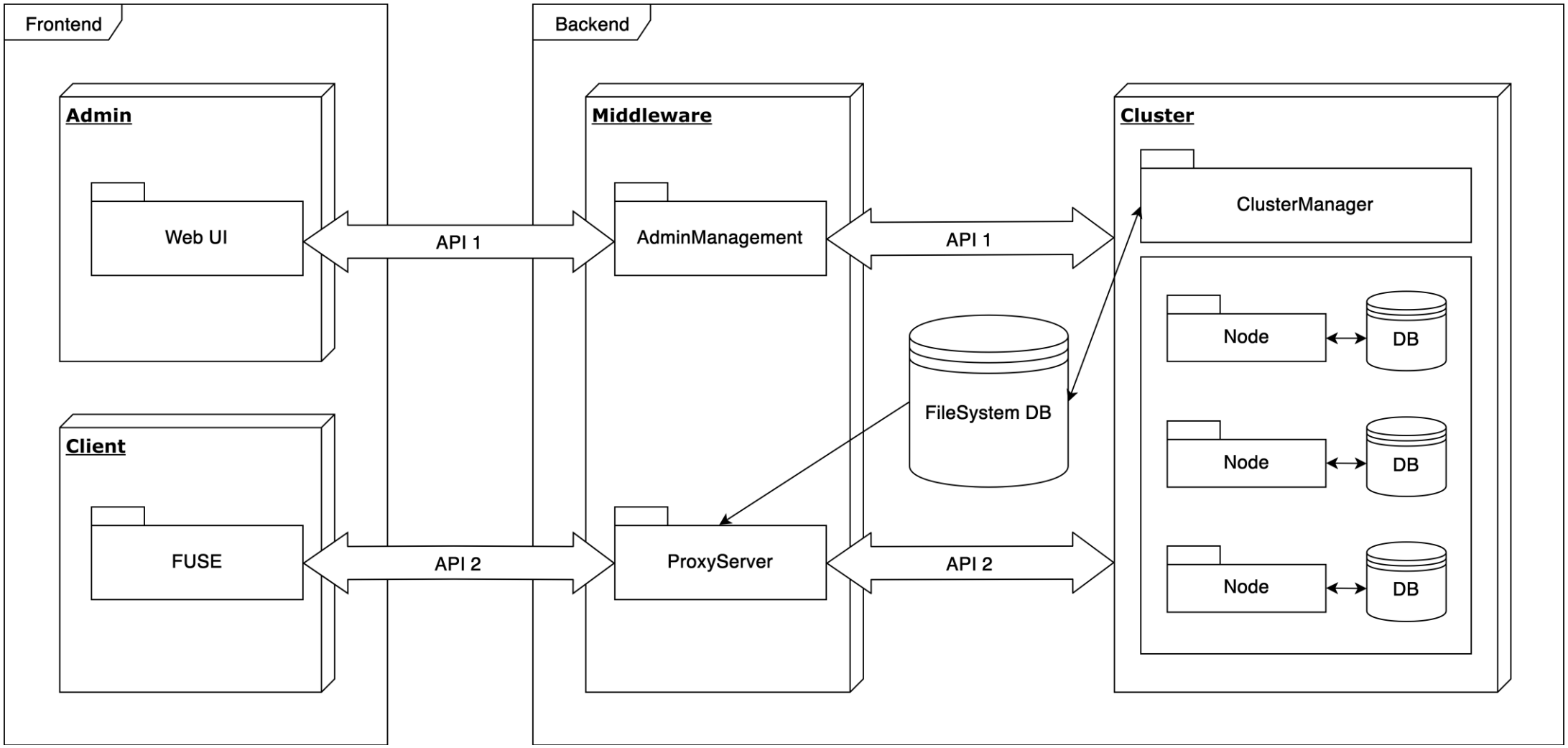
Копії графічних матеріалів



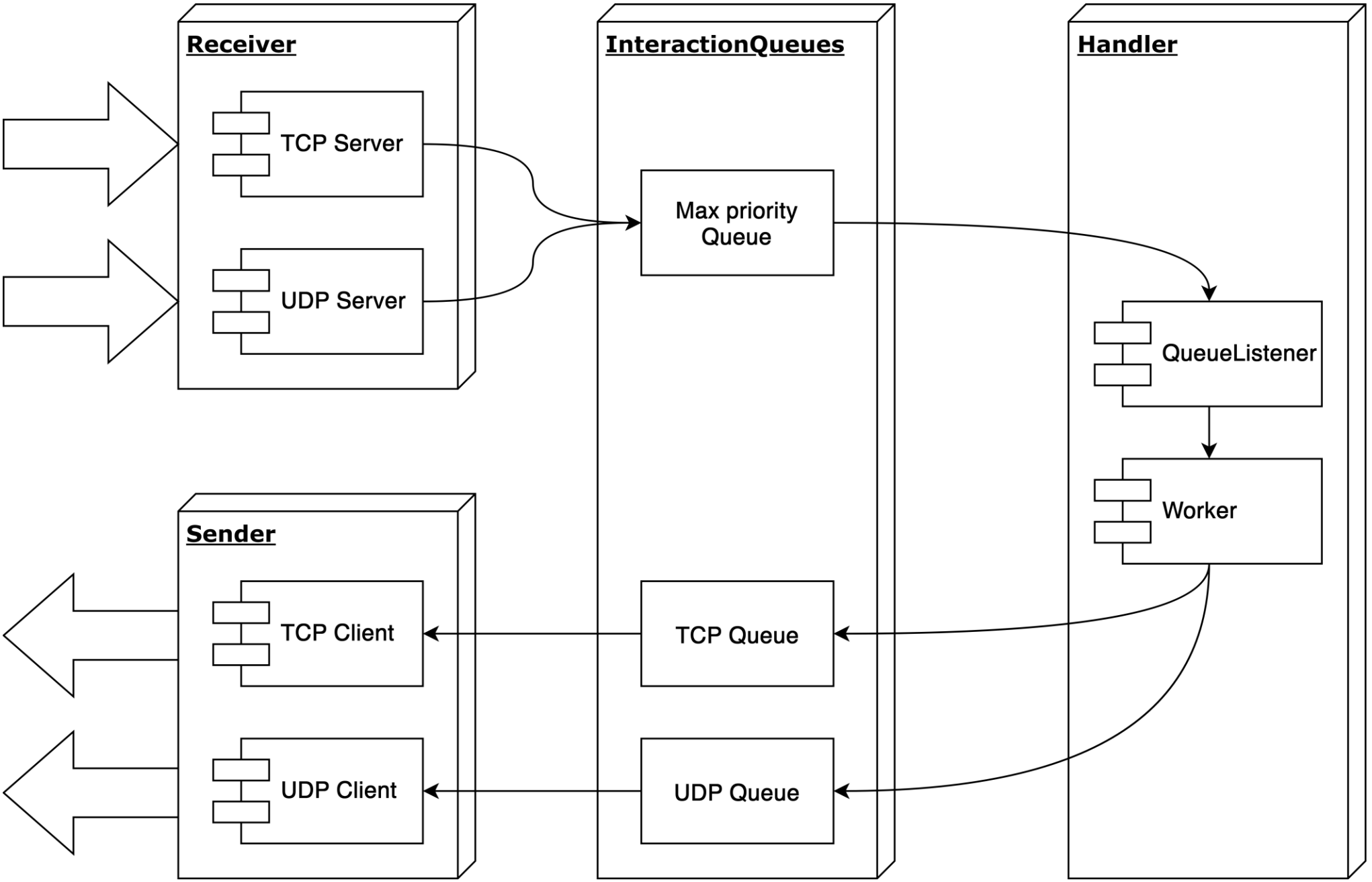
ДП.045200-06-99
Розподілена файлова система. Підсистема управління даними.
Схема роботи операції запису. Діаграма послідовностей



ДП.045200-07-99
Розподілена файлова система. Підсистема управління даними.
Схема роботи операції читання. Діаграма послідовностей



Захарченко Віталій Ігорович, група КП-51



Захарченко Віталій Ігорович, група КП-51

Додаток 2

Лістинг програм

utilities.py

```
import socket
import time
from queue import PriorityQueue, Queue
from threading import Thread, Lock, Event, Timer


class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args,
**kwargs)
        return cls._instances[cls]


class NamedSingleton(Singleton):
    def __call__(cls, name, *args, **kwargs):
        if name not in cls._instances:
            cls._instances[name] = super(Singleton, cls).__call__(name,
*args, **kwargs)
        return cls._instances[name]


class BaseQueue(object):
    def __init__(self):
        self.__queue = Queue()

    def insert(self, item):
        self.__queue.put(item)

    def remove(self):
        if not self.__queue.empty():
            return self.__queue.get()
        return None

    def empty(self):
        return self.__queue.empty()


class MaxPriorityItem(object):
    def __init__(self, item, priority):
        self.item = item
        self.priority = priority

    def __lt__(self, other):
        return self.priority > other.priority


class MaxPriorityQueue(BaseQueue):
    def __init__(self):
        self.__queue = PriorityQueue()

    def insert(self, item, priority=0):
        self.__queue.put(MaxPriorityItem(item, priority))

    def remove(self):
        if not self.__queue.empty():
            return self.__queue.get().item
        return None
```

```
class Socket(object):
    @staticmethod
    def create_tcp(blocking=True):
        new_socket = socket.socket(type=socket.SOCK_STREAM)
        new_socket.setblocking(blocking)
        return new_socket

    @staticmethod
    def create_udp():
        new_socket = socket.socket(type=socket.SOCK_DGRAM)
        return new_socket

    @staticmethod
    def create_and_bind_tcp(server_address, blocking=False):
        server_socket = Socket.create_tcp(blocking)
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server_socket.bind(server_address)
        return server_socket

    @staticmethod
    def create_and_bind_udp(server_address):
        server_socket = Socket.create_udp()
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server_socket.bind(server_address)
        return server_socket


class StoppedThread(Thread):
    def __init__(self, target=None, name=None, args=None, **kwargs):
        super().__init__(target=target, name=name, args=args, kwargs=kwargs)
        self.stopper = Event()

    def is_alive(self):
        return not self.stopper.is_set()

    def run(self):
        self._target(self.is_alive, *self._args, **self._kwargs)

    def start(self):
        super().start()
        print(self.name, "started")

    def stop(self):
        print(self.name, "stopping...")
        self.stopper.set()
        print(self.name, "stopped")


class TaskThread(StoppedThread):

    def __init__(self, target=None, name=None, delay=60, *args, **kwargs):
        super().__init__(target=target, name=name, args=args, kwargs=kwargs)
        self.lock = Lock()
        self.task_queue = []
        self.__delay = delay
        self.__timer = None
        self.__is_timer_set = Event()

    def run(self):
        while not self.stopper.is_set():

            if bool(self.task_queue):

                if self.__is_timer_set.is_set():
```

```
                    self.__timer_stop()
                    self.__is_timer_set.clear()

                data = self.__get_task()
                self._target(data, *self._args, **self._kwargs)

            elif not self.__is_timer_set.is_set():
                self.__is_timer_set.set()
                self.__timer = Timer(self.__delay, self.stop)
                self.__timer.start()

    def __get_task(self):
        self.lock.acquire()
        data = self.task_queue.pop(0)
        self.lock.release()
        return data

    def __timer_stop(self):
        self.__timer.cancel()

    def stop(self):
        super().stop()
        self.__timer_stop()
        print(self.name, "Sleeping...")

    def _restart(self):
        print("Waking up ...")
        self.stopper.clear()
        self.__is_timer_set.clear()
        self._started.clear()
        self.start()

    def add_task(self, data):
        self.lock.acquire()
        self.task_queue.insert(len(self.task_queue), data)
        self.lock.release()
        if self.stopper.is_set():
            self._restart()


class Cache(metaclass=NamedSingleton):
    def __init__(self, _):
        self.__cache_dict = {}
        self.__mutex = Lock()

    def __setitem__(self, key, item):
        self.__mutex.acquire()
        self.__cache_dict[key] = item
        self.__mutex.release()

    def __delitem__(self, key):
        self.__mutex.acquire()
        del self.__cache_dict[key]
        self.__mutex.release()

    def get(self, key, default=None):
        try:
            res = self.__getitem__(key)
        except KeyError:
            res = default

        return res

    def __getitem__(self, key):
```

```
        try:
            self.__mutex.acquire()
            res = self.__cache_dict[key]
            self.__mutex.release()
            return res
        except Exception as exp:
            print(exp)

    def __str__(self):
        return str(self.__cache_dict)

    def keys(self):
        return list(self.__cache_dict.keys())

    def values(self):
        return list(self.__cache_dict.values())

    def items(self):
        return list(self.__cache_dict.items())
```

ClusterManagerDispatcher.py

```
from copy import copy

from lib.ClusterManager.ClusterDispatcher.ClusterManagerMapper import
ClusterManagerMapper
from utils import Interaction, TaskThread, Cache, Config

CF = Config()


class ClusterManagerDispatcher(object):
    def __init__(self, **kwargs):
        super().__init__()
        self._request_inter = Interaction("receive")
        self._sender_inter = Interaction("sender")

        self._handlers = {
            "read": TaskThread(target=kwargs.get("read", self._read),
name="read"),
            "write": TaskThread(target=kwargs.get("write", self._write),
name="write"),
            "create": TaskThread(target=kwargs.get("create", self._create),
name="create"),
            "mkdir": TaskThread(target=kwargs.get("mkdir", self._mkdir),
name="mkdir"),
            "status": TaskThread(target=kwargs.get("status", self._status),
name="status"),
        }

        self._cache = Cache("cluster_manager")
        self._cache['node_load'] = {}

        self._mapper = ClusterManagerMapper()

    def dispatch(self, data, address):
        command, *payload = data.decode("utf-8").split('&')
        print(command, payload)
        handler = self._handlers.get(command, None)

        if handler:
            handler.add_task((payload, address))
        else:
            print("Bad command: ", command)

    def add_handler(self, key, handler_func):
        self._handlers[key] = TaskThread(target=handler_func, name=key)
        self._handlers[key].start()

    def start(self):
        for handler in self._handlers.values():
            handler.start()

    def stop(self):
        for handler in self._handlers.values():
            handler.stop()
            handler.join()

    def __recalc_load(self, node_id=None):
        if node_id:
            self._cache['node_load'][node_id] =
self._cache[node_id]['free_size'] * self._cache[node_id][
                'pack_idle_count']
        else:
```

```
            for key in self._cache.keys():
                self._cache['node_load'][key] = self._cache[key]['free_size']
* self._cache[key]['pack_idle_count']

    def __balance(self, package_count):
        res = []

        for _ in range(package_count):
            node_load = copy(self._cache['node_load'])
            node_to_add = min(node_load.items(), key=lambda item: item[1])[0]
            res.append(node_to_add)
            self._cache[node_to_add]['free_size'] -= int(CF.get("Package",
"data"))
            self._cache[node_to_add]['pack_idle_count'] += 1

        return res

    def __serialize_dict(self, dict_to_ser):
        key = []
        values = []

        for item in dict_to_ser.items():
            key.append(','.join(item[0]))
            node, package_id = item[1]
            values.append(','.join((str(node), str(package_id))))

        key = '|'.join(key)
        values = '|'.join(values)

        return "&".join((key, values))

    # FILE SYSTEM OPERATIONS

    def _read(self, data, *args, **kwargs):
        payload, address = data

        fd, pathname, max_package_count, offset = payload

        package_list = []

        packages = self._mapper.query('select_package_by_pathname', pathname)

        searched_next_pack_id = None

        for _ in range(len(packages)):

            for package in packages:
                package = list(package)
                next_pack_id = package[2]
                if not next_pack_id:
                    next_pack_id = None
                else:
                    next_pack_id = int(next_pack_id)
                if next_pack_id == searched_next_pack_id:
                    package[2] = str(len(package_list))
                    searched_next_pack_id = int(package[1])
                    package = map(lambda x: str(x), package)
                    package = ','.join(package)
                    package_list.insert(0, package)
                    break

        message = "&".join(("load", str(fd), *package_list))
```

```
        self._sender_inter.insert((message, (CF.get("Middleware", "ip"),
int(CF.get("Middleware", "port")))))

    def _write(self, data, *args, **kwargs):
        payload, address = data
        pathname, package_count = payload

        package_count = int(package_count)

        print("1")

        node_list = self.__balance(package_count)
        print("1.5")
        self._mapper.query("update_file_status", ("False", pathname))

        order_num, file_id = self._mapper.query("select_file_info",
pathname)[0]
        print("2")
        order_num = int(order_num)

        pack_id_list = []
        result_dict = {}

        for node in node_list:
            parent_id = self._mapper.query("select_parent_id", file_id)
            if parent_id:
                parent_id = int(parent_id[0][0])

            package_id = int(self._mapper.query("insert_package", (node,
file_id))[0][0])

            if parent_id:
                self._mapper.query("update_package", (package_id, parent_id))

            pack_id_list.append(package_id)
            result_dict[(pathname, str(order_num))] = (node, package_id)
            order_num += 1

        print("3")

        if order_num == package_count:
            self._mapper.query("update_file_data", (pack_id_list[0],
pathname))

        self._mapper.query("update_file_order_num", (order_num, pathname))

        message = "cache_add&" + self.__serialize_dict(result_dict)

        print("HERE: ", message, CF.get("Middleware", "ip"))

        self._sender_inter.insert((message, (CF.get("Middleware", "ip"),
int(CF.get("Middleware", "port")))))

    def _create(self, data, *args, **kwargs):

        payload, address = data

        pathname, response_ip, response_port = payload

        file_id = self._mapper.query("insert_file", pathname,
last_row_id=True)[0][0][0]

        dir_pathname = pathname.rsplit('/', 1)[0]
```

```
        if not dir_pathname:
            dir_pathname = '/'

        self._mapper.query("update_directory_data", (file_id, dir_pathname))

        request = "&".join(("open", pathname, response_ip, response_port))

        self._sender_inter.insert((request, (CF.get("Middleware", "ip"),
int(CF.get("Middleware", "port")))))

    def _mkdir(self, data, *args, **kwargs):

        payload, _ = data

        pathname = payload[0]

        file_id = self._mapper.query("insert_directory", pathname)[0][0]

        pathname = str(pathname)

        parent_dir_pathname = pathname.rsplit('/', 1)[0]

        if not parent_dir_pathname:
            parent_dir_pathname = '/'

        if parent_dir_pathname != pathname:
            self._mapper.query("update_directory_data", (file_id,
parent_dir_pathname))

    def _status(self, data, *args, **kwargs):
        payload, _ = data

        file_id, status = payload

        self._mapper.query("update_file_status_by_file_id", (status,
        file_id))
```

HealthDispatcher.py

```
from threading import Timer

from lib.HealthMonitor.HealthMonitor.HealthMapper import HealthMapper
from utils import Interaction, TaskThread, Cache, Config

CF = Config()


class HealthDispatcher(object):
    def __init__(self, **kwargs):
        super().__init__()
        self._request_inter = Interaction("receive")
        self._sender_inter = Interaction("sender")

        self._handlers = {
            "alive": TaskThread(target=kwargs.get("alive", self._alive),
name="alive"),
            "status": TaskThread(target=kwargs.get("status", self._status),
name="status"),
            "init": TaskThread(target=kwargs.get("init", self._init),
name="init"),
        }

        self._cache = Cache("cluster_manager")
        self._cache['node_load'] = {}

        self._mapper = HealthMapper()
        # self._cache['fd'] = {}

    def dispatch(self, data, address):
        command, *payload = data.decode("utf-8").split('&')
        print(command, payload)
        handler = self._handlers.get(command, None)

        if handler:
            handler.add_task((payload, address))
        else:
            print("Bad command: ", command)

    def add_handler(self, key, handler_func):
        self._handlers[key] = TaskThread(target=handler_func, name=key)
        self._handlers[key].start()

    def start(self):
        for handler in self._handlers.values():
            handler.start()

    def stop(self):
        for handler in self._handlers.values():
            handler.stop()
            handler.join()

    def _alive(self, data, *args, **kwargs):
        payload, _ = data
        node_id, status = payload

        node_id = int(node_id)

        if self._cache.get(node_id, None):
            if self._cache[node_id]['timer']:
                self._cache[node_id]['timer'].cancel()
```

```
        if status == "True":
            self._cache[node_id]['timer'] = Timer(int(CF.get("Node",
"node_live_timeout")), self._alive,
                                                  args=[((node_id, "False"),
())])
            self._cache[node_id]['timer'].start()
        elif self._cache.get(node_id, None):
            del self._cache[node_id]
            del self._cache['node_load'][node_id]

    def _status(self, data, *args, **kwargs):
        payload, _ = data
        new_size, node_id, pack_id, status = payload

        node_id = int(node_id)

        self._mapper.query("update_package_status", (status, pack_id))

        res, file_id = self._mapper.query("get_unready_package", (pack_id,
pack_id))

        res = res[0]
        file_id = file_id[0]

        self._mapper.query("update_file_size", (new_size, file_id))

        self._cache[node_id]['pack_idle_count'] -= 1
        self._cache['node_load'][node_id] =
self._cache[node_id]['pack_idle_count']*self._cache[node_id]['free_size']

        if not res:
            message = "&".join(("status", str(file_id), "True"))
            self._sender_inter.insert((message, (CF.get("Receiver", "ip"),
int(CF.get("Receiver", "tcp_port")))))

    def _init(self, data, *args, **kwargs):
        payload, address = data
        tcp_port, udp_port, free_size = payload

        res = self._mapper.query("get_node_by_address", (address[0],
int(tcp_port)))

        if not res:
            node_id = self._mapper.query("insert_new_node", (address[0],
tcp_port, udp_port))[0][0]
        else:
            node_id = int(res[0][0])

        res = "&".join(("init", str(node_id)))
        self._sender_inter.insert((res, (address[0], int(tcp_port))))

        self._cache[node_id] = {'timer': None,
                                'ip': address[0],
                                'tcp_port': int(tcp_port),
                                'udp_port': int(udp_port),
                                'free_size': int(free_size),
                                'pack_idle_count': 0}
        self._cache['node_load'][node_id] = 0
```

TCPHandler.py

```
from lib.TCPHandler.TCPMapper import TCPMapper
from utils import BaseHandler, Interaction, TaskThread, Cache, Config

CF = Config()

CLUSTER_MANAGER_ADDRESS = (CF.get("Cluster Manager", "ip"),
int(CF.get("Cluster Manager", "port")))


class TCPHandler(BaseHandler):

    def __init__(self, **kwargs):
        super().__init__()
        self._request_inter = Interaction("receive")
        self._tcp_sender_inter = Interaction("tcp_sender")
        self._udp_sender_inter = Interaction("udp_sender")
        self._handlers = {
            "open": TaskThread(target=kwargs.get("open", self._open),
name="open"),
            "flush": TaskThread(target=kwargs.get("flush", self._flush),
name="flush"),
            "read": TaskThread(target=kwargs.get("read", self._read),
name="read"),
            "write": TaskThread(target=kwargs.get("write", self._write),
name="write"),
            "getattr": TaskThread(target=kwargs.get("getattr",
self._getattr), name="getattr"),
            "readdir": TaskThread(target=kwargs.get("readdir",
self._readdir), name="readdir"),
            "create": TaskThread(target=kwargs.get("create", self._create),
name="create"),
            "mkdir": TaskThread(target=kwargs.get("mkdir", self._mkdir),
name="mkdir"),
            "load": TaskThread(target=kwargs.get("load", self._load),
name="load"),
            "cache_add": TaskThread(target=kwargs.get("cache_add",
self._cache_add), name="cache_add")
        }

        self._cache = Cache("middleware")

        self._cache['fd'] = {}
        self._cache['pathname'] = {}
        self._cache['package'] = {}
        self._cache['user'] = {}

        self._mapper = TCPMapper()

    def __create_fd(self):
        """0...127"""
        keys = self._cache['fd'].keys()

        for i in range(128):
            if i not in keys:
                return i

        return -1

    def execute(self, data, address):
        command, *payload = data.decode("utf-8").split('&')
        print(command, payload)
        handler = self._handlers.get(command, None)
```

```
        if handler:
            handler.add_task((payload, address))
        else:
            print("Bad command: ", command)

    def add_handler(self, key, handler_func):
        self._handlers[key] = TaskThread(target=handler_func, name=key)
        self._handlers[key].start()

    def start(self):
        for handler in self._handlers.values():
            handler.start()

    def stop(self):
        for handler in self._handlers.values():
            handler.stop()
            handler.join()

    # CACHE OPERATION
    def _cache_add(self, data, *args, **kwargs):
        payload, address = data

        keys, values = payload

        keys = keys.split('|')
        values = values.split('|')

        for key in keys:
            pathname, order_num = key.split(',')
            fd = int(self._cache['pathname'][pathname])
            order_num = int(order_num)
            self._cache['package'].update({(fd, order_num):
values.pop(0).split(',')})

    def _cache_del(self, data, *args, **kwargs):
        print("cache_del method: ", data)

    # FUSE OPERATION
    def _open(self, data, *args, **kwargs):

        payload, address = data
        pathname, *response_address = payload

        if len(response_address) == 2:
            response_ip = response_address[0]
            response_port = response_address[1]
        else:
            response_ip = address[0]
            response_port = response_address[0]

        res = self._mapper.query("get_file_by_pathname", pathname)[0]

        if not res:
            return self._tcp_sender_inter.insert(("0&File Doesn't Exists",
(response_ip, int(response_port))))

        file_type, size, order = res

        fd = self.__create_fd()

        # TODO add exception
        if fd == -1:
            return 0
```

```
        self._cache['fd'].update({fd: {'type': file_type, 'size': size,
'order': order, 'pathname': pathname}})
        self._cache['pathname'].update({pathname: fd})

        res = "&".join((str(1), str(fd), str(order)))

        self._tcp_sender_inter.insert((res, (response_ip,
int(response_port))))

    def _flush(self, data, *args, **kwargs):
        payload, address = data

        fd, response_port = payload

        pathname = self._cache['fd']['pathname']

        del self._cache['pathname'][pathname]
        del self._cache['fd'][int(fd)]

        # TODO add synchronization

        self._tcp_sender_inter.insert(('1', (address[0],
int(response_port))))

    def _getattr(self, data, *args, **kwargs):

        payload, address = data
        path_name, response_port = payload

        res = self._mapper.query("get_file_attr_by_pathname", path_name)
        if not res:
            return self._tcp_sender_inter.insert(("0&File Doesn't Exists",
(address[0], int(response_port))))

        file_type, size = res[0]

        res = "&".join((str(1), file_type, "777", str(size)))

        self._tcp_sender_inter.insert((res, (address[0],
int(response_port))))

    def _load(self, data, *args, **kwargs):
        payload, address = data

        fd = int(payload.pop(0))

        pack_count = len(payload)

        client_address = self._cache["user"][fd]

        message = "&".join(("1", str(pack_count)))

        self._tcp_sender_inter.insert((message, (client_address['ip'],
client_address['tcp_port'])))

        for value in payload:
            node_id, pack_id, order_num = value.split(",")
            ip, tcp_port = self._mapper.query("get_node_ip_tcp_port",
int(node_id))[0]

            self._cache["package"].update({ int(pack_id): int(order_num) })

            message = "&".join(("read", pack_id))
```

```
            self._tcp_sender_inter.insert((message, (ip, int(tcp_port))))

    def _read(self, data, *args, **kwargs):
        payload, address = data
        fd, max_package_count, offset, udp_port, tcp_port = payload
        fd = int(fd)
        self._cache["user"].update({ fd:  {"ip": address[0], "tcp_port":
int(tcp_port), "udp_port": int(udp_port)} })

        pathname = self._cache['fd'].get(fd, None)

        status = self._mapper.query("get_file_status",
pathname['pathname'])[0]

        if not status or status == "False":
            self._tcp_sender_inter.insert(("0&File isn't ready", (address[0],
int(tcp_port))))

        message = "&".join(('read', str(fd), pathname['pathname'],
str(max_package_count), str(offset)))

        self._tcp_sender_inter.insert((message, CLUSTER_MANAGER_ADDRESS))

    def _write(self, data, *args, **kwargs):

        payload, address = data

        fd, package_count, _ = payload

        pathname = self._cache['fd'][int(fd)]['pathname']

        # TODO add exists check

        # res = self._mapper.query("get_file_by_pathname", pathname)
        #
        # if not res:
        #     return self._tcp_sender_inter.insert(("0&File Doesn't Exists",
(address[0], int(response_port))))

        self._tcp_sender_inter.insert(("&".join(("write", pathname,
package_count)), CLUSTER_MANAGER_ADDRESS))

    def _readdir(self, data, *args, **kwargs):
        payload, address = data
        path_name, response_port = payload

        res = self._mapper.query("get_direct_child", path_name)

        if not res:
            return self._tcp_sender_inter.insert(("0&Directory Doesn't
Exists", (address[0], int(response_port))))

        res = [element[0].rsplit('/', 1)[1] for element in res]

        res = "&".join((str(1), *res))

        self._tcp_sender_inter.insert((res, (address[0],
int(response_port))))

    def _create(self, data, *args, **kwargs):

        payload, address = data
        pathname, response_port = payload
```

```
        if self._mapper.query('get_file_attr_by_pathname', pathname):
            return self._tcp_sender_inter.insert(("0&File Already Exists",
(address[0], int(response_port))))
        else:
            request = "&".join(("create", pathname, address[0],
response_port))
            self._tcp_sender_inter.insert((request, CLUSTER_MANAGER_ADDRESS))

    def _mkdir(self, data, *args, **kwargs):
        payload, address = data
        pathname, response_port = payload

        if self._mapper.query('get_file_attr_by_pathname', pathname):
            return self._tcp_sender_inter.insert(("0&Directory Already
Exists", (address[0], int(response_port))))
        else:
            request = "&".join(("mkdir", pathname))
            self._tcp_sender_inter.insert((request, CLUSTER_MANAGER_ADDRESS))
            return self._tcp_sender_inter.insert(('1', (address[0],
int(response_port))))
```

UDPHandler.py

```
import struct
from functools import reduce

from lib.UDPMapper import UDPMapper
from utils import BaseHandler, Cache, TaskThread, Interaction, Config

CF = Config()


def wrap(payload):
    format, value = payload
    return struct.pack(format, value)


def pack(*args):
    res = map(lambda x: wrap(x), args)
    return reduce(lambda x, y: x + y, res)


class UDPHandler(BaseHandler):

    def __init__(self, **kwargs):
        super().__init__()
        self._request_inter = Interaction("receive")
        self._udp_sender_inter = Interaction("udp_sender")
        self._cache = Cache("middleware")

        self._cache['fd'] = {}
        self._cache['pathname'] = {}
        self._cache['package'] = {}
        self._cache['user'] = {}

        self._mapper = UDPMapper()

        self._handlers = {"udp_read": TaskThread(target=kwargs.get("read",
self._read), name="read"),
                          "udp_write": TaskThread(target=kwargs.get("write",
self._write), name="write")}

    def execute(self, data, address):

        fd = struct.unpack("i", data[:4])[0]

        if fd == -1:
            self._handlers['udp_read'].add_task((data, address))
        else:
            self._handlers['udp_write'].add_task((data, address))

    def start(self):
        for handler in self._handlers.values():
            handler.start()

    def stop(self):
        for handler in self._handlers.values():
            handler.stop()
            handler.join()

    def _write(self, data, *args, **kwargs):
        data, address = data

        fd = struct.unpack("i", data[:4])[0]
        number = struct.unpack("I", data[4:8])[0]
```

```
        data_size = struct.unpack("I", data[8:12])[0]

        node_id, package_id = self._cache['package'].get((fd, number), (None,
None))

        if node_id:

            del self._cache['package'][(fd, number)]

            buffer = pack(("i", -1), ("I", int(package_id)), ("I",
data_size), (CF.get("Package", "data") + "s", data[12: -2]), ("H", 0))

            ip, udp_port = self._mapper.query("get_node_address_by_node_id",
node_id)[0]

            self._udp_sender_inter.insert((buffer, (ip, int(udp_port))))
        else:
            self._request_inter.insert(("udp", data, address), 0)

    def _read(self, data, *args, **kwargs):
        data, _ = data

        pack_id = struct.unpack("I", data[4:8])[0]

        pathname = self._mapper.query("get_pathname_by_pack_id",
pack_id)[0][0]

        fd = int(self._cache["pathname"][pathname])

        order_num = self._cache['package'].get(pack_id, -1)

        if order_num != -1:
            del self._cache['package'][pack_id]

        data = pack(("i", fd), ("I", order_num)) + data[8:-2] + pack(('H',
0))

        client_address = self._cache['user'].get(fd, None)

        if client_address:
            # if
                # del self._cache['user'][fd]
            self._udp_sender_inter.insert((data, (client_address['ip'],
client_address['udp_port'])))
```

Storage.py

```
import os
import struct
from threading import Timer
from functools import reduce

from lib.Storage import StorageMapper
from utils import Config, BaseHandler, TaskThread, Interaction

CF = Config()


def wrap(payload):
    format, value = payload
    return struct.pack(format, value)


def pack(*args):
    res = map(lambda x: wrap(x), args)
    return reduce(lambda x, y: x + y, res)


class Storage(BaseHandler):

    def __init__(self, **kwargs):
        super().__init__()

        self.__tcp_sender = Interaction("tcp_sender")
        self.__udo_sender = Interaction("udp_sender")

        self._mapper = StorageMapper()

        self._block_size = int(CF.get("Storage", "block_size"))
        self._filed_memory = 0
        self._size = int(CF.get("Storage", "size"))
        self._seek = 0

        self._node_id = None

        if os.path.isfile(CF.get("Storage", "file_path")):
            self._file = open(CF.get("Storage", "file_path"), 'r+b')
            self._filed_memory =
int(self._mapper.query("select_filled_package")[0][0])
        else:
            self._file = open(CF.get("Storage", "file_path"), 'w+b')
            self._file.seek(self._size)
            self._file.write(b'\n')
            self._file.flush()
            print("Initial Load...")
            for i in range(self._size // self._block_size):
                print(str(100 * i / self._size * self._block_size) + "%")
                self._mapper.query("initial_insert", i)

        self._handlers = {"read": TaskThread(target=kwargs.get("read",
self.tcp_read), name="read"),
                          "write": TaskThread(target=kwargs.get("write",
self.udp_write), name="write"),
                          "init": TaskThread(target=kwargs.get("init",
self._init), name="init")}

    def __del__(self):
        self._file.close()
```

```
    def execute(self, data, address):
        try:
            command, *payload = data.decode("utf-8").split('&')
        except UnicodeDecodeError as e:
            payload = data
            command = "write"

        print(command, payload)

        # if not payload:
        #     payload = command
        #     command = "write"
        handler = self._handlers.get(command, None)

        if handler:
            handler.add_task((payload, address))
        else:
            print("Bad command: ", command)

    def start(self):
        for handler in self._handlers.values():
            handler.start()

        message = "&".join(("init", CF.get("Receiver", "tcp_port"),
CF.get("Receiver", "udp_port"),
                            str(self._size - self._filed_memory)))

        self.__tcp_sender.insert((message, (CF.get("HealthMonitor", "ip"),
int(CF.get("HealthMonitor", "port")))))

    def stop(self):
        for handler in self._handlers.values():
            handler.stop()
            handler.join()

    def tcp_read(self, data, *args, **kwargs):
        payload, address = data

        package_id = payload[0]

        result = self._mapper.query("get_package", package_id)

        if result:
            block_offset, inblock_offset, real_size = result[0]
            self._read(package_id, buffer_size=real_size,
block_offset=block_offset)
        else:
            # TODO add exception
            pass

    def _read(self, pack_id, buffer_size=int(CF.get("Storage",
"block_size")), block_offset=0, in_block_offset=0):
        print(block_offset, self._block_size + in_block_offset)
        print(block_offset * self._block_size + in_block_offset)
        self._file.seek(block_offset * self._block_size + in_block_offset)
        buffer = self._file.read(buffer_size - in_block_offset)
        buffer = pack(("i", -1), ("I", int(pack_id)), ("I",
int(buffer_size)), (str(self._block_size) + "s", buffer), ("H", 0))  # TODO
add checksum

        self.__udo_sender.insert((buffer, (CF.get("Middleware", "ip"),
int(CF.get("Middleware", "udp_"

"port")))))
```

```
    def udp_write(self, data, *args, **kwargs):
        payload, address = data

        package = payload

        package_id = struct.unpack("I", package[4:8])[0]
        size = struct.unpack("I", package[8:12])[0]
        data = package[12:12 + size]

        free_pack = self._mapper.query("get_free_package")

        if free_pack:
            id, block_offset, inblock_offset = free_pack[0]
            self._write(data, block_offset, inblock_offset)
            self._mapper.query("update_package", (package_id, 0, size, id))

            message = "&".join(("status", str(size), str(self._node_id),
str(package_id), "True"))

            self.__tcp_sender.insert((message, (CF.get("HealthMonitor",
"ip"), int(CF.get("HealthMonitor", "port")))))
        else:
            # TODO add else statement and handler
            pass

    def _write(self, data_to_write, block_offset=0, in_block_offset=0):
        self._file.seek(block_offset * self._block_size + in_block_offset)
        self._filed_memory += len(data_to_write)
        self._file.write(data_to_write)
        self._file.flush()

    def __is_alive_request(self):
        self.__tcp_sender.insert(
            ("alive&" + str(self._node_id) + "&True",
(CF.get("HealthMonitor", "ip"), int(CF.get("HealthMonitor", "port")))))
        Timer(int(CF.get("HealthMonitor", "repeat_timeout")),
self.__is_alive_request).start()

    def _init(self, data, *args, **kwargs):
        payload, _ = data
        self._node_id = int(payload[0])

        self.__is_alive_request()
```

Додаток 3

Копія презентації

Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Розподілена файлова система. Підсистема управління даними

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

Київ 2019

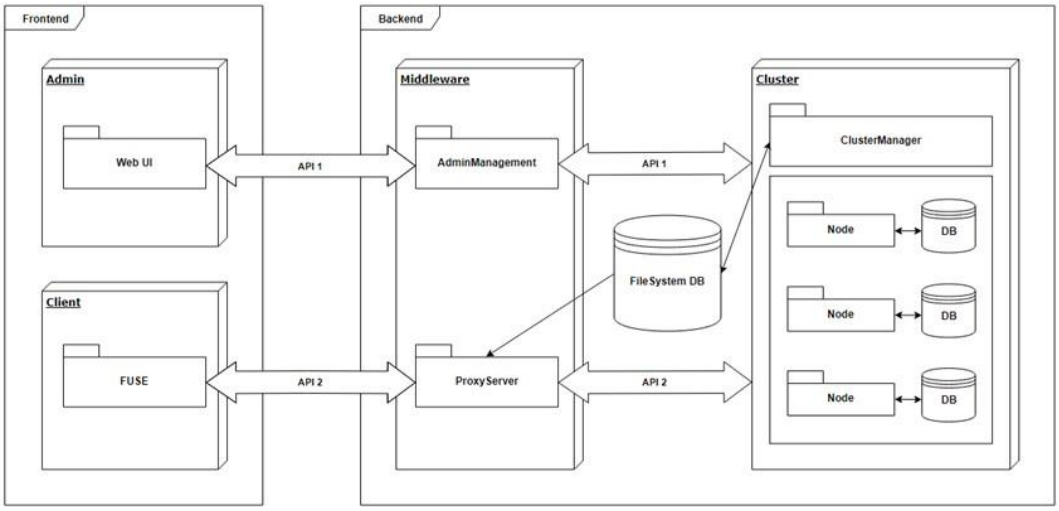
Постановка задачі

- Розроблення серверної частини розподіленої файлової системи
  - Реалізація серверу для взаємодії з кластером
  - Реалізація кластер-менеджера та нод
  - Проектування бази даних

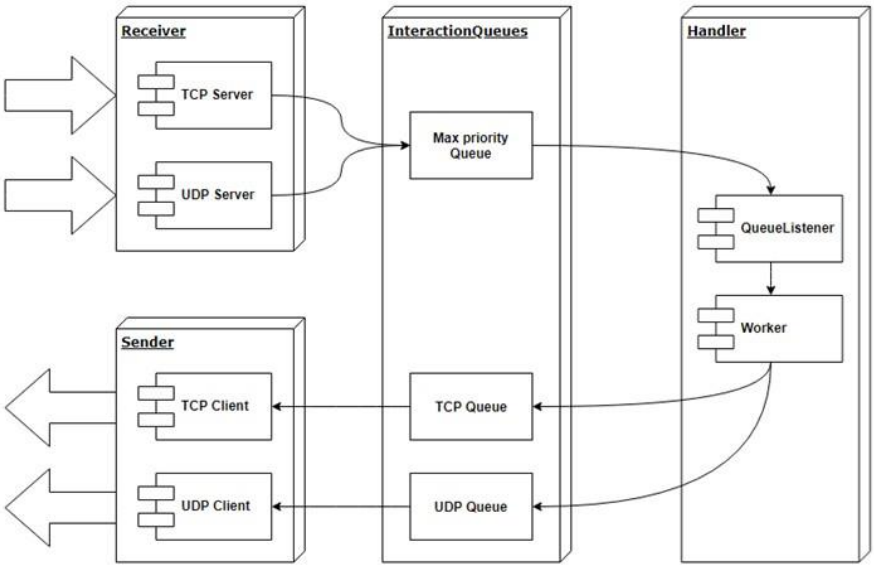
Аналіз існуючих рішень

- Ceph
  - недоліки: неможливість створення декількох файлових систем
- GlusterFS
  - недоліки: кожен обліковий запис повинен відображатися на об’ємі пам’яті файлової системи
- HDFS
  - недоліки: не допускається внесення жодних змін до даних, які зберігаються в системі HDFS

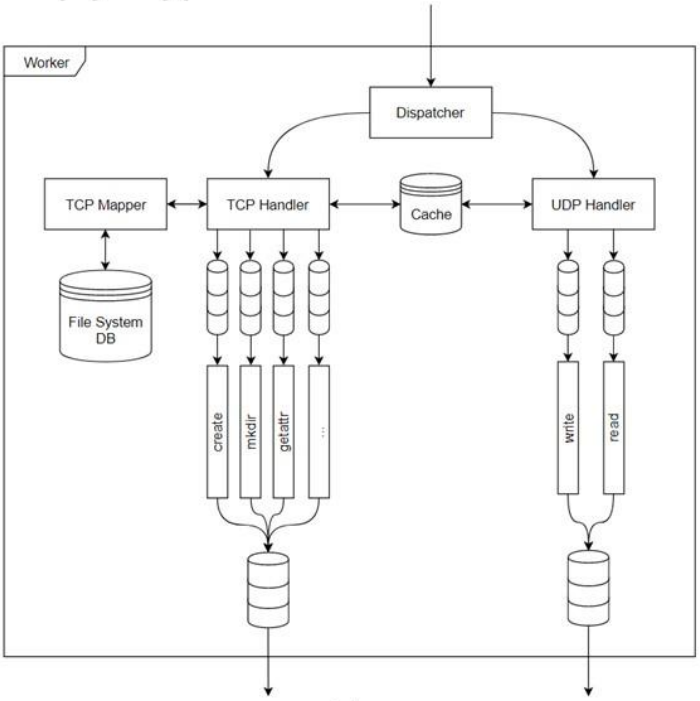
Структура розподіленої файлової системи



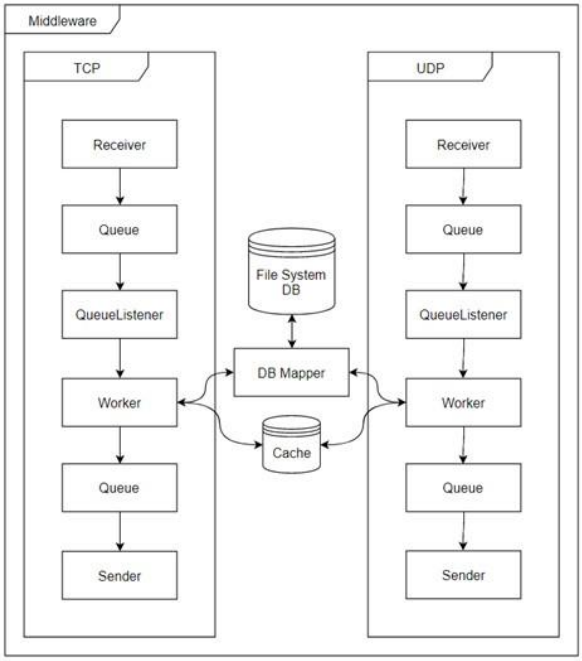
Структура передачі даних



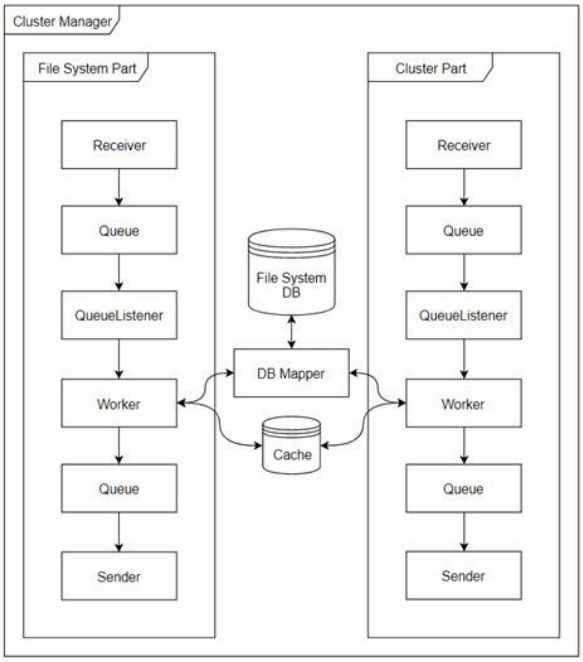
Структура компонента Worker



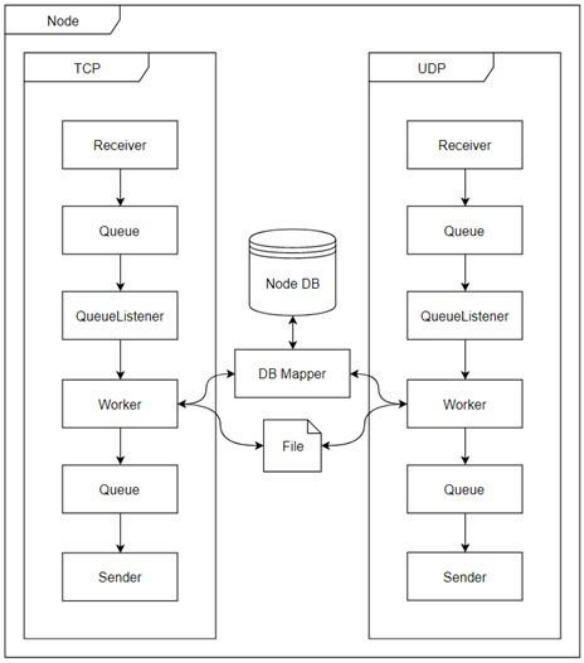
Структура Middleware



Структура Cluster Manager

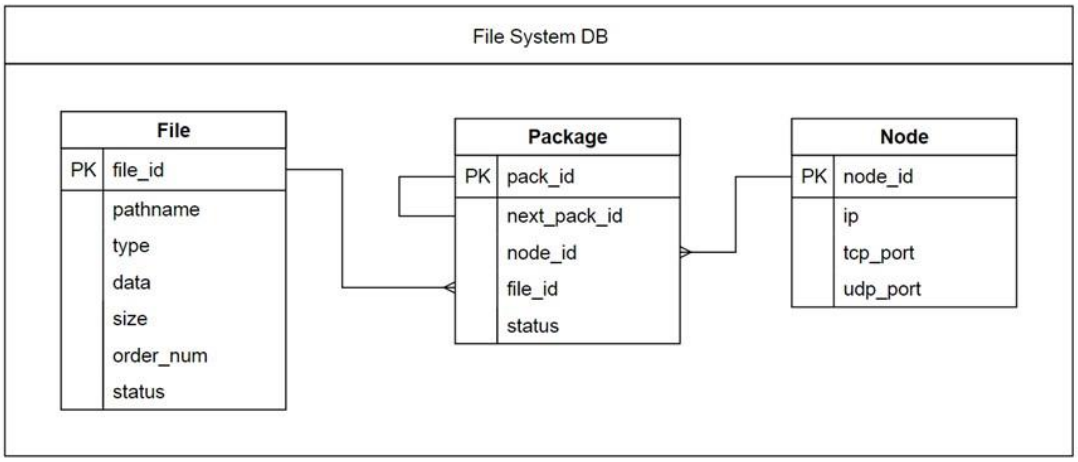


Структура Cluster Node

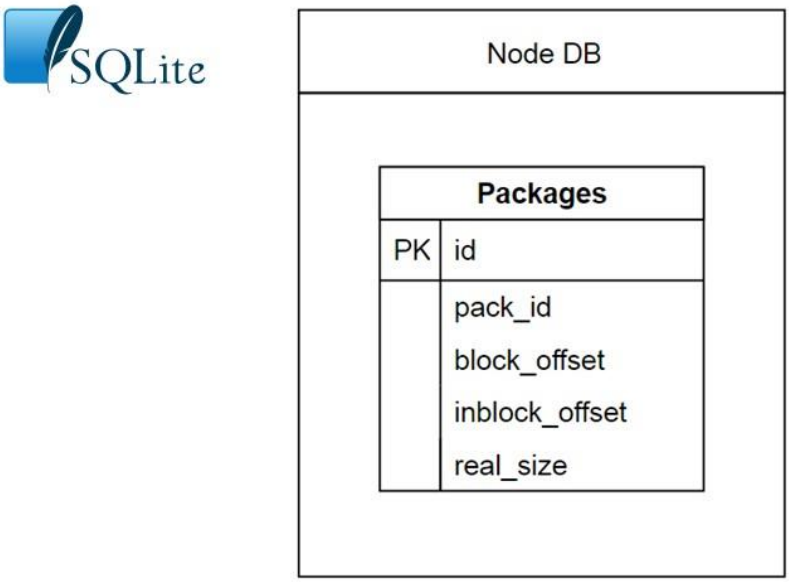


База даних файлової системи

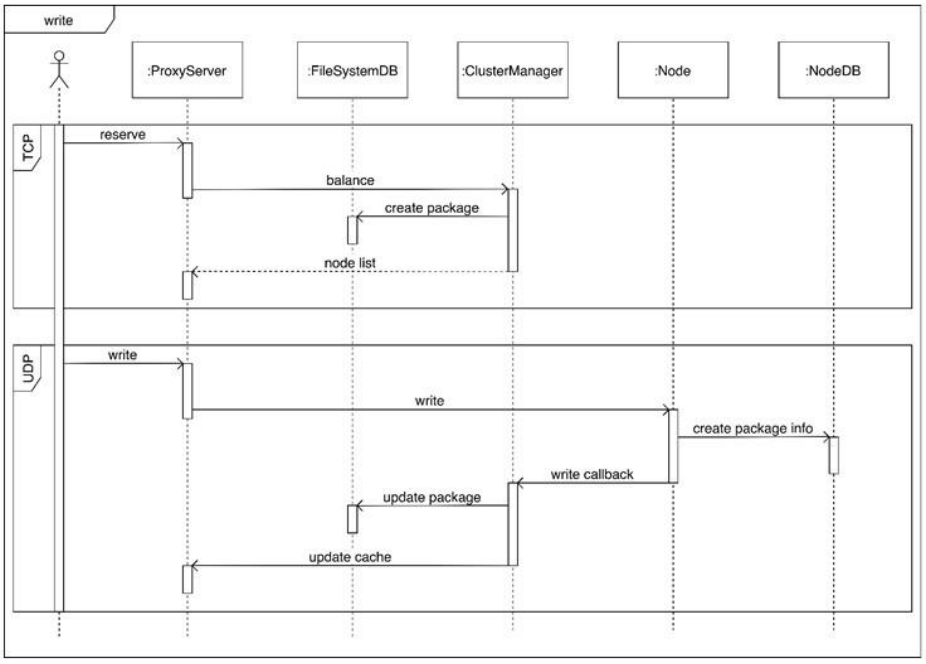
PostgreSQL



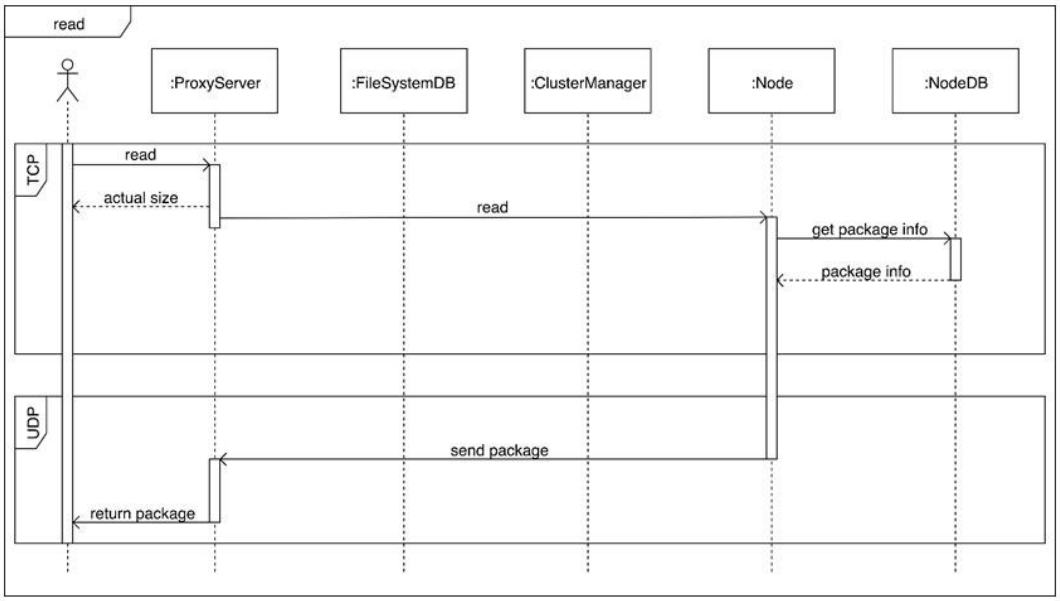
База даних ноди



Операція запису даних



Операція зчитування даних



Отримані результати

Можливі шляхи подальшого вдосконалення

- Розширення функціональних можливостей системи
- Перехід від TCP до TLS
- Шифрування даних
- Модернізація механізму балансування навантаження
- UDP/TCP Patching

Висновки

- Розроблено підсистему управління даними розподіленої файлової системи, яка 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 УПРАВЛІННЯ ДАНИМИ

Програма та методика тестування

ДП.045200-04-51

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.І. Захарченко

2018

ЗМІСТ

1. ОБ’ЄКТ ВИПРОБУВАНЬ

Об’єктом випробувань є підсистема управління даними розподіленої файлової системи, створена за допомогою мови програмування Python, з використанням реляційних баз даних: PostgreSQL та SQLite.

2. МЕТА ТЕСТУВАННЯ

Метою тестування обрано перевірку роботи підсистеми управління даними, а саме:

1) функціональна працездатність;
2) коректне збереження пакету даних;
3) коректна робота балансувальника навантаження;
4) коректна робота механізму передачі даних;
5) відповідність вимогам Технічного завдання.

3. МЕТОДИ ТЕСТУВАННЯ

Для тестування обрано метод Black Box Testing. Перевіряється відповідність самого програмного продукту 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 підсистеми управління даними виконується засобом динамічного ручного тестування.

Працездатність перевіряється шляхом перевірки роботи усіх функціональних можливостей взаємодії користувача з підсистемою управління даними, а саме:

1) створення нового файлу;
2) запису даних в існуючий файл;
3) зчитування частини файлу;
4) зчитування всього файлу;
5) надання інформації про вміст директорії;
6) закриття файлу.

Крім того відбувається тестування сервера керування кластеру на предмет правильності обробки команд кластера, а саме:

1) додавання нового вузла в кластер;
2) балансування навантаження на кластер (по мінімальному навантаженню та вільному місцю);
3) перевірка стану вузла;
4) перевірка автоматичного виключення вузла з кластера у випадку непрацездатності останнього.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ І.А. Дичка

«___» _____ 2019 р.

РОЗПОДІЛЕНА ФАЙЛОВА СИСТЕМА. ПІДСИСТЕМА УПРАВЛІННЯ ДАНИМИ

Керівництво користувача

ДП.045200-05-34

«ПОГОДЖЕНО»

Керівник проекту:

_____ Ю.В. Вунтесмері

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ В.І. Захарченко

2019

ЗМІСТ

1. Опис структури підсистеми управління даними

Підсистема управління даними призначена для отримання, обробки та формування відповіді на команди користувача. Структурно підсистема складається з трьох складових: кластера, сервера управління кластером (Cluster Manager) та проксі сервера. Кожна з вищеназваних частин є незалежною від інших і можуть розміщуватись на різних фізичних пристроях. Крім того використовуються дві реляційні бази даних: PostgreSQL та SQLite. Перша, для зберігання інформації про структуру файлової системи та метадані системи. Друга використовується на вузлах кластеру для збереження метаданих самого вузла.

2. Запуск підсистеми управління даними

Так як підсистема управління даними складається з декількох незалежних одна від одної частин, кожна з яких потребує налаштування перед запуском, то централізованого запуску всієї підсистеми не існує. Для запуску підсистеми необхідно запустити кожну її складову в конкретному порядку, завчасно змінивши деякі налаштування в відповідному файлі. Процедура старту роботи підсистеми складається з наступних кроків:

1. Запуск бази даних файлової системи (PostgreSQL). Також на цьому кроці можна імпортувати в базу структуру таблиць з відповідного файлу (schema.sql), але цей крок є опціональний, адже відповідні команди виконуються при старті всіх частин підсистеми (вони створюють необхідні для них таблиці).
2. Редагування налаштувань проксі-сервера та сервера керування кластером з заміною полів в частині, що відповідає за базу даних. До таких полів відносяться: host, port, database, user, password, schema. Перші два параметри необхідні для підключення до бази даних в мережі інтернет. Параметри з третього до 5 включно, необхідні для авторизації в базі даних. Останній параметр

використовується для опису необхідних таблиць і зав’язків в базі даних.

3. Запуск проксі-сервера та сервера керування кластером. Ці сервера є незалежними один від одного, тому можуть бути запущені в будь-якому порядку.
4. Запуск вузлів кластера, з попереднім редагуванням налаштувань в полях Cluster_Manager.host, Cluster_Manager.port. Цей пункт повторюється для кожного окремого вузла кластера.

3. Опис можливостей файлової системи

Файлова система надає можливість користувачу зберігати та зчитувати дані файлу. Для цього підсистема управління даними надає функціонал для таких операцій:

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 функцій обробників необхідно, щоб користувач надіслав відповідні запити, після чого команди будуть додані в сортувальник, а далі в чергу виконання.

4. Вимкнення підсистеми управління даними

Для завершення роботи підсистеми управління даними необхідно повторити зворотній до запуску порядок дій (окрім пунктів зміни налаштувань), тобто необхідно вимкнути кластер (всі його вузли), потім сервер керування кластером, проксі сервер та базу даних.