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

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

Галузь знань 12 – Інформаційні технології

Спеціальність 123 –Комп'ютерна інженерія

на тему «Система зберігання даних на основі OpenStack Object Storage»

КвРКІП. 2303194.23.03.04 ПЗ

Виконав: студент 2 курсу, група КІ2м-23-3 Сергій ЖАРЧИНСЬКИЙ
Підпис Ім'я, прізвище

Керівник д-р. техн. наук, професор Василь ЯЦКІВ
Науковий ступінь, вчене звання Підпис Ім'я, прізвище

До захисту допускаю:
Зав. кафедри КІПС, доктор філософії, доцент
Ольга ПАВЛОВА
19 05 2025 р.

Хмельницький, 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Освітній рівень магістр

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма Освітньо-наукова програма «комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ

Зав. кафедри Ольга ПАВЛОВА

"01" 09 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Сергію ЖАРЧИНСЬКОМУ

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Система зберігання даних на основі OpenStack Object Storage

Керівник проекту (роботи) Василь ЯЦКІВ, д.т.н., професор

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 08.01.2025 №8

2. Строк подання студентом проекту (роботи) на кафедру 01.05.2025 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

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

Аналіз архітектури OpenStack Swift, кільцевої структури для розподілу даних.

Дослідження принципів масштабування та забезпечення довговічності даних.

Методи оптимізації продуктивності OpenStack Swift.

Розробка і демонстрація прикладу для реальних задач зберігання великих обсягів даних.

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

6. Консультанти розділів кваліфікаційної роботи магістра

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Сергій ЛИСЕНКО, професор кафедри КПС		
Антиплагіат	Андрій НІЧЕПОРУК, доцент кафедри КПС		

7. Дата видачі завдання «_01_»_____09_____2024р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) кваліфікаційної роботи магістра	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики КвРМ з керівником	01.09.2024	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.10.2024	виконано
3	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	01.11.2024	виконан
4	Робота над розділом 2 – розробка моделей для вирішення поставленої задачі	01.12.2024	виконан
5	Робота над науковою статтею	01.02.2025	виконан
6	Робота над розділом 3 – розробка методів для вирішення поставленої задачі	15.02.2025	виконан
7	Робота над розділом 4 – проектування та розробка ПЗ для вирішення поставленої задачі, експериментальна частина	01.04.2025	виконан
8	Оформлення пояснювальної записки згідно вимог	18.04.2025	викона
9	Попередній захист ДРМ	29.04.2025	викона
10	Захист ДРМ на засіданні ЕК	До 15.05.2025	

Студент _____ Сергій ЖАРЧИНСЬКИЙ
Підпис Ім'я, прізвище

Керівник роботи _____ Василь ЯЦКІВ
Підпис Ім'я, прізвище

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: «Система зберігання даних на основі OpenStack Object Storage»

Автор роботи: Сергій ЖАРЧИНСЬКИЙ

Керівник роботи: д.т.н., професор Василь ЯЦКІВ

Пояснювальна записка: 74 с., 13 рис., 8 табл., 2 дод., 40 джерел.

OPENSTACK SWIFT, СИСТЕМА ЗБЕРІГАННЯ ДАНИХ, РЕПЛІКАЦІЯ, РОЗПОДІЛЕНЕ СХОВИЩЕ, ХМАРНА ІНФРАСТРУКТУРА, ОБ'ЄКТНЕ СХОВИЩЕ, МАСШТАБОВАНІСТЬ, ВІДМОВОСТІЙКІСТЬ

Об'єктом дослідження є процеси зберігання даних з використанням OpenStack Swift.

Предметом дослідження є алгоритми та механізми організації, управління та забезпечення надійності зберігання даних у OpenStack Object Storage.

Метою кваліфікаційної роботи магістра є побудова системи зберігання даних на основі OpenStack Object Storage.

Для розв'язання поставлених задач використовувалися методи порівняльного аналізу, моделювання, архітектурного проєктування, симуляційного тестування та практичного впровадження.

Наукова новизна отриманих результатів:

− набув подальшого розвитку метод організації зберігання даних з використанням модифікованого хешування у кластерних середовищах;

− набула подальшого розвитку інформаційна технологія об'єктного зберігання з використанням OpenStack Swift.

На основі проведених досліджень розроблена архітектура і компоненти програмного забезпечення для розгортання надійної, масштабованої та високопродуктивної системи зберігання даних з інтеграцією до хмарної платформи OpenStack.

Практична значимість отриманих результатів полягає у створенні економічно доцільної альтернативи комерційним СЗД з високим рівнем продуктивності та відмовостійкості, яка може бути використана в освітніх, наукових і комерційних дата-центрах.

Розділ 1. Проведено огляд предметної області, аналіз існуючих рішень і визначено вимоги до СЗД.

Розділ 2. Побудовано теоретичну модель архітектури системи на основі Swift, описано особливості масштабування, реплікації та узгодженості.

Розділ 3. Розроблено алгоритми доступу до даних, архітектуру кластера, описано реалізацію високої пропускної здатності.

Розділ 4. Проведено практичну реалізацію системи, її конфігурацію та автоматизацію розгортання за допомогою Kolla-ansible, проведено тестування й оптимізацію.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

FUSE (filesystem in userspace) – модуль для ядер UNIX - подібних операційних систем з відкритим вихідним кодом, відноситься до вільного програмного забезпечення, дозволяє користувачам без привілеїв створювати їх власні файлові системи без необхідності переписувати код ядра

HPC (High Performance Computing) – високопродуктивні обчислення

POSIX (Portable Operating System Interface for uniX) – набір стандартів, що описують інтерфейси між операційною системою і прикладною програмою (системний API), бібліотеку мови С, набір додатків і їх інтерфейсів

UUID (User Unique IDentifier) – унікальний ідентифікатор користувача.

БД – база даних

ПЗ – програмне забезпечення

РСЗД – розподілена система зберігання даних

СЗД – система зберігання даних

СУБД – система управління базами даних

ФС – файлова система

БРК – багато регіональний кластер

API (Application Programming Interface) – інтерфейс прикладного програмування

IaaS (Infrastructure as a Service) – інфраструктура як сервіс

QoS (Quality of Service) – якість обслуговування, використовується в контексті управління продуктивністю і пріоритетами доступу до ресурсів сховища.

CLI (Command Line Interface) – інтерфейс командного рядка

RDMA (Remote Direct Memory Access) – віддалений прямий доступ до пам’яті

ВСТУП

У сучасну епоху вимоги до систем зберігання даних зростають експоненційно. Користувачі створюють і споживають більше даних, ніж будь-коли раніше. Соціальні мережі, онлайн-відео, користувацький контент, ігрові сервіси – усі ці фактори сприяють зростанню швидкого та легкого доступу до сховищ даних, а також, що можуть масштабуватися без обмежень. Різноманітні компанії та установи стикаються з дедалі більшими вимогами до зберігання інформації. У галузях геології та біологічних наук машиногенеровані дані стають значно ціннішими, коли до них забезпечено швидкий і зручний доступ [1].

Підприємства накопичують більше інформації про свої проєкти, а їхні співробітники очікують миттєвого доступу до даних та можливості спільної роботи. Що ж становить основний обсяг цих даних? Переважна їхня частина є “неструктурованими” даними. Це означає, що вони не мають заздалегідь визначеної моделі та, як правило, зберігаються у вигляді файлів, а не записів у базах даних (які є структурованими даними). Основна маса таких неструктурованих даних - це зображення, відео, електронні листи, документи та файли різних форматів, що створюються мільярдами користувачів на величезній кількості пристроїв, кількість яких постійно зростає. Додатково до цього, значний обсяг даних генерується пристроями які під'єднані до інтернету, такими як датчики та камери, що використовуються в різних галузях промисловості. Отже, при переході на хмарну модель, фірми можуть своєчасно використовувати нові сервіси та динамічно реагувати на появу нових вимог для свого бізнесу. Широкий попит і поширення хмарних технологій, а так само велика увага до систем аналізу великих даних, вимагають використання СЗД, які в змозі впоратися з масовим попитом, з обсягами, що перевищують петабайт, з можливістю масштабування ємності без загроз досягнення будь-яких фізичних обмежень [2].

Актуальність роботи полягає в тому, що при повсюдному впровадженні хмарних технологій є принципово новий підхід до побудови СЗД в центрах обробки даних.

Метою кваліфікаційної роботи магістра є побудова системи надійного зберігання даних на основі OpenStack Object Storage.

Поставлена мета досягається розв’язанням таких основних завдань:

– проаналізувати доступні системи об’єктного зберігання даних, зокрема Open Source рішення для Linux/Unix, та оцінити їхні можливості;

– визначити переваги та недоліки OpenStack Object Storage (Swift) у порівнянні з іншими технологіями для розподіленого зберігання;

– спроєктувати архітектуру кластера OpenStack Swift з урахуванням вимог до надійності, масштабованості та продуктивності.

Об’єктом дослідження є процеси зберігання даних з використанням OpenStack Swift.

Предметом дослідження є алгоритми та механізми організації, управління та забезпечення надійності зберігання даних у OpenStack Object Storage.

Наукова новизна отриманих результатів:

– детально описано основні складові системи надійного зберігання даних на основі OpenStack Object Storage, зокрема механізми реплікації, балансування навантаження та управління простором зберігання;

– досліджено особливості конфігурації та масштабування OpenStack Object Storage для забезпечення високої доступності та продуктивності;

– практично реалізовано систему зберігання даних, що забезпечує ефективне управління великими обсягами неструктурованих даних.

Практична значимість отриманих результатів полягає у тому, що розроблена система надійного зберігання даних на базі OpenStack Object Storage дозволяє створити економічно вигідну альтернативу дорогим комерційним рішенням, зберігаючи при цьому високу продуктивність, масштабованість та відмовостійкість [3].

Для розв’язання поставлених задач використовувалися методи дослідницький та практичний.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Вимоги до системи, яка розробляється

РСЗД призначена для зберігання даних, доступних для хмарної платформи OpenStack. Під даними в контексті хмарної платформи будемо вважати [3, 4]:

– ефемерні диски серверів (ephemeral Nova volumes);

– постійні диски (Cinder volumes);

– образи віртуальних серверів (Glance images).

В результаті створення даної системи зберігання даних повинні бути забезпечені наступні показники СЗД: масштабованість та відмовостійкість.

В системі функціонально визначені такі підсистеми:

– розподілене сховище даних;

– інтерфейси управління СЗД та хмарною платформою OpenStack.

Забезпечення пристосовності системи повинно виконуватися за рахунок:

– розширення сумарної ємності системи при додаванні нового сервера;

– зберігання;

– зменшення сумарної ємності системи при видаленні сервера;

– зберігання (наприклад, в результаті збою сервера);

– своєчасної заміни обладнання, яке вийшло з ладу;

– модернізації архітектури і інтерфейсу відповідно до нових вимог;

– своєчасного адміністрування сервера;

– оперативного реагування на побажання користувачів.

Надійність СГД повинна забезпечуватися за рахунок:

– відсутності єдиної точки відмови в архітектурі;

– розподіленого зберігання даних з автоматичною реплікацією мережею між серверами;

– можливість відновлення необхідного рівня відмовостійкості при виході з ладу частини обладнання;

– можливості зміни рівня відмовостійкості (фактора реплікації) для різних пулів.

Система повинна надавати інтерфейс командного рядка для управління. Інформаційна безпека в системі повинна здійснюватися за рахунок доступу до управління по захищених протоколах SSH / HTTPS та аутентифікації СГД всередині кластера (між серверами) [4].

Вимоги до розподіленого сховища інформації :

– використання тільки технологій з відкритими вхідними кодами;

– список підтримуваних операційних систем: Red Hat Enterprise Linux, CentOS, Fedora;

– автоматична реплікація даних між серверами;

– можливість відновлення даних після збою частини обладнання;

– можливість задання особливих параметрів реплікації даних з урахуванням можливих failure доменів;

– нативна інтеграція з OpenStack (nova, cinder, glance і swift);

– можливість використання моментальних знімків (снепшот);

– можливість використання твердотільних накопичувачів для більш вимогливих до продуктивності системи зберігання клієнтів.

Вимоги до інтерфейсу управління сховищем даних:

– надання інформації за запитом;

– забезпечення цілісності інформації;

– забезпечення коректного відображення інформації.

Вимоги до ПЗ:

– Операційна система: система повинна бути спроєктована для роботи під керуванням Red Hat Enterprise Linux 7 або аналогічних ОС (Fedora Linux 20,CentOS 7).

– СУБД: як БД (якщо потрібно для Web Інтерфейсу) має використовуватися СУБД MySQL або MariaDB.

Для створення системи необхідно провести наступні види робіт:

- Аналіз доступних Linux / Unix OpenSource РСЗД (включаючи розподілені файлові системи) на прикладі Ceph, GlusterFS, LustreFS, Swift, pNFS;
- Вибір тієї РСЗД, яка найбільш повно задовольняє всім критеріям;
- Вибір найкращих сценаріїв використання обраних СЗД;
- Проєктування архітектури;
- Оптимізація продуктивності;
- Інтеграція з хмарною платформою OpenStack.

1.2 Огляд платформи OPENSTACK

Проєкт OpenStack – це хмарна платформа з відкритим вихідним кодом. Він був заснований в 2010 році Rackspace і NASA для допомоги організаціям у використанні хмарних обчислень. Відмінними рисами даного проєкту є простота реалізації, масштабованість і великий набір функцій. Над проєктом працюють експерти в сфері хмарних обчислень з усього світу. [5,6] OpenStack надає рішення Інфраструктура-як сервіс (IaaS) при допомозі різних програмних компонентів. Кожен компонент пропонує API, що полегшує їх інтеграцію.

Ключові особливості OpenStack:

- швидке розгортання віртуальних серверів;
- мережа, яка програмно визначається (Software-defined network - SDN);
- єдина точка управління інфраструктурою;
- малі витрати зусиль для підтримки інфраструктури.

OpenStack підходить для таких сценаріїв як хмарне навантаження для створення і управління безліччю віртуальних машин з відносно коротким життєвим циклом (підхід свиноферми), середовища розробки й тестування та створення інфраструктури для Big data (Apache Hadoop або Apache Sparc).

OpenStack не підходить для повноцінної заміни класичних корпоративних платформ віртуалізації, таких як VMware vSphere, особливо в сценаріях із тривалим життєвим циклом віртуальних машин. Його архітектура більше орієнтована на

динамічне, масштабоване середовище, типове для хмарних застосунків і мікросервісів, а не для стабільної експлуатації довготривалих сервісів. Класичні платформи віртуалізації мають більш зрілі механізми забезпечення безперервності бізнесу, такі як live migration, автоматичне резервування та глибока інтеграція з апаратним забезпеченням. Хоча OpenStack має базові засоби відновлення після відмов, вони не досягають рівня надійності, необхідного для критичних корпоративних систем. Тому в середовищах, де пріоритетом є максимальна доступність і стійкість до збоїв, OpenStack доцільно використовувати як доповнення до основної інфраструктури, а не як її повну заміну. Разом з тим в OpenStack присутні наступні обмеження:

– відсутні кошти забезпечення високої доступності віртуальних серверів;

– обмежені операції з віртуальними машинами (загальні для всіх хмар);

– додаткове апаратне забезпечення (NIC, PCI пристрої) НЕ може бути додано через портал самоврядування (Dashboard), тільки через консоль;

– зміна розміру віртуальних машин вимагає великих трудовитрат.

Нижче наведено список сервісів OpenStack [7].

Портал самообслуговування (Dashboard), проект Horizon. Надає веб-портал самообслуговування для взаємодії з основними сервісами OpenStack. Дозволяє виконувати такі операції, як запуск віртуального сервера, привласнення публічної IP адреси і налаштування доступу.

Обчислення (Compute), проєкт Nova. Управляє життєвим циклом віртуальних серверів в середовищі OpenStack. Обов'язками сервісу є призупинення, планування та виведення з експлуатації віртуальних серверів на вимогу.

Мережа (Networking), проєкт Neutron. Надає підключення до мережі як послугу для сервісів OpenStack, таких як OpenStack Compute. Надає API користувачам для визначення мереж і приєднання до них. Володіє змінною архітектурою, яка підтримує безліч популярних мережевих вендорів і

технологій.Об’єктне сховище (Object Storage), проект Swift. Зберігає і добуває довільні об’єкти неструктурованих даних за допомогою RESTful, заснованому на HTTP API. Він високо відмовостійкий завдяки реплікації даних і архітектури, яка масштабується.

Блочне сховище (Block Storage), проект Cinder. Забезпечує блочне сховище для запуску віртуальних серверів. Його архітектура драйверів , що замінюються, полегшує створення і управління блоковими пристроями.

Сервіс ідентифікаціїї (Identity service), проект Keystone. Надає сервіс аутентифікації і авторизації для інших сервісів OpenStack. Також надає каталог endpoints для всіх сервісів OpenStack.

Сервіс образів (Image service), проект Glance. Зберігає і добуває образи дисків віртуальних машин. OpenStack Compute використовує цей сервіс під час створення віртуальних машин.

Сервіс обліку використання ресурсів (Telemetry), проект Ceilometer. Спостерігає за хмарою OpenStack і проводить виміри для виставлення рахунків, тесту продуктивності, масштабованості і статичних цілей.

Сервіс оркестрації (Orchestration), проект Heat. Організовує кілька складових хмарних додатків, використовуючи нативний шаблонний формат HOT або шаблонний формат AWSCloudFormation, за допомогою OpenStack- native REST API і CloudFormation-сумісний Query API. Сервіс БД (Database service), проект Trove. Забезпечує масштабований і надійний хмарний БД-як-сервіс функціонал для реляційних і нереляційних СКБД.

OpenStack є модульною платформою, яка складається з окремих компонентів, кожен з яких виконує визначену функцію в хмарній інфраструктурі. Основними компонентами є сервіси обчислень, зберігання даних, мережевої взаємодії, автентифікації та оркестрації. Ці сервіси взаємодіють між собою через стандартизовані API, що забезпечує гнучкість та розширюваність системи. Кожен з компонентів може бути розгорнутий окремо або в складі єдиного середовища в залежності від потреб користувача. Нижче наведено загальну архітектуру платформи, яка демонструє взаємозв’язки між її ключовими елементами.

Загальна архітектура платформи OpenStack відображена на рисунку 1.1.

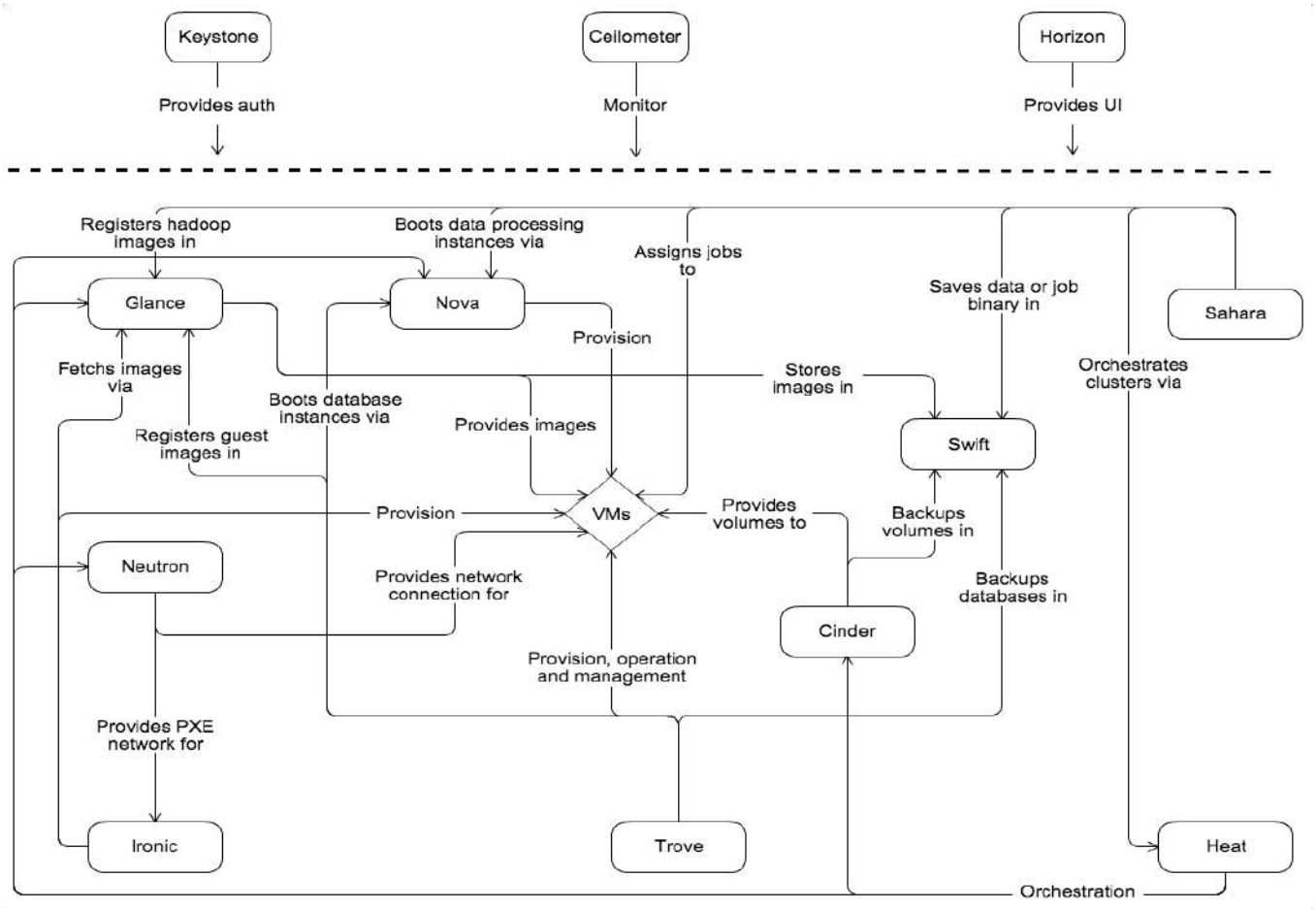


Рисунок 1.1 - Загальна архітектура платформи OpenStack

1.3 Огляд основних розподілених ФС

Метою цього порівняння є вибір найбільш оптимального рішення в області зберігання даних. Рішення повинно задовольняти вимогам комплексу ПЗ «OpenStack» [8].

В ході досліджень було проведено порівняння наступних РСЗД: LustreFS (v 2.0); pNFS (NFS 4.1); GlusterFS; IPFS; Swift.

Порівняння РСЗД проводилося виходячи з наступних критеріїв:

– масштабованість рішення;

– відмовостійкість рішення і методи догляду від «split brain»;

– оперативні витрати на управління рішенням (OpPEX);

– сумісність рішення з ПЗ «Openstack» (включаючи компоненти);швидкість I / O в послідовних операціях (читання і запис);швидкість I / O в випадкових операціях (читання і запис);підтримка реплікації (снепшот);

– Підтримка можливостей журналювання на SSD дисках;

– POSIX сумісність;

– підтримка квот;

– підтримка на рівні ядра;

– поширеність і активність розробки;

– початкові витрати на рішення.

Вся інформація, необхідна про проведення порівняння, бралася з офіційної документації [9,10].

LusterFS є типовою розподіленою ФС з виділеним сервером метаданих [11,12]. Вона оптимізована для дуже великих кластерів (~ 10 000 вузлів). Реплікацію даних і снепшот не підтримує. Доступ до даних здійснюється через клієнт Lustre. На цей момент вважається застарілою і слабо підтримується.

ФС LustreFS складається з трьох основних функціональних модулів:

– сервер метаданих (metadata server - MDS), який містить метадані про простори імен (в т.ч. імена файлів, каталогів, права доступу, карту розміщення файлів);

– сервери зберігання (object storage server - OSS), що зберігають дані файлів;

– клієнти, що використовують ці дані.

Lustre надає всім клієнтам уніфікований простір імен для всіх файлів і даних у ФС, використовуючи стандартну семантику POSIX, а також забезпечує паралельний доступ до запису та читання файлів у ФС.

pNFS забезпечує можливість паралельного читання і запису файлів з багатьох вузлів [13,14]. Використовує виділений сервер метаданих, до якого звертаються

NFS клієнти. Підтримує як розподілений (distributed), так і striped режими. Доступ до даних здійснюється через клієнт NFS. Open source версія не має «Release» статусу, комерційні версії несумісні між собою.

РСЗД pNFS складається з трьох основних функціональних модулів: •

– NFS сервер метаданих, який зберігає метадані про простори імен (в т.ч. імена файлів, каталогів, права доступу, а також карту розміщення файлів); •

– сервери зберігання даних (файлів); •

– клієнти NFS, які отримують інформацію від сервера NFS і які звертаються до серверів зберігання.

GlusterFS працює в призначеному для користувача просторі за допомогою технології FUSE і не має виділеного сервера метаданих [15]. GlusterFS може об’єднати сховища даних (на існуючих ФС ext4, XFS), що знаходяться на різних серверах, в одну паралельну мережеву ФС. РСЗД GlusterFS містить тільки один функціональний модуль – це вузли зберігання (Storage Node, brick) [16, 17].

На кожному такому вузлі працює служба glusterd, яка надає доступ клієнтам до локальних ФС. Таких brick-ів на одному сервері може бути безліч.

Наприклад, на одному диску або масиві може бути кілька розділів з різними ФС, кожна з яких буде окремим brick-ом. Кілька серверів об’єднуються в кластер або пул зберігання даних (Trusted Storage Pool в термінології GlusterFS). В рамках пулу зберігання підтоми на різних вузлах об’єднуються в логічні томи (volumes) різної конфігурації.

Основна відмінність від попередніх ФС у тому, що метадані зберігаються не на окремому сервері, а в додаткових атрибутах файлів в inode.

Міжпланетна файлова система - це децентралізований протокол обміну файлами, який має велике значення для застосувань у блокчейні. Він являє собою принциповий відхід від моделі клієнт-сервер, яка є типовою для таких протоколів, як HTTP [18].

Натомість IPFS надає метод обміну файлами за допомогою однорангової (P2P) мережі, який не залежить від централізованих серверів. Це досягається

завдяки використанню кількох базових технологій, зокрема розподілених хеш-таблиць (DHT), структур даних MerkleDag і протоколу BitSwap.

Розподілені хеш-таблиці (DHT) дозволяють ефективно знаходити й отримувати контент у мережі. У свою чергу, MerkleDag забезпечує унікальну та ефективну ідентифікацію та зв'язування контенту (тобто файлів і директорій) за допомогою контент-ідентифікаторів (CID) [19]. BitSwap, натхненний протоколом BitTorrent, забезпечує кооперативний обмін даними між вузлами, покращуючи децентралізоване отримання файлів.

Переваги IPFS численні. По-перше, система стійка до цензури та не має єдиної точки відмови.

Оскільки це система з контентною адресацією, кожне посилання на файл є унікальним і незмінним, що робить його корисним для архівних цілей. Крім того, IPFS ефективно розподіляє контент, зменшуючи потребу у зайвій пропускній здатності та дисковому просторі, оскільки файли передаються лише тоді, коли це необхідно, уникаючи надлишковості.

Окрім цього, IPFS дозволяє використовувати кешований контент офлайн, що робить його ідеальним для систем із обмеженим або нестабільним доступом до інтернету. Загалом, IPFS є новаторським рішенням, яке забезпечує більш надійний, ефективний та доступний спосіб обміну файлами.

Swift є рішенням з відкритим кодом, яке наразі використовується у найбільших хмарних об'єктних сховищах, включаючи Rackspace Cloud Files, HP Cloud, IBM SoftLayer Cloud та численні приватні кластери [20]. Swift може використовуватись як автономна система зберігання або як частина хмарного обчислювального середовища. Працює на стандартних дистрибутивах Linux і на типовому серверному обладнанні x86.

Як і Amazon S3, Swift має архітектуру остаточної узгодженості (eventual consistency), що робить його ідеальним для побудови великих, високо розподілених інфраструктур, орієнтованих на обробку неструктурованих даних у глобальному масштабі [21].

Усі об’єкти, що зберігаються в Swift, мають URL-адресу, а застосунки можуть зберігати та отримувати дані через RESTful HTTP API відповідно до стандартів. Об’єкти можуть містити розширені метадані, які можуть бути проіндексовані та доступні для пошуку.

Усі об’єкти зберігаються у вигляді кількох копій і реплікуються у максимально незалежні зони доступності та регіони. Система масштабується шляхом додавання додаткових вузлів, що дозволяє досягти економічно вигідного лінійного розширення сховища. Вийшлі з ладу вузли та диски можуть бути замінені без зупинки кластера.

Об’єктне сховище також забезпечує стійкий блочний рівень зберігання для використання з обчислювальними інстансами OpenStack для задоволення потреб зберігання з боку віртуальних машин. Це також включає управління знімками (snapshot management) для резервного копіювання даних, що зберігаються на блочних томах, включаючи OpenStack Object Storage як частину повної інтеграції з хмарним стеком OpenStack. Користувач може використовувати API OpenStack після проходження автентифікації через службу ідентифікації для створення та управління ресурсами у хмарному середовищі. Існують різні способи надсилання API-запитів до сховища OpenStack Swift, зокрема інструмент командного рядка curl, командний рядок OpenStack для клієнтів та REST-клієнти.

Системи об’єктного зберігання організовують дані у вигляді ієрархії: Account, Container та Object. Account — це верхній рівень ієрархії, Container — це, як правило, “відерця”, а Object зберігає вміст даних, наприклад документи, зображення тощо. Можна також використовувати параметри запитів, такі як marker, limit, end-marker, для здійснення переходу по великому списку контейнерів або об’єктів. Хмарне об’єктне сховище OpenStack Swift надає весь необхідний функціонал для локальних датацентрів аналогічно до хмарного сховища Amazon S3.

У таблиці 1.1 представлено підсумкове порівняння параметрів ФС на підставі даних з офіційної документації. Результати наведено за шкалою від 022 до 3, де 0 -

не реалізовано зовсім, 3 реалізовано повністю. Рейтинг – сума показників усіх оцінених параметрів для кожної ФС.

Таблиця 1.1 – Перелік перевірок і результатів випробувань

№з/п	Найменування параметру	Lustre FS	pNFS	Gluster FS	Swift	IPFS
1.	Масштабованість рішення	2	1	2	3	3
2.	Відмовостійкість рішення і шляхи обходу «Split brain»	1	0	1	2	1
3.	Оперативні витрати на управління рішенням (OpPEX)	1	1	3	3	1
4.	Сумісність рішення з ПЗ «Openstack»	0	0	3	3	0
5.	Підтримка реплікації (Снепшот)	0	0	2	2	2
6.	Підтримка квот	2	0	2	3	0
7.	Підтримка на рівні ядра	0	2	3	3	0
8.	Поширеність і активність розробки	0	0	3	3	2
9.	Початкові витрати на рішення (CaPEX)	0	0	3	3	1
10.	Рейтинг	6	4	22	25	10

За результатами порівняння для створення СЗД була обрана реалізація на базі OpenStack Swift у режимі реплікації об’єктного сховища без використання додаткових файлових систем.

1.3 Об’єктне сховище в порівнянні з іншими типами сховищ

Різні типи даних мають різні шаблони доступу, і тому найкраще зберігаються на різних типах систем зберігання. Існує три широкі категорії зберігання даних: блочне сховище, файлове сховище та об’єктне сховище.

Блочне сховище зберігає структуровані дані, які представлені як блоки однакового розміру (скажімо, 212 біт на блок) без будь-якої інтерпретації бітів. Часто такий тип зберігання корисний, коли застосунок має потребу в тісному контролі над структурою даних. Поширене використання блочного сховища — бази даних, які можуть використовувати “сирий” блочний пристрій для ефективного читання і запису структурованих даних. Додатково, файлові системи використовуються для абстрагування блочного пристрою, що потім робити усе — від запуску операційних систем до зберігання файлів.

Файлове сховище – це те, що ми найчастіше бачимо як користувачі настільних комп’ютерів. У найпростішій формі файлове сховище бере жорсткий диск і надає файлову систему для зберігання неструктурованих даних. Дата-центр містить системи, які надають файлову систему через мережу. Хоча файлове сховище забезпечує корисну абстракцію поверх пристрою зберігання, виникають складнощі при масштабуванні системи. Файлове сховище потребує сильної узгодженості, що створює обмеження при зростанні системи та її роботі під великим навантаженням. Крім того, файлові системи часто потребують додаткових функцій (таких як блокування файлів), які створюють перешкоди для ефективної роботи з великими обсягами даних.

Об’єктне сховище не надає доступу до “сирих” блоків даних; також воно не пропонує доступ на основі файлів. Натомість воно надає доступ до цілих об’єктів або блобів даних – зазвичай через API, специфічне для цієї системи. Об’єкти доступні через URL-адреси за допомогою HTTP-протоколів, подібно до того, як вебсайти доступні через браузери. Об’єктне сховище абстрагує ці розміщення як URL, щоб система зберігання могла зростати і масштабуватися незалежно від основних механізмів зберігання. Це робить об’єктне сховище ідеальним для систем, яким потрібно зростати і масштабуватися за обсягом, одночасними підключеннями, або обома параметрами.

Однією з головних переваг об’єктного сховища є його здатність розподіляти запити до об’єктів між великою кількістю серверів зберігання. Це забезпечує надійне, масштабоване сховище для великих обсягів даних за відносно низькою вартістю.

У міру масштабування система може продовжувати представляти єдиний простір імен. Це означає, що застосунок або користувач не має – і дехто сказав би, не повинен мати – знань про те, яка система зберігання буде використана. Це зменшує навантаження на оператора, на відміну від файлової системи, де операторам можливо доведеться керувати кількома томами сховища. Оскільки система об’єктного зберігання надає єдиний простір імен, немає необхідності розбивати дані та надсилати їх у різні сховища, що може збільшити складність та спричинити плутанину.

1.5 Висновки до першого розділу

У першому розділі було проведено детальний аналіз предметної області та сформульовано вимоги до системи надійного зберігання даних (СЗД), яка інтегрується з хмарною платформою OpenStack. Визначено основні функціональні підсистеми, необхідні характеристики масштабованості, відмовостійкості, інформаційної безпеки та управління. Окрему увагу приділено вимогам до використання відкритого програмного забезпечення, підтримки сучасних операційних систем, інтеграції з компонентами OpenStack (Nova, Cinder, Glance, Swift) та забезпеченню гнучкості адміністрування і експлуатації.

Було розглянуто архітектуру OpenStack як однієї з провідних хмарних платформ із відкритим вихідним кодом, її модульну побудову та основні сервіси, серед яких особливу роль у контексті СЗД відіграє об’єктне сховище Swift. OpenStack демонструє високий рівень масштабованості, інтегрованості та підтримки типових сценаріїв розгортання хмарних інфраструктур.

У результаті порівняльного аналізу найбільш поширених розподілених файлових систем (LustreFS, pNFS, GlusterFS, IPFS, Swift) за цілою низкою критеріїв

(масштабованість, відмовостійкість, сумісність з OpenStack, реплікація, підтримка квот, активність розробки тощо), було виявлено, що Swift має найвищий сумарний рейтинг і повною мірою задовольняє вимогам до побудови надійної, розширюваної та інтегрованої системи зберігання даних для хмарної інфраструктури.

Таким чином, за результатами дослідження для реалізації системи надійного зберігання даних обрано OpenStack Swift у режимі реплікації об’єктного сховища без використання додаткових файлових систем, що дозволяє досягти високої гнучкості, масштабованості та відмовостійкості в межах хмарної платформи.

2 ТЕОРЕТИЧНА ЧАСТИНА

2.1 Модель даних Swift

OpenStack Swift дозволяє користувачам зберігати неструктуровані об’єкти даних із канонічною назвою, що містить три частини: обліковий запис (account), контейнер (container) та об’єкт (object).

Використання однієї або кількох з цих частин дозволяє системі сформувати унікальне місце зберігання для даних.

Розташування зберігання облікового запису (/account) – це унікально назване місце зберігання, яке містить метадані (описову інформацію) про сам обліковий запис, а також список контейнерів у ньому.

Розташування зберігання контейнера (/account/container) – це визначене користувачем місце зберігання всередині облікового запису, де зберігаються метадані про сам контейнер і список об’єктів у ньому.

Розташування зберігання об’єкта (/account/container/object) – це місце, де зберігаються сам об’єкт даних та його метадані.

Оскільки частини з’єднуються разом для утворення повного шляху, імена контейнерів та об’єктів не повинні бути унікальними в межах усього кластера.

Якщо три об’єкти з іменем «Блакитний об’єкт» завантажено в різні контейнери або облікові записи, кожен з них має унікальне розташування, як показано на рисунку 2.1:

- /АкаунтА /Контейнер1 / БлакитнийОб’єкт;
- /АкаунтА /Контейнер2 / БлакитнийОб’єкт;
- /АкаунтБ /Контейнер1 / БлакитнийОб’єкт.

Таким чином, система забезпечує унікальність об’єктів не за їхньою назвою, а за повним шляхом у структурі зберігання. Це дозволяє зберігати об’єкти з однаковими іменами в різних просторах без ризику їх перезапису або конфлікту. Такий підхід значно спрощує організацію даних для різних додатків та користувачів у межах одного кластера.

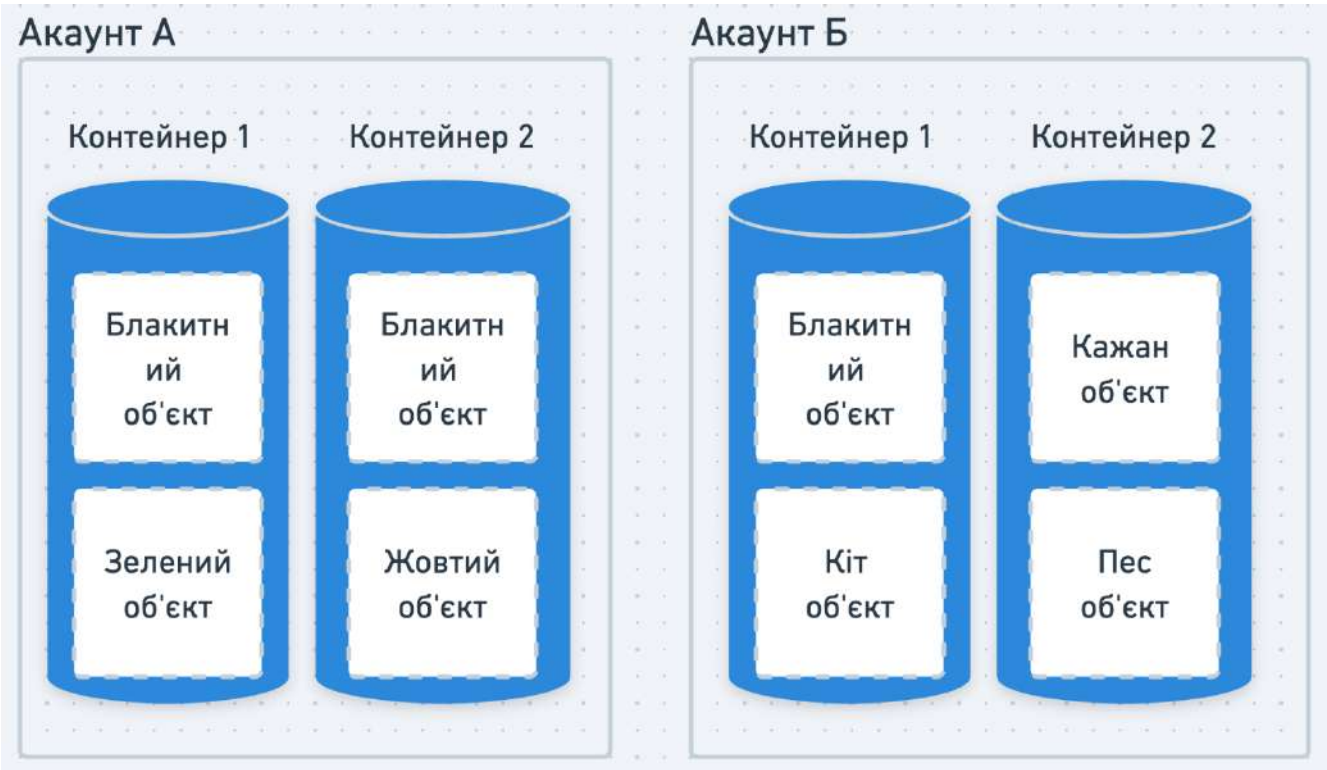


Рисунок 2.1 – Приклад співіснування об’єктів із однаковим ім’ям у різних контейнерах та акаунтах

Після розгляду структури адресації місць зберігання в OpenStack Swift, доцільно детальніше проаналізувати три основні типи ресурсів, що формують логічну модель даних у системі.

Облікові записи – це кореневе місце зберігання для даних. Їх іноді порівнюють із томами файлових систем. Облікові записи є унікально названими просторами в системі.

Кожен обліковий запис має базу даних, яка зберігає як метадані самого облікового запису, так і список усіх контейнерів у ньому.

Заголовки метаданих облікових записів зберігаються у вигляді пар ключ–значення (name–value) з тими ж гарантіями надійності, що й усі дані Swift.

Варто зазначити, що бази даних облікових записів не містять об’єктів або їх метаданих. Вони використовуються, коли користувач (авторизована особа або сервіс) запитує метадані облікового запису або список усіх контейнерів в ньому.

Облікові записи можуть бути призначені для доступу одного користувача або багатьох користувачів, залежно від встановлених дозволів.

Поширені приклади:

– облікові записи з доступом одного користувача для індивідуальних користувачів або сервісів типу “резервне копіювання”;

– облікові записи з багатокористувацьким доступом для проєктів із командною роботою.

Користувачі Swift із відповідними правами можуть створювати, змінювати та видаляти контейнери й об’єкти всередині облікового запису.

Контейнери — це визначені користувачем сегменти простору імен облікового запису, які забезпечують місце зберігання, де розміщуються об’єкти.

Хоча контейнери не можна вкладати один в один, вони концептуально схожі на каталоги або папки у файлових системах.

Оскільки контейнери знаходяться на один рівень нижче від облікового запису (/аккаунт/контейнер), вони не мають глобальних вимог до унікальності імен.

Наявність двох контейнерів з однаковим іменем у різних облікових записах, наприклад /АккаунтА/Контейнер1 та /АккаунтБ/Контейнер1, є абсолютно допустимою.

Кількість контейнерів, які користувач може створити в обліковому записі, не обмежена.

Так само, як кожен обліковий запис має базу даних, кожен контейнер також має власну базу даних, яка зберігає метадані контейнера та записи про кожен об’єкт у ньому.

Метадані контейнера зберігаються як пари ключ–значення (name–value) з тими ж гарантіями надійності, що й усі дані в Swift.

Приклад метаданих – заголовок X-Container-Read, який визначає список користувачів, що можуть читати об’єкти в контейнері.

Всередині облікового запису користувачі можуть створювати та використовувати контейнери для логічного групування даних. Також можна додавати метадані до контейнерів і використовувати функції Swift для надання

контейнерам різних властивостей – наприклад, перетворення їх на статичні вебсайти.

Об’єкти – це дані, які зберігаються в OpenStack Swift. Це можуть бути фотографії, відео, документи, журнальні файли, резервні копії баз даних, знімки файлових систем або будь-які інші дані.

Кожен об’єкт зазвичай включає самі дані та метадані.

Заголовки метаданих зберігаються у вигляді пар ключ–значення (name–value) з такими ж гарантіями надійності, як і об’єктні дані, і не потребують додаткової затримки для доступу.

Метадані можуть містити важливу інформацію про об’єкт. Наприклад:

– відеостудія може зберігати формат і тривалість відео разом із самим контентом;

– документ може містити інформацію про автора;

– дані про процес її отримання.

Кожен об’єкт повинен належати контейнеру. Коли об’єкт зберігається в кластері, користувачі завжди звертаються до нього через повне розташування об’єкта /аккаунт/контейнер/об’єкт. Кількість об’єктів у контейнері не обмежена.

З точки зору користувача, місце розташування об’єкта – це спосіб його знайти.

2.2 Архітектура OpenStack Swift

OpenStack Object Storage (Swift) використовується для резервованого, масштабованого зберігання даних за допомогою кластерів стандартизованих серверів, що дозволяють зберігати петабайти доступної інформації [21]. Це довгострокова система зберігання для великих обсягів статичних даних, які можуть бути отримані та оновлені. Object Storage використовує розподілену архітектуру без централізованої точки керування, що забезпечує вищу масштабованість, відмовостійкість і довговічність.

Об’єкти записуються на кілька апаратних пристроїв, а програмне забезпечення OpenStack відповідає за реплікацію та цілісність даних у межах кластера. Кластери зберігання масштабуються горизонтально шляхом додавання нових вузлів. У разі відмови вузла OpenStack відтворює його вміст з інших активних вузлів. Оскільки OpenStack використовує програмну логіку для забезпечення реплікації та розподілу даних між різними пристроями, можуть застосовуватися недорогі стандартні жорсткі диски та сервери замість дорожчого обладнання.

Object Storage є ідеальним рішенням для економічно ефективного масштабованого зберігання. Він забезпечує повністю розподілену платформу сховища з доступом через API, яку можна інтегрувати безпосередньо в застосунки або використовувати для резервного копіювання, архівації та довготривалого зберігання даних. Його архітектура акцентує увагу на масштабованості, надійності та продуктивності, що робить її придатною для рішень у сфері хмарного зберігання. Нижче наведені ключові компоненти та концепції архітектури Swif [2. 3]:

Проксі-сервер відповідає за зв'язування всіх інших компонентів архітектури Swift. Для кожного запиту він шукає розташування облікового запису, контейнера або об'єкта в кільці (див. нижче) та маршрутизує запит відповідно.

Кільце представляє собою відображення між іменами сутностей, збереженими на диску, та їх фізичним розташуванням.

Політики зберігання надають можливість постачальникам об'єктного зберігання диференціювати рівні сервісу, функції та поведінку впровадження Swift.

Сервер об'єктів простий сервер блоб-зберігання, який може зберігати, отримувати та видаляти об'єкти, що зберігаються на локальних пристроях.

Сервер контейнерів відповідає за обробку списків об'єктів. Він не знає, де розташовані ці об'єкти, а знає лише які об'єкти містяться в конкретному контейнері.

Подібний на сервер контейнерів, є сервер облікових записів за винятком того, що він відповідає за списки контейнерів, а не об'єктів.

Реплікація розроблена для підтримки системи в консистентному стані у випадку тимчасових помилок, таких як мережеві збої або відмови дисків. Рисунок 2.2 відображає високорівневу архітектуру.

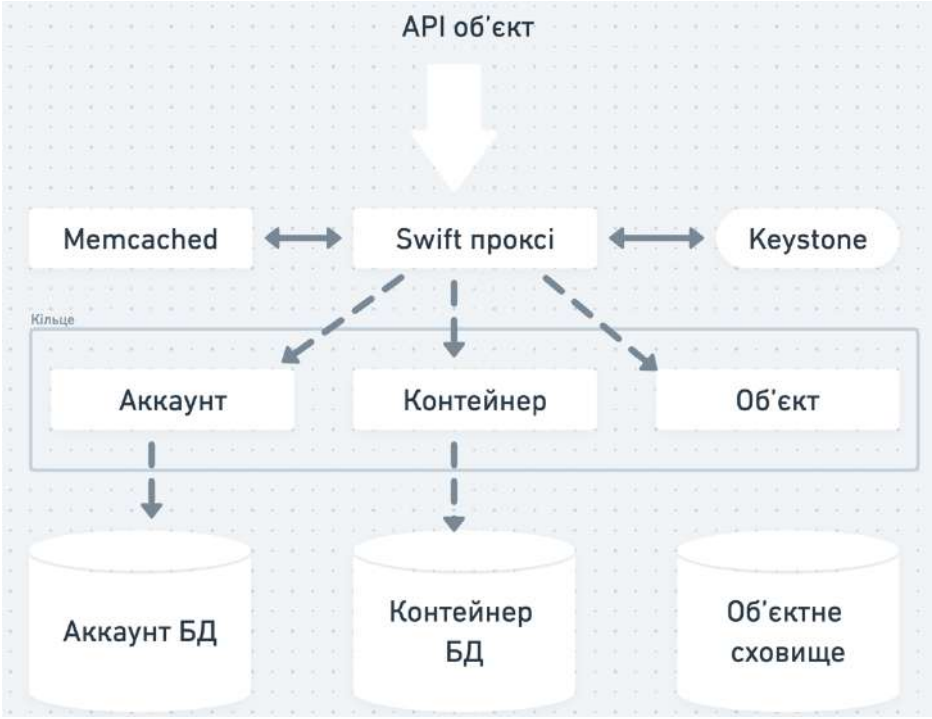


Рисунок 2.1 – Високорівнева архітектура Swift

Акаунти, контейнери та об’єкти є ресурсами, які в кінцевому підсумку зберігаються на фізичному обладнанні. Машину, яка виконує процеси Swift називають вузлом (node). Кластер Swift – це група вузлів, які разом запускають повний набір процесів і служб, необхідних для функціонування як розподілена система зберігання.

Щоб забезпечити стійкість та ізолювати відмови, вузли в кластері можна організувати в регіони та зони .

Swift дозволяє визначати частини кластера, які фізично розділені, як регіони. Регіони часто визначаються географічними межами. Наприклад, новий регіон може бути визначений для кількох стійок серверів (вузлів), які розміщені в місці з вищою затримкою, поза основним розташуванням інших вузлів кластера. Кластер повинен

мати щонайменше один регіон; як наслідок, існує багато кластерів з одним регіоном, де всі вузли належать до одного регіону. Коли кластер використовує два або більше регіонів, він називається багаторегіональним кластером (MRC).

Багаторегіональні кластери мають певні особливості при читанні та записі даних. Зокрема, коли БРК отримує запит на читання, Swift надає перевагу копіям даних, які знаходяться ближче, вимірюючи це за затримкою. Це називається читальна пріоритетність (read affinity).

Swift також пропонує записну пріоритетність (write affinity), але не за замовчуванням. За замовчуванням Swift намагається одночасно записати дані в кілька місць у БРК, незалежно від регіону. Це добре підходить для середовищ з низькою затримкою, де запити на запис, передача даних і відповіді можуть виконуватися швидко.

У випадку з високою затримкою може бути корисним увімкнути записну пріоритетність. З увімкненою write affinity кожен запит на запис створює необхідну кількість копій даних локально. Потім копії передаються асинхронно до інших регіонів. Це дозволяє швидше виконувати запис, ціною збільшеної неузгодженості між регіонами до завершення асинхронної передачі.

У межах регіонів Swift дозволяє налаштовувати зони доступності (availability zones) для ізоляції відмов. Зона доступності повинна визначатися як окремий набір фізичного обладнання, чия відмова ізольована від інших зон. У великому розгортанні зони доступності можуть визначатися як окремі приміщення в дата-центрі, можливо, відокремлені між собою міжмережевими екранами та живлені від різних постачальників електроенергії. У розгортанні в межах одного дата-центру зони доступності можуть бути різними стійками. Хоча кластер повинен мати щонайменше одну зону, набагато частіше кластери мають багато зон.

Вузли – це фізичні сервери, які запускають один або кілька серверних процесів Swift. Основні серверні процеси Swift – це проксі, аккаунт, контейнер та об’єкт. Вузол, що запускає процеси серверів аккаунт або контейнер, також буде зберігати дані акаунтів або контейнерів і їхні метадані. Вузол, що запускає процес об’єктного сервера, буде зберігати об’єкти та їхні метадані.

Збірка вузлів, які запускають усі необхідні процеси для функціонування Swift як розподіленої системи зберігання, називається кластером. Як було обговорено в цьому розділі, існує багато способів логічно групувати вузли в кластері, переважно за фізичними критеріями. Додатковим способом групування вузлів є політики зберігання.

Політики зберігання – це спосіб для адміністраторів Swift визначити простір у кластері, який можна налаштувати різними способами, включно з місцем розташування, реплікацією, апаратним забезпеченням і томами, щоб задовольнити конкретні потреби зберігання даних. Це потужний і гнучкий інструмент. Політики зберігання встановлюються на рівні об’єкта, а реалізуються на рівні контейнера.

Це розширена гнучкість кластера об’єктного зберігання Swift означає, що адміністратори можуть створювати політики зберігання, які орієнтовані на підвищену стійкість даних, географічні обмеження тощо.

2.3 Зберігання даних

Зберігати дані на диску та надавати API до них не складно. Складність полягає в обробці збоїв. Процеси узгодженості відповідають за пошук і виправлення помилок, спричинених пошкодженням даних або апаратними збоями. Вони є однією з основних причин, чому Swift надійним.

Багато процесів виконуються під “капотом”, щоб забезпечити цілісність і доступність даних. Процеси узгодженості виконуються у фоновому режимі на вузлах, де запущені процеси сервера облікових записів, контейнерів або об’єктів. Процес проксі-сервера не має пов’язаних процесів узгодженості, оскільки він не зберігає дані облікового запису, контейнера чи об’єкта. Конкретні процеси узгодженості, які виконуються, залежать від процесу сервера:

– процеси сервера облікових записів підтримуються процесами узгодженості аудитора;

– процеси контейнерного сервера підтримуються аудитором контейнера, реплікатором контейнера, контейнером, засобами оновлення та процесами узгодженості синхронізації контейнера;

– процеси сервера об’єктів підтримуються аудитором об’єктів, реплікатором об’єктів, засобами оновлення об’єктів і процесами узгодженості терміну дії об’єктів.

Аудитори та реплікатори є основними процесами узгодженості, що використовуються для всіх трьох серверних компонентів зберігання даних: акаунтів, контейнерів та об’єктів. Процеси оновлення також застосовуються до серверів контейнерів та об’єктів. Інші механізми узгодженості є специфічними для окремих серверних процесів.

Процеси узгодженості аудиторів працюють у фоновому режимі на всіх вузлах, що зберігають дані. Тобто, якщо на вузлі запущений сервер акаунтів, контейнерів або об’єктів, то там також буде працювати відповідний аудитор. Аудитори акаунтів, контейнерів та об’єктів безперервно сканують диски на своєму вузлі, щоб переконатися, що збережені дані не зазнали пошкодження файлової системи. Якщо виявлено помилку, аудитор переміщує пошкоджені дані до карантинної області.

Процеси узгодженості реплікаторів гарантують, що в кластері зберігається достатня кількість копій останньої версії даних там, де це потрібно. Це дозволяє кластеру залишатися в узгодженому стані, навіть якщо виникають тимчасові помилки, такі як збої мережі або вихід з ладу дисків. Процеси реплікації акаунтів, контейнерів та об’єктів працюють у фоновому режимі на всіх вузлах, що запускають відповідні служби. Реплікатор безперервно аналізує свій локальний вузол і порівнює акаунти, контейнери або об’єкти з копіями на віддалених вузлах у кластері. Якщо один із віддалених вузлів містить застарілий або відсутній ресурс, реплікатор надсилає свою локальну копію на цей вузол для оновлення або заміни даних. Це називається оновленням реплікації. Варто зазначити, що реплікатори лише надсилають свої локальні дані на віддалені вузли, але не отримують віддалені копії, якщо їхні власні дані відсутні або застарілі. Реплікатор також обробляє

видалення об’єктів і контейнерів. Видалення об’єкта починається зі створення tombstone-файлу (нульовий за розміром файл, назва якого закінчується на .ts), який стає останньою версією об’єкта. Реплікатор поширює tombstone-файл як найновішу версію серед інших реплік, і після цього об’єкт повністю видаляється з системи. Видалення контейнера можливе лише в разі відсутності об’єктів у ньому. Після цього база даних контейнера позначається як видалена, а реплікатори поширюють цю зміну, що призводить до остаточного видалення контейнера з кластера.

Запит на видалення акаунту змінює його статус на “deleted”, що робить його недоступним для використання. Також це маркує акаунт для подальшої обробки процесом account reaper. Коли цей процес виявляє акаунт, позначений як видалений, він починає поступове видалення всіх пов’язаних із ним об’єктів і контейнерів, зрештою видаляючи сам акаунт. Щоб запобігти випадковому видаленню, процес може бути налаштований на затримку перед остаточним видаленням даних.

Процес узгодженості оновлення контейнерів відповідає за підтримку актуальності списків контейнерів в акаунтах. Крім того, він оновлює кількість об’єктів, кількість контейнерів і обсяг використаного місця в метаданих акаунта. Процес оновлення об’єктів виконує схожу функцію, але для списків об’єктів у контейнерах. Проте цей процес є резервним. Основна відповідальність за оновлення списків об’єктів у контейнерах лежить на сервері об’єктів. Якщо ж він не може виконати цю операцію, процес оновлення об’єктів перебирає її на себе, забезпечуючи правильне оновлення списків контейнерів, а також кількості об’єктів і використаного простору в метаданих контейнера.

Цей процес дозволяє автоматично видаляти позначені об’єкти після досягнення встановленого терміну. Процеси видалення відповідають за остаточне очищення цих даних.

Для розподілу даних облікових записів, контейнерів і об'єктів між різними вузлами в кластері використовується спеціальна система пошуку. Коли процес на будь-якому з вузлів потребує знайти частину даних, він звертається до локальної копії кілець і визначає всі розташування даних, звертаючись до кільця облікового

запису, контейнера або одного з кілець об'єктів. Оскільки Swift зберігає кілька копій кожного фрагмента даних, пошук місця зберігання даних охоплює кілька локацій, після чого процес запитує обране розташування для отримання даних.

Кільця є наборами таблиць пошуку, що розподілені по кожному вузлу в кластері. Swift використовує модифіковану версію узгодженого хешування для побудови кілець. Кільця створюються та оновлюються за допомогою утиліти ring-builder. Створення кільця відбувається у два етапи: оновлення файлів builder'а та ребалансування кілець.

Під час створення кластера Swift використовує утиліту ring-builder для створення файлів ініціалізації, які є своєрідним планом для побудови кілець. Окремий файл ініціалізації створюється для акаунтів, контейнерів і кожної політики зберігання об'єктів. Ці файли містять інформацію, таку як кількість реплік, час блокування реплік та розташування дисків у кластері.

Під час зміни ємності кластера Swift використовує ring-builder для оновлення файлу ініцілізації з новою інформацією про диски.

Важливо зазначити, що інформація в кожному файлі ініціалізації є окремою від інших. Наприклад, можливо додати деякі диски до файлів ініціалізації акаунтів і контейнерів, але не до файлів об'єктів.

Коли кільце Swift створюється, значення кількості реплік (replica count) визначає, скільки копій кожної партиції буде розміщено в кластері. Наприклад, якщо у встановлено кількість реплік три, то кожнен том буде репліковано в загальній кількості трьох копій. Кожна з трьох реплік буде зберігатися на окремому пристрої в кластері для забезпечення надмірності. Коли об'єкт розміщується в кластері, його хеш буде відповідати певному тому і буде скопійований до всіх трьох розташувань цього тому.

Чим більше реплік, тим краще дані захищені від втрати у разі відмови (особливо якщо репліки розподілені між дата-центрами та географічними регіонами).

Щоб зрозуміти принцип роботи кілець та розміщення даних, спершу потрібно розглянути основи хешування і механізм узгодженого хешування. Потім

аналізується модифіковане узгоджене хешування у Swift, що використовується для створення кілець Swift. Хешування можна порівняти з використанням друкованого набору енциклопедії. Щоб знайти інформацію, використовується кілька перших літер слова (проста форма хешування), обираючи відповідний том за діапазоном літер. Для розподілу даних на кількох дисках у Swift використовуються більш складні хеш-функції, що дозволяють оптимізувати пошук і зберігання даних. У таблиці 2.1 наведено приклад, який ілюструє, як об’єкти розподіляються між фізичними дисками.

Таблиця 2.1 – Розподіл об’єктів між накопичувачами за допомогою базової хеш-функції

Об’єкт	Хеш	Відображення	Визначення на диску
Image 3	1213f717f7f754f050d0246fb7d6c43b	hash(Image 3) % 4 = 3	Drive 3
Music 1	4b46f1381a53605fc0f93a93d55bf8be	hash(Music 1) % 4 = 1	Drive 1
Music 2	ecb27b466c32a56730298e55bcace257	hash(Music 2) % 4 = 0	Drive 0
Image 3	1213f717f7f754f050d0246fb7d6c43b	hash(Image 3) % 4 = 3	Drive 3
Music 1	4b46f1381a53605fc0f93a93d55bf8be	hash(Music 1) % 4 = 1	Drive 1
Image 1	b5e7d988cfdb78bc3be1a9c221a8f744	hash(Image 1) % 4 = 2	Drive 2
Image 2	943359f44dc87f6a16973c79827a038c	hash(Image 2) % 4 = 3	Drive 3

У випадку зберігання об’єктів на сервері, що містить чотири накопичувачі (або розділи), позначені як Диск 0, 1, 2 і 3, використовується базова хеш-функція для визначення місця розташування даних.

Зокрема, дані об’єкта піддаються хешуванню за допомогою алгоритму MD5, що забезпечує отримання значення фіксованої довжини.

Результатом роботи алгоритму є шістнадцяткове число довжиною 32 символи, яке може бути перетворене у десятковий формат. Далі отримане хеш-значення ділиться на кількість доступних накопичувачів, у даному випадку – на чотири.

Після виконання операції ділення залишок визначає, на який саме диск буде розміщено об’єкт.

Таким чином, кожен об’єкт отримує унікальне значення відображення, яке визначає його фізичне розташування у системі зберігання.

Такий метод забезпечує швидкий і детермінований вибір розташування без необхідності додаткового пошуку, приклад роботи показано на рисунку 2.2.

Завдяки цьому система здатна масштабуватись горизонтально, не потребуючи суттєвого перебудовування всієї структури.

У разі додавання або видалення диска змінюється лише частина хеш-простору, що мінімізує кількість переміщень об’єктів.

Це дозволяє підтримувати цілісність даних навіть у процесі динамічного оновлення конфігурації.

Загалом, така стратегія хешування забезпечує не лише ефективний розподіл, а й стабільність при зміні топології системи.

Крім того, алгоритм легко реалізується програмно і не потребує складної логіки балансування навантаження.

Він добре масштабується на великі обсяги даних та кількість вузлів, що особливо важливо для розподілених середовищ.

Завдяки використанню детермінованої функції розміщення, можливе швидке відновлення карти об’єктів навіть після відмови частини системи.

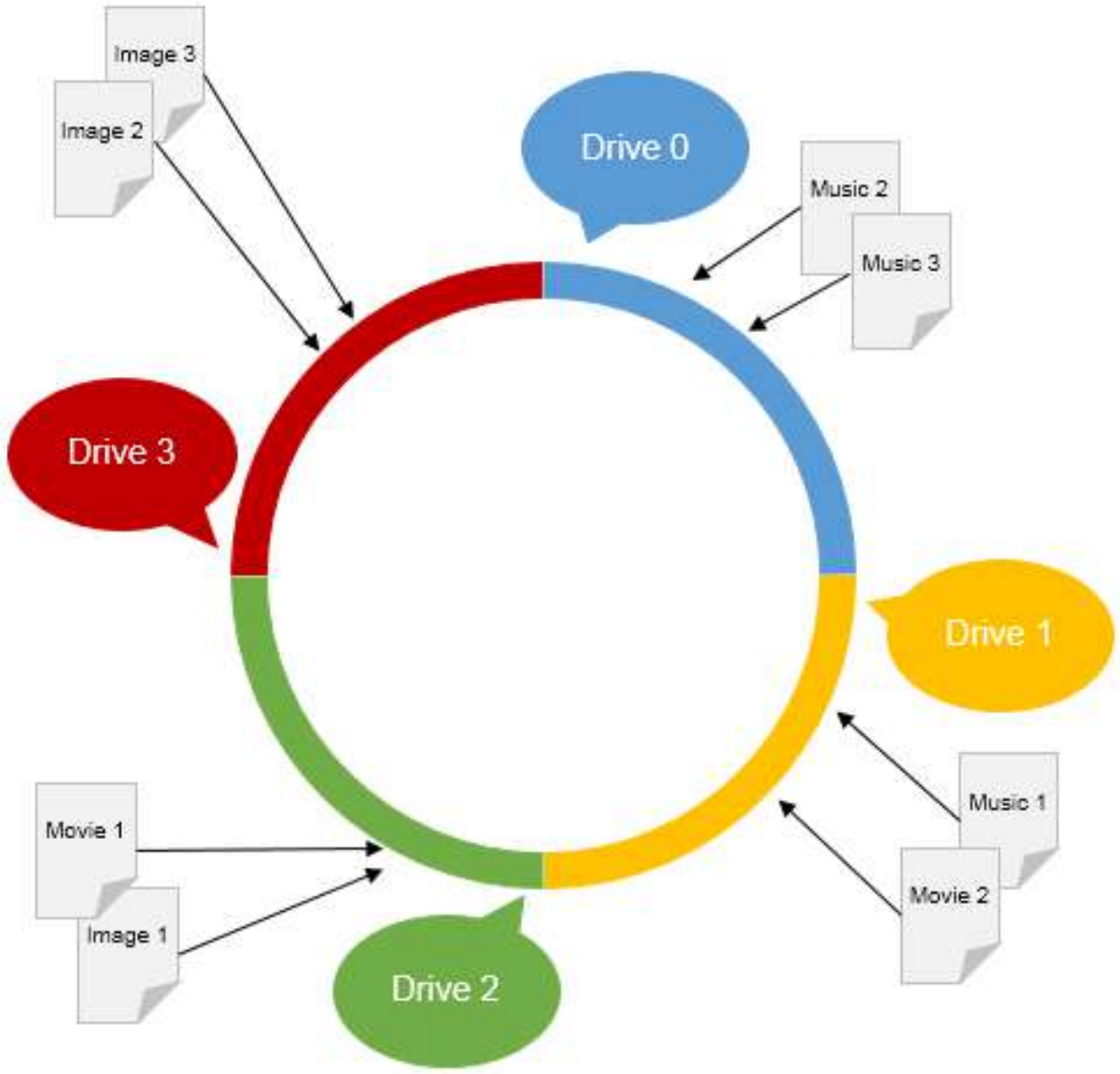


Рисунок 2.2 – Розподіл об’єктів між накопичувачами

2.3 Множинні маркери в алгоритмі консистентного хешування

Як відомо, в алгоритмі хешування кожному накопичувачу призначається один великий діапазон хеш-значень для розміщення об’єктів. Використання множинних маркерів допомагає рівномірно розподілити об’єкти між накопичувачами, що сприяє балансуванню навантаження.

Замість того щоб кожен накопичувач мав один великий діапазон хеш-значень, множинні маркери дозволяють розділити цей діапазон на менші сегменти, які розподіляються між різними накопичувачами сервера.

Наприклад, якщо необхідно зберегти 20 об’єктів на 4 накопичувачах, кожен із яких має свій діапазон хеш-значень однакової довжини, може статися так, що 14 з цих об’єктів будуть спрямовані на накопичувач 0, а решта рівномірно розподіляться між накопичувачами 1, 2 і 3. У такому випадку баланс у кільці порушується, оскільки накопичувач 0 містить значно більше значень, ніж інші. Саме тут менші діапазони хеш-значень допомагають у балансуванні навантаження.

Як вже зазначалося, алгоритм хешування використовує множинні маркери для накопичувачів, щоб відповідати за кілька менших діапазонів хеш-значень замість одного великого. Це має два позитивні ефекти: по-перше, у разі додавання нового накопичувача, він не отримує об’єкти лише з одного сусіднього диска, а натомість приймає дані з багатьох існуючих накопичувачів, що забезпечує більш рівномірне і передбачуване балансування навантаження. Кожен новий диск отримує кілька невеликих ділянок хеш-простору, що зменшує обсяг переміщення даних та покращує ефективність масштабування. По-друге, якщо накопичувач виходить з ладу або вилучається, його об’єкти не передаються одному або кільком конкретним вузлам, а рівномірно розподіляються по всьому кластеру. Завдяки цьому зменшується ймовірність перевантаження окремих дисків під час відновлення. Така гнучка схема роботи з хеш-простором робить систему більш стійкою до змін і надає переваги як у плані продуктивності, так і в обслуговуванні.

Завдяки такому підходу загальний розподіл об’єктів у кільці стає більш рівномірним, що дозволяє уникнути перевантаження окремих накопичувачів. Вага, тобто кількість об’єктів, що зберігаються на кожному з них, вирівнюється, що критично важливо для стабільності та прогнозованої продуктивності системи.

Це особливо актуально в середовищах із великим обсягом даних, де нерівномірний розподіл може призвести до сповільнення доступу або навіть відмов. Множинні маркери також спрощують масштабування: при додаванні або видаленні накопичувача лише частина об’єктів потребує перенесення. У підсумку, така структура хеш-кільця забезпечує динамічне балансування навантаження без значних витрат на перебалансування. схема роботи системи приведена на рисунку 2.3.

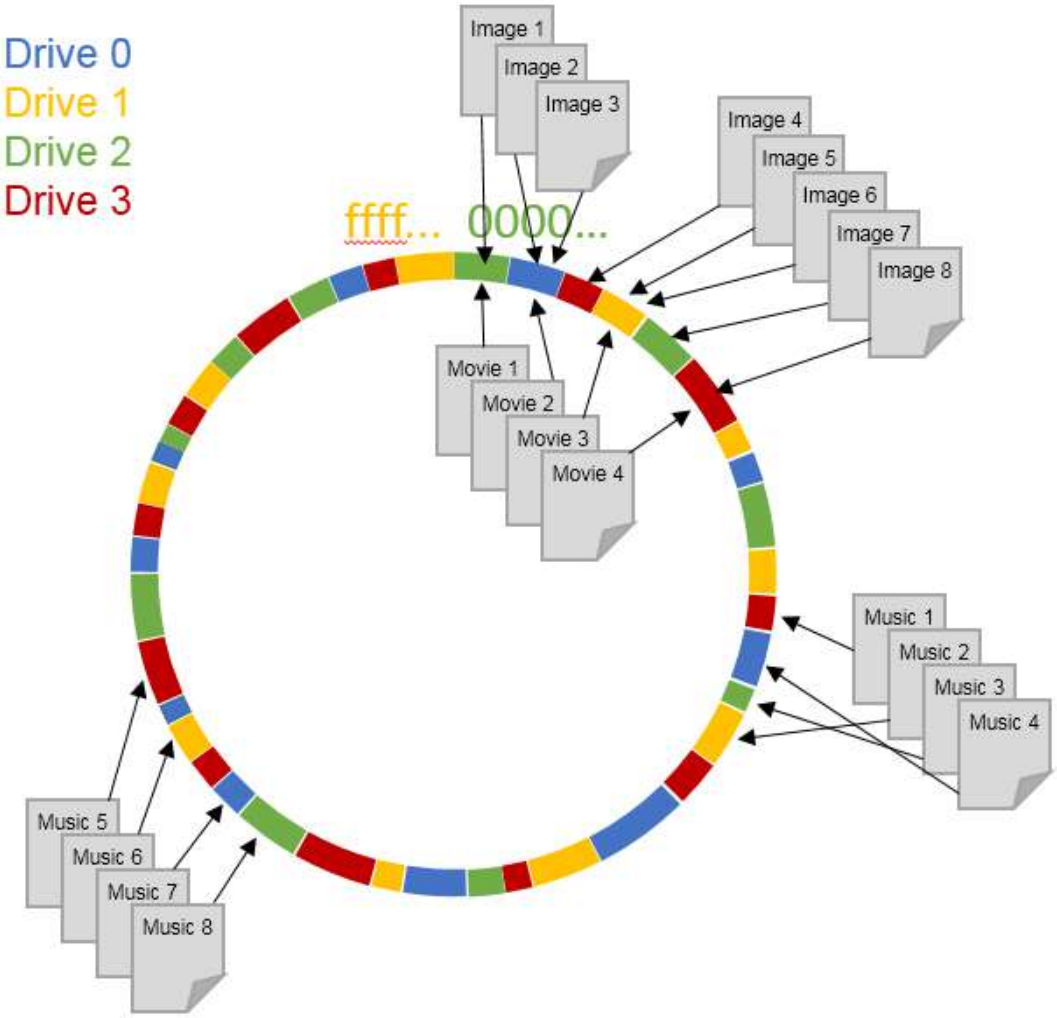


Рисунок 2.3 – Розподіл об’єктів за допомогою множинних маркерів

2.4 Масштабованість і висока доступність

OpenStack Swift розроблений як відмовостійке та масштабоване об’єктне сховище, яке забезпечує високу доступність (High Availability, HA) даних за допомогою механізмів реплікації, балансування навантаження та відмовостійкості вузлів.

Swift використовує розподілену архітектуру, у якій немає єдиної точки відмови. Всі сервери в кластері можуть незалежно виконувати свої завдання.

OpenStack Swift масштабується до величезних розмірів, але має певні труднощі зі збільшенням масштабів. Для керування всіма накопичувачами він зберігає інформацію про пристрої кластера у спеціальних файлах, які називаються

кільцями (rings). Дані розподіляються по кластеру за допомогою модифікованого алгоритму консистентного хешування [21]. При додаванні, видаленні або зміні ваги пристрою необхідно оновлювати файли кілець і поширювати їх на всі вузли Swift. Після отримання нових файлів вузол запускає процес реплікації.

Розповсюдження нових файлів кілець на всі вузли та переміщення реплік у нові місця одночасно є складним завданням. Тому був створений спеціальний параметр “min part hours”, який виконує дві функції. По-перше, він визначає мінімальний час, після якого кільце можна збалансувати знову. По-друге, під час переміщення реплік Swift спочатку переміщує лише одну репліку, залишаючи інші заблокованими на місці, поки не завершиться рух першої або не мине встановлений час.

Однак основною проблемою масштабованості Swift залишається те, що додавання, видалення або зміна ваги пристрою викликає масові переміщення даних, що може призвести до тимчасового зниження продуктивності кластера. Наприклад, якщо до кластера додається диск об’ємом 4 ТБ, а використання дискового простору в ньому становить 50%, це призведе до переміщення 2 ТБ даних на новий диск. При пропускній здатності 10 Гбіт/с та повному завантаженні цей процес займе близько 27 хвилин, за умови, що вихідні накопичувачі мають достатню швидкість передачі, новий накопичувач може приймати дані з такою швидкістю, а мережеві комутатори підтримують таку передачу. На практиці зниження продуктивності кластера може тривати годинами.

Щоб уникнути деградації продуктивності, є можливість поступового збільшення місткості, але наразі не існує відкритих методів, які б автоматизували цей процес.

Для автоматизації поступового збільшення місткості можна повторно використати глобальне блокування. У цьому випадку корисним є Zookeeper. Оскільки однакові файли кілець повинні бути доступними на всіх вузлах, їх необхідно зберігати у глобальному сховищі. Для цього можна використати базу даних, але це додасть зайву технологію, яка не призначена для таких завдань. Фактично, Swift може використовуватися для зберігання власних кілець. Перед

першим оновленням необхідно зберегти початкові порожні кільця в OpenStack Swift, що дозволить йому використовувати власне сховище для їх зберігання.

Для оновлення кілець і поступових змін місткості на всіх серверах використовується SaltStack. Зазначені технології не є обов'язковими, а є лише практичними рекомендаціями. Можна реалізувати аналогічні механізми іншими засобами, зокрема за допомогою розподіленої файлової системи або процесів синхронізації.

Основний алгоритм ґрунтується на використанні глобальних блокувань і розподіленого сховища. Важливо, щоб оновлення кілець відбувалося тільки на одному вузлі в певний момент часу. Глобальні блокування гарантують, що лише один вузол може змінювати кільця одночасно, а розподілене сховище забезпечує узгодженість даних між вузлами.

Алгоритм оновлення кілець відбувається так: SaltStack надсилає завдання на оновлення кілець до відповідних вузлів раз на певний інтервал часу (наприклад, раз на годину). Якщо поточні кільця відрізняються від збережених, вони оновлюються.

Алгоритм зміни кілець:

– SaltStack порівнює поточний стан кілець із очікуваним;

– якщо виявлено зміни, перевіряється дата останнього оновлення;

– якщо минуло більше часу, ніж визначено параметром change_interval, надсилається запит на зміну кілець;

– вузол намагається отримати блокування в Zookeeper;

– якщо блокування отримано, виконується оновлення кілець (наприклад, якщо вагу диска визначати у ГБ, то можна змінювати 25 одиниць кожні 30 хвилин на диск);

– після оновлення кільця зберігаються в OpenStack Swift;

– блокування звільняється.

Для керування Zookeeper рекомендується Apache Curator, який містить готові алгоритми для роботи з блокуваннями.

Оскільки Apache Curator є Java-бібліотекою, для керування OpenStack Swift бажано використовувати Java-клієнт. Однак наразі не існує стабільної Java-бібліотеки, яка коректно обробляє розриви з’єднання TCP для Swift. Через це рекомендовано використовувати S3 API, оскільки для нього є AWS Java SDK for Amazon S3 (aws-java-sdk-s3).

Для роботи з S3 API у Swift необхідно встановити Swift3, який додає підтримку S3 API, а також налаштувати проксі-сервери OpenStack Swift. Це дозволить використовувати AWS Java SDK for S3 для роботи з кластером Swift.

Завдяки запропонованим алгоритмам можна масштабувати кластер Swift без ризику “штормів реплікації” та порушення узгодженості даних.

2.5 Комунікація та координація в OpenStack Swift

OpenStack Swift є високонавантаженою системою об'єктного зберігання, яка забезпечує безперервну обробку запитів на запис і зчитування даних. Як і його природний тотем, Swift перебуває в постійному русі, виконуючи безліч операцій одночасно. Запити на запис призводять до створення декількох копій даних, що розподіляються між різними вузлами кластеру, тоді як запити на зчитування можуть використовуватися для відновлення резервних копій або обслуговування веб-контенту та онлайн-ігор [22,23].

Кожен запит, що надходить у систему, потребує перевірки автентифікації та авторизації, перш ніж Swift зможе його обробити. Окрім цього, у фоні постійно працюють різні серверні та узгоджувальні процеси, що забезпечують збереження та узгодженість даних. Вся ця активність вимагає ефективної комунікації та координації між компонентами системи.

Ключовим елементом архітектури Swift є проксі-сервер (Proxy Server), який є єдиною точкою взаємодії кластеру з зовнішніми клієнтами. Саме проксі-сервер реалізує Swift API, що визначає набір HTTP-запитів і правил, за якими система взаємодіє з клієнтами. Проксі-сервер виконує наступні основні функції:

– прийом HTTP-запитів на запис, зчитування, видалення даних;

– визначення прав доступу користувачів;

– перенаправлення запитів до відповідних серверів зберігання (Account, Container, Object Server);

– координація реплікації та узгодженості даних у кластері.

Оскільки проксі-сервер є єдиним компонентом, що взаємодіє із зовнішнім середовищем, він забезпечує стандартизований інтерфейс взаємодії, що відповідає специфікації Swift API [24]. Це гарантує підтримку сумісності з іншими сервісами та клієнтами, що працюють через HTTP.

При відправленні запиту до Swift-кластера він має містити наступні компоненти:

– storage URL (URL сховища);

– автентифікаційна інформація;

– HTTP-метод (HTTP verb).

Storage URL виконує дві основні функції: по-перше, він визначає маршрут запиту до кластера, а по-друге, вказує місце, де саме в кластері має бути виконано запит.

Storage URL складається з двох основних частин перша – це розташування кластера (наприклад, swift.example.com/v1/) ця частина використовується мережею для маршрутизації запиту до вузла, на якому працює проксі-сервер. Друга частина це – розташування даних (/аккаунт/контейнер/об’єкт), що визначає точне місце збереження даних, яке може бути представлене у трьох форматах: акаунт; контейнер, об'єкт.

Перший етап обробки запиту у Swift – це перевірка автентичності відправника. Це здійснюється шляхом порівняння наданої автентифікаційної інформації з даними на сервері автентифікації.

Існує два способи автентифікації в Swift:

– надсилання автентифікаційних даних разом із кожним запитом.

– використання автентифікаційного токена, отриманого за допомогою спеціального автентифікаційного запиту перед виконанням основних запитів.

Це схоже на відвідування компанії, де можна або щоразу пред’являти посвідчення особи на стійці реєстрації, або отримати тимчасовий бейдж, що дозволяє вільний вхід і вихід протягом дня без додаткової перевірки.

Під час автентифікації Swift також перевіряє рівень доступу користувача до даних. Ця інформація використовується на наступному етапі обробки запиту.

Незважаючи на те, що користувач може мати дійсні облікові дані для доступу до Swift-кластера, це не означає, що він має право виконувати будь-які дії. Проксі-сервер перевіряє авторизацію перед виконанням запиту [25].

Наприклад, якщо користувач надсилає запит із правильними автентифікаційними даними, але намагається додати об'єкт до чужого акаунта, система його автентифікує, але запит буде відхилено через відсутність необхідних прав.

Якщо дія дозволена, проксі-сервер передає запит відповідним вузлам у кластері для його виконання. Вузли виконують необхідні операції та надсилають результати назад до проксі-сервера, який формує HTTP-відповідь для користувача.

2.6 Висновки до другого розділу

У другому розділі було детально розглянуто теоретичні засади функціонування системи надійного зберігання даних на основі OpenStack Object Storage. Встановлено, що модель даних Swift ґрунтується на трирівневій ієрархії: облікові записи, контейнери та об’єкти. Така логічна структура забезпечує масштабоване, організоване та кероване зберігання великої кількості неструктурованих даних.

Особливу увагу приділено архітектурі Swift, яка реалізована як повністю розподілена система без єдиної точки відмови. Це дозволяє досягати високої доступності й надійності, забезпечуючи автоматичне відновлення при збої окремих вузлів. Проксі-сервер виступає в ролі єдиного інтерфейсу взаємодії з клієнтом, маршрутизуючи запити та координуючи їх обробку з іншими компонентами кластеру.

Було проаналізовано принципи функціонування механізмів зберігання, зокрема процесів реплікації, аудиту, оновлення та узгодженості, які забезпечують цілісність даних навіть у разі пошкодження носіїв або втрати зв’язку. Важливим компонентом цієї системи є структура кілець, що базується на модифікованому алгоритмі консистентного хешування з використанням множинних маркерів, що дозволяє гнучко масштабувати систему та ефективно балансувати навантаження між накопичувачами.

Окремо розглянуто проблематику масштабування Swift, включно з викликами, пов’язаними з ребалансуванням даних під час зміни топології кластера. Запропоновано практичні підходи до автоматизації цього процесу, зокрема використання глобальних блокувань, розподілених сховищ та інструментів управління конфігурацією, що сприяє підвищенню ефективності адміністрування.

Таким чином, OpenStack Swift демонструє надійну і масштабовану архітектуру для обробки великих обсягів об’єктних даних, що робить його доцільним вибором для побудови сучасних хмарних систем зберігання.

3 РОЗРОБЛЕННЯ АЛГОРИТМІВ, ТЕХНОЛОГІЙ ТА ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Архітектура системи надійного зберігання даних

Система надійного зберігання даних побудована на основі OpenStack Object Storage (Swift) та інтегрується з іншими компонентами хмарної платформи OpenStack для забезпечення високого рівня надійності, безпеки, масштабованості та зручності у використанні. Архітектура системи складається з декількох логічних рівнів, кожен з яких виконує окремі функції [26,27]. На рисунку 3.1 приведена схема взаємодії користувача з системою.

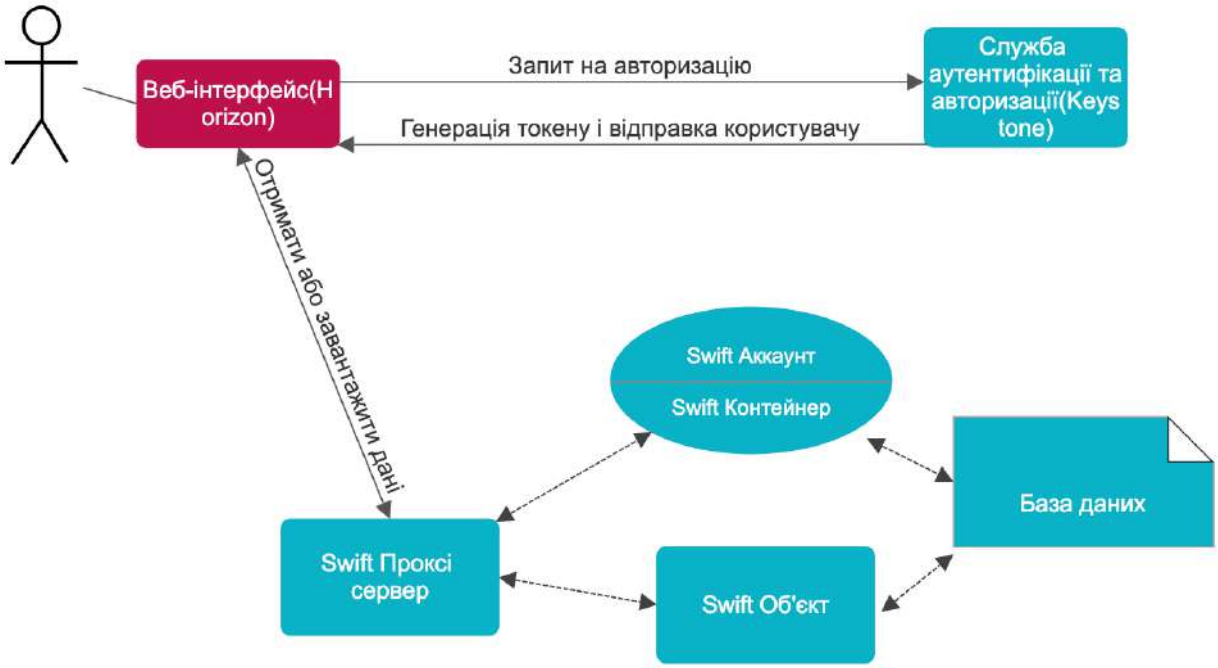


Рисунок 3.1 – Взаємодія користувача з системою зберігання даних на базі OpenStack Object Storage

На клієнтському рівні реалізовано взаємодію кінцевих користувачів або зовнішніх систем із сховищем даних [28]. Забезпечено кілька способів доступу:

OpenStack Horizon – веб-інтерфейс адміністративного та користувацького доступу до сховища. Horizon дозволяє виконувати базові операції над об’єктами, створювати контейнери, переглядати статус сервісів тощо.

RESTful API – інтерфейс прикладного програмування, що дає змогу інтегрувати систему з іншими застосунками або автоматизувати взаємодію зі сховищем.

Клієнтські бібліотеки – зокрема, Python SwiftClient та командна оболонка (CLI), які дозволяють програмно керувати об’єктами, контейнерами та авторизацією в системі.

Для реалізації контролю доступу до об’єктного сховища використовується компонент Keystone централізована служба аутентифікації та авторизації в OpenStack який включає

– підтримка багатокористувацького середовища шляхом створення проектів, ролей та облікових записів користувачів;

– видача токенів доступу, які використовуються для подальшої взаємодії клієнтів із сервісом Swift

– можливість інтеграції з зовнішніми системами аутентифікації (LDAP, SSO).

Серверний рівень (OpenStack Swift) відповідає безпосередньо за збереження та обробку об’єктів даних. Проксі сервер проміжний сервер, що приймає HTTP-запити від клієнтів, виконує аутентифікацію через Keystone та маршрутизує запити до відповідних вузлів зберігання. Фізичні або віртуальні сервери, які безпосередньо зберігають об’єкти (файли), контейнери та метадані розміщуються на вузлах зберігання. Система реплікації та контролю цілісності (Replication & Consistency) - забезпечує автоматичне створення копій об’єктів на декількох вузлах, а також перевірку та відновлення даних у разі їх пошкодження чи втрати [29].

Для забезпечення високого рівня керованості, спостережуваності та захисту даних в архітектуру включено також сервіси моніторинг і сповіщення, а також шифрування данних.

За допомогою систем Prometheus та Grafana реалізовано збір метрик, побудову графіків, контроль стану сервісів та сповіщення у разі інцидентів, а шифрування даних реалізується через сервіс Barbican (керування секретами в

OpenStack) для збереження та управління ключами шифрування, що забезпечує конфіденційність і захист інформації.

3.2 Розробка апаратно-мережевої архітектури

Ефективне розгортання системи надійного зберігання даних на базі OpenStack Object Storage вимагає ретельно спланованого вибору апаратного забезпечення та побудови відповідної мережевої інфраструктури. Як і у більшості комплексних рішень, для забезпечення стабільної роботи та оптимальної продуктивності кластер Swift повинен бути адаптований під конкретні потреби й навантаження [30, 31].

Однією з ключових переваг Swift є його сумісність із широким спектром апаратних конфігурацій – від недорогого загальнодоступного обладнання до рішень рівня enterprise. Це дозволяє уникати прив'язки до конкретного постачальника, оперативно оновлювати компоненти інфраструктури та зменшувати витрати без необхідності повної модернізації системи.

Система надійного зберігання даних на базі OpenStack Object Storage працює на 64-бітній архітектурі x86 та спроєктована для використання з апаратним забезпеченням загального призначення (commodity hardware). Буде використовувати використовувати продуктивні процесори з тактовою частотою в діапазоні 2–2.5 ГГц, особливо для вузлів, які обробляють проксі-запити.

Процесорна потужність критична насамперед для проксі-вузлів, але також має значення для об'єктних вузлів, якщо вони мають високу щільність накопичувачів [32, 33].

Кількість ядер було розрахована за формолою:

$$cores = \frac{k * \frac{1}{2} \frac{c * GHz}{d}}{GHz} \text{ z}, \tag{3.1}$$

де k – максимальна кількість дисків в серверному шасі;

GHz – такотова частота процесора, співідошення до ядра і диску 1:2;

d – об’єм одного диску.

Виходячи з цієї формули для системи з 18 жорстких дисків об’ємом 4 ТБ кожен, потрібно що найменше 9 ядер.

Окрім процесора, оперативна пам’ять системи кешує inodes файлової системи XFS та працює як буферний кеш для швидкої видачі часто доступних об’єктів.

Для оптимальної роботи було вирішено використовувавати 1 ГБ оперативної пам’яті на кожен диск. Здатність утримувати всі inodes у RAM забезпечить найвищу продуктивність. Відповідно до цієї логіки, чим більше inodes буде кешуватись, тим швидше працюватиме кластер. Стандартний розмір inode у XFS — 256 байт. Окрім об’єктів, inodes займають також hash_dir, suffix_dir та каталоги розділів на диску. Загалом, системі потрібно принаймні 1 КБ оперативної пам’яті на об’єкт.

Усі вузли потребують диска для операційної системи. Зазвичай використовується пара дисків, об’єднаних у RAID1, для операційної системи (ОС), щоб запобігти відмові всього вузла в разі виходу з ладу одного диска [34].

. При проєктуванні кластера важливо правильно визначити необхідний обсяг накопичувачів, їхню кількість, а також кількість серверів для розміщення цих дисків.

Розрахунок починається з визначення загального обсягу “сирого” сховища, який залежить від бажаного обсягу доступних даних і кількості реплік у кластері. З урахуванням накладних витрат на файлову систему.

Було визначенно, що загальний обєм кластеру має мати 400 ТВ дискового простору, розмір одного диска 4 ТВ.

Формула для розрахунку кількості жорстких дисків:

$$\left\lceil \frac{\text{Загальний об}'\text{єм}}{\text{Розмір диска}} \right\rceil = \text{Загальна кількість дисків.} \qquad (3.2)$$

Потрібно визначте кількість серверів, необхідних для вузлів зберігання, поділивши кількість жорстких дисків на кількість дискових відсіків у шасі.

$$\left\lceil \frac{\text{Загальна кількість дисків}}{\text{Кількість слотів в диску}} \right\rceil = \text{Кількість серверів.} \quad (3.3)$$

Отже, для системи надійного зберігання даних розміром 400 ТВ дискового простору потрібно 6 серверів, і загальною кількістю дисків 108.

Ефективне функціонування системи зберігання даних на базі OpenStack Object Storage значною мірою залежить від правильно спроєктованої та надійної мережевої інфраструктури. У процесі проєктування мережі було враховано інтеграцію з наявним мережевим середовищем, так і специфічні вимоги самої платформи Swift, яка реалізує розподілену архітектуру з численними вузлами. Кожен з цих вузлів взаємодіє з іншими за кількома логічно та фізично розділеними мережевими сегментами, кожен з яких має свою функціональну роль і технічні обмеження [35, 36].

Зовнішня мережа забезпечує доступ клієнтів та API до кластера. Через неї здійснюються всі запити до проксі-серверів, які, у свою чергу, спрямовують запити до інших вузлів. У випадку використання TLS (SSL), шифрування зазвичай завершується на проксі, а далі трафік передається незашифрованим по довіреній внутрішній мережі.

Зовнішній сегмент надійно захищений міжмережевими екранами, оскільки він взаємодіє з відкритим Інтернетом або з клієнтською частиною внутрішньої мережі.

Кластерна мережа є основним каналом для обміну даними між проксі-вузлами та вузлами облікових записів, контейнерів і об’єктів [32]. Саме ця мережа піддається найбільшому навантаженню у випадку великих обсягів записів. Наприклад, при записі 100 МБ об’єкта, проксі приймає його через зовнішній інтерфейс, а потім передає трьома потоками до об’єктних вузлів, що фактично створює навантаження в трикратному розмірі від початкового. Саме кластерна

мережа часто виявляється першим «вузьким місцем» у високонавантажених конфігураціях. Усі вузли мають бути взаємодоступними на рівні L2 або L3, а комунікація між стійками забезпечується через ToR (Top-of-Rack) комутатори, які додатково об’єднані через spine-комутатори.

Позасмугова мережа керування (out-of-band management) призначена для адміністрування серверів навіть у випадку втрати доступності основної мережі. Ця мережа базується на апаратних засобах керування, таких як IPMI, iLO, DRAC [].

Конфігурація передбачає використання інтерфейсів 10 GbE для зовнішнього, кластерного та реплікаційного трафіку, а також 1 GbE для out-of-band і керуючих підключень. Налаштована агрегація кількох сегментів через VLAN поверх одного NIC, об’єднаних через bonding.

Значну увагу було приділено пропускній здатності інтерфейсів. Система має обслуговувати до 100 запитів на секунду, кожен обсягом 1 МБ, сумарне навантаження на зовнішню мережу становитиме 100 МБ/с або приблизно 0.8 Гбіт/с. Хоча технічно цього можна досягти через канал у 1 Гбіт/с, було враховувано накладні витрати TCP, супутній трафік (запити автентифікації, моніторинг, оновлення ОС тощо), саме тому 10 GbE мережа розглядається як базовий стандарт для системи надійного зберігання, що забезпечує масштабованість та готовність до зростання навантажень [37, 38].

Побудова мережевої інфраструктури для Swift базується на принципах функціонального розподілу, ізоляції, резервування та масштабованості. Всі ці фактори критично важливі для досягнення високої доступності, цілісності даних та стабільної продуктивності системи зберігання в умовах реального виробничого середовища.

3.3 Розробка алгоритмів доступу та зберігання даних

Система зберігання даних, реалізована на основі OpenStack Swift, потребує чітко продуманих алгоритмів, які забезпечують не лише надійність і відмовостійкість, а й ефективне використання ресурсів [68, 69]. У цьому підрозділі

розглянуто основні підходи до збереження цілісності даних - реплікацію та ерозійне кодування, які застосовуються для мінімізації ризику втрати даних.

Існують дві ситуації, які можуть впливати на доступність даних:

– постійна втрата даних;

– тимчасова недоступність.

Тимчасова недоступність може бути спричинена різними факторами, такими як розриви з'єднань або збої серверів. У цій роботі термін недоступність використовується для опису ситуацій, коли дані повністю втрачені і недоступні.

Нехай p – це загальна доступність даних, a – (дельта) - середня доступність одного диска. Для спрощення аналізу припускаємо, що доступність кожного диска є незалежною від інших.

Найбільш прямолінійний підхід – реплікація. Swift за замовчуванням зберігає три копії кожного об'єкта на різних вузлах зберігання. Це дозволяє системі миттєво перемикатися на іншу копію в разі втрати або збою одного з серверів.

Алгоритм працює так:

– коли об'єкт завантажується, Proxy Server обирає три різні вузли зберігання відповідно до хеш-кільця;

– кожна з копій записується окремо, незалежно одна від одної;

– при читанні клієнт отримує копію з найшвидшого вузла.

Нехай k – це фактор реплікації, тобто кількість копій. Тоді загальна доступність обчислюється за формулою:

$$p = 1 - (1 - a)^k. \quad (3.4)$$

Іншим підходом є система ерозійного кодування, у цьому підході (k, m) вихідні дані діляться на k фрагментів даних, а також створюються m паритетних фрагментів. Загальна кількість фрагментів: n = k + m.

Система здатна витримати втрату менше m фрагментів. Тобто, щонайменше k фрагментів необхідні для відновлення даних.

Припускаючи, що кожен фрагмент зберігається на окремому диску, загальна доступність обчислюється як сума ймовірностей втрати менш ніж m фрагментів:

$$p = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}\binom{n}{m}a^{n-m}(1-a)^m = \sum_{i=0}^{n}\binom{n}{i}a^{n-i}(1-a)^i. \qquad (3.5)$$

Одним із ключових параметрів при проектуванні системи зберігання є співвідношення між кількістю корисних даних та загальним обсягом використаного простору, тобто надлишковість або просторові витрати (space overhead). Цей показник безпосередньо впливає на економічність системи та її масштабованість.

У системах із реплікацією надлишковість визначається просто: вона залежить від кількості копій даних. Наприклад, при потрійній реплікації (3x) кожен об’єкт зберігається тричі, і тільки одна копія є «корисною», тобто ефективність становить лише 33%, а просторові витрати - 200%.

У випадку ерозійного кодування надлишковість визначається кількістю паритетних фрагментів відносно кількості фрагментів з даними.

Наприклад, код RS-(6,3) має надлишковість 50%, що означає ефективність зберігання на рівні 66.67%. Це у два рази краще, ніж при 3x реплікації з аналогічною стійкістю до втрати трьох вузлів.

Хоча збільшення кількості паритетних фрагментів покращує відмовостійкість. Це також впливає на ускладнення інфраструктури. Наприклад, код RS-(12,6) має ту ж саму надлишковість (50%), що й RS-(6,3), але потребує збереження фрагментів на більшій кількості дисків. Це, у свою чергу:

- збільшує імовірність збоїв (оскільки більше вузлів залучено);
- підвищує мережеве навантаження;
- може викликати затримки при декодуванні об’єктів, особливо якщо фрагменти зберігаються в різних дата-центрах або географічних локаціях.

Таблиця 3.1 – Порівняння доступності та надлишковості для схем реплікації та ерозійного кодування

Схема	Максимальна кількість відмов	Доступність	Просторові витрати
Без захисту	0	99.5%	0%
2x реплікація	1	99.9975%	100%
3x реплікація	2	99.99999%	200%
RS-(4,2)	2	99.99975%	50%
RS-(6,3)	3	99.99999%	50%
RS-(10,2)	2	99.99734%	20%

Отже, як видно з порівняння, ерозійне кодування дозволяє суттєво зменшити обсяг надлишкових даних, не поступаючись при цьому у доступності. Це особливо важливо для систем із великим обсягом даних, де простір є цінним ресурсом. Саме тому в обраній системі зберігання реалізовано політику зберігання з використанням Reed-Solomon кодів, з оптимальним балансом між ефективністю, відмовостійкістю та продуктивністю.

3.4 Розробка алгоритмів для забезпечення високої пропускної здатності

Система надійного зберігання даних на основі OpenStack Object Storage буде використовуватись користувачами для завантаження та отримання програмного забезпечення, великих наборів даних, образів віртуальних машин (VM) та конфігураційних файлів. Залежно від того, звідки буде здійснюватись доступ до кластера, його використання можна класифікувати на два сценарії, як показано на рисунку 3.2. Ці два сценарії позначені червоним (пунктирна лінія) та зеленим (суцільна лінія) [65, 67].

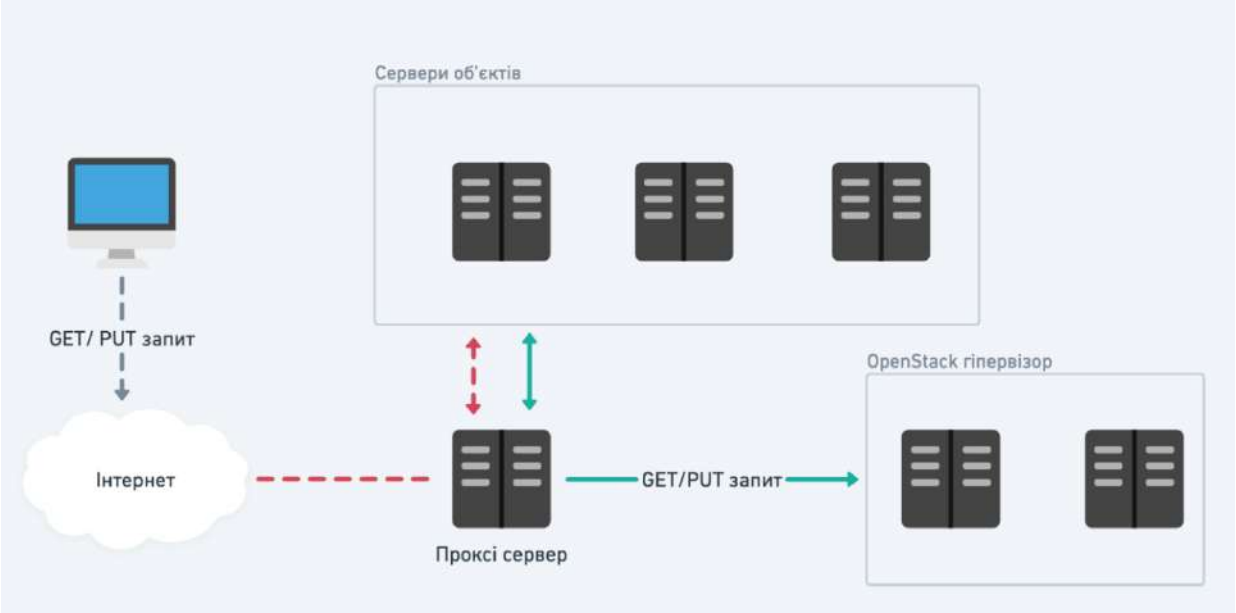


Рисунок 3.2 – Сценарій використання системи зберігання даних

Перший сценарій передбачає, що користувач звертається до кластера ззовні, через Інтернет, тобто не з локальної мережі. Другий сценарій стосується доступу до кластера зсередини локальної мережі, зазвичай, це bare-metal вузли обчислень OpenStack або віртуальні машини, які працюють на цих вузлах. Другий сценарій є більш типовим, оскільки більшість користувачів використовують Swift для збереження та отримання файлів.

Як видно з рисунка, проксі-сервер виступає вузьким місцем для всіх запитів і обмежує загальну пропускну здатність кластера.

Було запропоновано використовувати проксі-сервер лише як сервер метаданих, а не як точку пересилання даних.

У цьому варіанті клієнт звертається до проксі за метаданими об’єкта (зокрема, про те, де він зберігається), а потім безпосередньо взаємодіє з об’єктними серверами через RDMA. Це дозволяє виконувати реплікацію паралельно, без маршрутизації через проксі. Такий підхід значно підвищує масштабованість системи та показано на рисунку 3.3.

Окрім завантаження та отримання об’єктів, інші запити в кластері стосуються керування контейнерами, обліковими записами та об’єктами. Такі операції не створюють значного мережевого чи дискового навантаження, оскільки

не передбачають передачу самих об’єктів, тобто операцій GET/PUT. Дослідження показують, що затримка при виконанні таких операцій становить менше однієї секунди [70].

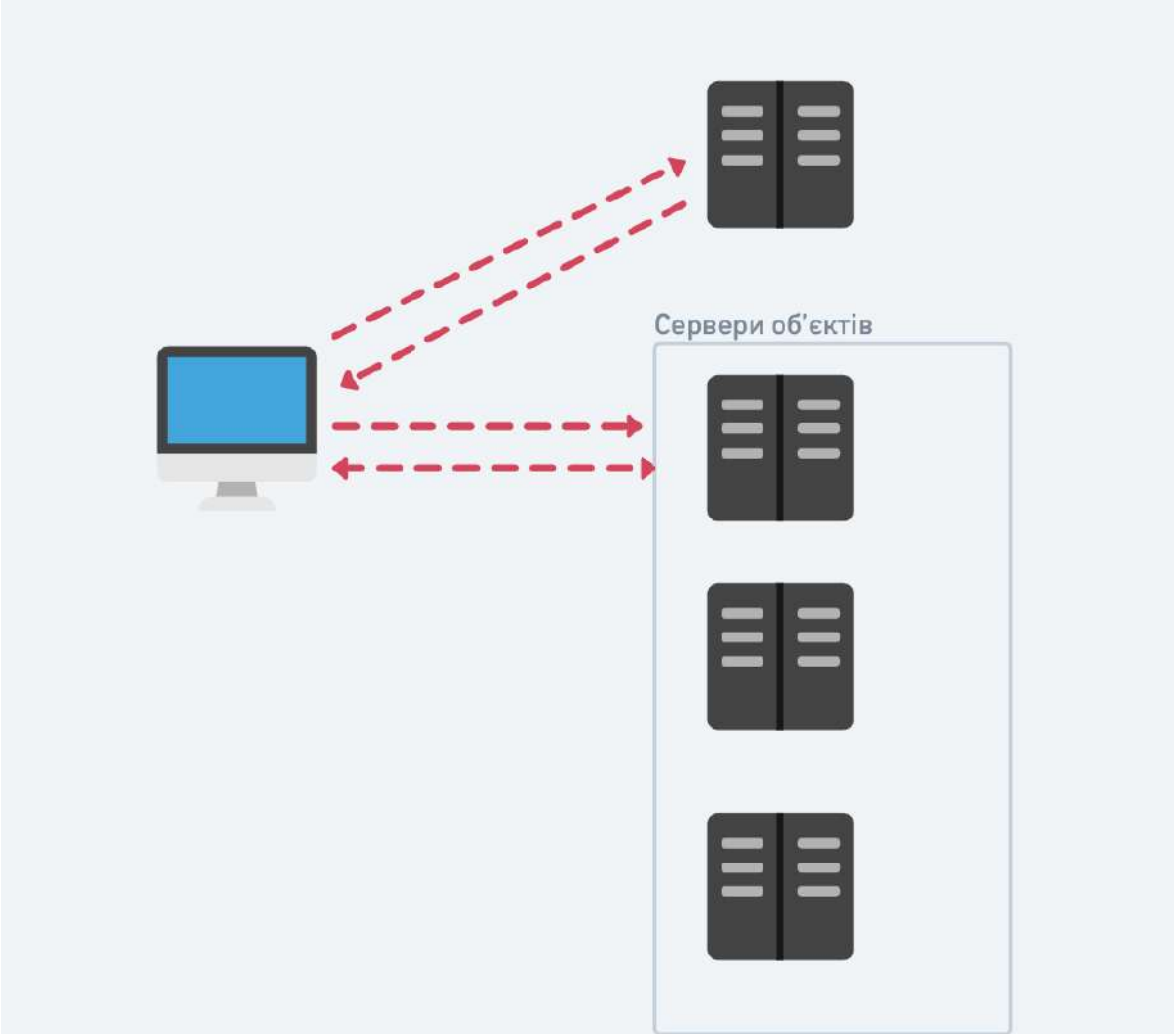


Рисунок 3.3 – Дизайн на основі сервера метаданих

Проте завантаження та отримання об’єктів, особливо великих, спричиняють суттєве навантаження на мережу та систему введення/виведення. Код Swift написаний мовою Python, а мережеві запити реалізовано через TCP-сокети. Як відомо, продуктивність Python нижча у порівнянні з іншими поширеними мовами програмування. Крім того, TCP має кілька відомих вузьких місць, зокрема контекстні перемикання та додаткові копії буферів для кожного повідомлення.

Для вирішення цієї задачі було використано переваги високошвидкісних мережевих інтерконектів, таких як InfiniBand

InfiniBand – це стандарт комп'ютерних мережевих комунікацій, який використовується у високопродуктивних обчисленнях для досягнення високої пропускної здатності та низької затримки. Цей високошвидкісний інтерконект загального призначення широко застосовується у суперкомп'ютерах по всьому світу.

Однією з ключових особливостей InfiniBand є технологія прямого доступу до пам'яті на відстані RDMA (Remote Direct Memory Access). RDMA дозволяє одному процесу читати або змінювати вміст пам'яті іншого віддаленого процесу без участі операційної системи або процесора на стороні отримувача.

InfiniBand забезпечує передачу даних з повним обходом ОС, тобто комунікація виконується в просторі користувача (userspace) та реалізується у форматі "нульового копіювання" (zero-copy). Усі протокольні обробки виконуються апаратно (hardware offload), що забезпечує високу продуктивність передавання даних [64].

Крім того, InfiniBand підтримує протокол IP over InfiniBand (IPoIB), який дозволяє запускати традиційні додатки на базі сокетів поверх InfiniBand-обладнання.

Оскільки мережеві операції та I/O становлять істотну частину часу при GET/PUT-запитах, це відкриває можливість використання семантики RDMA для прискорення цих операцій у Swift. Основне завдання полягає в розробці масштабованої комунікаційної інфраструктури на базі RDMA, яка не лише покращує мережеву взаємодію, а й дозволяє поєднувати її з I/O, зменшуючи загальний час обробки запиту.

Основна мета – знизити витрати часу на обчислення хешу, мережеву взаємодію та I/O, водночас покращуючи масштабованість і пропускну здатність системи надійного зберігання даних, зберігаючи той самий рівень відмовостійкості [31].

3.5 OpenStack Swift vs Azure Cloud

Бюджет відіграє важливу роль у стратегії LTDP. Він допомагає зрозуміти, наскільки економічно ефективно вибрати OpenStack Swift як систему надійного зберігання даних на основі OpenStack Object Storage у порівнянні з комерційними хмарами, такими як Azure. Microsoft Azure обрано з кількох причин. По-перше, Microsoft Azure надає інструмент під назвою Total Cost of Ownership (TCO), який дозволяє потенційним клієнтам порівняти витрати між локальними і хмарними рішеннями при міграції lift-and-shift. По-друге, інші великі постачальники хмарних послуг не мають інструменту, який можна порівняти з TCO Azure. По-третє, Кен Росенталь(2014) та Ренді Бредінг (2012) окремо проводили порівняння витрат між локальним зберіганням та зберіганням в AWS, тому корисно провести порівняння витрат, відмінне від AWS, щоб розширити наше розуміння витрат на хмарне зберігання [66].

З використанням калькулятора TCO ми ввели фактичні специфікації поточного OpenStack Swift у центрі обробки даних нашої організації. Це включає:

– шість ідентичних фізичних серверів для зберігання, кожен з 32 ГБ оперативної пам’яті, 2 процесори, 8 ядер на процесор, 2 потоки на ядро, що еквівалентно 32 процесорам;

– дві віртуальні машини 16 ГБ оперативної пам’яті та 8 процесорів для Swift Proxy та 4 ГБ оперативної пам’яті та 2 процесори для аутентифікації;

– об’єм пам’яті для зберігання на HDD (використана і невикористана): 432 ТБ;

– щомісячне вихідне використання мережі: 2 ТБ;

– відкрите програмне забезпечення, операційні системи віртуалізація; програмне забезпечення додатків.

На стороні Azure обрано локальне відмовостійке зберігання (LRS), а не географічне відмовостійке зберігання (GRS), щоб відобразити реалії локальної конфігурації. Інструмент TCO припускає, що не налаштовано резервне копіювання або відновлення після аварії. Основною метою є імітація міграції lift-and-shift з

локальної інфраструктури в хмару з використанням функціоналу Infrastructure as a Service (IaaS) [64].

Витрати розподіляються на п’ять категорій: обчислення, дата-центр, мережеві послуги, зберігання та IT-праця. Вартість обчислень додатково розкладається на апаратне забезпечення, програмне забезпечення, електрику та віртуалізацію. Загальні витрати, оцінені TCO протягом п’яти років для обох варіантів (локальний і Azure), наведені в стовпцях 2 і 3 таблиці 4.7.1. Для порівняння, фактичні витрати на нашому локальному обладнанні представлені в стовпці 4 таблиці 4.7.1 [37].

Щодо витрат на обчислення, за замовленням ми маємо вартість апаратного забезпечення $39,330.36, що є ціною фізичних серверів. Оскільки зазвичай для обслуговування цих серверів додається 15% до початкової ціни покупки, загальна вартість апаратного забезпечення за п’ять років складає $45,229.91. Для електрики припускається, що кожен фізичний сервер споживає 1000 Вт безперервно, що дає загальну вартість електрики за п’ять років у розмірі $26,280 за ціною $0.10/кВт·год, що трохи вище середньоринкової ставки. Оскільки використовуються безкоштовні операційні системи та програмне забезпечення, витрати на програмне забезпечення складають 0. Загальна вартість для обчислень складає $71,509.91, що є 63% від вартості обчислень в Azure і 23% від оцінки TCO для локальної конфігурації. Очевидно, що TCO значно завищує витрати на локальні обчислення, і витрати на обчислення в Azure вищі за фактичні локальні витрати.

Щодо витрат на зберігання, фактична вартість всіх HDD склала $26,772. Оскільки HDD схильні до частих збоїв, для обслуговування припускається витрати в 50% на 5 років, що дає додаткові $13,386. Загальні витрати на зберігання складають $40,158. Вартість зберігання в Azure за п’ять років складає $764,411.90, що значно дорожче за фактичні витрати. Оцінка TCO для локального зберігання становить $195,349.68, що складає 25% від вартості зберігання в Azure. Це також велике завищення порівняно з фактичними витратами.

Для витрат на мережу TCO припускає, що апаратні засоби фаєрволів, комутатори, маршрутизатори та інші мережеві компоненти є окремими для

OpenStack Swift. Однак насправді ці апаратні засоби розподіляються між багатьма іншими додатками в організації. У нашому кластері багато з цих компонентів надаються безкоштовно, тому припускається, що витрати на мережу однакові для обох варіантів.

Щодо витрат на IT-працівників, TCO недооцінює ці витрати. Це пов'язано з припущенням про низьку погодинну ставку і кількість годин роботи. У реальності системний адміністратор витрачає близько 15% своїх робочих годин на підтримку локальної системи OpenStack Swift, що призводить до витрат на IT-працівників в розмірі $70,500.00 за п'ять років.

Найбільш суперечливою та неоднозначною частиною витрат є витрати на дата-центр. TCO оцінює витрати на дата-центр за п'ять років у розмірі $874,406.70, з яких основна частина припадає на монтаж і встановлення серверів, що становить близько 96%. Важливо зазначити, що наш кластер використовує власні дата-центри, і ці витрати для них не існують. В результаті, для деяких організацій ця частина витрат не має сенсу.

Можна зробити обґрунтоване припущення, що:

– TCO зазвичай завищує витрати на обчислення та зберігання для локальних варіантів;

– витрати на обчислення та зберігання в Azure вищі за витрати на локальних системах;

– вартість зберігання в Azure є дуже високою для великих даних.

Для малих організацій, які не можуть дозволити собі підтримувати дата-центри, Azure Cloud може бути більш економічно ефективним. Однак для великих організацій, що мають можливість поділити витрати на дата-центр або отримують їх за рахунок материнської організації, локальні варіанти залишаються більш вигідними, особливо при роботі з великими наборами даних.

Таблиця 3.5 - Порівняння витрат між фактичним локальним сховищем Swift та відповідною системою, перенесеною в Azure Cloud

Категорія	Openstack Swift ($)	Azure Cloud ($)
Сервери	39330.36	0.00
Програмне забезпечення	0.00	0.00
Електрика	26280	0.00
Дата Центр	0.00	0.00
Мережа	1536.00	1536.00
Сховище	40 158.00	764 411.90
Працівники	70 500.00	19 609.11

3.6 Висновки до третього розділу

У третьому розділі було детально розглянуто архітектуру, алгоритми та технології, що використовуються для побудови системи надійного зберігання даних на основі OpenStack Object Storage. Наведено опис основних рівнів архітектури: клієнтського, серверного, рівня аутентифікації, моніторингу та шифрування, які взаємодіють між собою з метою забезпечення високої надійності, масштабованості та зручності експлуатації системи.

У рамках розробки алгоритмів зберігання даних було проведено порівняльний аналіз підходів реплікації та ерозійного кодування. Показано, що ерозійне кодування, зокрема із застосуванням кодів Ріда-Соломона, забезпечує ту ж саму або навіть вищу доступність даних при значно менших просторових витратах. Це робить його ефективним вибором для масштабованих хмарних сховищ.

Також у цьому розділі було проаналізовано вузькі місця типових реалізацій Swift, зокрема пов’язані із використанням TCP-протоколу, інтерпретованої мови Python та обмеженнями проксі-сервера. З метою підвищення пропускної здатності

системи було запропоновано вдосконалення. Запропоновані поліпшення включають:

– використання RDMA-комунікації через InfiniBand для зниження затримок;

– оптимізацію обчислення контрольної суми за допомогою xxHash замість MD5;

– перенесення логіки реплікації ближче до клієнта;

– реалізацію серверу метаданих замість класичного проксі-маршрутизатора.

Таким чином, розроблена система поєднує переваги OpenStack Swift як перевіреного рішення з сучасними підходами до підвищення продуктивності, що робить її придатною для використання у масштабованих та ресурсоємних середовищах.

4 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

4.1 Конфігурація продуктивного середовища

Конфігурація продуктивного середовища для розгортання OpenStack Swift в таблицях 4.1 і 4.2 Система складається з вузла автентифікації, проксі-вузла та шести вузлів зберігання. Ці шість вузлів розташовані в одному регіоні та розподілені між трьома фізично окремими дата-центрами. Кожен вузол знаходиться у власній зоні, що забезпечує високу доступність та відмовостійкість системи [38, 39].

Кожен вузол зберігання має такі характеристики:

– 32 ГБ оперативної пам’яті (RAM);

– 18 жорстких дисків об’ємом 4 ТБ кожен, що дає загальний обсяг 72 ТБ на один вузол;

– два процесорні сокети, кожен з вісьмома ядрами та двома потоками на ядро, що в сумі складає 32 логічних процесори.

Загальний обсяг сховища платформи складає 432 ТБ. Завдяки такій архітектурі система може витримати відмову двох дата-центрів одночасно для операцій читання та одного дата-центру для операцій запису. Якщо ж відмова відбувається на рівні зони, система здатна функціонувати навіть при виході з ладу до чотирьох зон для читання і до трьох зон для запису.

Ролі кожного вузла були визначені відповідно до їх призначення в архітектурі системи та вимог до навантаження. Вузол автентифікації відповідає за обробку запитів ідентифікації користувачів, проксі-вузол – за маршрутизацію запитів до об’єктного сховища, а вузли зберігання – за фізичне збереження об’єктів даних. Такий розподіл дозволяє досягти оптимального балансування ресурсів і забезпечити масштабованість приватної хмарної інфраструктури. Кожен компонент системи виконує чітко визначену функцію, що спрощує адміністрування кластера та підвищує його загальну надійність [40,41].

Таблиця 4.1 – Назви, ролі, типи та виділені обчислювальні ресурси для кожного обчислювального вузла, що складає приватну хмару сховища OpenStack Swift

Назва	Роль	Пам’ять	ЦПУ	Диск	Сховище	Зона	Локація
auth	Аутентифікація	4Гб	2	N/A	N/A	N/A	N/A
proxy	Swift проксі	8Гб	8	N/A	N/A	N/A	N/A
snode01	Об’єктне сховище	32ГБ	32	18	72Тб	Зона 1	ДЦ 1
snode02	Об’єктне сховище	32ГБ	32	18	72Тб	Зона 2	ДЦ 1
snode03	Об’єктне сховище	32ГБ	32	18	72Тб	Зона 3	ДЦ 2
snode04	Об’єктне сховище	32ГБ	32	18	72Тб	Зона 4	ДЦ 2
snode05	Об’єктне сховище	32ГБ	32	18	72Тб	Зона 5	ДЦ 3

Для забезпечення коректної роботи всіх компонентів системи було налаштовано окремі мережеві параметри для кожного вузла [42,43]. Кожен сервер отримав унікальну IP-адресу в межах виділеної підмережі та має спільний шлюз для виходу у зовнішні мережі. Розподіл мережевих налаштувань забезпечує ефективну взаємодію між вузлами та оптимізує маршрутизацію запитів у кластері. Для кожного типу сервісу виділено окремі порти, що мінімізує ризик конфліктів і підвищує безпеку системи. У таблиці нижче наведено перелік IP-адрес, запущених сервісів та відповідних портів для кожного вузла.

Важливим аспектом є також ізоляція службового трафіку зберігання від зовнішнього трафіку користувачів, що покращує продуктивність і знижує затримки. Такий підхід дозволяє гнучко масштабувати мережу в майбутньому без

суттєвих змін у поточній конфігурації. Детальну мережеву конфігурацію показано у додатку Б.

Таблиця 4.2 - Мережеві налаштування та сервіси, що працюють на кожному вузлі

Назва	IP адреса	Шлюз	Сервіси	Порт
auth	192.168.1.10/24	192.168.1.1	httpd, MySQL	5000
proxy	192.168.1.11/24	192.168.1.1	Проксі сервіс	8080
snode01	192.168.1.12/24	192.168.1.1	Акаунт, контейнер, сервіс об’єктів	6000,6001,6002
snode02	192.168.1.13/24	192.168.1.1	Акаунт, контейнер, сервіс об’єктів	6000,6001,6002
snode03	192.168.1.14/24	192.168.1.1	Акаунт, контейнер, сервіс об’єктів	6000,6001,6002
snode04	192.168.1.15/24	192.168.1.1	Акаунт, контейнер, сервіс об’єктів	6000,6001,6002
snode05	192.168.1.16/24	192.168.1.1	Акаунт, контейнер, сервіс об’єктів	6000,6001,6002

Вузли зберігання базуються на комерційних серверах Supermicro. Сервери мають 24 слоти для дисків у передній частині корпусу та 12 додаткових слотів у задній частині, приклад такого серверу показанно на рисунку 4.1. Розширення дискового простору може здійснюватися шляхом заповнення всіх слотів або заміни існуючих дисків на нові з більшою ємністю. Додатково, можливе підключення SAS3-експандера для збільшення кількості дискових накопичувачів. Якщо виникає необхідність у збільшенні ресурсів системи, можна придбати та додати нові вузли сховища [44].

Об’єм оперативної пам’яті (32 ГБ) та обчислювальна потужність (32 CPU) забезпечують стабільну роботу фонових сервісів, відповідальних за перевірку автентичності, цілісності та узгодженості даних у кластері.

Рисунок 4.1 - Сервер Supermicro з 36 слотами для дисків, 32 процесорами (CPU) та 32 ГБ оперативної пам’яті (RAM)

Для всіх вузлів використовується операційна система CentOS, яка є стабільним та безкоштовним дистрибутивом Linux із широкою підтримкою спільноти. OpenStack Swift сумісний з різними дистрибутивами, такими як RedHat, Debian, Ubuntu, openSUSE та SUSE Enterprise. Це гарантує відсутність проблем із застаріванням програмного забезпечення та полегшує його підтримку [45].

Процес встановлення OpenStack Swift добре документований у офіційній документації, яка регулярно оновлюється відповідно до випусків нових версій

OpenStack (релізи виходять кожні шість місяців). Також доступні інструкції з встановлення попередніх версій.

Загалом, існують три основні типи встановлення OpenStack Swift:

– встановлення проксі-вузлів (Proxy Nodes);
– встановлення вузлів зберігання (Storage Nodes);
– встановлення вузлів автентифікації (Authentication Nodes).

Для спрощення, автоматизації та стандартизації процесу розгортання була розроблена використана систе скриптів на базі Ansible. Ansible дозволяє автоматизувати розгортання та налаштування всіх компонентів системи, що значно знижує ймовірність помилок при встановленні та конфігурації.

4.2 Автоматизація розгортання за допомогою Kolla-ansible

Для розгортання хмарної інфраструктури OpenStack у рамках даного проєкту було використано Kolla Ansible [46,47] – інструмент, що забезпечує автоматизоване та контейнеризоване розгортання компонентів OpenStack.

Проєкт Kolla має на меті надати готові до використання в промисловій експлуатації контейнери та засоби розгортання для побудови хмар на базі OpenStack. Kolla Ansible є досить жорстко структурованим за замовчуванням, однак дозволяє повну кастомізацію, що робить його гнучким інструментом для адміністраторів.

Його особливістю є здатність забезпечити швидке початкове розгортання навіть користувачам із мінімальним досвідом, з можливістю подальшого поглибленого налаштування відповідно до специфічних потреб інфраструктури. Kolla Ansible буде встановлена на проксі сервері який буде виконувати роль серверу налаштування.

Для подальшої роботи необхідно встановити саму платформу Kolla Ansible та її залежності [48, 49]. Нижче наведено послідовність дій зі встановлення та первинного налаштування які будуть виконуватись на проксі-сервері згідно з офіційною документацією:

Перед початком встановлення переконуємось, що обрана операційна система підтримується, а також наявні необхідні компоненти, такі як Python 3, pip, Ansible, Docker або Podman.

Оновлюємо системні пакети.

yum update -y && yum upgrade

Далі потрібно встановити всі потрібні python залежності

sudo dnf install git python3-devel libffi-devel gcc openssl-devel python3-libselinux

Встановлюємо kolla-ansible

pip install git+https://opendev.org/openstack/kolla-ansible@master

sudo mkdir -p /etc/kolla

sudo chown $USER:$USER /etc/kolla

cp -r /var /venv/share/kolla-ansible/etc_examples/kolla/* /etc/kolla

cp /path/to/venv/share/kolla-ansible/ansible/inventory/all-in-one .

kolla-ansible install-deps

Перехедимо до встанловлення Openstack Swift, перед вставновленням потрібно згенерувати усі пароль та згенерувати ssl сертифікти.

kolla-genpwd

kolla-ansible -I multinode certification

Наступним обов’язковим кроком є налаштування конфігураційних файлів, які визначають параметри розгортання: globals.yml і multinode. Ці файли є основою для правильного функціонування системи, оскільки вони описують як фізичну структуру кластера, так і логіку його розгортання.

Файл globals.yml містить глобальні змінні, які впливають на всі компоненти під час розгортання. Це головний конфігураційний файл Kolla Ansible [50]

Файл multinode є інвентарним файлом Ansible, який описує, на яких вузлах будуть розгортатися відповідні ролі (сервіси OpenStack). Це стандартний YAML-файл, сумісний із Ansible [51], який дозволяє Kolla визначити які вузли беруть участь у розгортанні, які саме сервіси будуть запущені на кожному з них.

Для роботи служби об’єктного зберігання OpenStack Swift необхідно попередньо підготувати окремі блочні пристрої на кожному вузлі зберігання. Усі диски мають бути відформатовані у файлову систему XFS та позначені спеціальним розділом з міткою KOLLA_SWIFT_DATA. Ця мітка використовується Kolla Ansible для автоматичного виявлення пристроїв, придатних до використання у Swift.

```
#/bin/bash
# excluding system disks sda and sdb
DISKS=$(lsblk -dno NAME | grep -E '^sd' | grep -Ev '^sda$|^sdb$' | sort)
# in case of nmve
#DISKS=$(lsblk -dno NAME | grep -E '^sd' | sort)
index=0
for d in $DISKS; do
   echo "Formatting /dev/${d} → label: d${index}"
   parted /dev/${d} -s -- mklabel gpt mkpart KOLLA_SWIFT_DATA 1 -1
   sudo mkfs.xfs -f -L d${index} /dev/${d}1
   (( index++ ))
done
```

У типовому серверному шасі присутні 18 дисків. Системні диски (sda, sdb) виключаються з ініціалізації, оскільки містять операційну систему. На всіх інших пристроях створюється одна GPT-розмітка, один розділ і файлову систему з міткою у форматі d0, d1, d2 тощо.

Наступний крок – сказати операційній системі змонтувати нові файлові системи XFS у певні місця на пристроях, щоб Swift міг їх знайти та почати записувати дані.

Створюємо директорію у /srv/node/ для кожного пристрою як місце для монтування файлової системи.

```
mkdir -p /srv/node/d0
mkdir -p /srv/node/d1
mkdir -p /srv/node/d2
```

mkdir -p /srv/node/d1

Для кожного пристрою (dev), який хочемо змонтувати, використовуємо команду mount та запускаємо її для кожного пристрою:

mount -t xfs -o noatime,nodiratime,logbufs=8 -L /dev/sda1 /srv/node/d0

Після монтування дисків потрібно створити користувача з правами читання/запису до директорій, де змонтовані пристрої. Стандартний користувач Swift має ім’я swift. Створюємо користувача та надаємо йому права власності на директорії.

useradd swift

chown -R swift:swift /srv/node

Далі необхідно налаштувати політики зберігання, які визначають, як дані будуть розміщуватися всередині системи. Політики зберігання дозволяють створювати окремі області в кластері з різними характеристиками, такими як рівень реплікації, вибір обладнання, фізичне розташування даних або налаштування партицій. Це забезпечує гнучкість у відповідності до специфічних вимог до зберігання.

Щоб налаштувати політики зберігання, потрібно відредагувати конфігураційний файл swift.conf, який знаходиться у каталозі /etc/swift/. Параметри політик слід розміщувати після секції [swift-hash]. Кожна політика описується за допомогою окремого блоку [storage-policy:N], де N — унікальний індекс політики (починаючи з 0).

Кожен блок політики обов’язково має містити ім’я політики (name =). За бажанням можна вказати, чи є політика за замовчуванням (default = yes) або позначити її як застарілу (deprecated = yes). Індекс політики повинен бути унікальним і незмінним після створення, а ім’я повинно складатися лише з букв, цифр або дефісів [52].

[swift-hash]

swift_hash_path_prefix = changeme

swift_hash_path_suffix = changeme

[storage-policy:0]

name = level1

[storage-policy:1]

name = level2

default = yes

[storage-policy:2]

name = level3

deprecated = yes

Політика з індексом 0 є обов’язковою і використовується за замовчуванням, якщо не задано іншу політику. Це також гарантує зворотну сумісність для кластерів, створених до впровадження політик зберігання у Swift.

Якщо необхідно змінити політику за замовчуванням, достатньо відредагувати відповідний блок і вказати default = yes. При цьому важливо оцінити можливий вплив на існуючі дані та роботу системи.

Позначення політики як застарілої (deprecated = yes) дозволяє підтримувати існуючі контейнери, але забороняє створення нових контейнерів із цією політикою. Це корисно для поступового виведення старих або менш ефективних конфігурацій без порушення існуючих даних.

Після оголошення політик потрібно створити відповідні файли ініціліазації для кожної політики. Для кожної оголошеної політики створюється файл object-N.builder, де N — індекс політики. Наприклад, для політики storage-policy:1 буде створено файл object-1.builder.

Формування нових кілець на основі цих файлів дозволить системі правильно розподіляти об’єкти згідно з обраною політикою зберігання. Код формування кілець показано у додатку В.

Після базового налаштування та підготовки інвентарного файлу, наступним кроком є увімкнення та конфігурування служби об’єктного зберігання Swift. Для цього необхідно відредагувати конфігураційний файл globals.yml, який знаходиться у каталозі /etc/kolla/.

Правимо файл globals.yml і додаємо (або розкоментовуємо) такі рядки:

enable_swift: "yes"

swift_storage_interface: "eth0"

swift_replication_interface: "eth0"

swift_devices_match_mode: "strict"

swift_devices_name: "KOLLA_SWIFT_DATA"

Ці параметри активують компонент Swift, вказують, через який мережевий інтерфейс здійснюється доступ до дисків та реплікація, а також задають правила розпізнавання дисків. Зокрема, swift_devices_name має відповідати мітці (label) файлової системи на дисках, які буде використовувати Swift [53].

У даному випадку використовується режим strict, тобто будуть обрані тільки ті розділи, що мають точну мітку KOLLA_SWIFT_DATA. Це дозволяє уникнути помилкового використання системних або службових дисків.

Запускаємо розгортання. Розгортання може тривати до 15 хвилин поки система буде налаштована належним чином.

kolla-ansible deploy -i multimode

Після завершення процесу розгортання компонентів системи, необхідно переконатися, що всі сервіси функціонують коректно, кільця створено правильно, а вузли зберігання доступні. Це є критичним етапом, який дозволяє виявити потенційні помилки конфігурації або проблеми з доступністю ресурсів, перш ніж система буде введена в експлуатацію.

Перш за все слід впевнитися, що всі необхідні сервіси було успішно запущено у контейнерах. Для цього на одному з керуючих вузлів виконується команда:

docker ps -a | grep swift

Контейнери повинні перебувати у статусі Up, що свідчить про їх успішну ініціалізацію.

Останнім етапом перевірки є безпосереднє тестування можливості завантаження та зчитування об’єктів через CLI OpenStack. Для цього створюється контейнер та завантажується тестовий файл:

openstack container create test-container

openstack object create test-container test.txt

openstack object list test-container

openstack object save test-container test.txt

У разі успішного функціонування системи об’єктного зберігання, команда створення контейнера завершується без помилок, що свідчить про доступність сервісу swift-proxy. Завантаження файлу до контейнера виконується коректно, при цьому об’єкт зберігається в одному з доступних дисків відповідно до розрахованої конфігурації кільця. Подальше зчитування цього файлу підтверджує, що зберігання працює не лише на запис, але й на читання, і об’єкт повертається у своєму первісному вигляді, що свідчить про коректність розподілу, збереження та відтворення даних у системі [54].

4.2 Оновлення ПЗ та обслуговування

Загалом, обслуговування системи надійного зберігання даних на основі Openstack Object Storage потребує відносно невеликих зусиль. Його можна поділити на дві основні категорії: оновлення програмного забезпечення та апаратне обслуговування.

OpenStack Swift випускає нові версії кожні шість місяців. Версії, що на два-три цикли старші за останню стабільну, зазвичай не підтримуються – вони не отримують оновлень чи виправлень безпеки. Використання застарілої версії створює дві основні проблеми:

– підвищений ризик безпеки через відсутність патчів;

– складність оновлення до останньої стабільної версії, якщо поточна версія занадто стара.

Якщо в новій версії змінилася схема бази даних, оновлення може бути вкрай складним або навіть неможливим без проміжних версій. Тому кращою практикою є оновлення кластера відповідно до офіційного циклу випуску. Це також відповідає принципам довготривалого цифрового збереження (LTDP), допомагаючи уникнути застарівання програмного забезпечення.

Щоб спростити оновлення та підвищити надійність, використовуємо скрипти в Kolla Ansible [40], які автоматизують процес оновлення.

Другий аспект оновлення – це оновлення ядра операційної системи. Цей процес потребує поетапного перезавантаження всіх вузлів системи, щоб уникнути простою.

Оновлення з однієї основної версії ОС на іншу (наприклад, з CentOS 6 до CentOS 7) потребує тестування на відокремленому середовищі перед розгортанням у продакшн [55, 56]. Це помірно складний процес, що вимагає ретельного планування для запобігання втраті або недоступності даних.

Жорсткі диски (HDDs), що є основними носіями даних в OpenStack Swift, з часом зношуються та можуть виходити з ладу. Для проактивного виявлення проблем у кластері використовується скрипт cron, який щодня перевіряє кожен диск на некориговані (погані) сектори.

Якщо на диску з’являються дефектні сектори, вони автоматично маркуються як несправні, і файлова система більше не використовує їх для читання чи запису. Однак, якщо кількість таких секторів зростає надто швидко, це ознака наближення відмови диска.

Для візуалізації здоров’я дисків використовується теплова карта (heatmap) (Рисунок 4.2). Наприклад, на вузлі snode03 виявлено 1 пошкоджений сектор на диску /dev/sdm, тоді як snode06 має два диски з дефектами: /dev/sdk (296 пошкоджених секторів) та /dev/sdo (12 секторів).

Досвід показує, що якщо кількість пошкоджених секторів менше 100, диск може працювати ще довгий час. Але якщо їх кількість швидко зростає, краще замінити диск заздалегідь. Дійсно, диск із 296 пошкодженими секторами з рисунку 4.2 вийшов з ладу через кілька днів.

Дослідники Kamarthi et al. (2009) запропонували складніший метод моніторингу HDD, використовуючи три параметри:

- кількість пошкоджених секторів;
- помилки ECC (корекція помилок апаратного рівня);
- швидкість читання/запису.

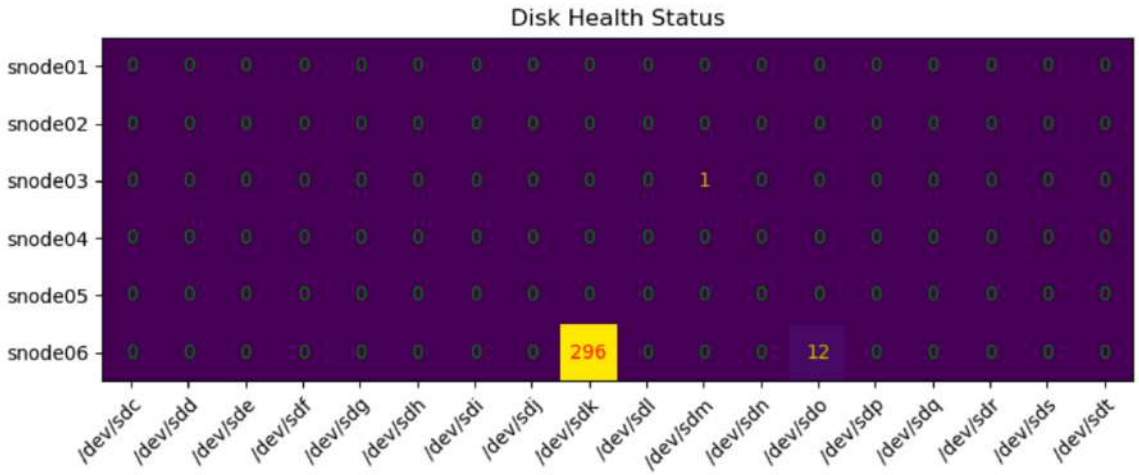


Рисунок 4.2 - Стан працездатності жорстких дисків

Вони створили нейронну мережу, яка оцінює залишковий термін служби HDD з точністю 88%. Хоча їхній підхід складніший, він підтверджує ефективність нашого методу моніторингу.

Вихід з ладу жорстких дисків є нормальним явищем. OpenStack Swift автоматично відновлює дані на нових дисках без простою системи чи втрати даних.

Заміна несправного диска виконується у кілька кроків:

– встановлення нового HDD;

– форматування його у XFS;

– монтування з правильними правами доступу.

Збережені на ньому цифрові об’єкти автоматично відновлюються із резервних копій на інших дисках.

Жорсткі диски та інші носії неухильно старіють, що може призвести до випадкового псування чи зміни цифрових даних. У OpenStack Swift оновлення носіїв виконується так само просто, як заміна диска, що вийшов з ладу.

Якщо відбувається випадкова зміна файлів, фонові процеси аудиту автоматично виявляють та виправляють помилки. Цей механізм був протестований та підтверджений у кластері.

4.3 Тестування системи

OpenStack Swift розроблений як масштабована система, здатна зберігати мільярди цифрових об’єктів і сотні петабайтів даних без значних збоїв. Для аналізу продуктивності завантаження та завантаження файлів з контейнерів з різною кількістю об’єктів було обрано контейнер CWRC (що містить 405 635 об’єктів) і порожній контейнер Demo [57, 58].

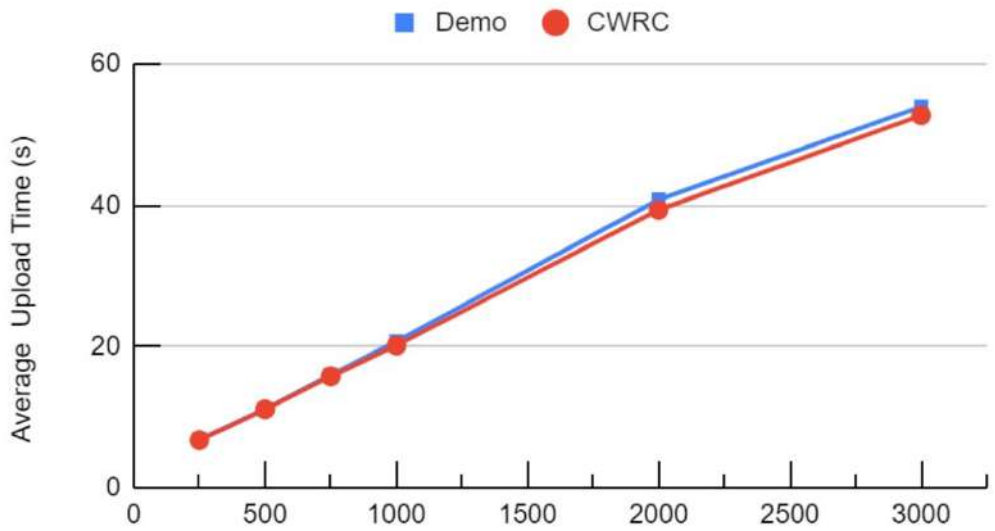


Рисунок 4.3 – Порівняння часу завантаження для об’єктів із різними розмірами у двох різних контейнерах

Для тестування використовувались шість об’єктів різного розміру: 250MB, 500MB, 750MB, 1GB, 2GB та 3GB.

Кожен об’єкт завантажувався в обидва контейнери та завантажувався з них по три рази, після чого визначався середній час виконання операцій для порівняння.

Результати тестування показано на рисунку 4.6.1 (час завантаження) та рисунку 4.6.2 (час вивантаження).

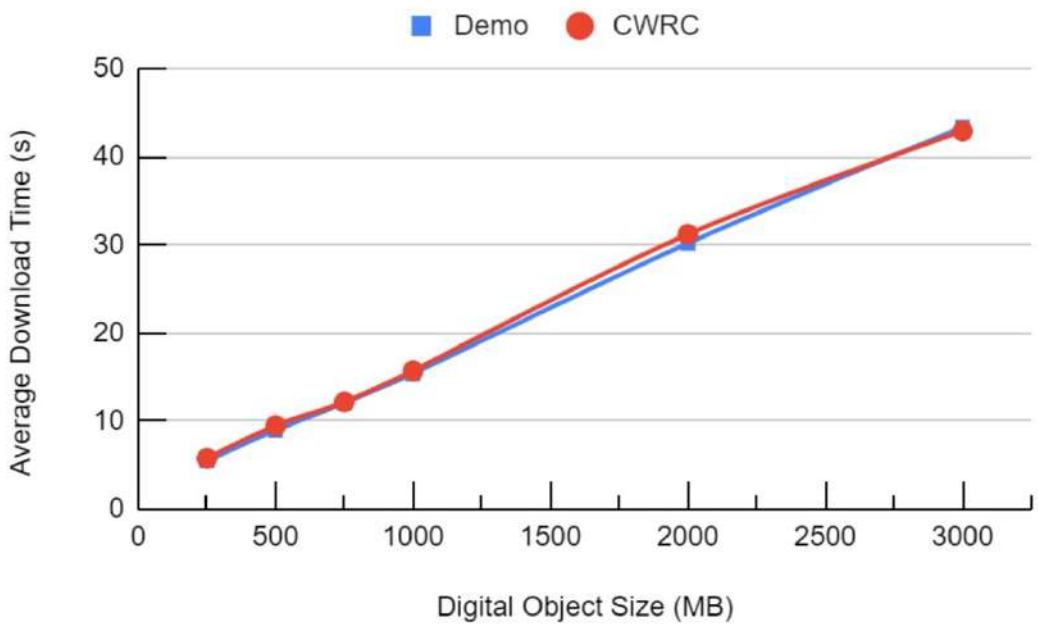


Рисунок 4.4 - Порівняння часу вивантаження для об’єктів із різними розмірами у двох різних контейнерах

Результати свідчать, що час завантаження та вивантаження практично не залежить від розміру контейнера: однаковий файл займає приблизно однаковий час для завантаження або вивантаження, незалежно від того, чи це великий контейнер CWRC чи порожній контейнер Demo. Зі збільшенням розміру файлу час операцій зростає майже лінійно.

У науковому середовищі дослідження продуктивності та масштабованості OpenStack Swift проводили Toor. (2012), використовуючи експериментальні дані Великого адронного колайдера (LHC) в Європейській організації з ядерних досліджень (CERN). LHC щорічно генерує близько 15 PB даних, що вимагає виняткової масштабованості як за обсягом збереження, так і за продуктивністю введення/виведення.

Toor ставили дві основні цілі:

– дослідити продуктивність системи при вилученні даних різного розміру блоків (4KB, 64KB, 1MB) із великого набору даних;

– виміряти продуктивність при зберіганні великої кількості об’єктів в одному контейнері.

Запропонована методика відрізняється від дослідження Toor. тим, що аналізувалося завантаження та вилучення повноцінних цифрових об’єктів при зміні їхнього розміру, а не розміру блоку даних. Для порівняння використовувалися два контейнери з різною кількістю об’єктів, тоді як у згаданому дослідженні був залучений лише один великий контейнер. Крім того, тестування виконувалося в мережевому середовищі з пропускною здатністю 1 Гбіт/с, що у 10 разів повільніше, ніж 10-гігабітна інфраструктура, використана у Toor [59, 60].

Попри ці відмінності, результати обох експериментів підтверджують високу масштабованість OpenStack Swift у роботі з великими наборами даних.

4.4 Оптимізація OpenStack Swift

Оптимізація продуктивності системи зберігання на основі OpenStack Swift є критично важливим етапом для досягнення високої ефективності, масштабованості та надійності кластеру. Завдяки широким можливостям конфігурації компонентів Swift — проксі-сервера, серверів облікових записів, контейнерів і об’єктів — адміністратор може детально налаштовувати поведінку системи відповідно до поточних навантажень і особливостей інфраструктури.

У Swift основну роботу виконують worker-процеси. Кожен сервер (proxy, account, container, object) передає обробку запитів окремим воркерам, які можуть паралельно обробляти багато запитів завдяки бібліотеці Eventlet. Eventlet реалізує неблокуючий ввід/вивід за допомогою “зелених потоків”, які споживають менше ресурсів у порівнянні з потоками операційної системи.

Зазвичай, кількість воркерів встановлюється відповідно до кількості ядер процесора, але це значення можна змінювати в конфігураційних файлах (proxy-server.conf, account-server.conf, container-server.conf, object-server.conf). Один worker здатен обробляти до 1024 запитів одночасно. Очікується, що система обслуговуватиме, 10 000 одночасних запитів, потрібно налаштувати щонайменше

10 воркерів. Таке масштабування дозволяє уникнути затримок, пов’язаних із блокуванням потоків при інтенсивній роботі з диском або мережею.

Особливу увагу було приділено проксі-серверу, адже саме він отримує запити від клієнтів і є центральною точкою маршрутизації. Його навантаження часто є процесорозалежним, тому налаштовано воркери із розрахунку один на кожне ядро CPU.

Сервери облікових записів, контейнерів та об’єктів мають більш складне завдання: вони не лише отримують мережеві запити від проксі, а й виконують операції з диском, зокрема читання та запис до баз SQLite чи самих об’єктів. Через це встановлено достатню кількість worker-процесів, щоби файлові I/O-операції не блокували Eventlet-потоки.

Розглянемо параметр threads-per-disk та об’єктні сервери. Для забезпечення високої паралельності об’єктного сервера було використано велика кількість worker-процесів. Починаючи з Swift 1.9, з’явилась можливість детальніше керувати навантаженням за допомогою параметра threads-per-disk. Цей параметр визначає, скільки потоків створюється для обслуговування кожного фізичного диска. Встановлено рекомендоване значення –4 потоки на диск.

Під час обробки запитів проксі-сервер і сервери об’єктів читають і записують дані блоками. Правильно підібраний розмір блоку дозволяє зменшити кількість системних викликів і покращити використання ресурсів мережі та CPU. Існує кілька типів блоків, які можна налаштувати.

Клієнтський розмір блоку (client chunk size) визначає, скільки даних проксі-сервер читає або записує від або до клієнта за один раз. Розмір блоку для об’єктів (object chunk size) регулює обсяг даних, який передається між проксі-сервером та об’єктними, контейнерними або обліковими серверами. Дисковий розмір блоку (disk chunk size) визначає обсяг даних, які сервер об’єктів читає з диска або записує на нього. Мережевий розмір блоку (network chunk size) визначає обсяг даних, який передається між сервером об’єктів і проксі-сервером.

Для внутрішньої мережі зберігання, яка підтримує Jumbo Frames і має високу пропускну здатність, встановлюються великі розміри блоків. Це дозволяє зменшити кількість мережевих операцій та знизити навантаження на процесор.

OpenStack Swift також містить низку фонових процесів (демонів), які забезпечують цілісність, узгодженість і актуальність даних. Аудитори (auditors) перевіряють збережені дані на наявність пошкоджень (bit rot), навіть якщо існують кілька реплік [61, 62]. Реплікатори (replicators) синхронізують репліки об’єктів, забезпечуючи поступову узгодженість системи. Оновлювачі (updaters) актуалізують метадані, включаючи кількість об’єктів та обсяг використаного простору. Процес Reaper остаточно видаляє об’єкти після того, як користувач запитав їх видалення.

Хоча ці процеси життєво важливі для стабільності системи, вони створюють значне навантаження на підсистему вводу/виводу. Тому існують параметри для обмеження їхньої активності:

– interval — визначає час у секундах (або мілісекундах) між виконанням наступного проходу;

– concurrency — кількість елементів, які обробляються одночасно;

– slowdown (для updaters) — визначає затримку між обробкою кожного елемента.

Правильне налаштування цих параметрів дозволяє уникнути пікового навантаження на CPU та диск, особливо в періоди активного запису або обробки великої кількості об’єктів [63].

Реплікаторам було встановлено параметри concurrency 8 і run_pause 30 секунд, що дозволяє контролювати інтенсивність процесу реплікації. Для об’єктного аудитора задано обмеження по кількості файлів, що перевіряються за секунду (files_per_second=10), зменшуючи навантаження на диск.

Reaper виконує видалення у фоновому режимі з високою інтенсивністю concurrency = 10, було зменшинно це значеннящоб уникнути пікових навантажень.

4.8 Висновки до четвертого розділу

У цьому розділі було детально описано процес побудови продуктивного середовища для системи зберігання даних на основі OpenStack Swift, реалізованого в межах приватної хмари. Продемонстровано, що обрана архітектура з розподіленням вузлів між трьома фізично ізольованими дата-центрами забезпечує високий рівень доступності, відмовостійкості та масштабованості.

Завдяки використанню автоматизованих інструментів, таких як Kolla Ansible, вдалося досягти стандартизованого, гнучкого та повторюваного процесу розгортання, що суттєво знижує ймовірність людських помилок та пришвидшує впровадження нових версій OpenStack.

Особлива увага приділялась оптимізації системи як на етапі налаштування мережевих параметрів, розмірів блоків і конфігурації worker-процесів, так і при регулюванні роботи фонових служб. Такий підхід дозволив збалансувати навантаження між компонентами системи, покращити час обробки запитів і мінімізувати затримки.

Проведене функціональне тестування підтвердило працездатність усіх компонентів Swift та відсутність втрат при операціях читання/запису навіть за умов значного навантаження. Крім того, експериментальні дані засвідчили ефективність оптимізаційних налаштувань, а також підтвердили масштабованість рішення.

Загалом, реалізоване рішення демонструє можливість побудови стійкої, ефективної та гнучкої системи об’єктного зберігання, яка може бути адаптована під потреби наукових, освітніх або комерційних установ, що працюють з великими обсягами даних.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було спроектовано і реалізовано систему надійного зберігання даних на основі OpenStack Object Storage.

У першому розділі проаналізовано архітектуру OpenStack Swift та його можливості для зберігання даних, сформульовані вимоги до розроблюваної РСЗД, зроблено докладний опис хмарної платформи OpenStack.

У другому розділі описано архітектуру і досліджено особливості конфігурації продуктивного середовища OpenStack Swift. Проаналізовано масштабованість, високу доступність компонентів Swift і процесу реплікації та узгодженості даних у кластер, спроектовано кластер сховища, що включає вузли автентифікації, проксі-сервери та сховища, розподілені між трьома дата-центрами.

У третьому розділі здійснено опис основних рівнів архітектури: клієнтського, серверного, рівня аутентифікації, моніторингу та шифрування, які взаємодіють між собою з метою забезпечення високої надійності, масштабованості та зручності експлуатації системи та проаналізовано вузькі місця типових реалізацій Swift.

У четвертому розділі проведено тестування продуктивності завантаження та отримання даних у контейнерах з різною кількістю об’єктів та проаналізовано підходи до обслуговування кластера, включаючи оновлення ПЗ та виявлення збоїв апаратного забезпечення.

Результати роботи підтвердили, що OpenStack Swift забезпечує масштабованість, стійкість до відмов та ефективну обробку великих обсягів даних. Використання автоматизації на базі Ansible значно спрощує обслуговування системи, зменшуючи ризики людських помилок.

У майбутньому планується подальша оптимізація продуктивності та розширення функціональних можливостей, зокрема впровадження додаткових стратегій балансування навантаження та покращення моніторингу стану системи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. The Most Widely Deployed Open Source Cloud Software in the World. URL: https://www.openstack.org (дата звернення: 24.03.2025).

2. Mell P., Grance T. The NIST Definition of Cloud Computing. Version 15, 10-7-09. URL: http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc (дата звернення 8.03.2025).

3. Raza M. Public Cloud Growth Trends and the Future Outlook. URL: https://www.bmc.com/blogs/cloud-growth-trends / (дата звернення 8.03.2025).

4. Bumgardner Cody K.V., Pipes Jay. OpenStack in Action: Manning: 2016. 384 с.

5. Antonopoulos N. Cloud Computing. Principles, Systems and Applications: New York: Springer-Verlag: 2010. 379 с.

6. Adkins S., Belamaric J., Giersch V. et al. OpenStack Cloud Application Development: John Wiley & Sons: 2015. 168 с.

7. Store Cluster Quick Start. Ceph Documentation. URL: http://ceph.com/docs/dumpling/start/quick-ceph-deploy/ (дата звернення 16.02.2025).

8. Latham R. Parallel I/O in practice. *Tutorial of SC15*. 2015.

9. Architecture. Ceph Documentation. URL: http://ceph.com/docs/master/architecture (дата звернення: 24.02.2025).

10. OpenStack Swift. Developer Documentation. URL: http://docs.openstack.org/developer/swift/ (дата звернення: 30.03.2025).

11. Arnold Joe. OpenStack Swift: O'Reilly Media: 2014. 338 с.

12. Trautwein D. Design and evaluation of IPFS a storage layer for the decentralized web. *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022. P. 739-752.

13. Bieri C. An overview into the InterPlanetary File System (IPFS) use cases, advantages, and drawbacks. *Communication Systems XIV*. 2021. 28.

14. Hildebrand D., Honeyman P. Exporting storage systems in a scalable manner with pNFS. *NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*. 2005.P.18-27.

15. Kapadia A., Varma S., Rajana K. Implementing Cloud Storage with OpenStack Swift Design. United Kingdom. Packt Publishing. 2014. 136 с.

16. Jackson Kevin. OpenStack Cloud Computing Cookbook. Packt Publishing. 2018. 398 с.

17. Hildebrand D. pNFS and Linux working towards a heterogeneous future. Center for Information Technology Integration. 2007.

18. Khedher O. Mastering OpenStack: Packt Publishing: 2015. 400 с.

19. Evaluating OpenStack Single-Node Deployment. URL: https://access.redhat.com/articles/1127153 (дата звернення: 30.03.2025).

20. Fifield Tom, Fleming Diane, Gentle Anne et al. OpenStack Operations Guide. O'Reilly Media. 2014. 330 с.

21. Radez Dan. OpenStack Essentials. Packt Publishing. 2016. 182 с.

22. Markelov A. Certified OpenStack Administrator Study Guide. Apress. 2016. 185 с.

23. Distributed storage performance for OpenStack clouds. URL: http://www.principledtechnologies.com/Red%20Hat/RedHatStorage_Ceph_1113.pdf (дата звернення: 30.03.2025).

24. Swift Overview. OpenStack Documentation. URL: https://docs.openstack.org/swift/latest/ (дата звернення: 30.03.2020).

25. OpenStack Swift Large Object Support. OpenStack Documentation. URL: https://docs.openstack.org/swift/latest/overview_large_objects.html (дата звернення: 10.02.2025).

26. Ceph Storage Architecture. Ceph Documentation. URL: https://ceph.com/ceph-storage/ (дата звернення: 12.02.2025).

27. Ceph Object Storage Features. Ceph Documentation.URL: https://ceph.io/en/features/ (дата звернення: 12.02.2025).

28. Linux Kernel Documentation – Filesystems. Linux Kernel Documentation. URL: https://www.kernel.org/doc/html/latest/filesystems/index.html (дата звернення: 15.02.2025).

29. Cloud Computing Security Issues and Challenges – IEEE Xplore. URL: https://ieeexplore.ieee.org/document/6148083 (дата звернення: 18.02.2025).

30. GlusterFS Storage for Cloud Environments. GlusterFS Documentation. URL: https://www.gluster.org/ (дата звернення: 20.02.2025).

31. Lustre Filesystem Overview. Lustre Documentation URL: https://lustre.org/ (дата звернення: 22.02.2025).

32. OpenStack Object Storage Best. OpenStack Documentation. URL: https://docs.openstack.org/swift/latest/admin/object-storage-best-practices.html (дата звернення: 24.02.2025).

33. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems: O'Reilly Media: 2017. 616 с.

34. Cooper B., Silberstein A., Tam E. et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment.* 2008. P. 1277–1288.

35. Tanenbaum A. S., van Steen M. Distributed Systems Principles and Paradigms. Pearson. 2017. 672 с.

36. White T. Hadoop: The Definitive Guide. O'Reilly Media. 2015. 688 с.

37. Shvachko K., Kuang H., Radia S., Chansler R. The Hadoop Distributed File System. Sunnyvale. 2010. 205 с.

38. Fox G., Hey T., Gannon D. Distributed Computing: Principles, Algorithms, and Systems: Springer: 2018. 403 с.

39. Abadi D. The Design of the Database System in the Cloud Foundations and Trends in Databases. Now Publishers. 2009. 120 с.

40. Lakshman A., Malik P. Cassandra A Decentralized Structured Storage System. 2010. 35–40 с.

41. Bondi A. Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice. Addison-Wesley. 2014. 448 с.

42. Hidvegi G. OpenStack Administration with Ansible 2 Harness the Power of Ansible 2 to Automate Deployment and Orchestration of OpenStack Services. Packt Publishing. 2018. 300 c.

43. Kolla Ansible Quick Start. Kolla Documentation. URL: https://docs.openstack.org/kolla-ansible/latest/user/quickstart.html (дата звернення: 24.02.2025).

44. Bell C. MySQL High Availability Tools for Building Robust Data Centers. O'Reilly Media. 2010. 624 c.

45. Hochstein L. Ansible Up and Running: Automating Configuration Management and Deployment the Easy Way. O'Reilly Media. 2017. 398 c.

46. Biswas P., Patwa F., Sandhu R. Content level access control for openstack swift storage. *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. 2015. P. 123-126.

47. Chekam T. T. et al. On the synchronization bottleneck of OpenStack Swift-like cloud storage systems. *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. 2016. P. 1-9.

48. Gugnani S., Lu X., Panda D. K. Swift-X: Accelerating OpenStack swift with RDMA for building an efficient HPC cloud. *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*.2017. P. 238-247.

49. Pepple K. Deploying openstack. O'Reilly Media. 2011.

50. COSSON M. G. S. C. Performance evaluation of Openstack Swift. 2024.

51. Jain P. Cloud service orchestration based architecture of OpenStack Nova and Swift. *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2016. P. 2453-2459.

52. Chen Y. et al. Mass: Workload-aware storage policy for openstack swift. *Proceedings of the 49th International Conference on Parallel Processing*. 2020. P. 1-11.

53. Karthika G. K., Malathy R. C., Keerthana M. Improving Scalability Of Storage System.

54. Noertjahyana A. Private cloud storage implementation using OpenStack Swift. Petra Christian University, 2018.

55. Saullo a., Guttadoro d. Data protection in policy evolution management of base and surface encryption layers in OpenStack swift. 2015.

56. Eiríksson Ó. Developing an OpenStack Public Cloud Storage .2016

57. Weis A. Measuring and Modeling the Performance of OpenStack. 2016.

58. Campbell T. Troubleshooting OpenStack. Packt Publishing Ltd. 2016.

59. Höppli R., Bohnert T. M., Militano L. Hera object storage. A seamless, automated multi-tiering solution on top of OpenStack swift. *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. 2018. P. 24-31.

60. Lv W. Design and implementation of an encrypted cloud storage system based on OpenStack Swift. *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS).* 2016. P. 373-377.

61. Salib E. H. Empowering Undergraduate Students With Cloud Computing Skills. A Proposal for OpenStack-Centric Education. 2*024 IEEE Frontiers in Education Conference (FIE.* 2024. P. 1-9.

62. Vainio A. Automated Software Configuration for Cloud Deployment. 2020.

63. Patil M. et al. OpenStack Cloud Deployment for Scientific Applications. *2021 International Conference on Computing, Communication and Green Engineering (CCGE)*. 2021. P. 1-7

64. Vitucci C. et al. Implementation and deployment of a server at the edge using openstack components. *Proceedings of the 19th International Conference on Networks (ICN 2020)*. IARIA. 2020.

65. Denton J. OpenStack Networking Essentials. Packt Publishing Ltd. 2016.

66. Xiang L., Shaobin L. Hybrid cloud networking design based on Openstack architecture. *Journal of Physics Conference Series*. 2020. Т. 1693. №. 1. С. 012011.

67. Solberg M., Silverman B. OpenStack for architects. Packt Publishing Ltd. 2017.

68. Fifield T. OpenStack Operations Guide: Set up and manage your openstack cloud. O'Reilly Media. 2014.

69. Haja D. How to orchestrate a distributed OpenStack. *Conference on Computer Communications Workshops*. 2018. P. 293-298.

70. Premsankar G., Ahokas K., Luukkainen S. Design and implementation of a distributed mobility management entity on OpenStack. *IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 2015. P. 487-490.

ДОДАТОК А
(обов’язковий)

МЕРЕЖЕВА АРХІТЕКТУРА СИСТЕМИ ЗБЕРІГАННЯ ДАНИХ НА ОСНОВІ OPENSTACK OBJECT STORAGE

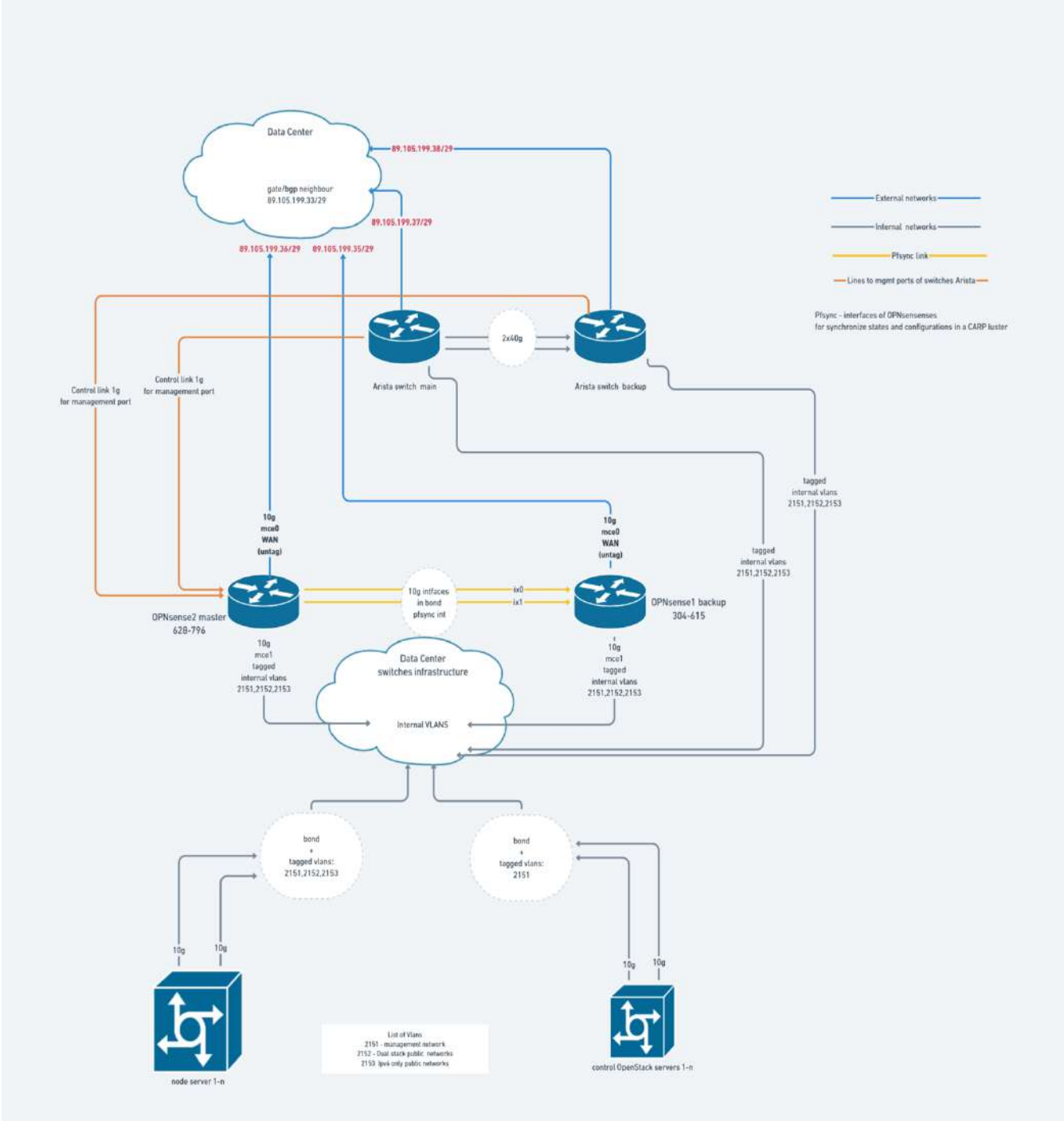


Рисунок А.1 – Мережева архітектура системи

ДОДАТОК Б
(обов'язковий)

СЕРТИФІКАТ УЧАСНИКА ВСЕУКРАЇНСЬКОЇ КОНФЕРЕНЦІЇ

Сертифікат № 2024-011-1

Міністерство освіти і науки України
Хмельницький національний університет

СЕРТИФІКАТ

Жарчинський Сергій Миколайович

учасник XVI Всеукраїнської науково-практичної конференції
«Актуальні проблеми комп'ютерних наук АПКН-2024»
24 години участі (0,8 ECTS credits)

Голова оргкомітету АПКН-2024

Олег СИНЮК
проректор Хмельницького національного
університету з наукової роботи,
доктор технічних наук, професор

м. Хмельницький
15-16 листопада 2024

E-mail: apkt.khnu@gmail.com

ДОДАТОК В

(не обов’язковий)

СТВОРЕННЯ КІЛЕЦЬ, ОБ’ЄКТІВ, ТА АККАУНТІВ.

Створення та заповнення кільця об’єктів (Object Ring)

```
docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
$KOLLA_SWIFT_BASE_IMAGE swift-ring-builder
/etc/kolla/config/swift/object.builder create 10 3 1
for node in ${STORAGE_NODES[@]}; do
  for i in {0..2}; do
    docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
    $KOLLA_SWIFT_BASE_IMAGE swift-ring-builder \
    /etc/kolla/config/swift/object.builder add r1z1-${node}:6000/d${i} 1
  done
done
```

Створення та заповнення кільця акаунтів (Account Ring)

```
docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
$KOLLA_SWIFT_BASE_IMAGE swift-ring-builder
/etc/kolla/config/swift/account.builder create 10 3 1
for node in ${STORAGE_NODES[@]}; do
  for i in {0..2}; do
    docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
    $KOLLA_SWIFT_BASE_IMAGE swift-ring-builder \
    /etc/kolla/config/swift/account.builder add r1z1-${node}:6001/d${i} 1
  done
done
```

Створення та заповнення кільця контейнерів (Container Ring)

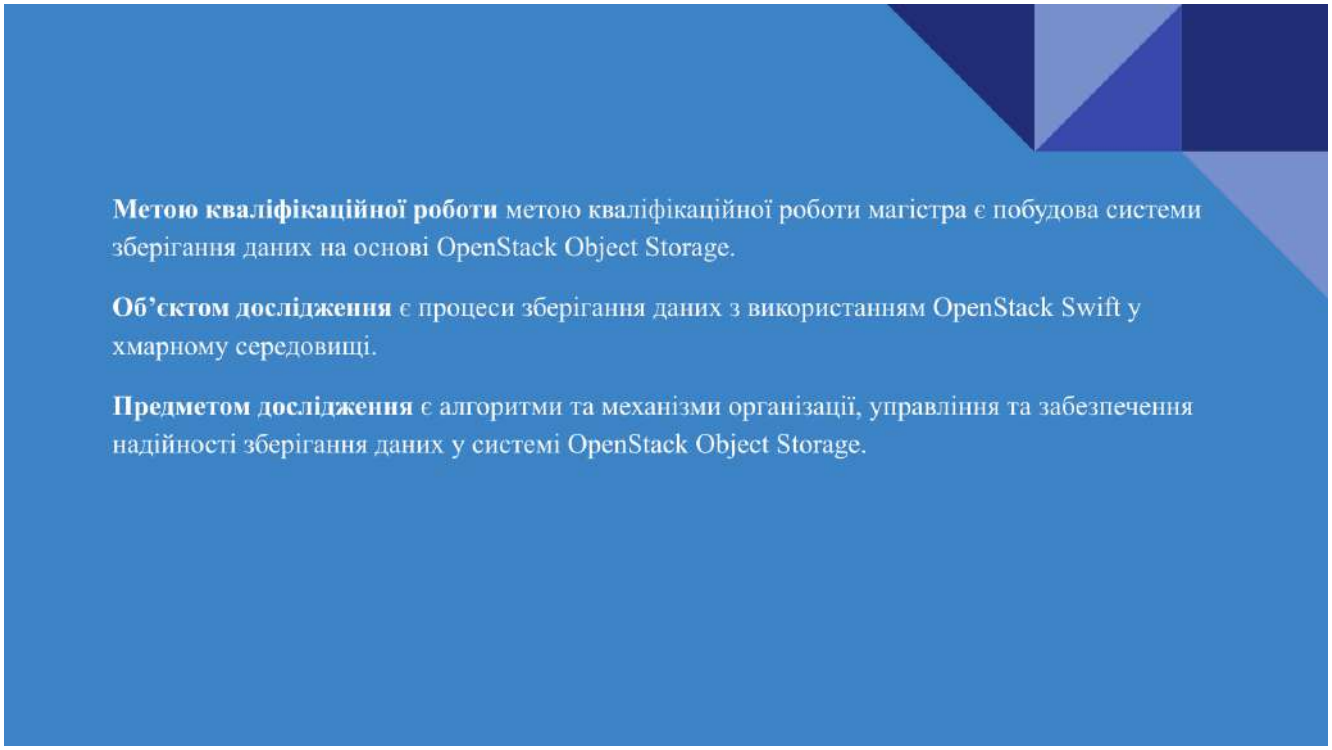
```
docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
$KOLLA_SWIFT_BASE_IMAGE swift-ring-builder
/etc/kolla/config/swift/container.builder create 10 3 1
```

```
    for node in ${STORAGE_NODES[@]}; do
      for i in {0..2}; do
        docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
        $KOLLA_SWIFT_BASE_IMAGE swift-ring-builder \
        /etc/kolla/config/swift/container.builder add r1z1-${node}:6002/d${i} 1
      done
    done
    docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
    $KOLLA_SWIFT_BASE_IMAGE swift-ring-builder
/etc/kolla/config/swift/container.builder create 10 3 1
    for node in ${STORAGE_NODES[@]}; do
      for i in {0..2}; do
        docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
        $KOLLA_SWIFT_BASE_IMAGE swift-ring-builder \
        /etc/kolla/config/swift/container.builder add r1z1-${node}:6002/d${i} 1
      done
    done
    for ring in object account container; do
      docker run --rm -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
      $KOLLA_SWIFT_BASE_IMAGE swift-ring-builder
      etc/kolla/config/swift/${ring}.builder rebalance
    done
```

ДОДАТОК Г

(обов’язковий)

ПРЕЗЕНТАЦІЯ РОБОТИ



НАУКОВА НОВИЗНА ОТРИМАНИХ РЕЗУЛЬТАТІВ

Для розв'язання поставлених задач використовувалися методи порівняльного аналізу, моделювання, архітектурного проєктування, симуляційного тестування та практичного впровадження.

Наукова новизна отриманих результатів:

- набув подальшого розвитку метод організації зберігання даних з використанням модифікованого хешування у кластерних середовищах;

ПРАКТИЧНА ЗНАЧИМІСТЬ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Практична значимість отриманих результатів полягає у тому, що розроблена система на базі OpenStack Object Storage є економічно доцільною альтернативою комерційним рішенням. Вона забезпечує високу продуктивність, масштабованість та відмовостійкість, що дозволяє застосовувати її в освітніх, наукових і комерційних дата-центрах, а також у приватних хмарах із вимогами до надійного зберігання великих обсягів неструктурованих даних.

ПУБЛІКАЦІЯ

Опубліковано публікацію у Збірнику наукових праць за матеріалами XVI Всеукраїнська науково-практична конференція "Актуальні проблеми комп'ютерних наук АПКН – 2024", (Хмельницький – 2024. –С. 205–208)

АКТУАЛЬНІСТЬ ТЕМИ ДОСЛІДЖЕННЯ

У сучасному цифровому світі обсяги неструктурованих даних стрімко зростають, що створює високі вимоги до надійності, масштабованості та ефективності систем їх зберігання.

Більшість традиційних СЗД не забезпечують належного рівня відмовостійкості або гнучкого масштабування, що критично для хмарних обчислювальних платформ.

З огляду на потреби наукових, освітніх, державних і комерційних структур, актуальним є створення відкритої, доступної та економічно вигідної системи зберігання, яка б ефективно працювала в хмарному середовищі, інтегрувалася з OpenStack і могла витримувати великі обсяги трафіку та даних.

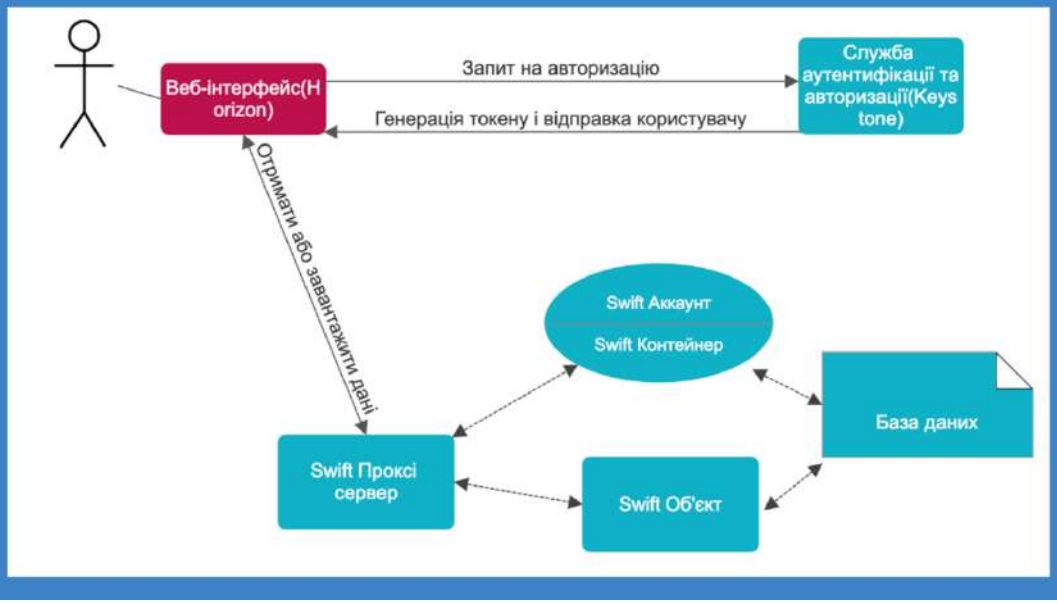
СТРУКТУРА СИСТЕМИ НАДІЙНОГО ЗБЕРІГАННЯ ДАНИХ НА ОСНОВІ OPENSTACK SWIFT

Розроблена система має модульну та розподілену архітектуру, яка забезпечує масштабованість, відмовостійкість і високу продуктивність. Її структура включає такі основні компоненти:

- **Proxy-сервер** — приймає запити від користувачів, маршрутизує їх до відповідних вузлів зберігання, відповідає за аутентифікацію та балансування навантаження.
- **Account-сервери** — керують обліковими записами, метаданими та ієрархією контейнерів.
- **Container-сервери** — відповідають за структуру контейнерів, облік об'єктів та швидкий доступ до їх списку.
- **Object-сервери** — безпосередньо зберігають об'єкти (файли), забезпечуючи реплікацію та доступність.
- **Ring-файли (об'єктне хеш-кільце)** — визначають логічну схему розподілу даних між вузлами та забезпечують узгоджений доступ.
- **Keystone** — сервіс аутентифікації й авторизації, який керує доступом до ресурсів OpenStack.
- **Horizon** — графічний веб-інтерфейс для адміністрування OpenStack і візуалізації стану кластеру.
- **Kolla-Ansible** — забезпечує автоматизацію розгортання системи, конфігурацію мережі, сервісів та моніторинг.

Уся система функціонує в ізольованому середовищі на базі OpenStack з підтримкою масштабування у горизонтальній площині, що дозволяє розширювати обсяг сховища без переривання роботи сервісів.

ВЗАЄМОДІЯ КОРИСТУВАЧА З СИСТЕМОЮ ЗБЕРІГАННЯ ДАНИХ НА БАЗІ OPENSTACK OBJECT STORAGE



МЕРЕЖЕВА ІНФРАСТРУКТУРА СИСТЕМИ

Ефективне функціонування системи зберігання даних на базі OpenStack Object Storage значною мірою залежить від правильно спроєктованої мережі. У проєкті реалізовано декілька логічно та фізично розділених мережевих сегментів:

- **Зовнішня мережа (External)**
  Забезпечує доступ клієнтів до API та Horizon. Через неї проксі-сервери приймають запити та маршрутизують їх до відповідних компонентів. Цей сегмент захищений міжмережевими екранами і підтримує TLS-шифрування, яке завершується на проксі-серверах .
- **Мережа кластерної взаємодії (Storage)**
  Використовується для обміну даними між вузлами зберігання. Реалізована через інтерфейси 10 GbE, що забезпечують високу пропускну здатність для реплікації та синхронізації.
- **Мережа керування (Management + OOB)**
  Призначена для адміністрування та моніторингу. Позасмугова мережа (out-of-band) базується на IPMI, iLO, DRAC для аварійного доступу навіть при відмові основної мережі.
- **VLAN та Bonding**
  Мережеві інтерфейси об'єднані через bonding, а логічна ізоляція досягається за допомогою VLAN, що зменшує витрати на фізичну інфраструктуру та підвищує керованість.

Система орієнтована на обробку до 100 запитів/сек обсягом 1 МБ кожен (≈0.8 Гбіт/с), тому 10 GbE обрано як базовий стандарт для зовнішнього та реплікаційного трафіку.

Практичне впровадження системи дозволило перевірити її функціональні та експлуатаційні характеристики в умовах тестового навантаження. Результати експериментів підтвердили коректну роботу механізмів реплікації, стабільний розподіл об'єктів по вузлах, а також збереження доступності даних при відмові окремих компонентів. Здійснено симуляцію навантаження на систему з великою кількістю запитів читання й запису, що продемонструвало високу пропускну здатність та ефективність кешування на рівні proxy-серверів.

Також у роботі запропоновано удосконалення базової архітектури Swift. Зокрема, розроблено концепцію Swift-X, яка включає оптимізовану маршрутизацію запитів, використання високошвидкісних інтерфейсів на базі InfiniBand, хеш-функцію xxHash замість MD5 для пришвидшення розрахунків та сервер метаданих замість традиційного proxy для зниження затримок. Ці рішення забезпечують підвищену ефективність системи та скорочення часу відповіді при доступі до даних.

Окрім технічних аспектів, виконано аналіз економічної доцільності. Вартість побудови системи на базі OpenStack значно нижча, ніж комерційні рішення провідних виробників, при збереженні функціональності, безпеки та гнучкості. Це відкриває можливості для впровадження в освітніх закладах, державних установах, малих і середніх компаніях.

Таким чином, запропонована система поєднує в собі відкритість, надійність, продуктивність та адаптивність до вимог сучасних інфраструктур. Вона може бути основою для розгортання приватних або гібридних хмар, які потребують ефективного об'єктного зберігання з можливістю масштабування та інтеграції з іншими сервісами OpenStack.

ВИСНОВКИ

У межах дипломної роботи була проведена всебічна дослідницька та проєктна робота, спрямована на розробку сучасної системи надійного зберігання даних із використанням платформи OpenStack Object Storage (Swift). Робота включала в себе теоретичний аналіз, архітектурне проєктування, практичну реалізацію та експериментальну перевірку працездатності системи в тестовому середовищі.

Першочергово було виконано огляд сучасних технологій зберігання даних, таких як Ceph, GlusterFS, Lustre, IPFS, а також порівняно їх із OpenStack Swift за ключовими технічними критеріями. На основі порівняльного аналізу визначено, що саме Swift має оптимальне співвідношення між продуктивністю, масштабованістю, простотою інтеграції та відкритістю коду, що дозволяє адаптувати його під конкретні потреби організації.

У процесі моделювання архітектури було сформовано структурну модель системи з чітким розподілом обов'язків між proxy-серверами, вузлами зберігання (account, container, object) і сервісами автентифікації (Keystone). Визначено способи взаємодії між компонентами, канали передачі даних та механізми забезпечення узгодженості інформації у кластері. Окремо розроблено мережеву інфраструктуру, що включає сегменти для трафіку користувачів, внутрішньої комунікації та реплікації.

Особлива увага була приділена засобам автоматизації. Для підвищення ефективності розгортання та підтримки обрано інструмент Kolla-Ansible, який забезпечив швидке налаштування сервісів у кластері та спростив оновлення конфігурацій. Це дало змогу створити систему, готову до масштабування та експлуатації у продуктивному середовищі.

Дякую за увагу

Протокол аналізу звіту подібності експертом

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

Автор: Сергій ЖАРЧИНСЬКИЙ

Співавтор:

Назва: Жарчинський_Система надійного зберігання даних на основі OpenStack Object Storage

Експерт:

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:12.6%

Коефіцієнт подібності 2:9.2%

Мікропробіли: 0

Заміна букв: 5

Інтервали: 0

Білі знаки: 1

Дата створення звіту: 2025-05-05 17:16:44.0

Після аналізу Звіту подібності констатую наступне:

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

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

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

Обгрунтування:

2025-05-06 Доцент Андрій Нічепорук

Дата експерт

Завідувачу кафедри КІС
доктору філософії, доценту
Ользі ПАВЛОВІЙ

Жарчинського Сергія Миколайовича

ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-23-3

ЗАЯВА

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

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

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

5 травня 2025 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Здобувач: Сергій Жарчинський

Тема: Система зберігання даних на основі OpenStack Object Storage

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи магістра:

Кількість листів креслень _=__; кількість сторінок записки _75_____

1. Короткий зміст роботи та прийнятих рішень _У роботі запропоновано систему надійного зберігання даних на основі OpenStack Object Storage

2. Висновок про відповідність роботи дипломному завданню _____ Кваліфікаційна робота магістра відповідає виданому завданню_____

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: _У першому розділі проведено огляд предметної області та визначено вимоги до системи зберігання даних у хмарному середовищі. Досліджено відомі рішення та засоби в цій сфері. У другому розділі здійснено огляд сучасних підходів до побудови розподілених систем зберігання даних та виконано порівняльний аналіз.. У третьому розділі запропоновано архітектуру вдосконаленої системи зберігання з урахуванням вимог до масштабованості, відмовостійкості та швидкодії У четвертому розділі розглянуто результати моделювання та експериментального дослідження запропонованої системи, а також оцінено її ефективність у порівнянні з базовою реалізацією._____

4. Позитивні сторони роботи: Запропонована система надійного зберігання даних на основі OpenStack Object Storage забезпечує масштабованість, відмовостійкість і ефективність розподілу даних, а також дозволяє зменшити час доступу до об'єктів і підвищити цілісність інформації в умовах динамічного хмарного середовища.

5. Негативні сторони роботи: У роботі присутні певні логічні неточності щодо опису механізмів реплікації та хешування в системах об'єктного зберігання даних, а також взаємодії між компонентами OpenStack при обробці запитів на читання та запис..

6. Оцінка графічного оформлення та пояснювальної записки роботи: –

7. Відгук про роботу в цілому: В загальному робота виконана на невисокому рівні.

8. Інші зауваження: –

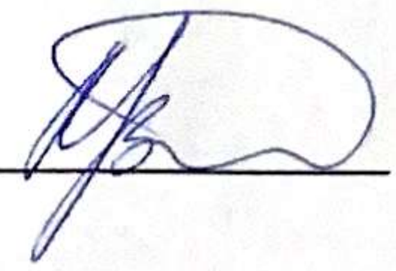
9. Оцінка кваліфікаційної роботи магістра:

Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи магістра вважаю, що робота заслуговує оцінки «добре» 4.00 (C)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи)

д.т.н., професор, Мартинюк В.В., завідувач кафедри автоматизації, комп'ютерно-інтегрованих технологій та робототехніки

"1 травня" 2025р.



РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ
КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

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

Назва: Система зберігання даних на основі OpenStack Object Storage

Автор: Жарчинський Сергій Миколайович

Спеціальність: 123 – Комп'ютерна інженерія

Освітня програма: освітньо-наукова

Науковий керівник: Василь ЯЦКІВ, д.т.н., професор

Після аналізу звіту подібності зроблено такий висновок:

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

Підтвердження:

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

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;
3) в якості запозичень в окремих місцях системою зафіксовано послідовності технічних фрагментів, які є типовими для наукових і технічних текстів і не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
4) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів з україномовними скороченнями індексів у формулах, що не є модифікацією тексту.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 12.60% і адресується до 134 першоджерела; та системою Anti-Plagiarism складає 0.0%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ Василь ЯЦКІВ

Гарант ОП _____ Олег САВЕНКО

Завідувач кафедри КІІС _____ Ольга ПАВЛОВА

Mon May 05 15:40:55 EEST 2025, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.274 Educational

The maximum coincidence with one document 0.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 13%**

	Document		Sum coincidence on the DB	
	Symbols	Lexemes	Symbols	Lexemes
ID: 240839 Title: МКР Система надійного зберігання даних на основі OpenStack Object Storage Added in a DB: 2025-05-05 Authors: Сергій ЖАРЧИНСЬКИЙ Heads: Василь ЯЦЬКІВ Consultants: Opponents:	118564	1077	899 (1%)	14 (1%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes