Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інфокомунікацій
(повна назва)
Кафедра Інфокомунікаційної інженерії імені В.В. Поповського
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Дослідження кластера серверів з використанням віртуальної IP-АТС Asterisk з метою підвищення доступності та відмовостійкості
(тема)

Виконав:
студент 2 курсу, групи ІКІм-22-1
Цилюрик В.Є.
(прізвище, ініціали)

Спеціальність: 172 Телекомунікації та радіотехніка
(код і повна назва спеціальності)
Тип програми: освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма: Інфокомунікаційна інженерія
(повна назва освітньої програми)

Керівник: доцент кафедри ІКІ ім. В.В. Поповського
Токар Л.О.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ Лемешко О.В.
(підпис) (прізвище, ініціали)

2024р.

Харківський національний університет радіоелектроніки

Факультет Інфокомунікацій
(повна назва)
Кафедра Інфокомунікаційної інженерії імені В.В. Поповського
(повна назва)
Рівень вищої освіти другий (магістерський)
Спеціальність 172 Телекомунікації та радіотехніка
(код і повна назва)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Інфокомунікаційна інженерія
(повна назва)

ЗАТВЕРДЖУЮ
Зав. кафедри _____
(підпис)
«____» _____2024р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студент Цилюрик Вадим Євгенійович
(прізвище, ім'я, по батькові)

1. Тема роботи: Дослідження кластера серверів з використанням віртуальної IP-АТС Asterisk з метою підвищення доступності та відмовостійкості
затверджена наказом по університету від «19» жовтня 2023р. №1212 Ст.
2. Термін подання студентом роботи до екзаменаційної комісії 25.01.2024р.
3. Вихідні дані до роботи: вимоги контейнеризації на основі платформи Docker для роботи додатків в будь-якому середовищі, вимоги до серверів HP з різними параметрами для оцінки продуктивності кластеру, використання платформи Kubernetes для контролю та управління кластеру на основі Asterisk, використання механізмів реплікації Raft.
4. Перелік питань, що потрібно опрацювати в роботі:
1) Аналіз питань застосування Asterisk у якості основного елементу для кластерізації
2) Визначення концепцій доступності та відмовостійкості в контексті Asterisk та кластерних систем
3) Рішення для забезпечення доступності та відмовостійкості, включаючи реплікацію даних, балансування навантаження, автоматичне переключення та резервне копіювання
4) Розробка нового рішення для кластеру Asterisk з метою підвищення доступності та відмовостійкості на основі контейнеризації Docker з автоматичним управлінням Kubernetes з використанням платформи AWS

5. Перелік графічного матеріалу із зазначенням креслень, плакатів, комп'ютерних ілюстрацій: Демонстраційний матеріал у вигляді ppt-презентації;

6. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		(підпис)	(дата)
Основна частина	доцент Токар Любов Олександрівна		21.01.24

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	01.09.2023	Виконано
2	Збір матеріалів для дослідження	01.10.2023	Виконано
3	Розробка 1 розділу	12.10.2023	Виконано
4	Розробка 2 розділу	10.11.2023	Виконано
5	Розробка 3 розділу	14.12.2023	Виконано
6	Розробка 4 розділу	23.12.2023	Виконано
7	Оформлення кваліфікаційної роботи	15.01.2024	Виконано

Дата видачі завдання 01 вересня 2023 року

Студент _____ Цилюрик В.Є.
(підпис) (прізвище, ініціали)

Керівник роботи _____ доцент Токар Л.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 78 с., 36 рис., 1 табл., 29 джерел, 3 додатки.

АТС, ASTERISK, DOCKER, KUBERNETES, RAFT, КЛАСТЕРІЗАЦІЯ

Об’єкт дослідження – процеси дослідження та розробки кластеризованих серверів.

Предмет дослідження – методи і способи підвищення доступності та відмовостійкості системи кластерізації серверів на базі віртуальної IP-АТС Asterisk.

Мета роботи – дослідження та розробка нових підходів для підвищення доступності та відмовостійкості кластеризованих серверів на базі віртуальної IP-АТС Asterisk.

Методи досліджень – аналіз, моделювання, практична реалізація, тестування.

Проведено аналіз моделі M/M/c, яка є фундаментальною в теорії масового обслуговування, в якій час між приходами заявок та час обслуговування підпорядковуються експоненційному розподілу. Ця модель дозволяє оцінити ефективність системи кластерізації та реплікації Raft з кількома каналами обслуговування.

В роботі проведено ряд досліджень, спрямованих на оцінку використання технологій Asterisk у складних розподілених середовищах, підвищення якості послуг, та зниження ризиків зупинки роботи системи у разі збоїв або атак. Основні аспекти, на які зосереджено увагу, включають методи контейнеризації на базі Docker, управління через Kubernetes та використання механізмів реплікації Raft для забезпечення консистентності даних у кластері.

ABSTRACT

Explanatory note: 78 p., 36 fig., 1 tabie, 25 sources, 3 extras.

ATS, ASTERISK, DOCKER, KUBERNETES, RAFT, CLUSTERIZATION

The object of research is the research and development processes of clustered servers.

The subject of the research is methods and methods of increasing the availability and fault tolerance of the server clustering system based on the Asterisk virtual IP PBX.

The purpose of the work is research and development of new approaches to increase the availability and fault tolerance of clustered servers based on Asterisk virtual IP PBX.

Research methods – analysis, modeling, practical implementation, testing.

The analysis of the M/M/c model, which is fundamental in the theory of mass service, is carried out, in which the time between the arrivals of applications and the service time are subject to an exponential distribution. This model allows you to evaluate the efficiency of the Raft clustering and replication system with multiple service channels.

The work includes a number of studies aimed at evaluating the use of Asterisk technologies in complex distributed environments, improving the quality of services, and reducing the risk of system downtime in the event of failures or attacks. Key areas of focus include containerization techniques based on Docker, management through Kubernetes, and the use of Raft replication mechanisms to ensure data consistency across a cluster.

ЗМІСТ

Перелік скорочень, умовних позначень, символів, одиниць і термінів…..... 7
Вступ…….……..... 9
1 Аналіз питань застосування ASTERISK у якості основного елементу для кластерізації ……..... 11
1.1 Розвиток і концепція Asterisk ……..……..... 11
1.2 Архітектурні особливості Asterisk…….…... 12
1.3 Поточні підходи до кластерізації Asterisk …..... 13
1.4 Потенціал та перспективи Asterisk у кластерних системах 15
1.5 Основні компоненти IP телефонії…….. 16
1.6 Затримки мережі в IP-телефонії……. 18
2 Визначення концепцій доступності та відмовостійкості..... 24
2.1 Основні поняття та визначення доступності та відмовостійкості ……..... 24
2.2 Методи та технології забезпечення високої доступності та відмовостійкості в кластерних системах……..... 26
3 Рішення для забезпечення доступності та відмовостійкості кластера.... 36
3.1 Аналіз методів реплікації даних……. 36
3.2 Алгоритм реплікації даних Raft..... 42
3.3 Опис математичної моделі……. 52
4 Розробка рішення для кластеру ASTERISK на основі контейнеризації Docker з автоматичним управлінням Kubernetes та з використанням платформи AWS..... 57
4.1 Архітектура та компоненти запропонованого рішення …..... 57
4.2 Інтеграція Docker для контейнеризації Asterisk ….……. 60
4.3 Управління кластером Asterisk через Kubernetes..... 62
4.4 Дослідження продуктивності Kubernetes……... 63
4.5 Використання AWS для розгортання та масштабування рішення 69
Висновки..... 74
Перелік джерел посилання..... 76
ДОДАТОК А Код програми для аналізу математичної моделі……. 79
ДОДАТОК Б Dockerfile Aterisk…… 81
ДОДАТОК В Створення Kubernetes Deployment…… 83

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

АТС – автоматична телефонна станція
МСЕ – Міжнародний союз електрозв'язку
ОС – операційна система
AWS – Amazon Web Services
AMI – Asterisk Manager Interface
AGI – Asterisk Gateway Interface
ARI – Asterisk RESTful Interface
AMI – Advanced Metering Infrastructure
API – Application Programming Interface
DDoS – Distributed Denial-of-Service
DNS – Domain Name System
DRBD – Distributed Replicated Block Device
DTMC – Discrete-Time Markov Chain
CLI – Command Line Interface
CPU – Central Processing Unit
CSS – Cascading Style Sheets
EC2 – Elastic Compute Cloud
EKS – Amazon Elastic Kubernetes Service
ECS – Amazon Elastic Container Service
ECR – Amazon Elastic Container Registry
ELB – Elastic Load Balancing
IAM – Identity and Access Management
IP – Internet Protocol
IAX2 – Inter Asterisk eXchange version 2
IBFT – Istanbul Byzantine Fault Tolerant
ITU-T – International Telecommunication Union Telecommunication
IVR – Interactive Voice Response
K8S – Kubernetes
LAN – Local Area Network
MFA – Multi-Factor Authentication
NAT – Network Address Translation

QOS – Quality of Service
OSI – Open Systems Interconnection
POTS – Plain Old Telephone Service
RAID – Redundant Array of Independent Disks
RAM – Random Access Memory
RBAC – Role Based Access Control
RPC – Remote Procedure Call
RSA – Rivest Shamir Adleman
RTP – Real-time Transport Protocol
SSH – Secure Shell
SLA – Service Level Agreement
SIP – Session Initiation Protocol
VoIP –Voice over IP
VPC – Virtual Private Cloud
WebRTC – Web Real-Time Communications
UDP – User Datagram Protocol

ВСТУП

Сучасний світ характеризується стрімким розвитком технологій та зростанням важливості інформаційних систем у всіх сферах життя. Центральне місце в цій системі займає комунікація, а точніше, її надійність та доступність. В контексті цього дослідження, велику увагу приділено системі кластерізації серверів на базі віртуальної IP-АТС Asterisk, яка є вирішальною для забезпечення стійкості комунікаційних сервісів.

Об'єктом дослідження у роботі виступає система кластерізації серверів на базі віртуальної IP-АТС Asterisk, її компоненти та механізми, які забезпечують доступність та відмовостійкість в різних умовах функціонування. Зокрема, увага приділяється методам контейнеризації на основі платформи Docker, механізмам управління та контролю за допомогою платформи Kubernetes, а також використанню механізмів реплікації Raft для забезпечення консистентності даних у кластері.

Ці аспекти вивчаються для того, щоб забезпечити максимально ефективне та надійне використання технологій Asterisk у складних розподілених середовищах, а також для підвищення якості послуг та зниження загроз зупинки роботи системи у випадках збоїв або атак.

Актуальність даного дослідження полягає у необхідності забезпечення неперервної роботи інформаційних систем, особливо в умовах зростаючих обсягів даних та вимог до їх доступності. Вивчення та вдосконалення систем на базі Asterisk дозволяє значно підвищити ефективність комунікаційних процесів, знижуючи при цьому ризики втрати даних та впливу зовнішніх загроз.

Розвиток та інтеграція сучасних технологій, таких як Docker та Kubernetes, у систему кластерізації Asterisk відкриває нові можливості для підвищення надійності та доступності серверів. Це стає особливо важливим для організацій, які прагнуть забезпечити безперебійну роботу своїх сервісів в умовах високої конкуренції та постійно змінюваного цифрового середовища.

Доступність серверів можна розглядати як здатність системи або сервісу бути у доступі для користувачів у визначений часовий період без втрати якості обслуговування. Відмовостійкість, у свою чергу, визначається як здатність системи відновлюватися після збоїв та продовжувати нормальну роботу без

значущих затримок. Обидва цих параметра є важливими для забезпечення неперервної роботи організації та задоволення потреб її клієнтів.

Віртуальна IP-АТС Asterisk вже довгий час є популярним рішенням для організацій, які хочуть мати гнучкість та контроль над своїми комунікаційними сервісами. Проте, як і будь-яке програмне забезпечення, Asterisk також має свої слабкі сторони, що стосуються доступності та відмовостійкості. У цьому контексті дослідження нових підходів до кластерізації Asterisk, особливо з використанням сучасних технологій, таких як Docker та Kubernetes, може принести значущі переваги для організацій різних розмірів.

Метою роботи є дослідження можливостей підвищення доступності та відмовостійкості кластера серверів з використанням віртуальної IP-АТС Asterisk на основі сучасних технологій контейнеризації та оркестрації.

1 АНАЛІЗ ПИТАНЬ ЗАСТОСУВАННЯ ASTERISK У ЯКОСТІ ОСНОВНОГО ЕЛЕМЕНТУ ДЛЯ КЛАСТЕРІЗАЦІЇ

1.1 Розвиток і концепція Asterisk

Asterisk є однією з найвідоміших відкритих систем для створення IP-АТС (IP-телефонії). Ця система була розроблена Марком Спенсером у 1999 році і з того часу пройшла довгий шлях розвитку.

На початковому етапі свого існування Asterisk був невеликим проєктом, створеним для розв'язання конкретних потреб в IP-телефонії. Проте завдяки своїм відкритим кодом і гнучкості, система швидко набирала популярність серед розробників та мережевих адміністраторів.

З роками Asterisk почав розширювати свою функціональність. Від простої системи IP-телефонії він перетворився на потужний інструмент для створення різноманітних телекомунікаційних рішень, включаючи call-центри, конференц-зв'язок, голосову пошту. Архітектурно Asterisk базується на модульному принципі, що дозволяє легко додавати нові функції та інтегрувати систему з іншими продуктами [1].

Завдяки активному співтовариству та регулярним випускам оновлень, Asterisk постійно адаптується до змінюваних умов ринку та технологічних трендів. Форуми, конференції, присвячені Asterisk, з'явилися по всьому світу, де спеціалісти обмінювалися досвідом, ідеями та кращими практиками. Саме завдяки цьому динамічному співтовариству, Asterisk швидко адаптувався до нових викликів ринку, відповідаючи на потреби своїх користувачів. Наприклад, були впроваджені підтримка протоколу Web Real-Time Communications (WebRTC), захист від Distributed Denial-of-Service-атак (DDoS), а також багато інших нововведень.

На сьогоднішній день Asterisk займає одне з провідних місць у світі IP-телефонії. Його використовують як малі компанії для внутрішньої комунікації, так і великі корпорації для створення широкомасштабних телекомунікаційних інфраструктур.

Asterisk, починаючи з своїх скромних коренів як проєкт для внутрішньої потреби, розвинувся в одне з найпопулярніших та найбільш потужних рішень у світі IP-телефонії. Його гнучкість, модульність та активність розробників роблять

його ідеальним вибором для різних телекомунікаційних потреб.

1.2 Архітектурні особливості Asterisk

Asterisk, будучи лідером у сфері відкритої IP-телефонії, має багатий набір архітектурних особливостей, які роблять його унікальним і гнучким рішенням для різних завдань. Модульну архітектуру Asterisk показано на рис. 1.1 [2].

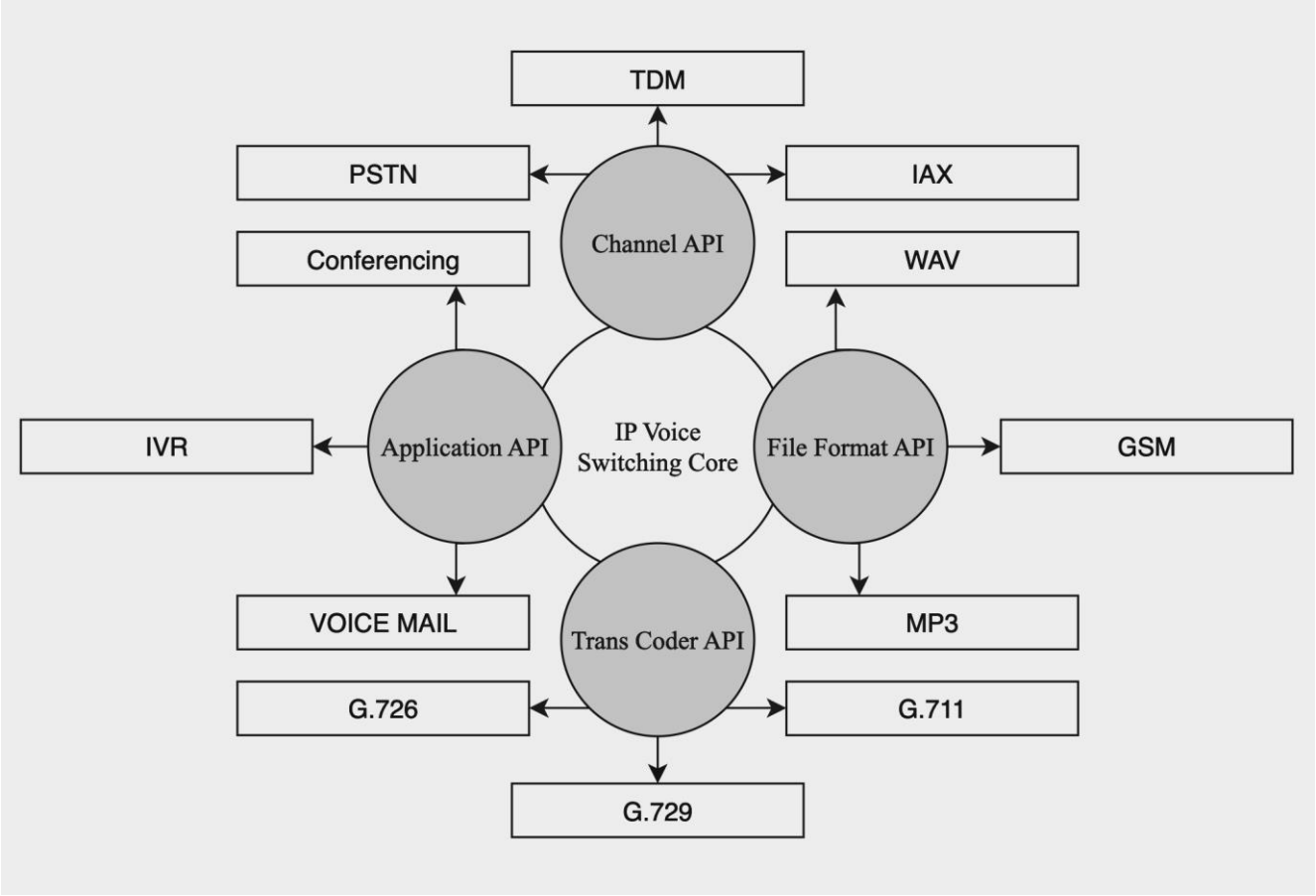

Рисунок 1.1 – Модульна архітектура Asterisk

Ядро Asterisk (Asterisk Core): це центральний компонент, який забезпечує управління дзвінками, обробку подій та контроль за роботою інших модулів.

Dialplan: це механізм маршрутизації, який визначає, як Asterisk повинен обробляти вхідні та вихідні дзвінки.

Модулі каналів (Channel Modules): ці модулі дозволяють Asterisk підтримувати різні протоколи телефонії, такі як Session Initiation Protocol (SIP), Inter Asterisk eXchange version 2 (IAX2), Plain Old Telephone Service (POTS) тощо.

Прикладні модулі (Application Modules): ці модулі додають різноманітну

функціональність до системи, включаючи голосову пошту, конференції, Interactive Voice Response (IVR) тощо.

Одна з ключових особливостей Asterisk - це його модульна структура. Користувачі можуть вибрати та завантажити лише ті модулі, які їм потрібні, що дозволяє оптимізувати ресурси системи. Така модульність також сприяє легкості розширення функціоналу: нові можливості можна додавати шляхом розробки та додавання нових модулів.

Asterisk пропонує декілька інтерфейсів для інтеграції з зовнішніми системами [3]:

– Asterisk Manager Interface (AMI): інтерфейс для управління та моніторингу Asterisk. Він дозволяє стороннім додаткам взаємодіяти з ядром Asterisk у реальному часі;

– Asterisk Gateway Interface AGI (AGI): цей інтерфейс дозволяє розширити функціональність Asterisk за допомогою зовнішніх скриптів та програм;

– Asterisk Restful Interface (ARI): сучасний інтерфейс, що дозволяє розробникам створювати власні додатки, які взаємодіють з Asterisk через RESTful Application Programming Interface (API).

Усі ці інтерфейси роблять Asterisk надзвичайно гнучким рішенням, що може бути інтегровано в різноманітні IT-інфраструктури та додатки.

1.3 Поточні підходи до кластерізації Asterisk

Кластерізація в інформаційних системах виступає як ключова технологія, що забезпечує масштабованість, доступність та відмовостійкість. У контексті IP-АТС на базі Asterisk це особливо актуально, адже сучасний бізнес потребує високої надійності телефонної комунікації.

Однією з найпростіших методик кластерізації для Asterisk є Domain Name System (DNS) балансування навантаження (рис. 1.2) [4].

Ця методика передбачає використання DNS для перенаправлення вхідних дзвінків на різні сервери Asterisk, дозволяючи рівномірно розподілити навантаження та покращити загальну продуктивність системи. DNS-балансування навантаження є одним із підходів до розподілу вхідного трафіку між декількома серверами або вузлами.

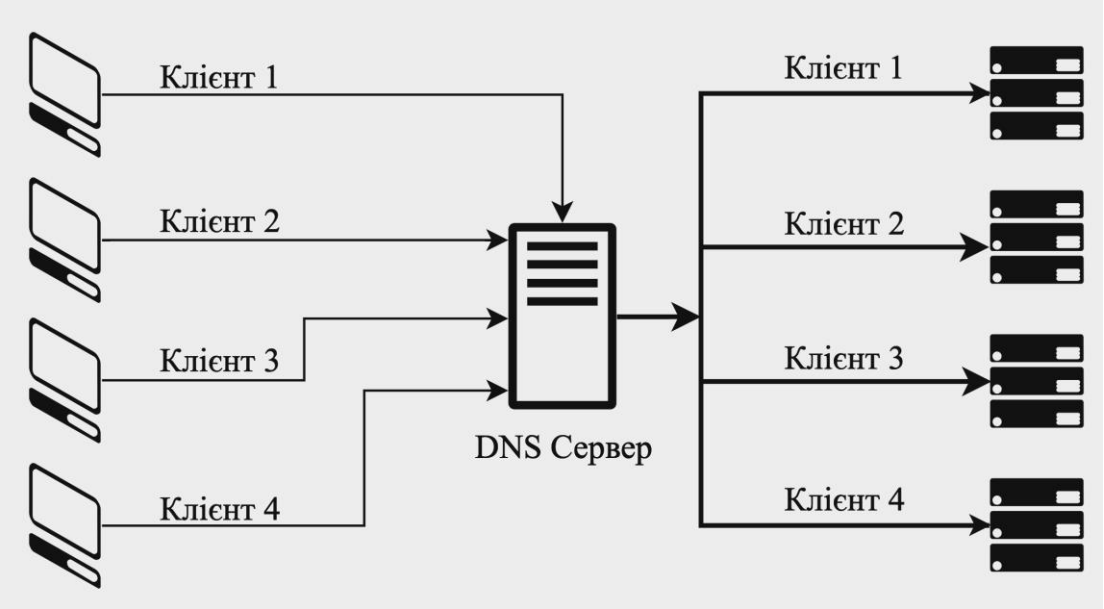

Рисунок 1.2 – Схема DNS-балансування навантаження між клієнтами та серверами

Застосування DNS для балансування навантаження базується на принципі відповіді на DNS-запити різними IP-адресами, що вказують на різні сервери. Такий підхід може бути дуже ефективним для балансування навантаження в мережах з великим обсягом трафіку.

Однак для реального забезпечення відмовостійкості та високої доступності часто використовуються більш складні методи. Наприклад, shared-nothing архітектура, де кожен вузол у кластері є повністю автономним і не ділить жодних ресурсів з іншими вузлами. Це підвищує надійність системи, адже невдача одного вузла не вплине на роботу інших.

В такій архітектурі кожен вузол системи має власні локальні ресурси, такі як обчислювальна потужність, зберігання даних та пам'ять, і не ділить їх з іншими вузлами. Це контрастує з архітектурами, де ресурси діляться між вузлами.

Основна ідея shared-nothing полягає в тому, щоб забезпечити високу доступність, масштабованість та відмовостійкість, уникаючи при цьому «вузьких» місць і точок єдиного збою.

У такій архітектурі, якщо один вузол відмовляє, це не впливає на роботу інших вузлів. Крім того, систему можна легко масштабувати, додаючи нові вузли без необхідності змін в існуючих.

Shared-nothing архітектура часто використовується в базах даних, обчислювальних кластерах та інших розподілених системах, де потрібна висока доступність та відмовостійкість. Вона дозволяє декільком серверам паралельно

обробляти запити або виконувати обчислення, кожен з яких працює з власним набором даних, не перешкоджаючи роботі інших.

Однією з основних переваг shared-nothing є те, що вона може пропонувати лінійну масштабованість: додавання нового вузла збільшує загальну продуктивність системи приблизно на величину цього вузла [5].

Також важливо зазначити, що, хоча shared-nothing архітектура має багато переваг, вона також може створити додаткові виклики, особливо у ситуаціях, коли потрібна консистентність даних між вузлами. Тому важливо враховувати ці аспекти при виборі архітектури для конкретного застосування.

Реплікація бази даних також є важливою частиною кластерізації в Asterisk. Використовуючи технології, такі як Distributed Replicated Block Device (DRBD) або Corosync, можна синхронізувати дані між різними вузлами, що забезпечує узгодженість інформації та гарантує, що у випадку відмови одного сервера, інший кластер зможе взяти на себе його функції без втрати даних.

На рівні протоколу також існують специфічні методи кластерізації. Серед них варто виділити SIP-кластерізацію, яка забезпечує балансування SIP-дзвінків між різними вузлами. Існують спеціалізовані рішення, такі як Kamailio або OpenSIPS, які оптимізовані для цього завдання.

Для забезпечення високодоступної кластерізації можна використовувати комбінації програмного забезпечення, такого як Pacemaker і Corosync. Ці інструменти забезпечують автоматичний перехід дзвінків на резервний вузол у разі відмови основного сервера.

Зрештою, підхід до кластерізації в будь-якій системі, включаючи Asterisk, має базуватися на конкретних потребах та ресурсах організації. Важливо аналізувати та тестувати різні підходи, щоб знайти найефективніший для конкретного випадку.

1.4 Потенціал та перспективи Asterisk у кластерних системах

Потенціал та перспективи Asterisk у кластерних системах дедалі більше привертають увагу спеціалістів і організацій, які прагнуть до оптимізації своїх комунікаційних рішень.

Asterisk, як IP-АТС, вже відомий своєю високою гнучкістю, модульністю та можливістю інтеграції з іншими системами. Це дозволяє йому легко адаптуватися до різних бізнес-сценаріїв і вимог. У контексті кластерних систем такі

характеристики Asterisk роблять його ідеальним кандидатом для створення відмовостійких, масштабованих та високодоступних рішень.

Однією з ключових переваг Asterisk є його можливість працювати в різних архітектурних рішеннях, включаючи розподілені та кластерні конфігурації. Це забезпечує гнучкість при розгортанні та масштабуванні, дозволяючи організаціям легко розширювати свої комунікаційні рішення відповідно до їх потреб [6].

По мірі зростання обсягів даних та кількості користувачів, вимоги до стабільності та доступності комунікаційних систем також зростають. Тут Asterisk може виступити як відповідь на ці виклики, надаючи можливість створення кластерів для балансування навантаження, автоматичного відновлення після збоїв і забезпечення неперервної роботи.

Використовуючи сучасні технології, такі як контейнеризація та оркестрація з допомогою Kubernetes, Asterisk може надати ефективні рішення для великих розподілених мереж та систем.

Таким чином, потенціал та перспективи Asterisk в кластерних системах відкривають нові горизонти для організацій, які прагнуть до модернізації своїх комунікаційних інфраструктур. Зокрема, у великих корпоративних мережах та середніх підприємствах, де стабільність та надійність є критично важливими.

Asterisk також може стати ключовим елементом в системах, що потребують географічно розподіленої інфраструктури, дозволяючи ефективно управляти даними та комунікаційними потоками між різними регіонами або навіть континентами. Це особливо актуально для глобальних компаній, які мають офіси та дата-центри в різних частинах світу.

З урахуванням усіх цих факторів можна стверджувати, що Asterisk має великий потенціал у ролі основного компонента для кластерних систем в майбутньому. Його гнучкість, відкритість та активна розробка гарантують, що Asterisk залишиться на передовій комунікаційних технологій протягом багатьох років.

1.5 Основні компоненти IP телефонії

Для нормальної роботи телефонії необхідно, щоб вона відповідала певним вимогам та стандартам. SIP - протокол сигналізації, призначений для організації, зміни та завершення сесій зв'язку. SIP незалежний від транспортних технологій, однак при встановленні з'єднання переважно використовувати User Datagram

Protocol (UDP). Для передачі самої голосової та відеоінформації рекомендується застосовувати Real-time Transport Protocol (RTP), але можливість використання інших протоколів не виключена.

Стандарт H.323 - рекомендація International Telecommunication Union Telecommunication (ITU-T), що визначає набір стандартів для передачі мультимедійних даних по мережах з пакетною передачею. Чотирма основними компонентами Voice over IP (VoIP) з'єднання є:

– термінал;

– VoIP шлюз;

– контролер зони;

– VoIP телефон.

У SIP визначені два типи сигнальних повідомлень - запит та відповідь.

Також існує шість процедур:

– INVITE (запрошення) – запрошує користувача взяти участь у сесії зв'язку (служить для встановлення нового з'єднання; може містити параметри для узгодження);

– BYE (роз'єднання) – завершує з'єднання між двома користувачами;

– OPTIONS (опції) – використовується для передачі інформації про підтримувані характеристики (ця передача може здійснюватися безпосередньо між двома агентами користувачів або через сервер SIP);

– ACK (підтвердження) – використовується для підтвердження отримання повідомлення або для позитивної відповіді на команду INVITE;

– CANCEL (скасування) – припиняє пошук користувача;

– REGISTER (реєстрація) – передає інформацію про місцезнаходження користувача на сервер SIP, який може транслювати її на сервер адрес (Location Server).

Схематичне зображення SIP-сесії в мережі зображено на рис. 1.3.

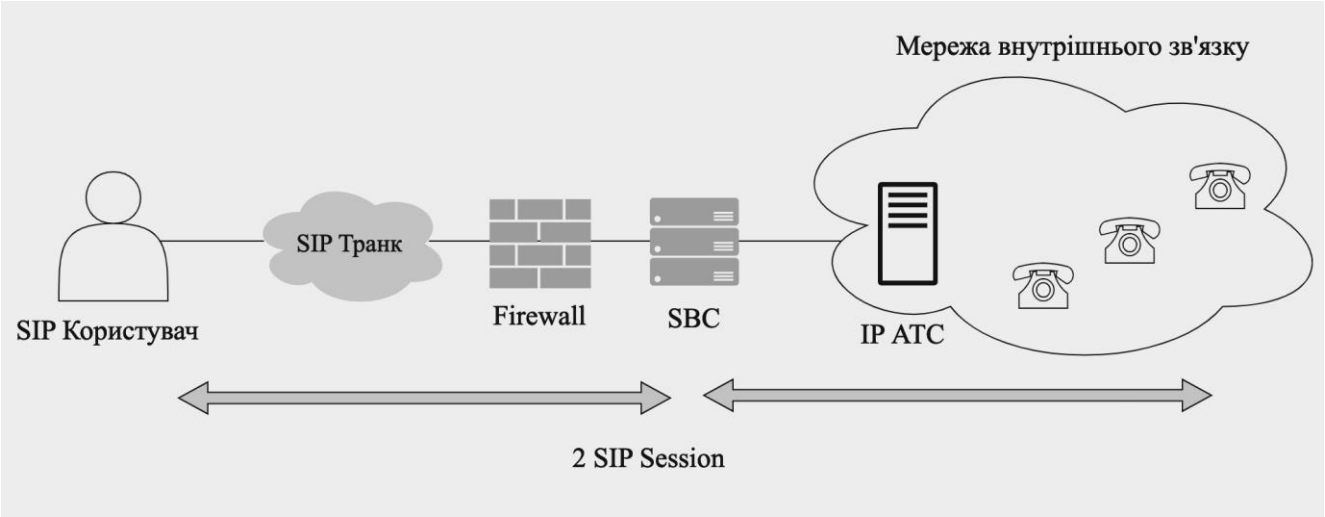

Рисунок 1.3 – Схематичне зображення SIP-сесії в мережі

Процес передачі пакетів даних відбувається за допомогою кодеків. Аудіокодеком називають програму або алгоритм, який стискає, або розтискає цифрові звукові дані, дозволяючи знизити вимоги до пропускної спроможності каналу передачі даних. В IP-телефонії на сьогоднішній день найбільш поширене перетворення за допомогою кодека G.729, а також стиснення G.711 за A-законом (alaw) та μ-законом (ulaw).

1.6 Затримки мережі в IP-телефонії

Сучасний характер інформаційних потоків у мережах з IP-маршрутизацією переживає значні зміни. IP-мережі використовуються не тільки для пересилання даних, але й для прослуховування музики, перегляду відео, обміну голосовими повідомленнями, організації мультимедійних конференцій, швидкісного керування/контролю, он-лайн ігор та інших додатків у реальному часі. Однак протокол IP спочатку не призначався для передачі даних у реальному часі. Загальна затримка передачі голосу ділиться на дві частини: затримка при кодуванні та декодуванні у шлюзах або кінцевому обладнанні користувачів та затримка, що внесена самою мережею.

Для зменшення загальної затримки можна вжити два підходи: оптимізувати інфраструктуру мережі для мінімізації затримок та зменшити час обробки голосових даних у шлюзі. Щоб скоротити затримку у мережі, важливо зменшити кількість транзитних маршрутизаторів і забезпечити їх високошвидкісними з'єднаннями. Для вирівнювання варіацій затримки можна використовувати

ефективні методи, такі як резервування мережних ресурсів.

Щодо IP-телефонії, її часто помилково вважають лише частиною сервісів, що надаються інтернет-провайдерами. Насправді, технології передачі голосових даних широко застосовуються в приватних та корпоративних IP-мережах і навіть у спеціалізованих окремих мережах IP-телефонії.

Корпоративні мережі часто характеризуються меншими розмірами та простою керованою топологією, з передбачуваним характером трафіку. Такі проблеми, як перевантаження локальної мережі, яка не може забезпечити прийнятну якість обслуговування, можна вирішити шляхом ізоляції серверів і клієнтів, що обробляють специфічний трафік, та сегментацією мережі. Це можна зробити за допомогою налаштування комутаторів Ethernet або шляхом додавання портів до маршрутизатора.

Виділені мережі IP-телефонії, які часто використовуються для міжміського та міжнародного зв'язку, найкраще будувати за ієрархічним принципом. На вході в мережу важливо забезпечити підключення мовних шлюзів, а в середині мережі - високошвидкісну передачу пакетів. Така структура спрощує розширення мережі та впровадження нових послуг. Завдання проєктування такої мережі не є складним, оскільки характер трафіку визначений, а пропускна спроможність легко розраховується завдяки однорідності трафіку.

Затримка є параметром, властивим будь-якій мережі передачі даних з комутацією пакетів. Для передачі мовних даних у реальному часі в мережах з комутацією пакетів значною мірою залежить від розміру затримки. Тут під затримкою розуміється сукупний час, протягом якого пакет перетинає мережу IP-телефонії від відправника до одержувача. Затримка викликає незручності під час діалогу, призводячи до перекриття розмов.

Можна виділити основні джерела сквозної затримки передачі мови:

1) Затримка накопичення, також відома як алгоритмічна. Ця затримка викликана перетворенням наборів мовних відліків мови в стиснутий кадр, які виконуються у вокодері. Цей мовний кадр не генерується, доки вокодер не зібрав усі мовні відліки у буфері. Розмір затримки залежить від типу вокодера та варіюється від мікросекунд до декількох десятків мілісекунд.

2) Затримка обробки, також відома як обчислювальна, спричинена процесом кодування та інкапсуляції кадрів у IP-пакети. Затримка обробки залежить від часу роботи процесора, обчислювальної потужності процесора та типу алгоритму обробки. Для зменшення навантаження на пакетну мережу зазвичай кілька кадрів

вокодера об'єднуються в один IP-пакет.

3) Мережна затримка залежить від передавального фізичного середовища, протоколів каналу, які використовуються для передачі мовних даних. Крім того, на мережну затримку впливають буфери, які використовуються для усунення джитера затримки пакетів на приймальному кінці. Таким чином, розмір мережної затримки залежить від ємності мережі та процесів передачі пакетів у мережі.

Згідно з рекомендацією G.114 Міжнародного союзу електрозв'язку (МСЕ) за значенням односторонньої затримки від абонента А до абонента Б можна виділити чотири рівні якості зв'язку:

– перший рівень до 150 мс – відмінна якість зв'язку;

– другий рівень до 200 мс – добра якість зв'язку;

– третій рівень до 400 мс – прийнятна якість зв'язку;

– четвертий рівень понад 450 мс – погана якість зв'язку.

Затримка в IP-телефонії, що організована через глобальну мережу Інтернет, зазвичай знаходиться на другому рівні, оскільки інтернет-провайдер не може гарантувати відмінну якість за межами своєї мережі, а затримки в такій мережі мінливі. Натомість провайдери, спеціалізовані на IP-телефонії, працюють через виділені канали. Час затримки у таких провайдерів знаходиться на першому та другому рівні. На третьому рівні працює IP-телефонія, що організована через супутниковий зв'язок. Середня сумарна затримка при використанні IP-телефонії має знаходитися в межах 150–250 мс. Вимоги до затримки для VoIP наведено на рис. 1.4.

Затримка означає загальну затримку між моментом, коли відправник говорить, і моментом, коли аудіоінформація подається одержувачу. В розмові одержувач може чути слова відправника декілька секунд після того, як вони були вимовлені.

Затримка може виникати через неефективність мережі VoIP або через фізичну відстань між користувачами. Зазвичай затримку розглядають як негативний фактор, що впливає на якість розмови. Вона може переривати природний ритм, з яким люди зазвичай спілкуються.

Рисунок 1.4 – Вимоги до затримки для VoIP

Однак у деяких випадках затримка може бути корисним інструментом у боротьбі з джитером і втратою пакетів. Якщо ці проблеми починають сильно впливати на якість дзвінка, то навмисна затримка в системі може допомогти збільшити час, який мережа має на подачу корисного аудіо одержувачу. Тестування джитера в IP Service-level Agreement (SLA) за допомогою UDP наведено на рис. 1.5.

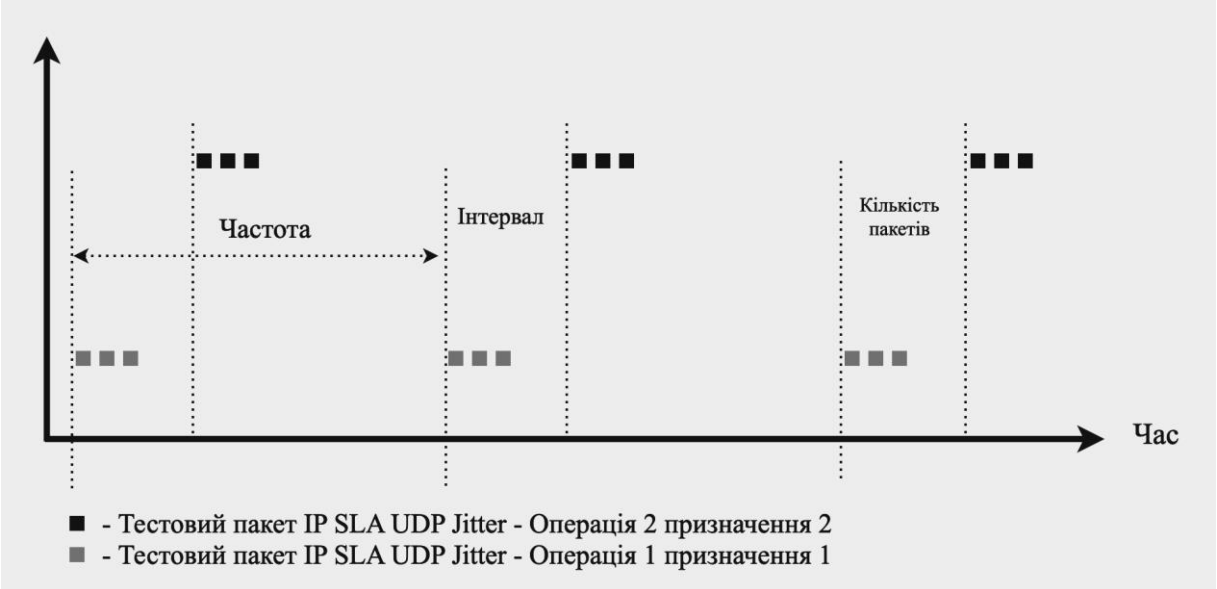

Рисунок 1.5 – Тестування джитера в IP SLA за допомогою UDP

Коли розрив між користувачами стає занадто великим, важко підтримувати значущу розмову, яка включає всі нюанси, яких ми очікуємо при висловлюванні

ідей. Мережі, які використовують затримку для зменшення впливу джитера та втрати пакетів, повинні знайти баланс між можливими перевагами та негативним ефектом довгих пауз у розмові.

Розмова відчуває дивні паузи, коли обидві сторони діалогу намагаються знайти спосіб дати кожному учаснику час для висловлювань. Такий же розрив у розмові може статися, коли затримка занадто велика в дзвінках VoIP. Це може створювати плутанину в важливих ділових зустрічах та між колегами, які використовують інтернет-телефонію для підтримки віддаленої робочої сили.

Вимоги VoIP до затримки та джиттера, визначені компанією Cisco, показують, що буфер джиттера від 20 до 50 мс може підтримувати здорову мережу VoIP у поєднанні з очікуваними затримками в черзі та розповсюдженні мережі. Загалом, якщо затримка між абонентами зберігається менше 150 мс, дзвінки можуть працювати так, як очікується користувачами.

Таким чином, сумарна затримка в IP-телефонії складається із затримок на цифрування, генерації голосового пакета, а також затримок при передачі по каналах, обробці та комутації пакета в проміжних вузлах, локальній комутації на вузлі прийому, декомпресії та перетворенні у аналоговий вигляд.

Коли голосові дані або інші відомості розбиваються на пакети для транспортування через IP-мережу, ці пакети часто доходять до кінцевого пункту призначення в різному порядку та в різний час. Така нерівномірність часу доставки викликає певні аномалії у передачі голосу, які сприймаються у вигляді шумів та клацань. Джитер можна класифікувати на три види:

– джитер, залежний від даних: виникає через обмеження пропускної здатності або помилки в елементах мережі;

– джитер, пов'язаний із затримкою розповсюдження: виникає через затримки між послідовними передачами даних в мережі.

– випадковий джитер: це результат теплового шуму в системі.

Параметри затримки та їх імовірності є ключовими для розробки ефективної стратегії обробки даних та вибору відповідних налаштувань. Зрозуміло, що часова структура потоку голосових пакетів зазнає змін, тому необхідно використовувати буфер для виправлення нерегулярностей пакетного трафіку, який виникає через затримки в мережі та потенційні зміни порядку пакетів у неперервному мовному потоці в режимі реального часу. Налаштування буфера визначаються компромісом між затримкою голосового сигналу в двосторонньому режимі та рівнем втрати пакетів. Втрата пакетів є іншою важливою проблемою в IP-

телефонії.

Причини джитера можуть бути наступними:

– вплив мережі: час проходження пакетів через мережу є непостійним і складно прогнозованим. При низькому навантаженні на мережу, маршрутизатори та комутатори можуть швидко обробляти пакети, але при високому навантаженні пакети можуть зазнавати затримок. Чим більше пристроїв проходить пакет, тим вищі затримки і, відповідно, джитер;

– вплив операційної системи: більшість програм IP-телефонії - це стандартні додатки, що працюють у різних операційних системах (ОС), таких як Windows чи Linux. Вони взаємодіють з периферійними пристроями через програмний інтерфейс. Операційні системи часто не можуть забезпечити точний розподіл часу процесора між процесами і обробляти одночасно декілька переривань від зовнішніх пристроїв, що призводить до затримок. Для мінімізації цього впливу деякі виробники використовують ОС реального часу, які мають кращі механізми розподілу процесорного часу. Вони також часто виділяють функції, що вимагають точного часового управління (наприклад, обмін даними між кодеками і мережним інтерфейсом), на окремий високошвидкісний процесор.

2 ВИЗНАЧЕННЯ КОНЦЕПЦІЙ ДОСТУПНОСТІ ТА ВІДМОВОСТІЙКОСТІ

2.1 Основні поняття та визначення доступності та відмовостійкості

Доступність є ключовим параметром в області інформаційних технологій, який визначає спроможність системи або компонента виконувати свої функції у заданий проміжок часу без перебоїв. Вимірюється у відсотках, що вказують, яку частку часу система залишається доступною для користувачів.

Зазвичай доступність системи оцінюється за допомогою метрики SLA або договору про рівень обслуговування.

SLA – це частина договору між постачальником послуги та її клієнтом, що формалізує рівень обслуговування, який постачальник обіцяє надати. SLA може включати різні аспекти послуг, включно з доступністю, продуктивністю, часом відгуку на запити і більше.

SLA важливий як для постачальників, так і для клієнтів. Для клієнтів він визначає очікуваний рівень сервісу та гарантії, що їх можна вимагати у разі невиконання домовленостей. Для постачальників SLA є засобом демонстрації відданості якості обслуговування та допомагає уникнути нерозумінь та спорів з клієнтами.

SLA зазвичай визначає відсоток часу, протягом якого сервіс або система повинні бути доступними. Це часто виражається у відсотках доступності.

Для досягнення високої доступності можуть бути використані різноманітні методики та технології, такі як дублювання компонентів, балансування навантаження, резервне копіювання, автоматизоване відновлення після збоїв, моніторинг стану системи та інші.

Системи з високою доступністю часто використовуються в критичних додатках, таких як банківські системи, системи медичного обслуговування, телекомунікаційні мережі та інші, де простій або втрата даних може призвести до значних фінансових втрат або негативних наслідків для користувачів чи організації [7].

Однак слід зазначити, що повна доступність, тобто 100% робочий час без будь-яких перебоїв, є майже недосяжною в реальних умовах через різні фактори, такі як обслуговування обладнання, оновлення програмного забезпечення, несподівані апаратні чи програмні збої тощо. Тому ключовим є підхід до

планування, дизайну та обслуговування системи з урахуванням потреби в максимальній доступності для конкретних бізнес-вимог.

Відмовостійкість є ключовою характеристикою багатьох сучасних систем, вказуючи на здатність системи продовжувати свою роботу, незалежно від відмов окремих компонентів. Така характеристика в основному важлива для високонавантажених та критичних систем, особливо в ситуаціях, коли відмова може призвести до серйозних фінансових збитків, втрати даних або інших критичних наслідків.

Один з підходів до досягнення відмовостійкості - це резервування. Це передбачає наявність додаткового обладнання, програмного забезпечення або комунікаційних каналів, готових взяти на себе обов'язки збійного компонента. Такий підхід забезпечує високий рівень захисту від відмов, але може бути вартісним у впровадженні та підтримці [8].

Другий важливий аспект - розподіленість систем. В таких системах декілька незалежних вузлів працюють разом, щоб забезпечити неперервність сервісу, навіть при відмові одного або декількох вузлів. Розподілені системи використовують реплікацію даних та завдань серед різних вузлів, що дозволяє відновлювати операції без затримки в разі відмов. Технології такі як кластеризація та розподілені бази даних сприяють підвищенню відмовостійкості шляхом автоматичного перемикання між вузлами у разі відмови.

Інший ключовий компонент відмовостійкості - це моніторинг та виявлення відмов у реальному часі. Швидке виявлення та реагування на проблеми дозволяє системі автоматично відновлюватися або перенаправляти завдання на справні компоненти. Сучасні системи моніторингу можуть автоматично виявляти аномалії в роботі та сповіщати адміністраторів про потенційні проблеми.

Також важливо звертати увагу на проєктувальний етап системи. Проєктування з наголосом на відмовостійкість з самого початку може значно підвищити надійність системи в майбутньому. Це включає в себе вибір високоякісного обладнання, програмування з урахуванням можливих відмов та планування відновлення даних.

Таким чином, відмовостійкість - це не лише про технології, а й про стратегію та планування. Постійний аналіз ризиків, планування на випадок відмов та готовність до швидкого відновлення операцій є важливими компонентами створення і підтримки відмовостійкої системи.

2.2 Методи та технології забезпечення високої доступності та відмовостійкості в кластерних системах

Методи та технології забезпечення високої доступності та відмовостійкості стали важливими компонентами сучасних ІТ-інфраструктур. Завдяки їм компанії можуть забезпечувати безперебійну роботу своїх систем і служб, незалежно від можливих відмов та збоїв [9].

Висока доступність забезпечується за допомогою реплікації даних, автоматичного перемикання між серверами або дата-центрами та швидкого відновлення після відмов. Кластеризація, яка дозволяє групувати кілька серверів у єдиний віртуальний ресурс, є одним з ключових методів досягнення високої доступності. Якщо один із серверів у кластері відмовляє, завдання автоматично переключаються на інший, справний сервер.

Для забезпечення відмовостійкості активно використовуються резервне копіювання та реплікація даних. Це дозволяє відновлювати систему після відмови, не втрачаючи важливої інформації. Також важливо використовувати моніторинг систем, який може виявляти проблеми на ранніх стадіях та попереджати адміністраторів про потенційні відмови.

Віртуалізація також відіграє важливу роль у забезпеченні високої доступності та відмовостійкості. Віртуальні машини можна легко мігрувати між фізичними серверами, що дозволяє підтримувати роботу системи навіть при відмові окремого обладнання. Загальну архітектуру віртуалізації надано на рис. 2.1.

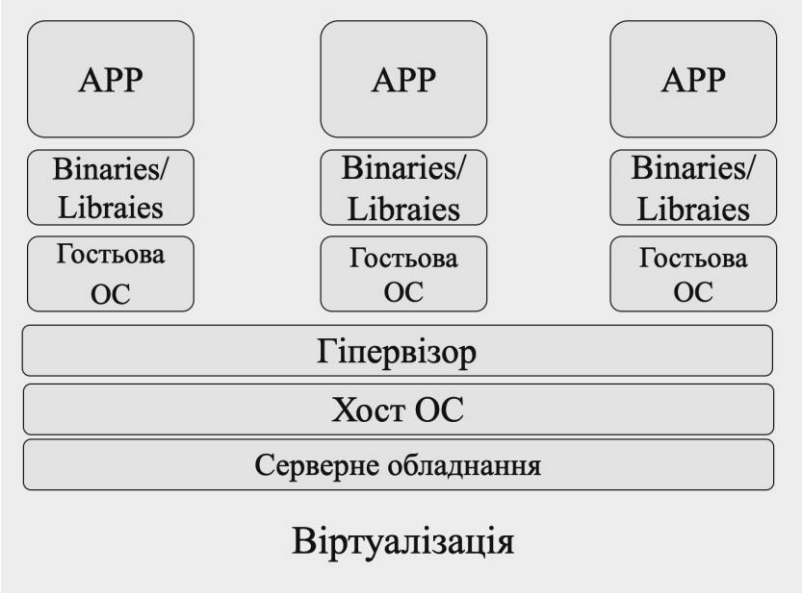

Рисунок 2.1 – Архітектура віртуалізації

Останніми роками все більше уваги приділяється хмарним технологіям, які забезпечують гнучкість та масштабованість систем, а також дозволяють автоматично розподіляти ресурси між декількома дата-центрами.

Для забезпечення безперебійної роботи систем і служб важливо регулярно перевіряти їх стійкість до різних видів відмов і використовувати найсучасніші методи та технології для їхнього усунення.

Сучасні рішення для забезпечення високої доступності та відмовостійкості також включають в себе георедундантність, коли дата-центри розташовані в різних географічних локаціях. Це не тільки розподіляє ризики, але і дозволяє забезпечити безперебійний доступ до ресурсів для користувачів з різних регіонів. Схема процесу неперервної інтеграції та розгортання для двох дата-центрів в Amazon Web Services (AWS) надана на рис. 2.2.

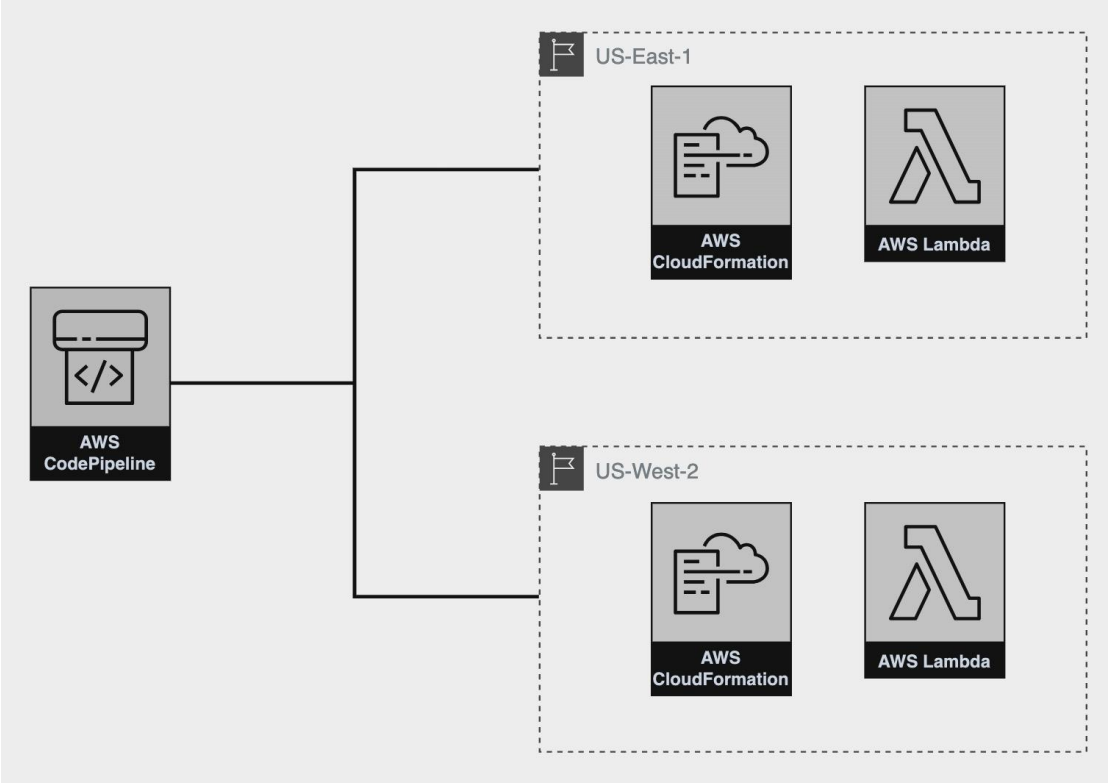

Рисунок 2.2 – Схема процесу неперервної інтеграції та розгортання в AWS

Додаткова увага приділяється системам детектування й автоматичного відновлення після відмов. Такі системи моніторять інфраструктуру в реальному часі, виявляють потенційні проблеми та автоматично запускають процедури відновлення для забезпечення безперебійної роботи сервісів.

Також важливо зазначити роль програмного забезпечення в забезпеченні

високої доступності та відмовостійкості. Сучасні системи управління даними, такі як бази даних та системи управління контентом, часто включають в себе механізми реплікації, кешування та резервного копіювання, що допомагає збільшити стабільність та доступність даних [10].

Забезпечення високої доступності та відмовостійкості потребує комплексного підходу, який поєднує апаратні, програмні та організаційні рішення. Важливо регулярно оновлювати технології, проводити аудит систем на предмет потенційних слабких місць та навчати персонал найкращим практикам у галузі забезпечення безперебійної роботи IT-інфраструктури.

2.2.1 Кластеризація на основі реплікації

Кластеризація на основі реплікації є ключовим елементом у сучасних розподілених системах, забезпечуючи високий рівень доступності та відмовостійкості. Основна ідея полягає у створенні декількох копій даних на різних вузлах кластера, що дозволяє системі продовжувати працювати без перебоїв навіть у випадку збоїв або відмов окремих компонентів. Реплікація не тільки підвищує надійність системи, але й може покращити продуктивність, оскільки запити до даних можуть оброблятися локально на кожному вузлі, зменшуючи час відгуку та навантаження на мережу.

Одним з головних викликів при реплікації даних є забезпечення їхньої послідовності та синхронізації між різними копіями. Це вимагає ретельного планування та впровадження механізмів узгодження, особливо в системах з високою швидкістю зміни даних. Існують різні стратегії реплікації, від строго послідовної, що гарантує, що всі копії завжди синхронізовані, до більш гнучких підходів, які дозволяють деяку розбіжність між копіями для підвищення продуктивності та масштабованості.

Ще одним аспектом реплікації є вибір між синхронною та асинхронною реплікацією. Синхронна реплікація забезпечує високий рівень послідовності, оскільки зміни в даних не вважаються завершеними, поки вони не будуть відтворені на всіх вузлах. Однак це може призвести до затримок, оскільки час відгуку системи залежить від найповільнішого вузла. Натомість, асинхронна реплікація дозволяє системі швидше реагувати, оскільки зміни в даних затверджуються незалежно від статусу реплікації, що може бути особливо корисним у великих та географічно розподілених системах.

Окрім вибору стратегії реплікації, важливим аспектом є управління навантаженням та моніторинг стану різних вузлів у кластері. Ефективне розподілення навантаження між вузлами може значно покращити загальну продуктивність та доступність системи. Це включає не тільки рівномірний розподіл запитів між вузлами, але й адаптацію до змін у навантаженні та можливих відмовах компонентів. Моніторинг стану вузлів дозволяє оперативно виявляти та реагувати на проблеми, забезпечуючи неперервну роботу системи.

Враховуючи важливість реплікації даних, особливу увагу слід приділити безпеці. Репліковані дані повинні бути захищені від несанкціонованого доступу та змін, особливо у випадках, коли вузли кластера розташовані в різних фізичних місцях або використовуються хмарні послуги. Це вимагає впровадження сильних механізмів автентифікації, шифрування та контролю доступу.

Загалом, кластеризація на основі реплікації є могутнім інструментом для побудови високодоступних та відмовостійких систем. Вона вимагає виваженого підходу до управління даними, узгодження між вузлами та забезпечення безпеки.

З огляду на сучасні вимоги до масштабованості та доступності великих розподілених систем, реплікація є ключовим компонентом, що забезпечує ефективну та надійну роботу додатків та послуг.

2.2.2 Варіант організації мережі серверів за допомогою Content Delivery Network

Content Delivery Network (CDN) - це мережа серверів, розгорнутих по всьому світу, яка оптимізує доставку веб-сторінок та іншого веб-контенту користувачам, враховуючи їх географічне положення. Основна мета CDN - скоротити час відгуку, тобто час, необхідний віддаленому серверу для відправлення даних користувачу.

Відомо, що швидкість завантаження веб-сторінки на комп'ютері користувача залежить від відстані, яка розділяє сервер і користувача. Причина цього полягає в особливостях технології TCP/IP, яка використовується для розповсюдження інформації в Інтернеті. Часові затримки в цій технології залежать від кількості маршрутизаторів, розташованих між сервером і користувачем, адже чим їх більше, тим вища ймовірність перебоїв у з'єднанні або втрат даних на перевантажених каналах.

Для мінімізації втрат часу через зазначені вище причини можна

використовувати технологію CDN (мережу доставки контенту). Вона дозволяє управляти навантаженням при передачі трафіку та розподілити навантаження між віддаленими серверами, таким чином зменшуючи навантаження на головний канал та мережні вузли. Зменшення кількості переходів між різними мережами під час передачі даних також сприяє прискоренню завантаження контенту з Інтернету та підвищенню якості даних.

Таким чином, розміщення контенту на децентралізованих серверах за допомогою CDN скорочує мережний шлях передачі даних, прискорюючи завантаження веб-сторінки. Розміщення серверів ближче до користувачів може підвищити загальну пропускну здатність системи. Наприклад, наявність одного порту зі швидкістю 100 Мбіт/с не гарантує таку ж швидкість на всіх ділянках мережі, оскільки пропускна здатність головного каналу може бути всього 10 Мбіт/с. Проте, використання 10 розподілених серверів може забезпечити сумарну пропускну здатність до 1000 = 10*100 Мбіт/с.

Явний недолік одноточкової дистрибуції зображений на рисунку 2.3. У разі відмови головного сервера, користувач не зможе отримати контент.

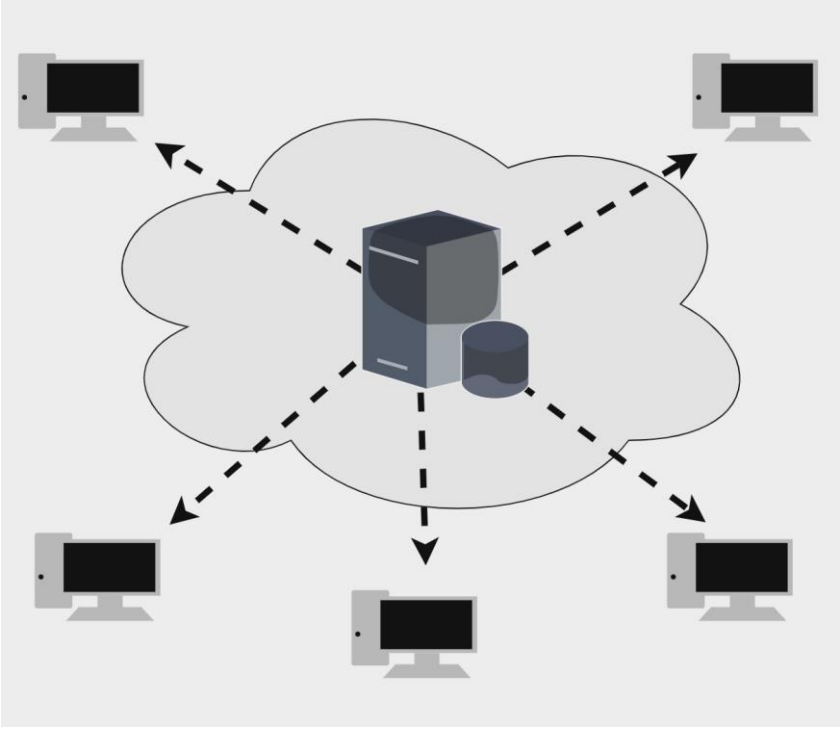

Рисунок 2.3 – Схема одноточкової дистрибуції контенту

Принцип роботи CDN полягає в кешуванні статичного контенту, такого як зображення, стилі Cascading Style Sheets (CSS), JavaScript-файли та мультимедійні

файли на серверах, розташованих якомога ближче до користувача. Це означає, що коли користувач запитує веб-сторінку, то статичний контент завантажується з найближчого сервера, а не з основного сервера, що значно прискорює завантаження.

Без використання CDN запит буде проходити через безліч мережених вузлів, що призведе до значної затримки. Однак, якщо веб-сайт використовує CDN, запит користувача буде перенаправлено на найближчий до нього сервер CDN. Це значно скорочує час завантаження, оскільки дані передаються на менші відстані.

У разі відмови одного з серверів, користувач все ще може отримати контент. Схему розподіленої мережі доставки контенту надано на рис. 2.4.

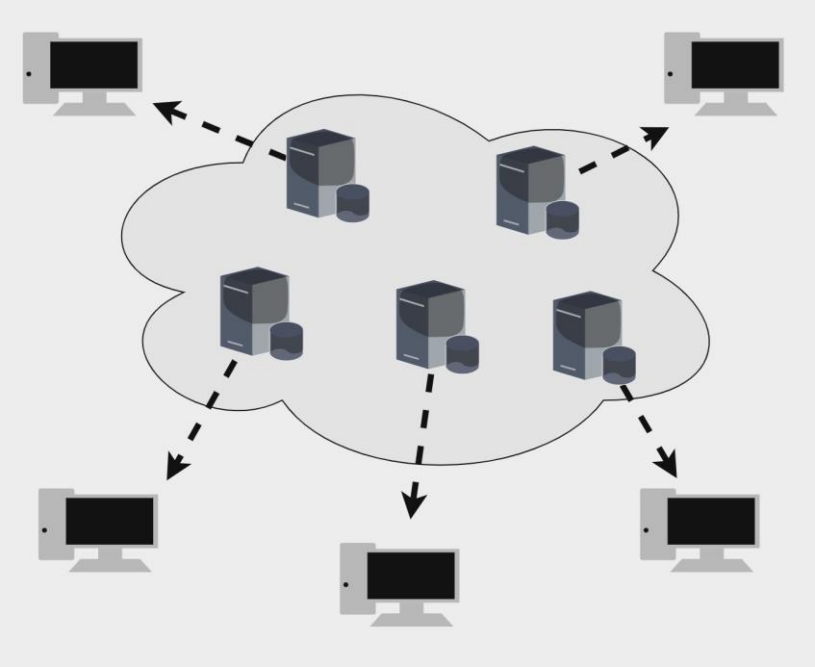

Рисунок 2.4 – Схема розподіленої мережі доставки контенту

Крім прискорення завантаження веб-сторінок, CDN також збільшує доступність та надійність веб-сайтів. У випадку збою чи перевантаження основного сервера запити можуть бути перенаправлено на інший сервер CDN, що забезпечує безперебійну роботу сайту. Також CDN захищає сайти від DDoS-атак, розподіляючи трафік на декілька серверів і тим самим знижуючи навантаження на кожен окремий сервер.

CDN не тільки корисний для великих сайтів зі значним трафіком, але й може бути вигідним для малих та середніх підприємств, оскільки покращує

користувацький досвід і допомагає підвищити рейтинг сайту в пошукових системах завдяки швидшому завантаженню сторінок.

Таким чином, використання CDN - це важливий крок у оптимізації продуктивності веб-сайтів, який допомагає покращити швидкість завантаження, збільшити доступність та безпеку, а також покращити загальний користувацький досвід. Зі зростанням обсягу он-лайн контенту та збільшенням кількості користувачів Інтернету по всьому світу, CDN стає все більш значущим інструментом в арсеналі веб-розробників та адміністраторів сайтів.

2.2.3 Балансування навантаження

Балансування навантаження є фундаментальним аспектом управління ресурсами в сучасних обчислювальних середовищах, особливо в контексті розподілених систем, таких як великі веб-сервіси, хмарні обчислення та корпоративні мережі. Основна мета балансування навантаження полягає у рівномірному розподілі робочого навантаження серед різних ресурсів, таких як сервери, мережні з'єднання або дискові пристрої, для оптимізації використання ресурсів, максимізації пропускної здатності, мінімізації часу відгуку та уникнення перевантаження будь-якого одного ресурсу.

Балансування навантаження може бути реалізоване на різних рівнях. На найнижчому рівні, воно може стосуватися розподілу обчислювальних процесів всередині одного сервера між його процесорами та ядрами. На більш високому рівні, балансування навантаження може бути застосоване до мережі серверів, розподіляючи запити або сесії між різними серверами. У складних системах, таких як хмарні обчислення, балансування навантаження може включати розподіл ресурсів між цілими центрами обробки даних, розташованими в різних географічних регіонах.

Існують різні стратегії та алгоритми для балансування навантаження. Найпростішим є круговий метод Round-Robin, де запити послідовно розподіляються між доступними серверами (рис. 2.5).

Метод Round-Robin є одним із найпростіших і найпоширеніших алгоритмів балансування навантаження, який застосовується для рівномірного розподілу обробки запитів між серверами, що входять до пулу. Цей метод працює за принципом циклічного вибору сервера для кожного нового запиту, тобто перший запит надсилається до першого сервера, другий – до другого, і так далі, поки не

буде досягнуто останнього сервера в списку, після чого цикл повторюється знову з першого сервера.

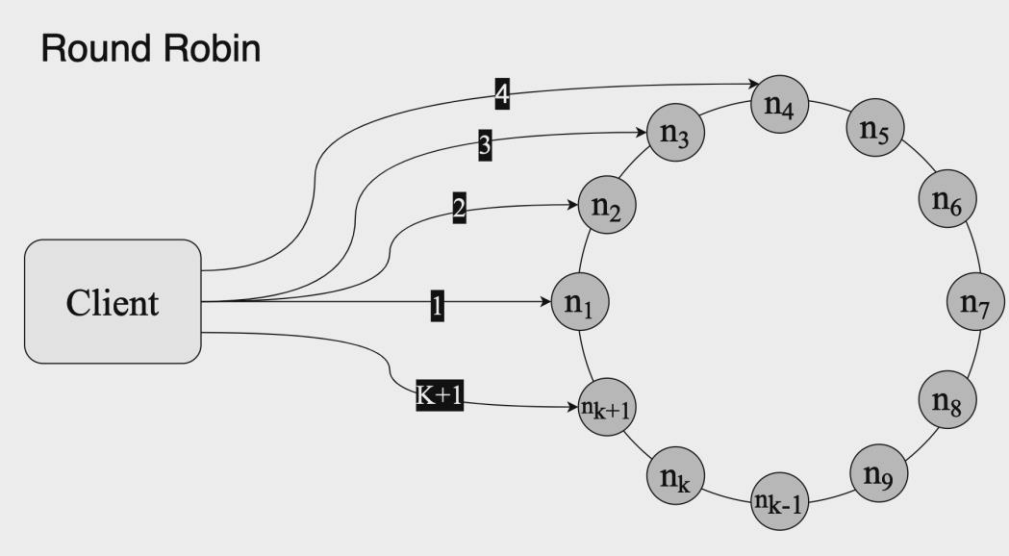

Рисунок 2.5 – Схема алгоритму Round Robin для балансування навантаження

Перевагою Round-Robin є його простота та легкість у впровадженні, оскільки він не вимагає складного моніторингу стану серверів або врахування їхнього поточного навантаження. Це робить Round-Robin відмінним вибором для систем з великою кількістю однотипних та однаково потужних серверів, де навантаження очікується бути приблизно однаковим на всіх вузлах.

Однак, метод Round-Robin має певні недоліки. Оскільки він не бере до уваги реальне навантаження на сервери, можлива ситуація, коли окремі сервери можуть бути перевантажені, тоді як інші - недостатньо завантажені. Це може статися, наприклад, коли запити є дуже нерівномірними за своєю обчислювальною вартістю або коли окремі сервери мають різну пропускну здатність. У таких випадках, більш доцільним може бути застосування вдосконаленого методу Round-Robin, який враховує ваги серверів. Такий підхід дозволяє розподіляти навантаження з урахуванням потужності кожного сервера, тобто сервери з вищою пропускною здатністю отримують більше запитів порівняно з менш потужними. Це допомагає уникнути ситуацій, коли потужні сервери недостатньо завантажені, а менш потужні - перевантажені.

Також існують інші варіації Round-Robin, такі як Dynamic Round-Robin, де алгоритм адаптується до змін у навантаженні в реальному часі, змінюючи розподіл запитів на основі поточного стану серверів. Це може включати в себе

моніторинг часу відгуку серверів, їх доступності та інших параметрів, що дозволяє алгоритму бути більш гнучким та ефективним у розподілі навантаження.

В цілому, метод Round-Robin і його варіації залишаються популярними через свою простоту та ефективність у багатьох сценаріях, особливо коли необхідно швидкий та простий спосіб балансування навантаження.

Важливим елементом балансування навантаження є виявлення та управління відмовами. У високодоступних системах балансувальники навантаження повинні швидко виявляти непрацюючі або перевантажені сервери та перенаправляти трафік на інші, працюючі вузли. Це забезпечує безперервну доступність сервісів навіть у разі локальних збоїв або відмов окремих компонентів. Для цього часто використовуються механізми здоров'я (health checks), які періодично перевіряють стан серверів.

Балансування навантаження також відіграє ключову роль у масштабуванні систем. У міру зростання навантаження, наприклад, через збільшення кількості користувачів веб-сервісу, система може автоматично додавати додаткові ресурси, такі як сервери або обчислювальні потужності. Це дозволяє зберігати високий рівень продуктивності та доступності без необхідності вручну переконфігурувати всю систему.

Однією зі складностей балансування навантаження є управління сесіями, особливо у випадках, коли потрібно збереження стану користувача. Наприклад, у веб-додатках, де користувач здійснює серію пов'язаних запитів, важливо забезпечити, щоб всі ці запити оброблялися одним і тим же сервером. Для цього використовуються різні механізми зберігання сесій, такі як стійке балансування навантаження (persistent load balancing) або розподілене зберігання сесій.

Також важливим аспектом є адаптивність системи балансування навантаження. Сучасні системи повинні бути гнучкими, щоб швидко адаптуватися до змін у вхідному трафіку та робочому навантаженні. Це включає в себе здатність до масштабування як вгору (додавання ресурсів при зростанні навантаження), так і вниз (відключення ресурсів при зменшенні навантаження), що дозволяє оптимізувати витрати на інфраструктуру та підтримувати високий рівень продуктивності. Автоматизація та використання штучного інтелекту можуть допомогти в прогнозуванні майбутніх навантажень та оптимізації ресурсів відповідно до поточних потреб.

Ще одним важливим аспектом є безпека в контексті балансування навантаження. Оскільки балансувальники навантаження часто є точками входу до

мережі або системи, вони мають бути забезпечені від різних видів атак, таких як DDoS (розподілені атаки типу «відмова у обслуговуванні») та інших загроз. Це вимагає впровадження розширених механізмів безпеки, таких як фільтрація трафіку, захист від зловмисного програмного забезпечення та регулярне оновлення безпеки.

Для захисту від DDoS атак потрібно використовувати комплексний підхід. Це включає в себе як технічні рішення, так і організаційні заходи. Технічні рішення можуть включати в себе використання анти-DDoS рішень, таких як спеціалізоване обладнання чи хмарні сервіси, що моніторять і фільтрують трафік, виявляючи та блокуючи шкідливий трафік.

Використання хмарних технологій може значно знизити ризик DDoS атак. Хмарні провайдери, такі як AWS, Google Cloud або Microsoft Azure, зазвичай пропонують вбудовані засоби захисту від DDoS, які можуть масштабувати ресурси та фільтрувати трафік для запобігання перевантаження систем.

Одним з ефективних методів є розподіл ресурсів і створення резервних копій критично важливих компонентів. Це може включати в себе розподіл навантаження через кілька серверів або центрів обробки даних, що робить систему менш вразливою до атак на один конкретний ресурс.

Таким чином, балансування навантаження є критично важливим для підтримання високої продуктивності, доступності та надійності в сучасних обчислювальних середовищах, що вимагає уважного підходу до планування, реалізації та керування, а також постійного моніторингу та адаптації до змінюваних умов. Завдяки інноваціям у технологіях та розвитку нових методів, балансування навантаження продовжує еволюціонувати, пропонуючи ще більшу гнучкість та ефективність для задоволення вимог сучасних обчислювальних систем.

3 РІШЕННЯ ДЛЯ ЗАБЕЗПЕЧЕННЯ ДОСТУПНОСТІ ТА ВІДМОВОСТІЙКОСТІ КЛАСТЕРА

3.1 Аналіз методів реплікації даних

Реплікація даних - це важливий процес у сфері інформаційних технологій та обчислювальної науки, який використовується для забезпечення надійності та доступності даних в розподілених системах. Цей процес полягає у створенні та підтримці копій даних на різних серверах або вузлах, щоб забезпечити можливість доступу до даних у випадку відмови обладнання, програмного забезпечення чи інших непередбачуваних подій.

Реплікація може бути використана для забезпечення доступу до даних у випадку відмови сервера, підвищення продуктивності через розподілення навантаження або для забезпечення географічної резервності даних.

Існують різні типи реплікації, такі як повна реплікація, де всі дані копіюються повністю, часткова реплікація, де копіюються лише певні дані або таблиці та інші.

У повній реплікації всі дані, що зберігаються в основній базі даних, копіюються повністю на одну чи декілька інших реплікативних серверів. Це означає, що кожна реплікативна копія містить усю інформацію, і всі зміни, які вносяться до основної бази даних, автоматично відображаються на всіх реплікативних серверах. Повна реплікація забезпечує високу доступність даних, але може вимагати значних ресурсів для зберігання та синхронізації даних.

У випадку часткової реплікації лише певні дані або таблиці з основної бази даних копіюються на реплікативні сервери. Це може бути корисним, коли не всі дані мають однакову важливість або коли необхідно зменшити обсяг реплікації для економії ресурсів.

Синхронна реплікація передає дані в реальному часі, очікуючи підтвердження успішного запису на всіх копіях. Асинхронна реплікація передає дані без очікування підтвердження, що дозволяє зберегти високу продуктивність.

У випадку асинхронної реплікації можуть виникати конфлікти даних, і важливо мати механізми для їх вирішення та забезпечення консистентності даних.

Реплікацію даних повинно бути супроводжено заходами забезпечення доступності та захисту даних від несанкціонованого доступу.

Реплікація даних є важливою стратегією для забезпечення надійності, доступності та продуктивності систем обробки даних. Вона застосовується у багатьох галузях, включаючи розподілені бази даних, хмарні обчислення та розподілені системи збереження даних. Ефективне використання реплікації даних вимагає ретельного планування та розробки стратегій управління даними.

Синхронна реплікація вимагає, щоб всі зміни в даних було негайно відображено на всіх репліках, що забезпечує строгу консистентність даних. Вона важлива для систем, де потрібна гарантія, що всі копії даних завжди мають однаковий стан (рис. 3.1).

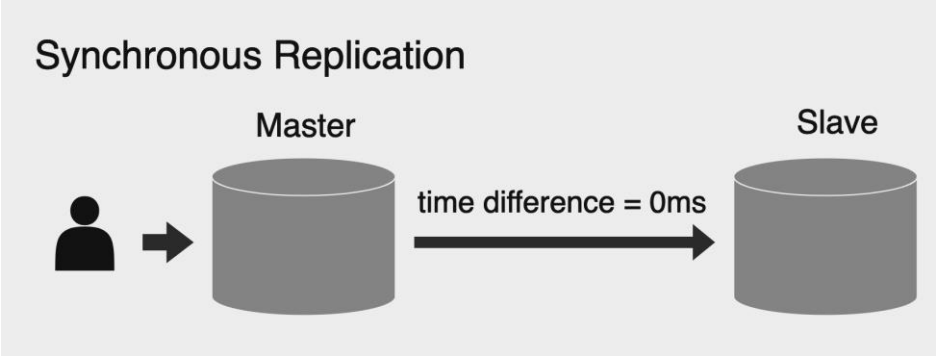

Рисунок 3.1 – Діаграма синхронної реплікації

Такий метод реплікації ідеально підходить для транзакційних систем, де важлива точність та надійність даних. Однак, синхронна реплікація може призводити до зниження продуктивності, оскільки операції з даними завершуються лише після їх копіювання на всі репліки, що може вимагати додаткового часу та ресурсів.

Основна відмінність синхронної реплікації від асинхронної полягає в тому, що дані передаються в реальному часі, і операції очікують підтвердження успішного запису на всіх реплікативних серверах. Це означає, що обмін даними є блокуючим, і виконання операції залежить від завершення реплікації.

Синхронна реплікація забезпечує високий рівень надійності, оскільки дані не вважаються успішно записаними, поки вони не збережені на всіх серверах. Це допомагає уникнути можливих конфліктів та забезпечує консистентність даних на всіх копіях.

Синхронна реплікація може впливати на продуктивність системи через затримки, пов'язані із очікуванням завершення реплікації. Для систем з великою кількістю даних та вимогами до низької затримки це може бути важливим

фактором.

Синхронна реплікація часто використовується у важливих застосунках, де надійність та безпека даних є критично важливими. Це може включати фінансові системи, системи управління базами даних та інші.

Синхронна реплікація є потужним інструментом для забезпечення надійності та консистентності даних у розподілених системах. Однак вона може вимагати великих ресурсів і впливати на продуктивність системи через затримки. Вибір між синхронною та асинхронною реплікацією повинен бути зроблений з огляду на конкретні вимоги та характеристики додатку.

Асинхронна реплікація даних полягає у збереженні копій даних на вторинних системах з певною затримкою відносно первинної системи. Цей метод використовується для зниження впливу процесу реплікації на загальну продуктивність системи. Дані передаються на вторинні системи після їх оновлення на первинній системі, що може призводити до невеликих розбіжностей у стані даних між системами на момент реплікації. Цей підхід є ефективним для систем, де миттєва консистентність даних не є критично важливою, але важлива висока продуктивність та мінімізація навантаження на мережу. Асинхронна реплікація використовується у системах, де важлива висока продуктивність, і де невеликі розбіжності в даних не є критичними. Це може включати системи великих даних, де обсяги інформації є величезними і де важливіше швидко обробляти дані, ніж забезпечувати їх миттєву консистентність. Однак це також призводить до ризиків, оскільки в момент збою системи, останні дані можуть бути не синхронізовані, що потребує додаткових механізмів відновлення та перевірки даних (рис. 3.2).

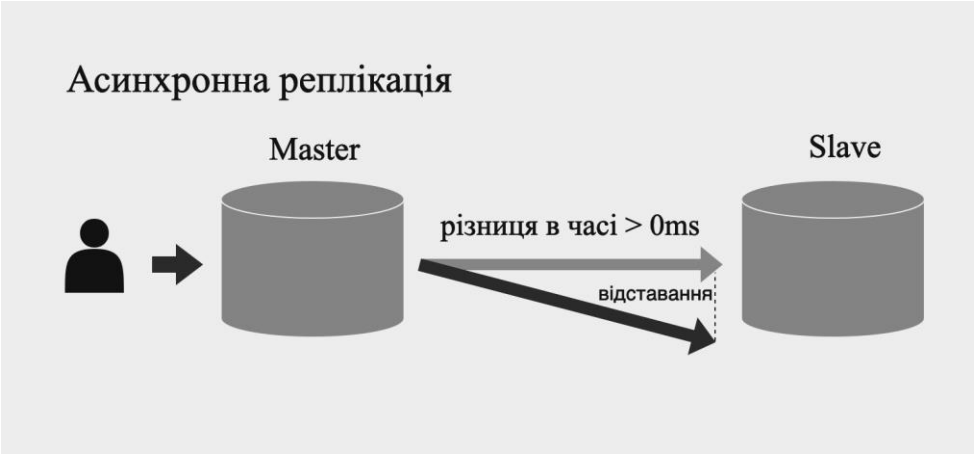

Рисунок 3.2 – Діаграма асинхронної реплікації

Головною особливістю асинхронної реплікації є відсутність необхідності у синхронній передачі даних. Джерело може надсилати дані на призначення, не блокуючи свою роботу та не очікуючи завершення операції. Це дозволяє підтримувати високу продуктивність та швидкодію в розподіленій системі.

Однією з основних переваг асинхронної реплікації є висока продуктивність та здатність обробляти великі обсяги даних. Джерело може негайно продовжувати роботу, що особливо важливо для систем із вимогами до низької затримки.

Проте асинхронна реплікація також має свої недоліки. Можлива втрата даних, якщо джерело відмовить до завершення реплікації. Крім того, можуть виникати проблеми з консистентністю даних між джерелами, що вимагають розробки механізмів для їх вирішення.

Асинхронна реплікація знаходить широке застосування в сучасних розподілених системах. Вона використовується для забезпечення надійності даних у базах даних, для забезпечення відмовостійкості у системах з обліковими записами, для забезпечення доступності даних у хмарних обчисленнях і багатьох інших сценаріях.

У розподілених системах бази даних часто реплікуються для забезпечення доступності даних та надійності. Асинхронна реплікація дозволяє оновлювати копії бази даних без затримок для користувачів. Асинхронна реплікація використовується для створення резервних копій даних та їх зберігання на віддалених серверах. Це допомагає у відновленні даних у випадку втрати або пошкодження оригінальних даних.

У хмарних сервісах дані також можуть реплікуватися на різні сервери.

Реплікація на рівні блоків – це метод, який забезпечує синхронізацію даних шляхом копіювання блоків інформації між різними фізичними пристроями зберігання. У цьому методі, зміни, що відбуваються у блоку даних на одному диску, автоматично реплікуються на інший диск. Це дозволяє створити точні копії даних на різних пристроях, забезпечуючи високий рівень відмовостійкості і доступності даних.

Реплікація на рівні блоків використовується для синхронізації фізичних дискових просторів шляхом копіювання блоків даних між системами зберігання. Це дозволяє створювати дзеркальні копії даних на рівні апаратного забезпечення, забезпечуючи високий рівень відмовостійкості. Важливою особливістю цього методу є те, що він дозволяє відновлювати систему на рівні блоків без необхідності відновлення кожного файлу окремо. Однак, цей метод може

вимагати більше місця для зберігання та може бути менш гнучким у порівнянні з файлово-орієнтованими методами реплікації.

У основі реплікації на рівні блоків лежать ключові принципи, включаючи копіювання фрагментів даних на різних серверах або вузлах мережі, збереження цілісності даних між репліками та впровадження розподіленого управління для виявлення і управління конфліктами та синхронізації реплік. Це забезпечує належну координацію та взаємодію між різними копіями даних.

Такий підхід широко використовується у системах Redundant Array of Independent Disks (RAID), які гарантують, що навіть у разі відмови одного диску дані залишаються доступними та незмінними (рис. 3.3).

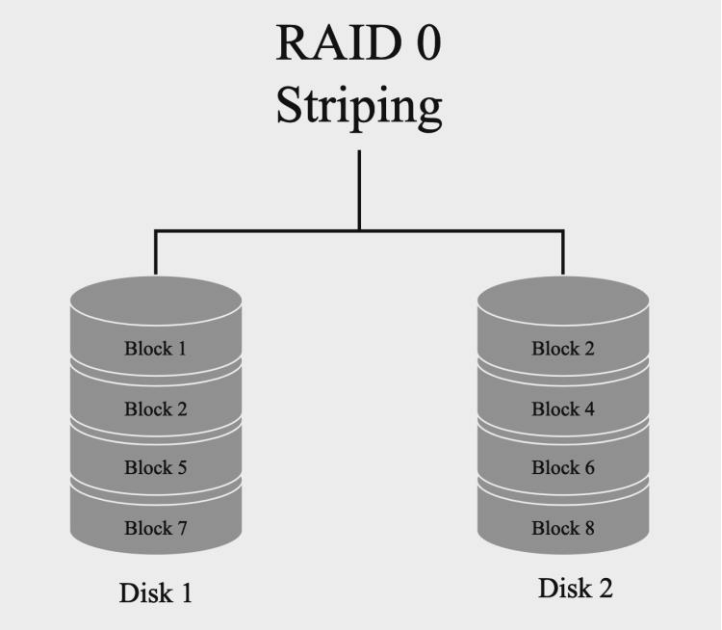

Рисунок 3.3 – Діаграма блочної реплікації RAID

Реплікація на рівні блоків може приймати різні форми. У випадку Master-master реплікації дозволяється одночасний доступ до даних для читання та запису на двох або більше вузлах. Це допомагає підвищити доступність та швидкість доступу до даних.

Master-slave реплікація передбачає, що один вузол є головним (майстером) і може записувати дані, тоді як інші (slave) можуть тільки читати або використовувати дані в режимі «тільки для читання». Це забезпечує цілісність даних, але може обмежити швидкість запису.

Цей підхід до реплікації має широкий спектр застосувань. Він може бути використаний для оптимізації роботи з базами даних, де реплікація блоків даних

допомагає забезпечити надійність та швидкодіючий доступ до інформації. Також цей підхід застосовується в системах з розподіленою файловою системою, що полегшує роботу з файлами та забезпечує їх доступність.

Об'єктна реплікація – це метод реплікації даних, який копіює або синхронізує об'єкти даних (які можуть включати файли, блоки даних чи інші структуровані формати) між різними системами зберігання (рис. 3.4).

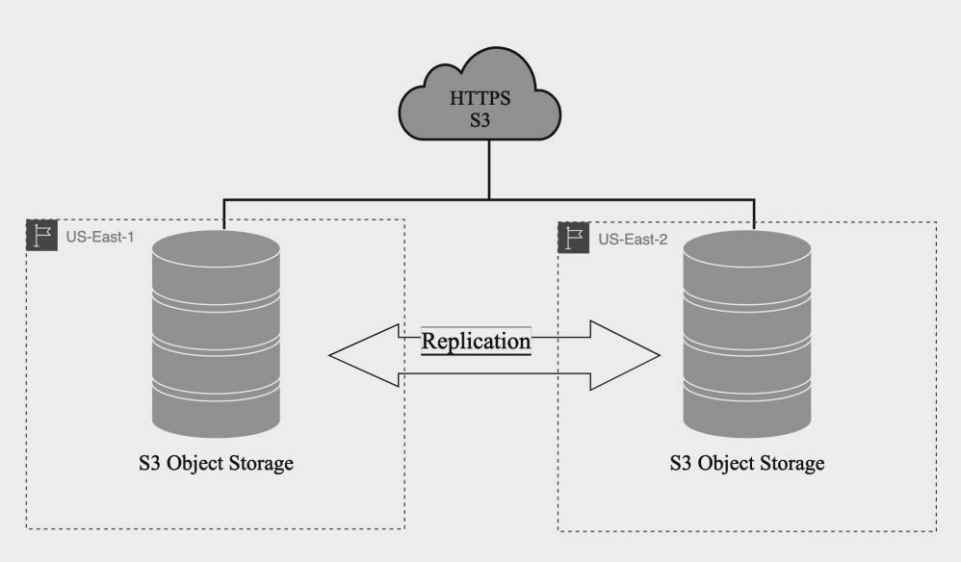

Рисунок 3.4 – Діаграма об'єктної реплікації

Об'єктна реплікація відрізняється від традиційних методів, таких як реплікація на рівні файлів або блоків, оскільки зосереджена на реплікації самого об'єкта даних з його метаданими та унікальними характеристиками. Цей метод широко використовується в хмарних та розподілених системах зберігання, де необхідно забезпечити надійність та доступність великих обсягів даних. Він дозволяє легко масштабувати системи зберігання і забезпечує швидкий доступ до даних, але також може вимагати складнішого управління та забезпечення додаткового простору для зберігання.

Об'єктна реплікація є важливим поняттям у сфері розподілених систем і обчислень, і вона відіграє важливу роль у забезпеченні надійності та доступності розподілених систем. Цей підхід дозволяє створювати копії об'єктів або даних і розміщувати їх на різних серверах або вузлах мережі. Це забезпечує не тільки зменшення ризику втрати даних внаслідок відмови обладнання або програмного забезпечення, але й підвищення швидкості обробки запитів користувачів.

Основні принципи об'єктної реплікації передбачають створення копій

об'єктів і синхронізацію змін в цих копіях. Це означає, що будь-яка зміна, яка відбувається в одному екземплярі об'єкта, автоматично відображається в усіх інших репліках цього об'єкта. Таким чином, забезпечується узгодженість та цілісність даних в розподіленій системі.

У світі об'єктної реплікації існують різні види. Активна реплікація передбачає, що всі репліки об'єктів активні і можуть приймати запити від клієнтів. Пасивна реплікація, навпаки, означає, що тільки одна репліка активна, а інші є резервними копіями, які активуються у випадку відмови основної репліки. Також існує змішана реплікація, де поєднуються як активні, так і пасивні репліки для оптимізації доступності та швидкості обслуговування користувачів.

Об'єктна реплікація має широкий спектр застосувань. Вона дозволяє підвищити надійність і доступність розподілених систем, зменшуючи ризик втрати даних і забезпечуючи неперервну роботу послуг для користувачів. Також цей підхід використовується для підвищення пропускної здатності великих об'ємів даних, що є важливим аспектом у сучасному світі обчислень та мереж.

Проте об'єктна реплікація не лише вирішує проблеми, але і вносить виклики. Один з них - це керування конфліктами, які виникають при оновленні реплік і потребують розробки механізмів для їх вирішення. Крім того, використання об'єктної реплікації може збільшити витрати на мережу через додаткове навантаження на неї для синхронізації реплік. Також розробка і підтримка систем об'єктної реплікації може бути складною завданням через їхню складність та потребу в управлінні багатьма репліками.

Кожен з цих методів має свої переваги та обмеження, тому важливо вибрати підхід, що найкраще відповідає специфічним вимогам та характеристикам системи.

3.2 Алгоритм реплікації даних Raft

Розподілені системи, як важлива частина сучасних обчислювальних мереж, вимагають надійних механізмів для досягнення консенсусу та управління даними. Алгоритм Raft, розроблений як альтернатива складнішим алгоритмам, таким як Paxos, забезпечує ефективне та зрозуміле рішення для реплікації журналу в розподілених системах. Цей алгоритм вирішує проблему досягнення консенсусу, забезпечуючи, що кожен вузол в системі може безпечно та надійно виконувати однакові операції в однаковому порядку.

В роботі проведено аналіз літератури використання алгоритму консенсусу Raft. В роботі [11] запропоновано використання алгоритму консенсусу Raft як більш зрозумілого й модульного підходу в порівнянні з іншими алгоритмами консенсусу.

Імовірність поділу розподіленої мережі з використанням простого, але точного аналітичного підходу, проаналізовано у [12]. Імовірність поділу мережі представлено як функцію розміру мережі, швидкості втрати пакетів та тривалості тайм-ауту. Для перевірки використовувався симулятор Raft. За допомогою моделі запропоновано теоретично передбачити час та ймовірність поділу мережі, а також оптимізувати параметри алгоритму консенсусу Raft.

Модель для імітації Blockchain з використанням теорії масового обслуговування запропоновано у [13]. Модель створено з використанням черги M/M/1 у якості пулу пам'яті та черги M/M/c у якості пулу майнінгу. Цей метод простий, але ефективний для виявлення важливих показників мережі: кількості транзакцій на блок, часу майнінгу кожного блоку, пропускної здатності системи/транзакцій за секунду, кількості пулів пам'яті, часу очікування в пулі пам'яті, кількості непідтверджених транзакцій у всій системі, загальної кількості транзакцій та кількості згенерованих блоків.

Модифікований консенсусний алгоритм стійкості до відмови для вирішення проблем масштабованості Istanbul Byzantine Fault Tolerant (IBFT) в приватних Blockchain-системах розглянуто в [14]. Використано інструмент для тестування Blockchain Hyperledger Caliper, який дозволяє оцінити продуктивність платформ з відкритим вихідним кодом, таких як Quorum та Hyperledger Fabric. Використання Fabric протоколів Solo і Raft як служби замовлення транзакцій у версії 1.4 забезпечує надійну пропускну здатність та низьку затримку, що робить Hyperledger Fabric хорошим кандидатом для корпоративного рішення на основі Blockchain. З іншого боку, реалізація Raft та IBFT у Quorum як протоколів консенсусу, які відповідають за підтримку ланцюжка блоків, робить його більш логічним варіантом для тестування модифікацій.

В роботі [15] запропоновано механізм вибору лідерів за допомогою алгоритму Raft з урахуванням стабільності мережі. Розроблений метод знизив можливість поділу мережі за допомогою Raft. Крім того, це дало змогу запобігти подальшому зниженню продуктивності, коли мережа вузлів Blockchain нестабільна.

У роботі [16] розглянуті функціональні переваги та недоліки алгоритму

Raft. Розглянуто наступні характеристики: безпека, довіра між учасниками, пропускна здатність та масштабованість. Зроблено висновок, що жоден алгоритм, окрім Raft, не продемонстрував повного домінування у всіх аспектах порівняння.

У роботі [17] розглянуто чергу M/G/1 з ланцюгом Маркова з дискретним часом Discrete-Time Markov Chain (DTMC) у системі Blockchain. Теорію масового обслуговування теоретично проаналізовано для процесів генерації блоків та побудови Blockchain. Середній час підтвердження транзакції, кількість транзакцій у черзі у точці прибуття, кількість транзакцій у точках відправлення та інші параметри спостерігалися за допомогою Python.

Крім того, розглянуто марківську модель масового обслуговування форми Mb/M/1 як теоретичну основу для аналізу системи Blockchain на основі доказу повноважень, яка фокусується на стохастичній поведінці транзакцій та для аналізу консенсусного протоколу повноважень. Представлено дані, які проведені на Python. Зазначено, що оптимальна пропускна здатність блоку досягається при невеликій кількості транзакцій.

3.2.1 Основи алгоритму Raft

У контексті розподілених систем, реплікація даних є ключовим фактором для забезпечення високої доступності та витривалості. Raft дозволяє системам ефективно керувати реплікацією, забезпечуючи консистентність даних між вузлами. Це особливо важливо в сценаріях, де втрата або неконсистентність даних може призвести до серйозних наслідків.

Алгоритм Raft базується на концепції лідера. Всі вузли в кластері є або слідкувачами (followers), або кандидатами, поки один з них не стає лідером. Лідер керує процесом реплікації даних, відправляючи оновлення журналу іншим вузлам. Такий підхід спрощує процес реплікації та знижує ймовірність розбіжностей у даних.

Розробка алгоритму Raft була мотивована потребою в більш зрозумілому та менш складному рішенні для досягнення консенсусу в розподілених системах, ніж наявні на той час алгоритми. З часом Raft здобув популярність у спільноті розробників завдяки своїй простоті та ефективності. Його чітка структура та легкість у розумінні зробили його популярним вибором для багатьох великих розподілених систем.

У контексті Kubernetes, Raft може бути використано для крування

розподіленою системою еталонів (etcd), яка є ключовою складовою для зберігання та синхронізації конфігурації кластера Kubernetes. Etcd використовує Raft для забезпечення сильної консистентності (strong consistency) стану кластера, що є критично важливим для забезпечення надійної та передбачуваної оркестрації контейнерів.

Etcd використовує Raft - алгоритм консенсусу, розроблений для управління реплікованим журналом у розподіленій системі. Raft дозволяє кластеру etcd працювати як єдиний цілісний компонент, незважаючи на фізичну розподіленість між вузлами. Це досягається завдяки реплікації журналу команд між всіма вузлами, що входять до кластеру etcd, забезпечуючи сильну консистентність та високу доступність. Консенсусна архітектура etcd з використанням алгоритму Raft показана на рис. 3.5.

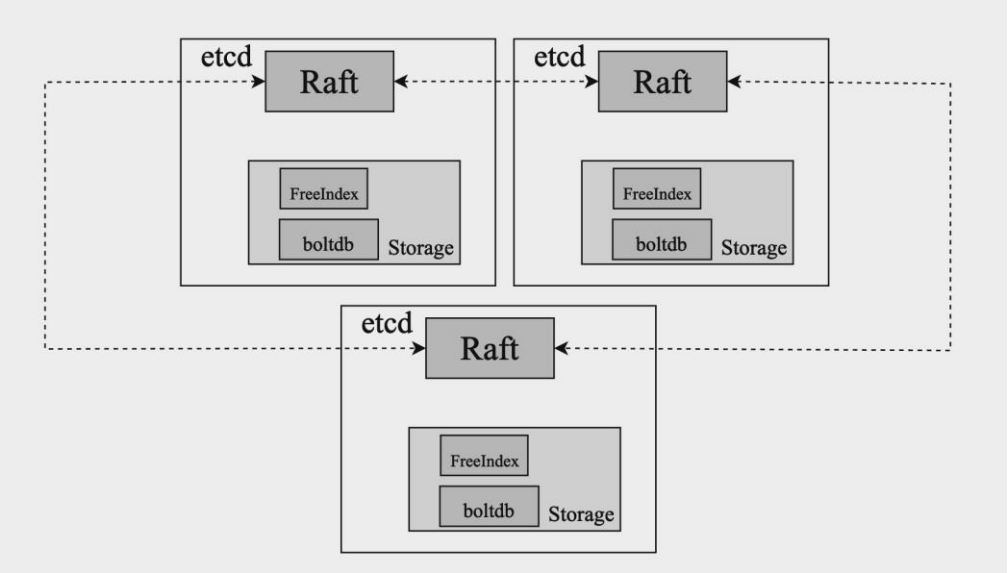

Рисунок 3.5 – Консенсусна архітектура etcd з використанням алгоритму Raft

Надійність Kubernetes в значній мірі залежить від надійності etcd, оскільки etcd зберігає важливий стан, такий як конфігурація мережі, сервіси, політики безпеки та інше. Якщо дані в etcd не будуть консистентними або якщо сервіс стане недоступним, це може призвести до серйозних збоїв в роботі кластера.

Raft гарантує, що навіть у разі збоїв у кількох вузлах, etcd зможе зберігати стан кластеру без втрат. Коли лідер отримує зміни стану, він не застосовує ці зміни до свого локального стану до тих пір, поки не отримає підтвердження, що зміни були репліковані на більшість вузлів. Такий підхід запобігає «розходженню мозку» (split-brain), коли різні частини кластера можуть мати суперечливі стани.

Завдяки використанню etcd та Raft, Kubernetes може забезпечити високий рівень витривалості та надійності, необхідних для сучасних розподілених систем. Raft особливо корисний для середовищ, де потрібна сильна консистентність та здатність швидко відновлюватися після збоїв.

Однією з основних характеристик Raft є його структура з чітко визначеними ролями (лідер, кандидат, слідкувач) та простими правилами переходу між цими станами. Це робить процес лідерства та реплікації даних прозорим та передбачуваним. Крім того, використання випадкових тайм-аутів для ініціації виборів знижує ймовірність конфліктів та забезпечує ефективний вибір лідера. Схему взаємодії лідера та послідовників в алгоритмі Raft показано на рис. 3.6.

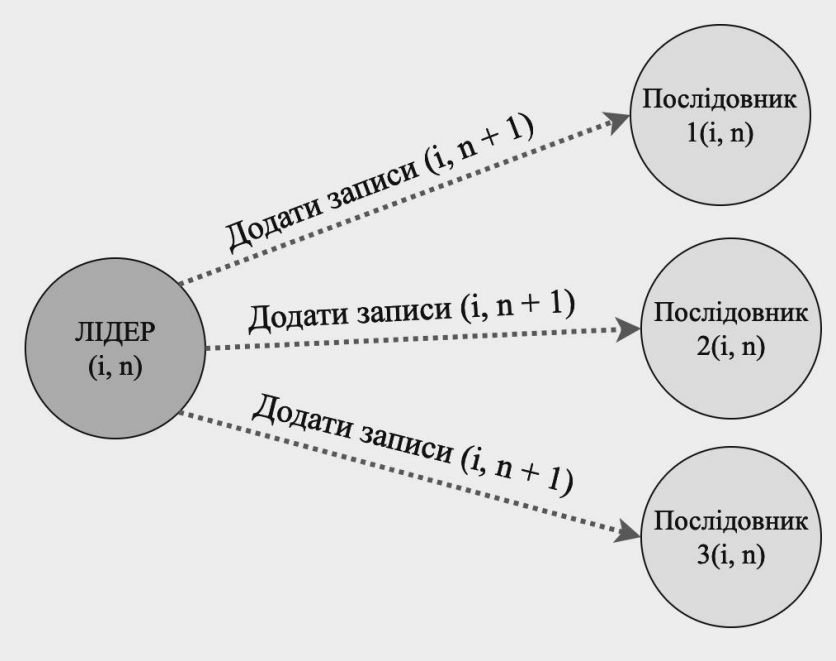

Рисунок 3.6 – Схема взаємодії лідера та послідовників в алгоритмі Raft

На сьогоднішній день Raft є одним з основних алгоритмів для досягнення консенсусу у розподілених системах. Його здатність забезпечувати надійну реплікацію даних та високий рівень доступності робить його незамінним інструментом в арсеналі розробників, що працюють із складними розподіленими системами.

Простота методу консенсусу алгоритму Raft робить його легшим для розуміння та застосування в порівнянні з іншими алгоритмами консенсусу, такими як Paxos. Також, Raft забезпечує кращі гарантії безпеки, ніж існуючі алгоритми консенсусу. Він гарантує, що система залишається послідовною, навіть

коли відбувається кілька збоїв або розділів мережі. Відомі блокчейн-платформи, такі як R3 Corda та Quorum, що використовують Raft як протокол консенсусу.

3.2.2 Процес вибору лідера

У Raft лідер відіграє ключову роль у координації дій між різними вузлами кластера. Лідер не тільки відповідає за реплікацію журналу записів до інших вузлів, але й гарантує їх консистентність та порядок. Вибір лідера є критичним кроком у роботі Raft, оскільки від цього залежить стабільність та ефективність усієї системи.

Процес вибору лідера в Raft ініціюється, коли вузол переходить у стан кандидата. Це відбувається у випадку, якщо вузол не отримує достатньої кількості серцебиттів (heartbeats) від поточного лідера протягом певного тайм-ауту. Кандидат ініціює вибори, генеруючи голосування серед інших вузлів кластера.

Під час виборчого процесу, кандидат відправляє запити на голосування до інших вузлів у кластері. Щоб вузол став лідером, він повинен отримати більшість голосів від інших вузлів. У випадку, якщо декілька вузлів стають кандидатами одночасно, система може не досягнути консенсусу, що призведе до повторення виборчого процесу після наступного тайм-ауту. Даний процес показано на діаграмі (рис. 3.7).

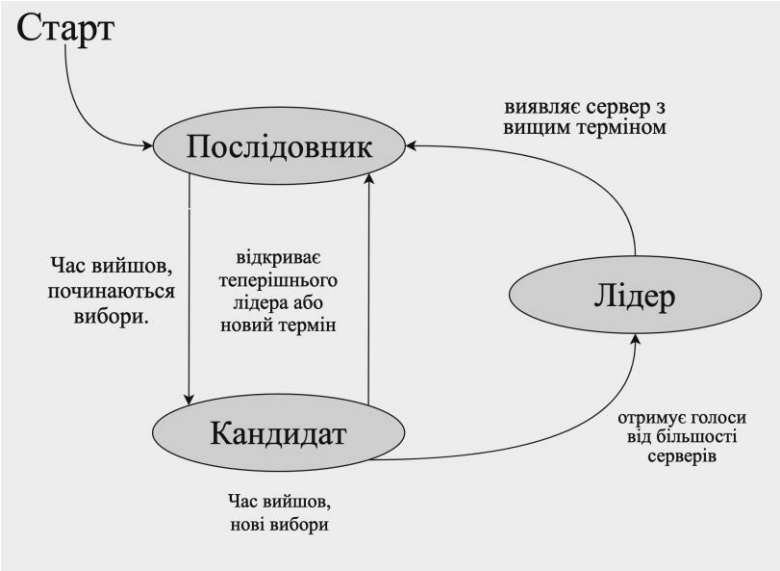

Рисунок 3.7 – Діаграма станів виборів лідера в алгоритмі Raft

Тайм-аути в алгоритмі Raft є випадково згенерованими для кожного вузла,

що запобігає одночасному переходу декількох вузлів у стан кандидата. Цей механізм знижує ймовірність конфліктів та забезпечує швидкість реагування системи на втрату лідера, забезпечуючи неперервність роботи. Вибір лідера є не лише технічним процесом, але й механізмом, що гарантує, що всі вузли працюють.

3.2.3 Реплікація журналу

Реплікація журналу в алгоритмі Raft є фундаментальним процесом, що забезпечує консистентність даних між вузлами в розподіленій системі. Цей процес відбувається під керівництвом лідера, який відповідає за ведення й синхронізацію журналу змін на всіх вузлах кластера. У журналі міститься серія записів, які вузли повинні виконати у визначеному порядку.

Після обрання лідера починається обслуговування запитів клієнтів. Кожен запит містить команду, яка повинна бути виконана на розподіленому консистентному автоматі. Лідер додає команду до свого журналу як новий запис, а потім паралельно для кожного сервера викликає AppendEntries, щоб реплікувати цей запис. Коли запис успішно репліковано, то лідер застосовує команду з нового запису у своєму консистентному автоматі та повертає результат обчислення клієнту. Якщо деякі вузли відмовляють чи працюють повільно, або мережні пакети для них втрачаються, лідер знову викликає AppendEntries (навіть якщо він уже відповів клієнту), поки всі вузли в кінцевому підсумку не збережуть цей запис у своїх журналах.

Кожен запис журналу зберігає команду консистентного автомата разом з номером епохи, коли запис був отриманий лідером. Номери епох у записах журналу використовуються для виявлення невідповідностей між журналами. Кожен запис журналу також має цілочисельний індекс, який визначає його позицію в журналі.

Лідер вирішує, коли можна виконати наступний запис у журналі у розподіленому консистентному автоматі; ці записи називаються зафіксованими. Raft гарантує, що зафіксовані записи не будуть втрачено та в кінцевому підсумку будуть виконано на всіх доступних консистентних автоматах. Запис журналу фіксується лідером, який створив запис, при цьому це відбувається у той момент, коли запис успішно реплікується на більшості серверів кластера. Ця операція також фіксує всі попередні записи в журналі лідера, включаючи записи, створені

попередніми лідерами. Лідер слідкує за найбільшим індексом зафіксованого запису та включає цей індекс у виклики AppendEntries (включаючи при відправці періодичних контрольних повідомлень), тому інші сервери в кінцевому підсумку отримують актуальне значення індексу останнього запису. Після того, як вузол дізнається, що запис журналу був зафіксований, він застосовує цей запис у своєму локальному розподіленому консистентному автоматі, причому записи обробляються у порядку, який відповідає порядку в журналі.

Raft гарантує наступні властивості:

– якщо два записи у різних журналах мають однаковий індекс і номер епохи, то вони містять одну й ту ж команду;

– якщо два записи у різних журналах мають однаковий індекс і номер епохи, то всі попередні команди в цих журналах ідентичні.

Перше правило випливає з того, що лідер створює не більше одного запису з фіксованим індексом та фіксованим номером епохи, і жоден запис ніколи не змінює своє положення у журналі. Друге правило забезпечується простою перевіркою узгодженості під час виконання процедури AppendEntries. Послідовність дій стек-орієнтованої машини надано на рис. 3.8.

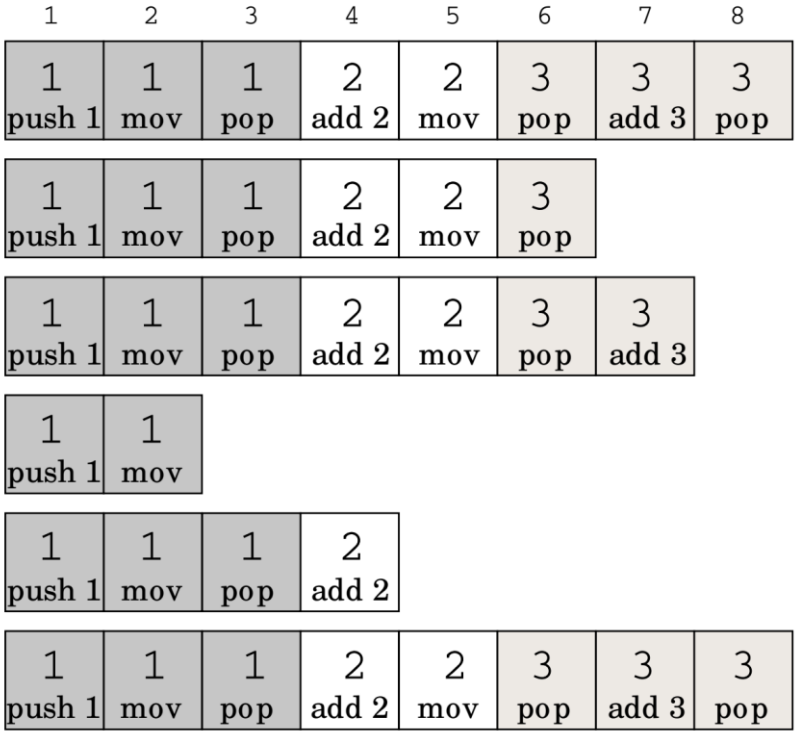

Рисунок 3.8 – Послідовність дій стек-орієнтованої машини

Під час надсилання запису за допомогою виклику AppendEntries, лідер

додає індекс та номер епохи запису, який безпосередньо передує новому запису.

Якщо вузол не знаходить у своєму журналі запис з таким же індексом та епохою, то він відмовляється приймати новий запис. Таким чином, кожного разу, коли AppendEntries успішно завершується, лідер знає, що журнал вузла ідентичний його власному журналу.

Під час нормальної роботи журнали лідера та вузлів залишаються узгодженими, тому перевірка узгодженості в AppendEntries ніколи не закінчується невдало. Однак, у нестандартних ситуаціях, наприклад, у разі відмови лідера, журнал може опинитися в неузгодженому стані (старий лідер міг не встигнути реплікувати всі записи у своєму журналі). Журнал змін лідера в розподіленій системі консенсусу надано на рис. 3.9.

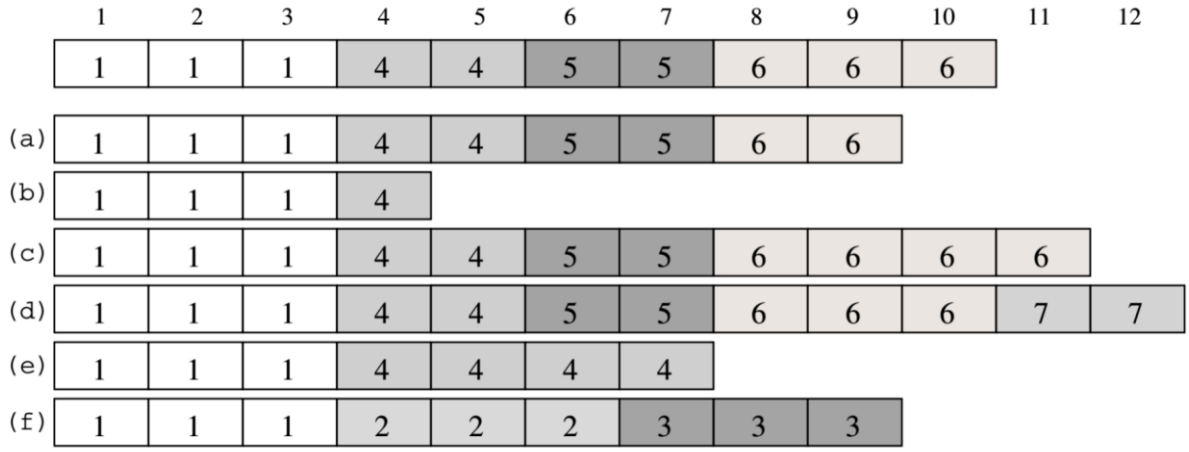

Рисунок 3.9 – Журнал змін лідера в розподіленій системі консенсусу

Невідповідності можуть накопичуватися в результаті серії відмов лідерів і вузлів. Рис. 3.9 відображає ситуації, у яких журнали вузлів можуть відрізнятися від нового лідера. Вузол може не мати записів, які є у лідера (так як лідер не встиг реплікувати всі записи), він може мати додаткові записи, яких немає у лідера (так як старий лідер почав реплікацію запису, але відмовив під час виконання цієї операції), а також можуть бути записи, яких немає ні в одному з серверів. У журналі вузла може бути кілька таких записів.

У алгоримі Raft лідер забезпечує непротирічність, змушуючи журнали вузлів дублювати свій журнал. Це означає, що конфліктуючі записи в журналах вузлів будуть перезаписані елементами з журналу лідера.

Щоб узгодити журнал, лідер повинен знайти останній запис у журналі, де два журнали співпадають, видалити всі записи у журналі вузла після цієї точки та

відправити вузлу всі записи лідера після цієї точки. Усі ці дії відбуваються під час перевірки узгодженості, яка виконується AppendEntries. Лідер зберігає змінну next_index для кожного вузла, вона містить індекс запису журналу лідера, який буде відправлений цьому вузлу наступним. Коли сервер вперше стає лідером, він ініціалізує всі значення next_index номером свого останнього запису в журналі плюс один. Якщо журнал вузла суперечить лідеру, перевірка узгодженості в процедурі AppendEntries відмовить при наступному виклику. Після відмови лідер зменшує next_index і повторює AppendEntries. У підсумку, значення AppendEntries досягне точки, де лідер і вузол мають ідентичні журнали. Коли це станеться, перевірка узгодженості AppendEntries успішно виконається, потім процедура видалить конфліктуючі записи в журналі вузла, а потім додасть записи з журналу лідера (якщо такі існують). Після успішного виконання процедури AppendEntries журнал вузла буде узгоджений з лідером, принаймні до кінця поточної епохи.

З цим механізмом лідеру не потрібно вживати жодних спеціальних дій для відновлення журналу. Йому достатньо почати звичайну роботу, і журнали автоматично будуть узгоджені під час відмови перевірки узгодженості процедури AppendEntries. Лідер ніколи не перезаписує або видаляє записи у своєму журналі.

Цей механізм дозволяє Raft отримувати, реплікувати та фіксувати нові записи, поки більшість серверів працюють.

3.2.4 Тайм-аути та нові терміни

В контексті алгоритму Raft, тайм-аути та терміни є двома фундаментальними концепціями, які сприяють стабільності та надійності реплікації даних в розподілених системах. Тайм-аути використовуються для ініціювання виборів та запобігання конфліктам, тоді як терміни допомагають забезпечити послідовність і визначити часові рамки діяльності кластера.

Тайм-аути в Raft мають критичне значення у процесі виборів лідера. Коли вузол не отримує повідомлень від лідера протягом певного інтервалу часу (тайм-аут), він переходить у стан кандидата та ініціює нові вибори. Цей механізм гарантує, що кластер може швидко реагувати на втрату лідера та забезпечити безперервність управління даними.

Тайм-аути в Raft випадково варіюються між заданими мінімальним та максимальним значеннями, зазвичай в межах від 150 до 300 мс. Ця випадковість

запобігає ситуації, коли декілька вузлів стають кандидатами одночасно, що могло б призвести до «гонок» за статус лідера та подальших конфліктів.

Настроювання тайм-аутів є ключовим для забезпечення стабільності роботи кластера. Занадто короткі тайм-аути можуть спричинити часті та непотрібні вибори лідера, тоді як занадто довгі тайм-аути можуть затримувати відновлення системи після втрати лідера.

Кожен термін в Raft визначається монотонно зростаючим числом, яке інкрементується при кожних нових виборах. Термін служить як хронологічний маркер у життєвому циклі кластера, вказуючи на період активності певного лідера або перехід до нових виборів.

3.3 Опис математичної моделі

Процес консенсусу Raft - це розподілений алгоритм консенсусу, який використовується для підтримання узгодженості між кластером комп'ютерів. Таким чином, Raft розкладає проблему консенсусу на відносно незалежні підпроблеми, які можна описати наступним чином:

– вибори лідера: нового лідера потрібно обирати, коли існуючий лідер зазнає невдачі;

– реплікація журналу: лідер повинен приймати записи журналу від клієнтів і реплікувати їх по кластеру, змушуючи інші журнали погоджуватися з власним.

В контексті моделі черг, процес реплікації журналу процесу консенсусу Raft розглядається як система черги з декількома серверами. Для цих систем консенсусу використовується модель черги M/M/c, яку використано в роботі, де прибуття моделюються як процес Пуасона, а часи обслуговування мають експоненційний розподіл.

У процесі консенсусу Raft сервер відноситься до вузла-послідовника у розподіленій системі, який бере участь у протоколі консенсусу. Кожен сервер підтримує копію журналу системи та спілкується з іншими серверами, щоб переконатися, що вони погоджуються з поточним станом журналу.

Запити клієнтів надходять до кожного вузла-послідовника згідно з процесом Пуасона з інтенсивністю λ, і вимагають часу обслуговування $1/\mu$.

Процес прибуття: як тільки лідер обраний, він починає обслуговувати запити клієнтів. Кожен запит клієнта містить команду, яка має бути виконана

реплікованими становими машинами. Лідер додає команду до свого журналу як новий запис.

Таким чином, процес прибуття можна моделювати як процес Пуасона, з інтенсивністю, визначеною частотою повідомлень лідера, які відправляються послідовникам. λ - це інтенсивність прибуття запитів клієнта від вузла-лідера до вузлів-послідовників.

В роботі проведено опис алгоритму консенсусу Raft за допомогою моделі черги M/M/c.

Модель M/M/c є однією з базових моделей у теорії масового обслуговування, яка використовується для аналізу систем з кількістю обслуговуючих пристроїв (серверів) більше ніж один. Термін M/M/c описує систему, де M відноситься до Markovian (експоненційний розподіл часів між приходами клієнтів та обслуговуванням), а c позначає кількість серверів у системі.

Процес обслуговування такий: лідер додає команду до свого журналу як новий запис, потім видає Remote Procedure Call (RPC)-запити AppendEntries паралельно кожному з інших послідовників для реплікації запису. Коли запис було безпечно репліковано, лідер застосовує запис до своєї станової машини і повертає результат цього виконання клієнту.

Час обслуговування також моделюється як процес Пуасона - експоненційний розподіл з інтенсивністю, визначеною часом, який потрібен послідовникам для реплікації записів журналу лідера. μ - це швидкість обслуговування кожного послідовника (каналу), що представляє швидкість, з якою записи журналу реплікуються від послідовника до лідера.

Прийняття запитів клієнтів до середньої швидкості λ відповідно до моделі Пуассона:

$$\lambda_n = \lambda. \tag{3.1}$$

Існує c паралельних ідентичних послідовників (серверів), кожен з яких працює згідно з експоненційним розподілом з середньою швидкістю μ. Для n клієнтських запитів, швидкість обслуговування сервера може бути отримана в наступних ситуаціях:

– $n\langle c$, де всі клієнтські запити можуть бути обслужені одночасно. Черга

буде відсутня. Кількість простоюючих серверів у цьому випадку становитиме $c - n$. Тоді $\mu n = n\mu$, $n = 0, 1, 2, ..., c$.

– $n \geq c$, де всі сервери зайняті та максимальна кількість клієнтських запитів, що очікують в черзі, становитиме $n - c$. Тоді $\mu_n = c_\mu$ для $n = 0, 1, 2, ...$

Таким чином, швидкість обслуговування μ_n може бути визначена як:

$$\mu_n = \begin{cases} n\mu & n \leq c \\ c\mu & c \leq n \end{cases}. \tag{3.2}$$

Цей аналіз дає розуміння того, як алгоритм Raft працює в різних контекстах і може бути застосований для оптимізації процесів проєктування системи.

Продуктивність Raft може бути підвищена шляхом зміни характеристик системи, таких як, кількість вузлів-послідовників, їх обслуговуюча спроможність, надходження запитів від клієнтів тощо. Результати проаналізовано між компромісами пропускної здатності, затримки та ймовірності затримки. Це може допомогти системним дизайнерам робити практичні вибори щодо розподілу ресурсів та налаштування системи.

Ймовірність затримки повідомлення:

$$P(\lambda, \mu, c) = \frac{\lambda}{c\mu}. \tag{3.3}$$

Очікувана затримка:

$$L(\lambda, \mu, c) = \begin{cases} \frac{1}{\mu}, & \\ \frac{c}{\lambda(c\mu - \lambda)}, & \lambda \langle c\mu \end{cases}. \tag{3.4}$$

У роботі проведено моделювання в середовищі Visual code на Python , щоб продемонструвати зміну основних параметрів для відповідного механізму консенсусу Raft моделі технології блокчейн у контексті системи черги.

Для різних значень μ при $\lambda \in [150, 600]$ досліджено такі параметри, як

кількість послідовників, тобто каналів, очікуваний час запиту клієнта у системі, тобто затримка, ймовірність затримки повідомлення тощо. На рис. 3.10 показано графіки обробки запитів клієнтів для $\mu = 650$.

Показано графік очікуваного часу запиту клієнта, який відноситься до затримки, та ймовірності затримки для $\mu = 650$. Видно, що по мірі збільшення швидкості, очікуваний час запиту клієнта збільшується для $c = 3$.

Для $c = 5$ та $c = 11$ це означає, що при більшій кількості послідовників параметр затримки та затримки повідомлень не залежать від швидкості прибуття.

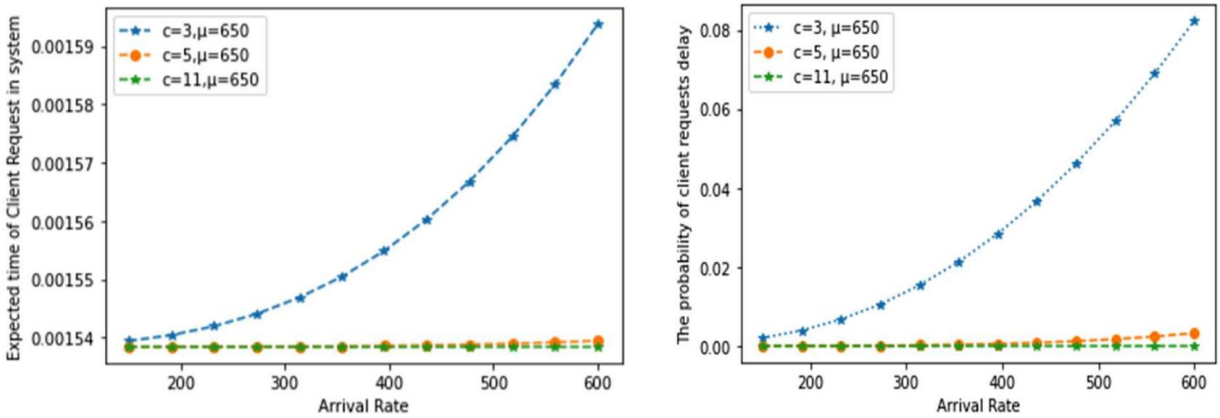

Рисунок 3.10 – Графіки обробки запитів клієнтів

На рис. 3.11 показано графіки обробки запитів клієнтів для інших значень μ.

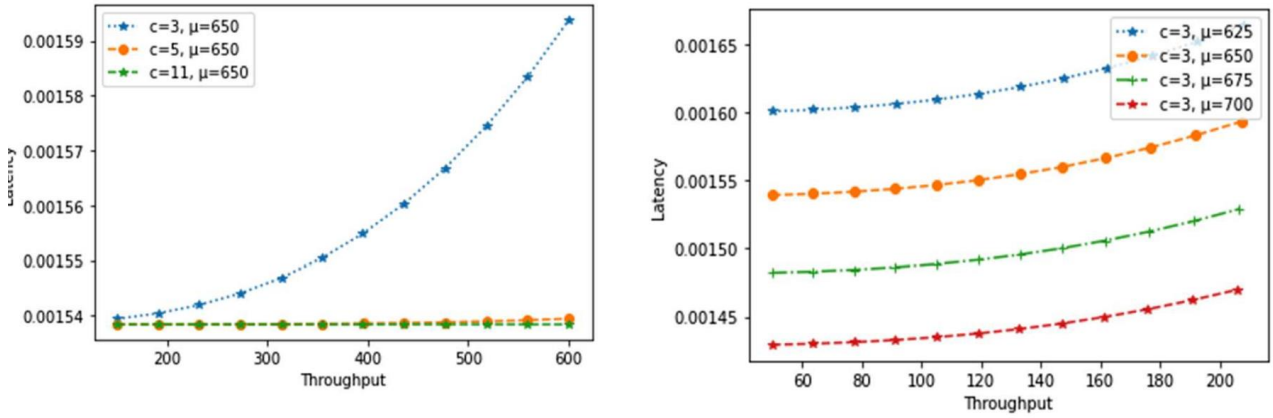

Рисунок 3.11 – Графіки очікуваного часу запиту клієнта

Показано графіки очікуваного часу запиту клієнта, який визначається як затримка, та ймовірності затримки повідомлення для $c = 3$ та різних μ. По мірі збільшення швидкості прибуття очікуваний час запиту клієнта зростає для $c = 3$

та різних μ.

На рис. 3.12 показано графіки аналізу очікуваного часу клієнтських запитів та ймовірності затримок у системі.

Показано графіки очікуваного часу запиту клієнта, який визначається як затримка, та ймовірності затримки повідомлення для комбінацій c та μ. По мірі збільшення швидкості прибуття очікуваний час запиту клієнта зростає для комбінацій c та μ.

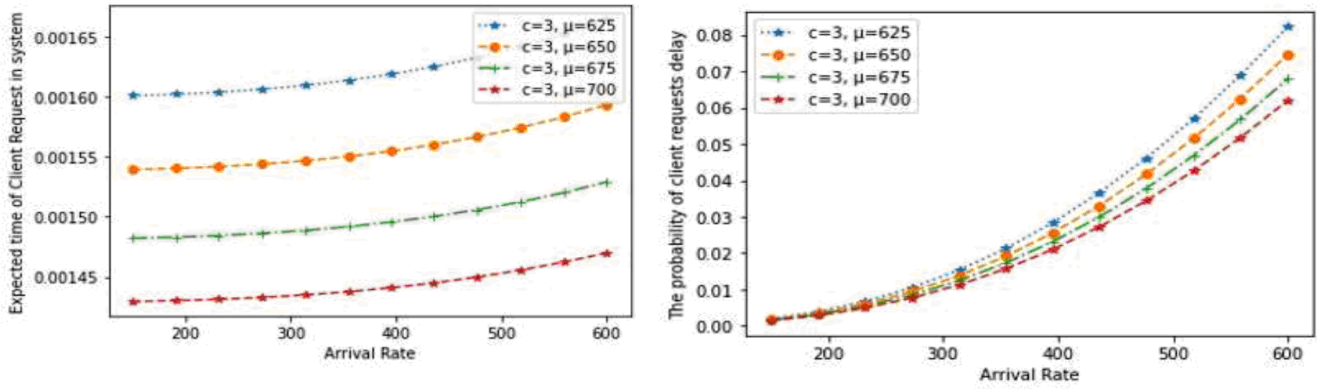

Рисунок 3.12 – Графіки аналізу очікуваного часу клієнтських запитів та ймовірності затримок у системі

Проаналізовано показники продуктивності, як очікуваний час запитів клієнтів у системі, затримка, очікувана кількість запитів клієнтів у системі та черзі, пропускна здатність, ймовірність затримки повідомлення, масштабованість тощо алгоритму консенсусу Raft. Код програми для математичної моделі для проведення експериментів на Python надано в Додатку А.

4 РОЗРОБКА РІШЕННЯ ДЛЯ КЛАСТЕРУ ASTERISK НА ОСНОВІ КОНТЕЙНЕРИЗАЦІЇ DOCKER З АВТОМАТИЧНИМ УПРАВЛІННЯМ KUBERNETES ТА З ВИКОРИСТАННЯМ ПЛАТФОРМИ AWS

4.1 Архітектура та компоненти запропонованого рішення

Архітектура нового рішення базується на принципах мікросервісів та контейнеризації. Asterisk впакований у Docker контейнер, який є основною одиницею розгортання. Kubernetes відповідає за управління цими контейнерами, гарантує їх доступність та відмовостійкість. AWS надає інфраструктуру та додаткові сервіси для оптимізації та моніторингу системи.

Docker - це платформа для розробки, доставки та запуску додатків у контейнерах. У контексті даної роботи Docker використовується для контейнеризації компонентів Asterisk, що забезпечує додатковий рівень ізоляції, гнучкості та масштабованості [18]. Схему роботи Docker показано на рис. 4.1.

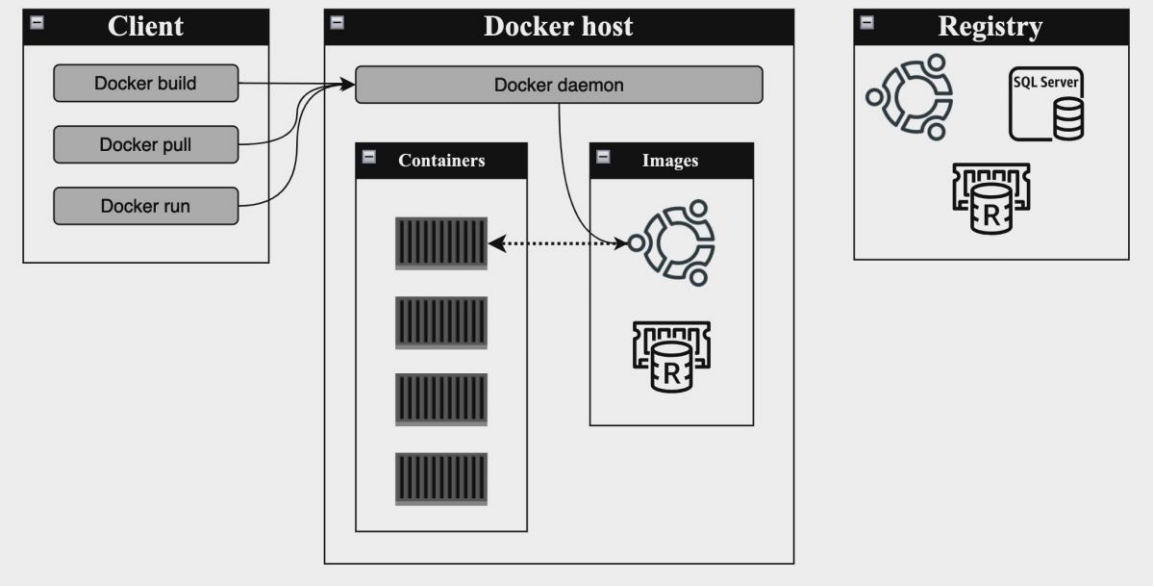

Рисунок 4.1 – Схема роботи Docker

Основні засади такого рішення полягають у його перевагах:

1) Ізоляція. Контейнери Docker надають ізольоване середовище для кожного екземпляра Asterisk. Це означає, що зміни, які відбуваються в одному контейнері, не впливають на інші контейнери. Така ізоляція полегшує управління версіями, тестуванням та розгортанням.

2) Гнучкість. Docker дозволяє легко пакувати, розповсюджувати та

встановлювати додатки разом із їхніми залежностями. Все, що потрібно для запуску Asterisk - це контейнерний образ, який можна швидко розгортати на будь-якому сервері або в хмарі.

3) Масштабованість. Завдяки K8S можна легко масштабувати Asterisk, додаючи або видаляючи контейнери в залежності від потреб. Контейнери можна розподіляти між різними серверами чи вузлами хмари для оптимального розподілу навантаження.

4) Безпека. Docker надає додатковий рівень безпеки завдяки своїм властивостям ізоляції. Він забезпечує контроль над ресурсами, які може використовувати контейнер, та ізолює процеси в контейнері від головної системи.

5) Постійність. З Docker можна використовувати томи для зберігання даних, що забезпечить постійність даних за межами контейнерів. Це особливо корисно для систем Asterisk, де інформація про дзвінки, конфігурації та інші дані має залишатися незмінною незалежно від стану контейнера.

6) Оптимізація ресурсів. Контейнери Docker споживають менше ресурсів, ніж традиційні віртуальні машини, оскільки вони не потребують окремої операційної системи. Це дозволяє оптимізувати використання серверних ресурсів та збільшувати кількість контейнерів на одному фізичному або віртуальному сервері.

7) Мережні можливості. Docker дозволяє створювати ізольовані мережні середовища для кожного контейнера, що полегшує налаштування мережних з'єднань для Asterisk. Це може бути корисно для створення безпечних зон, сегрегації трафіку та інших мережних потреб.

8) Відновлення після збоїв. У разі будь-яких проблем з контейнером або його компонентами, Docker дозволяє швидко відновити роботу. Завдяки зберіганню стану контейнера в образі, можна легко створити новий екземпляр контейнера з того ж самого образу, гарантуючи однакові умови роботи.

9) Інтеграція з Kubernetes та AWS. Docker нативно підтримується Kubernetes, що полегшує автоматизацію, розгортання, масштабування та управління контейнерами на платформі AWS. AWS також надає свої сервіси для роботи з Docker, такі як Amazon Elastic Kubernetes Service (EKS) або Amazon Elastic Container Service (ECS).

Kubernetes - це відкрита система для автоматизації розгортання, масштабування та управління контейнеризованими додатками. У контексті роботи Kubernetes використовується для управління контейнерами Asterisk,

забезпечуючи їх високу доступність та відмовостійкість на платформі AWS.

Однією з ключових особливостей Kubernetes є його здатність автоматично масштабувати кількість контейнерів в залежності від навантаження. Це забезпечує оптимальний розподіл ресурсів і ефективну роботу системи.

Kubernetes має можливість автоматично замінювати контейнери, які не працюють коректно, рестартувати не реагуючі контейнери, а також відсікати контейнери, які не відповідають зазначеному здоров'ю системи.

Kubernetes може розподіляти трафік між контейнерами, що дозволяє рівномірно розподілити навантаження. Також Kubernetes автоматично забезпечує виявлення служб у межах кластера за допомогою імені чи DNS.

Kubernetes дозволяє робити безпечні оновлення без втрати доступності, а також може відкатити зміни в разі виявлення проблем.

Kubernetes дає можливість ефективно та безпечно управляти секретами та конфігураційними даними без перебудови образу контейнера.

Kubernetes може автоматично монтувати системи зберігання даних, необхідні для роботи контейнерів.

Kubernetes підтримує різні типи систем зберігання даних, включаючи локальні диски, cloud providers (наприклад, Amazon EBS, Google Cloud Persistent Disk) та мережні файлові системи.

Kubernetes забезпечує взаємодію між контейнерами всередині одного поду (мінімальна одиниця в Kubernetes) і дозволяє комунікації між різними подами через служби.

Kubernetes дозволяє розділяти ресурси на «namespaces», що ізолює додатки одне від одного. Також Kubernetes підтримує управління доступом на рівні ролей Role Based Access Control (RBAC), що дозволяє гранульно контролювати доступ до ресурсів кластера.

З Kubernetes можна легко інтегрувати системи логування та моніторингу, що забезпечують збір, зберігання та аналіз логів та метрик роботи контейнерів.

Використовуючи платформу AWS разом з Kubernetes, можна користуватися перевагами хмарних сервісів, таких як автоматичне масштабування, управління сховищами даних, мережні рішення, безпека та інші. AWS надає сервіси, які спрощують управління та розгортання Kubernetes кластерів, такі як Amazon EKS [19].

Amazon Web Services є однією з найбільших хмарних платформ, яка надає широкий спектр послуг для розгортання, управління та масштабування

застосунків. Ця платформа може бути особливо корисною при реалізації нового рішення для кластеру Asterisk з метою підвищення доступності та відмовостійкості.

AWS надає інфраструктуру, яка дозволяє розгортати Asterisk у високодоступному та відмовостійкому середовищі. Сервіси такі як Amazon EC2 дозволяють запускати і масштабувати віртуальні сервери, а Amazon EBS забезпечує блокове сховище для даних [20].

Додатково, Elastic Load Balancing (ELB) може бути використано для розподілу вхідного трафіку між декількома екземплярами Asterisk, що дозволяє збільшити продуктивність та відмовостійкість системи.

З точки зору контейнеризації, AWS надає EKS сервіс, який дозволяє легко розгортати, управляти та масштабувати застосунки на основі контейнерів в середовищі Kubernetes. EKS інтегрується з основними сервісами AWS і дозволяє використовувати стандартний інтерфейс Kubernetes для управління робочим навантаженням.

AWS також пропонує сервіс Amazon Elastic Container Registry (ECR), де можна зберігати і управляти Docker-образами. Це забезпечує централізоване сховище для образів контейнерів, яке легко інтегрується з EKS та іншими сервісами AWS.

Що стосується безпеки, то AWS надає набір інструментів для управління ідентифікацією та доступом, шифрування даних та моніторингу ресурсів. Всі ці інструменти можна налаштувати так, щоб забезпечити високий рівень безпеки для Asterisk кластеру.

Використання AWS для розгортання кластеру Asterisk на основі контейнеризації Docker і автоматичного управління Kubernetes дозволяє створити масштабовану, відмовостійку та високодоступну систему без потреби великих капіталовкладень у фізичне обладнання.

4.2 Інтеграція Docker для контейнеризації Asterisk

Інтеграція Docker для контейнеризації Asterisk представляє собою процес створення та використання Docker-контейнера для розгортання системи Asterisk, яка відома своєю здатністю до інтерактивних телефонних дзвінків та відеоконференцій. Цей підхід дозволяє спростити управління Asterisk та його залежностями, забезпечуючи ізоляцію та переносимість серед різних

обчислювальних середовищ.

В рамках цього процесу необхідно виконати кілька ключових кроків. Спочатку необхідно встановити Docker на сервері або комп'ютері, на якому планується запуск Asterisk. Після чого створюється Dockerfile, що описує процедуру створення Docker-образу, включаючи вибір базового образу, встановлення Asterisk та всі необхідні залежності.

Необхідно створити Dockerfile. Dockerfile - це текстовий файл, що містить всі команди, які користувач може викликати в командному рядку для створення образу. У випадку з Asterisk, Dockerfile включає інструкції для вибору базового образу (наприклад, Debian), встановлення необхідних пакетів, завантаження та встановлення Asterisk, а також визначення портів, які будуть відкриті контейнером.

Після створення Dockerfile необхідно виконати команду збірки образу в терміналі. Це робиться за допомогою команди docker build. Ця команда повинна включати тег -t для присвоєння імені образу та шлях до каталогу, де знаходиться Dockerfile. Наприклад: docker build -t my-asterisk. Тут my-asterisk - це ім'я, яке присвоюється образу, та означає поточний каталог, де знаходиться Dockerfile, він продемонстрований в додатку Б.

Після успішного завершення збірки образу, його можна запустити, створивши контейнер. Для цього використовується команда docker run. Ця команда повинна включати параметри для мапінгу портів та ім'я образу. Наприклад: docker run -d --name asterisk-server -p 5060:5060 -p 10000-20000:10000-20000/udp my-asterisk. Тут asterisk-server - це ім'я, яке буде присвоєно контейнеру, -p 5060:5060 -p 10000-20000:10000-20000/udp - це мапінг портів, а my-asterisk - ім'я образу, створеного на попередньому кроці.

Робочий контейнер Docker зі встановленою системою Asterisk, який можна використовувати для подальших досліджень або розробок.

Після створення Docker-образу відбувається створення Docker-контейнера за допомогою команди docker run, де можна налаштувати параметри контейнера, такі як порти, змінні середовища та мережні налаштування.

Внутрішню конфігурацію Asterisk можна налаштувати, зайшовши в контейнер за допомогою команди docker exec або внесенням змін до конфігураційних файлів у Docker-образі перед створенням контейнера.

Управління та масштабування контейнера Asterisk можна здійснювати за допомогою інструментів моніторингу та керування Docker, таких як Docker

Compose або Kubernetes.

Після розгортання слід провести тестування та налаштування системи для забезпечення нормального функціонування. Також варто регулярно створювати резервні копії важливих даних та дотримуватися найкращих практик забезпечення безпеки.

Оновлення Docker-образу Asterisk та його залежностей має відбуватися на регулярній основі для забезпечення актуальності системи.

Встановлення інструментів моніторингу дозволяє слідкувати за продуктивністю контейнера Asterisk і своєчасно виявляти та вирішувати можливі проблеми. Таким чином, інтеграція Docker для контейнеризації Asterisk дозволяє забезпечити зручність розгортання та масштабування телефонного сервера в обчислювальному середовищі.

4.3 Управління кластером Asterisk через Kubernetes

У роботі досліджено процес інтеграції системи Asterisk з Kubernetes для створення високоефективного кластера, що забезпечить ефективне управління та масштабування інфраструктури телефонії та відеоконференцій. Спочатку проводиться ретельний аналіз вимог до системи та розроблено оптимальну архітектуру кластера, зосереджуючись на стратегіях масштабування та забезпечення високої доступності. Наступним кроком є встановлення Kubernetes на серверах та налаштування його для оркестрації контейнерів з Asterisk. Модель системи для розгортання Docker контейнерів показано на рис. 4.2.

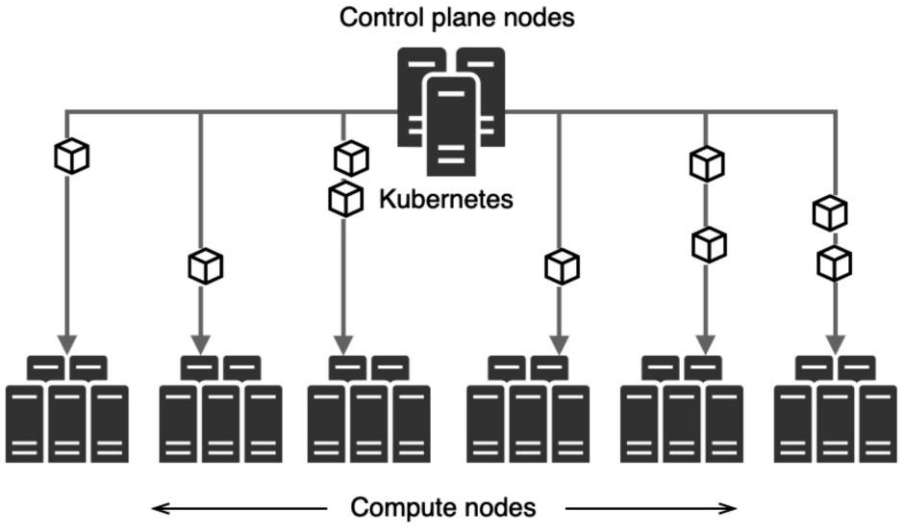

Рисунок 4.2 – Розгортання Docker контейнерів

Паралельно розроблюється спеціальний Docker-образ Asterisk, в якому містяться всі необхідні компоненти та налаштування розробленої системи. Цей образ буде використовуватися для розгортання контейнерів на вузлах Kubernetes (рис. 4.3).

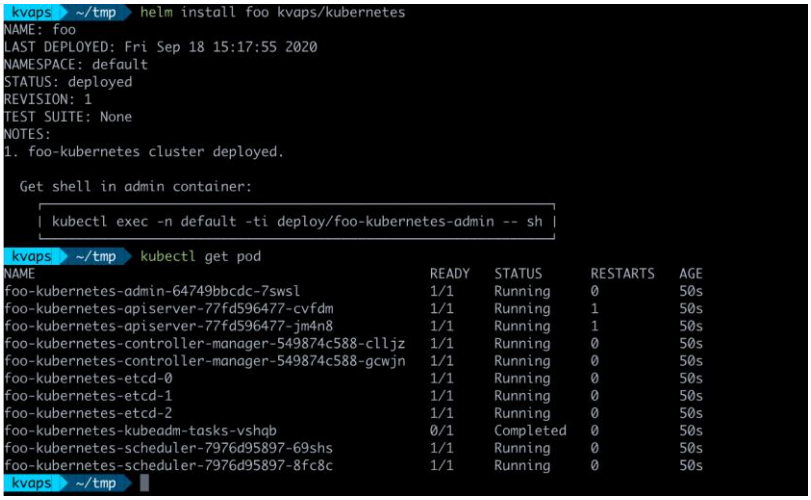

Рисунок 4.3 – Command Line Interface (CLI) налаштування K8s cluster

Після створення образу розроблено конфігураційні файли Kubernetes, описуючи всі необхідні параметри, які визначають, як система буде працювати в розподіленому середовищі. Після цього використовуються Kubernetes-команди для розгортання контейнерів Asterisk на відповідних вузлах.

На останньому етапі особливу увагу приділено питанням безпеки з налаштуванням мережних політик для забезпечечення автентифікації та авторизації. Після встановлення системи проводиться тестування та оцінка її продуктивності та надійності з вирішенням усіх виявлених проблем.

Завершальним етапом є створення документації, що містить конфігураційні файли та інструкції для управління та підтримки системи. В результаті цього процесу створено високоефективний та масштабований кластер, який дозволяє легко управляти та розвивати інфраструктуру IP-телефонії та відеоконференцій, гарантуючи високу якість обслуговування та стабільність роботи.

4.4 Дослідження продуктивності Kubernetes

В роботі проводиться моделювання на кластерах, які складаються з кількох вузлів. Kubernetes розгорнуто для оркестрації контейнерів Docker. Роль майстра Kubernetes полягає зі включенням на усіх майстер-вузлах (конфігурація multi-

master). Інші машини використано як робочі вузли і для DDoS-атак. Робочі вузли надсилають запити у замкнутому циклі, тобто вони чекають відповіді на запит, перш ніж надіслати новий запит, як визначено в [21].

У кластері ініціалізовано протокол реплікації на майстер-вузлах. Потім два робочі вузли транслюють свої запити. Після отримання запиту головні вузли обмінюються повідомленнями для досягнення консенсусу. Для вимірювання продуктивності Kubernetes використано мікротест 0/0, де як запит, так і повідомлення у відповідь порожні [22].

DDoS-атаки використано для моделювання поведінки системи за допомогою команди Hping3 [23,24].

Впроваджено помилки на основі DDoS-атак «Central Processing Unit (CPU) Load» та «Network Flooding» таким чином [25,26]:

– помилка «CPU Load» виникає зі збільшенням кількості користувачів, які постійно надсилають запити до головного вузла;

– помилка «Network Flooding» може бути ініційована одними головними вузлами по відношенню до інших. Припущено, що атакуючі машини одночасно націлені на головний вузол. Таким чином, послідовно і безперервно надсилаються запити розміром 65495 байт у відкритому циклі, тобто запити, що не чекають відповідей. Використано команду: Hping3 -f IP address of targeted master node -d 65495.

Здійснено оцінку продуктивності Kubernetes з використанням алгоритму реплікації BFT-SMaRt, причому дані взяті з [27], та порівняння його з платформою Kubernetes з використанням алгоритму реплікації Raft.

Для дослідження розглянуто два сценарії:

1) Сценарій 1. Розглянуто платформу Kubernetes, в якій спочатку кількість збоїв у кластері нижча за максимальну, що допускається чинним алгоритмом реплікації, причому $f \langle \frac{n-1}{2}$ та $f' \langle \frac{n-1}{3}$, що відповідає використанню Kubernetes з Raft та Kubernetes з BFT-SMaRt алгоритмами реплікації. Потім проводиться DDoS-атака на один головний вузол та оцінка часу консенсусу для кожної платформи.

2) Сценарій 2. На відміну від сценарію 1 початкова кількість збоїв встановлюється рівним максимальному, яке може бути допущено використовуваним алгоритмом реплікації. Потім здійснюються DDoS-атаки на одному майстер-вузлі. У цьому сценарії проводиться оцінка встановленого часу

досягнення консенсусу, а також споживання ресурсів вузлом, що піддається DDoS-атакам (CPU, Random Access Memory (RAM) та доступна пропускна здатність каналу зв'язку). Ресурси вимірюються за допомогою команд IPerf3 для пропускної здатності та top для CPU та RAM [28,29].

Результати дослідження згідно зі сценарієм 1 представлено в таблиці 4.1.

Таблиця 4.1 - Залежність часу досягнення консенсусу від швидкості DDOS-атак для сценарію 1.

Швидкість DDoS-атак, Гбіт/с	Час досягнення консенсусу, мкс для Kubernetes з Raft		Час досягнення консенсусу, мкс для Kubernetes з BFT-SMaRt	
	5 K8s	7 K8s	5 K8s	7 K8s
0	1701,91	2048,25	2746,45	3161,83
2	2004,38	2132,93	2940,87	3179,45
4	2178,72	2471,39	3362,42	4521,79
4,5	2201,37	2501,73	3525,17	4632,38
5	2287,65	2623,87	3612,93	4729,98
5,5	2304,12	2702,99	3867,32	4970,93
6	2331,12	2732,25	4053,53	4970,93

У таблиці показано час досягнення консенсусу в залежності від швидкості DDoS-атак Kubernetes з Raft та Kubernetes з BFT-SMaRt для кластера з 5 та 7 майстер-вузлів відповідно.

Для обох платформ час досягнення консенсусу незначно збільшується пропорційно до рівня DDoS-атак. Справді, навіть наявність додаткових помилок f та f' відповідає максимально допустимому числу збоїв. Отже, робота платформ продовжується без суттєвого погіршення.

Однак, Kubernetes з Raft забезпечує більш короткий час досягнення консенсусу, ніж Kubernetes з BFT-SMaRt. Це очікувано, оскільки алгоритм реплікації Raft розроблено з невеликим обміном консенсусними повідомленнями між головними вузлами, порівняно з BFT-SMaRt. Таким чином, рекомендовано вибирати платформу Kubernetes з Raft, якщо ризик перевищення максимальної кількості помилок, продиктованих Raft, дуже низький.

Сценарій 2 описує рис. 4.4, у якому показано залежність часу досягнення

консенсусу від швидкості DDoS-атак для кластера з 5 майстер-вузлів.

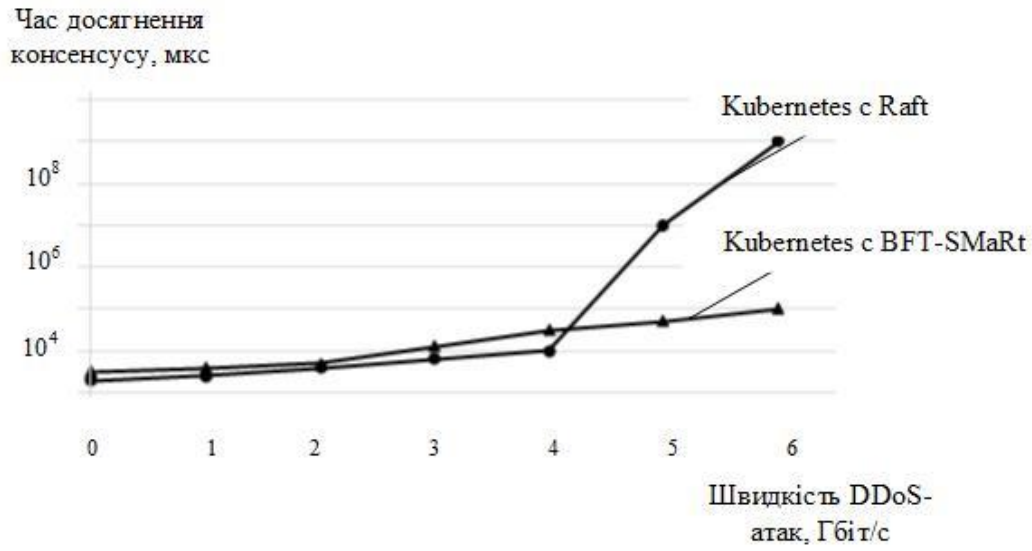

Рисунок 4.4 – Час досягнення консенсусу залежно від швидкості DDOS-атак (сценарій 2, n = 5)

Результати показують, що час досягнення консенсусу збільшується зі збільшенням швидкості DDoS-атак. Коли швидкість атаки нижче 4,25 Гбіт/с, Kubernetes із Raft показує трохи кращу продуктивність, ніж у Kubernetes із BFT-SMaRt. В цьому випадку вузол, що піддається DDoS-атаці, протистоїть завдяки своїм достатнім ресурсам. Однак при швидкості атаки вище 4,25 Гбіт/с для Kubernetes з Raft консенсус швидко і значно погіршується. В основному це пов'язано з уразливістю протоколу реплікації Raft перед помилками. Дійсно, вузол, що піддається DDoS-атаці, поводитиметься неналежним чином, наприклад, не вчасно відповідатиме на запити інших вузлів. Таким чином, з цього моменту Raft ініціює зміну керівництва кластера, оскільки він уже не здатний досягти консенсусу зі своїм чинним лідером. Цей тригер значно уповільнює досягнення консенсусу на платформі Kubernetes із Raft.

Тим часом, Kubernetes з BFT-SMaRt протистоїть усім DDoS-атакам і здатний досягти часу консенсусу в середньому у 1000 разів краще, ніж Kubernetes з Raft.

На рис. 4.5 показано час досягнення консенсусу в залежності від швидкості DDoS-атак у тому ж середовищі, що і на рис. 4.4, але для кластера з 7 майстер-вузлами.

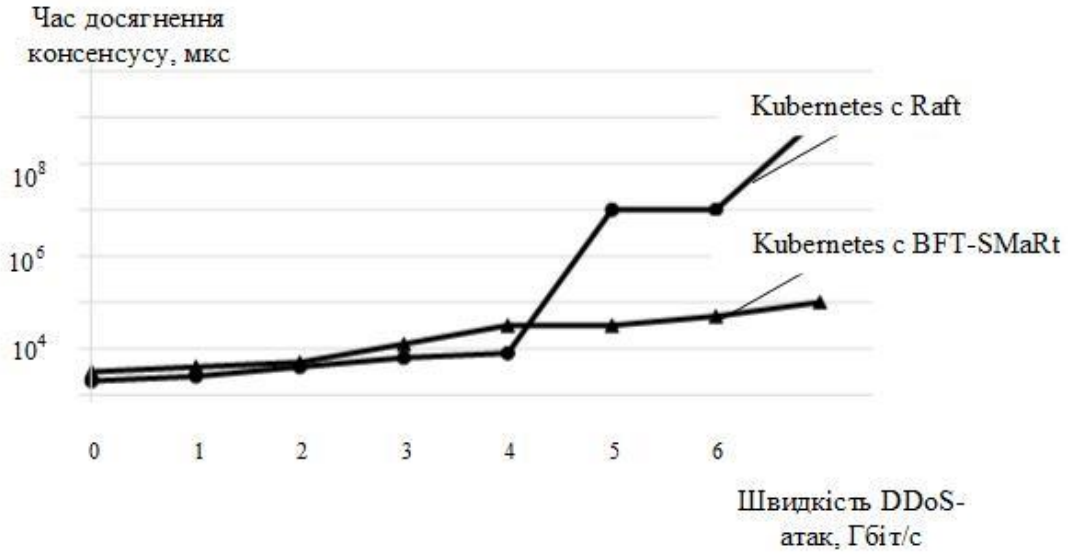

Рисунок 4.5 – Час досягнення консенсусу в залежності від швидкості DDOS-атак (сценарій 2, n = 6)

Помічено, що при n = 5 консенсус встановлюється швидше завдяки меншій кількості повідомлень. У міру збільшення n Kubernetes з Raft стає більш сприйнятливим до DDoS-атак. Справді, швидке зниження продуктивності починається зі швидкості атаки 4,1 Гбіт/с для n = 7 порівняно з 4,25 Гбіт/с для n = 5, зважаючи на те, що Kubernetes з BFT-SMaRt здатний встановити консенсус за розумний час навіть при високих швидкостях атак.

На рис. 4.6–4.8 представлено характеристики CPU, RAM та пропускної здатності вузла, що піддається DDoS-атаці, для сценарію 2 при n = 7.

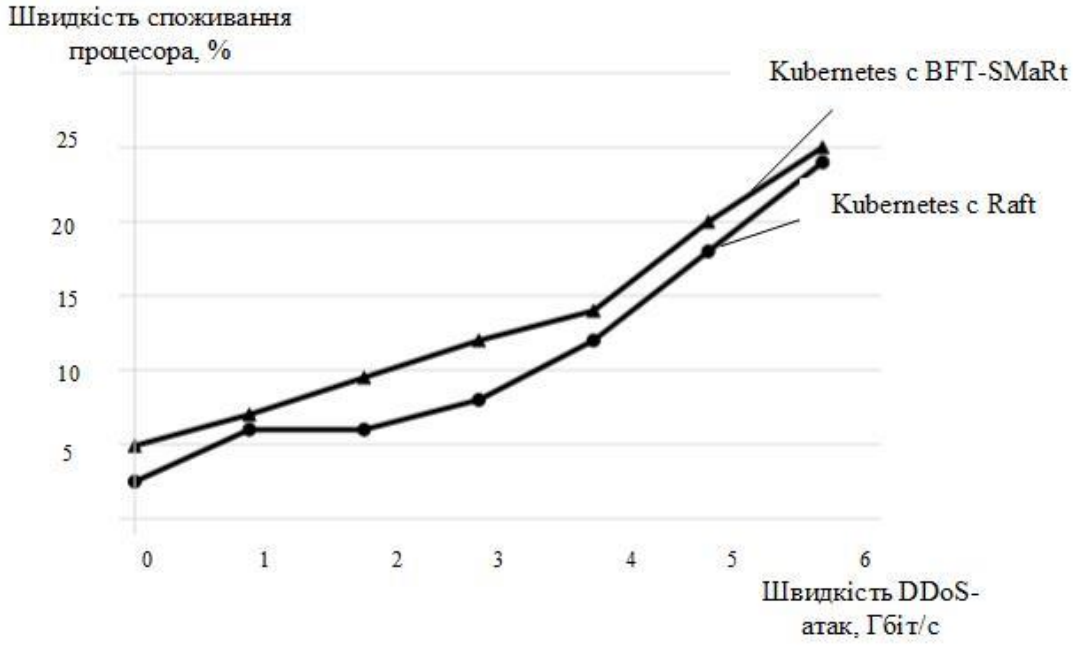

Рисунок 4.6 – Швидкість споживання CPU залежно від швидкості DDoS-атак (сценарій 2, n = 7)

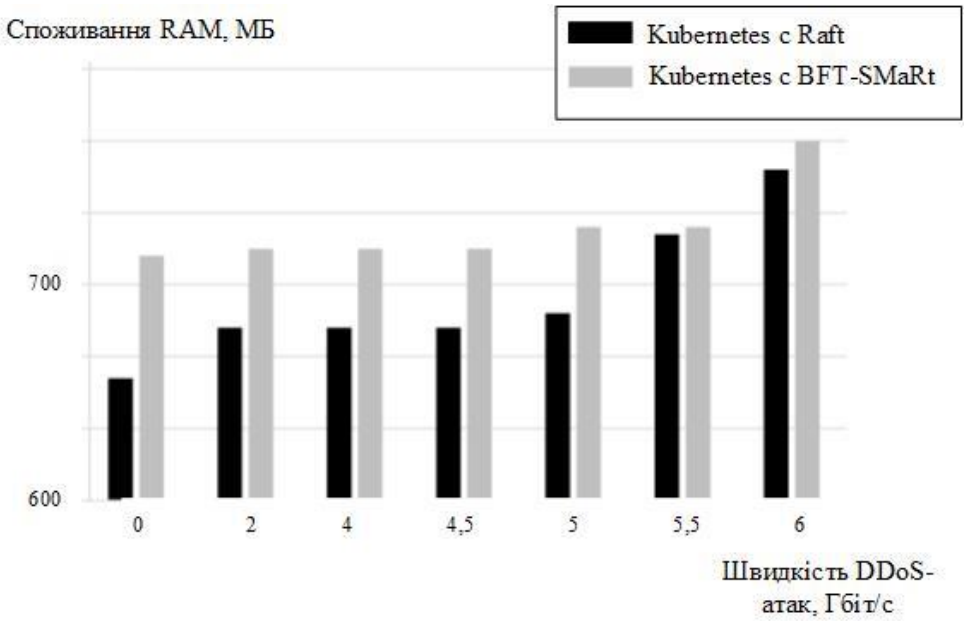

Рисунок 4.7 – Залежність споживання оперативної пам'яті від швидкості DDoS-атак (сценарій 2, n = 7).

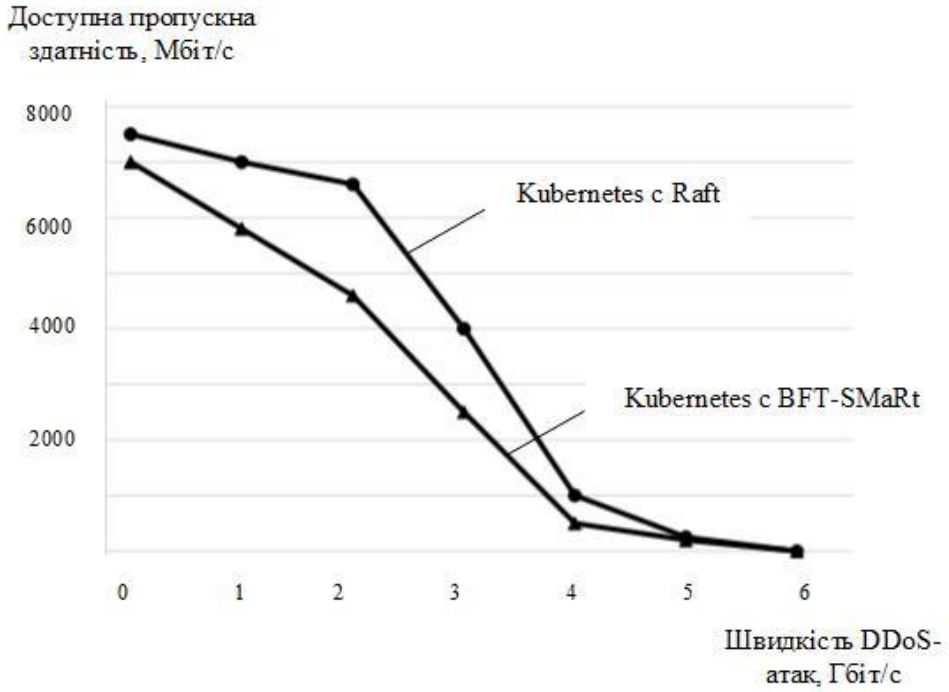

Рисунок 4.9 – Залежність доступної пропускної здатності від швидкості DDoS-атак (сценарій 2, n = 7).

Коли швидкість DDoS-атаки нижче за 4,5 Гбіт/с, то Kubernetes з BFT-SMaRt використовує набагато більше ресурсів, ніж Kubernetes з Raft. Це очікується, оскільки для досягнення консенсусу в Kubernetes з використанням BFT-SMaRt потрібна більша кількість обміну повідомленнями. Однак зі швидкістю атаки вище 4,5 Гбіт/с Kubernetes з BFT-SMaRt та Kubernetes з Raft мають практично однаковий рівень використання ресурсів. Справді, Raft починає вносити зміни до кластера, щоб відновити його стабільність, що призведе до збільшення витрати ресурсів більше, ніж звичайно.

4.5 Використання AWS для розгортання та масштабування рішення

В роботі розглядається використання AWS для розгортання та масштабування інтеграції системи Asterisk з Kubernetes, яку розроблено. Дослідження цієї теми має на меті детально розглянути можливості облікового запису AWS як інфраструктурного рішення для забезпечення високої доступності, масштабованості та безпеки розгорнутої системи (рис. 4.10).

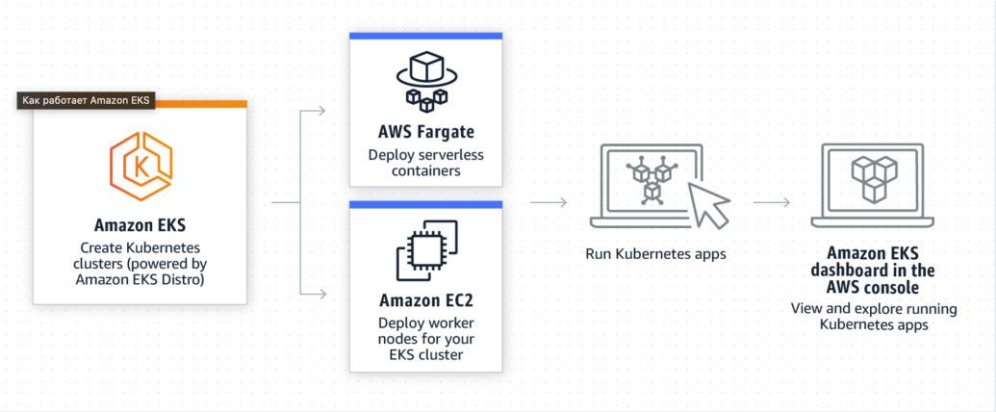

Рисунок 4.10 – Схема налаштування EKS cluster

На першому етапі дослідження проводиться аналіз вимог до системи та визначається, які AWS-сервіси та ресурси будуть використовуватися для ефективного розгортання та керування системою Asterisk. Враховуються такі аспекти, як вибір регіону AWS, налаштування мережі та безпеки, а також вибір типів інстансів для робочих навантажень.

Далі вивчаються можливості Kubernetes на AWS, включаючи використання Kubernetes Engine на AWS або ручне розгортання кластера на інфраструктурі AWS. Розглядаються такі аспекти, як налаштування мережі, завдання для вузлів кластера, та можливості автоматичного масштабування.

Після цього проводиться розробка Docker-образу Asterisk, який відповідає потребам системи, та встановлення Docker на інстансах AWS для подальшого розгортання контейнерів.

При налаштуванні Identity and Access Management (IAM) у AWS виконано наступні кроки:

1) Створення користувача IAM. Перш за все, необхідно створити користувача IAM у AWS Management Console. Для цього в розділі IAM вибрано «Users» у боковому меню. Необхідно натиснути «Add user» та ввести ім'я користувача. Також визначається тип доступу, який буде мати цей користувач, наприклад, програмне забезпечення API або доступ до консолі AWS.

2) Надання прав доступу. Після створення користувача необхідно надати йому відповідні права доступу. Це можна зробити, прикріпивши до користувача вже існуючі політики AWS або створивши власні політики доступу. Політики визначають, які дії користувач може виконувати і з якими ресурсами AWS (рис. 4.11).

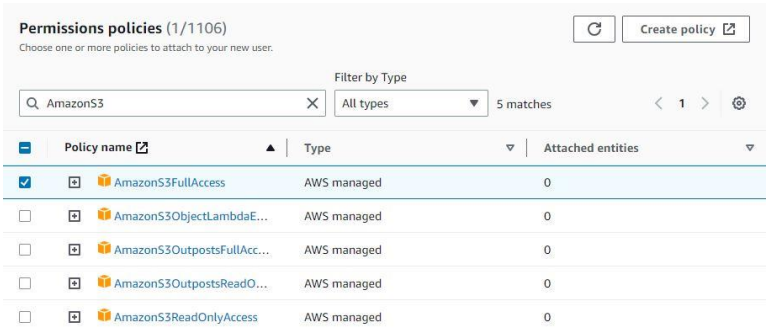

Рисунок 4.11 – Налаштування політик доступу AWS

3) Використання груп для керування користувачами. Для спрощення керування декількома користувачами створено групи IAM й додано користувачів до цих груп. Групи дозволяють надавати однакові права доступу кільком користувачам одночасно (рис. 4.12).

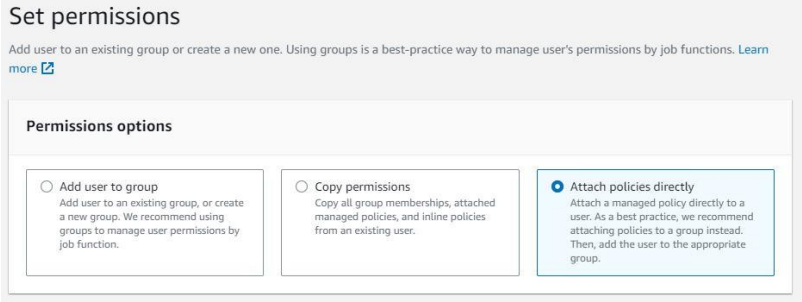

Рисунок 4.12 – Додавання групи доступів в AWS

4) Включення багатофакторної автентифікації Multi-Factor Authentication (MFA). Для збільшення безпеки рекомендується включити багатофакторну автентифікацію для користувачів IAM. Це додає додатковий рівень захисту при вході в систему (рис. 4.13).

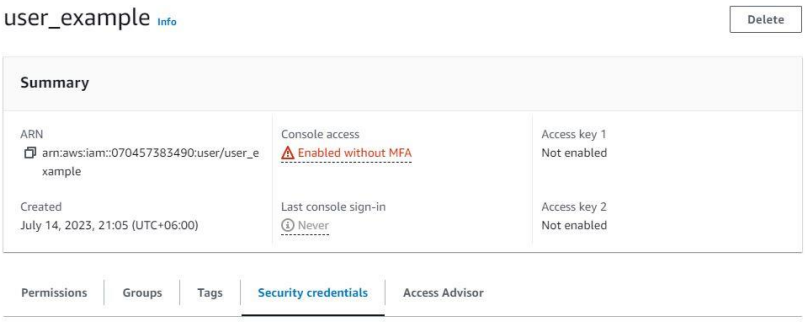

Рисунок 4.13 – Налаштування MFA в AIM

5) Створення ролей IAM для AWS сервісів. Ролі IAM дозволяють давати дозволи необхідні різним сервісам AWS для виконання дій від імені вашого акаунта. Наприклад, роль може дозволити EC2 інстансу доступ до S3 бакета (рис. 4.14).

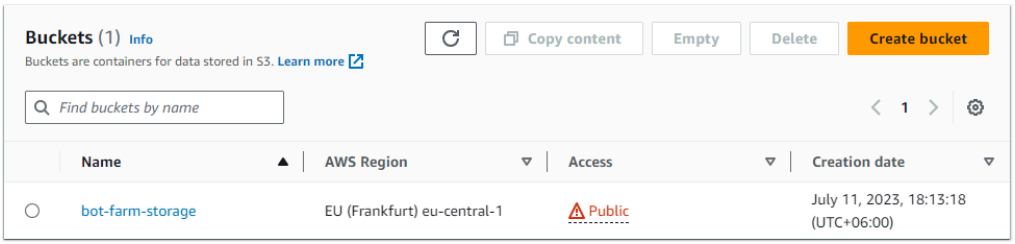

Рисунок 4.14 – Налаштування ролі для S3

6) Перевірка та аудит налаштувань IAM. Необхідність регулярно перевіряти налаштування IAM полягає в тому, щоб перевірити, що політики доступу залишаються актуальними та відповідають найкращим практикам безпеки. AWS надає інструменти, такі як IAM Access Advisor, які допомагають в аналізі використання дозволів.

Для забезпечення безпеки використовується низка сервісів AWS, таких як AWS IAM для керування доступом, а також AWS Security Groups та Network ACLs для обмеження доступу до ресурсів.

Для розгортання Docker образу Asterisk в EKS використано Docker образ Asterisk. Використовуючи Dockerfile для Asterisk, він виконує команду збірки образу: - docker build -t my-asterisk

Після того, як образ Asterisk буде створено, потрібно завантажити його в ECR. Спочатку отримано пароль для входу в ECR та виконано вхід:

- aws ecr get-login-password --region your-region | docker login --username AWS --password-stdin your-account-id.dkr.ecr.your-region.amazonaws.com.

Потім тегуються Docker образ та завантажується в ECR:

- docker tag my-asterisk:latest your-account-id.dkr.ecr.your-region.amazonaws.com/my-asterisk:latest docker push your-account-id.dkr.ecr.your-region.amazonaws.com/my-asterisk:latest.

Далі, коли є Docker образ в ECR, потрібно налаштувати EKS кластер. Це можна зробити через AWS Management Console або за допомогою AWS CLI. Якщо використано AWS CLI, то команда буде виглядати так:

- aws eks create-cluster --name my-eks-cluster --role-arn arn:aws:iam::your-

account-id:role/EKSRole --resources-vpc-config subnetIds=subnet-aster,subnet-abcdef34,securityGroupIds=sg-aster.

Далі потрібно налаштувати kubectl для роботи з EKS кластером. Це робиться шляхом оновлення конфігурації:

- kubectl- aws eks update-kubeconfig --name my-eks-cluster.

Створюється файл asterisk-deployment.yaml для Asterisk образу. Він представленний у додатку В.

Після створення файлу він застосується за допомогою команди:

- kubectl apply -f asterisk-deployment.yaml

Після розгортання системи на AWS проводиться тестування та моніторинг продуктивності, щоб перевірити, як система працює в обраному середовищі. В разі виявлення проблем або оптимізаційних можливостей вони виправляються та документуються.

Завершальною частиною цього рішення є створення документації, яка включає в себе опис всіх кроків та конфігураційних файлів, що були використані під час розгортання та масштабування системи на AWS. Цей підхід дозволяє ефективно використовувати інфраструктуру AWS для підняття високопродуктивного кластера Asterisk зі збереженням високої доступності та безпеки.

ВИСНОВКИ

В кваліфікаційній роботі досліджено використання хмарних технологій для забезпечення гнучкості та масштабованості систем. Розглянуто контейнеризацію Docker, яка дозволяє ізолювати процеси та робити їх відмовостійкими, а також систему управління контейнерами Kubernetes для автоматичного масштабування та відновлення послуг. Показано, що контейнеризація допомагає уникнути конфліктів між залежностями різних додатків, що є критичним для запобігання помилок у системах, які вимагають високої стабільності.

Проаналізовано архітектурні особливості Asterisk у сенсі підходів та рішень до кластерізації з метою підвищення надійності телефонної комунікації. Розглянуто основні технології та методи архітектурної побудови кластерів серверів для забезпечення відмовостійкості та високої доступності.

Наведено типи реплікації даних та обґрунтовано її використання для забезпечення надійності та доступності даних в розподілених системах. Проведено аналіз літератури використання алгоритму консенсусу Raft та наведено основи цього алгоритму. Проведено аналіз математичної моделі процессу консенсусу Raft, використано модель СМО M/M/c.

Проведено моделювання в середовищі Visual code на Python для дослідження зміни основних параметрів для відповідного механізму консенсусу Raft моделі технології блокчейн у контексті системи черги. Отримано графіки очікуваного часу запиту клієнта та ймовірність затримки у системі для різних значень μ та c. Доведено, що по мірі збільшення швидкості прибуття очікуваний час запиту клієнта зростає. Розроблено код програми для математичної моделі для проведення експериментів на Python.

В кваліфікаційній роботі розроблено рішення для кластеру Asterisk на основі контейнеризації Docker з автоматичним управлінням Kubernetes та з використанням платформи AWS. Таке рішення направлено на підвищення доступності та відмовостійкості кластерів серверів. Для цього проведено інтеграцію Docker для контейнеризації Asterisk, налаштувано внутрішню конфігурацію Asterisk, проведено тестування та налаштування системи.

У роботі досліджено процес інтеграції системи Asterisk з Kubernetes для створення високоефективного кластера. Для цього створено модель системи для

розгортання Docker контейнерів, розроблено спеціальний Docker-образ Asterisk, в якому містяться всі необхідні компоненти та налаштування, розроблено конфігураційні файли Kubernetes, налаштовано мережні політики для забезпечечення автентифікації та авторизації.

В роботі проведено дослідження продуктивності Kubernetes. Моделювання проводиться на кластерах, які складаються з кількох вузлів. Для моделювання поведінки системи здійснено DDoS-атаки з використанням команди Hping3. Проведено оцінку продуктивності платформи Kubernetes з використанням алгоритмів реплікації BFT-SMaRt та Raft, для дослідження розглянуто два сценарії. Проведено оцінку наступних параметрів: часу досягнення консенсусу для кожної платформи, а також споживання ресурсів вузлом, що піддається DDoS-атакам. У якості таких вузлів досліджено CPU, RAM та доступна пропускна здатність каналу зв'язку. Ресурси вимірюються за допомогою команд IPerf3 для пропускної здатності та top для CPU та RAM.

Доведено, що при різній кількості майстер-вузлів для обох платформ час досягнення консенсусу незначно збільшується пропорційно до рівня DDoS-атак, причому час досягнення консенсусу збільшується зі збільшенням швидкості DDoS-атак. Однак, загалом по мірі збільшення вузлів системи Kubernetes з Raft стає більш сприйнятливим до DDoS-атак, хоча Kubernetes з BFT-SMaRt використовує набагато більше ресурсів.

В роботі розглядається використання AWS для розгортання та масштабування інтеграції розробленої системи Asterisk з Kubernetes. Проведено виконання основних етапів цього процессу: розробку Docker-образу Asterisk, встановлення Docker на інстансах AWS для подальшого розгортання контейнерів.

Проведено: налаштування IAM у AWS, створення користувача IAM, надання прав доступу, використання груп для керування користувачами, включення багатофакторної автентифікації MFA, створення ролей IAM для AWS сервісів, перевірка та аудит налаштувань IAM.

У цілому, дослідження підкреслює важливість використання хмарних технологій, контейнеризації та систем управління для створення високонадійних, масштабованих та гнучких рішень. Правильне використання та інтеграція цих технологій може значно підвищити ефективність роботи систем на основі Asterisk, роблячи їх ідеальним вибором для сучасних інфокомунікаційних потреб.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kubernetes Docs. What is Kubernetes? URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes (дата звернення 02.09.2023).

2. Токар Л.О., Колтаков О.А., Цилюрик В.Є. Створення тестового стенду Call-центру для балансування навантаження серверів Asterisk у кластері. *Радіотехніка*. Харків, 2023. № 212. С. 186–196.

3. Kubernetes Docs. Kubernetes Components. URL: https://kubernetes.io/docs/concepts/overview/component (дата звернення 22.09.2023).

4. Kubernetes Docs. Deployments. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment (дата звернення 23.09.2023).

5. Ansible Documentation. Basic Concepts. URL: https://docs.ansible.com/ansible/latest/network/getting_started/basic_concepts.html (дата звернення 30.09.2023).

6. StackOverFlow. Is it possible to run containers on android devices? URL: https://stackoverflow.com/questions/53527277/is-it-possible-to-run-containers-onandroid-devices (дата звернення 10.10.2023).

7. Цилюрик В.Є., науковий керівник Токар Л.О. Хмарна АТС на базі системи віртуалізації. *Радіоелектроніка та молодь у XXI столітті: Матеріали 24-го Міжнародного молодіжного форуму*. Харків, 2020. С. 72–73.

8. Prometheus. What is Prometheus? URL: https://prometheus.io/docs/introduction/overview/#what-is-prometheus (дата звернення 15.10.2023).

9. DevOpsCube. How to Setup Prometheus Monitoring On Kubernetes Cluster. URL: https://devopscube.com/setupprometheus-monitoring-on-kubernetes (дата звернення 15.10.2023).

10. Grafana Dashboards. Node Exporter Full. URL: https://grafana.com/grafana/dashboards/1860 (дата звернення 15.10.2023).

11. Memon R.A., Li J.P., Ahmed J. Simulation Model for Blockchain Systems Using Queuing Theory. *Electronics*. 2019. Vol. 8 (2). P. 1–19.

12. Baliga A., Subhod I., Kamat P. & Chatterjee S. Performance evaluation of the quorum blockchain platform. URL:

file:///C:/Users/RedLine/Downloads/Performance_Evaluation_of_the_Quorum_Blockchain_Pl.pdf (дата звернення 15.11.2023).

13. Li Q. L., Ma J. Y. & Chang Y. X. Blockchain queue theory. *Lecture Notes in Computer Science*. 2018. P. 25–40.

14. Ongaro D. & Ousterhout J. In search of an understandable consensus algorithm. *2014 USENIX Annual Technical Conference*. 2014. P. 305–319.

15. Kim D., Doh I. & Chae K. Improved raft algorithm exploiting federated learning for private blockchain performance enhancement. *2021 IEEE International Conference on Information Networking (ICOIN)*. 2021. P. 828–832.

16. Vora S., Thakkar N., Gor R. A Study of Performance Measures and Throughput of Raft Consensus Algorithm. *International Journal for Research in pplied Science & Engineering Technology (IJRASET)*. 2023. Vol. 45. P. 862–869.

17. Yang G., Lee K., Lee K., Yoo Y., Lee H. & Yoo C. Resource Analysis of Blockchain Consensus Algorithms in Hyperledger Fabric. *IEEE Access*. 2022. Vol. 10. P. 74902–74920.

18. Prometheus Docs Architecture. URL: https://prometheus.io/docs/introduction/overview/#architecture (дата звернення 15.10.2023).

19. Docker Hub. Stress containerized application. URL: https://hub.docker.com/r/progrium/stress (дата звернення 15.10.2023).

20. Смаковський Д.С., Войналович В.А. Horizontal autoscaling of microservices in a cloud-based Kubernetes cluster. *Сучасні проблеми наукового забезпечення енергетики: матеріали XVIII Міжнародної науково-практичної конференції молодих вчених та студентів*, Київ, 2020. С. 28–30.

21. Schroeder B. et al. Open versus closed: A Cautionary tale. *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006)*. 2007. P. 239–252.

22. Castro M., Liskov B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*. 2002. Vol. 20. No. 4. P. 398–461.

23. Sanfilippo, Hping3. URL: http://www.hping.org/hping3.html (дата звернення 18.10.2023).

24. Ops B. Denial-of-service Attack-DOS Using Hping3 with Spoofed IP in Kali Linux. URL: https://www.blackmoreops.com/2015/04/21/denial-of-service-attack-dos-using-hping3-with-spoofed-ip-in-kali-linux/2/ (дата звернення 28.11.2023).

25. Gupta D. Towards Performance and Dependability Benchmarking of Distributed Fault Tolerance Protocols. URL https://theses.hal.science/tel-

01376741/document (дата звернення 28.12.2023).

26. Gupta D., Perronne L., Bouchenak S. BFT-BEnch: A framework to evaluate BFT protocols. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. 2016. P. 109–112.

27. Diouf G. M., Elbiaze H. & Jaafar W. On Byzantine fault tolerance in multi-master Kubernetes clusters. *Future Generation Computer Systems*. 2020. Vol. 109. P. 407–419.

28. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. URL: https://iperf.fr/fr/iperf-doc.php (дата звернення 05.01.2024).

29. Biswas S. A guide to the linux top command. URL: https://www.booleanworld.com/guide-linux-top-command/ (дата звернення 05.01.2024).