ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження технологій балансування корпоративних мереж»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Єник Віталій
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Єник Віталій
група КНДМ-61

Керівник: Сергій Сєрих
науковий ступінь, вчене звання к.т.н., доцент

Рецензент:
науковий ступінь, вчене звання
_____
(Ім'я, ПРІЗВИЩЕ)

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук
Ступінь вищої освіти Магістр
Спеціальність Комп'ютерні науки
Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Єника Віталія Сергійовича
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Дослідження технологій балансування корпоративних мереж

керівник кваліфікаційної роботи Сергій Сєрих к.т.н., доцент,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, паринципи балансування мереж, вимоги до мереж державних підприємств.

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

Дослідження принципів побудови балансованих мереж

Аналіз технологій та методів забезпечення балансу в мережах та можливості їх застосування в мережах підприємств

Розробка моделі сбалансованої мережі

5. Перелік графічного матеріалу: презентація

1. Архітектура SDN мережі
2. Архітектура OpenFlow мережі

6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
	Вибір методів та підходів до дослідження	05.11-12.11.23	
	Дослідження технологій балансу мереж	13.11-19.11.23	
	Аналіз особливостей технологій забезпечення балансу у мережі підприємста	20.11-25.11.23	
	Дослідження фізичних, апаратних, програмних, мережевих методів балансу мережі	27.11-03.12.23	
	Застосування методів та технологій	04.12-10.12.23	
	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти _____ Єник Віталій

(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник

кваліфікаційної роботи _____ Сергій Сєрих

(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 49 стор.,23 таб., 16 рис., 10 джерел.

Наукове завдання – дослідження технологій та методів забезпечення балансування мережі та розробка моделі сбалансованої мережі.

Мета роботи – дослідити та запровадити методи та технології підвищення балансу у мережі.

Об'єкт дослідження – модель збалансованої мережі

Предмет дослідження – технології та методи забезпечення балансу

Короткий зміст роботи: В даній роботі проведено аналіз сучасних технологій та методів забезпечення балансу мереж. Здійснено огляд існуючих засобів сбалансування. Досліджено шляхи побудови сбалансованої мережі, зокрема звернуто увагу на інтеграцію.

Далі у роботі розглянуто конкретний приклади балансування трафіку в мережі.

Завершально в роботі розроблено модель сбалансованої мережі, яка враховує методи балансування нового зразку. Запропонована модель сприяє створенню ефективного механізму балансування мережевої інфраструктуру.

КЛЮЧОВІ СЛОВА: БАЛАНСУВАННЯ, ТЕХНОЛОГІЇ БАЛАСУВАННЯ ТРАФІКУ, МОДЕЛЬ СБАЛАНСОВАНОЇ МЕРЕЖІ

ABSTRACT

The text part of the qualification work for the master's degree: 49 pages, 23 tables, 16 figures, 10 sources.

Scientific task - research of technologies and methods of network balancing and development of a balanced network model.

Purpose - to study and implement methods and technologies for improving the balance in the network.

Object of study - a model of a balanced network

Subject of research - technologies and methods for ensuring balance

Summary of the work: This paper analyzes modern technologies and methods for ensuring network balance. An overview of the existing balancing tools is made. The ways to build a balanced network are investigated, with a focus on integration.

Next, the paper considers a specific example of traffic balancing in a network.

Finally, the paper develops a model of a balanced network that takes into account the new balancing methods. The proposed model contributes to the creation of an effective mechanism for balancing the network infrastructure.

KEYWORDS: BALANCING, TRAFFIC BALANCING TECHNOLOGIES, BALANCED NETWORK MODEL

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

DP (Data Plane) Рівень даних

CP (Control Plane) Рівень керування

SDN (Software Defined Network) Програмно визначена мережа

SDMN (Software Defined Mobile Network) Програмно визначена моб. мережа

QoS (Quality of service) Якість обслуговування

CapEx (Capital Expenditure) Капітальні видатки

OpEx (Operation Expenditure) Операційні витрати

NOS (Network operating system) Мережева операційна система

OF (OpenFlow) Протокол зв’язку

TCP (Transmission Control Protocol) Протокол передачі даних

UDP (User Datagram Protocol) Протокол передачі даних

IP (Internet Protocol) Міжмережевий протокол

ARP (Address Resolution Protocol) Протокол визначення адрес

ВСТУП

Балансування корпоративних мереж – це важливий стратегічний підхід, спрямований на досягнення оптимального функціонування та ефективного управління мережевою інфраструктурою підприємства. Цей підхід визначається потребою в забезпеченні високої надійності, продуктивності та загальної ефективності, враховуючи зростання обсягів трафіку та структурну складність корпоративних мереж. Для досягнення цих стратегічних цілей використовуються різноманітні технології та методи балансування.

Балансування корпоративних мереж являється стратегічним інструментом для підприємств, що прагнуть підтримувати високий рівень ефективності та надійності своєї мережевої інфраструктури. Ці технології допомагають оптимізувати використання ресурсів, забезпечуючи стійкість до викликів сучасного бізнес-середовища та підтримуючи стабільність у навантаженому мережевому середовищі.

Важливо розвивати мережі, щоб відповідати майбутньому попиту на велику пропускну здатність, розширений спектр послуг з новими конкретними вимогами та оптимальне використання ресурсів. Збільшення обсягу користувачів і послуг призводить до зростання потреб у пропускній здатності мереж. Забезпечення інфраструктури, що відповідає цьому очікуваному зростанню трафіку, є необхідною передумовою для майбутнього розвитку мереж.

Мета даного дослідження полягає в розгляді способів збалансування в програмно-визначених мережах. У рамках цієї роботи реалізовано механізм балансування навантаження за допомогою програмного забезпечення. Новий підхід SDN вирішує проблеми, пов'язані з вартістю, надаючи гнучкість конфігурації, скорочуючи час розгортання, автоматизуючи процеси та полегшуючи побудову мережі, при цьому не вимагаючи глибоких знань про програмне забезпечення або обладнання конкретних виробників.

1 МЕТОДИ БАЛАНСУВАННЯ КОРПОРАТИВНИЙ МЕРЕЖ

Серверне Балансування забезпечення рівномірного розподілу трафіку між серверами для максимального використання ресурсів та попередження перевантажень. Серверне балансування є ключовою стратегією в інфраструктурі обчислювальних систем, спрямованою на оптимізацію використання ресурсів та підвищення високої доступності сервісів. Ця технологія використовується для розподілу трафіку між різними серверами у мережі, забезпечуючи ефективне та безперебійне функціонування систем. Серверне балансування стає невід'ємною частиною інфраструктури високопродуктивних систем, забезпечуючи ефективність, надійність та швидкість обслуговування користувачів.

Мережеве Балансування створене для оптимізація пропускної здатності та надійності шляхів між мережевими пристроями. Мережеве балансування є важливою стратегією для великих та розподілених мереж, забезпечуючи необхідну стабільність, ефективність та надійність у обміні даними. Це балансування є стратегічним підходом до оптимізації роботи мережевих ресурсів, спрямованим на рівномірний розподіл трафіку та забезпечення високої доступності та продуктивності. Ця технологія використовується для ефективного використання доступних мережевих шляхів і покращення загальної ефективності системи.

DNS Балансування проходить через використання DNS для ефективного розподілу запитів між різними серверами, оптимізуючи процес обробки. DNS балансування — це стратегічний підхід до оптимізації доступу до ресурсів мережі шляхом розподілу трафіку між різними серверами за допомогою системи доменних імен. Ця технологія дозволяє покращити швидкість завантаження веб-сайтів, забезпечити високу доступність та вирішити проблеми перевантаження серверів.

Балансування за допомогою SDN Застосування принципів цього балансування для програмного управління трафіком та оптимізації маршрутизації на рівні центрального контролера. Програмно-визначені мережі представляють іноваційний підхід до структуризації мереж, який стрімко отримує популярність. Цей метод ґрунтується на розділенні площини передачі даних та управління, що

дає операторам мереж значні переваги у вигляді централізованого програмного управління. Ця централізована модель управління та контролю дозволяє операторам отримувати загальне уявлення про мережу, уникнувши керування тисячами рядків конфігурації, розсіяних між численними мережевими пристроями. Основна мета програмно-визначених мереж полягає в забезпеченні високого рівня гнучкості для ефективного адаптування стану та поведінки мережі.

Географічне балансування навантаження враховує місцезнаходження користувачів для розподілу їхніх запитів на найближчі сервери, забезпечуючи оптимальний час відповіді. Географічне балансування навантаження є важливою стратегією для оптимізації доступності, надійності та швидкості обслуговування в мережах, де розташування серверів має ключове значення. Ця стратегія спрямована на максимізацію роботи серверів, розподілених у різних географічних регіонах, та забезпечення ефективного використання ресурсів. Географічне балансування навантаження визначається потребами конкретного бізнесу та його користувачів, але взагалі воно допомагає покращити продуктивність та надійність обслуговування за рахунок оптимального розподілу серверів у просторі.

Балансування За Допомогою Апаратних Прискорювачів використовує спеціалізовані апаратні засобіи для ефективного управління трафіком та розподілу його між різними ресурсами. Балансування навантаження за допомогою апаратних прискорювачів є важливою стратегією для оптимізації роботи серверів та мережі в цілому. Ця техніка використовує спеціалізоване обладнання для розподілу трафіку між різними серверами, забезпечуючи ефективну обробку запитів та покращену продуктивність системи. Балансування навантаження за допомогою апаратних прискорювачів є ефективним інструментом для забезпечення оптимальної роботи мережі, зменшення затримок та підвищення загальної ефективності системи.

У сучасному світі бізнесу, де конкуренція надзвичайно жорстка, ефективність та гнучкість корпоративної інфраструктури стають визначальними факторами успіху. В цьому контексті програмно-визначені мережа виступає як ключовий каталізатор для досягнення виняткового рівня оптимізації та інновацій в управлінні мережевим середовищем підприємства.

Програмно-визначені мережі – це передовий підхід до створення та управління мережами, який базується на відокремленні логіки управління від фізичної інфраструктури мережі.

Гнучкість та адаптивність програмно-визначені мережі надає корпораціям можливість швидко адаптуватися до змін в бізнес-середовищі. Централізоване управління дозволяє легко налаштовувати мережеві параметри та впроваджувати нові функції без необхідності ручного втручання в кожному пристрої окремо.

Збільшена продуктивність програмно-визначені мережі спрощує керування ресурсами та розподіл трафіку, що призводить до оптимізації роботи мережі та підвищення продуктивності.

Централізоване керування та моніторинг системи керування робить моніторинг та управління мережею більш ефективним та прозорим.

Зменшення витрат за рахунок автоматизованого керування та оптимізації ресурсів програмно-визначені мережі дозволяє зменшити витрати на обслуговування та розвиток мережі.

Підвищена безпека програмно-визначені мережі дозволяє легше впроваджувати та керувати політиками безпеки, забезпечуючи вищий рівень захисту корпоративних ресурсів.

Результатом впровадження програмно-визначені мережі стає не лише покращення технічних показників мережі, але і створення гнучкої та підтримуючої інфраструктури, яка відповідає потребам сучасного бізнесу. Завдяки SDN корпорації мають змогу впроваджувати інновації швидше, забезпечуючи конкурентні переваги в динамічному світі технологій.

Балансування мереж - це стратегічний підхід до оптимізації та управління мережевою інфраструктурою з метою підвищення надійності, ефективності та продуктивності. Існує кілька методів балансування мереж, і кожен з них має свої переваги та недоліки. Давайте порівняємо різні види балансування мереж та розглянемо, чому SDN може вважатися одним з кращих методів:

– Балансування навантаження на рівні мережі (Network Load Balancing):

Переваги цього метода це розподіл трафіку між серверами для підвищення продуктивності та надійності. Недоліки цього метода це обмежена гнучкість в управлінні мережею; не завжди ефективний для змінного трафіку.

– Балансування навантаження на рівні додатків (Application Load Balancing):

Переваги цього метода це орієнтований на додатки розподіл трафіку; здатний робити розумні рішення для конкретних додатків. Недоліки цього метода це може вимагати додаткового апаратного чи програмного обладнання; обмежена управлінська гнучкість.

– DNS балансування (DNS Load Balancing):

Переваги цього метода це розподіл трафіку на основі DNS-запитів; простота налаштування. Недоліки цього метода це затримка в оновленні DNS-записів; не ефективний для короткочасних змін у трафіку.

– SDN (Software-Defined Networking):

Переваги цього метода це централізоване та гнучке управління мережею; здатність динамічно реагувати на зміни в трафіку; підтримка віртуалізації. Недоліки цього метода це висока складність реалізації в порівнянні з традиційними методами.

Хоча SDN може вимагати великої ініціативи для реалізації, його гнучкість та здатність ефективно працювати в умовах змінного трафіку роблять його привабливим вибором для компаній, які прагнуть оптимізувати роботу своїх мереж.

1.1 Огляд мереж і роль балансування трафіку

Класичні архітектури мереж мають значні обмеження, які необхідно подолати, щоб відповідати сучасним вимогам IT. Сучасна мережа повинна мати здатність масштабування, щоб адаптуватися до зростаючого робочого навантаження з вищою гнучкістю, при цьому зберігаючи мінімальні витрати. Класичний підхід виявляється обмеженим через ряд проблем:

Складність, велика кількість мережевих протоколів та функцій для конкретних випадків використання значно підвищує складність мережі. Функції, як правило, залежать від виробника або виконуються за допомогою власних команд.

Непослідовні політики, налаштування політик безпеки та обслуговування якості (QoS) у сучасних мережах часто потребує ручного втручання або створення сценаріїв для сотень або тисяч мережевих пристроїв. Ця вимога робить внесення змін у політику надзвичайно складним завданням для організацій, які не готові інвестувати значні зусилля у навчання роботі зі скриптовими мовами чи інструментами для автоматизації конфігурацій.

Нездатність до масштабування, при зміні робочих навантажень додатків та зростанні попиту на пропускну здатність мережі, IT-відділ повинен або обмежуватися статичною мережею, або розширювати її разом із зростанням потреб організації. На жаль, більшість традиційних мереж забезпечені статично, що вимагає значного планування та перепроектування мережі при збільшенні кількості кінцевих точок, послуг або пропускної спроможності.

Класичні архітектури мереж недостатньо відповідають вимогам сучасних підприємств, операторів зв'язку та кінцевих користувачів. Завдяки розгалуженим зусиллям у галузі, технологія програмно-визначених мереж (SDN) трансформує мережеву архітектуру.

Система управління мережами програмно-визначені мережі відрізняється від класичних мережевих моделей, таких як традиційні комутатори та маршрутизатори, рядом переваг і інновацій. Гнучкість та централізоване управління централізоване програмне управління дозволяє адміністраторам здійснювати гнучке та централізоване керування всією мережею через контролер.

А ось класична мережа управління розподілене між окремими комутаторами та маршрутизаторами, що ускладнює керування та конфігурацію.

Програмно-визначені мережі являється кращою для реалізації через простоту конфігурації завдяки централізованому контролю, який дозволяє адміністраторам швидко змінювати та адаптувати мережеві політики.

В класична мережа кожен окремий пристрій потребує окремої конфігурації, що може бути складним у великих мережах.

Висока гнучкість трафіку можливість ефективно керувати трафіком за допомогою багатошляхової маршрутизації, балансування та оптимізації шляхів. Класична мережа: Обмежена гнучкість управління трафіком та обробки багатошляхового руху.

Швидка реакція на зміни SDN можливість швидко реагувати на зміни в мережі через централізований контроль та програмне керування. Навідміно від класичної мережі, де зміни вимагають внесення змін на кожному окремому пристрої, що може зайняти багато часу.

Підтримка віртуалізації та облачних рішень в сітях SDN легка інтеграція з віртуальними середовищами та облачними інфраструктурами. Класична мережа: Обмежена підтримка віртуалізації та інтеграції з хмарами.

Висока швидкодія та масштабованість в мережі SDN забезпечує високу швидкість обробки та легку масштабованість завдяки централізованому керуванню. Класична мережа: Може виявити проблеми з швидкістю обробки та масштабованістю великих мереж. Враховуючи ці переваги, SDN стає привабливим вибором для організацій, які шукають ефективне та гнучке управління мережами.

1.2 Програмно визначена мережа

Програмно-визначена мережа (SDN) представляє собою перспективну технологію в області комп'ютерних мереж, що викликає значний інтерес на сьогоднішній день. Зараз SDN розглядається як новий підхід до структуризації комп'ютерних мереж, який надає дослідникам можливість управляти мережевими сервісами шляхом абстрагування функціональності нижчого рівня. Це досягається шляхом розділення системи управління мережею, яка приймає рішення про направлення трафіку (рівень управління), та систем пересилання, які направляють трафік до обраного пункту призначення (рівень передачі даних). Таким чином, мережа стає безпосередньо програмованою і дозволяє абстрагувати

інфраструктуру для програм і мережевих сервісів. Основна ідея SDN полягає в забезпеченні вищої гнучкості та ефективної маршрутизації потоків трафіку.

Відокремлення централізованого управління від основної площини даних стало предметом інтенсивного дослідження в мережевій спільноті, оскільки це істотно спрощує управління мережею та сприяє її розвитку в різноманітних напрямках. Це відкриває можливості для тестування та розгортання нових протоколів та програм в мережі без впливу на існуючий мережевий трафік. Введення нової інфраструктури може відбуватися без особливих труднощів, а проміжні вузли легко інтегруються в програмне управління, відкриваючи можливості для нових рішень у вирішенні проблем, таких як управління складним ядром стільникових мереж.

1.1.1 Архітектура SDN

Підхід SDN відкриває можливість управління мережевими послугами за допомогою абстракції функціональності нижчого рівня. Замість необхідності робити вивчення низькорівневих деталей мережевих пристроїв, таких як управління пакетами та потоками, адміністраторам тепер достатньо працювати з абстракціями, які надає архітектура SDN. Це досягається через відокремлення площини управління від площини даних, що відповідає багаторівневій архітектурі, яка представлена на рис. 1.1.

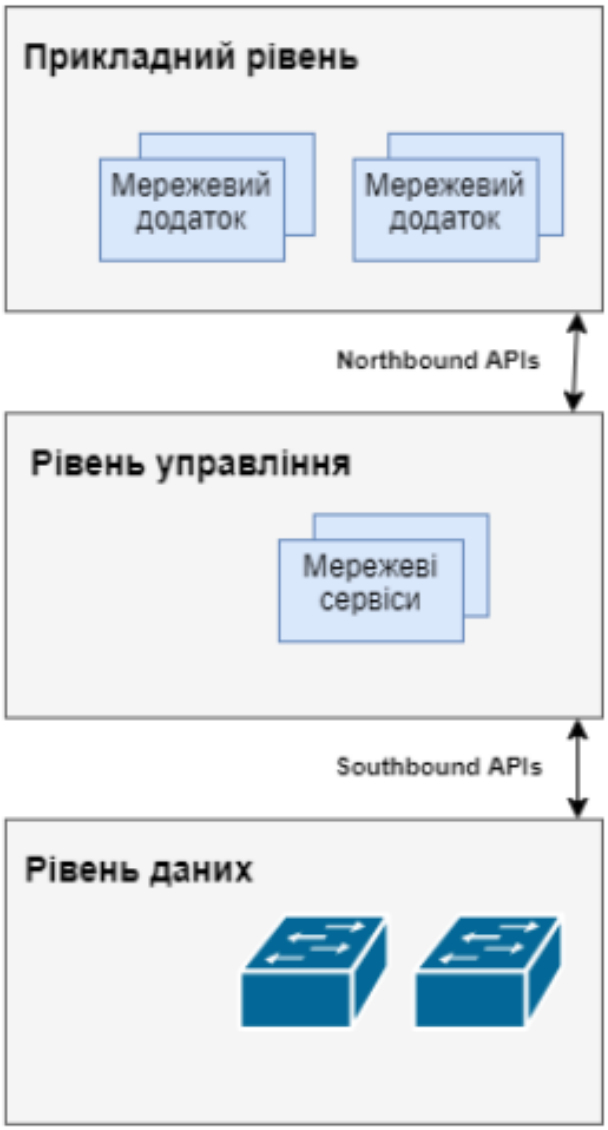


Рисунок 1.1 – Архітектура SDN мережі

З рис. 1.1 можна визначити, що мережа складається з основних компонентів:

Рівень даних: Це можуть бути традиційні апаратні комутатори, які підтримують протоколи типу OpenFlow, або програмні комутатори, такі як OpenVSwitch. Апаратні комутатори гарантують вищу продуктивність, тоді як програмні комутатори забезпечують більшу гнучкість.

Southbound API: Для взаємодії з мережевими пристроями та пересиланням контролеру SDN потрібен інтерфейс. Цей інтерфейс надає контролеру SDN інструкції щодо обробки пакетів, оповіщення про прибуття пакетів, зміни статусу, статистичну інформацію і т.д., зазвичай використовуючи протокол OpenFlow.

Рівень управління: Контролери зазвичай взаємодіють із ключовими службами, такими як служби топології, інвентаризації, статистики та відстеження хостів. Служби топології розглядають зв'язки між пристроями пересилання, а служби інвентаризації відстежують основну інформацію про пристрої SDN. Служби статистики оновлюють лічильники на основі вхідного та вихідного трафіку, а служби відстеження хостів визначають місце знаходження хоста за його IP- та MAC-адресою.

Northbound API: Це представлено через REST API, яке забезпечує взаємодію контролера SDN з програмами та сервісами, що працюють у мережі.

Прикладний рівень: Це набір програм, які можуть працювати на базі SDN, наприклад, програми для організації трафіку, забезпечення безпеки, моніторингу, балансування навантаження тощо.

1.1.2 Функція контролера

Контролер відповідає за координацію та управління ресурсами всієї мережі та за розкриття абстрактного єдиного представлення всіх компонентів для додатків. Ця концепція аналогічна тій, яка використовується в звичайній комп'ютерній системі, де операційна система розташована між апаратним та користувацьким простором і відповідає за управління апаратними ресурсами та надання спільних послуг для програм користувача.

Контролери SDN розрізняються за своєю базовою архітектурою, моделлю програмування та іншими концепціями. При виборі контролера враховують безліч міркувань, таких як вибір мови програмування, що може впливати на продуктивність та час налаштування контролера. Також важливі фактори включають користувацьку базу та підтримку спільноти. При виборі контролера слід враховувати інтерфейси Southbound і Northbound, підтримувані ним, а також те, чи використовується він в освітньому, дослідному або виробничому середовищі.

Існує кілька популярних контролерів SDN, написаних різними мовами та надають широкий спектр послуг. До найвідоміших контролерів входять:

– Floodlight - це ще один відомий контролер для програмно-визначених мереж (SDN). Ось основні характеристики та аспекти Floodlight:

Java-базований контролер: Floodlight реалізований на мові програмування Java. Це надає йому переносимість та широку сумісність з різними платформами.

Підтримка OpenFlow: Як і інші контролери SDN, Floodlight підтримує протокол OpenFlow. Він дозволяє взаємодіяти з комутаторами, використовуючи цей стандартний протокол.

Модульна архітектура: Floodlight має модульну архітектуру, що полегшує розширення та модифікацію функціональності за допомогою розширень та модулів.

REST API: Забезпечує REST API для взаємодії та управління контролером. Це робить можливим зручний доступ та використання контролера з інших додатків чи інструментів.

Активна спільнота: Floodlight також є проектом з відкритим вихідним кодом, і має активну спільноту розробників, що допомагає у підтримці та розвитку проекту.

Розширені можливості обробки подій: Контролер може обробляти різноманітні події в мережі, такі як надходження пакетів, зміни в стані комутаторів та інші.

Підтримка плагінів: Є можливість використання плагінів для розширення функціональності контролера.

Інтеграція з іншими проектами SDN: Floodlight може бути використаний у поєднанні з іншими проектами SDN та іншими компонентами для реалізації конкретних завдань.

Floodlight є популярним вибором серед дослідників, розробників та користувачів для вивчення та розробки програмно-визначених мереж.

– Ryu - це контролер для програмно-визначених мереж (SDN), який створено як проект з відкритим вихідним кодом. Ось деякі ключові характеристики та аспекти контролера Ryu:

Мова програмування Python: Ryu реалізований повністю на мові програмування Python. Це робить його зручним для розробки, тестування та модифікації. Python - це мова високого рівня, що полегшує розробку та швидкість ітерацій.

Підтримка OpenFlow: Як і більшість контролерів SDN, Ryu підтримує протокол OpenFlow, який є стандартом для взаємодії між контролерами та комутаторами в програмно-визначених мережах.

Модульна архітектура: Ryu має модульну архітектуру, що дозволяє розробникам легко додавати нові функції та розширювати функціональність за допомогою модулів.

Розширені функції обробки подій: Ryu надає розширені можливості для обробки подій в мережі, таких як перехоплення пакетів, обробка правил потоку та реагування на зміни в стані мережі.

Інтерфейс REST API: Ryu пропонує REST API для забезпечення зручного взаємодії з іншими компонентами системи чи сторонніми додатками.

Підтримка плагінів: Є можливість використання плагінів для додавання додаткових функціональних можливостей. Це робить Ryu гнучким для різноманітних сценаріїв використання.

Активний розвиток та спільнота: Ryu є проектом з відкритим вихідним кодом, і він має активну спільноту розробників та користувачів. Регулярно виходять нові версії, виправлення помилок та вдосконалення.

Вбудовані додатки: Ryu надає деякі вбудовані додатки, які можна використовувати або модифікувати для реалізації конкретних завдань.

Ryu може використовуватися для вивчення, тестування та розробки програмно-визначених мереж в середовищах відлагодження та експериментів.

– NOX (Network Operating System) - це інша реалізація контролера для програмно-визначених мереж (SDN). Ось деякі ключові аспекти NOX:

Мова програмування C++: NOX написаний на мові програмування C++, що робить його ефективним та швидким у виконанні. Це може бути важливим для великих мереж або завдань, де швидкість обробки даних має значення.

Модульна архітектура: NOX також має модульну архітектуру, яка дозволяє розробникам додавати та видаляти функціональність за необхідності. Модулі можуть взаємодіяти один з одним для реалізації комплексних завдань.

Підтримка OpenFlow: NOX підтримує протокол OpenFlow, що забезпечує стандартизований спосіб взаємодії між контролером та комутаторами в SDN.

Розширення для обробки подій: NOX використовує систему обробки подій, що дозволяє ефективно реагувати на події в мережі та приймати рішення щодо маршрутизації.

Підтримка IPv6: NOX має підтримку IPv6, що важливо для майбутнього розвитку мережевих технологій.

Зручний інтерфейс для розробників: NOX намагається забезпечити зручний інтерфейс для розробників, щоб полегшити створення різноманітних контролерів.

Активний розвиток: Хоча початкова версія NOX була розроблена у рамках дослідницьких проектів, розвиток та підтримка були припинені. Проте, існують спроби створити форки та розширені версії NOX, щоб продовжити його використання.

Важливо відзначити, що NOX не так активно розвивається порівняно з іншими контролерами, такими як OpenDaylight чи ONOS. Розробники можуть вибрати NOX в залежності від конкретних потреб та вимог їхнього проекту.

– POX - це фреймворк для розробки програмно-визначених мереж (SDN). Він написаний на мові програмування Python і надає можливості для створення та взаємодії з контролером SDN. Ось деякі ключові аспекти POX:

Мова програмування Python: POX використовує Python, що робить його доступним та зручним для розробників, які обирають цю мову.

Простота використання: POX спрощує створення контролера SDN та взаємодію з мережевим обладнанням. Він призначений для швидкого розгортання та розробки простих контролерів.

Модульна архітектура: POX має модульну архітектуру, що дозволяє вам легко додавати та видаляти модулі в залежності від конкретних потреб.

Підтримка OpenFlow: POX надає підтримку протоколу OpenFlow, який є широко використовуваним у SDN для взаємодії між контролером та комутаторами.

Широкі можливості: POX може бути використаний для вирішення різноманітних завдань, включаючи власні проекти досліджень, розробку тестових мереж та вивчення концепцій SDN.

Інтерактивний режим: Фреймворк дозволяє вам взаємодіяти з ним у режимі реального часу, що спрощує відлагодження та тестування коду.

Підтримка різних версій OpenFlow: POX може працювати з різними версіями протоколу OpenFlow, що робить його гнучким для використання в різноманітних середовищах.

POX може бути корисним для розробки простих та експериментальних контролерів SDN, де важлива простота та гнучкість. Він є одним із варіантів для швидкого пристосування та експериментування з ідеями, пов'язаними з програмно-визначеними мережами.

– OpenDaylight (ODL) - це відкрите програмне забезпечення для програмно-визначених мереж (SDN). Як контролер SDN, ODL надає фреймворк для розробки та управління мережевими послугами. Ось деякі ключові аспекти OpenDaylight:

Відкритий код: ODL є проектом з відкритим кодом, що розробляється спільнотою розробників. Він доступний на основі ліцензії Eclipse Public License 1.0 (EPL), що робить його доступним для використання та модифікації за різних умов.

Масштабованість: ODL призначений для роботи в масштабі, що дозволяє керувати великими мережами та враховувати потреби сучасних комплексних інфраструктур.

Підтримка різноманітних протоколів: ODL підтримує різноманітні мережеві протоколи, такі як OpenFlow, NETCONF, BGP, SNMP, і інші. Це дозволяє взаємодіяти з різноманітними мережевими пристроями.

Модульна архітектура: ODL має модульну архітектуру, яка дозволяє додавати та видаляти функціональність відповідно до конкретних потреб. Модулі можна розробляти та інтегрувати окремо.

Інтеграція з іншими проектами: ODL працює у співпраці з іншими важливими проектами, такими як OpenStack, для підтримки хмарних інфраструктур та віртуалізації мережі.

Розширені функції управління мережею: ODL дозволяє встановлювати та керувати мережевими політиками, а також реалізовувати різноманітні сервіси, такі як мережева віртуалізація та балансування навантаження.

Спільнота користувачів та розробників: Як відкритий проект, ODL має активну спільноту розробників та користувачів. Це забезпечує підтримку, а також постійний розвиток та вдосконалення.

OpenDaylight використовується в різних областях, включаючи телекомунікації, хмарні обчислення та центри обробки даних для побудови розумних та ефективних мережевих інфраструктур.

– ONOS (Open Network Operating System) - це проект, що входить до складу Linux Foundation і представляє собою операційну систему для великих програмно-визначених мереж (SDN). Ось основні характеристики та аспекти ONOS:

Централізований контролер SDN: ONOS діє в якості централізованого контролера для програмно-визначених мереж, управління якими зосереджено в одному центрі.

Підтримка OpenFlow та інших протоколів: ONOS підтримує протокол OpenFlow, а також інші стандарти і протоколи, що використовуються в програмно-визначених мережах.

Розподілені можливості: ONOS прагне до високої доступності та масштабованості, надаючи розподілені можливості для роботи з великими мережами.

Архітектура мікрослужб: ONOS використовує архітектуру мікрослужб, що полегшує розширення та підтримку нових функцій через використання окремих служб.

Інтеграція з іншими технологіями: ONOS підтримує інтеграцію з іншими технологіями, такими як OpenStack, для підтримки віртуалізації та хмарних рішень.

Гнучкість у використанні: ONOS може використовуватися в різноманітних варіантах, включаючи централізовані та розподілені сценарії, залежно від потреб користувача.

Відкритий вихідний код: Як і інші проекти Linux Foundation, ONOS є відкритим вихідним кодом. Це сприяє розвитку та співпраці в галузі програмно-визначених мереж.

Підтримка для додатків та сервісів: ONOS дозволяє розробникам створювати свої додатки та сервіси для взаємодії з контролером та впровадження різноманітних функцій.

ONOS використовується в різних сферах, включаючи провайдерів послуг, підприємства та дослідницькі установи для управління та оптимізації мережевої інфраструктури в умовах SDN.

– Cisco Application Centric Infrastructure (ACI) - це фреймворк для створення програмно-визначених мереж, розроблений компанією Cisco. Ось деякі ключові аспекти та функції Cisco ACI:

Архітектура SDN: Cisco ACI є рішенням SDN, яке надає централізований та автоматизований підхід до управління мережею.

Policy-Driven Networking: Однією з ключових особливостей ACI є політикозорієнтована модель. Управління мережевим трафіком здійснюється на основі політик, що спрощує конфігурацію та забезпечує послуги на рівні застосунків.

Інтеграція з хмарами: Cisco ACI підтримує інтеграцію з хмаровими сервісами та публічними хмарами, що дозволяє побудувати гібридні інфраструктури та легко переміщати ресурси між приватним та публічним хмарами.

Мультивендорність: ACI підтримує відкриті стандарти та мультивендорність, що дозволяє використовувати різноманітні обладнання в межах інфраструктури.

Автоматизація та Оркестрація: Cisco ACI надає засоби для автоматизації конфігурацій та оркестрації мережевих послуг, що сприяє ефективному управлінню та розгортанню.

Технологія Application Network Profiles (ANP): ACI використовує ANP для опису та управління вимогами до мережі для конкретних застосунків чи послуг.

Візуалізація та моніторинг: Засоби візуалізації та моніторингу дозволяють адміністраторам ефективно слідкувати за станом мережі, трафіком та роботою застосунків.

Інтеграція з іншими Cisco продуктами: ACI може інтегруватися з іншими продуктами та рішеннями Cisco, такими як сервіси безпеки, системи моніторингу та інші.

Cisco ACI широко використовується в центрах обробки даних, корпоративних мережах та хмарових середовищах для підвищення ефективності, гнучкості та безпеки мережевих інфраструктур.

1.1.3 Огляд OpenFlow

Дотримуючись принципу SDN щодо розподілу площин управління та даних, OpenFlow надає стандартизований метод управління трафіком в комутаторах та обміну інформацією між комутаторами та контролером, як показано на рис. 1.2. Комутатор OpenFlow складається із двох логічних компонентів. Перший компонент містить одну або кілька таблиць потоків, що відповідають за зберігання інформації, необхідної комутатору для пересилання пакетів. Другий компонент - це клієнт OpenFlow, який представляє собою простий API, що дозволяє комутатору взаємодіяти з контролером.

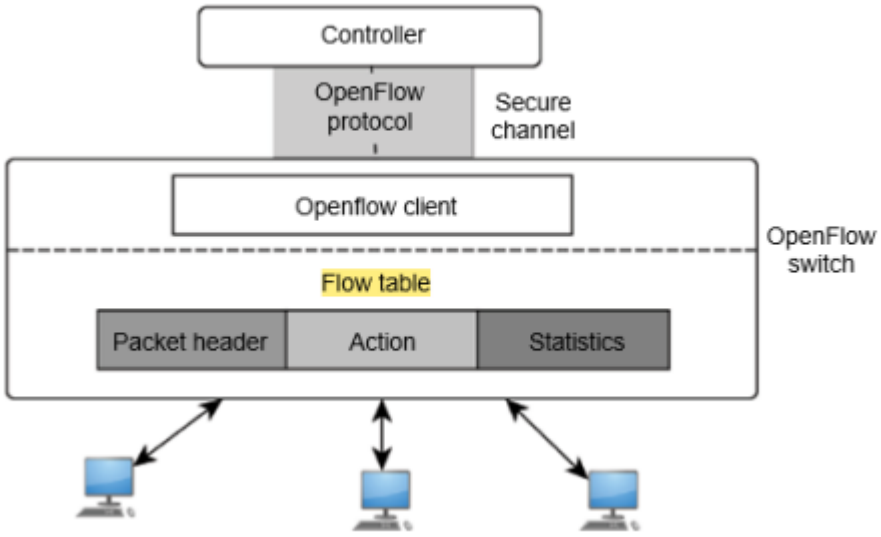


Рисунок 1.2 – Архітектура OpenFlow комутатора

Таблиці потоків містять записи про потоки, кожен з яких визначає набір правил, що керують обробкою пакетів, належать до даного потоку. Кожен запис у таблиці потоків має три поля: заголовок пакета, що ідентифікує потік; дію, яка вказує, як пакет повинен бути оброблений; та статистику, яка відстежує інформацію, таку як кількість пакетів та байтів для кожного потоку та час з останнього пересилання пакета цього потоку.

Коли пакет приходить на комутатор OpenFlow, його заголовок аналізується, і пакет порівнюється з потоком, який має схожий заголовок пакета. Якщо відповідний потік знайдено, виконується дія, визначена в полі action. Дії включають пересилання пакета на певний порт для маршрутизації через мережу, пересилання пакета для обробки контролером або відхилення пакета. Якщо пакет не може бути зіставлений з жодним потоком, він обробляється відповідно до дії, визначеної в записі потоку table-miss.

Обмін інформацією між комутатором і контролером відбувається через надсилання захищених повідомлень за стандартним способом, визначеним протоколом OpenFlow. Таким чином, контролер може маніпулювати потоками, що знаходяться в таблиці потоків комутатора (додавати, оновлювати або видаляти записи про потоки) або реагувати на їх виникнення. Оскільки контролер може

взаємодіяти з комутатором за допомогою протоколу OpenFlow, операторам мережі не потрібно безпосередньо взаємодіяти з комутатором.

Однією з особливостей OpenFlow є можливість використовувати "дикі карти" для полів заголовка пакета, тобто відповідність заголовку пакета не обов'язково повинна бути точною. Цей підхід дозволяє різним мережевим пристроям, таким як маршрутизатори, комутатори та проміжні пристрої, мати схожу поведінку при пересиланні, розрізняючись лише тим, які поля заголовка вони використовують для зіставлення та які дії вони виконують. OpenFlow дозволяє використовувати будь-яку підмножину цих полів заголовка для застосування правил потоків трафіку, що робить його концептуально універсальним для різних типів мережевих пристроїв.

1.1.4 Керування трафіком в програмно визначених мережах

Продовжуючи розгляд проблем управління трафіком в SDN, можна визначити дві основні категорії проблем: гранулярність управління та застосування політик.

1. Гранулярність керування: Гранулярність управління відноситься до того, наскільки дрібнозернистими або крупнозернистими повинні бути операції контролера щодо пакетів, що проходять через мережу. У традиційних мережах кожен пакет розглядається окремо, і відбувається прийняття рішення про маршрутизацію. Однак цей підхід стає нездійсненним для реалізації в SDN при великому обсязі мережі, оскільки всі пакети повинні пройти через контролер, який будує маршрут для кожного окремого пакета.

Часто контролери SDN використовують підхід на основі потоків, де кожен пакет відноситься до певного потоку відповідно до певних властивостей. Контролер встановлює новий потік, вивчаючи перший пакет, що прибув для цього потоку, і налаштовує відповідним чином комутатори. Грубший підхід може бути застосований для забезпечення контролю на основі відповідності агрегованого потоку замість використання окремих потоків.

Основний компроміс полягає в навантаженні на контролер у порівнянні з QoS (Quality of Service), що надається мережевим додаткам. Чим дрібніший

контроль, тим вищий QoS. Однак більш деталізований контроль може вести до менш оптимальних маршрутів, що може погіршити QoS.

2. Дотримання політик: Друга проблема пов'язана з тим, як мережеві політики застосовуються контролером до мережних пристроїв. Є два основних підходи: реактивна та проактивна модель управління.

Реактивна модель управління: Комутаційний пристрій звертається до контролера лише тоді, коли необхідно прийняти рішення про новий потік. Це робить управління мережею більш гнучким, але може викликати зниження продуктивності через час, необхідний для передачі першого пакета потоку на перевірку контролеру.

Проактивна модель управління: Контролер наперед заповнює таблиці потоків для будь-якого трафіку, що може проходити через комутатори, і передає правила на всі комутатори мережі. Це усуває затримку, викликану зверненням до контролера, але може вимагати більше ресурсів.

Кожен підхід має свої переваги та недоліки, і вибір залежить від конкретних вимог та характеристик мережі.

1.3 Представлення мереж

Звичайні мережі також стикаються з викликами в забезпеченні високоякісних послуг своїм користувачам. Різноманітні обмеження можна класифікувати за кількома ключовими аспектами:

Обмеження масштабованості: Зі зростанням обсягу мережевого трафіку, особливо внаслідок нових вимог, таких як передача великого обсягу даних, стає важкою задачею забезпечити достатню пропускну здатність та ефективність мережі. Статичні мережі із зайвою пропускною здатністю стають менш гнучкими та вартісними для масштабування.

Складне управління мережею: Вирішення проблем управління мережею вимагає значного досвіду та ресурсів. Багато пристроїв транзитного зв'язку не мають єдиної інтерфейсної платформи для зручного управління, що ускладнює такі завдання, як конфігурація та впровадження політик.

Ручна конфігурація мережі: Системи управління мережею часто вимагають великої кількості ручної праці. Забезпечення навіть базового рівня безпеки вимагає кваліфікованих операторів. Ручна конфігурація, крім того, піддається помилкам, а їх виправлення коштує часу та ресурсів.

Складні та дорогі мережеві пристрої: Деякі пристрої транзитного зв'язку повинні виконувати великий обсяг роботи, що охоплює моніторинг трафіку, білінг, управління якістю обслуговування, контроль доступу та інші функції. Це робить їх складними та вартісними для підтримки.

Вищі витрати: Оператори не можуть легко поєднувати пристрої різних виробників, що безпосередньо підвищує капітальні витрати. З іншого боку, ручна конфігурація та обмеженість вибору обладнання збільшують оперативні витрати.

Негнучкість: Стандартизація для впровадження нових послуг у звичайних мережах може бути тривалим процесом, вимагаючи багато часу на розгортання та активацію нових сервісів. Це обмеження стає особливо критичним у контексті зростання трафіку та потреб у нових сервісах.

1.4 Мережеві SDN

Адаптація концепцій SDN та віртуалізації до звичайних мереж є ключовим кроком у вирішенні зазначених проблем. SDN не лише ефективно подолає ці труднощі, але й принесе значний приріст у гнучкості, масштабованості та продуктивності телекомунікаційних мереж. Хоча початково концепція SDN була спрямована на фіксовані мережі, адаптація до звичайних мереж враховує їхні унікальні вимоги, такі як ефективне управління мобільністю, вищий рівень обслуговування якості (QoS), активне використання тунелювання в пакетній передачі та інші аспекти.

Концепція SDMN, що визначається як Software-Defined Mobile Networking, пропонує розширення парадигми SDN, адаптоване до специфічностей та функціональних особливостей звичайних мереж. Це включає в себе здатність ефективно керувати мобільністю, забезпечувати вищий рівень обслуговування та активно використовувати мережеві тунелі для оптимізації пакетної передачі.

Крім того, SDMN володіє вищим рівнем обізнаності щодо послуг та забезпечує ефективне використання мережевих ресурсів порівняно із вихідними концепціями SDN. Такий підхід враховує усі аспекти та вимоги звичайних мереж, сприяючи їхньому оптимальному функціонуванню.

Архітектура SDMN в наш час спрямовує традиційну мережу до моделі, орієнтованої на потоки, що використовує доступне обладнання та логічно централізований контролер. SDMN представляє собою підхід до створення мереж, де управління відокремлюється від специфічного для телекомунікаційного обладнання і передається програмному додатку, контролеру. Комутатори та роутери з підтримкою SDN керуються через контролер SDN. Компоненти звичайної мережі можуть бути розгорнуті як віртуальні елементи у хмарі оператора. Крім того, для програмування та підвищення продуктивності програмно визначених мережових комутаторів, SDMN можна використовувати із застосуванням сучасних методик гнучкого програмування. Ці програмні методології можна розробляти, вдосконалювати та модернізувати у значно коротший термін, ніж розробка сучасних пристроїв для традиційних мереж зв'язку. У цьому підході кожен оператор отримує можливість розробляти власні мережеві концепції для задоволення конкретних потреб своїх абонентів і оптимізувати свою мережу для досягнення кращої продуктивності. Останніми роками багато дослідників працює над впровадженням SDMN.

Сутність Software-Defined Mobile Networking (SDMN) полягає в технологічному рішенні, яке розділяє керуючі функції (Control Plane - CP) та функції обробки даних (Data Plane - DP) в мобільних мережах. Це забезпечує централізоване керування мережею та дозволяє впровадження інноваційних технологій для оптимізації роботи мережі та поліпшення якості обслуговування для абонентів.

Рівень Data Plane (рівень інфраструктури) включає в себе різноманітні мережеві елементи, такі як комутатори та інші пристрої, що здійснюють функції комутації пакетів та пересилання даних. В цьому рівні базові станції

підключаються до комутаторів на кордоні, що забезпечує ефективну обробку трафіку в мобільних мережах.

Особливо важливою рисою архітектури SDMN є те, що вона залишається прозорою для існуючих радіотехнологій. Це означає, що можливість використання різних технологій і стандартів мобільного зв'язку не обмежується архітектурою SDMN. Прикордонні комутатори в опорній мережі взаємодіють з Інтернетом для оптимізації трафіку для забезпечення оптимального функціонування звичайних абонентів, що підвищує якість обслуговування та забезпечує ефективну передачу даних через мережу.

Таким чином, SDMN впроваджує інноваційний підхід до управління та оптимізації мобільних мереж, забезпечуючи централізоване керування та покращення роботи мережі для забезпечення найкращого досвіду для кінцевих користувачів.

Контролер мережі є логічно централізованим елементом, який здійснює управління функціональністю комутаторів рівня DP (Data Plane). Використовуючи протокол управління, такий як OpenFlow, контролер встановлює правила потоку в кожному комутаторі рівня DP для оптимальної маршрутизації трафіку в межах мережі.

Важливим аспектом є те, що інтерфейс прикладного програмування Southbound API визначає кордон між мережним контролером та рівнем DP. Це забезпечує ефективний обмін інформацією між контролером та комутаторами, що дозволяє здійснювати централізоване управління мережею та визначати оптимальні стратегії маршрутизації трафіку. Такий підхід робить контролер ключовим елементом для забезпечення ефективної та гнучкої роботи програмно-визначених мереж.

Прикладний рівень програмно-визначених мереж включає керуючі та бізнес-додатки, які раніше були частиною звичайної мережі. До цих додатків входять елементи, такі як Home Subscriber Server, Authentication, Authorization, and Accounting (AAA), Mobility Management Entity та Policy and Charging Rules Function

(PCRF). Ці елементи тепер функціонують як програми, які працюють поверх мережевої операційної системи.

Кордон між прикладним рівнем та мережевим контролером знаходиться на рівні Northbound API, що дозволяє забезпечувати взаємодію та обмін інформацією між цими компонентами. Елементи управління виконують традиційні функції, спрямовані на управління мобільністю, розподілом ресурсів та керуванням трафіком. Такий підхід дозволяє зберігати та оптимізувати функціональність звичайних мереж на прикладному рівні програмно-визначених мереж.

2 АЛГОРИТМ БАГАТОШЛЯХОВОЇ МАРШУТИЗАЦІЇ З ВИКОРИСТАННЯМ РАНІШЕ СФОРМОВАНИХ ШЛЯХІВ

2.1 Балансування трафіку

Балансування трафіку - це комплекс методів для розподілу робочого навантаження між різними мережами або компонентами мережі, такими як канали зв'язку, обчислювальні пристрої, пристрої зберігання даних та користувачі. Основна мета - досягнення оптимального використання ресурсів, максимізації пропускної спроможності та мінімізації часу відгуку. Цей підхід також спрямований на уникнення перевантажень та забезпечення високої якості обслуговування. У випадку, коли є кілька ресурсів для виконання конкретної функціональності, балансування навантаження може служити інструментом для максимізації ефективності мережі.

Існує два типи алгоритмів балансування трафіку:

Статичний алгоритм: Цей метод підходить для систем з низькою варіативністю навантаження та передбачає наявність попередньої інформації про ресурси системи.

Динамічний алгоритм: У цьому випадку шукається найлегший комутатор у всій системі, і він отримує перевагу при розподілі навантаження. З огляду на необхідність комунікації з мережею в реальному часі, що може збільшити трафік у системі, поточний стан системи використовується для прийняття рішень щодо управління навантаженням.

Підходи до балансування навантаження можна розділити на такі типи:

Кругова переадресація (Round Robin forwarding): Простий метод розподілу трафіку між групою серверів. Балансувальник трафіку послідовно відправляє клієнтські запити кожному комутатору у списку. Коли досягається кінець списку, балансувальник повертається назад і розпочинає процес знову.

Найменше з'єднання: Цей підхід враховує поточне завантаження сервера та відправляє запит на сервер із найменшою кількістю активних сесій.

Підходи на основі хешування: Використовує хеш-функцію для розподілу клієнтів між конкретними серверами без використання статичних даних.

Зважений час відповіді: В цьому підході інформація про час відгуку постійно надходить від серверів, щоб визначити, який сервер є найшвидшим протягом певного періоду часу. Наступний запит на доступ до сервера надсилається саме на цей сервер, що дозволяє рівномірно розподілити навантаження на пул доступних серверів.

Адаптивний: Цей підхід використовує комбінацію відомостей з верхніх та нижніх рівнів мережі для прийняття рішень щодо розподілу запитів. Інформація про дані з 4 та 7 рівнів мережі поєднується з даними з 2 та 3 рівнів мережі, враховуючи стан сервера, стан додатків, що працюють у мережі, стан мережної інфраструктури та рівень перевантаження.

2.2 Багатошляхова маршрутизація

Описаний механізм, згідно з дослідженням, ґрунтується на використанні кількох шляхів, при цьому враховує фактичне завантаження кожного шляху для здійснення прийняття рішень щодо пересилання. Цей підхід базується на розрахунку k шляхів, доступних для пересилання потоків між вихідним та кінцевим вузлом, і обирається шлях з найменшим завантаженням для обробки конкретного вхідного потоку. Основною метою є уникнення перевантажених точок у мережі, і, крім того, розподіл трафіку між існуючими шляхами сприяє полегшенню управління несправностями.

Конкретний процес виглядає наступним чином:

З'явленням нового потоку перевіряється, чи існує відповідний шлях.

Якщо k-шлях між вихідним та кінцевим вузлом відсутній, створюється новий k-шлях.

Обчислюється вартість всіх шляхів у наборі k-шляхів (в даній версії враховується лише завантаження), і потік призначається шляху з найменшою вартістю.

Зчитується загальна статистика мережі та порівнюється з попередніми значеннями.

Реєструється споживання ресурсів користувачами (агрегована пропускна спроможність користувача).

Відновлюються метрики всіх каналів (і шляхів) (доступний бітрейт, швидкість втрати пакетів, вартість).

Перехід до кроку 3 (або 1, якщо виявлений новий потік).

2.3 Конструювання трафіку

Завдяки використанню раніше визначених шляхів, централізований метод конструювання трафіку дозволяє оптимізувати формування шляхів. У процесі конструювання трафіку використовується набір мінімальних шляхів, які не перетинаються. Це сприяє зниженню ймовірності повторного формування шляхів при зміні топології мережі чи параметрів каналів передачі даних.

Наприклад, для графу, представленого на рис. 2.1, існує 4 множини мінімальних неперетинних шляхів між вершинами p_2 і p_{13}.

$$W_1 = \{B_1(2,13), B_2(2,13)B_3(2,13)\}$$
$$W_2 = \{B_4(2,13), B_2(2,13)B_3(2,13)\}$$
$$W_3 = \{B_4(2,13), B_2(2,13)B_5(2,13)\}$$
$$W_4 = \{B_1(2,13), B_2(2,13)B_5(2,13)\}$$
$$B_1(2,13) = (p_2 \rightarrow p_1 \rightarrow p_4 \rightarrow p_5 \rightarrow p_6 \rightarrow p_{12} \rightarrow p_{13})$$
$$B_2(2,13) = (p_2 \rightarrow p_3 \rightarrow p_7 \rightarrow p_8 \rightarrow p_{13})$$
$$B_3(2,13) = (p_2 \rightarrow p_{15} \rightarrow p_{11} \rightarrow p_{10} \rightarrow p_{14} \rightarrow p_{13})$$
$$B_4(2,13) = (p_2 \rightarrow p_1 \rightarrow p_4 \rightarrow p_6 \rightarrow p_{12} \rightarrow p_{13})$$
$$B_5(2,13) = (p_2 \rightarrow p_{15} \rightarrow p_{11} \rightarrow p_{10} \rightarrow p_9 \rightarrow p_{14} \rightarrow p_{13})$$

У ролі показника якості будемо використовувати кількість перемикань між вузлами у звичайних комп'ютерних мережах. Зазвичай цей показник активно використовується в класичних мережах, набагато частіше, ніж у активних мережах.

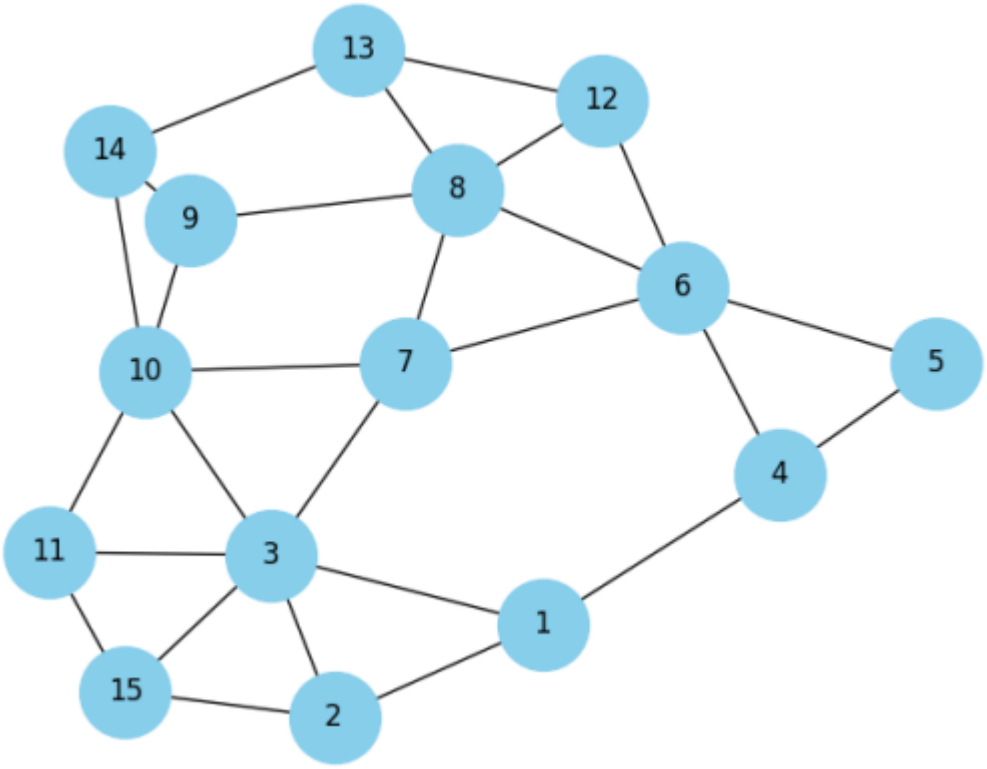


Рисунок 2.1 - Граф мережі

Під час використання алгоритму багатошляхової маршрутизації за вектором дистанції для визначення маршруту від вершини p_2 до вершини p_{13} формується Таблиця 2.1. У цій таблиці містяться вектори шляхів від вихідної вершини p_s до цільової вершини p_d через кожну суміжну вершину p_a.

Таблиця 2.1 - Вектори дистанції вершини p_2

$R_i(s,d)$	p_s	p_d	M_j
$R_1(2,13)$	p_2	p_{13}	6
$R_2(2,13)$	p_2	p_{13}	4
$R_3(2,13)$	p_2	p_{13}	5
$R_4(2,13)$	p_2	p_{13}	5
$R_5(2,13)$	p_2	p_{13}	6

Створюємо таблиці векторів дистанції від усіх проміжних вузлів p_i на даному маршруті до кінцевого вузла p_d. Для конкретного маршруту $B_1(2,13) = (p_2 \rightarrow p_1 \rightarrow p_4 \rightarrow p_5 \rightarrow p_6 \rightarrow p_{12} \rightarrow p_{13})$ формуємо таблиці 2.2 - 2.6 векторів маршруту.

Таблиця 2.2 - Вектори дистанції вершини p_1

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_1(1,13)$	p_1	p_{13}	p_4	5

Таблиця 2.3 - Вектори дистанції вершини p_4

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_1(4,13)$	p_4	p_{13}	p_5	4

Таблиця 2.4 - Вектори дистанції вершини p_5

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_1(5,13)$	p_5	p_{13}	p_6	3

Таблиця 2.5 - Вектори дистанції вершини p_6

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_1(6,13)$	p_6	p_{13}	p_{12}	2

Таблиця 2.6 - Вектори дистанції вершини p_{12}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_1(12,13)$	p_{12}	p_{13}	p_{13}	1

Для шляху $B_2(2,13) = (p_2 \rightarrow p_3 \rightarrow p_7 \rightarrow p_8 \rightarrow p_{13})$ будуємо таблиці 2.7 - 2.9 вектору шляху.

Таблиця 2.7 - Вектори дистанції вершини p_3

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_2(3,13)$	p_3	p_{13}	p_7	3

Таблиця 2.8 - Вектори дистанції вершини p_7

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_2(7,13)$	p_7	p_{13}	p_8	2

Таблиця 2.9 - Вектори дистанції вершини p_8

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_2(8,13)$	p_8	p_{13}	p_{13}	1

Для шляху $B_3(2,13) = (p_2 \rightarrow p_{15} \rightarrow p_{11} \rightarrow p_{10} \rightarrow p_{14} \rightarrow p_{13})$ будуємо таблиці 2.10 - 2.13 вектору шляху.

Таблиця 2.10 - Вектори дистанції вершини p_{15}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_3(15,13)$	p_{15}	p_{13}	p_{11}	4

Таблиця 2.11 - Вектори дистанції вершини p_{11}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_3(11,13)$	p_{11}	p_{13}	p_{10}	3

Таблиця 2.12 - Вектори дистанції вершини p_{10}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_3(10,13)$	p_{10}	p_{13}	p_{14}	2

Таблиця 2.13 - Вектори дистанції вершини p_{14}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_3(14,13)$	p_{14}	p_{13}	p_{13}	1

Для шляху $B_4(2,13) = (p_2 \rightarrow p_1 \rightarrow p_4 \rightarrow p_6 \rightarrow p_{12} \rightarrow p_{13})$ будуємо таблиці 2.14 – 2.17 вектору шляху.

Таблиця 2.14 - Вектори дистанції вершини p_1

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_4(1,13)$	p_1	p_{13}	p_4	4

Таблиця 2.15 - Вектори дистанції вершини p_4

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_4(4,13)$	p_4	p_{13}	p_6	3

Таблиця 2.16 - Вектори дистанції вершини p_6

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_4(6,13)$	p_6	p_{13}	p_{12}	2

Таблиця 2.17 - Вектори дистанції вершини p_6

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_4(12,13)$	p_{12}	p_{13}	p_{13}	1

Для шляху $B_5(2,13) = (p_2 \rightarrow p_{15} \rightarrow p_{11} \rightarrow p_{10} \rightarrow p_9 \rightarrow p_{14} \rightarrow p_{13})$ будуємо таблиці 2.18 – 2.22 вектору шляху.

Таблиця 2.18 - Вектори дистанції вершини p_{15}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_5(15,13)$	p_{15}	p_{13}	p_{11}	5

Таблиця 2.19 - Вектори дистанції вершини p_{11}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_5(11,13)$	p_{11}	p_{13}	p_{10}	4

Таблиця 2.20 - Вектори дистанції вершини p_{10}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_5(10,13)$	p_{10}	p_{13}	p_9	3

Таблиця 2.21 - Вектори дистанції вершини p_9

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_5(9,13)$	p_9	p_{13}	p_{14}	2

Таблиця 2.22 - Вектори дистанції вершини p_{14}

$R_j(s,d)$	p_s	p_d	p_a	M_j
$R_5(14,13)$	p_{14}	p_{13}	p_{13}	1

Для оптимізації процедури формування нових шляхів, побудуємо матрицю векторів оптимальних шляхів на основі шляху $B_2(2,13) = (p_2 \rightarrow p_3 \rightarrow p_7 \rightarrow p_8 \rightarrow p_{13})$

На рис 2.2 представлена блок схема конструювання трафіку.

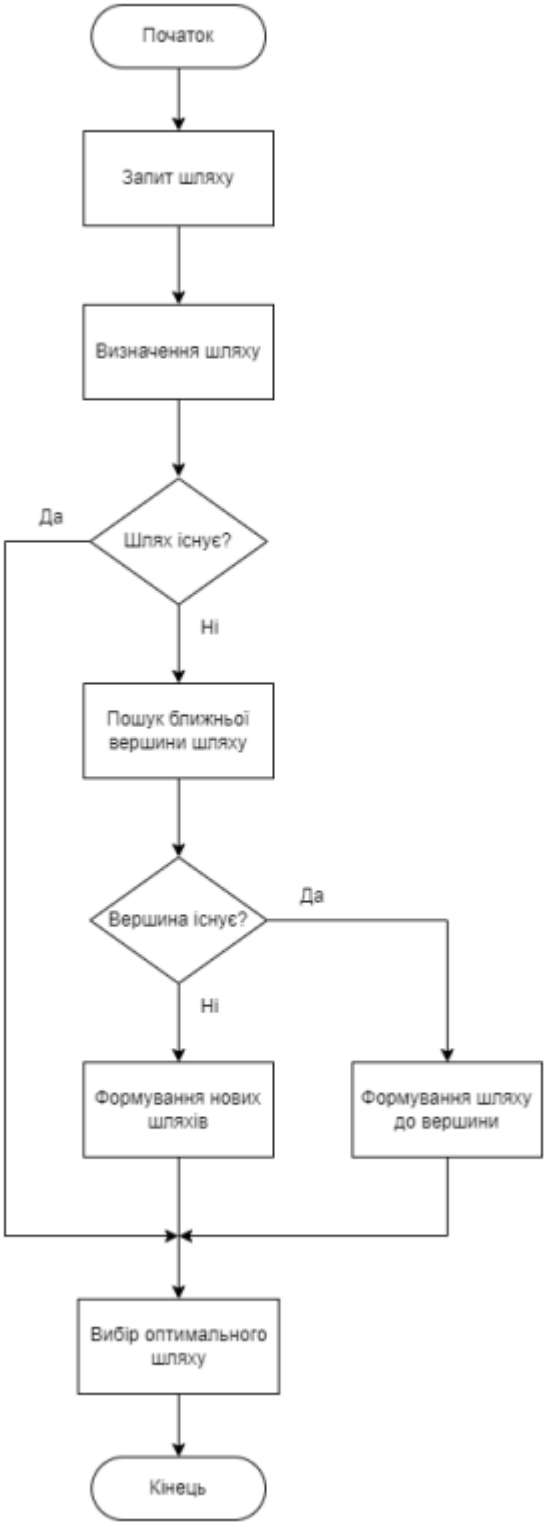


Рисунок 2.2 – Алгоритм конструювання трафіку

Розглянемо логіку псевдокод формування шляху.

Позначення:

– p: точки (комутатори)

– $R_j(s,d)$: Вектор шляху.

– $B_i(p_s,p_d\ )$: шлях із вершини p_s до вершини p_d;

– p_d: кінцева вершина шляху;

– p_s: початкова вершина шляху;

– $E = ||e_{i,j}||$ множина граней графа шляхів;

– $A = \ ||\ ai,j\ ||$: матриця суміжних вершин с початковими вершинами шляхів від вершин p_i до вершин p_j ;

– ai,j -- ближня вершина p_a до вершини v_i шляху в напрямку вершини v_{j} ;

– $P_k = \{p_k, k = 1{,}2..r\ \}$: множина r вершин, суміжних з вершинами множини $P_z = \{p_z, z = 1{,}2..w\}$;|

– $R_i(p_s,p_d) = (\ p_s,p_d,p_a,M_i)$: вектор шляху із вершини p_ss у вершину p_d

– $Q_i(p_s,p_d\ )$: кількість граней шляху $L_i(p_s,p_d,\ )$;

– M_i : метрика шляху $L_i(p_s,\ p_d\ )$;

Наприклад, розглянемо процес створення вектору $R_j\ (p_1,p_{13}p_a)$ для маршруту $L_i(p_4{,}p_{13})$ від вершини p_4 до вершини p_{13}. Для маршруту $L_i(p_1{,}p_{13})$ у таблиці $D = ||d_{i,j}||$ відсутнє значення $d_{1,13}$.

Етап 1.

При $i = 1$ множина вершин $P_1 = \{p_1\}$. За допомогою таблиці $E = ||e_{i,j}||$ визначаємо множину вершин $P_2 = \{p_2,p_3,p_4\}$, які є суміжними з вершиною $p_1 \in P_1$. Формуємо шлях із вершини $p_1 \in\ P_i$. до вершин $P_2 = \{p_2,p_3p_4\}$,.

Шлях $B_1 = (p_1,p_2) =\ B(p_1 \rightarrow p_2)$. Вектор шляху із вершини p_2 у вершину p_2 дорівнює $B_1(p_1p_2p_3)$. Метрика шляху $B_1(p_1,p_2)$ дорівнює $M_1(p_1,p_2) = 1$.

З таблиці $D = ||d_{i,j}||$ для вершини p_2 визначаємо вектор $R_1(p_2 p_{13} p_3)$. шляху $B_1(p_2 p_{13}) = (p_2 \to p_3 \to p_7 \to p_8 \to p_{13})$. Метрика шляху $B_1(p_2 p_{13})$ дорівнює $M_1(p_2, p_{13}) = 4$.

Шлях $B_2 = (p_1, p_3) = B\ (p_1 \to p_3)$. Вектор шляху із вершини p_1 у вершину p_3 дорівнює $R_2(p_1 p_3 p_3)$. Метрика шляху $B_2(p_1, p_3)$ дорівнює $M_2(p_1, p_3) = 1$.

З таблиці $D = ||d_{i,j}||$ для вершини v_3 визначаємо вектор $R_1(p_3 p_{13} p_7)$ шляху $R_1(p_3 p_{13}) = (p_3 \to p_7 \to p_8 \to p_{13})$. Метрика шляху $B_1(p_3 p_{13})$ дорівнює $M_1(p_3, p_{12}) = 3$.

Формуємо шлях $B_2(p_1, p_{13}) = B_2(p_1, p_3) + B_1 = (p_3, p_{13})$ із вектором $R_2(v_1, v_3 v_{13})$.

Шлях $B_2(p_1, p_{13}) = (p_1 \to p_3 \to p_7 \to p_8 \to p_{13})$. з метрикою $M_2(p_1, p_{13}) = 4$.

Шлях $B_3 = (p_1, p_4) = (p_1 \to p_4)$. Вектор шляху із вершини p_1 у вершину p_4 дорівнює $R_3(p_1 p_4 p_4)$. Метрика шляху $B_3(p_1, p_4)$ дорівнює $M_3(p_1, p_4) = 1$.

Таким чином, після першої ітерації формуються шляхи:

$B_1(p_1, p_{13}) = (p_1 \to p_2 \to p_3 \to p_7 \to p_8 \to p_{13}),\ M_1(p_1, p_{13}) = 5;$

$B_2(p_1, p_{13}) = (p_1 \to p_3 \to p_7 \to p_8 \to p_{13}),\ M_2(p_1, p_{13}) = 4;$

$B_3(p_1, p_4) = (p_1 \to p_4),\ M_3(p_1, p_4) = 1.$

Значення векторів $R_2(p_1 p_3 p_{13})$ і $R_3(p_1 p_4 p_4)$. зберігаються в таблицю $E = ||e_{i,j}||$.

Етап 2.

Шлях $B_3(p_1, p_4) = (p_1 \to p_4)$ формує вершини p_5 і p_6, які є суміжними з кінцевою вершиною p_4. Формується шлях $B_3(p_1, p_5) = (p_1 \to p_4 \to p_5)$, з метрикою $M_1(p_1, p_5) = 2$ і шлях $B_4(p_1, p_6) = (p_1 \to p_4 \to p_6)$ з метрикою $M_1(p_1, p_6) = 2$. Вектор шляху із вершини p_1 у вершину p_5 дорівнює $R_3(p_1 p_5 p_4)$. Вектор шляху із вершини p_1 у вершину p_2 дорівнює $R_4(p_1 p_6 p_4)$.

Таким чином, після другої ітерації формуються шляхи:

$B_3(p_1, p_5) = (p_1 \to p_4 \to p_5),\ M_3(p_1, p_5) = 2.$

$B_4(p_1, p_6) = (p_1 \to p_4 \to p_6),\ M_4(p_1, p_6) = 2.$

Значення векторів $R_3(p_1 p_5 p_4), R_3(p_1 p_6 p_4)$ записуються в таблицю $E = ||e_{i,j}||$.

Етап 3.

Шлях $B_4(p_1, p_6) = (p_1 \to p_4 \to p_6)$ формує вершинуp_{12}, яка є суміжною з кінцевою вершиною p_6. Формується шлях $B_4(p_1, p_{12}) = (p_1 \to p_4 \to p_6 \to p_{12})$ з метрикою $M_4(p_1, p_{12}) = 3$. Вектор шляху із вершини p_1 у вершину p_{12} дорівнює $R_4(p_1 p_{12} p_4)$.

Таким чином, після третьої ітерації формується шлях:

$B_4(p_1, p_{12}) = (p_1 \to p_4 \to p_6 \to p_{12})$, $M_4(p_1, p_{12}) = 3$.

Значення векторів $R_4(p_1 p_{12} p_4)$ записуються у таблицю $E = ||e_{i,j}||$.

Етап 4.

Шлях $B_4(p_1, p_{12}) = (p_1 \to p_4 \to p_6 \to p_{12})$ формує вершину p_{13}, яка є суміжною з кінцевою вершиною p_{13},. Формується шлях $B_4(p_1, p_{13}) = (p_1 \to p_4 \to p_6 \to p_{12} \to p_{13})$ з метрикою $M_4(p_1, p_{13}) = 4$. Вектор шляху із вершини p_1 у вершину p_{13} дорівнює $R_4(p_1 p_{13} p_4)$.

Таким чином, після четвертої ітерації формується шлях:

$B_4(p_1, p_{13}) = (p_1 \to p_4 \to p_6 \to p_{12} \to p_{13})$, $M_4(p_1, p_{13}) = 4$.

Значення векторів $R_4(p_1 p_4 p_{13})$ записуються у таблицю $E = ||e_{i,j}||$.. Формуємо таблицю 2.24 вектору дистанції до вершини p_d.

Таблиця 2.24 - Вектори дистанції вершини p_1

$R_i(s,d)$	p_s	p_d	p_a	M_i
$R_2(1,13)$	p_1	p_{13}	p_3	4
$R_4(1,13)$	p_1	p_{13}	p_4	4

Будуємо матрицю векторів оптимальних шляхів на основі шляху $B_2(p_1, p_{13}) = (p_1 \to p_3 \to p_7 \to p_8 \to p_{13})$

3 СИМУЛЯЦІЯ СИСТЕМИ

3.1 Огляд інструментів симуляції

3.1.1 Mininet

Mininet - це інструмент для створення віртуальних мереж для тестування та експериментів в галузі комп'ютерних мереж. Він надає можливість створювати віртуальні мережі, що містять віртуальні хости, комутатори, контролери та канали зв'язку. Mininet дозволяє розробникам емулювати складні мережеві топології та тестувати різноманітні сценарії без потреби в реальному обладнанні.

Віртуальні компоненти: Mininet створює віртуальні хости, комутатори та інші мережеві компоненти, які імітують реальні пристрої. Це дозволяє розробникам тестувати свої додатки в віртуальному середовищі.

Легкість використання: Mininet надає простий інтерфейс командного рядка, що дозволяє легко створювати, конфігурувати та запускати віртуальні мережі.

Підтримка Python API: Для створення мереж та проведення експериментів в Mininet передбачено Python API. Це дозволяє автоматизувати та керувати процесом емуляції.

Тестування SDN: Mininet часто використовується для тестування та розробки програм для програмованих мереж (SDN). Розробники можуть емулювати SDN-контролери та перевіряти їхню взаємодію з мережею.

Параметризація топології: Mininet дозволяє задавати різні параметри топології мережі, такі як пропускна здатність, затримка та інші, для тестування різних умов мережі.

Відкритий доступ: Mininet є відкритим програмним забезпеченням та доступним для використання за ліцензією Apache 2.0.

Mininet дозволяє розробникам ефективно тестувати та емулювати різноманітні сценарії мережевого взаємодії в контрольованому віртуальному середовищі.

3.1.2 Ryu

Ryu - це фреймворк для розробки програмного забезпечення для управління мережами з використанням концепцій програмованих мереж (Software-Defined Networking, SDN). Він надає розробникам інструменти та бібліотеки для створення SDN-додатків, які можуть контролювати поведінку мережі та її елементів.

Підтримка протоколів SDN: Ryu підтримує різні протоколи управління мережевими пристроями, такі як OpenFlow та OF-config. Це дозволяє розробникам створювати програми для централізованого керування мережевим обладнанням.

Python: Фреймворк розроблений на мові програмування Python, що робить його доступним та зручним для розробників з різним досвідом.

Модульність: Ryu є модульним та розширюваним. Розробники можуть використовувати окремі модулі, розширювати їх функціональність та створювати власні додатки.

Спрощене API: Робота з Ryu включає в себе використання простого та зрозумілого API, що полегшує процес розробки.

Ліцензія Apache 2.0: Код фреймворку розміщено під ліцензією Apache 2.0, що надає відкритий доступ та можливість використання для різних цілей.

Підтримка SDN-контролера: Ryu може виступати в ролі SDN-контролера, оброблюючи інструкції, які надходять від SDN-додатків та управляючи поведінкою мережі.

Використання в наукових дослідженнях: Фреймворк часто використовується в наукових дослідженнях та експериментах для дослідження нових концепцій та ідей у сфері SDN.

Загалом, Ryu є потужним інструментом для розробки та експериментів у сфері програмованих мереж, який дозволяє розробникам створювати та тестувати SDN-рішення.

3.1.3 Python

У рамках цього дослідження використовувався мова програмування Python для розробки програми, яка реалізує алгоритм балансування трафіку та визначення топології в звичайних мережах.

Python - це високорівнева, інтерпретована, об'єктно-орієнтована мова програмування, яка відзначається простим та читабельним синтаксисом. Вона була розроблена Гвідо ван Россумом і вперше випущена у 1991 році. Python став однією з найпопулярніших мов програмування завдяки своїй простоті, гнучкості та великому спільнотові користувачів.

Читабельний синтаксис: Python відомий своїм простим і легко читабельним синтаксисом, що робить його ідеальним для початківців та досвідчених розробників.

Інтерпретована мова: Python використовується у вигляді інтерпретованої мови, що дозволяє виконувати код без необхідності компіляції. Це полегшує розробку та тестування програм.

Об'єктно-орієнтована: Python підтримує об'єктно-орієнтовану парадигму програмування, що дозволяє використовувати об'єкти та класи для структурування коду.

Багатий стандартний бібліотек: Python постачається з великою стандартною бібліотекою, яка включає різноманітні модулі та фреймворки для вирішення різних завдань.

Крос-платформенність: Код, написаний на Python, може запускатися на різних операційних системах, включаючи Windows, macOS і Linux.

Велика спільнота: Python має активну та розгалужену спільноту розробників, яка надає підтримку, допомогу та розробляє різноманітні бібліотеки та фреймворки.

Використання в різних областях: Python використовується в різних областях, таких як веб-розробка, аналіз даних, штучний інтелект, машинне навчання, наукові дослідження та інше.

3.1.4 Iperf

iPerf - це відкрите програмне забезпечення, яке надає можливість вимірювання пропускної здатності мережі. Цей інструмент дозволяє тестувати швидкість передачі даних між двома вузлами в мережі. Основні функції iPerf включають:

Вимірювання пропускної здатності: iPerf може генерувати трафік і вимірювати, яка кількість даних може бути передана через мережу за одиницю часу. Це дозволяє оцінити ефективність мережі та виявити можливі обмеження шляху передачі даних.

Тестування різних протоколів: iPerf підтримує вимірювання пропускної здатності для різних мережевих протоколів, таких як TCP, UDP та SCTP. Це дозволяє користувачам аналізувати різні аспекти мережевої продуктивності.

Мережеві тестування: iPerf може використовуватися для тестування мережі з різних точок, визначаючи пропускну здатність та відмічаючи будь-які аномалії чи проблеми, які можуть виникнути під час передачі даних.

Сервер-клієнтська архітектура: iPerf використовує модель клієнт-сервер для вимірювання пропускної здатності. Один інстанс iPerf працює в режимі сервера, а інший - в режимі клієнта, щоб забезпечити передачу даних через мережу.

Параметризація тестів: iPerf дозволяє користувачам налаштовувати різні параметри тесту, такі як розмір пакетів, тривалість тесту, порти та інші. Це робить інструмент гнучким для використання в різних сценаріях тестування.

3.1.5 VM VirtualBox

VirtualBox - це безкоштовний та відкритий гіпервізор (віртуалізаційний рішення), розроблений компанією Oracle. Він надає зручний і ефективний спосіб створення та управління віртуальними машинами (ВМ) на одному фізичному комп'ютері. Основною метою VirtualBox є віртуалізація апаратних ресурсів для запуску ізольованих операційних систем (ОС) на одному комп'ютері.

Підтримка різних операційних систем: VirtualBox дозволяє вам встановлювати та запускати різні операційні системи від Windows і Linux до macOS та інших.

Низькі вимоги: Віртуальні машини використовують ресурси фізичного комп'ютера, проте VirtualBox спроектований для ефективного використання цих ресурсів, забезпечуючи при цьому зручний інтерфейс для користувача.

Спільна підтримка гостьових ОС: VirtualBox підтримує велику кількість гостьових операційних систем, включаючи різні версії Windows, Linux, macOS, інші Unix-подібні ОС, а також DOS та інші.

Зручний інтерфейс: Вбудований графічний інтерфейс VirtualBox забезпечує простий та зрозумілий спосіб управління віртуальними машинами, а також підтримує управління через командний рядок.

Підтримка віртуальних пристроїв: VirtualBox може емулювати різні віртуальні пристрої, такі як віртуальні жорсткі диски, оптичні диски, мережеві адаптери, USB-пристрої та інші.

Спільна кліпборд і перетягування: VirtualBox надає можливість обміну даними між гостьовою та хостовою ОС за допомогою спільного кліпборду і функції перетягування файлів.

Спільні папки: Ви можете налаштовувати спільні папки між хостовою та гостьовою операційними системами для легкого обміну файлами.

VirtualBox є популярним інструментом для вивчення та тестування різноманітних операційних систем, розробки та тестування програм, а також для забезпечення ізольованого середовища для різних завдань.

3.2 Реалізація алгоритму

Для успішної реалізації алгоритму багатошляхової маршрутизації в нашому проекті використовується структура даних, відома як граф. Граф дозволяє нам відображати мережу у вигляді множини вершин, які з'єднані гранями, кожна з яких може мати вагу. У нашому випадку вершинами є роутери, а грані представляють з'єднання між двома роутерами з власними метриками.

Клас Graph містить масиви вершин і граней, а також методи для додавання нових вершин і граней. Класи Node і Edge використовуються для представлення вершин і граней відповідно.

Алгоритм багатошляхової маршрутизації реалізований через інтерфейс класу LoadBalancingAlgorithm, який отримує ідентифікатори початкового та кінцевого комутаторів, а також граф, що відображає мережу. Алгоритм ітерується через грані на шляху для знаходження найкоротшого шляху, і його завершення гарантується.

Додатково, ми використовуємо умовні позначення, такі як довжина шляху між вершинами та кількість переходів. Алгоритм забезпечує завершення, якщо умова найкоротшого шляху для всіх вершин виконується.

Таким чином, ми використовуємо графову структуру для ефективного моделювання мережі та застосовуємо алгоритм багатошляхової маршрутизації для забезпечення оптимальних маршрутів у звичайних мережах.

3.3 Реалізація контролера

Для розробки контролера був використаний фреймворк Ryu. Ryu базується на подієво-орієнтованій парадигмі програмування, де хід виконання програми визначається подіями. Програми Ryu слухають події, які реалізуються різними класами. Події можуть виникати як внаслідок внутрішніх процесів у Ryu, так і через програми, що використовують Ryu. Програми створюють свої власні події, використовуючи методи, надані класом RyuApp. Наприклад, метод send_event може бути використаний для генерації події.

З іншого боку, програма, розроблена за допомогою Ryu, може позначити свій інтерес до конкретного типу події, використовуючи метод-обробник. У Ryu це реалізується за допомогою структури Python, відомої як "декоратор". Декоратор, визначений у Ryu для обробки подій, називається set_ev_cls і знаходиться в модулі ryu.controller.handler.

Детальний процес моделі застосунку Ryu можна розглянути на рис. 3.1.

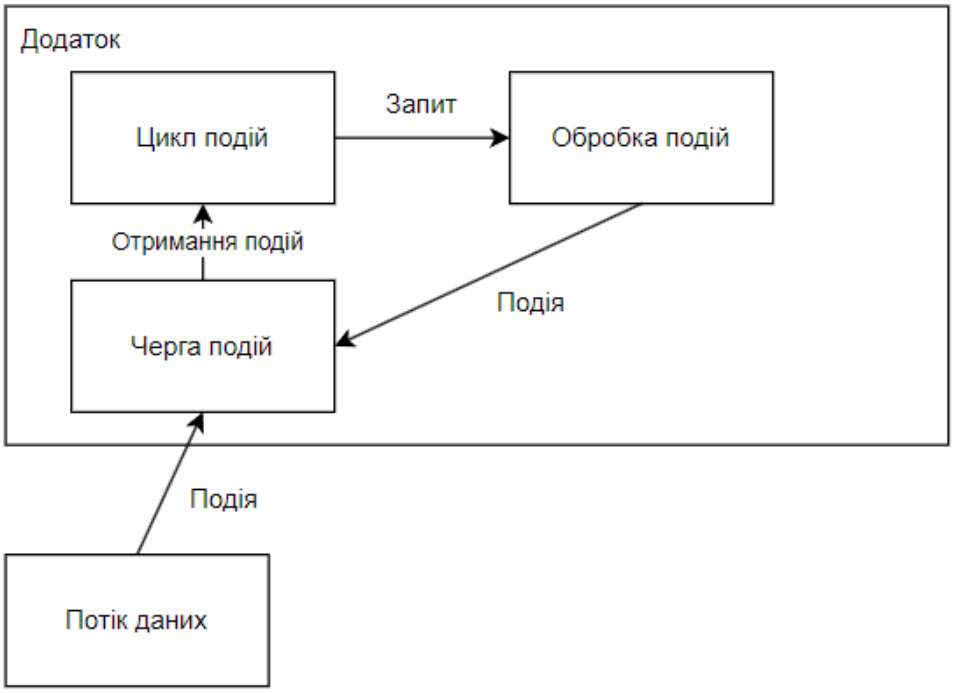


Рисунок 3.1 – Модель Ryu застосунку

Комутатор OpenFlow (див. рис. 3.2) складається з однієї або кількох таблиць потоків і таблиць груп, які виконують пошук і пересилання пакетів, а також каналу OpenFlow до зовнішнього контролера. Контролер управляє перемикачем через протокол OpenFlow. З використанням цього протоколу контролер може додавати, оновлювати та видаляти записи потоку як реакцію на пакети, так і в проактивному режимі. Термін "Datapath" використовується для позначення звичайних комутаторів OpenFlow, оскільки вони реалізують лише пересилання пакетів, порівняно з контролерами OpenFlow, які забезпечують інтелектуальне управління шляхами.

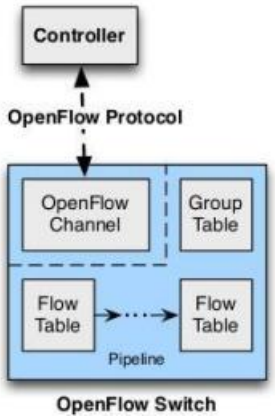


Рисунок 3.2 - Основні компоненти комутатора OpenFlow

Для фреймворка OpenFlow був використаний модуль ryu.controller.ofp_event, який надає класи подій, що описують прийом повідомлень OpenFlow від звичайних мережних шляхів (комутаторів).

Клас EventOFPPacketIn відповідає повідомленням про надходження пакетів. Контролер OpenFlow автоматично розкодовує повідомлення OpenFlow, отримані від комутаторів, і передає ці події додаткам Ryu, які виявили зацікавленість, використовуючи декоратор set_ev_cls.

Класи подій OpenFlow мають ряд атрибутів, які визначають їхню функціональність. Один з ключових атрибутів - msg, описує конкретне повідомлення OpenFlow, яке сприймається системою. Додатково, атрибут msg.datapath ідентифікує конкретний комутатор OpenFlow, від якого отримано вказане повідомлення.

У контексті програмування подій OpenFlow використовується декоратор set_ev_cls. Цей декоратор надає можливість визначити метод як обробник певної події. Отже, метод, який декорований цим декоратором, вважається обробником подій і викликається системою у відповідь на виникнення конкретної події OpenFlow.

Щодо маршрутизації, процес включає наступні етапи:

Застосування політики балансування навантаження: Визначаються комутатори, які будуть обрані для обробки трафіку з урахуванням політики балансування навантаження.

Створення відповідності для пакетів визначається структура пакетів, для якої буде використовуватися дана політика балансування.

Складання списку дій формується список конкретних дій, які повинні бути виконані для будь-якого пакету, що відповідає визначеній відповідності.

Запис правила на комутаторі створюється та встановлюється правило для обробки потоку пакетів на вибраних комутаторах.

Надсилання поточного пакету обрані комутатори виконують визначені дії для обробки отриманих пакетів.

Такий підхід дозволяє ефективно маршрутизувати трафік в мережі, забезпечуючи балансування навантаження та відповідність встановленим політикам.

Вибір комутатора відбувається на основі політики балансування навантаження, призначаючи цілочисельний ідентифікатор кожному комутатору. Створюється відповідність для пакетів на основі певних полів, таких як EtherType, мережевий протокол, адреса призначення мережі та адреса порту призначення. Формується список дій для обробки пакетів з аналогічною структурою. Потім створюється правило для цього потоку на комутаторі, вказуючи жорсткий та м'який тайм-аути для керування часом життя потоку. Нарешті, поточний пакет відсилається на вибраний комутатор. Цей процес дозволяє забезпечити ефективну маршрутизацію з використанням політик балансування навантаження в звичайних мережах.

4 ТЕСТУВАННЯ СИСТЕМИ

4.1 Топологія мережі

Перевірка роботи програми виконуватиметься на стандартній мережі, яка складається з 6 хостів і 17 комутаторів. Ми будемо використовувати кількість переходів як метрику ефективності. Схема цієї мережі зображена на рис. 4.1

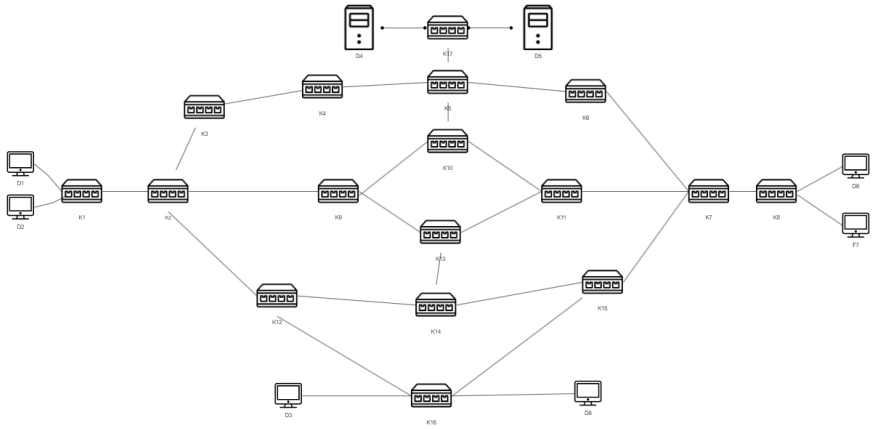


Рисунок 4.1 – Топологія для тестування

4.2 Тестування програми

Виконаємо перевірку доступності хоста 6 із хоста 1. Задачею є виявлення чотирьох найкоротших шляхів між h1 і h6 в рамках визначеної топології.

1. K1 – K2 – K9 – K10 – L11 – K7 – K8
2. K1 – K2 – K9 – K13 – K11 – K7 – K8
3. K1 – K2 – K12 – K14 – K15 – K7- K8
4. K1 – K2 – K12 – K16 – K15 – K7- K8

Використовуючи команду ping, ми перевіримо стан з'єднання між D1 і D2. Див. рис. 4.2

```
mininet> D1 ping -c1 D6
PING 10.10.10.6 (10.10.10.6) 56(84) bytes of data.
64 bytes from 10.10.10.6: icmp_seq=1 ttl=64 time=563 ms

--- 10.10.10.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 592.766/592.766/592.766/0.000 ms
mininet> h1 ping -c1 h6
PING 10.10.10.6 (10.10.10.6) 56(84) bytes of data.
64 bytes from 10.10.10.6: icmp_seq=1 ttl=64 time=0.513 ms

--- 10.10.10.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.511/0.511/0.511/0.000 ms
```

Рисунок 4.2 - Статистика пінгу

З вікна управління контролером можна спостерігати, що контролер обрав шлях № 2 як оптимальний маршрут між D1 і D6. Цей вибір відповідає фактичній ситуації. Див. рис. 4.3

```
vita@vita-VirtualBox:~/SDN/LoadBalancer$ ryu-manager --observe-links controller.py
Loading app controller.py
Loading app ryu.controller.ofp_handler
Loading app ryu.topology.switches
instantiating app controller.py of Controller
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
[1, 2, 9, 13, 11, 7, 8] = metric: 7
```

Рисунок 4.3 – Вікно запуску контролера

За допомогою команди ovs-ofctl можна отримати всі записи про потоки у таблицях комутатора. На зображеннях 4.4 – 4.9 наведені записи про потоки для комутаторів шляху №2. Хост D1, з IP-адресою 10.10.10.1, для досягнення хоста D6, який має IP-адресу 10.10.10.6, матиме вхідний мережевий інтерфейс, доступний через встановлені комутатори.

В комутаторі є вихідні дії (output actions), які перенаправляють потік на вихідний мережевий інтерфейс відповідно до поля: actions=output:K1-eth2. У випадку багатошляхової маршрутизації запис потоку матиме кілька полів bucket і actions, де кожне поле представляє один шлях.

На зображенні 4.10 можна відзначити, що для комутатора K17 відсутні записи для маршрутизації потоку, оскільки комутатор не брав безпосередньої участі в встановленні зв’язку між хостом D1 і D6.

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K1
cookie=0x0, duration=45.9765, table=0, n_packets=17, n_bytes=1020, idle_age=1, priority=65535, dl_dst=01:80:c2:00:00:0e, dl_t
ype=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=33.8325, table=0, n_packets=98, n_bytes=5886, idle_age=1, tp,nw_src=10.10.10.6,nw_dst=10.10.10.1 actions
=output:"K1-eth2"
cookie=0x0, duration=33.8324, table=0, n_packets=48, n_bytes=2016, idle_age=33, priority=1,arp,arp_spa=10.10.10.6,arp_tpa=10.
10.10.1 actions=output:"K1-eth2"
cookie=0x0, duration=33.8325, table=0, n_packets=98, n_bytes=98, idle_age=41, ip,nw_src=10.0.0.1,nw_dst=10.0.0.6 actions=outp
ut:"K1-eth3"
cookie=0x0, duration=33.8323, table=0, n_packets=4, n_bytes=168, idle_age=36, priority=1,arp,arp_spa=10.10.10.1,arp_tpa=10.10
.10.6 actions=output:"K1-eth3"
cookie=0x0, duration=45.9765, table=0, n_packets=32, n_bytes=4094, idle_age=13, priority=1,ipv6 actions=drop
cookie=0x0, duration=45.9765, table=0, n_packets=5, n_bytes=346, idle_age=41, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.4 – Записи про потоки для K1

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K2
cookie=0x0, duration=363.185s, table=0, n_packets=554, n_bytes=32640, idle_age=0, priority=65535, dl_dst=01:80:c2:00:00:0e, d
l_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=350.564s, table=0, n_packets=2, n_bytes=98, idle_age=325, tp,nw_src=10.10.10.6,nw_dst=10.10.10.1 actions
=output:"K2-eth4"
cookie=0x0, duration=350.564s, table=0, n_packets=47, n_bytes=1974, idle_age=317, priority=1,arp,arp_spa=10.0.0.6,arp_tpa=10.
0.0.1 actions=output:"K2-eth4"
cookie=0x0, duration=350.564s, table=0, n_packets=1, n_bytes=636, idle_age=317, priority=1,arp,arp_spa=10.0.0.1,arp_tpa=10.0.
0.6 actions=output:"K2-eth2"
cookie=0x0, duration=350.564s, table=0, n_packets=108, n_bytes=16311, idle_age=319, priority=1,ipv6 actions=drop
cookie=0x0, duration=350.423s, table=0, n_packets=10, n_bytes=455, idle_age=325, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.5 – Записи про потоки для K2

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K9
cookie=0x0, duration=460.2605, table=0, n_packets=526, n_bytes=30960, idle_age=0, priority=65535, dl_dst=01:80:c2:00:00:0e, d
l_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=448.8085, table=0, n_packets=44, n_bytes=1940, idle_age=448, priority=1,arp,arp_spa=10.10.10.6,arp_tpa=1
0.10.10.1 actions=output:"K9-eth1"
cookie=0x0, duration=460.2605, table=0, n_packets=75, n_bytes=12019, idle_age=460, priority=1,ipv6 actions=drop
cookie=0x0, duration=460.2605, table=0, n_packets=20, n_bytes=874, idle_age=456, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.6 – Записи про потоки для K9

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K13
cookie=0x0, duration=482.262s, table=0, n_packets=540, n_bytes=32400, idle_age=1, priority=65535, dl_dst=01:80:c2:00:00:0e, d
l_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=471.588s, table=0, n_packets=51, n_bytes=98, idle_age=479, tp,nw_src=10.10.10.6,nw_dst=10.10.10.1 action
s=output:"K13-eth1"
cookie=0x0, duration=471.588s, table=0, n_packets=45, n_bytes=1890, idle_age=471, priority=1,arp,arp_spa=10.10.10.6,arp_tpa=1
0.10.10.1 actions=output:"K13-eth1"
cookie=0x0, duration=482.164s, table=0, n_packets=60, n_bytes=9380, idle_age=13, priority=1,ipv6 actions=drop
cookie=0x0, duration=482.262s, table=0, n_packets=34, n_bytes=1656, idle_age=472, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.7 – Записи про потоки для K13

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K7
cookie=0x0, duration=729.423s, table=0, n_packets=1055, n_bytes=6700, idle_age=0, priority=65535, dl_dst=01:80:c2:00:00:0e, d
l_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=717.564s, table=0, n_packets=51, n_bytes=98, idle_age=725, tp,nw_src=10.10.10.6,nw_dst=10.10.10.1 action
s=output:"K7-eth4"
cookie=0x0, duration=717.564s, table=0, n_packets=47, n_bytes=1974, idle_age=717, priority=1,arp,arp_spa=10.10.10.6,arp_tpa=1
0.10.10.1 actions=output:"K7-eth4"
cookie=0x0, duration=717.564s, table=0, n_packets=53, n_bytes=2226, idle_age=717, priority=1,arp,arp_spa=10.10.10.1,arp_tpa=1
0.10.10.6 actions=output:"K7-eth2"
cookie=0x0, duration=717.564s, table=0, n_packets=108, n_bytes=16611, idle_age=719, priority=1,ipv6 actions=drop
cookie=0x0, duration=729.423s, table=0, n_packets=10, n_bytes=705, idle_age=725, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.8 – Записи про потоки для K7

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K8
cookie=0x0, duration=796.530s, table=0, n_packets=299, n_bytes=17940, idle_age=2, priority=65535, dl_dst=01:80:c2:00:00:0e, d
l_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=784.922s, table=0, n_packets=51, n_bytes=98, idle_age=792, tp,nw_src=10.10.10.6,nw_dst=10.10.10.1 action
s=output:"K8-eth1"
cookie=0x0, duration=784.922s, table=0, n_packets=47, n_bytes=1974, idle_age=785, priority=1,arp,arp_spa=10.10.10.6,arp_tpa=1
0.0.0.1 actions=output:"K8-eth1"
cookie=0x0, duration=784.922s, table=0, n_packets=53, n_bytes=2226, idle_age=785, priority=1,arp,arp_spa=10.10.10.1,arp_tpa=1
0.0.0.6 actions=output:"K8-eth2"
cookie=0x0, duration=796.530s, table=0, n_packets=7, n_bytes=627, idle_age=793, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.9 – Записи про потоки для K8

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-flows K17
cookie=0x0, duration=843.371s, table=0, n_packets=316, n_bytes=18960, idle_age=0, priority=65535, dl_dst=01:80:c2:00:00:0e, d
l_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=842.457s, table=0, n_packets=30, n_bytes=3304, idle_age=244, priority=1,ipv6 actions=drop
cookie=0x0, duration=843.371s, table=0, n_packets=4, n_bytes=336, idle_age=841, priority=0 actions=CONTROLLER:65535
```

Рисунок 4.10 – Записи про потоки для K17

Швидкість передачі в мережах оцінюється за допомогою терміна "пропускна спроможність", зазвичай вираженої у кбіт/с, Мбіт/с та Гбіт/с. Цей параметр також визначається як фактична швидкість передачі даних по мережі, завжди залишаючись меншою за теоретичну пропускну здатність каналу. Це обумовлено обмеженнями фізичного середовища, доступною обчислювальною потужністю компонентів системи та поведінкою кінцевого користувача.

Mininet володіє командою, яка оцінює пропускну здатність між мережевими пристроями у моделі, яку вона моделює.

Перевірка пропускної здатності включає проведення експерименту 'iperf' між хостами D1 і D6. На рис. 4.11 наведено статистику для кожного порту комутатора K1. Кількість втрачених пакетів становить нуль.

```
vita@vita-VirtualBox:~$ sudo ovs-ofctl -O OpenFlow15 dump-ports K1
OFPT_PORT_REPLY (OF1.5) (xid=0x2): 4 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0  tx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0
, crc=0 duration=1888.379s
  port "K1-eth1": rx pkts=14, bytes=1076, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=750, bytes=47735, drop=0, errs=0, fr
ame=0, over=0, crc=0 duration=1888.449s
  port "K1-eth2": rx pkts=734155, bytes=48454366, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=736145, bytes=32190844981, d
rop=0, errs=0, coll=0 duration=1888.442s
  port "K1-eth3": rx pkts=736145, bytes=32190844981, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=734887, bytes=484500875,
drop=0, errs=0, coll=0 duration=1888.442s
```

Рисунок 4.11 – Статистика трафіку для комутатора K1

ВИСНОВКИ

В заключенні можна зазначити, що SDN визначається як перспективний та передовий метод балансування корпоративних мереж. Ця технологія пропонує інтегрований підхід до управління мережею, що дозволяє піднятися на новий рівень ефективності та гнучкості.

SDN - це технологія, що надалі еволюціонує, і вона прагне вирішувати проблеми поточних структур мережі, що виникають внаслідок зростаючого попиту на ресурси, викликаний появою нових технологій. Сучасне збільшення та очікувані потреби в області інформаційних технологій та телекомунікацій найближчими роками вимагають суттєвих змін у структурі існуючих мереж. Програмно-визначені мережі стають альтернативою, призначеною для вирішення теперішніх та майбутніх проблем телекомунікаційних мереж.

Тенденції, такі як мобільність користувачів, віртуалізація серверів, хмарна інфраструктура, а також необхідність швидко реагувати на зміни в умовах бізнесу, ставлять перед мережами значні вимоги, з якими сучасні традиційні мережеві архітектури не можуть впоратися. Програмовані мережі надають сучасну активну мережеву архітектуру, яка дозволяє перетворити традиційні мережеві магістралі в різноманітні платформи надання послуг.

Шляхом розділення площини управління та площини даних, архітектура SDN на основі OpenFlow відокремлює базову інфраструктуру від застосунків, які її використовують. Це дозволяє мережі стати такою ж програмованою та керованою в масштабі, як і комп'ютерна інфраструктура. SDN мережі сприяють віртуалізації, дозволяючи персоналу керувати серверами, додатками, сховищами та мережами за допомогою єдиного підходу та набору інструментів. Впровадження SDN в операторському середовищі або в корпоративному центрі обробки даних може поліпшити керованість, масштабованість та гнучкість мережі.

Майбутнє мережеве середовище все більше залежатиме від програмного забезпечення, що прискорить темпи інновацій, подібно тому, як це відбувається в області обчислювальних систем та систем зберігання даних. SDN обіцяє перетворити сучасні статичні мережі в гнучкі, програмовані платформи з інтелектом для динамічного розподілу ресурсів, масштабом для підтримки великих центрів обробки даних та віртуалізацією, необхідною для підтримки динамічних, високоавтоматизованих та безпечних хмарних середовищ. З урахуванням численних переваг та імпресивної динаміки розвитку галузі, SDN рухається шляхом становлення новим стандартом для мереж.

Баланс навантаження було досягнуто за допомогою визначення найкращого короткого шляху між двома хостами в спеціально створеній топології. Вибір оптимального шляху був здійснений на основі найменшої вартості. Після визначення найкращого шляху потоки були направлені на відповідні комутатори, які були підключені до хостів. Затримка шляху після балансування навантаження виявилася значно зменшеною, що свідчить про ефективне балансування навантаження. Також статистика ping між двома хостами демонструє ефективне управління трафіком та успішне досягнення балансування навантаження. Отже, вдалося оптимізувати пропускну спроможність та забезпечити низьку затримку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Software-defined networking – Wikipedia : [Електронний ресурс] //Вікіпедія - вільна енциклопедія. – Режим доступу: https://en.wikipedia.org/wiki/Software-defined_networking

2. Software-defined mobile networking – Wikipedia : [Електронний ресурс] //Вікіпедія - вільна енциклопедія. – Режим доступу: https://en.wikipedia.org/wiki/Software-defined_mobile_network

3. List of SDN controller software – Wikipedia : [Електронний ресурс] //Вікіпедія - вільна енциклопедія. – Режим доступу: https://en.wikipedia.org/wiki/List_of_SDN_controller_software

4. R. Kumar Singh, N. S. Chaudhari and K. Saxena / Load Balancing in IP/MPLS Networks: A Survey // SciRes. - 2012. - №2. - C. 151 -153

5. Kulakov Y., Kohan A., Kopychko S., Cherevatenko R / Load Balancing in Software Defined Networks Using Multipath Routing // ICCSEEA 2020. – 2021.

6. M. Liyanage, A. Gurtov, M. Ylianttila / Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture // IEEE Explore.- 2015. –C. 5-9

7. Zhou, Y., Xiao, L., Ruan, L.,/ A Method for Load Balancing based on Software-Defined Network // Advanced Science and Technology Letters.- 2014, pp.40-48.

8. Eppstein, D. / Finding the k shortest paths // SIAM J. Comput. .- 1999.- C. 652 -674 9. Agarwal S., Kodialam M., Lakshman T. / Traffic engineering in software defined networks // IEEE International Conference on Computer Communications.-2013. - C. 2211–2219

10.Ryubook 1.0 documentation : [Електронний ресурс] // RYU SDN Framework . – Режим доступу: https://book.ryu-sdn.org

ДОДАТОК 1

Код

```
class Node:
    def __init__(self, id):
        self.id = int(id)

    def __eq__(self, other_node):
        return self.id == other_node.id

class Edge:
    def __init__(self, from_node, to_node, weight):
        self.from_node = from_node
        self.to_node = to_node
        self.weight = weight

class Graph:
    def __init__(self):
        self.nodes = []
        self.edges = []

    def add_nodes(self, *added_nodes):
        for node in added_nodes:
            self.nodes.append(node)

    def add_edge(self, edge):
        self.edges.append(edge)

class LoadBalancerAlgorithm:
    def __init__(self, graph):
```

```
            self.graph = graph
            self.paths = {}

        def call(self, src, dst):
            dist = [float("Inf")] * len(self.graph.nodes)
            dist[src.id] = 0
            for _ in range(len(self.graph.nodes) - 1):
                for edge in self.graph.edges:
                    if dist[edge.from_node.id] != float("Inf") and dist[edge.from_node.id] +
edge.weight < dist[edge.to_node.id]:
                        dist[edge.to_node.id] = dist[edge.from_node.id] + edge.weight
                        self.paths[edge.to_node.id] = edge.from_node.id

            for edge in self.graph.edges:
                if dist[edge.from_node.id] != float("Inf") and dist[edge.from_node.id] +
edge.weight < dist[edge.to_node.id]:
                    return

            path = []
            node = dst.id
            while node != src.id:
                path.insert(0, node)
                node = self.paths.get(node, None)
                if node is None:
                    return None
            path.insert(0, node)

            print(f'{path[::-1]} = metric: {len(path)}')
            return path
    # Імпорти
```

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, arp, ethernet, ipv4, ipv6, ether_types
from ryu.topology.api import get_switch, get_link
from ryu.app.wsgi import ControllerBase
from ryu.topology import event
from collections import defaultdict
import random
import time
from dva import Graph, Edge, Node, LoadBalancerAlgorithm

# Клас Controller
class Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Controller, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.topology_api_app = self
        self.datapath_list = {}
        self.arp_table = {}
        self.switches = []
        self.hosts = {}
        self.multipath_group_ids = {}
        self.group_ids = []
        self.adjacency = defaultdict(dict)
```

```
        self.out_ports_map = {}
        self.graph = Graph()

    def add_ports_to_paths(self, paths, first_port, last_port):
        paths_p = []
for path in paths:
    p = {}
    in_port = first_port
    for s1, s2 in zip(path[:-1], path[1:]):
        out_port = self.adjacency[s1][s2]
        p[s1] = (in_port, out_port)
        in_port = self.adjacency[s2][s1]
    p[path[-1]] = (in_port, last_port)
    paths_p.append(p)
return paths_p
    def generate_openflow_gid(self):
       import random
def generate_unique_group_id(self):
    # Генеруємо випадковий ідентифікатор групи
    n = random.randint(0, 2**32)
    # Перевіряємо, чи він унікальний
    while n in self.group_ids:
        # Якщо не унікальний, генеруємо новий
        n = random.randint(0, 2**32)

    # Повертаємо унікальний ідентифікатор групи
    return n
    def update_flow_table(self, src, src_port, dst, dst_port, src_ip, dst_ip):
        from collections import defaultdict
```

```
def update_flow_table(self, src, src_port, dst, dst_port, src_ip, dst_ip):
    # Отримуємо шляхи від алгоритму балансування
    paths = [LoadBalancerAlgorithm(self.graph).call(Node(src), Node(dst))]

    # Додаємо порти до шляхів
    ports_to_paths = self.add_ports_to_paths(paths, src_port, dst_port)

    # Зберігаємо вихідний порт для src
    self.out_ports_map[src] = ports_to_paths[0][src][1]

    # Оновлюємо потокову таблицю для кожного вузла
    for node in set(sum(paths, [])):
        dp = self.datapath_list[node]
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser
        ports = defaultdict(list)
        actions = []
        i = 0

        # Для кожного шляху оновлюємо відповідність портів
        for path in ports_to_paths:
            if node in path:
                in_port = path[node][0]
                out_port = path[node][1]

                # Перевіряємо, чи порт вже відомий
                if (out_port, len(paths[i])) not in ports[in_port]:
                    ports[in_port].append((out_port, len(paths[i])))
                i += 1
```

```
# Додаємо правила в потокову таблицю для визначеного вузла
for in_port in ports:
    match_ip = ofp_parser.OFPMatch(
        eth_type=0x0800,
        ipv4_src=src_ip,
        ipv4_dst=dst_ip
    )
    match_arp = ofp_parser.OFPMatch(
        eth_type=0x0806,
        arp_spa=src_ip,
        arp_tpa=dst_ip
    )
    out_ports = ports[in_port]
     # Визначаємо дії для одного або декількох портів
    if len(out_ports) == 1:
        self.indicate_packet(
            out_ports[0][0],
            match_ip,
            match_arp,
            ofp_parser,
            dp
        )
    elif len(out_ports) > 1:
        paths_weight = sum([len(path) for path in paths])
        self.indicate_group(
            node,
            src,
            dst,
            match_ip,
            match_arp,
```

```
                out_ports,
                paths_weight,
                ofp_parser,
                ofp,
                dp
            )
def indicate_packet(self, port, match_ip, match_arp, ofp_parser, dp):
    # Визначаємо дії для пересилання пакета на вказаний порт
    actions = [ofp_parser.OFPActionOutput(port)]

    # Додаємо правило для IP-пакетів
    self.add_flow(dp, priority=32768, match=match_ip, actions=actions)
    # Додаємо правило для ARP-пакетів
    self.add_flow(dp, priority=1, match=match_arp, actions=actions)
def add_flow(self, dp, priority, match, actions):
    # Метод для додавання правила в потокову таблицю
    ofp = dp.ofproto
    ofp_parser = dp.ofproto_parser
    # Створюємо запит на додавання правила
    mod = ofp_parser.OFPFlowMod(
        datapath=dp,
        priority=priority,
        match=match,
        actions=actions,
        instructions=[],
        cookie=0,
        cookie_mask=0,
        table_id=0,
        command=ofp.OFPFC_ADD,
        idle_timeout=0,
```

```
            hard_timeout=0,
            flags=0
        )
        # Відправляємо запит на контролер
        dp.send_msg(mod)
    def indicate_group(self, node, src, dst, match_ip, match_arp, out_ports, paths_weight,
                       ofp_parser, ofp, dp):
        # Ідентифікатор групи OpenFlow
        ofpgroup_id = None
        group_new = False

        # Перевірка, чи група вже існує
        if (node, src, dst) not in self.multipath_group_ids:
            group_new = True
            self.multipath_group_ids[node, src, dst] = self.generate_openflow_gid()

        # Отримання ідентифікатора групи
        ofpgroup_id = self.multipath_group_ids[node, src, dst]

        # Створення списку відділень (buckets) для групи
        buckets = []
        for port, weight in out_ports:
            # Визначення ваги для відділення
            bucket_weight = int(round((1 - weight/paths_weight) * 10))

            # Визначення дії для відділення
            bucket_action = [ofp_parser.OFPActionOutput(port)]

            # Додавання відділення до списку
```

```
            buckets.append(
                ofp_parser.OFPBucket(
                    weight=bucket_weight,
                    watch_port=port,
                    watch_group=ofp.OFPG_ANY,
                    actions=bucket_action
                )
            )
        # Додавання чи модифікація групи в потоковій таблиці
        if group_new:
            req = ofp_parser.OFPGroupMod(
                dp, ofp.OFPGC_ADD, ofp.OFPGT_SELECT, ofpgroup_id, buckets
            )
            dp.send_msg(req)
        else:
            req = ofp_parser.OFPGroupMod(
                dp, ofp.OFPGC_MODIFY, ofp.OFPGT_SELECT, ofpgroup_id, buckets
            )
            dp.send_msg(req)
        # Створення дій для встановлення потокових правил
        actions = [ofp_parser.OFPActionGroup(ofpgroup_id)]
        # Додавання потокових правил для IP-пакетів
        self.add_flow(dp, priority=32768, match=match_ip, actions=actions)
        # Додавання потокових правил для ARP-пакетів
        self.add_flow(dp, priority=1, match=match_arp, actions=actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        # Отримання атрибутів OpenFlow протоколу та парсера
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
```

```
        # Створення інструкції для застосування дій
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]

        # Створення об'єкта потокового правила з вказаними параметрами
        if buffer_id:
            mod = parser.OFPFlowMod(
                datapath=datapath,
                buffer_id=buffer_id,
                priority=priority,
                match=match,
                instructions=inst
            )
        else:
            mod = parser.OFPFlowMod(
                datapath=datapath,
                priority=priority,
                match=match,
                instructions=inst
            )

        # Відправлення повідомлення про додавання потокового правила
        datapath.send_msg(mod)
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def _switch_features_handler(self, ev):
        # Отримання об'єкта даних комутатора та атрибутів OpenFlow протоколу
та парсера
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
```

```
        parser = datapath.ofproto_parser

        # Створення порожнього об'єкта потокового правила (відповідності)
        match = parser.OFPMatch()

        # Створення списку дій для виведення пакетів на контролер
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]

        # Додавання потокового правила на комутатор для виведення пакетів на
контролер
        self.add_flow(datapath, 0, match, actions)

        @set_ev_cls(event.EventLinkAdd, MAIN_DISPATCHER)
        def link_add_handler(self, ev):
            s1 = ev.link.src
            s2 = ev.link.dst
            n1 = Node(s1.dpid)
            n2 = Node(s2.dpid)
            self.graph.add_nodes(n1, n2)
            edge = Edge(n1, n2, 1)
            self.graph.add_edge(edge)
            self.adjacency[s1.dpid][s2.dpid] = s1.port_no
            self.adjacency[s2.dpid][s1.dpid] = s2.port_no

        @set_ev_cls(event.EventLinkDelete, MAIN_DISPATCHER)
        def link_delete_handler(self, ev):
            s1 = ev.link.src
            s2 = ev.link.dst
            # Exception handling if switch already deleted
```

```
try:
    del self.adjacency[s1.dpid][s2.dpid]
    del self.adjacency[s2.dpid][s1.dpid]
except KeyError:
    pass
```

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНЫВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

ДИПЛОМНА РОБОТА

На ступінь вищої освіти магістр

Із спеціальності 122 Комп ’ютерні технології

Дослідження технологій балансування корпоративних мереж

Виконав: студент 6 курсу, групи КНДМ-61

Єник Віталій Сергійлвич

Керівник: к. т. н., Сєрих Сергій Олександрович

Київ - 2023

ЗАГАЛЬНІ ХАРАКТЕРИСТИКИ ДИПЛОМНОЇ РОБОТИ

Тема:	Дослідження технологій балансування корпоративних мереж
Мета дослідження:	Дослідити та продемонструвати методи балансування трафіку
Наукове завдання:	Розробка моделі збалансованої системи
Об’єкт дослідження:	Процес запровадження балансування трафіку
Предмет дослідження:	Локальна мережа підприємства

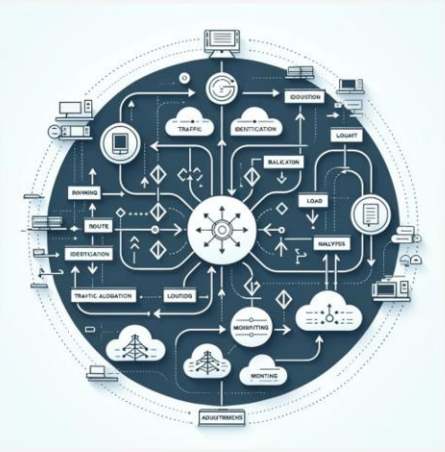
Методибалансуваннямереж

1. **Балансування навантаження на рівні маршрутизації**
2. **Балансування навантаження на рівні серверів**
3. **Балансування на рівні Domain Name System**
4. **Балансування за допомогоюContent Delivery Network**
5. **Балансування на рівні мережіSDN**

Балансування навантаження на рівні маршрутизації:

Суть методу: Рівномірний розподіл трафіку між різними шляхами для запобігання перевантаженню окремих маршрутів.

Принцип дії: Система визначає доступні маршрути та автоматично розподіляє трафік, підлаштовуючись під зміни в навантаженні.



Балансування навантаження на рівні серверів:

5

Суть методу: Рівномірне розподілення запитів між серверами для забезпечення однакового навантаження на кожному сервері.

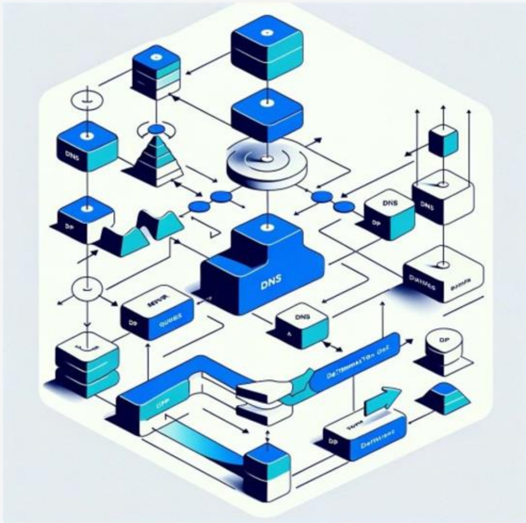
Принцип дії: Використання спеціальних балансувальників навантаження, які моніторять роботу серверів і розподіляють запити від користувачів.

Балансування на рівні DNS:

6

Суть методу: Визначення IP-адрес сервера на основі DNS-запитів і розподіл трафіку між різними серверами.

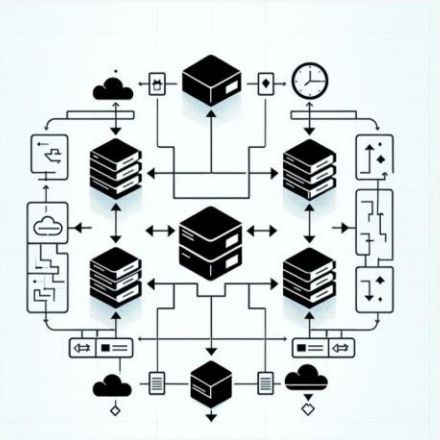
Принцип дії: DNS-сервер автоматично вибирає IP-адрес сервера, що має менше навантаження, і вказує його як відповідь на запит.



Балансування за допомогою CDN:

Суть методу: Використання розподіленої мережі серверів (Content Delivery Network) для швидкого та ефективного обслуговування контенту.

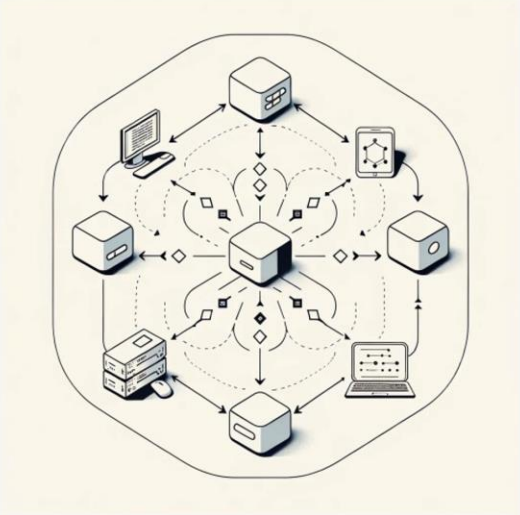
Принцип дії: Контент копіюється на сервери CDN, що знаходяться ближче до кінцевих користувачів, забезпечуючи швидку доставку та розподіл трафіку.



Балансування на рівні мережі SDN:

Суть методу: Використання програмно-визначених мереж для централізованого керування трафіком та оптимізації маршрутів.

Принцип дії: Контролер SDN приймає рішення щодо оптимального маршруту для кожного пакету, забезпечуючи розподіл навантаження.



Порівняння методів

Параметр/Метод	Балансування на рівні серверів	DNS-балансування	Балансування на рівні маршрутизації	CDN-балансування	SDN-балансування
Ефективність	★★★	★★	★★★★	★★★	★★★★★
Масштабованість	★★★★	★★★	★★★	★★★★	★★★★★
Гнучкість	★★★★	★★	★★★	★★★	★★★★★
Контроль	★★★	★	★★★	★★	★★★★★
Вартість	★★★	★★★★	★★★	★★★	★★★

★ - низький рівень, ★★★★★ - високий рівень

Балансування за допомогою SDN

Контролер SDN

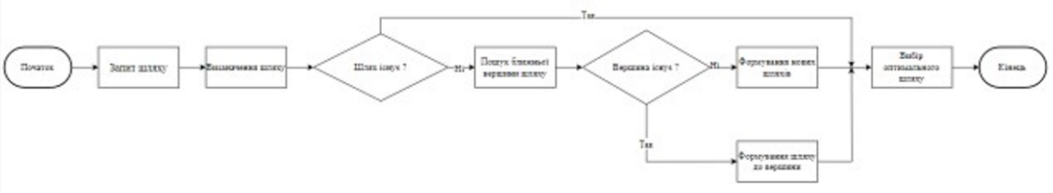
Південний Інтерфейс (Southbound Interface)

Північний Інтерфейс (Northbound Interface)

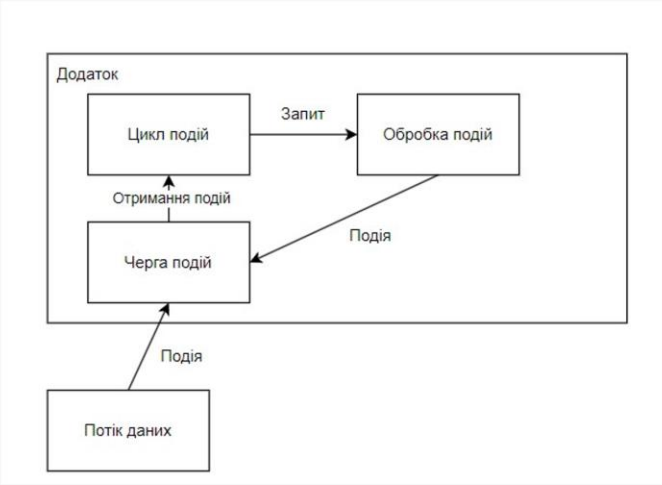
Мережеві пристрої (Комутатори, Маршрутизатори)

Динамічна маршрутизація та управління трафіком

Алгоритм конструювання трафіку



Модель Ryu застосунку



Висновки

Балансування на рівні DNS, маршрутизації , CDN та SDN - кожен з методів має свої унікальні переваги та застосування.
SDN надає виняткову гнучкість, масштабованість та контроль, що робить його ідеальним для сучасних великих та складних мережевих середовищ.
Централізоване управління , яке пропонує SDN, сприяє ефективній координації та оптимізації мережевих ресурсів.
Продовжуючи розвиток, технології балансування, особливо SDN, будуть відігравати ключову роль у підтримці зростаючих потреб у мережевій пропускній здатності та гнучкості.
Підприємства повинні розглядати впровадження технологій SDN для підвищення ефективності та адаптивності своїх мережевих інфраструктур.