**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«Київський політехнічний інститут імені Ігоря Сікорського»**
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»
УДК ________________

«До захисту допущено»
Завідувач кафедри
__________ *Стіренко С.Г.*
(підпис) (ініціали, прізвище)
"____" _________ 2020 р.

# Магістерська дисертація

зі спеціальності: *123. Комп'ютерна інженерія*
(код та назва напряму підготовки або спеціальності)

Спеціалізація: *123. Комп'ютерні системи та мережі*

на тему: Спосіб балансування навантаження в масштабних програмно-конфігурованих мережах

Виконав (-ла): студент (-ка) 2 курсу, групи ІВ-81мн
(шифр групи)

Кузнєцов Ярослав Іванович
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник ст. викладач, к.т.н. Габінет А.В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант ________________
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент ________________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент ________________
(підпис)

Київ – 2020 року

**Національний технічний університет України**
**«Київський політехнічний інститут імені Ігоря Сікорського»**

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Спеціалізація *123. Комп'ютерні системи та мережі*

ЗАТВЕРДЖУЮ
Завідувач кафедри
*Стіренко С.Г.*
(підпис) (ініціали, прізвище)
« » ________ 2020 р.

## ЗАВДАННЯ
**на магістерську дисертацію студенту**

Кузнєцову Ярославу Івановичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Спосіб балансування навантаження в масштабних програмно-конфігурованих мережах

Науковий керівник дисертації Габінет Артем Вікторович, кандидат технічних наук, старший викладач
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «***24***» ***03*** ***2020*** р. № ***910-с***

2. Строк подання студентом дисертації ____________________

3. Об'єкт дослідження Процес пошуку оптимального шляху між вузлами в програмно-конфігурованих мережах.

4. Предмет дослідження Методи маршрутизації та визначення оптимальності маршруту в програмно-конфігурованих мережах.

5. Перелік завдань, які потрібно розробити: 1) Дослідження структури та принципів побудови програмно-конфігурованих мереж;
2) Дослідження методів масштабування програмно-конфігурованих мереж;
3) Дослідження методів забезпечення якості обслуговування в класичних та програмно-конфігурованих мережах;
4) Дослідження методів маршрутизації в програмно-конфігурованих мережах;

5) Дослідження методів рішення задач багатокритеріальної оптимізації;
6) Розробка методу балансування трафіку в програмно-конфігурованих мережах;
7) Розробка програмного модулю на основі розробленого методу;
8) Ілюстрація роботи моделі та аналіз отриманих результатів.

6. Консультанти розділів дисертації:

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Основна частина | доктор технічних наук Кулаков Ю.О. | | |
| | | | |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання ______________________________

Календарний план

| № з/п | Назва етапів виконання магістерської дисертації | Строк виконання етапів дисертації | Примітка |
|---|---|---|---|
| 1 | Вивчення літератури | 10.11.2019 | |
| 2 | Складання і узгодження технічного завдання | 4.12.2019 | |
| 3 | Написання вступної частини та огляд рішень | 26.01.2020 | |
| 4 | Написання основної частини | 18.03.2020 | |
| 5 | Виправлення помилок | 25.04.2020 | |
| 6 | Оформлення документації | 12.05.2020 | |
| | | | |

Студент ________________ Я.І. Кузнєцов
(підпис) (ініціали, прізвище)

Науковий керівник дисертації ________________ А.В. Габінет
(підпис) (ініціали, прізвище)

# РЕФЕРАТ

## на магістерську дисертацію

виконану на тему: Спосіб балансування навантаження в масштабних програмно-конфігурованих мережах

студентом: Кузнєцовим Ярославом Івановичем

Робота складається із вступу, трьох розділів та одного додатку. Загальний обсяг роботи: 69 аркуші основного тексту, 41 ілюстрація, 6 таблиць. При підготовці використовувалася література з 38 різних джерел.

**Актуальність.** Дедалі більшу популярність набуває стрімінгові сервіси, зростає кількість мережевих сервісів та абонентів мережі, що потребують забезпечення якості обслуговування. Традиційні мережі наближаються до своєї межі ефективності. Альтернативою традиційним мережам є централізована архітектура SDN.

Ідея технології SDN не є новою, проте активне впровадження технології припадає лише на останні 15 років. Однією із важливих задач при організації мережі є задача балансування трафіку, що включає в собі маршрутизацію та забезпечення якості обслуговування. Використання нової технології вимагає розробки нових алгоритмів та протоколів, або адаптації традиційних, задача балансування трафіку є актуальною.

**Мета і завдання дослідження.** Метою магістерської роботи є розроблення методу, що дозволить маршрутизувати трафік по оптимальним шляхам, уникаючи високої затримки та забезпечення більш рівномірного завантаження мережі шляхом балансування трафіку, з використанням можливостей програмно-конфігурованих мереж.

Для досягнення мети дослідження поставлено і вирішено такі завдання:

- дослідження структури та принципів побудови програмно-конфігурованих мереж;
- дослідження методів масштабування програмно-конфігурованих мереж;

- дослідження методів забезпечення якості обслуговування в класичних та програмно-конфігурованих мережах;
- Дослідження методів маршрутизації в програмно-конфігурованих мережах;
- Дослідження методів рішення задач багатокритеріальної оптимізації;
- Розробка методу балансування трафіку в програмно-конфігурованих мережах;
- Розробка програмного модулю на основі розробленого методу;
- Ілюстрація роботи моделі та аналіз отриманих результатів.

**Об'єкт дослідження –** процес пошуку оптимального шляху між вузлами в програмно-конфігурованих мережах.

**Предмет дослідження** – методи маршрутизації та визначення оптимальності маршруту в програмно-конфігурованих мережах.

**Методи досліджень.** Для досягнення поставлених в магістерській роботі задач, використано методи теорії графів, методи моделювання, методи вирішення задач багатокритеріальної організації.

Проведене дослідження дає можливість використання розробленої моделі в SDN мережах в якості застосунку контролера та виконувати емуляцію роботи мережі для прогнозування трафіку та завантаженості компонентів мережі.

**Особистий внесок здобувача.** Магістерське дослідження є самостійно виконаною роботою, в якій відображено особистий авторський підхід та особисто отримані теоретичні та прикладні результати, що відносяться до вирішення задачі маршрутизації та контролю трафіку в SDN мережах. Формулювання мети та завдань дослідження проводилось спільно з науковим керівником.

**Практична цінність.** Отримані результати можуть використовуватися у майбутніх дослідженнях за напрямками:

- вдосконалення методів маршрутизації;
- аналіз та прогнозування трафіку в SDN;
- балансування навантаження в SDN мережах.

**Ключові слова**

Програмно-конфігуровані мережі, конструювання трафіку, маршрутизація, балансування навантаження в SDN, Quality of Service в SDN, багатокритеріальна оптимізація.

**ABSTRACT**

**on the master thesis**

executed on a subject: A method of load balancing in large-scale software defined networks

by the student: Kuznietsov Yaroslav Ivanovych

Current work consists of receipt, three sections and one application. Total amount of work: 69 pages of the main text, 41 illustrations, 6 tables. By preparation a literature from 38 different sources was used.

**Topic Relevance.** The increasing popularity acquires streaming services, the number of network services and subscribers of network grows that need ensuring quality of service. Traditional networks come nearer to the limit of efficiency. An alternative to traditional networks is the centralized architecture of SDN.

The idea of SDN technology is not new, however active implementation of technology falls only on the last 15 years. The problem of balancing of traffic is one of important tasks at the organization of network that includes in itself routing and ensuring quality of service. Use of new technology demands development of new algorithms and protocols, or adaptation traditional, the problem of balancing of traffic is relevant.

**Research goal.** The research goal of the master's thesis is development of a method that will allow to route traffic on optimal ways, avoiding a high delay and ensuring more uniform loading of network by balancing of traffic, with use of opportunities of software defined networks.

For achievement of the goal of a research it is put and solved the following tasks:

- Research of structure and the principles of construction software defined networks;
- Researches of methods scaling software defined networks;
- Researches of methods of ensuring quality of service in classical and software defined networks;

- Researches of methods of routing in software defined networks;
- Researches of methods of solving problems of multicriterial optimization;
- Development of a method of balancing of traffic in software defined networks;
- Development of the program module on the basis of the developed method;
- Illustration of work of model and the analysis of the received results.

**Object of research –** Process of search of an optimal way between nodes in software defined networks.

**Subject of research** – Methods of routing and determination of optimality of a route in software defined networks.

**Methods of research.** For achievement of the tasks set in the master's thesis, it is used methods of the theory of graphs, modeling methods, methods of the solution of tasks of the multicriteria organization.

The conducted research gives the chance of use of the developed model in SDN networks as an application of the controller and to carry out emulation of network functioning for forecasting of traffic and load of components of network.

**Scientific contribution.** The master research is independently done work in which it is reflected personal author's approach and personally received theoretical and applied results relating to the solution of a problem of routing and control of traffic in SDN networks. A formulation of the purpose and tasks the research was conducted together with the research supervisor.

**Practical value of obtained results.** The received results can be used in future researches on the directions:

- to improvement of methods of routing;
- the analysis and forecasting of traffic in SDN;
- balancing of loading in SDN networks.

**Keywords**

Software defined networks, traffic designing, routing, load balancing in SDN, Quality of Service in SDN, multicriterial optimization.

# ПОЯСНЮВАЛЬНА ЗАПИСКА

## до магістерської дисертації

на тему: «Спосіб балансування навантаження в масштабних програмно-конфігурованих мережах»

Київ — 2020 року

**Зміт**

Перелік скорочень 3

ВСТУП 4

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД SDN МЕРЕЖ ........ 5

1. Архітектура SDN ........ 5

2. Протокол OpenFlow ........ 6

2.1. Компоненти комутатора на базі OpenFlow ........ 6

2.1.1. Flow table ........ 7

2.1.2. Group table ........ 9

2.1.3. Meter table ........ 9

3. Основні проблеми архітектури SDN ........ 10

3.1. Ефективність та надійність одного SDN контролера ........ 10

3.2. Масштабування SDN мереж ........ 11

3.3. Розподілення трафіку в мережі ........ 13

4. Моделі забезпечення QoS ........ 14

4.1. Best Effort ........ 15

4.2. IntServ ........ 15

4.3. DiffServ ........ 17

5. QoS в SDN мережах ........ 31

Висновки до розділу ........ 34

РОЗДІЛ 2. БАЛАНСУВАННЯ ТРАФІКУ ........ 36

1. Алгоритм пошуку найкоротшого шляху ........ 36

2. Алгоритми маршрутизації QoS ........ 37

3. Алгоритм балансування на основі багатьох метрик ........ 39

Висновки до розділу ........ 53

РОЗДІЛ 3. РОЗРОБКА МОДУЛЮ БАЛАНСУВАННЯ ТРАФІКУ В SDN ........ 55

1. Огляд інструментів розробки для SDN мереж ........ 55

2. Архітектура модулю балансування трафіку в SDN мережі ........ 58

3. Моделювання методу балансування трафіку ........ 59

Висновки до розділу ........ 67

ВИСНОВКИ ........ 68

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ........ 70

Додаток А ........................................................................................................ 73

## Перелік скорочень

| | |
|---|---|
| SDN | Software Defined Network (укр. Програмно-конфігурована мережа) |
| QoS | Quality of Service (укр. Якість обслуговування) |
| IntServ | Integrated Service (укр. Інтегроване обслуговування) |
| DiffServ | Differentiated service (укр. Диференційоване обслуговування) |
| FIFO | First In, First Out (укр. Першим прийшло, першим пішло) |
| DABS | Delay Aided Bandwidth Search (укр. Пошук полоси пропускання за затримкою) |
| DEBF | Dual Extended Bellman-Ford (укр. Подвійний розширений Белман-Форд) |
| MEFPA | Multiconstrained Energy Function based Pre-computation Algorithm (укр. Алгоритм попереднього обчислення на основі енегретичної функції з багатьма обмеженнями) |
| lldp | Link Layer Discovery Protocol |

# ВСТУП

Інтернет перетворився на величезну мережу, що повинна забезпечувати широкий спектр послуг. Надання мережевих послуг включає такі можливості, як забезпечення переадресації, маршрутизації за різними критеріями, якість обслуговування (QoS), безпеки та менеджменту. Так, як зростає не лише кількість послуг, що повинні забеспечуватися мережею, а і їх складність, проектування, доставка та експлуатація рішень стало складною справою, що почало вимагати знань різних рівнів. Ця ситуація погіршується різноманітністю аппаратних рішень та мережевих протоколів.[1] Для зменшення складності обслуговування, контролю та конфігурації мережі, а також для зменшення вартості пристроїв було запропоновано розділити рівень даних та рівень контролю. Було розроблено парадигму програмованих мереж, що отримала назву SDN.[2] SDN забезпечує можливості програмних додатків динамічно програмувати окремі мережеві пристрої і, отже, контролювати поведінку мережі в цілому. Дана парадигма дає можливість реалізації нових ідей для управлінням мережами та трафіком, а також спрощує життя системним адміністраторам. Фактором, що не дає можливості використовувати концепції SDN є той факт, що дослідження в області побудови SDN мереж, їх росширення, забезпечення надійності та управління трафіком ще триває.

Саме тому тема магістерської дисертації присвячена балансуванню трафіку в SDN мережах. Дослідження та вирішення проблеми балансування трафіку в SDN мережах дасть можливість якнайшвидше запровадити SDN не лише в датацентри компаній, що займаються дослідженням та розробкою аппаратних рішень для SDN, а і в корпоративні та локальні мережі. Це відкриє нові можливості в області надавання послуг та поліпшенні їх якості.

# РОЗДІЛ 1
# АНАЛІТИЧНИЙ ОГЛЯД SDN МЕРЕЖ

## 1.1 Архітектура SDN

В даній роботі буде розглядатися архітектура SDN, за визначенням, що було дано Open Networking Foundation, тому що його прийняли і використовують найбільші гравці в мережевій індустрії. В архітектурі SDN рівні керування трафіком та передачі данних розділені. [3]

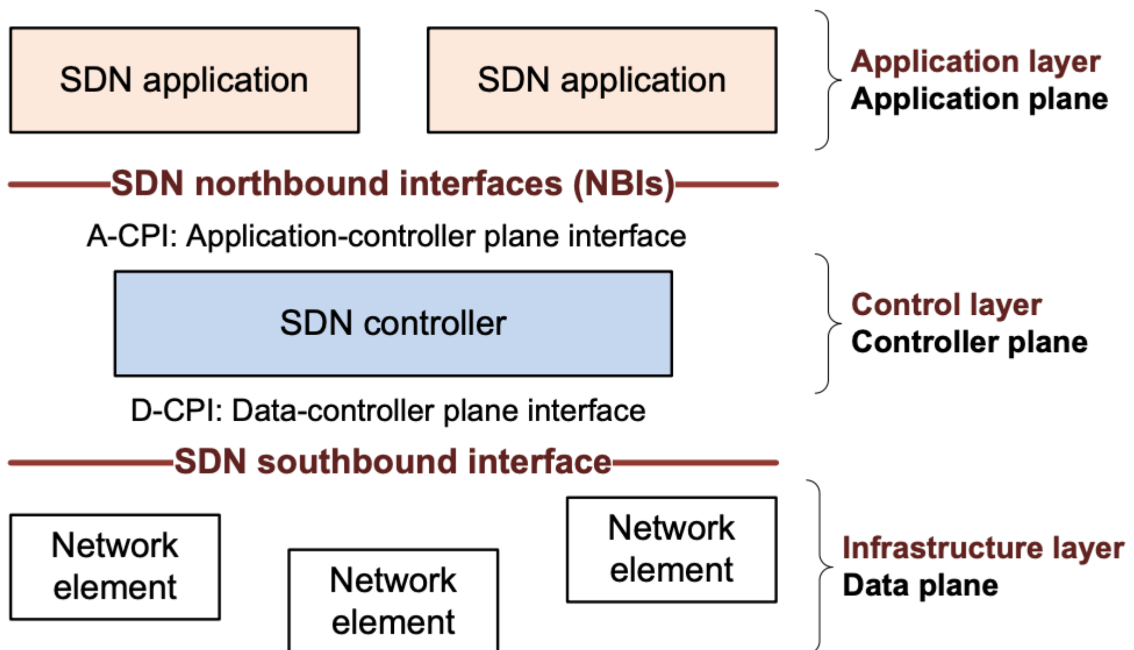



Рис. 1.1. Архітектура SDN мережі

На рис.1.1 зображено структуру SDN, що складається з трьох рівнів. Найнижчий рівень називається рівнем інфраструктури, також називається площиною даних. Він містить мережеві елементи пересилки. Цей рівень займається пересилкою даних, збором статистики та моніторингом локальної інформації. Наступний рівень називається рівнем контролю, або управляючою площиною. Даний рівень відповідає за програмування та управління рівнем даних. Цей рівень використовує інформацію, яка надається рівнем даних і визначає роботу мережі та рівня передачі. Для комунікації цих двох рівнів використовується стандартизовний інтерфейс, що називається southbound interface (укр. південний інтерфейс). Останній рівень називається рівнем застосунків. Цей рівень містить застосунки, які можуть забезпечити нові

можливості для мережі, такі як інформаційна безпека, адміністративні інструменти, а також можуть бути допоміжними засобами для рівня контролю. Цей рівень може отримати глобальну картину мережі та надати рішення для рівня контролю, базуючись на інформації про мережу. Для комунікації між рівнем контролю та рівнем застосунків використовується стандартизований інтерфейс, що називається northbound inteface (укр. північний інтерфейс).

Існує багато протоколів, що забезпечують комунікацію рівнів контролю та даних, наприклад, ForCES (Forwarding and Control Element Separation) [4,5,8], SoftRouter [6], Locator/ID Separation Protocol (LISP) [7], OpenFlow[8]. Але в даній роботі буде розглянуто лише OpenFlow протокол в якості Southbound Interface, тому що це протокол, який найбільш широко застосований.

### 1.2 Протокол OpenFlow

Архітектура SDN передбачає активний обмін інформацією між рівнем даних та рівнем контролю. При цьому збільшується кількість службового трафіку між контроллером і вузлами комутації та зменшується кількість службового трафіку між вузлами комутації безпосередньо.

Для вирішення даної задачі було створено OpenFlow протокол. Основною ідеєю OpenFlow є винесення рівня управління з маршрутизатору на контроллер, що є частиною рівня контролю. В зв'язку з тим, що маршрутизатор при цьому змінює свою зону відповідальності і відповідає лише за передачу пакетів, і більше не вирішує, в якому напрямку, вартість такого маршрутизатору знижується. В свою чергу OpenFlow забезпечує підключення комутаторів до контролеру та оновлення інформації маршрутизації на комутаторах.

#### 1.2.1 Компоненти комутатора на базі OpenFlow

Логічний комутатор OpenFlow складається з однієї або декількох таблиць потоків і таблиці груп, які виконують пошук і пересилку пакетів, і одного або декількох каналів OpenFlow на зовнішній контролер. Комутатор взаємодіє з контролером, і контролер управляє комутатором по протоколу OpenFlow. [9]

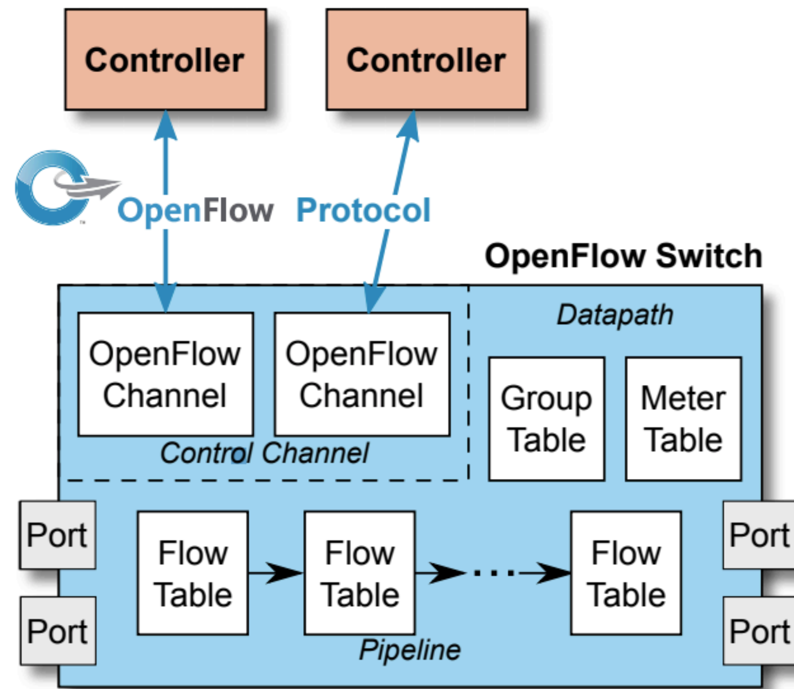



Рис. 1.2 Головні компоненти OpenFlow комутатора [9]

OpenFlow надає наступні компоненти: таблиці потоків (flow tables), групові таблиці (group tables), таблиці метрик (meter tables), лічильники (counters). Ці інструменти дають можливість управляти потоками та розподіляти трафік за певними правилами в залежності від швидкості та інтенсивності надходження пакетів. Детальний розгляд даних компонентів описаний в наступних розділах.

### 1.2.1.1 Flow table

Таблиці потоків (flow tables) складаються з записів потоків (flow entries). Кожен з таких записів містить поля співпадання, пріорітету, лічильника, інструкції для виконання, часу закінчення терміну валідності потоку, cookie, прапорів. Запис таблиці потоку ідентифікується по полях відповідності та пріоритету: поля відповідності і пріоритет, взяті разом, ідентифікують унікальний запис потоку в певній таблиці потоку. Запис потоку, який містить знаки підстановки для всіх полів (всі поля опущені) і має пріоритет, рівний 0, називається записом потоку з помилками таблиці. Інструкція введення потоку

може містити дії, які повинні бути виконані над пакетом в деякій точці конвеєра (див рис 1.3). Дія set-field може вказувати деякі поля заголовка для перезапису.

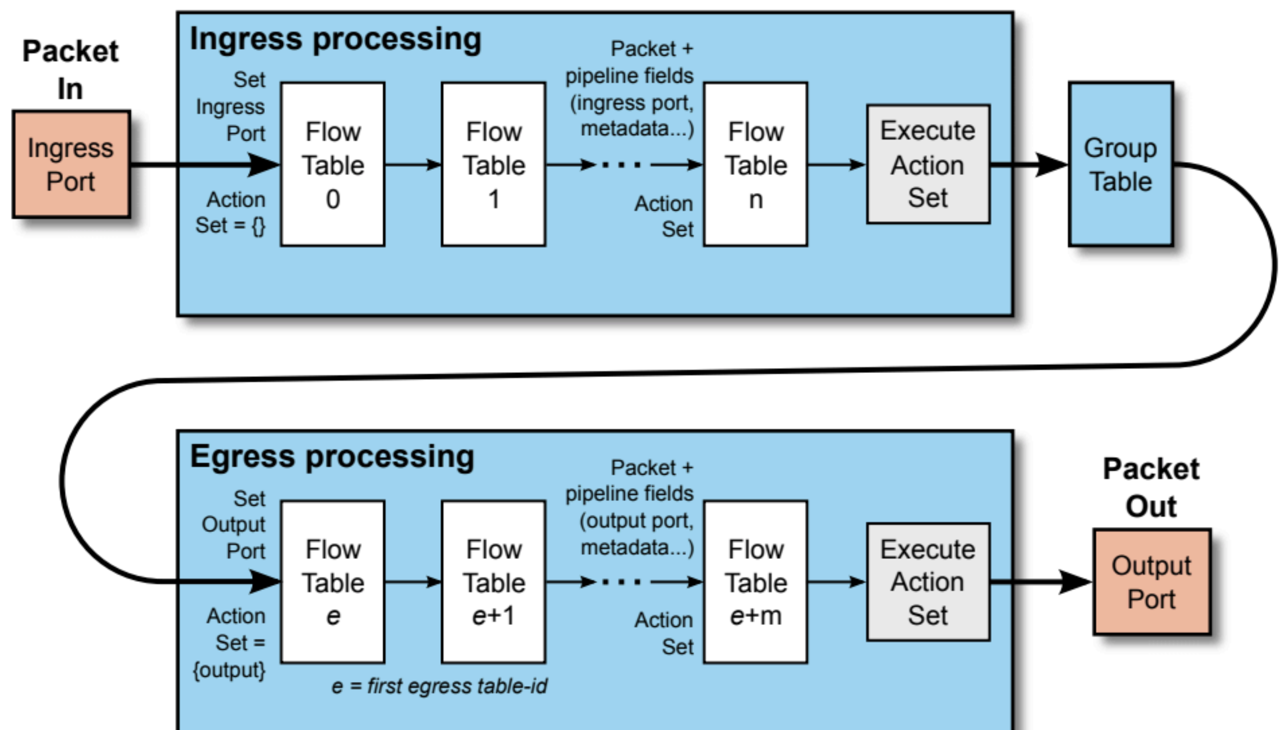



Рис. 1.3. Проходження пакетів через конвеєр обробки. [9]

Поля заголовка пакета витягуються з пакету, і витягуються поля конвеєра пакетів. Поля заголовка пакета, що використовуються для пошуку в таблиці, залежать від типу пакета і зазвичай включають різні поля заголовка протоколу, такі як адреса джерела Ethernet або адреса призначення IPv4. На додаток до заголовків пакетів, також можуть виконуватися зіставлення з вхідним портом, полем метаданих та іншими полями конвеєра.

Кожна таблиця потоку повинна підтримувати запис потоку, що пропускає таблицю, щоб обробляти пропущені таблиці. Запис потоку з помилками в таблиці визначає, як обробляти пакети, несумісні з іншими записами потоку в таблиці потоків, і може, наприклад, посилати пакети контролера, відкидати пакети або направляти пакети в наступну таблицю. [9]

### 1.2.1.2 Group table

Групові таблиці складаються з групових записів. Здібність потокових записів вказувати на группи дає можливість забезпечити додатковими методами переадресації. Кожен запис групової таблиці має наступні поля: груповий ідентифікатор, тип групи, лічильники та список дій над пакетом. Запис групи може складатися з нуля або декількох блоків дій, за винятком групи непрямого типу, яка завжди має один блок дій. Група без блоків дій відкидає пакет.

### 1.2.1.3 Meter table

Таблиці метрик складається з записів метрик, що визначають метрики кожного потоку. Поточні метрики дають можливість встановити обмеження швидкості, просту операцію QoS, що обмежує набір потоків для обраної смуги пропускання.  Також поточні метрики дають можливість організувати більш складну обробку потоку з врахуванням QoS, наприклад на основі вимірювання, яке може класифікувати набір пакетів по декількох категоріях на основі його швидкості.

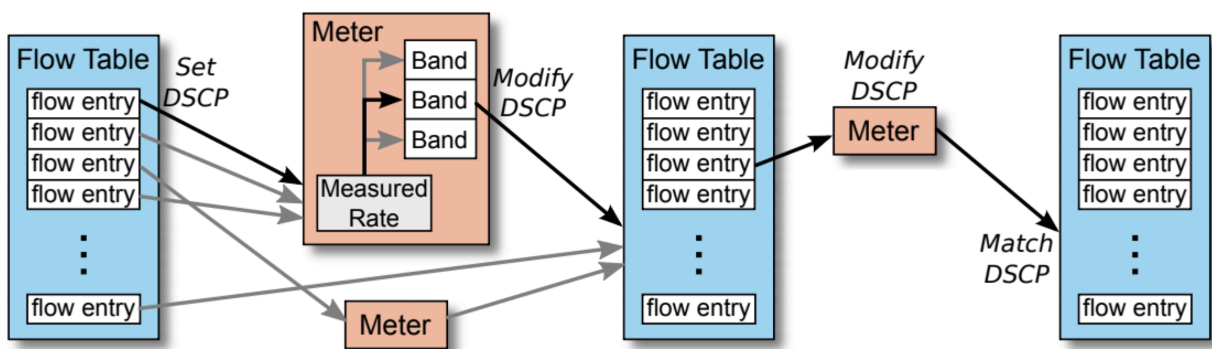



Рис. 1.4. Таблиці метрик. [9]

Пакети можуть проходити через декілька метрик при використанні метрик в послідовних таблицях потоків, що при співпаданні перенаправлять в таблицю метрик (див. 1.4)

Кожна полоса (meter band) складається з типу полоси, цільової швидкості для полоси, гранулярності полоси, лічильників та dscp маркування. Спрощена модель полягає в тому, що для кожного пакета вимірювач застосовує діапазон

метрики з найвищою цільової швидкістю, яка нижче поточної вимірюваної швидкості. [9]

## 1.3 Основні проблеми архітектури SDN

Архітектура SDN мереж дає можливість гнучко конфігурувати та управляти мережею без фізичної зміни топології та знижує вартість обладнання через винесення логіки обробки даних та прийняття рішень в контролер. Але такий підхід унеможливлює використання багатьох наробок людства, що знайшли своє застосування в класичних мережах. Також таких підхід додає ряд нових проблем, що потребують вирішення.

### 1.3.1 Ефективність та надійність одного SDN контролера

В SDN мережі логічно централізований контролер організує розподілений набір комутаторів для надання мережевих послуг високого рівня кінцевим серверним додаткам. Контролер може переналаштувати комутатори для адаптації до вимог трафіку і відмовам обладнання, що спостерігається через події. Наприклад, контролер SDN приймає події, що стосуються змін топології, статистики трафіку та пакетів, та відповідає командами, що встановлюють нові правила переадресації на комутаторах. Саме глобальна видимість подій в мережі дозволяє легко реалізувати такі можливості, як балансування навантаження, оптимальний розподіл трафіку та безпека.

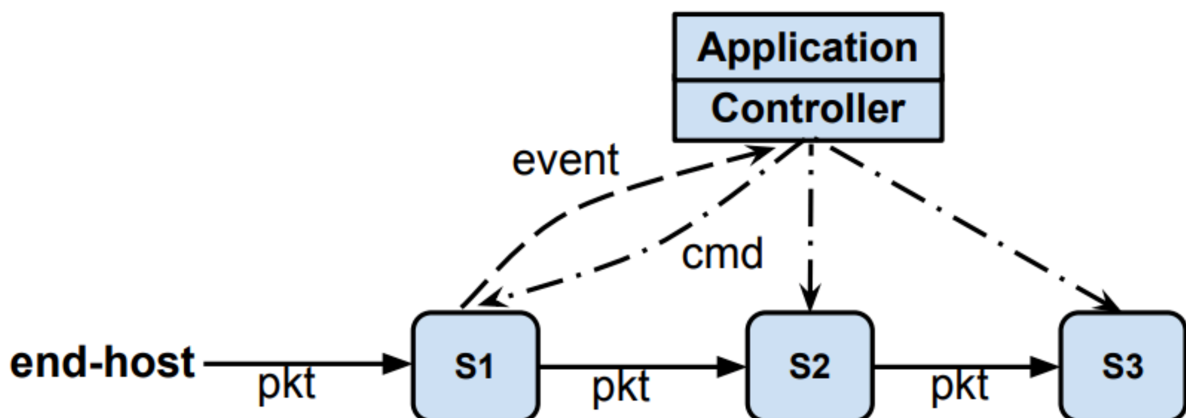



Рис. 1.5. Модель SDN системи. [10]

Не зважаючи на концептуальну простоту централізованого управління, один контролер легко перетворюється на точку падіння, що призводить до збоїв в мережі та неправильної обробки пакетів[10]. Якщо розглядати контролери на базі OpenFlow, OpenFlow SDN є потоковим рішенням, де потенційно кожен перший пакет повинен проконсультуватися з контролером для прийняття рішення про пересилання. Це не масштабується, якщо кількість вузлів збільшується, оскільки кількість унікальних потоків може рости нелінійно. Внаслідок цього кількість правил переадресації, які необхідно змінити для відновлення після збою, може бути дуже великим і продовжувати час відновлення. [11] З цього випливає висновок, що для забезпечення стабільної роботи SDN мережі необхідно запровадити механізм, що дозволить масштабувати мережу, додаючи контролери та забезпечуючи спроможність системи продовжувати свою роботу навіть при виведенні зі строю декількох контролерів.

### 1.3.2 Масштабування SDN мереж

Для вирішення задачі масштабування SDN мереж було запропоновано багато різноманітних рішень. В даному розділі буде розглянуто базові мултиконтроллерні архітектури.

**Рівнозначні контролери**

На рис. 1.6 в топології мережі є два контролера, кожен з яких керує частиною

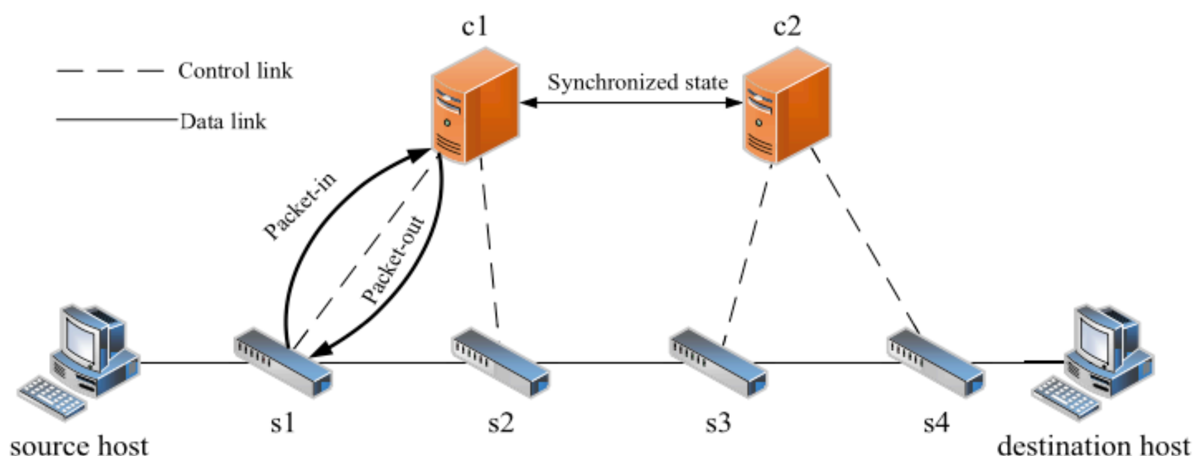



Рис. 1.6. Модель SDN системи. [12]

мережі. У цьому сценарії (c1) та (c2) спільно використовують одну і ту ж логіку логічно централізованим чином, так що коли нові пакети надходять на (s1), обидва (c1) та (c2) можуть безпосередньо встановлювати шлях пересилання у всіх відповідних комутаторах. Таким чином, це може ефективно зменшити тиск обробки потоку в одному контролері. Тим часом, ці два контролера є резервними, це дасть можливість уникнути проблеми однієї точки падіння. Але дана архітектура

**Дві базові мультиконтролерні архітектури**

При розміщенні багатьох контролерів ключовим моментом являється проєктування архітектури мультиконтроллерної системи. Базовою мультиконтроллерною архітектурою можна поділити на два види: однорівнева та ієрархічна. Однорівнева система поділена на багадо доменів, кожен домен управляється одним контролером, який має інформацію лише про локальну мережу. Контроллери комунікують один з одним за допомогою східного та західного інтерфесу (eastbound and westbound interface), таким чином вони можуть отримати глобальну карту мережі. Типова плоска архітектура зображена на рис.1.7. Представниками даної архітектури є Hyperflow [13] и Onix [14]

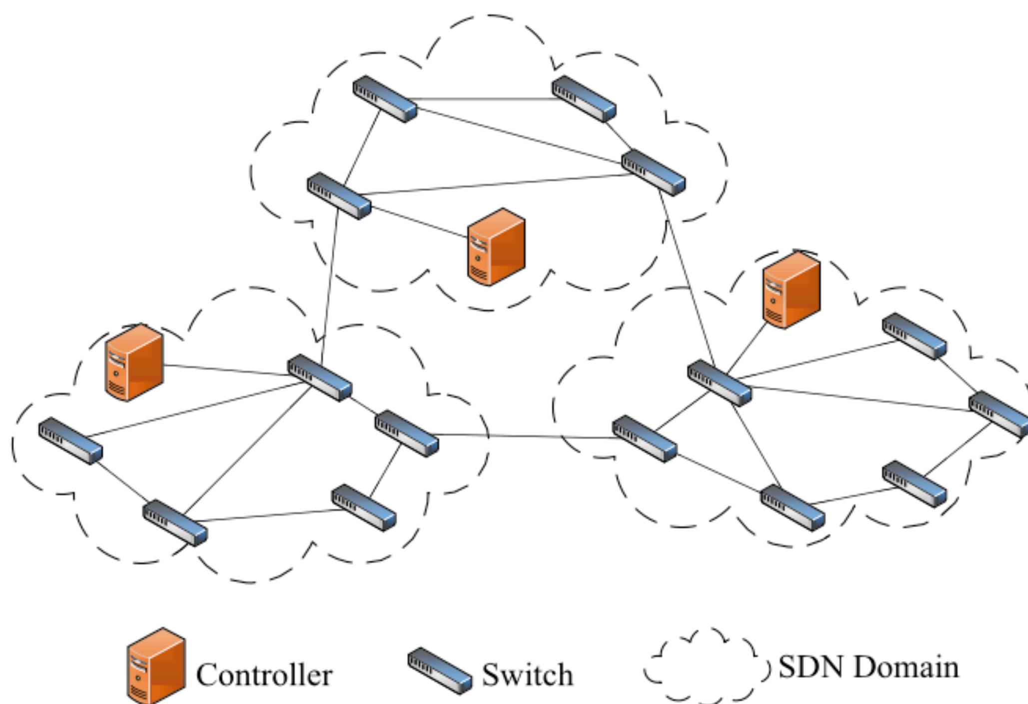



Рис. 1.7. Плоска архитектура мультиконтролерної системи. [12]

Ієрархічна архітектура зазвичай використовує дворівневі контролери: доменні контролери, що управляють комутаторами в своїх локальних доменах та запускає локальні управляючі додатки та кореневий контрол, що управляє контролерами домену та підтримує глобальну карту мережі. До таких структур належить Kandoo[15]. В Kandoo кореневий контроллер комунікує з доменними контроллерами, щоб отримати інформацію про домен, при цьому доменні контролери не комунікують один з одним. На рис.1.8 зображено базову ієрархічну архітектуру.

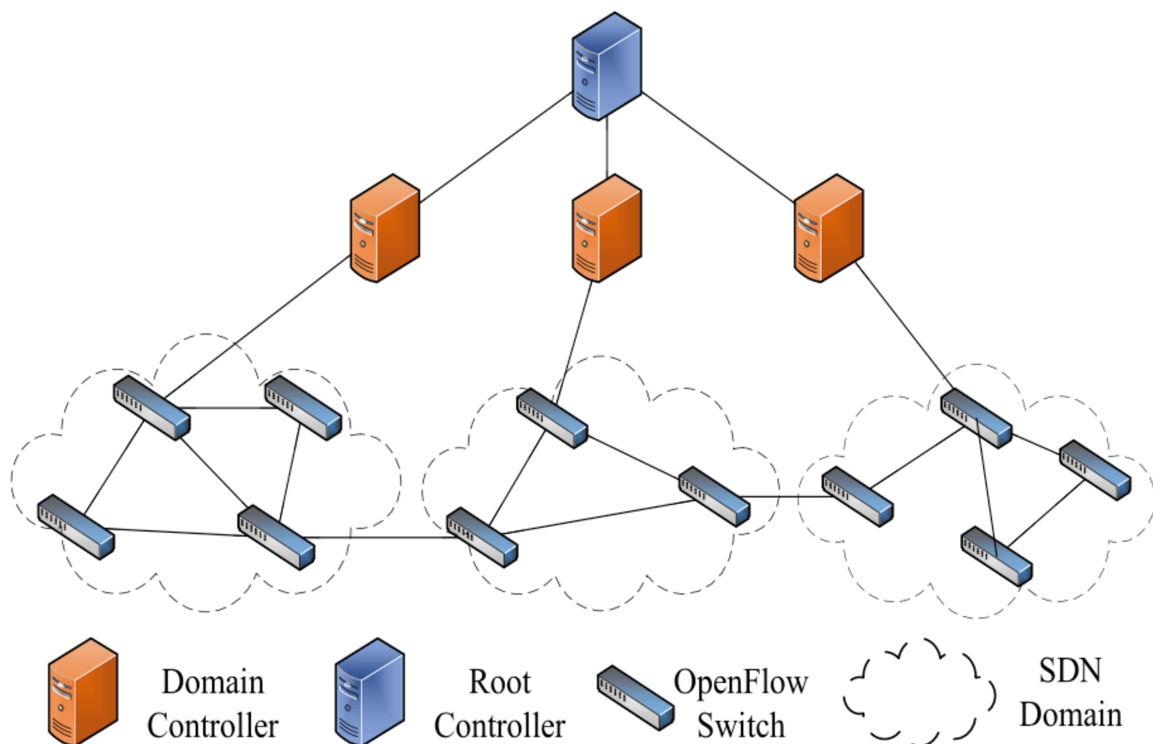



Рис. 1.8. Ієрархічна архитектура мультиконтролерної системи. [12]

### 1.3.3 Розподілення трафіку в мережі

На сьогодні було створено неймовірну кількість пропрієтарних та відкритих рішень для вирішення задачі маршрутизації в мережах. Ці рішення знайшли своє втілення в таких протоколах, як OSPF, EIGRP, RIP, IGRP, IS-IS і т.д. Але більшість з них працювала за умови, що кожен маршрутизатор повинен приймати рішення самостійно відносно того, куди перенаправляти вхідні пакети. Та разом з приходом нової концепції програмно-конфігурованих мереж, які були покликані розділити рівень даних та рівень контролю і покласти

відповідальність за прийняття рішень на контроллер, стало необхідно винайти нові рішення, що допоможуть розподіляти трафік мережею без надмірного навантаження контроллера та генерації надмірного службового трафіку. В базових рішеннях SDN контроллерів реалізовано алгоритм OSPF, що шукає найкоротший шлях, але це не вирішує питання розділення навантаження між шляхами до різних кінцевих клієнтів, а також не враховуються показники доступності обраного шляху. Саме ця задача, а саме маршрутизація з урахуванням QoS буде вирішуватися в даній роботі.

### 1.4 Моделі забезпечення QoS

В останні роки запит на мультимедійний трафік значно виріс. Мультимедійні додатки містять такий функціонал, як відеоконференції, дистанційне навчання, ігри, розраховані на багато користувачів і т.д. Такий тип додатків потребує спільного використання великого об'єму даних користувачами, що розподілені по всьому світу. Саме збільшення об'єму даних, що генерують дані додатки, дало початок механізму під назвою Quality of Service. В Microsoft говорять, що "метою QoS є надання переважного сервісу доставки для додатків, яким він потрібний, за рахунок забезпечення достатньої пропускної спроможності, контролю затримки, а також зменшення втрати даних" [16]. Компанія Cisco стверджує, що QoS "означає здатність мережі надавати якісніше обслуговування вибраному мережевому трафіку по різних технологіях, включаючи Frame Relay, Asynchronous Transfer Mode (ATM), мережі Ethernet і 802.1, SONET і IP- мережі, що маршрутизуються, які можуть використати будь-кого або все з цих базових технологій" [17]. Dictionary.com говорить, що QoS - це "властивості продуктивності мережевої послуги, можливо, включаючи пропускну спроможність, затримку передачі і пріоритет. Деякі протоколи дозволяють пакетам або потокам включати вимоги QoS "[18].

QoS - це частко жорсткі умови, що направлені на ефективність мережі для забезпечення виконання вимог умов додатків та групу технологій, що дозволяють мережі забезпечувати гарантію продуктивності. QoS являється найбільш важливим фактором для виконня. Якщо QoS в мережі не функціонує,

голові IP виклики або відео конференції будуть незадовільними та непередбачуваними. Таким чином, механізм QoS грає важливу роль в розподіленій мультимедійній системі.[19]

В зв'язку з потребою в запровадженні QoS в мережі було розроблено багато архітектур, що були розроблені для вирішення даної проблеми. В даній роботі будуть розглянуті наступні архітектури: Best Effort[20], IntServ[21] та DiffServ[22].

### 1.4.1 Best Effort

Механізм звичайного best-effort трафіку працює в мережі Інтернет останні 30 років, та не можна сказати, що найкращим чином.

Він виглядає наступним чином: всі категорії трафіку рівні між собою, нікому не віддається перевага, немає ніякої гарантії затримки та полоси. Механізм звичайного best-effort трафіку підтримує все, від передачі звичайних файлів і мейлів та веб-трафіку до відео- і аудио-трансляцій та голосового спілкування.

Даний механізм не оптимальний. Однак досвід роботи в Інтернеті показав, що механізм best-effort трафіку має велике значення, тому що дозволяє всім користувачам отримувати частину ресурсів і при цьому запобігає перевантаженням. Механізм best-effort, як реалізовано в дійсному Інтернеті, потребує мінімальних технічних вимог до інфраструктури, відсутні технічні вимоги до планування, управління чергами або запровадження механізмів в маршрутизаторах. [23]

Однак простота та статичність механізму Best-Effort не призводять до того, що його ніде не використовують. Він знаходиться в мережах з високою пропускною здатністю і відсутністю перевантажень, в місцях, де немає необхідності особливим чином відноситись до якого-небудь трафіку.

### 1.4.2 IntServ

Механізм IntServ включає два види послуг, орієнтованих на трафік в реальному часі: гарантований та прогнозований. Даний механізм інтегрує ці

| Service Class Name | Traffic Characteristics | Tolerance to | | |
|---|---|---|---|---|
| | | Loss | Delay | Jitter |
| Network Control | Variable size packets, mostly inelastic short messages, but traffic can also burst (BGP) | Low | Low | Yes |
| Telephony | Fixed-size small packets, constant emission rate, inelastic and low-rate flows | Very Low | Very Low | Very Low |
| Signaling | Variable size packets, some what bursty short-lived flows | Low | Low | Yes |
| Multimedia Conferencing | Variable size packets, constant transmit interval, rate adaptive, reacts to loss | Low - Medium | Very Low | Low |
| Real-Time Interactive | RTP/UDP streams, inelastic, mostly variable rate | Low | Very Low | Low |
| Multimedia Streaming | Variable size packets, elastic with variable rate | Low - Medium | Medium | Yes |
| Broadcast Video | Constant and variable rate, inelastic, non-bursty flows | Very Low | Medium | Low |
| Low-Latency Data | Variable rate, bursty short-lived elastic flows | Low | Low - Medium | Yes |
| OAM | Variable size packets, elastic & inelastic flows | Low | Medium | Yes |
| High-Throughput Data | Variable rate, bursty long-lived elastic flows | Low | Medium - High | Yes |
| Standard | A bit of everything | Not Specified | | |
| Low-Priority Data | Non-real-time and elastic | High | High | Yes |

послуги з контрольованим використанням каналів і призначена для ефективної роботи з багатоадресовою та одноадресовою передачею. IntServ запровадив абстракцію "потоку", як поток зв'язаних дейтаграм, що являється результатом активності одного користувача і потребує одного й того ж QoS. Наприклад, поток може складатися з одно транспортного з'єднання або одного відеопотоку між парою хостів. Тобто, поток має одне джерело та N адресатів. Таким чином

| Service Class Name | DSCP Name | DSCP Value | Application Examples |
|---|---|---|---|
| Network Control | CS6,CS7 | 110000,110000 | Network routing |
| Telephony | EF | 101110 | IP Telephony bearer |
| Multimedia Conferencing | AF41,AF42 AF43 | 100010,100100 100110 | H.323/V2 video conferencing (adaptive) |
| Multimedia Streaming | AF31,AF32 AF33 | 011010,011100 011110 | Streaming video and audio on demand |
| Low-Latency Data | AF21,AF22 AF23 | 010010,010100 010110 | Client/server transactions Web-based ordering |
| High-Throughput Data | AF11,AF12 AF13 | 001010,001100 001110 | Store and forward applications |
| Standard | DF (CS0) | 000000 | Undifferentiated applications |

N-шляхова телеконференція, як правило, потребує N потоків, по одному вихідному в кожному вузлі. Для IntServ маршрутизатор повинен реалізувати відповідне QoS для кожного потоку відповідно до моделі послуг. Функція маршрутизатора, що створює різні якості обслуговування, називається "управління трафіком". Механізм IntServ включає чотири компонента: планувальник пакетів (the packet scheduler), процедуру контроля допуску, класифікатор та протокол резервування шляху.

Пакетний планувальник управляє пересиланням різних потоків пакетів за допомогою набору черг і, можливо, інших механізмів, таких як таймери. Планувальник пакетів повинен бути реалізований в точці постановки пакетів в чергу; Це вихідний рівень драйвера типовою операційної системи і відповідає протоколу канального рівня. Деталі алгоритму планування можуть бути специфічними для конкретної вихідний середовища.

Для цілей управління трафіком і обліку кожному вхідному пакету повинен бути присвоєний деякий клас. Всі пакети одного класу отримують однакову обрабку планувальником пакетом. Присвоєння виконується класифікатором.

Вибір класу може базуватися на вмісті заголовку пакету або деякого додаткового номера класифікації, доданого до кожного пакету. Клас - це абстракція, яка може бути локальною для конкретного маршрутизатора; один і той же пакет може бути класифікований різними маршрутизаторами уздовж шляху по-різному. Наприклад, магістральні маршрутизатори можуть вибрати відображення багатьох потоків на кілька агрегованих класів, в той час як маршрутизатори, розташовані ближче до периферії, де агрегування значно менше, можуть використовувати окремий клас для кожного потоку.

Управління доступом реалізовує алгоритм прийняття рішень, який маршрутизатор або хост використовує для розуміння, чи можливо надати новому потоку необхідний QoS без впливу на гарантії, що були дані раніше. Управління доступом використовується в кожному вузлі для прийняття локального рішення про дозвіл або відмову, в той час, як хост робить запит на QoS по будь-якому шляху через Інтернет. Алгоритм управління доступом повинен відповідати моделі сервісу і він є частиною управління трафіком.

Четвертим та останнім компонентом IntServ являється протокол налаштування резервування, який необхідний для створення та підтримки специфічного для потока стану на хостах кінцевих точок і маршрутизаторів на шляху потоку. Цей протокол називається RSVP (ReSerVation Protocol). [21] Для того, щоб зрозуміти, як дана модель повинна працювати, наведемо приклад. До установки TCP сесії або до початку обміном даними, кінцеві хости відправляють RSVP Path із зазначенням необхідної смуги. І якщо обом повернувся RSVP Resv - вони можуть почати комунікувати. При цьому, якщо доступних ресурсів немає, то RSVP повертає помилку, і хости не зможуть спілкуватися або підуть по BestEffort. Тобто для будь-якого потоку в інтернеті сьогодні буде спочатку відбуватися обмін сигнальними повідомленнями з вимогами виділити ресурс для потоку. Врахуємо, що це вимагає ненульових витрат CPU і пам'яті на кожному транзитному вузлі, а також відкладає фактичне взаємодія на деякий час. І нам стає зрозуміло, чому IntServ виявився фактично мертвонародженою ідеєю - нульова масштабованість.

### 1.4.3 DiffServ

Архітектура DiffServ базується на простій моделі, в якій трафік, що поступає в мережу, класифікується за станом на межах мережі і йому призначається певна поведінка (див. рис 1.9). Кожен агрегат поведінки ідентифікується одним кодовим значенням DiffServ. В межах мережі пакети пересилаються відповідно до поведінки, що була присвоєна певному класу DiffServ. У цьому розділі розглядаються ключові компоненти функції класифікації і визначення стану трафіку, а також способи забезпечення DiffServ за допомогою поєднання визначення стану трафіку і пересилки на основі PHB.

Домен DS є безперервним набором вузлів DS, які працюють із загальною політикою конфігурації послуг і набором груп PHB, реалізованих на кожному вузлі. Домен DS має чітко певну межу, що складається з граничних вузлів DS, які класифікують і, можливо, визначають трафік, що входить, для забезпечення того, щоб пакети, які проходять через домен, були відповідним чином помічені для вибору PHB з однієї з груп PHB, підтримуваних в домені. Вузли в домені DS вибирають поведінку пересилки для пакетів на основі їх кодової точки DS, зіставляючи це значення з одним з підтримуваних PHB. Домен DS зазвичай складається з однієї або декількох мереж під одним і тим же управлінням; Наприклад, інтрамережа організації або Інтернет-провайдер. Адміністрування домена відповідає за забезпечення виділення і/або резервування адекватних ресурсів для підтримки угод про рівень обслуговування, пропонованих доменом. Основними компонентами даної архітектури є: класифікатор (classifier), маркувальник (marker), вимірювач трафіку (meter), shaper, dropper. Ці компоненти виконують деякий функціонал, що буде розглянуто в даному розділі, а саме, класифікацію трафіку, вимірювання трафіку, колоризація пакетів, обробка пакетів за кольором, обробка черг, відкидання пакетів та серіалізація. [22] В наступних розділах буде розглянуто, яким чином архітектура DiffServ працює.

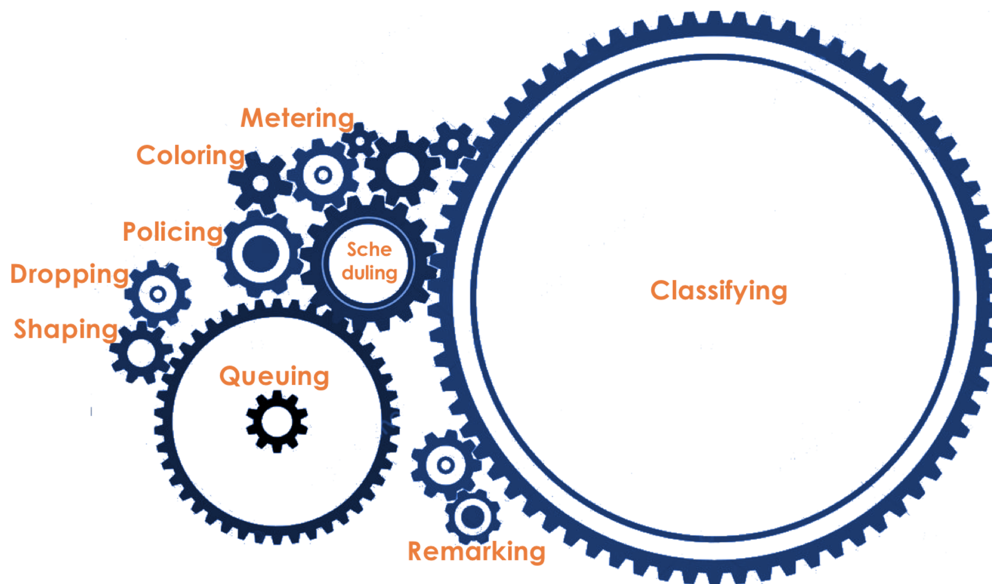



Рис. 1.9. Принцип роботи DiffServ.[29]

**Класифікація трафіку**

Класифікатори пакетів вибирають пакети в потоці трафіку на основі вмісту деякої частини заголовка пакету. Існує два типи класифікаторів. Класифікатор BA (Behavior Aggregate) класифікує пакети тільки на основі значення поля DS. Класифікатор MF (Multi - Field) обирає пакети на основі значення комбінації одного або декількох полів заголовка, таких як адресу джерела, адресу призначення, поле DS, ідентифікатор протоколу, номер порту джерела і порту призначення, і іншої інформації, такий як вхідний інтерфейс.

Адміністратор визначає набір класів сервісу, які мережа може надавати, і зіставляє їм деякий цифрове значення. [22] На вході в DS-домен не можна довіряти звонішній класифікації, тому проводиться класифікація за допомогою класифікатора способом: на основі адрес, протоколів або інтерфейсів визначається клас сервісу і відповідне цифрове значення DS за допомогою класифікатора MF. Всередині DS-домену можна довіряти маркування в полі DS, тому всередині домену використовується Behavior Aggregate класифікатор. В деяких випадках можна довіряти зовнішньому маркуванню.

DiffServ вводять поняття Per-Hop Behavior (PHB), яке визначає спосіб

обробки трафіку, що належить певному агрегату поведінки, на окремому вузлі мережі. У заголовках IP-пакетів PHB не вказані як такі, замість цього використовуються значення Differentiated Services Codepoint (DSCP). Існує всього 64 можливих значення DSCP, але такого обмеження на кількість PHB немає. В мережевому домені існує локально певне зіставлення між значеннями DSCP і PHB. Стандартизовані PHB рекомендують відображення DSCP, але оператори мережі можуть вибирати альтернативні відображення. [23]

Хоча диференційовані послуги є загальною архітектурою, яка може використовуватися для реалізації різних послуг, визначено й охарактеризовано три основні режими переадресації для загального використання. Це базове поведінку переадресації за замовчуванням (DF) для гнучкого трафіку, поведінка гарантованої переадресації (AF) і поведінку прискореної переадресації (EF) для нееластичного трафіку в реальному часі. Те, що для AF рекомендується чотири точки коду і що для EF рекомендується одна точка коду, є вільним вибором, і архітектура допускає будь-який розумний кількість класів AF і EF одночасно.[24]

**Default Forwarding (DF)**

Стандартне пересилання. Якщо класу трафіку не призначена модель поведінки спеціально, він буде оброблятися саме по Default Forwarding. Best Effort - це значить, що пристрій зробить все можливе, але нічого не гарантує. Можливі відкидання, зміна порядку, непередбачувані затримки.

Така модель підходить для невимогливих додатків, на кшталт пошти або завантаження файлів. [24]

**Assured Fowarding (AF)**

Гарантована пересилання. Це покращений BE. Тут з'являються деякі гарантії, наприклад, смуги. Відкидання і плаваючі затримки все ще можливі, але вже в набагато меншому ступені. Модель підходить для мультимедійних: Streaming, відео-конференц-зв'язку, онлайн-ігр. [25]

**Expedited Forwarding (EF)**

Екстрена пересилання. Всі ресурси і пріоритети кидаються сюди. Це модель для додатків, яким потрібні відсутність втрат, короткі затримки, стабільний джиттер, але вони не жадібні до смуги. Як, наприклад, телефонія або сервіс емуляції дроти (CES - Circuit Emulation Service). [26] Втрати, зміна порядку і плаваючі затримки в EF вкрай маловірогідні.

**Class Selector (CS)**

Селектор класів забезпечує підтримку історичних визначень кодових точок і вимог PHB. Поле DS селектора класу забезпечує обмежену зворотну сумісність з традиційною практикою (pre DiffServ). Зворотна сумісність вирішується двома способами. По-перше, існують моделі поведінки для кожного переходу, які вже широко використовуються і розробники хотіли вирішити їх подальше використання в мережах, сумісних з DS.[27]

**Класи сервісу**

IETF розробила основні категорії додатків і класи сервісу, що необхідно для них забезпечити.

| Application Categories | Service Class | Signaled | Flow Behavior | G.1010 Rating |
|---|---|---|---|---|
| Application Control | Signaling | Not applicable | Inelastic | Responsive |
| Media-Oriented | Telephony | Yes | Inelastic | Interactive |
| | Real-Time Interactive | Yes | Inelastic | Interactive |
| | Multimedia Conferencing | Yes | Rate Adaptive | Interactive |
| | Broadcast Video | Yes | Inelastic | Responsive |
| | Multimedia Streaming | Yes | Elastic | Timely |
| Data | Low-Latency Data | No | Elastic | Responsive |
| | High-Throughput Data | No | Elastic | Timely |
| | Low-Priority Data | No | Elastic | Non-critical |
| Best Effort | Standard | Not Specified | | Non-critical |

Рис. 1.10. Категорії додатків. [24]

Рис. 1.11. Вимоги до характеристик мережі. [24]

Рис. 1.12. Рекомендовані імена класів та відповідні значення DSCP. [24]

**Вимірювання показників трафіку**

Вимірувач трафіку вимірює тимчасові властивості потока пакетів, обраних класифікатором по профілю трафіку, що було задано в TCA. Вимірювач передає інформацію про стан в інші функції задавання стану, щоб ініціалізувати конкретні дії для кожного пакету, який відповідає, або не відповідає профілю[22].

**Marker**

Маркери пакетів встановлюють поле DS пакета в конкретну кодову точку, додаючи позначений пакет в конкретну групу поведінки DS. Маркер може бути налаштований для маркування всіх пакетів, які направляються до нього в одну кодову точку, або може бути налаштований для маркування пакета в одну з набору кодових точок, які використовуються для визначення PHB в групі PHB, відповідно до стану вимірювача. Коли маркер змінює кодову точку в пакеті, кажуть, що пакет "повторно позначений" [22]

**Обробка пакетів за значенням (Dropper)**

Дропери відкидають частину або всі пакети в потоці трафіку, щоб привести потік у відповідність з профілем трафіку. Цей процес відомий як "управління" потоком. Слід зазначити, що дропер може бути реалізований в якості особливого шейпера шляхом установки розміру буфера шейпера рівному нулю або кільком пакетам. [22]

**Черга (Queue)**

Черга - це структура даних, яка містить пакети, що очікують надсилання. Пакети можуть затримуватися під час перебування в черзі, можливо, через брак смуги пропускання або через низький пріоритету. Існує кілька способів реалізації черги. Однак простий моделлю системи організації черг є набір структур для пакетних даних, які ми будемо називати чергами, і механізм

вибору наступного пакета з них, який ми називаємо планувальником. Після обробки пакет повинен потрапити в одну з черг (Queuing). Для кожного класу сервісу виділена окрема черга, що і дозволяє їх диференціювати, застосовуючи різні PHB. Черги повинні організовуватися в системи черг з різними пріоритетами. А далі одна черга (з голосовими даними) буде рухатися швидко, але з обмеженою смугою, інша повільніше (потокове мовлення), зате з широкою смугою, а якийсь ресурси дістануться за залишковим принципом. Але в межах кожної окремої черги діє те ж правило - не можна висмикнути пакет з середини - тільки з його головах. Кожна чергу володіє певною обмеженою довжиною. З одного боку це диктується апаратними обмеженнями, а з іншого тим, що немає сенсу тримати пакети в черзі дуже довго.

В разі переповнення черг пакетами використовуються алгоритми, що дозвляють відкидати пакети. Серед таких алгоритмів - Tail Drop, Head Drop, Random Early Detection та Weighted Random Early Detection.

Tail Drop - найбільш простий механізм управління чергою. Він полягає у відкиданні всіх нових пакетів, що не поміщаються в буфері.

Head Drop - алгоритм, механізм якого полягаю у відкиданні пакету, що стоять в черзі дуже довго. Це дозволяє новим пакетам отримати шанс прийти вчасно до кінцевого вузлу.

Серед більш інтелектуальних алгоритмів -- Random Early Detection та Weighted Random Early Detection. Обидва ці алгоритми фіксують потенційне перенавантаження та відкидають пакети у випадковому порядку.

WRED зазвичай відкидає пакети вибірково на підставі пріоритету IP. Пакети з більш високим пріоритетом IP рідше відкидаються, ніж пакети з нижчим пріоритетом. Таким чином, трафік з більш високим пріоритетом доставляється з більш високою ймовірністю, ніж трафік з більш низьким пріоритетом. Однак можна також налаштувати WRED на ігнорування IP-пріоритету при прийнятті рішень про відкиданні, щоб забезпечити поведінку без зважування RED. WRED корисний на будь-якому вихідному інтерфейсі, де очікується перевантаження. Однак WRED зазвичай використовується в базових маршрутизаторах мережі, а не на кордоні. Прикордонні маршрутизатори

призначають IP-прецеденти пакетам у міру їх надходження в мережу. WRED використовує ці прецеденти для визначення того, як він обробляє різні типи трафіку.

**Shaper**

Шейпери затримують деякі або всі пакети в потоці трафіку, щоб привести потік у відповідність з профілем трафіку. Шейпер зазвичай має буфер кінцевого розміру, і пакети можуть бути відкинуті, якщо немає достатнього буферного простору для зберігання затриманих пакетів. Але між шейпером та чергами є деякий прошарок під назвою планувальник (scheduler).

**Scheduler**

Планувальник - це механізм управління перевантаженнями, що займається управлінням пакетами, що знаходяться в різних чергах та витягуванням пакетів з різних черг за певною логікою дій. Далі будет розглянуто різноманітні алгоритми роботи планувальника.

**Існуючі алгоритми обробки черг**

**Алгоритм FIFO**

Алгоритм FIFO (First In, First Out) - найпростіший варіант обробки черг. По-суті, QoS відсутній, тому що весь трафік обробляється однаково в одній черзі. FIFO фактично ніяк не розділяє класи (див. рис 1.13).

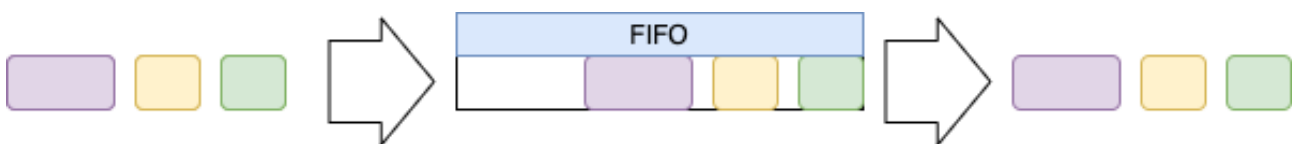



Рис. 1.13. FIFO

Якщо черга починає заповнюватися, затримка і джиттер починають рости, змінити це не можна, тому що не можна забрати важливий пакет з середини черги. Наприклад, TCP сессії можуть зайняти всю чергу, при цьому затримуючи маленькі голосові пакети в черзі. По-факту, всі класи зливаються в CS0.

FIFO зараз являється базовим планувальником з однією чергою для всього транзитного трафіку. Але FIFO стало недостатньо, тому що різний тип трафіку потребує різної обробки. Для вирішення даної проблеми було запропоновано додати декілька черг та створити схеми планування.

**Алгоритм Priority-Queue**

Алгоритм Priority-Queue - це спроба розділити сервіс на класи. Згідно даному алгоритму, трафік розкладується по декільком чергам, згідно з класом.

Рис. 1.14. Priority-Queue

Планувальник забирає пакети з кожної черги послідовно (рис. 1.14). Спочатку він забирає всі пакети з самої приоритетної черги, потім з менш приорітетної і так по черзі. Планувальник не починає забирати пакети з низькою приорітетністю до тих пір, пока черга з найбільшим приорітетом не буде пустою. Коли планувальник доходить до найменш пріоритетної черги, він починає спочатку. Якщо в момент обробки низькоприорітетних пакетів приходить пакет в більш приорітетну чергу, диспетчер переключається на неї та тільки після закінчення її обробки, повертається до інших.

PQ добре підходить для таких видів трафіку, як протокольні пакети та голос, де затримки мають критичне значення, при цьому об'єм даних не великий. Але вадою PQ полягає в тому, що у випадку перевантаження пріорітетної черги диспетчер ніколи не перейде до інших черг. Також непотрібно очікувати, що для кожної черги буде виділено своя полоса. Для зміни цього недоліку було запропоновано ввести обмеження на швидкість оброблюваного трафіку. Але це непросто контролювати.

**Алгоритм Fair Queuing**

За алгоритмом Fair Queuing для кожного потоку пакетів створюється черга (див. рис.1.15). Разом з тим, диспетчер оперує не числом пакетів, а числом бітів, що можна передати з кожної черги. Так агресивний TCP-потік не може затопити серіалізатор на виході і все черги отримують рівні можливості. Але це працювало лише в теорії.

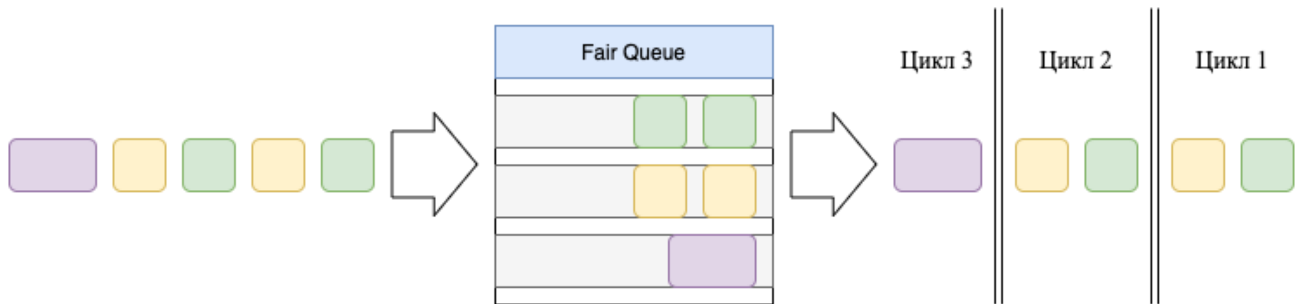
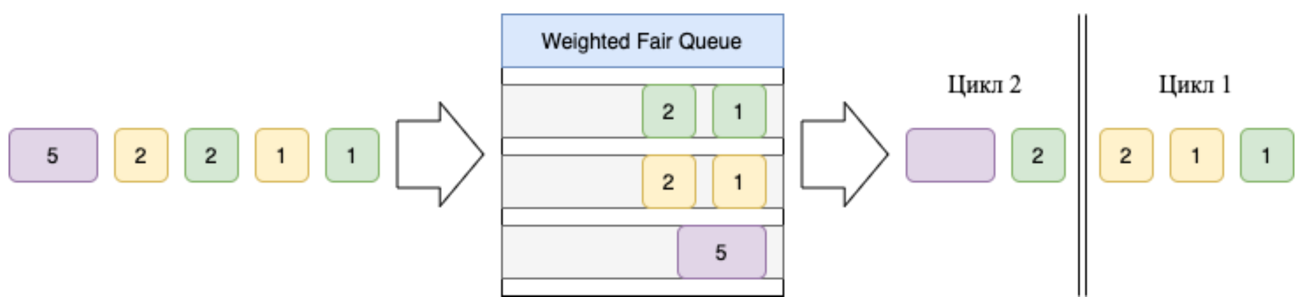



Рис. 1.15. Fair Queue

На практиці ж виявилося, що в цього алгоритма є декілька критичних недоліків. По-перше, дуже дорого створювати під кожен поток свою чергу, обраховувати вагу кожного пакету і контролювати об'єм даних, що виходить з черги. По-друге, відсутність можливості запровадження різних пріорітетів для різних черг. І найголовніше, затримки рівні у всіх потоків, тобто, якщо будет 100 черг, то одна черга будет оброблятися один раз за цикл. Ця концепція так і не була запроваджена в реальне використання. Для вирішення другої і третьої проблеми, перерахованих в попередній тезі, було запропоновано алгоритм під назвою Weighted Fair Queuing.

**Алгоритм Weighted Fair Queuing**

Алгоритм Weighted Fair Queue працює наступним чином. Кожній черзі присвоюється вага і черга може віддати об'єм трафіку, кратному даній вазі (див. рис. 1.16). Вага вираховується на базі IP Precedence і довжині пакету. Тому, чим вище IP Precedence, тим менше вага пакету. Тобто, пакети, що мають невеликий



розмір, отримують більше всього ресурсів, а великі пакети, тобто з низьким пріоритетом, очікують.

Рис. 1.16. Weighted Fair Queue

Хоча цей алгоритм був запроваджений для вирішення проблем, що виникли з FQ, він був не здатен забезпечити їх вирішення. Обробка потоків в окремих чергах не була зручною та не дозволяла потокам, що потребували отримати короткі затримки та стабільний джиттер їх отримати. Та коли побачила світ архітектура DiffServ, було запропоновано алгоритм Class-Based Weighted Fair Queue.

**Алгоритм Class-Based Weight Fair Queuing**

Алгоритм Class-Based Weighted Fair Queue створює вісім черг. Кожна черга відповідає класу трафіку. Вага назначається класам трафіку. Тобто, DSCP визначає, в яку чергу додавати пакет, призначена вага - скільки полоси доступно цій черзі. При такому підході черги з потоками, що потребували низької затримки, відвідуються частіше. Але цей алгоритм не дає жодних гарантій QoS.

**Алгоритм Class-Based Weighted Fair Queue з Low-Latency Queue**

Алгоритм Class-Based Weighted Fair Queue з Low-Latency Queue працює за наступною схемою (див. рис. 1.17) . Одна з черг стає Low-Latency Queue, тобто, чергою з низькими затримками. В той час, як всі стабільні черги працюють за алгоритмом Class-Based Weighted Fair Queue, між чергою Low-Latency Queue та іншими чергами працює планувальник Priority Queue. Тобто, в той час, як в LLQ є пакети, всі інші черги очікують, як тільки пакети в LLQ закінчились, алгоритм починає обробляти інші черги. Для того, щоб непріоріттені черги не простоювали, в LLQ можна поставити обмеження на полосі.

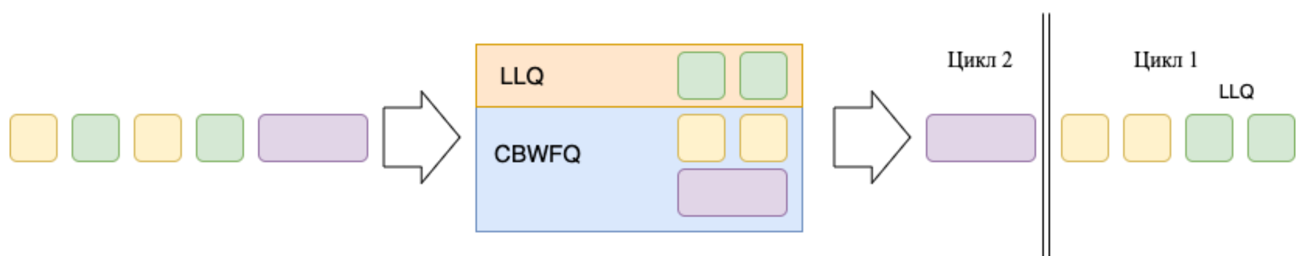


Рис. 1.17. Class-Based Weighted Fair Queue з Low-Latency Queue

**Алгоритм Round Robin**

Алгоритм Round Robin проходить послідовно всі черги та виймає рівне число пакетів (див. рис. 1.18).

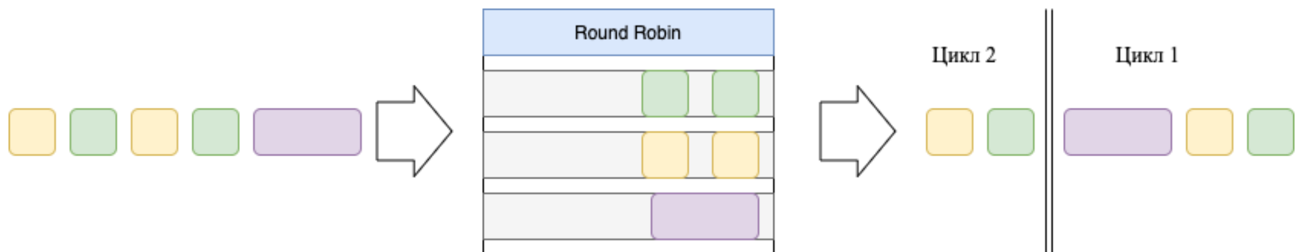
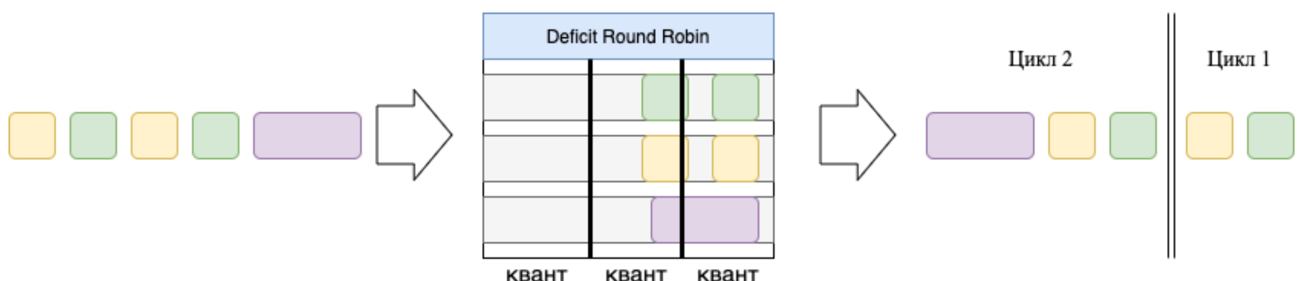



Рис. 1.18. Round Robin

Даний алгоритм є дуже простим в реалізації, але він не оцінював розмір макетів, саме тому він не був використаний на практиці. Те ж саме чекало і на Weighted Round Robin. Він задавав вагу чергам на основі IP Precedence. В цьому алгоритмі виймалася не однакова кількість пакетів, а кратне вазі черги, при цьому не можна було задавати різну вагу чергам.

**Алгоритм Deficit Weighted Round Robin**

Алгоритм Deficit Round Robin є більш цікавим. В даному алгоритмі, кожна черга має окрему кредитну лінію в бітах. При проході планувальника випускається стілки пакетів, на скільки вистачає кредитів. Від суми кредиту віднімається розмір пакету, що є в голові черги. Так відбувається до тих пір, пока в черги не закінчуються кредити. Якщо першому пакету не вистачає кредиту, то черга пропускає свій хід. Перед кожним новим проходом кредит кожної черги збільшується на перве значення, що називається кватном (див. рис. 1.19).



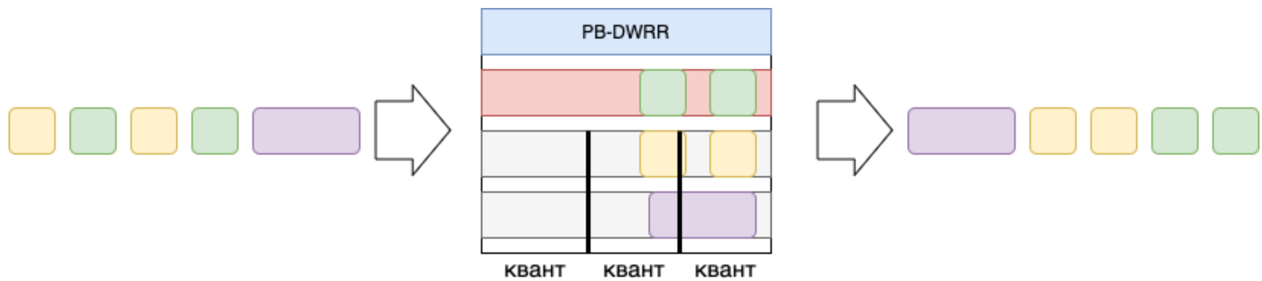



Рис. 1.19. Deficit Round Robin

Deficit Weighted Round Robin відрізняється від DRR лише тим, що для різних черг різний квант, в залежності від того, яку полосу необхідно дати трафіку в черзі. Таким чином всі черги откримують гарантовану полосу.

**Алгоритм Priority-Based Deficit Weighted Round Robin**

Алгоритм Priority-Based Deficit Weighted Round Robin відрізняється від DWRR тим, що цей алгоритм додає нову пріорітетну чергу, тобто чергу, пакети в якій обробляються частіше (див. рис. 1.20). Існує декілька підходів до реалізації PB-DWRR. Одна із його варіацій - кожен пакет, що прийде в приорітетну чергу, буде оброблений планувальником. В іншому варіанті звернення до пріоритетної черги відбувається при кожному переході диспетчера до іншої черги. Також для них вводиться кредит і квант, що не дозволяє приорітетній черзі забирати всю полосу.

Рис. 1.20. Priority-Based Deficit Weighted Round Robin

## 1.5 QoS в SDN мережах

QoS в SDN мережах, а саме в Openflow, обмежений, так як він складається з двох компонентів, черги та вимірювача. Стандартів по типу IntServ або DiffServ в SDN немає. Даний підхід дає свободу для реалізації власних алгоритмів забезпечення QoS, з одного боку, але з іншого, це додає складності в реалізацію QoS в OpenFlow.

Через відсутність стандарту QoS в OpenFlow вченими було запропоновано декілька моделей для різних цілей: деякі для вирішення комплексної задачі забезпечення QoS, інші для вирішення вузького профілю задач.

HiQoS [30] забезпечує багатошляхову QoS з використанням стандарту DiffServ. HiQoS складається з двох компонентів: компоненту, що розділяє трафік на класи та компоненту, що відповідає за маршрутизацію. Компонент, що займається розподілом трафіку за класом, забезпечує гарантію необхідної полосу для кожного сервісу, користуючись механізмом черг на SDN комутаторах. Компонент багатошляхової маршрутизації знаходить шляхи з одного хосту в інший, враховуючи оптимальність шляху за багатьма критеріями (див. рис. 1.21). Маршрути для потоків обчислюються з врахуванням поточного використання смуги пропускання в якості ваги, і потоки одного і того ж класу можуть використовувати різні шляхи. Хоча трафік з низькою затримкою класифікується за власним класом, контролер не контролює затримку в реальному часі і використовує тільки поточне використання смуги пропускання при обчисленні маршрутизації. Це базується на припущенні, що вищий коефіцієнт використання (смуга пропускання і черга) дорівнює більш високій затримці. Замість того, щоб гарантувати рівень потоку, HiQoS гарантує смугу пропускання для кожного рівня класу.

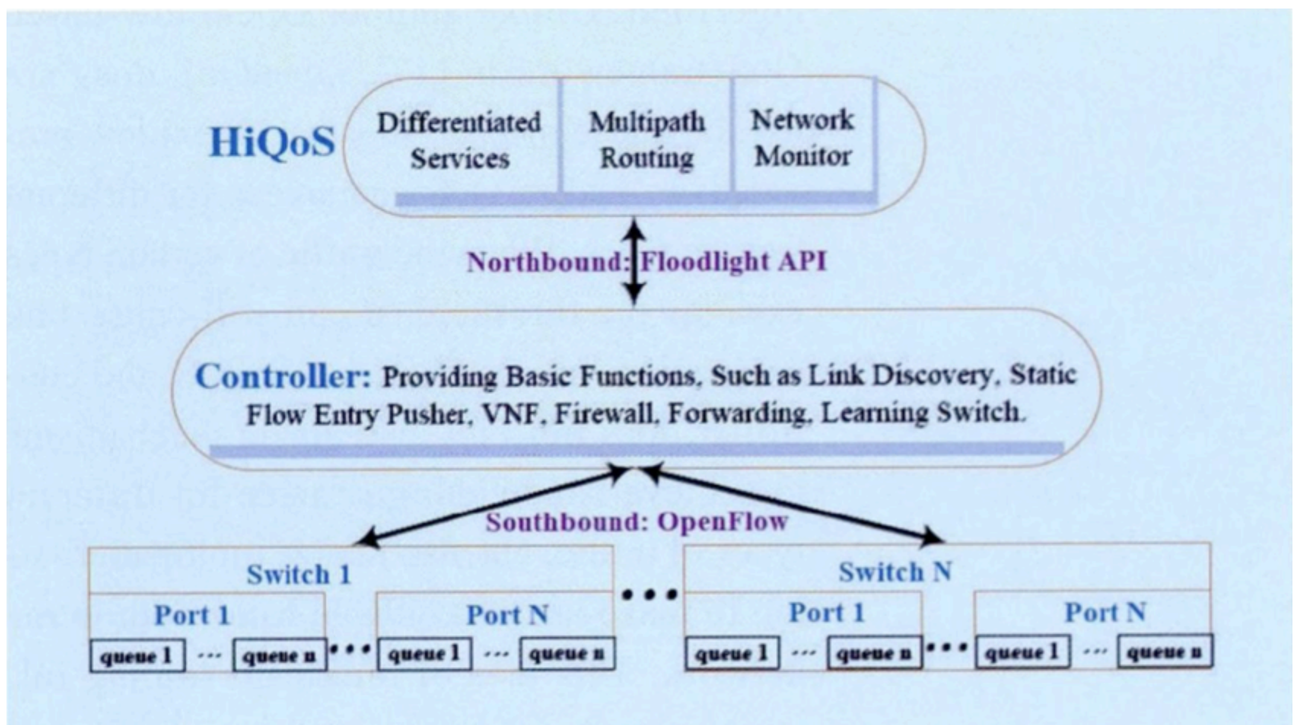



Рис. 1.21. Архітектура HiQoS. [30]

OpenQoS [31], на відміну від HiQoS сфокусований на управлінні мультимедійним трафіком та його маршрутизації (див. рис. 1.22). OpenQoS не

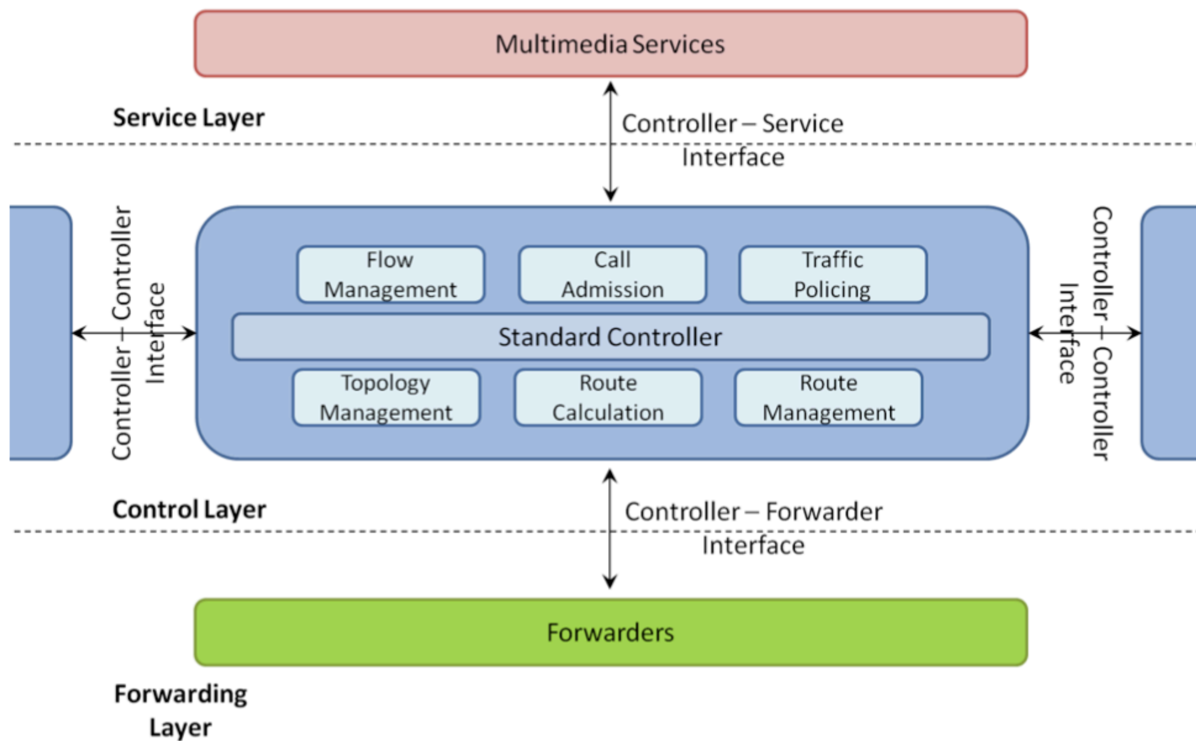



Рис. 1.22. Архітектура контролера OpenQoS. [31]

використовує приорітети в чергаха (як в DiffServ) та резервацію ресурсі (як в IntServ). Замість того, OpenQoS використовує QoS маршрутизацію, для мультимедійного трафіку використовується алгоритм Constrained Shortest Path, для звичайного трафіку використовується алгоритм пошуку найкоротшого шляху. Дана архітектура реалізована в контролері Foodlight.

Було запропоновано [32] використовувати шляхи, що були попередньо обраховані, замість звернення до контролеру при надходженні нових потоків. Даний підхід допомагає при масштабуванні високонавантажених мереж. Шляхи прораховуються кожні декілька секунд. При зверненні до контролеру, попередньо прорахований шлях відправляється у відповідь.

Те, що SDN не має стандарту QoS дає можливість запропонувати свою QoS архітектуру, базовану на вже існуючих рішеннях.

**Висновки до розділу**

Розглянуто та проаналізовано принцип роботи SDN, основні можливості протоколу OpenFlow та способи проєктування мереж на базі SDN, а також різні архітектури QoS в класичних мережах та в мережах SDN. З аналізу можливостей протоколу OpenFlow та відомих рішень в проєктуванні та маршрутизації в SDN мережах, а також існуючих рішень для запровадження QoS в мережах можна зробити наступні висновки:

1. Аналіз SDN архітектур показав, що хоча один контролер є дуже нестабільним рішенням, при відмові якого можливий збій роботи всієї мережі, але при розширенні мережі за допомогою додавання багатьох контролерів відкриваються нові неймовірні можливості для управління та адміністрації мережі. Враховуючи те, що в основі SDN стоїть принцип программування всіх необхідних можливостей, що відносяться до рівня контролю та додатків, SDN дає неймовірні можливості для реалізації різноманітного функціоналу без додаткових витрат.

2. Аналіз можливостей протокола OpenFlow показав, що дане рішення дає можливість проєктувати великі мережі SDN, динамічно змінювати поведінку обробки вхідних пакетів комутаторами, топологію мережі, контролювати трафік, його інтенсивність, направленість. Також OpenFlow має достатньо засобів для реалізації QoS.

3. Аналіз основних проблем архітектури SDN показав, що дане рішення має свої недоліки та переваги. Багато проблем в проєктуванні та експлуатації SND мереж ще вирішуються, а саме проблеми безпеки, відмовостійкості, масштабування, балансування трафіку та забезпечення QoS.

4. Аналіз архітектур QoS в класичних мережах показав, що BE широко застосвоється, але не відповідає вимогам часу. IntServ -- цікава ідея, що могла б найти своє застосування в невеликих локальних мережах, але її підхід до резервування ресурсів на всьому шляху не дає можливості даній ідеї знайти своє застосування в глобальній мережі Інтернет. DiffServ, в

свою чергу, є найбільш довершеною архітектурою з усіх попередньо розглянутих, тому що вона дає можливість кожному сегменту мережі вирішувати, яким чином управляти трафіком та при цьому дає широкий спектр інструментів для управління трафіку на різних етапах обробки.

5. Аналіз архітектур QoS в SDN мережах показав, що дана технологія не має стандартів в наданні QoS, таких, як DiffServ та IntServ. Серед спектру інструментів, що реалізовано в протоколі OpenFlow, присутня лише незначна частина компонентів QoS, такі як черги та таблиці метрик. QoS в SDN на базі OpenFlow протоколу забезпечується здебільшого за допомогою QoS маршрутизації.

6. Аналіз існуючих рішень по забезпеченню QoS в SDN на базі OpenFlow показав, що всі вони намагаються реалізувати якусь частину стандартних архітектур. Але на даний момент в OpenFlow маршрутизаторах відсутня можливість вибору способу обробки черг, також можливість реалізувати свій алгоритм забезпечення QoS дає як свободу розробникам, так і не дає можливості отримати користувачам рішення "з коробки". Тож в даній роботі буде запропоновано свій алгоритм QoS маршрутизації.

# РОЗДІЛ 2
# БАЛАНСУВАННЯ ТРАФІКУ

## 2.1 Алгоритм пошуку найкоротшого шляху

Алгоритм пошуку найкоротшого шляху, або Shortest Path First, базується на алгоритмі пошуку найкоротшого шляху Дейкстри, який працює з графами, що складаються з вершин, з'єднаних ребрами. Кращий маршрут визначається як мінімальна сума вартостей всіх ребер на шляху. В реальних умовах вершини являють собою маршрутизатори, а ребра - лінки між ними. В якості вартості ребра можуть виступати різні метрики: пропускна здатність, затримки каналу, завантаження каналу, кількість хопов і т.д. Метрика - це числовий показник. Чим нижче метрика, тим краще. В класичному вигляді в якості метрики використовують кількість хопів.

Алгоритм працює наступним чином. Кожній вершині з V присвоюється мінімальний відоме відстань від цієї вершини до початкової. Алгоритм працює покроково - на кожному кроці він «відвідує» одну вершину і намагається зменшувати мітки. Робота алгоритму завершується, коли всі вершини відвідані.
Мітка початкової вершини дорівнює 0, мітки інших вершин - нескінченності. Це відображає те, що відстані від початкової до інших вершин поки невідомі. Всі вершини графа позначаються як невідвідані. Якщо всі вершини відвідані, алгоритм завершується. В іншому випадку з ще не відвіданих вершин обирається вершина u, що має мінімальну позначку. Розглядаються всі маршрути, в яких u є передостаннім пунктом. Вершини, з'єднані з вершиною u ребрами, називаються сусідами цієї вершини. Для кожного сусіда розглядається нова довжина шляху, що дорівнює сумі поточної мітки u і довжини ребра, що з'єднує u з цим сусідом. Якщо отримана довжина менше мітки сусіда, мітка замінюється цією довжиною. Розглянувши всіх сусідів, вершина u позначається як відвідана і крок повторюється.

Хоча даний алгоритм виконує свою функцію і знаходить найкоротший шлях, він не застерігає мережу від перенавантаження. Даний алгоритм не забезпечує рівномірний розподіл трафіку по мережі, а лише надає найкоротший шлях від пункту А до пункту Б. Навіть без моделювання можна зрозуміти, що

ситуація, при котрій всі потоки трафіку проходять по одному й тому ж шляху, перенавантажуючи маршрутизатори на ньому, дуже вірогідна. Саме для того, що вирішити дану проблему було створено цілий ряд алгоритмів, щоураховують різні критерії при виборі шляху.

**2.2 Алгоритми маршрутизації QoS**

Алгоритм пошуку шляху являється головною стратегією QoS маршрутизації. Замість використання алгоритму пошуку найкоротшого шляху, основаного на статично сконфігурованих метриках, алгоритм повинен обирати декілька шляхів, що задовільняють набір критеріїв. Але алгоритми вирішення цієї задачі мають високу обчислювальну складність. Дана проблема називається Multi-Constrained Path проблемою. Оскільки оптимальне рішення цього типу задач для багатьох незалежних метрик є NP-повним, зазвичай використовують еврістичні або апроксимальні алгоритми. Складну проблему Mulit-Constrained Path було розбито на три шляхи часткового рішення, а саме, Bandwidth Restricted Path, Restricted Shortest Path та Metrics Combination. [33]

Bandwidth Restricted Path [34] проблему вирішують алгоритми сімейства Widest-Shorest Path та Shorest-Widest Path. Метою алгоритмів WSP є вибір найкоротшого шляху, який є можливим шляхом відповідно до обмеження смуги пропускання потоків. Основний метрикою, що розглядається в алгоритмах WSP, є кількість хешів, а другий метрикою є доступна смуга пропускання. Алгоритм працює наступним чином. На першому етапі обчислюються всі існуючі найкоротші шляхи між кожним джерелом і всіма адресатами в мережі. На другому етапі смуга пропускання використовується для розриву зв'язків між трактами, які мають однакову кількість транзитних ділянок, і вибирається тракт, який має найбільшу кількість доступної смуги пропускання. WSP можна обчислити за допомогою модифікованих версій алгоритмів Bellman-Ford або Dijkstra. Метою алгоритмів SWP є пошук шляхів з найбільшою кількістю доступної смуги пропускання, найширшого шляху. У другій фазі, якщо існує кілька шляхів з однаковою кількістю доступної смуги пропускання, то

вибирається найкоротший шлях відповідно до використовуваної метрикою довжини, або кількістю стрибків, або наскрізний затримкою.

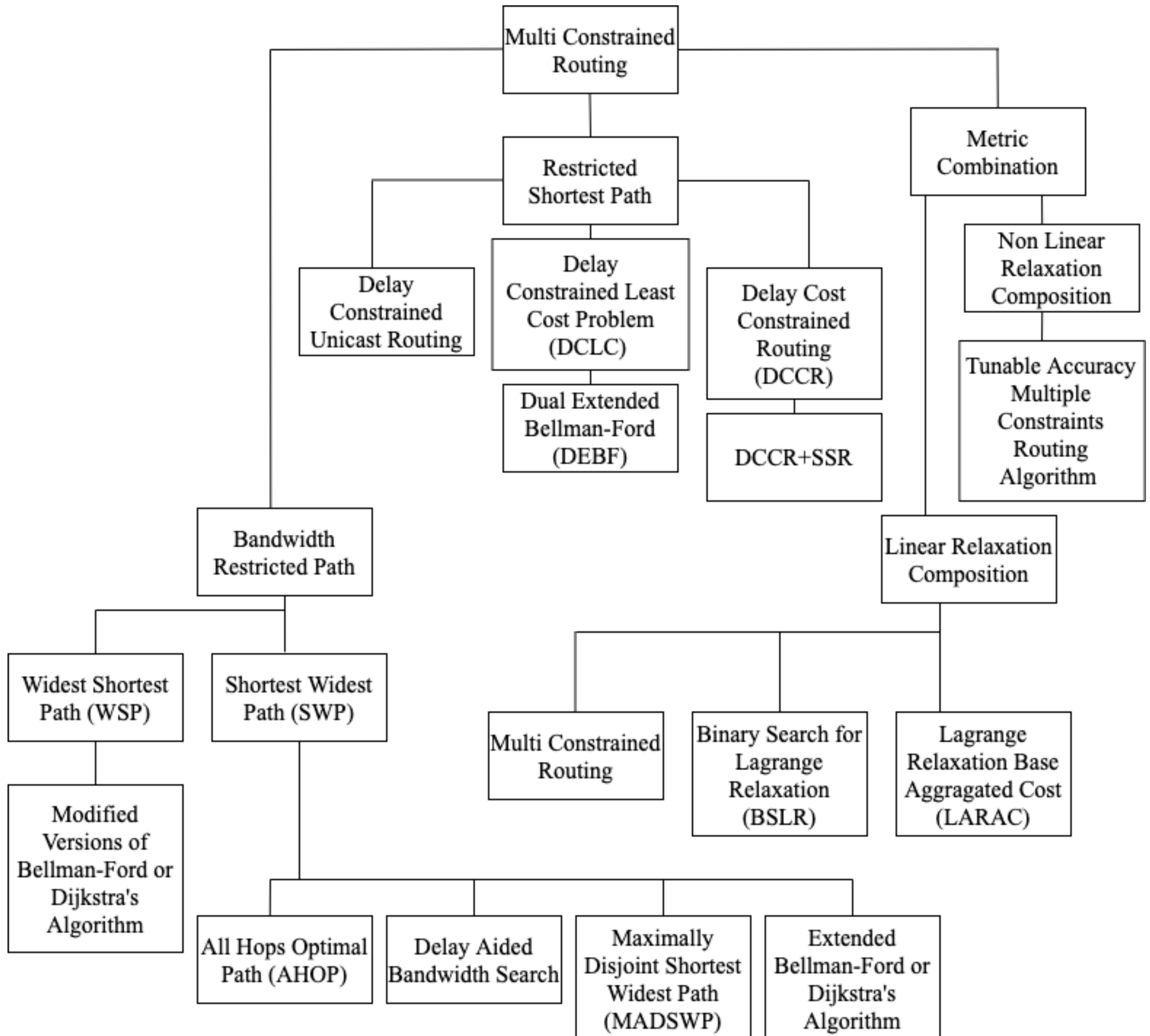



Рис. 2.1. Алгоритми QoS маршрутизації

Restricted Shortest Path [35] проблема є особливим випадком проблеми MCP, коли використовуються дві адитивні метрики. Цей клас проблем маршрутизації вирішується алгоритмами, що знаходять можливі шляхи, що відповідають одному з обмежень, і в залежності від того шукає інші оптимальні шляхи у відповідності до інших обмежень. До алгоритмів, що вирішують RSP проблему відносяться Dual Extended Bellman-Ford (DEBF), Delay-Cost Constrained Routing (DCCR) та інші.

Комбінація метрик зменшує складність проблеми RSP, об'єднуючи обидві метрики в одну метрику, а потім використовуючи традиційний алгоритм найкоротшого шляху для обчислення шляху, який мінімізує результуючу метрику. Комбінування метрик [36]: може використовувати лінійну, нелінійну і релаксаційну композицію Лагранжа. До таких алгоритмів можна віднести Multi-constrained Energy Function based Pre-computation Algorithm (MEFPA) та Jaffe Approximation Algorithm (JAA). В наступному підрозділі буде розглянуто власний алгоритм.

### 2.3 Алгоритм балансування на основі багатьох метрик

Задача балансування трафіку в мережі була преставлена, у вигляді трьох етапів, а саме, збір метрик, багатокритеріальна оптимізація та пошук оптимального шляху за результатами оптимізації.

Оптимальність шляху залежить від безлічі факторів, наприклад надійності та завантаженості маршрутизатора, затримки та доступної ширини полоси. Дані характеристики можна представити у формі вектору скалярних значень. Пошук оптимального вибору на основі безлічі критеріїв називається багатокритеріальною оптимізацією. Якщо в задачах з декількома критеріями оптимальності йдеться не про різнородні критерії деякої системи, а про співставлення однорідних критеріїв різних її підсистем, то такі задачі називаються задачами векторної оптимізації.

Задача багатокритеріальної оптимізації формулюється наступним чином. Представимо через X вектор змінних моделі, нехай X=$\{x_1, x_2, \ldots x_n\}$. Позначимо i-й критерій оптимальності через $Z_i$, i=1,...,m. Позначимо через Q множину допустимих значень змінних моделі. Тоді для кожного вектора X, що належить до множини Q, визначено значення i-го критерія оптимальності $Z_i(X)$, i= 1,...,m. В задачі багатокритеріальної оптимізації кожному допустимому значенню вектора X (X належить до множини Q) ставиться в співвідношення вектор критеріїв $Z(X) = \{Z_1(X), Z_2(X), \ldots, Z_m(X)\}$. Таким чином, вектор критеріїв Z - це вектор-функція m змінних $Z_i$, i=1,...,m; при цьому кожна компонента $Z_i$ , i=1,...,m ставить у співвідношення кожному вектору X з

множини Q значення i-го критерія оптимальності Zi(X).

До ряду методів рішення багатокритеріальної оптимізації належать субоптимізація, метод послідовних уступок, лексикографічна оптимізація, метод узагальненого критерію і т.д. В розробленому алгоритмі використовується метод узагальненого критерію.

Метод узагальненого критерію полягає у спробі "згорнути" критерії в один узагальнений скалярний критерій, оптимізація якого призводить до оптимального рішення вцілому. Метод узагальненого критерію пропонує перейти від m локальних критеріїв $Z_i$, i=1,...,m до так називаємої згортки критеріїв:

$$W(X) = \sum_{i=1}^{m} a_i \cdot Z_i(X) \to max,$$

де X належить множині допустимих рішень Q, а ваги $a_i$ визначають важливість критеріїв.

В якості критеріїв оцінки оптимальності було обрано три характеристики маршрутизаторів та шляхів, а саме, поточна ширина полоси (bandwidth), навантаження (load), або кількість пакетів, що пришло на всі порти маршрутизатора, та затримка (delay). Визначимо локальні критерії оптимальності:

$$Z_1 = \frac{B}{B_N} \quad (2.1),$$

$$Z_2 = (-\frac{L}{L_N}) \quad (2.2),$$

$$Z_3 = (-\frac{D}{D_N}) \quad (2.3),$$

$$Z_1(i) \to max,$$

$$Z_2(i) \to max,$$

$$Z_3(i) \to max,$$

де $B_N, L_n, D_N$ - нормуючі значення ширини полоси, навантаження і затримки. Побудуємо згортку критеріїв:

$$W = a_1 \cdot Z_1 + a_2 \cdot Z_2 + a_3 \cdot Z_3,$$

де $a_1, a_2, a_3$ - коефіцієнти ваги.

$$W(i) = a_1 \cdot \frac{B(i)}{B_N} + a_2 \cdot (-\frac{L(i)}{L_N}) + a_3 \cdot (-\frac{D(i)}{D_N}) \qquad (2.4),$$

де i = 1, ... , m, $B_N = B_{max}$ , $L_N = L_{max}$, $D_N = D_{max}$ .

Після обрахунку згортки для кожного критерію необхідно підготувати дані для пошуку найоптимальнішого шляху. Для цього обирається максимальне значення згортки у всій множині рішень:

$$W_{max} = W(i) \rightarrow max.$$

Далі кожне значення згортки обраховується по наступній формулі:

$$W(i) = W_{max} - W(i) + 1 \qquad (2.5).$$

Після виконання даної дії всі оптимальні рішення будуть мати мінімальні значення. Таке перетворення дає можливість знайти оптимальне рішення за допомогою алгоритма пошуку найкоротшого шляху. В запропонованому алгоритмі для пошуку оптимального шляху пропонується застосувати алгоритм Беллмана-Форда.

Далі представлено механізм роботи даного алгоритму. Маршрутизатори та зв'язки між ними представляється у вигляді графу G(V, E) з ваговою функцією $w : E \rightarrow R$. Вага шляху розраховується як:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

де шлях $p = (v_0, v_1, \dots, v_k)$.

Алгоритм Беллмана-Форда представляє собою декілька фаз. Спочатку створюється масив відстаней $d[0...n-1]$. На початку роботи алгоритму цей масив заповнюється наступним чином: $d[v] = 0$ , де $v-$ номер стартової вершини; всі інші елементи $d[]$ рідні $\infty$. На кожній фазі відвідуються всі ребра графу і алгоритм намагається виконати релаксацію вздовж кожного ребра $(a, b)$ вартості $c$. Релаксацією вздовж ребра називають спробу покращити значення

$d[b]$ значенням $d[a] + c$. Стверджується, що достатньо $n - 1$ фази алгоритму, щоб корректно обрахувати довжини всіх найкоротших шляхів у графі. Для недосяжних вершин відстань $d[]$ залишиться рівним $\infty$.

Але для вирішення задачі маршрутизації не достатньо лише знайти вагу найкоротшого шляху, а необхідно запам'ятати цей шлях. Найкоротший шлях можна представити у вигляді найкоротшого шляху до якоїсь вершини $p[a]$, до якої додали вершину $a$. Тому можна створити масив $p[0..n - 1]$, в якому для кожної вершини будемо зберігати батьківську вершину, тобто попередню в найкоротшому шляху, що веде до неї. Результатом виконання даного алгортиму і буде оптимальний шлях.

Та постає питання оптимізації даного алгоритму, зважаючи на те, сам алгоритм містить відносно велику кількість розрахунків та є деякі частини алгоритму, що потребують перерахунку лише у окремих випадках. До таких частин алгоритму відносяться збір статистики та обрахунок характеристик мережі, визначення локальних критеріїв оптимальності, згортки та підготовка згортки до пошуку найоптимальнішого шляху. Для цього потрібно зауважити, що будь яку адитивну функцію можна представити у вигляді композиції функцій:

$$f(x) = f_0[f_1(x_1, x_2), f_2(x_3, x_4), f_3(x_5, x_6)].$$

Представимо:

$$Z_1 \leftarrow f_1(B_i, B_{max}) = \frac{B_i}{B_{max}},$$

$$Z_2 \leftarrow f_2(L_i, L_{max}) = -\frac{Li}{L_{max}},$$

$$Z_3 \leftarrow f_3(D_i, D_{max}) = -\frac{Di}{D_{max}}.$$

$$W \leftarrow f_3(Z_1, Z_2) = a_1 \cdot Z_1 + a_2 \cdot Z_2 + a_3 \cdot Z_3 = a_1 \cdot f_1 + a_2 \cdot f_2 + a_3 \cdot f_3.$$

$$W_{rev} \leftarrow f_4(W_i, W_{max}) = W_{max} + 1 - W_i = W_{max} - f_4(f_1, f_2, f_3) + 1$$

Тоді таку задачу можна відобразити у вигляді дерева ( див. рис. 3.2 ).

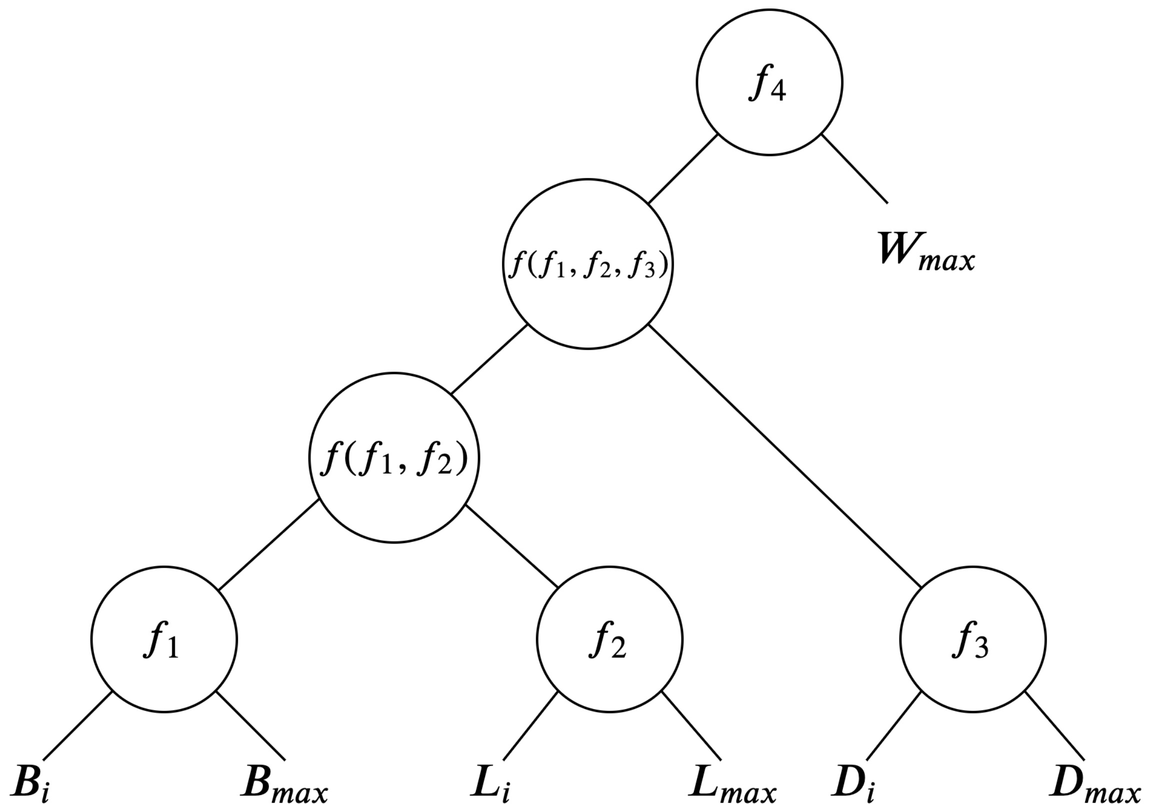



Рис. 2.2. Представлення функції у вигляді дерева

З рис.2.2 видно, що функції залежать від аргументів та переобрахунок залежить від частоти зміни значень аргументів. Повний перерахунок потрібно робити лише у ситуаціях, коли змінюються значення $B_{max}, L_{max}, D_{max}$ та $W_{max}$ у функціях, чиї аргументи змінюються та у всіх батьківських функціях.

Уявимо матрицю, кожна клітинка якої зберігає значення обрахунків функції $f(x) = f(x_1, x_2)$, дані для якої ця функція отримує з двох інших матриць з відповідними значеннями. Тоді можна відобразити все дерево обчислень у вигляді матриць та зв'язків між ними (див. рис. 2.3). Таким чином можна представити функцію у вигляді композиції функцій, кожна з яких залежить від двох змінних та відобразити обчислення функції у вигляді дерева, кожна вершина якого вказує на результати обчислень функції, а кожна гілка - на значення, що використовує дана функція. Це дозволяє зберегти результати обчислень для функцій, чиї аргументи не змінювалися та обичислити лише ті функції, аргументи яких змінились. Це дає простір для розпаралелення обчислень та скорочує кількість обчислень, скорочуючи час виконання обрахунку критеріїв для пошуку оптимального шляху.

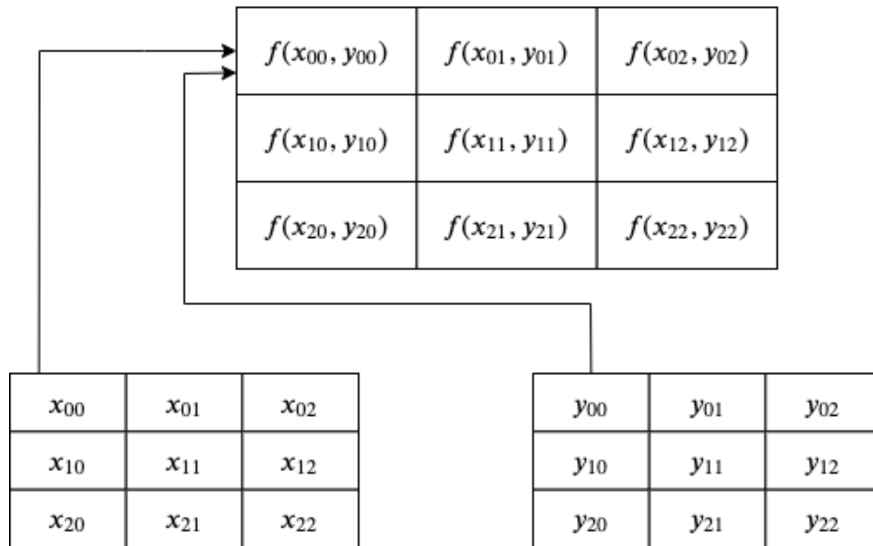



Рис. 2.3. Представлення обчислень у вигляді матриць.

Задача пошуку оптимальних рішень буде мати наступний вигляд, якщо її представити у вигляді матриць (див. рис. 2.4). Можна з легкістю зрозуміти, що функції $f_1, f_2$ та $f_3$ є незалежними одна від одної, що дає можливість розраховувати ці функції, тобто локальні критерії оптимальності, можна розрахувати паралельно. Після завершення обрахунку локальних критеріїв оптимальності, проходить обчислення згортки та підготовка результатів згортки до пошуку найоптимальнішого шляху.

Як згадувалося раніше, серед характеристик, що було обрано для розрахунку критеріїв оптимальності, знаходяться поточна ширина полоси, завантаженість маршрутизатора та затримка полоси.

В даній роботі пропонується розраховувати поточну ширину полоси за наступною формулою:

$$bandwidth_{current} = \frac{bandwidth_{capacity}}{10^3} - \frac{speed \cdot 8}{10^6} \qquad (2.6),$$

де $bandwidth_{capacity}$ - загальна ємність полоси, speed - кількість бітів, що проходять через даний маршрутизатор в секунду.

$$speed = \frac{n_{current} - n_{prev}}{t},$$

де $n_{current}$ - кількість пакетів на даний момент, $n_{prev}$ - кількість пакетів на початку замірювання, $t$ - час проведення замірів.

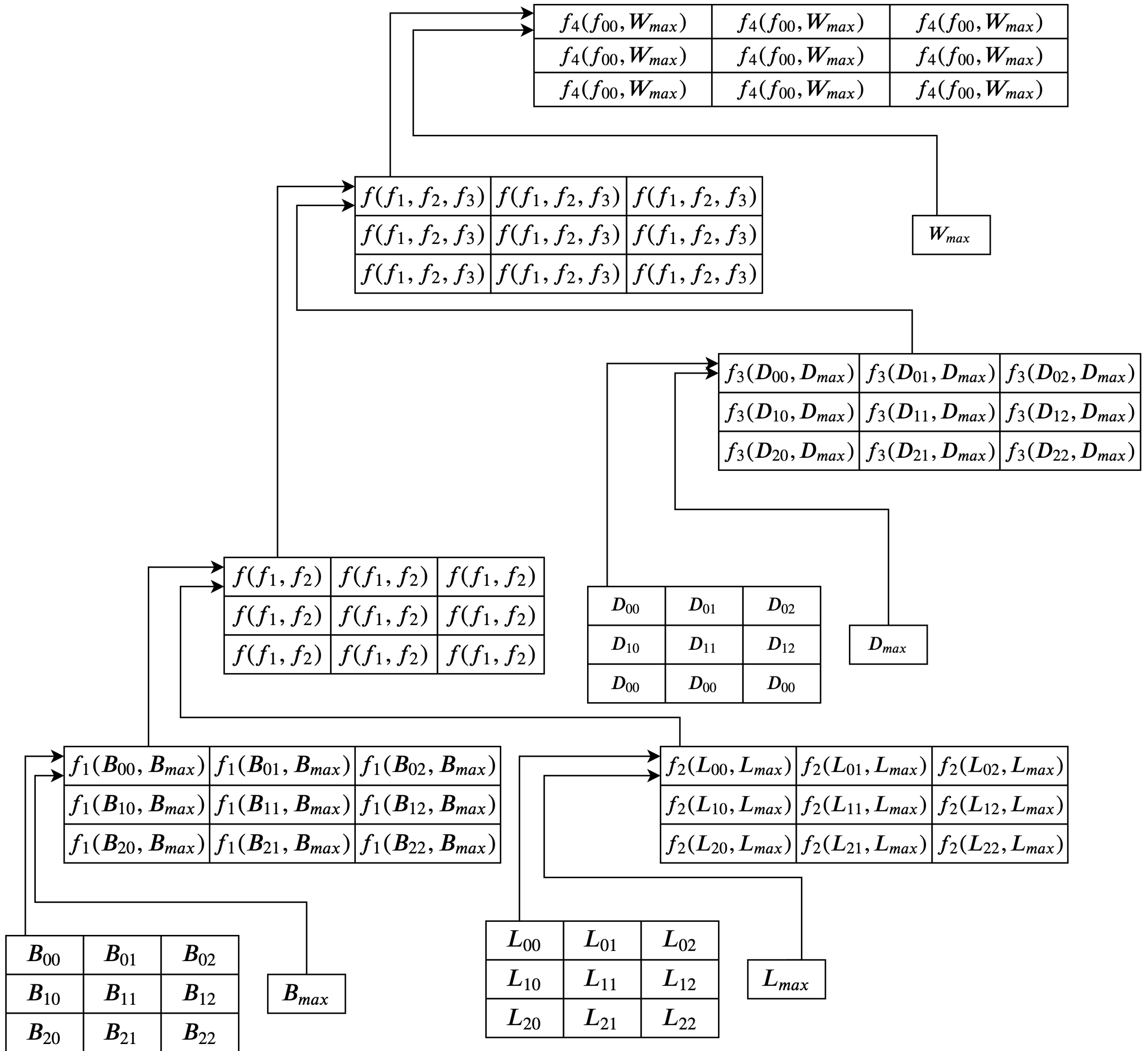



Рис. 2.4. Задача пошуку оптимальних рішень у вигляді матриць.

Завантаженість маршрутизатора пропонується розраховувати за такою формулою:

$$load_{port} = \frac{n_{current} - n_{prev}}{t},$$

де $n_{current}$ - кількість пакетів, що прийшло на порт на даний момент, $n_{prev}$ - кількість пакетів, що пройшло через порт на початку замірювання, $t$ - час проведення замірів. Таким чином для визначення загальної навантаженості маршрутизатору використовується наступна формула:

$$load_{switch} = \sum_{i=1}^{m} load_{port}(i) \quad (2.7),$$

де m - загальна кількість портів на маршрутизаторі, $load_{port}(i)$ - завантаженість $i$-го порту.

Затримка полоси пропонується обраховувати наступним чином. Так, як в OpenFlow метриках немає такої характеристики, як затримка полоси, потрібно спочатку зрозуміти, що ця характеристика означає та яким чином можна її визначити (див. рис. 2.5). Для того, щоб отримати показник затримки полоси, необхідно з SDN контролеру встановити час відправки echo-запиту, відправити цей запит на маршрутизатори, в свою чергу маршрутизатори відправлять один одному lldp пакети і встановлять затримку запиту та відповіді. Таким чином зат-

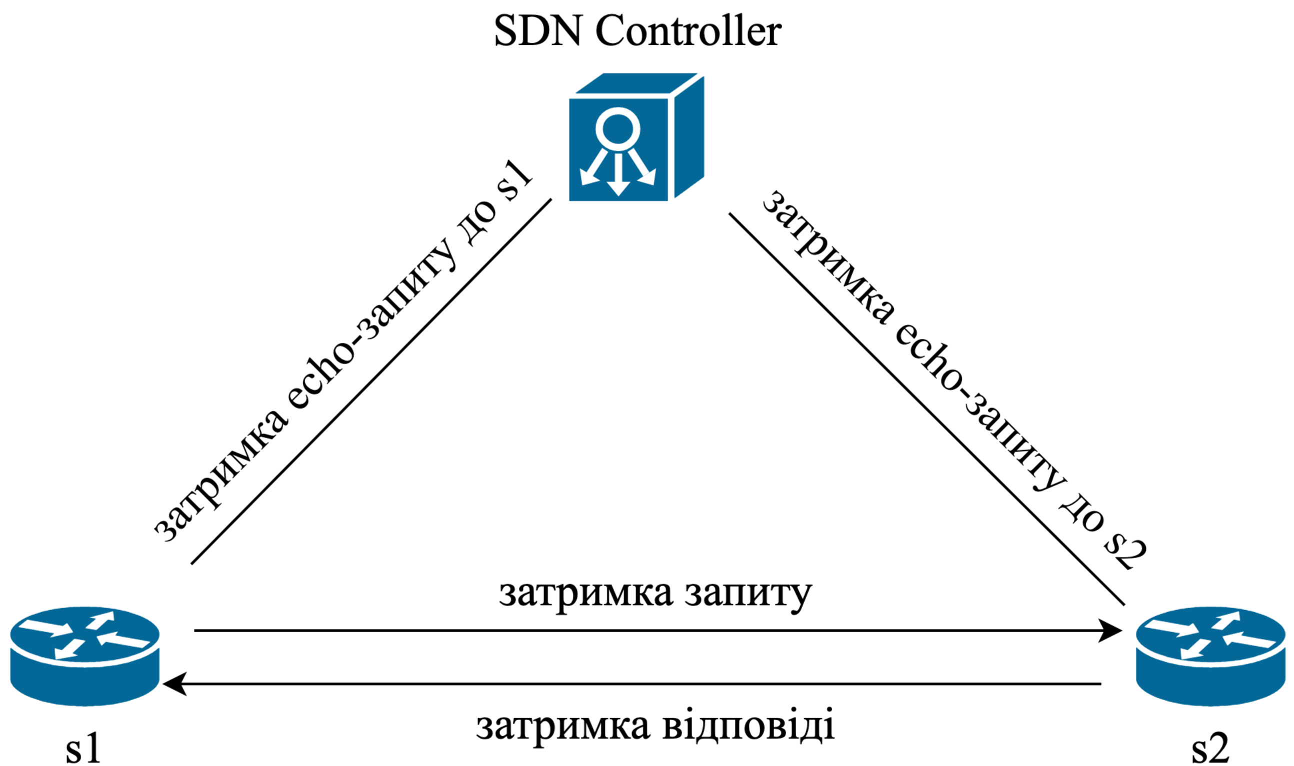



Рис. 2.5. Визначення затримки полоси.

римка полоси обчислюється за наступною формулою:

$$delay = \frac{(delay_{fwd} + delay_{reply} - echo_latency_{src} - echo_latency_{dst}}{2} \quad (2.8),$$

де $delay_{fwd}$ - затримка lldp запиту від маршрутизатора-джерела, $delay_{reply}$ - затримка lldp запиту від маршрутизатора-отримувача, $echo_latency$ - час echo-

запиту від контролера до маршрутизатора та відповіді.

В даній роботі пропонується проводити моніторинг мережі та періодично обраховувати її характеристики, на основі даних характеристик обраховувати локальні критерії оптимальності, згортки та підготовку згортки до пошуку оптимального шляху. Обрахунок же шляху робити по запиту, на основі попередньо сформованих результатів розрахунку оптимальності. Давайте проведемо приклад роботи даного алгоритму.

**Приклад**

Для демонстрації роботи даного алгоритму, представимо топологію мережі, в якій всі маршрутизатори з'єднано у вигляді топології решітки. Кожне з'єднання маршрутизаторів представлене у вигляді вектору характеристик $\overrightarrow{v}(i) = (bandwidth(i),\ load(i),\ delay(i))$. Потрібно знайти оптимальний

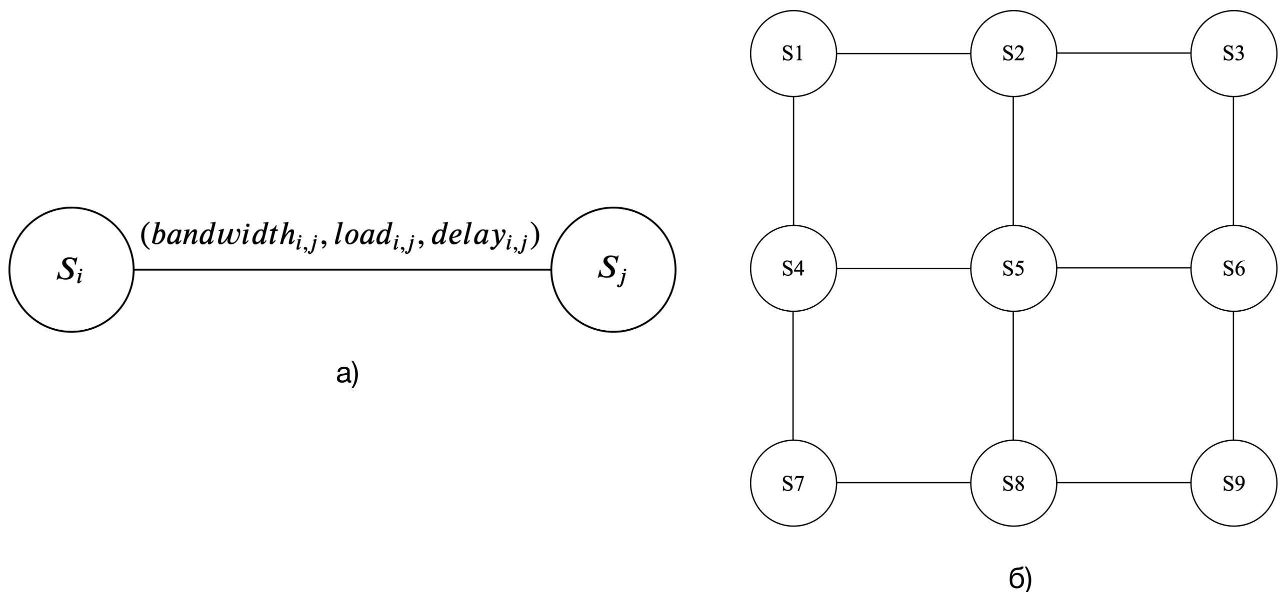



Рис. 2.6. а) Характеристики полоси з'єднання маршрутизаторів $S_i$ та $S_j$.
б) Топологія мережі - решітка.

шлях від маршрутизатора $S_1$ до маршрутизатора $S_9$.

**Етап 1**

Розрахуємо характеристики полоси з'єднання маршрутизаторів за формулами (2.6), (2.7) та (2.8). Базуючись на отриманих даних, мережа буде мати вигляд, як зображено на рис. 2.7.

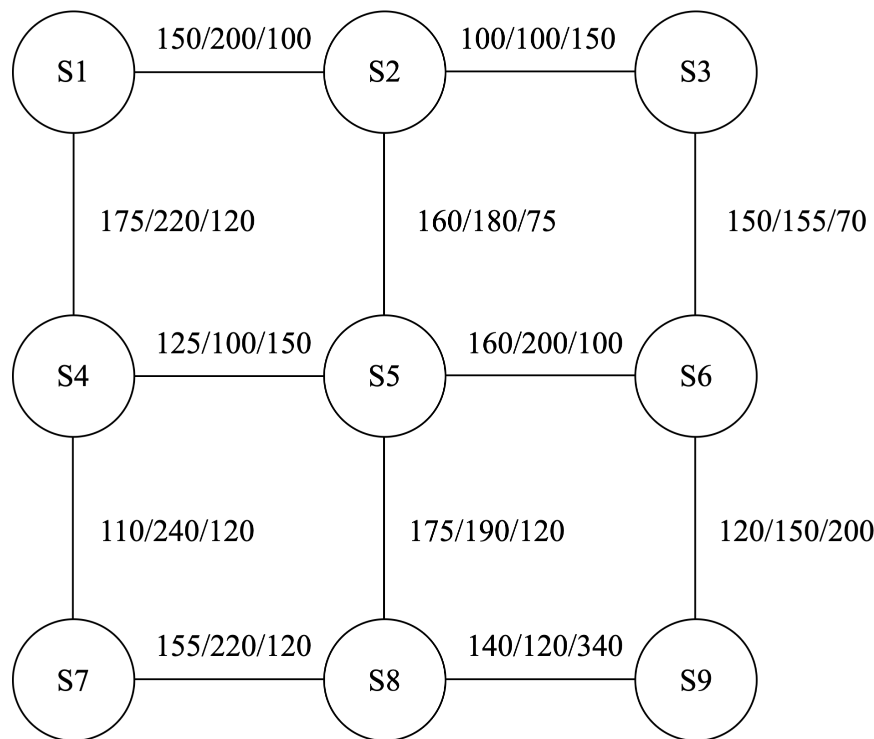



Рис. 2.7. Мережа з характеристиками полоси.

**Етап 2**

Розрахуємо локальні критерії оптимальності за формулами (2.1),(2.2) та (2.3). Припустимо, що вагові критерії будуть наступними $a_1 = 0.33$, $a_2 = 0.33$ та $a_3 = 0.33$. Максимальне значення ширини полоси: $B_{max} = 175$. Максимальне значення завантаженості маршрутизатора: $L_{max} = 240$. Максимальне значення затримки полоси: $D_{max} = 340$. Розрахуємо локальні критерії оптимальності за шириною полоси, завантаженістю маршрутизатора та затримкою

Таблиця 2.1. Локальний критерій оптимальності ширини полоси.

| $S_i \setminus S_j$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | 0 | 0.85714 | 0 | 1.0 | 0 | 0 | 0 | 0 | 0 |
| $S_2$ | 0.85714 | 0 | 0.57142 | 0 | 0.91428 | 0 | 0 | 0 | 0 |
| $S_3$ | 0 | 0.57142 | 0 | 0 | 0 | 0.85714 | 0 | 0 | 0 |
| $S_4$ | 1.0 | 0 | 0 | 0 | 0.71428 | 0 | 0.62857 | 0 | 0 |
| $S_5$ | 0 | 0.91428 | 0 | 0.71428 | 0 | 0.91428 | 0 | 1.0 | 0 |
| $S_6$ | 0 | 0 | 0.85714 | 0 | 0.91428 | 0 | 0 | 0 | 0.68571 |
| $S_7$ | 0 | 0 | 0 | 0.62857 | 0 | 0 | 0 | 0.88571 | 0 |
| $S_8$ | 0 | 0 | 0 | 0 | 1.0 | 0 | 0.88571 | 0 | 0.8 |
| $S_9$ | 0 | 0 | 0 | 0 | 0 | 0.68571 | 0 | 0.8 | 0 |

полоси та помістимо їх в таблиці 2.1, 2.2 та 2.3 відповідно.

Таблиця 2.2. Локальний критерій оптимальності завантаженості маршрутизатора.

| $S_i \setminus S_j$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | 0 | 0.83333 | 0 | 0.91666 | 0 | 0 | 0 | 0 | 0 |
| $S_2$ | 0.83333 | 0 | 0.41666 | 0 | 0.75 | 0 | 0 | 0 | 0 |
| $S_3$ | 0 | 0.41666 | 0 | 0 | 0 | 0.64583 | 0 | 0 | 0 |
| $S_4$ | 0.91666 | 0 | 0 | 0 | 0.4166 | 0 | 1.0 | 0 | 0 |
| $S_5$ | 0 | 0.75 | 0 | 0.4166 | 0 | 0.8333 | 0 | 0.79166 | 0 |
| $S_6$ | 0 | 0 | 0.64583 | 0 | 0.8333 | 0 | 0 | 0 | 0.625 |
| $S_7$ | 0 | 0 | 0 | 1.0 | 0 | 0 | 0 | 0.9166 | 0 |
| $S_8$ | 0 | 0 | 0 | 0 | 0.79166 | 0 | 0.9166 | 0 | 0.5 |
| $S_9$ | 0 | 0 | 0 | 0 | 0 | 0.625 | 0 | 0.5 | 0 |

Таблиця 2.3. Локальний критерій оптимальності затримки полоси.

| $S_i \setminus S_j$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | 0 | 0.2941 | 0 | 0.35294 | 0 | 0 | 0 | 0 | 0 |
| $S_2$ | 0.2941 | 0 | 0.4411 | 0 | 0.22058 | 0 | 0 | 0 | 0 |
| $S_3$ | 0 | 0.4411 | 0 | 0 | 0 | 0.20588 | 0 | 0 | 0 |
| $S_4$ | 0.35294 | 0 | 0 | 0 | 0.44117 | 0 | 0.35294 | 0 | 0 |
| $S_5$ | 0 | 0.22058 | 0 | 0.44117 | 0 | 0.29411 | 0 | 0.3529 | 0 |
| $S_6$ | 0 | 0 | 0.20588 | 0 | 0.29411 | 0 | 0 | 0 | 0.5882 |
| $S_7$ | 0 | 0 | 0 | 0.35294 | 0 | 0 | 0 | 0.3529 | 0 |
| $S_8$ | 0 | 0 | 0 | 0 | 0.3529 | 0 | 0.3529 | 0 | 1.0 |
| $S_9$ | 0 | 0 | 0 | 0 | 0 | 0.5882 | 0 | 1.0 | 0 |

**Етап 3**

Після обрахунку локальних критеріїв оптимальності можна розрахувати часткові згортки критеріїв. Для розрахунку необхідно скористатися формулою (2.4), підставивши значення $a_1 = 0.33$, $a_2 = 0.33$ та $a_3 = 0.33$. Результати обчислень будуть записані до Таблиці 2.4. З результатів обчислень згортки

видно, що максимальне значення згортки $W_{max} = 0.00179$. Це буде найоптимальнішим рішенням на всій площині рішень.

Таблиця 2.4. Таблиця згорток критеріїв.

| $S_i \setminus S_j$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | $-\infty$ | -0.08919 | $-\infty$ | -0.0889 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ |
| $S_2$ | -0.08919 | $-\infty$ | -0.0944 | $-\infty$ | -0.1317 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ |
| $S_3$ | $-\infty$ | -0.09449 | $-\infty$ | $-\infty$ | $-\infty$ | 0.00179 | $-\infty$ | $-\infty$ | $-\infty$ |
| $S_4$ | -0.0889 | $-\infty$ | $-\infty$ | $-\infty$ | -0.0473 | $-\infty$ | -0.239 | $-\infty$ | $-\infty$ |
| $S_5$ | $-\infty$ | -0.01857 | $-\infty$ | -0.047 | $-\infty$ | -0.0703 | $-\infty$ | -0.0477 | $-\infty$ |
| $S_6$ | $-\infty$ | $-\infty$ | 0.00179 | $-\infty$ | -0.0703 | $-\infty$ | $-\infty$ | $-\infty$ | -0.174 |
| $S_7$ | $-\infty$ | $-\infty$ | $-\infty$ | -0.239 | $-\infty$ | $-\infty$ | $-\infty$ | -0.1266 | $-\infty$ |
| $S_8$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | -0.0477 | $-\infty$ | -0.1266 | $-\infty$ | -0.017 |
| $S_9$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | -0.174 | $-\infty$ | -0.231 | $-\infty$ |

**Етап 4**

Тепер використаємо формулу (2.5) для перетворення згортки на матрицю для пошуку оптимального шляху. Результати запишемо до Таблиці 2.5.

Таблиця 2.5. Таблиця для пошуку оптимального шляху.

| $S_i \setminus S_j$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | $\infty$ | 1.09098 | $\infty$ | 1.09075 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $S_2$ | 1.09098 | $\infty$ | 1.09628 | $\infty$ | 1.13351 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $S_3$ | $\infty$ | 1.09628 | $\infty$ | $\infty$ | $\infty$ | 1.0 | $\infty$ | $\infty$ | $\infty$ |
| $S_4$ | 1.09075 | $\infty$ | $\infty$ | $\infty$ | 1.04914 | $\infty$ | 1.2408 | $\infty$ | $\infty$ |
| $S_5$ | $\infty$ | 1.02037 | $\infty$ | 1.04914 | 0 | 1.07212 | $\infty$ | 1.04949 | $\infty$ |
| $S_6$ | $\infty$ | $\infty$ | 1.0 | $\infty$ | 1.07212 | $\infty$ | $\infty$ | $\infty$ | 1.17586 |
| $S_7$ | $\infty$ | $\infty$ | $\infty$ | 1.24083 | $\infty$ | $\infty$ | $\infty$ | 1.12844 | $\infty$ |
| $S_8$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 1.04949 | $\infty$ | 1.12844 | $\infty$ | 1.01924 |
| $S_9$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 1.17586 | $\infty$ | 1.23279 | $\infty$ |

**Етап 5**

На цьому этапі необхідно застосувати алгоритм Белмана-Форда для пошука найоптимальнішого шляху.

Даний алгоритм перебирає послідовно всі ребра і проводить релаксацію вздовж кожного ребра. Якщо дистанція до вершини, що зараз розглядається, більша за дистанцію до його сусіда разом з вагою переходу до поточної вершини, нова дистанція записується у дану вершину. Тож в даному випадку будемо записувати лише релаксацію:

- маршрутизатор 2, сусідній маршрутизатор 1, попередня вага = inf, нова вага = 1.09098, найоптимальніший шлях з маршрутизатора 2 в маршрутизатор 1;
- маршрутизатор 4, сусідній маршрутизатор 1, попередня вага = inf, нова вага = 1.090759, найоптимальніший шлях з маршрутизатора 4 в маршрутизатор 1;
- маршрутизатор 3, сусідній маршрутизатор 2, попередня вага = inf, нова вага = 2.1872, найоптимальніший шлях з маршрутизатора 3 в маршрутизатор 2 ;
- маршрутизатор 5, сусідній маршрутизатор 2, попередня вага = inf, нова вага = 2.2245, найоптимальніший шлях з маршрутизатора 5 в маршрутизатор 2 ;
- маршрутизатор 6, сусідній маршрутизатор 3, попередня вага = inf, нова вага = 3.1872, найоптимальніший шлях з маршрутизатора 6 в маршрутизатор 3 ;
- маршрутизатор 5, сусідній маршрутизатор 4, попередня вага = 2.2245023, нова вага = 2.139, найоптимальніший шлях з маршрутизатора 5 в маршрутизатор 4;
- маршрутизатор 7, сусідній маршрутизатор 4, попередня вага = inf, нова вага = 2.33159, найоптимальніший шлях з маршрутизатора 7 в маршрутизатор 4;
- маршрутизатор 8, сусідній маршрутизатор 5, попередня вага = inf, нова вага = 3.1894, найоптимальніший шлях з маршрутизатора 8 в маршрутизатор 5;
- маршрутизатор 9, сусідній маршрутизатор 6, попередня вага = inf, нова вага = 4.3631353, найоптимальніший шлях з маршрутизатора 9 в маршрутизатор 6;
- маршрутизатор 9, сусідній маршрутизатор 8, попередня вага = 4.3631353, нова вага = 4.2086491, найоптимальніший шлях з маршрутизатора 9 в маршрутизатор 8;

За результатами виконання було знайдено найоптимальніший шлях переходу з маршрутизатора $S_1$ до маршрутизатора $S_9$: $S_1 \to S_4 \to S_5 \to S_8 \to S_9$. Даний шлях було зображено на рис. 2.8.

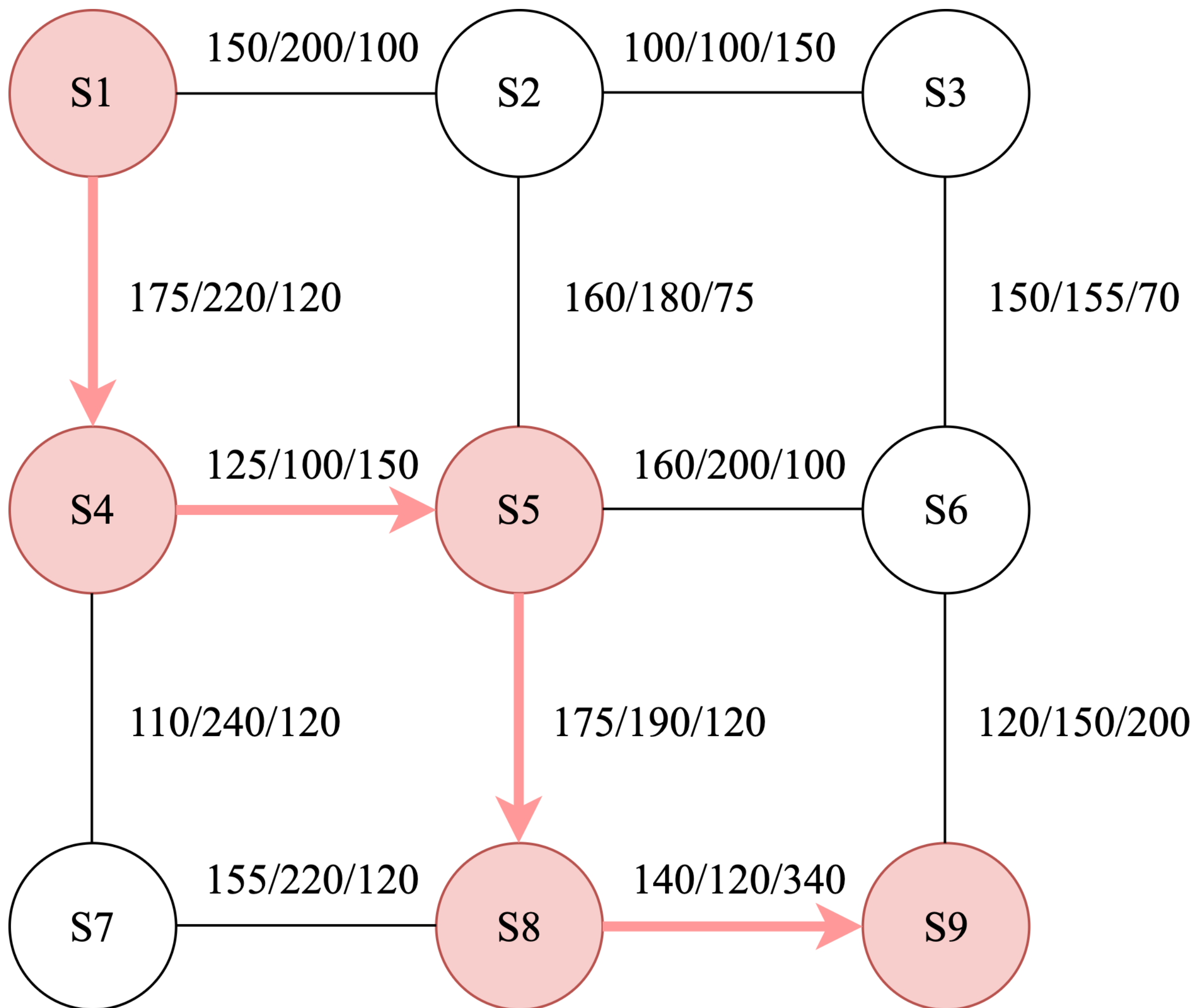



Рис. 2.8. Найбільш оптимальний шлях в мережі від $S_1$ до $S_9$.

З даного прикладу можна з легкістю побачити, що через таку кількість критеріїв дуже складно зрозуміти, який шлях є оптимальним, та запропонований алгоритм дає змогу припустити, який саме є оптимальним.

**Висновки до розділу**

Запропоновано метод балансування трафіку в масштабних мережах за допомогою алгоритму, в основі якого лежить прийняття рішень на базі багатьох критеріїв, або багатокритеріальна оптимізація. Перевагами даного метода є:

1. Обчислення даного алгоритму мають слабку зв'язність, це дає можливість розбити виконання алгоритму на етапи, виконуючи їх в різні періоди роботи системи. Наприклад, обчислення метрик може проводитися асинхронно, по виклику події маршрутизатором, у відповідь на запит метрик контролером, обчислення локальних критеріїв оптимальності можна виконувати паралельно, тому що вони виконуються незалежно одна від одної, а обчислення згортки та підготовка згортки до пошуку оптимального шляху можна  виконувати послідовно відносно одна одної, але паралельно, відносно обчислень. Такий підхід дає можливість скоротити час виконання різних етапів алгоритму.

2. Збереження результатів різних етапів обчислень і зміна значень різних характеристик лише при зміні значення, більше або менше ніж на 15% та обробка лише тих функцій, чиї аргументи були змінені дає можливість значно скоротити кількість обчислень та знімає значну кількість навантаження з контролеру.

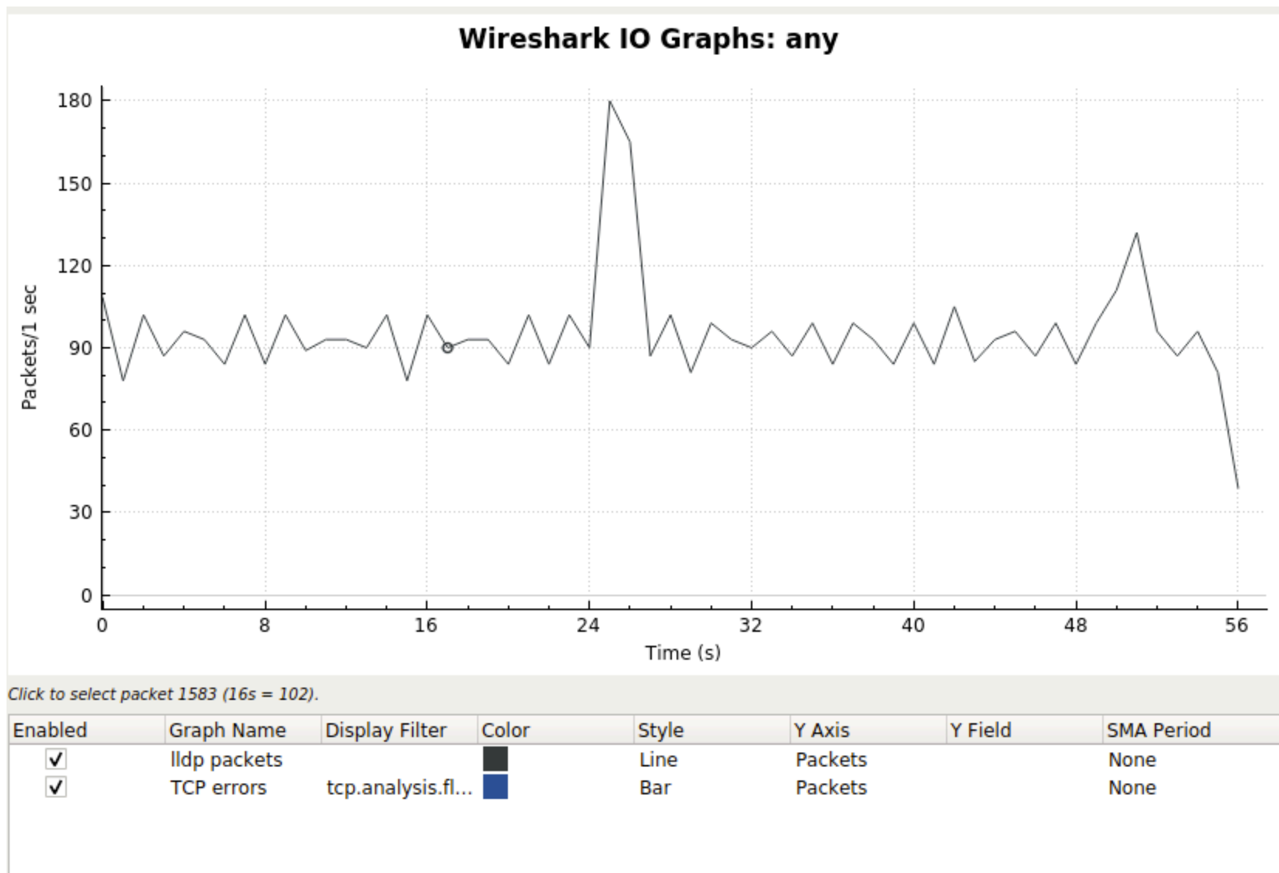
3. Даний алгоритм дає можливість обрахувати оптимальний шлях для різних класів трафіку і дає можливість додати специфічну логіку обробки пакетів різних класів, додати черги та направляти різні класи трафіку до черг з різним рейтом.

4. В зв'язку з тим, що в даному випадку використовується алгоритм Белмана-Форда для пошуку оптимального шляху, цей процес можна виконувати паралельно, так як даний алгоритм має здатність до розпаралелювання.

Недоліками даного метода є:

1. Постійний моніторинг характеристик мережі, а саме, затримки, навантаженості та ширини полоси, потребує значних обчислювальних ресурсів контролера та збільшує кількість системного трафіку в мережі.

2. Затримки полоси, як базової метрики OpenFlow протоколу не існує, тому




необхідно додатково реалізовувати функціонал для отримання затримки полоси.

# РОЗДІЛ 3
# РОЗРОБКА МОДУЛЮ БАЛАНСУВАННЯ ТРАФІКУ В SDN

## 3.1 Огляд інструментів розробки для SDN мереж.

Існує велика кількість інструментів для розробки, проектування, моделювання та тестування SDN мереж. Для емуляції SND мереж існує інструмент під назвою Mininet.

Mininet - мережевий емулятор, що створює мережу віртуальних хостов, комутаторів, контроллерів і з'єднань. Хости Mininet працюють під управлінням стандартного мережевого програмного забезпечення Linux, а його комутатори підтримують OpenFlow для дуже гнучкої призначеної для користувача маршрутизації і програмно-конфігурованої мережі. Mininet підтримує дослідження, розробку, навчання, створення прототипів, тестування, відладку і будь-які інші завдання, які можуть отримати вигоду від наявності повної експериментальної мережі на ноутбуку або іншому ПК.

Mininet:

- забезпечує просту і недорогу мережеву тестову базу для розробки додатків OpenFlow;
- дозволяє декільком паралельним розробникам працювати незалежно в одній топології;
- підтримує регресійні тести системного рівня, які є повторюваними і легко упакованими;
- дає можливість складного тестування топології без необхідності підключення фізичної мережі;
- включає інтерфейс командного рядка з підтримкою топології і OpenFlow для відладки або виконання мережевих тестів;
- підтримує довільні призначені для користувача топології і включає базовий набір топологий, що параметризуються, може використатися без програмування, але також надає простий і розширюваний API Python для створення і експериментування над мережею;

- Mininet забезпечує простий спосіб отримання правильної поведінки системи (і, в тій мірі, в якій це підтримується аппаратним забезпеченням, продуктивністю) і проведення експериментів з топологіями.

Мережі Mininet виконують реальний код, включаючи стандартні мережеві додатки Unix/Linux, а також реальне ядро Linux і мережевий стек (включаючи будь-які доступні розширення ядра, якщо вони сумісні з просторами імен мережі). Через це розроблений і тестований код на Mininet для контроллера OpenFlow, модифікованого комутатора або хоста може перейти до реальної системи з мінімальними змінами, для реального тестування, оцінки продуктивності і розгортання. Важливо, що це означає, що конструкція, працююча в Mininet, зазвичай може бути встановлена безпосередньо на апаратні комутатори для передачі пакетів з лінійною швидкістю.[37]

В розробці та тестуванні модулю в даній роботі було використано Mininet в якості емулятора мережі.

Наступним кроком в розробці модулю для SDN є вибір контролеру на базі SDN. Контроллер є основним компонентом будь-якої інфраструктури SDN, оскільки він має глобальний вигляд усієї мережі, включаючи облаштування SDN площини даних. Він сполучає ці ресурси з додатками управління і виконує дії потоку, визначувані політикою додатків на пристроях.[38] В Таблиці 3.1 приведені різні контролери з характеристиками для порівняння.

Таблиця 3.1. Контролери на базі SDN та їх характеристики [38].

| **Назва** | **Мова програмування** | **Архітектура** | **Northbound API** | **Southbound API** | **Багатопоточність** | **Модульність** |
|---|---|---|---|---|---|---|
| Ryu | Python | Централізована | REST | OpenFlow 1.0-1.5 | + | + |
| Beehive | Go | Розподілена, ієрархічна | REST | OpenFlow 1.0, 1.2 | + | + |
| DCFabric | C, Javascript | Централізована | REST | Openflow 1.3 | + | + |
| Disco | Java | Розподілена, пласка | REST | Openflow 1.0 | - | + |

*Продовження Таблиці 3.1.* Контролери на базі SDN та їх характеристики [38].

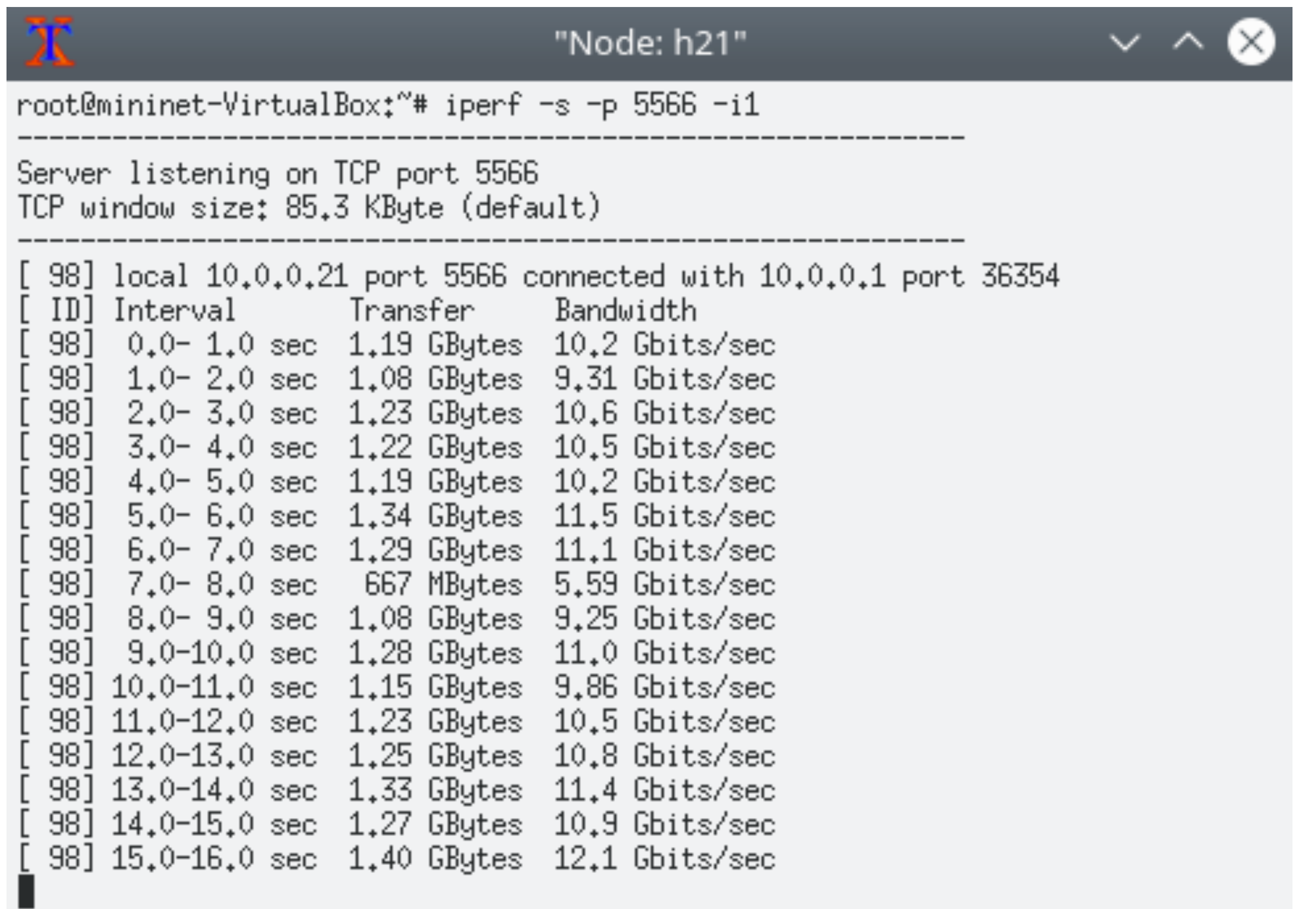
| **Назва** | **Мова програм ування** | **Архітектура** | **Northbou nd API** | **Southbound API** | **Багатопоточ ність** | **Модуль ність** |
|---|---|---|---|---|---|---|
| Kandoo | C, C++, Python | Розподілена, ієрархічна | Java RPC | OpenFlow 1.0-1.2 | + | + |
| Maestro | Java | Централізована | ad-hoc | OpenFlow 1.0 | + | + |
| POX | Python | Централізована | ad-hoc | OpenFlow 1.0 | - | + (низька) |
| NOX | C++ | Централізована | ad-hoc | OpenFlow 1.0 | + (Nox-MT ) | + (низька) |
| Onix | C++ | Розподілена, пласка | Onix API | OpenFlow 1.0, OVSDB | + | + |
| OpenDaylight | Java | Розподілена, пласка | REST, RESTCON F, XMPP, NETCONF | OpenFlow 1.0, 1.3 | + | + |
| Beacon | Java | Централізована | ad-hoc | OpenFlow 1.0 | + | + |
| Faucet | Python | Централізована | - | OpenFlow 1.3 | + | - |
| Floodlight | Java | Централізована | REST, Java RPC, Quantum | OpenFlow 1.0, 1.3 | + | + |
| FlowVisor | C | Централізована | JSON RPC | OpenFlow 1.0, 1.3 | - | - |
| HyperFlow | C++ | Розподілена, пласка | - | OpenFlow 1.0 | + | + |

В даній роботі в якості контролеру було обрано Ryu, тому що розробка модулів виконується на Python, а це дає можливість швидко розробляти прототипи та перевіряти припущення. Цей контролер має хорошу документацію, приклади готових модулів та книгу по використанню його фреймворку. Також він підтримує багатопоточність та підтримує всі версії протоколу Openflow. Саме через такі переваги було обрано даний контролер у якості майданчика для розробки та моделювання.

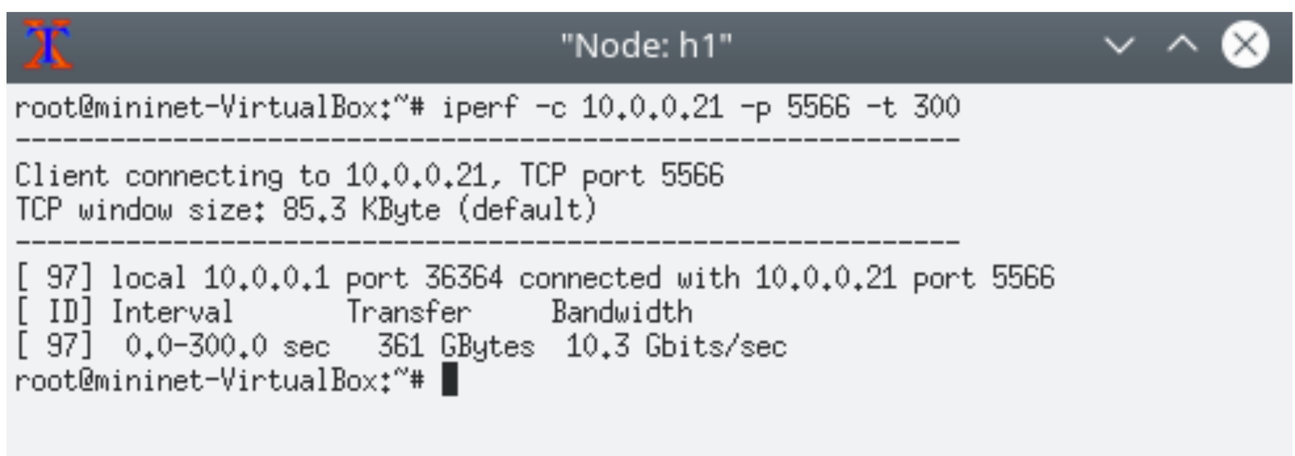
Для перевірки правильності роботи модулю, діагностики мережі та аналізу протоколів і пакетів, що пересуваються по мережі, було використано програму Wireshark. Перейдемо до розгляду запропонованої архітектури модулю балансування трафіку в SDN мережі.

### 3.2 Архітектура модулю балансування трафіку в SDN мережі.

В даній роботі було запропоновано таку архітектуру модулю балансування трафіку в SDN мережі (див. рис 3.1): Awareness App, Delay Detector App, Network Monitor App, Route Detector App та Main App.




Awareness App - це додаток, що представляє собою спостерігач за




топологією. Він надає необхідну інформацію про топологію мережі, стан

```
path -> [1, 15, 13, 11]
Path installation finished in  0.0034961700439453125
path -> [11, 13, 15, 1]
Path installation finished in  0.004245281219482422
```

зв'язків та реагує на появу нових маршрутизаторів та зв'язків в мережі, а також вихід зі стану вже існуючих. Це дає можливість зосередити логіку контролю стану топології в одному додатку та надавати доступ до топології іншим. Цей додаток використовують інші додатки, що працюють з метриками та побудовою оптимального шляху.

Delay Detector App - це додаток, що займається вимірюванням затримки полоси, використовуючи метод, що описано в Розділі 2.3 даної роботи. Цей до-

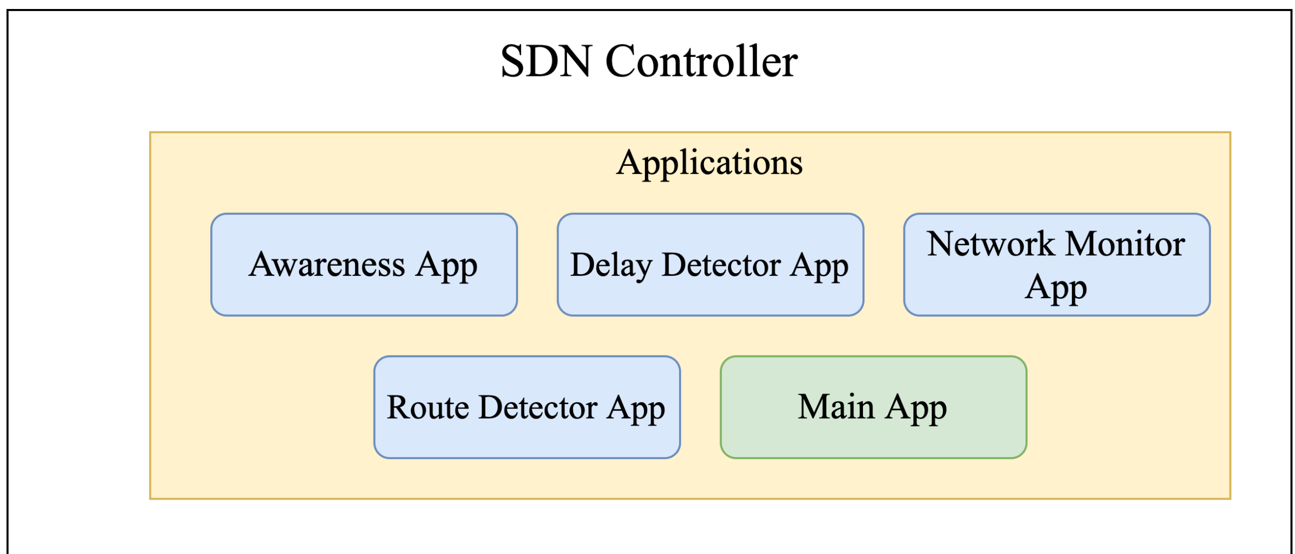



Рис. 3.1. Архітектура модулю балансування трафіку в SDN мережі.

даток відправляє echo-запит до всіх маршрутизаторів, ті в свою чергу відправляють один одному lldp пакети, що заміряють час проходу від одного маршрутизатора до іншого. Після чого дані пакети відправляються на контролер, інкапсульовані в Openflow пакет виду Packet-In. Отримана інформація заноситься в карту топології з даними про затримку на з'єднаннях маршрутизаторів.

Network Monitor App - це додаток, що виконує моніторинг мережі, періодично запитуючи метрики маршрутизаторів. Він відповідає за обрахунок таких метрик, як вільна ширина полоси та швидкість потоку пакетів,що приходять на порти маршрутизатора, а також швидкість flow пакетів.

Route Detector App - це додаток, що на основі метрик, що були обраховані

додатками Network Monitor App та Delay Detector App, а також топології, що надає Awareness App, робить попередню обробку даних метрик та обраховує оптимальність переходів з одного маршрутизатора в інший. Також даний додаток надає можливість іншим модулям отримувати оптимальний шлях з одного маршрутизатора в інший.

Main App - це додаток, що є точкою входу для всіх модулів, вони ініціалізуються в даному додатку. Також він обробляє пакети типу Packet-In, що прийшли на контролер, та після чого викливає метод Route Detector App, що займається пошуком оптимального шляху, після чого, на основі отриманого шляху, встановлює необхідні правила перенаправлення потоку вздовж всього шляху на кожен маршрутизатор.

### 3.3 Моделювання методу балансування трафіку.

Для тестування працездатності методу балансування трафіку в даній роботі було згенеровано топологію дерево з 1 віддаленим контролером, 16 хостами, 15 маршрутизаторами та 30 зв'язками між ними. Згенеровано дану топологію було в мережевому емуляторі mininet (див. рис. 3.2).

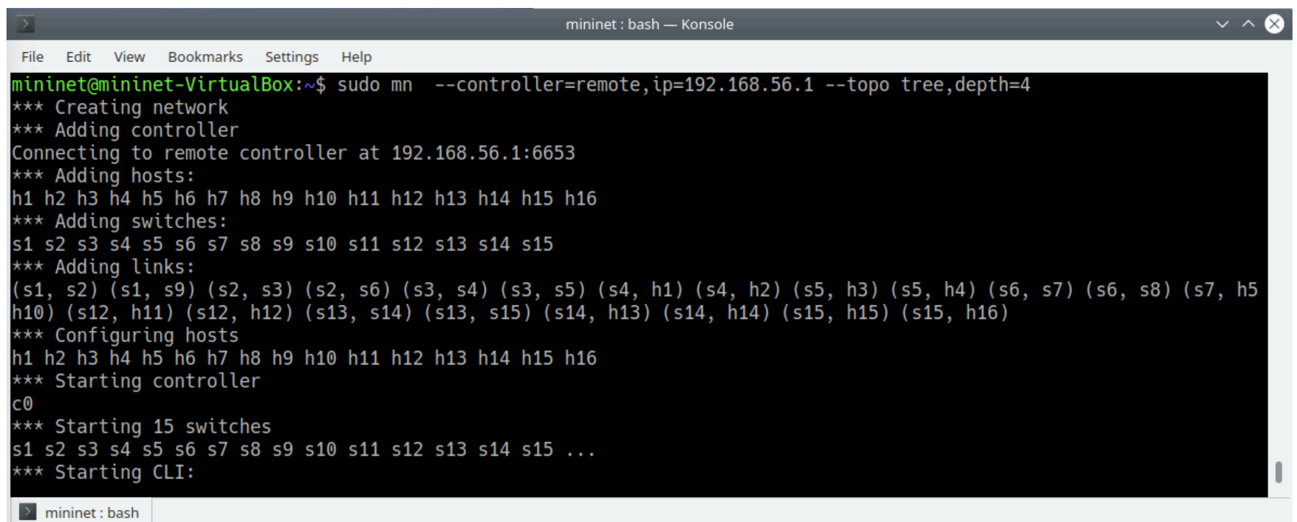



Рис. 3.2. Генерація топології в мережевому емуляторі mininet.

В цій роботі було задумано згенерувати більшу кількість вузлів, але, нажаль, було виявлено помилку в Ryu контролері або mininet, через яку на контролер не приходять пакети типу Packet-In, якщо кількість маршрутизаторів більше, ніж 15.

Далі було запущено контролер, реалізований на базі фреймворку Ryu. Він має архітектуру, що описана в Розділі 3.2. Після запуску контролеру з додатком qos_route_app, запускаються всі додатки, що пов'язані з ним (див. рис.3.3).

```
/usr/local/bin/python3.7 /Users/yaroslavkuznetsov/diploma/qos_routing_2/debug.py
debugging is available (--enable-debugger option is turned on)
loading app qos_route_app
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of NetworkAwareness
creating context network_awareness
instantiating app None of NetworkRouteDetector
creating context network_route_detector
instantiating app None of NetworkMonitor
creating context network_monitor
instantiating app None of NetworkDelayDetector
creating context network_delay_detector
instantiating app qos_route_app of QoSRouteApp
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
```

Рис. 3.2. Запуск Ryu контроллера з додатками.

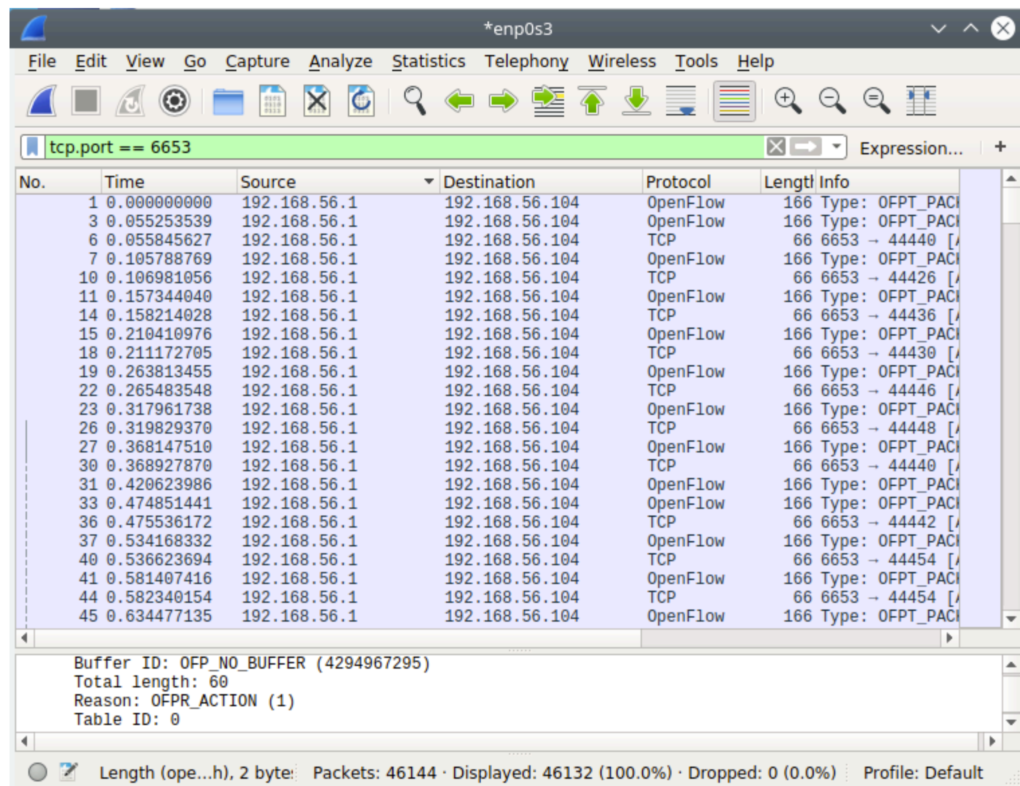



Рис. 3.3. Моніторинг мережі контролером.

Після запуску створення топології та запуску контролеру, необхідно прослідкувати за тим, що відбувається в мережі. Контролер збирає інформацію, що необхідна для збору метрик, як це зображено на рисунку 3.3. Як можна

помітити на графіку на рисунку 3.4, lldp трафік є достатньо інтенсивним. Це може стати причиною збільшення затримки на лінії.

Рис. 3.4. Графік швидкості lldp трафіку.

Якщо на маршрутизатор прийде пакет, поля якого не будуть відповідати жодному з правил, записаних в таблицях маршрутизатора, цей пакет буде відправлено на контролер у формі Packet-In. Для того, щоб продемонструвати, як себе поводить контролер у такому випадку, потрібно відправити якийсь запит до хоста, адресу якого маршрутизатор ще не знає. Найпростішим способом це зробити є відправити ICMP пакет за допомогою утиліти ping. Після чого маршрутизатор відправить ARP запит на всі з ним зв'язані маршрутизатори та на контролер. Контролер в свою чергу прийме даний пакет, обробить його та відправить у відповіть flow на даний контролер та на всі маршрутизатори на шляху до цільового маршрутизатора. Даний процес сображенo на рисунках 3.5 та 3.6.

```
*s1-eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
arp || icmp
No. Time        Source            Destination       Protocol Length Info
  5 4.623027047 ee:d6:9f:01:69:6a Broadcast         ARP      42 Who has 10.0.0.10? Tell 10.0.0.1
  6 4.623310101 86:f4:91:e9:ea:54 ee:d6:9f:01:69:6a ARP      42 10.0.0.10 is at 86:f4:91:e9:ea:54
  7 4.625763163 10.0.0.1          10.0.0.10         ICMP     98 Echo (ping) request  id=0x737b, se
  8 4.626123054 10.0.0.10         10.0.0.1          ICMP     98 Echo (ping) reply    id=0x737b, se
 10 5.601693600 10.0.0.1          10.0.0.10         ICMP     98 Echo (ping) request  id=0x737b, se
 11 5.601749282 10.0.0.10         10.0.0.1          ICMP     98 Echo (ping) reply    id=0x737b, se
 15 6.605858976 10.0.0.1          10.0.0.10         ICMP     98 Echo (ping) request  id=0x737b, se
 16 6.606159638 10.0.0.10         10.0.0.1          ICMP     98 Echo (ping) reply    id=0x737b, se

    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: ee:d6:9f:01:69:6a (ee:d6:9f:01:69:6a)
    Sender IP address: 10.0.0.1
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.0.0.10
Internet Control Message Protocol: Protocol
```

Рис. 3.5. Відправка ARP запиту до всіх сусідів.

```
*enp0s3
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
ip.src_host == 192.168.56.104 || ip.dst_host == 192.168.56.104                Expression...  +
ne            Source          Destination     Protocol  Length Info
).787231360 192.168.56.104  192.168.56.1    OpenFlow  150 Type: OFPT_PACKET_IN
).787480137 192.168.56.1    192.168.56.104  TCP        66 6653 → 44956 [ACK] Seq=14610 Ack=5986 Win=2046 Len=0
).802036476 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802082122 192.168.56.104  192.168.56.1    TCP        66 44960 → 6653 [ACK] Seq=10520 Ack=9714 Win=83 Len=0 T
).802180607 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802209663 192.168.56.104  192.168.56.1    TCP        66 44960 → 6653 [ACK] Seq=10520 Ack=9818 Win=83 Len=0 T
).802278207 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802364493 192.168.56.104  192.168.56.1    TCP        66 44960 → 6653 [ACK] Seq=10520 Ack=9922 Win=83 Len=0 T
).802472753 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802490423 192.168.56.104  192.168.56.1    TCP        66 44960 → 6653 [ACK] Seq=10520 Ack=10026 Win=83 Len=0
).802506407 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802514889 192.168.56.104  192.168.56.1    TCP        66 44942 → 6653 [ACK] Seq=15694 Ack=14414 Win=83 Len=0
).802589531 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802605355 192.168.56.104  192.168.56.1    TCP        66 44942 → 6653 [ACK] Seq=15694 Ack=14518 Win=83 Len=0
).802624790 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802630397 192.168.56.104  192.168.56.1    TCP        66 44942 → 6653 [ACK] Seq=15694 Ack=14622 Win=83 Len=0
).802640502 192.168.56.1    192.168.56.104  OpenFlow  170 Type: OFPT_FLOW_MOD
).802645067 192.168.56.104  192.168.56.1    TCP        66 44942 → 6653 [ACK] Seq=15694 Ack=14726 Win=83 Len=0

    Buffer ID: OFP_NO_BUFFER (4294967295)
    Out port: 0
    Out group: 0
  ▸ Flags: 0x0000
    Pad: 0000
  ▾ Match
      Type: OFPMT_OXM (1)
      Length: 26
    ▸ OXM field
    ▾ OXM field
        Class: OFPXMC_OPENFLOW_BASIC (0x8000)
        0001 011. = Field: OFPXMT_OFB_IPV4_SRC (11)
        .... ...0 = Has mask: False
        Length: 4
        Value: 10.0.0.10
    ▾ OXM field
        Class: OFPXMC_OPENFLOW_BASIC (0x8000)
        0001 100. = Field: OFPXMT_OFB_IPV4_DST (12)
        .... ...0 = Has mask: False
        Length: 4
        Value: 10.0.0.1
      Pad: 000000000000
  ▸ Instruction

0070  00 00 00 01 00 1a 80 00  0a 02 08 00 80 00 16 04   ........ ........
0080  0a 00 00 0a 80 00 18 04  0a 00 00 01 00 00 00 00   ........ ........
0090  00 00 00 04 00 18 00 00  00 00 00 00 00 10 00 00   ........ ........
00a0  00 01 ff e5 00 00 00 00  00 00                     ........ ..

Value (openflow_v4.oxm.value_ipv4addr), 4 bytes    Packets: 7939 · Displayed: 7937 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

Рис. 3.6. Отримання відповіді від контролера та подальша відправка пакетів до цільового маршрутизатора.

Рис. 3.7. Відповідь на запит маршрутизатора та внесення змін в flow-таблиці маршрутизаторів на шляху.

```
path -> [4, 3, 2, 1, 9, 10, 12]
Path installation finished in  0.006928920745849609
path -> [12, 10, 9, 1, 2, 3, 4]
Path installation finished in  0.003137826919555664
```

Після того, як контролер отримав запит на пошук шляху, він виконує пошук оптимального шляху, якщо метрики вже були прораховані, або ж прораховує найкоротший шлях, використовуючи в якості метрики кількість

Рис. 3.8. Шляхи, встановлені контролером.

переходів. Наступним кроком буде встановлення flow-таблиць на всьому шляху.

Тепер згенеруємо велику мережу з великою кількістю шляхів та способів переходу. Це буде топологію виду меш, з 15 маршрутизаторами, 30 хостів, як зображено на рис. 3.9.

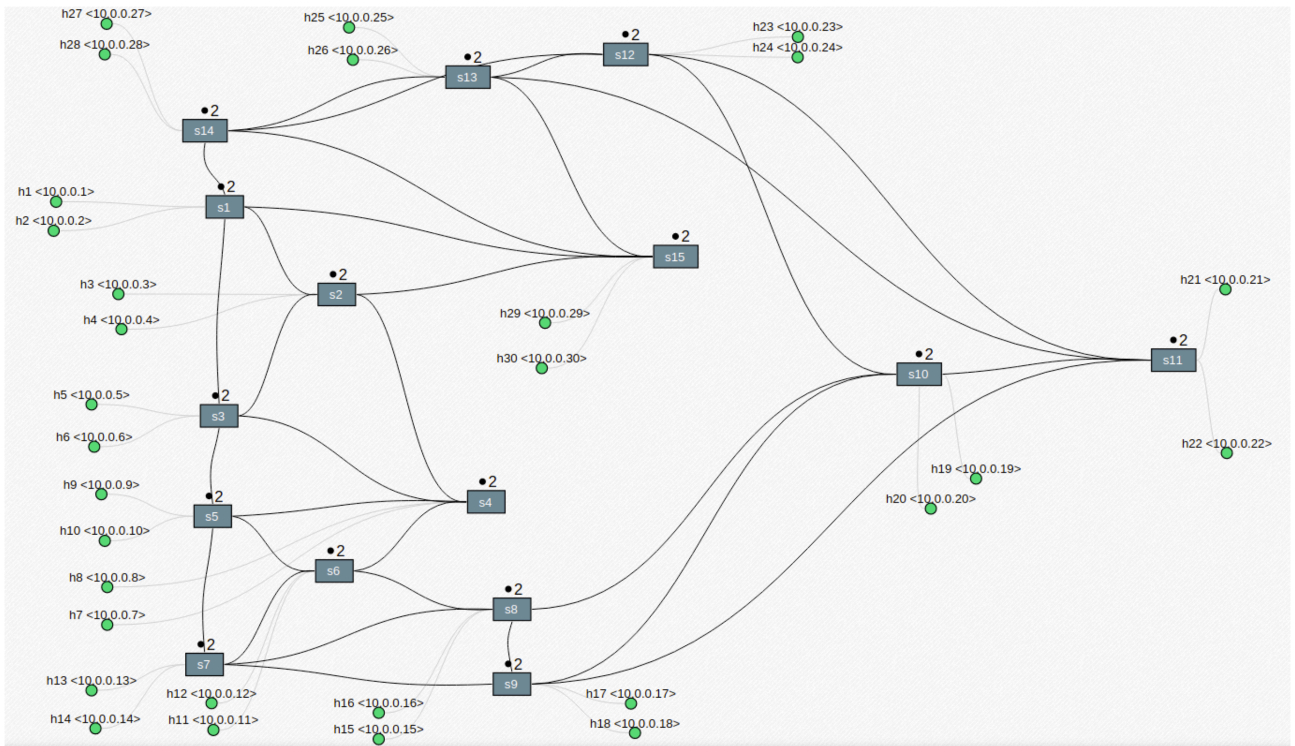



Рис. 3.9. Мережа для тестування балансування навантаження.

Після ініціалізації контролеру, спробуємо відправити ICMP пакети з h1 в h21. Для цього виконаємо команду в CLI mininet'у: h1 ping h21. Контролер знаходить опитмальний шлях з маршрутизатору s1 до маршрутизатору s11, результатом цього є шлях виду $S1 \to S14 \to S12 \to S11$ та зворотній шлях $S11 \to S12 \to S14 \to S1$. Результат виконання пошуку оптимального шляху зображено на рис. 3.8.

```
path -> [1, 14, 12, 11]
Path installation finished in  0.0015118122100830078
path -> [11, 12, 14, 1]
Path installation finished in  0.0035228729248046875
```

Рис. 3.8. Результат пошуку оптимального шляху.

Для того, щоб перевірити, чи справляється даний метод з задачею балансування трафіку в мережі, необхідно навантажити даний шлях. Для цього використаємо утиліту iperf. Запустимо tcp сервер на хості h21 та почнемо відправляти велику кількість агресивного tcp трафіку з h1 до h21 по попередньо встановленому маршруту. Це дасть змогу змінити метрики вздовж даного шляху, після чого значення оптимальності даного шляху зміниться.

Рис. 3.9. Tcp сервер на хості h21.

Рис. 3.10. Відправка агресивного tcp трафіку з h1 в h21.

Рис. 3.11. Результат пошуку оптимального шляху з h1 в h22.

Як видно з результатів виконання алгоритму пошуку оптимального шляху, було обрано змінено маршрут передачі пакетів з s1 в s11 для пари хостів h1 та h22. Алгоритм успішно виконує поставлену перед ним задачу, а саме, задачу балансування трафіку в масштабній мережі.

**Висновки до розділу**

Запропонований метод балансування трафіку в масштабних мережах було розроблено та змодельовано. Були виявлені практичні переваги та недоліки. Перевагами даного метода є:

1. Низька складність та висока швидкість пошуку коротшого шляху від маршрутизатора до маршрутизатора.
2. Роздільність розрахунку критеріїв оптимальності, згорток і т.д., та пошуку оптимального шляху по запиту на основі попередньо брахованих даних.

Недоліками даного метода є:

1. Зі збільшенням кількості маршрутизаторів в мережі збільшується кількість службового трафіку. Це згубно впливає на швидкість відповіді контролера на запити маршрутизаторів про шлях до іншого маршрутизатора. Необхідно змінити спосіб визначення затримки на полосі між маршрутизаторами.
2. Залежність від метрик та їх актуальності ускладнює визначення оптимальності шляху. Через те, що надмірна кількість службового трафіку може викликати неспроможність контролера відповідати на запити, неможливо мати завжди актуальні дані про стан мережі та оптимальність того чи іншого проходу.

# ВИСНОВКИ

Магістерська робота присвячена вирішенню задачі балансування трафіку в масштабних SDN мережах.

Було проведено теоретичний огляд та аналіз особливостей технології SDN, протоколу Openflow, моделей забеспечення QoS в класичних мережах та в мережах SDN.

Також було розглянуто існуючи способи маршрутизації та аргументовано можливості їх застосування в мережах, побудованих за архітектурою SDN. Була розглянута проблема балансування трафіку в SDN мережах, та можливий спосіб вирішення даної проблеми з використанням методу багатокритеріальної оптимізації. Проведене теоретичне дослідження та аналіз існуючих рішень підтвердив необхідність розробки нового підходу до вирішення задачі балансування трафіку в SDN мережах, у зв’язку з недостатнім використанням можливостей Openflow у традиційних підходах, що збільшує затримку при ремаршрутизації трафіку та не забезпеченням рівномірного навантаження мережі.

Було обрано та аргументовано вибір алгоритму пошуку шляхів, що базується на попередньому обрахуванні метрик, локальних критеріїв оптимальності, згорток та пошук опитмального шляху в графі, користуючись алгоритмом Белмана-Форда. Даний алгоритм направлений на забезпечення рівномірного завантаження каналів мережі завдяки направленню новых пакетів по оптимальному шляху.

Для перевірки результатів роботи алгоритму, було розроблено програмний модуль, розроблений на основі Ryu Framework, що виступає додатком контролера. Програма дозволяє робити моніторинг стану мережі, її метрик, таких як затримка, ширина полоси та завантаженість портів маршрутизаторів, розраховувати оптимальність шляху на всій площині рішень та знаходити оптимальний шлях, керувати маршрутизаторами та flow-таблицями.

Тестування роботи алгоритму показало ефективний вибір оптимального маршруту. Теоритичні та практичні дослідження показують, що алгоритм

ефективно справляється з вибором та встановленням оптимального шляху в мережі , що судить про вирішення проблеми балансування навантаження в SDN мережі.

**Література**


1. M. Boucadair "Software-Defined Networking: A Perspective from within a Service Provider Environment" [Електронний ресурс] / Boucadair // IETF RFC 7149 - Режим доступу до ресурсу: https://tools.ietf.org/html/rfc7149.

2. Campbell, A., De Meer, H., Kounavis, M., Miki, K., Vicente, J., and D. Villela, "A Survey of Programmable Networks", ACM SIGCOMM Computer Communi-cation Review, Volume 29, Issue 2, pp. 7-23, September 1992.

3. Open Networking Foundation. Режим доступу до ресурсу: https://www.opennetworking.org/.

4. Doria, A.; Salim, J.H.; Haas, R.; Khosravi, H.; Wang, W.; Dong, L.; Gopal, R.; Halpern, J. "Forwarding and Control Element Separation (ForCES) Protocol Specification." RFC 5810 (Proposed Standard), 2010. Режим доступу до ресурсу: https://datatracker.ietf.org/doc/rfc5810/

5. Yang, L.; Dantu, R.; Anderson, T.; Gopal, R. "Forwarding and Control Element Separation (ForCES) Framework." RFC 3746 (Informational), 2004. Режим доступу до ресурсу: https://datatracker.ietf.org/doc/rfc3746/

6. Lakshman, T.V.; Nandagopal, T.; Ramjee, R.; Sabnani, K.; Woo, T. The SoftRouter Architecture. In Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets), San Diego, CA, USA, 15–16 November 2004.

7. Zheng, H.; Zhang, X. Path Computation Element to Support Software-Defined Transport Networks Control. Internet Draft (Informational), 2014. Режим доступу до ресурсу: https://datatracker.ietf.org/doc/draft-zheng-pce-for-sdn-transport/

8. Hares, S. Analysis of Comparisons between OpenFlow and ForCES. Internet Draft (Informational), 2012. Режим доступу до ресурсу: https://datatracker.ietf.org/doc/draft-hares-forces-vs-openflow/

9. Open Networking Foundation. OpenFlow Switch Specification, Version 1.5.1 ( Protocol version 0x06 ). Режим доступу до ресурсу: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

10. Katta, N., Zhang, H., Freedman, M., Rexford, J.: Ravana: controller fault-tolerance in software-defined networking. In: 2015 1st ACM SIGCOMM Symposium on Software Defined Networking Research, pp. 4:1–4:12, June 2015

11. Kim, H., Santos, J.R., Turner, Y., Schlansker, M., Tourrihes, J., Feamster, N.:Coronet: fault tolerance for software defined networks. In: 2012 20th IEEE International Conference on Network Protocols (ICNP), pp. 1–2, October 2012

12. Tao Hu, Zehua Guo, Peng YI, Thar Baker and Julong Lan, "Multi-controller Based Software-Defined Networking: A Survey", Режим доступу до ресурсу: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8314783

13. D. Dotan and R. Y. Pinter, “HyperFlow: An integrated visual query and dataflow language for end-user information analysis,” in *Proc. IEEE Symp. Vis. Lang. Human-Centric Comput. (VL/HCC)*, Sep. 2005, pp. 27–34.

14. R. Y. Shtykh and T. Suzuki, “Distributed data stream processing with Onix,” in *Proc. IEEE 4th Int. Conf. Big Data Cloud Comput.*, Sydney, NSW, Australia, Dec. 2014, pp. 267–268

15. S. H. Yeganeh and Y. Ganjali, “Kandoo: A framework for efficient and scalable offloading of control applications,” in *Proc. 1st Workshop HotSDN*, 2012, pp. 19–24.

16. Microsoft Windows Server TechCenter, "What is QoS?," Microsoft Corporation, 2003. Режим доступу до ресурсу: http://www.microsoft.com/techne t/prodtechnol/windowsserver2003/library/TechRef/1c1f53a6-da9e-496f-be84- b91e2763dbeb.mspx

17. Cisco Systems Inc, Internetworking Technologies Handbook. Indianapolis: Cisco Press, 2003.

18. Dictionary.com, "Quality of Service," Lexico Publishing Group, LLC, 1998. Режим доступу до ресурсу: http://dictionary.reference.com/search?q=quality%20of%20 service.

19. Usman Ahmad, "QoS architectures: a detailed review", International Journal of Reviews in Computing, Sep. 2012, pp. 32-47

20. A. Danthine, O. Bonaventure, "From Best Effort to Enhanced QoS", Deliverable R2060/ULg/CIO/DS/P/004/b1 of the RACE CIO project, 51 p. (SART 93/15/15)

21. R. Braden "Integrated Services in the Internet Architecture: an Overview" RFC 1633 (Informational), 1994. Режим доступу до ресурсу: https://tools.ietf.org/html/rfc1633

22. Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z. and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, December 1998. Режим доступу до ресурсу: https://tools.ietf.org/html/rfc2475

23. D. Black, S. Brim, "Per Hop Behavior Identification Codes", [Електронний ресурс] // IETF RFC 3140 - Режим доступу до ресурсу: https://tools.ietf.org/html/rfc3140

24. D. Black, S. Brim, "Per Hop Behavior Identification Codes", [Електронний ресурс] // IETF RFC 3140 - Режим доступу до ресурсу: https://tools.ietf.org/html/rfc3140

25. J. Heinanen, F. Baker, Telia Finland, "Assured Forwarding PHB Group", [Електронний ресурс] // IETF RFC 2597 - Режим доступу до ресурсу: https://tools.ietf.org/html/rfc2597

26. B. Davie, A. Charny, "An Expedited Forwarding PHB (Per-Hop Behavior)", [Електронний ресурс] // IETF RFC 3246 - Режим доступу до ресурсу: https://tools.ietf.org/html/rfc3246

27. K. Nichols, S. Blake, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers" , [Електронний ресурс] // IETF RFC 3246 - Режим доступу до ресурсу: https://tools.ietf.org/html/rfc2474

28. Brodal, Gerth (1996). "Priority Queues on Parallel Machines". *Algorithm Theory - SWAT 96*: 416–427. doi:10.1007/3-540-61422-2_150.

29. "Сети для самых матёрых. Часть пятнадцатая. QoS", 18 августа 2018. Режим доступу до ресурсу: https://habr.com/ru/post/420525/

30. Y. Jinyao, Z. Hailong, S. Qianjun, L. Bo, and G. Xiao. "HiQoS: An SDN-based multipath QoS solution". In: *Communications, China* 12.5 (2015), pp. 123–133.

31. H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. "OpenQoS: An Open-Flow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks". In: *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*. IEEE. 2012, pp. 1–8.

32. M. R. Celenlioglu and H. A. Mantar. "An SDN Based Intra-Domain Routing and Re- source Management Model". In: *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE. 2015, pp. 347–352.

33. Marilia Curado, Edmundo Monteiro, "A Survey of QoS Routing Algorithms", In: Conference: International Conference on Information Technology (ICIT2004), January 2004

34. Q. Ma, P. Steenkiste, "Routing Traffic with Quality-of-Service Guarantees in Integrated Services Networks", Proceedings of Workshop on Network and Operating Systems Support for Digital Audio and Video, July 1998

35. F. Kuipers, T. Korkmaz, M. Krunz, P. Van Mieghem, "Overview of Constraint-Based Path Selection Algorithms for QoS Routing", IEEE Communications Magazine, special issue on IP-Oriented Quality of Service, December 2002.

36. A. Jüttner, B. Szviatovszki, I. Mécs, Z. Rajkó, "Lagrange Relaxation Based Method for the QoS Routing Problem", IEEE INFOCOM 2001, Anchorage, Alaska, April 22-26, 2001.

37. Mininet overview. Режим доступу до ресурсу: http://mininet.org/

38. Liehuang Zhu, Md Monjurul Karim, Kashif Sharif, Fan Li, Xiaojiang Du and Mohsen Guizani, "SDN Controllers: Benchmarking & Performance Evaluation", 12 Feb 2019

# ДОДАТОК А

mesh_topology.py

```
1.from mininet.topo import Topo
2.
3.class test(Topo):
4.    "Demo Setup"
5.
6.    def __init__( self, enable_all = True ):
7.        "Create custom topo."
8.
9.        Topo.__init__( self )
10.
11.        # Init values
12.        switches = 10   # total switchs
13.        cons = 2        # connections with next switch
14.        if cons >= switches:
15.                cons = switches - 1
16.        hosts = 10      # nodes per switch
17.
18.        # Create host and Switch
19.        # Add link :: host to switch
20.        for s_num in range(1,switches+1):
21.                switch = self.addSwitch("s%s" %(s_num))
22.                for h_num in range(1,hosts+1):
23.                        host = self.addHost("h%s" %(h_num + ((s_num - 1) * hosts)))
24.                        self.addLink(host,switch)
25.
```

```
26.        # Add link :: switch to switch
27.        for src in range(1,switches+1):
28.                for c_num in range(1,cons+1):
29.                        dst = src + c_num
30.                        if dst <= switches:
31.                                print("s%s" %src,"s%s" %dst)
32.                                self.addLink("s%s" %src,"s%s" %dst)
33.                        else:
34.                                dst = dst - switches
35.                                if src - dst > cons:
36.                                        print("s%s" %src,"s%s" %dst)
37.                                        self.addLink("s%s" %src,"s%s" %dst)
38.
39.topos = { 'test': ( lambda: test() ) }
```

qos_route_app.py

```
1.# conding=utf-8
2.import logging
3.import random
4.import struct
5.import time
6.from collections import defaultdict
7.
8.import networkx as nx
9.from operator import attrgetter
10.from ryu import cfg
11.from ryu.base import app_manager
```

```
12.from ryu.controller import ofp_event
13.from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
14.from ryu.controller.handler import CONFIG_DISPATCHER
15.from ryu.controller.handler import set_ev_cls
16.from ryu.ofproto import ofproto_v1_3
17.from ryu.lib import hub
18.from ryu.lib.packet import packet
19.from ryu.lib.packet import ethernet
20.from ryu.lib.packet import ipv4, ipv6
21.from ryu.lib.packet import arp
22.
23.from ryu.topology import event, switches
24.from ryu.topology.api import get_switch, get_link
25.
26.import network_awareness
27.import network_monitor
28.import network_delay_detector
29.import network_route_detector
30.import settings
31.CONF = cfg.CONF
32.
33.class QoSRouteApp(app_manager.RyuApp):
34.
35.    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
36.    _CONTEXTS = {
37.        "network_awareness": network_awareness.NetworkAwareness,
38.        "network_route_detector": network_route_detector.NetworkRouteDetector,
```

```
39.     "network_monitor": network_monitor.NetworkMonitor,
40.     "network_delay_detector": network_delay_detector.NetworkDelayDetector,
41. }
42.
43. WEIGHT_MODEL = {'hop': 'weight', 'delay': "delay", "bw": "bw"}
44.
45. def __init__(self, *args, **kwargs):
46.     super(QoSRouteApp, self).__init__(*args, **kwargs)
47.     self.name = 'shortest_forwarding'
48.     self.awareness = kwargs["network_awareness"]
49.     self.monitor = kwargs["network_monitor"]
50.     self.delay_detector = kwargs["network_delay_detector"]
51.     self.route_detector = kwargs['network_route_detector']
52.     self.hosts = {}
53.     self.datapaths = {}
54.     self.arp_table = {}
55.     self.threads = []
56.     self.switches = []
57.     self.adjacency = defaultdict(dict)
58.     self.multipath_group_ids = {}
59.     self.group_ids = []
60.
61.     # self.weight = self.WEIGHT_MODEL[CONF.weight]
62.     # self.monitor = hub.spawn(self._monitor)
63.
64. def add_ports_to_paths(self, paths, first_port, last_port):
65.     '''
```

```
66.     Add the ports that connects the switches for all paths
67.     '''
68.     paths_p = []
69.     for path in paths:
70.         p = {}
71.         in_port = first_port
72.         for s1, s2 in zip(path[:-1], path[1:]):
73.             out_port = self.adjacency[s1][s2]
74.             p[s1] = (in_port, out_port)
75.             in_port = self.adjacency[s2][s1]
76.             # print("sw -> %s , %s" % (s1, s2))
77.             # print("port -> %s -> %s" % (p[s1][0], p[s1][1]))
78.         p[path[-1]] = (in_port, last_port)
79.         paths_p.append(p)
80.     return paths_p
81.
82. def generate_openflow_gid(self):
83.     '''
84.     Returns a random OpenFlow group id
85.     '''
86.     n = random.randint(0, 2 ** 32)
87.     while n in self.group_ids:
88.         n = random.randint(0, 2 ** 32)
89.     return n
90.
91. def install_paths(self, src, first_port, dst, last_port, ip_src, ip_dst):
92.     computation_start = time.time()
```

```
93.     paths = [self.route_detector.find_optimal_path(src, dst)]
94.     # paths = self.get_optimal_paths(src, dst)
95.     # pw = []
96.     # for path in paths:
97.     #     pw.append(self.get_path_cost(path))
98.     #     print(path, "cost = ", pw[len(pw) - 1])
99.     # sum_of_pw = sum(pw) * 1.0
100.     # print( paths )
101.     paths_with_ports = self.add_ports_to_paths(paths, first_port, last_port)
102.     # print(paths_with_ports)
103.     switches_in_paths = set().union(*paths)
104.
105.     for node in switches_in_paths:
106.
107.         dp = self.datapaths[node]
108.         ofp = dp.ofproto
109.         ofp_parser = dp.ofproto_parser
110.
111.         ports = defaultdict(list)
112.         actions = []
113.         i = 0
114.
115.         for path in paths_with_ports:
116.             if node in path:
117.                 in_port = path[node][0]
118.                 out_port = path[node][1]
119.                 if (out_port, 1) not in ports[in_port]:
```

```
120.                    ports[in_port].append((out_port, 1))
121.                i += 1
122.
123.            for in_port in ports:
124.
125.                match_ip = ofp_parser.OFPMatch(
126.                    eth_type=0x0800,
127.                    ipv4_src=ip_src,
128.                    ipv4_dst=ip_dst
129.                )
130.                match_arp = ofp_parser.OFPMatch(
131.                    eth_type=0x0806,
132.                    arp_spa=ip_src,
133.                    arp_tpa=ip_dst
134.                )
135.
136.                out_ports = ports[in_port]
137.                # print out_ports
138.
139.                if len(out_ports) > 1:
140.                    group_id = None
141.                    group_new = False
142.
143.                    if (node, src, dst) not in self.multipath_group_ids:
144.                        group_new = True
145.                        self.multipath_group_ids[
146.                            node, src, dst] = self.generate_openflow_gid()
```

```
147.            group_id = self.multipath_group_ids[node, src, dst]
148.
149.            buckets = []
150.            # print "node at ",node," out ports : ",out_ports
151.            for port, weight in out_ports:
152.                bucket_weight = int(round(10))
153.                bucket_action = [ofp_parser.OFPActionOutput(port)]
154.                buckets.append(
155.                    ofp_parser.OFPBucket(
156.                        weight=bucket_weight,
157.                        watch_port=port,
158.                        watch_group=ofp.OFPG_ANY,
159.                        actions=bucket_action
160.                    )
161.                )
162.
163.            if group_new:
164.                req = ofp_parser.OFPGroupMod(
165.                    dp, ofp.OFPGC_ADD, ofp.OFPGT_SELECT, group_id,
166.                    buckets
167.                )
168.                dp.send_msg(req)
169.            else:
170.                req = ofp_parser.OFPGroupMod(
171.                    dp, ofp.OFPGC_MODIFY, ofp.OFPGT_SELECT,
172.                    group_id, buckets)
173.                dp.send_msg(req)
```

```
174.
175.                 actions = [ofp_parser.OFPActionGroup(group_id)]
176.
177.                 self.add_flow(dp, 32768, match_ip, actions)
178.                 self.add_flow(dp, 1, match_arp, actions)
179.
180.             elif len(out_ports) == 1:
181.                 actions = [ofp_parser.OFPActionOutput(out_ports[0][0])]
182.
183.                 self.add_flow(dp, 32768, match_ip, actions)
184.                 self.add_flow(dp, 1, match_arp, actions)
185.         print("Path installation finished in ", time.time() - computation_start)
186.         return paths_with_ports[0][src][1]
187.
188.     def add_flow(self, datapath, priority, match, actions, buffer_id=None):
189.         # print "Adding flow ", match, actions
190.         ofproto = datapath.ofproto
191.         parser = datapath.ofproto_parser
192.
193.         inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
194.                                              actions)]
195.         if buffer_id:
196.             mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
197.                                     priority=priority, match=match,
198.                                     instructions=inst)
199.         else:
200.             mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
```

```
201.                               match=match, instructions=inst)
202.         datapath.send_msg(mod)
203.
204.     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
205.     def _packet_in_handler(self, ev):
206.         msg = ev.msg
207.         datapath = msg.datapath
208.         ofproto = datapath.ofproto
209.         parser = datapath.ofproto_parser
210.         in_port = msg.match['in_port']
211.
212.         pkt = packet.Packet(msg.data)
213.         eth = pkt.get_protocol(ethernet.ethernet)
214.         arp_pkt = pkt.get_protocol(arp.arp)
215.
216.         # avoid broadcast from LLDP
217.         if eth.ethertype == 35020:
218.             return
219.
220.         if pkt.get_protocol(ipv6.ipv6):  # Drop the IPV6 Packets.
221.             match = parser.OFPMatch(eth_type=eth.ethertype)
222.             actions = []
223.             self.add_flow(datapath, 1, match, actions)
224.             return None
225.
226.         dst = eth.dst
227.         src = eth.src
```

```
228.    dpid = datapath.id
229.
230.    if src not in self.hosts:
231.        self.hosts[src] = (dpid, in_port)
232.
233.    out_port = ofproto.OFPP_FLOOD
234.
235.    if arp_pkt:
236.        # print dpid, pkt
237.        src_ip = arp_pkt.src_ip
238.        dst_ip = arp_pkt.dst_ip
239.        if arp_pkt.opcode == arp.ARP_REPLY:
240.            self.arp_table[src_ip] = src
241.            h1 = self.hosts[src]
242.            h2 = self.hosts[dst]
243.            out_port = self.install_paths(h1[0], h1[1], h2[0], h2[1], src_ip, dst_ip)
244.            self.install_paths(h2[0], h2[1], h1[0], h1[1], dst_ip, src_ip)  # reverse
245.        elif arp_pkt.opcode == arp.ARP_REQUEST:
246.            if dst_ip in self.arp_table:
247.                self.arp_table[src_ip] = src
248.                dst_mac = self.arp_table[dst_ip]
249.                h1 = self.hosts[src]
250.                h2 = self.hosts[dst_mac]
251.                out_port = self.install_paths(h1[0], h1[1], h2[0], h2[1], src_ip,
dst_ip)
252.                self.install_paths(h2[0], h2[1], h1[0], h1[1], dst_ip, src_ip)  # reverse
253.
```

```
254.        # print pkt
255.
256.        actions = [parser.OFPActionOutput(out_port)]
257.
258.        data = None
259.        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
260.            data = msg.data
261.
262.        out = parser.OFPPacketOut(
263.            datapath=datapath, buffer_id=msg.buffer_id, in_port=in_port,
264.            actions=actions, data=data)
265.        datapath.send_msg(out)
266.
267.    @set_ev_cls(event.EventSwitchEnter)
268.    def switch_enter_handler(self, ev):
269.        switch = ev.switch.dp
270.        ofp_parser = switch.ofproto_parser
271.
272.        if switch.id not in self.switches:
273.            self.switches.append(switch.id)
274.            self.datapaths[switch.id] = switch
275.
276.            # Request port/link descriptions, useful for obtaining bandwidth
277.            # req = ofp_parser.OFPPortDescStatsRequest(switch)
278.            # switch.send_msg(req)
279.
280.    @set_ev_cls(event.EventSwitchLeave, MAIN_DISPATCHER)
```

```
281.  def switch_leave_handler(self, ev):
282.      print(ev)
283.      switch = ev.switch.dp.id
284.      if switch in self.switches:
285.          self.switches.remove(switch)
286.          del self.datapaths[switch]
287.          del self.adjacency[switch]
288.
289.  @set_ev_cls(event.EventLinkAdd, MAIN_DISPATCHER)
290.  def link_add_handler(self, ev):
291.      s1 = ev.link.src
292.      s2 = ev.link.dst
293.      self.adjacency[s1.dpid][s2.dpid] = s1.port_no
294.      self.adjacency[s2.dpid][s1.dpid] = s2.port_no
295.
296.  @set_ev_cls(event.EventLinkDelete, MAIN_DISPATCHER)
297.  def link_delete_handler(self, ev):
298.      s1 = ev.link.src
299.      s2 = ev.link.dst
300.      # Exception handling if switch already deleted
301.      try:
302.          del self.adjacency[s1.dpid][s2.dpid]
303.          del self.adjacency[s2.dpid][s1.dpid]
304.      except KeyError:
305.          pass
306.
307.
```

network_route_detector.py

```
1.from __future__ import division
2.
3.import copy
4.
5.from ryu import cfg
6.from ryu.base import app_manager
7.from ryu.base.app_manager import lookup_service_brick
8.from ryu.controller import ofp_event
9.from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
10.from ryu.controller.handler import set_ev_cls
11.from ryu.ofproto import ofproto_v1_3
12.from ryu.lib import hub
13.from ryu.topology.switches import Switches
14.from ryu.topology.switches import LLDPPacket
15.import networkx as nx
16.import time
17.import settings
18.
19.CONF = {
20.    "weight": 'delay'
21.}
22.
23.
24.class NetworkRouteDetector(app_manager.RyuApp):
25.    """
26.        NetworkRouteDetector is a Ryu app for collecting link delay.
```

```
27. """
28.
29. OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
30.
31. def __init__(self, *args, **kwargs):
32.     super(NetworkRouteDetector, self).__init__(*args, **kwargs)
33.     self.name = 'network_route_detector'
34.     self.awareness = lookup_service_brick('awareness')
35.
36.     self.delay_graph = {}
37.     self.prev_delay_graph = {}
38.     self.delay_graph_safe = {}
39.     self.prev_delay_graph_safe = {}
40.
41.     self.bandwidth_graph = {}
42.     self.prev_bandwidth_graph = {}
43.     self.bandwidth_graph_safe = {}
44.     self.prev_bandwidth_graph_safe = {}
45.
46.     self.speed_graph = {}
47.     self.prev_speed_graph = {}
48.     self.speed_graph_safe = {}
49.     self.prev_speed_graph_safe = {}
50.
51.     self.optimal_criterias_delay_graph = {}
52.     self.optimal_criterias_speed_graph = {}
53.     self.optimal_criterias_bandwidth_graph = {}
```

```
54.
55.     self.optimality_graph = {}
56.     self.prev_optimality_graph = {}
57.     self.reversed_optimality_graph = {}
58.     self.prev_reversed_optimality_graph ={}
59.
60.     self.max_delay = None
61.     self.max_bandwidth = None
62.     self.max_speed = None
63.     self.max_convolution = None
64.
65.     self.delay_updated = False
66.     self.bandwidth_updated = False
67.     self.speed_updated = False
68.     self.all_stats_updated = False
69.     self.monitor = hub.spawn(self._route_metrics_monitor)
70.     self.a1 = 0.3
71.     self.a2 = -0.3
72.     self.a3 = -0.3
73.
74. def _route_metrics_monitor(self):
75.     while True:
76.         # hub.sleep(settings.ROUTE_METRICS_COMPUTING_PERIOD)
77.         hub.sleep(10)
78.         self.calculate_metrics()
79.         hub.sleep(50)
80.
```

```
81.   def handle_bandwidth_update(self, bw_graph):
82.       # print('handle_bandwidth_update')
83.       self.bandwidth_updating()
84.       for src, dst_list in bw_graph.items():
85.           filtered_dst_list = {dst_node: dst_nodes for dst_node, dst_nodes in dst_list.items() if dst_node != src}
86.           self.update_bandwidth(src, filtered_dst_list)
87.           # print(src, dst_list)
88.       self.bandwidth_graph_safe = copy.copy(self.bandwidth_graph)
89.       self.prev_bandwidth_graph_safe = copy.copy(self.prev_bandwidth_graph)
90.       # print(self.bandwidth_graph_safe)
91.       self.bandwidth_is_updated()
92.       # print('bandwidth =>> %s' % self.prev_bandwidth_graph)
93.
94.   def update_bandwidth(self, src, dst_list):
95.       for dst, bandwidth in dst_list.items():
96.           self.update_parameter(src, dst, bandwidth, self.bandwidth_graph, self.prev_bandwidth_graph)
97.   """
98.       { src : { dst1: delay, dst2: delay} }
99.   """
100.  def handle_delay_update(self, delay_graph):
101.      self.delay_updating()
102.      for src, dst_list in delay_graph.items():
103.          filtered_dst_list = {dst_node: dst_nodes for dst_node, dst_nodes in dst_list.items() if dst_node != src}
104.          self.update_delay(src, filtered_dst_list)
105.
```

```
106.        self.delay_graph_safe = copy.copy(self.delay_graph)
107.        self.prev_delay_graph_safe = copy.copy(self.prev_delay_graph)
108.        # self.delay_graph
109.        # print('handle_delay_update')
110.        # print(self.bandwidth_graph_safe)
111.        self.delay_is_updated()
112.        # print('prev delay =>> %s' % self.prev_delay_graph)
113.        # print('delay =>> %s' % self.delay_graph)
114.
115.    def update_delay(self, src, dst_list):
116.        for dst, delay in dst_list.items():
117.                                self.update_parameter(src, dst, delay, self.delay_graph,
self.prev_delay_graph)
118.
119.    """
120.        dp_port -- datapath -> port
121.    """
122.    def handle_speed_update(self, dp_port, speed):
123.        # print('handle_speed_update')
124.        # (src_dpid, dst_dpid) => (src_port, dst_port)
125.        self.speed_updating()
126.        links = {links: ports for links, ports in self.awareness.link_to_port.items()
127.                if dp_port[0] in links
128.                  and dp_port[1] in ports}
129.
130.        self.update_speed(links, speed)
131.        self.speed_is_updated()
```

```
132.         # print('prev_speed =>> %s, dp_port =>> %s ' % (self.prev_speed_graph,
dp_port))
133.       # print('speed =>> %s, dp_port =>> %s ' % (self.speed_graph, dp_port))
134.
135.    def update_speed(self, links, speed):
136.
137.
138.       for link, ports in links.items():
139.          (src, dst) = link
140.          # (src_port, dst_port) = ports
141.                      self.update_parameter(src, dst, speed, self.speed_graph,
self.prev_speed_graph)
142.
143.       self.speed_graph_safe = copy.copy(self.speed_graph)
144.       self.prev_speed_graph_safe = copy.copy(self.prev_speed_graph)
145.
146.    def update_parameter(self, src, dst, value, graph, prev_graph):
147.       if graph.get(src) is None:
148.          graph.setdefault(src, {})
149.
150.       if prev_graph.get(src) is None:
151.          prev_graph.setdefault(src, {})
152.
153.       if graph.get(src).get(dst) is None:
154.          graph[src][dst] = 0
155.          # updated_graph[src][dst] = False
156.
157.       prev_val = graph.get(src).get(dst)
```

```
158.        prev_graph[src][dst] = prev_val
159.
160.        if self.need_to_update(prev_val, value):
161.            graph[src][dst] = value
162.            # updated_graph[src][dst] = True
163.            # add watcher here
164.
165.    def local_convolution(self, z1, z2, z3):
166.        if z1 is None: z1 = 0
167.        if z2 is None: z2 = 0
168.        if z3 is None: z3 = 0
169.        return self.a1 * z1 + self.a2 * z2 + self.a3 * z3
170.
171.    def local_optimal_criteria(self, x, x_max):
172.        if x_max == 0: return 0
173.        return x / x_max
174.
175.    def local_reversed_convolution(self, w, w_max):
176.        return w_max - w + 1
177.
178.    def calc_optimal_criterias(self):
179.
180.        self.calc_optimal_criterias_universal(
181.            self.prev_bandwidth_graph_safe,
182.            self.bandwidth_graph_safe,
183.            self.max_bandwidth,
184.            self.optimal_criterias_bandwidth_graph
```

```
185.     )
186.
187.     self.calc_optimal_criterias_universal(
188.         self.prev_delay_graph_safe,
189.         self.delay_graph_safe,
190.         self.max_delay,
191.         self.optimal_criterias_delay_graph
192.     )
193.
194.     self.calc_optimal_criterias_universal(
195.         self.prev_speed_graph_safe,
196.         self.speed_graph_safe,
197.         self.max_speed,
198.         self.optimal_criterias_speed_graph
199.     )
200.
201.     # for src, dist_list in self
202.
203. def find_max(self, graph):
204.     max_val = None
205.     for src, dist_list in graph.items():
206.         curr_max_link = max(dist_list, key=lambda dst: dist_list[dst])
207.         curr_max_val = dist_list[curr_max_link]
208.         if max_val is None or max_val < curr_max_val:
209.             max_val = curr_max_val
210.     return max_val
211.
```

```
212.     def calc_optimal_criterias_universal(self, prev_graph, cur_graph, prev_max, optimal_criteria_graph):
213.         max_val = self.find_max(cur_graph)
214.         # prev_max = max_val
215.         for src, dst_list in cur_graph.items():
216.             if optimal_criteria_graph.get(src) is None:
217.                 optimal_criteria_graph.setdefault(src, {})
218.             for dst, val in dst_list.items():
219.                 if max_val != prev_max or prev_graph[src][dst] != cur_graph[src][dst]:
220.                     optimal_criteria_graph[src][dst] = self.local_optimal_criteria(cur_graph[src][dst], max_val)
221.
222.         prev_max = max_val
223.
224.     def calc_convolutions(self):
225.         self.prev_optimality_graph = copy.copy(self.optimality_graph)
226.         for src, dst_list in self.optimal_criterias_bandwidth_graph.items():
227.             if self.optimality_graph.get(src) is None:
228.                 self.optimality_graph.setdefault(src, {})
229.             if self.optimal_criterias_bandwidth_graph.get(src) is None:
230.                 self.optimal_criterias_bandwidth_graph.setdefault(src, {})
231.             if self.optimal_criterias_delay_graph.get(src) is None:
232.                 self.optimal_criterias_delay_graph.setdefault(src, {})
233.             if self.optimal_criterias_speed_graph.get(src) is None:
234.                 self.optimal_criterias_speed_graph.setdefault(src, {})
235.
236.             for dst, val in dst_list.items():
237.                 # print(src, dst, self.optimal_criterias_delay_graph)
```

```
238.             self.optimality_graph[src][dst] = \
239.                 self.local_convolution(
240.                     self.optimal_criterias_bandwidth_graph.get(src).get(dst),
241.                     self.optimal_criterias_delay_graph.get(src).get(dst),
242.                     self.optimal_criterias_speed_graph.get(src).get(dst)
243.                 )
244.
245.     def calc_reversed_convolutions(self):
246.         max_conv = self.find_max(self.optimality_graph)
247.         for src, dst_list in self.optimality_graph.items():
248.             if self.reversed_optimality_graph.get(src) is None:
249.                 self.reversed_optimality_graph.setdefault(src, {})
250.             for dst, val in dst_list.items():
251.                 if self.max_convolution != max_conv or \
252.                         self.optimality_graph[src][dst] != self.prev_optimality_graph[src]
[dst]:
253.                     self.reversed_optimality_graph[src][dst] = \
254.                         self.local_reversed_convolution(self.optimality_graph[src][dst],
max_conv)
255.         self.max_convolution = max_conv
256.
257.         # print(self.prev_reversed_optimality_graph)
258.         self.prev_reversed_optimality_graph =
copy.copy(self.reversed_optimality_graph)
259.
260.     def graph_to_matrix(self, graph):
261.         switch_ids = graph.keys()
262.         switch_ids_sorted = sorted(switch_ids)
```

```
263.        last_switch_id = switch_ids_sorted[-1]
264.        matrix_graph = [[float("Inf")] * last_switch_id] * last_switch_id
265.        for src, dst_list in graph.items():
266.            for dst, val in dst_list.items():
267.                matrix_graph[src-1][dst-1] = graph.get(src).get(val)
268.        return matrix_graph
269.
270.    def find_optimal_path(self, src, dst):
271.        # graph = self.graph_to_matrix(self.reversed_optimality_graph)
272.        graph = self.prev_reversed_optimality_graph
273.        if len(graph.keys()) == 0:
274.            graph = self.awareness.dict_graph
275.
276.        distance, predecessor = dict(), dict()
277.        for node in graph:
278.            distance[node], predecessor[node] = float('inf'), None
279.        distance[src] = 0
280.
281.        for _ in range(len(graph) - 1):
282.            for node in graph:
283.                for neighbour in graph[node]:
284.                    if distance[neighbour] > distance[node] + graph[node][neighbour]:
285.                        distance[neighbour] = distance[node] + graph[node][neighbour]
286.                        predecessor[neighbour] = node
287.            # Step 3: Check for negative weight cycles
288.        for node in graph:
289.            for neighbour in graph[node]:
```

```
290.                        assert distance[neighbour] <= distance[node] + graph[node]
[neighbour], "Negative weight cycle."
291.        # print(distance)
292.        # print(predecessor)
293.
294.        current = dst
295.        path = []
296.        while current is not None:
297.            path.append(current)
298.            if predecessor.get(current) is None:
299.                break
300.            current = predecessor[current]
301.
302.        rev_path = list(reversed(path))
303.        print("path -> %s" % rev_path)
304.        return rev_path
305.
306.    def calculate_metrics(self):
307.            self.all_stats_updated = (self.bandwidth_updated and self.delay_updated
and self.speed_updated)
308.        # print(self.bandwidth_updated , self.delay_updated, self.speed_updated)
309.        if self.all_stats_updated is True:
310.            self.calc_optimal_criterias()
311.            self.calc_convolutions()
312.            self.calc_reversed_convolutions()
313.            # self.find_optimal_path(6)
314.            print('stats calculated')
315.            self.clear_stats()
```

```
316.
317.    def need_to_update(self, prev_val, val):
318.        prev_val_more = prev_val + prev_val / 15
319.        prev_val_less = prev_val - prev_val / 15
320.        return val <= prev_val_less or val >= prev_val_more
321.
322.    def clear_stats(self):
323.        self.all_stats_updated = self.bandwidth_updated = False
324.        self.delay_updated = self.speed_updated = False
325.        # print(' we are here ', self.updated_delay_graph)
326.
327.    def bandwidth_updating(self):
328.        self.bandwidth_updated = False
329.
330.    def bandwidth_is_updated(self):
331.        self.bandwidth_updated = True
332.
333.    def delay_updating(self):
334.        self.delay_updated = False
335.
336.    def delay_is_updated(self):
337.        self.delay_updated = True
338.
339.    def speed_updating(self):
340.        self.speed_updated = False
341.
342.    def speed_is_updated(self):
```

```
343.        self.speed_updated = True
```

network_monitor.py

```
1.import copy
2.from operator import attrgetter
3.from ryu import cfg
4.from ryu.base import app_manager
5.from ryu.base.app_manager import lookup_service_brick
6.from ryu.controller import ofp_event
7.from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
8.from ryu.controller.handler import CONFIG_DISPATCHER
9.from ryu.controller.handler import set_ev_cls
10.from ryu.ofproto import ofproto_v1_3
11.from ryu.lib import hub
12.from ryu.lib.packet import packet
13.import settings
14.from printdebug import debug, debug_enable
15.
16.from printdebug import printobject
17.import pdb
18.
19.CONF = {
20.    "weight": 'bw'
21.}
22.
23.
24.class NetworkMonitor(app_manager.RyuApp):
```

```
25. """
26.     NetworkMonitor is a Ryu app for collecting traffic information.
27. """
28. OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
29.
30. def __init__(self, *args, **kwargs):
31.     super(NetworkMonitor, self).__init__(*args, **kwargs)
32.     self.name = 'monitor'
33.     self.datapaths = {}
34.     self.port_stats = {}
35.     self.port_speed = {}
36.     self.flow_stats = {}
37.     self.flow_speed = {}
38.     self.stats = {}
39.     self.port_features = {}
40.     self.free_bandwidth = {}
41.     self.awareness = lookup_service_brick('awareness')
42.                                 self.network_route_detector =
lookup_service_brick('network_route_detector')
43.     self.graph = None
44.     self.capabilities = None
45.     self.best_paths = None
46.     # Start to green thread to monitor traffic and calculating
47.     # free bandwidth of links respectively.
48.     self.monitor_thread = hub.spawn(self._monitor)
49.     self.save_freebandwidth_thread = hub.spawn(self._save_bw_graph)
50.
```

```
51.  @set_ev_cls(ofp_event.EventOFPStateChange,
52.              [MAIN_DISPATCHER, DEAD_DISPATCHER])
53.  def _state_change_handler(self, ev):
54.      """
55.          Record datapath's info
56.      """
57.      datapath = ev.datapath
58.      if ev.state == MAIN_DISPATCHER:
59.          if not datapath.id in self.datapaths:
60.              self.logger.debug('register datapath: %016x', datapath.id)
61.              self.datapaths[datapath.id] = datapath
62.      elif ev.state == DEAD_DISPATCHER:
63.          if datapath.id in self.datapaths:
64.              self.logger.debug('unregister datapath: %016x', datapath.id)
65.              del self.datapaths[datapath.id]
66.
67.
68.  def _monitor(self):
69.      """
70.          Main entry method of monitoring traffic.
71.      """
72.      hub.sleep(3)
73.      while CONF.get('weight') == 'bw':
74.          self.stats['flow'] = {}
75.          self.stats['port'] = {}
76.          for dp in self.datapaths.values():
77.              self.port_features.setdefault(dp.id, {})
```

```
78.             self._request_stats(dp)
79.             # refresh data.
80.             self.capabilities = None
81.             self.best_paths = None
82.         hub.sleep(settings.METRICS_DISCOVERING_PERIOD)
83.
84.
85.   def _save_bw_graph(self):
86.       """
87.         Save bandwidth data into networkx graph object.
88.       """
89.       while CONF.get('weight') == 'bw':
90.         self.graph = self.create_bw_graph(self.free_bandwidth)
91.         bw_graph = self.create_bw_only_graph(self.graph)
92.         self.logger.debug("save_freebandwidth")
93.         self.network_route_detector.handle_bandwidth_update(bw_graph)
94.         # print('herere')
95.         hub.sleep(settings.METRICS_DISCOVERING_PERIOD)
96.
97.
98.   def _request_stats(self, datapath):
99.       """
100.          Sending request msg to datapath
101.      """
102.      self.logger.debug('send stats request: %016x', datapath.id)
103.      ofproto = datapath.ofproto
104.      parser = datapath.ofproto_parser
```

```
105.
106.        req = parser.OFPPortDescStatsRequest(datapath, 0)
107.        datapath.send_msg(req)
108.
109.        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
110.        datapath.send_msg(req)
111.        # print('requested port stats')
112.        req = parser.OFPFlowStatsRequest(datapath)
113.        datapath.send_msg(req)
114.
115.    def get_min_bw_of_links(self, graph, path, min_bw):
116.        """
117.            Getting bandwidth of path. Actually, the mininum bandwidth
118.            of links is the bandwith, because it is the neck bottle of path.
119.        """
120.        _len = len(path)
121.        if _len > 1:
122.            minimal_band_width = min_bw
123.            for i in range(_len-1):
124.                pre, curr = path[i], path[i+1]
125.                if 'bandwidth' in graph[pre][curr]:
126.                    bw = graph[pre][curr]['bandwidth']
127.                    minimal_band_width = min(bw, minimal_band_width)
128.                else:
129.                    continue
130.            return minimal_band_width
131.        return min_bw
```

```
132.
133.    def get_best_path_by_bw(self, graph, paths):
134.        """
135.            Get best path by comparing paths.
136.        """
137.        capabilities = {}
138.        best_paths = copy.deepcopy(paths)
139.
140.        for src in paths:
141.            for dst in paths[src]:
142.                if src == dst:
143.                    best_paths[src][src] = [src]
144.                    capabilities.setdefault(src, {src: settings.MAX_CAPACITY})
145.                    capabilities[src][src] = settings.MAX_CAPACITY
146.                    continue
147.                max_bw_of_paths = 0
148.                best_path = paths[src][dst][0]
149.                for path in paths[src][dst]:
150.                    min_bw = settings.MAX_CAPACITY
151.                    min_bw = self.get_min_bw_of_links(graph, path, min_bw)
152.                    if min_bw > max_bw_of_paths:
153.                        max_bw_of_paths = min_bw
154.                        best_path = path
155.
156.                best_paths[src][dst] = best_path
157.                capabilities.setdefault(src, {dst: max_bw_of_paths})
158.                capabilities[src][dst] = max_bw_of_paths
```

```
159.        self.capabilities = capabilities
160.        self.best_paths = best_paths
161.        return capabilities, best_paths
162.
163.    def create_bw_only_graph(self, graph):
164.        # for src, dist_list in list(graph.nodes):
165.        bw_only = {}
166.        for src, dst, data in graph.edges.data():
167.            if bw_only.get(src) is None:
168.                bw_only.setdefault(src, {})
169.
170.            bw_only[src][dst] = data.get('bandwidth', 0)
171.
172.        return bw_only
173.
174.    def create_bw_graph(self, bw_dict):
175.        """
176.            Save bandwidth data into networkx graph object.
177.        """
178.        try:
179.            graph = self.awareness.graph
180.            link_to_port = self.awareness.link_to_port
181.            for link in link_to_port:
182.                (src_dpid, dst_dpid) = link
183.                (src_port, dst_port) = link_to_port[link]
184.                if src_dpid in bw_dict and dst_dpid in bw_dict:
185.                    bw_src = bw_dict[src_dpid][src_port]
```

```
186.                bw_dst = bw_dict[dst_dpid][dst_port]
187.                bandwidth = min(bw_src, bw_dst)
188.                # add key:value of bandwidth into graph.
189.                graph[src_dpid][dst_dpid]['bandwidth'] = bandwidth
190.            else:
191.                graph[src_dpid][dst_dpid]['bandwidth'] = 0
192.        return graph
193.    except:
194.        self.logger.info("Create bw graph exception")
195.        if self.awareness is None:
196.            self.awareness = lookup_service_brick('awareness')
197.        return self.awareness.graph
198.
199. def _save_freebandwidth(self, dpid, port_no, speed):
200.    # Calculate free bandwidth of port and save it.
201.    port_state = self.port_features.get(dpid).get(port_no)
202.    if port_state:
203.        capacity = port_state[2]
204.        curr_bw = self._get_free_bw(capacity, speed)
205.        self.free_bandwidth[dpid].setdefault(port_no, None)
206.        self.free_bandwidth[dpid][port_no] = curr_bw
207.    else:
208.        self.logger.info("Fail in getting port state")
209.
210. def _save_stats(self, _dict, key, value, length):
211.    if key not in _dict:
212.        _dict[key] = []
```

```
213.        _dict[key].append(value)
214.
215.        if len(_dict[key]) > length:
216.            _dict[key].pop(0)
217.
218.    def _get_speed(self, now, pre, period):
219.        if period:
220.            return (now - pre) / (period)
221.        else:
222.            return 0
223.
224.    def _get_free_bw(self, capacity, speed):
225.        # BW:Mbit/s
226.        return max(capacity/10**3 - speed * 8/10**6, 0)
227.
228.    def _get_time(self, sec, nsec):
229.        return sec + nsec / (10 ** 9)
230.
231.    def _get_period(self, n_sec, n_nsec, p_sec, p_nsec):
232.        return self._get_time(n_sec, n_nsec) - self._get_time(p_sec, p_nsec)
233.
234.    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
235.    def _flow_stats_reply_handler(self, ev):
236.        """
237.            Save flow stats reply info into self.flow_stats.
238.            Calculate flow speed and Save it.
239.        """
```

```
240.        body = ev.msg.body
241.        dpid = ev.msg.datapath.id
242.        self.stats['flow'][dpid] = body
243.        self.flow_stats.setdefault(dpid, {})
244.        self.flow_speed.setdefault(dpid, {})
245.        for stat in sorted([flow for flow in body if flow.priority == 1],
246.                           key=lambda flow: (flow.match.get('in_port'),
247.                                             flow.match.get('ipv4_dst'))):
248.            if len(stat.instructions) != 0 \
249.                    and len(stat.instructions[0].actions) != 0:
250.                key = (stat.match.get('in_port'),  stat.match.get('ipv4_dst'),
251.                       stat.instructions[0].actions[0].port)
252.                value = (stat.packet_count, stat.byte_count,
253.                         stat.duration_sec, stat.duration_nsec)
254.                self._save_stats(self.flow_stats[dpid], key, value, 5)
255.
256.                # Get flow's speed.
257.                pre = 0
258.                period = settings.METRICS_DISCOVERING_PERIOD
259.                tmp = self.flow_stats[dpid][key]
260.                if len(tmp) > 1:
261.                    pre = tmp[-2][1]
262.                    period = self._get_period(tmp[-1][2], tmp[-1][3],
263.                                              tmp[-2][2], tmp[-2][3])
264.
265.                speed = self._get_speed(self.flow_stats[dpid][key][-1][1],
266.                                        pre, period)
```

```
267.
268.             self._save_stats(self.flow_speed[dpid], key, speed, 5)
269.
270.     @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
271.     def _port_stats_reply_handler(self, ev):
272.         """
273.             Save port's stats info
274.             Calculate port's speed and save it.
275.         """
276.         # print("start speed count")
277.         body = ev.msg.body
278.         dpid = ev.msg.datapath.id
279.         self.stats['port'][dpid] = body
280.         self.free_bandwidth.setdefault(dpid, {})
281.
282.         for stat in sorted(body, key=attrgetter('port_no')):
283.             port_no = stat.port_no
284.             if port_no != ofproto_v1_3.OFPP_LOCAL:
285.                 key = (dpid, port_no)
286.                 value = (stat.tx_bytes, stat.rx_bytes, stat.rx_errors,
287.                          stat.duration_sec, stat.duration_nsec)
288.
289.                 self._save_stats(self.port_stats, key, value, 5)
290.
291.                 # Get port speed.
292.                 pre = 0
293.                 period = settings.METRICS_DISCOVERING_PERIOD
```

```
294.            tmp = self.port_stats[key]
295.            if len(tmp) > 1:
296.                pre = tmp[-2][0] + tmp[-2][1]
297.                period = self._get_period(tmp[-1][3], tmp[-1][4],
298.                                          tmp[-2][3], tmp[-2][4])
299.
300.            speed = self._get_speed(
301.                self.port_stats[key][-1][0] + self.port_stats[key][-1][1],
302.                pre, period)
303.
304.            self._save_stats(self.port_speed, key, speed, 5)
305.            self._save_freebandwidth(dpid, port_no, speed)
306.                            self.network_route_detector.handle_speed_update(key,
self.port_speed[key][-1])
307.        # ev.set()
308.
309.        @set_ev_cls(ofp_event.EventOFPPortDescStatsReply,
MAIN_DISPATCHER)
310.    def port_desc_stats_reply_handler(self, ev):
311.        """
312.            Save port description info.
313.        """
314.        msg = ev.msg
315.        dpid = msg.datapath.id
316.        ofproto = msg.datapath.ofproto
317.
318.        config_dict = {ofproto.OFPPC_PORT_DOWN: "Down",
319.                       ofproto.OFPPC_NO_RECV: "No Recv",
```

```
320.                ofproto.OFPPC_NO_FWD: "No Farward",
321.                ofproto.OFPPC_NO_PACKET_IN: "No Packet-in"}
322.
323.        state_dict = {ofproto.OFPPS_LINK_DOWN: "Down",
324.                ofproto.OFPPS_BLOCKED: "Blocked",
325.                ofproto.OFPPS_LIVE: "Live"}
326.
327.        ports = []
328.        for p in ev.msg.body:
329.            ports.append('port_no=%d hw_addr=%s name=%s config=0x%08x '
330.                    'state=0x%08x curr=0x%08x advertised=0x%08x '
331.                    'supported=0x%08x peer=0x%08x curr_speed=%d '
332.                    'max_speed=%d' %
333.                    (p.port_no, p.hw_addr,
334.                     p.name, p.config,
335.                     p.state, p.curr, p.advertised,
336.                     p.supported, p.peer, p.curr_speed,
337.                     p.max_speed))
338.
339.            if p.config in config_dict:
340.                config = config_dict[p.config]
341.            else:
342.                config = "up"
343.
344.            if p.state in state_dict:
345.                state = state_dict[p.state]
346.            else:
```

```
347.            state = "up"
348.
349.          port_feature = (config, state, p.curr_speed)
350.          self.port_features[dpid][p.port_no] = port_feature
351.
352.  @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
353.  def _port_status_handler(self, ev):
354.      """
355.          Handle the port status changed event.
356.      """
357.      msg = ev.msg
358.      reason = msg.reason
359.      port_no = msg.desc.port_no
360.      dpid = msg.datapath.id
361.      ofproto = msg.datapath.ofproto
362.
363.      reason_dict = {ofproto.OFPPR_ADD: "added",
364.                     ofproto.OFPPR_DELETE: "deleted",
365.                     ofproto.OFPPR_MODIFY: "modified", }
366.
367.      if reason in reason_dict:
368.
369.          print("switch%d: port %s %s" %
370.                (dpid, reason_dict[reason], port_no))
371.      else:
372.          print("switch%d: Illeagal port state %s %s" % (port_no, reason))
373.
```

```
374.    def show_stat(self, type):
375.        '''
376.            Show statistics info according to data type.
377.            type: 'port' 'flow'
378.        '''
379.        if settings.TOSHOW is False:
380.            return
381.
382.        bodys = self.stats[type]
383.        if(type == 'flow'):
384.            # printobject(bodys)
385.            print('datapath         ''   in-port        ip-dst      '
386.                  'out-port packets  bytes  flow-speed(B/s)')
387.            print('---------------- ''  -------- ----------------- '
388.                  '-------- -------- -------- -----------')
389.            for dpid in bodys.keys():
390.                for stat in sorted(
391.                    [flow for flow in bodys[dpid] if flow.priority == 1],
392.                    key=lambda flow: (flow.match.get('in_port'),
393.                                      flow.match.get('ipv4_dst'))):
394.                    print('%016x %8x %17s %8x %8d %8d %8.1f' % (
395.                        dpid,
396.                        stat.match['in_port'], stat.match['ipv4_dst'],
397.                        stat.instructions[0].actions[0].port,
398.                        stat.packet_count, stat.byte_count,
399.                        abs(self.flow_speed[dpid][
400.                            (stat.match.get('in_port'),
```

```
401.                         stat.match.get('ipv4_dst'),
402.                         stat.instructions[0].actions[0].port)][-1])))
403.            print('\n')
404.
405.        if(type == 'port'):
406.            print('datapath            port   ''rx-pkts  rx-bytes rx-error '
407.                  'tx-pkts  tx-bytes tx-error  port-speed(B/s)'
408.                  ' current-capacity(Kbps)  '
409.                  'port-stat   link-stat')
410.            print('----------------   -------- ''-------- -------- -------- '
411.                  '-------- -------- -------- '
412.                  '----------------  ----------------   '
413.                  '   -----------    -----------')
414.             format = '%016x %8x %8d %8d %8d %8d %8d %8d %8.1f %16d %16s
%16s'
415.            for dpid in bodys.keys():
416.                for stat in sorted(bodys[dpid], key=attrgetter('port_no')):
417.                    if stat.port_no != ofproto_v1_3.OFPP_LOCAL:
418.                        print(format % (
419.                            dpid, stat.port_no,
420.                            stat.rx_packets, stat.rx_bytes, stat.rx_errors,
421.                            stat.tx_packets, stat.tx_bytes, stat.tx_errors,
422.                            abs(self.port_speed[(dpid, stat.port_no)][-1]),
423.                            self.port_features[dpid][stat.port_no][2],
424.                            self.port_features[dpid][stat.port_no][0],
425.                            self.port_features[dpid][stat.port_no][1]))
426.            print('\n')
```

network_delay_detector.py

```
1.# Copyright (C) 2016 Li Cheng at Beijing University of Posts
2.# and Telecommunications. www.muzixing.com
3.#
4.# Licensed under the Apache License, Version 2.0 (the "License");
5.# you may not use this file except in compliance with the License.
6.# You may obtain a copy of the License at
7.#
8.# http://www.apache.org/licenses/LICENSE-2.0
9.#
10.# Unless required by applicable law or agreed to in writing, software
11.# distributed under the License is distributed on an "AS IS" BASIS,
12.# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
13.# implied.
14.# See the License for the specific language governing permissions and
15.# limitations under the License.
16.
17.from __future__ import division
18.from ryu import cfg
19.from ryu.base import app_manager
20.from ryu.base.app_manager import lookup_service_brick
21.from ryu.controller import ofp_event
22.from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
23.from ryu.controller.handler import set_ev_cls
24.from ryu.ofproto import ofproto_v1_3
25.from ryu.lib import hub
```

```
26.from ryu.topology.switches import Switches
27.from ryu.topology.switches import LLDPPacket
28.import networkx as nx
29.import time
30.import settings
31.
32.CONF = {
33.    "weight": 'delay'
34.}
35.
36.
37.class NetworkDelayDetector(app_manager.RyuApp):
38.    """
39.        NetworkDelayDetector is a Ryu app for collecting link delay.
40.    """
41.
42.    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
43.
44.    def __init__(self, *args, **kwargs):
45.        super(NetworkDelayDetector, self).__init__(*args, **kwargs)
46.        self.name = 'delaydetector'
47.        self.sending_echo_request_interval = 0.1
48.        # Get the active object of swicthes and awareness module.
49.        # So that this module can use their data.
50.        self.sw_module = lookup_service_brick('switches')
51.        self.awareness = lookup_service_brick('awareness')
```

```
52.                     self.network_route_detector =
lookup_service_brick('network_route_detector')
53.
54.     self.delay_graph = {}
55.     self.datapaths = {}
56.     self.echo_latency = {}
57.     self.measure_thread = hub.spawn(self._detector)
58.
59.  @set_ev_cls(ofp_event.EventOFPStateChange,
60.          [MAIN_DISPATCHER, DEAD_DISPATCHER])
61.  def _state_change_handler(self, ev):
62.     datapath = ev.datapath
63.     if ev.state == MAIN_DISPATCHER:
64.        if not datapath.id in self.datapaths:
65.           self.logger.debug('Register datapath: %016x', datapath.id)
66.           self.datapaths[datapath.id] = datapath
67.     elif ev.state == DEAD_DISPATCHER:
68.        if datapath.id in self.datapaths:
69.           self.logger.debug('Unregister datapath: %016x', datapath.id)
70.           del self.datapaths[datapath.id]
71.
72.  def _detector(self):
73.     """
74.        Delay detecting functon.
75.        Send echo request and calculate link delay periodically
76.     """
77.     hub.sleep(3)
```

```
78.     while CONF.get('weight') == 'delay':
79.         # print('start echo')
80.         self._send_echo_request()
81.         self.create_link_delay()
82.         try:
83.             self.awareness.shortest_paths = {}
84.             self.logger.debug("Refresh the shortest_paths")
85.         except:
86.             self.awareness = lookup_service_brick('awareness')
87.         #
88.         # self.show_delay_statis()
89.         self.create_delay_graph()
90.         # print('here it is !!!')
91.         self.network_route_detector.handle_delay_update(self.delay_graph)
92.         # hub.sleep(settings.DELAY_DETECTING_PERIOD)
93.         hub.sleep(settings.METRICS_DISCOVERING_PERIOD)
94.
95. def _send_echo_request(self):
96.     """
97.         Seng echo request msg to datapath.
98.     """
99.     # print(self.datapaths.values())
100.    for datapath in self.datapaths.values():
101.        parser = datapath.ofproto_parser
102.        echo_req = parser.OFPEchoRequest(datapath,
data="%.12f".encode('ascii') % time.time())
103.        datapath.send_msg(echo_req)
```

```
104.            # Important! Don't send echo request together, Because it will
105.            # generate a lot of echo reply almost in the same time.
106.            # which will generate a lot of delay of waiting in queue
107.            # when processing echo reply in echo_reply_handler.
108.            # print('echo %d' % datapath.id)
109.            hub.sleep(self.sending_echo_request_interval)
110.
111.    @set_ev_cls(ofp_event.EventOFPEchoReply, MAIN_DISPATCHER)
112.    def echo_reply_handler(self, ev):
113.        """
114.            Handle the echo reply msg, and get the latency of link.
115.        """
116.        now_timestamp = time.time()
117.        try:
118.            latency = now_timestamp - eval(ev.msg.data)
119.            self.echo_latency[ev.msg.datapath.id] = latency
120.        except:
121.            return
122.
123.    def get_delay(self, src, dst):
124.        """
125.            Get link delay.
126.                        Controller
127.                        |       |
128.        src echo latency|       |dst echo latency
129.                        |       |
130.                   SwitchA-------SwitchB
```

```
131.
132.              fwd_delay--->
133.                 <----reply_delay
134.          delay = (forward delay + reply delay - src datapath's echo latency
135.      """
136.      try:
137.          fwd_delay = self.awareness.graph[src][dst]['lldpdelay']
138.          re_delay = self.awareness.graph[dst][src]['lldpdelay']
139.          src_latency = self.echo_latency[src]
140.          dst_latency = self.echo_latency[dst]
141.
142.          delay = (fwd_delay + re_delay - src_latency - dst_latency)/2
143.          return max(delay, 0)
144.      except:
145.          return float('inf')
146.
147.  def _save_lldp_delay(self, src=0, dst=0, lldpdelay=0):
148.      try:
149.          self.awareness.graph[src][dst]['lldpdelay'] = lldpdelay
150.      except:
151.          if self.awareness is None:
152.              self.awareness = lookup_service_brick('awareness')
153.          return
154.
155.  def create_link_delay(self):
156.      """
157.          Create link delay data, and save it into graph object.
```

```
158.        """
159.        try:
160.            for src in self.awareness.graph:
161.                for dst in self.awareness.graph[src]:
162.                    if src == dst:
163.                        self.awareness.graph[src][dst]['delay'] = 0
164.                        continue
165.                    delay = self.get_delay(src, dst)
166.                    self.awareness.graph[src][dst]['delay'] = delay
167.        except:
168.            if self.awareness is None:
169.                self.awareness = lookup_service_brick('awareness')
170.            return
171.
172.    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
173.    def packet_in_handler(self, ev):
174.        """
175.            Parsing LLDP packet and get the delay of link.
176.        """
177.        msg = ev.msg
178.        try:
179.            src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
180.            dpid = msg.datapath.id
181.            if self.sw_module is None:
182.                self.sw_module = lookup_service_brick('switches')
183.
184.            for port in self.sw_module.ports.keys():
```

```
185.                if src_dpid == port.dpid and src_port_no == port.port_no:
186.                    delay = self.sw_module.ports[port].delay
187.                    self._save_lldp_delay(src=src_dpid, dst=dpid,
188.                                          lldpdelay=delay)
189.
190.        except LLDPPacket.LLDPUnknownFormat as e:
191.            return
192.
193.    def create_delay_graph(self):
194.        for src in self.awareness.graph:
195.            for dst in self.awareness.graph[src]:
196.                delay = self.awareness.graph[src][dst]['delay']
197.                if self.delay_graph.get(src) is None:
198.                    self.delay_graph.setdefault(src, {})
199.                self.delay_graph[src][dst] = delay
200.
201.                # self.logger.info("%s<-->%s : %s" % (src, dst, delay))
202.
203.    def show_delay_statis(self):
204.        if settings.TOSHOW and self.awareness is not None:
205.            self.logger.info("\nsrc   dst      delay")
206.            self.logger.info("---------------------------")
207.            for src in self.awareness.graph:
208.                for dst in self.awareness.graph[src]:
209.                    delay = self.awareness.graph[src][dst]['delay']
210.                    self.logger.info("%s<-->%s : %s" % (src, dst, delay))
211.
```

network_awareness.py

```
1.# Copyright (C) 2016 Li Cheng at Beijing University of Posts
2.# and Telecommunications. www.muzixing.com
3.#
4.# Licensed under the Apache License, Version 2.0 (the "License");
5.# you may not use this file except in compliance with the License.
6.# You may obtain a copy of the License at
7.#
8.# http://www.apache.org/licenses/LICENSE-2.0
9.#
10.# Unless required by applicable law or agreed to in writing, software
11.# distributed under the License is distributed on an "AS IS" BASIS,
12.# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or
13.# implied.
14.# See the License for the specific language governing permissions and
15.# limitations under the License.
16.
17.# conding=utf-8
18.import logging
19.import struct
20.import copy
21.import networkx as nx
22.from operator import attrgetter
23.from ryu import cfg
24.from ryu.base import app_manager
25.from ryu.controller import ofp_event
```

```
26.from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
27.from ryu.controller.handler import CONFIG_DISPATCHER
28.from ryu.controller.handler import set_ev_cls
29.from ryu.ofproto import ofproto_v1_3
30.from ryu.lib.packet import packet
31.from ryu.lib.packet import ethernet
32.from ryu.lib.packet import ipv4
33.from ryu.lib.packet import arp
34.from ryu.lib import hub
35.
36.from ryu.topology import event, switches
37.from ryu.topology.api import get_switch, get_link
38.import settings
39.import json
40.import pdb
41.
42.CONF = cfg.CONF
43.
44.
45.class NetworkAwareness(app_manager.RyuApp):
46.    """
47.        NetworkAwareness is a Ryu app for discover topology information.
48.        This App can provide many data services for other App, such as
49.        link_to_port, access_table, switch_port_table,access_ports,
50.        interior_ports,topology graph and shorteest paths.
51.    """
52.    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

```
53.
54.  def __init__(self, *args, **kwargs):
55.      super(NetworkAwareness, self).__init__(*args, **kwargs)
56.      self.topology_api_app = self
57.      self.name = "awareness"
58.      self.link_to_port = {}       # (src_dpid,dst_dpid)->(src_port,dst_port)
59.      self.access_table = {}       # {(sw,port) :[host1_ip]}
60.      self.switch_port_table = {}  # dpip->port_num
61.      self.access_ports = {}       # dpid->port_num
62.      self.interior_ports = {}     # dpid->port_num
63.      self.dict_graph = {}
64.      self.graph = nx.DiGraph()
65.      self.pre_graph = nx.DiGraph()
66.      self.pre_access_table = {}
67.      self.pre_link_to_port = {}
68.      self.shortest_paths = None
69.
70.      # Start a green thread to discover network resource.
71.      self.discover_thread = hub.spawn(self._discover)
72.
73.  def _discover(self):
74.      i = 0
75.      while True:
76.          # self.show_topology()
77.          if i == 5:
78.              self.get_topology(None)
79.              i = 0
```

```
80.           hub.sleep(settings.METRICS_DISCOVERING_PERIOD)
81.           i = i + 1
82.
83.   @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
84.   def switch_features_handler(self, ev):
85.       """
86.           Initial operation, send miss-table flow entry to datapaths.
87.       """
88.       datapath = ev.msg.datapath
89.       ofproto = datapath.ofproto
90.       parser = datapath.ofproto_parser
91.       msg = ev.msg
92.       self.logger.info("switch:%s connected", datapath.id)
93.
94.       # install table-miss flow entry
95.       match = parser.OFPMatch()
96.       actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
97.                                         ofproto.OFPCML_NO_BUFFER)]
98.       self.add_flow(datapath, 0, match, actions)
99.
100.   def add_flow(self, dp, p, match, actions, idle_timeout=0, hard_timeout=0):
101.       ofproto = dp.ofproto
102.       parser = dp.ofproto_parser
103.
104.       inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
105.                                            actions)]
106.
```

```
107.        mod = parser.OFPFlowMod(datapath=dp, priority=p,
108.                                idle_timeout=idle_timeout,
109.                                hard_timeout=hard_timeout,
110.                                match=match, instructions=inst)
111.        dp.send_msg(mod)
112.
113.    def get_host_location(self, host_ip):
114.        """
115.            Get host location info:(datapath, port) according to host ip.
116.        """
117.        for key in self.access_table.keys():
118.            if self.access_table[key][0] == host_ip:
119.                return key
120.        self.logger.info("%s location is not found." % host_ip)
121.        return None
122.
123.    def get_switches(self):
124.        return self.switches
125.
126.    def get_links(self):
127.        return self.link_to_port
128.
129.    def get_graph(self, link_list):
130.        """
131.            Get Adjacency matrix from link_to_port
132.        """
133.        for src in self.switches:
```

```
134.          if self.dict_graph.get(src) is None:
135.              self.dict_graph.setdefault(src, {})
136.          for dst in self.switches:
137.              if src == dst:
138.                  self.graph.add_edge(src, dst, weight=0)
139.                  self.dict_graph[src][dst] = float('Inf')
140.              elif (src, dst) in link_list:
141.                  self.graph.add_edge(src, dst, weight=1)
142.                  self.dict_graph[src][dst] = 1
143.      return self.graph
144.
145.
146.  def create_port_map(self, switch_list):
147.      """
148.          Create interior_port table and access_port table.
149.      """
150.      for sw in switch_list:
151.          dpid = sw.dp.id
152.          self.switch_port_table.setdefault(dpid, set())
153.          self.interior_ports.setdefault(dpid, set())
154.          self.access_ports.setdefault(dpid, set())
155.
156.          for p in sw.ports:
157.              self.switch_port_table[dpid].add(p.port_no)
158.
159.  def create_interior_links(self, link_list):
160.      """
```

```
161.            Get links`srouce port to dst port  from link_list,
162.            link_to_port:(src_dpid,dst_dpid)->(src_port,dst_port)
163.        """
164.        for link in link_list:
165.            src = link.src
166.            dst = link.dst
167.            self.link_to_port[
168.                (src.dpid, dst.dpid)] = (src.port_no, dst.port_no)
169.
170.            # Find the access ports and interiorior ports
171.            if link.src.dpid in self.switches:
172.                self.interior_ports[link.src.dpid].add(link.src.port_no)
173.            if link.dst.dpid in self.switches:
174.                self.interior_ports[link.dst.dpid].add(link.dst.port_no)
175.
176.    def create_access_ports(self):
177.        """
178.            Get ports without link into access_ports
179.        """
180.        for sw in self.switch_port_table:
181.            all_port_table = self.switch_port_table[sw]
182.            interior_port = self.interior_ports[sw]
183.            self.access_ports[sw] = all_port_table - interior_port
184.
185.    def k_shortest_paths(self, graph, src, dst, weight='weight', k=1):
186.        """
187.            Great K shortest paths of src to dst.
```

```
188.     """
189.     generator = nx.shortest_simple_paths(graph, source=src,
190.                                          target=dst, weight=weight)
191.     shortest_paths = []
192.     try:
193.         for path in generator:
194.             if k <= 0:
195.                 break
196.             shortest_paths.append(path)
197.             k -= 1
198.         return shortest_paths
199.     except:
200.         self.logger.debug("No path between %s and %s" % (src, dst))
201. 
202. def all_k_shortest_paths(self, graph, weight='weight', k=1):
203.     """
204.         Creat all K shortest paths between datapaths.
205.     """
206.     _graph = copy.deepcopy(graph)
207.     paths = {}
208. 
209.     # Find ksp in graph.
210.     for src in _graph.nodes():
211.         paths.setdefault(src, {src: [[src] for i in range(k)]})
212.         for dst in _graph.nodes():
213.             if src == dst:
214.                 continue
```

```
215.             paths[src].setdefault(dst, [])
216.             paths[src][dst] = self.k_shortest_paths(_graph, src, dst,
217.                                                     weight=weight, k=k)
218.     return paths
219.
220. # List the event list should be listened.
221. events = [event.EventSwitchEnter,
222.           event.EventSwitchLeave, event.EventPortAdd,
223.           event.EventPortDelete, event.EventPortModify,
224.           event.EventLinkAdd, event.EventLinkDelete]
225.
226. @set_ev_cls(events)
227. def get_topology(self, ev):
228.     """
229.         Get topology info and calculate shortest paths.
230.     """
231.     switch_list = get_switch(self.topology_api_app, None)
232.     self.create_port_map(switch_list)
233.     self.switches = self.switch_port_table.keys()
234.     links = get_link(self.topology_api_app, None)
235.     self.create_interior_links(links)
236.     self.create_access_ports()
237.     self.graph = self.get_graph(self.link_to_port.keys())
238.     # self.shortest_paths = self.all_k_shortest_paths(
239.     #     self.graph, weight='weight', k=CONF.k_paths)
240.
241. def register_access_info(self, dpid, in_port, ip, mac):
```

```
242.     """
243.         Register access host info into access table.
244.     """
245.     if in_port in self.access_ports[dpid]:
246.         if (dpid, in_port) in self.access_table:
247.             if self.access_table[(dpid, in_port)] == (ip, mac):
248.                 return
249.             else:
250.                 self.access_table[(dpid, in_port)] = (ip, mac)
251.                 return
252.         else:
253.             self.access_table.setdefault((dpid, in_port), None)
254.             self.access_table[(dpid, in_port)] = (ip, mac)
255.             return
256.
257. @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
258. def _packet_in_handler(self, ev):
259.     """
260.         Hanle the packet in packet, and register the access info.
261.     """
262.     msg = ev.msg
263.     datapath = msg.datapath
264.
265.     parser = datapath.ofproto_parser
266.     in_port = msg.match['in_port']
267.     pkt = packet.Packet(msg.data)
268.
```

```
269.        eth_type = pkt.get_protocols(ethernet.ethernet)[0].ethertype
270.        arp_pkt = pkt.get_protocol(arp.arp)
271.        ip_pkt = pkt.get_protocol(ipv4.ipv4)
272.
273.        if arp_pkt:
274.            arp_src_ip = arp_pkt.src_ip
275.            arp_dst_ip = arp_pkt.dst_ip
276.            mac = arp_pkt.src_mac
277.
278.            # Record the access info
279.            self.register_access_info(datapath.id, in_port, arp_src_ip, mac)
280.
281.import sys
282.
283.from ryu.cmd import manager
284.
285.
286.def main():
287.    sys.argv.append('--ofp-tcp-listen-port')
288.    sys.argv.append('6653')
289.    sys.argv.append('--observe-links')
290.    # sys.argv.append('ryu/ryu/app/gui_topology/gui_topology.py')
291.    sys.argv.append('qos_route_app')
292.    # sys.argv.append('ofctl_rest')
293.
294.    # sys.argv.append('test_controller')
295.    # sys.argv.append('--verbose')
```

```
296.    sys.argv.append('--enable-debugger')
297.    manager.main()
298.
299.
300.if __name__ == '__main__':
301.    main()
```