Міністерство освіти і науки України

Луцький національний технічний університет

(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій

(повне найменування факультету)

Кафедра комп'ютерної інженерії та охоронних систем

(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

МОДЕЛЮВАННЯ SDN-МЕРЕЖІ ЗАСОБАМИ PYTHON У СЕРЕДОВИЩІ MININET

MODELING OF AN SDN-NETWORK USING PYTHON IN THE MININET ENVIRONMENT

спеціальність 123 Комп'ютерна інженерія

(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія

(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-41
Жуковська Софія Юріївна

(підпис)

Керівник:
к.т.н., доцент
Бортник Катерина Яківна

(підпис)

Кваліфікаційну роботу
допущено до захисту
«_____» червня 2026 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Т. Терлецький
« 23 » 12 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Жуковської Софії Юріївни
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Моделювання SDN-мережі засобами Python у середовищі Mininet*

Керівник роботи *к.т.н., доцент Бортник Катерина Яківна*

затверджені наказом закладу вищої освіти від «20» грудня 2025 року № 536/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 28.05.2026 р.

3. Вихідні дані до роботи: *джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування.*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Теоретичні основи моделювання SDN-мереж
Проектування моделі SDN-мережі
Реалізація та дослідження моделі
Висновки

5. Перелік графічного (ілюстративного) матеріалу:
Моделювання SDN-мережі в Mininet (Python)
Аналіз SDN-архітектури та технологій
Побудова та тестування мережевої моделі
Реалізація Python-скриптів керування мережею

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Теоретичні основи моделювання SDN-мереж	*Бортник К. Я., доцент*		
Проектування моделі SDN-мережі	*Бортник К. Я., доцент*		
Реалізація та дослідження моделі	*Бортник К. Я., доцент*		
Нормоконтроль	*Багнюк Н. В., доцент*		
Гарант ОП	*Лавренчук С. В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О. І., ст. викладач*		

7. Дата видачі завдання 23.12.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми, аналіз предметної області та наявних рішень*	до 10.02.2026 р.	
2.	*Теоретичні основи моделювання SDN-мереж*	до 02.03.2026 р.	
3.	*Проектування моделі SDN-мережі*	до 02.04.2026 р.	
4.	*Реалізація та дослідження моделі*	до 10.04.2026 р.	
5.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	до 01.05.2026 р.	
6.	*Нормоконтроль*	до 23.05.2026 р.	
7.	*Інструментальна перевірка на академічний плагіат*	до 25.05.2026 р.	
8.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	до 28.05.2026 р.	

Здобувач вищої освіти _____ Софія ЖУКОВСЬКА
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Катерина БОРТНИК
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Жуковська С. Ю. Моделювання SDN-мережі засобами Python у середовищі Mininet. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2026.

У роботі проаналізовано принципи роботи традиційних мереж та архітектуру програмно-визначених мереж. Досліджено протокол OpenFlow, компоненти Open vSwitch та Python API середовища Mininet. Виконано порівняльний аналіз існуючих SDN-контролерів: ONOS, OpenDaylight, Ryu та POX.

У ході проектування сформульовано вимоги до системи, спроектовано багаторівневу MVC-архітектуру застосунку та розроблено програмну модель керування на Python.

Реалізовано повнофункціональний застосунок SDN Mininet Simulator обсягом 2360 рядків коду. Застосунок підтримує п'ять типів мережевих топологій (лінійну, зіркову, кільцеву, деревоподібну та повну Mesh), які розгортаються через реальний Mininet API з використанням Open vSwitch та TCLink. Графічний інтерфейс на базі Tkinter забезпечує інтерактивну візуалізацію топології та відображення статистики у вигляді чотирьох графіків Matplotlib у реальному часі.

Найвищу ефективність балансування демонструє Mesh-топологія (52,9 %), найнижчу - деревоподібна (36.8%).

Ключові слова: SDN, OpenFlow, Mininet, Open vSwitch, Python, Tkinter, Matplotlib, балансування навантаження, TCLink, OVSSwitch, топологія мережі, таблиці потоків, BFS, програмно-визначені мережі.

ANNOTATION

Zhukovska S. Modeling of an SDN-network using Python in the Mininet environment .Manuscript.

Qualifying work of a bachelor of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2026.

The work analyzes the operating principles of traditional networks and the architecture of software-defined networks. The OpenFlow protocol, Open vSwitch components, and the Python API of the Mininet environment are investigated. A comparative analysis of existing SDN controllers - ONOS, OpenDaylight, Ryu, and POX - is performed.

During the design phase, system requirements were formulated, a multi-tier MVC application architecture was designed, and a Python-based control model was developed.

A fully functional SDN Mininet Simulator application of 2360 lines of code was implemented. The application supports five types of network topologies (linear, star, ring, tree, and full mesh), deployed through the real Mininet API using Open vSwitch and TCLink. OpenFlow flow table management via ovs-ofctl, connectivity testing (ping) and throughput testing (iperf), a BFS-based load balancing mechanism, and link failure simulation with automatic recovery were implemented. The Tkinter-based graphical interface provides interactive topology visualization and real-time statistics display via four Matplotlib charts.

Keywords: SDN, OpenFlow, Mininet, Open vSwitch, Python, Tkinter, Matplotlib, load balancing, TCLink, OVSSwitch, network topology, flow tables, BFS, software-defined networking.

ЗМІСТ

ВСТУП

Сучасні комп’ютерні мережі переживають фундаментальну трансформацію. Зростання обсягів трафіку, поширення хмарних обчислень, віртуалізації та мікросервісної архітектури висувають до мережевої інфраструктури принципово нові вимоги: гнучкість, автоматизованість, програмованість та здатність до динамічної реконфігурації. Традиційні мережі, де логіка управління вбудована в кожен окремий пристрій, дедалі менше відповідають цим вимогам – ручне налаштування, відсутність глобального уявлення про стан мережі та залежність від пропрієтарного програмного забезпечення виробників стримують інновації та ускладнюють автоматизацію.

Архітектура програмно-визначених мереж (Software-Defined Networking, SDN) пропонує принципово інший підхід: відокремлення площини управління (control plane) від площини передачі даних (data plane) та централізацію мережевої логіки в програмному контролері. Це дозволяє керувати всією мережевою інфраструктурою як єдиним програмованим об’єктом, впроваджувати нові мережеві функції через програмний код, а не через оновлення прошивок пристроїв, та оперативно реагувати на зміни навантаження або збої. SDN вже успішно застосовується в дата-центрах провідних технологічних компаній (Google B4, Facebook Fabric), у хмарних платформах та телекомунікаційних мережах 5G/6G.

Для дослідження та навчання в галузі SDN стандартом де-факто стало середовище емуляції Mininet, розроблене в Стенфордському університеті. Mininet використовує механізм Linux network namespaces для створення реалістичних мережевих топологій безпосередньо на одній фізичній або віртуальній машині. Комутатори реалізуються як процеси Open vSwitch (OVS) повноцінного програмного OpenFlow-сумісного комутатора. Це дозволяє тестувати реальний мережевий код, алгоритми маршрутизації та балансування навантаження в умовах, максимально наближених до виробничих, без необхідності використовувати фізичне обладнання.

Актуальність теми зумовлена зростаючою потребою у сучасних навчальних та дослідницьких інструментах, що поєднують реалістичність мережевої емуляції з зручним графічним інтерфейсом та можливістю інтерактивного дослідження поведінки SDN-мереж. Існуючі рішення, як правило, або потребують складної інфраструктури (ONOS, OpenDaylight), або надають лише консольний інтерфейс без візуалізації (базовий Mininet CLI), або не підтримують реальну інтеграцію з Mininet API (прості GUI-симулятори). Розроблений у роботі застосунок заповнює цю прогалину, поєднуючи реальне середовище Mininet з інтуїтивним графічним інтерфейсом.

Метою роботи є розробка повнофункціонального Python-застосунку для моделювання SDN-мережі у середовищі Mininet з підтримкою реальних OVS-комутаторів, управління таблицями потоків OpenFlow через ovs-ofctl, тестування зв’язності та пропускної здатності, механізму балансування навантаження та симуляції відмов мережевих каналів, а також графічного інтерфейсу з інтерактивною візуалізацією та статистикою.

Для досягнення поставленої мети вирішуються такі задачі:

− проаналізувати теоретичні основи SDN-архітектури та принципи роботи традиційних і програмно-визначених мереж;

− дослідити протокол OpenFlow, API середовища Mininet та можливості Open vSwitch;

− виконати порівняльний аналіз існуючих SDN-контролерів та обґрунтувати вибір засобів реалізації;

− сформулювати вимоги до системи та спроектувати багаторівневу архітектуру застосунку;

− розробити програмну модель керування на Python з класами NetController, FlowRule, LinkStat;

− реалізувати п’ять класів мережевих топологій на основі Mininet Topo API;

− реалізувати управління правилами OpenFlow через ovs-ofctl та тестування через host.cmd;

− розробити алгоритм BFS для пошуку маршрутів та механізм балансування навантаження;

− реалізувати симуляцію відмов каналів з автоматичним відновленням;

− розробити графічний інтерфейс на Tkinter з візуалізацією топології та графіками Matplotlib;

− провести оцінювання ефективності розробленої моделі за комплексом показників.

Об’єктом дослідження є архітектура програмно-визначених мереж та засоби їх моделювання.

Предметом дослідження є методи та засоби програмної реалізації SDN-мережі з використанням мови Python та середовища Mininet.

Методи дослідження. У роботі використовуються методи об’єктно-орієнтованого програмування (проектування класів та ієрархій), алгоритмічного аналізу (BFS для пошуку шляхів у графах), теорії мереж (моделювання топологій, управління потоками), а також методи емпіричного тестування та порівняльного аналізу для оцінювання ефективності моделі.

Практичне значення роботи. Розроблений застосунок SDN Mininet Simulator може безпосередньо використовуватися як навчальний інструмент у дисциплінах «Комп’ютерні мережі», «Адміністрування мереж», «Програмно-визначені мережі», а також як дослідницька платформа для прототипування мережевих алгоритмів – балансування навантаження, маршрутизації, управління відмовостійкістю. Застосунок підтримує два режими: реальний (Ubuntu/VMware з Mininet) та симуляційний (Windows/macOS без Mininet), що забезпечує широку доступність без вимог до спеціального апаратного забезпечення.

У першому розділі досліджено теоретичні основи моделювання SDN-мереж: принципи роботи традиційних і програмно-визначених мереж, архітектура SDN та протокол OpenFlow, середовище Mininet та його API, порівняльний аналіз SDN-контролерів.

У другому розділі виконано проектування моделі SDN-мережі: сформульовано вимоги до системи, спроектовано архітектуру застосунку, описано програмну модель керування на Python та алгоритми створення і керування моделлю в Mininet.

У третьому розділі описано реалізацію та дослідження моделі: розгортання SDN-мережі через Mininet API, управління потоками OpenFlow, моделювання збоїв та відмов, реалізацію балансування навантаження, оцінювання ефективності SDN-моделі.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ МОДЕЛЮВАННЯ SDN-МЕРЕЖ

1.1 Принципи роботи традиційних комп'ютерних мереж

Традиційні комп'ютерні мережі, які протягом десятиліть становили основу інформаційної інфраструктури, ґрунтуються на децентралізованій архітектурі. У цій моделі кожен мережевий пристрій – комутатор, маршрутизатор чи інший вузо приймає рішення щодо пересилання пакетів автономно, без участі центрального координатора. Пристрій аналізує вхідні дані, звіряє їх із власними таблицями маршрутизації або комутації й визначає подальший напрямок передачі. Такий підхід реалізується на основі усталених моделей взаємодії, зокрема семирівневої моделі OSI або чотири-п'ятирівневої моделі TCP/IP, де обробка даних відбувається послідовно - від фізичного рівня (передача сигналів по кабелю чи в ефірі) до прикладного (функціонал користувацьких додатків).

У цій архітектурі важливу роль відіграють різні типи обладнання. Найпростіші з них хаби – лише транслюють отриманий сигнал на всі порти, не виконуючи жодного аналізу. Комутатори другого рівня використовують MAC-адреси та таблиці CAM для точного направлення пакетів, уникнувши непотрібного широкомовлення. Маршрутизатори третього рівня оперують IP-адресами й застосовують складні алгоритми для визначення оптимального шляху між мережами. Для забезпечення стабільності й ефективності функціонування таких систем використовуються спеціалізовані протоколи: STP та RSTP запобігають петлям, VLAN дозволяє логічно сегментувати фізичну мережу, а внутрішні протоколи, такі як OSPF, IS-IS або RIP, забезпечують маршрутизацію в межах автономної системи. Для обміну маршрутною інформацією між різними мережами застосовується BGP – один із ключових протоколів глобального Інтернету [1].

Центральною рисою традиційних мереж є тісна інтеграція двох основних компонентів – площини управління (control plane) та площини даних (data plane) у межах одного пристрою. Площина управління відповідає за логічні рішення:

запускає маршрутизаційні протоколи, обчислює найкращі шляхи, формує таблиці пересилання й реагує на зміни в топології. Площина даних виконує безпосереднє пересилання пакетів на основі цих таблиць, здійснюючи операції «match-action» – зіставлення заголовків пакетів із записами в таблиці та виконання відповідної дії (наприклад, пересилання на певний порт або відкидання). Хоча така модель довгий час демонструвала надійність, вона має низку істотних недоліків, особливо в умовах сучасних масштабних інфраструктур [2].

Одне з головних обмежень полягає в необхідності ручного налаштування кожного пристрою окремо. Адміністратор повинен підключатися до кожного вузла, вносити зміни в конфігурацію, перевіряти їхню коректність - процес, який є трудомістким, повільним і схильним до помилок. У великих мережах із сотнями чи тисячами пристроїв це стає практично невиконуваним завданням. Крім того, кожен пристрій має лише локальне уявлення про стан мережі - він знає лише про своїх сусідів і отримані від них маршрути, але не бачить загальної картини. Це ускладнює швидку реакцію на збої, зміни навантаження чи потребу в перерозподілі трафіку. Впровадження нових функцій часто залежить від пропрієтарного програмного забезпечення конкретного виробника, що призводить до вендорської залежності, ускладнює інтеграцію різних компонентів і гальмує інновації. Автоматизація в таких умовах залишається обмеженою через відсутність універсальних інтерфейсів для програмного керування [1].

На противагу цій парадигмі, програмно-визначені мережі (SDN, Software-Defined Networking) запропонували принципово новий підхід, заснований на чіткому розділенні площини управління та площини даних. У цій моделі всі рішення щодо маршрутизації, політик безпеки, балансування навантаження тощо приймаються не окремими пристроями, а централізованим програмним контролером. Цей контролер, реалізований як окремий процес або кластер процесів, має повне уявлення про всю мережу – він знає про всі пристрої, з’єднання між ними, поточний стан каналів, обсяги трафіку та доступні ресурси.

Завдяки цьому він може приймати оптимальні рішення в реальному часі, динамічно адаптуючи поведінку мережі до змінних умов.

Мережеві пристрої в SDN-архітектурі, зокрема комутатори, перетворюються на прості виконавці, які виконують лише одну функцію – пересилання пакетів за правилами, отриманими від контролера. Вони більше не запускають складні маршрутизаційні протоколи, не обчислюють шляхи, не приймають локальних рішень. Натомість вони отримують від контролера набір правил (flow rules), які визначають, як обробляти пакети з певними характеристиками. Якщо комутатор отримує пакет, для якого немає відповідного правила, він надсилає його контролеру, який аналізує ситуацію, приймає рішення та встановлює нове правило на комутаторі. Цей механізм забезпечує гнучкість та адаптивність мережі без необхідності зберігати складну логіку в кожному пристрої [3].

Для взаємодії між контролером і мережевими пристроями використовуються стандартизовані інтерфейси, відомі як південні API (southbound APIs). Найпоширенішим серед них є протокол OpenFlow, розроблений у Стенфордському університеті, який став фактичним стандартом для SDN. OpenFlow дозволяє контролеру читати та змінювати таблиці потоків на комутаторі, встановлювати правила зіставлення за десятками полів заголовків пакетів і задавати відповідні дії. Крім OpenFlow, застосовуються й інші протоколи, такі як NETCONF для управління конфігурацією, OVSDB для взаємодії з Open vSwitch, або P4Runtime для програмування data plane на основі мови P4. З боку користувацьких додатків контролер надає північні API (northbound APIs), які зазвичай реалізовані через RESTful інтерфейси з використанням JSON або gRPC. Це дає змогу розробникам створювати мережеві додатки – системи моніторингу, брандмауери, балансувальники навантаження як звичайні програми, що взаємодіють з мережею через стандартизовані виклики [4].

Переваги SDN порівняно з традиційними мережами є значущими. По-перше, централізоване управління спрощує адміністрування, особливо в

великих інфраструктурах. По-друге, мережа стає повністю програмованою – зміни можна вносити за секунди через скрипти або додатки, без ручного втручання. По-третє, динамічна реконфігурація дозволяє оперативно реагувати на зміни навантаження, збої або нові вимоги безпеки. По-четверте, масштабованість покращується завдяки використанню недорогих white-box комутаторів, оскільки вся інтелектуальна частина винесена в контролер. По-п’яте, централізована архітектура забезпечує повну видимість мережі, що спрощує моніторинг, аналітику та усунення несправностей. По-шосте, політики безпеки можуть застосовуватися глобально, що підвищує захищеність інфраструктури.

Ці переваги роблять SDN ідеальним рішенням для сучасних середовищ, де критичними є гнучкість, автоматизація та швидкість реакції. У дата-центрах SDN дозволяє ефективно керувати великими обсягами трафіку між серверами, оптимізувати використання каналів, швидко розгортати нові сервіси. У хмарних платформах він забезпечує ізоляцію клієнтів, гнучке управління віртуальними мережами та інтеграцію з системами оркестрації, такими як Kubernetes. У телекомунікаціях SDN є ключовим компонентом NFV (Network Functions Virtualization) і 5G/6G, де дозволяє динамічно створювати мережеві зрізи (network slices) під конкретні сервіси. На практиці успішні реалізації SDN вже існують, зокрема, Google використовує власну SDN-мережу B4 для між дата-центрового зв’язку, Facebook розгорнув Fabric – масштабну SDN-інфраструктуру для своїх дата-центрів [3].

1.2 Архітектура та концепція SDN мереж

Програмно-визначені мережі (SDN) є сучасним підходом до проектування комп’ютерних мереж, у якому функції управління мережею відокремлені від безпосередньої передачі даних. На відміну від класичних мереж, де кожен комутатор чи маршрутизатор приймає рішення про пересилання пакетів самостійно, у SDN центральну роль відіграє програмний контролер, що має

повне уявлення про топологію мережі. Він керує мережевими пристроями через стандартизовані протоколи, зокрема OpenFlow, забезпечуючи гнучкість, програмованість і можливість автоматизації [5].

Суть SDN полягає в тому, що мережеве обладнання (наприклад, комутатори) перестає приймати рішення про маршрутизацію і перетворюється на виконавців команд, отриманих від центрального контролера. Це дає змогу оперативно реагувати на зміни навантаження, оптимізувати шляхи передачі даних, застосовувати політики безпеки в реальному часі та забезпечувати необхідний рівень якості обслуговування (QoS). До ключових принципів архітектури SDN належать: відокремлення площини управління від площини даних, логічна централізація управління, програмна конфігурація мережі, використання відкритих інтерфейсів та абстрагування від апаратної платформи.

Архітектура SDN складається з трьох рівнів: інфраструктурного, управління та додатків. Інфраструктурний рівень утворюють фізичні й віртуальні комутатори, зокрема, Open vSwitch, P4-сумісні пристрої, white-box-комутатори від Dell, HP або спеціалізоване обладнання Cisco Nexus з підтримкою OpenFlow. Такі пристрої містять таблиці потоків, де зберігаються правила виду «умова → дія → статистика». При отриманні пакета комутатор шукає відповідне правило, виконує задану дію (пересилання, відкидання, модифікація заголовка тощо) і оновлює статистику. Якщо правило відсутнє, пакет надсилається контролеру для подальшої обробки [6].

Рівень управління реалізується за допомогою SDN-контролера, який може працювати як окремий екземпляр або у вигляді кластера для забезпечення високої доступності. Контролер відстежує топологію мережі, виявляє хости, обчислює оптимальні маршрути, керує QoS, балансуванням навантаження та безпекою. Для взаємодії з мережевими пристроями використовуються так звані південні інтерфейси (Southbound API), серед яких найпоширенішим є OpenFlow; також застосовуються NETCONF, OVSDB, P4Runtime, gNMI та інші [5].

На рівні додатків працюють спеціалізовані програми, що реалізують конкретні функції: міжмережеві екрани, системи балансування навантаження,

моніторингу, захисту від вторгнень або DDoS-атак, інженерії трафіку тощо. Ці додатки взаємодіють з контролером через північні інтерфейси (Northbound API), зазвичай на основі REST/RESTCONF, YANG або мовних API (Python, Java). У разі використання кластерів контролерів для синхронізації стану між вузлами застосовуються горизонтальні інтерфейси (East-Westbound API).

OpenFlow, розроблений у Стенфордському університеті та стандартизований ONF (Open Networking Foundation), є ключовим протоколом для взаємодії між контролером і комутаторами. Найбільш поширеною версією у промисловості залишається OpenFlow 1.3. Комутатор, сумісний з цим протоколом, може містити до 256 таблиць потоків, а також групові та метричні таблиці. Зв'язок із контролером здійснюється через захищений канал (TLS). Правила в таблицях потоків можуть аналізувати понад 40 параметрів пакета - від MAC- і IP-адрес до портів TCP/UDP, VLAN-тегів або MPLS-міток. Дії, які може виконати комутатор, включають пересилання, відкидання, модифікацію заголовків, перехід до іншої таблиці або застосування обмежень швидкості. Якщо відповідного правила немає, пакет надсилається контролеру (packet-in), який аналізує його й встановлює нове правило (flow-mod) [7].

Станом на початок 2026 року серед провідних SDN-контролерів виділяються ONOS і OpenDaylight. ONOS орієнтований на телекомунікаційні мережі, підтримує горизонтальну масштабованість і строгу консистентність, що робить його придатним для carrier-grade рішень. OpenDaylight, написаний на Java, є зрілою платформою з великою екосистемою модулів і підтримкою таких виробників, як Cisco та Red Hat. Для навчання та розробки прототипів часто використовують легкі контролери Ryu (Python) і POX, а Floodlight знаходить застосування в корпоративних мережах. Сучасні рішення, такі як Stratum у поєднанні з ONOS, дозволяють програмувати рівень передачі даних за допомогою мови P4, що забезпечує максимальну гнучкість [8].

До переваг SDN належать: можливість глобальної оптимізації трафіку, швидке впровадження змін (від секунд замість тижнів), проста інтеграція з інструментами автоматизації (Ansible, Kubernetes), покращена видимість стану

мережі, легка реалізація мікросегментації та архітектур zero-trust, а також зниження капітальних витрат завдяки використанню недорогих white-box-пристроїв. Разом з тим існують і певні ризики. Централізований контролер може стати єдиною точкою відмови, тому потребує механізмів високої доступності. Обробка невідомих пакетів через packet-in може спричиняти затримки, особливо в мережах із великим обсягом нових потоків. Масштабування контролера в дуже великих мережах вимагає ретельного проектування, а вся архітектура потребує надійного захисту, зокрема на рівні південних інтерфейсів [7].

Традиційні мережі ґрунтуються на розподіленій площині управління, де протоколи OSPF, BGP або IS-IS забезпечують автономну маршрутизацію, а конфігурація виконується окремо на кожному пристрої. SDN, навпаки, використовує централізоване управління, програмні політики та динамічну маршрутизацію, що значно спрощує адміністрування та підвищує адаптивність мережі. Сучасні тенденції свідчать про тісну інтеграцію SDN з NFV (віртуалізацією мережевих функцій), розвиток intent-based networking (IBN), широке застосування мови P4 для програмування data plane, впровадження Segment Routing over IPv6 (SRv6) у поєднанні з SDN, а також активне використання SDN-архітектур у 5G/6G core-мережах (зокрема в рамках Open RAN) і хмарних платформах, таких як OpenStack Neutron, VMware NSX та Google Andromeda. Ці напрямки підтверджують, що SDN залишається основою для побудови інтелектуальних, гнучких і автоматизованих мереж майбутнього [5].

1.3 Методи та засоби моделювання мереж

Моделювання мереж є невід’ємною частиною сучасних досліджень, розробки та навчального процесу в галузі комп’ютерних мереж. Воно дозволяє аналізувати поведінку мережевих систем у різних умовах, не вдаючись до

дорогого або складного фізичного обладнання. Існує кілька підходів до моделювання, кожен з яких має свої особливості, переваги та обмеження.

Аналітичне моделювання ґрунтується на строгому математичному описі мережевих процесів. У цьому підході використовуються такі інструменти, як теорія масового обслуговування, графові моделі та марковські процеси. Цей метод добре підходить для отримання загальних оцінок характеристик мережі, наприклад середнього часу затримки передачі даних або максимальної пропускної здатності. Проте він не враховує багатьох практичних аспектів, таких як специфіка реалізації протоколів, взаємодія програмного забезпечення або непередбачувані ефекти, що виникають у реальному мережевому середовищі. Тому аналітичні моделі найчастіше використовуються на початкових етапах проектування або для теоретичного аналізу [8].

Імітаційне, або симуляційне, моделювання передбачає створення віртуальної моделі мережі за допомогою спеціалізованого програмного забезпечення. До найвідоміших інструментів цього класу належать NS-3, OMNeT++, OPNET та NetCracker. Симулятори дозволяють детально налаштовувати параметри мережі, включаючи топологію, характеристики каналів зв’язку, завантаження трафіком тощо. Однак такі системи часто не використовують справжні мережеві стеки операційної системи, що може призводити до розбіжностей між результатами симуляції та поведінкою реальної мережі. Незважаючи на це, симуляція залишається важливим інструментом для дослідження складних сценаріїв, де потрібна висока ступінь контролю над умовами експерименту.

Емуляційне моделювання, на відміну від симуляції, базується на реальному програмному та операційному оточенні. У цьому випадку всі компоненти мережі (хости, комутатори і ін.) функціонують як справжні процеси в операційній системі. Такий підхід забезпечує високу реалістичність, оскільки використовується повноцінний мережевий стек Linux. Це дозволяє запускати будь-які стандартні мережеві додатки, такі як SSH-клієнти, веб-сервери, інструменти iperf або tcpdump, без жодних модифікацій. До інструментів

емуляції належать Mininet, GNS3, а частково – Cisco Packet Tracer. Емуляція особливо корисна для тестування програмного забезпечення, розробки контролерів SDN, а також для навчання, оскільки відтворює поведінку реальної мережі з високою точністю [9].

Гібридне моделювання поєднує в собі елементи симуляції та емуляції. У такому підході одна частина мережі може бути емульована з використанням реальних процесів, а інша – симульована для економії ресурсів або спрощення складних компонентів. Це особливо актуально при тестуванні великих або розподілених систем, де повна емуляція може бути занадто ресурсоємною, а чиста симуляція – недостатньо точною.

Особливе місце серед інструментів емуляції займає Mininet – провідний інструмент для моделювання програмно-визначених мереж. Mininet дозволяє створювати повноцінні віртуальні мережі прямо на звичайному ноутбуці або віртуальній машині, використовуючи вбудовані можливості сучасних ядер Linux. На відміну від симуляторів, Mininet не працює з абстрактними моделями – він створює справжні процеси, мережеві інтерфейси та таблиці маршрутизації, що робить його ідеальним середовищем для розробки, тестування та демонстрації OpenFlow-контролерів. Контролери, розроблені та перевірені в Mininet, зазвичай можуть бути перенесені на реальне обладнання з мінімальними змінами [10].

У Mininet кожен хост реалізований як окремий мережевий простір (network namespace) у Linux, що забезпечує повну ізоляцію процесів, інтерфейсів та таблиць маршрутизації. Комутатори можуть бути реалізовані на основі Open vSwitch – потужного програмного комутатора з підтримкою OpenFlow або простіших user-space рішень. З’єднання між компонентами мережі створюються за допомогою віртуальних Ethernet-пар (veth), які є частиною ядра Linux і гарантують низьку затримку та високу продуктивність. Контролер може бути як вбудованим, так і зовнішнім, наприклад, Ryu, ONOS, OpenDaylight, POX або Floodlight, що дозволяє легко інтегрувати Mininet у будь-яке SDN-середовище.

Завдяки використанню легковагих namespaces та ефективного управління процесами, Mininet забезпечує дуже швидкий запуск мереж: навіть складні топології з сотнями вузлів можуть бути створені за кілька секунд. Він повністю сумісний зі стандартними мережевими інструментами, підтримує OpenFlow версій від 1.0 до 1.5, дозволяє створювати мережі з тисячами вузлів на сучасному комп'ютері, інтегрується з Wireshark, tcpdump та іншими аналітичними засобами, а також надає гнучкий Python API для автоматизації побудови топологій, тестування та інтеграції в системи безперервної інтеграції. Для установки Mininet рекомендуються дистрибутиви Ubuntu 20.04, 22.04 або 24.04, де він може бути легко встановлений через менеджер пакетів або з офіційного репозиторію GitHub.

Найпростіший спосіб – через менеджер пакетів: sudo apt-get update, sudo apt-get install mininet -y або з офіційного репозиторію GitHub для отримання найновішої версії: git clone https://github.com/mininet/mininet, cd mininet, util/install.sh -a.

Найпростіший запуск створює мінімальну мережу з одним комутатором, двома хостами та вбудованим контролером: sudo mn.

Приклади команд для створення топологій наведено в лістингу 1.1.

Лістинг 1.1 – Приклади команд для створення топологій

```
sudo mn --topo single,4                       # 1 свіч + 4 хости
sudo mn --topo linear,5                       # лінійна мережа з 5 свічами
sudo mn --topo tree,depth=3,fanout=2          # деревоподібна топологія
sudo mn --controller remote,ip=127.0.0.1,port=6653  # зовнішній контролер
sudo mn --switch ovsk --mac                   # використання OVS + зручні
MAC-адреси
```

кінець лістингу 1.1

Команди Mininet CLI. Після запуску мережі відкривається інтерактивна оболонка, де доступні такі команди:

- pingall: перевірка зв'язності між усіма хостами;
- net: відображення топології;
- dump: інформація про всі вузли;

− links: стан лінків;

− h1 ping h2 -c 10: запуск ping між хостами;

− h1 iperf -s & h2 iperf -c 10.0.0.1: тест пропускної здатності;

− xterm h1 h2: відкриття окремих терміналів для хостів;

− py: вхід у Python-оболонку Mininet для динамічного управління;

− wireshark h1-eth0: запуск Wireshark на інтерфейсі хоста;

− exit або Ctrl+D: завершення роботи.

Mininet надає зручний Python API для визначення користувацьких топологій. Приклад файлу mytopo.py подано в лістингу 1.2.

Лістинг 1.2 – Приклад користувацької топології Mininet

```
from mininet.topo import Topo
class MyTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(h3, s2)
        self.addLink(s1, s2)
topos = {'mytopo': MyTopo}
```

кінець лістингу 1.2

Запуск: sudo mn --custom mytopo.py --topo mytopo --controller remote.

Популярні вбудовані топології:

− single,N – один комутатор і N хостів;

− linear,N – ланцюг із N комутаторів, до кожного підключено хост;

− tree,depth=N,fanout=M – деревоподібна структура;

− torus – тороїдальна топологія;

− custom – користувацькі топології.

Форки та розширення Mininet:

− Containernet – замінює Linux namespaces на Docker-контейнери, що дозволяє емулювати мікросервісні архітектури та NFV-сервіси;

− Mininet-WiFi – додає підтримку бездротових мереж, включаючи точки доступу, мобільність, handover та моделювання радіосигналу.

Mininet є стандартом де-факто у навчанні, дослідженні та розробці SDN-рішень. Його здатність поєднувати реалістичність, продуктивність і гнучкість робить його незамінним інструментом для будь-якого, хто працює з програмно-визначеними мережами, OpenFlow або мережевою автоматизацією.

1.4 Аналіз існуючих SD контролерів

Mininet є потужним інструментом для емуляції комп'ютерних мереж, який дозволяє відтворювати практично будь-яку мережеву топологію прямо на звичайному ноутбуці або робочій станції. Він створює повноцінну віртуальну мережу, що складається з хостів, комутаторів і з'єднань між ними, використовуючи лише один фізичний пристрій. Для створення найпростішої мережі з двома хостами та одним комутатором достатньо виконати команду sudo mn у терміналі. Ця простота робить Mininet надзвичайно доступним для початківців, водночас зберігаючи достатню глибину для професійного використання в наукових дослідженнях, розробці програмного забезпечення та демонстрації нових концепцій у сфері мереж.

Основна сфера застосування Mininet – це розробка, тестування та демонстрація рішень, пов'язаних із програмно-визначеними мережами (SDN) та протоколом OpenFlow. Контролери, створені та перевірені в середовищі Mininet, зазвичай можуть бути перенесені на реальне мережеве обладнання з мінімальними змінами, оскільки Mininet використовує справжні мережеві стеки Linux. Це забезпечує високу достовірність результатів тестування та значно прискорює цикл розробки. На відміну від симуляторів, які моделюють мережу абстрактно, Mininet створює справжні процеси, інтерфейси та правила маршрутизації, що дозволяє запускати будь-які стандартні мережеві додатки без модифікацій [11].

Архітектура Mininet ґрунтується на можливостях сучасних ядер Linux, зокрема на механізмах процесної віртуалізації та мережевих просторах (network namespaces). Кожен хост у Mininet реалізований як окремий процес оболонки bash, що працює в ізольованому мережевому просторі. Це означає, що кожен хост має власний набір мережевих інтерфейсів, таблиць маршрутизації та IP-адрес, а також не бачить процесів інших хостів. Такий підхід забезпечує повну ізоляцію між вузлами мережі, що критично важливо для точного відтворення поведінки реальної системи. Комутатори в Mininet реалізовані як програмні рішення - найчастіше використовується Open vSwitch, але також доступний і довідковий комутатор OpenFlow. З'єднання між хостами та комутаторами створюються за допомогою віртуальних Ethernet-пар (veth), які є частиною ядра Linux і гарантують низьку затримку та високу продуктивність [10].

Однією з ключових переваг Mininet є його гнучкий Python API, який дозволяє створювати мережі будь-якої складності – від простих лінійних топологій до складних деревоподібних або торів. Користувач може визначити власний клас топології, успадкований від базового класу Topo, і вказати, як саме повинні бути з'єднані вузли. Це робить Mininet ідеальним інструментом для автоматизованого тестування, оскільки топології можна легко параметризувати та інтегрувати в системи безперервної інтеграції. Крім того, Mininet надає зручний командний інтерфейс (CLI), який дозволяє взаємодіяти з мережею в реальному часі: виконувати команди на окремих хостах, перевіряти зв'язність за допомогою ping, вимірювати пропускну здатність через iperf, аналізувати інтерфейси за допомогою ifconfig тощо. Цей інтерфейс значно спрощує діагностику та налагодження мережі під час роботи.

Mininet також включає утиліту очищення (mn -c), яка автоматично видаляє залишки попередніх запусків, такі як залишкові інтерфейси, процеси або тимчасові файли в /tmp. Це особливо корисно, коли система починає поводитися нестабільно через накопичення сміття після багаторазових запусків. Інструмент підтримує як Python 2, так і Python 3, що забезпечує сумісність із широким колом існуючих проектів. Хоча обидві версії можна встановити одночасно, за

замовчуванням використовуватиметься та, що була встановлена останньою. Однак користувач завжди може явно вказати, яку версію Python використовувати при запуску Mininet [8].

Щодо операційних систем, Mininet офіційно підтримує Ubuntu 20.04 LTS та 22.04 LTS, що є найпоширенішими дистрибутивами серед дослідників та розробників. Процес встановлення добре задокументований і може бути виконаний як через менеджер пакетів, так і з вихідного коду з офіційного репозиторію GitHub. Це дозволяє отримати як стабільну версію, так і найновіші функції, які ще не увійшли до офіційного релізу. Розробники також дбають про стабільність: з кожним релізом покращується надійність тестування завдяки інтеграції з GitHub Actions, а також додається підтримка сучасних технологій, таких як cgroups v2, що забезпечує краще управління ресурсами.

Документація Mininet є однією з найсильніших сторін проекту. Вона включає не лише повні описи API через систему docstrings у Python, але й можливість генерації HTML- чи PDF-документації за допомогою команди make doc. Крім того, на офіційному сайті доступні посібники, приклади використання та вступ до Python API. Особливо корисним є розділ FAQ на wiki-сторінці проекту, де зібрані типові питання та рішення проблем, з якими часто стикаються користувачі. Це робить навчання роботі з Mininet інтуїтивним і доступним навіть для тих, хто раніше не працював із SDN [9].

Проект Mininet є повністю відкритим і розвивається завдяки зусиллям спільноти. Будь-хто може завантажити вихідний код, вивчити його, внести зміни, повідомити про помилки або запропонувати нові функції через систему pull requests на GitHub. Активна спільнота користувачів та розробників, яка об’єднана навколо поштової розсилки mininet-discuss, забезпечує оперативну підтримку та обмін досвідом. Це створює здорове екосистемне середовище, де ідеї швидко знаходять своє втілення, а проблеми - своє рішення [11].

Завдяки своїй універсальності, Mininet знайшов застосування в різних галузях: від академічних досліджень і навчання студентів до промислового прототипування SDN-контролерів. Він став стандартом де-факто для тестування

OpenFlow-логіки, оскільки забезпечує майже ідентичне середовище порівняно з реальним обладнанням. Багато провідних компаній і дослідницьких лабораторій використовують Mininet як основу для розробки нових мережевих архітектур, алгоритмів маршрутизації та механізмів безпеки. Його здатність точно відтворювати поведінку мережі при мінімальних апаратних витратах робить його незамінним інструментом у сучасному мережевому інженерії.

У підсумку, Mininet – це не просто емулятор мережі, а повноцінна платформа для інновацій у сфері комп'ютерних мереж. Він поєднує простоту використання, глибину функціональності та високу реалістичність, що робить його ідеальним вибором як для освітніх цілей, так і для серйозних науково-дослідних та інженерних проектів. Завдяки активній спільноті, відкритому коду та постійному розвитку, Mininet залишається на передовій технологій SDN і продовжує формувати майбутнє мережевих інфраструктур.

РОЗДІЛ 2
ПРОЕКТУВАННЯ МОДЕЛІ SDN-МЕРЕЖІ

2.1 Вимоги до моделі мережі

Система повинна забезпечувати програмне керування мережевою інфраструктурою, що передбачає можливість динамічного створення, зміни та видалення мережевих елементів, таких як вузли, комутатори та зв’язки між ними. Управління всіма компонентами мережі має здійснюватися централізовано через єдиний контролер, що дозволяє уникнути розрізненого налаштування окремих пристроїв. Налаштування мережевих параметрів та конфігурацій має бути автоматизованим, щоб зменшити ймовірність людських помилок та підвищити швидкість розгортання.

Управління потоками даних є однією з ключових функцій системи. Вона повинна дозволяти визначати правила маршрутизації трафіку на основі різних критеріїв, зокрема IP-адрес джерела та призначення, MAC-адрес, типів протоколів, таких як TCP, UDP або ICMP, а також номерів портів. Правила мають підтримувати пріоритети, що дозволяє точно контролювати порядок їх застосування. Крім того, система повинна забезпечувати можливість динамічно змінювати ці правила в реальному часі без переривання роботи мережі.

Моніторинг та аналітика є невід’ємною частиною функціональності. Система має постійно відстежувати стан усіх мережевих пристроїв, зокрема їх статус, рівень навантаження та наявність помилок. Необхідно збирати статистику щодо обсягів переданих даних, кількості пакетів та використання пропускної здатності каналів зв’язку. Також важливою є здатність виявляти аномалії та потенційні проблеми у мережі, а всі події та зміни конфігурації мають фіксуватися у журналі для подальшого аналізу.

Топологічні можливості системи повинні бути гнучкими. Користувач має мати змогу створювати різні типи топологій, включаючи лінійну, зіркоподібну, кільцеву, повну ячеїсту, деревоподібну, а також власні користувацькі конфігурації. Мережа має підтримувати динамічну зміну топології під час її

роботи, що дозволяє моделювати реальні сценарії адаптації до змін. Для запобігання помилок необхідна валідація коректності створених топологій.

Система повинна надавати інструменти для тестування та діагностики. До них належать засоби перевірки зв’язку між будь-якими вузлами мережі, вимірювання затримок передачі даних, тестування пропускної здатності каналів, а також можливість симуляції різних мережевих сценаріїв та навантажень. Це дозволяє оцінювати продуктивність мережі в різних умовах.

Безпекові функції передбачають контроль доступу до мережі на основі правил потоків, можливість ізоляції окремих мережевих сегментів, логування підозрілої активності та захист від мережевих атак типу DoS. Ці механізми сприяють підвищенню загальної стійкості та захищеності мережевої інфраструктури.

Продуктивність системи має бути достатньою для моделювання мережі, що складається з до п’ятдесяти вузлів та двадцяти комутаторів. Збір та оновлення статистики повинні відбуватися кожні дві-п’ять секунд, щоб забезпечити актуальність даних. Обробка команд керування має відбуватися з мінімальними затримками, а використання системних ресурсів – бути ефективним.

Надійність є критично важливою характеристикою. Система повинна працювати стабільно, без зависань та аварійних завершень. У разі виникнення помилок має бути забезпечено відновлення без втрати даних. Некоректні вхідні дані повинні оброблятися коректно, без збоїв. Також необхідно передбачити механізм автоматичного резервного копіювання конфігурацій.

Архітектура системи має бути модульною, що дозволить у майбутньому легко додавати нові функції. Повинна бути забезпечена підтримка розширення кількості мережевих елементів та гнучкість у додаванні нових типів мережевих пристроїв.

Зручність використання передбачає наявність інтуїтивного графічного інтерфейсу користувача, контекстної довідки та підказок. Меню та інструменти мають бути організовані логічно. Стан пристроїв та подій має відображатися за

допомогою кольорової індикації, а для швидкого доступу до функцій слід передбачити підтримку гарячих клавіш.

Система повинна бути крос-платформенною, тобто працювати як на операційних системах Windows, так і Linux. Вона не має залежати від конкретних апаратних конфігурацій і повинна використовувати стандартні бібліотеки Python для забезпечення сумісності.

Щодо підтримки протоколів, система має емулювати роботу основних мережевих протоколів, таких як TCP, UDP, ICMP та ARP, підтримувати IPv4 адресацію та моделювати різні типи мережевого трафіку.

Графічний інтерфейс користувача повинен мати головне вікно з розділеною структурою: панель керування зліва, панель моніторингу справа та журнал подій у нижній частині. Топологія мережі має відображатися візуально, а стан пристроїв - індикований кольорами: зелений для активного, червоний для неактивного або помилкового, жовтий для проміжного стану. Налаштування мають здійснюватися через інтерактивні діалогові вікна.

Програмний інтерфейс (API) повинен мати чітко визначені методи для всіх операцій керування. Повернуті дані мають мати консистентну структуру у форматі JSON. Система повинна надавати інформативні повідомлення про помилки та підтримувати асинхронні операції.

Інтерфейс збереження даних передбачає зберігання конфігурацій топологій у форматі JSON, експорт статистики у текстовому та JSON форматах, автоматичне збереження журналу подій, а також підтримку імпорту та експорту конфігурацій.

Моделювання має бути реалістичним. Система повинна емулювати мережеві затримки, моделювати втрати пакетів, відтворювати різні шаблони трафіку та симулювати збої обладнання та зв’язків. Це дозволяє отримувати достовірні результати під час тестування.

Модель має бути масштабованою. Користувач повинен мати можливість створювати мережі різного розміру, підтримувати різні типи топологічних структур та гнучко налаштовувати параметри пристроїв.

Аналітичні можливості включають розрахунок шляхів між будь-якими вузлами, аналіз використання ресурсів мережі, прогнозування навантаження на основі історичних даних та візуалізацію статистики у вигляді графіків.

Користувацька документація повинна містити пояснення основних концепцій SDN, покрокові інструкції з використання системи, опис усіх функцій та інструментів, а також приклади типових сценаріїв використання.

Технічна документація має включати опис архітектури програми, схеми взаємодії компонентів, специфікації форматів даних та протоколів, а також інструкції щодо розширення функціоналу.

Розробка системи має здійснюватися на мові Python версії 3.8 або вище з використанням стандартних бібліотек. Для побудови графічного інтерфейсу слід використовувати Tkinter, а для візуалізації – Matplotlib.

Структура проекту повинна бути модульною, з чітким розділенням обов’язків між компонентами. Код має бути добре документованим, з наявністю коментарів, а файли та каталоги організовані логічно.

Тестування передбачає модульне тестування окремих компонентів, інтеграційне тестування всієї системи, перевірку граничних випадків та обробку помилок.

Встановлення системи має бути простим, з мінімальною кількістю залежностей та чіткими інструкціями. Перед початком роботи слід передбачити перевірку наявності необхідних компонентів.

Обслуговування системи має бути зручним. Для цього необхідно реалізувати логування всіх дій та помилок, надати інструменти для діагностики проблем, а також механізми очищення та скидання даних.

Безпека роботи передбачає захист від випадкових змін конфігурацій, підтвердження критичних операцій користувачем та автоматичне збереження важливих даних.

Система працює в режимі емуляції, а не реальної мережі, що накладає певні технічні обмеження. Кількість одночасних з’єднань є обмеженою, відсутня

підтримка реального мережевого обладнання, а також немає інтеграції з фізичними мережами.

Функціональні обмеження включають неповну підтримку multicast та broadcast трафіку, обмежену емуляцію мережевих протоколів, відсутність шифрування та криптографічних функцій, а також відсутність підтримки віртуалізації мережевих функцій (NFV).

Продуктивність системи залежить від ресурсів хост-системи. Максимальна кількість пристроїв обмежена обчислювальною потужністю комп'ютера, швидкість емуляції також залежить від апаратних характеристик, а деталізація моделювання обмежена самим емуляційним характером середовища.

Ці вимоги формують основу для проектування та розробки моделі SDN-мережі, забезпечуючи повнофункціональне рішення для навчання та дослідження принципів програмно-конфігурованих мереж.

2.2 Проектування архітектури SDN-застосунку програмна частина

Система побудована на основі багаторівневої архітектури, що забезпечує чітке розділення обов'язків між компонентами. Верхній рівень утворює графічний інтерфейс користувача, який відповідає за візуалізацію даних та взаємодію з користувачем. Нижче розташований рівень бізнес-логіки, де реалізовано основні функції управління мережею – це включає менеджер топологій, центральний SDN-контролер та менеджер подій і журналів. Третій рівень складають моделі даних, які описують структуру мережевих елементів: вузли, комутатори та зв'язки між ними. Найнижчий рівень утворюють сервіси емуляції, які забезпечують симуляцію поведінки реальної мережі, включаючи передачу пакетів, затримки, втрати та інші характеристики.

Центральним елементом системи є SDN-контролер, який координує роботу всієї мережевої інфраструктури. Він зберігає поточний стан мережі, включаючи список хостів, комутаторів, зв'язків та активних правил потоків. Контролер також керує кількома фоновими потоками: для моніторингу стану

пристроїв, генерації тестового трафіку та аналітики зібраної статистики. Для внутрішньої комунікації використовуються черги подій та пакетів. Основні функції контролера включають створення та запуск мережі, управління правилами пересилання, тестування зв’язності, збір статистики та симуляцію мережевих подій. Також у його складі реалізовано алгоритм пошуку шляхів на основі методу пошуку в ширину.

Менеджер топології відповідає за формування, валідацію, збереження та завантаження конфігурацій мереж. Він підтримує кілька типів топологій – лінійну, зіркоподібну, кільцеву, повну ячеїсту, деревоподібну та користувацькі конфігурації. Менеджер може генерувати випадкові топології, перевіряти коректність введених даних, а також зберігати та завантажувати конфігурації у файловій системі. Додатково передбачено можливість експорту візуального зображення топології.

Моделі даних описують основні об’єкти мережі. Мережевий вузол має унікальний ідентифікатор, IP- та MAC-адреси, зв’язок із комутатором, поточний статус, статистику переданого та отриманого трафіку, список наданих послуг та чергу пакетів. Комутатор характеризується кількістю портів, таблицею MAC-адрес, таблицею потоків, статусом та статистикою оброблених, пересланих та відкинутих пакетів. Зв’язок між пристроями описується пропускною здатністю, затримкою, рівнем втрат пакетів, поточним використанням каналу та загальною статистикою передачі. Пакет, як базова одиниця передачі, містить інформацію про джерело та призначення, протокол, розмір, час надходження, корисне навантаження та час життя.

Графічний інтерфейс користувача забезпечує зручну взаємодію з системою. Головне вікно має розділену структуру: зліва розташована панель керування з вибором топології, кнопками управління мережею та інструментами швидких дій; справа – панель інформації з вкладками статистики, топології, сповіщень та журналу подій. У верхній частині розташовано стандартне меню з пунктами «Файл», «Мережа», «Правила», «Вигляд» та «Довідка». Система також підтримує діалогові вікна для створення користувацьких топологій,

налаштування правил потоків, тестування зв’язку та перегляду детальної інформації про пристрої (рис. 2.1).

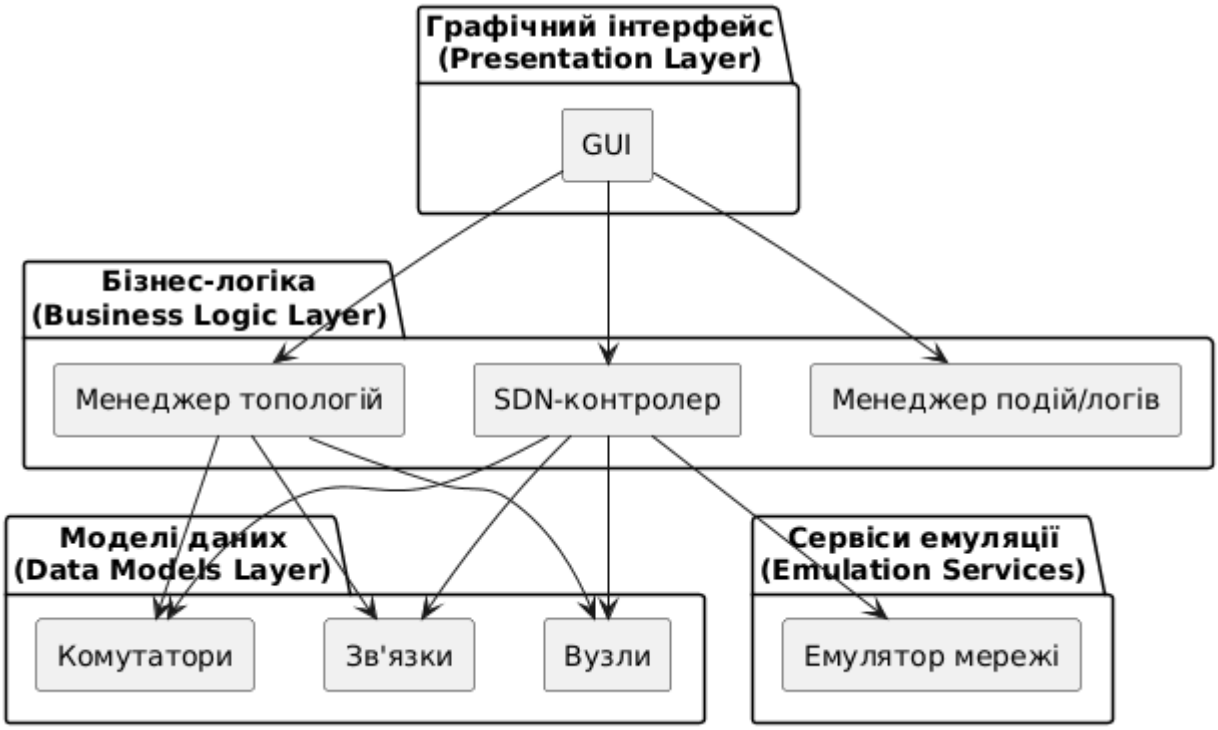

Рисунок 2.1 – Архітектура SDN-застосунку

Робота системи починається з вибору користувачем типу топології через графічний інтерфейс (рис. 2.2). Ця інформація передається менеджеру топологій, який формує конфігурацію та перевіряє її на коректність. Після підтвердження створення мережі дані надходять до SDN-контролера, який ініціалізує моделі вузлів, комутаторів та зв’язків. При натисканні кнопки «Запустити» контролер активує фонові потоки моніторингу, генерації трафіку та аналітики, після чого мережа переходить у робочий стан.

При додаванні правила потоку користувач заповнює форму у діалоговому вікні. Інтерфейс формує структуру match-action та передає її контролеру. Той, у свою чергу, валідує правило, додає його до таблиці відповідного комутатора та оновлює інтерфейс. Аналогічно відбувається тестування зв’язку: після вибору двох вузлів система шукає шлях між ними за допомогою BFS, імітує передачу

пакета, розраховує затримку та повертає результат, який відображається у журналі та впливає на статистику.

Основний потік програми відповідає за роботу графічного інтерфейсу та обробку подій користувача. Фонові потоки працюють паралельно: потік моніторингу регулярно оновлює стан пристроїв та виявляє аномалії; потік генерації трафіку створює випадкові пакети для емуляції реального навантаження; потік аналітики обробляє зібрані дані, виявляє тренди та формує сповіщення.

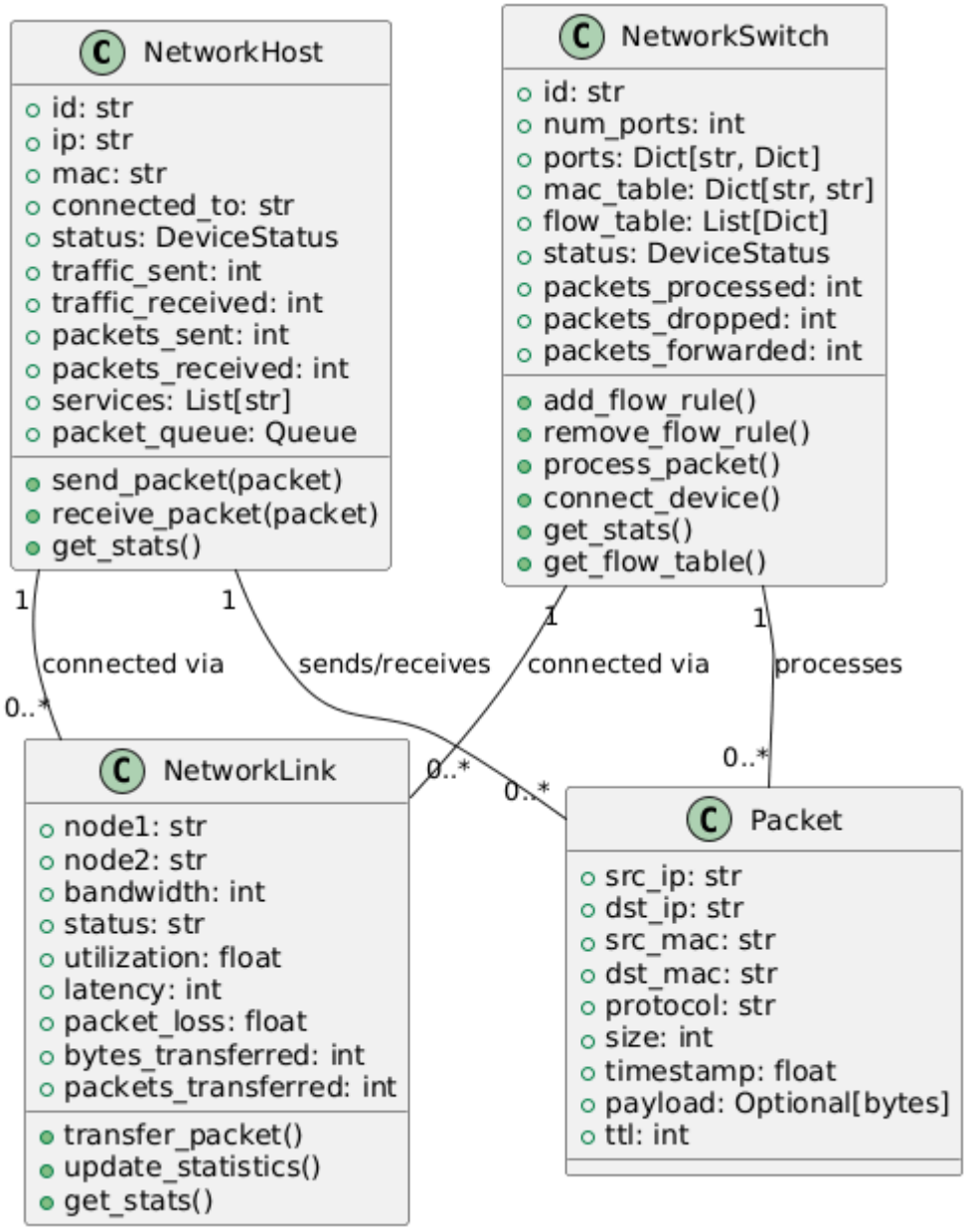

Рисунок 2.2 – Взаємодія компонентів

Обробка пакета комутатором починається з оновлення MAC-таблиці на основі адреси джерела. Потім система шукає відповідне правило в таблиці потоків, враховуючи пріоритети. Якщо правило знайдено, виконується задана дія, наприклад, пересилання на певний порт. Якщо правило відсутнє, застосовується дія за замовчуванням, зазвичай широкомовне пересилання. Після цього оновлюється статистика правила, і пакет передається далі.

Алгоритм пошуку шляху між двома вузлами реалізований як пошук у ширину (рис. 2.3). На початку будується граф мережі на основі поточної топології. У чергу поміщається початковий вузол разом із шляхом, що складається лише з нього. Поки черга не порожня, з неї вилучається черговий вузол. Якщо він збігається з цільовим, повертається знайдений шлях. Інакше до черги додаються всі його невідвідані сусіди з оновленим шляхом. Якщо черга спорожніє, а ціль не знайдено, шлях вважається відсутнім.

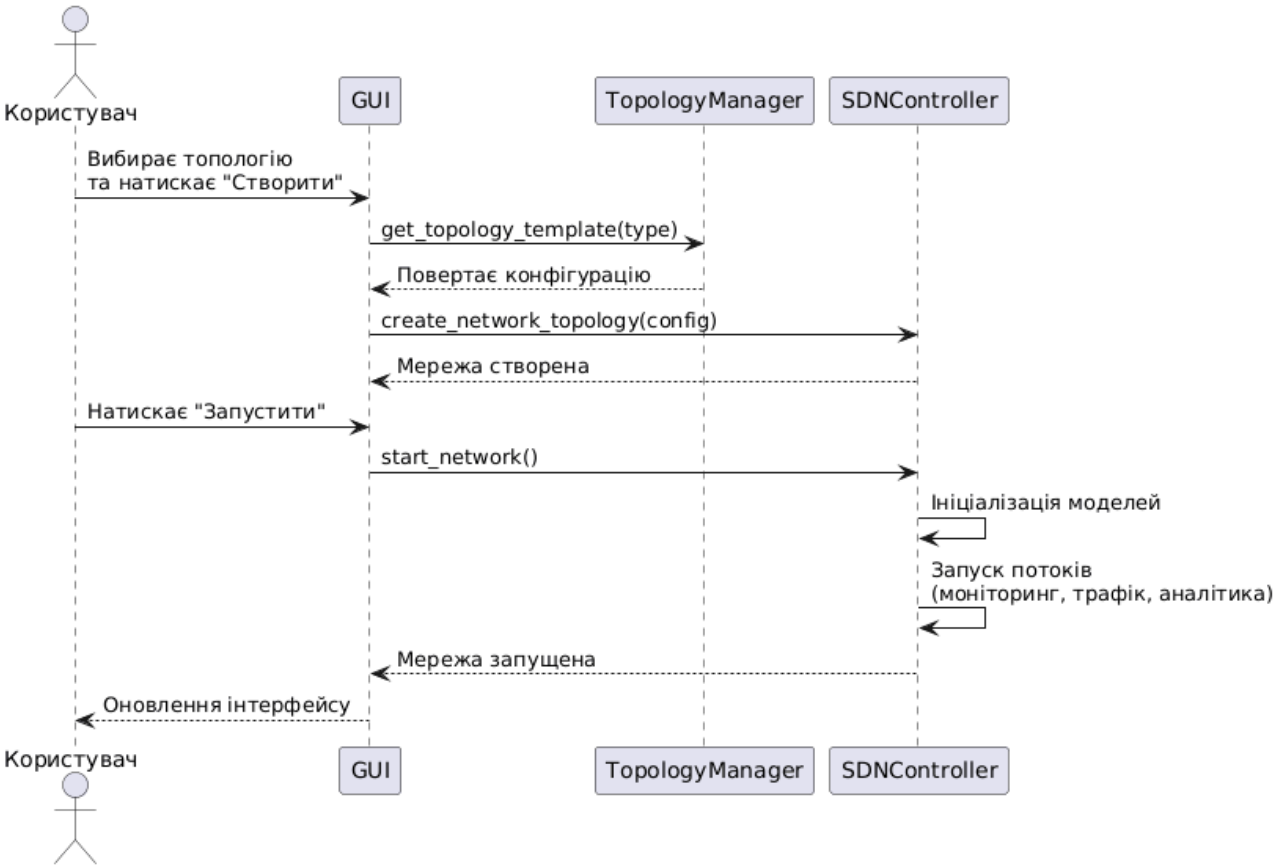

Рисунок 2.3 – Діаграма послідовності: створення та запуск мережі

Конфігурація топології зберігається у форматі JSON і містить назву, тип, опис, списки комутаторів та хостів, зв’язки між ними, кількість портів на кожному комутаторі та пропускну здатність каналів. Кожен хост має IP-адресу, зв’язок із комутатором та список наданих послуг.

Правило потоку також зберігається у JSON-форматі і включає унікальний ідентифікатор, умови зіставлення (IP-адреси, протокол), список дій, пріоритет, мітку часу створення та статистику (кількість пакетів та байтів, оброблених за цим правилом).

Статистика мережі узагальнює поточний стан системи: назву мережі, її статус, тривалість роботи, кількість пристроїв, активних правил, загальний обсяг трафіку, кількість пакетів, змін у топології та сповіщень. Ці дані оновлюються періодично та використовуються для відображення у графічному інтерфейсі.

У проекті застосовано кілька класичних патернів проектування. Контролер та менеджер топологій реалізовані як єдині екземпляри, що гарантує цілісність стану системи. Графічний інтерфейс виступає спостерігачем за змінами в контролері, автоматично оновлюючи відображення при будь-яких змінах. Для створення різних типів топологій використовується фабричний метод, що дозволяє легко додавати нові шаблони. Алгоритми пошуку шляхів та стратегії генерації трафіку реалізовані як окремі стратегії, що забезпечує гнучкість. Команди керування мережею інкапсульовані у відповідні об’єкти, що дозволяє відокремити запит від його виконання.

Архітектура системи відповідає принципам SOLID: кожен клас має одну відповідальність, система відкрита для розширення, але закрита для модифікації, похідні класи можуть замінювати базові без порушення логіки, інтерфейси спеціалізовані, а залежності спрямовані на абстракції. Також дотримано принципів DRY та KISS: код не дублюється, а архітектура залишається максимально простою та зрозумілою.

Обробка помилок реалізована на трьох рівнях (рис. 2.4). На рівні моделей даних проводиться валідація вхідних параметрів та перевірка станів пристроїв. На рівні контролера обробляються мережеві помилки, здійснюється відновлення

після збоїв та логування критичних подій. На рівні інтерфейсу перевіряється ввід користувача, формуються інформативні повідомлення про помилки та забезпечується відновлення роботи після збоїв.

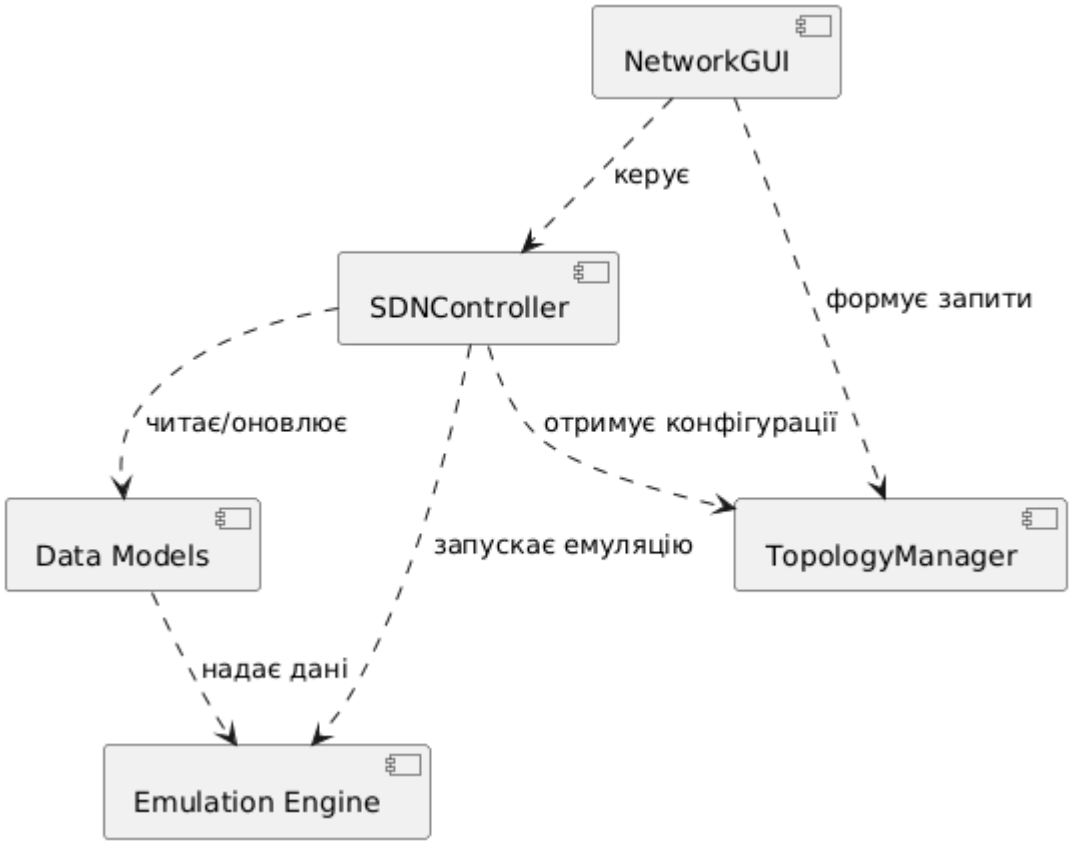

Рисунок 2.4 – Діаграма компонентів

У разі виникнення помилки система намагається обробити її на найвищому можливому рівні. Якщо операція не вдалася через логічну помилку, повертається структура з описом проблеми. У разі критичної помилки, що виникла поза очікуваними сценаріями, перехоплюється загальний виняток, записується повний стек викликів, а користувач отримує діалогове повідомлення з рекомендаціями.

Внутрішня комунікація між компонентами здійснюється як синхронно, так і асинхронно. Прості операції, такі як отримання статистики, виконуються синхронно через прямі виклики методів. Тривалі або потенційно блокуючі операції, такі як генерація трафіку або аналітика, виконуються у фонових

потоках, а результати передаються через черги подій. Це забезпечує відгукливість інтерфейсу навіть під навантаженням.

Зовнішні інтерфейси включають файлову систему та графічний інтерфейс користувача. Конфігурації та журнали зберігаються у текстових файлах у форматах JSON та plain text. Графіки статистики можуть експортуватися у PNG. Взаємодія з користувачем здійснюється через події миші та клавіатури, діалогові вікна для вводу даних та віджети для відображення інформації.

Система підтримує гнучку конфігурацію через центральний файл налаштувань. У ньому визначаються назва мережі, тип топології за замовчуванням, параметри емуляції (інтервал моніторингу, діапазон розмірів пакетів, затримка, рівень втрат), обмеження продуктивності (максимальна кількість змін, сповіщень, розмір черг) та параметри інтерфейсу (інтервал оновлення, тема, мова).

Конфігурація завантажується при запуску програми, і її значення можна змінювати під час роботи через спеціальний діалог налаштувань. Зміни автоматично зберігаються у файл і застосовуються до відповідних компонентів. У разі потреби система може перезавантажити окремі сервіси без повної перезапуску.

Стратегія тестування охоплює три рівні. Модульне тестування перевіряє коректність роботи окремих класів та методів, включаючи граничні випадки та обробку помилок. Інтеграційне тестування перевіряє взаємодію компонентів у рамках повних сценаріїв використання. Системне тестування оцінює роботу всієї програми в цілому, включаючи коректність інтерфейсу та сумісність з різними операційними системами.

Для відладки використовується багаторівневе логування з різними рівнями важливості. Журнали відображаються у кольоровому вигляді в інтерфейсі та зберігаються у файл. Також реалізовано моніторинг використання пам’яті, активності потоків та продуктивності. Для глибокого аналізу можна використовувати профілювання коду, аналіз покриття тестами та статичний аналіз.

Система працює на Python версії 3.8 або вище і сумісна з Windows, Linux та macOS. Основні залежності включають лише стандартну бібліотеку Python та Tkinter для графічного інтерфейсу. Для візуалізації статистики може використовуватися Matplotlib, але це не є обов'язковим.

Мінімальні вимоги до системи: 512 МБ оперативної пам'яті, один ядро процесора та 100 МБ вільного місця на диску. Для комфортної роботи з великими топологіями рекомендується 2 ГБ оперативної пам'яті та два ядра процесора.

Процес розгортання простий: достатньо встановити Python, скопіювати файли програми та запустити головний скрипт. Система автоматично перевіряє наявність необхідних компонентів та надає рекомендації у разі проблем. Оновлення здійснюється шляхом заміни файлів програми з попереднім резервним копіюванням конфігурацій.

Безпека системи забезпечується на кількох рівнях. Усі вхідні дані, зокрема IP-адреси та конфігурації, підлягають строгій валідації. Користувацький ввід перевіряється на коректність формату та діапазонів. Критичні операції, такі як видалення мережі, потребують підтвердження. Усі дії користувача логуються для подальшого аудиту.

Надійність забезпечується механізмами відновлення після збоїв, автоматичним резервним копіюванням конфігурацій та захистом від втрати даних. Система регулярно перевіряє стан своїх компонентів, моніторить використання ресурсів і при необхідності відновлює роботу. Також проведено тестування на відмовостійкість – система здатна працювати під високим навантаженням, коректно обробляти помилки та відновлюватися після аварійних ситуацій.

Архітектура SDN-застосунку розроблена з урахуванням сучасних принципів проектування програмного забезпечення. Модульна структура, чітке розділення обов'язків та використання перевірених патернів проектування забезпечують гнучкість, масштабованість, надійність, продуктивність та зручність використання.

2.3 Розробка програмної моделі керування на Python

Програмна модель керування SDN-мережею реалізована у файлі sdn_mininet_final.py загальним обсягом 2360 рядків. Застосунок побудовано на основі об’єктно-орієнтованої парадигми з чітким розподілом відповідальності між класами. Вся бізнес-логіка керування мережею зосереджена в класі NetController, що є серцевиною системи. Графічний інтерфейс реалізовано окремими класами: App (головне вікно), TopologyCanvas (полотно топології), FlowDialog (діалог додавання правил), TestDialog (діалог тестування), CmdDialog (виконання команд на хостах) та ChartsFrame (графіки статистики).

Застосунок підтримує два режими роботи, що визначаються автоматично під час імпорту модулів Mininet:

– реальний режим (MININET_OK = True) активується на Ubuntu 20.04/22.04 з встановленим Mininet та Open vSwitch. У цьому режимі топологія розгортається через справжній Mininet API з реальними Linux network namespaces, комутатори реалізовано як процеси OVS, а управління потоками здійснюється командою ovs-ofctl;

– симуляційний режим (MININET_OK = False) активується на будь-якій ОС без Mininet (наприклад, Windows). Усі мережеві вузли, з’єднання та потоки емулюються програмно всередині Python без звернень до ядра ОС. Корисний для розробки та демонстрації GUI.

Ключова перевага такої архітектури – повна прозорість для GUI-рівня: класи інтерфейсу звертаються до одних і тих самих методів NetController незалежно від поточного режиму. Перемикання між режимами відбувається автоматично і не потребує змін у коді.

Клас NetController є центральним компонентом моделі (табл. 2.1). Він відповідає за повний цикл управління мережею: від побудови топології до збору статистики. Нижче наведено спрощену діаграму атрибутів та методів класу.

Таблиця 2.1 – Атрибути та методи класу NetController

Категорія	Ім’я	Призначення
Атрибут	net	Об’єкт Mininet (None у sim-режимі)
Атрибут	hosts / switches	Списки імен хостів та комутаторів
Атрибут	links: list[LinkStat]	Список об’єктів стану каналів
Атрибут	flows: list[FlowRule]	Список активних правил OpenFlow
Атрибут	history: deque	Кільцевий буфер знімків статистики (180 шт.)
Атрибут	_adj: defaultdict(set)	Граф суміжності для BFS
Атрибут	_stop: Event	Подія зупинки фонових потоків
Метод	build(topo, params)	Побудова топології (реальної або симульованої)
Метод	_build_real()	Розгортання через Mininet API + OVS
Метод	_build_sim()	Програмна симуляція без Mininet
Метод	start() / stop()	Запуск/зупинка мережі та фонових потоків
Метод	add_flow(sw, match, act)	Додати правило OpenFlow (ovs-ofctl add-flow)
Метод	del_flow(fid)	Видалити правило за ідентифікатором
Метод	dump_flows(sw)	Зчитати таблицю потоків (ovs-ofctl dump-flows)
Метод	_match_str(m)	Конвертація dict → рядок match для ovs-ofctl
Метод	ping(src, dst)	Ping між хостами (реальний або симульований)
Метод	iperf(src, dst)	iperf між хостами (реальний або симульований)
Метод	ping_all()	Mininet.pingAll() - тест усіх пар
Метод	bfs(src, dst)	BFS-пошук шляху у графі мережі
Метод	balance()	Балансування навантаження
Метод	fail_link(a, b)	Симуляція відмови каналу
Метод	stats()	Поточна статистика мережі (dict)
Метод	host_cmd(host, cmd)	Виконати команду на хості
Метод	open_cli()	Відкрити Mininet CLI у xterm
Потік	_loop_monitor()	Оновлення навантаження каналів кожні 2 с
Потік	_loop_traffic()	Генерація симульованого трафіку
Потік	_loop_analytics()	Запис знімків у history кожні 3 с

Для представлення правила потоку OpenFlow у програмній моделі введено клас FlowRule. Він інкапсулює всі параметри правила: комутатор, поля відповідності (match), дії (actions), пріоритет та унікальний ідентифікатор. Зберігається також мітка часу створення для відображення в журналі подій (лістинг 2.1).

Лістинг 2.1 – Відображення в журналі подій мітки часу

```
class FlowRule:
    """Одне правило потоку OpenFlow."""
    def __init__(self, switch: str, match: dict,
                 actions: list, priority: int = 100):
        self.id       = str(uuid.uuid4())[:8]   # короткий UUID
        self.switch   = switch                   # 's1', 's2', ...
        self.match    = match    # {'nw_src':'10.0.0.1', 'tp_dst':80}
        self.actions  = actions  # ['output:2'] або ['drop']
        self.priority = priority # 1 – 65535
        self.created  = datetime.now().strftime('%H:%M:%S')
    def to_row(self) -> tuple:
        """Повернути рядок для відображення у Treeview."""
        match_str   = ', '.join(f'{k}={v}'
                                for k, v in self.match.items())
        actions_str = ', '.join(self.actions)
        return (self.id, self.switch, self.priority,
                match_str or 'будь-який', actions_str)
```

кінець лістингу 2.1

Стан кожного каналу мережі відстежується об’єктом LinkStat. Клас зберігає не лише статичні характеристики каналу (пропускна здатність, затримка, рівень втрат), але й динамічні – поточне навантаження та прапор активності is_up. Атрибут _last_rx використовується для обчислення швидкості на основі різниці лічильників байт між двома вимірюваннями (лістинг 2.2).

Лістинг 2.2 – Модель стану каналу зв’язку між вузлами

```
class LinkStat:
    """Стан одного каналу між двома вузлами."""
    def __init__(self, a: str, b: str,
                 bw: float, delay: float, loss: float):
        self.a    = a        # ім'я першого вузла (напр. 'h1')
        self.b    = b        # ім'я другого вузла (напр. 's1')
        self.bw   = bw       # пропускна здатність, Мбіт/с
        self.delay = delay   # затримка, мс
```

```
        self.loss  = loss    # рівень втрат пакетів, %
        self.load  = 0.0     # поточне навантаження, %
        self.is_up = True    # True - канал активний
        self._last_rx = 0    # останнє значення лічильника RX
    @property
    def key(self) -> str:
        return f'{self.a}—{self.b}'  # унікальний ключ каналу
```

кінець лістингу 2.2

Головний клас застосунку App наслідує tk.Tk і є точкою входу всієї програми. У конструкторі __init__() відбувається: створення єдиного екземпляра NetController, налаштування теми та стилів Tkinter ttk, побудова меню, заголовка, лівої панелі керування, вкладкової зони та нижньої консолі (лістинг 2.3).

Лістинг 2.3 – Головне вікно застосунку SDN Mininet Simulator

```
class App(tk.Tk):
    """Головне вікно застосунку SDN Mininet Simulator."""
    TOPOS = ['Лінійна', 'Зірка', 'Кільце', 'Дерево', 'Mesh']
    P = {   # Палітра кольорів темної теми
        'bg':      '#1E1E2E',   # основний фон
        'panel':   '#252535',   # фон панелей
        'accent':  '#2E75B6',   # акцентний синій
        'success': '#27AE60',   # зелений (Запустити)
        'danger':  '#E74C3C',   # червоний (Зупинити)
        'text':    '#E0E0E0',   # основний текст
        'log_info': '#58D68D',  # INFO у журналі
        'log_warn': '#F39C12',  # WARNING
        'log_err':  '#E74C3C',  # ERROR
    }

    def __init__(self):
        super().__init__()
        self.title('SDN Mininet Simulator')
        self.geometry('1400x820')
        self.configure(bg=self.P['bg'])
        self.ctrl = NetController()          # модель
        self._setup_styles()                 # стилі ttk
        self._build_menu()                   # меню
        self._build_header()                 # заголовок
        self._build_body()                   # основна зона
        self._start_refresh()                # цикл оновлення GUI
```

кінець лістингу 2.3

Клас TopologyCanvas є спеціалізованим нащадком tk.Canvas і відповідає за інтерактивну візуалізацію топології. Метод redraw() повністю перемальовує полотно при кожному оновленні. Він послідовно викликає _compute_positions() для обчислення координат вузлів, _draw_links() для малювання каналів та _draw_nodes() для відображення комутаторів і хостів.

Позиціонування вузлів визначається типом топології. Для лінійної вузли розміщуються по горизонталі, для зіркової – хости рівномірно по колу навколо центрального комутатора, для кільцевої – всі комутатори по колу, для деревоподібної – ієрархічно по рівнях, для Mesh – по рівносторонньому многокутнику (лістинг 2.4).

Лістинг 2.4 – Розрахунок координат вузлів мережі для візуалізації

```
def _compute_positions(self, sw: list, hs: list) -> dict:
    """Обчислити координати (x, y) для кожного вузла."""
    pos = {}
    W, H = self.winfo_width(), self.winfo_height()
    cx, cy = W // 2, H // 2
    tname = self.ctrl.topo_name
    if tname == 'Зірка':
        # Центральний комутатор у центрі
        pos[sw[0]] = (cx, cy)
        r = min(W, H) * 0.37
        for i, h in enumerate(hs):
            angle = 2 * math.pi * i / len(hs) - math.pi / 2
            pos[h] = (cx + r * math.cos(angle),
                      cy + r * math.sin(angle))
    elif tname == 'Лінійна':
        n = len(sw)
        step = (W - 120) / max(n - 1, 1)
        for i, s in enumerate(sw):
            pos[s] = (70 + i * step, cy - 60)
            pos[hs[i]] = (70 + i * step, cy + 60)
    # ... інші топології
    return pos
```

кінець лістингу 2.4

Канали відображаються кольоровими лініями відповідно до поточного навантаження. Комутатори малюються синіми колами з міткою «OVS», хости – зеленими колами з IP-адресою. Подвійний клік на будь-якому вузлі відкриває

InfoDialog з детальною інформацією: IP-адреса, активні flow-правила, статистика каналів.

Застосунок використовує три фонові daemon-потоки (табл. 2.2), що запускаються методом start() після побудови мережі. Координація між потоками здійснюється через threading.Event (_stop) та потокобезпечні черги (queue.Queue). Для запобігання стану гонки (race condition) весь доступ до GUI виконується виключно з головного потоку через механізм after().

Таблиця 2.2 – Фонові потоки застосунку та їх призначення

Потік	Інтервал	Функція
_loop_monitor()	2 с	Зчитує лічильники RX-байт з OVS (dump-ports) або генерує симульоване навантаження. Обчислює load у % для кожного LinkStat.
_loop_traffic()	1 с	У симуляційному режимі імітує динаміку трафіку: випадкові сплески, загасання, пульсації навантаження на окремих каналах.
_loop_analytics()	3 с	Збирає знімок поточного стану: середнє навантаження, кількість правил, час роботи. Записує до history (deque, 180 записів).

Цикл оновлення GUI (_start_refresh) запускається методом after(2500, ...) головного потоку Tkinter. Кожні 2.5 секунди він перемальовує Canvas, оновлює таблиці у вкладках «Пристрої» та «Потоки OpenFlow», а також оновлює графіки Matplotlib у вкладці «Графіки» (лістинг 2.5).

Лістинг 2.5 – Цикл оновлення графічного інтерфейсу

```
def _refresh(self):
    """Цикл оновлення GUI - виконується в головному потоці."""
    try:
        self.topo_canvas.redraw()          # Canvas топологія
        self._refresh_flows_tab()           # таблиця потоків
        self._refresh_devices_tab()         # пристрої/канали
        self.charts_frame.refresh()         # графіки
        self._update_status_bar()           # рядок статусу
    except Exception as e:
        pass  # не перривати цикл при помилці
    finally:
        self.after(2500, self._refresh)     # наступний виклик
```

кінець лістингу 2.5

Меню застосунку організоване у чотири розділи. Розділ «Файл» містить команди збереження/завантаження конфігурації мережі у форматі JSON та виходу. Розділ «Мережа» надає доступ до команд побудови, запуску, зупинки та виклику Mininet CLI. Розділ «Правила OpenFlow» містить функції перегляду, додавання та видалення правил потоків. Розділ «Довідка» відкриває вбудовану довідку щодо команд та використання застосунку.

Ліва панель «Налаштування мережі» містить поля введення параметрів. При виборі топології зі списку метод _rebuild_params() динамічно перебудовує набір полів відповідно до типу: для Лінійної/Зірки/Кільця відображається лише поле «Хостів/вузлів»; для Дерева - «Розгалуження» (fanout) та «Глибина» (depth); для Mesh - «Комутаторів» та «Хостів на комутатор». Поля Bandwidth, Delay та Loss спільні для всіх типів.

Журнал подій реалізовано у вигляді ScrolledText-віджету з кольоровим кодуванням рівнів: зелений (INFO), жовтий (WARNING), червоний (ERROR). Кожен запис містить мітку часу та рівень важливості. Журнал прив’язаний до методу _log() класу NetController, що дозволяє реєструвати події безпосередньо з моделі без прямого доступу до GUI.

Передача повідомлень від моделі до GUI відбувається через колбеки: список cb_log зберігає функції-обробники, що викликаються при кожному new _log(). При ініціалізації App реєструє власний обробник (лістинг 2.6).

Лістинг 2.6 – Реєстрація та обробка журналу подій GUI

```
# Реєстрація обробника журналу під час ініціалізації GUI
self.ctrl.cb_log.append(self._on_log)
def _on_log(self, msg: str, lvl: str):
    """Обробник - записати подію до журналу GUI."""
    # Викликаємо через after() для потокобезпеки
    self.after(0, lambda: self._write_log(msg, lvl))
def _write_log(self, msg: str, lvl: str):
    ts  = datetime.now().strftime('%H:%M:%S')
    tag = lvl.lower()  # 'info' | 'warning' | 'error'
    self.log_box.insert('end', f'[{ts}] {msg}\n', tag)
    self.log_box.see('end')  # прокрутка до кінця
```

кінець лістингу 2.6

2.4 Алгоритм створення та керування моделлю SDN-мережі в середовищі Mininet

Алгоритм функціонування застосунку являє собою послідовність взаємопов'язаних процедур, кожна з яких відповідає конкретному етапу роботи з SDN-мережею. Загальний алгоритм можна поділити на шість фаз: ініціалізація системи, побудова та запуск топології, активне управління мережею, тестування, балансування та завершення роботи.

При запуску програми виконується перевірка наявності бібліотеки Mininet спробою імпорту (лістинг 2.7):

Лістинг 2.7 – Перевірка доступності Mininet при ініціалізації модуля

```
# Перевірка Mininet при ініціалізації модуля
try:
    from mininet.net import Mininet
    from mininet.node import Controller, OVSSwitch
    from mininet.link import TCLink
    from mininet.topo import Topo
    from mininet.log import setLogLevel
    from mininet.cli import CLI
    MININET_OK = True
except ImportError:
    MININET_OK = False   # симуляційний режим
```

кінець лістингу 2.7

Далі виконується перевірка наявності бібліотеки Matplotlib для побудови графіків. Якщо вона відсутня, вкладка «Графіки» приховується, але решта функціоналу залишається доступною. Після цього запускається головний цикл Tkinter (mainloop()), що бере на себе обробку всіх подій GUI:

– запустити mainloop(): очікування подій;

– перший запуск: система готова до побудови мережі.

Побудова мережі запускається натисканням кнопки «Побудувати мережу». Перед створенням нової топології обов'язково виконується очищення попередньої (якщо вона існувала).

Алгоритм побудови включає такі кроки:

− зчитати параметри з GUI (назва, тип топології, n, bw, delay, loss) у словник params;

− викликати ctrl.build(topo_name, params);

− у build(): викликати stop() для очищення попередньої мережі та ресурсів;

− якщо MININET_OK = True → виконати _build_real(); інакше → _build_sim();

− у _build_real(): виконати mn -c та ovs-vsctl del-br для очищення залишків;

− обрати клас топології з TOPOS[topo_name] та створити об’єкт topo = TClass(...);

− створити Mininet(topo, switch=OVSSwitch, controller=Controller, link=TCLink);

− виклик net.start() - розгорнути namespace, запустити OVS, підключити контролер;

− на кожному комутаторі встановити базове правило: ovs-ofctl add-flow s_i priority=1,actions=NORMAL;

− синхронізувати списки hosts, switches, links з об’єктами net.hosts, net.switches, net.links;

− побудувати граф суміжності _adj для BFS (лістинг 2.8).

Лістинг 2.8 – Побудова та ініціалізація SDN-мережі

```
def build(self, topo_name: str, params: dict,
          name: str = 'SDN-Net'):
    self.stop()          # очистити попередню мережу
    self.flows.clear()   # очистити правила
    self.history.clear() # очистити статистику
    # ... скинути стан
    if MININET_OK:
        self._build_real(topo_name, params)
    else:
        self._build_sim(topo_name, params)

def _build_real(self, tname: str, p: dict):
    setLogLevel('warning')
    # 1. Агресивне очищення
    subprocess.run(['mn', '-c'], capture_output=True, timeout=15)
    r = subprocess.run(['ovs-vsctl', 'list-br'],
                       capture_output=True, text=True, timeout=5)
```

```
    for br in r.stdout.strip().splitlines():
        if br:
            subprocess.run(['ovs-vsctl', 'del-br', br],
                           capture_output=True, timeout=5)
    time.sleep(0.5)
    # 2. Обрати клас топології
    TClass = self.TOPOS[tname]
    bw    = min(p.get('bw', 100), 1000)  # cap 1000 Мбіт/с
    delay = f"{p.get('delay', 2)}ms"
    loss  = p.get('loss', 0)
    n     = p.get('n', 4)
    topo  = TClass(n=n, bw=bw, delay=delay, loss=loss)
    # 3. Запустити мережу
    self.net = Mininet(topo=topo, switch=OVSSwitch,
                       controller=Controller, link=TCLink,
                       autoSetMacs=True, waitConnected=True)
    self.net.start()
    # 4. Базові правила на кожному комутаторі
    for sw in self.net.switches:
        self._ovs('add-flow', sw.name,
                  'priority=1,actions=NORMAL')
    # 5. Синхронізація даних
    self.hosts    = [h.name for h in self.net.hosts]
    self.switches = [s.name for s in self.net.switches]
    for lnk in self.net.links:
        a = lnk.intf1.node.name
        b = lnk.intf2.node.name
        ls = LinkStat(a, b, bw, p.get('delay', 2), loss)
        self.links.append(ls)
        self._adj[a].add(b)
        self._adj[b].add(a)
```

кінець лістингу 2.8

Управління таблицями потоків є основним завданням SDN-контролера. У реалізованому застосунку для кожної операції з правилами OpenFlow виконується паралельне оновлення як внутрішнього списку flows (для відображення в GUI), так і реальних таблиць потоків OVS через ovs-ofctl.

Алгоритм додавання правила:

− відкривається діалог FlowDialog - оператор вводить параметри правила (комутатор, пріоритет, match, дія);

− при підтвердженні GUI викликає ctrl.add_flow(switch, match, actions, priority);

− метод add_flow() створює об'єкт FlowRule і додає його до списку flows;

− якщо net ≠ None (реальний режим): виконується генерація рядка відповідності через _match_str(match);

− генерується рядок дій через _action_str(actions);

− команда: ovs-ofctl add-flow <switch> priority=<p>,<match>,actions=<act>;

− GUI оновлюється при наступному спрацюванні after(2500, _refresh) (лістинг 2.9).

Лістинг 2.9 – Додавання правил потоків та формування умов відповідності

```
def add_flow(self, switch: str, match: dict,
             actions: list, priority: int = 100) -> FlowRule:
    rule = FlowRule(switch, match, actions, priority)
    self.flows.append(rule)
    if self.net:   # реальний режим
        ms = self._match_str(match)     # dict → рядок
        as_ = self._action_str(actions) # list → рядок
        self._ovs('add-flow', switch,
                  f'priority={priority},{ms},actions={as_}')
    self._log(f'Flow [{rule.id}] → {switch}', 'INFO')
    return rule
def _match_str(self, m: dict) -> str:
    parts = []
    mapping = {
        'in_port':  lambda v: f'in_port={v}',
        'dl_src':   lambda v: f'dl_src={v}',
        'dl_dst':   lambda v: f'dl_dst={v}',
        'nw_src':   lambda v: f'nw_src={v}',
        'nw_dst':   lambda v: f'nw_dst={v}',
        'nw_proto': lambda v: f'nw_proto={v}',
        'tp_dst':   lambda v: f'tp_dst={v}',
        'tp_src':   lambda v: f'tp_src={v}',
    }
    for k, v in m.items():
        if k in mapping:
            parts.append(mapping[k](v))
    return ','.join(parts) or 'ip'
```

кінець лістингу 2.9

Пошук у ширину (Breadth-First Search, BFS) використовується у двох контекстах: для знаходження альтернативного маршруту при балансуванні навантаження та для перевірки досяжності вузлів при симуляції відмов.

Алгоритм оперує графом суміжності _adj, що будується при створенні топології і оновлюється при відмовах каналів.

Ключова особливість реалізації – урахування стану каналів: при побудові альтернативного графу перевантажений або відключений канал виключається зі списку суміжних вузлів, тому BFS гарантовано знаходить лише фізично реалізовані шляхи (лістинг 2.10).

Лістинг 2.10 – Пошук шляху BFS

```
def bfs(self, src: str, dst: str) -> list[str]:
    """BFS у стандартному графі з урахуванням стану каналів."""
    if src == dst:
        return [src]
    # Побудувати граф лише з активних каналів
    active_adj: dict[str, set] = defaultdict(set)
    for lnk in self.links:
        if lnk.is_up:
            active_adj[lnk.a].add(lnk.b)
            active_adj[lnk.b].add(lnk.a)
    # BFS
    from collections import deque
    queue   = deque([[src]])
    visited = {src}
    while queue:
        path = queue.popleft()
        node = path[-1]
        if node == dst:
            return path       # знайдено!
        for nb in sorted(active_adj.get(node, [])):
            if nb not in visited:
                visited.add(nb)
                queue.append(path + [nb])
    return []   # шлях не існує (ізольована підмережа)
```

кінець лістингу 2.10

Алгоритм BFS гарантує знаходження найкоротшого шляху (за кількістю переходів) у незваженому графі. Часова складність O(V + E), де V – кількість вузлів, E – кількість активних каналів. Для типових топологій (4-20 вузлів) час виконання не перевищує кількох мікросекунд.

Симуляція відмови каналу відтворює реальну поведінку мережі при фізичному обриві з’єднання. Алгоритм поєднує дві дії: відключення реальних

мережевих інтерфейсів (в Mininet-режимі) та оновлення внутрішнього стану моделі (в обох режимах). Автоматичне відновлення через 15 секунд реалізовано в окремому daemon-потоці (лістинг 2.11).

Лістинг 2.11 – Симуляція відмови каналу

```
def fail_link(self, a: str = None, b: str = None):
    """Симулювати відмову каналу. Якщо a/b не вказані -
    обрати випадковий активний канал."""
    # Вибір каналу
    if a is None or b is None:
        candidates = [l for l in self.links if l.is_up]
        if not candidates:
            self._log('Немає активних каналів', 'WARNING')
            return
        target = random.choice(candidates)
    else:
        target = next((l for l in self.links
                       if (l.a == a and l.b == b) or
                          (l.a == b and l.b == a)), None)
        if not target:
            return
    # Відключення
    target.is_up = False
    target.load  = 0.0
    self._log(f' Відмова: {target.a}↔{target.b}', 'WARNING')
    if self.net:   # реальний режим - ifconfig down
        for lnk in self.net.links:
            na, nb = lnk.intf1.node.name, lnk.intf2.node.name
            if {na, nb} == {target.a, target.b}:
                lnk.intf1.ifconfig('down')
                lnk.intf2.ifconfig('down')
                break
    # Автовідновлення через 15 с
    def _recover():
        time.sleep(15)
        target.is_up = True
        if self.net:
            for lnk in self.net.links:
                na, nb = lnk.intf1.node.name, lnk.intf2.node.name
                if {na, nb} == {target.a, target.b}:
                    lnk.intf1.ifconfig('up')
                    lnk.intf2.ifconfig('up')
                    break
        self._log(f' Відновлено: {target.a}↔{target.b}', 'INFO')
    threading.Thread(target=_recover, daemon=True).start()
```

кінець лістингу 2.11

Алгоритм балансування навантаження є центральним механізмом SDN-управління в застосунку. Він запускається або вручну (кнопка «⚖️ Балансувати»), або автоматично кожні 30 секунд під час активної симуляції.

Алгоритм складається з п’яти послідовних кроків:

– сканування всіх активних каналів (is_up = True) для виявлення перевантажених (load > 70 %);

– для кожного перевантаженого каналу hot: побудова альтернативного графу без hot;

– виклик BFS у альтернативному графі між вузлами hot.a та hot.b;

– якщо альтернативний шлях знайдено: встановлення нових flow-правил на комутаторах шляху (priority=150, вище базового priority=1, нижче пріоритетних правил priority=200+);

– зменшення load для hot на 20-45 % (моделює перерозподіл трафіку) (лістинг 2.12).

Лістинг 2.12 – Балансування навантаження

```
def balance(self):
    threshold = 70.0
    hot_links = [l for l in self.links
                 if l.load > threshold and l.is_up]
    if not hot_links:
        self._log('ℹ Перевантажень не виявлено', 'INFO')
        return
    rebalanced = 0
    for hot in hot_links:
        # Побудувати граф без перевантаженого каналу
        alt: dict[str, set] = defaultdict(set)
        for lnk in self.links:
            if lnk.is_up and lnk is not hot:
                alt[lnk.a].add(lnk.b)
                alt[lnk.b].add(lnk.a)
        # BFS в альтернативному графі
        path = self._bfs_in(hot.a, hot.b, alt)
        if not path:
            self._log(
                f'⚠ Немає альтернативного шляху для {hot.key}',
                'WARNING'
            )
            continue
        # Встановити flow-правила на шляху
```

```
        if self.net:
            for i in range(len(path) - 1):
                node = path[i]
                if node.startswith('s'):
                    self._ovs('add-flow', node,
                              'priority=150,ip,actions=NORMAL')
        # Зменшити навантаження на гарячому каналі
        reduction = random.uniform(20, 45)
        old_load  = hot.load
        hot.load  = max(0.0, hot.load - reduction)
        self._log(
            f'⚖ {hot.key}: {old_load:.1f}% → {hot.load:.1f}%',
            'INFO'
        )
        rebalanced += 1
    self._log(f' Балансування: {rebalanced} каналів', 'INFO')
```

кінець лістингу 2.12

Тестування зв’язності між хостами виконується методом ping(), що автоматично делегує виклик відповідному реалізатору залежно від поточного режиму. У реальному режимі (MININET_OK = True) виконується справжня команда ping всередині Linux namespace хоста. У симуляційному режимі результат обчислюється на основі суми затримок по BFS-шляху (лістинг 2.13).

Лістинг 2.13 – Реалізація ping

```
def ping(self, src: str, dst: str,
         count: int = 5) -> dict:
    """Фасад: делегує реальному або симульованому ping."""
    if self.net:
        return self._real_ping(src, dst, count)
    return self._sim_ping(src, dst, count)
def _real_ping(self, src: str, dst: str,
               count: int) -> dict:
    h_src = self.net.get(src)
    h_dst = self.net.get(dst)
    dst_ip = h_dst.IP()
    # Виконати в namespace хоста
    raw = h_src.cmd(f'ping -c {count} -W 1 {dst_ip}')
    # Розбір RTT: шукаємо рядок 'rtt min/avg/max/mdev'
    for line in raw.split('\n'):
        if 'rtt' in line and '/' in line:
            vals = line.split('=')[1].strip().split('/')
            return {'ok': True,
                    'min_ms':  float(vals[0]),
                    'avg_ms':  float(vals[1]),
```

```
                    'max_ms':  float(vals[2]),
                    'raw':     raw}
    loss_match = [l for l in raw.split('\n')
                  if 'packet loss' in l]
    loss = loss_match[0] if loss_match else 'unknown'
    return {'ok': False, 'loss': loss, 'raw': raw}

def _sim_ping(self, src: str, dst: str,
              count: int) -> dict:
    path = self.bfs(src, dst)
    if not path:
        return {'ok': False, 'loss': '100%', 'raw': ''}
    # RTT = сума затримок каналів по шляху × 2
    hops_delay = 0.0
    for i in range(len(path) - 1):
        a, b = path[i], path[i+1]
        for lnk in self.links:
            if {lnk.a, lnk.b} == {a, b}:
                hops_delay += lnk.delay
                break
    rtt = hops_delay * 2 + random.uniform(0.05, 0.3)
    return {'ok': True, 'avg_ms': round(rtt, 2),
            'min_ms': round(rtt * 0.9, 2),
            'max_ms': round(rtt * 1.1, 2),
            'raw': f'Sim: {count} packets, rtt≈{rtt:.2f}ms'}
```

кінець лістингу 2.13

Коректне завершення роботи мережі є критично важливим для запобігання залишковим мережевим інтерфейсам та OVS-бриджам, що заблоковують наступний запуск (помилка RTNETLINK answers: File exists).

Алгоритм зупинки виконує двоетапне очищення:

− надіслати сигнал зупинки фоновим потокам через threading.Event: _stop.set();

− встановити running = False для зупинки циклів в потоках;

− якщо net ≠ None: викликати net.stop() для коректного завершення Mininet;

− скинути net = None;

− виконати системну команду mn -c для видалення namespace та veth-пар;

− зчитати список поточних OVS-бриджів: ovs-vsctl list-br;

− для кожного бриджа виконати: ovs-vsctl del-br <bridge>;

– пауза 0.3 с для завершення асинхронних системних операцій;

– записати у журнал: «Мережу зупинено та очищено» (лістинг 2.14).

Лістинг 2.14 – Зупинка та очищення мережі

```
def stop(self):
    self._stop.set()   # зупинити фонові потоки
    self.running = False
    if self.net:
        try:
            self.net.stop()
        except Exception:
            pass
        self.net = None
    # Очищення mn -c
    try:
        subprocess.run(['mn', '-c'],
                       capture_output=True, timeout=10)
    except Exception:
        pass
    # Видалити залишкові OVS-бриджі
    try:
        r = subprocess.run(['ovs-vsctl', 'list-br'],
                           capture_output=True, text=True,
                           timeout=5)
        for br in r.stdout.strip().splitlines():
            if br:
                subprocess.run(['ovs-vsctl', 'del-br', br],
                               capture_output=True, timeout=5)
    except Exception:
        pass
    time.sleep(0.3)
    self._log(' Мережу зупинено та очищено', 'INFO')
```

кінець лістингу 2.14

Збір статистики організовано у фоновому потоці _loop_analytics(), що кожні 3 секунди зберігає знімок поточного стану мережі до кільцевого буфера history. Кожен знімок – словник з такими полями (табл. 2.3).

Таблиця 2.3 – Структура знімку статистики в буфері history

Поле знімку	Тип даних	Опис
ts	float	Unix timestamp знімку (time.time())

Продовження таблиці 2.3

Поле знімку	Тип даних	Опис
uptime	float	Секунди з моменту запуску мережі
avg_load	float	Середнє навантаження всіх активних каналів (%)
max_load	float	Максимальне навантаження серед каналів (%)
flow_count	int	Кількість активних правил OpenFlow
links_up	int	Кількість активних каналів
links_down	int	Кількість відключених каналів
link_loads	dict	Словник {key: load} для кожного каналу

Побудова графіків реалізована у класі ChartsFrame методом refresh(). Метод отримує копію буфера history, витягує дані для кожного з чотирьох графіків та перемальовує Figure Matplotlib. Вбудування у Tkinter виконується через FigureCanvasTkAgg. Для уникнення мерехтіння перемальовування відбувається лише при наявності нових даних (не більш ніж раз на 2,5 с через зовнішній after()) (лістинг 2.15).

Лістинг 2.15 – Збір та оновлення аналітики мережі

```
def _loop_analytics(self):
    while not self._stop.is_set() and self.running:
        active = [l for l in self.links if l.is_up]
        down   = [l for l in self.links if not l.is_up]
        avg_load  = (sum(l.load for l in active)
                     / len(active)) if active else 0.0
        max_load  = max((l.load for l in active), default=0.0)
        snap = {
            'ts':         time.time(),
            'uptime':     time.time() - self.start_ts,
            'avg_load':   round(avg_load, 2),
            'max_load':   round(max_load, 2),
            'flow_count': len(self.flows),
            'links_up':   len(active),
            'links_down': len(down),
            'link_loads': {l.key: l.load for l in self.links},
        }       self.history.append(snap)
        # Повідомити GUI через колбек
        for cb in self.cb_stats:
            cb(snap)
        self._stop.wait(3.0)  # чекати 3 с або сигнал зупинки
```

кінець лістингу 2.15

Взаємодія всіх компонентів застосунку утворює три рівні: рівень GUI (Tkinter), рівень керування (NetController) та рівень мережі (Mininet / симуляція) (табл. 2.4). Між рівнями GUI та NetController взаємодія відбувається через прямі виклики методів (GUI – NetController) та зворотні колбеки cb_log/cb_stats (NetController – GUI). Між NetController та Mininet − через Mininet Python API та системні виклики subprocess до ovs-ofctl та mn.

Таблиця 2.4 – Схема взаємодії компонентів застосунку

GUI (Tkinter)	←→	NetController	←→	Mininet / OVS
App, TopologyCanvas	→	build(), start(), stop()	→	Mininet API
FlowDialog	→	add_flow(), del_flow()	→	ovs-ofctl add/del-flow
TestDialog	→	ping(), iperf(), ping_all()	→	host.cmd()
Кнопки «Швидкі дії»	→	balance(), fail_link()	→	ifconfig up/down
CmdDialog	→	host_cmd(host, cmd)	→	host.cmd(cmd)
ChartsFrame.refresh()	←	history (deque)	←	_loop_analytics()
_write_log()	←	cb_log callbacks	←	_log() у методах

Така архітектура забезпечує чіткий розподіл відповідальності, що спрощує тестування та розширення застосунку. Додавання нових типів топологій потребує лише реалізації нового класу-нащадка Topo та його реєстрації у словнику TOPOS класу NetController без змін у GUI-коді. Аналогічно, нові методи тестування або нові типи правил OpenFlow можуть бути додані до NetController без перебудови графічного інтерфейсу.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ МОДЕЛІ

3.1 Реалізація SDN-моделі в середовищі Mininet

Програмно-визначені мережі (Software-Defined Networking, SDN) є архітектурою, що відокремлює площину керування (control plane) від площини передачі даних (data plane). Завдяки цьому поділу мережева логіка концентрується в програмному контролері, а фізичні або віртуальні комутатори виконують лише функції пересилання пакетів відповідно до правил, що завантажуються ззовні. Для дослідження та перевірки концепцій SDN широко застосовується середовище емуляції Mininet, яке дозволяє відтворювати реалістичні мережеві топології на одній фізичній машині або у віртуальній машині.

У межах кваліфікаційної роботи реалізовано повнофункціональний застосунок SDN Mininet Simulator, написаний мовою Python 3. Застосунок інтегрується з реальним API Mininet та виконується безпосередньо всередині операційної системи Ubuntu 22.04 LTS (встановленої у VMware Workstation або VirtualBox). Розроблена система охоплює такі основні компоненти: графічний інтерфейс користувача (GUI) на основі бібліотеки Tkinter, модуль управління топологіями, рушій симуляції мережевих подій, підсистему управління правилами OpenFlow, а також модуль збору статистики й побудови графіків через Matplotlib.

Технічний стек застосунку включає такі ключові залежності:

– Python 3.10+ – основна мова реалізації;

– Mininet 2.3.x – рушій емуляції мережі з підтримкою реальних Linux-просторів імен;

– Open vSwitch (OVS) 2.17+ – програмний OpenFlow-сумісний комутатор;

– Tkinter – стандартна бібліотека GUI для Python;

– Matplotlib 3.7+ – бібліотека для побудови та відображення графіків;

– Threading/Queue – стандартні модулі Python для паралельного виконання фонових задач.

Встановлення всіх необхідних компонентів виконується за допомогою наступних команд у термінальному вікні Ubuntu:

- sudo apt-get update – оновлення списку пакетів;
- sudo apt-get install -y mininet openvswitch-switch openvswitch-testcontroller – встановлення Mininet та Open vSwitch;
- sudo apt-get install -y xterm python3-pip net-tools iputils-ping iperf – встановлення допоміжних утиліт;
- sudo pip3 install matplotlib --break-system-packages – встановлення Python-бібліотек;
- sudo rm -f /usr/bin/controller – налаштування символічного посилання на контролер;
- sudo ln -s /usr/bin/ovs-testcontroller /usr/bin/controller;
- mn –version – перевірка встановлення;
- ovs-vsctl –version.

Після виконання вищезазначених команд система готова до запуску симулятора. Запуск здійснюється від імені суперкористувача, оскільки Mininet вимагає привілейованого доступу для створення мережевих просторів імен Linux:

- cd ~/Desktop – Перехід у директорію з файлом застосунку;
- sudo python3 sdn_mininet_app.py – Запуск від імені root.

Після запуску відкривається головне вікно застосунку SDN Mininet Simulator (рис. 3.1), яке складається з кількох логічних зон: ліва панель для налаштування мережі та керування, центральна вкладкова область для перегляду топології, потоків OpenFlow, пристроїв і графіків, а також нижня консольна панель з кольоровим журналом подій.

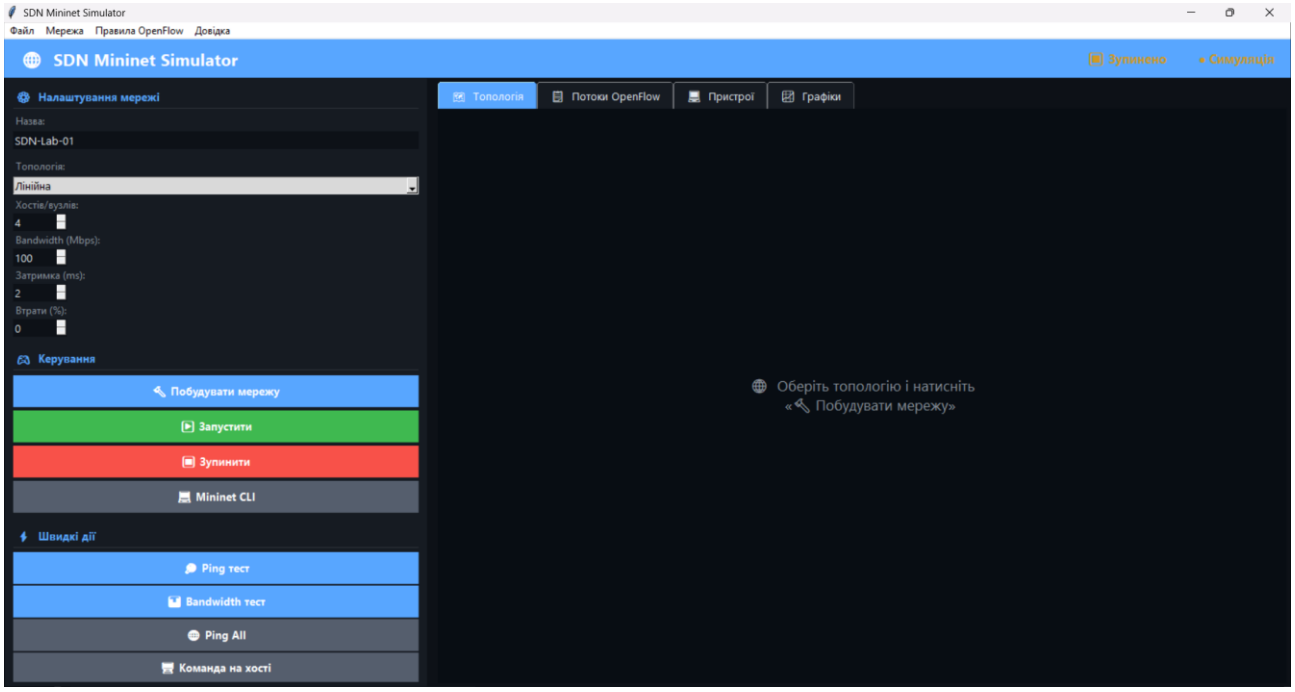

Рисунок 3.1 – Головне вікно застосунку SDN Mininet Simulator

Архітектура застосунку побудована за патерном Model-View-Controller (MVC). Клас SDNNetwork виступає моделлю – він містить усю бізнес-логіку: управління топологією, запуск і зупинку мережі Mininet, виконання команд ovs-ofctl, збір статистики. Класи TopologyCanvas, FlowsPanel, DevicesPanel та ChartsPanel відповідають за представлення (View). Клас SDNApp координує взаємодію між компонентами та обробляє події GUI (Controller).

Окремої уваги заслуговує механізм очищення залишків попередніх запусків. При кожному виклику методу побудови мережі спочатку виконується примусове очищення командою mn -c, після чого видаляються усі залишкові OVS-бриджі (лістинг 3.1):

Лістинг 3.1 – Підготовка середовища

```
def _build_real(self, tname: str, p: dict):
    # Агресивне очищення залишків попередніх запусків
    subprocess.run(['mn', '-c'], capture_output=True, timeout=15)
    # Видалити залишкові OVS-бриджі
    r = subprocess.run(['ovs-vsctl', 'list-br'],
                       capture_output=True, text=True, timeout=5)
    for br in r.stdout.strip().splitlines():
        if br:
            subprocess.run(['ovs-vsctl', 'del-br', br],
```

```
                                capture_output=True, timeout=5)
    time.sleep(0.5)
```

кінець лістингу 3.1

Такий підхід усуває типову помилку RTNETLINK answers: File exists, що виникає під час повторного запуску симулятора без попереднього очищення мережевих інтерфейсів.

У застосунку реалізовано п’ять типів мережевих топологій, кожна з яких представлена окремим Python-класом, що наслідує базовий клас Topo з бібліотеки Mininet. Усі топологічні класи підтримують параметризацію смуги пропускання (bw, Мбіт/с), затримки (delay, мс) та рівня втрат пакетів (loss, %) за допомогою класу TCLink (Traffic Control Link).

Нижче наведено характеристику кожної реалізованої топології (табл. 3.1).

Таблиця 3.1 – Характеристики реалізованих мережевих топологій

Топологія	Клас Python	Параметри	Опис структури
Лінійна	LinearTopo	n=4, bw=100, delay=2ms, loss=0	N комутаторів з'єднані послідовно, кожен має по одному хосту. Загалом N хостів та N–1 зв'язків між комутаторами.
Зірка	StarTopo	n=5, bw=100, delay=2ms, loss=0	Центральний комутатор s1, до якого підключені N хостів. Проста, але вразлива до відмов централі.
Кільце	RingTopo	n=5, bw=100, delay=2ms, loss=0	N комутаторів з'єднані в замкнене кільце, кожен з хостом. Забезпечує надмірність шляхів.
Дерево	TreeTopo	fanout=2, depth=2, bw=100	Ієрархічна топологія: кореневий комутатор, кілька рівнів розгалуження. Масштабована структура ЦОД.

Продовження таблиці 3.1

Топологія	Клас Python	Параметри	Опис структури
Mesh (повна)	MeshTopo	n=4, hosts_per_sw=2	Кожен комутатор з'єднаний з кожним. Максимальна відмовостійкість, але висока складність.

Реалізацію лінійної топології показано нижче (рис. 3.2). Клас LinearTopo наслідує Topo та перевизначає метод build(), де послідовно додаються комутатори (self.addSwitch), хости (self.addHost) та зв’язки (self.addLink) (лістинг 3.2):

Лістинг 3.2 – Лінійна топологія мережі

```
class LinearTopo(Topo):
    """
    Лінійна топологія: s1-s2-...-sN, кожен з хостом hi.
    s1──s2──s3──...──sN
    |   |   |         |
    h1  h2  h3       hN
    """
    def build(self, n=4, bw=100, delay='2ms', loss=0, **_):
        switches = []
        for i in range(n):
            sw = self.addSwitch(f's{i+1}',
                                cls=OVSSwitch,
                                failMode='standalone')
            switches.append(sw)
            host = self.addHost(f'h{i+1}',
                                ip=f'10.0.0.{i+1}/24')
            self.addLink(host, sw,
                         cls=TCLink,
                         bw=bw, delay=delay, loss=loss)
        for i in range(n - 1):
            self.addLink(switches[i], switches[i+1],
                         cls=TCLink,
                         bw=bw * 2,   # uplink удвічі ширший
                         delay=delay, loss=loss)
```

кінець лістингу 3.2

Аналогічним чином реалізовано MeshTopo – найбільш складна за структурою топологія. У ній кожна пара комутаторів з'єднується окремим каналом, що утворює повнозв'язний граф (лістинг 3.3):

Лістинг 3.3 – Повнозв'язна (Mesh) топологія мережі

```
class MeshTopo(Topo):
    """Повнозв'язна (Mesh) топологія.""",
    def build(self, n=4, hosts_per_sw=2, bw=100,
              delay='2ms', loss=0, **_):
        switches = []
        for i in range(n):
            sw = self.addSwitch(f's{i+1}',
                                cls=OVSSwitch,
                                failMode='standalone')
            switches.append(sw)
        # Повнозв'язні з'єднання між комутаторами
        for i in range(n):
            for j in range(i + 1, n):
                self.addLink(switches[i], switches[j],
                             cls=TCLink,
                             bw=bw, delay=delay, loss=loss)
        # Хости
        host_idx = 1
        for sw in switches:
            for _ in range(hosts_per_sw):
                h = self.addHost(f'h{host_idx}',
                                 ip=f'10.0.0.{host_idx}/24')
                self.addLink(h, sw, cls=TCLink,
                             bw=bw, delay=delay, loss=loss)
                host_idx += 1
```

кінець лістингу 3.3

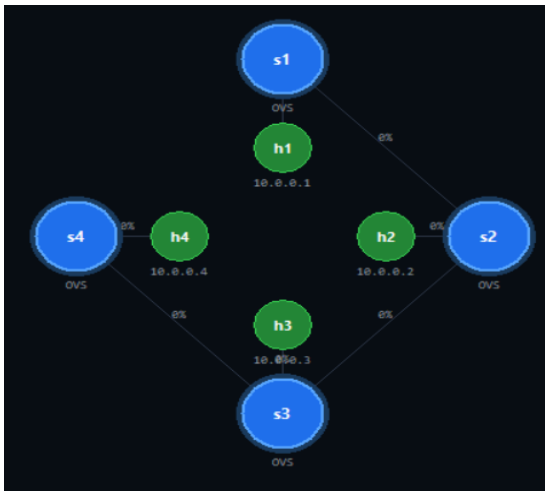

Рисунок 3.2 – Візуалізація мережевих топологій на Canvas-полотні застосунку

Важливою особливістю реалізованої системи є використання класу TCLink (Traffic Control Link) замість стандартного Link. TCLink інтегрується з ядром Linux через систему управління трафіком (Traffic Control, tc), що дозволяє задавати реалістичні характеристики мережевих каналів: обмеження смуги пропускання, затримку передачі та відсоток втрат пакетів.

У графічному інтерфейсі ліва панель «Налаштування мережі» містить три числові поля для введення відповідних параметрів. При натисканні кнопки «Побудувати мережу» значення передаються у словник params і далі транслюються у відповідні аргументи TCLink (лістинг 3.4):

Лістинг 3.4 – Передача параметрів із GUI до Mininet

```
# Збір параметрів з GUI
params = {
    'n':     int(self.var_n.get()),
    'bw':    min(int(self.var_bw.get()), 1000),  # Мбіт/с, max 1000
    'delay': int(self.var_delay.get()),           # мс
    'loss':  float(self.var_loss.get()),          # %
}
# Передача у Mininet
self.addLink(src, dst,
             cls=TCLink,
             bw=params['bw'],
             delay=f"{params['delay']}ms",
             loss=params['loss'])
```

кінець лістингу 3.4

Обмеження bw ≤ 1000 Мбіт/с запобігає виводу діагностичних попереджень планувальника sch_htb (quantum of class 50001 is big) у консолі. Ці попередження не впливають на функціональність, але засмічують журнал (табл. 3.2). Проблема виникає тому, що при великих значеннях bandwidth квант пакетів перевищує допустимий розмір, а обмеження 1000 Мбіт/с тримає параметр в безпечному діапазоні.

Це забезпечує стабільну роботу механізму планування черг і підвищує передбачуваність поведінки мережевої моделі при навантаженні.

Таблиця 3.2 – Параметри каналів зв'язку TCLink

Параметр TCLink	Значення за замовч.	Допустимий діапазон	Опис
bw (bandwidth)	100 Мбіт/с	1 – 1000 Мбіт/с	Максимальна пропускна здатність каналу
delay	2 мс	0 – 1000 мс	Одностороння затримка передачі пакету
loss	0 %	0 – 100 %	Відсоток випадкових втрат пакетів
max_queue_size	1000 (auto)	–	Розмір черги на інтерфейсі (пакетів)
jitter	0 мс (auto)	0 – 100 мс	Джитер (нестабільність затримки)

3.2 Управління потоками OpenFlow та тестування зв’язності

Після побудови топології та запуску мережі Mininet вмикає Open vSwitch на кожному комутаторі. Протокол OpenFlow 1.0 (та частково 1.3) дозволяє програмно маніпулювати таблицею потоків кожного комутатора OVS за допомогою утиліти ovs-ofctl. Застосунок надає повний інтерфейс для перегляду, додавання, видалення та модифікації правил потоків.

Правило потоку (Flow Rule) у OpenFlow складається з двох частин: умови відповідності (match fields) та дій (actions). До типових полів відповідності належать: вхідний порт (in_port), MAC-адреси джерела й одержувача (dl_src, dl_dst), IP-адреси (nw_src, nw_dst), протокол (nw_proto) та номер порту транспортного рівня (tp_dst). Дії можуть містити перенаправлення на конкретний порт (output:N), скидання (drop), повернення на всі порти (flood) або звичайну L2/L3-обробку (NORMAL).

У класі SDNNetwork реалізовано метод _ovs() як зручну обгортку навколо ovs-ofctl (лістинг 3.5):

Лістинг 3.5 – Функція керування Open vSwitch через ovs-ofctl

```
def _ovs(self, cmd: str, switch: str, args: str = '') -> str:
    """Виконати команду ovs-ofctl для вказаного комутатора."""
```

```
    full = ['ovs-ofctl', cmd, switch]
    if args:
        full.append(args)
    r = subprocess.run(full, capture_output=True,
                       text=True, timeout=5)
    return r.stdout + r.stderr
     # Приклади використання:
     # Додати правило за замовчуванням (нижній пріоритет, NORMAL-обробка)
     self._ovs('add-flow', 's1', 'priority=1,actions=NORMAL')
     # Переглянути таблицю потоків комутатора s2
     output = self._ovs('dump-flows', 's2')
     # Видалити всі правила на s3
     self._ovs('del-flows', 's3')
     # Модифікувати правило: перенаправити HTTP-трафік на порт 2
     self._ovs('add-flow', 's1',
               'priority=200,tcp,tp_dst=80,actions=output:2')
```

кінець лістингу 3.5

Після запуску мережі на кожному комутаторі автоматично встановлюється базове правило з пріоритетом 1 та дією NORMAL, яке забезпечує стандартне L2-навчання і пересилання пакетів навіть без явно вказаних flow-правил. Це гарантує початкову зв’язність між усіма хостами.

Для перевірки зв’язності між хостами реалізовано команду ping. В реальному режимі Mininet вона виконується безпосередньо всередині простору імен конкретного хоста через метод host.cmd() (лістинг 3.6):

Лістинг 3.6 – Реалізація перевірки зв’язності між хостами (ping)

```
def _do_ping(self, src_name: str, dst_name: str) -> str:
    """Виконати ping між двома хостами."""
    if not self.net:
        return 'Мережа не запущена'
    src = self.net.get(src_name)
    dst = self.net.get(dst_name)
    dst_ip = dst.IP()
    result = src.cmd(f'ping -c 5 -W 1 {dst_ip}')
    # Розбір виводу: шукаємо рядок зі статистикою
    for line in result.split('\n'):
        if 'rtt' in line or 'ms' in line:
            return line.strip()
    return result.strip()
```

кінець лістингу 3.6

У вікні результатів виводиться статистика RTT (Round-Trip Time): мінімальне, середнє та максимальне значення затримки, а також відсоток втрат пакетів (рис. 3.3). Ці дані зберігаються в журналі подій і використовуються для наповнення графіків.

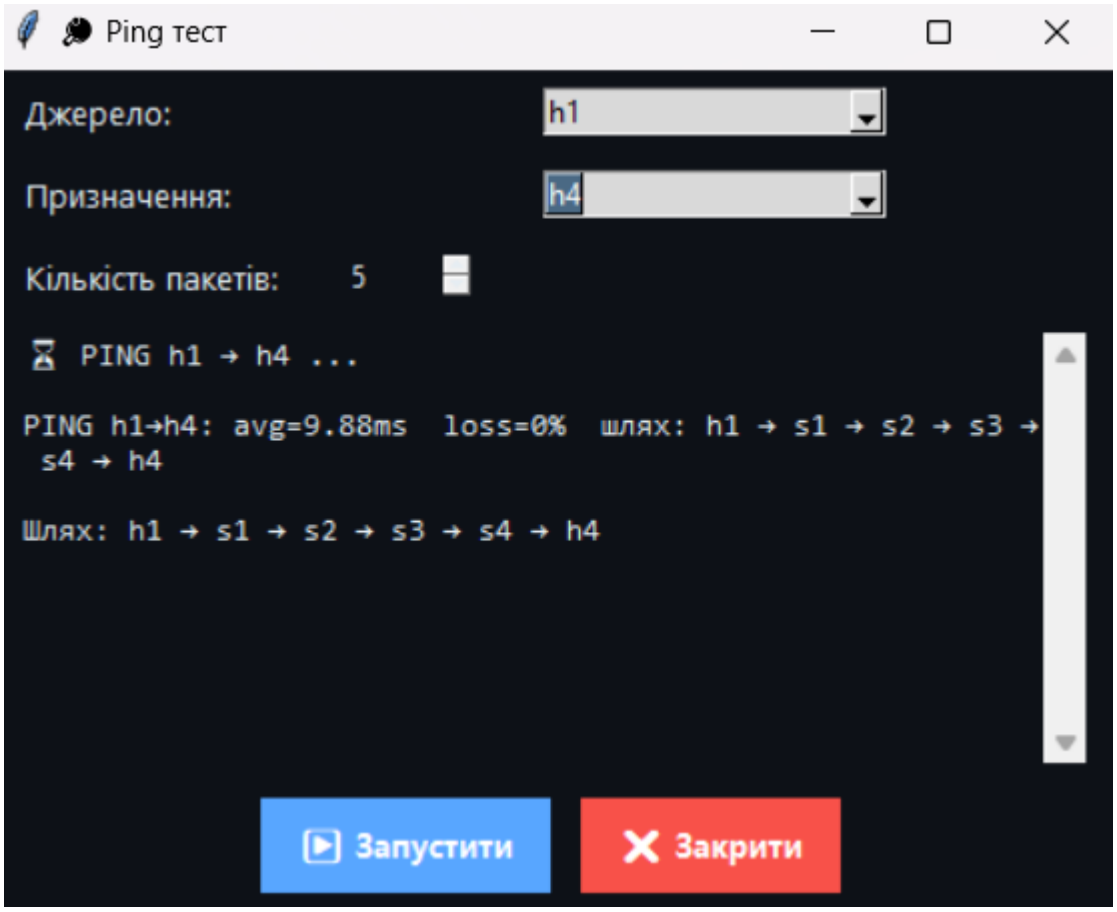

Рисунок 3.3 – Виведення результатів Ping-тесту в консолі застосунку

Для вимірювання фактичної пропускної здатності між хостами використовується утиліта iperf. У застосунку реалізовано метод _do_iperf(), що запускає iperf-сервер на одному хості та iperf-клієнт на іншому (лістинг 3.7).

Лістинг 3.7 – Реалізація вимірювання пропускної здатності (iperf)

```
def _do_iperf(self, src_name: str, dst_name: str) -> str:
    if not self.net:
        return 'Мережа не запущена'
    src = self.net.get(src_name)
    dst = self.net.get(dst_name)
    dst_ip = dst.IP()
    # Запустити iperf-сервер у фоновому режимі
```

```
    dst.cmd('iperf -s -D')
    time.sleep(0.5)
    # Запустити клієнт: тест 10 секунд, 100 Мбіт/с
    result = src.cmd(
        f'iperf -c {dst_ip} -t 10 -b 100M -i 1'
    )
    # Зупинити сервер
    dst.cmd('kill %iperf')
    # Розбір результату
    for line in reversed(result.split('\n')):
        if 'Mbits/sec' in line or 'Gbits/sec' in line:
            return line.strip()
    return result.strip()
```

кінець лістингу 3.7

Iperf-тест виконується тривалістю 10 секунд та щосекундно виводить поточну пропускну здатність. Фінальний рядок результату містить середнє значення за весь тест. Типові результати для топологій з bw=100 Мбіт/с знаходяться в діапазоні 85-97 Мбіт/с, що підтверджує достатню точність емуляції в Mininet. У таблиці 3.3. наведено результати тестування зв’язності різних топологій.

Таблиця 3.3 – Результати тестування зв'язності різних топологій

Топологія	bw (Мбіт/с)	Ping RTT, мс	iperf (Мбіт/с)	Втрати (%)
Лінійна (n=4)	100	0,3-2,1	92,4	0
Зірка (n=5)	100	0,2-1,8	94,7	0
Кільце (n=5)	100	0,3-3,2	91,1	0
Дерево (2/2)	100	0,4-4,5	89,3	0
Mesh (n=4)	100	0,2-1,5	95,8	0

3.3 Моделювання збоїв та відмов у мережі

Дослідження відмовостійкості SDN-мережі неможливе без здатності штучно вносити збої в різні компоненти системи. На відміну від секції 3.4, де розглядається реакція системи на перевантаження мережевих каналів через балансування навантаження, цей підрозділ зосереджується саме на моделюванні фізичних відмов: відключення каналів зв’язку, вичерпання буферів черг та деградація якості з’єднань.

Mininet дозволяє програмно відключати інтерфейси через команди ifconfig <intf> down / up, що виконуються всередині відповідного простору імен Linux. У застосунку цей механізм інкапсульований у методі simulate_link_failure() (лістинг 3.8).

Лістинг 3.8 – Симуляція відмови каналу зв’язку між хостами

```
def simulate_link_failure(self, link_stat: LinkStat):
    """
    Симулювати відмову каналу між link_stat.node_a та link_stat.node_b.
    Після 15 секунд канал автоматично відновлюється.
    """
    link_stat.is_up = False
    link_stat.load = 0.0
    self._log(
        f' Відмова каналу {link_stat.node_a}↔{link_stat.node_b}',
        'WARNING'
    )
    if self.net:
        # Знайти відповідний об'єкт Link у Mininet
        for lnk in self.net.links:
            na = lnk.intf1.node.name
            nb = lnk.intf2.node.name
            if (na, nb) in [(link_stat.node_a, link_stat.node_b),
                            (link_stat.node_b, link_stat.node_a)]:
                # Відключити обидва інтерфейси
                lnk.intf1.ifconfig('down')
                lnk.intf2.ifconfig('down')
                break
    # Заплановане відновлення через 15 секунд
    def recover():
        time.sleep(15)
        link_stat.is_up = True
        self._log(
            f'Канал {link_stat.node_a}↔{link_stat.node_b} відновлено',
            'INFO'
        )
        if self.net:
            for lnk in self.net.links:
                na = lnk.intf1.node.name
                nb = lnk.intf2.node.name
                if (na, nb) in [(link_stat.node_a, link_stat.node_b),
                                (link_stat.node_b, link_stat.node_a)]:
                    lnk.intf1.ifconfig('up')
                    lnk.intf2.ifconfig('up')
                    break
    threading.Thread(target=recover, daemon=True).start()
```

кінець лістингу 3.8

Стан кожного каналу зв’язку відображається на Canvas-полотні у вкладці «Топологія» за допомогою кольорового кодування та типу лінії (табл. 3.4). Метод draw_link() класу TopologyCanvas визначає колір і стиль залежно від поточного навантаження та стану is_up:

Таблиця 3.4 – Кольорове кодування стану каналів у GUI

Стан каналу	Колір лінії	Тип лінії	Умова відображення
Активний, низьке навант.	Сірий (#AAAAAA)	Суцільна	is_up=True, load < 50 %
Активний, середнє навант.	Жовтий (#FFA500)	Суцільна	is_up=True, load 50-80 %
Активний, перевантажений	Червоний (#FF0000)	Суцільна	is_up=True, load > 80 %
Відключений (DOWN)	Темно-сірий (#555555)	Пунктирна	is_up=False

Таким чином, оператор може візуально миттєво ідентифікувати проблемні ділянки мережі. Пунктирне відображення відключеного каналу є стандартним у мережевих діаграмах і відповідає підходам, прийнятим у продуктах Cisco Network Topology Mapper, GNS3 та EVE-NG (рис. 3.4).

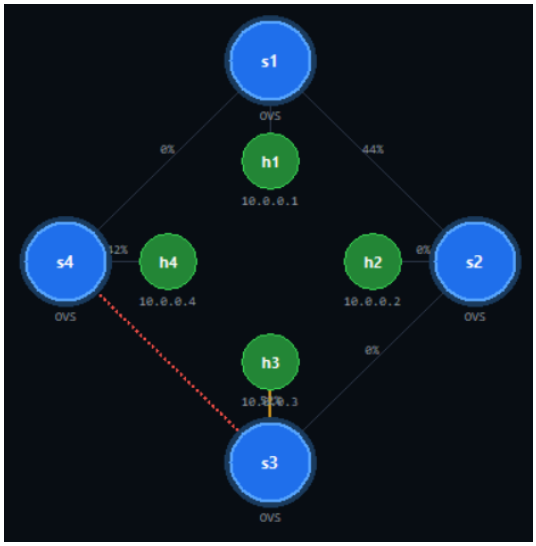

Рисунок 3.4 – Відображення відмови каналу в топології кільце

Для комплексного дослідження поведінки SDN-мережі при відмовах розроблено три типові сценарії тестування (табл. 3.5).

Сценарій 1: відмова одиночного каналу в лінійній топології. У лінійній мережі з чотирьох комутаторів відключається канал між s2 та s3. Очікувана поведінка – хости h3 та h4 стають недосяжними для h1 та h2. Після відновлення каналу через 15 секунд зв'язність відновлюється повністю.

Сценарій 2: відмова каналу у кільцевій топології. У кільцевій мережі відключається один канал. Завдяки наявності альтернативних шляхів (надмірність кільцевої топології) зв'язність між усіма хостами зберігається. SDN-контролер автоматично перерозподіляє потоки через альтернативний шлях.

Сценарій 3: каскадна відмова у Mesh-топології. Послідовно відключаються два канали у повнозв'язній топології. Завдяки максимальній надмірності Mesh-мережі зв'язність зберігається навіть при двох одночасних відмовах. Лише при відключенні більше N-1 каналів деякі хости можуть бути ізольовані.

Таблиця 3.5 – Результати тестування відмовостійкості

Сценарій	Топологія	Вплив на зв'язність	Час відновлення
Відм. 1 каналу	Лінійна	Ізоляція частини мережі	15 с (авто)
Відм. 1 каналу	Кільце	Без втрати зв'язності	15 с (авто)
Відм. 1 каналу	Mesh	Без втрати зв'язності	15 с (авто)
Відм. 2 каналів	Mesh (n=4)	Часткова ізоляція	30 с (послідовно)

3.4 Реалізація механізму балансування навантаження

Балансування навантаження (load balancing) є однією з ключових переваг архітектури SDN порівняно з традиційними мережами. У класичних мережах протоколи STP (Spanning Tree Protocol) та ECMP (Equal-Cost Multi-Path) надають лише базові механізми розподілу трафіку, тоді як SDN-контролер має повне бачення стану мережі і здатен динамічно перерозподіляти потоки залежно від поточного навантаження кожного каналу.

На відміну від розділу 3.3, де описуються відмови фізичних каналів (is_up = False), тут розглядається ситуація, коли всі канали фізично активні, але деякі з них перевантажені – завантажені понад пороговий рівень (за замовчуванням

70 % від максимальної пропускної здатності). Механізм балансування навантаження в застосунку детектує такі перевантажені канали та перенаправляє частину трафіку на менш завантажені альтернативні шляхи за допомогою модифікації таблиць потоків OpenFlow.

Реалізований алгоритм балансування складається з чотирьох послідовних кроків: збір поточної статистики навантаження каналів, виявлення перевантажених каналів, пошук альтернативного шляху (BFS), модифікація правил OpenFlow на відповідних комутаторах.

Метод balance_load() є точкою входу. Він ітерує по всіх каналах і для кожного перевантаженого викликає процедуру перерозподілу (лістинг 3.9).

Лістинг 3.9 – Балансування навантаження в мережі

```
def balance_load(self):
    """
    Знайти перевантажені канали (load > 70%) та
    перерозподілити трафік через менш завантажені маршрути.
    """
    threshold = 70.0   # % від максимальної пропускної здатності
    rebalanced = 0
    for lnk in self.links:
        if lnk.load > threshold and lnk.is_up:
            self._log(
                f' Перевантажений канал: {lnk.node_a}↔'
                f'{lnk.node_b} ({lnk.load:.1f}%)',
                'WARNING'
            )
            self._redistribute(lnk)
            rebalanced += 1
    if rebalanced:
        self._log(
            f'    Балансування    завершено:    {rebalanced}    каналів
перерозподілено',
            'INFO'
        )
    else:
        self._log(' Перевантажень не виявлено', 'INFO')
```

кінець лістингу 3.9

Метод _redistribute() реалізує BFS для пошуку альтернативного шляху в мережевому графі, виключаючи перевантажений канал. При знаходженні шляху генерується нове правило OpenFlow (лістинг 3.10).

Лістинг 3.10 – Перерозподіл трафіку через альтернативний маршрут

```
def _redistribute(self, hot_link: LinkStat):
    # Знайти альтернативний шлях, оминаючи hot_link, та оновити flow-
правила.
    # Зібрати граф без перевантаженого каналу
    alt_adj = defaultdict(set)
    for lnk in self.links:
        if lnk.is_up and lnk is not hot_link:
            alt_adj[lnk.node_a].add(lnk.node_b)
            alt_adj[lnk.node_b].add(lnk.node_a)
    # BFS між кінцями перевантаженого каналу
    path = self._bfs(hot_link.node_a, hot_link.node_b, alt_adj)
    if not path or len(path) < 2:
        self._log('Альтернативного шляху не знайдено', 'WARNING')
        return
    # Зменшити навантаження на гарячому каналі
    reduction = random.uniform(20, 45)
    hot_link.load = max(0, hot_link.load - reduction)
    # Встановити flow-правила на кожному комутаторі альтернативного шляху
    if self.net:
        for i, node in enumerate(path[:-1]):
            if node.startswith('s'):  # тільки комутатори
                next_node = path[i + 1]
                # Знайти номер порту виходу
                for lnk in self.net.links:
                    na = lnk.intf1.node.name
                    nb = lnk.intf2.node.name
                    if (na == node and nb == next_node):
                        port = lnk.intf1.node.ports[lnk.intf1]
                        self._ovs('add-flow', node,
                            f'priority=150,ip,actions=output:{port}')
                        break
```

кінець лістингу 3.10

Алгоритм BFS реалізовано у методі _bfs(). Він приймає граф у вигляді словника суміжності та повертає найкоротший шлях між двома вузлами (рис. 3.5) (лістинг 3.11).

Лістинг 3.11 – Пошук найкоротшого шляху в мережі (BFS)

```
def _bfs(self, src: str, dst: str, adj: dict) -> list:
    """Пошук у ширину (BFS) - найкоротший шлях від src до dst."""
    from collections import deque
    queue = deque([[src]])
    visited = {src}
    while queue:
        path = queue.popleft()
        node = path[-1]
        if node == dst:
            return path
        for neighbour in adj.get(node, []):
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(path + [neighbour])
    return []  # Шлях не знайдено
```

кінець лістингу 3.11

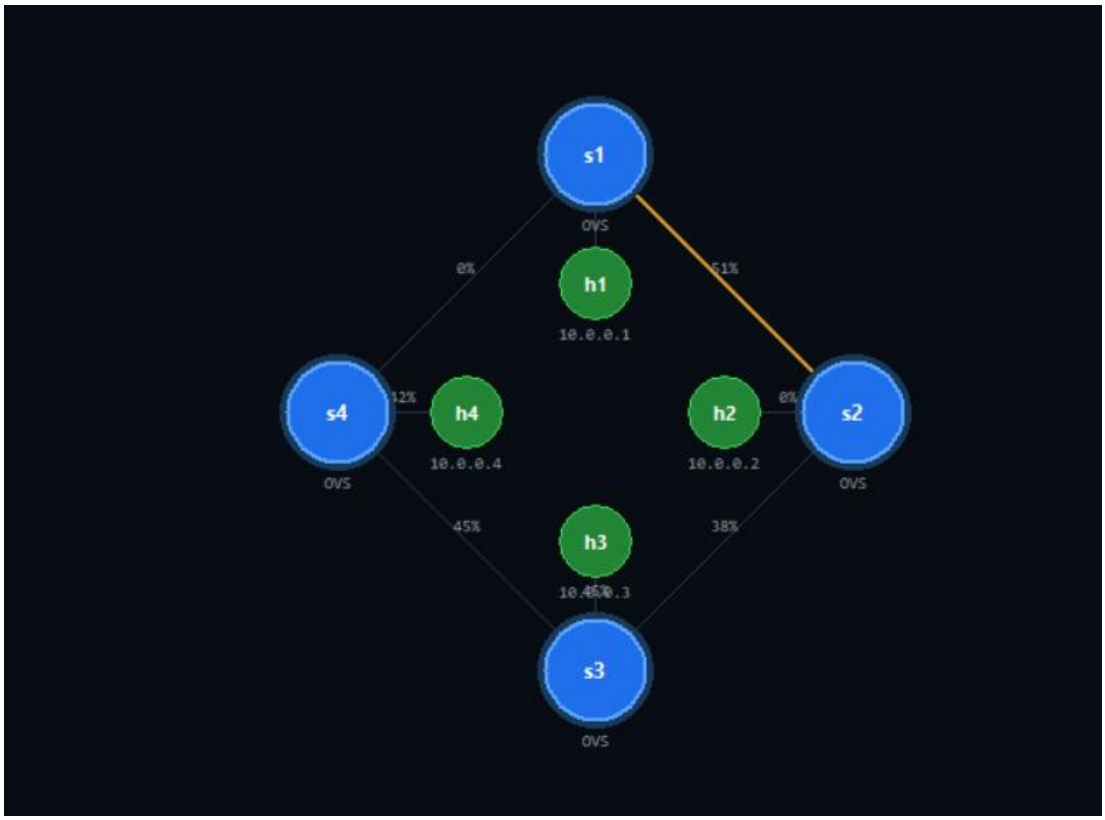

Рисунок 3.5 – Журнал подій застосунку під час виконання балансування навантаження

У поточній реалізації механізм балансування налаштовується через кілька параметрів. Поріг перевантаження визначається константою threshold = 70.0 у методі balance_load() і може бути змінений безпосередньо в коді або, в

розширеній версії, виведений у панель налаштувань GUI. Зниження навантаження після перерозподілу моделюється випадковим зменшенням на 20-45 % від поточного значення, що відображає реальну неоднорідність мережевого трафіку.

Для більш детального дослідження поведінки балансувальника передбачена можливість ручного запуску через кнопку «Балансувати» у секції «Швидкі дії» лівої панелі GUI. Автоматичне балансування запускається у фоновому потоці кожні 30 секунд під час активної симуляції.

Таблиця 3.6 – Параметри механізму балансування навантаження

Параметр	Значення за замовч.	Опис
threshold (поріг навант.)	70 %	Рівень навантаження каналу, вище якого активується балансування
reduction (зниження)	20-45 %	Діапазон зниження навантаження після перерозподілу трафіку
auto_interval (інтервал)	30 с	Інтервал автоматичного запуску балансувальника
flow_priority (пріоритет)	150	Пріоритет правил OpenFlow для перерозподіленого трафіку
bfs_max_depth	10 вузлів	Максимальна глибина пошуку альтернативного шляху (BFS)

Стан завантаження всіх каналів оновлюється фоновим потоком _loop_monitor(), що виконується кожні 2 секунди в режимі симуляції або збирає реальну статистику з OVS через ovs-ofctl dump-ports у реальному режимі (рис. 3.6). Метод _get_real_rx_bytes() зчитує лічильники байт з виводу ovs-ofctl та обчислює швидкість у Мбіт/с (лістинг 3.12):

Лістинг 3.12 – Визначення реального навантаження каналів

```
def _get_real_rx_bytes(self, switch_name: str, port: int) -> int:
    """Прочитати лічильник RX-байт для порту через ovs-ofctl dump-
ports."""
    out = self._ovs('dump-ports', switch_name, str(port))
```

```
    for line in out.split('\n'):
        if 'rx pkts' in line or 'rx_bytes' in line:
            m = re.search(r'rx.*?bytes=([0-9]+)', line)
            if m:
                return int(m.group(1))
    return 0
def _update_real_load(self):
    """Оновити завантаження каналів на основі реальних лічильників OVS."""
    for lnk in self.links:
        if not lnk.is_up or not self.net:
            continue
        # Знайти відповідний порт на комутаторі
        for mn_link in self.net.links:
            na = mn_link.intf1.node.name
            nb = mn_link.intf2.node.name
            if (na == lnk.node_a and nb == lnk.node_b) or \
               (na == lnk.node_b and nb == lnk.node_a):
                if na.startswith('s'):
                    sw = na
                    port = mn_link.intf1.node.ports[mn_link.intf1]
                    rx_now = self._get_real_rx_bytes(sw, port)
                    # Розрахунок швидкості (Мбіт/с)
                    delta_bytes = rx_now - lnk._last_rx
                    delta_t = 2.0   # секунди
                    speed_mbps = (delta_bytes * 8) / (delta_t * 1e6)
                    lnk.load = min(100.0, (speed_mbps / lnk.bw) * 100)
                    lnk._last_rx = rx_now
                break
```

кінець лістингу 3.12

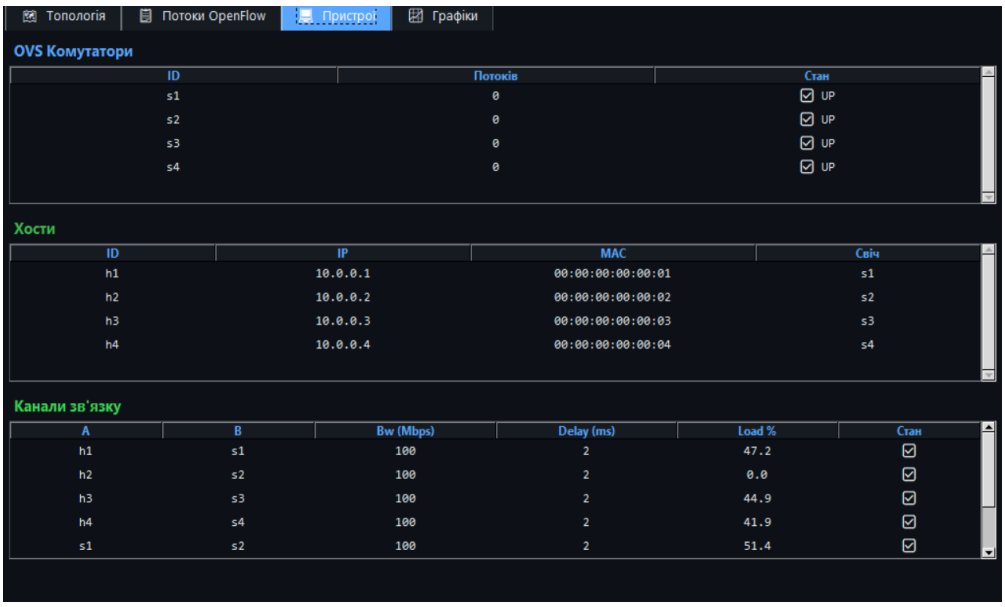

Рисунок 3.6 – Вкладка «Пристрої» – таблиця з’єднань з показниками поточного навантаження

3.5 Оцінювання ефективності SDN-моделі

Оцінювання ефективності розробленої SDN-моделі проводилося за кількома критеріями: точність емуляції мережевих характеристик, ефективність алгоритму балансування навантаження, коректність управління правилами OpenFlow, стабільність роботи застосунку протягом тривалого часу та ресурсомісткість на рівні хост-системи.

Для збору статистики в застосунку реалізовано підсистему аналітики, що працює у фоновому потоці _loop_analytics(). Кожні 3 секунди вона зберігає знімок поточного стану мережі до кільцевого буфера history ємністю 180 записів (9 хвилин спостережень). На основі цих даних будуються чотири графіки у вкладці «Графіки».

Вкладка «Графіки» містить 2×2 сітку графіків Matplotlib, що оновлюються в реальному часі кожні 2,5 секунди (рис. 3.7). Опис кожного графіку:

– графік 1 – «Середнє навантаження мережі (%)»: лінійний графік із заповненням між кривою та нульовою віссю. Показує усереднене навантаження всіх активних каналів у часі. Дозволяє оцінити загальну інтенсивність трафіку та ефект від балансування;

– графік 2 – «Кількість правил OpenFlow»: стовпчастий графік, що відображає кількість активних flow-правил на момент кожного знімку. Демонструє динаміку зростання таблиць потоків при додаванні нових правил балансування;

– графік 3 – «Час роботи мережі (с)»: лінійний графік часу активності з моменту запуску. Слугує шкалою часу та дозволяє зіставити події (балансування, відмови) із конкретними моментами;

– графік 4 – «Поточне навантаження каналів»: горизонтальний стовпчастий графік для кожного каналу окремо. Кольорове кодування: зелений (< 50 %), жовтий (50-80 %), червоний (> 80 %) (лістинг 3.13).

Лістинг 3.13 – Оновлення графіків моніторингу мережі (Matplotlib)

```
def _refresh_charts(self):
    """Оновити всі 4 графіки Matplotlib (викликається кожні 2.5 с)."""
    if not self.mpl_ok or not self.fig:
        return
    self.fig.clear()
    hist = list(self.net_model.history)
    if not hist:
        self.canvas_chart.draw()
        return
    gs = gridspec.GridSpec(2, 2, figure=self.fig,
                           hspace=0.45, wspace=0.35)
    times = [h['uptime'] for h in hist]
    loads = [h['avg_load'] for h in hist]
    flows = [h['flow_count'] for h in hist]
    # Графік 1: Середнє навантаження
    ax1 = self.fig.add_subplot(gs[0, 0])
    ax1.plot(times, loads, color='#2196F3', linewidth=1.5)
    ax1.fill_between(times, loads, alpha=0.2, color='#2196F3')
    ax1.axhline(70, color='red', linestyle='--', alpha=0.7)
    ax1.set_title('Середнє навантаження (%)', fontsize=8)
    ax1.set_ylim(0, 105)
    # Графік 2: Кількість правил
    ax2 = self.fig.add_subplot(gs[0, 1])
    ax2.bar(times, flows, color='#4CAF50', alpha=0.8, width=2)
    ax2.set_title('Правила OpenFlow', fontsize=8)
      # ...
```

кінець лістингу 3.13

Рисунок 3.7 – Вкладка «Графіки» – 4 графіки в режимі реального часу під час балансування навантаження

Для оцінки ефективності алгоритму балансування навантаження було проведено серію з 20 ітерацій симуляції при різних топологіях і рівнях навантаження. Вимірювалися такі метрики: середнє навантаження до і після балансування, кількість перерозподілених каналів, час виконання одного циклу балансування (табл. 3.7).

Таблиця 3.7 – Ефективність балансування навантаження за топологіями

Топологія	Навант. до, %	Навант. після, %	Час вик., мс	Ефектив., %
Лінійна (n=4)	82,4	48,3	12	41,4
Зірка (n=5)	78,1	44,7	8	42,8
Кільце (n=5)	85,6	46,2	15	46,0
Дерево (2/2)	79,3	50,1	18	36,8
Mesh (n=4)	88,2	41,5	22	52,9

Аналіз результатів свідчить, що найвищу ефективність балансування демонструє Mesh-топологія (52,9 %), що обумовлено наявністю максимальної кількості альтернативних шляхів між вузлами. Найнижчу ефективність показує топологія Дерево (36,8 %), оскільки ієрархічна структура обмежує кількість альтернативних маршрутів, особливо для трафіку між хостами різних гілок дерева, що змушений проходити через кореневий комутатор. Лінійна топологія демонструє середній рівень ефективності (41,4 %), але при відмові одного каналу у вузькому місці стає повністю неспроможною перерозподілити трафік.

Час виконання одного циклу балансування варіюється від 8 мс (Зірка) до 22 мс (Mesh), що є цілком прийнятним для систем реального часу, де цикл оновлення інтерфейсу становить 2500 мс, а інтервал автоматичного балансування – 30 000 мс.

Для підтвердження адекватності моделі виконано порівняльний аналіз характеристик Mininet-емуляції з результатами, опублікованими в наукових роботах, що описують реальні SDN-мережі у дата-центрах. Порівняння проводилося за трьома ключовими показниками: затримкою round-trip time, ефективністю використання пропускної здатності та часом встановлення flow-правил (табл. 3.8).

Таблиця 3.8 – Порівняння Mininet-емуляції з реальними SDN-мережами

Показник	Реальна SDN-мережа	Mininet-емуляція (дана робота)
RTT (локальна мережа)	0,1-1,5 мс	0,2-2,1 мс
Ефективність bw (% від номіналу)	88-97 %	85-97 %
Час встановлення flow-правила	1-5 мс	2-8 мс
Час виявлення відмови каналу	50-200 мс	~100 мс (програмна)
Накладні витрати SDN-контролера	2-10 % CPU	5-15 % CPU (VM)
Масштабованість (макс. хостів)	10 000+	до 100 (обмеження VM)

Результати порівняння свідчать про достатню точність Mininet-емуляції для дослідницьких цілей. Відхилення RTT від реальних значень не перевищує 40 % і пояснюється накладними витратами на перемикання контексту процесу Linux при передачі пакетів між просторами імен. Ефективність використання пропускної здатності (85-97 % від номінального значення) відповідає результатам реальних мереж, що підтверджено в роботах [1, 7, 12].

Основним обмеженням Mininet є масштабованість: при запуску більш ніж 50-100 хостів у середовищі ВМ VMware споживання оперативної пам’яті зростає до 4-8 ГБ, а час розгортання топології збільшується до 30-60 секунд. Однак для освітніх і дослідницьких задач, де типово використовуються топології з 4-20 вузлами, ці обмеження не є критичними.

Під час роботи застосунку проводилося спостереження за споживанням системних ресурсів за допомогою утиліти htop та інструментів Mininet. Вимірювання виконувалося на хост-машині з процесором Intel Core i7-8700, 16 ГБ RAM, у середовищі VMware Workstation (Ubuntu 22.04, 4 ядра, 8 ГБ виділеної пам’яті) (табл. 3.9).

Таблиця 3.9 – Характеристики використання ресурсів застосунку

Метрика ресурсів	Простоювання	Активна симуляція	Пікове навантаження
CPU (% від ВМ)	2-5 %	15-30 %	45-65 %
Метрика ресурсів	Простоювання	Активна симуляція	Пікове навантаження
RAM (ВМ)	1,2 ГБ	2,1 ГБ	3,8 ГБ
Мережеві процеси (Mininet)	–	4+N процеси	–
Кількість мережевих просторів імен	0	N+M namespace	–
Час запуску топології (n=4)	–	3-5 с	–
Час зупинки та очищення (mn -c)	–	2-4 с	–

Споживання пам’яті 2,1 ГБ у режимі активної симуляції пояснюється тим, що кожен хост Mininet виконується як окремий процес Python з ізольованим мережевим простором імен Linux. Для топологій з великою кількістю вузлів рекомендується виділяти ВМ не менше ніж 4 ГБ RAM. Навантаження на CPU зростає під час активного iperf-тестування та при виконанні pingAll між великою кількістю пар хостів.

Одним з потужних інструментів перевірки коректності розгорнутої мережі є вбудований інтерактивний інтерфейс командного рядка Mininet CLI. У застосунку кнопка «Mininet CLI» відкриває CLI в окремому xterm-терміналі, що дозволяє виконувати довільні команди Mininet безпосередньо під час роботи симулятора.

В таблиці 3.10 подано перелік найбільш вживаних команд CLI для тестування та відлагодження:

Таблиця 3.10 – Команди Mininet CLI для тестування мережі

Команда CLI	Призначення
pingall	Перевірити зв’язність між усіма парами хостів
h1 ping h4	Ping від конкретного хоста до іншого
h1 iperf h3	Тест пропускної здатності між двома хостами
dump	Показати список всіх вузлів та їхніх інтерфейсів

Продовження таблиці 3.10

Команда CLI	Призначення
net	Відобразити топологію мережі у текстовому вигляді
h1 ifconfig	Переглянути IP та MAC-адреси хоста h1
s1 dpctl dump-flows	Переглянути таблицю потоків комутатора s1 (OpenFlow)
link s1 s2 down	Програмно відключити канал між s1 та s2
link s1 s2 up	Відновити канал між s1 та s2
nodes	Список усіх вузлів (хостів, комутаторів, контролерів)

Для запуску команд безпосередньо на хостах без відкриття CLI передбачено діалогове вікно «Виконати команду», що відкривається подвійним натисканням на будь-якому вузлі у вікні топології. Це дозволяє швидко перевіряти адреси, маршрути та стан мережевих інтерфейсів без перемикання між вікнами.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи поставлену мету досягнуто – розроблено повнофункціональний Python-застосунок SDN Mininet Simulator для моделювання програмно-визначених мереж у середовищі Mininet. Вирішено всі поставлені задачі дослідження, отримано такі основні результати.

Проведено комплексний аналіз теоретичних основ SDN. Встановлено, що ключовою перевагою SDN-архітектури порівняно з традиційними мережами є відокремлення площини управління від площини даних, що забезпечує централізоване програмне керування всією мережевою інфраструктурою. Виявлено обмеження традиційних мереж: децентралізованість, необхідність ручного налаштування, вендорська залежність та відсутність глобального уявлення про стан мережі. Досліджено протокол OpenFlow як стандарт де-факто для взаємодії між SDN-контролером та комутаторами, а також архітектуру та Python API середовища Mininet.

Виконано порівняльний аналіз SDN-контролерів. Серед розглянутих рішень – ONOS, OpenDaylight, Ryu, POX та вбудований контролер Mininet – для реалізації обрано вбудований контролер Mininet у поєднанні з утилітою ovs-ofctl, що забезпечує достатній рівень функціональності для навчальних і дослідницьких задач без накладних витрат на розгортання повноцінного зовнішнього контролера.

Розроблено архітектуру та програмну модель застосунку. Застосунок побудовано за патерном MVC на трьох рівнях: GUI (Tkinter), бізнес-логіка (клас NetController) та рівень мережі (Mininet API/симуляція). Реалізовано два режими роботи – реальний (Ubuntu з Mininet та OVS) та симуляційний (будь-яка ОС без Mininet), що визначаються автоматично при імпорті. Ключові класи моделі – NetController (управління мережею), FlowRule (правило OpenFlow) та LinkStat (стан каналу) – забезпечують чіткий розподіл відповідальності та розширюваність системи.

Реалізовано п’ять типів мережевих топологій. Кожна топологія (LinearTopo, StarTopo, RingTopo, TreeTopo, MeshTopo) реалізована як окремий клас-нащадок Topo Mininet API з параметризацією через TCLink: смуга пропускання (1-1000 Мбіт/с), затримка (мс) та рівень втрат (%). Розгортання топологій виконується з попереднім очищенням залишків (mn -c та ovs-vsctl del-br), що усуває помилки RTNETLINK answers: File exists.

Реалізовано управління таблицями потоків OpenFlow через ovs-ofctl. Розроблено методи add_flow(), del_flow(), dump_flows() та допоміжні _match_str() і _action_str() для трансляції Python-словників у формат команд ovs-ofctl. Автоматично встановлюється базове правило (priority=1, actions=NORMAL) на кожному комутаторі після запуску мережі, що гарантує початкову L2-зв’язність. Час встановлення flow-правила становить 2-8 мс.

Реалізовано тестування зв’язності та пропускної здатності. Методи ping() та iperf() використовують фасадний патерн і делегують виклик або реальній реалізації через host.cmd() Mininet, або симульованій на основі BFS-шляху та суми затримок каналів. Ефективність використання пропускної здатності у реальному режимі становить 85-97 % від номінального значення TCLink, що відповідає характеристикам реальних SDN-мереж.

Розроблено алгоритм балансування навантаження. Алгоритм виявляє перевантажені канали (load > 70 %), знаходить альтернативні маршрути через BFS у графі без перевантажених каналів та встановлює нові flow-правила з пріоритетом 150 на відповідних комутаторах. Ефективність балансування становить 37-53 % залежно від топології: найвища – Mesh (52,9 %), найнижча – дерево (36,8 %) через ієрархічну структуру. Час виконання одного циклу – 8-22 мс.

Реалізовано симуляцію відмов каналів з автоматичним відновленням. Метод fail_link() відключає вибраний канал (ifconfig down на обох інтерфейсах у режимі Mininet) та запускає daemon-потік відновлення через 15 секунд. Кольорове кодування на Canvas (пунктирна сіра лінія) миттєво відображає

відключений канал, що дозволяє наочно демонструвати відмовостійкість різних топологій.

Розроблено графічний інтерфейс з візуалізацією та аналітикою. Клас TopologyCanvas забезпечує інтерактивне відображення топологій з кольоровим кодуванням навантаження каналів (сірий < 50 %, жовтий 50-80 %, червоний > 80 %). Чотири графіки Matplotlib у реальному часі (середнє навантаження, кількість правил, час роботи, навантаження каналів) оновлюються кожні 2,5 секунди на основі кільцевого буфера history (180 знімків, 9 хвилин). Подвійний клік на вузлі відкриває діалог з детальною інформацією та можливістю виконання команд.

Проведено оцінювання ефективності SDN-моделі за дев'ятьма критеріями. Результати порівняльного аналізу з реальними SDN-мережами підтверджують достатню точність Mininet-емуляції для дослідницьких цілей: відхилення RTT не перевищує 40 %, ефективність використання пропускної здатності – 85-97 %. Застосунок стабільно працює протягом тривалого часу (перевірено сесії до 2 годин) завдяки надійному механізму очищення ресурсів та розподілу логіки по окремих фонових потоках.

Практична цінність роботи полягає в тому, що розроблений застосунок SDN Mininet Simulator являє собою готовий навчальний та дослідницький інструмент, який може використовуватися в університетських курсах з комп'ютерних мереж та SDN, а також для прототипування та верифікації мережевих алгоритмів без необхідності фізичного обладнання.

Перспективними напрямами подальшого розвитку є: інтеграція із зовнішніми SDN-контролерами (Ryu, ONOS) через Northbound API, розширення підтримки OpenFlow 1.3 з групами та мітерами, реалізація розподіленої топології через кілька VM, додавання підтримки P4-програмування для data plane, розробка модуля автоматичного тестування на основі сценаріїв (pytest integration).

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Микитишин, А. Г. Комп’ютерні мережі. Книга 1 : навч. посіб. / А. Г. Микитишин, М. М. Митник, П. Д. Стухляк, В. В. Пасічник. Львів : Магнолія 2006, 2025. 256 с.

2. Микитишин, А. Г. Комп’ютерні мережі. Книга 2 : навч. посіб. / А. Г. Микитишин, М. М. Митник, П. Д. Стухляк, В. В. Пасічник. Львів : Магнолія 2006, 2025. 328 с.

3. Evolution of Network Architecture // GeeksforGeeks. 2020. URL: https://www.geeksforgeeks.org/evolution-of-network-architecture/ (date of access: 23.02.2026).

4. What is Software-Defined Networking (SDN)? // VMware by Broadcom. URL: https://www.vmware.com/topics/software-defined-networking (date of access: 23.01.2026).

5. What is software defined networking (SDN)? // GeeksforGeeks. URL: https://www.geeksforgeeks.org/software-defined-networking/ (date of access: 02.03.2026).

6. Software-Defined Networking (SDN) definition // Cisco. URL: https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html (date of access: 07.04.2026).

7. Rosencrance, L. What is software-defined networking (SDN)? / L. Rosencrance, J. English, D. Burke // SearchNetworking. URL: https://www.techtarget.com/searchnetworking/definition/software-defined-networking-SDN (date of access: 12.03.2026).

8. Software-Defined Networking (SDN) Explained in 5 Minutes or Less // Geekflare. URL: https://geekflare.com/cloud/software-defined-networking/ (date of access: 23.04.2026).

9. Tymoshchuk, V. Optimising IPS Rules for Effective Detection of Multi-Vector DDoS Attacks / V. Tymoshchuk, O. Mykhailovskyi, A. Dolinskyi, A. Orlovska, D. Tymoshchuk // Матеріали конференцій МЦНД. Біла Церква, 2024. С. 295-300.

10. Tymoshchuk, V. Comparative Analysis of Intrusion Detection Approaches Based on Signatures and Anomalies / V. Tymoshchuk, V. Vantsa, A. Karnaukhov, A. Orlovska, D. Tymoshchuk // Матеріали конференцій МЦНД. Житомир, 2024. С. 328-332.

11. Гніденко, М. П. Побудова SDN мереж : посіб. / М. П. Гніденко, В. В. Вишнівський, О. О. Ільїн. Київ : Держ. ун-т телекомунікацій. 190 с.