НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
Інформаційних технологій (назва факультету(ННІ))	Комп'ютерних систем, мереж та кібербезпеки (назва кафедри)
_____ Болбот І.М., д.т.н, проф. (підпис) (ПІБ, вчене звання і ступінь)	_____ Касаткін Д.Ю., к. пед.н., доц. (підпис) (ПІБ, вчене звання і ступінь)
«___»_____2025 р.	«___»_____2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Застосування SDN у системах Інтернету речей»

Спеціальність 123 «Комп’ютерна інженерія»
(код і найменування)

Освітня програма Комп’ютерні системи і мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

д.т.н., доцент	_____	Шкарупило В.В.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Керівник магістерської кваліфікаційної роботи

д.т.н., доцент	_____	Коваленко О.Є.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Виконав _____ Гурдуяла Р.Є.
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
к.пед.н., доц. _____ Касаткін Д.Ю.
(вчене звання і ступінь) (підпис) (ПІБ)
«__» _____ 20__ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Гурдуялі Руслану Євгенійовичу
(прізвище, ім'я, по батькові)

Спеціальність 123 «Комп'ютерна інженерія»
(код і найменування)

Освітня програма Комп'ютерні системи та мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи: «Застосування SDN у системах Інтернету речей»
затверджена наказом ректора НУБіП України від "29" жовтня 2024р. № 1941 «С»

Термін подання завершеної роботи на кафедру 14 листопада 2025 р.

Вихідні дані до магістерської кваліфікаційної роботи середовище розробки – Python,P4, тип системи – мережа, SDN, гіпервізор - VMware, операційна система - Ubuntu, ONOS, тип свічів – bmv2, OpenVswitch

Перелік питань, що підлягають дослідженню:
1. Аналіз предметної області
2. Встановлення та налаштування системи
3. Тестування побудованої системи

Перелік графічного матеріалу (за потреби) Топологія побудованої мережі в Miniedit

Дата видачі завдання " 29 **"** жовтня 2024 р.

Керівник магістерської кваліфікаційної роботи _____ Коваленко О.Є.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Гурдуяла Р.Є. .
(підпис) (прізвище та ініціали)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз предметної області	29.10.2024 р - 15.12.2025 р	Виконано
2	Проектування системи	12.02.2025 р - 15.03.2025 р	Виконано
3	Реалізація системи	15.03.2025 р - 20.08.2025 р	Виконано
4	Тестування системи	21.08.2025 р - 22.10.2025 р	Виконано
5	Оформлення пояснювальної записки	22.10.2025 р - 10.11.2025 р	Виконано
6	Оформлення графічного матеріалу	10.11.2025 р - 14.11.2025 р	Виконано

Студент _____ **Гурдуяла Р. Є.**

(підпис) (ініціали та прізвище)

Керівник проекту (роботи) _____ **О.Є. Коваленко**

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 53 сторінки, 26 рисунків, 3 таблиці, 4 лістингів, 2 додатки, 10 джерел.

Об’єкт аналізу – SDN мережа для роботи IoT.

Мета роботи – розроблення відмовостійкої мережі для роботи в системах інтернету речей.

Проект складається з трьох розділів.

У першому розділі розглядаємо, поставлені задачі, огляд існуючих компонентів систем та можливості подальшого розвитку проекту.

Другий розділ присвячено встановленню програмних засобів та налаштуванню системи.

У третьому розділі результати тестування системи та огляд результатів.

У результаті було розроблено віртуальну комп’ютерну мережу SDN для роботи в системах інтернету речей.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ОС	–	Операційна система
GUI	–	Graphical User Interface
ПЗ	–	Програмне забезпечення
SDN	–	Software-Defined Networking
ВМ	–	Віртуальна машина
BMv2	–	Behavioral model switch
IoT	–	Internet of things

ВСТУП

Сучасна епоха характеризується поєднанням двох принципово різних понять: «Інтернет» і «Речі». Незважаючи на великі відмінності, їм вдалося співіснувати в мережі, яка отримала назву Інтернет речей (IoT).

IoT — це концепція, при якій пристрої, які мають власну ОС, датчики і ПЗ. Ці пристрої можуть збирати, обмінюватися даними через мережу Інтернет. Наприклад, маємо вдома IP-камеру з датчиком руху та телефон з додатком цієї камери. Камера може відправляти нам фото чи відео при спрацюванні датчику руху. І таких прикладів можливо навести безліч, від кавоварки до освітлення кімнати. Тепер крім того, що люди спілкуються з машинами та один з одним, IoT дає змогу машинам спілкуватися між собою. Машини тепер підключені та взаємодіють одна з одною, незалежно, що призводить до збільшення даних у сітчастій мережі. Враховуючи цей фактор, нам потрібна більш структурована та стабільна мережа, ніж її класичний вид. Тут нам на допомогу можуть прийти мережі SDN.

SDN — це мережа, яка керується програмними засобами. Якщо точніше, уявимо традиційну мережу, яка складається з комутаторів, свічів та точок доступу. Всі вони керуються своїми окремими налаштуваннями як в плані безпеки, так і застосування цих пристроїв. Завдяки SDN ми можемо централізовано проводити управління мережею. Тож SDN — це програмно конфігурована мережа, яка відокремлює рівень управління мережею від рівня передачі даних та керується через своє програмне забезпечення.

Програми IoT можна знайти в усіх сферах життя, від суспільства до промисловості та навколишнього середовища. Завдяки контекстному розпізнаванню, кількість областей застосування постійно зростає. З точки зору технології та варіантів використання, IoT пройшов постійний цикл еволюції та оновлення. У результаті ми спостерігали, як архітектура IoT еволюціонувала від свого традиційного стану до високоефективних, розумних та програмно-визначених систем із віртуалізацією. Мобільні та персоналізовані медичні

послуги на основі IoT можуть бути реалізовані за допомогою персональних комп'ютерних пристроїв і мобільного доступу до Інтернету. Від точного землеробства до виробництва, переробки, зберігання, дистрибуції та споживання харчових продуктів, IoT у ланцюзі постачання харчових продуктів (FSC) може охопити весь процес від ферми до подачі продуктів на стіл. Фізичні об'єкти можна контролювати в режимі реального часу від відправлення і прибуття до пункту призначення, за допомогою Інтернету речей у транспортуванні, підвищуючи ефективність, надійність, безпеку та якість товарів, що доставляються. У сфері видобутку корисних копалин IoT може допомогти працівникам уникнути нещасних випадків. Пожежна організація може виявляти небезпеки в режимі реального часу та запобігати інцидентам, реалізуючи IoT у сценаріях гасіння пожеж. Очікується, що завдяки IoT суспільство зможе краще розвинутися в цих сферах. Крім того вважається, що IoT має позитивний вплив на промислове виробництво, завдяки досягненню більшої автоматизації та моніторингу і контролю. Також вважається, що ця концепція допоможе мінімізувати витрати з точки зору операційних і капітальних витрат.

Інтернет речей зростає з експоненційною швидкістю завдяки технологічному прогресу, що дозволяє все більшій кількості пристроїв або «речей» діяти розумно та підключатися до мережі. Програми, мережі та датчики складають трирівневу архітектуру IoT. Користувачі можуть взаємодіяти з ретельними обчисленнями IoT через прикладний рівень. Мережевий рівень, як випливає з назви, відповідає за мережеві операції та пересилання даних, тоді як сенсорний рівень відповідає за збір даних.

Проте, системи IoT мають свої проблеми, а саме, вони вимагають гнучкої конфігурації та реконфігурації задля забезпечення динамічної продуктивності. Ємність і здатність традиційної архітектури та мережевих підходів не може задовольнити вимоги архітектури IoT. Також безпека має вирішальне значення як на межі мережі, так і на її ядрі. У сучасному світі IoT має бути динамічним, гнучким і достатньо розподіленим, щоб забезпечити

доступність у разі катастрофи. Крім того, зі збільшенням обсягу даних, що генеруються пристроями IoT, потрібні ефективні та інтелектуальні методи розподілу каналів і багаторівнева структура рівня керування, щоб уникнути вузьких місць зв'язку та забезпечити плавну передачу даних. Інтернет речей не має функцій чи можливостей самостійно вирішити ці проблеми або задовольнити вимоги сучасного світу. Таким чином, це породило концепції програмно визначеної мережі (SDN) і віртуалізації мережевих функцій (NFV).

SDN має першорядне значення в революції традиційних мережевих архітектур, надаючи більш гнучкий, масштабований і програмований підхід до керування мережею. Відокремлюючи площину керування від площини даних, SDN забезпечує централізований контроль, дозволяючи адміністраторам динамічно розподіляти мережеві ресурси та ефективно впроваджувати зміни. Значення SDN полягає в його здатності оптимізувати мережеве забезпечення, підвищити масштабованість і спростити мережеве керування, що призводить до підвищення ефективності роботи. SDN знаходить застосування в різних секторах, включаючи центри обробки даних, телекомунікації та корпоративні мережі. У центрах обробки даних SDN полегшує оркестровку ресурсів і забезпечує оптимальний потік трафіку, сприяючи кращій загальній продуктивності. У телекомунікаціях SDN дозволяє створювати гнучкі та програмовані мережі, прокладаючи шлях для таких інновацій, як 5G. Підприємства отримують переваги від SDN, досягаючи більшого контролю над своєю мережевою інфраструктурою, підтримуючи динамічні бізнес-вимоги та підвищуючи безпеку за допомогою централізованого керування політикою. Загалом SDN відіграє ключову роль у перебудові мережевих архітектур, щоб вони відповідали вимогам сучасних, динамічних і інтенсивних додатків.

Щодо віртуалізації ми маємо зрозуміти ідею NFV. Віртуалізація — це процес створення чогось логічного (і віртуального) з фізичного джерела. Метою віртуалізації є відокремлення апаратного забезпечення від програмного забезпечення. NFV — це розподіл доступної смуги пропускання

та ресурсів на канали, які можна використовувати та розподіляти незалежно шляхом створення віртуальних мереж у межах однієї мережі й архітектури в результаті цього явища. Віртуалізація розділяє відокремлені елементи керування мережевого рівня та площину даних горизонтально, відокремлюючи функції мережі від фізичних функцій.

Впровадження SDN в архітектурах IoT допоможе вирішити проблеми з сумісністю, безпекою та надійністю, це в свою чергу зможе допомогти нам створити кращі і розумніші та ефективніші рішення на основі передових технологій, таких як автономне обчислення та ШІ.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постанова індивідуального завдання

Метою роботи є розроблення моделі відмовостійкої IoT-мережі, побудованої на основі технології SDN. Така модель повинна забезпечувати безперервність керування у випадку збою основного контролера та надавати можливість аналізувати стан мережі в реальному часі. Для досягнення цієї мети передбачається створення архітектури з двома контролерами — основним і резервним — а також реалізація програмованої логіки обробки пакетів за допомогою мови P4. Результатом має стати мережа, здатна автоматично адаптувати маршрути відповідно до отриманих телеметричних даних і зберігати працездатність навіть за наявності збоїв у системі керування. Щоб зрозуміти як побудувати таку мережу ми маємо:

- Переглянути вже існуючі рішення для побудови таких систем. Їхні різновиди та направлення, комерційні та відкриті приклади, й обрати найвідповідніший до наших задач.
- Створення системи що буде включати в себе збір телеметрії.
- Налаштування контролера мережі який буде забезпечувати збір телеметрії та прийняття рішень на основі отриманих даних. Також реалізація функцій резервного перекидання функцій з основного контролеру на резервний.
- Створення та налагодження віртуальних свічів які будуть передавати телеметрію на контролер та витримувати високі навантаження трафіком.

Проведення тестування системи в різних сценаріях використання для оцінки її ефективності, надійності та безпеки.

Аналіз результатів тестування, виявлення та усунення можливих недоліків, а також підготовка рекомендацій щодо подальшого вдосконалення

системи, використання технологій машинного навчання, функцію автоматичного резервного копіювання , уникнення втрати пакетів тощо.

1.2 Огляд обраних програмних засобів

1.2.1 Контролер ONOS

ONOS (Open Network Operating System) – це відкрита мережева операційна система для управління SDN (Software Defined Networking) мережами і їх компонентами, такими як комутатори і канали, виконуючи програми або модулі для надання послуг зв'язку кінцевим хостам та сусіднім мережам. Її особливість в тому, що вона поєднує в собі функціонал SDN-контролера, мережевої і серверної ОС. За рахунок такої комбінації інструмент дозволяє стежити за всім, що відбувається в мережах, і спрощує міграцію від традиційної архітектури до SDN.

Рисунок 1.1 – Логотип ОС ONOS

ONOS підтримує як налаштування, так і управління мережею в реальному часі, усуваючи необхідність запуску протоколів управління маршрутизацією і комутацією всередині мережевої структури. Ця система може працювати як розподілена система на декількох серверах, що дозволяє використовувати ресурси ЦП та пам'яті кількох серверів, забезпечуючи відмовостійкість у разі збою сервера і потенційно підтримуючи оновлення обладнання та програмного забезпечення в реальному часі або по черзі без переривання мережевого трафіку. Також ONOS має деякі, аналогічні серверним операційним системам, фунцкії, включаючи API-інтерфейси (Application programming interface), розподіл ресурсів і дозволи, а також CLI (Command-line interface), GUI (Graphical user interface) і системні програми для більшої зручності для користувача. ONOS керує всією мережею, а не одним пристроєм, що може значно спростити управління, налаштування і розгортання нового програмного забезпечення та обладнання.

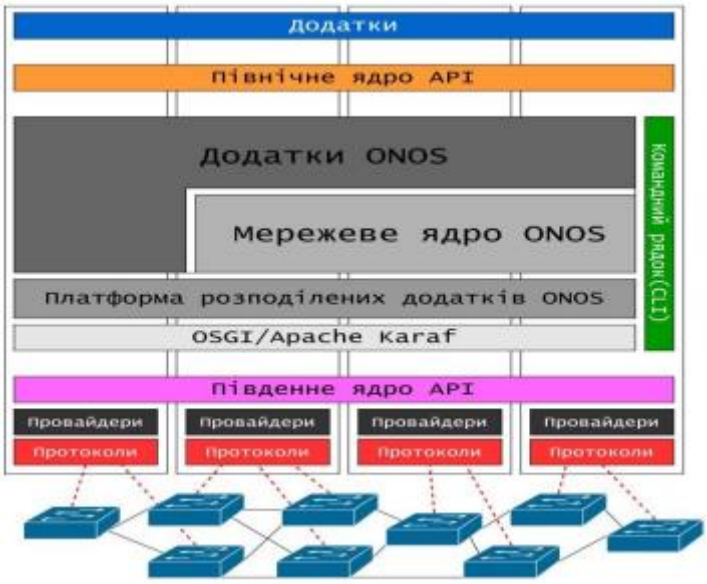


Рисунок 1.2 – Розподілена архітектура ONOS

Ядро ONOS і основні служби, а також додатки ONOS написані на Java у вигляді пакетів, які завантажуються в контейнер Karaf OSGi (Open Services Gateway initiative). Apache Karaf – це контейнер OSGi, що надає інфраструктуру для запуску OSG і пакетів. OSGi – це компонентна система

для Java, яка дозволяє встановлювати і запускати модулі динамічно в одній JVM (Java Virtual Machine). Оскільки ONOS працює в JVM, він може працювати на декількох базових платформах ОС.

Система призначена для роботи у вигляді кластера вузлів, які ідентичні з точки зору їх програмного стека і можуть витримувати відмову окремих елементів, не викликаючи порушень в її здатності контролювати роботу мережі.

Ядро ONOS відповідає за відстеження інформації про мережеве середовище і поширення його застосування або синхронно через запит, або асинхронно через слухаючі зворотні виклики. Ядро також відповідає за збереження стану вибору і синхронізацію стану між вузлами кластера. У ядрі відбувається основний логічний алгоритм. Цей блок забезпечує контроль станів, сповіщень, високу доступність та масштабованість.

1.2.2 ОС Ubuntu desktop

Рисунок 1.3 – Логотип ОС Ubuntu

Ubuntu – це безкоштовна та універсальна система на основі ядра Linux. Створена була як для звичайних користувачів, так як має GUI, так і для досліджень адже працює на ядрі Linux. Вона поєднує стабільність вищезгаданого ядра і відкритість коду, що робить її природним вибором для наукових і технічних проєктів.

Операційна система забезпечує повну сумістність з Mininet, ONOS, P4, bmv2 та іншими фреймворками які ми будемо використовувати в ході

виконання роботи. В цій ОС можна одночасно запускати контролер, свічі й с систему збору телеметрії INT. Це все працює стабільно завдяки прямій підтримці бібліотек gRPC, Python та компіляторів p4c.

Також до плюсів цієї ОС можна зарахувати те що вона розробляється спільнотою з відкритим кодом, що гарантує безпеку, регулярні оновлення та підтримку дослідницьких проектів. Ця система себе зарекомендувала як гнучка ОС яка дає змогу повністю контролювати середовище, змінювати елементи ядра, мережеві модулі та драйвери під власні потреби.

1.2.3 Платформа віртуалізації VMware

Рисунок 1.4 – Логотип платформи віртуалізації VMware

У ролі лабораторного середовища ми будемо використовувати платформу VMware для створення віртуальних машин і запуску ОС в ній. В цьому середовищі ми можемо вільно експериментувати з мережею, контролерами, комутаторами без ризику для основної системи.

Основний принцип роботи платформи віртуалізації в тому що вона створює ізольоване середовище в якому ми маємо окремий комп’ютер з своїм процесором, пам’яттю, жорстким диском і мережею. Це дозволяє одночасно розгортати кілька компонентів SDN інфраструктури, наприклад, одну машину з контролером, іншу з комутатором, ще одну як IoT-вузол.

Для кожної машини ми можемо розподіляти апаратні ресурси, можна задати кількість ядер, обсяг оперативної пам’яті, підключати окремі мережеві адаптери і керувати трафіком між ними. Такі умови дозволяють нам моделювати різні сценарії для перевірки відмовостійкості або балансування навантаження між контролерами.

Також можна відзначити що ПЗ VMware надає можливість швидко робити контрольні точки, що спрощує тестування. Наприклад якщо якийсь експеримент з кодом контролера або свіча завершиться помилкою, ми легко можемо повернутися до попереднього стану.

Серед плюсів цього програмного забезпечення є те що воно має інструменти інтеграції з хост-системою, а саме:

- Спільний буфер обміну який спрощує обмін файлами між основною системою та віртуальною ОС.
- Швидке підключення зовнішніх носіїв та мережевих інтерфейсів, що полегшує перенесення логів, результатів експериментів.

Окрему увагу я думаю треба приділити тому, що VMware дозволяє експортувати створену віртуальну машину у форматі .ova та .vmx, що робить середовище переносним. А це означає що всю систему можливо перенести с одного ПК на інший.

1.2.4 Емулятор мережі Mininet

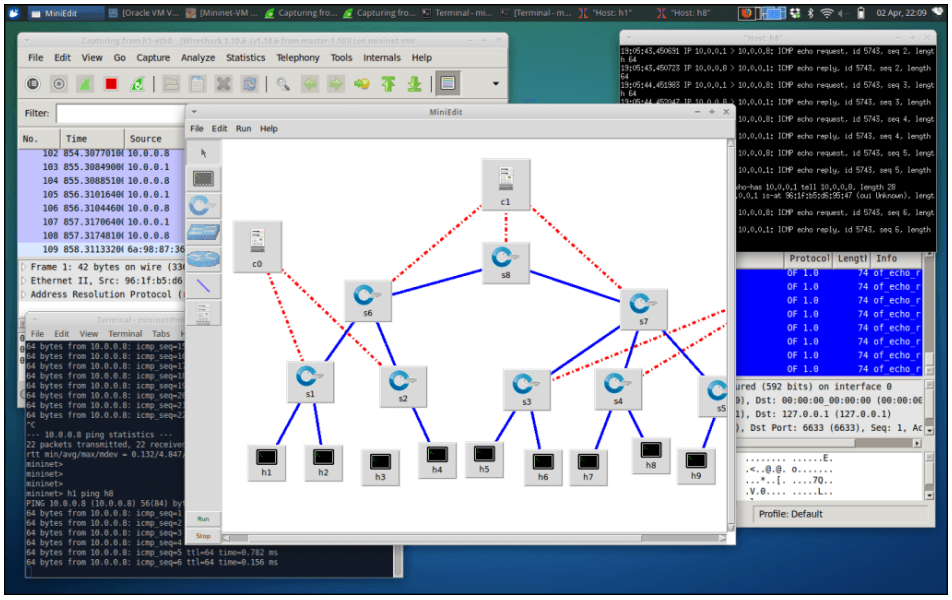


Рисунок 1.5 – Емулятор мережі Mininet

На рис 1.5 зображені Основні органи керування емулятором Mininet та його графічне середовище Miniedit, який дозволяє замість написання коду, створювати та налаштовувати мережу за допомогою графічних елементів.

Mininet – це програмне середовище створене для дослідження, тестування та розробки архітектур SDN. Воно дозволяє повноцінно відтворити повноцінну мережу з комутаторів, хостів і контролерів, не використовуючи фізичне обладнання. Перевагою використання було те що Mininet не вимагає потужного обладнання. Всю конфігурацію можна створити через командний рядок ОС Linux, або за допомогою графічного інтерфейсу Miniedit.

1.2.5 bmv2 свічі та мова P4

Bmv2 свічі це програмна модель комутатора, створена для роботи з мовою P4. Він вміє відтворювати поведінку реального мережевого пристрою, який можна перепрограмувати під власні потреби. У таких проектах як у нас, цей тип свіча використовується для тестування логіки, обробки пакетів.

На відміну від звичайних комутаторів у яких функції в основному фіксовані, цей тип комутатора дозволяє визначити як саме обробляти трафік. Сам свіч bmv2 працює з мовою P4 яка описує структуру пакетів, таблиці пошуку, дії при їх проходженні й правила пересилання. Після того як програма була скомпільована, вона завантажується в свіч і починає діяти відповідно до заданої логіки.

1.3 Порівняльний аналіз

Під час вибору середовища для побудови мережі було розглянуто декілька рішень, такі як VirtualBox, Docker, GNS3.

Найменш стабільним в цьому плані виявився VirualBox. Попри відкритість і простоту, дане ПЗ погано справляється з запуском кількох комутаторів bmv2 та контролерів ONOS. Були виявлені збої в обробці пакетів та непередбачувані затримки передачі.

Технологія Docker пропонує мінімальне використання ресурсів, але її головне обмеження полягає у відсутності повної віртуалізації ядра. Docker-контейнери розділяють одне ядро з хост системою і через це не дозволяють створювати окремі незалежні мережеві простори з власними таблицями маршрутизації чи протоколами.

Використання системи GNS3 я виявив недоцільним, через те що це середовище підтримує тільки класичні мережі, але не підтримує сучасні SDN контролери з їх інтерфейсами.

Для запуску ПЗ Mininet була обрана ОС Ubuntu Linux через її гнучкість, стабільність та відкритість. На відміну від Windows, в цій ОС ми маємо прямий доступ до мережевих стеків ядра, а також підтримку сучасних версій Python, gRPC, компілятора p4c, та оптимізоване ядро для роботи в віртуальних середовищах. Інші дистрибутиви такі як Debian та Fedora мають аналогічне стандартну базу пакетів, але поступаються швидкістю встановлення та

обсягом спільноти. Arch Linux та Cent OS вимагають додаткового налаштування ядра, що також ускладнює швидке налаштування.

Щодо середовищ моделювання існують різні аналоги Mininet, серед яких NS-3, Containernet, Emulab та CORE Network Emulator.

NS-3 має високу точність у відтворенні протоколів та фізичного рівня, але орієнтований більше на симуляцію традиційних мереж і не підтримує інтеграцію з SDN. Система Emulab надає хмарне середовище для наших експериментів, але не дозволяє локального тестування. CORE і Containernet використовують контейнерну архітектуру та не мають повної сумісності з P4Runtime і обмежені в налаштуванні топологій з використанням програмованих свічів bmv2.

1.4 Перспективи розвитку проекту

Проект має перспективи для подальшого розвитку, які можуть поліпшити його функціональність:

1)Розширення функціональних можливостей завдяки алгоритму машинного навчання який може бути впроваджений в контролер ONOS. Це може допомогти контролеру чіткіше аналізувати трафік, прогнозувати перевантаження трафіку та швидше коригувати його перенаправлення.

2)Подальше вдосконалення телеметрії, а саме впровадження телеметрії INT. Це створить додаткові умови для більше детальнішого аналізу трафіку, а це, в свою чергу, допоможе якісніше прогнозувати збої в роботі мережі.

3)Ще одним напрямком може стати розподіленість контролерів в різних середовищах. Наприклад якщо один буде на підприємстві а інший в хмарі. Таким чином ми можемо заробити систему більш стабільною до глобальних збоїв і дозволить забезпечити безперервність обслуговування при втраті зв’язку.

2 ВСТАНОВЛЕННЯ ТА НАЛАШТУВАННЯ СИСТЕМИ

2.1 Опис та налаштування програмних засобів

2.1.1 Запуск та налаштування VMware workstation 17 pro

Програмне забезпечення VMware Workstation 17 Pro створена для створення та управління віртуальними машинами, що дозволяє запускати декілька операційних систем одночасно на одному фізичному комп’ютері з повною емуляцією апаратного середовища. Далі на рисунку 2.1 показана встановлене ПЗ та його головне вікно.

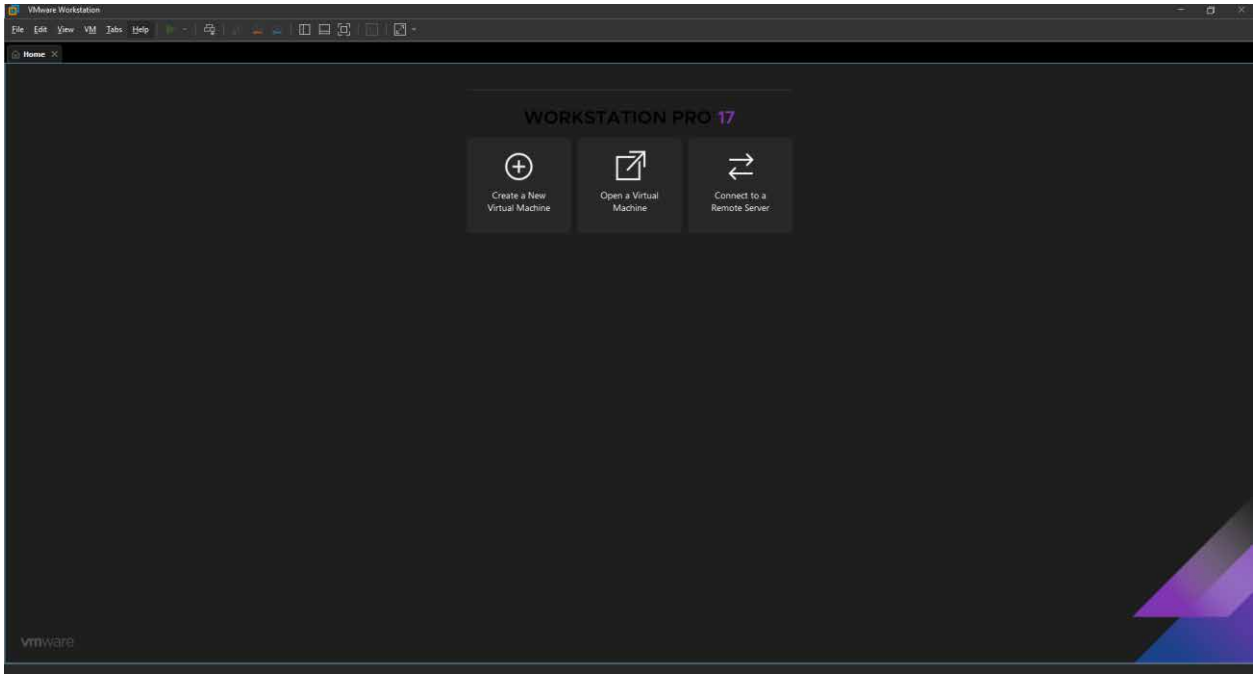


Рисунок 2.1 – Головне вікно ПЗ VMware

Для справної роботи програми нам потрібний ПК з такими мінімальними характеристиками:

1. Процесор з підтримкою 64-бітної архітектури та технологіями Intel VT-x або AMD-V;
2. Не менше 4 гігабайтів оперативної пам’яті (рекомендовано 8 ГБ);

3. Сама програма займає 1.5 ГБ місця на диску. Але потрібне додаткова пам'ять для жорстких дисків.
4. Операційна хост-система. Чудово підійде Windows 10/11 (64 bit) або Linux.

Рисунок 2.3 – Перший етап створення ВМ

ПЗ VMware дає на вибір два варіанти створення ВМ (Рисунок 2.3). В типовому режимі ми швидко створюємо машину в декілька етапів з стандартними характеристиками. В модифікованому варіанті можемо вказати з якими версіями буде працювати наша ВМ, тип мережевого адаптера, тип контролеру диску та форматом ВМ.

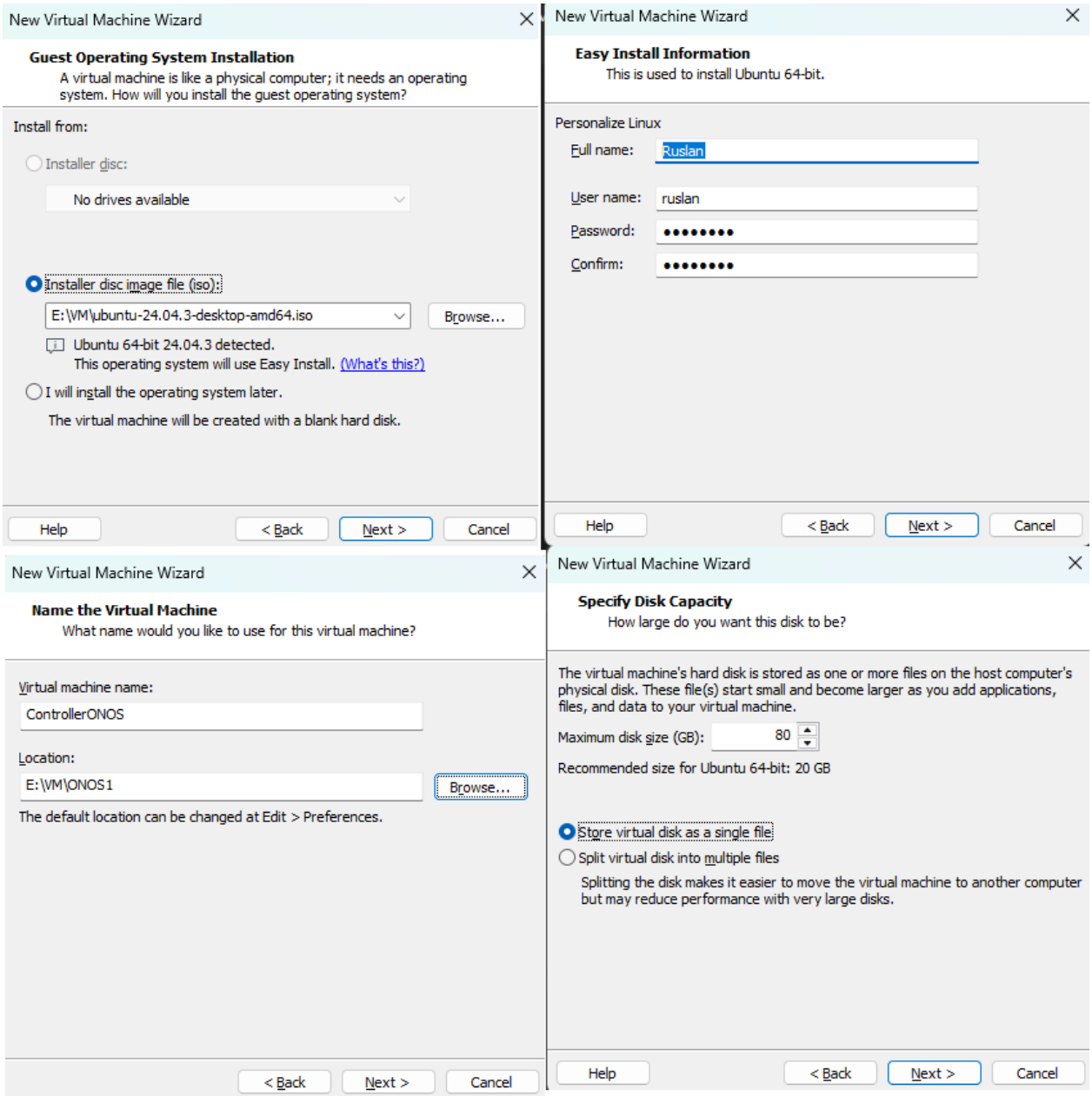


Рисунок 2.4 – Типове встановлення ВМ

При стандартному встановленні ми маємо вказати шлях до .iso файлу з потрібною ОС на якій буде працювати наш контролер. Далі вказуємо ім’я та пароль адміністратора та ім’я віртуальної машини. Вказуємо розмір для віртуального жорсткого диску. В кінці встановлення ми бачимо фінальну сторінку з всіма характеристиками які ми вибрали та натискаємо finish.

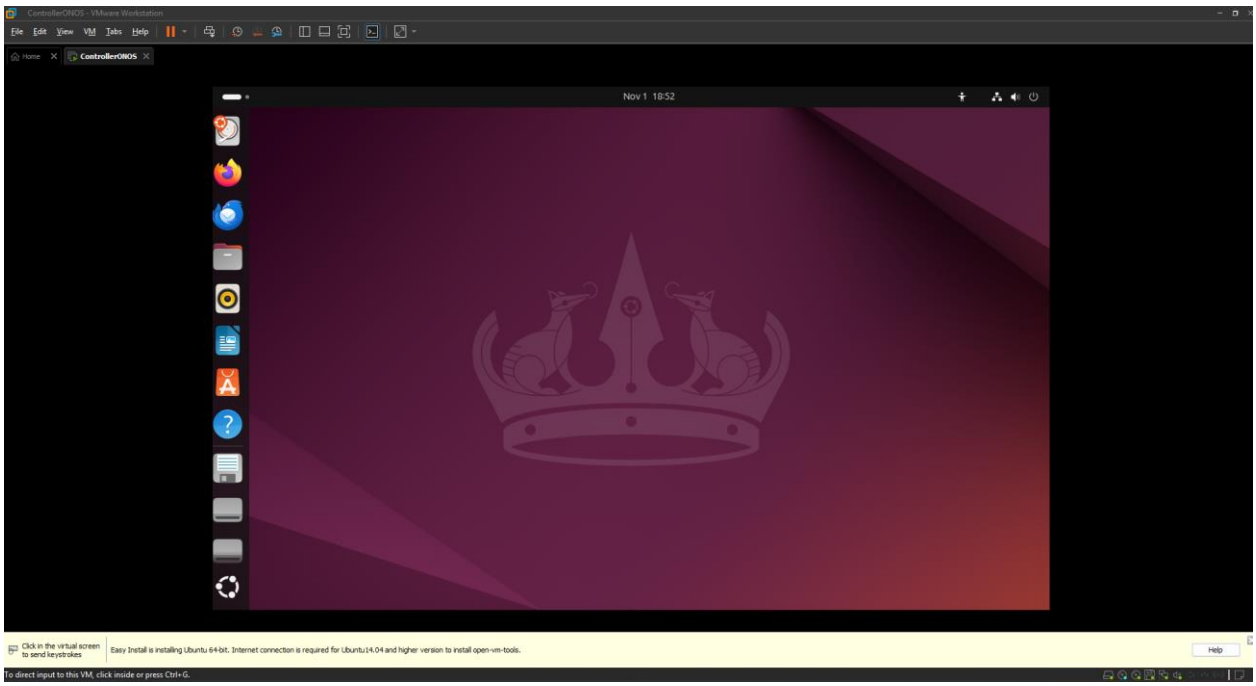


Рисунок 2.5 – Встановлена та запущена ОС Ubuntu

Тепер після встановлення ми маємо повноцінну ОС як працює окремо від основної операційної системи. Після завершення інсталяції ми можемо починати налаштовувати наш контролер.

2.1.3 Встановлення та налаштування Mininet

Тепер, коли віртуальне середовище готове ми можемо починати встановлювати та налаштовувати Mininet, який використовується для моделювання та функціонування SDN мережі.

Встановлюється це ПЗ в терміналі, що забезпечує прямий доступ до системних ресурсів Linux. Е для початку нам потрібно виконати базове оновлення системи щоб гарантувати що всі пакети будуть останніх версій і в нас не буде конфліктів під час інсталяції. Команда для оновлення бібліотек пишеться так, sudo apt update && sudo apt upgrade -y.

Також після оновлення нам потрібно встановити залежності для справної роботи системи та роботи з віртуальними інтерфейсами.

Основний пакет Mininet завантажується з офіційного репозиторію Github, та встановлюється завдяки скрипту install.sh.

Після встановлення ми можемо перевірити працездатність середовища за допомогою команди sudo mn --test pingall.

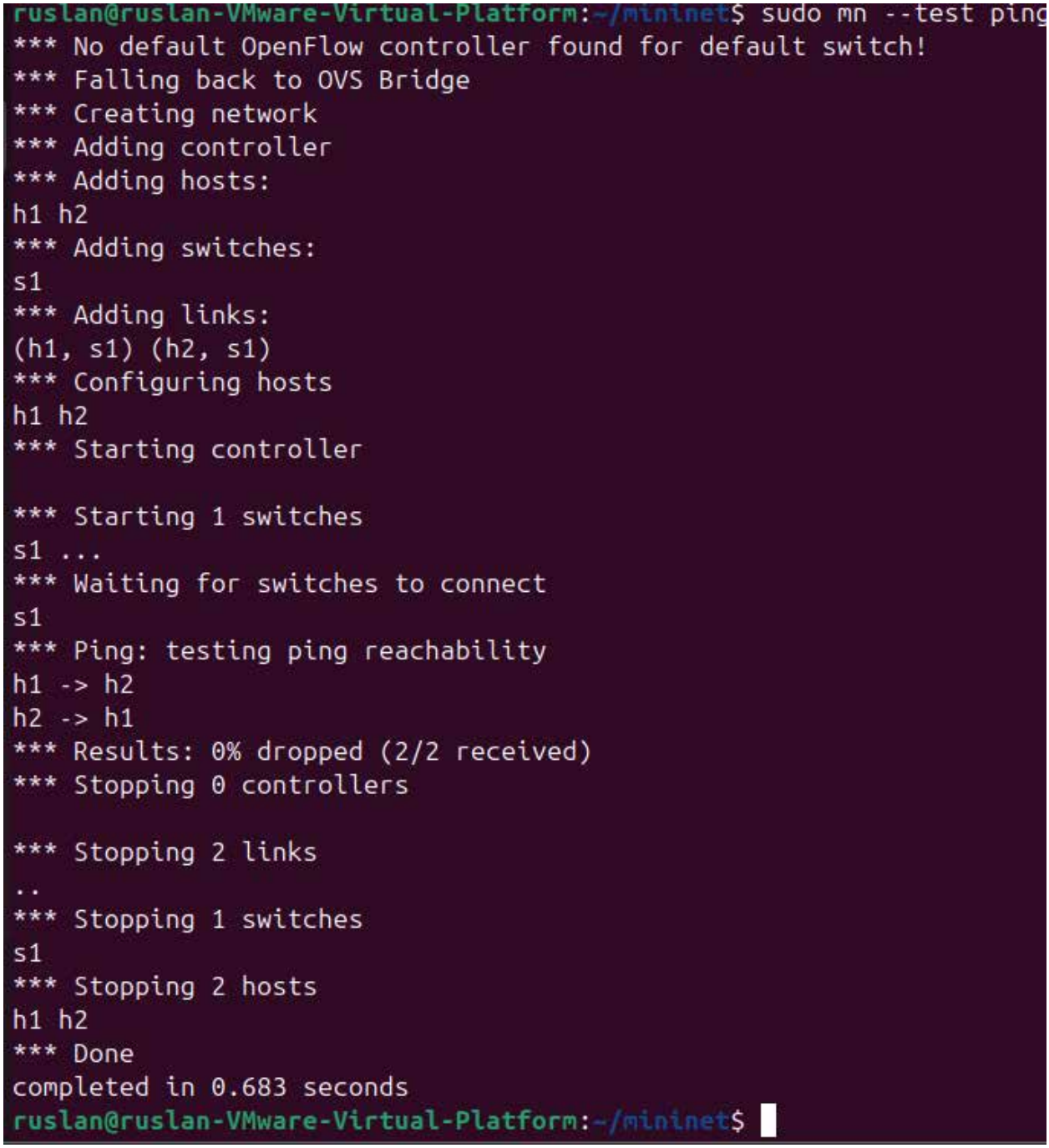


Рисунок 2.6 – Виконання команди sudo mn --test pingall

Як бачимо з рисунку 2.6, команда запустила Mininet, створила тестову мережу та обмінялася пакетами між тестовими хостами.

2.1.4 Встановлення та налаштування ONOS

Після того як ми встановили емулятор мережі, можемо починати встановлювати контролер ONOS. Цей елемент мережі є основним так як відповідає за взаємодію між рівнем керування та комутаторами, дозволяє програмно створювати, змінювати та оптимізувати маршрути передачі даних.

Для встановлення нам потрібно завантажити офіційний репозиторій з Github та встановити його завдяки інсталятору Bazel. Інсталяція контролеру в нашу ОС займає десь від 15-20 хвилин.

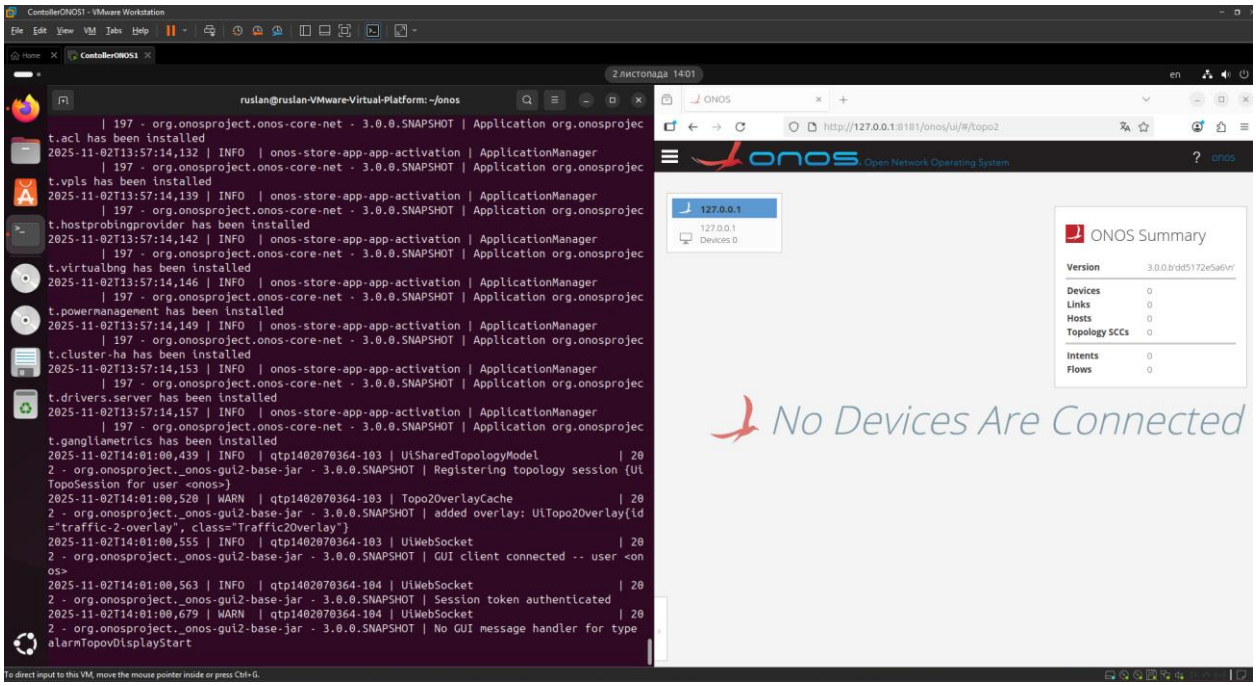


Рисунок 2.7 – Веб інтерфейс контролера ONOS

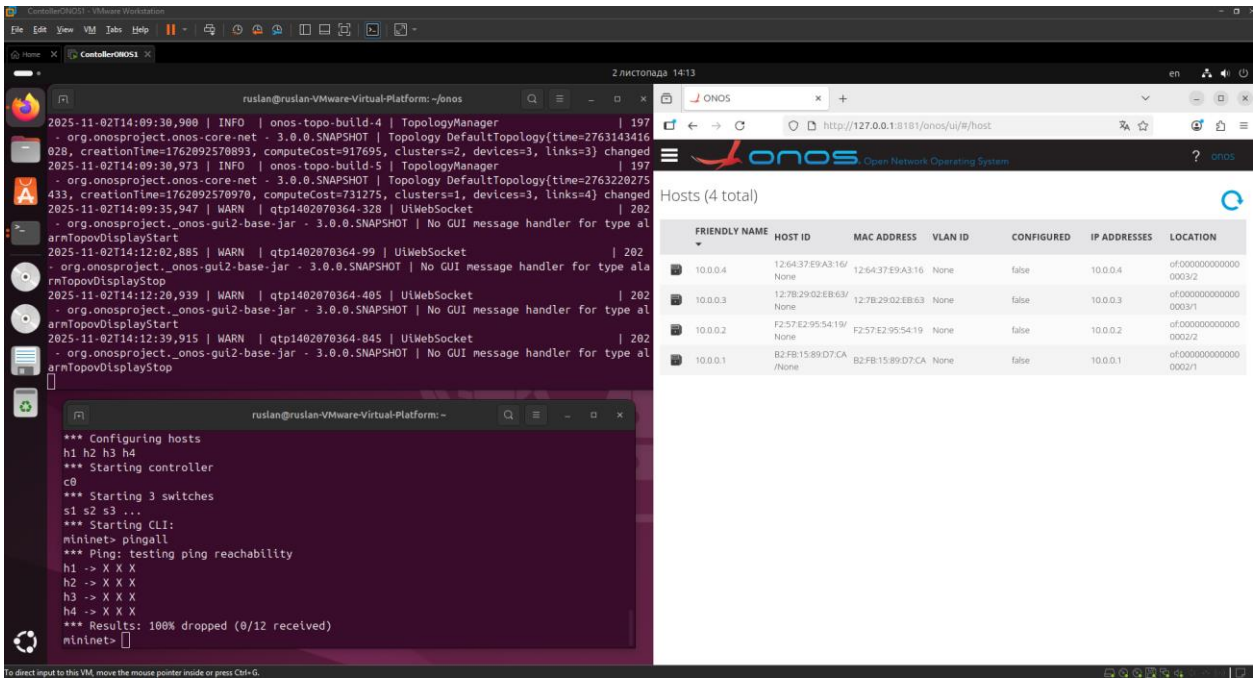


Рисунок 2.8 – Тестовий запуск Mininet з контролером ONOS

Клонуємо систему і називаємо її ONOS2(Backup) для створення нашого другого контролеру. В налаштуваннях ВМ треба вказати параметри мережі Host-Only, для того щоб у нас була своя мережа окрема від основної мережі хост-ПК. В процесі налаштування наші ВМ отримують різні IP адреси. Виконуємо команду ping (рисунок 2.9) щоб пересвідчитись що два контролери обмінюються пакетами.

```
ruslan@onos1:~$ ping 192.168.150.128
PING 192.168.150.128 (192.168.150.128) 56(84) bytes of data.
64 bytes from 192.168.150.128: icmp_seq=1 ttl=64 time=0.774 ms
64 bytes from 192.168.150.128: icmp_seq=2 ttl=64 time=0.392 ms
64 bytes from 192.168.150.128: icmp_seq=3 ttl=64 time=0.398 ms
64 bytes from 192.168.150.128: icmp_seq=4 ttl=64 time=0.404 ms
64 bytes from 192.168.150.128: icmp_seq=5 ttl=64 time=0.528 ms
^C
--- 192.168.150.128 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4110ms
rtt min/avg/max/mdev = 0.392/0.499/0.774/0.146 ms
```

Рисунок 2.9 – Виконання команди ping з контролера ONOS1

Тепер нам потрібно створити кластер з двох контролерів щоб вони змогли замінювати одне одного у випадку аварії.

Зв’язуються два контролери через протокол SSH без пароля щоб наші контролери могли синхронізуватись без введення паролів. Далі ми створюємо конфіг-файл cluster.json з наступним вмістом (лістинг 2.1).

Лістинг 2.1 – Код конфігурація файлу cluster.json

```
{
  "nodes": [
    { "id": "onos-1", "ip": "192.168.150.129", "port": 9876, "hostname": "onos1" },
    { "id": "onos-2", "ip": "192.168.150.128", "port": 9876, "hostname": "onos2" }
    ]
}
```

Цей файл описує вузли які утворюють кластер і визначає їх IP-адреси, порти та імена. Порт 9876 використовується ONOS для внутрішнього обміну між контролерами.

В обох системах в файлі bashrc ми додаємо змінні середовища в кінці файлу (лістинг 2.2).

Лістинг 2.2 – Код змінних середовищ файлу bashrc

```
export ONOS_IP=192.168.150.129     # для onos1
export ONOS_IP=192.168.150.128     # для onos2
export ONOS_NIC=ens33
export ONOS_OPTS="-Djava.net.preferIPv4Stack=true -Donos.node.ip=$ONOS_IP -Donos.cluster.metadata.uri=file:/home/ruslan/onos/config/cluster.json"
source ~/onos/tools/dev/bash_profile
```

Ці змінні вказують контролеру який IP використовувати для власної ідентифікації а також звідки брати інформацію про ідентифікацію кластеру. Після цього обидва вузли мають бути перезапущені а зміни в файлах активовано командою source ~/.bashrc.

Для перевірки стану кластеру ми використовуємо команду curl та отримуємо відповідь наведену в рисунку 2.10.

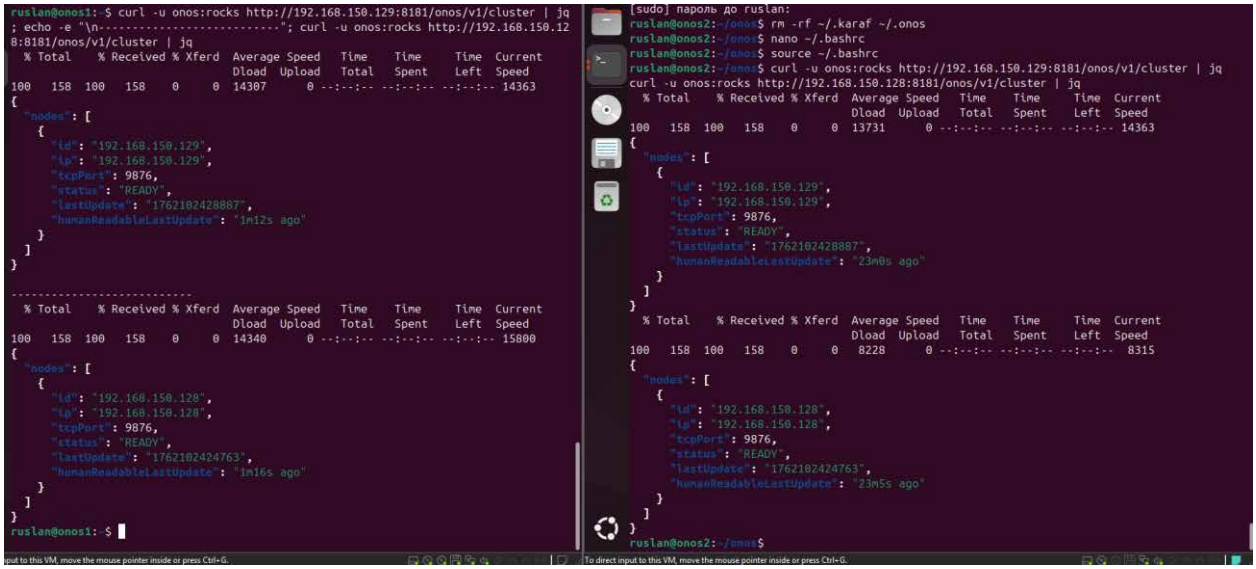


Рисунок 2.10 – Відповідь з обох контролерів

На рисунку видно що два контролери відповідають по своїх IP адресах, а отже кластер було створено успішно.

2.1.5 Встановлення та налаштування свічів behavioral-model та P4Runtime

Behavioral model або його абревіатура BMv2, це віртуальний комутатор який був розроблений для експериментів так як він не прив’язаний до фізичних можливостей обладнання. В цьому типі свіча ми можемо самі визначати логіку обробки пакетів через мову P4.

P4Runtime – це канал через який контролер “спілкується” з комутатором. Саме через нього контролер завантажує в свіч таблиці правил, конфігурацію, отримує телеметрію.

Працює це все за принципом:

- P4-код визначає як буде працювати мережа та компілюється в конфігурацію.
- Через канал P4Runtime ця конфігурація завантажується в свіч BMv2.
- Контролер ONOS через цей ж канал слідкує за станом мережі.

Встановлюється це все таким чином. Для початку нам потрібно створити нову ВМ з нашою ОС Ubuntu. Після встановлення ОС ми оновлюємо її та встановлюємо основні залежності без яких робота bmv2 неможлива. Щодо залежностей, то їх буває два типи, залежності для збірки системи та запуску. До першого відносяться компілятори, автовузли, заголовки. До другого, динамічні бібліотеки. Ці залежності нам потрібні для всіх додатків тому ми будемо встановлювати їх постійно під різні задачі, наприклад для справної роботи bmv2 нам потрібен Thrift для того щоб свіч міг керуватись через API. Також нам потрібен компілятор P4C. На цьому етапі ми також встановлюємо Mininet та P4Runtime. Щоб перевірити чи встановились наші додатки можна викликати опис їх версії. Якщо все добре то система просто видасть номер версії.

```
ruslan@switch1:~/PI/proto/p4runtime/py$ p4c-bm2-ss --version
p4c-bm2-ss
Version 1.2.5.8 (SHA: a97290474 BUILD: Release)
ruslan@switch1:~/PI/proto/p4runtime/py$ simple_switch --version
1.15.0-2bdd0b7b
```

Рисунок 2.11 – Версії встановлених елементів.

Далі нам потрібно написати код який і буде формувати логіку поведінки нашого свіча. Код наведено в лістингу 2.3.

Лістинг 2.3 – Код функціонування свіча

```
#include <core.p4>
#include <v1model.p4>

typedef bit<9>  egressSpec_t;
typedef bit<32> ipv4Addr_t;
typedef bit<48> macAddr_t;

header ethernet_t { macAddr_t dstAddr; macAddr_t srcAddr;
bit<16> etherType; }
header ipv4_t {
  bit<4>   version; bit<4> ihl; bit<8> diffserv; bit<16>
totalLen;
  bit<16> identification; bit<3> flags; bit<13> fragOffset;
  bit<8> ttl; bit<8> protocol; bit<16> hdrChecksum;
  ipv4Addr_t srcAddr; ipv4Addr_t dstAddr;
}

struct headers_t { ethernet_t ethernet; ipv4_t ipv4; }
struct metadata_t { }

parser MyParser(packet_in packet,
                out headers_t hdr,
                inout metadata_t meta,
                inout standard_metadata_t stdmeta) {
  state start {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
      0x0800: parse_ipv4;
```

```
            default: accept;
          }
        }
        state parse_ipv4 {
          packet.extract(hdr.ipv4);
          transition accept;
```

Продовження лістингу 2.3 – Код функціонування свіча

```
        }
      }

      control MyVerifyChecksum(inout headers_t hdr, inout
metadata_t meta) { apply { } }

      control MyIngress(inout headers_t hdr,
                        inout metadata_t meta,
                        inout standard_metadata_t stdmeta) {

        action action_forward(egressSpec_t port) {
stdmeta.egress_spec = port; }

        table forward {
          key = { hdr.ethernet.dstAddr : exact; }
          actions = { action_forward; NoAction; }
          size = 1024;
          default_action = NoAction();
        }

        apply { forward.apply(); }
      }

      control MyEgress(inout headers_t hdr,
                       inout metadata_t meta,
                       inout standard_metadata_t stdmeta) { apply
{ } }
```

```
control MyComputeChecksum(inout headers_t hdr, inout
metadata_t meta) { apply { } }

control MyDeparser(packet_out packet, in headers_t hdr) {
  apply {
    packet.emit(hdr.ethernet);
```

Продовження лістингу 2.3 – Код функціонування свіча

```
    packet.emit(hdr.ipv4);
  }
}
V1Switch(MyParser(), MyVerifyChecksum(), MyIngress(),
   MyEgress(), MyComputeChecksum(), MyDeparser()) main;
```

Це P4-16 програма під bmv2 свіча. Для початку вона виконує прості задачі, такі як L2-форвардинг, парсить Ethernet/IPv4. Ingress знаходиться таблия forward яка зіставляє MAC призначення з портом і виставляє egress_spec. Deparser формує вихідні заголовки.

Далі нам потрібно запустити наші ONOS свічі та увімкнути потрібні додатки та додати наш пайплайн. Тобто наш контролер тепер знає як спілкуватись з свічем через P4Runtime і яку програму на ньому очікувати.

Після налаштування свіча ми бачимо що він з’явився на веб-панелі обох контролерів(рисунок 2.13), з цієї ж веб-панелі ми можемо переглядати інформацію про порти цих свічів (рисунок 2.12).

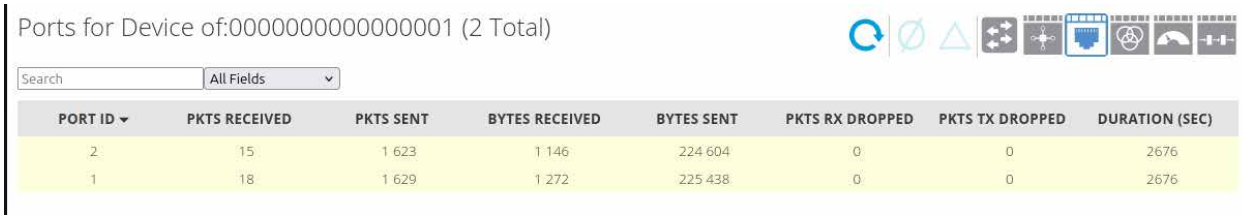


Рисунок 2.12 – Список портів та їх стан

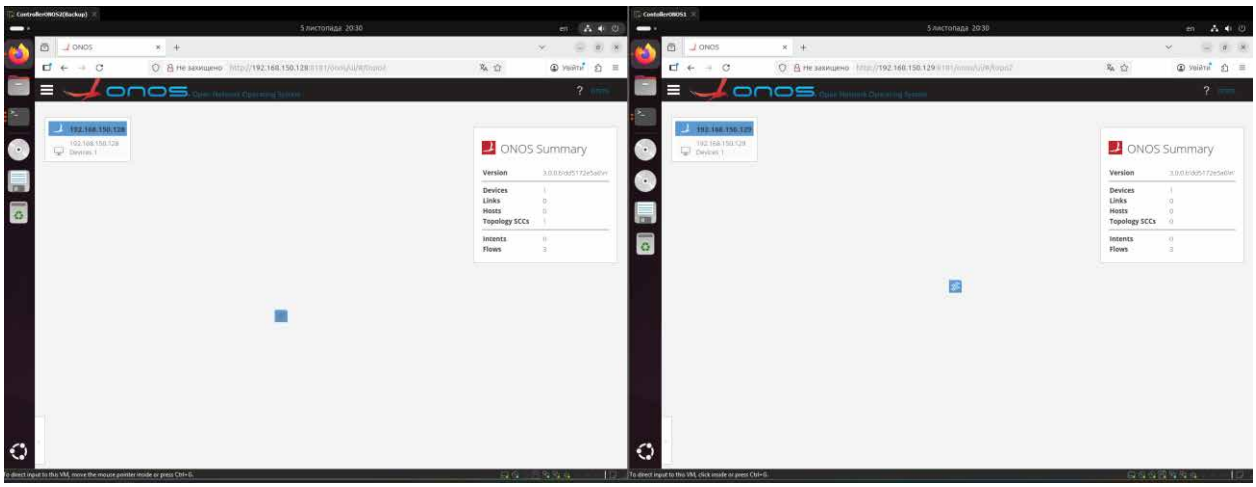


Рисунок 2.13 – Веб панелі контролерів з підключеним одним свічем

Аналогічно як з контролерами ми можемо клонувати свіч та створити нові з вже встановленими залежностями та потрібними додатками. Щоб не плутатись ми називатимемо їх Switch1, Switch2, Switch3 (рисунок 2.14). Таким чином все що нам залишається це змінити налаштування та почати підключати хости.

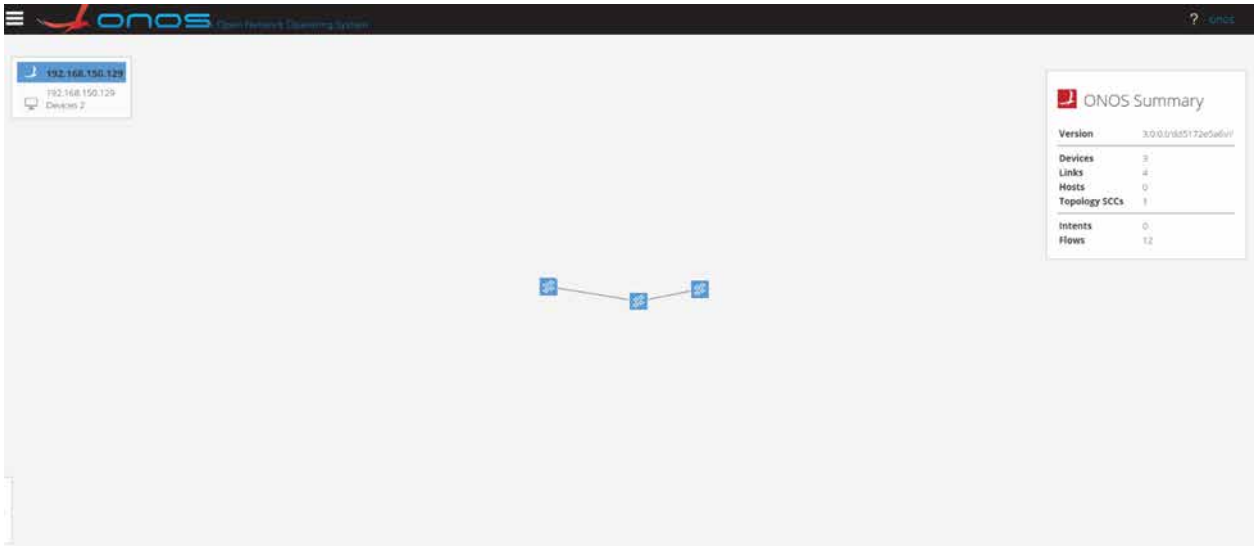


Рисунок 2.14 – Веб інтерфейс контролера з підключеними свічами

3 ТЕСТУВАННЯ ПОБУДОВАНОЇ СИСТЕМИ

3.1 Задачі тестування

Для побудування надійної системи потрібно довести її стресостійкість шляхом тестування. Основою будь якої мережі SDN є її контролер та власне сама мережа. Будь-яка помилка в топології чи в налаштуванні, може призвести до збою в роботі сервісу, наприклад падіння пропускної здатності, ріст затримок передачі пакетів та їх втрата. Задачою тестування є перевірка на:

- Відновлення зв’язку при збоях, якого можна досягнути завдяки функції перебудування шляху пакетів.
- Забезпечення швидкості передачі пакетів при великих навантаженнях спричинених великою кількістю команд.
- Масштабованість системи.

Тестування будемо проводити за допомогою утілити iperf. Ця функція призначена для вимірювання пропускної здатності мереж. Використовувати будемо її для вимірювання швидкості передачі даних між нашими вузлами шляхом створення контрольованого трафіку.

Принцип роботи такий, один вузол запускається в режимі сервера й він приймає вхідні з’єднання, а інший вузол буде працювати в режимі клієнта, та ініціюватиме передавання даних.

Під час тесту ця утиліта сформує список який буде складатись з обсягу передачі даних, середню пропускну здатність каналу, кількістю повторних передач, затримки та втрати пакетів.

3.1.1 Тестування перебудови шляху пакетів

Одним з основним тестів нашої мережі буде перевірка на спроможність системи змінити маршрут пакетів до хостів у разі збою одного з свічів. Цей тест допоможе оцінити, наскільки швидко контролер реагує на зміну стану

мережі, як швидко відновлюється зв'язок між вузлами, та які обсяги даних ми можемо втратити при таких збоях.

Для цього тесту ми використовуватимемо кілька наших програмованих свічів, підєднаних до контролеру ONOS, який і буде відновлювати зв'язок між вузлами та розподілювати навантаження. Між хостами створюється сталий потік трафіку за допомогою утиліти iperf, шо дає змогу фіксувати часи відгуку, пропускну здатність і можливі втрати пакетів у реальному часі. Після стабілізації трафіку ми будемо відключати свіч та дивитись як швидко трафік назад прийде в норму.

У процесі тестування фіксується момент виникнення події, час за який контролер виявляє зміну стану порту, а також момент відновлення передачі пакетів.

Перед початком тесту ми запускаємо наш контролер. Після цього в нашому середовищі запускаємо свічі та завантажуємо в них конфігурацію, яка буде описувати структуру мережі, а саме кількість хостів і з’єднання між ними. Код конфігурації наведено в лістингу 3.1.

Лістинг 3.1 - код конфігурації мережі

```
from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel

ONOS_IP = '192.168.150.129'
ONOS_PORT = 6653

def main():
    setLogLevel('info')
    net = Mininet(controller=None, switch=OVSSwitch,
link=TCLink, build=False)
```

Продовження лістингу 3.1 - код конфігурації мережі

```
        c0 = net.addController('c0',
controller=RemoteController, ip=ONOS_IP, port=ONOS_PORT)

        s1 = net.addSwitch('s1', protocols='OpenFlow13',
failMode='secure')
        s2 = net.addSwitch('s2', protocols='OpenFlow13',
failMode='secure')
        s3 = net.addSwitch('s3', protocols='OpenFlow13',
failMode='secure')

        h1 = net.addHost('h1', ip='10.0.0.1/24')
        h2 = net.addHost('h2', ip='10.0.0.2/24')
        h3 = net.addHost('h3', ip='10.0.0.3/24')
        h4 = net.addHost('h4', ip='10.0.0.4/24')
        h5 = net.addHost('h5', ip='10.0.0.5/24')
        h6 = net.addHost('h6', ip='10.0.0.6/24')

        net.addLink(h1, s1); net.addLink(h2, s1)
        net.addLink(h3, s2); net.addLink(h4, s2)
        net.addLink(h5, s3); net.addLink(h6, s3)

        net.addLink(s1, s2)
        net.addLink(s2, s3)

        net.build()
        c0.start()
        s1.start([c0]); s2.start([c0]); s3.start([c0])

        for sw in (s1, s2, s3):
            sw.cmd(f'ovs-vsctl set bridge {sw.name}
protocols=OpenFlow13')
            sw.cmd(f'ovs-vsctl set-fail-mode {sw.name} secure')
```

Продовження лістингу 3.1 - код конфігурації мережі

```
        CLI(net)
        net.stop()

    if __name__ == '__main__':
        main()
```

Цей код ініціює завантаження нашої мережі, яка містить 6 хостів та 3 свіча. Перевірити та вивести список хостів та їх підключення можна за допомогою команди net (Рисунок 3.1). Використовуємо команду pingall яка перевірить з'єднання між хостами (Рисунок 3.2).

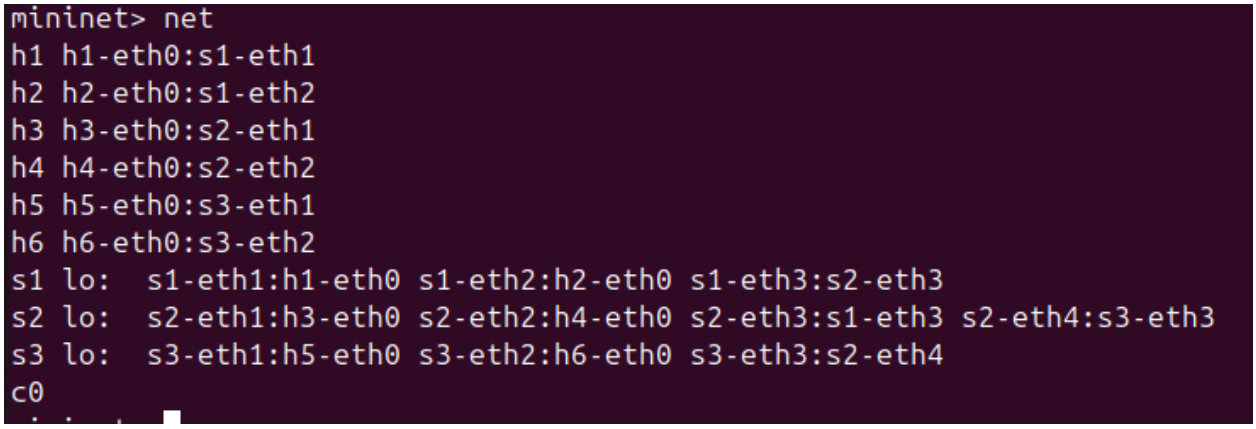


Рисунок 3.1-Виконання команди net.

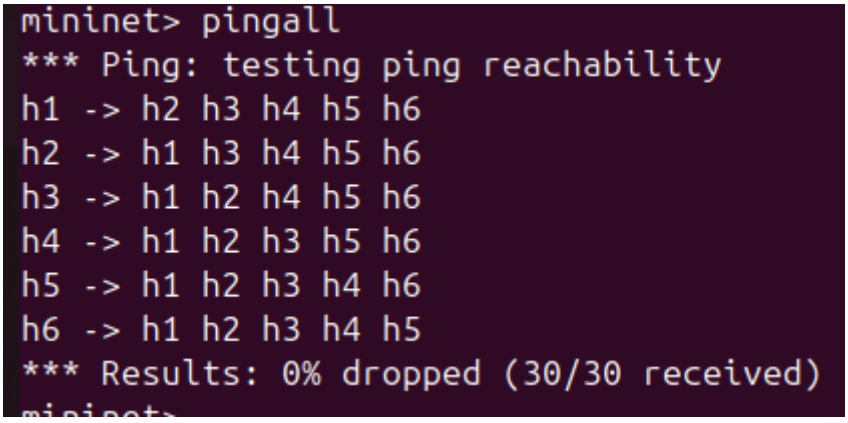


Рисунок 3.2 – Виконання команди pingall

Після побудови та перевірки мережі можемо починати тестування на перебудову шляху при збоях. Під час справної роботи мережі пакети передаються найкоротшим шляхом, який визначається контролером на основі таблиць потоків. Для симуляції збою ми, за допомоги утиліти iperf, на хості який буде сервером застосовуємо команду iperf -s яка ініціює очікування вхідні з'єднання. А на стороні клієнта виконується команда iperf -c 10.0.0.2 -t 20, що створює безперервний потік даних тривалістю 20 секунд у напрямку серверу. Під час передачі ми будемо бачити поточну швидкість в МБ/С. Коли ми навмисно відключимо свіч, трафік тимчасово зникне, а швидкість передачі впаде до нуля. У цей момент контролер виявить помилку та перебудує маршрут через інші активні зв'язки, а швидкість та трафік знову прийдуть в норму. Результати виконання наведені в таблиці 3.1.

Таблиця 3.1 – Результати виконання команди iperf

№	ID Потоку	Проміжок часу в секундах	Обсяг переданих даних	Середня швидкість	Кількість повторно переданих сегментів TCP
1	[5]	0.00-1.00	4.31 GBytes	37.0 Gbits/sec	0
2	[5]	1.00-2.00	4.36 GBytes	37.5 Gbits/sec	0
3	[5]	2.00-3.00	4.28 GBytes	36.8 Gbits/sec	0
4	[5]	3.00-4.00	4.40 GBytes	37.9 Gbits/sec	0
5	[5]	4.00-5.00	4.26 GBytes	36.7 Gbits/sec	0
6	[5]	5.00-6.00	4.33 GBytes	37.2 Gbits/sec	0
7	[5]	6.00-7.00	4.35 GBytes	37.4 Gbits/sec	0
8	[5]	7.00-8.00	4.29 GBytes	36.9 Gbits/sec	0
9	[5]	8.00-9.00	4.27 GBytes	36.7 Gbits/sec	0

10	[5]	9.00-10.00	4.35 GBytes	37.4 Gbits/sec	0
11	[5]	10.00-11.00	0.32 GBytes	2.8 Gbits/sec	0
12	[5]	11.00-12.00	0.00 GBytes	0.0 Gbits/sec	0
13	[5]	12.00-13.00	0.15 GBytes	1.3 Gbits/sec	0
14	[5]	13.00-14.00	2.17 GBytes	18.7 Gbits/sec	0
15	[5]	14.00-15.00	3.52 GBytes	30.3 Gbits/sec	0
16	[5]	15.00-16.00	4.05 GBytes	34.8 Gbits/sec	0
17	[5]	16.00-17.00	4.29 GBytes	36.8 Gbits/sec	0
18	[5]	17.00-18.00	4.36 GBytes	37.5 Gbits/sec	0
19	[5]	18.00-19.00	4.30 GBytes	37.0 Gbits/sec	0
20	[5]	19.00-20.00	4.32 GBytes	37.2 Gbits/sec	0

У таблиці наведено результати роботи мережі до та після перебудови шляху пакетів. Ми запустили команду iperf та генерували трафік. До 10 секунди робота мережі була стабільною. Починаючи з 10 секунди, я вимкнув порти на свічі s1, ввівши команду portstate of:0000000000000001 1 disable та portstate of:0000000000000001 2 disable, в Apache Karaf. Ця команда повністю відключила порти які підключені до хостів h1 та h2. На 11 секунді в нас повністю зник трафік. Починаючи з цього моменту трафік був переправлений контролером через свіч s2, а робота мережі відновилась.

3.1.2 Дослідження впливу flow-rule на якість передачі

Як вже було описано, для керування мережею контролер відправляє або flow-rule. Це команди які контролер відправляє комутатору. Ці команди містять інформацію про правила маршрутизації, параметри потоків і вказівки для свіча як діяти при отримані певних пакетів та збоях. Наприклад вказівка пересилати трафік через інший порт або змінити адресу призначення, чи записати статистику передачі даних. Якщо контролер надсилає надто багато команд за короткий проміжок часу, свіч може почати витрачати ресурси не на передачу пакетів а на оновлення власних таблиць потоків. Це призведе до короткочасної втрати з'єднання або навіть повної втрати. Саме тому, за допомогою цього тесту я спробую визначити яка кількість команд може призвести до описаних вище, збоїв.

Під час тесту я буду поступово збільшувати кількість керуючих команд, а результати цього збільшення буду визначати за допомогою утиліти iperf. Вона допоможе визначити затримки, втрати пакетів та пропускну здатність мережі. Порівнюючи результати при різних рівнях навантаження, можна буде побачити наскільки ефективно контролер і свічі витримують ріст кількості команд і як це буде впливати на якість роботи мережі.

Для початку тесту потрібно зробити теж саме що в попередньому пункті, а саме, запустити контролер та завантажити конфігурацію (лістинг 3.1) в свічі. Завантажуємо утиліту ipref та логування передачі даних між хостами. В якості серверу який буде приймати запити ми вибираємо хост h6, всі інші хости будуть клієнтами (Рисунок 3.3). Як бачимо з рисунку 3.4, хост справно приймає пакети.

```
mininet> h6 iperf3 -s &
-----------------------------------------------------------
Server listening on 5201 (test #1)
-----------------------------------------------------------
[2] 5271
mininet> h1 iperf3 -c 10.0.0.6 -t 20 -i 1 > /tmp/h1.log &
mininet> h2 iperf3 -c 10.0.0.6 -t 20 -i 1 > /tmp/h2.log &
mininet> h3 iperf3 -c 10.0.0.6 -t 20 -i 1 > /tmp/h3.log &
mininet> h4 iperf3 -c 10.0.0.6 -t 20 -i 1 > /tmp/h4.log &
mininet> h5 iperf3 -c 10.0.0.6 -t 20 -i 1 > /tmp/h5.log &
```

Рисунок 3.3 – Навантаження хоста h6

```
mininet> sh grep "Gbits/sec" /tmp/h*.log
/tmp/h3.log:[  5]   0.00-1.00   sec  3.69 GBytes  31.7 Gbits/sec    1   6.32 MBytes
/tmp/h3.log:[  5]   1.00-2.00   sec  4.23 GBytes  36.4 Gbits/sec    1   4.78 MBytes
/tmp/h3.log:[  5]   2.00-3.00   sec  4.37 GBytes  37.6 Gbits/sec    2   4.81 MBytes
/tmp/h3.log:[  5]   3.00-4.00   sec  4.37 GBytes  37.6 Gbits/sec    1   4.83 MBytes
/tmp/h3.log:[  5]   4.00-5.00   sec  4.23 GBytes  36.3 Gbits/sec    1   3.63 MBytes
/tmp/h3.log:[  5]   5.00-6.00   sec  3.90 GBytes  33.5 Gbits/sec    0   3.83 MBytes
/tmp/h3.log:[  5]   6.00-7.00   sec  4.40 GBytes  37.8 Gbits/sec    0   4.08 MBytes
/tmp/h3.log:[  5]   7.00-8.00   sec  4.40 GBytes  37.8 Gbits/sec    0   4.28 MBytes
/tmp/h3.log:[  5]   8.00-9.00   sec  4.33 GBytes  37.2 Gbits/sec    0   4.45 MBytes
/tmp/h3.log:[  5]   9.00-10.00  sec  4.26 GBytes  36.6 Gbits/sec    1   4.45 MBytes
/tmp/h3.log:[  5]  10.00-11.00  sec  4.40 GBytes  37.8 Gbits/sec    1   4.45 MBytes
/tmp/h3.log:[  5]  11.00-12.00  sec  4.50 GBytes  38.6 Gbits/sec    2   3.23 MBytes
/tmp/h3.log:[  5]  12.00-13.00  sec  4.49 GBytes  38.6 Gbits/sec    2   3.24 MBytes
/tmp/h3.log:[  5]  13.00-14.00  sec  4.57 GBytes  39.3 Gbits/sec    0   3.26 MBytes
/tmp/h3.log:[  5]  14.00-15.00  sec  4.29 GBytes  36.8 Gbits/sec    0   3.27 MBytes
/tmp/h3.log:[  5]  15.00-16.00  sec  4.44 GBytes  38.2 Gbits/sec    0   3.27 MBytes
/tmp/h3.log:[  5]  16.00-17.00  sec  4.58 GBytes  39.3 Gbits/sec    0   3.28 MBytes
/tmp/h3.log:[  5]  17.00-18.00  sec  4.54 GBytes  39.0 Gbits/sec    0   3.28 MBytes
/tmp/h3.log:[  5]  18.00-19.00  sec  4.25 GBytes  36.5 Gbits/sec    0   3.37 MBytes
/tmp/h3.log:[  5]  19.00-20.00  sec  4.40 GBytes  37.7 Gbits/sec    2   2.36 MBytes
/tmp/h3.log:[  5]   0.00-20.00  sec  87.1 GBytes  37.4 Gbits/sec   14             sender
/tmp/h3.log:[  5]   0.00-20.01  sec  87.1 GBytes  37.4 Gbits/sec                  receiver
mininet>
```

Рисунок 3.4 – Передача пакетів на хост h6

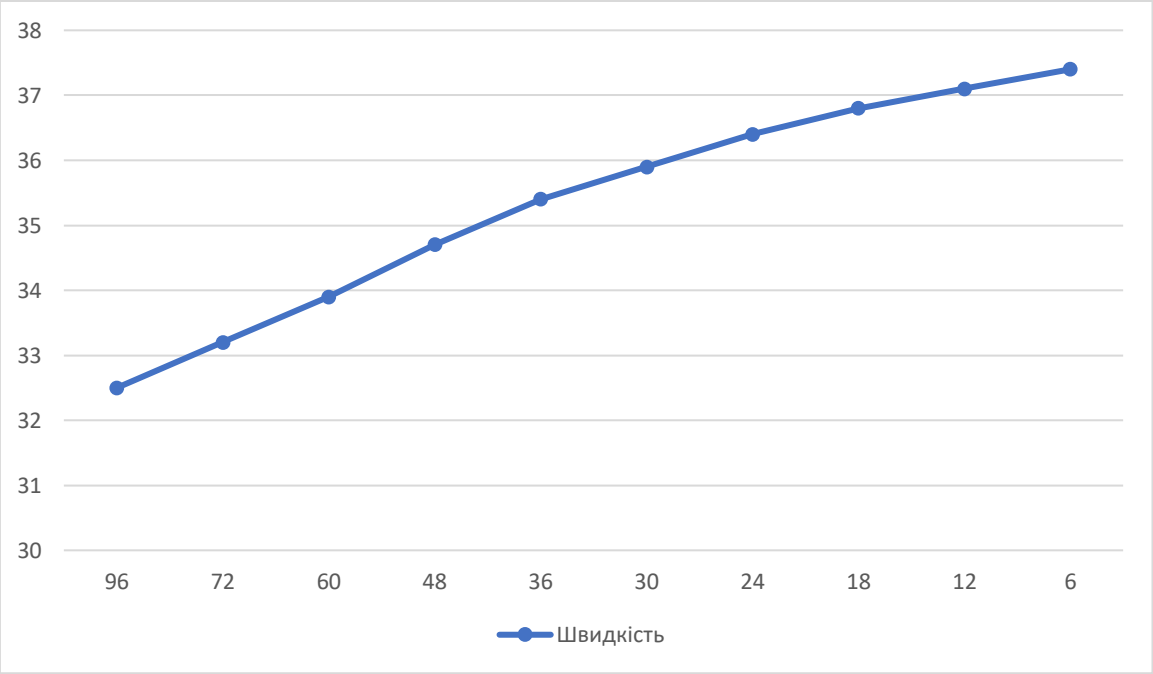
Під час передачі пакетів ми починаємо створювати правила на обмін пакетів між хостами. Тобто, передача пакетів з хосту h1 на h2 це одне правило а пересилка з h2 на h1, вже інше. Так як ці хости в нас підключені до всіх 3 свічів то це означає що ці правила розповсюджуються на свічі s1, s2 та s3. Таким чином тільки на пересилання пакетів між хостами потрібно 6 правил.

Таким чином по черзі ми додаємо правила на кожний свіч та замірюємо як буде знижуватись пропускна здатність мережі (Таблиця 3.2).

Таблиця 3.2 – Результати вимірювання трафіку

К-сть правил	Потоки	Середня швидкість	Затримка ms	Втрати %
6	1	37.4	0.15	0.00
12	2	36.9	0.17	0.00
18	3	36.3	0.18	0.00
24	4	35.7	0.20	0.01
30	5	35.0	0.22	0.01
36	6	34.4	0.25	0.01
48	8	33.2	0.28	0.02
60	10	32.1	0.30	0.03
72	12	31.2	0.32	0.04
96	16	30.1	0.35	0.06

Графік 3.1 – Визначення швидкості по відношенню до кількості команд



Дослідження показує що з збільшенням flow правил у мережі, зростає і невелике зниження середньої пропускної здатності та незначне збільшення

передачі пакетів. При зростанні кількості пакетів від 6 до 96 швидкість зменшилась на 7.3 Gbit/s, а затримка на 0.20 ms. Що до пакетів то їх втрати залишились на мінімальному рівні, всього лиш 0.06%.

Отримані дані підтверджують, що збільшення кількості команд може підвищити навантаження на комутатори та знизити пропускну здатність, але не на скільки критично щоб призвести до збоїв та зупинки роботи певних вузлів мережі.

3.1.3 Дослідження швидкості підключення хостів

IoT мережі складаються з величезної кількості різних пристроїв, такі як, сенсори, мобільні пристрої. Серед таких пристроїв є такі які можуть підключатись і відключатись час від часу. Серед їх списку є датчкик вологи, датчики руху, датчики температури. Роблять вони це для економії електроенергії. Принцип їх роботи такий, вони та підключаються до мережі, передають данні на сервер, відмикають радіомодуль та працюють далі.

Якщо контролер не зможе швидко виявити пристрій та дати йому доступ до мережі, то це може викликати затримки, втрати перших пакетів, збільшення часу реакції на якісь критичні події.

Саме тому метою цього дослідження є наскільки швидко контролер може виявити це пристрій, додати його до мережі та забезпечити передачу трафіку між іншими вузлами.

Для проведення тесту ми будемо використовувати вже існуючий хост який будем включати та виключати під час роботи мережі. Час появи хоста в мережі та затримку першої успішної відповіді ми будемо замірювати двома способами. Час коли хост з’явився в мережі та час коли контролер побачив цей хост та дав йому можливість відповідати.

Для початку запускаємо нашу топологію та виконуємо pingall щоб контролер побачив всі наші хости (Рисунок 2.5).

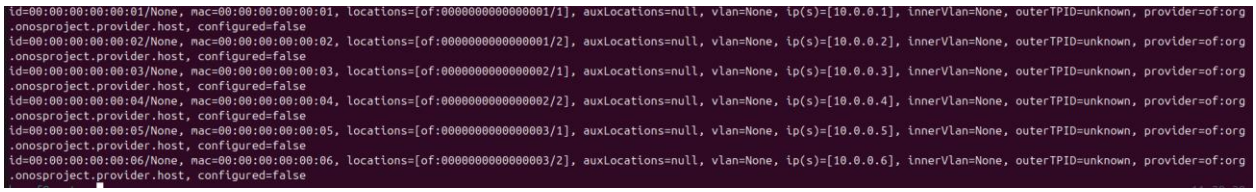


Рисунок 3.5 – Активні хости

Тепер за допомогою команди h6 ifconfig h6-eth0 down ми вимикаємо з’єднання хоста з свічем та виводимо час відключення (Рисунок 3.8).

В консолі Apache Karaf вводимо команду hosts і бачимо що в списку відсутній шостий хост (Рисунок 3.6).

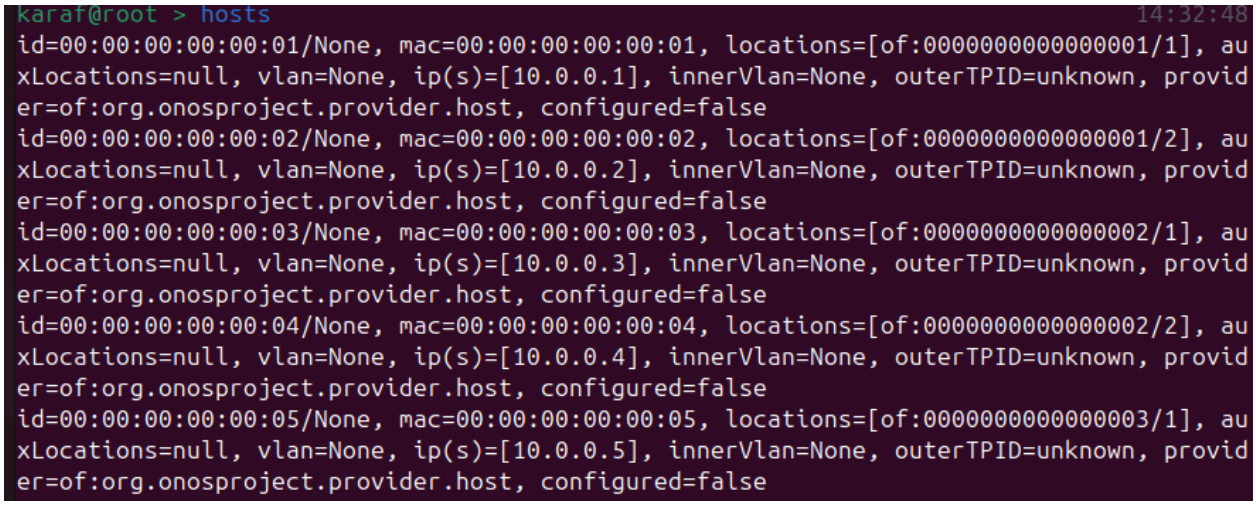


Рисунок 3.6 – Перевірка хостів

Далі все що нам потрібно це запустити 6 хост та подати команду ping. Ця команда генерує Packet-in до контролера, яка потрібна щоб повідомити контролеру що ось є хост і його потрібно додати в таблицю та видати flow-правила. Після цього можна вважати що пристрій був підключений та може обмінюватись трафіком з іншими вузлами мережі.

Після того як ми підняли наш хост та отримали правила ми можемо знову перевірити час (Рисунок 3.8).

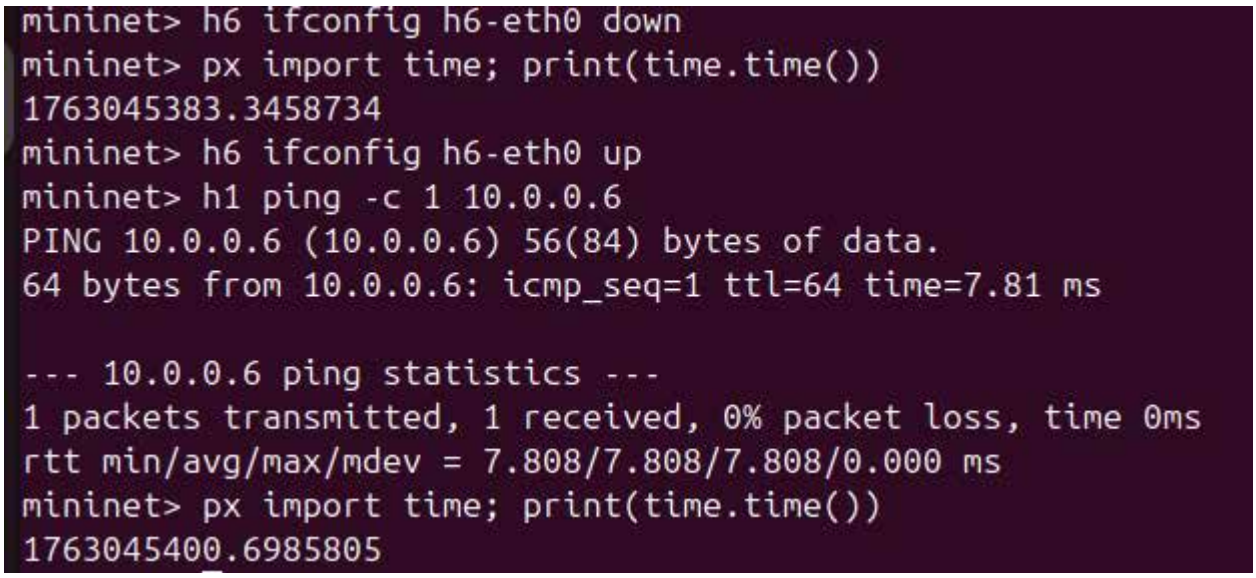


Рисунок 3.8 – Увімкнення хоста та отримання часу.

Таблиця 3.3 – Результати вимірювання затримки пошуку хоста

Подія	Час
Вимкнення хоста	1763045383 с 3458734 мс
Час після першого успішного пінгу	1763045400 с 6985805 мс
Затримка виявлення хоста	17.35 с
Затримка першого ICMP пакета	7.81 с

Як бачимо з рисунку 3.8 між вимкненням, увімкненням та виконанням команди ping пройшло всього лиш 17.35 секунд, та 7.81 після того як встановився маршрут.

ВИСНОВОК

В цій дипломній роботі було розроблено систему SDN для використання в IoT. Система є гнучкою в налаштуванні та бюджетною так як не потребує фізичного обладнання для побудови мережі і може бути використаною як стенд для навчання цим типам мереж.

У ході виконання роботи було вирішено наступні задачі:

1. Були проаналізовані перспективи її розвитку, актуальність системи, та перспективи розвитку.
2. Було створено робочу мережу SDN, а саме, два контролери які працюють в кластері, три свічі bmv2 які вміють перенаправляти трафік.
3. Результати роботи були протестовані, перевірено якість пересилання пакетів в мереж та її стресостійкість.

Для подальшого розвитку проекту можна запровадити ще більше метрик для збору телеметрії, додати балансування навантаження між контролерами, та штучний інтелект для ще кращого виявлення вразливостей та порушень стабільності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ubuntu Desktop [Електронний ресурс] – Режим доступу до ресурсу: https://ubuntu.com/desktop.

2. VMware [Електронний ресурс] – Режим доступу до ресурсу: https://www.vmware.com/.

3. ПОБУДОВА SDN-КОНТРОЛЕРА НА БАЗІ ВІДКРИТОЇ МЕРЕЖЕВОЇ ОПЕРАЦІЙНОЇ СИСТЕМИ ONOS

4. Stallings W. Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud. 2016. [Електронний ресурс] – Режим доступу до ресурсу: https://www.oreilly.com/library/view/foundationsofmodern/9780134175478/copyright.xhtml.

5. ONOS Project [Електронний ресурс] – Режим доступу до ресурсу: https://opennetworking.org/onos/.

6. Mininet [Електронний ресурс] – Режим доступу до ресурсу: https://mininet.org/

7. Towards Fault Tolerance Management Systems in SDN. M. Mbodila, 2022 [Стаття] – Режим доступу до ресурсу: https://scispace.com/pdf/towards-fault-tolerance-management-systems-in-sdn-2mcns6vy.pdf.

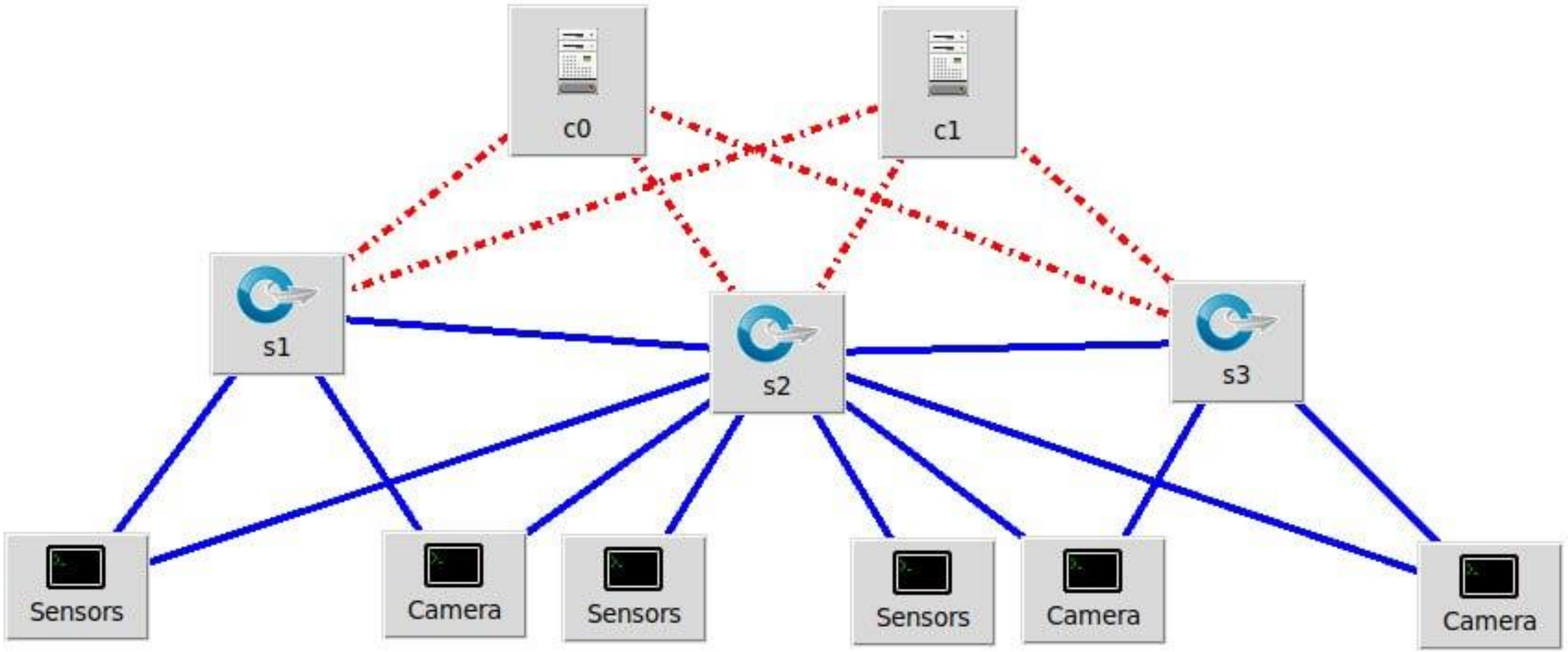
8. A new approach in fault tolerance in control level of SDN. Y. Narimani & Es. Zeinali, 2025. [Стаття] – Режим доступу до ресурсу: https://ijnaa.semnan.ac.ir/article_9028_a3b555ea1eb75a92e2b3dbd46a22f2d5.pdf.

9. ONOS: Towards an Open, Distributed SDN OS. 2014. [Стаття] – Режим доступу до ресурсу: https://scispace.com/pdf/onos-towards-an-open-distributed-sdn-os-25228im0dz.pdf.

10. Fault Tolerance in SDN Data Plane Considering Network and Application Based Metrics [Стаття] – Режим доступу до ресурсу: https://arxiv.org/pdf/1912.11849.

ДОДАТОК А

Топологія мережі візуалізована в Miniedit



ДОДАТОК Б

Код функціонування свіча bmv2

```
    }
  }

  control MyVerifyChecksum(inout headers_t hdr, inout metadata_t
meta) { apply { } }

  control MyIngress(inout headers_t hdr,
                    inout metadata_t meta,
                    inout standard_metadata_t stdmeta) {

    action action_forward(egressSpec_t port) { stdmeta.egress_spec
= port; }

    table forward {
      key = { hdr.ethernet.dstAddr : exact; }
      actions = { action_forward; NoAction; }
      size = 1024;
      default_action = NoAction();
    }

    apply { forward.apply(); }
  }

  control MyEgress(inout headers_t hdr,
                   inout metadata_t meta,
                   inout standard_metadata_t stdmeta) { apply { } }

  control MyComputeChecksum(inout headers_t hdr, inout metadata_t
meta) { apply { } }

  control MyDeparser(packet_out packet, in headers_t hdr) {
    apply {
      packet.emit(hdr.ethernet);
      packet.emit(hdr.ipv4);
```

```
    }
}
V1Switch(MyParser(), MyVerifyChecksum(), MyIngress(),
    MyEgress(), MyComputeChecksum(), MyDeparser()) main;
```

Код скрипту для створення mininet топології

```
from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel

ONOS_IP = '192.168.150.129'
ONOS_PORT = 6653

def main():
    setLogLevel('info')
    net = Mininet(controller=None, switch=OVSSwitch,
link=TCLink, build=False)

    c0 = net.addController('c0',
controller=RemoteController, ip=ONOS_IP, port=ONOS_PORT)

    s1 = net.addSwitch('s1', protocols='OpenFlow13',
failMode='secure')
    s2 = net.addSwitch('s2', protocols='OpenFlow13',
failMode='secure')
    s3 = net.addSwitch('s3', protocols='OpenFlow13',
failMode='secure')

    h1 = net.addHost('h1', ip='10.0.0.1/24')
    h2 = net.addHost('h2', ip='10.0.0.2/24')
    h3 = net.addHost('h3', ip='10.0.0.3/24')
    h4 = net.addHost('h4', ip='10.0.0.4/24')
```

```
        h5 = net.addHost('h5', ip='10.0.0.5/24')
        h6 = net.addHost('h6', ip='10.0.0.6/24')

        net.addLink(h1, s1); net.addLink(h2, s1)
        net.addLink(h3, s2); net.addLink(h4, s2)
        net.addLink(h5, s3); net.addLink(h6, s3)

        net.addLink(s1, s2)
        net.addLink(s2, s3)

        net.build()
        c0.start()
        s1.start([c0]); s2.start([c0]); s3.start([c0])

        for sw in (s1, s2, s3):
            sw.cmd(f'ovs-vsctl set bridge {sw.name}
protocols=OpenFlow13')
            sw.cmd(f'ovs-vsctl set-fail-mode {sw.name} secure')


        CLI(net)
        net.stop()

    if __name__ == '__main__':
        main()
```