RIGA TECHNICAL UNIVERSITY

MASTER THESIS

RIGA TECHNICAL UNIVERSITY

Faculty of Electronics and Telecommunications

Department of Electronics and Telecommunications

Wai Yan Kyaw

Telecommunication Technologies and Networks Management

211AEM007

P4-Enabled Programmable Networking for Enhancing IoT Communication Efficiency

MASTER THESIS

Scientific advisor:

Supervisor

Elans Grabs

Abstract

The primary goal of this thesis is to conduct a comprehensive study and analysis of P4 technology to assess its integration with IoT devices within SDN environments on both wired and wireless environments. With the Internet of Things (IoT) experiencing rapid expansion, network sizes are expected to grow exponentially. Consequently, the management of such networks will become increasingly complex. The integration with IoT devices into SDN networks can offer substantial benefits by providing the flexibility and programmability required for the effective management and administration of each network component.

Keywords: IoT; SDN; P4 language; Datapaths, Mininet, Mininet-WiFi

Acknowledgements

I would like to express my heartfelt gratitude to my professors, Juan Pedro Muñoz Gea and Elans Grabs for their unwavering support and guidance throughout this thesis. Their patience, encouragement, and invaluable advice have been instrumental in my journey to completion.

I am also deeply grateful to my parents, for their continuous support and belief in my abilities. Their love and encouragement have provided me with the strength to persevere and achieve my goals.

Contents

1 Introduction **10**

2 State of the art technologies **12**

2.1 SDN technology 12

2.1.1 Architecture 12

2.1.2 OpenFlow 13

2.2 Internet of Things (IoT) 13

2.2.1 IoT architecture 14

2.2.2 Topologies 15

2.3 P4 technology 16

2.3.1 P4 architecture 17

2.3.2 Development and deployment process 18

2.4 Mininet 19

2.4.1 Mininet architecture 20

2.4.2 Linux namespaces 21

2.4.3 Linux cgroups 21

2.4.4 Linux virtual ethernet pairs (veth pairs) 21

2.4.5 Mininet CLI 21

2.4.6 Mininet-WiFi 22

2.4.7 Mininet-IoT 23

3 Design and Analysis **24**

3.1 Basic functionalities 24

3.2 Wired test platforms 25

3.3 Wireless test platforms 26
3.4 Contribution on gitHub 27

4 Development and Evaluation **28**
4.1 P4 use cases on wired environment 28
4.1.1 Drop 29
4.1.2 Pass 37
4.1.3 Echo Server 37
4.1.4 L3 Forwarding 47
4.1.5 Broadcast 55
4.2 P4 use cases on wireless environment 64
4.2.1 BMv2 integration in Mininet-WiFi 65
4.2.2 Drop 68
4.2.3 Pass 74
4.2.4 Echo Server 74
4.2.5 L3 Forwarding 78
4.2.6 Broadcast 85
Conclusion 90
References 90

A Appendix **95**
1. Specification of physical machine 95
1. Specifications of virtual machine and installation P4 96
1.1 P4 installation in wired media 96
1.2 P4 installation in wireless media 97

List of Figures

2.1 SDN architecture[1] 13
2.2 IoT architecture[2] 14
2.3 IoT topologies 15
2.4 Difference usages of data planes programmability[3] 17
2.5 P4 architecture[3] 18
2.6 The process of P4 implementation[3] 19
2.7 Mininet topology[4] 20
2.8 Configurable data plane[4] 20
2.9 Mininet CLI 22

3.1 Loading P4 program on the BMv2 25
3.2 IEEE 802 workgroups 26

4.1 P4 program compilation process 30
4.2 An overview of a single switch's configuration (BMv2) 30
4.3 Network Topology (Drop) 31
4.4 Drop topology 32
4.5 IP addressing of hosts 33
4.6 Testing ping between h1 and h4 34
4.7 Capturing packets between h1 and h4 35
4.8 Testing ping between h2 and h3 36
4.9 Capturing packets between h2 and h3 36
4.10 Network topology (echo server) 42
4.11 Topology of echo server 44
4.12 IP addresses of hosts (h1 and h4) 45

4.13 Testing echo ping 45
4.14 Captured traffic in BMv2-switch-1 (s1-eth1) 46
4.15 Operation of the tables in P4 [5] 47
4.16 Network topology (L3 Forwarding) 50
4.17 L3 forwarding topology 52
4.18 IP addresses of hosts 53
4.19 Testing ping result between hosts 54
4.20 Captured ICMP in s1 (s1-eth3) 54
4.21 No captured packet in s3 (s3-eth3) 55
4.22 Testing ping result between hosts 55
4.23 Captured ICMP in s1 (s1-eth4) 56
4.24 No captured packet in s3 (s2-eth3) 56
4.25 Network topology (broadcast) 59
4.26 Broadcast topology 61
4.27 IP addresses of hosts 62
4.28 Captured traffic in tcpdump 62
4.29 Captured ARP traffic in s1 (s1-eth3) 63
4.30 Captured ARP traffic in s2 (s2-eth2) 63
4.31 Captured ICMP traffic in s2 (s2-eth3) 63
4.32 UML of the entry point of the BMv2 - Mininet Interface 66
4.33 UML interface BMv2 - Mininet 66
4.34 Mininet-WiFi architecture [6] 67
4.35 Linux Wireless Subsystem Architecture[7] 68
4.36 Drop topology 70
4.37 Ip addressing of hosts) 71
4.38 Testing ping between sta1 and h1 72
4.39 Capturing packets between sta1 and h1 72
4.40 Testing ping between sta1 and sta2 73
4.41 Capturing packets between sta1 and sta2 73
4.42 Echo server topology 76

4.43 IP addressing of sta1 and h1 76
4.44 Testing echo ping 77
4.45 Capturing packets in P4RuntimeAP (ap1) 77
4.46 Topology (L3 Forwarding) 80
4.47 IP addressing of sta1, sta2 and sta3 81
4.48 Test ping result between hosts 82
4.49 Capturing ICMP in P4RuntimeAP (ap1) 82
4.50 No captured packet in P4RntimeAP (ap3) 83
4.51 Test ping result between hosts 83
4.52 No captured packet in P4RntimeAP (ap3) 83
4.53 Capturing ICMP in P4RuntimeAP (ap1) 84
4.54 Topology (Broadcast) 86
4.55 IP addressing of sta1, sta2 and sta3 87
4.56 Test ping result between hosts 88

A.1 Specification of physical machine 95
A.2 Specification of virtual machine 96
A.3 Specification of virtual machine 97

CHAPTER 1

Introduction

The convergence of Software-Defined Networking (SDN) and Internet of Things (IoT) technologies create a compelling space with both exciting possibilities and complex issues in network management for both academic research and industry. In today's world, data flows grow rapidly due to the widespread connectivity of devices to the Internet, particularly IoT devices. Managing data from IoT network within core network has become extremely complex. As IoT devices multiply and network sizes grow exponentially, traditional network management approaches are no longer sufficient and managing such networks will become increasingly complex. Hence, it is important to leverage cutting-edge technologies, for example, Software-Defined Networking to address these challenges and effectively manage such a continuously expanding and complex network. Moreover, efficiently managing the data flow within IoT networks plays a crucial in today's technological environment.

This project focuses on a comprehensive study and analysis of the P4(Programming Protocol-Independent Packet Processors) technology, with the primary objective of understanding its role in the integration of IoT devices into Software Defined Network (SDN) environments. P4 offers a unique opportunity to address the intricacies of managing IoT networks within SDN frameworks. The combination of SDN and P4 in IoT scenarios offers the potential for more adaptability, scalability, and precise control. Being able to program and change network devices in real-time allows for a more efficient and flexible approach to managing IoT devices.

In the forthcoming chapters, we delve into the state of the art technologies, design and analysis, development and evaluation of IoT using P4. This study seeks to unlock the potential of SDN technologies, coupled with P4 programming, in the context of the rapidly

growing IoT environments. As IoT devices continue to proliferate, the management and administration of these devices within SDN environments become paramount. This project serves as an initial step toward this endeavor, clearing the path for creative solutions to address emerging issues in network management and the integration of IoT devices.

Let's embark on this adventure of exploration, learning, and creativity as we explore the fusion of SDN and IoT, empowered by the capabilities of P4 programming.

CHAPTER 2

State of the art technologies

This chapter's objective is to provide the core concepts relevant to the project, as well as an overview of the primary tools that are utilized during development. The purpose of this chapter is to construct a robust theoretical framework that will enable optimal analysis and design before entering the development stage.

2.1 SDN technology

Software Defined Network (SDN) is a transformative technology in the field of networking, offering dynamic and centralized control over network infrastructure. It decouples the control plane from the data plane, enabling network administrators to programmatically configure and manage network resources[8]. SDN's key advantage lies in its flexibility, allowing for efficient resource utilization, rapid adaptation to changing network conditions, and the ability to provide differentiated services. By abstracting network control and simplifying management through a centralized controller, SDN facilitates automation, improved scalability, and enhanced security in modern networking environments.

2.1.1 Architecture

SDN architecture typically consists of three key components: the application layer, the control layer, and the data plane. The application layer hosts network applications, including SDN controllers and network management tools. The control layer, represented by SDN controllers, serves as the decision-making center, translating application requests into network policies and configurations. The data plane, comprising network devices like switches and routers, forwards traffic based on the control plane's instructions. Communication

between these layers occurs through standardized protocols, such as OpenFlow (see Figure [2.1]). This architecture allows for flexibility and adaptability in network management, making SDN a vital technology in today's complex and dynamic networking landscapes.

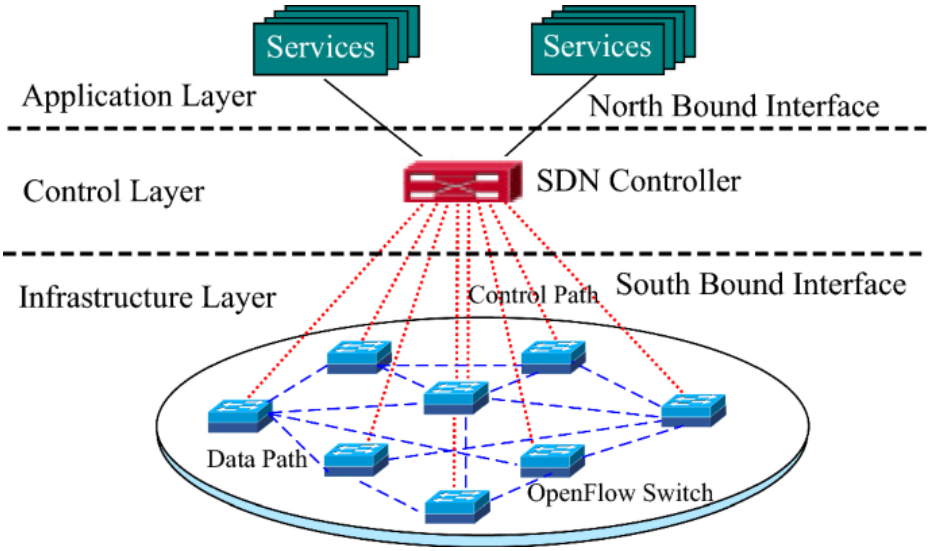


Figure 2.1: SDN architecture[1]

2.1.2 OpenFlow

OpenFlow is a crucial networking protocol used in Software Defined Network (SDN). It simplifies network management and control by separating the control plane from the data plane, enabling centralized control over network devices. With OpenFlow, network administrators can define how packets are forwarded through network switches, making it a key technology for building agile and programmable networks.

2.2 Internet of Things (IoT)

IoT, or the Internet of Things, is a game-changing technology that is rapidly reshaping the digital landscape. It encompasses a vast ecosystem of interconnected devices, sensors, and objects, all capable of gathering and transmitting data over the internet[9]. At the heart of IoT's power is its ability to bridge the physical and digital realms, enabling the seamless exchange of information between everyday objects and the virtual world. IoT applications are incredibly diverse, ranging from smart homes and industrial automation to healthcare

and agriculture. This technology offers the potential to enhance efficiency, productivity, and convenience across numerous domains.

2.2.1 IoT architecture

The IoT must possess the capacity to connect an immense number of diverse objects through the Internet, ranging from billions to potentially trillions. Consequently, a highly adaptable layered framework becomes essential. Two fundamental models are prevalent in this context: one is the three-layer architecture, encompassing the Application, Network, and Perception Layers, and the other is the five-layer architecture, involving objects, object abstraction, service management, application, and business as follow.

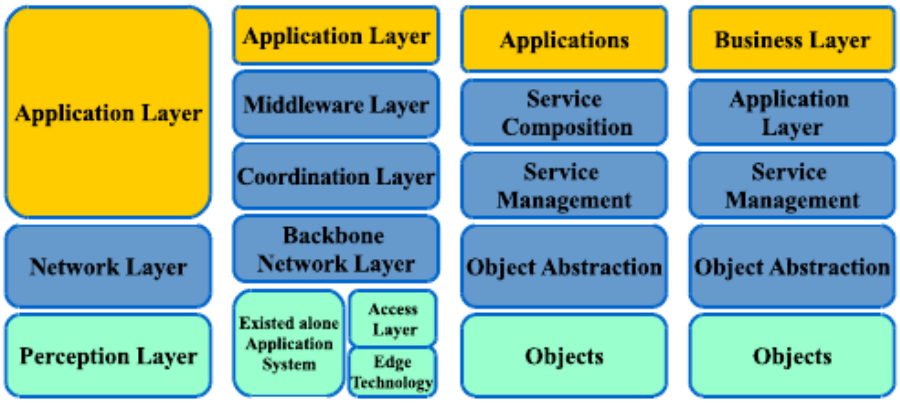


Figure 2.2: IoT architecture[2]

It is said that the five-layer architecture is the most suitable model for IoT application. In the following, we provide a concise explanation of the five-layer architecture.

Objects Layer: The initial layer, known as the Objects or perception layer, represents the physical sensors within the IoT system responsible for collecting data. This layer encompasses various sensors and actuators used for tasks as monitoring location, temperature, motion, and other parameters.

Object Abstraction Layer: This layer secures data transferring from the Objects layer to the Service Management layer using a variety of technologies, including WiFi and Bluetooth Low Energy etc.

Service management Layer: The Service Management or Middleware layer links services with requester using addresses and names. It empowers IoT developers to interact with diverse objects regardless of hardware specifics and handles data processing, decision-making, and service delivery over network protocols.

Application Layer: This layer offers requested services and delivers high-quality smart services across various verticals, including smart environment to customer needs.

Business Layer: This layer manages all IoT system operations and services, including creating business models, graphs, and flowcharts using data from the application layer. It's responsible for designing, analyzing, implementing, evaluating, monitoring, and enhancing IoT system components.

2.2.2 Topologies

In the context of IoT (Internet of Things) networks, several topologies can be used to interconnect devices and sensors. The choice of topology depends on the specific requirements of the application such as star, mesh, tree, bus, ring, line, cellular and cluster in Figure [2.3].

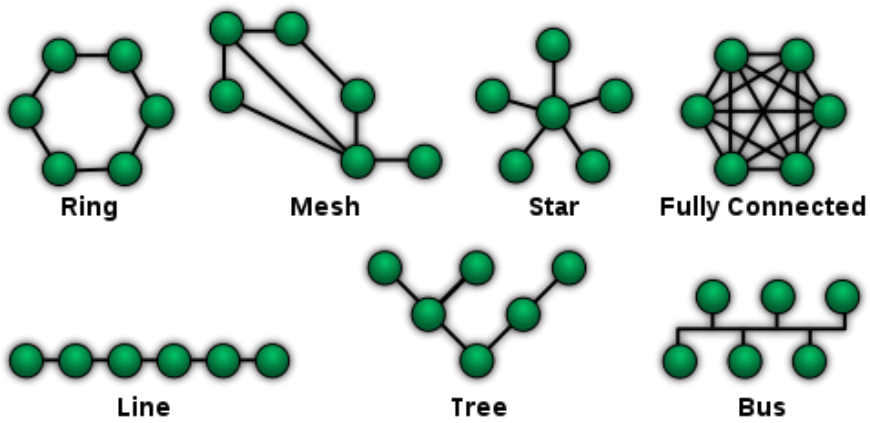


Figure 2.3: IoT topologies

Among different access technologies are used for IoT devices, star, mesh, and peer-to-peer topology schemes are mainly used. Star topology is common with a central base station for long distance and short distance technologies such as cellular, LPWA, and Bluetooth. Medium-range tech often uses star, peer-to-peer, or mesh, allowing devices to communicate directly or through intermediaries, creating complex networks. For instance, indoor Wi-Fi uses a star network for Access Point (AP)s, while outdoor Wi-Fi relies on mesh for wider coverage.

2.3 P4 technology

Programmable data planes allow users to define their own data plane algorithms for network devices including appropriate data plane Application Programming Interfaces (APIs) which may be leveraged by user-defined Software Defined Network (SDN) control[3]. This adaptability is crucial for customizing networks, whether it is for specific commercial devices in 5G or data centers, or for rapid prototyping in industry and academia. The programming language known as Programming Protocol-independent Packet Processor (P4) has risen as the prevailing abstraction, language, and framework for data plane programming. In addition, it is developed and standardized by an open community[10] with supports across a variety of software and hardware platforms[1]. In the following Figure [2.4], we can observe the varying uses of data plane programmability, from traditional networking to full network programmability through data plane programming.

[1] http://p4.org/

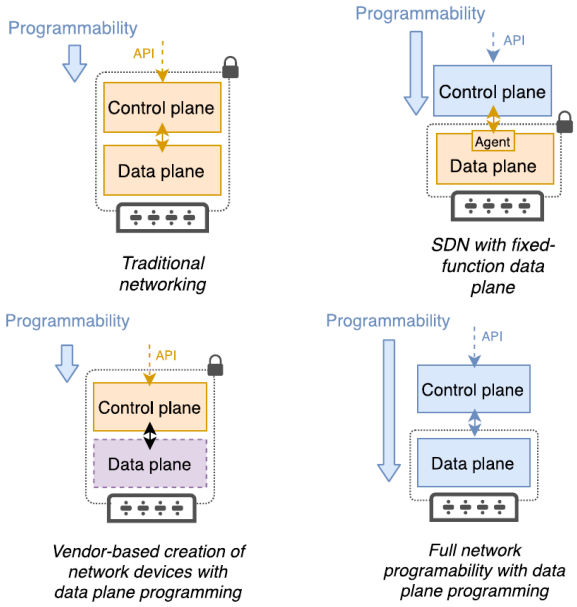


Figure 2.4: Difference usages of data planes programmability[3]

At present, the most commonly adopted approach for data plane programming is the utilization of Protocol-Independent Packet Processors (P4). This concept, programming language, and abstraction have gained extensive use. It was initially introduced in a research paper in 2014 by Bosshart and colleagues[11].

2.3.1 P4 architecture

Before we mention about P4 (Programming Protocol-independent Packet Processor), we need to mention about Protocol-Independent Switching Architecture (PISA). It is based on the concept of a programmable match-action pipeline that well matches modern switching hardware. PISA consists of a programmable parser, a programmable deparser, and a programmable match-action pipeline in between consisting of multiple stages[3]. PISA is a powerful and flexible network switch architecture designed to support various network protocols, making it a versatile tool for network administrators and engineers. Unlike traditional switches that are fixed to specific network protocols, PISA provides a programmable and protocol-independent framework for packet processing. Besides, it enables innovation and adaptability in network design.

Figure [2.5] shows P4 architecture based PISA switch in which PISA allows users to

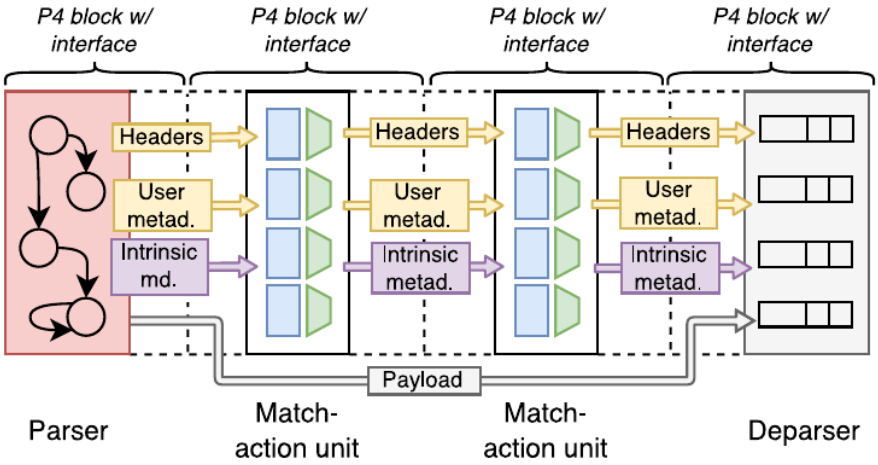


Figure 2.5: P4 architecture[3]

define and implement their own packet processing logic, making it excellent platform for leveraging P4-based data plane programming. In this case, we need to point out what P4 is. Programming Protocol-independent Packet Processor (P4) is a domain-specific language which is designed for defining how network packets should be processed in data plane devices, such as switches and routers. P4 is used in SDN to enable the flexible and programmable configuration of network devices.

In addition, P4 operates alongside SDN control protocols like OpenFlow and can address some of the limitations associated with the OpenFlow protocol in contemporary networks. In OpenFlow networks, certain fundamental safeguards, such as protection against ARP poisoning, DHCP snooping, rate limiting for broadcast/multicast traffic, and MAC address restrictions for ports, rely on the controller's decisions. This reliance on the controller for these basic protections can introduce Denial of Service vulnerabilities into the network. P4 technology offers a solution to this by extending programmable rules to the data plane, enabling the implementation of these essential protections directly within the network devices.

2.3.2 Development and deployment process

First, we need to mention about two versions of P4-14 and P4-16. P4-16 represents the latest evolution of the P4 programming, offering an enhanced framework for defining

protocol-independent packet processing logic, making it a pivotal component of modern software-defined networking. In contrast, P4-14, an earlier version of the language, laid the groundwork for P4 by introducing the concept of abstracting the data plane and enabling programmable network device customization.

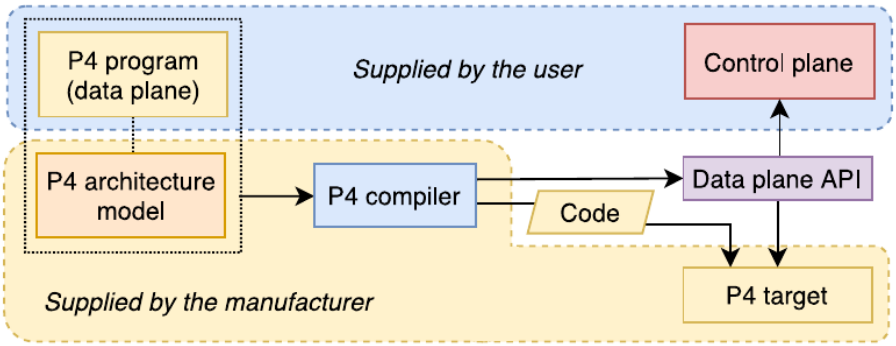


Figure 2.6: The process of P4 implementation[3]

Figure [2.6] presents an overview of how P4 programs are deployed and implemented. P4 targets are nodes that can be either software or specialized hardware, containing packet processing pipelines with P4-programmable and fixed-function components. The configuration of these pipelines varies according to the specific target and is detailed in a P4 architectural model supplied by the vendor. Users supply P4 programs to define algorithms for the P4-programmable components, and P4 compilers from manufacturers translate these programs into target-specific code for execution on the P4 target.

2.4 Mininet

Mininet serves as a network emulator and, more precisely, a network emulation orchestration system. It operates by creating a virtual network on a single Linux kernel, encompassing a mix of end-hosts, switches, routers, and links[12]. This lightweight virtualization approach transforms a single system into a complete network, allowing it to share the Linux kernel, usages, and code under the same operation system. Essentially, every host in Mininet emulates the behavior of an actual physical device, including the capability to provide remote access and run a variety of programs. The network connections within Mininet closely resemble actual Ethernet interfaces, complete with defined link speeds and

delays. Packets are handled as if they were traversing through a genuine Ethernet switch, router, or middlebox, complete with specified queuing. In Fig [2.7], it illustrates a Mininet topology setup with two hosts directly linked to a switch.

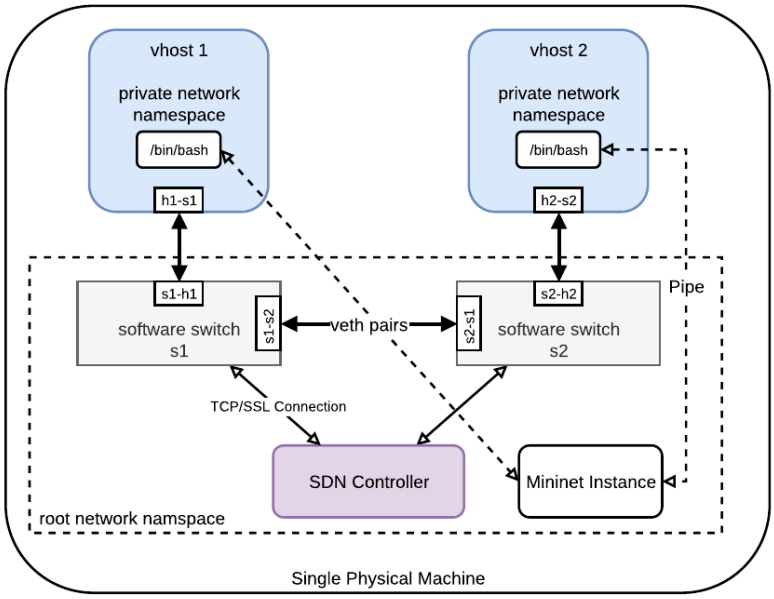


Figure 2.7: Mininet topology[4]

2.4.1 Mininet architecture

Mininet relies on use of a variety of embedded isolation and virtualization functions offered by the Linux core. Its architecture is deeply integrated with the Linux kernel, which forms the foundation for its emulation capabilities.

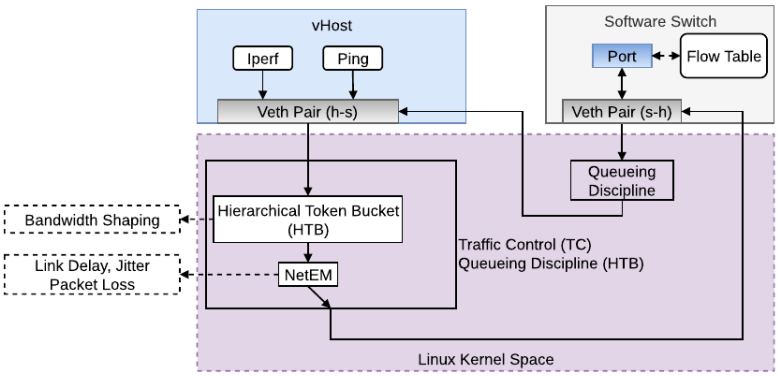


Figure 2.8: Configurable data plane[4]

Figure [2.8] shows that Mininet is using tools under Linux kernel for enabling SDN-based simulations with a fully customizable data plane. In addition, we need to know about software tools such as veth, cgroups and namespaces that are integrated with Mininet to provide a fully programmable emulation.

2.4.2 Linux namespaces

Namespaces in Mininet are a fundamental features that enables the isolation and virtualization of network resources. They provide separate network environments within a shared Linux kernel, allowing for the creation of distinct network entities. Namespaces facilitate the establishment of custom network topologies and are essential for conducting network experiments and tests in a controlled and reproducible manner.

2.4.3 Linux cgroups

Linux Control Groups (cgroups) provide a means of controlling and managing resource allocation for network emulation. This feature allows users to allocate and restrict system resources as CPU, memory, and network bandwidth to various network components, ensuring controlled and predictable behavior in network experiments and tests.

2.4.4 Linux virtual ethernet pairs (veth pairs)

In Mininet, veth pairs serve the purpose of linking separate hosts to a software switch. Veth pairs are primarily provided by the Linux kernel for establishing connections between network namespaces. A veth pair consists of two distinct interfaces, each interface operating as a standard Ethernet interface. Packets sent from one interface are immediately forwarded to the other interface using veth.

2.4.5 Mininet CLI

Mininet features a Command Line Interface (CLI) that can be used to interact with a network. It offers a range of handy commands, including the capability to open xterm windows and execute commands on specific nodes within your network. To activate the

CLI for a network, you simply need to pass the network object to the CLI() constructor[13]:

Code Importing Mininet CLI in python

```
from mininet.cli import CLI
# launch the Mininet Command Line Interface (CLI)
CLI(net)
```

Once the CLI start, we can use some commands to check the current network situation such as **net** for viewing the network topology, **pingall** for connectivity between hosts, and link status of hosts.

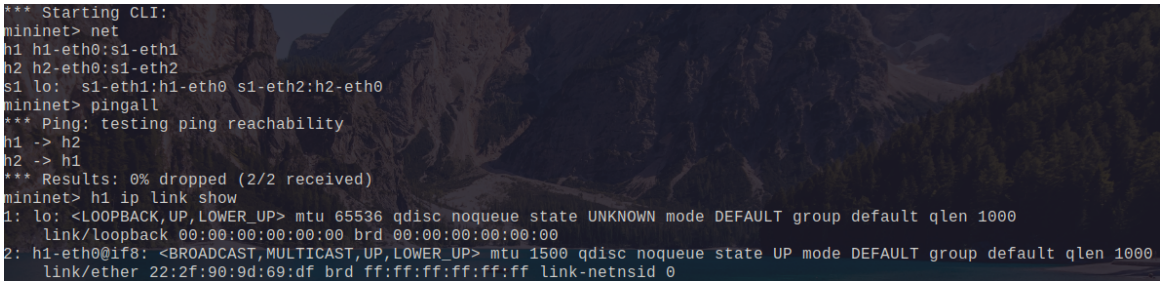


Figure 2.9: Mininet CLI

2.4.6 Mininet-WiFi

Mininet-WiFi is an emulator specifically designed to simulate wireless networks conforming to the IEEE 802.11 standard. It originated as a fork of Mininet, sharing its core principles of lightweight virtualization through Namespaces and Veths. As a result, all Mininet scripts are compatible with Mininet-WiFi. [6].

The wireless functionalities in Mininet-WiFi are an extension built upon the existing framework of Mininet. To achieve this, the developers leveraged the wireless subsystem within the Linux Kernel and the mac80211-hwsim module for emulating interfaces and creating simulated wireless environments. For a more comprehensive understanding of this tool, it is advisable to refer to **subsection 4.2.1**, where an in-depth analysis is provided on the additional class hierarchies introduced in Mininet-WiFi, its internal operations, and its interaction with the kernel module for generating wireless scenarios.

2.4.7 Mininet-IoT

The Mininet-IoT tool is a network emulator optimized for low-capacity networks operating within the IEEE 802.15.4 standard and utilizing the 6LoWPAN adaptation layer[14]. Derived from Mininet-WiFi, which itself originated from Mininet, Mininet-IoT essentially adopts the same lightweight virtualization methods as Mininet. With its roots in both Mininet-WiFi and Mininet, Mininet-IoT retains compatibility with all scripts used for deploying wired and WiFi topologies.

The main difference between Mininet-IoT and Mininet-WiFi is the module used to simulate interfaces and wireless environments. While Mininet-WiFi utilizes the mac80211-hwsim module, Mininet-IoT employs the mac802154-hwsim kernel module (requiring a kernel version higher than 4.18.x). Despite this distinction, both tools manage nodes, interfaces, and links in the same way as Mininet-WiFi. To streamline maintenance and provide users with low-capacity links in wireless setups, Ramon Fontes[15], the primary developer, created a versatile class to handle Kernel modules in Mininet-WiFi and migrated the entire Mininet-IoT project to Mininet-WiFi, consolidating the core functionality into a single project.

CHAPTER 3

Design and Analysis

This chapter organizes basic functionalities of P4 by using different use cases in order to establish the data handling processes for IoT devices. This evaluation is aimed at realizing the objectives outlined in the Introduction (Chapter 1).

3.1 Basic functionalities

P4 technology provides the opportunity to employ a hardware-independent language for specifying the data path in IoT networks. To meet the objectives outlined in the Introduction, a set of fundamental functionalities for IoT networks has been chosen for proposal. These functionalities will be consolidated within use cases that are to be developed using P4 technology.The resulting use cases are as follows:‘

Drop	:	In this use case, we want to see if it is possible to drop packets.
Pass	:	In this use case, we want to check if it is possible to pass the packets, without affecting the data plane programmed.
Echo Server	:	In this scenario, our main focus is to assess how effectively both technologies allow us to filter and parse packets based on their header by creating a server that will respond to all incoming Echo requests.
L3 Forwarding	:	This use case will be a more sophisticated function that processes packets according to a routing table.
Broadcast	:	In this scenario, we want send the packets we receive on our network to get responses from the device we are trying to reach[16].

In the context of P4 programs, an additional component required for testing the developed functionality is the Behavioral Model version 2 (BMv2), a software switch designed specifically for P4 code testing. As illustrated in Figure [3.1], after compiling the P4 program, you will obtain a JSON file[17]. This file informs BMv2 about the specific datapath it needs to implement.

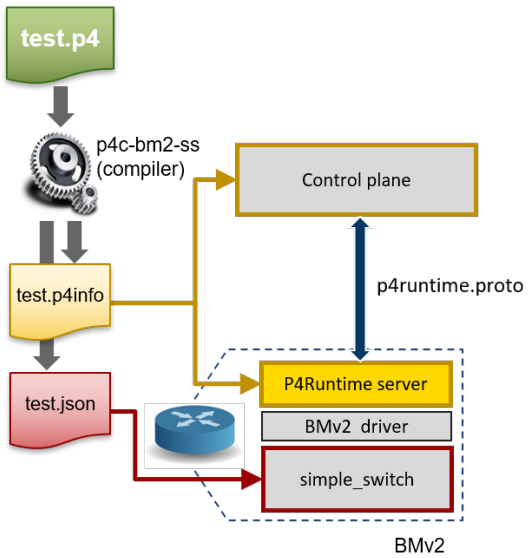


Figure 3.1: Loading P4 program on the BMv2

Considering the integration with IoT devices into SDN environment using P4 in the introduction (Chapter-1), use cases mention above will be created for the development. The next step involves selecting the platforms for evaluating the fundamental functionalities developed. Figure [3.1] shows how P4 program loads on the BMv2 switch.

3.2 Wired test platforms

To evaluate the developments of P4, the IEEE 802.3 (Ethernet) standard will be used. Regarding the platform selected for assessing P4 programs, we will employ Mininet. As previously discussed in the literature review (Chapter 2), Mininet serves as a valuable tool for simulating SDN network topologies. The choice of Mininet is driven by the fact that the P4lang team has developed an interface with BMv2, enabling seamless integration of the reference soft-switch for P4 code testing as an additional node type within the Mininet environment.

3.3 Wireless test platforms

To assess the advancements of P4 within wireless environments, the IEEE 802.11 (WiFi) standard serves as the initial focus. This choice is based on the current availability of more tools and developments compared to the IEEE 802.15.4 standard. The IEEE 802.11 standard is widely recognized as one of the most utilized wireless network standards globally and undergoes continual updates from the IEEE to enhance performance and accommodate new functionalities [30]. This particular working group is among the most dynamic within the IEEE's efforts to standardize local area networks (refer to Figure 3.2).

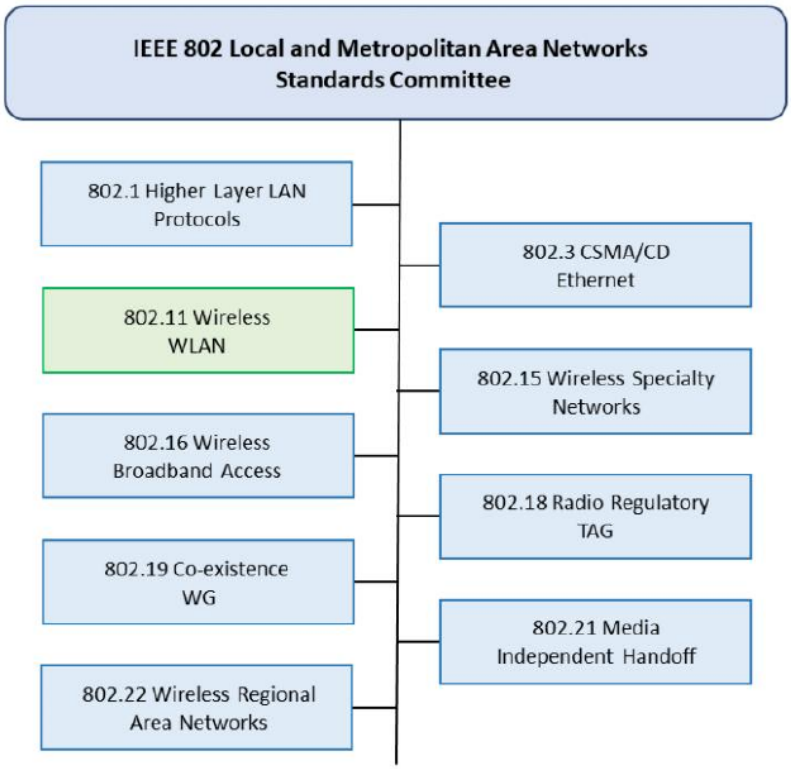


Figure 3.2: IEEE 802 workgroups

The success of the project heavily relies on integrating components effectively. Specifically, the assessment of P4 programs within wireless settings hinges on the ability to incorporate BMv2 into the Mininet-WiFi framework. The proposed steps for achieving this integration are as follows:

- Examine and analyze the interface generated by the p4Lang team, connecting BMv2 with Mininet.

- Examine the Linux Wireless subsystem and analyze the internal workings of Mininet-WiFi. This aspect holds significant importance as it is crucial to comprehend the underlying mechanisms and operations of Mininet-WiFi at a granular level.
- Creating the integration involves enabling BMv2 to handle the interfaces generated by Mininet-WiFi within Access Point (AP) nodes. Moreover, BMv2 should be capable of operating within a Network Namespace.

This requirement was essential to prevent bypass issues within Ad-Hoc networks. Running all instances of BMv2 within the same Network Namespace was unfeasible because specific packets would directly reach the final BMv2 without adhering to the designated path outlined in the emulated network.

3.4 Contribution on gitHub

GitHub is a comprehensive platform for version control, code hosting, collaboration, and project management. It supports development teams in tracking changes, managing code repositories, and working together on software projects. GitHub offers features for organizing, automating, and documenting projects, and it has a thriving community of developers[18]. For this project, we will be used GIT version **2.34.1** and GitHub as a platform to host our project code[1].

Link to the P4-Project repository : https://github.com/Waiyankyawbelu/P4-Project

The repository's documentation is primarily conveyed through README.md files found in all directories, providing information for test replication and software utilization. It also includes theoretical explanations to aid visitors in understanding the tests' nature, expectations, and potential conclusions. The repository serves a dual purpose, storing code and disseminating project content.

[1] https://github.com/Waiyankyawbelu

CHAPTER 4

Development and Evaluation

4.1 P4 use cases on wired environment

This subsection contains all the P4 technology use cases, which are available in the GitHub repository[1]. You can find the installation instructions and dependencies in the appendix. The P4 use cases are categorized into three parts to enhance clarity and organization for readers.

Introduction : This subsection will cover additional theoretical explanations, personal insights on the use case, and remarks regarding the P4 code that was developed.

Compilation and implementation : In this subsection, we will guide the reader on compiling the P4 program and setting up the environment. Notably, both of these processes are combined within a single Makefile, making the replication of use cases a smoother experience.

Evaluation : Finally, we will assess the performance of the use case by utilizing the Mininet CLI.

To assist readers in tracking the progress of the P4 use cases on wired environment, we have provided Table[4.1]. It details the specific location of each use case within the **P4-Project** repository.

[1] https://github.com/Waiyankyawbelu/P4-Project

Table 4.1: Summary documentation on P4 use cases

Use Case	Link to the GitHub repository
Drop	Link to code
Pass	Link to code
Echo Server	Link to code
L3 Forwarding	Link to code
Broadcast	Link to code

4.1.1 Drop

In this scenario, we demonstrate the feasibility of discarding all incoming packets using a P4 program. However, merely having a P4 program is insufficient for testing this capability, as it necessitates a compatible platform that supports the P4 language. BMv2 is a crucial component in the realm of P4 programs. It serves as a software switch and a reference model for emulating network device behaviors. Its primary function is to execute P4 programs, interpreting the defined packet processing logic.

Specifically, BMv2 allows P4 programs to dictate the conditions under which incoming packets should be dropped, effectively preventing them from progressing through the network. The ”Drop” action in P4 is often defined as a set of instructions or rules that signal to BMv2 to discard packets based on specified criteria.

To compile the P4 program, we will utilize the p4c compiler, which is the reference compiler for the P4 language. It's a modular tool that offers flexibility in selecting various compilation targets. The compilation process for P4 programs comprises two stages. In the first stage, a ***.p4info** file is generated, containing all the essential runtime attributes of the P4 program, such as table identifiers, structure, and actions. The second stage, the backend compilation, utilizes the generated *.p4info file to create the necessary files for programming the target device.

For instance, when targeting BMv2, the backend compiler produces a *.json[17] file. This file is sufficient to configure the datapath as specified in the P4 program. We will use the p4c compiler with the target ”p4c-bm2-ss,” which supports the **v1model** architecture[19] and is compatible with P4− simple−switch - BMv2.

To initiate the scenario, we've created a Makefile to compile our P4 program, resulting in *.p4info and *.json files. Next, we will run the **run−exercise.py** script, which will set up the network topology defined in the scenario/topology.json file using Mininet. Each switch in the topology will incorporate our P4 program logic, all managed within one BMv2 instance. The diagrams below provide an overview of a single switch's configuration.

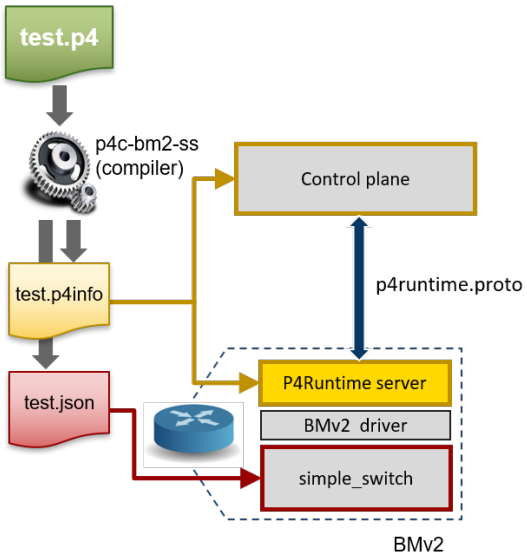


Figure 4.1: P4 program compilation process

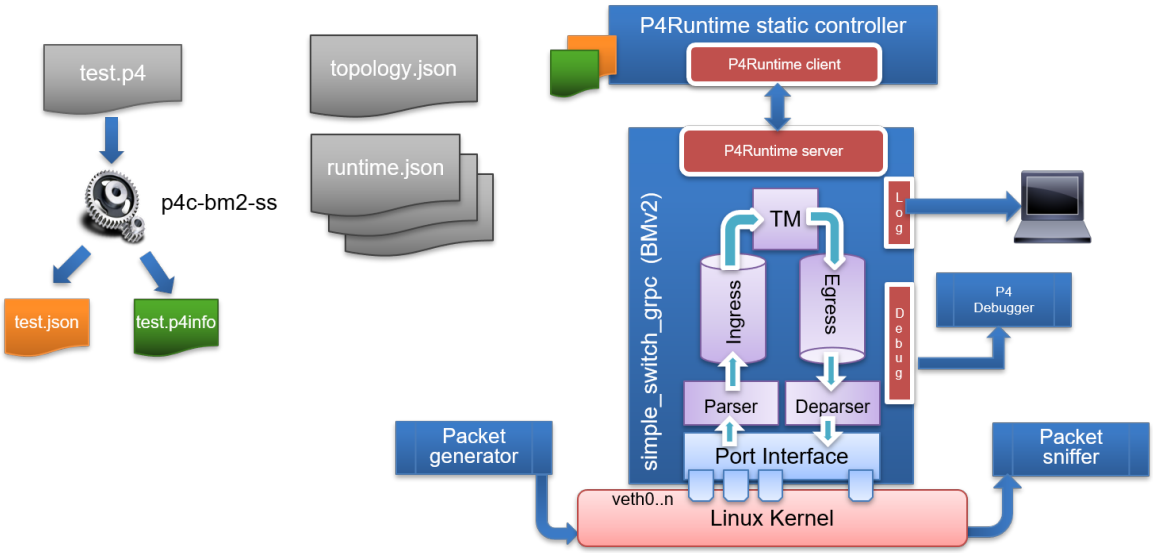


Figure 4.2: An overview of a single switch's configuration (BMv2)

Before running the Makefile, we will illustrate the network topology for the ”Drop” use case as follow.

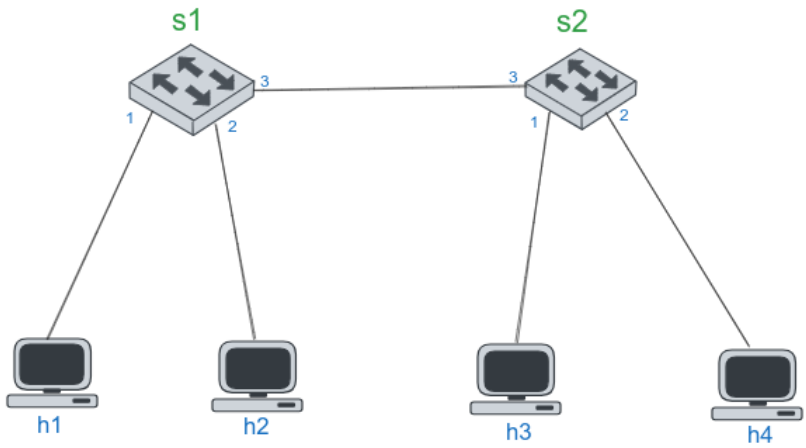


Figure 4.3: Network Topology (Drop)

First we need to apply a Makefile which automates the tasks of program compilation and loading, as well as the cleanup activities once the use case demonstration is completed. To initiate the implementation of the use case, follow the instructions detailed in code box [4.1].

Code 4.1: Compilation of the P4 program

```
# Go to the drop directory
cd P4-Project/P4/Drop/
# Run the Makefile
sudo make run
```

Furthermore, the Makefile will generate auxiliary files needed for loading the P4 program into BMv2, along with the creation of directories for pcaps, logs, and builds when the scenario is initiated. Once you have verified the functionality of the use case, you should utilize the ”stop” target provided by the Makefile for both Mininet and the P4 program.

Code 4.2: Stop the P4 program

```
# Stop the Makefile
sudo make stop
```

It is crucial to perform a cleanup, which involves removing files such as build, logs, and pcaps, in the following manner:

Code 4.3: Clean the P4 program

```
# Clean the Makefile
sudo make clean
```

Function check

After executing the ”make run” command within this directory, the use case has defined topology will be established, as depicted in Figure [4.4]. This topology comprises

- ”s1” (BMv2) connected with two hosts labeled as ”h1” and ”h2”
- ”s2” (BMv2) connected with two hosts labeled as ”h3” and ”h4”.

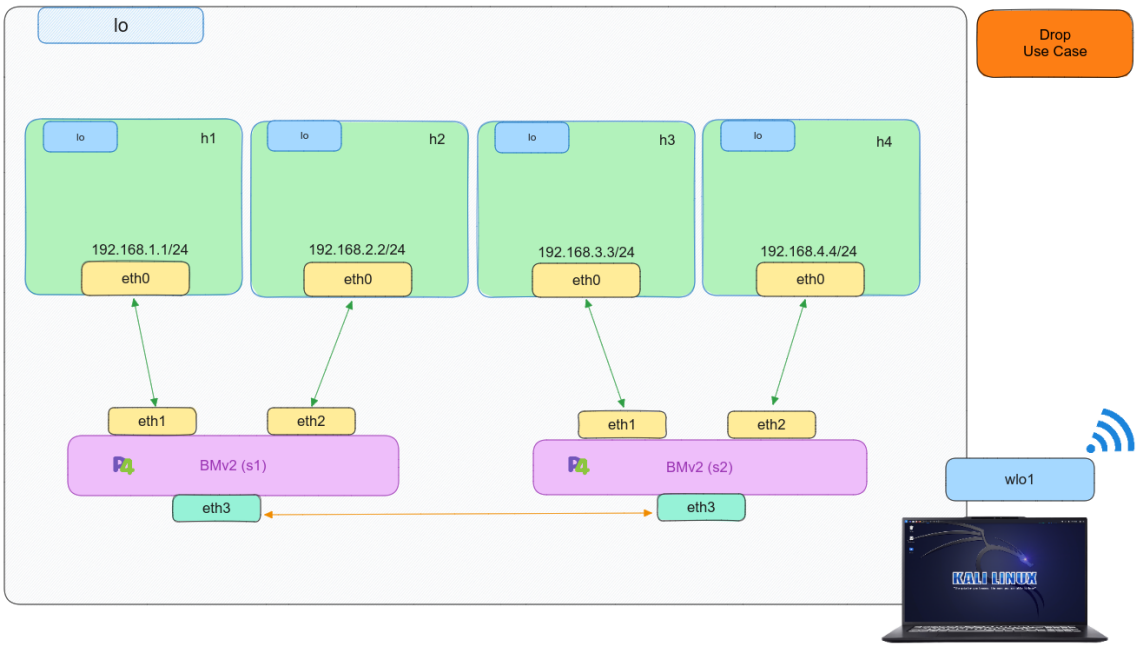


Figure 4.4: Drop topology

The topology's description can be found within the scenario directory in a JSON file named "topology.json". This file also specifies the locations of the files detailing the control plane for each "switch" in the topology. In all use cases, the naming conventions established by the P4Lang[20] organization have been adhered to, with file names following the format "sX-runtime.json," where "X" corresponds to the switch's position in the Mininet topology. To assess the operation of the use case, the Mininet Command Line Interface (CLI) will be accessible,we will open the xterm terminals for host1, host2, host3 and host4.

Code 4.4: Function check - Drop

```
mininet > xterm h1 h2 h3 h4
```

Before we test the results, we need to check the IP address of each host in xterm as the following Figure [4.5].

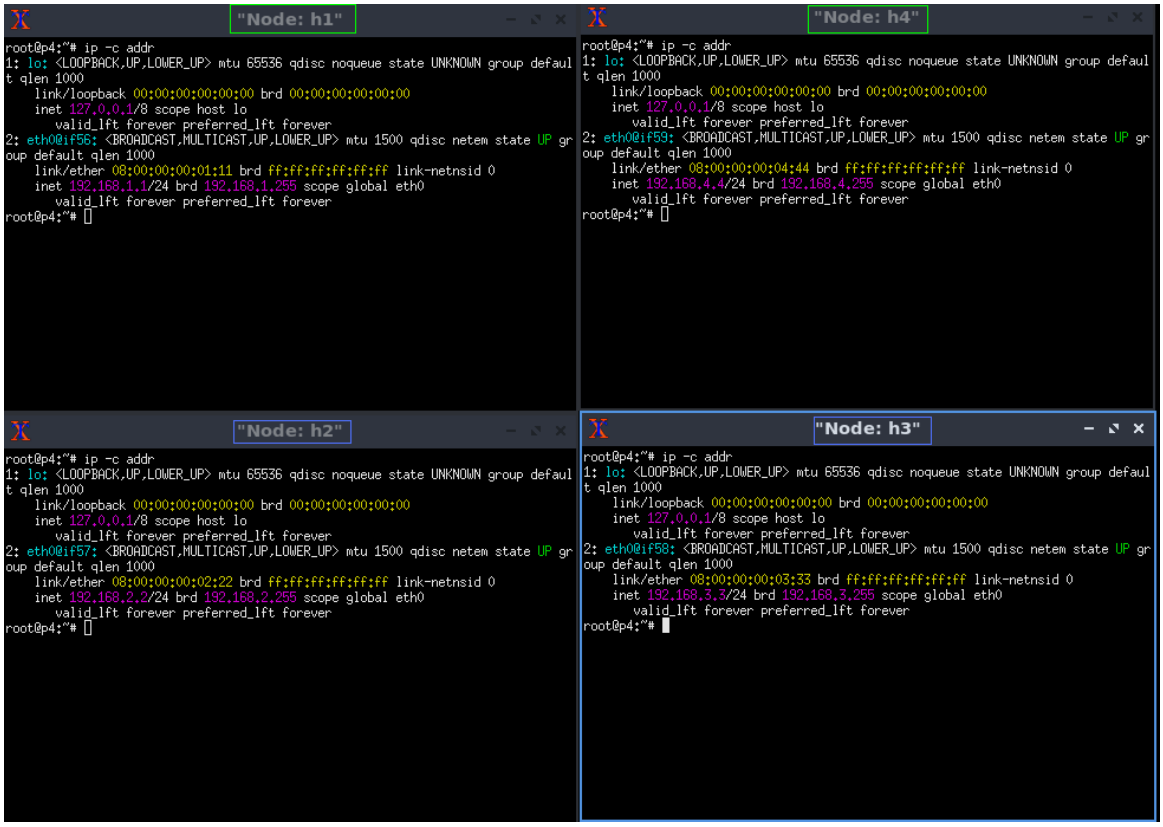


Figure 4.5: IP addressing of hosts

First, we initiate two ping requests. The first is from host1 (192.168.1.1) to host4

(192.168.4.4)(see in Figure [4.6]), and the second is from host4 (192.168.4.4) back to host1 (192.168.1.1). As evident in the results, the packets are intentionally dropped. This behavior is achieved by implementing code in **drop.p4** in the directory of ”**Drop**” to drop packets arriving at the ingress ports of the switches.

To further analyze how the packets were dropped, we capture network traffic using **Wireshark**[21]. In the Wireshark Figure [4.7], it is noticeable that only ARP (Address Resolution Protocol) packets are present. These ARP packets are used to discover the correct destination, but due to our intended code, the system do not successfully resolve the destination.

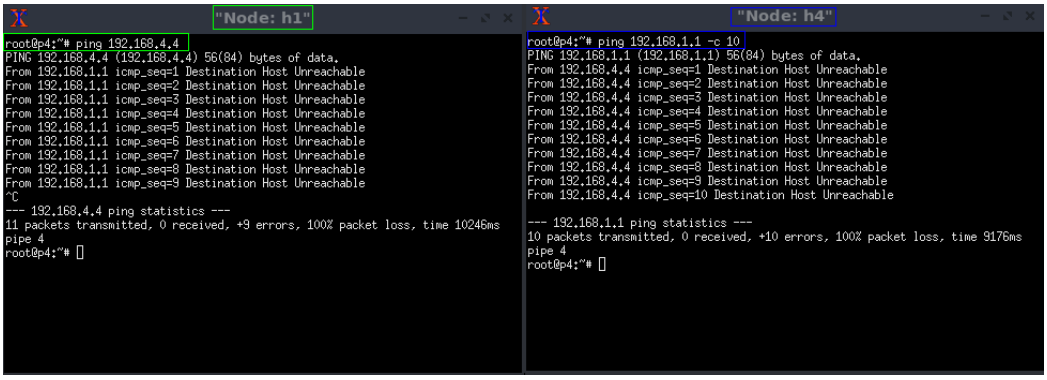


Figure 4.6: Testing ping between h1 and h4

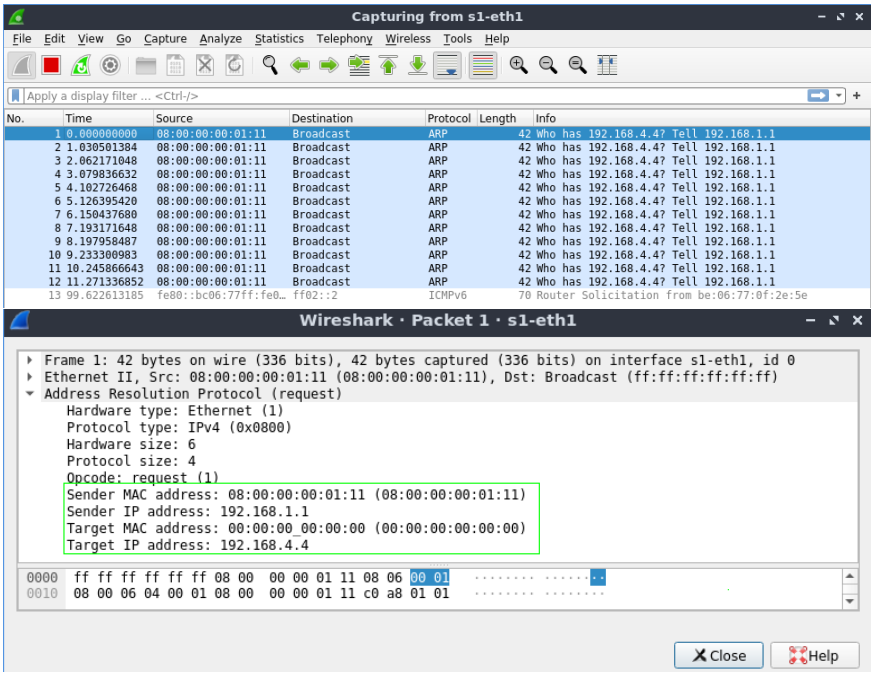


Figure 4.7: Capturing packets between h1 and h4

Next, we conduct a second test by initiating ping requests between host2 (192.168.2.2) and host3 (192.168.4.3), and then from host3 back to host2 (see in Figure [4.8]). As in the first scenario, the packets are intentionally dropped, which is achieved through the code implemented for this purpose in drop.p4. To gain a deeper understanding of how the packets are dropped in this scenario, we capture network traffic using Wireshark. The Wireshark Figures reveal that, similar to the previous scenario, only ARP (Address Resolution Protocol) packets are present.

These ARP packets are used in the process of discovering the correct destination, but due to our intended code, the destination is not successfully resolved. The following Figure [4.9] show the Wireshark result of the second test.

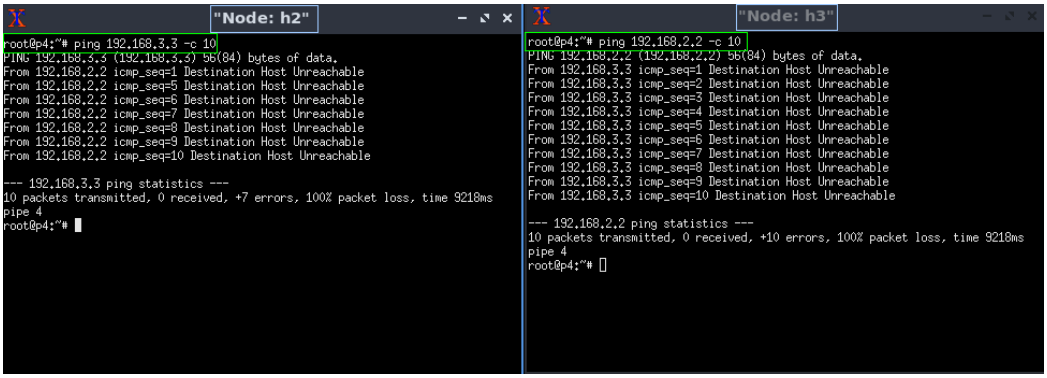


Figure 4.8: Testing ping between h2 and h3

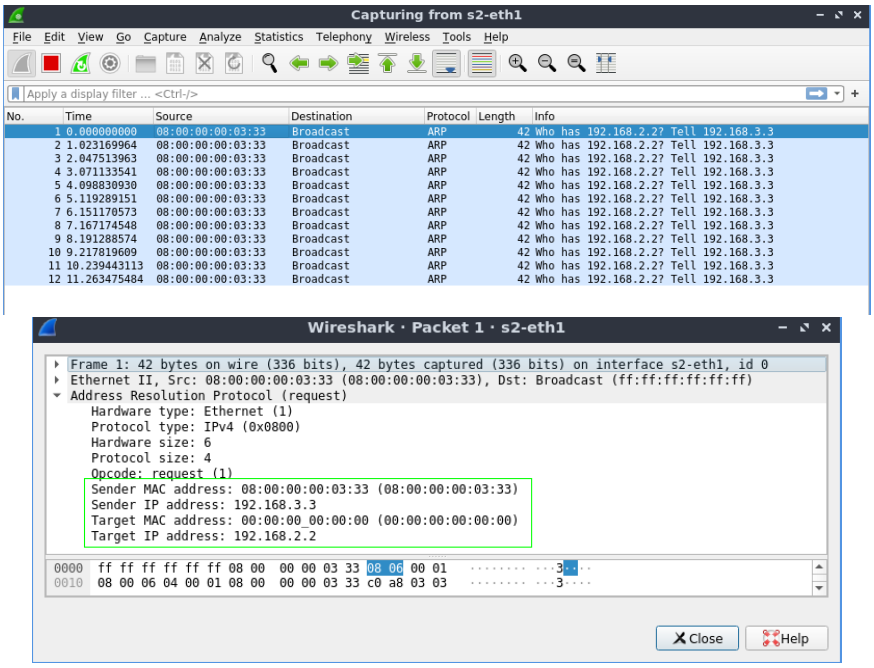


Figure 4.9: Capturing packets between h2 and h3

Additionally, we can examine the build, logs and pcaps directories generated at the start of the scenario to check the scenario.

Code 4.5: Function check - Drop

```
# Monitor the logs of the switch (s1)
less logs/s1.log
```

Code 4.6: Function check - Drop

```
# View JSON file
less build/drop.json
```

Code 4.7: Function check - Drop

```
# Monitor pcap file
wireshark pcaps/s1-eth1_in.pcap
```

4.1.2 Pass

In the context of P4, it is important to highlight that no P4 program has been developed for this specific use case(Pass). This steams from the unique nature of P4 programming, where each program is expected to explicitly define the entire path. P4 programs must indicate outline the entire processing pipeline with the BMv2 switches. For example, we need to define the datapath or specific instructions in **sX-runtime.json** files when the packets arrive to the BMv2 switches. In addition, the Mininet environment will be used to execute each scenario such as **Drop**, **Pass**, **Echo Server**, **L3 Forwarding** and **Broadcast**. Consequently, there is no direct way to delegate packet processing in P4, particularly for this specific use case.

4.1.3 Echo Server

In this scenario, we demonstrate an echo server that arrange all ICMP incoming packets using a P4 program. However, merely having a P4 program is insufficient for testing this capability, as it necessitates a compatible platform that supports the P4 language as we mentioned in drop use case (subsection 4.1.1).

In this use case, the P4 program needs to have the capability to inspect incoming packets, analyze their content, and apply a filtering process. Only those packets that pass through this filtering criteria should trigger responses. This involves introducing new states within the parser to examine whether ICMP [22] headers are present in the incoming packets. Therefore, the initial step is to declare the required ICMP headers to enable the analysis of

the packet headers upon arrival. In addition, we put this P4 program to be compatible with IPv6 addressing.

Code 4.8: ICMP and ICMPv6 header structure - Echo Server

```
header icmp_t {
        bit<8> type;
        bit<8> code;
        bit<16> checksum;
}
header icmp6_t {
        bit<8> type;
        bit<8> code;
        bit<16> checksum;
}
```

We also have to put IPv4 [23] and IPv6 [24] headers structure for this P4 program as follow.

Code 4.9: IPv6 header structure - Echo Server

```
header ipv6_t {
        bit<4> version;
        bit<8> nextHdr;
        bit<20> flowLabel;
        bit<16> payloadLen;
        bit<8> hopLimit;
        bit<8> trafficClass;
        ip6Addr_t srcAddr;
        ip6Addr_t dstAddr;
}
```

Code 4.10: IPv4 header structure - Echo Server

```
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;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

After defining the headers, we need to define macros for the Ethernet types we wish to handle as IPv4 and IPv6. We also need to define protocol codes for network headers to ensure only packets containing ICMP information are processed. In Code 4.11, you will find these macro definitions and extracts from the RFC explaining the encoding of the IPv6 headers.

Code 4.11: Macros for L2 and L3 - Echo Server

```
/* --- Info for Layer 2  --- */
const bit<16> etherType_ipv4 = 0x0800;
const bit<16> etherType_ipv6 = 0x86dd;
/* --- Info for Layer 3  --- */
const bit<8> ipProtocol_icmp = 0x01;
const bit<8> ipProtocol_icmpv6 = 0x3a;
const bit<8> icmpEchoRequestType = 0x08;
const bit<8> icmpEchoRequestCode = 0x00;
const bit<8> icmpEchoReplyType = 0x00;
const bit<8> icmpEchoReplyCode = 0x00;
```

With the filtered ICMP packets in hand, the next step is to implement the logic for altering the packet headers effectively so that we can provide a proper response to the sender of the ping. To achieve this, we will employ an action, essentially a function. This action will involve swapping both the MAC source-destination addresses and the source-destination IP addresses. Additionally, it will modify the ICMP header to indicate that it's a reply. Lastly, the packet will be sent out through the same port through which it initially entered the BMv2. This approach guarantees that the packet will reach its original sender. You will see Code 4.12 and 4.13 in ingress processing for the IPv4 and IPv6.

Code 4.12: Echo action for IPv4 - Echo Server

```
 action echo (){
     macAddr_t temp_mac = hdr.ethernet.srcAddr;
     ip4Addr_t temp_ip = hdr.ipv4.srcAddr;
     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
     hdr.ethernet.dstAddr = temp_mac;
     hdr.ipv4.srcAddr = hdr.ipv4.dstAddr;
     hdr.ipv4.dstAddr = temp_ip;
     hdr.icmp.type = icmpEchoReplyType;
     hdr.icmp.code = icmpEchoReplyCode;
     standard_metadata.egress_spec =
standard_metadata.ingress_port;
 }
```

Code 4.13: Echo action for IPv6 - Echo Server

```
 action echo6() {
     macAddr_t temp_mac = hdr.ethernet.srcAddr;
     ip6Addr_t temp_ip6 = hdr.ipv6.srcAddr;
     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
     hdr.ethernet.dstAddr = temp_mac;
     hdr.ipv6.srcAddr = hdr.ipv6.dstAddr;
     hdr.ipv6.dstAddr = temp_ip6;
     hdr.icmp6.type = icmpEchoReplyType;
     hdr.icmp6.code = icmpEchoReplyCode;
     standard_metadata.egress_spec =
standard_metadata.ingress_port;
 }
```

Final step for ingress processing is to check validity for IP header and ICMP header for either IPv4 or IPv6. If the packets contains such headers, its validity bit will be true and the packets will be forward. Next in Code 4.14, we can see the validity checking.

Code 4.14: Check Validity - Echo Server

```
apply {
    if(hdr.ipv4.isValid() && hdr.icmp.isValid()){
            echo();
    }else if (hdr.ipv6.isValid() && hdr.icmp6.isValid()){
            echo6();
    }
}
```

Compilation and Implementation

To compile the P4 program, we will utilize the p4c compiler, which is the reference compiler for the P4 language (more information about the compilation process in subsection 4.1.1 of drop use case). Before running the Makefile, we will illustrate the network topology for the ”Echo Server” use case which is similar to ”Drop” network topology.

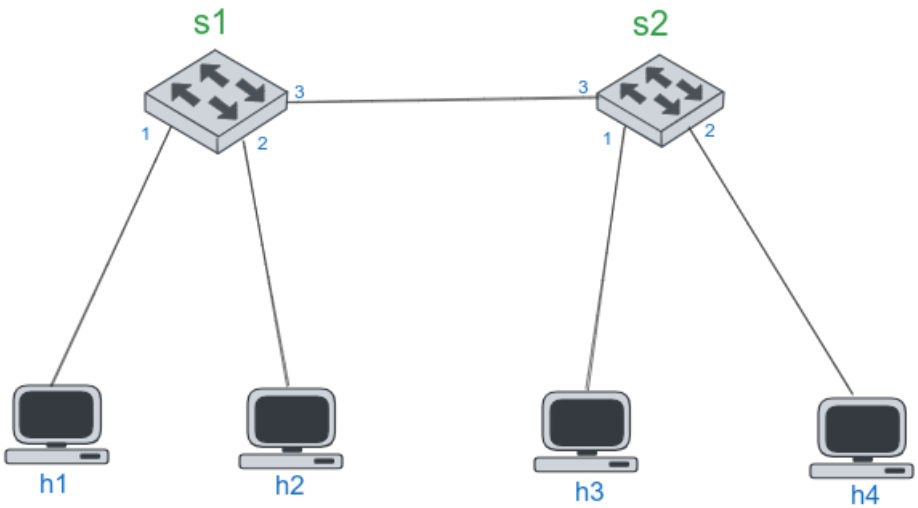


Figure 4.10: Network topology (echo server)

First we need to apply a Makefile which automates the tasks of program compilation and loading, as well as the cleanup activities once the use case demonstration is completed. To initiate the implementation of the use case, follow the instructions detailed in Code 4.15.

Code 4.15: Compilation of the P4 program - Echo Server

```
# Go to the Echo_server directory
cd P4-Project/P4/Echo_server/
# Run the Makefile
sudo make run
```

Furthermore, the Makefile will generate auxiliary files needed for loading the P4 program into BMv2, along with the creation of directories for pcaps, logs, and builds when the scenario is initiated. Once you have verified the functionality of the use case, you should utilize the ”stop” target provided by the Makefile for both Mininet and the P4 program.

Code 4.16: Stop the P4 program - Echo Server

```
# Stop the Makefile
sudo make stop
```

It is crucial to perform a cleanup, which involves removing files such as build, log, and pcaps after finished P4 program building.

Code 4.17: Clean the P4 program - Echo Server

```
# Clean the Makefile
sudo make clean
```

Function check

After executing the ”make run” command within this directory, the use case has defined topology will be established, as depicted in Figure [4.11]. This topology comprises:

- ”s1” (BMv2) connected with two hosts labeled as ”h1” and ”h2”
- ”s2” (BMv2) connected with two hosts labeled as ”h3” and ”h4”.

The topology’s description can be found within the scenario directory in a JSON file named ”topology.json”. This file also specifies ARP entries included in the hosts using

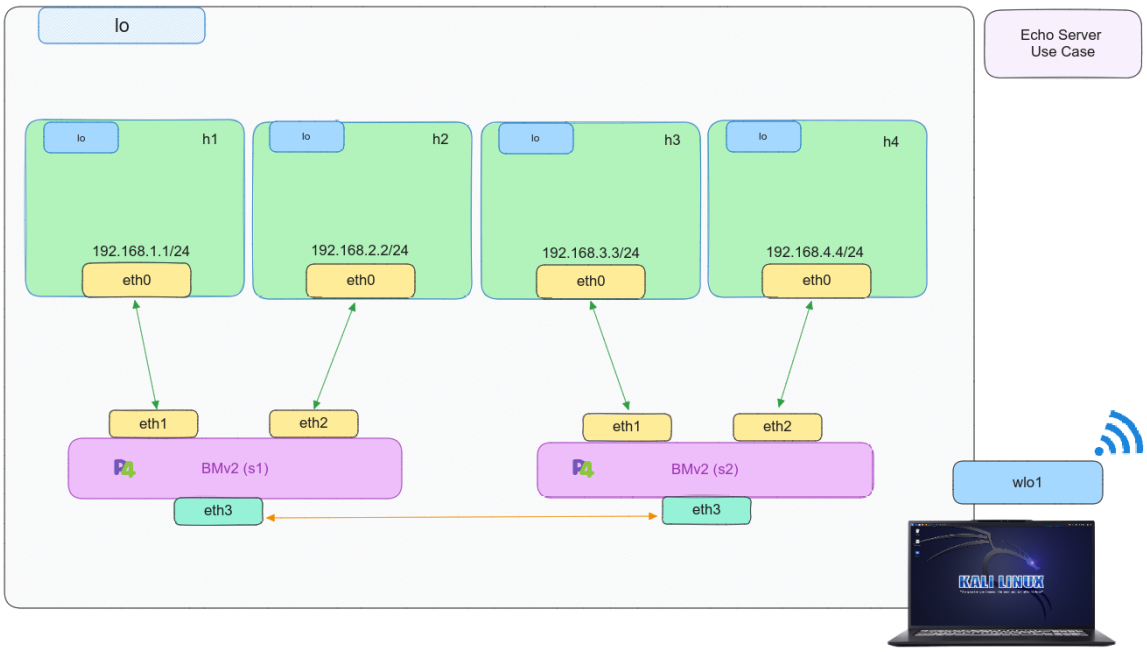


Figure 4.11: Topology of echo server

information. This setup ensures that ARP resolution is not triggered when sending ECHO-Requests. In this scenario, IPv4 will be used for the demonstration.

To assess the operation of the use case, the Mininet Command Lind Interface (CLI) will be accessible, we will open the xterm terminals for host1 and host4.

Code 4.18: Function check - Echo Server

```
mininet > xterm h1 h4
```

Before we proceed with testing the results, it's important to verify the IP address of every host using xterm, as illustrated in the following Figure [4.12].

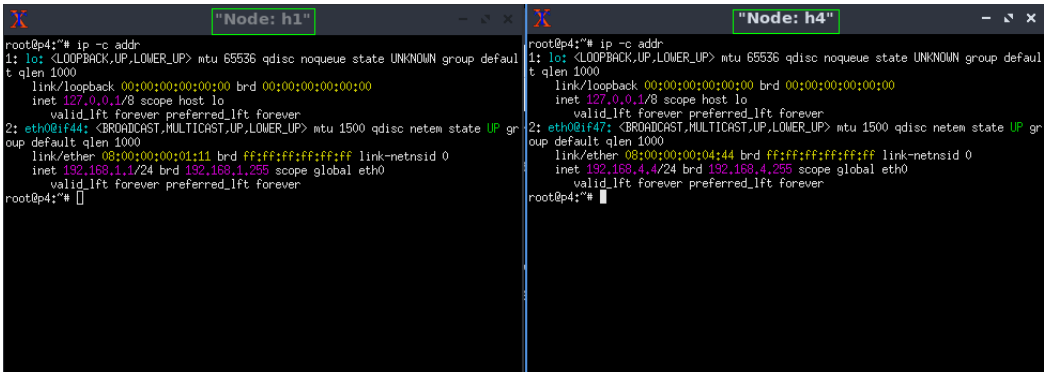


Figure 4.12: IP addresses of hosts (h1 and h4)

Next, we successfully conduct a ping (echo) test from host1 to host4, and the results are displayed in the Figure below.

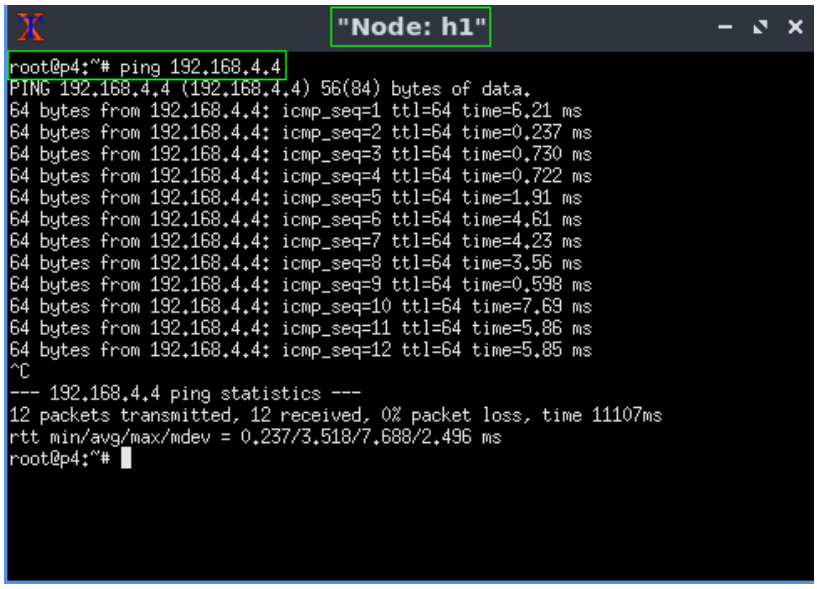


Figure 4.13: Testing echo ping

Additionally, we capture ICMP packet results in Wireshark between host1 and host4, as shown in the Figure below.

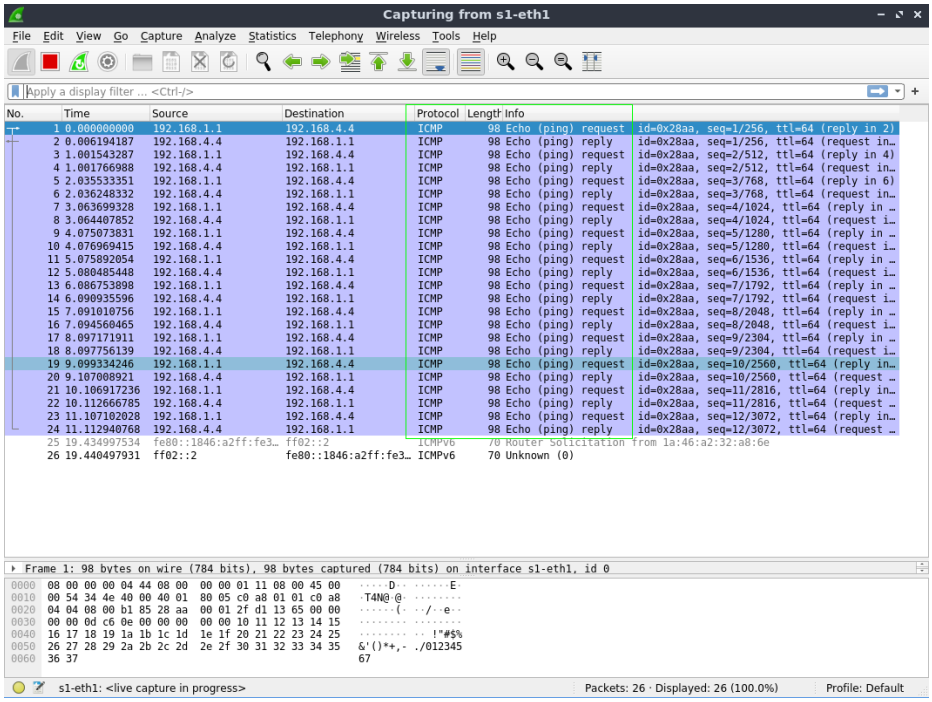


Figure 4.14: Captured traffic in BMv2-switch-1 (s1-eth1)

Moreover, we can examine the build, logs and pcaps directories generated at the start of the scenario to check the scenario.

Code 4.19: Function check - Echo Server

```
# Monitor the logs of the switch (s1)
less logs/s1.log
```

Code 4.20: Function check - Echo Server

```
# View JSON file
less build/echo_server.json
```

Code 4.21: Function check - Echo Server

```
# Monitor pcap file
wireshark pcaps/s1-eth1_out.pcap
```

4.1.4 L3 Forwarding

In this use case, we will focus on implementing basic layer 3 network forwarding, essentially turning the ”switch” into a simple router with limited capabilities. It’s crucial to recall the earlier notes regarding BMv2 and its behavior relies on the P4 program to define its datapath and its interaction with the control plane. In the P4 program, you need to outline the table structure by specifying available actions, their parameters, matching criteria, keys for lookup, and the maximum entry count. You can refer to Figure [4.15] for a concise overview of table functionality.

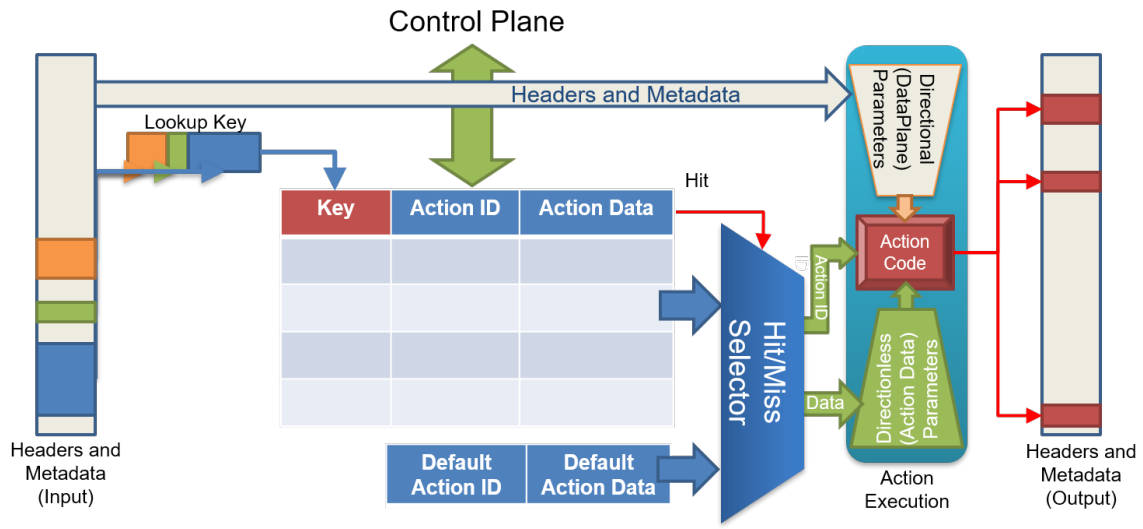


Figure 4.15: Operation of the tables in P4 [5]

To configure the table entries and action parameters from the control plane, either via P4Runtime or JSON using the sX-runtime.json files loaded through CLI-BMv2, you will associate IP with a forwarding action. In this scenario, we put the table entries in JSON using the sX-runtime.json files.

Code 4.22: Table entries - L3 Forwarding

```
{
  "table": "MyIngress.ipv4_lpm",
  "default_action": true,
  "action_name": "MyIngress.drop",
  "action_params": { }
},
{
  "table": "MyIngress.ipv4_lpm",
  "match": {
    "hdr.ipv4.dstAddr": ["192.168.1.1", 32]
  },
  "action_name": "MyIngress.ipv4_forward",
  "action_params": {
    "dstAddr": "08:00:00:00:01:00",
    "port": 3
  }
}
```

In **L3-forwarding.p4**, we define action which specifies the output port, destination MAC, and Time To Live (TTL) decrement for packet forwarding. In addition, we also define the table in ingress processing parser. You can see action and table in Code 4.23 and Code 4.24.

Code 4.23: Action to carry out forwarding - L3 Forwarding

```
 action ipv4_forward(macAddr_t dstAddr, egressSpec_t
port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
 }
```

Code 4.24: Define table in P4 - L3 Forwarding

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        ipv4_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
}
```

With the action in place, the final step is to integrate the table into our P4 program's pipeline, and this would enable layer 3 forwarding to occur.

Code 4.25: Apply action and table in P4 program - L3 Forwarding

```
apply {
    if (hdr.ipv4.isValid()) {
        ipv4_lpm.apply();
    }
}
```

Compilation and implementation

To compile the P4 program, we will utilize the p4c compiler, which is the reference compiler for the P4 language (more information about the compilation process in subsubsection 4.1 of drop use case). Before running the Makefile, we will illustrate the network topology for the "L3 Forwarding" use case.

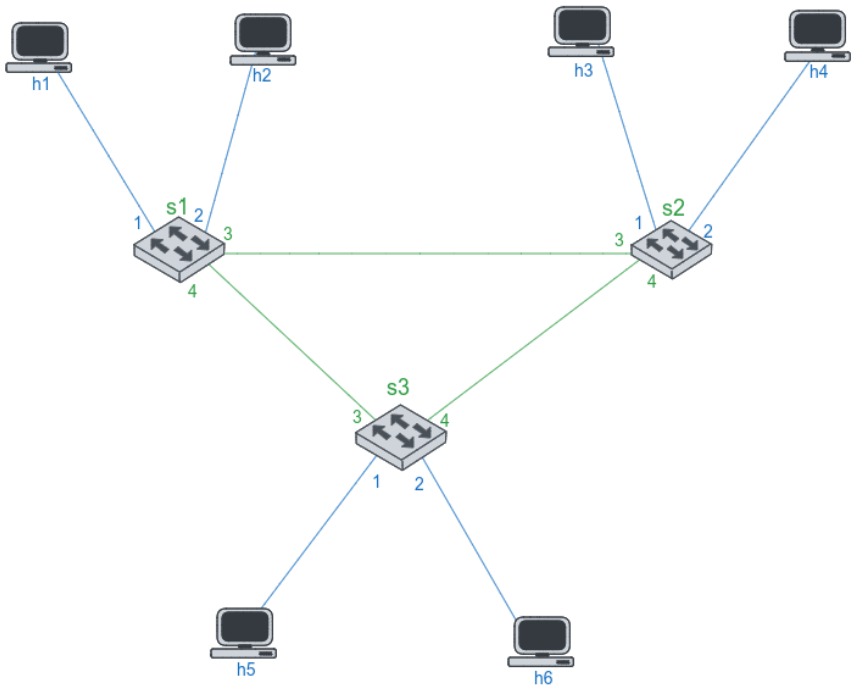


Figure 4.16: Network topology (L3 Forwarding)

To get started, the initial step involves using a Makefile to streamline the processes of compiling and loading the program, along with handling cleanup tasks once the use case demonstration is finished. To implement the use case, refer to the instructions provided in Code 4.26.

Code 4.26: Compilation of the P4 program - L3 Forwarding

```
# Go to the L3_forwarding directory
cd P4-Project/L3_forwarding/
# Run the Makefile
sudo make run
```

Additionally, the Makefile will produce the necessary supplementary files for loading the P4 program into BMv2, and it will establish directories for pcaps, logs, and builds at the commencement of the scenario. After confirming the proper operation of the use case, it is advisable to employ the ”stop” target included in the Makefile for halting both Mininet and the P4 program.

Code 4.27: Stop the P4 program - L3 Forwarding

```
# Stop the Makefile
sudo make stop
```

It is crucial to perform a cleanup, which involves removing files like build, log, and pcaps, in the following manner:

Code 4.28: Clean the P4 program - L3 Forwarding

```
# Clean P4 program
sudo make clean
```

Function check

After executing the ”make run” command within this directory, the use case has defined topology will be established, as depicted in Figure [4.17]. This topology comprises

- ”s1” (BMv2) connected with two hosts labeled as ”h1” and ”h2”
- ”s2” (BMv2) connected with two hosts labeled as ”h3” and ”h4”
- ”s3” (BMv2) connected with two hosts labeled as ”h5” and ”h6”.

The topology’s description can be found within the scenario directory in a JSON file named ”topology.json”. This file also specifies ARP entries included in the hosts using information. This setup ensures that ARP resolution is not triggered when sending ECHO-Requests. It suggests the presence of a non-existent gateway and adds an ARP entry with the MAC address of the imaginary gateway. Consequently, all packets will be sent with the MAC address specified in the entry, bypassing the need for ARP resolution.

To assess the function of the use case, the Mininet Command Line Interface (CLI) will be accessible, we will open the xterm terminals for host1, host2, host3 and host5.

Code 4.29: Function check - L3 Forwarding

```
mininet > xterm h1 h2 h3 h5
```

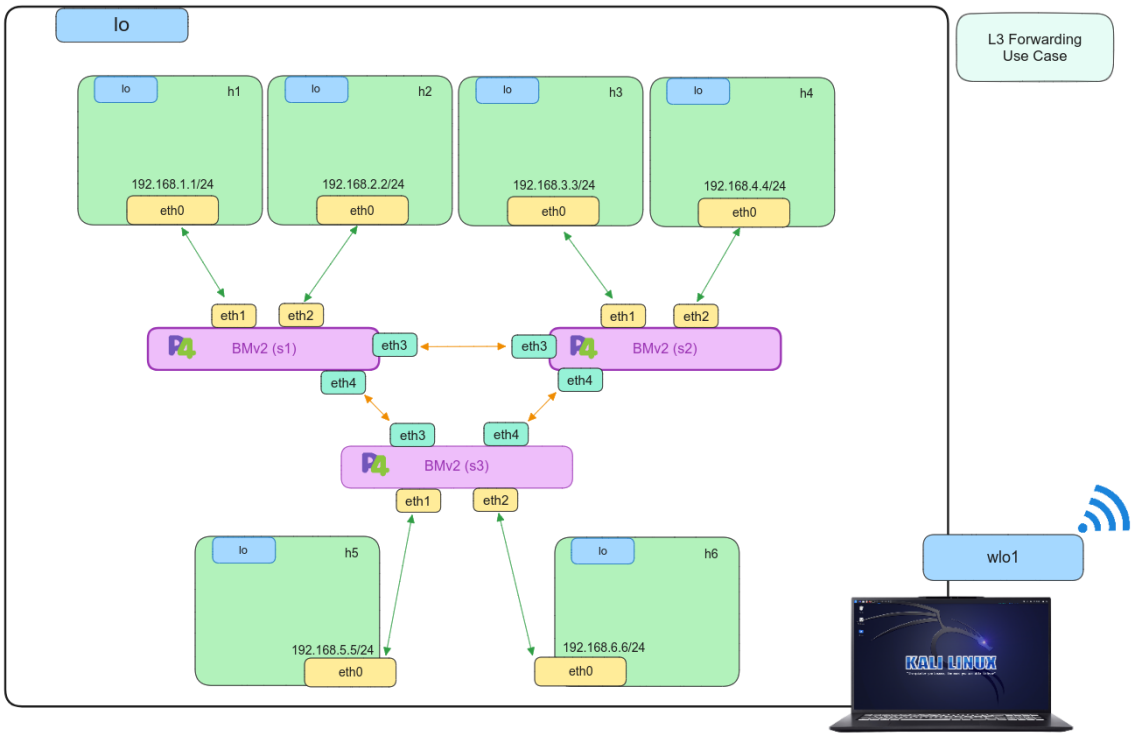


Figure 4.17: L3 forwarding topology

After executing 'sudo make run' for this use case, we will first verify the IP addresses of the hosts in order to conduct ping tests between hosts. The Figure below displays the IP addresses of the hosts.

We test ping communication between hosts connected to three Bmv2 switches. In the initial test, we ping from host1, connected to Bmv2-switch-1 (s1-eth1), to host3, connected to Bmv2-switch-2 (s2-eth1). The following Figure [4.19] demonstrates the successful delivery of packets from host1 to host3 using L3 forwarding.

As you're aware, L3 forwarding eliminates the need for broadcast-based packet delivery between switches. Each switch maintains a routing table, which reduces latency when forwarding packets between switches. In the two Figures below, the first Figure [4.20] displays ICMP results between host1 and host3 captured in BMv2-switch-1 (s1-eth3) and BMv2-switch-2(s2-eth3). In contrast, the second Figure [4.21] illustrates empty packets captured in BMv2-switch-s1(s1-eth4) and BMv2-switch-3(s3-eth3). This is because the routing table of BMv2-switch-1 indicates that the destination is in BMv2-switch-2 (s2-eth3)

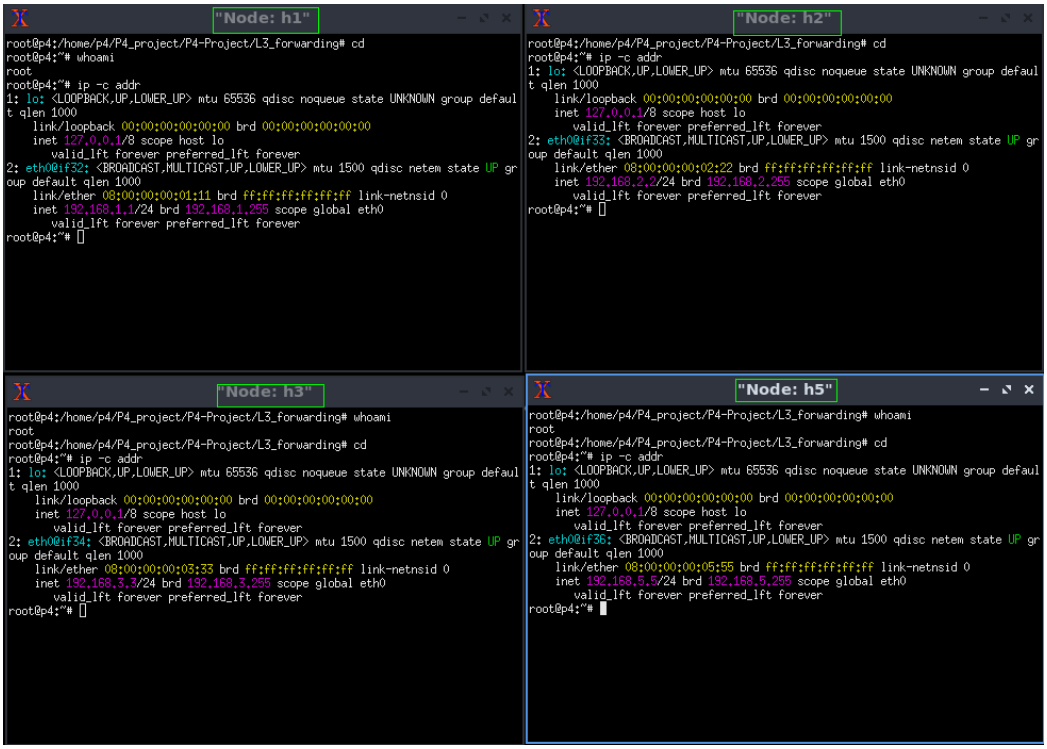


Figure 4.18: IP addresses of hosts

which is connected to BMv2-swithc-2(s2-eth3) but not in BMv2-switch-3 (s3-eth3) which is connected to BMv2-switch-1(s1-eth4).

Furthermore, we conduct a similar test between BMv2-swith-1 and BMv2-switch-3 with results mirroring the previous scenario. The ping test between host2 in BMv2-switch-1 and host5 in BMv2-switch-3 is also successful (see in Fig [4.22]).

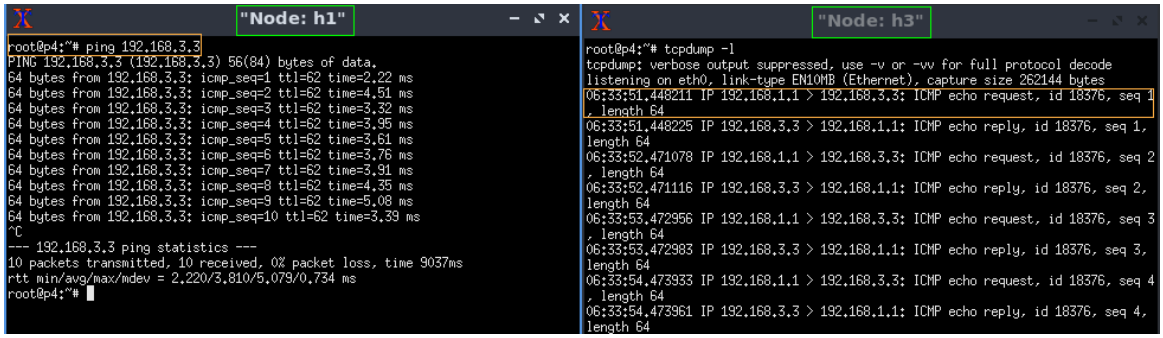


Figure 4.19: Testing ping result between hosts

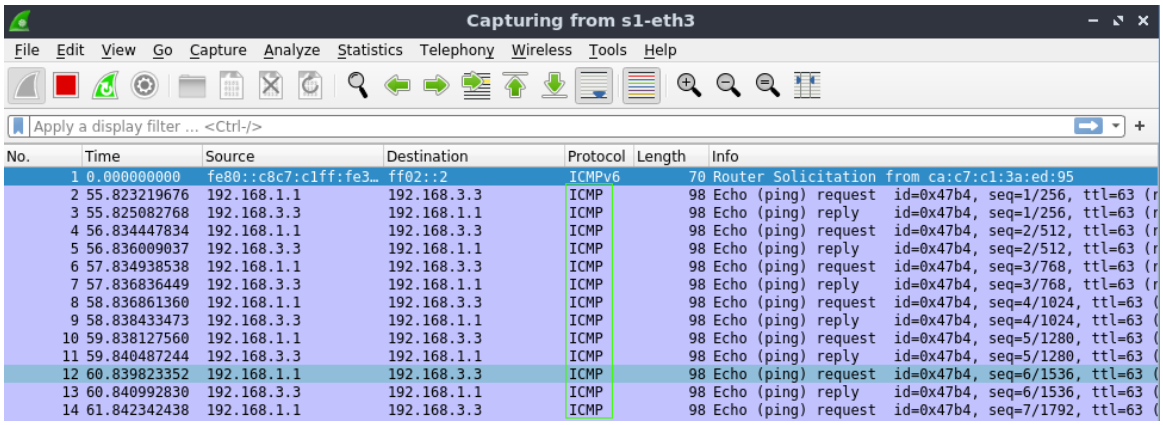


Figure 4.20: Captured ICMP in s1 (s1-eth3)

The captured packet results confirm that the packets did not traverse to the BMv2-switch-2 due to the routing table configuration in BMv2-switch-1. Consequently, no packets were captured from the BMv2-switch-2 side. The Figures below illustrate the packet capture results.

Moreover, we have the option to inspect the directories created at the beginning of the scenario, namely the build, logs, and pcaps directories, to assess the scenario.

Code 4.30: Function check - L3 Forwarding

```
# Monitor the logs of the switch (s3)
less logs/s4.log
```

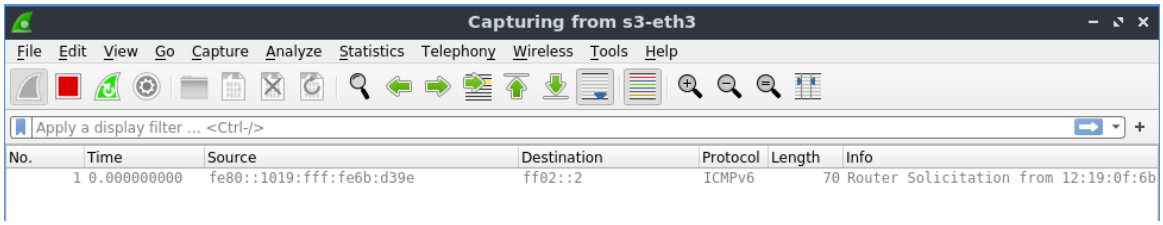


Figure 4.21: No captured packet in s3 (s3-eth3)

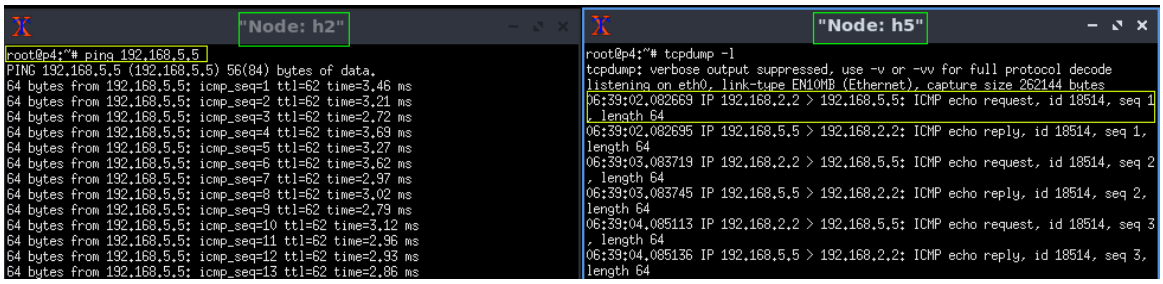


Figure 4.22: Testing ping result between hosts

Code 4.31: Function check - L3 Forwarding

```
# View JSON file
less build/l3_forwarding.json
```

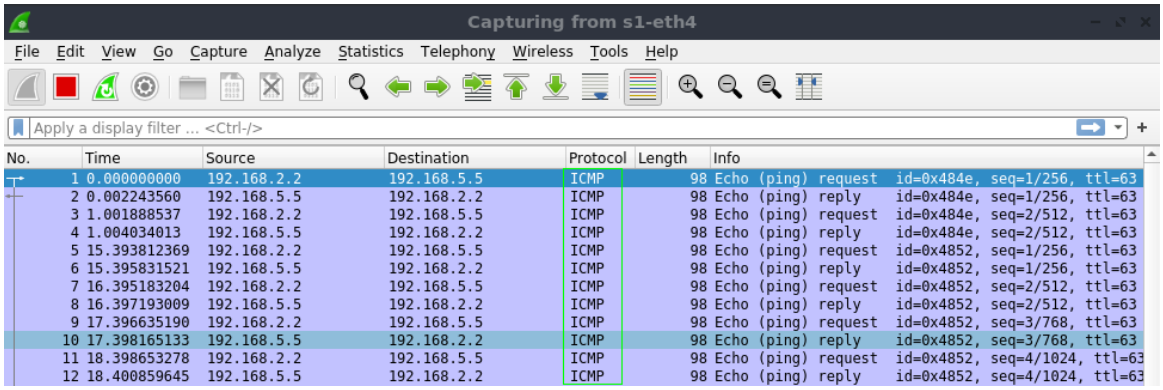
Code 4.32: Function check - L3 Forwarding

```
# Monitor pcap file
wireshark pcaps/s2-eth3_in.pcap
```

4.1.5 Broadcast

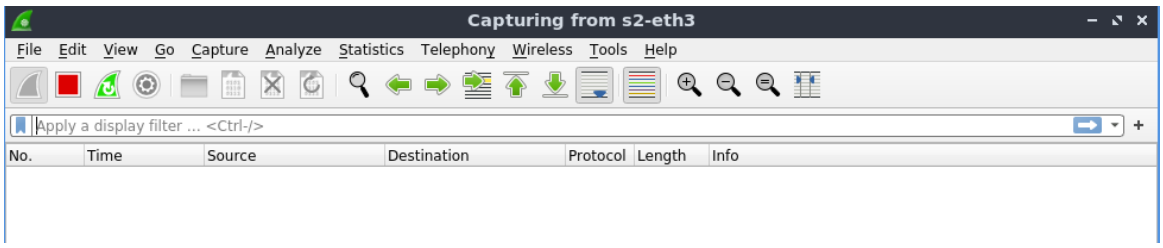
This is our final use case, focusing on broadcasting in P4 programming. Broadcasting operates at the Link Layer (Layer 2), and it involves flooding the received packets to all devices within the network segment, as it doesn't have a specific destination address.

In P4, broadcasting packets is achieved through multicast groups. These groups allow packets to be sent to a specific set of port numbers within the "switch." Each multicast group is identified uniquely and defines the number of packet duplicates to send and the associated port numbers. Configuration of these multicast groups takes place within the control plane of the "switch" and can be done using P4Runtime or by employing JSON files



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.2.2	192.168.5.5	ICMP	98	Echo (ping) request id=0x484e, seq=1/256, ttl=63
2	0.002243560	192.168.5.5	192.168.2.2	ICMP	98	Echo (ping) reply id=0x484e, seq=1/256, ttl=63
3	1.001888537	192.168.2.2	192.168.5.5	ICMP	98	Echo (ping) request id=0x484e, seq=2/512, ttl=63
4	1.004034013	192.168.5.5	192.168.2.2	ICMP	98	Echo (ping) reply id=0x484e, seq=2/512, ttl=63
5	15.393812369	192.168.2.2	192.168.5.5	ICMP	98	Echo (ping) request id=0x4852, seq=1/256, ttl=63
6	15.395831521	192.168.5.5	192.168.2.2	ICMP	98	Echo (ping) reply id=0x4852, seq=1/256, ttl=63
7	16.395183204	192.168.2.2	192.168.5.5	ICMP	98	Echo (ping) request id=0x4852, seq=2/512, ttl=63
8	16.397193009	192.168.5.5	192.168.2.2	ICMP	98	Echo (ping) reply id=0x4852, seq=2/512, ttl=63
9	17.396635190	192.168.2.2	192.168.5.5	ICMP	98	Echo (ping) request id=0x4852, seq=3/768, ttl=63
10	17.398165133	192.168.5.5	192.168.2.2	ICMP	98	Echo (ping) reply id=0x4852, seq=3/768, ttl=63
11	18.398653278	192.168.2.2	192.168.5.5	ICMP	98	Echo (ping) request id=0x4852, seq=4/1024, ttl=63
12	18.400859645	192.168.5.5	192.168.2.2	ICMP	98	Echo (ping) reply id=0x4852, seq=4/1024, ttl=63

Figure 4.23: Captured ICMP in s1 (s1-eth4)



No.	Time	Source	Destination	Protocol	Length	Info

Figure 4.24: No captured packet in s3 (s2-eth3)

like sX-runtime.json.

Code 4.33: Multicast table entry example in JSON file - Broadcast

```
"multicast_group_entries" : [
  {
    "multicast_group_id" : 1,
    "replicas" : [
      {
        "egress_port" : 1,
        "instance" : 1
      },
      {
        "egress_port" : 2,
        "instance" : 1
      },
      {
        "egress_port" : 3,
        "instance" : 1
      },
      {
        "egress_port" : 4,
        "instance" : 1
      }

    ]
  }
]
```

The JSON file specifies that any packets associated with multicast group ID 1 will trigger the creation of packet duplicates sent out through ”switch” ports 1, 2, 3, and 4. As a result, the broadcast action is defined as follows:

Code 4.34: Action to carry out P4 program - Broadcast

```
action mac_forward(egressSpec_t port) {
    standard_metadata.egress_spec = port;
```

Besides, we also need to define table in P4 code based on table key. It means that if the packet is Ethernet destination address, it will do the actions. You can see this in Code 4.35.

Code 4.35: Action to carry out P4 program - Broadcast

```
table mac_lookup {
    key = {
        hdr.ethernet.dstAddr : exact;
    }
    actions = {
        multicast;
        mac_forward;
        drop;
    }
    size = 1024;
    default_action = multicast;
}
```

With the action in place, the final step is to integrate the table into our P4 program's pipeline, and this would enable layer 2 broadcasting to occur.

Code 4.36: Action to carry out P4 program - Broadcast

```
apply {
    if (hdr.ethernet.isValid())
        mac_lookup.apply();
}
```

Compilation and implementation

For P4 program compilation, we will employ the p4c compiler, recognized as the standard compiler for the P4 language (detailed compilation process in subsection 4.1.1 of the relevant use case). Before executing the Makefile, we will provide an overview of the network topology for the ”Broadcast” use case.

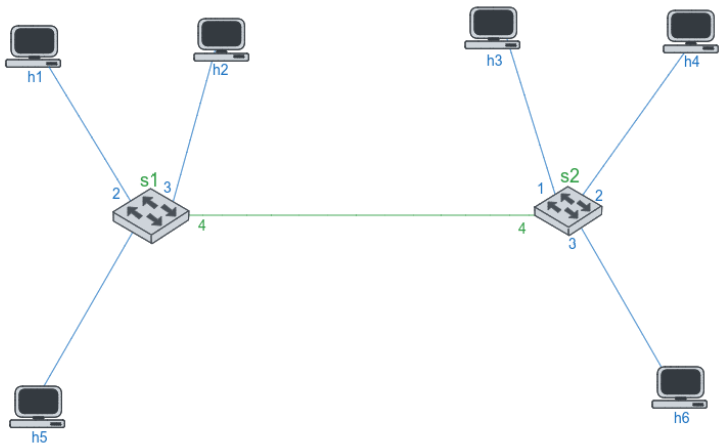


Figure 4.25: Network topology (broadcast)

To initiate the process, the first step is to employ a Makefile to simplify the compilation and loading of the program. Additionally, it will manage the cleanup tasks once the use case demonstration concludes. To implement the use case, please follow the guidelines outlined in Code 4.37.

Code 4.37: Compilation of the P4 program - Broadcast

```
# Go to the Broadcast directory
cd P4-Project/P4/Broadcast/
# Run the Makefile
sudo make run
```

Furthermore, the Makefile will generate auxiliary files needed for loading the P4 program into BMv2, along with the creation of directories for pcaps, logs, and builds when the scenario is initiated. Once you have verified the functionality of the use case, you should utilize the ”stop” target provided by the Makefile for both Mininet and the P4 program.

Code 4.38: Stop the P4 program - Broadcast

```
# Stop the Makefile
sudo make stop
```

It is crucial to perform a cleanup, which involves removing files such as build, log, and pcaps, in the following manner:

Code 4.39: Clean the P4 program - Broadcast

```
# Clean the P4 program
sudo make clean
```

Function check

After executing the ”make run” command within this directory, the use case has defined topology will be established, as depicted in Figure [36]. This topology comprises

- ”s1” (BMv2) connected with two hosts labeled as ”h1”, ”h2” and ”h3”
- ”s2” (BMv2) connected with two hosts labeled as ”h4”, ”h5” and ”h6”.

The topology’s description can be found within the scenario directory in a JSON file named ”topology.json”.

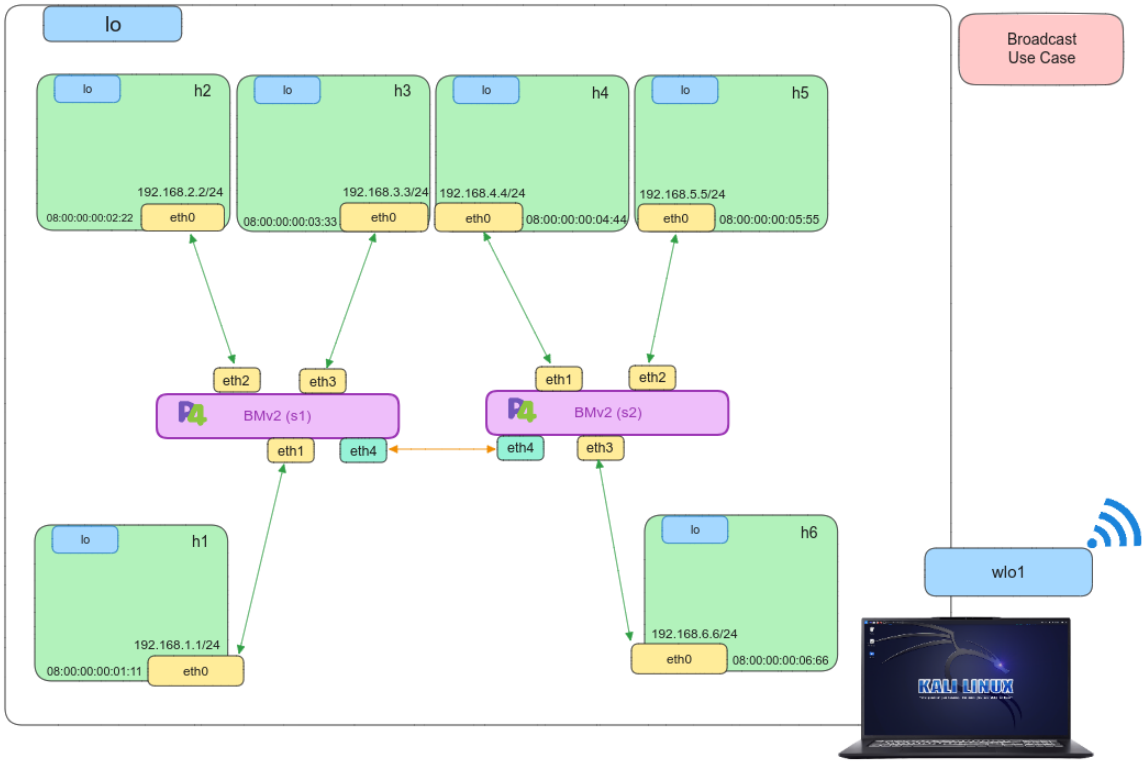


Figure 4.26: Broadcast topology

To assess the function of the use case, the Mininet Command Line Interface (CLI) will be accessible, we will open the xterm terminals for host1, host3, host5 and host6.

Code 4.40: Function check - Broadcasting

```
mininet > xterm h1 h3 h5 h6
```

Before conducting the test, we need to verify the IP addresses of each host. Here are the IP addresses of the hosts.

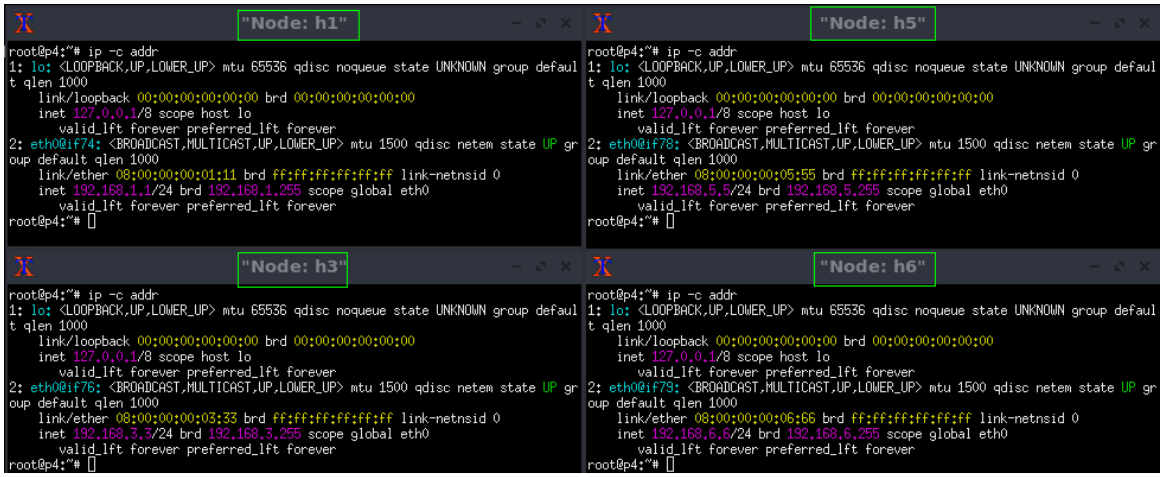


Figure 4.27: IP addresses of hosts

Next, we conduct a ping test from host1 in BMv2-switch-1 to host6 in BMv2-switch-2. Additionally, we capture the packets between one of the local hosts (host3) connected to BMv2-switch-1 and host5 connected to BMv2-switch-2.

During the test, when host1 wants to send packets to host6, BMv2-switch-1 receives the information, and broadcasts ARP requests to local hosts and BMv2-switch-2. Similarly, BMv2-switch-2 broadcasts ARP requests to its local hosts. Ultimately, host6 receives the ARP request and communicates it back to its BMv2-switch-2 regarding the destination. Subsequently, BMv2-switch-2 relays this information to BMv2-switch-1, and BMv2-switch-1 informs the source host1. This results in successful packet transmission between host1 and host6. The results, both in tcpdump and Wireshark, are depicted in the following Figures.

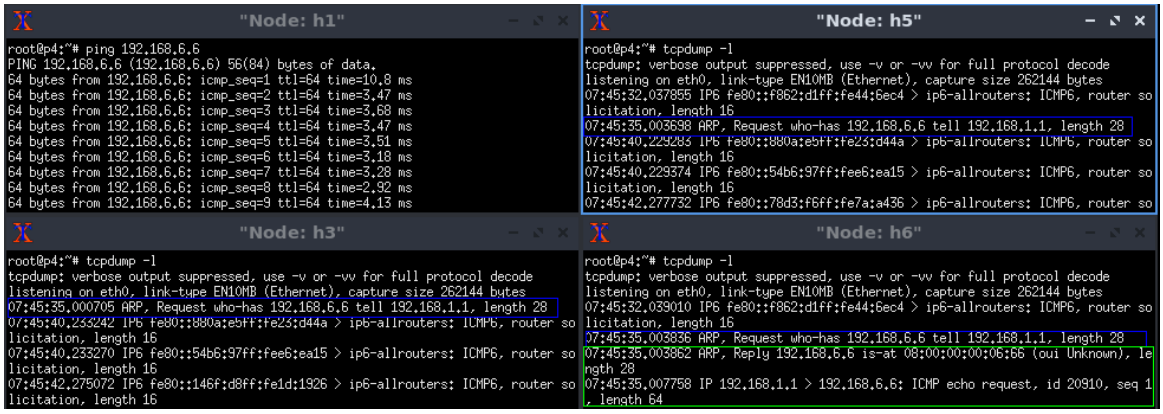


Figure 4.28: Captured traffic in tcpdump

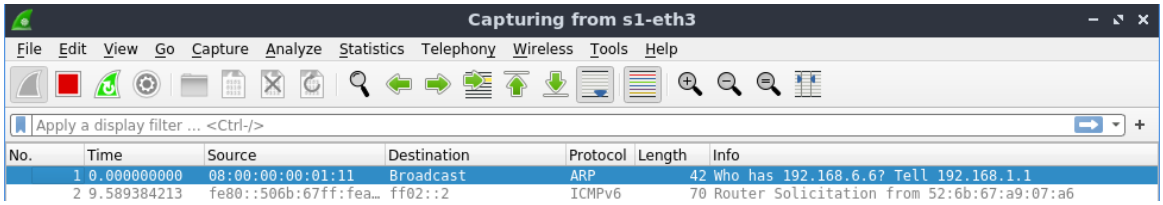


Figure 4.29: Captured ARP traffic in s1 (s1-eth3)

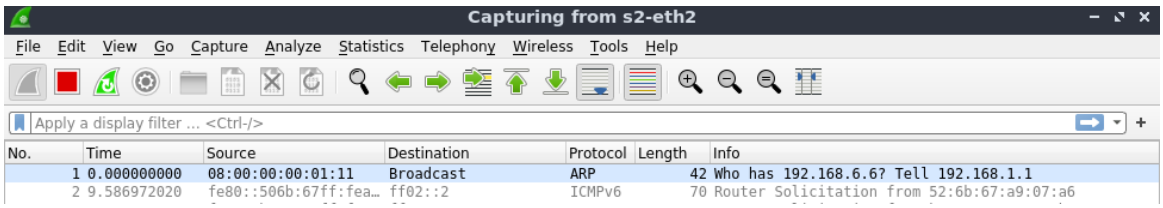


Figure 4.30: Captured ARP traffic in s2 (s2-eth2)

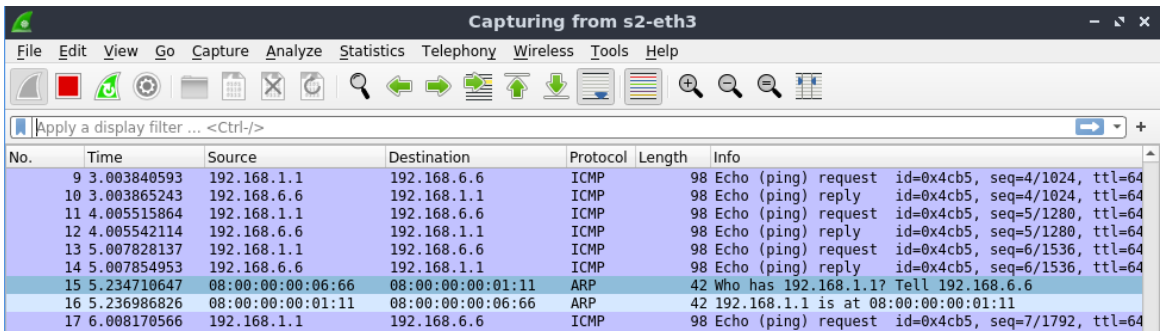


Figure 4.31: Captured ICMP traffic in s2 (s2-eth3)

Moreover, we have the option to inspect the directories created at the beginning of the scenario, namely the build, logs, and pcaps directories, to assess the scenario.

Code 4.41: Function check - Broadcast

```
# Monitor the logs of the switch (s2)
less logs/s2.log
```

Code 4.42: Function check - Broadcast

```
# View JSON file
less build/broadcast.json
```

Code 4.43: Function check - Broadcast

```
# Monitor pcap file
wireshark pcaps/s2-eth2_in.pcap
```

4.2 P4 use cases on wireless environment

This subsection will present various use cases utilizing P4 technology in wireless settings. **Appendix** outlines the steps required for installing the dependencies of P4 technology. Additionally, installing Mininet-WiFi is a prerequisite.

To enhance clarity and organization, these use cases, as earlier ones, have been segmented into distinct parts. This approach aims to streamline readability by referring the reader directly to relevant subsections where concepts or developments are similar to those in P4 use cases within wired environments.

Introduction : This subsection will cover additional theoretical insights, self-explanations regarding the use case, and comments on the created P4 code.

Compilation : This subsection will guide the reader on the process of compiling the P4 program.

Setting up the stage : This part will explain how to set up the stage.

Evaluation : Ultimately, an assessment will be conducted regarding the implementation of the use case through the Mininet-WiFi Command Line Interface (CLI).

To assist readers in tracking the progress of the P4 use cases on wireless environment, we have provided Table[4.2]. It details the specific location of each use case within the **P4-Project** repository.

Table 4.2: Summary documentation on P4 use cases

Use Case	Link to the GitHub repository
Drop	Link to code
Pass	Link to code
Echo Server	Link to code
L3 Forwarding	Link to code
Broadcast	Link to code

As mentioned in **Chapter 3's Analysis and Design** subsection, Mininet-WiFi serves as the chosen platform for evaluating P4 program performance in wireless environments. However, as this platform lacks native support for BMv2-compatible nodes, a prerequisite involves developing an integration of **BMv2** within Mininet-WiFi.

4.2.1 BMv2 integration in Mininet-WiFi

This subsection will discuss the integration process of BMv2 into Mininet-WiFi, which is divided into three parts. The initial two parts will delve into fundamental concepts and earlier advancements essential for understanding and addressing the creation of the BMv2 interface within Mininet-WiFi.

Analysis of the BMv2 interface - Mininet

The P4lang team aimed to test P4 programs on a soft-switch, BMv2, but lacked a platform to connect it with other entities. They integrated BMv2 with Mininet by introducing a new class, '**P4Switch**,' inheriting from Mininet's Switch class. They utilized ***.json** files to define network topology and control plane information. The **scenario.py** script orchestrates the topology using Mininet's Python API. The ExerciseRunner and ExerciseTopo classes process the topology. Figure [4.32] illustrates the relationship among entrypoint classes. The P4Switch class, inheriting from Mininet's Switch, includes daughter classes like P4RuntimeSwitch for P4Runtime configuration. Figure [4.33] provides a broader view of this class relationship.

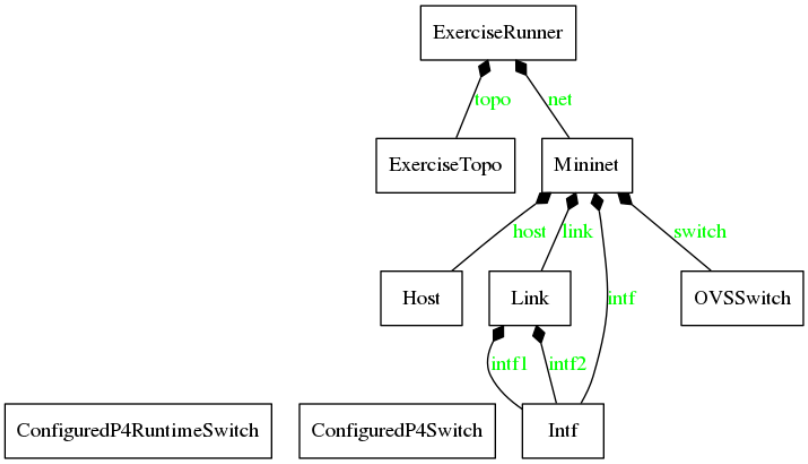


Figure 4.32: UML of the entry point of the BMv2 - Mininet Interface

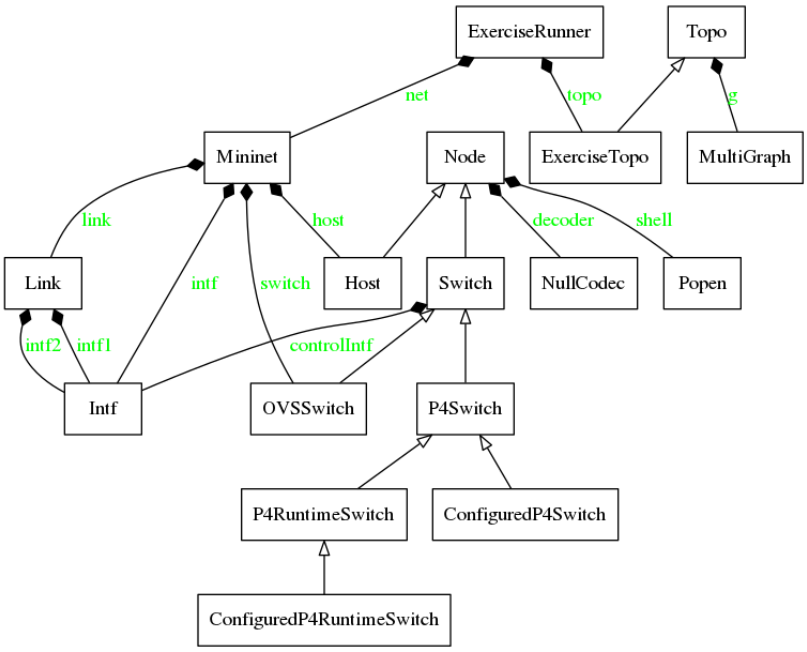


Figure 4.33: UML interface BMv2 - Mininet

Analysis of the Internal Operation of Mininet-WiFi

After reviewing the Mininet-P4 API's internal operations, the next phase involves a thorough examination of Mininet-WiFi. Derived from Mininet, Mininet-WiFi shares much of its class hierarchy and network simulation capabilities. However, its wireless interface emulation relies on the Linux Kernel's wireless subsystem. Similar to Mininet, Mininet-WiFi utilizes the mnexec9 tool to initiate separate bash processes in distinct Network

Namespaces for each network node. Upon emulation completion, these processes are terminated, removing any referencing conditions for the Network Namespaces, which are then eliminated by the Kernel.

The isolation among network nodes necessitates the virtualization of wireless capabilities, utilizing the mac80211-hwsim module within the Linux Kernel for generating wireless interfaces. This module communicates with the mac80211 framework to manage media access within the wireless interface. Additionally, within Kernel space, the cfg80211 acts as an API for configuring 802.11 headers, utilizing the user space netlink interface nl80211 for configuration. Access point configuration is facilitated by the HostApd program, simulating standard access point functionality by specifying access point settings and the operational interface. Figure [4.34] presents a summarized overview of the fundamental architecture of Mininet-WiFi.

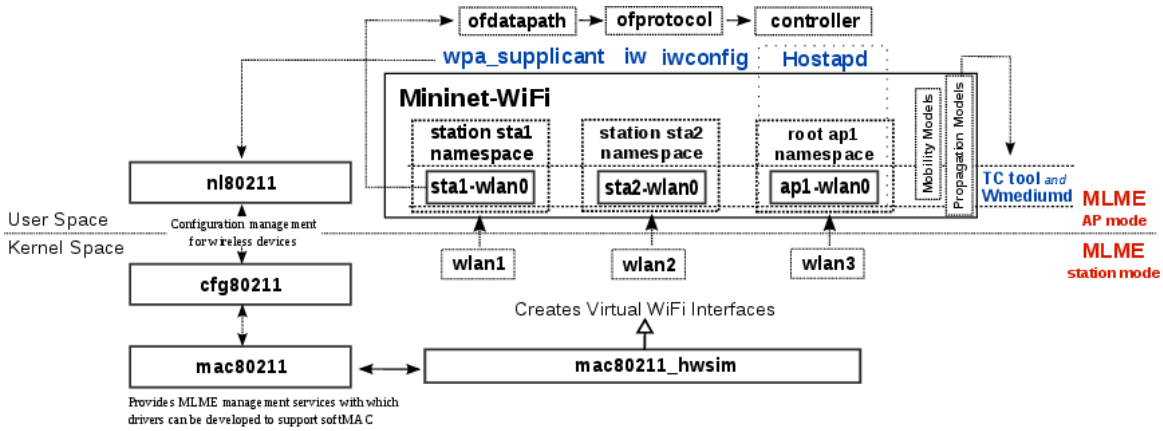


Figure 4.34: Mininet-WiFi architecture [6]

Linux Wireless Subsystem

The Linux wireless system has various modules handling hardware setup and data transmission under IEEE 802.11. The mac80211-hwsim module in this system creates virtual wireless interfaces in Mininet-WiFi, aiding developers in testing wireless card drivers and supporting different configurations. Wrappers around mac80211-hwsim improve its functions by utilizing the Netlink library to communicate with the Kernel subsystem for added settings like RSSI or interface names. The following Figure [4.35] shows the architecture

of the Linux wireless subsystem.

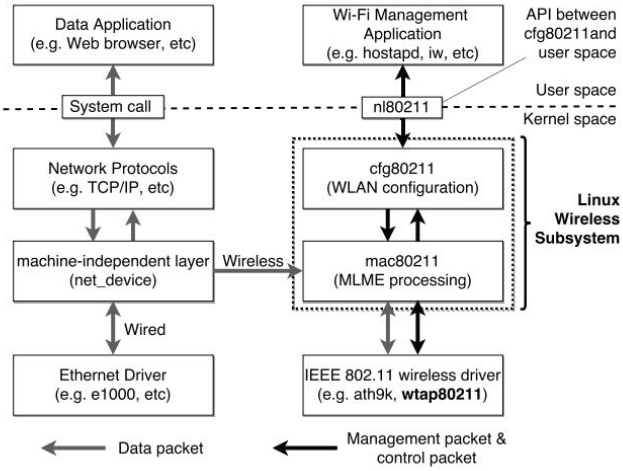


Figure 4.35: Linux Wireless Subsystem Architecture[7]

Tools such as mac80211-hwsim-mgmt are used by Mininet-WiFi to manage wireless interface creation for nodes. Unlike traditional Ethernet interfaces, the wireless subsystem has two layers: a physical layer managing listening channels and an access layer for virtual interfaces connected to a wireless card. This approach allows several virtual interfaces linked to the same emulated WiFi card, mainly functioning as Ethernet interfaces except for those in monitor mode.

4.2.2 Drop

This use case aims to demonstrate the capability of discarding received packets using a P4 program on a virtual wireless interface. To test this functionality, the behavioral-model switch software, BMv2, is utilized alongside the previously established integration with Mininet-WiFi to recreate network topologies.

As this use case is similar to the wired P4 **Drop** explained earlier (refer to subsubsection 4.1.1), with no notable environmental differences discussed in the previous subsection, only instructions on compilation and implementation will be provided. It is essential to note that to replicate this use case, ensure that Mininet-WiFi dependencies with BMv2 support are adequately installed.

Compilation

For this use case, a Makefile simplifies compilation, eliminating the need for users to learn the p4c compiler process. To delve deeper into the compilation stages or targets based on architecture, refer back to **Drop**. Simply follow the steps outlined for the compilation process.

Code 4.44: Compilation P4 program - Drop

```
# Entering the directory
cd P4-Project/P4-wireless/Drop
# Using the Makefile
sudo make
```

After executing the 'make' command, a directory structure is created for the use case launch. Within the 'build' directory, the compiler generates a JSON file containing all the necessary information to configure BMv2.

Set up the stage

Similar to the compilation process, a Python script is provided to automate the scenario setup, outlining the topology to be employed in this specific use case. To begin, navigate to the directory of the particular use case we wish to configure, and then execute the following command.

Code 4.45: Starting the scenario - Drop

```
# Under the directory P4_Project/P4-wireless/Drop
sudo python scenario.py
```

Function Check

After executing the script, the use case will establish the topology outlined in Figure [4.36], comprising four hosts and two instances of the P4RuntimeAP nodes.

- ”ap1” as a P4RuntimeAP, will possess two interfaces - one wireless (sta1) and one Veth pair (h1).
- ”ap2” as a P4RuntimeAP, will possess two interfaces - one wireless (sta2) and one Veth pair (h2).

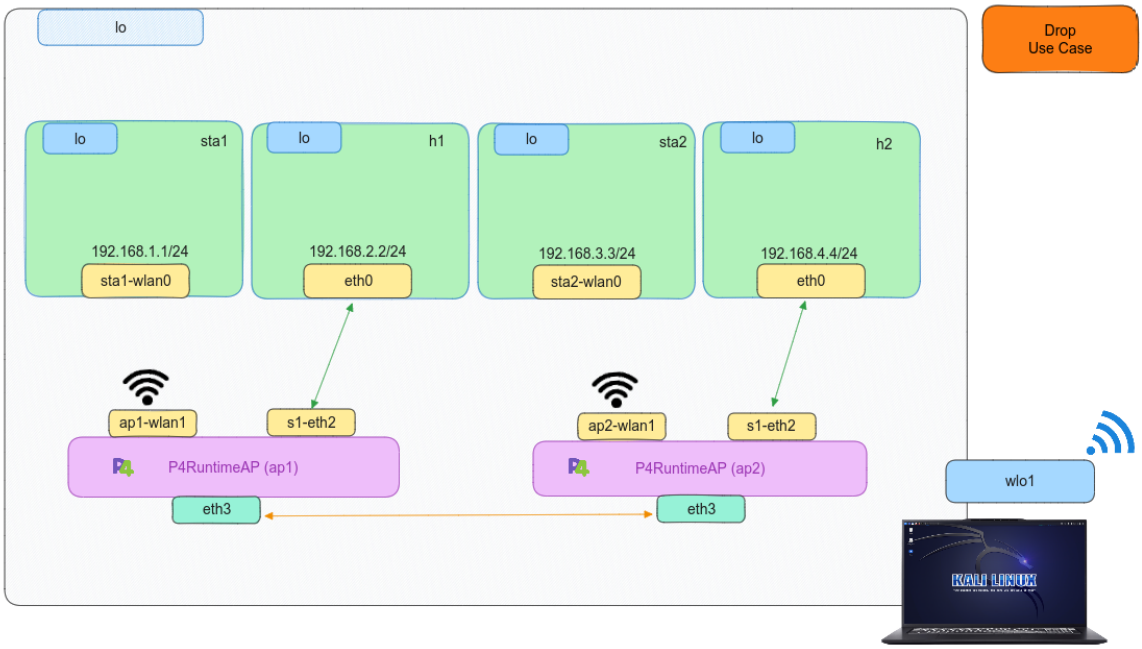


Figure 4.36: Drop topology

To assess the operation of the use case, the Mininet Command Line Interface (CLI) will be accessible, we will open the xterm terminals for wireless host1 (sta1), wireless host2 (sta2) and host1 (h1).

Code 4.46: Function check - Drop

```
mininet > xterm sta1 sta2 h1
```

Before we test the results, we need to check the IP address of each host in xterm as the following Figure [4.37].

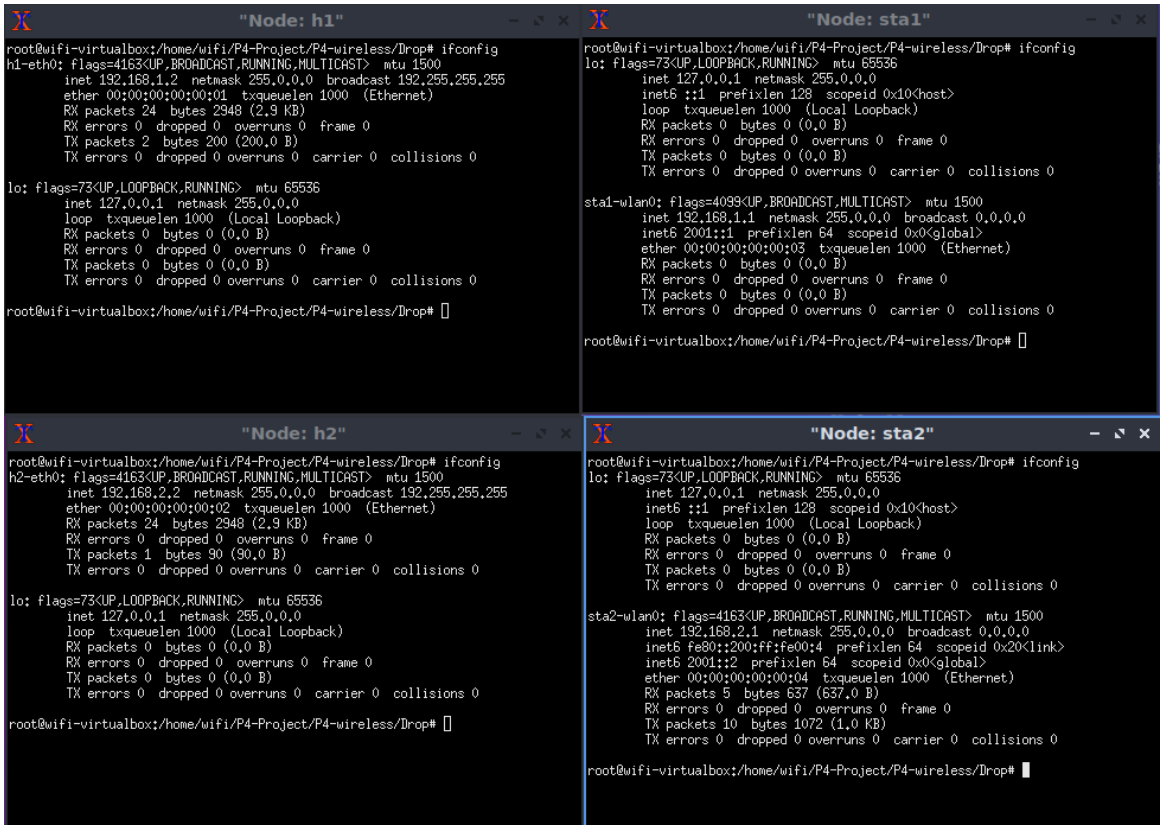


Figure 4.37: Ip addressing of hosts)

First, we initiate two ping requests. The first is from wireless host1 [sta1] (192.168.1.1) to host1 [h1] (192.168.1.2)(see in Figure [4.38]), and the second is from host1 [h1] (192.168.1.2) back to wireless host1 [sta1] (192.168.1.1). As evident in the results, the packets are intentionally dropped. This behavior is achieved by implementing code in **drop.p4** in the directory of **Drop** under the **P4-wireless** folder to drop packets arriving at the ingress ports of the P4RuntimeAPs.

To further analyze how the packets were dropped, we capture network traffic using Wireshark. In the Wireshark Figure [4.39], it is noticeable that there is no response from destination. It is due to our intended code and the system do not successfully resolve the destination.

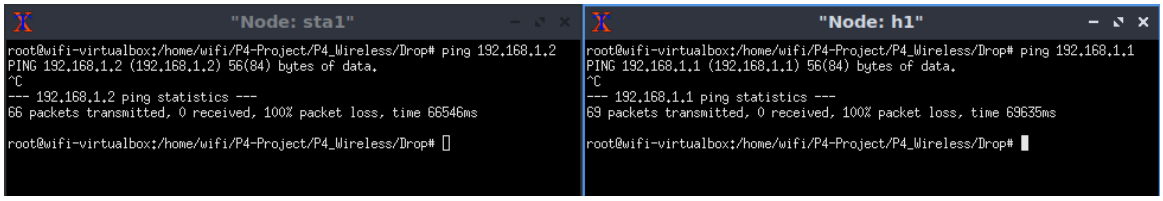


Figure 4.38: Testing ping between sta1 and h1

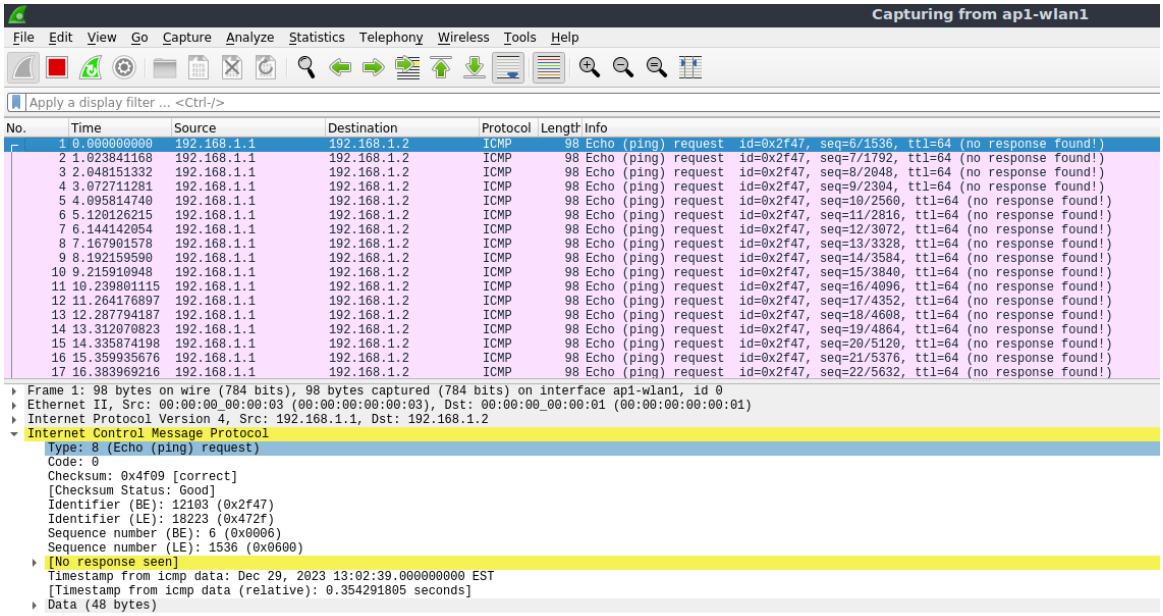


Figure 4.39: Capturing packets between sta1 and h1

Next, we conduct a second test by initiating ping requests between wireless host1 [sta1] (192.168.1.1) and wireless host2 [sta2] (192.168.2.1), and then from wireless host2 [sta2] back to wireless host1 [sta1] (see in Figure [4.40]). As in the first scenario, the packets are intentionally dropped, which is achieved through the code implemented for this purpose in **drop.p4**. To gain a deeper understanding of how the packets are dropped in this scenario, we capture network traffic using Wireshark. The Wireshark Figures reveal that, similar to the previous scenario, there is no response from the destination. The following Figure [4.41] show the Wireshark result of the second test.

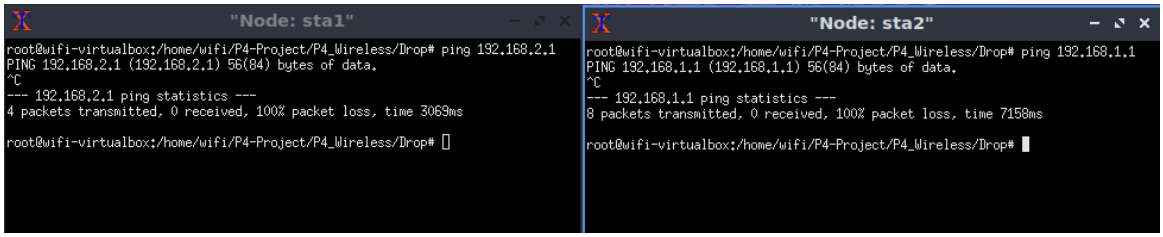


Figure 4.40: Testing ping between sta1 and sta2

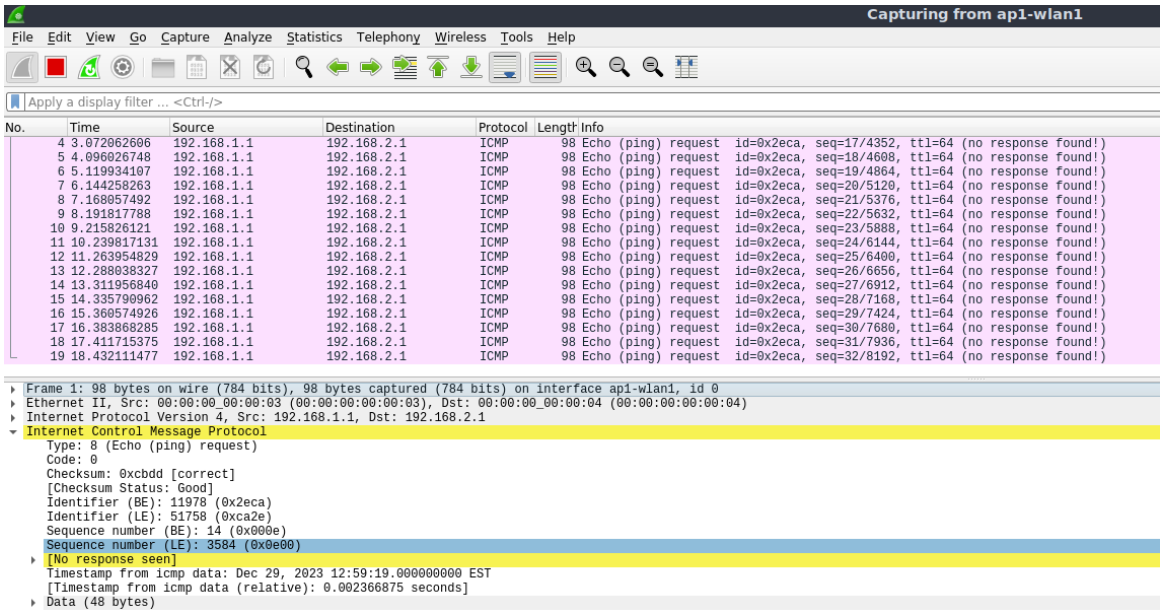


Figure 4.41: Capturing packets between sta1 and sta2

Additionally, we can examine the build, logs and pcaps directories generated at the start of the scenario to check the scenario.

Code 4.47: Function check - Drop

```
# Monitor the logs of the ap1-wlan0
less logs/ap1.logs
```

Code 4.48: Function check - Drop

```
# View JSON file
less build/drop.json
```

Code 4.49: Function check - Drop

```
# Monitor pcap file
wireshark pcaps/ap1-eth3-in.pcap
```

4.2.3 Pass

It can be seen, there is no P4 program in this directory, just like in the case of P4 use case "**Pass**" (subsubsection 4.1.2) in wired environments.

In P4, the environment where the use cases will be carried out will be Mininet-WiFi with APs (BMv2) and host. The APs (BMv2) are a software switch in which we can inject P4 code, with which we can define its datapath.

4.2.4 Echo Server

In this use case, we aim to create an echo server that replies to all incoming pings reaching the BMv2. To test this functionality, a platform supporting the P4 language is required. We will utilize the BMv2 switch software to test the P4 programs, alongside the established integration with Mininet-WiFi to replicate network topologies. P4Runtime will configure BMv2, utilizing the P4RuntimeAP class.

This use case has been previously detailed in **Echo Server** of wired P4 (refer to subsubsection 4.1.3). As there are no significant differences introduced in the change of environment, only instructions regarding compilation and execution will be provided. It is essential to note that to replicate this use case, ensure that Mininet-WiFi dependencies with BMv2 support are adequately installed.

Compilation

This use case offers a simplified compilation process using a Makefile, sparing users from the intricacies of learning the p4c compiler process. For further insights into compilation stages or architecture-based targets, refer to the **Echo Server** (4.1.3). Follow the provided steps to complete the compilation process.

Code 4.50: Compilation P4 program - Echo Server

```
# Entering the directory
cd P4_Project/P4-wireless/Echo_Server
# Using the Makefile
sudo make
```

Upon running the 'make' command, a directory structure is established to facilitate the launch of the use case. Within the 'build' directory, the compiler generates a JSON file that encompasses all the essential details required for BMv2 configuration.

Set up the stage

As the compilation process, there is a Python script available to automate the scenario setup, detailing the topology for this specific use case. In order to start this use case, go to the directory of the desired use case for configuration and run the specified command as follow.

Code 4.51: Starting the scenario - Echo Server

```
# Under the directory P4-Project/P4-wireless/Echo_Server
sudo python scenario.py
```

Function check

After executing the script, it will create the network structure described in Figure [4.42], consisting of four hosts and two P4RuntimeAP node instances.

- "ap1" as a P4RuntimeAP, will possess two interfaces - one wireless (sta1) and one Veth pair (h1).
- "ap2" as a P4RuntimeAP, will possess two interfaces - one wireless (sta2) and one Veth pair (h2).

To assess the operation of the use case, the Mininet Command Lind Interface (CLI) will be accessible, we will open the xterm terminals for wireless hosts (sta1) and host1 (h1).

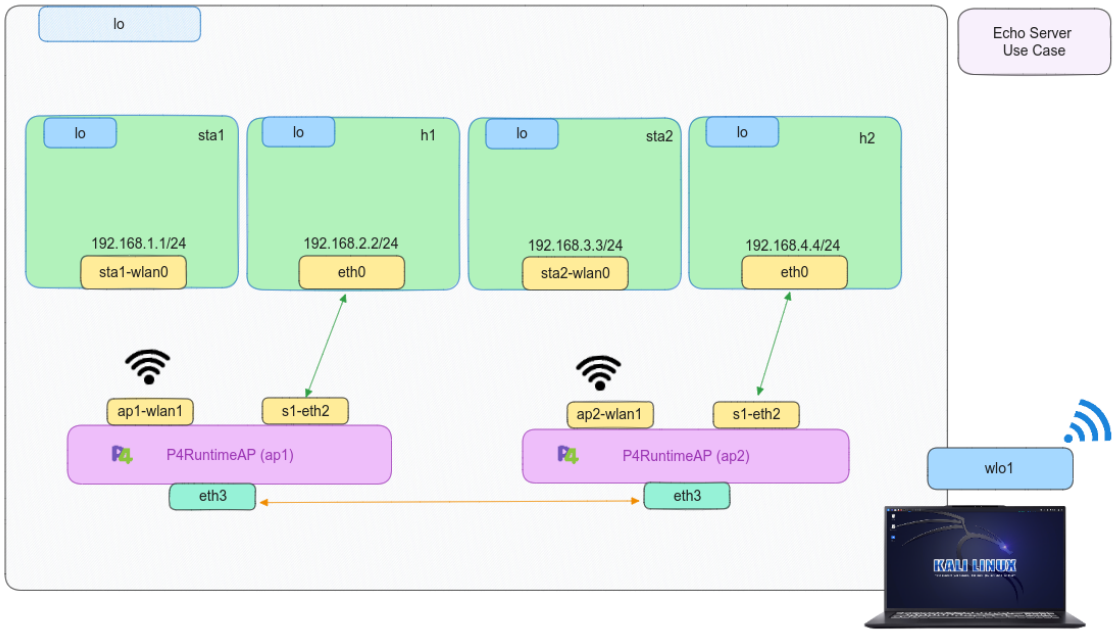


Figure 4.42: Echo server topology

Code 4.52: Function check - Drop

```
mininet > xterm sta1 h1
```

Before we proceed with testing the results, it is important to verify the IP address of every host using xterm, as illustrated in the following Figure [4.43].

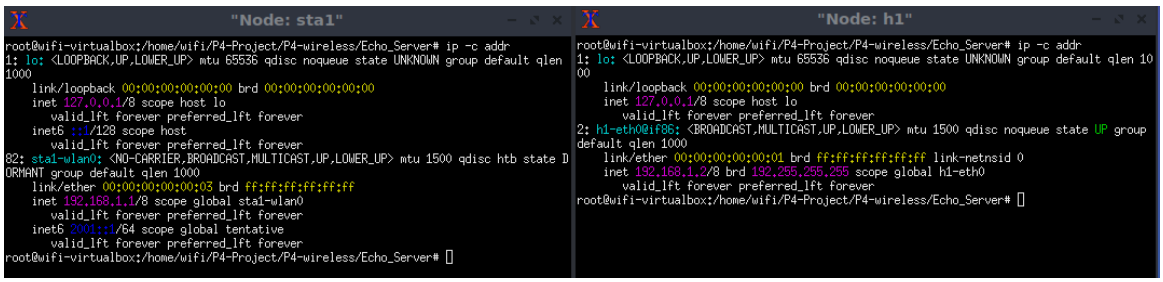


Figure 4.43: IP addressing of sta1 and h1

Next, we successfully conduct a ping (echo) test from wireless host1 [sta1] to host1 [h1], and the results including Wireshark are displayed in the Figure below.

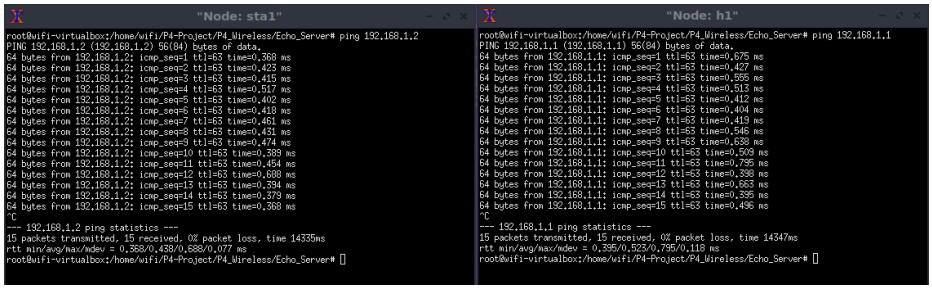


Figure 4.44: Testing echo ping

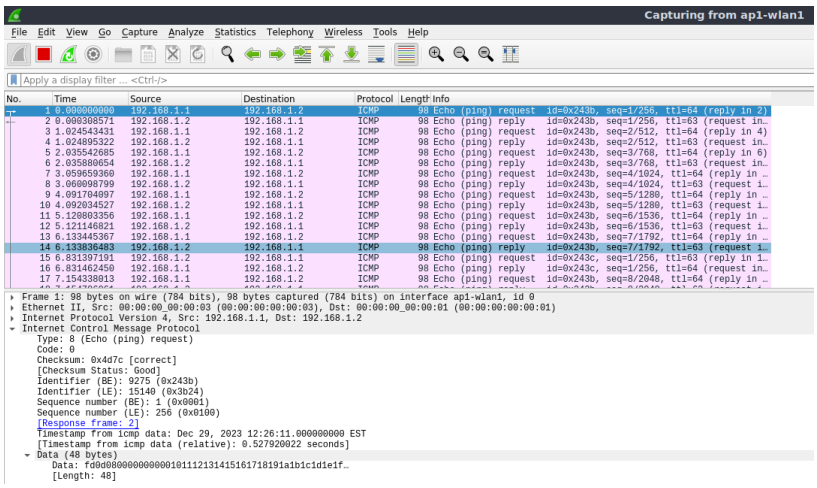


Figure 4.45: Capturing packets in P4RuntimeAP (ap1)

Additionally, we can examine the build, logs and pcaps directories generated at the start of the scenario to check the scenario.

Code 4.53: Function check - Echo Server

```
# Monitor the logs of P4RuntimeAP (ap1)
less logs/ap1.log
```

Code 4.54: Function check - Echo Server

```
# View JSON file
less build/echo_server.json
```

Code 4.55: Function check - Echo Server

```
# Monitor pcap file
wireshark pcaps/ap1-eth1-out.pcap
```

4.2.5 L3 Forwarding

In this use case, our aim is to implement forwarding at the network layer (layer 3), transforming our 'AP' into a basic router with minimal functionalities. Similar to prior cases, we will utilize the BMv2 switch software to test P4 programs and the established integration with Mininet-WiFi for network topology recreation. Once again, BMv2 configuration will be done via P4Runtime, necessitating the use of the P4RuntimeAP class.

All the details about this use case have been previously covered in **L3 Forwarding** of wired P4 (refer to subsubsection 4.1.4), and there are no significant differences introduced in the environment change, as mentioned earlier. Therefore, only instructions on how to compile and execute it will be provided. It is important that the Mininet-WiFi dependencies supporting BMv2 are properly installed to replicate this use case.

Compilation

Similar to earlier cases, a Makefile has been provided for compiling this use case, removing the necessity for direct use of the p4c compiler by the user. For detailed insights into the compilation process, stages involved, JSON generation, its injection into BMv2, or various targets based on architecture, we suggest referring back to **Drop** case of wired media (4.1.1). Follow the provided steps to complete the compilation process.

Code 4.56: Compilation P4 program - L3 Forwarding

```
# Entering the directory
cd P4-Project/P4-wireless/L3_Forwarding
# Using the Makefile
sudo make
```

Set up the stage

Similar to the compilation process, a Python script is provided to automatically set up the scenario and define the topology for this particular use case. To begin this use case, navigate to the directory associated with the desired use case configuration and execute the following command.

Code 4.57: Starting the scenario - L3 Forwarding

```
# Under the directory P4-Project/P4-wireless/L3_Forwarding
sudo python scenario.py
```

Function check

Once the execution, the script will establish the network layout outlined in Figure [4.46], comprising three hosts and three instances of P4RuntimeAP nodes.

- ”ap1” as a P4RuntimeAP, will possess three interfaces - one wireless (sta1), one to ”ap2” and one connected to ”ap3”.
- ”ap2” as a P4RuntimeAP, will possess three interfaces - one wireless (sta2), one to ”ap1” and one connected to ”ap3”.
- ”ap3” as a P4RuntimeAP, will possess three interfaces - one wireless (sta3), one to ”ap1” and one connected to ”ap2”.

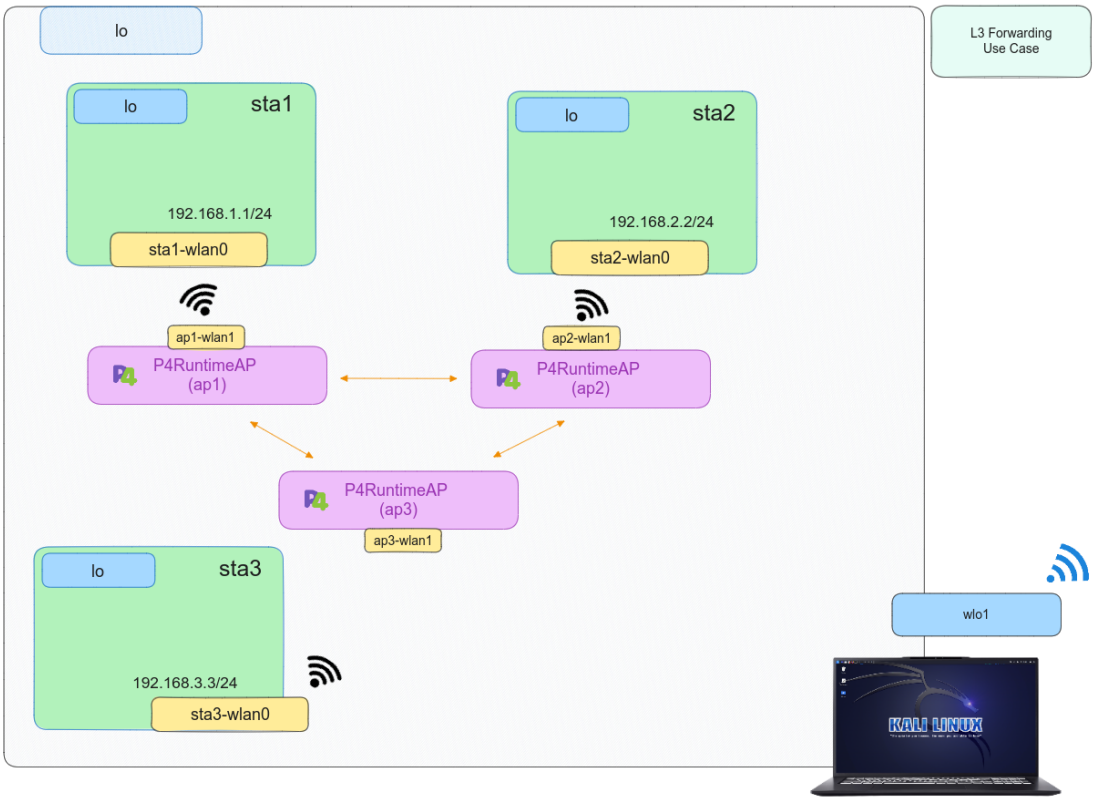


Figure 4.46: Topology (L3 Forwarding)

To assess the function of the use case, the Mininet Command Line Interface (CLI) will be accessible, we will open the xterm terminals for three wireless host of sta1, sta2 and sta3.

Code 4.58: Function check - L3 Forwarding

```
mininet > xterm sta1 sta2 sta3
```

After executing the script for this use case, we will first verify the IP addresses of the hosts in order to conduct ping tests between hosts. The Figure below displays the IP addresses of the hosts.

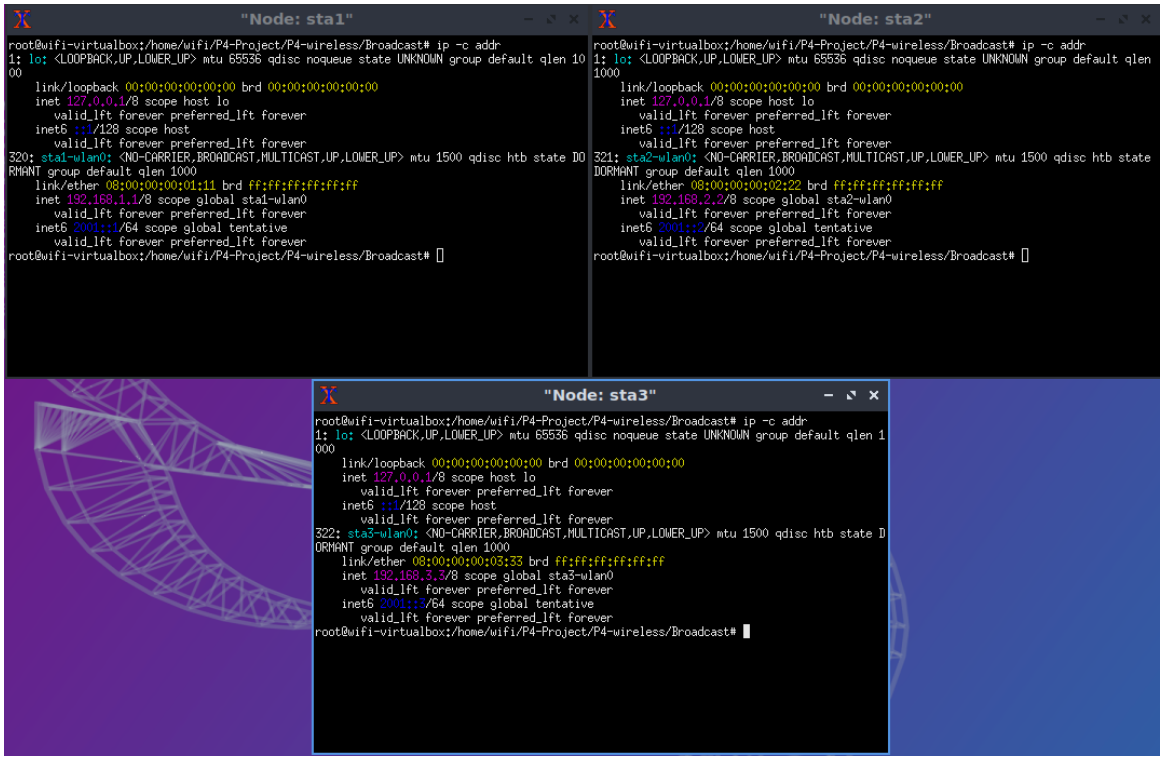


Figure 4.47: IP addressing of sta1, sta2 and sta3

We test ping communication between hosts connected to three P4RuntimeAP nodes. In the initial test, we ping from wireless host1 [sta1], connected to P4RuntimeAP (ap1) to wireless host2, connected to P4RuntimeAP (ap2). The following Figure [4.48] demonstrates the successful delivery of packets from sta1 to sta2 using L3 forwarding.

As you're aware, L3 forwarding eliminates the need for broadcast-based packet delivery between nodes. Each node maintains a routing table, which reduces latency when forwarding packets between P4RuntimeAPs. In the two Figures below, the first Figure [4.49] displays ICMP results between sta1 and sta2 captured in P4RuntimeAP (ap1). In contrast, the second Figure [4.50] illustrates empty packets captured in P4RuntimeAP (ap3). This is because the routing table of P4RuntimeAp (ap1) indicates that the destination is in P4RuntimeAP (ap2) node but not in APRuntimeAP (ap3) node.

```
"Node: sta1"
root@wifi-virtualbox:/home/wifi/P4-Project/P4-wireless/Broadcast# ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=0.344 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.605 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=0.360 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=64 time=0.310 ms
64 bytes from 192.168.2.2: icmp_seq=5 ttl=64 time=0.574 ms
64 bytes from 192.168.2.2: icmp_seq=6 ttl=64 time=0.163 ms
64 bytes from 192.168.2.2: icmp_seq=7 ttl=64 time=0.152 ms
64 bytes from 192.168.2.2: icmp_seq=8 ttl=64 time=0.326 ms
64 bytes from 192.168.2.2: icmp_seq=9 ttl=64 time=0.333 ms
64 bytes from 192.168.2.2: icmp_seq=10 ttl=64 time=0.356 ms
64 bytes from 192.168.2.2: icmp_seq=11 ttl=64 time=0.288 ms
64 bytes from 192.168.2.2: icmp_seq=12 ttl=64 time=0.124 ms
64 bytes from 192.168.2.2: icmp_seq=13 ttl=64 time=0.326 ms
64 bytes from 192.168.2.2: icmp_seq=14 ttl=64 time=0.132 ms
64 bytes from 192.168.2.2: icmp_seq=15 ttl=64 time=0.374 ms
^C
--- 192.168.2.2 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14350ms
rtt min/avg/max/mdev = 0.124/0.317/0.605/0.136 ms
root@wifi-virtualbox:/home/wifi/P4-Project/P4-wireless/Broadcast#
```

```
"Node: sta2"
root@wifi-virtualbox:/home/wifi/P4-Project/P4-wireless/Broadcast# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.399 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.386 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.150 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.520 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=0.356 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=0.631 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=0.330 ms
64 bytes from 192.168.1.1: icmp_seq=9 ttl=64 time=0.435 ms
64 bytes from 192.168.1.1: icmp_seq=10 ttl=64 time=0.645 ms
64 bytes from 192.168.1.1: icmp_seq=11 ttl=64 time=0.349 ms
64 bytes from 192.168.1.1: icmp_seq=12 ttl=64 time=0.142 ms
64 bytes from 192.168.1.1: icmp_seq=13 ttl=64 time=0.203 ms
64 bytes from 192.168.1.1: icmp_seq=14 ttl=64 time=0.341 ms
64 bytes from 192.168.1.1: icmp_seq=15 ttl=64 time=0.325 ms
^C
--- 192.168.1.1 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14354ms
rtt min/avg/max/mdev = 0.142/0.374/0.631/0.150 ms
root@wifi-virtualbox:/home/wifi/P4-Project/P4-wireless/Broadcast#
```

Figure 4.48: Test ping result between hosts

Capturing from ap1-wlan1

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=1/256, ttl=64 (reply in 2)
2	0.000248544	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=1/256, ttl=64 (request in 1)
3	1.004023716	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=2/512, ttl=64 (reply in 4)
4	1.004273136	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=2/512, ttl=64 (request in 3)
5	2.012648550	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=3/768, ttl=64 (reply in 6)
6	2.013029340	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=3/768, ttl=64 (request in 5)
7	3.040262626	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=4/1024, ttl=64 (reply in 8)
8	3.040638422	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=4/1024, ttl=64 (request in 7)
9	4.061628906	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=5/1280, ttl=64 (reply in 10)
10	4.061669484	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=5/1280, ttl=64 (request in 9)
11	5.087795174	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=6/1536, ttl=64 (reply in 12)
12	5.088061808	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=6/1536, ttl=64 (request in 11)
13	6.108362812	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=7/1792, ttl=64 (reply in 14)
14	6.108601645	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=7/1792, ttl=64 (request in 13)
15	7.134657070	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=8/2048, ttl=64 (reply in 16)
16	7.134902614	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=8/2048, ttl=64 (request in 15)
17	8.160839208	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=9/2304, ttl=64 (reply in 18)
18	8.161184073	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=9/2304, ttl=64 (request in 17)
19	9.184504786	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=10/2560, ttl=64 (reply in 20)
20	9.184774420	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=10/2560, ttl=64 (request in 19)
21	10.216607642	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=11/2816, ttl=64 (reply in 22)
22	10.216873137	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=11/2816, ttl=64 (request in 21)
23	11.229582313	192.168.1.1	192.168.2.2	ICMP	98	Echo (ping) request id=0x5cce, seq=12/3072, ttl=64 (reply in 24)
24	11.229955440	192.168.2.2	192.168.1.1	ICMP	98	Echo (ping) reply id=0x5cce, seq=12/3072, ttl=64 (request in 23)

```
▸ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface ap1-wlan1, id 0
▸ Ethernet II, Src: 08:00:00:00:01:11 (08:00:00:00:01:11), Dst: 08:00:00:00:02:22 (08:00:00:00:02:22)
▸ Internet Protocol Version 4, Src: 192.168.1.1, Dst: 192.168.2.2
▾ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xe0c0 [correct]
    [Checksum Status: Good]
    Identifier (BE): 23758 (0x5cce)
    Identifier (LE): 52828 (0xce5c)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [Response frame: 2]
    Timestamp from icmp data: Dec 30, 2023 07:28:27.000000000 EST
    [Timestamp from icmp data (relative): 0.993980315 seconds]
  ▸ Data (48 bytes)
```

Figure 4.49: Capturing ICMP in P4RuntimeAP (ap1)

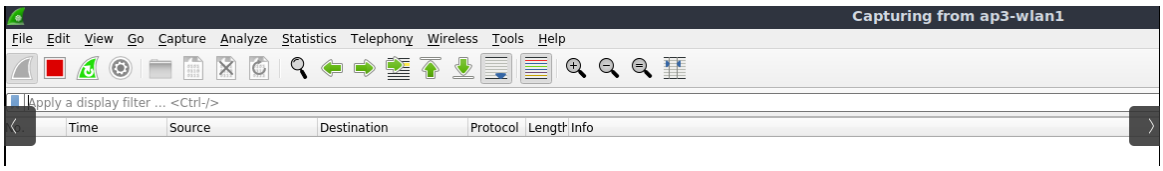


Figure 4.50: No captured packet in P4RntimeAP (ap3)

Furthermore, we conduct a similar test between P4RuntimeAP (ap1) and P4RuntimeAP (ap3) with results mirroring the previous scenario. The ping test between wireless host1 [sta1] in P4RuntimeAP (ap1) and wireless host3 [sta3] in P4RuntimeAP (ap3) is also successful (see in Figure [4.51]).

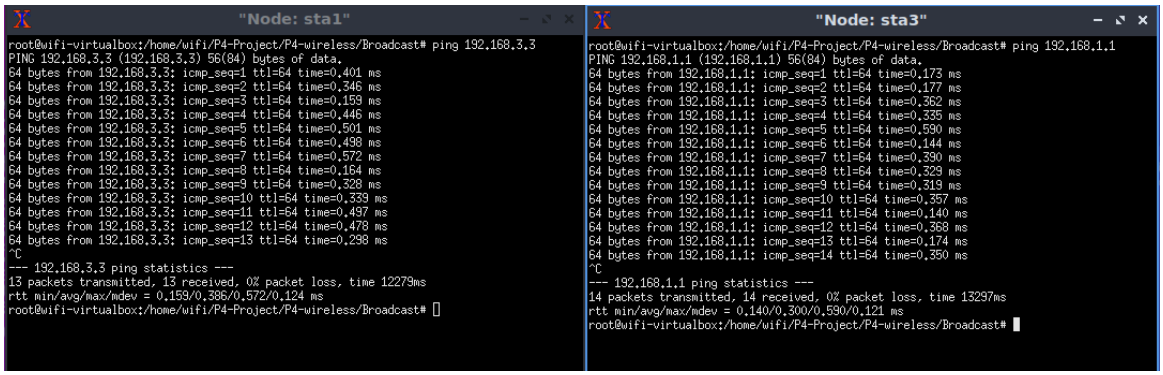


Figure 4.51: Test ping result between hosts

The captured packet results confirm that the packets did not traverse to the P4RuntimeAP (ap2) due to the routing table configuration in P4RuntimeAP (ap1). Consequently, no packets were captured from P4RuntimeAP (ap2). The Figures below illustrate the packet capture results.

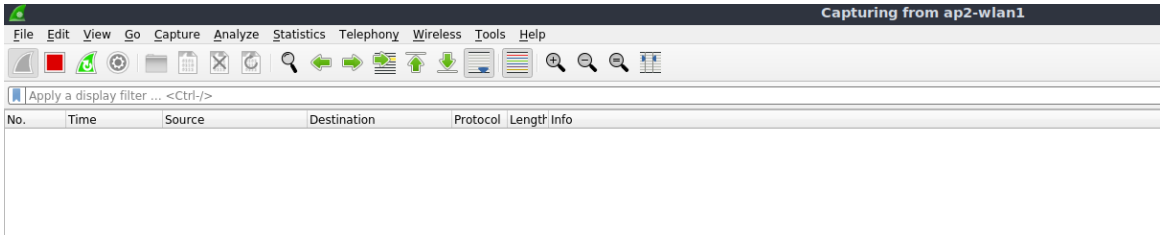


Figure 4.52: No captured packet in P4RntimeAP (ap3)

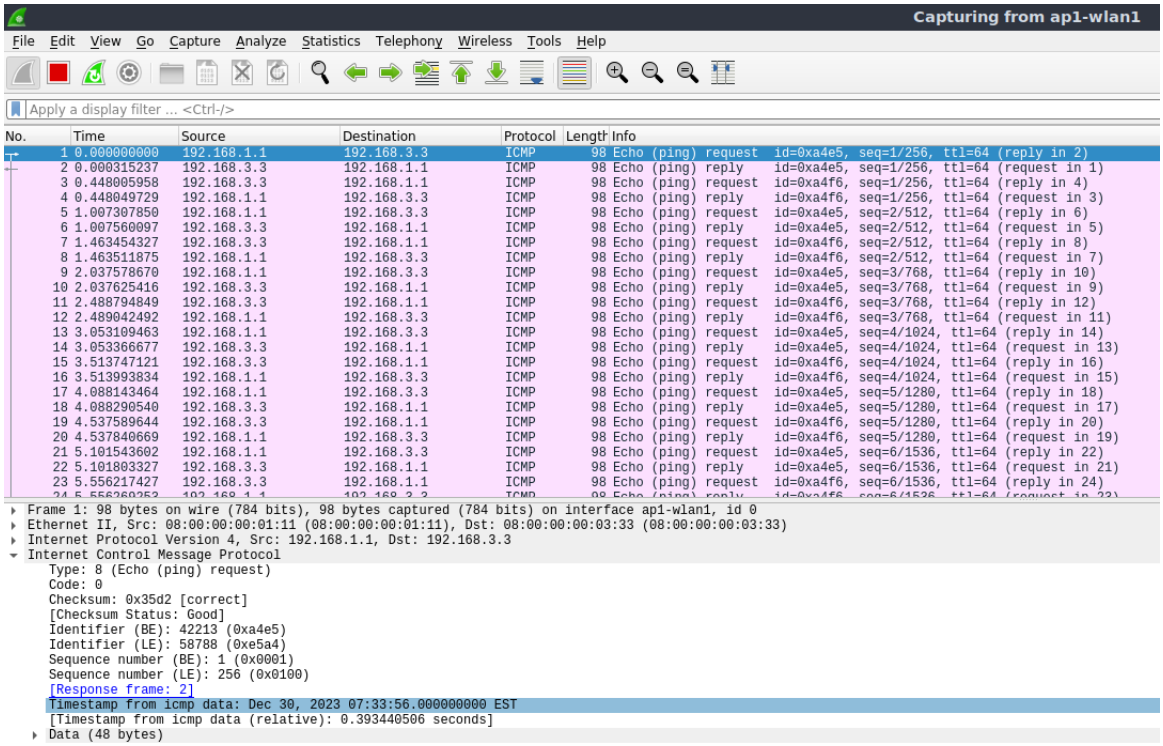


Figure 4.53: Capturing ICMP in P4RuntimeAP (ap1)

Moreover, we have the option to inspect the directories created at the beginning of the scenario, namely the build, logs, and pcaps directories, to assess the scenario.

Code 4.59: Function check - L3 Forwarding

```
# Monitor the logs of P4RuntimeAP (ap2)
less logs/ap2.log
```

Code 4.60: Function check - L3 Forwarding

```
# View JSON file
less build/l3_forwarding.json
```

Code 4.61: Function check - L3 Forwarding

```
# Monitor pcap file
wireshark pcaps/ap2-eth2_in.pcap
```

4.2.6 Broadcast

In this test, we will create a p4 program for broadcasting, specifically at the link level (layer 2). Similar to prior cases, the BMv2 software switch will be employed to test the P4 programs, along with the established integration with Mininet-WiFi to replicate the network topologies.

This scenario was previously explained in the '**Broadcast**' of wired P4 (see subsubsection 4.1.5). Since no significant changes occur in the environment, only guidance on compiling and running the program will be given. It's important to ensure that the Mininet-WiFi dependencies supporting BMv2 are correctly installed to replicate this scenario.

Compilation

For the compilation of this use case, as in the previous use cases, a Makefile has been prepared, therefore it is not necessary for the user to make direct use of the p4c compiler. If you want to know more about how the compilation process works, what stages there are, how the generated json is injected into BMv2, or what different targets there are depending on the architecture, we recommend you to go back to **Drop** (4.1.1). Follow the provided steps to complete the compilation process.

Code 4.62: Compilation P4 program - Broadcast

```
# Entering the directory
cd P4-Project/P4-wireless/Broadcast
# Using the Makefile
sudo make
```

Set up the stage

As the compilation procedure, there's a Python script available to automate scenario setup and define the topology for this specific use case. To initiate this use case, go to the directory linked to the desired configuration for this use case and run the given command.

Code 4.63: Starting the scenario - Broadcast

```
# Under the directory P4-Project/P4-wireless/Broadcast
sudo python scenario.py
```

Function check

After the execution has defined, this will establish the network layout outlined in Figure [4.54], comprising three hosts and a instances of P4RuntimeAP node.

- ”ap1” as a P4RuntimeAP, will possess three wireless interfaces - sta1, sta2 and sta2.

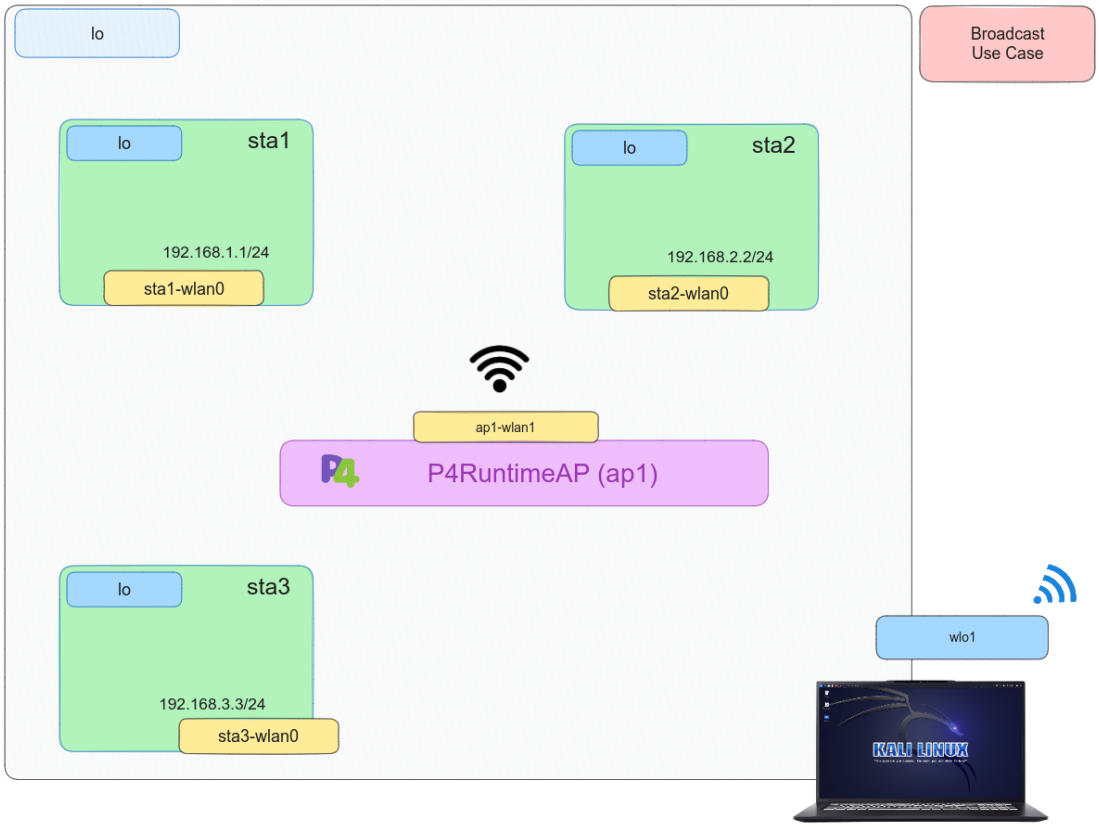


Figure 4.54: Topology (Broadcast)

To assess the function of the use case, the Mininet Command Line Interface (CLI) will be accessible, we will open the xterm terminals for three wireless hosts of sta1, sta2 and sta4.

Code 4.64: Function check - L3 Forwarding

```
mininet > xterm sta1 sta2 sta3
```

Before conducting the test, we need to verify the IP addresses of each host. Here are the IP addresses of the hosts.

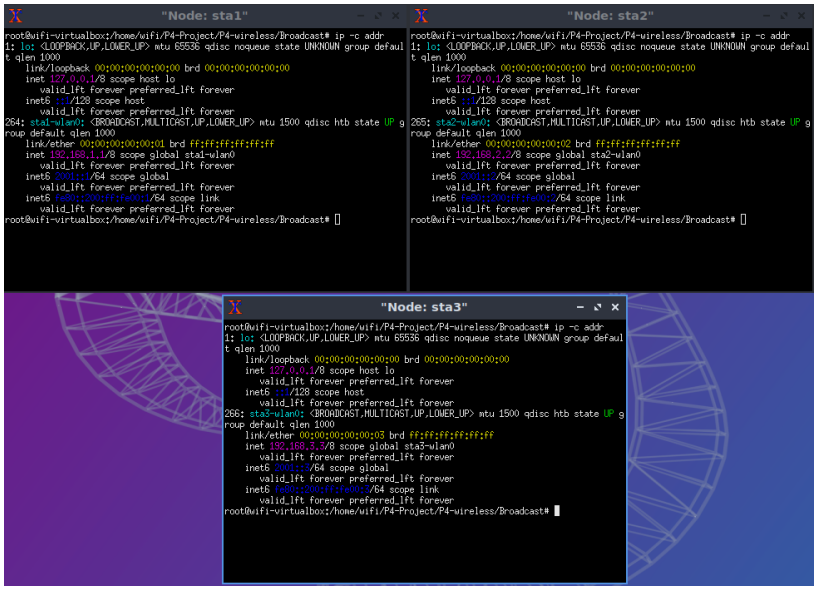


Figure 4.55: IP addressing of sta1, sta2 and sta3

Next, we conduct a ping test from wireless host1 [sta1] to wireless host2 [sta2], from wireless host1 [sta1] to wireless host2 [sta2] and then from wireless host2 [sta2] to wireless host3 [sta3]. This result in successful packet transmission between them. The following Figure show the ping testing result between hosts.

```
*** Adding stations/hosts
*** Adding P4APs
*** Creating links
*** Starting network
*** Starting controller(s)

*** Starting L2 nodes
ap1 .⚡  simple_switch_grpc @ 267682 thrift @ 60825

*** Running CLI
*** Starting CLI:
mininet-wifi> sta1 ping -c 1 sta2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=0.473 ms

--- 192.168.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.473/0.473/0.473/0.000 ms
mininet-wifi> sta1 ping -c 1 sta3
PING 192.168.3.3 (192.168.3.3) 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=0.318 ms

--- 192.168.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.318/0.318/0.318/0.000 ms
mininet-wifi> sta2 ping -c 1 sta3
PING 192.168.3.3 (192.168.3.3) 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=0.292 ms

--- 192.168.3.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.292/0.292/0.292/0.000 ms
mininet-wifi>
```

Figure 4.56: Test ping result between hosts

Additionally, we have the choice to examine the directories formed at the start of the scenario, specifically the build, logs, and pcaps directories, to evaluate the scenario.

Code 4.65: Function check - Broadcast

```
# Monitor the logs of P4RuntimePA (ap1)
less logs/ap1.log
```

Code 4.66: Function check - Broadcast

```
# View JSON file
less build/broadcast.json
```

Code 4.67: Function check - Broadcast

```
# Monitor pcap file
wireshark pcaps/ap1-wlan0-in.pcap
```

Conclusion

As we conclude this thesis, it is important to acknowledge our journey of exploration, learning and creativity. The fusion of SDN and IoT, empowered by the capabilities of P4 programming, promises to reshape the landscape of network management. It is worth noting that while most scenarios have proven successful, we recognize that challenges may persist, as highlighted by the issue encountered in the ”Pass” use case in both wired and wireless environments.

In the ever-evolving world of technology, our work signifies not just an endpoint but a stepping stone towards a future where the synergy of SDN, IoT, and P4 will play a pivotal role in shaping the network management landscape. The adventure continues, and we look forward to the ongoing efforts of researchers and professionals in this field to build upon our findings and drive innovation in the integration of IoT within SDN environments.

Bibliography

[1] Rohit Kumar Das, Fabiola Hazel Pohrmen, Arnab Kumar Maji, and Goutam Saha. Ft-sdn: a fault-tolerant distributed architecture for software defined network. *Wireless personal communications*, 114:1045–1066.

[2] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4):2347–2376.

[3] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals. *Advances, and Applied Research. arXiv*, 2101.

[4] Frank HP Fitzek, Fabrizio Granelli, and Patrick Seeling. *Computing in Communication Networks: From Theory to Practice*. Academic Press.

[5] Vanbevar, Laurent. Advanced Topics in Commnication Networks, 2022. URL: https://video.ethz.ch/lectures/d-itet/2022/autumn/227-0575-00L/c1df0f1b-d89b-4328-b9d7-7dfd26a5bb46.html.

[6] Ramon R. Fontes, Samira Afzal, Samuel H. B. Brito, Mateus A. S. Santos, and Christian Esteve Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 384–389, 2015.

[7] Arata Kato, Mineo Takai, and Susumu Ishihara. Design and implementation of a wireless network tap device for ieee 802.11 wireless network emulation. In *2017 Tenth International Conference on Mobile Computing and Ubiquitous Network (ICMU)*, pages 1–6, 2017.

[8] Thomas D Nadeau and Ken Gray. *SDN: Software Defined Networks: An authoritative review of network programmability technologies*. " O'Reilly Media, Inc.", 2013.

[9] David Hanes, Gonzalo Salgueiro, Patrick Grossetete, Robert Barton, and Jerome Henry. *IoT fundamentals: Networking technologies, protocols, and use cases for the internet of things*. Cisco Press.

[10] P4 Open Source Programming Language. P4. URL: https://p4.org/.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[12] An Instant Virtual Network on your Laptop. Mininet. URL: http://mininet.org/.

[13] Introduction to Mininet. Mininet. URL: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet.

[14] Ramon Fontes, Christian Rothenberg. Wireless network emulation with mininet-wifi, 2019. URL: https://mininet-wifi.github.io/book/.

[15] Ramon Fontes. About ramon fontes. URL: https://github.com/ramonfontes.

[16] David Carrascal, Elisa Rojas, Joaquin Alvarez-Horcajo, Diego Lopez-Pajares, and Isaías Martínez-Yelmo. Analysis of p4 and xdp for iot programmability in 6g and beyond. *IoT*, 1(2):605–622.

[17] Introducing JSON. JSON standard. URL: https://www.json.org/.

[18] GitHub. Github. URL: https://github.com/.

[19] Pilar Manzanares-Lopez, Juan Pedro Muñoz-Gea, and Josemaria Malgosa-Sanahuja. Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry. *IEEE Access*, 9:20391–20409, 2021.

[20] P4Lang Organization. P4 language. URL: https://github.com/p4lang.

[21] wireshark.org. Wireshark. URL: https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html.

[22] Network Working Group. Internet control message protocol. URL: https://www.rfc-editor.org/rfc/rfc792.html.

[23] Network Working Group. Ip protocol. URL: https://www.rfc-editor.org/rfc/rfc791.html#section-1.

[24] Network Working Group. Internet protocol, version 6 (ipv6) specification. URL: https://www.rfc-editor.org/rfc/rfc1883.html.

List of Acronyms

AP	Access Point
APIs	Application Programming Interfaces
ARP	Address Resolution Protocol
BMv2	Behavioral Model version 2
cgroups	Control Groups
CLI	Command Line Interface
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
GIT	Global Information Tracker
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LPWA	Low-Power Wide-Area
MAC	Media Access Control
P4	Programming Protocol-independent Packet Processor
PISA	Protocol-Independent Switching Architecture
RFC	Request for Comments

SDN	Software Defined Network
TTL	Time To Live
veth	Virtual Ethernet

APPENDIX A

Appendix

1. Specification of physical machine

This appendix includes specifications and depedencies of installing P4 projects and equipment. Below are all the materials that have been used in the project.

Processor: 11th Gen Intel i7-1165G7 (8) @

Memory: 16GB LPDDR4

Graphic: Intel TigerLake-LP GT2 [Iris Xe]

Operating System: Ubuntu 22.04.2 LTS x86-64

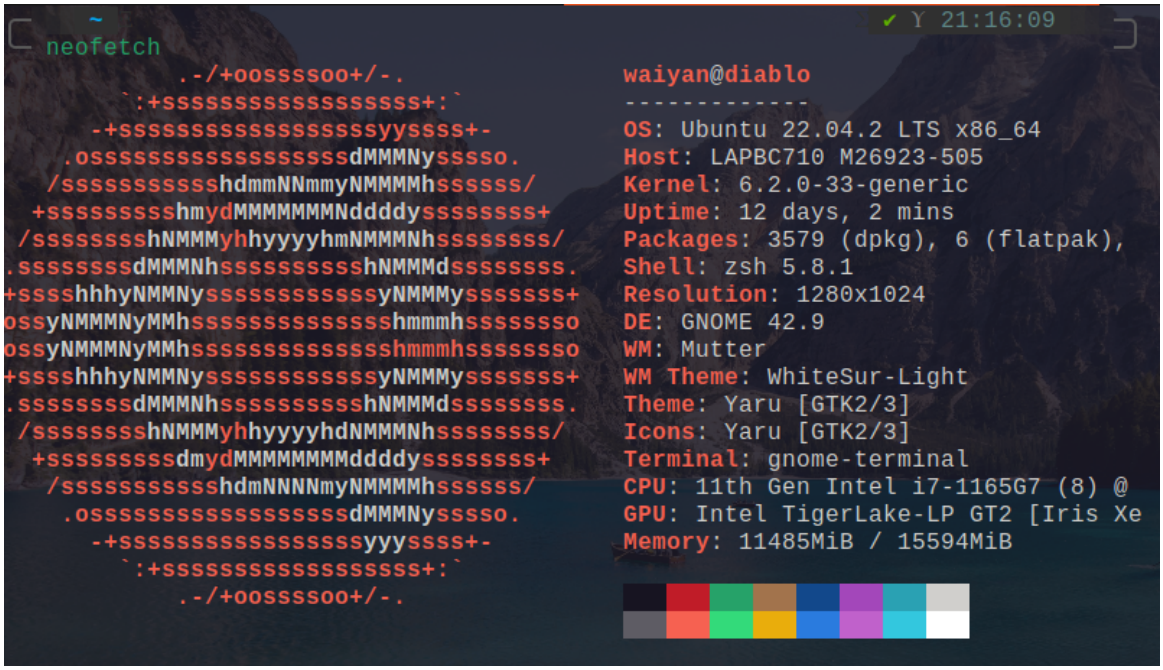


Figure A.1: Specification of physical machine

1. Specifications of virtual machine and installation P4

1.1 P4 installation in wired media

We have described the specifications of virtual machine for P4 and how to deploy P4 on it for wired environment. When it comes to assessing the use cases, you have had two choices. You can choose to set up a virtual machine and install various dependencies manually for P4, or you can go for a pre-configured virtual machine that comes with all the required software pre-installed for P4:

- Installing P4 development tools on an existing system (Option 1)
- Utilizing release VM images that already contain P4 development tools (Option 2)

For our thesis, we have selected 'option 2' and utilized the 'Release' VM images (dated '2023-Aung-01') which updtaes every month for minor changes with the Ubuntu 20.04 operation system.

Table A.1: Specification of Testing

Date published	Operation system	Development VM Image Link	Tested working on Linux
2020-Aug-01	Ubuntu 20.04	3.0 GByte VM image	Ubuntu 22.04.2 LTS

Below Figure shows the specification of P4 virtual machine.

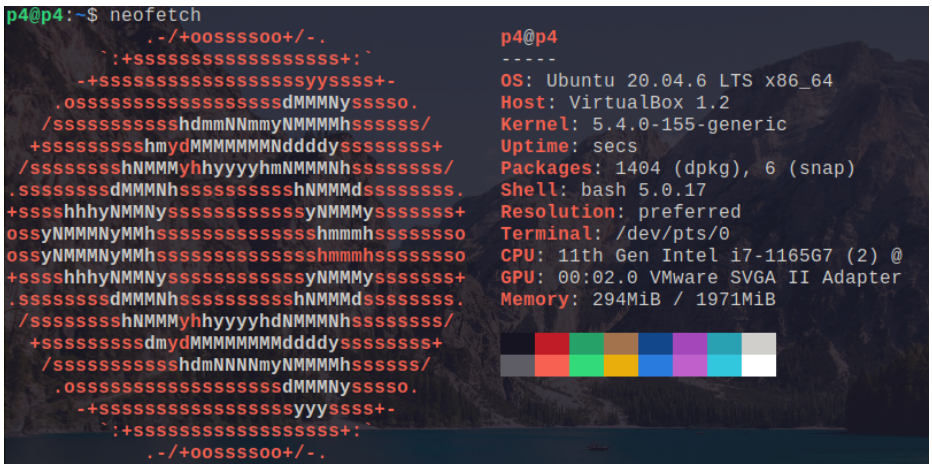


Figure A.2: Specification of virtual machine

1.2 P4 installation in wireless media

We have described the specifications of virtual machine for P4 and how to deploy P4 on it for wireless environment. When it comes to assessing the use cases, you have had two choices. You can choose to set up a virtual machine and install various dependencies manually for P4, or you can go for a pre-configured virtual machine that comes with all the required software pre-installed for P4:

- Installing P4 development tools on an existing system (Option 1)
- Utilizing release VM images that already contain P4 development tools (Option 2)

For our thesis, we have selected 'option 2' and utilized the 'Release' VM images which includes requirements of P4 for Mininet-WiFi.

Table A.2: Specification of Testing

Operation system	Development VM Image Link	Tested working on Linux
Lubuntu 20.04	6.7 GByte VM image	Ubuntu 22.04.02 LTS

Below Figure shows the specification of P4 virtual machine.

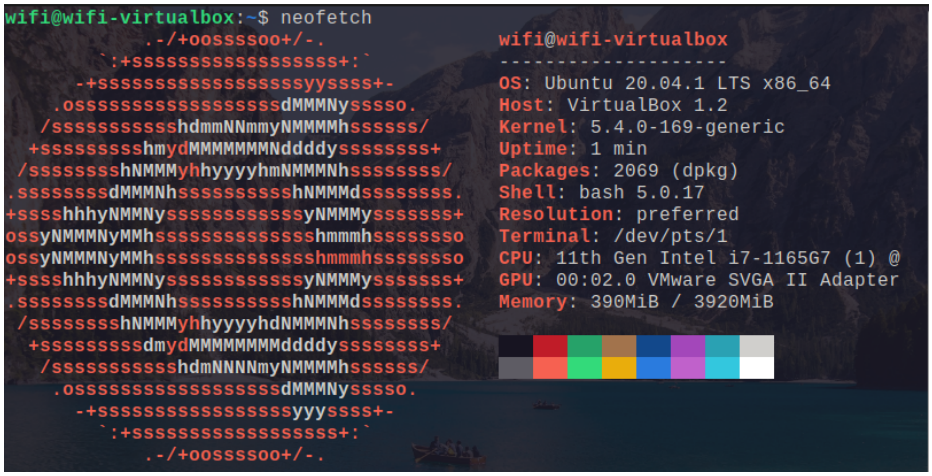


Figure A.3: Specification of virtual machine