Mesana Adam

Develop a Switch Learning Module on Floodlight Controller in Software Defined Networking (SDN)

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

26 April 2025

Abstract

Author:	Mesana Adam
Title:	Develop a Switch Learning Module on Floodlight Controller in SDN
Number of Pages:	48 pages
Date:	26 April 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart IoT Systems and Cloud Computing
Supervisors:	Janne Salonen, Principal Lecturer

In traditional existing networks, conducting real-world experiments on large scale production networks presents significant challenges. These networks have remained relatively unchanged for years, with slow advancements in innovation. Traditional networks rely heavily on vendor specific software and operate as closed systems. This “inside the box” approach has limited the flexibility and scalability of the network, leading to the proposal of a new network architecture.

Software defined Networking (SDN) represents a transformative paradigm designed to enhance network management by enabling centralized, flexible, and programmable control of the network. The fundamental principle is the separation of the control plane from the data plane which includes the underlying network infrastructure.

This thesis project aimed to develop a software defined networking application using the Floodlight controller as a main controller to implement flow rules on Open vSwitch (OVS). The focus is on achieving connectivity between hosts at the layer 2 (data link layer) of the network. Additionally, the thesis aims to gain knowledge with SDN technology by studying its architecture, components, and underlying functionality, and taking the initial step of setting up the environment and developing an SDN module that operates on top of the controller. The application development platform includes VMware running Ubuntu Linux, with Eclipse IDE used for building the SDN application in Java on top of the Floodlight controller, and Mininet acting as an emulator for the networking infrastructure.

The primary aim of this project was achieved, establishing connectivity at the data link layer.

Keywords: SDN, Floodlight controller, OpenFlow, Open vSwitch, Data Link Layer, Mininet, Eclipse IDE, Control plane, Data plane

Contents

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
L2	Layer 2 of the network protocol stack
L3	Layer 3 of the network protocol stack
L4	Layer 4 of the network protocol stack
MAC	Media Access Control
OS	Operating System
OVS	Open vSwitch
QoS	Quality of Service
SDN	Software Defined Networking
TCP	Transmission Control Protocol

UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine

1 Introduction

In earlier times, IT resources such as storage, computing, and networking were managed separately. The systems controlling these resources were often isolated in different locations, and strict access policies were enforced for security purposes. However, the demand for affordable and integrated computing, storage, and networking resources required organizations to integrate these elements in data centers. Over time, as data centers evolved, these departmental servers migrated to centralized facilities for easier management and the ability to share resources across the enterprise. This transformation was further accelerated by VMware, which revolutionized computing with programs that create virtualized environments, allowing several virtual machines to operate on one physical server. Initially designed for engineers to run both Linux and Windows on the same machine. VMware's technology allowed virtual operating systems to behave like large files that could be manipulated, moved, copied, and even paused, entering a suspended state when not in use. The technological flexibility uncovered challenges related to resource utilization and operational efficiency. IT departments often purchased resources in advance to meet predicted demand, leading to unused equipment consuming power and space. Amazon, facing rapid business growth, encountered this issue and created Amazon Web Services (AWS) to utilize idle resources. AWS allowed Amazon to rent out excess capacity to external users, providing a more dynamic and efficient use of computing, storage, and networking power [1, pp. 1-3].

While data centers were advancing, network devices largely remained unchanged, focusing on speed rather than flexibility. Network devices come with a management interface for configuration and management. However, their functionality was restricted by the device firmware, making the process of configuring and managing network devices still complex. Networking equipment vendors controlled both the hardware and software of their devices, making it difficult for enterprises to test and deploy with new routing protocol technologies without waiting years for vendors to implement requested features [1].

In traditional networking architecture, the management and operation of networks are closely tied to the hardware device, such as routers and switches. The hardware networking devices perform both the data forwarding (data plane) and the decision making (control plane) functions within a distributed architecture. While this operating approach has been the backbone of network infrastructure for many years, it imposes several limitations that restrict the flexibility and expansion needed for modern networks. Recognizing this gap, new approaches to networking began to emerge that are changing the way the traditional model for operating and controlling networks works. Software Defined Networking (SDN) is a new network operating model that uses a centralized architecture. A significant disadvantage of traditional networks is that network architectures have remained relatively static over time, which has contributed to a slower rate of innovation. Another issue is the reliance on closed, vendor specific systems. Each network device typically comes bundled with proprietary software. For example, a router from Cisco runs Cisco specific software, while a switch from Juniper operates using Juniper's software. This software is designed to work only with the vendor's own hardware, creating a closed system.

Traditional networks adopt an "inside the box" approach and follow a rigid methodology, where network devices are managed and optimized individually rather than as part of an integrated system. This lack of centralization limits the ability to dynamically adapt or program the network, making it difficult to respond to modern demands. Furthermore, routing and switching hardware was considered excessively expensive. Simultaneously, the price of scalable computing resources was quickly dropping, enabling access to thousands of processors. This understanding led to the idea of utilizing this processing capability to operate a centralized control plane [1].

Software-Defined Networking (SDN) is considered as a paradigm shift in the way networking is structured and managed. Offering enhanced flexibility, scalability, and ease of management. Its rapid adoption led major tech companies such as Google, Facebook, and others to establish and fund the Open Networking Foundation (ONF), aiming to promote SDN adoption through the development of open standards [2, pp. 15]. In 2012 at the Open Networking Summit (ONS),

Google revealed that it uses SDN to interconnect its data centers, making its wide area network (WAN) reliant on this innovative technology [3].

SDN centralizes the control of networks making use of a controller. For instance, the controller can perform all control plane functions, replacing the control plane that was integrated in each networking device. OpenFlow is the first standard protocol that was developed by a group of engineers from Stanford University, designed specifically for devices with only data planes. This open interface provides instructions to networking devices from a centralized controller [4, pp. 7-10]. This separation in SDN enables networks to become more flexible and programmable, fostering a more innovative and adaptable architecture.

The objective of this thesis project was to develop an SDN application to ensure connectivity between hosts at layer 2. This served as an initial step to gain hands-on experience with SDN technology and to create and test my own Java algorithm across various network topologies. Additionally, the project aimed to deepen my understanding of the SDN and to build practical skills in working with this technology.

2 Key Features of Software Defined Networking

The primary function of computer networks is to transfer data between end devices. Networks operate effectively because devices and software adhere to established rules. These rules are defined as standards and protocols, which outline specific agreements on how different aspects of the network should function [5, pp. 12-15]. Networks are difficult to manage, and they are composed of various hardware devices, each performing different roles in the network. Devices include switches, routers, and middleboxes such as firewalls, network address translators (NATs), and load balancers [6]. In both traditional networking and Software Defined Networking, networks maintain the process of forwarding data, but the methods used have changed [7, pp. 358-363]. Software defined networking represents a new way of managing networks that seeks to deal with flaws in the existing traditional model. Shifting from a distributed architecture to a

centralized and manageable architecture that is more appropriate for the very large networks that are common in today's large data centers. This section of the thesis describes the key features of Software Defined Networking.

2.1 Decoupling the Control and Data Planes

A fundamental feature of Software Defined Networking is the division between data plane and control plane [8]. The data plane directs traffic from the ingress port to the appropriate egress port on a device. It is responsible for handling incoming network traffic, by processing them based on attributes such as IP addresses, VLAN ID, and MAC addresses, determining actions such as forwarding, dropping, copying, or modifying incoming packets [2]. The control plane, responsible for managing and programming the data plane's operations. It determines the forwarding state used by the data plane to direct packet traffic. The data plane and the control plane serve distinct purposes and are structurally different, each designed to perform specific functions within the network. In contrast to traditional networks, where each device independently manages its own control plane. SDN centralizes this control using a central controller, removing it from the individual networking device.

2.2 Controller-Based Management and Lightweight Devices

SDN simplifies hardware devices by decoupling the data and control planes, allowing a central system to handle the management in a network. Instead of each networking device making independent decisions by executing complex programs that consist of extensive lines of code, SDN removes this complexity from individual devices and places it within a central controller [9]. The controller sends initial instructions to the lightweight devices in the network. These instructions provide guidelines on how to handle incoming packets. As a result, network devices can quickly process and forward packets based on these predefined rules, improving efficiency and reducing the complexity of individual devices.

2.3 Control Plane Abstractions

SDN outlines three key abstractions in the networking control plane, distributed state, forwarding, and specification [10]. These abstractions help simplify the traditionally complex tasks of network control plane, making networks easier to manage and program. The distributed state abstraction offers a high-level view of the network topology through a centralized controller. Instead of dealing with the individual network devices' states separately, programmers can focus on the overall network. Making decisions based on a unified perspective. The forwarding abstraction is required to conceal the complexities of low-level hardware and software, enabling programmers to specify how packets should be handled without needing to understand the specifics of a manufacturer's hardware. The approach to achieve this was through the OpenFlow protocol, a standard interface that provides access to the networking device and enables to store flow entries [10]. The specification abstraction, also known as configuration abstraction, is needed to simplify network configuration, allowing the programmer to define network goals without being burdened by the details of physical implementation for each networking device. In SDN, the central controller plays a crucial role by acting as the brain of the network. To enable automated network management, the controller provides open interfaces that allow communication between applications and network devices. Northbound and southbound interfaces are commonly used to differentiate between interfaces for applications (northbound) and devices (southbound). The OpenFlow protocol, a southbound API, allows the controller to configure network devices. Meanwhile, the northbound API connects the controller to application, facilitating the development of algorithms for efficient network management. Three main advantages for SDN application using the northbound interface include:

- Simplify network programming by using developer-friendly libraries such as REST, making it easier for developers to interact with the network. [11].

- Provides a high-level view of the network instead of dealing with individual nodes [11].

- Hides complex protocol details, making it easier to build SDN applications [11].

By using this approach, applications can be created to be compatible with a diverse variety of devices from various manufacturers, each featuring distinct operating system specifications.

2.4 Open Interfaces

SDN's software interfaces are open source, standardized, and clearly documented [12]. The standard-source interfaces provide researchers, and startups the opportunity to develop new approaches for various network functions. Researchers can benefit from the ease of piloting and testing new ideas, accelerating the pace of network technology development without being locked to proprietary systems [13]. This speed of network technology development fosters collaboration, leading to faster innovation and more effective solutions for current networking challenges.

3 SDN Architecture and Components

The architecture of software defined networking includes applications, controllers, and switches [14]. These components appear in three layers: application, control, and data layers, as shown in figure 1.

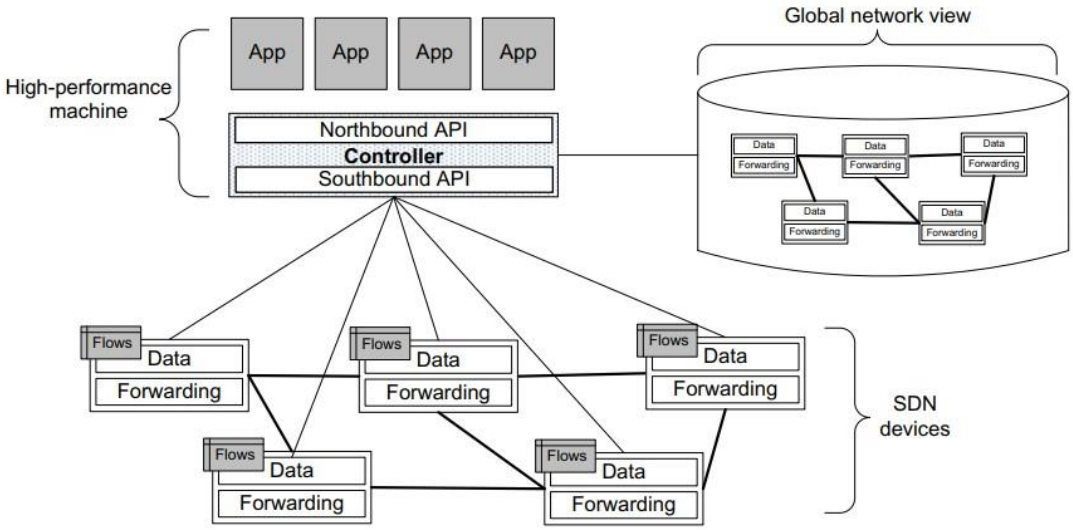

Figure 1. Showing the SDN architecture and components [13].

Switches (referred to as SDN Devices in the figure 1) are responsible for forwarding network packets based on decisions made by their flow tables, which consists of Flow Rules also called Flow Entries defined by the controller. A flow refers to a series of packets sent and received between two endpoints within a network and is identified by various attributes, such as IP addresses, TCP/UDP ports, Ingress ports and others. Each flow rule specifies the actions a switch must take for packets belonging to that flow, which operates in one-way manner. Once a packet is received, if it corresponds with a flow rule from the flow table, the switch will execute the appropriate action set by the controller. If no match is found, the packet will be removed or sent forward to the controller, depending on the switch and the version of the OpenFlow protocol that is being used [15].

The controller enables an SDN application to specify how the switch operates and assists the application to process packets received from the switch to the controller. The controller has a broad perspective of the network. Applications are

created on the controller and communicate with it using a northbound API. This interface can vary depending on the programming language used to develop the application. Applications can be classified into two types [13]:

- Proactive applications: flow rules in these types of applications are predefined and static. The controller sends flow rules to the switch ahead of time. They are then stored in the switch until changed by the application or other supporting application.
- Reactive applications: flow rules are generated in anticipation of certain events (such as packet arriving at the controller).

3.1 SDN Switches

One of the core components of an SDN based network is the SDN switch, which plays a critical role in forwarding packets based on the instructions provided by the controller. An SDN switch is composed of three key components, an API for communication with the controller, an abstraction layer represented by flow tables, and a packet-handling module that determines how packets are processed.

The API for controller communication serves as the interface between the switch and the SDN controller. The southbound API OpenFlow is the protocol used which enables the switch to receive instructions from the controller. OpenFlow allows the controller to install, modify, or delete flow rules dynamically in the switch. When an incoming packet arrives, the switch checks its flow tables to find a matching rule. If a match is found, the switch follows the action specified in that rule without needing further instructions from the SDN controller. In this case the switch independently handles the packet based on pre-installed flow rules. However, when no match rule is found in its flow tables, the switch sends the packet to the controller for further instructions. In this thesis, the open-source switch Open vSwitch (OVS) [16] is used due to its wide adoption.

3.2 SDN Controllers

The controller, through applications running on top of it, manages all network functions and makes decisions such as load balancing, forwarding, routing, and others. SDN controllers come equipped with several pre-built application modules that manage specific network tasks. Some common modules found in SDN controllers include the hub module, which provides basic packet forwarding, similar to how a traditional hub operates. Additionally, the learning switch module, which enables the controller to dynamically learn and store MAC addresses, allowing it to make forwarding decisions without requiring manual configuration. Other modules include a basic firewall module, a router module, and various additional components that enhance network performance and control.

Although a standardized northbound interface has not been established, various SDN controllers have implemented different versions of northbound interfaces. For example, the Floodlight controller offers a Java-based interface and a RESTful interface. allowing developers to create SDN applications efficiently. Table 1 summarizes popular open-source SDN controllers, their programming language, developers, and features, such as Graphical User Interface (GUI) availability.

Table 1. Summarizing popular open-source SDN controllers.

Controller	Programming language	Vendor	GUI
NOX [17]	C++/Python	Nicira Networks	Yes
POX [18]	Python	Nicira Networks	Yes
Maestro [19]	Java	Rice University	No
Beacon [20]	Java	Stanford University	Yes

RISE [21]	C and Ruby	NEC	No
Floodlight [22]	Java	Big Switch	Yes
RYU [23]	Python	Nippon Telegraph and Telephone	No
OpenDaylight [24]	Java	LF Networking	Yes

3.3 SDN Applications

The primary role of an SDN application is to execute its designated network functions, such as load balancing, firewalling, or other specified operations. These applications operate on top of the SDN controller and respond dynamically to network events. Once the controller has configured network hardware and provided the application with an overview of the network topology, the application mainly responds to events. The functionality of each SDN application varies, and its behaviour is influenced by events from the controller and external inputs, including network monitoring systems like NetFlow, Intrusion Detection Systems (IDS), or BGP peers. SDN applications register as listeners for specific events, and the controller calls application-specific procedures whenever such events occur [25].

3.4 OpenFlow Protocol

The OpenFlow protocol was introduced in 2008 [26] by a group of researchers who initially collaborated at Stanford University before its development was formalized under the non-profit organization OpenFlow.org. In 2009, the first stable version, OpenFlow 1.0.0, was released. The Open Network Foundation (ONF) later in 2011 adopted the OpenFlow protocol and took over the responsibility for managing and advancing its specifications [26].

OpenFlow establishes the communication mechanism between the control plane and the data plane, it is the first standardized interface created specifically for SDN. Starting with version v.1.1. OpenFlow defines a standardized method for communication between the controller plane and the data plane of network devices [15]. This communication occurs through a series of two-way messages, allowing the controller to configure the SDN device's behaviour by creating, modifying, or deleting flow rules. As was mentioned above, an OpenFlow switch operates based on flow tables, which store a set of rules that determine how incoming packets should be processed. These rules are installed and managed by applications running on top of the SDN controller. In OpenFlow v.1.1, a flow rule consists of three key fields, Match fields, Counters, and Instructions. Matching Fields are used to identify packets based on header information at different network layers, including Layer 2 (Data Link), Layer 3 (Network), and Layer 4 (Transport). The switch maintains statistics, such as the number of packets that match a particular rule, using Counters. Instructions define the actions the switch must take on a packet that matches a rule. One of the most common instructions is forwarding the packet to a specific physical port. However, OpenFlow also supports special dummy ports that perform additional functions, including the following: Local, All, and Controller [15]. Local port dictates that the packet should be passed to the local OpenFlow controller of the switch, if further processing is needed. All broadcasts the packet to all ports on the switch, except the one it was received on. Controller indicates that the switch should forward the packet directly to the central controller.

The OpenFlow protocol is deployed on both ends of the interface connecting the SDN controller and the network infrastructure devices. This interaction is facilitated through a standardized set of messages exchanged over a secure TCP connection, using Transport Layer Security TLS for encryption. Each OpenFlow message begins with an OpenFlow header that specifies message type, message length, the protocol version number, and transaction ID. Messages are divided into three general categories, Symmetric, Controller-to-Switch, and Asynchronous, as shown in table 2, each comprising multiple sub-types [15].

Symmetric messages can be exchanged by both the controller and the switch at any time. HELLO messages are used when a secure channel is first established, the controller and the switch exchange HELLO messages to determine the highest OpenFlow version that both supports. The session will use the lowest common version to ensure compatibility. ECHO messages are used throughout the connection's lifetime to check if the connection is still active. They also help in measuring network delay and bandwidth between the connection of the controller and the switch.

Asynchronous messages are sent from the switch to the controller without being explicitly requested. They notify the controller about network events that may require its attention, such as new packets. One of the most important asynchronous messages especially for this thesis project is the PACKET_IN message, which occurs when a switch receives a packet that does not match any flow rule in its flow table. Instead of discarding the packet, it forwards the packet to the controller using PACKET_IN messages. The controller then decides how to handle it. A FLOW_REMOVED message type signifies that a flow entry has been removed from the switch's flow table, either due to expiration or intentional deletion, the switch informs the controller via a FLOW_REMOVED message. PORT_STATUS messages provide critical information about changes in a port state. ERROR messages are sent whenever the switch encounters an issue, notifying the controller of the error.

Messages sent from the controller to the switch play a critical role in managing network behaviour. These messages fall into five main categories: Switch Configuration, Commands from the Controller, Statistics, Queue Configuration, and Barrier messages. Switch Configuration Messages allow the controller to configure various settings on the switch. This is done using the SET_CONFIG message, which is a one-way message that allows the controller to set configuration parameters within the switch. Additionally, the controller can retrieve these settings using request and reply messages.

The controller needs to determine the switch capabilities. To achieve this, the controller uses a pair of FEATURES messages. This exchange is particularly important during the establishment of a Transport Layer Security TLS session, as it helps the controller understand the specific features supported by the switch. There are three messages that belong to the (commands from the controller) category, PACKET_OUT, FLOW_MOD, and PORT_MOD. The controller uses the first message to send data packets to the switch to pass them through the data plane, the second message is used to modify pre-existing flow rules in the switch, and the third message is used to modify the state of one of the switch's ports.

Statistics Messages are used by the controller and enable it to query information related to the flow tables of the switch. The controller sends a STATS request message, and the switch responds with a STATS_REPLY, providing critical data for monitoring network operations. To ensure proper execution order, a BARRIER message pair is used by the controller. The controller sends a BARRIER_REQUEST message to instruct the switch to complete all previously messages before processing new ones. Once the switch has finished executing these received messages, it responds with a BARRIER_REPLY, confirming their completion. Queue Configuration messages allow the controller to understand and manage how switches organize traffic into different queues on a port. It consists of the QUEUE_GET_CONFIG_REQUEST and QUEUE_GET_CONFIG_REPLY pair. With this information, the controller can assign specific flows to designated queues, thereby improving QoS [13, p. 102].

Table 2. OpenFlow message types.

Message Category		
Symmetric Messages Sent in both directions	**Asynchronous Messages** Initiated by the switch	**Controller to Switch**
HELLO	FLOW_REMOVED	Features
ECHO	PORT_STATUS	Configuration
VENDOR	ERROR	PACKET_OUT
	PACKET_IN	FLOW_MOD

3.4.1 OpenFlow Protocol Version 3

OpenFlow version v.1.3 (OFv1.3) represents a significant advancement in the development of the OpenFlow protocol, making it one of the most widely adopted and stable versions in Software Defined Networks today [27]. It has played an important role in both industry and academic research, becoming the foundation for many SDN controller implementations. One of the key reasons for the widespread adoption of OFv1.3 is its long-term stability. Unlike earlier versions that introduced frequent changes, version 1.3 brought a comprehensive set of enhancements that were not quickly followed by another version. The following sections will highlight the most important features introduced in OFv1.3, with a specific focus on those relevant to this project.

3.4.2 Flow Rules

In OFv1.3, flow rules define how switches handle network traffic. These rules are composed of the following components, as shown in figure 2, Match fields, Priority, Counters, Instructions, Timeout, and Cookie. Together, they allow for precise and flexible traffic management within an SDN enabled network. An important enhancement in OFv1.3 is the OpenFlow Extensible Match (OXM) descriptor, which improves how packets are matched to flow rules. OXM uses a Type-Length-Value (TLV) structure to define match conditions, allowing the switch to inspect various packet header fields [13]. This flexibility enables matching based on Ethernet, VLAN, MPLS, IPv4, and IPv6 header fields, among others. Previous OpenFlow versions had a more rigid matching mechanism, making it difficult to introduce future extensions and limit the flexibility of matching.

Figure 2. The components of the OpenFlow v1.3 flow table entry.

3.4.3 Matching Mechanism

OpenFlow v1.3 introduces an enhanced matching mechanism that utilizes multiple flow tables instead of a single table and supports pipeline processing, allowing packets to be processed across multiple stages before a forwarding decision is made. Each switch must contain at least one flow, with tables numbered sequentially starting from zero. Packet processing always begins in the first table, and the rest of the tables can be used depending on the output resulting from matching with one of the rules of the first table [27]. If a packet matches a flow rule, the associated instructions are executed, and it is explicitly directed to another table using a Goto statement. However, a flow rule can forward the packet only to higher-numbered tables, ensuring that processing moves forward in the pipeline. If a packet does not match any of the rules, a tablemiss situation occurs. In this case, the packet's handling outcome depends on the default flow rule configured in the switch. It can either be dropped or sent to the controller through a Packet-In message.

4 The Empirical Environment Components

In this section, I implemented the experimental environment for Software Defined Networks that supports the OpenFlow protocol. This environment emulates programmatically defined networks in real life. It should be noted that the environment is not a simulator as is the case in programs such as NS2, NS3 or OMNET++, but rather an emulator that is not very different from the real networked environment. The aim of this environment is to provide a realistic platform for developing and testing SDN applications. In this chapter, the components of the test setup used to build and prepare the Software Defined Networking environment are described.

4.1 Platform Specification

The hosting machine platform (my laptop) has a processor of Intel(R) Core(TM) i5-8265U CPU @ 1.60 GHz, 2 Core(s), and 4 Logical Processor(s) with 16GB of RAM. I have used VMware Workstation 16 Player to install the Ubuntu 18.04 Linux operating system inside. Ubuntu OS has been chosen to provide tools and software packets that support the SDN environment with all its components. The Eclipse IDE is used to build interactive SDN applications in Java on top of the controller that run in parallel with the Floodlight controller. The environment is also equipped with the Java 8 platform, which provides software offices and APIs to run programs in the Java language.

4.2 Floodlight Controller

Floodlight is an open-source SDN controller written in Java. In an SDN architecture, the controller is often referred to as the “brain” of the network because it has a global view and makes centralized decisions about how traffic should flow across the network. Important features of Floodlight controller include [28]:

1. OpenFlow Support: Floodlight fully supports the OpenFlow protocol, which is the communication standard between the controller and SDN-enabled devices such as switches. This allows Floodlight to program the flow tables on switches, determining how data packets are forwarded through the network.

2. Making Centralized Decisions: Instead of having each switch make independent decisions, Floodlight centralizes the control in the network. It manages the entire network’s behavior from one location. This allows for better traffic management, network optimization, and dynamic reconfiguration to adapt to changing conditions.

3. RESTful API: Through its RESTful API, Floodlight allows external applications to communicate with the controller. This is particularly helpful for developers who want to build custom applications that automate network functions.

4. Java based and Extensible: Since Floodlight is written in the Java, it can run on various platforms like Linux, Windows, or macOS. Its modular design enables developers to easily add or customize features by creating or modifying modules based on specific network needs.

Floodlight is built using a modular approach. It is made up of different modules, each responsible for a specific feature or function. These modules can be grouped into two types, controller modules and application modules. Controller modules in Floodlight provide core services needed for a Software Defined Network. They are responsible for implementing common functions that most network applications need to interact with a network. These modules handle essential tasks such as discovering network events and communicating with network devices. Application modules provide additional network solutions for different purposes that need to be implemented in the network. The following are some of the controller and application modules provided by the controller, with a brief description of their roles [22]:

- FloodlightProvider: This module acts as the central event handler within the controller, managing the switches and converting incoming OpenFlow messages into Java-based events that can be processed by other modules. It is also responsible for organizing the order in which events are delivered to listening modules. Messages such as PacketIn, FLOW-REMOVED, and PORT-STATUS, notify the controller of important network events. For a module to listen for these OpenFlow messages, it must implement the IOFMessageListener interface. In addition to handling OpenFlow events, the FloodlightProvider manages the registration of modules, ensuring that the appropriate modules receive the relevant notifications.

- OFSwitchManager: This module is responsible for managing all the OpenFlow switches within the network's data plane. It facilitates the communication between the controller and the switches by handling messages like FLOW-MOD and Packet-Out. Other modules that wish to interact with switches must register through the IOFSwitchService. The OFSwitchManager uses the Java Netty library to manage threads, ensuring secure and efficient communication with switches. Each OpenFlow message is processed by a Netty thread, and the logic for handling these messages is distributed across all modules.

- DeviceManagerImpl: It focuses on tracking devices within the network. It identifies and classifies devices based on Packet-In messages from switches, using an Entity Classifier that typically relies on MAC addresses and VLAN IDs. The DeviceManager records each device's IP address and attachment points. If a device becomes unreachable, the DeviceManager automatically removes its recorded attachment points and IP addresses, keeping the network state updated and accurate.

- Learning Switch: A Learning Switch application operates as a Layer 2 (L2) switch.

- Hub: Unlike the Learning Switch, the Hub does not learn or record device locations. Instead, it simply floods any incoming packets to all active ports, mimicking the behavior of a traditional network hub.

Installing the Floodlight controller on Ubuntu involves a series of steps that ensures all required components and dependencies are properly configured:

- The first step is installing Java 8. This is done by executing the command: "`sudo apt install openjdk-8-jdk openjdk-8-jre`"

- Next, the required dependencies need to be installed: "`sudo apt install build-essential ant python2-dev openjdk-8-jdk maven git`"

- Once the environment is set, the Floodlight source code can be downloaded from GitHub using the following command: “`sudo git clone https://github.com/floodlight/floodlight.git`”

- The next step is to navigate into the newly created floodlight directory and initialize and update the submodules: “`cd floodlight`”, “`git submodule init`”, “`git submodule update`”

- The Floodlight is then built using ant: “`ant`”

- Installing Eclipse as an IDE for Floodlight.

- To set up, develop, and execute the Floodlight controller within the Eclipse: “`ant eclipse`”

4.3 Java IDE: Eclipse

Eclipse is a popular Java Integrated Development Environment (IDE). It is a software tool that helps developers write, debug, and manage code efficiently. It offers features like code suggestions, error highlighting, and project management. Making it ideal for Java based projects. Eclipse is also popular among developers working with Floodlight controller, which is build using Java. It is an excellent choice for developers aiming to contribute to the Floodlight project or create custom applications that integrate with it.

To get started with Floodlight in Eclipse, the first step is downloading the IDE from the official website (https://www.eclipse.org/downloads/packages/). Once Eclipse is installed as was mentioned in the earlier section, developers need to configure it for the Floodlight project. This is done by running the command “`ant eclipse`”. The “`ant eclipse`” command configures the Floodlight project to be compatible with Eclipse.

4.4 Software Module Loaded System

Floodlight uses a local Module System to manage and control which modules are loaded and executed. This system is highly configurable, allowing developers to modify a custom configuration file to determine which modules should be included in the Floodlight controller runtime. The system consists of a few main parts, including [22], Module Loader, modules and services, a configuration file, and a file containing a list of available modules.

A module in Floodlight is essentially a Java class that implements the IFloodlightModule interface and can offer one or more services. A service is an interface that inherits from IFloodlightService. The Configuration File is a text file formatted using Java's standard properties format (key-value pairs). The key "Floodlight.modules" specifies the list of modules that should be loaded, and the value is defined as a list of module names as shown in the figure 3. The line 20 in the figure shows the name of my proposed module.

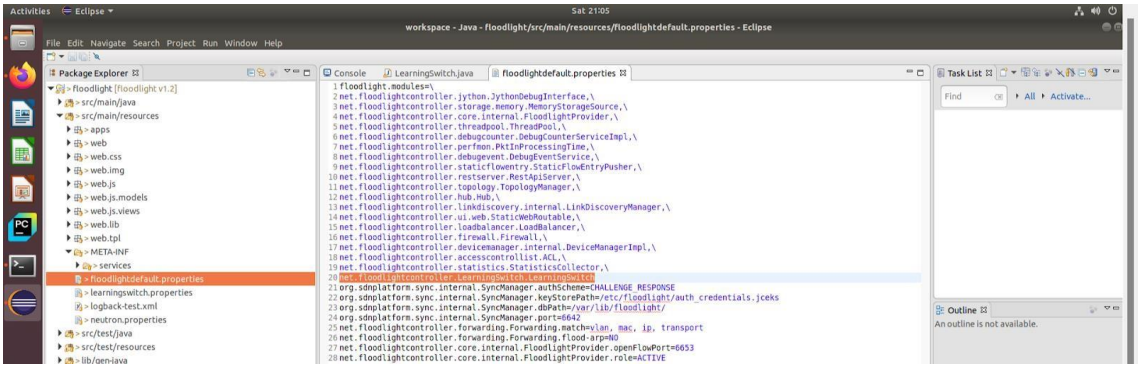

Figure 3. The SDN modules loaded by Floodlight including the proposed LearningSwitch module.

Once the environment was prepared, including the setup of Eclipse and the installation of Mininet, I started developing a new module in Floodlight. The following steps outline the process of creating a new module, which in this case is named LearningSwitch:

1. **Creating the Module Class and implementing required interfaces.**
    - To create a new Module in Eclipse, navigate to the src/main/java folder in the Floodlight project.

- Create a new Java class in the package box.

(net.floodlightcontroller.custommodule → “net.floodlightcontroller.LearningSwitch”).

- Name the class appropriately based on its functionality.
- Add the IFloodlightModule and IOFMessageListener interfaces to respond to OpenFlow messages.
- Click finish to add the class in Eclipse.

By now, you will end up with a custom Floodlight module, which includes required interfaces, functions, and dependencies.

2. **Adding Dependencies and Initialization**

   Adding a few dependencies that the module will use, such as the IFloodlightProviderService, ArrayList, Logger, HashMap. Then defining member variables into the class and implementing functions.

3. **Handling OpenFlow Messages**
   - To handle specific OpenFlow messages (e.g., PACKET_IN), register the module as a listener in the startUp() method:
   - Implement the receive() method to define the behavior when an OpenFlow message is received:
4. **Registering the Module with Floodlight**
   - Open the following file: `src/main/resources/META-INF/services/net.floodlightcontroller.core.module.IFloodlightModule`
   - Add the fully qualified name of the new module (e.g. “`net.floodlightcontroller.LearningSwitch.LearningSwitch`”).
   - Modify the following `src/main/resources/floodlightdefault.properties` as well to include the module: “`net.floodlightcontroller.LearningSwitch.LearningSwitch`”

- Run the controller in Eclipse by right clicking `Main.java` and selecting **Run As > Java Application**.

4.5 Open vSwitch

Open vSwitch (OVS) is an open-source virtual switch, primarily used in virtualized and cloud-based environments. One of the main functions of OVS is to support network virtualization. It allows virtual machines (VMs), containers, and physical network devices to communicate with each other. In data centers and cloud environments, where resources are often virtualized, OVS ensures that all these components can interact effectively within the network [29].

In Software Defined Networking, OVS plays an important role by providing the needed network infrastructure. OVS is compatible with a wide variety of hypervisors, such as VMware, Xen, Hyper-V and KVM. It is also used in container orchestration platforms, such as Kubernetes, and cloud platform such as OpenStack. These platforms rely on OVS to create virtual networks that can scale [29]. In this thesis project, I have used the version 2.8.90 of the OVS switch, which was released in 2017. It is stable and compatible with SDN projects.

4.6 Mininet Emulator

Mininet is an open-source network emulator that is particularly useful for working with Software-Defined Networking. It enables users to build and emulate a virtual network on a single computer, eliminating the need for physical network hardware. This makes Mininet excellent tool for testing and experimenting with SDN in a virtual environment. Mininet operates by using network namespaces, which are lightweight virtual machines on a Linux Kernel. These namespaces replicate the behavior of different network components, enabling users to build customized networks for their SDN experiments. Key features of Mininet include [30]:

- Topology creation: Mininet enables users to design custom networks through a Python-based API. Users can design a network with controllers, switches, nodes, hosts, and links according to their requirements.
- Emulation of SDN switches: Mininet is capable of emulating OpenFlow switches, which are crucial in SDN systems for managing data flows across a network. This functionality enables users to test SDN behavior without needing physical switches.
- Hosts and controllers: Users can include hosts in the network and link them to SDN controllers such as Floodlight, OpenDaylight, or Ryu.

Mininet is widely adopted emulator within the Software Defined Networking community, mainly designed for research and education. It enables users to experiment with network configurations, understand the underlying principles of SDN, and create prototypes of SDN-based applications [31].

5 The Conducted Experiment

Software Defined Networking (SDN) is a revolutionary approach to network management that allows network administrators to control and manage network resources dynamically through software applications. After setting up a complete SDN environment, this section shows the output of the proposed application using Mininet to emulate the network, OVS to provide virtual switches, and Floodlight as the controller to manage the network's behavior.

5.1 Building SDN Topologies

The experiments conducted in this project include three SDN topologies consisting of one Floodlight controller and a variable number of OVS switches. The objective is to evaluate the proposed Learning Switch algorithm across different network topologies, ensuring Layer 2 (Data Link layer) connectivity between hosts. These experiments aim to validate the algorithm's effectiveness

in establishing communication and to analyze its performance across different scenarios. In the experiment, several topologies were designed with varying sizes based on the number of switches, devices (hosts), and connections between them. Three specific networks, SDN_net_1, SDN_net_2, and SDN_net_3 were created to present small, medium, and large SDN topologies, respectively. These topologies are illustrated in figures (7), (8), and (9).

To build these topologies, the Miniedit tool was used. Miniedit makes it easier to design network topologies visually without needing to write extensive code. After the topologies were created using Miniedit, they were saved as Python scripts, allowing for later execution and testing. To create the topologies, the Mininet tool was run by executing the following command, as demonstrated in figure 4.

```
mesana@ubuntu:~$ sudo -s
[sudo] password for mesana:
root@ubuntu:~# ./mininet/examples/miniedit.py
```

Figure 4. Executing the following command to run MiniEdit.

During the topology setup in Mininet, it is necessary to modify the controller configuration. I set the controller port to 6653 and specified the controller type as Remote Controller, as illustrated in figure 5.

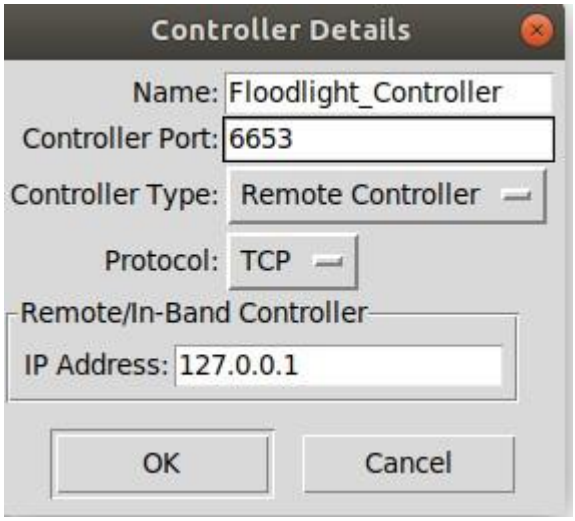

Figure 5. Floodlight Controller details.

First, the SDN controller was started using the Eclipse development platform (the controller's code was run directly from Eclipse). Once the controller was up and

running, it automatically created a TCP connection with the switches. The connection is established on the standard port 6653, which is the default communication port used by the Floodlight controller for OpenFlow traffic. After the connection was established, the OpenFlow protocol became active. Next, I need to remove the built-in learning switch module and add my LearningSwitch module to the properties file on the Floodlight controller. I also added the net.floodlightcontroller.hub.Hub to the floodlightdefault.properties on the controller.

Figure 6. Java module applications that are loaded in the controller properties file.

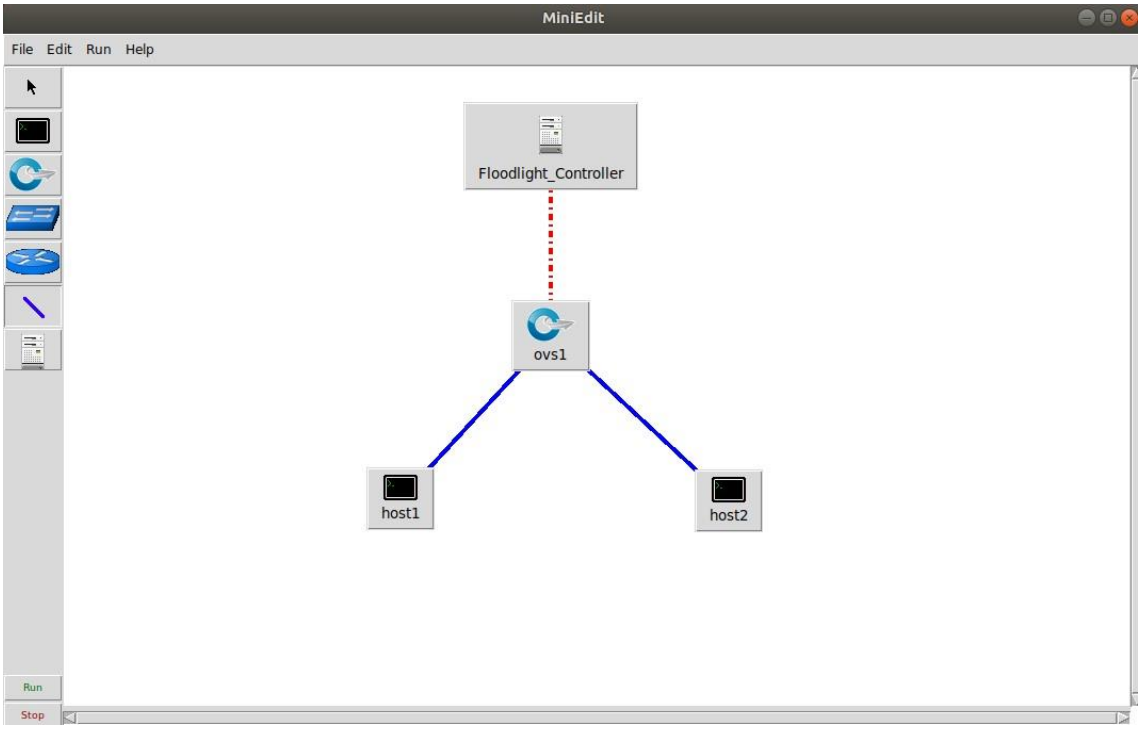

Figure 7. The net_1 topology.

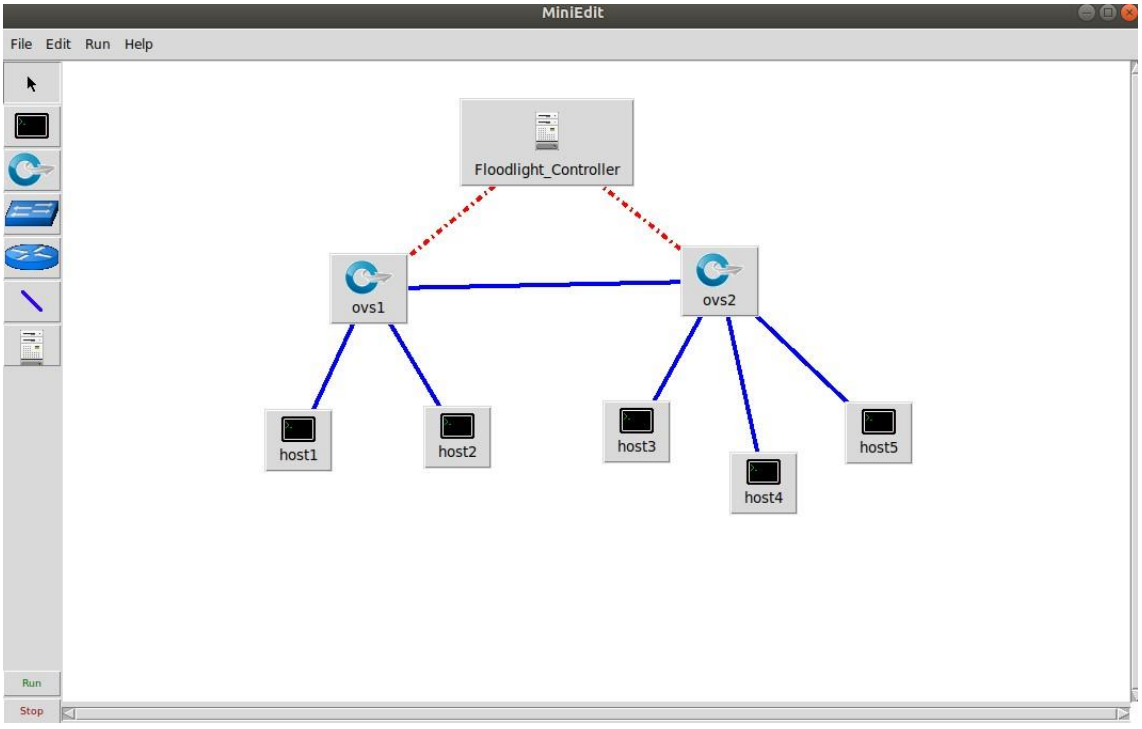

Figure 8. The net_2 topology.

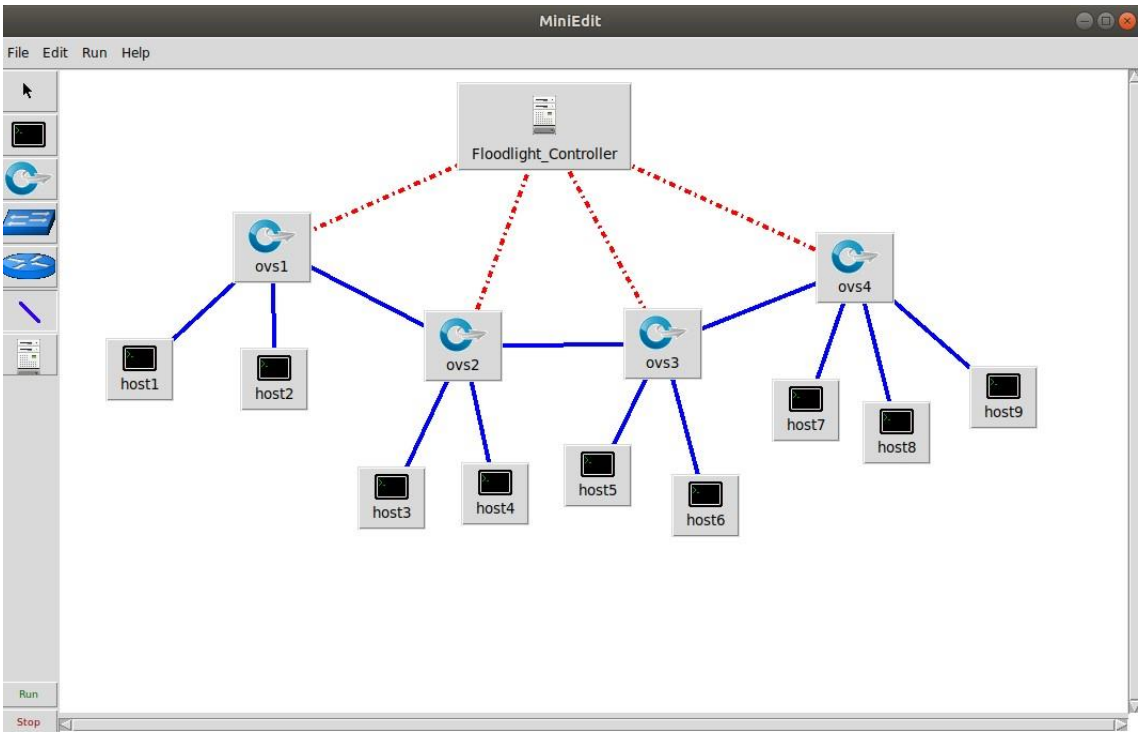

Figure 9. The net_3 topology.

5.2 ICMP Protocol

The Internet Control Message Protocol (ICMP) is a TCP/IP network layer protocol, primarily used for reporting errors and providing network diagnostics. It enables network devices such as routers and hosts to communicate issues related to data transmission. Network administrators often use ICMP-based tools like ping and traceroute to diagnose and troubleshoot connectivity problems.

ICMP messages are transmitted in the form of datagrams, which are self-contained data units similar to packets. Each ICMP message is encapsulated within an IP packet, with the ICMP data placed in the section of the IP data. These messages also include the entire IP header of the original packet, allowing the receiving system to identify which packet encountered a problem.

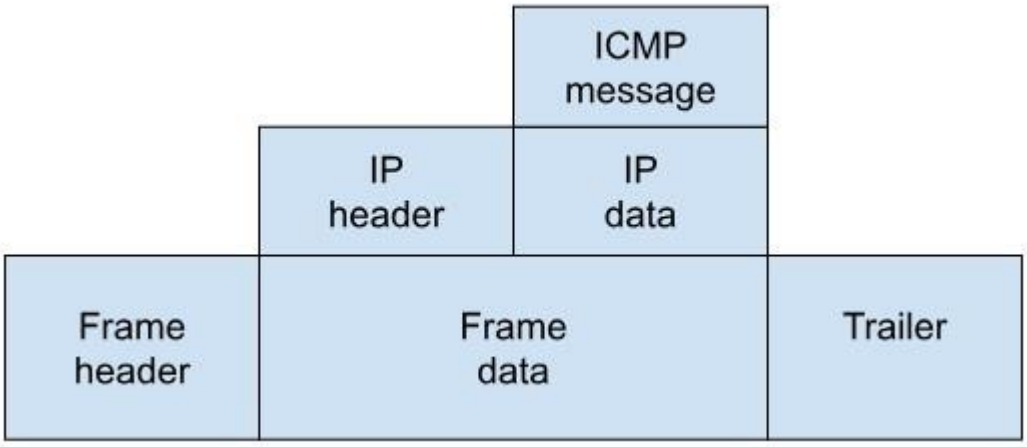

Figure 10. Encapsulation of ICMP packet.

ICMP messages fall into two main categories: Error Reporting Messages, and Query Messages. Both types share the same initial three structure fields as shown in figure 11, which includes the following fields in the first 4 bytes: Type, Code, and Checksum. Then the rest of the ICMP message.

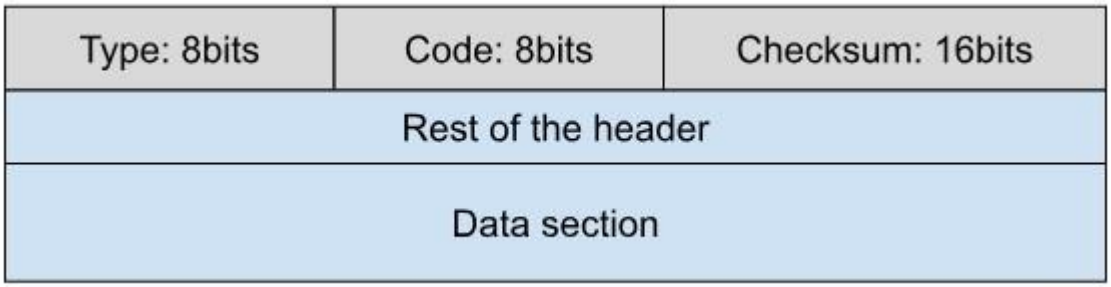

Figure 11. The general structure of ICMP messages.

ICMP always sends error messages to the original sender. There are five primary types of ICMP error messages: Destination unreachable, Source, quench, Time exceeded, Parameter problems, Redirection. ICMP also supports query messages, which are used for diagnostics and network information requests. Examples include Echo request and reply, Timestamp request and reply, Address mask request and reply, Router solicitation and advertisement.

One of the most common uses of ICMP is the Echo Request and Echo Reply mechanism. When a host sends an ICMP Echo Request, it is essentially inquiring whether the destination host is available and responsive. If the destination host receives the request, it responds with an ICMP Echo Reply. This back-and-forth exchange is used in the ping command to test connectivity between two endpoints.

ICMP message types are identified by a Type field:

- Type 8: Echo Request sent by a host or router to test connectivity with another device.
- Type 0: Echo Reply sent in response to an echo request, confirming the device is active and responsive.

Table 3. The structure of the echo-request and echo reply message includes several key fields.

The structure of an ICMP	
Type 8 bits	Indicates the ICMP message type (8 for Echo Request, 0 for Echo Reply)
Code 8 bits	Provides further context for the message type (often 0 for Echo Request/Reply)
Checksum 16 bits	Error-checking field for the ICMP header and data
Identifier 16 bits	Used to match Echo Requests to Replies
Sequence Number 16 bits	Helps in identifying the order of packets
Payload	Optional data carried in the ICMP message, usually used for testing round-trip time

These two types play an important role in the LearningSwitch application built for the Floodlight SDN controller. The application uses ICMP Echo Reply packets to trigger the creation of rules on the OpenFlow switch. While the Echo Request is used to initiate communication, it is the Echo Reply that contains sufficient information to build flow rules.

5.3 Proposed Algorithm of Learning Switch

It should be noted that the primary objective of this project is to develop my own SDN LearningSwitch application on the Floodlight controller, despite the existence of a built-in switch learning module. The motivation behind this approach is to gain hands-on experience with Software Defined Networking (SDN) by designing and implementing an application from scratch. This serves as an initial step in understanding the SDN technology, its architecture, and the interaction between the controller and network devices. The code of the proposed algorithm will be explained in two parts as shown in figures 12 and 13, respectively.

In this section, I present a new algorithm solution for SDN_as a Java application that is loaded and executed by the Floodlight controller. The following section explains the design and implementation of the SDN application that functions as a layer 2 learning switch. The core objective of the application is to learn the network topology by identifying connected switches and end-hosts, which serves as a foundation for installing flow rules that guide packet forwarding.

5.3.1 Building the HashMap of the topology

The main logic of the application is embedded within the `receive()` function, which is invoked each time the controller receives an OpenFlow message. In this module there are two core data structures, a list of switches `("switches")`, and a topology HashMap `("topology")`. The list of the switches maintains all switches known to the controller. It helps ensure that each switch is only added once, providing a record of all network switches encountered during runtime. The

topology HashMap maps each switch to an array list of host devices connected to it. The value associated with each switch key is an ArrayList of HostInfo objects, each containing the MAC address and input port of a host/Ingress port identifying where the host is connected. Figure 6 shows how the HashMap design of the topology looks like.

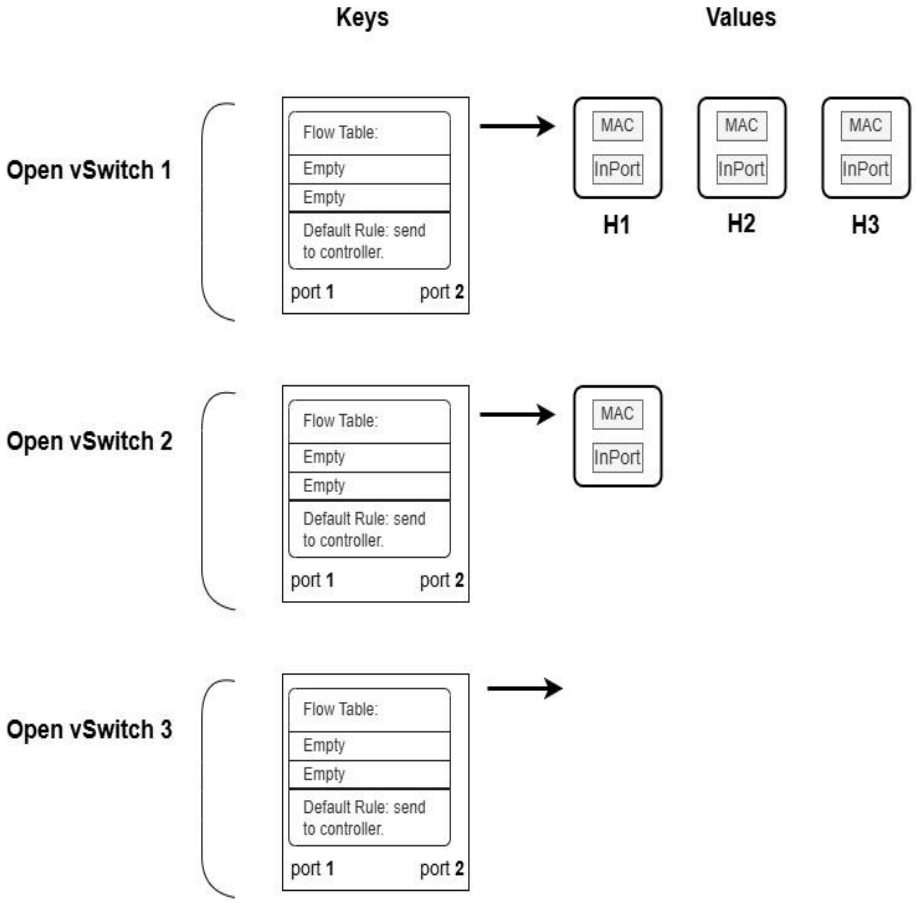

Figure 12. Building the HashMap of the topology, algorithm 1 of the LearningSwitch application.

Algorithm 1 demonstrates how the LearningSwitch application maintains a view of the SDN network topology (line 1 to 11). The application listens for incoming OpenFlow messages through the `receive()` function. Among various types of OpenFlow messages, the application verifies that the incoming message is of type Packet_IN (line 6), which are sent by switches to the controller when they receive packets that do not match any existing flow rules.

Algorithm 1: Building the topology Hash Map

Event: An OpenFlow message is received by the Floodlight controller
Input: The corresponding switch *sw*, An OpenFlow message *msg* from *sw*
Output: An Array list of switches, The Hash Map of our SDN network *topology*

```
1    Create a new array list of OpenFlow Switches
2    Create a Hash Map of Switches as the key and array of Hosts as a value
3    if Switches list does not contain sw then
4    Switches.add(sw)
5    Add sw as a key in topology and empty array of Hosts as a value
6    if the type of msg is PACKET_IN then
7    Create a new host object
8    host.src_mac := PACKET_IN.get(Source MAC address)
9    host.in_port := PACKET_IN.get(The ingress port)
10   if topology hash map does not contain host then
11   Add host to the topology where the key is sw
```

When an OpenFlow message is received, the application performs two main tasks:

1. Detecting Switches

The switch from which the packet originated is first checked against the switches list. If it is not already present, it is added. Simultaneously, an empty list is initialized in the topology HashMap to store hosts connected to the switch (line 1 to 5).

2. Host Learning

Before adding the host to the topology, a check is performed to ensure it has not already been recorded (line 10). This avoids duplication in scenarios where multiple packets are received from the same host. By doing so, the application efficiently learns the location of each host in the network based on its MAC address and the port it is connected to on the switch.

5.3.2 Building Flow Rules on the Switches

When starting with a new network topology, all switches have empty flow tables except for the default rule, which instructs them to send all unmatched packets to the controller. Since the switches in the topology don't have any pre-installed logic to handle traffic, they cannot make forwarding decisions on their own. To manage this, the hub module which is a built-in application within Floodlight is activated. This module doesn't build any specific rules but tells the switches to flood any packet they receive out of all ports, ensuring that traffic reaches its destination. In the LearningSwitch application, the second part of the algorithm as shown in the figure 13 focuses on building flow rules on the switch using the ICMP echo reply message.

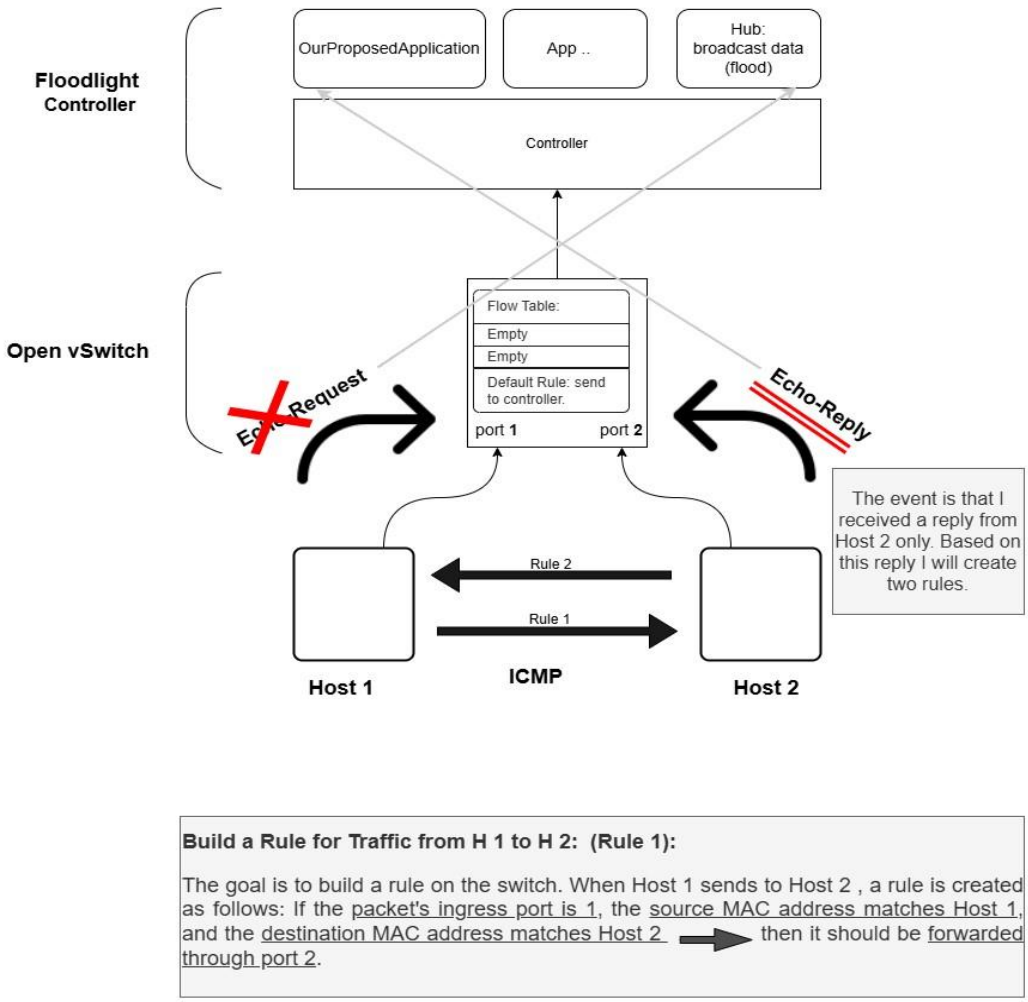

Figure 13. Building the rules based on the reply, algorithm 2 of the LearningSwitch application.

The ping command causes each host in the network to send an ICMP echo request to every other host. According to the ICMP protocol, each receiving host responds with an ICMP echo reply. When host 1 pings host 2, it sends an ICMP echo request. This echo request packet is forwarded to the switch. Since the switch does not have any pre-installed rules, it sends the packet to the controller. The hub module is enabled in the controller to flood_the packet to all ports, ensuring the echo request reaches its destination which is host 2. Host 2 replies

with an echo reply. This echo reply packet is sent to the switch, which again lacks a rule for it, so it forwards it to the controller. At this point, the LearningSwitch application captures the echo reply and begins building rules.

Algorithm 2 demonstrates how the application builds rules (line 1 to 25). It is important to mention that the echo request lacks sufficient information to build a flow rule. Specifically, it does not include the destination MAC address. The echo reply, on the other hand, contains all the needed details: source MAC address, destination MAC address, and the egress port (the port it should be sent out from). However, it still lacks one piece of information: the ingress port (the port the original echo request came in on). This value will be retrieved from the HashMap built earlier in the application, which stores the host to its respective port on the switch.

Algorithm 2: Building Flow Rules on the Switches

Event: A *Packet_IN* message is received by the Floodlight controller **Input:** The corresponding switch *sw*, The Hash Map of our SDN network *topology*, An IP Packet *ippacket* from a *host* connected to *sw* **Output:** The flow Rules will be installed into the switch *sw*

```
1   if the type of ippacket is ICMP then
2   if the ICMP type is Echo Reply then
3   src_mac := ippacket.get(Source MAC Address)
4   dst_mac := ippacket.get(Destination MAC Address)
5   in_port := topology.get(ingress port of a host that has src_mac)
6   Create a Match for the first flow rule
7   Match.InPort := in_port
8   Match.SrcMAC := dst_mac
9   Match.DstMAC := src_mac
10  Create a FlowMOD message
11  FlowMOD.Match := Match
12  FlowMOD.TimeOut := 1000 seconds
13  FlowMOD.Priority := 115
14  FlowMOD.Action := Forward to the output port from Packet_IN
15  Send the created Flow Rule to the switch sw
16  Create a Match for the second flow rule
17  Match.InPort := the ingress port from Packet_IN
18  Match.SrcMAC := src_mac
19  Match.DstMAC := dst_mac
20  Create a FlowMOD message
21  FlowMOD.Match := Match
22  FlowMOD.TimeOut := 1000 seconds
23  FlowMOD.Priority := 115
24  FlowMOD.Action := Forward to the output port in_port 25  Send the
    created Flow Rule to the switch sw
```

In order to build a rule on the switch, four elements are needed: three match fields and one action field. Based on this echo reply, I create a rule that has three matching conditions (line 6): Ingress port which is taken from the topology HashMap (line 5 and 7), identifying where h1 is connected. Source MAC address which is the host 1's MAC (line 4 and 8). Destination MAC address which is the

host 2's MAC address (line 3 and 9). With these elements the rule can be generated and applied. However, an issue occurs when the network topology becomes more complex. In larger topologies, the switch might already have a rule from previous traffic, and the echo reply from another host might not reach the controller. If the controller misses one Packet_In event, such as a reply, it fails to build a corresponding rule, leaving that host unreachable through the LearningSwitch application. To solve this issue the application creates two rules for every echo reply during the same event, one of which is the opposite of the other (line 16). The implementation starts by checking for a Packet_In event in the controller. The packet is first cast from Ethernet to IPv4, then to ICMP (line 1), since ICMP operates at Layer 3 and depends on IPv4 encapsulation. Once the packet is confirmed to be ICMP and the message type is 0 echo reply (line 2), the rule-building logic is triggered.

The necessary fields are extracted (line 3 to 5):

- Source MAC.
- Destination MAC.
- Ingress port from the HashMap.

Once these values are obtained, a match structure is created as shown in line 6 and 16. Since the first rule is for traffic from h1 to h2 and it is based on the echo reply, the MAC addresses are reversed. Then, the Flow_mod message, which is an OpenFlow responsible for building rules and has three types: add, modify, and delete, is constructed using ("OFFlowMod") (line 10 and 20), specifying the match, idle timeout, priority, and the output action. A second, opposite rule is then built for traffic flowing from h2 to h1 (line 16).

The main objective of this algorithm is to build rules on the switches. After the rules are installed, switches in the topology should no longer generate Packet_IN messages, as they can independently handle the traffic.

5.4 Experimental Scenario

After successfully developing the Floodlight module and designing the network topologies, it becomes essential to test the designed algorithm. To evaluate the functionality of the SDN application, three different network topologies were created using Mininet and saved as Python scripts: net1.py, net2.py, and net3.py.

This section focuses specifically on the experimental scenario using the net3.py topology. It outlines all necessary steps to execute the test and presents the observed output of the module, including the performance validation, the flow entries installed on each switch, and interpretation of the results. Running the Experiment on the third network topology to begin testing the algorithm:

1. Start the Floodlight Controller: Run the Floodlight controller from Eclipse.

2. Execute the Topology in Mininet: Before starting Mininet, it is recommended to clear any existing network states using the command: “`mn -c`”. Then, the desired topology can be launched using: “`python net3.py`”.

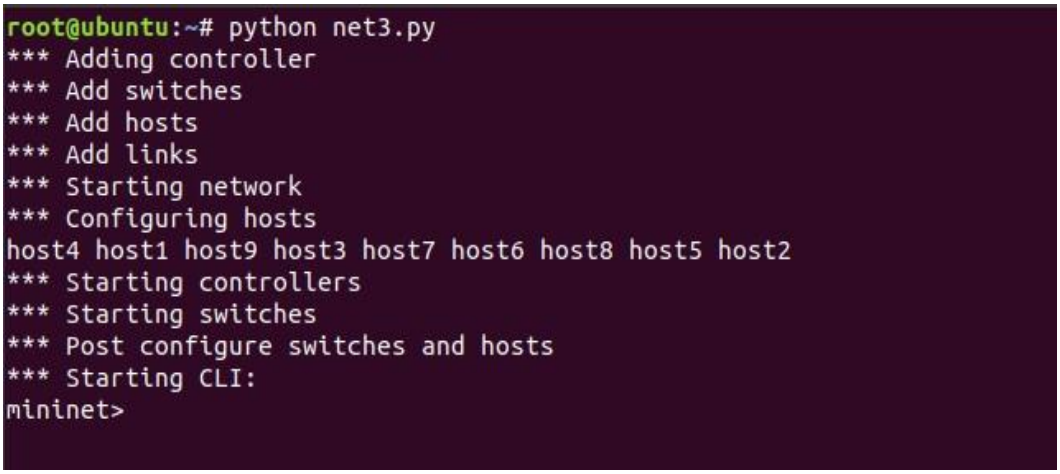

Figure 14. Executing the network topology.

3. Wait until the console output confirms that all hosts have been detected and connected by the controller (based on the log messages generated by the proposed application).

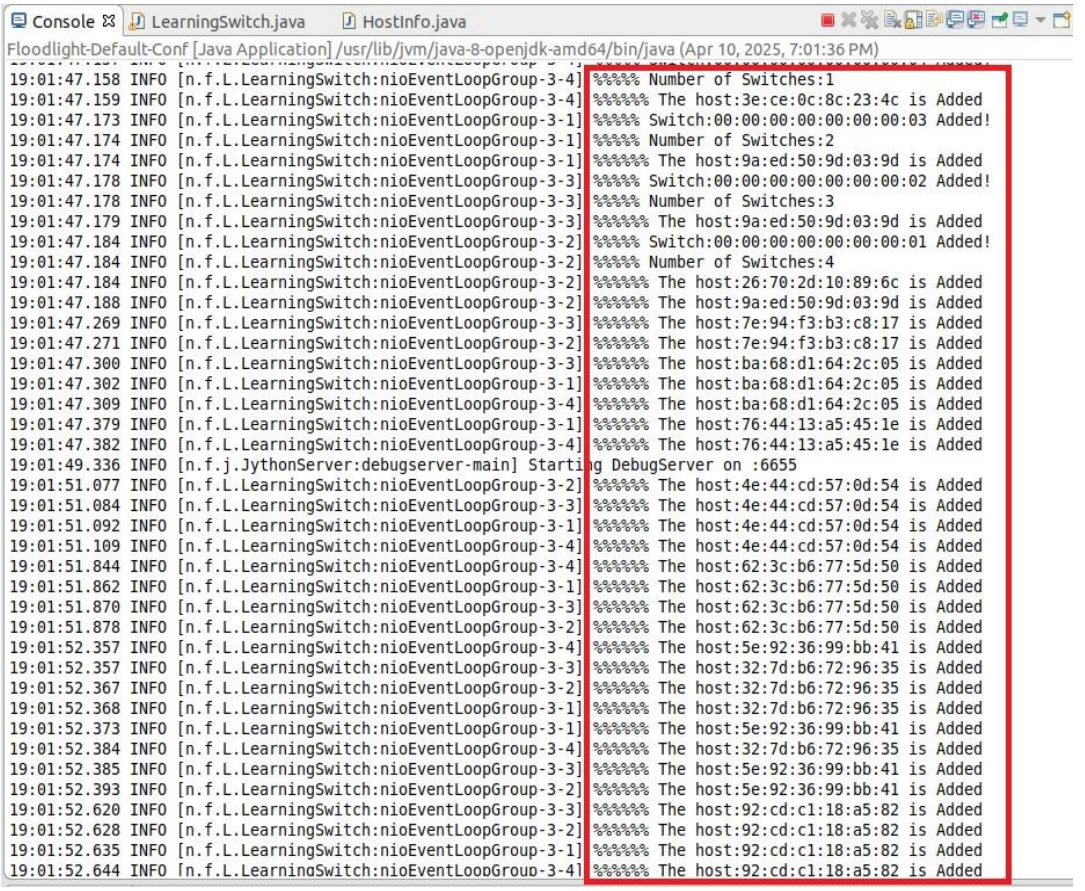

Figure 15. The console output confirming that all hosts have been added.

4. Triggering the Application Logic: To trigger the algorithm within the LearningSwitch module, the pingall command is issued: “pingall”. The pingall command causes each host to ping every other host, generating PACKET_IN events handled by the Floodlight module. These events are displayed in Eclipse's console as log messages. The initial run will result in many PACKET_IN messages, indicating that the controller is processing new flows and installing flow rules accordingly.

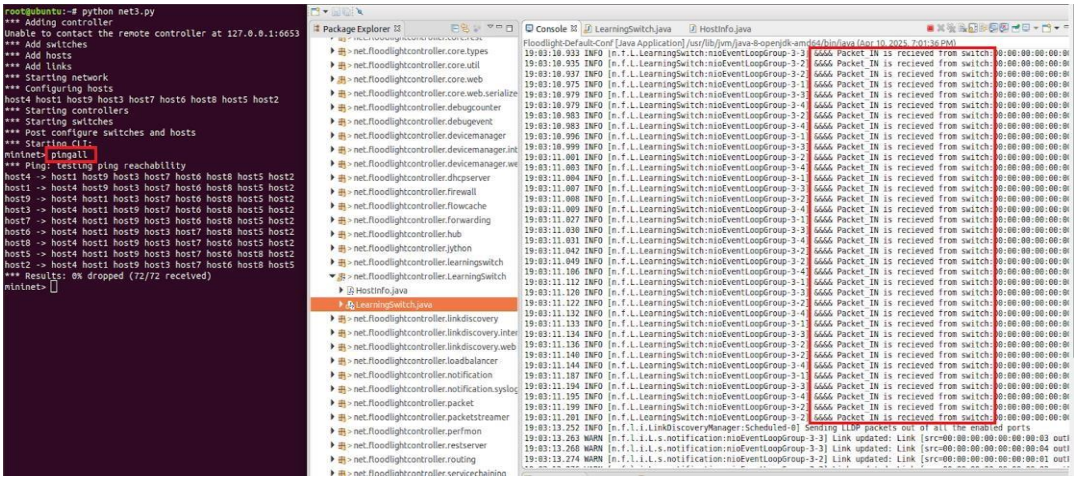

Figure 16. Triggering the application with “pingall” and displaying the Packet_IN events.

To verify the algorithm is functioning as intended, run “`pingall`” a second time. This time, there should be no Packet_IN logs displayed, indicating that the switches now have flow rules installed during the first run and can handle traffic directly without involving the controller.

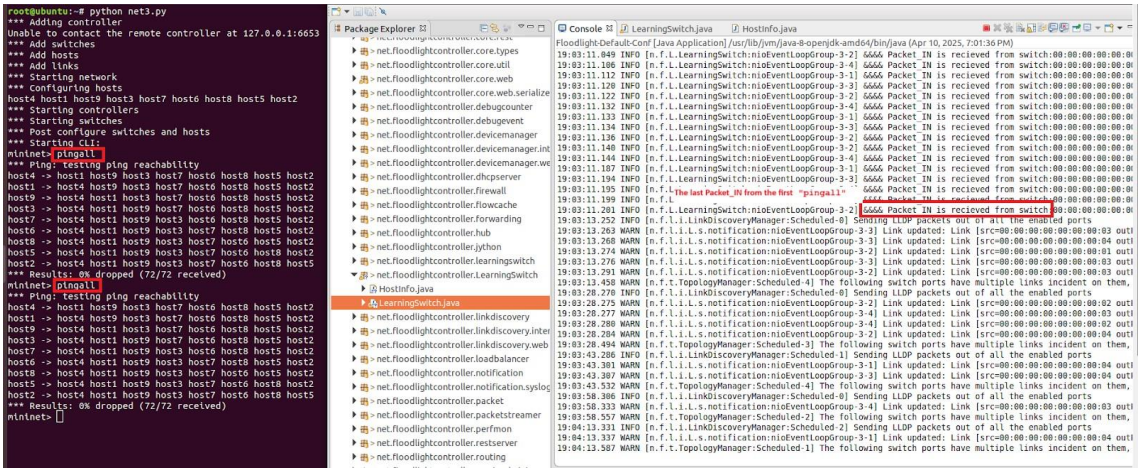

Figure 17. Verifying the algorithm by showing that there are no Packet_IN logs displayed.

To further inspect and analyze the flow rules applied to the switches, the Floodlight web interface is accessed via http://127.0.0.1:8080/ui/index.html. This graphical user interface offers a comprehensive overview of the network’s current state and consists of several key tabs: Dashboard, Topology, Switches, and Hosts. The Dashboard provides general information, including the number of

switches and hosts, while the Topology tab visually maps out the entire network structure.

Figure 18. Showing the Dashboard tab of the Floodlight web interface.

The most important information about the flow rules can be found in the Switches tab. As shown in figure 19. The flow rules installed by the SDN application are listed for each switch. Each switch includes one default rule, and the other rules were created by the LearningSwitch application. Each rule contains the following fields Match, Apply Actions, Packets, Timeout, and other fields as well. Match is the Conditions that incoming packets must meet (e.g., destination IP, source IP, input port). Apply Actions defines how the packet should be handled, such as forwarding to a specific port. Packets is a counter indicating how many packets matched the rule. Timeout indicates the duration the rule remains active (e.g., 1000 seconds).

Flows (31)

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=32:7d:b6:72:96:35	actions:output=2	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=2 eth_dst=32:7d:b6:72:96:35 eth_src=4e:44:cd:57:0d:54	actions:output=1	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=1 eth_dst=0a:04:fd:7f:bd:0d eth_src=32:7d:b6:72:96:35	actions:output=3	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=3 eth_dst=32:7d:b6:72:96:35 eth_src=0a:04:fd:7f:bd:0d	actions:output=1	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=5e:92:36:99:bb:41	actions:output=2	---	---	---	---	---	---	7	518	146	1000
0	0x0	115	in_port=2 eth_dst=5e:92:36:99:bb:41 eth_src=4e:44:cd:57:0d:54	actions:output=1	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=92:cd:c1:18:a5:82	actions:output=2	---	---	---	---	---	---	7	518	146	1000
0	0x0	115	in_port=2 eth_dst=92:cd:c1:18:a5:82 eth_src=4e:44:cd:57:0d:54	actions:output=1	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=62:3c:b6:77:5d:50	actions:output=2	---	---	---	---	---	---	7	518	146	1000
0	0x0	115	in_port=2 eth_dst=62:3c:b6:77:5d:50 eth_src=4e:44:cd:57:0d:54	actions:output=1	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=86:41:3a:1b:81:2f	actions:output=2	---	---	---	---	---	---	7	518	146	1000
0	0x0	115	in_port=2 eth_dst=86:41:3a:1b:81:2f eth_src=4e:44:cd:57:0d:54	actions:output=1	---	---	---	---	---	---	6	420	146	1000
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=82:c7:5e:0b:85:49	actions:output=2	---	---	---	---	---	---	7	518	146	1000

Figure 19. The list of all the flows installed on The OVS Switch 00:00:00:00:00:00:00:01.

0	0x0	115	in_port=1 eth_dst=0a:04:fd:7f:bd:0d eth_src=2a:45:8e:f8:50:79	actions:output=3	---	---	---	---	---	---	6	420	145	1000
0	0x0	115	in_port=3 eth_dst=2a:45:8e:f8:50:79 eth_src=0a:04:fd:7f:bd:0d	actions:output=1	---	---	---	---	---	---	6	420	145	1000
0	0x0	0		actions:output=controller	---	---	---	---	---	---	243	18461	231	0

Figure 20. Showing the last three flows on the OVS Switch 00:00:00:00:00:00:00:01 with the default rule.

Flows (31)

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
0	0x0	115	in_port=1 eth_dst=4e:44:cd:57:0d:54 eth_src=32:7d:b6:72:96:35	actions:output=2	---	---	---	---	---	---	6	420	146	1000

Figure 21. The first flow entry installed on the OVS Switch 00:00:00:00:00:00:00:01.

The figure above presents a screenshot of the first flow entry installed on an OVS switch which is identified as 00:00:00:00:00:00:00:01. This flow entry is part of a flow table shown in figure 19, which serves as a set of instructions that determine how the switch should process incoming packets based on defined match conditions and corresponding actions. Each flow entry is generated and managed by the Software-Defined Networking controller.

In This In this specific example, the first rule outlines a set of match criteria and an associated action to be taken when those criteria are met. The Match field specifies that a packet must arrive at port 1 "`in_port=1`", have a destination Ethernet address of 4e:44:cd:57:0d:54 "`eth_dst`", and a source Ethernet address of 32:7d:b6:72:96:35 "`eth_src`". When a packet satisfies all these conditions, the switch is instructed to forward the packet out of port 2, as indicated by the Apply Actions field "`actions=output:2`".

Table 4. Showing the three conditions of the Match field, and the action field of the first rule installed on the first OpenFlow switchOVS1.

Field	Description
Match 3 Matching conditions	the Packet is entering from port 1
	the Ethernet destination address is 4e:44:cd:57:0d:54
	the Ethernet source address is 32:7d:b6:72:96:35
Apply Actions	the Packet will be forwarded to port 2

The Packets field represents a counter that tracks the number of packets that have matched this particular flow rule since it was installed. At the time this screenshot was taken, the counter shows that 6 packets have matched the rule. This value increases as more traffic that fits the rule conditions is processed, such as when additional ping commands (e.g., pingall) are executed to test network connectivity. Furthermore, the Timeout field specifies a value of `1000` seconds,

which determines how long the flow entry will remain active in the table without receiving matching packets. If no such packets arrive during this interval, the rule will be removed automatically by the switch.

The LearningSwitch algorithm functioned correctly in all three topologies, and it was able to install the appropriate flow entries on the switches.

6 Conclusion

This thesis has focused on understanding and working with Software-Defined Networking (SDN) technology. The increasing adoption of SDN technology in modern networks has brought numerous benefits, such as flexibility, scalability, and centralized management. The main objective of this thesis was to develop an SDN application using the Floodlight controller to enable communication between hosts at the Layer 2 (Data Link Layer). Various SDN network topologies were implemented using Mininet as a platform for network emulation to evaluate the performance of the proposed algorithm. Testing confirmed that the approach successfully established host-to-host connectivity.

Furthermore, this work served as an initial step to equip readers with essential knowledge about SDN before setting up a test or development environment. In addition to gaining practical experience with the SDN technology and working with the Floodlight controller.

The shift that SDN provides from a distributed to a centralized control model allows the controller to make decisions for the entire network. This centralization of control-plane decisions offers network operators a great opportunity to simplify network management, enhance network efficiency, and facilitate the development and deployment of new network services and applications. The work in this thesis can be extended to support layer 3 by expanding the application to handle Layer 3 (Network Layer) communication. Additionally, future enhancements could include creating more advanced methods to protect SDN from attacks such as detecting any unauthorized changes with flow rules on switches or implementing detection systems that monitor traffic patterns. Lastly, testing the entire process

from application development on the controller to deployment and evaluation on real hardware equipment, would also be insightful.

References

1. Nadeau, T.D. and Gray, K. 2013. SDN: Software Defined Networks. O'Reily Media.

2. Kreutz, D. et al. 2015. Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE.

3. Shenker, S. 2012. Gentle Introduction to Software-Defined Networking. Available from: https://youtu.be/eXsCQdshMr4?si=ZemCMTQMyJBgpK9G.

4. Coker, O. and Azodolmolky, S. 2017. Software-Defined Networking with OpenFlow – Second Edition. O'Reily.

5. Odom, W. 2024. 200-301 Official Cert Guide, Volume 1, 2nd Edition.

6. Feamster, N. Rexford, J. and Zegura, E. 2014. The Road to SDN: An Intellectual History of Programmable Networks. ACM Computer Communication Review.

7. Odom, W. 2024. 200-301 Official Cert Guide, Volume 2, 2nd Edition.

8. Open Networking Foundation (ONF). 2025. [Online]. Available: https://opennetworking.org/sdn-definition/ .

9. Bannour, F. et al. 2018. Distributed SDN Control: Survey, Taxonomy, and Challenges. IEEE Communications Surveys.

10. Shenker, S. et al. 2011. The Future of Networking, and Past of Protocols. Open Networking Summit.

11. Zhou, W. et al. 2014. REST API Design Patterns for SDN Northbound API. IEEE.

12. Shin, M. Nam, K. and Kim, H. 2012. Software-defined Networking (SDN): A reference architecture and open APIs. IEEE.

13. Göransson, P. Black, C. and Culver, T. 2017. Software Defined Networks: A Comprehensive Approach. Second Edition.

14. Li, T. Chen, J. and Fu, H. Application Scenarios on SDN: An Overview.

15. Open Network Foundation (ONF). 2009. OpenFlow Switch Specification. Version 1.0.0.

16. Big Switch Networks – Switch Light. Available from: https://opennetworking.org/sdn-resources/sdn-products/big-switch-networksswitch-light/

17. Gude, N. et al. NOX: Towards an Operating System for Networks. ACM Computer Communication Review.

18. POX an OpenFlow Controller. Nicira, About POX. Available online: https://github.com/noxrepo/pox.

19. MAESTRO. Available online: https://code.google.com/archive/p/maestroplatform/.

20. Erickson, D. 2013. The Beacon OpenFlow Controller. Proceedings of the second ACM SIGCOMM workshop on Hot Topics in SDN.

21. Ishii, S. et al. 2012. Extending the RISE Controller for the Interconnection of RISE and OS3E/NDDI. IEEE International Conference on Networks.

22. Floodlight project. Available online: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview.

23. RYU. Available online: https://github.com/osrg/ryu.

24. OpenDaylight Technical Overview. Available online: https://www.opendaylight.org/project/technical-overview.

25. Jarraya, Y. Madi, T. and Debbabi, M. 2014. A Survey and a Layered Taxonomy of SDN. IEEE Communications Surveys and Tutorials.

26. Imran, H. Yesvanth, R. and Yuvarajapathi, V. Implementing OpenFlow, Exploring the Present and Future SDN Ecosystem. IEEE.

27. Open Networking Foundation (ONF). 2012. OpenFlow Switch Specification. Version 1.3.0.

28. Bholebawa, I, Z. Dalal, U, D. 2017. Performance Analysis of SDN/OpenFlow Controllers: POX Versus Floodlight.

29. Tu, W. Wei, Y, H. Antichi, G. and Pfaff, B. Revisiting the Open vSwitch Dataplane Ten years Later. Proceedings of the 2021 ACM SIGCOMM 2021 Conference.

30. Shinoda, A, A. et al. 2014. Using Mininet for emulation and prototyping SDN. IEEE Conference on Communications and Computing.

31. Kaur, K. Singh, J. and Ghumman, N, S. 2014. Mininet as Software Defined Networking Testing Platform.