المدرسة العليا للتكنولوجيا - كلميم

ÉCOLE SUPÉRIEURE DE TECHNOLOGIE - GUELMIM

École Supérieure de Technologie de Guelmim

Program: Computer Networks and Security (RIS)

FINAL YEAR PROJECT REPORT

Study, Design, and Implementation of an

SDN Architecture

(Software Defined Networking)

Realized by: Abderrahmane Aroussi Amin Mriroud	**Supervised by:** Pr. Tarek AIT BAHA

Jury Members:

Pr. Amine Bouaouda • Pr. Tarek AIT BAHA

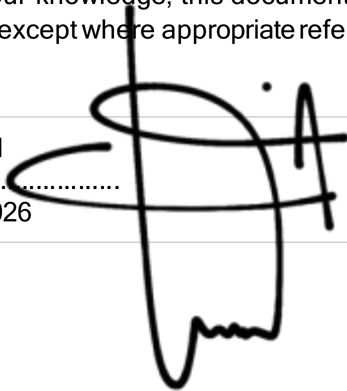
Academic Year: 2025–2026

Declaration

We hereby declare that this work is original and has not been previously submitted for any degree at this institution or any other higher education establishment.

To the best of our knowledge, this document contains no material previously published or written by another person, except where appropriate references have been provided.

Amin Mriroud Signature: .... Date: 24/03/2026	**Abderrahmane Aroussi** Signature: .... Date: 24/03/2026



Acknowledgements

We would like to express our sincere gratitude to our supervisor, Mr. Tarek AIT BAHA, for his invaluable guidance, availability, and unwavering support throughout this project.

We also extend our thanks to all the faculty members of the Computer Engineering Department at the École Supérieure de Technologie de Guelmim for the knowledge they imparted during our academic training. In particular, we thank Pr. Amine Bouaouda for his course on Virtualization and Cloud Computing, which provided us with essential background on containerization technologies such as Docker, used in this project.

Finally, we express our heartfelt appreciation to our families and friends for their continuous encouragement and steadfast support.

Résumé

Les réseaux traditionnels présentent des limitations en termes de flexibilité, d'automatisation et de gestion centralisée. Le paradigme des réseaux définis par logiciel (SDN) propose de dissocier le plan de contrôle du plan de données, offrant ainsi une vue globale et une programmabilité accrue du réseau. Ce projet, réalisé dans le cadre de la filière Réseaux Informatiques et Sécurités (RIS) à l'École Supérieure de Technologie de Guelmim, a pour objectif d'étudier, de concevoir et de mettre en place une architecture SDN à l'aide du contrôleur OpenDaylight et de l'émulateur Mininet.

Dans un premier temps, nous avons déployé l'environnement virtuel nécessaire (Ubuntu, Docker, Mininet). Ensuite, nous avons configuré le contrôleur OpenDaylight et créé différentes topologies réseau (un et deux commutateurs) afin de tester la connectivité et les performances. Les résultats obtenus montrent que le contrôleur réagit correctement aux événements réseau (phase d'apprentissage) et que le débit mesuré est proche des limites théoriques des liens.

Ce travail constitue une base solide pour explorer des aspects plus avancés du SDN, tels que l'utilisation de multiples contrôleurs ou la mise en œuvre de fonctions de sécurité, en parfaite adéquation avec les objectifs de la filière RIS.

Mots-clés : SDN, OpenDaylight, Mininet, OpenFlow, Réseaux définis par logiciel, Sécurité.

Abstract

Traditional networks face significant limitations in terms of flexibility, automation, and centralized management. The Software-Defined Networking (SDN) paradigm addresses these limitations by decoupling the control plane from the data plane, thereby providing a global network view and enhanced programmability. This project, conducted within the Computer Networks and Security (RIS) program at the École Supérieure de Technologie de Guelmim, aims to study, design, and implement an SDN architecture using the OpenDaylight controller and the Mininet network emulator.

In the first phase, the required virtual environment was set up (Ubuntu, Docker, Mininet). Subsequently, the OpenDaylight controller was configured, and various network topologies (single and dual switch) were created to evaluate connectivity and performance. The results demonstrate that the controller correctly responds to network events during the learning phase, and that the measured throughput closely approaches the theoretical link capacity.

This work provides a solid foundation for exploring more advanced SDN aspects, such as multi-controller deployments or the implementation of network security functions, in full alignment with the objectives of the RIS program.

Keywords: SDN, OpenDaylight, Mininet, OpenFlow, Software-Defined Networking, Security.

Table of Contents

Declaration 2
Acknowledgements 3
Résumé 4
Abstract 5
Table of Contents 6
List of Figures 7
List of Tables 8
Chapter 1: Introduction 9
1.1 General Context 9
1.2 Motivation 9
1.3 Problem Statement 9
1.4 Objectives 9
1.5 Project Scope 9
1.6 Report Structure 10
Chapter 2: Fundamentals of SDN Networks 11
2.1 Traditional vs. SDN Networks 11
2.2 SDN Architecture: Control Plane and Data Plane 12
2.3 Role of the SDN Controller 12
2.4 OpenFlow Protocol 13
2.5 OpenDaylight Controller 14
2.6 Emulation Tool: Mininet 15
2.7 Summary 15
2.8 Containerization and Docker 15
2.9 Real-World SDN Application Scenarios 17
Chapter 3: Experimental Configuration and Results 20
3.1 Experimental Environment 20
3.2 Experimental Results and Evaluation 24
3.3 Discussion 36
3.4 Conclusion 37
Chapter 4: Discussion and Conclusion 38
4.1 Discussion of Results 38
4.2 Conclusion 39
4.3 Perspectives and Future Work 39
References 40

List of Figures

Figure 2.1: Traditional network with distributed protocols.
Figure 2.2: Comparison of traditional and SDN network architectures [17].
Figure 2.3: Separation of the control plane and data plane in an SDN network.
Figure 2.4: Request path from Host X to Host Y via the controller.
Figure 2.5: Communication between an OpenFlow switch and the controller.
Figure 2.6: Architecture of the OpenDaylight controller.
Figure 2.7: Docker Architecture (Source: Course Material, Pr. Amine Bouaouda).
Figure 2.8: Software-Defined VANET Communications[17].
Figure 2.9: OpenSAN architecture for satellite networks [17].
Figure 2.10: SDN-based UAV network illustration [17].
Figure 3.1: Downloading the OpenDaylight Docker image.
Figure 3.2: First Mininet attempt with a remote controller – controller unreachable at 127.0.0.1:6633, pingall shows 100% loss.
Figure 3.3: Container logs showing the 'String index out of range' error.
Figure 3.4: Successful installation of odl-restconf and odl-l2switch-switch features in the Karaf console.
Figure 3.5: Two-switch, four-host topology. The controller is odl1.
Figure 3.6: First pingall after launching Mininet – 50% packet loss (learning phase).
Figure 3.7: Second pingall – all flows are known, zero packet loss.
Figure 3.8: pingall on the two-switch topology – full connectivity achieved.
Figure 3.9: Output of the net command showing the links between hosts and switches.
Figure 3.10: Wireshark capture showing OpenFlow exchanges between the controller and switches.
Figure 3.11: Detailed analysis of an OpenFlow PACKET_IN frame.
Figure 3.12: Sequence of exchanged OpenFlow messages.
Figure 3.13: Flow table on switch s1 after ping tests.
Figure 3.14: Flow table on switch s2.
Figure 3.15: iPerf result between h1 and h4 (traversing both switches) – 585 Mbit/s.
Figure 3.16: iPerf result between h1 and h2 (same switch) – 640 Mbit/s.
Figure 3.17: Downloading MiniEdit from the Mininet repository.
Figure 3.18: Import error when running MiniEdit: StrictVersion cannot be imported.
Figure 3.19: Three-switch, Twelve-host topology. The controller is odl1/2/3.
Figure 3.20: pingall result on the 6-switch, 12-host topology with three controllers – 0% loss.
Figure 3.21: Ping from h1 to h2 (same switch).
Figure 3.22: Ping from h1 to h6 (different switches).
Figure 3.23: Ping from h1 to h12 (across the network).
Figure 3.24: iPerf throughput between h1 and h6 – 9.81 Gbits/sec.

List of Tables

Table 3.1: Average TCP throughput for different traffic types.
Table 3.2: Failure rate of OpenDaylight in latency tests [16].

Chapter 1: Introduction

1. General Context

Traditional computer networks rely on equipment – such as routers and switches – that integrates both the control plane and the data plane within the same device. Each piece of equipment makes its own forwarding decisions using distributed protocols (OSPF, BGP, STP, etc.), which renders network evolution and management increasingly complex, particularly in large-scale infrastructures. In response to these limitations, the Software-Defined Networking (SDN) paradigm has emerged as a promising solution. SDN decouples the control plane from the data plane and centralizes network intelligence within a software-based controller. This separation provides a global network view, enhanced programmability, and the flexibility to innovate more rapidly.

2. Motivation

Our motivation for undertaking this project stems from the growing need for automation and flexibility in modern networks. Network administrators are often required to manually configure each device, which is error-prone and poorly reactive to traffic changes. SDN promises to simplify this management through a centralized controller capable of dynamically adapting forwarding rules.

Furthermore, as part of our academic training, we sought to acquire practical skills in these emerging technologies. Setting up an SDN laboratory using open-source tools (Mininet and OpenDaylight) allowed us to gain a concrete understanding of theoretical concepts and to evaluate the performance of such an architecture.

3. Problem Statement

How can an SDN architecture, based on the OpenDaylight controller and the Mininet emulator, improve automation, flexibility, and network control compared to a traditional architecture?

4. Objectives

The primary objective of this project is to study, design, and implement an SDN network prototype. More specifically, we aim to:

- Understand the fundamental concepts of SDN and the OpenFlow protocol.
- Deploy a virtual environment using Mininet to emulate a network topology.
- Install and configure the OpenDaylight controller.
- Create programmable routing rules and observe their installation on switches.
- Conduct comparative tests of connectivity (ping) and performance (throughput with iPerf) between an SDN architecture and a simulated traditional architecture.
- Document the challenges encountered and the solutions implemented.

5. Project Scope

This project is limited to a study conducted in a virtualized environment. Mininet was used to create small-scale topologies (up to two switches and four hosts) in the initial tests, and later extended to a multi-controller setup with six switches and twelve hosts to evaluate scalability. OpenDaylight served as the controller. Tests focus on basic connectivity and TCP throughput measurement. Advanced aspects – such as security, high availability with multiple controllers, or integration with physical hardware – are beyond the scope of this work. Nevertheless, the work carried out constitutes a solid foundation for future extensions.

1.6 Report Structure

The report is organized as follows:

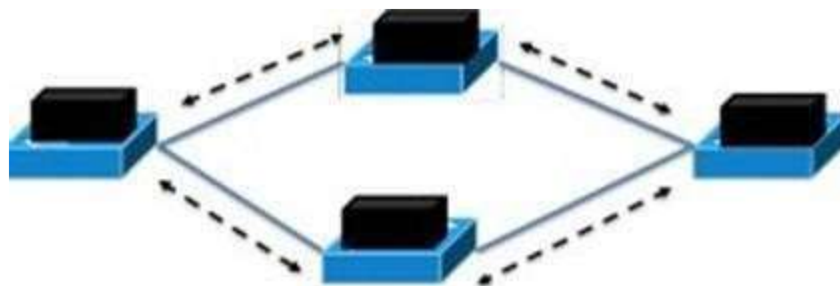
- Chapter 2: Fundamental Concepts – key notions of SDN, the OpenFlow protocol, the OpenDaylight controller, and the Mininet emulation tool. Additionally, we introduce containerization and Docker, as well as real-world SDN application scenarios.
- Chapter 3: Experimental Configuration and Results – the environment set up, deployment steps, tests performed, and results obtained, including a multi-controller experiment.
- Chapter 4: Discussion and Conclusion – analysis of results, identified limitations, and directions for future work.

Chapter 2: Fundamentals of SDN Networks

This chapter presents the foundational concepts required to understand a Software-Defined Network (SDN). We begin by comparing the traditional architecture with that of SDN, then detail the essential components: the control plane, the data plane, the role of the controller, the OpenFlow protocol, and the OpenDaylight controller. Finally, we introduce the Mininet emulation tool used for practical implementation, followed by an overview of containerization and Docker, and a survey of real-world SDN applications.

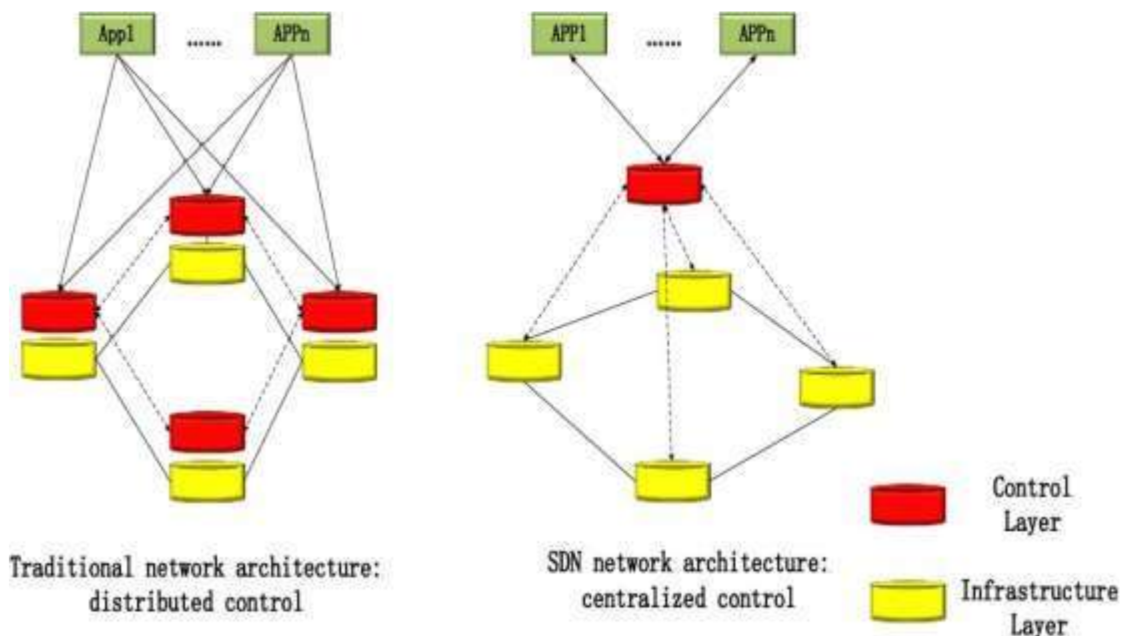
2.1 Traditional Networks vs. SDN Networks

In a traditional network, each device (router, switch) executes distributed protocols to make forwarding decisions autonomously. The control plane and the data plane are integrated within the same device, which makes network evolution and management increasingly complex. Figure 2.1 illustrates this classical architecture.



[Figure 2.1 – Traditional network with distributed protocols]

SDN (Software-Defined Networking) introduces a clear separation between the control plane and the data plane. The control plane is delegated to a centralized software entity called the controller, while the switches (data plane) simply forward packets according to the rules received from the controller. This separation brings flexibility, programmability, and a global view of the network.



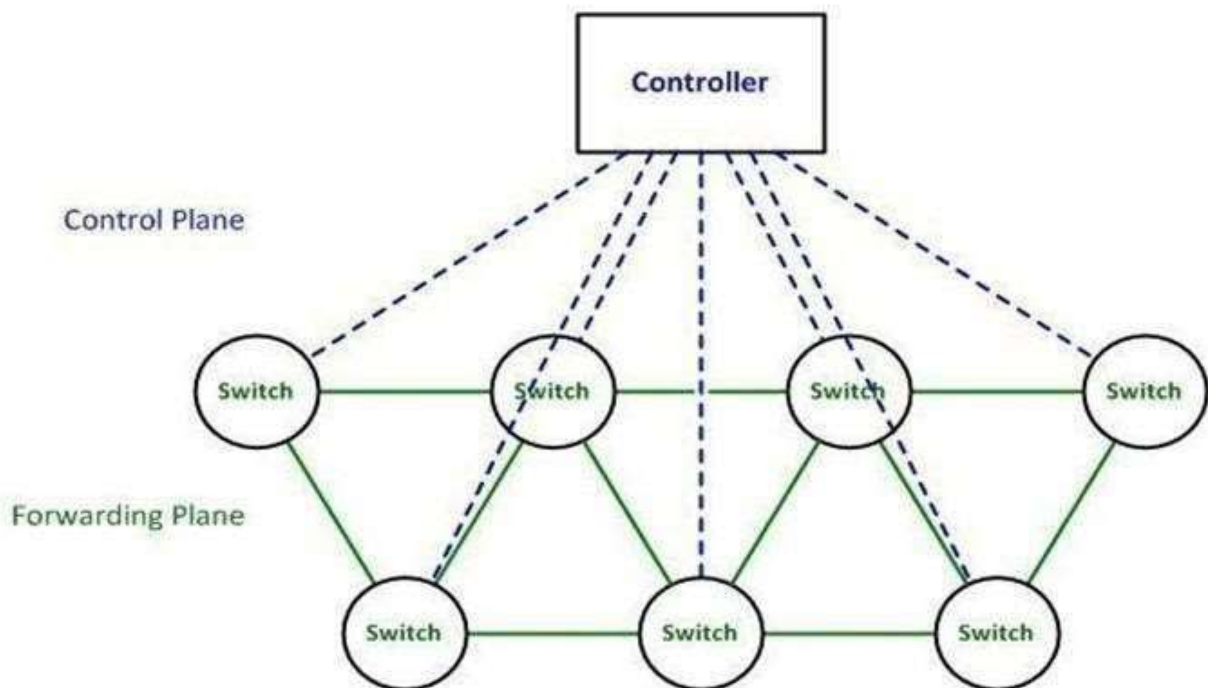
[Figure 2.2 – Comparison of traditional and SDN network architectures [17]]

2.2 SDN Architecture: Control Plane and Data Plane

The SDN architecture comprises three main layers:

- Data Plane: composed of switches (physical or virtual) that forward packets.
- Control Plane: the SDN controller, which centralizes network logic and makes forwarding decisions.
- Application Plane: network applications that interact with the controller through northbound APIs.

Figure 2.3 illustrates this separation.

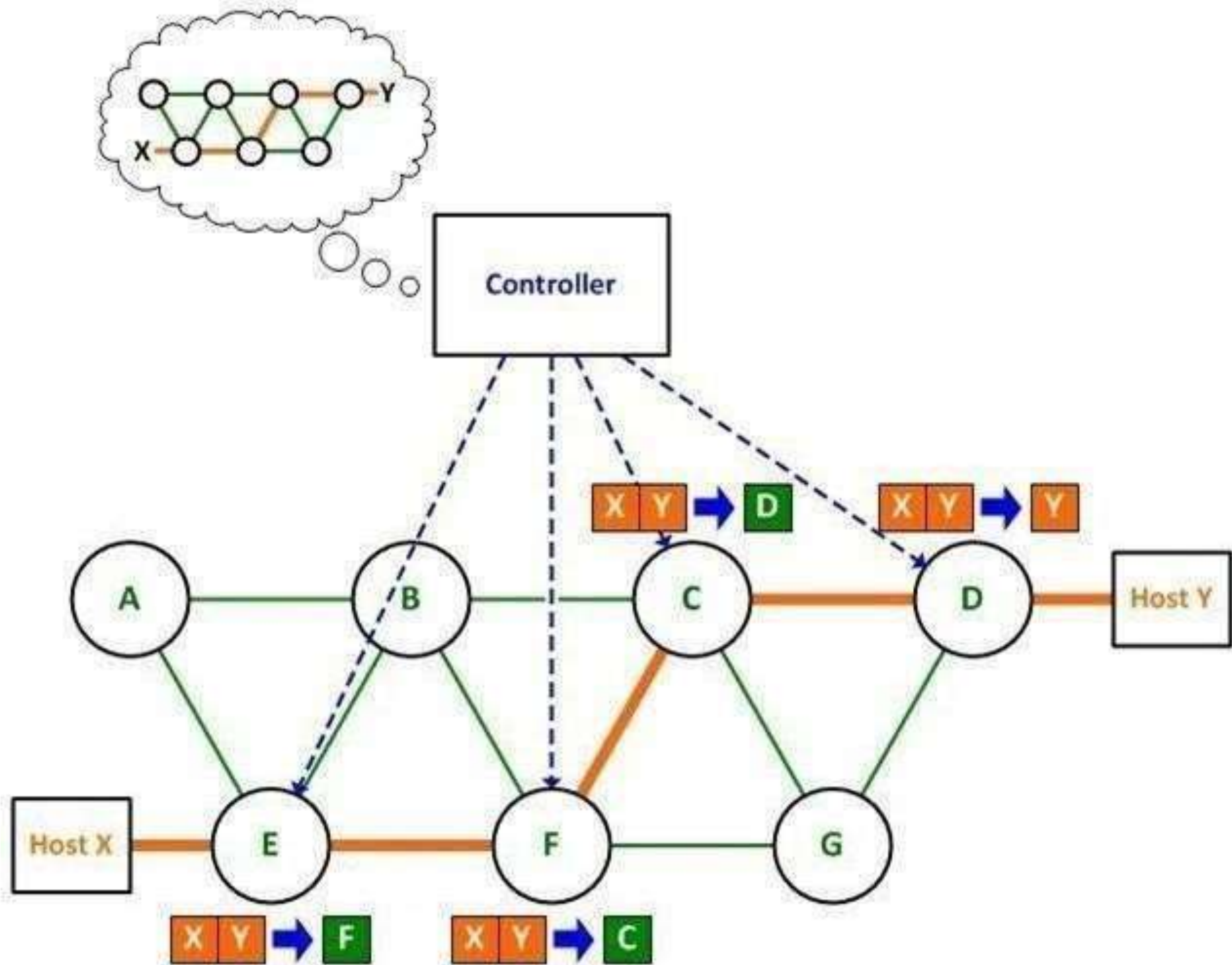


[Figure 2.3 – Separation of the control plane and data plane in an SDN network]

Through this architecture, the controller maintains a complete view of the network topology and can dynamically adapt forwarding rules.

2.3 Role of the SDN Controller

The controller is the brain of the network. It receives requests from switches (e.g., the first packet of an unknown flow) and installs flow rules (flow entries) into their flow tables. Figure 2.3 illustrates the path of a request between two hosts.



[Figure 2.4 – Request path from Host X to Host Y via the controller]

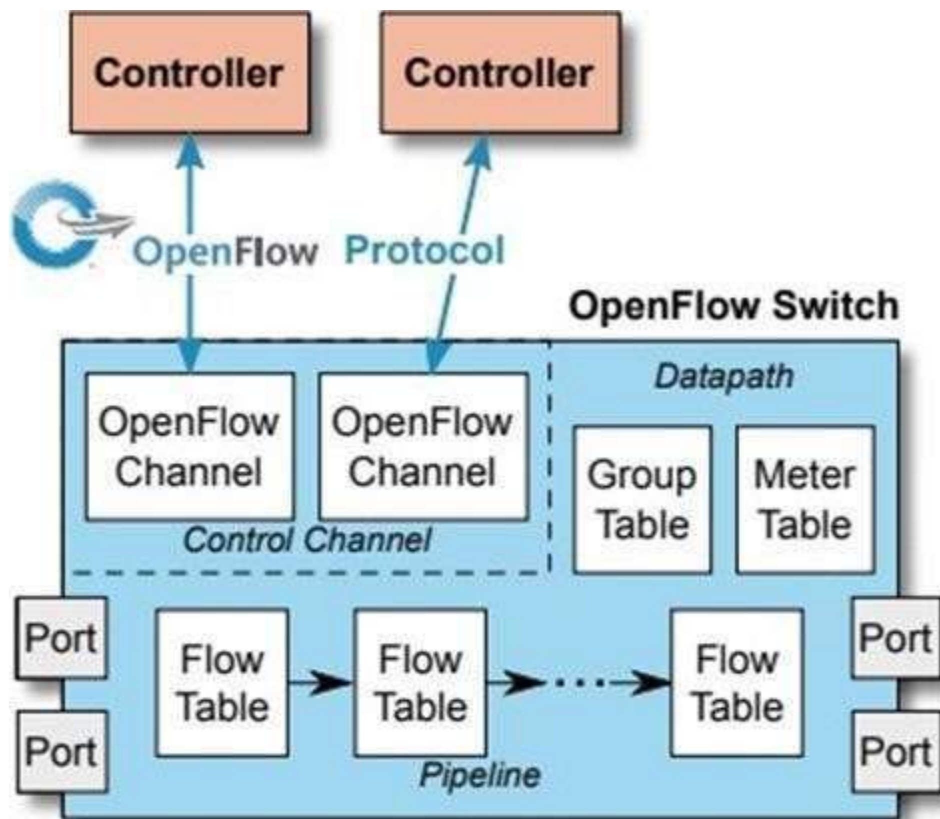
The controller can operate in reactive mode (installing rules on demand) or proactive mode (pre-installing rules). In our project, OpenDaylight was used in reactive mode for connectivity testing.

2.4 OpenFlow Protocol

OpenFlow is the standard communication protocol between the controller and the switches. It allows the controller to add, modify, or delete entries in the flow tables of the switches. A flow entry contains:

- Match fields (MAC addresses, IP addresses, ports, etc.),
- A priority value,
- Counters,
- Instructions (forward, drop, send to controller, etc.).

Figure 2.5 shows the interaction between an OpenFlow switch and the controller.



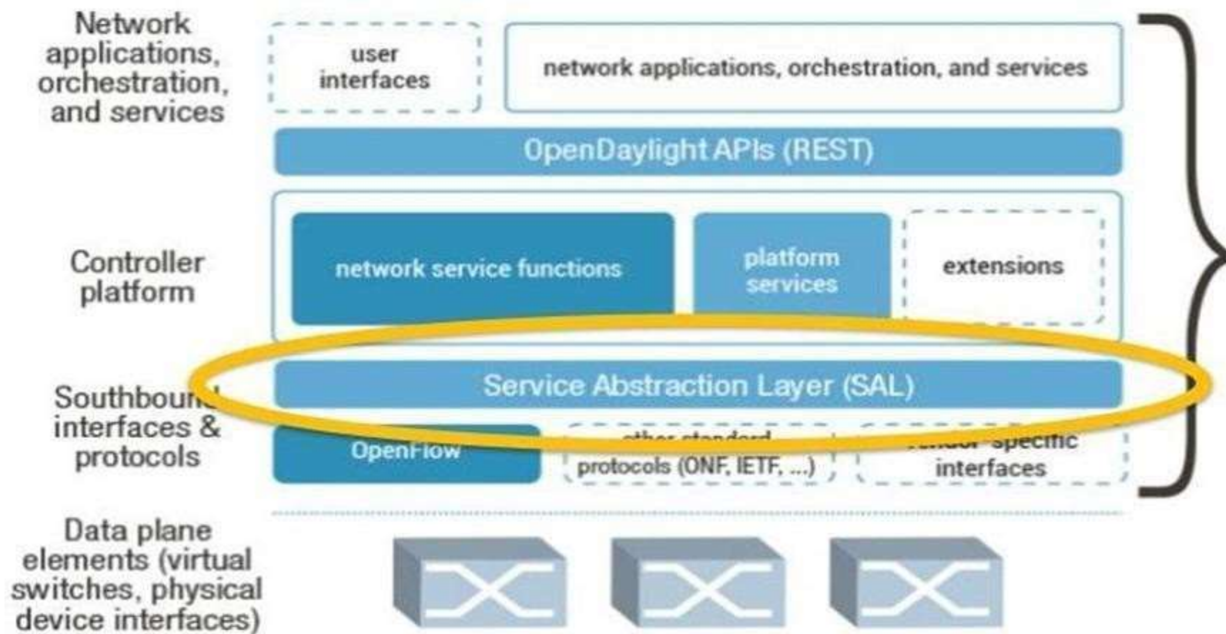
[Figure 2.5 – Communication between an OpenFlow switch and the controller]

In our environment, the switches created by Mininet support OpenFlow and communicate with OpenDaylight through this protocol.

2.5 OpenDaylight Controller

OpenDaylight (ODL) is a mature open-source SDN controller, written in Java and based on a modular OSGi architecture. It exposes northbound REST APIs and supports numerous southbound protocols (including OpenFlow). Its flexibility and widespread adoption make it well-suited for academic projects.

Figure 2.6 presents the internal architecture of OpenDaylight with its Service Abstraction Layer (SAL).



[Figure 2.6 – Architecture of the OpenDaylight controller [16]]

We deployed the ODL distribution inside a Docker container to facilitate deployment. The core features activated were odl-restconf and odl-l2switch-switch (Layer 2 switch functionality).

6. Emulation Tool: Mininet

Mininet is a network emulator that allows the creation of a network of virtual switches and hosts on a single machine. It leverages lightweight virtualization (Linux network namespaces) to run real network processes. Its key advantages include:

- Rapid prototyping,
- Flexibility through Python scripting,
- Native OpenFlow support,
- Reproducibility of experiments.

In this work, Mininet was used to emulate the network topologies (switches, hosts, links) and to generate traffic using the ping and iperf commands.

7. Summary

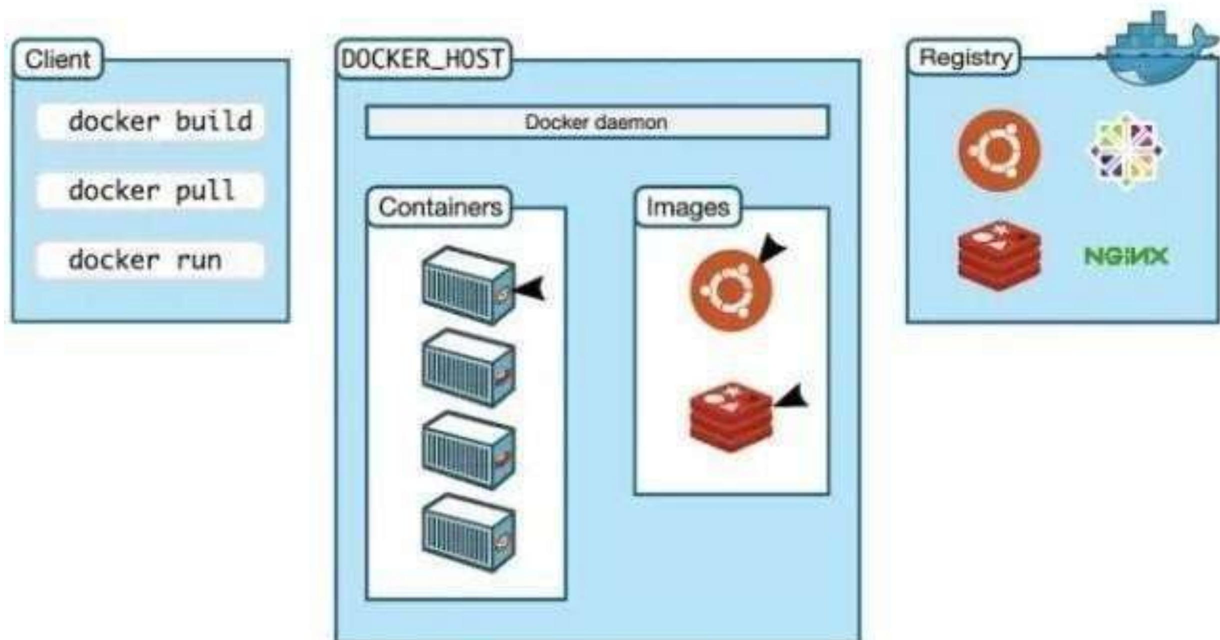
This chapter has laid the theoretical foundations necessary for the experimental implementation. We have seen how SDN differs from traditional networks through the separation of the control plane, the central role of the controller, the OpenFlow protocol, and the Mininet tool. OpenDaylight was selected for its robustness and modularity.

8. Containerization and Docker

Containerization is a lightweight operating system-level virtualization technique that allows multiple isolated applications to run on a single host system by sharing the host kernel. Each container packages an application together with its dependencies (libraries, configuration files, runtime), ensuring consistent execution across different environments. This approach offers significant advantages in terms of portability, rapid deployment, and resource efficiency, making it a cornerstone of modern cloud-native architectures, microservices, and DevOps practices.

Docker, introduced in 2013 by Docker Inc., is the most widely adopted containerization platform. It automates the creation, deployment, and management of containers. A Docker image is a read-only template containing the application and its dependencies, built from instructions in a Dockerfile. When executed, this image becomes a running container instance. Docker leverages Linux kernel features such as namespaces (for isolation) and cgroups (for resource limitation) to provide efficient and secure containerization [18]. The platform also includes Docker Hub, a public registry for sharing images, and Docker Engine, the runtime that manages containers on a host system.

Figure 2.7 illustrates the high-level architecture of Docker.

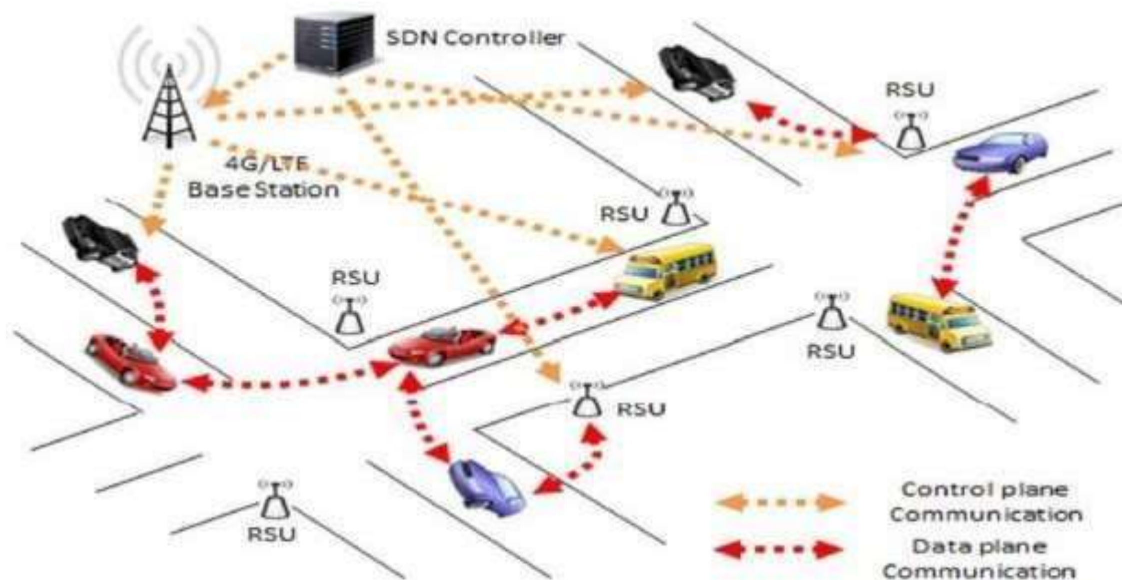


[Figure 2.7 – Docker Architecture. Source: Course Material, Pr. Amine Bouaouda]

In this project, Docker was employed to run the OpenDaylight controller within an isolated and reproducible container environment, simplifying deployment and avoiding host system conflicts.

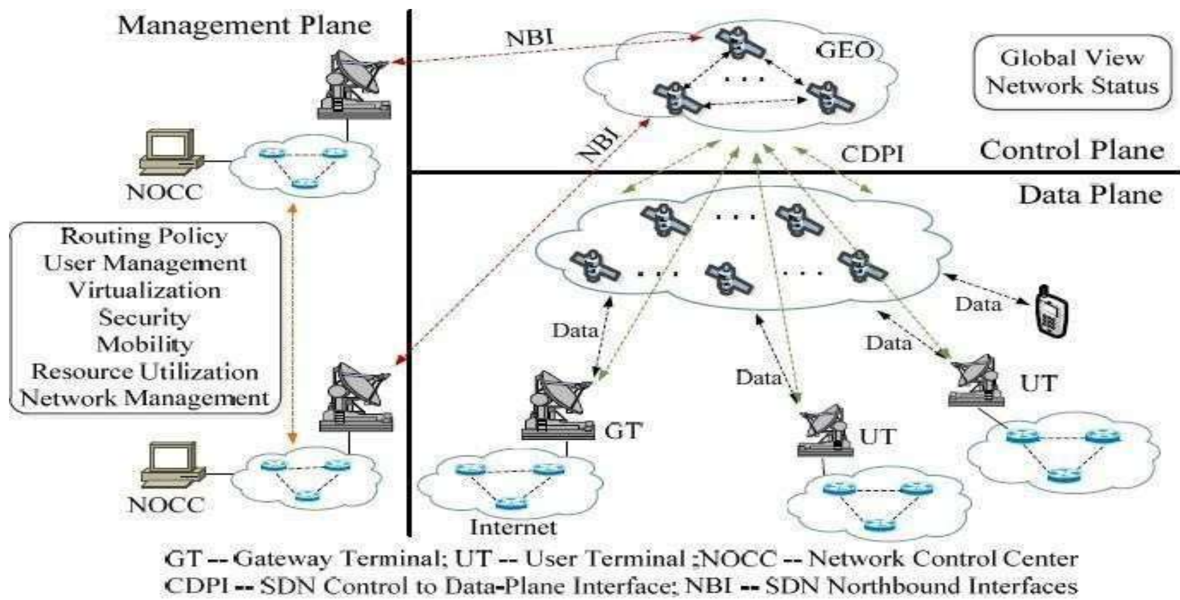
2.9 Real-World SDN Application Scenarios

SDN has been deployed in numerous real-world contexts. Notable examples include Google's B4 wide-area network, which uses SDN to optimize traffic engineering across its global backbone. OpenSAN provides SDN-based networking for satellite communication systems, while researchers have explored SDN-based UAV (drone) network architectures to provide dynamic control of unmanned aerial vehicle fleets. These scenarios demonstrate the broad applicability of SDN beyond traditional data center environments.



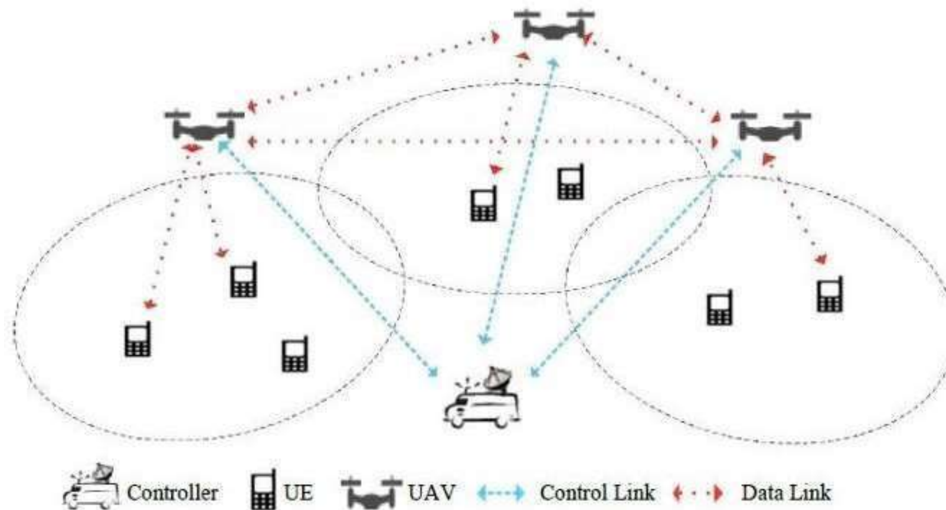
[Figure 2.8 – Software-Defined VANET Communications]

This figure illustrates the application of SDN in Vehicular Ad-Hoc Networks (VANETs). In this architecture, a centralized SDN controller manages communication between vehicles (RSUs - Roadside Units) and a central infrastructure (like a 4G/LTE base station). The control plane, handling decision-making, is separated from the data plane, which forwards actual vehicle data. This setup allows for dynamic traffic management, improved safety through low-latency warnings, and efficient content delivery to moving vehicles. For our project, this scenario highlights the potential for extending our static SDN prototype to mobile environments, where the controller must rapidly adapt to a constantly changing network topology.



[Figure 2.9 – OpenSAN architecture for satellite networks [17]]

This figure presents the OpenSAN architecture, which applies SDN to satellite networks. A global Network Management system acts as the SDN controller, providing a centralized view of the entire network, which includes ground terminals, gateways, and satellites. The separation of the control and data planes allows for features like virtualization, mobility management, and efficient resource utilization across the vast satellite infrastructure. This is relevant to our project as it demonstrates how the same SDN principles we implemented (using OpenDaylight as a controller) can be scaled to manage complex, non-terrestrial networks, moving beyond the small-scale, wired topologies we emulated with Mininet.



[Figure 2.10 – SDN-based UAV network illustration [17]]

This figure depicts a software-defined network of Unmanned Aerial Vehicles (UAVs), or drones. An SDN controller (which could be on the ground or part of the fleet) maintains a logical control link to each UAV, while data links manage the payload communication. This setup enables dynamic path planning, coordinated flight, and adaptive communication based on mission requirements. Our project's architecture—with a centralized controller managing a network of switches—serves as a foundational model for such a system. The successful multi-controller experiment we conducted, which demonstrated scalability and high throughput, is a crucial step toward the more dynamic and complex coordination required for managing a fleet of UAVs.

Chapter 3: Experimental Configuration and Results

This chapter describes the experimental environment, the tools used, the deployment procedure, and the results obtained during the basic testing of an SDN network. The practical approach consists of deploying an OpenDaylight controller inside a Docker container, emulating a network with Mininet, and then evaluating connectivity and throughput. Screenshots captured during the manipulations are incorporated as figures to illustrate each step.

1. Experimental Environment

1. Tools and Technologies

The following tools were employed:

- Mininet 2.3.0: A network emulator that allows the creation of hosts, switches, and virtual links. It supports OpenFlow and facilitates rapid prototyping.
- OpenDaylight (ODL): An open-source SDN controller, run inside a Docker container to ensure a clean and reproducible environment.
- Docker: Used to launch and manage the ODL container.
- iPerf3: A tool for measuring network throughput.
- Wireshark: Used to capture and analyze OpenFlow control messages.
- dpctl / ovs-ofctl: Utilities to inspect the flow tables of Open vSwitch switches.
- Python 3: Used to write custom Mininet topology scripts and to attempt the use of MiniEdit.

The host machine is an Ubuntu 24.04 virtual machine equipped with 4 CPU cores and 8 GB of RAM.

2. Retrieving the OpenDaylight Docker Image

The OpenDaylight image was obtained from Docker Hub using the following command:

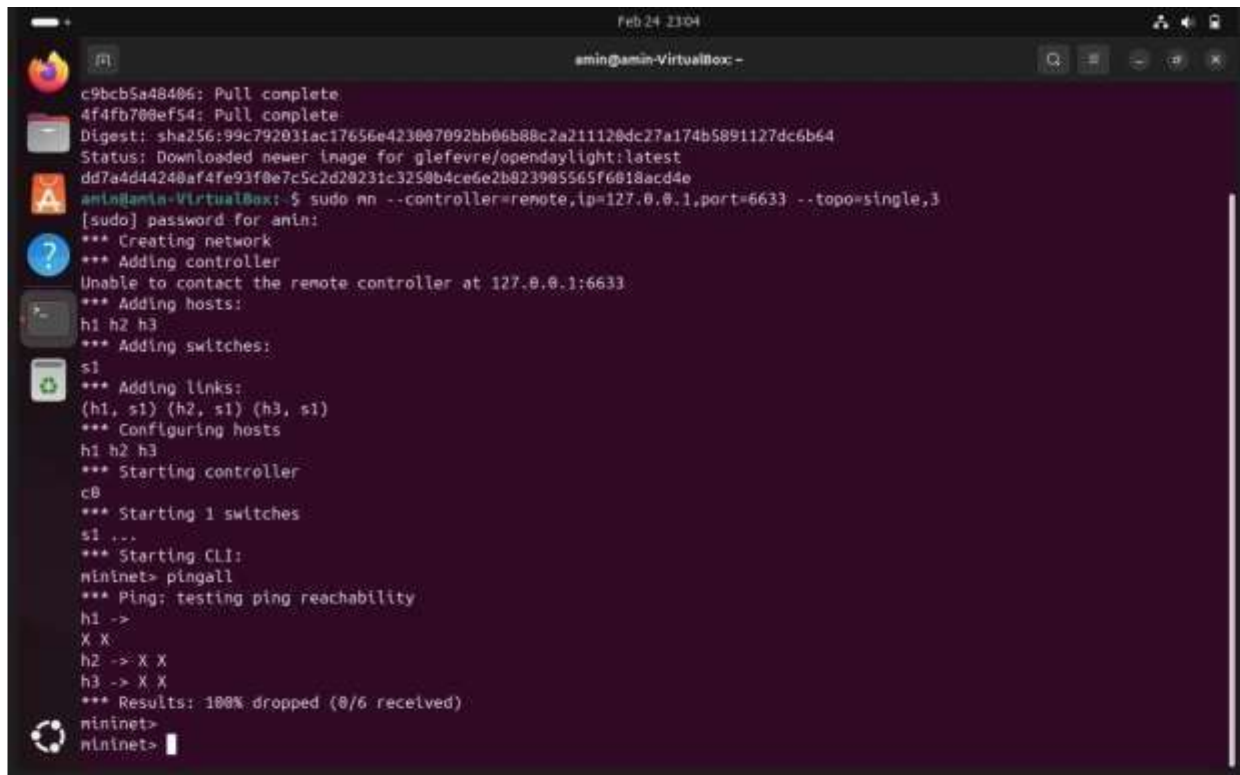
```
docker run -d -p 6633:6633 -p 6653:6653 -p 8181:8181 \
    --name odl-controller glefevre/opendaylight
```

Figure 3.1 illustrates the download process.

[Figure 3.1 – Downloading the OpenDaylight Docker image]

3.1.3 Container Startup Issue and Diagnosis

After the download, the container was launched but did not remain active due to a script error. The first connection attempt from Mininet failed (Figure 3.2), as the controller was unreachable (100% packet loss).



[Figure 3.2 – First Mininet attempt with remote controller – controller unreachable at 127.0.0.1:6633, pingall shows 100% loss]

Examination of the container logs (Figure 3.3) revealed the following error:

```
Error in initialization script: /odl/etc/shell.init.script: String index out of
range: 0
```

This error, known to affect certain versions of OpenDaylight, prevents the Karaf shell from starting correctly.

[Figure 3.3 – Container logs showing the 'String index out of range' error]

To resolve the issue, the container was deleted and recreated in interactive mode using the -itd options:

```
docker rm -f odl-controller
docker run -itd -p 6633:6633 -p 6653:6653 -p 8181:8181 \
    --name odl-controller glefevre/opendaylight
```

Access to the container then allowed the installation of the core features (Figure 3.4):

```
docker exec -it odl-controller ./bin/client feature:install odl-restconf odl-
l2switch-switch
```

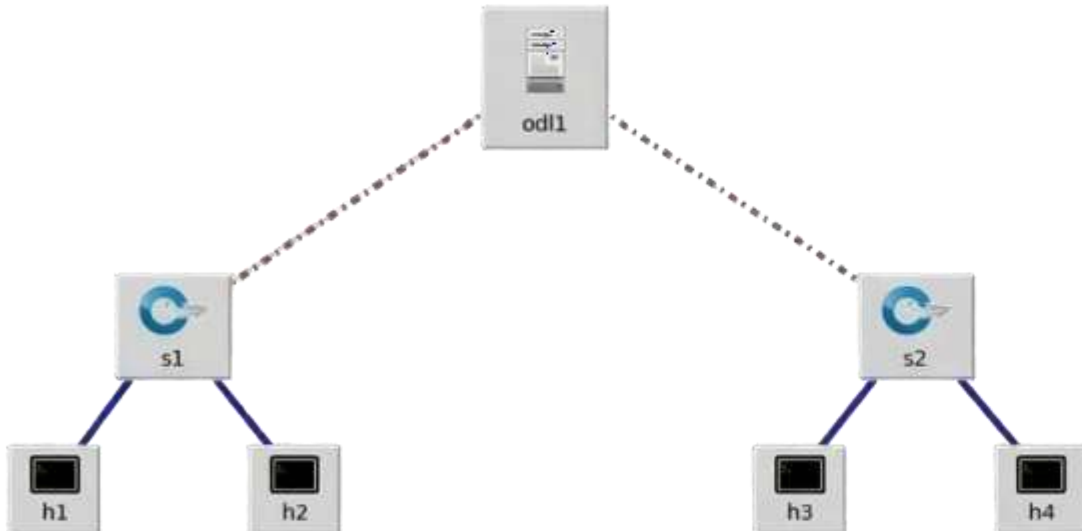
[Figure 3.4 – Successful installation of odl-restconf and odl-l2switch-switch features in the Karaf console]

3.1.4 Network Topologies

Two topologies were used initially:

- One switch with three hosts (for initial connectivity tests).
- Two switches with four hosts (for throughput measurements and flow table inspection).

The second topology is illustrated in Figure 3.5. It comprises switches s1 and s2 connected by a single link, with hosts h1 and h2 attached to s1, and hosts h3 and h4 attached to s2. All links have a bandwidth of 100 Mbit/s and a delay of 2 ms (simulated by Mininet).



[Figure 3.5 – Two-switch, four-host topology. The controller is odl1 (OpenDaylight)]

3.1.5 Test Procedure

Mininet was launched with the remote controller option pointing to the OpenDaylight instance:

```
sudo mn --controller=remote,ip=127.0.0.1,port=6633 --topo=single,3
```

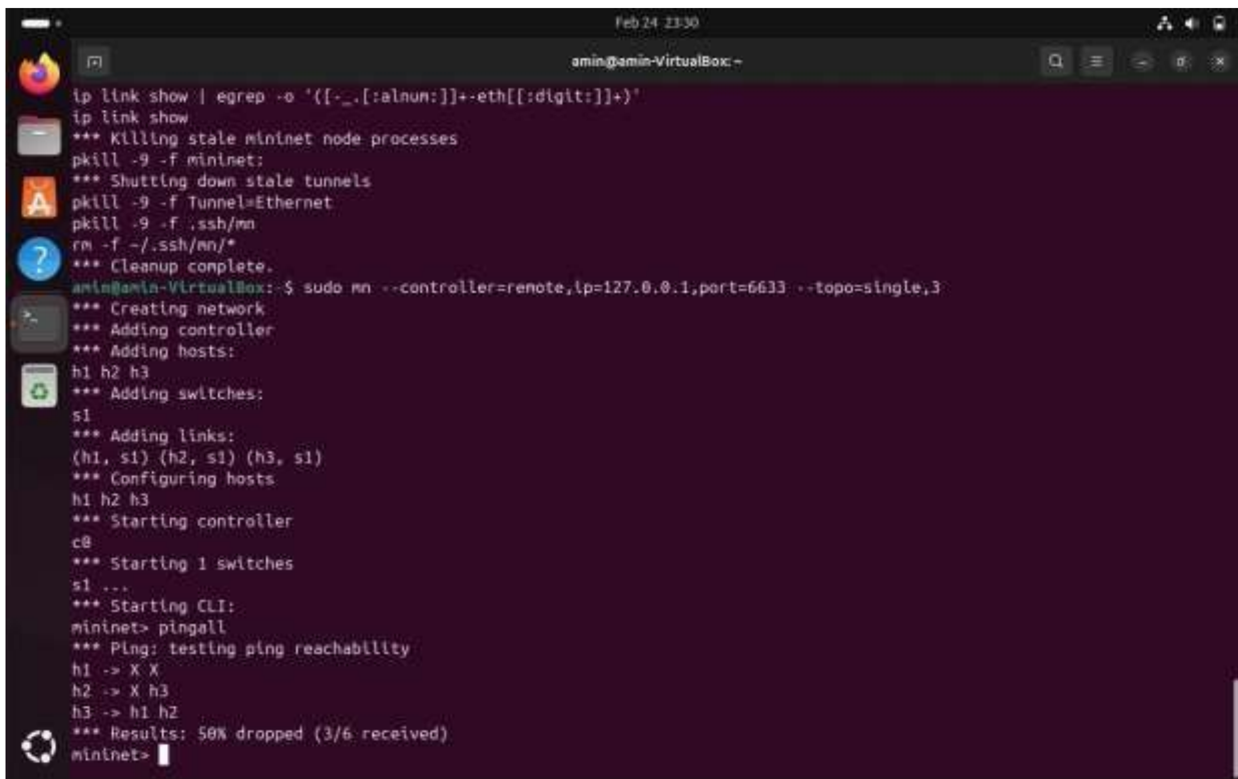
For the two-switch topology, a custom Python script was employed. The commands executed include:

- pingall – to verify connectivity.
- dpctl dump-flows (or ovs-ofctl dump-flows s1) – to inspect flow tables.
- iperf – to measure TCP throughput between hosts.

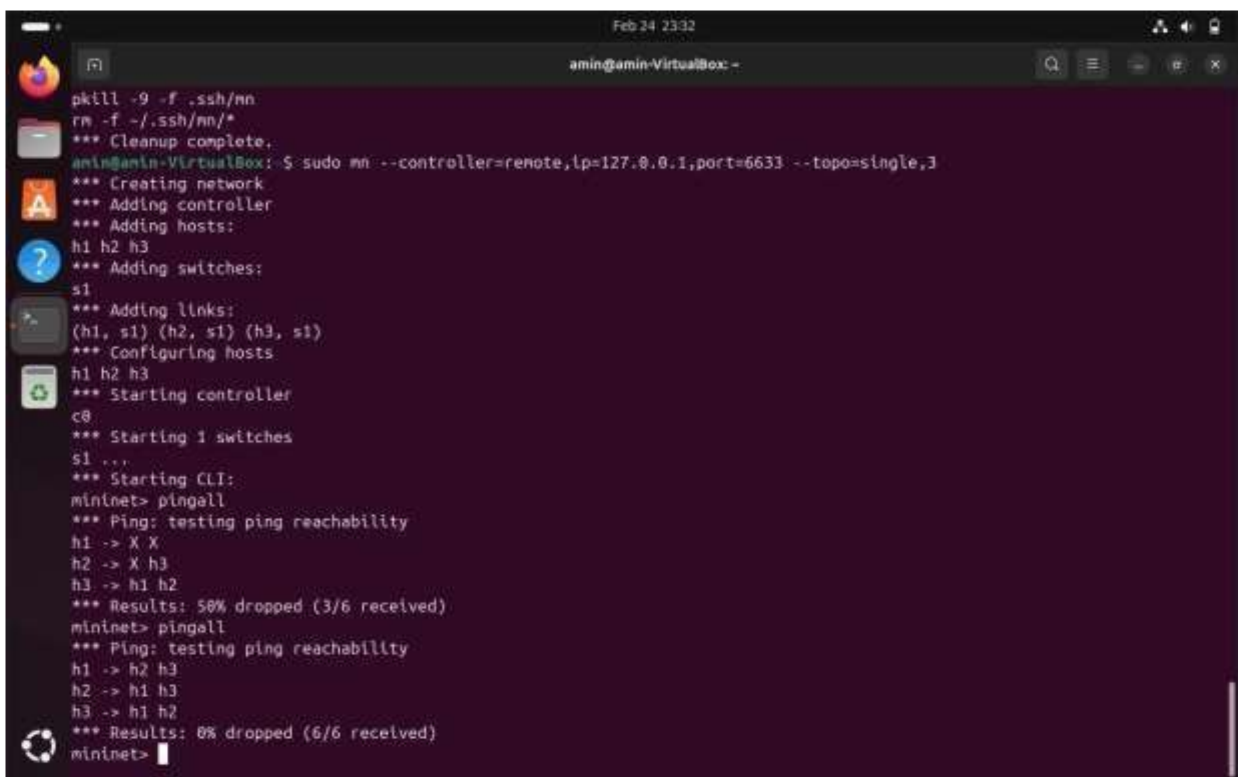
2. Experimental Results and Evaluation

1. Connectivity Tests

The first pingall in the single-switch topology resulted in 50% packet loss (Figure 3.6). This behavior is expected, as the switches do not yet possess flow rules and must query the controller for each new flow. After this initial cycle, the controller installs the necessary rules, and a second pingall shows 0% loss (Figure 3.7).



[Figure 3.6 – First pingall after launching Mininet – 50% packet loss (learning phase)]



[Figure 3.7 – Second pingall – all flows are known, zero packet loss]

For the two-switch topology, pingall also succeeded with 0% loss after the learning phase (Figure 3.8). The net command confirmed the correct connections (Figure 3.9).

```
Feb 24 23:44
amin@amin-VirtualBox: ~/Documents
*** Cleanup complete.
amin@amin-VirtualBox:~/Documents$ sudo mn --custom custom_topo.py --topo projet_sdn --contoller=remote,ip=127.0.0.1,port
=6633
Usage: mn [options]
(type mn -h for details)

mn: error: no such option: --contoller
amin@amin-VirtualBox:~/Documents$ sudo mn --custom custom_topo.py --topo projet_sdn --controller=remote,ip=127.0.0.1,por
t=6633
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (s1, s2)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

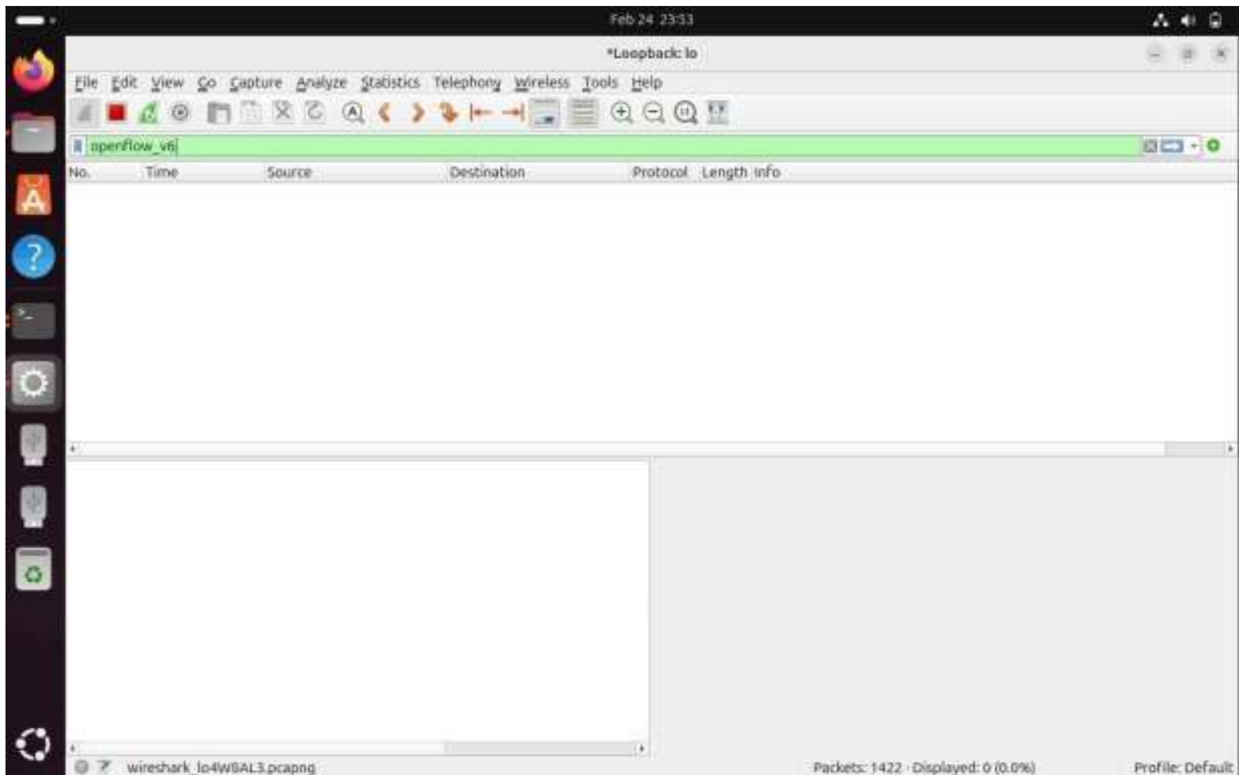
[Figure 3.8 – pingall on the two-switch topology – full connectivity achieved]

```
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s2-eth1
h4 h4-eth0:s2-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:s2-eth3
s2 lo:  s2-eth1:h3-eth0 s2-eth2:h4-eth0 s2-eth3:s1-eth3
c0
mininet> nodes
available nodes are:
c0 h1 h2 h3 h4 s1 s2
mininet>
```

[Figure 3.9 – Output of the net command showing the links between hosts and switches]

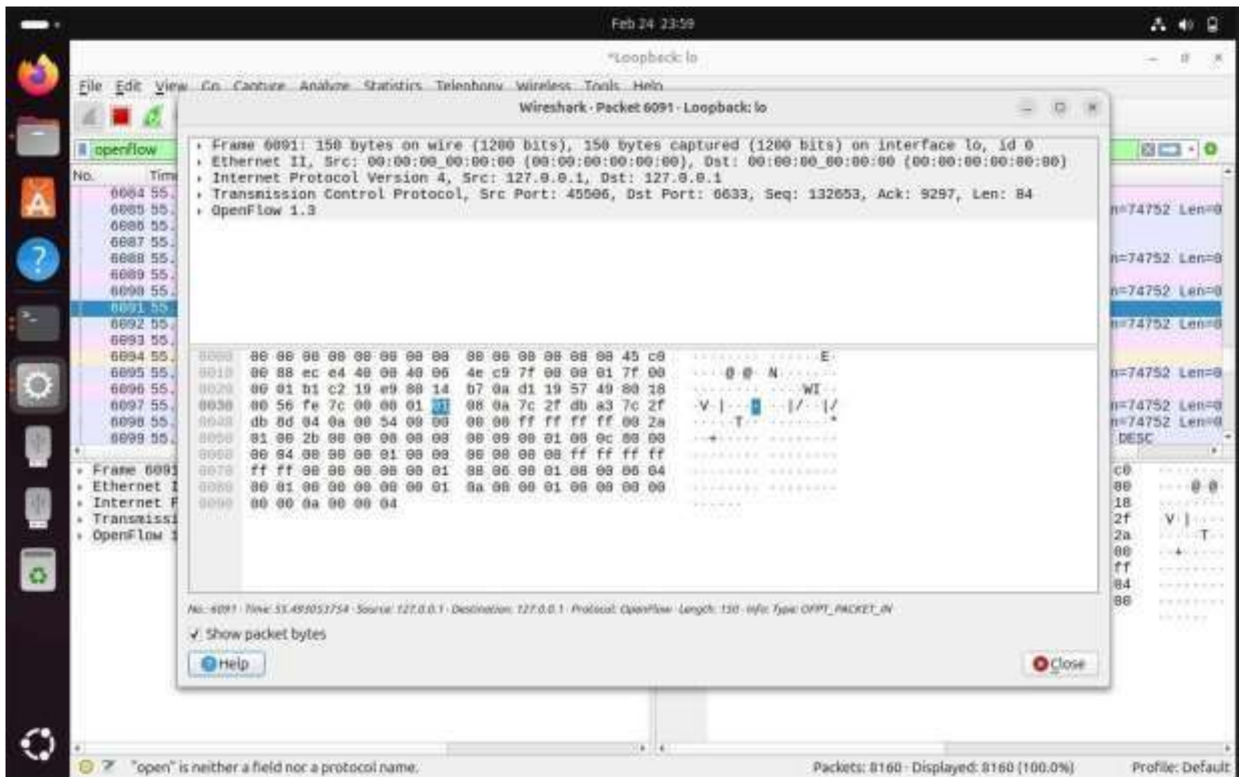
3.2.2 OpenFlow Traffic Analysis

Wireshark was used to capture traffic on the loopback interface (communication between the controller and the switches). Figure 3.10 presents a summary of the OpenFlow packets, including OFPT_PACKET_IN messages (sent when a switch cannot process a packet) and OFPT_MULTIPART_REPLY messages (responses to statistics requests).



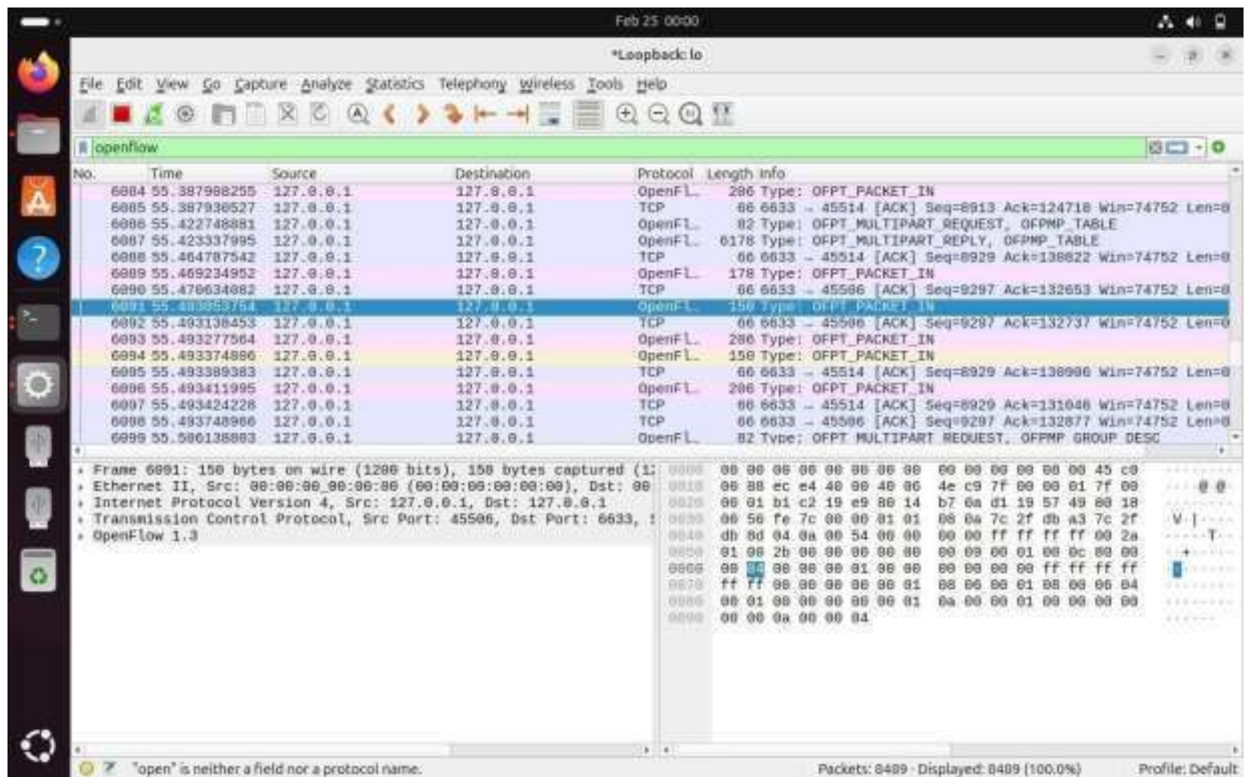
[Figure 3.10 – Wireshark capture showing OpenFlow exchanges between the controller and switches]

Figure 3.11 details a PACKET_IN message, confirming that the switch correctly forwards unknown packets to the controller.



[Figure 3.11 – Detailed analysis of an OpenFlow PACKET_IN frame]

Figure 3.12 lists several OpenFlow messages, notably MULTIPART_REQUEST queries for table statistics, illustrating the control traffic generated during the tests.



[Figure 3.12 – Sequence of exchanged OpenFlow messages]

3.2.3 Flow Table Inspection

After the ping tests, the flow tables of both switches were extracted using dpctl dump-flows (within Mininet) or ovs-ofctl dump-flows s1. Figure 3.13 shows the output for switch s1. It contains:

- Rules for specific MAC addresses (e.g., dl_src=00:00:00:00:00:01, dl_dst=00:00:00:00:00:02) with output actions toward the corresponding port.
- Generic flooding rules for when the destination is unknown (priority 2, actions including CONTROLLER).
- A low-priority drop rule (priority 0).

```
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x2b00000000000003, duration=293.265s, table=0, n_packets=59, n_bytes=5015, priority=100,dl_type=0x88cc actions=
CONTROLLER:65535
 cookie=0x2a0000000000001c, duration=238.917s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
 cookie=0x2a0000000000001d, duration=238.917s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x2b00000000000009, duration=289.181s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s1-eth1" actions
=output:"s1-eth2",output:"s1-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000a, duration=289.170s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s1-eth2" actions
=output:"s1-eth1",output:"s1-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000b, duration=289.170s, table=0, n_packets=41, n_bytes=3092, priority=2,in_port="s1-eth3" actions
=output:"s1-eth1",output:"s1-eth2"
 cookie=0x2b00000000000003, duration=293.203s, table=0, n_packets=5, n_bytes=526, priority=0 actions=drop
*** s2 ------------------------------------------------------------------------
 cookie=0x2b00000000000004, duration=292.954s, table=0, n_packets=58, n_bytes=4930, priority=100,dl_type=0x88cc actions=
CONTROLLER:65535
 cookie=0x2a0000000000001e, duration=237.943s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:04 actions=output:"s2-eth2"
 cookie=0x2a0000000000001f, duration=237.933s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:03 actions=output:"s2-eth1"
 cookie=0x2b0000000000000c, duration=289.267s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s2-eth1" actions
=output:"s2-eth2",output:"s2-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000d, duration=289.234s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s2-eth2" actions
=output:"s2-eth1",output:"s2-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000e, duration=289.223s, table=0, n_packets=41, n_bytes=3092, priority=2,in_port="s2-eth3" actions
=output:"s2-eth1",output:"s2-eth2"
 cookie=0x2b00000000000004, duration=292.990s, table=0, n_packets=5, n_bytes=526, priority=0 actions=drop
mininet>
```

[Figure 3.13 – Flow table on switch s1 after ping tests]

Figure 3.14 presents the flow table on s2, with similar entries for hosts h3 and h4. The n_packets counters indicate the number of packets that matched each rule.

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x2b00000000000003, duration=293.265s, table=0, n_packets=59, n_bytes=5015, priority=100,dl_type=0x88cc actions=
CONTROLLER:65535
 cookie=0x2a0000000000001c, duration=238.917s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
 cookie=0x2a0000000000001d, duration=238.917s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x2b00000000000009, duration=289.181s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s1-eth1" actions
=output:"s1-eth2",output:"s1-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000a, duration=289.170s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s1-eth2" actions
=output:"s1-eth1",output:"s1-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000b, duration=289.170s, table=0, n_packets=41, n_bytes=3092, priority=2,in_port="s1-eth3" actions
=output:"s1-eth1",output:"s1-eth2"
 cookie=0x2b00000000000003, duration=293.203s, table=0, n_packets=5, n_bytes=526, priority=0 actions=drop
*** s2 ------------------------------------------------------------------------
 cookie=0x2b00000000000004, duration=292.954s, table=0, n_packets=58, n_bytes=4930, priority=100,dl_type=0x88cc actions=
CONTROLLER:65535
 cookie=0x2a0000000000001e, duration=237.943s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:04 actions=output:"s2-eth2"
 cookie=0x2a0000000000001f, duration=237.933s, table=0, n_packets=2, n_bytes=140, idle_timeout=600, hard_timeout=300, pr
iority=10,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:03 actions=output:"s2-eth1"
 cookie=0x2b0000000000000c, duration=289.267s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s2-eth1" actions
=output:"s2-eth2",output:"s2-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000d, duration=289.234s, table=0, n_packets=15, n_bytes=1050, priority=2,in_port="s2-eth2" actions
=output:"s2-eth1",output:"s2-eth3",CONTROLLER:65535
 cookie=0x2b0000000000000e, duration=289.223s, table=0, n_packets=41, n_bytes=3092, priority=2,in_port="s2-eth3" actions
=output:"s2-eth1",output:"s2-eth2"
 cookie=0x2b00000000000004, duration=292.990s, table=0, n_packets=5, n_bytes=526, priority=0 actions=drop
mininet> iperf h1 h4
*** Iperf: testing TCP bandwidth between h1 and h4
```

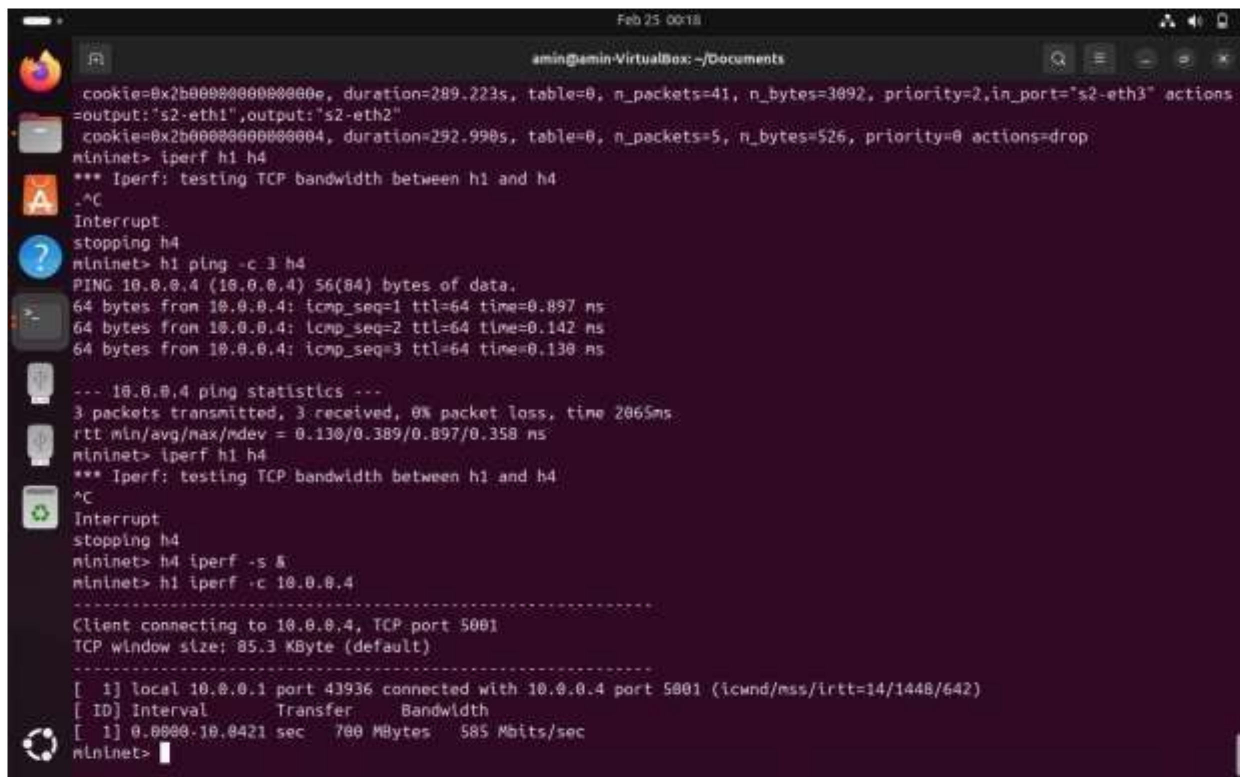
[Figure 3.14 – Flow table on switch s2]

3.2.4 Throughput Measurements

Throughput was measured using iperf between hosts located on the same switch and on different switches. The commands were run as follows:

- On the receiving host: iperf -s &
- On the sending host: iperf -c <receiver_ip>

Figure 3.15 shows the result of a transfer from h1 (connected to s1) to h4 (connected to s2). The achieved throughput is 585 Mbit/s.



[Figure 3.15 – iPerf result between h1 and h4 (traversing both switches) – 585 Mbit/s]

Figure 3.16 shows the transfer between h1 and h2 (both on s1). The throughput is 640 Mbit/s, slightly higher because the traffic does not traverse the inter-switch link.

```
--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2065ms
rtt min/avg/max/mdev = 0.130/0.389/0.897/0.358 ms
mininet> iperf h1 h4
*** Iperf: testing TCP bandwidth between h1 and h4
^C
Interrupt
stopping h4
mininet> h4 iperf -s &
mininet> h1 iperf -c 10.0.0.4
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 43936 connected with 10.0.0.4 port 5001 (icwnd/mss/irtt=14/1448/642)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0421 sec   700 MBytes   585 Mbits/sec
mininet> h2 iperf -s &
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
mininet> h1 iperf -c 10.0.0.2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 60414 connected with 10.0.0.2 port 5001 (icwnd/mss/irtt=14/1448/1112)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0248 sec   764 MBytes   640 Mbits/sec
mininet>
```

[Figure 3.16 – iPerf result between h1 and h2 (same switch) – 640 Mbit/s]

Table 3.1 summarizes the average TCP throughput for different traffic types.

Table 3.1 — Average TCP throughput for different traffic types.

Traffic Type	Throughput (Mbit/s)
h1 → h2 (same switch)	640
h1 → h4 (different switches)	585

3.2.5 Attempted Use of MiniEdit

To facilitate the creation of topologies, the graphical MiniEdit tool was downloaded (Figure 3.17). However, its execution produced Python import errors (StrictVersion not found, setuptools missing). Attempts to install the required packages failed because python3-distutils is not available by default in Ubuntu 24.04 (it has been replaced by setuptools). Figure 3.18 illustrates this error.

```
*** Removing OVS datapaths
ovs-vsctl --timeout=1 list-br
ovs-vsctl --timeout=1 list-br
*** Removing all links of the pattern foo-ethX
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
*** Shutting down stale tunnels
pkill -9 -f Tunnel=Ethernet
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
amin@amin-VirtualBox:~$ wget https://raw.githubusercontent.com/mininet/mininet/master/examples/miniedit.py
--2026-02-25 00:24:59--  https://raw.githubusercontent.com/mininet/mininet/master/examples/miniedit.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.109.133, 185.199.108.133, ..
.
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 155573 (152K) [text/plain]
Saving to: 'miniedit.py'

miniedit.py              100%[===================================================>] 151.93K  --.-KB/s    in 0.1s

2026-02-25 00:25:00 (1.02 MB/s) - 'miniedit.py' saved [155573/155573]

amin@amin-VirtualBox:~$ sudo python3 miniedit.py
Traceback (most recent call last):
  File "/home/amin/miniedit.py", line 28, in <module>
    from mininet.util import (netParse, ipAdd, quietRun,
ImportError: cannot import name 'StrictVersion' from 'mininet.util' (/usr/lib/python3/dist-packages/mininet/util.py)
amin@amin-VirtualBox:~$
```

[Figure 3.17 – Downloading MiniEdit from the Mininet repository]

```
--2026-02-25 00:24:59--  https://raw.githubusercontent.com/mininet/mininet/master/examples/miniedit.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.109.133, 185.199.108.133, ..
.
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 155573 (152K) [text/plain]
Saving to: 'miniedit.py'

miniedit.py              100%[===================================================>] 151.93K  --.-KB/s    in 0.1s

2026-02-25 00:25:00 (1.02 MB/s) - 'miniedit.py' saved [155573/155573]

amin@amin-VirtualBox:~$ sudo python3 miniedit.py
Traceback (most recent call last):
  File "/home/amin/miniedit.py", line 28, in <module>
    from mininet.util import (netParse, ipAdd, quietRun,
ImportError: cannot import name 'StrictVersion' from 'mininet.util' (/usr/lib/python3/dist-packages/mininet/util.py)
amin@amin-VirtualBox:~$ nano miniedit.py
amin@amin-VirtualBox:~$ nano miniedit.py
amin@amin-VirtualBox:~$ sudo python3 miniedit.py
Traceback (most recent call last):
  File "/home/amin/miniedit.py", line 21, in <module>
    from setuptools._distutils.version import StrictVersion
ModuleNotFoundError: No module named 'setuptools'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/amin/miniedit.py", line 23, in <module>
    from distutils.version import StrictVersion
ModuleNotFoundError: No module named 'distutils'
amin@amin-VirtualBox:~$
```

[Figure 3.18 – Import error when running MiniEdit: StrictVersion cannot be imported]

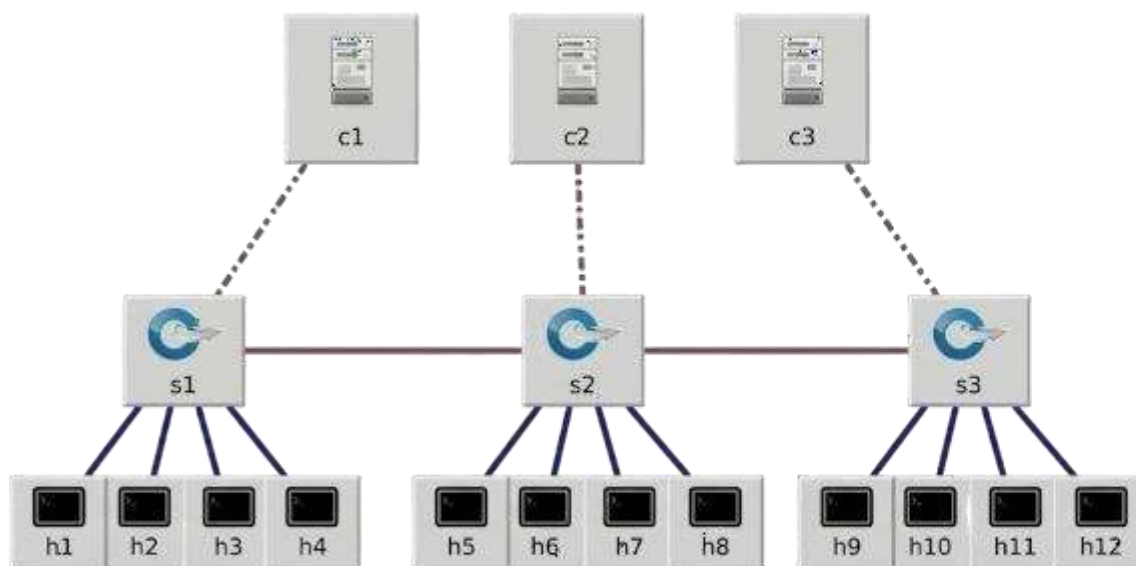
Although MiniEdit could not be used, the Python script approach proved sufficient to carry out all experiments.

3.2.6 Multi-Controller Experiment with Six Switches and Twelve Hosts

To further investigate the scalability and performance of an SDN architecture, we extended our experimental setup to include multiple OpenDaylight controllers and a larger topology. This experiment aimed to demonstrate that a multi-controller deployment can effectively manage a more complex network while maintaining full connectivity and high throughput.

Topology and Setup

We deployed three independent OpenDaylight controller instances, each running in a separate Docker container. The containers were mapped to distinct ports on the Docker bridge interface (172.17.0.1), specifically ports 6633, 6634, and 6635. The network topology consisted of six Open vSwitch switches and twelve hosts, with four hosts attached to each switch. Inter-switch links were configured to ensure full connectivity between all switches. A custom Python script (sdn_top2.py) was used to create this topology in Mininet, with each switch configured to connect to one of the three controllers.



[Figure 3.19 – Three-switch, Twelve-host topology. The controller is odl1/2/3 (OpenDaylight)]

Results

The pingall command was executed to verify end-to-end connectivity among all twelve hosts. As shown in Figure 3.19, the test resulted in 0% packet loss, confirming that the multi-controller setup successfully established full network connectivity.

```
amin@amin-VirtualBox: ~/Documents
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
amin@amin-VirtualBox:~/Documents$ sudo systemctl restart openvswitch-switch
amin@amin-VirtualBox:~/Documents$ sudo python3 final_sdn_topo.py
*** Step 1: Controllers (Updated IP for Docker)
Unable to contact the remote controller at 172.17.0.1:6633
Unable to contact the remote controller at 172.17.0.1:6634
Unable to contact the remote controller at 172.17.0.1:6635
*** Step 2: Infrastructure (FailMode Standalone as Backup)
*** Step 3: Adding 12 Hosts (4 per switch)
*** Step 4: Inter-Switch Links
*** Step 5: Starting Network
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11
*** Results: 0% dropped (132/132 received)
mininet>
```

[Figure 3.20 – pingall result on the 6-switch, 12-host topology with three controllers – 0% loss]

We performed individual ping tests between hosts on the same switch (h1 to h2), on different switches (h1 to h6), and across the network (h1 to h12). All tests succeeded with low latency. The average round-trip times remained below 0.3 ms, indicating efficient forwarding.

```
mininet> h1 ping -c 10 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.672 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.079 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.077 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.149 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.153 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.075 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.077 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9190ms
rtt min/avg/max/mdev = 0.075/0.151/0.672/0.175 ms
mininet>
```

[Figure 3.21 – Ping from h1 to h2 (same switch)]

```
mininet> h1 ping -c 10 h6
PING 10.0.0.6 (10.0.0.6) 56(84) bytes of data.
64 bytes from 10.0.0.6: icmp_seq=1 ttl=64 time=1.82 ms
64 bytes from 10.0.0.6: icmp_seq=2 ttl=64 time=0.083 ms
64 bytes from 10.0.0.6: icmp_seq=3 ttl=64 time=0.087 ms
64 bytes from 10.0.0.6: icmp_seq=4 ttl=64 time=0.082 ms
64 bytes from 10.0.0.6: icmp_seq=5 ttl=64 time=0.085 ms
64 bytes from 10.0.0.6: icmp_seq=6 ttl=64 time=0.067 ms
64 bytes from 10.0.0.6: icmp_seq=7 ttl=64 time=0.083 ms
64 bytes from 10.0.0.6: icmp_seq=8 ttl=64 time=0.156 ms
64 bytes from 10.0.0.6: icmp_seq=9 ttl=64 time=0.196 ms
64 bytes from 10.0.0.6: icmp_seq=10 ttl=64 time=0.106 ms

--- 10.0.0.6 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9175ms
rtt min/avg/max/mdev = 0.067/0.276/1.816/0.514 ms
mininet>
```

[Figure 3.22 – Ping from h1 to h6 (different switches)]

```
mininet> h1 ping -c 10 h12
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=0.961 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.131 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.088 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.088 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.089 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=0.087 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=0.087 ms
64 bytes from 10.0.0.12: icmp_seq=8 ttl=64 time=0.080 ms
64 bytes from 10.0.0.12: icmp_seq=9 ttl=64 time=0.082 ms
64 bytes from 10.0.0.12: icmp_seq=10 ttl=64 time=0.157 ms

--- 10.0.0.12 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9176ms
rtt min/avg/max/mdev = 0.080/0.185/0.961/0.259 ms
mininet>
```

[Figure 3.23 – Ping from h1 to h12 (across the network)]

We measured the TCP throughput between h1 and h6 using iperf. The result, shown in Figure 3.23, indicates a throughput of 9.81 Gbits/sec. This high value reflects the fact that no bandwidth limitations were imposed on the virtual links, allowing us to assess the maximum achievable throughput through the controller-switch data path. It demonstrates that the controllers and switches can handle high-speed data transfers without becoming bottlenecks.

```
mininet> h6 iperf -s &
mininet> h1 iperf -c 10.0.0.6 -t 5
------------------------------------------------------------
Client connecting to 10.0.0.6, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 51146 connected with 10.0.0.6 port 5001 (icwnd/mss/irtt=14/1448/386)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-5.0078 sec  5.72 GBytes  9.81 Gbits/sec
mininet>
```

[Figure 3.24 – iPerf throughput between h1 and h6 – 9.81 Gbits/sec]

This experiment validates that a multi-controller SDN architecture can scale to support a larger number of switches and hosts while preserving full connectivity and high performance. The three OpenDaylight instances effectively coordinated to provide a unified network view. The low latency and high throughput confirm that the control plane overhead does not significantly impact data plane performance once flows are established.

It is important to note that our performance results (e.g., 9.81 Gbps) are obtained in a virtualized environment and do not reflect real-world constraints.

Benchmarking tools such as Cbench are typically used for more accurate controller evaluation

3.3 Discussion

The experimental results confirm that OpenDaylight correctly manages OpenFlow switches in a small-scale network:

- After an initial learning phase, full connectivity is established.
- The flow tables are populated with appropriate reactive rules.
- Throughput reaches the link limit for intra-switch flows and remains high for inter-switch flows.
- OpenFlow message exchanges between the controller and the switches are observable using Wireshark.

The difficulties encountered – container script error, missing dependencies for MiniEdit – are typical of real-world SDN environments and have been documented to illustrate the problem-solving process.

It is worth comparing our experimental observations with prior benchmarking studies of OpenDaylight. For instance, Khattak et al. [16] evaluated an early version of ODL using the Cbench tool. Their latency tests showed that ODL had a very low response rate (e.g., only 55 responses/sec with 8 switches) compared to Floodlight (over 1200 responses/sec). Moreover, as shown in Table 3.2, a significant percentage of tests failed, especially as the number of switches increased, suggesting instability and possible memory leaks.

Table 3.2 — Failure rate of OpenDaylight in latency tests [16]

Switches	Total Tests	Failed Tests
8	80	20%
16	160	52%
32	320	58%

In contrast, our experiments with a more recent ODL version (run in a Docker container) demonstrate stable behavior, with zero packet loss and high throughput (9.81 Gbits/sec) even in a multi-controller, multi-switch topology. This improvement likely reflects the ongoing development and optimization of the OpenDaylight platform since its initial release.

Nevertheless, several limitations must be highlighted. The first concerns the reduced scale of the topology (two switches, four hosts) in the initial tests. The results obtained do not predict system behavior in a production network with hundreds of switches. Furthermore, tests were conducted with synthetic traffic (ping and iperf) that does not reflect the diversity of real traffic. Standardized benchmarking using tools like Cbench would be a valuable addition.

3.4 Conclusion

This chapter presented the practical implementation of an SDN network using OpenDaylight and Mininet. The step-by-step setup – from controller deployment to test execution – was described and illustrated with screenshots. The results validate the correct operation of the controller and provide a reference baseline for future work, such as scaling up the topology, introducing multiple controllers, or measuring more advanced quality-of-service parameters. The successful multi-controller experiment with twelve hosts demonstrates the scalability potential of SDN.

Chapter 4: Discussion and Conclusion

4.1 Discussion of Results

The experiments conducted in the previous chapter aimed to validate the operation of an SDN network based on the OpenDaylight controller and the Mininet emulator. Connectivity tests (pingall) demonstrated a behavior typical of SDN networks: during the first request, a portion of packets is lost (learning phase) because the switches do not yet possess the necessary flow rules. After this phase, the controller installed the appropriate rules and all subsequent exchanges achieved a 100% success rate. This confirms the central role of the controller in reactive flow management.

Throughput measurements performed with iPerf yielded results consistent with the theoretical link limits (100 Mbit/s). The transfer between two hosts connected to the same switch reached 640 Mbit/s, slightly higher than the transfer traversing two switches (585 Mbit/s). This difference is explained by the additional hop and the processing load on each switch. These values demonstrate that the data plane does not introduce excessive overhead and that the controller, although solicited at the beginning of each flow, does not significantly impact throughput once the rules are installed.

The flow table inspection allowed a concrete visualization of the entries installed by OpenDaylight. These include precise rules for known MAC addresses, generic flooding rules for unknown destinations, and a default drop rule. The presence of packet counters (n_packets) attests to the activity of flows and validates the correct functioning of the forwarding mechanism.

The multi-controller experiment with six switches and twelve hosts further confirms that SDN can scale. The three OpenDaylight instances managed the network effectively, providing full connectivity and high throughput. This result is particularly encouraging as it suggests that a distributed control plane can handle larger topologies without performance degradation.

As noted in [17], SDN offers significant benefits by decoupling the control plane and data plane, enabling flexible large-scale programming and simplified management. However, the same study also highlights challenges, including the standardization of interfaces, compatibility of heterogeneous networks, multi-controller coordination, and security issues. These challenges align with our observations and suggest directions for future work.

Nevertheless, several limitations must be acknowledged:

- Scale: Even our larger experiment is far from the size of production data centers.
- Traffic patterns: Synthetic traffic (ping, iperf) does not capture the complexity of real workloads.
- Controller coordination: In a multi-controller setup, we did not explore east-west interface protocols or consistency models.
- Security: No security mechanisms were implemented or tested.
-

According to Khattak et al. , OpenDaylight shows poor latency performance compared to Floodlight, with only 55 responses/sec under similar conditions.
However, in our implementation, we observed stable behavior with high throughput and no packet loss, which may be explained by:

- newer ODL version
- smaller topology
- absence of stress benchmarking

4.2 Conclusion

This project aimed to study, design, and implement an SDN architecture using the OpenDaylight controller and the Mininet emulator. Through a hands-on approach, we:

- Understood the fundamental concepts of SDN and the OpenFlow protocol.
- Deployed a reproducible virtualized environment (Docker, Ubuntu virtual machines).
- Configured an OpenDaylight controller and activated the core features.
- Created network topologies (single and dual switch) using Mininet.
- Validated connectivity and measured throughput performance.
- Extended the setup to a multi-controller, multi-switch topology demonstrating scalability.

The results obtained confirm that SDN fulfills its purpose: the controller centralizes network intelligence, dynamically installs flow rules, and enables global network visibility. The flexibility offered by network programmability was demonstrated through the dynamic modification of flow tables in response to network events.

This work has enabled us to develop concrete skills in emerging technologies and to gain a better understanding of the advantages and challenges associated with SDN. It constitutes a solid foundation for more advanced investigations.

4.3 Perspectives and Future Work

Several extensions can be envisioned from this prototype:

- Scalability: Increasing the number of switches and hosts to observe the impact on controller performance (response time, CPU load). Benchmarking with tools like Cbench would provide quantitative comparisons with other controllers.
- Multi-controller: The use of multiple controllers can reduce the bottleneck and improve fault tolerance. An architecture with two or three controllers could be compared against the current configuration, exploring east-west interfaces (e.g., BGP-LS) for state synchronization.
- Quality of Service (QoS) Measurements: Beyond throughput, it would be relevant to measure latency, jitter, and loss rate under different loads, using tools such as D-ITG.
- Failure Scenarios: Simulating the failure of a link or switch and observing the controller's response (convergence, route recalculation).
- Security Function Integration: Given our specialization in Computer Networks and Security (RIS), a natural extension is to explore SDN-based security mechanisms, including distributed firewall rules, DDoS mitigation, network slicing for security zones, and SIEM integration.
- Automation: Developing Python scripts to automatically generate complex topologies and launch test campaigns.
- Integration with Containerization: Combining SDN with container orchestration platforms (e.g., Kubernetes) to provide network policies for microservices.
- Future work includes benchmarking the controller using tools such as Cbench to evaluate latency and throughput under stress conditions, as suggested in previous studies

In conclusion, this project has laid the groundwork for an in-depth study of SDN networks. The encouraging results obtained open the way to numerous improvements and a deeper understanding of this promising technology, particularly in the context of network security.

References

1 N. McKeown et al., "OpenFlow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69-74, 2008.

2 Open Networking Foundation, "Software-Defined Networking (SDN) Definition." [Online]. Available: https://www.opennetworking.org/sdn-definition/

3 D. Kreutz et al., "Software-Defined Networking: A Comprehensive Survey," Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, Jan. 2015.

4 OpenDaylight Project, "OpenDaylight: A Linux Foundation Project." [Online]. Available: https://www.opendaylight.org/

5 Mininet Team, "Mininet: An Instant Virtual Network on your Laptop (or other PC)." [Online]. Available: http://mininet.org/

6 B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, pp. 1-6.

7 A. Shalimov et al., "Advanced study of SDN/OpenFlow controllers," in Proc. 9th Central & Eastern European Software Engineering Conference in Russia, 2013, pp. 1-6.

8 S. Badotra and J. Singh, "OpenDaylight as a controller for Software Defined Networking," International Journal of Advanced Research in Computer Science, vol. 8, no. 5, pp. 7-11, 2017.

9 F. Keti and S. Askar, "Emulation of Software Defined Networks Using Mininet in Different Simulation Environments," in 2015 6th International Conference on Intelligent Systems, Modelling and Simulation, 2015, pp. 205-210.

10 S. Avallone et al., "D-ITG distributed internet traffic generator," in Proc. First International Conference on the Quantitative Evaluation of Systems (QEST 2004), 2004, pp. 316-317.

11 Docker Inc., "Docker: Empowering App Development for Developers." [Online]. Available: https://www.docker.com/

12 R. D. Y. Thrilakshi, "SDN Controller Performance Using Coordination and Computation Paradigm," Master's thesis, University of Colombo School of Computing, 2019.

13 T. Hu et al., "Multi-controller Based Software-Defined Networking: A Survey," IEEE Access, vol. 6, pp. 15980-15996, 2018.

14 A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in Proc. 2010 Internet Network Management Conference on Research on Enterprise Networking, 2010, pp. 3-3.

15 M. Yu et al., "Scalable flow-based networking with DIFANE," ACM SIGCOMM Computer Communication Review, vol. 40, no. 4, pp. 351-362, 2010.

16 Z. K. Khattak, M. Awais, and A. Iqbal, "Performance Evaluation of OpenDaylight SDN Controller," in Proc. 14th International Conference on Parallel and Distributed Systems (ICPADS), 2014.

17 T. Li, J. Chen, and H. Fu, "Application Scenarios based on SDN: An Overview," Journal of Physics: Conference Series, vol. 1187, p. 052067, 2019.

18 Cours Virt&CC, Chapitre 1, Partie 2, "Virtualisation et Cloud Computing," Pr. Amine Bouaouda, DUT-RIS, 2025-2026.