

Master Thesis

**SDN based Network Management in Emulated environment**

|  |  |
| --- | --- |
| Submitted by: | Harshal Rajan Vaze |
| Matriculation no.: | 1269879 |
| First examiner: | Prof. Dr. Ulrich Trick |
| Second examiner: | Prof. Dr. Armin Lehmann |
| External supervisor: | Dr. Peter Gröschke |
| Date of start: | 01.05.2022 |
| Date of submission: |  |

Statement

I confirm that I have written this thesis on my own. No other sources were used except those referenced. Content which is taken literally or analogously from published or unpublished sources is identified as such. The drawings or figures of this work have been created by myself or are provided with an appropriate reference. This work has not been submitted in the same or similar form or to any other examination board.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date, signature of the student

Content

[1 Introduction 5](#_Toc113574905)

[1.1 Aim and Motivation 5](#_Toc113574906)

[1.2 Problem Statement 6](#_Toc113574907)

[1.3 Thesis Structure 6](#_Toc113574908)

[2 Theoretical Background 7](#_Toc113574909)

[2.1 Software defined Networking 7](#_Toc113574910)

[2.2 SDN Controllers 7](#_Toc113574911)

[2.2.1 ONOS 7](#_Toc113574912)

[2.2.2 ODL 7](#_Toc113574913)

[2.2.3 Ryu 7](#_Toc113574914)

[2.3 Software Switches 8](#_Toc113574915)

[2.4 Virtual Emulated Environment 8](#_Toc113574916)

[2.4.1 Mininet 8](#_Toc113574917)

[2.4.2 GNS3 8](#_Toc113574918)

[2.5 OpenFlow Protocol 9](#_Toc113574919)

[3 Requirements Analysis 10](#_Toc113574920)

[3.1 General Objectives 10](#_Toc113574921)

[3.2 Work Plan 10](#_Toc113574922)

[3.3 Previous Work 11](#_Toc113574923)

[4 Realization 12](#_Toc113574924)

[4.1 Installation 12](#_Toc113574925)

[4.2 Implementation with GNS3 12](#_Toc113574926)

[4.3 Implementation with Mininet 16](#_Toc113574927)

[4.4 Problems identified 16](#_Toc113574928)

[4.5 Use Cases 16](#_Toc113574929)

[4.6 Use case-1: Basic Network Architecture 16](#_Toc113574930)

[4.6.1 Introduction 16](#_Toc113574931)

[4.6.2 Flows 17](#_Toc113574932)

[4.6.3 Intents 18](#_Toc113574933)

[4.7 Use case-2: Testing the network with multiple Controllers 18](#_Toc113574934)

[4.7.1 Introduction 18](#_Toc113574935)

[4.7.2 Different Controllers with their Devices 19](#_Toc113574936)

[4.7.3 Path Identifier 19](#_Toc113574937)

[4.8 Use case-3: Testing network with isolated L2 overlay networks 20](#_Toc113574938)

[4.8.1 Introduction 20](#_Toc113574939)

[4.8.2 Configuration 20](#_Toc113574940)

[4.9 Use case-4: Testing network with IPv6 addressing 20](#_Toc113574941)

[4.9.1 Introduction 20](#_Toc113574942)

[4.9.2 IPv6 tunnelling over IPv4 21](#_Toc113574943)

[4.10 Use case-5: Integrating SDN with legacy network 22](#_Toc113574944)

[4.10.1 Introduction 22](#_Toc113574945)

[4.10.2 Configuration 23](#_Toc113574946)

[5 Summary and Perspectives 24](#_Toc113574947)

[6 Abbreviations 25](#_Toc113574948)

[7 References 27](#_Toc113574949)

[8 Appendix 28](#_Toc113574950)

# Introduction

In this era of network virtualization and automation, many functions are moving towards virtualized and more centralized control, allowing for more dynamic functions and easier optimization. For networking it would mean spinning up the virtualized versions of traditional network functions allowing for more complex decisions making, automating the networks, and changing network configuration more efficiently.

Software-Defined Networking (SDN) is an approach to networking that uses software-based controllers and application programming interfaces (APIs) to communicate with underlying hardware infrastructure and direct traffic on a network. This method is different from that of traditional networks, where the configuration of dedicated hardware devices like Routers and Switches needs to be done node by node to control network traffic. SDN can create and control a virtual network or also control a traditional hardware via software. Because the control plane is software-based, SDN is much more flexible than traditional networking. It allows administrators to control and manage the network from a centralized user interface, without adding more hardware.

A SDN controller is an application in a software-defined networking architecture that manages flow control for improved network management and overall network performance. The SDN controller platform typically runs on a server and uses protocols to tell switches how to forward the packets. SDN controllers direct traffic according to forwarding policies that a network operator puts in place, thereby minimizing manual configurations for individual network devices. By taking the control plane off of the network hardware and running it instead as software, the centralized controller facilitates automated network management and makes it easier to integrate and administer various applications. In effect, the SDN controller serves as a sort of operating system (OS) for the network.

The SDN controller is the core of a software-defined network. It resides between network devices at one end of the network and applications at the other end. Any communication between applications and network devices must go through the controller. The SDN controller communicates with applications via northbound interfaces. The Open Networking Foundation (ONF) created a working group in 2013 focused specifically on northbound APIs and their development. The industry never settled on a standardized set, however, largely because application requirements vary so widely. Typically northbound RESTful APIs secured with TLS are used to push the configuration changes from the application to the SDN controller. The SDN controller talks with individual network devices using a southbound interface with the help of well-known OpenFlow protocol and other protocols such as NetConf, BGP, SNMP, etc. These southbound protocols allow the controller to configure network devices and choose the optimal network path for application traffic. OpenFlow was created by ONF in 2011 and latest version of OpenFlow protocol is OpenFlow 1.5.1.

## Aim and Motivation

There are various open-source SDN controllers developed by various groups of developers and are ready to be deployed and tested in the production environment. SDN controllers are primarily made to provide some policies and centralized managements for the networking layer, which will further allow inter-working configurations between the interfaces. Iptables, network namespaces, and Open vSwitch are components of the Linux kernel's L3 IP routing, Linux bridges, and SDN controller technologies. Being open-source controllers, not all of the available SDN controllers are developed on the same scale of mechanism in terms of supporting these various protocols, especially new versions of protocols and many of them are not developed yet to completely support all the services required for direct integration with the real world network. Therefore, the aim of this Master Thesis is to research and study different open-source SDN controllers in terms of their functionality with services and protocols.

After research, the aim would to build and setup the virtual network with network elements in the emulation software for testing the functionality of SDN controller. Further, to develop some use cases for demonstrating the services and protocols used in the network.

## Problem Statement

Due to advances in the Information and Communication Technology, the configuration and management of the network components becomes highly complex and time-consuming. A fundamental characteristic of SDN is the logically centralized, but physically distributed controller component. SDN offers to batch-configure automatically multiple components in one step, while the traditional way would mean to log into each device. Many operators struggle with the migration from IPv4 to IPv6, SDN with its centralized control and the possibility to reduce human error due to increased poses an opportunity to help make this migration easier. The controller maintains a global network view of the underlying forwarding infrastructure and programs the forwarding entries based on the policies defined by network services running on top of it. The traditional networking approach has very limited facilities to explore these aspects of networking and the goal would be to study these futuristic characteristics of networking.

## Thesis Structure

This thesis work is structured in five main chapters: chapter 1 gives the introduction to the topic and discusses the statement of the problem. Chapter 2 reviews the theoretical background knowledge of the topics associated with this thesis. Furthermore it also describes the in-depth information about the components used in this Thesis. In chapter 3, requirement analysis, the main objectives of the thesis are discussed focusing on question ‘which open-source SDN controllers are available to be tested in emulated environment?’. The realization of the implementation in different environments along with some use cases are presented and discussed in the chapter 4 named, realization. This chapter focuses on the question ‘how to implement suitable environment for testing the use cases?’. Finally, at end a summary, perspectives and future work are provided in chapter 5.

# Theoretical Background

## Software defined Networking

Dr. Martin Casado developed an architecture to separate control and forwarding functions of networking de-vices, migrating control to a centralized policy server. This architecture evolved to what is now known as Software Defined Networking (SDN) today. One of the first challenges was the need for a common South Bound Interface (SBI) protocol between the SDN Controller and the forwarding networking device. OpenFlow developed by the Open Networking Foundation (ONF) and is used over a secure channel (Transport Layer Security (TLS) over Transmission Control Protocol (TCP) port 6633 and 6653) to modify the group and flow tables in a OpenFlow networking device. OpenFlow has evolved to version 1.5.1.

Graphical user interface, application

Description automatically generated

Over the last few years, the need for a new approach to networking has been expressed to overcome the many issues associated with current networks. In particular, the main vision of the SDN approach is to simplify networking operations, optimize network management and introduce innovation and flexibility as compared to legacy networking architectures.

Network management is becoming more difficult and challenging given that the static and inflexible architecture of legacy networks is ill-suited to cope with today’s increasingly dynamic networking trends, and to meet the QoE requirements of modern users. This fact has fuelled the need for the enforcement of complex and high-level policies to adapt to current networking environments, and for the automation of network operations to reduce the tedious workload of low-level device configuration tasks.

The SDN initiative led by the Open Networking Foundation (ONF), on the other hand, proposes a new open architecture to address current networking challenges with the potential to facilitate the automation of network configurations , and better yet, fully program the network. Unlike the conventional distributed network architecture where network devices are closed and vertically-integrated bundling software with hardware, the SDN architecture raises the level of abstraction by separating the network data and control planes. That way, network devices become simple forwarding switches whereas all the control logic is centralized in software controllers providing a flexible programming framework for the development of specialized applications and for the deployment of new services. Such aspects of SDN are believed to simplify and improve network management by offering the possibility to innovate, customize behaviours and control the network according to high-level policies expressed as centralized programs, therefore bypassing the complexity of low-level network details and overcoming the fundamental architectural problems like high complexity and configuration. Added to these features is the ability of SDN to easily cope with the heterogeneity of the underlying infrastructure thanks to the SDN Southbound interface abstraction.

Data plane:

The data plane, also known as the forwarding plane, consists of a distributed set of forwarding network elements (mainly switches) in charge of forwarding packets. In the context of SDN, the control-to-data plane separation feature requires the data plane to be remotely accessible for software-based control via an open vendor-agnostic Southbound interface.

Both OpenFlow [24] and ForCES [25] are well-known candidate protocols for the Southbound interface. They both follow the basic principle of splitting the control plane and the forwarding plane in network elements and they both standardize the communication between the two planes. However, these solutions are different in many aspects, especially in terms of network architecture design. Standardized by IETF, ForCES (Forwarding and Control Element Separation) [25] introduced the separation between the control plane and the forwarding plane. In doing so, ForCES defines two logic entities that are logically kept in the same physical device: the Control Element (CE) and the Forwarding Element (FE). However, despite being a mature standard solution, the ForCES alternative did not gain widespread adoption by major router vendors. On the other hand, OpenFlow [24] received major attention in both the research community and the industry. Standardized by the ONF [23], it is considered as the first widely accepted communication protocol for the SDN Southbound interface. OpenFlow enables the control plane to specify in a centralized way the desired forwarding behaviour of the data plane. Such traffic forwarding decisions reflect the specified network control policies and are translated by controllers into actual packet forwarding rules populated in the flow tables of OpenFlow switches.

Control plane:

Regarded as the most fundamental building entity in SDN architecture, the control plane consists of a centralized software controller that is responsible for handling communications between network applications and devices through open interfaces. More specifically, SDN controllers translate the requirements of the application layer down to the underlying data plane elements and give relevant information up to SDN applications.

The SDN control layer is commonly referred to as the Network Operating System (NOS) as it supports the network control logic and provides the application layer with an abstracted view of the global network, which contains enough information to specify policies while hiding all implementation details. Typically, the control plane is logically centralized and yet implemented as a physically distributed system for scalability and reliability reasons as discussed in Sections III and IV. In a distributed SDN control configuration, East-Westbound APIs[40] are required to enable multiple SDN controllers to communicate with each other and exchange network information.

Despite the many attempts to standardize SDN protocols, there has been to date no standard for the East-West API which remains proprietary for each controller vendor. Although a number of East-Westbound communications happen only at the data-store level and do not require additional protocol specifics, it is becoming increasingly advisable to standardize that communication interface in order to provide wider interoperability between different controller technologies in different autonomous SDN networks.

On the other hand, an East-Westbound API standard requires advanced data distribution mechanisms and involves other special considerations. This brings about additional SDN challenges, some of which have been raised by the state-of the art distributed controller platforms discussed in Sections III and IV, but have yet to be fully addressed.

Application Plane:

The SDN application plane comprises SDN applications which are control programs designed to implement the network control logic and strategies. This higher-level plane interacts with the control plane via an open Northbound API. In doing so, SDN applications communicate their network requirements to the SDN controller which translates them into Southbound-specific commands and forwarding rules dictating the behaviour of the individual data plane devices. Routing, Traffic Engineering (TE), firewalls and load balancing are typical examples of common SDN applications running on top of existing controller platforms. In the context of SDN, applications leverage the decoupling of the application logic from the network hardware along with the logical centralization of the network control, to directly express the desired goals and policies in a centralized high-level manner without being tied to the implementation and state distribution details of the underlying networking infrastructure.

Concurrently, SDN applications make use of the abstracted network view exposed through the Northbound interface to consume the network services and functions provided by the control plane according to their specific purposes. That being said, the Northbound API implemented by SDN controllers can be regarded as a network abstraction interface to applications, easing network programmability, simplifying control and management tasks and allowing for innovation. In contrast to the Southbound API, the Northbound API is not supported by an accepted standard. Despite the broad variety of Northbound APIs adopted by the SDN community (see Figure 2), we can classify them into two main categories:

* The first set involves simple and primitive APIs that are directly linked to the internal services of the controller platform. These implementations include:
  + - Low-level ad-hoc APIs that are proprietary and tightly dependent on the controller platform. Such APIs are not considered as high-level abstractions as they allow developers to directly implement applications within the controller in a low-level manner. Deployed internally, these applications are tightly coupled with the controller and written in its native general-purpose language. NOX in C++ and POX in Python are typical examples of controllers that use their own basic sets of APIs.
    - APIs based on Web services such as the widely-used REST API. This group of programming interfaces enables independent external applications (Clients) to access the functions and services of the SDN controller (Server). These applications can be written in any programming language and are not run inside the bundle hosting the controller software. Floodlight is an example of an SDN controller that adopts an embedded Northbound API based on REST.
* The second category contains higher level APIs that rely on domain-specific programming languages such as Frenetic [41], Procera [42] and Pyretic [43] as an indirect way for applications to interact with the controller. These APIs are designed to raise the level of abstraction in order to allow for the flexible development of applications and for the specification of high-level network policies.

## SDN Controllers

Distributed SDN Control: Survey, Taxonomy and Challenges [1]

Table

Description automatically generated

Despite the undeniable strengths of SDN, there have always been serious concerns about the ability to extend SDN to largescale networks. Some argue that these scalability limits are basically linked to the protocol standards being used for the implementation of SDN. OpenFlow [24] in particular, although recognized as a leading and widely-deployed SDN Southbound technology, is currently being rethought for potentially causing excessive overheads on switches (switch bottleneck). Scalable alternatives to the OpenFlow standard which propose to revisit the delegation of control between the controller and the switches with the aim of reducing the reliance on SDN the control plane, have been discussed in II-A. Another entirely different approach to addressing the SDN scalability and reliability challenges, which is advocated by the present paper, is to physically distribute the SDN control plane. This has led to a first categorization of existing controller platforms into centralized and distributed architectures.

Centralized SDN control

A physically-centralized control plane consisting of a single controller for the entire network is a theoretically perfect design choice in terms of simplicity. However, a single controller system may not keep up with the growth of the network. It is likely to become overwhelmed (controller bottleneck) while dealing with an increasing number of requests and concurrently struggling to achieve the same performance guarantees. Obviously, a centralized SDN controller does not meet the different requirements of large-scale real-world network deployments. Data Centers and Service Provider Networks are typical examples of such large-scale networks presenting different requirements in terms of scalability and reliability. More specifically, a Data Center Network involves tens of thousands of switching elements. Such a great number of forwarding elements which can grow at a fast pace is expected to generate a huge number of control events that are enough to overload a single centralized SDN controller [44, 45]. Studies conducted in [46] show important scalability implications (in terms of throughput) for centralized controller approaches. They demonstrate that multiple controllers should be used to scale the throughput of a centralized controller and meet the traffic characteristics within realistic data centers. Unlike data centers, Service Provider Networks are characterized by a modest number of network nodes. However, these nodes are usually geographically distributed making the diameter of these networks very large [44]. This entails a different type of controller scalability issues for centralized controller approaches, more specifically, high latencies. In addition to latency requirements, service provider networks have large numbers of flows that may generate overhead and bandwidth issues. In general, Wide Area Network (WAN) deployments typically impose strict resiliency requirements. In addition, they present higher propagation delays as compared to data center networks. Obviously, a centralized controller design in a SDWAN cannot achieve the desired failure resiliency and scaleout behaviors [47]. Several studies have emphasized the need for a distributed control plane in a SD-WAN architecture: They indeed focused on placing multiple controllers on real WAN topologies to benefit both control plane latency and faulttolerance [48, 49]. That said, the potential scalability, reliability and vulnerability concerns associated with centralized controller approaches have been further confirmed through studies [7, 50] on the behavior of state-of-the-art centralized SDN controllers such as NOX [51], Beacon [52] and Floodlight [53] in different networking environments.

In particular, NOX classic [51], the world’s first-generation OpenFlow controller with an event-based programming model, is believed to be limited in terms of throughput. Indeed, it cannot handle a large number of flows, namely a rate of 30k flow initiation events per second [7, 54]. Such a flow setup throughput may sound sufficient for an enterprise network, but, it could be arguable for data-center deployments with high flow initiation rates [46]. Improved versions of NOX have been consequently developed by the same community (Nicira Networks) such as NOX-MT [55] for better performance and POX [56] for a more developer-friendly environment. However, while none of these centralized designs is believed to meet the above scalability and reliability requirements of large-scale networks, they have gained greater prominence as they were widely used for research and educational purposes. Additionally, Floodlight [53] which is a very popular Javabased OpenFlow controller from Big Switch Networks, suffers from serious security and resiliency issues. For instance, Dhawan et al. [57] have reported that the centralized SDN controller is inherently susceptible to Denial-of-Service (DoS) attacks. Another subsequent version of Floodlight, called SEFloodlight, has therefore been released to overcome these problems by integrating security applications. However, despite the introduced security enhancements aimed at shielding the centralized controller, the latter remains a potential weakness compromising the whole network. In fact, the controller still maintains a single point of failure and bottlenecks even if its latest version is less vulnerable to malicious attacks. On the other hand, given its obvious performance and functionality advantages, the open-source Floodlight has been extensively used to build other SDN controller platforms supporting distributed architectures such as ONOS [58] and DISCO [59].

Distributed SDN control

Alternatively, physically-distributed control plane architectures have received increased research attention in recent years since they appeared as a potential solution to mitigate the issues brought about by centralized SDN architectures (poor scalability, Single Point of Failure (SPOF), performance bottlenecks, etc). As a result, various SDN control plane designs have been proposed in recent literature. Yet, we discern two main categories of distributed SDN control architectures based on the physical organization of SDN controllers: A flat SDN control architecture and a hierarchical SDN control architecture (see Figure 3).

1) Flat SDN control:

The flat structure implies the horizontal partitioning of the network into multiple areas, each of which is handled by a single controller in charge of managing a subset of SDN switches. There are several advantages to organizing controllers in such a flat style, including reduced control latency and improved resiliency. Onix [60], Hyperflow [61] and ONOS [58] are typical examples of flat physically-distributed controller platforms which are initially designed to improve control plane scalability through the use of multiple interconnected controllers sharing a global network-wide view and allowing for the development of centralized control applications. However, each of these contributions takes a different approach to distribute controller states and providing control plane scalability. For example, Onix provides a good scalability through additional partitioning and aggregation mechanisms. To be more specific, Onix partitions the NIB (Network Information Base) giving each controller instance responsibility for a subset of the NIB and it aggregates by making application reduce the fidelity of information before sharing it between other Onix instances within the cluster. Similar to Onix, each ONOS instance (composing the cluster) that is responsible for a subset of network devices holds a portion of the network view that is also represented in a graph. Different from Onix and ONOS, every controller in HyperFlow has the global network view, thus getting the illusion of control over the whole network. Yet, HyperFlow can be considered as a scalable option for specific policies in which a small number of network events affect the global network state. In that case, scalability is ensured by propagating these (less frequent) selected events through the event propagation system. Furthermore, different mechanisms are put in place by these distributed controller platforms to meet fault-tolerance and reliability requirements in the event of failures or attacks. Onix [60] uses different recovery mechanisms depending on the detected failures. Onix instance failure is most of the time handled by distributed coordination mechanisms among replicas whereas network element/link failures are under the full responsibility of applications developed atop Onix. Besides, Onix is assumed reliable when it comes to connectivity infrastructure failures as it can dedicate the failure recovery task to a separate management backbone that uses a multipathing protocol.

Likewise, Hyperflow [61] focuses on ensuring resiliency and fault tolerance as a means for achieving availability. When a controller failure is discovered by the failure detection mechanisms deployed by its publish/subscribe WheelFS [62] system, HyperFlow reconfigures the affected switches and redirects them to another nearby controller instance (from a neighbour’s site). Alongside this ability to tackle component failures, HyperFlow is resilient to network partitioning thanks to the partition tolerance property of WheelFS. In fact, in the presence of a network partitioning, WheelFS partitions continue to operate independently, thus favouring availability. Similarly, ONOS [58] considers fault-tolerance as a prerequisite for adopting SDN in Service Provider networks. ONOS’s distributed control plane guards against controller instance failures by connecting, from the onset, each SDN switch to more than one SDN controller; its master controller and other backup controllers (from other domains) that may take over in the wake of master controller failures. Load balancing mechanisms are also provided to balance the mastership of switches among the controllers of the cluster for scalability purposes. Besides, ONOS incorporates additional recovery protocols, such as the Anti-Entropy protocol [63], for healing from lost updates due to such controller crashes. Recent SDN controller platform solutions [64, 65, 66, 67, 68, 69] focused specifically on improving fault-tolerance in the distributed SDN control plane. Some of these works assumed a simplified flat design where the SDN control was centralized. However, since the main focus was placed at the fault-tolerance aspect, we believe that their ideas and their fault-tolerance approaches can be leveraged in the context of medium to large scale SDNs where the network control is physically distributed among multiple controllers.

In particular, Botelho et. al [70] developed a hybrid SDN controller architecture that combines both passive and active replication approaches for achieving control plane fault tolerance. SMaRtLight adopts a simple Floodlight [53]-based multi-controller design following OpenFlow 1.3, where one main controller (the primary) manages all network switches, and other controller replicas monitor the primary controller and serve as backups in case it fails. This variant of a traditional passive replication system relies on an external data store that is implemented using a modern active Replicated State Machine (RSM) built with a Paxoslike protocol (BFT-SMaRt [71]) to ensure fault-tolerance and strong consistency. This shared data store is used for storing the network and application state (the common global NIB) and also for coordinating fault detection and leader election operations between controller replicas that run a lease management algorithm.

In case of a failure of the primary controller, the elected backup controller starts reading the current state from the shared consistent data store in order to mitigate the cold-start (empty state) issue associated with traditional passive replication approaches, and thereby ensure a smoother transition to the new primary controller role. The limited feasibility of the deployed controller fault tolerance strategy is warranted by the limited scope of the SMaRtLight solution which is only intended for small to medium-sized SDN networks. On the other hand, in large-scale deployments, adopting a simplified Master-Slave approach, and more importantly, assuming a single main controller scheme where one controller replica must retrieve all the network state from the shared data store in failure scenarios, have major disadvantages in terms of increased latency and failover time.

Similarly, the Ravana controller platform proposal [66] addresses the issue of recovering from complete fail-stop controller crashes. It offers the abstraction of a fault-free centralized SDN controller to unmodified control applications which are relieved of the burden of handling controller failures. Accordingly, network programmers write application programs for a single main controller and the transparent masterslave Ravana protocol takes care of replicating, seamlessly and consistently, the control logic to other backup controllers for fault-tolerance. The Ravana approach deploys enhanced Replicated State Machine (RSM) techniques that are extended with switchside mechanisms to ensure that control messages are processed transactionally with ordered and exactly-once semantics even in the presence of failures. The three Ravana prototype components, namely the Ryu [72]-based controller runtime, the switch runtime, and the control channel interface, work cooperatively to guarantee the desired correctness and robustness properties of a fault-tolerant logically centralized SDN controller. More specifically, when the master controller crashes, the Ravana protocol detects the failure within a short failover time and elects the standby slave controller to take over using Zookeeper [73]-like failure detection and leader election mechanisms. The new leader finishes processing any logged events in order to catch up with the failed master controller state. Then, it registers with the affected switches in the role of the new master before proceeding with normal controller operations.

2) Hierarchical SDN control:

The hierarchical SDN control architecture assumes that the network control plane is vertically partitioned into multiple levels (layers) depending on the required services. According to [74], a hierarchical organization of the control plane can improve SDN scalability and performance. To improve scalability, Kandoo [31] assumes a hierarchical two-layer control structure that partitions control applications into local and global. Contrary to DevoFlow [28] and DIFANE [27], Kandoo proposes to reduce the overall stress on the control plane without the need to modify OpenFlow switches. Instead, it establishes a two-level hierarchical control plane, where frequent events occurring near the data path are handled by the bottom layer (local controllers with no interconnection running local applications) and non-local events requiring a network-wide view are handled by the top layer (a logically centralized root controller running non-local applications and managing local controllers).

Despite the obvious scalability advantages of such a control plane configuration where local controllers can scale linearly as they do not share information, Kandoo did not envision fault-tolerance and resiliency strategies to protect itself from potential failures and attacks in the data and control planes. Besides, from a developer perspective, Kandoo imposes some kandoo-specific conditions on the control applications developed on top of it, in such a way that makes them aware of its existence. On the other hand, Google’s B4 [75, 76], a private intradomain software-defined WAN connecting their data centers across the planet, proposes a two-level hierarchical control framework for improving scalability. At the lower layer, each data-center site is handled by an Onix-based [60] SDN controller hosting local site-level control applications. These site controllers are managed by a global SDN Gateway that collects network information from multiple sites through sitelevel TE services and sends them to a logically centralized TE server which also operates at the upper layer of the control hierarchy. Based on an abstract topology, the latter enforces high-level TE policies that are mainly aimed at optimizing bandwidth allocation between competing applications across the different data-center sites. That being said, the TE server programs these forwarding rules at the different sites through the same gateway API. These TE entries will be installed into higher-priority switch forwarding tables alongside the standard shortest-path forwarding tables. In this context, it is worth mentioning that the topology abstraction which consists in abstracting each site into a super-node with an aggregated super-trunk to a remote site is key to improving the scalability of the B4 network. Indeed, this abstraction hides the details and complexity from the logically centralized TE controller, thereby allowing it to run protocols at a coarse granularity based on a global controller view and, more importantly preventing it from becoming a serious performance bottleneck. Unlike Kandoo [31], B4 [75] deploys robust reliability and fault-tolerance mechanisms at both levels of the control hierarchy in order to enhance the B4 system availability. These mechanisms have been especially enhanced after experiencing a large-scale B4 outage. In particular, Paxos [77] is used for detecting and handling the primary controller failure within each data-center site by electing a new leader controller among a set of reachable standby instances. On the other hand, network failures at the upper layer are addressed by the logically centralized TE controller which adapts to failed or unresponsive site controllers in the bandwidth allocation process. Additionally, B4 is resilient against other failure scenarios where the upper-level TE controller encounters major problems in reaching the lower-level site controllers (e.g. TE operation/session failures). Moreover, B4 guards against the failure of the logically centralized TE controller by geographically replicating TE servers across multiple WAN sites (one master TE server and four secondary hot standbys). Finally, another fault recovery mechanism is used in case the TE controller service itself faces serious problems. That mechanism stops the TE service and enables the standard shortest-path routing mechanism as an independent service. In the same spirit, Espresso [78] is another interesting SDN contribution that represents the latest and more challenging pillar of Google’s SDN strategy. Building on the previous three layers of that strategy (the B4 WAN [75], the Andromeda NFV stack and the Jupiter data center interconnect), Espresso extends the SDN approach to the peering edge of Google’s network where it connects to other networks worldwide. Considered as a large-scale SDN deployment for the public Internet, Espresso, which has been in production for more than two years, routes over 22% of Google’s total traffic to the Internet. More specifically, the Espresso technology allows Google to dynamically choose from where to serve content for individual users based on real-time measurements of end-to-end network connections. To deliver unprecedented scale-out and efficiency, Espresso assumes a hierarchical control plane design split between Global controllers and Local controllers that perform different functions. Besides, Espresso’s software programmability design principle externalizes features into software thereby exploiting commodity servers for scale. Moreover, Espresso achieves higher availability (reliability) when compared to existing router-centric Internet protocols. Indeed, it supports a fail static system, where the local data plane keeps the last known good state to allow for control plane unavailability without impacting data plane and BGP peering operations. Finally, another important feature of Espresso is that it provides full interoperability with the rest of the Internet and the traditional heterogeneous peers.

### ONOS

A picture containing diagram

Description automatically generated

### ODL

### Ryu

## Software Switches

Table

Description automatically generated

## Virtual Emulated Environment

### Mininet

Mininet is a tool specially designed for software defined networks. It is an emulator of a network and it is used to visualize and test the OpenFlow switches and application of software-defined networks in a virtualized environment. Mininet provides the space to develop and test software-defined network applications without the need to set up a physical environment. It gives a network testbed thereby allowing to develop and test applications that are using OpenFlow protocols. It also gives the flexibility to integrate python API, thereby paving the way for creating and experimenting with networks. It also has a CLI which is aware of topology and OpenFlow thereby allowing to debug or run network tests for applications. Mininet is majorly used as a learning tool to test, experiment, and learn about software-defined networks and it is preferable because it is very fast and helps to create customizable topologies.

Advantages of Mininet :

* It is very fast and takes very little time for booting.
* It is easy to install and use.
* It saves money because the emulators are cost-effective instead of testing with hardware devices.
* It is also very easy to connect with other network devices.
* It has high availability.

### GNS3

GNS3 (Graphical Network Simulation 3) is an open source, free network software emulator , which emulates and interconnects networking devices, including routers, switches, firewalls, and other devices which are interconnectable within OSI model. GNS3 is used to configure, test and troubleshoot virtual and real networks. Latest version (Version 2.2.33.1) of GNS3 support devices from multiple network vendors such as Cisco virtual switches, Cisco SD-WAN components, Open vSwitch, Brocade vRouters, Cumulus Linux switches, Docker instances, Juniper vRR, HPE VSRs, Nokia vSIM, multiple Linux appliances and many others. Furthermore, GNS3 offers no limitation on the number of devices supported or complexity of network topologies. The only possible limitations are in the CPU and memory of the hardware that runs it.

GNS3 emulates the hardware of a device and runs real images in the virtual device, so it can be used to design complex networks and do simulations about them. Since it runs real images, it is necessary to have the images of the devices to be simulated. GNS3 offers on his official website different appliances that have and already configured image and can be used to emulate a device.

GNS3 consists of two software components:

* Client part: The GNS3-all-in-one software (GUI)
* Server part: The GNS3 virtual machine (GNS3 VM)

The GNS3-all-in-one is the graphical user interface (GUI) where the topologies can be created. When the topologies are created, the devices are hosted and run by a server process. The options for the server part are the following:

* Local GNS3 server: run on the same PC where the GUI is installed.
* Local GNS3 VM: run on the same PC using virtualization software such as VirtualBox or VMware.
* Remote GNS3 VM: run remotely using VMware ESXi or in the cloud.

For this Thesis VirtualBox was preferred to host the devices on the GNS3 VM.

Advantages of GNS3:

* Free and Open Source software
* No monthly or yearly license fees
* No limitation on number of devices (only limitation is the user hardware: CPU and memory)
* Supports multiple switching options (Open vSwitch, Cumulus Linux Switches, IOU/IOL Layer 2 images, VIRL IOSvL2):
* Supports all VIRL images (IOSv, IOSvL2, IOS-XRv, CSR1000v, NX-OSv, ASAv)
* Supports multi-vendor environments
* Can be run with or without hypervisors
* Supports both free and commercial hypervisors (VirtualBox, Hyper V, VMware workstation, ESXi, Fusion)
* Downloadable, free, pre-configured and optimized appliances available to simplify deployment
* Native support for Linux without the need for need for additional virtualization software
* Software from multiple vendors freely available
* Large and active community (800,000+ members)

Disadvantages:

* Cisco as well as other software images need to be supplied by user (download from Cisco.com, or purchase VIRL license, or copy from physical device).
* Not a self-contained package, but requires a local installation of software (GUI).
* GNS3 can be affected by PC’s setup and limitations because of local installation (firewall and security settings, company laptop policies etc.).

## OpenFlow Protocol

In more specific terms, and according to the original version 1.0.0 of the standard defined in [26], an OpenFlow-enabled Switch consists of a flow table and an OpenFlow secure channel to an external OpenFlow controller. Typically, the forwarding table maintains a list of flow entries; Each flow entry comprises match fields containing header values to match packets against, counters to update when packets match for flow statistics collection purposes, and a set of actions to apply to matching packets. Accordingly, all incoming packets processed by the switch are compared against the flow table where flow entries match packets based on a priority order specified by the controller. In case a matching entry is found, the flow counter is incremented and the actions associated with the specific flow entry are performed on the incoming packet belonging to that flow. According to the OpenFlow specification [26], these actions may include forwarding a packet out on a specific port, dropping the packet, removing or updating packet headers, etc. If no match is found in the flow table, then the unmatched packet is encapsulated and sent over the secure channel to the controller which decides on the way it should be processed. Among other possible actions, the controller may define a new flow for that packet by inserting new flow table entries. Despite the advantages linked to the flexibility and innovation brought to network management, OpenFlow [24] suffers from scalability and performance issues that stem mainly from pushing all network intelligence and control logic to the centralized OpenFlow controller, thus restricting the task of OpenFlow switches to a dumb execution of forwarding actions. To circumvent these limitations, several approaches [27, 28, 29, 30] suggest revisiting the delegation of control between the controller and switches and introducing new SDN switch Southbound interfaces. Notably, DevoFlow [28] claims to minimize switch-tocontroller interactions by introducing new control mechanisms inside switches. That way, switches can make local control decisions when handling frequent events, without involving controllers whose primary tasks will be limited to keeping centralized control over far fewer significant events that require network-wide visibility. Despite introducing innovative ideas, the DevoFlow alternative has been mainly criticized for imposing major modifications to switch designs [31]. On the other hand, stateful approaches [29, 32, 33], as opposed to the original stateless OpenFlow abstraction, motivate the need to delegate some stateful control functions back to switches in order to offload the SDN controller. These approaches face the challenging dilemma of programming stateful devices (evolving the data plane) while retaining the simplicity, generality and vendor-agnostic features offered by the OpenFlow abstraction. In particular, the OpenState proposal [29] is a stateful platform-independent data plane extension of the current OpenFlow match/action abstraction supporting a finite-state machine (FSM) programming model called Mealy Machine in addition to the flow programming model adopted by OpenFlow. That model is implemented inside the OpenFlow switches using additional state tables in order to reduce the reliance on remote controllers for applications involving local states like MAC learning operations and port-knocking on a firewall.

Despite having the advantage of building on the adaptation activity of the OpenFlow standard and leveraging its evolution using the (stateful) extensions provided by recent versions (version 1.3 and 1.4), OpenState faces important challenges regarding the implementation of a stateful extension for programming the forwarding behaviour inside switches while following an OpenFlow-like implementation approach. The feasibility of the hardware implementation of OpenState has been addressed in [34]. Finally, the same authors extended their work into a more general and expressive abstraction of OpenState called OPP [35] which supports a full extended finite-state machine (XFSM) model, thereby enabling a broader range of applications and complex stateful flow processing operations. In the same spirit, the approach presented in [36] explored delegating some parts of the controller functions involving packet generation tasks to OpenFlow switches in order to address both switch and controller scalability issues. The InSP API was introduced as a generic API that extends OpenFlow to allow for the programming of autonomous packet generation operations inside the switches such as ARP and ICMP handling. The proposed OpenFlow-like abstractions include an InSP Instruction for specifying the actions that the switch should apply to a packet being generated after a triggering event and a Packet Template Table (PTE) for storing the content of any packet generated by the switch. According to [36], the InSP function, like any particular offloading function, faces the challenging issue of finding the relevant positioning with respect to the broad design space for delegation of control to SDN switches. In their opinion, a good approach to conceiving (eventually standardizing) a particular offloading function should involve a programming abstraction that achieves a fair compromise between viability and flexibility, far from extreme solutions that simply turn on well-known legacy protocol functions (e.g. MAC learning) or push a piece of code inside the switches [37, 38]. The authors of FOCUS [39] express the same challenges but, unlike the above proposals, they reject a performance based design choice that requires adding new hardware primitives to OpenFlow switches in the development of the delegated control function. Instead, they promote a deployable software-based solution to be implemented in the switch’s software stack to achieve a balanced trade-off between the flexibility and cost of the control function delegation process.

# Requirements Analysis

Due to advances in the Information and Communication Technology, the configuration and management of the network components becomes highly complex and time-consuming. A fundamental characteristic of SDN is the logically centralized, but physically distributed controller component. SDN offers to batch-configure automatically multiple components in one step, while the traditional way would mean to log into each device. Many operators struggle with the migration from IPv4 to IPv6, SDN with its centralized control and the possibility to reduce human error due to increased poses an opportunity to help make this migration easier. The controller maintains a global network view of the underlying forwarding infrastructure and programs the forwarding entries based on the policies defined by network services running on top of it. The traditional networking approach has very limited facilities to explore these aspects of networking and the goal would be to study these futuristic characteristics of networking.

## General Objectives

The detailed requirements for this thesis and the involved tasks are stated below:

* Build a suitable network with different network devices in the emulation software.
* Manage different services and network configurations with SDN controller in an emulated environment.
* Create and distribute the network configurations for network devices.
* Develop a rationale and setup an IPv4 and IPv6 scheme for the network.
* Provide services and user groups that have different requirements.
* Evaluate advantages and disadvantages of network with SDN controller over traditional network.
* Proof and validation of functioning failover mechanisms to improve resilience.

Throughout this thesis, the following research questions are answered:

* Research possible open-source SDN controllers to implement.
* Research alternative configuration methods with the goal of finding the best possible method to configure and manage the network through Network Controller.
* How to provide different paths in the network with different QoS properties?
* Algorithms that are responsible for the optimization of the paths.
* When a service is accessible at multiple times, how to choose the best one.

## Work Plan

Work flow for this Thesis is as follows;

* Research about the SDN controllers.
* Research about the emulation software.
* Installation and testing of different SDN controllers.
* Configuration of these SDN controllers.
* Installing and activating the features required for the services and applications running on the SDN controller.
* Installing different network devices preferably open-source devices to run inside the emulated environment.
* Setting up the connection between these networking devices and the SDN controller.
* Document the difficulties and errors found while installation and debugging the connectivity between the different network devices and SDN controller
* Creating and implementing different network topologies.
* Creating use cases suitable for SDN controller in the emulated environment.
* Implementing and testing these use cases in the emulated environment.

## Previous Work

Through the study and research about implementation of SDN controllers in virtual environments, the following points were found;

* Currently various Open-source SDN controllers are available to be implemented and tested in the environment. To name few, OpenDayLight (ODL), Open Network Operating System (ONOS), Ryu and Faucet.
* ONOS and ODL are built to have centralized architectures. Hence, they tend to be easier to maintain and confer lower latency between the tightly coupled southbound APIs and northbound APIs.
* Faucet is built to have distributed architectures which generally are more complex to maintain and deploy but can allow the platform to scale more effectively. By decoupling the processing of PCE, Telemetry and Southbound interface traffic, each function can be scaled independently to avoid performance bottlenecks.
* Whereas, Ryu is different to the other options, although having a core set of programs that are run as a platform, it is better thought of as a toolbox, with which SDN controller functionality can be built.
* ONOS and ODL are written in Java, for which development resources are abundant in the market, with good supporting documentation and libraries available.
* Ryu and Faucet are written in Python, a well-supported language and has an active community developing the framework. The documentation is concise and technical, aimed at developers to maximize the utility of the system.
* Both ODL and ONOS benefit from large developer and user communities under the Linux Foundation Networking banner. Many large international players are involved in the development and governance of these projects, which could add to the longevity and security over time.
* Ryu and Faucet are well supported, targeted controllers. Due to the emerging nature of the field, both options look to have a bright future, with a simpler, streamlined approach to change submission and testing.
* Most of the software defined networking testbeds are implemented and analysed with the Mininet environment.
* Mininet being network emulator is used to deploy quick network topologies with Open vSwitch supporting SDN protocols along with in-built SDN controller. These are ran as virtual instances inside the Mininet environment.
* In GNS3, real images of the network devices are used which are almost identical to the real world network devices. Quite few experimental research was found where SDN controller was tested with network topologies created using the network devices in GNS3 environment.
* New versions of OpenDayLight controller do not support the GUI (DLUX application features) and L2 switch applications. Study reveals the projects associated with these applications were separated from OpenDayLight.

# Realization

After researching about various SDN controllers, few suitable SDN controllers were selected for building a virtual environment on the emulation software. These SDN controllers were Open Network Operating System (ONOS), OpenDayLight (ODL) and Ryu. Similarly, to create the network with network elements, two emulation software are selected, namely, Mininet and GNS3.

## Installation

For setting up the testbed for this Thesis work, all the major components were installed on the Oracle VM VirtualBox virtualization application. Different virtual machines were created for each SDN controller.

Mininet-VM and GNS3 VM were also running on different instances of Virtual Machines. Mininet-VM is capable of creating topologies with Open vSwitches, Hosts and SDN Controller all in itself. When topologies are created in GNS3 using the all-in-one software GUI client, the devices created are need to be hosted and run by a server process, for this GNS3 VM instance is used. All SDN controllers have their complete installation in respective virtual machines.

Graphical user interface, application

Description automatically generated

*Figure 4.1: Different virtual machines instances in VirtualBox*

Topologies created inside the GNS3 and Mininet are able to connect to the outside SDN controller using OpenFlow protocol. SDN controllers are running as a service, meaning after installation the controllers VMs were configured to directly run them as SDN controller without need of any separate command to start the controller.

## Implementation with GNS3

After complete installation and necessary configuration of the components required for this Thesis work, a topology with four Open vSwitches and routers was created in the GNS3 application. For testing purpose of the routers are deployed at the far ends of the topology.

In GNS3, to connect any network device to the SDN controller a NAT interface is used. A L2 switch (Switch1 in Figure 4.2) is used so that all devices can be directly connected to the NAT interface with just single interface. SDN controller used here is ONOS controller.

Chart

Description automatically generated

*Figure 4.2: Topology created in the GNS3 with different Network devices*

Before connecting the Open vSwitches to the SDN controller, OpenFlow protocol version (her OpenFlow version 1.3) needs to be specified.

Bridge br0 is the management interface on the Open vSwitch which accepts the configuration commands from the SDN controller. To connect this interface to the SDN controller, external IP address of the SDN controller (here, 192.168.0.113) and port number of OpenFlow protocol needs to be specified (here, 6653).

Text

Description automatically generated

*Figure 4.3: Commands and confirmation of Open vSwitch connection to the ONOS controller*

As seen in Figure 4.3, interface Br0 was connected to the controller at the given IP address and the Boolean value was set to **True**. To confirm the connection from the controller side, ONOS GUI can be ran on the browser. The *onos-gui* feature must be installed in ONOS. The GUI listens on port 8181. The base URL is /onos/ui; for example, to access the GUI on localhost, use: <http://localhost:8181/onos/ui>

Graphical user interface

Description automatically generated

*Figure 4.4: Topology view on the ONOS controller GUI*

The ONOS GUI is a single-page web-application, providing a visual interface to the ONOS controller. ONOS GUI provides a great and easy to understand information about the ONOS controller and the topology connected to with it. Information such as applications installed on controller, number of devices connected, number of hosts, port numbers used in the topology, number of packets transferred between the links, and many more information is easily accessible through ONOS GUI. The ONOS Cluster Node Panel indicates the cluster members (controller instances) in the cluster. The Summary Panel gives a brief summary of properties of the network topology. The Topology Toolbar provides push-button / toggle-button actions that interact with the topology view.

A Wireshark was ran between the ONOS controller and one of the Open vSwitch to release the packets transferred between them. Southbound protocol OpenFlow was also captured in this link and figure 4.5 shows the Wireshark capture of the same.

Table

Description automatically generated

*Figure 4.5: Topology view on the ONOS controller GUI*

In figure 4.6 we can observe the OpenFlow protocol version 1.3 being used for communication between the ONOS controller and Open vSwitches.

Graphical user interface, text, application

Description automatically generated

*Figure 4.6: Topology view on the ONOS controller GUI*

## Implementation with Mininet

## Problems identified

During the installation of the components used in this Thesis few challenges were faced. Similarly while implementing and testing the connectivity between the SDN controller and the network devices few problems were identified. Following are the problems encountered and possible fixes for the same;

* Installation of SDN controllers is easier when they are first downloaded as a packet rather than directly installing from source code.
* Open vSwitch application available for GNS3 on GNS3 Marketplace (named, Open vSwitch with management interface) has some errors and needs to be fixed before using.
* New versions of OpenDayLight controller do not support the GUI and L2 switch applications since these projects were separated from OpenDayLight.
* Unlike previous versions of OpenFlow protocol, latest version of this protocol needs to be specified on all the networking devices.
* Before connecting the Open vSwitch to the controller, Open vSwitch needs to be specified the version of OpenFlow protocol to be used.
* Also, need to specify the latest version of OpenFlow protocol while executing the topology commands on Mininet.

## Use Cases

## Use case-1: Basic Network Architecture

### Introduction

A simple topology with four Open vSwitches and routers was created in the GNS3 application. For testing purpose of the routers are deployed at the far ends of the topology.

To connect any network device running in the GNS3 software with the device outside the GNS3 software, i.e. in local machine or in Internet, a NAT interface is used. In this scenario, a SDN controller is installed on a virtual machine outside the GNS3 application and hence a NAT interface is used to connect the Open vSwitches with the SDN controller. A L2 Switch or Hub can be used so that all devices can be directly connected to the NAT interface with just single interface. SDN controller used here is ONOS controller.

Before connecting the Open vSwitches to the SDN controller, OpenFlow protocol version (here OpenFlow version 1.4) needs to be specified.

Bridge br0 is the management interface on the Open vSwitch which accepts the configuration commands from the SDN controller. To connect this interface to the SDN controller, external IP address of the SDN controller (here, 192.168.0.114) and port number of OpenFlow protocol needs to be specified (here, 6653).

Graphical user interface, chart

Description automatically generated

*Topology created in the GNS3 with different Network devices*

Graphical user interface, website

Description automatically generated

*Topology view on the ONOS controller GUI*

In this sub-chapter further explanation about Devices, Hosts, Ports of devices, Links, Packet processors and Applications required by ONOS to process the packet will be discussed.

### Flows

In this sub-chapter following points will be discussed,

* Creating Flows for traffic
* Different Flow configuration methods
  + Through configuration API
  + Open vSwitch CLI
  + ONOS CLI

### Intents

The Intent Framework is a subsystem that allows applications to specify their network control desires in form of policy rather than mechanism and these policy-based directives are *Intents*.

In this sub-chapter following points will be discussed,

* Creating Intents for specific traffic between devices
* Different Intent configuration methods
  + Through configuration API
  + ONOS CLI
  + Through ONOS GUI

## Use case-2: Testing the network with multiple Controllers

### Introduction

In this Use case multiple SDN controllers were deployed to control the devices in the same network. A Spine-Leaf topology was created with three controllers, six devices and multiple hosts. In the following figure, three controllers can be observed in three different colours (navy blue, blue & red) having three different IP addresses (172.17.0.5, 172.17.0.6 & 172.17.0.7). Cluster of these three SDN controllers was created to act together as a unified and coherent distributed system.

The topology observed in the following figure was created in the Mininet environment and the SDN controllers were deployed as the Docker containers.

A screenshot of a computer

Description automatically generated with medium confidence

*Topology view on the ONOS controller GUI*

Graphical user interface, application

Description automatically generated

*Cluster Nodes with three ONOS controllers*

### Different Controllers with their Devices

In this sub-chapter following points will be discussed,

* Distribution of devices amongst the Controllers
* Selection of Master Controller amongst the Controllers
* Failure of one Controller

Graphical user interface

Description automatically generated with medium confidence

*Failure of one controller*

### Path Identifier

In this sub-chapter following points will be discussed,

* Links between the devices/hosts
* Path identification for traffic between the devices/hosts
* Proof and validation of functioning failover mechanisms of link between the devices

## Use case-3: Testing network with isolated L2 overlay networks

### Introduction

In this Use case virtually separate paths were created to isolate the traffic flow between the end points. For this Virtual Private LAN Service (VPLS) was implemented in this Use case. The goal was to connect multiple end-points in an OpenFlow network, creating isolated L2 broadcast overlay networks. While legacy technologies require the manual configuration of multiple devices in the network, VPLS tries to make the process easier for network operators. Hosts that get connected together can send in either untagged or VLAN tagged traffic, using either the same or different VLAN IDs. Two different VPLS were created, first, blue VPLS connecting Host H1 with Server 1 and second, red VPLS connecting Host H2 with Server 2 as observed in following figure.

Chart, radar chart

Description automatically generated

*Topology created in the GNS3 with different Network devices*

### Configuration

In this sub-chapter the following points will be discussed,

* Configuration of VPLS on SDN controller
* Working of VPLS
* Studying VPLS application of ONOS controller

## Use case-4: Testing network with IPv6 addressing

### Introduction

For testing the SDN controller functionality with the IPv6 addressing this Use case was implemented. In this Use case, the network was created consisting of IPv6 addresses and IPv4 addresses. The network was simultaneously tested for both the addresses. In the below figure two IPv6 Hosts are represented as H1& H2 and two IPv4 Hosts are represented as H1-4 & H2-4.

Diagram

Description automatically generated with medium confidence

*Topology created in the GNS3 with different Network devices*

### IPv6 tunnelling over IPv4

In this sub-chapter following points will be discussed,

* Creating three networks; two IPv6 networks and one IPv4 network
* Configuration of all these networks on SDN controller
* Creating IPv6 tunnel through IPv4 network
* Evaluating how packets are forwarded through OpenFlow protocol

Chart

Description automatically generated with medium confidence

*Topology created in the GNS3 with different Network devices*

## Use case-5: Integrating SDN with legacy network

### Introduction

In this Use case SDN network was integrated into the legacy networks. Four different legacy networks were created with Border routers R1, R2, R3 & R4 in different Autonomous Systems (AS) as shown in the following figure. These Border routers were configured with BGP protocol. The SDN network was implemented in different AS than these Border routers. The goal was to integrate the SDN network into these legacy networks and study the functionality of SDN controller with the BGP protocol.

Diagram

Description automatically generated

*Topology created in the GNS3 with different Network devices*

Chart

Description automatically generated with low confidence

*Topology view on the ONOS controller GUI*

### Configuration

In this sub-chapter following points will be discussed,

* Creating and configuring all these networks
* Configuration of all these networks on SDN controller

Studying SDN-IP application of ONOS controller

# Summary and Perspectives

# Abbreviations

**0**

…

**3**

3GPP Third Generation Partnership Project

…

**A**

AS Autonomous System

…

**B**

**C**

**D**

DLUX

**G**

GNS3

**H**

**L**

**N**

NBI Northbound Interface

NetConf Network Configuration Protocol

**O**

ONF Open Networking Foundation

ODL OpenDayLight

ONOS Open Network Operating System

**S**

SBI Southbound Interface

SDN-IP

SNMP Simple Network Management Protocol

…

**V**

VPLS Virtual Private LAN Service

**Z**

…

# References

|  |  |
| --- | --- |
| [1] | B. a. L. R. Team, „Distributed SDN Control: Survey, Taxonomy and Challenges,“ *IEEE Communications Surveys & Tutorials,* p. 25, December 2017. |

# Appendix